PARALLEL ALGEBRAIC MULTIGRID FOR THE PRESSURE POISSON EQUATION IN A FINITE ELEMENT NAVIER-STOKES SOLVER

By

Chun Sun

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

DOCTOR OF PHILOSOPHY

Major Subject: MECHANICAL ENGINEERING

Approved by the
Examining Committee:

_____
Kenneth Jansen, Thesis Adviser

_____
Jacob Fish, Member

_____
Eldar Giladi, Member

_____
Mark Shephard, Member

Rensselaer Polytechnic Institute
Troy, New York

April 2008
(For Graduation May 2008)

CONTENTS

LIST OF TABLES

LIST OF FIGURES

ACKNOWLEDGMENT

First and foremost, I want to thank Prof. Kenneth Jansen, for being my advisor in the last four years. I have learned a lot from him, academically and beyond. His insightful advice, patient guidance, constant support and encouragement are essential to my completion. His fine intuition and deep understanding of linear algebra, finite element, and fluid dynamics have been very important to this thesis. He also gave me so much trust during the research, and created a harmonic and free working environment for me. He has constantly helped me on development and documentation with great effort, often in tremendous small detail that beyond my imagination. His support for my work, life, and career development are invaluable, and my words simply can not express enough about that. His work ethic has been an enormous influence on me. He guides each of his students, including me, with his full heart and consideration, while arranging a lot of other responsibilities in a nice parallel way. He has been, and always will be, a high standard and eternal guidance of me. For me, he has redefined the word "advisor" to a new sky-high level. It has been my extreme privilege to know him and work with him.

I want to thank Prof. Jacob Fish, for being my committee member, and giving me the chance of collaboration in a lot of areas. He generously helped me with the understanding of his innovative Generalized Global Basis method, and further implementation in my project. Also as an expert in the area of Algebraic Multigrid, he gave me so many useful resources, information and guidelines. He also helped me to start multigrid research in his multi-physics class. During my teaching assistant duty with his finite element class, his assignments have helped me building my fundamentals for my future research. His advice and help in my research, teaching, and career decision have been irreplaceable important to me.

I want to thank Prof. Eldar Giladi, for being my committee member, and providing me with a mathematics perspective for this thesis. His comments and advice has been most essential for me to understand the new methods in linear algebra especially in the polynomial smoother. Also I want to express my gratitude

for his remote support, and traveling from Boston for my defense.

I want to thank Prof. Mark Shephard, for a lot of things. I want to thank him for being my committee member and giving me advice on my project. I want to thank him for his guidance that helped greatly to build my documentation skills and attitude as a professional researcher in my early days here. His classes, fundamentals of finite element and finite element programming, have polished my understanding of FEM and software development. I want to thank him mostly for leading the SCOREC and CCNI, which provided computational resources that are incomparable to most of the other places. I was very lucky to run applications on the IBM BlueGene/L that ranked the first in all the universities around the world. Let alone the countless "smaller" clusters that are available. I have been using memory and CPU that most students in other schools can not even imagine. Moreover, SCOREC and CCNI are such wonderful places that have been attracting various top researchers, thus I has been benefited by just talking to them. My thesis would not be possible without Prof. Shephard leading these two wonderful centers, SCOREC and CCNI.

I want to thank Lockheed Martin Company, Knolls Atomic Power Lab (KAPL) for financially supporting me during the years for this parallel AMG project. I want to thank Dr. Ted Bagwell from KAPL for his continuous collaboration and support.

I want to thank ACUSIM Company, for providing software library for this project. Personally I want to thank the CEO of ACUSIM, Dr. Farzin Shakib, for the generous, countless insightful discussions, innovative ideas and inspiring suggestions. Also for sharing part of his source code in the early stage of research. His code does not only serve as a side-benchmark for my project, but also sets a high standard of industry-level, efficiency-oriented coding for me. His deep understanding of math, computer science, and engineering reflected between the source code lines, has been, and always will be, my goal and model.

I want to thank Dr. Haim Waisman, for his generous help of Generalized Global Basis method. I want to thank Shaun Simmons for his early investigation of Algebraic Multigrid. I want to thank my colleagues and friends, Dr. Onkar Sahni and Min Zhou, for their warm-hearted help and useful discussions over a lot of areas

including parallel scaling, pre-post processing, meshing etc. They are essential to this project.

I want to thank Dr. Klaus Stüben from Fraunhofer SCAI, for his support at the early development of AMG and sharing license of commercial SAMG software. I want to thank people from Trilinos project in Sandia national lab, for providing an open-source multi-level solver as one of my benchmark. I want to thank people in ARPACK/PARPACK project from Rice University, for providing an open-source eigenvalue package.

I want to thank computer administrators Christophe Dupre and Adam Todorski, for keeping clusters, workstations, supercomputers all up and running well, and sharing knowledge of Linux cluster operations with me. I want to thank secretary Marge Verville and Darwisah Burgess, for taking care of everything other than the academics.

Finally, I want to thank my family especially my wife Li Chen, for her unconditional trust, constant encouragement, and sweet love over the years.

ABSTRACT

The focus of this thesis is on a parallel Algebraic Multigrid (AMG) algorithm and its application within the parallel Navier-Stokes finite element (FEM) solver, *PHASTA*. Specifically we focus on solving the Pressure Poisson Equation (PPE) arising from the finite element discretization of the incompressible Navier-Stokes equations using parallel AMG as the preconditioner to a Conjugate Gradient (CG) solver.

First a review of recent serial and parallel AMG algorithms is provided to set the stage for a detailed description of our state-of-the-art multilevel serial AMG implementation. Classical Ruge-Stüben AMG is set as the base algorithm. It is then altered in several aspects to better fit for our flow problems. Also, recent smoother advancements like polynomial smoothing are tested and implemented with different options. A recently developed external accelerator for multigrid, the General Global Basis (GGB) method, is studied and implemented. In the serial AMG study, a dramatic reduction (usually above 90% in our test problems) in the number of iteration vectors is observed.

The parallelization of the PPE solver is then studied in great detail. The difficulties for AMG are carefully studied. These difficulties include the matrix-matrix product form of PPE from multiplicand matrices that are locally incomplete, the special parallel data structures designed for a simple Krylov solver, and the mesh-based partitioning scheme. These difficulties make it impractical to construct a parallel AMG cycle that is identical to the serial case. A new AMG algorithm for parallelization that works close to the serial one but is easy to parallelize, is then proposed. This new parallel AMG algorithm separates the smoothing process for different levels of AMG, and makes use of the PPE construction at the finest level to apply a complete smoothing operation only at that level. The implementation of this new algorithm is carefully matched within the framework of the current parallel FEM software. A serial validation of the algorithm is also carried out.

The algorithms are also contrasted to assess efficiency. Techniques like freezing setups for multiple solves are studied to increase the efficiency for long time runs

in the numerical examples. For several examples tested, significant reduction in the number of iteration vectors does not degrade as the number of processors is increased. The scaling of the algorithm is tested on IBM Bluegene supercomputer, and the excellent computational time scalability of the original FEM plus CG solve is maintained with each algorithm improvement up to 16k processors.

CHAPTER 1
INTRODUCTION

In this thesis we want to explore the design and application of parallel Algebraic Multigrid (AMG) as a preconditioner to accelerate the solution of linear systems. In this chapter, we will discuss the motivation for this effort and the various options available in the current literature before giving a short introduction to the AMG method. Finally the testbed on which our research is based will be introduced.

1.1 Motivation

In engineering applications of computational mechanics there often is a need for solving large, sparse systems of linear algebraic equations. For example, these equations may arise from the discretization by various methods of partial differential equations (PDE). Non-explicit discretizations like Finite Difference (FD), Finite Volume Method (FVM), and Finite Element Method (FEM) are typical and popular methods that are widely used in different applications. For today's problems, each of these method can easily result in linear systems of algebraic equations with billions of degrees of freedom. These linear systems often require a substantial amount of computational time to solve. Solvers which try to accelerate this process need to be developed.

Also with the advances in technology of parallel computers, supercomputers that have thousands of processors, are becoming more available in the recent years. Thus arise the need for the research of techniques of solving large linear system in a massive parallel context. While many of the complex solver algorithms were developed from the serial era when parallelization was not an option, the new supercomputers are restricted with limited simple solver algorithms. So there are demands in new algorithms that can both accelerate the solution process, and be capable of extension to massive parallel processors.

In the field of computational fluid dynamics (CFD), there are many systems of equations that need solutions to large systems. For example, the governing in-

compressible Navier-Stokes equations are highly non-linear, and usually a Poisson like pressure equation needs to be solved as well. These linear solves often become the dominant use of computational resources in the application of CFD. Sometimes when the condition goes stiff, current solvers can not converge within a reasonable time limitation. So a stronger, faster, more robust solver to these expensive systems is becoming an urgent need.

Given these motivations above, a substantial amount of research effort has been focused on the development of a variety of parallel solvers on different problems. Despite this effort, for many problems of interest especially in CFD, it still remains an expensive task to solve large, sparse linear system $\mathbf{Ax} = \mathbf{b}$.

1.2 Linear Equations Solution Options

While a direct solve like complete LU decomposition or Gauss elimination [1] gives accurate results, it is often impractical to apply for large problems, especially when working with parallel computers. Furthermore, since these systems of equations represent discretizations of approximate nature, an exact solution of the algebraic system is not necessarily required. Iterative methods were developed to give approximate solutions with potentially less computational effort. There exists a variety of iterative methods. The most basic are the classical smoothers like damped Jacobi, and Gauss-Seidel [1] which were developed to smooth the error components in the solution vector by iterations. Moving up in complexity, there exists a family known as Krylov methods which were developed to minimize a residual norm. These methods approximate the solution by a set of vectors created from matrix-vector products of the right hand side and the original matrix only. Krylov methods can be preconditioned by replacing one or more vectors in this series. Krylov methods include Conjugate Gradient (CG) [1] for symmetric positive definite(s.p.d.) systems and Generalized Minimal RESidual (GMRES) [2] for asymmetric systems.

It has been observed that when employing classical iteration methods, low-frequency error components are reduced much more slowly than high-frequency error components [3, 1, 4]. The multigrid principle was motivated by this observation. In a nutshell, multigrid is a process to bring a system of equations that is large (with

significant low-frequency error), to a reduced system which is coarser/smaller. If this process is performed properly, the low-frequency error components in original fine system become high-frequency error components in new coarse system. Thus, the error is better reduced by classical iterations applied at the coarser level. Multigrid methods project residual vectors from the fine level to the coarse level and project solution vectors from the coarse level level to the fine level [4, 5]. This process is called "coarse grid correction". It is used to modify the solution vectors on the fine grid thereby accelerating the whole solution process[4, 5, 6].

A natural application of this idea is Geometric Multigrid (GMG) [7, 8], which is often used on structured (or at least nested) meshes. An alternative is AMG, which only takes the left hand side matrix of the problem $\mathbf{A}$ as input, and creates levels of coarser systems to accelerate the solution process. While both GMG and AMG have provided substantial acceleration when compared to basic iterative solvers like Jacobi or GS (what is typically referred to as a "Stand-Alone AMG or GMG"), even better performance has been observed when these methods are used as a preconditioner of Krylov methods [4, 9, 10].

While there has been a substantial amount of research in general AMG solvers, e.g. a library that can solve $\mathbf{Ax} = \mathbf{b}$ in a black-box style [11, 12], we are more interested in developing a preconditioner that can be integrated with our current linear solver and parallel scheme. We expect it to be fully customizable, robust and able to be tuned to our special needs. For example, we would like to control the frequency of setup phase of AMG and match it to our existing solution strategy. It also must be compatible with our current parallel partitioning both in data parallel and control parallel which enables us to use the same communication patterns and data structures to save memory and CPU resources. The need for a customizable implementation stems from subtle but important differences in the systems that we intend to study under this research. Black box solvers like SAMG [11] were designed for M-Matrices which our method does not guarantee. Furthermore, the data parallel scheme of black box solvers[11, 13] do not fit into our data distribution scheme (e.g., it may require overlapping boundary nodes and/or control of the partitioning by the linear solver).

Our systems arise from discretizations on irregular meshes, often with very high aspect ratios. They may or may not make use of hierarchical basis functions resulting in a more complex linear system structure. The classical iterative methods stated above are not efficient for these problems. Sometimes, the systems are so poorly conditioned that the solver will not even converge by classical iterative methods. Therefore, in this work, we investigate the use of AMG as a preconditioner to Krylov methods. Our goals are: first, reduce the computational effort required in the solution of our linear equations systems; second, extend the algorithm to massive parallel in the scale of thousands processors while maintaining scalability.

1.3 Review of AMG

1.3.1 Serial

The AMG method was popularized by Ruge and Stüben, *et al.* [3]. It has grown even more popular in the past several years and is still under development. Basically, AMG involves three key steps including:

1. selecting coarse nodes, (Prepare to build a coarse system)
2. building interpolation operators, (Operators used to build coarse system and mapping between coarse and fine systems)
3. smoothing at each level. (Use coarse system to smooth the fine-grid low-frequency errors)

The various forms of AMG that have been proposed involve various combinations of these key steps. In what follows we give a summary of several popular choices.

In Ruge and Stüben's basic AMG, coarsening is based on the selection of strong correlated unknowns[3, 4, 10, 5]. The strong or weak coupling between unknowns are decided based on the coefficients in the matrix. Each fine node is chosen such that it will be strongly connected to at least one coarse node. The interpolation operator is an approximation to the fine node value based on all the coarse nodes it is strongly connected. The choice of the AMG smoother varies. Usually classical

smoothers like Gauss-Seidel are chosen, as in Stüben's classical AMG [4] and later implemented in its commercial library [11]. There are also alternatives like: ILU (Incomplete LU decomposition) as the smoother such as [14] proposed by Saad *et al.* (also implemented in [11]), sparse inverse proposed by Bröker *et al.* [15], and polynomial smoothing based on a matrix-vector product proposed by Adams *et al.* [16].

A second class of AMG is aggregation-type AMG [17, 18, 19, 20, 21, 22, 6, 23]. Smooth aggregation (SA) is the typical approach of this kind of AMG. In the original SA, coarsening is defined by building aggregates; fine nodes are singularly associated with coarse nodes; and the interpolation operators are thus piecewise constants [17, 5, 21]. In this approach each "coarse node" represents a group of nodes that should be strongly connected with it and get interpolation from it instead of from many coarse nodes as in classical AMG. This involves a careful choice of aggregation. Different approaches like AMGe[17, 22, 18], which is AMG based on element interpolation, reconstructs the topological structure from the fine system by analyzing the coefficients of the original matrix, and uses the topological information for further aggregation process. Some other types of aggregation AMG include: 1) double-pair AMG, which constructs a coarse matrix by double pairwise aggregation[20], 2) and adaptive SA (αSA) whose coarsening process are adaptive to the smoothing process of a prototype vector[17, 21]. More detailed reviews on this topic can be found at [5, 21].

Though the concept of AMG is straightforward, there arise systems that are slow to converge. The slow convergence can be associated with some eigenvalues of the AMG amplification matrix that are greater than or very close to 1 in magnitude. Recently, Waisman and Fish [9, 24, 25] created a technique known as the Generalized Global Basis (GGB) wherein a finite number of eigenvalues (and associated eigenvectors) of the AMG amplification matrix are computed. Then the eigenvectors are used to restrict the full system to a small/coarse system where a direct solver can efficiently eliminate them. Finally, the transpose of the eigenvectors provides the "coarse grid correction" to the original/fine system. While this method is more like a filter than an alternative AMG method (indeed its kernel is based upon the

user selection of a multigrid method), it can provide substantial further acceleration to stand-alone AMG or AMG used in conjunction with a Krylov solver.

1.3.2 Parallel

Considering our final goal of parallelization on AMG, it is important to understand how different versions of AMG handle the parallel issue. Although most are still in the development stage, several attempts have been made workable. As expected, none of these approaches are "perfect". The convergence performance remains inferior to serial AMG. We will discuss this degradation in detail along with other issues arising from parallelization in a later section. Here we briefly review existing parallel approaches by AMG components.

The difficulty of parallelizing the classical smoother, especially Gauss-Seidel, was re-iterated by several different researchers [15, 16, 26]. Although somewhat easier, block Jacobi, faces substantial difficulties. These challenges have motivated researchers to consider alternatives. For example, polynomial smoothing was proposed by Adams *et al.* [16, 27]. In polynomial smoothing, only matrix vector products are needed to execute the smoother to minimize the spectrum of resulting error iteration matrix. It has similar convergence to Gauss-Seidel but is much easier to make parallel since it is based on matrix-vector products. Sparse approximate inverse is another alternative. It construct the smoother $\mathbf{M}$ by minimizing $\mathbf{I} - \mathbf{MA}$, so it is also inherently parallel. However, the reported convergence has some noticeable degradation relative to Gauss-Seidel (a factor of about 2) [15].

Parallelization of the coarsening process has also been discussed by various researchers [28, 29, 30, 31, 32, 13]. The coarsening setup based on aggregation-type AMG has several options as reviewed by Hensen and Yang in [32]. Most of these algorithms require special treatments of the boundary nodes, which involves a sequential work and communication, thus impairing parallel efficiency. Examples are works described by Tuminaro *et al.* [31] and also by Anwander *et al.* [30]. Another popular approach is referred to as the maximally independent sets (MIS) method, which can group nodes in parallel [31, 13, 32] but still requires a boundary treatment. Although by communication it is possible to reproduce the result in serial AMG,

it is usually requires prohibitively large and irregular communication, re-numbering and changing in data structures resulting higher complexity numbers (as shown in [32, 22, 31, 13, 30]).

The parallel scheme proposed by Stüben as described in [28] provides an alternative approach. The coarsening process is performed by first choosing the partition boundary nodes as "coarse". After carrying these boundary nodes all the way to the coarsest level, a serial AMG is performed among boundary nodes themselves. It is claimed that by doing this and other extra steps the coarsening process can be preserved and not damaged by parallelization. It is also claimed that chaotic blocked Gauss-Seidel [1], as an inevitable result from parallelization, will have less effect on the overall convergence.

More detail on the parallelization issues will be presented in Chapter 5.2, Chapter 5.3 and Chapter 5.4 when we discuss parallel issues in more detail.

1.4 AMG Notation

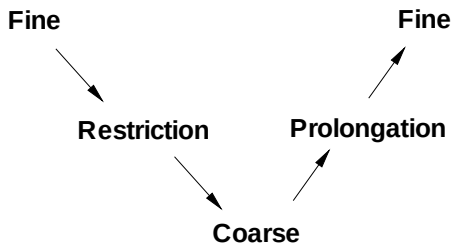

Figure 1.1: Multigrid Flowchart

Figure 1.1 shows a basic two-level multigrid cycle. Starting from the fine mesh, AMG applies a restriction operator to obtain the problem's representation on a coarse mesh. After the (approximate) solution is obtained on the coarse mesh, AMG prolongates the correction to the fine level using the interpolation operator. Following Stüben's notation, upper case H is the coarse level and lower case h is

[1]The term "chaotic" was introduced by Tuminaro *et al.* [31] to represent each processor performing a Gauss-Seidel without any special treatment of the boundary which will of course not be a correct Gauss-Seidel.

the fine level. The restriction operator is defined as Q_h^H and interpolation as Q_H^h. Additionally, the restriction is defined as the transpose of interpolation [33, 34, 4]:

$$Q_h^H = \left(Q_H^h\right)^T \tag{1.1}$$

Given the above operators, the system on the coarse level can be obtained by the relation:

$$\underbrace{A_H}_{Coarse} = \underbrace{Q_h^H}_{Restriction} \underbrace{A_h}_{Fine} \underbrace{Q_H^h}_{Interpolation} \tag{1.2}$$

The above relation is known as the Galerkin condition.

Note the following additional relations:

$$\underbrace{A_H}_{(C\times C)} = \underbrace{Q_h^H}_{(C\times F)} \underbrace{A_h}_{(F\times F)} \underbrace{Q_H^h}_{(F\times C)} \tag{1.3}$$

where F is the number of fine nodes and C is the number of coarse nodes.

1.5 Testbed

As noted earlier, our goal is to specialize the AMG method to our flow solver. To understand what that entails, we give a brief introduction to our flow solver.

The unstructured-grid flow solver *PHASTA* has been formulated using a stabilized finite element method in space to spatially discretize the incompressible Navier-Stokes equations [35, 36, 37, 38, 39]. The resulting set of non-linear ordinary differential equations are integrated in time with the generalized-α method. The resulting non-linear equations undergo a Newton linearization [36] which leads us to a system of Algebraic Matrix Equations:

$$\begin{pmatrix} \mathbf{K} & \mathbf{G} \\ -\mathbf{G^T} & \mathbf{C} \end{pmatrix} \begin{pmatrix} \Delta\dot{\mathbf{u}} \\ \Delta\dot{\mathbf{p}} \end{pmatrix} = - \begin{pmatrix} \mathbf{R_m} \\ \mathbf{R_c} \end{pmatrix} \tag{1.4}$$

While it is possible to solve (1.4) monolithically, exceptionally tight tolerance is required to obtain satisfactory accuracy since there is a dramatic spread between the eigenvalues associated with the continuity equation relative to the those associated

with the momentum equations. This tight tolerance is impractical for problems of interest which has motivated the consideration of an alternative approach wherein $\Delta\dot{p}$ is solved approximately first to precondition (1.4). The equation for $\Delta\dot{p}$ is obtained by static condensation of (1.4) which leads us to the discrete pressure Poisson equation (PPE):

$$\left[\mathbf{G}^T\hat{\mathbf{K}}^{-1}\mathbf{G} + \mathbf{C}\right]\Delta\dot{p} = \left[-\mathbf{G}^T\hat{\mathbf{K}}^{-1}\mathbf{R}_m - \mathbf{R}_c\right], \tag{1.5}$$

where $\hat{\mathbf{K}}^{-1}$ is an approximation of $\mathbf{K}^{-1}$ found by taking only the diagonal entries of the latter. (1.5) is the first system of equations ($\mathbf{Ax} = \mathbf{b}$) that must be solved.

In *PHASTA*, an external library LesLIB from ACUSIM [40] is used to solve this (1.5) by the Conjugate Gradient(CG) method. After the (1.5) is solved, LesLIB uses GMRES to solve system (1.4). Within this commercial library, vectors from the solution of (1.5) are used to ensure that the (1.4) holds a tight tolerance for the continuity equation, thus allowing high quality solutions with rather loose tolerances. As this is a proprietary algorithm the details of this process are unavailable.

The data structure of the matrices of interest above, and also its fit to a mesh-based parallel implementation, will be further discussed in Chapter 5.2 later.

1.6 Summary

Based on our review of the current literature, we chose Stüben's AMG [4] to start our research into accelerating the solution of (1.5) but we have remained flexible in our implementation so as to enable extension to the other schemes described above. In this thesis we first focus on providing a more detailed description of serial AMG as a preconditioner for CG in Chapter 2. In Chapter 3, we will describe the GGB method in more detail. Alternative smoothing methods are prepared in Chapter 4. In Chapter 5 we will introduce the parallel AMG in detail. In Chapter 6 we will present efficiency study and results from large problems. Finally, in Chapter 7, conclusions and future research will be proposed.

CHAPTER 2
SERIAL AMG

In this section we will present the classical AMG algorithm [4] implemented in *PHASTA Incompressible* in serial. We will also discuss several key issues affecting convergence. Finally we will present preliminary serial results for some test cases.

2.1 Brief Description

As noted earlier, AMG is a combination of several basic linear algebra steps. It is common to break these steps into two parts; 1) those associated with the setup of the AMG iteration and 2) the actual cycle which is typically repeated many times in a particular solve. The setup phase in turn can be broken into various components that include; i) formation of the system of equation, ii) coarsening of the system at various levels, and iii) formation of the interpolation operators. Each of these steps is described below.

2.1.1 Setup Phase: Formation of $\mathbf{Ax} = \mathbf{b}$

Our target application of AMG is to solve the $\mathbf{Ax} = \mathbf{b}$ that arises from the discrete Pressure Poisson Equation (PPE) in *PHASTA*. We start from the data structures that already exist in the code. Recall the form of the system of equations:

$$\begin{pmatrix} \mathbf{K} & \mathbf{G} \\ -\mathbf{G^T} & \mathbf{C} \end{pmatrix} \begin{pmatrix} \Delta\dot{\mathbf{u}} \\ \Delta\dot{\mathbf{p}} \end{pmatrix} = - \begin{pmatrix} \mathbf{R_m} \\ \mathbf{R_c} \end{pmatrix}, \tag{2.1}$$

in which matrix $\mathbf{K}$ comes from the tangent of the momentum equation with respect to the acceleration, $\mathbf{G}$ comes from tangent of the momentum equation with respect to the pressure time derivative, and $\mathbf{C}$ comes from the tangent of the continuity equation with respect to pressure time derivative. Note that by neglecting certain terms from the tangent allows the tangent of the continuity equation with respect to acceleration to be represented by the negated transpose of $\mathbf{G}$[38, 37].

Applying static condensation of the time derivative of velocity variable results in the following left hand side and right hand side of the PPE explicitly[37].

$$\mathbf{A} = [LHS]_{PPE} = \mathbf{G^T \hat{K}^{-1} G} + \mathbf{C} \tag{2.2}$$

$$\mathbf{R} = [RHS]_{PPE} = -\mathbf{G^T \hat{K}^{-1} R^m} - \mathbf{R^c} \tag{2.3}$$

The matrices, coming from the Finite Element discretization $\mathbf{K}, \mathbf{G}, \mathbf{C}$, are the data source stored in sparse format (compressed row) that we use to form PPE. Prior to this work, our flow solver, only required the storage of $\mathbf{K}, \mathbf{G}, \mathbf{C}$ since the iterative solvers only require the capacity to evaluate matrix vector products (which can be done through a sequential multiplication of the tangent matrices as will be discussed in Section 5.1.1). However for AMG, an explicit form of the left hand side is required to perform the matrix coefficient analysis necessary to perform the coarsening.

In (1.4) and (1.5), $\mathbf{K}, \mathbf{G}, \mathbf{C}$ all have the same sparsity pattern. Each of these matrices is a direct assembly from integrals of shape functions and local solution values. Being sparse, they have the same matrix structures corresponding to the nodal connectivity of the geometric mesh and thus the sparsity is known prior to our solving phase. That is to say, if two nodes with global nodal number i and j are connected in the geometric mesh through an element, there will be non-zero entries $\mathbf{K}_{ij}$ in the matrix $\mathbf{K}$ (same for matrices $\mathbf{G}, \mathbf{C}$). These matrices carry different numbers of degrees of freedom for each non-zero entry. i.e. $\mathbf{K}$ has 9 values at each non-zero entry for 3D problem because of the momentum-acceleration coupling on three spatial directions. Similarly, $\mathbf{G}$ has 3 values for each non-zero entry and $\mathbf{C}$ has 1 value. However, the node-level sparsity pattern of these three matrices are the same.

We use the fill pattern matrix to describe the sparsity pattern and nodal connectivity. It is defined as a boolean matrix. Say, the entry is 1 for two nodes that are connected, 0 if two nodes that are not connected. In the code, it is stored in sparse format in *colm* and *rowp* arrays, as in the compressed row format [41].

colm stores the count of non-zeros for each equation block (i.e., the number of column-blocks the current equation has non-zero interactions with), *rowp* stores the row pointer for each matrix entry. In the discussion here and in the next chapters, we write the fill pattern matrix as $\mathbf{B}$ in general. In this way, we can write $\mathbf{K}$ as $\mathbf{K} = lhsK(\mathbf{B}, 1..9)$, and also $\left(\mathbf{G^T} : \mathbf{C}\right) = lhsP(\mathbf{B}, 1..4)$ as these are the data structures that have a same sparsity pattern of the fill pattern matrix $\mathbf{B}$, and the second rank denotes the block size. In these matrices, each non-zero entry of the corresponding fill pattern matrix represents a data block that contains either 9 or 4 numbers. We will use these symbols in the Appendix to describe algorithms later.

However, we cannot use the fill pattern matrix $\mathbf{B}$ to represent the PPE matrix. Since the PPE is formed from matrix-matrix products as in (1.5), the matrix sparsity pattern is changed by the matrix-matrix multiplication. Therefore, we must form the PPE explicitly and store it in new arrays with its own sparsity pattern. This formation, though not trivial, can be done with modest effort in serial case. However when we go to parallel later, it will be shown that the sparsity storage pattern we just mentioned above, will bring significant difficulties because of the multiplication. The formation step will become more complicated when we try to form PPE in parallel in Chapter 5.3.

Note that in practice, we form the left hand side PPE with diagonal pre-scaling. That is, we replace $\mathbf{A} = \mathbf{D^T A D}$, where $\mathbf{D} = diag(\mathbf{A})^{-\frac{1}{2}}$, to keep the diagonal entries of the pre-scaled PPE to be unity. This pre-scaled system will be used in the AMG preconditioner, which can further improve the condition number of the actual matrix to solve.

2.1.2 Setup Phase: Coarsening

After we have prepared the pre-scaled PPE, The first step in preparing an AMG V-cycle is to identify the nodes on a given level which will comprise the coarse nodes on the next coarser level: a process known as coarsening. Following Stüben [4], we do C/F splitting where C stands for a coarse node and F stands for a fine node. Two heuristic principles guide the splitting: 1: Each fine node (F) should strongly depend on at least one coarse node (C), and 2: no points in C should

depend strongly on other points in C.

The input of this process is the fine matrix arising from the Pressure Poisson Equation (e.g., the PPE matrix). The output of this process is an array recording strong correlation set, and a flag array marking each unknown with "Undecided(U)", "Coarse(C)" or "Fine(F)".

The first step is to generate a data set S, which stores information of strong dependencies of each i^{th} equation. Since each two related unknowns have such a dependency, the shape of S takes the sparsity pattern of the PPE matrix. So we can also write S as S_{ij}. i.e. matrix $\mathbf{S}$. The initial values of S_{ij} are all zeroes. To store strong correlation set $\mathbf{S}$ and $\mathbf{S}^T$, the following steps are performed. First, for each unknown i, we find the largest negative coefficient in the i^{th} row, and store it in g_i. Then we scan over all the non-zeroes of the fine matrix. For each negative matrix entry A_{ij}, we compare its value with g_i. If $A_{ij} < \epsilon_{str} \times g_i$, we say that j is in the strong correlation set of the i^{th} unknown, $j \in S_i$, where ϵ_{str} is a threshold coefficient describing how strong we desire each two unknowns are related. Notice that we are not using absolute values of matrix entries, which means we only consider negative values. (This treatment will be discussed more in later Section 2.3.3.) We actually implement this by adding 1 into S_{ij}, i.e. $S_{ij} \Longleftarrow S_{ij} + 1$. Also we compare A_{ij} with g_j. If such relation $A_{ij} < \epsilon_{str} \times g_j$, we know $j \in S_i^T$, which means $i \in S_j$. We do this by adding 2 into S_{ij}, i.e. $S_{ij} = S_{ij} + 2$. After treatment for all matrix entries, $\mathbf{S} = \{(i,j) \,|S_{ij} = 1.or.S_{ij} = 3\}$, $\mathbf{S^T} = \{(i,j) \,|S_{ij} = 2.or.S_{ij} = 3\}$. Details of implementation can be found in Appendix A.1.

The second step is to select coarse nodes and fine nodes based on strong correlation matrices $\mathbf{S}$ and $\mathbf{S}^T$. We will use a *measure of importance* array λ to determine the order of node selection. The λ array gives each node a score to determine the selection, and λ array changes while the picking process goes on. Following Stüben [4], the formula of λ array is:

$$\lambda_i = \left|S_i^T \cap F\right| + 2\left|S_i^T \cap U\right|, \quad i \in U\,, \tag{2.4}$$

where F, U,and C are defined in as "fine","undecided", and "coarse" set. The node with largest λ_i will be picked as the first coarse node. Then we alter the sets to

reflect the coarse node chosen and its strongly coupled "neighbors".

We describe the coarsening process in Appendix A.1 in a way that more closely mirrors the efficient algorithm implemented.

2.1.3 Setup Phase: Formulation of Interpolation Operator

With the coarsening algorithm defined and the detailed implementation described, the next step is to derive the interpolation operator Q_H^h in terms of the previous results.

Q_H^h is a coarse-to-fine operator or the interpolation operator. A left multiply of the interpolation operator to a coarse vector gives us a fine vector. So the dimension of the interpolation operator should be $n_1 \times n_2$, where n_1 is the dimension of the fine matrix, n_2 is the dimension of the coarse matrix. The i^{th} entry of the fine vector will be a weighted summation of the values of the coarse vector, and the weights are recorded in the i^{th} row of Q_H^h. The formulas of the weights can be given by either standard or direct interpolation. The formulas associated with direct interpolation are given next. For problems of our interest, standard interpolation, while resulting a denser coarse level matrix with more complex implementation, does not give better convergence rate. We will limit our discussion within direct interpolation from now on.

Direct Interpolation: First we define coarse/fine and strong correlation set:

$$C_i = C \cap N_i, \qquad C_i^s = C_i \cap S_i \tag{2.5}$$

$$F_i = F \cap N_i, \qquad F_i^s = F_i \cap S_i \tag{2.6}$$

where N_i is non-zero set of the i^{th} row in fine matrix.
Define $P_i = C_i^s$, we approximate:

$$a_{ii}e_i + \sum_{j \in N_i} a_{ij}e_j = 0 \Rightarrow a_{ii}e_i + \alpha_i \sum_{k \in P_i^-} a_{ik}^- e_k + \beta_i \sum_{k \in P_i^+} a_{ik}^+ e_k = 0, \tag{2.7}$$

where a superscript "+" meaning a positive entry in the row, and a superscript "-"

meaning a negative entry in the row.

We may define α_i and β_i in this form:

$$\alpha_i = \frac{\sum_{j \in N_i} a_{ij}^-}{\sum_{k \in P_i^-} a_{ik}^-} \tag{2.8}$$

$$\beta_i = \frac{\sum_{j \in N_i} a_{ij}^+}{\sum_{k \in P_i^+} a_{ik}^+} \tag{2.9}$$

Rearranging (2.7) we have:

$$e_i = \sum_{k \in P_i} w_{ik} e_k \tag{2.10}$$

$$\text{where } \; w_{ik} = \begin{cases} -\alpha_i a_{ik}/a_{ii} & \left(k \in P_i^-\right) \\ -\beta_i a_{ik}/a_{ii} & \left(k \in P_i^+\right) \end{cases} \tag{2.11}$$

Thus each entry of vector in the fine space e_i can be interpolated by the weighted sum of a set of e_k in a coarse space (P_i). We can build the whole interpolation matrix Q_H^h by constructing the interpolation weights row by row. Then we can transpose the interpolation matrix to get our restriction matrix $Q_h^H = \left(Q_H^h\right)^T$. An efficient implementation of the mathematical formulas described above is described in Appendix A.2. Also we will start to use $Q = Q_h^H$, and $Q^T = \left(Q_H^h\right)^T$ for subsequent representations to enable a more compact exposition of the formulas using these symbols.

The setup phase, as discussed above, consists of three main steps: (a) formation of problem of interest (PPE), (b) coarsening, (c) formation of interpolation operators. After these steps are applied, we have the coarse matrix built using (1.2). Then, the same steps are applied on the coarse matrix to build another level of coarser matrix. The procedure is thus executed recursively to build the multilevel AMG setup.

2.1.4 AMG V-cycle

In AMG, a V-cycle is a basic element. It takes the problem on the fine level, restricts the system to the coarse level, solves it in coarse level, and finally prolongates the correction back to the fine level. We call it a "V-cycle" because the flow chart follows a path of "V" shape. However, this image can be extended to a multi-level V-cycle. That is, we use another V-cycle in the "coarse" level solve, and get an approximate solution instead of an exact solution from the coarse level. This leads to a process that can be done recursively to produce as many levels as desired in a V cycle.

Given the recursive nature of the V-cycle, it is sufficient to discuss the neighboring two levels in a multi-level V-cycle process. The input of this process is a set of equations in the fine level, the output of the process is the AMG approximated solution of the fine level. We start from the equation $A^h,\ u_0^h\ , f^h$, where u_0^h is the initial guess for the linear system on fine level $A^h x^h = f^h$. Then we perform a pre-smoothing (e.g. Gauss-Seidel) on the guess: $u_1^h = Pre\_Smoother\left(u_0^h\right)$. Then we calculate the residual of the fine system $r^h = A^h u_1^h - f^h$. This gives us a measure of error in the fine level, telling us how far the guessed answer is off from the accurate answer. After this we restrict our error vector r^h to the coarse space: $r^h \Rightarrow r^H$. We also have already restricted the left hand side fine matrix A^h to the coarse space: $A^h \Rightarrow A^H$.

Next we "solve" the coarse equations and get an error vector in the coarse level: $e^H = \left(A^H\right)^{-1} r^H$. We say "solve" because we might perform different actions here: if this is not the coarsest level we would perform another V-cycle recursively for this solve, otherwise, for the coarsest level, we would solve with either a direct solver or an iterative solver. In the first scenario, a multi-level V-cycle is performed.

The final step corresponds to the "right half" of the letter "V" by prolongating the solution back to the fine space. First we interpolate the error vector in the coarse space to the fine space: $e^H \Rightarrow e^h$. Then we update our guessed solution: $u_2^h = u_1^h - e^h$, and perform another post-smoothing sweep on the result: $u_3^h = Post\_Smoother\left(u_2^h\right)$. This u_3^h will be our final approximate solution of this V-cycle.

Note that if using the Gauss-Seidel sweep as the smoother in the fine level, we

must pay careful attention to the order of G-S sweep to keep the V-cycle process symmetric, because the V-cycle is used as a preconditioner to Conjugate Gradient as CG requires a symmetric preconditioner. Specifically, we need to reverse the order of G-S smoothing. i.e. if we do forward-GS smoothing (smoothing from 1 to the number of unknowns in ascending order) before going to a deeper level, we need to do a backward-GS smoothing (smoothing from the number of unknowns back to one, decrementing by one) after. Also note, we use "red-black" smoothing for Gauss-Seidel, in which fine nodes are smoothed before the coarse nodes for pre-smoothing, and the order is reversed for post-smoothing. We will, in later chapters, discuss other smoother options like polynomial smoothing, together with extended boundary layer Gauss-Seidel.

Since one V-cycle gives us an approximate solution u to the linear system (A, f), we will further write this as $u = V^{-1}(A, f)$. An efficient implementation of a recursive V-cycle algorithm is described in Appendix A.3:

2.1.5 Conjugate Gradient Solver Accelerated with AMG

As noted earlier, we can write one iteration of an AMG's V-cycle in the mathematical form of:

$$u_{m+1} = V^{-1} u_m, \tag{2.12}$$

where m is the iteration number. V^{-1} indicates an AMG V-cycle preconditioning.

To use AMG as a preconditioner of CG has no difference from the usual preconditioned CG except that the preconditioning matrix is replaced with a V-cycle. The procedure is described in Appendix A.4:

We have implemented the above AMG V-cycle as a preconditioner to *lesLIB*, the linear solver used in PHASTA developed by ACUSIM. *lesLIB* provides us with an interface to enable us to make our AMG V-cycle perform as its preconditioner. The result of combining CG and AMG gives a dramatic acceleration to the solving process, as shown next.

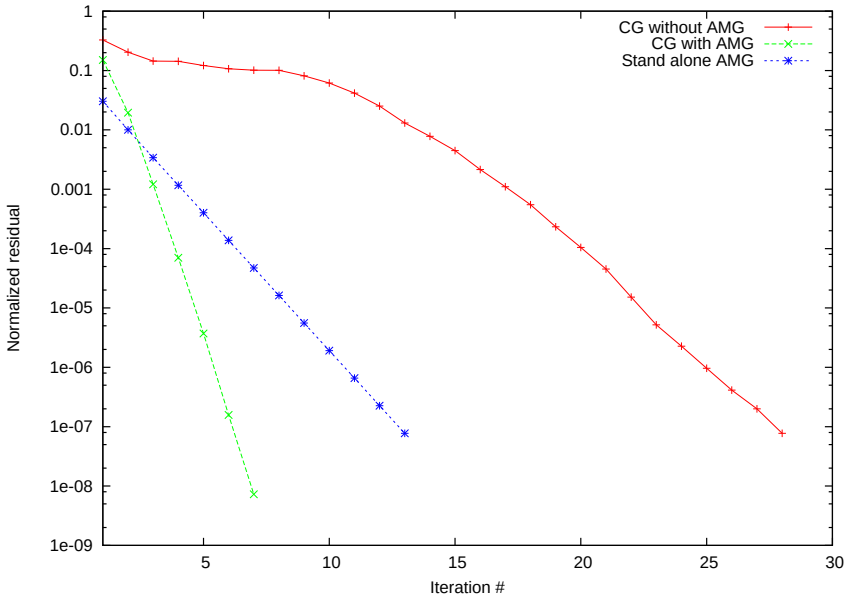

Figure 2.1: AMG Serial: $11 \times 11 \times 11$ **isotropic tube case**

2.2 Demonstration Cases

The methods described in previous sections were implemented within PHASTA and tested on two problems: a simple cube and an airfoil. Details are given in the following two subsections. Discussion of the results as well as an interpretation of the relative importance of improving various aspects of the AMG method will be given after all problems are introduced (in Section 2.3).

2.2.1 Test Case

The first case is flow in an isotropic channel (11^3 nodes). The resulting Pressure Poisson Equation from the problem is an 1331×1331 matrix. It is large enough for us to perform AMG, and small enough to be tested in a short time. The results are as shown in Figure 2.1.

From this plot we can see CG with AMG as V-cycle gives a very good convergence rate. The residual is reduced to 10^{-7} after 7 iterations. This is substantially better than CG alone. In this case, a 2-level AMG is used. The dimension of second level coarse matrix is 144×144, e.g., the system is reduced by a factor of about 9. We have also included results from the stand-alone AMG solver and we observe that although the residual after the first iteration of Stand-alone AMG is lower than

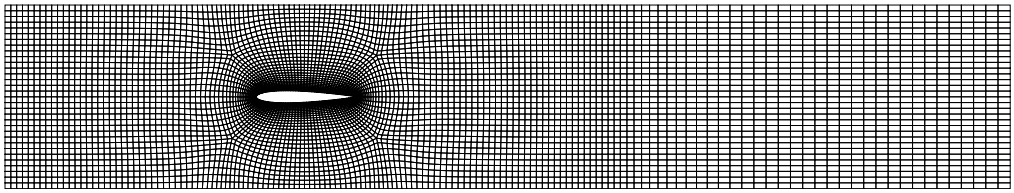

Figure 2.2: Airfoil Grid

that of the CG+AMG, the slope of CG+AMG is better than stand-alone AMG and it reaches the solution tolerance faster. This cube case is mainly for debugging and rationalize the implementation process.

2.2.2 Airfoil Steady State Case

Next we applied the solver to flow over NACA 0012 airfoil (cross-section shown as Figure 2.2) as our bench mark case. It has 112659 unknowns. The first run corresponded to an initial transient as the entire flow field was initialized to a speed of four parallel to the airfoil while the airfoil surface was modeled with a no slip boundary condition. This situation is typical of the first step of our computations and this poses significant challenges to the CG solver due to the rapid change in the pressure through fairly high aspect ratio elements at the stagnation point near the leading edge. In the test case shown below (Figure 2.3 and Figure 2.4.), we have set the convergence tolerance to be 10^{-7}.

Here, all AMG runs are 5-level, other parameters are the standard ones described in the previous section. We can see clearly that the solution converges dramatically faster than the original CG solver after we use AMG as preconditioner. And by combining these two methods, the AMG preconditioned CG solver beats both CG alone and AMG alone in performance by a factor of about 10 in terms of iterations.

For comparison we used commercial software SAMG, a commercial software developed by Stüben [4]. Specifically, we use SAMG as one AMG V-cycle to compare our RPI-AMG and obtain qualitatively similar performance. We observe that both CG + AMG and SAMG have hit a plateau at the first few iterations, and they have

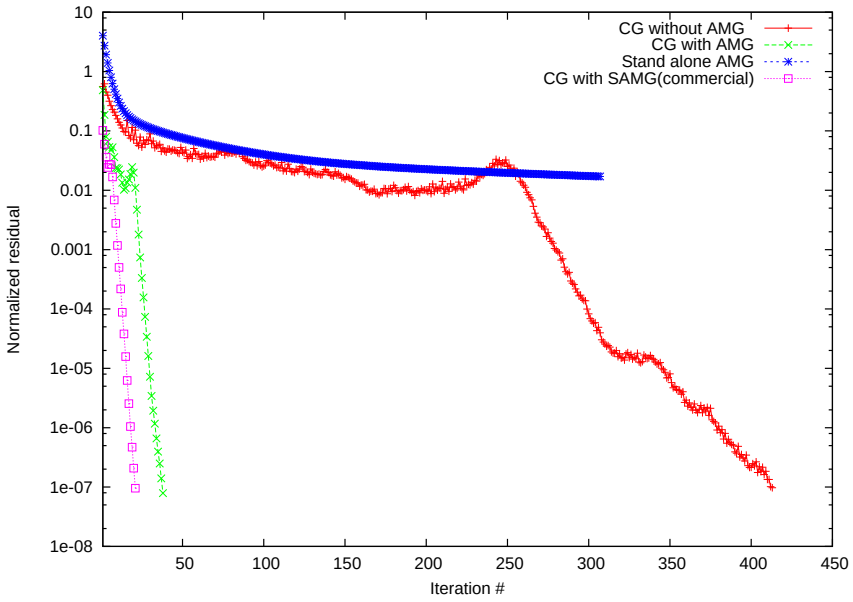

Figure 2.3: Airfoil Steady State case in Serial

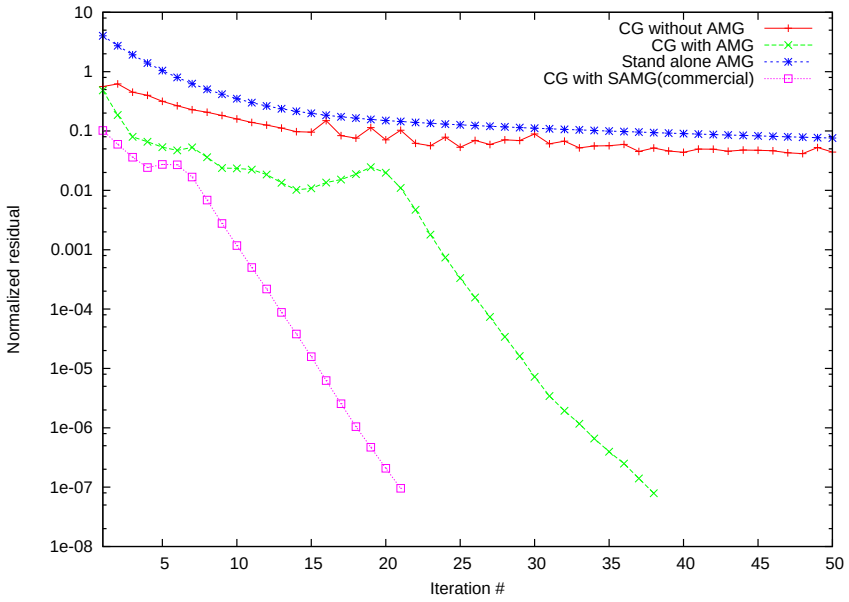

Figure 2.4: Airfoil Steady State case in Serial(Zoom In)

the same slope after the solution begins to converge as low as 10^{-3}. However, their point of departure is somewhat different.

2.3 Discussions

In this section we will discuss different implementation issues that might or might not slow down the convergence. These are issues associated with our flow problem and flow solvers. Mainly, we will be focusing on how the solution residual reduces with the number of iteration steps. We will focus on different aspects that might increase or decrease that number, keeping the residual tolerance fixed.

2.3.1 Level selection for multilevel AMG

Since the setup of AMG can be done recursively, it is always a question about where to stop and decide the maximum, highest (coarsest) level is achieved. A usual approach is to check the current coarsest level matrix, to see whether it is too dense that it can not be further coarsened efficiently; or it is small enough in size that a direct solve would be trivial. There can be good estimates for these criteria to make AMG for a single solve slightly more efficient. However, in these approaches, it is always required that some experiments be done with artificial inputs. This would make an algorithm less robust.

For the matrices from our problem, we observed that the number of coarsening levels, as long as it gives a solvable coarsest level matrix, does not have a large impact for the overall convergence. An example is shown for the airfoil case in Figure 2.5, where the size of the coarsest level ranging from 10 in 9 levels, to about four hundred in 4 levels.

Given the insensitive behavior for the number of levels to the same convergence criteria, we designed our coarsening algorithm in a robust way: we coarsen the matrix recursively to a stage that the coarsening can not be done any more (number of coarse nodes becomes fixed with each coarsening sweep). Usually, with this criteria, we end up with only several coarse nodes being carried on to the highest level. It is possible to have sub-efficient coarsening in higher levels, but since the size of the matrices are going down quickly especially in higher levels, the computational

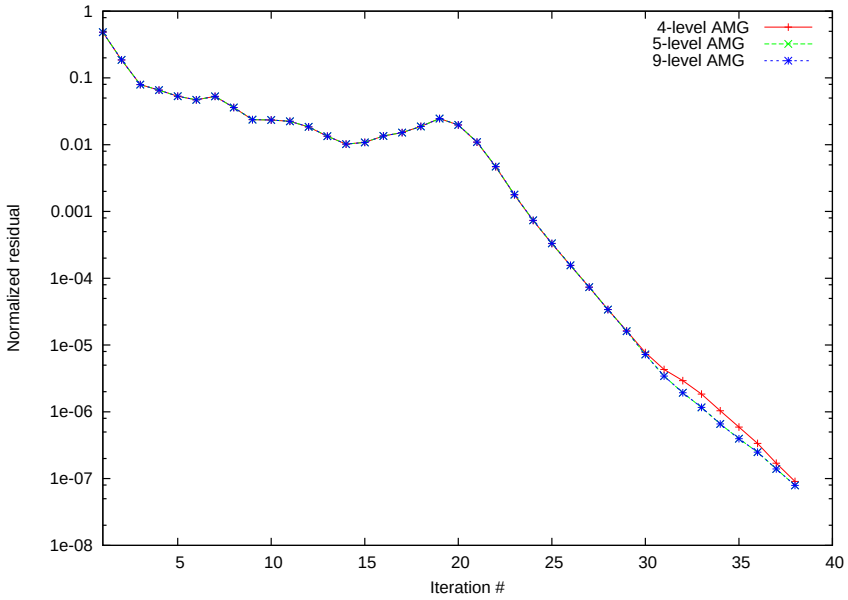

Figure 2.5: Airfoil Steady State case in Serial(different max-levels)

cost is still negligible compared to the whole problem (dominated by fine levels). Thus we end up with a robust coarsening algorithm without artificial inputs.

The merit of this approach is not obvious now but will be better observed when we move to the later section on parallel coarsening. When synchronization becomes an issue, the advantages of a more robust algorithm will be more apparent.

2.3.2 Matrix properties

Most of the theoretical studies of AMG with convergence proof assume that A is an M-matrix and indeed the method has been designed with this in mind. An M-matrix is a matrix that has positive and dominant diagonal entries. However, the matrices that come from the finite element discretization of (1.5) are not M-matrices. Although the diagonal entries are positive, off-diagonal entries are often large and positive too. In the airfoil case described above, the matrix has numerous off-diagonal entries larger than the diagonal term in absolute value after scaling. Despite not being a perfect M-matrix, AMG is still effective for this problem.

Also Stuben's AMG requires that the row sum of the matrix to be zero, which is the theoretical foundation of how AMG restricts/prolongates error. This is usually destroyed by pre-scaling of the original PPE matrix. Interpolation are not done

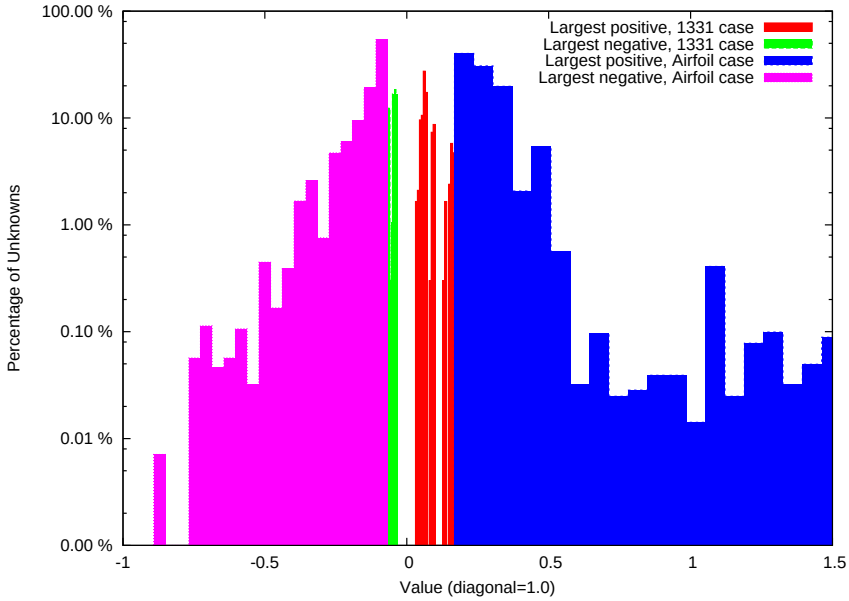

Figure 2.6: Matrix Property Check: Historgram of Large positive off-diagonal entries

perfectly in this case but still reflects the multilevel mapping in an efficient way to ensure the convergence.

In Figure 2.6 and Figure 2.7 these properties are examined in detail for a small test case and a larger airfoil case. However, although our matrices are "imperfect" for the algorithm, we still get a much better convergence in AMG.

2.3.3 Coarse/Fine splitting

In the setup phase of selection of coarser nodes, the first step is to generate "strong correlation set" S_i for each equation. In this step we ignore all the off-diagonal positive entries in the matrix. We only find most negative entry and compare its scaled value with other negative entries. Thus, all the off-diagonal positive entries in one equation are regarded as "weak correlated" to this unknown. In fact, off-diagonal positive entries play an important rule in the fine-tuning of classical AMG. How we treat these off-diagonal positive entries and recognize its correlation to other unknowns will affect the convergence rate. Despite an extensive literature survey to find an existing strategy for handling strong off-diagonal positive coefficients, it appears that whatever work has been done has not been published

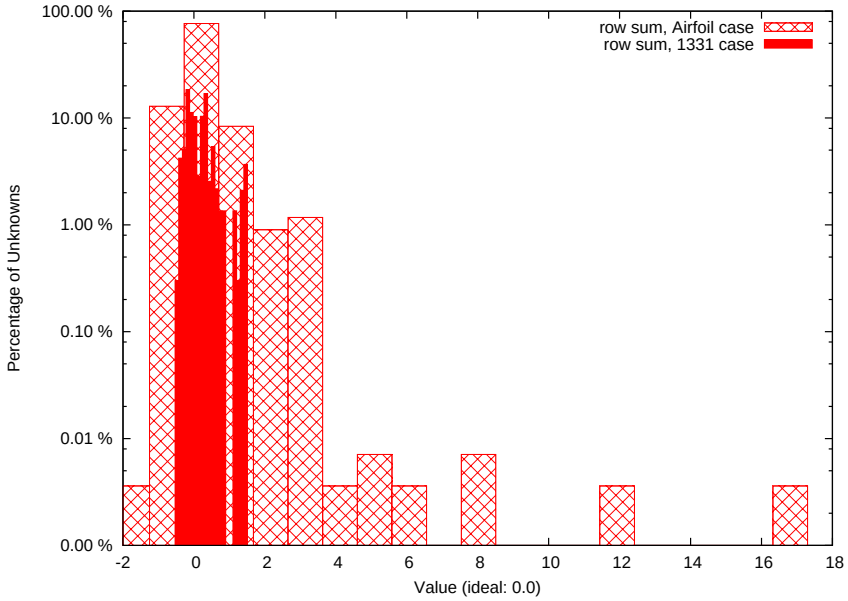

Figure 2.7: Matrix Property Check: Histogram of row-sum

[42]. The effect was observed by comparing our AMG implementation with SAMG implementation, whose details are not known but claimed to be have better off-diagonal positive entry treatment. The number of coarse nodes selected by our AMG and SAMG are different, which suggests that SAMG uses a better strategy to do C/F splitting than that described in the open literature. For now, the ignorance of off-diagonal positive entries is applied until insight of alternative treatments of positive entries are available. Tests with published treatments have not been fruitful (e.g. Stuüben's published method[4] of adding some of the strong positive entries to be redefined as coarse variable, has impaired the performance of AMG for our matrices) .

2.3.4 Interpolation

There are various ways to do interpolation. The simplest interpolation is the direct interpolation that we mentioned above. There are also more complex standard interpolation which has a more complex algorithm and uses more coarse nodes to approximate fine nodes, while keeping a fixed percentage of its total F-C connectivity, according to Stuben's algorithm [4]. We've tested both and our results suggest that direct interpolation gives better results than standard interpolation.

This appears to be partially because standard interpolation connects more coarse nodes to a fine one in such a way that is motivated by geometric arguments, which may not be applicable in our case.

2.3.5 Smoothing

Smoothing plays an important rule. Usually two Gauss-Seidel smoothing sweeps are applied in each level of the V-cycle, in backorder and forward-order respectively:

$$u_i = \frac{1}{a_{ii}} \left(b_i - \sum_{j \neq i}^{N} a_{ij} u_j \right) \tag{2.13}$$

which is performed in a descending or ascending order of i. To follow [4] and achieve better convergence, we changed the order to C/F for post-smoothing and F/C for pre-smoothing. That is, for post-smoothing, we do coarse nodes first then fine nodes, and for pre-smoothing, we do fine nodes first then coarse nodes. Of course, the two smoothing passes are done in reversed orders to satisfy the symmetric nature of a preconditioner of Conjugate Gradient. For the airfoil steady state case, this reduces our CG vectors by about 10 percent, though this should vary with the problem chosen. We also looked at using Jacobi sweep as the smoothing in AMG. This is a weaker smoother but it is of interest because it would be easily parallelized. However the results were not encouraging. For complicated problems, AMG with Jacobi stagnates even when employed with various damping factors, causing CG to not converge. The reason for this will be studied using a spectrum analysis in a later chapter using GGB. Also there will be more discussion on smoothing in the parallel section.

2.3.6 Non-critical factors

It is possible to use just smoothers instead of a direct solver at the coarsest level. However, our study shows that this has a negligible effect on total convergence rate for the problems we have considered thus far. Specifically, if we use the airfoil steady state case with different coarsest solvers to check the convergence curve, we see in Figure 2.8, they are almost identical. A second factor that proved unimportant was truncation. Truncation in the context of AMG means that when constructing

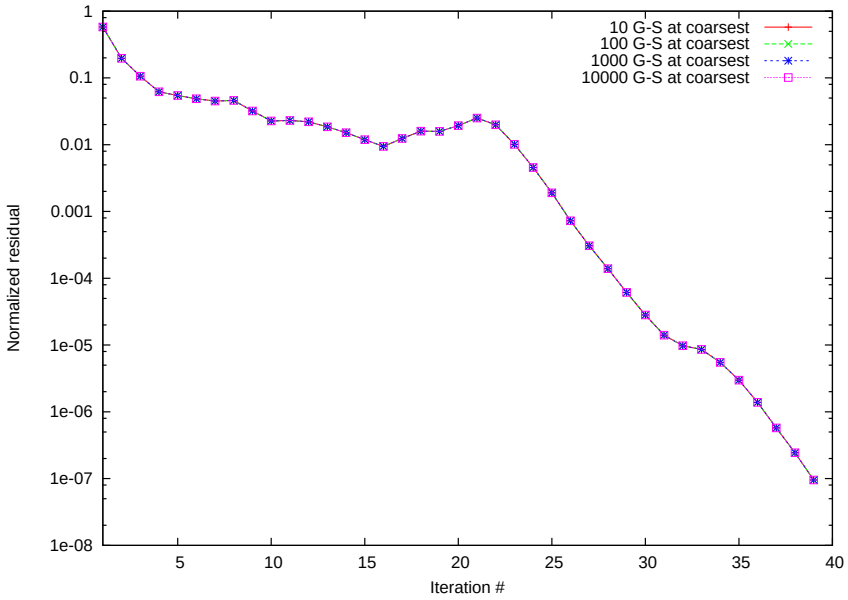

Figure 2.8: Airfoil Steady State case in Serial, Different Coarsest smoother

interpolation/extrapolation operators, you neglect nonzero values whose absolute value are less than a threshold to save memory and to reduce computational time. The threshold considered ranged from 0.5 to 0.9 and yet no considerable degradation are observed in convergence rate, while the computational time improved slightly. The final factor which seems to be unimportant is the strong correlation coefficient, ϵ. It does not have a significant effect on the convergence rate in terms of number of iterations especially when a multilevel (more than two) AMG is used. However, increasing this parameter can result significant increase of the non-zeros of coarser matrices, which increases the CPU time for coarser level solve. Usually we keep this at 0.25 for all problems.

CHAPTER 3
GGB with AMG

When applying AMG to problems that it has not been customized for (e.g. non M-Matrix), sub-optimal performance is often observed. While the ideal approach would involve optimizing each step until the equation system and algorithm are mathematically matched (with a mathematical proof of convergence), the pace of progress in this area has not kept up with the needs in engineering. An interesting alternative method that mainly focuses on the "imperfections" and deals with them in a robust way has recently been developed. This method is termed the "Generalized Global Basis" (GGB) method [9, 25, 24, 43, 44].

When we push AMG to parallel, we may hope that GGB is able to fix degradation caused by parallel, a hypothesis that will be examined later. At that time, we will discuss GGB in parallel and see how this accelerator can be integrated into the flow solver. In this chapter we first discuss GGB in serial.

3.1 Brief Description

The ultimate goal remains to solve the linear system $\mathbf{Ax} = \mathbf{b}$ using iterative methods. We want to find $\mathbf{x}_a$ such that the L2-norm $||\mathbf{r}_a||$ of the residual $\mathbf{r}_a = \mathbf{Ax}_a - \mathbf{b}$ is smaller than our tolerance ϵ_{tol}. We start off from an initial guess $\mathbf{x}_0$, then after n iteration of any given method (i.e. AMG's V-cycle), $\mathbf{x}_0$ is updated to $\mathbf{x}_n$. We define the "error vector" to be $\mathbf{e}_i \equiv \mathbf{x}_i - \mathbf{x}$ where $\mathbf{x}$ is the exact solution of the system.

At each iteration, $\mathbf{e}_i$ is updated to $\mathbf{e}_{i+1}$. This process can be written as $\mathbf{e}_{i+1} = \mathbf{E}\mathbf{e}_i$, where $\mathbf{E}$ is the iteration matrix for error vectors. Since we want the L2-norm of error vector series tend to zero as we increase the iteration number, the spectral radius of $\mathbf{E}$ should be less than unity. In fact, for the iteration process to converge fast, small eigenvalues of $\mathbf{E}$ are more desirable. In mathematical form, we say:

$$\mathbf{e}_n = \mathbf{E}^n \mathbf{e}_0 \tag{3.1}$$

$$\mathbf{e}_n \to 0 \text{ , if } \rho(\mathbf{E}) < 1 \tag{3.2}$$

The GGB method has two steps. The setup step is the analysis of the iteration matrix for error vectors $\mathbf{E}$ for any given iteration cycle. In this step, GGB picks out eigenvectors that have the largest eigenvalues in magnitude in $\mathbf{E}$, which are "problematic" modes that slow down the iteration process. Then GGB constructs an additional restriction/prolongation operator by spanning these eigenvectors. A very coarse system is then constructed from the original matrix $\mathbf{A}$ and the interpolation operators in a similar way to AMG coarse matrices, which only contains the modes that need to be eliminated.

The application step is just an additional V-shape-cycle after a regular iteration, which we refer to as a "G-cycle". In this step, GGB uses the newly built restriction/prolongation operator to scale the system into a smaller one with only the problematic modes. Then it directly solves the smaller system and prolongates the solution back. After this, the original solution to the fine system will be modified by GGB's result to eliminate the components which cause a slow convergence. There are more details in references [9, 24, 25].

Given AMG's V-cycle as the iteration cycle to be accelerated by GGB, this process can be given mathematically:

From the AMG process defined in previous sections, we can deduce the iteration matrix for error vector:

$$\mathbf{e}_{i+1} = \mathbf{E}\mathbf{e}_i \tag{3.3}$$

$$= \mathbf{S_l T S_r}\, \mathbf{e}_i, \tag{3.4}$$

where $\mathbf{S}_l$, $\mathbf{S}_r$ are post-smoothing and pre-smoothing, respectively, and $\mathbf{T}$ is the AMG solve matrix that brings the system to coarse level, solve it and bring back. This matrix varies with different AMG settings. For a 2-level AMG with direct solve at

coarsest level, $\mathbf{T}$ can be written as:

$$\mathbf{T} = \mathbf{I} - \mathbf{Q}\mathbf{A}_\mathbf{2}^{-\mathbf{1}}\mathbf{Q}^\mathbf{T}\mathbf{A}, \tag{3.5}$$

where $\mathbf{A}$, $\mathbf{A_2}$ are the original fine matrix and second level coarse matrix, respectively, and $\mathbf{Q}$ and $\mathbf{Q}^T$ are prolongation/restriction operator for the AMG part, respectively.

If $\nu_1 ... \nu_k$ are the eigenvectors of $\mathbf{E}$ from the largest k eigenvalues, the prolongation operator of a G-cycle can be constructed as the span of the highest modes:

$$\mathbb{Q}_\mathbf{f} = [\nu_1 ... \nu_k] \tag{3.6}$$

Finally, the mathematical form for G-cycle of error vectors:

$$\mathbf{e}_i' = \mathbf{F}_{GGB}\mathbf{e}_i \tag{3.7}$$

$$= (\mathbf{I} - \mathbb{Q}_\mathbf{f}(\mathbb{Q}_\mathbf{f}^\mathbf{T}\mathbf{A}\mathbb{Q}_\mathbf{f})^{-\mathbf{1}}\mathbb{Q}_\mathbf{f}^\mathbf{T}\mathbf{A})\mathbf{e}_i \tag{3.8}$$

In the actual application of GGB to AMG, the error iteration matrix $\mathbf{E}$ becomes the AMG preconditioning matrix for error vectors, though usually not in explicit form. This can derived from (2.12) for residual vector. The form is:

$$\mathbf{e}_{i+1} = (\mathbf{I} - V^{-1}\mathbf{A})\mathbf{e}_i \tag{3.9}$$

$$= \mathbf{E}\mathbf{e}_i \tag{3.10}$$

Using (3.9), we can calculate the eigenvalues and eigenvectors of $\mathbf{E}$ only given a process for preconditioner V^{-1}, rather than having explicit form of $\mathbf{E}$ itself.

3.2 GGB Implementation

The open-source ARPACK/PARPACK package is used in our GGB implementation [45]. We use ARPACK to complete the first step described in previous section of the GGB algorithm, that is, to solve eigenvalue/eigenvector problem for a linear system. There are several advantages of ARPACK. It uses implicitly restarted

Arnoldi method to compute a subset of eigenvalue/eigenvectors, which is computationally efficient. More importantly, it uses a reverse communication interface to the library which is convenient for integration with AMG (will elaborate next). Last, it is open source allowing it to be customized to our needs.

The reverse communication interface of ARPACK requires that the user provide a matrix-vector product $\mathbf{q} = \mathbf{Ap}$, to solve the eigen-problem of $\mathbf{A}$. ARPACK will generate an initial vector $\mathbf{p}$, then after the user-defined function performs an $\mathbf{Ap}$ to produce $\mathbf{q}$, ARPACK is called again with the result $\mathbf{q}$ as input. This eliminates the need to build the error iteration matrix $\mathbf{E}$ in explicit form. Instead, we can reuse AMG V-cycle routines that iterate the solution vector to analyze the iteration matrix of error vectors by (3.9).

In ARPACK, the accuracy of eigenvalue/eigenvectors is a user-specified criteria. In our problems, we use a relatively loose tolerance of 10^{-3}. By decreasing the number, e.g. to 10^{-7}, the accuracy of eigenvalue/eigenvectors improves. However, there are more iterations required for this improvement and thus more computational time. Overall, we want to use GGB to improve our convergence for CG preconditioned with AMG. In that context, a tighter tolerance does not appear to have a noticeable impact on the convergence rate.

After the setup is complete, with $\mathbb{Q}_{\mathbf{f}}$ defined, we are now able to describe the G-cycle which is quite similar to AMG's V-cycle. The main difference is that the G-cycle is a 2-level cycle with prolongation/restriction operator being the span of eigenvectors from the GGB setup step. Details of construction of G-cycle together with AMG's V-cycle are shown in Figure 3.1.

This setup destroyed the symmetry in the sense of AMG and GGB operations. One might expect this break from symmetry to adversely affect the preconditioning of the conjugate gradient method but, we observed that, for the problems we have considered thus far, this is not the case though others have had different experiences[25, 9]. While we have only studied a small number of problems and this may be a problem in future applications, a symmetric V-G-V cycle roughly doubles the amount of work required for preconditioning. Since the symmetric cycle does not reduce the number of CG vectors by a factor of two compared to a V-G cycle,

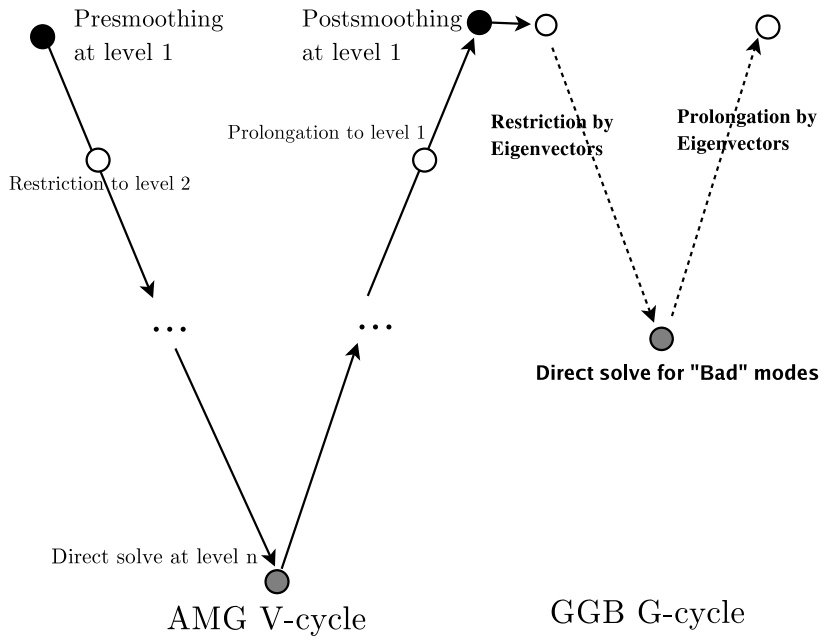

Figure 3.1: AMG V-cycle followed by GGB G-cycle

it has, heretofore, not been justified from a computational cost perspective.

3.3 GGB Results

To understand GGB's potential, we first study it in serial. The first case is our test case of flow in isotropic channel that we used previously. We applied GGB with only 10 eigenvectors. i.e. 10 eigenvectors with largest magnitude of eigenvalues are picked out to form the G-cycle. A mildly positive result was observed as CG iteration goes down from 7 to 6. This result can be explained due to the better nature of this small matrix which is close to an M-matrix which is handled well by AMG alone, leaving few if any eigenvalues near 1 (e.g. no bad modes in AMG process to fix). And it results a better coarsening from AMG alone so there are very little eigenmodes that has eigenvalues close to 1.

The airfoil steady state case is also tested with GGB application with the same parameters. As the problem is more complicated and with further deviation from M-matrix, better improvement with GGB is observed. With the G-cycle, the CG iterations go from 37 to 19 for the same convergence tolerance of 10^{-7}, as shown in Figure 3.2 and Figure 3.3.

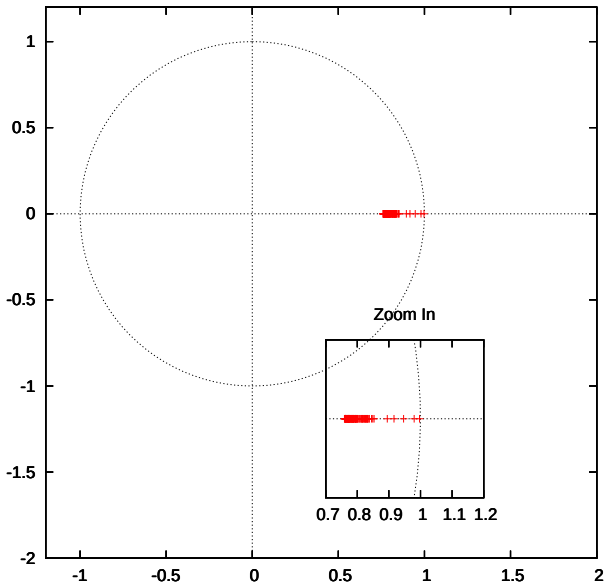

Figure 3.2: Spectrum of AMG iteration matrix w/ Gauss-Seidel smoothing, Airfoil case

Case	CG	CG+AMG(V)	CG+AMG+GGB (VG)
Channel Flow n=1331	30	7	6
Airfoil n=112659	413	37	19
Case	CG	CG + AMG(VV)	CG+AMG+GGB(VGV)
Channel Flow n=1331	30	4	4
Airfoil n=112659	413	26	14

Table 3.1: AMG and GGB Serial Test, 10 eigenvectors, solve to 10^{-7}, in number of iterations

It is possible to use more than 10 eigenvectors in GGB. However, this would be more expensive in computational time and memory cost in terms of setup and per iteration cost. If the troublesome modes (e.g., those near 1) are covered in the first 10, this additional cost will not be offset by enough reduction in iterations. This parameter is problem dependent, for larger tougher problems this number should be raised.

We also tested applying the G-cycle in a symmetric way by putting a post V-cycle after. This makes our CG preconditioner; "V-cycle, G-cycle, V-cycle", which we call a "VGV-cycle" in Table 3.1. A "VG-cycle" represents a preconditioner with

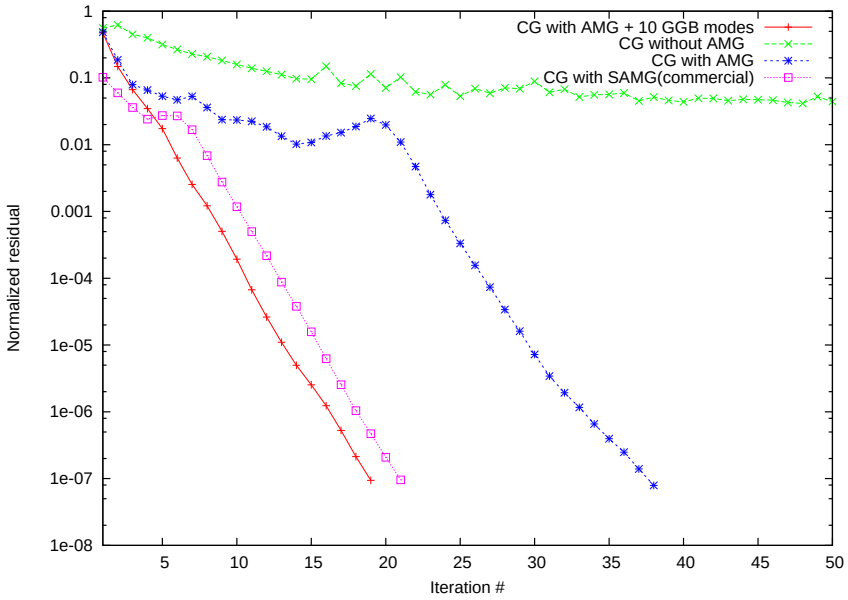

Figure 3.3: Airfoil steady state with GGB

a V-cycle followed by a G-cycle. The "VGV-cycle" does not look promising because V-cycle is expensive and would have to reduce the number of iterations by nearly half to be justified.

We also studied damped Jacobi as a replacement for Gauss-Seidel as our AMG smoother. This was motivated by the fact that damped Jacobi is trivial to make parallel and if GGB could make it viable, we would have the prospect of not suffering with the difficulties associated with the chaotic Gauss-Seidel parallel smoother. However, GGB can not fix all the bad modes introduced by Jacobi. By looking into the spectrum of AMG V-cycle with Jacobi smoother as shown in Figure 3.4, we find that there are dozens of modes with eigenvalue's magnitude larger than unity, and more close to unity. In this case, GGB will grow very expensive and close to solve a direct eigen problem if we want to eliminate all of the bad modes. When a reasonable number of eigen vectors were used (10) the convergence problem persists (e.g., slow/no convergence on tough problems like the airfoil) So we conclude that GGB will help Gauss-Seidel smoothed AMG greatly but cannot rescue a weak smoother like Jacobi.

Note from Figure 3.3 that the convergence plateau that was previously observed in the Airfoil disappeared with the application of GGB. The improvement

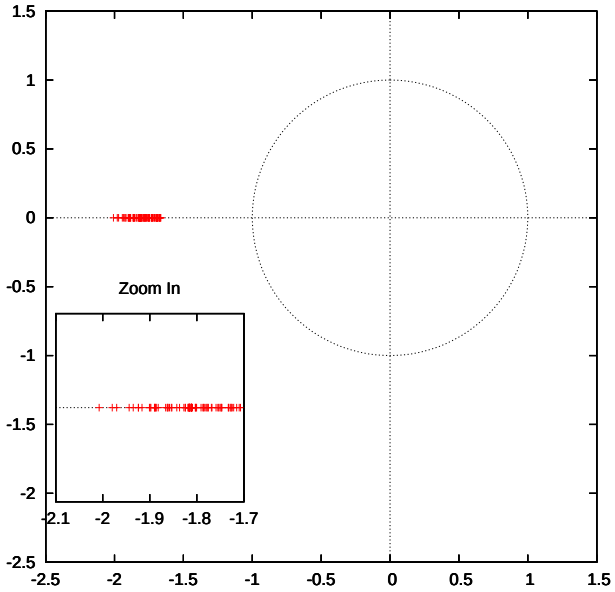

Figure 3.4: Spectrum of AMG iteration matrix w/ damped Jacobi smoothing, Airfoil case

even surpasses SAMG suggesting that GGB is able to compensate for unpublished fine tuning to AMG that is present in SAMG but not available to our AMG. Cost issues and parallelization of GGB will be discussed in later chapters.

CHAPTER 4
POLYNOMIAL SMOOTHING

In addition to Gauss-Seidel which is the most commonly used smoother in AMG, we would like to explore further options for smoothing. The motivation of this investigation is the intrinsic difficulty of parallelization of Gauss-Seidel. Since we are going to apply AMG in parallel, there is need for a smoother that would meet the requirement of parallelization yet maintain the efficiency as in serial. In practice, Gauss-Seidel is not such a smoother. While Jacobi smoother is easy to parallelize, it is not strong enough to smooth out error components in an approximate solution [16, 30, 15].

Polynomial smoothing method provides a promising option. It only requires matrix-vector products being carried out correctly. Compared to Gauss-Seidel, this is much easier to implement in parallel. Efficiency-wise, it was claimed [16] to be comparable to Gauss-Seidel in smoothing out errors. Since it fits in the requirement of being efficient and also easy to parallelize, in this chapter we explore the development of polynomial smoothing within parallel AMG. However, in this section, we will mainly focus on the serial performance of polynomial smoothing. Further parallelization of this smoother will be discussed in later chapters.

Our development follows the polynomial smoothing method proposed by Adams *et al.* [16] but also contains some modifications suggested within the open-source computational software *Trilinos* from Sandia National Lab [12, 46].

4.1 Brief Description

To solve a symmetric positive definite linear system $\mathbf{Ax} = \mathbf{b}$ using iteration-based method, we start with an initial guess. Then the approximate solution is updated at each iteration step with a computation involving the right hand side vector $\mathbf{b}$, the current iterate of the solution $x^{(n)}$, and the matrix $\mathbf{A}$. For a polynomial smoother, the basic form of this iteration is:

$$x^{(n+1)} = x^{(n)} + \sum_{0 \le j \le m} \alpha_j \mathbf{A}^j (b - \mathbf{A}x^{(n)}) \tag{4.1}$$

Superscript n indicates the iteration count. Parameter m indicates the order/degree of polynomial smoothing. If $m = 0$, this method reduces to relaxed Jacobi method as the above update equation reduces to a Jacobi iteration $\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \alpha_0(\mathbf{b} - \mathbf{A}\mathbf{x}^{(n)})$. The α_j's are the polynomial coefficients and are precomputed before smoothing [16]. If α_j's are carefully chosen, the approximate solution from (4.1) will minimize the spectral radius of the error propagation matrix for the linear system $\mathbf{Ax} = \mathbf{b}$. The method of selecting these coefficients is provided in [16] as well as in the source code of *Trilinos* [12, 46], and will be given below. (4.1) serves as the smoother step in AMG and is intended to replace the Gauss-Seidel sweep as the smoother.

The MLS (Multilevel Smoother Polynomial) smoother is introduced in [16] for smoothing construction. Here we just give the formulas for implementation. For more mathematical details please refer to [16, 27]. We will use S_l to define the pre-smoothing operator for level l. The pre-smoothing operator is defined as:

$$S_l = p_1(A_l) \equiv \left(I - \frac{1}{r_1} A_l\right) \cdots \left(I - \frac{1}{r_d} A_l\right), \tag{4.2}$$

where $r_k = \frac{\rho(A_l)}{2}(1 - \cos\frac{2k\pi}{2m+1})$, $k = 1, \cdots, m$, m being the MLS degree and $\rho(A_l)$ is the spectral radius of matrix A_l. Usually $\rho(A_l)$ is estimated from the largest eigenvalue of A_l.

For post-smoothing, MLS [16] proposes the construction of an intermediate matrix $A_l^S = S_l^2 A_l$. It was shown [16, 27, 47], that (4.2) minimize the spectrum of A^S. Using A^S, one can then define the post smoothing operator:

$$\hat{S}_l = p_2(A_l) \equiv I - \frac{\omega}{\bar{\rho}(A_l^S)} A_l^S, \tag{4.3}$$

where $\bar{\rho}(X)$ defines the upper bound on the spectral radius of the matrix X. The relaxation parameter ω is usually set between 1 and 2.

The equations above can be reduced to series of matrix-vector products $\mathbf{q} = \mathbf{Ap}$ with explicitly given coefficients. i.e. α_j's from (4.1). Once the coefficients are

prepared for each level of AMG, we will be able to combine these with parallel matrix-vector products and use the polynomial smoother in the preconditioner to solve $\mathbf{Ax} = \mathbf{b}$.

4.2 Testing and Comparison

Before implementing polynomial smoother into *PHASTA*, a series of stand-alone numerical tests were performed in *Trilinos*[12, 46]. The purpose of these tests were mainly for validation of the algorithm and examining the efficiency of MLS compared to Gauss-Seidel. Although *Trilinos* uses aggregation type AMG as its coarsening scheme, which is different from the classical AMG used in *PHASTA*, the efficiency of the smoother, nevertheless, can still be estimated by comparing Gauss-Seidel and MLS smoothing with the same aggregation AMG setup for coarsening.

As shown in (4.3), the post- MLS smoother requires double application of the pre-smoother and one more matrix-vector product. Also, each of the pre- and post-MLS smoother requires eigenvalue calculations. As shown in (4.2) and (4.3), we need to estimate the spectral radius of matrices in the calculation of $\rho(X)$. Although only the largest eigenvalue for the input matrix X is needed, these eigenvalue problems can still be expensive. While a pre-calculation of the largest eigenvalue is possible, the application of post-smoothing operator still remains a significant expense compared to the pre-smoother. We want to evaluate if this cost is necessary for our problems by stand-alone *Trilinos* tests. The test involves source code level debugging and inspection with lack of support from the documents. The test details are described in Appendix B. In conclusion, the numerical experiment suggests that we can use "less-computational-work" MLS pre-smoother as both pre- and post-smoother and still keep, or rather gain, efficiency, as they both converge to the same correct solution within same order of iterations for our Pressure Poisson problem.

Given test procedures and detailed results in Appendix B, we want to explore different possibilities of polynomial smoother with our classical AMG. Here we propose several algorithms for testing.

The original MLS algorithm was developed together with sa-AMG. Thus, the smoother is left multiplied to prolongation operators to form a new prolongation

operator:

$$\hat{Q} = SQ, \tag{4.4}$$

and the restriction operator reads:

$$\hat{Q}^T = Q^T S \tag{4.5}$$

This leads to coarser matrix:

$$A^H = \hat{Q}AQ = Q^T SASQ \tag{4.6}$$

However in classical AMG, because the interpolation operator is no longer piecewise constants as in sa-AMG, combining smoother and interpolation operator like in (4.4) will result in highly denser coarse level matrices A^H with much more non-zeros, and will raise the complexity of the algorithm to a prohibitively high level which makes it less effective. For this reason, we do not combine smoother and interpolation operators to form $\hat{Q}$. Instead, we will use original interpolation operator Q, and use S to represent smoother.

To avoid this complexity issue, in our first set of candidate approaches to polynomial smoothing, we propose three alternative approaches/deviations from the algorithm in [16]. For the first alternative from this set, we propose replacing the coarser level solve $(Q^T SASQ)^{-1}$ with classical AMG coarser level solve $(Q^T AQ)^{-1}$, but then use smoothers on interpolation and restriction operators as in sa-AMG form; For the second alternative, we propose replacing all $\hat{Q}$ with Q (that also results the coarser level replacement as in the previous alternative); For the third alternative, we propose replacing all $\hat{Q}$ with Q and also added an S smoothing to the post smoothing process. These alternatives are referred as Algorithm 1a, 1b, and 1c respectively, and the details can be found in Appendix C.2.

Algorithm	q=Ap	Error Propagation Matrix
1a	11	$E_{1a} = \hat{S}(I - SQ(Q^TAQ)^{-1}Q^TSA)S$
1b	7	$E_{1b} = \hat{S}(I - Q(Q^TAQ)^{-1}Q^TA)S$
1c	9	$E_{1c} = \hat{S}S(I - Q(Q^TAQ)^{-1}Q^TA)S$
2	14	$E_2 = \hat{S}S(I - Q(Q^TAQ)^{-1}Q^TA)\hat{S}S$
3	4	$E_3 = S(I - Q(Q^TAQ)^{-1}Q^TA)S$

Table 4.1: Polynomial Smoothing algorithm alternatives

Another candidate alternative is to use two full smoothers on pre- and post-smoothing, and sticking with classical AMG by having the non-smoothed coarse level. We refer it as Algorithm 2. The final attempt was made to only use S smoother at pre- and post- smoothing process. This is the cheapest smoother but a further deviation from the original. We refer it as Algorithm 3. These two smoothers are motivated by the *Trilinos* test as shown in Appendix B.

All the proposed candidate smoothers use classical AMG kernel instead of sa-AMG, so they all lack solid mathematical convergence proof. We test them as numerical experiments. But as the previous *Trilinos* test suggests, they might be able to converge to the solution with reasonable iterations. Since they are much alike in the mathematical forms, we just write the summary part of these alternatives as shown in Table 4.1. We also listed the number of matrix-vector products required for these alternatives. Number of matrix-vector products is an important major measure for cost, so we are concerned not only in the number of iterations but also the real CPU time consumption. For a detailed implementation please refer to Appendix C.2 to Appendix C.4, in which Algorithm 1a, Algorithm 2, Algorithm 3, are given in pseudo code.

Using these different approaches, we run AMG on Airfoil case to 10^{-7} as the benchmark problem. Results are shown in Figure 4.1 (without GGB) and Figure 4.2 (with GGB). We summarize the results in Table 4.2.

Conclusion: Algorithm 3 is the best in terms of CPU time. Other algorithms either stagnate or require a lot of Ap-products that won't compensate by the reduction of number of iterations (algorithm 1c); or the spectrum of $\mathbf{E}$ is so bad that GGB is less effective (algorithm 2).

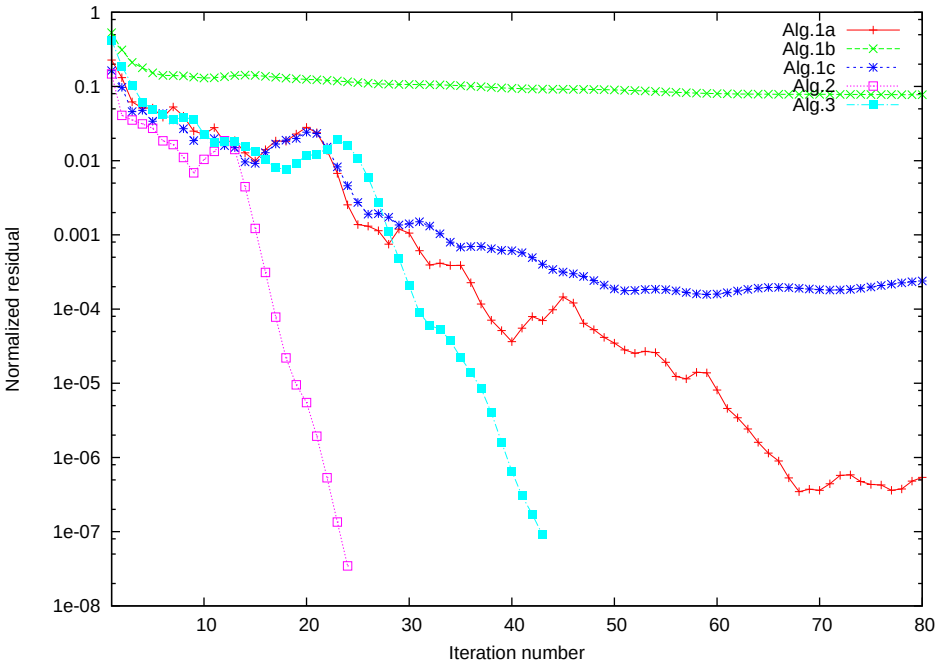

Figure 4.1: Algorithm comparison of polynomial smoothing, Airfoil case

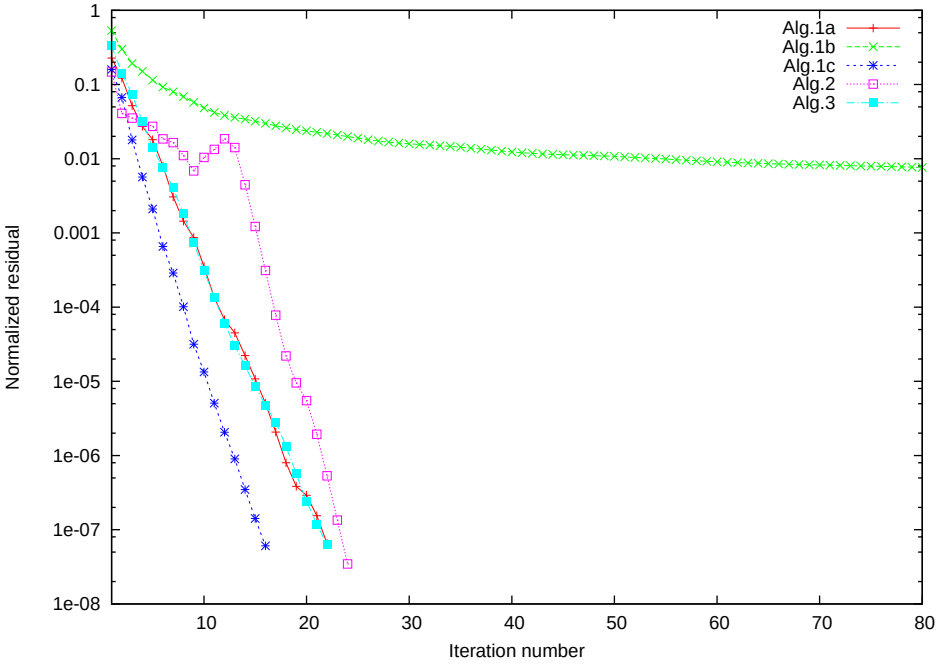

Figure 4.2: Algorithm comparison of polynomial smoothing and GGB, Airfoil case

N	Number of iterations solve to 10^{-7}.
N^g	Number of iterations solve to 10^{-7} with GGB acceleration.
T_t, T_t^g	Total time for solving with/without GGB
T_s, T_s^g	Total time for setup with/without GGB

Name	q=Ap	N	T_t	T_s	N^g	T_t^g	T_s^g
1a	11	91	204.10	27.68	22	53.18	1082.06
1b	7	Stagnate	—	—	Stagnate	—	—
1c	9	Stagnate	—	—	16	32.28	333.74
2	14	24	68.24	27.69	24	72.68	2260.12
3	4	43	36.31	27.68	22	22.06	170.96

Table 4.2: Summary of Polynomial Smoothing Algorithms

Considering convergence rate and also the CPU cost, we choose algorithm 3 as the polynomial smoother for our problem, Algorithm 3 will be tested against traditional Gauss-Seidel in the next subsection both in number of vectors and CPU time towards convergence.

When applying MLS smoother, (4.1) is used for each level on pre- and post-smoothing with different sets of α_j's. The kernel of the MLS smoother on different levels are **Ap**-products given **A**'s being different coarsened PPE on higher levels. This is promising for parallel MLS smoothing.

Equation (4.2), as referenced from [16], are derived based on a pre-scaled matrix with diagonals being unity. For the PPE extraction described before, the resulting matrix does not satisfy this property. This property is easily corrected in the PPE extraction phase and in the construction of coarsened matrices by an additional diagonal scaling.

4.3 Results and Efficiency of MLS

Polynomial smoothing (MLS) as the smoother of AMG is tested using the Airfoil case as we did in Section 2.2.2 and the results are shown below in Figure 4.3. In terms of the number of iterations, polynomial smoothing converges at a similar rate as Gauss-Seidel when used at 2nd-order. It converges faster than Gauss-Seidel when used in 4th-order. Also, polynomial smoothing does not introduce more "problematic modes" into the AMG cycle, since we observe that it can be accelerated by

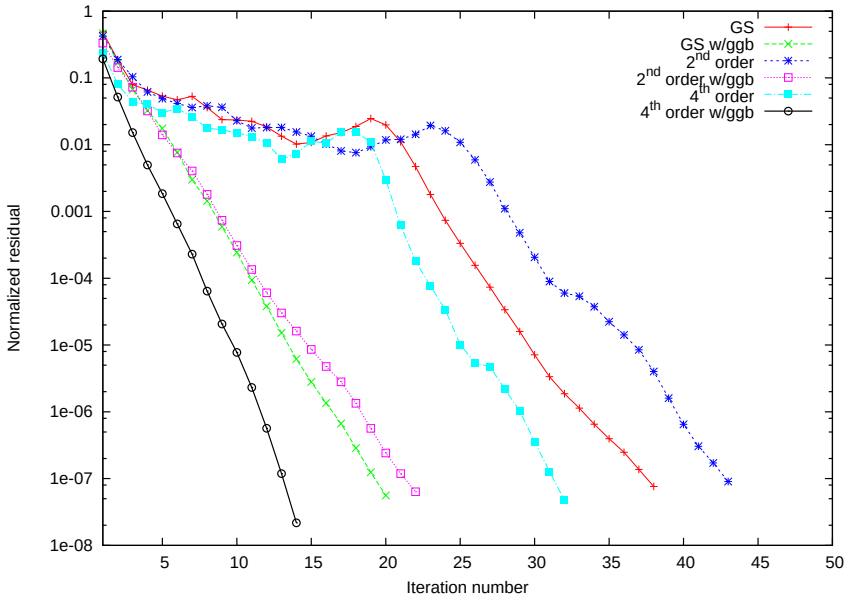

Figure 4.3: Smoother comparison of GS and MLS, Airfoil case

GGB in a way that is comparable to Gauss-Seidel. It can be observed that the convergence curve of 2nd-order polynomial smoothing accelerated by GGB is hardly distinguishable from its Gauss-Seidel counterpart, and 4th-order one clearly stands out with faster convergence on a per-iteration basis.

However, more computational work is required in each step of polynomial smoothing compared to Gauss-Seidel. For example, 2nd-order polynomial smoothing requires 2 matrix-vector products at each smoothing step, and 4th-order MLS doubles that work. Compared to Gauss-Seidel, which only requires about one matrix-vector product at each smoothing step, the serial polynomial smoother is not as efficient as Gauss-Seidel. However, we noticed that for AMG applications, all the pre-smoothing starts with zero initial guess. Taking advantage of the zero initial guess, 2nd-order polynomial smoothing will only use 3 matrix-vector products for each preconditioning iteration (one for pre-smoothing because of the initial zero guess, two for post-smoothing), while Gauss-Seidel use 1.5 matrix-vector product (0.5 for pre-smoothing because of the initial guess, one for post-smoothing. This will be explained later in Section 6.1.1). The difference is then reduced to 1.5 matrix-vector product between Gauss-Seidel and 2nd-order polynomial smoothing. We also compare Gauss-Seidel and polynomial smoothing in terms of serial computational

case	total time	per vector	total w/ GGB	per vector w/GGB
Gauss-Seidel	21.4 (sec)	0.56	14.9	0.71
2nd-order MLS	36.1	0.83	21.1	1.00
4th-order MLS	52.9	1.65	25.7	1.83

Table 4.3: Computational time for polynomial smoothing, steady state Airfoil case in serial

time shown in Table 4.3. In terms of total computational time, the increase is moderate. Moreover, when going to parallel, we eliminate the need for boundary chaotic Gauss-Seidel, which might potentially increase the number of iterations by a much larger factor, whose cost must be compared with this moderate increase in CPU time per vector for polynomial smoothing. The value of polynomial smoothing and its significance will be more clear in the later sections on parallel results.

CHAPTER 5
PARALLEL AMG

The purpose of using Algebraic Multigrid is to improve the efficiency of linear solver in *PHASTA*. However, if AMG can not be applied in parallel, or parallel AMG suffers a significant loss in efficiency, it would be less attractive. So our goal here is to design an AMG algorithm used in the Finite Element Navier-Stokes solver(*PHASTA*) that can be well parallelized and still maintains the efficiency that AMG brings to the solver. Having a parallel AMG would make the algorithm useful not only to the lab test problems in serial but also to the real world large applications that require massively parallel supercomputers.

In the following sections, we will review the AMG algorithm and discuss various approaches to parallelization. First we will review the parallel scheme associated with the Finite Element part of *PHASTA* to understand the essential data structures to be used in parallel AMG. Then we will study the issues that bring the degradation of AMG from serial to parallel. Several alternative ways to get around these issues are proposed. After that, we introduce a "Reduce serial" model to study the issues further, and to verify the methods we use for parallel AMG. These modifications and amendments to the original AMG are described in detail. At the end of the section. We will discuss the implementation of the parallel AMG algorithm with the finite element solver. And in next chapter, we will use various cases to test parallel AMG, near-perfect scaling and huge savings on number of iterations will be observed.

Firstly, in this section, the current parallel scheme used in the Finite Element solver *PHASTA* is discussed. Having this as a background knowledge enables us to further discuss the source of difficulties that arise from AMG parallelization and its integration with the Finite Element solver.

5.1 *PHASTA* in Parallel

Before explaining how we will make AMG parallel, we first review the current parallel algorithm in *PHASTA*. This is necessary for two reasons. First, the current linear solver without AMG in *PHASTA* gives the foundation of parallel data storage and communication pattern. Second, the current solver provides parallel matrix-free algorithm for PPE solve, which handles the linear algebra operations of PPE by access only to the finite element matrices $\mathbf{K}, \mathbf{G}, \mathbf{C}$. For a Finite Element solver without AMG, this scheme gives mathematically identical result to the serial, while keeping the communication to a minimum level which is a key ingredient in its demonstrated strong scalability. These are important guidelines for our parallel AMG and are foundations on which our parallel AMG should be built on. From this review we will understand what the parallel scheme is and how AMG might fit into this scheme.

5.1.1 Distributed equations over parts

The finite element solver partitions the problem by elements. That is, the partitioning process distributes the finite elements but not nodes uniquely across the processors. We will use the word part to describe one group of elements and the word partition to refer to the collection of all of the parts (e.g., all the parts together make a partition of the original/total mesh). For linear solvers, this element-based partitioning has, historically, been less desirable than distributing nodes/unknowns among processors (unknown-based partitioning). This historical bias against element-based partitioning, is that the boundary unknowns are duplicated across the parts. Across the boundary, elements on different parts need to share one or more of the same nodes. These nodes are usually duplicated in the parts which requires sharing to ensure local stiffness matrix construction by each element, making data redundancy inevitable in such a treatment. However, element-based partitioning makes finite element formation easier and scalable, and still maintains the ability to perform an iterative linear solve. In *PHASTA* element based partitioning is used, as we will discuss below.

The matrices $\mathbf{K}, \mathbf{G}, \mathbf{C}$ in (1.4) and (1.5) come from finite element discretiza-

tion and have a connectivity structure directly related to the geometric mesh. This has been discussed in Section 2.1.1. After parallel partitioning, these matrices are generated locally and no communication is needed during the construction. However these locally generated matrices are not complete on boundaries, because for each boundary node, all the elements that connect to it need to be taken into account to create the final equation in the stiffness matrices. These stiffness matrices are distributed among different parts. However, they hold a simple but important assembly relation for completeness as we discuss below.

Let us recall that the fill pattern matrix $\mathbf{B}$ that is a boolean matrix resulting from the connectivity info of the nodes, as shown in Section 2.1.1, which is now stored in parallel over parts. If we use $\mathbf{B}$ to represent any of the $\mathbf{K}, \mathbf{G}, \mathbf{C}$ matrices that are directly generated from finite element mesh, and we solve problems using n parts, $n \geq 2$, on each part we only have a subset of unknowns. Thus $\mathbf{B}$ is distributed over the parts. i.e. we have:

$$\mathbf{B}_{global} = \mathbf{B}_1 \oplus \mathbf{B}_2 \oplus \cdots \oplus \mathbf{B}_{N_p} \tag{5.1}$$

$$= \mathbb{A}_{p=1}^{N_p} \mathbf{B}_p, \tag{5.2}$$

where we use $\oplus$ to represent an assembly between two parts, $\mathbb{A}$ to indicate the assembly for all the parts, and N_p is the number of total parts. Also, the vector is distributed over the parts alike (with repetition of nodes that appear on part boundaries).

$$\mathbf{u}_{global} = \mathbf{u}_1 \oplus \mathbf{u}_2 \oplus \cdots \oplus \mathbf{u}_{N_p} \tag{5.3}$$

$$= \tilde{\mathbb{A}}_{p=1}^{N_p} \mathbf{u}_p \tag{5.4}$$

5.1.2 Basic communication and operations

In (5.4), the "assembly" operation $\tilde{\mathbb{A}}$ for vectors has already been implemented in *PHASTA*. The details of this operation can be found in [48]. Basically, we pick out the entries in the vector corresponding to the shared partition boundary nodes

between two neighbouring parts. Then we communicate them by either arithmetic summation or replication, to make the vector complete. We call an entry in some vector or matrix to be "complete", if it has the same value as the global problem, i.e. as if we run a serial case. So, for a boundary node which has values generated over two or more parts, "complete" means a summation on the corresponding entries over these parts.

Using the communication described above, we can make a matrix-vector **Ap**-product globally correct in parallel with matrices satisfying (5.2) but incomplete locally at each part. By saying "globally correct" we mean that we calculate $\mathbf{q} = \mathbf{Ap}$ for the data sets described in the whole problem domain and will result a same-as-serial result vector $\mathbf{q}$, although the actual $(\mathbf{q}, \mathbf{A}, \mathbf{p})$ are distributed over the parts. The procedure is:

1. We start from a set of vector $\mathbf{p}$'s that satisfies (5.4). For each two neighbouring parts, one is defined as "master" part, the other is defined as "slave" part. A node, if living on both parts, has complete numerical value on the master, and has zero numerical value on the slave. This makes a simple summation of the local norms among parts equivalent to the correct global value as in serial. This is the initial stage of our operation. We will see after a consequence of operations, we can return to this stage with resulting $\mathbf{q}$'s.

2. For each pair of neighbouring parts, we communicate $\mathbf{p}$ in these parts. Recall that $\mathbf{p}^m$ has all complete values on master part and $\mathbf{p}^s$ is has some zero values on slave part(s), Using communication, we copy the values on boundary nodes of $\mathbf{p}^m$ onto its counterpart of slave part(s), such that $\mathbf{p}^s$ has complete values on all nodes (in *PHASTA* this is performed in the routine "*commOut*").

3. On each part i, we do $\mathbf{q}_i = \mathbf{A}_i\mathbf{p}_i$ locally. No communication is occurred during this stage. Note since the $\mathbf{A}_i$'s are incomplete as they satisfy (5.2), so consequently the resulting $\mathbf{q}_i$'s will all have incomplete values both on master parts and slave parts.

4. For each pair of neighboring parts, we communicate $\mathbf{q}$, by doing a summation of partition boundary nodes of the master part and the slave part(s). This

results in a complete boundary node value. We put this value on the master part for each node: $\mathbf{q}^m = \mathbf{q}^m + \mathbf{q}^s$. The summation has been done for all the slave nodes through a series of part-part communications. Then we zero out the corresponding values on all the slave part(s) (in *PHASTA* this assembly from slave to master and subsequent zeroing of the slave is performed in the routine "*commIn*").

5. Now we have a set of vector $\mathbf{q}$'s that satisfies (5.4), and have the same property as in the initial stage. Together they represent a global matrix-vector product result. This completes one operation of matrix-vector product and can be done repeatedly.

Please keep in mind that the assembly operation means summation of the counterpart of partition boundary nodes represented on each part rather than a simple add. In other words, this assembly operation is node-to-node with a given mapping.

This **Ap**-product is the kernel of the parallel Krylov(CG,GMRES) solver for (1.4) and (1.5). Recall the CG algorithm described in Appendix A.4, we found that only matrix-vector products is required by CG solver (and so is GMRES). The **Ap**-product for the PPE must be done carefully (carrying out the actual product as a series of products). To properly handle this situation requires breaking the PPE **Ap**-product into several parts:

$$\mathbf{A} = [LHS]_{PPE} = \mathbf{G^T \hat{K}^{-1} G} + \mathbf{C} \tag{5.5}$$

for matrix-vector product:

$$\mathbf{Au} = \left(\mathbf{G^T \hat{K}^{-1} G} + \mathbf{C}\right)\mathbf{u} \tag{5.6}$$

We know that $\mathbf{G}, \mathbf{K}, \mathbf{C}$ all satisfy (5.2). So we break the matrix-vector product into:

1. Let $\mathbf{A_1} = \mathbf{G}$, $\mathbf{p_1} = \mathbf{u}$, we do **Ap**-product to get $\mathbf{q_1} = \mathbf{A_1 p_1} = \mathbf{Gu}$.

2. Let $\mathbf{A_2} = \hat{\mathbf{K}}^{-1}$, $\mathbf{p_2} = \mathbf{q_1}$, we do

   Ap-product to get $\mathbf{q_2} = \mathbf{A_2 p_2} = \hat{\mathbf{K}}^{-1}\mathbf{Gu}$.

3. Let $\mathbf{A_3} = \mathbf{G^T}$, $\mathbf{p_3} = \mathbf{q_2}$, we do

   Ap-product to get $\mathbf{q_3} = \mathbf{A_3 p_3} = \mathbf{G^T}\hat{\mathbf{K}}^{-1}\mathbf{Gu}$.

4. Let $\mathbf{A_4} = \mathbf{C}$, $\mathbf{p_4} = \mathbf{u}$, we do

   Ap-product to get $\mathbf{q_4} = \mathbf{A_4 p_4} = \mathbf{Cu}$.

   Now we have $\mathbf{q}_4$ satisfying (5.4).

5. Let $\mathbf{q} = \mathbf{q_3} + \mathbf{q_4}$. That is, arithmetic plus on local part, assemble over parts, we have the final matrix-vector product of:

   $\mathbf{q} = \left(\mathbf{G^T}\hat{\mathbf{K}}^{-1}\mathbf{G} + \mathbf{C}\right)\mathbf{u}$.

Care must also be taken with the calculation of norms. For a part boundary node, we put all the non-zero values on the master node, and all zeroes on the slave node. Thus the contribution to a global norm from the partition boundary node is a simple summation over the parts.

5.1.3 Data structure for current parallel

To perform the assembly process that was describe above, we need information about partition boundary nodes. In *PHASTA*, this mapping information is stored in the *ilwork* array.

For each part, the communication job is divided into several "tasks". Each task handles the communication with another distinct part. For each task, *ilwork* stores information about whether the current node is a master/slave, the part-id of its neighbour, and the length of part boundary nodes involved in this task. Then the boundary nodes are listed by an order that is physically the same as its neighbour. This enables access to these partition boundary nodes by the local numbering while keeping the global order fixed.

5.2 Issues of AMG parallelization

In general, parallelization brings issues to the existing serial algorithm in two aspects: Memory (parallel storage) and CPU (parallel execution). This is also true for AMG.

The problems caused by storage results in a difficulty in the matrix preparation. First, parallel finite element solvers store data in different parts. This means that one part does not need to be aware of the full global matrix entries, but only the sub-matrix it has stored locally. However, from the discussion above we have seen that serial AMG requires an analysis of all the coefficients of the global matrix. This conflict, that AMG needs the whole matrix while local part only has a portion of the matrix, must be resolved in the parallel AMG solver.

Second, the AMG algorithm in *PHASTA* solves the Pressure Poisson Equation (PPE). This PPE is defined mathematically by matrix-matrix multiplications as seen in (1.5). Without AMG, the iterative solve does not require an explicit form of the PPE even in serial as we just discussed. Instead, we store the matrices $(\mathbf{K}, \mathbf{G}, \mathbf{C})$ that directly come from the Finite Element analysis. These are only multiplicands of the final PPE form that the AMG algorithm requires. So, locally on a part, we don't even have the globally complete portion of the matrix of interest, but only its multiplicands. This treatment is sufficient for the Krylov iterative solver as long as we provide global correct matrix-vector product. In the original parallel scheme for our finite element solver, a "globally correct" vector was the only concern, rather than a "globally correct" matrix. This approach is not sufficient for AMG, which requires access to matrices. We have encountered this challenge because we would like to develop an AMG that does not significantly alter the parallel data structures which have proven highly scalable with Krylov methods. To address this challenge, we still need to design a parallel AMG algorithm with only this local, incomplete information, to perform well without extra storage of significant global data.

Third, in the essential smoothing part in current serial AMG, Gauss-Seidel requires a correct ordering on execution. Parallelization makes Gauss-Seidel algorithm difficult to have a correct ordering (more detailed discussion in Section 5.2.4.) For this reason, as discussed before, we have prepared polynomial smoothing as an

alternative to Gauss-Seidel. In the following subsections, we will discuss each of the problems in detail.

5.2.1 Matrix preparation

AMG requires the complete matrix to perform the coefficient analysis, finding out strong correlation, computing the weights of interpolation, etc. When we go to parallel, this complete matrix, namely PPE, does not exist. Instead, only distributed matrices exist among parts, and in the work described above, only $\mathbf{q} = \mathbf{Ap}$ was correct. In other words, prior to AMG, *PHASTA* was only concerned about getting a "globally correct" vector as opposed to a "globally correct" matrix.

Furthermore, recall that our matrix of interest is Pressure Poisson Equation, which is the result of matrix-matrix multiplication of several matrices that is distributed over parts. Since it is only required that the assembly of these matrices is complete, as shown in (5.2), we don't have a globally correct *on part* multiplicand matrix to form what we need: the global correct PPE. For unknowns that live on the boundary between two parts, the corresponding entries in $\mathbf{K}, \mathbf{G}, \mathbf{C}$ matrices would have two (or more) partial values on two (or more) parts coming from finite element assembly on each part. The summation of these two (or more) partial values is globally correct, but each local part is incomplete. However here we need these two entries to be globally correct to perform multiplication. One way to achieve this goal would be to introduce additional communication patterns for matrices, not vectors, to enable the assembly of an on-part globally correct matrix instead of only vector.

Even if we did this, however, there are also missing matrix entries that are connected between non-neighboring parts by a "bridge" part. These entries exist in a global sense because the PPE left-hand-side matrix $\mathbf{G^T\hat{K}^{-1}G + C}$ is such that nodes coming from two parts with one same neighboring part will produce a matrix entry connected by the $\hat{\mathbf{K}}^{-1}$ in the partition boundary. Consider such a case that node k is a boundary node between part 1 and part 2. Let node i be an interior node in part 1 and be connected to k. Let node j be an interior node in part 2, also connected to k. So we have $\mathbf{G}_{ik}, \hat{\mathbf{K}}_{kk}^{-1}, \mathbf{G}_{kj}^{T}$. As the result of a global serial matrix product, there will be an entry between node i and j, giving a PPE matrix

entry $\mathbf{A}_{ij}$. Yet i, j are interior nodes in different parts. These entries are difficult to construct locally because the local part does not store any nodes from the interior of the other part. For illustration please see Figure 5.1 and Figure 5.2. For now these entries are ignored in a global sense. This issue will be addressed in the next section.

$\mathbf{A}_{ij} = \mathbf{G}_{ik}\hat{\mathbf{K}}_{kk}^{-1}\mathbf{G}_{kj}^{T} + \cdots \neq 0$

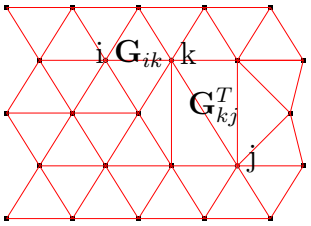

Figure 5.1: PPE construction in Serial

$\mathbf{A}_{ij}$ not available from local matrix products

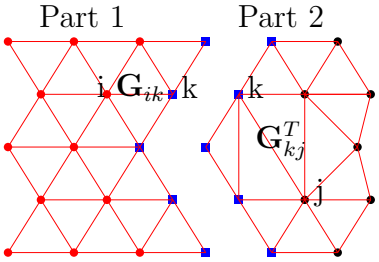

Figure 5.2: PPE construction in Parallel

5.2.2 Fine/Coarse Splitting

To perform the fine/coarse splitting in the same way as in the serial code is, in practice, difficult. The serial procedure involves a whole matrix scanning as described before. A complete clone of the serial code is not practical due to the volume of communication and the increased complexity of data structures required. However, since the parallel global matrix that AMG can see is a blocked diagonal matrix with overlap on the boundary nodes (more details in Section 5.3.1), we can approximate the C/F result in a purely local manner and treat boundary nodes differently. We may treat boundary nodes between parts as "fine" in the first level. That is, we eliminate all the boundary nodes going from the original AMG matrix to the first coarsened matrix. Their interpolation are purely local to each part. By doing this the problem is avoided as we go from the second level to the third level. This choice is a substantial deviation from the original AMG since it completely ignores the connectivity across part boundaries. A second option described in [28] improves this deficiency. Instead of setting boundary nodes as "fine" to eliminate them in the upper levels, they are set to be "coarse" and are carried up to the

coarsest level. By doing this, information is not lost going between levels. Finally, we can also treat boundary nodes as a separate group for coarsening. The best choice will be more clear in later sections.

5.2.3 Interpolation/Restriction

Another issue is the construction of interpolation/restriction operators. This is complicated because nodes on partition boundaries need to take/give numerical values from/to two (or more) parts, which is not easy with the current communication paradigm. Also the scheme for interpolation/restriction operation is associated with the way we split fine/coarse nodes. The solution to this problem requires investigating different methods of making the interpolation operator as shown in Section 1.3.2. For example, in [28], it is shown that direct interpolation, which is optimal for our problems, eliminates the need for extra information to construct interpolation operators on a scheme for which boundary nodes are kept to higher level with little communication. Investigations into more complicated alternatives are possible too [30, 31]. In the next section this problem together with parallel coarsening is pursued.

5.2.4 Smoothing

The smoothing step in AMG is the critical step to eliminate error components. Damped Jacobi smoothing while trivial to make parallel is not strong enough. Gauss-Seidel, which provides excellent results in the serial case is impractical to make exact in parallel, which has been discussed in several previous papers [30, 15, 16, 28]. Gauss-Seidel smoothing requires a fixed order of the unknowns. However since the matrix is distributed over the parts, and, with boundary nodes existing among two or more parts, the update sequence cannot be performed in parallel. Instead, on each part there is a locally complete Gauss-Seidel smoothing, in which boundary nodes are smoothed by different parts and end up with different values. While this can be corrected by communication, the overall Gauss-Seidel sequence is broken into a *Serial* Gauss-Seidel, which is not an acceptable alternative. Furthermore, we need to reverse the order of Gauss-Seidel when we do post-smoothing in AMG. This is to satisfy the symmetric nature of a preconditioner of CG. Since the order of

pre-smoothing is already out of control and "boundary chaotic"[31], it is really impractical to track the order and reverse it on post-smoothing. Alternative smoothers such as polynomial smoothing have been selected based on their ability to be made parallel. Details of this parallelization effort are considered in a later section.

5.3 Parallel AMG II: Solution and Verification

To solve or get around the issues stated in the previous section, we need to make adjustments to the process of generating the PPE matrix, the coarsening scheme and the smoothers. The goal is not to reconstruct an AMG cycle that is identical to the serial case as this appears impractical. In this section and next section, we introduce an algorithm that works close to the serial one but is easy to parallelize. Tests have shown that the altered algorithm introduces some performance degradation relative to serial but these defects, together with the sub-optimal coarsening due to the PPE's non-M matrix property, are small enough that they can corrected using the generalized global basis method (GGB).

The slightly altered parallel algorithm is simulated on one processor as a serial algorithm. We call this "reduced-serial" AMG, because it reduces a parallel AMG onto a serial run. It will make use of the data which describes the parts (e.g., which elements go each part and what each part's local node number corresponds to in a global numbering) to introduce different defects to mimic the parallel AMG behavior (relative to serial). Since both the parallel description and the serial description are available, this code is able to mimic not only the (correct) serial case and the (defective) parallel case but it can also mimic the presence of any defect in isolation (e.g., only parallel coarsening) while keeping the other parts of AMG as in (correct) serial. The goal of this approach is to find an acceptable compromise; an algorithm that maintains the good performance of serial while maintaining a straightforward parallel implementation. It also enables us to get an estimation on the upper bound of convergence rate of a practical parallel AMG. It serves as a bridge from serial to parallel.

Given the reduced-serial idea, we will test and integrate various solutions to parallel issues in the reduced-serial mode, and find an acceptable algorithm that both solves the issues and is amenable to parallelization.

5.3.1 Parallel matrix preparation

Recall the $\mathbf{B}$ matrix representing direct finite element stiffness matrices $\mathbf{K}, \mathbf{G}, \mathbf{C}$ as in (5.2). For simplicity, we use a 2-part problem to illustrate how the parallel ma-

trix preparation is achieved. First, all the unknowns are divided into three groups: part-1 interior, part-2 interior, and the boundary of part-1 and part-2. In the following discussion, they will be represented by subscript 1, 2, and b respectively. Part indices will be represented by superscript (1),(2) respectively.

From the finite element discretization, there will be an entry between two groups interacting (e.g., first subscript is associated with the equation number (row) and the second subscript is associated with solution vector dependence (column)). These two interactions come from two groups that we have just mentioned. The resulting matrix entry will be have subscripts indicating the two groups of interactions, and superscript indicating which part the matrix belongs to if necessary for clarity.

We compare different matrix forms in serial and parallel, written in groups and sub-matrix form. In the serial case for any of the direct finite element matrices $\mathbf{K}, \mathbf{G}, \mathbf{C}$ we have

$$\mathbf{B} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{1b} & 0 \\ \mathbf{B}_{b1} & \mathbf{B}_{bb} & \mathbf{B}_{b2} \\ 0 & \mathbf{B}_{2b} & \mathbf{B}_{22} \end{bmatrix} \tag{5.7}$$

In serial case shown above, there are no matrix entries between interiors of two different parts. This is a natural result since two nodes from the interior of different parts do not share an element. Thus they are not dependent on each other, leading to zero sub-matrix entries. But keep in mind this is no longer true for PPE (as was shown in Figure 5.2).

In the parallel case, for processor (1) we have

$$\mathbf{B}^{(1)} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{1b} \\ \mathbf{B}_{b1} & \mathbf{B}_{bb}^{(1)} \end{bmatrix}, \tag{5.8}$$

which can be viewed as the left-upper 2×2 sub-matrix in (5.7).
For processor (2) we have:

$$\mathbf{B}^{(2)} = \begin{bmatrix} \mathbf{B}_{bb}^{(2)} & \mathbf{B}_{b2} \\ \mathbf{B}_{2b} & \mathbf{B}_{22} \end{bmatrix}, \tag{5.9}$$

which can be viewed as the lower-right 2×2 sub-matrix in (5.7).

The superscript in term $\mathbf{B}_{bb}^{(p)}$ indicates that, for entries between two boundary nodes, the matrix value is not complete on either part. The summation of the values on two parts gives the correct global value (we did not to actually carry out the summation previously in pure Krylov solvers, see Section 5.1.2):

$$\mathbf{B}_{bb} = \mathbf{B}_{bb}^{(1)} + \mathbf{B}_{bb}^{(2)}, \tag{5.10}$$

or:

$$\begin{aligned} \mathbf{B} &= \mathbf{B}^{(1)} \oplus \mathbf{B}^{(2)} \\ &= \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{1b} & 0 \\ \mathbf{B}_{b1} & \mathbf{B}_{bb}^{(1)} + \mathbf{B}_{bb}^{(2)} & \mathbf{B}_{b2} \\ 0 & \mathbf{B}_{2b} & \mathbf{B}_{22} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{1b} & 0 \\ \mathbf{B}_{b1} & \mathbf{B}_{bb} & \mathbf{B}_{b2} \\ 0 & \mathbf{B}_{2b} & \mathbf{B}_{22} \end{bmatrix}. \end{aligned} \tag{5.11}$$

When used for preparation of (2.2), as needed to get the PPE matrix on each part, it is not correct to use $\mathbf{B}_{bb}^{(1)}$ or $\mathbf{B}_{bb}^{(2)}$. Instead, we require this to be the same as we used in the serial code, that is, $\mathbf{B}_{bb}$. To do this, we need to communicate $\mathbf{B}_{bb}^{(1)}$, and $\mathbf{B}_{bb}^{(2)}$, and assign the summation to each of them. This is not trivial because the sparsity structure of sub-matrices for boundaries are not the same across parts. However this is can be accomplished by extracting sub-matrices and performing a sparse matrix summation in addition to the communication, as discussed in D.1.

After matrix communication, (5.8) and (5.9) becomes:

$$\mathbf{B}^{(1)} = \begin{bmatrix} \mathbf{B}_{11} & \mathbf{B}_{1b} \\ \mathbf{B}_{b1} & \mathbf{B}_{bb} \end{bmatrix} \tag{5.12}$$

and

$$\mathbf{B}^{(2)} = \begin{bmatrix} \mathbf{B}_{bb} & \mathbf{B}_{b2} \\ \mathbf{B}_{2b} & \mathbf{B}_{22} \end{bmatrix}. \tag{5.13}$$

Note that this form of the matrices has lost the property that the summation/assembly of these matrices is exactly the serial matrix; a property lost in making them more useful for PPE preparation. This loss is insignificant since, when it comes to other operations such as matrix-vector product, the original matrices are used instead of the completed ones. And actually the completed matrices are used only in the setup phase for PPE preparation as temporary variables.

Now we have all the ingredients for PPE construction on a part. Still using the 2-part example, written in group form we have:

on part (1)

$$\mathbf{A}^{(1)} = \begin{bmatrix} \mathbf{G}_{11}^T & \mathbf{G}_{1b}^T \\ \mathbf{G}_{b1}^T & \mathbf{G}_{bb}^T \end{bmatrix} \begin{bmatrix} \hat{\mathbf{K}}_{11}^{-1} & 0 \\ 0 & \hat{\mathbf{K}}_{bb}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{G}_{11} & \mathbf{G}_{1b} \\ \mathbf{G}_{b1} & \mathbf{G}_{bb} \end{bmatrix} + \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{1b} \\ \mathbf{C}_{b1} & \mathbf{C}_{bb} \end{bmatrix}, \tag{5.14}$$

and on part (2)

$$\mathbf{A}^{(2)} = \begin{bmatrix} \mathbf{G}_{bb}^T & \mathbf{G}_{b2}^T \\ \mathbf{G}_{2b}^T & \mathbf{G}_{22}^T \end{bmatrix} \begin{bmatrix} \hat{\mathbf{K}}_{bb}^{-1} & 0 \\ 0 & \hat{\mathbf{K}}_{22}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{G}_{bb} & \mathbf{G}_{b2} \\ \mathbf{G}_{2b} & \mathbf{G}_{22} \end{bmatrix} + \begin{bmatrix} \mathbf{C}_{bb} & \mathbf{C}_{b2} \\ \mathbf{C}_{2b} & \mathbf{C}_{22} \end{bmatrix}. \tag{5.15}$$

Notice in the expression of the multiplicand sub-matrices, there are no superscripts (1) or (2). This is to make clear that each of these sub-matrices has structure

and values identical to its corresponding serial counterpart.

We also give our expression of the PPE in serial:

$$\mathbf{A} = \begin{bmatrix} \mathbf{G}_{11}^T & \mathbf{G}_{1b}^T & 0 \\ \mathbf{G}_{b1}^T & \mathbf{G}_{bb}^T & \mathbf{G}_{b2}^T \\ 0 & \mathbf{G}_{2b}^T & \mathbf{G}_{22}^T \end{bmatrix} \begin{bmatrix} \hat{\mathbf{K}}_{11}^{-1} & 0 & 0 \\ 0 & \hat{\mathbf{K}}_{bb}^{-1} & 0 \\ 0 & 0 & \hat{\mathbf{K}}_{22}^{-1} \end{bmatrix} \begin{bmatrix} \mathbf{G}_{11} & \mathbf{G}_{1b} & 0 \\ \mathbf{G}_{b1} & \mathbf{G}_{bb} & \mathbf{G}_{b2} \\ 0 & \mathbf{G}_{2b} & \mathbf{G}_{22} \end{bmatrix} + \begin{bmatrix} \mathbf{C}_{11} & \mathbf{C}_{1b} & 0 \\ \mathbf{C}_{b1} & \mathbf{C}_{bb} & \mathbf{C}_{b2} \\ 0 & \mathbf{C}_{2b} & \mathbf{C}_{22} \end{bmatrix} \tag{5.16}$$

Next, we expand and compare these three matrices coming from parallel and serial. For parallel:

on part (1)

$$\begin{aligned} \mathbf{A}^{(1)} &= \begin{bmatrix} \mathbf{G}_{11}^T\hat{\mathbf{K}}_{11}^{-1}\mathbf{G}_{11} + \mathbf{G}_{1b}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{b1} + \mathbf{C}_{11} & \mathbf{G}_{11}^T\hat{\mathbf{K}}_{11}^{-1}\mathbf{G}_{1b} + \mathbf{G}_{1b}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{bb} + \mathbf{C}_{1b} \\ \mathbf{G}_{b1}^T\hat{\mathbf{K}}_{11}^{-1}\mathbf{G}_{11} + \mathbf{G}_{bb}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{b1} + \mathbf{C}_{b1} & \mathbf{G}_{b1}^T\hat{\mathbf{K}}_{11}^{-1}\mathbf{G}_{1b} + \mathbf{G}_{bb}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{bb} + \mathbf{C}_{bb} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{A}_{11}^{(1)} & \mathbf{A}_{1b}^{(1)} \\ \mathbf{A}_{b1}^{(1)} & \mathbf{A}_{bb}^{(1)} \end{bmatrix}, \end{aligned} \tag{5.17}$$

and on part (2)

$$\begin{aligned} \mathbf{A}^{(2)} &= \begin{bmatrix} \mathbf{G}_{bb}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{bb} + \mathbf{G}_{b2}^T\hat{\mathbf{K}}_{22}^{-1}\mathbf{G}_{2b} + \mathbf{C}_{bb} & \mathbf{G}_{bb}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{b2} + \mathbf{G}_{b2}^T\hat{\mathbf{K}}_{22}^{-1}\mathbf{G}_{22} + \mathbf{C}_{b2} \\ \mathbf{G}_{2b}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{bb} + \mathbf{G}_{22}^T\hat{\mathbf{K}}_{22}^{-1}\mathbf{G}_{2b} + \mathbf{C}_{2b} & \mathbf{G}_{2b}^T\hat{\mathbf{K}}_{bb}^{-1}\mathbf{G}_{b2} + \mathbf{G}_{22}^T\hat{\mathbf{K}}_{22}^{-1}\mathbf{G}_{22} + \mathbf{C}_{22} \end{bmatrix} \\ &= \begin{bmatrix} \mathbf{A}_{bb}^{(2)} & \mathbf{A}_{b2}^{(2)} \\ \mathbf{A}_{2b}^{(2)} & \mathbf{A}_{22}^{(2)} \end{bmatrix}. \end{aligned} \tag{5.18}$$

For the serial case, we do an expansion and replace some of the entries by the parallel result where they are the same:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11}^{(1)} & \mathbf{A}_{1b}^{(1)} & \mathbf{G}_{1b}^T \hat{\mathbf{K}}_{bb}^{-1} \mathbf{G}_{b2} \\ \mathbf{A}_{b1}^{(1)} & \mathbf{A}_{bb}^{(1)} + \mathbf{A}_{bb}^{(2)} - \mathbf{G}_{bb}^T \hat{\mathbf{K}}_{bb}^{-1} \mathbf{G}_{bb} & \mathbf{A}_{b2}^{(2)} \\ \mathbf{G}_{2b}^T \hat{\mathbf{K}}_{bb}^{-1} \mathbf{G}_{b1} & \mathbf{A}_{2b}^{(2)} & \mathbf{A}_{22}^{(2)} \end{bmatrix} \tag{5.19}$$

However, the summation/assembly of the two parallel cases is:

$$\mathbf{A}^{(1)} \oplus \mathbf{A}^{(2)} = \begin{bmatrix} \mathbf{A}_{11}^{(1)} & \mathbf{A}_{1b}^{(1)} & 0 \\ \mathbf{A}_{b1}^{(1)} & \mathbf{A}_{bb}^{(1)} + \mathbf{A}_{bb}^{(2)} & \mathbf{A}_{b2}^{(2)} \\ 0 & \mathbf{A}_{2b}^{(2)} & \mathbf{A}_{22}^{(2)} \end{bmatrix}, \tag{5.20}$$

Note that, completing the boundary-boundary terms has caused them to be counted twice which accounts for the discrepancy in the 2-2 sub-matrix between (5.20) and (5.19). To fix the sub-matrix for boundary-boundary entries $\mathbf{A}_{bb}$, we can simply carry out the $\mathbf{G}_{bb}^T \hat{\mathbf{K}}_{bb}^{-1} \mathbf{G}_{bb}$ calculation on only one of the parts (instead of each part). The result is a modified $\mathbf{A}_{bb}^{(2)}$ but we will still use the same symbol. Now other than $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$, which do not exist on parallel case, we have the summation/assembly property, not only for matrices that come from Finite Element discretization, but also for their products used in construction of the PPE.

We put the summation of the two boundary matrices in two parts into each of them:

$$\mathbf{A}_{bb}^{(1)} \Longleftarrow \mathbf{A}_{bb}^{(1)} + \mathbf{A}_{bb}^{(2)}, \quad \mathbf{A}_{bb}^{(2)} \Longleftarrow \mathbf{A}_{bb}^{(1)} + \mathbf{A}_{bb}^{(2)} \tag{5.21}$$

This treatment synchronizes the boundary matrices for the ease in later coarsening operation. We can still maintain a correct **Ap**-product as noted above by carrying out the complete product with these boundary-boundary matrices on only one of the part and neglecting it on all others.

In summary, after the boundary sub-matrix adjustments described above, we have the PPE in parallel form:

in parallel, for part (1)

$$\mathbf{A}^{(1)} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{1b} \\ \mathbf{A}_{b1} & \mathbf{A}_{bb} \end{bmatrix}, \tag{5.22}$$

and for part (2)

$$\mathbf{A}^{(2)} = \begin{bmatrix} \mathbf{A}_{bb} & \mathbf{A}_{b2} \\ \mathbf{A}_{2b} & \mathbf{A}_{22} \end{bmatrix}. \tag{5.23}$$

Notice this is the result after parallel matrix preparation we just discussed so we can omit the superscript for sub-matrices since they are consistent with their counterpart in serial case.

This is equivalent to an altered serial form:

$$\tilde{\mathbf{A}} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{1b} & 0 \\ \mathbf{A}_{b1} & \mathbf{A}_{bb} & \mathbf{A}_{b2} \\ 0 & \mathbf{A}_{2b} & \mathbf{A}_{22} \end{bmatrix} \tag{5.24}$$

$$= \mathbf{A}^{(1)} \oplus \mathbf{A}^{(2)}, \tag{5.25}$$

which is different from the original serial form:

$$\mathbf{A} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{1b} & \mathbf{A}_{12} \\ \mathbf{A}_{b1} & \mathbf{A}_{bb} & \mathbf{A}_{b2} \\ \mathbf{A}_{21} & \mathbf{A}_{2b} & \mathbf{A}_{22} \end{bmatrix}. \tag{5.26}$$

Next, we will design an algorithm that minimizes the impact of losing of $\mathbf{A}_{12}$ and $\mathbf{A}_{21}$. We will see that, for coarsening and interpolation, and smoothing at high levels, they can be ignored. For smoothing at the first level, the actual PPE is important but can be recovered.

5.3.2 Parallel Coarsening and Interpolation

From the previous section concerning parallel matrix preparation, substantial effort was undertaken to ensure that each part only holds a portion of the global matrix. A complete scanning of global matrix as is done in serial is not scalable to very large processor counts. Instead, we pursue a parallel coarsening and interpolation algorithm that treats interior and boundary nodes separately as suggested by [28].

Recall the mathematical meaning of coarsening and interpolation, which is the core of classical AMG. A set of unknowns are divided into two categories: fine nodes and coarse nodes. Starting with a vector with size equal to the number of degrees of freedom, part of its entries are associated with fine nodes and part of its entries are associated with coarse nodes. The numerical value on the fine nodes are interpolated/extrapolated by the values of those with coarse nodes. The interpolation operator gives the relation and weights of the coarse-to-fine interpolation. In AMG, most of the fine nodes are connected with at least one coarse node and get values from these coarse nodes, as shown in Figure 5.3 for the serial case.

In parallel, it is not easy for a fine node to get interpolation values from coarse nodes in other parts. Furthermore, since we duplicate boundary nodes in two neighboring parts, the interpolation operation must be consistent for boundary nodes between two neighbors.

The solution to these issues is that we coarsen interior nodes and boundary nodes separately for each part. Thus, in the interior of a part, the fine nodes only get interpolation values from coarse nodes that are also from the same interior region. Since this is purely local on a part, no communication is needed. On the boundary of two parts, the coarsening and interpolation operations are also restricted within the same boundary nodes. Although these boundary entries are duplicated across neighboring parts, it is much easier to synchronize the coarsening and interpolation because only the duplicated matrices are involved. As shown in Figure 5.4, for our 2-part example partition, coarsening and interpolation are done separately for the three groups: two interior groups and one boundary group. In a case with n parts there would be n distinct interior groups and, asymptotically $O(40n)$ boundary

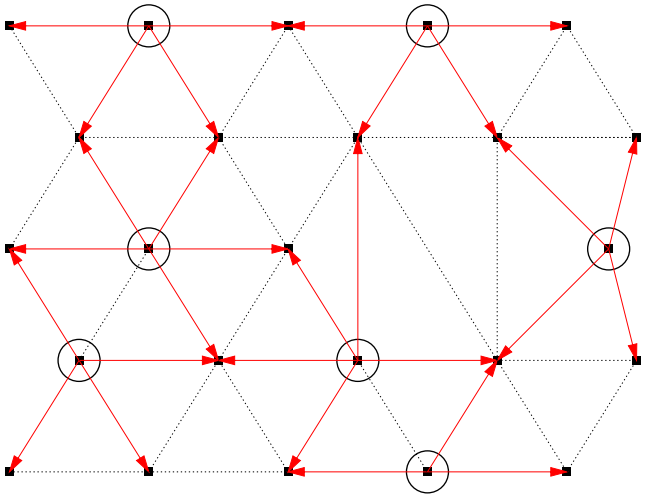

Figure 5.3: Illustration of Coarsening and interpolation in serial

(2D example for illustration). Solid dots are nodes, among which circled nodes are coarse nodes. Solid red arrows point from coarse to fine node indicating interpolation. Dashed lines between nodes indicate that no interpolation occurs within the node pair.

groups. These asymptotes are not provable for arbitrary mesh and model cases but they have been observed to hold for a wide variety of cases for n equaling 4k, 8k, 16k.

In the 2-part example, the coarsening within the groups results in an interpolation operator constructed by blocked diagonal sub-matrices:

for part (1)

$$\mathbf{Q}^{(1)} = \begin{bmatrix} \mathbf{Q}_{11} & 0 \\ 0 & \mathbf{Q}_{bb} \end{bmatrix} \tag{5.27}$$

and for part (2)

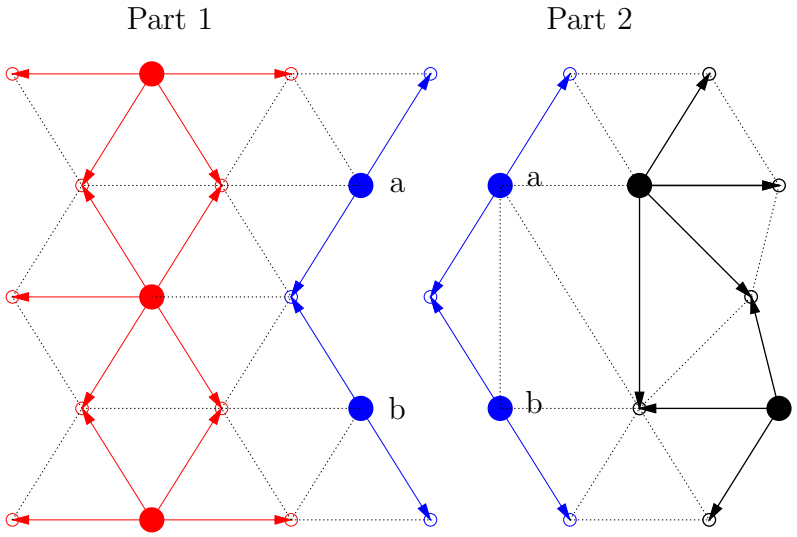

Figure 5.4: Illustration of Coarsening and interpolation in parallel

(2D example for illustration). Empty dots are fine nodes on each parts. Large solid dots are coarse nodes. Arrows are the interpolation relations. Dashed lines show no interpolation involved. Red indicates the interior of part 1. Black indicates the interior of part 2. Blue indicates the shared boundary between part 1 and 2. a, b are two coarse nodes for the boundary. The coarsening is done within one group (interior or boundary but not mixed).

$$\mathbf{Q}^{(2)} = \begin{bmatrix} \mathbf{Q}_{bb} & 0 \\ 0 & \mathbf{Q}_{22} \end{bmatrix} \tag{5.28}$$

The interpolator operator is block diagonal because the interpolation does not happen between two different groups. The sub-matrix $\mathbf{Q}_{pp}$ (where p indicates the group) comes only from the coarsening of the sub-matrix $\mathbf{A}_{pp}$. By doing this we are neglecting the information coming from cross-group matrices such as $\mathbf{A}_{1b}$. This choice, made for parallel efficiency, is anticipated to yield sub-optimal performance compared to the standard C/F splitting, however, it remains to be seen whether it will impair the convergence rate by a substantial amount. As we test this out in

the next section, we will see that the set-backs caused by incomplete coarsening are minor and well worth the gain in parallel efficiency.

Note also that the blocked structure of the interpolation operator makes it much easier to construct higher order AMG matrices since the possible cross-terms are gone. Proceeding forward in the construction of higher order AMG matrices:

at level 2, for part (1)

$$\mathbf{A^{2}}^{(1)} = \begin{bmatrix} \mathbf{Q}_{11}^T\mathbf{A}_{11}\mathbf{Q}_{11} & \mathbf{Q}_{11}^T\mathbf{A}_{1b}\mathbf{Q}_{bb} \\ \mathbf{Q}_{bb}^T\mathbf{A}_{b1}\mathbf{Q}_{11} & \mathbf{Q}_{bb}^T\mathbf{A}_{bb}\mathbf{Q}_{bb} \end{bmatrix} \tag{5.29}$$

and for part (2)

$$\mathbf{A^{2}}^{(2)} = \begin{bmatrix} \mathbf{Q}_{bb}^T\mathbf{A}_{bb}\mathbf{Q}_{bb} & \mathbf{Q}_{bb}^T\mathbf{A}_{b2}\mathbf{Q}_{22} \\ \mathbf{Q}_{22}^T\mathbf{A}_{2b}\mathbf{Q}_{bb} & \mathbf{Q}_{22}^T\mathbf{A}_{22}\mathbf{Q}_{22} \end{bmatrix}. \tag{5.30}$$

If we do the same coarsening in the serial case and coarsen the modified PPE in (5.24), we get:

$$\tilde{\mathbf{A}}^{\mathbf{2}} = \begin{bmatrix} \mathbf{Q}_{11}^T\mathbf{A}_{11}\mathbf{Q}_{11} & \mathbf{Q}_{11}^T\mathbf{A}_{1b}\mathbf{Q}_{bb} & 0 \\ \mathbf{Q}_{bb}^T\mathbf{A}_{b1}\mathbf{Q}_{11} & \mathbf{Q}_{bb}^T\mathbf{A}_{bb}\mathbf{Q}_{bb} & \mathbf{Q}_{bb}^T\mathbf{A}_{b2}\mathbf{Q}_{22} \\ 0 & \mathbf{Q}_{22}^T\mathbf{A}_{2b}\mathbf{Q}_{bb} & \mathbf{Q}_{22}^T\mathbf{A}_{22}\mathbf{Q}_{22} \end{bmatrix} \tag{5.31}$$

We have already seen earlier in the matrix preparation that we have the desired summation-complete property for PPE in (5.25). By introducing a coarsening scheme based on grouped unknowns, we can maintain the similar summation-complete property for higher levels of AMG matrices too. For higher level coarse matrices, the summation/assembly of the parallel matrices is the same as the serial matrix.

$$\tilde{\mathbf{A}}^{\mathbf{l}} = \mathbf{A}^{\mathbf{l}^{(1)}} \oplus \mathbf{A}^{\mathbf{l}^{(2)}} \tag{5.32}$$

This application of one or more of the required simplifications of the parallel algorithm onto the serial case (e.g. using (5.31) in a serial run) will be referred to later as a "reduced-serial" case. To summarize, in the reduced-serial case we have the PPE matrix (still using the 2-part example)

$$\tilde{\mathbf{A}} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{1b} & 0 \\ \mathbf{A}_{b1} & \mathbf{A}_{bb} & \mathbf{A}_{b2} \\ 0 & \mathbf{A}_{2b} & \mathbf{A}_{22} \end{bmatrix} \tag{5.33}$$

and interpolation operator given by

$$\mathbf{Q} = \begin{bmatrix} \mathbf{Q}_{11} & 0 & 0 \\ 0 & \mathbf{Q}_{bb} & 0 \\ 0 & 0 & \mathbf{Q}_{22} \end{bmatrix}. \tag{5.34}$$

5.3.3 Verification: The Reduced Serial Study

We have already designed an alternative way to prepare the modified PPE matrix, as well as the coarsening and interpolation. These alternatives were designed to suite the distributed matrix data structure of our flow solver and to be easy to parallelize. The polynomial smoother and GGB are trivial to parallelize due to their computational kernel being a matrix-vector product. However, before we implement the algorithm in parallel and design its detailed communication, we want to verify and study its performance in serial since some differences with the serial algorithm were required during the previously described parallel algorithm design. Specifically, we want to see the impact of each modification to the serial algorithm. This can most efficiently be studied using a "reduced-serial" version of AMG, This reduced-serial version of AMG takes the parallel partitioning data and simulates parallel

AMG in a serial run. The reduced-serial case gives us the opportunity to study the defects of our alternative algorithm separately which can provide us with a platform to explore the upper bound on the real parallel performance.

For the reduced serial case, first we prepare the partitioning data for different numbers of processors. These data are stored in external files. The reduced serial algorithm reads in these files and thus has the ability to simulate the behavior as in parallel for a certain number of parts. Having this information, it is possible to construct matrices and interpolation operators that are the same as in the parallel case.

Since Gauss-Seidel has parallel issues, we use second order polynomial smoothing as the smoother. Also, GGB is applied in some cases with ten eigenvalues and eigenvectors. The reduced-serial case is tested on the first step of the airfoil case. We use parallel partitioning information up to 8 processors for this test.

We have chosen these three separate cases to isolate issues that have the most impact on the parallelization. The different setups and results are shown in Table 5.1 where each of these three cases is compared with and without using GGB.

The performance of the first two rows in Table 5.1 is poor. In the first two rows we use both coarsening from parallel and smoothing from incomplete matrices at all AMG levels. We see a large increase in number of iterations from serial to 8 processors. Even with GGB the performance degrades suggesting that the AMG V-cycle has more stiff modes as we increase the number of processors. This degradation could be caused by two reasons; either from the defective matrix preparation or the alternative coarsening algorithm.

To distinguish which of these caused this degradation, a second test (rows three and four in the table) without the defective matrix preparation but only the alternative coarsening algorithm (e.g., we changed the reduced-serial case such that the coarsening followed the parallel approach (flawed) but the matrix preparation and smoothing used the serial algorithms (flawless)). As shown in the next two rows, the results are much improved and are hardly distinguishable from the serial one. This suggests that the coarsening plays a minor rule in the degradation while corrupted smoothing is a more important factor.

Setup	use GGB?	1(serial)	2	4	8
Block-diagonal coarsening, Smoothing on incomplete matrices at all levels	no	43	52	55	58
	yes	22	34	43	59
Block-diagonal coarsening, Smoothing on complete matrices at all levels	no	43	44	45	45
	yes	22	21	21	22
Block-diagonal coarsening, Smoothing on complete matrix at first level (PPE), on incomplete matrices at higher levels	no	43	**44**	**45**	**46**
	yes	22	**21**	**21**	**22**

Table 5.1: Reduced-serial study of parallel AMG on the airfoil case converged to 10^{-7}.

In the last two rows, we tested the algorithm closest to what we will do in the actual parallel case: parallel-impacted coarsening and higher level matrix smoothing, but complete matrix smoothing on the finest level. As noted earlier, using the repeated matrix-vector product form, this algorithm is actually possible to execute in parallel. From the results in the table we observe almost no degradation from the serial version to 8 processors. So we conclude that this algorithm is not only easy to parallelize, but also maintains the performance of the serial version. In the next section, we show results obtained from a parallel implementation of this approach.

5.4 Parallel AMG III: Implementation Remarks

Given the parallel algorithm proposed and verified in the previous section. We can now construct the communication and detailed implementation in parallel. Most of the communication implementation can be found in Appendix D. With the correct parallel matrix-vector product and vector communication well integrated, here we discuss other remarks on coarsening, GGB, and smoothing.

5.4.1 Parallel Coarsening

In addition to the parallel coarsening algorithm that we have discussed in the previous Section 5.3.2, there are two minor notes as described below.

First, boundary unknowns are coarsened separately from the boundary-bound -ary submatrix. Although the submatrices for a pair of neighbouring parts should be the same, they still can result in different C/F splitting due to different local numbering. So coarsening is only done for on one part for the group, and the master part broadcasts the result to slave parts to keep the coarsening synchronized.

Second, different parts need to stop coarsening at the same level to ensure that AMG ends up with the same number of (multi)levels on each part. Since coarsenings are done locally, the stop point needs to be synchronized. For the boundary group, we must end up with same level coarsening with same number of coarse nodes. From the serial analysis we found that the convergence rate is insensitive to the maximum levels used (Section 2.3.1). Therefore, our coarsening setup was designed to coarsen to a level that no further coarsening can be made. This is either maximum level allowed, which is the same global number across the parts, or to a very small group of coarse nodes. Since the system is small enough, we can use smoothing on the coarsest solve since the overall convergence is insensitive to this choice too (Section 2.3.6). So we end up with a highly robust choice of coarsening with no communication of controlling parameters.

5.4.2 Parallel GGB

For a parallel GGB (Section 3), some extra, trivial work is involved in the implementation.

First, in the eigenmodes calculation, the tool *PARPACK* [45] has already been made parallel but it expects a global vector to be separated in different parts without a duplicated boundary. However we do have a duplicated boundary in *PHASTA*, so a mapping is created in the GGB setup phase to change a *PHASTA* vector into and from a GGB vector. Then, the *PHASTA* vector is put into a AMG V-cycle for the eigenvalue analysis.

Second, after eigenvectors are calculated, the smaller system with problematic modes is generated, which usually only consists of about a dozen modes which leads to a very small matrix. This matrix is so small that it is reasonable to broadcast it to all the parts (once in the setup phase) which makes the solution phase faster.

To perform GGB's G-cycle, we use the GGB restriction vector to restrict error modes:

$$\mathbf{e}^H = {\mathbb{Q}_\mathbf{f}}^T \mathbf{e}^h, \tag{5.35}$$

where $\mathbb{Q}_\mathbf{f} = [\nu_1 \ \cdots \ \nu_k]$ are the span of problematic eigenvectors. In parallel, these eigenvectors are truncated. Their segments are stored in different parts locally and have the same size as the local error vector. The communication and norm calculation of these vectors follow the same pattern as the first level vectors.

For a local part p, the mathematical forms are:

$$\mathbf{e}^{\mathbf{H}(p)} = [\nu_1^{(p)} \ \cdots \ \nu_k^{(p)}]^T \mathbf{e}^{\mathbf{h}(p)} \tag{5.36}$$

Then $\mathbf{e}^{\mathbf{H}(p)}$ is used as the RHS to solve problematic small matrix and prolongated back similarly. These are all local operations since the eigenmodes are fixed from the setup.

5.4.3 Gauss-Seidel with parallel

From the discussion above (Section 4) we know that Gauss-Seidel smoother is intrinsically difficult to parallelize. That is the main reason that we use polynomial

smoothing for parallel AMG. We want to explore if there is a possibility to re-use Gauss-Seidel with special boundary treatment.

As we have discussed in Section 5.2.1 and Section 5.3.1, the nodes from FEM mesh can be divided into interior and boundary groups. For PPE, a matrix-matrix multiplication introduced distance-two neighbours between parts. This operation has corrupted the original interior group and made part of it sensitive to boundary and neighbouring values. From the previous discussions in Section 5.2.1, we see that all the matrix entries that have been effected share a boundary neighbour with another part (i.e. The affected interior nodes are direct neighbours with the boundary nodes). Other than these, all the interior nodes left, are not effected. The resulting PPE submatrices generated from these unaffected matrix nodes, even in parallel with matrix multiplications, are identical to their serial counterparts.

This property suggests that we can extend our boundary group to one neighbour further. By doing that, the extended boundary group contains all nodes that are sensitive to PPE parallelization, and the shrunken interior group contains all nodes that remains globally complete. The shrunken interior group, since it is complete for PPE, can be smoothed by traditional Gauss-Seidel, leaving the extended boundary with a different treatment.

The desired way to smooth the extended boundary group is to use second order polynomial smoothing. However, a complete matrix-vector product is required even for the boundary nodes, if we want to perform the smoothing correctly. Computational cost wise, this is no saving compared to pure second-order polynomial smoothing, even with shrunken-interior smoothed by Gauss-Seidel. So this option, though feasible, was firstly eliminated for efficiency considerations.

Another way to smooth the extended boundary group is to use Jacobi smoothing, however as the analysis in Section 3.3 showed, this smoother is not strong enough for our problem. An alternative is damped Jacobi. With polynomial smoothing in mind, we can use it to setup our damping factor α_0 as suggested in Section 4.1, to make our damped Jacobi smoother become first-order polynomial smoothing. By doing so we can increase a bit strength for the boundary smoother.

Overall, if we want to use Gauss-Seidel smoother as discussed above, we end

up with interior Gauss-Seidel smoother, (which is a bit stronger than 2nd order MLS), and boundary first order polynomial smoother, (which is much weaker than 2nd order MLS). We will see the results in the next chapter.

CHAPTER 6
NUMERICAL RESULTS

In this chapter we present numerical results for parallel Algebraic Multigrid for PPE. First we will discuss the efficiency of AMG algorithm. Then we will validate freezing the setup stage for multiple runs for time accurate tests. Finally we will discuss the parallel performance, both in savings of iterations and CPU time scaling.

6.1 Efficiency Study

In this section, we study the real CPU timing for AMG serial algorithm. A detailed AMG profiling is provided together with our approach to the transient case. In a transient case the system and the PPE are solved for many times (typically 2 or 3 times per time step as there are 2 or 3 linearizations per step and hundreds or thousands of steps). As we are discussing serial performance, we will first document performance using the Gauss-Seidel smoother, followed by examining how frequently the setup phase needs to be carried out. This in turn will motivate a study of whether the coarsening can be carried out using only geometric information (e.g., independent of solution and thus could be carried out during pre-processing).

6.1.1 Profiling of AMG Algorithm

The performance of AMG is evaluated in terms of CPU time in the airfoil steady state case, because this case is large and reasonably complex. The timing results are shown in the following Table 6.1. The comparisons are done with both Gauss-Seidel and 2nd-order polynomial smoothing in serial first.

One obvious fact in this table is that the setup of AMG has a relatively expensive cost. Relatively speaking, its work is equivalent to 30 to 40 preconditioned iterations for setup without GGB, and this number goes up to around 140 for setup with GGB. While this large setup cost makes the method impractical when only one solve is required, this issue will be less critical when we solve a similar problem with same GGB setup as we will discuss in the next section. By doing GGB setup only

once and using the resulting G-cycle more, we are able to distribute the overhead to each step and decrease the ratio of setup over solve. When we do multiple solutions on similar systems, the cost in setup will pay off eventually. We are going to discuss this "freeze" of setup in the next subsections.

Given that setup cost is hypothesized (at this point) to be a one-time consumption, we focus on solution time only from now on. Efficiency wise, it is observed that involving GGB is usually a good choice. With a slight increase of computational cost about 25%, we obtain a reduction in number of iterations by around 50%. This reduction results in direct savings in CPU time for solution process by about 40%, as shown in row "Prec' Cycle" in Table 6.1.

Next we further break down the solution process and take a look at the cost within each iteration. Within the preconditioner, the major cost of computation is the matrix-vector (Ap) product. For non-preconditioned CG, one Ap-product is needed for each iteration. With AMG preconditioning, this number increases. Let's first consider the finest level. In pre-smoothing, even having taken advantage of zero initial guess, we have 1 Ap-product for 2nd-order polynomial smoothing (the first matrix-vector product will give zero with the initial zero guess, knowing this will save actuall looping over all the matrix entries for a full matrix-vector product), or 0.5 Ap-product for Gauss-Seidel (The updating process in Gauss-Seidel can ignore the multiplication with the lower or upper triangle of the matrix, depending on the ordering, because of the multiplicant of initial zero guess). Then we need 1 Ap-product for residual calculation to be projected to coarser level. For post-smoothing, we need 2 Ap-products for 2nd-order MLS, or 1 Ap-product for Gauss-Seidel. This makes 2.5 Ap-products for Gauss-Seidel, or 4 Ap-products for 2nd-order MLS for total smoothing. The interpolation/restriction costs much less than an Ap-product ($< 10\%$) and usually can be ignored. Also considering the computational cost in higher levels, which is proportional to the complexity of AMG at 1.3, we have the last row of Table 6.1 in agree with theoretical estimation for the columns without GGB. Considering GGB will introduce one more Ap-product, plus an overhead interpolation/restriction operations and coarse solve, The estimation would agree with column (c) and (d) in Table 6.1 too.

	Gauss-Seidel	2nd-order Polynomial smoother
Without GGB	(a)	(b)
With 10 GGB modes	(c)	(d)

	No AMG	(a)	(b)	(c)	(d)
Setup	0	15.2	27.7	140.6	170.8
Prec' Cycle	0	21.4	36.5	14.8	21.5
Solver Total	66.8	43.7	72.2	160.6	196.8
# of iter	413	38	43	20	21
cost /iter	0.162	0.75	1.03	1.00	1.24
ratio to CG	×1	×4.6	×6.4	×6.2	×7.6

Table 6.1: Time profiling of AMG in seconds, Airfoil case serial

lhsK/lhsP	A^n	I, I^T	Total	Total AMG	Total AMG ratio
294MB	178MB	10MB	482MB	188MB	39%

Table 6.2: Memory usage of Airfoil case

Using 2nd-order polynomial smoothing as an example. Ideally, each iteration with AMG will cost (1(pre-smoothing)+1(coarse grid residual calculation)+2(post-smoothing))*1.3(complexity) + 1(GGB) + 1(CG) + (overhead from interpolation/restriction of both GGB and AMG, addition and subtraction, scaling etc) = 7.2(Ap-products)+(overhead), which agrees nicely with the measured number 7.6 in the "Ratio to CG" row and last column of Table 6.1.

We also measured the memory consumption using AMG. The additional memory includes interpolation/restriction operators and coarser matrices. For airfoil steady state case with 5-level AMG and a moderate truncation number at 0.5, the measure of extra memory usage is given in Table 6.2. Note that PHASTA also uses memory for storing vectors, consequently this a modest jump to the total memory.

6.1.2 Freezing of the Setup Phases

From the previous section it is observed that CG+AMG, or CG+AMG+GGB, decreases the number of CG vectors dramatically. However, the overhead for setup is equivalent to 30 CG vectors for the AMG (only) preconditioner. If GGB is used to accelerate AMG, the setup overhead increases further by an additional

140 CG vectors with AMG preconditioning, depending on the accuracy and GGB eigenmodes required.

In reality we need to solve the PPE system every time we perform a Newton iteration. When a steady state is sought, one iteration per time step is used but many time steps may be used. For time accurate flows, two or more iterations are performed per time step and typically $O(1000)$ (or more) time steps are performed. Within each time step, the left hand side of the linear problem (in many cases) changes slowly enough [39] that we don't have to re-setup AMG's prolongation/restriction operators and coarser level matrices again. Thus the AMG setup can easily be frozen for two or more solves (often many more). Also, since GGB is based on the AMG V-cycle itself, we don't have to calculate GGB's eigenvalue/eigenvectors again. Furthermore, when transient calculations are performed, the time step is so small that the left hand side PPE matrix change very slowly. This is especially true for the PPE case since $\mathbf{G}$ is purely based upon the mesh and, for small time steps $\hat{\mathbf{K}}^{-\mathbf{1}}$ is dominated by the mass matrix which is also based purely on the mesh. In these cases, we are able to use one setup for a large number of solves. To study this effect, the airfoil case is studied with time varying boundary conditions. The simulation is started from time step 400 (after the initial transient has passed and the flow is in a "limit cycle") and ended at time step 500 with two iterations per time step. Together, this gives a 200-solve process with a single setup performed prior to the first step.

The total time for the setup and solve for CG, CG+AMG, CG+AMG+GGB are not very different. This is because the airfoil transient problem is not very difficult to solve. As shown in Figure 6.2, even for a tight tolerance of 10^{-7} only 120 CG vectors are required. AMG reduces this number to about 24. As discussed earlier, this factor of 5 matches the V-cycle to CG cost ratio resulting in comparable times. Problems like the airfoil steady state that was solved in the previous section are better suited to AMG since the iteration reduction factor was about 20 (413 to 21).

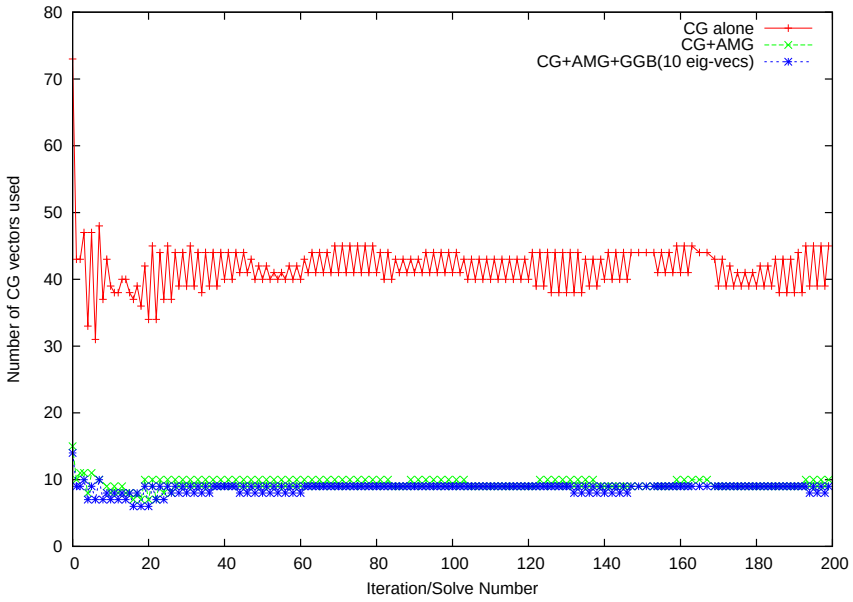

Figure 6.1: Airfoil Transient Case: one setup 200 solves, to 10^{-3}

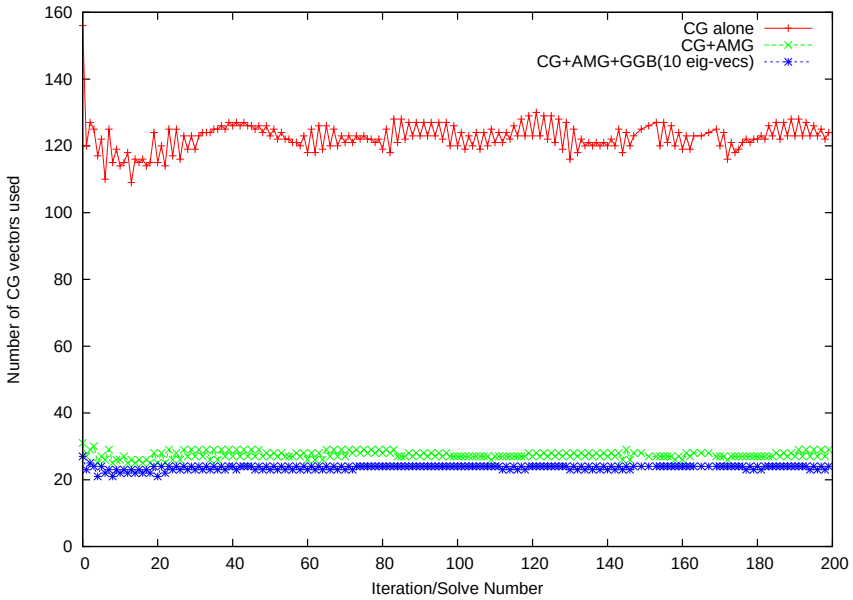

Figure 6.2: Airfoil Transient Case: 1 setup 200 solves, to 10^{-7}

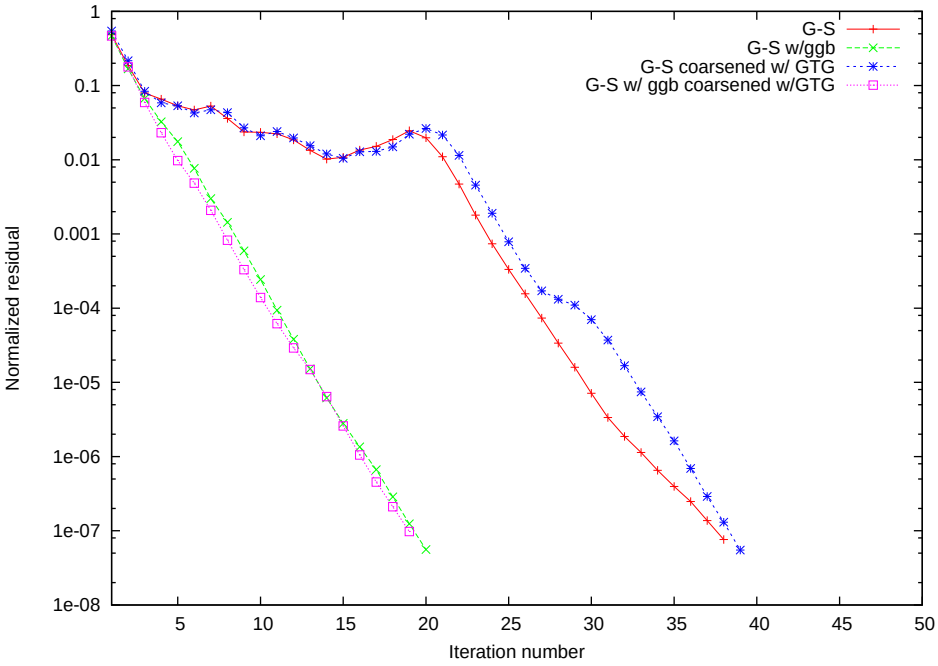

Figure 6.3: Airfoil steady state, coarsen with $\mathbf{G^T G}$, solve to 10^{-7}, Gauss-Seidel smoothing.

6.1.3 Coarsening with geometric information

Since the AMG properties of the PPE do not change much during multiple solves for one problem, we suspect that coarsening can be done based on matrices constructed with geometric information only.

Recall that,

$$\mathbf{A} = [LHS]_{PPE} = \mathbf{G^T \hat{K}^{-1} G} + \mathbf{C} \tag{6.1}$$

in which $\mathbf{K}$ and $\mathbf{C}$ are time-dependent. We pick out the non-time-dependent part $\mathbf{G}$ and use the matrix: $\mathbf{G^T G}$ to be the source of coarsening of AMG. Therefore, the coarsening is based on geometric (static if mesh is not changing) information only. To be clear, only the selection of coarse/fine nodes is determined by this matrix while all other aspects of the AMG algorithm use the correct matrix.

This idea is tested on Airfoil case for both steady state and time accurate solutions. C/F splitting is generated using $\mathbf{G^T G}$. Then the coarsening information

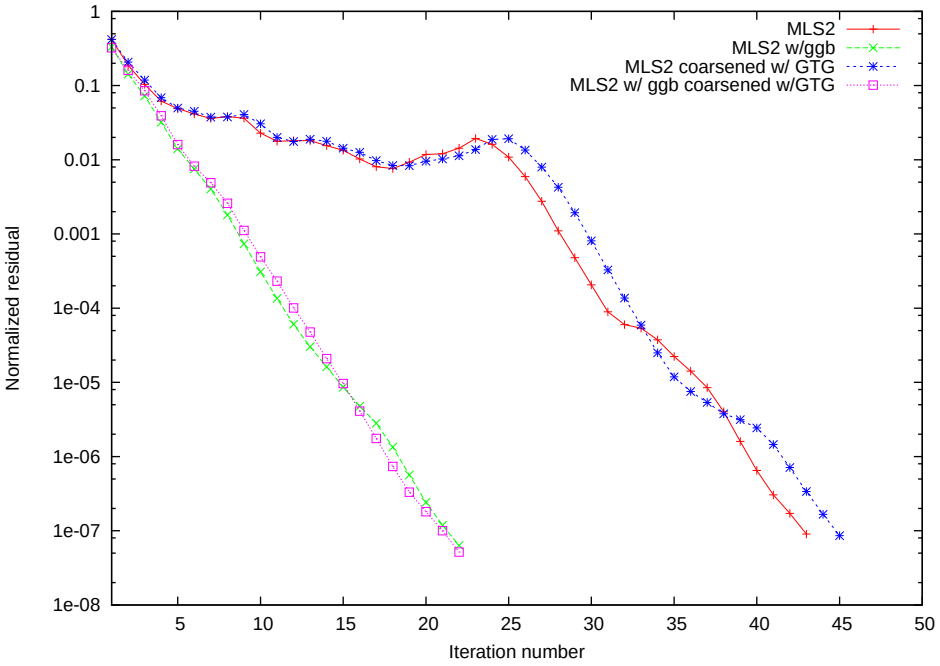

Figure 6.4: Airfoil steady state, coarsen with $\mathbf{G^TG}$, solve to 10^{-7}, 2nd order polynomial smoothing.

is stored in a external file before the actual solve. The solver then reads it and proceed with interpolator building and other parts of AMG with the given coarsening. Results are shown for the steady state case in Figure 6.3 using the Gauss-Seidel smoother and in Figure 6.4 using the MLS smoother. In each case, the degradation is very slight, if noticeable, for tolerance down to 10^{-7}. If the solver is accelerated by GGB their is still little if any effect with the change of coarsening.

For the time accurate case, similar to the test in Section 6.1.2, the solver reads in coarsening information based on $\mathbf{G^TG}$ only at the first solve. Then the interpolator and smoother are built accordingly, and kept unchanged during the whole 200 linear solves. Similar to the previous test, we solve continuity equation to 10^{-3} and momentum equation to 10^{-2}. The result turned out to be identical to the coarsening with full PPE in terms of number of iterations used as shown in Figure 6.1. However we should keep in mind that the time accurate problem is not a tough one, and we don't need very high accuracy in this problem. So only a very small number of CG iterations are needed. Yet the convergence rate remains at the high level with the coarsening given by the geometric part of PPE only.

These results suggest that indeed, the coarsening can be done solely with geometric information and held fixed for an entire run. This opens the prospect of doing the coarsening outside of the flow solver, during the pre-processing phase. This further opens the possibility of acceleration to the original linear solve, together with other tweaks, this remains an opportunity for future research.

6.2 Parallel Convergence and Scaling

We have seen serial performance of AMG (Section 2.2.1), with GGB acceleration (Section 3.3), and with polynomial smoothing (Section 4.3). In each of the configurations, savings in number of iterations are huge. In this section we give results in the same measurement but in parallel.

In this section, there are three test cases for our algorithm. First, the "airfoil case" as we have used in the previous chapters. This case is about flow over NACA 0012 airfoil, with $112,659$ unknowns. Two kinds of setup are used. We solve the hardest first PPE solve, which is referred to as "steady state case", for convergence study. And solve for 100 time steps from 400 time step with total of 200 solves as mentioned in Section 6.1.2, which is referred to as "time accurate case", for parallel scaling study. All the tests about this case, are performed on 64-node Xeon cluster.

Second, the "cavity case" with 1.6M unknowns. This case is a prototype case about flow over a duct with a cavity, with preliminary turbulent inflow boundary conditions. Again, "steady state" is tested at the first linear solve for convergence study. Then "time accurate case" is tested for 100 time steps and 3 solves each time step, for scaling study. All the tests about this case, are performed on a 512-node (1024-processor) SUR IBM BlueGene.

Third, the "aortic aneurysm case"[49] with about 20M unknowns (or 105M elements). We want to test this case for the size and availability on large processor count. In this case we solve time accurate for 100 time steps for parallel scaling study. All the tests about this case, are performed on the 16384-node (32768-processor) CCNI IBM BlueGene.

In all the three test cases, we use one AMG setup for all solves as we have validated in the previous section. Since we test a certain case with different number

# of Procs	Plain CG	AMG(MLS2)	AMG(MLS2) +GGB	AMG(ext.GS) +GGB
1(Serial)	413	43	21	20
2	413	45	21	24
4	413	44	21	24
8	413	46	22	27
16	413	48	22	27
32	413	51	23	29
64	413	54	23	35

Table 6.3: Convergence in parallel: Airfoil steady state
Vector counts for Airfoil steady state case solving to 10^{-7} with different number of processors for the first linear solve.

of processors, strong scaling is focused rather than weak scaling.

6.2.1 Convergence in Parallel

For a same case, we expect to see that the number of iterations converging to a same criteria remains at a constant level for different number of processors. Starting from the airfoil case, we test this case from serial (1 processor) to 64 processors. The results are shown in Table 6.3. Here we tested both the polynomial smoothing at (column 3 and 4) and modified Gauss-Seidel with extended boundary as we discussed in Section 5.4.3 at (column 5). It is observed that with extended boundary Gauss-Seidel, even accelerated by GGB, still suffers a lot from the increase of partitioning parts. In fact, considering CPU time, only negligible gain against polynomial smoothing was observed for small processor count (no more than eight). For larger processor counts, polynomial smoothing firmly keeps the number of iterations in the same level as serial, while extended Gauss-Seidel fails to do so. Considering that our problem might go to thousands of processors, we will only use second order polynomial smoothing for the remaining tests.

We are also interested in the total vector counts in parallel for time accurate setup for all three cases. This test provides a more general idea for multiple solves in parallel, and gives the scaling result with more samples on iteration count. The results for three cases are shown in Table 6.4. Three cases are tested within different scopes of parallelization, from serial to 16K-processors. It is observed that

# of Procs	Airfoil, 200 solves, $n \approx 100K$	Cavity, 300 solves, $n \approx 1.6M$	Aortic, 400 solves, $n \approx 20M$
1	2213 (1)	-	-
2	2104 (1.05)	-	-
4	2117 (1.04)	-	-
8	2134 (1.04)	-	-
16	2096 (1.06)	-	-
32	2102 (1.05)	-	-
64	2082 (1.06)	4150 (1)	-
128	-	4167 (0.996)	-
256	-	4247 (0.977)	-
512	-	4287 (0.968)	-
1024	-	4500 (0.922)	3153 (1)
2048	-	-	3387 (0.931)
4096	-	-	3470 (0.909)
8192	-	-	3456 (0.912)
16384	-	-	3821 (0.825)

Table 6.4: Convergence in parallel for time accurate cases
Total iteration vector counts for each case, numbers in brackets show the impact on scaling by normalization of the same case with smallest number of processors.

total vector counts remain at the same level as the parallel setup with least parts (or serial setup for Airfoil case). Also, the total number of iterations goes up at the largest processor count for each case. Considering that we are testing for strong scaling, the problem size per part decreases as number of parts goes up. At higher parallel environment we end up with large portion of boundary unknowns. Also recall that our parallel AMG algorithm is dependent on interior and boundary groups, so the performance is consequently weakened with less interior unknowns and more boundary unknowns.

With confidence given by the previous tests about AMG maintaining convergence rate as we increase the number of processors, we performed the first solve of the cavity case against non AMG, as shown in Figure 6.5, the number of iterations has significantly dropped from 641 to 25.

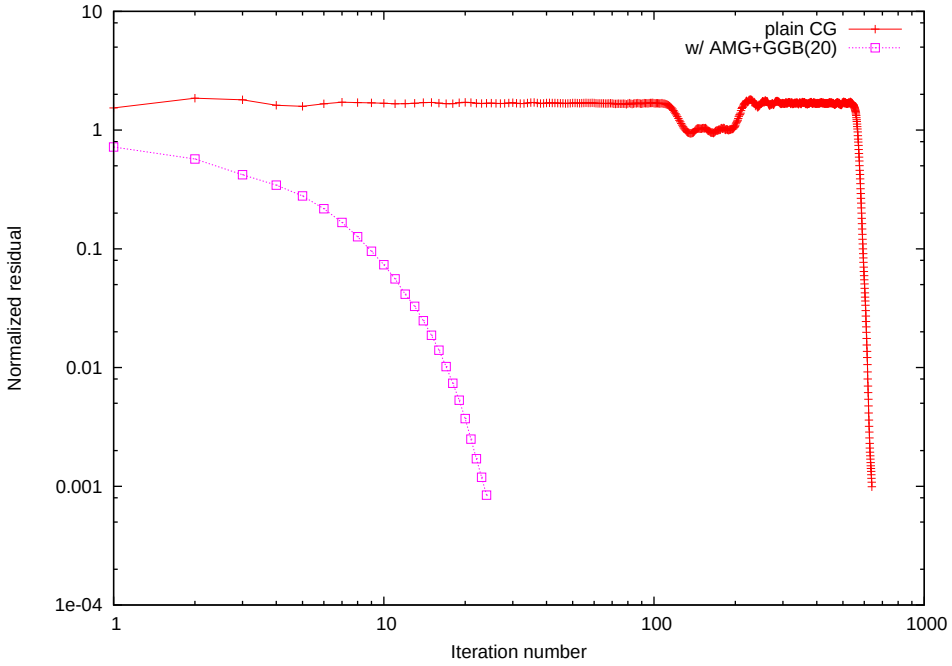

Figure 6.5: 1.6M case, First linear solve, 20 GGB modes, 2nd order polynomial smoothing. 128 Procs

6.2.2 Scaling on CPU time

The parallel AMG is tested for scaling on the three cases. The scaling number are dependent on actual machines but nevertheless reveals the scalability of the algorithm itself. Results are shown in Table 6.5, Table 6.6 and Table 6.7 for the three cases, with different size, on different computers. It is observed that with AMG as a preconditioner, the linear solve in our software package maintains its scalability at the same level. However, the scale number drops faster with AMG enabled than pure CG as expected.

This degradation has several reasons. First, we have more iteration vectors with higher counts of processors. As shown in the cavity and aortic case in Table 6.4, the number of total iterations degradates with scaling 0.922 and 0.825 respectively, which will cause at least the same amount of scaling degradation on CPU time. This is the main reason for the degradation of scaling.

Second, coarsening will introduce imbalance on coarse AMG levels. The increased imbalance of equations on higher level will further impair scaling together with the increasing density of coarser matrices. While the cost of communication is

proportional to the number of equations, the cost of computation is proportional to the number of non-zeros. Hence, the load imbalance on the number of equations, which makes some part carrying more than average number of equations, will be amplified and will result further imbalance of the number of non-zeros with sparse matrix density over than linear dependance. If we consider a simple example of a processor carrying 10% more equations, it will result 10% more non-zeros if the number of non-zeros of the matrix is linearly proportional to the number of equations, which is usually the case for the first AMG level PPE when the matrix is sparse. However, the 10% more equations will result 21% more non-zeros on that processor if the matrix is a full matrix. When we build coarse level matrices for AMG, we do have full or near full matrix on the coarser levels. This is a further degradation from the first level (the original solver). Further more, coarsening operation alone will introduce imbalance to number of equations (so the 10% in the example will likely increase too). A measure of imbalance on the cavity case with 512 processors is presented in Table 6.8, where we can see the imbalance is drastically increased with more AMG levels. Fortunately, cost on higher levels is not the major part of the overall computation, so the drastic increase of imbalance on higher levels will only slightly degrade the scaling number. In conclusion, the computation in higher levels is more "sensitive" to the load imbalance and will impair the scaling performance.

In addition to the two major reasons we just discussed, we have more communication volume because of the higher level Ap-products. Also we have more communication tasks due to the fact that each level should communicate seperatedly. More tasks bring more overhead, and communication by pieces of small blocks is less effective than communication by large data block, which we do in the finest level. The drop caused by these reasons is in addition to the drop caused by the original CG solver, reducing the scaling from 0.83 to 0.72 for the cavity flow case (last row of Table 6.6), and from 0.97 to 0.76 for the aortic case (last row of Table 6.7). From the tables, it is also clearly seen that setup of AMG usually drop faster because matrix-communication are involved. Yet with all the negatives, solver with AMG still maintains scaling number at a high level with reasonable partitioning.

# of Procs	Prec' time(sec) (scale)		Setup time(sec) (scale)		Solver time(sec) (scale)		Solver time(sec) w/o AMG (scale)		max.n /avg.n
serial	4644	-	345	-	8669	-	8537	-	-
2	2999	1	213.13	1	5894	1	4207	1	1.01
4	1443	1.04	97.63	1.09	2842	1.04	2250	0.93	1.03
8	710	1.06	55.59	0.96	1408	1.05	1049	1.003	1.03
16	344	1.09	23.05	1.16	656	1.12	512	1.03	1.03
32	183	1.02	14.52	0.92	348	1.06	269	0.98	1.03
64	127	0.74	10.46	0.64	241	0.76	180	0.73	1.05

Table 6.5: Parallel Scaling test: Airfoil, 64-node Xeon cluster

# of Procs	Prec' time(sec) (scale)		Setup time(sec) (scale)		Solver time(sec) (scale)		Solver time(sec) w/o AMG (scale)		max.n /avg.n
64	4309	1	355	1	7672	1	6099	1	1.04
128	2209	0.98	192	0.92	3963	0.97	3184	0.96	1.09
256	1184	0.91	94	0.94	2139	0.90	1611	0.95	1.05
512	667	0.81	51.7	0.86	1130	0.85	834	0.91	1.05
1024	396	0.68	28.9	0.77	664	0.72	460	0.83	1.06

Table 6.6: Parallel Scaling test: Cavity, 512-node BlueGene

# of Procs	Prec' time(sec) (scale)		Setup time(sec) (scale)		Solver time(sec) (scale)		Solver time(sec) w/o AMG (scale)		max.n /avg.n
1024	2769	1	511	1	5517	1	4627	1	1.07
2048	1481	0.93	248	1.03	2870	0.96	2445	0.95	1.07
4096	753	0.92	130	0.98	1447	0.95	1177	0.98	1.08
8192	412	0.84	98	0.65	804	0.86	597	0.97	1.09
16384	258	0.67	45	0.71	452	0.76	299	0.97	1.10

Table 6.7: Parallel Scaling test: Aortic, 16K-node BlueGene

Level	Equations				Non-zeros			
	max	min	avg	max/avg	max	min	avg	max/avg
1	4235	3829	4038	1.049	412854	376647	396658	1.041
2	1504	903	1108	1.357	159495	89884	116416	1.370
3	391	180	284	1.377	34415	14595	23860	1.44
4	205	62	100	2.05	10582	2534	4731	2.237

Table 6.8: Measure of imbalance on different AMG levels (Cavity case on 512-procs)

CHAPTER 7
SUMMARY AND FUTURE WORK

In this chapter we present possible projects for future research. Then we will give a summary of the contributions of thesis to parallel algebraic multigrid.

7.1 Future Work

7.1.1 Improvements on AMG and implementation

We have discussed the savings in number of iteration vectors used. And we have seen parallel scaling for AMG. Yet we have not extensively seen problems getting total CPU time savings with AMG provided. This is because of two reasons: First, AMG saves more iterations for tough problems with many vectors to converge; second, our test problems only have first few solves that are tough and stiff enough for AMG to solve. If we only solve the first few steps, the solution time did decrease drastically, but we can not ignore the expensive setup time in this scenario. If we solve the problem for a long time with a lot of time steps, setup cost can be neglected but our test problem quickly converge to steady state and lose the stiffness that desired by AMG. To overcome this dilemma, we propose consideration of two approaches that will be described next.

The first approach, is to apply AMG to problems that generate PPE which maintains stiffness for a longer time. For example, unsteady problem with time varying boundary conditions that keep pushing the solution of pressure equation to a tough level.

The second approach is to integrate the current AMG algorithm with possible physical properties of the specified problem. Then using these properties to determine when a pressure solve might be tough or not, we can use AMG only when the solution process is tough enough. This automated switch for AMG might be further improved by checking the residual for plain conjugate gradient. From the convergence curve we see that only when a stagnation plateau is hit did AMG save actual CPU time. There might be open questions for a conjugate gradient solver to

switch between original and preconditioned systems, but with better interface and deeper understanding of the Krylov methods, smart AMG might be a worthwhile possibility for investigation.

Our current AMG uses classical coarsening and interpolation strategy. While it fits our problem nicely, we might want to explore other approaches in smooth-aggregation area. The off-diagonal positive entries have always been a problem for classical AMG. Now with GGB this problem seems to be less important because GGB directly removed problematic modes in AMG, including problems created by inappropriate positive entries treatment. However in addition to GGB, other methods for handing this issue can be considered, like adaptive methods that calculate slow converging modes from smoother, and decide coarsening adaptively from the result. Future work should also focus on different smoothers like incomplete LU, sparse inverse, etc, to fully explore the possibility to accelerate the current AMG solve.

During our research, there have been systems from industry that we came across, and found out they are difficult to solve or to even coarsen. These problems usually arise from very high aspect ratio meshes and with tough boundary conditions to create a highly stiff PPE matrix. Investigation on these problems will require new tools and new advancements for AMG maybe from the basics. Yet these problems remain challenging and intriguing, and provide room for AMG to improve.

7.1.2 AMG for the Coupled Continuity-Momentum Equations

The effort described heretofore was focused on applying AMG to the discrete pressure Poisson equation which is used within ACUSIM's *leslib* to accelerate convergence of the coupled continuity-momentum solve required in Navier-Stokes simulation. The motivation for this focus is that, heretofore, the iterative solver stagnation has primarily manifested itself in the discrete pressure Poisson equation solve, not in the continuity-momentum solver. Put another way, at least for time accurate problems as the mesh gets more refined the computational work becomes dominated by the discrete pressure Poisson equation.

Successful completion of the previous chapters of work is expected to alleviate

this situation and, as a consequence it is entirely possible that the coupled continuity-momentum solve will become equal to or perhaps even dominate the computational cost. Once this situation occurs we are compelled to extend our research to include AMG for the coupled continuity-momentum solve as well.

Two significant new difficulties arise when shifting attention to the coupled continuity-momentum solve. The first is the loss of symmetry. Heretofore we have not done extensive research into solutions to this issue but it does appear that this problem is not insurmountable. Future work could focus on the development of AMG as a preconditioner to GMRES which will hopefully enjoy an acceleration similar to that of the CG solve shown here. We have done some preliminary studies on this system and the principle challenge appears to be in the development of a suitable coarsening operator. Specifically, if one chooses to coarsen nodes (e.g. coarsening out all 4 flow variables when leaving a node on the fine mesh) there are 4 equations to consider regarding the strength of their coupling to other nodes (which of course have 4 variables). We are aware of some work which suggests that the PPE coarsening may be the best choice for the coupled continuity-momentum but the literature is far from conclusive. Alternatively, one can look at the continuity-momentum system as simply a system of equations and coarsen equation-by-equation coupled to variable-by-variable. While it is easy to raise physical concerns (related to ones intuition/fear of difficulties arising from a loss of a conservative system at each "coarse node") it would appear to be rather simple to try in our current framework and worth future investigation.

7.1.3 AMG for other problems

7.1.3.1 Application to Scalar Transport Equations

In *PHASTA* we also solve scalar transport equations. Application of AMG on these equations are interesting and necessary. The linear systems arise from them are asymmetric. Since the coupled continuity-momentum systems is also asymmetric, considering these systems can be a stepping-stone between our current approach with AMG on Pressure Poisson solve and the coupled system solve. Research on AMG with scalar problems will enable us to isolate asymmetric issues from other

issues like the choice of the coarsening variable for the vector system or the non-linearity of the coupled continuity-momentum system.

7.1.3.2 Higher Order Basis Simulations

Though *PHASTA* is currently capable of performing higher-order simulations by making use of the higher order basis, this option has not come into practical use for very challenging problems due to the dramatic rise in cost for CG-GMRES solve in *PHASTA* for transient solutions of higher-order simulations.

Specifically, while [36] illustrated that the hierarchical basis was capable of substantially reducing the computational burden on steady simulations, the same has not been demonstrated on unsteady flows. The reduction in computational burden was due to a dramatic reduction in the number of degrees of freedom required to provide a solution of a given accuracy. It was realized by using initial conditions from lower order simulations as "initial conditions" for the next higher order simulation. This bootstrapping is straightforward for steady simulations but is not straight forward for our current CG-GMRES solve. To some extent, this "bootstrapping" is a multigrid concept and we anticipate that AMG will greatly improve our ability to obtain solutions on unsteady problems when employing the hierarchical basis. Ideally, the AMG will directly improve the solver performance through its ability to coarsen the mesh to "focus" on the lower order basis solution coefficients (which will be the stronger couplings) on increasingly coarser grids.

7.1.3.3 Problems other than Fluid Dynamics

Fluid-Dynamics problem, namely Navier-Stokes, has been a tough problem to solve for very long time. Our approach on iterative methods to solve it has been proven successful. There might be other problems which are currently expensive to solve. The techniques used in the current AMG development might be able to accelerate the solution process to these problems.

7.2 Summary

In this thesis, a parallel Algebraic Multigrid (AMG) algorithm is developed and implemented within the current parallel finite element flow solver *PHASTA* on unstructured meshes. Classical AMG and its performance on our Pressure Poisson Equation(PPE) is studied, and great savings in the number of iterations was achieved. Then General Global Basis method and its application with AMG were explored. For test problems, even better convergence was achieved with GGB. Polynomial smoothing was examined with different approach and variations. Tests on third-party library and our own implementation within *PHASTA* have been carried out to determine the best option for a parallel smoother. The structure of current finite element parallel approach is carefully studied to set base for parallel AMG. Together with careful construction of local PPE and submatrix synchronization, grouped coarsening, parallel smoother and GGB, a parallel AMG algorithm was finally proposed after exploration on reduced serial tests.

This algorithm provides Conjugate Gradient with a parallel AMG for PPE without explicitly construction of the global matrix, which has the form of global matrix-matrix product of parallel matrices. The results on savings in the number of iterations in parallel remains relatively flat with increasing number of processors, which confirms the effectiveness of this AMG preconditioner. Scaling tests on CPU time show that this algorithm scales well up to 16K nodes on IBM BlueGene. Overall we have a parallel, scalable, effective Algebraic Multigrid preconditioner for Conjugate Gradient solver on matrix-free Pressure-Poisson Equation arising from Navier-Stokes equations.

LITERATURE CITED

[1] Gene. H. Golub and Charles. F. Van Loan. Matrix computations. The Johns Hopkins Univ. Press, 1983.

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

[3] J. W. Ruge and K. Stüben. Algebraic multigrid (AMG). In S. F. McCormick, editor, *Multigrid Methods*, pages 73–130. SIAM, Philadelphia, PA, 1987.

[4] K. Stüben. An introduction to algebraic multigrid. In U. Trottenberg, C. W. Oosterlee, and A. Schüller, editors, *Multigrid*, pages 413–532. Academic Press, London, 2000. Appendix A.

[5] K. Stüben. A review of algebraic multigrid. *Journal of Computational and Applied Mathematics*, 128:281309, 2001.

[6] Fish J and Belsky V. Generalized aggregation multilevel solver. *Int. J. Numer. Meth. Engng.*, 40(23):4341–4361, 1997.

[7] P.Wesseling and C.W.Oosterlee. Geometric multigrid with applications to computational fluid dynamics. *Journal of Computational and Applied Mathematics*, 128:311–334, 2001.

[8] P.Wesseling. *An Introduction to Multigrid Methods*. Wiley, Chichester, 1992.

[9] Haim Waisman. *Multiscale Methods Based on Multigrid Principles*. PhD thesis, Rensselaer Polytechnic Institute, 2005.

[10] Chihiro Iwamura, Franco Costa, Igor Sbarski, Alan Easton, and Nian Li. An efficient algebraic multigrid preconditioned conjugate gradient solver. *Comput. Methods Appl. Mech. Engrg.*, 192:2299–2318, 2003.

[11] Fraunhofer SCAI. http://www.scai.fraunhofer.de/samg.0.html.

[12] Sandia National Lab. http://software.sandia.gov/trilinos/.

[13] M. F. Adams, H.H. Bayraktar, T.M. Keaveny, and P. Papadopoulos. Applications of algebraic multigrid to large-scale finite element analysis of whole bone micro-mechanics on the ibm sp. In *ACM/IEEE Proceedings of SC2003: High Performance Networking and Computing*, 2003.

[14] Yousef Saad and Jun Zhang. Enhanced multi-level block ilu preconditioning strategies for general sparse linear systems. *Journal of Computational and Applied Mathematics*, 130:99–118, 2001.

[15] Oliver Bröker and Marcus J. Grote. Sparse approximate inverse smoothers for geometric and algebraic multigrid. *Applied Numerical Mathematics*, 41:6180, 2002.

[16] Mark Adams, Marian Brezina, Jonathan Hu, and Ray Tuminaro. Parallel multigrid smoothing: Polynomial versus gauss-seidel. *Journal Of Computational Physics*, 188:593–610, 2003.

[17] Brezina M, Falgout R, Maclachlan S, Manteuffel T, Mccormick S, and Ruge J. Adaptive algebraic multigrid. *SIAM Journal on Scientific Computing*, 27(4):1261–1286, 2006.

[18] Wabro M. Amge - coarsening strategies and application to the oseen equations. *SIAM Journal on Scientific Computing*, 27(6):2077–2097, 2006.

[19] Bruno Koobus and Charbel Farhat. A variational multiscale method for the large eddy simulation of compressible turbulent fows on unstructured meshes-application to vortex shedding. *Comp. Meth. Appl. Mech. Engng.*, 193:1367–1383, 2004.

[20] Notay Y. Aggregation-based algebraic multilevel preconditioning. *SIAM JOURNAL ON MATRIX ANALYSIS AND APPLICATIONS*, 27(4):998–1018, 2006.

[21] Brezina M, Falgout R, MacLachlan S, Manteuffel T, McCormick S, and Ruge J. Adaptive smoothed aggregation α-sa). *SIAM Journal on Scientific Computing*, 25(6):1896–1920, 2004.

[22] JIM E. JONES and PANAYOT S. VASSILEVSKI. Amge based on element agglomeration. *SIAM Journal on Scientific Computing*, 23(1):109–133, 2001.

[23] Fish J, Qu Y, and Suvorov. A towards robust two-level methods for indefinite systems. *Int. J. Numer. Meth. Engng.*, 45(10):1433–1456, 1999.

[24] Haim Waisman, Jacob Fish, Raymond S. Tuminaro, and John Shadid. The generalized global basis (ggb) method. *Int. J. Numer. Meth. Engng.*, 61(8):1243–1269, 2004.

[25] Haim Waisman, Jacob Fish, Raymond S. Tuminaro, and John Shadid. Acceleration of the generalized global basis (ggb) method for nonlinear problems. *Journal of Computational Physics*, 210(1):274–291, 2005.

[26] M. F. Adams and J. Demmel. Parallel multigrid solver algorithms and implementations for 3D unstructured finite element problem. In *ACM/IEEE Proceedings of SC99: High Performance Networking and Computing*, Portland, Oregon, November 1999.

[27] Brezina M., Heberton C. I., Mandel J, and Vaněk P. An iterative method with convergence rate chosen a priori. Technical report, UCD/CCM Report 140, 1999.

[28] Arnold Krechel and Klaus Stüben. Parallel algebraic multigrid based on subdomain blocking. *Parallel Computing*, 27:1009–1031, 2001.

[29] P. Bastian et al. Advances in high-performance computing: Multigrid methods for partial differential equations and its applications. Technical report, University of Heidelberg, 2000.

[30] A.Anwander et al. A parallel algebraic multigrid solver for finite element method based on source localization in the human brain. Technical report, Max Planck Institute, 2001.

[31] Ray. S. Tuminaro and Charles Tong. Parallel smoothed aggregation multigrid : Aggregation strategies on massively parallel machines. Technical report, Sandia National Laboratory and Lawrence Livermore National Laboratory, 2000.

[32] Van Emden Henson and Ulrike Meier Yang. Boomeramg: A parallel algebraic multigrid solver and preconditioner. *Applied Numerical Mathematics*, 41:155–177, 2002.

[33] Kenneth E. Jansen and Chun Sun. Amg and the phasta incompressible navier- stokes solver: Pressure poisson equation. Technical report, KAPL Report, 2005.

[34] W. L. Briggs, V. E. Henson, and S. F. McCormick. *A Multigrid Tutorial*. SIAM, Philadelphia, PA, 2 edition, 2000.

[35] Thomas J. R. Hughes. *The Finite Element Method: Linear Static And Dynamic Finite Element Analysis*. Prentice Hall, Englewood Cliffs, NJ, 1987.

[36] Christian Whiting. *Stabalized Finite Element Methods For Fluid Dynamics Using A Hierarchical Basis*. PhD thesis, Rensselaer Polytechnic Institute, Troy, New York, December 1999.

[37] Christian H. Whiting, Kenneth E. Jansen, and Saikat Dey. Hierarchical basis for stabilized finite element methods for compressible flows. *International Journal for Numerical Methods in Fluids*, 1999.

[38] Kenneth E. Jansen. A stabilized finite element method for computing turbulence. *Computer methods in applied mechanics and engineering*, 174:299–317, 1999.

[39] K. E. Jansen, C. H. Whiting, and G. M. Hulbert. A generalized alpha method for integrating the filtered navier-stokes equations with a stabilized finite element method. *Scorec Report*, 10-1999, 1999.

[40] Farzin Shakib. ®Acusolve, http://www.acusim.com.

[41] Y. Saad. *Iterative Methods for Sparse Linear Systems, 1st Edition*. http://www-users.cs.umn.edu/ saad/books.html, 1996.

[42] K. Stüben. Private emails.

[43] Fish J and Qu Y. Global-basis two-level method for indefinite systems. part 1:convergence studies. *Int. J. Numer. Meth. Engng.*, 49(3):439–460, 2000.

[44] Qu Y and Fish J. Global-basis two-level method for indefinite systems. part 2:computational issues. *Int. J. Numer. Meth. Engng.*, 49(3):461–478, 2000.

[45] Rich Lehoucq et al. http://www.caam.rice.edu/software/arpack/.

[46] M.W. Gee, C.M. Siefert, J.J. Hu, R.S. Tuminaro, and M.G. Sala. ML 5.0 smoothed aggregation user's guide. Technical Report SAND2006-2649, Sandia National Laboratories, 2006.

[47] Marian Brezina. *Robust Iterative Methods on Unstructured Meshes*. PhD thesis, University of Colorado at Denver, 1997.

[48] Onkar Sahni, Kenneth E. Jansen, Christian H. Whiting, and Mark Shephard. Scalable finite element solver on massively parallel computers.

[49] Onkar Sahni. *Automated Adaptive Viscous Flow Simulations*. PhD thesis, Rensselaer Polytechnic Institute, 2007.

[50] Message Passing Interface. http://www-unix.mcs.anl.gov/mpi/.

APPENDIX A
ALGORITHMS FOR AMG

A.1 The Coarsening Algorithm

Goal: for n-unknowns in linear system $\mathbf{A_{n\times n}}$, find "fine" nodes and "coarse" nodes using Stüben's algorithm.

We pick up coarsest nodes from the largest: $\lambda_i = \left|S_i^T \cap U\right| + 2\left|S_i^T \cap F\right|$,where S_i is a set of strongly coupled neighbors for row i, and U are the undecided nodes, F are the already chosen fine nodes.

Setup strong correlated set S, and $\mathbf{S^T}$

1. Initialize array $S_{i,j} = 0$, $i, j = 1..n$. S is sparse stored as $S_{1..nnz}$, where nnz is the number of non-zeroes in matrix $\mathbf{A}$. S is used to store information on strong correlations.

2. Setup $S_{1..nnz}$

   (a) Pick out maximum negative entry in i^{th} row. $g_j^{max} = max\left(-A_{ij}\right)$, $j = 1..n, A_{ij} < 0$

   (b) $\forall A_{ij} < 0$, $.and. -A_{ij} > \epsilon \times g_i^{max}$, where $\epsilon = 0.25$ is an arbitrary constant determining the threshold for strong correlation.
   Mark: $S_{ij} = S_{ij} + 1$, $S_{ji} = S_{ji} + 2$
   The actual procedure requires looping over the matrix entries. For each A_{ij}, if $A_{ij} < \epsilon g_i^{max}$, mark $S_{ij} = S_{ij} + 1$, if $A_{ij} < \epsilon g_j^{max}$, mark $S_{ij} = S_{ij} + 2$.

   (c) After treatment for all matrix entries, $\mathbf{S} = \{(i,j)\,|S_{ij} = 1.or.S_{ij} = 3\}$, $\mathbf{S^T} = \{(i,j)\,|S_{ij} = 2.or.S_{ij} = 3\}$
   Define $\mathbf{S}_\alpha \equiv \{\mathbf{S}|i = \alpha\}$,$\mathbf{S}_\alpha^\mathbf{T} \equiv \left\{\mathbf{S^T}|i = \alpha\right\}$

Split F/C

1. (Initialize) $\forall i, i = 1..n$, $F_i = 1$ i.e. i^{th} unknown is "UNDECIDED". $\lambda_i = 0$, i.e. initial value of λ is 0

2. (Initialize) $\forall i$, $\mathbf{S}_i = \emptyset$, set this unknown i to be a "FINE" node. i.e. $F_i = 2$

3. Initialize λ values : $\forall i$, $\lambda_i = \lambda_i + F_j \left\{ j \in \mathbf{S_i^T} \right\}$

4. Find m, $\lambda_m = max \left\{ \lambda_i \right\}$

5. While $\lambda_m > 0$

   (a) set this unknown m to be a "COARSE" node. i.e. $F_m = 0$

   (b) $\forall j$, $j \in \mathbf{S_m^T}$, if j not defined, i.e. $F_j = 1$, then mark j as "FINE" node. i.e. $F_j = 2$
   $\forall k$, $k \in \mathbf{S_j}$, if k not defined, i.e. $F_k = 1$, then $\lambda_k = \lambda_k + 1$ because, for these k's, one unknown in their $\mathbf{S^T}$ became "FINE".

   (c) $\forall j$, $j \in \mathbf{S_m}$, if j not defined, i.e. $F_j = 1$, then $\lambda_j = \lambda_j - 1$ because, for these j's, one unknown in their $\mathbf{S^T}$ became "COARSE".

   (d) Find m, $\lambda_m = max \left\{ \lambda_i, F_i = 1 \right\}$, m points to the "UNDEFINED" unknown with the largest λ value.

   (e) Loop 5, until all the nodes are defined either "FINE" or "COARSE"

6. (Finalize): $\forall i$, $F_i = 1$, set this unknown i to be a "FINE" node, i.e. $F_i = 2$, this is to check whether there are anything left.

7. Calculate matrix size at second level, $n_2 = |F_i = 0|$, number of all the fine unknowns.

A.2 Direct Interpolation Algorithm

Objective: To create operators to restrict a fine system to a coarser one, and to interpolate a coarse system to a finer one.

$$\mathbf{A^H} = \mathbf{Q_h^H A^h Q_H^h}, \mathbf{e^H} = \mathbf{Q_h^H e^h}, \mathbf{e^h} = \mathbf{Q_H^h e^H} \tag{A.1}$$

Define: $\mathbf{I^c} \equiv \mathbf{Q_H^h}$, $\quad (\mathbf{Q^c})^\mathbf{T} \equiv \mathbf{Q_h^H}$, where superscript c stands for the current level of AMG.

1. Allocate memory for pointer to pointers *amg_Q(1..n).* i.e. $amg\_Q(i)$ is a pointer to a block of memory for i^{th} row in matrix $\mathbf{Q}^c$.

2. $\forall i,\ i = 1..n$

   (a) If i^{th} node is a coarse node, we simply copy the coarse node information on fine node to coarse node. That is: row i only has one entry: $\mathbf{Q^c}_i = 1$ continue the loop of i (e.g. go to 2 and increment i).

   (b) If the i^th node is a fine node, set diagonal part: $diag = A_{ii}$

   (c) we do direct interpolation

      i. $m = 0$

      ii. for current i, loop the non-zeroes of A_{ij}, set $\hat{a}_m = A_{ij}$, $\hat{N}_m = j$: $\forall A_{ij} \neq 0$,$\hat{P}_m = 1$:, for all coarse nodes with strong correlation.

      iii. *note* this $\hat{N}_m$ records the column position of each A_{ij}.

      iv. $m = m + 1$; continue loop A_{ij}.

   (d) $\alpha = \sum_{\hat{N}_j \neq 0} A_{ij} / \sum_{\hat{P}_j = 1} A_{ij}/diag$, $\forall A_{ij} > 0$
   $\beta = \sum_{\hat{N}_j \neq 0} A_{ij} / \sum_{\hat{P}_j = 1} A_{ij}/diag$, $\forall A_{ij} < 0$

   (e) m, number of non zeros in the i^{th} row.

   (f) Allocate memory for $amg\_Q\,(i, 1..m)$, $amg\_Q\_rowp\,(i, 1..m)$, $amg\_Q\_colm\,(i) = m$

   (g) set up Q

      i. loop over m, $j = \hat{N}_m$

      ii. If $\hat{a}_j < 0$ then $amg\_Q\,(i, j) = \hat{a}_j \alpha$

      iii. else $amg\_Q\,(i, j) = \hat{a}_j \beta$

      iv. $amg\_rowp\,(i, j) = j$

3. Matrix transpose
   generate $(\mathbf{Q^c})^\mathbf{T}$. Standard sparse matrix transpose algorithm

4. Now we have $\mathbf{Q^c}$ and $(\mathbf{Q^c})^T$ represented in sparse form.

A.3 AMG V-cycle

V-cycle to solve $\mathbf{A}^n u^n = f^n$.

amg_ V_ cycle $(n,\ \mathbf{A}^n,\ )$

1. IF (current level n is max(deepest) level) THEN
   Solve the system using either direct or a finite set of iterations and return u^n
   ENDIF
2. Forward smoother sweep on u^n: $u_1^n = Pre\_Smoother\,(u^n)$
3. Residual calculation : $r^n = \mathbf{A}^n u_1^n - f^n$
4. Restriction : $r^{n+1} = ((\mathbf{Q^n}))^T r^n$
5. Recursive function call : $e^{n+1} = \text{amg\_ V\_cycle}\,(n+1,\ \mathbf{A}^{n+1},\ r^{n+1})$
6. Prolongation : $e^n = \mathbf{Q^n} e^{n+1}$
7. Solution modification : $u_2^n = u_1^n - e^n$
8. Backward smoother sweep on u^n: $u_3^n = Post\_Smoother\,(u_2^n)$
9. Return solution : $u^n \Leftarrow u_3^n$

A.4 Conjugate-gradient with an AMG preconditioner:

To solve $\mathbf{Ax} = \mathbf{b}$:

$\mathbf{r} = \mathbf{b}$,$\mathbf{z} = \mathbf{b}$

$\mathbf{z} = \mathbf{V^{-1}r}$,/* *A function call to AMG library, replace* $\mathbf{z}$ *with the result coming out of AMG* */

$\mathbf{x} = 0$, $\mathbf{p} = \mathbf{z}$

while $|\mathbf{r}| >$ **tolerance**

$\mathbf{q} = \mathbf{Ap}$

$\alpha = \mathbf{r} \cdot \mathbf{z} / \mathbf{p} \cdot \mathbf{q}$

$\mathbf{x} = \mathbf{x} + \alpha\mathbf{p}$ /* *Solution is modified by vector* $\mathbf{p}$ */

$\mathbf{r} = \mathbf{r} - \alpha\mathbf{q}$ /* *The solution and the residual in CG are the sum of a set of vectors* $\mathbf{p}$,$\mathbf{q}$ */

$tmp = \mathbf{r} \cdot \mathbf{z}$

$\mathbf{z} = \mathbf{V^{-1}r}$, /* *A function call to AMG library, replace* $\mathbf{z}$ *with the result coming out of AMG* */

$\beta = \mathbf{r} \cdot \mathbf{z}/tmp$

$\mathbf{p} = \beta\mathbf{p} + \mathbf{z}$, /* *vector* $\mathbf{p}$ *is modified by vector* $\mathbf{z}$ */

APPENDIX B
TRILINOS TEST FOR POLYNOMIAL SMOOTHING

Consideration of the high expense when apply post- polynomial smoothing as in (4.3) motivated the study of two different smoother combinations: The first configuration is what was originally implemented in *Trilinos* (details later); in the second configuration we constructed only the pre-smoother using (4.2) and used it for both pre- and post- smoothing.

In the discussion here we set the testbed using *Trilinos* 6.0.18. Also note that we observed no substantial code change of our concern in a more recent version *Trilinos* 7.0.9 and similar results are observed with slightly different results (differ by ± 1 for the number of iterations). Available as an open source package, *Trilinos* enables us to track and modify its application of polynomial smoothing. Although no documentation has been found for polynomial smoothing in *Trilinos*, by reading and debugging the source code, we have successfully executed the "hidden" *Trilinos* code segment for polynomial smoothing. In the setup routine (*ml_struct.c*), *Trilinos* disabled the setup of the coefficients for MLS by ignoring the MLS degree parameter passed down and replacing it with a constant. Fixing that, and using a negative "MLS degree" value in the calling sequence, which also does not appear in the documents, we are able to invoke the necessary parts in *Trilinos* for polynomial smoothing testing.

The segment of source code is observed and debugged. We believe it performs in the following way: It uses a full pre-smoothing as in (4.2) and followed by a post-smoothing as in (4.3) to complete one smoothing step. This step is applied at both pre- and post- smoothing process for each AMG V-cycle. This is a deviation from the original document [16] and is expensive since (4.3) is applied twice for each iteration. This smoother is used unchanged in our first configuration. In the second configuration, the original post-smoother (4.3) was removed, leaving only the original pre-smoother (4.2) to serve as the smoother in both pre- and post-smoothing process.

We compare these two alternatives for the linear algebra system generated from the first step of the Airfoil case (Section 2.2.2), and a random vector as the right hand side. The MLS of 2nd order and 4th order are tested to converge to a tolerance of 10^{-7}. With the original treatment (the first configuration) of post-smoothing in *Trilinos*, 2nd-order MLS uses 18 iterations in 27 seconds; 4th-order MLS uses 14 iterations in 35 seconds. When using the pre-smoother instead of the post-smoother (the second configuration), 2nd-order MLS uses 35 iterations in 18 seconds; 4th-order MLS uses 22 iterations in 20 seconds. The first configuration does converge with fewer iterations, however, the total computational cost is higher than the second configuration. Put another way, the computational cost of construction and application of the complex post-smoother in the first configuration was not offset by the relatively modest reduction in iterations relative to the simpler (symmetric) smoother. This result can be problem dependent, but the tests above suggest that we use a simplified pre-smoother as the post-smoother for our problems.

APPENDIX C
ALGORITHMS FOR POLYNOMIAL SMOOTHING

C.1 Definitions

First we define the notations that will be used in the following discussions.

Math form	Descriptions and Definitions
A	Matrix of interest
b	right hand side of interest
x^*	Exact solution , $Ax^* = b$
x_j	Approximate solution at j^{th} outside (ie: CG or GMRES) iteration step.
$x_j^{(m)}$	Intermediate approximate solution at step m in AMG V-cycle, which inside the outside iteration step j.
$x^{(m)}$	We discuss the procedure mainly inside one AMG V-cycle, so we omit the subscript j in the previous form $x_j^{(m)}$.
$r^{(m)}$	Residual: $r^{(m)} = Ax^{(m)} - b$
$e^{(m)}$	Error: $e^{(m)} = x^{(m)} - x^*$
Q,Q^T	Multigrid Interpolation Operator, Restriction Operator.
I	Identity Matrix.
$\rho(A)$	Spectral radius of matrix

From the above we have:

$$Ae^{(m)} = r^{(m)} \tag{C.1}$$

Also define polynomial S:

$$S = (I - \alpha_1 A)(I - \alpha_2 A) \cdots (I - \alpha_d A) \tag{C.2}$$

, where d is the user-specified degree of polynomial smoothing; α_k is the pre-computed coefficients given by:

$$\frac{1}{\alpha_k} = \frac{\rho(A)}{2}(1 - \cos\frac{2k\pi}{2d+1}) \tag{C.3}$$

We define:

$$A^S = S^2 A \tag{C.4}$$

In [16], it was claimed that the choice of polynomial S as in (C.2) will minimize $\rho(A^S)$.

We also define polynomial $\hat{S}$:

$$\hat{S} = I - \frac{\omega}{\rho(A^S)} A^S \tag{C.5}$$

In [16], it was purposed that a smoothing step can be written as:

$$x^{(m+1)} = x^{(m)} + p(A)(b - Ax^{(m)}) \quad \text{(Page 6, equation 1)} \tag{C.6}$$

Notice this $p(A)$ is operated on residual vector $r^{(m)}$ instead of error vector $e^{(m)}$. This p is different in concept from $p_1(A)$ and $p_2(A)$ in [16]'s page 8, (3.1). However we want to construct this $p(A)$ such that the error propagation matrix is S for one smoothing step:

$$e^{(m+1)} = Se^{(m)} \tag{C.7}$$

This leads to a smoothing process, using (C.1):

$$x^{(m+1)} = x^{(m)} - (\alpha_1 + \alpha_2 \cdots)r^{(m)} + \cdots \tag{C.8}$$

, if we use second order MLS (We will use second order MLS in the rest of the report):

$$x^{(m+1)} = x^{(m)} - (\alpha_1 + \alpha 2)r^{(m)} + \alpha_1\alpha_2 A r^{(m)} \tag{C.9}$$

We denote this smoothing process as:

$$x^{(m+1)} \Longleftarrow S(x^{(m)}, b) \tag{C.10}$$

, we have b here as a parameter of S because of that $r^{(m)} = Ax^{(m)} - b$. In fact, we can also use $r^{(m)}$ to replace b as the parameter if $r^{(m)}$ is available directly, that is:

$$x^{(m+1)} \Longleftarrow S(x^{(m)}, r^{(m)}) \tag{C.11}$$

Notice that symbol S can be used as a matrix in polynomial form as in (C.2) if it is directly applied to a vector. It can also indicating a smoothing process if fully written out as a function in (C.10) and (C.11).

Also we want to construct another smoothing process that the error propagation matrix is $\hat{S}$ for one smoothing step:

$$e^{(m+1)} = \hat{S} e^{(m)} \tag{C.12}$$

This leads to a smoothing process:

$$x^{(m+1)} = x^{(m)} - \frac{\omega}{\rho(A^S)} S^2 r^{(m)} \tag{C.13}$$

We denote this smoothing process as:

$$x^{(m+1)} \Longleftarrow \hat{S}(x^{(m)}, b) \tag{C.14}$$

or:

$$x^{(m+1)} \Longleftarrow \hat{S}(x^{(m)}, r^{(m)}) \tag{C.15}$$

Then we compare different approaches to do polynomial smoothing based on classical AMG.

C.2 Variation of the polynomial smoother, alternative 1a

Starting from $x^{(1)}, b, A$, we describe the algorithm in Table C.1.

C.3 Variation of the polynomial smoother, alternative 2

C.4 Variation of the polynomial smoother, alternative 3

Math Form	q=Ap	Error Form	Notes
$r^{(1)} = Ax^{(1)} - b$	0		using zero initial guess.
$x^{(2)} = x^{(1)} - (\alpha_1 + \alpha_2) r^{(1)} + \alpha_1 \alpha_2 A r^{(1)}$	1	$e^{(2)} = S e^{(1)}$	$x^{(2)} \Longleftarrow S(x^{(1)}, r^{(1)})$
$r^{(2)} = Ax^{(2)} - b$	1		
$\hat{r}^{(2)} = S r^{(2)}$	2		Beginning of coarse level correction
$\hat{r}_H^{(2)} = Q^T \hat{r}^{(2)}$ $\hat{v}_H^{(2)} = (Q^T S A S Q)^{-1} \hat{r}_H^{(2)}$ $\hat{v}^{(2)} = Q \hat{v}_H^{(2)}$			Coarse Solve
$v^{(2)} = S \hat{v}_H^{(2)}$	2		End of coarse level correction
$x^{(3)} = x^{(2)} - v^{(2)}$		$e^{(3)} = (I - SQ(Q^T S A S Q)^{-1} Q^T S A) e^{(2)}$	
$r^{(3)} = Ax^{(3)} - b$	1		
$x^{(4)} = x^{(3)} - \frac{\omega}{\rho(A^S)} S^2 r^{(3)}$	4	$e^{(4)} = \hat{S} e^{(3)}$	$x^{(4)} \Longleftarrow \hat{S}(x^{(3)}, r^{(3)})$

Total	q=Ap	Error Propagation Matrix
	11	$E_{1a} = \hat{S}(I - SQ(Q^T S A S Q)^{-1} Q^T S A) S$

Table C.1: Pseudo code of polynomial smoothing Algorithm 1a

Math Form	q=Ap	Error Form	Notes
$r^{(1)} = Ax^{(1)} - b$	0		using zero initial guess.
$x^{(2)} = x^{(1)} - (\alpha_1 + \alpha_2) r^{(1)} + \alpha_1 \alpha_2 A r^{(1)}$	1	$e^{(2)} = S e^{(1)}$	$x^{(2)} \Longleftarrow S(x^{(1)}, r^{(1)})$
$r^{(2)} = Ax^{(2)} - b$	1		
$x^{(3)} = x^{(2)} - \frac{\omega}{\rho(A^S)} S^2 r^{(2)}$	4	$e^{(3)} = \hat{S} e^{(2)}$	$x^{(3)} \Longleftarrow S(x^{(2)}, r^{(2)})$
$r^{(3)} = Ax^{(3)} - b$	1		
$\hat{r}_H^{(3)} = Q^T r^{(3)}$ $\hat{v}_H^{(3)} = (Q^T A Q)^{-1} \hat{r}_H^{(3)}$ $v^{(3)} = Q \hat{v}_H^{(3)}$			Coarse Solve
$x^{(4)} = x^{(3)} - v^{(3)}$		$e^{(4)} = (I - Q(Q^T A Q)^{-1} P^T A) e^{(3)}$	
$r^{(4)} = Ax^{(4)} - b$	1		
$x^{(5)} = x^{(4)} - (\alpha_1 + \alpha_2) r^{(4)} + \alpha_1 \alpha_2 A r^{(4)}$	1	$e^{(5)} = S e^{(4)}$	$x^{(5)} \Longleftarrow S(x^{(4)}, r^{(4)})$
$r^{(5)} = Ax^{(4)} - b$	1		
$x^{(6)} = x^{(5)} - \frac{\omega}{\rho(A^S)} S^2 r^{(5)}$	4	$e^{(6)} = \hat{S} e^{(5)}$	$x^{(6)} \Longleftarrow S(x^{(5)}, r^{(5)})$

Total	q=Ap	Error Propagation Matrix
	14	$E_2 = \hat{S} S (I - Q(Q^T A Q)^{-1} Q^T A) \hat{S} S$

Table C.2: Pseudo code of polynomial smoothing Algorithm 2

Math Form	q=Ap	Error Form	Notes
$r^{(1)} = Ax^{(1)} - b$	0		using zero initial guess.
$x^{(2)} = x^{(1)} - (\alpha_1 + \alpha_2) r^{(1)} + \alpha_1 \alpha_2 A r^{(1)}$	1	$e^{(2)} = Se^{(1)}$	$x^{(2)} \Longleftarrow S(x^{(1)}, r^{(1)})$
$r^{(2)} = Ax^{(2)} - b$	1		
$\hat{r}_H^{(2)} = Q^T r^{(2)}$ $\hat{v}_H^{(2)} = (Q^T AQ)^{-1} \hat{r}_H^{(2)}$ $v^{(2)} = Q\hat{v}_H^{(2)}$			Coarse Solve
$x^{(3)} = x^{(2)} - v^{(2)}$		$e^{(3)} = (I - Q(Q^T AQ)^{-1} Q^T A) e^{(2)}$	
$r^{(3)} = Ax^{(3)} - b$	1		
$x^{(4)} = x^{(3)} - (\alpha_1 + \alpha_2) r^{(3)} + \alpha_1 \alpha_2 A r^{(3)}$	1	$e^{(4)} = Se^{(3)}$	$x^{(4)} \Longleftarrow S(x^{(3)}, r^{(3)})$

Total	q=Ap	Error Propagation Matrix
	4	$E_3 = S(I - Q(Q^T AQ)^{-1} Q^T A) S$

Table C.3: Pseudo code of polynomial smoothing Algorithm 3

APPENDIX D
ALGORITHMS FOR COMMUNICATION IN PARALLEL AMG

D.1 Communication of boundary matrix

It is necessary and non-trivial to communicate matrix entries between two boundary nodes. As we have seen in (5.10) and (5.21), we need do summation on the submatrix e.g. $\mathbf{B}_{bb}$ between partitions.

To do this, first we need to extract the submatrix $\mathbf{B}_{bb}$ that has the boundary-boundary entries, in sparse form. This can be done by scanning the matrix with the knowledge of boundary information. However, we should keep in mind that the sparse structures of boundary submatrices on two neighbouring partitions are not necessary the same, even though they come from a same set of boundary unknowns. An example is set in Figure 5.4. It is clear that on partition 1, matrix entry $\mathbf{B}_{ab}$ is non-existent. But on partition 2, we do have $\mathbf{B}_{ab}$. The next step is to send the extracted submatrix to the neighbouring partition. However, the lack of synchronization of sparsity structures of the two submatrices leads to a double communication. We need to communicate the structure of the submatrix (i.e. the *colm* array) first, then we may communicate the matrix itself (the *rowp* array and the values). After the communication, we add the neighbouring submatrix into the original global matrix. And since there are differences in sparsity structures, the original matrix $\mathbf{B}$ should also change its sparsity structure accordingly to maintain the extra entries.

Note that for the PPE matrix $\mathbf{A}$, there is one value for each matrix entry. When we communicate $\mathbf{G}, \mathbf{C}(lhsP)$, there are four values for each matrix entry but the procedures for both matrices are the same. They both follow the procedure of "Extract, Communicate, and Assemble". We give the detailed description below:

1. **For each communication task**, get the mapping between the whole local matrix and the sub-matrix of partition boundary nodes. Also create a reverse

mapping.

2. Build the *colm* (row array in row-wise matrix sparse storage, see Section 5.1.1) array for the sub-matrix in current node. Allocate an array with same size, call it *colm2*.

3. For master node, send *colm*, receive *colm2*; For slave node, receive *colm2*, send *colm*.

4. Wait for the completion of the communication for all the *colm*s by using the MPI [50] command *WAITALL*.

5. Build *rowp* array and *lhs* array, which stores the row indices and matrix entries, for the sub-matrix, in row-wise sparse format.

6. Allocate *rowp2* array and *lhs2* array from the information in *colm2*.

7. Communicate *rowp*,*rowp2*; *lhs*,*lhs2* arrays.

8. Complete the communication of above by *WAITALL*.

9. Specifically, in the context of our previous example: in partition 1, we have $\mathbf{B}_{22}^{(1)}$ in *colm, rowp, lhs*, also $\mathbf{B}_{22}^{(2)}$ in *colm2, rowp2, lhs2*. In partition 2, we have $\mathbf{B}_{22}^{(2)}$ in *colm, rowp, lhs*, also $\mathbf{B}_{22}^{(1)}$ in *colm2, rowp2, lhs2*.

10. Do a sparse matrix summation of the two submatrices.

11. Using the mapping and reverse mapping we created in the first step, we can put back the summed values into the corresponding positions in the whole local matrix. This is also done in a sparse matrix summation context.

12. Loop over all the tasks (neighbours) until all tasks are finished.

This communication of matrices only needs to be done once before the solution phase. It is within the AMG setup phase. Since we can do multiple linear solves with one setup, this overhead cost should be negligible for long runs.

D.2 Parallel Ap-product for coarser AMG levels

For matrix-vector product (**Ap**-product) of PPE, a solution has been given in Section 5.1. In this section we will implement **Ap**-products for coarser matrices.

We have taken care to preserve the summation-complete property for coarser level matrices, as in (5.32). This enables us to do **Ap**-products locally on partition and then communicate the boundary values to make the result correct on each partition.

Using a two-partition example as in the previous section, we have a correct result expressed in the serial version:

$$\mathbf{q} = \mathbf{A}\mathbf{p} \tag{D.1}$$

$$\begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_b \\ \mathbf{q}_2 \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{1b} & 0 \\ \mathbf{A}_{b1} & \mathbf{A}_{bb} & \mathbf{A}_{b2} \\ 0 & \mathbf{A}_{2b} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_b \\ \mathbf{p}_2 \end{bmatrix} \tag{D.2}$$

In parallel, we have AMG matrices on partition (1) as the upper-left 2×2 submatrix of $\mathbf{A}$, and the lower-right 2×2 submatrix of $\mathbf{A}$ on partition (2). Also, we can assume that we have vector whose boundary values are correct on master partition, say, partition (1):

$$\mathbf{p}_b^{(1)} = \mathbf{p}_b \tag{D.3}$$

First we do communication similar to *commOut* in Section (5.1.2), this is to broadcast boundary values to the slave partitions and make:

$$\mathbf{p}_b^{(2)} = \mathbf{p}_b^{(1)} = \mathbf{p}_b \tag{D.4}$$

Then we do **Ap**-product on each partition. On partition (1), the master partition, we have:

$$\begin{bmatrix} \mathbf{q}_1^{(1)} \\ \mathbf{q}_b^{(1)} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{11} & \mathbf{A}_{1b} \\ \mathbf{A}_{b1} & \mathbf{A}_{bb} \end{bmatrix} \begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_b \end{bmatrix} \tag{D.5}$$

Expand and compare we have the following on master partition:

$$\begin{bmatrix} \mathbf{q}_1^{(1)} \\ \mathbf{q}_b^{(1)} \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{A}_{11}\mathbf{p}_1 + \mathbf{A}_{1b}\mathbf{p}_b \end{bmatrix} \tag{D.6}$$

Ap-product on partition (2), the slave partition, gives:

$$\begin{bmatrix} \mathbf{q}_b^{(2)} \\ \mathbf{q}_2^{(2)} \end{bmatrix} = \begin{bmatrix} (\mathbf{A}_{bb}) & \mathbf{A}_{b2} \\ \mathbf{A}_{b2} & \mathbf{A}_{22} \end{bmatrix} \begin{bmatrix} \mathbf{p}_b \\ \mathbf{p}_2 \end{bmatrix} \tag{D.7}$$

We need to ignore the $\mathbf{A}_{bb}$ part in the slave nodes when we perform this calculation resulting in:

$$\begin{bmatrix} \mathbf{q}_b^{(2)} \\ \mathbf{q}_2^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{A}_{b2}\mathbf{p}_2 \\ \mathbf{q}_2 \end{bmatrix} \tag{D.8}$$

Next, we do communication similar to *commIn*, that is to add values from slave nodes to the master nodes resulting in:

$$\begin{bmatrix} \mathbf{q}_1^{(1)} \\ \mathbf{q}_b^{(1)} \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{A}_{11}\mathbf{p}_1 + \mathbf{A}_{1b}\mathbf{p}_b + \mathbf{q}_b^{(2)} \end{bmatrix} = \begin{bmatrix} \mathbf{q}_1 \\ \mathbf{q}_b \end{bmatrix} \tag{D.9}$$

This recovers our assumption that we have a vector whose boundary values are correct on the master partition. Therefore, we can proceed to future **Ap**-products.

It should be noticed that the communication *commOut* and *commIn* are not the same size as those are used in PPE, which we have discussed in Section 5.1.2.

Here, for each coarse AMG level, the communication is done for the vector of size of the coarse matrix. An auxiliary array is used to track the coarsening information and neighbouring information for each level. Then we are able to build an index array that saves information of coarse boundary unknowns. After that, for each communication task, we can map out the coarser nodes that need communication.

The **Ap**-product, once made parallel complete, is used in both polynomial smoothing and in the AMG V-cycle to calculate residual vector. For both applications here, no extra work is needed once the **Ap**-product is complete.