2005

Development of advanced numerical algorithms for modeling complex flows

Michael Gino Kapper
San Jose State University

Follow this and additional works at: https://scholarworks.sjsu.edu/etd_theses

Recommended Citation

Kapper, Michael Gino, "Development of advanced numerical algorithms for modeling complex flows" (2005). *Master's Theses*. 2734.
DOI: <https://doi.org/10.31979/etd.fc9s-bcvu>
https://scholarworks.sjsu.edu/etd_theses/2734

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

NOTE TO USERS

This reproduction is the best copy available.

UMI[®]

**DEVELOPMENT OF ADVANCED NUMERICAL ALGORITHMS FOR
MODELING COMPLEX FLOWS**

A Thesis

Presented to

The Faculty of the Department of

Mechanical Engineering

In Partial Fulfillment

of the Requirements for the Degree of

Master of Science

By

Michael Gino Kapper

May 2005

UMI Number: 1427189

Copyright 2005 by
Kapper, Michael Gino

All rights reserved.

INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

UMI[®]

UMI Microform 1427189

Copyright 2005 by ProQuest Information and Learning Company.

All rights reserved. This microform edition is protected against
unauthorized copying under Title 17, United States Code.

ProQuest Information and Learning Company
300 North Zeeb Road
P.O. Box 1346
Ann Arbor, MI 48106-1346

© 2005

Michael Gino Kapper

ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF
MECHANICAL ENGINEERING

Dr. Periklis Papadopoulos

Dr. Nikos Mourtos

Dr. Jean-Luc Cambier

Air Force Research Laboratory, Edwards AFB

APPROVED FOR THE UNIVERSITY

ABSTRACT

DEVELOPMENT OF ADVANCED NUMERICAL ALGORITHMS FOR MODELING COMPLEX FLOWS

by Michael Gino Kapper

The focus of the current research was to develop several advanced algorithms applicable to numerical modeling of complex flows. These algorithms are encompassed in several areas of fluid modeling, including parallelization, geometry, numerical methods, and visualization. They are presented in the context of a new software that has been developed as a result of the current research. The software has been fully parallelized and was designed using Object-Oriented methods for maximal flexibility, portability, and ease of maintenance. This thesis develops the logic and strategies behind the parallel architecture and details in full its components and their implementation. There is also a discussion on the fluid solvers and their implementation as well as performance results obtained in conjunction with the present architecture.

ACKNOWLEDGEMENTS

I am grateful to all those who have helped make this thesis what it is.

Firstly, I would like to thank Professor Periklis Papdopoulos, without whom none of this would have been possible. His constant drive and determination greatly influenced me to keep focused on the larger goals when it was difficult to see past the details.

I would like to thank Professor Nikos Mourtos for his calm and collectedness that helped to get me through the thesis class.

I am especially indebted to Dr. Jean-Luc Cambier for tolerating my seemingly never-ending tangents, which undoubtedly have lead to many key components of this research.

I would also like to thank everyone at the Air Force Research Laboratory who has in one way or another contributed to this work.

This work was supported by the Air Force Research Laboratory through a consultant position under the *Research in Propulsion Sciences* contract #F04611-99-0025 with ERC, Incorporated.

PREFACE

The software presented in this work has been developed in association with the U.S. Air Force. As a result, it is property of the U.S. Government and has not been authorized for unlimited public distribution. All inquiries into this matter are directed to the project manager, Dr. Jean-Luc Cambier of the Air Force Research Laboratory at Edwards Air Force Base.

P.O.C.
Dr. Jean-Luc Cambier
Propulsion Directorate – AFRL/PRSA
10 East Saturn Blvd.
Edwards AFB, CA 93524-7680
Tel: (661) 275-5676
E-mail: jean-luc.cambier@edwards.af.mil

CONTENTS

ACKNOWLEDGEMENTS	v
PREFACE	vi
LIST OF FIGURES	xi
LIST OF SYMBOLS USED	xiv
CHAPTER ONE—INTRODUCTION	1
1.1 Background and Motivation	1
1.2 Goals of Current Research	1
CHAPTER TWO—SOFTWARE ARCHITECTURE	2
2.1 Introduction	2
2.2 Ease of Development	2
2.3 Performance with an Object-Oriented Design	3
2.4 Scalability and Portability	4
2.5 Extensibility	5
2.6 Usability	6
CHAPTER THREE—PARALLELIZATION	7
3.1 Introduction	7
3.2 Approach	7
3.2.1 Shared-Memory Architecture	8
3.2.2 Distributed-Memory Architecture	9
3.2.3 Clustered-Memory Networks	10
3.3 The Client/Master/Server Framework	11
3.3.1 Server	12
3.3.2 Master	12
3.3.3 Client	13
3.4 Domain Decomposition	13

3.5	Synchronization	13
3.6	Inter-Nodal Communication	14
CHAPTER FOUR—GEOMETRY		17
4.1	Introduction.....	17
4.2	Compatibility	17
4.3	Structure and Storage.....	19
4.4	Accessibility.....	20
CHAPTER FIVE—PREPROCESSOR		22
5.1	Introduction.....	22
5.2	Overview	22
5.3	Approach.....	23
5.4	Connectivity Subroutine	24
5.5	Orientation	26
5.5.1	Two-Dimensional Subroutine.....	27
5.5.2	Three-Dimensional Subroutine.....	28
5.6	Boundary Ghost Cells	32
5.7	Complex Topologies.....	33
CHAPTER SIX—NUMERICAL METHODS.....		35
6.1	Introduction.....	35
6.2	Governing Equations	35
6.3	Numerical Formulation.....	37
6.4	Time Discretization.....	38
6.4.1	Explicit Euler	38
6.4.2	Total Variation Diminishing Runge-Kutta (TVD RK).....	39
6.5	Numerical Flux	40
6.5.1	Roe's Flux Difference Splitting (FDS)	40
6.5.2	Harten-Lax-van Leer-Einfeldt Riemann Solver (HLLE).....	41
6.5.3	Modified HLLE Flux (HLLEM).....	42
6.5.4	Equilibrium Flux Method (EFM).....	43

6.5.5	EFM with Osher Flux Splitting (EFMO).....	44
6.5.6	Total Variation Diminishing Flux (TVD).....	45
6.5.7	Piecewise Parabolic Method (PPM)	47
6.5.8	Weighted Essentially Non-Oscillatory (WENO).....	48
CHAPTER SEVEN—VISUALIZATION.....		51
7.1	Introduction.....	51
7.2	Integration	51
7.3	VisAD API.....	53
CHAPTER EIGHT—BENCHMARKS.....		55
8.1	Introduction.....	55
8.2	Numerical Solvers.....	55
8.2.1	Time Discretization.....	56
8.2.2	Mach 3 Flow Past a Forward-Facing Step.....	57
8.2.3	Diffraction of a Shock by a Finite Wedge	63
8.2.4	Diffraction of a Shock Down a Step	64
8.2.5	Sod’s Shock Tube Problem.....	64
8.2.5.1	One-Dimensional Solution.....	66
8.2.5.2	Two-Dimensional Solution.....	66
8.2.5.3	Three-Dimensional Solution.....	69
8.3	Parallel Performance	70
8.3.1	Shared-Memory Scalability	71
8.3.2	Distributed-Memory Scalability	72
CHAPTER NINE—CONCLUSIONS.....		75
9.1	Achievements and Contributions.....	75
9.2	Recommendations for Future Work.....	75
REFERENCES		77
APPENDIX A—METRIC TRANSFORMATIONS.....		81
A.1.	Two-Dimensional Finite-Volume Metrics.....	81
A.2.	Three-Dimensional Finite-Volume Metrics.....	83
APPENDIX B—BOUNDARY CONDITIONS		85

B.1.	Reflective Boundary Conditions.....	85
B.2.	Inflow Boundary Conditions.....	87
B.3.	Outflow Boundary Conditions.....	87
APPENDIX C—HIGH PERFORMANCE BLAS		88
C.1.	Subroutines	88
C.2.	Performance	88
APPENDIX D—RANKINE-HUGONIOT RELATIONS		91
APPENDIX E—NUMERICAL SCHLIEREN IMAGING.....		93

LIST OF FIGURES

Figure 1. Extending classes in Java.	6
Figure 2. Shared-memory architecture.	8
Figure 3. Distributed-memory architecture.....	9
Figure 4. Clustered-memory network configuration.	11
Figure 5. Client-Master-Server framework.....	11
Figure 6. Synchronization encapsulation.....	15
Figure 7. Boundary information exchange.	16
Figure 8. Structured grid assembly from coordinate values.	18
Figure 9. Mesh creating from structured grid.	19
Figure 10. Two-dimensional pointer information.....	20
Figure 11. Comparison between indicinal notation and software nomenclature.....	20
Figure 12. Two overlapping grids.....	24
Figure 13. Connectivity subroutine.....	25
Figure 14. Two-dimensional orientation configurations.	27
Figure 15. Two-dimensional orientation subroutine.....	28
Figure 16. Three-dimensional orientation configurations.	29
Figure 17. Local reference frame for cell face.....	30
Figure 18. Three-dimensional orientation subroutine.....	31
Figure 19. Extrapolated coordinates for boundary ghost cells.	32
Figure 20. Possible corner ghost cell configurations.....	34

Figure 21. Possible boundary ghost cell configurations.	34
Figure 22 Real-time data visualization with Graphical User Interface.....	52
Figure 23 Data mapping for visualization.	54
Figure 24. Numerical solution with explicit Euler time discretization showing oscillations.	56
Figure 25. Numerical solution with 2 nd -Order TVD Runge-Kutta time discretization, without oscillations.	57
Figure 26. Roe’s FDS solution of forward-facing step problem.	58
Figure 27. HLLE ($\beta = 1$) solution of forward-facing step problem.....	59
Figure 28. HLLE ($\beta = 0$) solution of forward-facing step problem.....	59
Figure 29. EFM solution of forward-facing step problem.	60
Figure 30. EFMO solution of forward-facing step problem.	60
Figure 31. TVD solution of forward-facing step problem.	61
Figure 32. Hybrid solution of forward-facing step problem.....	61
Figure 33. PPM solution of forward-facing step problem.	62
Figure 34. WENO solution of forward-facing step problem.	62
Figure 35. Numerical solution of a shock diffracting over a wedge compared with experimental results.	63
Figure 36. Numerical solution of a shock diffracting down a step compared with experimental results.	65
Figure 37 One-dimensional solution of Sod’s shock tube problem.....	67
Figure 38 Two-dimensional solution of Sod’s shock tube problem (isometric view).....	68

Figure 39 Two-dimensional solution of Sod's shock tube problem (side and top views).	68
Figure 40 Three-dimensional solution of Sod's shock tube problem.	69
Figure 41 Single node computation time versus number of decomposed blocks in domain.	72
Figure 42 Computation speed-up of 32 blocks versus number of nodes.	73
Figure 43 Parallelization efficiency of 32 blocks versus number of nodes.	74
Figure 44. Rotation transformation of two-dimensional interface.	81
Figure 45. Rotation transformation of three-dimensional interface.	83
Figure 46. Reflective boundary condition.	85
Figure 47. Optimized matrix multiplication subroutine in Java.	89
Figure 48. Optimized matrix multiplication subroutine in C.	89
Figure 49. Performance comparison of Java and C matrix multiplication.	90
Figure 50. Two flow states separated by a shock wave.	91

LIST OF SYMBOLS USED

Roman Symbols

a	speed of sound
A, B, C	flux Jacobians
e	internal energy per unit volume
E	total energy per unit volume
F, G, H	flux vectors
M	mach number
p	pressure
Q	vector of conserved quantities
R	universal gas constant
T	temperature
T	rotation transformation matrix
T^{-1}	inverse rotation transformation matrix
t	time
u, v, w	velocity components
$\hat{u}, \hat{v}, \hat{w}$	rotated velocity components
u', v', w'	transformed velocity components
V	vector of velocity components
W	vector of characteristic variables

Greek Symbols

Δ	finite-difference operator
λ	eigenvalue/characteristic speed
Λ	diagonal matrix of eigenvalues
α	wave strengths
ρ	density
γ	ratio of specific heats
θ, ϕ	angles of rotation

CHAPTER ONE

INTRODUCTION

1.1 Background and Motivation

As the performance and capabilities of computers continue to increase, so does the complexity of the problems that scientists and engineers attempt to solve. Despite this, most programs which are used to solve these problems do not attempt to manage the complexity of the problems which they seek to solve. If the complexity of the problem is minimized, the scientist can then focus on implementing the relevant physics needed to solve the actual problem at hand.

Supersonic fluid flow is certainly no exception to the list of practical problems where complexities are inherent. In solving a complex flow problem difficulties can be encountered in the areas of parallelization, geometry, visualization and the actual numerical solvers. It can be expected that the overall quality of a program can be maximized by focusing on these particular areas

1.2 Goals of Current Research

It is therefore the goal of this current research to develop a new software that incorporates several advanced algorithms to not only simplify, but to also improve upon existing flow solvers. Contained within the areas of parallelization, geometry, visualization and numerical solvers, the algorithms are expected to make the software unique among others in terms of capability.

CHAPTER TWO SOFTWARE ARCHITECTURE

2.1 Introduction

The design of the software architecture is perhaps the most important consideration in the application development process. This is due to the fact that the functionality of the application is deeply rooted in the software architecture. If the architecture does not suit the needs of the application, the application will be adversely affected, ultimately leading to its failure. This is why much thought and effort has gone into the design of the software architecture.

The task of designing the architecture was first approached by assessing the software's functional requirements. Because of the intended application as a complex flow solver, ease of development, Object-Oriented design, performance, scalability, portability, extensibility, and usability, are the key attributes that have been used in defining the overall software architecture. For numerous reasons to be expounded upon in the following sections, Java has been chosen as the language to address and incorporate these attributes into the software architecture.

2.2 Ease of Development

Given the task of constructing an application containing complex numerical and parallel algorithms, the last thing a programmer (a.k.a. engineer, scientist, etc.) needs is to be bogged-down by complex or cumbersome programming language formalities. Java helps tremendously in this regards by providing a simple syntax that allows programmers

to easily develop, debug, and share software modules. Studies have shown that Java programming results in fewer bugs and faster development (Markidis, 2004). As an example, Java does not require the programmer to explicitly allocate nor deallocate memory needed for variables as required by the C programming language. The Java Virtual Machine (JVM) contains a garbage collection utility which scavenges for unused variables and frees their memory resources. This helps significantly in minimizing memory leaks due to programmer errors.

2.3 Performance with an Object-Oriented Design

The complexity of problems can also be tamed with Java's strong Object-Oriented (OO) design. With an OO design, user-defined data structures known as objects can be created from the abstraction of common physical concepts. These objects can then be associated with certain behaviors, allowing full interaction with the object. This greatly improves the overall flexibility of the code.

There are various degrees to which a code can incorporate OO techniques, involving compromises between flexibility and performance. Fine-grained OO programming, which makes extensive use of encapsulation techniques as well as large arrays of objects, usually leads to poor performance unless advanced compilation techniques are utilized (Markidis, 2004). Full object encapsulation utilizes method calls to access and set data within an object, costing valuable computational resources. Arrays of objects also serve to limit performance by increasing the distance between elements in the computer's physical memory, thereby increasing data access time.

Coarse-grained OO programming, on the other hand, relies on a limited number of objects and method calls and uses arrays of primitive types as opposed to objects, resulting in faster executing applications that resemble procedural-type programming. It is worth noting here that procedural programming in Java can result in similar performance as C programming. Performance results for a matrix multiplication test case are given in Appendix C.

The approach taken in this work has been to implement an OO design that lies somewhere in-between the two extremes. This medium-grained approach sacrifices some performance by utilizing objects wherever possible to enhance code flexibility, but at the same time it does not embrace encapsulation to its full extent. This means that instead of accessing data by means of method calls, most data is acquired through public accessors. As a result, both flexibility and performance are optimized together to achieve a suitable balance.

2.4 Scalability and Portability

When dealing with engineering and scientific applications requiring large amounts of computational resources, scalability of the software is an absolute necessity. Most problems are too large to solve on a single processor PC and often require multiple processors as well as multiple machines. Taking this into account, the software architecture should be designed to function as a distributed application. That is, the architecture should be designed as a set of programs that run in different processes which communicate with each other to form a single application (Grosso, 2002). These

different processes can then be distributed to different processors or even different machines, thus achieving the desired level of scalability.

This approach can only be successful if there exists the ability to not only create these individual processes, but also to allow them to communicate with each other. With Java's network-aware environment, support for such distributed applications is provided through extensive multi-threading capabilities as well as the Remote Method Invocation (RMI) message-passing protocol. Using a combination of multi-threading and Java RMI, an application can be scaled to both distributed and shared memory networks.

Portability of the software must also be seriously considered if high-levels of scalability are to be achieved. Since Java byte code is portable across multiple platforms, various parallelization approaches across different platforms can easily be implemented and tested.

2.5 Extensibility

A key benefit of using objects in programming is that upgrading and extending code becomes highly simplified. Objects can be thought of as modular components in a code that can be added to the overall application to increase its capabilities. This allows modification of the code without affecting what has previously been implemented.

As a means of predetermining how certain classes can be extended, Java allows for abstract classes which simply serve to define the attributes that all subclasses must inherit. This is very useful when the same software is co-developed by several people.

As long as a programmer properly overrides the methods defined in the abstract class, compatibility of any subclass with the software is guaranteed.

Figure 1 illustrates an abstract class *Grid* that can be created with several abstract methods which must be overridden by the *Structured2DGrid*, *Structured3DGrid*, and all other subclasses

Figure 1. Extending classes in Java.

2.6 Usability

As the language of the internet, Java provides significant advantages over other languages in terms of usability. With standard interfaces for graphics and extensive support for Graphical User Interfaces (GUIs), Java allows users to easily interface with applications. User interaction with an application allows for remote steering and real-time visualization of the solution.

CHAPTER THREE PARALLELIZATION

3.1 Introduction

In order to solve problems of practical interest, a wealth of computational resources is needed to handle the demanding load. This is why scalability has been one of the key attributes in shaping the overall software architecture. The ability to utilize all available resources is crucial, especially when every bit of computational power makes a difference.

In terms of scientific computing, scalability translates into parallelization of the software on multiple processors and machines. That is, the design of the software as a set of programs that when executed, form a single application. This is the basis of a distributed application.

3.2 Approach

There exist many more options today than 20 or even 10 years ago for software parallelization. Once costing hundreds of thousands of dollars, multiprocessor machines are now widely available, making applications designed for shared-memory architectures highly desirable. The growth of the internet has also created a wealth of computing resources, forming what can be basically considered a massive distributed-memory network. When combined with the fact that Java is the language of the internet, the possibilities for parallelization are phenomenal.

With all of these open avenues there are several levels of parallelization that could be applied to the code. And instead of limiting the parallelization to any one, the approach has been to make the parallelization of the software network-independent.

The following sections describe various network and hardware configurations and how Java can be used to parallelize the software in each case.

3.2.1 Shared-Memory Architecture

In shared-memory architectures, all processors in the system are connected to a single bank of memory through a high speed bus as shown in Figure 2.

Figure 2. Shared-memory architecture.

This type of system is desirable from the standpoint that all processors access the same memory, and no additional effort from the programmer is required to allow a given processor to access a block of memory. However, in order to harness all of the available resources provided by the multiple processors, it is necessary to determine if there are any tasks in the code that can be separated and processed concurrently, with some level of independence from each other. If such a distinction can be made, then parallelization is accomplished by creating individual threads for these tasks which are then computed

concurrently. Applications which implement this technique are known as multi-threaded applications.

Java has extensive support for writing multi-threaded programs. High level thread support was included in the design of Java from the beginning, making the development of multi-threaded programs a relatively simple task. Once threads are created, the Java Virtual Machine (JVM) can exploit the multiple threads by running them simultaneously.

The disadvantage of a shared-memory architecture is that contention for memory can create a severe bottleneck if too many processors are connected to the system, thus degrading system efficiency. This issue can be remedied by complementing shared-memory architectures with distributed-memory networks.

3.2.2 Distributed-Memory Architecture

Distributed-memory architectures are comprised of a multitude of machines, each one containing a single processor and a local bank of memory as shown in Figure 3.

Figure 3. Distributed-memory architecture.

Just as shared-memory configurations, these networks also rely on multithreaded applications to achieve parallelization. The difference being, however, that instead of the individual threads being computed in parallel on just different processors, they are now

being computed on entirely different machines. This requires a protocol of some form to allow remote memory exchange between machines since each processor cannot directly access the memory used by another processor.

One of the ways to achieve this is to have global addressing of all the distributed memory so that each processor thinks that it is accessing a single address space. This approach does not require any extra effort by the programmer, but rather relies on additional hardware. The downside to global addressing is that flexibility is lost in terms of optimizing data exchange between servers.

As the alternative to global addressing, message passing involves the programmer to make explicit calls to remote machines to “fetch” the data that is required to maintain the consistency of the application. Java Remote Method Invocation (RMI) is one such protocol that is included with the Java API. Because Java RMI is implemented completely in the Java language, it is a simple and easy framework to use, allowing easy development of applications designed for distributed networks.

3.2.3 Clustered-Memory Networks

Clustered-memory networks (or simply just clusters) are essentially a hybrid of both shared- and distributed-memory networks. Figure 4 depicts a simple cluster in which several dual-processor machines are connected to one another in a distributed-type fashion.

Much more complicated variants can be imagined including clusters of clusters or even the internet. The benefits of clusters is that they can take advantage of the

Figure 4. Clustered-memory network configuration.

scalability capabilities of both types of architectures. Shared-memory computers, which can achieve higher parallelization efficiency, are used to handle as many threads as possible before their efficiency begins to degrade due to bottlenecking of the memory, at which point further parallelization is achieved by linking the computers through a distributed network.

3.3 The Client/Master/Server Framework

Making extensive use of Java's capabilities for developing distributed applications, a Client-Master-Server framework has been created. Drawing from the cluster paradigm, the framework is fully configurable for either shared- or distributed-memory hardware configurations as well as any combination thereof. Figure 5 illustrates the general configuration of the framework.

Figure 5. Client-Master-Server framework.

The individual components and their roles in the framework are the discussed in the following sections.

3.3.1 Server

In its current configuration, the servers can be thought of as the work horses of the system. They are responsible for executing the code that solves the relevant physical equations. The computational load on each server is broken up into multiple threads which are then handled by the hardware architecture of the server. The hardware architecture of each server can either be a single or multi-processor machine. Each server engages in message passing with other servers to exchange boundary information. This communication, which is deemed inter-nodal communication, is discussed in detail in Section 3.6.

3.3.2 Master

The master is a command and control center, responsible for gathering and distributing information to the servers and client, scheduling the tasks, providing critical I/O, and monitoring commands from the client (Cambier, 2004). One of the more important roles of the master is to engage in two-way communication with each server to ensure synchronization between the server threads, as discussed in Section 3.5.

3.3.3 Client

From the user's point of view, the client is the access point into the entire framework. By means of a Graphical User Interface (GUI), the client allows remote-steering of simulations by the user. The user is able to configure, modify, and visualize the problem and solution in real time, all in a user-friendly environment. Before a simulation is initiated, problem-dependent tasks such as the set-up and initialization of boundary conditions, initial conditions, and numerical solvers can be configured through the client. While a simulation is running, requests can be made to the master for data to be sent for visualization.

3.4 Domain Decomposition

It was previously mentioned that parallelization is achieved through multi-threading, without any regards as to what constitutes a thread. The physical domain of interest is decomposed into various "blocks" which contain a certain amount of data that must be processed. The threads then are created within each block, encapsulating the computational workload within a single method, *run()*. When the *run()* method is initialized, a new thread is created, allowing a separate thread for each block. The size and computational load of each thread can be tailored by changing the size of the blocks.

3.5 Synchronization

The synchronization of all iterations and communications is the responsibility of the master. The master utilizes various locks and the associated *wait/notify* methods

inherited from `java.lang.Object` to determine when all the servers have finished their computation and communication iterations and to cue them so that they may proceed with the next iteration. Once all of the blocks have been locked, the master is notified, and in turn, notifies all the blocks to begin the next iteration. The master is also in charge of global data which is processed from local data from each block. Such as the time step, or data that must be visualized.

The wait/notify mechanism allows one thread to wait for a notification from another thread so that it may proceed. While a thread is waiting, it uses virtually no processor resources. All inter-thread communication through the *wait/notify* mechanism has been encapsulated within special synchronization classes. As illustrated by the example code in Figure 6, synchronized methods are utilized to ensure that communication between threads is never sacrificed.

3.6 Inter-Nodal Communication

The most crucial concern regarding communication overhead is the optimization of the intra-/inter-server communications. At every time-step, it is necessary for each block in the domain to share its information with one or more neighboring blocks. If two neighboring blocks are being computed on the same server, the communication can be handled by standard local method calls. However, if two neighboring blocks are computed on different servers, the inter-server communication is involved and it becomes necessary to use Java-RMI or other suitable message passing protocol. Figure 7 illustrates how two neighboring blocks communicate with each other.

```

public class Sync
{
    private boolean value

    public Sync() {}

    public synchronized void setValueTrue()
    {
        value = true;
        notifyAll();
    }

    public synchronized void setValueFalse()
    {
        value = false;
        notifyAll();
    }

    public synchronized boolean getValue()
    {
        return value;
    }

    public synchronized void waitUntilTrue()
    {
        while( value == false )    wait();
        return true;
    }

    public synchronized void waitUntilFalse()
    {
        while( value == true )    wait();
        return true;
    }
}

```

Figure 6. Synchronization encapsulation.

There are two types of patches which allow data transfer between two blocks. The first of these being the source patch is an array of objects which contain references to data available in the boundary layer of cells. The data references are depicted by the dashed arrows in Figure 7.

Figure 7. Boundary information exchange.

The second is the target from which the ghost layer of cells receives its information. During block communication, there is a method call for each source/target patch pair in which the target patch is filled arrows which can either be a local call if both blocks are on the same server or an RMI call if each block is on a separate server. The exchange of boundary condition information between servers is done without involving the master: this prevents a bottle-neck when scaling to a large number of processors.

CHAPTER FOUR GEOMETRY

4.1 Introduction

One of the capabilities of the software presented in this work is to solve problems in a physical space, whether it be one-, two-, or three-dimensional. As such, the geometry of the problem must be well-defined and organized.

Addressing these concerns, a geometry package has been created, containing a hierarchy of structural components, which is based on a discretization of the physical domain. In many ways, the package can be thought of as an information database. The geometry package determines how data is structured, stored, accessed, modified, and ensures its compatibility.

4.2 Compatibility

The first compatibility issue that must be dealt with concerns the grid structure. Many different types of grid configurations exist for Computational Fluid Dynamics, including Unstructured, Structured, Cartesian, Uniform, and Curvilinear. Ideally, the software should be independent of the type of grid that is used to discretize the space. This is not an easy task, however, considering that numerical solvers are highly grid-dependent. For example, even though an efficient finite-difference solver can be implemented for a structured grid, the same cannot be done for an unstructured grid.

The numerical algorithms used in this work are all intended for use on structured grids, and as a result, unstructured grids have not yet been implemented. As can be seen

in Figure 8, structured grids are easier to deal with, facilitating construction from listing of coordinate values.

Figure 8. Structured grid assembly from coordinate values.

This does not mean, however, that the data structure has been limited to structured grids. Much effort has been made to include both structured and unstructured grids within the same code. For example, coordinates for the grid are stored in the Java array

```
double x[] [] = new double[#dimensions] [#coordinates]
```

This allows unstructured grids to be stored in the same coordinate array as structured grids.

4.3 Structure and Storage

Once the grid object has been created, the connectivity between the grid coordinates must be determined and stored. As shown in Figure 9, a grid can be seen as a collection of cells, surfaces, and vertices.

Figure 9. Mesh creating from structured grid.

The vertices which represent the coordinate points are the corners of the cells which are defined as the finite volumes (or areas) in space, enclosed by the surfaces (or lines) that connect the vertices. This interpretation of the grid forms the basis for the *Mesh* object within the geometry package. The *Mesh* is a highly Object-Oriented construct, containing a hierarchy of *Cell*, *Face* and *Node* objects. These objects are stored in the *Mesh* object in one-dimensional arrays. This, again, allows for compatibility of both structured and unstructured grids.

4.4 Accessibility

As with any database, the geometry package requires efficient and reliable accessibility of data. In the software, this is made possible primarily through pointers between objects. The *Cell*, *Face* and *Node* objects each hold a set of pointers to their immediate neighbors, as detailed in Figure 10.

Figure 10. Two-dimensional pointer information.

This makes finding and accessing local objects very simple. It also eliminates the need for i,j,k -ordered indices. To demonstrate this, a typical finite-volume stencil, as illustrated in Figure 11, can be used to compare standard i,j,k -ordered indices with the

Figure 11. Comparison between indicinal notation and software nomenclature.

pointers used in the software.

All numerical solver implemented in the code will loop through all faces in the domain and calculate the numerical flux at each face. Given that the flux is being computed at interface $i+1/2$, the neighboring cells and faces can be accessed by the following pointers.

```
faceL = face.face[0]
faceR = face.face[1]
cellL0 = face.cell[0]
cellR0 = face.cell[1]
cellL1 = face.face[0].cell[0]
cellR1 = face.face[1].cell[1]
```

The only overhead that is incurred by this approach is accessing the data from memory, which can assumed to be negligible. This may, however, actually be beneficial in three dimensions where accessing data from three-dimensional arrays can be quite costly. Very key also is that this approach is direction-independent—it is equally applicable in the i -direction as it is in either the j - or k -directions, sans modification. This means that the exact same code can be used on any face in the domain, irrespective of its direction.

CHAPTER FIVE PREPROCESSOR

5.1 Introduction

In Chapter Three (see p. 6) it was shown that remote method calls by means of Java RMI are required for parallelization of the software architecture on distributed networks. Remote calls are made from one server to another to access remote data necessary for local computations. However, without any sort of structure of the data that is to be sent or where the data is to be sent, remote method calls would result in gibberish communications. Information about which domains are connected to each other, as well as cell connectivities and their subsequent orientations, is crucial to a successful parallel communication strategy. As a result, a preprocessor has been developed to address these issues.

5.2 Overview

The preprocessor provides a link between the input grid and how the parallelization is executed. Given a structured grid file from a commercial grid generator, information about how the individual blocks are connected is lacking—only information specific to each block can be ascertained explicitly. As described in the previous chapter, only information about grid nodes, their coordinates, and how they are connected is made available. Although most commercial grid generators provide some sort of connectivity file, a standard format does not exist. And instead of adapting the current software to be

compatible with multiple formats, it is much more desirable to generate this connectivity information from the information provided by the input grid file, which is standardized.

This ability is not only desirable but also necessary when patching a plural of grids together having different origins. For example, if a commercial code is used to generate a grid about a highly complex geometry to which a simple Cartesian grid generated separately is to be attached, the connectivity must then be found independently of either grid generator. The preprocessor can take both grids as input and generate all relevant connectivity information. This is especially beneficial when attempting to merge different modules or components together such as a combustion chamber and a nozzle that have been created separately.

Greater flexibility is also allowed in the ability to modify existing grids. As another example, if it is found that certain blocks contained within a previously generated grid are not pertinent to the solution (i.e. domain is too large), they can be removed and the connectivity information can be reconfigured without regenerating the grid.

5.3 Approach

As previously mentioned, the primary information that the preprocessor relies on is simply a grid file containing the coordinate values of the grid points from each block. It is then with this information that all connectivities must be made.

The approach taken here is to match neighboring blocks and cells with each other by locating coordinate values around the border of each block that overlap with those of

its neighbor. This becomes obvious when the coordinate points of all the blocks are visualized in the same plot as shown in Figure 12.

Figure 12. Two overlapping grids.

Intuitively, the connection between the two blocks is obvious even though it is not explicitly stated. How to quantify that connection is the key to the preprocessor and is accomplished through a series of subroutines as follows.

5.4 Connectivity Subroutine

The purpose of the connectivity subroutine is simply to find the existing connectivities between the boundary cells of each block in the entire problem domain. This is accomplished by looping over all cell faces along the boundary of a block and attempting to find a match with another cell face. Relying heavily on the object-oriented design of the geometry package, the algorithm makes extensive use of the *Cell*, *Face*, *Node*, and *BlockFace* objects found within the *Mesh* class. This process is given in detail by the subroutine in Figure 13.

```

(Loop over all blocks)
For  $b_1 = 1, 2, \dots, N_{blocks}$ 
    (Loop over all faces in block  $b_1$ )
    For  $f_1 = 1, 2, \dots, N_{faces}$ 
        (Loop over all cells in face  $f_1$ )
        For  $c_1 = 1, 2, \dots, N_{cells}$ 
            (Loop over all blocks)
            For  $b_2 = 1, 2, \dots, N_{blocks}$ 
                (Loop over all faces in block  $b_2$ )
                For  $f_2 = 1, 2, \dots, N_{faces}$ 
                    (Loop over all cells in face  $f_2$ )
                    For  $c_2 = 1, 2, \dots, N_{cells}$ 
                        If match found between faces  $f_1$  and  $f_2$ 
                            cells  $c_1$  and  $c_2$  are connected
                            Compute orientation
                                If  $N_{dimensions} = 2$  (See Figure 15)
                                    If  $N_{dimensions} = 3$  (See Figure 18)
                                        (End loop over all cells in face  $f_2$ )
                                    (End loop over all faces in block  $b_2$ )
                                (End loop over all blocks)
                                No match found for cell  $c_1$ 
                                Compute extrapolated coordinates for cell  $c_1$ 
                                    (End loop over all cells in face  $f_1$ )
                                (End loop over all faces in block  $b_1$ )
                                (End loop over all blocks)

```

Figure 13. Connectivity subroutine.

The only assumptions made by the preprocessor are that 1) the faces and their corresponding cells along the boundary of a block are the only candidates for finding matches; and 2) no cell can be connected to itself. Other than these two conditions, no other restrictions are enforced by the preprocessor. This allows for several useful topologies including a block sharing a face with multiple other blocks, or even a grid with manifold dimension one defined in two-dimensional space being connected with a grid of manifold dimension two. The same basic algorithm can be applied to any spatial or topological dimension.

When a match is found between two cells, the cell indices along with their respective parent block indices are stored in memory. When all connectivities have been found, this “link” information is then written to file so that it may be used as input for the main solver.

5.5 Orientation

After a match is found between two cells, it is then necessary to find how the two cells are oriented with respect to one another. This is so that when information is transmitted from one cell to another, such as metric information or node coordinates, the data is copied to the correct location. The subroutines for one and two dimensions are different enough as to be treated separately.

5.5.1 Two-Dimensional Subroutine

The two-dimensional subroutine is relatively simple, with four possible orientation configurations as can be seen in Figure 14.

Figure 14. Two-dimensional orientation configurations.

Given two matching cells, their orientation with respect to one another can be determined directly from the indices of their overlapping faces. Employing a simple arithmetic check as outlined in Figure 15, the amount of rotation is dependent upon which faces are overlapping.

As an example, if the index of face f_1 is the same as that of face f_2 , then their respective cells are rotated 180° from each other. Consequently, the resulting algorithm is quite efficient.

```

If  $f_1 = f_2$ 
    cell  $c_2$  is rotated  $180^\circ$  ccw w.r.t. cell  $c_1$ 
Else
If  $f_1 + f_2 = 1$  OR  $f_1 + f_2 = 5$ 
    cell  $c_2$  is rotated  $0^\circ$  ccw w.r.t. cell  $c_1$ 
Else
If  $f_2 - f_1 = 1$  OR  $f_1 + f_2 = 2$  OR  $f_2 - f_1 = 3$ 
    cell  $c_2$  is rotated  $90^\circ$  ccw w.r.t. cell  $c_1$ 
Else
    cell  $c_2$  is rotated  $270^\circ$  ccw w.r.t. cell  $c_1$ 

```

Figure 15. Two-dimensional orientation subroutine.

5.5.2 Three-Dimensional Subroutine

The three-dimensional subroutine is significantly more complex as compared with its two-dimensional counterpart. As can be seen in Figure 16, there exist twenty-four possible orientation configurations. This comes from the fact that each cell has six faces and there are four possible rotations about each face.

Figure 16. Three-dimensional orientation configurations.

Unfortunately, the same arithmetic relations which utilize the face indices as in the two-dimensional case cannot be applied here. This is because there are too many redundancies resulting from the multiple combinations. Instead, the vector and scalar products are used to determine the orientations of the local reference coordinate frames of each cell. What is meant by the local reference frame is the set of normalized vectors, local to each face, which point in the i -, j -, and k -directions. Two of the normalized vectors are computed from the coordinates of opposing edges while the third is the result of the cross product of the first two. This is shown in Figure 17.

Figure 17. Local reference frame for cell face.

With every face having its own reference frame, the orientation of two faces (along with their respective cells) can be found according to the algorithm presented in. It should be noted that the algorithm offers just one of many possibilities to calculating the orientations. Table 1 provides a quick reference for the relation between the angle of orientation between two cells and the resulting values of the scalar and vector products.

```

           $i \cdot i$     $i \times i$ 
Compute  $j \cdot j$     $j \times j$ 
           $k \cdot k$     $k \times k$ 

If  $i \cdot i = 1$ 
    cell  $c_2$  is rotated  $0^\circ$  about the  $k$ -axis w.r.t. cell  $c_1$ 
    If  $j \cdot j = 1$ 
        cell  $c_2$  is rotated  $0^\circ$  about the  $i$ -axis w.r.t. cell  $c_1$ 
    Else If  $j \cdot j = -1$ 
        cell  $c_2$  is rotated  $180^\circ$  about the  $i$ -axis w.r.t. cell  $c_1$ 
    Else If  $j \times j = 1$ 
        cell  $c_2$  is rotated  $90^\circ$  about the  $i$ -axis w.r.t. cell  $c_1$ 
    Else
        cell  $c_2$  is rotated  $270^\circ$  about the  $i$ -axis w.r.t. cell  $c_1$ 
Else If  $i \cdot i = -1$ 
    cell  $c_2$  is rotated  $180^\circ$  about the  $k$ -axis w.r.t. cell  $c_1$ 
    If  $j \cdot j = 1$ 
        cell  $c_2$  is rotated  $0^\circ$  about the  $i$ -axis w.r.t. cell  $c_1$ 
    Else If  $j \cdot j = -1$ 
        cell  $c_2$  is rotated  $90^\circ$  about the  $i$ -axis w.r.t. cell  $c_1$ 
    ...
    etc...

```

Figure 18. Three-dimensional orientation subroutine.

θ	$a \cdot b$	$\ a \times b\ $
0°	+1	0
90°	0	+1
180°	-1	0
270°	0	-1

Table 1. Vector and scalar product results for various orientations.

5.6 Boundary Ghost Cells

If no match is found for a given cell, the cell is then found to lie along the boundary of the entire domain. As a result, there is no real physical information that the corresponding ghost cells can be filled with. Therefore, the ghost cells and, more specifically, their coordinate values must be extrapolated from those of the internal cells. This is accomplished by either a first- or second-order finite-difference extrapolation across the boundary face as shown in Figure 19.

$$\left. \frac{\partial^2 u}{\partial x^2} \right|_{i,j} = u_{i+1,j} - 2u_{i,j} + u_{i-1,j} = 0$$

a) 1st-Order Extrapolation

$$\left. \frac{\partial^3 u}{\partial x^3} \right|_{i,j} = u_{i+2,j} - 3u_{i+1,j} + 3u_{i,j} - u_{i-2,j} = 0$$

b) 2nd-Order Extrapolation

Figure 19. Extrapolated coordinates for boundary ghost cells.

5.7 Complex Topologies

All of the mesh configurations presented thus far have involved simple structured forms with structured layers of ghost cells. This approach works well as long as the implemented numerical solvers require only basic stencils that extend linearly in the i - j -, or k -directions. However, this approach neglects topologies that may be more complex such as five blocks intersecting at the same corner point. And if more involved solvers are desired such as staggered-mesh, or genuinely multi-dimensional solvers, then these complex topologies must be handled in one form or another. As a result, the preprocessor continues to a second “stage” after the inter-block connectivities are found. Here, the preprocessor looks at the coordinates of neighboring (left and right) ghost cells to determine if the ghost cells are connected. If they are, then the preprocessor establishes a connection between the two ghost cells. However, if they do not match, then the preprocessor proceeds through a complicated series of checks to locate their neighbors.

There are basically two subroutines for two different regions of the block. These include ghost cells that lie along the boundaries of each block and those that occupy the corner regions. Figures 20 and 21 illustrate the thirteen possible configurations for the corner regions and the eight possible configurations for the boundary regions respectively. While the theory has been established, the process is partially implemented and has yet to be tested.

Figure 20. Possible corner ghost cell configurations.

Figure 21. Possible boundary ghost cell configurations.

CHAPTER SIX NUMERICAL METHODS

6.1 Introduction

It is the focus of this chapter to outline the set of governing equations used in this research along with the numerical methods used to approximate their solutions. The selected governing equations are presented first, followed by various numerical methods.

6.2 Governing Equations

The specific focus of this research is to resolve complex features in supersonic fluid flow problems, including shockwaves, vortices, and other highly discontinuous structures. As such, the governing equations should include all of the relevant physics necessary to model such features.

For a number of reasons, the Euler equations of gas dynamics have been chosen as the governing equations. The Euler equations are capable of modeling compressible, rotational continuum fluids and therefore contain the mathematics needed to resolve discontinuities and vortices. The Euler equations are also free of viscous effects which are not necessary for resolving the desired features. This greatly simplifies the numerical algorithms and reduces their computational requirements.

In three dimensions, the time-dependent Euler equations in partial differential conservation law form are given by

$$\frac{\partial \mathbf{Q}}{\partial t} + \frac{\partial \mathbf{F}}{\partial x} + \frac{\partial \mathbf{G}}{\partial y} + \frac{\partial \mathbf{H}}{\partial z} = 0 \quad (1)$$

where $\mathbf{Q}$ is the vector of conserved variables and $\mathbf{F}$, $\mathbf{G}$, and $\mathbf{H}$ are the corresponding flux vectors. These four vectors have the explicit form:

$$\mathbf{Q} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E \end{bmatrix}, \quad \mathbf{F} = \begin{bmatrix} \rho u \\ \rho u^2 + p \\ \rho uv \\ \rho uw \\ u(E + p) \end{bmatrix}, \quad \mathbf{G} = \begin{bmatrix} \rho v \\ \rho uv \\ \rho v^2 + p \\ \rho vw \\ v(E + p) \end{bmatrix}, \quad \mathbf{H} = \begin{bmatrix} \rho w \\ \rho uw \\ \rho vw \\ \rho w^2 + p \\ w(E + p) \end{bmatrix} \quad (2)$$

where E is the total energy per unit volume and is given as the sum of the internal and kinetic energies of the fluid. Using the ideal gas relation, E is given by

$$E = \frac{p}{\gamma - 1} + \frac{1}{2} \rho (u^2 + v^2 + w^2) \quad (3)$$

where gamma is the ratio of specific heats.

The conservation laws can also be cast into integral form as

$$\int_{V(t_2)} \mathbf{Q} dV - \int_{V(t_1)} \mathbf{Q} dV + \int_{t_1}^{t_2} \oint_{S_t} \mathbf{n} \cdot \mathbf{F} dS dt = 0 \quad (4)$$

with $\mathbf{Q}$ and $\mathbf{F}$ are as given for the partial differential formulation. Eq. 4 simply states that both the change of the quantities of the $\mathbf{Q}$ vector within the volume V , as well as the flux leaving or entering the volume through the surface S , must sum to zero over the time interval, $\Delta t = t_2 - t_1$. If $\mathbf{Q}$ is continuous in time, then Eq. 4 can be written as

$$\frac{d}{dt} \int_{V(t)} \mathbf{Q} dV + \oint_{S_t} \mathbf{n} \cdot \mathbf{F} dS dt = 0 \quad (5)$$

6.3 Numerical Formulation

Discretization of the partial differential form of the governing laws leads to the finite-difference method, while discretizing the integral form leads to the finite-volume method. All numerical methods presented in this chapter, as well as implemented in the software, have been formulated in accordance with the finite-volume method.

The finite-volume method has been selected over the finite-difference (as well as other discretization techniques) for several different reasons. Firstly, volumes of arbitrary shape and complexity can be utilized to discretize the physical domain when complex geometries must be dealt with. Also, the integral form of the equations guarantees conservation of the variables in $\mathbf{Q}$, namely mass, momentum, and energy. Finally, since the integral equations are solved directly in the physical domain, no coordinate transformation is required. This is a big advantage when the computational expense of such transformations is considered.

A finite-volume discretization of the integral form of the conservation laws leads to

$$\frac{d}{dt}Q = -\frac{1}{V} \sum_{s=1}^N A_s T_s^{-1} \hat{f}_s \quad (6)$$

where N is the total number of surfaces, V is the volume of the cell, A_s is the area of surface s , $\hat{F}_s$ is the numerical flux through surface s , and T_s^{-1} is the inverse rotation matrix as derived in Appendix A.

6.4 Time Discretization

Once the integral form of the conservation laws has been spatially discretized, as in the previous section, a time-marching method can then be applied to fully discretize the equations.

Eq. 6 in the previous section involves discretization of only the spatial terms, leaving the choice for discretizing the temporal derivative still open to consideration. One way of approximating the time derivative is to use a time-marching method to obtain a time-accurate solution. This two-step approach of first discretizing the spatial operators followed by the time derivative is known as the semi-discrete approach and is chosen here due to its simplicity.

If the linear operator in Eq. 6 is replaced by L , the conservation law is simplified to

$$\frac{d}{dt}Q = L \quad (7)$$

where the time derivative of Q must be now approximated.

For this work, only explicit time-marching algorithms have been implemented. They include the explicit Euler method and two TVD Runge-Kutta methods.

6.4.1 Explicit Euler

The explicit Euler method is the simplest choice for temporal discretization. The method uses a first-order forward difference scheme to approximate the time derivative.

$$U_{i,j,k}^{n+1} = U_{i,j,k}^n + \Delta t L \quad (8)$$

The method is a simple and efficient way to advance a time-accurate solution, requiring only one computation of the linear operator per time step. However, the method is only first-order, resulting in a restrictive stability condition.

6.4.2 Total Variation Diminishing Runge-Kutta (TVD RK)

Runge-Kutta time-marching methods are higher-order, multi-step extensions of the explicit Euler method. They achieve high orders of accuracy through a series of predictor and corrector steps that can better estimate the slope (time derivative) of $\mathbf{Q}$.

A second-order Runge-Kutta scheme is given by Eqs. 9 and 10.

$$U_{i,j,k}^{n+1/2} = U_{i,j,k}^n + \Delta t L \quad (9)$$

$$U_{i,j,k}^{n+1} = \frac{1}{2}U_{i,j,k}^n + \frac{1}{2}U_{i,j,k}^{n+1/2} + \Delta t L \quad (10)$$

A third-order Runge-Kutta scheme is given by Eqs. 11-13.

$$U_{i,j,k}^{n+1/3} = U_{i,j,k}^n + \Delta t L \quad (11)$$

$$U_{i,j,k}^{n+2/3} = \frac{3}{4}U_{i,j,k}^n + \frac{1}{4}U_{i,j,k}^{n+1/3} + \Delta t L \quad (12)$$

$$U_{i,j,k}^{n+1} = \frac{1}{3}U_{i,j,k}^n + \frac{2}{3}U_{i,j,k}^{n+2/3} + \Delta t L \quad (13)$$

Both of the schemes presented above were shown to be Total Variation Diminishing (TVD) (Gottlieb, 1996).

The advantages of Runge-Kutta methods is that they are high-order and have increased stability regions over the explicit Euler method. However, this increased

stability comes at the price of requiring multiple computations of the linear operator per time step.

6.5 Numerical Flux

Various schemes for evaluating the numerical flux $\hat{f}$ in Eq. 6 are now presented in detail. The numerical fluxes are approximated through the reconstruction/evolution approach. The reconstruction step is a high-order, one-sided difference that better estimates the left and right states of each interface. This information is then used by the evolution step to determine the interface flux. The accuracy of the resulting scheme is determined by the order of the reconstruction step.

6.5.1 Roe's Flux Difference Splitting (FDS)

Roe's Flux Difference Splitting approximates the numerical flux with one-sided difference schemes that are applied based on the direction of the characteristic wave speeds, $\lambda_{i+1/2}^n$. Backward differencing is used for equations with positive wave speeds, while forward differencing is used for equations with negative wave speeds. This automatically introduces numerical dissipation into the scheme which is necessary for stability. The scheme is given as follows:

$$\hat{f}_{i+1/2}^{Roe} = \frac{1}{2} \left[f(u_i) + f(u_{i+1}) - X_{i+1/2} |\Lambda_{i+1/2}| \alpha_{i+1/2} \right] \quad (14)$$

$$|\Lambda_{i+1/2}| = \begin{bmatrix} |\lambda_{i+1/2}^1| & & \\ & \ddots & \\ & & |\lambda_{i+1/2}^n| \end{bmatrix} \quad (15)$$

$$\alpha_{i+1/2} = X^{-1} \Delta_{i+1/2} u \quad (16)$$

$$\Delta_{i+1/2} u = (u_{i+1} - u_i) \quad (17)$$

The scheme is extremely simple in concept and is easy to implement.

Unfortunately, it does have known issues when trying to resolve flows with strong shock or rarefaction waves, which requires an entropy fix.

6.5.2 Harten-Lax-van Leer-Einfeldt Riemann Solver (HLLE)

One of the short-comings of the Roe scheme is that when the energy of a flow is dominated by kinetic terms—such as in the case of strong rarefactions—the solver fails by predicting non-physical states with negative density or internal energy (Einfeldt, 1991). To remedy this situation, Einfeldt introduced a class of “positively-conservative” schemes, where the positivity of the density and internal energy are guaranteed. His positively-conservative HLLE scheme is given below.

$$u_{i+1/2} = \frac{b_{i+1/2}^r u_{i+1} - b_{i+1/2}^l u_i}{b_{i+1/2}^r - b_{i+1/2}^l} - \frac{f(u_{i+1}) - f(u_i)}{b_{i+1/2}^r - b_{i+1/2}^l} \quad (18)$$

$$\hat{f}_{i+1/2}^{HLLE} = \frac{b_{i+1/2}^+ f(u_i) - b_{i+1/2}^- f(u_{i+1})}{b_{i+1/2}^+ - b_{i+1/2}^-} - \frac{b_{i+1/2}^+ \cdot b_{i+1/2}^-}{b_{i+1/2}^+ - b_{i+1/2}^-} (u_{i+1} - u_i) \quad (19)$$

$$b_{i+1/2}^l = \min(u_{i+1/2} - c_{i+1/2}, u_i - \beta c_i) \quad (20)$$

$$b_{i+1/2}^r = \max(u_{i+1/2} + c_{i+1/2}, u_{i+1} + \beta c_{i+1}) \quad (21)$$

$$\beta = \sqrt{\frac{\gamma-1}{2\gamma}} \quad (22)$$

$$b_{i+1/2}^- = \min(b_{i+1/2}^l, 0) \quad (23)$$

$$b_{i+1/2}^+ = \max(b_{i+1/2}^r, 0) \quad (24)$$

Of note here is the β term given by Eq. 22. Numerical dissipation can be decreased at the expense of a smaller β .

The disadvantage of this scheme is the amount of numerical dissipation that is added due to the maximal and minimal wave speeds needed to satisfy the stability bounds.

6.5.3 Modified HLLE Flux (HLLEM)

Attempting to regain resolution lost in the linearly-degenerate fields by the HLLE scheme, Einfeldt introduced the modified HLLE (HLLEM) scheme (1991). The scheme simply adds anti-diffusion terms to the linearly-degenerate fields so that contact and slip-discontinuities could be better resolved. The scheme is as follows.

$$\hat{f}_{i+1/2}^{HLLEM} = \frac{1}{2} \left[f(u_i) + f(u_{i+1}) - X_{i+1/2} \hat{\Lambda}_{i+1/2} \alpha_{i+1/2} \right] \quad (25)$$

$$\hat{\Lambda}_{i+1/2}^n = \frac{b_{i+1/2}^+ + b_{i+1/2}^-}{b_{i+1/2}^+ - b_{i+1/2}^-} v_{i+1/2}^n - 2 \left(1 - \hat{\delta}_{i+1/2}^n \right) \frac{b_{i+1/2}^+ \cdot b_{i+1/2}^-}{b_{i+1/2}^+ - b_{i+1/2}^-} \quad (26)$$

$$\hat{\delta}_{i+1/2} = \begin{bmatrix} 0 \\ \frac{\hat{c}_{i+1/2}}{\hat{c}_{i+1/2} + |\bar{u}_{i+1/2}|} \\ \frac{\hat{c}_{i+1/2}}{\hat{c}_{i+1/2} + |\bar{u}_{i+1/2}|} \\ \frac{\hat{c}_{i+1/2}}{\hat{c}_{i+1/2} + |\bar{u}_{i+1/2}|} \\ 0 \end{bmatrix} \quad (27)$$

$$\bar{u}_{i+1/2} = \frac{b_{i+1/2}^r + b_{i+1/2}^l}{2} \quad (28)$$

6.5.4 Equilibrium Flux Method (EFM)

The Equilibrium Flux Method (Pullin, 1980) differs from the previously mentioned solvers in that it relies on the Boltzmann equations in the limit of infinite collisions to approximate the flux.

$$\hat{f}_{i+1/2}^{EFM} = f^+(u_i) + f^-(u_{i+1}) \quad (29)$$

$$f^\pm = \begin{bmatrix} \rho(cD + uW) \\ uf_1 + pW \\ vf_1 \\ wf_1 \\ f_1 \left[\frac{V^2}{2} + \frac{\gamma+1}{\gamma-1} \frac{c^2}{4} \right] + \frac{1}{2} puW \end{bmatrix} \quad (30)$$

$$V^2 = u^2 + v^2 + w^2 \quad (31)$$

$$D = \pm \frac{1}{2\sqrt{\pi}} \exp(-s^2) \quad (32)$$

$$W = \frac{1}{2} [1 \pm \text{erf}(s)] \quad (33)$$

$$s = u/c \quad (34)$$

$$c = \sqrt{2RT} = \sqrt{\frac{2p}{\rho}} \quad (35)$$

$$\text{erf}(s) = \frac{2}{\sqrt{\pi}} \int_0^s \exp(-t^2) dt \quad (36)$$

The EFM has the advantage of being positively-conservative, but it is also highly diffusive.

6.5.5 EFM with Osher Flux Splitting (EFMO)

Moschetta and Pullin (1997) attempted to decrease the diffusiveness of the EFM by creating a hybrid with Osher's Flux Difference Splitting scheme as given in the following equations.

$$\hat{f}_{i+1/2}^{EFMO} = \hat{f}_{i+1/2}^{EFM} + \Delta f \quad (37)$$

$$s = u/c \quad \Delta f = \begin{cases} +f^-(u_1) - f^-(u_2) & \text{if } u^* > 0 \\ -f^+(u_1) + f^+(u_2) & \text{otherwise} \end{cases} \quad (38)$$

$$A = a_i + a_{i+1} + \varepsilon \frac{\gamma-1}{2} (u_i - u_{i+1}) \quad (39)$$

$$s_i = p_i / \rho_i^\gamma \quad (40)$$

$$s_{i+1} = p_{i+1} / \rho_{i+1}^\gamma \quad (41)$$

$$\rho_1 = \rho_i \left[\frac{A}{a_i \left(1 + (s_{i+1}/s_i)^{1/2\gamma} \right)} \right]^{2/(\gamma-1)} \quad (42)$$

$$\rho_2 = \rho_1 (s_i/s_{i+1})^{1/\gamma} \quad (43)$$

$$p_1 = p_2 = p^* = s_i \rho_1^\gamma \quad (44)$$

$$a_1 = \sqrt{\gamma p_1 / \rho_1} \quad (45)$$

$$u^* = u_i - \frac{2\varepsilon}{\gamma - 1} (a_1 - a_i) \quad (46)$$

$$u_1 = u_2 = u^* \quad (47)$$

$$v_1 = v_i \quad (48)$$

$$v_2 = v_{i+1} \quad (49)$$

$$w_1 = w_i \quad (50)$$

$$w_2 = w_{i+1} \quad (51)$$

Although it is significantly less diffusive than EFM, EFMO exhibits certain stability issues such as the carbuncle phenomena.

6.5.6 Total Variation Diminishing Flux (TVD)

TVD schemes are obtained by applying a non-oscillatory first-order accurate scheme to an appropriately modified flux function. The resulting schemes are second-order accurate in smooth regions, but degrade to first-order near discontinuities. While numerous TVD schemes have been constructed, two are presented here. The first is originally due to Harten (1983) with further modifications by Cambier (1989; 1990).

$$\hat{f}_{i+1/2}^{TVD} = \frac{1}{2} \left\{ f(u_i) + f(u_{i+1}) + \frac{\Delta x}{\Delta t} X_{i+1/2} \left[g_i + g_{i+1} - \mathcal{Q}(v_{i+1} + \gamma_{i+1}) \alpha_{i+1/2} \right] \right\} \quad (52)$$

$$v_{i+1/2} = \frac{\Delta t}{\Delta x} \lambda_{i+1/2} \quad (53)$$

$$\alpha_{i+1/2} = X^{-1} \Delta_{i+1/2} u \quad (54)$$

$$\Delta_{i+1/2} u = (u_{i+1} - u_i) \quad (55)$$

$$g_i = s_{i+1/2} \max \left[0, \min \left(|\tilde{g}_{i+1/2}|, \tilde{g}_{i-1/2} s_{i+1/2} \right) \right] \quad (56)$$

$$g_i = s_{i+1/2} \max \left[0, \min \left(\mu |\tilde{g}_{i+1/2}|, \tilde{g}_{i-1/2} s_{i+1/2} \right), \min \left(|\tilde{g}_{i+1/2}|, \mu \tilde{g}_{i-1/2} s_{i+1/2} \right) \right] \quad (57)$$

$$s_{i+1/2} = \text{sgn}(\tilde{g}_{i+1/2}) \quad (58)$$

$$\tilde{g}_{i+1/2} = \frac{1}{2} \left[Q(v_{i+1/2}) - (v_{i+1/2})^2 \right] \alpha_{i+1/2} \quad (59)$$

$$\gamma_{i+1/2} = \begin{cases} 0 & \text{if } \alpha_{i+1/2} = 0 \\ (g_{i+1/2} - g_i) / \alpha_{i+1/2} & \text{otherwise} \end{cases} \quad (60)$$

The second version is due to Yee (1985; 1986) and is based on the approximate Riemann solver of Roe, but with a modification to the eigenvalues of the flux Jacobian which yields second-order spatial accuracy.

$$\hat{f}_{i+1/2}^{TVD} = \frac{1}{2} \left[f(u_i) + f(u_{i+1}) - X_{i+1/2} \hat{\Lambda}_{i+1/2} \alpha_{i+1/2} \right] \quad (61)$$

$$\hat{\lambda}_{i+1/2}^n = \begin{cases} \frac{1}{2} \psi(\lambda_{i+1/2}^n) \frac{(g_{i+1}^n + g_i^n)}{\alpha_{i+1/2}^n} - \psi(\lambda_{i+1/2}^n + \beta_{i+1/2}^n) & \alpha_{i+1/2}^n \neq 0 \\ \psi(\lambda_{i+1/2}^n) & \alpha_{i+1/2}^n = 0 \end{cases} \quad (62)$$

$$\beta_{i+1/2}^n = \frac{1}{2} \psi(\lambda_{i+1/2}^n) \begin{cases} (g_{i+1}^n + g_i^n) / \alpha_{i+1/2}^n & \alpha_{i+1/2}^n \neq 0 \\ 0 & \alpha_{i+1/2}^n = 0 \end{cases} \quad (63)$$

$$\psi(z) = \begin{cases} |z| & |z| \geq \varepsilon \\ (z^2 + \varepsilon^2) / 2\varepsilon & |z| < \varepsilon \end{cases} \quad (64)$$

$$g_i^n = \text{minmod}(\alpha_{i-1/2}^n, \alpha_{i+1/2}^n) \quad (65)$$

6.5.7 Piecewise Parabolic Method (PPM)

The Piecewise Parabolic Method discussed here is one of two reconstruction methods used to obtain high-order accuracy in approximating the numerical flux. While originally developed by Woodward and Collela (1986), the PPM given below is due to Huynh (1995). Oscillations in the solution are avoided through devised monotonicity constraints that preserve accuracy near extrema. The resulting scheme is third-order accurate.

The constraints are enforced via the following functions.

$$\text{minmod}(x, y) \equiv \text{median}(x, y, 0) \quad (66)$$

$$\text{minmod}(x, y) = \frac{1}{2} [\text{sgn}(x) + \text{sgn}(y)] \min(|x|, |y|) \quad (67)$$

$$\text{median}(x, y, z) = x + \text{minmod}(y - x, z - x) \quad (68)$$

with the third-order accurate reconstruction and constraints given by

$$u^* = \bar{u}_j + 2(\bar{u}_{j-1} - u_{j-1/2}) \quad (69)$$

$$u_{LAC} = 3\bar{u}_j + \bar{u}_{j-1} - 3u_{j-1/2} \quad (70)$$

$$u_{\min} = \min(\bar{u}_j, u^*, u_{LAC}) \quad (71)$$

$$u_{\max} = \max(\bar{u}_j, u^*, u_{LAC}) \quad (72)$$

$$u_{L,j}^{PPM} = \text{median}(u_{j+1/2}, u_{\min}, u_{\max}) \quad (73)$$

The method also includes a steepening technique that can capture contact discontinuities in two cells. This is given as follows.

$$v_{LIN} = \bar{u}_{j+1} + (\bar{u}_{j+1} - u_{j+3/2}) \quad (74)$$

$$v_{LAC} = 3\bar{u}_j + \bar{u}_{j+2} - 3u_{j+3/2} \quad (75)$$

$$v_{\min} = \min(\bar{u}_{j+1}, v_{LIN}, v_{LAC}) \quad (76)$$

$$v_{\max} = \max(\bar{u}_{j+1}, v_{LIN}, v_{LAC}) \quad (77)$$

$$v_{j+1/2} = \text{median}(u_{j+1/2}, v_{\min}, v_{\max}) \quad (78)$$

$$u_{L,j}^{PPM} = \text{median}(v_{j+1/2}, v_{\min}, v_{\max}) \quad (79)$$

The steepening is applied to only the linearly-degenerate fields, which give rise to contact and slip discontinuities. The steepening is applied to contacts when

$$\frac{d\rho}{dx} \Big/ \frac{dp}{dx} > \beta \quad (80)$$

and to slip discontinuities when

$$\frac{du_t}{dx} \Big/ \frac{du_n}{dx} > \beta \quad (81)$$

where β is usually taken to be 2.

6.5.8 Weighted Essentially Non-Oscillatory (WENO)

Weighted Essentially Non-Oscillatory (WENO) schemes (Jiang, 1996; Ren, 2003; Shi, 2002; Shu, 2001; Titarev, 2004; Zhou, 2001) are based upon Essentially Non-Oscillatory (ENO) schemes in that both schemes utilize the same set of numerical stencils to approximate the flux. The difference between the two is that while the ENO schemes use only one stencil determined by the given conditions, WENO schemes use a convex

combination of all possible stencils. This means that in smooth regions of the solution, the stencils can be combined in such a way as to maximize the order of accuracy, while in regions near discontinuities the stencil containing the discontinuity can be minimized as to prevent the solution from ‘jumping’ across the discontinuity. This gives WENO schemes their desirable ability to resolve complex features in a solution while still preserving discontinuities (Jiang, 1996).

The reconstructed value as calculated by the fifth-order finite volume WENO scheme is given by

$$u_L^{WENO} = \omega_1 u_L^{(1)} + \omega_2 u_L^{(2)} + \omega_3 u_L^{(3)} \quad (82)$$

where $u_L^{(i)}$ are three third reconstructions on three different stencils given by

$$u_L^{(1)} = \frac{1}{3} f(u_{j-2}) - \frac{7}{6} f(u_{j-1}) + \frac{11}{6} f(u_j) \quad (83)$$

$$u_L^{(2)} = -\frac{1}{6} u_{j-1} + \frac{5}{6} u_j + \frac{1}{3} u_{j+1} \quad (84)$$

$$u_L^{(3)} = \frac{1}{3} u_j + \frac{5}{6} u_{j+1} - \frac{1}{6} u_{j+2} \quad (85)$$

and the nonlinear weights ω_i

$$\omega_i = \frac{\tilde{\omega}_i}{\sum_{k=1}^3 \tilde{\omega}_k}, \quad \omega_k = \frac{\gamma_k}{(\beta_k + \varepsilon)} \quad (86)$$

with the linear weights γ_k

$$\gamma_1 = \frac{1}{10}, \quad \gamma_2 = \frac{3}{5}, \quad \gamma_3 = \frac{3}{10} \quad (87)$$

and the smoothness indicators β_k

$$\beta_1 = \frac{13}{12}(u_{j-2} - 2u_{j-1} + u_j)^2 + \frac{1}{4}(u_{j-2} - 4u_{j-1} + 3u_j)^2 \quad (88)$$

$$\beta_2 = \frac{13}{12}(u_{j-1} - 2u_j + u_{j+1})^2 + \frac{1}{4}(u_{j-1} - u_{j+1})^2 \quad (89)$$

$$\beta_3 = \frac{13}{12}(u_j - 2u_{j+1} + u_{j+2})^2 + \frac{1}{4}(3u_j - 4u_{j+1} + u_{j+2})^2 \quad (90)$$

CHAPTER SEVEN VISUALIZATION

7.1 Introduction

Data analysis through visualization is a crucial element for any flow solver. While there exist several state-of-the-art visualization packages for scientific data, many fluid software packages include either a crude visualization module or nothing at all. This creates a severe disconnect between the solution and its visualization which can translate into lost time in analysis and ultimately development. It is therefore an absolute necessity to integrate a fully-capable visualization package with the actual solver.

This is specifically why a package has been coded and incorporated into the solver architecture. The visualization package is capable of rendered, real-time visualization of transient solutions. Fully implemented in the Java language, the package is free of any compatibility concerns or performance issues from visualizing data between different platforms.

7.2 Integration

The visualization package has been integrated into the software through the client side of the architecture. With this particular configuration, the processing of the data for visualization can be kept separate from the master and servers while remaining an integrated component of the architecture. The client simply requests solution data across the network and then processes it into the appropriate graphics.

When the client is initialized, a Graphical User Interface (GUI) is created along with a menu of options including one to initialize the graphical display. If selected, a window will appear within the menu that contains the rendered data. This can be seen in Figure 22. On the right-hand side of the menu, a tabbed pane will appear with several control tabs. The options in this pane give the user a direct link to how the data is rendered, which data is rendered, etc.

Figure 22 Real-time data visualization with Graphical User Interface.

Perhaps the most advanced feature of the visualization package is that since it is integrated into the client side of the architecture, the client application can be killed in the middle of a simulation and then restarted to resume visualization without any discontinuity in the computations. And because of its platform independence, one can start a simulation at his workstation, leave the office on a trip while the simulation continues, and could theoretically reconnect on a PDA or even a cell phone to check the progress of the simulation.

7.3 VisAD API

The client-side visualization package utilizes the Visualization for Algorithm Development (VisAD) package from the University of Wisconsin (Hibbard, 2005). VisAD in turn utilizes the Java3D API. This means that the visualization package is 100% Java compatible making it platform independent. In VisAD, data to be visualized is first organized into classes called `RealTypes` and `RealTupleTypes`. There can be up to 8 `RealTypes` in the way that VisAD is currently implemented. These include two scalar quantities and six vector components for two three-dimensional vector fields. The `RealTypes` in turn are used to construct three `RealTupleTypes`. This is illustrated in Figure 23.

Figure 23 Data mapping for visualization.

The RealTypes are mapped to various DisplayRealTypes such as the color map, streamlines, or range. The grouping of different RealTypes together in the same RealTupleType means that the streamlines for example can be mapped to the height of another scalar quantity on a 3D surface plot. In the representation above, scalar1 will be mapped to the color map, vector1 will be mapped to flow field 1, vector2 will be mapped to flow field 2, and scalar2 will be mapped to the range. Since all RealTupleTypes contain scalar2, the color map and both flow fields will consequently be mapped to the range in the case of a 3D surface plot.

CHAPTER EIGHT BENCHMARKS

8.1 Introduction

The various numerical solvers previously discussed in Chapter Six (see p. 35) are benchmarked here for accuracy, stability, and performance in conjunction with the new parallel architecture. Numerical results obtained with the implemented solvers for standard test cases are presented along with a performance analysis of the parallelization scheme measured in terms of scalability.

8.2 Numerical Solvers

While numerous solvers have been presented in this work, it is the intention here to determine which solvers in particular are best suited for dealing with complex, supersonic flows. As such, the criteria for comparing various solvers will be how well the methods can resolve shockwaves, vortices, and other highly discontinuous structures

The methodology taken here was to test all solvers on the problem of a forward-facing step to determine the best reconstruction and evolution procedures. The resulting reconstruction/evolution combination was then used in the simulation of several other standard test cases.

The choice for time discretization is presented first to establish a uniform time-marching method that can be applied to all numerical flux solvers. This assures an unbiased comparison between the various solvers that may have different requirements to achieve a time-accurate solution.

8.2.1 Time Discretization

As previously mentioned, only explicit time-marching methods have been implemented in the software. These include the first-order accurate Euler and the second- and third-order TVD Runge-Kutta schemes (RK2, RK3). From the standpoint of speed and efficiency, the Euler method is more desirable since it requires only one spatial integration of the domain per time step, whereas the RK methods require two and three respectively. The RK methods are however more stable, permitting larger CFL numbers and requiring fewer time steps to reach the same solution. Also, several high-order spatial discretization methods, such as the WENO and Huynh PPM schemes, require the TVD-RK methods for stability at any CFL number. Initial tests with the PPM solver demonstrate this clearly. Figure 24 shows a solution of the PPM used in conjunction with the Euler method, exhibiting unphysical oscillations throughout the solution.

Figure 24. Numerical solution with explicit Euler time discretization showing oscillations.

When the same problem was computed with the RK2 method replacing the Euler method, the oscillations were not present, as shown in Figure 25.

Figure 25. Numerical solution with 2nd-Order TVD Runge-Kutta time discretization, without oscillations.

Not all numerical solvers produced the oscillations without the TVD Runge-Kutta methods. Nonetheless, the 2nd-Order Runge-Kutta method was used for all tests, with a CFL number of $\frac{1}{2}$, to ensure that all numerical methods were tested under the same conditions.

8.2.2 Mach 3 Flow Past a Forward-Facing Step

The problem of a forward-facing step in a wind tunnel was popularized Woodward and Collela (1984). It involves an initial flow field at Mach 3 everywhere which is incident upon the front of the step. These harsh conditions give rise to a number of complex flow features which can be used as an excellent indication of the capabilities of a particular solver. The problem is used here not only to analyze the various numerical

methods, but to also select the best overall method that will then be used to obtain solutions to other standard test problems. Of primary interest here, is to find a scheme which minimizes the artificial numerical boundary layer that is generated by a slight over-expansion of the gas as it turns the corner of the step.

The grid used to test the various numerical solvers is a simple Cartesian grid, of dimension 300x100 cells.

The solution obtained with Roe's FDS is shown in Figure 26. The method is incapable of producing strong shock or rarefaction waves. As can be seen from the broken bow shock, the scheme suffers from the carbuncle phenomenon. Despite these shortcomings, the scheme does exhibit a nice solution everywhere else, including the slip discontinuity, arising from the triple point shared with the Mach stem and bow shock.

Figure 26. Roe's FDS solution of forward-facing step problem.

The HLLE solver is perhaps the simplest and most robust flux solver. As can be seen in Figure 27, it is also one of the most diffusive. The first Mach stem which connects the front bow shock to the top wall is recessed significantly downstream. Also the, numerical boundary layer is quite pronounced, as the second Mach stem is very large.

Figure 27. HLLE ($\beta = 1$) solution of forward-facing step problem.

A second solution with the HLLE solver is obtained by setting β in Eqs. 20 and 21 equal to zero. As can be seen in Figure 28, this results in a significantly less diffusive scheme. Especially noteworthy is the very small size of the second Mach stem. However, the scheme does produce a non-physical rarefaction shock that emanates from the corner of the step to the top wall.

Figure 28. HLLE ($\beta = 0$) solution of forward-facing step problem.

The EFM solution shown in Figure 29 is remarkably similar to the solution obtained with the HLLE solver with β set to one. Of note are the thick bow shock and the nicely resolved rarefaction wave.

Figure 29. EFM solution of forward-facing step problem.

The EFMO solver solution presented in Figure 30 exhibits a dramatic boundary layer that renders this method essentially unusable. The thick boundary layer interacts with the reflected shock wave causing the shock to break-up and become significantly distorted.

Figure 30. EFMO solution of forward-facing step problem.

The solution of the TVD solver in Figure 31 shows similar behavior to that obtained with the EFMO method. Even though the slip discontinuity is well-resolved, a large numerical boundary layer prevents the correct solution.

Figure 31. TVD solution of forward-facing step problem.

With all of the flux solvers evaluated, a choice can be made as to which one will be utilized for the remainder of the test problems. Even though Roe's FDS produced one of the worst results in terms of stability, these issues can be resolved by using the HLLE solver to evaluate the flux in the presence of strong shocks and rarefaction waves. This method will be referred to from here on as the Hybrid flux. Figure 32 shows the solution obtained with the Hybrid flux. The solution does not exhibit any of the stability issues as seen with the stand-alone Roe flux.

Figure 32. Hybrid solution of forward-facing step problem.

With the choice for the evolution step decided, the reconstruction methods can then be tested. The first of these is Huynh's PPM solver as shown in Figure 33. All flow features are nicely resolved, including the contact and slip discontinuities.

Figure 33. PPM solution of forward-facing step problem.

The solution found using the fifth-order WENO reconstruction is shown in Figure 34.

Figure 34. WENO solution of forward-facing step problem.

Like the PPM method, the WENO scheme is also able to resolve critical features nicely. However, the PPM scheme is better able to resolve contact and slip discontinuities. And because the PPM method requires a smaller computational stencil because of its lower order, its execution time is around $\frac{1}{4}$ as much as that for the WENO method. This all combines to make the PPM more desirable, leading to its use as the reconstruction solver for the remainder of the test cases.

8.2.3 Diffraction of a Shock by a Finite Wedge

The test problem presented here is modeled after an experimental flow field given in the flow album edited by Van Dyke (1982). The problem involves a planar shock wave traveling at Mach 1.3 which diffracts over an equilateral triangle. Fifty blocks containing 100,000 cells each are used to simulate half of the wedge. The conditions behind the shock wave were calculated using the Rankine-Hugoniot relations from Appendix D.

The numerical and experimental results are given in Figure 35. As the shock diffracts over the wedge, an elaborate pattern of shocks, contact discontinuities and slip discontinuities is formed. Of primary interest are the vortices which are generated as the flow separates as it passes the corners of the wedge. The vortex sheets are nicely defined and traces of smaller vortices can be seen to be shedding from the corner as well. These features can be better resolved with a finer grid. Even so, the numerical results are in excellent agreement with the experiment.

Figure 35. Numerical solution of a shock diffracting over a wedge compared with experimental results.

8.2.4 Diffraction of a Shock Down a Step

This test problem is also modeled after an experimental flow field given in the flow album edited by Van Dyke (1982). The problem involves the diffraction of a planar shock wave traveling at Mach 2.4 down a backward-facing step. This has become a popular test problem due to instabilities caused by the strong shock traveling along the upper wall, as well as negative density issues caused by the strong rarefaction wave that is generated from the corner of the step. The computational domain used here is based on a Cartesian grid with 500,000 cells.

Figure 36 shows the solution obtained at three different times, along with their comparable experimental snap shots. The pattern of the incident shock wave, curved diffracted wave, and circular expansion wave grows linearly in time (Van Dyke, 1982). The rarefaction wave is sharply resolved, emanating from the corner at an angle of about 60° from the top of the step, which is in complete agreement with the experimental results.

8.2.5 Sod's Shock Tube Problem

Many shock tube experiments have been done over the years, making this problem especially relevant for a benchmark. Shock tubes have the capacity to provide a shock structure with an associated high-speed, high-energy gas flow and are used to study a wide range of physical problems (Wright, 1961).

Numerically, the shock structure of a shock wave, rarefaction wave, and a contact discontinuity are initialized by simulating a region of high-energy, high-density gas in the

Figure 36. Numerical solution of a shock diffracting down a step compared with experimental results.

center of the domain, surrounded by a region of low-energy, low-density gas. When the simulation is initialized, a rarefaction wave expands into the high-density region, while a shock wave propagates into the low-density region. The two regions are then separated by a contact discontinuity. For all cases, the two regions are defined from the following initial conditions, with the L subscript signifying the low-density, low-energy region and the H subscript signifying the high-density, high-energy region.

$$(\rho_L, u_L, p_L) = (0.125, 0, 0.1) \quad (91)$$

$$(\rho_H, u_H, p_H) = (1, 0, 1) \quad (92)$$

8.2.5.1 One-Dimensional Solution

The one-dimensional case of Sod's problem is a popular benchmark test since the exact solution is known. One hundred cells are used to compute the solution shown in Figure 37. Although the solution appears to be more diffusive than the staggered-mesh version of Huynh, both the shock wave and the contact discontinuity are resolved in about three cells.

8.2.5.2 Two-Dimensional Solution

The two-dimensional case is a simple extension of Sod's original problem. The results shown in Figure 38 were computed on a 400x400 cell Cartesian grid that was decomposed equally into 16 blocks. The region of high density and high energy is extended to a 200x200 cell square region in the center of the domain.

Figure 37 One-dimensional solution of Sod's shock tube problem.

The solutions along the horizontal and vertical axes passing through the center of the domain can essentially be seen as solutions of the one-dimensional Sod problem. This can be seen from the side view of the solution in Figure 39. However, the two-dimensional solution generates rarefaction waves and contact discontinuities, as can be seen in the corners of the top view of the solution in Figure 39.

Figure 38 Two-dimensional solution of Sod's shock tube problem (isometric view).

Figure 39 Two-dimensional solution of Sod's shock tube problem (side and top views).

8.2.5.3 Three-Dimensional Solution

The solution in three dimensions is presented here, not so much as a test for accuracy, but rather to demonstrate the software's 3D capability in terms of both solvers and visualization. Figure 40 shows the solution as obtained from a $20 \times 20 \times 20$ cell Cartesian domain with reflective boundary conditions.

The region of high density and high energy is extended to a cubic region in the center of the domain with dimensions $10 \times 10 \times 10$ cells.

Figure 40 Three-dimensional solution of Sod's shock tube problem.

8.3 Parallel Performance

The parallel efficiency of the architecture is now presented, measured in terms of its scalability. As previously mentioned, the new software architecture is independent of the hardware architecture on which it is executed. In specific, the software is optimized for both distributed-memory and shared-memory computer architectures.

It is worth noting here that while there are many parallel software packages, most are either programmed for distributed-memory or shared-memory architectures, while very few have been programmed for both. The software can run not only on either one configuration or the other, but also a combination of the two, while maintaining the maximum level of optimization.

For example, if the software is configured to run on several different servers of varying processor multiplicity, the code is optimized on each server individually as being a shared-memory architecture and also on the system of connected servers as a whole for being a distributed-memory architecture. The achievement of this type of optimization is greatly aided by the programming language itself—Java. Through the use of its Remote Method Invocation (RMI) protocols as well as its integrated multi-threading capabilities, Java makes software parallelization and concurrent programming as seamless as possible. The following two test cases demonstrate the performance and capabilities of Java and its implementation in the code on distributed- and shared-memory architectures respectively.

8.3.1 Shared-Memory Scalability

The shared-memory parallelization of the code is made possible through the multi-threaded capabilities, inherent in Java. In Java, each object can initiate a new thread by overriding the *run()* method inherent in every object. In this way, each block can run in its own thread, while Java automatically handles all concurrent processing of the threads.

This makes such parallelization efficient, easy, and most importantly, transparent. The transparency means that a domain can be decomposed into as many blocks as necessary, with each block representing one thread, and put on any given server, whether multi-processor or not, knowing that Java will manage the threads to take full advantage of the hardware architecture. In order to show the performance of concurrent processing in Java on a single server node, the following test case is presented. The problem involves a two-dimensional domain containing 129,600 cells, which is decomposed into 32 equally-sized blocks. We then decompose the domain for nine different configurations, consisting of one through eighty-one equally-sized blocks. Each configuration is then used to solve Sod's problem using the TVD transport solver.

The results in Figure 41 below show that the computation time decreases as the number of blocks increase, until the configuration with 25 blocks is reached. For a two-processor node, one might expect that maximum performance would be achieved with two blocks, but this is not the case with Java. The speed-up with multiple blocks can only be explained by the "hyper-threading" capabilities of current processors which are able to efficiently handle multiple threads. It appears as if the concurrent processing is

allowing some threads to process data while others are retrieving data from the cache, thus avoiding a crucial bottleneck.

Figure 41 Single node computation time versus number of decomposed blocks in domain.

8.3.2 Distributed-Memory Scalability

The distributed-memory parallelization of the code is made possible through Java RMI protocols. Remote calls are made from one server to another to exchange boundary and other necessary information. The distributed-memory scalability test demonstrates the speed at which Java RMI allows remote servers to communicate with each other. Again, as in the previous test case, we start with a two-dimensional domain containing 129,600 cells. The fifth-order WENO scheme, outlined in Section 6.5.8, is used to solve Sod's problem in two dimensions over the entire domain. The problem is then solved with five different server configurations, each corresponding to a different number of server nodes, with two processors per node. The first configuration consists of all 32

blocks in the domain being computed on one server node. Each consecutive configuration involves a greater number of server nodes with the 32 blocks divided equally among them until the maximum of 16 nodes is reached. Figure 42 shows the results in terms of computation speed-up versus the number of server nodes used.

Figure 42 Computation speed-up of 32 blocks versus number of nodes.

The speed-up is almost linear, signifying nearly ideal scalability. There is however a notable degradation in efficiency for the 16 node case, that can be seen in Figure 43. There are two possible explanations for this or a combination thereof. The first is that since this test case was performed using the fifth-order WENO scheme, three layers of boundary cells were required to be exchanged between each block at each temporal iteration. This results in a much larger communication overhead than if a first-order scheme, which requires only one layer of boundary cells, was used. The second

explanation stems from the fact that in the 16-node case, one block is computed per processor. It will be evident in the next section that this does not represent the most optimum configuration. To obtain optimal performance, the multi-threaded capabilities of Java must be utilized.

Figure 43 Parallelization efficiency of 32 blocks versus number of nodes.

CHAPTER NINE CONCLUSIONS

9.1 Achievements and Contributions

The primary focus of the current research was to develop and test several algorithms for the simulation of complex flows. These algorithms have been used as the basis in developing a new software that has been tested and validated. The software incorporates advanced techniques such as an Object-Oriented design, platform-independent parallelization, and real-time visualization, making it unique among its peers.

It has been shown that despite its Object-Oriented design, the software can still achieve high performance, dismissing previous notions that Java is not suitable for scientific computing. What is particularly promising, is the scalability efficiency the software was able to achieve on both shared-memory and distributed-memory networks.

Robustness and accuracy were also proven for the implemented numerical solvers, as shown by their solutions for several test cases.

9.2 Recommendations for Future Work

Even though the software developed in this research has been proven to be highly capable, there is still much room for improvement in terms of refinement, optimization, and further extension to more advanced physical modeling. These extensions, however, should be readily achievable, due to the level of effort already put into the software architecture. The most important improvements are outlined below.

The parallelization of the software can be improved upon by increasing the flexibility of the Client-Master-Server framework through added functionality of each component. This can include allowing servers to become additional masters themselves, that can then distribute and manage tasks of other servers. Further efficiency of the parallelization can be achieved by implementing load-balancing algorithms which can further take advantage of available computational resources.

Improvements to the geometry package, such as the inclusion of adaptive mesh refinement (AMR) techniques, will greatly increase the efficiency and accuracy of the numerical solvers. The numerical solvers themselves can also be further refined to increase accuracy and resolution of their solutions.

REFERENCES

- Billett, S. J., & Toro, E. F. (1997). Unsplit WAF-Type Schemes for Three-Dimensional Hyperbolic Conservation Laws. In E. F. Toro & J. Clark, (Eds.), In *Numerical Methods for Wave Propagation Problems* (pp. 75-124), Kluwer Academic Publishers.
- Cambier, J.-L., Adelman, H. G., & Menees, G. P. (1989). Numerical Simulations of Oblique Detonations in Supersonic Combustion Chambers, *J. Prop. and Power*, 5, 482-491.
- Cambier, J.-L., Adelman, H. G., & Menees, G. P. (1990). Numerical Simulations of an Oblique Detonation Wave Engine, *J. Prop. and Power*, 26, 315-323.
- Cambier, J.-L., Carroll, M., & Kapper, M. (2004). Development of a Hybrid Model for Non-Equilibrium High-Energy Plasmas, AIAA paper 2004-2166.
- Deng, X., & Zhang, H. (2000). Developing High-Order Weighted Compact Nonlinear Schemes, *J. Comp. Physics*, 165, 22-44.
- Einfeldt, B., Munz, C. D., Roe, P. L., & Sjögren, B. (1991). On Godunov-Type Methods near Low Densities, *J. Comp. Physics*, 92, 273-295.
- Gottlieb, S., & Shu, C.-W. (1996). Total Variation Diminishing Runge-Kutta Schemes, ICASE 1996-50.
- Grosso, W. (2002). *Java RMI*, O'Reilly & Associates, Inc., Sebastopol.
- Harten, A. (1983). High Resolution Schemes for Hyperbolic Conservation Laws, *J. Comp. Physics*, 49, 357-393.
- Harten, A. (1989). ENO Schemes with Subcell Resolution, *J. Comp. Physics*, 83, 148-184.
- Häuser, J. (1999). A Pure Java Parallel Flow Solver, AIAA paper 99-0549.
- Horstmann, C. S., & Cornell, G. (1999). *Core Java 1.2, Volume I—Fundamentals*, Sun Microsystems Press, San Jose.
- Horstmann, C. S., & Cornell, G. (1999). *Core Java 1.2, Volume II—Advanced Features*, Sun Microsystems Press, San Jose.
- ASC FLASH Center. (2004). FLASH User's Guide v. 2.4 <http://flash.uofchicago.edu>.

University of Chicago.

Hibbard, B. (2005). VisAD Home Page. <http://www.ssec.wisc.edu/~billh/visad.html>.

Hu, X. Y., & Khoo, B. C. (2003). Kinetic Energy Fix for Low Internal Energy Flows, *J. Comp. Physics*, 193, 243-259.

Huynh, H. T. (1995). Accurate Upwind Schemes for the Euler Equations, AIAA paper 95-1737-CP.

Huynh, H. T. (1995). A Piecewise-Parabolic Dual-Mesh Method for the Euler Equations, AIAA paper 95-1739-CP.

Hyde, P. (1999). *Java Thread Programming*, Sams Publishing, Indianapolis.

Jiang, G.-S., & Shu, C.-W. (1996). Efficient Implementation of Weighted ENO Schemes, *J. Comp. Physics*, 126, 202-228.

Kim, D., & Kwon, H. (2004). A Low Dissipative and Dispersive Scheme with a High Order WENO Dissipation for Unsteady Flow Analyses, AIAA paper 2004-2705.

Kim, S., Kim, C., Rho, O.-H., & Hong, S. K. (2003). Cures for the Shock Instability: Development of a Shock-Stable Roe Scheme, *J. Comp. Physics*, 185, 342-374.

Lomax, H., Pulliam, T. H., & Zingg, D. W. (2000). *Fundamentals of Computational Fluid Dynamics*, Springer-Verlag.

Markidis, S., Lapenta, G., Vanderheyden, W. B., & Budimlić, Z. (2004) Implementation and Performance of a Particle-In-Cell Code Written in Java, *Concurrency and Computat.: Pract. and Exper.*, 16, 1-17.

Markidis, S., Lapenta, G., & Vanderheyden, W. B. Parsek: Object Oriented particle-in-Cell Implementation and performance Issues, To be published in *Concurrency and Computat.: Pract. and Exper.*

Moschetta, J.-M., & Pullin, D. I. (1997). A Robust Low Diffusive Kinetic Scheme for the Navier-Stokes/Euler Equations, *J. Comp. Physics*, 133, 193-204.

Osher, S., & Chakravarthy, S. (1983). Upwind Schemes and Boundary Conditions with Applications to Euler Equations in General Geometries, *J. Comp. Physics*, 50, 447-481.

Padial-Collins, N. T., Vanderheyden, W. B., Zhang, D. Z., Dendy, E. D., & Livescu, D. (2004). Parallel Operation of CartaBlanca on Shared and Distributed Memory Computers, *Concurrency and Computat.: Pract. and Exper.*, 16, 1-17.

- Park, S. H., & Kwon, J. H. (2003). On the Dissipation Mechanism of Godunov-type Schemes, *J. Comp. Physics*, 188, 524-542.
- Prendergast, K. H., & Xu, K. (1993). Numerical Hydrodynamics from Gas-Kinetic Theory, *J. Comp. Physics*, 109, 53-66.
- Proceedings of the ACM Workshop on Java for High Performance Network Computing, Stanford University, Palo-Alto, CA, Feb. 28-Mar-1, 1998.
- Pullin, D. I. (1980). Direct Simulation Methods for Compressible Inviscid Ideal Gas Flow, *J. Comp. Physics*, 34, 231-244.
- Quirk, J. J. (1992). A Contribution to the Great Riemann Solver Debate, ICASE Report No. 1992-92.
- Ren, Y.-X., Liu, M., & Zhang, H. (2003). A Characteristic-Wise Hybrid Compact-WENO Scheme for Solving Hyperbolic Conservation Laws, *J. Comp. Physics*, 192, 365-386.
- Shi, J., Hu, C., & Shu, C.-W. (2002). A Technique of Treating Negative Weights in WENO Schemes, *J. Comp. Physics*, 175, 108-127
- Shu, C.-W. (2001). High Order Finite Difference and Finite Volume WENO Schemes and Discontinuous Galerkin Methods for CFD, ICASE Report No. 2001-11.
- Tannehill, J. C., Anderson, D. A., & Pletcher, R. H. (1997). *Computational Fluid Mechanics and Heat Transfer*, Taylor & Francis.
- Titarev, V. A., & Toro, E. F. (2004). Finite-Volume WENO Schemes for Three-Dimensional Conservation Laws, *J. Comp. Physics*, 201, 238-260.
- Toro, E.F. (1997). *Riemann Solvers and Numerical Methods for Fluid Dynamics*, Springer-Verlag, New York.
- Van Dyke, M. (1982). *An Album of Fluid Motion*, Parabolic Press, Inc., Stanford.
- Vinokur, M. (1989). An Analysis of Finite-Difference and Finite-Volume Formulations of Conservation Laws, *J. Comp. Physics*, 81, 1-52.
- Woodward, P. R., & Collela, P. (1984). Numerical Simulations of Two-Dimensional Fluid Flow with Strong Shocks, *J. Comp. Physics*, 54, 115-173.
- Woodward, P. R., & Collela, P. (1984). The Piecewise Parabolic Method (PPM) for Gas-

Dynamical Simulations”, *J. Comp. Physics*, 54, 174-201.

Wright, J. K. (1961) *Shock Tubes*, John Wiley & Sons, Inc., New York.

Xu, K., & Prendergast, K. (1994). Numerical Navier-Stokes Solutions from Gas-Kinetic Theory”, *J. Comp. Physics*, 114, 9-17.

Xu, K., Martinelli, L., & Jameson, A. (1995). Gas-Kinetic Finite-Volume Methods, Flux-Vector Splitting, and Artificial Diffusion, *J. Comp. Physics*, 120, 48-65.

Yee, H. (1985). Implicit TVD Schemes for Hyperbolic Conservation Laws in Curvilinear Coordinates, AIAA Paper 85-1513.

Yee, H. (1986). *Preceedings of the 6th GAMM Conference on Numerical Methods in Fluid Mecanics*, Gottingen, Federal Republic of Germany.

Zhou, T., Li, Y., & Shu, C.-W. (2001). Numerical Comparison of WENO Finite Volume and Runge-Kutta Discontinuous Galerkin Methods, *Journal of Scientific Computing*, 16 (2): 145-171.

APPENDIX A METRIC TRANSFORMATIONS

When the finite-volume method is used to discretize the Euler equations, a transformation of the velocity components is needed at each cell interface.

A.1. Two-Dimensional Finite-Volume Metrics

Given an interface rotated an angle θ about the x-axis as in Figure 44, the Finite-Volume approach requires the normal and tangential velocity components to the face.

Figure 44. Rotation transformation of two-dimensional interface.

After calculating the length of the interface ds by

$$ds = |\overline{01}| \quad (93)$$

the cosine and sine functions of the angle θ , which define the unit vector normal to the interface, are obtained via Eqs. 92 and 93 respectively.

$$\cos \theta = \frac{\overline{01}_x}{ds} \quad (94)$$

$$\sin \theta = -\frac{\overline{01}_y}{ds} \quad (95)$$

$$\mathbf{n} = \langle \cos \theta, \sin \theta \rangle \quad (96)$$

The transformation and inverse transformation can be stated in terms of the components of the normal vector, $\mathbf{n}$.

$$\mathbf{T}(\theta) = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \quad (97)$$

$$\mathbf{T}^{-1}(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta \\ \sin \theta & \cos \theta \end{bmatrix} \quad (98)$$

For any two-dimensional velocity vector $\mathbf{V}$, the velocity components normal and tangential to the interface can be found by multiplication with the transformation matrix $\mathbf{T}$.

$$\hat{\mathbf{V}} = \mathbf{T}(\theta) \mathbf{V} = \begin{bmatrix} \cos \theta & \sin \theta \\ -\sin \theta & \cos \theta \end{bmatrix} \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} u \cos \theta + v \sin \theta \\ v \cos \theta - u \sin \theta \end{bmatrix} = \begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} \quad (99)$$

In Eq. 97, $\hat{u}$ is the velocity component normal to the interface while $\hat{v}$ is the component tangential to the face. The components can then be transformed back by multiplication with the inverse transformation matrix $\mathbf{T}^{-1}$.

$$\mathbf{V} = \mathbf{T}^{-1}(\theta) \hat{\mathbf{V}} \quad (100)$$

A.2. Three-Dimensional Finite-Volume Metrics

The three-dimensional metrics require two angles, ϕ and θ , which define the orientation of the interface with respect to the y- and z-axes, as shown in Figure 45.

Figure 45. Rotation transformation of three-dimensional interface.

If the unit normal to the interface is given as

$$\mathbf{n} = \langle n_0, n_1, n_2 \rangle \quad (101)$$

then the sine and cosine components of both angles ϕ and θ can be found from the following four equations.

$$\sin \phi = n_2 \quad (102)$$

$$\cos \phi = \sqrt{1 - \sin^2 \phi} \quad (103)$$

$$\sin \theta = \frac{n_1}{\cos \phi} \quad (104)$$

$$\cos \theta = \frac{n_0}{\cos \phi} \quad (105)$$

The transformation and inverse transformation matrices can then be expressed in a similar fashion as for the two-dimensional case.

$$\mathbf{R}_Y(\phi) = \begin{bmatrix} \cos \phi & \sin \phi & 0 \\ 0 & 1 & 0 \\ -\sin \phi & \cos \phi & 0 \end{bmatrix} \quad (106)$$

$$\mathbf{R}_Z(\theta) = \begin{bmatrix} \cos \theta & \sin \theta & 0 \\ -\sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (107)$$

$$\mathbf{T}(\theta, \phi) = \mathbf{R}_Y(\phi) \mathbf{R}_Z(\theta) = \begin{bmatrix} \cos \phi \cos \theta & \cos \phi \sin \theta & \sin \phi \\ -\sin \theta & \cos \theta & 0 \\ -\sin \phi \cos \theta & -\sin \phi \sin \theta & \cos \phi \end{bmatrix} \quad (108)$$

$$\mathbf{R}_Y^{-1}(\phi) = \begin{bmatrix} \cos \phi & -\sin \phi & 0 \\ 0 & 1 & 0 \\ \sin \phi & \cos \phi & 0 \end{bmatrix} \quad (109)$$

$$\mathbf{R}_Z^{-1}(\theta) = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \quad (110)$$

$$\mathbf{T}^{-1}(\theta, \phi) = \mathbf{R}_Z^{-1}(\theta) \mathbf{R}_Y^{-1}(\phi) = \begin{bmatrix} \cos \phi \cos \theta & -\sin \theta & -\sin \phi \cos \theta \\ \cos \phi \sin \theta & \cos \theta & -\sin \phi \sin \theta \\ \sin \phi & 0 & \cos \phi \end{bmatrix} \quad (111)$$

APPENDIX B BOUNDARY CONDITIONS

Boundary conditions must be enforced for the physical domain to ensure proper flux through the cell faces which lie along the boundaries of the domain. Ghost cells are used in the software to allow the boundary conditions to be given by the appropriate conserved quantities, which will provide the desired flux at the cell interfaces. Determining the proper values of the conserved variables for the ghost cells is then the focus of the appendix. There are three types of boundary conditions which have been included in the code—reflective, inflow, and outflow—and are discussed in the following.

B.1. Reflective Boundary Conditions

If the collisions in a fluid are assumed to be elastic, then a fluid which strikes a surface with an angle θ will be reflected at the same angle as shown in Figure 46.

Figure 46. Reflective boundary condition.

If the conserved variables next to a wall is given by

$$\mathbf{Q}_{incident} = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ E \end{bmatrix} \quad (112)$$

the velocity components can be extracted to the following velocity vector.

$$\mathbf{V}_{incident} = \begin{bmatrix} u \\ v \end{bmatrix} \quad (113)$$

The components of the vector can then be rotated (see) to the reference frame of the wall as follows.

$$\mathbf{V}_{rotated} = \mathbf{T}(\theta) \begin{bmatrix} u \\ v \end{bmatrix} = \begin{bmatrix} \hat{u} \\ \hat{v} \end{bmatrix} \quad (114)$$

The velocity component normal to the wall, $\hat{u}$, is negated, and the resulting vector can then be rotated back to the physical reference frame

$$\mathbf{V}_{reflected} = \mathbf{T}^{-1}(\theta) \begin{bmatrix} -\hat{u} \\ \hat{v} \end{bmatrix} = \begin{bmatrix} u' \\ v' \end{bmatrix} \quad (115)$$

The proper conserved variables for the corresponding ghost cell can the be reconstructed.

$$\mathbf{Q}_{reflected} = \begin{bmatrix} \rho \\ \rho u' \\ \rho v' \\ E \end{bmatrix} \quad (116)$$

B.2. Inflow Boundary Conditions

The inflow boundary conditions are perhaps the simplest of all boundary conditions. The conservative variables for the ghost cells are simply given by the inflow boundary conditions.

$$Q_{\text{inf } low} = \begin{bmatrix} \rho_{\infty} \\ \rho_{\infty} u_{\infty} \\ \rho_{\infty} v_{\infty} \\ E_{\infty} \end{bmatrix} \quad (117)$$

B.3. Outflow Boundary Conditions

The outflow boundary conditions used in the code have been for the case of supersonic flow. That is, the conserved variables in the ghost cells are simply extrapolated from the nearest inside cell. This results in the derivative of the conserved variable being equal to zero across the boundary.

APPENDIX C

HIGH PERFORMANCE BLAS

Attempting to match execution speeds with those of the *C* programming language, the software makes extensive use of the *BLAS* (Basic Linear Algebra Subroutines) class developed specifically for this work. With a mix of vector and matrix multiplications as well as specialized functions, *BLAS* has proven its performance against the same algorithms implemented in optimized *C*. This appendix highlights some of the key features of the algorithms as well as a matrix multiplication performance comparison.

C.1. Subroutines

Presented in Figure 47 is the complete matrix multiplication method coded in the Java version of *BLAS*. Pointers to the row vectors of each matrix are used to reduce the accessing of data from the arrays, resulting in an efficient algorithm. The same method coded in *C* is given in Figure 48.

C.2. Performance

Both the Java and *C* methods were tested for performance by calling the multiplication methods 100 times as essentially a “warm-up” period followed by 100 additional calls that were timed. The matrices were filled with double floating-point values that were randomly generated. The Java code was compiled with compiler version 1.4.2_04 and executed with the `-server` JVM of the same version. The *C* code was compiled with the `-O3` and `-funroll-all-loops` options of the gcc compiler version 2.96.

All tests were performed on the Intel Pentium 4 architecture running the RedHat Linux operating system. Figure 49 illustrates the performance results.

```
public static double[][] mul(double[][] a, double[][] b)
{
    int m = a.length;
    int l = a[0].length;
    int n = b[0].length;

    double[][] c = new double[m][n];

    for(int i=0;i<m;i++)
    {
        double[] ai = a[i];
        double[] ci = c[i];
        for(int k=0;k<l;k++)
        {
            double aik = ai[k];
            double[] bk = b[k];
            for(int j=0;j<n;j++)
            {
                ci[j] += aik*bk[j];
            }
        }
    }
    return c;
}
```

Figure 47. Optimized matrix multiplication subroutine in Java.

```
void mul(double a[][], double b[][])
{
    for(int i=0;i<N;i++)
    {
        double *ci = *(c+i);
        double *ai = *(a+i);
        for(int k=0;k<N;k++)
        {
            double aik = ai[k];
            double *bk = *(b+k);
            for(int j=0;j<N;j++)
            {
                ci[j] += aik*bk[j];
            }
        }
    }
    return c;
}
```

Figure 48. Optimized matrix multiplication subroutine in C.

Figure 49. Performance comparison of Java and C matrix multiplication.

As the results show, Java and C performance for the matrix multiplication problem are nearly identical. This highlights the success of the BLAS methods implemented in Java as well as the ability of Java to perform on par with C, albeit on the Intel platform.

APPENDIX D RANKINE-HUGONIOT RELATIONS

For some of the numerical benchmarks presented in this work, it was necessary to simulate a planar shock wave traveling at some prescribed velocity through a quiescent flow field. Given the speed of the shock and the conditions ahead of the shock it was necessary to calculate the conditions upstream of the shock. This was accomplished by using the Rankine-Hugoniot relations summarized in this appendix. The Rankine-Hugoniot relations specify the conditions that must be satisfied by the physical parameters on both sides of a discontinuity.

For a planar shock wave traveling from left to right as illustrated in Figure 50,

Figure 50. Two flow states separated by a shock wave.

Eqs. 118-122 can be used to find the conditions upstream of the shock, given the conditions downstream along with the speed of the shock.

$$M_R = \frac{u_R}{a_R} \tag{118}$$

$$M_* = \frac{u_*}{a_R} \quad (119)$$

$$\rho_L = \rho_R \frac{(\gamma+1)(M_R - M_*)^2}{(\gamma-1)(M_R - M_*)^2 + 2} \quad (120)$$

$$p_L = p_R \frac{2\gamma(M_R - M_*)^2 - (\gamma-1)}{(\gamma+1)} \quad (121)$$

$$u_L = \frac{\rho_R}{\rho_L} (u_R - M_* a_R) + M_* a_R \quad (122)$$

APPENDIX E

NUMERICAL SCHLIEREN IMAGING

Most of the numerical solutions in this thesis have been presented as simulated schlieren-type images. This type of visualization technique is valuable when trying to analyze compressible flows with such features as shock waves, rarefaction waves, and vortices. These features are associated with high gradients that can be exploited to accentuate their characteristics. This is done by mapping the magnitude of the gradient of the density field to a grayscale colormap. To magnify smaller gradients, a nonlinear scaling was applied to the normalized gradient magnitude. An adjustable parameter is also provided to strengthen or weaken the nonlinear effect as necessary, subject to flow conditions.

The magnitude of the gradient is determined by

$$\|\nabla\rho\| = \sqrt{\left(\frac{\partial\rho}{\partial x}\right)^2 + \left(\frac{\partial\rho}{\partial y}\right)^2} \quad (123)$$

while the nonlinear shading function implemented is given by

$$shading = \exp\left(-k \frac{|\nabla\rho|}{|\nabla\rho|_{max}}\right) \quad (124)$$

where $|\nabla\rho|_{max}$ is the maximum gradient magnitude taken over the entire domain and k is an adjustable parameter. The parameter k was taken to be 10 for all simulations presented in this work.