[redacted]

INEL-96/125

May, 1996

Idaho National Engineering Laboratory

Scalability of Preconditioners as a Strategy for Parallel Computation of Compressible Fluid Flow

G. A. Hansen

Lockheed Idaho Technologies Company

DISCLAIMER

INEL--96/125
INEL-96/125

Scalability of Preconditioners as a Strategy for Parallel Computation of Compressible Fluid Flow

Glen A. Hansen

Published May 1996

Idaho National Engineering Laboratory
Lockheed Martin Idaho Technologies
Idaho Falls, Idaho 83415

Supported by the
U.S. Department of Energy
through DOE Idaho Operations Office
Contract DE-AC07-94ID13223

SCALABILITY OF PRECONDITIONERS AS A STRATEGY FOR PARALLEL COMPUTATION OF COMPRESSIBLE FLUID FLOW

A Dissertation

Presented in Partial Fulfillment of the Requirements for the

Degree of Doctor of Philosophy

with a

Major in Computer Science

in the

College of Graduate Studies

University of Idaho

by

Glen A. Hansen

April 5, 1996

Major Professors: John W. Dickinson, Ph.D.

Eugene Saghi, Ph.D.

AUTHORIZATION TO SUBMIT

DISSERTATION

This dissertation of Glen A. Hansen, submitted for the degree of Doctor of Philosophy and titled "Scalability of Preconditioners as a Strategy for Parallel Computation of Compressible Fluid Flow," has been reviewed in final form, as indicated by the signatures and dates given below. Permission is now granted to submit final copies to the College of Graduate Studies for approval.

Major Professor	John Dickinson John W. Dickinson	Date	4/11/96
Co-Advisor	Eugene Saghi Eugene Saghi	Date	4/11/96
Committee Members	Rod W. Douglass Rod W. Douglass	Date	4/11/96
	Dana Knoll Dana A. Knoll	Date	4/11/96
	Michael Barnett Michael Barnett	Date	4/11/96
Department Administrator	John Dickinson John W. Dickinson	Date	4/11/96
Engineering College Dean	_____ Richard T Jacobsen	Date	_____

Final Approval and Acceptance by the College of Graduate Studies

_____ Date _____

Jean'ne M. Shreeve

Abstract

Parallel implementations of a Newton-Krylov-Schwarz algorithm are used to solve a model problem representing low Mach number compressible fluid flow over a backward-facing step. The Mach number is specifically selected to result in a numerically "stiff" matrix problem, based on an implicit finite volume discretization of the compressible 2D Navier-Stokes/energy equations using primitive variables. Newton's method is used to linearize the discrete system, and a preconditioned Krylov projection technique is used to solve the resulting linear system. Domain decomposition enables the development of a global preconditioner via the parallel construction of contributions derived from subdomains. Formation of the global preconditioner is based upon additive and multiplicative Schwarz algorithms, with and without subdomain overlap. The degree of parallelism of this technique is further enhanced with the use of a matrix-free approximation for the Jacobian used in the Krylov technique (in this case, GMRES(k)).

Of paramount interest to this study is the implementation and optimization of these techniques on parallel shared-memory hardware, namely the Cray C90 and SGI Challenge architectures. These architectures were chosen as representative and commonly available to researchers interested in the solution of problems of this type. The Newton-Krylov-Schwarz solution technique is increasingly being investigated for computational fluid dynamics (CFD) applications due to the advantages of full coupling of all variables and equations, rapid nonlinear convergence, and moderate memory requirements. A parallel version of this method that scales effectively on the above architectures would be extremely attractive to practitioners, resulting in efficient, cost-effective, parallel solutions exhibiting the benefits of the solution technique.

The multiplicative Schwarz preconditioner did not yield performance advantages over the additive Schwarz version due to the coloring technique mandated for parallelism. Subdomain overlap also was not effective in providing solution scalability due to the extreme memory requirements of the method. A solution technique based on a parallel Jacobian formation algorithm, additive Schwarz preconditioning, and a matrix-free implementation did provide excellent performance on 8 C90 processors and 4 SGI processors when pseudo-transient continuation was employed as a check on the increase of linear iterations with the number of subdomains.

Acknowledgments

I wish to thank the many people that were involved in the development of this work.

Dr. Michael Barnett of Microsoft, Inc. acted as my advisor throughout the first half of this research while he was on the faculty of the University of Idaho. For the latter half of this work, Drs. Eugene Saghi and John Dickinson of the University of Idaho shared the advising duties.

I am grateful to the remaining members of my committee, Drs. Dana Knoll and Rod Douglass of the Idaho National Engineering Laboratory (INEL). Dr. Knoll provided the inspiration and considerable technical guidance toward this work; this task would have been impossible without his pioneering research in Newton-Krylov-Schwarz methods. I greatly appreciate Dr. Douglass' efforts in the review of several preliminary drafts of this thesis, suggestions for improvements, and countless hours of discussions about how a dissertation should be constructed.

I wish to thank several people at the INEL, Drs. Paul McHugh and Paul Jacobs, and Mr. Vince Mousseau, for all the technical help that they provided towards this work. I also wish to thank Dr. John Ramshaw, Science and Engineering Fellow at the INEL, for his many suggestions and support.

I am indebted to the INEL Long Term Research Initiative in Computational Mechanics (LTRI-CM), headed by Dr. Rod Douglass, for the financial support of this effort and many of the facilities used for this research. Cray Research, Inc. (Mr. Steve Baumann, Dr. Tom Ashbrook, and Dr. James Harcourt) provided the Cray time for development and countless hours of parallel execution at their facilities in Minneapolis. Without the support of the INEL LTRI and Cray, a study of this magnitude could not have been attempted.

I am grateful to the University of Idaho and the Computer Science Department for allowing me to pursue a Ph.D. degree in Idaho Falls. Additionally, I thank the department for the opportunity to develop and teach the course in *C++* at the Idaho Falls campus.

In closing, I wish to thank my wife, Jeanne, for her understanding and moral support during the course of this work. Finally, I would also like to express my appreciation to Jerold Klug P.E., for his inspiration and guidance into my first engineering study.

Contents

Authorization Form ii

Abstract iii

Acknowledgements v

Table of Contents vii

List of Figures xi

List of Tables xiv

List of Symbols xvii

1 Introduction 1

1.1 Solution of the Navier-Stokes Equations 3

1.2 Research Overview 9

1.2.1 History and Related Work 10

1.2.2 Mathematical Overview 15

1.2.3 Research Outline 22

1.2.4 Alternatives 24

1.3 Summary of Procedures and Results 26

2 The Mathematical Basis **29**

2.1 The Backward-Facing Step Problem 29

2.2 The Governing Equations 34

2.2.1 Non-Dimensionalization of the Governing Equations 38

2.2.2 Discretization of the Governing Equations 40

2.2.3 The Finite Volume Approximation of the Governing Equations 42

2.3 Boundary Conditions 63

2.4 The Non-linear Algebraic System of Equations 67

3 Solution of the Non-linear Algebraic System **71**

3.1 Newton's Method 72

3.2 The Inexact Newton's Method 75

3.3 Preconditioning 77

3.3.1 Additive Schwarz Preconditioning 81

3.3.2 Multiplicative Schwarz Preconditioning 85

3.3.3 Preconditioning of the Model Problem 91

3.4 Krylov Subspace Algorithms 96

3.4.1 Transpose-Free Quasi-Minimal Residual Method (TFQMR) 100

3.4.2 Generalized Minimal Residual Method (GMRES) 110

3.5 The Matrix-Free Technique 112

3.6 Mechanics 114

3.7 Summary 117

4 The Additive Schwarz Preconditioner **120**

4.1 Architecture Overview 121

4.1.1 Cray Optimization 122

4.1.2 SGI Optimization 124

4.2 Initial Results 125

4.2.1 The Jacobian Algorithm 131

4.2.2 The Preconditioner 133

4.3 Jacobian Granularity and Contention 139

4.4 Subdomain Overlap with Additive Schwarz 148

4.5 Summary 156

5 The Multiplicative Schwarz Preconditioner **159**

5.1 Results 162

5.2 Summary 165

6 The Matrix-Free Technique **167**

6.1 Robustness Concerns 171

6.2 Performance of the Matrix-Free Technique 172

6.3 Summary 182

7 Conclusions **184**

7.1 Optimal Architecture 186

7.2 Summary of Results and Future Research Topics 193

A Some Mechanics of Shared Memory Parallel Computation **200**

A.1 Applied Parallel Computation 201

A.2 Hardware Selection for Applied Parallel Computation 203

A.2.1 Requirements 205

A.2.2 The Optimal Architecture 210

A.2.3 The Comparison 211

A.2.4 Final Thoughts 215

A.3 Shared Memory Hardware Programming Basics 216

A.3.1 Cray Optimization Process 218

A.3.2 SGI Optimization Process 230

A.4 Parallel Processing In A Production Environment 235

B Sample Cray FLOWTRACE Output **238**

Bibliography **242**

List of Figures

1.1 The problem domain. 17

2.1 The backward-facing step. 30

2.2 Flow velocity. 30

2.3 Mach number contours. 31

2.4 Pictorial representation of a finite volume, Ω_e, assuming a rectilinear two-dimensional discretization. 44

2.5 The computational cell used for the development of the mass conservation equation approximation. 45

2.6 The computational cell modified for coincident velocity and density. 47

2.7 The x-momentum stencil. 52

2.8 The y-momentum stencil. 56

2.9 The energy stencil. 60

2.10 The problem domain. 64

2.11 No-slip y-momentum condition along an east wall. 66

2.12 Adiabatic temperature condition along an east wall. 66

3.1 The structure of the Jacobian matrix. 75

3.2 Simplified Jacobian matrix. 79

3.3 Partitioned Jacobian matrix, four subdomains. 80

3.4 Magnified subdomain. 80

3.5 Partitioned Jacobian matrix, 16 subdomains. 82

3.6 Partitioned Jacobian matrix, four subdomains with overlap. 83

3.7 Overlap of Subdomain 3. 84

3.8 Normal block numbering. 88

3.9 Renumbered blocking. 89

3.10 "Checkerboard" domain decomposition. 91

3.11 Flowchart for Newton-Krylov-Schwarz solution technique. 118

4.1 64×320 domain solution time. 127

4.2 Majority of execution time devoted to Jacobian formation and TFQMR iterations on the C90. 128

4.3 Convergence behavior of the Newton-Krylov-Schwarz algorithm. 130

4.4 Jacobian CPU time, speedup, and efficiency on the C90. 132

4.5 Partitioned Jacobian matrix, four subdomains with overlap. 148

4.6 Plot of overlap behavior versus number of subdomains. 149

4.7 Plot of overlap behavior versus number of subdomains for Cray 96×480 simulation. 153

5.1 Red-black coloring on stripwise, RBGb coloring on "checkerboard" decomposition. 161

A.1 An array stored in Cray banked memory. 229

A.2 Single processor memory access. 234

A.3 Two processor memory access. 234

List of Tables

2.1 Parameter values. 32

2.2 Dimensionless Parameters. 38

3.1 ILU memory requirements (adapted from McHugh [1]). 93

3.2 Schwarz memory requirements (from McHugh [1]). 93

3.3 Iterative behavior of several preconditioners (from McHugh [1]). 94

4.1 Contributions towards total CPU time. 129

4.2 Memory requirements for Cray 64×320 simulation. 130

4.3 Parallel speedup of the linear solution routine on the C90. 132

4.4 Solution algorithm performance data. 133

4.5 Overall code performance. 138

4.6 Overall performance. 139

4.7 New Jacobian performance. 140

4.8 TFQMR routine performance. 140

4.9 Speedup of the additive Schwarz preconditioner formation routine. 141

4.10 32×160 Onyx simulation iteration behavior. 143

4.11 SGI Onyx overall performance. 144

4.12 SGI Onyx Jacobian performance. 145

4.13 SGI Onyx TFQMR performance. 145

4.14 Speedup of the additive Schwarz preconditioner formation routine. 145

4.15 64×320 Onyx Run (4 Blocks). 146

4.16 Memory requirements for SGI 32×160 simulation. 148

4.17 32×160 Onyx simulation iteration behavior comparing overlap values. 149

4.18 Additive Schwarz, 4 domain case, showing effect of overlap on TFQMR iterations and CPU time. 150

4.19 Speedup values for 8 cell overlap problem. 150

4.20 Memory requirements for SGI 32×160 simulation with various overlap values. 151

4.21 Additive Schwarz, 16 domain case, showing effect of overlap on TFQMR iterations and CPU time for a 96×480 simulation. 152

4.22 Speedup values for 12 cell overlap, 96×480 problem using 8 processors. . . . 153

4.23 Speedup values for 12 cell overlap, 96×480 problem using 16 processors. . . 154

4.24 Memory requirements of 12 cell subdomain overlap on 96×480 16 domain problem. 155

5.1 32×160 Onyx simulation iteration behavior comparing additive Schwarz (AS) and multiplicative Schwarz (MS) preconditioning. 162

5.2 32×160 iteration behavior comparing additive Schwarz (AS) and multiplicative Schwarz (MS) preconditioning on the basis of DOP. 162

5.3 Overall code performance for 32×160 stripwise problem on 4 processor Onyx (* 8 block run on 4 processors) 163

5.4 Speedup in the Jacobian routine (* 8 block run on 4 processors). 164

5.5 Speedup of the TFQMR routine (* 8 block run on 4 processors). 164

5.6 Iteration behavior with multiplicative Schwarz preconditioning. 165

6.1 Matrix-free results using "stale" additive Schwarz preconditioning on a 4 subdomain, 32×160 problem on the Onyx. 171

6.2 Parameters for the Cray 32×160 runs. 173

6.3 32×160 matrix-free simulation iteration behavior on the Cray ($1 \times n$ stripwise blocking). 173

6.4 Speedup values for 32×160 problem. 174

6.5 Parameters for the Cray 64×320 runs. 176

6.6 64×320 matrix-free simulation iteration behavior ($n \times 1$ stripwise blocking). 177

6.7 Speedup values for 64×320 problem. 177

6.8 Parameters for the SGI 32×160 runs. 180

6.9 32×160 matrix-free simulation iteration behavior ($n \times 1$ stripwise blocking). 181

6.10 Speedup values for 32×160 problem on the SGI. 181

A.1 LINPACK benchmark results for machines under (or near) \$300K and over 100 Mflops performance (1/1/96). 207

A.2 LINPACK benchmark results for top four machines under (or near) \$300K and over 100 Mflops performance considering other imposed requirements. 209

A.3 NAS parallel benchmark results [2]. 209

A.4 Speedups within a production environment (Table from Cray Research [3]). . 236

List of Symbols

Math Operators:

$\int$	Integration operator
$\oint$	Line integration operator
∇	Del operator
D	Material derivative operator
∂	Partial derivative operator
S	Discrete cell equation set assembly operator

Governing Equation Derivation:

A	Area of a surface
c_v	Specific heat at constant volume
c_p	Specific heat at constant pressure
e	Fluid energy
F	Body force vector
Fr	Froude number ($Fr \equiv \frac{u_o}{\sqrt{gL}}$)
$\mathbf{g}$	Gravity body force vector ($[g_x, g_y]$ components)
g_x	x-direction gravity component
g_y	y-direction gravity component

Gr	Grashof number ($Gr \equiv \frac{g\beta(T_w - T_o)L^3}{u_o^2}$)
k	Thermal conductivity of the fluid
M_i	Flow inlet Mach number ($M_i \equiv \frac{u_o}{\sqrt{\gamma R T_o}}$)
$\hat{\mathbf{n}}$	Unit normal vector ($[n_x, n_y]$ components)
p	Pressure
Pe	Peclet number ($Pe \equiv RePr$)
Pr	Prandtl number ($Pr \equiv \frac{\mu_o c_p}{k_o}$)
$\mathbf{q}$	Heat flux vector
R	Gas constant
Ra	Rayleigh number ($Ra \equiv GrPr$)
Re	Reynolds number ($Re \equiv \frac{\rho_o u_o L}{\mu_o}$)
S	A general surface
T	Fluid temperature
t	Time
$\mathbf{u}$	Fluid velocity vector ($[u, v]$ components)
u	x-direction velocity
V	A general volume
v	y-direction velocity
$\mathbf{x}$	Cartesian coordinate vector ($[x, y]$ components)
x	Horizontal Cartesian coordinate variable
y	Vertical Cartesian coordinate variable

Numerical Methods:

$\mathbf{b}$	Right hand side

D	Diagonal matrix block
d	Newton damping coefficient
$\mathbf{F}$	Discrete governing equations vector
f_i	i-th component of $\mathbf{F}$
$\mathbf{J}$	Jacobian matrix
K_m	Krylov subspace of dimension m
L	Lower diagonal matrix block
N, n	Matrix dimension
$\mathbf{P}_r$	Right preconditioning matrix
$\mathbf{P}_l$	Left preconditioning matrix
U	Upper Diagonal matrix block
$\mathbf{w}$	Krylov vector
$\mathbf{x}$	State variable vector

Greek Symbols:

β	Coefficient of thermal expansion
$\partial\Omega$	Boundary of the domain of computation
$\delta\mathbf{x}$	Newton update vector
ϵ	Tolerance parameter for Krylov iteration
γ	Ratio of specific heats ($\gamma \equiv c_p/c_v$)
λ	Fluid second viscosity
μ	Fluid viscosity
Ω	Domain of computation
ρ	Fluid density

τ	Stress tensor

Subscripts:

0	Starting value for Krylov iteration
i	Column number
j	Block number, row number
m	Dimension of Krylov subspace
R, r	Right preconditioner
L, l	Left preconditioner

Superscripts:

0	Initial preconditioner iteration value
i	Iteration level
n	Newton iteration level

Operators;

$[\,]^T$	Transpose of [ ]

Chapter 1

Introduction

The efficient solution of large two- and three-dimensional Navier-Stokes problems has recently become quite important to many researchers. These simulations arise from the desire to understand the fluid dynamics, convective heat transfer, and mass transfer for ever more realistic situations. Typically, equations similar to those describing the Navier-Stokes problem arise in many fields of science, including:

- atmospheric modeling (weather, pollution tracking, etc.),
- aerospace,
- energy (powerplants, waste, etc.),
- electronics (cooling),
- groundwater modeling (contaminant transport, oil & gas production),
- materials processing (molding, spraying, etc.), and
- transportation (automotive).

For the remainder of this work, the solution of the Navier-Stokes equations (or similar equations) using computational techniques will be defined as computational fluid dynamics (CFD).

To simulate fluid dynamics phenomena computationally, one generally begins with the Navier-Stokes equations [4]

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{1.1}$$

$$\frac{\partial}{\partial t}(\rho \mathbf{u}) + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \tau + \rho \mathbf{F} \tag{1.2}$$

where

ρ = fluid density,

t = time,

$\mathbf{u}$ = fluid velocity vector, $\mathbf{u} = (u, v, w)$, with u, v, and w, corresponding to the x, y, and z direction velocity components,

p = fluid pressure,

τ = viscous stress tensor, and

$\mathbf{F}$ = body force term.

Additional equations are often used to describe the phenomena of energy transfer in the fluid if the specific application warrants it

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot (\rho \mathbf{u} e) = -p \nabla \cdot \mathbf{u} + \tau : \nabla \mathbf{u} - \nabla \cdot \mathbf{q} \tag{1.3}$$

$$e = e(p, \rho) \tag{1.4}$$

where

e = fluid internal energy, and

$\mathbf{q}$ = heat flux vector.

These equations are general in nature, and may be applied to practically any flow problem conceivable. However, it is often possible to simplify the equations to describe a specific class of flow and yield a less computationally challenging equation set.

For a few specific problems, the Navier-Stokes equations can be simplified sufficiently to obtain an exact mathematical solution. Usually, however, these equations cannot be simplified to this extent and still yield useful data about the flow regime of interest. It is desirable to simplify the equations as much as possible to facilitate the construction and execution of the simulation code, while retaining the desired physical accuracy of the results. This is clearly a balancing act.

Historically, CFD has focused on the solution of one- and two-dimensional versions of the Navier-Stokes equations to model the physical phenomena of interest. In most cases, however, the approximation of three-dimensional phenomena by a lesser dimensional space yields less than desirable results. Additionally, the drive for complete and accurate two-dimensional simulations often results in a complex computational model that may not yield acceptable solution times. In summary, the desire to develop better, more accurate models is driving the interest in large scale two- and true three-dimensional simulations. The direct result is, however, lengthy solution times.

1.1 Solution of the Navier-Stokes Equations

Above, an argument was made for the simplification of the general Navier-Stokes equations to describe only the phenomena required for the accurate solution of the specific problem

of interest. The general equations may be significantly simplified using a selection of the following constraints:

- incompressible flow (the material derivative $\frac{D\rho}{Dt} = 0$, infinite sound speed),
- one- or two-dimensional physical domain of interest,
- isentropic flow (viscosity μ and thermal conductivity k equal zero),
- constant properties (fluid density, specific volume, *etc.*), and
- other simplifications (Stokes hypothesis [5], Boussinesq approximation [4], *etc.*).

The application of the selected set of simplifications often results in a system of non-linear partial differential equations (PDEs) called the *governing equations* for the flow regime in the domain of interest. To this PDE set, initial and boundary conditions are added based on the initial flow characteristics and the behavior of the flow at the domain boundaries. The combination of the governing equations and the initial and boundary conditions forms a *well-posed* [6] problem (the problem has a unique solution that varies continously with the given inhomogeneous boundary data).

Though perhaps simplified, the governing equation set (with conditions) is usually too complex to be solved exactly by analytical means. This set may also result in equations too complex for solution using other techniques such as symbolic manipulators, high-level mathematics languages, *etc.* Typically, the governing equation set is approximated using discretization techniques, which ultimately reduce the PDE system to a (generally) non-linear system of algebraic equations. It is these non-linear algebraic equations that require a numerical solution.

These discrete approximations are usually based on the subdivision of the problem domain into a computation domain consisting of a mesh of computational cells (elements, volumes) that faithfully discretize the problem domain. In each of these cells, the governing equations are approximated by a set of non-linear algebraic equations. The method used to obtain this discretization and set of algebraic equations over the computational domain depends on the specifics of the problem and other factors. Several methods are commonly used,

- the finite difference method,
- the finite volume method,
- the finite element method, and
- spectral methods.

All these methods have advantages and disadvantages; some are better suited than others for given classes of problems or problem geometries.

Further complicating the choice of discretization method (that may impact the details of the method chosen) is the choice of the method used to solve the algebraic system resulting from the discretization process and initial/boundary conditions. The algebraic system may be solved *explicitly* (the unknown term in each algebraic equation is evaluated based on the known values of the other variables), or *implicitly* (the algebraic system requires simultaneous solution of the equations involving the unknowns), or some combination of the two.

Given an appropriate choice of governing equations, initial and boundary conditions, discretization technique, and algebraic solution method, a computer program (code) may be constructed to calculate the approximate solution to the desired problem. The accuracy of this solution is clearly dependent on the approximations performed. In general terms, accuracy may be improved by one (or a combination) of the following.

- Retaining more terms in the governing equations. Often, terms are removed in the simplification process that affect the details of the solution in favor of reasonable execution time.

- Adding problem dimensions. One- and two-dimensional approximations may not yield acceptable results if the phenomena of interest is three-dimensional in nature.

- Use of a more accurate discretization method. Low-order methods often yield quicker, but less accurate, solutions.

- Use of a finer discretizing mesh. Problems that have steep gradients often benefit with the use of a finer mesh. Direct simulation of fluid turbulence requires a fine mesh to resolve the small wavelength phenomena inherent in such a calculation.

However, all of these approaches lead to an increase in the time required to obtain a solution, occasionally drastically so. Again, the accuracy of the solution must be delicately balanced with the time required to obtain a solution to the problem.

The solution of complex two- and three-dimensional CFD problems are often called "Grand Challenges," with good reason. To best illustrate the magnitude of achieving a solution of a typical problem of this type, consider a two-dimensional solution to obtain a velocity field, with simple governing equations consisting of the mass conservation equation and u and v momentum equations (3 total). Consider use of the popular finite volume technique on a computation domain consisting of 200 discrete volumes per side (a large problem by current solution standards, realistically still too small for any detailed resolution of physical phenomena in complex domains and for turbulence simulation). Furthermore, consider a simultaneous solution of the algebraic equations using an exact direct technique. A domain with 200 volumes per side or $200^2 = 40,000$ total volumes in the discretization. For each

of these volumes, three algebraic equations are used to describe the flow within the volume (one algebraic equation for each of the governing equations to be approximated), resulting in 120,000 total equations to be solved. Using an implicit technique, these non-linear algebraic equations may be linearized and placed in a matrix form, $\mathbf{Ax} = \mathbf{b}$, where $\mathbf{A}$ is a matrix containing the equation coefficients, $\mathbf{x}$ is the vector of unknowns, and $\mathbf{b}$ is the right-hand-side vector containing constants. In this configuration, the number of equations (120,000) maps directly to the number of rows in the coefficient matrix. To obtain the solution of the matrix equation requires that the coefficient matrix $\mathbf{A}$ be inverted. Several techniques exist to perform the inversion. Current practices indicate that $O(n^3)$ operations are required to obtain an exact inverse for a dense matrix $\mathbf{A}$ (counting both addition and multiplication operations, the Gaussian elimination used in LINPACK [7] requires $2n^3/3 + 2n^2$ operations). Realistically, an efficient iterative technique on a sparse matrix should easily be more efficient than $O(n^3)$, but certainly cannot require less than $O(n^2)$ operations (recall, $O(n^2)$ operations are required to store the result alone). This example will assume that the inversion can be accomplished in $O(n^3)$ operations. Thus, a total of 1.73×10^{15} floating point operations will be required for one matrix inversion

$(120,000)^3 = 1.73 \times 10^{15}$ operations (flops, assuming matrix solution $\approx O(n^3)$ operations).

To perform this number of floating point calculations on a personal computer class machine would require 2.9 years (1.73×10^{15}/19 Mflops (Pentium 133MHz [7]) = 2.9 years) of continuous computation (24 hours per day, 365 days per year). For more efficient processors, the execution time decreases:

- an HP 735/99 workstation (120 Mflops) requires $\approx$ 167 days for the calculation, and

- a Cray T90 (one processor, vector code, 1.6 Gflops) needs $\approx$ 13 days to perform the work.

These times are for *one* matrix inversion; transient and iterative solutions require many such inversions over the solution of the problem. The use of full Gaussian elimination is a bit pessimistic and unnecessarily inflates the results for this example. However, this example may not always represent an extreme case. In fact, the governing equations used were overly simplistic. The problem was two-dimensional rather than the often more desirable three-dimensional domain, and a direct steady-state result was assumed. Realistically, a useful simulation would require more operations per inversion, with a corresponding increase in solution time. It is evident that the solution of 3D problems of this size may not be feasible due to the extremely long computation times required.

Given the necessity of the solution of larger and more complex two- and three-dimensional problems, CFD researchers and developers have only a few strategies to reduce the time required for a simulation.

- Wait until hardware advances yield acceptable solution times.
- Improve the $O(n^3)$ matrix solution efficiency. For the usual sparse matrices encountered, it is often possible to find algorithms with better efficiencies. However, solution efficiency cannot be improved beyond $O(n^2)$, the number of operations required to store the inverse in memory. Furthermore, due to the maturity of many simulation applications, it may be reasonable to assume that the optimal solution algorithm is already implemented. As such, improvements here may not be possible.
- Use parallel algorithms that allow concurrent matrix inversion. Ideally, these algorithms would scale to a large number of processors to allow the timely solution of an arbitrarily

complex simulation.

1.2 Research Overview

Given the above options, this research will concentrate on the third strategy: concurrent matrix inversion, or more generally, solution of the linear system. As an ideal goal, this research would result in a reformulation of the relationship between simulation complexity and solution time; as accuracy increases, the time required for solution also increases, often drastically. Practically, the aim is to exploit parallel computation for the implicit, fully-coupled solution of CFD problems. Specifically, parallelism will be employed in the concomitant non-linear algebra problem such that the number of processors used for the solution will increase with increasing problem (simulation) complexity to yield a constant amount of "work" per processor and an overall code execution time that remains roughly constant. For the purposes of this study, performance of this nature will be considered "ideally scalable." However, in practice, if an overall speedup approaches the number of processors used to achieve that speedup (see Appendix A), the performance will be termed "scalable." Given that an inversion technique is scalable, as the accuracy of a solution is increased, more processors are required for a given solution time. In effect, the limiting condition would be the number of processors available for use by the simulation, not the simulation time required for a solution.

The existence of a true scalable linear solution technique for general sparse matrices is unlikely. Realistically, it may be sufficient to develop methods that scale "well" up to n processors, where n is determined by available hardware. Currently, the majority of CFD analysis is performed on low-end hardware for economic reasons. Algorithm scalability to a large number of processors on high-end hardware will not generally benefit the intended

audience. Therefore, the main emphasis will be on hardware readily available to CFD analysts; shared-memory multiprocessors such as conventional vector/parallel supercomputers and SMP (Symmetric MultiProcessor) workstation class machines. For this class of hardware, scalability to $n = 32$ is presently a realistic upper bound. In fact, as of this writing, only Silicon Graphics sells a machine with $n > 16$ processors (for a related discussion, see Appendix A).

Within these limits, the primary goal of this research is the implementation of an $n \leq 16$ scalable parallel linear system solution technique for use with a Newton-Krylov non-linear solution method [8, 9, 10, 11, 12] applied to a compressible flow model using a two-dimensional backward-facing step problem as a means of testing the results. Other aspects of this problem (mesh refinement, physical properties, *etc*) are not studied as a part of this research.

1.2.1 History and Related Work

The motivation for this work may be traced to research in the two-dimensional solution of the Tokamak edge (*i.e.*, boundary layer) plasma fluid equations [13]. The Tokamak is a promising design for a nuclear fusion reactor that is based on magnetic confinement of the fusion plasma. Superconducting magnets are used to create a magnetic field of sufficient strength to confine the fusion reaction in the core of the Tokamak. The edge plasma is the boundary layer between the magnetic confinement field and the walls of the vessel. An accurate model of the edge plasma is critical for the proper design of a Tokamak.

Solution of the edge plasma flow is computationally very challenging. In addition to the fluid equations described previously, equations accounting for the atomic reactions are necessary. The result is a much larger, more complicated equation set to be solved. These equations have transport coefficients that are strong non-linear functions of the dependent

variables, and incorporate difficult boundary conditions. The Newton solver implemented by Knoll [13] proved to be a robust and efficient non-linear solution technique. Newton's method generally results in better coupling between the equations of the simulation than other common methods [14, 15], which manifests itself in faster and more robust solutions.

Given this initial success with Newton's method on complex simulations, research progressed towards the use of the method to solve more complete edge plasma flows, combustion problems, and difficult flow domains. Knoll [8] developed two benchmark problems to further refine the Newton technique, adding a numerical Jacobian evaluation, convergence enhancement features, and increasing the algorithm's efficiency. McHugh [9] extended this work using an inexact Newton's method, differing governing equations, and conjugate-gradient-"like" algorithms for the linear solution process. McHugh [10] further refined his earlier work on the linear solution algorithms, and implemented a matrix-free solution technique. Jacobs [12] studied the use of domain decomposition to develop a preconditioner for the transpose-free quasi-minimal residual (TFQMR) conjugate-gradient-"like" algorithm, and investigated the additive and multiplicative Schwarz methods. Johnson [16] examined the use of higher-order methods to improve solution accuracy. McHugh [17] developed a simulation of the ASME benchmark problems using many of the above methods. Finally, Knoll [18] has summarized much of this work. Progress on these techniques and their application continues unabated; from the recent literature these techniques are a fertile research area with interest within both academia and the various research laboratories.

Related Work

Idelsohn [19] presents an excellent comparison between finite volume and finite element methods for assorted problems. He stresses the advantages of each technique for the solution of

Navier-Stokes problems, and concludes that a hybrid of the two techniques (the finite element method is superior for the solution of symmetric terms, the finite volume technique is better suited to solve the non-self-adjoint terms) may provide more efficient solutions.

This work also capitalizes on the wide body of published research performed on Newton's method, Krylov techniques, and Schwarz methods. McHugh [20] provides an excellent description of several Krylov solution techniques. Golub [21] provides a history and overview of various Krylov techniques, and the Lanczos method. Axelsson [22] shows the development of the conjugate-gradient method for symmetric positive-definite matrices and extends this result via the Lanczos biorthogonalization procedure to obtain the biconjugate-gradient method (BCG) for the solution of non-symmetric indefinite problems. Freund [23] presents the TFQMR method. Saad [24] developed the restarted generalized minimal residual (GMRES) algorithm. Several other Krylov techniques have been developed (but not used in this study); the Bi-CGSTAB algorithm [25], CGS [26], and hybrid techniques [27], to name a few. Chronopoulos [28] discusses biorthogonal Lanczos methods and compares a restarted squared Lanczos method to restarted GMRES.

McHugh [1] provides a comparison and contrast of Schwarz, ILU, and some matrix-splitting preconditioning methods, focusing on issues of serial performance, memory requirements, and robustness of algorithms on compressible 2D flow on a backward-facing step problem. Cai provides an overview of Schwarz preconditioning [29] and a discussion of additive and multiplicative Schwarz algorithms (including coloring schemes) [30]. Pavarino [31] presents an additive Schwarz preconditioned method applied to petroleum reservoir simulation on an elliptic problem. Bramble [32] discusses additive and multiplicative preconditioning on elliptic problems.

Finally, several examples of directly related work appear in the literature. This study fo-

cuses on extending work by McHugh, Knoll, Jacobs, Mousseau, and Johnson [1, 8, 9, 10, 12, 16, 17, 18, 20, 33, 34, 35] to parallel architectures, based on preliminary work performed by this author [1, 36]. In [1], McHugh provides a comparison between ILU and Schwarz preconditioning on a low Mach number backward-facing step problem. He concludes that Schwarz preconditioning is generally more robust than ILU for small Mach number simulations. Work by Venkatakrishnan [37, 38] suggests that ILU preconditioning tends to perform better than other matrix-splitting techniques. These results indicate that Schwarz preconditioning may possess certain advantages for poorly conditioned problems such as the one considered in this study (see Chapter 3).

Bjørstad [39] employs decoupling of the PDE system (using pressure and saturation equations) in conjunction with additive Schwarz preconditioned Krylov techniques for the parallel solution of petroleum reservoir equations. Bjørstad solves the pressure equation using a preconditioned conjugate-gradient algorithm (the pressure equation results in a symmetric positive-definite linear system), and employs a preconditioned Bi-CGSTAB algorithm to solve the non-symmetric saturation equation. He also adds that "ASM preconditioning is very robust with respect to large variations in permeability," (ASM is an acronym for the additive Schwarz method). Venkatakrishnan [40] implements a modified-Newton-Krylov-ILU parallel solution of an aerodynamics problem at Mach 0.2. He compares the implicit modified-Newton-Krylov scheme with an explicit technique (usually trivially parallelizeable) and finds that the iterative technique provides superior performance due to superior convergence behavior. Venkatakrishnan simplifies certain Jacobian terms (hence the nomen modified-Newton) and thus is not guaranteed the full Newton quadratic convergence behavior.

Cai [41] provides results of Newton-Krylov-Schwarz and matrix-free solutions to an inviscid compressible flow problem. Shadid [42] examines a parallel implementation of several

Krylov solution techniques on an nCUBE 2 architecture. However, Shadid does not investigate Schwarz preconditioning or matrix-free methods, the set of model problems he studies are not closely related to this work. His efficiency results show that parallelization of the Krylov technique is feasible on certain architectures and would be an obvious extension to this study. Ajmani [43] solves a compressible two-dimensional backward-facing step problem using preconditioned Krylov techniques. He does not examine Schwarz preconditioning or matrix-free techniques and employs larger Mach and Reynolds numbers for the solution. Choquet [44] examines a parallel Newton-Krylov finite element solution to compressible flow. Choquet concentrates on aerodynamics problems and uses diagonal or ILU preconditioning. Of particular interest is Choquet's comparison of the convergence of the Newton linearization step using both exact and numerical Jacobian derivations. For his example problem, use of an exact implementation decreases the number of iterations required for convergence by approximately 14%. This small improvement may not justify the use of exact implementations, especially considering the complexity of these expressions.

In summary, the work reported herein provides several contributions to the field of the parallel implicit solution of Navier-Stokes problems (specifically low Mach number compressible flow on a backward-facing step domain) using Newton-Krylov-Schwarz solution techniques. The overall goal of this study is an efficient mapping of the Newton-Krylov-Schwarz solution procedure to selected parallel architectures while preserving the robustness, convergence behavior, and generality of the technique that was demonstrated on a single processor [1]. Specific goals are listed below.

1. Provide robust, parallel solutions to steady-state viscous compressible flow on a backward-facing step at a Reynolds number of 100 and inlet Mach number of 0.0025. This problem

is numerically extremely challenging due to the large off-diagonal terms in the Jacobian and is an excellent test of the robustness of the solution algorithm. Serial solution of this problem has been achieved by McHugh [1]. This study is the first to tackle such a problem focusing on parallel solution efficiency.

2. Investigate the mapping of implementations of this Newton-Krylov-Schwarz solution algorithm on the Cray C90 and SGI Onyx. While there have been several investigations of subsets and variations of these techniques, this is the first to analyze the parallel behavior of the full inexact Newton algorithm with a global Krylov linear solution using Schwarz preconditioning methods on a mapping problem of this type.

3. Examine a parallel matrix-free implementation of the above methods, using pseudo-transient relaxation in conjunction with a lagged Jacobian and preconditioner formation strategy to reduce the influence of the preconditioner formation on the algorithm execution time.

4. Finally, suggest the "ideal" hardware configuration for parallel Newton-Krylov-Schwarz Navier-Stokes solutions for problems similar to the model problem.

1.2.2 Mathematical Overview

The efficiency of various methods used for the linear solution process and the suitability of a given method for a particular problem is somewhat dependent on the governing equations, boundary and initial conditions, and discretization technique employed in the solution. The linear solution method may also be strongly affected by the details of the specific application. It is tempting to select a simple problem with simple equations to allow concentration on the parallel aspects of the solution process. Unfortunately, this approach is of little interest as the

goal is to examine the solution of realistic problems. Additionally, results from a simplistic model problem may not map to complex problems of more general interest. Therefore, to provide real benefits to current topics in CFD research the governing equations, initial and boundary conditions, discretization techniques, and application specifics from current CFD research areas are selected as a basis for this study.

Industry has also recognized the difficulty of mapping basic research using a simplistic basis into realistic application domains. In response, the American Society of Mechanical Engineers (ASME) Heat Transfer Division has defined a set of "more realistic and difficult" benchmark problems that may be used as a basis for research and that allow the verification of software accuracy and validity [45]. One of these benchmarks is a model of fluid flow and heat transfer in a backward-facing step geometry. For the present study, the solution of the benchmark backward-facing step problem using a finite volume discretization and fully implicit solution techniques [17] is selected as the physical basis for examination of the algebraic equation solver.

The benchmark problem of steady, two-dimensional, compressible fluid flow over a backward-facing step may be simulated with the use of the following equations for $(x, y) \in \Omega$, the physical domain of the problem (Figure 1.1)

Continuity:

$$\frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0 \tag{1.5}$$

Momentum:

$$\begin{aligned} \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} &= -\frac{\partial p}{\partial x} + \frac{1}{R_e}\left\{\frac{\partial}{\partial y}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right]\right. \\ &\left. + \frac{\partial}{\partial x}\left[2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right]\right\} + \frac{\rho}{{Fr_x}^2} \end{aligned} \tag{1.6}$$

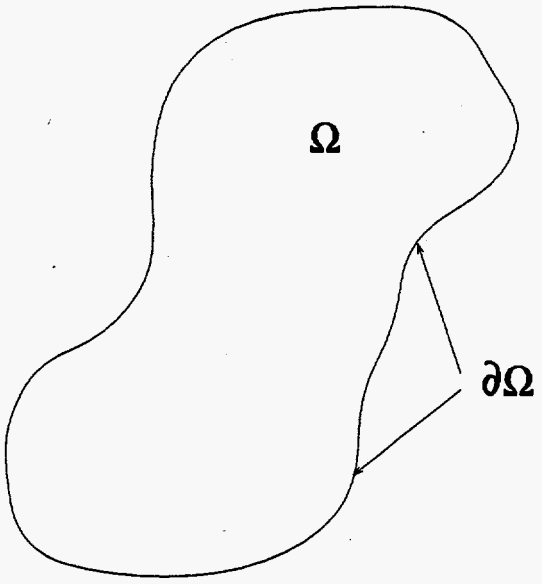

Figure 1.1: The problem domain.

$$\begin{aligned}\frac{\partial \rho uv}{\partial x}+\frac{\partial \rho v^2}{\partial y} &= -\frac{\partial p}{\partial y}+\frac{1}{R_e}\left\{\frac{\partial}{\partial x}\left[\mu\left(\frac{\partial u}{\partial y}+\frac{\partial v}{\partial x}\right)\right]\right. \\ &\left.+\frac{\partial}{\partial y}\left[2\mu\frac{\partial v}{\partial y}-\frac{2}{3}\mu\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)\right]\right\}+\frac{\rho}{{Fr_y}^2} \qquad (1.7)\end{aligned}$$

Energy:

$$\begin{aligned}\frac{\partial \rho uT}{\partial x}+\frac{\partial \rho vT}{\partial y} &= \frac{\gamma}{P_e}\left[\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}\right] \\ &-\gamma(\gamma-1)M_i^2 p\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right) \qquad (1.8)\end{aligned}$$

State:

$$p=\frac{\rho T}{\gamma M_i^2}, \qquad (1.9)$$

and subject to conditions on ρ, u, v, T, p on the boundary $\partial\Omega$, with the nomenclature

u - x velocity component,

v - y velocity component,

p - pressure,

T - temperature,

ρ - fluid density,

R_e - flow Reynolds number,

P_e - flow Peclet number,

γ - ratio of specific heat capacities,

M_i - inlet flow Mach number, and

Fr - Froude number.

The Newton-Krylov method is a fully implicit scheme used for the solution of the non-linear algebraic equations that result from the discretization of the governing partial differential equations. The finite volume discretization of the governing equations results in a non-linear system,

$$\{g_1(\mathbf{x}), g_2(\mathbf{x}), g_3(\mathbf{x}), g_4(\mathbf{x}), g_5(\mathbf{x})\}, \tag{1.10}$$

for each discrete volume cell (Ω_e) composing the physical domain. The state variable, $\mathbf{x}$, can be expressed as

$$\mathbf{x} = [\mathbf{x}_1, \mathbf{x}_2, \mathbf{x}_3, \mathbf{x}_4, \mathbf{x}_5]^T = (\rho, u, v, p, T). \tag{1.11}$$

The system of equations (1.10) in each cell may be re-written in the following form:

$$\{f_1(\mathbf{x}) = 0, f_2(\mathbf{x}) = 0, f_3(\mathbf{x}) = 0, f_4(\mathbf{x}) = 0, f_5(\mathbf{x}) = 0\}. \tag{1.12}$$

The contributions of each of the above equations in each cell may then be assembled together

to form the global non-linear system over the computational domain

$$\mathbf{F}(\mathbf{x}) = 0. \tag{1.13}$$

Application of Newton's method requires the iterative solution of the linear system,

$$\mathbf{J}^n \, \delta\mathbf{x}^n = -\mathbf{F}(\mathbf{x}^n), \tag{1.14}$$

where n is the iteration number and the elements of the Jacobian are defined by

$$\mathbf{J}^n_{\text{row, column}} = \frac{\partial f_{\text{row}}}{\partial x^n_{\text{column}}}, \tag{1.15}$$

and the new solution approximation is obtained from

$$\mathbf{x}^{n+1} = \mathbf{x}^n + d \, \delta\mathbf{x}^n. \tag{1.16}$$

The constant, $d \in (0,1]$, in Equation 1.16 is used to damp the Newton updates. The damping strategy is designed to prevent the calculation of non-physical variable values (i.e., negative temperature), and to scale large variable updates when the solution is far from the true solution. This iteration is continued until the Euclidean norm of either $\delta\mathbf{x}$ or $\mathbf{F}(\mathbf{x})$ is below some suitable tolerance level.

Newton's method is attractive because it converges quite rapidly when given an appropriate initial estimate. In fact, Newton's method is the standard used to compare rapidly convergent methods for solving the non-linear system (Equation 1.13) [46].

The application of Newton's method results in a linear algebraic system (Equation 1.14) to

be solved each Newton iteration. This system may be solved directly (with banded Gaussian elimination, for example). Direct methods are generally difficult to parallelize and require an excessive amount of computation especially as an exact solution to the linear system is not needed until the final Newton iteration. The use of an iterative Krylov technique (*i.e.*, a conjugate-gradient-"like" method), such as the transpose-free quasi-minimal residual technique (TFQMR) [46], to solve Equation 1.14 gives rise to an "inexact" Newton's method. Inexact methods can be used to prevent excessive computation by the linear solution algorithm when the non-linear iteration is far from convergence. In this technique, the tolerance level for the iterative linear solution process is tightened as the Newton residual decreases according to

$$\frac{\|\mathbf{J}^n \delta\mathbf{x}^n + \mathbf{F}(\mathbf{x}^n)\|}{\|\mathbf{F}(\mathbf{x}^n)\|} < \epsilon, \tag{1.17}$$

where ϵ is a user-specified tolerance parameter.

For realistic, complex problems, solution of the linear system may be quite difficult (or impossible). Often this system is poorly conditioned (*i.e.*, the Jacobian matrix contains a wide disparity in eigenvalues). Preconditioning is used to improve the condition number of the system to facilitate solution. A preconditioner ($\mathbf{P}_l$) may be applied to the left[1] of the Jacobian, resulting in the expression

$$\mathbf{P}_l^{-1}\mathbf{J}^n \, \delta\mathbf{x}^n = -\mathbf{P}_l^{-1}\mathbf{F}(\mathbf{x}^n). \tag{1.18}$$

Effective preconditioning requires that the preconditioner be a reasonable approximation of $\mathbf{J}$ and that systems of the form $\mathbf{P}_l\mathbf{v} = \mathbf{b}$, which arise within the TFQMR iteration, can be solved efficiently. One popular class of preconditioners is based upon incomplete factorizations

[1]Right preconditioning is also a valid approach (see Section 3.3) and results in a slightly different expression.

(ILU) of the Jacobian matrix [47]. However, ILU preconditioners often do not scale well with problem size [48] and exhibit data dependencies that hinder parallel implementations.

The formation and inversion of the preconditioner may quickly dominate the Newton-Krylov solution time on larger problems. This task potentially requires $O(n^3)$ operations (if LINPACK Gaussian elimination is used to invert the preconditioner, a total of $2n^3/3 + 2n^2$ operations are needed for the inversion task [7]). Two possibilities exist that may improve the efficiency of this task.

- The formation and inversion of the preconditioner can be performed concurrently on multiple processors. Beyond the concept of a linear execution time decrease as a function of the number of processors employed, this technique typically creates a preconditioner by assembly of several preconditioner subblocks obtained in parallel (*e.g.,* domain-based preconditioning). Construction of the preconditioner using multiple smaller subblocks theoretically provides much greater efficiency than a simple work division among several processors would indicate (using four subblocks decreases the number of operations to $4 \times (2[n/4]^3/3 + 2[n/4]^2) \approx 1/16 \times (2n^3/3)$, a factor of nearly 64 for large n).

- It is likely possible to improve the overall $O(n^3)$ inversion time for a sparse linear system significantly, as discussed previously. However, for the remainder of this study it will be assumed that the optimal inversion algorithm is insufficient to provide the desired performance increase.

The additive and multiplicative Schwarz algorithms [30] are examples of domain-based preconditioners that lend themselves to parallel implementations.

1.2.3 Research Outline

The thesis begins by providing the theoretical development of the governing equations in two dimensions, followed by a finite volume approximation of the equations as background material. Given the approximate equations, Newton's Method is then applied to linearize the approximations, followed by the application of Krylov-subspace-based algorithms to solve the resulting linear system.

A preconditioner is usually employed to accelerate the convergence of the Krylov method. Indeed, for stiff, poorly-conditioned matrices, preconditioning is often required to obtain a solution. The quality of the preconditioner strongly influences the convergence behavior of the Krylov technique. The optimal preconditioner is one that balances the minimization of the number of Krylov iterations with the number of operations required to develop the preconditioner, resulting in a preconditioner that minimizes the time spent in the linear solution procedure. Based on operation count (approximately $O(n^3)$), the formation of the preconditioner is possibly the dominant task in the solution procedure for large two-dimensional problems. As the remainder of the process requires $o(n^3)$ operations, large problem sizes increase the importance of obtaining a quality preconditioner.

Given this background information, the research presented here concentrates on obtaining a quality preconditioner based on parallel algorithms. Ideally, the preconditioner will not only minimize the number of Krylov iterations required per Newton step, but will also be amenable to parallel construction that is scalable beyond a small number of subblocks (or processors, assuming each processor is dedicated to obtaining its respective subblock of the preconditioner). This research examines the additive Schwarz method in detail and considers the feasibility of algorithm modifications that exhibit scalability without degradation of the

global preconditioner. Additionally, the multiplicative Schwarz algorithm (while not parallel in the base form) may be modified for parallel execution, forming a second candidate for study. Depending on the outcome of these efforts, further study of other algorithms (such as a matrix-free approach) may be appropriate in the search for a scalable preconditioner.

Previous research [12] examines the additive and multiplicative Schwarz methods used towards the development of a linear solve preconditioner using domain decomposition techniques. Of the algorithms studied, only additive Schwarz with no domain overlap is parallel without subblock dependencies in base form. At some reduced level of efficiency, additive Schwarz methods with overlap can be parallelized. Multiplicative Schwarz techniques require a renumbering (coloring) operation to resolve data dependencies. Jacobs [12] compares and contrasts serial implementations of additive and multiplicative Schwarz preconditioners on a 24×96 cell backward-facing step problem.

With this background research, it is clear that parallel implementations of the additive and multiplicative Schwarz method should be examined. To study the feasibility of these approaches, the author performed preliminary work on a parallel implementation of the additive Schwarz algorithm on a Cray C90 [36]. The additive Schwarz algorithm was successfully implemented in parallel and demonstrated scalability to four processors. The eight-processor case, however, was disappointing. This loss of scalability was largely due to an increase in the number of Krylov iterations required to solve the eight-block case. This behavior indicates that for the additive Schwarz algorithm with no subdomain overlap, the quality of the resultant preconditioner decreases rapidly as the number of subblocks (processors) increases.

To achieve the primary goal of this research, further work should be invested in the study of subdomain overlap and how it affects the behavior of the additive Schwarz method. As of this date, overlap is not expected to result in an acceptable scalable preconditioner (even

small values of subdomain overlap may result in extreme memory requirements). Beyond the study of overlap with the additive Schwarz method, this research will examine the parallel implementation of a renumbered (colored) multiplicative Schwarz method. Study of these algorithms will proceed as outlined below.

1. Implementation of a parallel, shared-memory version of the algorithm.
2. Testing and optimization of the algorithm.
3. Benchmarking of the algorithm.

1.2.4 Alternatives

Many alternatives to the path outlined for this research exist. The selection of a physical problem defines the governing equations to use in the development of the simulation. Along the path chosen for this study (the governing equations, finite volume discretization, and Newton-Krylov-Schwarz techniques), several decision points were encountered where alternative decisions were possible:

- selection of the discretization technique (finite difference, finite volume, or finite element),
- selection of the method to linearize the non-linear algebraic system resulting from the discretization process,
- solution of the linear system resulting from the previous step, and
- the use of preconditioning, if necessary.

Recall that the motivation for the selection of the discretization method and the use of Newton's method for a linearization technique serves to narrow the field of alternatives. The

previous development assumes the use of a Jacobian matrix as part of the Newton-Krylov solution scheme. An alternative, the matrix-free implementation, shows much promise in comparison to conventional techniques. With most Krylov projection methods, the Jacobian matrix appears as matrix-vector products of the form $\mathbf{Jw}$, where $\mathbf{w}$ is an intermediate vector used internally in the Krylov technique. For an inexact Newton's method, the actual Jacobian matrix need not be calculated, because only the product,

$$\mathbf{Jw} \approx \frac{\mathbf{F}(\mathbf{x}+\epsilon\mathbf{w}) - \mathbf{F}(\mathbf{x})}{\epsilon}, \tag{1.19}$$

is needed for the solution. For the matrix-free technique, only the matrix-vector products $\mathbf{Jw}$ are stored and manipulated; the Jacobian $\mathbf{J}$ is never actually calculated.

Alternatives also exist in the selection of algorithms for the development of the preconditioner. Additive and multiplicative Schwarz allow the development of a preconditioner with the use of subblocks. Given appropriate modifications to the algorithms, the construction of a global preconditioner based on the assembly of these subblocks can be performed in parallel.

It is clear that many alternatives to the research outlined in this opening exist and some of the alternatives have the potential to yield useful results. Based on the knowledge and background information to date, the finite volume discretized Newton-Krylov-Schwarz solution of the 2D compressible Navier-Stokes equations is competitive with, and in many aspects superior to, other usable techniques to solve this problem. Given the robustness and suitability of Newton-Krylov techniques for the solution of this problem and the computational complexity of the preconditioner formation in comparison with the remainder of the solution algorithm, concentration of this research on the preconditioner formation algorithm is clearly

justified.

1.3 Summary of Procedures and Results

In this introduction, motivation was provided for the efficient solution of compressible Navier-Stokes-based problems on two- and three-dimensional domains, using approximate computational methods. It was argued that as discretization refinement, problem dimension, and the complexity of the governing equations increases, the construction of a quality Newton-Krylov preconditioner may quickly overwhelm the other routines in the simulation in terms of operation count and, proportionately, execution time. With this being the case, solution of these types of problems (at least in a meaningful time frame) will require an algorithm that constructs a quality preconditioner in a minimum of time. Current research suggests that the most promising approach to this end would involve a scalable preconditioner construction method based on subblocks.

To obtain a scalable preconditioner, further research on the additive Schwarz method targeted towards maintaining preconditioner quality as the number of preconditioner subblocks is increased is clearly warranted. Additionally, a parallel implementation of the multiplicative Schwarz method should be examined. Most likely, other algorithms and variations should be reviewed for applicability towards providing an efficient and robust parallel preconditioner.

To summarize this work, addressing the above arguments, two parallel implementations of Newton-Krylov-Schwarz algorithms were used to solve the low Mach number compressible fluid flow model problem. The first technique involved a direct Newton-Krylov-Schwarz solution, with the second using a pseudo-transient relaxed matrix-free implementation of the technique. In both cases, Newton's method was used to linearize the discrete system, and

a preconditioned Krylov projection technique was used to solve the resulting linear system. Domain decomposition enabled the development of a global preconditioner via the parallel construction of contributions derived from subdomains (by assigning each subdomain contribution to an independent processor). Formation of the global preconditioner was based upon additive and multiplicative Schwarz algorithms, with and without subdomain overlap. For the second case, the degree of parallelism of the technique was further enhanced with the use of a matrix-free approximation for the Jacobian used in the Krylov technique. Furthermore, with the use of the pseudo-transient algorithm, additional robustness was added by enhancing the sphere of convergence within the Newton algorithm. Relaxation also allowed a reduction in the influence of the preconditioner on the overall solution time by "lagging" the formation of the preconditioner, amortizing this penalty over several Newton iterations.

The multiplicative Schwarz preconditioner did not yield performance advantages over the additive Schwarz version, because the superior convergence behavior was effectively offset by the required coloring technique used to address data dependencies within the method. The Schwarz preconditioner quickly degraded in quality as the number of subdomains were increased, resulting in a sharp increase in the number of Krylov iterations required for the simulation. This behavior severly limited the degree of parallelism that could be employed within the preconditioner. Subdomain overlap was somewhat successful in reducing this effect and resulted in some degree of scalability to 16 C90 processors. However, overlap values that provided the best performance resulted in extreme memory requirements. Additionally, even with overlap, preconditioner degradation with an increase in the number of subdomains, along with remaining serial code within the Krylov solve, continued to limit scalability.

A solution technique based on a parallel Jacobian formation algorithm, additive Schwarz preconditioning without overlap, and a matrix-free implementation did provide excellent per-

formance on 8 C90 processors and 4 SGI processors when pseudo-transient continuation was employed as a check on the increase of linear iterations with the number of subdomains. Furthermore, this study concluded that further work on the algorithms and hardware mapping concerns along with additional hardware features would likely result in scalability to larger machines.

To place this result in perspective, the model problem studied requires a very powerful preconditioning technique due to the low Mach number inlet condition. If this were not the case, better scalability results could certainly have been achieved. As such, the model problem selected demonstrates the "worst-case" scalability that would be obtained with these techniques. It is evident that the results and conclusions of this study are specific to the model problem. However, these results may be applicable to a much wider variety of situations if the results are viewed as a lower-bound to the performance that may be achieved on a "general" simulation.

Chapter 2

The Mathematical Basis

This chapter outlines the theoretical development of the governing equations for two-dimensional, compressible fluid flow including energy effects from the Navier-Stokes and energy equations. Given the governing equation set, a finite volume discretized form is developed to obtain the non-linear set of algebraic equations that represent the flow on the computational domain. To complete the basis, the Newton-Krylov technique is developed and applied to the non-linear system to obtain a linear algebraic system for solution.

2.1 The Backward-Facing Step Problem

Prior to the derivation of the governing system and solution technique, it is necessary to define the model problem to be examined. As discussed in the introduction of this study, the two-dimensional steady-state solution of low Mach number compressible flow over a backward-facing step provides a non-trivial test case for the Newton-Krylov-Schwarz method.

Consider the physical layout of a two-dimensional backward-facing step domain (see Figure 2.1). This figure illustrates the domain, where the reference y axis is horizontal parallel

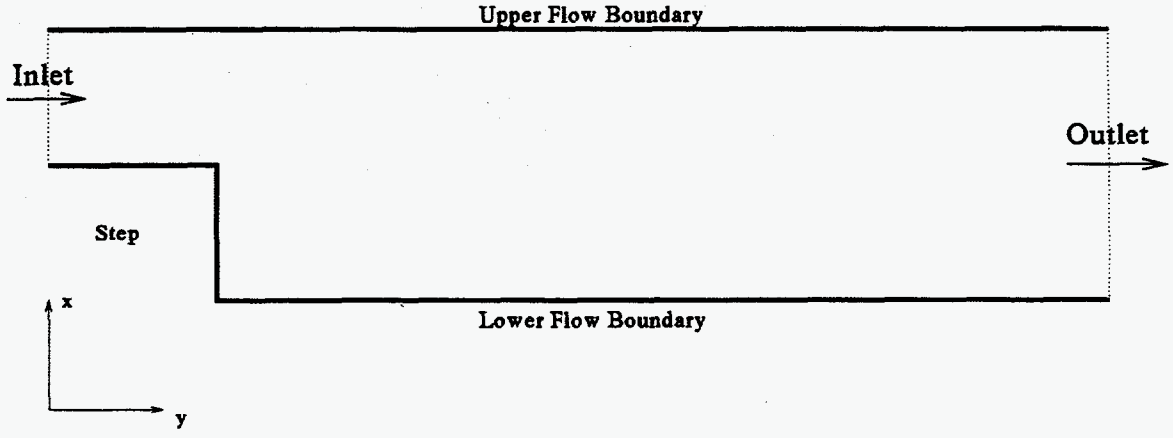

Figure 2.1: The backward-facing step.

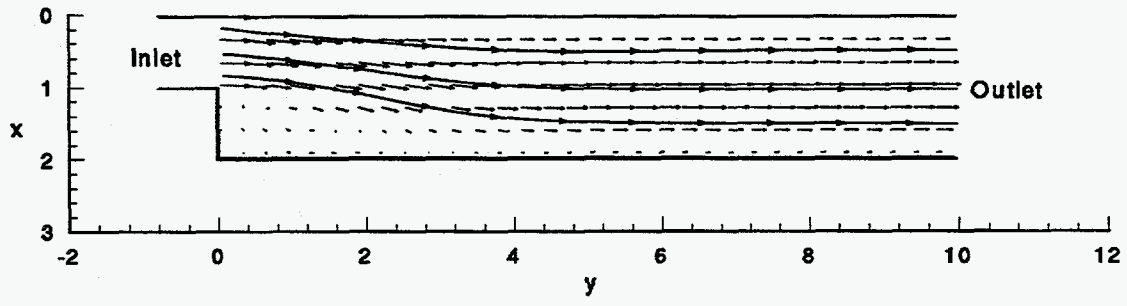

Figure 2.2: Flow velocity.

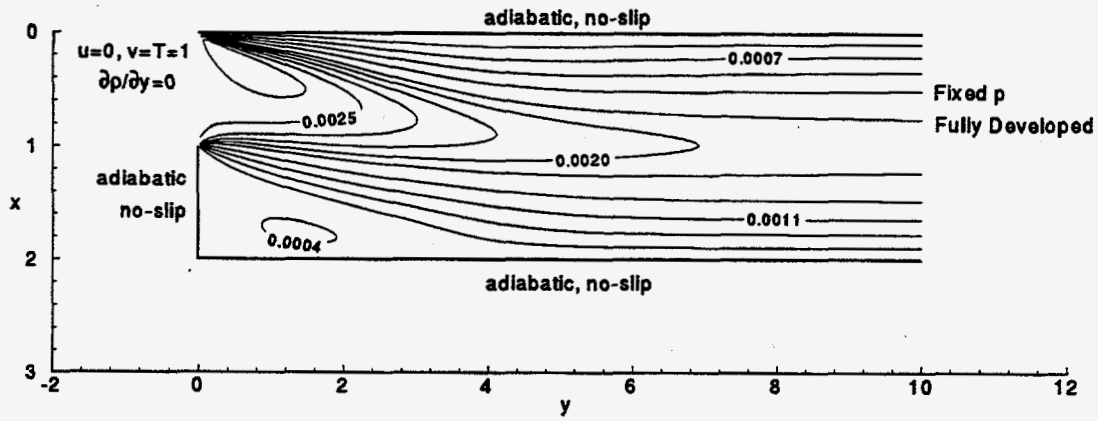

Figure 2.3: Mach number contours.

to the major fluid flow direction, and the x axis is vertical perpendicular to the flow. The flow enters the domain at the left boundary, proceeds horizontally, and exits at the right of the figure. Figure 2.2 shows the results of an actual flow simulation on this domain, indicating both the fluid velocity field and a set of streaklines to indicate the flow path. Finally, Figure 2.3 displays a contour plot of Mach number on the domain (again, from an actual simulation).

To complete the definition of the model problem requires specifying the independent parameters in the governing equations (such as γ, μ, *Re*, *etc.*). The majority of the results in this study will be based on a default problem; if a particular set of results deviates from the default parameters this will be noted in the section where the results are presented. The default problem uses the parameter values shown in Table 2.1.

The domain physically extends 10 units along the y direction, 2 units in the x direction. The channel walls along with the step are adiabatic, with the no-slip condition applied. Inlet conditions are $u = 0$, $v = T = 1$, and $\frac{\partial \rho}{\partial y} = 0$, with outlet conditions of fixed pressure and fully developed flow. The inlet Mach number is 0.0025, and a flow Reynolds number of 100 is used for all simulations unless otherwise noted.

$x = 2$	$y = 10$
$k = 1$	$\gamma = 1.4$
$\mu = 1$	$\lambda = -\frac{2}{3}$
$M_i = 0.0025$	$Fr = \infty$
$Re = 100$	$Pr = 0.70$
$Pe = 70$	

Table 2.1: Parameter values.

In the introduction, the case for parallel Navier-Stokes solutions was presented. It was argued that scientists and engineers need to solve ever larger and more complex problems. An example was presented indicating the computational difficulty involved in solving moderately sized two-dimensional problems. Additionally, the solution of compressible flow over a backward-facing step was selected as a non-trivial benchmark problem (an ASME "more realistic and difficult problem" [45]). In computational solutions there are two categories of difficulty; problems may be difficult to solve due to limited resources and/or they may be difficult due to characteristics that result in a poorly conditioned solution procedure. Recall that the best solution procedure is applicable to a wide variety of problems.

- The solution technique should be *robust*. It should yield a converged, accurate solution to a wide variety of problems without extensive user interaction. Numerically "stiff" solutions should be accommodated.

- The technique should be computationally efficient. It should enable the solution of large problems in a reasonable time.

- The technique should also be memory efficient to enable solutions to the desired problem

on available hardware.

This backward-facing step model problem is potentially challenging due to its "size" (a fine level of discretization will result in a large number of unknowns), and its numerical complexity (the algebraic system is difficult to solve due to its mathematical characteristics). One technique to enhance the spacial accuracy of a discrete solution relies on a refinement of the discretization of the domain to achieve the desired level of accuracy (resulting in increased memory requirements and longer execution times). For this study, the "size" of the model problem will be selected primarily to adequately challenge the hardware platform of interest, with accuracy being of peripheral interest. However, in all cases examined here, the base discretization refinement is sufficient that further refinement only produces marginal accuracy improvements. Achieving a numerically challenging model problem is accomplished by concentrating on the low Mach number compressible flow regime. The solution of the compressible flow equations in this regime requires a robust numerical technique to obtain a result. In general, this model problem becomes progressively more difficult to solve (numerically) as the Mach number is decreased below 0.2.

The low Mach number compressible flow problem is computationally challenging because very stiff Jacobians arise due to the appearance of large off-diagonal terms associated with the pressure dependencies in the momentum equation (proportional to $1/M_i^2$). These terms result in a poorly conditioned Jacobian (*i.e.*, a wide disparity in eigenvalues). Consequently, effective preconditioning is an extremely important consideration in these flow regimes. Although normally considered incompressible, compressible flow simulations of this flow regime are still important in situations where significant density variations occur (*i.e.*, flow with significant heat transfer effects).

In summary, the backward-facing step model problem provides a challenging test for the solution technique. It is scalable, providing an adequate growth of computation and memory requirements with problem size. Additionally, the model mandates a robust solution algorithm due to the wide disparity of eigenvalues (physically corresponding to multiple length scales) inherent in the problem definition. To further illustrate the difficulty of this model problem, popular iterative solution techniques employing ILU preconditioning do not yield convergence on this problem [1]. The remainder of this study will be devoted to efficient solutions of this model problem on various architectures of interest, focusing on the parallel aspects of the Newton-Krylov-Schwarz solution techniques.

2.2 The Governing Equations

In differential form, the general fluid mechanics equations of interest are [4]

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.1}$$

$$\frac{\partial}{\partial t}(\rho \mathbf{u}) + \nabla \cdot (\rho \mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \tau + \rho \mathbf{F} \tag{2.2}$$

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot (\rho \mathbf{u} e) = -p \nabla \cdot \mathbf{u} + \tau : \nabla \mathbf{u} - \nabla \cdot \mathbf{q} \tag{2.3}$$

$$\rho = \rho(p, e) \tag{2.4}$$

with the viscous stress tensor (in index notation) given as

$$\tau_{ij} = \lambda \frac{\partial u_k}{\partial x_k} \delta_{ij} + \mu \left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right). \tag{2.5}$$

All simulations performed in this research are targeted towards a steady-state solution to the flow equations. With this in mind, none of the transient terms are required in the

governing equations. This restriction simplifies the mass continuity equation (Equation 2.1) to

$$\nabla \cdot (\rho \mathbf{u}) = 0. \tag{2.6}$$

For two-dimensional rectilinear analysis, the viscous stress tensor assumes the form

$$\tau_{xx} = 2\mu \frac{\partial u}{\partial x} + \lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \tag{2.7}$$

$$\tau_{yy} = 2\mu \frac{\partial v}{\partial y} + \lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \tag{2.8}$$

$$\tau_{xy} = \tau_{yx} = \mu \left[\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right]. \tag{2.9}$$

The vector expression $\nabla \cdot \tau$ may be expressed as

$$\nabla \cdot \tau = \left\{ \begin{array}{l} \frac{\partial}{\partial x} \left[\lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + 2\mu \frac{\partial u}{\partial x} \right] + \frac{\partial}{\partial y} \left[\mu \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] \\ \frac{\partial}{\partial x} \left[\mu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[\lambda \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) + 2\mu \frac{\partial v}{\partial y} \right] \end{array} \right\}. \tag{2.10}$$

The Stokes hypothesis [49],

$$\lambda = -\frac{2}{3}\mu, \tag{2.11}$$

is used to further simplify the viscous stress tensor. The Stokes hypothesis is only rigorously valid for monatomic gases, but is widely assumed to hold for other fluids [4].

Finally, the form of the body force vector $\mathbf{F}$ must be determined. For this model problem,

$$\mathbf{F} = \mathbf{g}. \tag{2.12}$$

With the above assumptions and simplifications, the momentum equation becomes

$$\nabla \cdot (\rho \mathbf{u}\mathbf{u}) = -\nabla p + \nabla \cdot \tau + \rho \mathbf{g}, \tag{2.13}$$

or in x and y component form

$$\begin{aligned}\frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} = &-\frac{\partial p}{\partial x} + \frac{\partial}{\partial y}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right] \\ &+ \frac{\partial}{\partial x}\left[2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right] + \rho g_x\end{aligned} \tag{2.14}$$

$$\begin{aligned}\frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} = &-\frac{\partial p}{\partial y} + \frac{\partial}{\partial x}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right] \\ &+ \frac{\partial}{\partial y}\left[2\mu\frac{\partial v}{\partial y} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right] + \rho g_y.\end{aligned} \tag{2.15}$$

The energy equation (Equation 2.3),

$$\frac{\partial}{\partial t}(\rho e) + \nabla \cdot (\rho \mathbf{u} e) = -p\nabla \cdot \mathbf{u} + \tau : \nabla \mathbf{u} - \nabla \cdot \mathbf{q}, \tag{2.16}$$

may be simplified with the use of the mass continuity equation (Equation 2.6). Additionally, if the viscous dissipation term ($\tau : \nabla \mathbf{u}$) is neglected, and Fourier's Law [49] is used to represent $\mathbf{q}$,

$$\mathbf{q} = -k\nabla T, \tag{2.17}$$

and the energy equation becomes

$$\rho \frac{De}{Dt} = -p\nabla \cdot \mathbf{u} + \nabla \cdot (k\nabla T). \tag{2.18}$$

Further simplification requires an equation of state.

This study will be restricted to the compressible flow of ideal (perfect) gases. Thus, the characteristics of an ideal gas may be applied to establish an equation of state for the fluid. For ideal gases, the following relationships describe the thermodynamic behavior [49].

$$p = \rho RT \tag{2.19}$$

$$e = e(\rho, T) = c_v T \tag{2.20}$$

$$c_v \equiv \left.\frac{De}{DT}\right|_{\rho} \tag{2.21}$$

With these relationships, the energy equation may be expressed as

$$c_v \nabla \cdot (\rho \mathbf{u} T) = -p \nabla \cdot \mathbf{u} + \nabla \cdot (k \nabla T). \tag{2.22}$$

This completes the derivation of the governing equations, which can be summarized with the following set.

$$\nabla \cdot (\rho \mathbf{u}) = 0 \tag{2.23}$$

$$\begin{aligned} \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} &= -\frac{\partial p}{\partial x} + \frac{\partial}{\partial y}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right] \\ &\quad + \frac{\partial}{\partial x}\left[2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right] + \rho g_x \end{aligned} \tag{2.24}$$

$$\begin{aligned} \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} &= -\frac{\partial p}{\partial y} + \frac{\partial}{\partial x}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right] \\ &\quad + \frac{\partial}{\partial y}\left[2\mu\frac{\partial v}{\partial y} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right] + \rho g_y \end{aligned} \tag{2.25}$$

$$c_v \nabla \cdot (\rho \mathbf{u} T) = -p \nabla \cdot \mathbf{u} + \nabla \cdot (k \nabla T) \tag{2.26}$$

$$p = \rho RT \tag{2.27}$$

$$e = c_v T \tag{2.28}$$

$$c_v \equiv \left.\frac{De}{DT}\right|_\rho \tag{2.29}$$

2.2.1 Non-Dimensionalization of the Governing Equations

To non-dimensionalize the governing equations, the dimensionless parameters given in Table 2.2 are used [5].

$u_i' = \frac{u_i}{u_o}$	$x_i' = \frac{x_i}{L}$	$p' = \frac{p}{\rho_o u_o^2}$
$T' = \frac{T}{T_i}$	$\rho' = \frac{\rho}{\rho_o}$	$k' = \frac{k}{k_o}$
$\gamma = \frac{c_p}{c_v}$	$\mu' = \frac{\mu}{\mu_o}$	$c_p - c_v = R$
$M_i = \frac{u_o}{\sqrt{\gamma R T_i}}$	$Fr = \frac{u_o}{\sqrt{g_i L}}$	$Re = \frac{\rho_o u_o L}{\mu_o}$
$Pr = \frac{\mu_o c_p}{k_o}$	$Pe = RePr = \frac{\rho_o c_p u_o L}{k_o}$	

Table 2.2: Dimensionless Parameters.

Using these parameters, the continuity equation is unchanged. This equation assumes the form (after dropping the primes)

$$\nabla \cdot (\rho \mathbf{u}) = 0. \tag{2.30}$$

A similar process results in the dimensionless momentum equations, which take the form (after dropping the primes)

$$\begin{aligned} \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} &= -\frac{\partial p}{\partial x} + \frac{1}{Re}\left\{\frac{\partial}{\partial y}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right]\right. \\ &\quad \left. + \frac{\partial}{\partial x}\left[2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right]\right\} + \frac{\rho}{{Fr_x}^2} \\ \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} &= -\frac{\partial p}{\partial y} + \frac{1}{Re}\left\{\frac{\partial}{\partial x}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right]\right. \end{aligned} \tag{2.31}$$

$$+\frac{\partial}{\partial y}\left[2\mu\frac{\partial v}{\partial y}-\frac{2}{3}\mu\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)\right]\Bigg\}+\frac{\rho}{{Fr_y}^2}. \tag{2.32}$$

The energy equation becomes (again, after dropping the primes)

$$\nabla\cdot(\rho\mathbf{u}T)=\frac{\gamma}{Pe}\nabla\cdot(k\nabla T)-\gamma(\gamma-1)M_i^2 p\nabla\cdot\mathbf{u}. \tag{2.33}$$

Finally, the equation of state becomes

$$p=\frac{\rho T}{\gamma M_i^2}. \tag{2.34}$$

This result completes the non-dimensionalization process of the governing equations for the two-dimensional backward-facing step problem. At this point, it is useful to summarize the assumptions made in the theoretical development process before moving on to the finite volume approximations.

1. The development assumed a two-dimensional rectangular coordinate system (right-handed).

2. Only steady-state solutions were desired.

3. Thermal buoyancy effects were ignored ($Fr_x = Fr_y = \infty$).

4. Heat generation within the fluid (by chemical reactions, *etc.*), is ignored.

5. The equations are only valid for ideal (perfect) gases. The following simplifications were used as a result of this limitation.

$$p = \rho RT \tag{2.35}$$

$$e = c_v T \tag{2.36}$$

$$c_v \equiv \left.\frac{De}{DT}\right|_\rho \tag{2.37}$$

$$c_p - c_v = R \tag{2.38}$$

6. The Stokes hypothesis ($\lambda = -\frac{2}{3}\mu$) applies.

7. Fourier's Law applies ($\mathbf{q} = -k\nabla T$).

8. Viscous energy dissipation ($\tau : \nabla \mathbf{u}$) is ignored.

2.2.2 Discretization of the Governing Equations

In the introduction to this study, the problem of efficient and robust computational solutions to the Navier-Stokes equations was presented. Several solution techniques were discussed, all originated with the full Navier-Stokes equations. It was argued that these equations were too complex (and computationally challenging) to solve directly. The previous section discussed a sequence of simplification operations that may be performed on the base Navier-Stokes equations to reduce the complexity of the equations without significantly degrading the accuracy of the solution for a specific problem set (in this case, compressible fluid flow over a backward-facing step domain). However, at this point, the governing equations are still in the form of non-linear partial differential equations (PDEs). In general, the resulting governing equations cannot be simplified sufficiently to enable solution in functional form.

To employ a digital computer for the solution of a system of PDEs, the problem must be mapped into an equivalent discrete problem. For this specific case, the simplified non-linear partial differential equations developed in the previous section must be transformed to discrete expressions to enable a computational solution.

Discrete approximations are usually based on the subdivision of the problem domain into a computation domain consisting of a mesh of discrete computational cells (elements or volumes) that faithfully cover the problem domain. In each of these cells, the governing equations are approximated by a set of non-linear algebraic equations. The method used to obtain the discretization and the set of algebraic equations over the computational domain depends on the specifics of the problem and other factors. Several methods commonly used include:

- the finite difference method,
- the finite volume method,
- the finite element method, and
- spectral methods.

Each method has advantages and disadvantages; some are better suited than others for given classes of problems. Further complicating the choice of discretization method is the choice of the method used to solve the algebraic system resulting from the discretization process and initial/boundary conditions. The algebraic system may be cast and solved *explicitly*, *implicitly*, or using some combination of the two. Implicit techniques are based on a fully-coupled solution of the cell equations, where a system of algebraic equations are solved each time step or iteration level. Explicit methods are generally simpler than implicit methods. The discrete approximation of the governing equation is cast such that only one term of each cell equation is unknown. A semi-implicit technique casts some terms at the new time (or iterate), and others at the old time; as such it falls between implicit and explicit and exhibits solution characteristics of both techniques. In common with implicit techniques, semi-implicit methods require simultaneous solution of the system.

Implicit techniques are of interest as they allow a fully-coupled solution of the equations and are not stability limited to a *Courant* (or similar) wavespeed criterion. Explicit techniques do not lend themselves to the solution of problems with a wide disparity of time (length) scales, because the shortest wavelength phenomena must be resolved during the solution for stability. While the advantage of a fully-coupled solution is of peripheral interest to this study, basing this work on implicit methods allows the application of its contributions to problems of this type. Implicit methods, and more specifically, Newton-Krylov techniques are increasingly being investigated for computational fluid dynamics (CFD) applications due to the advantages of full coupling of all variables and equations, rapid non-linear convergence, and moderate memory requirements [11, 50, 51, 52].

Given the complete representation of the governing equations in dimensionless form, an implicit form of the finite volume technique can be selected to approximate the equations on the computational domain.

2.2.3 The Finite Volume Approximation of the Governing Equations

The finite volume approximation method is based on the concept that the physical domain, Ω, can be subdivided into E finite sized subdomains Ω_e, such that

$$\Omega = \bigcup_{e=1}^{E} \Omega_e. \tag{2.39}$$

The method requires that there are neither spacial overlaps between the Ω_e subdomains (*i.e.*, for any two sub-volumes Ω_i and Ω_j, $\Omega_i \cap \Omega_j = \emptyset$), or holes in Ω (*i.e.*, Ω is composed of disjoint closed subsets, Ω_e). Then, if each of the five equations describing the flow can be represented as $\mathcal{L}_i(\mathbf{x}) = 0$, where $\mathbf{x}$ is a vector of the dependent variables, the subdomain

method of weighted residuals [53] leads to the expression

$$\int_{\Omega} \mathcal{L}_i(\mathbf{x})\, d\Omega = \bigcup_{e=1}^{E} \int_{\Omega_e} \mathcal{L}_i(\mathbf{x})\, d\Omega = 0, \qquad i = 1, \ldots, 5. \tag{2.40}$$

Thus, for each Ω_e,

$$\int_{\Omega_e} \mathcal{L}_i(\mathbf{x})\, d\Omega = 0, \qquad i = 1, \ldots, 5. \tag{2.41}$$

From each of these equations, an algebraic expression is then obtained describing the relationship between the dependent variables, $\mathbf{x}$, in the sub-volume Ω_e. As an example of this process, consider the continuity equation

$$\mathcal{L}_1(\mathbf{x}) = \frac{\partial \rho u}{\partial x} + \frac{\partial \rho v}{\partial y} = 0. \tag{2.42}$$

This expression may be cast in divergence form

$$\mathcal{L}_1(\mathbf{x}) = \nabla \cdot \rho \mathbf{u} = 0. \tag{2.43}$$

For sub-volume Ω_e, the finite volume expression for the continuity equation becomes

$$\int_{\Omega_e} \mathcal{L}_1(\mathbf{x})\, d\Omega = \int_{\Omega_e} \nabla \cdot \rho \mathbf{u}\, d\Omega = 0. \tag{2.44}$$

The application of Gauss' Theorem for a given vector $\mathbf{v}$ contained within a volume V having a bounding surface S with outward unit normal $\hat{\mathbf{n}}$ gives the identity

$$\int_{V} \nabla \cdot \mathbf{v}\, dV \equiv \oint_{S} (\hat{\mathbf{n}} \cdot \mathbf{v})\, dS. \tag{2.45}$$

Thus,

$$\int_{\Omega_e} \nabla \cdot \rho \mathbf{u}\, d\Omega = \oint_{S=\partial\Omega_e} \hat{\mathbf{n}} \cdot \rho \mathbf{u}\, dS = 0, \tag{2.46}$$

where $\partial\Omega_e$ denotes the boundary (surface) of element Ω_e.

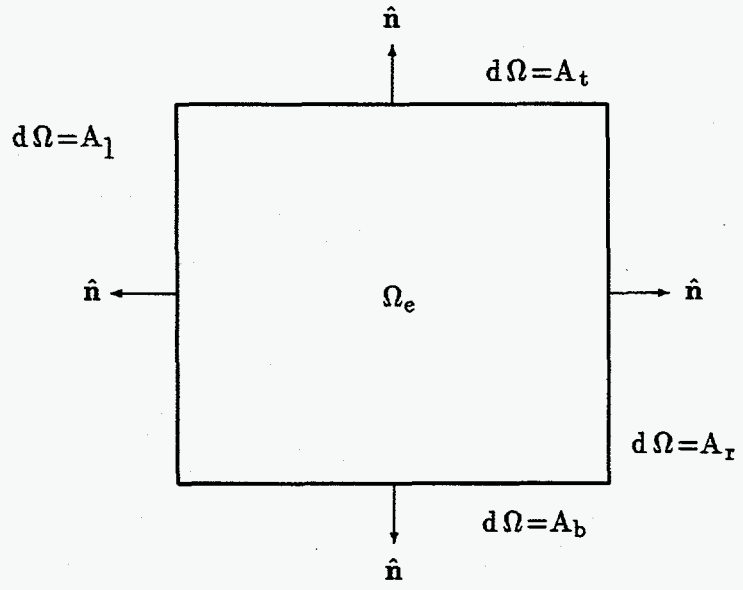

Figure 2.4: Pictorial representation of a finite volume, Ω_e, assuming a rectilinear two-dimensional discretization.

By inspection (see Figure 2.4), Equation 2.46 may be expressed in the algebraic form

$$\oint_{S=\partial\Omega_e} \hat{\mathbf{n}} \cdot \rho \mathbf{u}\, dS = (\rho u)_r A_r - (\rho u)_l A_l + (\rho v)_t A_t - (\rho v)_b A_b = 0. \tag{2.47}$$

In a similar manner, the remaining governing equations (2.30–2.34) may be cast into algebraic form. The resulting non-linear algebraic system, when assembled for all E values, forms the system that is the input to the Newton-Krylov algorithm.

Given the theory of the subdomain method of weighted residuals (*i.e.*, the "finite volume technique"), it is now possible to rigorously derive the discrete approximations to the

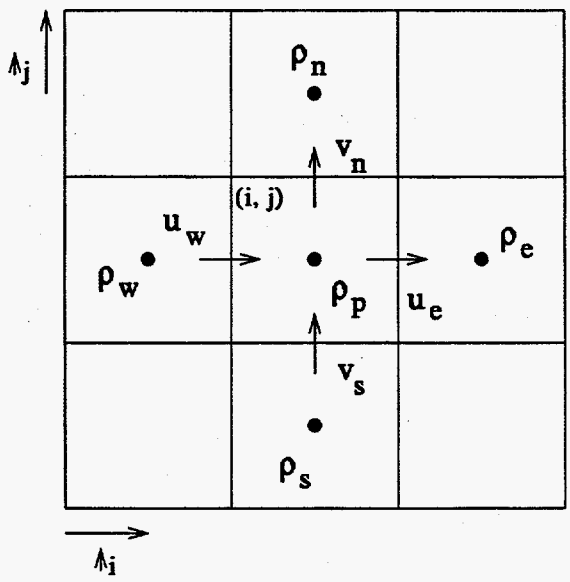

Figure 2.5: The computational cell used for the development of the mass conservation equation approximation.

simplified non-linear partial differential governing equation set previously developed.

The Mass Conservation Equation

The mass conservation equation is straightforward to develop from the dimensionless governing equation

$$\nabla \cdot (\rho \mathbf{u}) = 0. \tag{2.48}$$

The finite volume process begins with integrating this equation over a finite volume consisting of a computational cell (shown in Figure 2.5)

$$\int_V \nabla \cdot (\rho \mathbf{u}) \; dV = 0. \tag{2.49}$$

With the use of Gauss' Theorem, this volume integral can be converted to a surface integral

$$\oint_S \rho \mathbf{u} \cdot \hat{\mathbf{n}} \, dS = 0. \tag{2.50}$$

Evaluating this integral on the computational cell results in the equation

$$(\rho u)_e A_e - (\rho u)_w A_w + (\rho v)_n A_n - (\rho v)_s A_s = 0. \tag{2.51}$$

At this point, it is necessary to examine the $(\rho u)_k$ terms, above. At first glance, one is tempted to simplify these to $\rho_k u_k$. If this simplification is performed, the computational cell assumes the form shown in Figure 2.6. Furthermore, for the purpose of illustration, consider only the x component of the variables and assume $A_e = A_w$. Thus, on the grid shown, the expression

$$\rho_e u_e = \rho_w u_w, \tag{2.52}$$

describes the continuity equation in the x direction for cell (i, j). At this point, imagine defining a new cell that is offset $\frac{1}{2}$ cell width in the positive i direction. The vertices of this cell lie on the points ρ_s, ρ_n and the points ρ_{see}, ρ_{nee} (not shown in in Figure 2.6 for simplicity). For the purposes of this discussion, this new offset cell will be named cell $(i + \frac{1}{2}, j)$. For cell $(i + \frac{1}{2}, j)$, the equivalent expression to (2.52) is

$$\rho_{ee} u_{ee} = \rho_p u_p. \tag{2.53}$$

As an example, consider a uniform flow-field where the quantity $\rho_k u_k$ equals 100 over the domain. This statement satisfies Equations 2.52 and 2.53 for all cells on the domain. Also

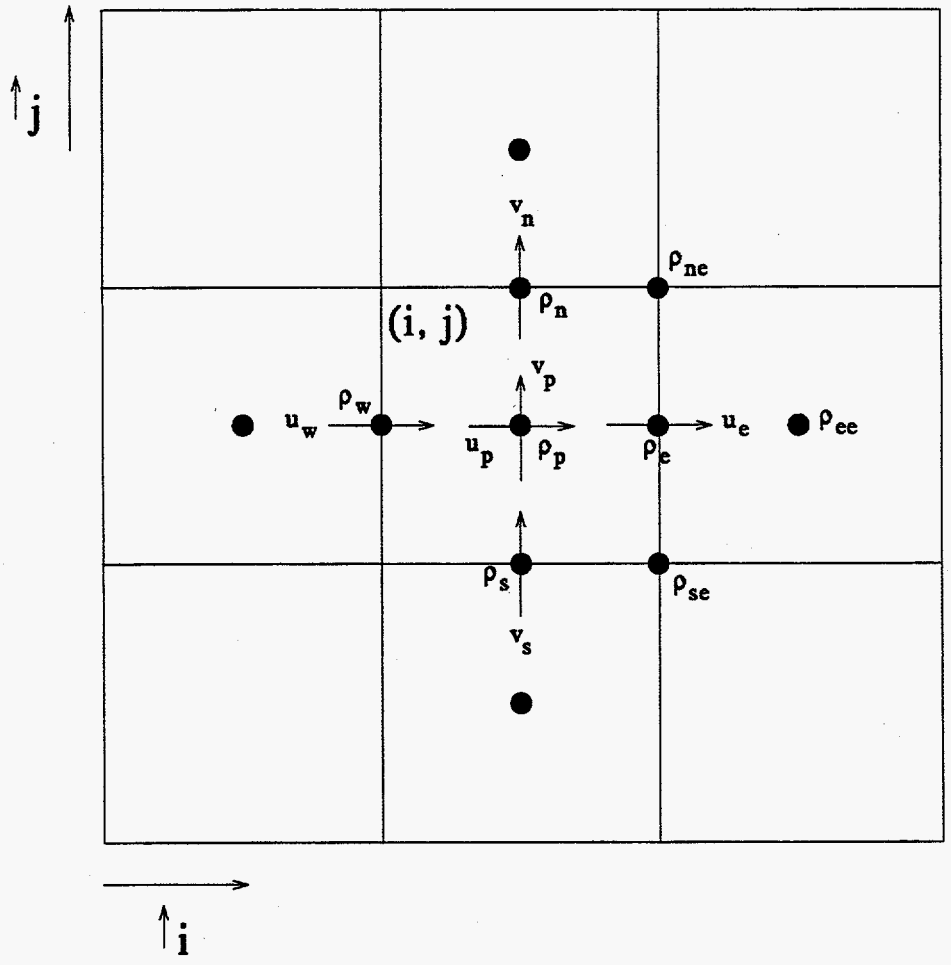

Figure 2.6: The computational cell modified for coincident velocity and density.

consider a flow-field where the quantity $\rho_k u_k$ equals 100 for just the cells of (i,j) form, and let $\rho_k u_k$ equal -50 for the cells of $(i+\frac{1}{2},j)$ form. Again, the continuity equations are satisfied exactly. However, this result is clearly non-physical and an obvious error. The difficulty with this scheme is the lack of coupling between the (i,j) cells and the $(i+\frac{1}{2},j)$ cells in the discrete approximation. This difficulty (often coined the $2\Delta x$ *instability*, because the wavelength of the anomaly is $2\Delta x$) can be eliminated with the use of a higher-order discretization technique of the governing system or the staggered velocity-density grid shown in Figure 2.5. The staggered grid will be employed in this study because the higher-order discretization techniques are somewhat more complex to implement. For further study of this difficulty, Patankar [54] provides an excellent presentation of coincident versus staggered grid discretizations.

Consulting the staggered grid shown in Figure 2.5, it is obvious that the velocities u_k are located on the "cell" faces, with ρ_k located in the cell centers. As these quantities are not located coincidently, special treatment is necessary. Considering cell (i,j), the u_k values are correctly located at the cell boundaries. However, the ρ_k values are not (the only ρ value that actually "belongs" to cell (i,j) is ρ_p, at the center of the cell). As a first attempt, one may simply "average" the ρ_k appropriately. To express $(\rho u)_w$, a linear average of ρ_w and ρ_p could be used

$$(\rho u)_w = \frac{1}{2}(\rho_w + \rho_p)u_w. \tag{2.54}$$

With convective terms, such as $(\rho u)_k$, it is possible to encounter another non-physical result similar to that encountered in the previous example. Again, consider a simplified

one-dimensional form of the above continuity equation with $A_e = A_w$

$$(\rho u)_e = (\rho u)_w. \tag{2.55}$$

Furthermore, employing Equation 2.54 to express the density results in

$$\rho_w u_w - \rho_e u_e = \rho_p (u_e - u_w). \tag{2.56}$$

Consider a flow in a cell where $u_e = 2$, $u_w = 1$, $\rho_p = 3$, and $\rho_w = 1$. For the continuity equation on this cell to be satisfied, ρ_e must equal -1. Clearly a negative density is a non-physical result. Furthermore, ρ_p must fall between ρ_w and ρ_e. From this result, a simple linear interpolation of density (such as Equation 2.54) is clearly insufficient. Several techniques may be employed to remedy this behavior [54], an upwind scheme was chosen for this study due to its simplicity.

The upwind scheme is based on the "upwinding," or *backward differencing*, of the convected variable. Physically, in cell (i, j), the velocity u_w is "convecting" a density. Consider the velocity u_w, as shown in Figure 2.5. This velocity signifies that a volume of fluid in cell $(i-1, j)$ is moving into cell (i, j), with a velocity u_w. This volume of fluid has a density ρ_w, not some averaged value. Thus, the correct form of $(\rho u)_w$ is $\rho_w u_w$ (for u_w moving from $(i-1, j)$ to (i, j)). One must also allow for a negative u_w, which complicates the expression

$$(\rho u)_w = \rho_w [\![0, u_w]\!] - \rho_p [\![0, -u_w]\!]. \tag{2.57}$$

The notation $[\![a, b]\!]$ denotes the maximum value of (a, b). This expression clearly results in the correct interpretation of $(\rho u)_w$, independent of the direction of u_w. Comparing this

result with the previous example using interpolated density on the cell faces, it is clear that non-physical densities are prevented.

Upwinding the convective terms results in

$$(\rho u)_w = \rho_w [\![0, u_w]\!] - \rho_p [\![0, -u_w]\!] \equiv \text{cfluxw} \tag{2.58}$$

$$(\rho u)_e = \rho_p [\![0, u_e]\!] - \rho_e [\![0, -u_e]\!] \equiv \text{cfluxe} \tag{2.59}$$

$$(\rho u)_n = \rho_p [\![0, u_n]\!] - \rho_n [\![0, -u_n]\!] \equiv \text{cfluxn} \tag{2.60}$$

$$(\rho u)_s = \rho_s [\![0, u_s]\!] - \rho_p [\![0, -u_s]\!] \equiv \text{cfluxs}. \tag{2.61}$$

This result may be further simplified by realizing that for an arbitrary cell (i, j),

$$A_w = A_e = \Delta y_j$$

$$A_n = A_s = \Delta x_i,$$

resulting in the simplified form

$$(\text{cfluxe} - \text{cfluxw})\Delta y_j + (\text{cfluxn} - \text{cfluxs})\Delta x_i = 0. \tag{2.62}$$

The x-Momentum Equation

The momentum equations developed previously

$$\begin{aligned} \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} &= -\frac{\partial p}{\partial x} + \frac{1}{Re}\left\{\frac{\partial}{\partial y}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right]\right. \\ &\quad \left. + \frac{\partial}{\partial x}\left[2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right]\right\} + \frac{\rho}{{Fr_x}^2} \\ \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} &= -\frac{\partial p}{\partial y} + \frac{1}{Re}\left\{\frac{\partial}{\partial x}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right]\right. \end{aligned} \tag{2.63}$$

$$+\frac{\partial}{\partial y}\left[2\mu\frac{\partial v}{\partial y}-\frac{2}{3}\mu\left(\frac{\partial u}{\partial x}+\frac{\partial v}{\partial y}\right)\right]\Bigg\}+\frac{\rho}{{Fr_y}^2}, \tag{2.64}$$

may be written in vector form

$$\nabla\cdot(\rho\mathbf{uu})=-\nabla\cdot\left(p\hat{I}\right)+\frac{1}{Re}\nabla\cdot\tau+\frac{\rho}{Fr^2}\hat{I}. \tag{2.65}$$

Integrating over a finite volume results in

$$\int_V\nabla\cdot(\rho\mathbf{uu})\;dV=-\int_V\nabla\cdot\left(p\hat{I}\right)\;dV+\frac{1}{Re}\int_V\nabla\cdot\tau\;dV+\int_V\frac{\rho}{Fr^2}\hat{I}\;dV. \tag{2.66}$$

Gauss' Theorem allows this result to be rewritten as

$$\oint_S(\rho\mathbf{uu})\cdot\hat{\mathbf{n}}\;dS=-\oint_S p\hat{I}\cdot\hat{\mathbf{n}}\;dS+\frac{1}{Re}\oint_S\tau\cdot\hat{\mathbf{n}}\;dS+\int_V\frac{\rho}{Fr^2}\hat{I}\;dV, \tag{2.67}$$

or for the x-component

$$\oint_S(\rho\mathbf{u})\,u\cdot\hat{\mathbf{n}}\;dS=-\oint_S p\hat{\imath}\cdot\hat{\mathbf{n}}\;dS+\frac{1}{Re}\oint_S\tau_x\cdot\hat{\mathbf{n}}\;dS+\int_V\frac{\rho}{{Fr_x}^2}\;dV. \tag{2.68}$$

The derivation of the first term is rather complicated. Several steps are required to achieve an appropriate approximation; these steps will be discussed in detail. The first term may be approximated as

$$\begin{aligned}&\oint_S(\rho\mathbf{u})\,u\cdot\hat{\mathbf{n}}\;dS\approx\\&\quad\hat{\mathbf{n}}_w\cdot(\rho\mathbf{u})_w u_w A_w+\hat{\mathbf{n}}_e\cdot(\rho\mathbf{u})_e u_e A_e+\hat{\mathbf{n}}_n\cdot(\rho\mathbf{u})_n u_n A_n+\hat{\mathbf{n}}_s\cdot(\rho\mathbf{u})_s u_s A_s\\&=\;(\text{cflxe})A_e-(\text{cflxw})A_w+(\text{cflxn})A_n-(\text{cflxs})A_s,\end{aligned} \tag{2.69}$$

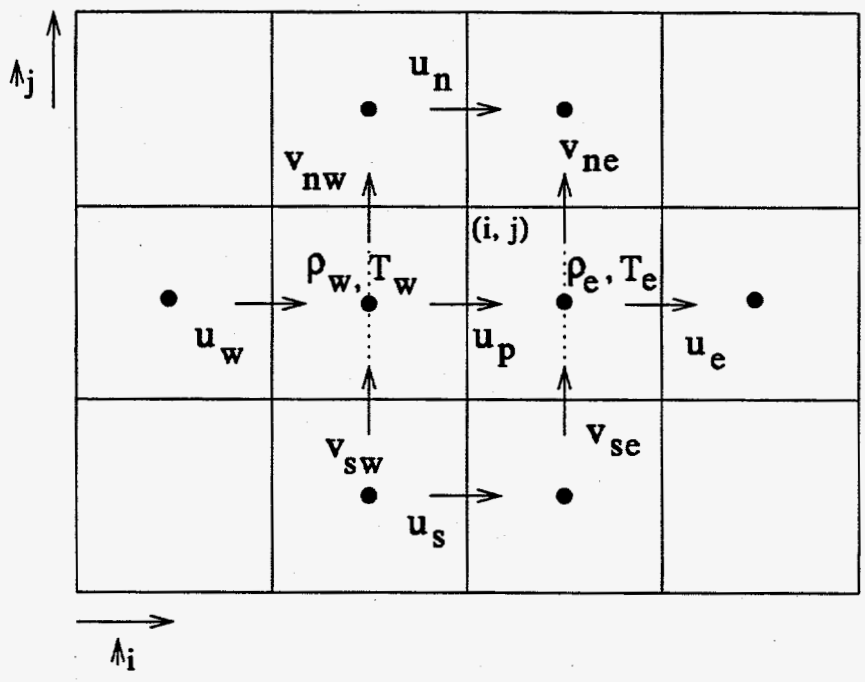

Figure 2.7: The x-momentum stencil.

where the areas of the cells are

$$A_e = A_w = \Delta y_j$$

$$A_n = A_s = \frac{1}{2}(\Delta x_i + \Delta x_{i-1}).$$

Upwinding is used to develop the convection approximations

$$(\rho \mathbf{u})_e u_e = (\rho \mathbf{u})_E u_{\mathrm{up}} = u_p [\![0, (\rho \mathbf{u})_E]\!] - u_e [\![0, -(\rho \mathbf{u})_E]\!] = \mathrm{cflxe}$$

$$\text{where } (\rho \mathbf{u})_E = \rho_e \frac{u_p + u_e}{2} \tag{2.70}$$

$$(\rho \mathbf{u})_w u_w = (\rho \mathbf{u})_W u_{\mathrm{up}} = u_w [\![0, (\rho \mathbf{u})_W]\!] - u_p [\![0, -(\rho \mathbf{u})_W]\!] = \mathrm{cflxw}$$

$$\text{where } (\rho \mathbf{u})_W = \rho_w \frac{u_p + u_w}{2} \tag{2.71}$$

$$(\rho \mathbf{u})_n u_n = (\rho \mathbf{u})_N u_{\mathrm{up}} = u_p [\![0, (\rho \mathbf{u})_N]\!] - u_n [\![0, -(\rho \mathbf{u})_N]\!] = \mathrm{cflxn}$$

$$\text{where } (\rho \mathbf{u})_N = \rho_n \frac{\Delta x_i v_{nw} + \Delta x_{i-1} v_{ne}}{\Delta x_i + \Delta x_{i-1}} \tag{2.72}$$

$$(\rho u)_s u_s = (\rho \mathbf{u})_S u_{\text{up}} = u_s [\![0, (\rho \mathbf{u})_S]\!] - u_p [\![0, -(\rho \mathbf{u})_S]\!] = \text{cflxs}$$

$$\text{where } (\rho \mathbf{u})_S = \rho_s \frac{\Delta x_i v_{sw} + \Delta x_{i-1} v_{se}}{\Delta x_i + \Delta x_{i-1}}, \tag{2.73}$$

where the notation $[\![a, b]\!]$ represents $\max(a, b)$. This result may be further simplified by noting that

$$\begin{aligned} \rho_n &= \frac{\Delta y_j \left(\Delta x_i \rho_w + \Delta x_{i-1} \rho_e\right) + \Delta y_{j+1} \left(\Delta x_i \rho_{nw} + \Delta x_{i-1} \rho_{ne}\right)}{\left(\Delta x_i + \Delta x_{i-1}\right)\left(\Delta y_j + \Delta y_{j+1}\right)} \\ \rho_s &= \frac{\Delta y_j \left(\Delta x_i \rho_{sw} + \Delta x_{i-1} \rho_{se}\right) + \Delta y_{j-1} \left(\Delta x_i \rho_w + \Delta x_{i-1} \rho_e\right)}{\left(\Delta x_i + \Delta x_{i-1}\right)\left(\Delta y_j + \Delta y_{j-1}\right)}. \end{aligned} \tag{2.74}$$

The pressure term takes the form

$$\oint_S \hat{\mathbf{n}} \cdot p\hat{\imath}\, dS \approx p_e A_e - p_w A_w = (p_e - p_w)\, \Delta y_j. \tag{2.75}$$

The viscous term may be expressed as

$$\begin{aligned} &\oint_S \hat{\mathbf{n}} \cdot \tau_x \, dS \approx \\ &\qquad \hat{\mathbf{n}}_e \cdot (\tau_x \cdot \hat{\imath})_e A_e + \hat{\mathbf{n}}_w \cdot (\tau_x \cdot \hat{\imath})_w A_w + \\ &\qquad \hat{\mathbf{n}}_n \cdot (\tau_x \cdot \hat{\jmath})_n A_n + \hat{\mathbf{n}}_s \cdot (\tau_x \cdot \hat{\jmath})_s A_s \\ &= (\tau_x \cdot \hat{\imath})_e A_e - (\tau_x \cdot \hat{\imath})_w A_w + \\ &\qquad (\tau_x \cdot \hat{\jmath})_n A_n - (\tau_x \cdot \hat{\jmath})_s A_s. \end{aligned} \tag{2.76}$$

An expression for τ was developed previously

$$\tau_x = \left\{2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right), \mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right\} \tag{2.77}$$

$$\tau_y = \left\{\mu\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right), 2\mu\frac{\partial v}{\partial y} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right\}, \tag{2.78}$$

resulting in

$$(\tau_x \cdot \hat{i})_e = 2\mu_e\left(\frac{\partial u}{\partial x}\right)_e - \frac{2}{3}\mu_e\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)_e = \text{dflxe} \tag{2.79}$$

$$(\tau_x \cdot \hat{i})_w = 2\mu_w\left(\frac{\partial u}{\partial x}\right)_w - \frac{2}{3}\mu_w\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)_w = \text{dflxw} \tag{2.80}$$

$$(\tau_x \cdot \hat{j})_n = \mu_n\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)_n = \text{dflxn} \tag{2.81}$$

$$(\tau_x \cdot \hat{j})_s = \mu_s\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)_s = \text{dflxs}, \tag{2.82}$$

or using differencing for the discrete approximation of the differential terms

$$(\tau_x \cdot \hat{i})_e = 2\mu_e\left(\frac{u_e - u_p}{\Delta x_i}\right) - \frac{2}{3}\mu_e\left(\frac{u_e - u_p}{\Delta x_i} + \frac{v_{ne} - v_{se}}{\Delta y_j}\right) = \text{dflxe} \tag{2.83}$$

$$(\tau_x \cdot \hat{i})_w = 2\mu_w\left(\frac{u_p - u_w}{\Delta x_{i-1}}\right) - \frac{2}{3}\mu_w\left(\frac{u_p - u_w}{\Delta x_{i-1}} + \frac{v_{nw} - v_{sw}}{\Delta y_j}\right) = \text{dflxw} \tag{2.84}$$

$$(\tau_x \cdot \hat{j})_n = 2\mu_n\left(\frac{u_n - u_p}{\Delta y_{j+1} + \Delta y_j} + \frac{v_{ne} - v_{nw}}{\Delta x_i + \Delta x_{i-1}}\right) = \text{dflxn} \tag{2.85}$$

$$(\tau_x \cdot \hat{j})_s = 2\mu_s\left(\frac{u_p - u_s}{\Delta y_j + \Delta y_{j-1}} + \frac{v_{se} - v_{sw}}{\Delta x_i + \Delta x_{i-1}}\right) = \text{dflxs}. \tag{2.86}$$

This can be expressed over a general momentum cell as

$$\oint_S \hat{\mathbf{n}} \cdot \tau_x \, dS \approx$$
$$\left[2\mu_e\left(\frac{u_e - u_p}{\Delta x_i}\right) - \frac{2}{3}\mu_e\left(\frac{u_e - u_p}{\Delta x_i} + \frac{v_{ne} - v_{se}}{\Delta y_j}\right)\right] A_e -$$
$$\left[2\mu_w\left(\frac{u_p - u_w}{\Delta x_{i-1}}\right) - \frac{2}{3}\mu_w\left(\frac{u_p - u_w}{\Delta x_{i-1}} + \frac{v_{nw} - v_{sw}}{\Delta y_j}\right)\right] A_w +$$

$$\left[2\mu_n\left(\frac{u_n-u_p}{\Delta y_{j+1}+\Delta y_j}+\frac{v_{ne}-v_{nw}}{\Delta x_i+\Delta x_{i-1}}\right)\right]A_n -$$
$$\left[2\mu_s\left(\frac{u_p-u_s}{\Delta y_j+\Delta y_{j-1}}+\frac{v_{se}-v_{sw}}{\Delta x_i+\Delta x_{i-1}}\right)\right]A_s. \tag{2.87}$$

This result is easily verified to be identical to a normal incompressible central difference $\nabla \cdot \mathbf{u}$ representation when a uniform mesh ($\Delta x_i = \Delta x_{i+1}, \Delta y_j = \Delta y_{j+1}\ \forall i,j \in \Omega$), and μ = constant is used

$$\Delta x \Delta y\left[\frac{u_e-2u_p+u_w}{\Delta x^2}+\frac{u_n-2u_p+u_s}{\Delta y^2}\right]. \tag{2.88}$$

Finally, the body force term can be expressed as

$$\int_V \frac{\rho}{{Fr_x}^2}\hat{\imath}\,dV = \frac{1}{{Fr_x}^2}\int_V \rho\,dV \approx \frac{(\rho_e+\rho_w)}{4{Fr_x}^2}(\Delta x_{i-1}+\Delta x_i)\,\Delta y_j. \tag{2.89}$$

With the above approximations, the x-momentum equation assumes the form

$$[(\text{cflxe})A_e-(\text{cflxw})A_w+(\text{cflxn})A_n-(\text{cflxs})A_s]+[p_eA_e-p_wA_w]-$$
$$\frac{1}{Re}\left[(\tau_x\cdot\hat{\imath})_e A_e-(\tau_x\cdot\hat{\imath})_w A_w+(\tau_x\cdot\hat{\jmath})_n A_n-(\tau_x\cdot\hat{\jmath})_s A_s\right]-$$
$$\left[\frac{(\rho_e+\rho_w)}{2{Fr_x}^2}A_nA_w\right]=0. \tag{2.90}$$

The y-Momentum Equation

Using a similar process to that illustrated for the x-momentum equation, the y-momentum equation can be developed. In component form, the y-momentum equation can be expressed as (see Equation 2.67)

$$\oint_S(\rho\mathbf{u})v\cdot\hat{\mathbf{n}}\,dS = -\oint_S p\hat{\jmath}\cdot\hat{\mathbf{n}}\,dS+\frac{1}{Re}\oint_S\tau_y\cdot\hat{\mathbf{n}}\,dS+\int_V\frac{\rho}{{Fr_y}^2}\,dV. \tag{2.91}$$

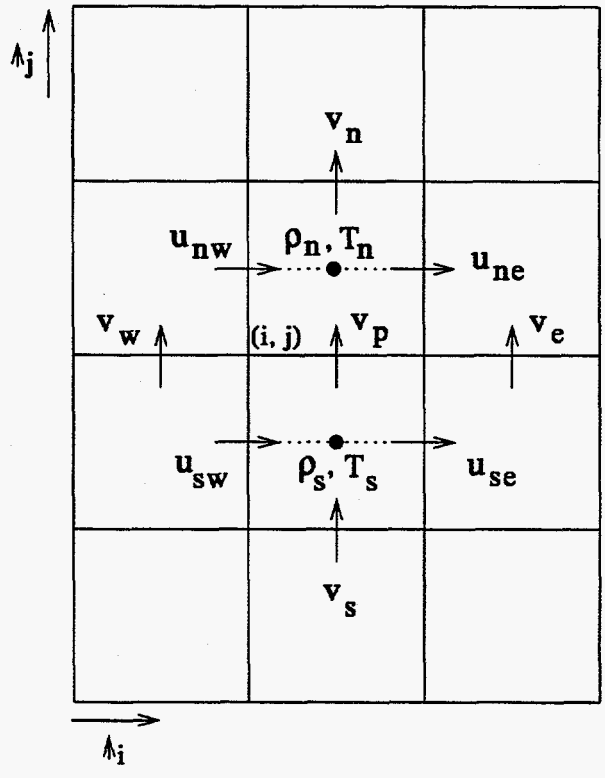

Figure 2.8: The y-momentum stencil.

Each of the terms in the above equation will be examined in a similar manner to the process followed for the x-momentum equation. The convective term is slightly different than the x-momentum representation

$$\oint_S (\rho \mathbf{u})\, v \cdot \hat{\mathbf{n}}\, dS \approx$$

$$\hat{\mathbf{n}}_w \cdot (\rho \mathbf{u})_w v_w A_w + \hat{\mathbf{n}}_e \cdot (\rho \mathbf{u})_e v_e A_e + \hat{\mathbf{n}}_n \cdot (\rho \mathbf{u})_n v_n A_n + \hat{\mathbf{n}}_s \cdot (\rho \mathbf{u})_s v_s A_s$$

$$= \quad (\text{cflxe}) A_e - (\text{cflxw}) A_w + (\text{cflxn}) A_n - (\text{cflxs}) A_s. \tag{2.92}$$

The areas of the cells are

$$A_e = A_w = \frac{1}{2}(\Delta y_j + \Delta y_{j-1})$$

$$A_n = A_s = \Delta x_i.$$

Upwinding is again used to develop the convection approximations

$$\begin{aligned}
(\rho \mathbf{u})_e v_e &= (\rho \mathbf{u})_E v_{\text{up}} = v_p [\![0, (\rho \mathbf{u})_E]\!] - v_e [\![0, -(\rho \mathbf{u})_E]\!] = \text{cflxe}, \\
&\qquad \text{where } (\rho \mathbf{u})_E = \rho_e \frac{\Delta y_{j-1} u_{ne} + \Delta y_j u_{se}}{\Delta y_j + \Delta y_{j-1}} \qquad (2.93) \\
(\rho \mathbf{u})_w v_w &= (\rho \mathbf{u})_W v_{\text{up}} = v_w [\![0, (\rho \mathbf{u})_W]\!] - v_p [\![0, -(\rho \mathbf{u})_W]\!] = \text{cflxw}, \\
&\qquad \text{where } (\rho \mathbf{u})_W = \rho_w \frac{\Delta y_{j-1} u_{nw} + \Delta y_j u_{sw}}{\Delta y_j + \Delta y_{j-1}} \qquad (2.94) \\
(\rho \mathbf{u})_n v_n &= (\rho \mathbf{u})_N v_{\text{up}} = v_p [\![0, (\rho \mathbf{u})_N]\!] - v_n [\![0, -(\rho \mathbf{u})_N]\!] = \text{cflxn}, \\
&\qquad \text{where } (\rho \mathbf{u})_N = \rho_n \frac{v_p + v_n}{2} \qquad (2.95) \\
(\rho \mathbf{u})_s v_s &= (\rho \mathbf{u})_S v_{\text{up}} = v_s [\![0, (\rho \mathbf{u})_S]\!] - v_p [\![0, -(\rho \mathbf{u})_S]\!] = \text{cflxs}, \\
&\qquad \text{where } (\rho \mathbf{u})_S = \rho_s \frac{v_p + v_s}{2}, \qquad (2.96)
\end{aligned}$$

where

$$\begin{aligned}
\rho_e &= \frac{\Delta x_{i+1} (\Delta y_{j-1} \rho_n + \Delta y_j \rho_s) + \Delta x_i (\Delta y_{j-1} \rho_{ne} + \Delta y_j \rho_{se})}{(\Delta x_i + \Delta x_{i+1})(\Delta y_j + \Delta y_{j-1})} \\
\rho_w &= \frac{\Delta x_{i-1} (\Delta y_{j-1} \rho_n + \Delta y_j \rho_s) + \Delta x_i (\Delta y_{j-1} \rho_{nw} + \Delta y_j \rho_{sw})}{(\Delta x_i + \Delta x_{i-1})(\Delta y_j + \Delta y_{j-1})}. \qquad (2.97)
\end{aligned}$$

The pressure term takes the form

$$\oint_S \hat{\mathbf{n}} \cdot p\hat{\jmath}\, dS \approx p_n A_n - p_s A_s = (p_n - p_s)\, \Delta x_i. \qquad (2.98)$$

The viscous term can be expressed in the form

$$\oint_S \hat{\mathbf{n}} \cdot \tau_y \, dS \approx$$

$$\hat{\mathbf{n}}_e \cdot (\tau_y \cdot \hat{\imath})_e \, A_e + \hat{\mathbf{n}}_w \cdot (\tau_y \cdot \hat{\imath})_w \, A_w +$$

$$\hat{\mathbf{n}}_n \cdot (\tau_y \cdot \hat{\jmath})_n \, A_n + \hat{\mathbf{n}}_s \cdot (\tau_y \cdot \hat{\jmath})_s \, A_s$$

$$= (\tau_y \cdot \hat{\imath})_e \, A_e - (\tau_y \cdot \hat{\imath})_w \, A_w +$$

$$(\tau_y \cdot \hat{\jmath})_n \, A_n - (\tau_y \cdot \hat{\jmath})_s \, A_s. \tag{2.99}$$

An expression for τ_y was developed previously (Equation 2.78)

$$\tau_y = \left\{ \mu \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right), 2\mu \frac{\partial v}{\partial y} - \frac{2}{3}\mu \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right\}, \tag{2.100}$$

resulting in

$$(\tau_y \cdot \hat{\imath})_e = \mu_e \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right)_e = \text{dflxe} \tag{2.101}$$

$$(\tau_y \cdot \hat{\imath})_w = \mu_w \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right)_w = \text{dflxw} \tag{2.102}$$

$$(\tau_y \cdot \hat{\jmath})_n = 2\mu_n \left(\frac{\partial v}{\partial y} \right)_n - \frac{2}{3}\mu_n \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right)_n = \text{dflxn} \tag{2.103}$$

$$(\tau_y \cdot \hat{\jmath})_s = 2\mu_s \left(\frac{\partial v}{\partial y} \right)_s - \frac{2}{3}\mu_s \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right)_s = \text{dflxs}, \tag{2.104}$$

or using differencing for the discrete approximation of the differential terms

$$(\tau_y \cdot \hat{\imath})_e = 2\mu_e \left(\frac{v_e - v_p}{\Delta x_{i+1} + \Delta x_i} + \frac{u_{ne} - u_{se}}{\Delta y_j + \Delta y_{j-1}} \right) = \text{dflxe} \tag{2.105}$$

$$(\tau_y \cdot \hat{\imath})_w = 2\mu_w \left(\frac{v_p - v_w}{\Delta x_i + \Delta x_{i-1}} + \frac{u_{nw} - u_{sw}}{\Delta y_j + \Delta y_{j-1}} \right) = \text{dflxw} \tag{2.106}$$

$$(\tau_y \cdot \hat{j})_n = 2\mu_n \left(\frac{v_n - v_p}{\Delta y_j}\right) - \frac{2}{3}\mu_n \left(\frac{u_{ne} - u_{nw}}{\Delta x_i} + \frac{v_n - v_p}{\Delta y_j}\right) = \text{dflxn} \qquad (2.107)$$

$$(\tau_y \cdot \hat{j})_s = 2\mu_s \left(\frac{v_p - v_s}{\Delta y_{j-1}}\right) - \frac{2}{3}\mu_s \left(\frac{u_{se} - u_{sw}}{\Delta x_i} + \frac{v_p - v_s}{\Delta y_{j-1}}\right) = \text{dflxs}. \qquad (2.108)$$

This can be expressed over a general momentum cell as

$$\begin{aligned}
&\oint_S \hat{\mathbf{n}} \cdot \tau_y \, dS \approx \\
&\left[2\mu_e \left(\frac{v_e - v_p}{\Delta x_{i+1} + \Delta x_i} + \frac{u_{ne} - u_{se}}{\Delta y_j + \Delta y_{j-1}}\right)\right] A_e - \\
&\left[2\mu_w \left(\frac{v_p - v_w}{\Delta x_i + \Delta x_{i-1}} + \frac{u_{nw} - u_{sw}}{\Delta y_j + \Delta y_{j-1}}\right)\right] A_w + \\
&\left[2\mu_n \left(\frac{v_n - v_p}{\Delta y_j}\right) - \frac{2}{3}\mu_n \left(\frac{u_{ne} - u_{nw}}{\Delta x_i} + \frac{v_n - v_p}{\Delta y_j}\right)\right] A_n - \\
&\left[2\mu_s \left(\frac{v_p - v_s}{\Delta y_{j-1}}\right) - \frac{2}{3}\mu_s \left(\frac{u_{se} - u_{sw}}{\Delta x_i} + \frac{v_p - v_s}{\Delta y_{j-1}}\right)\right] A_s. \qquad (2.109)
\end{aligned}$$

Again, as was seen from the x-momentum development, this result is easily verified to be identical to a normal incompressible central difference $\nabla \cdot \mathbf{u}$ representation when a uniform mesh ($\Delta x_i = \Delta x_{i+1}, \Delta y_j = \Delta y_{j+1}\ \forall i, j \in \Omega$), and $\mu =$ constant is used

$$\Delta x \Delta y \left[\frac{v_e - 2v_p + v_w}{\Delta x^2} + \frac{v_n - 2v_p + v_s}{\Delta y^2}\right]. \qquad (2.110)$$

Finally, the body force term can be expressed as

$$\int_V \frac{\rho}{{Fr_y}^2} \hat{j} \, dV = \frac{1}{{Fr_y}^2} \int_V \rho \, dV \approx \frac{(\rho_n + \rho_s)}{4{Fr_y}^2} \Delta x_i (\Delta y_{j-1} + \Delta y_j). \qquad (2.111)$$

With the above approximations, the y-momentum equation assumes the form

$$[(\text{cflxe})A_e - (\text{cflxw})A_w + (\text{cflxn})A_n - (\text{cflxs})A_s] + [p_n A_n - p_s A_s] -$$

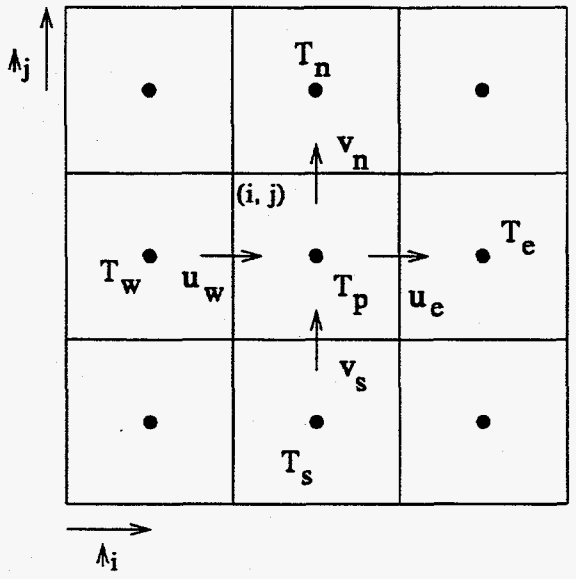

Figure 2.9: The energy stencil.

$$\frac{1}{Re}\left[(\tau_y \cdot \hat{\imath})_e A_e - (\tau_y \cdot \hat{\imath})_w A_w + (\tau_y \cdot \hat{\jmath})_n A_n - (\tau_y \cdot \hat{\jmath})_s A_s\right] - \left[\frac{(\rho_n + \rho_s)}{2Fr_y{}^2} A_n A_w\right] = 0. \tag{2.112}$$

The Energy Equation

Using a similar process to that illustrated for the x-momentum equation, the discrete energy equation may be developed. Consider the general form of the energy equation developed earlier

$$\nabla \cdot (\rho \mathbf{u} T) = \frac{\gamma}{Pe} \nabla \cdot (k \nabla T) - \gamma (\gamma - 1) M_i^2 p \nabla \cdot \mathbf{u}. \tag{2.113}$$

Integrating this equation over a cell volume, and using Gauss' Theorem, results in

$$\oint_S \hat{\mathbf{n}} \cdot (\rho \mathbf{u} T)\, dS = \frac{\gamma}{Pe} \oint_S \hat{\mathbf{n}} \cdot (k \nabla T)\; dS - \gamma (\gamma - 1) M_i^2 \int_V p \nabla \cdot \mathbf{u}\; dV. \tag{2.114}$$

Examining each of the terms in the above equation in a similar manner to the process fol-

lowed for the x-momentum equation, results in the following approximations. The convective term is slightly different than the x-momentum representation

$$\oint_S \hat{\mathbf{n}} \cdot (\rho \mathbf{u} T)\, dS = \oint_S (\rho T)\mathbf{u} \cdot \hat{\mathbf{n}} \approx$$
$$(\rho T)_e u_e A_e - (\rho T)_w u_w A_w +$$
$$(\rho T)_n v_n A_n - (\rho T)_s v_s A_s, \tag{2.115}$$

where

$$(\rho T)_e u_e = (\rho T)_e u_{\text{up}} = \rho_p T_p [\![0, u_e]\!] - \rho_e T_e [\![0, -u_e]\!] = \text{cflxe} \tag{2.116}$$

$$(\rho T)_w u_w = (\rho T)_w u_{\text{up}} = \rho_w T_w [\![0, u_w]\!] - \rho_p T_p [\![0, -u_w]\!] = \text{cflxw} \tag{2.117}$$

$$(\rho T)_n v_n = (\rho T)_n v_{\text{up}} = \rho_p T_p [\![0, v_n]\!] - \rho_n T_n [\![0, -v_n]\!] = \text{cflxn} \tag{2.118}$$

$$(\rho T)_s v_s = (\rho T)_s v_{\text{up}} = \rho_s T_s [\![0, v_s]\!] - \rho_p T_p [\![0, -v_s]\!] = \text{cflxs}. \tag{2.119}$$

The areas of the cells are

$$A_e = A_w = \Delta y_j$$
$$A_n = A_s = \Delta x_i.$$

The gradient term can be expressed in the form

$$\oint_S \hat{\mathbf{n}} \cdot (k \nabla T)\, dS \approx$$
$$k_e \left(\frac{\partial T}{\partial x}\right)_e A_e - k_w \left(\frac{\partial T}{\partial x}\right)_w A_w +$$
$$k_n \left(\frac{\partial T}{\partial y}\right)_n A_n - k_s \left(\frac{\partial T}{\partial y}\right)_s A_s, \tag{2.120}$$

where

$$
\begin{aligned}
k_e \left(\frac{\partial T}{\partial x}\right)_e &= 2k_e \left(\frac{T_e - T_p}{\Delta x_i + \Delta x_{i+1}}\right) = \text{dflxe} \\
k_w \left(\frac{\partial T}{\partial x}\right)_w &= 2k_w \left(\frac{T_p - T_w}{\Delta x_i + \Delta x_{i-1}}\right) = \text{dflxw} \\
k_n \left(\frac{\partial T}{\partial y}\right)_n &= 2k_n \left(\frac{T_n - T_p}{\Delta y_j + \Delta y_{j+1}}\right) = \text{dflxn} \\
k_s \left(\frac{\partial T}{\partial y}\right)_s &= 2k_s \left(\frac{T_p - T_s}{\Delta y_j + \Delta y_{j-1}}\right) = \text{dflxs}.
\end{aligned}
$$

This result (Equation 2.120) is easily verified to be identical to a normal central difference ($\nabla^2 T$) representation when a uniform mesh is used

$$
\Delta x \Delta y \left[\frac{T_e - 2T_p + T_w}{\Delta x^2} + \frac{T_n - 2T_p + T_s}{\Delta y^2}\right]. \tag{2.121}
$$

Finally, the remaining term is

$$
\int_V p \nabla \cdot \mathbf{u} \, dV \approx p_p \left(\nabla \cdot \mathbf{u}\right)_p V_p, \tag{2.122}
$$

where

$$
\left(\nabla \cdot \mathbf{u}\right)_p \approx \frac{u_e - u_w}{\Delta x_i} + \frac{v_n - v_s}{\Delta y_j}, \tag{2.123}
$$

and

$$
V_p = A_{\text{cell}} = \Delta x_i \Delta y_j. \tag{2.124}
$$

With the above approximations, the energy equation assumes the form

$$
\left[(\rho T)_e u_e A_e - (\rho T)_w u_w A_w + (\rho T)_n v_n A_n - (\rho T)_s v_s A_s\right] -
$$

$$\frac{\gamma}{Pe}\left[(\text{dflxe})A_e - (\text{dflxw})A_w + (\text{dflxn})A_n - (\text{dflxs})A_s\right] +$$

$$\gamma(\gamma-1)M_i^2 p_p \left(\nabla \cdot \mathbf{u}\right)_p A_n A_w = 0. \tag{2.125}$$

The State Equation

Expressing the equation of state in discrete form is the final step of the discretization of the governing equations. The form of the state equation was developed earlier (Equation 2.34)

$$p = \frac{\rho T}{\gamma M_i^2}. \tag{2.126}$$

This result may be expressed in discrete form as

$$p_p = \frac{\rho_p T_p}{\gamma M_i^2}. \tag{2.127}$$

In effect, this equation allows the calculation of a given cells pressure based on the density and temperature of the cell. The simplicity of the derivation aside, this equation is necessary to form a well-posed system and enable a solution to the problem.

2.3 Boundary Conditions

In general, a set of PDEs having space and time as independent variables requires both boundary and initial conditions to be well-posed. For the particular case of steady-state flow considered, initial conditions will not be required as the transient terms have been eliminated through the simplification process discussed in Section 2.2.

Boundary conditions are simply requirements placed on the dependent variables (for this study, ρ, u, v, p, T) at the boundary $\partial\Omega$ (see Figure 2.10) of the domain in which the governing

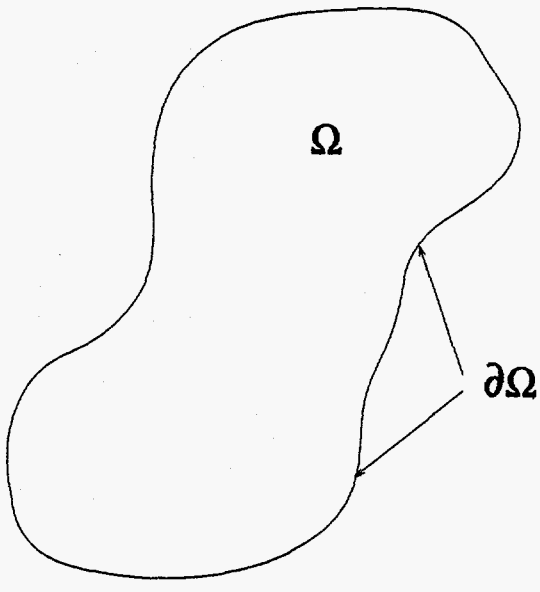

Figure 2.10: The problem domain.

equations are defined (*i.e.*, in Ω). That is, the governing equations describe the physical process in the domain Ω

$$L(u, v, \rho, p, T) = 0 \qquad \mathbf{x} \in \Omega, \tag{2.128}$$

however, they do not apply outside or at the boundary of the domain. The conditions at the boundary must be specified to obtain a well-posed system

$$l(u, v, \rho, p, T) = 0 \qquad \mathbf{x} \in \partial\Omega. \tag{2.129}$$

Boundary conditions may be of three classical forms, *Dirichlet*, *Neumann*, and *mixed*. Dirichlet conditions specify a boundary value as a constant or algebraic function, like the no-slip fluid condition on solid boundaries

$$l(u) = u = 0 \qquad x, y \subseteq \partial\Omega \tag{2.130}$$

$$l(v) = v = 0 \qquad x, y \subseteq \partial\Omega, \tag{2.131}$$

or a parabolic velocity inlet condition

$$l(u) = u = -\frac{1}{2} Re \frac{dp}{dx}(y)(1-y) \qquad x \subseteq \partial\Omega. \tag{2.132}$$

Neumann conditions are derivative conditions that specify a gradient as a constant or algebraic expression on $\partial\Omega$, similar to an adiabatic boundary

$$l(T) = q = -k\frac{\partial T}{\partial x} = 0 \qquad x \subseteq \partial\Omega. \tag{2.133}$$

Mixed conditions are a combination of the above

$$l(T) = k\nabla T \cdot \hat{\mathbf{n}} + h(T - T_r) = 0 \qquad x, y \subseteq \partial\Omega. \tag{2.134}$$

Specification of discrete forms of these conditions is related to the discretization of the governing equations. Consider the application of the no-slip velocity equation for the y-momentum equation along an east boundary (Figure 2.11). In discrete terms, the formulation of the no-slip condition would be

$$v_e = u_{ne} = u_{se} = 0. \tag{2.135}$$

As an example of a Neumann condition, consider the energy equation with an adiabatic east wall (Figure 2.12). To specify $\frac{\partial T}{\partial x} = 0$ along this wall,

$$T_e = T_p. \tag{2.136}$$

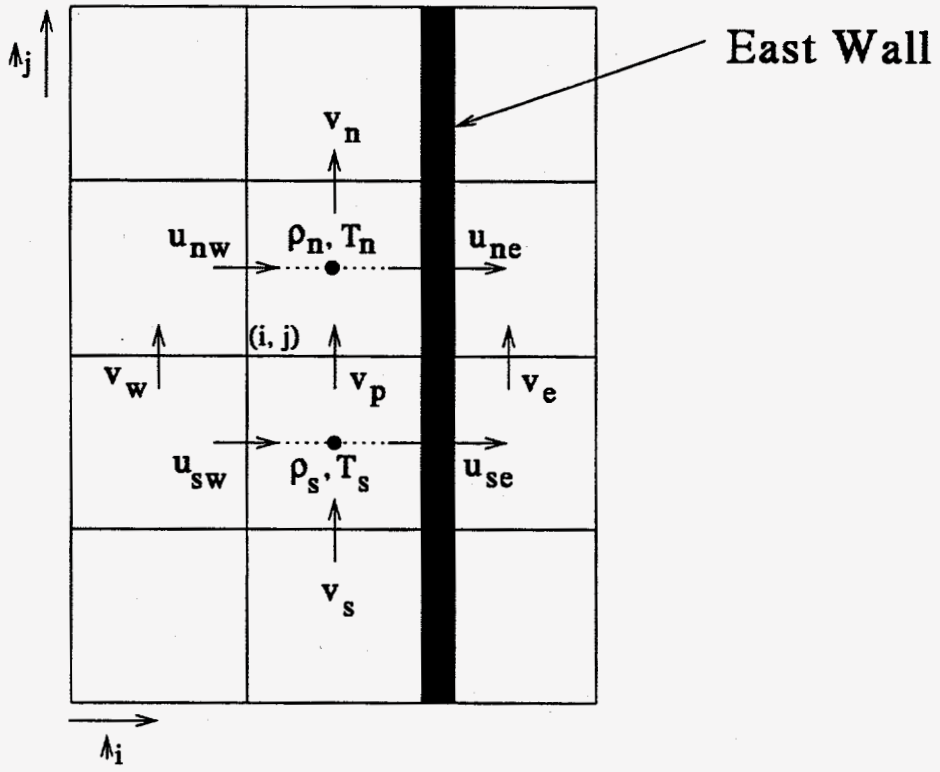

Figure 2.11: No-slip y-momentum condition along an east wall.

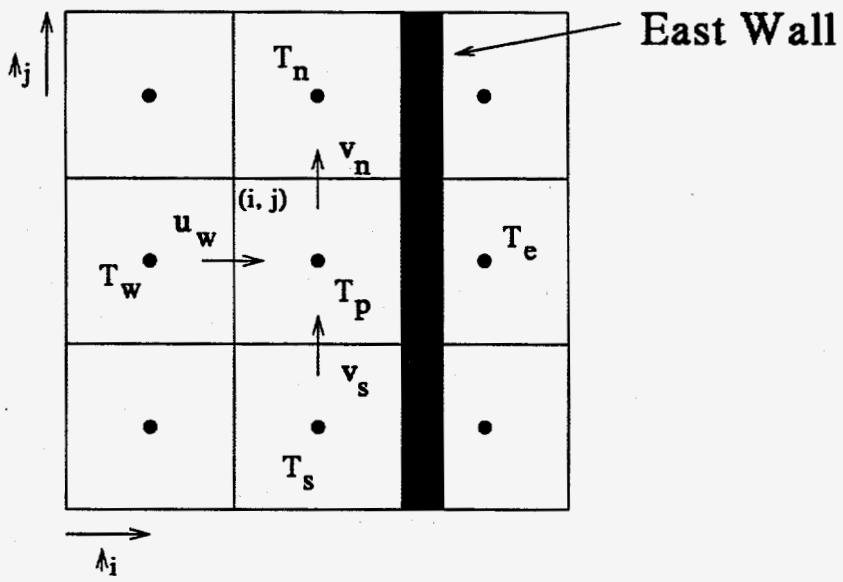

Figure 2.12: Adiabatic temperature condition along an east wall.

Consulting Equation 2.120, setting $T_e = T_p$ results in satisfying the heat flux derivative condition

$$q_e = -k_e \left(\frac{\partial T}{\partial x}\right)_e = 2k_e \left(\frac{T_p - T_e}{\Delta x_i + \Delta x_{i+1}}\right) = 0. \tag{2.137}$$

These concepts may be extended in a similar manner to describe all three types of boundary conditions in discrete form.

2.4 The Non-linear Algebraic System of Equations

The previous development transforms the continuous PDEs of the Navier-Stokes equations into (discrete) algebraic equations that illustrate the relationships between the variables located in each computational cell in the domain of interest ($\Omega \cup \partial\Omega$). In effect, each of the four PDEs, in conjunction with the equation of state, are expressed as five algebraic representations per computational cell. To summarize, let $\mathbf{x}$ be defined as the state vector

$$\mathbf{x} = (x_1, x_2, x_3, x_4, x_5) = (\rho, u, v, p, T), \tag{2.138}$$

in the continuous space of the problem domain. This definition results in the following equation set.

$$\begin{aligned} g_1(\mathbf{x}) &= \nabla \cdot (\rho \mathbf{u}) = 0 \\ f_1(\mathbf{x}) &= (\text{cfluxe} - \text{cfluxw})\Delta y_j + (\text{cfluxn} - \text{cfluxs})\Delta x_i = 0 \end{aligned} \tag{2.139}$$

$$\begin{aligned} g_2(\mathbf{x}) = {} & \frac{\partial \rho u^2}{\partial x} + \frac{\partial \rho uv}{\partial y} + \frac{\partial p}{\partial x} - \frac{1}{Re}\left\{\frac{\partial}{\partial y}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right]\right. \\ & \left. - \frac{\partial}{\partial x}\left[2\mu\frac{\partial u}{\partial x} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right]\right\} - \frac{\rho}{{Fr_x}^2} = 0 \end{aligned}$$

$$
\begin{aligned}
f_2(\mathbf{x}) &= [(\text{cflxe})A_e - (\text{cflxw})A_w + (\text{cflxn})A_n - (\text{cflxs})A_s] + [p_e A_e - p_w A_w] - \\
&\quad \frac{1}{Re}\left[(\tau_x \cdot \hat{i})_e A_e - (\tau_x \cdot \hat{i})_w A_w + (\tau_x \cdot \hat{j})_n A_n - (\tau_x \cdot \hat{j})_s A_s\right] - \\
&\quad \left[\frac{(\rho_e + \rho_w)}{2Fr_x{}^2} A_n A_w\right] = 0 \qquad (2.140)
\end{aligned}
$$

$$
\begin{aligned}
g_3(\mathbf{x}) &= \frac{\partial \rho uv}{\partial x} + \frac{\partial \rho v^2}{\partial y} + \frac{\partial p}{\partial y} - \frac{1}{Re}\left\{\frac{\partial}{\partial x}\left[\mu\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right)\right]\right. \\
&\quad \left. - \frac{\partial}{\partial y}\left[2\mu\frac{\partial v}{\partial y} - \frac{2}{3}\mu\left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right)\right]\right\} - \frac{\rho}{Fr_y{}^2} = 0 \\
f_3(\mathbf{x}) &= [(\text{cflxe})A_e - (\text{cflxw})A_w + (\text{cflxn})A_n - (\text{cflxs})A_s] + [p_n A_n - p_s A_s] - \\
&\quad \frac{1}{Re}\left[(\tau_y \cdot \hat{i})_e A_e - (\tau_y \cdot \hat{i})_w A_w + (\tau_y \cdot \hat{j})_n A_n - (\tau_y \cdot \hat{j})_s A_s\right] - \\
&\quad \left[\frac{(\rho_n + \rho_s)}{2Fr_y{}^2} A_n A_w\right] = 0 \qquad (2.141)
\end{aligned}
$$

$$
\begin{aligned}
g_4(\mathbf{x}) &= \nabla \cdot (\rho \mathbf{u} T) - \frac{\gamma}{Pe}\nabla \cdot (k \nabla T) + \gamma(\gamma - 1) M_i^2 p \nabla \cdot \mathbf{u} = 0 \\
f_4(\mathbf{x}) &= [(\rho T)_e u_e A_e - (\rho T)_w u_w A_w + (\rho T)_n v_n A_n - (\rho T)_s v_s A_s] - \\
&\quad \frac{\gamma}{Pe}[(\text{dflxe})A_e - (\text{dflxw})A_w + (\text{dflxn})A_n - (\text{dflxs})A_s] + \\
&\quad \gamma(\gamma - 1) M_i^2 p_p (\nabla \cdot \mathbf{u})_p A_n A_w = 0 \qquad (2.142)
\end{aligned}
$$

$$
\begin{aligned}
g_5(\mathbf{x}) &= p - \frac{\rho T}{\gamma M_i^2} = 0 \\
f_5(\mathbf{x}) &= p_p - \frac{\rho_p T_p}{\gamma M_i^2} = 0 \qquad (2.143)
\end{aligned}
$$

One immediately recognizes that $g_k(\mathbf{x})$ corresponds to the dimensionless form of the governing equations, with k an index $1 \leq k \leq 5$ signifying the continuity, x-momentum, y-momentum, energy, and state equations, respectively. The second function representation, $f_k(\mathbf{x})$, is the corresponding discrete form of $g_k(\mathbf{x})$.

The discrete equation set for each discrete cell, $f_1(\mathbf{x}), \ldots, f_5(\mathbf{x})$, may be collapsed into a shorthand notation. Clearly, a set of these algebraic equations exists for each and every cell (i,j) in Ω. Let $\mathbf{f}_{(i,j)}$ denote this equation set for the arbitrary cell (i,j)

$$\mathbf{f}_{(i,j)}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x}), f_4(\mathbf{x}), f_5(\mathbf{x})]^T = 0. \tag{2.144}$$

Note that this expression is valid for every cell (i,j) in $\Omega \cup \partial\Omega$, for $0 \leq i \leq I$ and $0 \leq j \leq J$, with the boundaries $\partial\Omega$ corresponding to $i = 0$, $i = I$, $j = 0$, and $j = J$. These cell contributions $\mathbf{f}_{(i,j)}(\mathbf{x})$ may be assembled over the domain $\Omega \cup \partial\Omega$ via the use of an assembly operator S, forming a non-linear algebraic system

$$\mathbf{F}(\mathbf{x}) = \sum_{i=0}^{I} \sum_{j=0}^{J} \mathbf{f}_{(i,j)}(\mathbf{x}) = 0. \tag{2.145}$$

To summarize, use of the assembly operator S over the two-dimensional domain $\Omega \cup \partial\Omega$ on the cell contribution equations (Equation 2.144) results in the non-linear algebraic system

$$\mathbf{F}(\mathbf{x}) = 0. \tag{2.146}$$

This result completes the derivation of the discrete system to be solved. The system has been cast into an implicit form to enable a simultaneous, fully-coupled solution. This solution is typically achieved in two steps.

1. An iterative linearization operation that results in a linear system to be solved each iteration.

2. Solution of the "inner" linear system. This may be accomplished directly, or via the

use of an iterative algorithm.

For this study, an inexact Newton technique will be employed as the linearization operator, with an iterative preconditioned Krylov technique used to solve the resulting linear system. This results in an outer Newton iteration loop for linearization, and an inner Krylov loop for the iterative solution of the linear system resulting from the Newton technique. This process will be explained in detail in the following chapter.

Chapter 3

Solution of the Non-linear Algebraic System

In the previous section, the discretized form of the governing equations were shown to assume the form of a non-linear system

$$\mathbf{F}(\mathbf{x}) = [f_1(\mathbf{x}), f_2(\mathbf{x}), f_3(\mathbf{x}), \ldots, f_N(\mathbf{x})]^T = 0, \tag{3.1}$$

with the state vector $\mathbf{x}$ expressed as

$$\mathbf{x} = [x_1, x_2, x_3, \ldots, x_N]^T . \tag{3.2}$$

To obtain a solution of this non-linear system, it is important to examine it's characteristics. The system

$$\mathbf{F}(\mathbf{x}) = 0, \tag{3.3}$$

may be defined as the non-linear mapping $\mathbf{F} : \mathcal{R}^n \rightarrow \mathcal{R}^n$ with the properties:

1. $\exists\ x^* \in \mathcal{R}^n$ with $\mathbf{F}(\mathbf{x}^*) = 0$,

2. $\mathbf{F}$ is continuously differentiable in the neighborhood of $\mathbf{x}^*$ (it is generally sufficient that the *Jacobian* exists and is continuous at $\mathbf{x}^*$), and

3. $\mathbf{F}'(\mathbf{x}^*)$ is non-singular.

Given an $\mathbf{F}(\mathbf{x}) = 0$ that meets this criterion, Newton's method is attractive as it converges quite rapidly given an appropriate initial estimate $\mathbf{x}_o$ [46]. In fact, Newton's method is the standard used to compare rapidly convergent methods for solving the non-linear system (Equation 3.1); a way of characterizing superlinear convergence is that each convergence step should asymptotically approach the Newton step in both magnitude and direction [55].

3.1 Newton's Method

Newton's method is an iterative technique used to linearize the algebraic system of equations. To obtain an approximation (x_o^1) of the root $f(x_o) = 0$ using Newton's method [56]

$$x_o^1 = x_o^o - \frac{f(x_o^o)}{f'(x_o^o)}, \tag{3.4}$$

or

$$f'(x_o^o)(x_o^1 - x_o^o) = -f(x_o^o), \tag{3.5}$$

or

$$f'(x_o^o)\Delta x_o^o = -f(x_o^o). \tag{3.6}$$

Given a system of N equations, Newton's method may be expressed as

$$\begin{bmatrix} \frac{\partial f_1^n}{\partial x_1^n} & \frac{\partial f_1^n}{\partial x_2^n} & \cdots & \frac{\partial f_1^n}{\partial x_N^n} \\ \frac{\partial f_2^n}{\partial x_1^n} & \frac{\partial f_2^n}{\partial x_2^n} & \cdots & \frac{\partial f_2^n}{\partial x_N^n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_N^n}{\partial x_1^n} & \frac{\partial f_N^n}{\partial x_2^n} & \cdots & \frac{\partial f_N^n}{\partial x_N^n} \end{bmatrix} \begin{Bmatrix} \Delta x_1^n \\ \Delta x_2^n \\ \vdots \\ \Delta x_N^n \end{Bmatrix} = - \begin{Bmatrix} f(x_1^n) \\ f(x_2^n) \\ \vdots \\ f(x_N^n) \end{Bmatrix}. \tag{3.7}$$

With the application of this method using consistent notation, the resultant linear system is obtained

$$\mathbf{J}^n \delta \mathbf{x}^n = -\mathbf{F}(\mathbf{x}^n). \tag{3.8}$$

The elements of the *Jacobian* ($\mathbf{J}$) for this system can be defined as

$$\mathbf{J}_{ij}^n \equiv \frac{\partial f_i^n}{\partial x_j^n}, \tag{3.9}$$

$\delta \mathbf{x}$ is the Newton iteration update vector, and the n superscript ($\{\cdot\}^n$) refers to the Newton iteration number. The Jacobian for this system is calculated numerically [8]. The new solution approximation in the Newton step is obtained by

$$\mathbf{x}^{n+1} = \mathbf{x}^n + d\delta \mathbf{x}^n, \tag{3.10}$$

where the constant d ($0 < d \leq 1$) is used to damp the Newton updates. Selection of this constant employs a strategy to prevent the Newton update from being driven into the non-physical domain (*i.e.*, the calculation of negative temperatures), and to scale large variable updates when the trial solution is far from the correct solution. This is accomplished by setting the parameter d based on the ratio of the thermodynamic variables and the Newton

update vector on the domain

$$d = \min\left[1, \min\left(\frac{\alpha \mathbf{x}}{\delta \mathbf{x}}\right)\right], \tag{3.11}$$

where α is a user specified damping value. The Newton iteration process continues until the trial solution is "sufficiently close" to the actual solution. In this context, "sufficiently close" is the point where the norm of the difference between the approximate solution and the exact solution is below a suitable tolerance level η

$$\max\left[\frac{|\delta \mathbf{x}|}{\max(1, |\mathbf{x}|)}\right] < \eta. \tag{3.12}$$

Using Equation 3.9, the Jacobian matrix may be readily calculated from the non-linear algebraic system (Equation 3.1). The Jacobian has a sparse, banded, pentadiagonal structure (Figure 3.1), and is generally non-symmetric and indefinite.

This discussion has lightly touched on the fact that the linear system $\mathbf{J}^n \delta \mathbf{x}^n = -\mathbf{F}(\mathbf{x}^n)$ must be solved to obtain $\delta \mathbf{x}^n$ at each Newton iteration. A direct solve, such as Gaussian elimination, could be used to find the solution. However, for most applications of interest in this area (especially the large two-dimensional problems of interest), such a technique would be prohibitively CPU and memory intensive. At this point, some observations may be made which are directed at overcoming this dilemma.

- Is an exact direct solution of the linear system really necessary when the Newton trial solution is far from the correct solution? Would a less intensive, less accurate, but more efficient method suffice?

- Given a more efficient method, could it's accuracy be improved as the Newton trial solution approaches the actual solution, thereby completely eliminating the need to

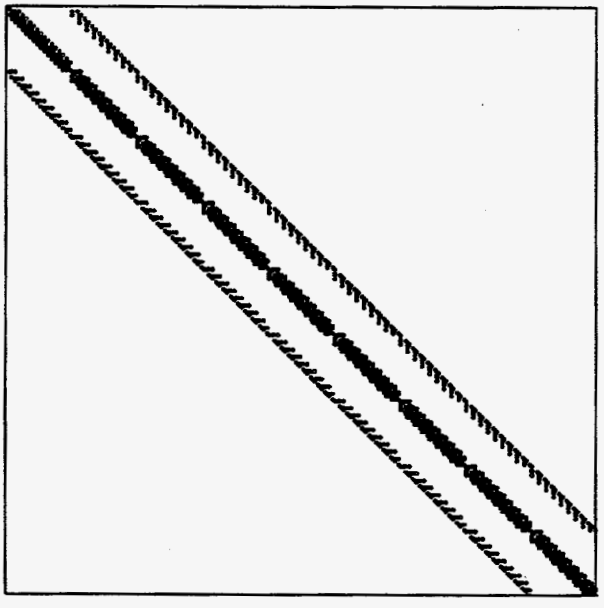

Figure 3.1: The structure of the Jacobian matrix.

perform a direct solve of the linear system?

The inexact Newton's method coupled with a Krylov-based iterative linear solution technique does indeed allow the accuracy of the solution to increase as the solution approaches convergence, reducing the work required when the solution is far from convergence. Additionally, Krylov methods are sufficiently accurate to completely dispense with a direct solution method in most cases.

3.2 The Inexact Newton's Method

The inexact Newton's method was developed to decrease the computational requirements of the linear solution for Newton iterations far from the solution of the system. This technique capitalizes on the behavior of iterative linear solution techniques. A desirable iterative linear

system solution scheme requires I iterations to converge on a solution to the linear system. As the number of iterations of the technique approach I, the approximate linear solution progressively approaches the true solution of the linear system. If the iteration procedure is interrupted, say at iteration i, where $0 < i < I$, the current result lies somewhere on the path from the initial "guess" ($i = 0$) to the true solution ($i = I$). This approximate result at i may be sufficiently accurate to allow progress toward the solution in the next enclosing Newton iteration.

Initially, this process appears to shift some of the work in achieving a solution from the solution of the linear system to the Newton iteration process. If the linear system iteration is interrupted too early, this may certainly occur. In fact, the linear system update to the Newton iteration may be of such low quality that the Newton iteration is shifted further from convergence than the previous iteration. However, if the linear system iteration is interrupted near convergence, the solution is often "sufficiently close" to a converged solution that the enclosing Newton iteration is negligibly affected. In many cases, the last few linear iterations, while significantly increasing runtime of the solution, do not contribute much to the overall solution efficiency. Experience has shown that a linear solution tolerance based on the norm of $\delta\mathbf{x}$ and the norm of $\mathbf{F}(\mathbf{x})$ is effective [9]

$$\frac{\| \mathbf{J}^n \delta\mathbf{x}^n + \mathbf{F}(\mathbf{x}^n) \|}{\| \mathbf{F}(\mathbf{x}^n) \|} < \epsilon. \tag{3.13}$$

The above expression indicates that the inner iteration process is truncated when the norm of the solution is less than the norm of the residual ($\mathbf{F}(\mathbf{x}^n)$) multiplied by a tolerance criterion (ϵ). As the Newton iteration process approaches convergence, the residual approaches zero, tightening the tolerance on the linear solution process. In effect, the linear solution

process "automatically" becomes more accurate as the Newton solution process approaches convergence.

In summary, the inexact Newton method relaxes the tolerance on the linear iterative solution process when the Newton iterate is far from convergence. This tolerance parameter is automatically tightened as the Newton process approaches convergence, thus reducing unnecessary linear solution iterations and increasing computational efficiency.

3.3 Preconditioning

The inexact Newton method described previously is a very powerful technique. The Newton iteration process has the potential to achieve quadratic (superlinear) convergence, the iterative linear solution technique is more efficient (and requires less memory) than a direct inversion method, and the dynamic tolerance facility increases the computational efficiency of the method. However, this method is based on the solution of the linear system

$$\mathbf{J}^n \delta\mathbf{x}^n = -\mathbf{F}(\mathbf{x}^n). \tag{3.14}$$

The speed of convergence of an algorithm depends on the condition number, $\kappa_2(\mathbf{J})$, of the Jacobian matrix $\mathbf{J}$ and the distribution of the eigenvalues of $\mathbf{J}$. $\kappa_2(\mathbf{J})$ is the ratio of the maximum to minimum eigenvalues of $\mathbf{J}$. If $\kappa_2(\mathbf{J})$ is large or the spectrum of the eigenvalues of $\mathbf{J}$ is scattered and wide, $\mathbf{J}$ is called *poorly conditioned* and the convergence rate may be quite slow. In fact, is not difficult to encounter problems where the solution technique will not yield convergence [43]. As this study is directed at large, challenging solutions, it is mandatory that the techniques employed work on a wide variety of difficult problems.

Preconditioning is a technique used to improve the condition number of the Jacobian

matrix. Ideally, the goal of preconditioning is to "force" the Jacobian towards the behavior of the identity matrix (**I**). This may be facilitated by multiplying both sides of the linear system by a preconditioning matrix (in this case, the inverse of the preconditioning matrix $\mathbf{P}_l$).

$$\mathbf{P}_l^{-1}\mathbf{J}^n\delta\mathbf{x}^n = -\mathbf{P}_l^{-1}\mathbf{F}(\mathbf{x}^n) \tag{3.15}$$

Equation 3.15 is called the "left preconditioned form" of the linear system. It is also possible to use right preconditioning.

$$\mathbf{J}^n\mathbf{P}_r^{-1}\mathbf{P}_r\delta\mathbf{x}^n = -\mathbf{F}(\mathbf{x}^n) \tag{3.16}$$

In the remainder of this study, left preconditioning is assumed unless specified otherwise.

Ideally, one would desire the product $\mathbf{P}_l^{-1}\mathbf{J}^n$ to form the identity matrix. In this case, it is obvious that once one computes $-\mathbf{P}_l^{-1}\mathbf{F}(\mathbf{x}^n)$, the solution $\delta\mathbf{x}^n$ is immediately known. This result, $\mathbf{P}_l^{-1}\mathbf{J}^n = \mathbf{I}$, occurs when $\mathbf{P}_l = \mathbf{J}$. However, inverting $\mathbf{P}_l$ requires an appreciable amount of work (as large as $O(n^3)$ operations for dense Gaussian elimination). Effective preconditioning requires that the preconditioner reasonably approximate **J** and that systems of the form $\mathbf{P}_l\mathbf{v} = \mathbf{b}$, which arise within the linear solution iteration, can be solved efficiently. Thus, it is reasonable to begin with $\mathbf{P}_l = \mathbf{J}$, and use an approximate technique to invert $\mathbf{P}_l$.

A popular class of preconditioners is based upon incomplete factorizations (ILU) of the Jacobian matrix [47]. However, ILU preconditioners often do not scale well with problem size [48], and are difficult to parallelize. As an attempt to overcome these difficulties, this study will examine domain-based preconditioning such as the additive and multiplicative Schwarz algorithms [48, 57].

Domain-based preconditioning is a method of partitioning the global Jacobian matrix to form a global preconditioner. Figure 3.2 depicts a simplified representation of the pentadi-

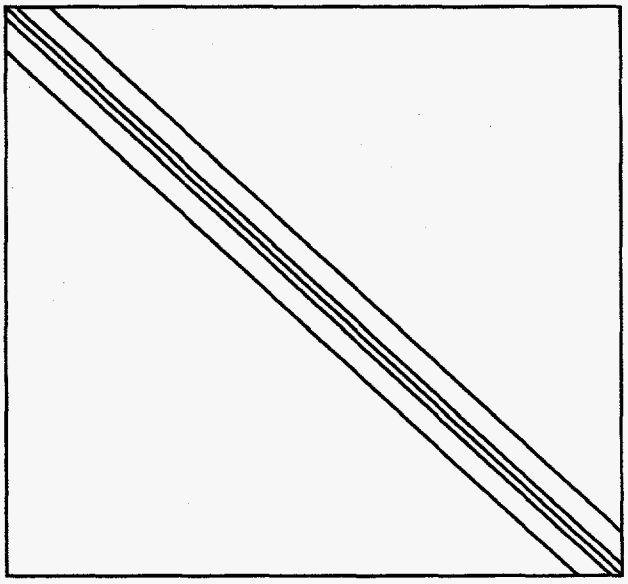

Figure 3.2: Simplified Jacobian matrix.

agonal Jacobian matrix. Domain-based preconditioning considers the sparsity of the matrix to partition it into subdomains, as shown in Figure 3.3. For sparse, diagonally-dominant systems, this method captures the majority of the Jacobian data in subdomains that occur along the main diagonal of the Jacobian matrix. Subdomains constructed in this manner are essentially independent and may be inverted individually and re-assembled back into a global preconditioner.

To better understand this mapping from the global Jacobian matrix to the subdomains, examine Figure 3.3. This figure shows the Jacobian matrix partitioned into four subdomains. The data for subdomains 1 and 3 are obtained from the upper-left rectangular quadrant of the Jacobian, while the data for subdomains 2 and 4 are obtained from the lower right quadrant. Figure 3.4 shows a magnified image of the upper-right quadrant (subdomains 1 and 3). This quadrant is divided into horizontal rectangular strips; subdomain 3 consists of the data falling in the shaded regions, while subdomain 1 consists of the data in the remaining

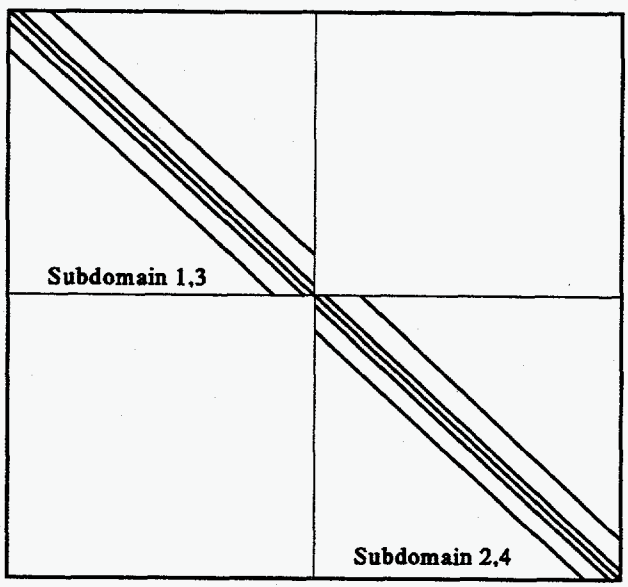

Figure 3.3: Partitioned Jacobian matrix, four subdomains.

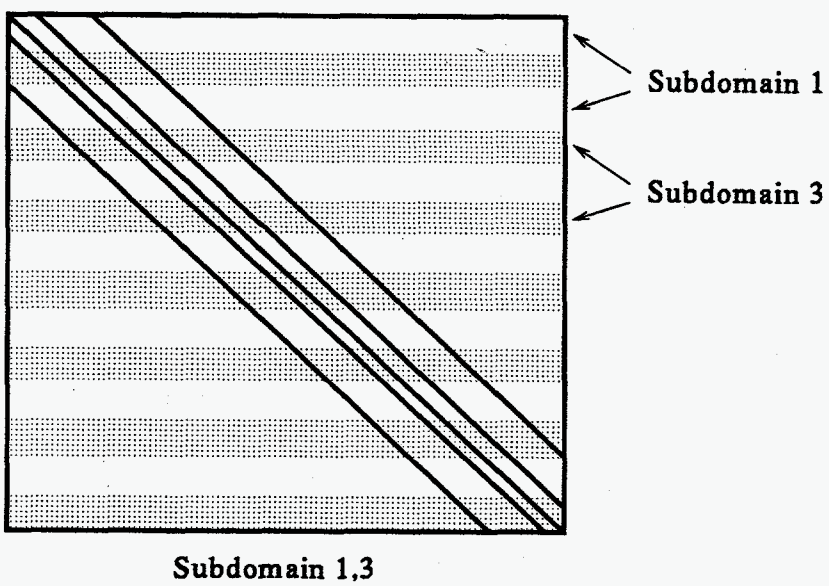

Figure 3.4: Magnified subdomain.

regions. Clearly, the data for the subdomains 1,3 and 2,4 are obtained from distinct, non-overlapping regions of the Jacobian. Additionally, in each quadrant, the data composing the individual subdomains originate from non-overlapping regions. These figures reflect the decomposition of the Jacobian matrix based on a four block physical decomposition of the problem in a two-dimensional grid (with two blocks in the x extent and two blocks in the y). A stripwise decomposition (with one block in the x extent and four blocks in the y extent) results in a different matrix decomposition (similar to that shown in Figure 3.4 extended over the entire Jacobian matrix). In this case, there are no quadrants, and the strips alternate in the subdomain sequence $[1, 2, 3, 4, 1, \ldots]$.

3.3.1 Additive Schwarz Preconditioning

The additive Schwarz method specifies the global preconditioner in terms of adding the preconditioner subdomains to obtain the preconditioner

$$\mathbf{P}_l^{-1} = \mathbf{J}_1^{-1} + \cdots + \mathbf{J}_p^{-1} = \sum_{i=1}^{p} \mathbf{J}_i^{-1}, \tag{3.17}$$

where $\mathbf{J}_i^{-1}$ is the inverse of the subdomains of the Jacobian shown in Figures 3.3 and 3.4. Four subdomains are shown in the figures, but p subdomains are possible (Figure 3.5 shows 16 subdomains). Again, in Figure 3.5, each of the four rectangular regions shown is further subdivided into a four subdomain strip sequence similar to that shown in Figure 3.4. For example, the rectangular region corresponding to subdomains 1,5,9,13 would consist of the alternating strip sequence $[1, 5, 9, 13, 1, \ldots]$.

It is interesting to note the effects of increasing the number of subdomains on the amount of data captured from the Jacobian for use in the preconditioner. Only the data on the

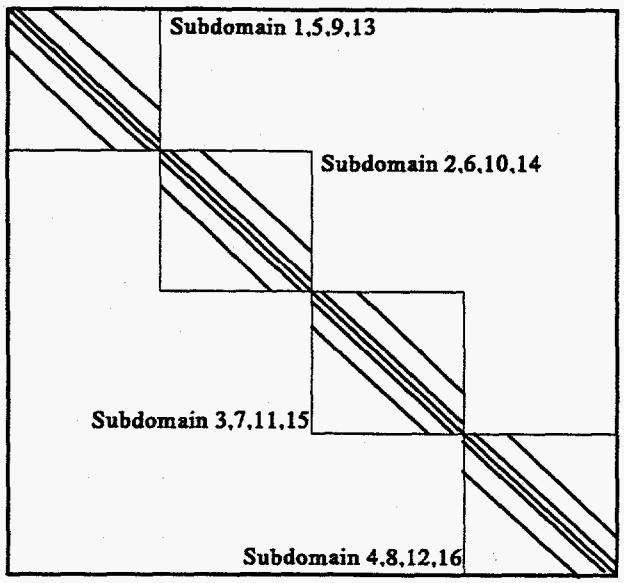

Figure 3.5: Partitioned Jacobian matrix, 16 subdomains.

subdomain interior is inverted for use in the preconditioner (in the figures, the data outside the subdomains was deliberately excluded from the diagrams to better visualize the data that constitutes the preconditioner). The data on the exterior of the subdomain is effectively discarded and does not contribute to the preconditioner. It is clear that more data is discarded in the 16 subdomain case than in the 4 subdomain case. As the number of subdomains is increased, less data from the Jacobian is inverted to form the preconditioner. Thus, the expected quality of the preconditioner degrades as the number of subblocks is increased.

To mitigate this degradation, subdomain overlap may be employed. For example, in Figure 3.6, all the Jacobian data is now part of the subdomains and will be inverted to form the preconditioner. However, increasing the amount of overlap increases the size of each of the subdomains, which in turn increases the number of operations required to invert each subdomain. In Figure 3.6, the regions corresponding to subdomains $(1,3)$ and $(2,4)$ overlap at the subdomain boundaries. Additionally, the rectangular strips in Figure 3.4 also overlap

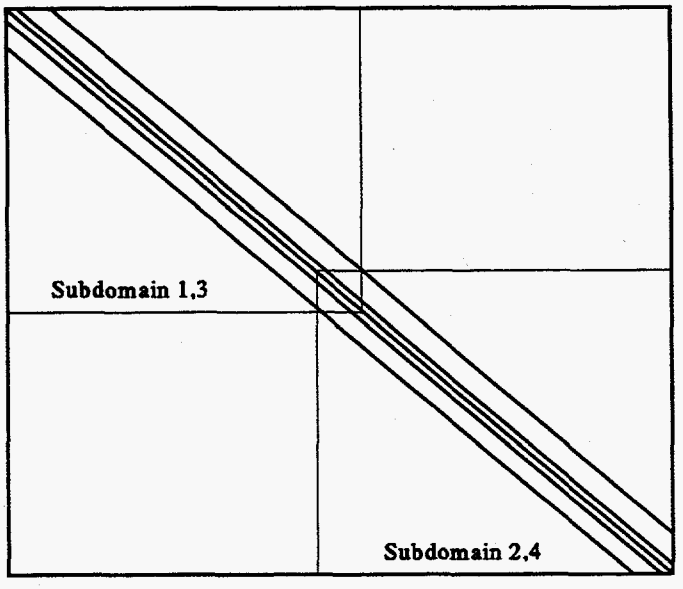

Figure 3.6: Partitioned Jacobian matrix, four subdomains with overlap.

the subdomains above and below. This is illustrated in Figure 3.7.

The additive Schwarz preconditioning technique (in common with the multiplicative technique) requires inversion of each of the preconditioner subdomains to form the inverse of the preconditioner. In this study, an exact direct technique (LINPACK banded Gaussian elimination) is used to perform this task. It is also possible to implement an inexact method (such as ILU(0)). The use of an approximate technique will generally decrease the memory requirements of the solution technique and could (depending on the method chosen) decrease the overall solution time. However, an inexact technique may decrease the quality of the overall preconditioner and negatively affect the scalability of the solution algorithm. For this reason, inexact subdomain inversion methods were not examined in this study.

Domain-based preconditioning has an algorithmic efficiency advantage beyond any parallelism concerns. Brute force inversion of the complete Jacobian (a non-domain technique) using a direct method requires $O(n^3)$ operations (the matrix is $n \times n$). With p subdomains,

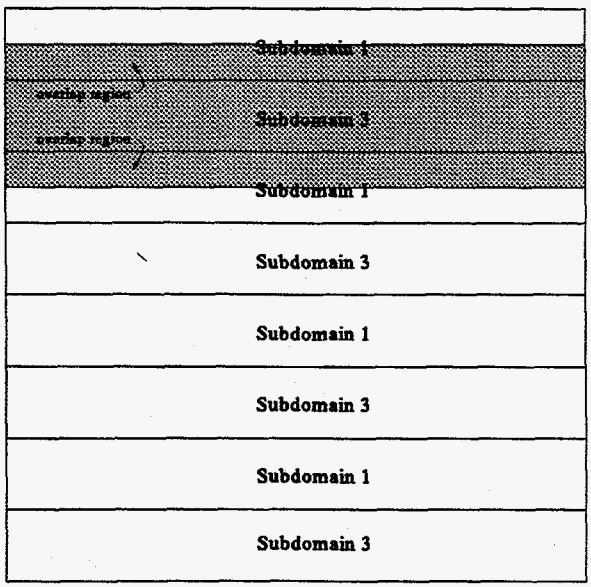

Figure 3.7: Overlap of Subdomain 3.

the effort required to invert each subdomain is $O\left(\frac{n^3}{p^3}\right)$ operations. If the subdomains were inverted serially, $p \cdot O\left(\frac{n^3}{p^3}\right)$ or $O\left(\frac{n^3}{p^2}\right)$ total operations would be required. All else being equal, the subdomain scheme would provide a speedup of p^2 due to algorithmic efficiency alone.

$$S = \frac{\text{full inversion}}{\text{blocked inversion}} = \frac{O\left(n^3\right)}{O\left(\frac{n^3}{p^2}\right)} = O\left(p^2\right) \tag{3.18}$$

The additive Schwarz method is of central interest to the present research because it allows concurrent formation of the preconditioner. The subdomain inversion and assembly processes are completely independent from a subdomain perspective, and may be performed on p processors, where p is the number of subdomains. Subdomain overlap, while increasing the quality of the preconditioner, reduces the portion of the preconditioner formation that may be performed in parallel. This follows from the fact that the assembly operation that forms the global preconditioner requires serialization when the overlapped region is updated.

Additive Schwarz preconditioning without subdomain overlap is analogous to block Jacobi preconditioning. To best visualize this equivalence and to better understand the function of additive Schwarz preconditioning, consider the following block-tridiagonal system

$$\begin{bmatrix} D_1 & U_2 & 0 \\ L_1 & D_2 & U_3 \\ 0 & L_2 & D_3 \end{bmatrix} \begin{Bmatrix} X_1 \\ X_2 \\ X_3 \end{Bmatrix} = \begin{Bmatrix} Y_1 \\ Y_2 \\ Y_3 \end{Bmatrix}. \tag{3.19}$$

A solution of this system based on additive Schwarz preconditioning with no overlap begins with an initial "guess"

$$X_1^0 = X_2^0 = X_3^0 = 0, \tag{3.20}$$

followed by an iterative procedure using the following algorithm.

$$\begin{aligned} X_1^{i+1} &= D_1^{-1}(Y_1 - 0 - U_2 X_2^i) \\ X_2^{i+1} &= D_2^{-1}(Y_2 - L_1 X_1^i - U_3 X_3^i) \\ X_3^{i+1} &= D_3^{-1}(Y_3 - L_2 X_2^i - 0) \end{aligned} \tag{3.21}$$

To compare this result with Equation 3.17 (without overlap), $D_j^{-1} = \mathbf{J}_j^{-1}$. This scheme is clearly identical to a block Jacobi technique. Additionally, the parallel nature of the algorithm is apparent, each of the expressions in Equation 3.21 is clearly independent of the others, and the results are independent of evaluation order.

3.3.2 Multiplicative Schwarz Preconditioning

The multiplicative Schwarz method is similar to the additive Schwarz technique. The blocking strategy and degradation of the preconditioner are identical. However, the preconditioner is

formed by using the multiplicative algorithm

$$\mathbf{P}_l^{-1}\mathbf{J}^n = \mathbf{I} - \left(\mathbf{I} - \mathbf{J}_1^{-1}\mathbf{J}^n\right)\left(\mathbf{I} - \mathbf{J}_2^{-1}\mathbf{J}^n\right)\cdots\left(\mathbf{I} - \mathbf{J}_p^{-1}\mathbf{J}^n\right) = \mathbf{I} - \prod_{i=1}^{p}(\mathbf{I} - \mathbf{J}_i^{-1}\mathbf{J}^n). \qquad (3.22)$$

Understanding the behavior of this algorithm requires some background on the Krylov iterative linear solution algorithm. The Krylov techniques used herein do not explicitly require the formation of the matrix-matrix product $\mathbf{P}_l^{-1}\mathbf{J}^n$. In fact, to solve the linear system using these methods only requires the formation of matrix-vector products of the form $\mathbf{P}_l^{-1}\mathbf{v}$, where $\mathbf{v}$ is a vector iterate produced within the Krylov solution procedure (see Section 3.6). This product results in the vector $\mathbf{w} = \mathbf{P}_l^{-1}\mathbf{v}$ and may be computed by

$$\begin{aligned} \mathbf{v}_1 &= \mathbf{J}_1^{-1}\mathbf{v} \\ \mathbf{v}_j &= \mathbf{v}_{j-1} + \mathbf{J}_j^{-1}(\mathbf{v} - \mathbf{J}\mathbf{v}_{j-1}), \qquad \text{for } j = 2, \ldots, n \\ \mathbf{w} &= \mathbf{v}_n, \end{aligned} \qquad (3.23)$$

where n corresponds to the total number of subdomains employed in the preconditioner. Clearly, this is no longer a parallel algorithm. Multiplicative Schwarz without subdomain overlap is analogous to block Gauss-Seidel preconditioning. Again, consider the block-tridiagonal system of Equation 3.19 with the initial "guess" of

$$X_1^0 = X_2^0 = X_3^0 = 0, \qquad (3.24)$$

followed by an iterative procedure using the algorithm

$$\begin{aligned} X_1^{i+1} &= D_1^{-1}(Y_1 - 0 - U_2 X_2^i) \\ X_2^{i+1} &= D_2^{-1}(Y_2 - L_1 X_1^{i+1} - U_3 X_3^i) \\ X_3^{i+1} &= D_3^{-1}(Y_3 - L_2 X_2^{i+1} - 0). \end{aligned} \tag{3.25}$$

Again, the above result correlates with Equation 3.22 when $D_j^{-1} = \mathbf{J}_j^{-1}$.

In Equation 3.25, the dependency of the equation for X_2^{i+1} on X_1^{i+1} found in the previous equation (a similar dependency exists between X_3^{i+1} and X_2^{i+1}) makes this iterative process sequential. However, the algorithm may yield a more efficient solution due to the the use of recently computed values in the serial sweep. It is possible to achieve a level of parallelism in this algorithm with the use of a subdomain renumbering operation

$$Color(w_i) = \min\{k > 0 \mid k \neq Color(w_j), \forall w_j \in \mathrm{Adj}(w_i)\}. \tag{3.26}$$

A blocking strategy is really nothing more than a subdomain numbering technique. Referring to the previous figures explaining the subdomain relationships with the Jacobian data, these subdomains were numbered for convenience. Given a particular problem, it may not always be desirable to perform computation on the subdomains in serial order. In fact, it may be advantageous to handle the subdomains by skipping every other one, forward, backward, or inside out. The blocking strategy is an abstraction that allows the "naming" of each of the subdomains. For example, with a four subdomain problem, it may be expedient to number the first subdomain with a "1", the second with a "2", and so on. For another problem, better results may be achieved by numbering the subdomains in reverse order. Block (subdomain) numbering becomes quite important when using coloring techniques. To best explain

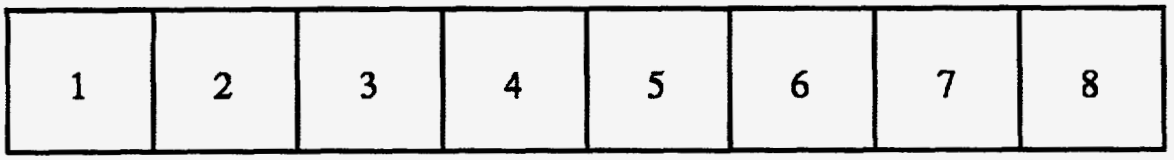

Figure 3.8: Normal block numbering.

coloring techniques, consider the single-dimensional blocking strategy shown in Figure 3.8 (in this case, each block corresponds to a distinct subdomain, "1" to subdomain 1, "2" to subdomain 2, and so on). This numbering results in the matrix form seen in the previous examples and shown below.

$$\begin{bmatrix} D_1 & U_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ L_1 & D_2 & U_3 & 0 & 0 & 0 & 0 & 0 \\ 0 & L_2 & D_3 & U_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & L_3 & D_4 & U_5 & 0 & 0 & 0 \\ 0 & 0 & 0 & L_4 & D_5 & U_6 & 0 & 0 \\ 0 & 0 & 0 & 0 & L_5 & D_6 & U_7 & 0 \\ 0 & 0 & 0 & 0 & 0 & L_6 & D_7 & U_8 \\ 0 & 0 & 0 & 0 & 0 & 0 & L_7 & D_8 \end{bmatrix} \tag{3.27}$$

The structure of this matrix is easily derived. In this expression, D represents a diagonal matrix block, U and L represent blocks located above and below the diagonal, respectively. The subscript (i in D_i) denotes the origin of the matrix block from the numbering scheme (the i^{th} numbered data is inserted into the i^{th} column). This simple representation assumes a physical stripwise decomposition and assumes the discrete stencil incorporates only local block data. For example, consider the block numbered "5" (D_5) above. The discrete stencil

Red	Black	Red	Black	Red	Black	Red	Black
1	5	2	6	3	7	4	8

Figure 3.9: Renumbered blocking.

only requires data from block "4" (L_4) to the left, and block "6" (U_6) to the right, to fully construct the coefficient matrix contribution for block "5."

Figure 3.9 illustrates a renumbered blocking scheme based on Equation 3.26. This renumbering operation vastly changes the structure of the matrix.

$$\left[\begin{array}{cccc|cccc} D_1 & 0 & 0 & 0 & U_{15} & 0 & 0 & 0 \\ 0 & D_2 & 0 & 0 & U_{25} & U_{26} & 0 & 0 \\ 0 & 0 & D_3 & 0 & 0 & U_{36} & U_{37} & 0 \\ 0 & 0 & 0 & D_4 & 0 & 0 & U_{47} & U_{48} \\ \hline L_{51} & L_{52} & 0 & 0 & D_5 & 0 & 0 & 0 \\ 0 & L_{62} & L_{63} & 0 & 0 & D_6 & 0 & 0 \\ 0 & 0 & L_{73} & L_{74} & 0 & 0 & D_7 & 0 \\ 0 & 0 & 0 & L_{84} & 0 & 0 & 0 & D_8 \end{array}\right] \tag{3.28}$$

As an example of this new structure, consider the new domain blocking shown in Figure 3.9. Block "5" (D_5) is now bordered on the left by block "1" (L_{51} in the fifth row, first column) and on the right by block "2" (L_{52} in the the fifth row, second column).

From Figure 3.9 and Equation 3.28, it can be seen that the blocks colored "Red" are placed in the upper half of the new matrix and the "Black" blocks placed in the lower half of the

matrix. This renumbering operation allows all the "Red" blocks to be inverted independently

$$\begin{aligned} X_1^{i+1} &= D_1^{-1}(Y_1 - 0 - U_{15}X_5^i) \\ X_2^{i+1} &= D_2^{-1}(Y_2 - U_{25}X_5^i - U_{26}X_6^i) \\ X_3^{i+1} &= D_3^{-1}(Y_3 - U_{36}X_6^i - U_{37}X_7^i) \\ X_4^{i+1} &= D_4^{-1}(Y_4 - U_{47}X_7^i - U_{48}X_8^i), \end{aligned} \tag{3.29}$$

followed by an inversion of the "Black" blocks

$$\begin{aligned} X_5^{i+1} &= D_5^{-1}(Y_5 - L_{51}X_1^{i+1} - L_{52}X_2^{i+1}) \\ X_6^{i+1} &= D_6^{-1}(Y_6 - L_{62}X_2^{i+1} - L_{63}X_3^{i+1}) \\ X_7^{i+1} &= D_7^{-1}(Y_7 - L_{73}X_3^{i+1} - L_{74}X_4^{i+1}) \\ X_8^{i+1} &= D_8^{-1}(Y_8 - 0 - L_{84}X_4^{i+1}). \end{aligned} \tag{3.30}$$

In this example, the coloring scheme allows the inversion of the four "Red" blocks in parallel, followed by a synchronization step, and then an inversion of the four "Black" blocks. This is clearly an improvement over the purely serial algorithm discussed earlier, but less desirable that the complete independence available with additive Schwarz. Given eight blocks, additive Schwarz allows the use of eight processors concurrently. Eight blocks using colored multiplicative Schwarz allows the concurrent use of only four processors, providing a degree of parallelism (DOP) half that of additive Schwarz

$$\text{DOP}_{\text{multiplicative Schwarz}} = \frac{\text{number of blocks}}{\text{number of colors}}. \tag{3.31}$$

The above results assume a stripwise decomposition of the solution domain.

It is possible to employ a "checkerboard" style of colored domain decomposition in the

Green	Blue	Green	Blue
Red	Black	Red	Black
Green	Blue	Green	Blue
Red	Black	Red	Black

Figure 3.10: "Checkerboard" domain decomposition.

preconditioner (Figure 3.10). With the four colors shown in the figure ("Red," "Black," "Green," and "Blue"), the DOP is one-fourth the additive Schwarz result. Due to the low DOP, this scheme will likely provide lower parallel efficiencies than the additive Schwarz algorithm. However, there may be cases where the checkerboard scheme provides a preconditioner that maps to a given problem better than that provided by additive Schwarz, in effect recouping some of the lost efficiency.

3.3.3 Preconditioning of the Model Problem

In this study, it has been suggested that Schwarz preconditioning may have certain advantages over the popular incomplete LU factorization methods (ILU) for the formation of a preconditioner.

- Schwarz methods (particularly additive Schwarz) contain inherent parallelism. It is possible to parallelize ILU methods. However, the need for pivoting to remove diagonal

zeros makes parallelization of these techniques difficult and limits the DOP that can be achieved. It may also be difficult to obtain sufficient granularity in the parallel ILU methods to provide efficient execution on contemporary shared-memory hardware. The large subblock granularity of the Schwarz techniques is readily apparent.

- ILU methods may not be sufficiently robust to provide reliable solutions to the model problem.

However, because solutions to large model problems of the configuration described in Chapter 2 are apt to be memory constrained, there is the added requirement that the selected preconditioning method be competitive in memory requirements with ILU techniques. An increase in memory required may be tolerable if the added robustness is significant, however, large increases in required memory over the ILU techniques (say an order of magnitude) are generally unacceptable. To examine the memory requirements of the methods, consider a study of serial Schwarz and ILU techniques with various levels of fill-in (ILU(k)) as performed by McHugh [1]. Table 3.1 illustrates the memory and fill-in for the global ILU technique for fill-in values of $k = 0$, 1, and 2 on a 16×80 backward-facing step model problem similar to the problem in this study. Table 3.2 shows the memory requirements of additive and multiplicative Schwarz preconditioning for various subdomain strategies on the same problem. Clearly, the Schwarz techniques require more memory, but as the number of subblocks are increased these methods become competitive with ILU techniques.

The row labeled 1×1 in Table 3.2 requires further explanation. For this data, a Schwarz technique with only one block was examined. This solution is simply a direct inversion of the complete Jacobian matrix using LINPACK banded Gaussian elimination.

ILU(k) Preconditioning (reverse row ordering)		
k	# non-zero diagonals	Memory (Mbytes)
0	35	1.4
1	59	2.4
2	94	3.9

Table 3.1: ILU memory requirements (adapted from McHugh [1]).

Domain-Based Preconditioning (Additive Schwarz (AS) and Multiplicative Schwarz (MS))				
# blocks in x-direction	# blocks in y-direction	# overlap cells	Reference name	Memory (Mbytes)
4	20	0	4x20-0-AS & 4x20-0-MS	2.4
2	10	0	2x10-0-AS & 2x10-0-MS	4.3
4	20	2	4x20-2-AS & 4x20-2-MS	5.3
2	10	2	2x10-2-AS & 2x10-2-MS	6.8
1	5	0	1x5-0-AS & 1x5-0-MS	8.3
1	1	0	1x1	8.3
1	5	2	1x5-2-AS & 1x5-2-MS	9.3

Table 3.2: Schwarz memory requirements (from McHugh [1]).

Re	Precond. Selection	Mach = 0.25			Mach = 0.025			Mach = 0.0025		
		n	$\overline{m}$	CPU (sec)	n	$\overline{m}$	CPU (sec)	n	$\overline{m}$	CPU (sec)
100	ILU(0)	NS	NS	NS	NS	NS	NS	NS	NS	NS
	4x20-0-AS	8	93	178	7	140	222	7	184	325
	4x20-0-MS	7	50	124	7	73	168	NS	NS	NS
	ILU(1)	NS	NS	NS	NS	NS	NS	NS	NS	NS
	ILU(2)	8	39	431	NS	NS	NS	NS	NS	NS
	2x10-0-AS	8	41	120	7	62	145	7	110	235
	2x10-0-MS	7	21	81	7	30	103	NS	NS	NS
	4x20-2-AS	8	82	251	7	109	305	8	141	435
	4x20-2-MS	7	28	132	7	44	188	7	72	286
	2x10-2-AS	7	40	134	7	54	169	7	71	210
	2x10-2-MS	7	18	93	7	26	120	7	47	188
	1x5-0-AS	8	19	106	7	26	114	7	39	151
	1x5-0-MS	7	9	70	7	11	76	7	19	103
	1x1	7	0	43	7	0	43	7	0	43
	1x5-2-AS	7	18	96	7	21	107	7	33	142
	1x5-2-MS	7	7	70	7	8	72	7	12	147
10	ILU(0)	9	109	319	NS	NS	NS	NS	NS	NS
	4x20-0-AS	7	85	145	6	140	191	6	166	222
	4x20-0-MS	7	48	121	6	64	131	NS	NS	NS
	ILU(1)	7	6	71	NS	NS	NS	NS	NS	NS
	ILU(2)	7	2	105	NS	NS	NS	NS	NS	NS
	2x10-0-AS	7	38	101	6	58	120	6	86	164
	2x10-0-MS	7	22	85	7	22	85	NS	NS	NS
	4x20-2-AS	7	58	179	6	73	184	5	97	197
	4x20-2-MS	7	22	115	7	25	125	5	40	126
	2x10-2-AS	6	30	96	6	39	114	5	49	115
	2x10-2-MS	6	14	69	6	15	74	5	28	92
	1x5-0-AS	7	15	86	6	21	89	5	23	79
	1x5-0-MS	6	8	63	5	9	56	6	12	73
	1x1	6	0	38	4	0	26	5	0	32
	1x5-2-AS	7	13	82	6	11	67	4	16	55
	1x5-2-MS	6	5	56	6	5	54	5	7	51

Table 3.3: Iterative behavior of several preconditioners (from McHugh [1]).

To best illustrate the solution robustness property, consider a comparison of selected Schwarz preconditioning techniques with ILU preconditioning (Table 3.3). This table compares the methods for the model problem at selected Reynolds (Re) and Mach numbers where n is the number of Newton iterations and $\overline{m}$ is the average number of inner (Krylov) iterations required per Newton iteration for a converged solution. Entries of NS in a column indicate a solution was not obtained.

It is clear from these results that ILU techniques do not provide robust solutions to low Mach number flow problems. The performance of the 1×1 "direct" solve is somewhat surprising, considering the number of operations performed ($O(n^3)$). The direct solve provided the minimal execution time of all the simulations, which may be partially attributed to elimination of the Krylov solve operations (with the true inverse of the Jacobian for a preconditioner, only a matrix-vector multiply is needed for the linear system solution each Newton iteration). Additionally, the LINPACK Gaussian elimination routine used for the inversion is highly optimized, while little scalar optimization has been performed in the Krylov solution code. Finally, the small size of this model problem (16×80) may skew the results in favor of the direct method. For larger problem sizes the difference in operation count between the direct method and the preconditioned Krylov techniques should result in better relative performance of the Krylov techniques, especially when the operations are spread over multiple processors. Of further note, the direct solve is quite memory intensive (see Table 3.2), surpassed only by a Schwarz method using subdomain overlap. These extreme memory requirements render the direct method infeasible for large problems.

3.4 Krylov Subspace Algorithms

The discussion of the solution of the non-linear algebraic system began with the use of Newton's method to linearize the system. Within each Newton iteration, a new linear system is formed and solved. The use of a suitable iterative linear solution technique in combination with the inexact Newton's method was previously explained as a strategy to reduce the computational effort and memory requirements for the linear system solution. Additionally, preconditioning was introduced as a method to improve the condition number of the Jacobian matrix to further decrease the computational effort required for a solution. In the development of the preconditioner, parallelism of the algorithms was discussed as an important requirement of an efficient linear solution process. This section on Krylov subspace algorithms will examine possible choices for the actual algorithm(s) used in the iterative solution of the preconditioned linear system.

Recall the form of the preconditioned linear system developed earlier

$$\mathbf{P}_l^{-1}\mathbf{J}^n\delta\mathbf{x}^n = -\mathbf{P}_l^{-1}\mathbf{F}(\mathbf{x}^n). \tag{3.32}$$

This system is of the form

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.33}$$

where, in general, the matrix $\mathbf{A}$ (and $\mathbf{J}$) is a non-symmetric indefinite matrix, and of sparse banded structure. Banded Gaussian elimination and other direct methods such as LU decomposition were discounted as solution techniques due to extreme computational requirements. Furthermore, the inexact Newton's method requires an iterative technique to be effective as described previously. Several approximate methods are available that may be useful on a

system of this form. Examples of these methods include:

- incomplete LU factorization methods (ILU),
- QR decomposition techniques,
- Jacobi related techniques,
- Gauss-Seidel-based techniques, and
- conjugate-gradient-"like" methods (Krylov techniques).

Of these methods, only the conjugate-gradient-"like" methods (Krylov techniques) possess all of the qualities desired for this study.

Matrix-splitting methods such as Gauss-Seidel and Jacobi often yield poor rates of convergence [20, 43]. The convergence rate of these and other relaxation techniques depends strongly on the spacial discretization adopted for the implicit operator. ILU techniques and QR decomposition also may exhibit slow convergence. These types of methods are not generally *robust.* To handle a wide diversity of problems and numerically challenging problems, a robust technique is desired for the solution of Equation 3.32.

A robust technique should have the following properties.

1. The solution technique is guaranteed to converge in n iterations or less (the solution effort is bounded).
2. The technique exhibits a finite termination property.

Krylov subspace methods are robust techniques for the solution of linear systems such as Equation 3.32. These methods (using exact precision mathematics) will converge within n iterations, and often yield satisfactory convergence in much less than n iterations (they

typically converge rapidly). These techniques compute approximations to $\mathbf{x}$ in the affine space

$$\mathbf{x}_o + K_m, \tag{3.34}$$

where $\mathbf{x}_o$ is the initial guess to the solution and K_m is the Krylov subspace of dimension m [58],

$$K_m(\mathbf{r}_o, \mathbf{A}) \equiv \text{span}(\mathbf{r}_o, \mathbf{A}\mathbf{r}_o, \mathbf{A}^2\mathbf{r}_o, \ldots, \mathbf{A}^{m-1}\mathbf{r}_o), \tag{3.35}$$

with

$$\mathbf{r}_o = \mathbf{b} - \mathbf{A}\mathbf{x}_o. \tag{3.36}$$

Conjugate-gradient-"like" algorithms are Krylov algorithms that are derived by relaxing one or both of the two properties that define a true conjugate-gradient method, namely optimality (error reduction) and short vector recurrences (constant work and storage requirements per iteration). Economical vector recurrences can be obtained, at the expense of optimality, via the Lanczos biorthogonalization procedure. Other methods, such as the Arnoldi-based GMRES (the non-restarted version), sacrifice short vector recurrences to achieve optimality. Examples of each will be examined in this study.

Several candidate Krylov techniques exist: the generalized minimal residual method (GMRES) [24], the transpose-free quasi-minimal residual method (TFQMR) [59], the conjugate-gradient-squared algorithm (CGS) [26], and the Bi-CGSTAB algorithm [25], to name a few. Of this set, the GMRES and TFQMR algorithms will be used in this study, due to thier excellent performance on related problems [1].

Krylov techniques remain a fertile research area [21, 60, 61, 62, 63, 64, 65, 66]. McHugh [20] presents an excellent summary of these techniques, and an overview of the mathematical basis

and development of the various methods.

These algorithms were developed to solve the system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.37}$$

or

$$0 = \mathbf{b} - \mathbf{A}\mathbf{x}. \tag{3.38}$$

A direct inversion method is based on an exact solution (within machine accuracy) to Equation 3.38. With an iterative technique we do not expect an exact solution, only a solution "sufficiently accurate" for our needs. In effect, we may select the number of iterations of the technique, m, to provide a desired level of accuracy

$$\lim_{m\to\infty} \{\mathbf{b} - \mathbf{A}\mathbf{x}^m\} = 0, \tag{3.39}$$

or, more practically, develop the algorithm to halt when a certain level of accuracy has been achieved. Thus, for a given iteration m, there is an error in the solution, defined as the *residual*

$$r_m \equiv \mathbf{b} - \mathbf{A}\mathbf{x}^m. \tag{3.40}$$

Minimal residual approaches are often based on the concept of minimizing the L_2-norm of the residual

$$\|\mathbf{r}_m\|_2 = \|\mathbf{b} - \mathbf{A}\mathbf{x}^m\|_2 . \tag{3.41}$$

This step is equivalent to the minimizing the functional

$$g(\mathbf{x}^m) = (\mathbf{b} - \mathbf{A}\mathbf{x}^m)^T(\mathbf{b} - \mathbf{A}\mathbf{x}^m) = \mathbf{x}^{mT}\mathbf{A}^T\mathbf{A}\mathbf{x}^m - 2\mathbf{b}^T\mathbf{A}\mathbf{x}^m + \mathbf{b}^T\mathbf{b}. \tag{3.42}$$

At this point, we desire a solution of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_o + \sum_{j=0}^{k} \alpha_j \mathbf{p}_j = \mathbf{x}_k + \alpha_k \mathbf{p}_k, \tag{3.43}$$

where α_k is a scaling factor and $\mathbf{p}_k$ is a search direction. Substituting Equation 3.43 into Equation 3.42, and minimizing the result with respect to α_k ($\partial g(\mathbf{x}_{k+1})/\partial\alpha_k = 0$) results in the expression

$$\alpha_k = \frac{(\mathbf{A}\mathbf{p}_k)^T\mathbf{r}_k}{(\mathbf{A}\mathbf{p}_k)^T\mathbf{A}\mathbf{p}_k} = \frac{(\mathbf{A}\mathbf{p}_k, \mathbf{r}_k)}{(\mathbf{A}\mathbf{p}_k, \mathbf{A}\mathbf{p}_k)}. \tag{3.44}$$

At this point, search directions are computed ($\mathbf{p}_k$) by selecting the appropriate Krylov subspace L_k [20] and then using either the Lanczos biorthogonalization procedure as defined in the following section, or the Arnoldi process as outlined in Section 3.4.2.

3.4.1 Transpose-Free Quasi-Minimal Residual Method (TFQMR)

The TFQMR solution technique is based on the Lanczos biorthogonalization procedure applicable to general non-symmetric matrices. This class of methods sacrifice optimality (minimizing the residual with respect to a fixed norm) to obtain short vector recurrences. It is not possible, in general, to satisfy an optimality condition with non-symmetric systems using short vector recurrences [67].

As an introduction to the TFQMR algorithm, consider a development of the biconjugate-gradient method (BCG) for the solution of non-symmetric indefinite problems [22, 25, 26, 62,

64, 68, 69]. The BCG method uses recurrence relations developed for the conjugate-gradient algorithm extended with the use of the Lanczos method. This algorithm is developed to solve the system

$$\mathbf{A}\mathbf{x} = \mathbf{b}, \tag{3.45}$$

by the use of an auxiliary system

$$\mathbf{A}^*\tilde{\mathbf{x}} = \tilde{\mathbf{b}}, \tag{3.46}$$

where $\mathbf{A}^*$ is the adjoint of $\mathbf{A}$ with respect to the inner product.

For the conjugate-gradient method, Equation 3.45 may be represented by the functional

$$f(\mathbf{x}) = \frac{1}{2}\left[(\mathbf{A}\mathbf{x} - \mathbf{b})^T(\mathbf{x} - \mathbf{A}^{-1}\mathbf{b})\right], \tag{3.47}$$

or

$$f(\mathbf{x}) = \frac{1}{2}(\mathbf{r}, \mathbf{A}^{-1}\mathbf{r}), \tag{3.48}$$

where $\mathbf{r} = \mathbf{A}\mathbf{x} - \mathbf{b}$ and $(\cdot, \cdot)$ denotes an inner product. To minimize the residual of the calculation, one must minimize this functional. Clearly, the direct solution $\mathbf{x} = \mathbf{A}^{-1}\mathbf{b}$ minimizes the functional, but is expensive to obtain as discussed previously. As a potentially less-expensive method to obtain the minima of the functional, the Krylov vector space

$$K_k = \text{span}(\mathbf{v}_o, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k), \tag{3.49}$$

spanned by the mutually conjugate orthogonal vectors $\mathbf{v}_o, \mathbf{v}_1, \mathbf{v}_2, \ldots, \mathbf{v}_k$, is iteratively searched to obtain the vector that minimizes the residual [22]. As the residuals (gradients) corresponding to each of the search vectors are orthogonal, an iterative update $\mathbf{x}_{k+1} = \mathbf{x}_k + \mathbf{v}, \forall \mathbf{v} \in K_k$

will "spiral" towards the $\mathbf{x}_{k+1}$ resulting in a minimum residual.

To minimize the functional, iterates of the form

$$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{v}_k, \tag{3.50}$$

are desired. This result may also be expressed in terms of the residual

$$(\mathbf{A}\mathbf{x}_{k+1} - \mathbf{b}) = (\mathbf{A}\mathbf{x}_k - \mathbf{b}) + \alpha_k \mathbf{A}\mathbf{v}_k, \tag{3.51}$$

or

$$\mathbf{r}_{k+1} = r_k + \alpha_k \mathbf{A}\mathbf{v}_k. \tag{3.52}$$

To minimize the functional $f(\mathbf{x})$, the derivative with respect to α is performed

$$\frac{\partial f(\mathbf{x})}{\partial \alpha}, \tag{3.53}$$

or

$$\lim_{\alpha \to 0} \frac{1}{\alpha} [f(\mathbf{x} + \delta\mathbf{x}) - f(\mathbf{x})] = 0, \tag{3.54}$$

where

$$\begin{aligned} f(\mathbf{x} + \delta\mathbf{x}) - f(\mathbf{x}) &= f(\mathbf{x} + \alpha\mathbf{v}) - f(\mathbf{x}) \\ &= \frac{1}{2} [\mathbf{r} + \alpha\mathbf{A}\mathbf{v}, \mathbf{A}^{-1}(\mathbf{r} + \alpha\mathbf{A}\mathbf{v})] - \frac{1}{2}(\mathbf{r}, \mathbf{A}^{-1}\mathbf{v}) \\ &= \frac{1}{2} [(\mathbf{r} + \alpha\mathbf{A}\mathbf{v}) \cdot (\mathbf{A}^{-1}\mathbf{r} + \alpha\mathbf{v}) - \mathbf{r} \cdot \mathbf{A}^{-1}\mathbf{r}] \\ &= \frac{1}{2}\alpha(\mathbf{r} \cdot \mathbf{v} + \mathbf{v} \cdot \mathbf{r}) + \frac{1}{2}\alpha^2 \mathbf{A}\mathbf{v} \cdot \mathbf{v} \end{aligned}$$

$$= \alpha(\mathbf{r},\mathbf{v}) + \frac{1}{2}\alpha^2(\mathbf{v},\mathbf{Av}). \tag{3.55}$$

Minimizing this result with respect to α results in

$$\alpha_k \equiv -\frac{(\mathbf{r}_k,\mathbf{v}_k)}{(\mathbf{v}_k,\mathbf{Av}_k)}. \tag{3.56}$$

This result may be further simplified by letting

$$\mathbf{v}_k = -\mathbf{r}_k + \beta_{k-1}\mathbf{v}_{k-1}, \tag{3.57}$$

so that

$$(\mathbf{r}_k,\mathbf{v}_k) = -(\mathbf{r}_k,\mathbf{r}_k) + \beta_{k-1}(\mathbf{r}_k,\mathbf{v}_{k-1}). \tag{3.58}$$

In addition,

$$(\mathbf{r}_k,\mathbf{v}_{k-1}) = 0, \tag{3.59}$$

because the current residual is orthogonal to the previous search direction. Thus,

$$(\mathbf{r}_k,\mathbf{v}_k) = -(\mathbf{r}_k,\mathbf{r}_k), \tag{3.60}$$

which simplifies Equation 3.56 to

$$\alpha_k = \frac{(\mathbf{r}_k,\mathbf{r}_k)}{(\mathbf{v}_k,\mathbf{Av}_k)}. \tag{3.61}$$

Equation 3.57 may be used to determine β

$$(\mathbf{v}_{k+1},\mathbf{Av}_k) = -(\mathbf{r}_{k+1},\mathbf{Av}_k) + \beta_k(\mathbf{v}_k,\mathbf{Av}_k). \tag{3.62}$$

Again, $(\mathbf{v}_{k+1}, \mathbf{A}\mathbf{v}_k) = 0$ because the current and previous search directions are orthogonal. Thus,

$$\beta_k = \frac{(\mathbf{r}_{k+1}, \mathbf{A}\mathbf{v}_k)}{(\mathbf{v}_k, \mathbf{A}\mathbf{v}_k)}. \tag{3.63}$$

This expression may be further simplified by noting

$$(\mathbf{r}_{k+1}, \mathbf{A}\mathbf{v}_k) = \frac{1}{\alpha_k}(\mathbf{r}_{k+1}, \mathbf{r}_{k+1} - \mathbf{r}_k) = \frac{1}{\alpha_k}(\mathbf{r}_{k+1}, \mathbf{r}_{k+1}), \tag{3.64}$$

and

$$\alpha_k(\mathbf{v}_k, \mathbf{A}\mathbf{v}_k) = (\mathbf{v}_k, \mathbf{r}_{k+1} - \mathbf{r}_k) = -(\mathbf{v}_k, \mathbf{r}_k) = (\mathbf{r}_k, \mathbf{r}_k), \tag{3.65}$$

resulting in

$$\beta_k = \frac{(\mathbf{r}_{k+1}, \mathbf{r}_{k+1})}{(\mathbf{r}_k, \mathbf{r}_k)}. \tag{3.66}$$

The previous sequence of steps outline the conjugate-gradient method for symmetric positive-definite matrices. To extend this result to non-symmetric indefinite matrices, the Lanczos method is used. This technique employs the auxiliary system (Equation 3.46), choosing the parameters α_k and β_k such that the residual vectors of each of the systems ($\mathbf{r}_o, \mathbf{r}_1, \ldots$ and $\tilde{\mathbf{r}}_o, \tilde{\mathbf{r}}_1, \ldots$) are biorthogonal. That is,

$$(\mathbf{r}_i, \tilde{\mathbf{r}}_j) = 0 \qquad i \neq j. \tag{3.67}$$

To complete this process, it is necessary to define a series of expressions for the auxiliary system (identical to the process used for the conjugate-gradient method) and develop α_k and

β_k to satisfy the biorthogonality condition. This results in the definitions

$$\begin{aligned} \tilde{\mathbf{x}}_{k+1} &= \tilde{\mathbf{x}}_k + \alpha_k \tilde{\mathbf{v}}_k \\ \tilde{\mathbf{r}}_{k+1} &= \tilde{r}_k + \alpha_k \mathbf{A}^* \tilde{\mathbf{v}}_k \\ \tilde{\mathbf{v}}_{k+1} &= -\tilde{\mathbf{r}}_{k+1} + \beta_k \tilde{\mathbf{v}}_k. \end{aligned} \tag{3.68}$$

By inspection, the terms α_k and β_k that satisfy the biorthogonality condition are

$$\alpha_k = \frac{(\tilde{\mathbf{r}}_k, \mathbf{r}_k)}{(\tilde{\mathbf{v}}_k, \mathbf{A}\mathbf{v}_k)} \tag{3.69}$$

$$\beta_k = \frac{(\tilde{\mathbf{r}}_{k+1}, \mathbf{r}_{k+1})}{(\tilde{\mathbf{r}}_k, \mathbf{r}_k)}. \tag{3.70}$$

This development results in the following expression of the BCG algorithm.

Algorithm 1 (BCG)

$\underline{\text{var}}$ $\langle \mathbf{x}_o, \tilde{\mathbf{r}}_o \neq 0,$

$\mathbf{r}_o = \mathbf{b} - \mathbf{A}\mathbf{x}_o, \mathbf{v}_o = \mathbf{r}_o,$

$\tilde{\mathbf{v}}_o = \tilde{\mathbf{r}}_o, \mathbf{v}_{-1} = \tilde{\mathbf{v}}_{-1} = 0,$

$\rho_{-1} = 1\rangle;$

$\underline{\text{do}}$ $k = 0, 1, 2, \ldots$

$\rho_k = \tilde{\mathbf{r}}_k^T \mathbf{r}_k;$

$\beta_k = \dfrac{\rho_k}{\rho_{k-1}};$

$\mathbf{v}_k = \mathbf{r}_k + \beta_k \mathbf{v}_{k-1};$

$\tilde{\mathbf{v}}_k = \tilde{\mathbf{r}}_k + \beta_k \tilde{\mathbf{v}}_{k-1};$

$\sigma_k = \tilde{\mathbf{v}}_k^T \mathbf{A}\mathbf{v}_k;$

$\alpha_k = \frac{\rho_k}{\sigma_k}$;

$\mathbf{x}_{k+1} = \mathbf{x}_k + \alpha_k \mathbf{v}_k$;

$\mathbf{r}_{k+1} = \mathbf{r}_k + \alpha_k \mathbf{A}\mathbf{v}_k$;

$\tilde{\mathbf{r}}_{k+1} = \tilde{\mathbf{r}}_k + \alpha_k \mathbf{A}^* \tilde{\mathbf{v}}_k$;

if $\|\mathbf{r}_{k+1}\|_2 <$ *conv criteria* **then exit**; **fi**

od

The BCG algorithm is attractive due to the short vector recurrence relationships used in its development. In the absence of round-off errors and algorithm breakdowns, this method exhibits the finite termination property described in the introduction. However, BCG is susceptable to breakdown if the term $(\tilde{\mathbf{r}}_k, \mathbf{r}_k)$ equals or approaches zero. Additionally, the conjugate $\mathbf{A}^*$ is often replaced by the matrix transpose $\mathbf{A}^T$.

The desire to eliminate the calculation of the matrix transpose and minimize the severity of algorithm breakdown has fueled the development of many transpose-free techniques. The technique employed in this study is the TFQMR algorithm. This algorithm has much in common with the BCG algorithm, but differs in a number of significant aspects. For the sake of brevity, a rigorous development of the TFQMR algorithm (similar to the explanation of the BCG algorithm) will not be presented. A few of the differences will be highlighted to provide an outline of the operation of the algorithm.

In the TFQMR algorithm, the residuals and search directions of the BCG method are replaced by the expressions

$$\mathbf{r}_k = \mathbf{R}_k(\mathbf{A})\mathbf{r}_o$$

$$\tilde{\mathbf{r}}_k = \mathbf{R}_k(\mathbf{A}^T)\tilde{\mathbf{r}}_o$$

$$\begin{aligned} \mathbf{v}_k &= \mathbf{S}_k(\mathbf{A})\mathbf{r}_o \\ \tilde{\mathbf{v}}_k &= \mathbf{S}_k(\mathbf{A}^T)\tilde{\mathbf{r}}_o, \end{aligned}$$

where $\mathbf{R}_k$ and $\mathbf{S}_k$ are polynomials of maximum degree k. Using the definition

$$\mathbf{r}_k^{\text{CGS}} = [\mathbf{R}_k(\mathbf{A})]^2 \mathbf{r}_o, \tag{3.71}$$

allows ρ to be expressed as

$$\rho_k = [\mathbf{R}_k(\mathbf{A}^T)\tilde{\mathbf{r}}_o]^T [\mathbf{R}_k(\mathbf{A})\mathbf{r}_o] = \tilde{\mathbf{r}}_o^T [\mathbf{R}_k(\mathbf{A})]^2 \mathbf{r}_o = \tilde{\mathbf{r}}_o^T \mathbf{r}_k^{\text{CGS}}. \tag{3.72}$$

The TFQMR algorithm is not developed using orthogonal residual representations as was the BCG method. The TFQMR method relies on the minimization of the coefficient of $\|\mathbf{P}_{k+1}\|_2$ in the expression

$$\|\mathbf{r}_k\|_2 \leq \left\| \left(\|\mathbf{r}_o\|_2 \mathbf{e}_1 - \tilde{\mathbf{T}}_k \mathbf{y}_k \right) \right\|_2 \|\mathbf{P}_{k+1}\|_2 , \tag{3.73}$$

where

$$\mathbf{T}_k = \begin{bmatrix} \alpha_1 & \beta_2 & 0 & \cdots & 0 \\ 1 & \alpha_2 & & & \vdots \\ 0 & & \ddots & & \beta_k \\ \vdots & & & 1 & \alpha_k \\ 0 & \cdots & 0 & 0 & 1 \end{bmatrix}, \tag{3.74}$$

$\mathbf{e}_1$ is a unit vector with a "1" in the first component, $\mathbf{y}_k$ is a vector of the minimization conditions

$$\mathbf{y}_k = [\alpha_{k1}, \alpha_{k2}, \ldots, \alpha_{kk}]^T , \tag{3.75}$$

and $\mathbf{P}_k$ and $\mathbf{W}_k$ are vectors of search directions

$$\mathbf{P}_k = [\mathbf{p}_1, \mathbf{p}_2, \ldots, \mathbf{p}_k] \tag{3.76}$$

$$\mathbf{W}_k = [\mathbf{w}_1, \mathbf{w}_2, \ldots, \mathbf{w}_k]. \tag{3.77}$$

The TFQMR algorithm does not minimize the residual in L_2, or any other fixed norm. The residual is minimized in a norm that changes with iteration number [64, 70]

$$\left\| \mathbf{D}_k^{-1} \mathbf{W}_k^T \mathbf{r}_k \right\|_2, \tag{3.78}$$

where

$$\mathbf{D}_k = \mathbf{W}_k^T \mathbf{P}_i. \tag{3.79}$$

The TFQMR algorithm is presented below.

Algorithm 2 (TFQMR)

$\underline{\text{var}}$ $\langle \mathbf{x}_o, \tilde{\mathbf{r}}_o \neq 0,$

$\mathbf{r}_o^{\text{CGS}} = \mathbf{b} - \mathbf{A}\mathbf{x}_o,$

$\mathbf{q}_o = \mathbf{d}_o = \mathbf{p}_{-1} = 0, \rho_{-1} = 1,$

$\nu_o = \eta_o = 0, \tau_o = \left\| \mathbf{r}_o^{\text{CGS}} \right\|_2 \rangle;$

$\underline{\text{do}}$ $k = 0, 1, 2, \ldots$

$\rho_k = \tilde{\mathbf{r}}_o^T \mathbf{r}_k^{\text{CGS}};$

$\beta_k = \dfrac{\rho_k}{\rho_{k-1}};$

$\mathbf{u}_k = r_k^{\text{CGS}} + \beta_k \mathbf{q}_k;$

$\mathbf{p}_k = \mathbf{u}_k + \beta_k (\mathbf{q}_k + \beta_k \mathbf{p}_{k-1});$

$\mathbf{v}_k = \mathbf{A}\mathbf{p}_k;$

$\sigma_k = \tilde{\mathbf{r}}_o^T \mathbf{v}_k;$

$\alpha_k = \dfrac{\rho_k}{\sigma_k};$

$\mathbf{q}_{k+1} = \mathbf{u}_k - \alpha_k \mathbf{v}_k;$

$r_{k+1}^{\text{CGS}} = r_k^{\text{CGS}} - \alpha_k \mathbf{A}(\mathbf{u}_k + \mathbf{q}_{k+1});$

do $m = 2k+1, 2k+2, \ldots$

$$\nu_m = \begin{cases} \sqrt{\left\|r_k^{\text{CGS}}\right\|_2 \left\|r_{k+1}^{\text{CGS}}\right\|_2}/\tau_{m-1} & ; \quad m \text{ odd} \\ \left\|r_{k+1}^{\text{CGS}}\right\|_2 /\tau_{m-1} & ; \quad m \text{ even} \end{cases}$$

$c_m = \dfrac{1}{\sqrt{1+\nu_m}};$

$\tau_m = \tau_{m-1}\nu_m c_m;$

$\eta_m = c_m^2 \alpha_k;$

$$\mathbf{d}_m = \begin{cases} \mathbf{u}_k + \dfrac{\nu_{m-1}^2 \eta_{m-1}}{\alpha_k}\mathbf{d}_{m-1} & ; \quad m \text{ odd} \\ \mathbf{q}_k + \dfrac{\nu_{m-1}^2 \eta_{m-1}}{\alpha_k}\mathbf{d}_{m-1} & ; \quad m \text{ even} \end{cases}$$

$\mathbf{x}_m = \mathbf{x}_{m-1} + \eta_m \mathbf{d}_m;$

Calculate $\|\mathbf{r}_m\|_2$ *or use* $\|\mathbf{r}_m\|_2 \leq \left(\sqrt{m+1}\right)\tau_m;$

if $\|\mathbf{r}_m\|_2 <$ *conv criteria* **then exit**; **fi**

od

od

The TFQMR algorithm eliminates the need to calculate the matrix transpose (it is *transpose-free*). Algorithms based on QMR (including TFQMR) do not suffer the same forms of breakdown as seen with BCG. However, QMR algorithms may still fail due to breakdowns in the Lanczos process. Thus, the TFQMR algorithm is clearly superior to BCG, but may not achieve a solution to all problems. The TFQMR algorithm is applicable to a wide class

of problems, and often provides excellent performance (as demonstrated in the following chapters).

3.4.2 Generalized Minimal Residual Method (GMRES)

Due to the remaining breakdown weaknesses in the TFQMR algorithm, a second algorithm, GMRES, was examined to provide an alternative to the cases where the TFQMR algorithm fails. The GMRES method also provides other advantages that may be important for certain conditions.

Use of the Arnoldi process to generate search directions, instead of a Lanczos biorthogonalization method, makes the GMRES algorithm considerably different from the TFQMR algorithm. The Arnoldi method uses a Gram-Schmidt orthogonalization procedure [71] to generate an orthonormal basis for the Krylov subspace. This process seeks to minimize

$$\|\mathbf{r}_k\|_2 = \|\mathbf{r}_o - \mathbf{A}\mathbf{P}_k\mathbf{y}_k\|_2 = \left\|\mathbf{r}_o - \mathbf{P}_{k+1}\tilde{\mathbf{H}}_k\mathbf{y}_k\right\|_2, \tag{3.80}$$

where

$$\tilde{\mathbf{H}}_k = \begin{bmatrix} h_{11} & h_{12} & h_{13} & \cdots & h_{1k} \\ h_{21} & h_{22} & & & \vdots \\ 0 & & \ddots & & h_{k-1,k} \\ \vdots & & & h_{k,k-1} & h_{kk} \\ 0 & \cdots & 0 & 0 & h_{k+1,k} \end{bmatrix}. \tag{3.81}$$

Use of the Arnoldi technique

$$\mathbf{r}_o = \|\mathbf{r}_o\|_2 \mathbf{p}_1 = \|\mathbf{r}_o\|_2 \mathbf{P}_{k+1}\mathbf{e}_1, \tag{3.82}$$

results in the residual expression

$$\|\mathbf{r}_k\|_2 = \left\| \left(\|\mathbf{r}_o\|_2 \, \mathbf{e}_1 - \tilde{\mathbf{H}}_k \mathbf{y}_k \right) \right\|_2 . \tag{3.83}$$

GMRES is based on a least squares minimization, thus an iterate for Equation 3.83 can always be found. This property makes a breakdown of the GMRES algorithm unlikely. The GMRES algorithm is presented below.

Algorithm 3 (GMRES)

$\underline{\text{var}}$ $\langle \mathbf{x}_o, \mathbf{r}_o = \mathbf{b} - \mathbf{A}\mathbf{x}_o,$

$\mathbf{p}_1 = \dfrac{\mathbf{r}_o}{\|\mathbf{r}_o\|_2} \rangle;$

$\underline{\text{do}}$ $k = 0, 1, 2, \ldots, n$

$\underline{\text{comment}}$: Arnoldi Process

$h_{lk} = (\mathbf{A}\mathbf{p}_l, \mathbf{p}_k);\ l = 1, 2, \ldots, k$

$\tilde{\mathbf{p}}_{k+1} = \mathbf{A}\mathbf{p}_k - \sum_{l=1}^{k} h_{lk}\mathbf{p}_l;$

$h_{k+1,k} = \|\tilde{p}_{k+1}\|_2 ;$

$\mathbf{p}_{k+1} = \dfrac{\tilde{\mathbf{p}}_{k+1}}{h_{k+1,k}};$

Update $\tilde{\mathbf{H}}_k$ *and QR factorization to solve*

$$\min_{\mathbf{y}_k \in K_k} \left\| \left(\|\mathbf{r}_o\|_2 \, \mathbf{e}_1 - \tilde{\mathbf{H}}_k \mathbf{y}_k \right) \right\|_2 ;$$

$\underline{\text{if}}$ $\|\mathbf{r}_k\|_2 <$ *conv criteria*

$\underline{\text{then}}$ *Calculate solution from* $\mathbf{x}_k = \mathbf{x}_o + \mathbf{P}_k \mathbf{y}_k$; $\underline{\text{exit}}$; $\underline{\text{fi}}$

$\underline{\text{od}}$

TFQMR relaxes the optimality condition in favor of achieving short vector recurrence relationships and GMRES seeks to minimize the residual while allowing the storage requirements

and computational effort to grow with the iteration count. This behavior is a significant disadvantage of the GMRES method, however it may be addressed by using a restarted version of the algorithm, GMRES(k), where k is the dimension of the Krylov subspace [24]. With the restarted version, the algorithm is only optimal during each iteration cycle, not throughout the entire solution process. Multiple restarts may result in slow convergence or algorithm stall.

In summary, an overview of four Krylov techniques was presented to provide a flavor of the mechanics of these techniques. Of these four, the derivation of the conjugate-gradient method for symmetric positive-definite matrices was examined in detail for use as a basis for the derivation of the BCG method. The final two algorithms, TFQMR and GMRES, were presented as useful generalizations of the BCG technique and as examples of two Krylov techniques that show much promise for this study. Detailed developments and analyses of these Krylov techniques are available in the literature [20, 24, 64, 72, 73]. Finally, the solution of linear systems using Krylov subspace techniques is an active area of research with new algorithms and modifications to current algorithms becoming available at a rapid rate.

3.5 The Matrix-Free Technique

One may argue that work on a matrix-free technique that "eliminates" the formation of the Jacobian matrix really does not belong in a research effort targeted at obtaining scalable preconditioners for a Newton-Krylov solution procedure. Clearly, the elimination of the need to form the Jacobian matrix decreases the operations and memory required to achieve a solution. In earlier chapters, it was argued that concentration of this effort on preconditioning would likely yield the most efficient implementation of a simulation code to solve the model

problem, since obtaining the preconditioner requires $O(n^3)$ operations. Because the Jacobian formation complexity is less (likely much less) than the preconditioner complexity, it was argued that the Jacobian formation time would become negligible for large domain, fine grid solutions.

The above arguments are true, but neglect a portion of the attractiveness of the matrix-free technique. It is clear that preconditioning is still required for the matrix-free technique; to improve the condition number of the system and thus lead to improved convergence behavior (and possibly enabling convergence on poorly conditioned systems). In order to obtain a preconditioner, the Jacobian matrix must be formed. This appears to be a dilemma; formation of the Jacobian is not necessary for the solution of the system, but is necessary to obtain the preconditioner. If this were truly the case for each Newton iteration, the matrix-free method would be nothing more than a curiosity, limited to the set of problems that do not require preconditioning.

However, it may be possible to form the Jacobian and develop a preconditioner on a subset of the Newton iterations rather than on every Newton iteration. Perhaps it is sufficient to form the Jacobian and develop a preconditioner once every other Newton iteration, or maybe once every five or ten Newton iterations. Clearly, the limiting case here is the quality of the preconditioner and how well it conditions the Jacobian for a given Krylov iteration. As a minimum, a preconditioner must be obtained for the first Newton iteration (the "stale" preconditioner is employed in the subsequent Newton iterations). Furthermore, the number of Newton iterations that can be performed before a new preconditioner is developed will depend on a balance between the convergence efficiency of the current preconditioner and the computational penalty of forming a new preconditioner. This balance is expected to be problem specific.

From the above discussion, it is obvious that if the number of times that the Jacobian and preconditioner are formed in a given solution is reduced, the importance of the efficiency of the preconditioner formation is lessened. As such, the operations that would be expended in calculating the preconditioner may be shifted to the Newton (and Krylov) iteration routine(s). Viewing the differing complexities of these functions, it is clearly desirable to minimize the number of preconditioner formation operations.

3.6 Mechanics

Recall that the inexact Newton method described previously linearizes the algebraic system

$$\mathbf{F}(\mathbf{x}) = 0. \tag{3.84}$$

The linearization process results in a linear system of the form

$$\mathbf{J}^n \delta\mathbf{x}^n = -\mathbf{F}(\mathbf{x}^n), \tag{3.85}$$

or

$$\mathbf{P}_l^{-1}\mathbf{J}^n \delta\mathbf{x}^n = -\mathbf{P}_l^{-1}\mathbf{F}(\mathbf{x}^n) \qquad \text{(left preconditioned form)}, \tag{3.86}$$

or

$$\mathbf{A}\mathbf{x} = \mathbf{b} \qquad \text{(generalized form)}, \tag{3.87}$$

to solve for each Newton iteration n. Furthermore, recall that in the TFQMR and GMRES algorithms, the coefficient matrix ($\mathbf{A}$ in the notation used in the Krylov section) never appeared alone but always as a matrix-vector product ($\mathbf{A}\mathbf{p}_k$ and $\mathbf{A}(\mathbf{u}_k + \mathbf{q}_{k+1})$ in the TFQMR

case, $\mathbf{Ap}_l$ and $\mathbf{Ap}_k$ in the GMRES case). This observation may be interpreted to mean that the matrix **A** would not be required if some other manner was available to calculate the matrix-vector product.

The matrix-free technique is simply an algorithm that may be employed to calculate these matrix vector products, **Aw**, or in Jacobian notation, **Jw**, for a given general vector **w**. The matrix-vector product may be expressed by finite-difference approximations of the form

$$\mathbf{Jw} \approx \frac{\mathbf{F}(\mathbf{x}+\xi\mathbf{w}) - \mathbf{F}(\mathbf{x})}{\xi}, \tag{3.88}$$

where ξ is a perturbation parameter

$$\xi = \frac{1}{N}\sum_{i=1}^{N}\xi_i, \tag{3.89}$$

and

$$\xi_i = \varepsilon x_i + \varrho. \tag{3.90}$$

In the above development, N is the dimension of the state vector **x**, x_i is the i^{th} component of the state vector, ε is a perturbation constant ($\approx O(\text{machine round-off}^2)$), and ϱ is a user-specified perturbation parameter (1.0×10^{-6} for this study) [20].

The matrix-free technique requires the following form of a linear system (inexact Newton) convergence criteria

$$\frac{\left\| \frac{\mathbf{F}(\mathbf{x}^n+\xi\delta\mathbf{x}^n)-\mathbf{F}(\mathbf{x}^n)}{\xi} + \mathbf{F}(\mathbf{x}^n) \right\|}{\|\mathbf{F}(\mathbf{x}^n)\|} < \epsilon. \tag{3.91}$$

The utility of this approximation is immediately obvious. Matrix-vector products are

notoriously computationally expensive to construct, while the above products may be constructed with the relatively inexpensive difference relation (Equation 3.88) in the matrix-free technique. Additionally, all of the operations to form the Jacobian matrix are now unnecessary. Preconditioning complicates the advantage, because as the Jacobian is required for the preconditioner. However, as discussed above, this complication may be mitigated for certain problems.

Finally, the matrix-free technique influences the reasoning employed to obtain a solution. Consider the use of the Newton-Krylov-Schwarz algorithms to obtain a steady-state solution to a problem from a given initial "guess." With the background discussed thus far, an analysis of operation counts would suggest that the number of Newton iterations should be minimized during the solution process. Each Newton iteration entails formation of the Jacobian ($O(n^2)$), and development and application of the preconditioner ($O(n^3)$). The Krylov linear solution method is very inexpensive, comparatively. With the matrix-free technique, a situation may exist whereby the Newton iterations are less expensive than the Krylov iterations if one considers the potential of lagging the Jacobian and preconditioner formation operations to amortize this penalty over several Newton iterations. In this case, one would desire a minimization of the Krylov iterations. Realistically, it would be helpful to "tune" the number of Newton and Krylov iterations for a particular problem to maximize solution efficiency. One method that provides a tuning parameter for the number of Newton and Krylov iterations is called *pseudo-transient relaxation.*

Pseudo-transient relaxation applied to the Newton algorithm adds artificial transient terms ($\frac{\mathbf{V}}{\Delta t^n}$) to the main diagonal of the Jacobian matrix in order to create a diagonally-dominant matrix and thereby improve the Newton convergence behavior. This technique

modifies the original linear algebra problem to one of the form

$$\left\{\frac{\mathbf{V}}{\Delta t^{\mathbf{n}}} + \mathbf{J}^{n}\right\} \delta \mathbf{x}^{n} = -\mathbf{F}(\mathbf{x}^{n}). \tag{3.92}$$

This artificial diagonal dominance allows user control over the ratio of Newton to Krylov iterations. Additionally, the pseudo-transient representation provides the welcome benefit of increasing the Newton radius of convergence for difficult solutions.

The effectiveness of this method (matrix-free with pseudo-transient relaxation) is likely problem dependent. Depending on the characteristics of the model problem, this method may be more or less computationally efficient than the direct Newton-Krylov-Schwarz technique described earlier. However, the matrix-free, pseudo-transient (MFPT) method is more robust overall, due to the increase in the radius of convergence of the Newton technique. The methods described to this point are clearly not capable of solving all non-linear algebraic systems. For solution convergence, it is necessary but not sufficient that the initial iteration of the Newton technique (the initial "guess") fall within the Newton radius of convergence. As this initial point may not be easily specified, it is highly desirable that the radius of convergence be as large as possible.

3.7 Summary

This chapter was devoted to an explanation of the inexact Newton-Krylov-Schwarz solution technique, beginning with the non-linear algebraic system and proceeding through all phases of solution. Preconditioning of the linear system was outlined in detail, along with a discussion of the matrix-free approximation and pseudo-transient relaxation of the Jacobian. Figure 3.11 presents a flowchart of the implementation of the solution techniques employed.

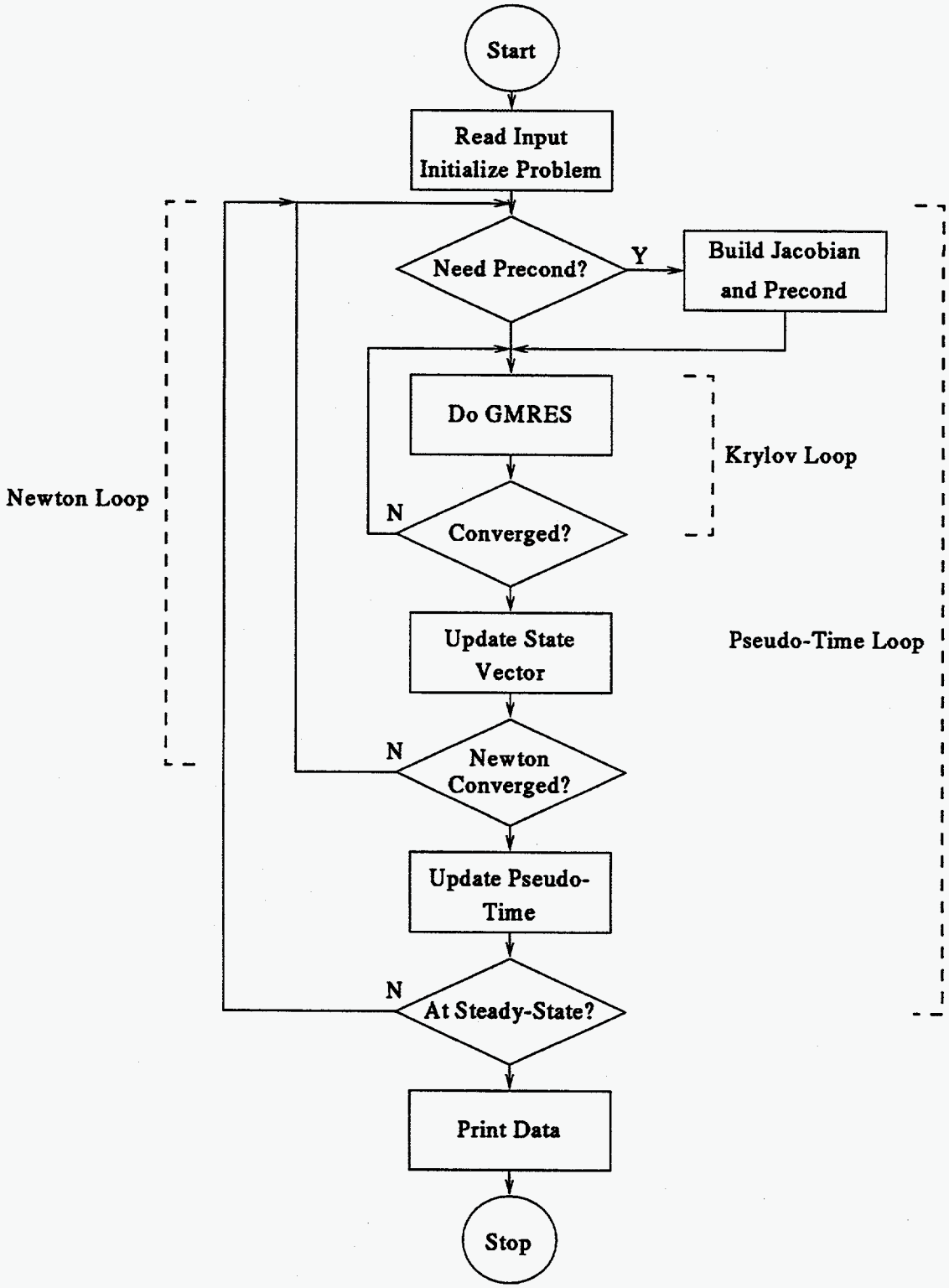

Figure 3.11: Flowchart for Newton-Krylov-Schwarz solution technique.

This discussion completes the specification of the theory, approach, and model problem specifics. The remainder of this study is devoted to an analysis of the mapping of these procedures onto various shared-memory multiprocessors. The primary goal is to develop and optimize the machine-algorithm mapping to result in minimal runtime, while focusing on the scalability of the preconditioning operation. However, this goal must be accomplished without a reduction in the robustness of the algorithm and within reasonable memory bounds.

Chapter 4

The Additive Schwarz Preconditioner

This chapter on the begins the analysis and optimization of a code that employs the Newton-Krylov-Schwarz algorithms described previously. As such, this chapter is not limited solely to a discussion of the preconditioner and its scalability; other topics that affect the preconditioner and the overall code efficiency will also be addressed. Information gathered from the study of these topics is in many cases uniformly applicable to the following chapters on the multiplicative Schwarz preconditioner and matrix-free methods.

Recall the purpose of this study; obtaining scalable performance in the construction and application of a preconditioner to provide efficient solutions to the model problem. In theory, given a scalable preconditioner, one may select the number of processors required to provide a solution of a problem of a particular size in the desired amount of time.

This chapter begins with an effort to obtain efficient parallel/vector solutions to the model problem on two different architectures; a Cray C90 supercomputer and a multiprocessor SGI

Challenge machine (Onyx model). Two sizes of the model problem were initially attempted:

- a 64×320 finite volume discretization ($81,920$ unknowns) on the Cray C90, and
- a similar 32×160 problem ($20,480$ unknowns) on the SGI Onyx.

4.1 Architecture Overview

For this study, the model problem was mapped to two different machine architectures. The first was an example of a traditional supercomputer, the Cray C90. Three Cray C90 machines were available for this research; the first was a four processor machine equipped with 256 Mwords (2.048 Gbytes) of main memory, the second had eight processors and 512 Mwords of memory, and the third was equipped with 16 processors and 512 Mwords of memory.

Architecturally, the Cray machines implement 256 banks of SRAM memory connected to the processors through a crossbar network. Pipelined access may be employed for efficient memory access. Memory is implemented using a "real" (physical) addressing mode into the banks, virtual memory and caching is not available. The processors operate at a clock speed of 4.2 ns, and contain two vector pipes and two functional units per processor. Thus, with eight processors $8 \times 2 \times 2 = 32$-way parallelism is possible.

The SGI Challenge machine has four processors. This machine is equipped with 1 Gbyte of conventional DRAM main memory connected to the four processors by a proprietary bus. Each processor is a MIPS R4400, containing 1 Mbyte of SRAM cache (512 K data, 512 K instruction), and is of superpipelined superscalar design. The cache controllers implement a snoopy bus protocol for coherence. This machine implements a demand-paged virtual memory scheme.

The simulation code for this study was implemented in well-structured FORTRAN 77.

The base version was developed to be quite portable, using an "elementary" style (see Appendix A). No attempts were made within the original code to increase its suitability toward any particular architecture; the development platform that preceded this work was a uniprocessor UNIX workstation.

Finally, the study was performed during multiuser operation on all hardware examined. The results on dedicated machines could vary significantly. Cray warns that dedicated access to their machines is necessary to fully evaluate the performance aspects.

4.1.1 Cray Optimization

The initial development work was performed on the four processor C90 and was limited to routine compilation tasks and verification of correct operation and results. The vectorization and parallelization passes and all optimization were disabled for the initial porting phase.

Following the initial porting phase, scalar optimization parameters were investigated to obtain an optimal configuration in preparation for a flow analysis of the code. For the flow analysis, the Cray utility `FLOWTRACE` was used to identify code blocks that warrant optimization. Additionally, Cray library calls corresponding to the LINPACK routines used in the original code were substituted to provide better efficiency. `FLOWTRACE` results indicated that several functions and subprograms were suitable for inlining (for sample FLOWTRACE output, see Appendix B). These routines were explicitly inlined with the use of compiler directives and the requisite compile options. Following this work, `FLOWTRACE` was again used to find that several routines comprised the bulk of the execution time for the simulation code. These routines were those involved in forming the Jacobian matrix, mainly:

- the u-momentum contribution routine *umom*,

- the v-momentum routine *vmom*,
- the routine for the mass-conservation contribution *cont*, and
- the temperature routine *temp*.

Additionally, the routine that forms and factors the preconditioner, *precond*, required an appreciable amount of computation time. Following these routines in percentage of execution time were several utility routines, *matmul* - general matrix-vector multiply, *mivmldd* - a banded linear equation solution routine, and *extrcv* - a data extraction routine.

Following this analysis, the vectorizing phase of the compilation system was invoked. Vectorization proved quite effective in reducing the overall runtime of the code. Reductions of over an order of magnitude were seen in some loops, resulting in nearly a factor of ten speedup overall. However, in a nested looping construct, only the innermost loops may be vectorized. The Jacobian and preconditioner formation routines contained many such structures and continued to dominate the execution time despite vectorized inner loops.

The logical progression of this study suggested that the enclosing loops in these routines should be uniformly divided and executed on multiple processors. However, the Cray C90 series machines have an appreciable amount of parallel overhead, attributable to the following sources.

- Semaphore wait time. At the end of a parallel region, all threads must synchronize prior to the main thread continuing. If the main thread finishes last, this time is zero. However, if not, the main thread must wait (for a time that depends on the load balance among the threads) until all child threads synchronize at the exit point.
- Extra autotasking code. Executable code is added by the autotasking mechanism to

create and manage the multiple threads. Some of this is executed by the master thread prior to *forking* the child threads and leads to additional serial overhead.

- Increase in memory bank contention. If contention exists on a single processor, it will be generally be greatly compounded with multiple CPU's.

- Decrease in vector performance. If parallelism is implemented in a manner such that the vector length is shortened or chaining is prevented, the vector efficiency is reduced.

On average, autotasking startup and executing the extra autotasking code on a dedicated machine requires 3600 clock cycles on the C90 [3]. Appendix A details many of the challenges of mapping an algorithm to the Cray architecture for best performance, and explains the memory contention problem in more detail.

4.1.2 SGI Optimization

The SGI Onyx optimization process was initiated near the end of the Cray study. A quick examination of the scalar behavior of the code using `prof -pixie` verified that the same routines that required significant CPU time on the Cray also required significant CPU time on the SGI. It was quickly concluded that any work to enhance parallelization on the Cray also benefited the SGI implementation.

The SGI is a cached, superscalar superpipelined architecture in contrast to the vector-based Cray architecture. As such, user-level vectorization was not possible on the SGI. In fact, for most efficient use of the SGI architecture, programs should be written to preserve data locality rather than vector efficiency. An extensive study to best match the serial behavior of the code to the cache-based SGI was not performed. Additionally, it was desirable to maintain excellent performance with a single copy of the code executable on either architecture. Thus,

SGI scalar optimizations that would severely impact the Cray vector performance of the code were avoided. However, a cursory study with the base code versus the Cray vectorized version on the SGI showed few differences that could be attributed to efforts to better vectorize the Cray version. Thus, this version was used "as is" on the SGI.

Aside from those issues specific to vectorization, the SGI suffers from the same parallel overhead concerns that afflict the Cray. The SGI uses a similar interleaved memory design. However, all memory accesses are via a local processor cache. The Cray uses a crossbar switch for memory bank access, while the SGI uses a proprietary bus. It is conceivable that fewer memory contention difficulties are encountered on the SGI due to the fewer processors, local processor cache, and the slower processor speed of the machine.

4.2 Initial Results

The backward-facing step model problem with an inlet Mach number of 0.0025 and a Reynolds number of 100 was selected to investigate the efficiency of the numerical solution algorithm described previously. Specifically, the vector/parallel aspects of this solution algorithm are highlighted.

An HP 735/125 workstation with 36Mwords (288Mbytes) of RAM was used as the baseline machine. The Cray runs were performed on an 8 processor C90 with 512Mwords of memory. All runs were performed in double precision on the workstation (64 bit, 52 bit mantissa) and single precision on the Cray (64 bit, 48 bit mantissa). Simulation results were identical to machine precision (the Cray has slightly less precision than the HP in the comparison, however this difference is not significant), and were verified for all comparisons. No attempts were made to achieve dedicated access to either machine, they were both operating in a

production, multi-user mode. Little effort was devoted to create an optimized version of the code for the HP platform. In fact, ignoring Cray vectorization and autotasking directives (that appear as comments to the HP compiler) the same version of the code was run on both the Cray and the HP. The HP run was compiled under full optimization and employed the HP LINPACK libraries where possible. The single processor Cray run used the Cray LINPACK library, inlining, and vectorized routines where prudent (based on `FLOWTRACE` results and the labor, cost, and feasibility of vectorizing various routines). The multiple CPU run also employed autotasking directives in order to distribute the formation of the Jacobian and the preconditioner across the available CPU's.

In base form, the Jacobian formation algorithm consisted of calls to four subroutines; *cont*, *temp*, *umom*, and *vmom*. Each of these routines contained three distinct sections.

1. The first section computes the residual for the interior finite volume cells (in Ω).

2. The second region formally creates the Jacobian matrix by sweeping over the matrix in an element by element fashion.

3. The last section initializes unused values in the Jacobian to zero, along with corresponding positions in the residual array.

`FLOWTRACE` indicated that only the first two sections warranted parallelization. Thus, for each of the subroutines, two parallel regions were created (8 total parallel regions executed for each Jacobian update).

For the initial runs, the additive Schwarz preconditioner formation and application tasks were also parallelized, because these routines directly followed the Jacobian formation routines in computational cost for this size of the model problem. The formation of the preconditioner was separated into two parallel sections; the first factors each of the subdomains while

the second performs a direct inversion of each factored subdomain followed by a solution of the resulting system

$$(\mathbf{LU})\mathbf{x} = \mathbf{y}, \tag{4.1}$$

for $\mathbf{x}$ (where $\mathbf{LU}$ is the result of the factorization procedure).

The following results do not isolate the effectiveness of the parallelization of the preconditioner for simplicity. The preconditioner operations add to the time required to obtain a solution to the linear system (*i.e.*, perform the Krylov iterations). Although execution times are presented for the entire linear solution process, the preconditioner is expected to overwhelm the remaining operations for larger problem sizes.

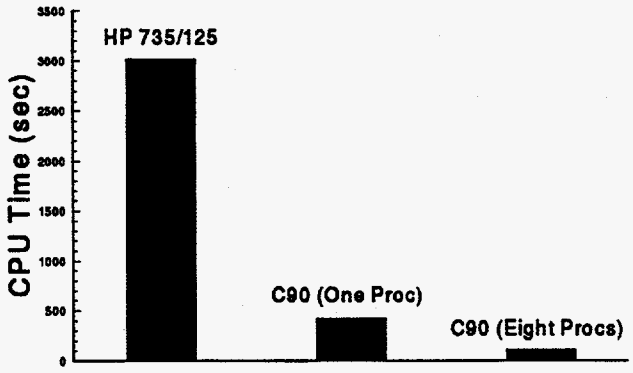

Figure 4.1: 64×320 domain solution time.

Figure 4.1 illustrates the CPU time required to obtain a solution for this model problem using a 64×320 grid (81,920 unknowns) with an 8 block additive Schwarz preconditioner. The HP required 3017 sec., the C90 using a single processor needed 424 sec., and the C90 with 8 processors spent 118 sec. to obtain a converged steady-state solution. In comparison with the baseline machine, the Cray C90 architecture clearly yields a significant performance advantage and is a viable platform for this study.

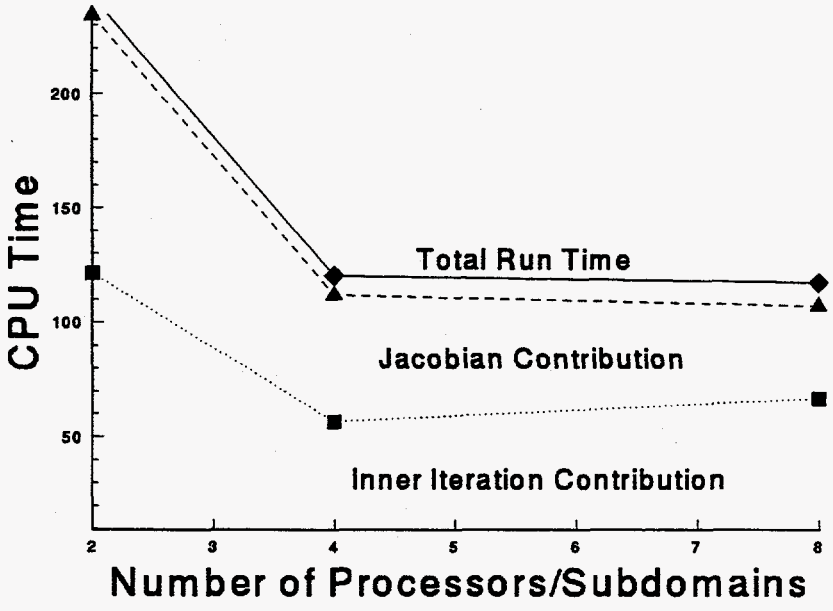

Figure 4.2: Majority of execution time devoted to Jacobian formation and TFQMR iterations on the C90.

Figure 4.2 illustrates the parallel performance on the C90 for two, four, and eight sub-blocks (mapping to an equal number of processors) for this simulation. This figure reinforces the `FLOWTRACE` results that indicate that the formation of the Jacobian and performing the inner (Krylov) iterations comprise the bulk of the execution time of the simulation. For the two-processor case, the inner iterations require 120 of 240 total seconds for execution. The combination of the Jacobian and inner iterations contribute 230 seconds of the total time (96%). Several other interesting phenomenon are apparent from this graph.

- The Jacobian routine displays a consistent decrease in execution time as the number of processors are increased.
- An increase in processors from two to four substantially decreased the inner iteration time. However, a further increase to eight processors resulted in an increase in execution time for this region. Recall that only the preconditioner portion of the inner iteration

time is parallel.

- As the number of processors was increased, the effect on the remaining code execution time was negligible. Because the remaining code is serial, changes in the runtime behavior of these routines with an increase in processors was not expected.

Num. of Subdomains	Linear Solve CPU Time (% of Total)	Jacobian Formation CPU Time (% of Total)	Total CPU Time (sec.)
2	50.0	46.5	242.30
4	47.0	46.0	120.24
8	56.9	34.3	117.67

Table 4.1: Contributions towards total CPU time.

Table 4.1 presents the percentage of Cray CPU time spent forming the Jacobian and the percentage of CPU time required to solve the linear system arising on each Newton step (the sum of the time required to form the preconditioner and to perform the TFQMR iterations) for 2, 4, and 8 processors. This data indicates that the formation of the Jacobian and the solution of the linear system continue to dominate the solution time when executed in parallel.

Figure 4.3 illustrates the convergence behavior of the Newton-TFQMR solution algorithm for this sequence of results. Clearly, on this model problem, monotonic Newton convergence has been achieved. This result further reinforces the selection of the Newton-Krylov-Schwarz solution technique as a viable solution technique for numerically challenging non-linear problems with a similar character to that of the model problem.

Table 4.2 illustrates the memory requirements for the solution of the 64×320 problem (no subdomain overlap). Memory usage for the solution is a function of the problem size, the number of subdomains in the preconditioner, subdomain overlap (if present), and the

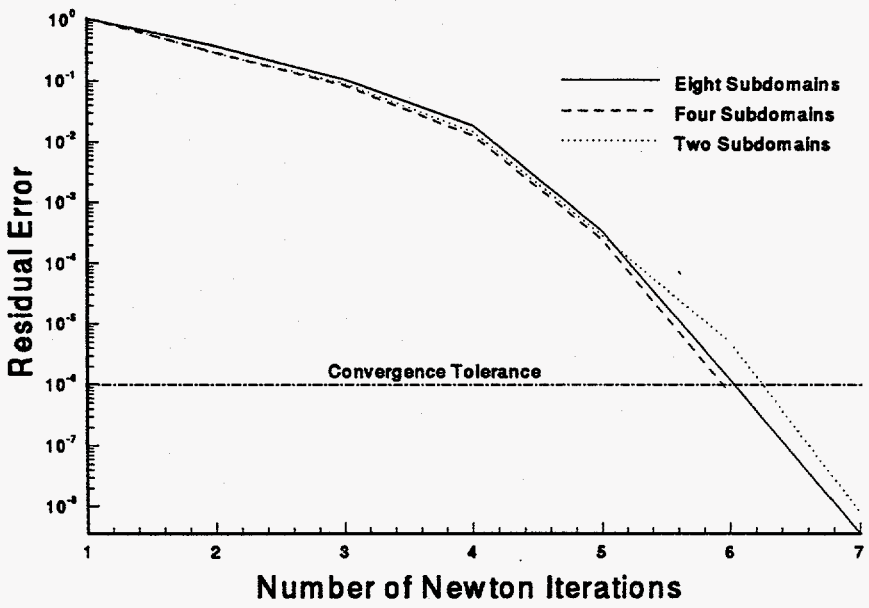

Figure 4.3: Convergence behavior of the Newton-Krylov-Schwarz algorithm.

decomposition strategy. The problem size dependence is easily seen; as the problem size is increased, the Jacobian contains more cell entries. As the Jacobian becomes larger, the memory required for the preconditioner increases along with the memory required to invert each preconditioner subdomain. If the number of subdomains is increased, each subdomain is smaller and the memory required for inversion decreases. Overlap makes the subdomains larger, requiring more memory as overlap is increased. The larger subdomains also increase

Num. of Subdomains	Memory Required (Mbytes)
2	259
4	133
8	70

Table 4.2: Memory requirements for Cray 64×320 simulation.

the amount of work performed within a parallel region while the serial sections in the remainder of the code are basically unchanged; thus increasing the *parallel granularity* of the code. Finally, the decomposition strategy (stripwise in the x direction, stripwise in the y direction, or checkerboard) changes the amount memory required for each of the subdomain matrices. As an example, the 8×1 blocking strategy (above) requires 70 Mbytes, a 1×8 strategy requires 510 Mbytes. From this data, it is easily seen that increasing the number of subdomains results in higher DOPs, decreased memory requirements, and greater algorithmic efficiency (due to a lower operation count to invert each subdomain). Thus, the focus of this study on achieving a scalable solution is clearly warranted.

4.2.1 The Jacobian Algorithm

The formation of the Jacobian parallelizes trivially, and exhibits excellent performance on two and four processors (see Figure 4.4). The two-processor speedup is 1.94, while the four-processor speedup is 3.96 (the ideal speedups are 2.0 and 4.0, respectively). The eight-processor speedup is 5.42, which is notably lower than the ideal value of 8. The decrease in parallel efficiency for the eight-processor case is most likely due to a combination of memory contention as the processors update the Jacobian matrix and reduced parallel granularity for this section of the code.

Parallel granularity is an important consideration when mapping parallel algorithms to a given architecture. To achieve optimal parallel performance, one should strive to maximize the time spent in parallel execution in proportion to the time spent in serial execution (for a constant amount of work to be performed). Clearly, this concept will maximize the speedup achieved within the simulation code (see Appendix A). Although this requirement is necessary for optimal execution, it is not sufficient. If the code is constructed using many

small parallel regions separated by synchronization operations, barriers, or serial regions, the overhead required to enter and leave parallel execution may overwhelm any speed increases obtained from the parallel code sections. As such, the parallel regions must form the bulk of the simulation operations and be "sufficiently large" to prevent parallel overhead from seriously degrading the performance of the execution. Clearly, *sufficient* parallel granularity is subjective, and dependent on the implementation and design of the algorithm and the design of the architecture of interest.

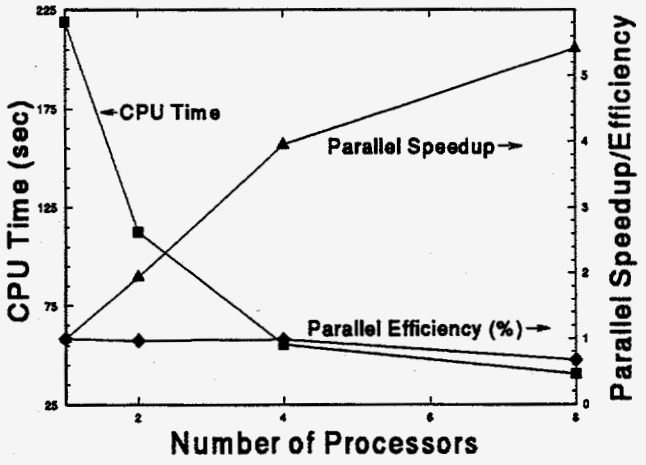

Figure 4.4: Jacobian CPU time, speedup, and efficiency on the C90.

Num. of Sub-domains	Serial CPU Time (sec.)	Parallel CPU Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2	198.48	121.35	1.64	82
4	130.49	56.56	2.31	58
8	196.82	66.94	2.94	37

Table 4.3: Parallel speedup of the linear solution routine on the C90.

4.2.2 The Preconditioner

The routines that form and apply the preconditioner also parallelize trivially. Parallel speedups of 1.64, 2.31, and 2.94 were obtained for the linear solution routine on two, four, and eight processors, respectively (see Table 4.3). Recall that the preconditioner formulation contributes to the time required in the linear solution routine (in the 8 CPU parallel case, only 14% of the linear solution time is spent in forming the preconditioner). Unfortunately, for the problem size considered, most of the remainder of this routine is serial. The large differences in parallel efficiency between the ideal and actual case are attributed to the decrease in global preconditioner effectiveness as the number of subdomains (processors) increases.

Num. Subdomains	Newton Iterations	Avg. TFQMR Iterations
2	7	16
4	6	32
8	7	70

Table 4.4: Solution algorithm performance data.

Table 4.4 illustrates the performance of the solution algorithm for the different subdomain blocking strategies employed. The number of Newton iterations is independent of the number of subdomains chosen. The discrepancy between 7 iterations for the two and eight block case and 6 for the four block case is due to the selection of the Newton convergence criteria. For this problem, the convergence criteria in conjunction with slightly different inner iteration results between the runs allowed the four block case to converge in only 6 iterations. This selection of the criteria was not planned; with a discrete process, such as iteration using a continuous convergence criterion, it is always possible to encounter situations where slight (often very slight) differences in the solution path can result in convergence behavior that

is on the "ragged edge" between two iteration counts. In this case, the convergence criteria could be tightened slightly to result in 7 iterations for all runs, or loosened slightly resulting in a uniform 6 iterations.

This table also illustrates an interesting trend of an increase in the number of TFQMR iterations with an increase in the number of subdomains. The increased number of iterations results from the lack of coupling information between subdomains in the global preconditioner. Thus, the study is presented with a dilemma; a large number of subdomains is desired from an operation count standpoint and for mapping to a large number of processors, but results in the undesirable effect of increasing the number of Krylov iterations to achieve convergence. To better illustrate this problem, let the inner iteration time τ be a combination of the time s required to execute the serial code in the Krylov algorithm and the time p required to execute the parallel preconditioner on a single processor, such that

$$\tau = s + p. \tag{4.2}$$

Furthermore, assume that the serial code time is a constant (does not vary with the number of subdomains or processors), that the number of Krylov iterations are directly proportional to the number of subdomains (and processors) n, and that 100% parallel efficiency can be achieved. Then, τ_p, the parallel execution time is

$$\tau_p = n\tau = n\left(s + \frac{p}{n}\right) = ns + p. \tag{4.3}$$

This result indicates that even if the amount of serial code in the Krylov solution routine was driven to zero, parallelization of the preconditioner is not a feasible method of reducing

the execution time of the code. Fortunately, this result is overly pessimistic; it ignores algorithmic efficiency improvements due to the inversion of progressively smaller subdomains as the number of subdomains increases. Furthermore, the results in Table 4.3 indicate that parallelization of the preconditioner does indeed reduce the inner iteration time, for up to at least to eight subdomains. One may quickly conclude that Equation 4.3 does not capture the true complexity of the inner iteration algorithm. However, this equation does suggest several topics that should be considered.

- Traditional techniques used to enhance parallelism (maximizing granularity, minimizing serial code, *etc.*) may not yield the desired results and/or may not perform as expected.
- Expectations of scalability of the inner iteration routine to large numbers of processors may not be warranted. In fact, these results suggest that there is a point of maximum efficiency corresponding to n processors (possibly eight processors in this case). Adding processors beyond this figure may not result in a further decrease in inner iteration time and could likely increase execution time.
- The increase in Krylov iterations as the number of subdomains is increased will prevent a truly scalable linear solution routine. Under ideal conditions, if the number of Krylov iterations did not increase with the number of subdomains, scalability could be achieved. However, it may be possible to obtain useful performance increases if the number of iterations increases at a slower rate than the number of subdomains

$$\kappa = kn \qquad k < 1, \tag{4.4}$$

where κ is the number of Krylov iterations, n is the number of subdomains, and k

is a proportionality function. Efforts should be directed at decreasing the value of k initially, followed by an attempt to increase the parallel efficiency of the preconditioner formation, and finally by an effort to decrease the serial component of the remaining inner iteration routines.

- For a given simulation, the value of k in Equation 4.4 will likely be problem specific.
- Speedup concerns aside, a secondary benefit of increasing the number of subdomains may be realized on distributed memory architectures. As the bulk of the memory required in a simulation is confined to the preconditioner, distributing this requirement to multiple processors in a distributed memory configuration dramatically decreases the memory required on the "master" process. Additionally, the number of subdomains (processors) employed for a solution could be determined by the memory requirements of the solution; large memory requirements could be spread over many processors enabling solution of problems too large to be attempted on a single processor or a shared-memory machine.

Finally, as a potential mitigating factor to large values of k, note that the use of subdomain overlap can be used to improve the effectiveness of the global preconditioner as the number of subdomains increases, because more of the Jacobian data is captured in each of the subdomains. However, as discussed previously, the larger subdomains associated with increased overlap result in higher operation counts for the subdomain inversion process and increased memory requirements for the solution algorithm. Lastly, the use of subdomain overlap decreases the parallelism of the additive Schwarz method, because access to memory must be serialized when the global preconditioner is updated with information from the subdomains in the overlap areas. Other alternatives that may show promise are listed below.

- The use of a "coarse grain/fine grain preconditioner." A "coarse grain" solution is employed to encapsulate the communication between the nearly decoupled local subdomains [30].

- Use of a different preconditioning strategy such as multiplicative Schwarz may provide better inner iteration behavior as the number of subdomains are increased.

- Use of the matrix-free technique may allow an amortization of the preconditioner and Jacobian formation penalty over several Newton iterations.

The coarse grain/fine grain approach is based on adding additional information to the additive Schwarz expression

$$\mathbf{P}_l^{-1} = \mathbf{J}_1^{-1} + \cdots + \mathbf{J}_p^{-1} = \sum_{i=1}^{p} \mathbf{J}_i^{-1}, \tag{4.5}$$

or expressing the global preconditioner space in terms of the subspaces (subdomains)

$$V = V_1 + V_2 + \cdots + V_i. \tag{4.6}$$

To encapsulate the coarse grid information on the subspace V_o, this subspace is simply added to the above expression

$$V = V_o + V_1 + V_2 + \cdots + V_i. \tag{4.7}$$

The space V_o contains the information concerning the communication of information between the remaining subspaces $V_1, \ldots, V_i$. To construct this information, a coarse grid operator is developed based on the governing equations and satisfying the boundary conditions on $\partial\Omega$. This approach is very effective at minimizing the degradation of the solution as the number

Num. of Sub-domains	Serial CPU Time (sec.)	Parallel CPU Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	426	242	1.8	88
4×1	326	120	2.7	68
8×1	424	118	3.6	45

Table 4.5: Overall code performance.

of subdomains is increased for certain problems [31, 32, 74, 75]. However, for the governing system used in this study, a suitable coarse grid operator has yet to be developed (this general concept is currently an active research topic).

The investigation of multiplicative Schwarz preconditioning will be performed in the following chapter. This technique shows promise in decreasing the Krylov iterations required for convergence [1].

Finally, matrix-free techniques do not directly impact the preconditioner (and linear solution) scalability. However, these techniques may reduce the frequency of preconditioner formation by the use of one preconditioner (a "stale" preconditioner) for several Newton iterations. If the matrix-free technique is scalable and the preconditioner formation may be relaxed sufficiently, useful runtime efficiency improvements may be obtained.

In summary, Table 4.5 illustrates the overall solution performance of the code on the C90 for 2, 4, and 8 processors, respectively. Clearly, excellent performance is achieved using two processors but the parallel efficiency quickly decreases as the number of processors is increased. This behavior is attributed to poor granularity and memory contention in the Jacobian routine, and the increase in Krylov iterations with subdomains discussed earlier. Additionally, granularity and contention problems may also exist with the preconditioner routines. The relative importance of these potential problems requires further study.

4.3 Jacobian Granularity and Contention

It was previously noted that the Jacobian formation routine is "embarrassingly" parallel, leading to a concern about the low parallel speedup on 8 Cray processors (5.4). In an attempt to investigate this result, the Jacobian routine was re-structured to maximize the parallel granularity by encapsulating all of the Jacobian formation routine in a single parallel loop (the base routine had three parallel regions). Additionally, an attempt was made to decrease memory contention in the routine by changing the location and spacing of arrays relative to the memory banks. These efforts resulted in the overall code performance indicated in Table 4.6. In this table, the row labeled 8×1 is the previous data for the 8 block case, the $8 \times 1^*$ row is an identical run using the new Jacobian routine.

Num. of Sub-domains	Serial CPU Time (sec.)	Parallel CPU Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
8×1	424	118	3.6	45
$8 \times 1^*$	294	113	2.6	33

Table 4.6: Overall performance.

Table 4.6 clearly indicates that the reformulated Jacobian reduces the execution time of the code. In fact, a 30% reduction in serial time was observed. The parallel version was also faster, but the 4% decrease was less than expected, in fact the opposite (large parallel reduction, small or zero serial reduction) was anticipated. To better understand this result, Table 4.7 isolates just the Jacobian formation routine performance.

The new Jacobian reduced execution time by 55% and the parallel time by 45%. As the serial performance increased by a greater percentage than the parallel performance, the speedup decreased from 5.5 to 4.5. Recall that the serial runs on the C90 were using full

Num. of Sub-domains	Serial Jacobian Time (sec.)	Parallel Jacobian Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
8×1	219	40	5.5	68
$8 \times 1^*$	99	22	4.5	56

Table 4.7: New Jacobian performance.

vectorization. The parallel runs also used full vectorization and in addition, autotasking. Because the amount, type, and ordering of the work performed was not changed from the original routine, the increase in serial performance is attributed to decreased memory contention and increased vectorization performance due to the increased granularity. The parallel performance also increased due to these efforts, however, the memory contention caused by the multiple processors lessened the rate of increase in comparison to the serial results.

Num. of Sub-domains	Serial TFQMR Time (sec.)	Parallel TFQMR Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
8×1	197	67	2.9	37
$8 \times 1^*$	187	83	2.3	28

Table 4.8: TFQMR routine performance.

For completeness, the effect of the new Jacobian algorithm on the TFQMR iteration routine is shown in Table 4.8. Because this algorithm was not modified for this Jacobian study, one would expect the results here to be identical. This was not the case. The serial results improved by 5% and the parallel results were 24% slower with the new Jacobian routine. This may also be explained by considering memory contention. For the serial run, decreasing contention in the Jacobian routine also decreased the contention in the TFQMR routine (some of the newly spaced arrays used in the Jacobian are also referenced in the

Num. of Sub-domains	Serial Precond. Time (sec.)	Parallel Precond. Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
$8 \times 1^*$	50	15	3.3	42

Table 4.9: Speedup of the additive Schwarz preconditioner formation routine.

TFQMR routine). However, in parallel mode, the new Jacobian access pattern increased the memory contention in the TFQMR routine.

Table 4.9 isolates the parallel preconditioner factorization from the TFQMR routine. From these results, the preconditioner factorization comprises 27% of the serial time and 18% of the parallel TFQMR algorithm for this problem size. Again, the speedup for this routine on 8 processors is not ideal. This section of code cannot be grain-packed further without parallelization of sections of the TFQMR algorithm itself. As discussed earlier, it is unlikely that parallelizing the TFQMR algorithm would be effective without first solving the preconditioner degradation problem. Furthermore, memory contention is also a concern in this routine.

The new Jacobian routine is near optimal in terms of parallel (and vector) granularity. It does not appear likely that further granularity improvements can be achieved at least for this problem size (recall that increasing problem size increases the parallel granularity inside the Jacobian algorithm as more operations are performed in parallel). Additionally, the complex memory access pattern of the Jacobian formation routine makes further gains in reducing memory contention by simple modifications of the FORTRAN common blocks in the code unlikely. Although incremental improvements are possible, approaching the theoretical speedup figure of eight using 8 processors is probably not feasible. Further work along this path has not resulted in any significant improvement in the 4.5 speedup noted in Table 4.7.

Furthermore, the remainder of the code appears sensitive to memory access optimization in the Jacobian routine. Clearly, the scalability of these methods with the current memory structure on the Cray architecture appears unlikely (at this problem size). The best approach would be a complete re-work of the memory structures and perhaps the code for the Jacobian formation that may be specific to the Cray architecture. This approach is considered later in this chapter.

The use of local processor data caching may also be an attractive method to reduce the penalty of memory contention. If the bulk of memory accesses could be directed at a local cache instead of a main memory pool, contention among processors for data items on a common bank may be significantly reduced. A good caching scheme would also implement a virtual memory mapping that would transparently locate often-used data in a particular processor cache. Given this scheme, less expensive DRAM memory could possibly be employed without a significant performance degradation, because the majority of accesses will be to cache instead of main memory. Unfortunately, a Cray-class machine implementing this arrangement is not available.

To consider this approach, an SGI Onyx multiprocessor was used to investigate the feasibility of implementing the Newton-Krylov-Schwarz scheme on a cache-based architecture.

The SGI Onyx differs in many ways from the Cray C90. They are both shared-memory architectures, but they differ in processor speed and performance, vector versus superpipelined superscalar design, number of available processors, and memory access methods. Due to these many differences, it is impossible to isolate any performance variations to just the cached versus banked memory design. Based on algorithm scalability, however, it may be possible to state that the Newton-Krylov-Schwarz method employed in this study maps better to a given architecture.

Num. Subdomains	Newton Iterations	Avg. TFQMR Iterations
2	7	11
4	6	24

Table 4.10: 32×160 Onyx simulation iteration behavior.

To perform this analysis, the code employing the coarse-grained Jacobian routine was examined. All memory access optimization developed for the Cray was replaced by a straightforward storage model that optimizes the use of available memory. As the SGI was limited in memory (1Gbyte), little leeway existed for memory access optimization. Additionally, because the memory access methods varied greatly between the Cray and SGI, techniques that optimized access on the Cray would likely have a reverse effect on the SGI. Finally, due to the reduced number of processors and slower execution speed of the SGI, a smaller problem (32×160) with only 4 subdomains was initially considered.

For the 32×160 simulation on the Onyx, again the Jacobian formation and TFQMR iteration routines comprised the bulk of the execution time. The Jacobian and preconditioner formation and application algorithms were parallelized. Unlike for the C90 case, a benefit was also gained by the parallelization of the algorithm that computes the matrix-vector products (this algorithm also contributes to the TFQMR iteration time). Two versions of this matrix-vector product algorithm were used in this study; a vector version with data dependencies that inhibited parallelization, and a parallel version with a branch that inhibited vectorization. The parallel version provided much better performance in parallel on the SGI, the vector version provided nearly a 10-fold performance improvement versus a 6-fold improvement of the parallel version on 8 processors on the C90. As such, the most appropriate version for the machine in question was employed.

Table 4.10 illustrates the iteration behavior of the model problem on the Onyx. This table confirms the nearly identical algorithm behavior on the machines; the increase in TFQMR iterations with subdomains may be noted along with the difference in Newton iterations between two and four subdomains due to the convergence criteria. Additionally, fewer TFQMR iterations are required on the Onyx. This difference is not due to machine differences, but to the different problem sizes run on the machines (the number of iterations and the rate of increase is likely problem specific).

Num. of Sub-domains	Serial CPU Time (sec.)	Parallel CPU Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	422	188	2.2	112
4×1	375	118	3.2	79

Table 4.11: SGI Onyx overall performance.

Table 4.11 displays the initial performance obtained on the SGI. The two- and four-processor speedup values are 2.2 and 3.2, respectively. Recall that these results are for the overall code execution, and that a significant portion of the solution consists of serial code. Furthermore, the superlinear 2.2 speedup on two processors is noteworthy. Clearly, caching of the working set of data in each of the two processor caches is sufficient to overwhelm parallel overhead, contention, and any Amdahl's law effect for this case, and results in an improved cache "hit ratio" over the single-processor case. Appendix A further explains how caching may result in larger than expected performance increases when the cache-size and problem-size combine for optimal performance.

In an effort to further examine the performance degradation in the four-processor case, each parallel section of the code was examined in further detail, beginning with the Jacobian

Num. of Sub-domains	Serial Jacobian Time (sec.)	Parallel Jacobian Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	96	42	2.3	114
4×1	77	21	3.7	92

Table 4.12: SGI Onyx Jacobian performance.

formation routine (Table 4.12). For the Jacobian routine, superlinear speedup (2.3) was seen for the two-processor case, and a respectable speedup of 3.7 was seen for the four-processor trial.

Num. of Sub-domains	Serial TFQMR Time (sec.)	Parallel TFQMR Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	321	141	2.3	114
4×1	294	92	3.2	80

Table 4.13: SGI Onyx TFQMR performance.

Finally, because the Jacobian formation and TFQMR solution routines comprise the bulk of the execution time of the code, the TFQMR routine was further examined (Table 4.13). Unlike the Jacobian routine, much of the TFQMR routine remains serial (only the preconditioner formation, application, and matrix-vector multiply routines are parallel). Again, it

Num. of Sub-domains	Serial Precond. Time (sec.)	Parallel Precond. Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	94	42	2.2	112
4×1	30	8	3.8	94

Table 4.14: Speedup of the additive Schwarz preconditioner formation routine.

is noteworthy that even with this handicap, a parallel speedup of 2.3 was achieved in the TFQMR routine on two processors.

The superlinear parallel speedups in the parallel sections of the code using two processors result in an overall superlinear speedup for the simulation. However, this effect does not scale to four processors. This behavior may be due to several factors.

- Four processors may increase bus contention to the point that any superlinear effects disappear.
- Parallel overhead with a larger number of processors may overwhelm any improvements due to an increase in cache hit efficiency.
- For the problem size selected, the cache hit efficiency for the two-processor case may be near optimal. Adding another two processors (and caches) in this case may not improve the hit ratio sufficiently to overcome the additional overhead of two additional processors.

To eliminate the last item from consideration, one may scale the problem size to reduce the cache hit ratio for the two-processor case. As an initial attempt, the simulation grid was doubled in each direction, forming a 64×320 grid (the identical problem run on the Cray).

Mode	Total Time (sec.)	Jacobian Portion (sec.)	TFQMR Portion (sec.)	Precond. Portion (sec.)
Serial	3426	488	2918	341
Parallel	1212	109	1083	92
Speed-up	2.8	4.5	2.7	3.7

Table 4.15: 64×320 Onyx Run (4 Blocks).

Table 4.15 presents the execution data for this larger problem. The Jacobian routine

indeed exhibited a superlinear speedup on four processors. However, an overall superlinear improvement was not seen, and neither the TFQMR nor the preconditioner formation routines were overly efficient. Recall that the granularity in the Jacobian routine is near optimal. This is not the case for the TFQMR routine and the preconditioner formation routine. Further work to increase the granularity in these routines (and minimize the serial portion of the TFQMR routine) may result in the achievement of superlinear speedups in these routines and in the overall simulation for the four-processor runs. However, it is not clear that superlinear effects will scale beyond two (or four) processors without further data.

The apparent improvement in scalability of the SGI over the C90 architecture is primarily due to cache effects. However, this benefit could disappear as the number of processors are increased. Note that when comparing the 64×320 problem on four processors, the Cray is an order of magnitude faster.

Further work on the Jacobian algorithm is probably not warranted. On the 32×160 problem (4 block parallel), the Jacobian amounts to 18% of the total execution time. This decreases to 9% on the larger problem. The TFQMR routine amounts to 89% of the total execution time on the larger problem. Clearly, as problem size increases further, the Krylov algorithm will dominate. This data suggests that further efforts should be focused on the scalability of the Krylov solution. Given a solution to the increase in inner iterations that accompanies an increase in the number of subdomains, parallelization of the TFQMR routine may be warranted.

Finally, Table 4.16 shows the memory requirements for the smaller SGI run. A similar decrease in the memory requirements with an increase in the number of subdomains is evident. The requirements for the 64×320 SGI run are identical to the requirements presented for the Cray data (Table 4.2), because the problems are identical.

Num. of Subdomains	Memory Required (Mbytes)
2	33
4	17

Table 4.16: Memory requirements for SGI 32×160 simulation.

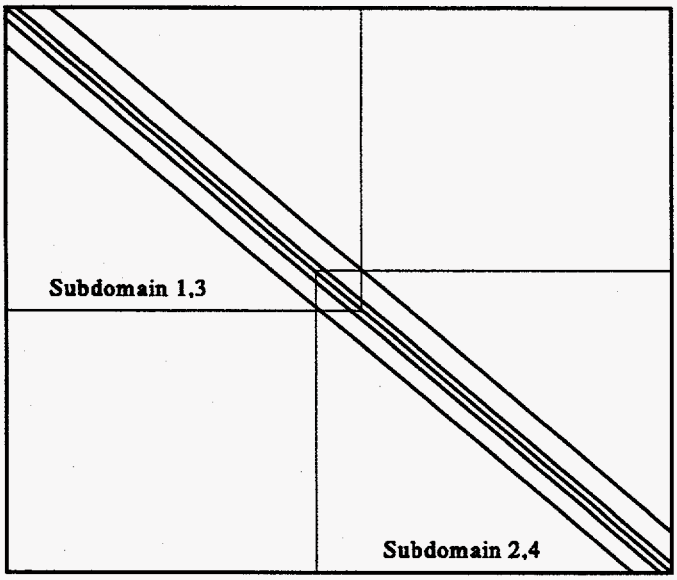

Figure 4.5: Partitioned Jacobian matrix, four subdomains with overlap.

4.4 Subdomain Overlap with Additive Schwarz

As an initial effort to investigate methods to mitigate the increase in Krylov iterations as the number of subdomains is increased, the additive Schwarz method employing subdomain overlap was examined. Recall that subdomain overlap includes a portion of the data from the Jacobian matrix adjacent to each of the subdomain regions in an attempt to increase the quality of the preconditioner by providing a better approximation to the Jacobian prior to the inversion of the subdomains (Figure 4.5).

Table 4.17 compares the effects of no subdomain overlap versus an 8 cell overlap (for each

Num. Subdomains	Avg. TFQMR Iterations	
	0 cell overlap	8 cell overlap
2	11	7
4	24	14
8	51	23
16	108	32

Table 4.17: 32×160 Onyx simulation iteration behavior comparing overlap values.

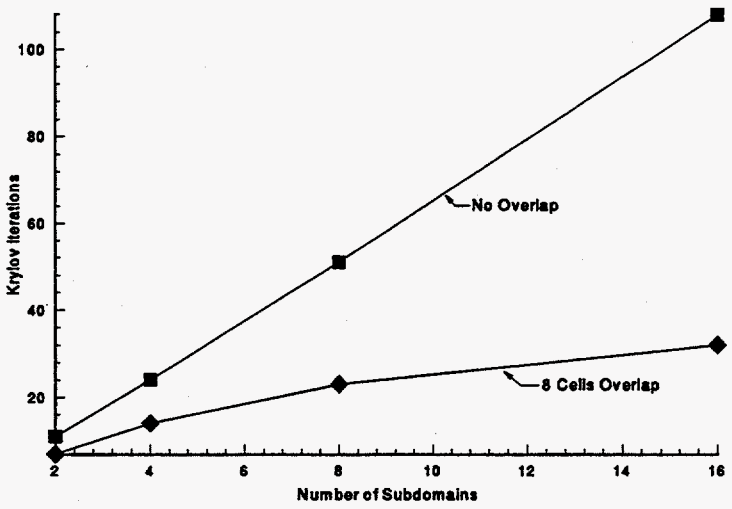

Figure 4.6: Plot of overlap behavior versus number of subdomains.

8 cell subdomain, the upper and lower 4 cells are solved as part of the neighboring domains) on a 32×160 run. Eight cells of subdomain overlap using additive Schwarz substantially decreases the number of TFQMR iterations for a given number of subdomains. Figure 4.6 is a plot of the data in Table 4.17. It is clear that there is a reduction in the slope of the increase over the zero overlap case. Furthermore, the slope of this line appears to decrease further as the number of subdomains is increased.

Table 4.18 illustrates the effect of overlap on parallel CPU time for a constant four subdomains on the 32×160 Onyx run. Recall that overlap requires serialization when the global

Overlap (cells)	Newton Iterations	Avg. TFQMR Iterations	Para. CPU Time
4	6	19	119
8	6	14	117
12	6	13	135
16	6	11	140

Table 4.18: Additive Schwarz, 4 domain case, showing effect of overlap on TFQMR iterations and CPU time.

preconditioner is assembled, resulting in the slightly poorer performance for all these runs (this result is clear when the 119 sec. 4 cell overlap run is compared to the 118 sec. run with no overlap). From the table, the minimum runtime is achieved with an 8 cell overlap. Overlap did not appear to significantly decrease the parallel CPU time. As overlap is increased beyond 8 cells, the CPU time increases as the solution on each subdomain approaches a direct solve of the entire Jacobian (from Figure 4.5, it is apparent that as each subdomain increases in size, it contains more of the Jacobian data).

Mode	Total Time (sec.)	Jacobian Portion (sec.)	TFQMR Portion (sec.)	Precond. Portion (sec.)
Serial	364	77	281	62
Parallel	117	21	91	18
Speed-up	3.1	3.7	3.1	3.4

Table 4.19: Speedup values for 8 cell overlap problem.

Finally, Table 4.19 illustrates the speedup figures for the above 8 cell overlap run using four subdomains. Contrasting this data with Tables 4.11, 4.12, 4.13, and 4.14, overlap appears to increase the serial performance of the TFQMR algorithm to a greater extent than for the parallel performance. The speedup in this routine drops from 3.2 to 3.1, which decreases the overall speedup from 3.2 to 3.1. Further, the speedup decreases from 3.8 to 3.4 in the Schwarz

preconditioner formation routine. Considering speedup values alone, it does not appear that overlap enhances the parallel scalability of the method, at least with this limited data.

Num. of Subdomains	Overlap (cells)	Memory Required (Mbytes)
2	0	33
4	0	17
2	4	46
4	4	32
2	8	61
4	8	48
2	12	78
4	12	72
2	16	97
4	16	98

Table 4.20: Memory requirements for SGI 32×160 simulation with various overlap values.

Table 4.20 presents the memory requirements for various levels of subdomain overlap on the 32×160 problem. Clearly, even small values of overlap significantly increase the memory requirements over the non-overlap case (Table 4.16). For example, on the 4 subdomain problem, eight cells of overlap increase the memory requirements from 17 Mbytes to 48 Mbytes. Furthermore, increasing overlap beyond a certain extent erases the advantage of a reduction in memory requirements with an increase in the number of subdomains. This is expected. Once the overlap value approaches the condition where the entire Jacobian is contained in each subdomain, the solution memory requirements will approach a value equal to the number of subdomains multiplied by the amount of memory required for a direct Jacobian inversion.

From this data, it appears that while subdomain overlap is effective in reducing the number of TFQMR iterations, it is only minimally effective in reducing runtime. However, it may be possible to compensate for the increased work performed in the subdomain solves

Overlap (cells)	Newton Iterations	Avg. TFQMR Iterations	Para. CPU Time
4	7	125	273
8	7	105	267
12	6	68	202
16	7	59	238

Table 4.21: Additive Schwarz, 16 domain case, showing effect of overlap on TFQMR iterations and CPU time for a 96×480 simulation.

by increasing the number of subdomains, provided the TFQMR iteration growth does not overwhelm any runtime improvements. Unfortunately, this effect cannot be studied on the Onyx due to the limit of four processors (the local machine had only a total of four processors installed). Furthermore, the memory demands of an overlap solution appear quite severe for even small values of overlap. Because many moderate to large simulations may be memory limited, employing overlap may not be an option. For the problem size considered, overlap does not provide any significant advantages on the four processor SGI architecture.

A 16 processor Cray C90 was used for the overlap study employing greater than four subdomains. Initially, a 16 processor parallel run was analyzed on the machine to obtain algorithm scalability data similar to that shown in Table 4.18 (see Table 4.21). Due to the superior performance of the Cray and the expected overhead in scaling to 16 processors, the problem discretization was increased to 96×480. Figure 4.7 shows that, for this problem size, subdomain overlap again decreases the number of TFQMR iterations. The decrease in required iterations is quite substantial. For example, a comparison of the two tables reveals that with 12 cells of overlap on the 16 block problem, 59 iterations are required versus 70 for the 8 block case with no overlap. Provided the increase of subdomain size with overlap may be mitigated by increasing the number of subdomains, it may be possible to achieve reasonable scalability for the overall solution.

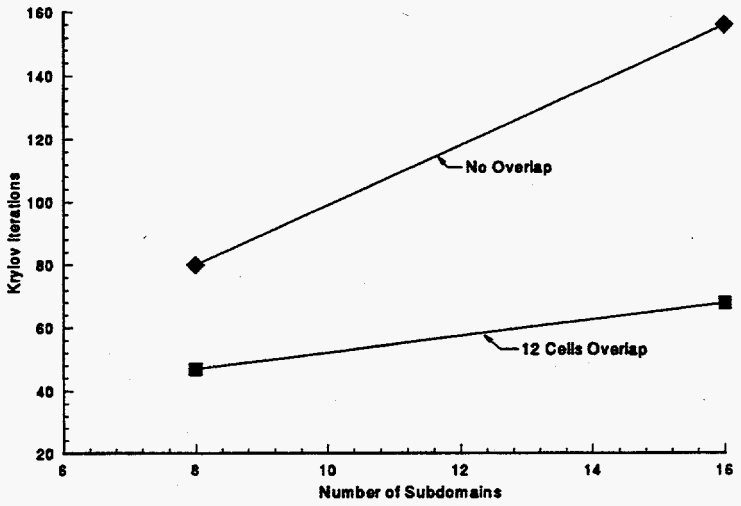

Figure 4.7: Plot of overlap behavior versus number of subdomains for Cray 96 × 480 simulation.

Mode	Total Time (sec.)	Jacobian Portion (sec.)	TFQMR Portion (sec.)	Precond. Portion (sec.)
Serial	888	217	645	231
Parallel	194	28	141	31
Speed-up	4.6	7.8	4.6	7.5

Table 4.22: Speedup values for 12 cell overlap, 96 × 480 problem using 8 processors.

Table 4.22 and Table 4.23 illustrate the actual runtime performance on 8 and 16 C90 processors, respectively. These tables show a noticeable performance gain over the earlier Cray data (at least in the Jacobian and preconditioner algorithms), however, several changes in the simulation were made between the two datasets.

- The parallel granularity in each of the parallel sections of the code was increased substantially by scaling the problem to 96 × 480 over the previous 64 × 320 simulation.

- Dynamic memory allocation was added to the code. Any arrays involved in the pre-

Mode	Total Time (sec.)	Jacobian Portion (sec.)	TFQMR Portion (sec.)	Precond. Portion (sec.)
Serial	993	186	777	216
Parallel	202	13	159	16
Speed-up	4.9	14.3	4.9	13.5

Table 4.23: Speedup values for 12 cell overlap, 96×480 problem using 16 processors.

conditioner were removed from the common data structure and dynamically allocated via `malloc()` in the main program.

- These results employ subdomain overlap and attempt to mitigate the increase in subdomain complexity by employing more subdomains, thus increasing the degree of parallelism of the preconditioner and Jacobian routines.

Clearly, these changes significantly enhance the performance of both the Jacobian and preconditioner routines. In fact, these two algorithms scale quite nicely, at least to 16 processors. The overall simulation performance does not reflect this increased efficiency, however. For example, a speedup of only 4.9 was achieved on 16 processors. Also noteworthy is the fact that the eight-processor run provided a minimum runtime for this problem (194 sec. versus 202 sec. for the 16 processor-case). From Table 4.23, the serial portion of the TFQMR time dominates the overall solution performance (159 sec. - 16 sec. = 143 sec. of the total 202 sec. runtime). The overall code performance and overall scalability are not likely to improve further without increasing the parallelism in the TFQMR routine.

To examine the potential of a parallel TFQMR routine with this problem, assume a speedup of 14 could be attained. This figure results in an overall runtime of approximately 98.5 sec. on 16 processors, all else being equal. A similar exercise on the eight-processor solution results in 139 sec. for a TFQMR speedup of 7.5. For this scenario, doubling the

number of processors results in a factor of 1.4 decrease in runtime. From this analysis, it is obvious that parallelization of the TFQMR routine will not likely provide true overall solution scalability from eight to 16 processors. The 16-processor result would, however, result in the minimal execution time seen thus far on this problem.

From these results, the benefits of subdomain overlap are apparent. The number of TFQMR iterations are drastically reduced using this technique. The reduction appears to allow an increase in subdomains to partially compensate for the additional work required in each subdomain inversion, leading to an increased degree of parallelism. The combination of overlap, a larger simulation, and a new memory structure clearly resulted in scalability of the Jacobian and preconditioner algorithms on the Cray architecture. However, even assuming that the TFQMR algorithm could be parallelized with the efficiencies denoted above, overall scalability cannot not be achieved due to the remaining increase in TFQMR iterations between eight and 16 subdomains.

Problem Size	Overlap (Mbytes)	No-overlap (Mbytes)
96×480	1639	122

Table 4.24: Memory requirements of 12 cell subdomain overlap on 96×480 16 domain problem.

There is one large drawback to an overlap scheme; the greatly increased memory requirements. Table 4.24 indicates the memory requirements of the above problem, both with and without overlap. The order of magnitude increase in memory requirements for this problem is clearly an unacceptable tradeoff for solution scalability. This example problem (96×480) is still quite small from a simulation standpoint, yet 1.6 Gbytes of memory are required for a scalable solution. Very few circumstances can be envisioned where these extreme re-

quirements could be accommodated on available hardware for a realistic three-dimensional simulation.

4.5 Summary

This chapter on the additive Schwarz preconditioner began with an overview of the mapping of a code employing the additive Schwarz scheme onto the two architectures of interest; a Cray C90 and SGI Challenge multiprocessor. Following this discussion, results were compared with a baseline HP platform to illustrate that significant performance increases were possible with the use of a parallel supercomputer. Additionally, the performance of the base code was analyzed to focus the parallel (and vector in the case of the C90) optimization effort.

It was discovered that the Jacobian and preconditioner routines indeed warranted further study. Furthermore, the preconditioner and Krylov iterations appeared to dominate the computation time as the problem size was increased. For small problems, the C90 did not provide outstanding results due to low parallel granularity in the parallel sections of the code and problems with memory contention that could not be easily overcome without significant code changes. Additionally, as the number of subdomains (processors) were increased, the increase in Krylov iterations due to a reduction in preconditioner effectiveness quickly limited the overall efficiency of the code.

The Onyx machine appeared to perform significantly better on the smaller problems due to cache effects. However, this could not be verified beyond four processors on the machine available for this study. Reasonable scalability of the algorithms was seen on both architectures if the number of processors was limited to four. This result should be quite encouraging to users that are limited to inexpensive hardware or provided with limited resources. It is

indeed possible to achieve worthwhile reductions in simulation runtime on workstation class shared-memory multiprocessors using the additive Schwarz preconditioner.

For those applications demanding scalability beyond four processors, the increase in Krylov iterations driven by preconditioner degradation was addressed. Several options were discussed and subdomain overlap was studied in detail. For smaller problems on the Onyx, overlap significantly reduced preconditioner degradation as the number of subdomains was increased. However, due to the increase in the number of subdomain operations, the runtime of the simulation was not improved significantly. A hypothesis was suggested concluding that it may be possible to increase the number of subdomains (and processors) to mitigate the increase in operations, allowing overlap to check the increase in Krylov iterations. A larger study employing a 16 processor Cray was performed to examine this possibility. Additionally, the memory structures in the code were re-worked and dynamic memory allocation was employed for convenience. This larger study confirmed that scalability of the Jacobian and preconditioner algorithms had been achieved. However, an overall scalability was not observed, and it was argued that even with a parallel Krylov solve, true overall scalability would remain elusive. This success was further diminished due to the extreme memory requirements of the overlapped subdomains and the necessity of parallelizing the Krylov algorithm to approach scalability of the overall solution procedure.

With these results, it appears unlikely that a simple Newton-Krylov procedure using additive Schwarz preconditioning as outlined in this study will result in a truly scalable parallel solution algorithm within reasonable memory bounds. Of the options presented previously, neglecting the use of a coarse grid operator (a possible but premature technique to reduce the Krylov iteration increase with the number of subdomains), the following two topics remain:

1. multiplicative Schwarz preconditioning, and

2. matrix-free techniques employing pseudo-transient relaxation.

The following chapters discuss these two methods in detail.

Chapter 5

The Multiplicative Schwarz Preconditioner

The multiplicative Schwarz preconditioner was examined as a possible remedy for the increase in Krylov iterations due to a degradation in the global preconditioner as the number of subdomains in the simulation is increased. Previous work by McHugh [1] suggests that the multiplicative Schwarz preconditioning technique results in a higher-quality global preconditioner than the additive method in base, non-overlapping form. Additionally, the overlap version of the technique also appears superior to the additive algorithm, at least for the values of cell overlap studied. However, the multiplicative algorithm results in identical memory requirements to the additive routine for comparable overlap values; thus overlap will again prove intractable due to its memory requirements.

The implementation of a parallel multiplicative Schwarz method followed the additive Schwarz process detailed in the preceding chapter. In fact, the simulation code was structured such that one may select the desired preconditioner via an input value. As such, the code

is largely unchanged from the version used to obtain the previous results; only the routines specific to the differences between the additive and multiplicative algorithms are distinct.

The major structural difference between the two algorithms is the degree of parallelism inherent in the preconditioner. Data dependencies in the base multiplicative Schwarz algorithm prohibit the direct parallelization that was employed in the additive form. Recall that $\mathbf{w} = \mathbf{P}_l^{-1}\mathbf{v}$ is computed by

$$\begin{aligned} \mathbf{v}_1 &= \mathbf{J}_1^{-1}\mathbf{v} \\ \mathbf{v}_j &= \mathbf{v}_{j-1} + \mathbf{J}_j^{-1}(\mathbf{v} - \mathbf{J}\mathbf{v}_{j-1}), \qquad \text{for } j = 2, \ldots, n \\ \mathbf{w} &= \mathbf{v}_n, \end{aligned} \tag{5.1}$$

and n corresponds to the total number of subdomains employed in the preconditioner (see Section 3.3.2). It is clear that the factor $\mathbf{v}_j$ is dependent on the formation of the term $\mathbf{v}_{j-1}$. If a coloring scheme

$$\mathit{Color}(w_i) = \min\{k > 0 \mid k \neq \mathit{Color}(w_j), \forall w_j \in \mathrm{Adj}(w_i)\}, \tag{5.2}$$

is employed to renumber the subdomains in the solution, it is possible to arrange the preconditioner algorithm such that the values of $\mathbf{v}_{j-1}$ are either $\mathbf{v}_1$, or can be obtained from a previous parallel calculation. This mathematical explanation can be easily visualized by considering a stripwise domain decomposition (as was used in this study). Figure 5.1 illustrates a stripwise, 4 subdomain problem (4×1) where the subdomains are colored in a red-black scheme. This figure also shows an 8 subdomain problem in a "checkerboard" decomposition (4×4), colored red-black-green-blue (RBGb).

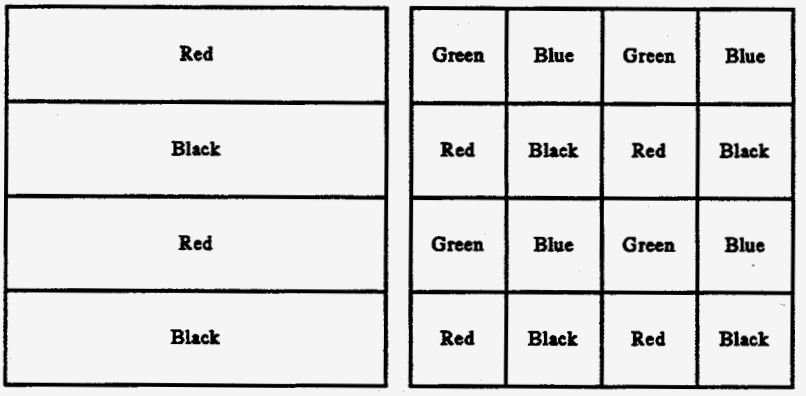

Figure 5.1: Red-black coloring on stripwise, RBGb coloring on "checkerboard" decomposition.

In the stripwise scheme, the two red subdomains may be calculated immediately and in parallel, because $\mathbf{v}_{j-1} = \mathbf{v}_1$. Following this operation, a synchronization step is needed followed by a parallel calculation of all the black subdomains ($\mathbf{v}_{j-1}$ for each black subdomain is now known from the results of the previous red subdomain calculation). This concept (slightly extended) also holds for the two-dimensional blocking in the "checkerboard" decomposition. However, from the discussion in Section 3.3.2, the degree of parallelism is now less than that observed with the additive Schwarz method. A stripwise decomposed multiplicative Schwarz algorithm using two colors has half the available parallelism

$$\text{DOP} = \frac{\text{number of blocks}}{2}. \tag{5.3}$$

This loss of parallelism is a concern. For the stripwise case, twice as many preconditioner blocks would be required for performance similar to that of the additive algorithm. Thus, for the multiplicative algorithm to be useful for this study, it must provide fewer Krylov iterations and a reduction in the increase of iterations with subdomains at a two-for-one disadvantage to the additive algorithm.

Num. Subdomains	Newton Iterations	AS Avg. TFQMR Iterations	MS Avg. TFQMR Iterations
2	7	11	5
4	6	24	13
8	7	51	23
16	6	106	53

Table 5.1: 32×160 Onyx simulation iteration behavior comparing additive Schwarz (AS) and multiplicative Schwarz (MS) preconditioning.

Num. Processors	AS Avg. TFQMR Iterations	MS Avg. TFQMR Iterations
2	11	13
4	24	23
8	51	53

Table 5.2: 32×160 iteration behavior comparing additive Schwarz (AS) and multiplicative Schwarz (MS) preconditioning on the basis of DOP.

5.1 Results

The investigation of the multiplicative Schwarz preconditioner was restricted to the Onyx platform and used identical model problems to those previously considered to allow for correlation of the results to the additive Schwarz data. Table 5.1 contrasts the iteration behavior of the two algorithms using a 32×160 domain with stripwise decomposition. The multiplicative results employed red-black coloring. Clearly, multiplicative Schwarz preconditioning reduces the number of Krylov iterations required for a given number of subdomains for this problem size.

Table 5.2 provides the same information, but normalized on the basis of the degree of parallelism of the two algorithms (the number of processors that may be effectively employed for each of the algorithms). This result indicates that, given a constant operation count

Num. of Sub-domains	Serial CPU Time (sec.)	Parallel CPU Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	318	240	1.3	66
4×1	315	147	2.1	54
$8 \times 1^*$	514	162	3.2	79

Table 5.3: Overall code performance for 32×160 stripwise problem on 4 processor Onyx (* 8 block run on 4 processors)

between the two algorithms, multiplicative Schwarz does not provide any advantages of decreasing Krylov iteration count when based on the available parallelism. At this point, it appears doubtful that the multiplicative algorithm will provide any benefit over the additive version unless the parallel performance is markedly improved (*i.e.*, the multiplicative algorithm results in a significantly lower operation count).

Table 5.3 illustrates the performance of the colored multiplicative Schwarz preconditioner based on the overall solution time. Since the Onyx was equipped with only four processors, the 8×1 run employed eight subdomains in the preconditioner to allow full effectiveness (recall, with this algorithm eight subdomains are required to achieve a DOP of four within the preconditioner). The two other runs, 2×1 and 4×1, employed two and four processors, respectively. Contrasting this result with the additive Schwarz data, it appears that the multiplicative algorithm displayed better overall serial performance (executed on a single processor) than the additive method, but worse parallel performance (serial; 315 sec. versus 375 sec. for the 4 block case, parallel; 147 sec. versus 118 sec.). Table 5.4 reveals slight differences in the performance of the Jacobian algorithm, which is likely due to different cache utilization. Finally, Table 5.5 shows that the performance of the TFQMR algorithm is superior in regard to serial time, but inferior in regard to parallel time.

Num. of Sub-domains	Serial Jacobian Time (sec.)	Parallel Jacobian Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	79	34	2.3	116
4×1	66	21	3.1	79
$8 \times 1^*$	82	24	3.4	85

Table 5.4: Speedup in the Jacobian routine (* 8 block run on 4 processors).

Num. of Sub-domains	Serial TFQMR Time (sec.)	Parallel TFQMR Time (sec.)	Parallel Speedup	Parallel Efficiency (%)
2×1	234	195	1.2	60
4×1	245	122	2.0	50
$8 \times 1^*$	427	134	3.2	80

Table 5.5: Speedup of the TFQMR routine (* 8 block run on 4 processors).

Clearly, these results paint a bleak picture for the multiplicative Schwarz algorithm. The overall performance results in conjunction with the TFQMR results indicate that the multiplicative algorithm has appreciably more overhead (in parallel execution) than the additive routine. Due to data dependence constraints, colored multiplicative Schwarz must serialize between parallel solves of colors. This synchronization step not only adds operations to the algorithm, it also decreases the parallel granularity of the algorithm by one-half. This result, coupled with (1) the loss of degree of parallelism, and (2) comparable Krylov iteration behavior, demonstrates that additive Schwarz is superior to multiplicative Schwarz for problems such as those examined in this study.

For completeness, a simplified overlap study was performed using the multiplicative Schwarz algorithm in a similar manner to the earlier additive results. Table 5.6 presents these results.

Overlap (blks–cells)	Newton Iterations	Avg. TFQMR Iterations	Para. CPU Time
4–0	6	13	147
4–2	6	10	138
4–4	6	9	140
8–0	6	23	162
8–2	6	19	138
8–4	6	19	162

Table 5.6: Iteration behavior with multiplicative Schwarz preconditioning.

As with the additive Schwarz study, subdomain overlap decreased the number of TFQMR iterations as the overlap was increased. Two cells of overlap significantly decreased the parallel CPU time for both the four and eight subdomain cases (particularly in the 8 block case). Again, as the overlap was increased beyond 2 cells, the CPU time increased. Using two cells of overlap and comparing the four domain case (only two processors are effectively used) to the eight domain case (all four processors used effectively), the increased Krylov iteration count in conjunction with the higher overhead and lower granularity completely neutralized the addition of the two processors. For this problem, the minimum runtime was obtained with the additive Schwarz algorithm.

5.2 Summary

The multiplicative Schwarz preconditioner does not appear to provide any advantages to the additive Schwarz method examined previously. In fact, several disadvantages exist that suggest that the additive method is more practical as a basis for a scalable solution procedure. The multiplicative results indicate that any decrease in the Krylov iterations are offset by a lower degree of parallelism when solutions comparing the same number of subdomains are considered. If the number of subdomains are increased to provide a similar degree of

parallelism, the multiplicative algorithm does not provide any decrease in the number of Krylov iterations over the additive method (for the model problem examined). Additionally, the synchronization overhead and reduced parallel granularity inherent in the multiplicative algorithm suggests that it would be impossible to approach the parallel results of the additive technique with the operation count of the two algorithms held constant. Subdomain overlap does not provide any benefits that would mitigate these disadvantages, and suffers from the same extreme memory requirements seen with the additive algorithm with overlap.

Chapter 6

The Matrix-Free Technique

The matrix-free technique is the third, and last algorithm considered as a strategy to obtain a scalable algorithm for the solution of the model problem under study. Unlike the previous two methods, this technique does not directly affect the preconditioning operation (the additive and multiplicative Schwarz algorithms form the preconditioner). In the ideal case, where preconditioning is not required, the matrix-free technique could be used to eliminate the necessity of forming the Jacobian to obtain a solution to the linear system.

The Jacobian matrix appears in the form of matrix-vector products ($\mathbf{Jw}$) in most Krylov projection methods (including the TFQMR and GMRES algorithms). As such, in the ideal case, these methods do not require the formation of the complete Jacobian matrix. Only the product

$$\mathbf{Jw} \approx \frac{\mathbf{F}(\mathbf{x} + \epsilon\mathbf{w}) - \mathbf{F}(\mathbf{x})}{\epsilon}, \tag{6.1}$$

appears in the linear solution algorithms. The need for preconditioning, however, complicates the scheme greatly as the Jacobian is required for the development of the preconditioner. In the previous solution results, a new Jacobian and preconditioner were formed for each

Newton iteration. Using this preconditioner, the linear (Krylov) solve iterates to obtain an approximate solution to the system. If the Newton solution results in the need for an appreciably different preconditioner each Newton iteration, the only option is to form one. However, if the preconditioner only changes slightly between Newton iterations, it may be feasible to form the preconditioner every n iterations, using the latest existing preconditioner for those iterations that fall between preconditioner formation steps. In effect, a "stale" preconditioner is used for several iterations, greatly reducing the impact of preconditioner formation operations on the overall solution behavior.

It is not sufficient to simply amortize the preconditioner over several Newton iterations using the base Krylov algorithms. These algorithms rely on matrix-vector products based on the Jacobian used to form the preconditioner to solve the linear system. If the preconditioner (and Jacobian) formation is lagged, the Krylov algorithms will also use a stale Jacobian to solve the linear system. This practice may significantly compromise the convergence behavior of the overall technique (likely much more so than by the use of stale preconditioning alone). To eliminate the need to use the stale Jacobian within the Krylov algorithm, the matrix-vector product terms in these algorithms may be replaced with an equivalent expression (Equation 6.1) that includes the convergence contribution of previous Krylov iterates. Replacement of the matrix-vector products with this expression results in a matrix-free Krylov algorithm that removes the requirement for the Jacobian in the solution procedure. Thus, the Jacobian is only required for the preconditioner, not for the solution of the linear system. The matrix-free technique may also reduce total operation count, and is largely parallel. Given a suitable implementation of the technique, it may also be possible to increase the degree of parallelism contained in the Krylov solution algorithm.

The Jacobian-based Krylov routine forms the preconditioner at the beginning of the first

iteration. The remainder of the routine (at least in the cases of TFQMR and GMRES) consists of small sections of code that are amenable to parallelization. Abstractly, these sections are analogous to *basic blocks* in conventional terms, however in this case a basic block is defined as the largest sequence of statements that may be encapsulated into a parallel region. To further this analogy, these basic blocks are usually separated from one another by a length of serial code or a synchronization operation (often a barrier or mutex section). Furthermore, the degree of parallelism of these basic blocks is often quite high, perhaps even as large as a measure of the Jacobian. As an example, recall the parallelization of the matrix-vector product routine performed on the SGI and described in the earlier discussion of the results for the additive Schwarz preconditioner. Unfortunately, the parallel granularity of the other basic blocks is relatively small. On any parallel architecture, the granularity of each parallel region should be kept as high as possible to reduce the influence of serialization before and after the basic block. This is of utmost importance on both the Cray and SGI machines. The overhead of entry and exit from parallel regions, barriers, and other serialization techniques is appreciable. To study the feasibility of parallelization of the Krylov iteration, parallel execution of the basic blocks (beyond the existing concurrent matrix-vector product routine) within the TFQMR algorithm was attempted on the SGI Onyx. This effort was abandoned when it became obvious that, with the exception of the matrix-vector product routine, granularity in the remaining basic blocks was not sufficient to yield an appreciable performance increase on the Onyx. A similar task was not attempted on the C90, however experience with the additive Schwarz work performed earlier suggested that, due to the speed of each processor without a corresponding decrease in overhead, this large granularity requirement would be more significant than on the Onyx. Due to these results, it appears that achieving a useful performance increase within the Krylov routine through parallelism (on these architectures)

would require a significant re-work of the algorithm.

The matrix-free routine (*mfmatmul*) replaces the existing matrix-vector product algorithm, and has a similar degree of parallelism. However, the granularity of this routine is much larger (more operations are performed within *mfmatmul* than the matrix-vector product routine). Furthermore, the structure of *mfmatmul* is quite similar to the Jacobian formation algorithm, making it possible to capitalize on the knowledge obtained from parallelization of the Jacobian. Finally, in a given simulation, if a sufficient number of Jacobian and preconditioner formation steps (with the attendant lower degree of parallelism caused by preconditioner degradation) are replaced by matrix-free operations, it may be possible to achieve a solution with a net savings in operation count (recall, the preconditioner is roughly $O(n^3)$ and the Jacobian and matrix-free algorithms are roughly $O(n^2)$). Additionally, due to the high granularity and degree of parallelism within *mfmatmul*, these added matrix-free steps have the potential for efficient parallel execution on the architectures of interest.

Table 6.1 shows some initial results of the matrix-free algorithm on an Onyx model problem. A restarted version of the GMRES Krylov algorithm (GMRES(k)) was used in favor of the TFQMR method used previously, because this algorithm appears to provide better convergence behavior when the matrix-free technique is used. This 32×160 result used a stripwise decomposition employing four subdomains and processors on the SGI Onyx. The matrix-free algorithm formed a new Jacobian and additive Schwarz preconditioner only on the first and second Newton iterations. The remaining iterations used a stale form of the second preconditioner.

From these results, it is clear that the matrix-free technique shows promise as a solution method. This scheme resulted in the minimum runtime achieved thus far in the study (93 sec., 21% better than the best additive Schwarz results at 117 sec.)

CPU Time	Serial	Parallel	Speed-up
Total (sec.)	258	93	2.8
Jacobian (sec.)	26	6	4.5
GMRES (sec.)	227	82	2.8
Precond (sec.)	12	3	3.5
mfmatmul (sec.)	104	31	3.3

Table 6.1: Matrix-free results using "stale" additive Schwarz preconditioning on a 4 subdomain, 32×160 problem on the Onyx.

6.1 Robustness Concerns

The previous results on the matrix-free technique are based on a specialized, simplistic problem. Generally, a true Newton iteration path to a steady-state solution results in appreciable preconditioner changes each Newton iteration; the use of a stale preconditioner in the manner shown above would not generally provide a converged solution. Additionally, it may not be possible to obtain a starting point (or initial "guess") for the Newton algorithm that lies within the radius of convergence of Newton's method. This latter difficulty is not unique to the matrix-free method, the previous results shown for the additive and multiplicative Schwarz algorithms are likewise highly dependent on the initial "guess."

A technique labeled "pseudo-transient relaxation" (see Section 3.6) may be employed to increase the diagonal dominance of the system, which effectively increases the Newton radius of convergence. Additionally, this technique also provides user control of the "damping" of the Newton updates. This capability allows the change in the preconditioner each Newton iteration to be reduced to better accommodate lagging of the preconditioner. Through this mechanism, the "time step" may be selected to vary the number of Newton and Krylov iterations required to obtain a solution. In effect, for a particular subdomain strategy, the number of Krylov iterations per Newton iteration may be directly controlled. Independent

of the number of subdomains employed, the number of Krylov iterations may be bounded below some arbitrary value δ by the appropriate selection of the time step. This scheme has the drawback of increasing the number of Newton iterations required for solution, in essence "shifting" the work required for a solution from the Krylov algorithms into the Newton algorithm. This will negatively affect the quadratic convergence of the Newton algorithm seen in the previous results.

The results beyond this point should not be expected to correlate and, aside from scalability, should not be compared to the previous work. The pseudo-transient relaxation scheme cannot always effectively compete with a direct Newton-Krylov steady-state solution on a problem where the direct method results in an efficient solution. However, the technique is generally applicable, while the direct steady-state solution is not. Direct solutions may only be achieved for simple, specialized problems. Additionally, without overlap or a coarse grid operator, scalability of the direct methods cannot be achieved, as evidenced in the previous two chapters.

6.2 Performance of the Matrix-Free Technique

Due to the excellent performance seen on the Cray C90 with the overlapped additive Schwarz method, the initial examination of the performance of the matrix-free technique was limited to this architecture. The Cray study commenced with an examination of a relatively small problem, 32×160, with general parameters as listed in Table 6.2. This set of data differs in many ways from the earlier steady-state results in that a pseudo-transient technique was used at a time step size of 1.0. Furthermore, a potential advantage of the matrix free technique was used; the Jacobian and preconditioner were formed every five Newton iterations instead

Problem Description	
Problem Size	32×160
Jacobian Updated	5 Newton Iter.
Pseudo Time Step Size	1.0
Inlet Mach Number	1.0×10^{-1}
Residual Tolerance	1.0×10^{-6}

Table 6.2: Parameters for the Cray 32×160 runs.

Num. Subdomains	Newton Iterations	Avg. GMRES(k) Iterations
4	177	12
8	177	21
16	177	51

Table 6.3: 32×160 matrix-free simulation iteration behavior on the Cray ($1 \times n$ stripwise blocking).

of for each iteration, as was previously required. Finally, the inlet Mach number was relaxed to 0.1 for expediency.

Table 6.3 illustrates the Newton and Krylov iteration behavior for this new problem as a function of the number of subdomains used in the preconditioner (with $1 \times n$ stripwise blocking). Again, this data suggests a decrease in preconditioner effectiveness as seen earlier, and is evidenced by the increase in Krylov iterations with the number of subdomains.

Table 6.4 shows the performance data of this problem with the $1 \times n$ blocking, on four, eight, and 16 processors, respectively. Note the use of eight processors with only four preconditioner blocks. For the matrix-free technique as implemented, possibly contrary to one's initial impression, the use of more processors than preconditioner subdomains may be justified and results in performance advantages. Recall that the degree of parallelism for the additive Schwarz preconditioner formation and application routines is identical to the number of distinct subdomains used to construct the preconditioner. The Jacobian possesses a

Mode	Total Time (sec.)	Jacobian Portion (sec.)	GMRES(k) Portion (sec.)	Precond. Portion (sec.)	mfmatmul Portion (sec.)
	1×4 blocking - Mem. req. 65 Mbytes				
Serial	1120	157	960	187	613
4 proc/Speed-up	383/2.9	43/3.7	337/2.8	59/3.2	193/3.2
8 proc/Speed-up	504/2.2	21/7.5	480/2.0	59/3.2	365/1.7
	1×8 blocking - Mem. req. 65 Mbytes				
Serial	1495	157	1335	106	1025
8 proc/Speed-up	291/5.1	22/7.1	266/5.0	15/7.1	183/5.6
	1×16 blocking - Mem. req. 65 Mbytes				
Serial	2940	157	2780	76	2334
16 proc/Speed-up	498/5.9	21/7.5	475/5.9	9/8.4	354/6.6

Table 6.4: Speedup values for 32×160 problem.

higher degree of parallelism, usually equal to the number of discretization cells along the x or y axis of the problem (x or y due to the nested loop structure of the Jacobian routine that consists of an indexed traversal over the two-dimensional domain). The matrix-free routine is structured quite similarly to the Jacobian formation routine. As such it has a similar degree of parallelism.

Previously, only the Jacobian and preconditioner routines were parallelized (the SGI results included parallelization of the matrix-vector product routine also). It was argued that as the preconditioner came to dominate execution time with growing problem size (the preconditioner requires roughly $O(n^3)$ operations versus roughly $O(n^2)$ for the Jacobian), the parallel performance of the code would approach the degree of parallelism of the preconditioner (with infinite problem size). Clearly, if this is the case, the preconditioner limits the scalability of the simulation code. The sole addition of the matrix-free technique does not change this behavior, because the number of operations is polynomially less than for the preconditioner (similar to the Jacobian). However, as discussed previously, it may be possible

to forgo the Jacobian and preconditioner formation each and every Newton iteration. This concept suggests that under certain conditions it is conceivable that the number of preconditioner formations may be reduced to the point that this routine no longer dominates the simulation.

Returning to the data, the four-processor run resulted in the minimal runtime and the greatest parallel speedup overall (2.9). The use of four additional processors (again, only used to full effect in the Jacobian and matrix-free routines) *slowed* execution time. Considering the Jacobian time alone, a respectable speedup was obtained when moving from four to eight processors. The time spent on the preconditioner did not change (the degree of parallelism did not change and only four processors were used to full effect). However, the time spent in the GMRES routine (recall that *mfmatmul* is contained within GMRES) increased substantially with the increase in processors. Because the *mfmatmul* algorithm is the only parallel section contained within GMRES, it is quite likely that this algorithm alone is responsible for the decrease in efficiency inside GMRES. Finally, as the *mfmatmul* algorithm has a degree of parallelism similar to the Jacobian routine, this efficiency decrease is surprising.

The lower sections of Table 6.4 illustrate the performance on this problem using 8 and 16 block preconditioners on 8 and 16 processors. Again, the serial time increases due to the degradation of the preconditioner requiring more Krylov iterations as the number of subdomains increases. As the number of Krylov iterations is increased, the number of calls to *mfmatmul* increases, translating to increased time in this routine. The decrease in the total preconditioner time is not unexpected due to the decrease in the size of each subdomain with increasing decomposition. Because the number of Newton iterations is independent of the number of subdomains, the preconditioner routine is called a constant 177 times for each blocking strategy.

Problem Description	
Problem Size	64×320
Jacobian Updated	10 Newton Iter.
Pseudo Time Step Size	0.5
Inlet Mach Number	1.0×10^{-1}
Residual Tolerance	1.0×10^{-4}

Table 6.5: Parameters for the Cray 64×320 runs.

From the parallel results on this problem, it is evident that the Jacobian and preconditioner routines scale quite well to eight processors, but not beyond. The *mfmatmul* routine offers a reasonable performance boost for up to eight processors (a speedup of 5.6). Because the GMRES portion consists of the preconditioner time plus the *mfmatmul* time and some serial time, the overall GMRES time reflects a bit lower performance than *mfmatmul* alone. However, *mfmatmul* appears to dominate this algorithm at this problem size.

Table 6.4 indicates that further work in mapping the *mfmatmul* algorithm to the Cray should be performed. The degree of parallelism of *mfmatmul* is limited only by the problem size. Thus, it is quite likely that a memory contention problem is negatively affecting the results, especially for 16 processors. The eight-processor, four-subdomain case clearly indicates a mapping problem within the algorithm that appears to be strongly influenced by the number of subdomains within the preconditioner.

To summarize this smaller problem, the eight-processor parallel run using 1×8 preconditioner blocking resulted in the minimal solution time. Furthermore, this particular dataset indicates that a level of scalability is achieved for up to eight processors, but not beyond.

Table 6.5 specifies the input parameters to a similar 64×320 C90 run using the matrix-free techniques. The iteration behavior of the larger problem is shown in Table 6.6. Note that distinctly fewer Newton iterations are required for this larger problem. This behavior

Num. Subdomains	Newton Iterations	Avg. GMRES(k) Iterations
4	92	16
8	92	30
16	92	76

Table 6.6: 64×320 matrix-free simulation iteration behavior ($n \times 1$ stripwise blocking).

is due to a "larger" convergence criterion for the overall solution, 1.0×10^{-4} for this larger problem in contrast to 1.0×10^{-6} for the smaller. The convergence criterion was modified not for algorithmic, mapping, or performance reasons, but to decrease the execution time of each of the simulations (the monthly Cray time available for this work was limited). Furthermore, the blocking was changed from $1 \times n$ to $n \times 1$ to reduce the memory requirements of the larger problem. These tolerance and blocking differences between datasets prevents detailed comparisons between the two problem sizes. However, some general comments may be made.

Mode	Total Time (sec.)	Jacobian Portion (sec.)	GMRES(k) Portion (sec.)	Precond. Portion (sec.)	mfmatmul Portion (sec.)
	4×1 blocking - Mem. req. 133 Mbytes				
Serial	2353	215	2128	103	1712
8 proc/Speed-up	1175/2.0	28/7.7	1137/1.9	27/3.8	1009/1.7
	8×1 blocking - Mem. req. 70 Mbytes				
Serial	3756	216	3530	79	3010
8 proc/Speed-up	615/6.1	31/7.0	574/6.1	10/7.9	440/6.8
	16×1 blocking - Mem. req. 38 Mbytes				
Serial	8586	216	8360	66	7300
16 proc/Speed-up	1781/4.8	38/5.7	1734/4.8	10/6.6	1388/5.3

Table 6.7: Speedup values for 64×320 problem.

Table 6.7 shows the execution data for this larger problem. With the larger problem, the trends and subsequent interpretation are quite similar to the smaller problem. Again, the eight-processor, eight-block case resulted in a minimal execution time. The scalability

to eight processors on this larger-grained problem improved somewhat (a speedup of 6.1 was achieved on eight processors).

From the above data, it is apparent that the matrix-free technique in conjunction with pseudo-transient relaxation results in a better performing, more robust solution method than the direct method.

1. The pseudo-transient technique increases the Newton sphere of convergence, allowing the use of the Newton-Krylov solution on a wide variety of challenging problems.

2. The matrix-free method may eliminate the need to form the Jacobian and preconditioner each Newton iteration. Judicious use of this capability has the potential to reduce the influence of the preconditioner and Jacobian formation on the execution time of the overall solution.

3. The use of matrix-free techniques in conjunction with pseudo-transient relaxation allows "tuning" of the number of Krylov iterations per Newton step, at the expense of increasing the total number of Newton iterations and the loss of superlinear convergence.

4. Overall solution scalability has clearly improved; the 64×320 problem using 8 blocks and 8 processors yielded a speedup of 6.1, the best results to date. Scalability to 16 blocks on 16 processors, however, appears unlikely.

With these points in mind, the matrix-free solution performs quite well on up to eight processors. At this point, considering the above data and the scalability difficulties that were observed with the additive Schwarz preconditioner, efforts to enhance the scalability of this method should be directed along two paths: (1) mapping of the *mfmatmul* algorithm to the hardware and (2) examination of the preconditioner scalability issue.

The *mfmatmul* algorithm has a high degree of parallelism. This routine should scale well beyond the current eight-processor limit. Again, this is a mapping problem of the algorithm to the hardware; the memory access patterns and granularity of the algorithm currently prevent scalability. A similar problem was addressed earlier in this study within the Jacobian construction routines. *mfmatmul* has a very similar character to the Jacobian construction routine but accesses the banked memory of the Cray in a very different manner. Due to the complexity of this algorithm, rework to effectively reduce this contention problem is non-trivial and will require a significant amount of further research.

The scalability of the additive Schwarz preconditioner (manifested by the increase in Krylov iterations with the number of subdomains) must also be addressed at some level. Three potential possibilities are apparent.

1. Use of the matrix-free technique in conjunction with pseudo-transient relaxation to reduce the number of preconditioner formation steps to a minimum. From the results, this is clearly a promising option. However, it is not clear if it will be effective on every problem. Furthermore, it is very likely that an optimal formation strategy exists (it may be quite difficult to compute, however). If the preconditioner is formed more often than necessary, the increased convergence rate of the overall solution does not offset the work required to form the preconditioner and the result is below optimal performance. If a new preconditioner is formed too seldom, the deterioration of the overall convergence rate will quickly overcome any savings of preconditioner formation time (if the solution converges at all). Realistically, an algorithm to compute the optimal preconditioner formation frequency will likely be strongly problem and method dependent. Currently, an attempt to approach this ideal amounts to an iterative optimization problem over

Problem Description	
Problem Size	32×160
Jacobian Updated	5 Newton Iter.
Pseudo Time Step Size	1.0
Inlet Mach Number	1.0×10^{-1}
Residual Tolerance	1.0×10^{-6}

Table 6.8: Parameters for the SGI 32×160 runs.

a multidimensional computational surface. Finally, it is certainly not clear that the influence of preconditioner formation can be reduced sufficiently to achieve scalability for a given (or arbitrary) problem.

2. As mentioned earlier, it is not necessary to capitalize on the available processors by selecting a preconditioner that uses an equivalent number of subdomains. In fact, one of the above data points illustrated the use of 8 processors on a 4 block preconditioned simulation (recall that the Jacobian and matrix-free algorithms can use all the available processors to full advantage, but during the preconditioner stage only a number of processors equal to the number of subdomains can be used). This attempt did not appear to provide any advantage for the single case considered, largely due to the poor performance of the *mfmatmul* algorithm. If this problem can be corrected, the reduced block concept has some potential (especially if it is combined with the first item, above).

3. Again, as mentioned in the direct solution results previously, the preconditioner degradation problem can be addressed directly with the use of a coarse grid/fine grid technique.

The performance of the Newton-Krylov additive Schwarz matrix-free techniques were next studied on the SGI Onyx. Table 6.8 lists the parameters for the 32×160 simulation. Note that the convergence tolerance has been returned to 1.0×10^{-6} and that the preconditioner

Num. Subdomains	Newton Iterations	Avg. GMRES(k) Iterations
2	177	10
4	177	18

Table 6.9: 32×160 matrix-free simulation iteration behavior ($n \times 1$ stripwise blocking).

Mode	Total Time (sec.)	Jacobian Portion (sec.)	GMRES(k) Portion (sec.)	Precond. Portion (sec.)	mfmatmul Portion (sec.)
	2×1 blocking - Mem. req. 33 Mbytes				
Serial	3646	467	3169	427	1094
2 proc/Speed-up	1872/1.9	195/2.4	1667/1.9	220/1.9	574/1.9
4 proc/Speed-up	1746/2.1	103/4.5	1633/1.9	299/1.4	312/3.5
	4×1 blocking - Mem. req. 17 Mbytes				
Serial	5044	651	4384	178	2508
4 proc/Speed-up	1392/3.6	180/3.6	1198/3.7	46/3.9	492/5.1

Table 6.10: Speedup values for 32×160 problem on the SGI.

is updated every 5 Newton iterations. In a similar vein to the Cray results, Table 6.9 reports the iteration behavior of the Newton-Krylov technique. The increase in Krylov iterations with the number of subdomains is again clearly evident.

Table 6.10 illustrates the solution execution times for variants of the model problem. In this case, 2×1 preconditioner blocking was examined using both two and four processors. Finally, a solution was achieved using a four block preconditioner with four processors employed. In this case, the four block, four-processor configuration minimized runtime at 1392 sec. (a speedup of 3.6). A major portion of this result is probably due to the superlinear cache performance exhibited in *mfmatmul* for this execution (a speedup of 5.1 on four processors). Similar to the direct additive Schwarz steady-state solution considered early in this study, these algorithms are scalable to four processors on this architecture.

Unlike the behavior exhibited by the Cray results, the two-block, four-processor results

indicated that it may be useful to consider this execution mode on cached architectures (recall though that the Cray results compared 4 blocks on 4 and 8 processors). However, this mode did not result in the minimal runtime nor maximal speedup. The Jacobian routine scaled well, and the *mfmatmul* algorithm did not exhibit the mapping problems to the degree seen on the C90. However, on this problem, the preconditioner time is a major portion of the overall runtime (also unlike the C90). The speedup reduction from 1.9 to 1.4 on this routine (likely due to cache behavior) was a factor in the overall performance; the *mfmatmul* speedup of 262 sec. was partially offset by the 79 sec. increase in preconditioner time, resulting in a GMRES improvement of only 34 sec. A meaningful comparison of this concept between the SGI and Cray is not possible with the available data. It is quite likely that an examination of this mode on a larger (in number of processors) SGI would result in the same behavior seen on the Cray. It is clear that this mixed block/processor mode would not currently be of interest in comparison with the four-block, four-processor results without further work on mapping the preconditioner and *mfmatmul* algorithms to the hardware under these conditions.

6.3 Summary

Recall that the matrix-free technique, in conjunction with pseudo-transient relaxation, was of interest to both increase the robustness of the overall solution technique and provide better overall solution scalability than the direct additive Schwarz solution examined previously. Pseudo-transient relaxation offers an increased sphere of Newton convergence (to provide a more general solution method) and allows user control of the Newton convergence behavior. The matrix-free method enables a linear solution without explicitly forming the Jacobian matrix. However, it was immediately obvious that the Jacobian was required to form the

preconditioner. It was also evident that it may not be necessary to form the preconditioner for each and every Newton iteration. In fact, depending on the problem, one can use a "stale" preconditioner judiciously to decrease the influence of preconditioner scalability on the solution algorithm.

The use of the matrix-free method resulted in scalable solutions on up to eight processors on the Cray C90, and four processors (all that were available) on the SGI Onyx. The Cray solution clearly did not scale to sixteen processors, both due to the increase in Krylov iterations with increasing number of subdomains and mapping problems (likely memory contention) of the *mfmatmul* routine to the C90 processors. It is clear from these results, however, that considering robustness, memory requirements, and scalability issues, the matrix-free technique as implemented is superior to the direct additive Schwarz solution described previously.

Chapter 7

Conclusions

This dissertation is logically divisible into two parts; a theoretical outline of the methods and model problem followed by results and discussion of the mapping of the employed techniques on the parallel machines studied. In this final chapter it is valuable to examine a summary of this study and how successfully the results addressed the goals of this research. Most importantly, it is necessary to examine the theoretical development and results with a critical eye towards the goals of the research, address any shortcomings, and suggest a path (or paths) for further research.

The introduction of this work provided an overview of the importance of the efficient solution of the Navier-Stokes equations to many industries and areas of science. Given this information, the process of obtaining a solution to these equations using discrete techniques was examined. It was argued that an implementation of an inexact Newton-Krylov-Schwarz technique could be a promising path for the solution of equations of this type. As stated in the introduction, the memory requirements and CPU time needed for the computation of Navier-Stokes discrete solutions limit the utility of the techniques (including the Newton-Krylov-Schwarz method). Finally, the use of parallelism to decrease the severity of these limits

(particularly the clock time required per solution) was suggested as a potential technology to allow larger, more complex, and more accurate simulations.

The target of this research was the development and study of an "$n \leq 16$ scalable parallel linear system solution technique for use with a Newton-Krylov non-linear algebraic solution method." More specifically, the following goals were listed in the introduction:

1. provide robust, parallel solutions to steady-state viscous compressible flow on a backward-facing step at a Reynolds number of 100 and inlet Mach number of 0.0025,
2. investigate the mapping of various implementations of Newton-Krylov-Schwarz solution algorithms on the Cray C90 and SGI Onyx,
3. examine a parallel matrix-free implementation of the above methods, using pseudo-transient relaxation in conjunction with a lagged Jacobian and preconditioner formation strategy to reduce the influence of the preconditioner formation on the algorithm execution time, and
4. suggest an "optimal" hardware configuration for parallel Newton-Krylov-Schwarz Navier-Stokes solutions of this type.

With the exception of the final topic (to be discussed later), this work remained faithful to the goals initially posed in the introduction. First, model problem was developed and the discrete representation of the governing equations was derived. The action of the inexact Newton technique on the resulting non-linear system was investigated next, followed by the derivation of the Schwarz preconditioning methods and a detailed explanation of the Krylov linear system solution process. Finally, the concepts of pseudo-transient relaxation and the matrix-free technique were examined in detail. These analyses form the theoretical component of this study.

The second component of this work was devoted to an experimental examination of the parallel performance of implementations of the aforementioned concepts. Numerous simulations on the architectures of interest were presented along with discussions of the results. To remain faithful to the goals of this work, all that remains is a summary of the results, a discussion of further study topics, and a qualitative discussion of hardware features that could be useful for future simulations of this type.

7.1 Optimal Architecture

The results of this work indicate a solution scalability to four processors on the SGI Onyx (recall however that only four processors were available) and eight processors on the Cray C90. Most, if not all, of the suggestions on how scalability could be extended to more processors on each architecture were couched in terms of algorithm and algorithmic mapping improvements. This is a very reasonable approach, as scalability beyond this point was negatively affected by preconditioner degradation, inefficient memory access behavior, and a lack of available parallelism within the Krylov technique. Furthermore, the software (and algorithms) is the only area within the control of the computational scientist. On the other hand, it may be possible to add hardware features or modifications that assist with algorithmic efficiency and mapping issues or allow the scientist more latitude with algorithm modifications. This section suggests possible hardware changes, additions, or improvements that could prove helpful in achieving scalable performance of CFD solutions based on Newton-Krylov-Schwarz methods or similar techniques.

Architecturally speaking, several general facts became apparent during this study.

- The Cray C90 architecture performed very well, overall, on this problem. Its archi-

tectural features, vector processing capability, memory access latency and bandwidth, and development tools combined to result in a very satisfactory environment for the simulation research. The Cray ran larger problems and achieved much faster rates than the SGI. For the computational scientist or developer, the compilers, analysis tools, and development environment on the Cray are second to none.

- Cache-based architectures may yield better than expected performance if the cache size, problem size, and number of processors can be selected to optimize cache hit ratios. The superlinear speedup obtained on particular problems due to cache effects on the SGI certainly provided an interesting result. It is an open question if these effects could be utilized to achieve better scalability in a system with greater parallelism, perhaps mitigating a portion of the algorithmic and mapping difficulties encountered.

- Efforts to increase granularity and decrease memory contention on banked vector machines increased vectorization effectiveness and parallel performance, often in unpredictable ways. Significant decreases in execution time appear possible by considering architectural details.

- The decline in popularity of vector processors suggests to many that modern superscalar processors may be approaching a performance parity with vector capable hardware. This conclusion may be true for certain applications, however it did not appear to hold for this study. Early work on vectorizing the code for the Cray quickly reaped significant performance benefits with this loop-laden, indexed array intensive simulation code. This result suggests that symmetric multiprocessor (SMP) vendors may be premature in dismissing vector capability for large scientific codes of a similar nature.

- Extreme memory capacities may be necessary to run larger problems.

- Throughout this study, the importance of dedicated access to both machines was obvious. It became difficult to obtain consistent results (and likely meaningful results) as the multiuser workloads grew heavy. It is likely not realistic to expect acceptable parallel application performance on heavily loaded machines.

It is also apparent that, given the Newton-Krylov-Schwarz algorithms as implemented in this study, neither of the two architectures were optimal for the solution of the model problem.

- The SGI was limited by the number of available processors (four). It was not possible to assess its performance in comparison to the Cray with 8 or 16 processors, or examine in detail the scalability of the cache effect.

- The R4400 processors employed on the SGI had a significant performance disadvantage when compared to a Cray processor. The Cray had a clock cycle advantage over the SGI, 4.2 ns. versus 6.6 ns., but the vector units and memory systems in combination with other effects allowed the Cray to perform much better than the clock cycle difference would seem to indicate.

- Efficient memory access was difficult to achieve on the Cray. The memory contention problems initially encountered in parallelizing the Jacobian appeared time and again at greater levels of severity as the scalability study increased in number of processors considered. These contention problems, while relatively easy to visualize in a simple loop construct, become very difficult to mitigate in this application due to access pattern complexity.

- In comparison to Cray's offering, the programming tools and compilers available on the SGI were lacking features and usability. The Cray compilation system supported

a large number of modes, options, and directives that allowed user specification of minute details of the optimization process. The SGI system was not nearly as flexible. Methods that would allow the user to greatly increase the system performance, such as filling the delay slot on branches and explicit inlining of particular routines, could not be easily specified to the compiler. Analogous operations on the Cray system were easily accomplished. Of particular interest was Cray's vectorization analyzer. This tool provided a listing that indicated the loops that vectorized, and more importantly, which did not. The reasons why vectorization was inhibited were explained, typically listing the offending variable on the offending line. Throughout the entire analysis (and barring user enthusiasm), the compiler never vectorized a routine incorrectly to produce a logic error.

- Both machines were limited by the quantity of main memory.

Any comparison of this type quickly becomes somewhat subjective. To determine, beyond a certain margin of error, the optimal architecture relies on a qualitative interpretation of the phenomenon encountered, the correct understanding of the phenomenon, and the derivation of a reasonable solution to correct the observed behavior. With this process in mind, one could quickly surmise that a combination of two technologies, a data caching mechanism similar to that seen on the SGI and the Cray hardware/software system, has a strong potential to yield a near optimal scenario.

All of the data seemed to suggest that the Cray hardware was near optimal, except when granularity (*i.e.* parallel overhead) or memory contention issues were encountered. The memory access system and its performance on this machine was quite impressive, even considering the access problems encountered. The *mfmatmul* algorithm would not scale to 16

processors due to memory access difficulties. This is a very complex algorithm that accesses all of the state variables in each of the discretization cells and all the corresponding "Jacobian-vector" locations each Krylov iteration. Considering this amount of data being accessed by two vector units on each of 16 processors with a cycle time of 4.2 ns. in a concurrent fashion is a bit overwhelming. The SGI system was not tested under conditions even remotely comparable (recall that the SGI uses a banked memory system on a proprietary bus accessed by four superpipelined superscalar processors running at a rate of 6.6 ns. through a distributed cache system).

It is quite likely that the addition of a local processor cache to each Cray processor could be used to good effect on scalar memory accesses, even on the SRAM-based C90. These scalar accesses cannot be accomplished efficiently using vector memory access; experience with workstations (and SMPs) indicate that data locality (optionally with a pre-fetch capability) best addresses these types of loads and stores. A distributed cache capability on the Cray should behave similarly. Additionally, this hardware could potentially reduce bank contention and provide more bandwidth for vector accesses.

It is not immediately obvious that it would be beneficial to use cache for vector accesses. Vector accesses can be accomplished at the rate of a single location per clock cycle on the C90 architecture (ignoring start-up time and contention). In the ideal case, the access time for a vector element cannot be improved further with cache. However, if vector accesses can be limited to the local processor cache, bus traffic will substantially decrease along with the vector load and store bank contention problem (see Appendix A). An overall efficiency gain with vector caching would require the decrease in contention and bus traffic to overwhelm the overhead of implementing a vector-based cache coherency mechanism. Vector (array) accesses also tend to migrate large amounts of data, especially in scientific applications. Experience

with workstations indicates that large accesses of this type, especially with non-unit stride, do not map well onto small (or even moderately sized) caches [76]. The Cray system, however, functions quite well under this sort of access. It is possible that an effective cached-vector memory system would require a large amount of cache per processor. Clearly, there appears to be justification in the further examination of a cached-vector memory capability for scientific applications.

It may be valuable to construct some form of "virtual" vector memory system on the Cray. To minimize contention, the arrays must be arranged in memory so that access to a given bank is not attempted more than once per c clock periods (see Appendix A). This can be accomplished by hand reasonably well for small applications. The Cray compilation system can perform array and common manipulation as an attempt to mitigate contention at compile time. This feature would likely be effective on small to mid-sized applications, as long as the application is limited to one source file. The simulation code employed for this study had multiple source files, with an include file containing the shared common definitions. With this structure, the contention analyzer was ineffective and created useless code (because the common definitions were included in each source file, the analyzer optimized each inclusion for each particular source file, resulting in a loss of consistency on the common "memory image" of the data). A more complete tool of this type, however, could prove effective in addressing contention statically. It may be possible to employ specialized hardware to address the contention problem dynamically, using an analysis of the memory access patterns to move arrays around within memory. Clearly, this technique resembles a conventional virtual memory system, as dynamic address translation would be required for memory access. The dynamic system would incur some runtime overhead, but the efficiency improvement due to a reduction in contention should easily offset this overhead. As a last point, extending this

dynamic translation may also evolve into a physically distributed, logically shared-memory system. For hardware scalability much beyond the current Cray system, bus traffic will mandate a distributed memory system. From an ease of programming standpoint, a shared-memory programming model is desirable. Extension of the translation concept to a true physically distributed system would be a logical path to address these concerns.

To summarize these ideas, consider the well-known "triangle" balance concept: processing power, memory access, and I/O must be balanced to provide the optimal general-purpose computer. The Cray system clearly provides more processing power in comparison to the SGI (although for this application, more would be welcome). The memory system on the Cray was designed to provide data to this aggregate processing power to keep the CPUs busy. However, for these algorithms as implemented, the effective memory access efficiency was not sufficient to sustain scalable performance beyond eight processors. Clearly, the hardware provided sufficient memory to CPU bandwidth with adequately low latencies, but the algorithms as implemented could not take full advantage of the Cray hardware (due to complex memory access patterns). It was discussed that software changes may not be the optimal method to increase the scalability of the algorithms; hardware assistance along the lines of scalar caching and/or virtual vector addressing may prove helpful. Additionally, virtual addressing may allow hardware scalability significantly beyond the level presently implemented by facilitating a physically distributed, logically shared-memory system.

Finally, distributed memory machines (including massively parallel machines) have not been examined in this study. Clearly, memory contention concerns will be significantly reduced on these machines, but communication time among processors (effectively ignored on the shared-memory machines) will be significant considering the amount and distribution of serial code remaining in the Newton-Krylov-Schwarz solution as currently implemented. In

fact, to achieve any level of meaningful performance on these architectures would require an overall re-design of the simulation code. Additionally, a large number of processors cannot be used for the preconditioner formation and application due to the degradation of the preconditioner quality as the number of subdomains is increased (recall the increase in Krylov iterations as the number of subdomains increases). However, it is likely possible to implement a matrix-free pseudo-transient solution that minimizes the importance of a scalable preconditioner in conjunction with a parallel Krylov technique [42] on an architecture of this type. Furthermore, work on a coarse grid preconditioner could mitigate or eliminate any scalability problems along these lines at some point in the future. As such, barring the development time of a distributed memory version of the simulation code, these type of architectures could provide real benefits without many of the disadvantages seen with the hardware (more correctly algorithm/hardware mapping) employed in this study.

This discussion is clearly hypothetical at this point. Much work (and a study similar to this one) would need to be performed on a representative architecture of this type to determine the feasibility of the approach, particularly in the area of preconditioning.

7.2 Summary of Results and Future Research Topics

The results were presented beginning with a study of additive Schwarz preconditioning, with and without subdomain overlap. Multiplicative Schwarz preconditioning was investigated in a similar manner. The final results examined were based on the matrix-free technique using additive Schwarz preconditioning and pseudo-transient relaxation. This discussion will not reiterate the material found in the previous chapters; only the salient points directly related to the overall goals will be summarized.

Newton-Krylov-Schwarz algorithms were used to solve a model problem of compressible 2D flow past a backward-facing step. Parallel/vector aspects of the solution algorithm were exploited on a 16 vector processor Cray C90 computer and a 4 processor cache-based SGI Onyx. The first results examined a direct steady-state solution using additive Schwarz preconditioning. A study of the algorithms indicated that the Jacobian formation and preconditioner formation and application routines were readily parallelized.

Observations indicated that the Jacobian formation routine performed well in parallel for two and four processors on the C90. However, this routine did not scale well beyond four processors, and exhibited poor performance on eight processors. Because the Jacobian formation algorithm is inherently parallel, the poor scalability beyond four processors was attributed to memory contention during the Jacobian update operations. The linear solution operation time (the preconditioner formation and TFQMR iterations) increased as the number of preconditioner subdomains was increased beyond four on the 64×320 volume model problem (81,920 unknowns). Two factors were identified to explain this behavior.

- On the 8-processor C90 test case, the formation of the preconditioner contributes 14% to the total linear solution time. Clearly, the bulk of the linear solution routine remains serial.

- As the number of subdomains increases, the number of TFQMR iterations required to solve the system (Equation 1.18) increases substantially. This behavior indicates that as the number of subdomains used to construct the preconditioner is increased, the effectiveness of the preconditioner decreases, requiring more iterations for solution. It was postulated that this problem may be partially alleviated with the use of subdomain overlap.

The second factor will quickly limit scalability of the solution beyond a small number of processors. It was also suggested that it may be possible to obtain preconditioners that scale more effectively with the number of subdomains, perhaps with: (1) an additive Schwarz scheme with overlap, or (2) a multiplicative Schwarz implementation incorporating a coloring scheme to allow concurrent preconditioner formation. It was also suggested that the use of a matrix-free pseudo-transient simulation may achieve scalability by allowing fewer preconditioner formation operations per full simulation.

The SGI Onyx 32×160 simulation exhibited superlinear speedup due to cache effects on the two-processor runs. It was verified that this superlinear effect could be achieved with four processors in the Jacobian formation if the problem was scaled to 64×320. Scaling alone did not provide a superlinear speedup in the TFQMR routine or the overall results on four processors, however. It was acknowledged that the TFQMR routine had not been optimized for maximum granularity, cache hit efficiency, or to minimize the serial code currently in the routine. With these changes, it is not clear that overall superlinear speedup can be achieved with four (or more) processors.

In comparison with the two and four processor Cray results, it appeared that the SGI provided better scalability with additive Schwarz techniques. It is possible that these techniques map better to a cache-based architecture. However, it is not possible to firmly conclude this point with this limited data and using the SGI Onyx due to the multitude of other differences (primarily processor speed and lack of vector capabilities) between the Onyx and the C90. The data certainly suggests that when minimal runtime (ignoring parallel scalability) is of concern, the Cray is the ideal platform. For the 64×320 four-processor run, the Cray is an order of magnitude faster.

Several additive Schwarz simulations were performed using overlap. Furthermore, signif-

icant code and algorithm changes were performed on both the Jacobian and preconditioner routines in an attempt to increase the parallel granularity of these algorithms on both architectures and to decrease memory contention on the Cray. These changes were quite effective.

- Subdomain overlap decreased the total number of TFQMR iterations and the rate of growth in iterations with an increase in the number of subdomains.
- The combination of code modifications and higher granularity due to larger overlapped subdomains resulted in a speedup of 14.3 in the Jacobian section and 13.5 in the preconditioner on 16 C90 processors.

Clearly, this data shows that the both the Jacobian and preconditioner scale with reasonable efficiency to 16 processors. However, the overall solution did not scale well, a speedup of only 4.9 was achieved. The significant amount of serial time remaining within the TFQMR routine overwhelms the parallel preconditioner time. Furthermore, it was also obvious that due to the severe memory requirements of overlapped subdomains, this technique is likely only of interest when memory requirements are not an issue. As such, overlap was abandoned as a method to ensure preconditioner quality with additive Schwarz.

A similar study of multiplicative Schwarz with and without overlap was performed. It was quickly determined that on a degree of parallelism basis, multiplicative Schwarz offered no advantages over additive Schwarz on the model problem. Again, overlap resulted in extreme memory requirements, and was abandoned. Due to this result, multiplicative Schwarz preconditioning was not addressed further within this study.

At this point in the study it was apparent that scalability could not be achieved effectively by concentrating on the preconditioner routines alone. Furthermore, the model problem, while numerically challenging to solve, was very simplistic in comparison to those problems

of current research interest. As such, a method combining a matrix-free solution technique in conjunction with pseudo-transient relaxation was developed to address both issues.

The effect of the preconditioner on the overall solution algorithm was decreased by "lagging" preconditioner formation operations, *i.e.*, the preconditioner was updated every m Newton iterations instead of each iteration. This method was successful and effectively addressed both the robustness concern for more complex problems and provided an overall speedup of 6.1 on 8 C90 processors. This results in an overall scalability to eight processors[1], with a parallel efficiency of 76% overall. Clearly, this result appears quite promising. An attempt to examine scalability to 16 processors was not successful, largely due to a marked mapping difficulty with the matrix-free algorithm on the C90 architecture. It was postulated that this is again due to memory contention (similar to the problem experienced earlier with the Jacobian algorithm). However, due to the memory access complexity of this routine, it is not likely that the contention problems can be easily solved.

The goal of achieving scalability to $n \leq 16$ processors remains somewhat elusive. Clearly, despite the fact that access to the C90 was not dedicated, scalability to 8 processors was demonstrated (these results were not corrected in any way for the multiuser workload). The results also indicate that scalability beyond this point is very unlikely. As of this date, this issue may not be of paramount importance to the practicing computational scientist solving these problems; Crays over 8 processors are quite expensive both from an initial expense perspective and a usage/maintenance viewpoint (CPU time is often charged as a multiple of the number of processors employed) and SMP workstations beyond 8 processors are very rare. In the future, however, scalability beyond 8 processors will likely become quite important.

Again, computational scientists wish to perform simulations in a minimal amount of time

[1]Appendix A discusses the scalability metric used.

within the resources available. This work has concentrated on the first desire, but has not neglected the second (overlap was abandoned due to memory requirements). Simulations are usually constrained *both* by memory and time limits. In fact, this study quickly found that it is not possible to solve problems of any interesting size while being limited to 256 Mwords (2 Gbytes) of Cray memory. As the model problem was scaled in size, the memory limit was reached long before the simulation time for the solution became an issue (an analogous problem was encountered on the SGI with its 1 Gbyte of main memory). Furthermore, on the Onyx, a configuration parameter or operating system limit did not allow access of the total main memory from a single process on the benchmark machine.

Unfortunately, this study appears to have posed more questions than it has answered. There are many opportunities for future research in this area; ranging from the physics of the problem to specialized hardware tailored for these techniques. Along the theme of this work, three areas of paramount interest are immediately evident.

- Reduction of the memory requirements of this solution technique for a given problem. The use of an inexact subdomain linear solution technique (perhaps ILU versus the implemented LINPACK Gaussian elimination) would be a promising initial candidate.
- Modification of the preconditioner algorithm to prevent (or decrease) the decay in preconditioner quality as the number of subdomains is increased. The use of a coarse grid/fine grid scheme may be an initial approach.
- Hardware or algorithm changes to enhance the scalability of the matrix-free routine.
- Development of a parallel Krylov solution technique for this architecture class.
- Mapping these techniques to a distributed memory architecture with significant per-

processor performance (the IBM SP2, Convex Exemplar, Cray T3E, *etc.*).

Finally, to provide a closing perspective, the model problem studied requires a very powerful preconditioning technique due to the low Mach number inlet condition. If this were not the case, better scalability results could certainly have been achieved. As such, the model problem selected demonstrates the "worst-case" scalability that would be obtained with these techniques. It is evident that the results and conclusions of this study are specific to the model problem. However, these results may be applicable to a much wider variety of situations if the results are viewed as a lower-bound to the performance that may be achieved on a "general" simulation.

In any event, this area of research appears quite promising and is fertile for further work and discoveries, especially as hardware improvements in conjunction with improved parallel preconditioner algorithms are developed over time. This author has only been further stimulated by this study to pursue future work in this area.

Appendix A

Some Mechanics of Shared Memory Parallel Computation

This study has focused on the shared-memory parallel solution of a specific problem using parallel algorithms. The research on this problem has resulted in the collection of a substantial amount of material addressing this topic.

In essence, this work has reinforced the concept that the development of efficient parallel algorithms is an integral portion of obtaining efficient parallel execution on hardware of interest. Clearly, the concentration of this work on the algorithmic theory and the presentation of the performance results has not adequately addressed the many implementation and architectural issues that proved important in this study. This appendix seeks to summarize the implementation and architectural issues encountered in the implementation of the solution algorithms on the various architectures examined.

This appendix is intended to present a general implementation and architectural overview based on the empirical data obtained from this study and the experiences therein. As such,

an attempt was made to generalize problem specific behavior observed in this study to a general form. As much of this information is based on observations under particular conditions, they may not be generally applicable to all conceivable problems under all conditions. Furthermore, proofs or theorems based on these observations are left for future research.

This appendix is organized as a set of notes, with the intention of providing a reference on shared-memory parallel computation. Within this framework, this appendix is a summary of some of the experiences of this author to date on applied parallel computation.

A.1 Applied Parallel Computation

In this context, applied parallel computation is the efficient parallel solution of large-scale problems in a production shared-memory environment. To date, there is a growing body of knowledge of the parallel solution of large-scale problems in a research environment. Many of these efforts have been very successful and have clearly advanced the understanding of specific problems under certain conditions. These successes have provided strong motivation for further research in parallel computation, algorithms, and hardware in order to migrate from research towards application/production.

Unfortunately, very little of this work has migrated from the research environment to provide any significant impact in analysis tools. Engineering analysis is largely still accomplished using serial methods, as in the past. Although parallel computation is a popular area, it cannot remain popular without clear analysis benefits. There are several reasons for the lack of parallel analysis tools (and interest in parallel execution in general) outside the research establishment.

1. Research has been focused on maximizing the performance of a particular problem

using the optimal algorithms to solve this problem. This is clearly very important, as this work drives algorithm and hardware advances. However, the analysts do not directly and immediately benefit. Analysis requires robust parallel algorithms capable of solving a large variety of problems with little operator training. Research efforts often solve a particular problem using finely tuned algorithms. These problems may be less complicated or of a different nature than those of analysis interest. Additionally, tuned solution algorithms often require significant changes for use on different problems, requiring a high level of operator training.

2. Parallel capable hardware remains quite expensive. The very recent popularity of SMP workstations indicates that this problem may not remain significant in the future. However, in the recent past, hardware that supports parallel execution of any kind (even inefficient network interconnected resources) has been too expensive to make any impact on engineering analysis.

3. Software supporting parallel execution is extremely expensive. The development of serial software is very expensive and often beyond the reach of many engineering groups. Even simple shared-memory capable software is many times more expensive to develop and requires specially-trained personnel for development. Distributed memory parallel software is yet more difficult and expensive (and is still mainly limited to pure research areas).

4. Many analysts (and more importantly, engineering managers) do not entertain the advantages of parallel execution due to a lack of understanding of the process. In some environments, computer simulation is not yet accepted as being a precursor to prototyping as part of the engineering design process, let alone parallel execution of

simulation tools.

5. Clearly, some areas, tasks, and applications cannot use parallel simulation for a benefit. If multiple executions of a tool are required to obtain a set of data (e.g., for a parametric study), it is much more effective to run multiple serial executions on the available processors rather than one parallel execution for each data point serially. In this case, the multiple serial executions effectively maximize throughput, providing a "parallel efficiency" of 100%.

These topics (and likely several more) must be adequately addressed for parallel processing and simulation to achieve widespread popularity within engineering analysis.

A.2 Hardware Selection for Applied Parallel Computation

The performance of a particular code on a given platform is strongly dependent on the structure, size, memory access patterns, layout, and style of the program, to name but a few factors. Additionally, the performance of the code on a particular architecture is strongly dependent on the design of the machine, and on how well the program exploits the available performance characteristics of the computer (*i.e.*, how well the code *maps* to the architecture of interest). As an example, it is often possible to construct two computer programs that perform similar (or identical) tasks, yet execute quite differently on a given machine (consider a program that exploits data locality and has a very large cache hit ratio versus another program that achieves virtually no cache hits). Furthermore, it is also usually possible to construct a program that maps well to a particular machine design, but executes poorly on a different architecture. In fact, one may concede that a machine with certain features (machine "A") is considerably faster than a machine with different features (machine "B")

on a particular set of programs, yet encounter a particular program performing a similar function that executes much faster on machine "B."

Clearly, the only way to select an optimal machine to run a particular code is a comparative process of timing the execution of the code on all architectures that may be applicable (*i.e.,* "benchmarking"). This obvious technique, however, leaves much to be desired. Benchmarking identifies the fastest machine for the given program on the input data examined on the set of machines with the configurations tested. This result leads to several questions.

1. If the input data to the program is changed to double the memory requirements of the execution, is the selected machine still the fastest of the group examined?

2. If the program is updated to a new, more efficient version with different memory access patterns, is the machine still the fastest of the lot?

3. If the program is designed to model fluid flow and heat transfer in a piping system using explicit solution techniques, will the selected machine be the fastest with another program solving the identical problem using implicit techniques?

This set of questions clearly suggest that a general purpose computer system should be selected by benchmarking the complete set of codes that will be run on the machine, usually weighting the results based upon order of importance, selecting the machine that consistently out-performs all others. This technique may be quickly dismissed due to the obvious cost of performing an analysis of this magnitude. Is there some technique that may be employed to select an optimal, or near optimal, general purpose machine for the workload of interest?

It is often elementary to construct a list of absolute requirements that must be met by the system. This step allows the elimination of clearly unacceptable machines from consideration. As an example, given a highly parallel program performing a Monte-Carlo technique or an

explicit fluid-flow calculation, a massively parallel system with thousands of processors would likely be the fastest architecture by a significant margin. However, such a machine would likely be very costly, possibly tens of millions of dollars. Additionally, the costs of executing a program on the machine should be considered, as it would likely require a significant number of man-hours of effort to manually modify a particular program to execute efficiently on the machine. As of this date, machines of this type are generally only applicable to strongly research-oriented environments; the ability of an ordinary user to develop and execute a "production" application efficiently and cost-effectively on such an architecture should not be assumed. As such, for most installations, the consideration of this architecture makes little sense.

Toward the other extreme, the consideration of a PC or "workstation" class machine can be quickly dismissed due to the limited processing capability and data throughput capacity of the system. For example, consider a large number of simultaneous scientific applications with a highly optimistic data requirement for each application; this scenario mandates a system with a processor and bus capacity many times that exhibited by workstation class machines.

A.2.1 Requirements

The ability to handle a true multiuser scientific workload requires a machine (or an aggregate, "cluster") that can handle the processor needs of the applications and service input/output (including memory access) requests without a significant degradation of "per application level" performance. This capability must be accomplished for a reasonable cost (acquisition cost plus maintenance costs plus any costs involved in moving applications to the machine and modifications required for efficient execution). With this summary of needs, several specific requirements are obvious.

- Multiprocessor Capability. It is well known that in a batch, throughput environment, matching available tasks with an equivalent number of processors maximizes throughput (for a given application assuming no bus or resource contention). Some applications may also be designed to support task-level parallelism; the use of multiple processors to complete a task in less time than required for a single processor.

- Large main memory. As a minimum, the system must contain memory sufficient to execute the largest program. On a shared-memory system additional memory is usually added to minimize memory contention under timesharing conditions. In their favor, shared-memory systems also achieve better memory utilization for a given workload, often maximizing the time between system upgrades and minimizing the system cost by decreasing the aggregate memory requirements.

- Upgradeability and expandability. The system should be readily (and inexpensively) upgraded to newer, faster processors, more processors, and larger memory than the base system.

- Software development environment. Given an appropriately constructed application in general form, the development environment (*e.g.*, compilers) should generally create an image that executes at peak efficiency on the hardware. As an example, given a parallel application, the compiler must recognize the parallelism and construct an image that executes efficiently on the available processors in parallel without user intervention or manual code restructuring.

- Reasonable costs. The system must be affordable in the initial purchase, later upgrades, maintenance, licensing, and administration.

Machine	$N = 100$ (Mflops)	$N = 1000$ (Mflops)
IBM RS/6000-R24 (71.5 MHz)	142	246
IBM POWER2-990 (71.5 MHz)	140	254
DEC 8400 5/300 (4 proc 300 MHz)	140	1351
IBM RS/6000-59H (66 MHz)	132	230
IBM POWER2 model 590 (66 MHz)	130	236
SGI POWER CHALLENGE (90 MHz, 4 proc)	126	2045
Cray J916 (4 proc. 10 ns)	121	743
DEC 2100 5/250 (4 proc 250 MHz)	119	317
IBM POWER2 model 58H (55 MHz)	101	197
SGI POWER CHALLENGE (75 MHz, 4 proc)	104	993

Table A.1: LINPACK benchmark results for machines under (or near) $300K and over 100 Mflops performance (1/1/96).

For a general purpose scientific system, it makes little sense to consider any system that is not competitive on a per-processor performance level with other machines in the price range allowable. Ideally, one would immediately narrow down the machines that would be considered to a set of the most desirable (say, the top 10 performers on the set of programs of interest). As this difficulty of comparison has previously been discussed, perhaps the machines could be ranked on performance based on a widely accepted (or at least understood) benchmark that has some meaning to the scientific workload envisioned. Furthermore, this technique may be quite valuable if it is used only to eliminate the consideration of clearly unacceptable machines, not as a technique to select the "best" machine from a group.

Many scientific applications involve solving dense systems of linear equations. Many of these applications employ calls to the LINPACK linear solution library. As such, a benchmark based on this library [7] may have some meaning to provide a rough comparison of hardware. Table A.1 lists the machines within (and close) to the $300K purchase window, with LINPACK $N = 100$ performance above 100 Mflops. The $N = 100$ level column is for a small solution of order 100, allowing no changes to the benchmark program beyond what

the compiler itself recognizes and performs. As a side note, a problem of this size will almost always fit in a moderate workstation cache. Additionally, as the LINPACK benchmark is not explicitly parallel, a compiler is not likely to recognize any parallelism within the code; as such these figures likely indicate performance on a single processor. The $N = 1000$ column is a larger problem of order 1000 (may still result in a high cache hit ratio in a well designed system). Furthermore, unlimited changes to the code are allowed for this benchmark, as long as the accuracy of the solution is retained. In the case of a multiprocessor machine, vendor modification of the benchmark typically guarantees optimal performance in an efficient parallel execution mode.

Discarding all machines below 100 Mflops single processor performance may at first seem arbitrary; considering that the fastest machine in the group is capable of 142 Mflops, any machines below 100 Mflops are then at least 30% slower than the fastest machine in the range on the LINPACK single processor results. The line must be drawn somewhere, it is not likely a very defensible position to purchase hardware that is 30% (or more) slower than a competitor considering the importance of single processor performance in a scientific environment. Based on this statement of requirements and a multiuser (30–50 users) scientific workload, a minimal entry level system would likely consist of 4 processors, 40 Gbytes disk storage, and 512 Mbytes main memory. Further enforcing the requirement that the initial cost be under (or near) $300K, very few systems remain for consideration. As a side note, the SGI machines with this configuration are somewhat above the $300K limit but aggressive vendor discounting could allow them to meet the cost requirements; as such they will continue to be considered at this time. The top four machines in parallel performance on the larger benchmark are consequently the 90 MHz SGI, the DEC 8400, the 75 MHz SGI, and finally the Cray (Table A.2). Of note, with modifications to the benchmark code allowed, the 90

Machine	$N = 100$ (Mflops)	$N = 1000$ (Mflops)
SGI POWER CHALLENGE (90 MHz, 4 proc)	126	2045
DEC 8400 5/300 (4 proc 300 MHz)	140	1351
SGI POWER CHALLENGE (75 MHz, 4 proc)	104	993
Cray J916 (4 proc. 10 ns)	121	743

Table A.2: LINPACK benchmark results for top four machines under (or near) $300K and over 100 Mflops performance considering other imposed requirements.

	Benchmark A - Multiples Faster than Y-MP/1								
Machine	EP	MG	CG	FT	IS	LU	SP	BT	Avg.
DEC 8400 5/300 (4 proc 300 MHz)	3.23	-	-	-	-	2.10	2.37	2.92	2.66
SGI P. CHAL. (90 MHz, 4 proc)	2.88	2.09	1.35	1.73	1.61	1.77	2.35	2.65	2.05
Cray J916 (4 proc. 10 ns)	2.93	2.07	2.70	2.57	3.00	2.47	2.08	2.49	2.54

	Benchmark B - Multiples Faster than C90/1								
Machine	EP	MG	CG	FT	IS	LU	SP	BT	Avg.
DEC 8400 5/300 (4 proc 300 MHz)	0.93	-	-	-	-	0.65	0.76	0.80	0.79
SGI P. CHAL. (90 MHz, 4 proc)	0.83	0.68	-	0.61	-	0.58	0.81	0.80	0.72
Cray J916 (4 proc. 10 ns)	0.86	0.69	0.81	0.82	0.93	0.85	0.73	0.72	0.80

Table A.3: NAS parallel benchmark results [2].

MHz SGI is fully 2.75 times faster than the Cray.

A similar exercise performed by consulting the NAS Parallel Benchmark [2] results in the selection of basically the same set of machines, but different levels of performance and rankings result (see Table A.3).

From this exercise, one becomes quite comfortable with the machines composing this group likely being the optimal set for scientific computation of similar workloads, as the benchmarks considered result in the selection of the same group. However, it is also quickly apparent that differentiation between the machines based on benchmarks is a futile effort. As such, can the benchmark results be combined with a knowledge of each architecture to reason which machine would be most applicable to the scientific workload?

A.2.2 The Optimal Architecture

Clearly, the architectures of the Cray, DEC, and SGI fall into two categories. The Cray is a shared-memory multiprocessor that directly accesses main memory via a bus without an intervening cache subsystem. Due to the design of the interleaved memory system, the bus, and the processors, the Cray can also pipeline memory accesses under certain conditions. Each Cray processor, in addition to the usual pipelined scalar processing capability, has a deep pipeline processing unit able to operate on multiple registers during a single operation without encountering data hazards (*i.e.*, a "vector" unit) for those applications that can use it. Based on this design, the Cray may be called a *real* memory multiprocessor.

The DEC and SGI machines place a local processor cache at each processor to attempt to mitigate the usual main memory access penalty. In theory, given a suitable cache design, the requested memory item will be found in the fast cache memory, eliminating the need to access the item from the relatively slower main memory. Also, the processors implement an instruction pipeline capability to perform scalar instructions in a minimum number of clock cycles per instruction (similar to the scalar unit on the Cray). Additionally, this design also implements the capability to issue multiple instructions each clock cycle (superscalar operation), in contrast with the Cray's vector pipelining. In theory, superscalar and vector processing provides an equivalent execution efficiency, all things being equal. Due to the experience with vector processing, compiler design, and the vector memory access system, current superscalar designs may not always be competitive, performance wise (for evidence of this, see the NAS Benchmarks in Table A.3). Because these machines implement a virtual memory model (each memory transaction is translated to a physical address prior to reference), these systems are often termed *virtual* memory multiprocessors.

Before considering the intricate details of each system and reasoning about the performance aspects of the two designs, it is necessary to stipulate what is meant by the term "scientific computation." Clearly, this category is strongly open to interpretation. In the context of this report, scientific computation will be interpreted to involve simulation, modeling, engineering analysis, solution of equations, numerical analysis, etc. This type of work is often computed with the use of large, indexed data structures in the form of arrays. Depending on the application, of course, these arrays range widely in size. However, as simulation needs become more detailed and precise, one increasingly encounters arrays greater than 100 Mbytes in size and complete executable images nearing 500 Mbytes. To summarize, it can likely be conceded that the efficient processing of large array data types is a very important aspect of scientific computation (perhaps the most important in many cases). Furthermore, the expectation of a polynomial growth in the size of these data types in the immediate future is also a defensible position, considering the move to three-dimensional analysis.

A.2.3 The Comparison

At the processor design level, it is easy to enter a discussion about the strengths and weaknesses of the Cray real memory approach versus the cached virtual memory approach. There are clearly many significant operational details that may be exploited by an application to provide a certain level of performance. However, these differences tend to be very application specific; the advantages of one design on an application often is a disadvantage for a different application. Reasoning about a general purpose architecture using a suite of dynamically changing codes using these details is likely not overly productive, in general terms.

As of this writing, the vector real memory approach has one clear and substantial advantage over the cached approach for large scientific codes, the memory access system. Vector

memory systems often have significant advantages when array memory accesses do not have unit stride [76]. Furthermore, if the array access patterns or array size is such that the working set of cache lines do not fit within the cache, scheduled direct bank memory accesses are clearly superior. Again, for the larger and more realistic benchmarks in the NAS suite, the direct memory access of the Cray machine is largely responsible for its performance on the tests.

To explore this concept in more detail, consider the earlier LINPACK results. These benchmarks are small enough that the working set of data will likely fit within the cache on the two cache-based machines. Furthermore, the array sizes are quite small and the overhead of accessing the Cray memory system is large in comparison to the time spent processing the data. Clearly, with this scenario, the cache machines perform at their best and the real memory machines are at their worst. This is not really an interesting comparison, especially considering that the LINPACK benchmark runs to completion in a matter of seconds on machines with this level of performance! Of much more interest to scientific users are the codes that run for hours, days, or even weeks. The NAS benchmarks are more realistic; not all of the benchmarks are likely to reside in cache on that architecture and the Cray memory overhead has fallen significantly as a fraction of total workload for problems of this average size. The realism is still limited. For example, benchmark A for LU decomposition executes in 135 seconds on the Cray. Clearly, this is still a very small problem.

A cache-based architecture reads a large array beginning with the processor requesting a memory fetch of an array element (often the first or last element of the array). Fetches are always directed at the cache. If the requested item is not resident in the cache, the cache controller fetches a cache line containing the item from main memory, with the processor (and pipelines) stalling until the request is satisfied. The time required for main memory

to return the line depends on the memory and bus speeds, bus width, cache line size, and other factors. Once the data arrives in the cache, the requested item may then be supplied to the processor, allowing it to resume processing. Because a cache line usually consists of several adjacent array elements, further requests for elements "close" to the first will be subsequently found in cache. The length of a cache line and the number of lines that may be contained within the cache are functions of the total cache size. Clearly, within a loop structure accessing arrays, one desires a 100% cache hit efficiency for each element access (*i.e.,* every array element requested by the processor will be resident in the cache prior to the request). This cannot be achieved, as the processor must initially fill the cache with the array, but if the cache is physically large enough to contain the array the cache hit efficiency will approach 100% as the work (and time spent) within the loop increases.

This behavior of cache-based machines has a very strong effect on application performance, often overwhelming any other considerations. If the array size is large in comparison to cache size, the cache hit efficiency in a loop construct will be low. If array-based loop constructs dominate the workload in an application (they often dominate in scientific applications), the memory subsystem cannot service the processor data requests without stalling the processor. As an example, consider a machine with a main memory system that can supply a request for a cache line in 100 ns., and a processor with a 10 ns. clock. Further assume a loop structure that accesses successive array elements one each processor clock cycle. If the element is found within cache each request, the processor is not delayed and runs at full speed. However, for every cache miss, the processor must stall for 10 cycles waiting for memory to supply the data. This discussion invites a question. If only 10% of the array accesses miss the cache (a 90% cache hit efficiency), what is the resultant processing efficiency?

To ignore overhead, assume that the loop is infinite in size. If the cache hit efficiency were

100%, the system would clearly require one processor cycle to process each array element. For an efficiency of 90%, the system requires 1.9 cycles per array element (on average)

$$\frac{(1 \text{ cycle})(9 \text{ items}) + (10 \text{ cycles})(1 \text{ item})}{10 \text{ items}} = 1.9 \text{ cycles/item.}$$

Assuming the loop is infinite yields the system efficiency

$$\text{Eff} = 1.0/1.9 = 53\%.$$

In effect, this system runs at 53% of full speed.

Now, consider the same simple example on a banked real memory system. With this arrangement, the access of the first array element by the processor proceeds at memory speed, stalling the processor. However, the processor does not request data one element at a time with this design, it requests a vector of data. The memory system, upon receipt of this request, supplies the first element, say from the first bank of memory, in 10 processor cycles as one would expect. As the memory system knows that the processor has requested a vector of data, it immediately supplies the successor array element from the second memory bank on the next processor cycle without the processor's intervention. This process continues essentially independent of the processor for a very large array of data. Looking at the above example, it is clear that after the initial 10 cycles of stall overhead for the first item, every subsequent item may be accessed each processor cycle. As such, for large arrays in infinite loops (as the above example) the processor runs at near 100% of design speed and does not stall waiting for memory.

It may not be apparent from this simple example, but many details intercede that may

narrow this performance advantage in real situations. Furthermore, the example is a drastically simplified model of each architecture; the example was designed to illustrate the basic operation of each system and foster an understanding of the general behavior of each system for large array constructs.

A.2.4 Final Thoughts

This section was written to illustrate pitfalls of using benchmark data to select a general-purpose computer system. However, an "approximate" method was described on how benchmarks could be combined with clear system requirements to narrow the search space of available systems. For the requirements deemed important in this section, two general architectures dominated the final set of machines; the real memory and virtual memory multiprocessors from Cray, DEC, and SGI.

The second major topic discussed the memory systems of each architecture in the form of a simplified model. This exercise suggested that vector real memory systems, such as implemented in the Cray system, has a potential performance advantage for the scientific problems of interest due to a vector memory accessing capability.

In closing, selection of a system must consider the issues discussed herein along with many other specific requirements such as porting costs, binary compatibility concerns, amortized system costs, etc. Above all, the final system can **only** be successful if the requirements of the *users* drive the selection process to the exclusion of all other perceived criteria.

A.3 Shared Memory Hardware Programming Basics

The simulation code for this study was implemented in well-structured FORTRAN 77. The base version was developed to be quite portable, using an "elementary" style (only basic, textbook optimizations were used). Ideally, from this base, a compiler could recognize parallelism implicit in the code and include the necessary structure in the intermediate or assembly language to execute the requisite sections (possibly "basic blocks") in parallel.

The compilers for both the Cray and SGI are capable of performing this task, at least to some extent. However, this method is limited, as the compiler does not have intelligence, so to speak. It is only capable of performing a sequence of analysis steps to determine if a particular sequence of code may be safely executed in parallel (these compilers typically consider only loop constructs at a high level). Loops are typically parallelized if analysis indicates that there are no data dependencies that will affect the result and that the loop contains sufficient work to overcome overhead inherent in the parallelization process. It is clear that this is a sufficient condition for safe parallelism of a construct, however it is not necessary in many cases. Performing analysis in this manner may "miss" a significant amount of parallelism.

- The loop construct may not contain dependencies, but it may obfuscate the compiler due to its complexity, use of indirect addressing, or as a result of the "author's style."
- The compiler skips loops containing function and subroutine calls due to the difficulty of dependency analysis (especially with global data, commons, aliasing, and equivalencing). As such, the highest level of parallelism often available is completely ignored.

A perfectly acceptable method to address these problems is to allow the programmer to "instruct" the compiler of the available parallelism, where it occurs, and how the compiler

can best handle real and imagined dependencies. These instructions (directives) are often implemented as comment lines, but with a special syntax recognized by the compiler. Directives are often used (they are really mandatory) in shared-memory compilers to allow the explicit specification of parallelism.

The second compiler-based parallelism opportunity is instruction level parallelism (ILP). Lately, ILP has become important to pipelined and superscalar processors. Parallelism at this level can often be analyzed effectively for data dependencies by either hardware, compilers, or both. This fine-grained parallelism is currently very important in achieving modern microprocessor performance levels, but is often dismissed for scientific computing due to the low degree of parallelism (DOP) inherent in ILP. It is often stated that, on average, a branch occurs every seven instructions. This limits the effectiveness of ILP to a DOP of roughly seven. This "rule of thumb" clearly does not hold for loop-intensive scientific computation, however. One may interpret vector processing as a parallel operation over a series of identical serial instructions (perhaps as many as 128) operating on different data streams (SIMD). One could envision a new generation of a superscalar processor that could accommodate a large DOP for scientific codes in the same manner. This processor could be designed as a "vector processor" that operated on different data streams using optionally different instructions (MIMD). In this context, the "new" superscalar processor embodies all the advantages of existing superscalar technology with the addition of the ability to handle many parallel instructions, and existing vector technology with the ability to handle different instructions over different data values. This capability may be further enhanced with the vector memory systems often used in vector machines.

The above discussion outlines the essential points of the differences between parallel and vector processing on machines like the Cray, and parallel and superscalar processing on the

SGI. In summary, the differences are conceptually that parallelism is a large-grained MIMD concept, and vectorization (superscalar) is an instruction level SIMD (MIMD) operation. It is important to note, however, that independent hardware exists that mirrors this hierarchy. The large-grained parallel contexts are concurrent across multiple CPUs on the machine, while the vector parallel (ILP) contexts are concurrent across elements of a vector unit (superscalar pipelines) on a given CPU. As such, combining these two "forms" of parallelism may result in a surprisingly high level of concurrency in a scientific application that can make use of the hardware.

At this point, it is most instructive to consider the programming details of each of these architectures.

A.3.1 Cray Optimization Process

The initial development work was performed on the four processor C90, and was limited to routine compilation tasks and verification of correct operation and results. The vectorization (`cft77`) and parallelization (`fmp`) passes and all optimization were disabled for the initial porting phase.

Following the initial porting phase, scalar optimization parameters were investigated to obtain an optimal configuration in preparation for a flow analysis of the code. For the flow analysis, a Cray utility, **FLOWTRACE**, was used to identify code blocks that warrant optimization. Additionally, all LINPACK routines used in the code were replaced with Cray library calls to provide better efficiency.

FLOWTRACE results indicated that several functions and subprograms were suitable for inlining. These routines were explicitly inlined with the use of compiler directives and the requisite compile options. Additionally, an examination of the available compiler options

resulted in a set that minimized execution time,

```
cf77 -Wf"-dp -I inliner -o aggress -A full"
```

where

`cft77`	invokes the FORTRAN compilation system,
`-Wf" "`	passes the enclosed options to the `cft77` phase,
`-dp`	converts double-precision code to single precision,
`-I inliner`	forces inlining of files found in the `./inliner` subdirectory,
`-o aggress`	turns on aggressive optimizations, and
`-A full`	uses a full addressing model (enables indirect addressing into extended memory).

Following this work, `FLOWTRACE` indicated that several routines comprised the bulk of the execution time for the simulation code. These routines were involved in forming the Jacobian matrix, mainly:

- the u-momentum contribution routine *umom*,
- the v-momentum routine *vmom*,
- the routine for the mass-conservation contribution *cont*, and
- the temperature routine *temp*.

Additionally, the routine that forms and factors the preconditioner, *precond*, required an appreciable amount of computation time. Following these routines in percentage of execution

time were several utility routines, *matmul* - general matrix-vector multiply, *mivmldd* - a banded linear equation solution routine, *extrcv* - a data extraction routine, and several others.

Following this analysis, the vectorizing phase of the compilation system was invoked, with diagnostic output directed to intermediate files. This output was studied, with particular attention to the routines listed above, to determine the success of the "automatic" vectorization abilities of the compiler. Using this information, the routines that did not vectorize were modified by hand to enable vectorization if feasible. In many cases, a particular routine that did not vectorize was not significant in the overall runtime of the code. Obviously, spending time vectorizing these routines (or blocks) would lead to minimal improvements in runtime. Also, a few routines had dependencies that could not be easily addressed and were likewise ignored. However, in most cases, it was possible to vectorize important code segments. Automatic vectorization is enabled by specifying the additional compile option, `-ZV`, in conjunction with the above scalar optimization flags.

The Cray vectorizing stage provided very informative output that assisted greatly with the vectorization step. In most cases, the compiler provided diagnostic information that directed the user to the exact statement that inhibited vectorization. Additionally, for routines where vectorization overhead would surpass the benefits, information was provided indicating this result. In general, the following conditions inhibit vectorization [3]:

- obsolete conditionals (three branch `if`'s, assigned and computed `GOTO`'s),
- backward branches (besides the loop itself),
- directives or command line options that suppress vectorization,
- branches into the loop from outside (also violates the ANSI standard),

- dependencies (recurrence and ambiguous subscripting).

Also, references to external code (functions, intrinsics, or subroutines) often cannot be vectorized, including

- I/O statements (generate library calls),
- references to functions without vector versions,
- references to external functions or subroutines that are not expanded inline, and
- **RETURN**, **STOP**, or **PAUSE** statements, as library calls are generated.

Overall, vectorizing the code proved to be nearly trivial due to this vectorizing diagnostic output. To best explain this process, consider the following example.

```
do i = 1,n
  a(i) = b(i) + c(i)
end do
```

Most of the loop constructs in the code matched this example. It is obvious that this fragment has no data dependencies between loop iterations, and may be easily vectorized. The `cft77` pass recognizes this, and inserts a directive automatically that informs the code generator to replace the loop with the appropriate vector constructs.

```
CDIR@ IVDEP
      do i = 1,n
        a(i) = b(i) + c(i)
      end do
```

This entire process was accomplished without user intervention or directives. However, situations were encountered where vectorization was possible, but `cft77` was unable to analyze the dependencies.

```
      do i = 1,n
        a(i * 4, j) = a(i * 4, k - 1)
     &                + b(j)
      end do
```

In this fragment, there is a potential dependency on the a array. Depending on the values of the variables `j` and `k`, an *unknown* dependency may exist (flow, antidependence, output, or no dependencies may exist depending on the values of the variables `j` and `k`). The automatic vectorization skips this loop, indicating the line number of the problem, and that there is a recurrence on `a`. Again, given proper values of the variables `j` and `k`, it may be completely safe to vectorize the loop. In this case, the user may insert a directive to indicate to `cft77` that the loop may be safely vectorized.

```
CDIR$ IVDEP
      do i = 1,n
        a(i * 4, j) = a(i * 4, k - 1)
     &                + b(j)
      end do
```

It is very important to thoroughly analyze these cases, blindly inserting the `CDIR$ IVDEP` directive on loops that truly contain dependencies will negatively affect the results of the operation.

Of the remaining unvectorized loops, a portion could be rewritten to eliminate dependencies and the remainder were deemed either insignificant in the overall execution time of the code or had recurrences that were not possible to address and were thus ignored.

Vectorization proved quite effective in reducing the overall runtime of the code. Reductions of over an order of magnitude were seen in some loops, resulting in nearly a factor of five reduction overall. However, in a nested looping construct, only the innermost loops may be vectorized. The Jacobian and preconditioner formation routines contained many such structures, and continued to dominate the execution time with vectorized inner loops.

The logical progression of this study suggested that the enclosing loops in these routines should be uniformly divided and executed on multiple processors. The `fmp` compilation pass recognizes Cray autotasking directives. In the absence of dependencies this pass will transparently execute multiple threads of the outer loop on multiple processors. A dependency analyzer, `fpp`, will insert the directives in a similar manner to the vectorizing pass. The `fpp` pass is invoked using the `-ZP` compile option; this option also invokes full vectorization (making the `-ZV` option redundant).

It was quickly discovered that very few of the code's outer loops were parallelized automatically. This automatic feature is defeated by potential dependencies in a similar manner to vectorization. Similarly, function and subroutine calls inhibited parallelization (the loops that were vectorized seldom contained user calls, the outer loops that could benefit by parallelization often called user routines). In cases in which it was apparent that potential dependencies were not of concern, directives were employed to force parallelization. The following example illustrates use of the autotasking directives.

```
do i = 1,n
```

```
         call routine(args)
         call another(args)
c        multiple vector loops
      end do
```

Given that there are no dependencies between successive i values in the loop construct above (either in ordering or data dependencies), CMIC$ directives may be used to parallelize the loop.

```
CMIC$ DO ALL SHARED(args)
CMIC$1 PRIVATE(args)
CMIC$2 NUMCHUNKS(arg)
      do i = 1,n
         call routine(args)
         call another(args)
c        multiple vector loops
      end do
```

The directive is more complicated than those used for vectorization. The CMIC$ segment informs the compiler that the line is an autotasking directive, where CMIC$1–CMIC$*n* signify continuation lines. The DO ALL construct informs the compiler to execute the following loop in parallel, dividing the n loop iterations into threads on multiple processors. The SHARED(args) statement informs the compiler that the variables args are shared among threads (and the serial region above and below the parallel loop). The argument args is a comma separated list of the variables (and arrays). The PRIVATE(args) keyword denotes those variables that are local to each thread (each thread has variables args that do not

share common locations in memory between threads). Finally, the **NUMCHUNKS(arg)** keyword determines the subdivision of the loop span $1, n$. The compiler attempts to break the span into **arg** portions of roughly equal size, assigning each contiguous chunk to a processor.

The above example illustrates (or suggests) the use of subroutines and functions inside of parallel regions. This can be dangerous, depending on how the called routines are implemented. In fact, the Cray *CF77 Optimization Guide* [3] warns against calling user-level routines from inside parallel regions, but does not thoroughly explain the dilemma. When calling routines, the arguments to the function (or subroutine) are scoped according to the **PRIVATE** or **SHARED** declarations in the autotasking directive. This remains the case inside the function (recall that FORTRAN variables are passed via reference). Inside the function, all data scoped internal to the function is treated as local to the thread. All data in common is scoped as **SHARED** unless the common declaration is explicitly declared **TASKCOMMON**, meaning the data is local to the thread of execution. In reality, well written modular code should be safe to call from within parallel regions. Given that the call arguments are typed correctly (**SHARED** or **PRIVATE**, as applicable), the only possible problem that could be encountered is using local variables in a common data block. The converse of this argument, the use of locally scoped globally shared variables, may only be accomodated by passing a reference to the global data through a call argument. This method is perfectly safe assuming correct scoping in the calling routine. The first case may occasionally be encountered in some applications, however, one would not typically use local variables in common as it is a poor programming practice, regardless of whether the code is destined for a parallel or for a serial environment. Furthermore, the use of common data is no longer necessary, as FORTRAN 90, C, and enhancements to FORTRAN 77 support structure-based data storage. Thus, if the original serial code was written properly (as was the compressible flow code used for this

study), parallelization across function and subroutine calls should be trivial.

The Cray C90 series machines have an appreciable amount of parallel overhead. This overhead is attributable to several sources.

- Semaphore wait time. At the end of a parallel region, all threads must synchronize prior to the main thread continuing. If the main thread finishes last, this time is zero. However, if not, the main thread must wait (depending on the load balance among the threads) an amount of time for all child threads to synchronize at the exit point.
- Extra autotasking code. Executable code is added by the autotasking mechanism to create and manage the multiple threads. Some of this is executed by the master thread prior to *forking* the child threads and leads to additional serial overhead.
- Increase or creation of memory bank contention among processors and vector units. Clearly, if a contention situation exists on a single processor, it will be greatly compounded with multiple CPUs.
- Decrease in vector performance. If parallelism is implemented in a manner such that the vector length is shortened or chaining is prevented, the vector unit(s) efficiency is reduced.

On average, autotasking startup and executing the extra autotasking code on a dedicated machine requires 3600 clock periods on the Y-MP C90 [3].

Memory contention generally will significantly reduce the performance of a parallel code region. Consider a simplified example of a hypothetical code on the Cray architecture examined. The C90 has 256 banks of memory most efficiently addressed in FORTRAN using column-major indexing. Given the segment,

```
do i = 1, n
  do j = 1, m
    a(i,j) = ...
  end do
end do
```

the rightmost index of the a array (j) varies faster than the leftmost (*i.e.*, row ordering). In this case, memory accesses occur sequentially along ("down") a single memory bank. However, due to the electrical characteristics of memory on the C90, access to a location in a memory bank causes that bank to be unavailable for further access for some number of clock periods. Cray [3] states that a vector load or without memory contention runs from 5 to 8 times faster than the same vector load or store with the greatest memory contention. For the above example, it is obvious that this method adds a delay of c cycles per inner loop iteration (where c depends on the bank-busy time of the system), significantly slowing the execution of the code. This problem can be easily solved by arranging the loops so that a is addressed by column.

```
do j = 1, m
  do i = 1, n
    a(i,j) = ...
  end do
end do
```

In this case, memory is accessed by spanning the banks. Each successive element access is to the next bank, eliminating the access delay. In fact on the Cray, these accesses may often be pipelined by the compiler, further improving memory access efficiency. As a side

note, arrays in the C language are addressed in row-major order instead of column-major for greatest efficiency (the mechanics are the same, the languages just define addressing models differently). The equivalent to the first example would be the correct method (i.e., most efficient) to nest the loops in C.

The above example oversimplifies the problems encountered by a large code with complex memory access patterns. To achieve highest performance, memory is often accessed via a vector load/store command that fetches data from multiple banks through a single command. This type of access also maps very well to large scientific applications that manipulate data stored in indexed arrays inside a loop construct. Consider the following algorithm.

```
do i = 1,1000
  a(i) = b(i) * c(i)
end do
```

The Cray compiler converts this construct to a sequence of vector operations (in pseudocode).

```
VLOAD V1,B
VLOAD V2,C
VMULT V3,V1,V2
VSTOR A,V3
```

For this particular loop, a vector load requires 17 clock cycles, a multiply requires 12 cycles, and a store requires 17 cycles. However, these operations may be "chained," or overlapped in time. Using chaining, a total of 41 clock periods are required to perform this loop [3].

Clearly, one can envision a situation where the load/store operations are such that chaining cannot be accomplished and no overlap between the loads may be tolerated. Consider an array stored in the Cray memory banks, as shown in Figure A.1. Note that the first element

A11	A12	A13	A14	A15	A16	A17	A18	A19	A20	A21	A22	A23	A24	A25	A26	A27	A28	A29
B11	B12	B13	B14	B15	B16	B17	B18	B19	B20	B21	B22	B23	B24	B25	B26	B27	B28	B29
C11	C12	C13	C14	C15	C16	C17	C18	C19	C20	C21	C22	C23	C24	C25	C26	C27	C28	C29
D11	D12	D13	D14	D15	D16	D17	D18	D19	D20	D21	D22	D23	D24	D25	D26	D27	D28	D29
1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19

Bank Number

Figure A.1: An array stored in Cray banked memory.

of the array "A" is stored in the first memory location in bank 1, the second element in the first location in bank 2, and so on. A vector load on the Cray allows loading of all (or part, depending on the size) of an array with a single command. For a vector load of "A," a request is made for the array, giving the starting and ending element to load. Loading the first element physically may require multiple clock periods (depending on the memory and CPU speed). Hypothetically, assume that 10 cycles are required to access memory. To load the first element then requires 10 cycles. However, each subsequent element follows the first from memory each clock cycle (recall, a significant portion of the array was requested through the vector load command). Now, further assume that a vector load of "B" immediately follows "A." In theory, these two loads could be chained such that the first element of "B" is delivered to its register the same clock cycle as the second element of "A," and so forth. As explained earlier, each bank may be accessed only once per c clock periods. As such, the loading of the element of "B" that corresponds to the same element of "A" must be delayed until c cycles have passed.

For this simple example, assuming that the array lengths of "A" and "B" are large in comparision, this overhead appears negligible. Furthermore, chaining may still occur with just a slight loss of efficiency. To consider a more complicated example, if "A" had a length

of 300 elements, the first and 257th element would both fall in the first bank. The first element of "B" would follow the last element of "A" in bank 45 (if it immediately follows "A" in the FORTRAN common declaration), and so on. To add further realism, re-name "A" to "U", "B" to "V", and further include arrays to hold the remaining state variables of the Jacobian formation problem. Finally, the storage to contain the Jacobian must also fall across the bank structure. At this point, imagine the complexity involved in performing vector loads from the state variable arrays, through the sliding stencil access function, followed by stores into the Jacobian memory locations while maintaining acceptable chaining efficiency. To further place this in perspective, recall that there are two vector units per processor, and potentially 16 processors being used on the C90, all attempting to access this memory structure concurrently. Clearly, in a realistic simulation code, relaxing memory contention for vector loads and stores to a particular bank is a very difficult problem, indeed.

A.3.2 SGI Optimization Process

The SGI Onyx optimization process was initiated near the end of the Cray study. A quick examination of the scalar behavoir of the code using `prof -pixie` verified that the same routines significant in CPU usage on the Cray were also significant on the SGI. It was quickly concluded that any work to enhance parallelization on the Cray also benefited the SGI.

The SGI is a cached, superscalar superpipelined architecture in contrast to the Cray's vector-based architecture. As such, user level vectorization was not possible on the SGI. In fact, to best use the SGI architecture, programs should not be based on vector efficiency, but on data locality methods. An extensive study to best match the serial behavior of the code to the cache-based SGI was not performed. Additionally, it was desireable to maintain excellent performance with a single copy of the code on both the SGI and Cray; any SGI scalar

optimizations could potentially impact the Cray vector performance of the code. However, a cursory study with the base code versus the Cray vectorized version on the SGI showed few differences that could be attributed to efforts to better vectorize the Cray version. Thus, this version was used "as is" on the SGI. The compiler options that provided best serial performance on the SGI were,

```
f77 -non_shared -jmpopt -O2 -mips2 -Olimit 2000 -Wo,-loopunroll,8
```

where

`f77`	invokes the FORTRAN compilation system,
`-non_shared`	links to the system archive libraries instead of shared dynamic libraries,
`-jmpopt`	attempts to load the delay slot with instructions,
`-O2`	invokes full scalar optimization,
`-mips2`	uses the MIPS2 instruction set (this set matches the Onyx hardware),
`-Olimit 2000`	increases optimization limit to 2000 basic blocks (at this level, all routines were fully optimized),
`-Wo`	passes the following comma separated option to the optimizer, and
`-loopunroll,8`	allows recursive inlining to eight levels. This level included all routines that benefited from inlining in the code.

SGI provides a similar parallel analyzer to the `fpp` tool on the Cray. This tool allegedly detects parallel regions and inserts the proper directives to multitask the code. However, this claim could not be substantiated as the analysis tool `pfa` was not available on the Onyx

chosen for this study. However, use of the **-mp** option in addition to those specified above allowed directive assisted parallelization in a similar manner to that employed on the Cray.

```
C$DOACROSS SHARE(args),
C$& LOCAL(args),
C$& MP_SCHEDTYPE=SIMPLE
      do i = 1,n
         call routine(args)
         call another(args)
c        multiple vector loops
      end do
```

The **C$** segment informs the compiler that the line is a parallel directive, where **C$&** signifies a continuation line. The **DOACROSS** construct informs the compiler to execute the following loop in parallel, dividing the n loop iterations into threads on multiple processors. The **SHARE(args)** statement informs the compiler of the variables that are shared among threads (and the serial region above and below the parallel loop). The argument **args** is a comma separated list of the variables (and arrays) that are shared. The **LOCAL(args)** keyword denotes those variables that are local to each thread (each thread has variables **args** that do not share common locations in memory among threads). On the SGI, the **SHARE** and **LOCAL** directives behave identically in form and function to the **SHARED** and **PRIVATE** declarations on the C90. Finally, the **MP_SCHEDTYPE=SIMPLE** keyword determines the subdivision of the loop span $1, n$. The compiler attempts to break the span into portions of roughly equal size (based on the number of processors available), assigning each contiguous chunk to a processor. There are other options provided (in addition to **SIMPLE**), but they incur greater parallel overhead

and were not useful for any of the parallel regions studied. Other than the directive syntax, the behavior of the directives were very similar between the SGI and Cray.

Aside from those issues specific to vectorization, the SGI suffers from the same parallel overhead concerns that afflicted the Cray. The SGI uses a similar interleaved memory design, however, all memory accesses are via a local processor cache. The Cray uses a crossbar switch for memory bank access, while the SGI uses a proprietary bus. It is conceivable that fewer memory contention difficulties would be encountered on the SGI due to the fewer processors, local processor cache, and the slower processor speed of the machine.

One must access array data in a column-major form on the SGI as was performed on the Cray to minimize memory contention, but for a different reason. On the SGI (and likely most RISC cache-based machines), column addressing addresses along a cache line, where row addressing addresses across cache lines. Clearly, the column addressing along a cache line results in much better data locality and will likely increase the cache hit ratio for a given loop (unless all data accessed fits in the cache).

Again, the above example illustrates but one consideration that must be examined for optimal performance. There appears to be a strong potential that caching of the working set of data in each of two processor caches is sufficient to overwhelm parallel overhead and contention, and results in an improved cache hit ratio over the single processor case provided the cache hit effectiveness in the single cache is less than optimal. To better illustrate this behavior, consider the idealized example shown in Figures A.2 and A.3.

Consider the Jacobian formation routine used in this study. To form the Jacobian in a single processor environment, all data read from or written to main memory must pass through the processor cache. Given a Jacobian update that accesses memory locations based on a loop index covering the values from $1, \ldots, n$ (*i.e.*, array addressing), all elements in

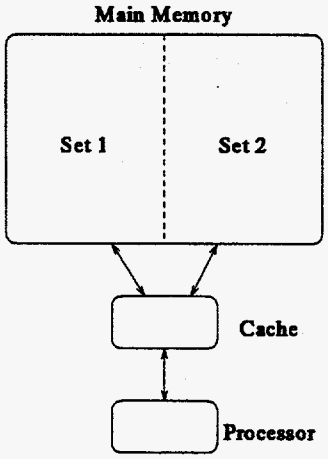

Figure A.2: Single processor memory access.

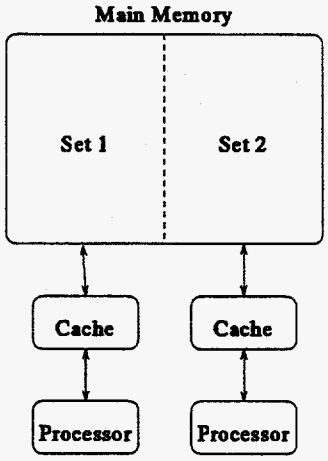

Figure A.3: Two processor memory access.

the array from $1, \ldots, n$ will eventually occupy the cache. For simplicity sake, consider this range to be divided into two sets, the first consisting of $1, \ldots, n/2$ elements, with the second spanning $(n/2+1), \ldots, n$ elements. Assuming a sequential access model, the processor will operate first on set 1, then on set 2 (as seen in Figure A.2). Furthermore, consider the case where sets 1 and 2 are sufficiently large that only one may be resident in cache. In this case, the first set is read to cache, operated on, written back, followed by an identical set of operations on the second set of data, each Jacobian update operation.

Now, consider the identical arrangement, but with two processors and private caches (see

Figure A.3). With this example, each of the two sets may occupy a processor cache throughout the entire update operation. If the entire Jacobian formation consists of many such loops over the same data ranges accessing the same data elements, the increase in efficiency in using two processors is clearly much greater than simply a factor of two greater processing capability as all cache migration is eliminated. This idealized example is oversimplified, but nicely explains how a superlinear speedup may be achieved on a cache-based multiprocessor.

A.4 Parallel Processing In A Production Environment

This study has presented a large amount of experimental results on the performance and scalability of parallel algorithms on shared-memory parallel architectures. In theory, one hopes for a speedup of m from a parallel code executed on m processors. Realistically, without superlinear cache effects, it will never be possible to achieve this level of performance. To best gauge the effectiveness of mapping a parallel algorithm onto a particular architecture, it is necessary to have an upper bound on the maximum realistic performance that may be expected.

For the remainder of this discussion, the only architecture considered will be based on real addressing (like the Cray), for simplicity. An expression for Amdahl's law may be developed that better describes effective speedup in a realistic environment [3]

$$S_r = \frac{W_{\text{production}}}{f_s + \frac{f_p(1+O)}{\min(N_{\text{sys}}, N_{\text{use}})}}, \tag{A.1}$$

where

S_r maximum realistic speedup,

f_p parallel fraction of program,

f_s serial portion of program $(1 - f_p)$,

O parallel overhead,

$W_{\text{production}}$ weighting factor for a production environment,

N_{sys} avg. number of processors available from the system, and

N_{use} avg. number of processors usable by the code.

Avg. processors	80% parallel code	20% parallel code
8	2.47	1.12
6	2.47	1.12
4	2.21	1.11
2	1.44	1.04
1.5	1.17	1.00
1.25	1.01	-1.04
1.1	-1.09	-1.06
1	-1.18	-1.08

Table A.4: Speedups within a production environment (Table from Cray Research [3]).

Experimental evidence has shown that for a particular workload that can make use of 5 processors ($N_{\text{use}} = 5$), that $W_{\text{production}} \approx 0.95$ and $O \approx 0.15$. These values for various numbers of processors available and different parallel percentages is indicated in Table A.4 for this workload. To provide an upper bound on a machine with 8 processors available, an application DOP of 8 processors, 100% parallel code, and the above efficiencies, the realistic speedup is 6.6 using the Amdahl formula. Comparing this result to the findings of this study quickly indicates that scalable performance of the pseudo-transient matrix-free model problem solution was achieved.

This section briefly touches on one reasonable scalability metric that can be employed in a production environment. There are many others equally satisfactory metrics. In a

production environment, the best metric is often performance per dollar. In the limit, the best application to perform a simulation (be it serial or parallel) achieves the highest performance for the lowest cost within the time scale and accuracy requirements of the task. This study, aside from the above discussion on hardware selection for computational needs, considered only performance scalability.

Appendix B

Sample Cray FLOWTRACE Output

Flowtrace Statistics Report
Showing Routines Sorted by CPU Time (Descending)
(CPU Times are Shown in Seconds)

Routine Name	Multi?	Tot Time	# Calls	Avg Time	Percentage	Accum%	
MIVMLDD	N	2.83E+01	5632	5.02E-03	20.87	20.87	*****
VMOM	Y	2.60E+01	6	4.33E+00	19.17	40.05	****
UMOM	N	2.37E+01	6	3.95E+00	17.48	57.52	****
TEMP	Y	1.39E+01	6	2.32E+00	10.29	67.82	**
BLDBLK2	N	8.94E+00	48	1.86E-01	6.60	74.42	*
CONT	Y	8.70E+00	6	1.45E+00	6.42	80.84	*
PRECNDD	N	5.26E+00	48	1.10E-01	3.88	84.72	
MATMUL	N	4.15E+00	1035	4.01E-03	3.07	87.79	
PRNT	N	3.23E+00	2	1.61E+00	2.38	90.17	
DIAG	N	3.01E+00	1	3.01E+00	2.22	92.39	
EXTRCV	N	2.49E+00	5632	4.42E-04	1.84	94.23	
ASSMBV	N	2.47E+00	5632	4.38E-04	1.82	96.05	
PRINT	N	1.79E+00	11	1.63E-01	1.32	97.37	
OUT	N	1.40E+00	1	1.40E+00	1.03	98.40	
STRMBFS	N	6.55E-01	1	6.55E-01	0.48	98.89	
VECADD	N	5.75E-01	3779	1.52E-04	0.42	99.31	
PRECOND	N	1.78E-01	6	2.96E-02	0.13	99.44	
COMPNS	N	1.72E-01	1	1.72E-01	0.13	99.57	
MINVMAS	N	1.55E-01	704	2.21E-04	0.11	99.68	
ORDER	N	1.22E-01	1	1.22E-01	0.09	99.77	
PQMRCGSL	N	1.07E-01	6	1.78E-02	0.08	99.85	

SOLV	N	8.40E-02	6	1.40E-02	0.06	99.91
VECMUL	N	7.35E-02	686	1.07E-04	0.05	99.97
UPDAT	N	1.54E-02	6	2.57E-03	0.01	99.98
ANULL	N	1.14E-02	6	1.89E-03	0.01	99.99
INPUT	N	8.55E-03	1	8.55E-03	0.01	99.99
MINVMUL	N	3.36E-03	704	4.77E-06	0.00	100.00
ANULL@95	Y	1.36E-03	5	2.72E-04	0.00	100.00
SETCONST	N	1.12E-03	1	1.12E-03	0.00	100.00
ROWL	N	5.13E-04	8	6.41E-05	0.00	100.00
SOLV@278	Y	2.52E-04	2	1.26E-04	0.00	100.00
PRECOND@710	Y	2.47E-04	16	1.54E-05	0.00	100.00
ETIME	N	1.77E-04	31	5.72E-06	0.00	100.00
INITLZ	N	1.40E-04	1	1.40E-04	0.00	100.00
ROW	N	1.01E-04	1	1.01E-04	0.00	100.00
COMPNS@270	Y	1.00E-04	18	5.56E-06	0.00	100.00
UPDAT@197	Y	9.44E-05	1	9.44E-05	0.00	100.00
ILOCSET	N	1.90E-05	1	1.90E-05	0.00	100.00
SETIFAC	N	1.24E-05	6	2.07E-06	0.00	100.00
GRID	N	4.07E-06	1	4.07E-06	0.00	100.00
Totals		1.35E+02	24065			

Flowtrace Statistics Report
Showing Routines Sorted by InLine Factor
(CPU Times are Shown in Seconds)

InLine Factors Greater Than 1 May Indicate Candidates for InLining.

Routine Name	Multi?	Tot Time	# Calls	Avg Time	Percentage	Accum%	InLine Factor	
NOTE: No routines had an inline factor greater than 1.								
MINVMUL	N	3.36E-03	704	4.77E-06	0.00	0.00	0.61	
VECADD	N	5.75E-01	3779	1.52E-04	0.42	0.43	0.10	
ASSMBV	N	2.47E+00	5632	4.38E-04	1.82	2.25	0.05	
EXTRCV	N	2.49E+00	5632	4.42E-04	1.84	4.09	0.05	
VECMUL	N	7.35E-02	686	1.07E-04	0.05	4.14	0.03	
ETIME	N	1.77E-04	31	5.72E-06	0.00	4.14	0.02	
COMPNS@270	Y	1.00E-04	18	5.56E-06	0.00	4.14	0.01	
MINVMAS	N	1.55E-01	704	2.21E-04	0.11	4.26	0.01	
SETIFAC	N	1.24E-05	6	2.07E-06	0.00	4.26	0.01	
MIVMLDD	N	2.83E+01	5632	5.02E-03	20.87	25.13	0.00	*****
VMOM	Y	2.60E+01	6	4.33E+00	19.17	44.30	0.00	****
UMOM	N	2.37E+01	6	3.95E+00	17.48	61.78	0.00	****
TEMP	Y	1.39E+01	6	2.32E+00	10.29	72.07	0.00	**
BLDBLK2	N	8.94E+00	48	1.86E-01	6.60	78.67	0.00	*
CONT	Y	8.70E+00	6	1.45E+00	6.42	85.09	0.00	*
PRECNDD	N	5.26E+00	48	1.10E-01	3.88	88.97	0.00	
MATMUL	N	4.15E+00	1035	4.01E-03	3.07	92.04	0.00	
PRNT	N	3.23E+00	2	1.61E+00	2.38	94.43	0.00	
DIAG	N	3.01E+00	1	3.01E+00	2.22	96.64	0.00	
PRINT	N	1.79E+00	11	1.63E-01	1.32	97.96	0.00	
OUT	N	1.40E+00	1	1.40E+00	1.03	99.00	0.00	
STRMBFS	N	6.55E-01	1	6.55E-01	0.48	99.48	0.00	
PRECOND	N	1.78E-01	6	2.96E-02	0.13	99.61	0.00	
COMPNS	N	1.72E-01	1	1.72E-01	0.13	99.74	0.00	

ORDER	N	1.22E-01	1	1.22E-01	0.09	99.83	0.00
PQMRCGSL	N	1.07E-01	6	1.78E-02	0.08	99.91	0.00
SOLV	N	8.40E-02	6	1.40E-02	0.06	99.97	0.00
UPDAT	N	1.54E-02	6	2.57E-03	0.01	99.98	0.00
ANULL	N	1.14E-02	6	1.89E-03	0.01	99.99	0.00
INPUT	N	8.55E-03	1	8.55E-03	0.01	100.00	0.00
ANULL@95	Y	1.36E-03	5	2.72E-04	0.00	100.00	0.00
SETCONST	N	1.12E-03	1	1.12E-03	0.00	100.00	0.00
ROWL	N	5.13E-04	8	6.41E-05	0.00	100.00	0.00
SOLV@278	Y	2.52E-04	2	1.26E-04	0.00	100.00	0.00
PRECOND@710	Y	2.47E-04	16	1.54E-05	0.00	100.00	0.00
INITLZ	N	1.40E-04	1	1.40E-04	0.00	100.00	0.00
ROW	N	1.01E-04	1	1.01E-04	0.00	100.00	0.00
UPDAT@197	Y	9.44E-05	1	9.44E-05	0.00	100.00	0.00
ILOCSET	N	1.90E-05	1	1.90E-05	0.00	100.00	0.00
GRID	N	4.07E-06	1	4.07E-06	0.00	100.00	0.00
Totals		1.35E+02	24065				

Flowtrace Environment Report

ORIGINAL USER EXECUTION

User Program Run on 11/04/94, at 15:19:22
Machine Serial Number 4809, a CRAY Y-MP C90,
with a Clock Speed of 4167 Picoseconds
Program was running in C90 mode.
Program was running MULTI-TASKED.
High-water mark for the stack was 113322130 (octal words)
or 19768408 (decimal words).
Original Executable Filename = ./Newt.x
Timestamp for this File: Fri Nov 4 15:18:54 1994

FLOWTRACE STATISTICS

Hashing Efficiency = 90.90%, using 40 Hashing Points
To store 44 Routine Names.
Hash Points with .gt. 1 name = 3.
Flowtrace Heap Usage was 1750 (octal words)
or 1000 (decimal words).
The size of the heap was increased once as Flowtrace was executing.

Flowtrace overhead for:

* ALL processing was 93451584 (clocks)
or 3.89E-01 (seconds).
Clocks/call was 3882
Seconds/call was 1.62E-05.
* ENTER processing was 62306242 (clocks)
or 2.60E-01 (seconds).
* EXIT processing was 31145342 (clocks)

```
              or 1.30E-01 (seconds).
 * First-Time ENTER processing was 77222 (clocks)
              or 3.22E-04 (seconds).

                        THIS FLOWVIEW EXECUTION
                        -----------------------

Flowview version = 80.8
Raw File Being Processed by flowview Now = flow.data
```

Bibliography

[1] P. R. McHugh, D. A. Knoll, V. A. Mousseau, and G. A. Hansen. An investigation of Newton-Krylov solution techniques for low Mach number compressible flow. ASME FED Summer Meeting, Hilton Head Island, S.C., August 1995.

[2] S. Saini and D. H. Bailey. NAS parallel benchmark results 12-95. Technical Report NAS-95-021, NASA Ames Research Center, Dec 1995.

[3] Cray Research, Inc. *CF77 Optimization Guide*, SG-3773 6.0 edition, 1993.

[4] Ronald L. Panton. *Incompressible Flow.* John Wiley & Sons, 1984.

[5] F. M. White. *Viscous Fluid Flow.* McGraw-Hill, Inc., New York, 1974.

[6] R. B. Guenther and J. W. Lee. *Partial Differential Equations of Mathematical Physics and Integral Equations.* Prentice-Hall, 1988.

[7] J. B. Dongarra. Performance of various computers using standard linear equations software. Technical report, University of Tennessee/Oak Ridge National Laboratory, Dec 1995.

[8] D. A. Knoll and P. R. McHugh. A fully implicit direct Newton solver for the Navier-Stokes equations. *Int. J. Num. Meth. Fluids*, 17:449–461, 1993.

[9] P. R. McHugh and D. A. Knoll. Fully coupled finite volume solutions of the incompressible Navier Stokes and energy equations using inexact Newton's method. *Int. J. Numer. Meth. Fluids*, 19:439–455, 1994.

[10] P. R. McHugh and D. A. Knoll. Inexact Newton's method solutions to the incompressible Navier-Stokes and energy equations using standard and matrix-free implementations. In *Proc. of 11th AIAA Computational Fluid Dynamics Conference*, pages 385–393, Orlando, FL., July 1993. AIAA-93-3332.

[11] X.-C. Cai, W. D. Gropp, D. E. Keyes, and M. D. Tidriri. Newton-Krylov-Schwarz methods in CFD. In F. Hebeker, R. Rannacher, and G. Wittum, editors, *Numerical Methods for the Navier-Stokes Equations*, volume 47 of *Notes on Numerical Fluid Mechanics*, pages 17–30, Brainishwig, 1994. Vieweg Verlag.

[12] P. G. Jacobs, V. A. Mousseau, P. R. McHugh, and D. A. Knoll. Domain decomposition based preconditioning strategies for Newton-Krylov solutions of the incompressible Navier-Stokes equations. In D. E. Keyes and J. Xu, editors, *Domain Decomposition Methods in Science and Engineering Computing (Proc. of the 7th Int. Conf. on Domain*

Decomposition Methods in Scientific and Engineering Computing), Providence, RI, October 1993. American Mathematical Society.

[13] D. A. Knoll, A. K. Prinja, and R. B. Campbell. A direct Newton solver for the two-dimensional Tokamak edge plasma fluid equations. *J. Comput. Phys.*, 104:418–426, 1993.

[14] S. P. Vanka. Block-implicit calculation of steady turbulent recirculating flows. *Int. J. Heat Mass Transfer*, 28(11):2093–2103, 1985.

[15] J. W. MacArthur and S. V. Patankar. Robust semidirect finite difference methods for solving the Navier-Stokes and energy equations. *Int. J. Num. Meth. Fluids*, 9:325–340, 1989.

[16] R. W. Johnson, P. R. McHugh, and D. A. Knoll. Defect correction with a fully coupled inexact Newton method. *Numer. Heat Transfer., Part B*, 26:173–188, 1994.

[17] P. R. McHugh and D. A. Knoll. Fully implicit solution of the benchmark backward facing step problem using finite volume differencing and inexact Newton's method. In B. Blackwell and D. W. Pepper, editors, *Benchmark Problems for Heat Transfer Codes*, pages 77–87, Anaheim CA., November 1992. ASME Winter Annual Meeting. ASME HTD-Vol. 222.

[18] D. A. Knoll, P. R. McHugh, and V. A. Mousseau. Newton-Krylov-Schwarz methods applied to the Tokamak edge plasma fluid equations. In D. Keyes, Y. Saad, and D. Truhlar, editors, *Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*, pages 75–95, Philadelphia, 1995. SIAM.

[19] S. R. Idelsohn and E. Oñate. Finite volumes and finite elements: Two 'good friends'. *International Journal for Numerical Methods in Engineering*, 37:3323–3341, 1994.

[20] P. R. McHugh. *An Investigation of Newton-Krylov Algorithms for Solving Incompressible and Low Mach Number Compressible Fluid Flow and Heat Transfer Problems Using Finite Volume Discretization.* PhD thesis, University of Idaho, 1995.

[21] G. H. Golub and D. P. O'Leary. Some history of the conjugate gradient and Lanczos algorithms: 1948-1976. *SIAM Review*, 31:50–102, 1989.

[22] O. Axelsson. *Iterative Solution Methods.* Cambridge University Press, New York, 1994.

[23] R. W. Freund. Transpose-free quasi-minimal residual methods for non-Hermitian linear systems. Numerical Analysis Manuscript 92-7, AT&T Bell Laboratories, Murray Hill, NJ., July 1992.

[24] Y. Saad and M. H. Schultz. GMRES: A generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J Sci. Stat. Comput.*, 7:856–869, 1986.

[25] H. A. van der Vorst. Bi-CGSTAB: A fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 13:631–644, 1992.

[26] P. Sonneveld. CGS, a fast Lanczos-type solver for nonsymmetric linear systems. *SIAM J. Sci. Stat. Comput.*, 10:36–52, 1989.

[27] P. N. Brown and Y. Saad. Convergence theory of nonlinear Newton-Krylov algorithms. Technical Report UCRL-102434 R 1, Lawrence Livermore National Laboratory, April 1992.

[28] A. T. Chronopoulos. On the squared unsymmetric Lanczos method. *J. Comput. and Appl. Math.*, 54:65–78, 1994.

[29] X.-C. Cai. A family of overlapping Schwarz algorithms for nonsymmetric and indefinite elliptic problems. In D. E. Keyes, Y. Saad, and D. G. Truhlar, editors, *Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*, chapter 1, pages 1–19. SIAM, 1995.

[30] X.-C. Cai and Y. Saad. Overlapping domain decomposition algorithms for general sparse matrices. Preprint 93-027 93-27, Army High Performance Computing Research Center, University of Minnesota, 1993.

[31] L. F. Pavarino and M. Ramé. Numerical experiments with an overlapping additive Schwarz solver for 3-D parallel reservoir simulation. *Int. J. Super. App.*, 9:3–17, 1995.

[32] J. H. Bramble, R. E. Ewing, R. R. Parashkevov, and J. E. Pasciak. Domain decomposition methods for problems with partial refinement. *SIAM J. Sci. Stat. Comput.*, 13(1):397–410, January 1992.

[33] D. A. Knoll and P. R. McHugh. Newton-Krylov methods for low Mach number combustion. 12th AIAA CFD Conference, San Diego, CA., June 1995.

[34] P. R. McHugh, D. A. Knoll, and R. W. Johnson. Fully implicit solutions of the benchmark problem using inexact Newton's method. In B. F. Blackwell and B. F. Armaly, editors, *Computational Aspects of Heat Transfer Benchmark Problems*, pages 83–91, New Orleans, LA., November 1993. ASME Winter Annual Meeting. HTD-Vol. 258.

[35] P. R. McHugh and D. A. Knoll. Comparison of standard and matrix-free implementations of several Newton-Krylov solvers. *AIAA J.*, 32(12):2394–2400, December 1994.

[36] G. A. Hansen, V. A. Mousseau, D. A. Knoll, and P. R. McHugh. Performance of a 2-D Navier-Stokes solution algorithm using Newton-Krylov techniques on shared-memory parallel/vector hardware. In *High Performance Computing 1995*. The Society for Computer Simulation, April 1995.

[37] V. Venkatakrishnan. Preconditioned conjugate gradient methods for the compressible Navier-Stokes equations. *AIAA J.*, 29:1092–1100, 1991.

[38] V. Venkatakrishnan and D. J. Mavriplis. Implicit solvers for unstructured meshes. *J. Comput. Phys.*, 105:83–91, 1993.

[39] P. E. Bjørstad and T. Kårstad. Domain decomposition, parallel computing and petroleum engineering. In D. E. Keyes, Y. Saad, and D. G. Truhlar, editors, *Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*, chapter 3, pages 39–56. SIAM, 1995.

[40] V. Venkatakrishnan. Parallel implicit methods for aerodynamic applications on unstructured grids. In D. E. Keyes, Y. Saad, and D. G. Truhlar, editors, *Domain-Based Parallelism and Problem Decomposition Methods in Computational Science and Engineering*, chapter 4, pages 57–74. SIAM, 1995.

[41] X.-C. Cai, W. D. Gropp, D. E. Keyes, and M. D. Tidriri. Parallel implicit methods for aerodynamics. In *Proc. of the 7th Int. Conf. on Domain Decomposition Methods in Scientific and Engineering Computing*, The Pennsylvania State University, October 1993.

[42] J. N. Shadid and R. S. Tuminaro. A comparison of preconditioned nonsymmetric Krylov methods on a large-scale MIMD machine. *SIAM J. Sci. Comput.*, 15(2):440–459, March 1994.

[43] Kumud Ajmani, Wing-Fai Ng, and Meng-Sing Liou. Preconditioned conjugate gradient methods for the Navier-Stokes equations. *Journal of Computational Physics*, 110:68–81, 1994.

[44] R. Choquet, P. Leyland, and T. Tefy. GMRES acceleration of iterative implicit finite element solvers for compressible Euler and Navier-Stokes equations. *International Journal for Numerical Methods in Fluids*, 20:957–967, 1995.

[45] B. F. Blackwell and D. W. Pepper, editors. *Benchmark Problems for Heat Transfer Codes*, volume HTD-Vol. 222, Anaheim, CA., November 1992. 1992 ASME Winter Annual Meeting.

[46] R. S. Dembo, S. C. Eisenstat, and T. Steihaug. Inexact Newton methods. *SIAM J. Numer. Anal.*, 19:400–408, 1982.

[47] J. A. Meijerink and H. A. van der Vorst. An iterative solution method for linear systems of which the coefficient matrix is a symmetric M-matrix. *Mathematics of Computation*, 31(137):148–162, 1977.

[48] X.-C. Cai, W. D. Gropp, and D. E. Keyes. A comparison of some domain decomposition and ILU preconditioned iterative methods for nonsymmetric elliptic problems. J. Numer. Lin. Alg. Applic., 1994.

[49] Pau-Chang Lu. *Introduction to the Mechanics of Viscous Fluids.* Hemisphere Publishing Corporation, 1977.

[50] K. Ajmani, M. S. Liou, and R. W. Dyson. Preconditioned implicit solvers for the Navier-Stokes equations on distributed-memory machines. Technical Report NASA Technical Memorandum 106449, NASA, National Aeronautics and Space Administration, Lewis Research Center, Cleveland,OH, 44135-3191, January 1994. AIAA-94-0408 : ICOMP-93-49.

[51] L. C. Dutto, W. G. Habashi, M. Robichaud, and M. Fortin. A parallel strategy for the solution of the fully-coupled compressible Navier-Stokes equations. In M. N. Dhaubhadel, M. S. Engelman, and W. G. Habashi, editors, *Advances in Finite Element Analysis in Fluid Dynamics.* American Society of Mechanical Engineers, 1993. FED-Vol 171.

[52] W. D. Gropp and D. E. Keyes. Domain decomposition methods in computational fluid dynamics. *International Journal for Numerical Methods in Fluids*, 14:147–165, 1992.

[53] B. A. Finlayson. *The Method of Weighted Residuals and Variational Principles.* Academic Press, 1972.

[54] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow.* Hemisphere, New York, 1980.

[55] J. E. Dennis, Jr. and Jorge J. Moré. Quasi-Newton methods, motivation and theory. *SIAM Rev.*, 19:46–89, 1977.

[56] A. Mizrahi and M. Sullivan. *Calculus and Analytic Geometry.* Wadsworth Publishing Company, 1982.

[57] M. Dryja and O. B. Widlund. Domain decomposition algorithms with small overlap. *SIAM Journal of Scientific Computing*, 15:604–620, 1994.

[58] Y. Saad and M. H. Schultz. Conjugate gradient-like algorithms for solving non-symetric linear systems. *Mathematics of Computation*, 44:417–424, 1985.

[59] R. W. Freund. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput.*, 14:470–482, 1993.

[60] S. F. Ashby, T. Manteuffel, and P. Saylor. A taxonomy for conjugate gradient methods. *SIAM J. Numer. Anal.*, 27:1542–1568, 1990.

[61] T. Barth and T. Manteuffel. Variable metric conjugate gradient methods. Center for Nonlinear Studies Newsletter LALP-94-003, Los Alamos National Lab., 1994.

[62] L. V. Curfman. *Solution of Convective-Diffusive Flow Problems with Newton-Like Methods.* PhD thesis, University of Virginia, 1993.

[63] J. H. Bramble, Z. Leyk, and J. E. Pasciak. Iterative schemes for nonsymmetric and indefinite elliptic boundary value problems. *Mathematics of Computation*, 60:1–22, 1993.

[64] R. W. Freund, G. H. Golub, and N. Nachtigal. Iterative solution of linear systems. *Acta Numerica*, pages 57–100, 1991.

[65] G. Golub and J. M. Ortega. *Scientific Computing, An Introduction with Parallel Computing.* Academic Press, Inc., New York, 1993.

[66] L. Zhou. *Krylov Subspace Methods for Linear and Nonlinear Systems.* PhD thesis, Utah State University, 1993.

[67] V. Faber and T. Manteuffel. Necessary and sufficient conditions for the existence of a conjugate gradient method. *SIAM J. Numer. Anal.*, 21:352–362, 1984.

[68] R. Fletcher. *Conjugate Gradient Methods for Indefinite Systems*, volume 506, pages 73–89. Springer-Verlag, Berlin, 1976.

[69] C. H. Tong. A comparative study of preconditioned Lanczos methods for nonsymmetric linear systems. Technical Report SAND91-8240, UC-404, Sandia National Laboratories Report, January 1992.

[70] S. Ashby, T. Manteuffel, and P. Saylor. *Preconditioned Polynomial Iterative Methods, A Tutorial.* University of Colorado, Denver, CO., April 1992.

[71] K. Hoffman and R. Kunze. *Linear Algebra.* Prentice-Hall Inc., Englewood Cliffs, New Jersey, 2nd edition, 1971.

[72] R. W. Freund and N. M. Nachtigal. QMR: A quasi-minimal residual method for non-Hermitian linear systems. *Numer. Math.*, 60:315–339, 1991.

[73] R. W. Freund and N. M. Nachtigal. An implementation of the QMR method based on coupled two-term recurrences. *SIAM J. Sci. Comput.*, 15:313–337, 1994.

[74] M. Dryja and O. B. Widlund. Schwarz methods of Neumann-Neumann type for three-dimensional elliptic finite element problems. *Comm. on Pure and Appl. Math.*, XLVIII:121–155, 1995.

[75] R. E. Ewing, O. P. Ileiv, S. D. Margenov, and P. S. Vassilevski. Numerical study of three multilevel preconditioners for solving 2D unsteady Navier-Stokes equations. *Comput. Methods Appl. Mech. Engrg.*, 121:177–186, 1995.

[76] J. L. Hennessy and D. A. Patterson. *Computer Architecture, A Quantitative Approach.* Morgan Kaufmann, San Francisco, CA, Second edition, 1996.