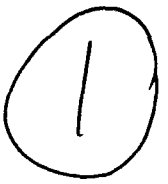
12-1993

A Numerical Study of High-Speed Missile Configurations Using a Block- Structured Parallel Algorithm

Douglas C. Blake

AFIT/GAE/ENY/93D-4



A NUMERICAL STUDY OF HIGH-SPEED MISSILE CONFIGURATIONS USING A BLOCK-STRUCTURED PARALLEL ALGORITHM

THESIS

Douglas C. Blake, Captain, USAF
AFIT/GAE/ENY/93D-4

The views expressed in this thesis are those of the author and do not reflect the official policy of position of the Department of Defense of the U. S. Government.

AFIT/GAE/ENY/93D-4

A NUMERICAL STUDY OF HIGH-SPEED MISSILE CONFIGURATIONS USING A BLOCK-STRUCTURED PARALLEL ALGORITHM

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Aeronautical Engineering

Douglas C. Blake, B.S.

Captain, USAF

December 1993

Acknowledgements

The principal undertaking of this work was to develop a computational algorithm beneficial to the fields of parallel computing and computational fluid dynamics. The work was not performed in a vacuum, and I acknowledge the contributions of several key individuals. First, I wish to thank my thesis advisor, Dr. Philip Beran, for his support, patience, and guidance. Thanks also go to the members of my thesis committee: Major Tom Buter and Lieutenant Colonel Wiliam Hobart. Major Buter provided key insight into the significance of several aspects of this work and was always frank and forthcoming in his opinions. His willingness to spend time with me is much appreciated. Lieutenant Colonel Hobart provided much-needed direction on parallel-computing issues.

Although not a member of my thesis committee, Captain Ken Moran, perhaps more than any other individual, aided immeasurably in the completion of this work. It was he who assisted in the development of the computer code, who tutored a novice in the field of Total Variation Diminishing Schemes, and who sacrificed time on his research in order to help me in mine. His knowledge and support are deeply appreciated.

I wish also to thank the many excellent instructors that I have had over the course of my educational experience, both here at AFIT and at previous institutions. It was they who prepared me for my sojourn through academia. Although I may be forgotten as their student, they will always be remembered as my teachers.

Most importantly, I wish to thank my wife. She has endured greater difficulty than I in the course of my studies and has not uttered a single complaint. I am in her debt, and to her I dedicate this work.

Douglas C. Blake

Table of Contents

List of Figures

List of Tables

Nomenclature

c_l	airfoil sectional lift coefficient
C_{m_α}	pitching moment derivative coefficient
C_d	drag coefficient
α	angle of attack
M_∞	freestream Mach number
ρ	density
u, v, w	velocity components in x, y and z coordinate directions
E_t	total energy
e	specific energy per unit mass
p	pressure
T	temperature
q	heat flux
γ	specific heat ratio
μ	kinematic viscosity coefficient
κ	coefficient of thermal conductivity
τ_{ij}	ij^{th} component of the stress tensor
E, F, G	flux vectors
$\hat{E}, \hat{F}, \hat{G}$	Harten-Yee TVD modified numerical inviscid flux vectors
$\bar{R}, \bar{S}, \bar{T}$	numerical viscous flux vectors
$\hat{i}, \hat{j}, \hat{k}$	unit vectors aligned along Cartesian xyz axes
$\hat{R}$	position vector
$\hat{n}dS$	surface vector
U	vector of conserved variables
δ_{ij}	Kronecker Delta function
J	Jacobian of transformation
V	arbitrary volume

S	surface bounding an arbitrary volume
Δt	discrete time interval
D	missile body diameter
$\|\|L\|\|_2$	L2 norm

Subscripts

x, y, z	differentiation with respect to x, y, or z (except as noted)

Superscripts

$*$	nondimensional quantity
n	time level

Abstract

A numerical analysis of the aerodynamic phenomena associated with the high-speed flight of a sharp-nosed, four-finned, high-fineness-ratio missile using a block-structured, parallel computer algorithm is presented. The Parallel Navier-Stokes Three-Dimensional (3D) Explicit Missile (PANS-3EM) algorithm utilizes a second-order-accurate, shock-capturing, Total Variation Diminishing (TVD) scheme and incorporates a Baldwin-Lomax turbulence model. PANS-3EM allows for extreme flexibility in the choice of computational domain decomposition and provides for a central "stepping-off point" for subsequent modification to run on either a shared- or distributed-memory computing machine. Developmental work consists of conceptualization and verification of the algorithm as well as parallel performance and scalability studies conducted on a variety of computing platforms.

Using PANS-3EM, the aerodynamic characteristics of the above-mentioned missile are investigated. Specifically, drag and pitching moment coefficient data are computed and compared against experimental flight data obtained from the United States Air Force Aeroballistic Research Facility at Eglin Air Force Base. Trends in the numerical data agree with experimental results reported by Eglin with the exception that an unexpected reversal of the stability characteristics exhibited by the flight-test missile at speeds in excess of Mach 3.75 are not confirmed by the computer code.

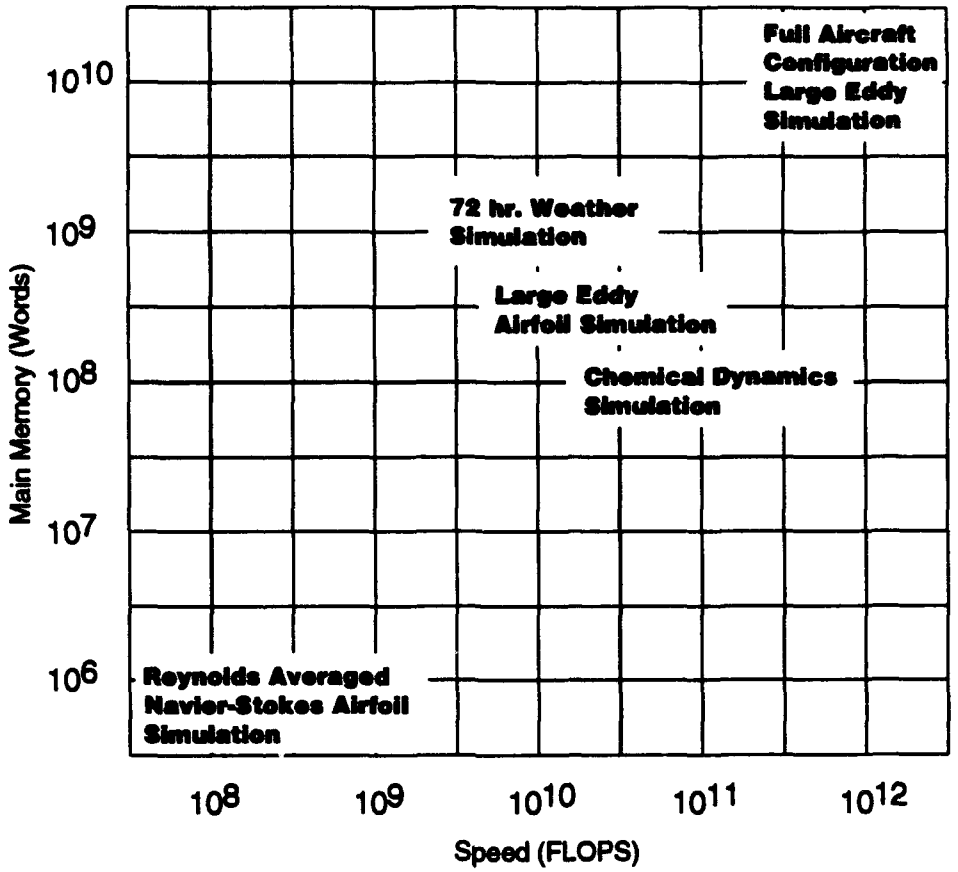
A NUMERICAL STUDY OF HIGH-SPEED MISSILE CONFIGURATIONS USING A BLOCK-STRUCTURED PARALLEL ALGORITHM

I. Introduction

1.1 Problem Motivation

Modern numerical algorithms capable of resolving complicated flow structures such as shocks and boundary layers place great demands on computing resources. Although full aircraft configuration Navier-Stokes calculations have been performed by Shang [32] and others, current generation supercomputers possess neither the speed nor the storage capacity to perform a complete aircraft model simulation including full-scale turbulence, chemical reaction, structural/aerodynamic coupling, and radar reflectivity models. For example, depending on the numerical method used, a physical discretization of between ten and twenty [6,30] node points are required to adequately resolve the smallest length scales associated with a problem. In the case of radar cross-section computations, a scale simulation of a 1 gigahertz signal illuminating a standard size U.S. fighter aircraft would require a grid containing approximately 10,000,000 node points [33]. Storage of flow field and electromagnetic variables as well as grid metrics for each node would easily exceed both the storage and computational capabilities of all current and next-generation computing machines. Furthermore, given the history of computing performance improvements inevitably leading to more ambitious projects of research, it can be safely assumed that a computing deficit will continue for the foreseeable future. Figure 1 depicts a few of the computationally intensive fluids-related problems that are of current interest.

While computing demands continue to increase, there are physical barriers to the continued speed improvements of traditional serial computing machines. Signals cannot propa-



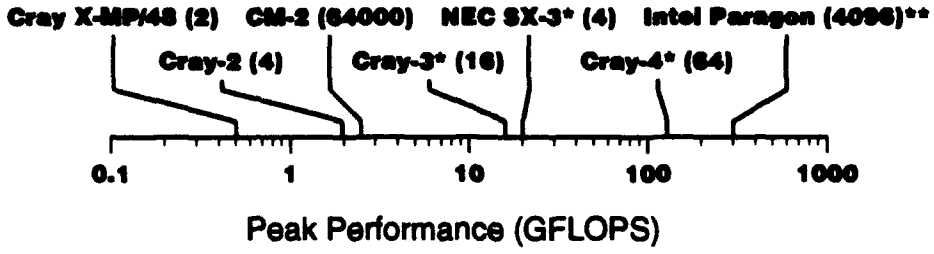
Data compiled from References [24] and [40].

Figure 1: Current and Projected Computational Requirements

gate faster than the speed of light, and thus components separated by some physical distance must necessarily have a finite response time. Reducing the physical size of the components can help, but components can only be made so small before barriers are reached in terms of minimum logic device sizes and heat dissipation requirements.

One extremely promising technology which could help to overcome these shortfalls is parallel computing. While there are many parallel computing paradigms (Single-Instruction, Single-Data (SISD); Single-Instruction, Multiple-Data (SIMD); Multiple-Instruction, Multiple-Data (MIMD); etc.[1]), all use multiple processing units to perform simultaneous calcula-

1. The reader is referred to Reference [24] for a discussion of the various parallel computing paradigms.



*estimated

**multiprocessor nodes (1024 nodes @ 4 processors/node)

Data compiled from References [9], [20], [22], and [40].

Figure 2: Supercomputer Performance

tions. These calculations can be performed on separate parts of a single problem, or they can be performed on completely unrelated problems. In either case, the effective throughput of the computer is increased since a larger amount of computational work can be performed in a given amount of time.

In addition to the many parallel computing paradigms, parallel computing can be implemented at many different performance levels. Although there are many different benchmarks for computing performance [39], a common metric is the number of floating point operations conducted in one second[1] (FLOPS). In terms of sheer computing power, supercomputers provide the highest level of performance. Until recently, the supercomputing realm has been occupied exclusively by machines with few but very powerful processing units. This trend is changing as is evident in Figure 2 which shows the capabilities of a few of the current and projected supercomputers. The number in parentheses next to each computer name denotes the maximum number of processors available on the machine. Although single-processor supercomputers are still in use, they are being replaced by multiple processor models which are capable of delivering much higher performance and throughput.

1. This number is typically measured in terms of million floating point operations per second (MFLOPS) or billion floating point operations per second (GFLOPS).

While supercomputers inarguably occupy the pinnacle of computing power, they are extremely expensive and therefore only available to a relatively small number of users. Consequently, another parallel computing approach is evolving in the *workstation-farm* concept wherein a number of physically distinct and possibly heterogeneous computing platforms are used in combination to attack a computing problem. Software tools such as Parallel Virtual Machine (PVM) [25], are designed specifically to exploit this type of parallelism. This approach has advantages of flexibility, scalability, availability, and affordability over the supercomputing approach. Furthermore, recent advances in workstation-level chip technology[1] have brought a large amount of economical computing power to the workstation level.

Despite the continued performance increase of computing machines, hardware alone is not enough to solve the computational deficit. Indeed, a powerful machine using weak algorithms and poorly designed computer codes can never reach its full potential. Smith [36] underscores the importance of software design in his hyperbole, "Is software everything? No; it's the only thing." In order to fully harness the power of parallel processing, it is necessary to develop new algorithms which lend themselves to parallelism; it is not enough to merely use sequential algorithms on parallel machines.

While the potential benefits of parallel algorithms are widely acknowledged, there are difficulties uniquely associated with parallel algorithm development. Unlike a serial machine which typically contains a single processor and a single bank of main memory, a parallel machine can be configured in a myriad of ways: from a large number of relatively low-powered processors (the Thinking Machines CM-2, for example) to a smaller number of very powerful processors (the Cray Y-MP, for example). Furthermore, the memory architecture can be *distributed* such that each processor contains a bank of local memory, or it can be *shared* whereby all processors access a single global memory bank. In certain instances such as the Thinking Machines CM-5, a single machine can utilize both shared and distributed memory [7]. All of these architectures pose interesting and challenging problems for the would-be parallel programmer.

---

1. Examples include the DEC Alpha AXP, MIPS R4400 and Sun SuperSparc processors.

1.2 Parallel Computing and Computational Fluid Dynamics (CFD)

The applicability of parallel computing to CFD was so obvious from the early inceptions that the *CFD programming language* was developed by the NASA Ames Research Center for use on the first SIMD parallel computer, the Illiac IV [18]. More recently, several researchers have demonstrated the performance gains possible in CFD computations using parallel computing on architectures as varied as shared-memory MIMD machines [29] and distributed-memory massively parallel MIMD machines [14]. However, given the difficulties associated with parallel algorithm development coupled with the fact that very powerful vector machines have been very pervasive in the scientific community for quite some time, the trend to embrace parallel computing for CFD applications has been slow.

A reexamination of Figure 2 may provide some insight into the future direction of parallel computing and CFD. In the past, the existence of powerful vector machines like the Crays tended to focus algorithm development work on codes designed to be highly vectorizable. Similarly, there is a synergistic relationship between the introduction new and powerful, massively-parallel machines like the Paragon and the development of algorithms which exploit the inherent parallelism in problems of interest. The tie between CFD and parallel computing, therefore, can only become stronger.

1.3 Numerical Algorithm Overview

Given the potential benefits of parallel computing and the demonstrated need for flexibility and speed in solving CFD problems, this study focuses on the development of an algorithm to simulate flow fields about high-speed missile configurations in a block-structured, parallel fashion. The primary objectives are twofold and of equal importance: 1) to develop an algorithm which decomposes the computational domain into a set of *subdomains* or *blocks* and then solves the governing equations in parallel over the blocks, and 2) to utilize the algorithm to investigate the aerodynamic characteristics of a sharp-nosed, four-finned, high-speed missile.

The numerical algorithm used to solve the governing equations is a second-order-accurate Total Variation Diminishing (TVD) scheme developed by Harten [16] and modified by

Yee [47]. It is an extension of the basic first-order-accurate Roe scheme [31], and achieves its second-order accuracy by appropriately modifying the numerical inviscid flux terms and extending the Roe scheme's three-point computational stencil (in the one-dimensional case) to include an additional two points.

Moran [26] has implemented the Harten-Yee TVD scheme in a computer code designed to study the high-speed flight of missiles. This code provided a baseline which was modified to incorporate domain decomposition and ultimately resulted in PANS-3EM.

1.4 Domain Decomposition

The Harten-Yee TVD scheme discussed in Section 1.3 is iteratively applied at the node points of the computational grid surrounding the missile configuration until a flow-field solution of desired accuracy is obtained. To increase the efficiency of the algorithm, it is possible to decompose the computational domain into subdomains or blocks so that calculations can be conducted simultaneously over the subdomains. Thus, it is beneficial to explore approaches by which the computational domain can be decomposed.

Several methods of domain decomposition have been examined by various researchers. Hammond and Barth [15] used a finite-volume approach which assigned each vertex of the computational mesh to a computer processor. Swisshelm, Horten, Alef, and others [41,19,1,40] utilize multigrid techniques which discretize the computational domain into a series of *embedded* grids which vary in coarseness and are useful for accelerating convergence and damping certain wave numbers of the solution which create unwanted numerical oscillations in regions of abrupt flow changes. Yadlin, Hauser, and Furukawa use a blocking or zonal decomposition [46,17,11] that divides the computational domain into a series of logically rectangular blocks which may or may not overlap. Parallelism occurs by advancing the solution simultaneously over each of the blocks. In all of these schemes, some type of synchronization is usually required among the parallel processes to ensure that the numerical solution remains stable.

Of the domain decomposition methods discussed here, the block structured method was best suited to the framework of Moran's computer code since it did not require code

restructuring or grid modification for implementation. Furthermore, with future expandability in mind, the method allows for extreme flexibility in grid partitioning and solution computation. For example, using a blocked approach, it is possible to simultaneously use the less-computationally-intensive Euler equations in blocks where viscous-related phenomena are negligible while other blocks incorporate the full Navier-Stokes equations.

1.5 Missile Configuration Study

Using the Harten-Yee TVD scheme discussed in Section 1.3 coupled with the domain-decomposition approach discussed in Section 1.4, it is feasible to conduct an efficient analysis of the aerodynamic performance of high-speed missile flight. Recent experimental research conducted by the United States Air Force Aeroballistic Research Facility at Eglin Air Force Base [13,44] centers on the determination of aerodynamic performance of generic, fin-stabilized, high-speed missile configurations. Several nose/fin configurations have been investigated including sharp and blunt ogive noses with four clipped delta fins. If the missile fins are considered as flat plates[1], then linearized supersonic flow theory (see Anderson [3], Chapter 11) predicts that the lift generated by the fins is related to Mach number by

$$c_l = \frac{4\alpha}{\sqrt{M_\infty^2 - 1}} \qquad (1)$$

Since the lift generated by the missile fins decreases with increasing Mach number, the pitching moment generated by the fins must also decrease. This trend is confirmed by the Eglin experimental data through Mach numbers approaching 3.75 at which time an unexpected reversal of the pitching moment derivative (C_{m_α}) curve occurs which has not yet been satisfactorily explained. It is the objective of this study to utilize PANS-3EM to study the aerodynamic performance about the sharp-nosed missile configuration with special interest devoted to the stability trends observed for increasing Mach numbers. Moran and Beran [27] have studied the blunt-nosed configuration exhaustively, and their work provides an excellent

1. The fins used on the experimental model were formed from thin sheet metal, with no camber or twist.

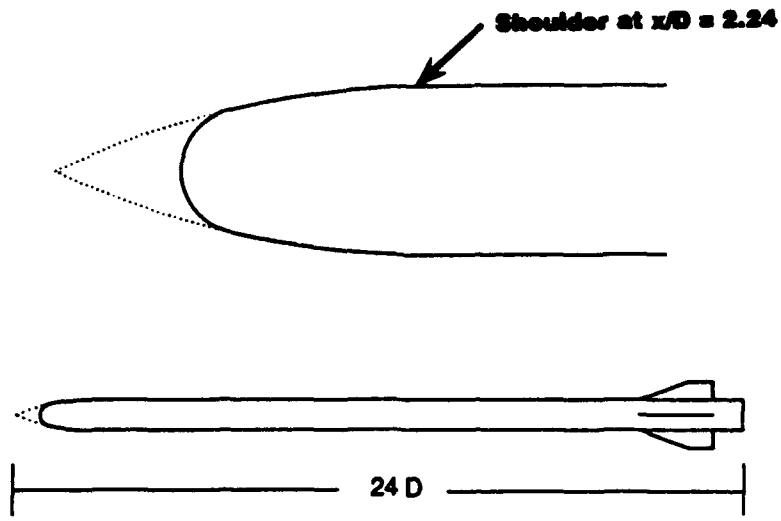


Figure 3: Missile Geometries

comparative reference to the work conducted in this investigation. Both sharp- and blunt-nosed missile configurations are depicted in Figure 3.

1.6 Document Roadmap

With the discussion of parallel computing and its applicability to CFD as a basis, Chapter 2 outlines the system of conservation laws and associated boundary conditions that govern the flow behavior in the high-speed flight regime and discusses the application of the Harten-Yee TVD scheme to the solution of those equations. Chapter 3 focuses on the specifics of the code development and the driving factors motivating the choice of the domain-decomposition technique used in this project. The numerical results obtained from the PANS-3EM code for the sharp-nosed missile are presented and discussed in Chapter 4 as are results for performance of the algorithm on a variety of computing platforms. The appendices contain supplementary information on both the numerical and code development aspects of this investigation. Appendix A contains a derivation of the nondimensional equations which govern the fluid behavior in the high-speed flight regime. Appendices B and C provide additional infor-

mation on important parallelization issues and document the changes made to Moran's code in the development of the PANS-3EM code. Finally, the source-code listings for the key domain-decomposition subroutines are found in Appendix D.

II. Numerical Algorithm

This chapter presents the equations and boundary conditions which govern the physics of the flow field around the high-speed missile. With the equations suitably expressed, an integral formulation is obtained which lends itself to solution via a finite-volume technique. A discussion of the Harten-Yee TVD algorithm which is used in conjunction with the finite-volume technique follows. Finally, the features of the algorithm which allow for parallelism are presented.

2.1 Equation Development

2.1.1 Governing Equations

The Navier-Stokes equations are statements of conservation laws of mass, momentum and energy. In conservative vector form for a three-dimensional Cartesian coordinate system, they can be stated as [2]

$$U_t + E_x + F_y + G_z = 0 \tag{2}$$

where

$$U = \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ E_t \end{bmatrix} \tag{3}$$

$$E = \begin{bmatrix} \rho u \\ \rho u^2 + p - \tau_{xx} \\ \rho uv - \tau_{xy} \\ \rho uw - \tau_{xz} \\ (E_t + p)u - \tau_{xx}u - \tau_{xy}v - \tau_{xz}w + q_z \end{bmatrix} \tag{4}$$

$$F = \begin{bmatrix} \rho v \\ \rho uv - \tau_{yx} \\ \rho v^2 + p - \tau_{yy} \\ \rho vw - \tau_{yz} \\ (E_t + p)v - \tau_{yx}u - \tau_{yy}v - \tau_{yz}w + q_y \end{bmatrix} \tag{5}$$

$$G = \begin{bmatrix} \rho w \\ \rho uw - \tau_{zx} \\ \rho vw - \tau_{zy} \\ \rho w^2 + p - \tau_{zz} \\ (E_t + p)w - \tau_{zx}u - \tau_{zy}v - \tau_{zz}w + q_z \end{bmatrix} \tag{6}$$

$$E_t = \rho\left(e + \frac{u^2 + v^2 + w^2}{2}\right) \tag{7}$$

$$\tau_{ij} = \mu\left(\frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i}\right) + 2\mu\delta_{ij}\frac{\partial u_k}{\partial x_k} \tag{8}$$

and

$$q_x = (-\kappa T)_x \tag{9}$$

$$q_y = (-\kappa T)_y \tag{10}$$

$$q_z = (-\kappa T)_z \tag{11}$$

The conservative form of the equations is used since the numerical solution to these equations is expected to contain shocks and other structures characterized by abrupt flow property changes [2, pg 51]. The vector U contains the *conserved variables* while the vectors E, F, and G are the *flux vectors*.

After nondimensionalization[1], equation 2 becomes

$$U_{t^*}^* + E_{x^*}^* + F_{y^*}^* + G_{z^*}^* = 0 \tag{12}$$

1. See Appendix A for a description of the nondimensionalization procedure.

where

$$U^* = \begin{bmatrix} \rho^* \\ \rho^* u^* \\ \rho^* v^* \\ \rho^* w^* \\ E_t^* \end{bmatrix} \tag{13}$$

$$E^* = \begin{bmatrix} \rho^* u^* \\ \rho^* u^{*2} + p^* - \tau_{xx}^* \\ \rho^* u^* v^* - \tau_{xy}^* \\ \rho^* u^* w^* - \tau_{xz}^* \\ (E_t^* + p^*) u^* - \tau_{xx}^* u^* - \tau_{xy}^* v^* - \tau_{xz}^* w^* + q_z^* \end{bmatrix} \tag{14}$$

$$F^* = \begin{bmatrix} \rho^* v^* \\ \rho^* u^* v^* - \tau_{yx}^* \\ \rho^* v^{*2} + p^* - \tau_{yy}^* \\ \rho^* v^* w^* - \tau_{yz}^* \\ (E_t^* + p^*) v^* - \tau_{yx}^* u^* - \tau_{yy}^* v^* - \tau_{yz}^* w^* + q_y^* \end{bmatrix} \tag{15}$$

$$G^* = \begin{bmatrix} \rho^* w^* \\ \rho^* u^* w^* - \tau_{zx}^* \\ \rho^* v^* w^* - \tau_{zy}^* \\ \rho^* w^{*2} + p^* - \tau_{zz}^* \\ (E_t^* + p^*) w^* - \tau_{zx}^* u^* - \tau_{zy}^* v^* - \tau_{zz}^* w^* + q_z^* \end{bmatrix} \tag{16}$$

$$E_t^* = \rho^* \left(e^* + \frac{u^{*2} + v^{*2} + w^{*2}}{2}\right) \tag{17}$$

$$\tau_{ij}^* = \frac{\mu^*}{Re}\left(\left(\frac{\partial u_i^*}{\partial x_j^*} + \frac{\partial u_j^*}{\partial x_i^*}\right) + 2\delta_{ij}\frac{\partial u_k^*}{\partial x_k^*}\right) \tag{18}$$

and

$$q_x^* = \frac{\mu^*}{(\gamma - 1) M_\infty^2 RePr} T_x^* \tag{19}$$

$$q_y^* = \frac{\mu^*}{(\gamma-1)M_\infty^2 RePr} T_y^* \tag{20}$$

$$q_z^* = \frac{\mu^*}{(\gamma-1)M_\infty^2 RePr} T_z^* \tag{21}$$

Equation 12 represents a system of five equations and seven unknowns. Thus, two additional equations are required to provide closure. These are obtained from the assumptions of a calorically and thermally perfect gas and yield the algebraic relations

$$p^* = (\gamma-1)\rho^* e^* \tag{22}$$

and

$$T^* = \frac{\gamma M_\infty^2 p^*}{\rho^*} \tag{23}$$

Since the form of the nondimensional equations is identical to that of the dimensional equations, the superscript "*" is dropped for convenience with the realization that the parameters Reynolds number, Mach number, and Prandtl number make their appearance only in the nondimensional case.

2.1.2 Finite Volume Formulation

Equation 12 can be integrated over an arbitrary volume to yield

$$\int_V U_t dV + \int_V (E_x + F_y + G_z) dV = 0 \tag{24}$$

Application of the Divergence Theorem to the second integral gives

$$\int_V U_t dV + \oint_S \vec{H} \cdot \hat{n} dS = 0 \tag{25}$$

where $\vec{H} \equiv E\hat{i} + F\hat{j} + G\hat{k}$ and S represents the surface bounding the volume, V.

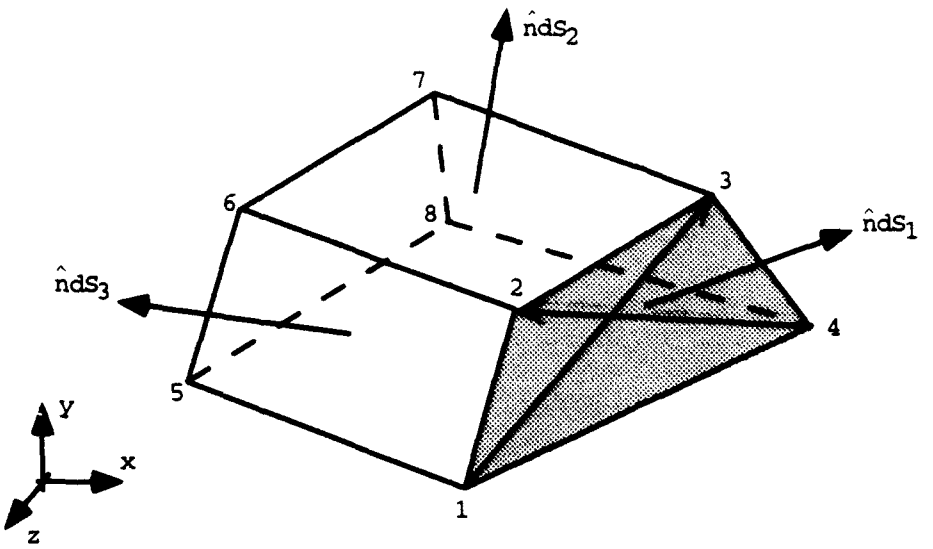


Figure 4: Finite Volume Element

If the grid is fixed in time, then

$$\frac{\partial}{\partial t}\int_V U dV + \oint_S \vec{H}\cdot \hat{n} dS = 0 \qquad (26)$$

In a finite-volume methodology, equation 26 is applied to each volumetric element. A typical such element is depicted in Figure 4. Calculation of the fluxes through faces of this element proceeds by examining the shaded face. The area of this face can be calculated by computing the cross product of the face diagonals, $\vec{R}_{31}$ and $\vec{R}_{24}$,

$$\hat{n} dS_1 = \frac{1}{2}(\vec{R}_{31}\times\vec{R}_{24}) \qquad (27)$$

where

$$\vec{R}_{31} = (x_3 - x_1)\hat{i} + (y_3 - y_1)\hat{j} + (z_3 - z_1)\hat{k} \qquad (28)$$

$$\vec{R}_{24} = (x_2 - x_4)\hat{i} + (y_2 - y_4)\hat{j} + (z_2 - z_4)\hat{k} \qquad (29)$$

Numerical subscripts on the surface vectors, $\hat{n}dS$, designate the element face while numerical

subscripts on the position vectors, $\vec{R}$, designate the element vertices.

The x-component of the numerical flux, f, through face 1 is thus given by

$$f_{1x} = E(\hat{n}dS_1 \cdot \hat{i}) \tag{30}$$

where the numerical subscript again designated the element face and the subscript x designates the flux direction. Similarly, the fluxes in the y and z-directions through this face are

$$f_{1y} = F(\hat{n}dS_1 \cdot \hat{j}) \tag{31}$$

$$f_{1z} = G(\hat{n}dS_1 \cdot \hat{k}) \tag{32}$$

Repeating this process for the remaining five faces of the volumetric element yields a total of eighteen projected area terms multiplied by an appropriate flux component. In practice, it is not necessary to compute all terms for each cell, since adjacent cells share common faces. In fact, values on three of the six faces are computed which yields nine flux components computed for each cell.

According to Vinokur [43], the volume for a general hexahedral cell can be computed as

$$V = \frac{1}{3}(\hat{n}dS_1 + \hat{n}dS_2 + \hat{n}dS_3) \cdot \vec{R}_{71} \tag{33}$$

Vinokur also notes that the quantity V is equal to the Jacobian, J.

With the cell volume determined, a first-order-accurate discretization of the temporal term and a finite volume discretization of the surface integral of equation 26 gives

$$U_{ijk}^{n+1} = U_{ijk}^{n} - \frac{\Delta t}{V} \sum_{m=1}^{6} (f_{mx}^{n} + f_{my}^{n} + f_{mz}^{n})_{ijk} \tag{34}$$

where ijk designates the volume element and m is the index of summation over the faces of the element. In obtaining equation 34, it has been assumed that the flow properties are constant over the interior of the volumetric element.

Wang [45] notes that solution to equation 34 yields the flow variables at the cell cen-

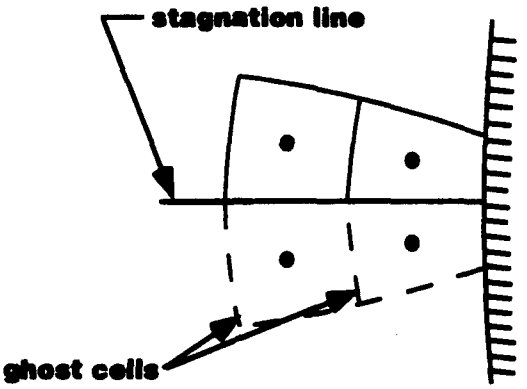


Figure 5: Ghost Cells Along Stagnation Line

ters at time level $n+1$. This, however, requires the determination of the fluxes at the cell faces. The method used in this study to compute the fluxes is the Harten-Yee TVD scheme which is discussed in Section 2.2.

2.1.3 Boundary Conditions

Boundary condition implementation is straightforward. Since the flow is supersonic at the inlet and outlet of the domain, freestream conditions are enforced at the inlet and a supersonic "no change" boundary condition is specified at the outflow plane. The no-slip condition is enforced on the missile body for solutions computed with the Navier-Stokes equations while Euler solutions use the impermeability and flow tangency conditions at the surface. The boundary condition on pressure is specified as a zero normal gradient, and the wall temperature is specified at a constant value. These boundary conditions are consistent with those utilized by Moran [27].

Specification of zero flux along the stagnation line is handled via a reflection condition through the use of *ghost cells*. In the case of zero angle of attack, circumferential and tangential velocity components are reflected across the stagnation line while pressure, density and axial velocity values are set equal. At angle of attack, an extrapolation technique is used. A sample set of ghost cells situated along the stagnation line appears in Figure 5.

2.2 Harten-Yee TVD scheme

Harten [16] proposed that the inviscid flux components of equation 2 could be modified to yield

$$U_{ijk}^{n+1} = U_{ijk}^{n} - \lambda\left(\left(\hat{E}_{i+\frac{1}{2},jk} - \hat{E}_{i-\frac{1}{2},jk}\right) + \left(\hat{F}_{i+\frac{1}{2},jk} - \hat{F}_{i-\frac{1}{2},jk}\right) + \left(\hat{G}_{i+\frac{1}{2},jk} - \hat{G}_{i-\frac{1}{2},jk}\right)\right)$$

$$+ \lambda\left(\left(\bar{R}_{i+\frac{1}{2},jk} - \bar{R}_{i-\frac{1}{2},jk}\right) + \left(\bar{S}_{i+\frac{1}{2},jk} - \bar{S}_{i-\frac{1}{2},jk}\right) + \left(\bar{T}_{i+\frac{1}{2},jk} - \bar{T}_{i-\frac{1}{2},jk}\right)\right) \qquad (35)$$

where $\hat{E}$, $\hat{F}$, and $\hat{G}$ represent the modified inviscid flux vectors, $\bar{R}$, $\bar{S}$, and $\bar{T}$ represent the viscous flux vectors, and

$$\lambda = \frac{\Delta t}{V} \qquad (36)$$

Here, V is taken as the volume of the finite volume element. The modified inviscid flux vectors are defined as

$$\hat{E}_{i+\frac{1}{2},j,k} = \frac{1}{2}\left((\xi_x)_{i+\frac{1}{2},jk}(E_{i+1,jk} + E_{ijk})\right)$$

$$+ \frac{1}{2}\left((\xi_y)_{i+\frac{1}{2},jk}(F_{i+1,jk} + F_{ijk})\right) + \frac{1}{2}\left((\xi_z)_{i+\frac{1}{2},jk}(G_{i+1,jk} + G_{ijk})\right)$$

$$+ \frac{1}{2}\left(R_{i+\frac{1}{2},jk}\beta_{i+\frac{1}{2},jk}\right) \qquad (37)$$

where the matrix R contains the eigenvectors associated with the modified wave speeds, and the term β "acts to limit the characteristic variable, thereby providing higher accuracy [27]." The terms ξ_x, ξ_y, and ξ_z are the *geometric terms* of the finite volume formulation and represent the projected areas of the element face into the x, y, and z coordinate planes.

Equation 37 calculates fluxes at a cell face by using information available at the cell

centers. This is known as a non-MUSCL approach. In contrast, a MUSCL approach calculates fluxes at cell faces by extrapolating information from the cell centers and evaluating flux values appropriately. The reader should note that the inviscid fluxes of equation 37 are upwind differenced, while the viscous components of equation 35 are centrally differenced.

2.3 *Inherent Parallelism*

Parallelism can generally be achieved when the following conditions are met [35,42]:

- The code works correctly in a serial implementation
- The code is limited by the Central Processing Unit (CPU) and not by other factors such as Input/Output (I/O), bus bandwidth, etc.
- The code contains portions which are independent of one another (no control dependence)
- The code portions which are independent do not attempt to write calculations into the same output variables (no data dependence)

The first, second, and fourth of these issues can be easily verified by examining the code to be modified. The third issue requires a more fundamental examination of the code's underlying algorithm. For the case of the TVD scheme used in this investigation, the *computational stencil* is depicted in Figure 6. This stencil is comprised of those node points that enter into the flow variable calculations at a given node point. Off-axes points in the figure are bound by the shaded box and are represented by the unfilled circles. These stencil points originate from the central differencing of the viscous terms in equation 35. Darkened circles represent those points which originate from the flux calculations of the Harten-Yee TVD scheme. These points can be decoupled through the use of *approximate factorization* [2] which results in the three separate one-dimensional stencils shown in Figure 7. These stencils are applied separately to the node points in a series of *sweeps*, one such sweep along each coordinate direction. Upon completion of the third sweep, the solution has advanced to the next time level. Examination of equation 34 reveals that the scheme is *explicit* in that calculations at a given node point use only known information from a previous time step. This explicit nature allows for a complete decoupling of the calculations at each node point. In theory, given enough processors, all node values can be calculated simultaneously. In practice, given the

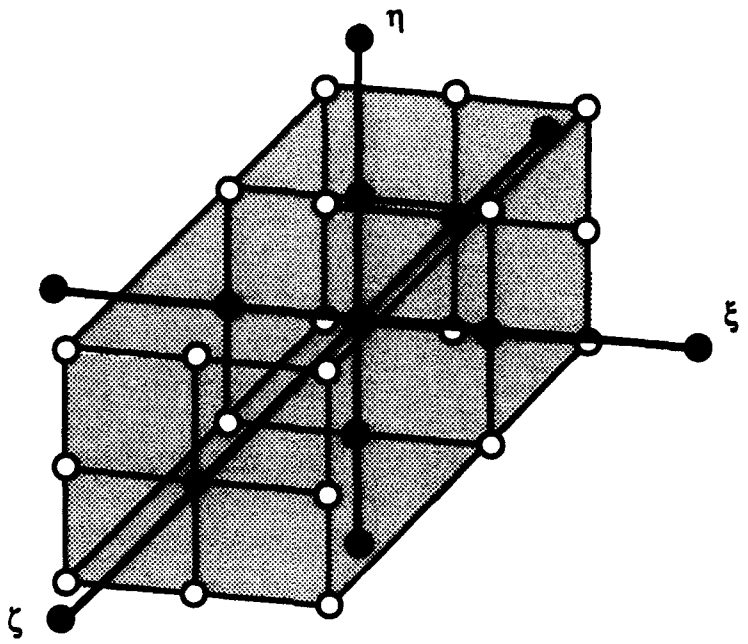


Figure 6: Computational Stencil

extremely large number of node points in a typical computational mesh, calculations are conducted in parallel over *lines* of grid points. This technique is especially amenable to implementation on a vector processor since identical calculations are performed at each node. This investigation develops an algorithm in which the calculations are performed in parallel over *blocks* of grid points. The utility of such a scheme is discussed in Chapter 3.

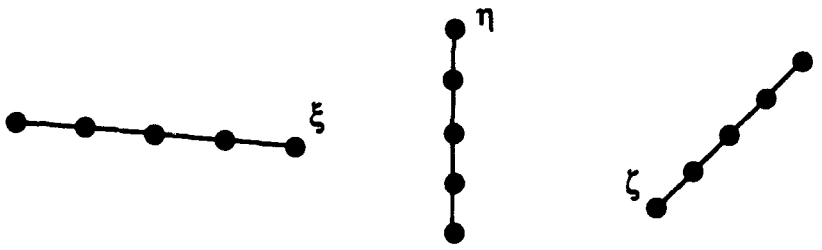


Figure 7: Decoupled Sweeps

III. Domain Decomposition

This section discusses the domain-decomposition approach used in this project. Issues affecting the decomposition approach such as load balancing and inter-domain communication are identified. General features of PANS-3EM that are of importance to future modification efforts are explained.

3.1 Background

As mentioned previously, the method of domain decomposition can take one of several forms. Since the computer code to be parallelized was already complete, validated, and in use, the chosen method of domain decomposition required easy incorporation into the framework of the existing code. Furthermore, since the target machine of implementation was not fixed, it was desired to develop a decomposition approach which was general enough to run on a variety of computing platforms with few modifications. Figure 8 shows a sketch of the objective of the PANS-3EM development process. PANS-3EM provides a stepping-off point for a subsequent modification or *port* to either a distributed- or shared-memory computing platform. As discussed in the Section 1.1, implementation on these two architectures requires very different programming approaches, and thus a code that can be modified for use on either platform provides a potential benefit in terms of reduced code development time.

A block-structured domain-decomposition approach provides for ready implementation on either platform. Furthermore, the size and number of blocks can be readily adapted to fit within the memory limitations on a given machine. This is especially useful in the case of a distributed-memory machine in which the data must be rigidly partitioned among the available processor memories.

When implementing a block-structured algorithm, two key areas must be addressed: load balance and inter-block communication.

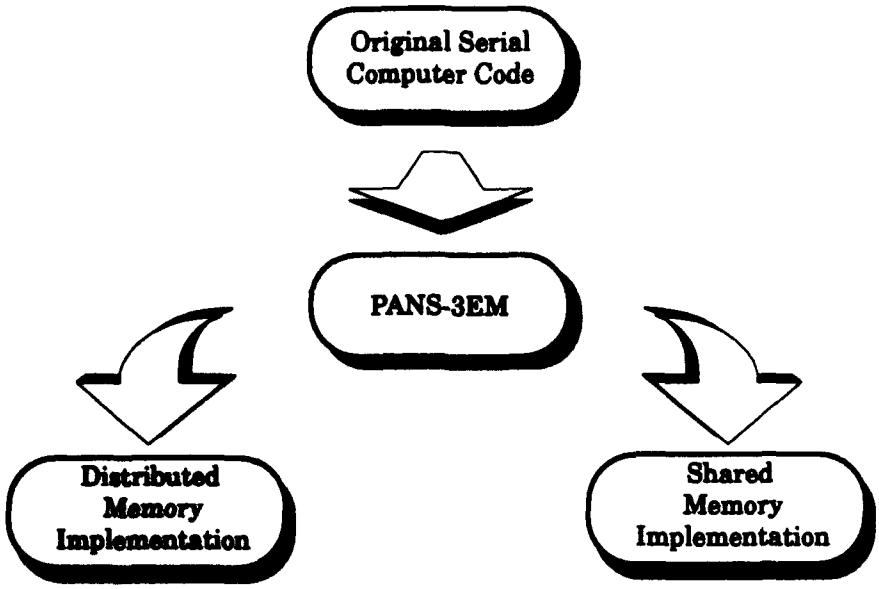
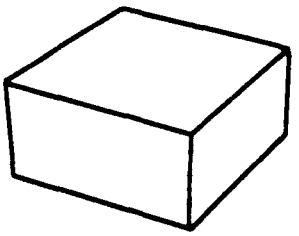


Figure 8: Code Development Model

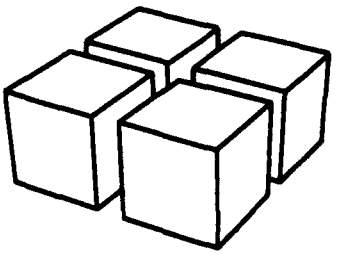
3.1.1 Load Balance

In a parallel code execution, the run time of the program can be no faster than that of the slowest process. Thus, in order to achieve optimal *turnaround time*[1], the work load must be divided among the processors such that all processors share as near equal portion of the computational task as possible. For the TVD scheme used in this investigation, the amount of computational effort required at each node is approximately equal. Thus, in order to achieve a proper load balance, the number of nodes assigned to each subdomain must be approximately equal. Given the most general block decomposition scenario, it may be possible to partition the domain such that each block contains very nearly the same number of node points. However, in order to speed development of PANS-3EM, the decomposition process was performed such that no block was allowed to have more than a single neighboring block along a given block face. This restriction forces block boundaries to be aligned and also implicitly requires neighboring block faces to contain the same number of computational node points. Valid and invalid decomposition examples are given in Figure 9. The reader should note that the shaded

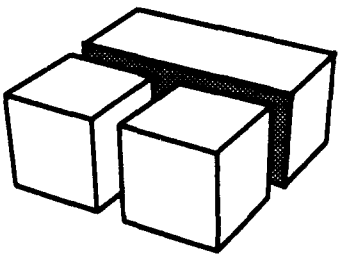
1. The author's definition of turnaround time is the physical time elapsed between submission of the computing job and retrieval of the computed solution.



Computational Domain



Valid Decomposition



Invalid Decomposition

Figure 9: Domain Decomposition Examples

face in the invalid decomposition sketch borders on two block faces which violates the stated restriction. This restriction is not unique to this project; Hauser also implements a similar restriction during domain decomposition [17].

3.1.2 Inter-block communication

A division of the computational domain into subdomains or blocks necessitates the communication of information across adjoining subdomain faces. This communication is part of the parallelization overhead and detracts from the performance of the parallel algorithm since processors must expend clock cycles either communicating or waiting to communicate instead of carrying out the desired calculations. For the TVD scheme used in this project, the two factors affecting the amount of information which must be communicated across the blocks are the logical size of the block face in terms of number of nodes and the size of the computational stencil. These two factors are illustrated in Figure 10. Since the computational stencil is five-points wide along any of the coordinate directions, points occupying the points either on a block boundary or immediately adjacent thereto must necessarily use information

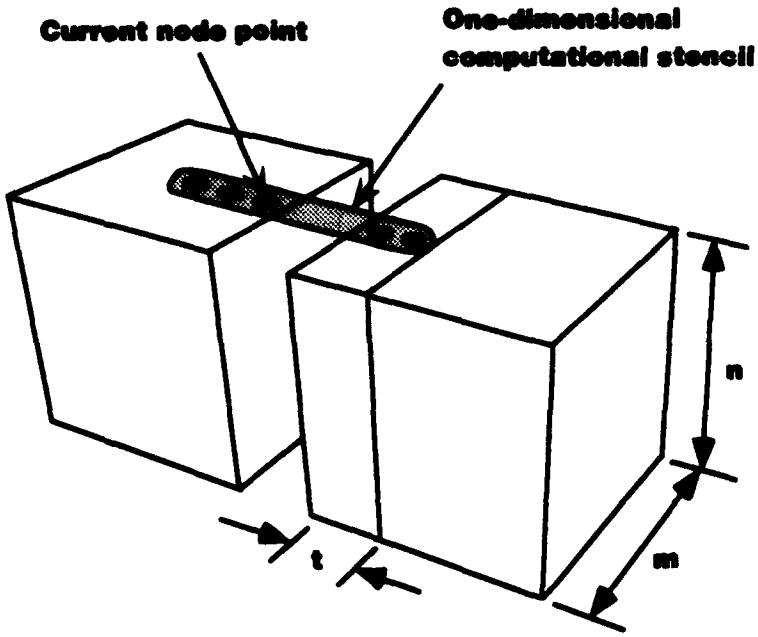


Figure 10: Inter-domain Communication Model

from nodes within an adjacent block. While the size of the block face depends on the logical partitioning of the computational domain, the size of the computational stencil is fixed due to the nature of the TVD scheme. From the figure, it is obvious that the number of node points along any face which require information from an adjacent block is $n \times m \times t$.

While relatively unimportant in a shared-memory implementation, the number of nodes occupying a face of a block becomes very important in a distributed-memory implementation. This is due to the fact that all across-block references must be handled through the explicit use of message passing. Reference [21] provides an excellent discussion on message passing for a distributed-memory architecture. As pointed out by Braaten [5], "the key factor for minimizing communication costs is to maximize the ratio of computation in each processor to the communication between processors." Braaten also notes that the computational work within a block is proportional to the number of nodes in a block while the communication work is proportional to the number of nodes on block boundaries. It is therefore desirable to maximize the internal node to boundary node ratio. Since no communication occurs on block

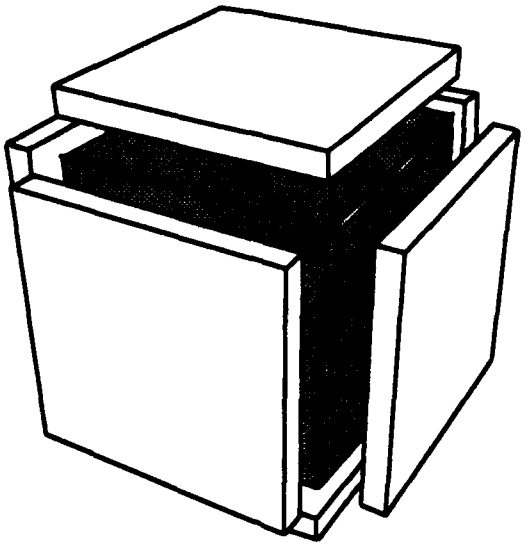


Figure 11: Computational Block and Buffer Arrays

boundaries which are part of the physical problem (i.e., those boundaries which are handled through the mathematical boundary conditions of the problem), these boundaries should not be considered in the ratio of internal to facial nodes within a block.

The key data structure normally used [5,17,46] when implementing the interblock communication model is a set of buffer arrays which are of dimension n x m x t where n and m are the dimensions of a block face and t is the number of points from which information must be communicated across block faces. The dimensions crucial to the interblock communication appear in Figure 10 while Figure 11 shows a general computational block surrounded by six buffer arrays.

While the buffer arrays are generally quite efficient in terms of computational speed, it is obvious that there is a storage penalty levied by their use. For example, a single buffer array for an 82-node-by-35-node block face and a five point computational stencil requires 5740 storage elements. Using buffer arrays in the domain decomposition chosen for this investigation would require approximately 22 megabytes of storage which translates to a 19% additional memory requirement above the serial version of the computer code. Because the serial version of the code requires approximately 115 megabytes of storage and since nearly

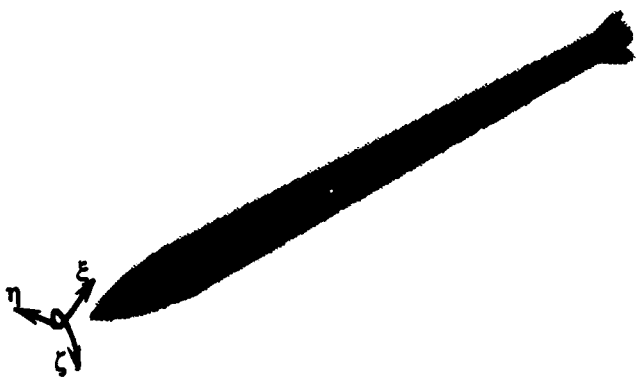


Figure 12: Missile Coordinate System

all locally-available computing platforms possess a maximum of 128 megabytes of storage capacity, the use of buffer arrays in the parallelization process was deemed unacceptable. Instead, a block table was constructed which managed the flow of information between the blocks with no storage overhead. This data structure is discussed more fully in Appendix B.

3.2 Selection of Computational Subdomains

PANS-3EM is designed to allow for an arbitrary number of computational blocks with a decomposition performed along any or all of the computational coordinate directions. However, as mentioned in Section 3.1.1, for this investigation, decomposition was performed along a single coordinate direction. The choice of the axis of decomposition was motivated by two primary factors: ease of incorporation into the existing code framework, and inter-block communication.

The computational domain used in this study consisted of 62 x 82 x 35 nodes in the ξ, η, and ζ coordinate directions, respectively. The coordinate system is depicted in Figure 12. In this case, computational blocks with a minimum number of facial nodes would be generated by a decomposition along the η coordinate axis. This decomposition would generate computa-

tional blocks with 62 x 35 node block faces across which information would be exchanged with neighboring blocks. However, it was anticipated that the addition of a base flow region would extend the number of points in the ξ direction and greatly complicate such a decomposition. Consequently, a decomposition along the ξ coordinate direction was chosen since it provided for the optimal combination of block facial nodes and integration into the existing computer code. Decomposition in this manner resulted in 2,870 elements along a face for which information must be exchanged. Had the decomposition occurred along the ζ coordinate direction, then 5,084 elements would have appeared along a face thus resulting in a 77% greater communication requirement. Again, this communication penalty does not occur on a shared-memory implementation. Decomposition along two or three axes would increase communication requirements even further since additional nonphysical boundaries would be created.

Four computational domains were used throughout the course of this investigation. This number was deemed an acceptable match for the computing machines locally available which typically consist of between two and four processing units. This decomposition resulted in three of the four domains containing 45,920 node points while the fourth domain contained 40,180 node points. While load balancing considerations would dictate an identical number of nodes in each domain, the disparity was necessitated by the restrictions stated previously. No disparity in number of node points occurs only for the condition

$$n \bmod m = 0 \tag{38}$$

where n is the number of node points along the axis of decomposition and m is the number of blocks along that axis.

3.3 Implementation of Domain Decomposition

With the structure of the computational blocks determined, generation of the blocks proceeded by first blocking a single two-dimensional grid plane, and then rotating the blocked grid plane around the missile's axis of symmetry. Figure 13 shows a blocked two-dimensional

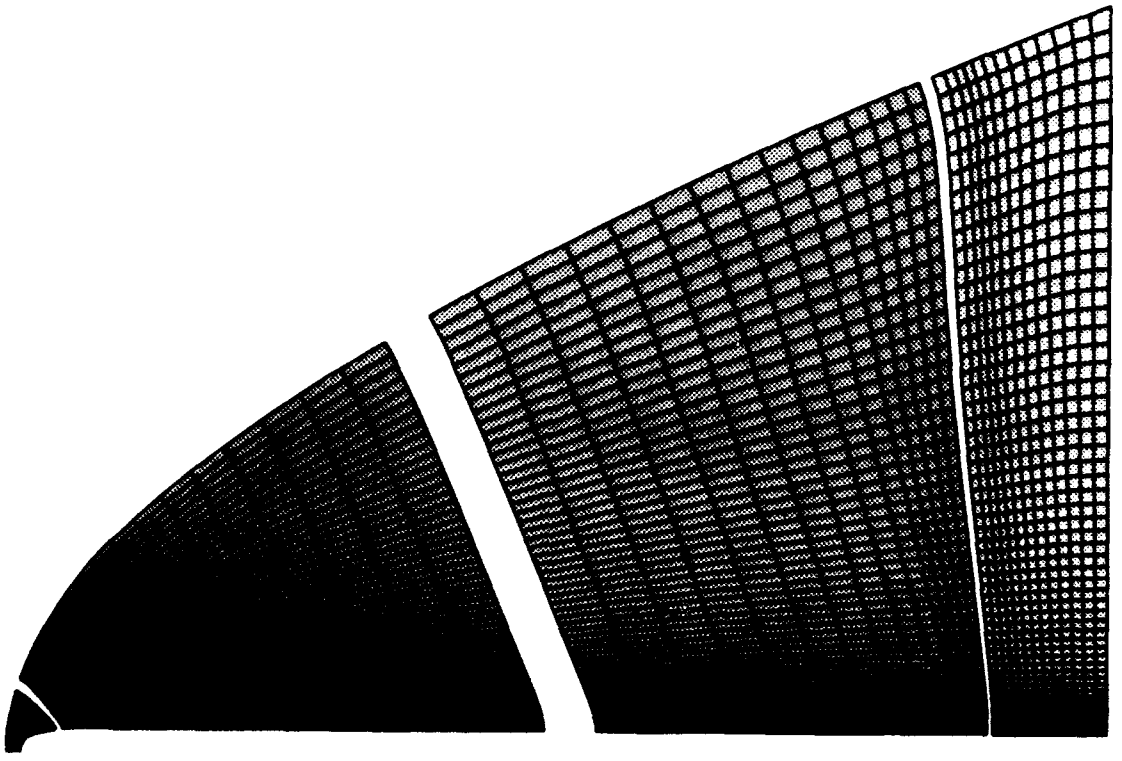


Figure 13: Two-Dimensional, Blocked Grid

grid plane while Figure 14 depicts the inner and outer hull of a three dimensional grid. The gaps in the grids represent the dividing lines between the computational blocks. Although the domains are of very different size in the physical space, they contain nearly the same number of computational nodes. The grids pictured in Figures 13 and 14 are for illustration of the domain-decomposition process and do not represent the actual grids used to study the sharp-nosed missile.

Implementation of the domain decomposition was performed by utilizing a four-dimensional array construct. With this implementation, a node in computational coordinate space, located with the ordered triple (i,j,k), has a one-to-one mapping to a node in the block coordinate space, located with the order quadruple (i',j',k',L). Since it is very impractical for a programmer or user to specify coordinates in block space, the program takes as input compu-

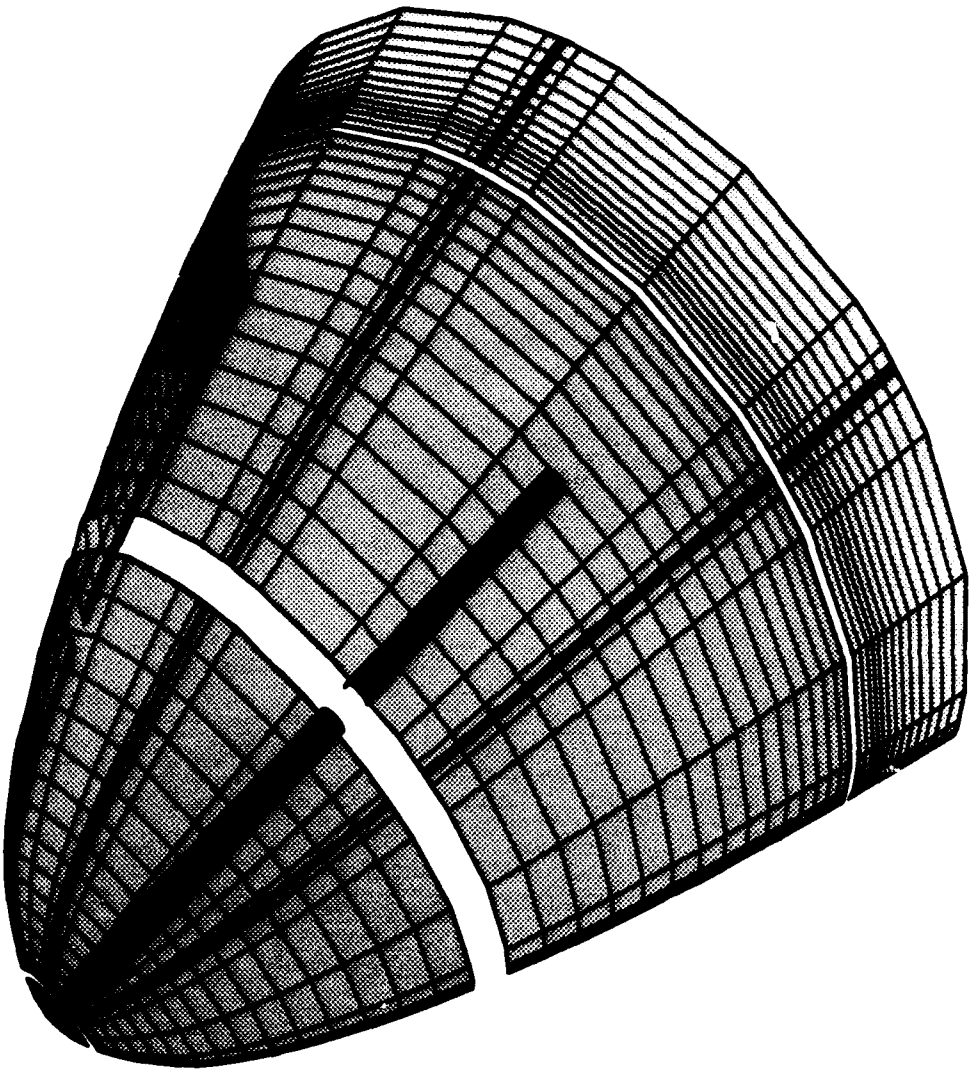


Figure 14: Three-Dimensional, Blocked Grid

tational coordinates and converts them to block coordinates. For the four domain decomposition used in this project, the mapping from computational to block coordinates takes the form

$$L = \left\lfloor \frac{i + ighost}{idim} \right\rfloor + 1 \tag{39}$$

$$i' = i + ighost - (L-1)\, idim \tag{40}$$

$$j' = j \tag{41}$$

$$k' = k \tag{42}$$

where *ighost* is the number of ghost cells in the i-coordinate direction and *idim* is the number of points in each block along the i-coordinate direction. The inverse transformation is given simply by solving equation 40 for the computational coordinate, i.

The proper handling of computational and block coordinates is fundamental to achieving a working computer code. The issues concerning the coordinate transformations are discussed fully in Appendix B.

IV. Results

Results for this investigation's two major areas of focus–algorithm development and aerodynamic analysis–are presented. Validation and performance results for the PANS-3EM code are provided. Additional results which include lessons learned from the algorithm development process and a road-map for future parallelization efforts appear in Appendices B, C, and D. In the aerodynamic arena, computed results for drag and pitching moment derivatives are compared to other numerical as well as experimental results. Underlying study principles such as computational grid resolution and fin modeling issues are discussed. Also included are the results of the turbulence model on the solution behavior.

4.1 Parallel Program Validation

Validation of PANS-3EM was accomplished by comparing results to the serial version of the code. With both codes run in serial mode, no differences were expected in any of the computations. A total of four separate validation runs were performed, each designed to test various aspects of the code. A summary of the runs and the test objectives appears in Table 1.

Table 1: PANS-3EM Validation Run Summary

Run Number	Description	Objective
1	Euler boundary conditions, no angle of attack, no turbulence, no fins	Test basic code including metric computations, Euler boundary condition computation routines, and sweep routines
2	Navier-Stokes boundary conditions, no angle of attack, no turbulence, fins	Test Navier-Stokes boundary condition routines, velocity and temperature gradient computation routines, and fin-handling loops
3	Navier-Stokes boundary conditions, angle-of-attack, no turbulence, fins	Test angle-of-attack-related data structures
4	Navier-Stokes boundary conditions, angle-of-attack, turbulence, fins	Test turbulence routine

The validation runs were performed using a 62 x 82 x 35 computational grid. With the exception of run number 1 in which the flow-field was initialized to free stream conditions, all runs were started from a partially-developed solution. This facilitated a more thorough test of the code since errors in gradient calculations were difficult to detect for a relatively uniform flowfield. Validation runs were performed using 100 iterations. Norm histories for validation runs 1 and 4 are depicted in Figures 15 and 16. Data for runs 2 and 3 are not presented since their behavior is similar to run 4. Tabular data for selected iterations also appears in each figure in order to verify that the computed norms are in fact identical.

In addition to the norm results, comparisons were performed on the conserved variables output from each code. The FORTRAN output format specification utilized was e18.12. With a 62 x 82 x 35 computational grid, each output file contained approximately 18 megabytes of numerical data. Files were then compared using the UNIX *diff* command. In all cases, all computed values were identical.

4.2 PANS-3EM Performance Testing

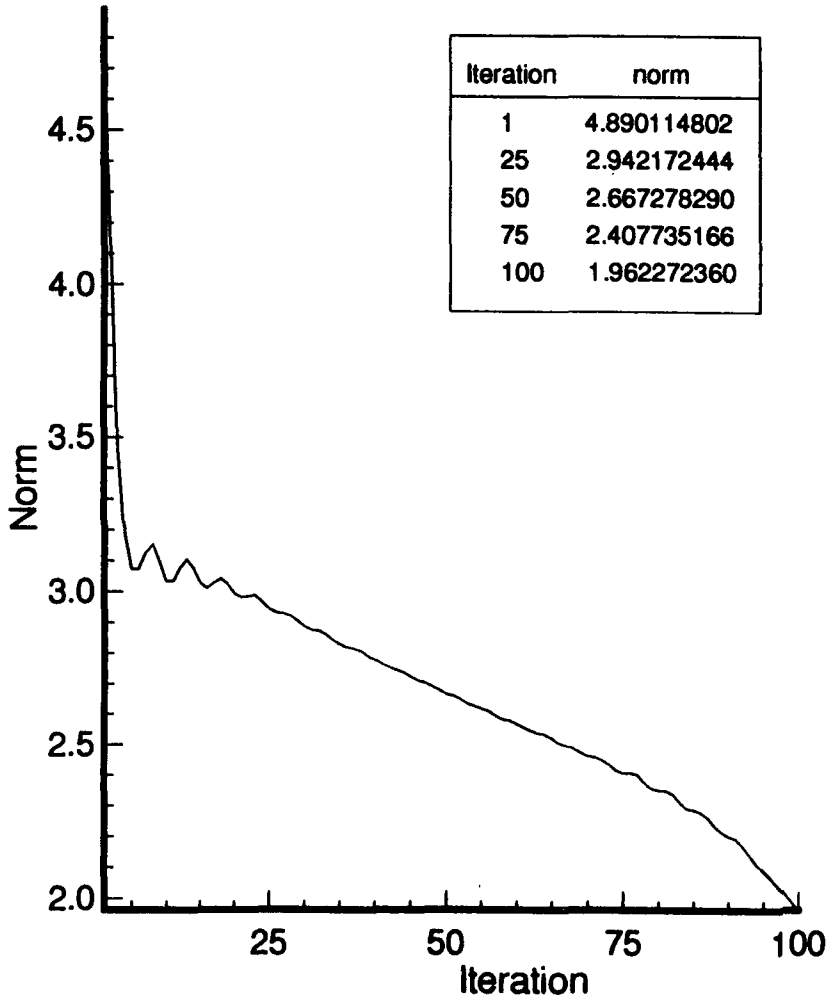
The performance of PANS-3EM was assessed through comparative timing runs on several shared-memory platforms. Table 2 contains a summary of the platforms used along with brief descriptions of the relevant hardware features of each platform.

Table 2: Code Test Platforms

Platform	Main Memory (Mbytes)	Number of Processors	Processor Type	Parallel Implementation
Sun 4/690	600	4	Scalar	none
Digital Equipment DEC 4000	360	2	Scalar	none
Silicon Graphics Power Series 4D/480	128	8	Scalar	Power FORTRAN Accelerator
Convex C220	128	2	Vector	Compiler Optimization

Serial Code

Iteration	norm
1	4.890114802
25	2.942172444
50	2.667278290
75	2.407735166
100	1.962272360



PANS-3EM Code

Iteration	norm
1	4.890114802
25	2.942172444
50	2.667278290
75	2.407735166
100	1.962272360

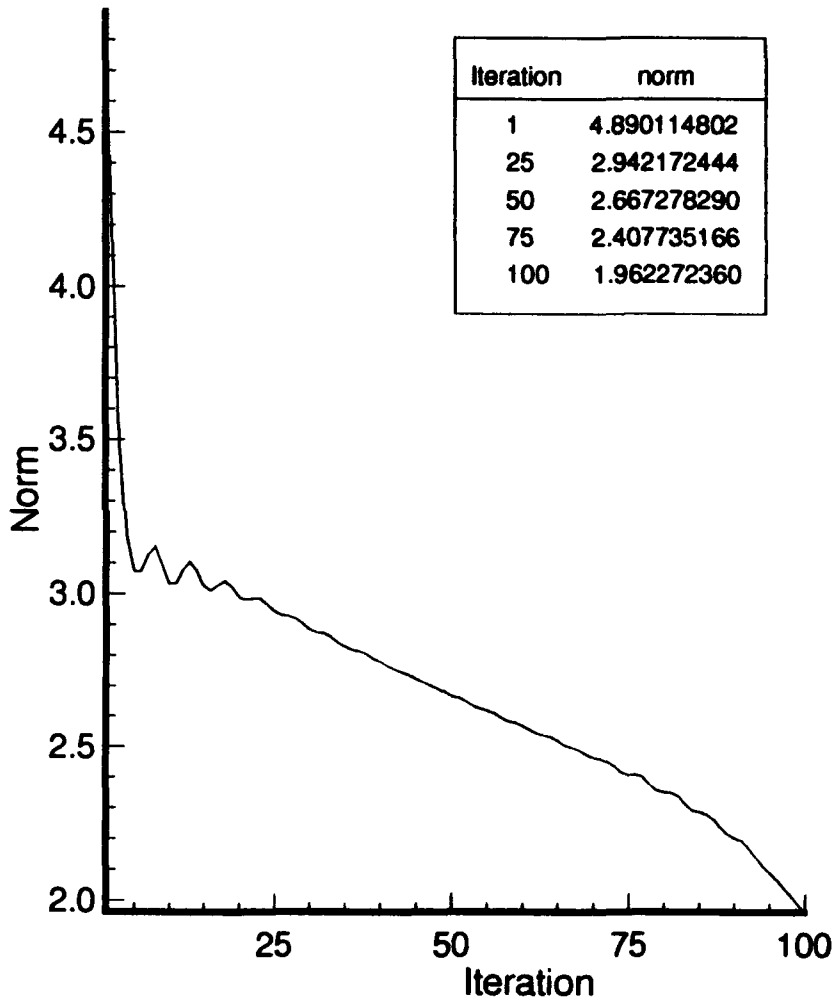


Figure 15: Code Validation for Test Case 1

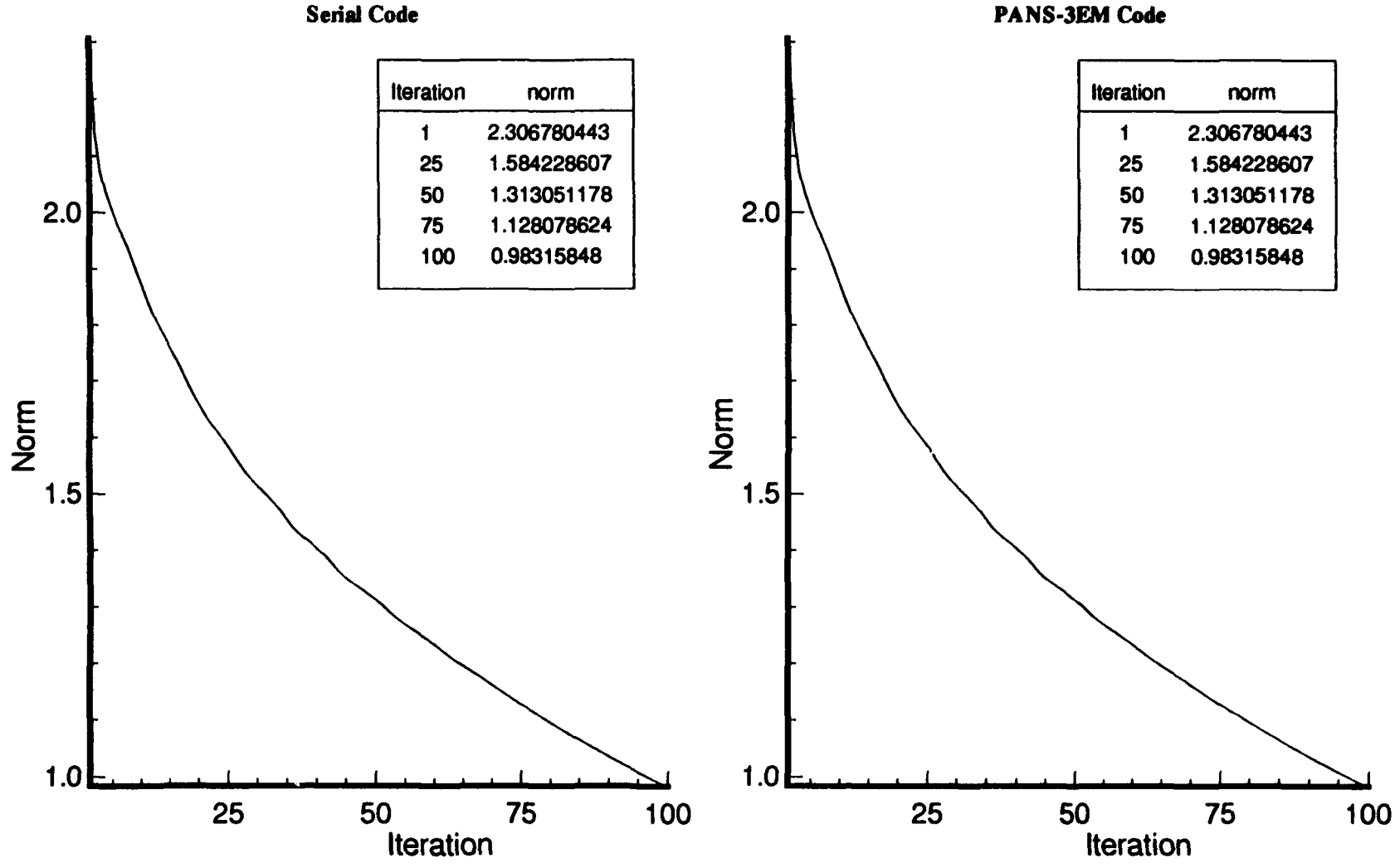


Figure 16: Code Validation for Test Case 4

4.2.1 Sun 4/690

Although this machine has four processors, no parallelization tools or parallelizing compiler directives were available. Timing runs were conducted on this machine due to the large amount of main memory which eliminated any memory paging effects that could bias the timing results.

4.2.2 Digital Equipment DEC 4000

Version 6.0 of the Digital Equipment (DEC) FORTRAN compiler was incapable of generating object code for a parallel implementation. This situation is currently being addressed by DEC, and when the problem is resolved, PANS-3EM will be implemented in parallel on this machine. Timing results for serial execution are presented for future comparative purposes.

4.2.3 Silicon Graphics 4D/480

Of the platforms tested, the Silicon Graphics FORTRAN compiler provides the greatest flexibility in the parallelization process. Power FORTRAN Accelerator (PFA), a Silicon Graphics code optimizing preprocessor, performs loop-level optimizations on source code files. More explicit control is available through the use of operating-system-level function calls and compiler directives. These tools provide the option of a more complete domain decoupling in the PANS-3EM code at the expense of a much greater degree of programming effort. Timing runs presented here were obtained with the use of PFA.

4.2.4 Convex C220

This machine was the only vector machine tested. The performance penalty associated with the PANS-3EM code was expected to be especially harsh since the modifications required in order to implement the domain decomposition severely reduced the code vectorizability. No explicit parallelizing tools were available for this machine and consequently a compiler directive was used to accomplish the parallelism.

4.3 Performance Data

In all performance tests, both versions of the code were compiled with identical compiler flags. For the Silicon Graphics and Convex runs, the performance times are thus indicative of the degree of efficiency with which the respective compilers are able to parallelize each version of the code.

Table 3 contains the results of the comparative performance testing of PANS-3EM and the serial code. As expected, in all cases the PANS-3EM code is slower than the original code. Since no parallelism or optimization was employed on the Sun or Digital Equipment machines, the performance difference is due exclusively to the additional code required to implement the domain decomposition. The DEC machine relies heavily on data and instruction caching to achieve maximum performance; the greater reduction in the performance numbers on this machine indicates that the new code does not lend itself to the employed caching algorithms as well as does the serial code. An examination of the coding methodology as described in Appendices B and D reveals that the domain-decomposition algorithm does not affect the order of complexity of the serial code. However, there is an increase in operation count that is reflected in the performance numbers.

Table 3: Performance Metrics

Platform	Processors Used/ Available	Serial Code CPU Time (sec)	PANS-3EM CPU Time (sec)	Percent Difference
Sun	1/4	2293.9	2347.5	-2.3
Digital Equipment	1/2	482.0	533.0	-10.6
Silicon Graphics	8/8	2926.8	3372.7	-15.2
Convex	2/2	320.6	420.6	-31.2

The 15 percent performance reduction of PANS-3EM over the serial version of the code for the Silicon Graphics implementation is composed of two parts: one part corresponding to the extra operation count and another part corresponding to the less efficient loop-level

parallelism implemented by the compiler. The four-dimensional array constructs used in the domain-decomposition approach require an additional looping structure in the computer code. The presence of this additional nested loop coupled with the necessary across-block-reference checks results in an added level of complexity which is not easily handled by the parallelizing compiler.

The performance reduction trend is carried even further in the case of the Convex. This is directly attributable to the reduced degree of vectorizability of the looping structures due to the domain-decomposition process. The issue of vectorizability is fully discussed in Appendix B, Section B.4. The Convex, a shared-memory vector architecture, was not a target machine of implementation, and consequently the domain-decomposition approach was not optimized for this machine. The reader should note that vastly different performance numbers are expected for a distributed memory vector processing computer such as the Intel Paragon or a distributed memory scalar environment such as a workstation farm since these implementations would require a complete decoupling of the computational domains which would reduce the degree of loop nesting and eliminate the reduction in code vectorizability inherent with the current version of PANS-3EM.

As mentioned previously, the reduction in performance numbers for PANS-3EM were expected due to the additional code required to implement the computational domain decomposition. This is a necessary precursor to a distributed-memory implementation. The slight reduction in performance is more than offset by the increased flexibility the PANS-3EM implementation provides. Furthermore, given sufficient computational tools, a complete decoupling of the computational domains can be conducted which has the potential of greatly improving performance. The issues associated with the domain decoupling are discussed in Appendix B.

4.4 Aerodynamic Results

Aerodynamic results presented here focus on the computed drag and pitching moment derivative coefficients for the sharp-nosed missile and the aerodynamic phenomena believed to have the greatest impact on the observed results. These factors include the stagnation

region pressure which affects the computed pressure drag, boundary layer resolution which affects the skin-friction drag, and flow-field resolution in the fin area which affects the pressure distribution over the fins and consequently greatly impacts the pitching moment derivative results. A discussion of each of these issues as well as a mention of the key computational issues and study methodology appears below.

4.4.1 Computational Issues

The computational grid used in this investigation consisted of 62 x 82 x 35 node points along the ξ, η, and ζ coordinate directions, respectively. Minimum wall spacing was fixed at 0.0001. The grid was designed to closely approximate the grid used in the work of Moran [27], and contains nearly the maximum number of node points that can be used on a 128 megabyte computing platform. However, certain aspects of the results indicate strongly that the grid was not sufficiently refined in order to resolve all relevant flow structures. Since a machine with larger memory capacity and sufficient computing power was not locally available, no grid refinement study was conducted. However, when the problems associated with parallel implementation on the DEC 4000 are fully resolved, this issue should be explored in depth.[1]

Generation of the grid was accomplished using GRIDGEN [38], an elliptical grid generator developed for Wright Laboratories. Bisymmetry of the flow-field was assumed (no missile yaw), and hence the computational domain extended over only half of the missile. Points were clustered in the stagnation and fin regions as well as in the boundary layer. Figure 17 depicts a two-dimensional macroscopic view of the grid as well as a close view of the grid in the stagnation region. Instead of a perfectly sharp nose, a slight rounding was incorporated in order to more closely match the missile sketch and shadowgram of Reference [13] and to improve the solution performance in the nose region. Examination of the axis scales on the inset graph reveals that the nose radius is extremely small–on the order of 2% of the body diameter. The three-dimensional grid was formed by rotating and duplicating the two-dimen-

1. See Section 4.2.2.

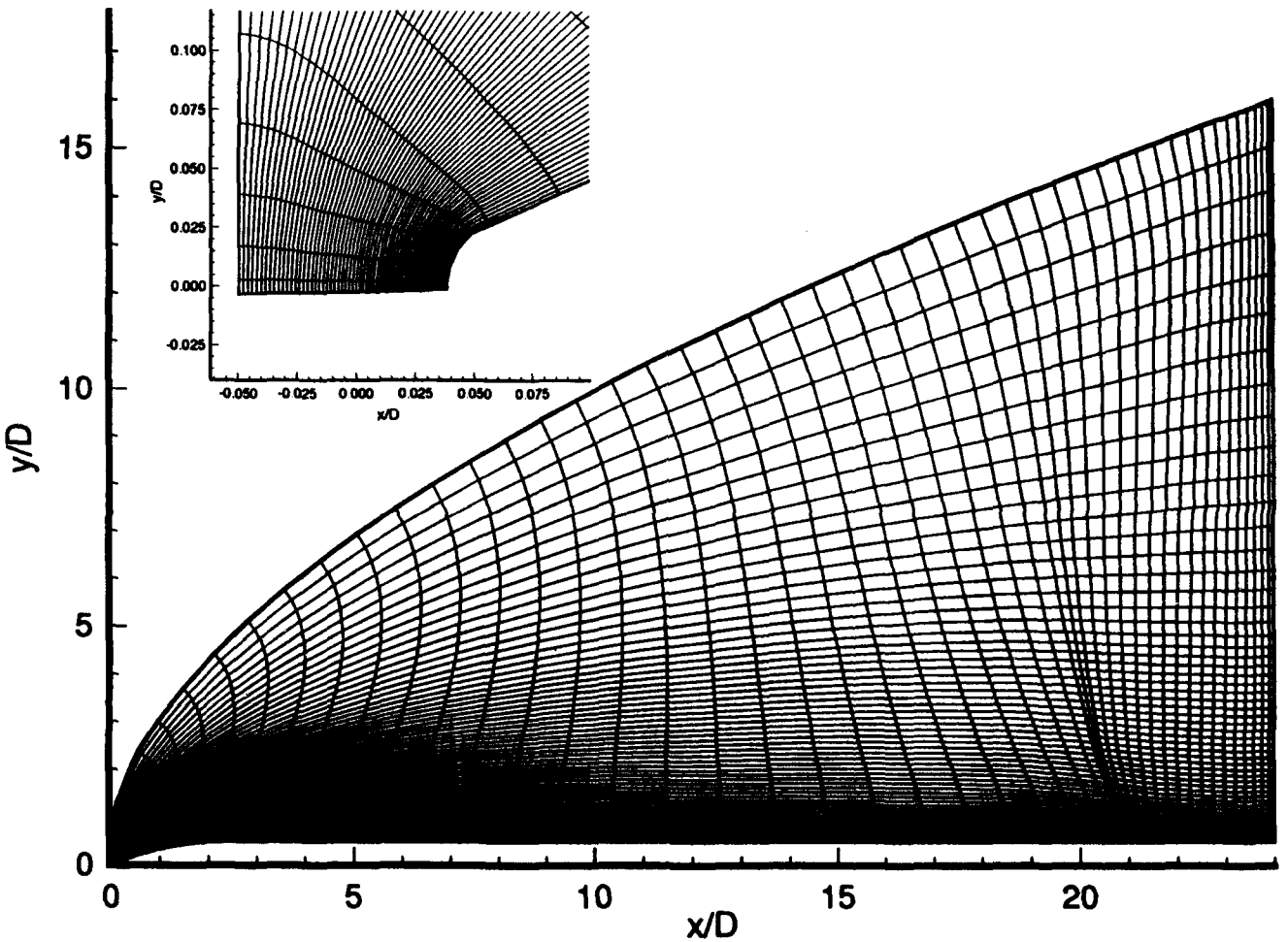


Figure 17: Detail of Two-Dimensional Grid Plane

sional grid about the missile's axis of symmetry.

Convergence criteria for the computer runs was established by Moran [28] based on previous two- and three-dimensional work performed using the Harten-Yee TVD algorithm. The norm for convergence calculations was defined using the equation

$$\|L\|_2 = \left\| \sum_n \Delta U^2 \right\|^{\frac{1}{2}} \qquad (43)$$

where n is summed over all interior nodes in the computational domain and ΔU represents a summation over the changes in values of the components of the conserved variable vector, U between successive time steps. The solution was deemed converged when the norm value given by equation 43 reached 1×10^{-4}. Approximately 36 CPU seconds were required per iteration on the Convex C220 with a typical case achieving convergence after 8000-10000 iterations.

4.4.2 Methodology

A total of 20 computer runs were performed. Freestream Mach number was varied between two and six in order to establish trends over the entire range of available experimental data. Drag coefficients were calculated for zero angle-of-attack missile and include computed contributions due to skin friction and pressure as well as an empirically-obtained contribution [12] due to the missile base region. Pitching moment derivative coefficients were obtained by assuming a linear variation in pitching moment over the small (five degree) variations in angle of attack.

The nature of the numerical algorithm requires an initial solution with a partially developed shock and boundary layer structure [27]. Accordingly, the flow field was initialized to freestream conditions, and 100 iterations were performed with the Reynolds number set artificially high in order to effectively negate any viscous effects. The viscous terms were then "turned on" and the associated boundary layer allowed to partially develop over several hundred more iterations. Finally, the turbulence model was activated and the solution allowed to

iterate until convergence. A summary of the parameters for each computer run appears in Table 4.

Table 4: Sharp-Nosed Missile Test Matrix

Run	Mach Number	Angle of Attack (deg)	Turbulence Model Active
1	2.0	0.0	
2	2.0	0.0	✔
3	2.0	5.0	
4	2.0	5.0	✔
5	3.0	0.0	
6	3.0	0.0	✔
7	3.0	5.0	
8	3.0	5.0	✔
9	3.5	0.0	
10	3.5	0.0	✔
11	3.5	5.0	
12	3.5	5.0	✔
13	4.5	0.0	
14	4.5	0.0	✔
15	4.5	5.0	
16	4.5	5.0	✔
17	6.0	0.0	
18	6.0	0.0	✔
19	6.0	5.0	
20	6.0	5.0	✔

4.4.3 Drag and Pitching Moment Derivative Coefficient Results

Computed pitching moment derivative results appear in Figure 18. The negative values of pitching moment coefficient indicate that a restoring moment is being generated by the fins. As freestream Mach number increases through Mach 3.5, the fins lose effectiveness as is

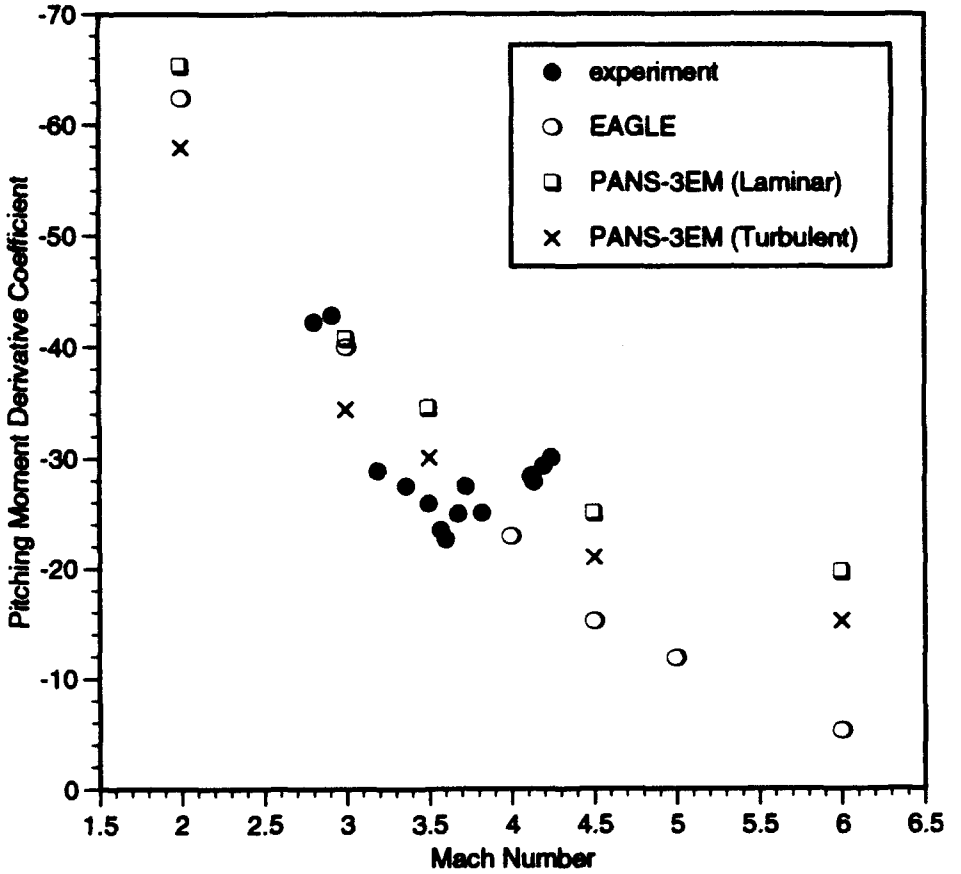


Figure 18: Sharp-Nosed Missile $C_{m\alpha}$ Results

evident by the decreasingly negative values of the pitching moment derivative coefficient. At approximately Mach 3.75, the trends in the experimental and computed results diverge, with the computed results continuing in a decreasing fashion while the experimental results reverse their trend. The reason for the reversal is still not understood and is the subject of continuing study [28,44]. All computed results predict approximately the same trend, with the EAGLE code predicting the most marked drop off in $C_{m\alpha}$. Since EAGLE is an inviscid code, its failure to predict the reversal of the stability characteristics indicates that the reversal is most likely driven by viscous-related phenomena.

Computed drag results appear in Figure 19. As with the pitching moment coefficient calculations, the trends agree with the experimental results. However, in all cases, the code

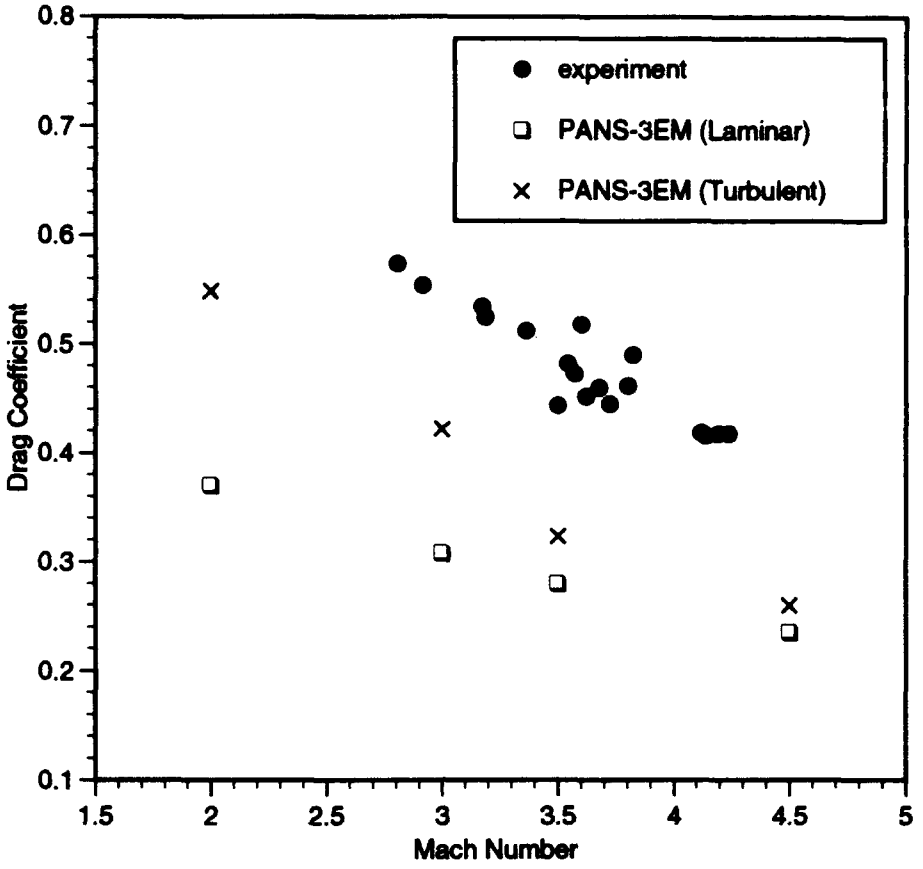


Figure 19: Sharp-Nosed Missile C_d Results

significantly under predicts the measured drag. As expected, turbulent drag is higher than laminar drag due to the much higher skin friction associated with the turbulent boundary layer.

Examination of Figures 18 and 19 reveals a slight change in the computed trends in the Mach 3.0-3.5 region. Although no unique flow structures were isolated to this Mach regime, it is possible that a more sophisticated fin model and a more refined grid may reveal such structures to which the stability reversal may be attributed.

The results presented in Figures 18 and 19 depend heavily on three primary regions of computation: the stagnation region, the boundary layer region, and the fin region. A discussion of the observed results for each of these areas follows.

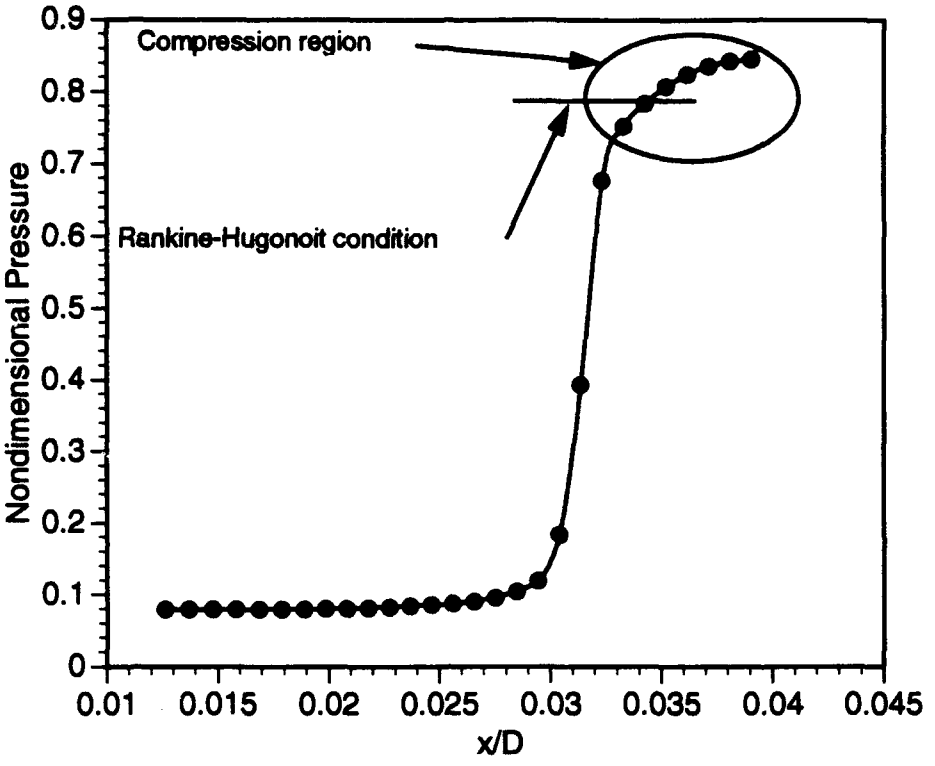


Figure 20: Stagnation Region Pressure Results for $M_\infty = 3.0$

4.4.4 Stagnation Region Calculations

Typical results for stagnation region pressure and density appear in Figures 20 and 21, respectively. Both figures show the detached nature of the shock as well as the compression region that occurs between the shock and the missile nose. A detached shock is expected since the missile nose is slightly rounded. The slight smearing of the shock evident in the figures and is a result of the coarseness of the grid as well as the artificial dissipation required to prevent the formation of nonphysical solutions in the region. Both of these subjects are discussed in Sections 4.5. Since the nose is slightly rounded, a normal shock develops along the stagnation line, and the Rankine-Hugonoit relations can be examined to ascertain the accuracy of the computed solution. Despite the slight smearing of the shock, computed values of both pressure and density satisfy the jump conditions to within 5%. A more accurate solution requires a higher level of grid refinement.

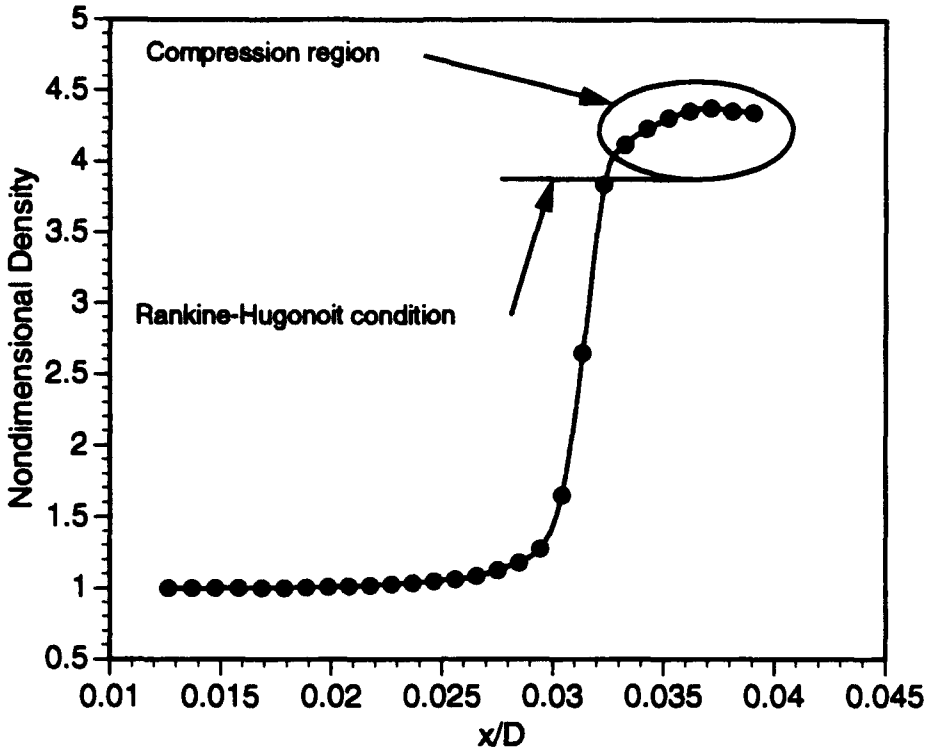


Figure 21: Stagnation Region Density Results for $M_\infty = 3.0$

Accurate computation of the flow in the stagnation region is important, since this region experiences relatively high pressures with a large component of the force acting in an axial direction. This subsequently enters into the drag calculations. Hence, small errors in this region can translate to measurable differences in drag results.

4.4.5 Boundary Layer Computations

Accurate determination of the skin friction requires that the boundary layer be adequately resolved. Figure 22 contains velocity vector plots of a small portion of the boundary layer in the nose region of the missile. Both laminar and turbulent boundary layer results are presented. The length of each vector corresponds to the magnitude of the velocity at a given point. The differences in the laminar and turbulent profiles are quite evident from the figure. Of special importance is the much higher wall velocity gradients in the turbulent case versus

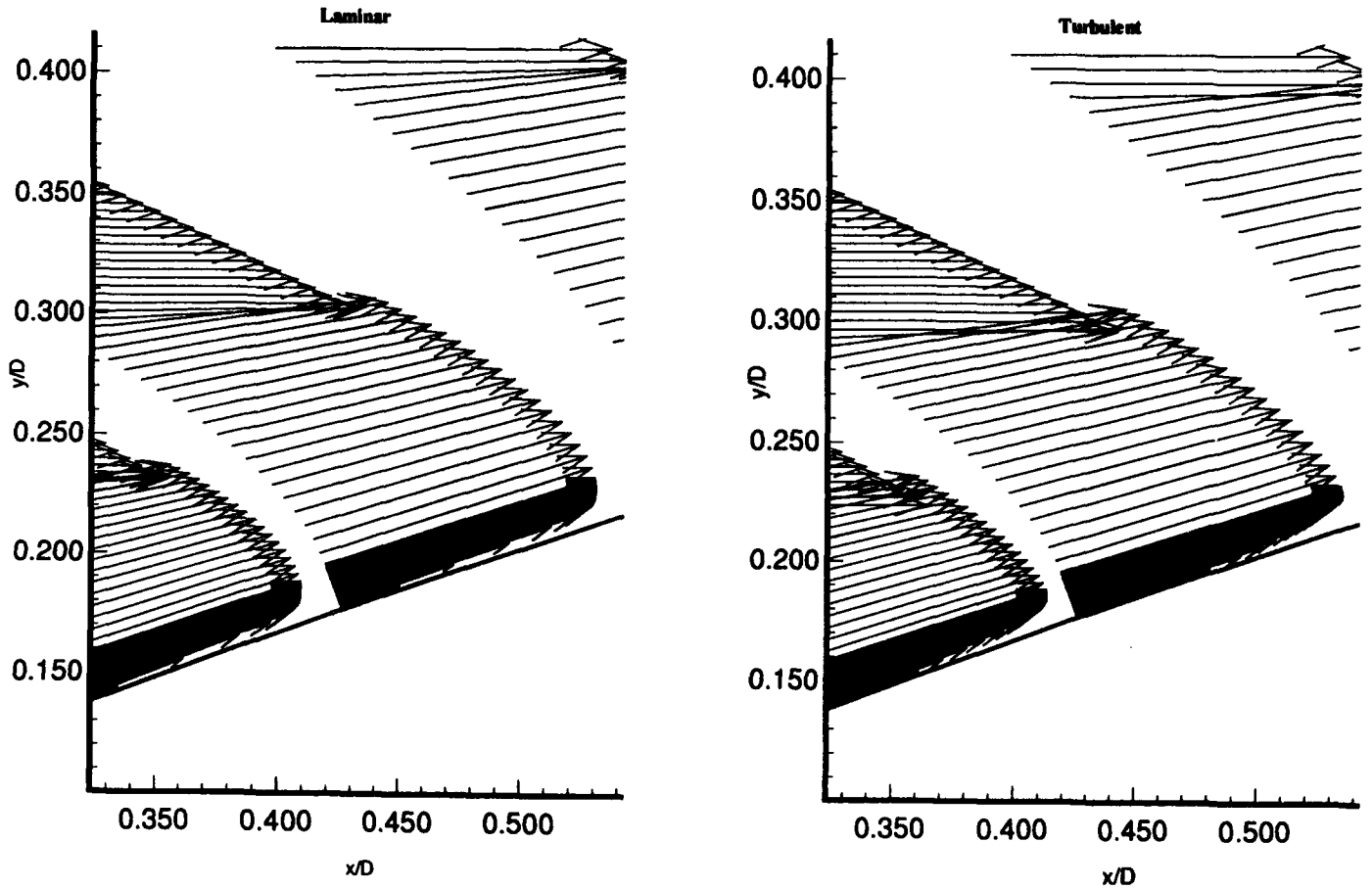


Figure 22: Comparison of Laminar and Turbulent Boundary Layer Velocity Profiles

that of the laminar case. This results in a higher skin friction for the turbulent case, and is precisely the reason for the higher drag values observed for turbulent versus laminar flow.

Although not especially important in this context, the shock location is clearly evident in the figure as the flow turns into itself. The TVD scheme is able to capture the shock in this location in a single grid point.

While Figure 22 depicts the wall shear stress, the relative thicknesses of the laminar and turbulent boundary layers are shown in Figure 23. The figure elucidates the Mach contours of the flow in the fin leading edge region. The fin is depicted as the darkest portion of the figure; the jagged leading edge of the fin is a result of the technique used to model the fin. The turbulent boundary layer is approximately three times thicker than the laminar boundary layer at this location. These trends are in agreement with theoretical results.

4.4.6 Fin-Region Calculations

Since the fins provide the majority of the pitching moment to the missile, it is extremely important to the accuracy of the solution that the flow be computed correctly in that region. The presence of the fins results in interesting and complicated flow structures as are depicted in Figure 24. The figure presents the velocity profiles at a station eight grid points downstream from the leading edge of the fins and are obtained for flight at 5° angle of attack. The differences in the flow qualities between the laminar and turbulent boundary layer computations are quite obvious. In particular, the laminar case is characterized by a region of flow between the fins which possesses a relatively high downward velocity component. This region is restricted to an area close to the missile body. In contrast, the turbulent case has a relatively small region of downward velocity which is localized in the proximity of the upper fin. It is quite possible that the thicker boundary layer associated with the turbulent flow allows for the bulk of the flow to be more easily entrained by the freestream. This is further evident from an examination of Figure 25 which shows the laminar case at a higher Mach number. The increased energy of the flow reduces significantly the downward velocity

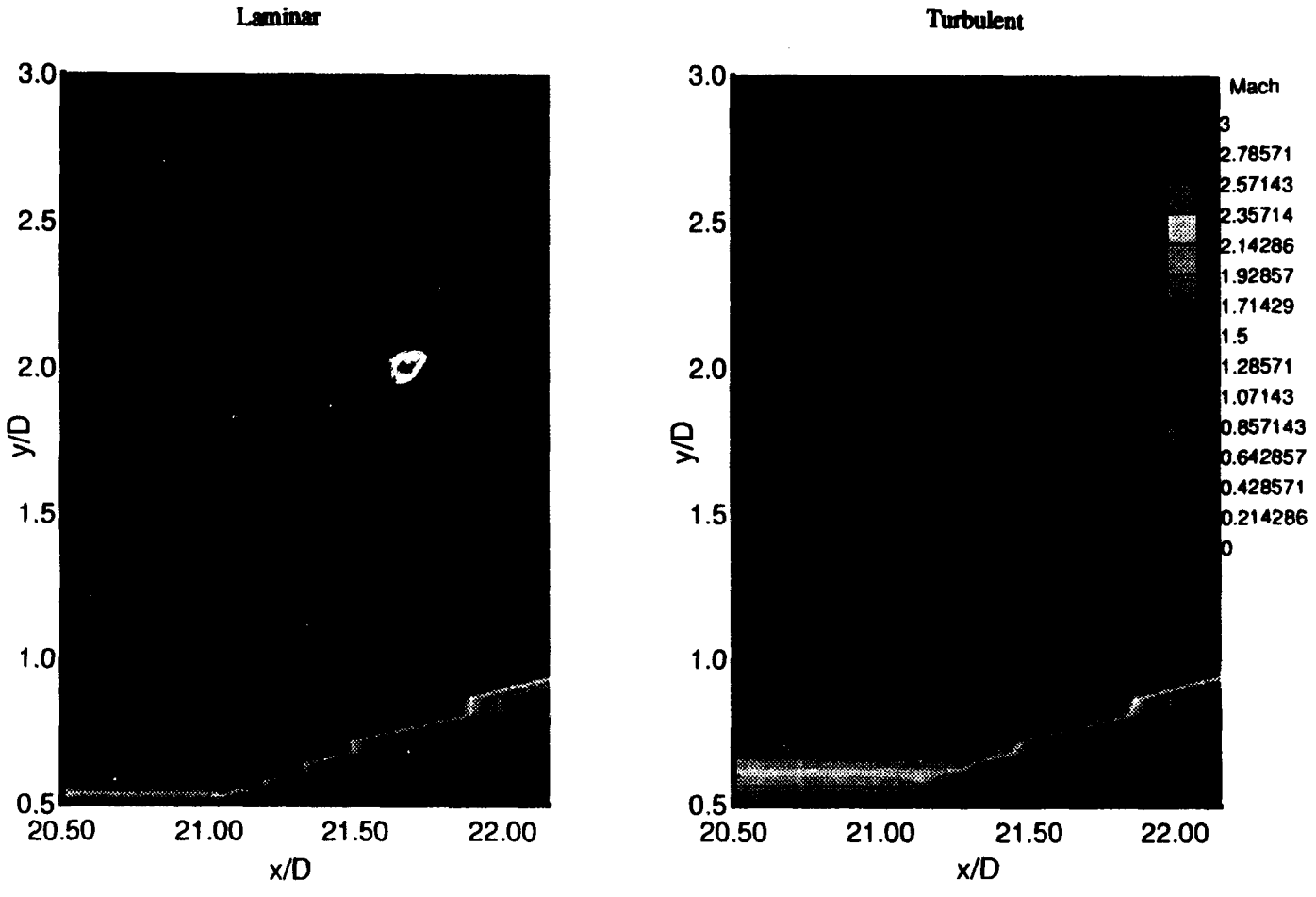


Figure 23: Mach Contours for Laminar and Turbulent Boundary Layers in Fin Region

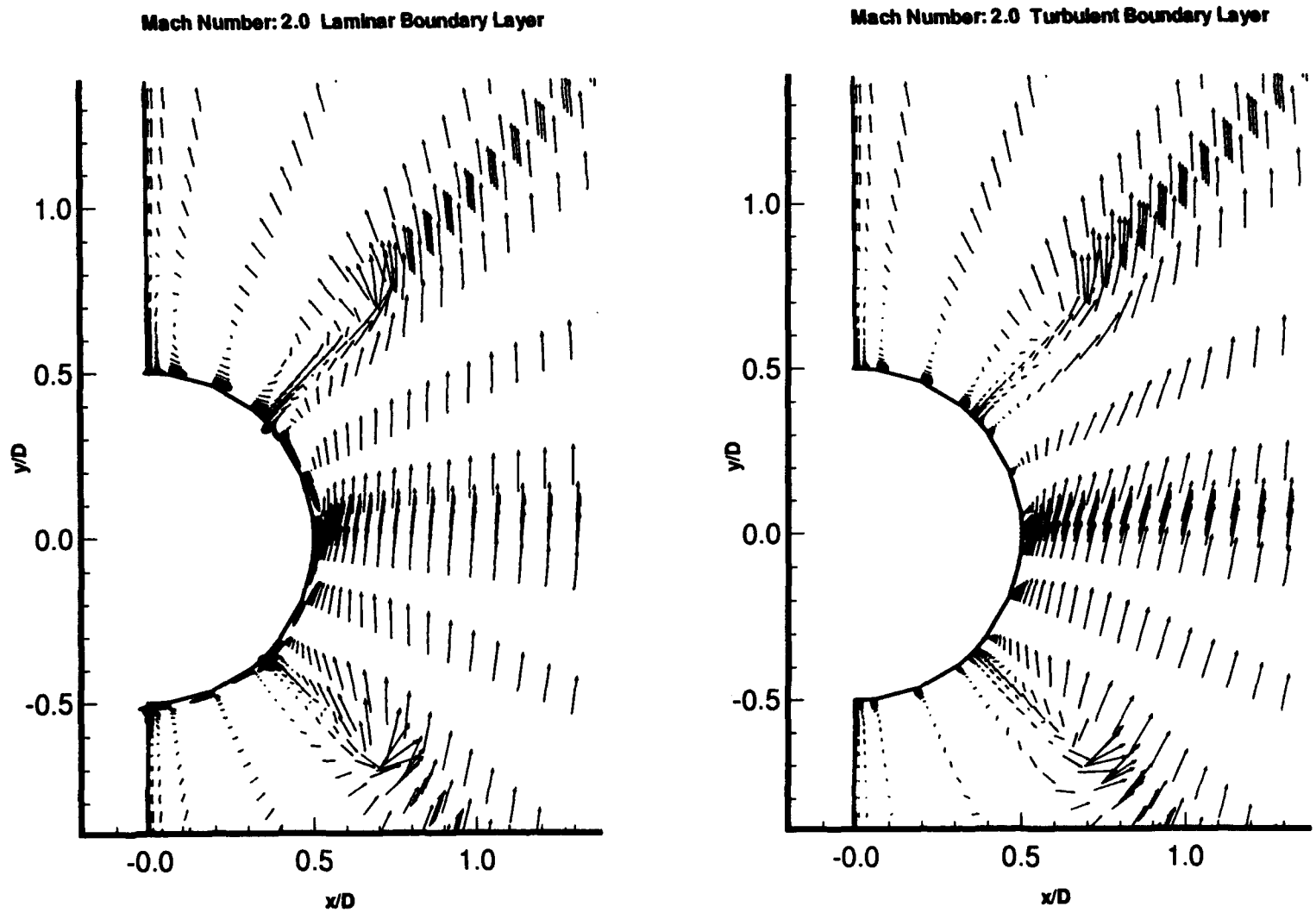


Figure 24: Fin Region Cross Plane Velocities for Laminar and Turbulent Boundary Layers

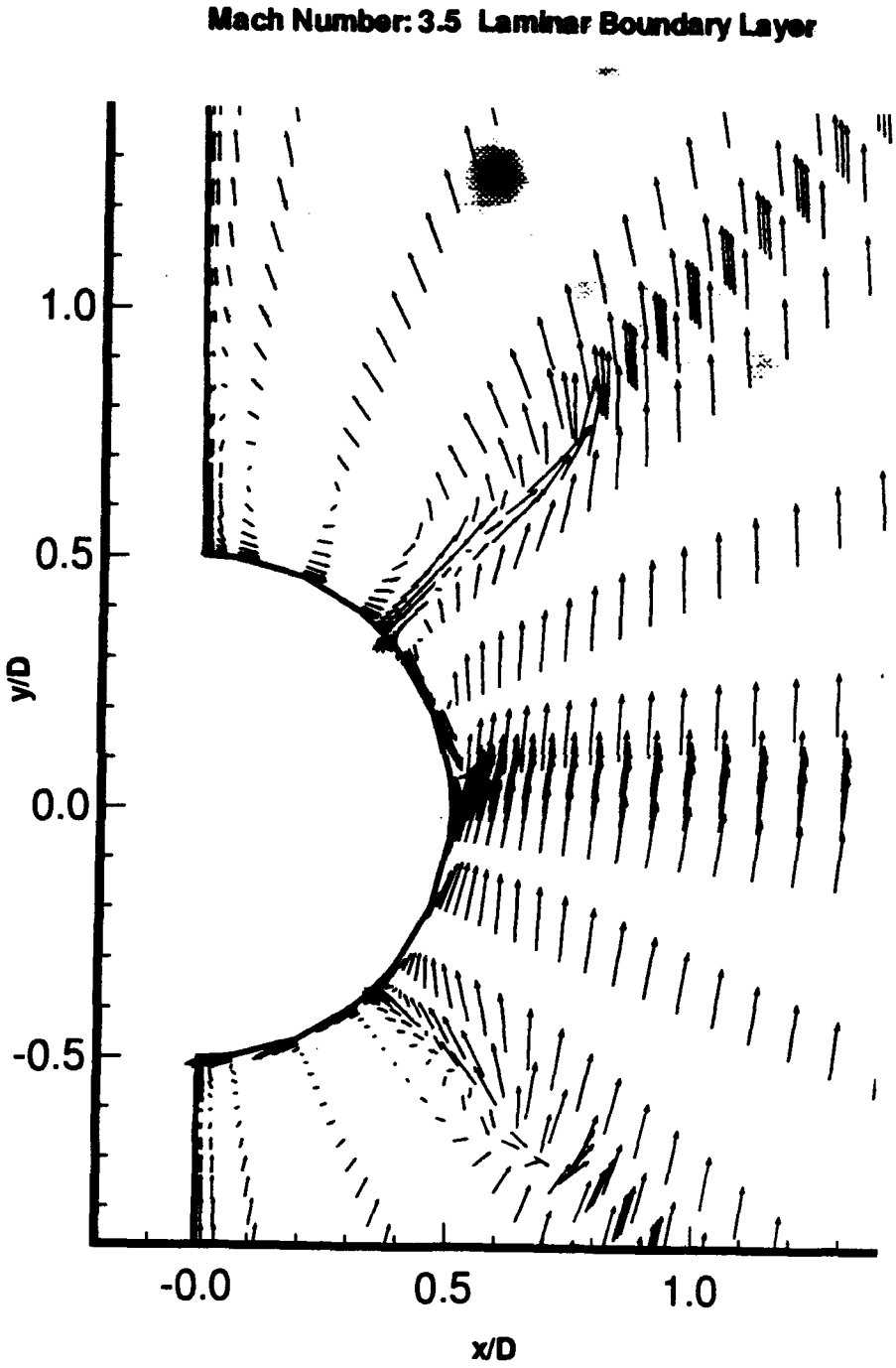


Figure 25: Cross Plane Velocity at Fins for M_∞ = 3.5 and Laminar Boundary Layer

component.

A momentum analysis using a control volume approach would indicate that a larger downward velocity component in the flow exiting the volume must translate to an upward force generated on the missile body. It is difficult, however, to make such a generalization since the overall momentum of the flow cannot be ascertained from a single cut through the computational domain.

Figures 26 examines the pressure distribution over the two fins and depicts the same view shown in Figure 24. The mixing effect of the turbulence results in "spillage" of the pressure from the lower to the upper surfaces of the fins. A more equal pressure distribution over the fin surfaces corresponds to a reduced net lift and subsequently to a reduced pitching moment.

4.5 Model Weaknesses

In reference to the experimentally-observed missile stability characteristics, Vitale [44] states, "A significant number of flights were conducted in this Mach range (3.75-4.40) and hence there is a high confidence level in the experimental data." Although the general trends in stagnation region pressure and density, boundary layer thickness, and wall shear stress agree with theoretical predictions, the discrepancy between the computed and experimentally observed pitching moment derivative and drag coefficients suggests a weakness in modeling portions of the geometries and physics of the problem. Consequently, several areas require further examination.

4.5.1 Grid Refinement

Several numerically observed phenomena indicate that the computational grid is inadequate for the intricacies and complexities of the problem. The inadequacy of the grid manifests itself in several phenomena. The first of these is depicted in Figure 27 which shows the formation of a nonphysical solution in the stagnation region. Instead of a near-instanta-

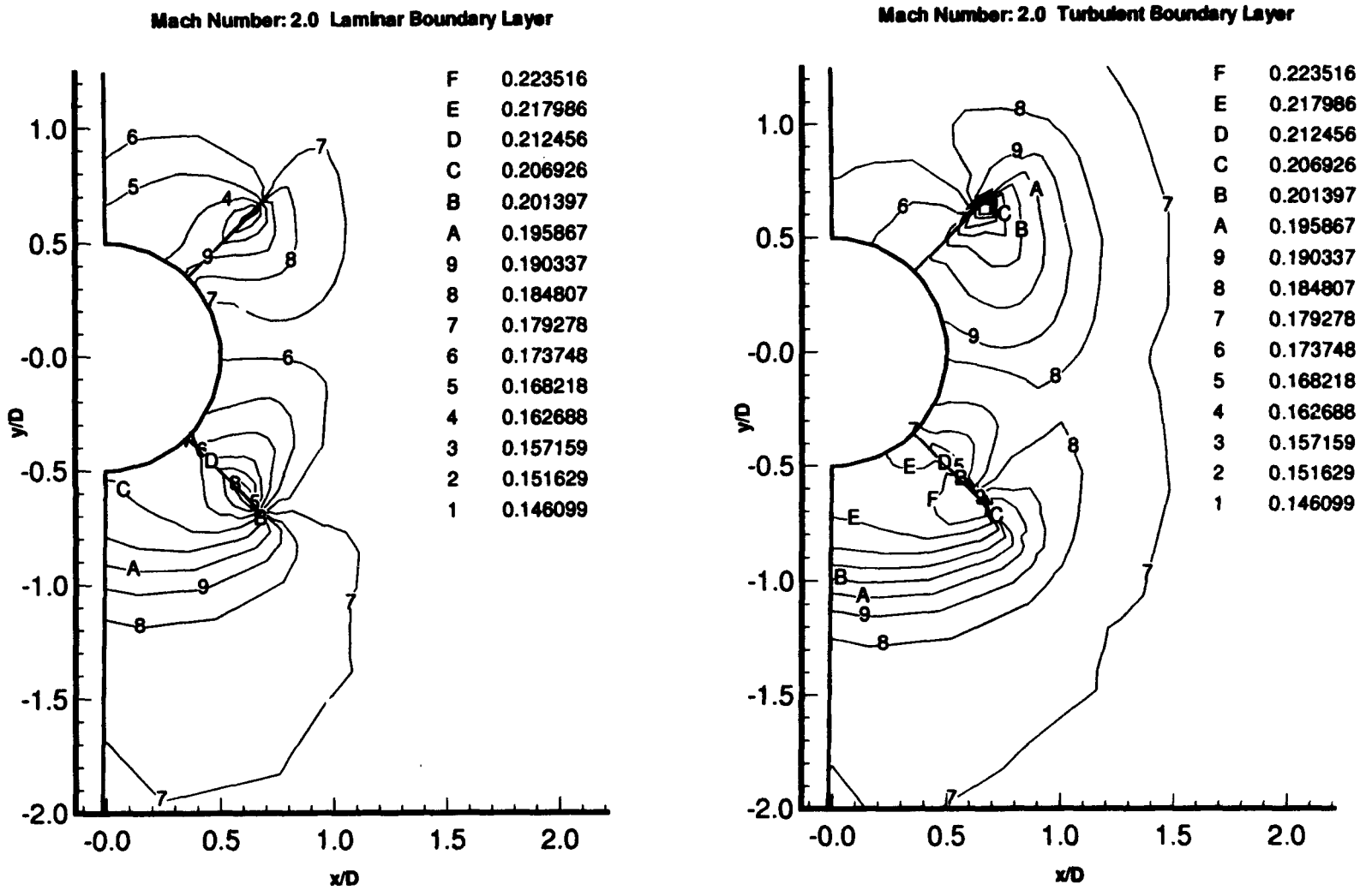


Figure 26: Fin Pressure Distributions for Laminar and Turbulent Boundary Layers

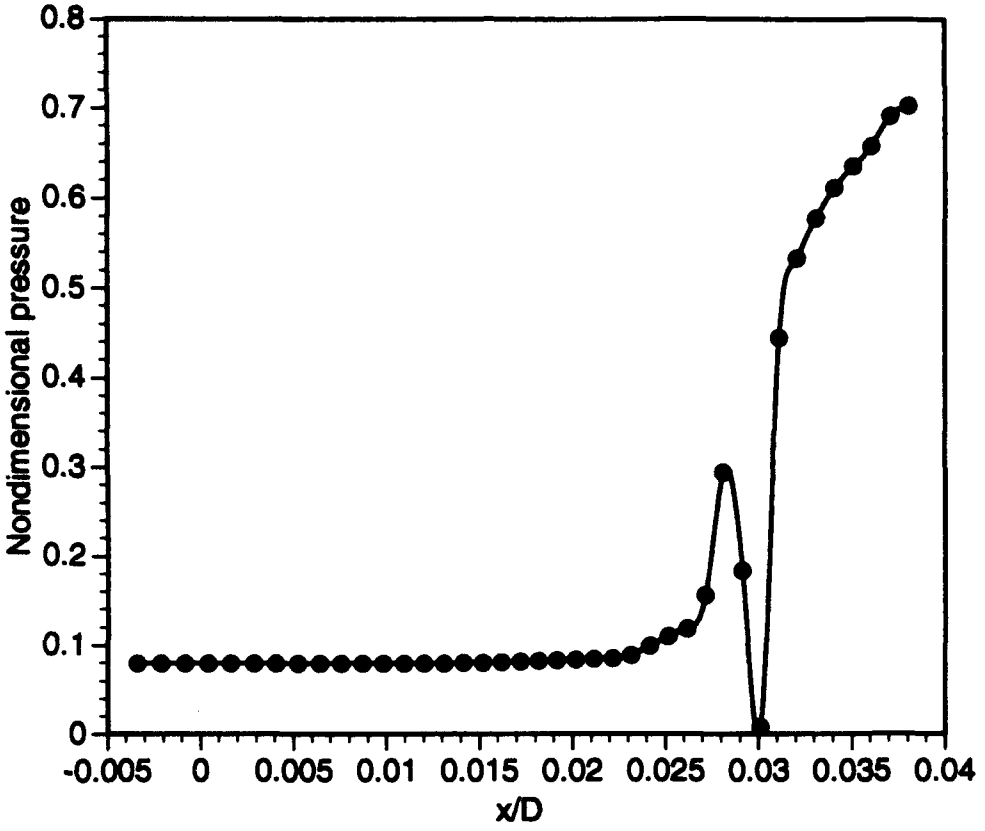


Figure 27: Developing Non-physical Solution

neous compression through the shock, there is an area of flow expansion as indicated by the rapid pressure drop. This is a well-documented occurrence of the Roe Scheme [10,27], and is known in the literature as a *carbuncle*. It is a nonphysical, entropy-violating expansion shock which occurs in the vicinity of sonic points and is caused by a nearly-zero value in one of the system eigenvalues at that point. The phenomena was observed in several of the solutions, and was only eliminated by adding artificial dissipation into the numerical scheme. Since dissipation represents an irretrievable loss of information, its excessive use is undesirable. Refinement of the grid in the stagnation region represents a more efficient method of mitigating the problem.

While a lack of grid refinement in the stagnation region can help to facilitate the formation of nonphysical solutions, grid coarseness in the fin region results in abnormal pres-

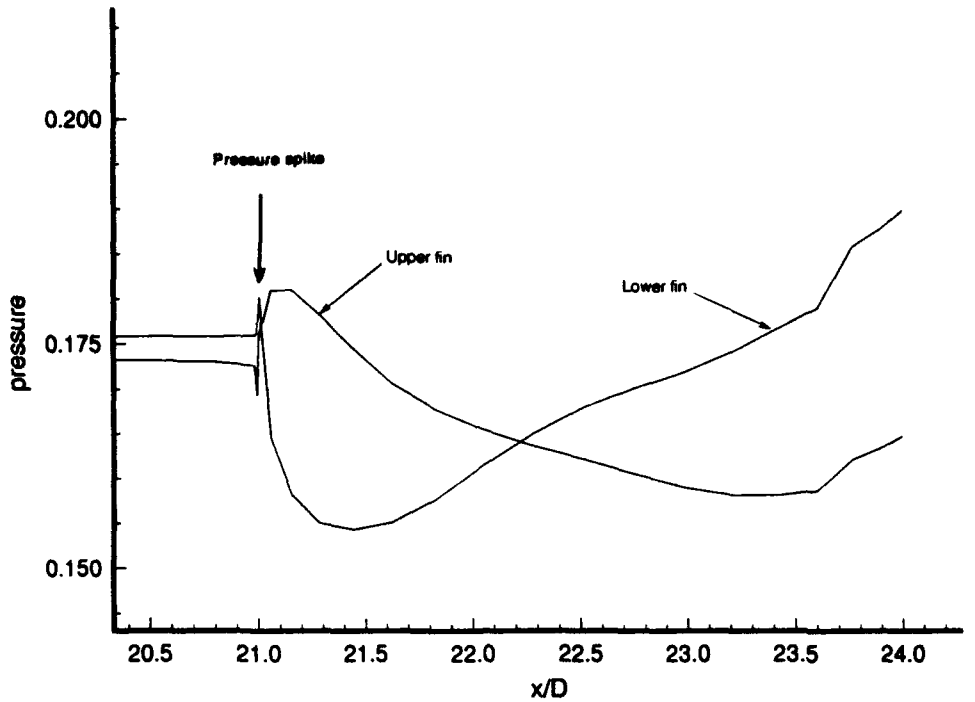


Figure 28: Abnormal Pressure Spike at Fin Leading Edge

sure distributions along the fins. This phenomena is depicted in Figure 28 which shows an abnormal pressure spike at the leading edge of the missile fin. The flow structure at the fin leading edge is extremely complicated due to the presence of shock/boundary layer interaction. As a result, the grid must be sufficiently refined to resolve the flow structures in this region.

The effect of insufficient grid refinement in the boundary layer manifests itself in the skin friction calculations. Moran [29] has examined the effects of wall spacing on the drag coefficient calculations and has found that decreasing the wall spacing from 0.001 to 0.000025 results in variations of as much of 20% in computed turbulent drag coefficient values. Laminar drag calculations, however, were relatively unaffected by this change. Quite obviously,

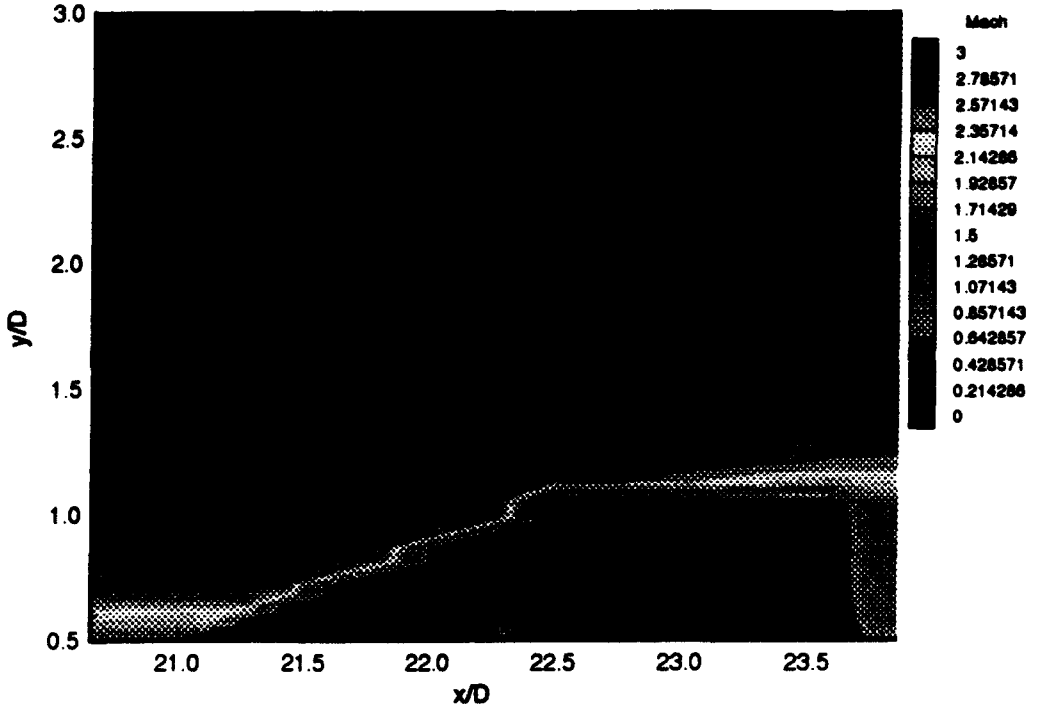


Figure 29: Fin Geometry

significantly more grid points are required in order to accurately model the problem.

4.5.2 Fin Model

The majority of the pitching moment is generated by the missile fins. It is therefore very important that the fin model accurately reflect the geometries of the actual missile. As mentioned previously, an infinitely-thin fin model was used in this study. Furthermore, no conformal gridding was used for the fins; they were simply implemented as a no-slip boundary condition. Because the fins are implemented as boundary conditions which are enforced at cell centers, the fin is thus restricted to only "exist" at cell centers. A finite discretization of the physical domain results in a fin model which does not completely reflect the physics of the actual fins. This is evident in Figure 29 which depicts the Mach contours around one of the missile fins. The darkest region of the figure represents the geometry of the fin. The inaccura-

cies inherent with this fin model are evident by the ragged profile of the fin leading edge. This causes perturbations in the flow field which are not reflected in the physical problem. A conformal gridding scheme would eliminate this inaccuracy.

A second source of inaccuracy stems from the infinitely-thin fin assumption. Figure 30 contains a series of images depicting cross-plane velocities at various positions along the body. Careful examination of Figure 30a reveals slight perturbations in the flow upstream of the fins. This indicates that information concerning the fins is propagated upstream through the subsonic boundary layer. Figure 30b shows the flow at the leading edge of the fins. The location of the fins is clearly visible by the flow deflection near the missile body. Figure 30c shows the velocity distribution four grid points aft of the fin leading edge while Figure 30d depicts the flow at the aft end of the missile. While the general behavior of the flow is as expected, the magnitudes of the cross-plane velocities are not indicative of a fin of finite thickness. Furthermore, since there is no area on the leading or tailing edges of the fins, there can be no pressure differential across these surfaces and subsequently no pressure drag. The work of Lakshmanan [23], however, indicates that there is indeed a pressure differential and a fairly substantial pressure drag contribution. This source of inaccuracy is the main reason the drag numbers reported in Figure 19 are significantly below the experimentally-obtained values.

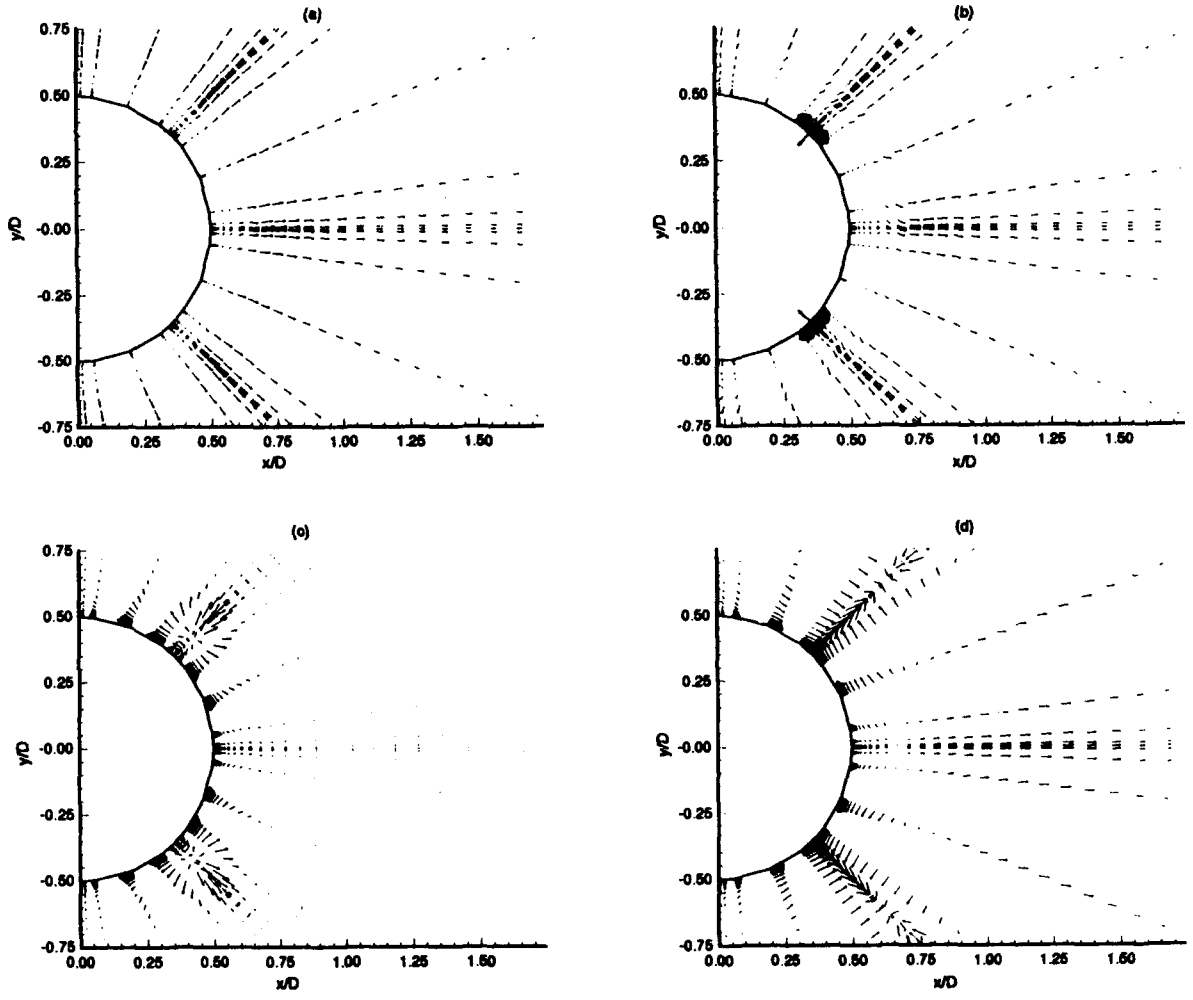


Figure 30: Cross-Plane Velocity Distributions at Various Missile Body Locations: (a) 10 Grid Points Forward of the Leading Edge, (b) At the Leading Edge, (c) 4 Points Aft of the Leading Edge, (d) Aft End of the Missile

V. Conclusions and Recommendations

5.1 Summary and Investigation Conclusions

The objectives of this investigation were

- to develop an algorithm capable of decomposing the computational domain into a series of subdomains or blocks such that calculations could be conducted in parallel over the blocks, and
- to use the code to study the aerodynamic characteristics of a sharp-nosed, high-speed missile configuration.

Both objectives were met. In terms of the first objective, PANS-3EM, a block-structured computer algorithm for the study of high-speed missile configurations was successfully developed and tested. The algorithm incorporates a block-table data structure which facilitates the management of the computational domains with very little memory penalty. This is extremely important given the memory limitations on many of the available computing platforms.

In its current form, the algorithm suffers performance penalties associated with the loop-level parallelism that was carried forward from the structure of the serial code. With further modification coupled with the use of operating-system-level parallelization routines, a more coarse-grained parallel approach can be taken to yield significant performance improvements. With the domain-decomposition routines in place, the code provides the flexibility for a subsequent modification to either shared- or distributed-memory computing platform.

With the first objective successfully completed, the aerodynamic characteristics of a sharp-nosed, high-speed missile configuration were investigated. Nose pressures and densities in the stagnation region accurately satisfied the Rankine-Hugonoit jump conditions. However, several solutions were plagued with the presence of a nonphysical expansion region in the vicinity of the sonic point which affected flow field and pressure distribution in the immediate vicinity of the stagnation point. The area over which this region affected the pressure distribution on the missile was not ascertained, and further examination of this phenomena is

warranted.

Particular attention was devoted to the behavior of the pitching moment derivative as a function of increasing Mach number. The reversal in the stability trend for increasing Mach number as obtained from experimental data was not verified by the computer code. Although a grid-sensitivity study was not conducted due to computing resource limitations, it is believed that the current grid does not possess the level of refinement necessary to capture all of the important aspects of the flow structure, especially in the stagnation region and the area of the fins. This was manifest in several phenomena including a 20% change in the computed turbulent drag coefficient as the wall spacing was refined from .001 to .000025. Additionally abnormal pressure spikes at the leading edge of the fins observed for the laminar boundary layer solutions. These spikes were not seen in the turbulent case, and it is believed that they would be eliminated given a sufficient level of grid refinement.

5.2 Suggested Areas for Further Study

5.2.1 Parallel Code Development

1) The domain-decomposition of PANS-3EM should be further explored so that the computational domains can be completely decoupled. This deccupling would facilitate a coarse-grained parallel approach which has the potential of much greater levels of performance and is a required step for implementation on a distributed-memory computing platform.

2) A comparative study of the use of buffer arrays to manage communication information vs. the block table would be beneficial in ascertaining the best use of computer resources in terms of memory/speed trade-offs. Since buffer arrays were not implemented in this investigation, no direct performance comparisons were possible.

3) The current code uses very large arrays to contain the flow variable and grid metric information. New, very-high performance, workstation-type computers depend heavily on memory caching in order to achieve maximum performance. The large data structures of the

current code do not map well to such a caching architecture, and thus it is difficult to achieve good computing performance. Taking advantage of the explicit nature of the TVD algorithm, it is possible to restructure the code such that computations are performed in a more *pipelined* fashion whereby all information for a given node at time level $n+1$ is computed before computations on the next node begins. The inherent parallelism of the algorithm is unaffected by such a modification.

5.2.2 Aerodynamic Issues

1) Given the sensitivity of the computed flow field on the calculation of the pitching moment coefficient, it may prove beneficial to explore the effect of various gridding techniques for the fins. Instead of treating the fins as an impermeability boundary condition enforced at certain computational mesh points, a conformal gridding approach may improve the accuracy of the calculations in the fin region. A finite-thickness fin model may also provide insight. Regardless of the method of fin implementation, a finer grid--both axially, radially, and azimuthally should be used.

2) Since boundary layer growth rates are different for laminar and turbulent cases, a study should be conducted concerning the effectiveness of the missile fins as a function of boundary layer thickness. This is especially important given the boundary layer/shock interaction phenomena that occurs at the leading edge of the fins.

3) The Baldwin-Lomax turbulence model used in this study is relatively easy to implement and has been shown to give acceptable results for fin/body junction calculations [23]. However, implementation of a two-equation model would provide a means of comparison of the performance of the turbulence model and its affect on the solution. A different model may provide the subtle difference necessary to verify the experimental data, or at the very least, it would provide a verification of the turbulence model currently in use.

Appendix A: Nondimensionalization of Governing Equations

Nondimensionalization of the governing equation follows the procedure of Beran [4] and is accomplished through use of the following parameters

$$x = x^* D \qquad y = y^* D \qquad z = z^* D$$

$$u = u^* U \qquad v = v^* U \qquad w = w^* U$$

$$T = T^* T_{ref} \qquad \rho = \rho^* \rho_{ref} \qquad t = t^* \left(\frac{D}{U}\right)$$

$$\mu = \mu^* \mu_{ref} \qquad p = p^* \rho_{ref} U^2 \qquad E_t = E_t^* \rho_{ref} U^2$$

where the nondimensional parameters are denoted by a superscript "*", D is the diameter of the missile body, and the subscript *ref* refers to a reference value.

Substituting these values into the governing equations yields the following set of non-dimensional equations.

Continuity

$$\frac{\partial(\rho^* \rho_{ref})}{\partial(t^* (D/U))} + \frac{\partial(\rho^* \rho_{ref} u^* U)}{\partial(x^* D)} + \frac{\partial(\rho^* \rho_{ref} v^* U)}{\partial(y^* D)} + \frac{\partial(\rho^* \rho_{ref} w^* U)}{\partial(z^* D)} = 0 \qquad (A.1)$$

Pulling constant terms outside the differentials and multiplying the equation by $(D/U)/\rho_{ref}$ gives

$$\frac{\partial(\rho^*)}{\partial(t^*)} + \frac{\partial(\rho^* u^*)}{\partial(x^*)} + \frac{\partial(\rho^* v^*)}{\partial(y^*)} + \frac{\partial(\rho^* w^*)}{\partial(z^*)} = 0. \qquad (A.2)$$

X-momentum

$$\frac{\partial(\rho^* \rho_{ref} u^* U)}{\partial(t^* (D/U))} + \frac{\partial(\rho^* \rho_{ref} u^{*2} U^2)}{\partial(x^* D)} + \frac{\partial(\rho^* \rho_{ref} u^* v^* U^2)}{\partial(y^* D)} + \frac{\partial(\rho^* \rho_{ref} u^* w^* U^2)}{\partial(z^* D)}$$

$$+\frac{\partial(p^*\rho_{ref}U^2)}{\partial(x^*D)} = \frac{\partial}{\partial(x^*D)}\left(\frac{2}{3}\mu^*\mu_{ref}\left(2\frac{\partial u^*U}{\partial x^*D} - \frac{\partial v^*U}{\partial y^*D}\right)\right) \qquad (A.3)$$

$$+\frac{\partial}{\partial(y^*D)}\left(\mu^*\mu_{ref}\left(\frac{\partial u^*U}{\partial y^*D} - \frac{\partial v^*U}{\partial x^*D}\right)\right) + \frac{\partial}{\partial(z^*D)}\left(\mu^*\mu_{ref}\left(\frac{\partial u^*U}{\partial z^*D} - \frac{\partial w^*U}{\partial x^*D}\right)\right)$$

where the viscous stress terms have been expanded into their velocity-derivative components. Again pulling constant terms out of the differentials and multiplying the resulting expression by $L/(\rho_{ref}U^2)$, equation A.3 becomes

$$\frac{\partial(\rho^*u^*)}{\partial t^*} + \frac{\partial(\rho^*u^{*2})}{\partial x^*} + \frac{\partial(\rho^*u^*v^*)}{\partial y^*} + \frac{\partial(\rho^*u^*w^*)}{\partial z^*} + \frac{\partial(p^*)}{\partial x^*} =$$

$$\frac{1}{Re}\left(\frac{\partial}{\partial x^*}\left(\frac{2}{3}\mu^*\left(2\frac{\partial u^*}{\partial x^*} - \frac{\partial v^*}{\partial y^*}\right)\right) + \frac{\partial}{\partial y^*}\left(\mu^*\frac{\partial u^*}{\partial y^*} - \frac{\partial v^*}{\partial x^*}\right) + \frac{\partial}{\partial z^*}\left(\mu^*\frac{\partial u^*}{\partial z^*} - \frac{\partial w^*}{\partial x^*}\right)\right) \qquad (A.4)$$

where *Re* is the Reynolds number based on missile body diameter.

A similar process yields expressions for the y- and z-momentum equations.

Energy

For the sake of brevity, the derivation of the energy equation is presented somewhat more succinctly. Constant terms are pulled outside the derivatives and terms involving the viscous stress components are not placed in terms of velocity derivatives. This results in

$$\frac{\rho_{ref}U^3}{D}\left(\frac{\partial E_t^*}{\partial t^*} + \frac{\partial(E_t^*u^*)}{\partial x^*} + \frac{\partial(E_t^*v^*)}{\partial y^*} + \frac{\partial(E_t^*w^*)}{\partial z^*}\right)$$

$$-\frac{\kappa_{ref}T_{ref}}{D^2}\left(\frac{\partial(\kappa^*\frac{\partial T^*}{\partial x^*})}{\partial x^*} + \frac{\partial(\kappa^*\frac{\partial T^*}{\partial y^*})}{\partial y^*} + \frac{\partial(\kappa^*\frac{\partial T^*}{\partial z^*})}{\partial z^*}\right) =$$

$$\frac{\mu_{ref}U^2}{D^2}\left((\tau^*_{xx} + \tau^*_{yx} + \tau^*_{zx})u^* + (\tau^*_{yx} + \tau^*_{yy} + \tau^*_{zy})v^* + (\tau^*_{zx} + \tau^*_{zy} + \tau^*_{zz})w^*\right)$$

$$-\frac{\rho_{ref}U^3}{D}\left(\frac{\partial(p^*u^*)}{\partial x^*}+\frac{\partial(p^*v^*)}{\partial y^*}+\frac{\partial(p^*w^*)}{\partial z^*}\right) \qquad (A.5)$$

After multiplying equation A.5 by $D/(\rho_{ref}U^3)$, the coefficient of the viscous terms is given as

$$\frac{\mu_{ref}}{\rho_{ref}UD}=\frac{1}{Re} \qquad (A.6)$$

and the coefficient of the heat-flux terms is found to be

$$\frac{\kappa_{ref}T_{ref}}{\rho_{ref}U^3D} \qquad (A.7)$$

Applying the definition of Mach number and Prandtl number to the terms of equation A.7 yields the relation

$$\frac{\kappa_{ref}T_{ref}}{\rho_{ref}U^3D}=\frac{1}{RePr(\gamma-1)M_{ref}^2} \qquad (A.8)$$

Finally, from Beran [4],

$$\kappa^*=\mu^* \qquad (A.9)$$

Thus, the energy equation becomes

$$\frac{\partial E_t^*}{\partial t^*}+\frac{\partial(E_t^*u^*)}{\partial x^*}+\frac{\partial(E_t^*v^*)}{\partial y^*}+\frac{\partial(E_t^*w^*)}{\partial z^*}-$$

$$\frac{1}{RePr(\gamma-1)M_{ref}^2}\left(\frac{\partial(\mu^*\frac{\partial T^*}{\partial x^*})}{\partial x^*}+\frac{\partial(\mu^*\frac{\partial T^*}{\partial y^*})}{\partial y^*}+\frac{\partial(\mu^*\frac{\partial T^*}{\partial z^*})}{\partial z^*}\right)=$$

$$\frac{1}{Re}\left((\tau^*_{xx}+\tau^*_{yx}+\tau^*_{zx})u^*+(\tau^*_{yx}+\tau^*_{yy}+\tau^*_{zy})v^*+(\tau^*_{zx}+\tau^*_{zy}+\tau^*_{zz})w^*\right)$$

$$-\left(\frac{\partial(p^*u^*)}{\partial x^*}+\frac{\partial(p^*v^*)}{\partial y^*}+\frac{\partial(p^*w^*)}{\partial z^*}\right). \qquad (A.10)$$

Appendix B: Parallelization Issues

This appendix discusses the issues which must be addressed during parallelization of a CFD code. It is provided as a road-map for future parallelization efforts. General issues including handling of boundary conditions and synchronization issues are presented along with specifics of the implementation of the parallel TVD scheme used in this project.

B.1 Boundary Conditions

Boundaries for the blocked domains can be divided into two groups: one group contains those boundaries which represent physical boundaries and are treated via the mathematical boundary conditions imposed by the problem physics, while the second group contains those boundaries which are introduced in the blocking process. Unlike the boundaries in the first group, these boundaries have no physical basis and consequently, their presence must not affect the solution in any way. Figure B.1 depicts these two types of boundaries. In the figure, the continuous computational domain has been cut into two blocks and the blocks have been rolled back to reveal the newly-created boundaries represented by the shaded surface on each block.

The domain decomposition does not alter the method by which the physically based boundary conditions are handled. However, since the boundary conditions associated with each face of the domain can be unique, this poses a problem for homogeneous parallel programming in which all threads of execution carry out the same set of instructions. For example, at the leading edge of the computational domain in this implementation, ghost points are reflected across the stagnation line in order to provide suitable handling of the boundary conditions and the required number of points for second-order accuracy of the finite-difference or finite-volume scheme. A typical code fragment which handles the boundary condition updates for these points follows. For this code fragment as well as all following fragments, the following convention is used:

- The code fragments appear in FORTRAN.

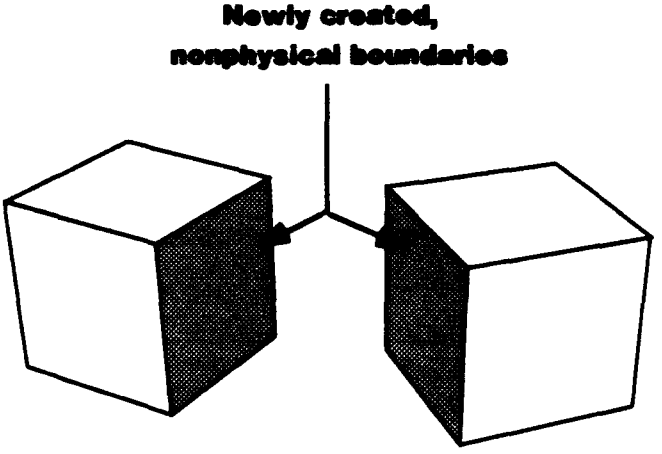


Figure B.1: Blocked Domain Boundary Types

- **Most fragments contains a serial code section and its corresponding PANS-3EM code section**
- **Sections of code of particular importance to the blocking process appear underlined.**
- **Lines preceded by a "/*" designate comment lines.**
- **The variable name *numblk* refers to the number of blocks in the block-structured computational domain.**
- **Look-ups to the block table are denoted as *iblktb*.**

Code Fragment 1: Boundary Condition Treatment

Serial Code

```
do 10 j = 1, jmax
do 10 k = 2, kmax-2
   rho(0,j,k) = 6.*rho(2,j,k)-8.*rho(3,j,k)+3.*rho(4,j,k)
   u(0,j,k)   = 6.*u(2,j,k)-8.*u(3,j,k)+3.*u(4,j,k)
   v(0,j,k)   = 6.*v(2,j,k)-8.*v(3,j,k)+3.*v(4,j,k)
   w(0,j,k)   = 6.*w(2,j,k)-8.*w(3,j,k)+3.*w(4,j,k)
   p(0,j,k)   = 6.*p(2,j,k)-8.*p(3,j,k)+3.*p(4,j,k)
   et(0,j,k)  = p(0,j,k)/gm1 + .5*rho(0,j,k)*
      (u(0,j,k)**2+v(0,j,k)**2+w(0,j,k)**2)

   vu(0,j,k,n,1) = rho(0,j,k)
   vu(0,j,k,n,2) = rho(0,j,k)*u(0,j,k)
   vu(0,j,k,n,3) = rho(0,j,k)*v(0,j,k)
   vu(0,j,k,n,4) = rho(0,j,k)*w(0,j,k)
   vu(0,j,k,n,5) = et(0,j,k)
10 continue
```

PANS-EM Code

```
do 10 L = 1, numblk
   call indxcp(L,0,0,1,jmax,kmin,kmax,iblkmn,iblkmx,
               jblkmn,jblkmx,kblkmn,kblkmx)
   do 10 i = iblkmn, iblkmx
   do 10 j = 1, jmax
   do 10 k = 1, kmax
      rho(i,j,k,L) = 6.*rho(i+2,j,k,L)-8.*rho(i+3,j,k,L)+3.*rho(i+4,j,k,L)
      u(i,j,k,L)   = 6.*u(i+2,j,k,L)-8.*u(i+3,j,k,L)+3.*u(i+4,j,k,L)
      v(i,j,k,L)   = 6.*v(i+2,j,k,L)-8.*v(i+3,j,k,L)+3.*v(i+4,j,k,L)
      w(i,j,k,L)   = 6.*w(i+2,j,k,L)-8.*w(i+3,j,k,L)+3.*w(i+4,j,k,L)
      p(i,j,k,L)   = 6.*p(i+2,j,k,L)-8.*p(i+3,j,k,L)+3.*p(i+4,j,k,L)
      et(i,j,k,L)  = p(i,j,k,L)/gm1 + .5*rho(i,j,k,L)*
                     (u(i,j,k,L)**2+v(i,j,k,L)**2+w(i,j,k,L)**2)

      vu(i,j,k,L,n,1) = rho(i,j,k,L)
      vu(i,j,k,L,n,2) = rho(i,j,k,L)*u(i,j,k,L)
      vu(i,j,k,L,n,3) = rho(i,j,k,L)*v(i,j,k,L)
      vu(i,j,k,L,n,4) = rho(i,j,k,L)*w(i,j,k,L)
      vu(i,j,k,L,n,5) = et(i,j,k,L)
10 continue
```

Although the restructured code allows all threads of execution to execute the loop, those blocks not containing the ghost points will merely continue since the *iblkmn* and *iblkmx* loop indices returned by the call to *indxcp* will be 0 and -1[1]. Depending on the parallel implementation and synchronization method used, execution threads which handle blocks not involved in this loop may be able to continue on to the next code segment. If loop-level parallelism is used, or if the code requires synchronization before the next code segment can execute, then threads will be forced to wait or *spin* until the thread handling the ghost point completes execution of the loop. This is a potential source of inefficiency which can greatly limit the performance gains of parallelism. The same problem holds true for the remaining boundaries of the computational domains and extends to any section of code which handles only a portion of the computational domain such as those loops which only operate over the missile fins. Future implementations should investigate a more complete decoupling of the domains. Several options are available, including an object-oriented programming approach in which the domains are constructed as *classes*. Boundary condition updates can be performed through a *class method* which can be implemented such that all blocks update boundary information simultaneously.

The reader should note that the *i+2*, *i+3*, and *i+4* array element references appearing in the above code fragment are not the correct way to handle the *i=2*, *i=3*, and *i=4* references

1. A discussion of the subroutine *indxcp* and its associated block table appears in section B.2.

of the original code segment and only appear as such here to avoid complication of the loop for this discussion. A discussion of the proper handling of these references appears in Sections B.3 and B.4.

B.2 Inter-Domain Communication

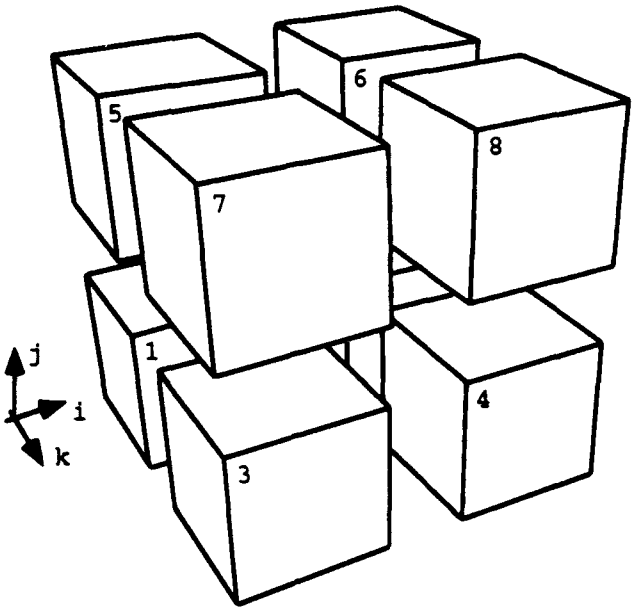
All across-block references can be thought of as inter-domain communication. Several methods were investigated for handling these references. In the current implementation, any *i-1* or *i+1* reference in the code constitutes a possible across-block reference. Although not implemented in the current version of the computer code, a *j-1*, *j+1*, *k-1*, or *k+1* reference also constitutes a potential across-block reference in a completely generic domain-blocking scheme.

Because the use of buffer arrays was determined to be impractical due to memory constraints, a block table data structure was developed to manage the flow of information between the blocks of the computational domain. The block table was implemented as a simple two-dimensional array. Figure A.2 shows a sample block table for a fully generic domain decomposition case while Figure A.3 shows the block table and domain decomposition used in this project. The boldly outlined cells in the block tables represent that portion of the block table actually implemented in the data structure. Although the block table serves a key purpose in the domain-decomposition process, blocking the domain by simply using the block table to index the program loops becomes very convoluted. This is illustrated in code fragment 2 which is taken from the subroutine which computes the velocity and temperature gradient terms (subroutine GRADIENT).

Code Fragment 2: Block Table Usage

Serial Code

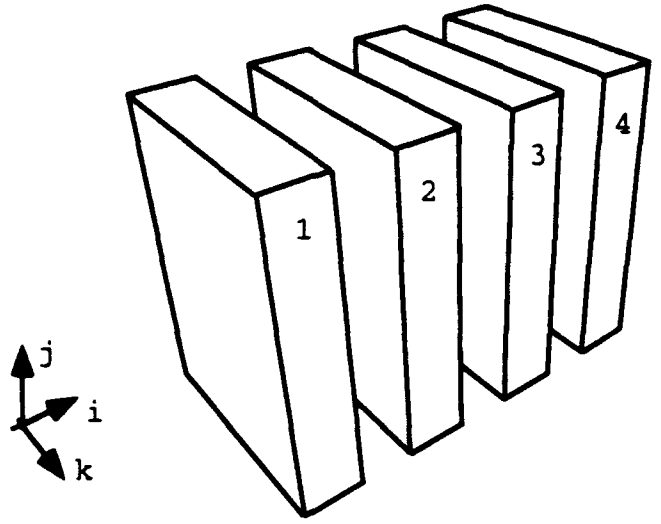
```
do 25 k = 1, kmax-1
do 25 j = 2, jmax-1
do 25 i = 2, imax-1
dudx(i,j,k)=.5*(.5*(xnip(i,j,k)+xnip(i-1,j,k))*
             (u(i+1,j,k)-u(i-1,j,k)) + .5*(xejp(i,j,k)+xejp(i,j-1,k))*
             (u(i,j+1,k)-u(i,j-1,k)) + .5*(xzkp(i,j,k)+xzkp(i,j,k-1))*
             (u(i,j,k+1)-u(i,j,k-1)))
25   continue
```



Block Num	Neighboring Block						Max Coordinate		
	imin	imax	jmin	jmax	kmin	kmax	i	j	k
1	0	2	0	5	0	3	31	41	18
2	1	0	0	6	0	4	61	41	18
3	0	4	0	7	1	0	31	41	35
4	3	0	0	8	2	0	61	41	35
5	0	6	1	0	0	7	31	81	18
6	5	0	2	0	0	8	61	81	18
7	0	8	3	0	5	0	31	81	35
8	7	0	4	0	6	0	61	81	35

max coordinates given for a 61 x 81 x 35 computational grid

Figure B.2: Generic block structure and associated block table



Block Num	Neighboring Block						Max Coordinate		
	imin	imax	jmin	jmax	kmin	kmax	i	j	k
1	0	2	0	0	0	0	15	81	35
2	1	3	0	0	0	0	31	81	35
3	2	4	0	0	0	0	47	81	35
4	3	0	0	0	0	0	61	81	35

Figure B.2: Project block structure and associated block table

Using the block table to reconstruct this loop requires special attention to the *i-1* and *i+1* across-block references. A total of four loops are thus required: two loops to handle the interior points for the blocks and two loops to handle the across-block references. The code is then restructured as follows.

PANS-3EM Code

```
/*Handle the first block except across-block references.
```

```
L = 1
do 10 k = 1, kmax-1
do 10 j = 2, jmax-1
do 10 i = 2, iblktb(L,7)-1
   dudx(i,j,k,L)=.5*(.5*(xnip(i,j,k,L)+xnip(i-1,j,k,L))*
        (u(i+1,j,k,L)-u(i-1,j,k,L)) + .5*(xejp(i,j,k,L)+xejp(i,j-1,k,L))*
        (u(i,j+1,k,L)-u(i,j-1,k,L)) + .5*(xzkp(i,j,k,L)+xzkp(i,j,k-1,L))*
        (u(i,j,k+1,L)-u(i,j,k-1,L)))
10 continue

/*Handle the remaining blocks, but avoid across-block references.

do 20 L = 2, numblk
do 20 k = 1, kmax-1
do 20 j = 2, jmax-1
do 20 i = 1, iblktb(L,7)-1
   dudx(i,j,k,L)=.5*(.5*(xnip(i,j,k,L)+xnip(i-1,j,k,L))*
        (u(i+1,j,k,L)-u(i-1,j,k,L)) + .5*(xejp(i,j,k,L)+xejp(i,j-1,k,L))*
        (u(i,j+1,k,L)-u(i,j-1,k,L)) + .5*(xzkp(i,j,k,L)+xzkp(i,j,k-1,L))*
        (u(i,j,k+1,L)-u(i,j,k-1,L)))
20 continue

/*Handle the i+1 across-block references.

do 30 L = 1, numblk-1
do 30 k = 1, kmax-1
do 30 j = 2, jmax-1
i = iblktb(L,7)
   dudx(i,j,k,L)=.5*(.5*(xnip(i,j,k,L)+xnip(i-1,j,k,L))*
        (u(1,j,k,iblktb(iblktb(L,2))-u(i-1,j,k,L)) +
        .5*(xejp(i,j,k,L)+xejp(i,j-1,k,L))*
        (u(i,j+1,k,L)-u(i,j-1,k,L)) + .5*(xzkp(i,j,k,L)+xzkp(i,j,k-1,L))*
        (u(i,j,k+1,L)-u(i,j,k-1,L)))
30 continue

/*Handle the i-1 across-block references

do 40 L = 2, numblk
do 40 k = 1, kmax-1
do 40 j = 2, jmax-1
i = 1
   dudx(i,j,k,L)=.5*(.5*(xnip(i,j,k,L)+
        xnip(iblktb(iblktb(L,7)),j,k,iblktb(L,1)))*
        (u(i+1,j,k,L)-u(iblktb(iblktb(L,7)),j,k,iblktb(L,1)) +
        .5*(xejp(i,j,k,L)+xejp(i,j-1,k,L))*
        (u(i,j+1,k,L)-u(i,j-1,k,L)) + .5*(xzkp(i,j,k,L)+xzkp(i,j,k-1,L))*
        (u(i,j,k+1,L)-u(i,j,k-1,L)))
40 continue
```

The methodology used in code fragment 2 is readily adaptable to a distributed-memory architecture since the indices which contain block table references would be implemented via message passing on a distributed-memory machine. Using the Intel Hypercube message-passing paradigm as an example [21], a message consists of five parts which include the message data type, message buffer, message length, receiving node identifier, and receiving process identifier. Examining the underlined variable *xnip*, the message data type is the type of the variable *xnip* (double-precision real), the buffer corresponds to the particular array element of *xnip*, the message length is 8 bytes for a double-precision array element, and the

receiving node identifier is denoted in *iblktb(L,1)*. The receiving process identifier depends on the implementation.

The utility of the buffer arrays mentioned in Chapter 3 is also obvious in code fragment 2. The existence of buffer arrays eliminates the need for special handling of the across-block references in the third and fourth loops. The arrays would simply be updated once at the beginning of each sweep and the indexing on the i variable incremented over the block boundaries. Again, while this method is very efficient in terms of execution speed (once the arrays are updated), the memory penalty can be prohibitive.

The implementation shown in code fragment 2 does not increase the order of complexity of the computer code; it is still $O(n)$ where n is the total number of grid points in the computational mesh. In fact, the only overhead incurred in this implementation is that associated with setting up the additional looping structures and with the double lookup of array indices given by *iblktb*(*iblktb*(*L*,7)). Despite this fact, the code is fraught with several weaknesses. They include:

- The code relies on the fact that the first block along the i direction is block 1 and the last block in the i direction contains the last i point. This may not be the case for a completely generic implementation.
- The volume of additional code generated makes the introduction of mistakes very likely.
- Each loop handles only a portion of the blocks, thus making an efficient parallel implementation very unlikely.
- The code structure does not handle computational-to-block coordinate transformations well.

While the first three weakness are problematic, the fourth makes the approach shown in code fragment 2 unusable.

B.3 Computational and Block Coordinates

Once the computational domain has been decomposed, the ordered triple locating a variable in the computational domain cannot be used to locate that variable in the block coordinate space. This poses a challenging problem in determining proper loop indices for the program loop constructs. An example involving the missile fins amply illustrates this point.

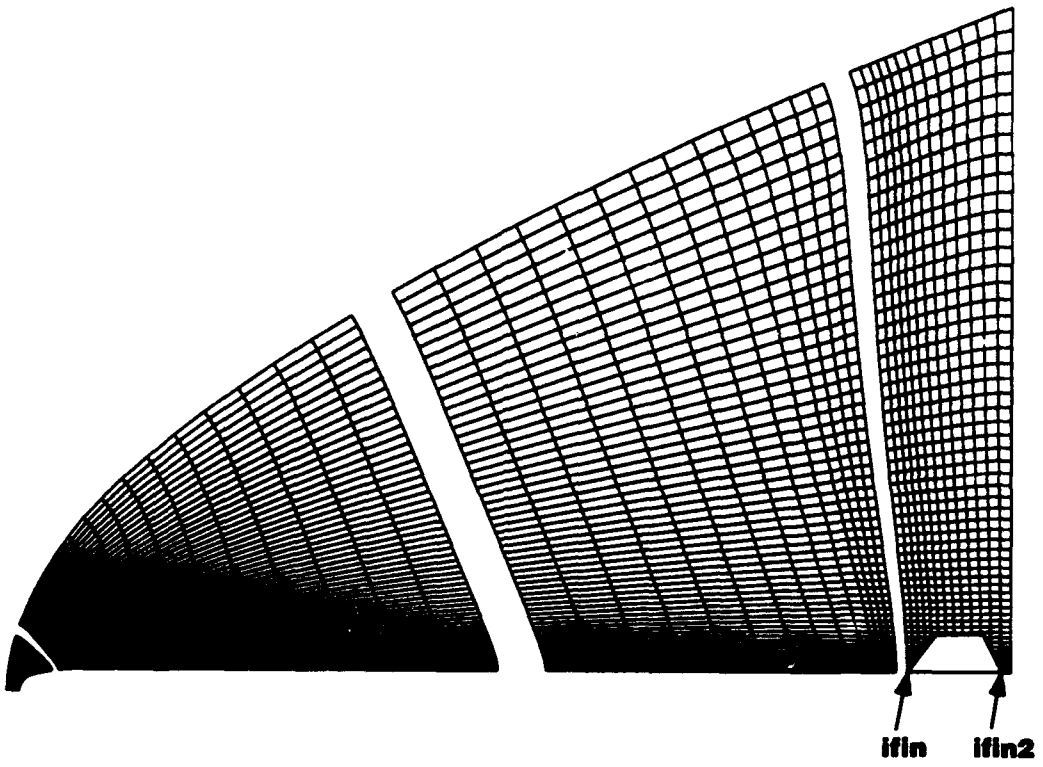


Figure B.3: Computational-to-Block Coordinate Transformation

Calculation of the gradients at the missile fins require special handling, since the fins are implemented as a impermeability boundary condition in the computer code. Consequently, there are code loops with indices of *ifin*, *ifin2*, *ifin-1*, *ifin2-2*, etc. While it is very straightforward to determine the location of *ifin2-1* given the location *ifin2* in the computational coordinate space, it is much less a trivial problem in the block coordinate space. Although the fin configuration in the domain decomposition used in this project did not pose a great problem, it is possible that a different decomposition could yield the configuration depicted in Figure B.3. Because the leading edge of the fin (*ifin*) appears on a block boundary, *ifin-1* is not found simply by looking one location to the left; instead, it appears in a different block. A loop construct with indices of *ifin-1* and *ifin2* would thus require careful handling. The following code fragment loops over the region of the physical domain from the leading edge of the missile fins to one point before the trailing edge of the missile fins.

Code Fragment 3: Loop Index Example

Serial Code

```
do 310 j = ifin, ifin2-1
do 310 j = ifin(j)+1, jmax-1

   signx = sign(1.00,alpha(i,j,1,2))
   darg1 = min(abs(alpha(i,j,1,2)),signx*alpha(i,j,1,1))
   g(i,j,1,2) = signx * max(0.00,darg1)

   signx = sign(1.00,alpha(i,j,2,2))
   darg1 = min(abs(alpha(i,j,2,2)),signx*alpha(i,j,2,1))
   g(i,j,2,2) = signx * max(0.00,darg1)

   signx = sign(1.00,alpha(i,j,3,2))
   darg1 = min(abs(alpha(i,j,3,2)),signx*alpha(i,j,3,1))
   g(i,j,3,2) = signx * max(0.00,darg1)

   signx = sign(1.00,alpha(i,j,4,2))
   darg1 = min(abs(alpha(i,j,4,2)),signx*alpha(i,j,4,1))
   g(i,j,4,2) = signx * max(0.00,darg1)

   signx = sign(1.00,alpha(i,j,5,2))
   darg1 = min(abs(alpha(i,j,5,2)),signx*alpha(i,j,5,1))
   g(i,j,5,2) = signx * max(0.00,darg1)
310 continue
```

If it is assumed that the block decomposition varies depending on the number of grid point and the number of available processors, then it cannot be known *a priori* in which block the leading and trailing edge of the fins will exist. Furthermore, any additional geometries or features to the missile which require special numerical computations will greatly exacerbate this problem. Code fragment 2 required special loops to handle across-block references, yet it is obvious that it is not possible to unequivocally state whether or not across-boundary references are required in a looping construct between two arbitrary i coordinates. This makes the code listed in code fragment 2 highly specific and impractical to generalize for handling arbitrary geometry or flow-field specific calculations.

B.4 Final Implementation

The coordinate transformation and across-block-reference problems were solved via the implementation of a subroutine which takes as an input a computational coordinate and computes proper loop indices in block-coordinate space. With this implementation, code fragment 3 becomes

PANS-3EM Code

```
do 10 L = 1, numblk
   call indxcp(L,ifin,ifin2-1,jmin,jmax,kmin,kmax,iblkmn,iblkmx,
               jblkmn,jblkmx,kblkmn,kblkmx)
   do 10 i = iblkmn, iblkmx
   do 10 j = ifin(L,i)+1, jmax-1

      signx = sign(1.00,alpha(i,j,L,1,2))
      darg1 = min(abs(alpha(i,j,L,1,2)),signx*alpha(i,j,L,1,1))
      g(i,j,L,1,2) = signx * max(0.00,darg1)

      signx = sign(1.00,alpha(i,j,L,2,2))
      darg1 = min(abs(alpha(i,j,L,2,2)),signx*alpha(i,j,L,2,1))
      g(i,j,L,2,2) = signx * max(0.00,darg1)

      signx = sign(1.00,alpha(i,j,L,3,2))
      darg1 = min(abs(alpha(i,j,L,3,2)),signx*alpha(i,j,L,3,1))
      g(i,j,L,3,2) = signx * max(0.00,darg1)

      signx = sign(1.00,alpha(i,j,L,4,2))
      darg1 = min(abs(alpha(i,j,L,4,2)),signx*alpha(i,j,L,4,1))
      g(i,j,L,4,2) = signx * max(0.00,darg1)

      signx = sign(1.00,alpha(i,j,L,5,2))
      darg1 = min(abs(alpha(i,j,L,5,2)),signx*alpha(i,j,L,5,1))
      g(i,j,L,5,2) = signx * max(0.00,darg1)
10 continue
```

The subroutine *indxcp* utilizes the block table in conjunction with the coordinate transformations given by equations 39 and 40 to compute proper loop indices for *iblkmn* and *iblkmx*. As an example, taking *ifin* = 44 and *ifin2-1* = 58, the computed values of the loop indices returned by *indxcp* are provided in Table B.1.

Table B.1: Example loop limiter values returned by *indxcp*

Block Number (L)	*iblkmn*	*iblkmx*
1	0	-1
2	0	-1
3	14	16
4	[illegible]	11

In this example, blocks 1 and 2 perform no computational work within the loop. Thus, processors handling those two loops can either be allowed to continue execution on other loops, or if synchronization is required, can be forced to wait for other processors to finish the loop.

Although the subroutine *indxcp* addresses the issue of computational to block coordinate transformations and eliminates any specialized loops which are specific to certain blocks– as seen in code fragment 2–across-block references are still not properly treated. For this reason, an additional routine was added to all loops containing across-block references. The use of this routine is shown in the following code fragment taken from the turbulent viscosity calculation subroutine.

Code Fragment 4: Across-Block Reference Check

Serial Code

```
do 116 i=itrans,iend-1
   if (vort(i,jbegin(i)-1,k) .lt. .1) then
      bstar = 0.
   else
      dpdxi = 0.5*(p(i+1,jbegin(i)-1,k)-p(i-1,jbegin(i)-1,k))
   endif
116 continue
```

PANS-3EM Code

```
do 116 L = 1, numblk
   call indxcp(L,itrans,iend-1,jmin,jmax,kmin,kmax,iblkmn,iblkmx,
               jblkmn,jblkmx,kblkmn,kblkmx)
   do 116 i = iblkmn, iblkmx
      if( i .eq. iblkmn ) then
         L1m = L
         im1 = i
         ip1 = i+1
         L1p = L
         idir = -1
         call offset( L1m, im1, idir, 1 )
      elseif( i .eq. iblkmx ) then
         L1m = L
         im1 = i-1
         L1p = L
         idir = 1
         call offset( L1p, ip1, idir, 1 )
      else
         L1m = L
         im1 = i-1
         L1p = L
         ip1 = i+1
      endif

      if( vort(i,jbegin(L,i)-1,k,L) .lt. .1 ) then
         bstar = 0.
      else
         dpdxi=0.5*(p(ip1,jbegin(L,i)-1,k,L1p)-p(im1,jbegin(L,i)-1,k,L1m))
      endif
116 continue
```

Subroutine *offset* takes as inputs the *i* coordinate in block coordinate space as the first two parameters of the subroutine along with the direction of the offset (either -1 or +1 for a

decrement or increment, respectively) in the third argument and finally the number of points to offset from the current block coordinate in the fourth parameter, and the new block coordinates are returned in the first two subroutine arguments. With this final addition to all loops containing across-block references, the block-decomposition process is complete. Advantages to this method include:

- The implementation is extremely flexible. Given the block table and proper coordinate transformation equations, any looping index can be properly treated.
- Modifications to the looping calculations are localized to a single subroutine rather than scattered through the code.
- A blocking implementation in the other two coordinate directions can be coded into the subroutine to allow for a completely generic block decomposition.
- No memory is wasted in buffer arrays.

The code listings for subroutines *indxcp* and *offset* appear in Appendix D. Examination of the code reveals that both subroutines are of $O(1)$ and thus they do not affect the complexity of the computer code. The overhead associated with the subroutine call can be reduced by inlining the subroutines. There are, however, performance penalties associated with the domain decomposition which are discussed in Chapter 4.

B.5 Vectorization issues

Although the serial code was written to be highly vectorizable, this was not a priority in the blocked version of the code since near-term target machines for implementation[1] were all RISC-based scalar machines. Consequently, the reduction of code vectorizability caused by the addition of the conditional such as that appearing in code fragment 4 was deemed an acceptable trade-off for the lower memory required when compared to the buffer-array concept. Code fragment 5 shows a typical looping construct similar to that given in code fragment 5 along with a vectorization summary from the Convex C220 Fortran compiler. The loop body is omitted in this example since it is not required to demonstrate the loop vectorizability.

---

1. Current implementation plans are for a shared memory DEC alpha-based system, and a distributed memory workstation farm concept. Longer-term plans call for implementation on the Intel i860™ XP-based Paragon, a vector machine.

Code Fragment 5: Vectorizability Issues

Serial Code

```
do 8881 k = 2, kmax-2
   do 100 j = 1, jmax-1
   do 100 i = 2, ifin-1

/*loop body with across-block references appears here

100   continue
8881  continue
```

The vectorization summary for this loop is given as

Iterative Variable	Reordering Transformation
K	Parallel
J	Scalar
I	FULL VECTOR

PANS-3EM Code

```
do 8881 k = 2, kmax-2
do 100 L = 1, numblk
   call indxcp(L,2,ifin-1,jmin,jmax,kmin,kmax,iblkmn,iblkmx,jblkmn,
               jblkmx, kblkmn,kblkmx)
   do 100 i = iblkmn, iblkmx
      if( i .eq. iblkmn ) then
         L1m = L
         im1 = i
         idir = -1
         call offset( L1, im1, idir, 1 )
      else
         im1 = i-1
         L1m = L
      endif

      do 100 j = 1, jmax-1

/*loop body with across-block references appears here

100   continue
8881  continue
```

The vectorization summary for this loop is

Iterative Variable	Reordering Transformation
L	Scalar
I	Scalar
J	FULL VECTOR Interchanged
K	Parallel

The conditional destroys the vectorizability over the *i* index. On the Convex C220, the compiler instead vectorized over the *j* index. This vectorization is not as efficient due to the fact that vectorization over the second array index results in a non-stride-one or non-contigu-

ous memory access which can be considerably slower than a stride-one memory access [8]; however, vectorization and non-stride-one memory access issues are highly compiler and machine-architecture specific [23,37], and the issue must be carefully examined on any vector machine of implementation. In this case, it may be possible to force vectorization over the *i* index with the use of compiler directives.

B.6 Synchronization Issues

Reference [35] notes that *coarse-grained* parallelism generally achieves better performance gains than a *fine-grained* parallel approach. Coarse-grained parallelism can be viewed as program-level or at least subroutine-level parallelism as opposed to fine-grained parallelism which achieves concurrency at the loop level. The code as implemented in this project uses a fine-grained parallel approach. Implementation of coarse-grained parallelism on the serial version of the code requires a great deal more code restructuring as well as certain software tools capable of explicit control over the parallel processes. Figures B.4 and B.5 depict the differences between coarse- and fine-grained parallelism. In the coarse-grained approach, the initialization of a process, known as *thread forking*, occurs very infrequently. On the other hand, thread forking occurs much more often in a fine-grained approach. Since each forking process requires approximately 400 clock cycles to execute [34], the fine-grained approach can require significantly greater execution times for CFD codes, which are very heavily loop structured.

While the fine-grained parallel approach does suffer from performance problems, synchronization issues are not as great a concern because each loop is completed by all processors before execution of the next loop begins. This is in contrast to the coarse-grained parallel approach for which it cannot be known in which order the parallel loops will execute. Code fragment 6 illustrates the potential synchronization problems.

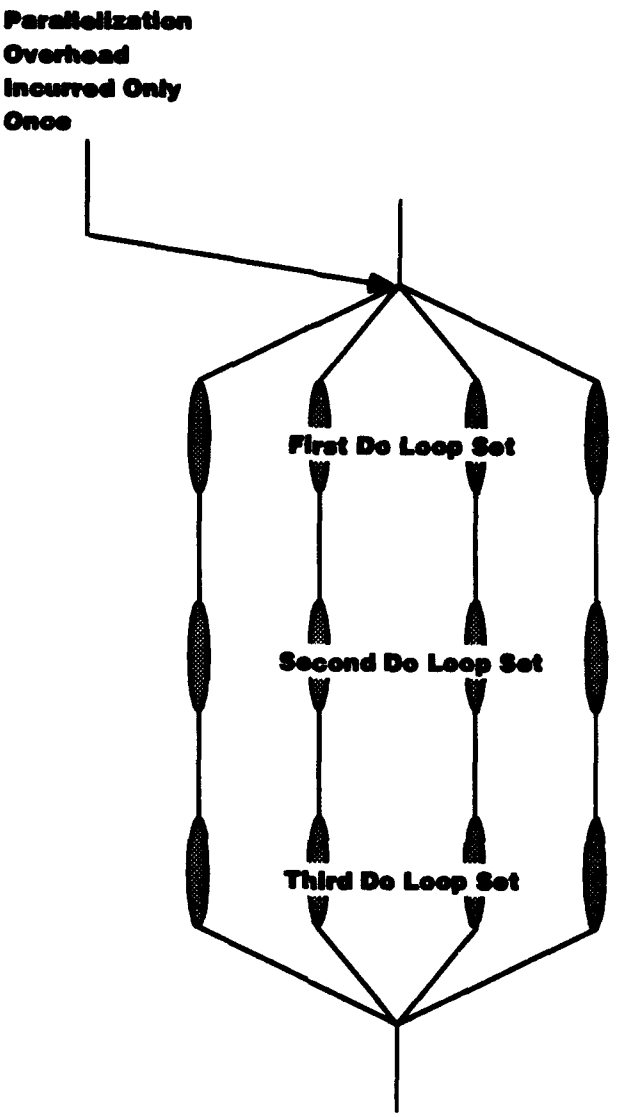


Figure B.4: Coarse-grained parallelism

Code Fragment 6: Synchronization Issues

Serial Code

```
do 2004 k = 1,kmax
do 2004 j = 1,jmax
do 2004 i = 2,imax
   jtild(i,j,k) = (1./3.*((xejp(i,j,k)+xnip(i,j,k)+xzkp(i,j,k))*
    (xpb(i,j,k)-xpb(i-1,j-1,k-1))+(yejp(i,j,k)+ynip(i,j,k)+yzkp(i,j,k))*
    (ypb(i,j,k)-ypb(i-1,j-1,k-1))+(zejp(i,j,k)+znip(i,j,k)+zzkp(i,j,k))*
    (zpb(i,j,k)-zpb(i-1,j-1,k-1))))
```

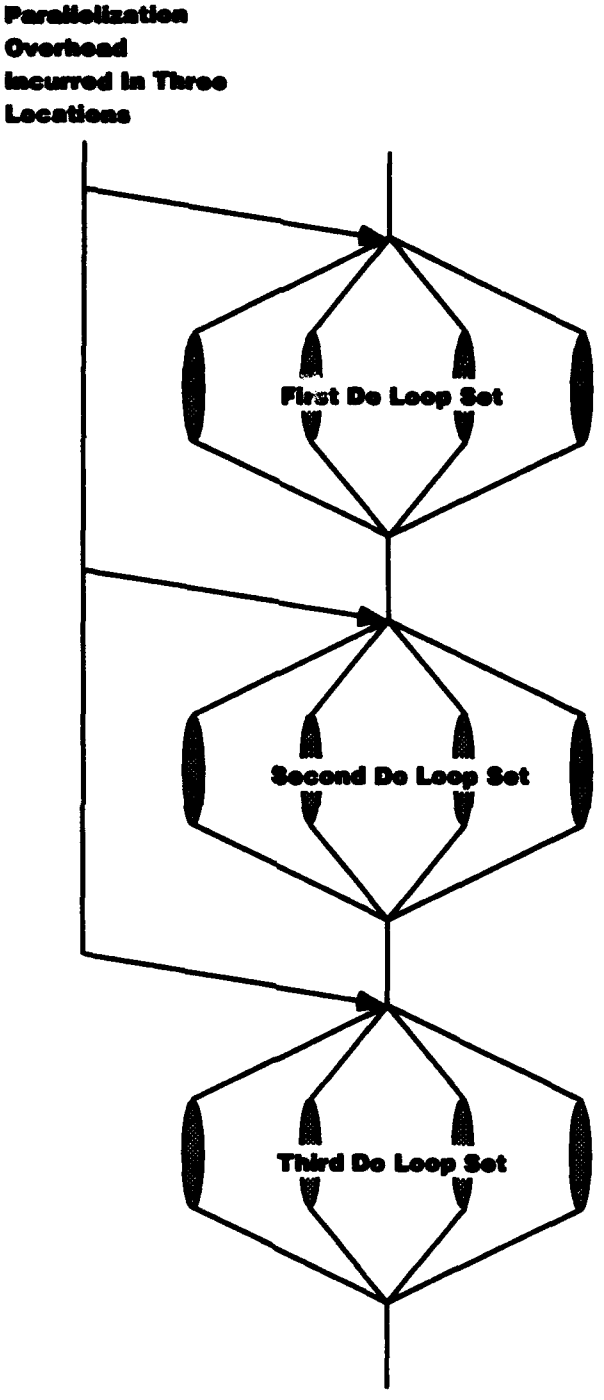


Figure B.5: Fine-grained parallelism

```
2004   continue

/*portion of code omitted here

do 2007 k=1,kmax-1
do 2007 j=1,jmax-1
do 2007 i=1,imax-1

/*portion of loop omitted here

   zkxipi(i,j,k) = sqrt(xnip(i,j,k)**2+ynip(i,j,k)**2 +
                  znip(i,j,k)**2)/(.5*(jtild(i+1,j,k)+jtild(i,j,k)))

/*portion of loop omitted here

2007 continue
```

PANS-3EM Code

```
do 2004 L = 1, numblk
   call indxcp(L,2,imax,jmin,jmax,kmin,kmax,iblkmn,iblkmx,
               jblkmn,jblkmx,kblkmn,kblkmx)
   do 2004 i = iblkmn, iblkmx
      if( i .eq. iblkmn ) then
         L1m = L
         im1 = L
         idir = -1
         call offset( L1m, im1, idir 1 )
      else
         L1m = L
         im1 = i-1
      endif

      do 2004 j = 1, jmax
      do 2004 k = 1, kmax
         jtild(i,j,k,L) = (1./3.*((xejp(i,j,k,L)+xnip(i,j,k,L)+
            xzkp(i,j,k,L))*(xpb(i,j,k,L)-xpb(im1,j-1,k-1,L1m))+
            (yejp(i,j,k,L)+ynip(i,j,k,L)+yzkp(i,j,k,L))*
            (ypb(i,j,k,L)-ypb(im1,j-1,k-1,L1m))+
            (zejp(i,j,k,L)+znip(i,j,k,L)+zzkp(i,j,k,L))*
            (zpb(i,j,k,L)-zpb(im1,j-1,k-1,L1m))))
2004   continue

/*portion of code omitted here

do 2007 L = 1, numblk
   call indxcp(L,1,imax-1,jmin,jmax,kmin,kmax,iblkmn,iblkmx,
               jblkmn,jblkmx,kblkmn,kblkmx)
   do 2007 i = iblkmn, iblkmx
      if( i .eq. iblkmx ) then
         L1p = L
         ip1 = L
         idir = 1
         call offset( L1p, ip1, idir 1 )
      else
         L1p = L
         ip1 = i+1
      endif

   do 2007 j = 1, jmax-1
   do 2007 k = 1, kmax-1

/*portion of loop omitted here
```

```
      zkxipi(i,j,k,L) = sqrt(xnip(i,j,k,L)**2+ynip(i,j,k,L)**2 +
        znip(i,j,k,L)**2)/(.5*(itild(ip1,j,k,L1p1)+jtild(i,j,k,L)))
/*portion of loop omitted here
2007 continue
```

In loop 2007, for $i = iblkmx$, the computation of the variable *zkxipi* uses an across block reference of the variable *jtild* where the value for *jtild* is computed in loop 2004. If code for each block is executing on separate processors, then it cannot be assumed that the required array element of *jtild* has been calculated when the across-block reference is made. Failure to synchronize the code before execution of loop 2007 will result in errors which are nondeterministic in nature and consequently extremely difficult to find.

The situation is resolved by utilizing an operating-system-level function call or compiler directive to ensure that all threads of execution reach a rendezvous point before the potentially incorrect reference is made.

B.7 Future Code Modification Issues

As discussed in Chapter 3, the objective of the development of PANS-3EM was to provide a code readily adaptable for implementation on either a shared- or distributed-memory computing platform. Each of these platforms require a different implementation style.

B.7.1 Shared-Memory Implementation

PANS-3EM has been successfully run on several shared-memory computing platforms. Parallelism can be achieved at the loop level in one of three methods. The first involves the simple use of a command line compiler flag which leaves the parallelization task exclusively to the compiler. Because modern compilers are relatively sophisticated at loop-level optimizations, this method can produce substantial performance improvements with little or no work on the part of the programmer.

The second method of achieving parallelism is to embed compiler directives within the source code. This method has the advantage of potentially greater performance gains than the

first method, but it requires more programming effort.

Finally the third method of achieving parallelism requires the existence of operating system level functions which provide the programmer explicit control over nearly all aspects of program parallelization. Unlike the first two methods, coarse-grained parallelism can be achieved. Therefore, this method provides the greatest potential benefit in terms of performance improvement. In order to utilize this method, PANS-3EM should be restructured to allow for a greater degree of domain decoupling than is possible in the current implementation. This requires the examination of the code to determine all data dependencies and synchronization points. The reader is referred to References [35] and [34] for excellent discussions on this type of shared-memory parallel programming.

B.7.2 Distributed-Memory Implementation

Since a distributed-memory machine generally has no block of common memory to contain variables used by all subdomains, implementation of PANS-3EM on such a platform requires the complete decoupling of the subdomains. This can be accomplished by modifying the use of the fourth index of the data arrays (the L index) to identify a particular processor on which a given subdomain executes. A decoupling of the domains provides the greatest potential performance improvement. Using this approach, it is possible to allow the time-integration to proceed at different rates over each of the subdomains depending on the nature of the flow and structure of the grid. Since Moran [28] has already examined the stability issues associated with local time stepping in the serial code, few stability-related problems should arise in this type of modification.

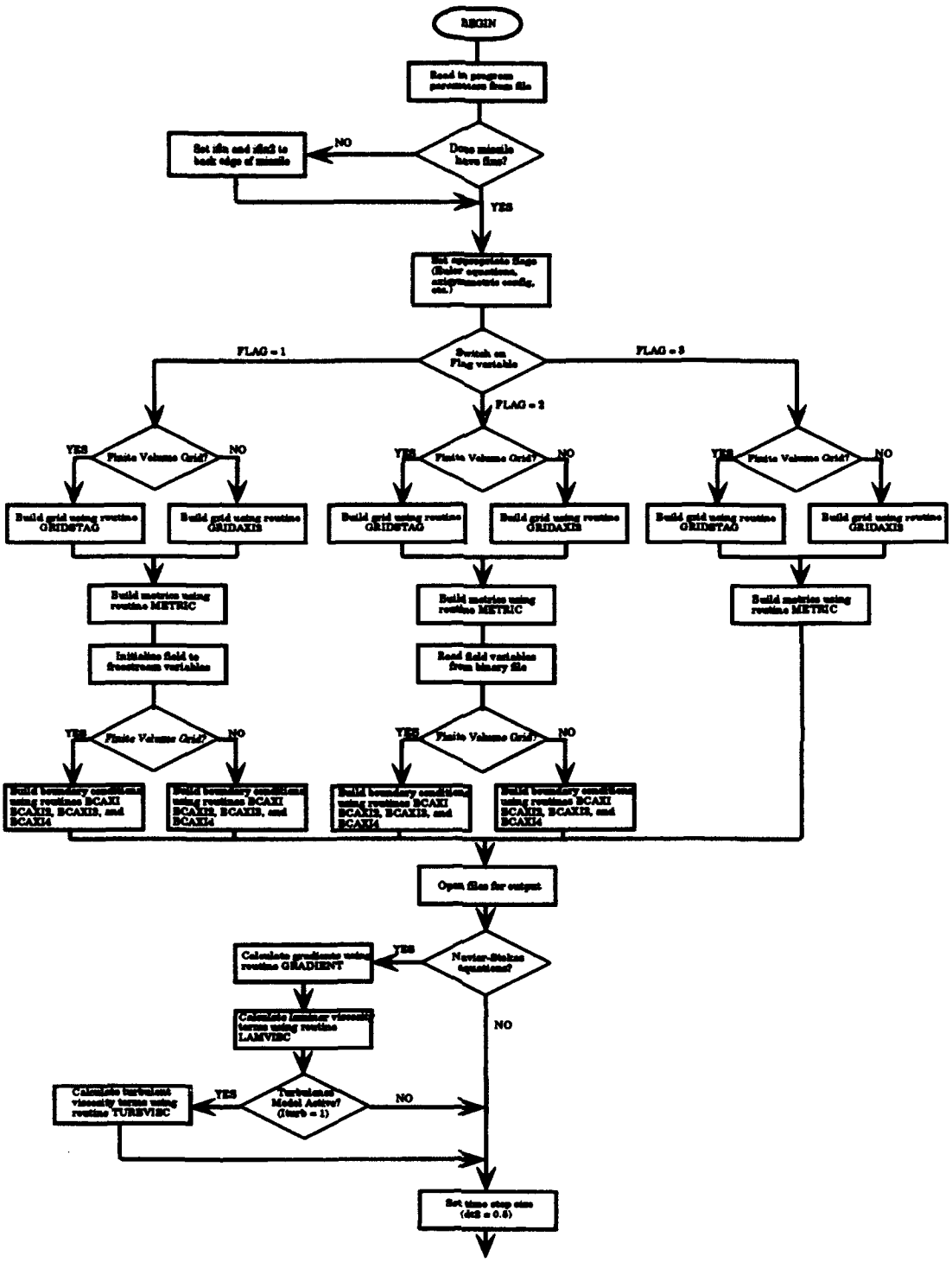
Other issues related to a distributed-memory implementation include the message passing capabilities and memory capacities of a particular machine. As the size of the problem increases or as the computational mesh is refined, memory limitations will affect the number of subdomains that must be used. A greater number of subdomains corresponds to a greater amount of information which must be communicated via messages. If this information is not communicated effectively, then parallel efficiency drops dramatically and the scalability of the

computer algorithm is adversely affected.

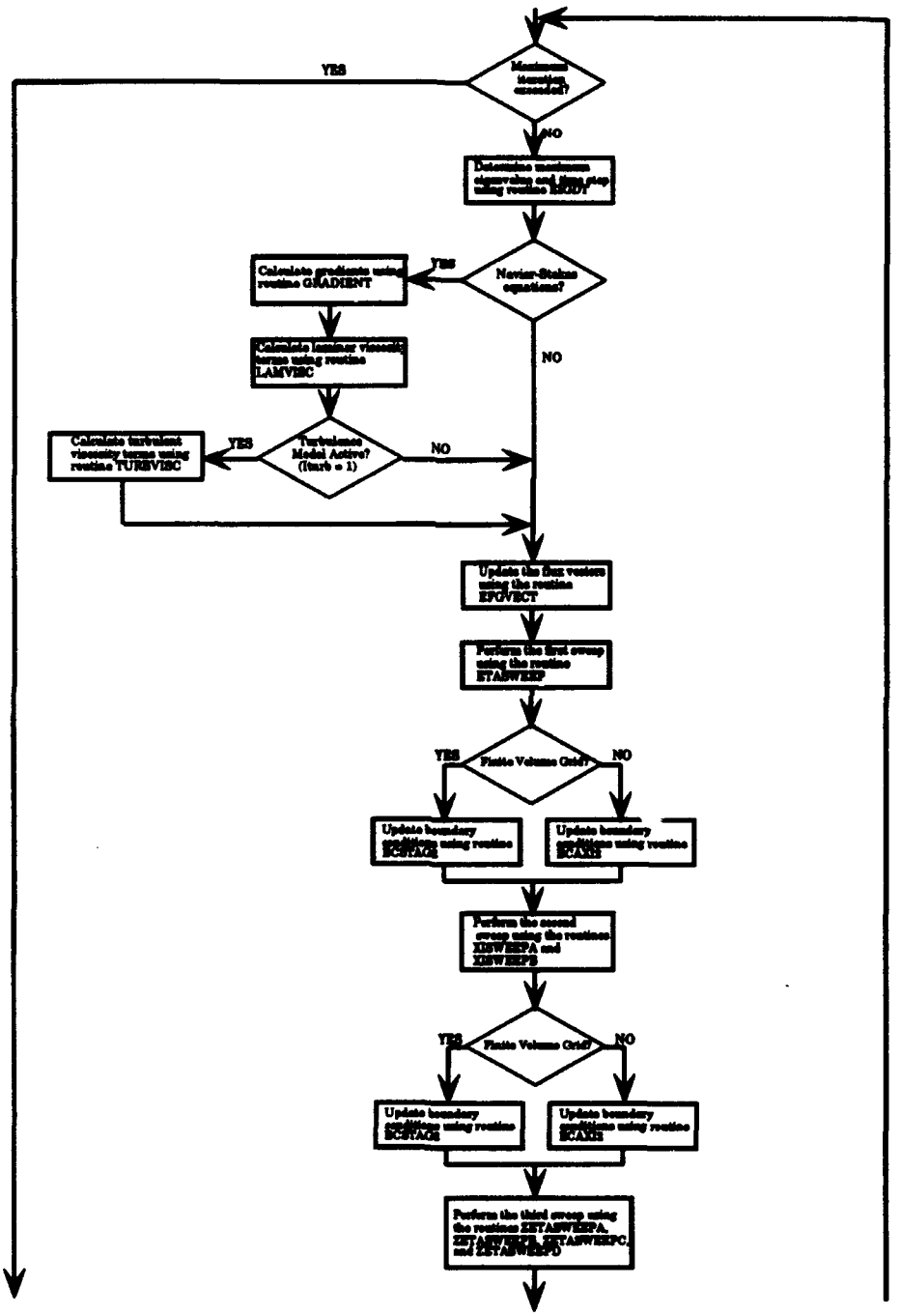
The message passing capabilities of the target machine will also affect the choice of data structures used to hold the messages. For example, it is possible to pass individual array elements as messages for across-block references. However, it may be more feasible to pass an entire array section than an individual array element for across-block references. If so, the buffer-array concept should be carefully examined since the performance gains may outweigh the incurred memory penalty.

Appendix C: Code Modification

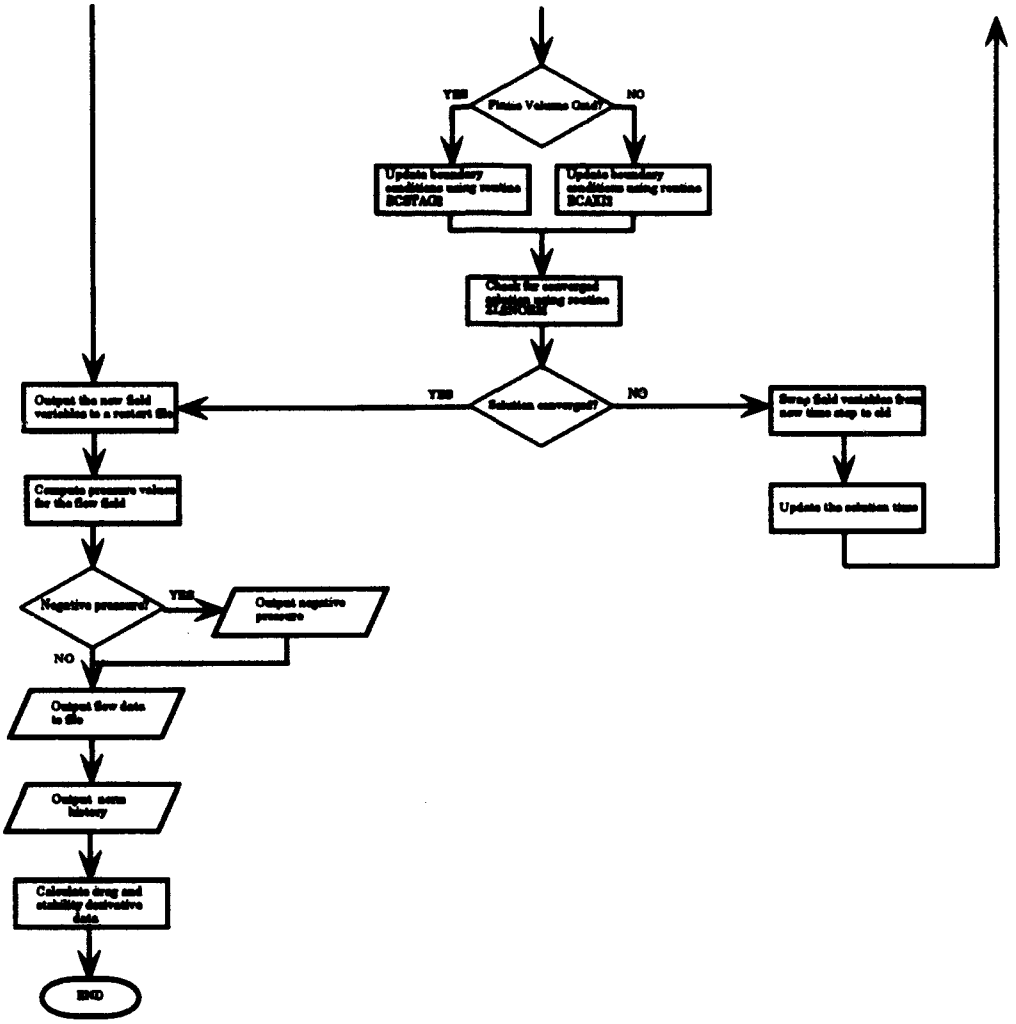
Approximately 21000 lines of code comprise the serial version of the computer program. The extra control and looping structures required for the subdomain implementation resulted in 7000 additional lines of code. In addition to loop modifications, certain structural aspects of the serial code were changed slightly during the modification process. A synopsis of those changes appears in this appendix in the form of high-level program flow charts. The code flow diagrams for the serial and parallel code versions appear in Figures B.1 and B.2 respectively.



C.1 Serial code flow diagram (1 of 3)



C.1 Serial code flow diagram (2 of 3)



C.1 Serial code flow diagram (3 of 3)

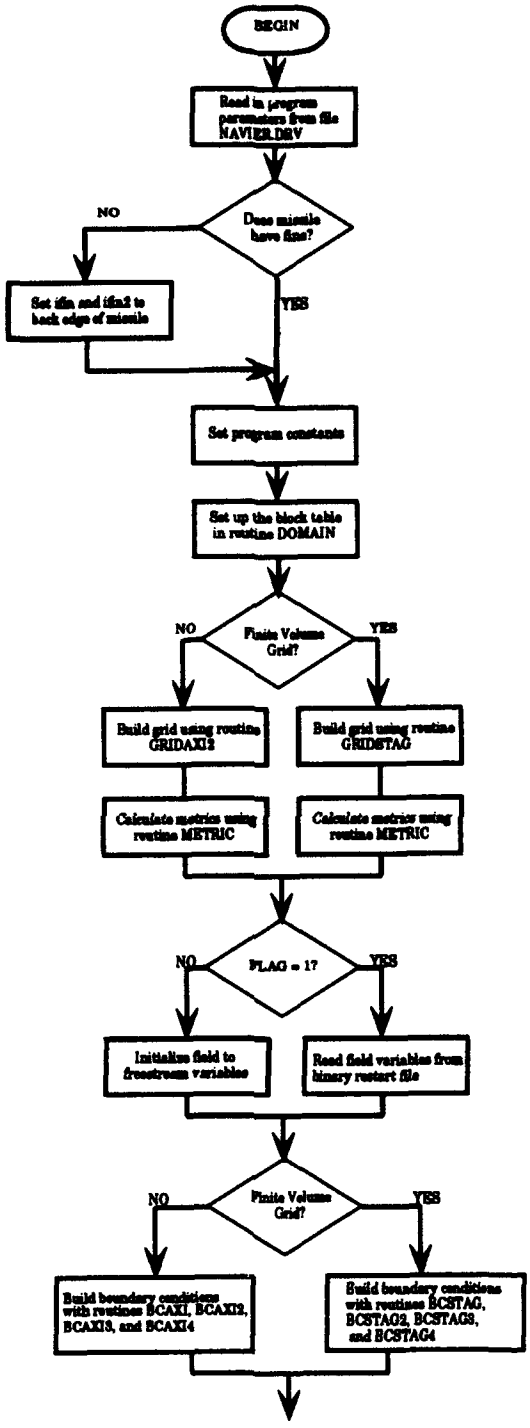


Figure C.2 PANS-3EM flow diagram (1 of 3)

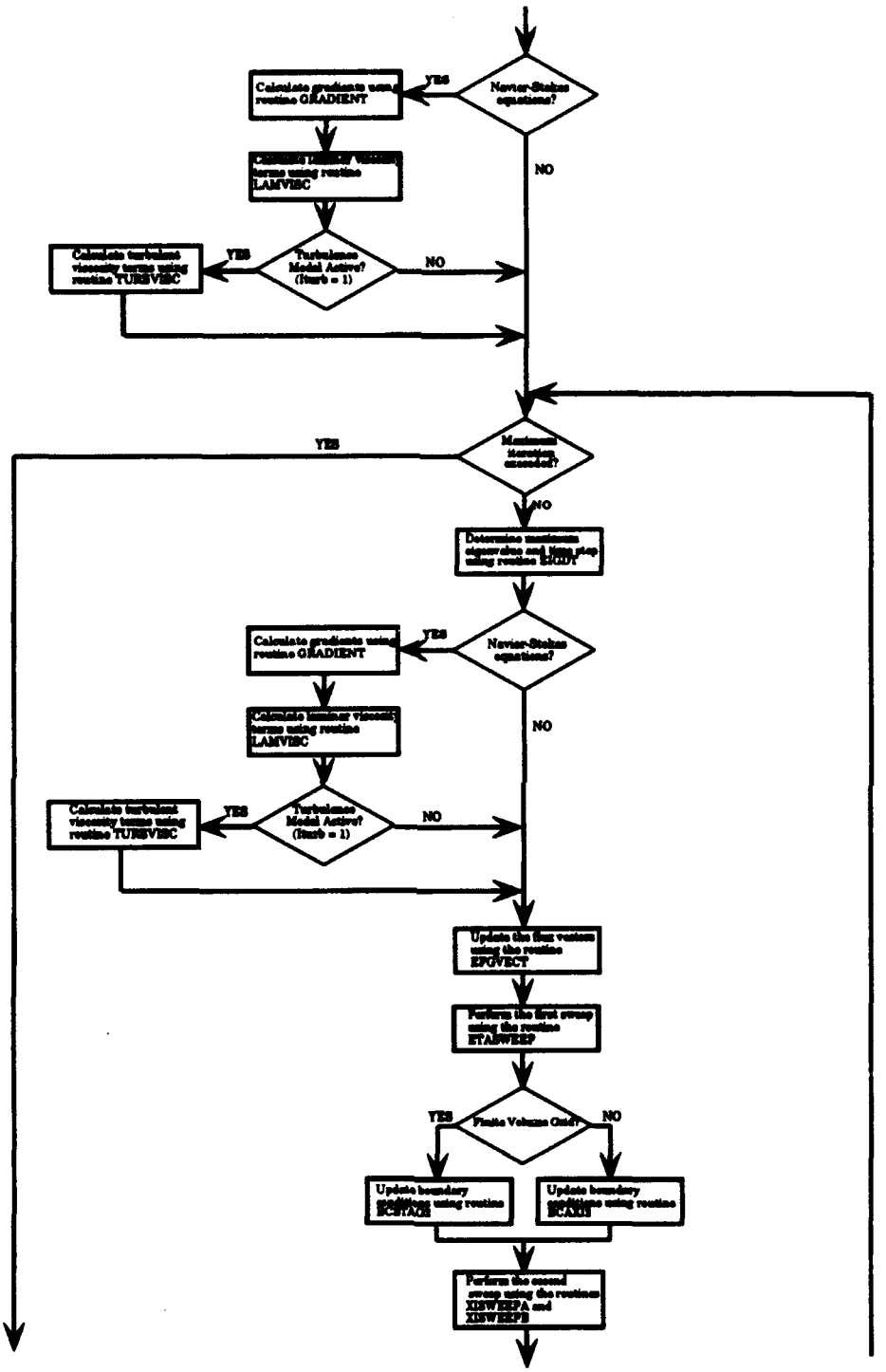


Figure C.2 PANS-3EM flow diagram (2 of 3)

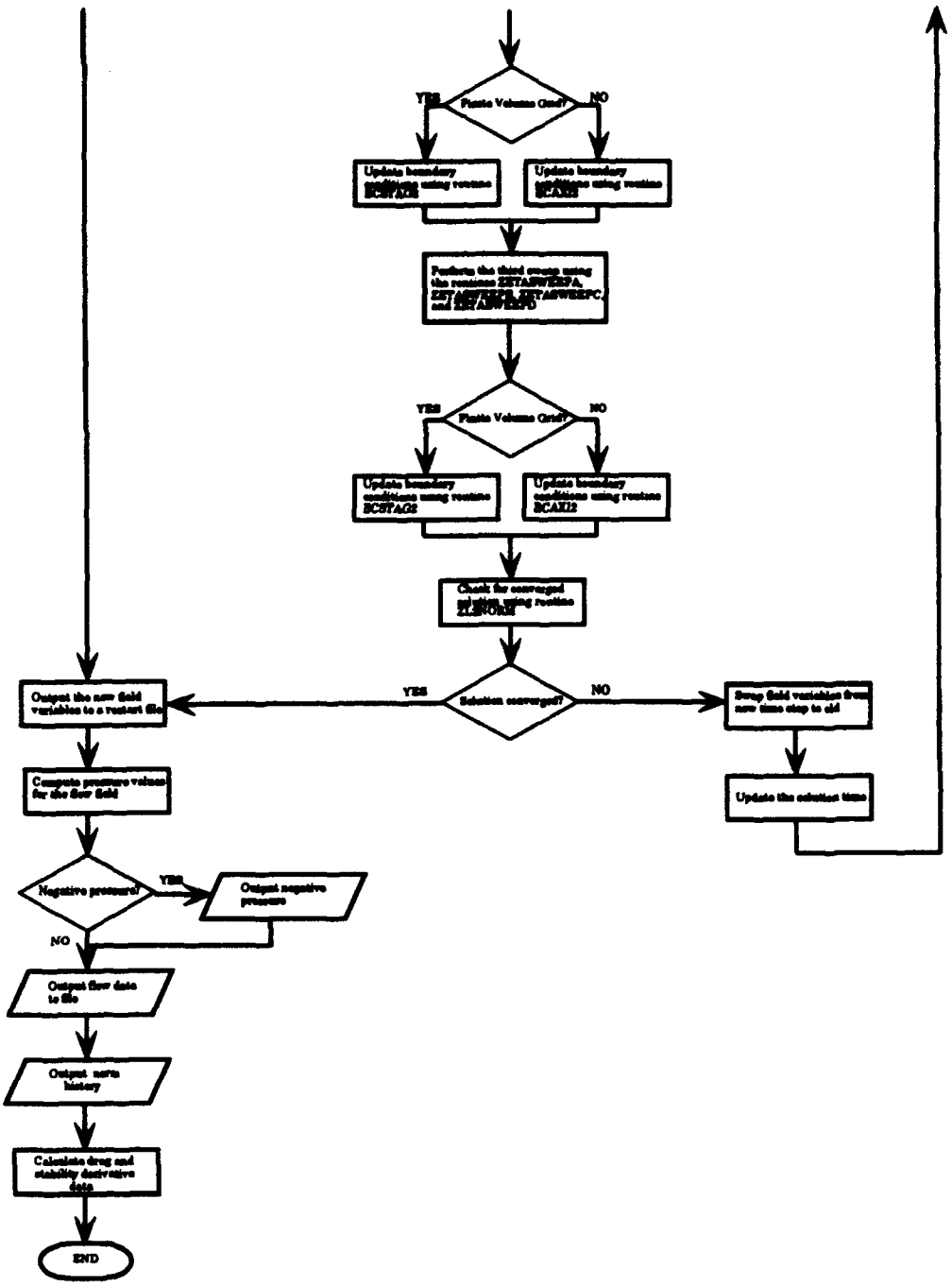


Figure C.2 PANS-3EM flow diagram (3 of 3)

Appendix D: Code Listings

This appendix contains listings of the routines which are crucial to the domain-decomposition approach taken in this investigation. Listings include

- the code listing for the domain decomposition subroutine,
- the code listing for the loop index computation subroutine,
- the code listing for the offset calculation subroutine,
- the code listing for the computational-to-block coordinate transformation subroutine.

Several features of the subroutines were not fully utilized in the current implementation of PANS-3EM but are provided for future expandability. A brief discussion of each follows.

D.1 Domain Decomposition Subroutine (subroutine domain)

This subroutine divides the computational domain into the subdomains and builds the block table described in Chapter 3 and Appendix B. The current implementation divides the computational domain as evenly as possible over the i coordinate direction. Given m nodes and n subdomains in the i-coordinate direction, the first $n-1$ subdomains will contain k points where

$$k = \left\lceil \frac{m}{n} \right\rceil \qquad (D.1)$$

and the last subdomain will contain $m - (n-1)k$ points in the i direction. With the lack of dynamic memory allocation capabilities in the current language of implementation, this approach can result in a maximum of $n-1$ wasted memory locations since all subdomains must be dimensioned the same. This problem can be easily solved if the language of implementation is changed, or the code is modified for use on a distributed-memory system.

Code Listing for Subroutine Domain

```
c*********************************************************************
c*  This subroutine sets up the subdomain block table which determines
c*  which blocks must exchange data across a face.  Current
c*  implementation restricts the blocks to be contiguous, with no
c*  jogs or shifts of the domain boundaries across the block boundaries.
c*  See the documentation for a description of the allowable block
c*  configurations.
c*
c*  Block table data structure:  The block table is implemented as a
c*     two dimensional array with the first index referencing the
c*     particular subdomain block.  The second index ranges from 1 to 9
c*     with each index representing the following:
c*
c*     1--the neighbor to the block on its IMIN side
c*     2--the neighbor to the block on its IMAX side
c*     3--the neighbor to the block on its JMIN side
c*     4--the neighbor to the block on its JMAX side
c*     5--the neighbor to the block on its KMIN side
c*     6--the neighbor to the block on its KMAX side
c*     7--the number of nodes within the block in the i direction
c*     8--the number of nodes within the block in the j direction
c*     9--the number of nodes within the block in the k direction
c*
c*     Fields 1-6 will contain a block number if the block has a
c*     neighbor on that side.  If the block has a boundary on that
c*     side, the field will contain a zero.
c*
c*     Fields 7-9 are required in case the blocks do not all contain
c*     the same number of nodes in a given coordinate direction.
c*
c*  Last Modification Date:  29 Aug 93
c*
c*  Comments:
c*
c*********************************************************************

      subroutine domain( )

        INCLUDE 'commons'

        integer  minj(numblk), maxj(numblk),
     &                mink(numblk), maxk(numblk)

c        character*1   answer
c        character*15  domfile

         imin = 1-ighost
         jmin = 0
         kmin = 0
c
c  Get input from the user to determine subdomain decomposition
c    preferences
c
        write(*,*) 'Is there a domain decomposition file (y/n)?'
        read(*,*) answer
        if( answer .eq. 'n' ) then
          write(*,*) 'Do you want the program do set up',
     &               ' subdomains (y/n)?'
          read(*,*) answer
          if( answer .eq. 'n' ) then
            write(*,*) 'No explicit parallelization will occur'
            numblk = 1
          else
            write(*,*) 'Input the number of subdomains (max = 12)'
            read(*,*) numblk
            write(*,*) 'Current decomposition technique divides',
     &                 ' subdomains evenly along the i axis'
```

```
            write(*,*) 'Enter the number of grid points in the',
     &                 ' i direction'
            read(*,*) imaximum
            iblknd = idim/numblk
c
c  Determine the maximum and minimum coordinate values that will
c   lie in each block.  These values are the actual coordinate values
c   including ghost points and are used to determine block
c   adjacencies in the next section of the code.
c
            mini(1) = 1-ighost
            maxi(1) = mini(1)+idim-1
            minj(1) = 0
            maxj(1) = jmax
            mink(1) = 0
            maxk(1) = kmax
            do 5 i = 2, numblk-1
              mini(i) = maxi(i-1)+1
              maxi(i) = mini(i)+idim-1
              minj(i) = 0
              maxj(i) = jmax
              mink(i) = 0
              maxk(i) = kmax
5           continue
            mini(numblk) = maxi(numblk-1)+1
            maxi(numblk) = imax
            minj(numblk) = 0
            maxj(numblk) = jmax
            mink(numblk) = 0
            maxk(numblk) = kmax
          endif

        else
          write(*,*) 'Input the name of the domain decomposition file'
          read(*,*) domfile
          open( UNIT=12, FILE=domfile, STATUS='OLD')
          read(12,*) numblk
          do 10 i = 1, numblk
            read(12,*) mini(i),maxi(i),minj(i),maxj(i),mink(i),maxk(i)
10        continue
        endif

        write(*,*) 'The program will divide the computational domain',
     & ' into', numblk, ' subdomains'
c
c  Now build the block table.  First determine the neighbors of each
c    of the blocks.
c
c  **NOTE:**  Current implementation of domain decomposition is trivial.
c    This code is really unnecessary, but will be required when a more
c    general decomposition structure is implemented.
c
        do 100 i = 1, numblk
          if( mini(i) .eq. imin ) then
            iblktb(i,1) = 0
          else
            do 20 j = 1, numblk
              if( maxi(j) .eq. mini(i)-1 .and. maxj(j) .eq. maxj(i) .
     &              .and. maxk(j) .eq. maxk(i) ) then
                iblktb(i,1) = j
                goto 25
              endif
20          continue
25        endif

          if( maxi(i) .eq. imax ) then
            iblktb(i,2) = 0
          else
            do 30 j = 1, numblk
              if( mini(j) .eq. maxi(i)+1 .and. maxj(j) .eq. maxj(i)
```

```
     &              .and. maxk(j) .eq. maxk(i) ) then
                  iblktb(i,2) = j
                  goto 35
                endif
30            continue
35          endif

            if( minj(i) .eq. jmin ) then
              iblktb(i,3) = 0
            else
              do 40 j = 1, numblk
                if( maxj(j) .eq. minj(i)-1 .and. maxi(i) .eq. maxi(j)
     &              .and. maxk(j) .eq. maxk(i) ) then
                  iblktb(i,3) = j
                  goto 45
                endif
40            continue
45          endif

            if( maxj(i) .eq. jmax ) then
              iblktb(i,4) = 0
            else
              do 50 j = 1, numblk
                if( minj(j) .eq. maxj(i)+1 .and. maxi(i) .eq. maxi(j)
     &              .and. maxk(j) .eq. maxk(i) ) then
                  iblktb(i,4) = j
                  goto 55
                endif
50            continue
55          endif

            if( mink(i) .eq. kmin ) then
              iblktb(i,5) = 0
            else
              do 60 j = 1, numblk
                if( maxk(j) .eq. mink(i)-1 .and. maxi(i) .eq. maxi(j)
     &              .and. maxj(j) .eq. maxj(i) ) then
                  iblktb(i,5) = j
                  goto 65
                endif
60            continue
65          endif

            if( maxk(i) .eq. kmax ) then
              iblktb(i,6) = 0
            else
              do 70 j = 1, numblk
                if( mink(j) .eq. maxk(i)+1 .and. maxi(i) .eq. maxi(j)
     &              .and. maxj(j) .eq. maxj(i) ) then
                  iblktb(i,6) = j
                  goto 75
                endif
70            continue
75          endif

100       continue
c
c  Now, load the maximum i index for each block. Note that this
c    assumes that the minimum i index for each block is 1 and
c    that the minimum j and k indices are 0.  This apparent
c    discrepancy is to minimize conversion pain at this time.
c
          do 110 i = 1, numblk
            iblktb(i,7) = maxi(i)-mini(i)+1
            iblktb(i,8) = maxj(i)
            iblktb(i,9) = maxk(i)
110       continue

        RETURN
        END
```

D.2 Loop Index Computation Subroutine (subroutine indxcp)

This subroutine uses the block table constructed in subroutine *domain* to determine the proper loop values for the *i* coordinate direction in block coordinates given the loop indices in computational coordinates. While no decomposition was performed in either the *j* or *k* coordinate directions, this subroutine can easily be modified to allow for such. The subroutine parameters necessary to provide for this modification are already incorporated into the subroutine argument list.

Code Listing for Subroutine Domain

```
c*********************************************************************
c*
c*  This subroutine calculates the proper indexes for all program
c*    loops given the minimum and maximum coordinates in computational
c*    coordinates.  Values are returned in the ()blkmn and ()blkmx
c*    subroutine parameters.
c*
c*********************************************************************

      subroutine indxcp(iblknm,ilft,irt,jlft,jrt,klft,krt,
     &    iblkmn,iblkmx,jblkmn,jblkmx,kblkmn,kblkmx)

        integer    iblknm, ilft, irt, jlft, jrt, klft, krt,
     &             iblkmn, iblkmx, jblkmn, jblkmx, kblkmn, kblkmx

        INCLUDE 'commons'

c
c  There are six cases to test for for each coordinate.  They are:
c
c      CASE 1:  The min and max coordinates are both less than the
c               minimum coordinate for that block.  Then that block
c               does not participate in the loop.
c
c      CASE 2:  The min and max coordinates are both greater than the
c               maximum coordinate for that block.  Then that block
c               does not participate in the loop.
c
c      CASE 3:  The min and max coordinates are both within the block.
c               Both coordinates are computed.
c
c      CASE 4:  The min coordinate is within the block and the max
c               coordinate is outside the block.  The min coordinate
c               is computed and the max coordinate is set to the max
c               block coordinate.
c
c      CASE 5:  The min coordinate is less than the min coordinate of
c               the block and the max coordinate is greater than the
c               max coordinate of the block.  Then the full block
c               participates in the loop.
c
c      CASE 6:  The min coordinate is less than the min coordinate of
c               the block and the max coordinate is within the block.
c               Then the min coordinate is set to the min block
c               coordinate and the max coordinate is calculated.
c
```

```
c  NOTE THAT ALL THESE CASES ASSUME A FORWARD SWEEP WITH THE MAX
c    COORDINATE OF GREATER VALUE THAN THE MINIMUM COORDINATE
c

c
c  Case 1
c
        if( irt .lt. mini(iblknm) ) then
          iblkmn = 0
          iblkmx = -1
          goto 1000
        endif

c
c  Case 2
c
        if( ilft .gt. maxi(iblknm) ) then
          iblkmn = 0
          iblkmx = -1
          goto 1000
        endif

c
c  Case 3
c
        if( ilft .ge. mini(iblknm) .and. irt .le. maxi(iblknm) ) then
          iblkmn = ilft-mini(iblknm)+1
          iblkmx = irt-mini(iblknm)+1
          goto 1000
        endif

c
c  Case 4
c
        if( ilft .ge. mini(iblknm) .and. irt .gt. maxi(iblknm) ) then
          iblkmn = ilft-mini(iblknm)+1
          iblkmx = iblktb(iblknm,7)
          goto1000
        endif
c
c  Case 5
c
        if( ilft .lt. mini(iblknm) .and. irt .gt. maxi(iblknm) ) then
          iblkmn = 1
          iblkmx = iblktb(iblknm,7)
          goto 1000
        endif

c
c  Case 6
c
        if( ilft .lt. mini(iblknm) .and. irt .le. maxi(iblknm) ) then
          iblkmn = 1
          iblkmx = irt-mini(iblknm)+1
          goto 1000
        endif

1000    continue
      RETURN
      END
```

D.3 Offset Calculation Subroutine (subroutine offset)

This subroutine is used to calculate a new block coordinate given from a block coordinate and an offset value. A discussion of the requirement for this subroutine appears in Section B.2.1. The subroutine uses the coordinate transformation relations given by equations 39-40. Should the method of domain decomposition change, then those equations and this subroutine must necessarily be modified.

Code Listing for Subroutine Offset

```
c*********************************************************************
c*
c*  This subroutine calculates a new block coordinate given a block
c*    coordinate, (L,i), and an offset direction and number of offset
c*    points.
c*
c*********************************************************************

      SUBROUTINE offset( iblknm, i, idir, inumpt )

        integer   iblknm, i, idir, inumpt

        INCLUDE 'params'

c
c  Convert the block and coordinate to computational coordinates.
c
        iglob = (iblknm-1)*idim + i - ighost
c
c  Calculate the new global coordinate.
c
        iglob = iglob + idir*inumpt
c
c  Convert the new coordinate back into block coordinates.
c
        iblknm = INT((iglob+ighost-.001)/idim)+1
        i = iglob+ighost - (iblknm-1)*idim

      RETURN
      END
```

D.4 Coordinate Transformation Calculation Subroutine (subroutine gl2blk)

This subroutine performs the transformation calculation for converting computational coordinates into block coordinates. Like subroutine *offset*, it uses the transformation equations.

Code Listing for Subroutine glb2blk

```
c**********************************************************************
c*
c*  This subroutine converts a computational i coordinate into the
c*    corresponding block coordinate.
c*
c**********************************************************************

      SUBROUTINE glb2blk( iblknm, i )

        integer  iblknm, i

        INCLUDE 'params'

        temp = i+ighost-.001

        iblknm = INT((i+ighost-.001)/idim)+1
        i = i+ighost - (iblknm-1)*idim

      RETURN
      END
```

Bibliography

1. Alef, M., "Concepts for Efficient Multigrid Implementation on SUPRENUM-like Architectures," *Parallel Computing 17*, pp. 1-16, 1991.

2. Anderson, D. A., Tannehill, J. C., and Pletcher, R. H., *Computational Fluid Mechanics and Heat Transfer*, McGraw-Hill, 1984.

3. Anderson, J. D., *Fundamentals of Aerodynamics*, McGraw-Hill, 1984.

4. Beran, P. S., "AERO 753 Class Notes," Air Force Institute of Technology, 1993.

5. Braaten, M. E., "Solution of Viscous Fluid Flows on a Distributed Memory Concurrent Computer," *International Journal for Numerical Methods in Fluids*, Vol. 10, pp. 889-905, 1990.

6. Cebici, T., Stewartson, K., and Whitelaw, J.H., "Calculation of Two-Dimensional Flow Past Airfoils," *Numerical and Physical Aspects of Aerodynamic Flows II*, Springer-Verlag, 1983.

7. *CMMD Reference Manual*, Thinking Machines Computers, 1993.

8. *Convex FORTRAN User's Guide*, Convex Computer Corporation, 1988.

9. Cray Computer Corp., *1992 Annual Report*, Feb 1993.

10. Dubois, F. and Mehlman, G., "Nonparameterized 'Entropy Fix' for Roe's Method," AIAA Journal, Vol. 31, No. 1, Jan 1993.

11. Furukawa, Masato, et al, "A Zonal Approach for Solving the Compressible Navier-Stokes Equations Using a TVD Finite Volume Method," *JSME International Journal*, Vol. 33, No. 4, 1990.

12. Gabeaud, A., "Base Pressures at Supersonic Velocities," Reader's Forum, Journal of the Aeronautical Sciences, Vol. 17, No. 8, pp. 525-6, August 1950.

13. Gates, R. S., et al, Aerodynamic Test and Analysis of a Slender Generic Missile Configuration," AIAA Paper 89-3368, August 1989.

14. Gustafson, J.L., Montry, G.R., and Benner, R.E., "Development of Parallel Methods for a 1024-Processor Hypercube", *SIAM Journal on Scientific and Statistical Computing*, Vol. 9, No. 4, July 1988.

15. Hammond, Steven W. and Barth, Timothy J., "Efficient Massively Parallel Euler Solver for Two-Dimensional Unstructured Grids," *AIAA Journal*, Vol. 30, No. 4, April 1992.

16. Harten, A., "High Resolution Schemes for Hyperbolic Conservation Laws," *Journal of Computational Physics*, Volume 49: 357-393 (1983).

17. Hauser, J. and Williams, R., "Strategies for Parallelizing a Navier-Stokes Code on the Intel Touchstone Machines," *International Journal for Numerical Methods in Fluids*, Vol. 15, pp.51-58, 1992.

18. Hord, R. M., *Parallel Computing in SIMD Architectures*, CRC Press, Inc., 1990.

19. Horton, G. and Knirsch R., "A Time-Parallel Multigrid-Extrapolation Method for Parabolic Partial Differential Equations," *Parallel Computing 18*, pp. 21-29, 1992.

20. Intel Corporation, *Paragon Supercomputers*, 1992.

21. iPSC®/2 and iPSC®/860 Fortran Routines, *Programmer's Reference Manual*, Intel Supercomputer Systems Division, 1991.

22. Karin, S.and Smith N. P., *The Supercomputer Era*, Harcourt Brace Jovahovich, 1987.

23. Lakshmanan, B. and Tiwari, S. N., "Study of Supersonic Intersection Flowfield at Modified Wing-Body Junctions," *AIAA Journal*, May 1993.

24. Levine, D. et al., "A Comparative Study of Automatic Vectorizing Compilers," *Parallel Computing*, Dec. 1991.

25. Lewis T. G. and El-Rewini, H., *Introduction to Parallel Computing*, Prentice-Hall, Inc., 1992.

26. Matrone, A., et al., "LINDA and PVM: A comparison between two environments for parallel programming," *Parallel Computing* 19, pp. 943-957, 1993.

27. Moran, K., *Static and Dynamic Stability Analysis of a Slender Generic Missile, Using Computational Fluid Dynamics*, Ph.D. Prospectus, Air Force Institute of Technology, July 1992.

28. Moran, K. and Beran, P. S., "Navier-Stokes Simulations of Slender Axisymmetric Shapes in Supersonic, Turbulent Flow," unpublished, 1993.

29. Moran, K., personal conversations, Apr-Oct, 1993.

30. Murman, E.M., et al, "Performance of Several CFD Loops on Parallel Processors", Report CFDL-TR-88-3, Massachusetts Institute of Technology, March 1988.

31. Roache, P. J., *Computational Fluid Dynamics*, Hermosa, 1982.

32. Roe, P. L., "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," *Journal of Computational Physics*, Vol. 43, pp. 357-72, 1981.

33. Shang, J.S. and Scherr, S.J., "Navier-Stokes Solution of the Flow Field Around a Complete Aircraft," AIAA Paper 85-1509, July 1985.

34. Shang, J.S., personal correspondence, September 1993.

35. Silverio, C. J. and Lohnes, C., *IRIS-4D Series Fortran 77 Programmer's Guide, Version 2.0*, Silicon Graphics Computer Systems, 1990.

36. Sloane, G. and Walsh, K., *Parallel Programming on Silicon Graphics Computer Systems*, Technical Documentation, Silicon Graphics, Inc, 1991.

37. Smith, N. P., ed., "Where Weak Flesh Meets the Uncompromising Machine," *Supercomputing Review*, Vol. I, 1988.

38. Smith, N. P., ed., "Contrasting Architectures of Minisupercomputers: Form Shapes Function," *Supercomputing Review*, Vol. I, 1988.

39. Steinbrenner, J. P., et al, "The GRIDGEN 3D Multiple Block Grid Generation System, Volume 1: Final Report," WRDC -TR-90-3022. Flight Dynamics Laboratory, Wright Research and Development Center, Wright-Patterson AFB, OH, July 1990.

40. Sun, Xian-He and Gustafson, John L., "Toward a Better Parallel Performance Metric," *Parallel Computing 17*, pp. 1093-1109, 1991.

41. Swisshelm, J. M., "Development of a Navier-Stokes Algorithm for Parallel-Processing Supercomputers," NASA TM-102188, May 1989.

42. Thompson, C. P., et al, "On the Parallelization of an Adaptive Multigrid Algorithm for a Class of Flow Problems," *Parallel Computing 18*, pp. 449-466, 1992.

43. *Titan Programmer's Guide*, Kubota Pacific Computer Inc., 1992.

44. Vinokur, Marcel, "An Analysis of Finite-Difference and Finite-Volume Formulations of Conservation Laws,"

45. Vitale, H. E., et al, "Aerodynamic Test and Ongoing Analysis of a Slender Generic Missile Configuration," AIAA Paper 91-2895.

46. Wang, J. C. T. and Widhopf, G. F., "A High-Resolution TVD Finite Volume Scheme for the Euler Equations in Conservation Form." *Journal of Computational Physics* 84, pp. 145-173, 1989.

47. Yadlin, Y. and Caughey, D. A., "Parallel Computing Strategies for Block Multigrid Implicit Solution of the Euler Equations," *AIAA Journal*, Vol. 30, No. 8, August 1992.

48. Yee, H. C., "A Class of High-Resolution Explicit and Implicit Shock-Capturing Methods," NASA TM-101088, Feb 1989.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE
A NUMERICAL STUDY OF HIGH-SPEED MISSILE CONFIGURATIONS USING A BLOCK-STRUCTURED PARALLEL ALGORITHM

5. FUNDING NUMBERS

6. AUTHOR(S)
Douglas C. Blake, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology
WPAFB, OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GAE/ENY/93D-4

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
Major Bill West
ASC/NA
WPAFB, OH 45433

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT *(Maximum 200 words)*

A numerical analysis of the aerodynamic phenomena associated with the high-speed flight of a sharp-nosed, four-finned, high-fineness ratio missile using a block-structured, parallel computer algorithm is presented. The algorithm, PANS-3EM, utilizes a second-order-accurate, shock-capturing, Total Variation Diminishing scheme and incorporates a Baldwin-Lomax turbulence model. PANS-3EM allows for extreme flexibility in the choice of computational domain decomposition and computing machine of implementation. Developmental work consists of conceptualization and verification of the algorithm as well as parallel performance and scalability studies conducted on a variety of computing platforms.

Using PANS-3EM, the aerodynamic characteristics of the missile are investigated. Drag and pitching moment coefficients are computed and compared against experimental flight data. Trends in the numerical data agree with experimental results with the exception that an unexpected reversal of the stability characteristics exhibited by the missile at speeds in excess of Mach 3.75 are not confirmed by the computer code.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Computational Fluid Dynamics; Parallel Processing (Computers); Navier-Stokes Equation; Decomposition; TVD Schemes; Supersonic Flow	114
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL