THE DEVELOPMENT OF A SCALABLE PARALLEL 3–D CFD ALGORITHM FOR TURBOMACHINERY

By

Edward Allen Luke

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computational Engineering
in the College of Engineering

Mississippi State, Mississippi
July 1993

Name: Edward Allen Luke

Date of Degree: July 30, 1993

Institution: Mississippi State University

Major Field: Computational Engineering

Major Professor: Dr. Donna Reese

Title of Study: THE DEVELOPMENT OF A SCALABLE PARALLEL 3–D CFD ALGORITHM FOR TURBOMACHINERY

Pages in Study: 75

Candidate for Degree of Master of Science

Two algorithms capable of computing a transonic 3–D inviscid flow field about rotating machines are considered for parallel implementation. During the study of these algorithms, a significant new method of measuring the performance of parallel algorithms is developed. The theory that supports this new method creates an empirical definition of scalable parallel algorithms that is used to produce quantifiable evidence that a scalable parallel application has been developed. The implementation of the parallel application and an automated domain decomposition tool is also discussed.

ACKNOWLEDGMENTS

First, I would like to give a big hoot and a holler to Dr. Tim Swafford for his invaluable editing suggestions. And lets all give a big round of applause to Dr. Donna Reese for her helpful suggestions and for listening to my regular whining sessions during the course of this research work (Bravo! The crowd goes wild!). I want to ring the cow bell for Dr. Mark Janus for putting up with my constant pestering and for his assistance in explaining various features of his turbomachinery application. I would also like to thank Tony Skjellum for his tough questions that got me scratching my head for a while.

In addition, I would like to express my thanks to NASA Lewis Research Center for providing financial support and access to the Intel Delta at Caltech, and to Rich Blech and Russ Claus, technical monitors.

Most of all, I want to credit my parents for nurturing my creative and inquisitive spirit.

E. A. Luke
Mississippi State University
June 1993

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I
INTRODUCTION

Using multiple computational engines concurrently in order to increase the performance of computing platforms is a strategy that has been in existence almost since the advent of computer systems. Computer architecture has felt the impact of this strategy in the form of pipelined and vector architectures, interleaved memory systems, floating point accelerators, input and output processors, and so on. Many of these "parallel" computers derived their design principles from observations of existing applications. When a large class of applications exhibit characteristics that could provide significant performance enhancement by concurrent utilization of computational resources, computer architectures were devised to take advantage of these characteristics. Applications that did not exhibit these characteristics adapted to the new computer architectures by modification of the implementation of algorithms and by the development of new algorithms. Automation of some of the processes involved in adapting an application to new computer architectures helped to speed the acceptance of these platforms by a wider audience of application developers.

The performance gains that pipelined and vector architectures can provide has begun to stabilize as the these technologies mature. In addition, performance gains resulting from new technologies that create faster electronic devices soon will be reaching physical barriers relating to device sizes and the maximum theoretical speed that signals can propagate.

General parallel computer architectures such as multicomputer ensembles have been suggested as an approach to gain higher performance computational platforms in light of the future limitations traditional computer architectures will encounter. Since general parallel computer architectures are designed based on the assumption that traditional architectures will soon reach theoretical limits rather than exploiting characteristics of existing applications, it is not surprising to discover that general parallel computer architectures place a greater burden on application developers than traditional computer architectures. Key to the success of general parallel computer architectures is the development of applications that can use the resources these computer architectures provide.

This thesis advances the science of general parallel computing in two important ways, 1) it introduces a new method for measuring and classifying parallel algorithms, and 2) it contributes to the base of applications that can utilize the resources general parallel architectures provide.

The method for measuring parallel algorithms that was developed in this thesis arose from the study of two parallel algorithms that are presented in Chapter II. One of these algorithms was developed to utilize vector architectures and the other was originally developed for purposes other than parallel processing. In this thesis, the algorithm that was developed for vector architectures is referred to as the parallel approximate factorization algorithm and the other as the modified algorithm. Intuition and experiences suggested that the modified algorithm would be superior in the context of massively parallel architectures but developing a solid argument for this case proved difficult. Attempts to use common metrics of parallel algorithm performance to make this case provided ambiguous results. The experience gained from this analysis led to a new perspective on parallel algorithm evaluation. This new perspective, based on the cost effectiveness of parallel computations, resulted in a new method for measuring parallel algorithms that provided unambiguous results. These findings are presented in Chapter III.

Chapter IV presents a summary of the details involved in the implementation of the modified algorithm developed in Chapter II and analyzed in Chapter III.

Chapter V presents the results of numerical experiments needed to complete the arguments presented in Chapter III, and finally Chapter VI discusses the implications of this research and suggests areas where future investigation is needed.

CHAPTER II
AN OVERVIEW OF THE NUMERICAL ALGORITHM

The first step in the process of developing an application for parallel processors is the identification of the components of an application's structure which impede or aid in accomplishing the goal of solving the problem on a number of processors concurrently. For the specific task of parallelizing an existing application, analysis of the application falls roughly into two categories: 1) the analysis of the general structural constraints presented by the computational model; and 2) the analysis of the specific constraints presented by the implementation of that computational model.

Before analysis of the computational model can begin, the computational model must be presented. This chapter will present aspects the computational model developed by Janus[1] that are relevant to the parallel implementation. The reader interested in the complete derivation of the computational model is referred to Janus's dissertation[1].

The Conservative Differential Form of the Euler Equations

The application of interest involves seeking a numerical solution to the Euler equations, a subset of the Navier–Stokes equations given the simplifying assumptions of a thermally nonconducting inviscid perfect gas with no body forces. The Euler equations can be written in conservative form where no dependent variables are present outside of the differential operators. In the conservative form, errors produced by discretization operators will not cause a violation of the conservation properties expressed in conservation law equations. The conservative form of the Euler equations written for the Cartesian coordinate system can be expressed as

$$\frac{\partial q}{\partial t} + \frac{\partial f(q)}{\partial x} + \frac{\partial g(q)}{\partial y} + \frac{\partial h(q)}{\partial z} = 0. \tag{2.1}$$

In equation (2.1), q represents the five dependent variables: mass, three components of momentum, and energy, whereas $f(q)$, $g(q)$, and $h(q)$ represent the flux vectors in each of the coordinate directions x, y, and z respectively. It should be pointed out that the actual equations solved in the present application are the Euler equations cast in a curvilinear coordinate system. Although this increases the complexity of the equations somewhat, it does not change the data dependencies that exist between computational cells. Since the curvilinear derivation would only serve to obscure the issues of consequence to the parallel algorithm evaluation, the derivation of the numerical model will be presented in the Cartesian coordinate system.

Discretization

Applying a first–order implicit temporal discretization with a finite volume spatial discretization to equation (2.1) yields

$$\frac{\Delta q^n}{\Delta t} + \frac{\delta_x f(q)^{n+1}}{\Delta x} + \frac{\delta_y g(q)^{n+1}}{\Delta y} + \frac{\delta_z h(q)^{n+1}}{\Delta z} = 0, \tag{2.2}$$

where

$$\Delta q^n = q^{n+1} - q^n,$$

$$\delta_x f(q)^{n+1} = f(q)^{n+1}_{x+\frac{\Delta x}{2},y,z} - f(q)^{n+1}_{x-\frac{\Delta x}{2},y,z},\ \delta_y g(q)^{n+1} = g(q)^{n+1}_{x,y+\frac{\Delta y}{2},z} - g(q)^{n+1}_{x,y-\frac{\Delta y}{2},z},$$

$$\delta_z h(q)^{n+1} = h(q)^{n+1}_{x,y,z+\frac{\Delta z}{2}} - h(q)^{n+1}_{x,y,z-\frac{\Delta z}{2}}.$$

The choice of implicit time integration in the derivation of equation (2.2) is the most significant with respect to parallelization of this algorithm because this choice introduces multiple unknowns (represented by functions evaluated at the $n + 1$ time level) into the discretized equation for a computational cell. Given the discretized equations for every cell, there are N equations and N unknowns that when linearized can be written as a linear system of the form $Ax = b$. A is a matrix that contains the coefficients of the unknowns, x is a vector of the unknowns referred to as the solution vector, and b is a vector containing a collection of known values. As an emphasis to the connection between the discretized equations and the linear system that results, the unknowns are grouped to the left–hand–side (LHS) of the equation and the knowns are grouped to the right–hand–side (RHS).

Solving the linear system that results from implicit time integration is difficult, even on sequential processors. In general, much effort is expended to adjust the form of the LHS so that a kernel of efficient linear system solvers such as tri–diagonal solvers or LU factorization solvers can be employed. Unfortunately, these schemes usually rely on backward substitution steps that create data dependencies forcing sequential or near sequential execution. Given these problems, the form of the LHS and the methods chosen to solve the linear system that results will be a subject of the present investigation.

Higher order temporal discretizations of equation (2.2) can be derived using a parameterized form developed by Beam and Warming [2]. The higher order schemes, which require the storage of Δq and residual terms from the previous iteration, primarily affect the RHS and as such have little impact on the form of the linear system. In fact, because a first order flux Jacobian extrapolation is used on the LHS even in schemes of higher order of accuracy, order of accuracy will not significantly change the results of the parallel algorithm analysis of the first order derivations given here.

Linearization

Equation (2.2) can't be used to construct a linear system of equations because the flux vectors *f(q)*, *g(q)*, and *h(q)* are non–linear functions of *q*. One approach to overcoming this problem is to approximate the flux vectors with functions that are linear combinations of the unknown variable q^{n+1}. This process, known as flux vector linearization, is accomplished by truncating the Taylor series for the function about time level *n* at the second term. This process yields the linearized flux vectors

$$\begin{aligned} f(q)^{n+1} &= f(q)^n + \left(\frac{\partial f(q)}{\partial q}\right)^n (q^{n+1} - q^n), \\ g(q)^{n+1} &= g(q)^n + \left(\frac{\partial g(q)}{\partial q}\right)^n (q^{n+1} - q^n), \\ h(q)^{n+1} &= h(q)^n + \left(\frac{\partial h(q)}{\partial q}\right)^n (q^{n+1} - q^n). \end{aligned} \tag{2.3}$$

The derivative terms in equation (2.3) are called the flux Jacobians. The flux Jacobians are 5 by 5 matrices represented by the naming convention

$$A = \left(\frac{\partial f(q)}{\partial q}\right)^n, B = \left(\frac{\partial g(q)}{\partial q}\right)^n, C = \left(\frac{\partial h(q)}{\partial q}\right)^n.$$

Substituting the linearized flux vectors of (2.3) into the discretized equation of (2.2) yields the equation

$$\frac{\Delta q^n}{\Delta t} + \frac{\delta_x(f(q)^n + A(\Delta q^n))}{\Delta x} + \frac{\delta_y(g(q)^n + B(\Delta q^n))}{\Delta y} + \frac{\delta_z(h(q)^n + C(\Delta q^n))}{\Delta z} = 0\,. \tag{2.4}$$

Collecting coefficients of the unknown terms to the LHS gives the equation

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A \cdot + \frac{\Delta t}{\Delta y}\delta_y B \cdot + \frac{\Delta t}{\Delta z}\delta_z C \cdot\right)\Delta q^n = -\Delta t\left(\frac{\delta_x f(q)^n}{\Delta x} + \frac{\delta_y g(q)^n}{\Delta y} + \frac{\delta_z h(q)^n}{\Delta z}\right). \quad (2.5)$$

Although equation (2.5) represents the general formulation of the discretized equations used in the Euler solver, it is not complete. Before continuing, observe the dots following the flux Jacobians on the LHS. The dots imply that the finite volume difference operator is applied after the distribution of Δq. The key ambiguity that remains in equation (2.5) arises from the fact that the finite volume difference operator is extracting information from cell faces, while the solution vector is in terms of the dependent variables defined to be located at cell centers. This implies that one needs to choose a method to extrapolate information from cell centers (where it is assumed known) to cell faces (where it is assumed unknown). An obvious extrapolation method would be an averaging process of neighboring Δq values. Unfortunately, it is known that this approach leads to schemes that tend to blur (or smear) shocks. On the other hand, an upwind extrapolation scheme derived from the mathematical character of the physics (which dictates that information is propagated in preferred directions) yields a solver that resolves shocks more sharply. However, before this approach can be taken the flux vectors must be reformulated in order that spatial difference operators can be defined to allow information to propagate in these preferred directions.

Flux Vector Splitting

The Euler equation's flux functions are homogeneous of degree one. Steger and Warming[3] exploited this observation to redefine the flux function in terms of the flux Jacobian. For example, the x–direction flux, $f(q)$, can be written as

$$f(q) = Aq. \tag{2.6}$$

Where A is the flux Jacobian matrix computed as before,

$$A = \frac{\partial f(q)}{\partial q}. \tag{2.7}$$

As shown in numerous references (see references [4] and [5]) the flux function can be split based on the signs of the eigenvalues of the flux Jacobian matrix A. Using the Steger–Warming[3] approach, flux Jacobians can be split and applied to equation (2.6) arriving at the equation

$$f(q) = A^{(+)}q + A^{(-)}q = f^{+}(q) + f^{-}(q), \tag{2.8}$$

where the eigenvalues of $A^{(+)}$ are non–negative and those of $A^{(-)}$ are non–positive.

Equation (2.8) can then be linearized in the same way as equation (2.3) and substituted in the discretized of equation (2.2) to arrive at

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A^{+}\cdot + \frac{\Delta t}{\Delta x}\delta_x A^{-}\cdot + \frac{\Delta t}{\Delta y}\delta_y B^{+}\cdot + \frac{\Delta t}{\Delta y}\delta_y B^{-}\cdot + \frac{\Delta t}{\Delta z}\delta_z C^{+}\cdot + \frac{\Delta t}{\Delta z}\delta_z C^{-}\cdot\right)\Delta q^{n} =$$

$$-\Delta t\left(\frac{\delta_x f^{+}(q)^{n}}{\Delta x} + \frac{\delta_x f^{-}(q)^{n}}{\Delta x} + \frac{\delta_y g^{+}(q)^{n}}{\Delta y} + \frac{\delta_y g^{-}(q)^{n}}{\Delta y} + \frac{\delta_z h^{+}(q)^{n}}{\Delta z} + \frac{\delta_z h^{-}(q)^{n}}{\Delta z}\right). \tag{2.9}$$

Note the notational difference in the flux Jacobian terms of (2.8) and (2.9) indicated by

$$A^{(+)} = T\Lambda^{(+)}T^{-1} \neq A^{+} = \frac{\partial f^{+}(q)}{\partial q}, \tag{2.10}$$

where the matrices T, $\Lambda^{(+)}$, and T^{-1} are presented elsewhere[4].

Remember the purpose of splitting the flux functions was to accommodate an upwinding extrapolation scheme based honoring the proper direction of information propagation. The reason why Steger–Warming flux splitting helps accomplish this can be seen by drawing an analogy between the quasi–linear and characteristic variable formulations of the Euler equations. The eigenvalues of the characteristic equation can be seen as wave speeds where the signs of the eigenvalues represent the direction of wave propagation. It can be shown by analogy to the quasi–linear form of the Euler equations that the eigenvalues of the conservative flux Jacobians are identical to the eigenvalues of the characteristic equation, thus the signs of the eigenvalues represent the direction of information propagation. Given this analysis, we can construct a first order extrapolation by which the flux vector q information is "moved" from cell centers to cell faces in the characteristic directions dictated by the signs of the corresponding eigenvalues. For example, using this extrapolation technique, a first–order finite volume discretization operator on the LHS can be written as

$$\delta_x(A^+\Delta q) = (A^+\Delta q)_{x,y,z} - (A^+\Delta q)_{x-\Delta x,y,z}\,, \tag{2.11}$$

$$\delta_x(A^-\Delta q) = (A^-\Delta q)_{x+\Delta x,y,z} - (A^-\Delta q)_{x,y,z}\,. \tag{2.12}$$

A scheme that uses Steger–Warming flux splitting theory has now been derived. For completeness, it should be pointed out that the Euler solver developed by Janus[1] utilizes a more advanced splitting scheme on the RHS known as flux difference splitting. The specific scheme used is the approximate Riemann solver originally developed by Roe[6] and implemented by Janus[1]. Unfortunately, linearizing the fluxes that result from the Roe scheme has proved difficult primarily because the Jacobians associated with Roe fluxes' are nearly impossible to determine analytically. As a compromise to computing the Roe flux Jacobians numerically, the present application uses

Steger–Warming flux vector splitting theory on the LHS, and Roe flux difference splitting theory on the RHS (The rational for this compromise is presented in reference [7]).

Approximate Factorization

Examination of the LHS of equation (2.9) yields the observation that there is a linear combination of seven unknowns: the Δq associated with a given cell plus those associated with its six neighboring cells. Taking advantage of the topologically orthonormal grid structure implied by the six point stencil and assuming explicit treatment of the boundary conditions, an ordering of the equations for the cells can be found such that the resulting A matrix (of the system $Ax=b$) defined by the LHS will be block septidiagonal, and thus is expensive to solve. A less computationally expensive method is derived by applying an approximate factorization to the LHS of equation (2.9) yielding the equation

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A^+ \cdot + \frac{\Delta t}{\Delta y}\delta_y B^+ \cdot + \frac{\Delta t}{\Delta z}\delta_z C^+ \cdot\right)\left(I + \frac{\Delta t}{\Delta x}\delta_x A^- \cdot + \frac{\Delta t}{\Delta y}\delta_y B^- \cdot + \frac{\Delta t}{\Delta z}\delta_z C^- \cdot\right)\Delta q^n =$$

$$-\Delta t\left(\frac{\delta_x f^+(q)^n}{\Delta x} + \frac{\delta_x f^-(q)^n}{\Delta x} + \frac{\delta_y g^+(q)^n}{\Delta y} + \frac{\delta_y g^-(q)^n}{\Delta y} + \frac{\delta_z h^+(q)^n}{\Delta z} + \frac{\delta_z h^-(q)^n}{\Delta z}\right) . \quad (2.13)$$

Because of the structure of each factor in the LHS of equation (2.13), this factorization can be viewed as an approximate LU decomposition of the matrix defined by the LHS of equation (2.9).

A two step procedure to solve this system can be implemented as follows

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A^+ \cdot + \frac{\Delta t}{\Delta y}\delta_y B^+ \cdot + \frac{\Delta t}{\Delta z}\delta_z C^+ \cdot\right)\Delta q^* = RHS , \quad (2.14)$$

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A^- \cdot + \frac{\Delta t}{\Delta y}\delta_y B^- \cdot + \frac{\Delta t}{\Delta z}\delta_z C^- \cdot\right)\Delta q^n = \Delta q^*, \tag{2.15}$$

where Δq^* represents an intermediate product of the factorization.

The approximate factorization scheme introduces error terms in the LHS of the numerical derivation. In the steady state case it is assumed that these errors will be driven out as time–steps advance because the Δq terms are driven to zero. Similar arguments are made for the unsteady case when Newton iterations are applied. It has also been shown by way of numerical experimentation that this two step scheme appears to be stable for large CFL values[8][9].

Newton Iterations

The discretized form of the Euler equations that appears in equation (2.13) can be used to find a steady state solution to the Euler equations by advancing the equations in time until the Δq^n solution values approach zero, thus giving a steady state solution. At this point, the terms that appear on the LHS have no effect on the solution (or the error). However, this is not the case when unsteady solutions are of interest. One approach to constructing a LHS implicit operator that does not effect the RHS in a way that is compatible with the derivations thus far can be illustrated by noting that the linearization procedure indicated in equations (2.3) is strikingly similar to that of a Newton method iteration.

The vector form of the Newton method can be written as (refer to [10])

$$L'(q^{p-1})(q^p - q^{p-1}) = - L(q^{p-1}) . \tag{2.16}$$

For the present case, the function $L(q)$ given by equation (2.2) is used for the RHS of equation (2.16), or

$$L(q^{n+1}) = q^{n+1} - q^n + \Delta t\left(\frac{\delta_x f(q)^{n+1}}{\Delta x} + \frac{\delta_y g(q)^{n+1}}{\Delta y} + \frac{\delta_z h(q)^{n+1}}{\Delta z}\right). \quad (2.17)$$

Using equation (2.17) in equation (2.16) and applying an approximate factorization yields the discretized equation given by

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A^+ \cdot + \frac{\Delta t}{\Delta y}\delta_y B^+ \cdot + \frac{\Delta t}{\Delta z}\delta_z C^+ \cdot\right)\left(I + \frac{\Delta t}{\Delta x}\delta_x A^- \cdot + \frac{\Delta t}{\Delta y}\delta_y B^- \cdot + \frac{\Delta t}{\Delta z}\delta_z C^- \cdot\right)\Delta q^{n+1,p} =$$

$$q^n - q^{n+1,p} - \Delta t(R^{n+1,p}),$$

where

$$\Delta q^{n+1,p} = q^{n+1,p+1} - q^{n+1,p}, \text{ and}$$

$$R^{n+1,p} = \frac{\delta_x f^+(q)^{n+1,p}}{\Delta x} + \frac{\delta_x f^-(q)^{n+1,p}}{\Delta x} + \frac{\delta_y g^+(q)^{n+1,p}}{\Delta y} +$$

$$\frac{\delta_y g^-(q)^{n+1,p}}{\Delta y} + \frac{\delta_z h^+(q)^{n+1,p}}{\Delta z} + \frac{\delta_z h^-(q)^{n+1,p}}{\Delta z}. \quad (2.18)$$

Equation (2.18) describes a two step solution procedure much like equations (2.14) and (2.15) that is implemented as follows

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A^+ \cdot + \frac{\Delta t}{\Delta y}\delta_y B^+ \cdot + \frac{\Delta t}{\Delta z}\delta_z C^+ \cdot\right)\Delta q^* = q^n - q^{n+1,p} - \Delta t(R^{n+1,p}), \quad (2.19)$$

$$\left(I + \frac{\Delta t}{\Delta x}\delta_x A^- \cdot + \frac{\Delta t}{\Delta y}\delta_y B^- \cdot + \frac{\Delta t}{\Delta z}\delta_z C^- \cdot\right)\Delta q^{n+1,p} = \Delta q^*. \quad (2.20)$$

Equations (2.19) and (2.20) represent the final form of the Euler solver used herein. Before the Newton iterations can begin, an initial condition must be established. If the last time–step values (time level n) are chosen as initial conditions for the first Newton iteration (time level n+1), then the first step of the Newton iteration is identical to equation (2.13) and thus assumed to represent a reasonable initial condition.

The Form of the Resulting Linear System

As suggested earlier, the form of the linear system that is implied by the implicit operators on the LHS plays a significant role in the analysis of the parallel algorithm. The form of the linear system is dictated by the implicit equation for each computational cell and by the ordering of variables in the solution vector. If the $\Delta q^{n+1,p}$ variables of equation (2.18) are ordered in the solution vector as Fortran orders array entries in physical memory, the resulting linear system is captured by a set of banded matrices as shown in Figure 2.1. In this figure, the dark central diagonal line represents the placement of the sum of the identity matrix and flux Jacobian matrices of equation (2.18) and the gray lines represent the placement of the remaining flux Jacobian matrices.

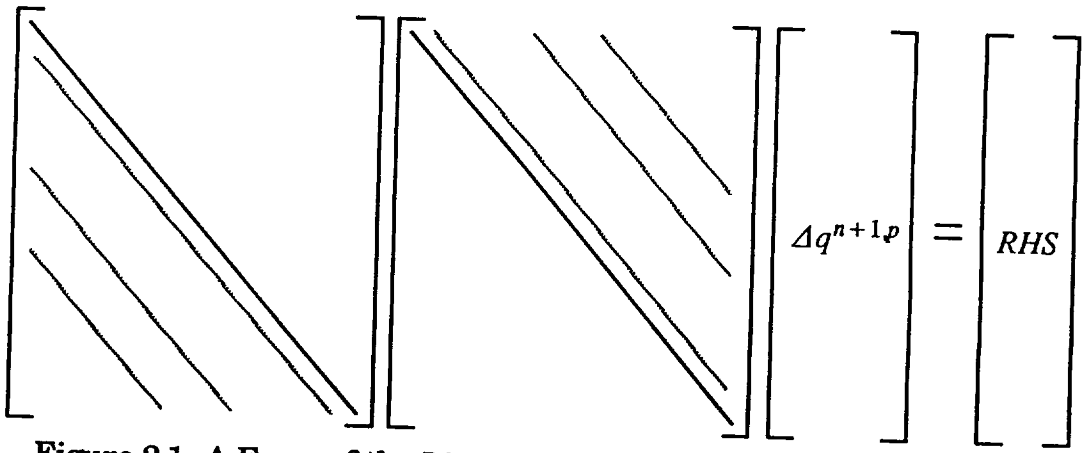

Figure 2.1 A Form of the Linear System Implied by Equation (2.18)

The two step method that results from the approximate factorization method given in equations (2.19) and (2.20) describe a process that involves a forward and backward substitution. The forward substitution that occurs during the solution of equation (2.19) is illustrated in Figure 2.2. In this initial step of the approximate factorization solution, the solution vector is computed from top to bottom by algebraic substitution. The second step of the approximate factorization solution involves a backward substitution that proceeds from the bottom to the top of the solution vector. These substitution steps, as they have been described, are sequential processes and as such dominate execution time when this algorithm is implemented on parallel computer systems.

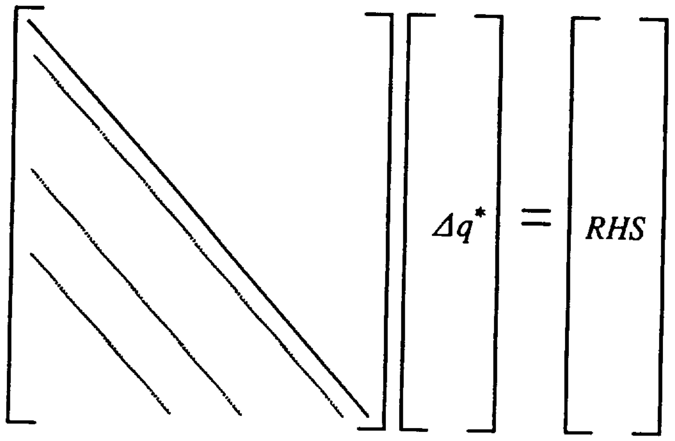

Figure 2.2 A Form of the Linear System Implied by Equation (2.19)

The Parallel Approximate Factorization Algorithm

When implementing the approximate factorization algorithm described by equations (2.19) and (2.20) the substitution procedures become the critical obstacle to obtaining high parallel performance. The reason the substitution procedures are essentially sequential operations can be found by examining the form of the linear system in Figure 2.2. Note that one band of the banded matrix is placed in close proximity (actually it is immediately adjacent) to the main diagonal. The adjacency of the first band forces a strict dependency (a recursive relationship in vector processing terms) in the description of each substitution step. It is possible, however, to move the bands away from the main diagonal by ordering the equations in the linear system based on diagonal planes as presented in [1] and [11]. The form of the linear system when diagonal plane ordering is chosen for the solution vector of equation (2.19) is illustrated in Figure 2.3. The dark gray boomerang shaped region indicates the region of placement for the flux Jacobians and the light gray stair–step line indicates how the substitution procedure can be divided into a sequence of parallel operations. In this stair–step, each step represents a step in the substitution process where all of the equations covered by the rise of the next step can be solved simultaneously.

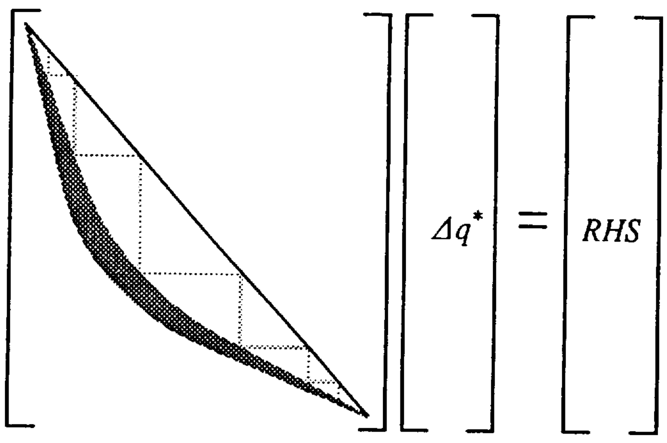

Figure 2.3 The Form of the Linear System Implied by Equation (2.19) Given a Diagonal Plane Based Ordering of the Solution Vector

Observations regarding the impact of diagonal plane ordering on the substitution procedure led to the development of an algorithm for the Cray vector architecture that demonstrated significant performance gains over the previous sequential substitution procedures[11]. Since the diagonal plane ordering allows many equation substitutions to occur simultaneously, this algorithm may also provide performance gains on general parallel architectures.

The Modified Algorithm

For reasons that will be described in Chapter III, the parallel approximate factorization algorithm is not appropriate for massively parallel computer systems. In light of this fact, an alternative approach to obtaining greater parallel performance in the substitution passes of the approximate factorization algorithm must be considered. One possible approach is to partition the solution vector and "decouple" the partitions so that substitutions can occur in each partition simultaneously. The partitions of the solution vector are decoupled by zeroing out entries in the linear system such that each partition can begin substitution processes independently. The decoupling process is illustrated when two partitions are considered for equation (2.14) in Figure 2.4. In this figure, the entries that fall in the shaded region are replaced with zeros in order to decouple the lower partition from the upper partition. A similar procedure would be applied to the backward substitution step described by equation (2.15). Although this decoupling introduces errors in the solution process, it is assumed that the Newton iterations will be able to reduce these errors in much the same way as these iterations reduce the error introduced by approximate factorization. In addition, Belk[12] demonstrated that convergence can be obtained when this method is used and that the location of shocks will be correct even when they cross partition boundaries.

This approach, identified as the modified algorithm, can provide additional parallelism, but at the cost of additional Newton iterations. For the three and four partition cases that Belk studied, the number of time–steps required to achieve a steady state solution was increased by 25%[12]. A 25% penalty seems a reasonable price to pay considering that a four partition case potentially could extract a four fold increase in performance on a parallel processor. A more comprehensive study of the cost effectiveness of this method will be a subject of investigation in this thesis.

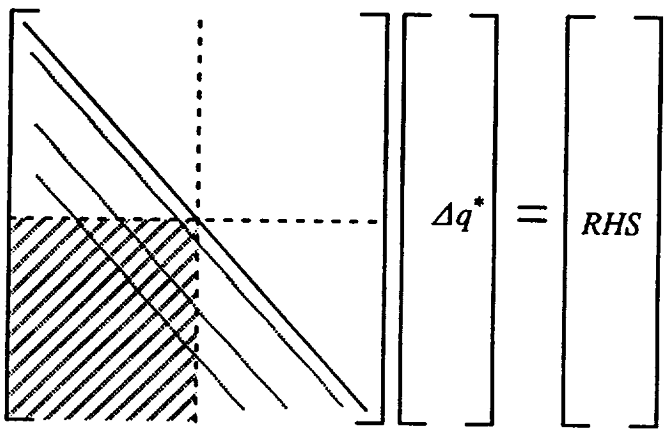

Figure 2.4 An Illustration of Decoupling Solution Vector Partitions

Implementation of the Modified Algorithm

It would seem that locating and zeroing the entries in the linear system for the modified algorithm might be a complex process. In the actual implementation the entries in the linear system are not actually zeroed, instead the coefficients (represented by the Δq values in the solution vector) of the flux Jacobians are zeroed as necessary when the substitution passes proceed. This process is accomplished by way of domain decomposition. In the domain decomposition approach, the grid is subdivided into a number of domains (or blocks) and these domains define the partition of the solution vector. When the forward or backward substitution occurs in each domain, zero Δq values are injected into the substitution procedure at domain boundaries resulting in an algorithm that is simple to implement.

CHAPTER III

ANALYSIS OF THE PARALLEL ALGORITHM

The objective of a parallel algorithm analysis is to make a value judgement based on the appropriateness of an algorithm for implementation on parallel computer systems. An essential component of this value judgement, and resulting categorization, is performance measurement captured by a set of defined performance metrics. The most common performance metric, the rate of operations performed per second, is valuable for assessing an algorithm's cost effectiveness for a given problem on a given hardware platform, but it possesses virtually no predictive capability on how an algorithm will perform in any other instance than the one that was measured. For sequential machines, predictive performance metrics are derived from memory access patterns, ratios of integer to floating point operations, and other measurements that can be used to model how subunits of a sequential architecture will be utilized. For parallel architectures the definition of new predictive performance metrics are evolving. This chapter will discuss several predictive performance metrics and adapt them to demonstrate that the parallel approximate factorization algorithm described in Chapter II is a poor algorithm for massively parallel implementation. The suitability of the modified algorithm will also be discussed.

Amdahl's Law

Amdahl's Law[13], a frequently referenced criterion for analysis of parallel algorithms, is also the source of the popular parallel performance metric, speedup. Speedup is defined as the ratio of the time required to execute an algorithm on one processor to the time required to execute the algorithm on N processors. Amdahl argued that if s percent of an algorithm's execution time was inherently sequential, and p percent of an algorithm's execution time was completely parallelizable, then the algorithm could at most execute in $1/s$ of the time required by a sequential processor. This argument is easily demonstrated given the time required to execute the sequential program will simply be the total $s + p = 100\%$, and the time required to execute on N parallel processors will be $s + p/N$. Given this, the computation of the performance metric speedup is easily expressed as

$$speedup = \frac{sequential\ execution\ time}{parallel\ execution\ time} = \frac{s+p}{s+p/N} = \frac{1}{s+p/N}. \qquad (3.1)$$

The maximum speedup can be determined by taking the limit of equation (3.1) as the number of processors approaches infinity given by

$$maximum\ speedup = \lim_{N \to \infty}\left(\frac{1}{s+p/N}\right) = \frac{1}{s}. \qquad (3.2)$$

This argument illuminates the severe limitations that parallel processing places on its applications since an algorithm that spends only ten percent of its work in sequential execution can at most gain a ten fold improvement in speed.

An important variation of the speedup metric is parallel efficiency defined by the average utilization of the parallel processing elements. This metric is defined by

$$\textit{parallel efficiency} = \frac{\textit{speedup}}{N}. \tag{3.3}$$

One potential drawback of the speedup and parallel efficiency metrics is that they tend to measure architectural constraints as well as algorithmic ones. Often an application with good speedups no longer achieves them when the application is optimized. This case occurs when the sequential fraction of the application is dominated by message exchange times that are not reduced when the parallel work is reduced by optimization. Because of this phenomena, the fastest algorithm implementations often give the poorest speedup measurements[14][15]. For this reason speedup and parallel efficiency can be valuable metrics for measuring the efficiency of an application on a given hardware platform but they are not useful for the purpose of categorizing parallel algorithms.

A Definition of Parallel Algorithms

In an informal definition, an algorithm might be defined as a recipe or a description of a set of processes required to achieve a defined goal. In this informal setting, a parallel algorithm might be defined as a recipe that describes concurrent processes. Likewise, a formal definition of parallel algorithms may be described by first expanding the definition of a sequential algorithm defined by an ordered set of operations given by

$$\mathcal{A} = \{O_1, O_{2,} \cdots, O_n\}. \tag{3.4}$$

Now define a partially ordered algorithm as an ordered set of sets given by

$$\mathcal{P} = \{\phi_1, \phi_2, \cdots, \phi_m\}, \tag{3.5}$$

$$\phi_1 = \{O_1, O_2, \cdots, O_j\}, \phi_2 = \{O_{j+1}, O_{j+2}, \cdots, O_k\}, \cdots$$

such that any algorithm constructed from an arbitrary ordering of the operators contained in sets ϕ_1 through ϕ_m produce an algorithm that is equivalent to $\mathcal{A}$. Two algorithms are equivalent if and only if applying all operations in each algorithm produces the same results in every case. A partially ordered algorithm is a parallel algorithm if each set ϕ_1 through ϕ_m exhibits the property of operation independence. The set of operations ϕ_i has the property of operation independence if and only if no two operations contained within ϕ_i read or write to the same data space written to by the other.

The character of the parallelism defined by a parallel algorithm can be captured in a parallelism profile defined for the parallel algorithm $\mathcal{P}$ as the set of ordered pairs given by

$$\psi(\mathcal{P}) = \left\{\left(P_{\phi_1}, T_{\phi_1}\right), \left(P_{\phi_2}, T_{\phi_2}\right), \cdots, \left(P_{\phi_m}, T_{\phi_m}\right)\right\}. \tag{3.6}$$

The degree of parallelism, P_{ϕ_i}, is defined by the number of elements in the set ϕ_i. The operation execution time, T_{ϕ_i}, is defined as the maximum execution time required to complete any operation in the set ϕ_i.

Scaled Speedup and Fixed-Time Size-Up

Traditionally, the set of applications considered as viable for computational solution grows as the power of computational engines grow. Likewise, the size of the problems considered for computational solution typically grows. This property can be attributed to two factors: 1) problem sizes considered on earlier computational engines tend to be less interesting because most have been solved, and 2) modeling of key physical phenomena becomes a limiting factor in many design processes giving rise to a demand for more complex and larger scale computational modeling capability. Gustafson[16] argues that Amdahl's Law should be reevaluated in light of larger parallel machines since the sequential fraction of an algorithm is typically a function of the size of the problem considered. Gustafson further argues that a scaled speedup metric (that is a measurement of speedup as both problem size and number of processors are increased) is a more appropriate measurement for parallel applications. A significant criticism of scaled speedup is that problem sizes should be driven by application requirements, not measurement requirements. (What value is there in knowing a problem of a large size can be solved efficiently when the need for the solution of a problem that size hasn't been clearly demonstrated?)

Further work of Sun and Gustafson[17] argues for the use of a fixed–time size–up metric as a measure of performance. In fixed–time size–up, the size of the problem that executes in a predetermined amount of time is recorded as the number of processors increase. It is argued that the fixed–time size–up metric is "fairer" than speedup because results are much less affected by optimization of the parallel work. A further assertion is made that fixed–time size–up is machine independent which implies that it could be a good metric for measuring an algorithm's performance independent of the hardware platform.

Scalability of the Parallel Approximate Factorization Algorithm

Since significant speedups were achieved using a Cray vector processor for the implementation of the forward and backward substitution phases of the approximate factorization algorithm[11], it is natural to ask if the parallelism used to gain vector processing speedups could be used in a multicomputer setting. This question has been addressed by Welch[18] where he developed a fine grained decomposition of the problem for multicomputers. Through the use of a communications coprocessor that was capable of scheduling tasks efficiently, granularity could be adjusted by appropriately mapping computational cells to processors[19]. A variety of mappings of cells to processors was considered. No speedups were achieved with this approach primarily due to small problem size and high message latencies. Whether larger problem sizes could exploit multicomputer parallelism remained an open question of this work.

The question of using larger problem sizes to utilize multicomputer parallelism was addressed in [20] where it was shown that the problem size required to utilize an increasing number of processors grows superlinearly. This suggests that if a large multicomputer is considered for implementation of the parallel approximate factorization algorithm, a significantly large problem size would be required to effectively use these processors.

In an attempt to further understand the nature of this algorithm, a measurement of the size–up metric will be made on an idealized parallel processor with the unique property of having zero message passing cost. The idea behind this measurement is that it will provide an upper bound on the expected performance of this algorithm. The execution time required by an idealized parallel processor with zero message passing cost can be modeled given the parallelism profile of a given parallel algorithm. Given this profile, the time required to compute a parallel algorithm $\mathcal{P}$ on an idealized multicomputer consisting of N processors is given by

$$\textit{execution time} = \sum_{i=1}^{m} \left\lceil \frac{P_{\phi_i}}{N} \right\rceil T_{\phi_i} . \tag{3.7}$$

The parallelism profile for the parallel approximate factorization algorithm can be defined in three stages of operations: 1) construction of the linear system, 2) performing the forward substitutions, and 3) performing the backward substitutions. These operations are performed on a three dimensional computational space composed of computational cells organized into a three dimensional array of dimensions m, n, and p. The first operation (constructing the linear system) is comprised of computing the right hand side of the equations and computing the flux Jacobians. These operations can be performed in any order and thus become the first step in the parallel algorithm. Since 60 percent of the work of this algorithm is spent performing this operation, and the work is distributed evenly between cells, the time required to execute this operation can be expressed as 0.6 normalized time units per cell.

To describe the ordering required to correctly perform the forward substitution consider that each cell in the computation space can be identified via three indices: i, j, and k where $1 \leq i \leq m$, $1 \leq j \leq n$, and $1 \leq k \leq p$. At first the cell identified by $i + j + k = 3$ (given by 1+1+1 = 3) is the only computational cell that can contribute to the forward sweep. Next, three cells defined by $i + j + k = 4$ (2+1+1 = 1+2+1 = 1+1+2 = 4) can be operated on concurrently. This process continues until only one cell, defined by $i + j + k = m + n + p$, can contribute to the computation. The backward substitution is identical except the order of these steps is reversed.

The description of the ordering required for the forward and backward substitution phases can be used to develop a parallel algorithm for the substitution process. The parallelism profile of this parallel algorithm for a 10 by 10 by 10 problem size is given in Figure 3.1. Since 40 percent of the work required by the complete algorithm is required to perform the forward and backward sweeps, a normalized time of 0.2 per computational cell is required to perform a forward or backward sweep operation.

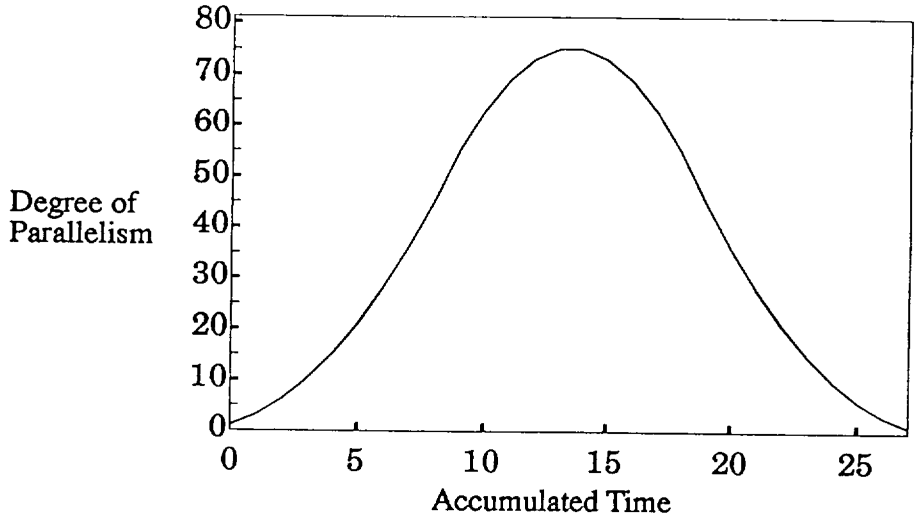

Figure 3.1 Degree of Parallelism for a 10x10x10 Problem Size

With the description of the parallel algorithm complete, the execution time on the idealized multicomputer can be realized using equation (3.7). The program listed in Appendix A was used to compute equation (3.7) and iteratively find the fixed–time size–up curve. Before computation of the size–up curve can begin, a time must be chosen. For a first pass the time required to execute a 10 by 10 by 10 problem size on a single processor was considered. The size–up curve for this case is given in Figure 3.2. The problem size was normalized to the 10 by 10 by 10 problem size and the dotted line represents an optimal size–up curve.

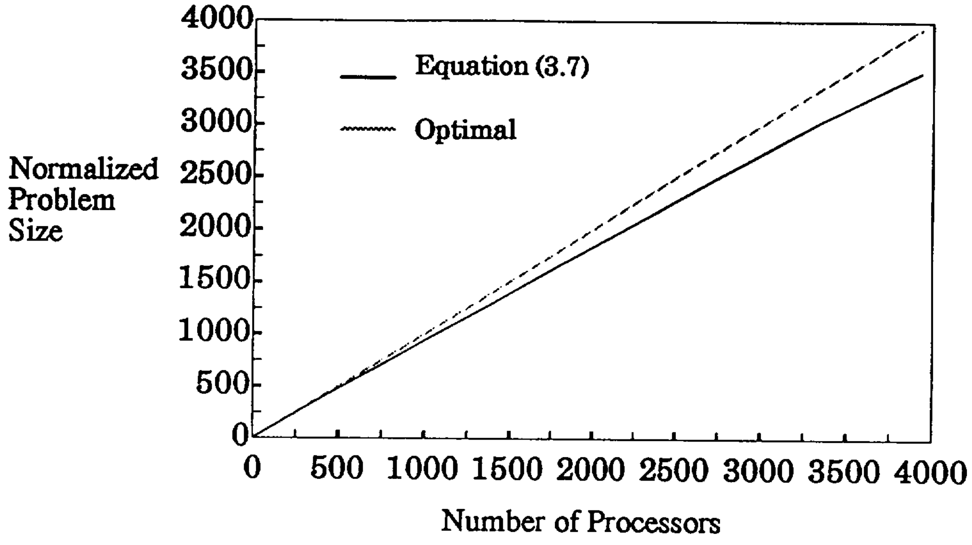

Figure 3.2 Size–Up Based on 10x10x10 Sequential Execution Time

The results shown in Figure 3.2 look very promising, but remember that these results are derived from an idealized model multicomputer. In practice the amount of parallelism that could actually be exploited would be much less because cell level granularity would be too fine to efficiently exploit on any current multicomputer system. One way to increase the granularity of the parallel algorithm would be to cluster cells and solve each cluster as if it were a sequential operation. A clustering of cells into cubes of size 3 by 3 by 3 would increase the granularity of the algorithm at the cost of reducing the available parallelism. Because the structure of the parallelism would essentially be the same, we can model the clustered algorithm with a smaller problem size. The 10 by 10 by 10 clustered solution can approximately be modeled by choosing the time for the fixed–time size–up curve to be the execution time on a single processor of a 3 by 3 by 3 problem. The results of this experiment are shown in Figure 3.3.

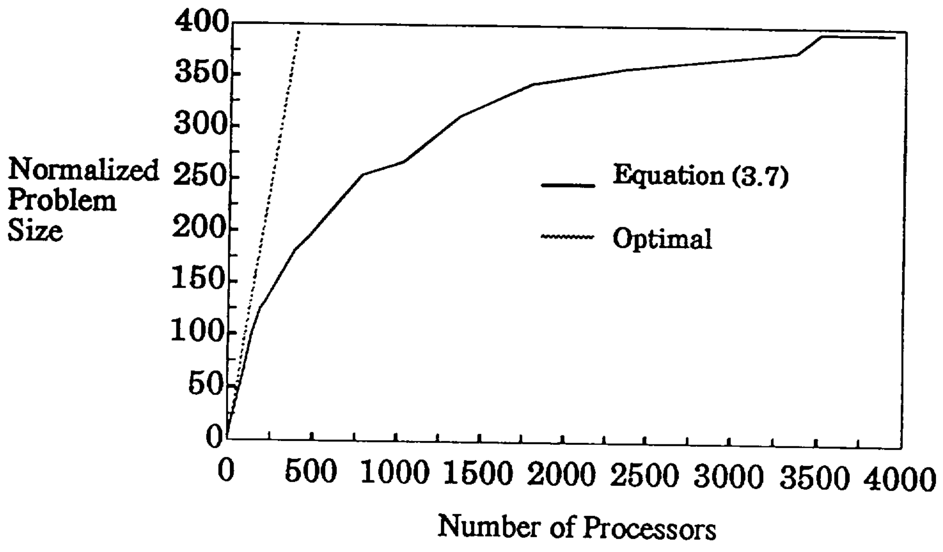

Figure 3.3 Size–Up Based on 3x3x3 Sequential Execution Time

The results shown in Figure 3.3 are less promising, but what is more profoundly disturbing is the wide variance in results of the size–up measurement in the two previous cases. It seems that the size–up curve can look arbitrarily good depending on the choice of the fixed–time constraint. A possible explanation of these variances is that by starting with the larger fixed–time constraint, the unused parallelism of the sequential case is "borrowed" during the initial stages of size–up. If this is the case, a better size–up measurement could be taken by first "boiling off" the sequential case parallelism by performing a fixed problem size scale–up as suggested by Amdahl's Law and then use the scaled execution time to do a fixed–time scale–up. The procedure for this approach would work as follows: 1) select an arbitrary problem size, 2) do a fixed–size scale–up until parallel efficiency is at 65 percent, and 3) use the execution time from fixed–size scale–up to do a fixed–time size–up. The rationale for choosing a parallel efficiency of 65 percent will be given later. When this approach is used the curve shown in Figure 3.4 results.

Figure 3.4 is based on the application of the previous procedure for an initial problem size of 10 by 10 by 10. The same basic curve resulted from applying the same procedure to problem sizes ranging from 3 by 3 by 3 to 20 by 20 by 20 suggesting that this method is a much more reliable measurement than simple size–up where the fixed–time constraint is left unspecified.

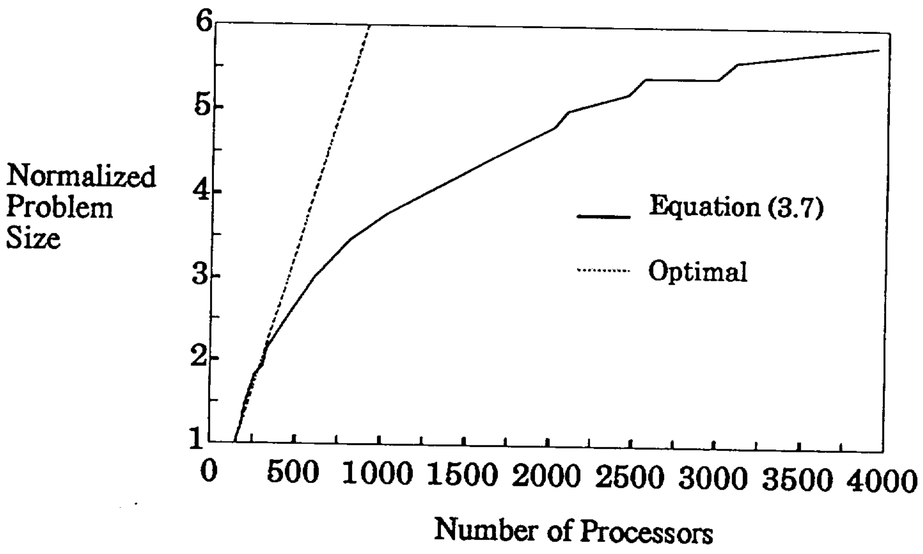

Figure 3.4 Size–Up After 65% Parallel Efficiency Scale–Up

Obviously the curve shown in Figure 3.4 is not promising if nearly 4000 additional processors are required to solve a problem only six times larger in the same amount of time. It is still hard to feel comfortable with these results since the first size–up curve did show a near optimal size–up curve. It seems very large problems could be solved with this algorithm on massively parallel processors. The confusion that results from examining the size–up metric can be linked to the fact that the size–up metric makes no attempt to measure cost effectiveness, much less optimize it.

Sterling and Laprade[21] used a cost effectiveness metric in the analysis of a simple parallel model given by

$$\text{Cost Effectiveness} = \frac{\text{Performance}}{\text{Resource Cost}}. \tag{3.8}$$

In this simple measure of cost effectiveness, performance is defined by the amount of work performed per amount of time and resource cost denotes the cost of computational resources. For parallel processors, the resource cost can be based on node–hours defined as the product of execution time and the number of processor used. Given this measure of resource cost, equation (3.8) can be written in more explicit terms as

$$\text{Cost Effectiveness} = \frac{\frac{\text{Work}}{t}}{Nt} = \frac{\text{Work}}{Nt^2}. \tag{3.9}$$

In equation (3.9), *Work* is defined as the number of operations required to solve the problem on a sequential architecture; and the variable t is defined as the time required to obtain a solution on the parallel processor.

Sterling and Laprade[21] showed that for their simple parallel model, cost effectiveness was maximized when parallel efficiency was at 50 percent. In order to determine the point of optimal cost effectiveness for the parallel approximate factorization algorithm, the cost effectiveness and efficiency were measured during a fixed–size scale–up of a 10 by 10 by 10 problem. Figure 3.5 represents the cost effectiveness versus efficiency curve resulting from this measurement demonstrating that optimal cost effectiveness occurs at 65% parallel efficiency.

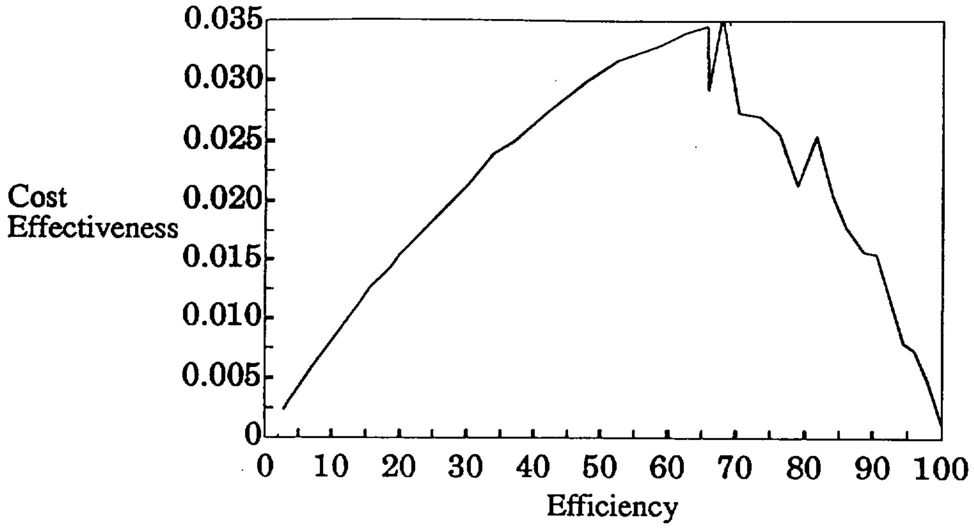

Figure 3.5 Cost Effectiveness Curve for a 10x10x10 Problem Size

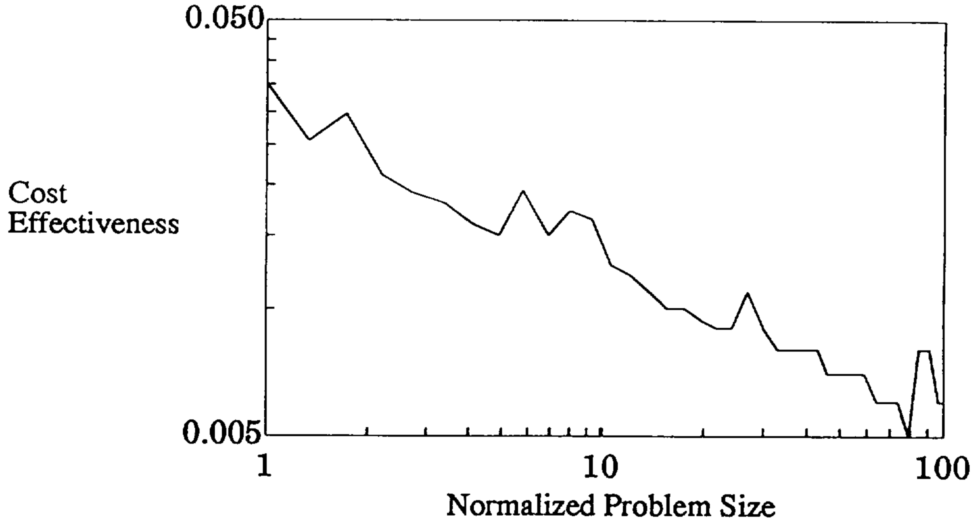

Figure 3.6 Optimal Cost Effectiveness Versus Problem Size

Perhaps a more interesting measurement of a parallel algorithm than the size–up measurement is the measure of the optimal cost effectiveness as the problem size scales. Figure 3.6 illustrates this measurement for the parallel approximate factorization algorithm. This figure demonstrates that if it is not cost effective to solve a small problem using the parallel approximate factorization algorithm, it will not be cost effective to solve larger problems. Considering this and the results of Welch[18], the parallel approximate factorization algorithm is considered to be of limited value as an algorithm for massively parallel computer systems.

Analysis of the Modified Algorithm

The forward and backward substitution passes are the key sources of scalability problems in the parallel approximate factorization algorithm. Upon examination, the transfer of Δq values from one computational cell to another is the key data dependency driving the ordering of the substitution passes. A possible way to improve the parallel structure of the approximate factorization algorithm would be to develop approximations for Δq that are free of dependencies. Belk[12] demonstrated that a computational space can be partitioned into blocks, and that a correct solution can be obtained with $\Delta q = 0$ values transferred at block boundaries. With this relaxed criterion for the exchange of Δq values, all blocks can execute concurrently within a solution iteration. The analysis of this parallel algorithm follows two courses: 1) analysis of the parallel structure of the algorithm, and 2) analysis of the numerical costs introduced by relaxing Δq exchanges.

The analysis of the parallel structure of the modified algorithm assumes that a computational problem space will be partitioned into blocks, and zero Δq boundary exchanges will be performed such that computation of each block can proceed independently within an iteration. Since the substitution phases that exist within blocks are implemented as sequential algorithms in the parallelized turbo–machinery application, the parallelism that results from the substitution phases within the blocks are not considered.

Given this model, a simulation of this algorithm's execution on the idealized multicomputer can be developed. The efficiency curve for this algorithm varies widely as the number of processors are scaled mainly due to the stair–step speedup that occurs as the number parallel elements become divisible by the number of processors. This behavior makes it impossible to plot the cost effectiveness versus efficiency curve used earlier to determine the point of optimal cost effectiveness. However, the point of optimal cost effectiveness can easily be read from a plot of the cost effectiveness metric versus the number of processors in the idealized multicomputer. The plot of the cost effectiveness and efficiency is given in Figure 3.7 for a problem case consisting of 27 blocks. In this case the point of optimal cost effectiveness is found when the number of processors is equal to 27. The efficiency for this case is 100 percent.

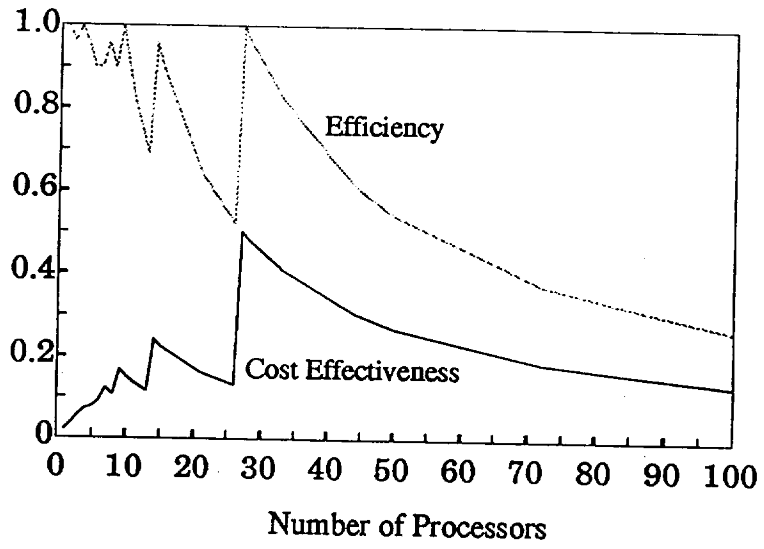

Figure 3.7 Cost Effectiveness and Efficiency as Processors Scale

Figure 3.7 shows that for at least one case, optimal cost effectiveness occurs when the index of parallelism is equal to the number of processors.

When N_p is the index of parallelism, W_s is the sequential work required by a single domain, t_s is the time required to execute a single domain, and $N = N_p$, the cost effectiveness equation becomes processor and problem size independent. The equation for cost effectiveness for the case where $N = N_p$ is given by

$$Cost\ Effectiveness = \frac{Work}{Nt^2} = \frac{W_s N_p}{N\ (t_s N_p/N)^2} = \frac{W_s}{t_s^{\,2}}. \tag{3.10}$$

Equation (3.10) demonstrates that if the problem is scaled up such that domains of equal size are added at the same rate as processors are added, the cost effectiveness will remain constant, and thus the parallel structure of this algorithm is scalable.

The analysis of the parallel structure of the modified algorithm did not consider the communication structure required to implement the algorithm. The communications structure that results from partitioning the domain is largely a mesh topology and is expected to map in a scalable way to architectures that implement mesh topology communication networks.

Measuring Numerical Costs in the Modified Algorithm

The modified algorithm contributes error terms to the left hand side of numerical formulation given in Chapter II. The assumption is that the Newton iterations used to converge to a final solution (at a given time–step) will also diminish the effect of error contributions from the modified algorithm. The performance of the modified algorithm can become degraded because additional Newton iterations may be required to reach convergence. Because the precise number of extra iterations that will be required to reach convergence can not be predicted, numerical experimentation is required to estimate these effects.

Because the cost effectiveness formulation is based on the work required by the sequential algorithm and the time required for solution on a parallel system, the cost effectiveness measurement can be used to evaluate the results of numerical experiments. Numerical experiments will proceed along similar lines as the previous analysis. First an optimal cost effectiveness should be determined by increasing the level of partitioning in a fixed–size problem. If the problem size can then be scaled up without reduction in maximum cost effectiveness, this algorithm will be considered scalable.

On the Cost Effectiveness Metric

Traditionally, measurements of parallel algorithm implementations have been limited to execution times, floating point performance, and speedup. The problem with these measurements is that they do not fit into any coherent theory by which algorithms can be classified. The cost effectiveness analysis presented in this chapter begins to solve this problem by introducing a classification that can be reliably measured, that is scalability. The basis of this analysis is the determination of the optimal cost effectiveness curve for increasing problem sizes. If measurements are not taken at the optimal cost effectiveness they are useless since the potential of the algorithm was not measured. This is the key problem with the fixed–time size–up measurement when the fixed–time is not set at the point of optimal cost effectiveness. That is, a fixed–time size–up measurement could be made entirely in a cost ineffective domain causing misleading results as was the case in Figure 3.2.

The power of the optimal cost effectiveness curve's ability to classify parallel algorithms can be demonstrated with the following example. First consider an optimal sequential algorithm that does not parallelize. Since this is a sequential algorithm, the execution time will be proportional to the work. Given this, the cost effectiveness as a function of problem size can be computed as

$$\mathit{Cost\ Effectiveness} = \frac{\mathit{Work}}{Nt^2} = \frac{W_{seq}}{\left(k\ W_{seq}\right)^2} = \frac{1}{k^2 W_{seq}}, \qquad (3.11)$$

where k is the constant of proportionality that relates *Work* to execution time. Equation (3.11) demonstrates that the cost effectiveness of a sequential application declines as a reciprocal of problem size.

Also considered in this example is a parallel algorithm that is not scalable such as the parallel approximate factorization algorithm. This algorithm is able to achieve a high cost effectiveness for small problem sizes but is unable to maintain this cost effectiveness as the problem size is increased. For the purposes of illustration, assume that this algorithm's cost effectiveness declines linearly.

Finally, consider a scalable parallel algorithm that maintains a constant cost effectiveness as the problem size is increased provided that enough processors are available. Now a plot of the optimal cost effectiveness of each algorithm can be used to determine the optimal algorithm based on problem size. For small problem sizes, the algorithm that is not scalable achieves a higher cost effectiveness than any of the other algorithms. Because the scalable algorithm can maintain its cost effectiveness as the problem size is increased, it will eventually become the most cost effective as is demonstrated in Figure 3.8.

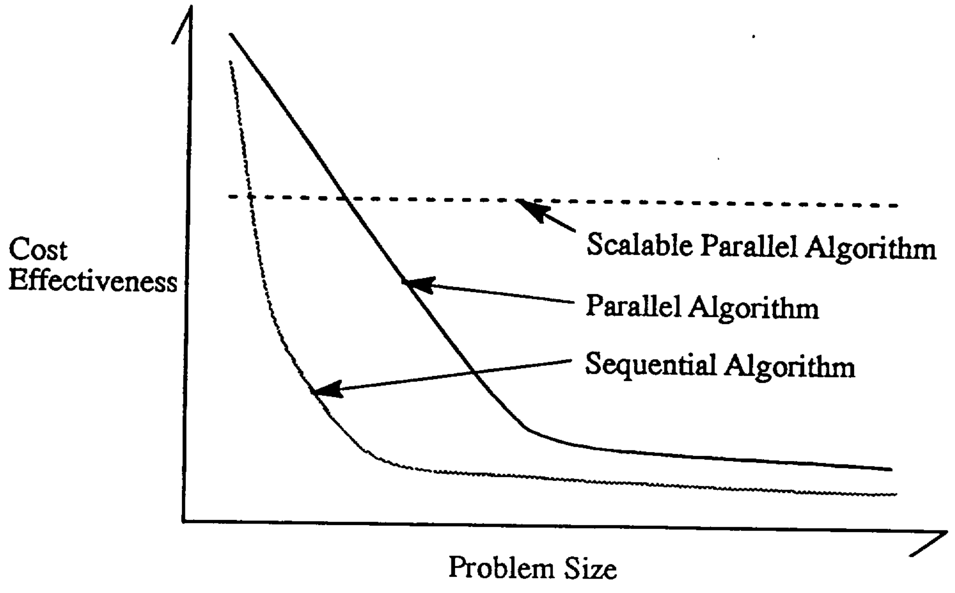

Figure 3.8 Maximum Cost Effectiveness versus Problem Size

One essential criterion required by a scalable algorithm if it is to maintain a constant cost effectiveness is the availability of processor resources. In practice, no parallel computing system has an infinite number of processors so at some point any scalable algorithm will reach a problem size beyond which no additional performance gains can be made. At this point the cost effectiveness should begin to decline as a reciprocal of the problem size for the same reasons that the sequential algorithm exhibited this behavior. Making this observation, it is possible to see how cost effectiveness can be used to determine which algorithm will be optimal under conditions of limited processor resources. For example, consider two scalable algorithms: algorithm A and algorithm B. Algorithm A is able to attain a higher cost effectiveness when processors are available because it can effectively utilize more processors on a given problem size than algorithm B. The implication of this relationship is that algorithm A will reach the limit of available processors at a smaller problem size than algorithm B. When this occurs algorithm A's cost effectiveness will begin to decline while the cost effectiveness of algorithm B will continue to remain constant which gives algorithm B the opportunity to become the most cost effective for larger problem sizes as illustrated in Figure 3.9.

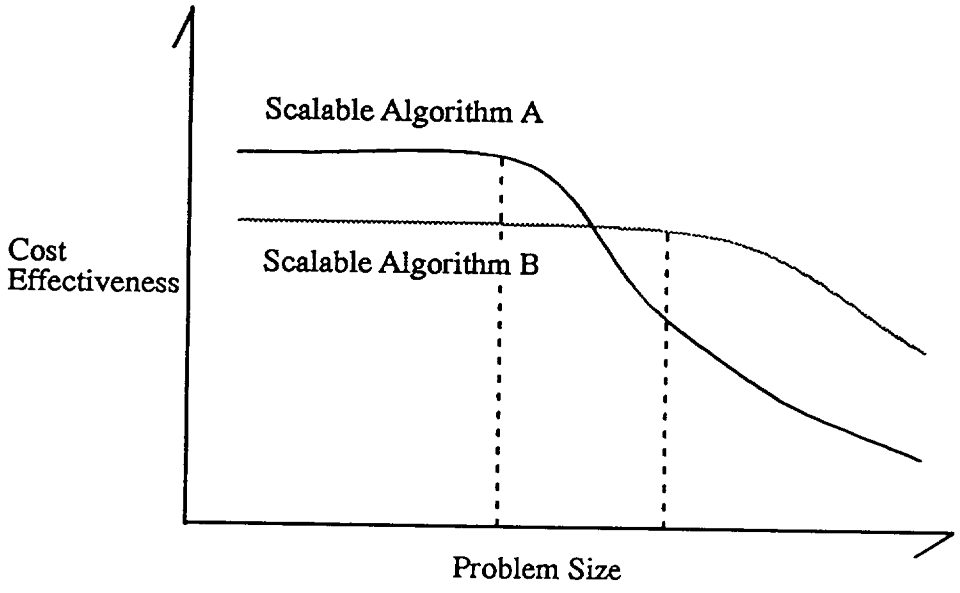

Figure 3.9 Maximum Cost Effectiveness as Processor Limits are Reached

The previous example can be used to describe why the parallel approximate factorization algorithm is appropriate for the Cray and not for massively parallel computer architectures. In the case of the Cray, a relatively small number of vector processing elements act concurrently with low data transfer costs. In this environment, the parallel approximate factorization algorithm gains the maximum advantage since low data transfer costs allow for a fine grained implementation. On the other hand, the modified algorithm will never be able to obtain enough concurrent computing resources to become cost effective since the Cray implements a relatively small vector size. In a massively parallel computer architecture the tables are turned since coarser grained computations are required to overcome communication costs and there is an abundance of concurrent computing resources. In this case, the parallel approximate factorization algorithm is never able to attain a cost effective solution because of granularity limits. In contrast, the modified algorithm takes advantage of coarse granularity and abundant parallelism to obtain cost effective solutions for large problem sizes.

The examples presented here demonstrate the power of the cost effectiveness analysis in determining the desirability of different parallel algorithms. It seems that this metric is highly relevant to parallel algorithm analysis and is a powerful tool to understanding the complex behaviors that parallel algorithms can exhibit.

CHAPTER IV
IMPLEMENTATION

The process of parallelizing a computer program developed for a particular application involves developing an evolving strategy for implementing the application on a parallel architecture. This process begins with an analysis of the algorithm used in the application. This analysis has resulted in the selection of the modified algorithm described in Chapter II as a potentially cost effective approach. The strategy for parallelization is further refined during study of the application's sequential implementation because decisions made when an algorithm is implemented on a sequential architecture can implicitly assume sequential execution. The final step in the process involves making compromises between reducing the degree of modification required of the sequential application and obtaining high utilization of a given parallel architecture.

The final result of the parallelization strategy is a set of incremental modifications applied to the sequential program such that the expression of parallel execution is accomplished by way of architectural and system software semantics. The turbomachinery application developed by Janus[1] was chosen for parallelization because it solved a class of problems that tax the computational capacity of traditional sequential architectures, and because its multi–block capability could be used to implement the modified algorithm described earlier.

Object–Oriented Fortran

Object–Oriented Fortran[22] was selected as the programming language for the parallel implementation of the turbomachinery application. Object–Oriented Fortran provides an interface loosely based on object–oriented design principles with an emphasis on execution and synchronization of multiple object instances on parallel computing platforms. Object–Oriented Fortran is a portable parallel programming language that can allow programs to port without change to a wide range of parallel architectures including networks of workstations. This feature alone allowed for the use of workstation style debuggers and allowed idle workstation cycles to be used during the development stages of the parallel application. In addition to these features, Object–Oriented Fortran provides a level of abstraction that makes dealing with mapping problems to processors and packaging messages less tedious than portable message passing libraries.

Preparation of the Application

The sequential turbomachinery application was modified in several stages in order to prepare it for the final parallel implementation. The first issue that was dealt with was the treatment of common block storage in the original application because common block storage was used as a mechanism for passing data between subroutines. This presented a problem in the parallel implementation on distributed memory architectures since the subtle coordination of a global data space as a means of communication between subroutines needed to be redefined in terms of the explicit message passing and synchronization semantics of the distributed memory model. In an effort to get a handle on where common block storage was used to communicate between what would be separate distributed memory modules, the common block mechanism was incrementally replaced with subroutine arguments as a way of passing data between subroutines. Although this approach tended to make the argument lists for subroutines large, it also made the data dependencies between subroutine calls much more obvious.

During the conversion from common block storage to subroutine arguments as a means of passing information between subroutines two points became apparent: 1) the sequential implementation of communication between blocks depended strongly on sequential execution for proper synchronization, and 2) "hardwired" parameters in the boundary condition and residual calculation subroutines would make testing a wide variety of domain decomposition strategies difficult. In light of these two findings, the next step in preparing the sequential application for parallel execution was to strip out all of the multi–block support and develop a general single block code that would have no "hardwired" parameters in its implementation of the boundary condition and residual calculation routines. This application could then be advanced to a parallel multi–block code by re–implementing the inter–block boundary management with distributed memory semantics. The development of the general single block code required many months of effort dominated by the process of removing common block storage as a means of passing data between subroutines.

Development of the Parallel Application

The development of the parallel application involved adding parallel constructs to the single block code that would enable multiple blocks to execute in parallel by way of the modified algorithm. A block is represented by a three dimensional region of the simulation space constructed from a structured grid that is a rectangular prism in computational space. The faces of the rectangular prism for a given block are divided into rectangular surface patches. These patches were used in the single block implementation of the solver to impose boundary conditions such as impermeable surfaces, inflow, or outflow. In the parallel case these boundary conditions were extended to include a connective boundary condition. The connective boundary condition is described by the address of another patch which becomes a repository of q variable information. The patch address consists of a unique block identifier and a patch identifier that is unique within a block. In addition to the transfer of q values through the use of the connective boundary condition, rotation operations could be applied to the momentum vector component of the q vector when required to impose symmetry about the x axis. When the connective boundary condition was added to the single block code, the application was capable of parallel solution of multi–block problems that did not involve relative motion between blocks. For turbomachinery problems an additional communication semantic was required, namely the communications peculiar to the interface between domains that did involve relative motion between blocks represented by blocks rotating at different angular velocities.

In the sequential application, transfer of information between boundaries shared by blocks in relative motion was handled with a communication management structure called a ring buffer. The ring buffer was a global buffer where information would be inserted from one block and subsequently read by another block. A moving cursor (pointer) to the ring buffer was used to handle the changing communication patterns caused by the rotating interface.

This approach was impractical to implement in the parallel application since it was based on the assumption that the two blocks accessing the interface would share a common ring buffer resource. In a parallel implementation, a global ring buffer would cause a bottle–neck in the communication since this global buffer would need to be managed by a single processor. To circumvent this problem, blocks need to directly communicate to the appropriate block on the other side of the ring buffer interface instead of going through a global buffer "middle man". The semantic capturing this direct communication across the ring buffer interface has been named a ring buffer connection. In the ring buffer connection, a patch is given the angular velocity of the blocks it is connecting to and a circular list of addresses to patches that will be communicated with as the rotation of the interface progresses. With this information, a patch involved in a ring buffer communication can partition its information (as a function of time–step and angular velocities) and pass this information directly to the appropriate partners on the other side of the ring buffer interface. A schematic of this connection for a single patch is given in Figure 4.1. In order to handle symmetric solutions, a rotation operator is included with each patch address. The ring buffer connection is managed by dividing the given patch into sub–patches that will match the patch divisions on the other side of the ring buffer connection and then sending these sub–patches to the appropriate blocks. When a block receives this partial patch it also receives an offset which it uses to store the information in the appropriate locations in the computational space. In addition to q variable information, the ring buffer connection also communicates information that is used to compute a new grid for a region between the rotating interfaces. This communication uses the same basic procedures involved in the communication of the q variables.

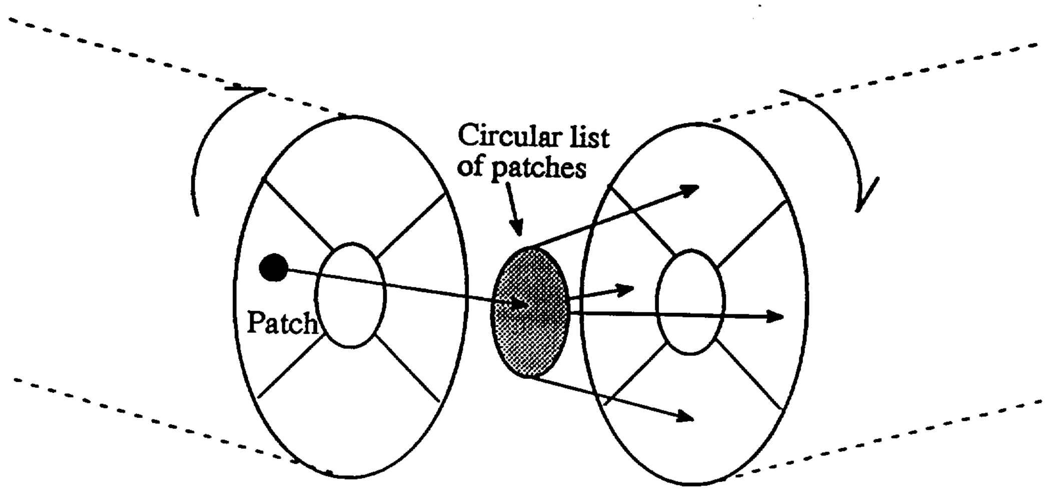

Figure 4.1 A Schematic of a Ring Buffer Connection for One Patch

The implementation of the ring buffer connection required complex procedures in order to calculate sub–patches from moving patch interfaces. Subroutines handling these connections were implemented in C and tested separately from the application so that high confidence in the implementation of this communication procedure could be achieved. The C language was chosen for this task over Fortran because of its advanced capabilities regarding dynamic memory management and complex data structures. The final parallel application was validated as correctly implementing the parallel communication semantics by performing an exact comparison of the results generated by the parallel implementation to those of the sequential implementation.

The result of the block oriented communication semantics implemented in the parallel turbomachinery application was an input file that is complex and filled with a large degree of redundant information. The complexity of this file made creating a corresponding input file for the parallel application a tedious process. This problem is addressed through the development of an automated domain decomposer that generates both a partitioned grid and input file for the parallel application.

The Automated Domain Decomposition Tool

The domain decomposition tool is responsible for partitioning the grid into sub–grids that will be used for parallel computation. Since the work required by a sub–domain is proportional to the number of cells in the domain, load balancing is achieved by making the number of cells in each sub–grid approximately equal. A key challenge of the domain decomposition tool was the development of an expression for boundary condition and connectivity information described by the original grid in such a way that this information could easily propagate to the decomposed grid. This information is captured in a file format that is based on a block structured language such as the C language. The input parser for this input file format was developed using the standard *lex* and *yacc* programming tools.

The input file defines its data by variable assignment and variable scoping. Variables in the input file can have four basic types: integer numbers, real numbers, character strings, and existential. The first three are well known types and need little description, whereas the existential variable type is a variable that is defined by its existence in the input file. This variable can be considered as a boolean type and is used to turn on various features of the decomposition tool.

The first variable scope of the input file is the global scope. This variable scope is used to define general parameters of the simulation and grid.

Table 4.1 contains a partial list of the general parameters used by the domain decomposition tool.

Table 4.1 Variables Accessible from the Global Variable Scope

Variable Name	Type	Variable Values
temporal_accuracy	STRING	"first–order" "three–point–backward"
spatial_accuracy	INTEGER	1 – 3
limiter	STRING	"minmod" "superbee" "van–leer"
number_of_cycles	INTEGER	1 – MAXINT
beta	REAL	$-2\pi - 2\pi$
flux_jacobian_update_frequency	INTEGER	1 – 2
number_of_refinement_iterations	INTEGER	1 – MAXINT
number_of_cycles_between_clicks	INTEGER	1 – MAXINT
grid_symmetry	INTEGER	1 – MAXINT
free_stream_mach_number	REAL	0.0 – 2.0

The READ_GRID command is used to read in the initial grid file for the simulation. This command is similar in format to a function call in the C language and is given two arguments: a file name, and the number of grid blocks in the file. The grid blocks that are read from the grid file are numbered as they occur in the file starting with one. The grid block numbering is used to identify the grid block when the boundary conditions and connectivity information is given.

The next variable scope used in the input file is the variable scope of the grid block. This variable scope is created by the BLOCK command. The BLOCK command has a syntax similar to a structure declaration in the C language in that it uses braces to delimit the context of the scope it creates. The arguments to the BLOCK command are the grid identifier created by the READ_GRID command and a name that will be assigned to the grid block for use in developing connections between grid blocks. The only variable currently used in the scope of the block command is the *divide_block* variable. The *divide_block* variable is used to tell the decomposer how many equally sized partitions in which to divide a grid block for parallel solution.

Within the scope of the BLOCK command several commands are available for describing boundary conditions associated with the grid. The first of these commands is the PATCH command. The PATCH command is used to create rectangular regions on the exterior surface of the computational space (representing surfaces of ξ, η, or ς = constant) represented by the geometry for the assignment of boundary and connectivity information. The PATCH command divides the exterior surfaces of the computational space into three classes of patches denoted by IJ_FACE, JK_FACE, and KI_FACE. Each PATCH command creates a pair of patches: one for the lowest index of the grid and one for the highest index of the grid. For example: when the grid points in a grid block are numbered by three indices such that i=[1,ni], j=[1,nj], k=[1,nk], a patch command that defines a pair of patches on the IJ_FACE defines patches for the exterior surfaces at k=1 and k=nk. The PATCH command assigns a name to the pair of patches it creates. The individual patches of the pair are denoted by following the name of the patch pair by a plus or a minus sign. The plus (minus) sign denotes that the patch on the high (low) index value of the grid will be used.

The TERMINAL command is also available within the scope of the BLOCK command. The TERMINAL command is used to assign lists of patches to terminal identifiers. Terminals represent end–points of communication and will be used to express boundary conditions and connections. The argument of the TERMINAL command is the name of the terminal that it creates. The TERMINAL command is followed by a list of patches enclosed in braces. When two terminals are connected to one another, the patches listed by the TERMINAL command will be connected in the order they appear.

The final command that is available to the scope of the BLOCK command is the TERMINAL_ATTRIBUTES command. This command is used to assign boundary conditions to terminals. The TERMINAL_ATTRIBUTES command creates a new variable scope for the terminal name that is given to the TERMINAL_ATTRIBUTES command. Two variables are recognized in the terminal variable scope by the domain decomposition application and they are listed in Table 4.2.

Table 4.2 Variables within the Scope of TERMINAL_ATTRIBUTES

Variable Name	Type	Variable Values
boundary_type	STRING	"impermeable" "inflow" "inflow–outflow" "outflow"
zero_area	EXISTENTIAL	None

The final variable scope created for the domain decomposition application is the variable scope created by the ZONE command. A zone is a collection of grid blocks which share the same angular velocity. Because all grid blocks share the same angular velocity within a zone, the connections between grid blocks within a zone are static. The only variable that the domain decomposition tool uses in the variable scope created by the ZONE command is the normalized angular velocity of the zone represented by the variable *dtdt*. In addition to the variable *dtdt*, there are several commands available within the scope of the ZONE command.

The first command within the scope of the ZONE command is the GRAB_BLOCK command. This command is used to assign a grid block to a zone.

The CONNECT command is used to create static connections between terminals defined in blocks assigned to the zone. The CONNECT command identifies two terminals that will communicate information during the simulation. A terminal is identified by the name of the grid block and the name of the terminal separated by a period. If the CONNECT command is followed by the keyword SYMMETRIC then momentum vectors communicated across this connection will be rotated in order to satisfy the geometric symmetry of the problem.

The final command available within the scope of the ZONE command is the RING_BUFFER command. Ring buffers are used to communicate information between zones since relative motion will be present in these connections. The RING_BUFFER command defines the computational dimension along which grid line shearing will occur as the zones rotate, the depth into the zone in which regridding will occur, and a list of terminals that will participate in the ring buffer interaction.

Finally, CONNECT commands in the global context are used to connect zones by connecting ring buffers defined in each zone. The syntax of this CONNECT command is the same as the CONNECT command of the zone except that no symmetric attribute may be applied and the arguments to the connect command are ring buffer identifiers rather than terminal identifiers.

An example input file for the unducted counter rotating prop fan problem studied in Chapter V is given in Appendix B.

CHAPTER V
RESULTS

The objective of this chapter is to present the results of numerical experiments aimed at answering two key questions: 1) is the parallelized algorithm a cost effective approach (compared to sequential solution) even though it requires extra computational effort brought on by additional Newton iterations, and 2) is the cost of additional Newton iterations controlled as larger problem sizes are considered.

Computing the Level of Convergence

The value of the Δq vectors computed during the solution process is a measure of how well the solution satisfies the discretized equations, where the form of the discretized equations depends upon the formulation. For example, when Newton iterations are not used the value of Δq represents how well the spatial numerical derivatives of the discretized equations are satisfied. This provides a good measure of the level of convergence for a steady state flow field since the temporal derivatives are zero in a steady state solution. On the other hand, the value of the Δq vectors computed during the Newton iteration process provide a measure of how well both the spatial and temporal numerical derivatives are satisfied, thus providing a measure of convergence for unsteady flow fields. Either set of these Δq vectors provide a local measurement of convergence for the discretized equations and can also provide a global measurement if an averaging process is applied to them. The averaging process applied to compute a global convergence level in this chapter is a root–mean–square average. This average favors a majority reduction in the magnitude of the Δq vectors. The value computed using this average is called the root–mean–square sum of the residual terms, or more simply the RMS residuals. For the parallel application, the RMS residuals are computed independently for the density, momentum, and energy terms of the Δq vector providing three independent measures of global convergence.

The Solution Process

In the sequential application, a typical simulation involves a two step process. The first step involves an impulsive start where the geometry of the simulation is thrust into a steady flow field of constant free stream Mach number. The simulation progresses through a complete turbomachine revolution in order to arrive at a solution that is relatively free of the transient conditions created from an impulsive start. Newton iterations are not used during the first revolution since high resolution answers of the transients caused by the impulsive start are not considered interesting. This first revolution can be considered as a means of approximating an initial condition for the simulation. After the first revolution, two additional revolutions are simulated using Newton iterations. The number of Newton iterations used during this stage of the simulation is the number required to reduce the RMS residuals by two orders of magnitude. The parallel application will employ the same procedure to obtain a solution.

Solution Results from the Parallel Application

The numerical experiments presented in this section were performed on a GE UDF8–8 counter–rotating unducted fan immersed in a Mach 0.7 flow field. Two cases are considered for this geometry: a zero angle of attack configuration and a configuration at ten degrees angle of attack. The zero degree angle of attack configuration can take advantage of symmetry to reduce the number of grid cells required by a factor of eight. These two configurations allow for the measurement of the scalability of the algorithm. As presented in chapter III, the zero angle of attack problem will be executed on larger numbers of processors until cost effectiveness is maximized. Then the problem will be scaled up by a factor of eight when the ten degree angle of attack case is considered. The cost effectiveness of the scaled–up case will be measured and if it remains relatively constant the algorithm will have demonstrated an ability to scale to larger configurations and remain cost effective thus demonstrating scalability.

An Intel iPSC/860 parallel computer comprised of 32 hypercube connected i860 processors with 8 megabytes of RAM on each processor was used to measure the parallel application for the zero angle of attack configuration. The number of processors applied to the zero angle of attack configuration was varied from 4 to 12 processors and the execution times required to complete three complete revolutions of the geometry was measured. At 6 processors enough memory was available on each node to allow flux Jacobian freezing to take place (flux Jacobian freezing means that the flux Jacobians are not re–computed after the second Newton iteration, but instead are stored and re-used). The savings in computations that flux Jacobian freezing produces is significant. Table 5.1 lists the execution times achieved as the number of processors applied to the problem increases.

The cost effectiveness for each case considered in Table 5.1 is computed by applying the equation

$$Cost\ Effectiveness = \frac{Problem\ Size}{N\left(t_{execution}\right)^2}. \tag{5.1}$$

This expression of cost effectiveness is equivalent to equation (3.9) when *Work* is defined in terms of problem size. Observation of the cost effectiveness for the Intel iPSC/860 simulations shown in Table 5.1 demonstrates that optimal cost effectiveness is achieved when the problem is decomposed for 10 processors.

When the angle of attack configuration is considered, the problem size grows by a factor of eight. If the maximum cost effectiveness occurs at 10 processors, it is expected that a maximum cost effectiveness for the angle of attack case will require 80 processors. Since only 32 processors are available on the Intel iPSC/860 parallel computer used for this experiment, another parallel computer was used for the cost effectiveness measurement of the angle of attack configuration. The Intel Delta parallel computer at the California Institute of Technology was used for this simulation. The Intel Delta parallel computer is comprised of roughly 512 mesh connected i860 processors with 16 megabytes of RAM per processor. The zero angle of attack configuration execution time on the Intel Delta for a 10 processor decomposition was measured as a reference point. The Intel Delta provided a modest improvement in execution time (over that of the iPSC/860) that can be attributed to differences in compiler and communication network performances.

The 80 processor simulation of the angle of attack configuration required a slightly longer execution time than the 10 processor simulation of the zero angle of attack case. This caused a slight drop in the cost effectiveness for the larger problem size. Since no additional Newton iterations were required by the angle of attack case, the increase in execution time can be attributed to communication costs. Since no attempts were made to optimize communication costs by appropriately mapping the problem decomposition to processors, it is expected that the execution time of the larger problem size will be reduced when the placement of blocks on processors is optimized.

Table 5.1 Execution Time Results of the Parallel Application

Processors	Newton Iterations	Jacobian	Execution Time	Problem Size	Cost Effectiveness
4 ipsc	6		5.56 hrs	23520 cells	188.17
6 ipsc	6		3.83 hrs	23520 cells	361.66
6 ipsc	6	freezing	2.96 hrs	23520 cells	447.41
8 ipsc	7	freezing	2.54 hrs	23520 cells	455.70
10 ipsc	7	freezing	2.17 hrs	23520 cells	499.48
12 ipsc	8	freezing	2.06 hrs	23520 cells	461.87
10 delta	7	freezing	2.09 hrs	23520 cells	538.45
80 delta	7	freezing	2.18 hrs	188160 cells	494.91

Convergence Results of the Parallel Application

Since the modified algorithm introduces additional errors into the solution algorithm, it is important to compare the effects these errors have on convergence to the fully coupled sequential algorithm. For these comparisons the RMS residuals that result from the sequential application using five Newton iterations in the final two revolutions is considered. Figure 5.1 shows the RMS residuals for the density component of the Δq vector for the sequential algorithm and the parallel algorithm for both angle of attack cases during the first prop revolution. The 10 block case is at zero angle of attack and the 80 block case is at 10 degrees angle of attack. The RMS residuals for the energy component are shown in Figure 5.2. It is apparent from these plots that the parallel algorithm was less capable of converging the energy equation than the density equation. Since the energy component of the Δq vector is larger on average than any other term, the conclusion can be drawn that the error created by forcing Δq to zero at the block boundary will inject the greatest error in the energy equation. This problem may be lessened by normalizing the equations in a way that balances the magnitudes of the components of the Δq vector.

The RMS residuals for the final Newton iteration during the last two prop revolutions are given in Figure 5.3 and Figure 5.4. These results show that the parallel algorithm performed better than the sequential at converging the density equations, but was less capable of converging the energy equation. The effects of differences in convergence rates of the different residual terms requires further study.

In order to verify that the parallel application converges to the correct solution, the pressure coefficients along the base and tip of the blades in the prop fan were considered. For comparison, the results at the 480th time–step of the 10 processor parallel solution were compared to the results of the sequential application. The results of these comparisons are shown in Figures

5.5 through 5.8. These comparisons demonstrate that the parallel application results are essentially equivalent to the sequential results.

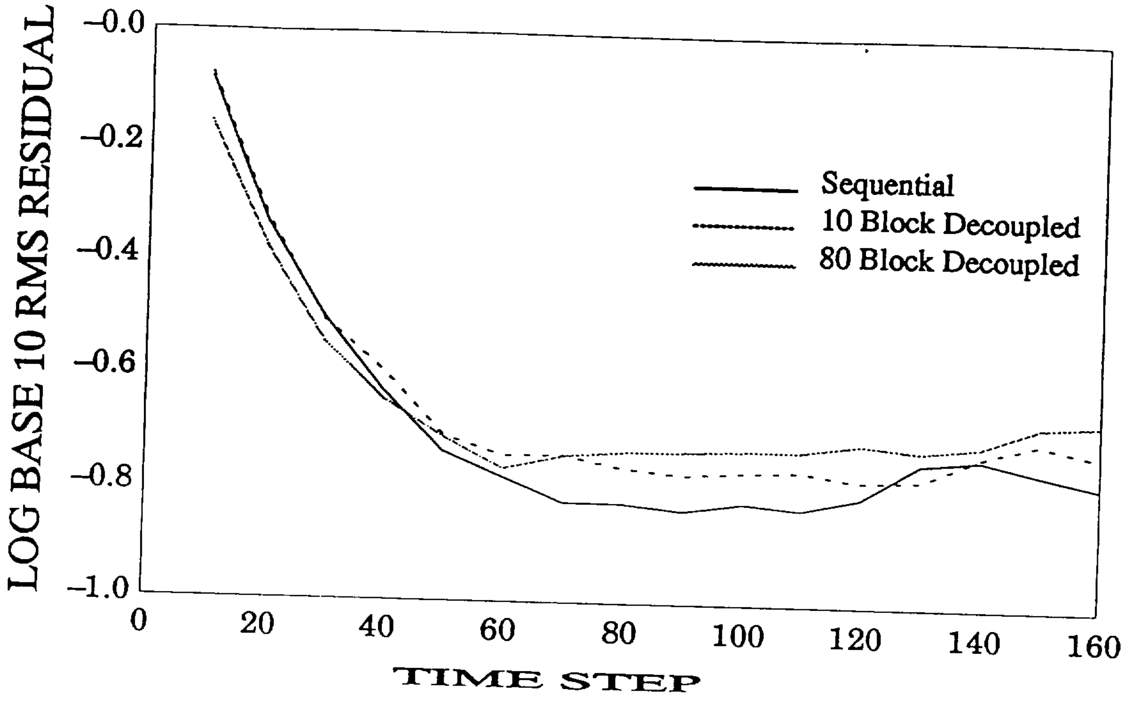

Figure 5.1 Density Residuals During the First Revolution

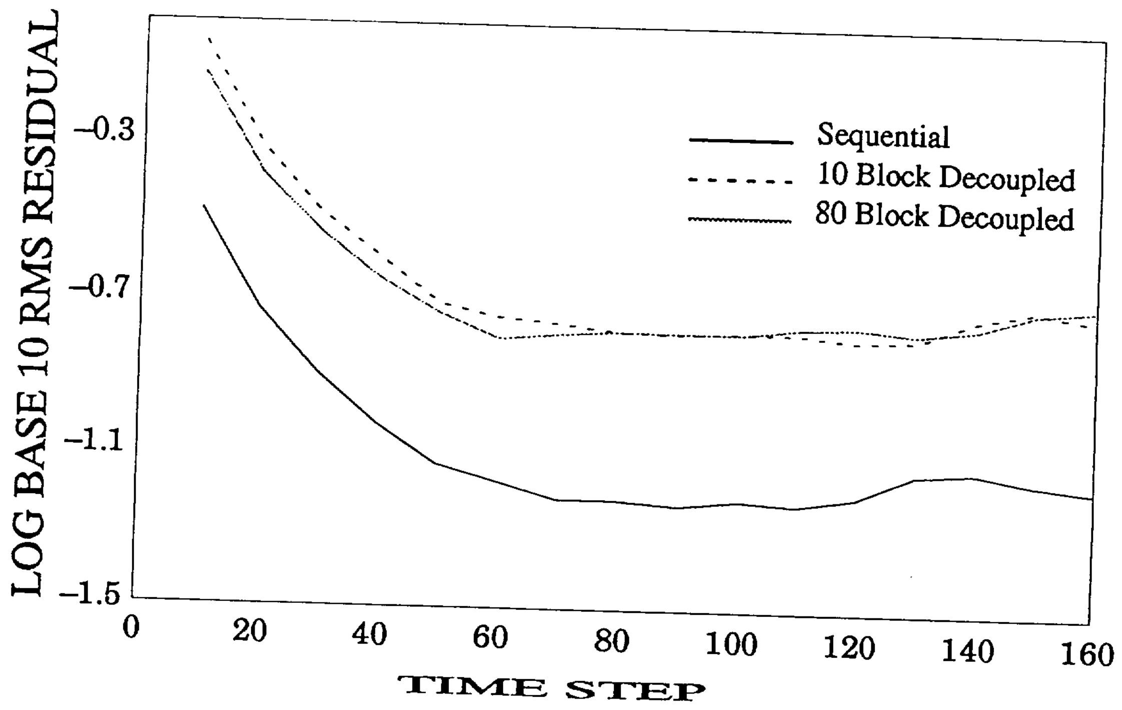

Figure 5.2 Energy Residuals During the First Revolution

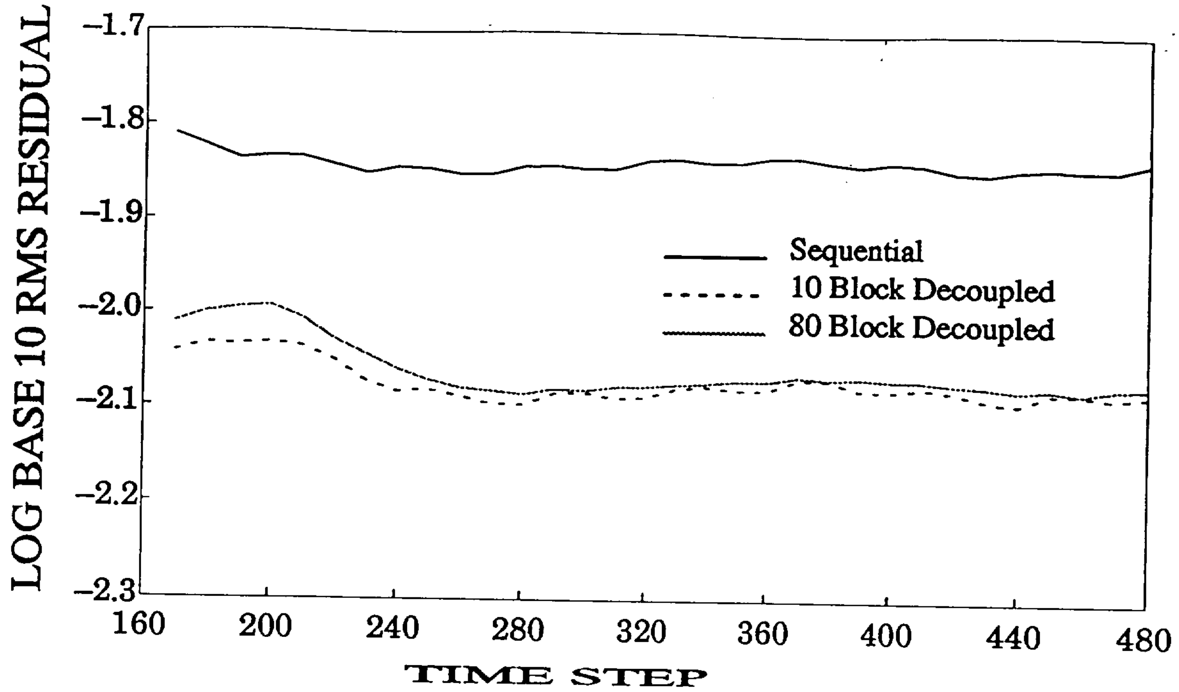

Figure 5.3 Density Residuals at the Final Newton Iteration

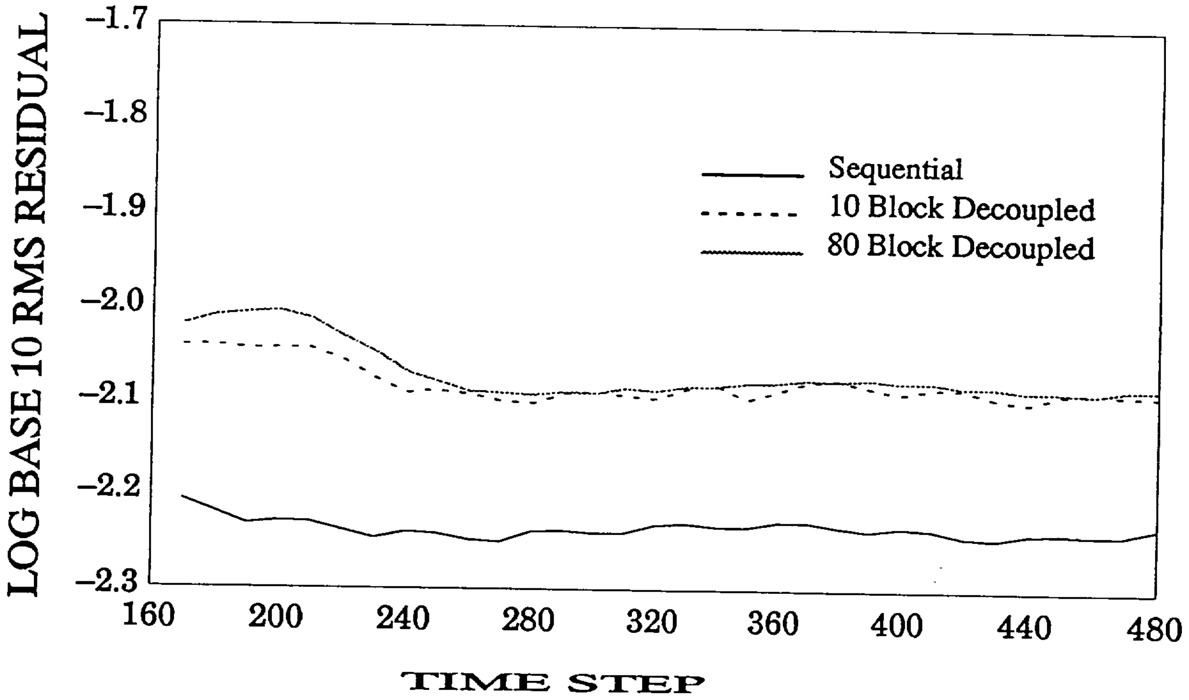

Figure 5.4 Energy Residuals at the Final Newton Iteration

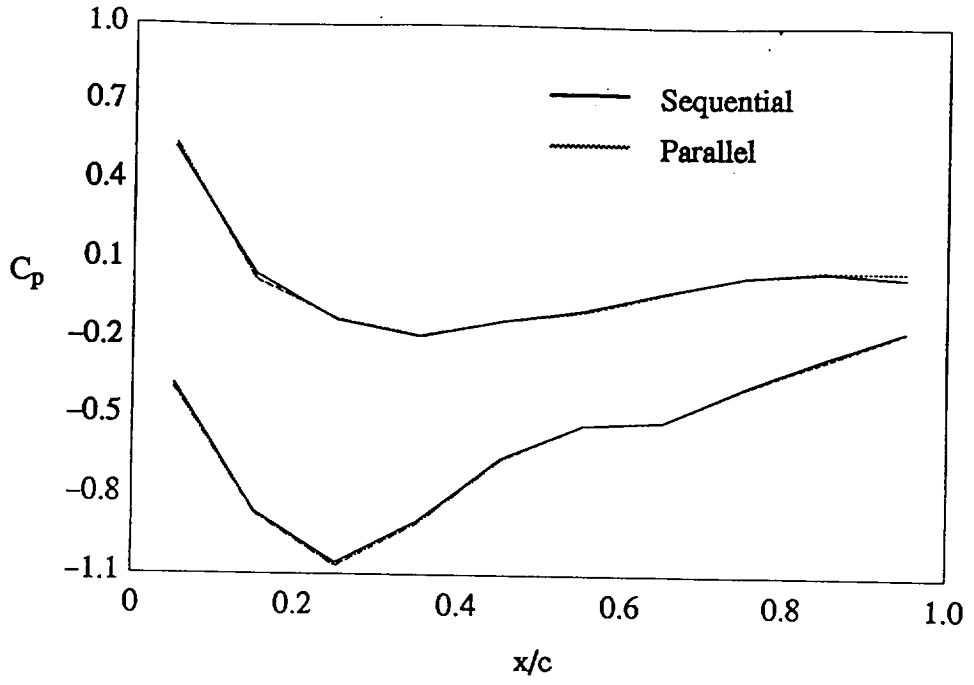

Figure 5.5 Pressure Coefficient Comparison at the Base of the Fore Blade

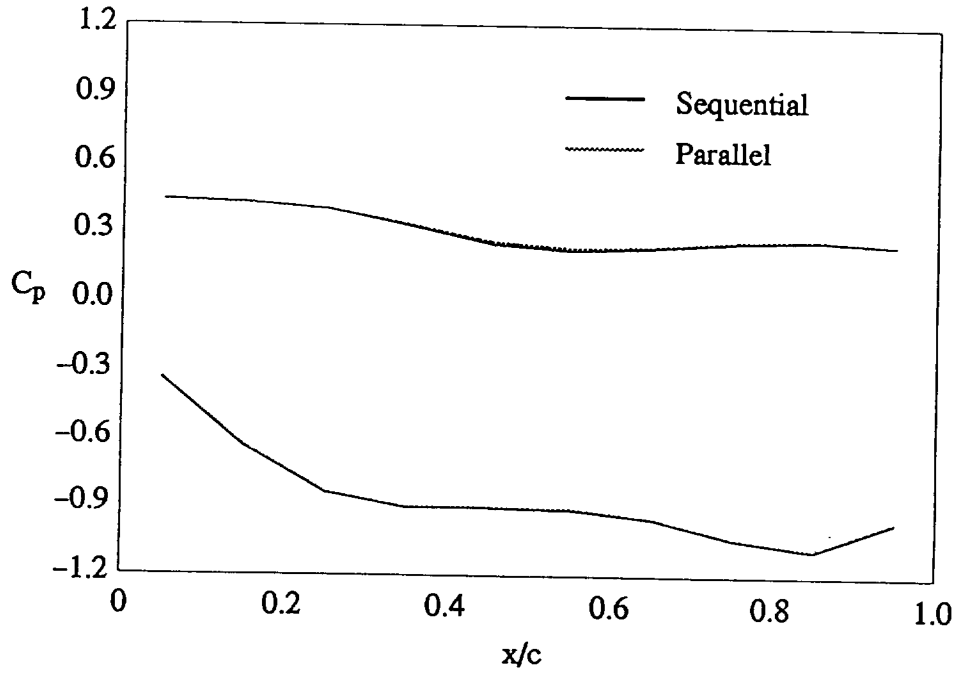

Figure 5.6 Pressure Coefficient Comparison at the Tip of the Fore Blade

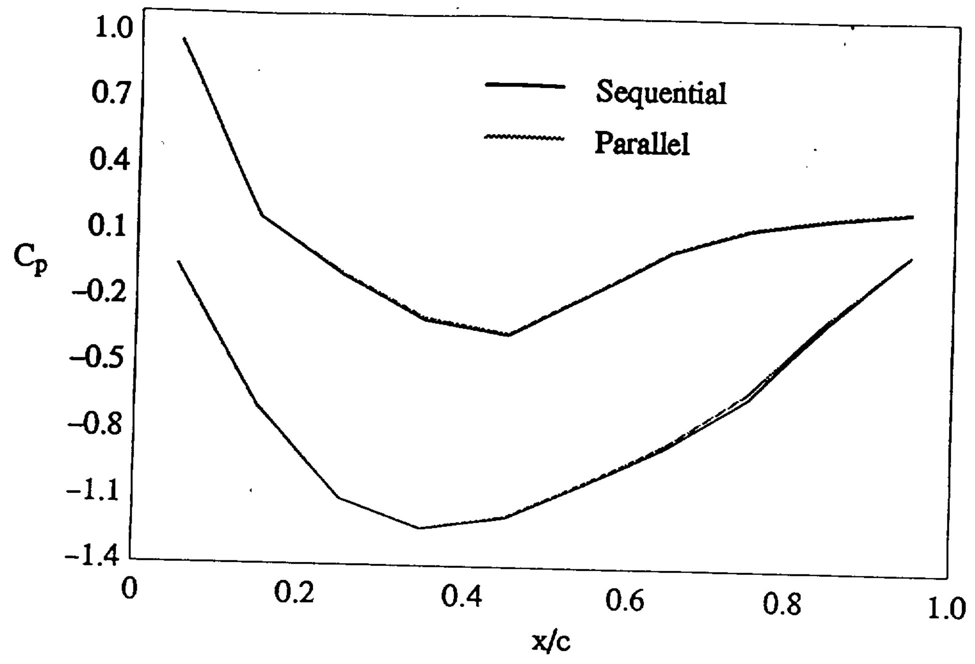

Figure 5.7 Pressure Coefficient Comparison at the Base of the Aft Blade

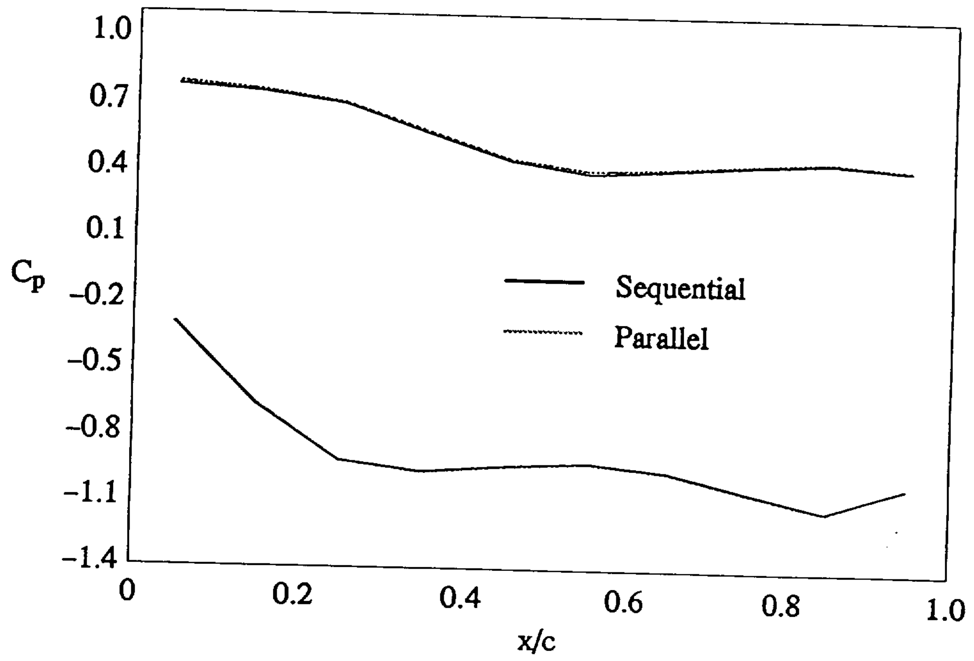

Figure 5.8 Pressure Coefficient Comparison at the Tip of the Aft Blade

CHAPTER VI
CONCLUSIONS

Two parallel algorithms, the parallel approximate factorization algorithm and the modified algorithm, have been considered as candidates for massively parallel multicomputer systems. Arguments have been made that the parallel approximate factorization algorithm is a poor choice for massively parallel multicomputer systems based on cost effectiveness. Numerical experiments have demonstrated that the modified algorithm is a cost effective means of gaining high performance on massively parallel multicomputer systems.

Observations of the constraints placed on parallel algorithm development can also be made based on cost effectiveness analysis and experiences related to the development of the parallel turbomachinery application. Since optimal sequential algorithms are based on minimizing instructions and optimal parallel algorithms are based on a balance between minimizing instructions and maximizing parallel utilization it is reasonable to assert that the optimal parallel algorithm is not identical to the optimal sequential algorithm. This observation was also made by Chan [23] where he argued that the set of efficient kernels presumed when scientific applications are developed must change when parallel implementations are considered. The cost effectiveness analysis also suggests that the optimal general algorithm will be polymorphic in that it will change form based on problem size and the number of processors available. This suggests that the structural complexity of optimal parallel algorithms will be greater than optimal sequential algorithms. The structural complexity of the parallel algorithm may also be increased due to limitations that the parallel architecture places on parallel algorithms as demonstrated in the implementation of ring buffer communications for the parallel turbomachinery application. If parallel algorithms are necessarily structurally more complex, the choice of Fortran as a base language for parallel algorithms must be called into question. Clearly, object–oriented paradigms designed to implement algorithms of high structural complexity must be considered more appropriate for parallel algorithm development in the future.

Before the modified algorithm can be considered as an appropriate parallel algorithm several additional issues must be investigated. The robustness of the modified algorithm for extremely complex geometries needs to be demonstrated. Investigation into the potential for violation of conservation law principles in the modified algorithm is suggested. Alternate iterative methods such as successive over relaxation (SOR), preconditioned iterative methods, and convergence accelerators such as multigrid methods should be compared to the modified algorithm in a cost effectiveness study. In addition, scalability of the modified algorithm for grids of finer resolution should be studied further. Adaptive methods where the either the grid or the domain decomposition is adapted in order to reduce iterations required by the modified algorithm is also suggested.

APPENDIX A

PARALLEL EMULATOR PROGRAM

```
#include <stdio.h>
#include <math.h>

/*
  Emulation program for LU decomposition based CFD algorithm.
  Emulates execution on a zero message latency parallel machine with
  variable number of processors.  This emulator only estimates
  relative execution times based on the parallelism profile of the
  algorithm.
  */

int parallelism_profile[100000] ;

/*
compute_parallelism_profile fills the array parallelism_profile with
the parallelism profile of the forward and backward substitution passes
of the LU decomposition solver.  The parallelism profile is determined
from the fact that all cells that lie on the plane {i+j+k = constant}
can be executed concurrently.  This function returns the number of
steps contained in the parallelism profile.  The arguments to this
function are the size of the computational space in i, j, and k
dimensions.
 */

int compute_parallelism_profile(ni,nj,nk)
int ni,nj,nk ;
{
    int i,j,k,profile_size ;

    profile_size = ni+nj+nk-2 ;
    bzero(parallelism_profile,profile_size*
                               sizeof(*parallelism_profile)) ;

    for(i=0;i<ni;i++)
      for(j=0;j<nj;j++)
        for(k=0;k<nk;k++)
          parallelism_profile[i+j+k]++ ;
    return profile_size ;
}
```

```
/*
compute_time computes the execution time for a problem with dimensions
ni,nj,nk when executed on N processors.  It computes the total time
including the time required to set up the LU decomposition problem.
The amount of time required to setup the LU decomposition is determined
from the measurement of 60% of execution time spent on this operation
during sequential execution. 20% is spent on the forward pass, and
20% is spent on the backward pass making a total execution time of 1
for a problem size of 1 cell.
 */

double compute_time(ni,nj,nk,N)
int ni,nj,nk,N ;
{
    double total_time = 0.0 ;
    int profile_size,i ;
    double SETUP_TIME_PER_CELL = 0.6 ;
    double SUBSTITUTION_TIME_PER_CELL = 0.2 ;


    total_time += SETUP_TIME_PER_CELL*ceil(((double)((ni)*(nj)*(nk)))/
                                            ((double)N)) ;

    profile_size = compute_parallelism_profile(ni,nj,nk) ;

    for(i=0;i<profile_size;i++)
      total_time += 2*SUBSTITUTION_TIME_PER_CELL*
        ceil(((double)parallelism_profile[i])/((double)N)) ;
    return total_time ;
}
```

```
/*
  We are now going to use our compute_time emulator to determine the
  sizeup metric for this parallel algorithm.  This is accomplished by
  first computing the time to execute the program with 1 processor
  on a given problem size.  Then we will compute the time required to
  solve a larger problem on N processors.  If we solve the problem in
  less time we will increase the size of the problem and try again.
  When we have found the size of the problem that can execute in the
  same amount of time, we will increment N and repeat the process.
  */
main()
{
    int n[3],I=0,J=1,K=2 ;
    double sequential_time,parallel_time ;
    int N = 1,sequential_size,i ;

/* The n array contains the size of the sequential reference process */
    n[I] = 3 ;
    n[J] = 3 ;
    n[K] = 3 ;
    /* sequential_size is used to normalize sizeup numbers */
    sequential_size = n[I]*n[J]*n[K] ;
    sequential_time = compute_time(n[I],n[J],n[K],N) ;
    printf("sequential time = %f\n",sequential_time) ;
    for(N=2,i=0;N<=4000;N=N+N/25+1) {
        do {
            parallel_time = compute_time(n[I],n[J],n[K],N) ;
            if(parallel_time < sequential_time) {
                n[i]++ ;
                i = (i+1)%3 ;
            }
        } while(parallel_time < sequential_time) ;
        printf("%d %f\n",N,
               ((float)n[I]*n[J]*n[K])/(float)sequential_size) ;
    }
    exit(0) ;
}
```

APPENDIX B

EXAMPLE INPUT FILE FOR THE DOMAIN DECOMPOSITION TOOL

```
simulation_name = "Counter-Rotating Prop Fan" ;

// temporal accuracy can be:
//    first-order
//    three-point-backward
//    trapezoidal-time-differencing    (Not recommended)
temporal_accuracy = "three-point-backward" ;
spatial_accuracy = 3 ;

// limiters:
//    minmod      (use with second and third order)
//    superbee    (use only with second order)
//    van-leer
limiter = "van-leer" ;

number_of_cycles = 480 ;

beta = 0.0 ;

// If 1 the flux jacobians will be computed each newton iteration
// if not 1 flux jacobians will not be computed after 2nd newton
//   iteration, this saves time but uses extra memory
flux_jacobian_update_frequency = 2 ;
residual_print_frequency = 1 ;

number_of_refinement_iterations = 7 ;

initial_cycles_of_zero_pressure_gradient_BCs = 20 ;
initial_cycles_of_first_order_calculations = 0 ;
initial_cycles_of_entropy_violation_dissipation = 9999 ;

free_stream_mach_number = 0.7 ;

number_of_cycles_between_clicks = 1 ;

grid_symmetry = 8 ;

READ_GRID("grid_2-block",2) ;
```

```
BLOCK(1,front_blades) {
    divide_block = 5 ;
    PATCH(x1,IJ_FACE,1,42,1,22) ;
    PATCH(x2,IJ_FACE,42,52,1,11) ;
    PATCH(x3,IJ_FACE,42,52,11,22) ;
    PATCH(x4,IJ_FACE,52,57,1,22) ;
    PATCH(p2,JK_FACE,1,22,1,11) ;
    PATCH(p3,KI_FACE,1,11,1,8) ;
    PATCH(p4,KI_FACE,1,11,8,57) ;

    TERMINAL(hub) {p4-} ;
    TERMINAL(blades) {x2-,x2+} ;
    TERMINAL(sting) {p3-} ;
    TERMINAL(exterior) {p3+,p4+} ;
    TERMINAL(front) {p2-} ;
    TERMINAL(rear) {p2+} ;
    TERMINAL(kplus) {x1+,x3+,x4+} ;
    TERMINAL(kminus) {x1-,x3-,x4-} ;

    TERMINAL_ATTRIBUTES(hub) {
        boundary_type = "impermeable" ;
    } ;
    TERMINAL_ATTRIBUTES(blades) {
        boundary_type = "impermeable" ;
    } ;
    TERMINAL_ATTRIBUTES(sting) {
        zero_area ;
    } ;
    TERMINAL_ATTRIBUTES(exterior) {
        boundary_type = "inflow-outflow" ;
    } ;
    TERMINAL_ATTRIBUTES(front) {
        boundary_type = "inflow" ;
    } ;
} ;
```

```
BLOCK(2,rear_blades) {
    divide_block = 5 ;
    PATCH(p1,IJ_FACE,1,10,1,22) ;
    PATCH(p2,IJ_FACE,10,20,1,11) ;
    PATCH(p3,IJ_FACE,10,20,11,22) ;
    PATCH(p4,IJ_FACE,20,57,1,22) ;
    PATCH(p5,JK_FACE,1,22,1,11) ;
    PATCH(p6,KI_FACE,1,11,1,57) ;

    TERMINAL(hub) {p6-} ;
    TERMINAL(exterior) {p6+} ;
    TERMINAL(front) {p5-} ;
    TERMINAL(rear) {p5+} ;
    TERMINAL(kplus) {p1+,p3+,p4+} ;
    TERMINAL(kminus) {p1-,p3-,p4-} ;
    TERMINAL(blades) {p2+,p2-} ;
    TERMINAL_ATTRIBUTES(hub) {
        boundary_type = "impermeable" ;
    } ;
    TERMINAL_ATTRIBUTES(blades) {
        boundary_type = "impermeable" ;
    } ;
    TERMINAL_ATTRIBUTES(exterior) {
        boundary_type = "inflow-outflow" ;
    } ;
    TERMINAL_ATTRIBUTES(rear) {
       boundary_type = "outflow" ;
    } ;
} ;
```

```
ZONE(front) {
    dtdt = -1.620 ;

    GRAB_BLOCK(front_blades) ;

    CONNECT(front_blades.kplus,front_blades.kminus):SYMMETRIC ;
    RING_BUFFER(ring_buffer,JK_FACE,KDIM,05) {front_blades.rear} ;
} ;

ZONE(rear) {
    dtdt = 1.620 ;

    GRAB_BLOCK(rear_blades) ;

    CONNECT(rear_blades.kplus,rear_blades.kminus):SYMMETRIC ;
    RING_BUFFER(ring_buffer,JK_FACE,KDIM,09) {rear_blades.front} ;
} ;

CONNECT(front.ring_buffer,rear.ring_buffer) ;
```

REFERENCES

[1] Janus, J.M., Advanced 3–D CFD Algorithm For Turbomachinery, Ph.D. Dissertation, Mississippi State University, May 1989.

[2] Beam, R.M. and Warming, R. F., "An Implicit Factored Scheme for the Compressible Navier–Stokes Equations," AIAA Journal, Vol. 16, No. 4, pp. 393–402, April 1978.

[3] Steger, J.L. and Warming, R.F., "Flux Vector Splitting of the Inviscid Gasdynamic Equations with Applications to Finite Difference Methods," Journal of Computational Physics, Vol. 40 (1981), pp. 263–293.

[4] Janus, J.M., The Development of a Three–Dimensional Split Flux Vector Euler Solver with Dynamic Grid Applications, M.S. Thesis, Mississippi State University, August 1984.

[5] Whitfield, D.L. and Janus, J.M., "Three–Dimensional Unsteady Euler Equations Solution Using Flux Vector Splitting," AIAA Paper No. 84–1552, June 1984.

[6] Roe, P.L., "Approximate Riemann Solvers, Parameter Vectors, and Difference Schemes," Journal of Computational Physics, Vol. 43 (1981), pp. 257–372.

[7] Whitfield, D.L., Janus, J.M., and Simpson, L.B., "Implicit FInite Volume High Resolution Wave–Split Scheme for Solving the Unsteady Three–Dimensional Euler and Navier–Stokes Equations on Stationary or Dynamic Grids," Mississippi State Engineering and Industrial Research Station Report No. MSSU–EIRS–ASE–88–2, Febuary 1988.

[8] Anderson, W.K., Thomas, J.L., and Whitfield, D.L., "Multigrid Acceleration of the Flux Split Euler Equations," AIAA Paper 86–0274.

[9] Anderson, W.K., Implicit Multigrid Algorithms For The Three–Dimensional Flux Split Euler Equations, Ph.D. Dissertation, Mississippi State University, August 1986.

[10] Whitfield, D.L., "Newton–Relaxation Schemes for Nonlinear Hyperbolic Systems," Mississippi State Engineering and Industrial Research Station Report No. MSSU–EIRS–ASE–90–3, October 1990.

[11] Janus, J.M., and Whitfield, D.L., "Advanced 3–D Viscous SSME Turbine Rotor Stator CFD Algorithms," NASA CR–178997, September 1986.

[12] Belk, D.M., Three–Dimensional Euler Equations Solutions on Dynamic Blocked Grids, Ph.D. Dissertation, Mississippi State University, August 1986.

[13] Amdahl, G., "Validity of the single–processor approach to achieving large scale computing capabilities," Proceedings of AFIPS Conference (1967), pp. 483–485.

[14] Hockney, R.W., "Characterizing computers and optimizing the FACR(1) poisson–solver on parallel unicomputers," IEEE Transactions on Computers, c.32 (10) (1983), pp. 933–941.

[15] Barton, M. and G. Withers, "Computing performance as a function of the speed, quantity, and cost of the processors," Proceedings of Supercomputing '89 (1989), pp. 759–764.

[16] Gustafson, J., "Reevaluating Amdahl's law," Communications of the ACM, Volume 31 (May 1988), pp. 432–533.

[17] Sun, X. and Gustafson, J., "Toward a better parallel performance metric," Parallel Computing, Volume 17 (1991,) pp. 1093–1109.

[18] Welch, W., "Message–Driven Solver for Euler Fluid Dynamics Equations," Proceedings of the ACM 26th Annual Southeast Regional Conference (April 1988), pp. 29–34.

[19] Welch, W., "Zero Overhead Message Passing on the MADEM," Proceedings of the ACM 26th Annual Southeast Regional Conference (April 1988), pp. 45–50.

[20] Luke, E.A., "Memory Constraints on Scaling Multicomputers," Proceedings of the ACM 27th Annual Southeast Regional Conference (April 1989), pp. 567–570.

[21] Sterling, T.L. and Laprade, K.C., "The Impact of Overhead on the Scalability of Multiprocessors for Parallel Processing," Proceedings of the ACM 26th Annual Southeast Regional Conference (April 1988), pp. 138–145.

[22] Reese, D. and Luke, E.A, "Object–Oriented Fortran for Development of Portable Parallel Programs," Proceedings of the Third IEEE Symposium on Parallel and Distributed Processing, (December 1991), pp. 608–615.

[23] Chan, T.F., "Hierarchical Algorithms and Architectures for Parallel Scientific Computing," Computer Architecture News, Volume 18, Number 3 (September 1990), pp. 318–329.