8-1-2002

Design and Implementation of a Multi-Block Parallel Algorithm for Solving Navier-Stokes Equations on Structured Grids

Nirmal Tatavalli Mittadar

DESIGN AND IMPLEMENTATION OF A MULTI-BLOCK PARALLEL ALGORITHM FOR SOLVING NAVIER-STOKES EQUATIONS ON STRUCTURED GRIDS

By

Nirmal Tatavalli Mittadar

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

August 2002

DESIGN AND IMPLEMENTATION OF A MULTI-BLOCK PARALLEL ALGORITHM FOR SOLVING NAVIER-STOKES EQUATIONS ON STRUCTURED GRIDS

By

Nirmal Tatavalli Mittadar

Approved:

Mark Janus Associate Professor of Aerospace Engineering (Major Advisor)	Edward Luke Assistant Professor of Computer Science (Committee Member)
Clarence Burg Assistant Professor of Computational Engineering (Committee Member)	
Boyd Gatlin Associate Professor of Aerospace Engineering and Graduate Coordinator	A. Wayne Bennett Dean of the College of Engineering

Name: Nirmal Tatavalli Mittadar

Date of Degree: August 3, 2002

Institution: Mississippi State University

Major Field: Computational Engineering

Major Professor: Dr. Mark Janus

Title of Study: DESIGN AND IMPLEMENTATION OF A MULTI-BLOCK PARALLEL ALGORITHM FOR SOLVING NAVIER-STOKES EQUATIONS ON STRUCTURED GRIDS

Pages in Study: 83

Candidate for Degree of Master of Science

A coarse-grain parallel multi-block algorithm was designed for CHEQNS - a multi-block solver for solving chemically reacting flows in local chemical equilibrium and has been implemented using the Message Passing Interface (MPI). The parallel implementation confirms to the Single Program Multiple Data (SPMD) model.

The parallel implementation uses synchronous update of fluxes across the block-block boundaries. The solution algorithm consists of block-decoupled Gauss-Seidel iterations. The coupling between the sub-domains on different processors occurs at the Newton iteration level. The parallel implementation is general and can accept an arbitrary arrangement of blocks in multi-block configuration with multiple blocks per processor.

The parallel implementation has been verified against the results from the sequential multi-block solver for different types of flows. The parallel

performance has been studied in terms of speed-up and efficiency. The influence of parallelization on the convergence was also studied.

DEDICATION

To my parents, my brother and my grandmothers.

ACKNOWLEDGMENTS

I am extremely grateful to my major professor Dr. Mark Janus for the valuable guidance and moral support that he has offered during the course of my thesis work. I would like to thank my committee members Dr. Edward Luke and Dr. Clarence Burg for serving on my graduate committee and for the valuable knowledge they have shared with me during course work. I would like to thank Dr. Edward Luke for his help with the parallel implementation and for his thesis template which I used in the formatting of this thesis document.

I would like to thank Dr. Robert Webster for his invaluable assistance in the present work. He has been a constant source of encouragement. I would like to thank Dr. Sreenivas Kidambi for his help and suggestions with different parallel issues.

I express my gratitude to Dr. Boyd Gatlin and the Engineering Research Center for providing facilities for the present work. I would like to thank Ms. Nannette for keeping my forms straight and Ms. Nancy Covington and Ms. Nita Hartness for their help. I would like to thank the Systems Administration people for offering timely help when it was required.

I would like to thank Mr. Victor Whitehead and Mr. Pete Huseman from Lockheed-Martin Astronautics for the financial support.

I extend special thanks to Vasanth, Kishore, Prashanth, Moin and Ravi for their help and for creating a home away from home. I extend my hearty thanks to all my friends here at MSU for their moral support.

Finally, I thank my parents and family members back home for their constant encouragement and support.

TABLE OF CONTENTS

APPENDIX Page

LIST OF FIGURES

FIGURE Page

FIGURE Page

FIGURE Page

CHAPTER I

INTRODUCTION

1.1 Sequential Limitations & Parallel Processing

Parallel processing is applicable to problems whose solution algorithm can be logically broken down into multiple steps such that within each step there exists sub-problems which are independent and can be solved simultaneously. Parallel processing is the simultaneous processing of such sub-problems either on different microprocessors as by a single computer with more than one central processing unit or by multiple computers connected together in a network. The relevance of parallel computing becomes apparent when we look at the limitations of the sequential methodology. There is an upper bound on the maximum processor speed that can be attained on a single processor. Clock speeds on workstations have increased in the past few years to 1 GHz. According to Moore's law, processor speed and memory increase by a factor of two every one and half years which is a factor of 1000 in 15 years. Computer clocks are approaching fundamental speed limits, the speed of light, which is the upper bound for the speed of signal transmission. The limitations of the *Moore's Law* would be reached in a decade as the limits of CMOS integrated circuits are reached. Higher performance can only be achieved by concurrent execution of programs on duplicated hardware - either on a single processor or using a multiple processor machine. Another important issue is the growing memory requirement as the problem size grows. The bigger

problem might not fit the memory of a single CPU. The traditional supercomputers thus have the following limitations,

- Single high performance processors are extremely expensive
- Significant cooling requirements are necessary
- Single processor performance is reaching its asymptotic limits

In these regards parallel computing offers a powerful alternative. The cost of a single CPU increases, at a rate much higher than linear, with increasing speed. However, with parallel processing this variation could be made near linear. Message passing parallel computers can be developed at a low cost and in a short time using off the shelf components and processors. Memory requirements can be satisfied by distributing the problem over many computer nodes. There are three goals of parallel processing. The primary objective of parallel processing is to reduce the time required to solve the problem. The next objective is to be able to solve problems of larger size in terms of time and memory. The third objective is to reduce cost associated with super-computing speeds by using cheap resources like building a cluster of work-stations as a parallel computer. However, one should also look at the difficulties associated with parallel computing. Designing efficient parallel algorithms usually requires lot of thinking. There is a high cost associated with program development.

1.2 Parallel Processing & Computational Science

Computational Science involves the simulation of complex physical, chemical, biological and engineering phenomena in complex geometries based on numerical techniques applied to the equations, typically partial differential

equations, describing those phenomena. In particular, Computational Fluid Dynamics (CFD) methods are generally based on the solution of a set of partial differential equations which describe the continuum behavior of the fluid; Euler equations for the inviscid flow and Navier-Stokes equations for the viscous flow. These differential equations are discretized using numerical techniques, like the finite element methods or finite volume methods, and the resulting set of algebraic equations is solved on a computational grid. The thrust for accurate representation of the physical phenomena often leads either to the implementation of complicated models or increasing the resolution by using very fine grids. The large-scale computations, indicated by this increase in problem size, are computer resource intensive and involve handling of sizeable amount of data necessary to describe the flow field. The limitations of the traditional supercomputers and the computationally intensive nature of the flow simulations have resulted in a shift towards the efficient use of high-performance parallel computers.

Parallel paradigm in CFD involves solving the flow simultaneously in different sub-domains or grid-points. Domain decomposition is employed in CFD either as a means to grid complex geometries or as a source of parallelism. If domain decomposition is used for the former reason then the domain decomposition step would be present in the sequential algorithm also and in this case one can expect significant speed-ups after parallelizing. However, if the motivation for domain decomposition is to efficiently use parallel computing resources, the speed-ups observed may not be as high as in the previous case. Parallel techniques used in CFD fall into two categories :

- *Grid point level*: The computations for each grid point or grid cell is performed on different virtual processor. This is a data parallel approach in

which different processors undertake the necessary computations of different grid points in a synchronous manner. The data-parallel approach has been used by earlier researchers on massively-parallel, SIMD computers.

- *Sub-domain level*: A control parallel methodology with the computational problem being divided into a number of almost independent tasks, with each processor performing a different task. This coarse-grain parallelism is implemented on shared-memory multi-processors or distributed memory MIMD computers.

In the present work a coarse-grain parallel algorithm is considered.

The following paragraphs describe related work by previous researchers. These illustrate the different perspectives in the application of parallel paradigm to CFD. Two prominent perspectives are performance in terms of speedup and efficiency and performance in terms of convergence.

In reference [6], the authors have examined four different parallel models of the multi-block method with a time dependent Euler Solver on Cray-T3D system. They considered (i) *data parallel* (ii) *message passing* (iii) *work sharing* and (iv) *explicit shared memory*. Data parallel implementation consisted of treating the blocks in a sequential manner with the solution in each block being computed using fine-grain grid-point level parallelism. They implemented message passing model using PVM. Work sharing was implemented using private local arrays and implicit communication. The explicit shared memory model was implemented using SHMEM library which consists of manufacturer specific functions for transfer of data from local address to remote address. They observed that distribution of data and work amongst the processing elements (PEs), exchange of data between PEs, and the synchronization between the PEs are important for high performance

on a distributed memory parallel system. They concluded that the single PE performance is the performance bottleneck and not inter-processor communication, when CFD computations are based on block structured grids.

The factors affecting the efficiency in the case of multi-block formulation for solving Navier-Stokes equation has been described in [4]. They formulate communication inefficiency in terms of the total number of grid points, the number of processors and the number of floating point operations per grid point per iteration. Their algorithm consisted of loosely synchronous cycles of computation followed by data exchange across the faces of the blocks. They conclude that a complex practical calculation, such as the accurate solution of the Navier-Stokes equations involving a complex geometry, can be expected to run on a massively parallel machine with high efficiency.

A parallel solver was developed by Pankajakshan, et. al. [5] for three-dimensional unsteady incompressible viscous flow using multi-block structured grids and using MPI for message passing. They modified linearized implicit solution algorithm for parallel execution using a block-decoupled sub-iterative strategy. They developed a heuristic performance estimate which couples parallel domain decomposition and grid generation with the characteristics of the targeted computer system. They concluded that the sequential convergence rate could be recovered at a reasonable cost by using sufficient number of sub-iterations.

Parallel solvers for complex three dimensional multi-phase flows have been developed by Xiao, et. al. [9]. They have implemented 1-D partition of the computational domain into sub-domains along the z-direction with each sub-domain assigned to a single processor. They solve the Navier-Stokes Equation applied to the multi-phase systems using pressure-based algorithm with the resulting linear system $Ax = b$ being solved by Bi-CGSTAB method with the

preconditioner formulated as a tridiagonal approximation. However, they had difficulties in attaining a high performance with higher number of processors which they have attributed to the communication between the processors and the reduction in vector length due to parallelization.

The present work is concerned with parallelizing a sequential multi-block chemically reacting flow solver 'CHEQNS' written by Cox [3]. CHEQNS couples a chemical equilibrium solver with a three-dimensional, Navier-Stokes flow solver that has been modified to include real gas effects (real gas in this context is reacting mixtures of *thermally perfect gases* in local thermo-chemical equilibrium). The black box chemistry solver computes the equilibrium composition and temperature of gas mixtures at constant density and internal energy, and provides the flow solver with the necessary thermodynamic and transport properties.

The sequential solver used in this work has been parallelized previously by Carino et. al [2], using Cray T3D Fortran Programming model called CRAFT (Cray Research Adaptive Fortran). A brief description of their approach is presented later. The present work consists of developing a portable parallel version of CHEQNS using Message Passing Interface (MPI) with the emphasis on the flow solver. A coarse grain approach has been adapted with blocks being distributed on different processors.

The thesis is organized as described in the following lines. Chapter II discusses the equations that are solved by CHEQNS and the multi-block algorithm. Chapter III discusses the parallel algorithm and implementation issues. Chapter IV presents the comparison of solutions from different test cases between the parallel implementation and the sequential solver. Chapter V discusses the parallel analysis and presents the issues related to parallel performance in terms of speedup and efficiency. Chapter VI presents the conclusions.

CHAPTER II

BACKGROUND

2.1 Black Box Solver

Chemically reacting flows can be classified based on relative magnitudes of reaction time τ_R and fluid dynamic time τ_{FD}, the time required by a particle to traverse the flow domain. Three distinct cases arise corresponding to,

$$\tau_R \ll \tau_{FD}$$

$$\tau_R \approx \tau_{FD}$$

$$\tau_R \gg \tau_{FD}.$$

In the first case, the reaction time is much lesser than the fluid dynamic time and hence the reaction attains chemical equilibrium. In this case at each volume cell in the flow domain and for all reactions occurring in the flow, the flow-field can be assumed to be in local chemical equilibrium. The second case is the most general case, finite-rate chemistry, in which the reaction may or may not go to completion. The third case corresponds to that of a frozen chemistry, in which the reaction has no time to occur. Here the backward reaction rates approach zero and there will be no change in the mass-fractions of species if diffusion is negligible. The 'CHEQNS' code solves flows in local chemical equilibrium. The black box solver

is capable of determining the composition of an arbitrary mixture of thermally perfect gases in local thermo-chemical equilibrium. A detailed account of the numerical formulation, chemical models and thermodynamic relations used in the solver can be found in [3].It should be mentioned at this point that the emphasis of the present work is on the flow solver part of CHEQNS.

2.2 Equations Describing Fluid Flow

The equations describing the fluid motion in the general curvilinear coordinates, $\xi = \xi(x, y, z, t)$, $\eta = \eta(x, y, z, t)$, $\zeta = \zeta(x, y, z, t)$, $\tau = t$ where x, y, z, t represent the Cartesian coordinates, written in the strong conservation form read as,

$$\frac{\partial Q}{\partial \tau} + \frac{\partial F}{\partial \xi} + \frac{\partial G}{\partial \eta} + \frac{\partial H}{\partial \zeta} = \frac{\partial F_v}{\partial \xi} + \frac{\partial G_v}{\partial \eta} + \frac{\partial H_v}{\partial \zeta} \tag{2.1}$$

where the dependent variable vector is,

$$Q = J \begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_o \end{bmatrix}, \tag{2.2}$$

the inviscid flux vectors are

$$K = J \begin{bmatrix} \rho\theta_k \\ \rho u\theta_k + k_x p \\ \rho v\theta_k + k_y p \\ \rho w\theta_k + k_z p \\ \rho e_o\theta_k + p(k_x u + k_y v + k_z w) \end{bmatrix}, \tag{2.3}$$

with,

$$\theta_k = k_t + k_x u + k_y v + k_z w$$

and,

$$\begin{aligned} K = F, \quad \theta_k = U \quad \text{for} \quad k = \xi \\ K = G, \quad \theta_k = V \quad \text{for} \quad k = \eta \\ K = H, \quad \theta_k = W \quad \text{for} \quad k = \zeta, \end{aligned}$$

and the viscous flux vectors are

$$K_v = J \begin{bmatrix} 0 \\ T_{k1} \\ T_{k2} \\ T_{k3} \\ \mathcal{Q}_k + uT_{k1} + vT_{k2} + wT_{k3} \end{bmatrix}, \tag{2.4}$$

with,

$$\begin{aligned} \mathcal{Q}_k &= q_x k_x + q_y k_y + q_z k_z \\ T_{k1} &= \tau_{xx} k_x + \tau_{xy} k_y + \tau_{xz} k_z \\ T_{k2} &= \tau_{xy} k_x + \tau_{yy} k_y + \tau_{yz} k_z \\ T_{k3} &= \tau_{xz} k_x + \tau_{yz} k_y + \tau_{zz} k_z \end{aligned} \tag{2.5}$$

where,

$$\begin{aligned} q_x &= -\kappa\left(\xi_x\frac{\partial T}{\partial \xi}+\eta_x\frac{\partial T}{\partial \eta}+\zeta_x\frac{\partial T}{\partial \zeta}\right) \\ q_y &= -\kappa\left(\xi_y\frac{\partial T}{\partial \xi}+\eta_y\frac{\partial T}{\partial \eta}+\zeta_y\frac{\partial T}{\partial \zeta}\right) \\ q_z &= -\kappa\left(\xi_z\frac{\partial T}{\partial \xi}+\eta_z\frac{\partial T}{\partial \eta}+\zeta_z\frac{\partial T}{\partial \zeta}\right) \end{aligned} \tag{2.6}$$

and for a Newtonian fluid,

$$\begin{aligned} \tau_{xx} &= \tfrac{2}{3}\mu\left[2\left(\xi_x\frac{\partial u}{\partial \xi}+\eta_x\frac{\partial u}{\partial \eta}+\zeta_x\frac{\partial u}{\partial \zeta}\right)-\left(\xi_y\frac{\partial v}{\partial \xi}+\eta_y\frac{\partial v}{\partial \eta}+\zeta_y\frac{\partial v}{\partial \zeta}\right)-\left(\xi_z\frac{\partial w}{\partial \xi}+\eta_z\frac{\partial w}{\partial \eta}+\zeta_z\frac{\partial w}{\partial \zeta}\right)\right] \\ \tau_{yy} &= \tfrac{2}{3}\mu\left[2\left(\xi_y\frac{\partial v}{\partial \xi}+\eta_y\frac{\partial v}{\partial \eta}+\zeta_y\frac{\partial v}{\partial \zeta}\right)-\left(\xi_z\frac{\partial w}{\partial \xi}+\eta_z\frac{\partial w}{\partial \eta}+\zeta_z\frac{\partial w}{\partial \zeta}\right)-\left(\xi_x\frac{\partial u}{\partial \xi}+\eta_x\frac{\partial u}{\partial \eta}+\zeta_x\frac{\partial u}{\partial \zeta}\right)\right] \\ \tau_{zz} &= \tfrac{2}{3}\mu\left[2\left(\xi_z\frac{\partial w}{\partial \xi}+\eta_z\frac{\partial w}{\partial \eta}+\zeta_z\frac{\partial w}{\partial \zeta}\right)-\left(\xi_x\frac{\partial u}{\partial \xi}+\eta_x\frac{\partial u}{\partial \eta}+\zeta_x\frac{\partial u}{\partial \zeta}\right)-\left(\xi_y\frac{\partial v}{\partial \xi}+\eta_y\frac{\partial v}{\partial \eta}+\zeta_y\frac{\partial v}{\partial \zeta}\right)\right] \\ \tau_{xy} &= \mu\left[\left(\xi_y\frac{\partial u}{\partial \xi}+\eta_y\frac{\partial u}{\partial \eta}+\zeta_y\frac{\partial u}{\partial \zeta}\right)+\left(\xi_x\frac{\partial v}{\partial \xi}+\eta_x\frac{\partial v}{\partial \eta}+\zeta_x\frac{\partial v}{\partial \zeta}\right)\right] \\ \tau_{yz} &= \mu\left[\left(\xi_z\frac{\partial v}{\partial \xi}+\eta_z\frac{\partial v}{\partial \eta}+\zeta_z\frac{\partial v}{\partial \zeta}\right)+\left(\xi_y\frac{\partial w}{\partial \xi}+\eta_y\frac{\partial w}{\partial \eta}+\zeta_y\frac{\partial w}{\partial \zeta}\right)\right] \\ \tau_{xz} &= \mu\left[\left(\xi_z\frac{\partial u}{\partial \xi}+\eta_z\frac{\partial u}{\partial \eta}+\zeta_z\frac{\partial u}{\partial \zeta}\right)+\left(\xi_x\frac{\partial w}{\partial \xi}+\eta_x\frac{\partial w}{\partial \eta}+\zeta_x\frac{\partial w}{\partial \zeta}\right)\right] \end{aligned} \tag{2.7}$$

and,

$$\begin{aligned} K_v = F_v, \quad \theta_k = U \quad \text{for} \quad k = \xi \\ K_v = G_v, \quad \theta_k = V \quad \text{for} \quad k = \eta \\ K_v = H_v, \quad \theta_k = W \quad \text{for} \quad k = \zeta, \end{aligned}$$

The Jacobian of the coordinate transformation is

$$J = det \left| \frac{\partial(x, y, z)}{\partial(\xi, \eta, \zeta)} \right|$$

In the preceding description, ρ represents the density of the fluid, p the pressure, e_o the total specific energy, κ the thermal conductivity and μ the viscosity. τ_{ij} represents the shear stress on the i^{th} face along the j^{th} direction and q_k the heat flux along the k^{th} direction. The Cartesian components of the velocity are u, v, w along the x, y, z directions respectively. The contravariant velocities normal to constant ξ, η, ζ surfaces are represented by U, V, W respectively. The inviscid flux vectors are represented by F, G, H while their viscous counterparts are represented by F_v, G_v, H_v. The equations in the above form are valid for homogeneous and isotropic flow field.

2.3 Flow Solver

The equations written in the preceding section are spatially and temporally discretized using the implicit cell-centered finite volume formulation (with $\Delta\xi = \Delta\eta = \Delta\zeta = 1$) and, for a first order temporal accuracy, the discretization can be written as,

$$\frac{\Delta Q^n}{\Delta\tau} + \delta_i(F - F_v)^{n+1} + \delta_j(G - G_v)^{n+1} + \delta_k(H - H_v)^{n+1} = 0 \qquad (2.8)$$

where $\Delta Q^n = Q^{n+1} - Q^n$, $n+1$ being the current time level , and δ_l is the central difference operator with $\delta_l = ()_{l+\frac{1}{2}} - ()_{l-\frac{1}{2}}$. The version of 'CHEQNS' solver written by Cox [3] is based on the thin layer Navier-Stokes approximation. Webster [7] extended the capabilities of 'CHEQNS' to solve the complete Navier-Stokes

equations coupled to a heat conduction model. The version used in the present work is intermediate to these and has capabilities of solving the complete Navier-Stokes equations but does not include the heat conduction model. The flow solver allows for implicit treatment of viscous fluxes. The flow solver implements an Approximate Riemann solver described in [3]. The linearized system of equations is solved using modified two-pass factorization developed by Whitfield [8]. The solver has provision for symmetric Gauss-Siegel iterations for zeroing the error due to approximate factorization and Newton iterations to converge the solution within a time-step in the case of unsteady problems where time accuracy is important.

2.4 Multi-Block Algorithm

The multi-block algorithm has been treated rigorously by Belk [1] for the two-pass scheme. In this brief development the ideas presented [1] in are extended for the the modified two-pass scheme primarily to form a basis for understanding the block-block "communication" of the boundary conditions. The derivation of the modified two-pass scheme can be found in Appendix A. The modified two-pass scheme for solving the linearized system arising from the discretization of the equations describing fluid flow can be written as,

$$\begin{aligned} [D - L\cdot]^n X^1 &= -R^n \\ [D + U\cdot]^n \Delta Q^n &= D^n X^1 \end{aligned} \tag{2.9}$$

where,

$$D^n X^1 = -R^n + L^n \cdot X^1$$

with,

$$D^n = \frac{I}{\Delta\tau} + (A^+_{i+\frac{1}{2}} - A^-_{i-\frac{1}{2}}) + (B^+_{j+\frac{1}{2}} - B^-_{j-\frac{1}{2}}) + (C^+_{k+\frac{1}{2}} - C^-_{k-\frac{1}{2}})$$

$$L \cdot X^1 = (A^+_{i-\frac{1}{2}} X^1_{i-\frac{1}{2},j,k} + B^+_{j-\frac{1}{2}} X^1_{i,j-\frac{1}{2},k} + C^+_{k-\frac{1}{2}} X^1_{i,j,k-\frac{1}{2}})$$

$$U \cdot \Delta Q^n = (A^-_{i+\frac{1}{2}} \Delta Q^n_{i+\frac{1}{2},j,k} + B^-_{j+\frac{1}{2}} \Delta Q^n_{i,j+\frac{1}{2},k} + C^-_{k+\frac{1}{2}} \Delta Q^n_{i,j,k+\frac{1}{2}})$$

where,

$$A^{\pm} = \frac{\partial (F - F_v)^{\pm}}{\partial Q},\ B^{\pm} = \frac{\partial (G - G_v)^{\pm}}{\partial Q},\ C^{\pm} = \frac{\partial (H - H_v)^{\pm}}{\partial Q},$$

where the superscript (+) is used to represent the components of flux vector associated with the positive eigenvalues of the corresponding flux jacobian and vice versa.

If we divide the flow domain into sub-domains such that the multi-block gridding scheme could be applied then for a two-block division of the domain Eq. 2.9 takes the form,

$$\begin{aligned}
[D_1 - L_1\cdot]^n X^1_1 &= -R^n_1 \\
[D_2 - L_2\cdot]^n X^1_2 + T^L_{2,1} X^1_1 &= -R^n_2 \\
[D_1 + U_1\cdot]^n \Delta Q^n_1 + T^U_{1,2} \Delta Q^n_2 &= D^n_1 X_1 \\
[D_2 + U_2\cdot]^n \Delta Q^n_2 &= D^n_2 X_2
\end{aligned} \tag{2.10}$$

where the subscripts (1) and (2) represent block 1 and block 2 respectively. $T^L_{2,1}$ and $T^U_{1,2}$ are rectangular matrices that contain the coefficients of quantities, X^1_1 in block 1 and ΔQ^n_2 in block 2 necessary to complete the spatial discretization in block 2 and block 1 respectively and the superscripts L and U indicate whether

they were part of the lower triangular or upper triangular matrices corresponding to the single block problem represented by Eq.2.9.

The first two equations correspond to the forward pass and the last two correspond to the backward pass in the blocks (1) and (2). Solving the forward pass in block 1 requires the elements in the first two cells across the boundary in block 2 from the previous time level to evaluate the flux at the boundary cell face. Similar to the forward pass in block 1, the backward pass in block 2 requires the elements in the first two cells across the boundary in block 1 from the previous time level. To solve the forward pass in block 2, the values in X_1^1 corresponding to the first cell off the boundary in block 1 are required. In the same manner, solving for ΔQ_1^n in block 1 during the backward pass requires the values in ΔQ_2^n corresponding to the first cell off the boundary in block 2. To obtain exact results as in the unblocked flow domain, we would have to perform in the following order.

(i) Solve the forward pass in block 1

(ii) Make available X_1^1 values to block 2

(iii) Solve the forward pass in block 2

(iv) Solve backward pass in block 2

(v) Make available ΔQ_2 values to block 2

(vi) Solve for ΔQ_1 in block 1 using backward pass.

The above sequence corresponds to the implicit implementation of the modified two-pass scheme. The corresponding code structure will have the block loop within the two-pass loop. Another approach suggested in [1], is the explicit implementation of the schemes like two-pass, modified two-pass and others that

involve passes like the forward pass and backward pass. In this implementation, the modified-two-pass scheme for example, is block-decoupled in the sense that X_1 and ΔQ_2 are approximated using available information and the forward and backward passes are executed within the block. In this scheme X_1^1 values may be approximated using ΔQ_1^n or ΔQ_1^{n-1} and the ΔQ_2^n by ΔQ_2^{n-1}. The use of approximate values for these quantities introduces a conservation error as the flux in the boundary cells are evaluated differently on either side of the block-block boundary.

Two different modes of flux updates at the boundaries have been described in [1],. In the synchronized scheme the fluxes at the boundaries are calculated explicitly using time level n information. In the unsynchronized scheme the calculation of the fluxes is done using the latest available information from the other blocks. It should be noted here that in the case of unsynchronized communication the fluxes at different boundaries in a block may not be from the same time-level. The details of the implementation of the flux updates are treated in the next chapter.

CHAPTER III

PARALLEL IMPLEMENTATION

The primary difference between a sequential methodology and a coarse grain parallel methodology for the solving flows on a multi-block grid is the way in which the information is exchanged between the sub-domains. Whether we can do without information exchange is another question altogether and it has been presented in [1], that certain amount of information exchange is imperative for the solution to be accurate enough for engineering practices. In the case of the sequential algorithm information exchange can be either through buffer copying or by using complicated logic. In the present case the sequential code employs copying from one buffer to another as a means of information exchange. An understanding of the information exchange in sequential algorithm provides the footage for understanding the communication in parallel. Parallel paradigm is not all about communication alone, it has other aspects like memory requirements and parallel I/O and these are treated later in this chapter.

3.1 Sequential Code Structure

The sequential code can be conceived as consisting of three primary units viz.,

1. Flow Solver Initialization - This consists of reading the input files and calculating the initial conditions from a set of input values and then

calculating the free-stream values, reading the boundary conditions for the different blocks and initializing arrays that store information regarding the accuracy at the boundaries, block-block connectivity etc., and setting up different flags that are associated with different options.

2. Block Initialization - This consists of reading the grid for the block, applying the initial conditions throughout the block. In the case of restart, the grid and the dependent variables at each cell center are read from the restart files.

3. Solving - In this part the solution algorithm is applied to the blocks for a specified number of time step iterations and the solution is printed out at a specified frequency.

The solution algorithm consists of a time-stepping loop, newton iteration loop and a block loop, the outer most being the time-stepping loop and the inner most the block loop. The symmetric Gauss-Seidel iteration loop is block-decoupled and is located within the block loop. In the block loop, for any block, the pointer values corresponding to various locations in the ribbon vector are calculated. The spatial metrics are then evaluated followed by the equilibrium composition and thermodynamic properties, and temporal metrics. The boundary conditions are then applied and the block-block boundary conditions are read from the resident boundary conditions array which will be explained in the section discussing memory issues. The linearized system of equations is then solved by block decoupled symmetric Gauss-Seidel iterations with alternating forward and backward passes. The boundary conditions and the dependent variables are then updated and the same set of operations are performed on the other blocks.

In a sequential algorithm the blocks are processed one after another and this has an implication on how the synchronous and the asynchronous updates

of fluxes at the boundaries are realized. To recall, the synchronous update of fluxes in the boundary cells uses the previous time level (time level n) information from cells across the boundaries in the neighboring blocks while the asynchronous update uses the latest available information from the cells across the boundaries in the neighboring blocks. Because the blocks are processed one after another starting from block 1, a synchronous update will be ensured if, after processing a block, the updated dependent variables are not made available to the neighboring blocks with higher block numbers. However, in the beginning of the next time-step iteration, all the blocks should make these values available to their higher numbered neighbors. This implies doubly storing the dependent variables corresponding to the boundary cells of block-block boundaries of a block. In any case the updated dependent variables must be made available to the neighbors with lower block numbers in the same time step. The flow of information for this case is illustrated in Figures 3.1 and 3.2 which correspond to the first and second time-step respectively for a flow domain consisting of 5 blocks. The coloring indicates the time-level of the data. The appendages along the boundaries of the blocks represent the arrays in which the data corresponding to the two layers of cells within the boundaries are stored. The values stored in these arrays are the boundary conditions for the neighboring blocks sharing those boundaries. It can be seen that, before a block is processed, all the appendages on the neighboring blocks corresponding to the block-block boundaries are at the same time-level. The asynchronous updates are far more simple to code. In this case, the dependent variables at the block boundaries are made available to the higher and lower numbered neighbors immediately after the block has been processed. The flow of information for this case is illustrated in Figures 3.3 and 3.4 corresponding to the first and second time-steps. The sequential algorithm has been outlined in Figure 3.5.

3.2 Parallel Algorithm

3.3 Objectives

The objective of the present work is to develop and implement a robust parallel algorithm for parallelizing a sequential solver for solving chemically reacting flows on multi-block structured grids addressing the following concerns:

1. The parallel code should produce solutions that are consistent with the sequential code.

2. It should have good performance metrics particularly speedup and efficiency.

3. It should be portable and scalable.

4. It should be able to handle multiple blocks per processor and should have the restart capability on different number of processors.

5. It should retain the sequential solver's capability to handle arbitrary block topology.

6. The parallelization should be generic and should be able to handle any mapping of blocks on to processors, so that in future load-balancing algorithms may be easily implemented.

The sequential code has previously been parallelized by Carino et. al. [2]. Their target architecture was Cray T3D. They used the T3D Fortran programming model, CRAFT (Cray Research Adaptive Fortran), in their work. CRAFT supports "work-sharing" program methodology through implicit communication and explicit message-passing through the logically-shared memory of T3D. In order to parallelize the flow solver without a major code rewrite, the

data structures were retained and the loop iterations were distributed among the processing elements (PEs).

They observed that the subroutines associated with the computation of equilibrium composition and thermodynamic properties (black box), and time-stepping (step) consumed 90 % of the execution time. They parallelized the loops within these subroutines which essentially corresponds to dividing a block between the different processors. The loops in the black box subroutine are associated with cell independent computations while those in the step subroutine have strong data dependencies. Communications between the PEs were through explicit message-passing using SHMEM (Shared Memory) library subroutines. Each PE has a copy of the block and it performs computations on the portion of the block assigned to it, after which it exchanges results with other PEs through SHMEM routines if the results are necessary for subsequent computations. All the PEs read the input and temporary files. PE 0 writes the output and temporary files and this is executed in a sequential fashion where, PE 0 accesses other PEs for current values to write to the temporary files.

Recalling from Chapter 1, CFD codes can be parallelized either at grid-point level or at sub-domain level, the former corresponds to fine grain parallelism and the latter to coarse grain parallelism. In the present work, a coarse-grain parallelization has been sought. A multi-block sequential algorithm naturally lends itself to coarse grain parallel implementation. The information exchange between blocks can be visualized in a parallel implementation as communication between processors which store the blocks. Another reason for seeking this approach is performance. In the coarse grain paradigm, there exists a higher ratio of computations to communications as compared to the fine grain alternative. This is important for the performance of the parallel algorithm. The parallel

algorithms which are presented in this work fall into the *Single Program, Multiple Data, (SPMD)* category. The implementation is based on the message passing model and has been accomplished using the Message Passing Interface (*MPI*). The rank 0 processor acts as a master processor onlywhen it reads the input and distributes it to other processors and when printing solutions corresponding to different blocks to a single file. At all other times, there is no master processor. All the processors perform computations on the blocks and synchronize to either exchange information or complete the exchange of information.

Conceptually the parallel code retains the same structure as the sequential code though there are differences with regard to I/O, memory allocation and communication. The parallel I/O and memory issues are discussed later in the chapter. The primary difference between the sequential and the parallel methodologies is the communication. Two algorithms, different in the communication pattern with respect to the communication of block-block boundary data, one based on blocking MPI primitives and the other based on non-blocking primitives have been implemented. Towards the end of this section, issues regarding asynchronous implementation are discussed.

When blocking calls in MPI are used for communication, the call returns only after the completion. Their nature implies that communication must occur separately after all the processors have processed all their local-blocks. This is acceptable in the synchronous flux update methodology where the data needed from the neighboring blocks corresponds to the previous time-level. The processors are synchronized after the block loop before entering the communication phase. The communication loop is sequential in the sense that the loop is over all the blocks and there is synchronization at the end of the loop, so that processors are in the same "state" every time the loop is executed, to ensure correctness of

communication. The loop can be thought to consist of two segments, the sending segment and the receiving segment. In the sending segment, the processor that owns the block sends the relevant data to processors owning the blocks which share a boundary with the given block. In the receiving segment, the processors owning the blocks which share a boundary with the given block receive the relevant data from the processor owning the given block. In this scheme, the processor that is sending the data is unique and this ensures that the correct data is being sent to the correct processor. In the case when a processor handles more than one block and the local blocks share a boundary, to ensure that the synchronized update method is implemented correctly, the double storing of the dependent variables and solution vectors mentioned previously is used in an advantageous way. Within the local block loop, the solution vector and the dependent variables corresponding to the boundaries are stored for neighbors with blocks numbers lower than the block being processed. And in the communication phase the storing is done for the neighboring blocks with block numbers higher than the given block. This means that the local block numbering must preserve the order property of the global block numbers. The outline of the solution algorithm is presented in Figure 3.6.

Nonblocking primitives do not wait for the send/receive operation to complete. This helps in overlapping computation with communication if the computation does not dependent on the communication. When the synchronized update methodology is considered, the computation requires only the previous time-level data while the communication is associated with the current time-level data. Complete independence of computation and communication can be achieved if we update all block-block boundary conditions before the loop over the local-blocks. This would ensure that data received during the current time-step is not used in the current time-step computations. Thus, in this implementation

the communication loop that followed the loop over local blocks in the case of blocking implementation is no longer required. It is replaced by a call to a function that blocks until all non-blocking communication finish. The solution algorithm is presented in Figure 3.7.

In the multi-block algorithm, as had been described in Chapter 2, the solution vector corresponding to the first layer of cells adjoining a block-block boundary needs to be exchanged between the blocks sharing the boundary. The sequential algorithm in the present case is block-decoupled with the solution vector being exchanged between neighboring blocks after the completion of the forward and backward passes associated with the Gauss-Seidel scheme. The solution vector, unlike the dependent variables, is always exchanged between the blocks asynchronously so as to ensure the availability of the most recent solution vector across the block-block boundary which is necessary for good convergence properties. However in the parallel implementation, the solution vector is exchanged synchronously. This is one respect in which the parallel implementation could degrade the performance in terms of convergence.

When the asynchronous method for the update of fluxes at boundaries is used, the boundaries of different blocks are at different time levels and, within the same block also different boundaries may be at different time-levels. This is because the asynchronous methodology strongly depends on the order in which blocks are processed. A general parallel implementation cannot process the blocks in the same order as sequential without performance loss in terms of speedup. This implies parallel asynchronous algorithm cannot produce the same solution as the sequential asynchronous algorithm. This was one of the reasons for not implementing asynchronous algorithm in parallel. The algorithm discussed previously using non-blocking communication can be modified for asynchronous method by moving the

boundary conditions update inside the loop over the local blocks. As there is no control as to when the non-blocking receive will be completed, there is a chance that a boundary may be only partially updated and this can lead to instabilities. Therefore, this algorithm may not implement the asynchronous update in the strict sense.

3.4 Grid Blocking

The phrase "grid blocking" has been used here in a broad sense to include mapping of blocks on to processors as well as the way in which blocks are numbered. In the parallel implementation of multi-block method, it is imperative to have a consistent methodology for numbering the blocks because

1. The blocks need to be connected logically in exactly the same way as they are in the physical space. Otherwise, we may be solving a different problem altogether.

2. Consistent block numbering is crucial for correct communication sequence and for matching sends and receives associated with the communication.

3. The correct application of boundary conditions and initial conditions, which are critical for correctness of solution, strongly depends on a consistent numbering of blocks.

4. A consistent numbering is needed for a generalized approach towards handling multiple blocks per processor with arbitrary block connectivity.

5. It would also be easier to implement load balancing algorithms at a later stage.

In the present implementation there are two different numberings associated with a block, global block number and the local block number. The global block number for a block is the same as the block number for the block in the sequential algorithm. Local block numbering is more relevant in the context where a processor handles more than one block. It is the numbering that is applied to the blocks stored on the same processor and in that sense is local to that processor. It offers a scheme or ordering for processing blocks within a processor and has no physical implication. However, the local block numbering could be exploited in load balancing. A scheduling order for the processing of blocks could be arrived at taking into considerations factors such as the block dimensions, number of boundaries shared with other blocks etc., and that order could as well be used for local block numbering.

The parallel implementation allows for user specified mappings of blocks on to processors. This has been done with a perspective for incorporating load balancing algorithms at a later date. The input mapping would then be the output from a load balancing algorithm. The default mapping used in the solver corresponds to a scattered distribution of blocks on to processors. This is illustrated in Figure 3.8. However, the local block numbering is always decided by the code based on the global block numbers of the blocks stored on a processor. The local block number of a block having the least global block number is assigned a local block number of 1, the one with next higher global block number 2 and so on which, is illustrated in Figure 3.8. This is done because the local block numbering needs to preserve the order of the corresponding global block numbers which is critical for the correct implementation of the synchronized update in the case where a processor owns more than one block with local blocks sharing boundaries. If a scheduling algorithm exploits the local block numbering then

provision must be made to ensure the correct implementation of the synchronized update in the case where a processor owns more than one block with local blocks sharing boundaries.

3.5 Memory Issues

Many aspects relating to memory management in the sequential code were retained in the parallel implementation. The design principles which were the basis for memory allocation in the sequential code were,

1. Minimizing memory requirements

2. Making efficient use of the high speed secondary memory available on some computers like the secondary storage device (SSD) on CRAY X-MP.

In a multi-block gridding of a flow domain, the blocks need not all have the same dimensions. So the multi-dimension arrays associated with storing various data like dependent variables, flux jacobians, metrics etc., could be of different dimensions and lengths for the different blocks. In order to provide adjustable dimension arrays and prevent wastage of memory on the secondary high speed memory, the sequential code used ribbon vectors utility available in FORTRAN. The advantage in using ribbon vector is specific arrays irrespective of their dimensioning can be easily accessed within subroutines by passing the array as a parameter in the calling statement and using standard FORTRAN multi-dimension array declaration within the subroutine. The code calculates the length of equivalent one dimensional array for storing different arrays associated with a block for all the blocks. The starting locations of the different arrays in the one-dimensional array are then assigned giving due consideration for enough space

between the start of one array and the next. However, when the secondary high speed memory is not used then the data associated with all the blocks would be stored in the ribbon vector sequentially one block after the next.

The values of the dependent variables in the first two cells adjoining a block to block interface and, the most recent values of the solution vector corresponding to the first cell adjoining a block-block interface for every block are stored in a set of resident boundary condition arrays. These arrays are updated every newton iteration or time step iteration depending on whether the code is setup for multiple newton iterations or not. Each block is dimensioned so that there is enough space for storing all the boundary condition information corresponding to the phantom cells while a block is being processed. When a block is being processed, the block-block boundary conditions for the block are obtained from the resident boundary condition arrays corresponding to the neighboring blocks and copied to the appropriate phantom cell location. This was advantageous in the sequential algorithm from the following perspectives,

1. It allowed easy implementation of "synchronized" calculation of the explicit flux balance at the block-block boundaries.

2. With regard to secondary high speed memory, a block needs to read or written to only once per time step/newton iteration.

3. This offered easy implementation for vectorizing calculations for a block.

In the present work, the blocks are distributed among the processors in such a way that each processor has at least one block. The memory requirement for a processor is primarily decided by the dimensions of the blocks that it would be processing. However, because FORTRAN 77 doesn't support dynamic memory

allocation, each processor is allocated memory which corresponds to maximum needed by any processor.The parallel code also makes use of the ribbon vector as it greatly simplifies coding involving arrays of different dimensions and different lengths. It also provides a level of transparency to the way blocks are processed within different subroutines.

As in the sequential code resident boundary conditions are stored permanently in separate memory locations. This has a three fold reason,

1. It provides a buffer for sending and receiving the dependent variables and solution vectors between processors whose blocks share a boundary. This eliminates the need for separate temporary buffers.

2. The boundary conditions are stored contiguously in the resident boundary arrays. Hence, with a knowledge of the length of a boundary interface alone, one can use these arrays directly in the Send and Receive primitives.

3. In the case of non-blocking communication it provides a unique address to which the non-blocking receive could direct the received data on completion. Because these arrays are not directly used in the computations they help in overlapping computation with communication.

If a block-block boundary is considered such that the blocks associated with it are stored on different processors then, the memory allocated in each of the processors for storing the boundary conditions, associated with this boundary, would be equal to the memory required to store all the variables corresponding to the first two layer of cells adjoining the boundary within the block that the processor is storing, together with that required to store all the variables in the first two layers of cells adjoining the boundary in the neighboring block. The

total memory allocated on both the processors for storing the boundary conditions associated with that boundary is then twice that allocated by the sequential code. That part of the memory which stores the boundary conditions of the neighboring block serves as the receive buffer for receiving the boundary conditions from the processor on which the neighboring block is stored, while the other part serves as the send buffer for sending the boundary conditions to the processor storing the neighboring block.

For the sequential case let,

$nb \rightarrow$ the total number of blocks.

$N_i \rightarrow$ the total number of boundaries that block i shares with other blocks.

$\mathcal{M}em_i \rightarrow$ be the memory requirement for storing all data corresponding to block i excluding that required for storing block-block boundary conditions in the resident boundary conditions array.

$\mathcal{M}\acute{e}m_{ij} \rightarrow$ be the memory requirement for storing data corresponding to resident boundary conditions in block i required by block j.

$$\mathcal{M}em^i_{block} = \mathcal{M}em_i + \sum_{j=1, i \neq j}^{N_i} \mathcal{M}\acute{e}m_{ij}$$

$$\mathcal{M}em_{total} = \sum_{i=1}^{nb} \mathcal{M}em_i + \sum_{i=1}^{nb} \sum_{j=1, i \neq j}^{N_i} \mathcal{M}\acute{e}m_{ij}$$

where $\mathcal{M}em^i_{block}$ and $\mathcal{M}em_{total}$ represent the total memory required for storing block i and that for all the blocks respectively.

For the parallel case let,

$nb_p \rightarrow$ the total number of blocks on processor p.

$N_I^p \rightarrow$ the total number of boundaries that block I on processor p shares with blocks on other processors.

$N_{Il}^p \rightarrow$ the total number of boundaries that block I on processor p shares with blocks within the same processor.

$\tilde{\mathcal{M}}em_I \rightarrow$ be the memory requirement for storing all data corresponding to block I excluding that required for storing block-block boundary conditions in the resident boundary conditions array.

$\tilde{\mathcal{M}}EM_{IJ} \rightarrow$ be the memory requirement for storing data corresponding to resident boundary conditions in local block I required by local block J.

$\tilde{\mathcal{M}}EM_{Ij} \rightarrow$ be the memory requirement for storing data corresponding to resident boundary conditions in local block I required by a non-local block j.

$$\kappa_{Il} = \begin{cases} 0 & \text{if } I \text{ doesn't share a boundary with } l; \\ 1 & \text{if } I \text{ shares a boundary with } l. \end{cases}$$

where l can be a block local or non-local to processor on which I is stored

$$\tilde{\mathcal{M}}em_{block}^I = \tilde{\mathcal{M}}em_I + \sum_{J=1, J\neq I}^{nb_p} \tilde{\mathcal{M}}EM_{IJ}\kappa_{IJ} + 2 * \sum_{j=1, j\neq i}^{nb} \tilde{\mathcal{M}}EM_{Ij}\kappa_{Ij}$$

$$\tilde{\mathcal{M}}em_{total}^p = \sum_{I=1}^{nb_p} \tilde{\mathcal{M}}em_I + \sum_{I=1}^{nb_p} \sum_{J=1, J\neq I}^{nb_p} \tilde{\mathcal{M}}EM_{IJ}\kappa_{IJ} + 2 * \sum_{I=1}^{nb_p} \sum_{j=1, j\neq i}^{nb} \tilde{\mathcal{M}}EM_{Ij}\kappa_{Ij}$$

where $\tilde{\mathcal{M}}em_{block}^I$ and $\tilde{\mathcal{M}}em_{total}^p$ represent the memory requirement for storing block I on processor p and the total memory requirement for storing all the blocks on processor p. Figures 3.9 and 3.10 schematically illustrate the memory requirements in the case of sequential and parallel implementations respectively.

The memory allocated to each processor is $max(\tilde{\mathcal{M}}em^{p}_{total})\, where\, p \in \{0, 1, \ldots, P-1\}$.. Therefore the total memory requirement for the parallel implementation is $P(max(\tilde{\mathcal{M}}em^{p}_{total}))$.

3.6 Parallel I/O

The CHEQNS solver requires four small input files. The input files are read by rank zero processor, the data that is read is then packed in character buffers and broadcasted to all other processors. The grid file needs to be read in if the code is started from time-step zero.In the sequential code the grid for each block is read within the block initialization loop. However, in the parallel implementation, the grid file is read by rank 0 processor and the grid blocks are distributed to the respective processors before initializing the blocks. The grid is copied on to the correct location in the ribbon vector for each block within the block initialization loop.

The parallel code has the provision for restart from a previous solution. It can be restarted using a different number of processors and, with a completely different mapping of the blocks on to processors. This has been accomplished by having different restart files for all the blocks. The idea of a single restart file was given up because it is costly in terms of time and memory. The grid is also written to the restart files. Hence, in the case of a restart, there is no reading and distribution of the grid blocks by rank '0' processor.

The code prints solution files at a specified frequency and has options for writing to a single solution file or to multiple solution files. In the latter case each processor writes the solution corresponding to a block independent of other processors to a different file while in the former case the solution is

accumulated on rank 0 processor and this is costly in terms of time and memory. The implementation does not treat the case where there could be a secondary high speed memory on each processor and the blocks are processed in such a way that there is only one block in memory at a given time while the others are stored in the secondary memory.

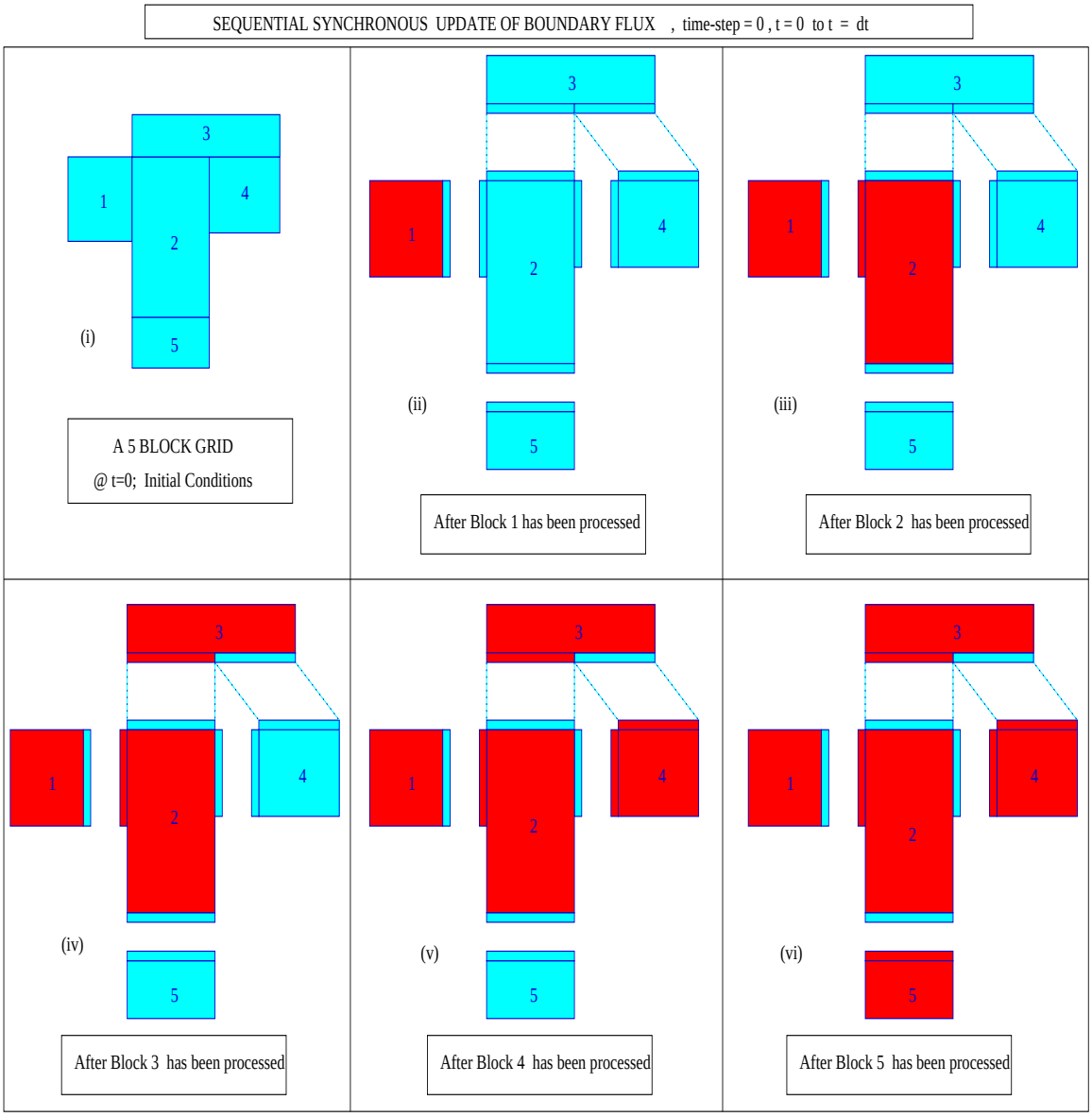

Figure 3.1: Synchronous update of boundary flux during the first time-step in the sequential algorithm

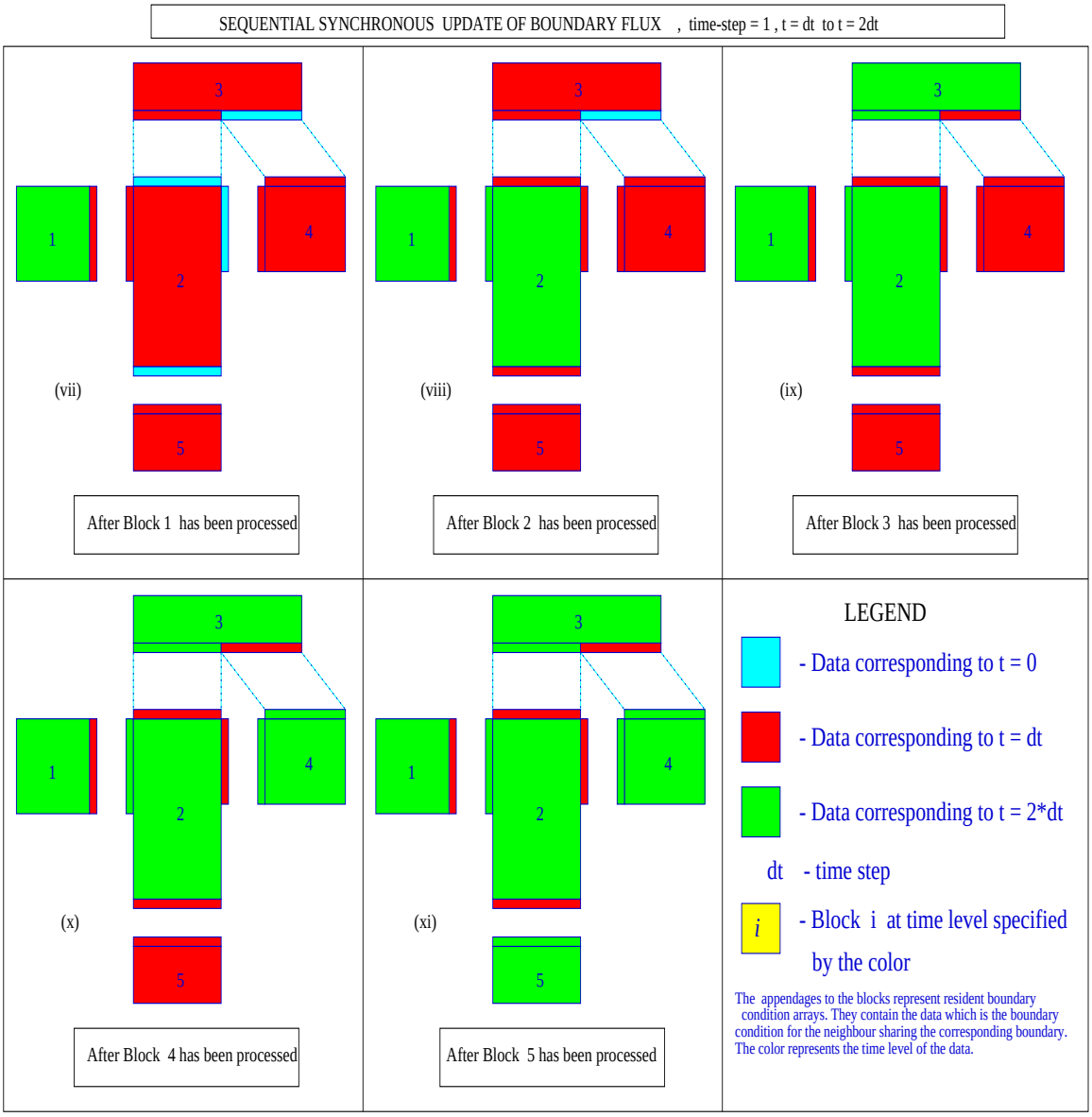

Figure 3.2: Synchronous update of boundary flux during the second time-step in the sequential algorithm

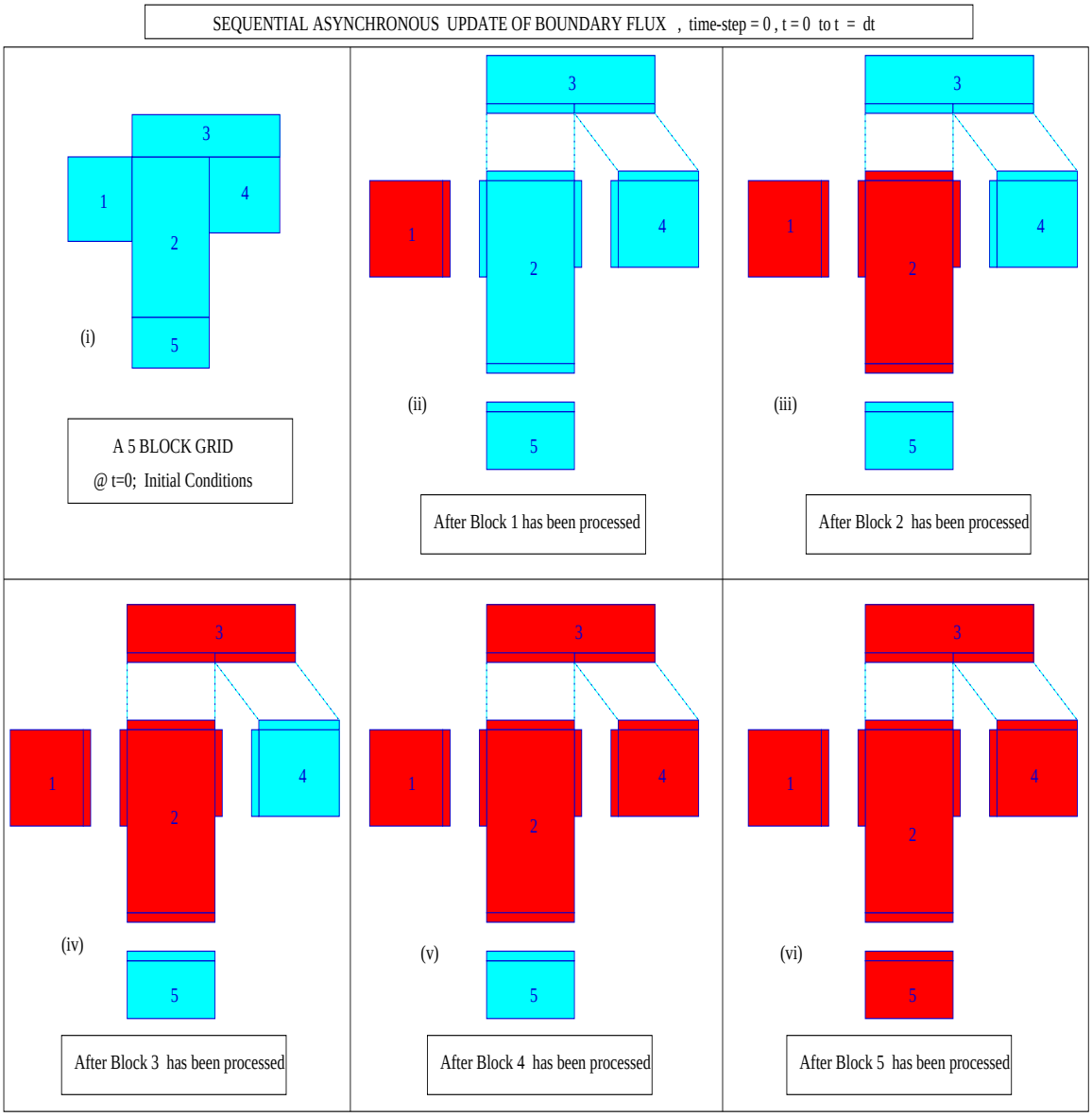

Figure 3.3: Asynchronous update of boundary flux during the first time-step in the sequential algorithm

Figure 3.4: Asynchronous update of boundary flux during the second time-step in the sequential algorithm

```
read Input Files
Initialize All Blocks

do istep = istep1,laststep // Time Stepping
      do new = 1,newton // Newton Iteration
            do ib = 1,nb // Block Loop
                  Calculate Space Metrics

                  Calculate Equilibrium Composition and Thermodynamic properties

                  if grid is moving then
                        Calculate Time Metrics
                  endif

                  Get Boundary Conditions

                  if Synchronous Update then
                        Make Qbc Available to higher number blocks
                  endif

                  do isg = 1,isgs // Symmetric Gauss-Siedel
                        forward sweep
                        backward sweep
                  enddo

                  if Asynchronous Update then
                        Make Latest Qbc Available to higher number blocks
                  endif

                  Make Qbc Available to lower number blocks

                  Make Solution Vector Available to Neighbours

                  Update Solution

                  if needed write(restart file)

                  if needed print(solution file)

            enddo // Block Loop
      enddo // Newton Iteration
enddo // Time Stepping
```

Figure 3.5: Outline of the Sequential Algorithm

OUTLINE OF THE PARALLEL IMPLEMENTATION USING BLOCKING MPI CALLS

```
do istep = istep1,laststep // Time Stepping
    do new = 1,newton // Newton Iteration
        if (new = newton) then
            if ((myid = 0).and.(time to write solution).and.(solution needs to be printed to a
            single file)) then
                do ib = 1,gnb
                    if (ib not stored in 0) then
                        call MPI_IRecv(solution) // Non-Blocking Call
                    endif
                enddo
            endif
        endif

        do ib = 1,nb // Local Block Loop
            Initialize Pointer Values
            Calculate Space Metrics
            Calculate Equilibrium Composition and thermodynamic properties

            if grid is moving then
                Calculate Time Metrics
            endif

            Get Boundary Conditions

            do isg = 1,isgs // Symmetric Gauss-Siedel
                forward sweep
                backward sweep
            enddo

            Make Qbc Available to lower number blocks

            Make Solution Vector Available to Neighbours

            Update Solution

            if solution needs to be printed then
                if solution is to be printed to a single file then
                    if (myid != 0) then
                        call MPI_ISend(solution to process 0) //Non-Blocking Call
                    else
                        write solution to appropriate Memory Location
                    endif
                else
                    write solution to the file
                endif
            endif

        enddo // Local Block Loop
```

Continued on next page ...

```
if (new = newton) then
    if ((myid = 0).and.(time to write solution).and.(solution needs to be printed to a single
    file)) then
        call MPI_Waitall( for solutions to be gathered on process 0)
    print(solution)
endif

call MPI_Barrier // Synchronization

    do ib = 1,gnb
        if (myrank = mapping of ib) then
            Make Qbc available to local higher number blocks.
            call MPI_Send(dQbc) // Solution Vector
            call MPI_Send(Qbc) // Dependent Variables
        else
            if (blocks share boundary with ib) then
                do ishb=1,nshb // Shared Boundary Loop
                    call MPI_Recv(dQbc)
                    call MPI_Recv(Qbc)
                enddo
            endif
        endif
        call MPI_Barrier
    enddo
enddo // Newton Iteration
enddo // Time Stepping
```

Figure 3.6: Outline of parallel algorithm using blocking communication of block-block boundary data

OUTLINE OF THE PARALLEL IMPLEMENTATION USING NON-BLOCKING MPI CALLS

```
do istep = istep1,laststep // Time Stepping
    do new = 1,newton // Newton Iteration
        if (new = newton) then
            if ((myid = 0).and.(time to write solution).and.(solution needs to be printed to a
            single file)) then
                do ib = 1,gnb
                    if (ib not stored in 0) then
                        call MPI_IRecv(solution) // Non-Blocking Call
                    endif
                enddo
            endif
        endif

        do ib = 1,gnb
            if (myrank = mapping of ib) then
                Get Block-Block Boundary Conditions
            else
                if (blocks share boundary with ib) then
                    do ishb=1,nshb // Shared Boundary Loop
                        call MPI_IRecv(dQbc)
                        call MPI_IRecv(Qbc)
                    enddo
                endif
            endif
        enddo

        do ib = 1,nb // Local Block Loop
            Initialize Pointer Values
            Calculate Space Metrics
            Calculate Equilibrium Composition and thermodynamic properties

            if grid is moving then
                Calculate Time Metrics
            endif

            Get Boundary Conditions

            do isg = 1,isgs // Symmetric Gauss-Siedel
                forward sweep
                backward sweep
            enddo

            Make Qbc Available to lower number local blocks
            Make Qbc Available to higher number local blocks
            call MPI_ISend(Qbc)
            Make Solution Vector Available to Local Neighbours
            call MPI_ISend(dQbc)
            Update Solution
```

Continued on next page ...

```
                if solution needs to be printed then
                    if solution is to be printed to a single file then
                        if (myid != 0) then
                            call MPI_ISend(solution to process 0) //Non-Blocking Call
                        else
                            write solution to appropriate Memory Location
                        endif
                    else
                        write solution to the file
                    endif
                endif

            enddo // Local Block Loop
            if (new = newton) then
                if ((myid = 0).and.(time to write solution).and.(solution needs to be printed to a
                single file)) then
                    call MPI_Waitall( for solutions to be gathered on process 0)
                print(solution)
            endif

            call MPI_Waitall( for all block-block boundary values to be exchanged)
            call MPI_Barrier // Synchronization

        enddo // Newton Iteration
    enddo // Time Stepping
```

Figure 3.7: Outline of parallel algorithm using non-blocking communication of block-block boundary data

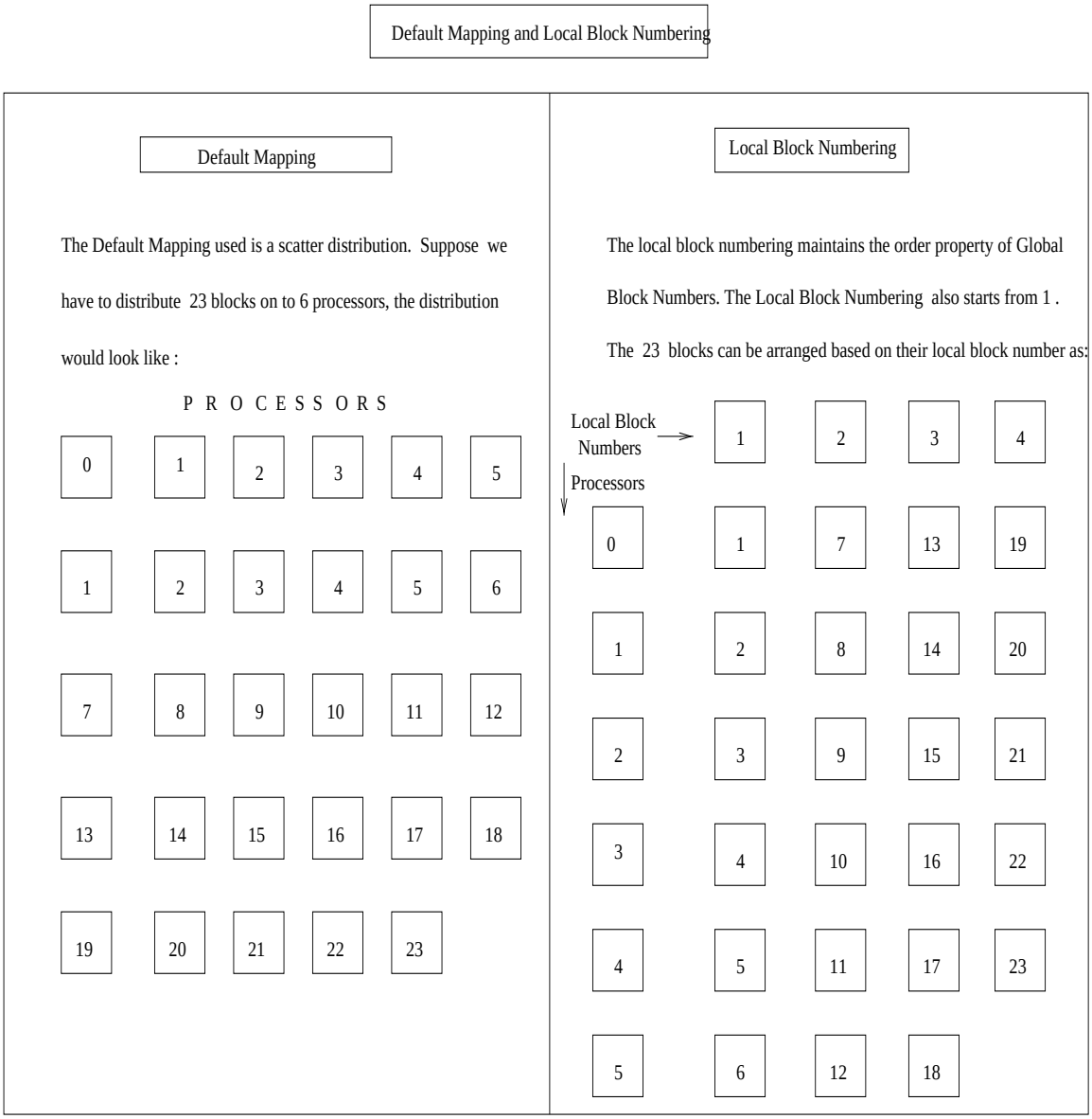

Figure 3.8: Default mapping of blocks to processors and local numbering in each processor

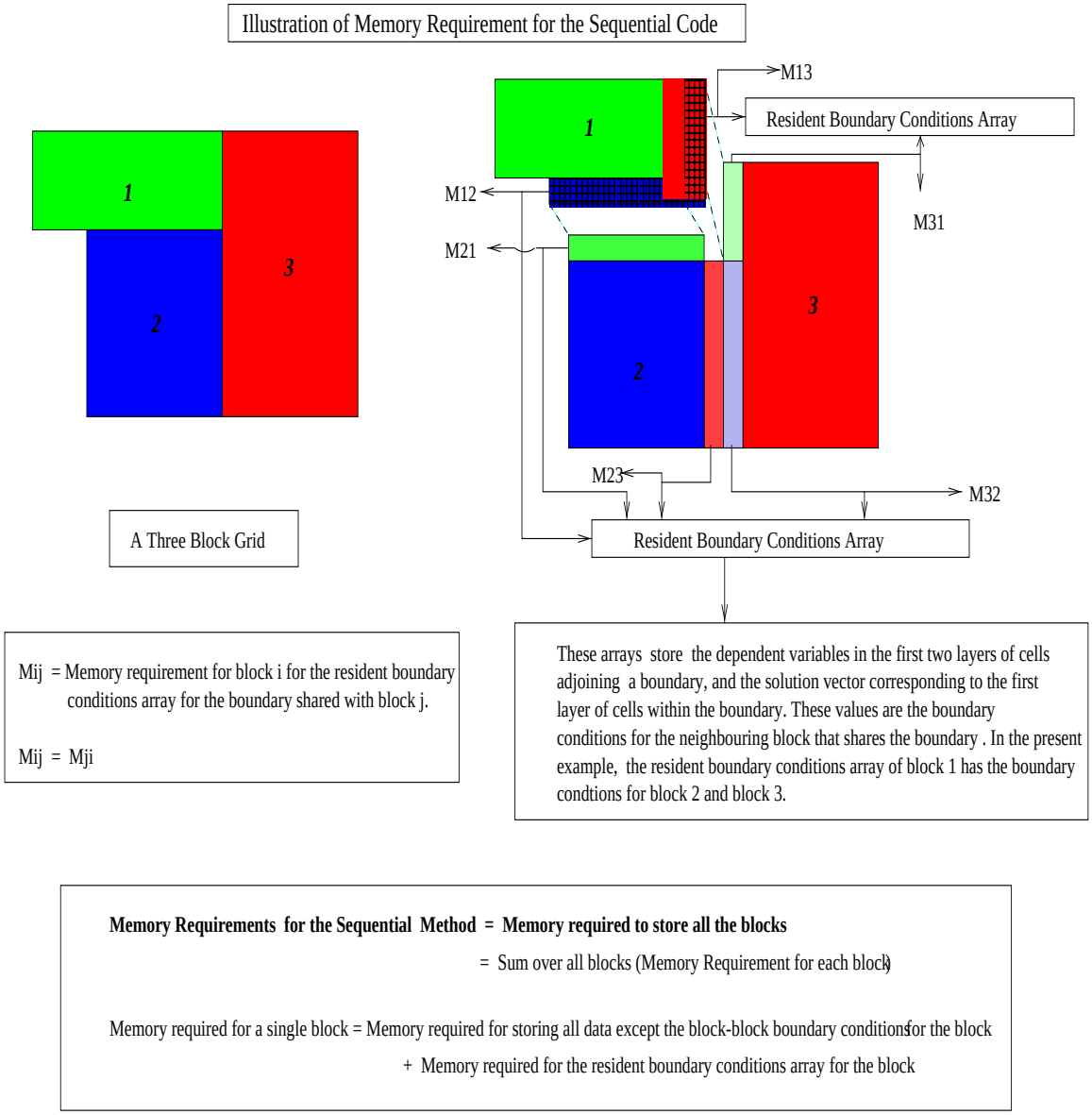

Figure 3.9: Memory allocation in the sequential case

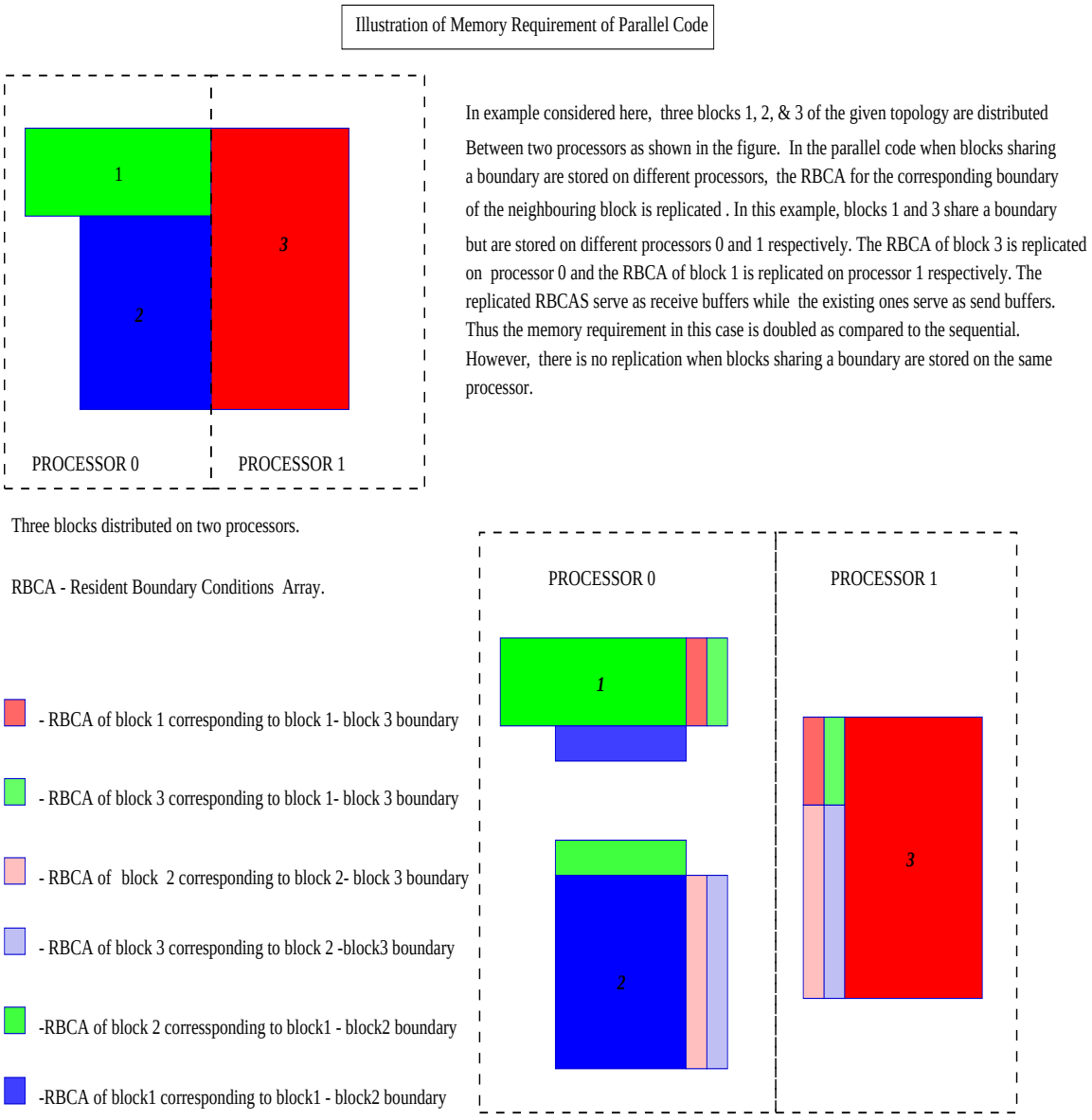

Figure 3.10: Memory requirements for the parallel code

CHAPTER IV

COMPARISON OF PARALLEL AND SEQUENTIAL RESULTS

The focus of the present work is on the flow solver part of CHEQNS. The validation was therefore limited to flow problems. The Gauss-Seidel iterations in the parallel implementation are block-decoupled and because of the parallel nature, the solution vector that is exchanged between the blocks that are stored on different processors lags the corresponding exchange in the case of multi-block sequential algorithm by a Newton iteration. Two important issues associated with the lagged exchange of solution vector are its influence on the correctness of the solution and the performance of the parallel algorithm in terms of convergence. The correctness of the solution was studied by comparing the sequential and parallel solutions for different test-cases. Convergence measurements indicate how far is a solution from steady state. In the case of steady state problems the L2-Norm of the residual at each time-step (the right hand side of the linearized system representing the flux balance across the faces of a control volume) gives an absolute measure of convergence. At steady-state the L2-Norm should be machine zero and its magnitude is an indication of the departure from steady-state. If the L2-Norm of the residual is normalized using the L2-Norm corresponding to the first time-step we get a relative measure for convergence and this value gives an indication on how fast can the solution algorithm drive the solution to steady-state. The lagged exchange of solution vector could affect the absolute and the relative convergences

and it could cost more in terms of number of iterations required to attain a given level of convergence. The estimates of absolute and relative convergence have been compared in two test-cases and will be presented later in this chapter. In the case of unsteady problems, the L2-Norm of the residuals corresponding to Newton iterations is an indication of the level of convergence within the given time-step. The rest of chapter presents a comparison of the results obtained from the sequential and parallel implementations with the perspectives discussed above.

The first test case considered was that of an inviscid transonic flow over NACA0012 airfoil at a Mach number of 0.755 and an angle of attack of 1.25^o at a Reynolds number of 8.5×10^5. A single block "C - O" grid of dimension 275 x 40 x 2 was generated with due consideration to orthogonality. This grid was then split in the i-direction into three blocks of dimensions 76 x 40 x 2, 125 x 40 x 2 and 76 x 40 x 2 respectively. The far field boundaries were placed at 40 chord lengths upstream of the airfoil and aft of the airfoil. The far field boundaries in directions normal to the upper and lower surfaces were also placed at forty chord lengths. The first spacing off the airfoil surface in the direction normal to the surface was placed at 0.001 chord lengths. A tight spacing was not necessary as only inviscid flow was being considered. The location of the shock on the upper surface was identified from the preliminary runs with the sequential code on the single block grid. The block cut on the upper surface was then made at the shock location so that the block cut lies in a region in where the flowfield is rapidly changing. The primary idea was to study the effect of the lagged exchanges near regions with sudden changes in flow field. Therefore, the obvious load imbalance arising from the different sizes of the blocks is not addressed here. The case was run using local time stepping at a CFL of 15 for 5000 time-step iterations. The surface plot of pressure co-efficient as obtained from the three runs viz., sequential single block, sequential

multi-block and parallel is presented in the Figure 4.1. These plots demonstrate that there is little or no influence due to the lagged exchange of solution vector on the solution. The absolute convergence shown in Figure 4.2 indicates a better convergence for the sequential multi-block and parallel algorithms over the single block sequential algorithm. The curves for the sequential multi-block and parallel algorithms exactly overlap. However, the relative convergence for the sequential single block is better than those of sequential multi-block and parallel algorithms as depicted in the Figure 4.3. This means that the sequential single block drives the solution to steady-state at a rate faster than both sequential multi-block and parallel algorithms.

The second test case considered was turbulent flow over a flat plate at high reynolds number. In this case the parallel solution is compared with the single block sequential. The single block grid was of dimension 158 x 60 x 3. The first spacing off the wall was 1.0 x10^{-6} times the plate-length. The farfield boundaries in the stream direction were located at seven plate-lengths from the center of the flat-plate in both upstream and downstream directions. The far field boundary in the direction normal to the plate was located at two plate-lengths. This grid was split in the stream direction into three blocks of dimensions 60 x 60 x 3, 60 x 60 x 3 and 40 x 60 x 3 respectively. Figure 4.4 shows the velocity profile, variation of U^+ with Y^+, as obtained from the sequential and parallel simulations at an observation point located at 0.98 plate-length from the leading edge. The absolute convergence shown in Figure 4.5 indicates a better convergence for the sequential multi-block and parallel algorithms over the single block sequential algorithm. The curves for the sequential multi-block and parallel algorithms exactly overlap. As opposed to the previous case, the relative convergence of the sequential multi-block and parallel algorithms is same as that

for the sequential single block algorithm for about 1000 time-step iterations as depicted in the Figure 4.6, after which the convergence rate for the single block sequential algorithm drops.

The third case involved steady viscous supersonic flow through the diverging section of a nozzle. The region of interest was both the nozzle section and the expansion plume. The grid consisted of two blocks of dimensions 81 x 41 x 2 and 88 x 41 x 2 whose k-surfaces are as shown in the Figure 4.7. The inlet condition at the throat was supersonic inflow at a temperature of 2748 K and a pressure of 17.03 atm. The k-planes are laterally shifted giving rise to a rectangular cross section for the nozzle, which resulted in an area ratio small enough to allow a high ambient pressure. The ambient temperature and pressure outside the nozzle were 250 K and 6000 Pa respectively. The sequential and the parallel cases were run for 10000 time-steps using minimum time-stepping with $dtmin = 0.001736$ with 1 newton iteration. The Figures 4.8 and 4.9 show the variation of centerline pressure ratio and temperature ratio with the area ratio. These variations correspond to the length along the centerline between the two arrow marks shown in Figure 4.7. The test-case was also used to verify the two different communication schemes discussed in the previous chapter. The solution pictures indicate that the lagged exchange does not affect the solution as opposed to the cylinder problem.

In order to test the performance of a code in the case of unsteady flows, the case of vortex shedding encountered in external flow over a cylinder was chosen. The grid used was a 4-block grid with each block of dimension 28 x 39 x 2 as shown in the Figure 4.11. The far field boundary was at 20 diameters from the center of the cylinder and the spacing off the wall of the cylinder for the first j-line was 10^{-4} diameters, with the diameter being 1 m. The sequential and the parallel codes were run at a Reynolds number of 6574 and a free stream Mach number of 0.3

with minimum time-stepping of $dtmin = 0.1736$ and with 3 Newton iterations and 3 Gauss-Seidel iterations . Figures 4.12 and 4.13 show a qualitative comparison of the Mach number distribution over the entire flow domain after 20,000 time-step iterations as obtained from the sequential and parallel codes respectively. The pictures display vortex shedding. The pictures indicate similar pattern for the distribution of Mach numbers though there are not exactly the same. The variation of the pressure coefficient at the observation point on the surface of the cylinder downstream of flow as shown in Figure 4.14 was used for comparing the results from the sequential and parallel codes. The comparison is shown in Figure 4.15 for a window of time. The time depicted is non-dimensionalized using the speed of sound and the diameter of the cylinder. The effect of lagged exchange of the solution vector reflects in these results. The $L2 - Norm$ of the Newton iteration residuals is shown in Figure 4.16 for the sequential and parallel cases at different time-steps. The L2-Norm for the parallel case shows a similar behavior like that of the sequential. The relative Newton iteration convergence of the parallel code as indicated by the L2-Norm is better than the sequential code.

The residual plots for sequential multiblock and the parallel implementation are in good agreement. However, these are off from the ones obtained using the sequential single-block as seen in the plots. The reasons for this is not understood.

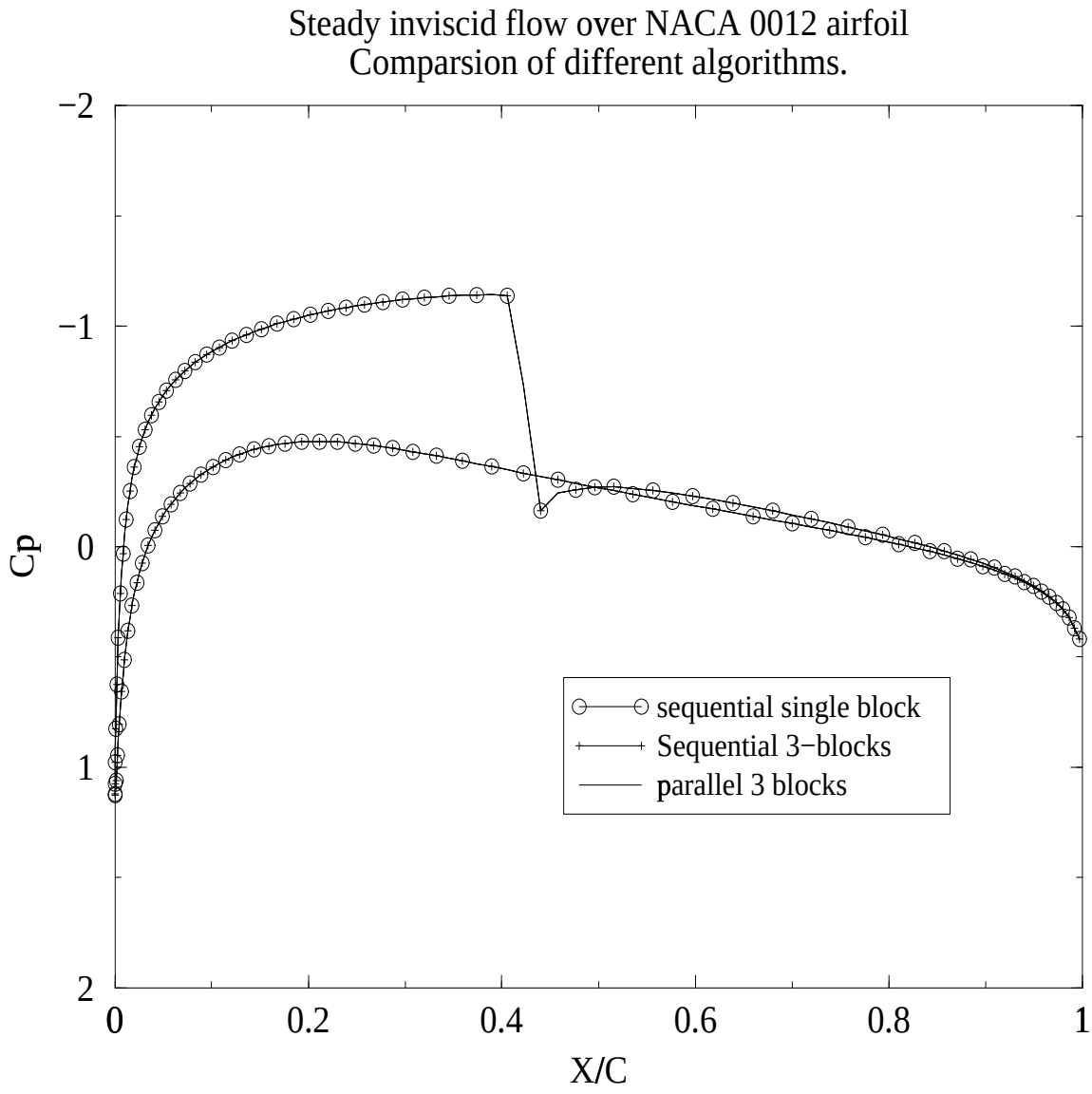

Figure 4.1: Surface pressure coefficient plot for NACA0012, $\alpha = 1.25^o, \text{Mach} = 0.755$

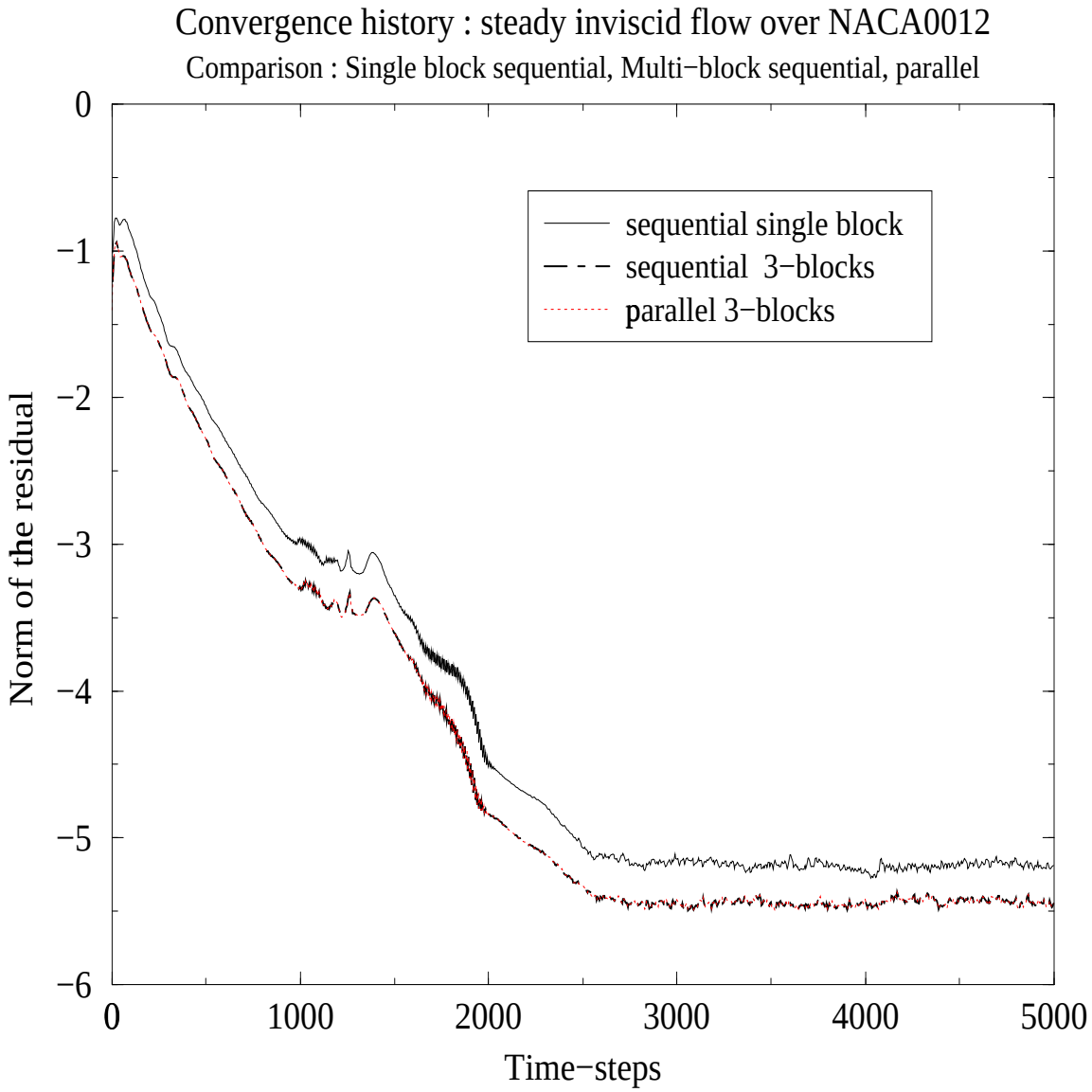

Figure 4.2: Absolute convergence measured in terms of L2-Norm(Residual) for inviscid flow over NACA0012

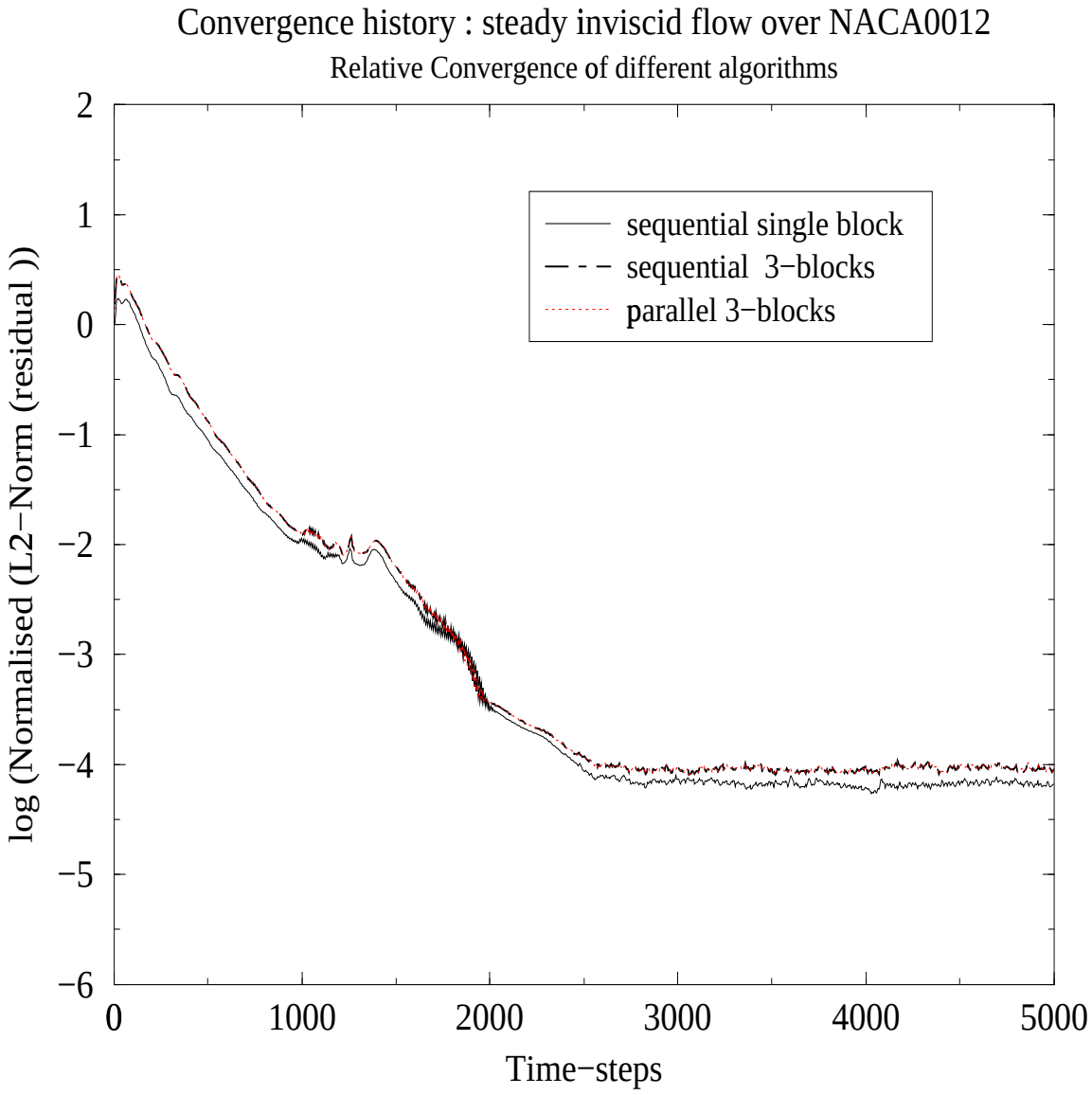

Figure 4.3: Relative convergence measured as L2-Norm(Normalized(Residual)) for inviscid flow over NACA0012

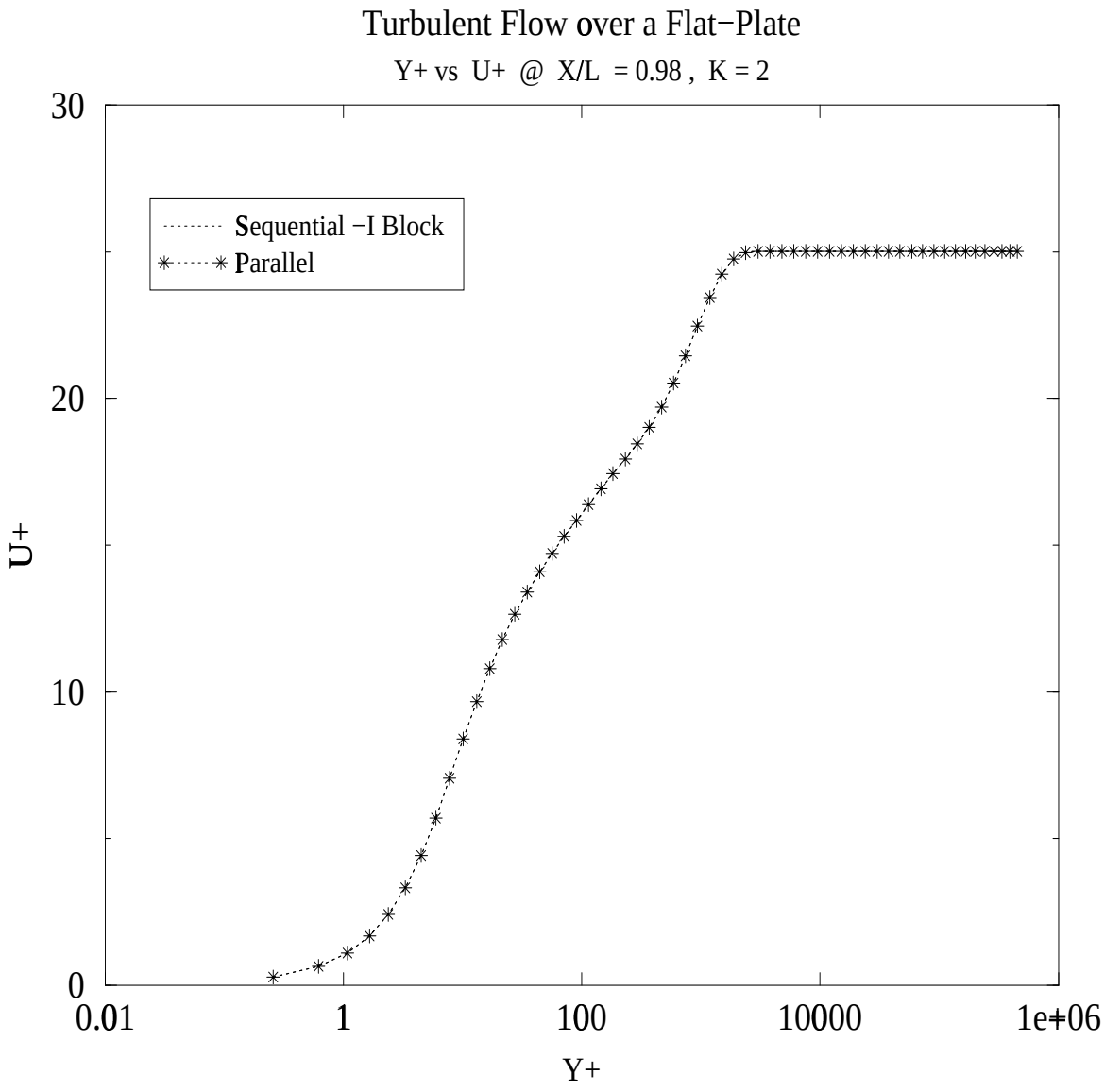

Figure 4.4: U^+ vs y^+ for turbulent flow over flat plate

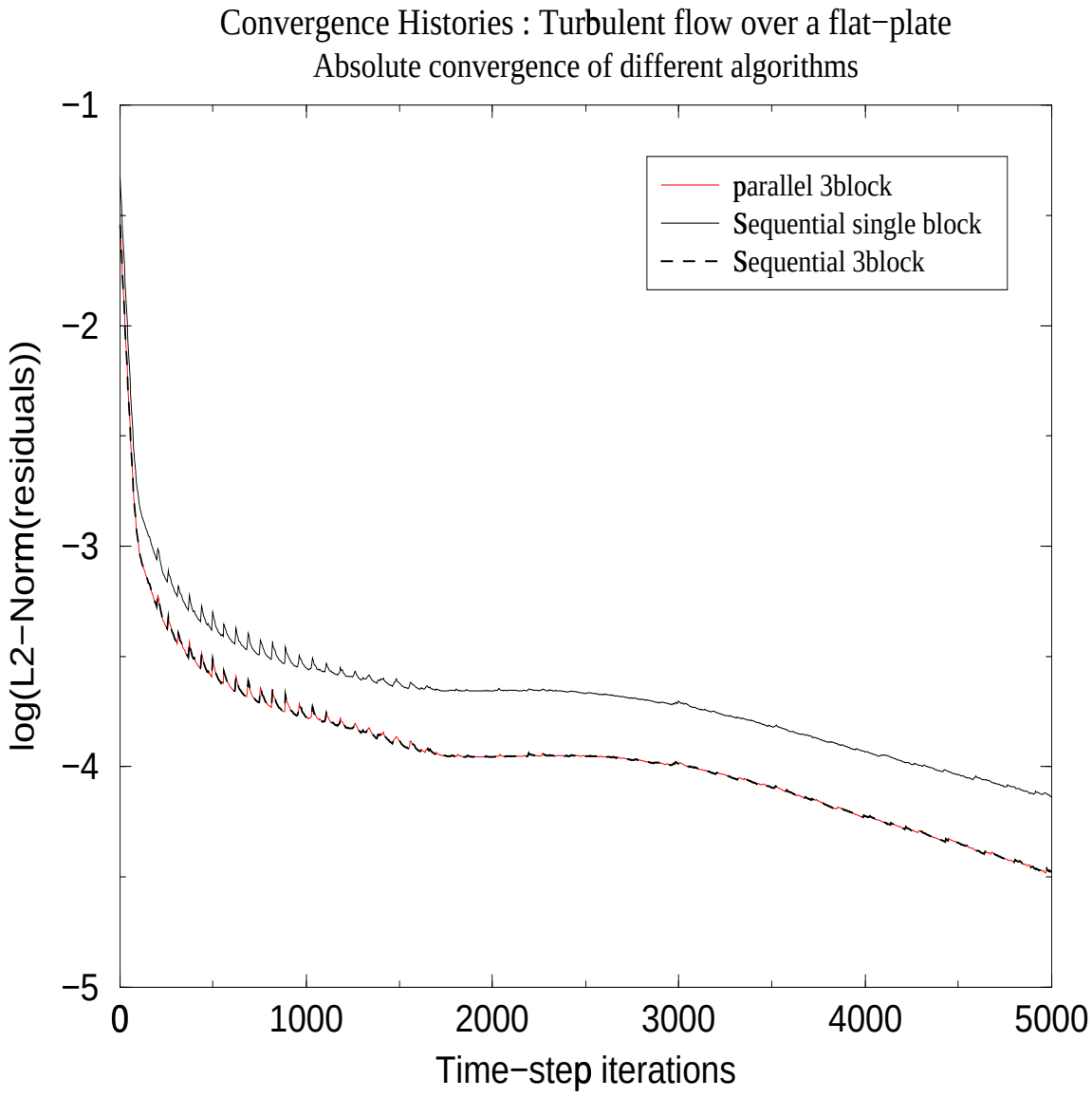

Figure 4.5: Absolute convergence measured in terms of L2-Norm(Residual) for turbulent flow over a flat plate

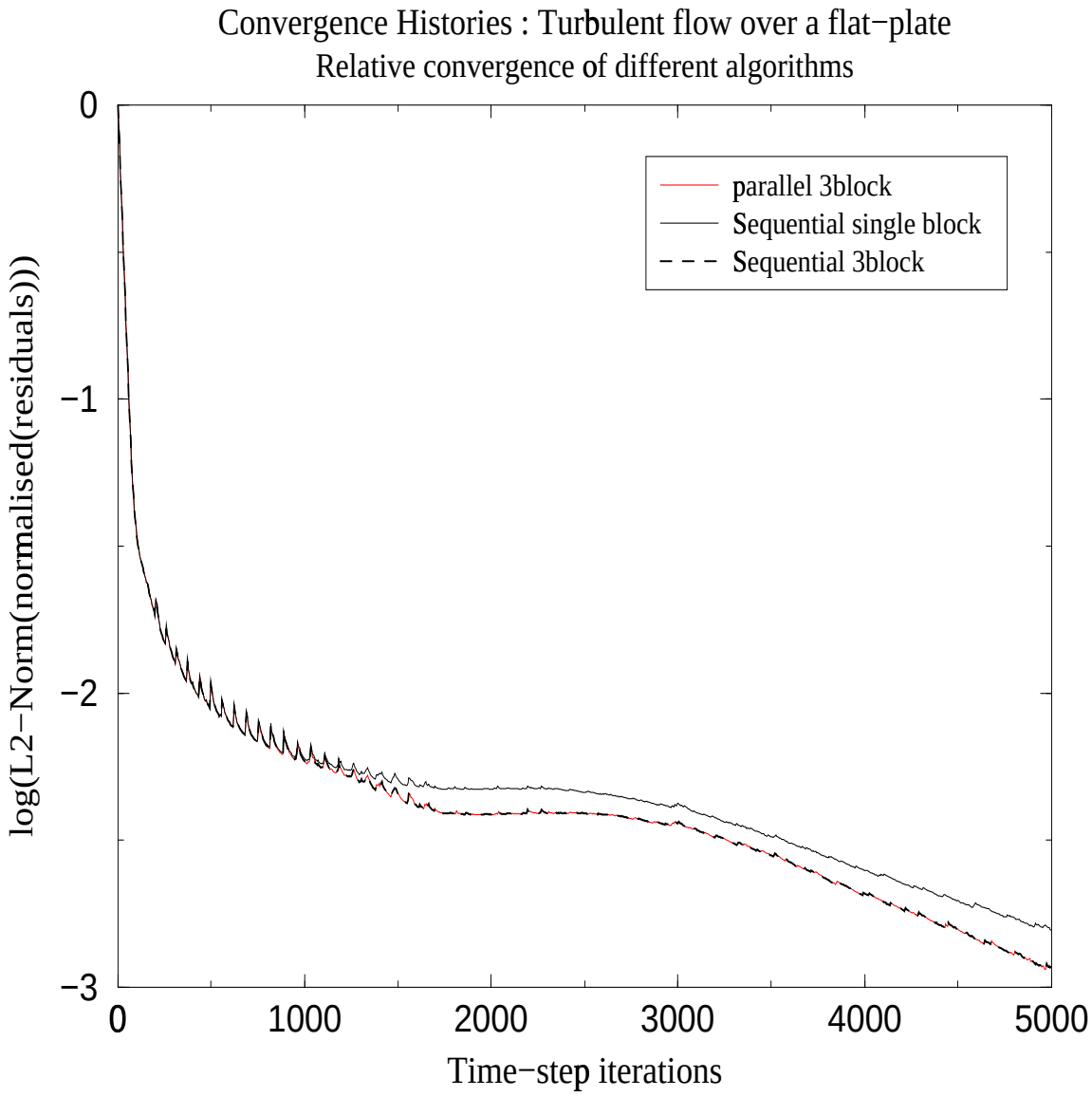

Figure 4.6: Relative convergence measured as L2-Norm(Normalized(Residual)) for turbulent flow over a flat plate

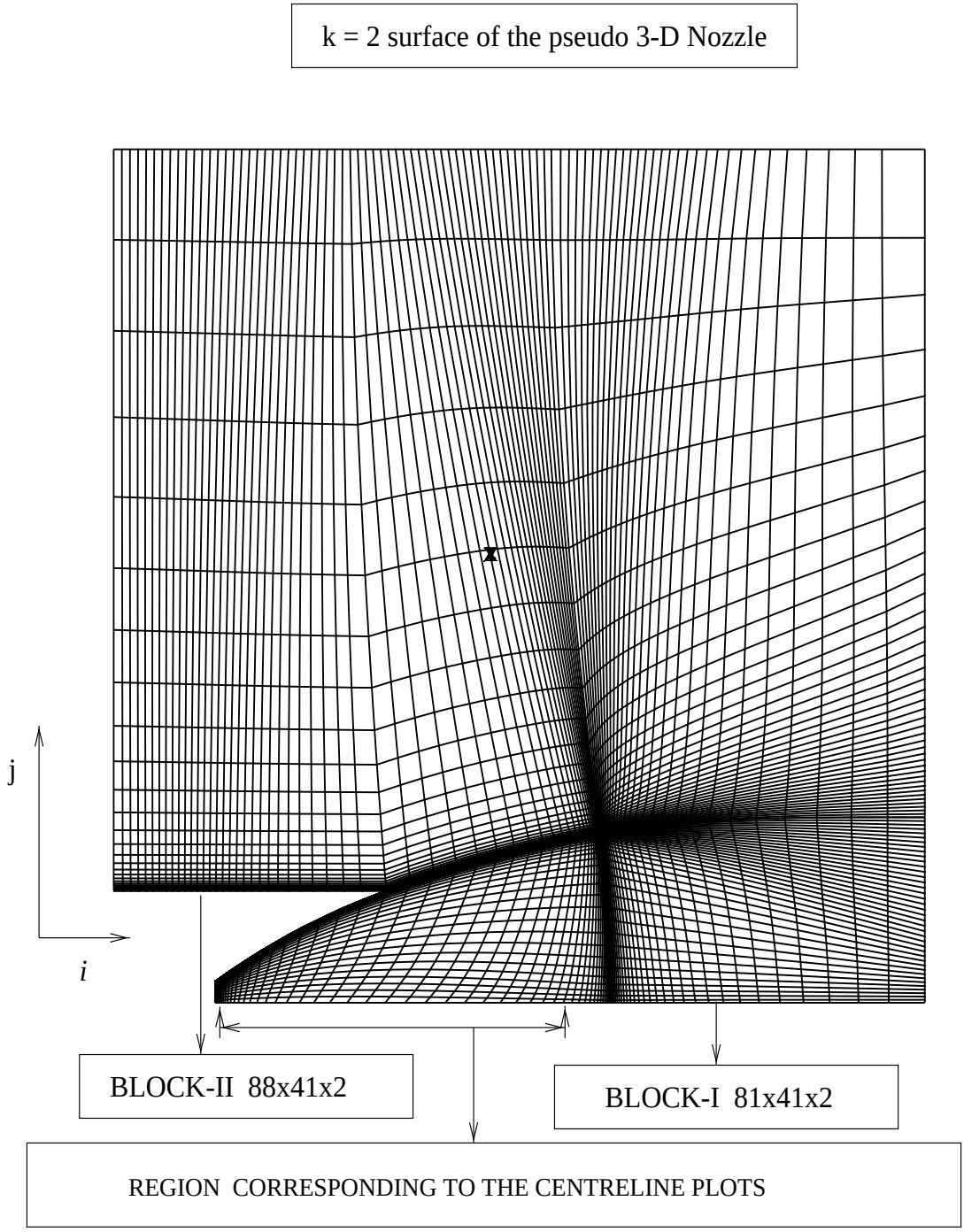

Figure 4.7: Nozzle grid showing blocking and the length along the nozzle corresponding to the centerline plots

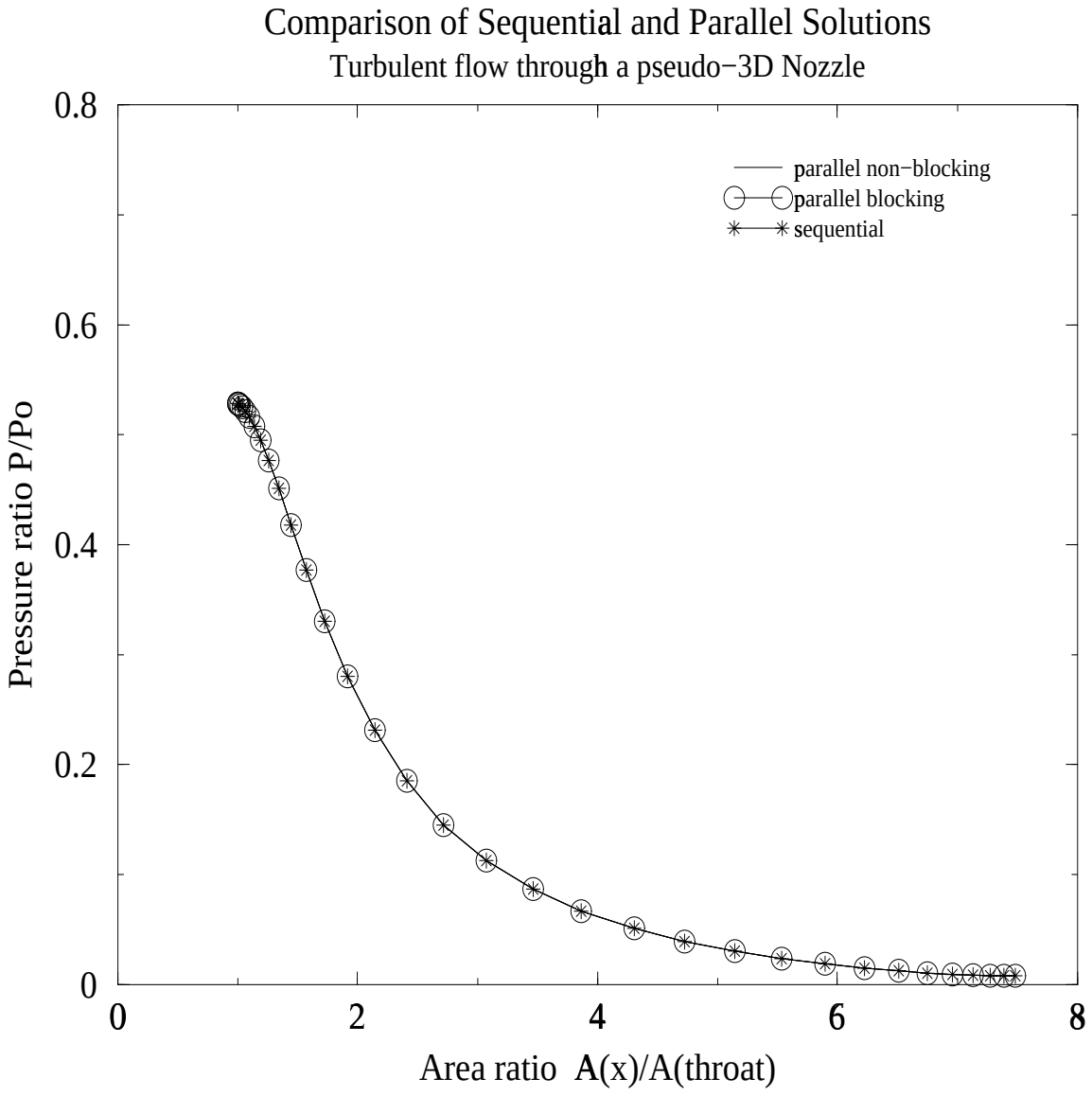

Figure 4.8: Centerline P/P_o vs area ratio

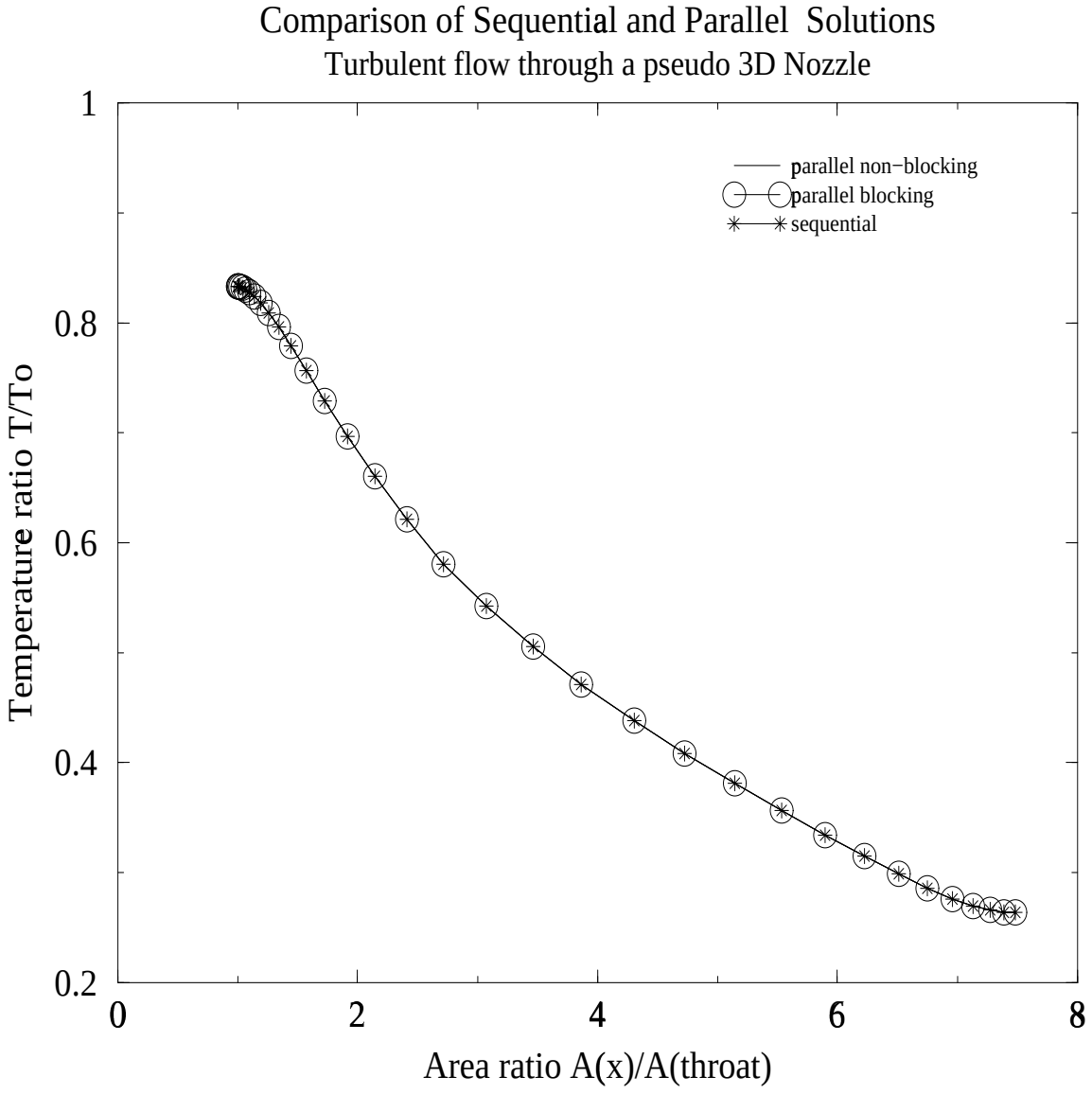

Figure 4.9: Centerline T/T_o vs area ratio

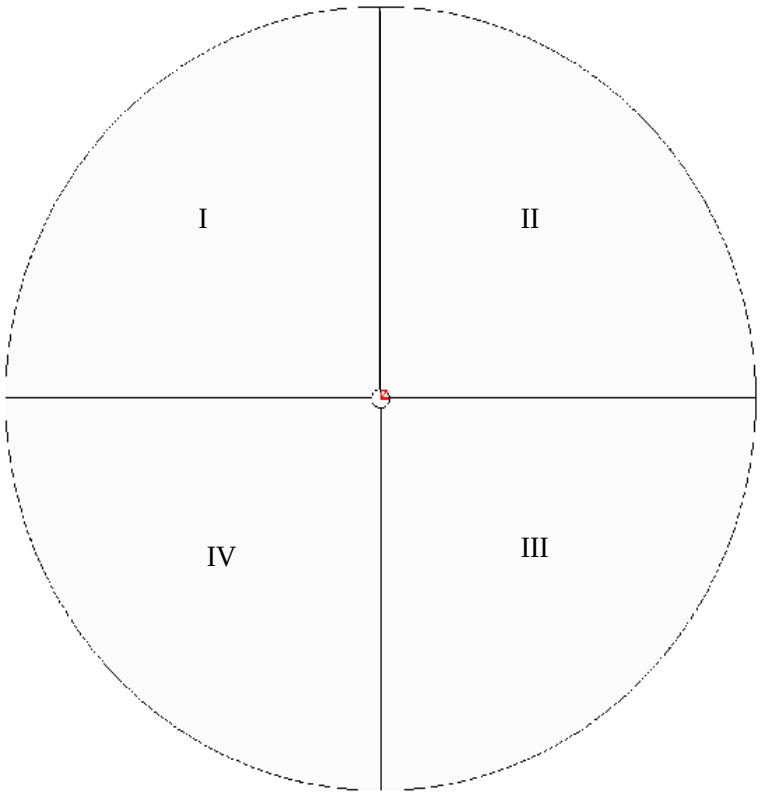

Figure 4.10: Four block 'O' grid around the cylinder

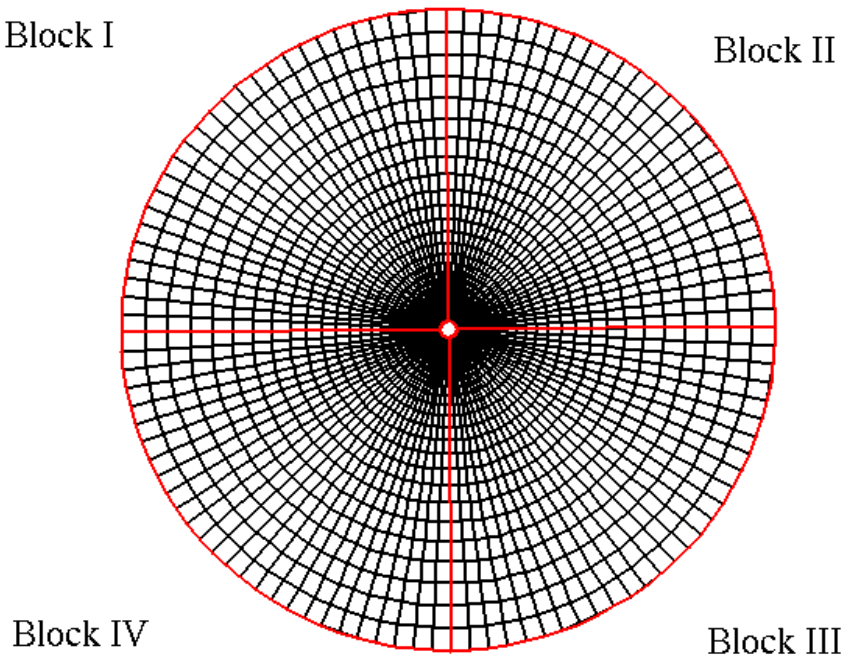

Figure 4.11: k-surface of the 'o' grid for the cylinder

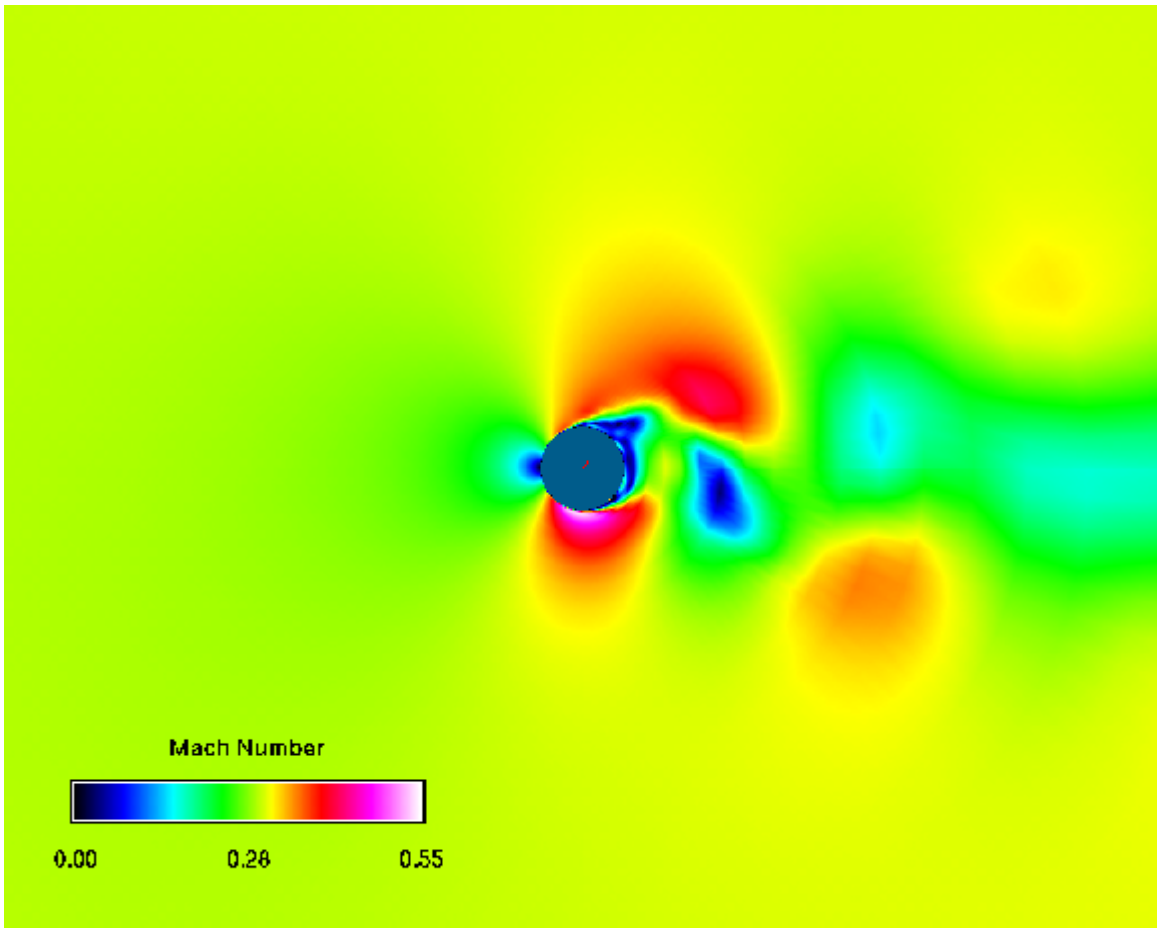

Figure 4.12: Mach number variation across the flow domain in the case of unsteady flow over a cylinder as obtained from the parallel code

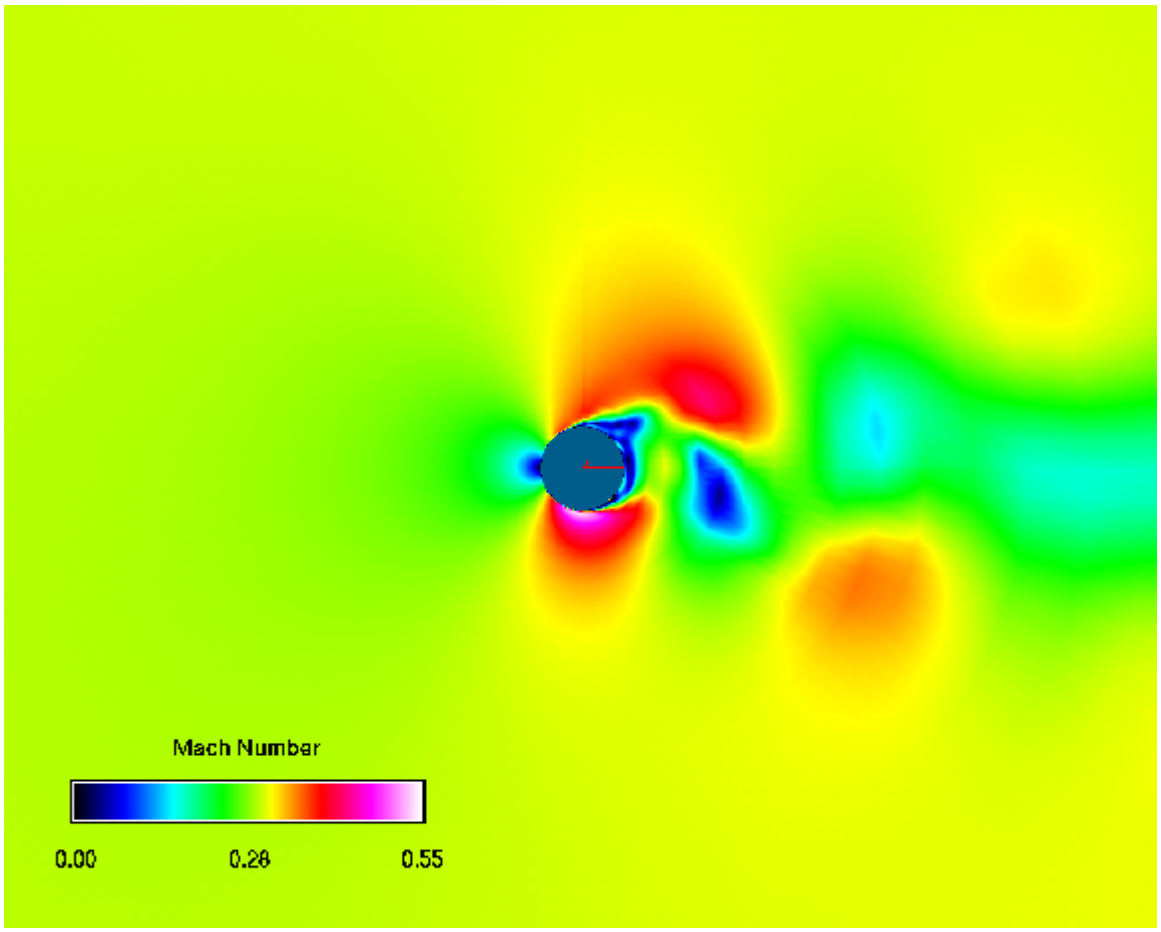

Figure 4.13: Mach number variation across the flow domain in the case of unsteady flow over a cylinder as obtained from the sequential code

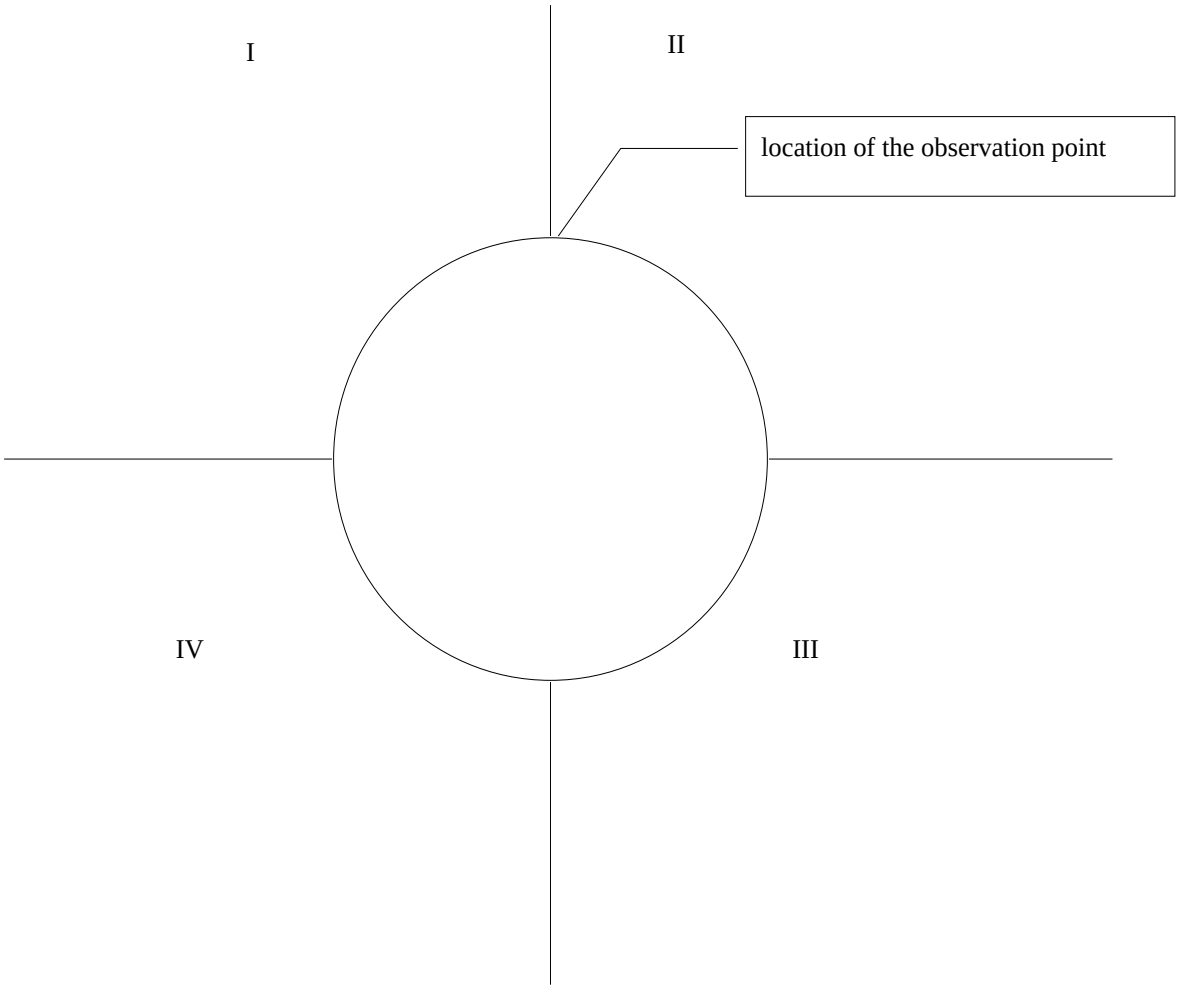

Figure 4.14: Location of the observation point in the domain

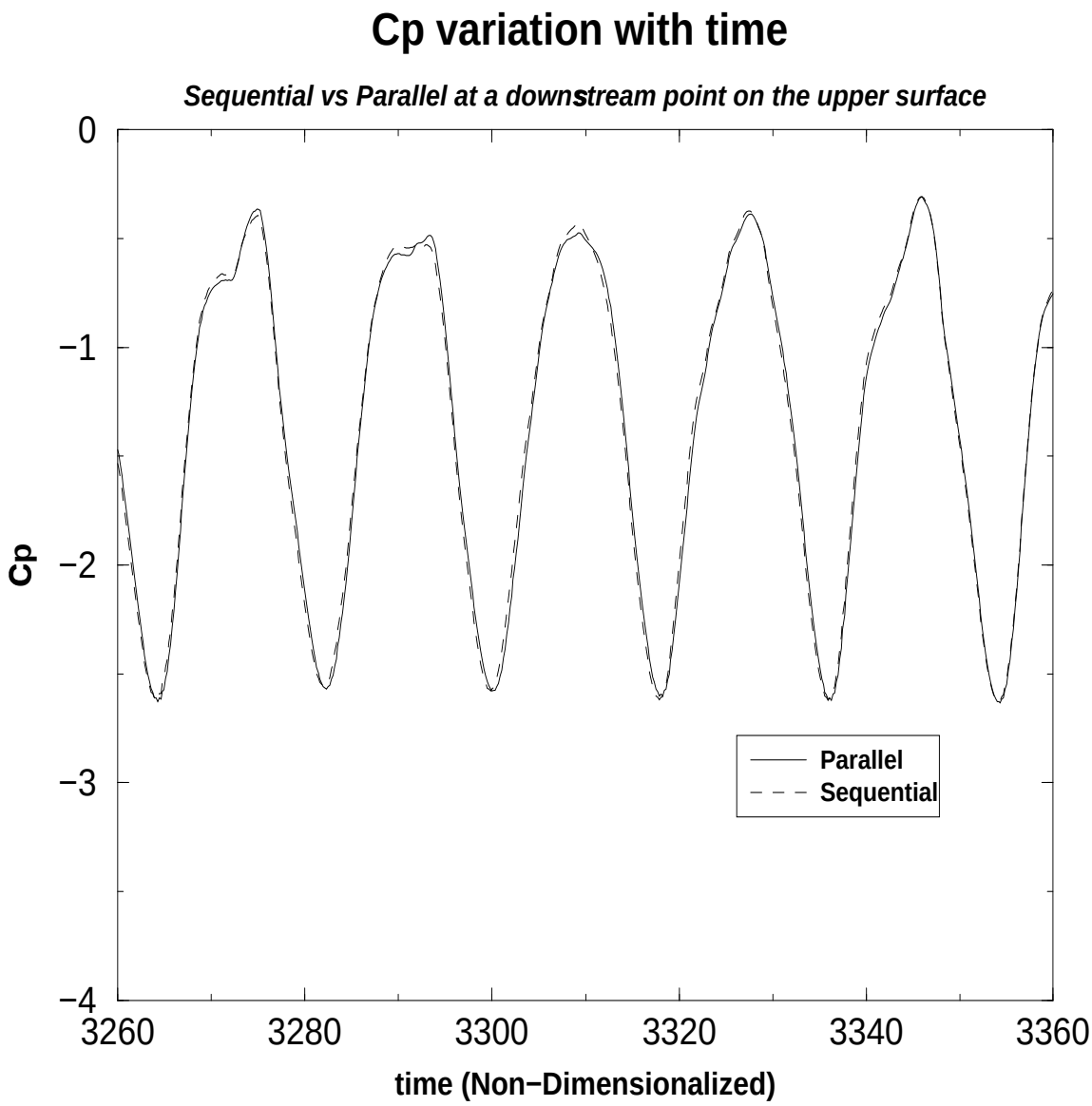

Figure 4.15: Time variation of pressure coefficient at an observation point on the upper surface of the cylinder

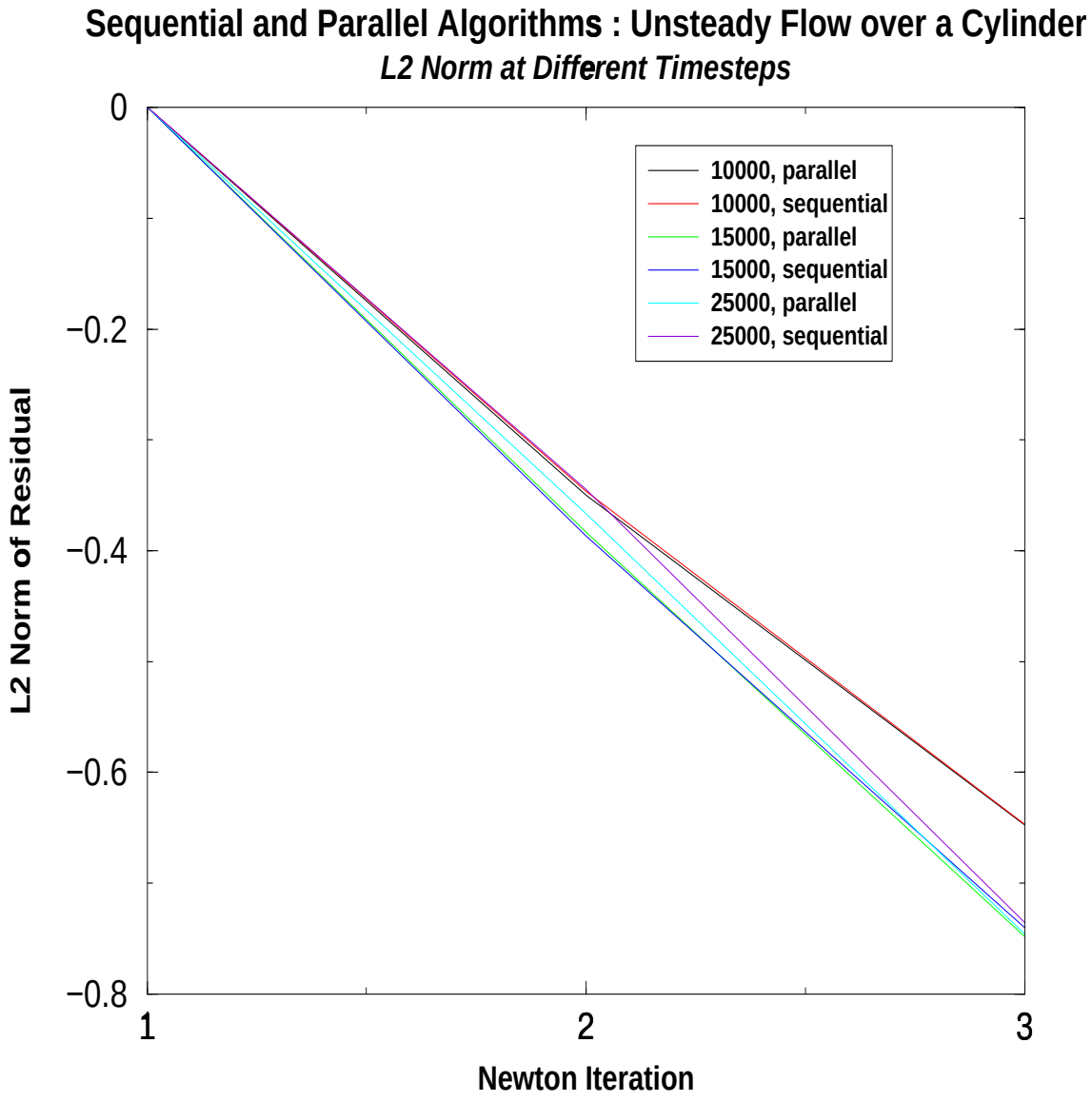

Figure 4.16: Comparison of L2-Norm of the residual for an unsteady flow over a cylinder for time-steps = 10000, 15000, 25000 ; dtmin = 0.1763 for sequential and parallel algorithms

CHAPTER V

PARALLEL ANALYSIS

Parallel algorithm together with the parallel architecture is known as parallel system. Performance of a parallel system is a measure of how well a system or its component accomplishes an assigned work. The definition of performance is strongly dependent on the perspective of the user. The different perspectives from which one could look at performance include problem size, time constraints, accuracy and cost. Problem sizes can vary over a wide range from problems that are fixed in size to those that can be scaled to use all the available resources. In solving a problem the time constraints could imply "*in as little a time as possible ...*" or " *within this time ...*". A parallel algorithm may not be able to reproduce the accuracy of the sequential solution and this is important in situations where inaccuracy cannot be tolerated and in such a case it might involve additional work to reproduce sequential accuracy. Cost is usually the constraining factor from the perspective of financial cost.

The two performance metrics that were used in the present work are speed-up and efficiency. *Speed-up* is defined as the ratio of the time required to solve a given problem using a single processor to that required for solving the same problem on many processors. If t_s represents the sequential time and t_p represents the time for solving the same problem on p processors then the speed-up S is given

by,

$$S = \frac{t_s}{t_p}$$

The ideal speed-up for a solving a problem on p processors is p. *Efficiency* is a measure of how efficiently the processors are being used, in terms of processor non-idle time. It is defined as the ratio of sequential cost to parallel cost and can be expressed as,

$$E = \frac{t_s}{pt_p}$$

where, E is the efficiency.

Scalability is associated with the change in the performance measures as the system characteristics are changed. Scalability analyses are used to identify limitations of parallel systems as the system-characteristics are changed. The characteristic that is usually varied is the number of processors. Different scaling models can be used in scalability analysis. The most widely used include Fixed Size Speed-up, Scaled Speed-up and Fixed Time Speed-up models. In the present work, a qualitative study of scalability is done along the lines of the Scaled Speed-up model. The Fixed Size Speed- up scale-up model is based on Amdahl's Law. In this model the speed-up is studied as a function of the number of processors for a fixed problem size. This usually reflects a rapid degradation in efficiency with increasing number of processors associated with the "immutable sequential fraction ... as defined by Amdahl". The Scaled Speed-up scale-up model is based on Gustafson's law. The problem is scaled at the same rate as the number of processors in this case. The primary idea here is to practically eliminate the immutable sequential fraction. Fixed Time Speed-up model is applicable to problems where there exists an upper bound on the run time for the parallel solution. The upper bound is the

time constraint and the model studies the largest problem size that can be solved using the available processors within this time constraint.

In the context of speed-up measurements it is important to define the sequential work. In the present work the sequential multi-block algorithm is the reference. Sequential work is defined in terms of computational time as the computational time required to run the sequential multi-block algorithm for a specified length of simulation time. The simulation time refers to the time scale associated with the problem being solved. The parallel implementation retains the sequential solution methodology and therefore has the same convergence properties associated with sequential multi-block algorithm. Therefore the solution to a flow problem takes the same amount of simulation time with both the sequential multi-block algorithm and the parallel implementation. The computational time required to simulate a problem of interest is primarily a function of the number of numerical computations. The dependency on the nature of problem manifests itself as the number of time-step iterations needed to generate the solution. But, for a fixed number of time-step iterations, the computational time depends on the block sizes and the type of computation being carried out like viscous or inviscid rather than the flow problem itself. Also, the same set of instructions are repeated each time-step. Hence, for timing measurements, the time needed to perform a fixed number of time-step iterations was measured.

The timing measurements were performed for a hypothetical flow in a cubical topology with 64 cubical blocks. The grid was generated by repetitively splitting a single block cubical grid. All boundaries other than block-block interface being treated as far-field. This grid does not represent a physical problem but it provides a large problem size with equal number of points in all the three directions. The measurements were made with two different block sizes, in one case

the blocks were of dimension 10 x 10 x 10 while in the other they were of dimension 19 x 19 x 19. The problem size also depends on the kind of simulation being carried out like viscous, inviscid, turbulent, thin layer Navier-Stokes and full Navier-Stokes because the number of computations is different in each type of simulation for a block of given dimensions. For each of the two different block sizes, two timing measurements were made one with inviscid calculations and the other with full Navier-Stokes with turbulence applied in all the three directions. The same set of measurements were performed on SUN Ultra HPC 10000 Supercomputer and SGI Challenge 10000 XL. The experiments for timing measurements could not be done on a dedicated machine. Hence, the I/O time was not included in the timing measurements as it is influenced by factors which could not be controlled.

Figures 5.1 and 5.2 show the variation of efficiency with increasing number of processors for two different problem sizes on two different architectures, and the result is presented for both inviscid and full Navier-Stokes turbulent simulations. The parallel efficiency of the full Navier-Stokes turbulent simulation is better than that of the inviscid simulation. This could be easily explained as the full Navier-Stokes turbulent simulations have higher computation to communication ratio for the same number of processors. The simulations associated with larger problem size, those associated with 64 blocks of dimension 19x19x19, display higher efficiencies when compared with those associated with smaller problem size for the same reason. The performance on the SUN Ultra HPC 10000 Supercomputer is much superior to that on SGI Challenge 10000 XL. There is a steep drop in efficiency for the 10 processors case on the SUN Ultra HPC 10000 Supercomputer and 6 processors case on the SGI Challenge 10000 XL. This is because of load imbalance arising from the scatter distribution of the 64 blocks on to 10 and 6 processors respectively. Figures 5.3 and 5.4 describe the same observations in

terms of speed-up. Figures 5.5 and 5.6 describe the scalability of the parallel implementation on two different architectures. Figure 5.5 indicates that on SUN Ultra HPC 10000 Supercomputer inviscid calculations are much more scalable than full Navier-Stokes calculations for the present parallel implementation. Figure 5.6 indicates that the scalability of the parallel implementation is poor on SGI Challenge 10000 XL.

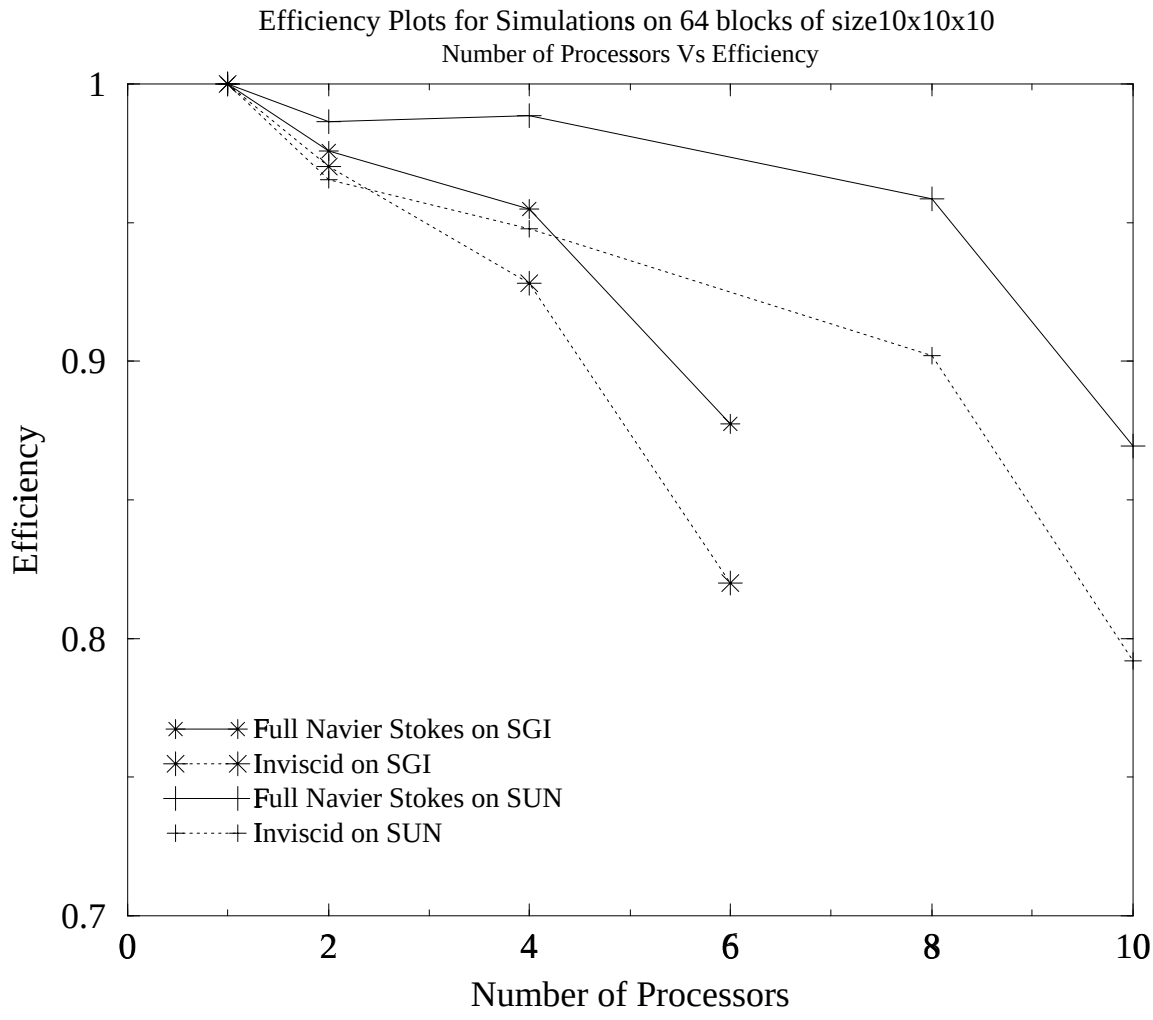

Figure 5.1: Efficiency plots for solving full Navier-Stokes and inviscid problems on 64 blocks of size 10x10x10 on SUN Ultra HPC 10000 and SGI Challenge 10000 XL architectures

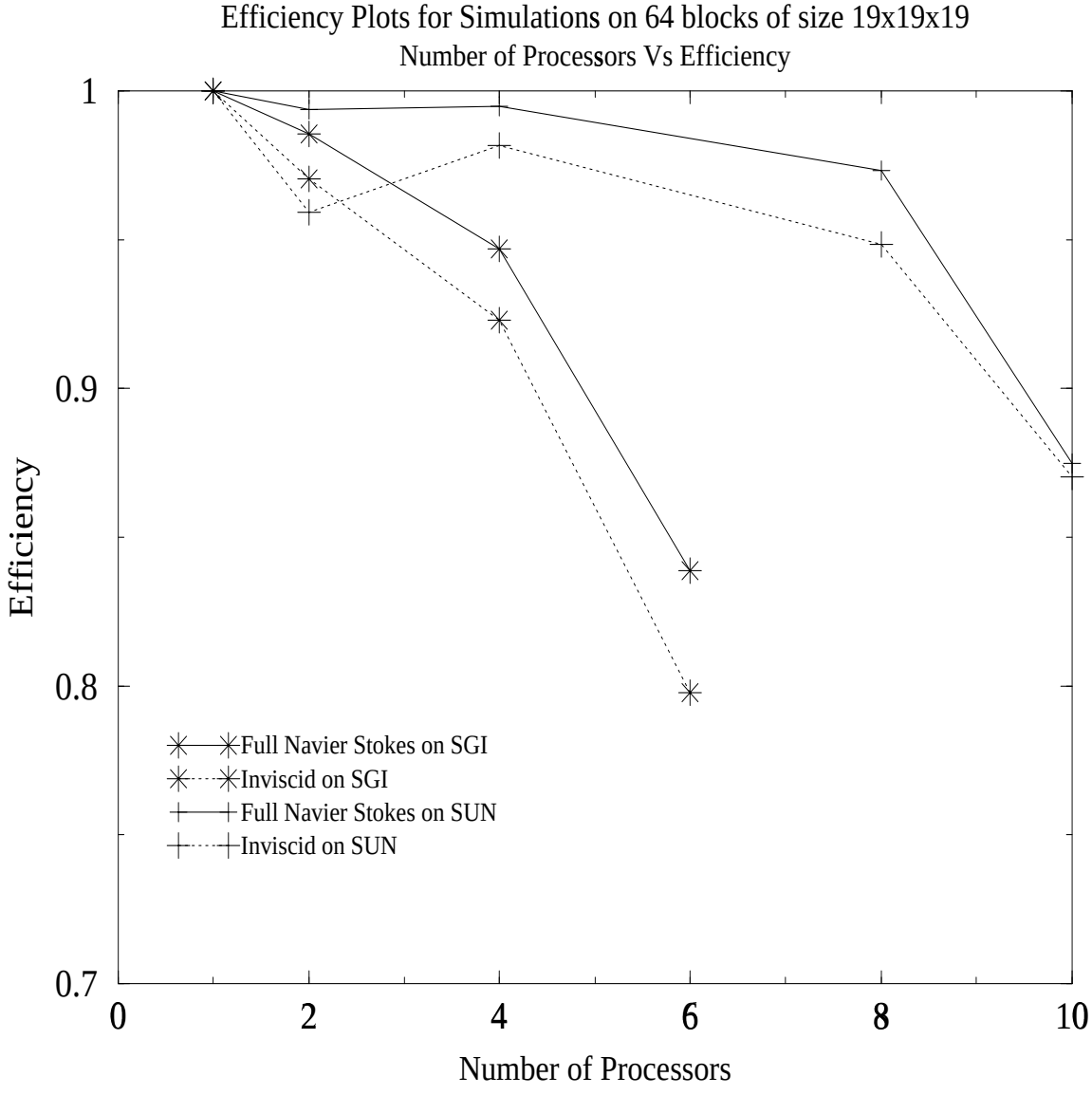

Figure 5.2: Efficiency plots for solving full Navier-Stokes and inviscid problems on 64 blocks of size 19x19x19 on SUN Ultra HPC 10000 and SGI Challenge 10000 XL architectures

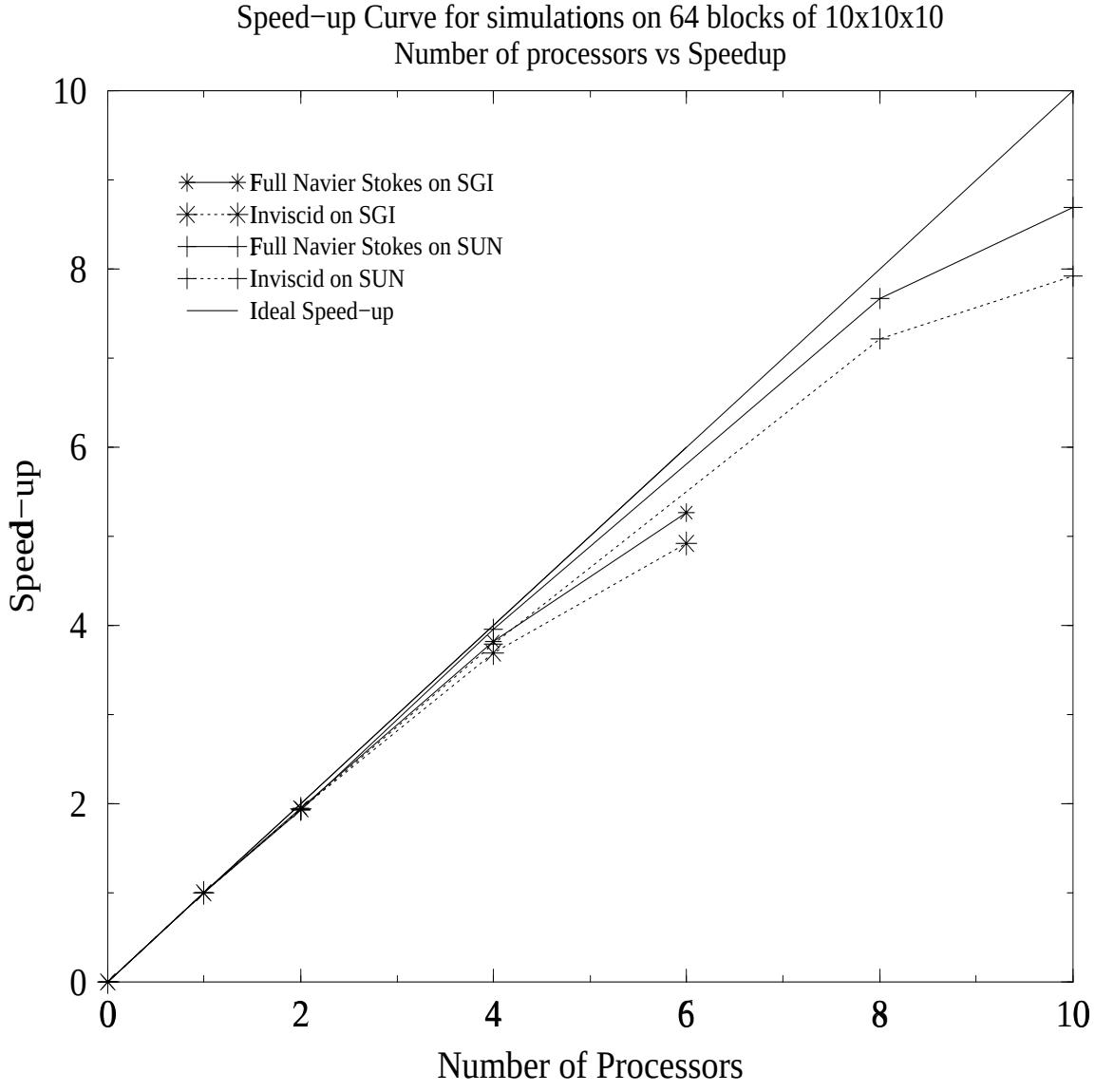

Figure 5.3: Speedup plots for solving full Navier-Stokes and inviscid problems on 64 blocks of size 10x10x10 on SUN Ultra HPC 10000 and SGI Challenge 10000 XL architectures

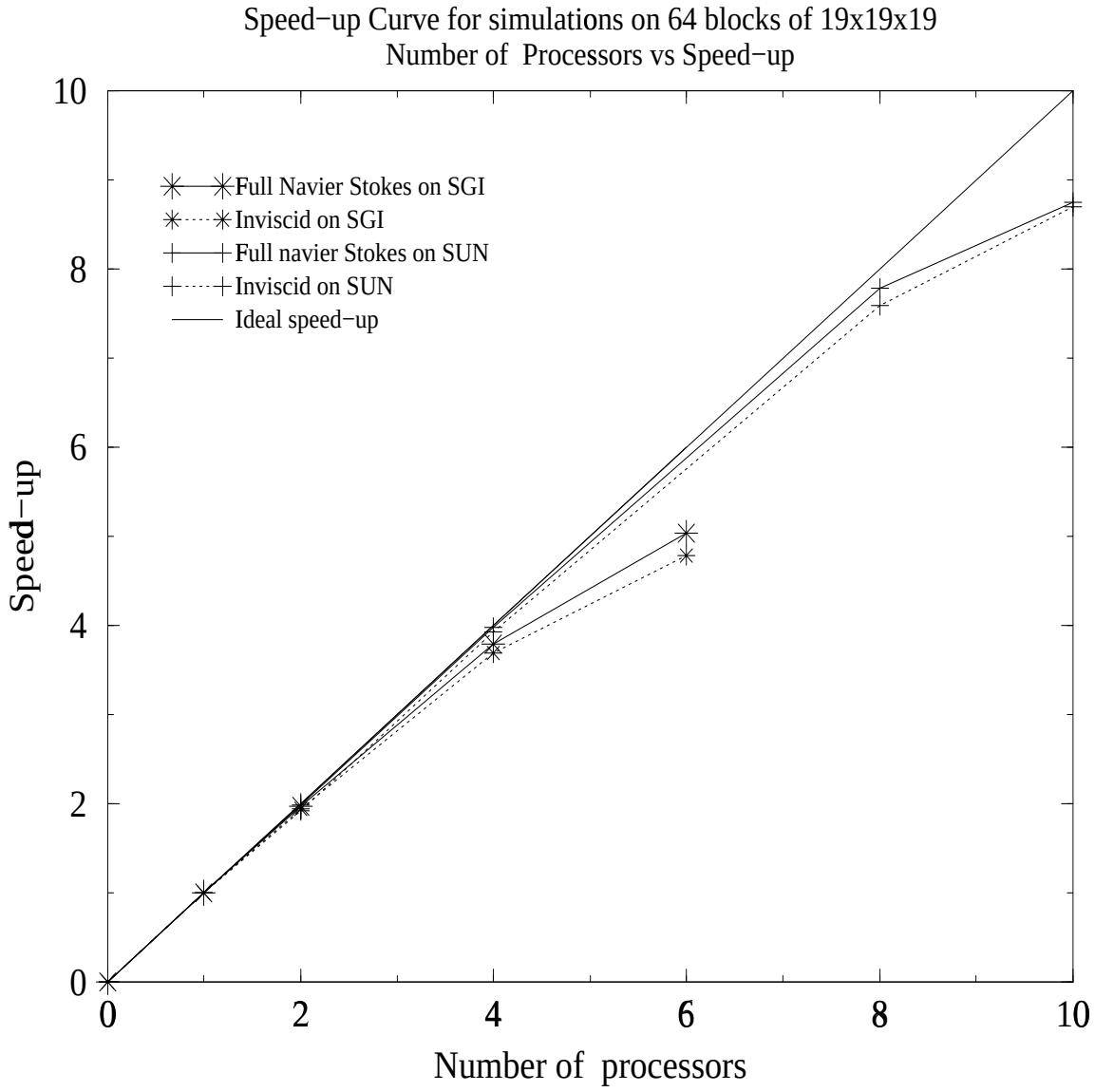

Figure 5.4: Speedup plots for solving full Navier-Stokes and inviscid problems on 64 blocks of size 19x19x19 on SUN Ultra HPC 10000 and SGI Challenge 10000 XL architectures

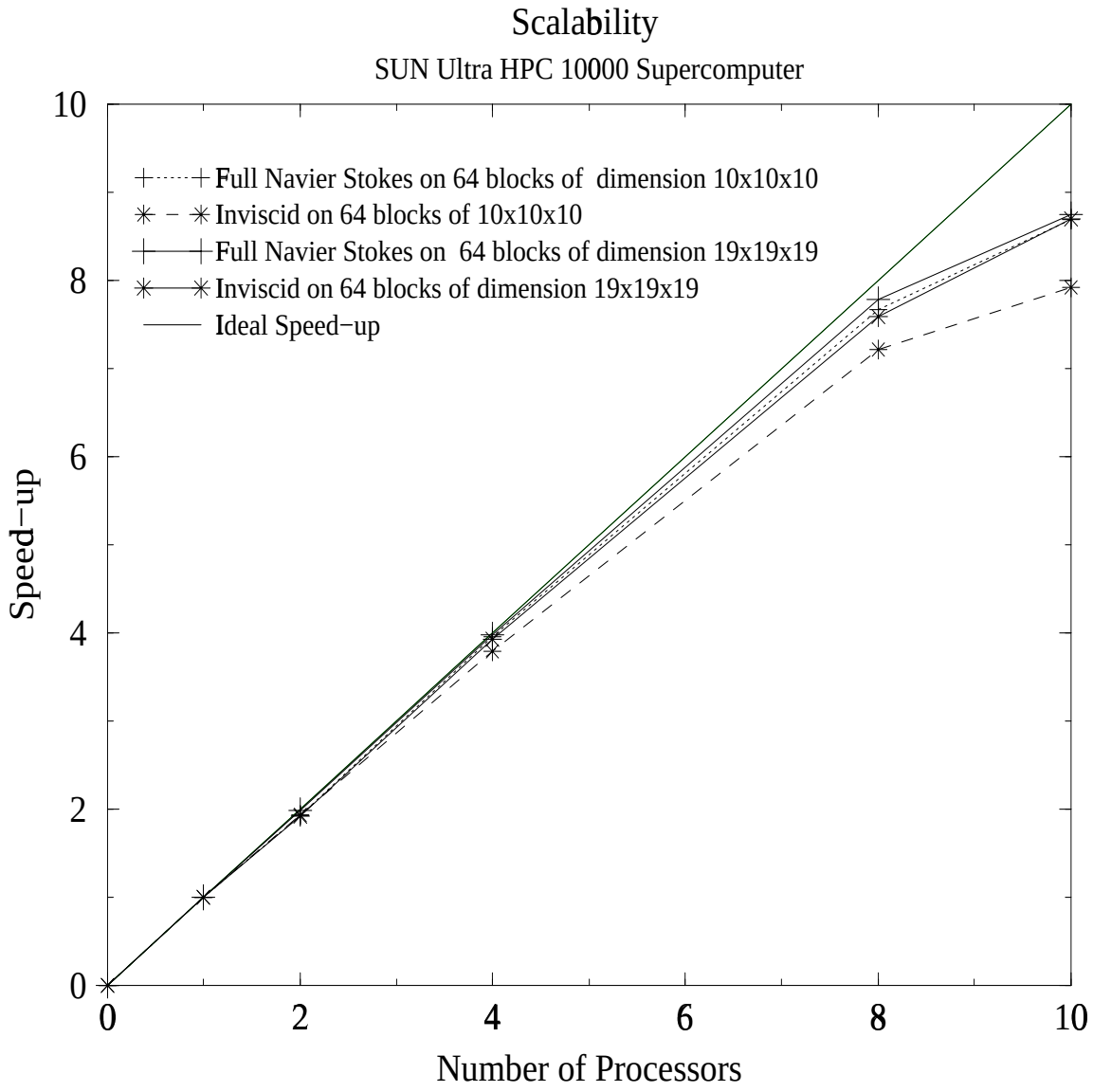

Figure 5.5: Qualitative measure of scalability of the parallel algorithm on SUN Ultra HPC architecture

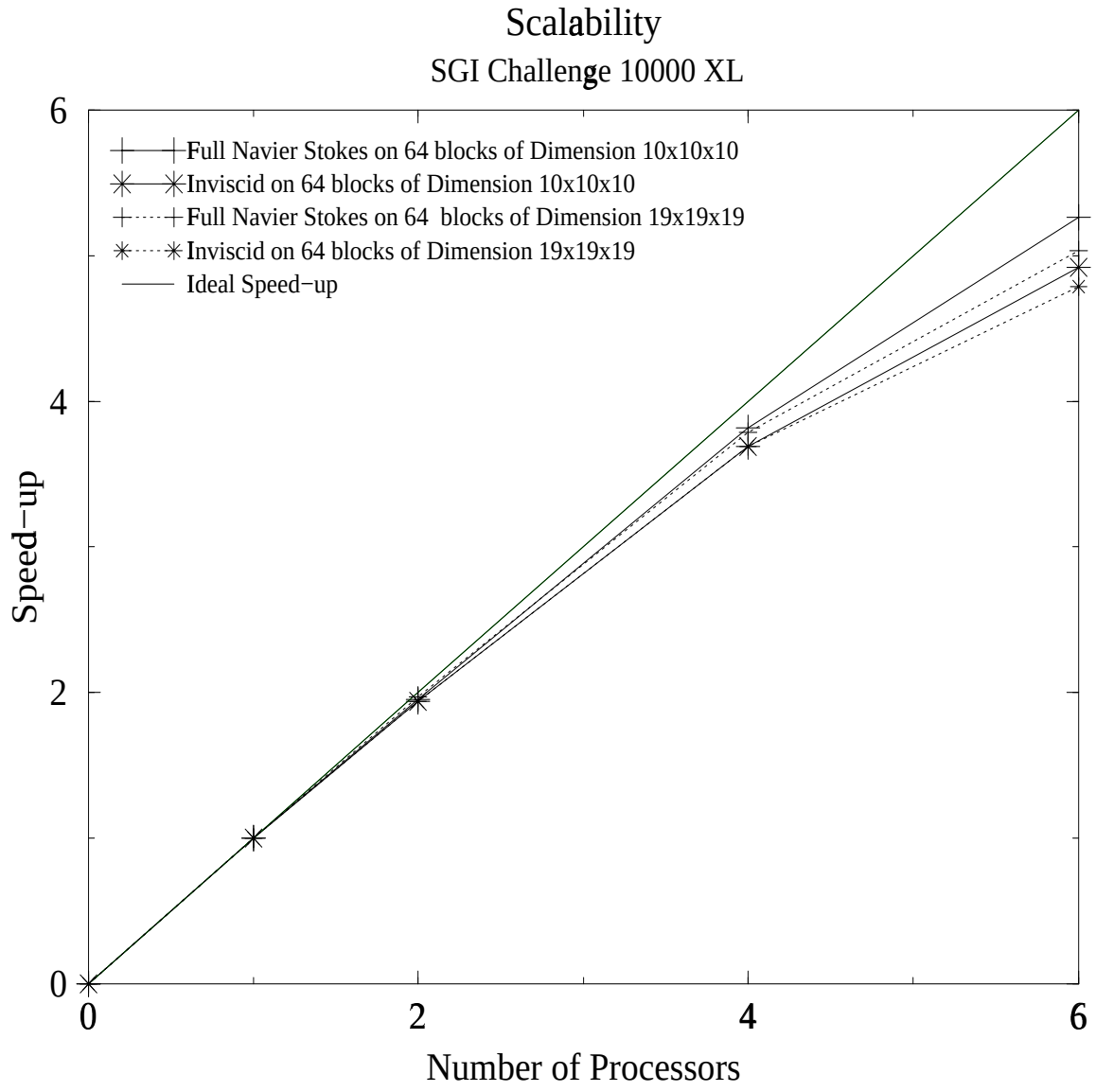

Figure 5.6: Qualitative measure of scalability of the parallel algorithm on SGI Challenge 10000 XL architecture

CHAPTER VI

CONCLUSIONS

A coarse-grain parallel algorithm for an implicit multi-block sequential solver for chemically reacting flows has been designed and implemented using the Message Passing Interface (MPI). The parallel implementation confirms to Single Program Multiple Processors (SPMD) model. Each processor has a copy of the executable in its memory. Rank 0 processor distributes the input data to all the processors. Each processor stores one or more blocks in its memory. The processors work independently and synchronize for exchanging block-block interface information. The parallel implementation is portable and runs on different architectures which support MPI.

The parallel implementation uses synchronous update of fluxes across the block-block boundaries. The solution algorithm consists of block-decoupled Gauss-Seidel iterations. The coupling between the sub-domains on different processors occurs at the newton iteration level. The parallelization is generic and can accept any arbitrary arrangement of blocks in multiblock configuration. The parallel code can handle multiple blocks per processor and has capabilities for restart on a different number of processors.

The parallel implementation has been verified against the results from the sequential multiblock solver for different types of flows. The parallel code shows good performance in terms of speed-up and efficiency up to 10 processors

for different problem sizes. However, its performance has not been tested with a greater number of processors. The parallel code does not implement any kind of load balancing algorithm and the performance characteristics are based on problems with little or no load imbalance. The performance in terms of convergence is same as that of the sequential multiblock solver. This could be improved by a tighter coupling between the sub-domains using block-coupled Gauss-Seidel iterations.

REFERENCES

[1] Belk, D., *Unsteady Three-Dimensional Euler Solutions on Dynamic Blocked Grids.* PhD thesis, Mississippi State University, August 1986.

[2] Carino, R. L., C. F. Cox, J. Zhu and P. Cinnella, "Parallel performance of a three-dimensional viscous multi-block real gas flow solver," *Proceedings of the First International Conference on Nonlinear Problems in Aviation and Aerospace*, pp. 103–108, 1996.

[3] Cox, C. F., *An Efficient Solver for Flows in Local Chemical Equilibrium.* PhD thesis, Mississippi State University, December 1992.

[4] Häuser, J., and R. Williams, "Strategies for parallelizing a navier-stokes code on the intel touchstone machines," *International Journal for Numerical Methods in Fluids*, vol. 15, pp. 51–58, 1992.

[5] Pankajakshan, R., and W. R. Briley, "Parallel solution of viscous incompressible flow on multi-block structured grids using mpi," *Parallel Computational Fluid Dynamics - Implementations and Results Using Parallel Computers*, pp. 601–608, 1996.

[6] Sawley, M. L., and J. K. Tegnér, "A comparison of different parallel programming models for computational fluid dynamics," *EPFL Supercomputing Review*, vol. 6, 1994.

[7] Webster, R. S., *A numerical study of the conjugate conduction-convection Heat Transfer Problem.* PhD thesis, Mississippi State University, December 2001.

[8] Whitfield, D. L., "Newton-relaxation schemes for nonlinear hyperbolic systems," *MSSU-EIRS-ASE-90-3,Mississippi State University*, 1990.

[9] Xiao, F., and T. Ebisuzaki, "Parallel implementation of a computational code for complex flows," *Computational Fluid Dynamics Journal*, pp. 13–18, 1999.

APPENDIX A

MODIFIED TWO-PASS SCHEME

The equations describing the fluid motion in the general curvilinear coordinates, $\xi = \xi(x, y, z, t)$, $\eta = \eta(x, y, z, t)$, $\zeta = \zeta(x, y, z, t)$, $\tau = t$ where x, y, z, t represent the Cartesian coordinates, written in the strong conservation form read as,

$$\frac{\partial Q}{\partial \tau} + \frac{\partial F}{\partial \xi} + \frac{\partial G}{\partial \eta} + \frac{\partial H}{\partial \zeta} = \frac{\partial F_v}{\partial \xi} + \frac{\partial G_v}{\partial \eta} + \frac{\partial H_v}{\partial \zeta} \tag{A.1}$$

The inviscid flux vectors are represented by F, G, H while their viscous counterparts are represented by F_v, G_v, H_v. The definitions for F, G, H, F_v, G_v, H_v and Q are presented in Chapter 2.

The implicit cell-centered finite volume formulation (with $\Delta\xi = \Delta\eta = \Delta\zeta = 1$) with a first order temporal accuracy, in the discretized integral form can be written as,

$$\frac{\Delta Q^n}{\Delta \tau} + \delta_i (F - F_v)^{n+1} + \delta_j (G - G_v)^{n+1} + \delta_k (H - H_v)^{n+1} = 0, \tag{A.2}$$

where $\Delta Q^n = Q^{n+1} - Q^n$, $n+1$ being the current time level , and δ_l is the central difference operator with $\delta_l = ()_{l+\frac{1}{2}} - ()_{l-\frac{1}{2}}$.

Applying upwind formulation the discretization (A.2) takes the form

$$\begin{aligned} \tfrac{\Delta Q^n}{\Delta \tau} + \quad & (\delta_i (F - F_v)^+ + \delta_i (F - F_v)^-)^{n+1} \\ + \quad & (\delta_j (G - G_v)^+ + \delta_j (G - G_v)^-)^{n+1} \\ + \quad & (\delta_k (H - H_v)^+ + \delta_k (H - H_v)^-)^{n+1} = 0. \end{aligned} \tag{A.3}$$

With the commonly used linearization strategy,

$$\begin{aligned} ((F - F_v)^\pm)^{n+1} &= ((F - F_v)^\pm)^n + A^\pm \Delta Q^n \\ ((G - G_v)^\pm)^{n+1} &= ((G - G_v)^\pm)^n + B^\pm \Delta Q^n \\ ((H - H_v)^\pm)^{n+1} &= ((H - H_v)^\pm)^n + C^\pm \Delta Q^n \end{aligned} \tag{A.4}$$

where

$$A^{\pm} = \left(\frac{\partial (F - F_v)^{\pm}}{\partial Q}\right)^n,\ B^{\pm} = \left(\frac{\partial (G - G_v)^{\pm}}{\partial Q}\right)^n,\ C^{\pm} = \left(\frac{\partial (H - H_v)^{\pm}}{\partial Q}\right)^n,$$

(A.3) can be written as,

$$\left(\frac{I}{\Delta\tau} + \delta_i A^+_{\cdot} + \delta_i A^-_{\cdot} + \delta_j B^+_{\cdot} + \delta_j B^-_{\cdot} + \delta_k C^+_{\cdot} + \delta_k C^-_{\cdot}\right)^n \Delta Q^n = -R^n \quad \text{(A.5)}$$

with $R^n = \delta_i F^n + \delta_j G^n + \delta_k H^n$ and "." implying matrix-vector multiplication of the appropriate terms.

Define

$$\begin{aligned} \delta_l M^+_{\cdot} &= \delta_i A^+_{\cdot} + \delta_j B^+_{\cdot} + \delta_k C^+_{\cdot} \\ \delta_l M^-_{\cdot} &= \delta_i A^-_{\cdot} + \delta_j B^-_{\cdot} + \delta_k C^-_{\cdot} \end{aligned} \quad \text{(A.6)}$$

Using these definitions, (A.5) can be written as

$$\left[\frac{I}{\Delta\tau} + \delta_l M^+_{\cdot} + \delta_l M^-_{\cdot}\right] \Delta Q^n = -R^n \quad \text{(A.7)}$$

Expanding (A.7) and using the definition,

$$D = \frac{I}{\Delta\tau} + M^+ - M^- \quad \text{(A.8)}$$

we obtain,

$$D_l \Delta Q^n_l - M^+_{l-1} \Delta Q^n_{l-1} + M^-_{l+1} \Delta Q^n_{l+1} = -R^n_l \quad \text{(A.9)}$$

Assuming D to be nonsingular, multiplying (A.9) by D^{-1} and rearranging we get

$$\left(I - D_l^{-1} M^+_{l-1} + D_l^{-1} M^-_{l+1}\right) \Delta Q^n = -R^n_l \quad \text{(A.10)}$$

The equation (A.10) is approximately factorised as

$$\left(I - D_l^{-1} M_{l-1}^{+}\right)\left(I + D_l^{-1} M_{l+1}^{-}\right)\Delta Q^n = -R_l^n \tag{A.11}$$

The equation (A.11) can be solved in the following forward and backward passes

$$\begin{aligned} \left(D_l - M_{l-1}^{+}\right) X &= -R^n \\ \left(D_l + M_{l+1}^{-}\right)\Delta Q^n &= D_l X \end{aligned} \tag{A.12}$$

The equations represented in (A.12) constitute the modified two-pass scheme. It should be noted that D is a block diagonal matrix, M^+ is a block lower triangular matrix with zeros on the diagonal, and M^- is a block upper diagonal matrix with zeros on the diagonal.