PARALLELIZATION OF THE EULER EQUATIONS ON UNSTRUCTURED GRIDS

by

Christopher William Stuteville Bruner

Dissertation submitted to the faculty of

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

DOCTOR OF PHILOSOPHY

in

AEROSPACE ENGINEERING

APPROVED:

_____

Robert W. Walters, Chairman

_____	_____
Bernard Grossman	Joseph A. Schetz
_____	_____
William L. Devenport	Joseph G. Hoeg

May 1996

Blacksburg, Virginia

Keywords: Computational Fluid Dynamics, Parallel Algorithms, Unstructured Grids.

PARALLELIZATION OF THE EULER EQUATIONS ON UNSTRUCTURED GRIDS

by

Christopher William Stuteville Bruner

Robert W. Walters, Chairman

Aerospace Engineering

(ABSTRACT)

Several different time-integration algorithms for the Euler equations are investigated on two distributed-memory parallel computers using an explicit message-passing paradigm: these are classic Euler Explicit, four-stage Jameson-style Runge-Kutta, Block Jacobi, Block Gauss-Seidel, and Block Symmetric Gauss-Seidel. A finite-volume formulation is used for the spatial discretization of the physical domain. Both two- and three-dimensional test cases are evaluated against five reference solutions to demonstrate accuracy of the fundamental sequential algorithms. Different schemes for communicating or approximating data that are not available on the local compute node are discussed and it is shown that complete sharing of the evolving solution to the inner matrix problem at every iteration is faster than the other schemes considered. Speedup and efficiency issues pertaining to the various time-integration algorithms are then addressed for each system. Of the algorithms considered, Symmetric Block Gauss-Seidel has the overall best performance. It is also demonstrated that using parallel efficiency as the sole means of evaluating performance of an algorithm often leads to erroneous conclusions; the clock time needed to solve a problem is a much better indicator of algorithm performance. A general method for extending one-dimensional limiter formulations to the unstructured case is also discussed and applied to Van Albada's limiter as well as Roe's Superbee limiter. Solutions and convergence histories for a two-dimensional supersonic ramp problem using these limiters are presented along with computations using the limiters of Barth & Jesperson and Venkatakrishnan — the Van Albada limiter has performance similar to Venkatakrishnan's.

Dedication

to my wife and children: Susan, Kristin, Justin, and Erin

Acknowledgments

I would like first to acknowledge the boundless patience and loving support of my wife Susan. Without her unending encouragement and prayers through all my academic pursuits, I could never have completed this work. I am deeply indebted to my academic advisor, Dr. Robert Walters. A very small amount of his enormous expertise and insight has (I hope) rubbed off on me, and it has been a pleasure to work under his tutelage. I would also like to thank the United States Navy, the Office of Naval Research, and the Naval Air Warfare Center for providing financial support for this work through the Long-term Training and In-house Laboratory Independent Research programs. Finally, I am tremendously grateful to Dr. Asha Varma of the Naval Air Warfare Center for believing in me and being my advocate when it seemed no one else would.

Thanks to Dr. Curtis R. Mitchell of CFD Research Corp. for his ONERA M6 grid and to Dr. William D. McGrory of AeroSoft, Inc. for his grid generators. Thanks also to Dr. Neal T. Frink of NASA Langley for the Hummel delta wing grids and to Dr. Andrew G. Godfrey, also of AeroSoft, for many enlightening (to me, anyway) discussions pertaining to CFD.

This work was supported in part by grants of high-performance computer time from the Department of Defense High-Performance Computing Modernization Office at the Wright-Patterson Air Force Base High-Performance Computing Major Shared Resource Center and at the Maui High-Performance Computing Center.

Table of Contents

Nomenclature

$\boldsymbol{A}^+, \boldsymbol{A}^-$	Flux jacobian matrices at a face
B	Baseline number of nodes for parallel performance
c_p	Specific heat at constant pressure
c_v	Specific heat at constant volume
$\boldsymbol{D}$	Diagonal block in the matrix problem arising in Euler implicit time integration
e_0	Stagnation energy per unit mass
$\boldsymbol{F}$	Flux vector
$\boldsymbol{F}_{num}$	Numerical flux vector
$\boldsymbol{f}^+, \boldsymbol{f}^-$	Upwind flux vectors
f_P	Fraction of problem that is parallelizable
h_0	Stagnation enthalpy per unit mass
$\boldsymbol{L}$	Lower-triangular matrix in inner problem arising in Euler implicit time integration
l	Inner iteration number; stage number for Jameson-style Runge-Kutta time integration
LHS, *RHS*	Left-hand side, right-hand side of an equation
M	Mach number
m	Number of stages in Jameson-style Runge-Kutta time integration
N	Number of compute nodes used for a given run; quantity associated with a cell's immediate neighbor
$\hat{\boldsymbol{n}}$	Unit normal vector associated with a face
p	Thermodynamic pressure
$\boldsymbol{Q}$	Conservative variable vector, $\boldsymbol{Q} = \lfloor \rho, \rho u, \rho v, \rho w, \rho e_0 \rfloor^T$
$\boldsymbol{Q}_L, \boldsymbol{Q}_R$	States on the left and right sides of a face, respectively

$\boldsymbol{q}$	Primitive variable vector, $\boldsymbol{q} = \lfloor \rho, u, v, w, p \rfloor^T$
R	Gas constant
$\boldsymbol{R}$	Residual vector
$\boldsymbol{r}_{face}$	Position vector of a face centroid wrt a cell centroid
wrt	"with respect to"
S	Surface domain of integration
GS, SGS, SOR, SSOR	Gauss-Seidel, Symmetric Gauss-Seidel, Successive Over-Relaxation, Symmetric Successive Over-Relaxation
t_B	Measured convergence time on the baseline number of nodes
t_N	Measured convergence time on N nodes
tol	User-specified tolerance for inner problem convergence in Euler implicit time integration
$\boldsymbol{U}$	Upper-triangular matrix in inner problem arising in Euler implicit time integration
u	Generic scalar variable
u, v, w	Components of velocity in Cartesian coordinates
$\boldsymbol{V}$	Velocity vector
x, y, z	Cartesian coordinates
α_l	Coefficient in Jameson-style Runge-Kutta, $\alpha_l = 1/(m + 1 - l)$
γ	Ratio of specific heats, $\gamma = c_p / c_v$
Γ_B	Set of ghost cells surrounding a cell $(\Gamma_B \subset \Gamma_I)$
Γ_I	Set of cells on the opposite side of faces in Φ_I
Γ_N	Set of cells that are neighbors of a cell $(\Gamma_N \equiv \Gamma_I + \Gamma_O)$
Γ_O	Set of cells on the opposite side of faces in Φ_O
Δ	Block diagonal matrix in inner problem arising in Euler implicit time integration
Δt	Timestep

η_P	Parallel efficiency
ρ	Mass density
Φ_B	Set of faces surrounding a cell which are on the global boundary ($\Phi_B \subset \Phi_I$)
Φ_I	Set of faces surrounding a cell which possess inward-directed normals wrt the cell
Φ_O	Set of faces surrounding a cell which possess outward-directed normals wrt the cell
σ	Parallel speedup
Ω	Volume domain of integration
ω	Relaxation parameter for the SOR schemes

Operators, diacriticals, etc.

$\|\boldsymbol{a}\|$	Vector or matrix norm of $\boldsymbol{a}$
$\|\boldsymbol{a}\|_p$	p-norm of a vector $\boldsymbol{a}$, e.g., $\|\boldsymbol{a}\|_p \equiv \left(\sum_i \left| a_i \right|^p \right)^{1/p}$
∇a	Gradient of a
$(a)^n$	Value of a at timestep n
$\Delta^n a$	Change in a over a timestep: $\Delta^n a \equiv a^{n+1} - a^n$
a_{min}, a_{max}	Minimum or maximum of a over a cell and all of its neighbors, e.g., $a_{min} \equiv \min\left(a_{cell}, a_{\Gamma_N}\right)$
a_{cell}	Value of a associated with a cell
a_{face}	Value of a associated with a face
a_N	Value of a associated with a cell's immediate neighbor

List of Figures

List of Tables

Introduction

Although sequential computer performance has historically increased exponentially, this trend cannot continue indefinitely. Due to the finite speed of light and other physical limitations, there is an absolute speed limit for sequential computers, and we are rapidly approaching that limit. Bergman and Vos (1991) estimate that computer performance on the order of 10^{12}-10^{18} floating-point operations per second is necessary to model an entire airplane (including the aerodynamics, structure, and propulsion and control systems) and obtain turnaround times short enough to impact a design. This kind of performance can only be achieved using parallel computers (Simon, et al. 1992, 28). Gropp, Lusk, and Skjellum (1994, 4) state, "Barriers to the widespread use of parallelism are in all three of the usual large subdivisions of computing: hardware, algorithms, and software." This effort addresses the second of these broad areas.

The purpose of this research effort is to identify viable parallel algorithms for the Euler equations as implemented in the context of unstructured grids. The approach is to implement several algorithms for solving the system of Euler equations in a single computer program and to time the algorithms using several test cases. For each case, parameters such as grid ordering and distribution, initial and boundary conditions, and convergence criteria are fixed for these timing runs.

The algorithms implemented in the code include both explicit and implicit types. The multi-stage Jameson-style Runge-Kutta explicit algorithm is included for its high degree of data locality as well as its popularity (Jameson, Schmidt & Turkel 1981); both single-stage (or Euler explicit) and four-stage variants are investigated. The Block Jacobi, Block Gauss-Seidel, and Block Symmetric Gauss-Seidel iterative schemes are also implemented to solve the matrix problem that arises with Euler implicit time integration (Stoer & Bulirsch 1980, 560-562; Golub & Van Loan 1989, 513-514).

Several authors have investigated parallel explicit time integration schemes on structured grids (Deshpande et al. 1993; Otto 1993; Underwood et al. 1993; Scherr 1995; Drikakis 1996; Peric & Schreck 1996; Stamatis & Papailiou 1996). Others have used par-

allel explicit time integration schemes on unstructured grids (Venkatakrishnan, Simon & Barth 1992; Brueckner, Pepper & Chu 1993; Simon & Dagum 1993; Weinberg & Long 1994; Brueckner, Pepper & Chu 1993; Morano & Mavriplis 1994; Scherr 1995; Helf, Birken & Küster 1996). There has been somewhat less work involving parallel implicit schemes, most of it in the context of structured grids (Chyczewski et al. 1993; Tysinger & Caughey 1993; Stagg et al. 1993; Ajmani, Liou & Dyson 1994; Candler, Wright & McDonald 1994; Drikakis, Schreck & Durst 1994; Hixon & Sankar 1994; Ajmani & Liou 1995; Povitsky & Wolfstein 1995), but some work has been done with implicit unstructured flow solvers (Venkatakrishnan 1994; Venkatakrishnan 1995; Lanteri & Loriot 1996; Silva & Almeida 1996). Lanteri and Loriot (1996) describe the performance of a production flow solver using Euler implicit time integration with a fixed number of Jacobi iterations for the inner problem. Silva and Almeida (1996) describe the parallel performance of the Generalized Minimum Residual (GMRES) algorithm (Saad & Schulz 1986), while Venkatakrishnan (1994 & 1995) examines the parallelization properties of the GMRES algorithm using two different preconditioners and compares the performance of each of these with a four-stage Jameson-style Runge-Kutta explicit algorithm. Despite these efforts, there has been very little work that compares the performance of several different algorithms on the same problem (Venkatakrishnan's work comparing GMRES and four-stage Runge-Kutta being the exception). This research is meant to address this shortcoming in the literature.

Parallel computers generally fall into two broad classes: shared-memory (Cray, Silicon Graphics) and distributed-memory (Intel iPSC/860, Touchstone Delta, and Paragon; IBM SP-1 and SP-2; TMC CM-5; workstation clusters). In the shared-memory architecture, all processors share the same physical memory, and hence each has equal access to data. Each processing unit (or *compute node*) has its own local memory in the distributed-memory architecture. Access to nonlocal data in the distributed-memory machine involves communication: data must be moved across some kind of connecting device between the compute nodes. Because all of the compute nodes share the same memory bus, no communication network is necessary in a shared-memory machine. However, some cooperation between processors is still necessary to maintain data integrity. All of the

so-called “massively parallel” computers in existence today are distributed-memory machines. This is because most shared-memory implementations are currently limited by the bandwidth of the memory bus to less than about twenty processors. With many more processors than this, the bus quickly becomes saturated and one or more processors become data-starved — that is, the memory accesses become the factor limiting aggregate performance. The distinction between shared- and distributed-memory is becoming blurred as some vendors introduce hybrid machines that have shared-memory compute nodes with several processors each, with the nodes connected using some kind of high-speed interconnection device. For example, SGI’s ChallengeArray consists of several multi-processor boxes, each with its own power supply and disks, with Hi-Performance Parallel Interface (HiPPI) connections between boxes; Intel’s Paragon is currently available with optional compute nodes configured with two i860 compute processors and one i860 message coprocessor. The Intel machine uses the same connection network for the optional nodes as for the standard nodes, which each have one compute processor.

Parallel computing paradigms also fall into two broad classes: functional decomposition and domain decomposition. These classes reflect how a problem is allocated to processes[*]. Functional decomposition divides a problem into several distinct *tasks* that may be executed in parallel; one field where this is popular today is that of Multidisciplinary Design Optimization. On the other hand, domain decomposition distributes *data* across processes, with each process performing more or less the same operations on the data. Domain decomposition is to be distinguished from the *Single-Instruction, Multiple-Data* (SIMD) model in Flynn’s taxonomy of computer architectures (Flynn 1972): in the SIMD model, each process executes the same instructions *at the machine level*, i.e., all processes proceed in lockstep, doing exactly the same operations on different data. This kind of parallelism (sometimes called *fine-grain* parallelism) is usually found at the loop level, whereas domain decomposition is applied at the problem level.

[*]Presumably each *process* runs on a different *processor*, but this is not required.

Domain decomposition may be further subdivided into the data-parallel and explicit message-passing paradigms. Since the separate processes participating in a computation must usually communicate somehow, the data-parallel paradigm is often simply a high-level computing-language construct sitting on top of message passing: this is always the case when a data-parallel program is run on a distributed-memory computer. Hence, the explicit message-passing paradigm is more general than the data-parallel paradigm because it is closer to the machine.

It is difficult to separate the programming paradigm used from the architecture of the hardware. While shared-memory parallel architectures lend themselves to a data-parallel programming paradigm, this paradigm is less well-suited to distributed-memory computers. In this paradigm, the programmer must tell the compiler how to partition data across compute nodes because the compiler sets up all of the message passing calls. Hence, there is less flexibility in specifying how to partition data across compute nodes than with explicit message passing. On the other hand, the programmer is freed from the chore of calling message functions and can concentrate on the application itself; it is usually much easier to parallelize existing sequential code using this paradigm than with explicit message passing.

For efficient use of more than about twenty processors, explicit message passing has been a more successful paradigm than data-parallel, perhaps because it forces the programmer to be painfully aware of the cost of sending a message: both the receiver and the sender must participate (not necessarily at the same time) in the message, i.e., each send must have a corresponding receive and vice versa. Therefore, the programmer is strongly encouraged to ensure that as much of his data as possible resides on the local node and to think carefully about domain decomposition. Since no compiler can be as intimate with an application's data structures as the programmer, it is unlikely that any compiler will ever do as good a job as the programmer in partitioning his data. Finally, explicit message passing is completely flexible in data partitioning because the compiler doesn't care how the data is distributed: the programmer has done all the work concerning what to send and how to send it.

Numerical discretizations of partial differential and integral equations lend themselves readily to parallelization; most computational fluid dynamics (CFD) formulations

fall into one of these two broad classes. This work uses the finite-volume formulation to solve the integral conservation equations of fluid dynamics directly, rather than first transforming the conservation laws to differential form. Unlike a finite-difference formulation, the finite-volume formulation does not restrict one to topologically Cartesian (so-called *structured*) grids, yielding enormous geometric flexibility.

Structured grids have an implicit connectivity: each grid point, cell, and face may be specified uniquely by its computational coordinates. The data structures used with structured grids lend themselves readily to vectorization, but structured grids are very hard to produce for general complex geometries. Also, structured grids often require the user to carry dense grids all the way into the farfield because there is no way to reduce the number of cells in a given logical coordinate plane. For example, if a problem requires a 100×100 grid on the surface of a wing, then the same 100×100 grid dimensions must be carried all the way through the computational domain. Figure 1 shows a structured grid about an object similar to an aircraft forebody, and illustrates the propagation of grid dimensions into the domain.

Multizonal capability was introduced to address the geometric modeling difficulties associated with structured grids. In this technique, several topologically Cartesian grids are joined to discretize a domain. These zones can be connected arbitrarily, given that no voids or overlaps are introduced. For many configurations, however, multiple zones are simply not enough: grid cells are often very highly skewed, and most of the time spent in an analysis, from problem geometry to computed solution, is still in the grid generation phase. Currently, most production CFD work is still performed on multizone structured grids.

In contrast to structured grids, unstructured grids have no implicit connectivity. Since unstructured grids inherently possess more geometric modeling flexibility than structured grids, the unstructured grid may require many fewer cells to adequately model a given geometry than a structured grid, thereby decreasing memory requirements. Unlike structured grids, grid generation is qualitatively the same for complex as well as simple domains. Therefore, more of the grid generation can be automated, speeding the grid generation pro-

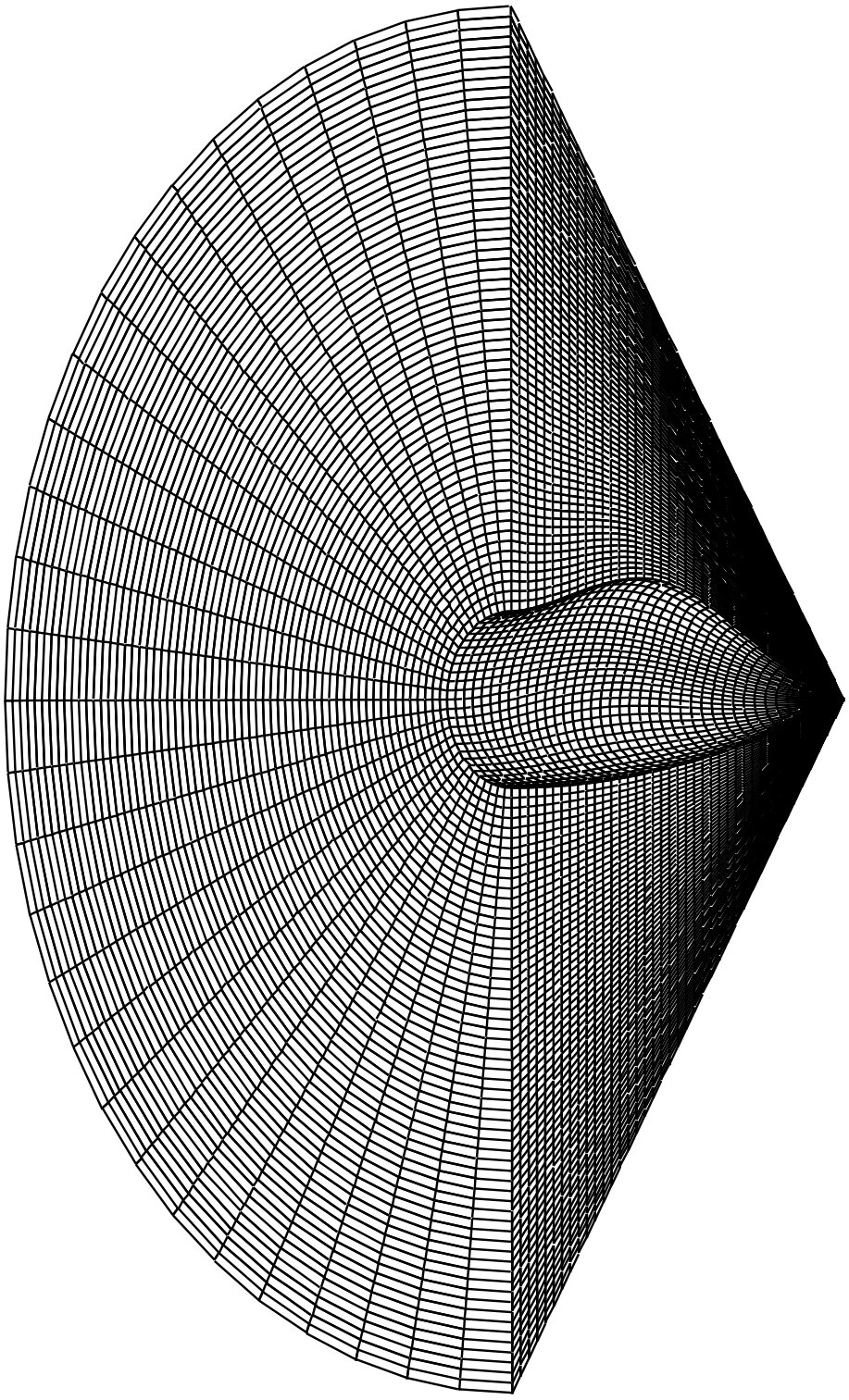


Figure 1: Single-zone structured grid about an object similar to an aircraft forebody.

cess considerably. For applications with complex geometry or requiring rapid turnaround time, the unstructured formulation appears to be the method of choice.

After presentation of the sequential code and the steps taken to parallelize it, several cases representative of the kinds of physical problems that may be solved using the Euler equations will be presented. The results of several hundred timing runs performed using these cases are then discussed; these results are summarized in two appendices. In these results, three measures will be defined and compared: the convergence time, parallel speedup, and parallel efficiency. It will be demonstrated that the algorithms with the highest parallel efficiency are not necessarily “best”.

Formulation

In this chapter, the basic formulation underlying the parallel code is presented. An attempt has been made to defer features that are peculiar to parallelization to a later chapter. Likewise, discussion of specific numerical techniques has also been deferred.

The Finite Volume Formulation

In the finite volume formulation, the integral expressions for conservation of mass, momentum, and energy over an arbitrary control volume are solved directly rather than first being transformed to differential form. Since these expressions are valid for any control volume, considerable flexibility is permitted in the definition of the control volume's shape. Neglecting body forces, these conservation laws over an arbitrary control volume Ω may be written in vector form as:

$$\frac{\partial}{\partial t}\int_{\Omega} \boldsymbol{Q}\, dV + \oint_{S} (\boldsymbol{F}\cdot\hat{\boldsymbol{n}})\, dS \;=\; 0 \tag{1}$$

where

$$\boldsymbol{Q} \equiv \begin{Bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ \rho e_0 \end{Bmatrix} \tag{2}$$

and

$$\boldsymbol{F}\cdot\hat{\boldsymbol{n}} \equiv \begin{Bmatrix} \rho V_n \\ \rho u V_n \\ \rho v V_n \\ \rho w V_n \\ \rho h_0 V_n \end{Bmatrix} \tag{3}$$

If we assume a perfect gas equation of state, the system may be closed by

$$p = (\gamma - 1)\,\rho e \equiv (\gamma - 1)\,\rho\left(e_0 - \frac{1}{2}\boldsymbol{V}\cdot\boldsymbol{V}\right) \tag{4}$$

In the finite-volume formulation, the physical domain is broken up into many small control volumes (*cells*) of simple shape. In this discretization, for a constant control volume, Equation (1) becomes

$$\Omega_i \frac{d}{dt}\overline{\boldsymbol{Q}}_i + \sum_{j=1}^{Nfaces_i} \left[\left(\boldsymbol{F}\cdot\hat{\boldsymbol{n}}S\right)_j\right]_i = 0 \qquad i = 1, Ncells \tag{5}$$

where $\overline{\boldsymbol{Q}}_i$ is the volume average of $\boldsymbol{Q}$ over the cell i and $\left(\left[\boldsymbol{F}\cdot\hat{\boldsymbol{n}}S\right]_j\right)_i$ is the average flux through face j of cell i.

Since Equation (1) holds for any control volume, we can apply Equation (1) to each cell individually, i.e., the flux balance in Equation (1) must be satisfied for all cells. Henceforth, the average will be assumed and the overbar will be dropped.

Upwind Differencing

The Euler equations of fluid dynamics are mixed hyperpolic-elliptic for $M < 1$ and hyperbolic for $M > 1$. It therefore makes mathematical as well as physical sense to use difference expressions that reflect this hyperbolic nature. Upwind differencing represents an attempt to include our knowledge of the character of the equations in our difference expressions. The basic idea behind upwinding was first proposed in a landmark paper by Courant, Isaacson, and Reeves (1952). Their paper dealt with the one-dimensional linear advection equation and proposed taking one-sided differences based on the sign of the wave speed.

In an upwind formulation for the Euler equations, there are two states on either side of a face which must be resolved into a single value for the flux through the face. Steger and Warming (1981) first proposed a technique called *Flux Vector Splitting* to split the flux vectors based on the signs of the eigenvalues of the characteristic form of the Euler equations. Their expressions for the flux have slope discontinuities for Mach numbers of ± 1. Van Leer (1982) extended the Flux Vector Splitting idea to make the flux functions continuously differentiable, giving better solutions at sonic transitions as well as better convergence behavior.

Techniques which solve a Riemann (shock-tube) problem for the two states on either side of the face represent an attempt to include more physics into the flux formulation. This basic idea is due to Godunov (1959). It is usually very expensive to solve this Riemann problem exactly; therefore, approximate Riemann solvers, which either solve the exact problem approximately or solve an approximate problem exactly, are popular. The most popular technique of this type is Roe's *Flux Difference Splitting* (Roe 1981), which computes an exact solution to an approximate Riemann problem.

The flux for any finite volume scheme may be expressed in terms of the states on either side of the face:

$$(\boldsymbol{F} \cdot \hat{\boldsymbol{n}})_{num} = \boldsymbol{f}(\boldsymbol{Q}_L, \boldsymbol{Q}_R, \hat{\boldsymbol{n}}) \tag{6}$$

Central difference schemes would substitute the average of $\boldsymbol{Q}_L$ and $\boldsymbol{Q}_R$ into Equation (3) to compute a value for the flux; upwind schemes combine fluxes computed individually from the two states. For example, Flux Vector Splitting uses

$$(\boldsymbol{F} \cdot \hat{\boldsymbol{n}})_{num} = \boldsymbol{f}^{+}(\boldsymbol{Q}_L, \hat{\boldsymbol{n}}) + \boldsymbol{f}^{-}(\boldsymbol{Q}_R, \hat{\boldsymbol{n}}) \tag{7}$$

Whether upwind or central, Jacobians of the numerical flux in Equation (6) are computed for each independent variable. The notation for the flux Jacobians is borrowed from Equation (7):

$$\boldsymbol{A}^{+} \equiv \frac{\partial}{\partial \boldsymbol{Q}_L}(\boldsymbol{F} \cdot \hat{\boldsymbol{n}})_{num} \tag{8}$$
$$\boldsymbol{A}^{-} \equiv \frac{\partial}{\partial \boldsymbol{Q}_R}(\boldsymbol{F} \cdot \hat{\boldsymbol{n}})_{num}$$

Upwind schemes such as Van Leer's Flux Vector Splitting and Roe's approximate Riemann solver take advantage of the hyperbolicity in the Euler equations (Van Leer 1982; Roe 1981). These methods are based on solutions of the one-dimensional Euler equations; the extension to higher dimensions treats each flux component as being locally one-dimensional (Hirsch 1990, 475-483). Both Van Leer's Flux Vector Splitting and Roe's Flux Difference Splitting schemes are implemented in the code; Roe's scheme is used for all of the timing runs.

Monotonicity, Extension to Higher Order, and Limiters

In the classic Godunov formulation, the solution variables are viewed as piecewise constant cell averages; a Riemann problem is solved at every face to obtain the flux through the face. Using simply the cell averages on either side of a face leads to a scheme with a first-order error term, which is usually unacceptably diffusive for practical calculations.

To obtain higher-order accuracy, Van Leer showed that it is sufficient to use higher-order extrapolations to the states on either side of a face. The spatial error in the solution will

then be of the same order as the error in the variable extrapolation; Van Leer (1979) called this *MUSCL Differencing*. In PUE3D (for **P**arallel **U**nstructured **E**uler, **3D**), the computer code written for this work, second-order gradient-based variable extrapolation is used to extend the scheme to higher order (Barth & Jesperson 1989):

$$\boldsymbol{Q}_{face} = \boldsymbol{Q}_{cell} + \boldsymbol{r}_{face} \cdot \nabla \boldsymbol{Q}_{cell} \tag{9}$$

where $\nabla \boldsymbol{Q}_{cell}$ is the average gradient of $\boldsymbol{Q}$ in the cell. Here, the solution variables are piece-wise linear, and the cell average is maintained.

Godunov (1959) introduced the concept of monotonicity in the context of numerical solutions to scalar conservation laws. For example, consider the linear advection equation:

$$\frac{\partial u}{\partial t} + \frac{\partial f}{\partial x} = 0$$

Write the solution at time step $n + 1$ in terms of the solution at time step n:

$$u_i^{n+1} = H\left(u_{i-k}^{n}, u_{i-k+1}^{n}, \ldots, u_{i+k}^{n} \right)$$

If

$$\frac{\partial}{\partial u_j} H\left(u_{i-k}, u_{i-k+1}, \ldots, u_{i+k}\right) \geq 0 \quad \forall\ i-k \leq j \leq i+k,$$

then the scheme is said to be *monotone* (Hirsch 1990, 525-527). Godunov also showed that linear schemes possessing this property are at most first-order accurate. This means that any scheme which computes fluxes based on states from Equation (9) cannot be made mono-tone without the introduction of some nonlinearity.

Harten (1983) introduced the following generalization of Godunov's monotonicity concept in one dimension: if the solution changes from timestep n to $n + 1$ such that

$$\left(\int \left| \frac{\partial u}{\partial x} \right| dx \right)^{n+1} \leq \left(\int \left| \frac{\partial u}{\partial x} \right| dx \right)^{n} \tag{10}$$

or, in discrete form,

$$\left(\sum_{i=1}^{N-1} \left| u_{i+1} - u_i \right| \right)^{n+1} \leq \left(\sum_{i=1}^{N-1} \left| u_{i+1} - u_i \right| \right)^{n} \tag{11}$$

then the scheme is said to be *Total Variation Diminishing*, or TVD. For a linear scheme, the TVD property is the same as monotonicity. For a nonlinear scheme, however, one can maintain the TVD property while achieving higher order (at least in one dimension) by using nonlinear functions called *limiters* to bound the solution variables such that Equation (11) holds. However, Goodman and LeVeque (1985) have shown that TVD schemes in multiple dimensions are at most first-order accurate.

Spekreijse (1987) introduced a different, less stringent definition of monotonicity. According to this definition, a scheme is monotone if the variable extrapolation to the face (in one dimension) is bounded by

$$\min\left(u_i, u_{i+1}\right) \leq u_{i+1/2} \leq \max\left(u_i, u_{i+1}\right), \tag{12}$$

where $u_{i+1/2}$ is the value of u at the face joining cells i and $i + 1$.

In structured CFD, the extension of Equation (12) to three dimensions is rather obvious: the gradients are limited separately in each logical direction so that

$$\begin{aligned}
\min\left(u_{i,j,k}, u_{i+1,j,k}\right) &\le u_{i+1/2,j,k} \le \max\left(u_{i,j,k}, u_{i+1,j,k}\right) \\
\min\left(u_{i,j,k}, u_{i,j+1,k}\right) &\le u_{i,j+1/2,k} \le \max\left(u_{i,j,k}, u_{i,j+1,k}\right) \\
\min\left(u_{i,j,k}, u_{i,j,k+1}\right) &\le u_{i,j,k+1/2} \le \max\left(u_{i,j,k}, u_{i,j,k+1}\right)
\end{aligned}$$

Unfortunately, no such convenient extension exists in the general case of an unstructured grid. Barth and Jesperson (1989) extended Equation (12) to three dimensions by using the maximum and minimum values over the cell and its neighbors to bound the extrapolated face variables:

$$u_{min} \le u_{face} \le u_{max}, \tag{13}$$

where

$$\begin{aligned}
u_{min} &= \min\left(u_{cell}, u_N\right) \\
u_{max} &= \max\left(u_{cell}, u_N\right)
\end{aligned} \qquad N \in \Gamma_N$$

Note that while extrapolated variables meeting this criterion satisfy Spekreijse's definition of monotonicity, both Godunov's definition of monotonicity and the TVD property may be violated.

All this is very nice and general, but most of the limiters in use today have been defined for Equation (12), not Equation (13), and in fact, most may be expressed in terms of the ratio of consecutive differences (Sweby 1984)

$$r = \frac{u_{i+1} - u_i}{u_i - u_{i-1}}, \tag{14}$$

and it is not immediately obvious how one would express *r* on an unstructured grid.

If we consider that Equation (14) is nothing more than the ratio of the central difference of u to the one-sided (or upwind) difference of u evaluated at $i + 1/2$, then a generalization to unstructured grids immediately suggests itself. For any face, define

$$r = \frac{(u_N - u_{cell}) / \mathbf{r}_N}{(u_{face} - u_{cell}) / \mathbf{r}_{face}}$$

or, since $\|\mathbf{r}_N\|_2 = 2\|\mathbf{r}_{face}\|_2$ on a uniform one-dimensional grid,

$$r = \frac{u_N - u_{cell}}{2\,(u_{face} - u_{cell})}$$

A formulation very similar to this was first proposed by Venkatakrishnan and Barth (1989). This formulation does not consider the maximum and minimum values over the cell and its neighbors. However, if we instead define the numerator using

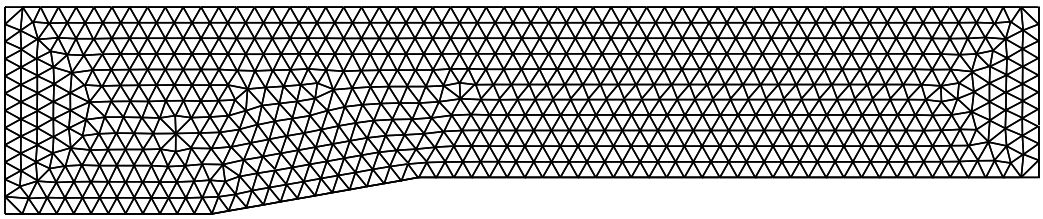
$$C = \begin{cases} u_{max} - u_{cell} & \text{if } u_{face} > u_{cell} \\ u_{min} - u_{cell} & \text{if } u_{face} < u_{cell} \\ 0 & \text{if } u_{face} = u_{cell} \end{cases}$$

then

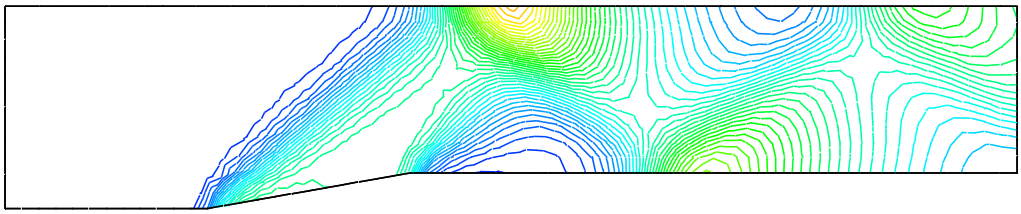
$$r = \frac{C}{2\,(u_{face} - u_{cell})},$$

and Equation (13) is satisfied. Note that $r \geq 0$.

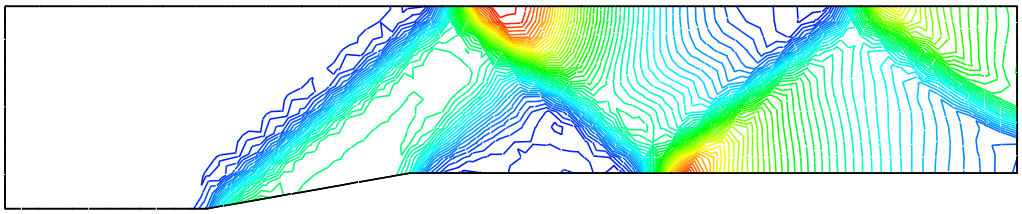
Several computer runs were performed using various limiters on a simple 10˚ ramp in a duct with a Mach 2 freestream; the grid and pressure contours are shown in Figure 2. The technique described above and in Bruner & Walters (1996) has been successfully applied to the modified Van Albada limiter as presented in Venkatakrishnan (1993) as well as



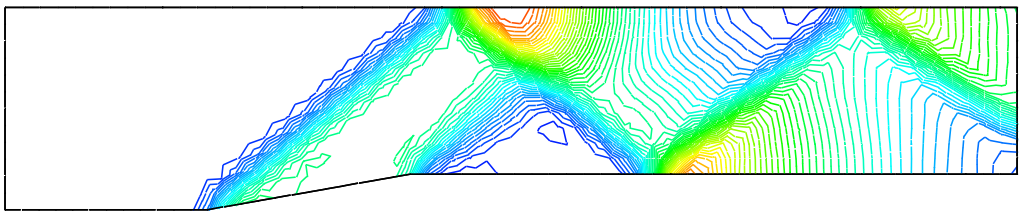
a) Computational grid.



b) Pressure contours: 1st order.

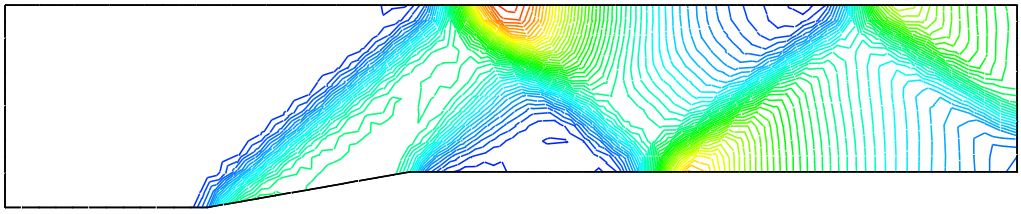


c) Pressure contours: unlimited.

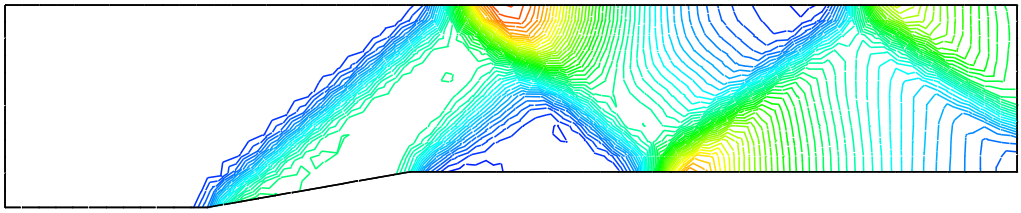


d) Pressure contours: Barth & Jesperson limiting.

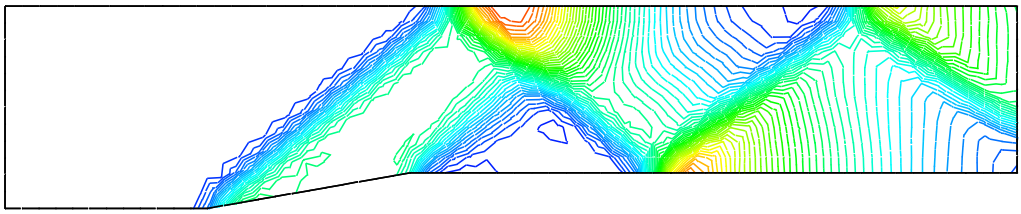
Figure 2: Grid and pressure contours on the 10˚ ramp used in the limiter comparison.



e) Pressure contours: Venkatakrishnan limiter, K=550.



f) Pressure contours: present technique applied to Van Albada's limiter, K=300.



g) Pressure contours: present technique applied to Roe's Superbee limiter.

Figure 2 (continued): Grid and pressure contours on the 10˚ ramp used in the limiter comparison.

to Roe's Superbee limiter (Van Albada, Van Leer & Roberts 1982; Roe 1985). As evidenced in Figures 2, 3 and 4, the Van Albada limiter compares favorably with the Venkatakrishnan limiter in terms of smoothness of the solution (Venkatakrishnan 1993), but the convergence using the Van Albada limiter with the selected values of K is not as fast. The parameter K in both the Venkatakrishnan and Van Albada limiters is used to avoid clipping smooth extrema in the flow: $K = 0$ corresponds to full limiting, while $K \to \infty$ gives the unlimited solution. The solution computed using the Van Albada limiter is smoother than the one using the Venkatakrishnan limiter because a smaller value for K was required for convergence, so that the Venkatakrishnan limiter permits more oscillations in the solution than the Van Albada limiter for these values of K.

Note that the results for the Barth and Jesperson limiter and Roe's Superbee limiter in Figures 2 and 3 are identical. Even the convergence history for Superbee is identical to that using the limiter of Barth and Jesperson (see Figure 4). This is because the Superbee limiter reduces to the Barth and Jesperson limiter for the case where $0 \le r \le 1$. When computing gradients based on face values computed using some weighted average of adjacent cell values, this is guaranteed; this is exactly what PUE3D does for second-order computations. If higher-order k-exact reconstruction were used instead (Barth & Frederickson 1990; Barth 1993; Mitchell 1994), then the limiters would behave differently from one another.

The next chapter addresses the numerical techniques used to integrate Equation (5) for the cell averages $\boldsymbol{Q}$.

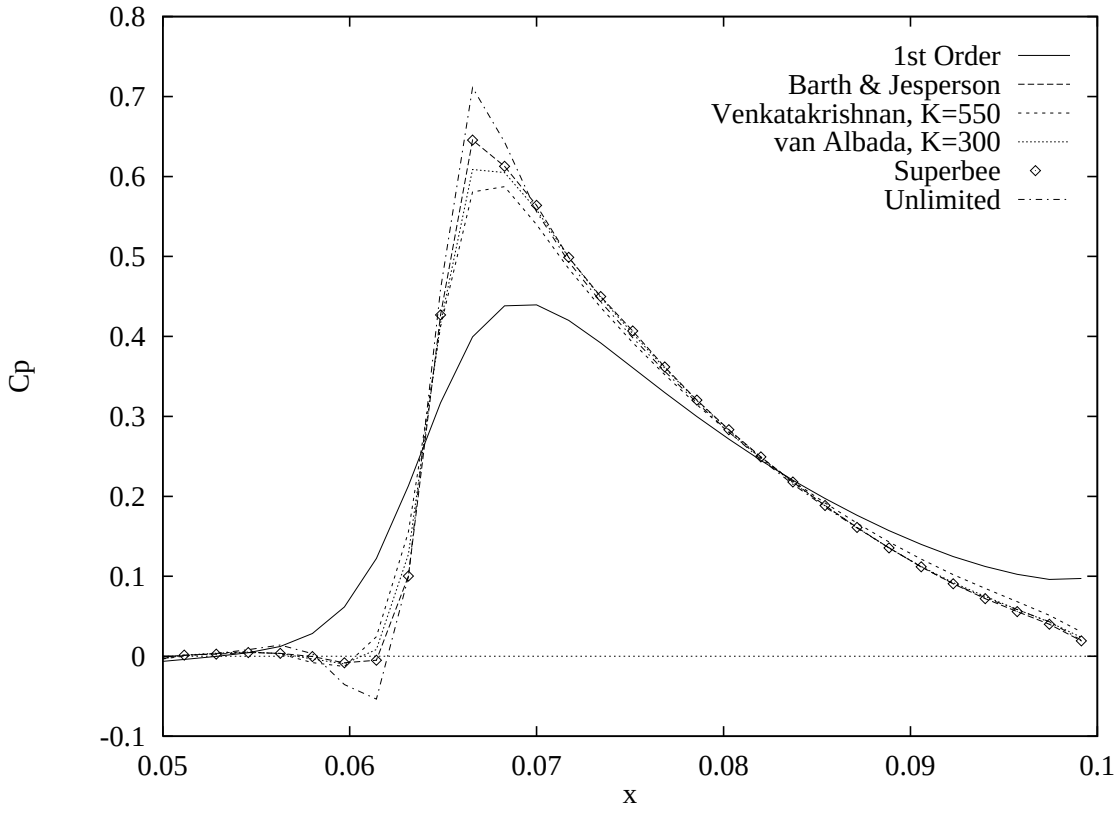


Figure 3: Lower surface pressure coefficient in the vicinity of the reflected shock using various limiters for Mach 2 flow over a 10˚ ramp. Note that the solution computed using Superbee is identical to that using Barth & Jesperson and is plotted as points on top of the curve for Barth.

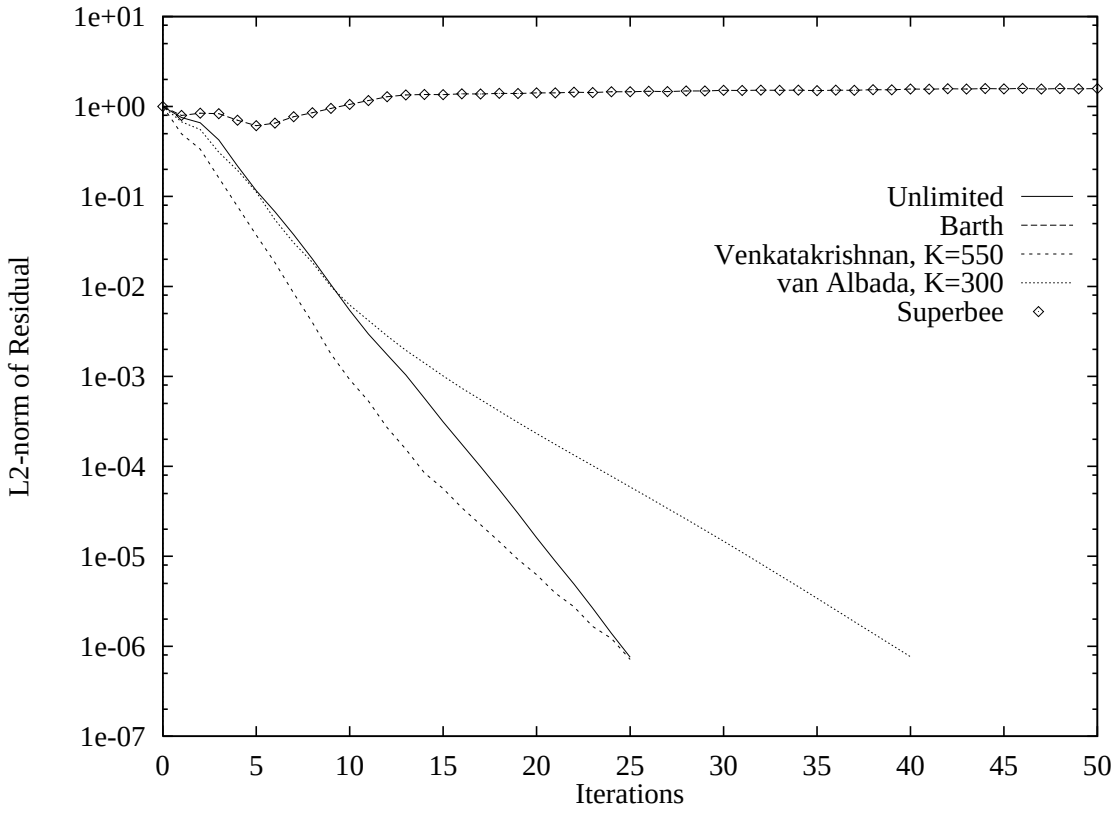


Figure 4: Convergence history using various limiters on the 10˚ ramp. Note that the convergence history for the Superbee computation is the same as that for Barth & Jesperson; Superbee is plotted as points on top of the curve for Barth & Jesperson.

Time Integration Algorithms

In this chapter, various techniques for solving Equation (5) are presented. Since Equation (5) is written as an ordinary differential equation in time, all of the techniques applicable to ODE's may be applied to the temporal evolution of the solution. As in the previous chapter, an attempt was made to divorce parallelization issues from numerical issues. However, this has attempt has not been completely successful because there is communication within each of the time integration algorithms presented here. Hence the discussion must include some parallelization issues, at least in the flowcharts for the algorithms, to show where in the algorithm the communication occurs.

Implicit vs. Explicit Algorithms

Equation (5) may be rewritten in the following way, again assuming a constant control volume:

$$\Omega \frac{d\boldsymbol{Q}}{dt} + \boldsymbol{R}(\boldsymbol{Q}) = 0 \tag{15}$$

where $\boldsymbol{R}$ contains all terms not included in the time derivative.

Discretizing the time derivative to first order,

$$\Omega \frac{\Delta^n \boldsymbol{Q}}{\Delta t} = -\boldsymbol{R} \tag{16}$$

The difference between implicit and explicit time integration schemes is the time step at which we evaluate $\boldsymbol{R}$.

Classic Euler explicit time integration may be written as

$$\Delta^n \boldsymbol{Q} = -\frac{1}{\Omega} \Delta t \boldsymbol{R}^n \tag{17}$$

while Euler implicit time integration simply evaluates $\boldsymbol{R}$ at time step $n + 1$:

$$\Delta^n \boldsymbol{Q} = -\frac{1}{\Omega}\Delta t \boldsymbol{R}^{n+1} \tag{18}$$

Linearizing $\boldsymbol{R}^{n+1}$,

$$\boldsymbol{R}^{n+1} = \boldsymbol{R}^n + \Delta^n \boldsymbol{R} \approx \boldsymbol{R}^n + \frac{\partial \boldsymbol{R}}{\partial \boldsymbol{Q}} \Delta^n \boldsymbol{Q} \tag{19}$$

So Equation (18) becomes

$$\left[\frac{\Omega}{\Delta t}\boldsymbol{I} + \left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{Q}}\right)^n\right] \Delta^n \boldsymbol{Q} = -\boldsymbol{R}^n \tag{20}$$

With the exception of $\Delta^n \boldsymbol{Q}$, all of the quantities in Equation (20) are known at timestep $n + 1$.

The left-hand side of Equation (20) is generally a large, sparse, *non-symmetric* matrix, although the matrix is block-symmetric. This means that many of the elegant algorithms developed for positive-definite matrices *will not work* on Equation (20). Also, since Equation (20) will be solved many hundreds or perhaps thousands of times, speed is paramount. On the other hand, if a steady-state solution is desired, it may be not be necessary to solve Equation (20) very accurately, since only the converged solution is of any interest.

Note that the coefficient matrix in Equation (20) may always be made block diagonally dominant* by choosing a small enough time step. This suggests a general iterative solution strategy for the inner problem. Let us rewrite the matrix problem in Equation (20) in the more familiar form:

$$\boldsymbol{A}\boldsymbol{x} = \boldsymbol{b} \tag{21}$$

where $\boldsymbol{A} = \left[\frac{\Omega}{\Delta t}\boldsymbol{I} + \left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{Q}}\right)^n\right]$, $\boldsymbol{x} = \Delta^n \boldsymbol{Q}$, and $\boldsymbol{b} = -\boldsymbol{R}^n$. All of the implicit algorithms in PUE3D solve Equation (21) using some form of operator splitting (Golub & Van Loan 1989, 508-509). This technique solves Equation (21) according to the following iteration sequence:

$$\boldsymbol{M}\boldsymbol{x}^{l+1} = \boldsymbol{N}\boldsymbol{x}^{l} + \boldsymbol{b} \tag{22}$$

where $\boldsymbol{A} = \boldsymbol{M} - \boldsymbol{N}$ and $\boldsymbol{M}$ is easier to invert than $\boldsymbol{A}$.

For later discussion, it will be convenient to split $\boldsymbol{A}$ according to

$$\boldsymbol{A} = -\boldsymbol{L} + \Delta - \boldsymbol{U}, \tag{23}$$

where Δ is a block-diagonal matrix of 5×5 blocks and $\boldsymbol{L}$ and $\boldsymbol{U}$ are lower- and upper-triangular matrices, respectively. Each block of Δ is associated with its own cell.

*If $\left\|\boldsymbol{D}^{-1}\right\|_1 \left(\sum \|\boldsymbol{L}\|_1 + \sum \|\boldsymbol{R}\|_1\right) < 1$, where $\boldsymbol{D}$ is the diagonal block associated with a given cell and $\boldsymbol{L}$ and $\boldsymbol{R}$ are the off-diagonal blocks associated with the cell, then the system is block diagonally dominant (Golub & Van Loan 1989, 171). The 1-norm used here is simply the maximum column sum of the matrix, i.e., $\|\boldsymbol{A}\|_1 \equiv \max_j \sum_{i=1}^{m} |a_{ij}|$, where $\boldsymbol{A}$ is any $m \times n$ matrix.

In all of the implicit algorithms in PUE3D, several inner iterations are performed to solve the matrix problem Equation (21); iteration proceeds until either

$$\frac{\|\boldsymbol{A}\boldsymbol{x}-\boldsymbol{b}\|_2}{\|\boldsymbol{b}\|_2} \leq tol \tag{24}$$

or until a specified number of inner iterations has been performed.

Note also that for $\Delta t \to \infty$, and using an *LHS* consistent with the *RHS*, including boundary conditions, Equation (20) reduces to Newton's method (Burden & Faires 1989, 555):

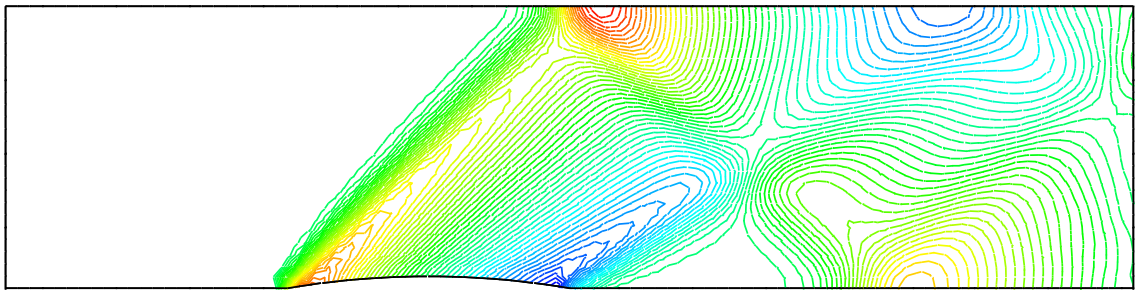
$$\boldsymbol{Q}^{n+1} = \boldsymbol{Q}^{n} - \left(\frac{\partial \boldsymbol{R}}{\partial \boldsymbol{Q}}\right)^{-1} \boldsymbol{R}^{n} \tag{25}$$

Therefore, since Newton's method converges quadratically, we should expect quadratic convergence of the outer problem near the solution as $\Delta t \to \infty$, provided the *LHS* and *RHS* are consistent.

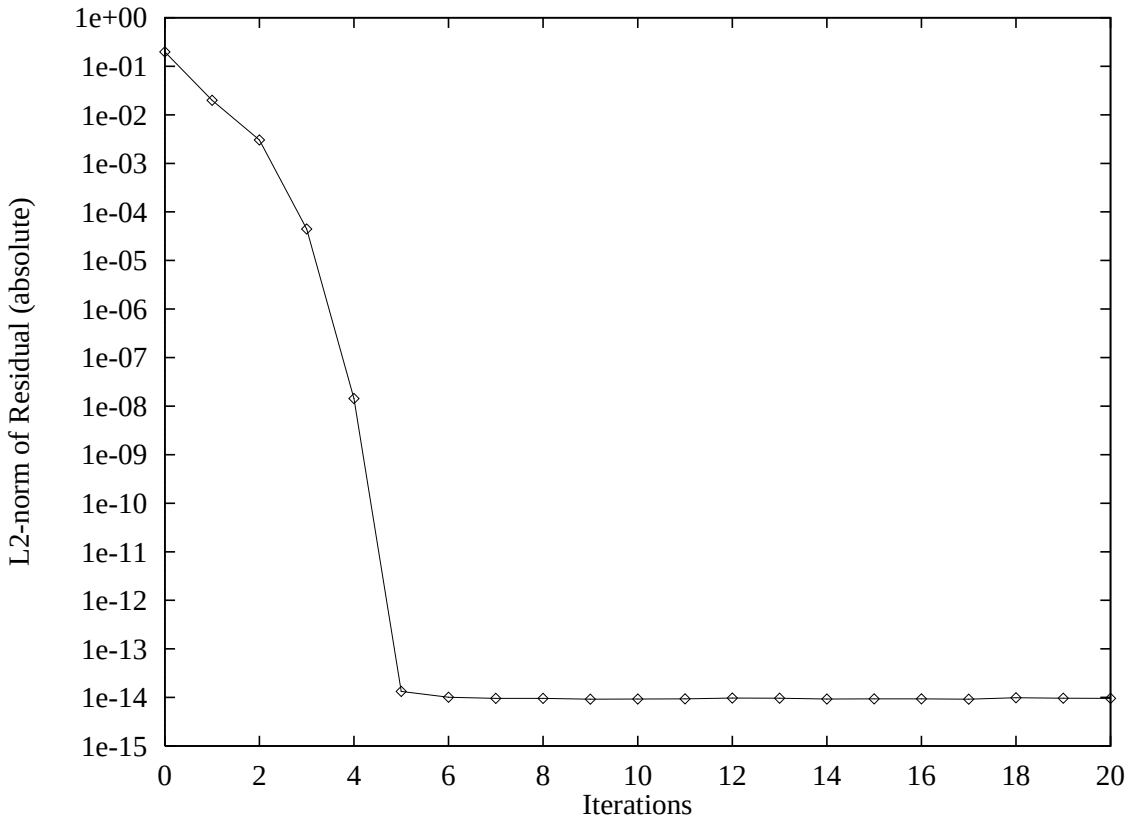
Figure 5 shows the convergence history for supersonic flow over the supersonic bump case described later in the "Test Cases" chapter. This problem was run using Roe's flux difference splitting (Roe 1981) with first-order spatial accuracy and implicit boundary conditions. As is apparent in the figure, quadratic convergence is achieved for very large CFL numbers.

Implementation of Implicit Boundary Conditions

Implicit boundary conditions are used in this research when any of the implicit time integration algorithms are used. *Implicit* in this context means that the boundary condition is evaluated at the next timestep as opposed to the current timestep.



a) pressure contours



b) convergence history

Figure 5: Quadratic convergence with numerical Jacobians for the first-order supersonic bump. The CFL number for this run was 10^{21}.

To begin this exposition, let us expand Equation (20):

$$\left[\frac{\Omega}{\Delta t}\boldsymbol{I} + \left(\frac{\partial \boldsymbol{R}_{cell}}{\partial \boldsymbol{Q}_{cell}}\right)^{n}\right]\Delta^{n}\boldsymbol{Q}_{cell} + \sum_{N \in \Gamma_N}\left(\frac{\partial \boldsymbol{R}_{cell}}{\partial \boldsymbol{Q}_{N}}\Delta^{n}\boldsymbol{Q}_{N}\right) = -\boldsymbol{R}^{n}_{cell} \tag{26}$$

The first term on the *LHS* of Equation (26) is a 5×5 block on the main diagonal in Equation (20); the second term includes all of the off-diagonal blocks. Both terms on the *LHS* of Equation (26) may be expanded using Equation (6) and Equation (8):

$$\frac{\partial \boldsymbol{R}_{cell}}{\partial \boldsymbol{Q}_{cell}} = \sum_{\Phi_O}\boldsymbol{A}^{+}S - \sum_{\Phi_I}\boldsymbol{A}^{-}S \tag{27}$$

and

$$\frac{\partial \boldsymbol{R}_{cell}}{\partial \boldsymbol{Q}_{N}}\Delta^{n}\boldsymbol{Q}_{N} = -\sum_{\Phi_I}\boldsymbol{A}^{+}S\Delta^{n}\boldsymbol{Q}_{\Gamma_I} + \sum_{\Phi_O}\boldsymbol{A}^{-}S\Delta^{n}\boldsymbol{Q}_{\Gamma_O} \tag{28}$$

Combining Equations (26) through (28), then we have for each cell,

$$\begin{aligned}\left(\frac{\Omega}{\Delta t}\boldsymbol{I} + \sum_{\Phi_O}\boldsymbol{A}^{+}S - \sum_{\Phi_I}\boldsymbol{A}^{-}S\right)\Delta^{n}\boldsymbol{Q}_{cell} \\ - \sum_{\Phi_I}\boldsymbol{A}^{+}S\Delta^{n}\boldsymbol{Q}_{\Gamma_I} + \sum_{\Phi_O}\boldsymbol{A}^{-}S\Delta^{n}\boldsymbol{Q}_{\Gamma_O} = -\boldsymbol{R}^{n}_{cell}\end{aligned} \tag{29}$$

As in many CFD codes, PUE3D stores boundary condition data in cells just outside the physical domain; these are often called *ghost* or *phantom* cells. The state stored in these ghost cells is then used along with the state in the first interior cell (for split-flux boundary conditions) to compute the flux through the boundary face. PUE3D also requires that all boundary faces have normal vectors that point into the computational domain. Because the

boundary faces all point into the computational domain, all of the boundary faces for a cell are in $\{\Phi_I\}$. Furthermore, because the boundary conditions depend only on the value of $\boldsymbol{Q}$ in the first interior cell (because the *LHS* in the implicit problem is only first-order), we can write

$$\Delta^n \boldsymbol{Q}_{\Gamma_B} = \underbrace{\frac{\partial \boldsymbol{Q}_{\Gamma_B}}{\partial \boldsymbol{Q}}}_{\boldsymbol{J}_{BC}} \Delta^n \boldsymbol{Q}_{cell}$$

so that Equation (29) becomes

$$\boldsymbol{D}\Delta^n \boldsymbol{Q}_{cell} - \sum_{\Phi_I - \Phi_B} \boldsymbol{A}^+ \Delta^n \boldsymbol{Q}_{\Gamma_I - \Gamma_B} + \sum_{\Phi_O} \boldsymbol{A}^- \Delta^n \boldsymbol{Q}_{\Gamma_O} = -\boldsymbol{R}^n_{cell} \qquad (30)$$

where $\boldsymbol{D} \equiv \left(\frac{\Omega}{\Delta t}\boldsymbol{I} + \sum_{\Phi_O} \boldsymbol{A}^+ - \sum_{\Phi_I} \boldsymbol{A}^- - \sum_{\Phi_B} \boldsymbol{A}^+ \boldsymbol{J}_{BC} \right)$ is the cell's diagonal block in the Jacobian matrix; this $\boldsymbol{D}$ corresponds to one block of Δ in Equation (23).

There are presently 12 different types of boundary condition implemented in PUE3D; more boundary condition types are expected to be added to the code. Rather than deriving a different $\boldsymbol{J}_{BC}$ for every boundary condition type, this Jacobian is obtained numerically using central differences; this is completely general and does much to improve the maintainability of the code. Each solution component in the first interior cell (the one adjacent to the boundary) is perturbed in turn by an amount equal to ± the square root of machine zero. Then new boundary conditions are computed for this perturbed state, and the difference between the boundary condition values for each of the perturbed states gives one part of the Jacobian. Because central differences are second-order accurate (Carnahan, Luther & Wilkes 1969, 430-431), the accuracy of the approximation is of the same order as the

roundoff error of the machine. The flux Jacobians in Roe's scheme are also computed numerically by perturbing each component of each state to obtain $\boldsymbol{A}^+$ and $\boldsymbol{A}^-$.

m-Stage Jameson-style Runge-Kutta

This is the only explicit algorithm implemented in the code. It is based loosely on the classic Runge-Kutta algorithm for numerical integration of ordinary differential equations (Burden & Faires 1989, 254-260), but has the advantage of requiring only two solutions to be stored, making this algorithm very efficient in terms of memory usage. Unfortunately, unlike true Runge-Kutta, this algorithm is at best only second-order accurate in time. Since all of the problems considered here are steady, this is not a serious problem.

A concise expression of the algorithm may be found in Venkatakrishnan (1995, 7) and is repeated below for four stages:

$$
\begin{aligned}
\boldsymbol{Q}^0 &= \boldsymbol{Q}^n \\
\boldsymbol{Q}^1 &= \boldsymbol{Q}^0 - \frac{\alpha_1 \Delta t \boldsymbol{R}\left(\boldsymbol{Q}^0\right)}{\Omega} \\
\boldsymbol{Q}^2 &= \boldsymbol{Q}^0 - \frac{\alpha_2 \Delta t \boldsymbol{R}\left(\boldsymbol{Q}^1\right)}{\Omega} \\
\boldsymbol{Q}^3 &= \boldsymbol{Q}^0 - \frac{\alpha_3 \Delta t \boldsymbol{R}\left(\boldsymbol{Q}^2\right)}{\Omega} \\
\boldsymbol{Q}^4 &= \boldsymbol{Q}^0 - \frac{\alpha_4 \Delta t \boldsymbol{R}\left(\boldsymbol{Q}^3\right)}{\Omega} \\
\boldsymbol{Q}^{n+1} &= \boldsymbol{Q}^4
\end{aligned}
$$

where $\alpha_l = 1 / (5 - l)$.

A flowchart for this algorithm as implemented in parallel is given in Figure 6 (Jameson et al. 1981). Note that the classic Euler explicit algorithm is recovered for $m = 1$ (Burden & Faires 1989, 239-240).

Block Jacobi

In the Block Jacobi algorithm (Stoer & Bulirsch 1980, 560-561), the off-diagonal blocks in Equation (30) are subtracted from the *RHS* at the beginning of every inner iteration. The splitting for this algorithm is simply

$$\boldsymbol{M} = \Delta$$
$$\boldsymbol{N} = \boldsymbol{L} + \boldsymbol{U}$$

so that the iteration becomes

$$\boldsymbol{x}^{l+1} = \Delta^{-1}\left[\boldsymbol{b} + (\boldsymbol{L} + \boldsymbol{U})\,\boldsymbol{x}^{l}\right]$$

Note that Δ may be inverted outside of the inner iteration loop.

A flow chart for the Block Jacobi algorithm showing parallel calls is presented in Figure 7.

Different Forms of Gauss-Seidel Iteration

In Block Gauss-Seidel iteration (Stoer & Bulirsch 1980, 561-562), the latest values for $\boldsymbol{x}$ are used as soon as they become available; the convergence rate of the inner problem should therefore be at least as fast as Block Jacobi iteration (Stoer & Bulirsch 1980, 545).

A minor modification of the basic Gauss-Seidel algorithm is to overshoot the next estimate for the solution vector $\boldsymbol{x}^{l+1}$; this is called Successive Over-Relaxation (Golub & Van Loan 1989, 510-511), or *SOR* for short. The overrelaxation parameter ω is used to specify how much overshoot is permitted. For stability, $0 \le \omega < 2$, with values of ω less than unity termed under-relaxation. The Gauss-Seidel algorithm is recovered for $\omega = 1$.

In numerical experiments performed as part of this investigation, it was found that there is little to be gained by using over-relaxation to solve the inner problem. What is re-

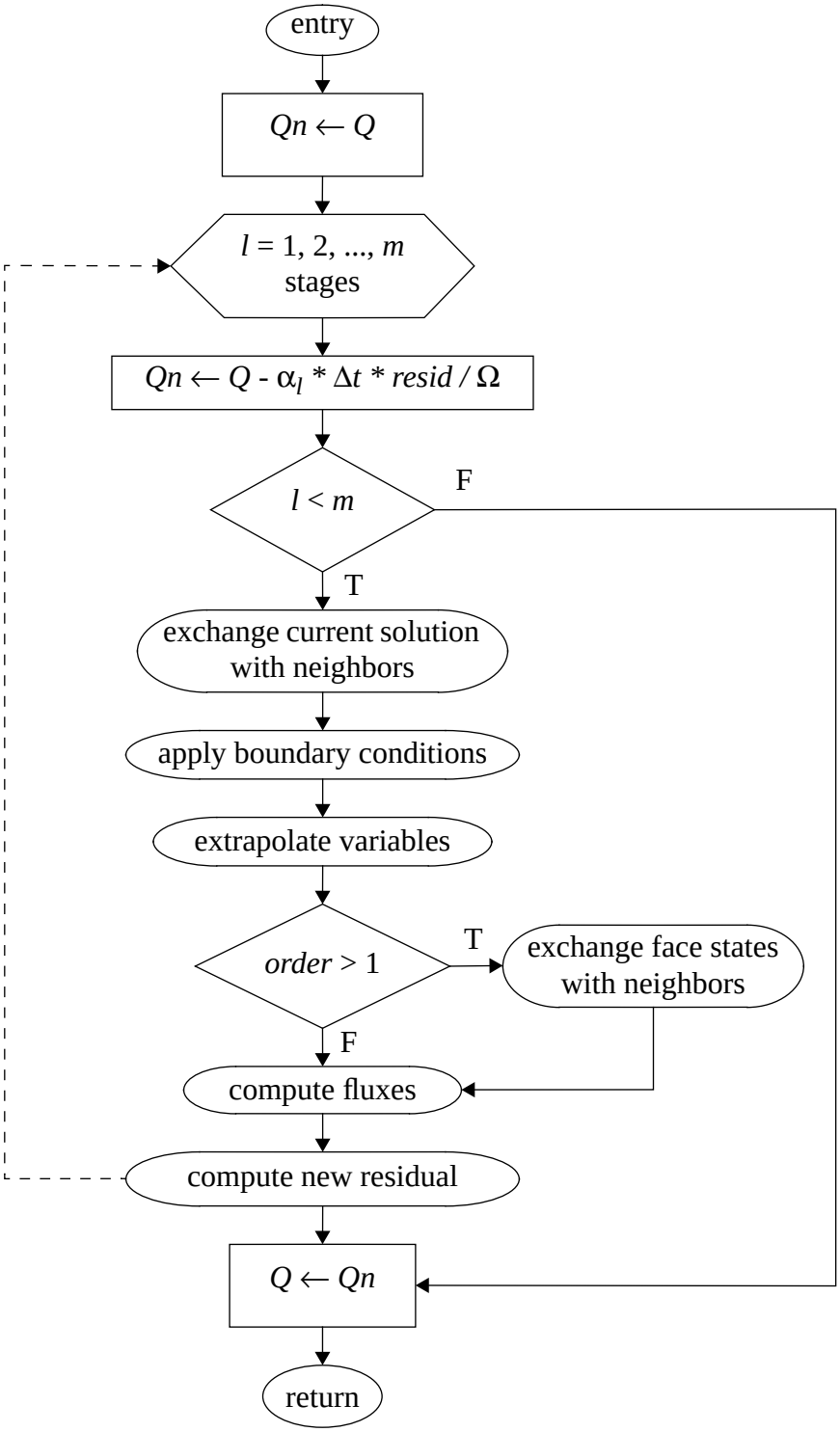


Figure 6: Flowchart for parallel Jameson-style Runge-Kutta.

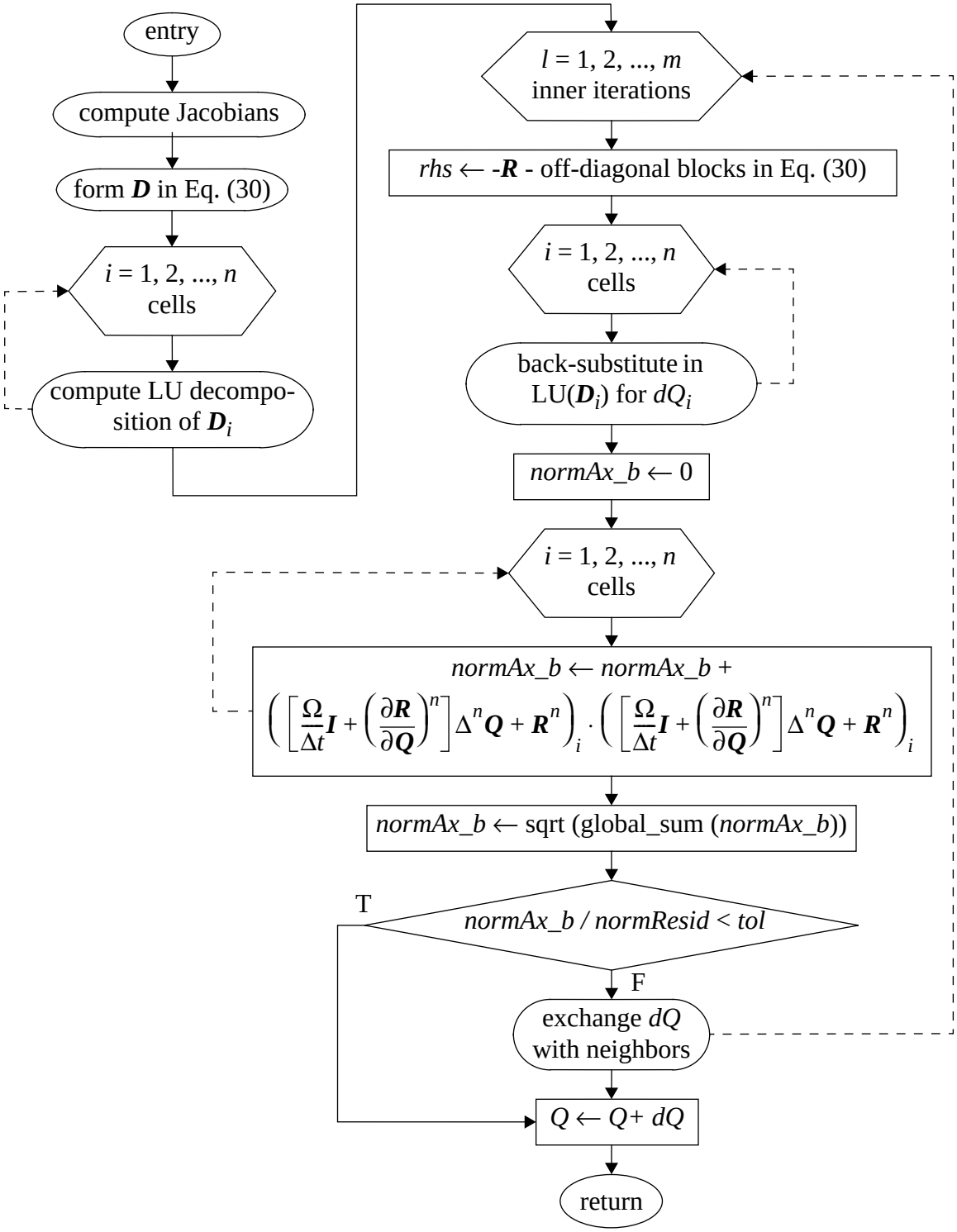


Figure 7: Flowchart for parallel Block Jacobi iteration.

ally required is an adaptive scheme for ω: since a different matrix problem is solved at every timestep, it is impossible to specify ω such that each matrix problem is both optimally accelerated and stable. Such a scheme is beyond the scope of this investigation; therefore, ω was set to unity for all of the timing runs.

Unidirectional Gauss-Seidel

This is the standard Block Gauss-Seidel iteration described above, with a slight variation: the user may specify that the iteration proceed in either the forward or reverse direction. It is thought that there may be a slight advantage when solving hyperbolic problems with the grid ordered along the flow direction to be able to iterate in the flow direction. The algorithm is identical to the Jacobi iteration outside of the inner iteration loop. The only difference is in the *RHS*, which uses data from the current iteration as well as the previous iteration, allowing one to place the assignment to *rhs* inside the back-substitution loop over the interior cells immediately below it in the Jacobi iteration.

The splitting for the Forward Gauss-Seidel iteration is

$$\begin{aligned} \boldsymbol{M} &= \Delta - \boldsymbol{L} \\ \boldsymbol{N} &= \boldsymbol{U} \end{aligned} \tag{31}$$

while that for Reverse Gauss-Seidel is

$$\begin{aligned} \boldsymbol{M} &= \Delta - \boldsymbol{U} \\ \boldsymbol{N} &= \boldsymbol{L} \end{aligned} \tag{32}$$

leading to the iteration

$$\boldsymbol{x}^{l+1} = \Delta^{-1}\left[\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{l} + \boldsymbol{L}\boldsymbol{x}^{l+1}\right]$$

for the Forward Gauss-Seidel case. As in the Jacobi iteration, Δ may be inverted outside the inner iteration loop.

A flowchart for the Forward Gauss-Seidel case is presented in Figure 8.

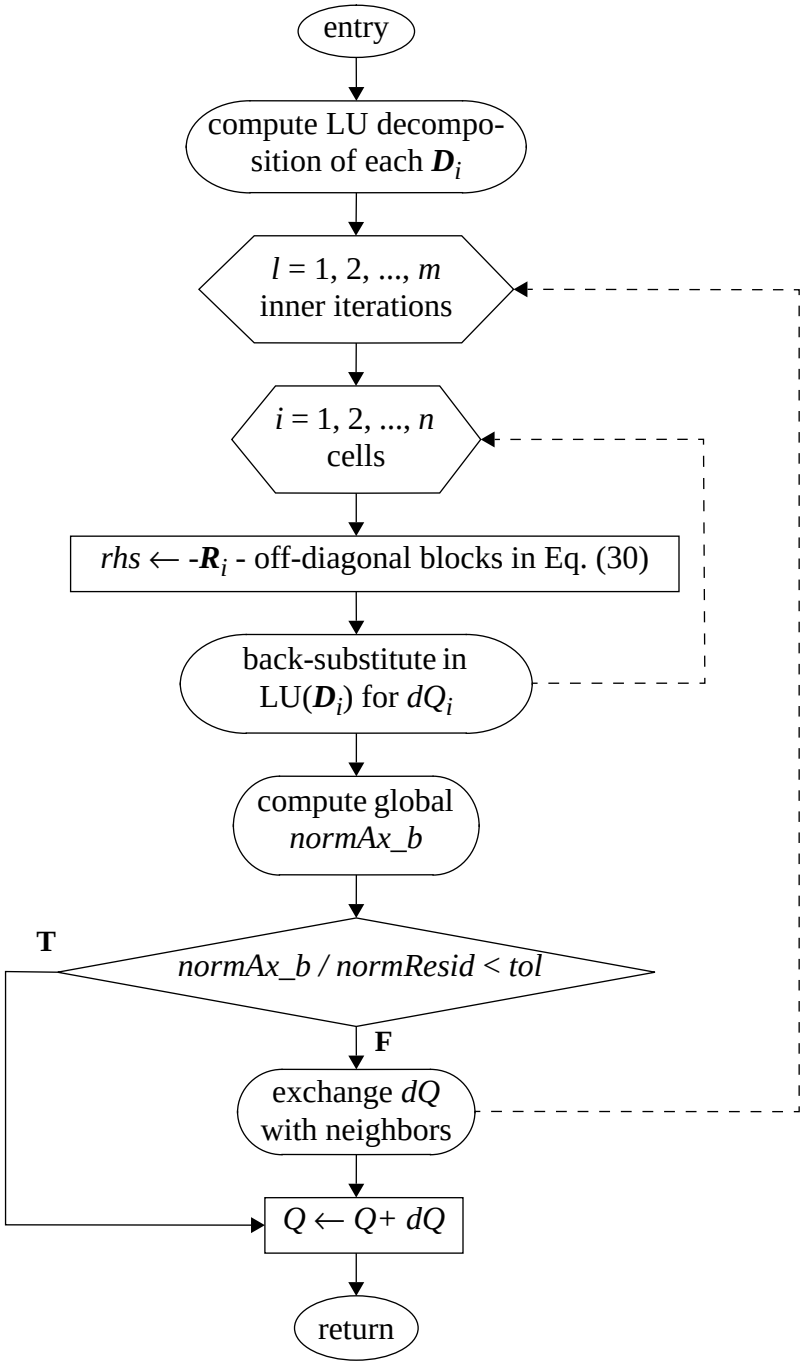


Figure 8: Flowchart for parallel unidirectional Gauss-Seidel (forward case shown).

Bidirectional (Symmetric) Gauss-Seidel (SGS)

In many problems, especially in three dimensions, it may be impossible to have the cell ID's increase monotonically in any one coordinate direction. Also, many practical problems are not hyperbolic. Therefore, solution information is propagated in both grid directions. Symmetric iteration removes the directional dependence of the inner problem by alternating sweeps through the $\boldsymbol{x}$ vector, first forward, then backward (see Golub & Van Loan 1989, pp. 513-514, for a description of Symmetric SOR; SGS is Symmetric SOR with $\omega = 1$). Each forward-backward pair is considered one inner iteration. There are really two splittings for this algorithm: one for the forward portion, and one for the backward portion. These two splittings are identical to those for Forward and Reverse Gauss-Seidel presented in Equations (31) and (32), leading to the following iteration:

$$\boldsymbol{x}^{l+1/2} = \Delta^{-1}\left[\boldsymbol{b} + \boldsymbol{U}\boldsymbol{x}^{l} + \boldsymbol{L}\boldsymbol{x}^{l+1/2}\right]$$

$$\boldsymbol{x}^{l+1} = \Delta^{-1}\left[\boldsymbol{b} + \boldsymbol{L}\boldsymbol{x}^{l+1/2} + \boldsymbol{U}\boldsymbol{x}^{l+1}\right]$$

The flowchart for this algorithm as implemented in parallel is presented in Figure 9.

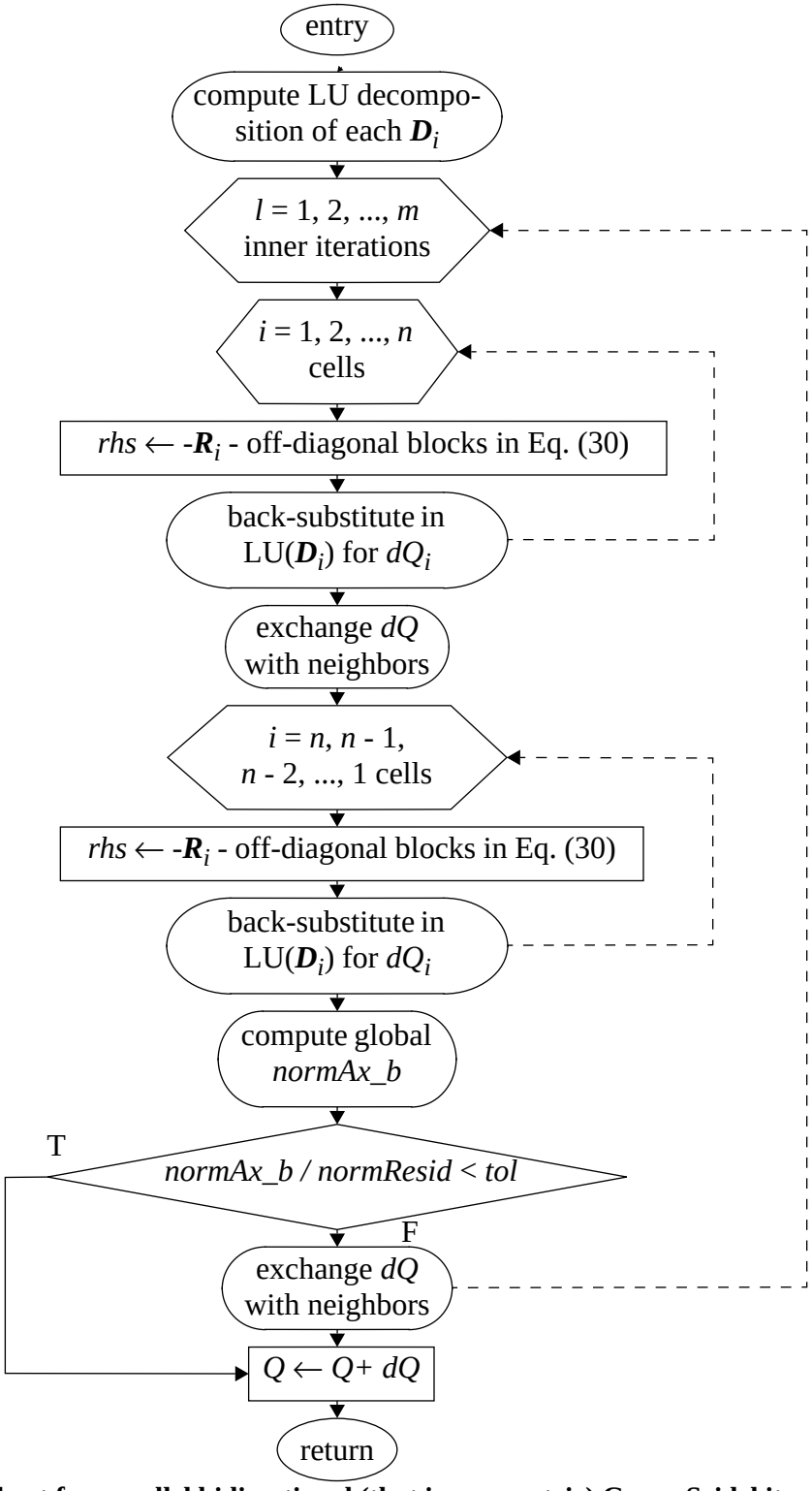


Figure 9: Flowchart for parallel bidirectional (that is, symmetric) Gauss-Seidel iteration.

Extension to Multiple Processors

PUE3D is written entirely in ANSI C; it was originally written by the author as a sequential code on Silicon Graphics workstations. Then library calls for the NX* parallel library were added after modifying the code's data structures to facilitate message passing. Finally, the code was ported to the MPI (Message Passing Interface) standard and is source-code compatible across all architectures with an MPI implementation (Snir, et al. 1996).

Different Parallel Processing Paradigms

Data Parallel

In this model, the parallelism comes entirely from the data. A data parallel code looks a lot like a sequential program, usually with some hints to the compiler on how to distribute the data. High Performance Fortran (Koelbel et al. 1993) is a data parallel language.

This paradigm is well-suited to shared-memory and vector computers but has not had much success on distributed-memory machines. Compilers for portable data parallel languages (such as HPF) are also a problem.

Shared Memory

This is both a paradigm and an architecture. In this paradigm, all processors share all memory. Therefore, a particular memory location may be accessed for reading by one processor while another processor is writing to that location. This paradigm therefore requires some kind of control, usually function calls that raise and lower semaphores, to control access. One advantage of this approach is that existing sequential code may be parallelized automatically at the loop level. A major problem is saturation of the main memory bus; true shared-memory machines (in the sense that all off-cache memory accesses take equal time) usually have at most 16 processors and therefore can be considered only "mildly" parallel.

*NX is the native message passing library on the Intel Paragon.

Message Passing

This paradigm distributes all the data among several processes with access to local data only. To access non-local data, a message must be explicitly sent and received, i.e., both the sending and receiving process must participate in the message. This paradigm has been extremely successful for two main reasons (Gropp, Lusk & Skjellum 1994, 4-8):

- Generality. This paradigm is applicable to all parallel architectures from shared-memory vector supercomputers to distributed-memory computers with high-speed communication subsystems to heterogeneous workstation clusters on slow networks.
- Performance. Since non-local data must be accessed by setting up and receiving messages, the programmer is painfully aware of the cost of using non-local data and is compelled to consider how best to distribute his data. Also, this paradigm gives the programmer maximum control over how and when his data is sent, which is lacking in the other paradigms. This makes it possible to write code that is more tolerant, for instance, of slow networks or heterogeneous processors.

Hardware Used

Two computers were used in this investigation: the Intel Paragon at the DoD's Major Shared Resource Center (MSRC) at Wright-Patterson Air Force Base (WPAFB) and the IBM SP-2 at the Maui High Performance Computing Center (MHPCC). At the time this research was performed, the Paragon had 352 compute nodes and the SP-2 had a total of 400.

Both of these machines use distributed memory, that is, each compute node has its own local memory and is connected to the other compute nodes by dedicated networking hardware. Because the data is distributed across several compute nodes, no single node has access to all of the global data. Since the nodes must communicate somehow, the performance of the interconnect can have a large effect on the overall performance of the parallel code. Although the connection topology (the way the compute nodes are connected) is different for these two machines, each machine may be viewed as being *fully connected*, i.e.,

the time to send a message to a “far” node (in the sense of the number of network elements the message must pass through) is comparable to that for a “near” node.

Intel Paragon

Each compute node of the Paragon has two identical Intel i860XP processors with 32 Mbytes of physical memory. One processor is dedicated to message passing, while the other is used solely for computation. This effectively hides most of the overhead to send a message by allowing the compute processor to continue computation while the message is being sent.

The compute nodes on the Paragon are connected in a two-dimensional mesh (see Figure 10). A disadvantage of any two-dimensional mesh topology is that the average bisection bandwidth* scales only as the square root of the number of compute nodes, so that the network performance degrades as more compute nodes are added. This is aggravated for large Paragon machines because the Paragon has a maximum of 16 nodes in the vertical direction, making the minimum bisection bandwidth constant for >256 compute nodes. Even so, the Paragon’s fast communications backplane results in excellent message passing performance.

IBM SP-2

The SP-2 is also a distributed memory architecture but is connected differently. Each cabinet (called a frame) contains a number of compute nodes connected to each other across IBM’s high-speed switch (see Figure 11), while the frames themselves are connected with one another across another high-speed switch (see Figure 12). One feature of this architecture is that it is inherently scalable: as more compute nodes are added, more communications links are also added, so that the bisection bandwidth (and therefore the expected communications performance) scales linearly with the number of compute nodes, although additional levels of intermediate switches may have to be added.

* The combined network bandwidth of all the communications channels cut by a “plane” dividing the machine logically into halves.

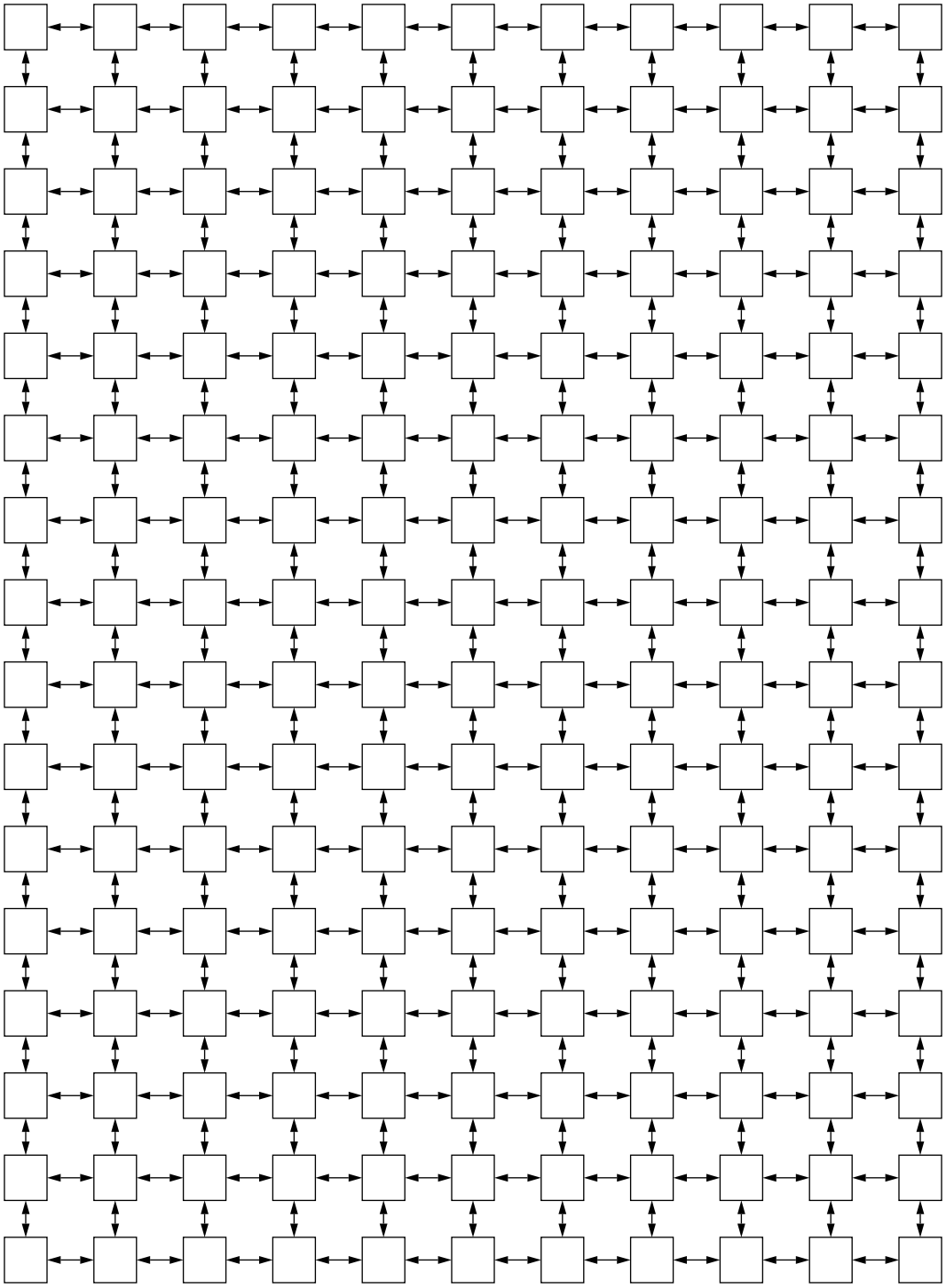


Figure 10: Connection topology for an Intel Paragon with 176 nodes. Each link is bi-directional.

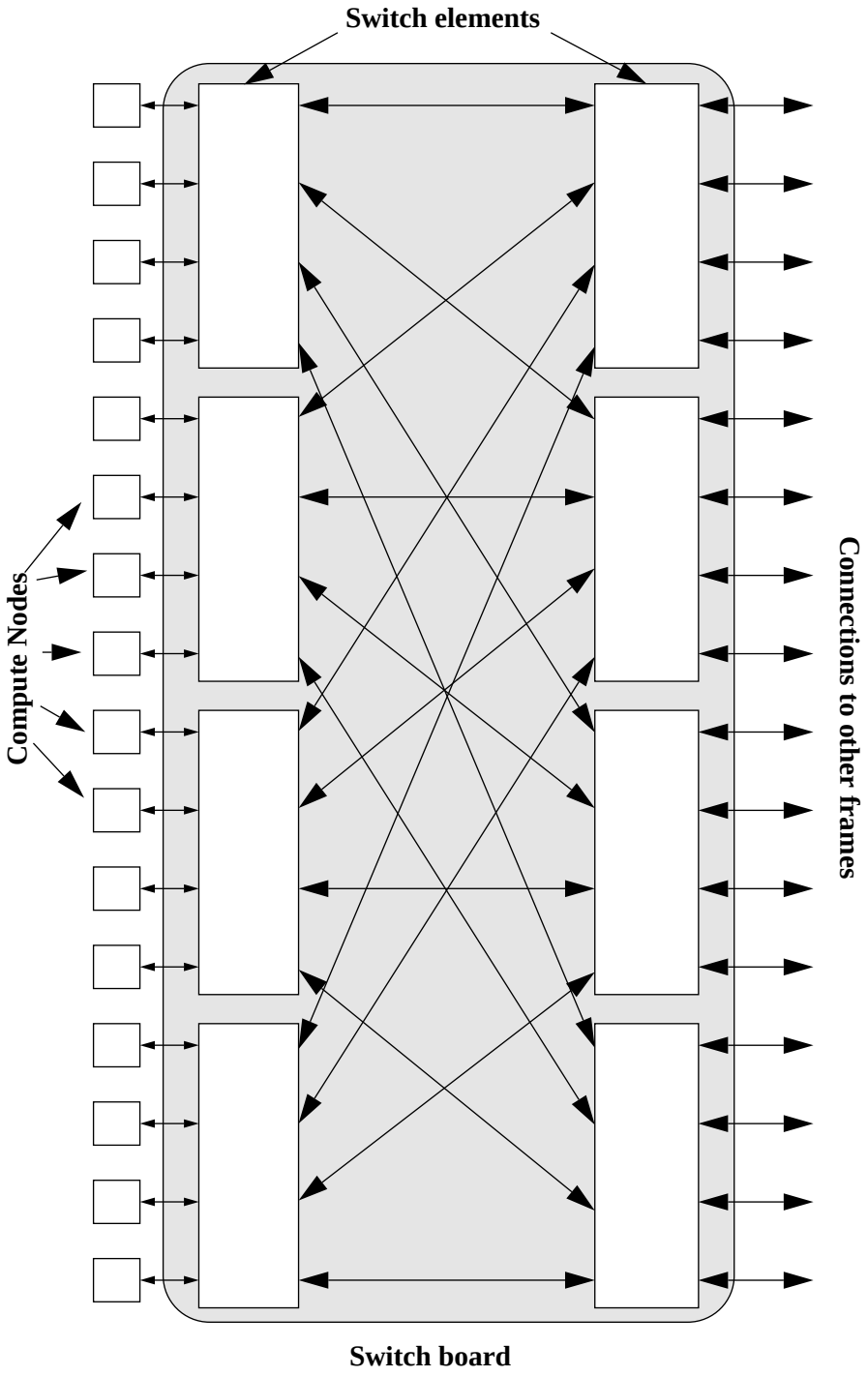


Figure 11: An SP-2 logical frame, showing the connections between compute nodes. Note that there are at least four distinct paths between any pair of nodes (IBM 1996).

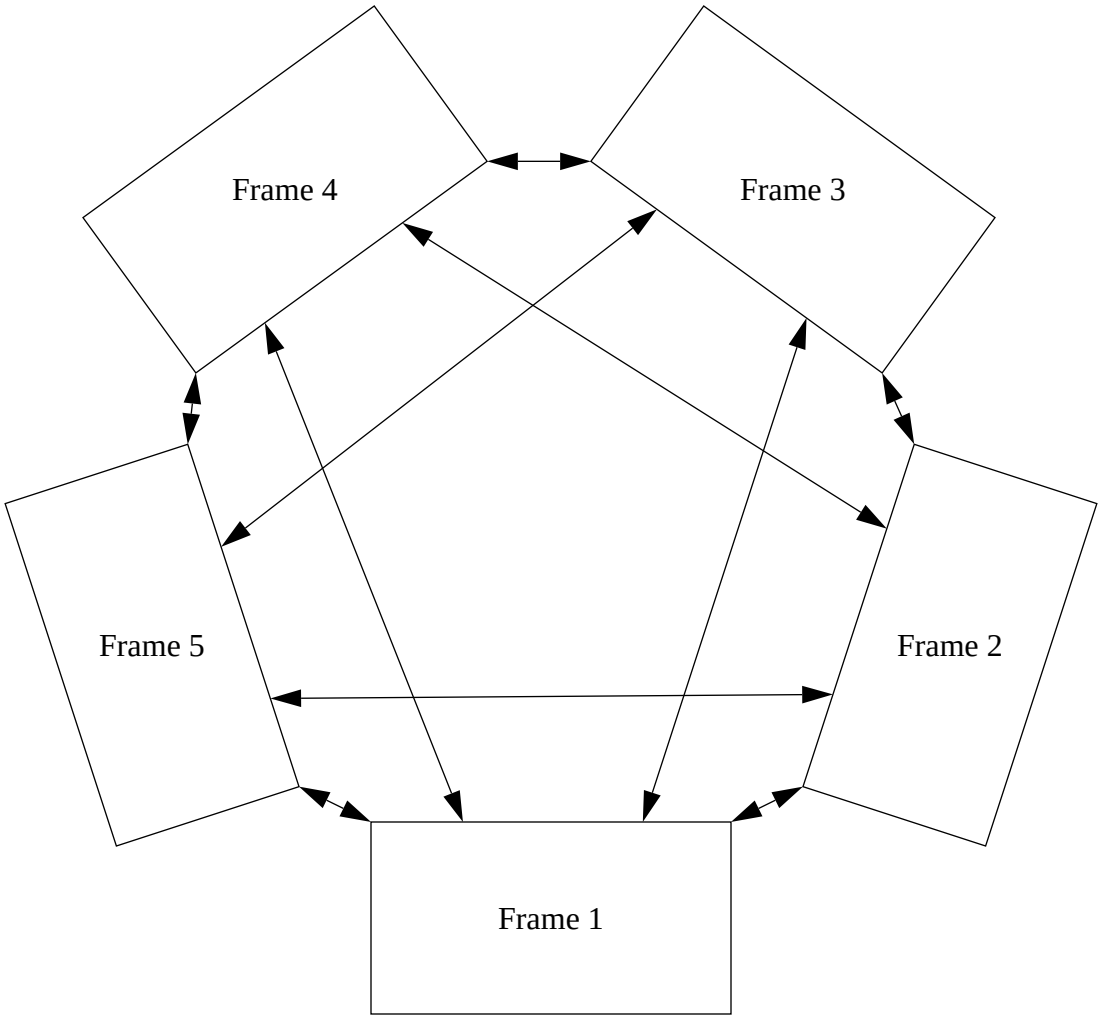


Figure 12: Interframe connections on an 80-node IBM SP-2. Each arrow represents four two-way communications links between frames (IBM 1996).

The maximum number of nodes in a physical frame is determined by the type of nodes (a logical frame always has 16 nodes, as shown in Figure 11). Up to 16 thin nodes or 8 wide nodes may be placed in a single physical frame; the characteristics of the different types of nodes as installed at the MHPCC are summarized in Table 1.

Table 1: Characteristics of the MHPCC SP-2 compute nodes.

Type	Processor	Data Cache (KB)	Memory (MB)	Bus Width	Local Disk (GB)
Thin	POWER2	64	64 or 128	64	1 or 2
Wide	POWER2	128	256 or 1024	256	2 or 4.5

Each switch element has four separate communications channels. Because each node is connected to all four channels of the switch, each node within a frame has at least four separate communication paths to every other node in the frame. This is designed to minimize the blocking of messages between compute nodes due to switch contention, giving more reliable communications performance.

Domain Decomposition

Because the communications time in a parallel code does nothing to improve the performance, it is desirable to minimize the communications overhead in any way possible. This section describes two different strategies for domain decomposition. Each of these strategies attempts to minimize communications cost by distributing the cells among the compute nodes in some optimal way.

Consider the following simple model for the time to send a message between two processors:

$$\text{time} = \text{latency} + \frac{\text{message size}}{\text{communications bandwidth}} \tag{33}$$

One way to minimize communications overhead is to perform some optimizations on the graph of the cell connectivity matrix (Pothen, Simon & Liou 1990; Simon 1991; Barnard,

Pothen & Simon 1995). This approach effectively neglects the latency by using total length of all inter-domain boundaries (which is proportional to the total length of all shared data) as the sole metric for minimization. It is appropriate for large problems, slow networks, or whenever many small messages are sent. This approach requires *a priori* knowledge of the number of domains (compute nodes) and therefore must often be done on-line to avoid keeping several grids available. Also, the more popular techniques for implementing this approach need eigenvalues of the connectivity matrix and can be quite expensive to use (Pothen, Simon & Liou 1990; Barnard, Pothen & Simon 1995). They do, however, produce the best grids for large problems (Barnard, Pothen & Simon 1995). Figure 13 gives an example of this type of partitioning for the RAE 2822 airfoil case discussed later.

Another approach for determining the optimal ordering for communications is to neglect the second term in Equation (33) and minimize the total number of messages exchanged. This is appropriate for small problems or fast communications, but only when messages can be consolidated (or "packed"), i.e., when only one or a few messages of a given type is sent to each correspondent every iteration. For example, all updates to the solution vector might be sent in a single message, rather than one message per cell. Also, since the minimization of the number of messages for this case corresponds to the minimization of the bandwidth of the cell connectivity matrix, the optimization may be done once and done off-line: only one grid is needed. Finally, a large knowledge base already exists in this area and is readily accessible in the literature. For a survey of methods, see Duff (1977); see also Duff, Erisman & Reid (1986, Ch. 8).

Because both machines used in this work have a high communications bandwidth and the problems considered here are not large, the second approach is adopted here; messages are always packed. For each of the grids used in this investigation, a Gibbs-Poole-Stockmeyer (GPS) reordering is used (Duff, Erisman & Reid 1986, 157). This algorithm is intended to produce a globally optimal minimum-bandwidth reordering. Also, since many different computer runs were performed, an effort was made to avoid increasing the number of independent variables in the investigation without a compelling reason to do so. While it is acknowledged that grid partitioning is a very important factor in parallel per-

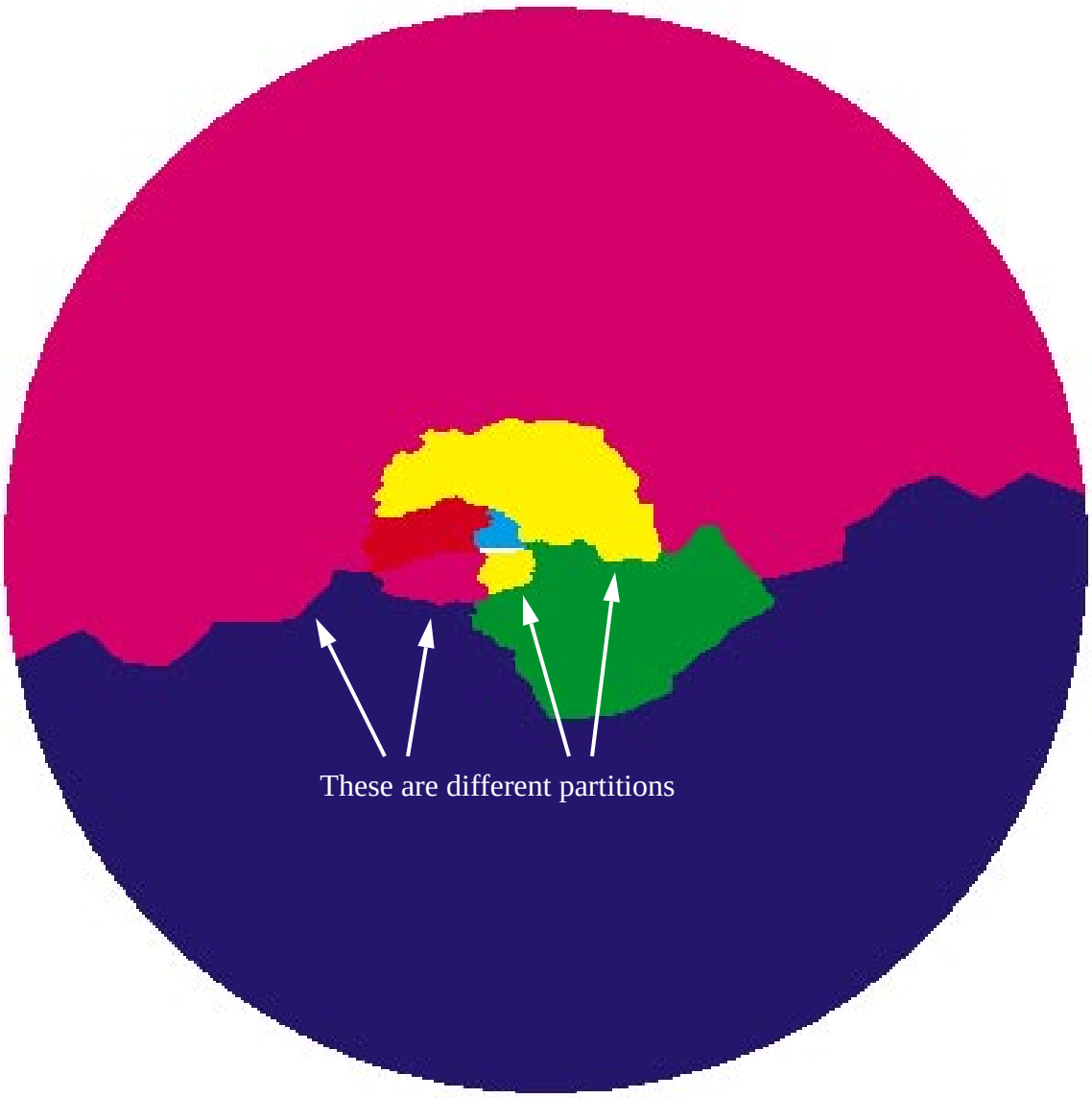


Figure 13: Typical 8-way partitioning of the RAE 2822 grid in Figure 28 using a graph partitioning algorithm. The colors correspond to different computational domains (or partitions). The airfoil is the small slit in the center of the figure.

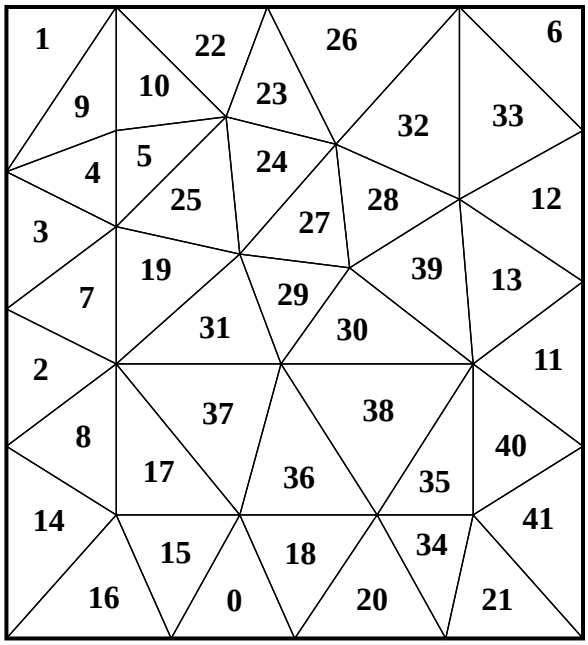
formance, it was thought to introduce too many additional considerations into the analysis and thereby muddy the waters: one might be left wondering whether the differences in efficiency between 32 and 64 nodes were due to differences in the parallelization properties of a given algorithm or due to the different grids used.

A concrete example will help to illustrate the ideas presented here. Consider the grid in Figure 14. Figure 14a shows a grid that may have been generated using the Advancing Front algorithm (Löhner & Parikh 1988), for example. Note the seemingly random placement of the cells.

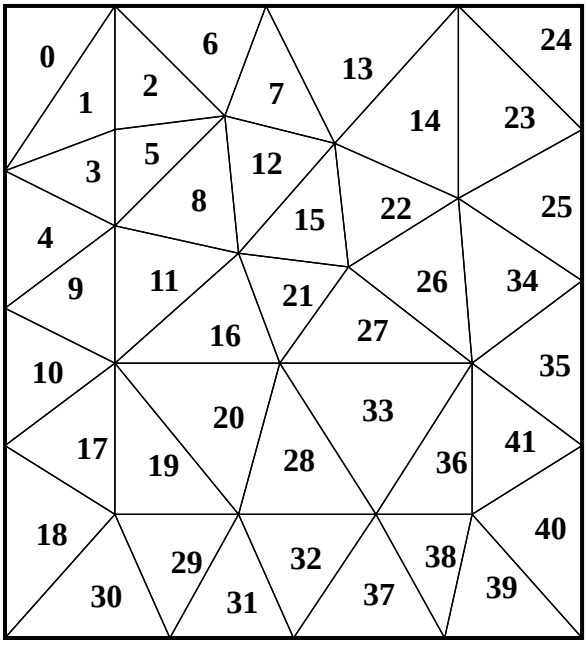
Figure 14b shows the same grid after reordering with the GPS algorithm. This algorithm chooses a starting cell with a small number of neighbors and labels it as the first cell (cell 0 in the C-style notation used here), forming a *level set* of one cell. Then all of cell 0's neighbors are numbered sequentially; this forms the second level set. The next level set is formed from the neighbors of the cells in the second level set that are not members of either of the first two sets. New cell ID's are then assigned sequentially to this set, and the algorithm proceeds until the entire grid is processed in this way. This is the Cuthill-McKee algorithm (Duff, Erisman & Reid 1986, 153-157), and is strongly dependent on the choice of starting cell. The GPS algorithm addresses this shortcoming by repeating this process using each of the cells in the final level set as a starting cell for a new reordering. This restarting is repeated until there is no further improvement in the bandwidth of the cell connectivity matrix. Figure 15 graphically shows the effect of reordering the example in Figure 14, while Figure 16 shows the partitioning for the same grid as in Figure 13. Note that the grid in Figure 16 has only nearest-neighbor communication, i.e., each compute node shares data with at most two other compute nodes. This minimizes the total number of messages.

Distribution of the Left-Hand Side Jacobian Matrix in Implicit Time Integration

What follows is a summary of work performed on the Intel Paragon at WPAFB last year as part of this investigation. The complete paper may be found in Bruner & Walters (1996); this summary is included for completeness. Second-order accurate fluxes and the Symmetic Gauss-Seidel (SGS) algorithm described above were used for all timing runs.

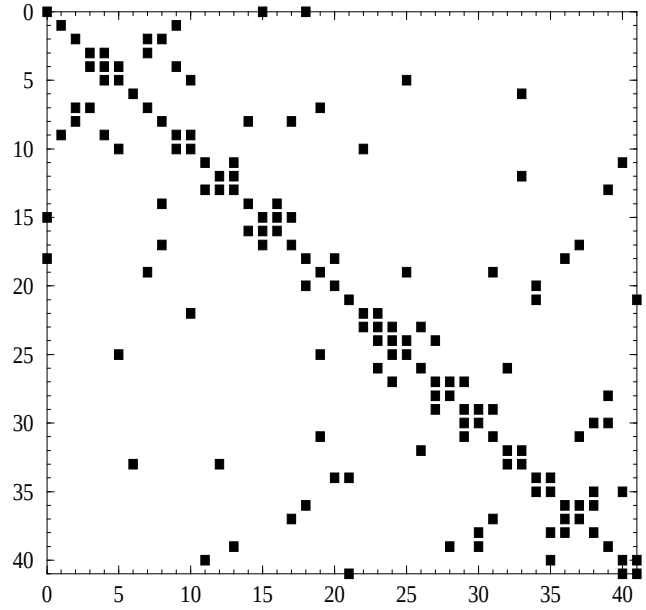


a) before reordering.

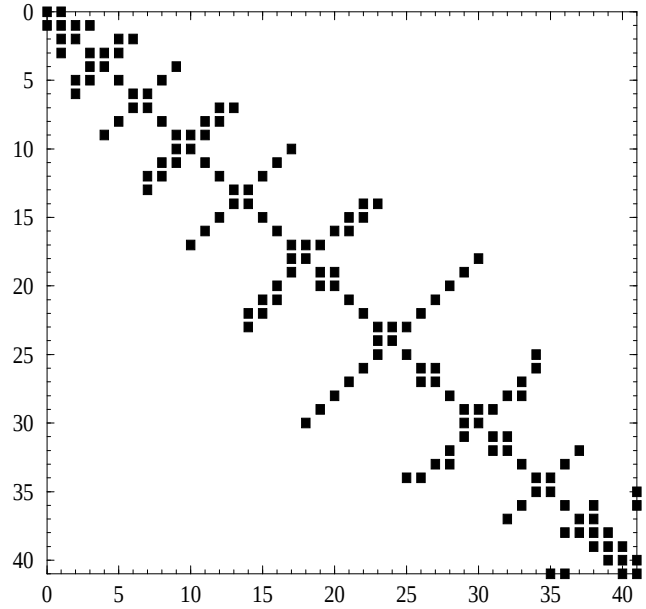


b) after reordering.

Figure 14: Example grid showing global cell IDs before and after reordering.



a) before reordering



b) after reordering

Figure 15: Cell connectivity matrix of the grid in Figure 14 before and after reordering with the Gibbs-Poole-Stockmeyer algorithm. Each point represents a face shared by the two cells given by the coordinates on the axes.

Figure 16: 8-way partitioning from the Gibbs-Poole-Stockmeyer reordering. This is the same grid as in Figure 13.

When implementing implicit time-integration schemes on a distributed-memory computer, compute nodes must have knowledge of the current solution and residual vector in the first non-local cell across a shared face; the converged solution is dependent only on these quantities. To enhance (or, in some cases, achieve) convergence when computing on multiple compute nodes, several ways of dealing with the non-local blocks in the left-hand side matrix and residual vector may be considered.

Each compute node reads its own portion of the grid file at startup. Cells are divided among the active compute nodes at runtime based on cell ID (Figure 17); only faces associated with local cells are read. Faces on the interface surface between adjacent computational domains are duplicated in both domains. Solution variables are communicated between domains at every timestep (this ensures that the computed solution is independent of the number of compute nodes). Fluxes through the faces on the interface surface are computed in both domains. Communication of the solution across domains is all that is required for first-order spatial accuracy, since $\boldsymbol{Q}_L$ and $\boldsymbol{Q}_R$ are simply the cell averages to first order. If the left and right states are computed to higher-order, then $\boldsymbol{Q}_L$ and $\boldsymbol{Q}_R$ are shared explicitly with all adjacent domains. The fluxes through each face are then computed in each domain to obtain the residual for each local cell.

Description of the domain interface region

For simplicity, consider the case of two processors, A and B, and a two-dimensional grid to be shared between them (Figure 17). Figure 18 shows a close-up view of the domain interface region of the coefficient matrix; each point represents a non-zero 5×5 block in the matrix. Off-diagonal blocks in zones II and III relate cells in both domains. Because A and B share solutions at every timestep, A has full knowledge of all of the blocks in zones I, II, and III, while B has full knowledge of the blocks in zones II, III, and IV. Because A and B communicate through the solution vector, it does not affect the steady-state solution for A to neglect blocks in zones II, III, and IV (similarly for B); however, one might expect the convergence to be enhanced by somehow including the effect of these blocks.

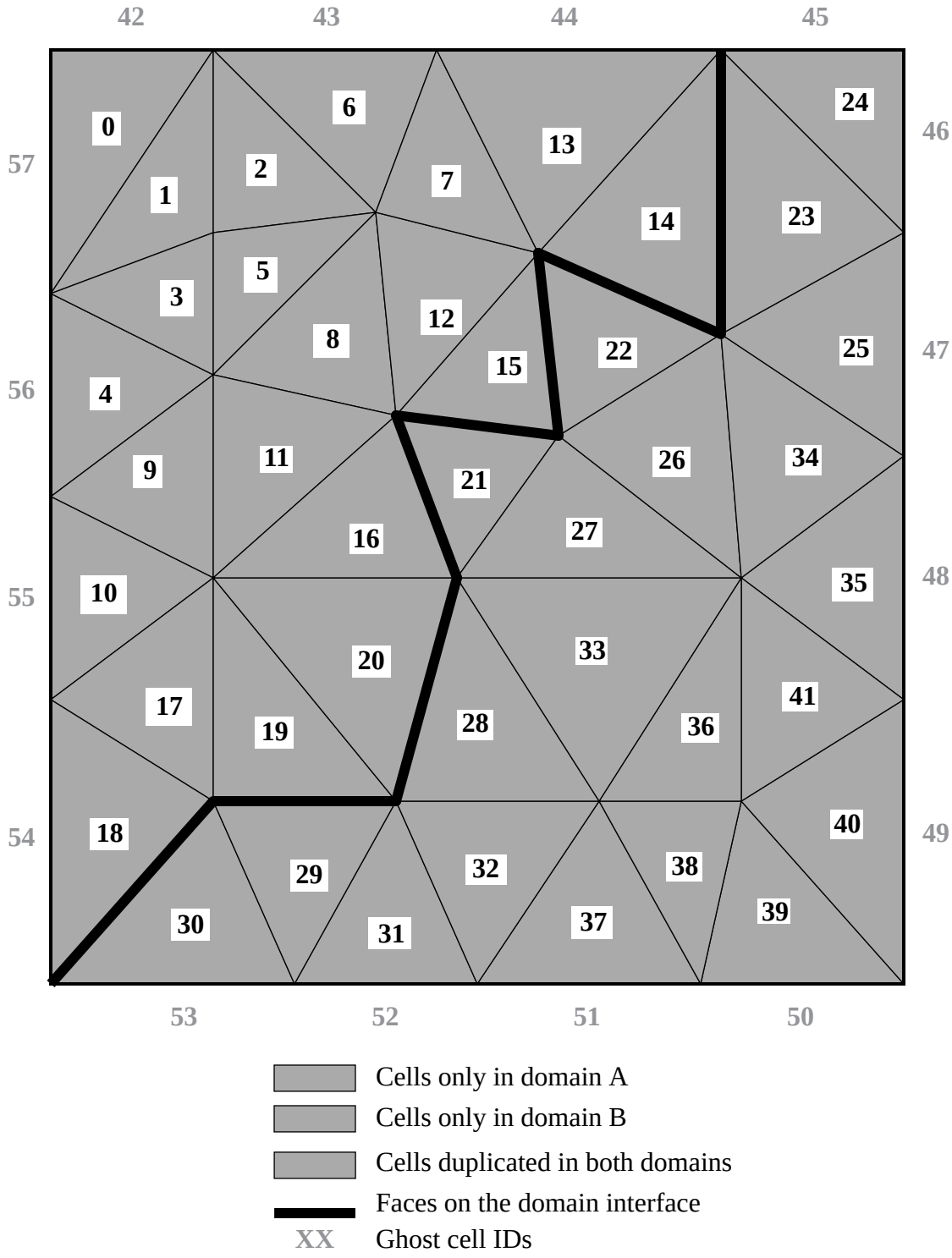


Figure 17: Geometric representation of the domain interface region, showing global cell IDs.

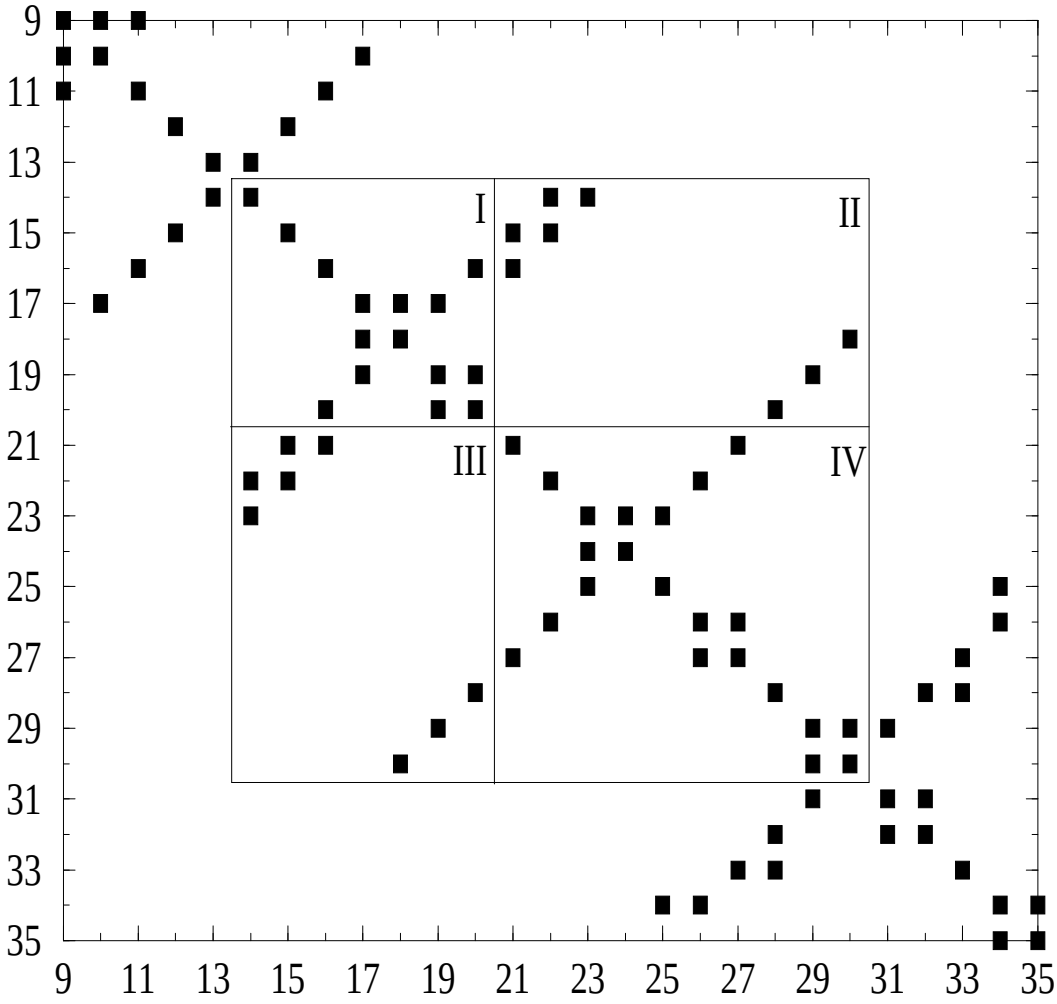


Figure 18: Close-up of the domain interface region in the coefficient matrix.

Using only the completely local blocks in the boxed-in area (zone I for A and IV for B) and neglecting all non-local blocks is referred to as Level I approximation.

A slightly more sophisticated way of dealing with the non-local blocks is for A to again neglect the off-diagonal blocks in zones II, III, and IV, but to approximate the diagonal blocks and right-hand side of zone IV by the face-adjacent neighbor diagonal blocks from zone I and by the face-adjacent parts of A's residual vector. Then we can solve for $\Delta^n \boldsymbol{Q}$ corresponding to B's cells outside the inner iteration loop. This is simply one block-Jacobi iteration for the non-local terms in $\Delta^n \boldsymbol{Q}$. Given these terms in $\Delta^n \boldsymbol{Q}$, we can include the approximate effects of the off-diagonal blocks in zone II, which pass over to the right-hand side for relaxation schemes. B would approximate zone I quantities to estimate the effect of blocks in zone III. This is referred to as Level II approximation.

In a similar way to Level II, we can share the diagonal blocks in zones I and IV, as well as the corresponding residual vectors, by explicit communication between A and B. Then we can solve for $\Delta^n \boldsymbol{Q}$ as before. This is Level III approximation.

Finally, if A communicates to B the components of $\Delta^n \boldsymbol{Q}$ which are non-local to B at every inner iteration, and vice versa, then the parallel implementation should have convergence similar to the serial version (for the Block Jacobi scheme, convergence is identical). However, it might be expected that the substantial increase in communications overhead associated with this scheme may more than offset any gains. Note that this scheme is the only scheme which includes the whole effect (through the evolving $\Delta^n \boldsymbol{Q}$) of all of the blocks in the domain interface region. This scheme is Level IV approximation.

The RAE 2822 airfoil and the ONERA M6 wing cases described in the next chapter were used to evaluate the performance of each of these schemes; the pertinent grid parameters for all test cases may be found in Table 2. Neither of these cases would fit in physical memory on one compute node of the Paragon. For this portion of the research, the performance on one compute node was calculated from the performance on a Silicon Graphics *Indigo*^{2} and the ratio of CPU speeds between the SGI machine and one node of the Paragon.

This ratio was calculated from the time to converge a 2-D supersonic ramp problem on each machine using one compute node.

Both the Paragon and the Indigo make use of a hierarchical memory system to improve performance. Both machines have an on-chip cache of static RAM that can be accessed in several clock cycles in addition to main memory, which takes tens of cycles to access. Due to the different cache sizes and different relative performance of the main and cache memory, the single-node performance of each machine depends on problem size and the order of memory accesses, so that the *actual* single-node performance of the Paragon differ by a variable factor from the Indigo performance; the method of calculating the single-node Paragon performance assumes a constant performance ratio. Therefore, parallel efficiencies of more than 100% are possible (see Figure 23).

For each case and approximation level, the CFL number used was the largest that could be run on 32 compute nodes, but there was no “tweaking”: the CFL number used was the largest of the sequence 1, 2, 5, 10, 20, 50, etc., that would converge. The CFL number was not changed as the number of compute nodes varied. This makes it easier to spot trends, but is somewhat sterile: in a practical environment, the CFL number would always be the largest runnable for the number of compute nodes.

The RAE 2822 timing runs converged the inner problem to 10^{-7} or 100 inner iterations, whichever came first. The outer problem residual was converged to 10^{-6} from the converged first-order solution. Due to CPU-time constraints, the timing runs for the M6 wing were converged only two orders of magnitude from the converged first-order solution. Also, the inner problem was only converged to 10^{-5} and was limited to a maximum of 50 iterations. The RAE 2822 case needed no limiting; the extended Van Albada limiter described above was used for the ONERA M6 case.

Figures 19-24 show the parallel performance of the various schemes for both cases. The performance of the Level II approximation was so poor compared to the other methods that results for this scheme are not presented here. The reason for the poor performance was the very low CFL number required for convergence. Contrary to what might be expected, the performance of the Level IV approximation is better than the other two schemes for all

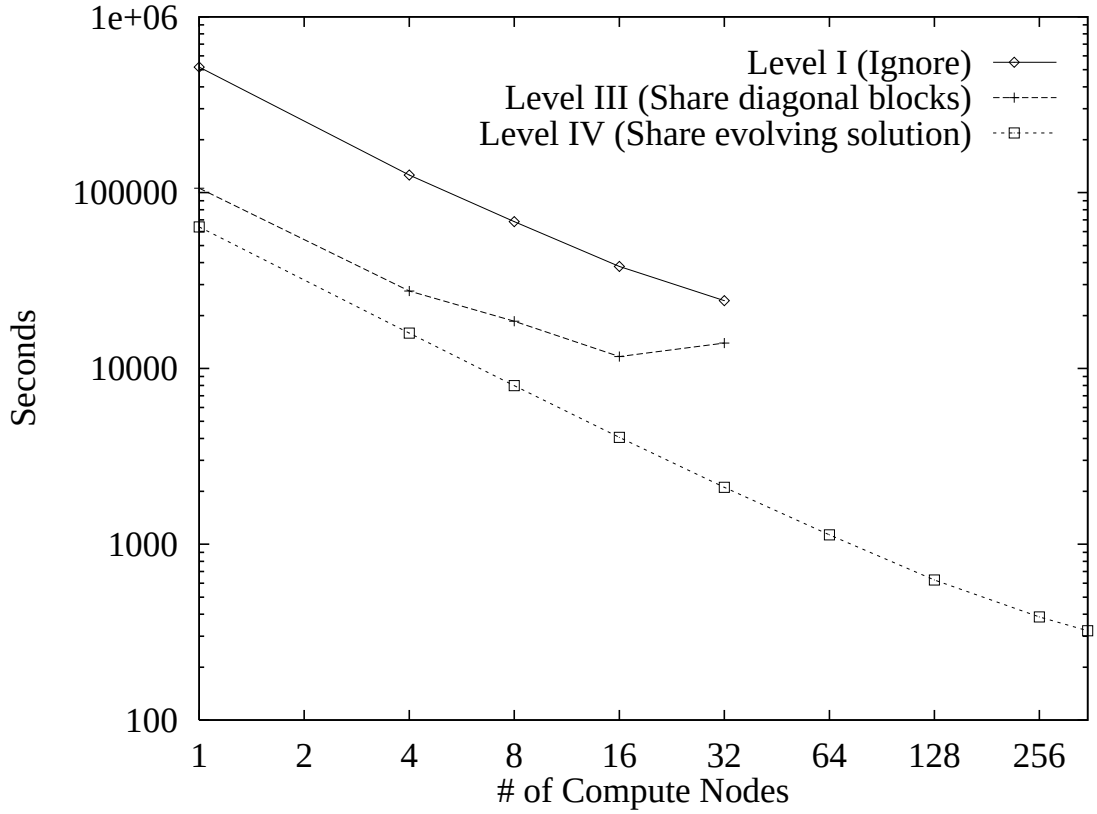


Figure 19: Convergence time using different approximation schemes on the Intel Paragon, RAE 2822 airfoil case.

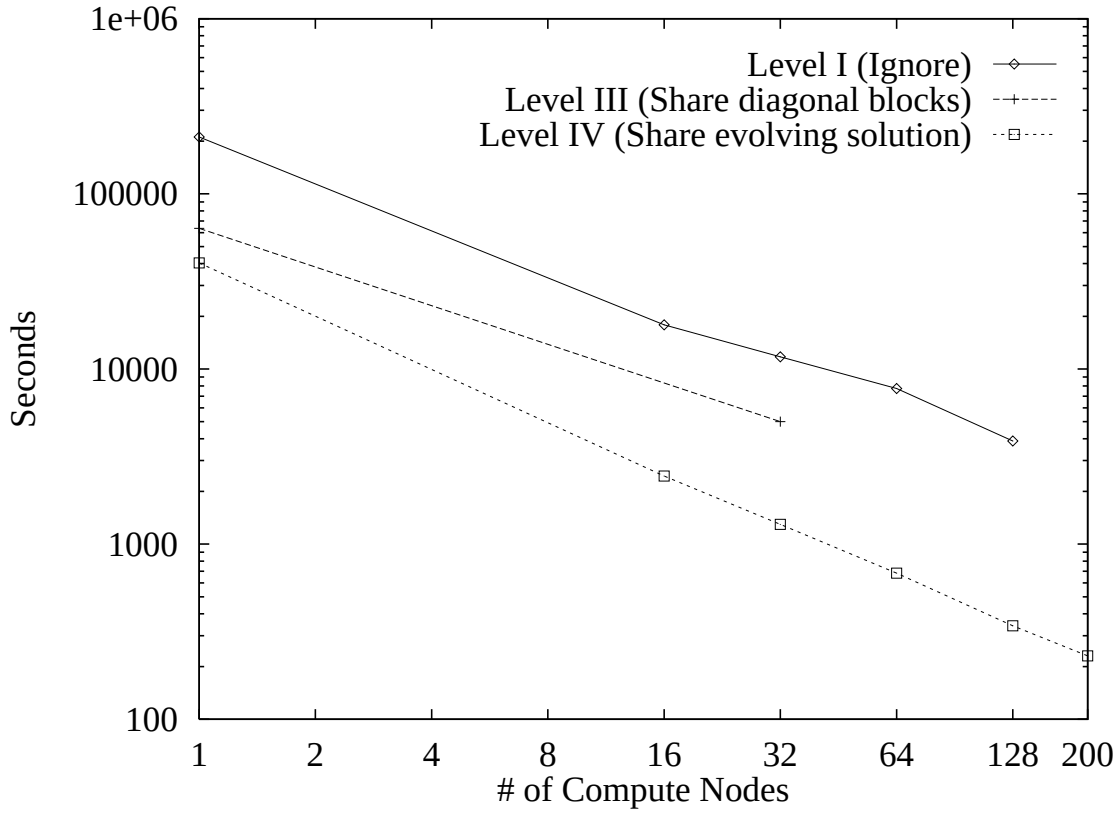


Figure 20: Convergence time using different approximation schemes on the Intel Paragon, ONERA M6 wing case.

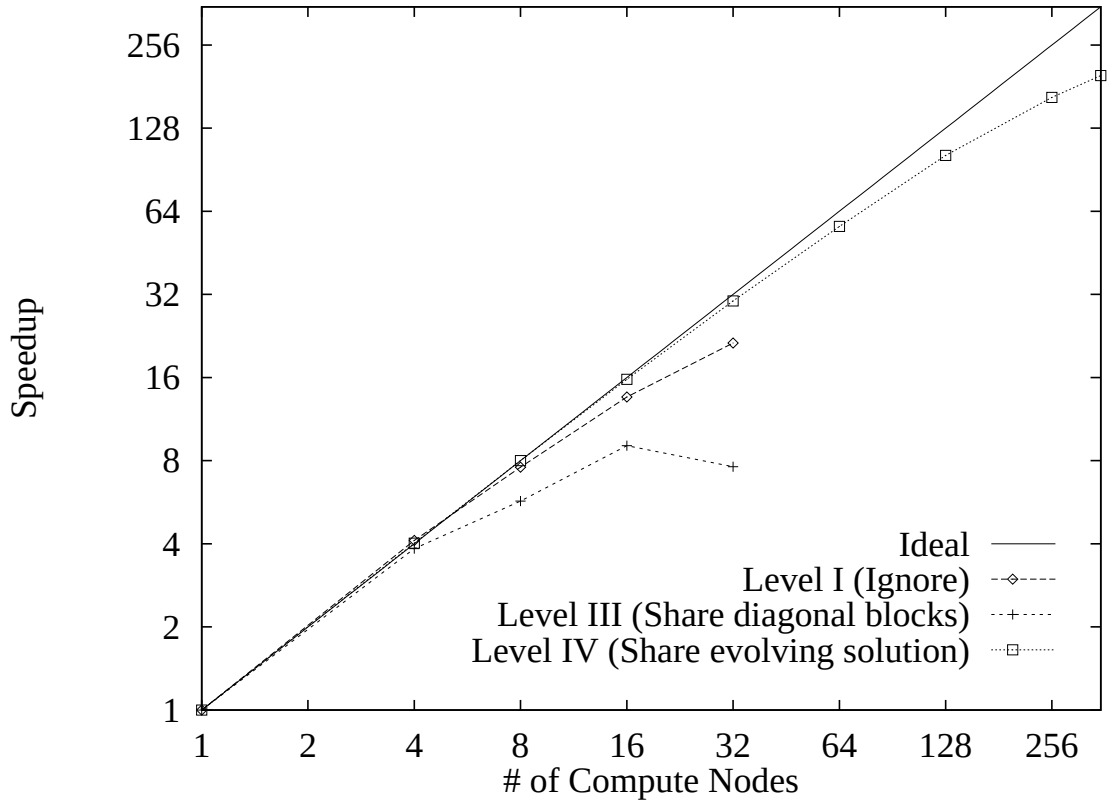


Figure 21: Parallel speedup using different approximation schemes on the Intel Paragon, RAE 2822 airfoil case.

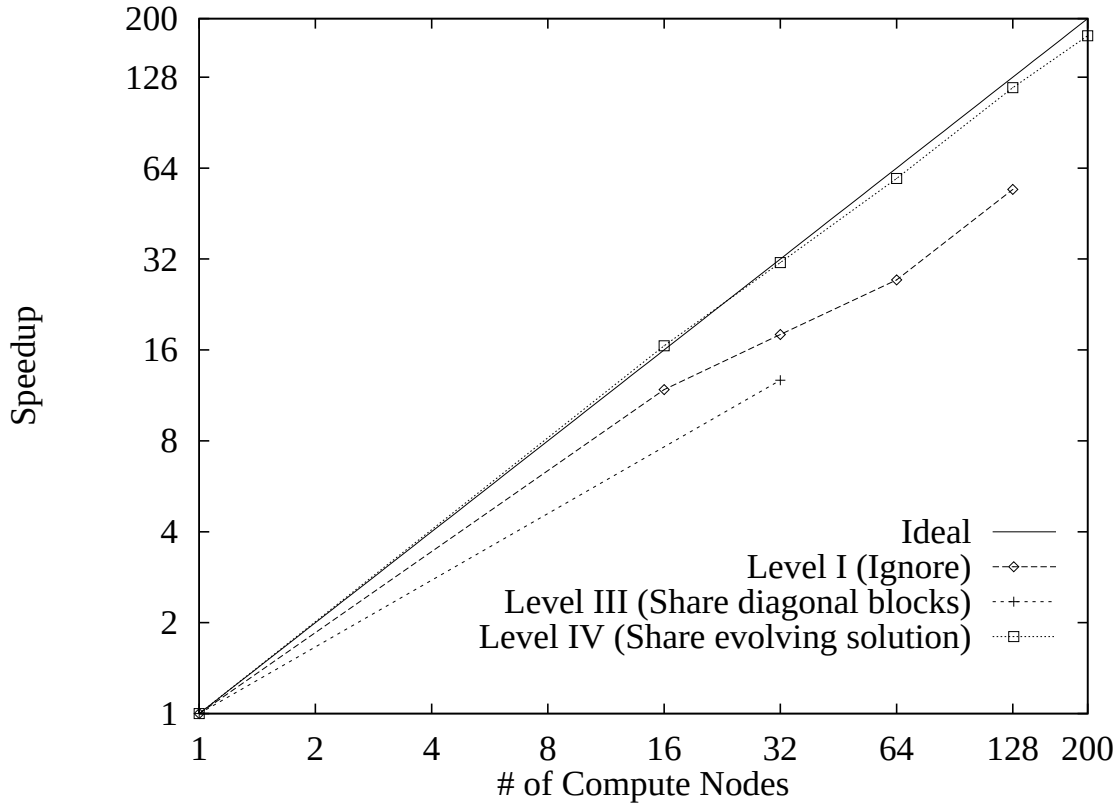


Figure 22: Parallel speedup using different approximation schemes on the Intel Paragon, ONERA M6 wing case.

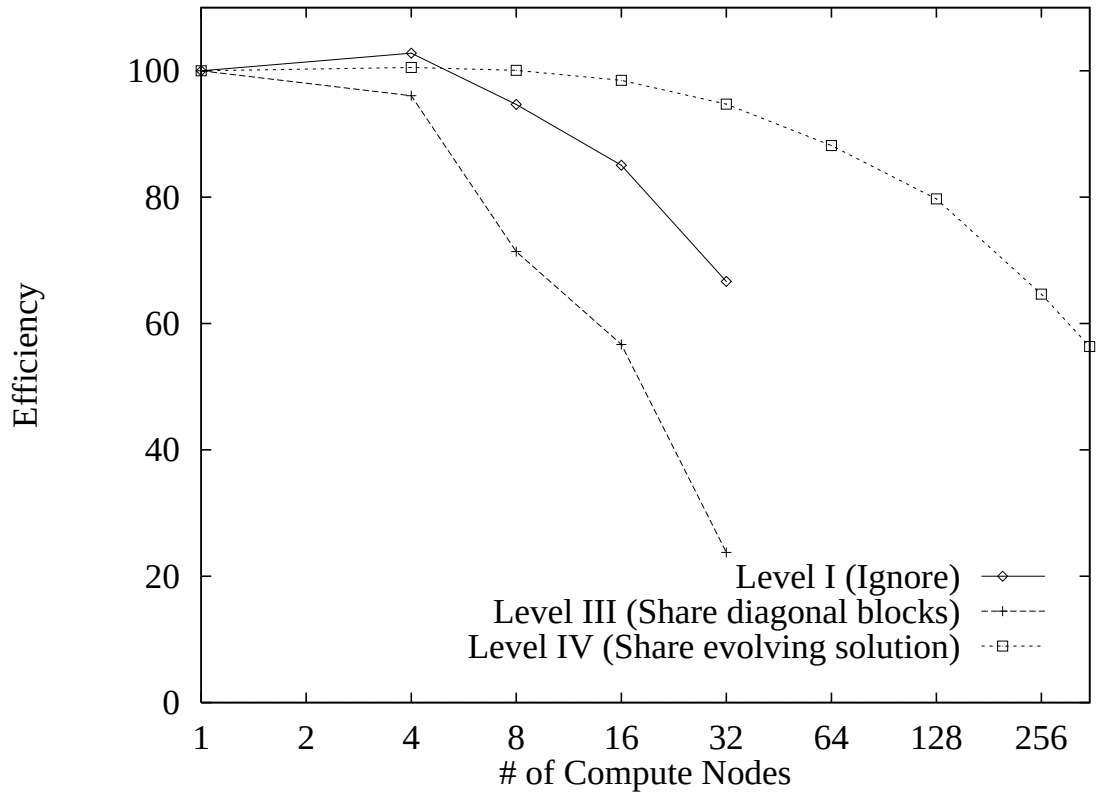


Figure 23: Parallel efficiency using different approximation schemes on the Intel Paragon, RAE 2822 airfoil case.

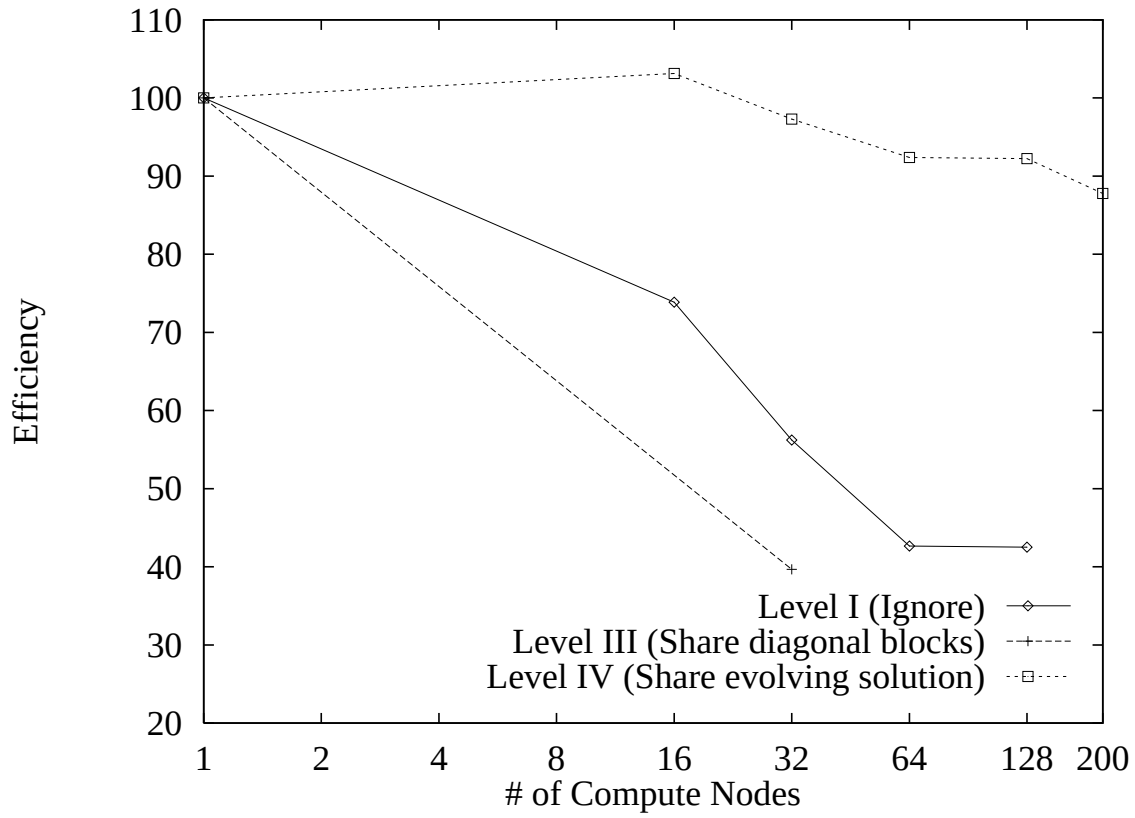


Figure 24: Parallel efficiency using different approximation schemes on the Intel Paragon, ONERA M6 wing case.

measures of performance. This scheme permitted the largest CFL numbers of all of the schemes compared (5 for Level I, 2 for Level II, 50 for Level III, and 200 for Level IV on the RAE 2822 airfoil case presented here); this is the reason for the good performance of this scheme despite its high communications overhead. It is also somewhat surprising that the Level I scheme performed as well as it did, being more robust (although still slower) than the Level III scheme for the ONERA M6 case.

Although the RAE 2822 case was run successfully on all 352 compute nodes of the Paragon used in this study, the M6 case could not be run on more than 200 nodes. Notice in Figure 16 that the partitions are very thin, i.e., there are many cells adjacent to a domain boundary. This is even worse in three dimensions. As the grid is distributed across more and more compute nodes, it is possible that one or more cells in a given domain may become completely surrounded by cells from neighboring domains. The author suspects that some problem with such orphan cells may be the cause of the problem with the M6 on more than 200 nodes, but has been unable to confirm this hypothesis since 128 nodes is the most that may be run on this Paragon without special permission.

In spite of the large communications overhead associated with Level IV approximation, the time-to-converge is superior to all other schemes on the Paragon. This is because the convergence behavior on any number of compute nodes is almost identical to the behavior on one compute node, permitting utilization of very large CFL numbers.

Also, due to the Paragon's high-speed communications backplane, the communications costs were very low, contributing to the high efficiency of this scheme; therefore, the non-local portions of the $\Delta^n \boldsymbol{Q}$ vector are exchanged at the end of every inner iteration for the balance of this investigation (Level IV approximation).

Data Structures

In the sequential version of the code, the cells are ordered so that all of the interior cells appear first in the solution and residual vectors, followed by the "ghost" cells, which store boundary condition information (see Figure 25a). The ghost cells are external to the

domain as far as the iteration is concerned and are simply a convenient mechanism to apply boundary conditions.

In the parallel code, there are two types of cells stored locally that require special treatment: the ghost cells local to the domain (if any) and the cells on the other side of the domain interface. Since a minimal change to the sequential code was sought, the interface cells were simply added to the end of the local cells (see Figure 25b).

For example, consider the domain in Figure 17. The global cell ID's for this domain map to local ID's as shown in Figure 26. Two things are noteworthy about Figure 26: first, the ordering of interface cells in both the receiving and sending domains is the same; second, the interface list on the receiving compute node is contiguous, even though there may be gaps in the sending node's list of interface cells. Also, although it is not shown in Figure 26, these properties hold for any number of domains, i.e., all of the received cells for each neighboring domain are contiguous. Thus it is possible for the receiving node to receive the interface cells directly into the local cell array, obviating the need for a receive buffer for cell data. This avoids an additional memory-to-memory copy and saves buffer space.

a) sequential code

b) parallel code

Figure 25: Schematic of cell ordering in the solution and $\Delta^n Q$ arrays.

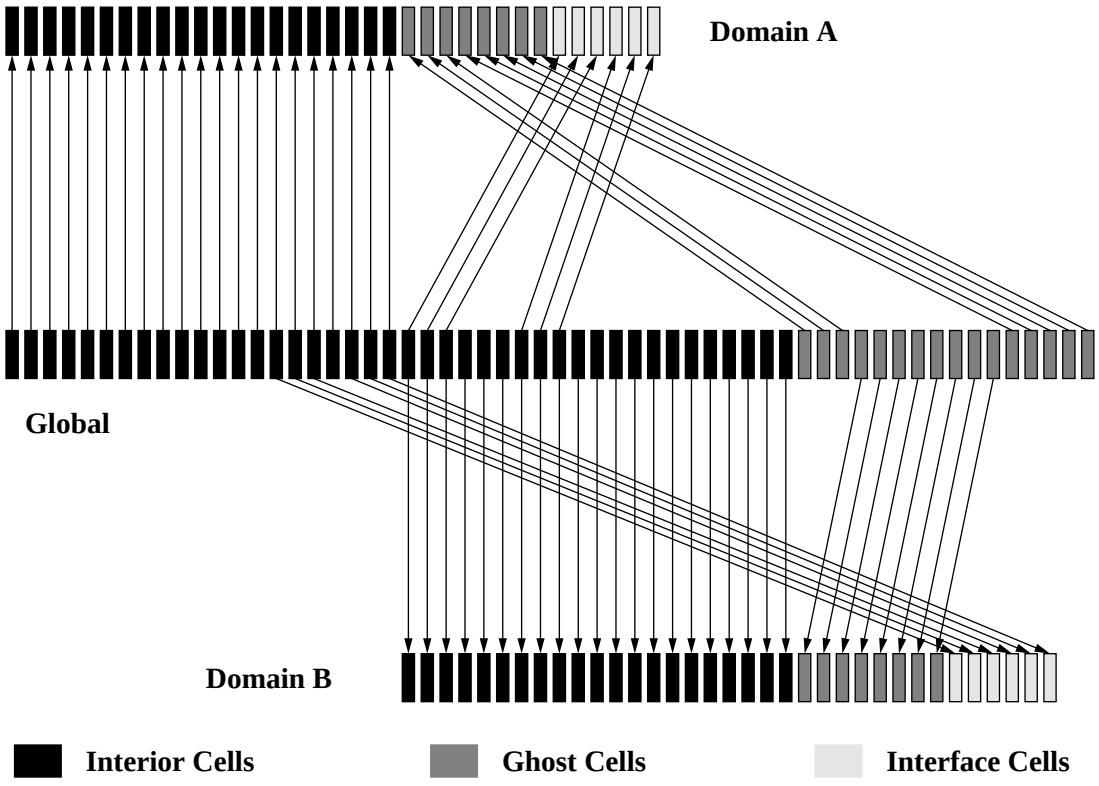


Figure 26: Mapping of the domain in Figure 17 across two compute nodes.

Test Cases

Several two- and three-dimensional test cases were run on different numbers of compute nodes, both to assess the parallelization properties of the algorithms considered here and to demonstrate the accuracy of the code. Numerical results are compared with experimental data wherever possible. It is important to note that the steady-state solution is always the same no matter how many compute nodes are used to compute it.

To ensure the broadest applicability of the results, the test cases were selected to include a broad range of flows: a 2D supersonic case, a 2D transonic case, a 3D subsonic case with strong vortices, a 3D supersonic case, and a 3D transonic case. In short, the test cases represent most types of flow for which the Euler equations with a perfect-gas equation of state are valid.

Table 2 contains a summary of pertinent grid parameters for each case, and Table 3 gives the CFL numbers and limiter parameter K used for all of the timing runs for each case. For each case, split flux boundary conditions are used at inflow and outflow boundaries, while full flux boundary conditions are used for the walls. The wall boundary condition turns the flow to eliminate the normal component of velocity rather than simply subtracting out the normal component — this preserves the stagnation enthalpy. Finally, for the three-dimensional cases, the symmetry boundary condition is implemented as a split-flux tangency BC. Implicit boundary conditions are used throughout. With the exception of the Hummel delta wing, all of the results presented in this chapter were computed using second-order fluxes with the extended Van Albada limiter described earlier and in Bruner & Walters (1996). The Hummel wing solution presented here was computed with no limiting.

Two-dimensional Cases

PUE3D is a three-dimensional code. Therefore each two-dimensional geometry is represented as a grid of finite thickness cells. There are no fluxes computed in the out-of-plane direction, and the characteristic length is independent of the domain thickness. The cell volumes are computed using only the areas of the faces for which fluxes are computed (Bruner 1995).

Table 2: Grid characteristics for each test case. The numbers in parentheses reflect the number of active faces in the computation, i.e., those for which fluxes are computed (two-dimensional cases only).

Case	Nodes	Faces	Cells	Boundary Faces
Supersonic bump	4,978	16,723 (7,233)	4,745	9,721 (231)
RAE 2822	12,050	41,569 (17,873)	11,848	23,898 (202)
Hummel delta wing	12,414	127,527	61,908	7,422
Hummel delta wing (fine grid)	76,865	837,936	412,567	25,604
Analytic forebody	77,531	221,400	72,000	10,800
ONERA M6 wing	17,992	196,652	96,207	8,476

Table 3: Run parameters used in the timing runs for each case.

Case	1st Order		2nd Order		
	CFL constant	CFL slope	CFL constant	CFL slope	Limiter parameter
Supersonic Bump	∞	0	20	2	2
RAE 2822	20	1000	100	20	None
Hummel delta wing	100	100	∞	0	None
Analytic forebody	10	200	∞	0	0
ONERA M6 wing	10	100	200	0	500

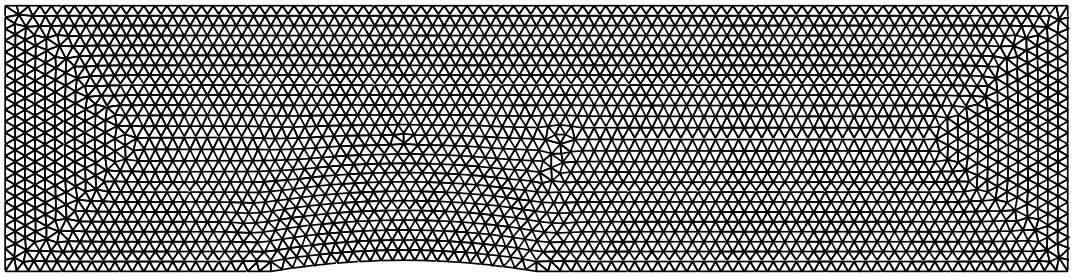
Supersonic Bump

This is a simple example of purely supersonic flow through a channel. Mach 1.65 flow enters from the left, forming a shock at the leading edge of a circular-arc airfoil (the "bump"). This shock is then reflected from a wall at the upper boundary of the domain and interacts with the expansion fan from the upper surface of the airfoil and the shock formed at the trailing edge. The inflow boundary was fixed at the freestream conditions; the outflow boundary is simple first-order extrapolation of the solution in the first interior cells. This case is the one used to show quadratic convergence with numerical Jacobians and implicit boundary conditions in Figure 5 on page 25. Second order pressure contours for this case are shown in Figure 27 along with the computational grid.

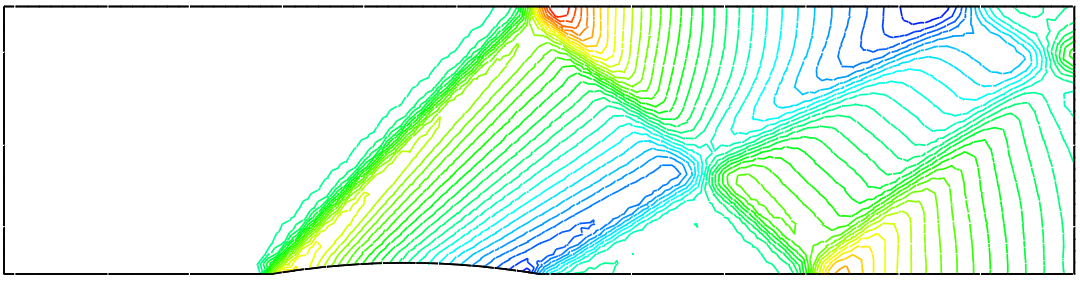
The cell IDs for this grid decrease uniformly in the flow direction. We should therefore expect the Reverse Gauss-Seidel algorithm to have the best convergence properties for this case.

RAE 2822 Airfoil

This case is a good example of two-dimensional transonic flow. A shock forms on the upper surface of an RAE 2822 airfoil at 3.19˚ angle of attack and Mach 0.73. Characteristic boundary conditions are applied all along the outer boundary. Excellent experimental data for this case may be found in Cook, et al. (1984). The computational grid is shown in Figure 28. Second-order pressure contours for the computed solution are shown in Figure 29, while Figure 30 shows the comparison with experiment.

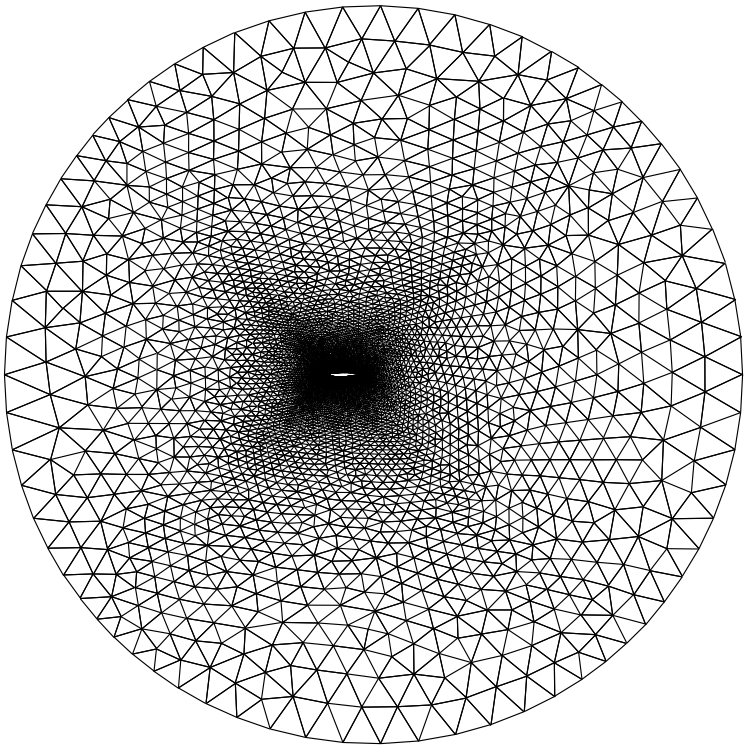


a) grid

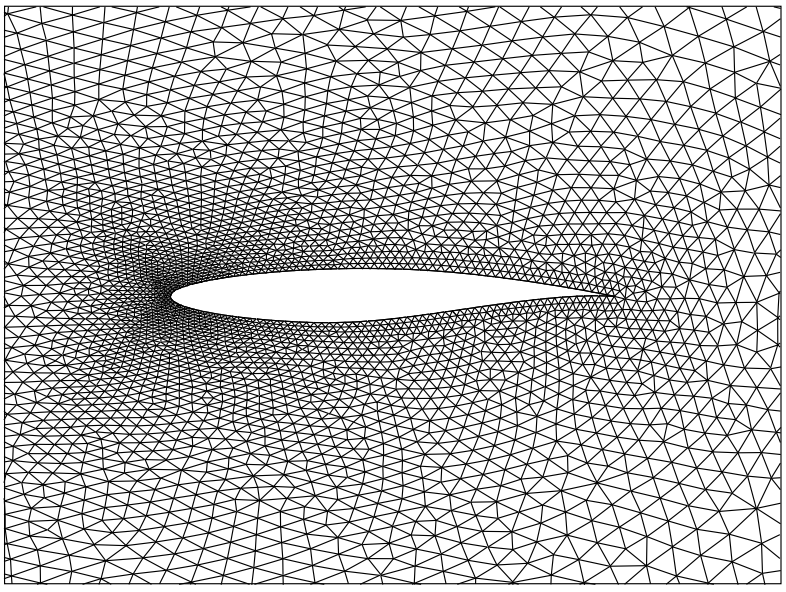


b) pressure contours

Figure 27: Grid and pressure contours for the supersonic bump case. The extended Van Albada limiter is used here.



a) Farfield.



b) Nearfield.

Figure 28: Computational grid used for the RAE 2822 airfoil case.

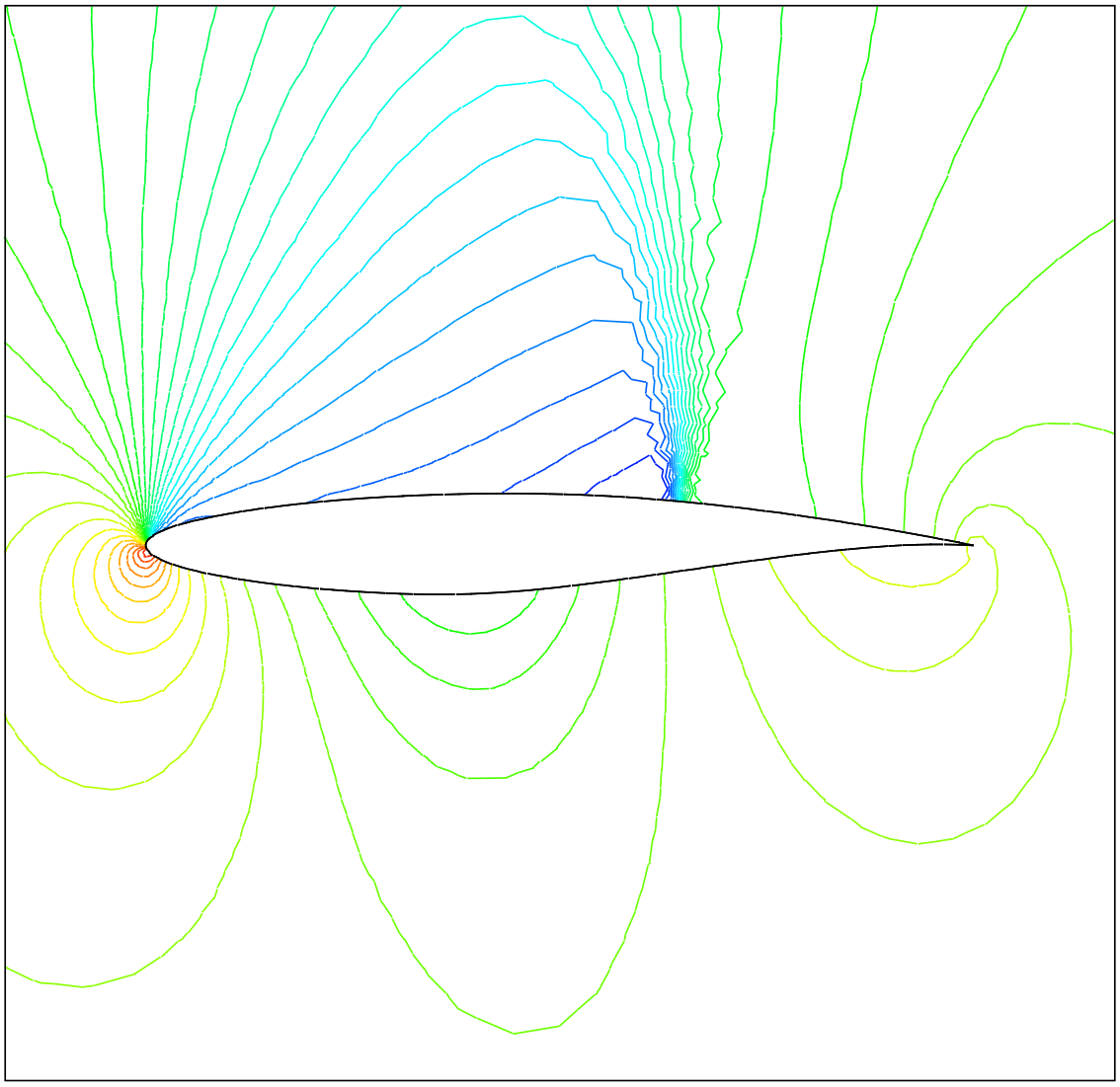


Figure 29: Computed pressure contours for transonic flow over an RAE 2822 airfoil using the extended Van Albada limiting described in the previous chapter and in Bruner & Walters (1996).

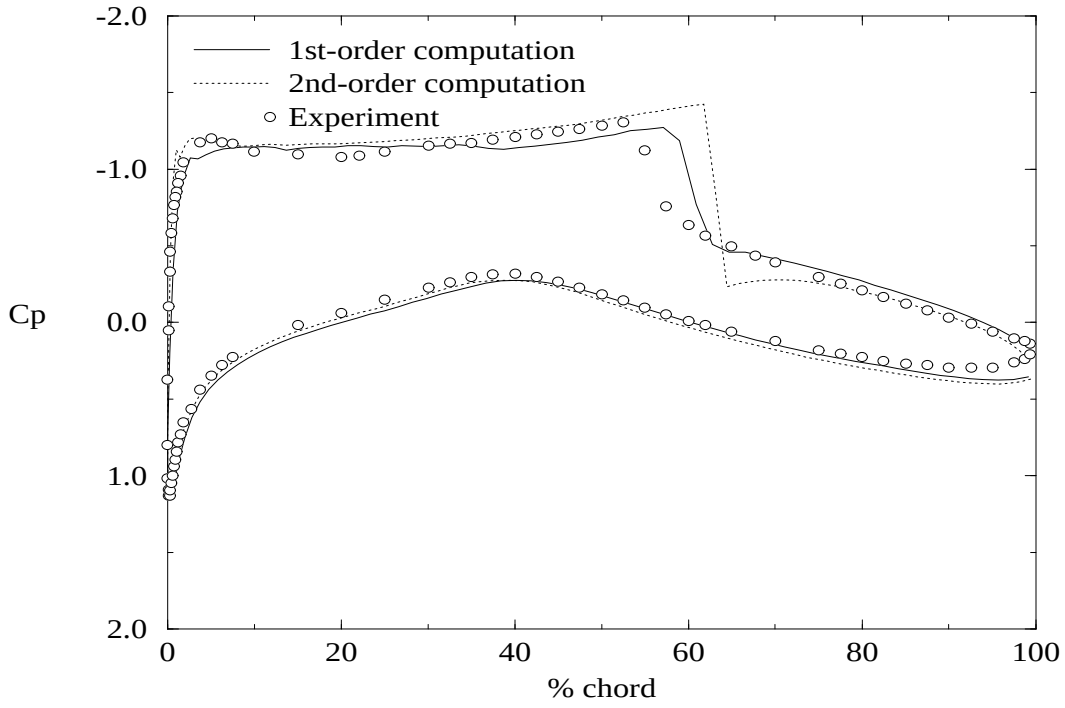


Figure 30: Comparison with experimental pressure coefficient data, RAE 2822 airfoil. The second-order solution is obtained using the extended Van Albada limiter described in the previous chapter and in Bruner & Walters (1996).

Three-dimensional Cases

Hummel Delta Wing

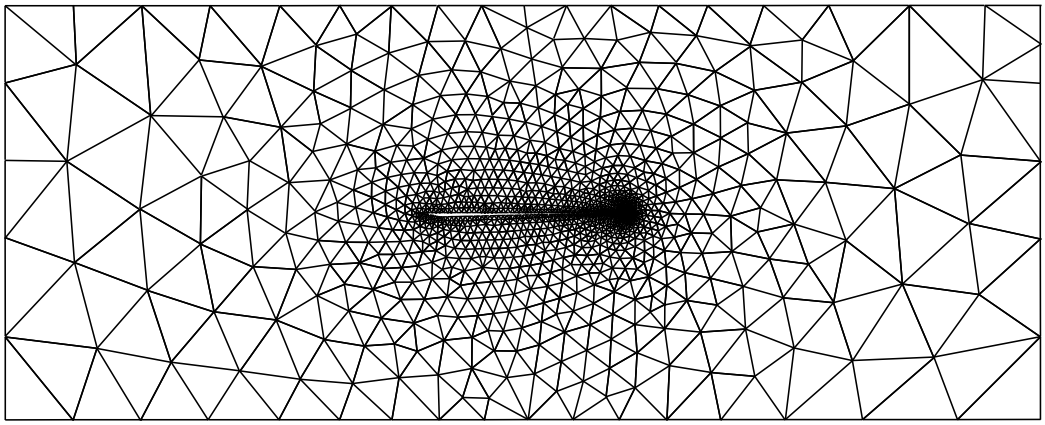
This subsonic flow was investigated by Hummel (1978). Mach 0.3 flow is modeled over a unit aspect ratio delta wing at an angle of attack of 20.5˚. Primary, secondary, and tertiary vortices are all present in the real flow; only the primary vortex is captured using the Euler equations. Characteristic boundary conditions were used for the entire outer boundary for this case.

The timing results were obtained on a fairly coarse grid (Figure 31), and the grid resolution does not permit a very good solution for the primary vortex behind the wing (see Figure 32). A solution obtained on a finer grid is shown in Figure 33, and the primary vortex is much more resolved.

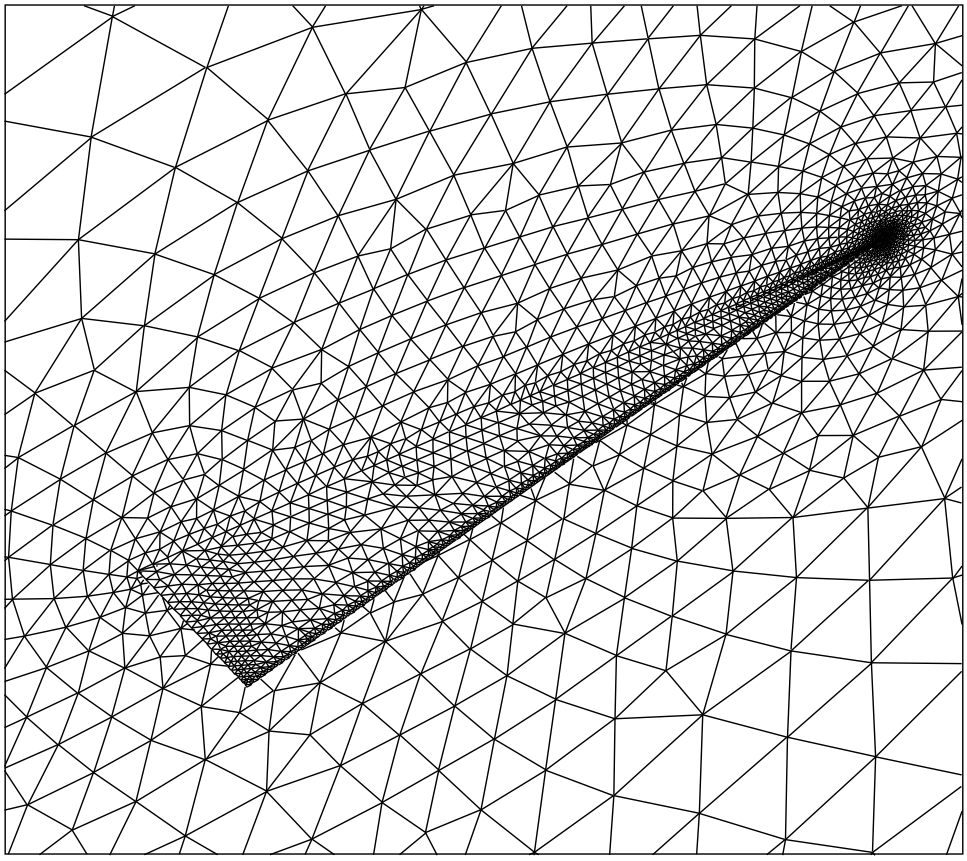
Analytic Forebody

Mach 1.7 flow at 0˚ angle of attack is modeled in this case. The grid is similar to the grid for this case supplied with the GASP v2.2 distribution but has twice the grid density in the body normal direction (AeroSoft 1992). This grid is composed solely of hexahedra and is shown in Figure 34; pressure contours on the body surface and exit and symmetry planes are shown in Figure 35. This solution was computed using PUE3D and second-order fluxes with the extended Van Albada limiting described earlier.

The structured-grid solution was obtained on the fine grid using GASP v2.2 with second-order fully-upwind fluxes and Van Albada limiting in each logical direction. Both codes used inflow boundary conditions that were fixed at the freestream values; the outflow boundary was first-order extrapolation. Figure 36 shows the experimental data and a second-order structured grid solution as well as both first- and second-order solutions computed with PUE3D using the extended Van Albada limiter. The differences in the solutions computed using the two codes may be attributed to the different techniques for computing and limiting extrapolated values at the faces: GASP uses logical coordinates while PUE3D is based on average gradients in each cell and the limiter extension described above.

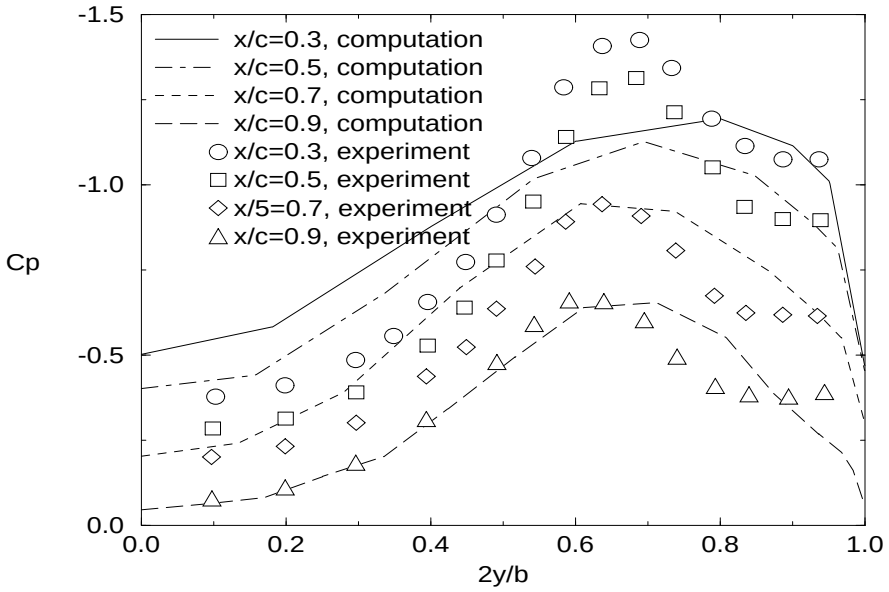


a) Farfield symmetry plane. Fluid enters the domain from the right.

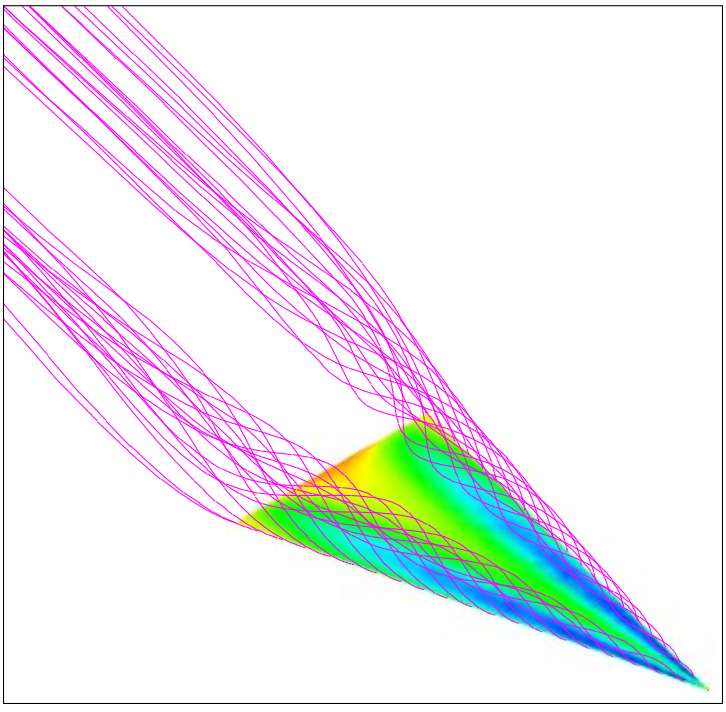


b) Surface grid and nearfield symmetry plane.

Figure 31: Computational grid used for the Hummel delta wing case.

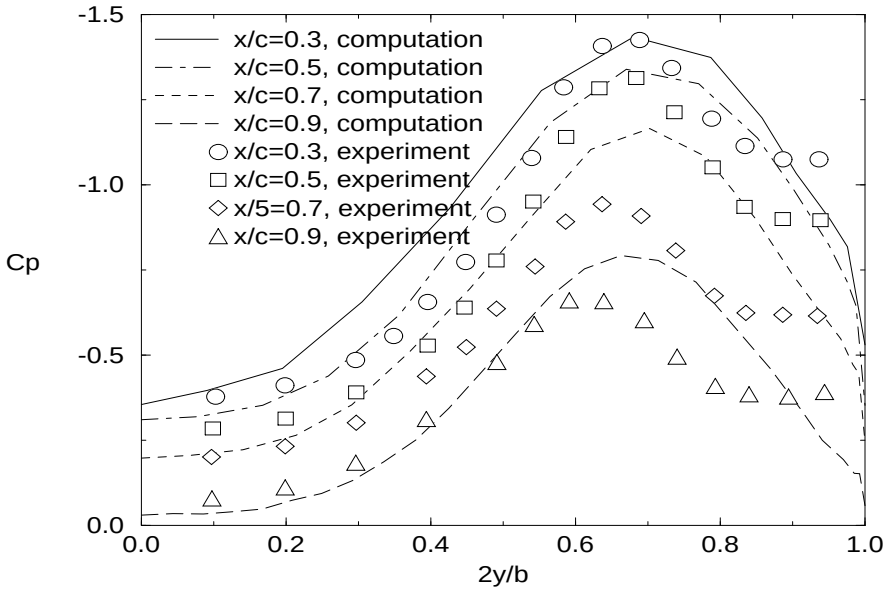


a) Pressure coefficient comparison with experiment.

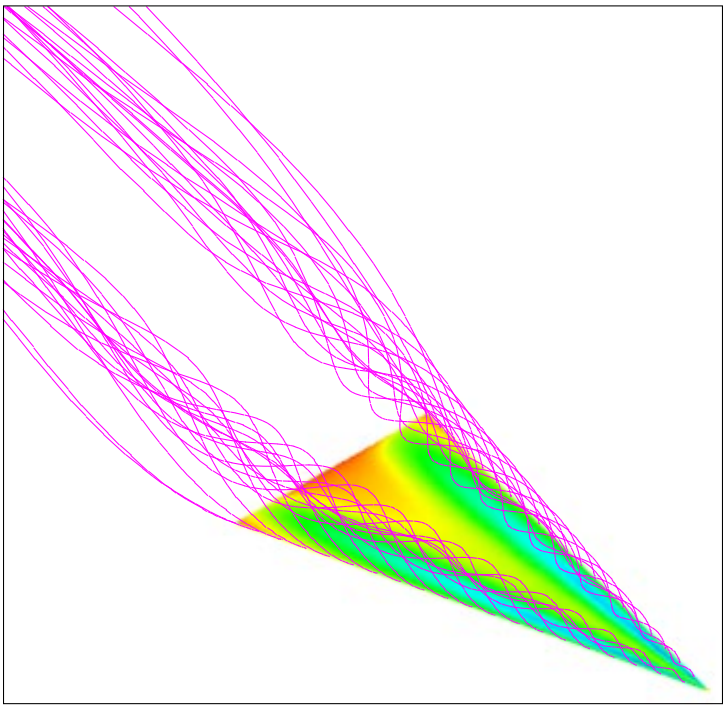


b) Particle traces and pressure contours.

Figure 32: Coarse-grid solution for the Hummel delta wing.

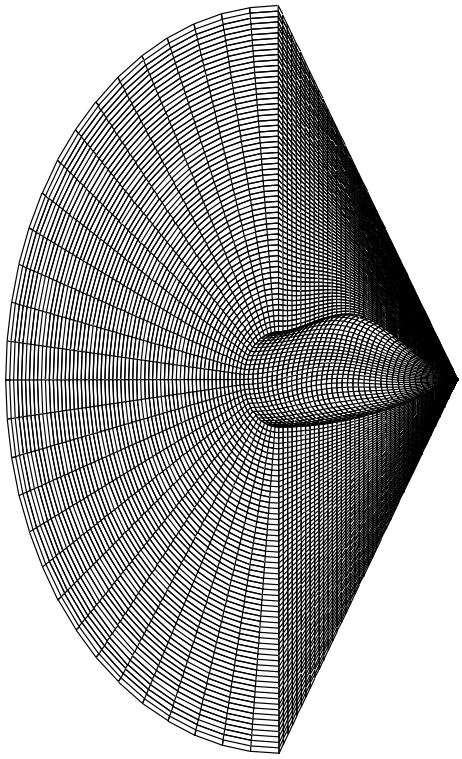


a) Pressure coefficient comparison with experiment.

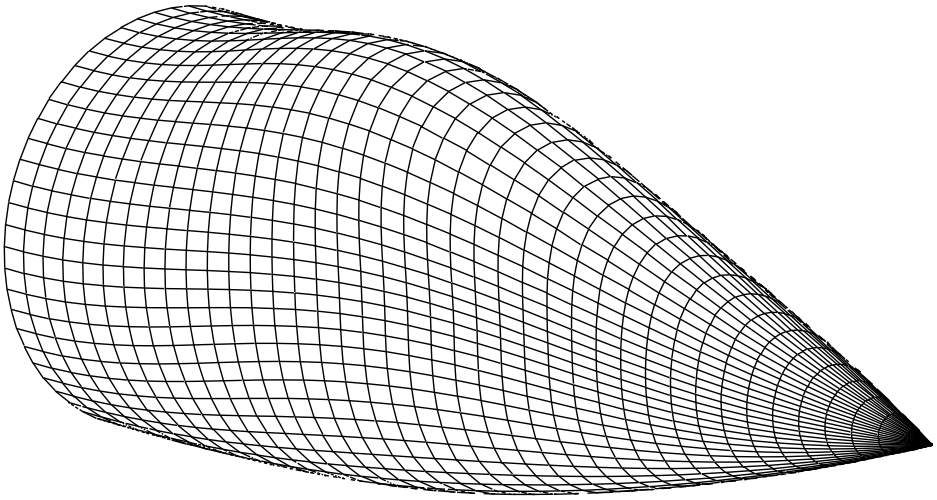


b) Particle traces and surface pressure.

Figure 33: Fine-grid solution for the Hummel delta wing.



a) Farfield and symmetry plane.



b) Surface grid.

Figure 34: Computational grid used for the analytic forebody case.

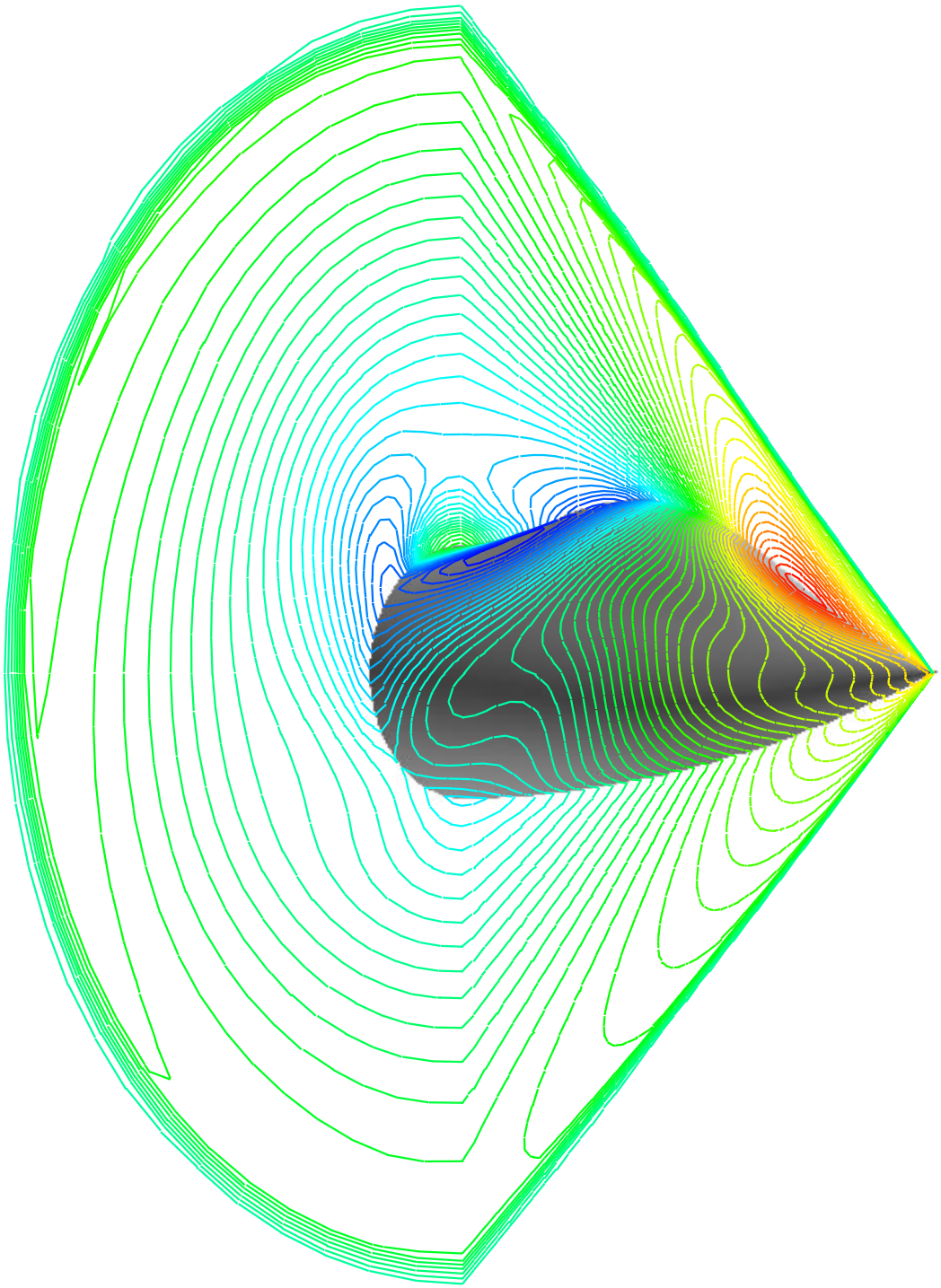


Figure 35: Pressure contours on the analytic forebody.

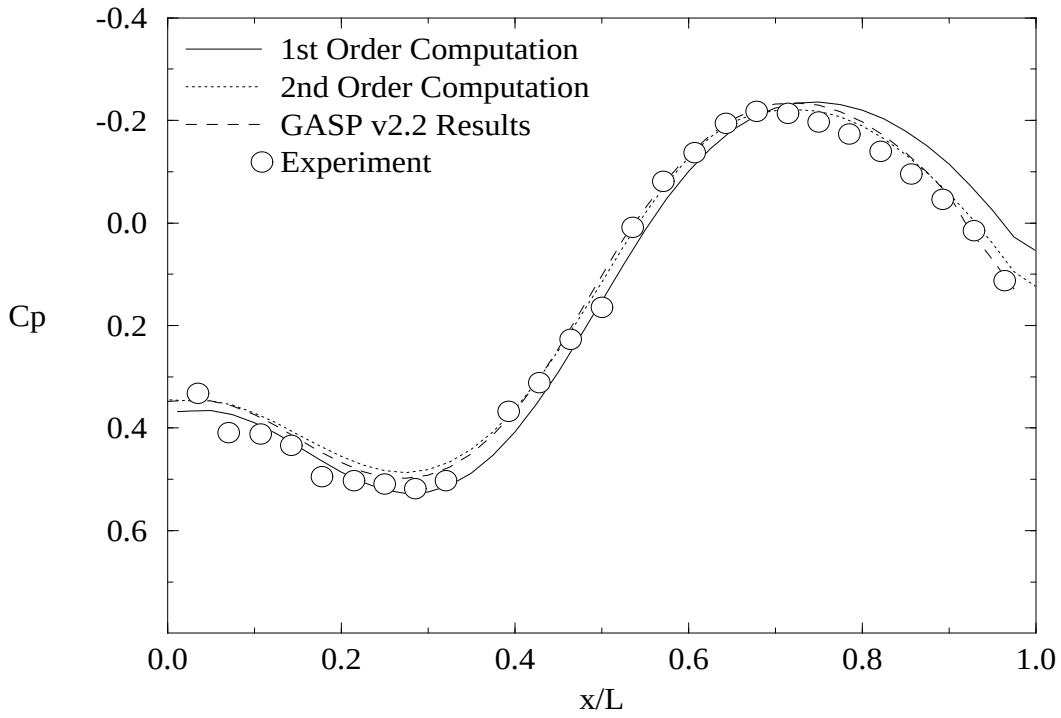
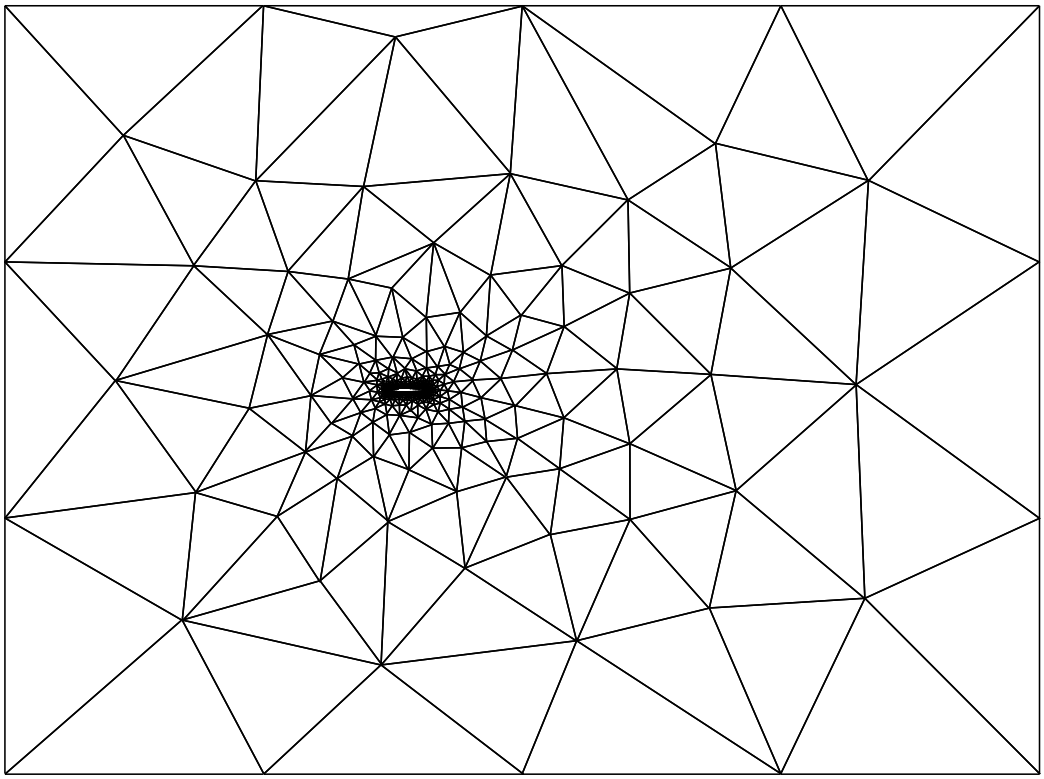


Figure 36: Pressure coefficient on the analytic forebody: comparison with experiment and structured code results.

ONERA M6 Wing

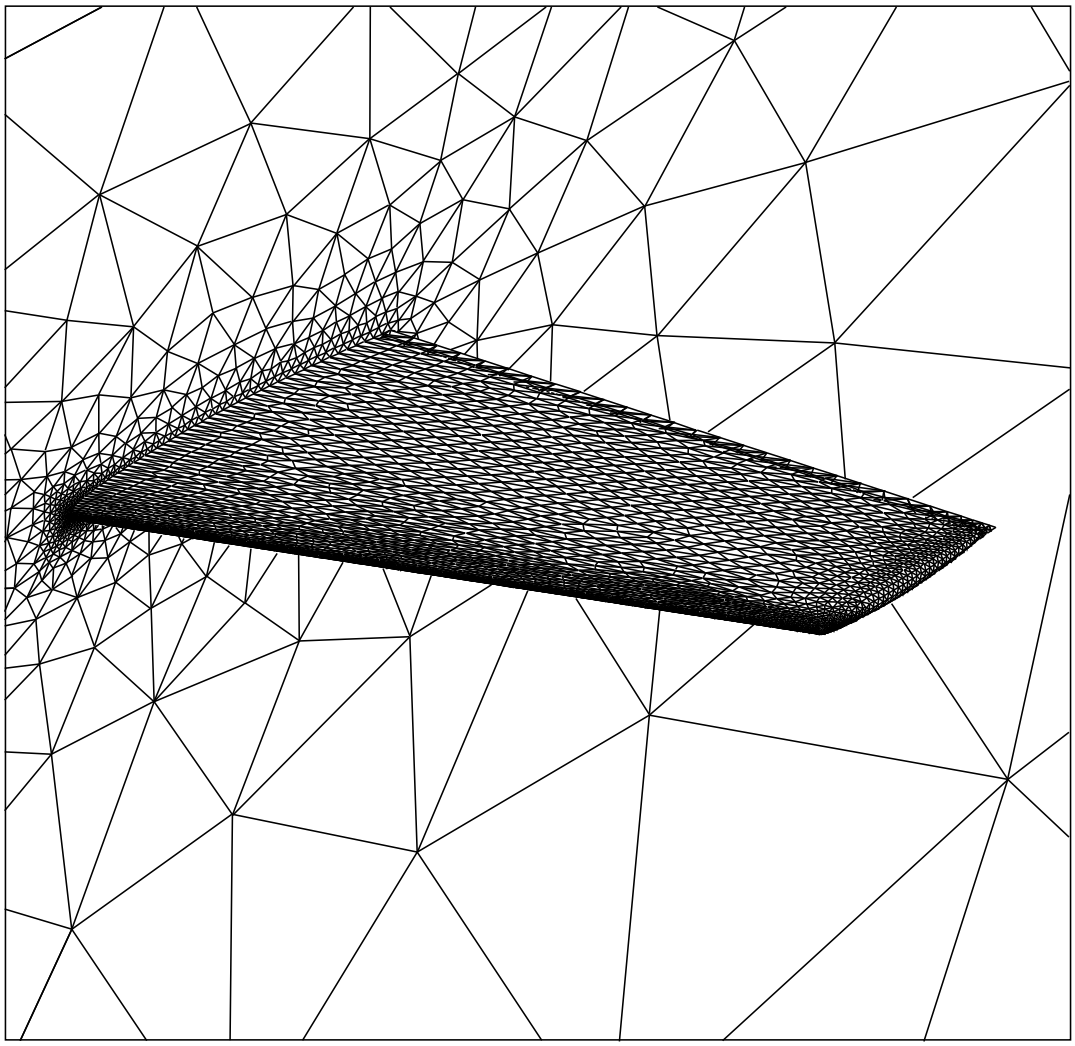
This ubiquitous test case is a good example of three-dimensional transonic flow. This case models Mach 0.84 flow over the ONERA M6 wing at 3.06˚ angle of attack, forming two shocks on the upper surface of the wing which coalesce at about 85% span.

The computational grid used for this case is shown in Figure 37. The computed second-order solution is plotted in Figure 38, while comparisons with experimental data are given in Figure 39 (Schmitt & Charpin 1984). Characteristic boundary conditions were applied at the inflow and outflow boundaries. A symmetry boundary condition was applied at both the symmetry plane and at the plane opposite the symmetry plane. Convergence could not be achieved with characteristic boundary conditions on this plane; the likely cause is erratic switching of the sign of the normal velocity at some of the faces on this plane when characteristic boundary conditions are used.



a) Farfield symmetry plane.

Figure 37: Computational grid for the ONERA M6 wing case.



b) Surface grid and nearfield symmetry plane.

Figure 37 (continued): Computational grid for the ONERA M6 wing case.

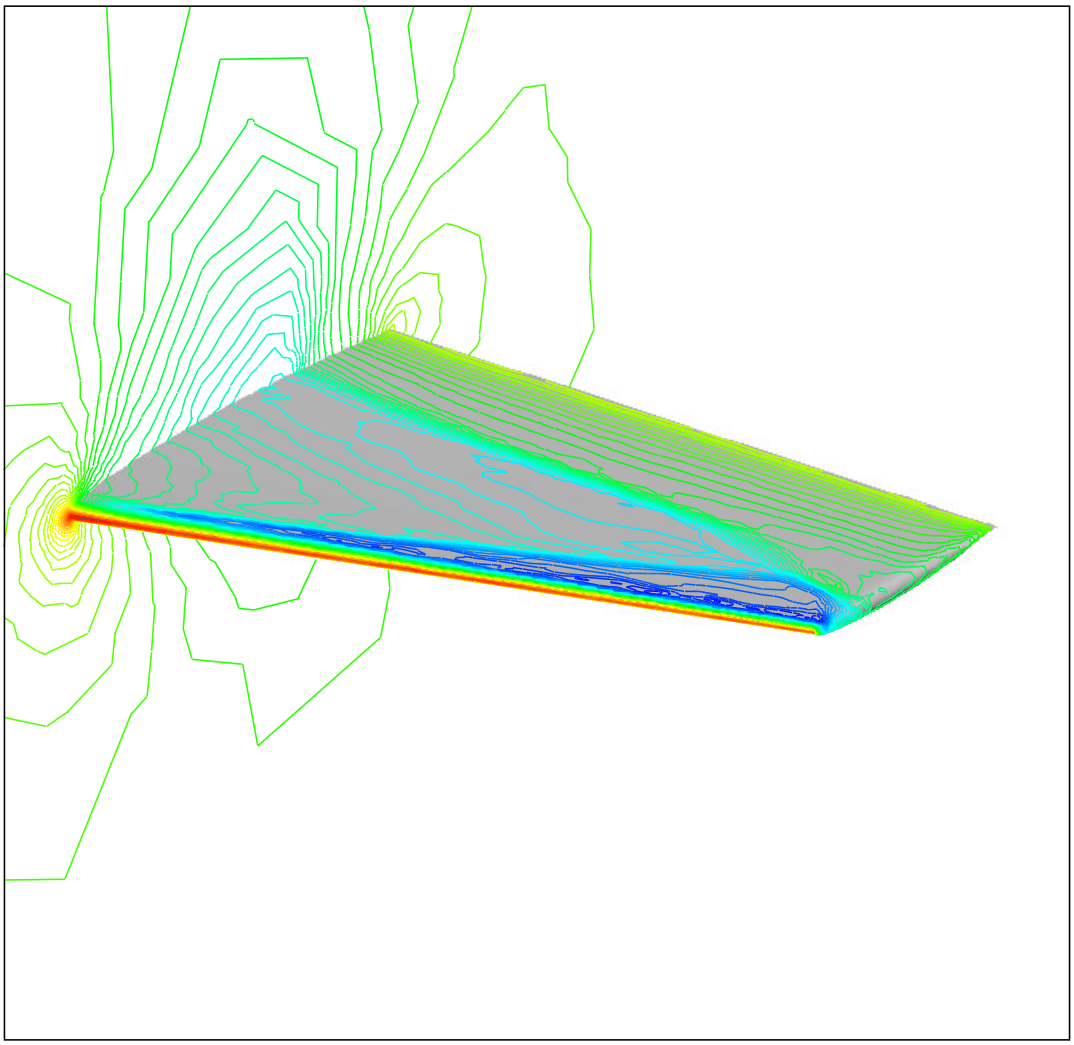
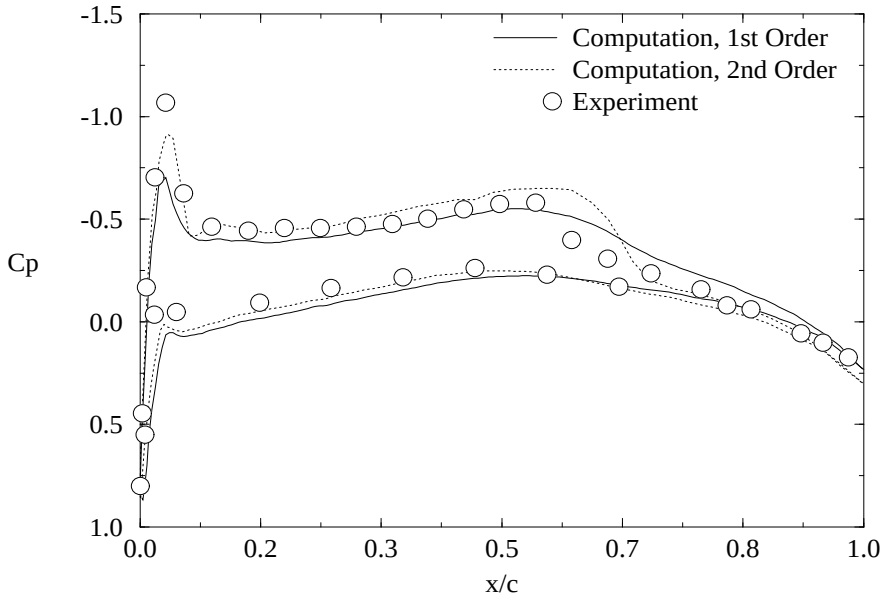
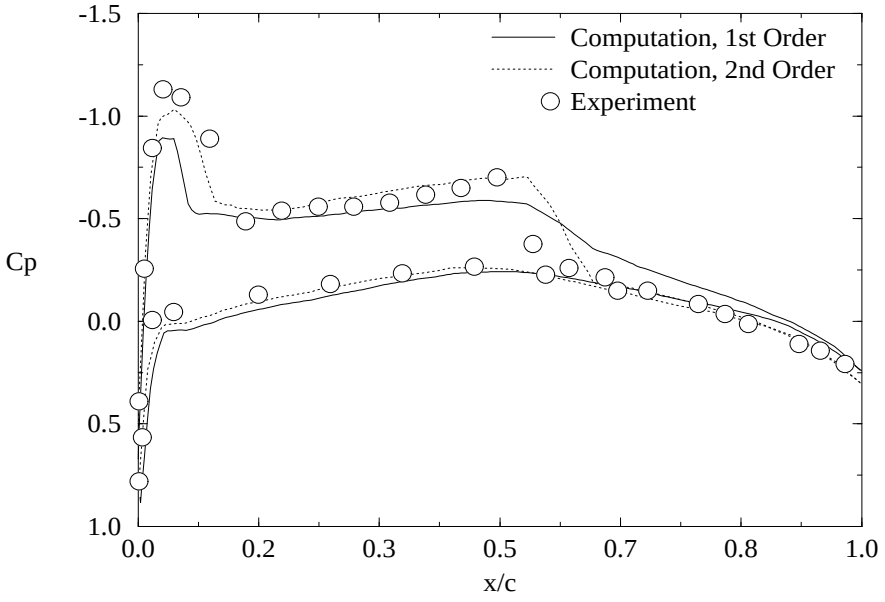


Figure 38: Computed second order pressure contours for the ONERA M6 wing.

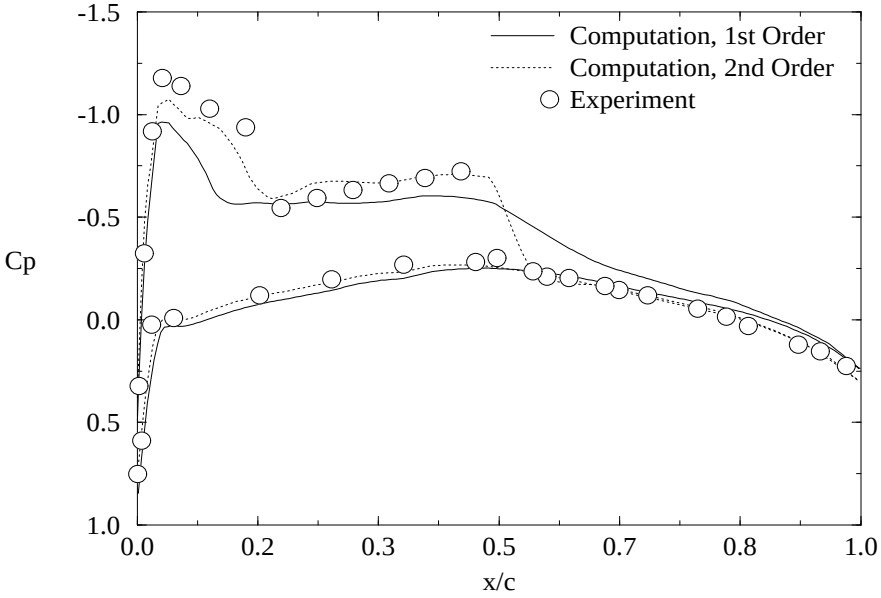


a) Results at 20% semispan.

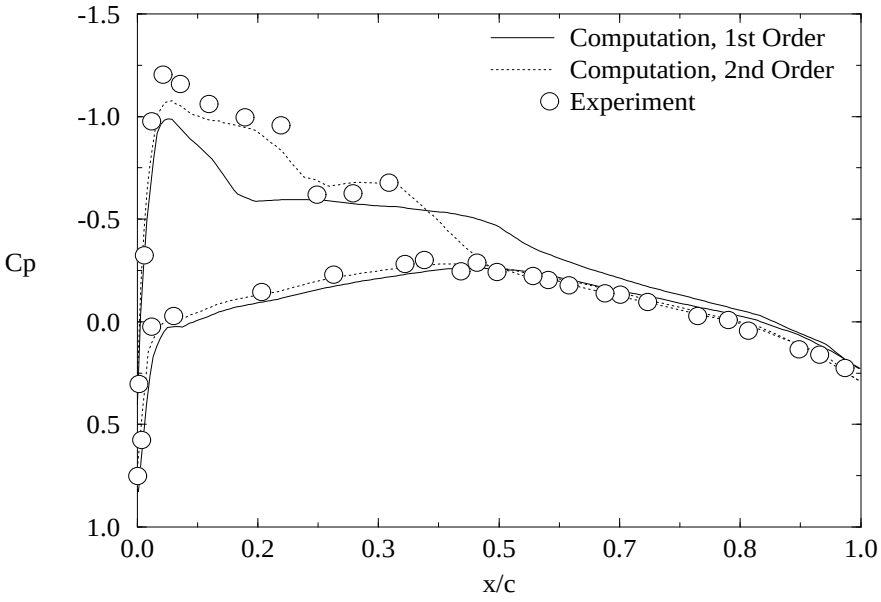


b) Results at 44% semispan.

Figure 39: Pressure coefficient on the ONERA M6 wing.

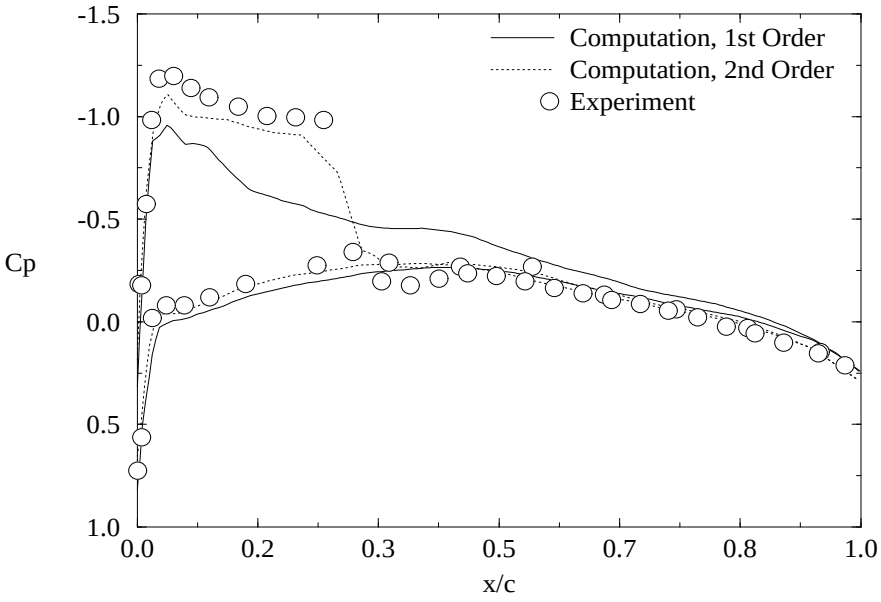


c) Results at 65% semispan.

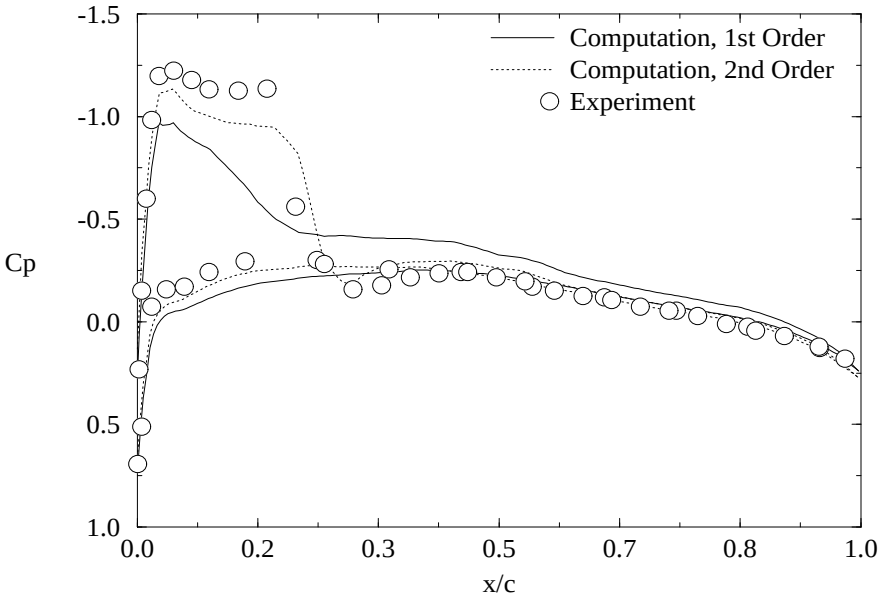


d) Results at 80% semispan.

Figure 39 (continued): Pressure coefficient on the ONERA M6 wing.

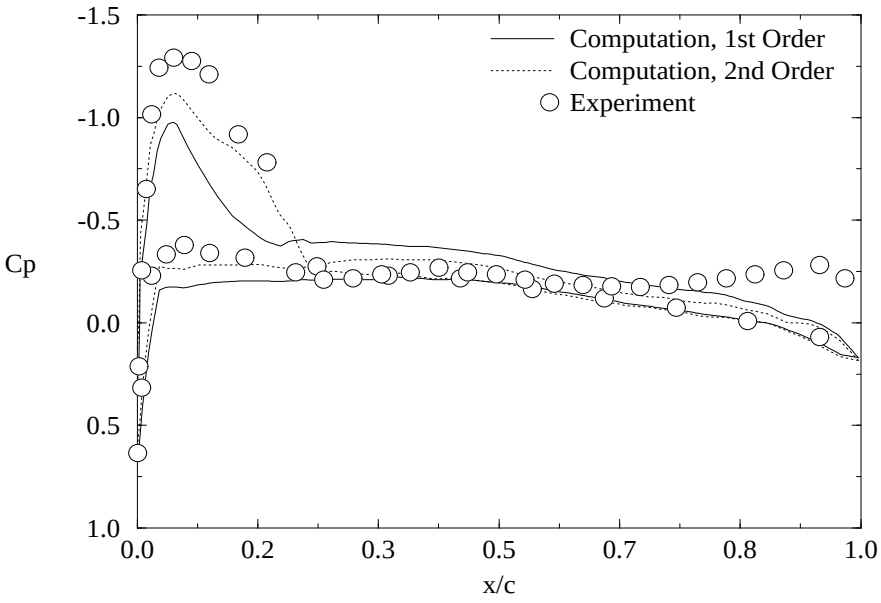


e) Results at 90% semispan.



f) Results at 95% semispan.

Figure 39 (continued): Pressure coefficient on the ONERA M6 wing.



g) Results at 99% semispan.

Figure 39 (continued): Pressure coefficient on the ONERA M6 wing.

Results and Discussion

Due to the large number of timing runs performed in this effort, all of the first-order timing results are presented graphically in Appendix A and the second-order results in Appendix B. For convenience, the figures discussed in this chapter are reproduced below.

Two kinds of runs were performed on the MHPCC SP-2. One kind uses exclusively thin nodes, while the other uses exclusively wide nodes. The performance of most unstructured CFD codes is limited by memory accesses rather than floating point operations, and PUE3D is no exception: even though the processors in the wide and thin nodes are identical, the differences in cache size and bus width are enough to have an impact on performance (Figure 40). Therefore, the thin-node and wide-node runs may be considered to have been performed on different machines for the sake of this investigation.

Note that not every algorithm was run for every case on every "machine" — this was considered a waste of computing time. However, most of the algorithms were run on the RAE 2822 case on every machine; all* of the algorithms were run on the Paragon for this case.

Three algorithms were run for each case on each machine: Block Jacobi using a fixed number of inner iterations (this is one of the most popular means used today for converging the inner problem on unstructured grids), Block Jacobi using a specified tolerance for the inner problem (this often has better performance than the previous variant), and Symmetric Gauss-Seidel (GS) with a fixed tolerance (this algorithm had the best overall performance in the preliminary timing runs on the supersonic bump and RAE 2822 airfoil cases).

Generally, the inner problem was converged to a tolerance of 0.1 or for 10 inner iterations. The exceptions to this rule are the second-order Hummel delta wing (which diverged with 10 Jacobi iterations), the analytic forebody (which required the inner problem to be converged to 0.01), and the fixed-iteration SGS algorithm for the first-order RAE 2822 case on the Paragon and the SP-2 and the supersonic bump on the Paragon,

*Only one of the unidirectional GS algorithms is presented here; both Forward and Reverse GS algorithms have essentially identical performance for the RAE 2822 case since this problem has no preferential direction.

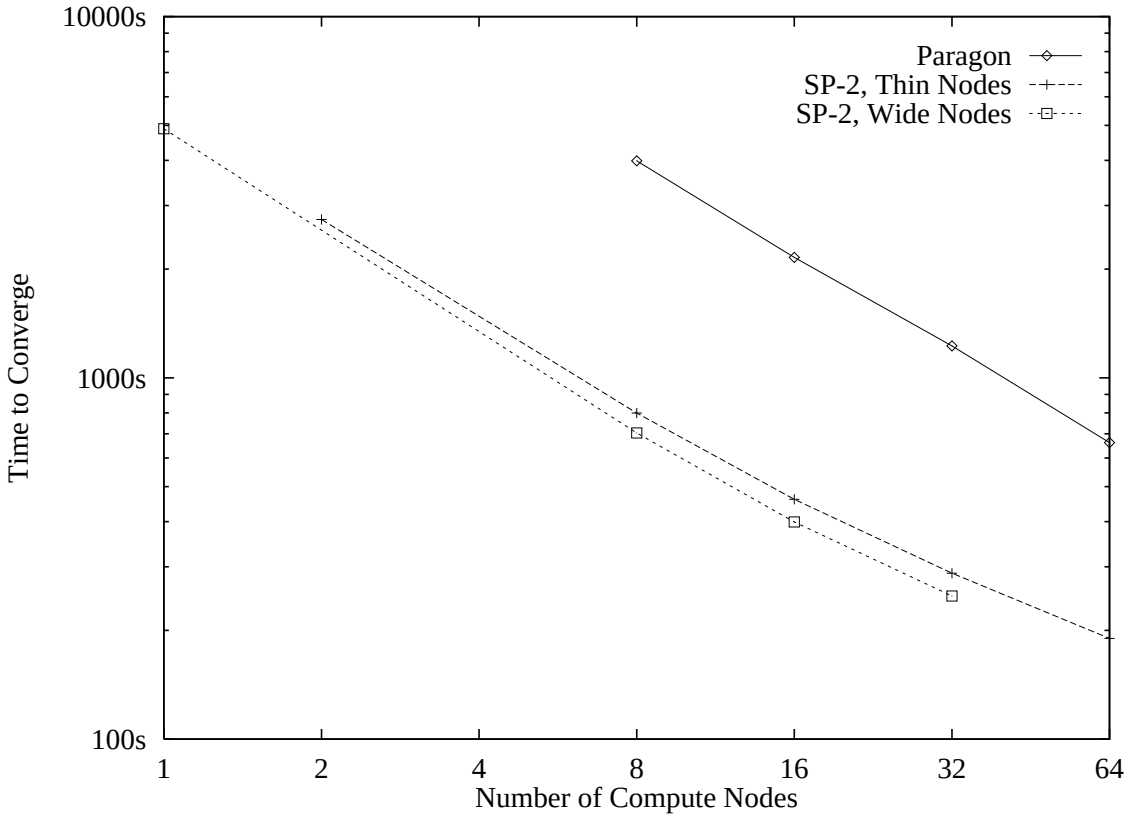


Figure 40: Convergence time comparison across machines for the first order Hummel delta wing case.

which uses 5 inner iterations. A look back at the flowchart for the SGS algorithm shown in Figure 9 on page 35 will remind the reader that an inner iteration for the SGS algorithm involves one forward and one backward sweep, so 5 iterations of SGS is comparable to 10 iterations of unidirectional GS.

Speedup

The parallel speedup is defined as

$$\sigma \equiv \frac{t_B}{t_N} \tag{34}$$

Normally, the baseline number of nodes in Equation (34) is one. However, for several cases (especially the larger grids on the Paragon), the problem is too large to fit in physical memory on a single node, and the baseline time is no longer based on serial time to converge. At least for the SP-2 runs, where the baseline is relatively small (1, 2, or 4), the results are still readily extendible to large numbers of nodes, since the small number of nodes used in the baseline is not large enough for significant parallel effects to occur.

Amdahl's Law (Amdahl 1967) gives the following expression for speedup, assuming that t_B in Equation (34) is the time on one compute node:

$$\sigma = \frac{N}{f_P + N(1 - f_P)} \tag{35}$$

Figure 41 shows the theoretical speedup for various parallel fractions using Amdahl's Law. Note that Figure 41 applies Amdahl's Law assuming that the same sequence of operations is performed in each run and that the parallel fraction does not change. For Block Jacobi with a fixed number of inner iterations as well as for the explicit schemes, the same sequence of operations is performed; this cannot be said of the other implicit schemes. For all of the schemes, the communications cost (as reflected in the number of cells and faces sent to neighbors) goes up nonlinearly with the number of compute nodes. Figure 42 reflects the

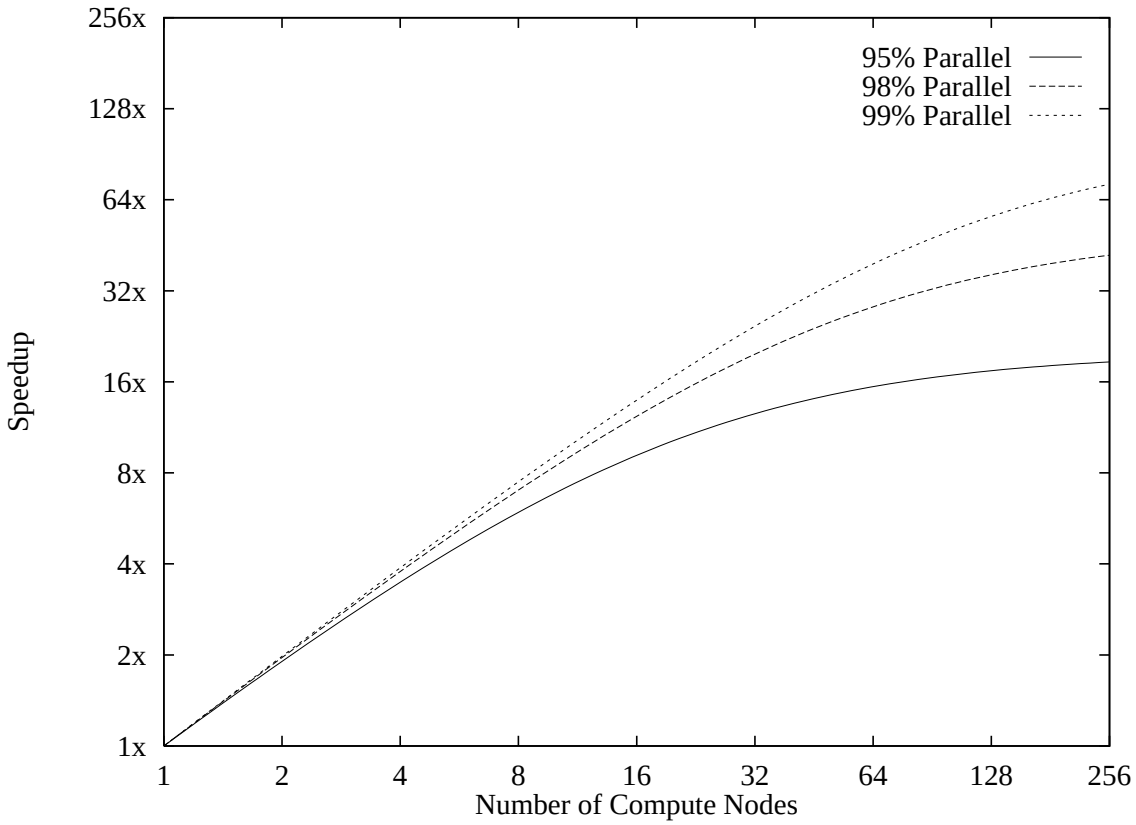


Figure 41: Speedup from Amdahl's Law.

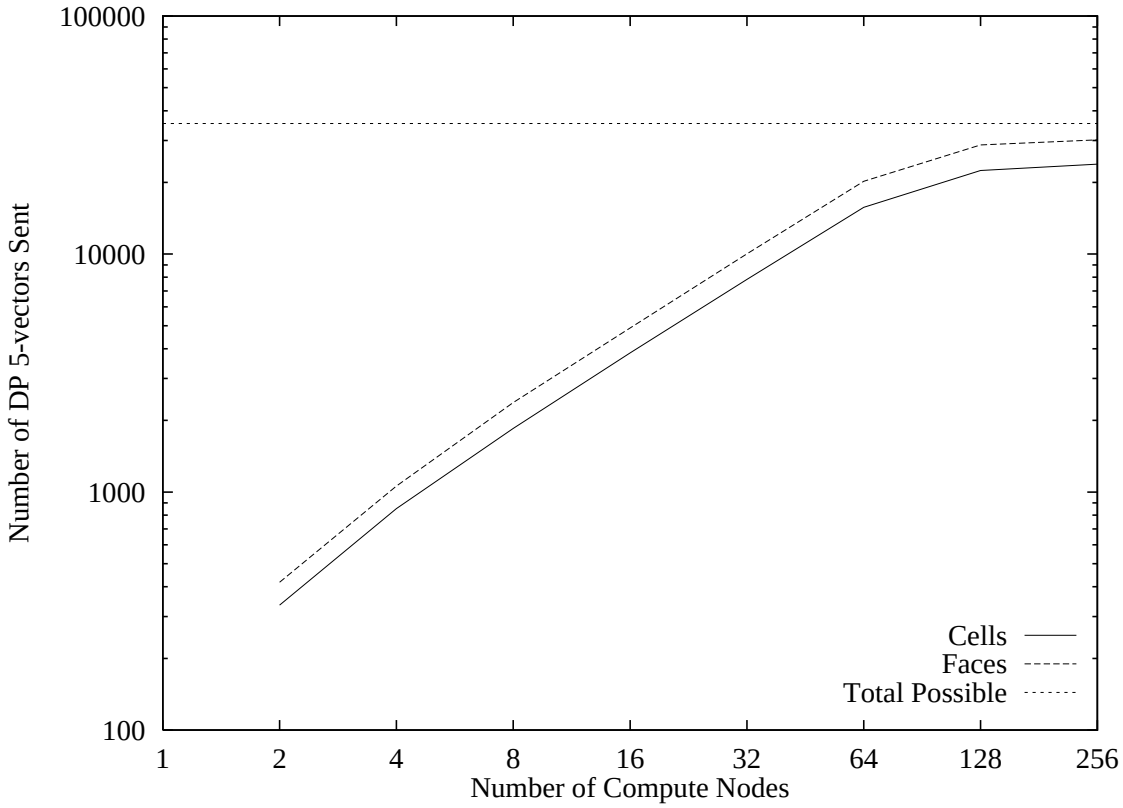


Figure 42: Number of double-precision vectors sent to a neighbor as a function of the number of compute nodes for the RAE 2822 airfoil grid. The total number of vectors it is possible to send is equal to twice the number of interior faces in the grid.

total number of double-precision vectors, each 40 bytes long, sent for the RAE 2822 airfoil grid.* The explicit algorithms send the equivalent of one set of cells and one set of faces every stage, while the implicit schemes send the equivalent of one set of cells per inner iteration and one set of faces every outer iteration. Note that the communications burden as measured by this parameter increases linearly until the domain is broken into about 64 partitions. By that time, the communications load for the problem has just about reached its maximum, so we might expect most of the penalty for increasing communications costs to occur before one reaches 128 nodes. This effect can be seen in Figure 43, and to a lesser extent in Figure 44.

The first-order analytic forebody speedup results presented in Figure 45 (Paragon) and Figure 46 (SP-2) bear some discussion. The inner problem using SGS is not converged sufficiently for this case, leading to the overall poor performance of this algorithm on this problem using 64 nodes. The SGS algorithm exits the inner problem too early on 64 nodes (see Figure 47), and the outer problem hence takes 142 iterations (vs. 43 on 32 nodes) to converge. Bear in mind that operations performed are not exactly the same on different numbers of compute nodes, and that the GS algorithms are most affected by adding partitions, since all shared quantities are treated at the end of each inner iteration (similar to Jacobi). The corresponding second-order results in Figure 48 (Paragon) and Figure 49 (SP-2) do not display the same behavior. It was necessary for the second-order runs to converge the inner problem to a tolerance of 0.01 rather than 0.1 as used in all other cases; it is therefore believed that the first-order case is marginal for convergence of the inner problem and that the SGS algorithm simply breaks down before the others.

In each of the plots presented here and in the appendices, the axes have been scaled so that ideal (i.e., linear) speedup is a diagonal line from the origin to the upper right-hand corner of the graph.

*This parameter is not strictly a reflection of the total communications cost, however, because some messages may occur simultaneously.

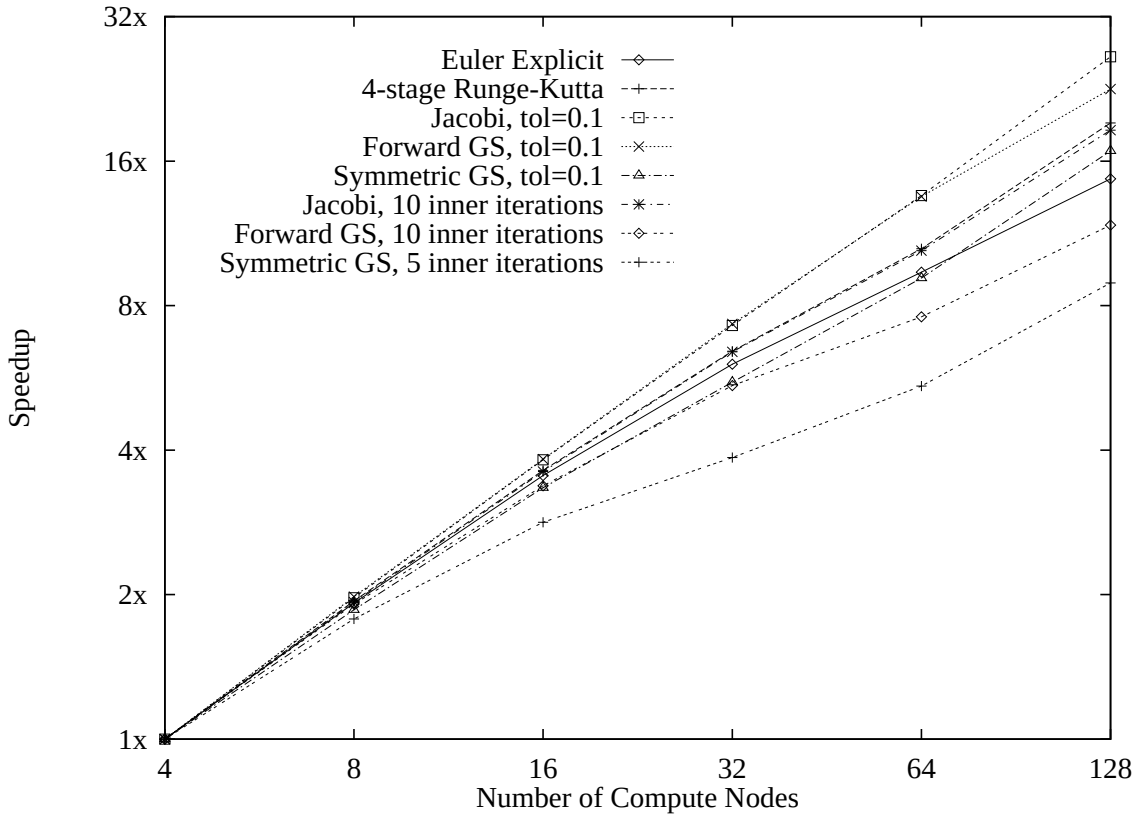


Figure 43: Speedup for the first-order RAE 2822 airfoil on the Intel Paragon.

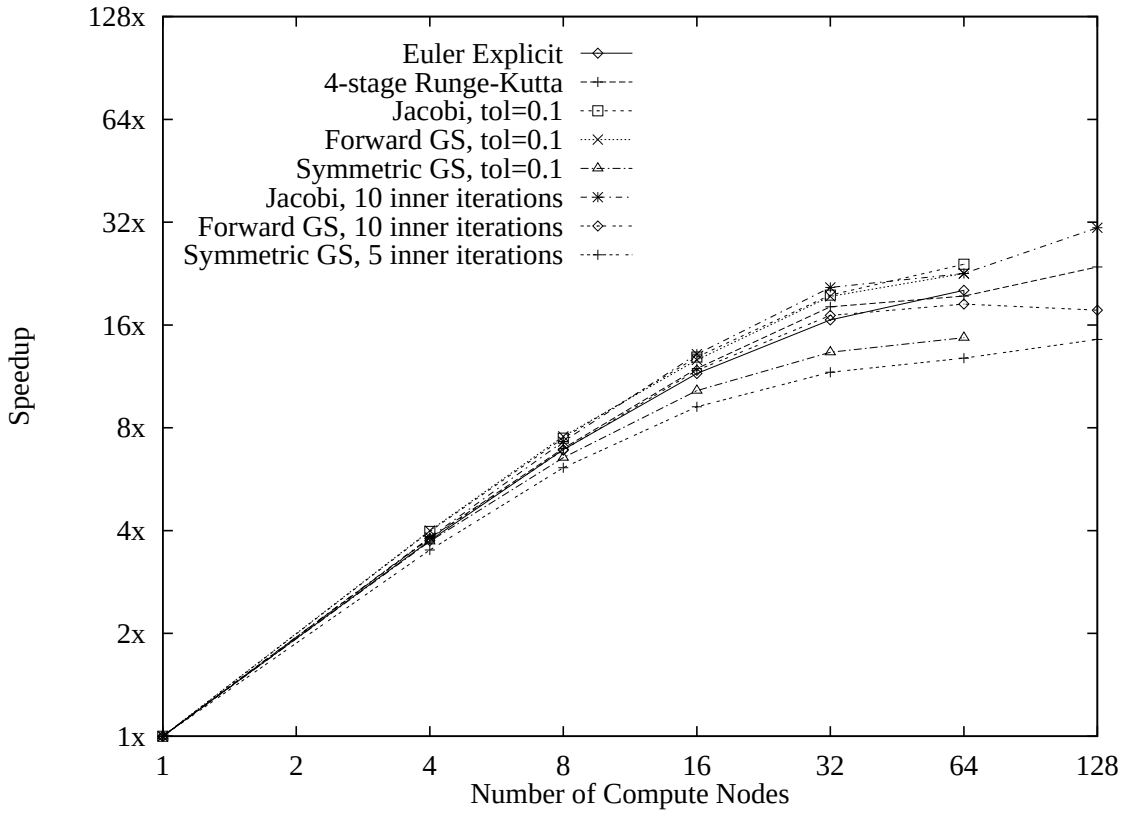


Figure 44: Speedup for the first-order RAE 2822 airfoil using thin nodes on the IBM SP-2.

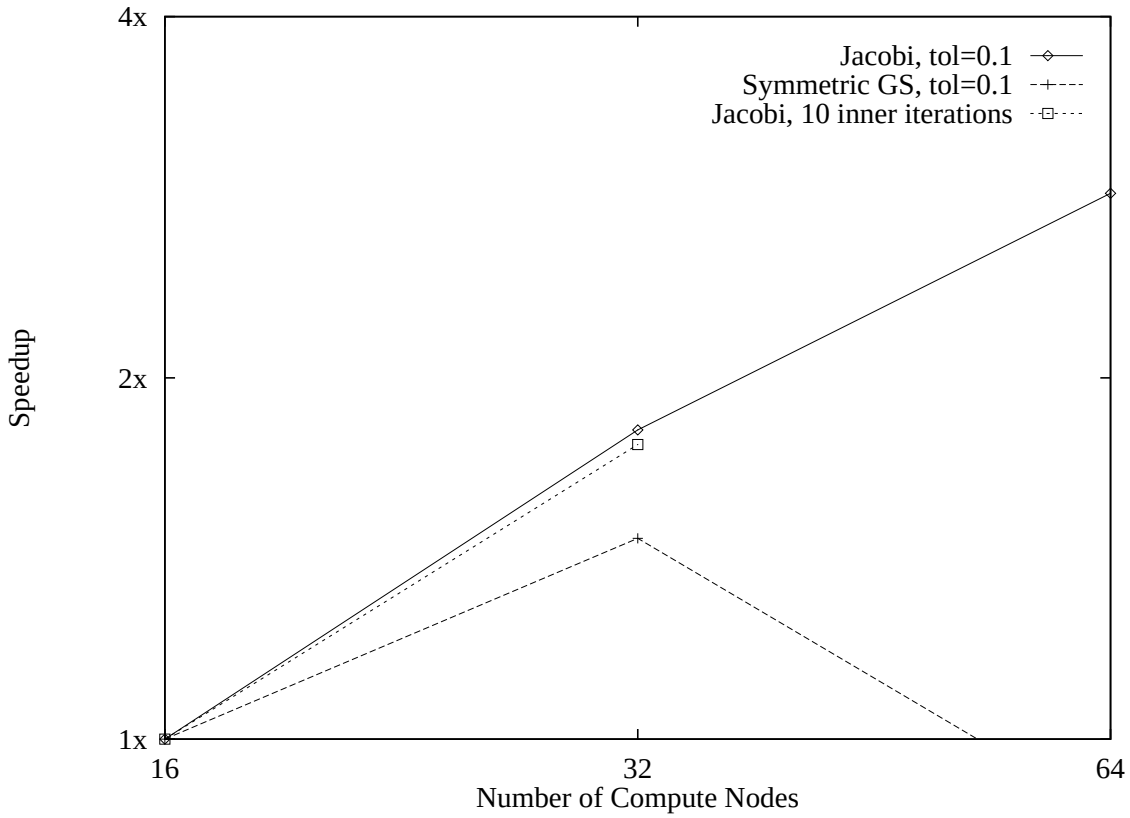


Figure 45: Speedup for the first-order analytic forebody on the Intel Paragon.

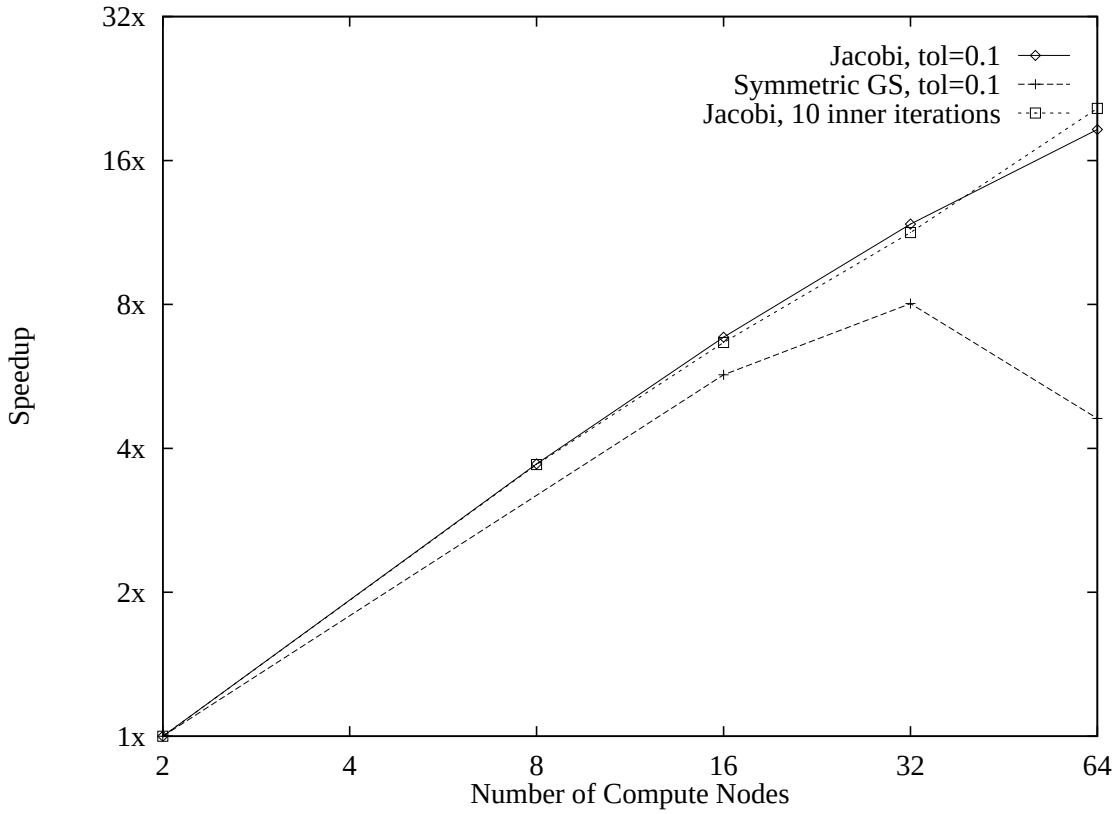


Figure 46: Speedup for the first-order analytic forebody using thin nodes on the IBM SP-2.

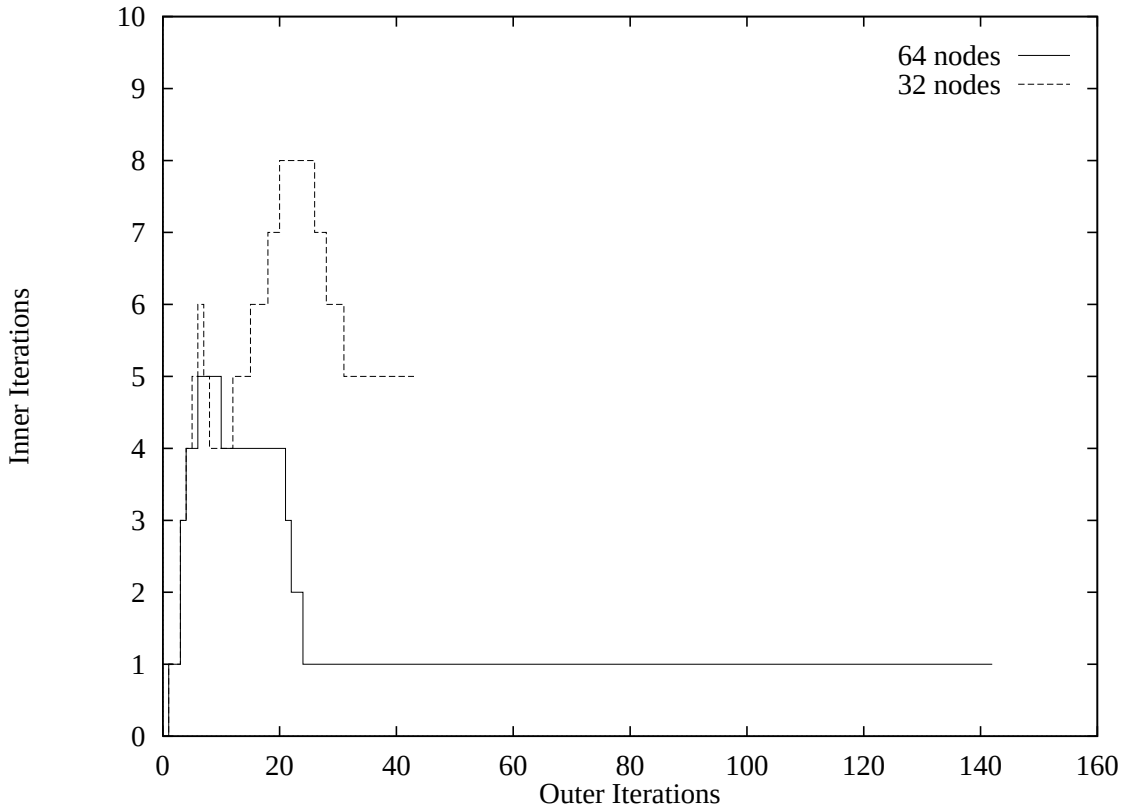


Figure 47: Number of iterations required by the SGS algorithm to converge the inner problem for the first-order analytic forebody case.

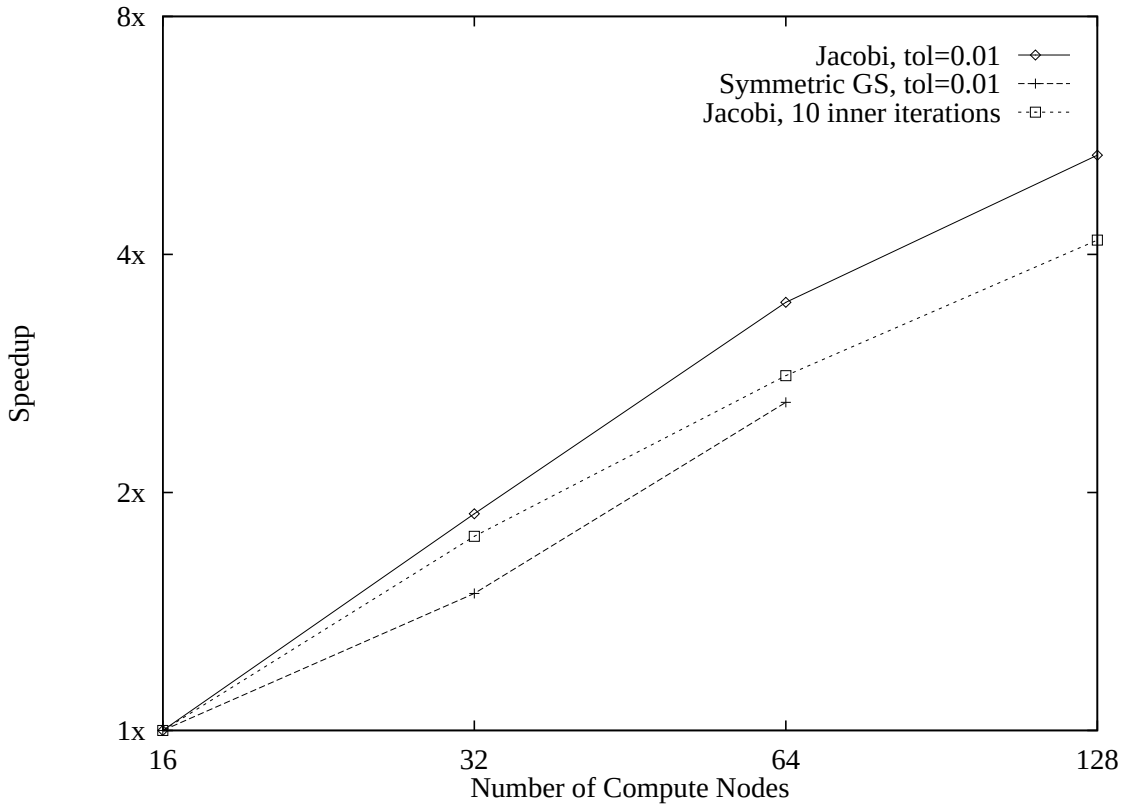


Figure 48: Speedup for the second-order analytic forebody on the Intel Paragon.

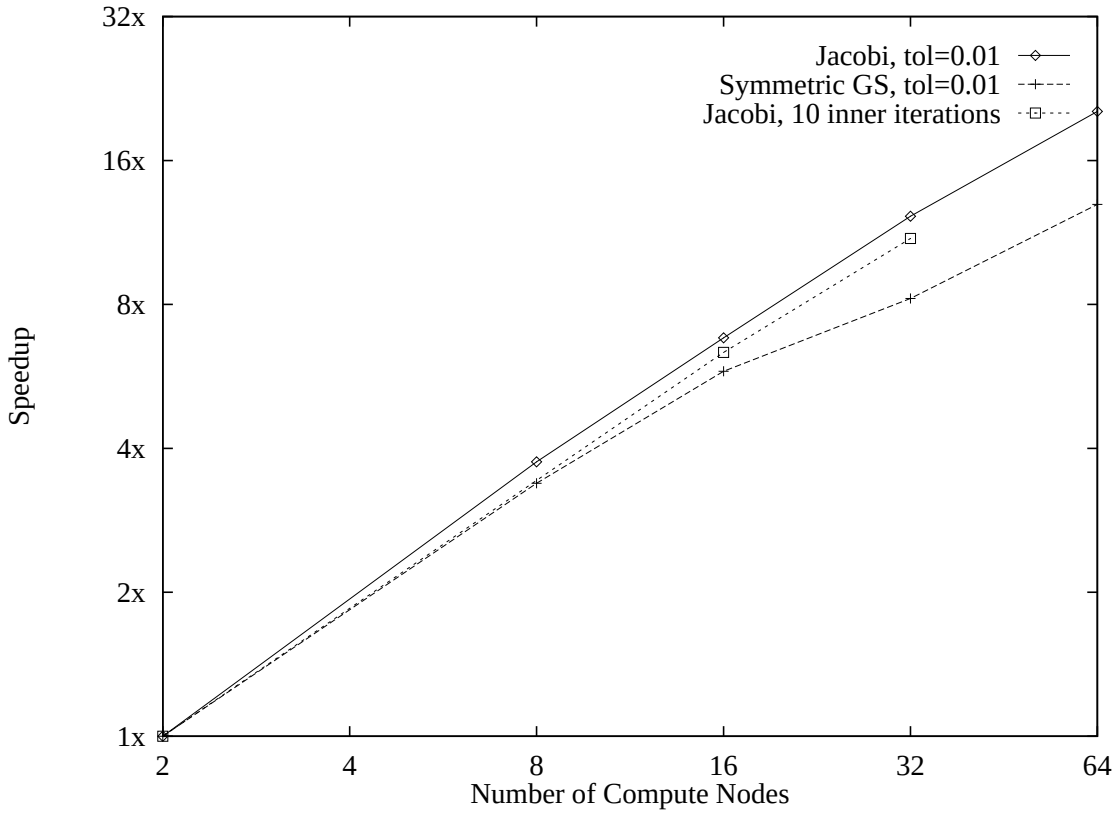


Figure 49: Speedup for the second-order analytic forebody using thin nodes on the IBM SP-2.

Parallel Efficiency

Parallel efficiency is defined as the ratio of the parallel speedup actually obtained to ideal (linear) speedup:

$$\eta_P \equiv \frac{B\sigma}{N}, \tag{36}$$

where B is the baseline number of nodes.

We can take Amdahl's Law in Equation (35) and combine it with Equation (36) to obtain the following expression for the efficiency:

$$\eta_P = \frac{1}{N + f_P - Nf_P} \tag{37}$$

The expected efficiency according to Equation (37) is presented in Figure 50, and we generally see shapes similar to one of the curves in Figure 50 for the actual efficiency. Since the communications costs go up roughly as in Figure 42, the parallel fraction changes with the number of compute nodes for every case, so it becomes meaningless to discuss parallel fractions for any of the cases considered, i.e., the cases would lie on a different curve for each number of compute nodes.

The parallel efficiency results for the first-order supersonic bump problem in Figure 51 are very misleading, and highlights the fallacy of using parallel efficiency as the sole criterion for evaluating an algorithm. The FGS algorithm has parallel efficiency greater than 100%! Could we be getting some benefit from splitting up the domain? A look at the convergence time for this problem (see Figure 52) shows that FGS is about in the middle of the pack as far as convergence time is concerned. The reason for the seemingly miraculous efficiency is that FGS goes *against* the flow direction for this problem. Since the GS algorithms share the evolving solution to the inner matrix problem at the end of each inner iteration, these terms behave like Block Jacobi terms in the GS problem. Thus information is communicated in the flow direction faster through these Jacobi-like terms than through

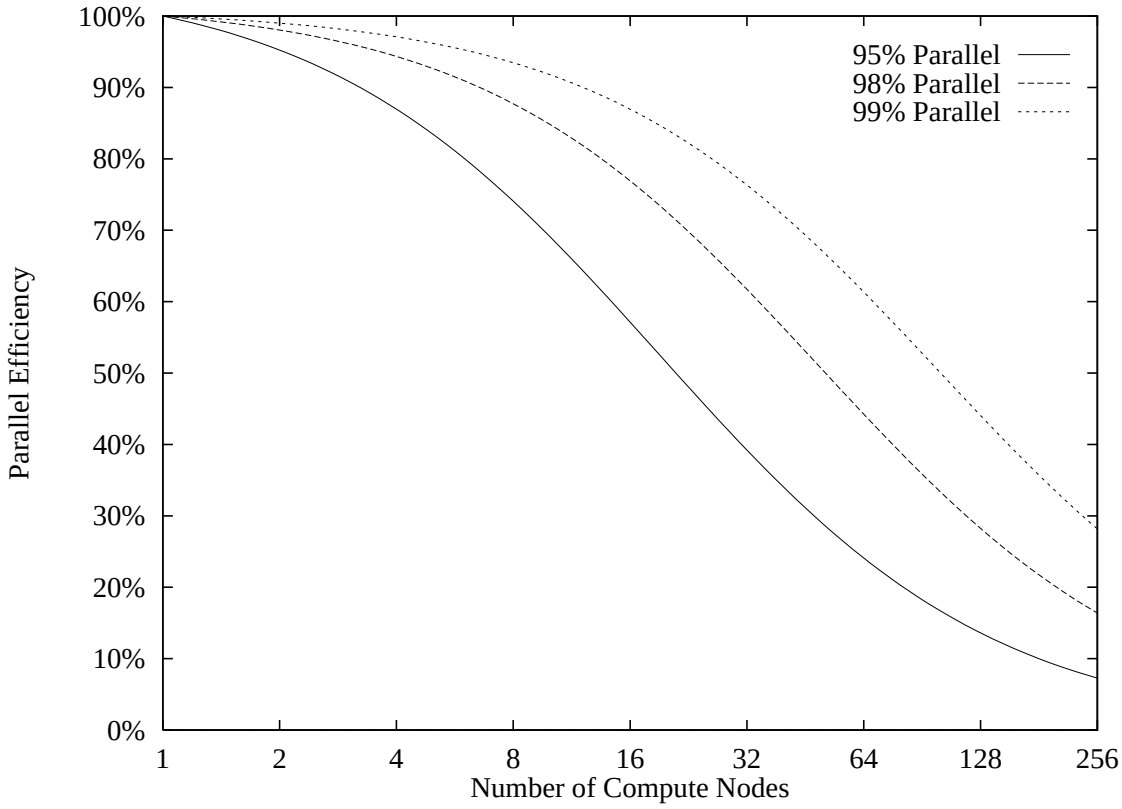


Figure 50: Parallel efficiency from Amdahl's Law.

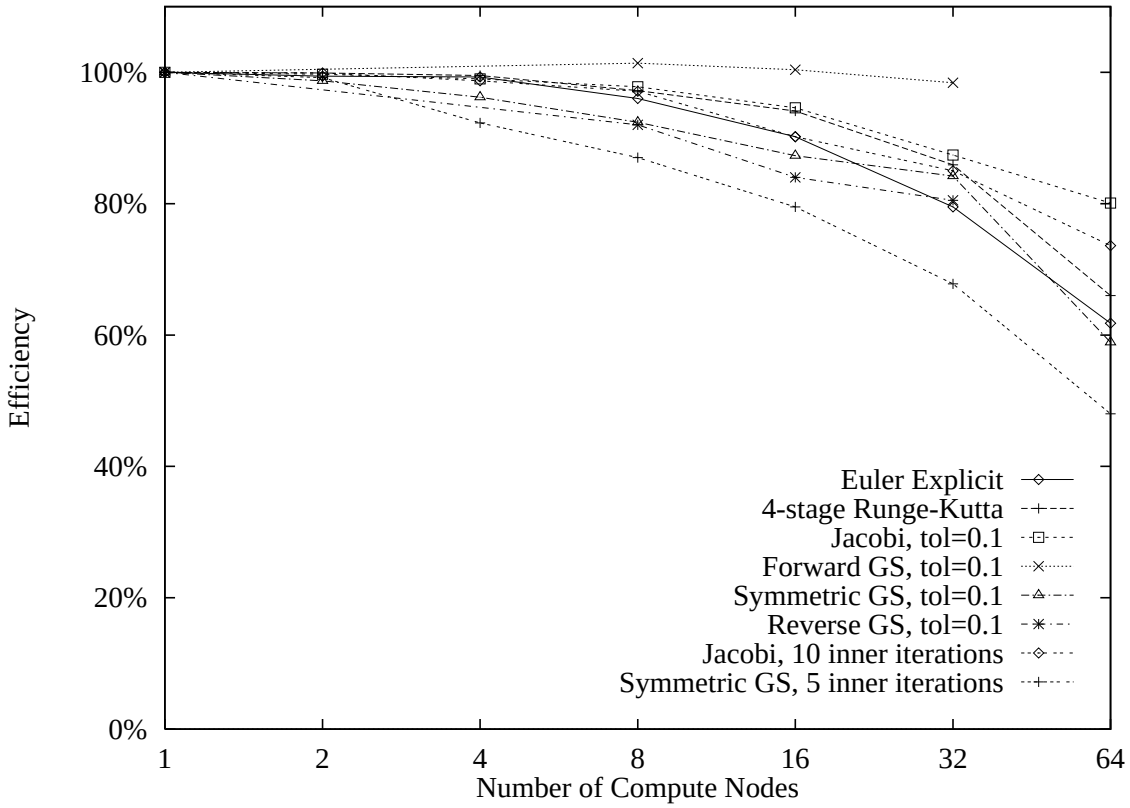


Figure 51: Parallel efficiency for the first-order supersonic bump on the Intel Paragon.

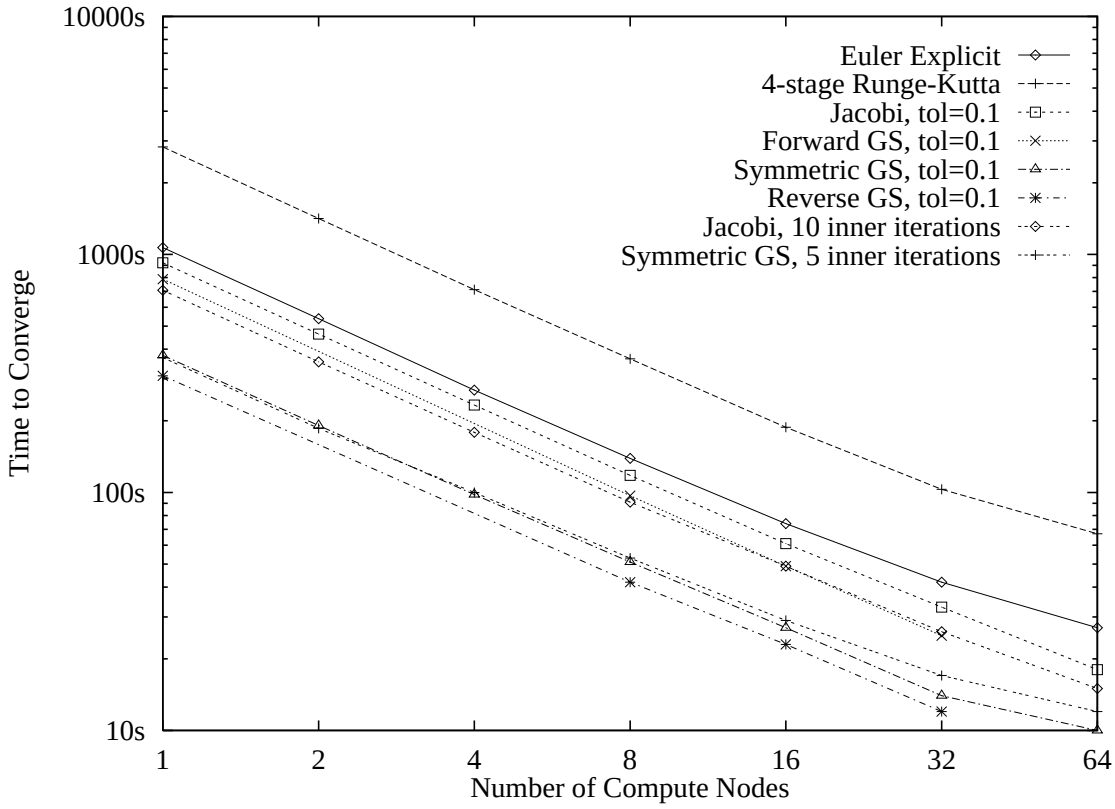


Figure 52: Convergence time for the first-order supersonic bump on the Intel Paragon.

the true GS portion of the matrix. A look at the RGS curve in Figure 52 will show that RGS is indeed the fastest algorithm for this problem.

Convergence Time

Because speedup and efficiency are derived from measurements of convergence time, it may seem somewhat peculiar to discuss the results for convergence time after the speedup and efficiency results. This has been done to make a point. It is often argued that algorithms with a high degree of data locality (usually this means explicit algorithms) should be considered as the best choice for parallelization. The argument runs something like this: if we have high data locality, we minimize communications costs, and therefore have the most efficient parallel algorithm. Any deficiencies in the serial performance of the algorithm are assumed to be more than made up for by the (supposed) parallel performance.

The problem with this argument lies in the definition of efficiency implied. Even if we consider parallel efficiency as defined in the previous section as the only figure of merit for an algorithm, then the conclusion that explicit algorithms have the "best" parallel properties is not universally correct: because explicit algorithms take many times the number of timesteps to converge to steady state as implicit schemes, there is often no advantage in communications. While there is also no question that explicit algorithms are much easier to parallelize, CFD codes should be written for *users*, not code developers, so this point is irrelevant. Finally, engineers are concerned with *getting an answer to a problem*, and don't particularly care if the computer hardware is being used to the fullest extent possible. The "most efficient" algorithm to an engineer is *the one that solves his problem fastest*, and explicit algorithms are simply not up to the task. Venkatakrishnan (1994) reaches a similar conclusion with regard to GMRES as compared to four-stage Jameson-style Runge-Kutta.

In line with the previous two sections, we can define an expected convergence time using Amdahl's Law as well. The expression is simply the reciprocal of Equation (35) for the speedup times t_B, or

$$\frac{t_N}{t_B} = 1 - f_P + \frac{f_P}{N} \tag{38}$$

Figure 53 shows qualitatively what we may expect the timing curves to look like, and the general shape is indeed what we see in Figures 56-65 and 86-95 — usually. Again, we note the anomalous analytic forebody case in Figures 54 and 55.

Each of the figures in the appendices has been plotted using a log-log scale to make it easier to distinguish the curves, especially for small numbers of compute nodes.

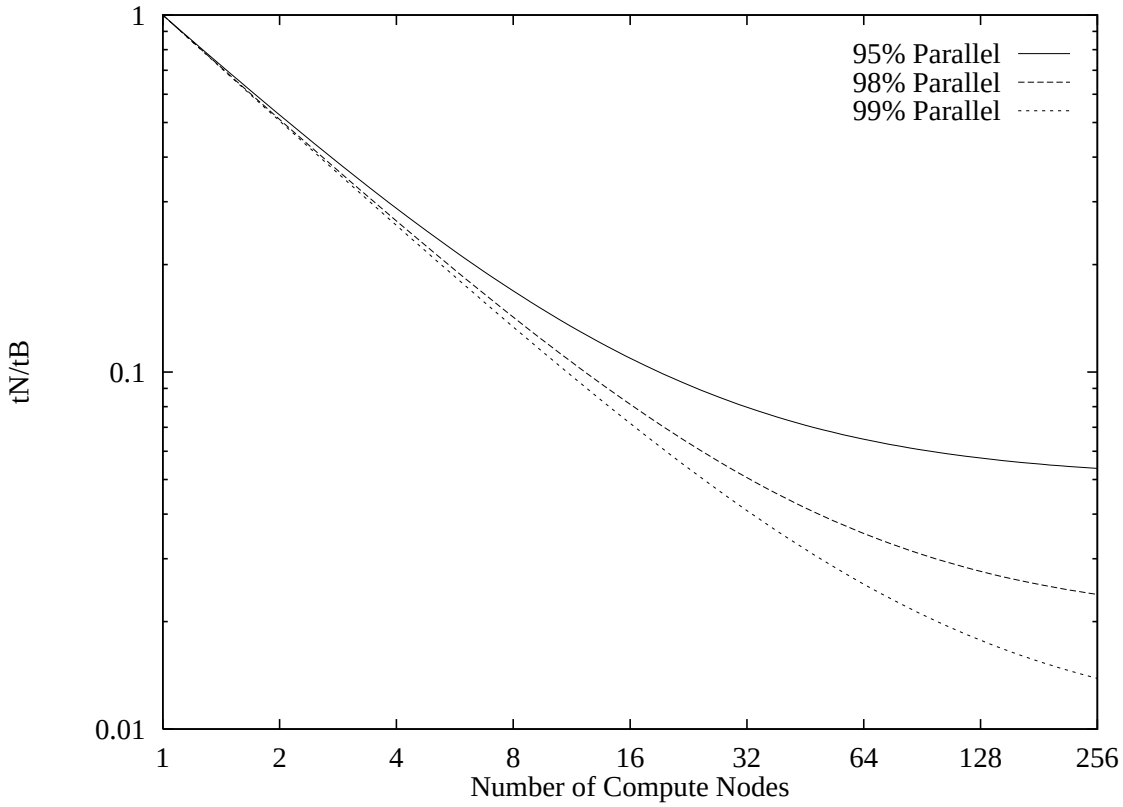


Figure 53: Shape of the convergence-time curves according to Amdahl's Law.

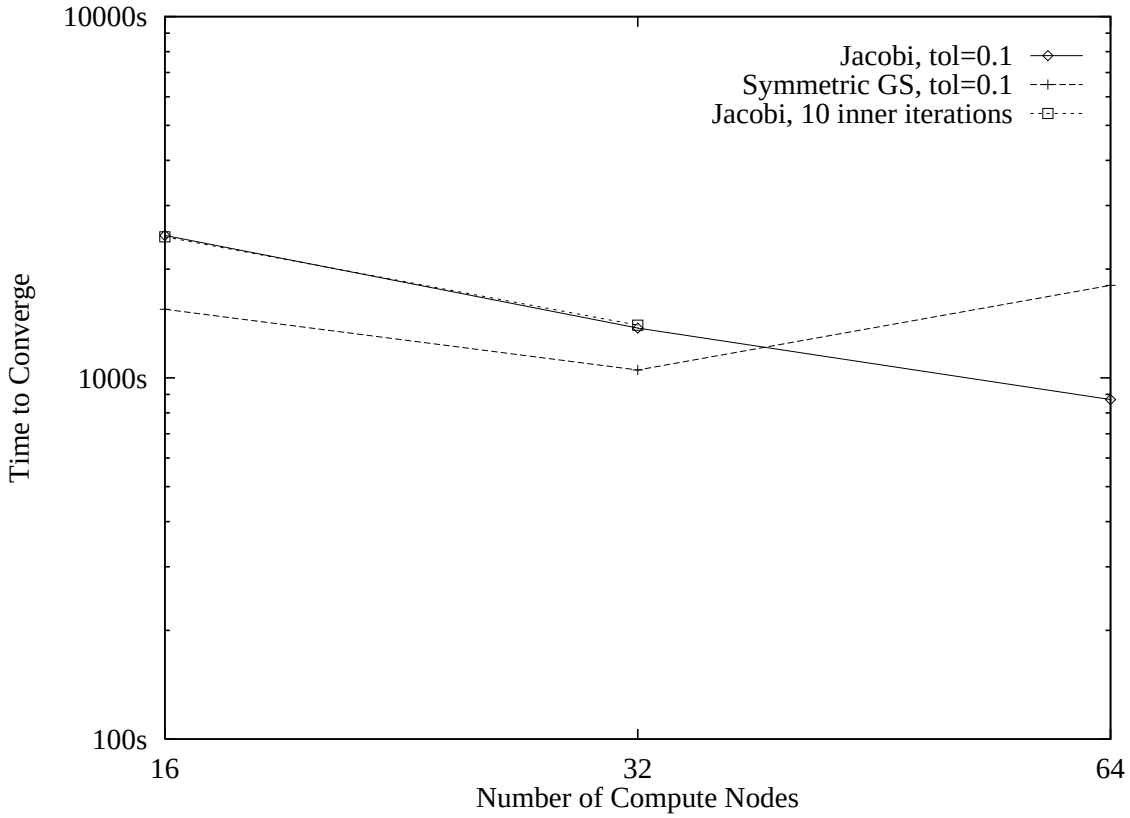


Figure 54: Convergence time for the first-order analytic forebody on the Intel Paragon.

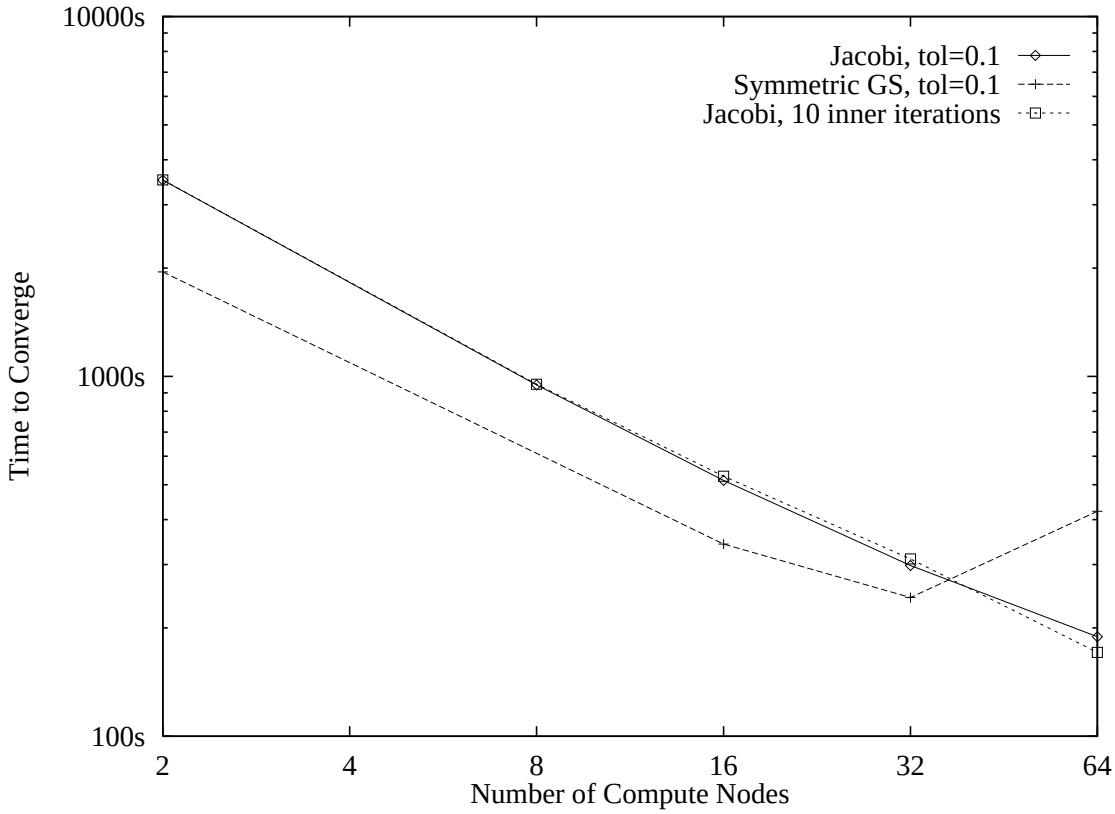


Figure 55: Convergence time for the first-order analytic forebody using thin nodes on the IBM SP-2.

Conclusions

Using parallel efficiency as the sole criterion for evaluating parallel algorithms neglects total time to converge a problem in favor of utilization of the hardware. This often leads to erroneous conclusions when considering which algorithm should be used to solve a given problem of engineering interest because, while the most efficient algorithm is presumed to be faster, this is not always the case.

Even though sharing the evolving solution vector in the inner matrix problem has the largest communications burden of the various approximation schemes considered, the superior convergence properties (due to the ability to run much larger timesteps) more than compensate for the increased communications cost, at least on those machines with a dedicated communications subsystem (as opposed to Ethernet).

Implicit algorithms for unstructured grids can be parallelized on distributed-memory computers and good performance can be achieved with an explicit message-passing paradigm, at least for moderate parallelism (< 1024 compute nodes). Part of the reason for the good performance is the paradigm itself: the programmer is forced to consider the high cost of using nonlocal data because he must coordinate the necessary communications himself. This is not to say that the data-parallel paradigm is inferior, just that the explicit message-passing paradigm is closer to the hardware and therefore easier for the programmer to see what the hardware is doing to perform his communication.

Among the implicit algorithms, Symmetric Gauss-Seidel converged to a specified tolerance has the best overall performance in terms of convergence time. Although there are certain cases where the Jacobi algorithm converged to a specified tolerance has comparable performance, and cases where a fixed number of Jacobi iterations has better performance, the SGS algorithm is faster overall. Also, even though the SGS algorithm often has a lower parallel efficiency than the Jacobi algorithms, it is still faster; the lower efficiency is mostly due to the degradation of the problem into pure Jacobi as compute nodes are added.

For moderately parallel problems, the poor serial performance of the explicit algorithms is not compensated for by their supposed parallel advantages. The argument for explicit algorithms assumes that a high degree of data locality guarantees good performance. While it is true that high data locality ensures lower communications costs and therefore higher parallel efficiency, it does not follow that higher efficiency leads to faster solutions. Furthermore, the benefits of the high data locality are not usually realized in practice. Due to stability restrictions, a very large number of iterations is required to reach steady state with the explicit algorithms. Because the explicit algorithms must share the evolving solution at every stage and every timestep, this guarantees a significant communications burden. Therefore it is not surprising that the efficiency of the explicit algorithms is usually no better than that for the implicit schemes. Even in parallel, explicit algorithms are not competitive with implicit algorithms for steady problems.

Glossary

bandwidth . . . . . Rate of sending data between compute nodes. Typically expressed in megabytes per second.

bisection bandwidth . . . . . The combined bandwidth of all of the communications channels cut by an imaginary bisector of a parallel machine.

coarse-grain parallelism . . . . Parallelism at the problem level. May be associated with either domain decomposition or functional decomposition.

compute node . . . . . Processing unit with its own memory in a distributed-memory parallel system. Typically consists of one processor and local memory, but may have more than one processor or local disk space.

data locality. . . . . Measure of a scheme's dependence on non-local data, i.e., data not present on the local compute node.

domain decomposition . . . . . Distributing a problem across compute nodes according to the data, so that each compute node performs similar computations on a portion of the problem.

explicit scheme . . . . . Time-integration scheme in which the next iterate may be expressed solely in terms of the current iterate. Leads to a scalar equation for the next iterate when integrating scalar equations.

fine-grain parallelism . . . . . Parallelism at the loop or machine-instruction level.

functional decomposition . . . Distributing a problem across compute nodes based on tasks. In a multidisciplinary design optimization problem, for example, one might assign one process to compute the aerodynamics, another to compute loads, and another to compute the structural response.

ghost cell . . . . . Cell outside the computational domain. Used in the computation, but not solved for in the computation. Usually used to store boundary values.

implicit scheme . . . . . Time-integration scheme in which the next iterate may not be expressed solely in terms of the current iterate. Leads to a system of equations for the next iterate even when integrating scalar equations.

latency . . . . . Communications overhead associated with setting up to send a message. Independent of message length. Typically expressed in microseconds.

limiter . . . . . Nonlinear function used to limit solution values extrapolated from the cell to all of its faces.

partition . . . . . Portion of problem data allocated to a particular compute node.

upwind scheme . . . . . A method for computing fluxes based on the Theory of Characteristics.

Bibliography

AeroSoft. 1992. *GASP version 2: The general aerodynamic simulation program.* May be obtained from AeroSoft, Inc., 1872 Pratt Dr., Suite 1275, Blacksburg, VA 24060-6363.

Ajmani, K. and M-S. Liou. 1995. Implicit conjugate-gradient solvers on distributed-memory architectures. Pres. at the *12th AIAA Comp. Fl. Dyn. Conf.,* Jun. 19-22, 1995, San Diego, CA. *AIAA-95-1695.*

Ajmani, K., M-S. Liou, and R. W. Dyson. 1994. Preconditioned implicit solvers for the Navier-Stokes equations on distributed-memory machines. Pres. at the *32nd AIAA Aerosp. Sci. Mtg.,* Jan. 10-13, 1994, Reno, NV. *AIAA-94-0408.*

Amdahl, G. 1967. The validity of the single processor approach to achieving large scale computing capabilities. *AFIPS Conf. Proc. Spring Joint Comp. Conf.* 30:483-485.

Barnard, S. T., A. Pothen and H. Simon. 1995. A spectral algorithm for envelope reduction of sparse matrices. *Numerical linear algebra with applications.* New York: Wiley.

Barth, T. J. 1993. Recent developments in high order k-exact reconstruction on unstructured meshes. Pres. at the *31st AIAA Aerosp. Sci. Mtg.,* Jan. 11-14, 1993, Reno, NV. *AIAA-93-0668.*

Barth, T. J. and D. C. Jesperson. 1989. The design and application of upwind schemes on unstructured meshes. Pres. at the *27th AIAA Aerosp. Sci. Mtg.,* Jan. 9-12, 1989, Reno, NV. *AIAA-89-0366.*

Barth, T. J. and P. O. Frederickson. 1990. Higher order solution of the Euler equations on unstructured grids using quadratic reconstruction. Pres. at the *28th AIAA Aerosp. Sci. Mtg.,* Jan. 8-11, 1990, Reno, NV. *AIAA-90-0013.*

Bergman, C. M., and J. B. Vos. 1991. Parallelization of CFD codes. *Comp. Meth. in Appl. Mech.* 89:523-528.

Brueckner, F. P., D. W. Pepper, and R. H. Chu. 1993. A parallel finite element algorithm for calculating three-dimensional inviscid and viscous compressible flow. Pres. at the *31st AIAA Aerosp. Sci. Mtg.,* Jan. 11-14, 1993, Reno, NV. *AIAA-93-0340.*

Bruner, C. W. S. 1995. Geometric properties of arbitrary polyhedra in terms of face geometry. *AIAA J.* 33:1350. (Technical Note).

Bruner, C. W. S., and R. W. Walters. 1996. A comparison of different levels of approximation in implicit parallel solution algorithms for the Euler equations on unstructured grids. Pres. at *Parallel CFD '95*, Jun. 26-28, 1995, Pasadena, CA. In *Parallel computational fluid dynamics – implementation and results using parallel computers,* pp. 137-144. Ed. S. Taylor, A. Ecer, G. Periaux, and N. Satofuka. Amsterdam: Elsevier.

Burden, R. L., and J. D. Faires. 1989. *Numerical analysis*. 5th ed. Boston: PWS Publ. Co.

Candler, G. V., M. J. Wright, and J. D. McDonald. 1994. A data-parallel LU-SGS method for reacting flows. Pres. at the *32nd AIAA Aerosp. Sci. Mtg.,* Jan. 10-13, 1994, Reno, NV. *AIAA-94-0410.*

Carnahan, B., H. A. Luther, and J. O. Wilkes. 1969. *Applied numerical methods.* New York: Wiley.

Chyczewski, T. S., F. Marconi, R. B. Pelz, and E. N. Curchitzer. 1993. Solution of the Euler and Navier-Stokes equations on parallel processors using a transposed/Thomas ADI algorithm. Pres. at the *11th AIAA Comp. Fl. Dyn. Conf.,* Jul. 6-9, 1993, Orlando, FL. *AIAA-93-3310.*

Cook, P. H., M. A. McDonald, and M. C. P. Firmin. 1979. Aerofoil RAE 2822 — pressure distributions, and boundary layer and wake measurements. In *Experimental database for computer program assessment*. AGARD AR-138.

Courant, R., E. Isaacson, and M. Reeves. 1952. On the solution of nonlinear hyperbolic differential equations by finite differences. *Comm. Pure and Appl. Math.* 5:243-255.

Deshpande, M., J. Feng, C. L. Merkle, and A. Deshpande. 1993. Implementaion of a parallel algorithm on a distributed network. Pres. at the *31st AIAA Aerosp. Sci. Mtg.,* Jan. 11-14, 1993, Reno, NV. *AIAA-93-0058.*

Drikakis, D. 1996. Development and implementation of parallel high resolution schemes in 3D flows over bluff bodies. Pres. at *Parallel CFD '95*, Jun. 26-28, 1995, Pasadena, CA. In *Parallel computational fluid dynamics – implementation and results using parallel computers*, pp. 191-198. Ed. S. Taylor, A. Ecer, G. Periaux, and N. Satofuka. Amsterdam: Elsevier.

Drikakis, D., E. Schreck, and F. Durst. 1994. A comparative study of numerical methods for incompressible and compressible flows on different parallel machines. Pres. at the *32nd AIAA Aerosp. Sci. Mtg.,* Jan. 10-13, 1994, Reno, NV. *AIAA-94-0412.*

Duff, I. S. 1977. A survey of sparse matrix research. *Proc. IEEE* 65:500-535.

Duff, I. S., A. M. Erisman, and J. K. Reid. 1986. *Direct methods for sparse matrices*. Oxford: Oxford Univ. Press.

Flynn, M. J. 1972. Some computer organizations and their effectiveness. *IEEE Trans. Comp.* C-21:948-960.

Godunov, S. K. 1959. A difference scheme for numerical computation of discontinuous solution of hydrodynamic equations. Translated from the Russian by US Joint Publ. Res. Service. 1969. *JPRS 7226*.

Golub, G. H., and C. F. Van Loan. 1989. *Matrix computations*. 2d ed. Baltimore: Johns Hopkins Univ. Press.

Goodman, J. B., and R. J. LeVeque. 1985. On the accuracy of stable schemes for 2D scalar conservation laws. *Math. Comp.* 45:15-21.

Gropp, W., E. Lusk, and A. Skjellum. 1994. *Using MPI*. Cambridge: MIT Press.

Harten, A. 1983. High resolution schemes for hyperbolic conservation laws. *J. Comp. Phys.* 49:357-393.

Helf, C., K. Birken, and U. Küster. 1996. Parallelization of a highly unstructured Euler-solver based on arbitrary polygonal control volumes. Pres. at *Parallel CFD '95*, Jun. 26-28, 1995, Pasadena, CA. In *Parallel computational fluid dynamics – implementation and results using parallel computers*, pp. 169-174. Ed. S. Taylor, A. Ecer, G. Periaux, and N. Satofuka. Amsterdam: Elsevier.

Hirsch, C. 1990. *Numerical computation of internal and external flows*. Vol. 2, *Computational methods for inviscid and viscous flows*. New York: Wiley.

Hixon, D. and L. N. Sankar. 1994. Unsteady compressible 2-D flow calculation on a MIMD parallel supercomputer. Pres. at the *32nd AIAA Aerosp. Sci. Mtg.*, Jan. 10-13, 1994, Reno, NV. *AIAA-94-0757*.

Hummel, D. 1978. On the vortex formation over a slender wing at large angles of incidence. In *High angle of attack aerodynamics — papers presented and discussions from the fluid dynamics panel symposium held at Sandefjord, Norway 4-6 October 1978*. AGARD CP-247.

IBM. 1996. SP-2 network topology webpage. http://ibm.tc.cornell.edu/ibm/pps/doc/css/node2.html

Jameson, A., W. Schmidt, and E. Turkel. 1981. Numerical simulation of the Euler equations by finite volume methods using Runge-Kutta time stepping schemes. Pres. at the *5th AIAA Comp. Fl. Dyn. Conf. AIAA-81-1259*

Koelbel, C. H., D. B. Loveman, R. S. Schreiber, G. L. Steele, and M. E. Zosel. 1993. *The high performance fortran handbook.* Cambridge: MIT Press.

Lanteri, S. and M. Loriot. 1996. Parallel solutions of three-dimensional compressible flows using a mixed finite element/finite volume method on unstructured grids. Pres. at *Parallel CFD '95*, Jun. 26-28, 1995, Pasadena, CA. In *Parallel computational fluid dynamics – implementation and results using parallel computers*, pp. 153-160. Ed. S. Taylor, A. Ecer, G. Periaux, and N. Satofuka. Amsterdam: Elsevier.

Löhner, R., and P. Parikh. 1988. Generation of three-dimensional unstructured grids by the advancing front method. *Intl. J. Num. Meth. Fluids.* 8:1135-1149.

Mitchell, C. R. 1994. Improved reconstruction schemes for the Navier-Stokes equations on unstructured meshes. Pres. at the *32nd AIAA Aerosp. Sci. Mtg.,* Jan. 10-13, 1994, Reno, NV. *AIAA-94-0642.*

Morano, E. and D. Mavriplis. 1994. Implementation of a parallel unstructured Euler solver on the CM-5. Pres. at the *32nd AIAA Aerosp. Sci. Mtg.,* Jan. 10-13, 1994, Reno, NV. *AIAA-94-0755.*

Otto, J. C. 1993. Parallel execution of a three-dimensional, chemically reacting, Navier-Stokes code on distributed-memory machines. Pres. at the *11th AIAA Comp. Fl. Dyn. Conf.,* Jul. 6-9, 1993, Orlando, FL. *AIAA-93-3307.*

Peric, M. and E. Schreck. 1996. Analysis of efficiency of implicit CFD methods on MIMD computers. Pres. at *Parallel CFD '95*, Jun. 26-28, 1995, Pasadena, CA. In *Parallel computational fluid dynamics – implementation and results using parallel computers*, pp. 145-152. Ed. S. Taylor, A. Ecer, G. Periaux, and N. Satofuka. Amsterdam: Elsevier.

Pothen, A., H. D. Simon and K-P. Liou. 1990. Partitioning sparse matrices with eigenvectors of graphs. *SIAM J. Matrix Anal. Appl.* 11:430-452.

Povitsky, A. and M. Wolfstein. 1995. Numerical solution of flow problems on a parallel computer. Pres. at the *12th AIAA Comp. Fl. Dyn. Conf.*, Jun. 19-22, 1995, San Diego, CA. *AIAA-95-1698.*

Roe, P. L. 1981. Approximate Riemann solvers, parameter vectors and difference schemes. *J. Comp. Phys.* 43:357-372.

Roe, P. L. 1985. Some contributions to the modelling of discontinuous flows. *Proc. 1983 AMS-SIAM summer seminar on large scale computing in fluid mechanics.* In *Lectures in applied mathematics* 22:163-193.

Saad, Y. and M. H. Schulz. 1986. GMRES: A generalized minimum residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. & Stat. Comp.* 7:856-869.

Scherr, S. 1995. Portable parallelization of block-structured flow solvers. Pres. at the *12th AIAA Comp. Fl. Dyn. Conf.,* Jun. 19-22, 1995, San Diego, CA. *AIAA-95-1760.*

Schmitt, V., and F. Charpin. 1979. Pressure distributions on the ONERA M6-wing at transonic Mach numbers. In *Experimental database for computer program assessment.* AGARD AR-138.

Silva, R. S. and R. C. Almeida. 1996. Performance of a Euler solver using a distributed system. Pres. at *Parallel CFD '95,* Jun. 26-28, 1995, Pasadena, CA. In *Parallel computational fluid dynamics – implementation and results using parallel computers,* pp. 175-182. Ed. S. Taylor, A. Ecer, G. Periaux, and N. Satofuka. Amsterdam: Elsevier.

Simon, H. D. 1991. Partitioning of unstructured problems for parallel processing. *Computing Sys. in Eng.* 2:135-148.

Simon, H. D. and L. Dagum. 1993. Experience in using SIMD and MIMD parallelism for computational fluid dynamics. *Appl. Num. Math.* 12:431-442.

Simon, H. D., W. R. Van Dalsem, and L. Dagum. 1992. Parallel computational fluid dynamics: current status and future requirements. NASA Tech. Rep. *RNR-92-004,* NASA Ames Res. Ctr., Moffett Field, CA.

Snir, M., S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. 1996. *MPI: The complete reference.* Cambridge: MIT Press.

Spekreijse, S. 1987. Multigrid solution of monotone second-order discretizations of hyperbolic conservation laws. *Math. Comp.* 49:135-155.

Stagg, A. K., D. D. Cline, J. N. Shadid, and G. F. Carey. 1993. A performance comparison of massively parallel parabolized Navier-Stokes solutions. Pres. at the *31st AIAA Aerosp. Sci. Mtg.,* Jan. 11-14, 1993, Reno, NV. *AIAA-93-0059.*

Stamatis, A. G. and K. D. Papailiou. 1996. Implementation of a fractional step algorithm for the solution of the Euler equations on scalable computers. Pres. at *Parallel CFD '95*, Jun. 26-28, 1995, Pasadena, CA. In *Parallel computational fluid dynamics – implementation and results using parallel computers*, pp. 161-168. Ed. S. Taylor, A. Ecer, G. Periaux, and N. Satofuka. Amsterdam: Elsevier.

Steger, J. L. and R. F. Warming. 1981. Flux vector splitting of the inviscid gas-dynamic equations with applications to finite difference methods. *Comp. Meth. in Appl. Mech. & Eng*. 13:175-188.

Stoer, J. and R. Bulirsch. 1980. *Introduction to numerical analysis.* New York: Springer-Verlag.

Sweby, P. K. 1984. High resolution schemes using flux limiters for hyperbolic conservation laws. *SIAM J. Numer. Anal*. 21:995-1011.

Townsend, J. C., D. T. Howell, I. K. Collins, and C. Hayes. 1979. Surface pressure data on a series of analytic forebodies at Mach numbers from 1.7 to 4.5 and combined angles of attack and sideslip. *NASA TM 80062.*

Tysinger, T. L. and D. A. Caughey. 1993. Distributed parallel processing applied to an implicit multigrid Euler/Navier-Stokes algorithm. Pres. at the *31st AIAA Aerosp. Sci. Mtg.*, Jan. 11-14, 1993, Reno, NV. *AIAA-93-0057.*

Underwood, M., D. Riggins, B. McMillin, E. Lu, and L. Reeves. 1993. The computation of supersonic combustor flows using multi-computers. Pres. at the *31st AIAA Aerosp. Sci. Mtg.*, Jan. 11-14, 1993, Reno, NV. *AIAA-93-0060.*

Van Albada, G. D., B. Van Leer, and W. W. Roberts. 1982. A comparative study of computational methods in cosmic gas dynamics. *Astron. and Astrophys*. 108:76-84.

Van Leer, B. 1979. Towards the ultimate conservative difference scheme. V. A second order sequel to Godunov's method. *J. Comp. Phys*. 32:101-136.

Van Leer, B. 1982. Flux vector splitting for the Euler equations. *Proc. 8th Intl. Conf. on Numerical Meth. in Fluid Dynamics.* Berlin: Springer-Verlag.

Venkatakrishnan, V. 1993. On the accuracy of limiters and convergence to steady state solutions. Pres. at the *31st AIAA Aerosp. Sci. Mtg.*, Jan. 11-14, 1993, Reno, NV. *AIAA-93-0880.*

Venkatakrishnan, V. 1994. Parallel implicit unstructured grid Euler solvers. *AIAA J.* 32:1985-1991.

Venkatakrishnan, V. 1995. Implicit schemes and parallel computing in unstructured grid CFD. Lecture notes prepared for the 26th computational fluid dynamics lecture series program of the von Karman Institute (VKI) for Fluid Dynamics, Rhode-Saint-Genese, Belgium, 13-17 March 1995. *NASA Cont. Rep. 195071 (ICASE Rep. 95-28).*

Venkatakrishnan, V., and T. J. Barth. 1989. Application of direct solvers to unstructured meshes for the Euler and Navier-Stokes equations using upwind schemes. Pres. at the *27th AIAA Aerosp. Sci. Mtg.,* Jan. 9-12, 1989, Reno, NV. *AIAA-89-0364.*

Venkatakrishnan, V., H. Simon, and T. Barth. 1992. A MIMD implementation of a parallel Euler solver for unstructured grids. *J. Supercomp.* 6:117-137.

Weinberg, Z. and L. N. Long. 1994. A massively parallel solution of the three dimensional Navier-Stokes equations on unstructured, adaptive grids. Pres. at the *32nd AIAA Aerosp. Sci. Mtg.,* Jan. 10-13, 1994, Reno, NV. *AIAA-94-0760.*

Appendix A: Summary of First-order Results

Time

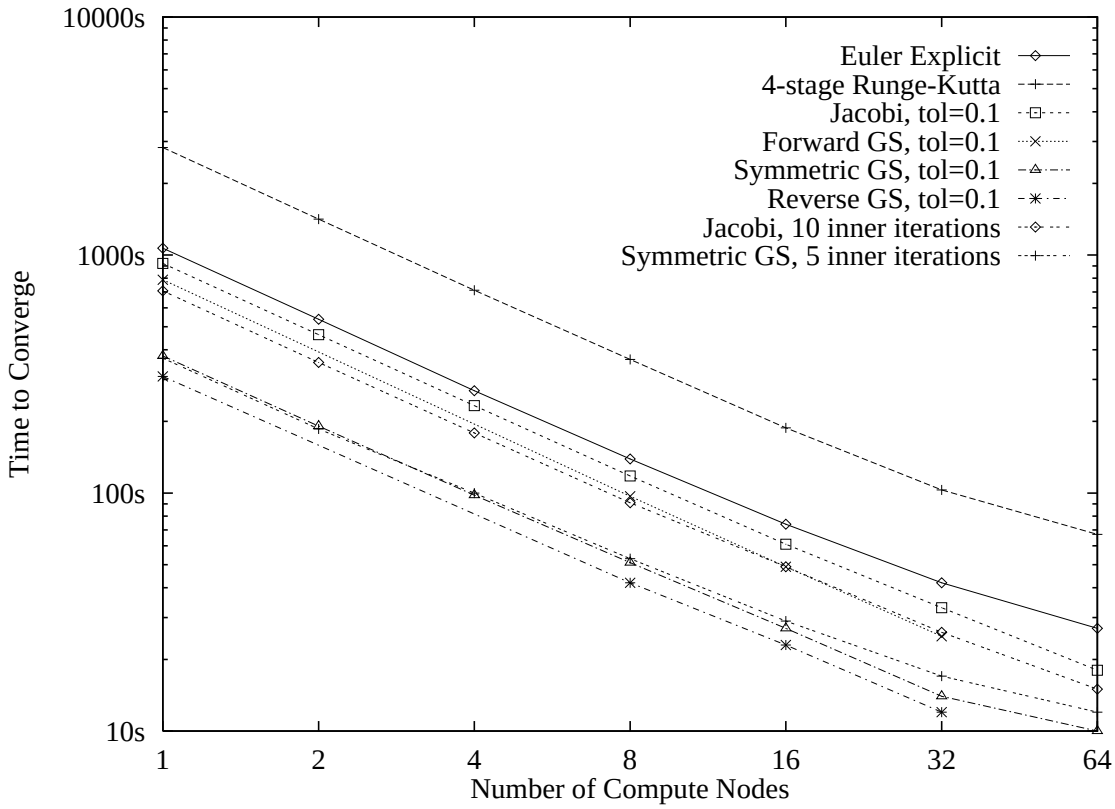


Figure 56: Convergence time for the supersonic bump on the Intel Paragon.

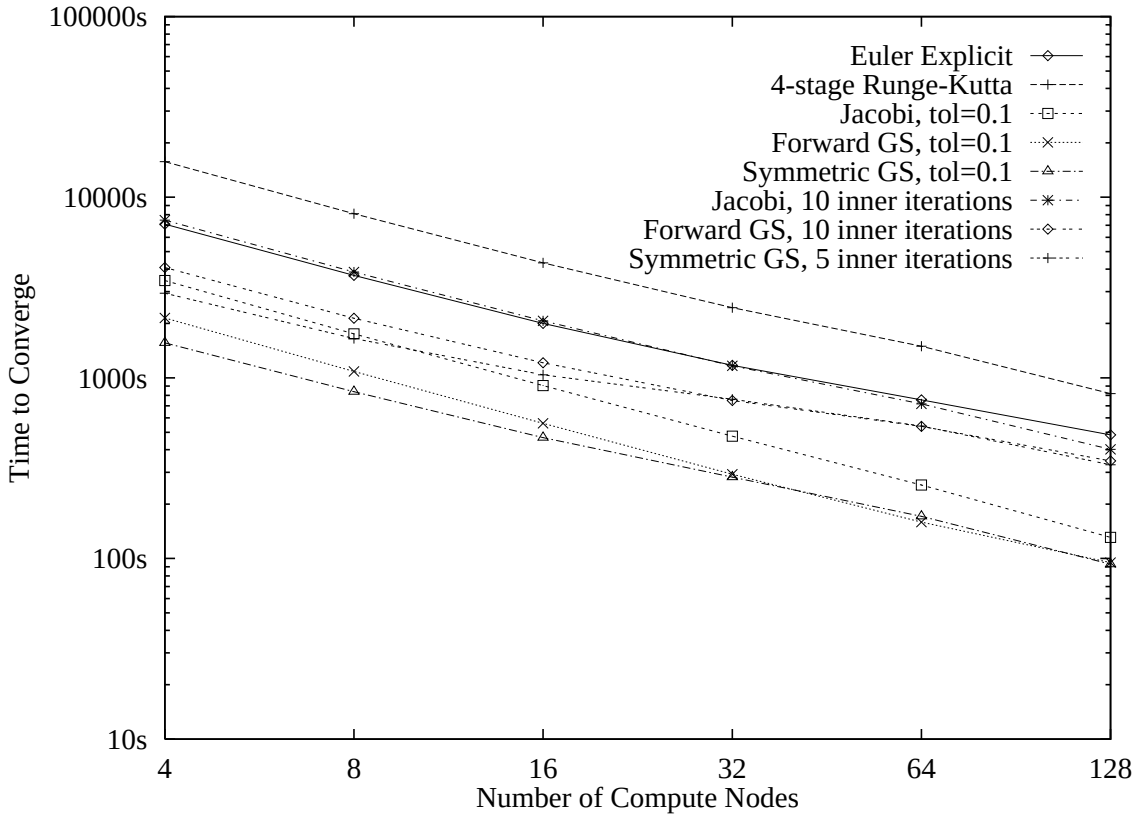


Figure 57: Convergence time for the RAE 2822 airfoil on the Intel Paragon.

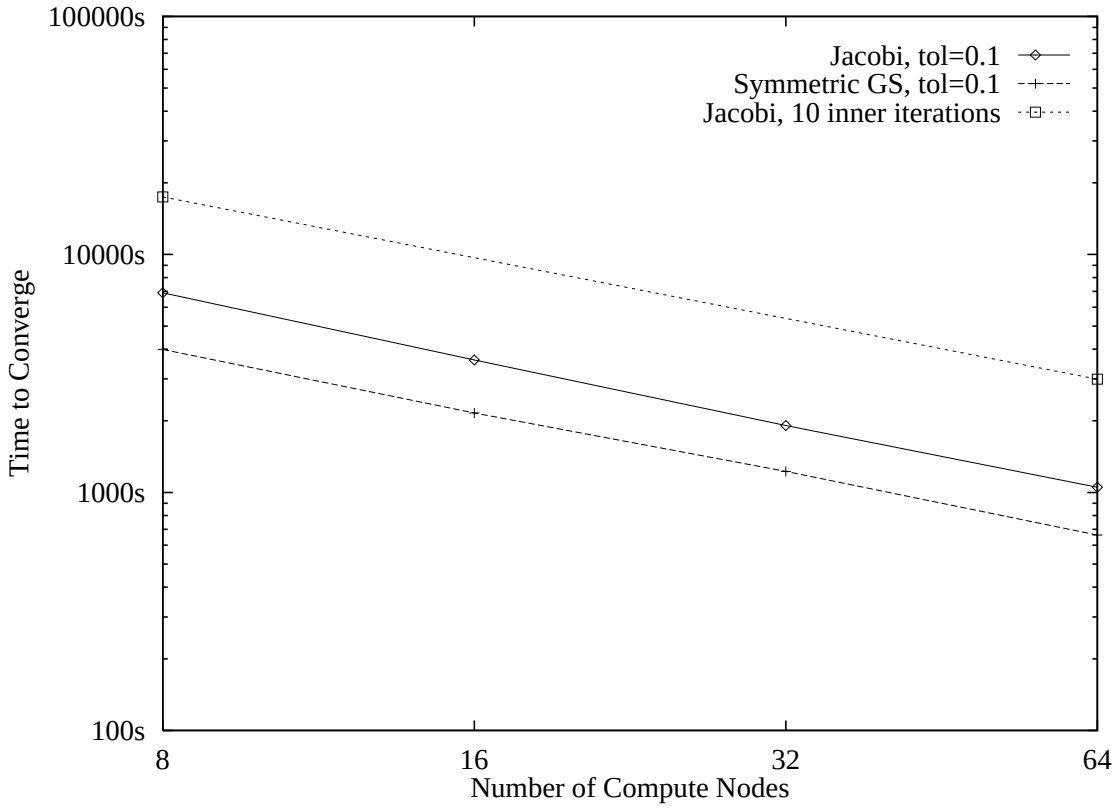


Figure 58: Convergence time for the Hummel delta wing on the Intel Paragon.

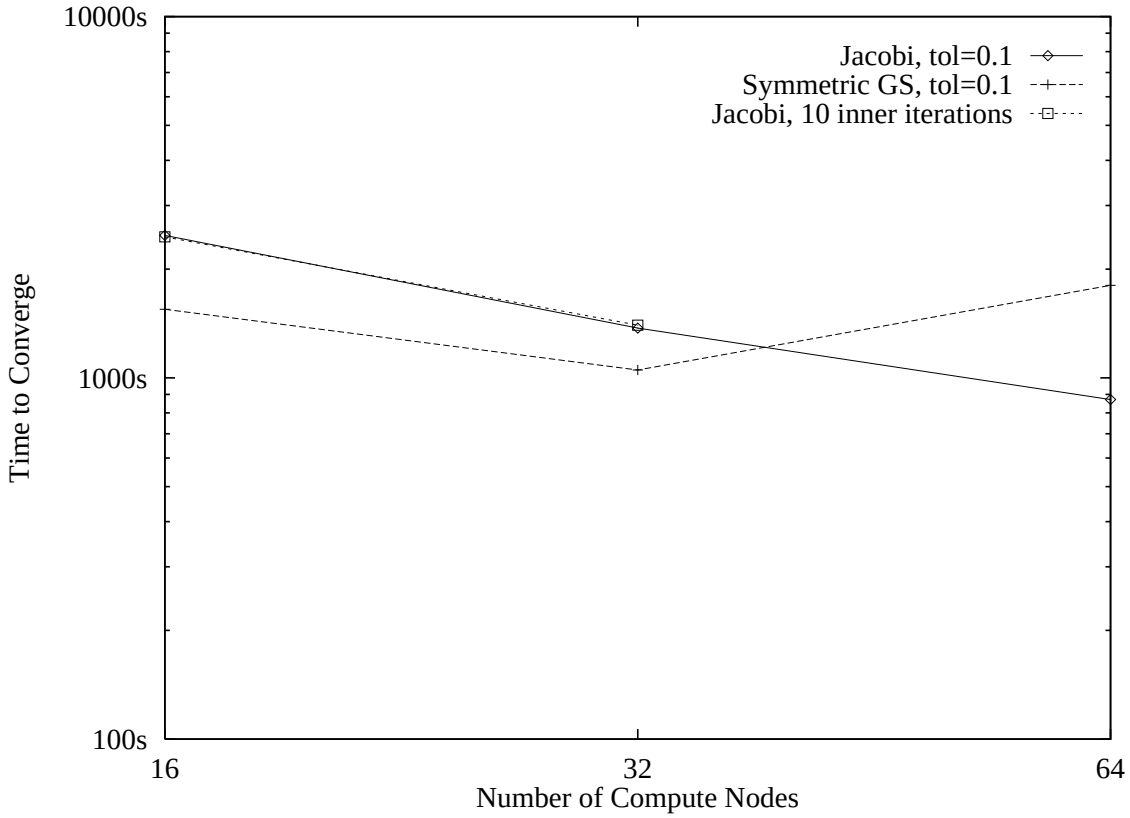


Figure 59: Convergence time for the analytic forebody on the Intel Paragon.

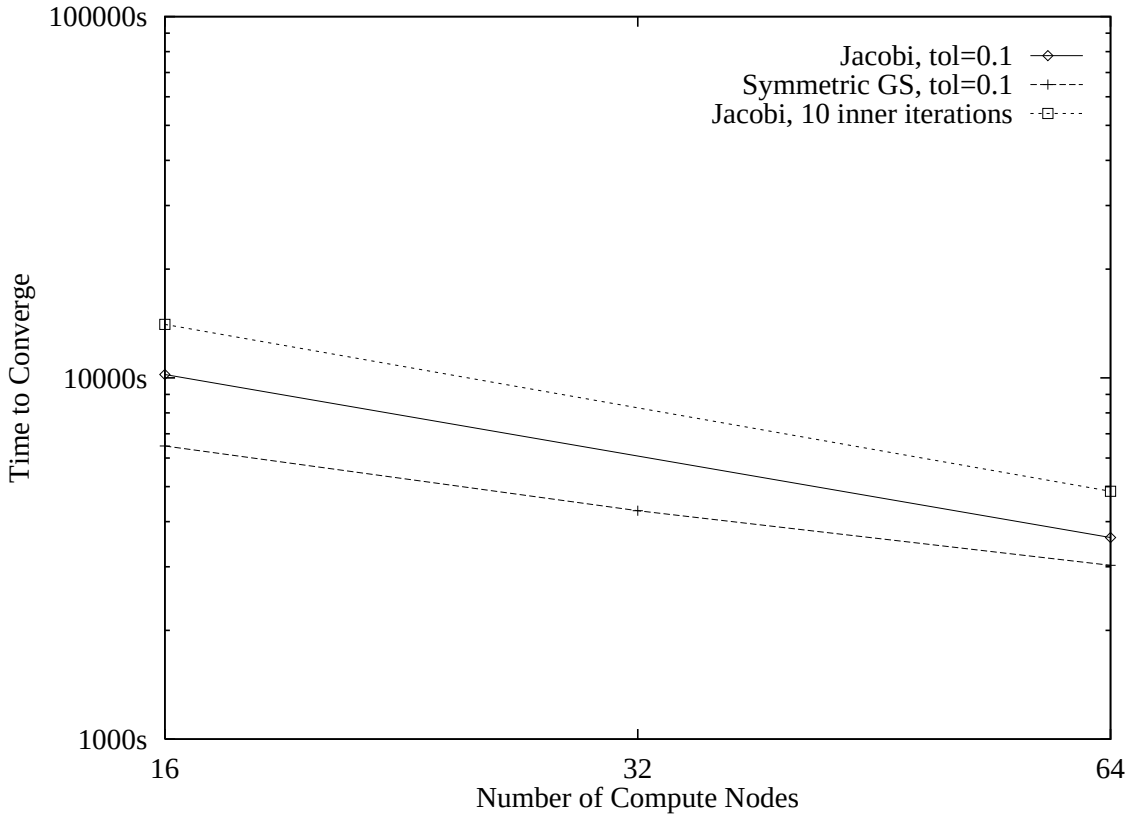
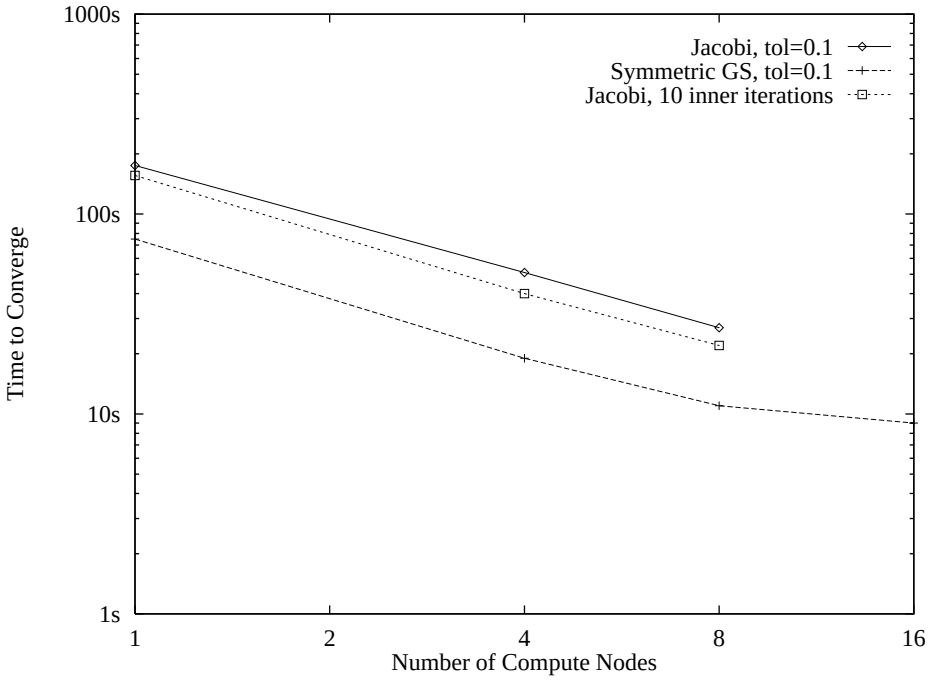
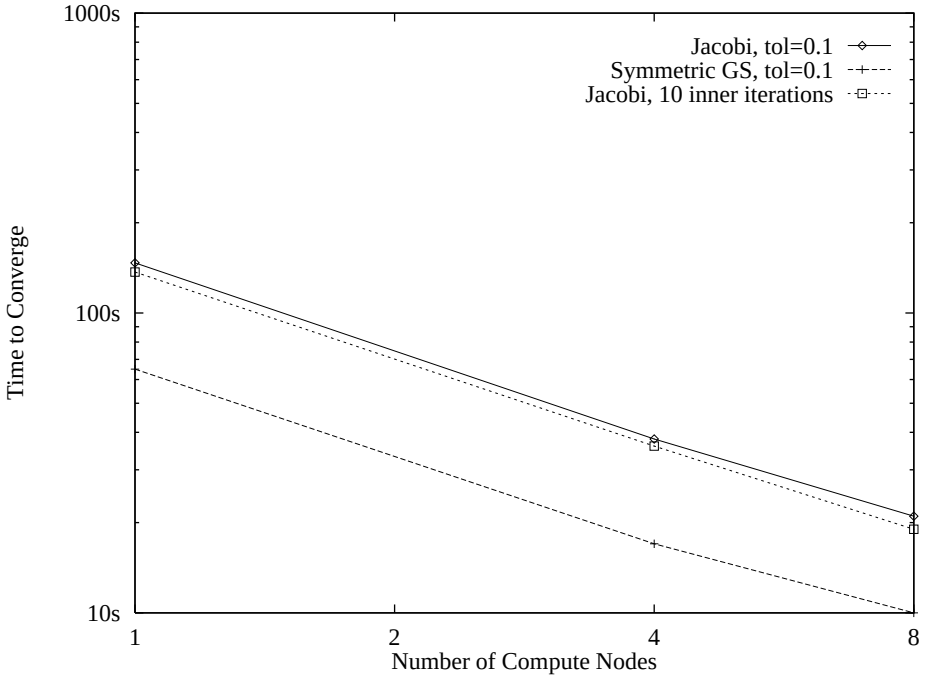


Figure 60: Convergence time for the ONERA M6 wing on the Intel Paragon.

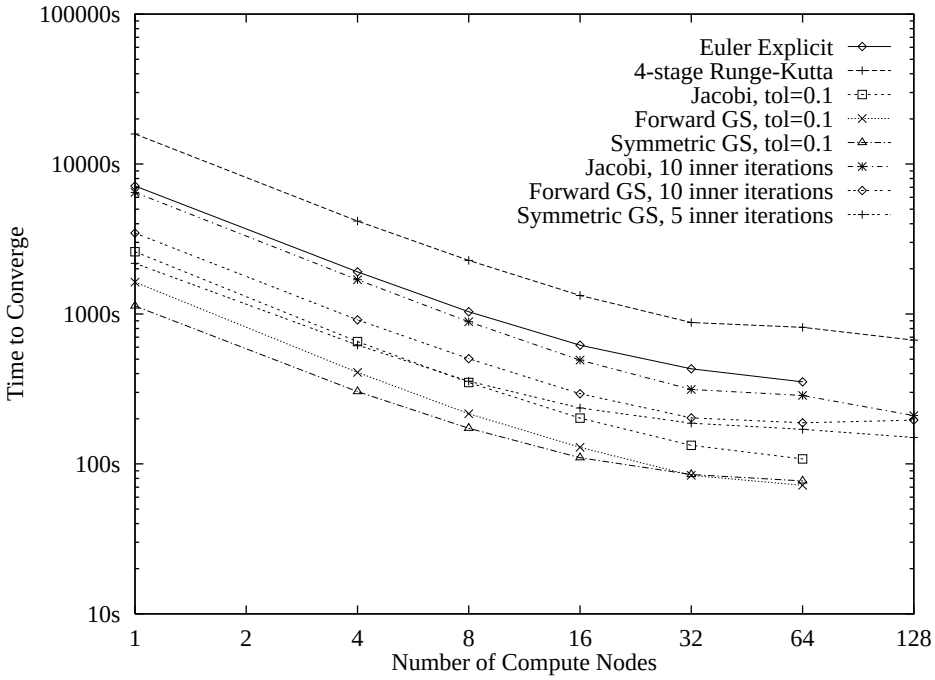


a) thin nodes

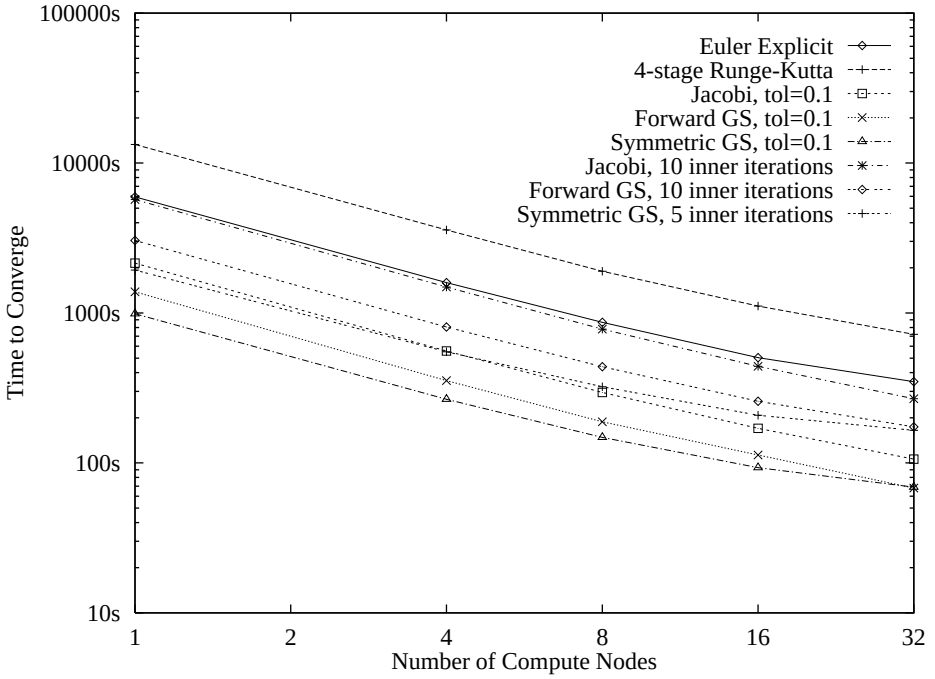


b) wide nodes

Figure 61: Convergence time for the supersonic bump on the IBM SP-2.

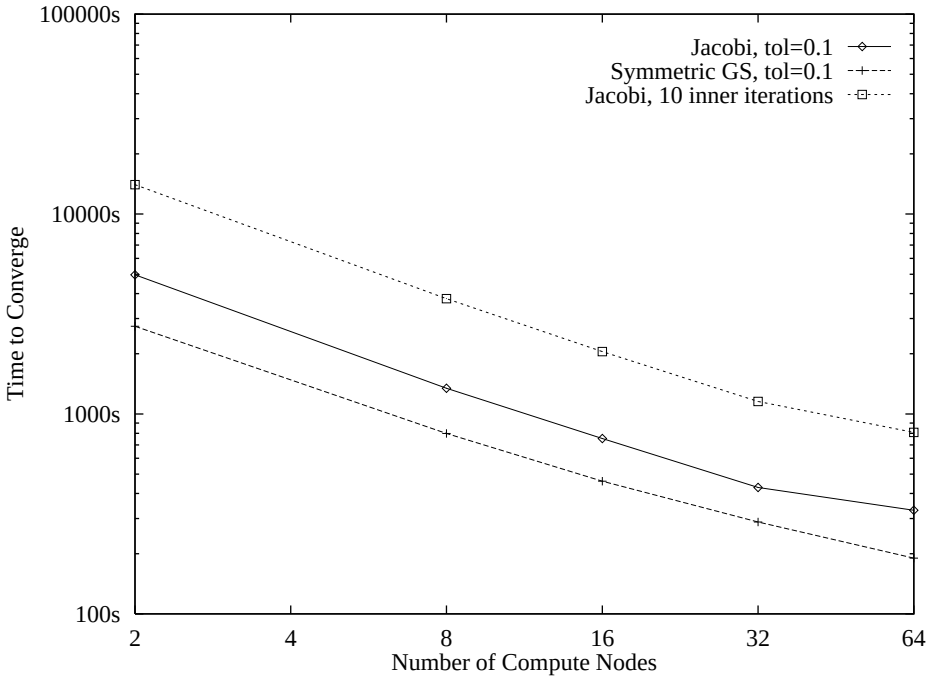


a) thin nodes

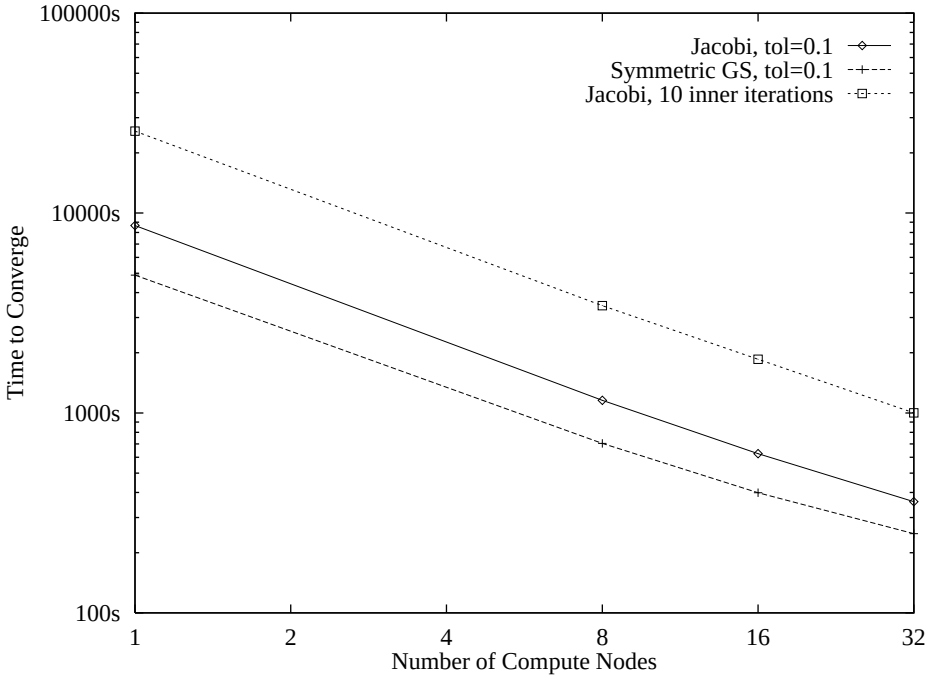


b) wide nodes

Figure 62: Convergence time for the RAE 2822 airfoil on the IBM SP-2.

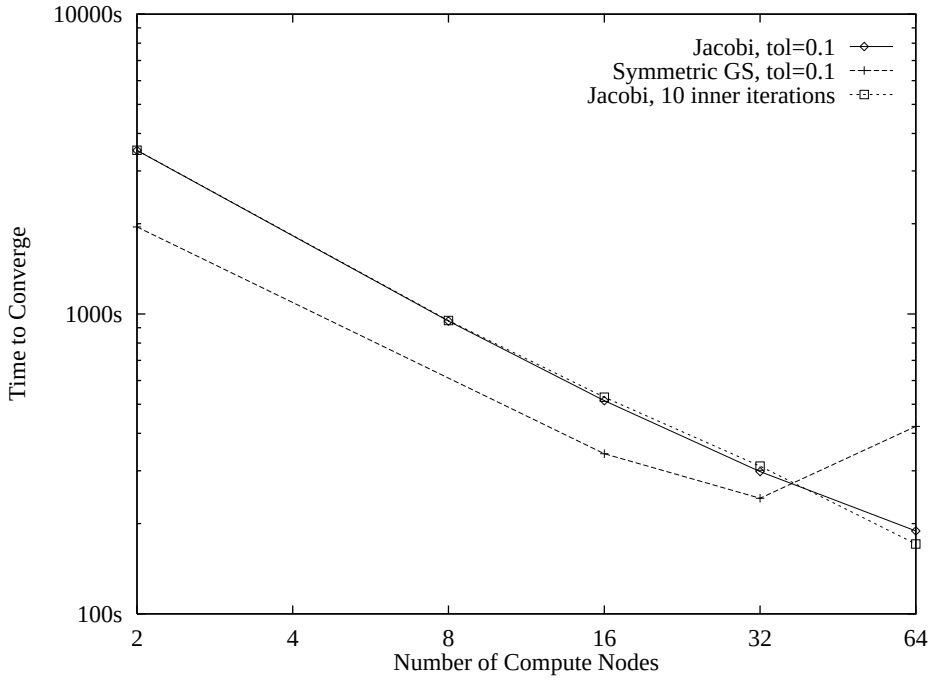


a) thin nodes

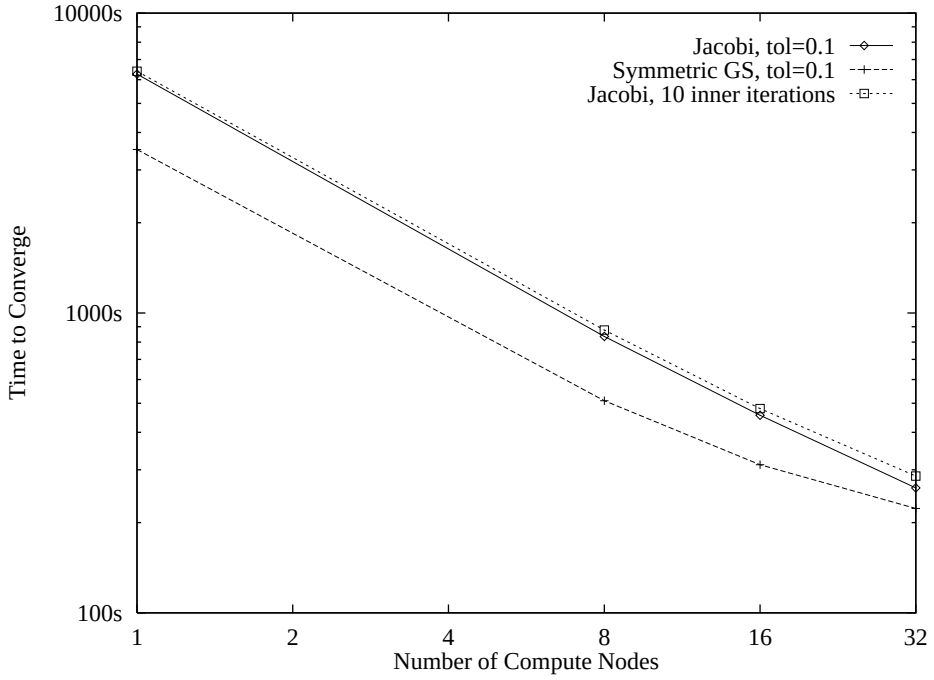


b) wide nodes

Figure 63: Convergence time for the Hummel delta wing on the IBM SP-2.

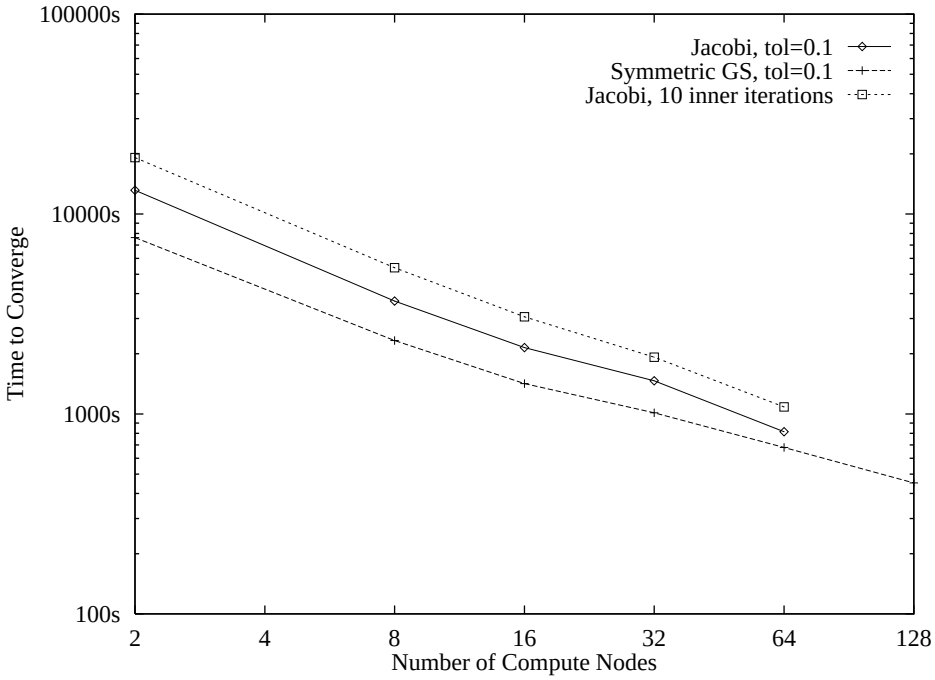


a) thin nodes

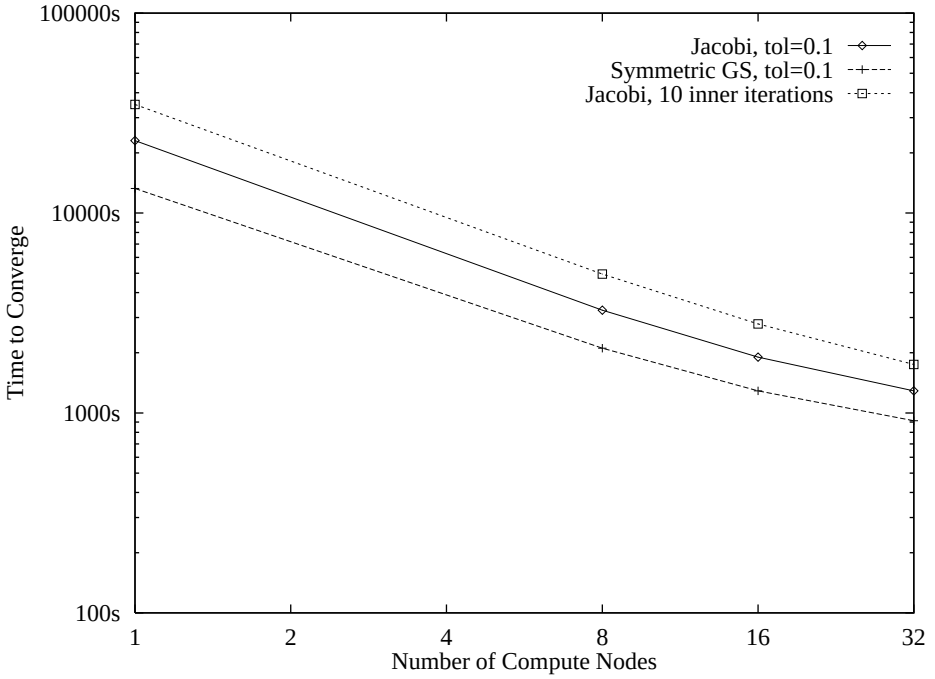


b) wide nodes

Figure 64: Convergence time for the analytic forebody on the IBM SP-2.



a) thin nodes



b) wide nodes

Figure 65: Convergence time for the ONERA M6 wing on the IBM SP-2.

Speedup

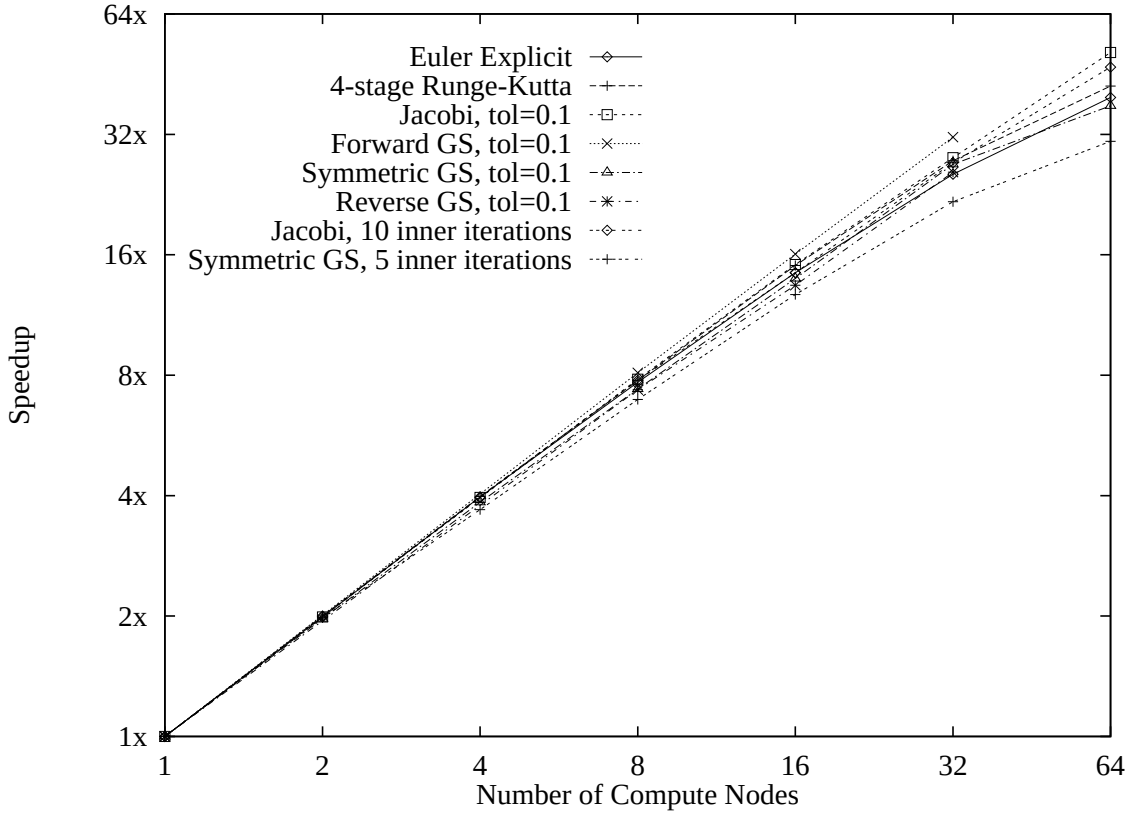


Figure 66: Speedup for the supersonic bump on the Intel Paragon.

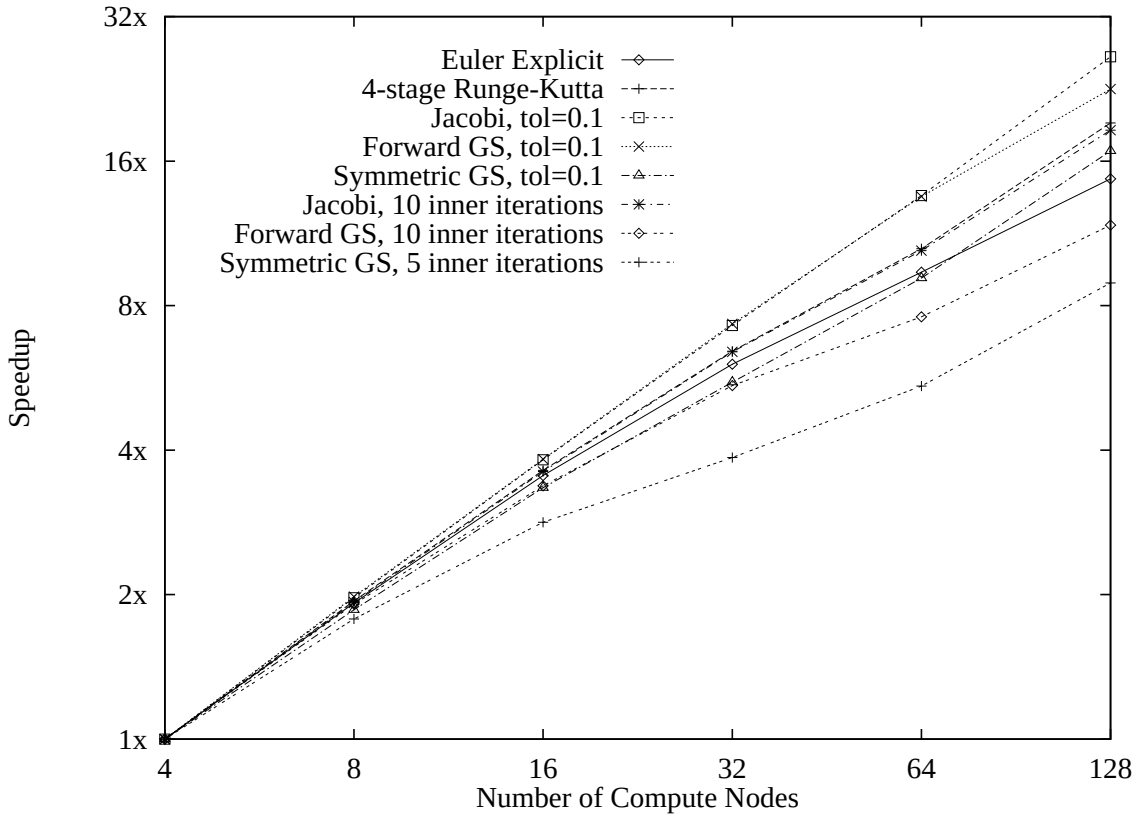


Figure 67: Speedup for the RAE 2822 airfoil on the Intel Paragon.

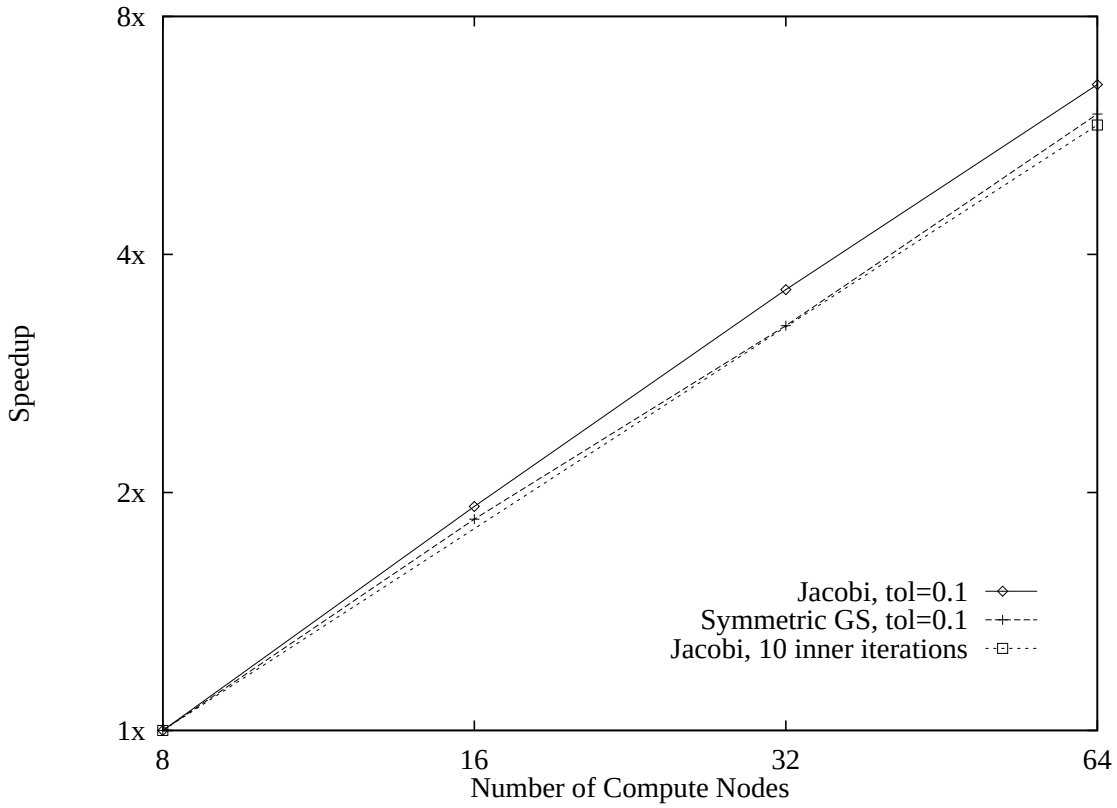


Figure 68: Speedup for the Hummel delta wing on the Intel Paragon.

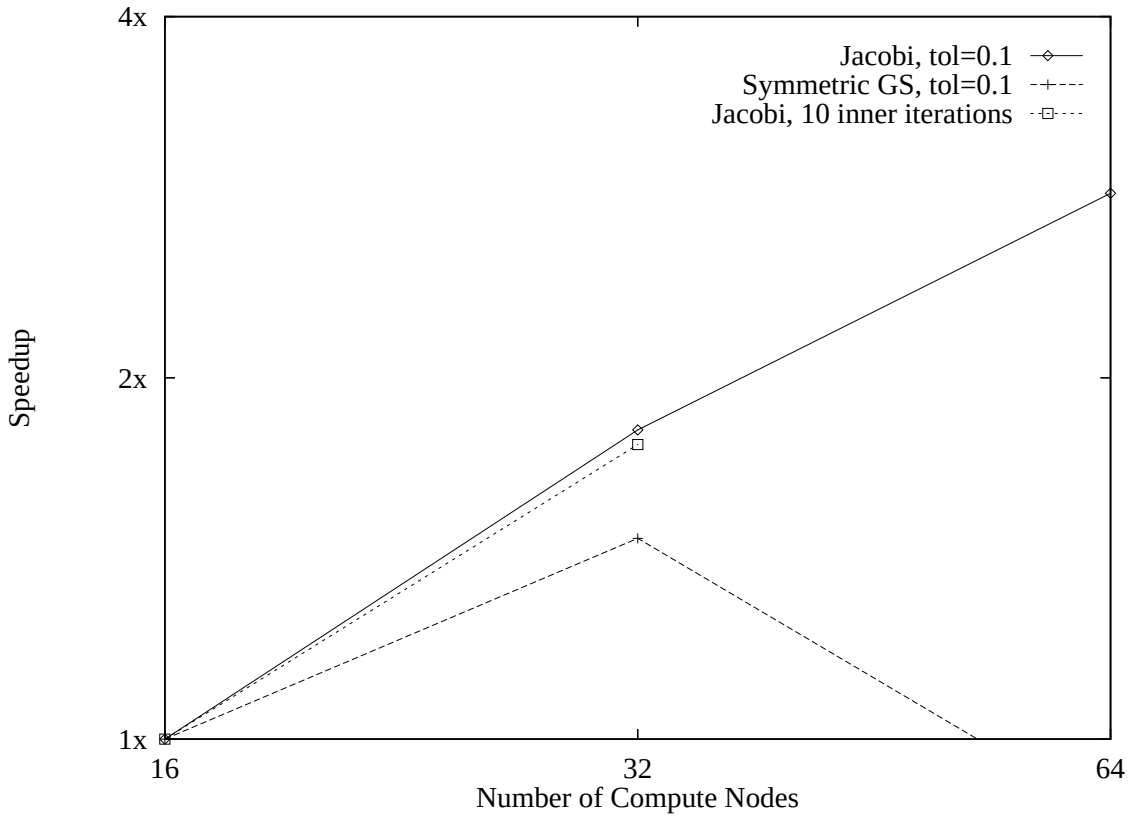


Figure 69: Speedup for the analytic forebody on the Intel Paragon.

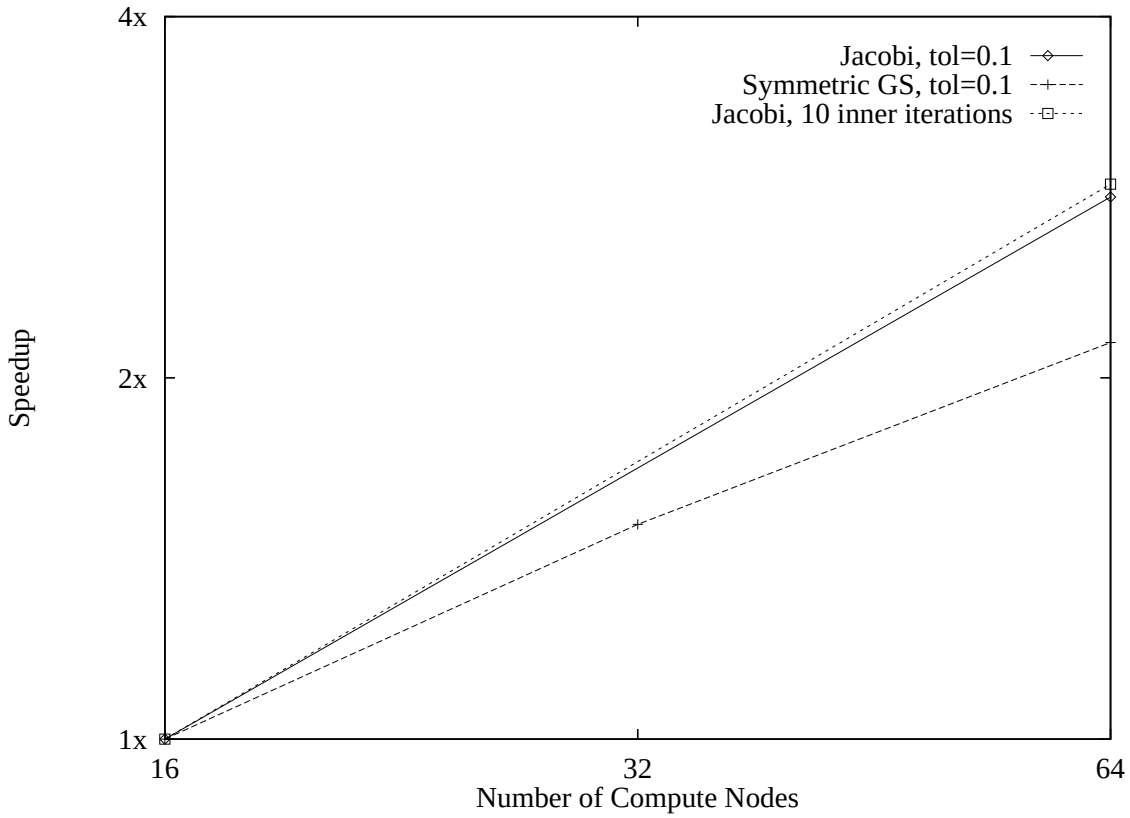
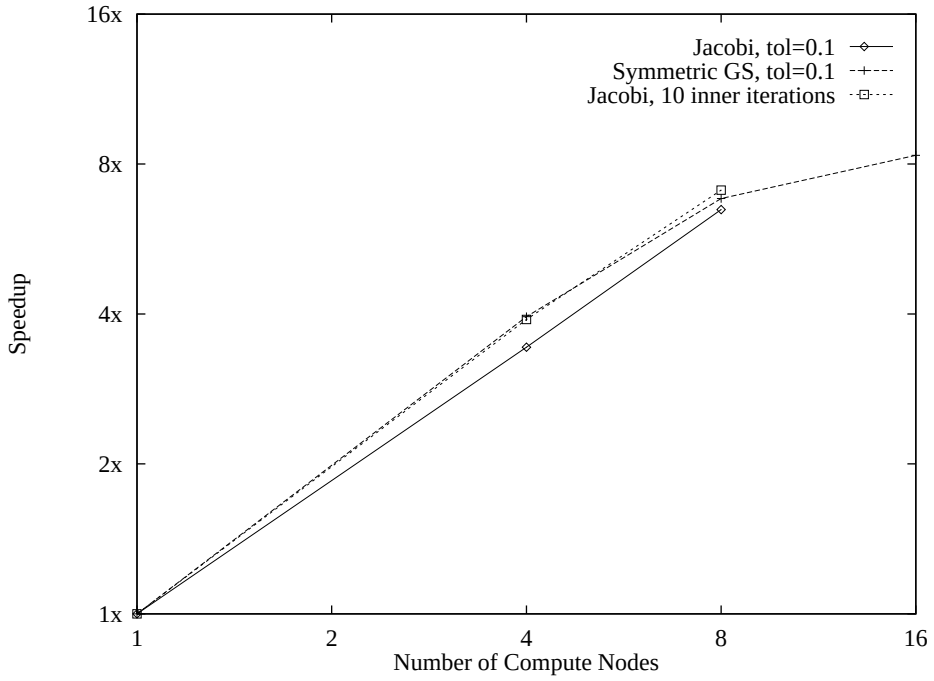
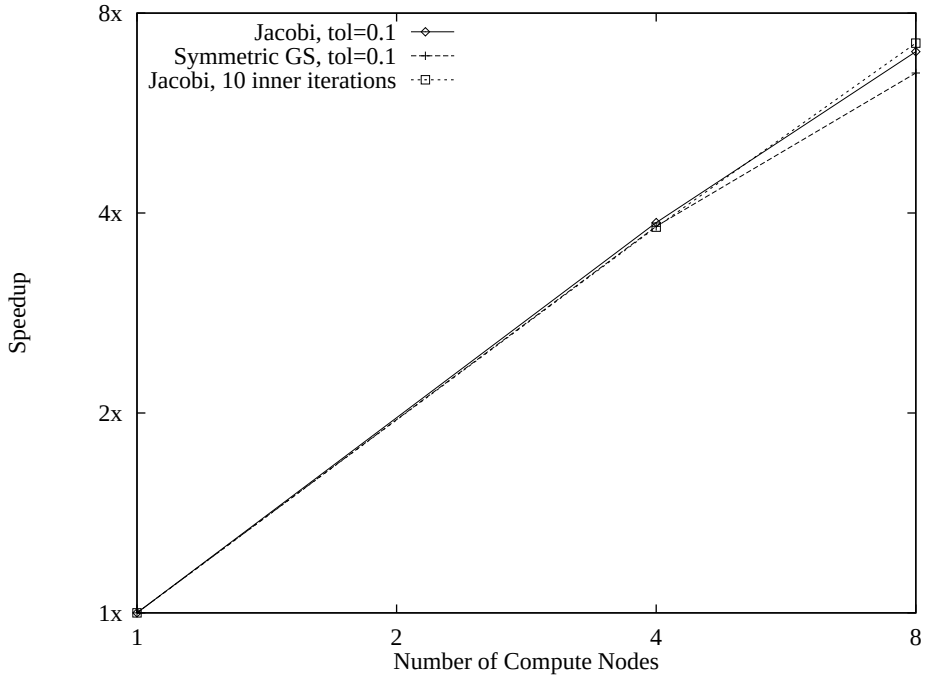


Figure 70: Speedup for the ONERA M6 wing on the Intel Paragon.

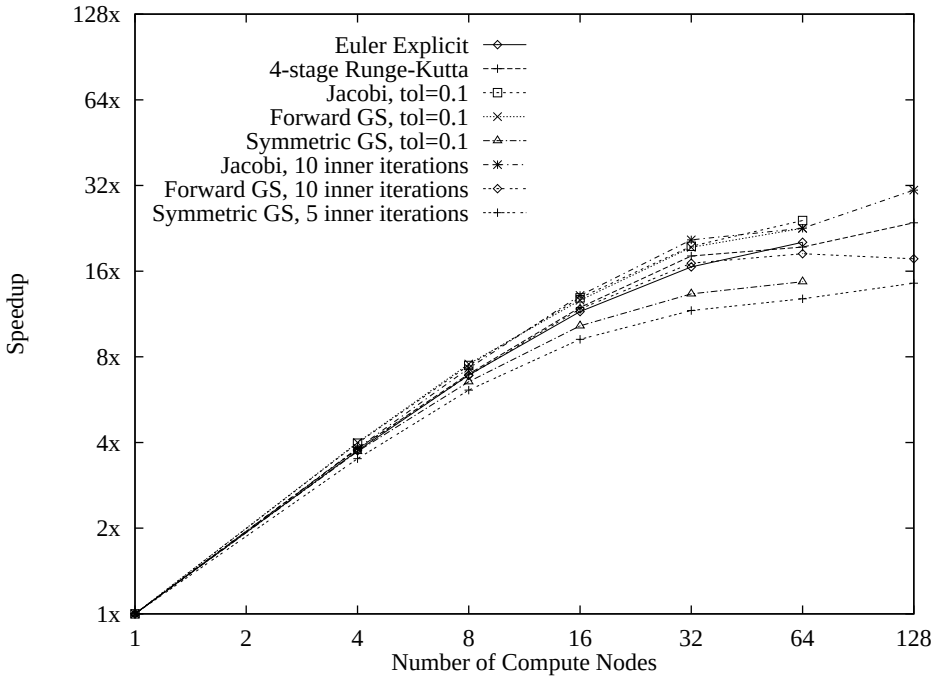


a) thin nodes

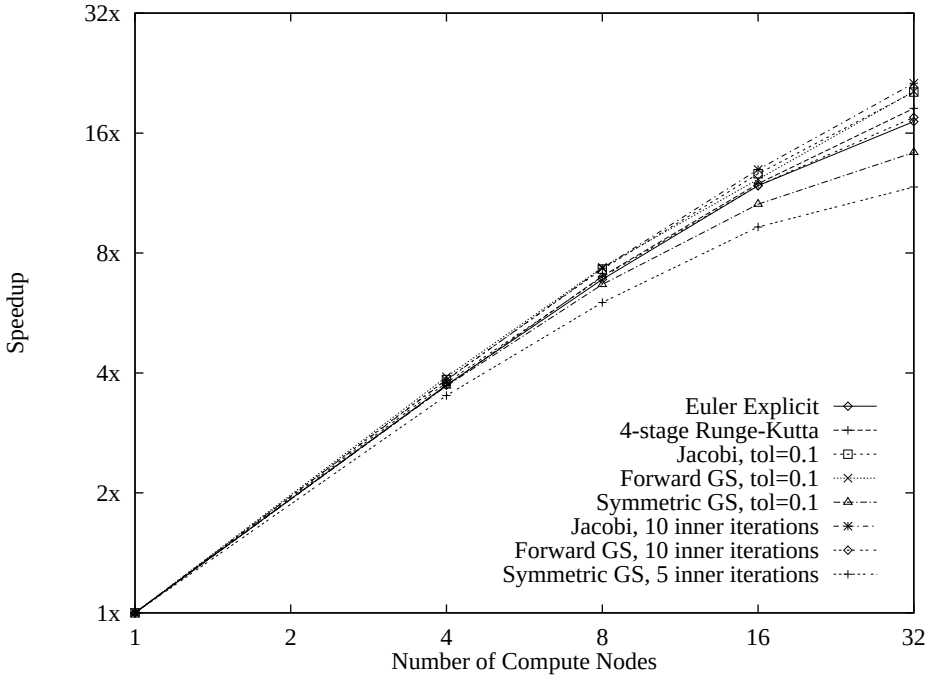


b) wide nodes

Figure 71: Speedup for the supersonic bump on the IBM SP-2.

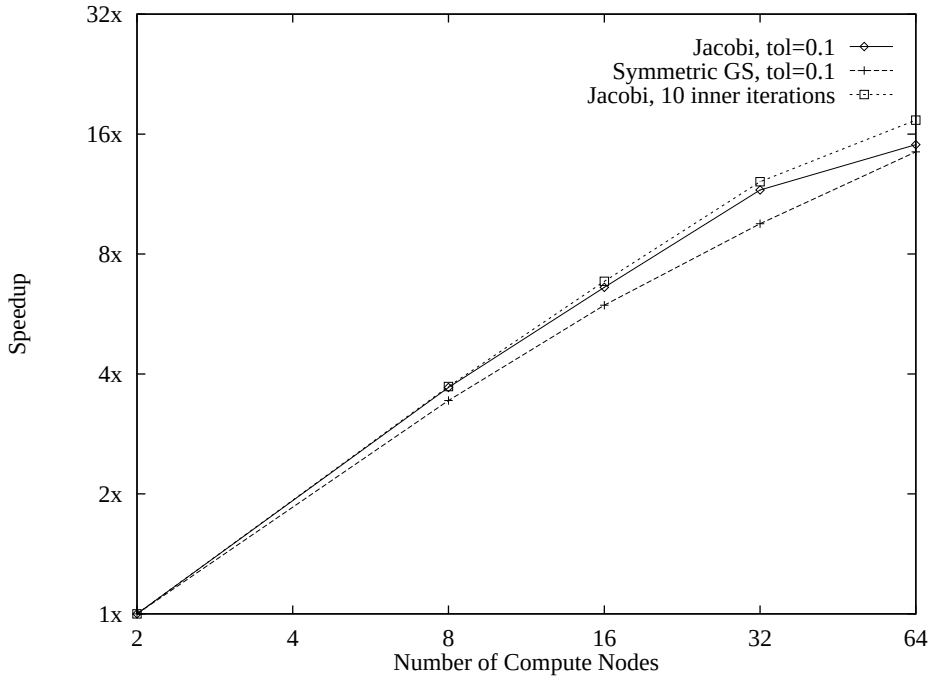


a) thin nodes

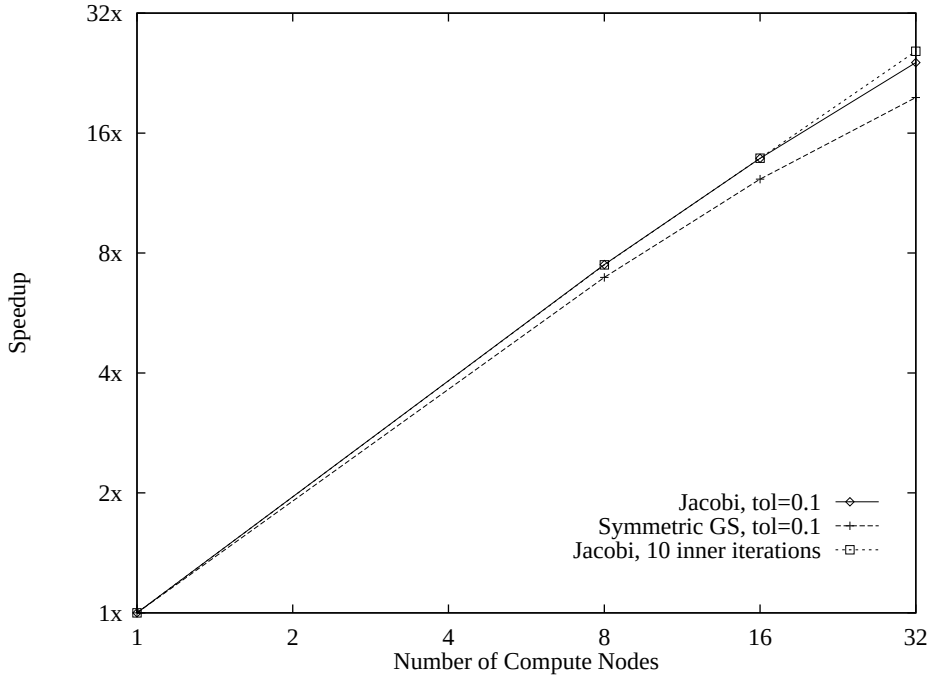


b) wide nodes

Figure 72: Speedup for the RAE 2822 airfoil on the IBM SP-2.

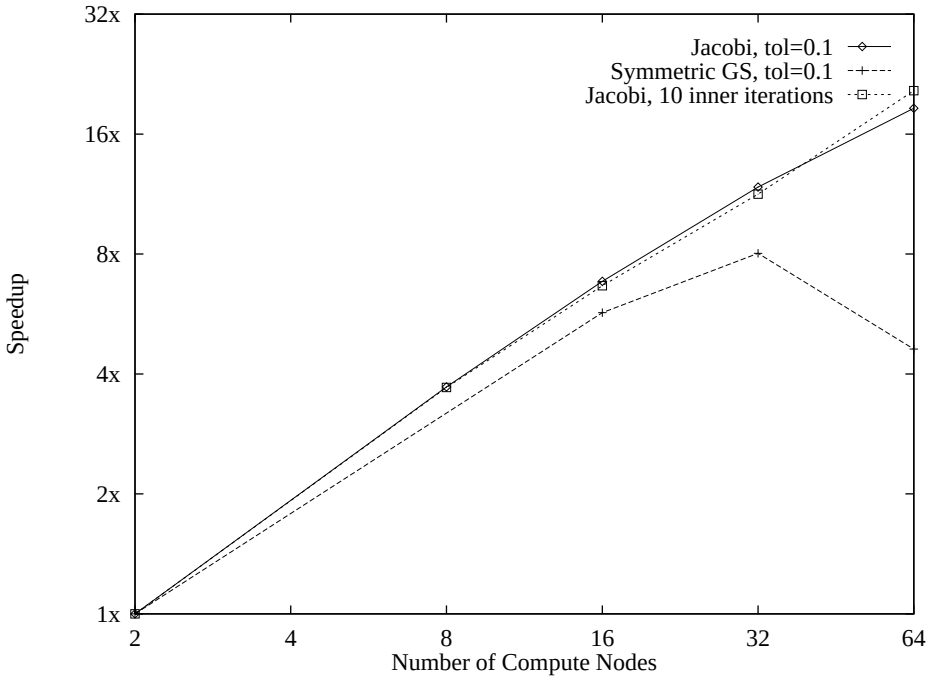


a) thin nodes

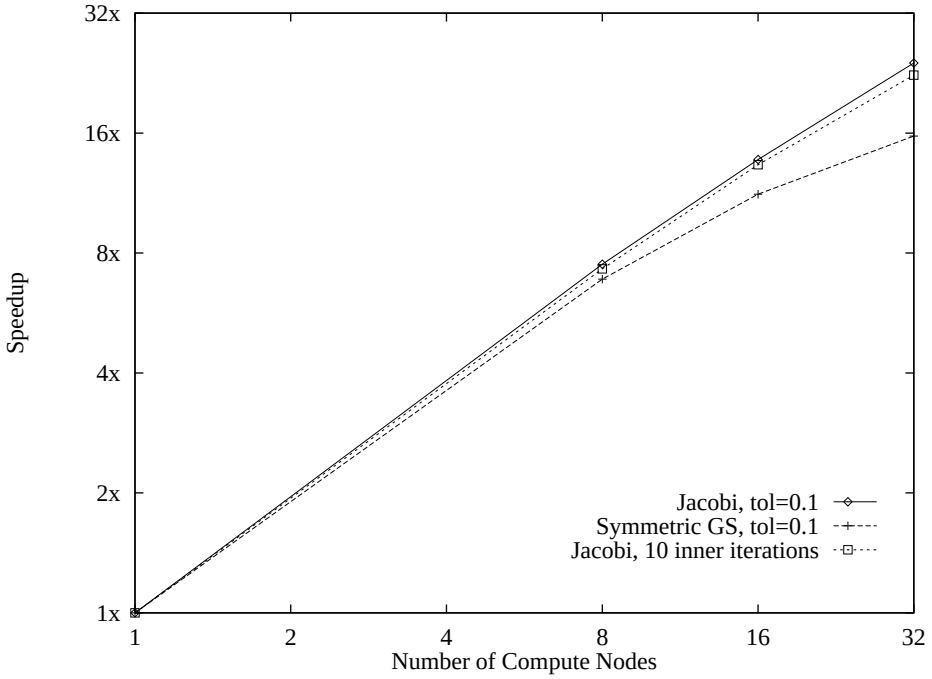


b) wide nodes

Figure 73: Speedup for the Hummel delta wing on the IBM SP-2.

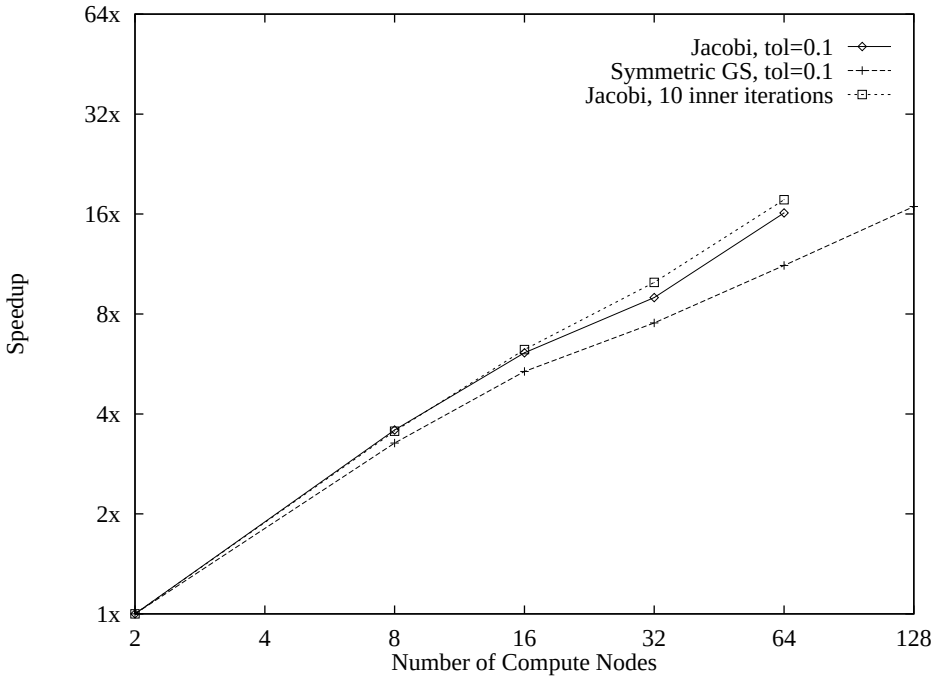


a) thin nodes

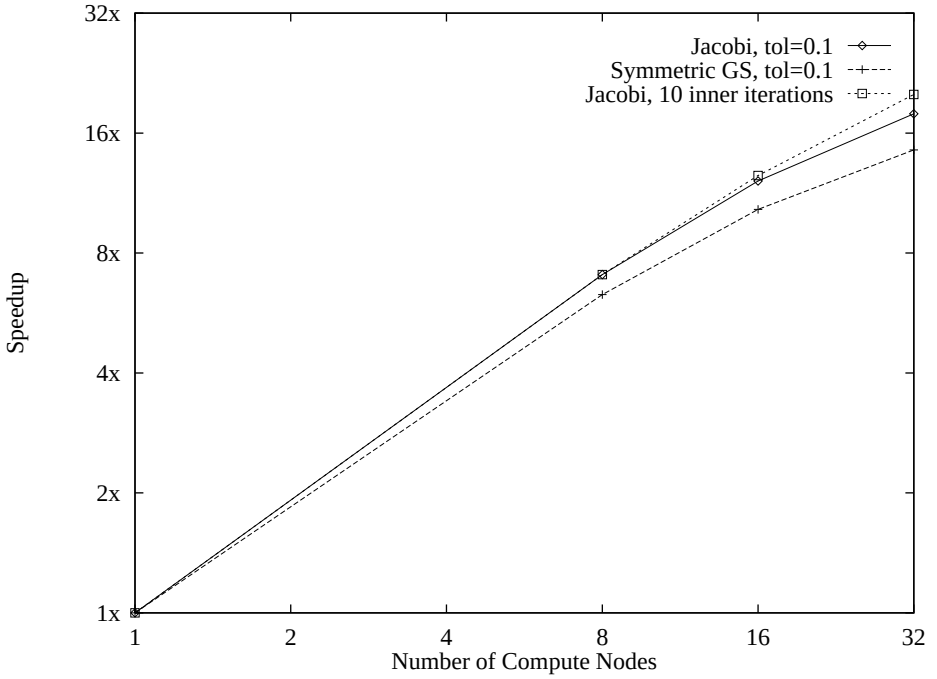


b) wide nodes

Figure 74: Speedup for the analytic forebody on the IBM SP-2.



a) thin nodes



b) wide nodes

Figure 75: Speedup for the ONERA M6 wing on the IBM SP-2.

Parallel Efficiency

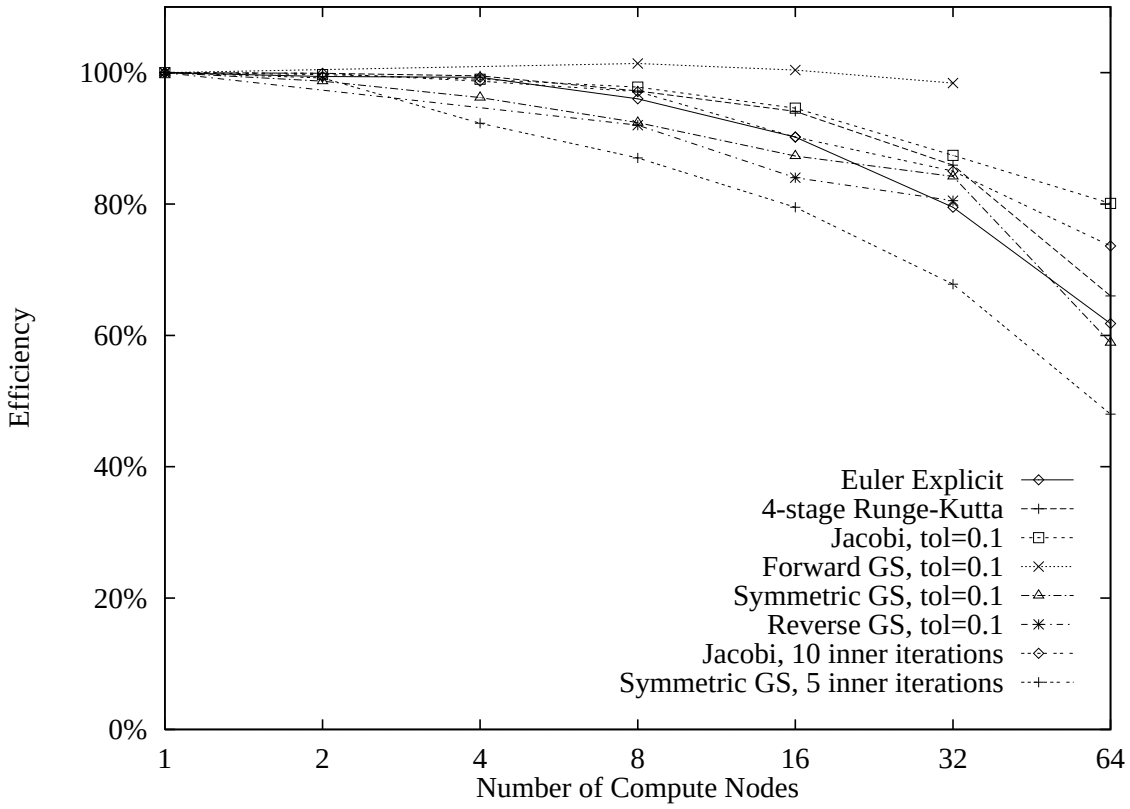


Figure 76: Parallel efficiency for the supersonic bump on the Intel Paragon.

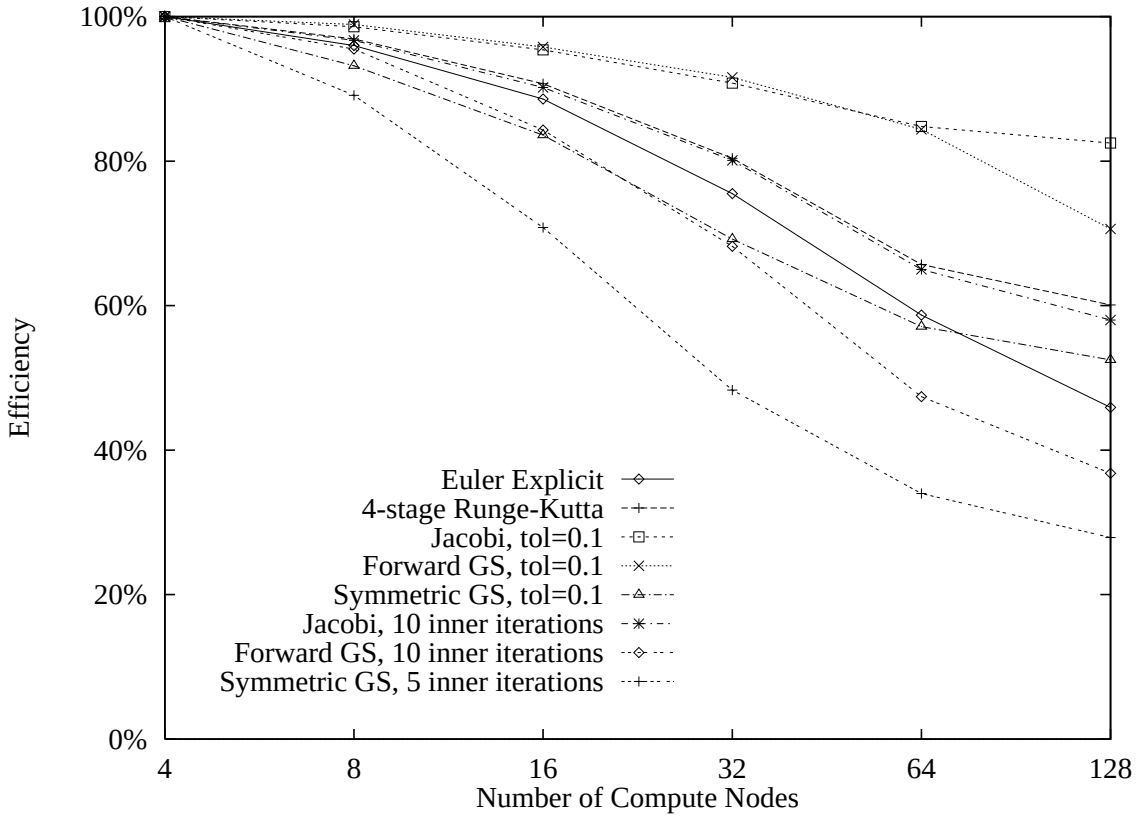


Figure 77: Parallel efficiency for the RAE 2822 airfoil on the Intel Paragon.

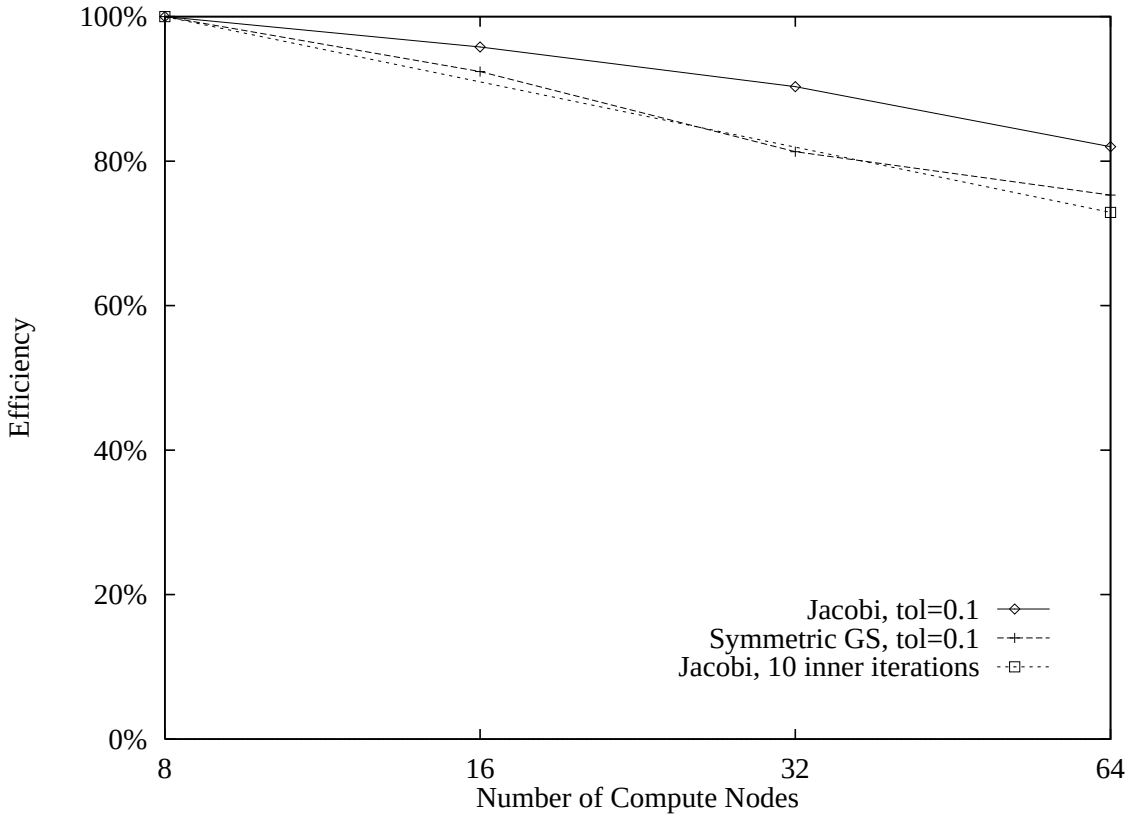


Figure 78: Parallel efficiency for the Hummel delta wing on the Intel Paragon.

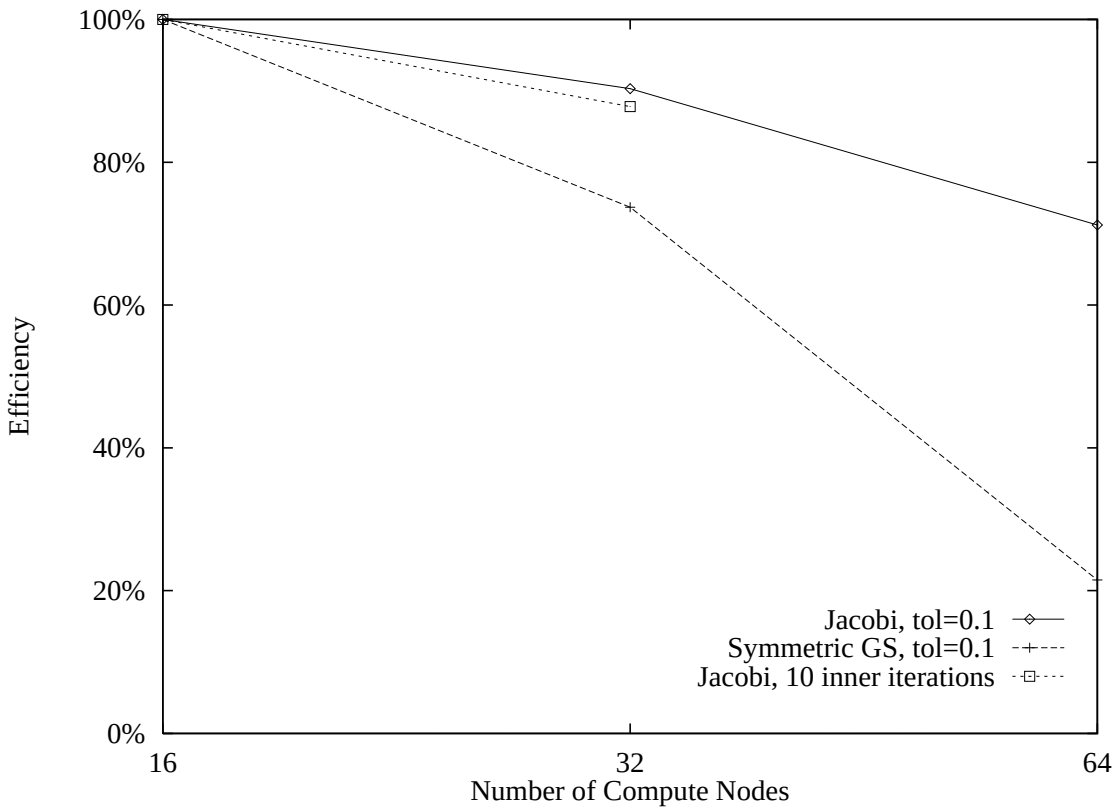


Figure 79: Parallel efficiency for the analytic forebody on the Intel Paragon.

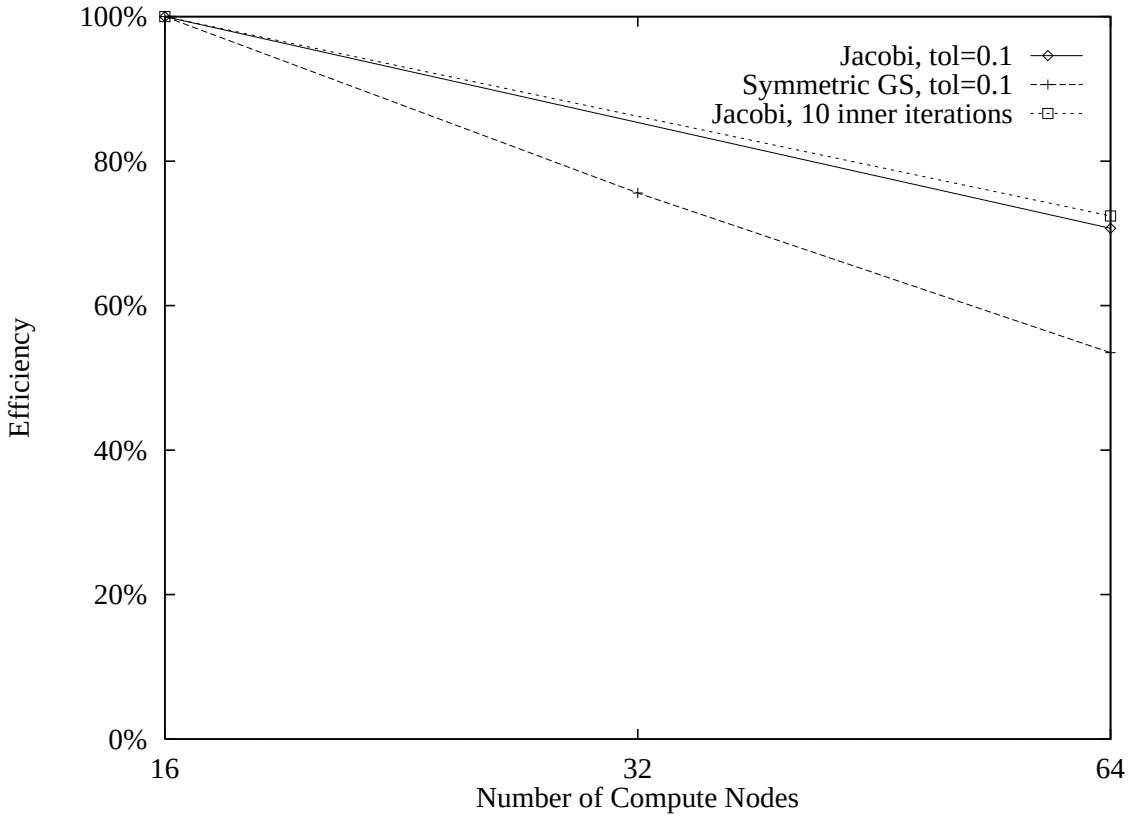
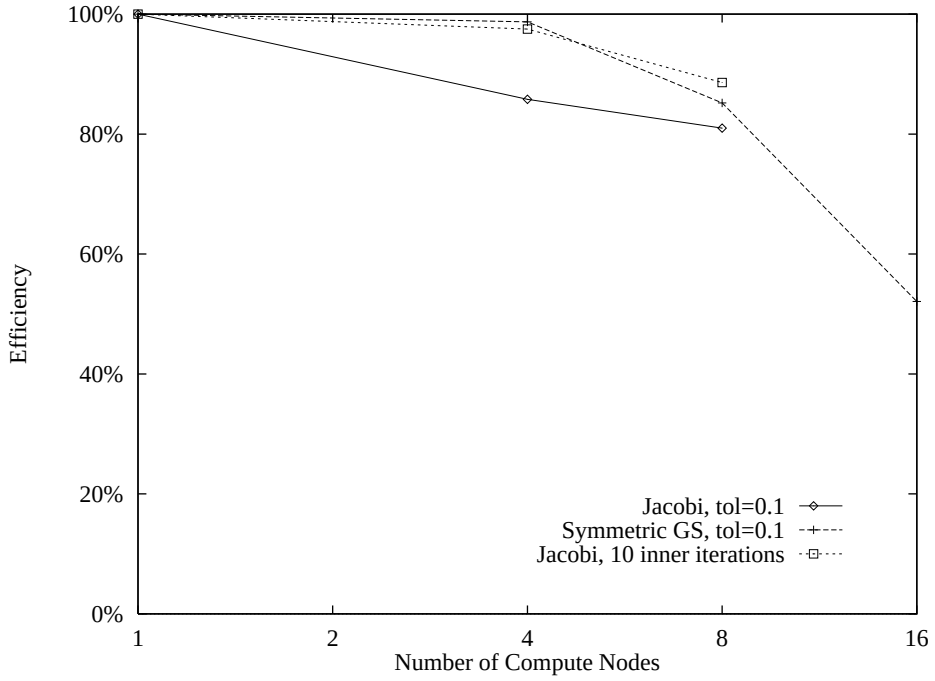
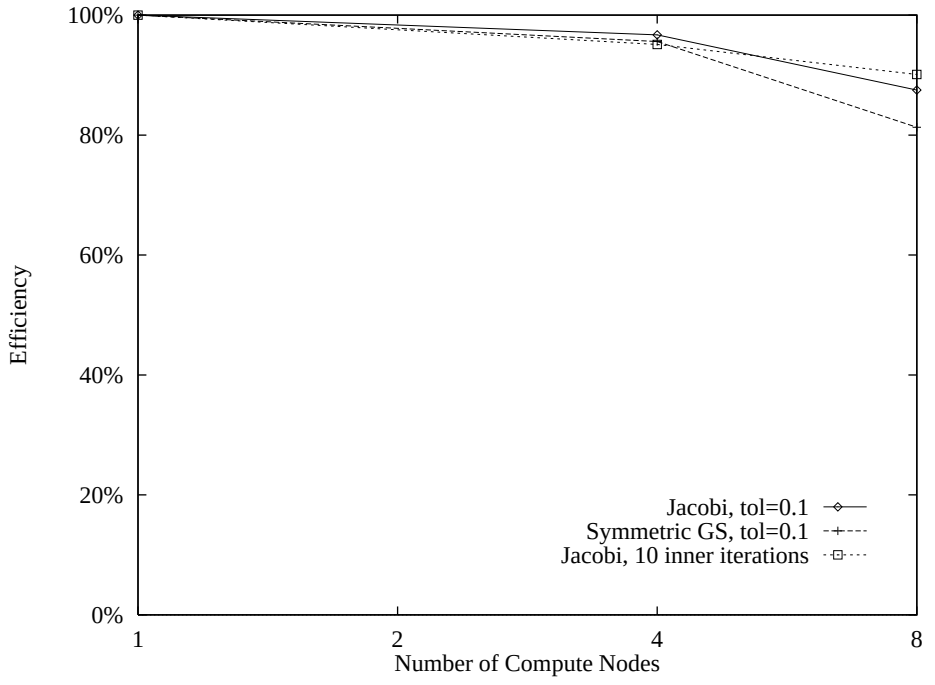


Figure 80: Parallel efficiency for the ONERA M6 wing on the Intel Paragon.

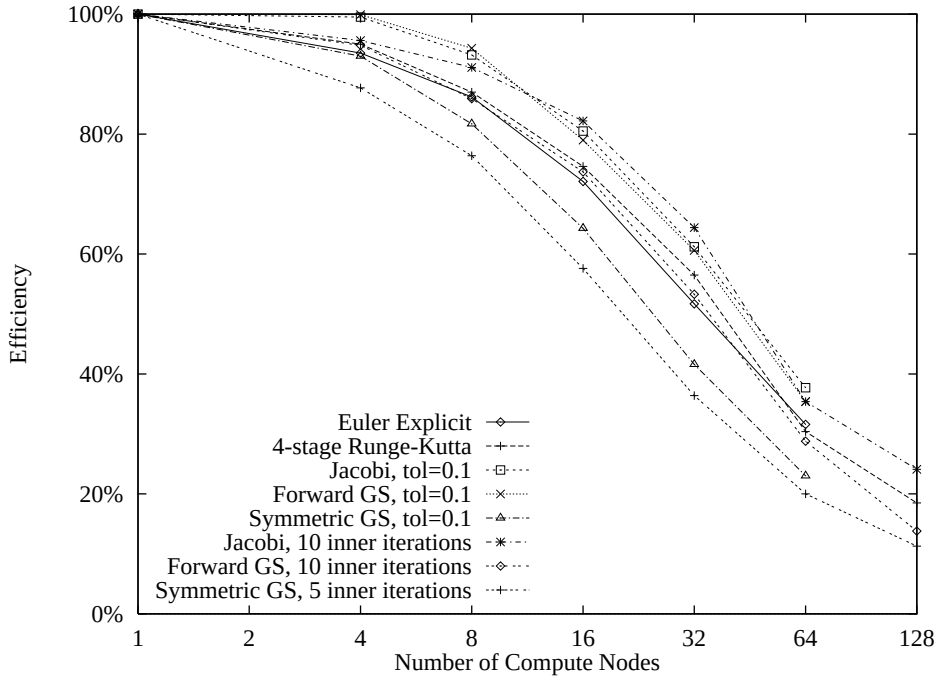


a) thin nodes

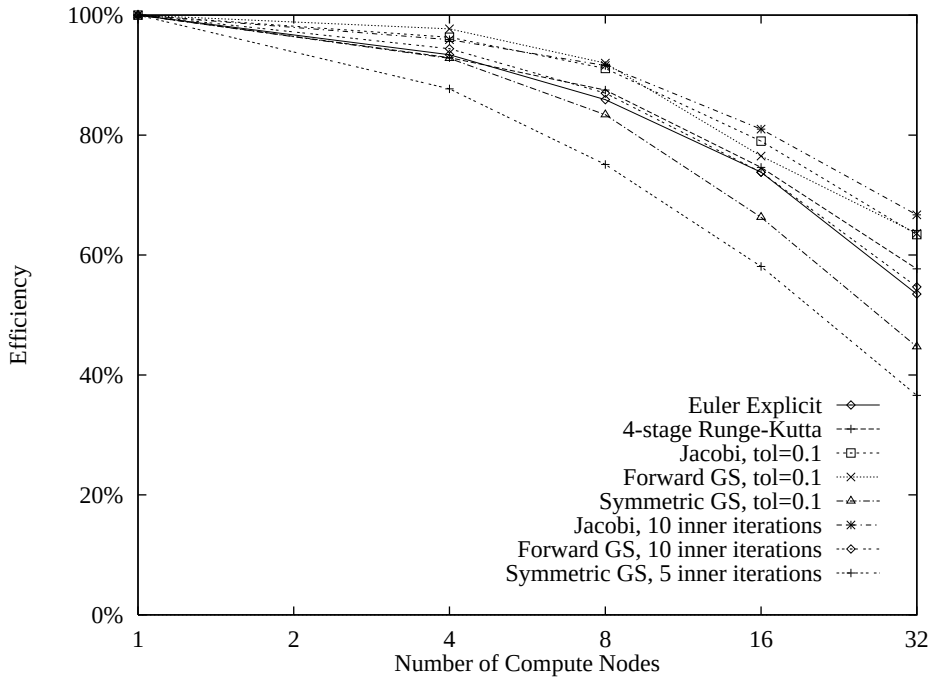


b) wide nodes

Figure 81: Parallel efficiency for the supersonic bump on the IBM SP-2.

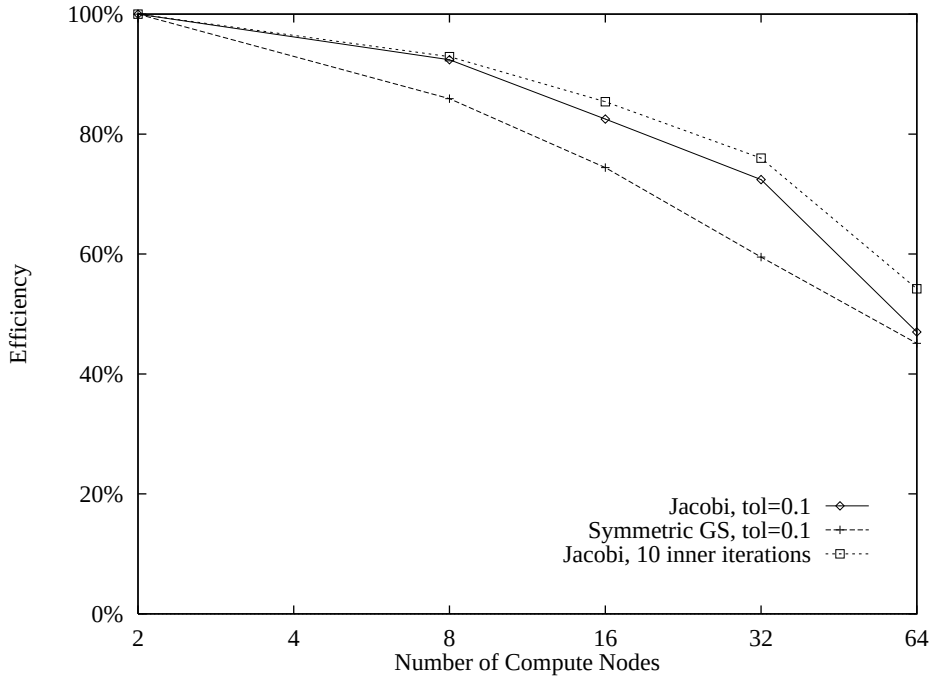


a) thin nodes

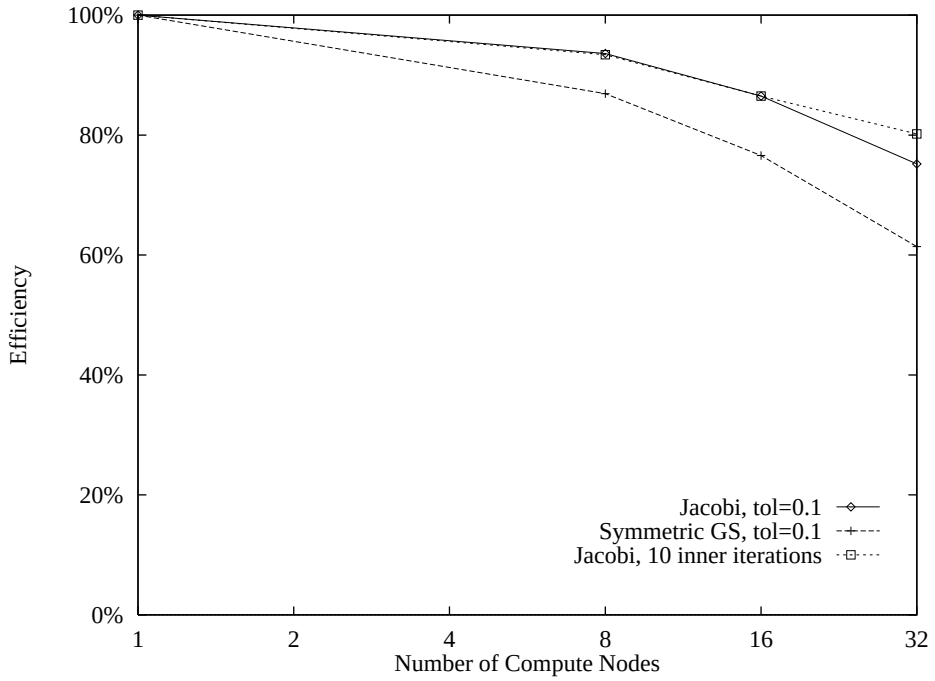


b) wide nodes

Figure 82: Parallel efficiency for the RAE 2822 airfoil on the IBM SP-2.

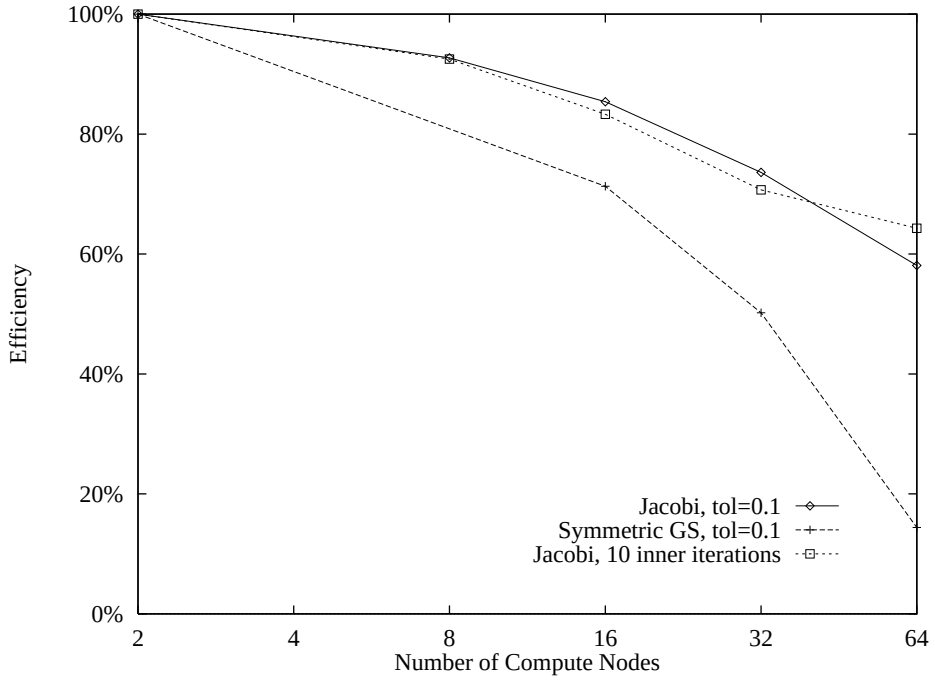


a) thin nodes

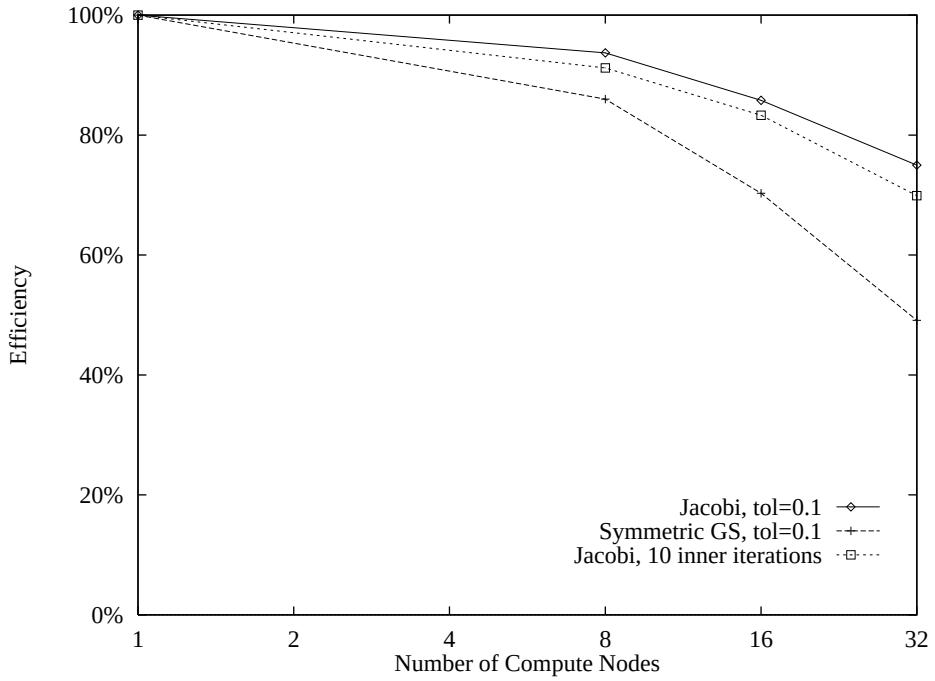


b) wide nodes

Figure 83: Parallel efficiency for the Hummel delta wing on the IBM SP-2.

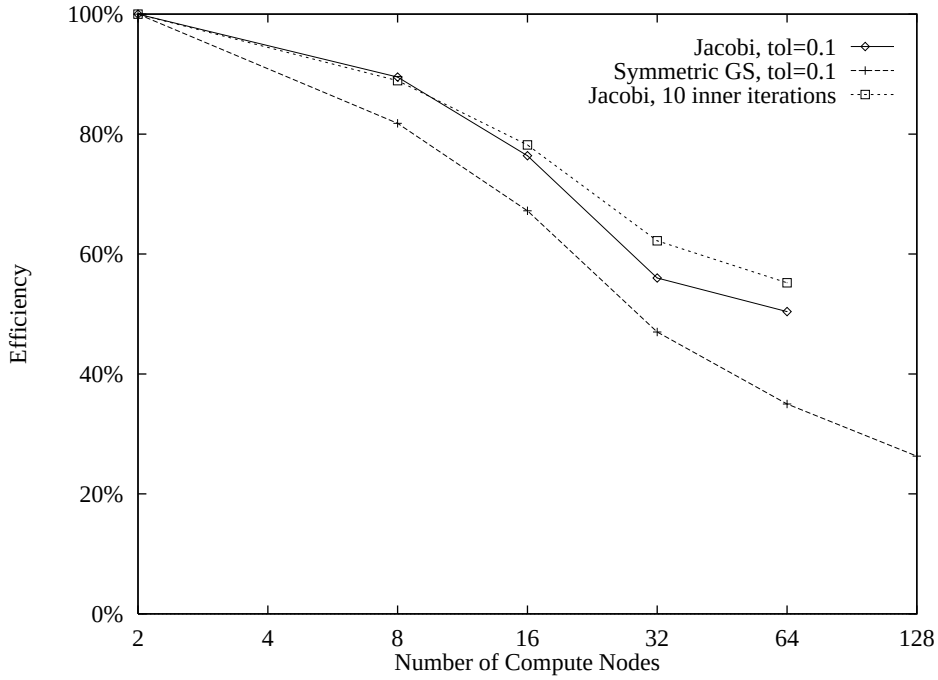


a) thin nodes

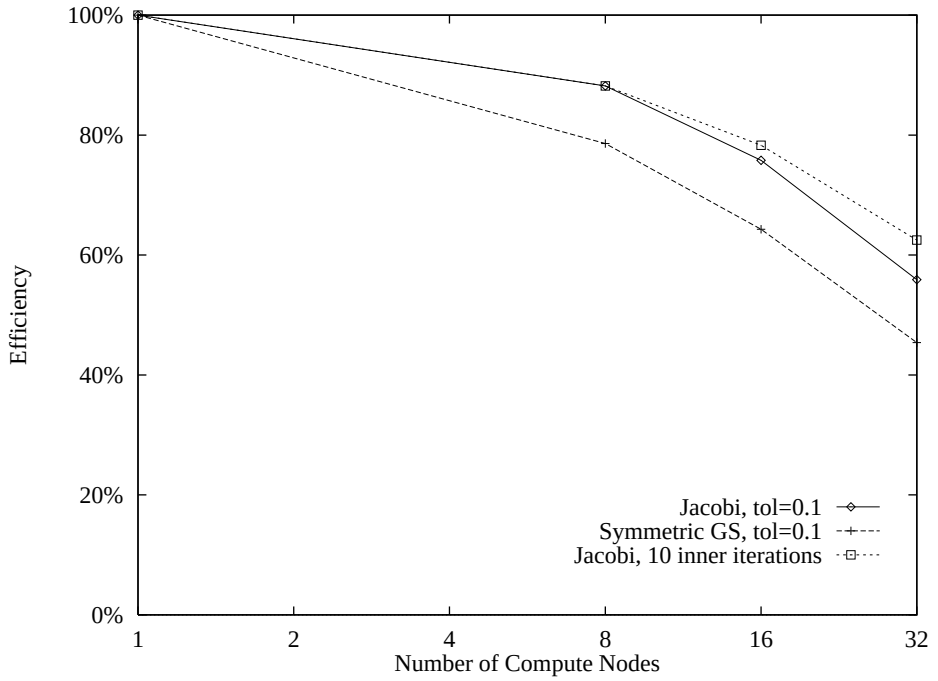


b) wide nodes

Figure 84: Parallel efficiency for the analytic forebody on the IBM SP-2.



a) thin nodes



b) wide nodes

Figure 85: Parallel efficiency for the ONERA M6 wing on the IBM SP-2.

Appendix B: Summary of Second-order Results

Time

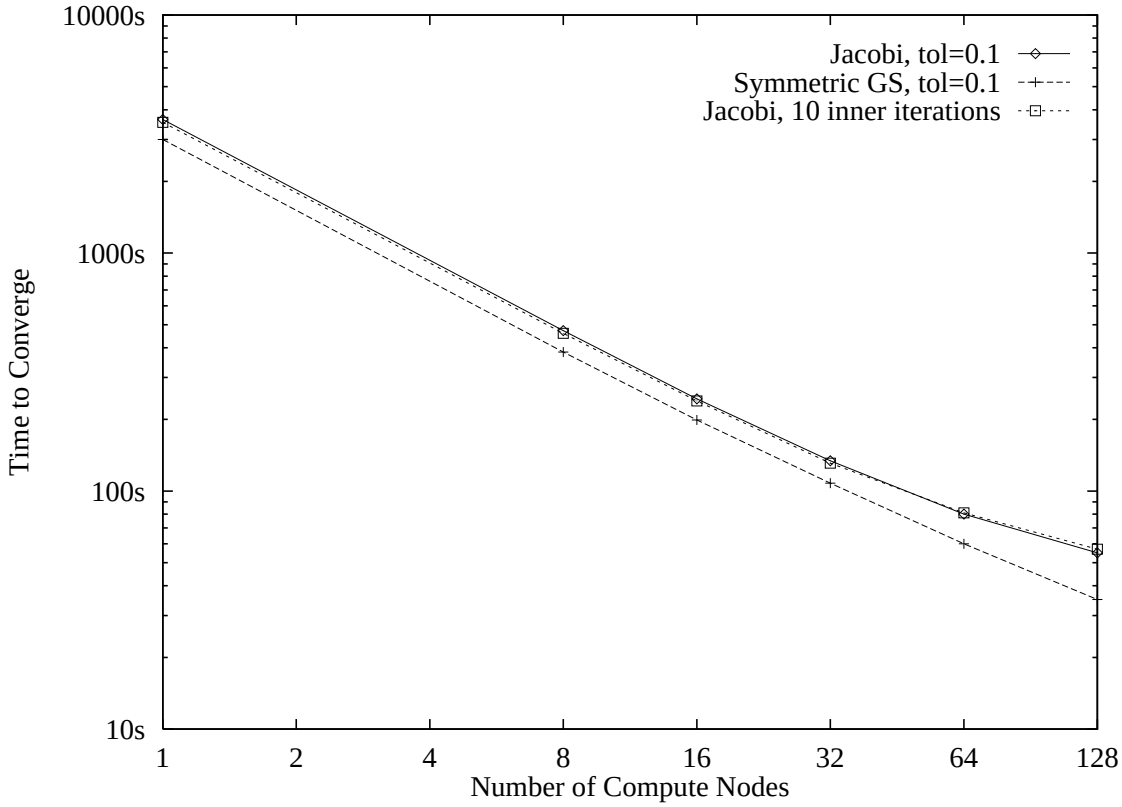


Figure 86: Convergence time for the supersonic bump on the Intel Paragon.

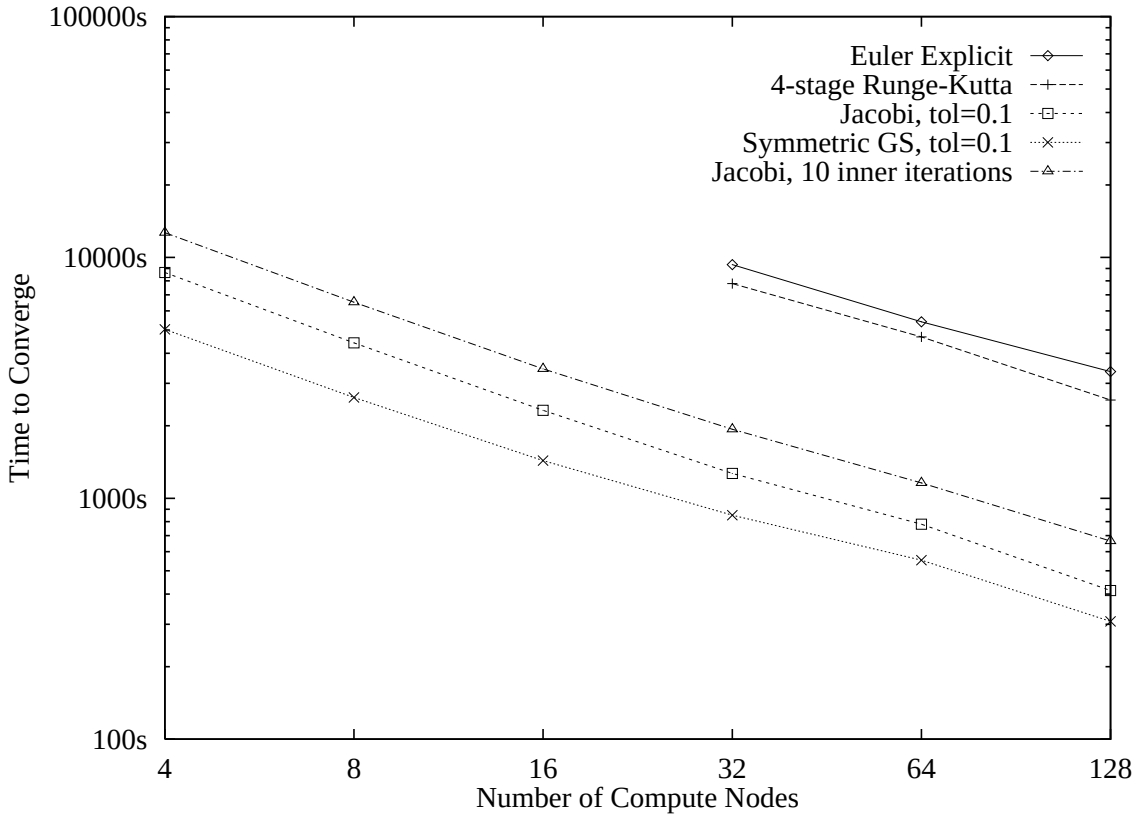


Figure 87: Convergence time for the RAE 2822 airfoil on the Intel Paragon.

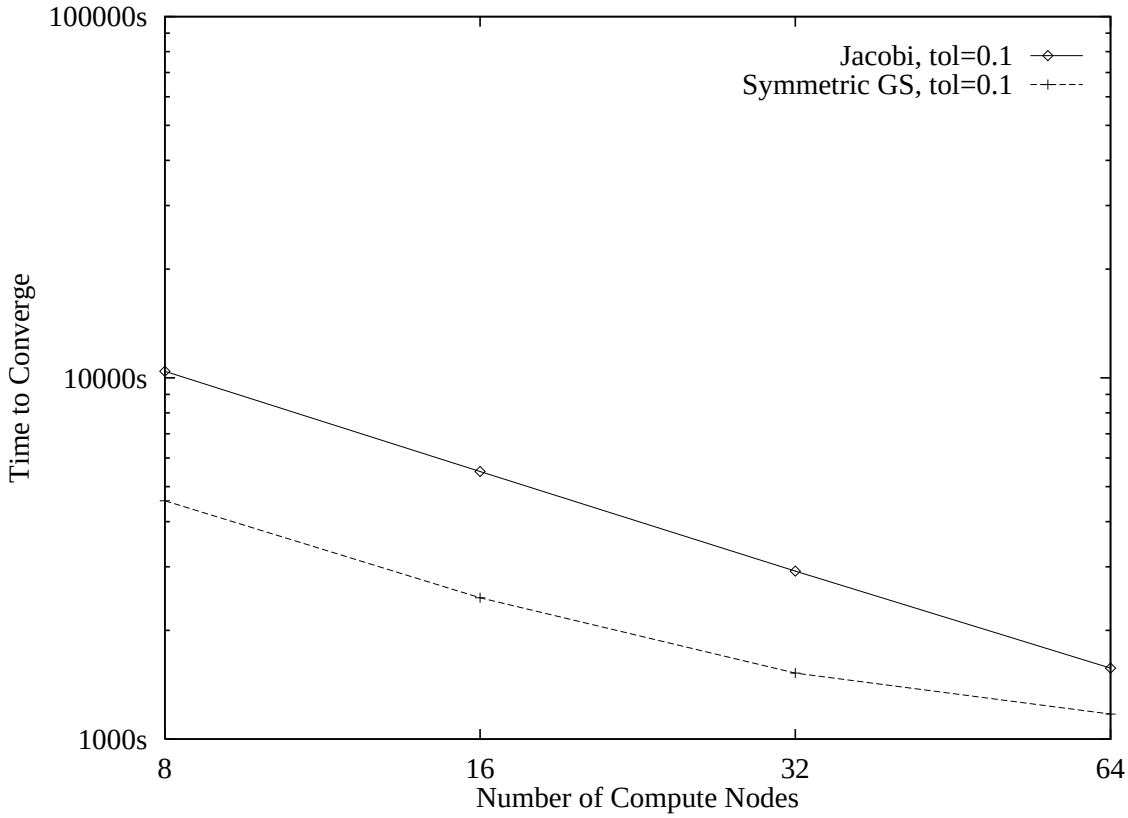


Figure 88: Convergence time for the Hummel delta wing on the Intel Paragon.

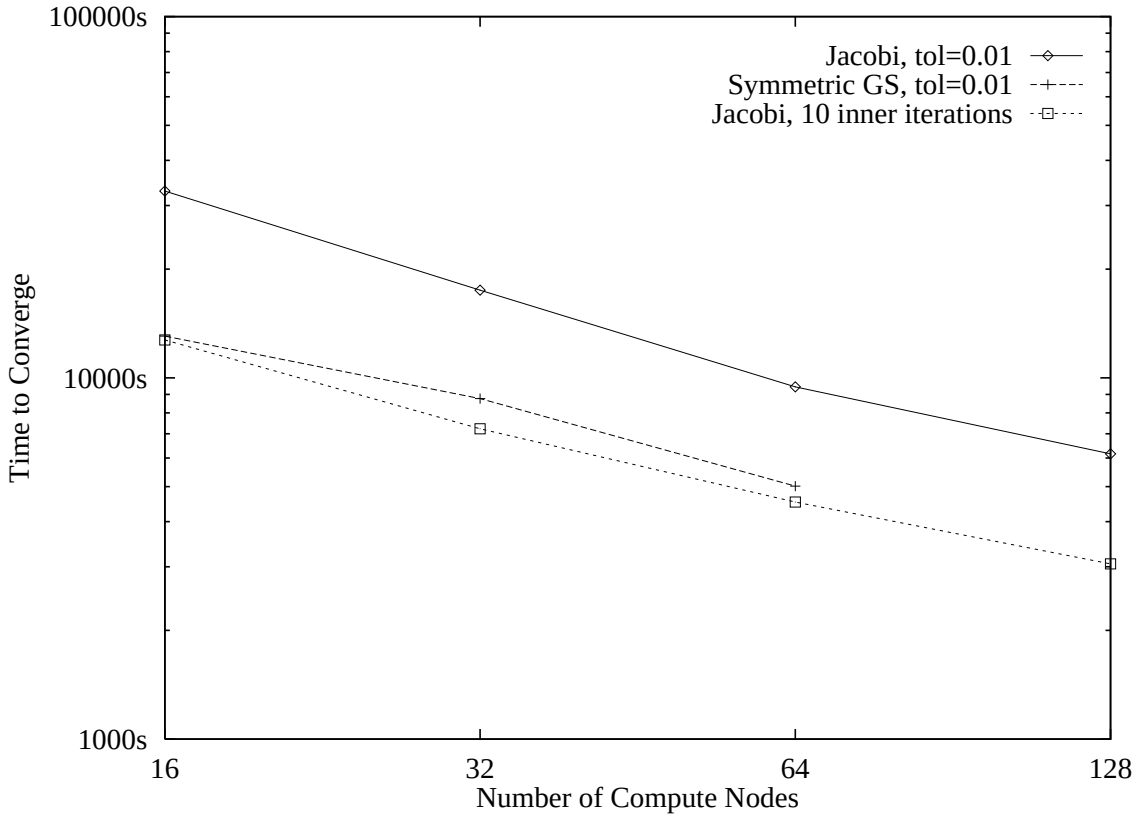


Figure 89: Convergence time for the analytic forebody on the Intel Paragon.

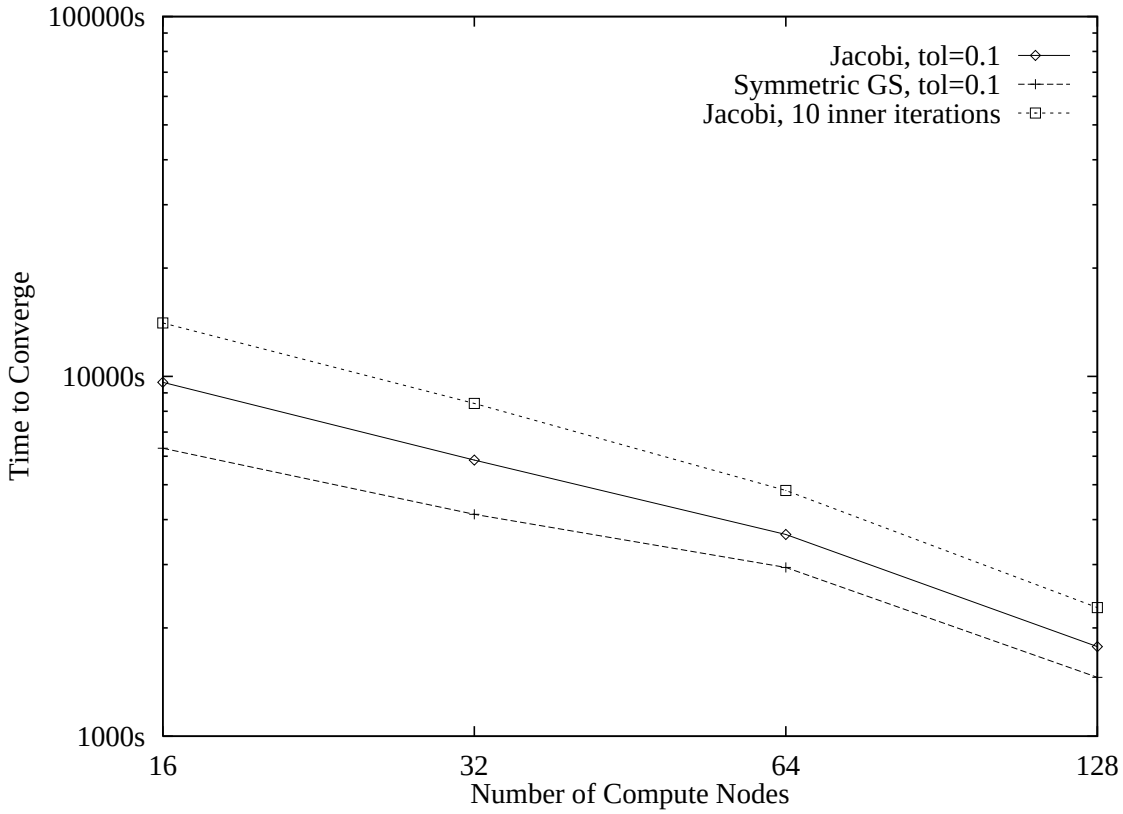
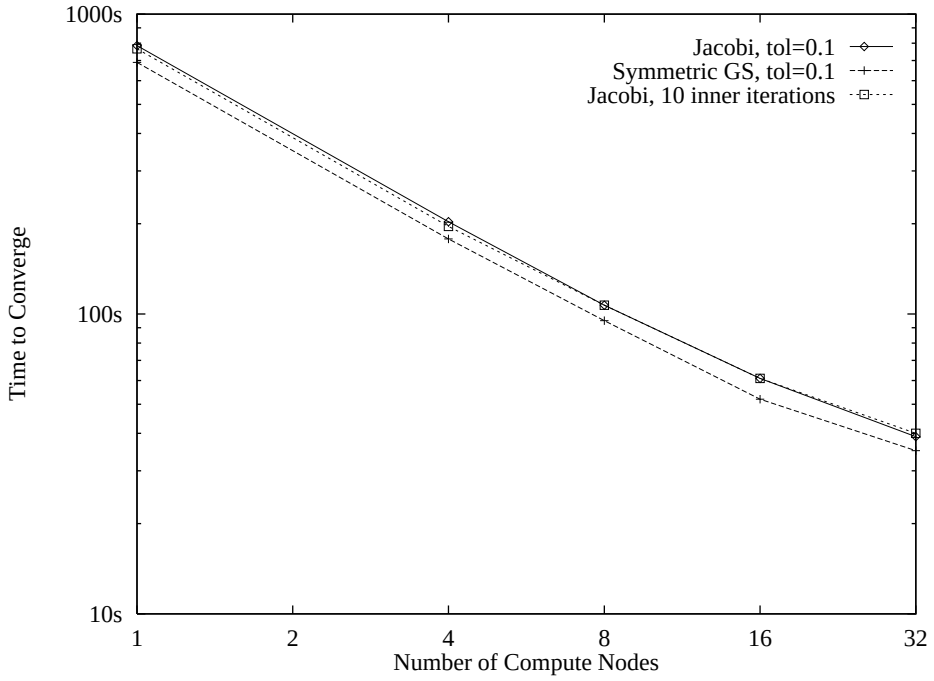
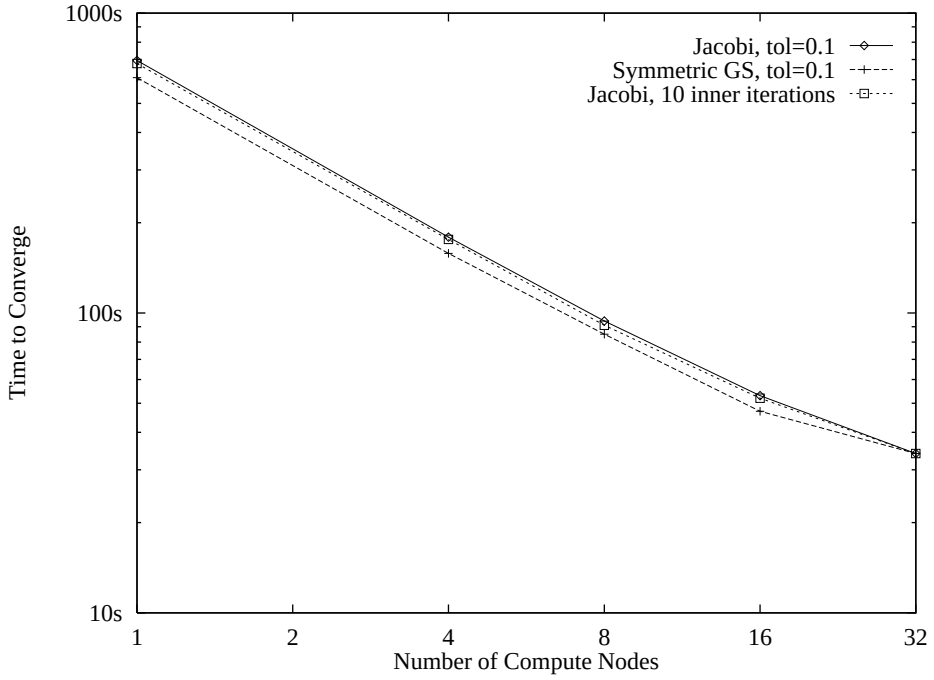


Figure 90: Convergence time for the ONERA M6 wing on the Intel Paragon.

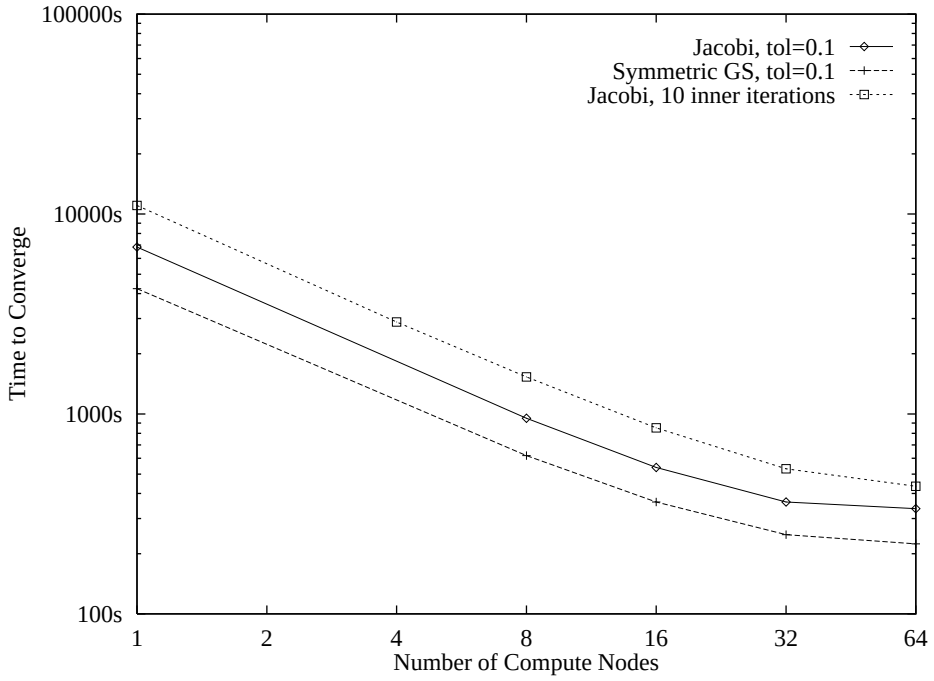


a) thin nodes

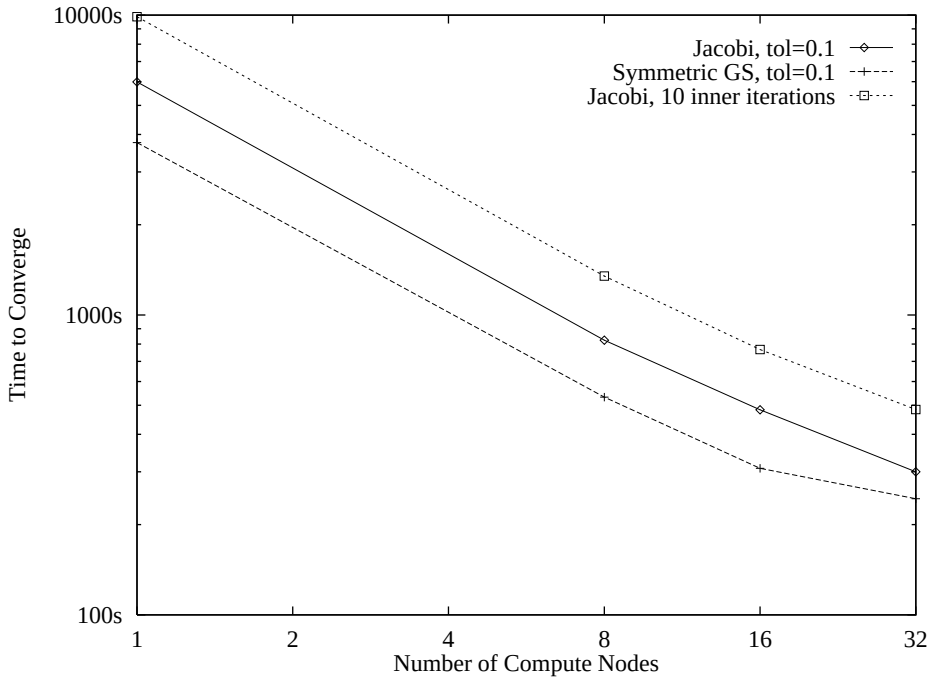


b) wide nodes

Figure 91: Convergence time for the supersonic bump on the IBM SP-2.

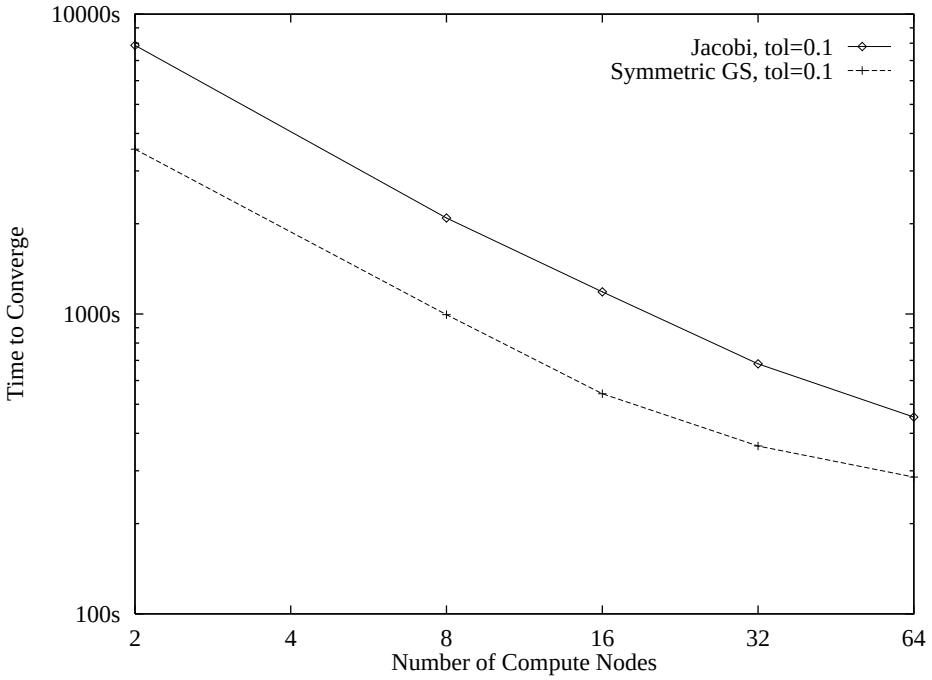


a) thin nodes

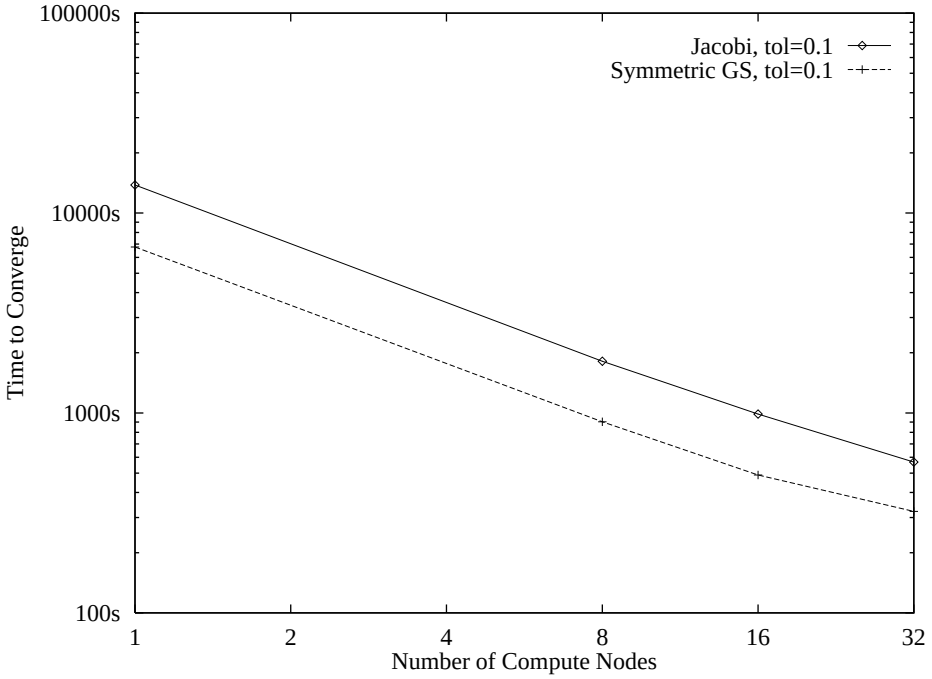


b) wide nodes

Figure 92: Convergence time for the RAE 2822 airfoil on the IBM SP-2.

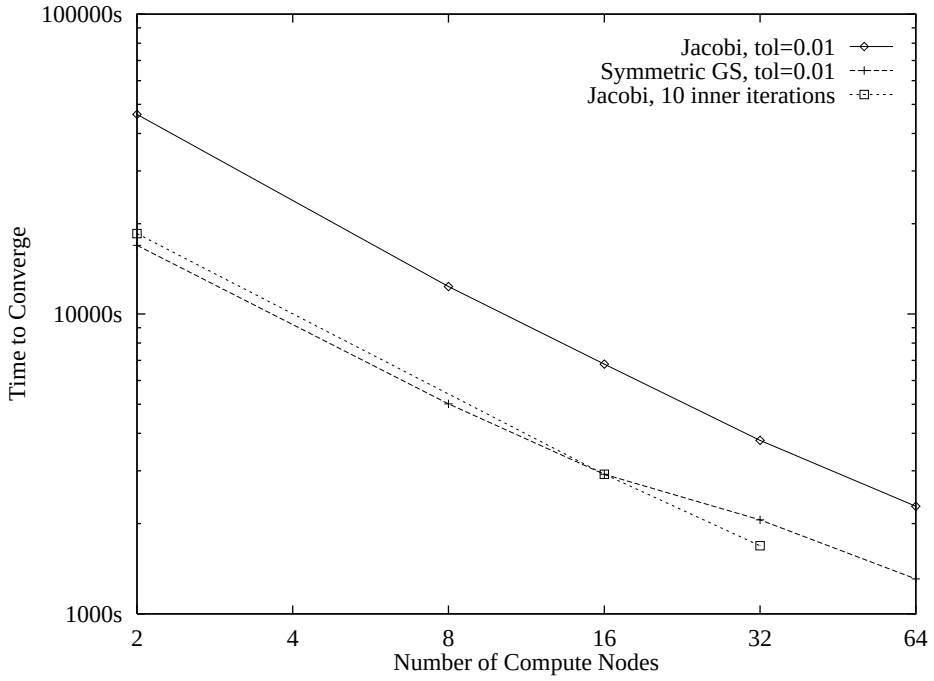


a) thin nodes

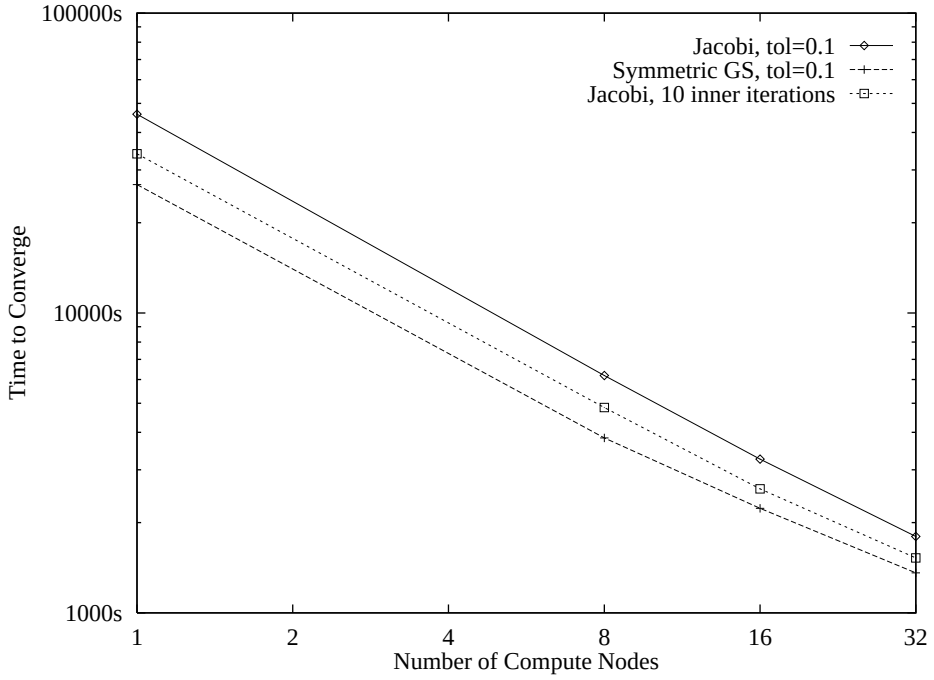


b) wide nodes

Figure 93: Convergence time for the Hummel delta wing on the IBM SP-2.

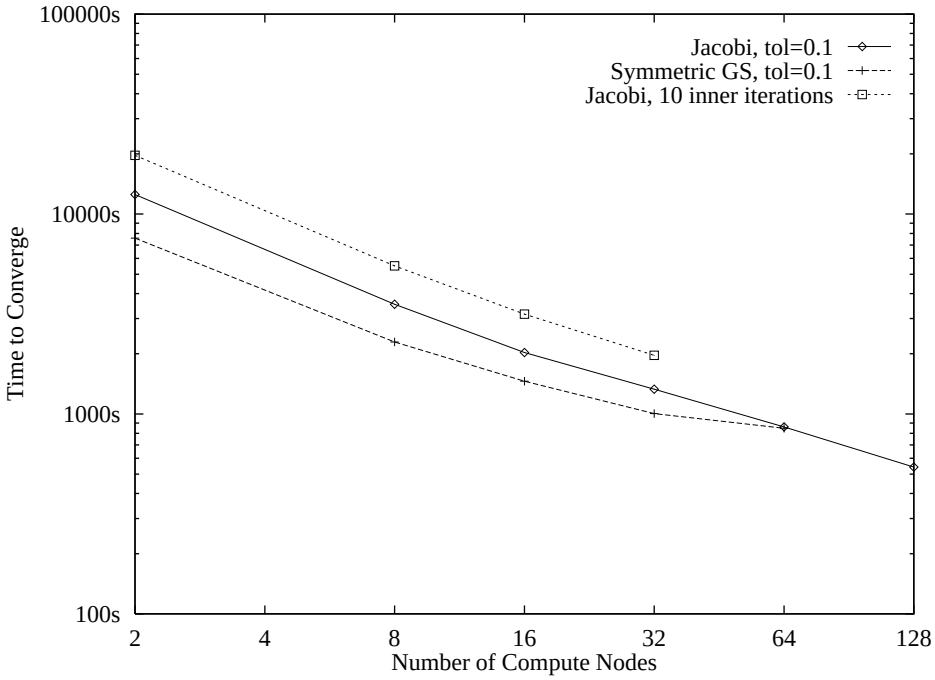


a) thin nodes

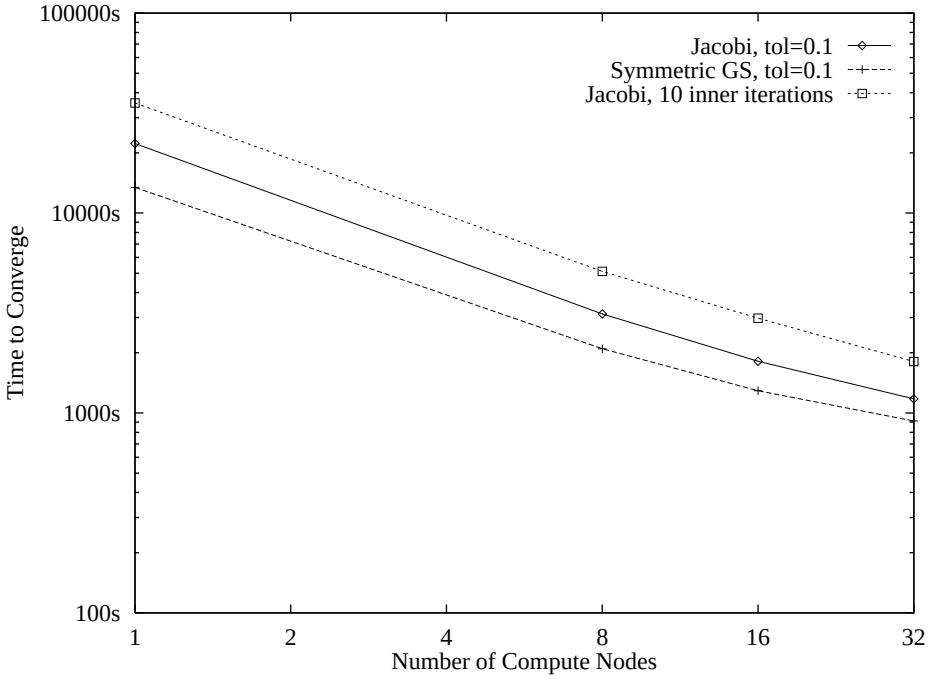


b) wide nodes

Figure 94: Convergence time for the analytic forebody on the IBM SP-2.



a) thin nodes



b) wide nodes

Figure 95: Convergence time for the ONERA M6 wing on the IBM SP-2.

Speedup

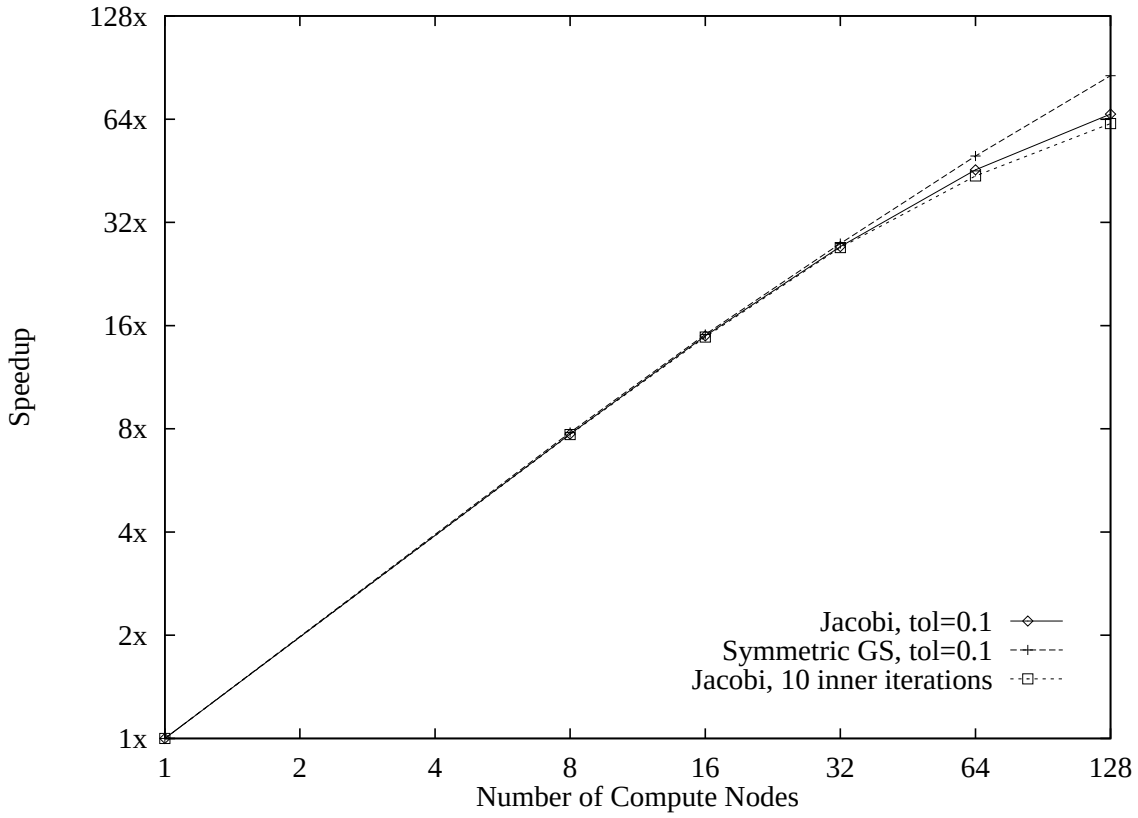


Figure 96: Speedup for the supersonic bump on the Intel Paragon.

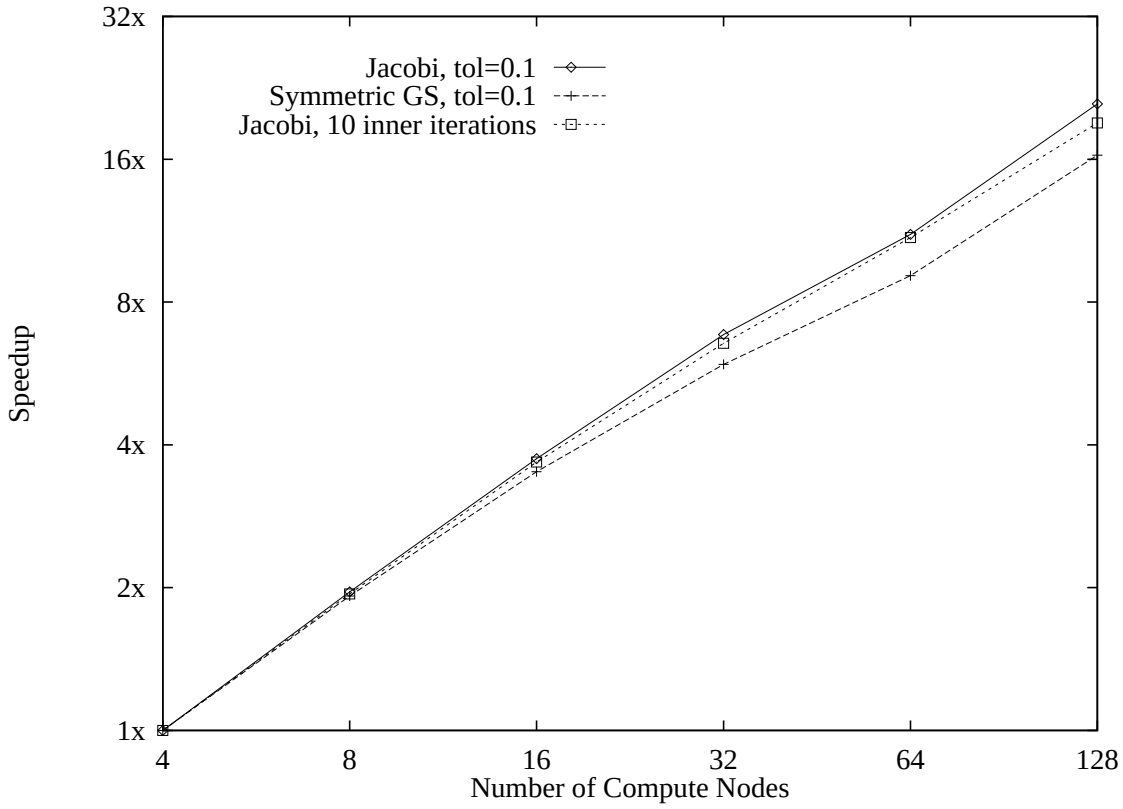


Figure 97: Speedup for the RAE 2822 airfoil on the Intel Paragon.

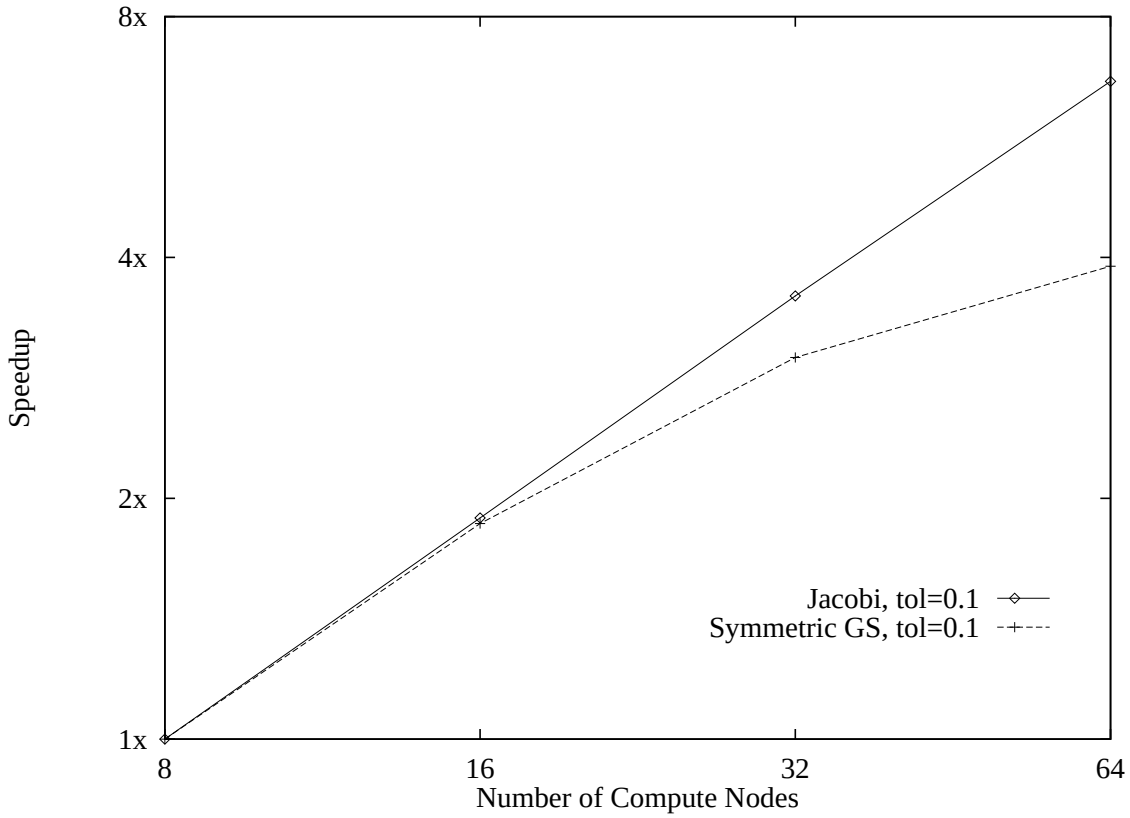


Figure 98: Speedup for the Hummel delta wing on the Intel Paragon.

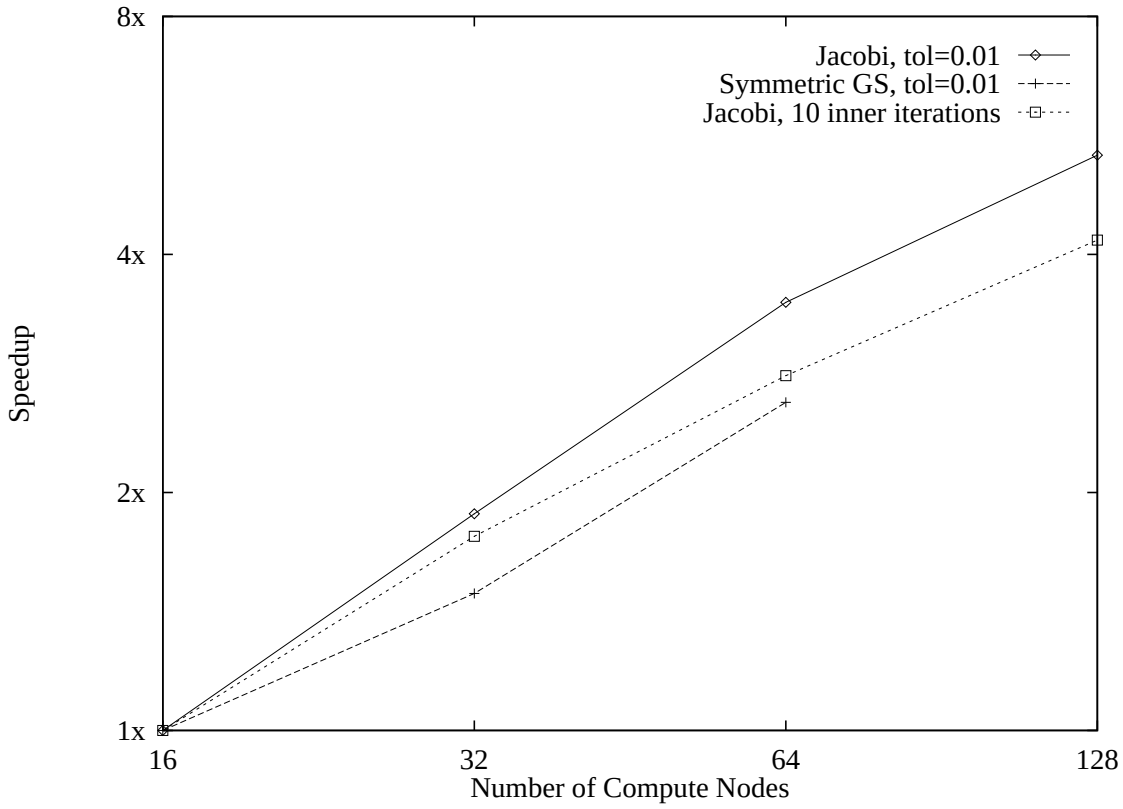


Figure 99: Speedup for the analytic forebody on the Intel Paragon.

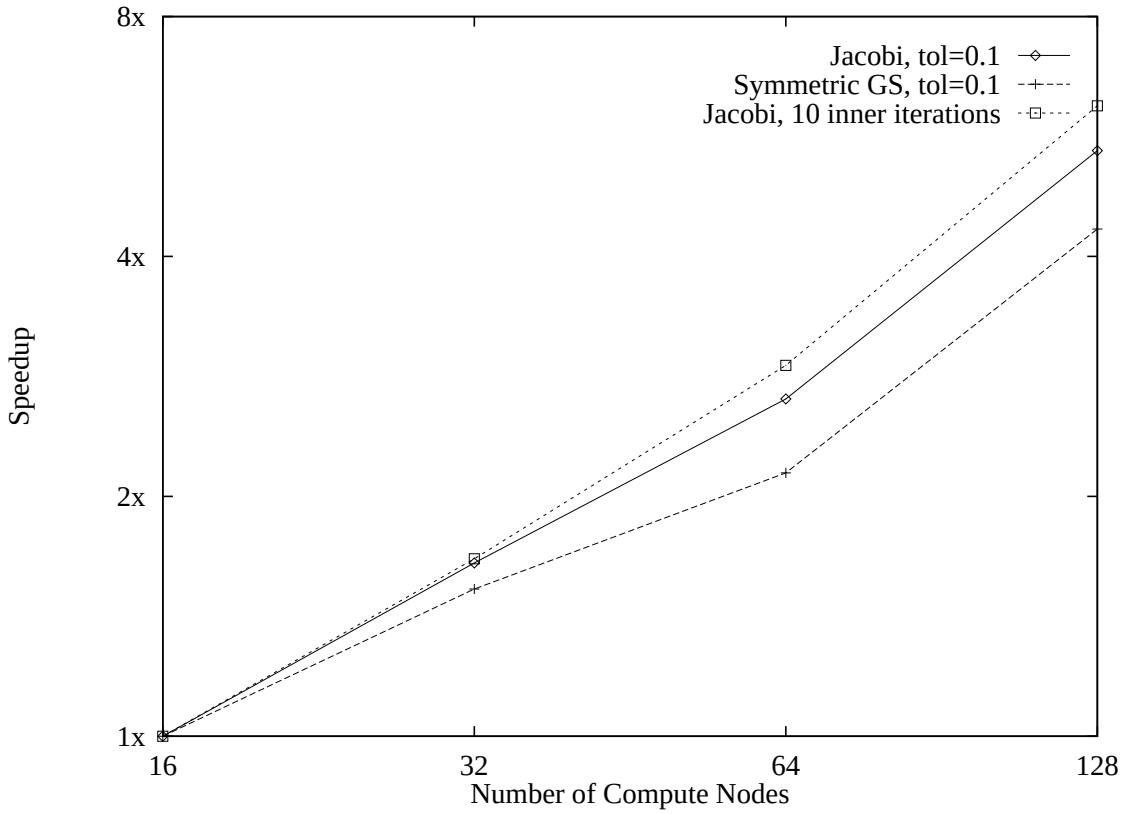
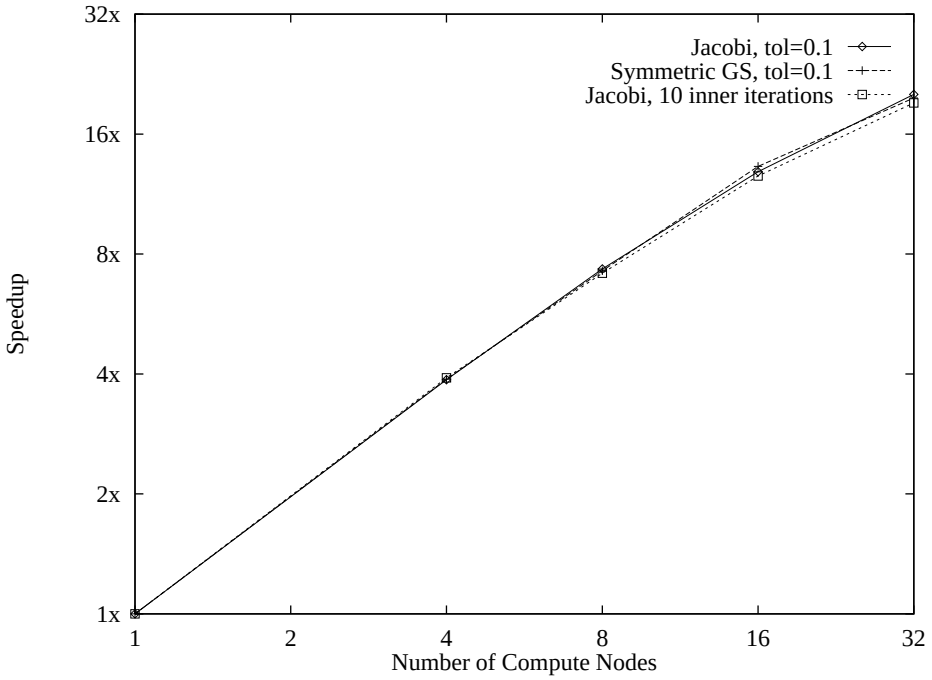
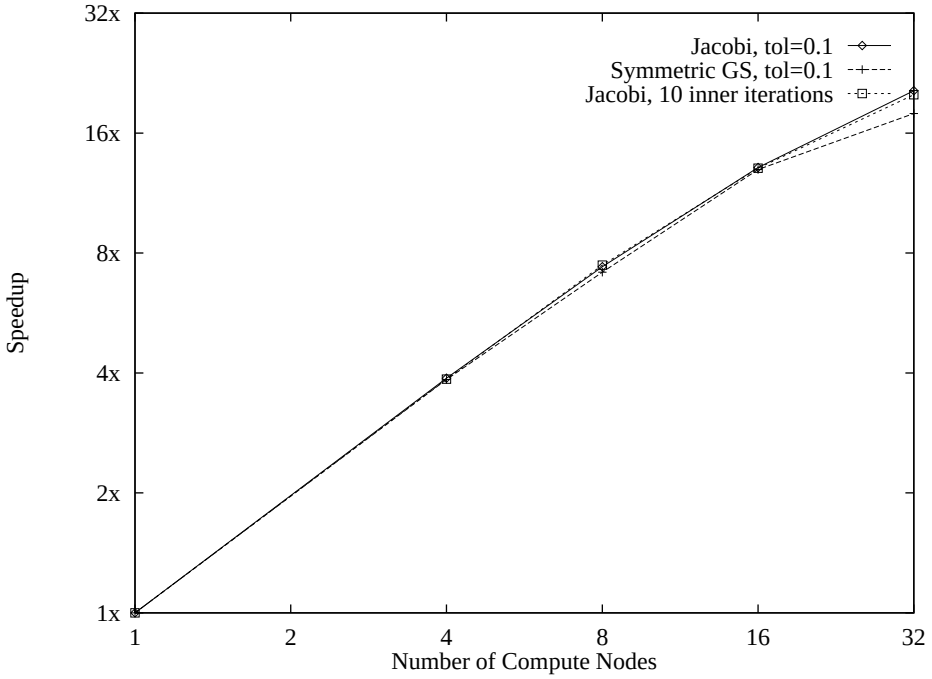


Figure 100: Speedup for the ONERA M6 wing on the Intel Paragon.

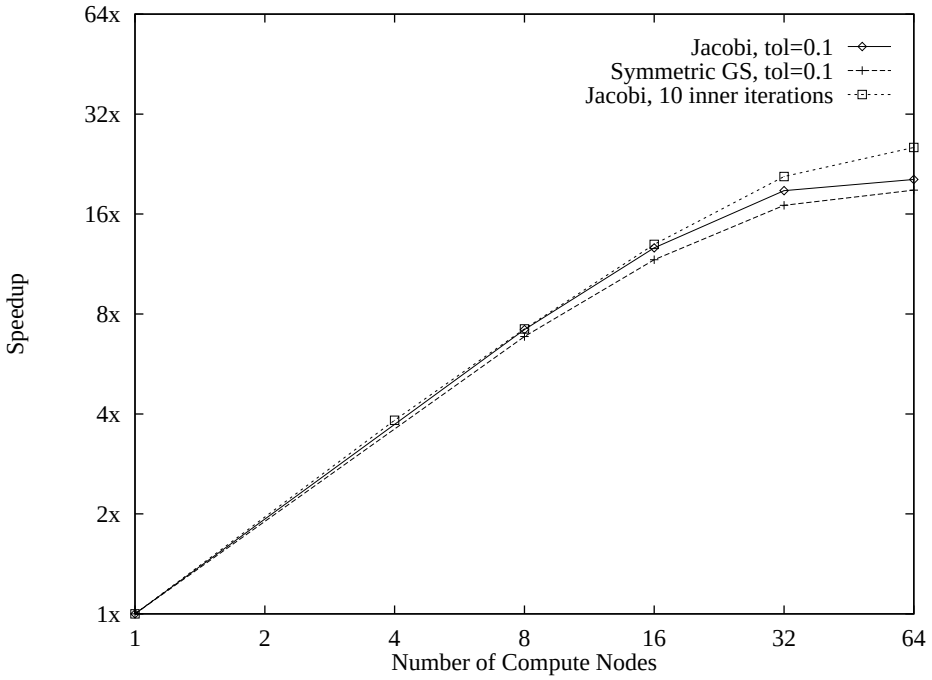


a) thin nodes

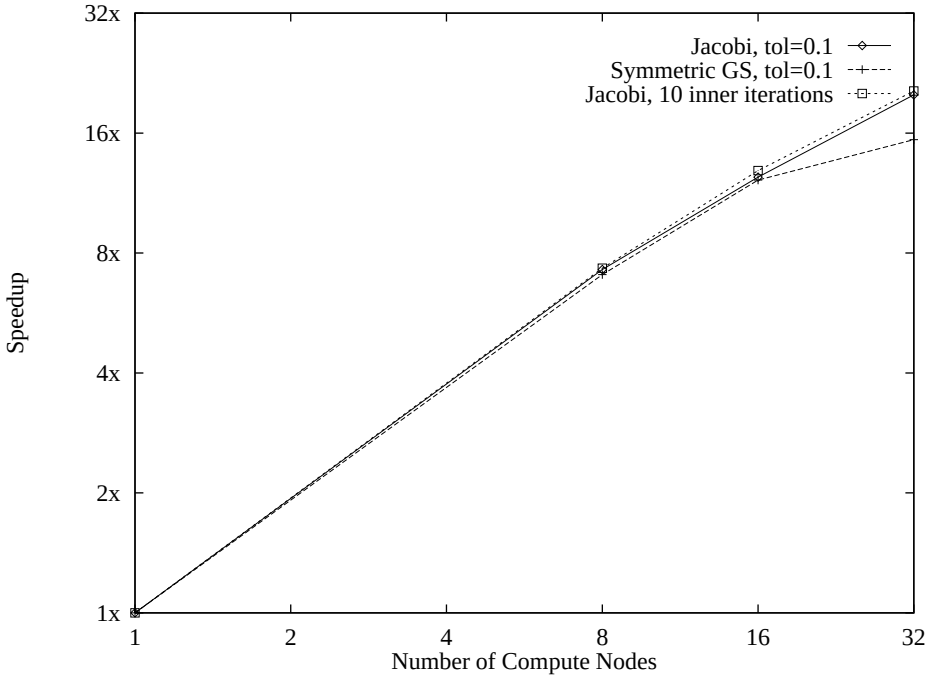


b) wide nodes

Figure 101: Speedup for the supersonic bump on the IBM SP-2.

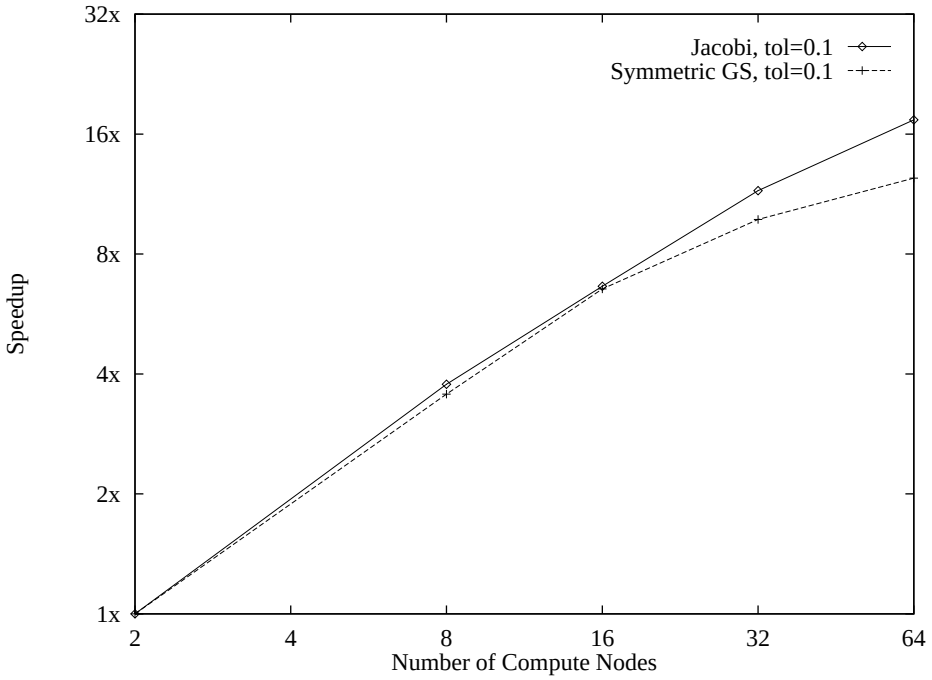


a) thin nodes

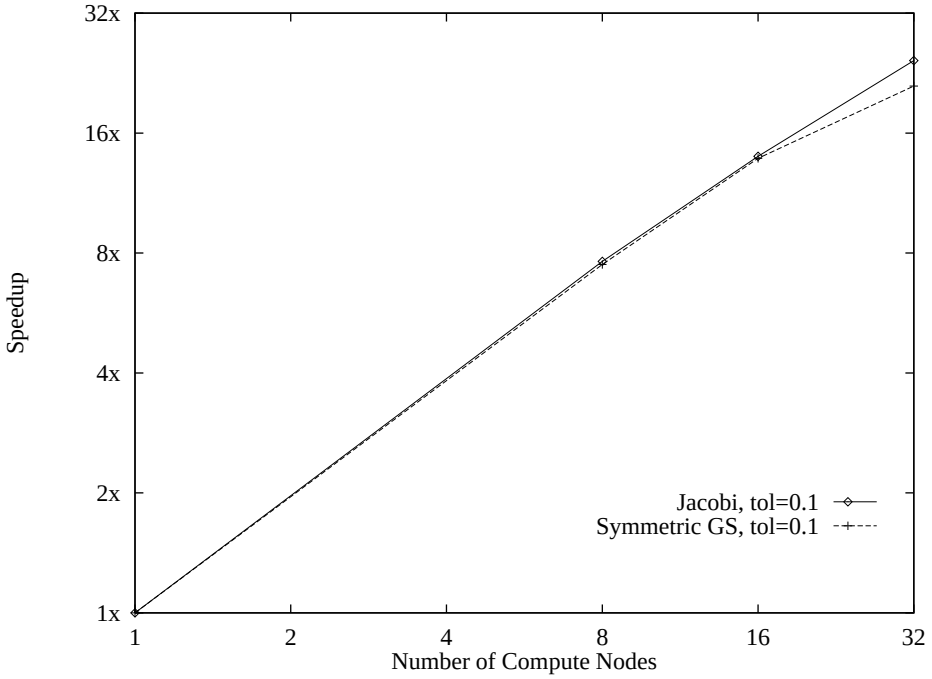


b) wide nodes

Figure 102: Speedup for the RAE 2822 airfoil on the IBM SP-2.

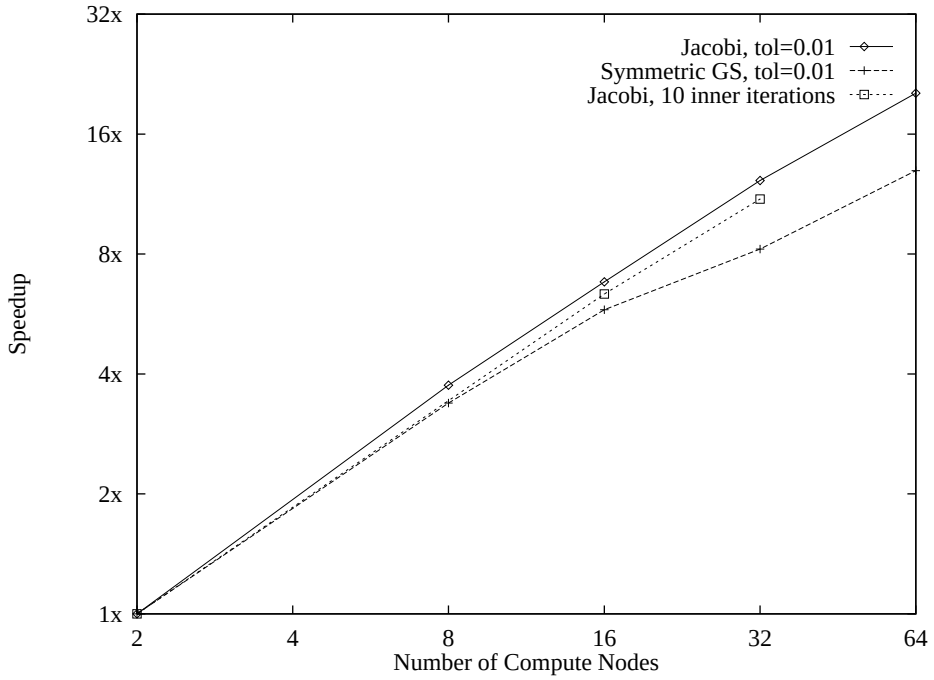


a) thin nodes

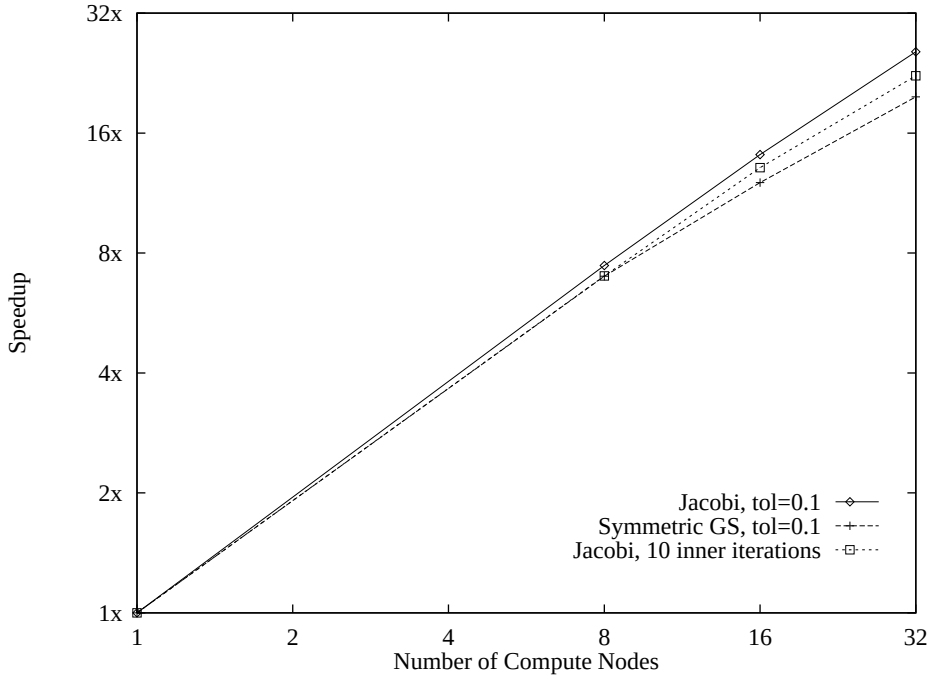


b) wide nodes

Figure 103: Speedup for the Hummel delta wing on the IBM SP-2.

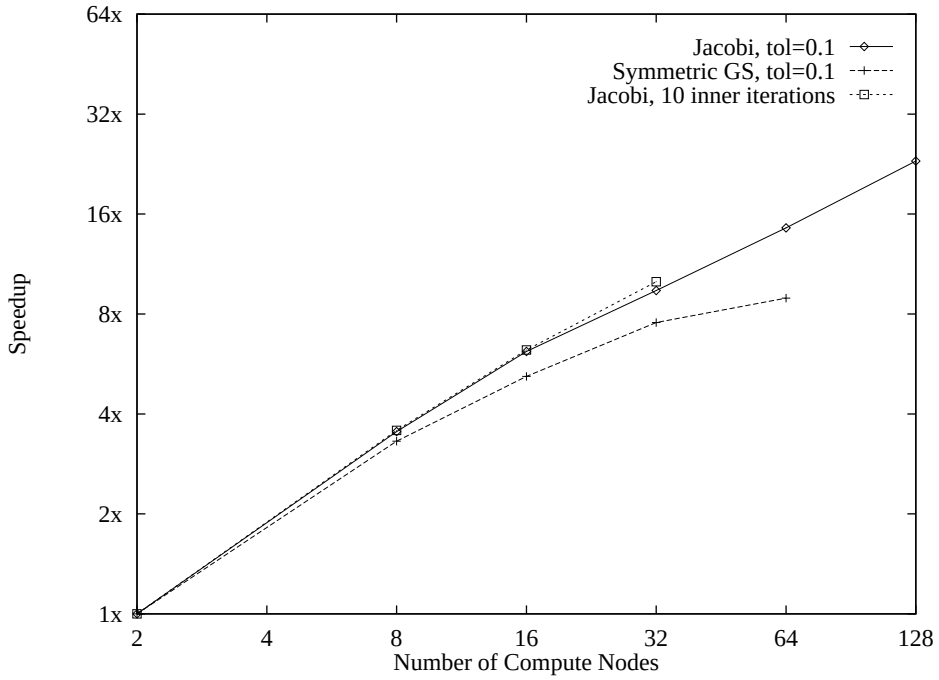


a) thin nodes

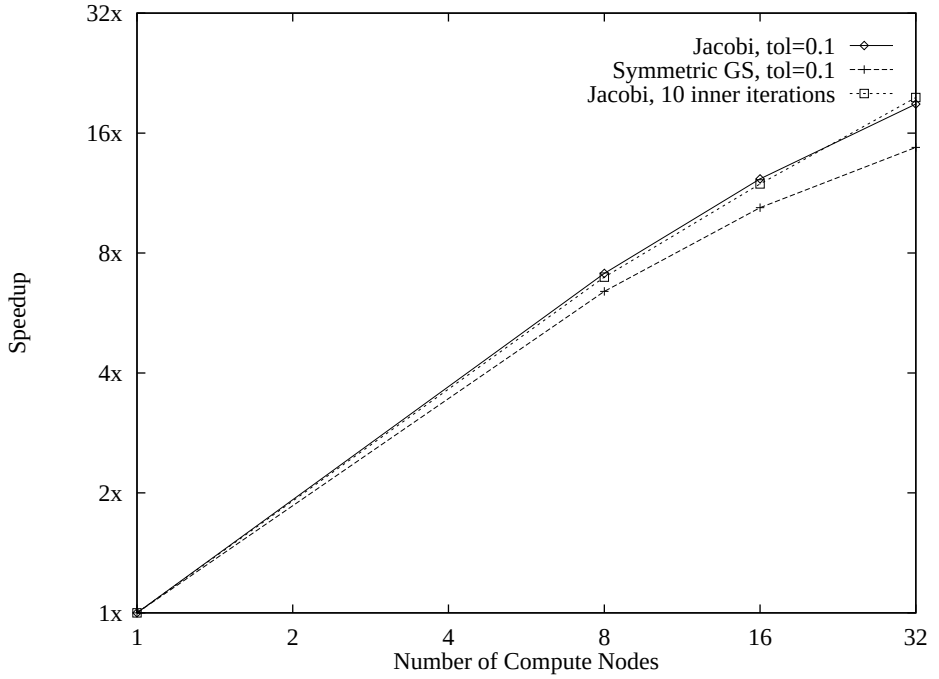


b) wide nodes

Figure 104: Speedup for the analytic forebody on the IBM SP-2.



a) thin nodes



b) wide nodes

Figure 105: Speedup for the ONERA M6 wing on the IBM SP-2.

Parallel Efficiency

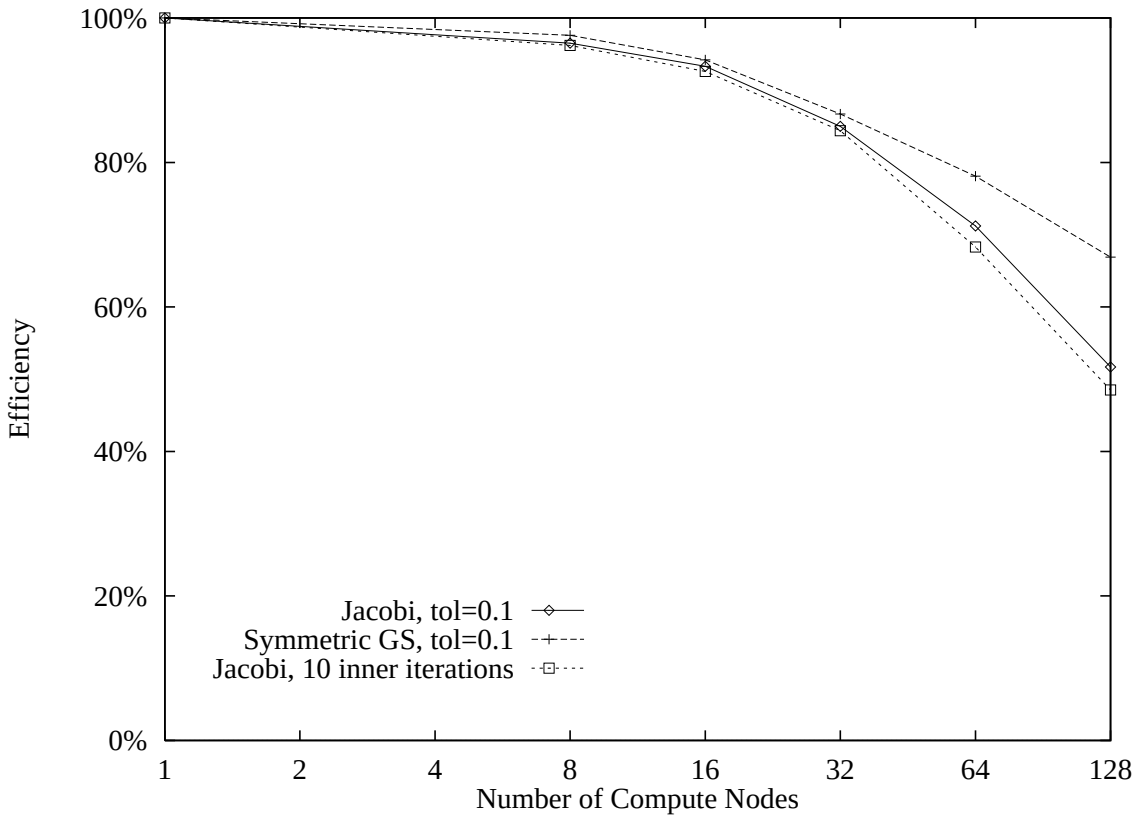


Figure 106: Parallel efficiency for the supersonic bump on the Intel Paragon.

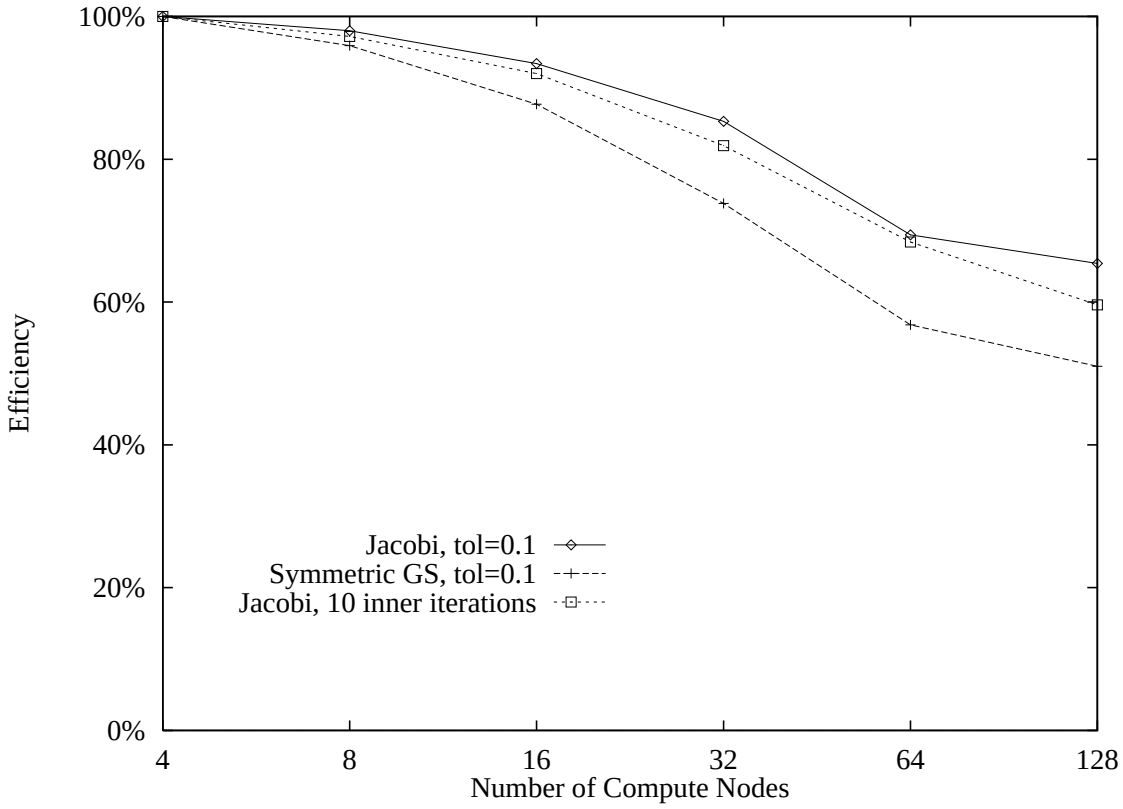


Figure 107: Parallel efficiency for the RAE 2822 airfoil on the Intel Paragon.

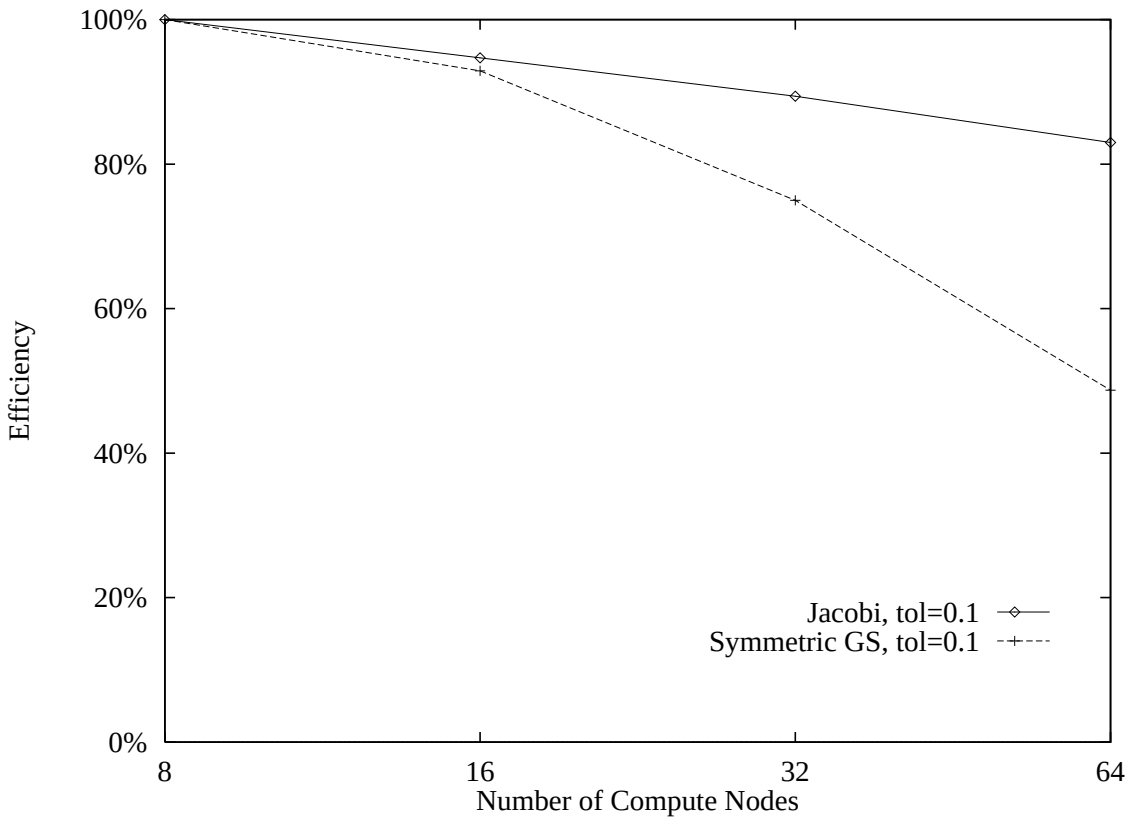


Figure 108: Parallel efficiency for the Hummel delta wing on the Intel Paragon.

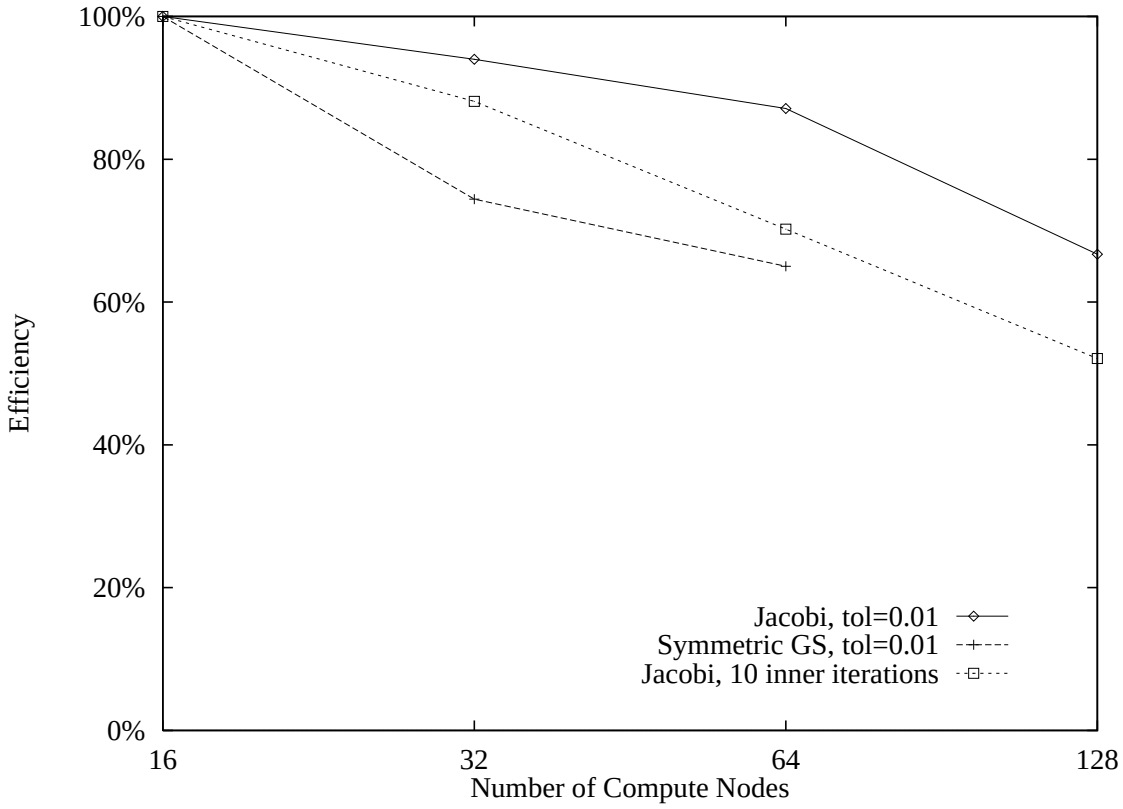


Figure 109: Parallel efficiency for the analytic forebody on the Intel Paragon.

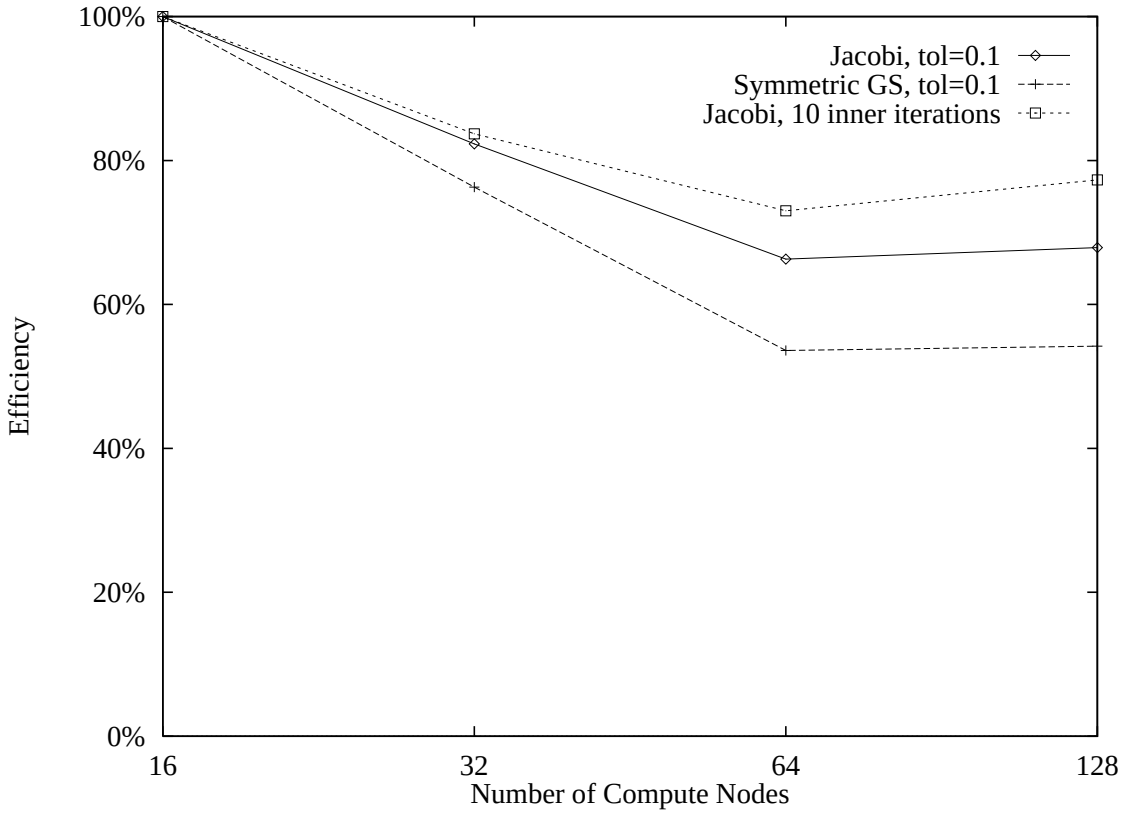
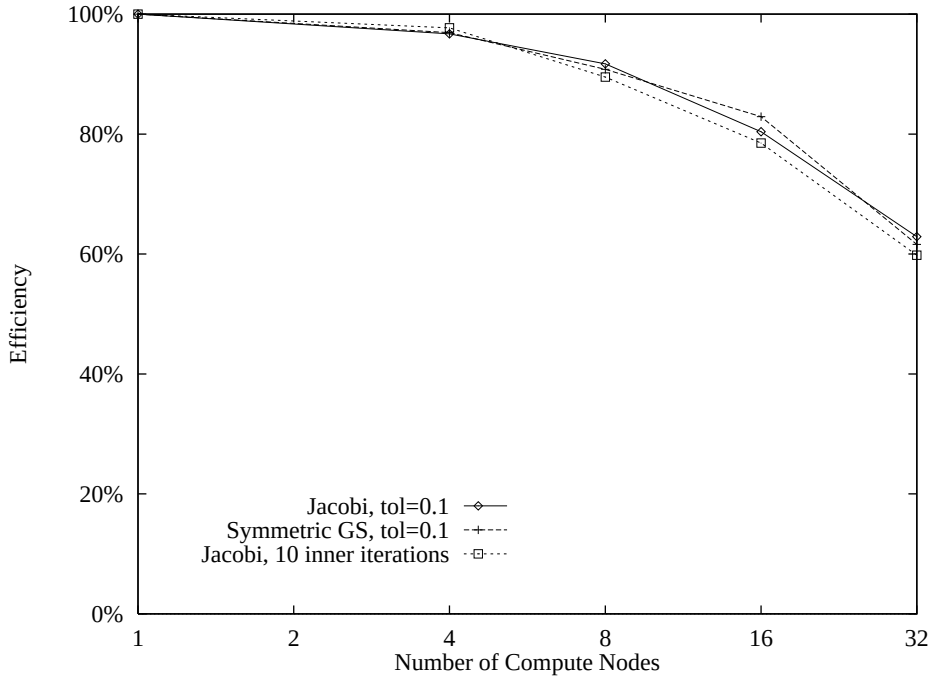
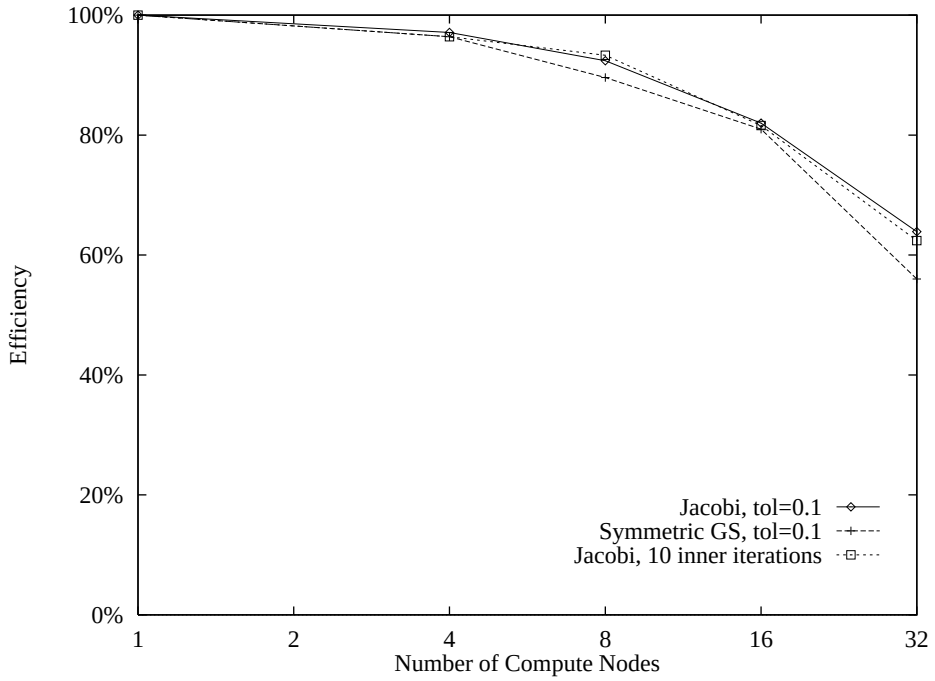


Figure 110: Parallel efficiency for the ONERA M6 wing on the Intel Paragon.

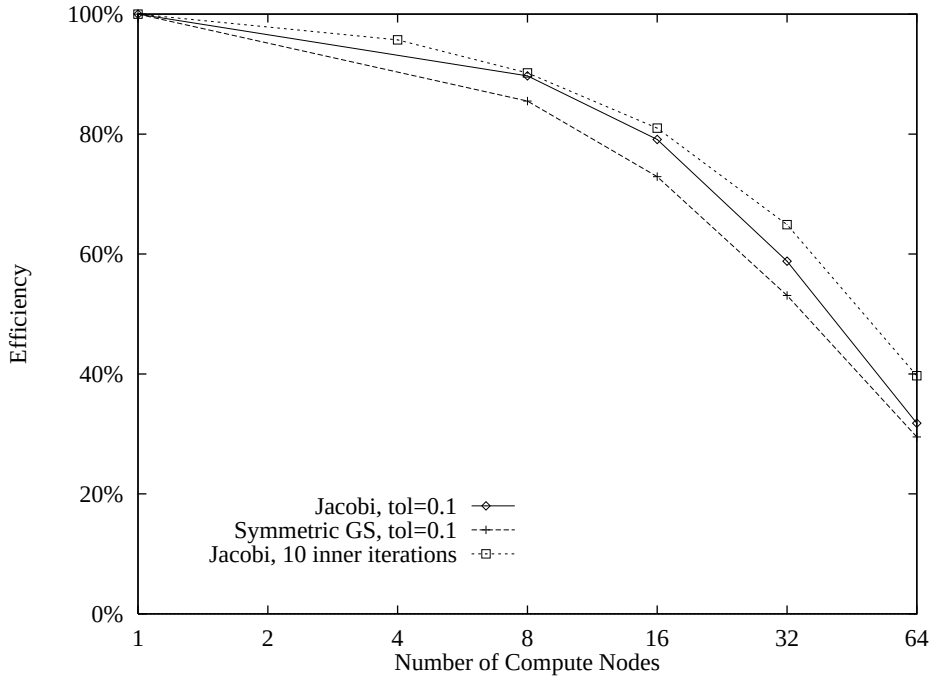


a) thin nodes

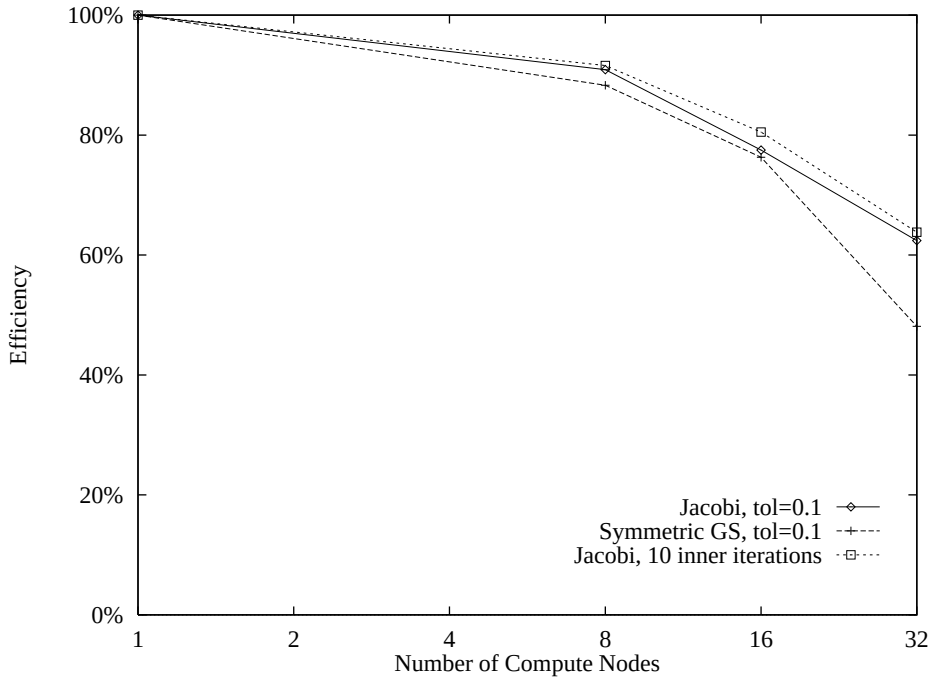


b) wide nodes

Figure 111: Parallel efficiency for the supersonic bump on the IBM SP-2.

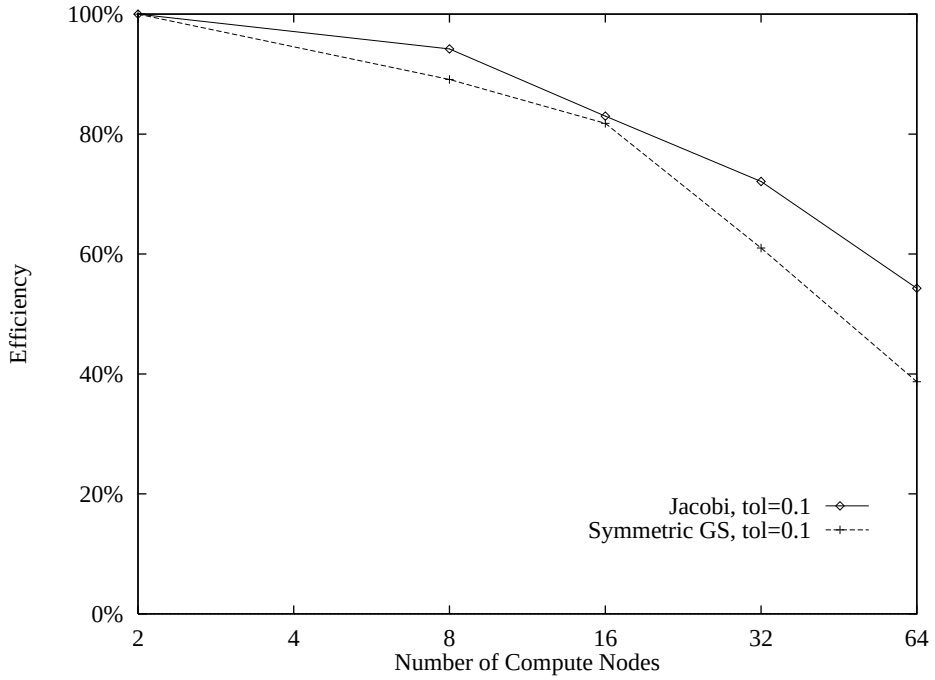


a) thin nodes

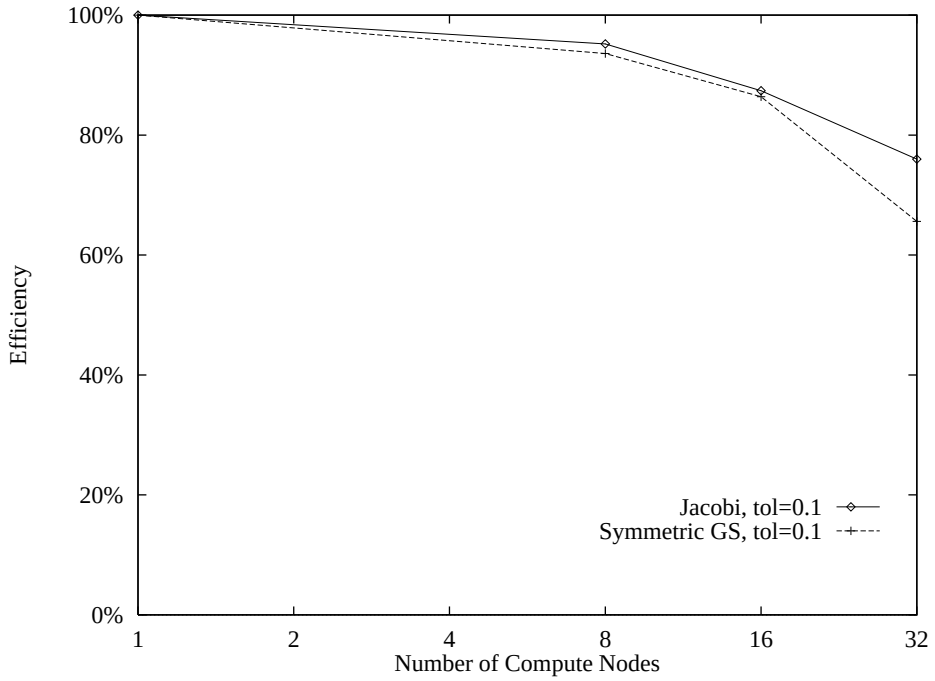


b) wide nodes

Figure 112: Parallel efficiency for the RAE 2822 airfoil on the IBM SP-2.

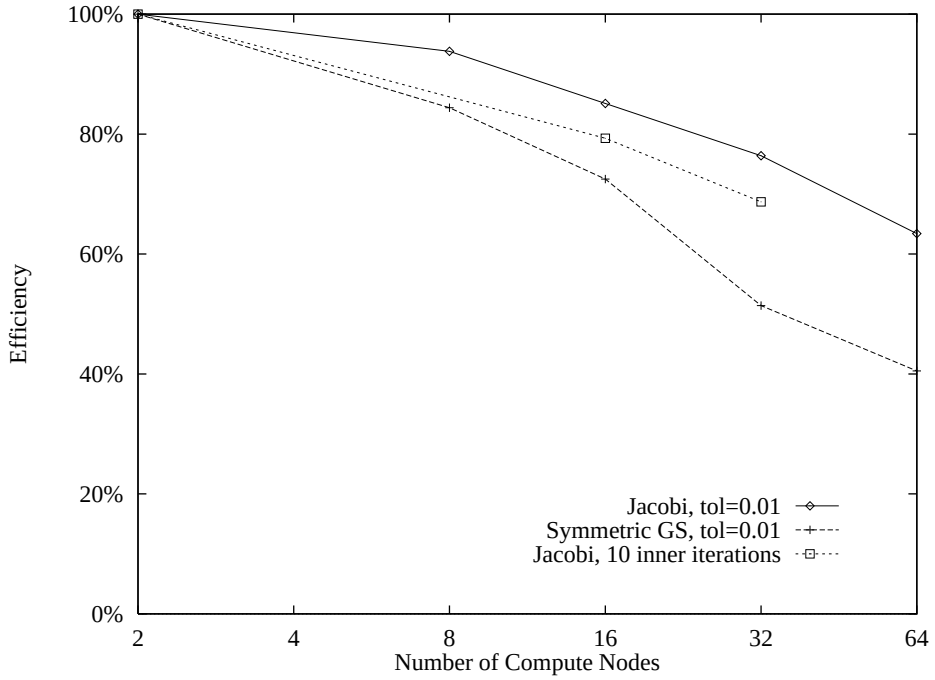


a) thin nodes

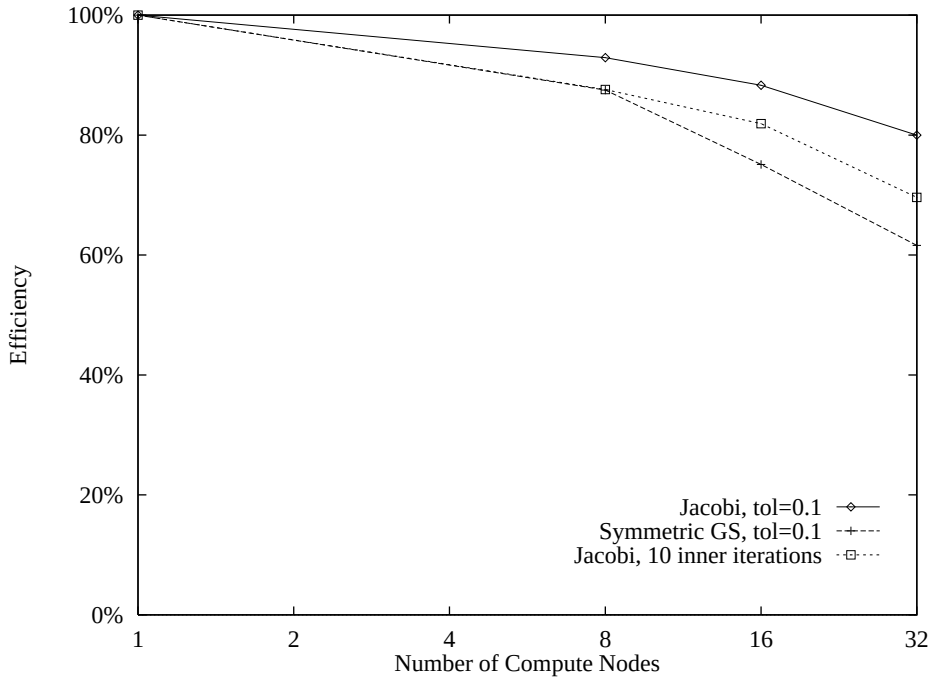


b) wide nodes

Figure 113: Parallel efficiency for the Hummel delta wing on the IBM SP-2.

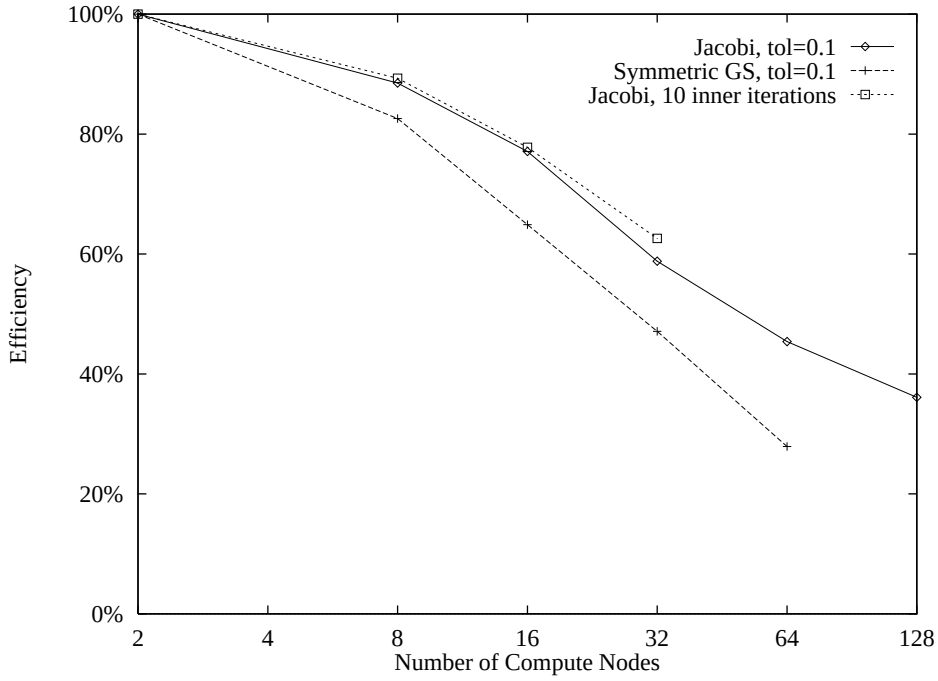


a) thin nodes

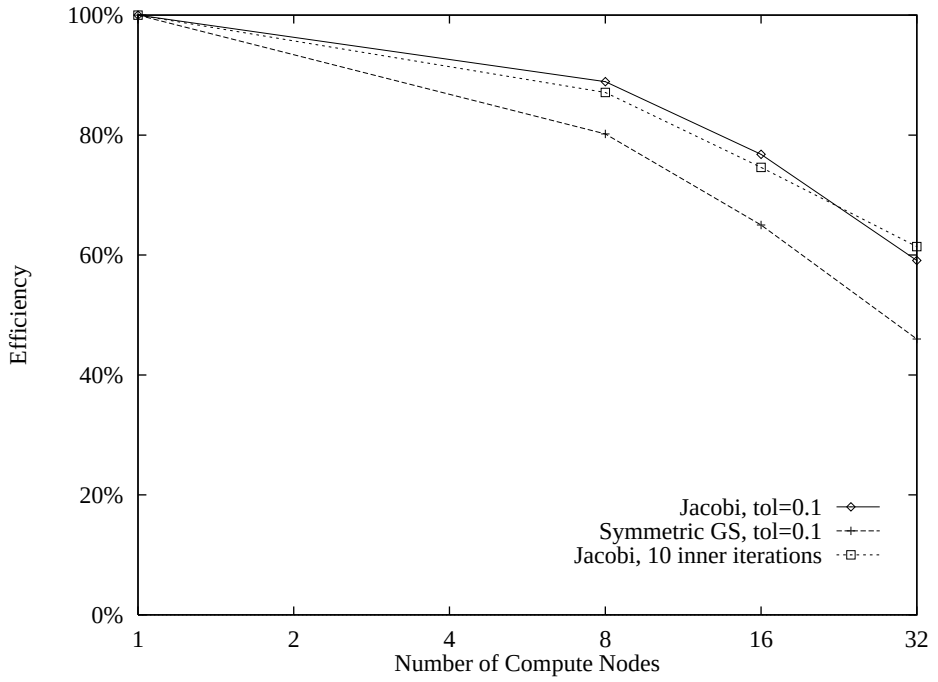


b) wide nodes

Figure 114: Parallel efficiency for the analytic forebody on the IBM SP-2.



a) thin nodes



b) wide nodes

Figure 115: Parallel efficiency for the ONERA M6 wing on the IBM SP-2.

Vita

Christopher William Stuteville Bruner was born on the 25 March 1960 at Indianapolis. He graduated from Purdue University in August 1987 with a Bachelor of Science degree in Aeronautical and Astronautical Engineering before earning a Master of Science degree in Aerospace Engineering from the University of Maryland in 1989.