INTERNATIONAL JOURNAL FOR NUMERICAL METHODS IN FLUIDS, VOL. 21, 445–466 (1995)

A VECTOR–PARALLEL SCHEME FOR NAVIER–STOKES COMPUTATIONS AT MULTI-GIGAFLOP PERFORMANCE RATES

A. A. LORBER AND G. F. CAREY

Computational Fluid Dynamics Laboratory, Aerospace Engineering and Engineering Mechanics, WRW 111, The University of Texas at Austin, Austin, TX 78712, U.S.A.

SUMMARY

A class of vector–parallel schemes for solution of steady compressible or incompressible viscous flow is developed and performance studies carried out. The algorithms employ an artificial transient treatment that permits rapid integration to a steady state. In the present work a four-stage explicit Runge–Kutta scheme employing variable local step size is utilized for the ODE system integration. The RK-4 scheme is restructured to allow vectorization and enhance concurrency in the calculation for a streamfunction–vorticity formulation of the flow problem. The parameters of the resulting RK scheme can be selected to accelerate convergence of the RK recursion. Four main procedures are considered which permit vector–parallel solution: a Jacobi update, a hybrid of the Jacobi and Gauss–Seidel method, red–black ordering and domain decomposition. Numerical performance studies are conducted with a representative viscous incompressible flow calculation. Results indicate that a scheme involving domain decomposition with a Gauss–Seidel type of update for the RK four-stage scheme is most effective and provides performance in excess of 8 Gflops on the Cray C-90.

KEY WORDS: vector–parallel computing; Navier–Stokes; Runge–Kutta; domain decompostion; CFD; gigaflop

1. INTRODUCTION

Most practical problems in computational fluid dynamics (CFD) are computationally intensive since realistic flows are complex and, in order to obtain accurate solutions, fine grids and complex physical models must be used. These needs translate into large memory requirements to obtain useful, meaningful solutions. In addition, fast execution speeds are required to obtain solutions in a reasonable amount of time. Moreover, design studies require that many solutions be computed. Vector and 'superscalar' processors are currently widely used to enhance performance rates. However, the demand for more complete models and finer resolution continues to press the need for increased execution speed. Unfortunately, the performance gains which can be obtained using vector architectures are physically limited. For example, the speed at which information can be passed within a computer's circuitry is limited by the speed of light and gate switching delays.[1] Other problems encountered are limits on packaging densities and heat dissipation rates. One solution is the use of multiprocessor architectures in which many computer processors work simultaneously (in parallel) on the flow problem.[2] Because of the speed-up potential offered and the economics of hardware duplication, the use of parallel architectures is rapidly increasing and becoming an integral part of large-scale CFD.

CCC 0271–2091/95/060445–22
© 1995 by John Wiley & Sons, Ltd.

Received August 1994
Revised February 1995

However, the complexity of multiprocessor algorithms and communication is obviously higher than that for single vector processors. In the present work we consider several issues fundamental to algorithms and parallel performance for a compressible viscous flow solver, including relating performance rate to the number of processors and algorithm used. The flow equation formulation exploits artificial relaxation in time to develop an explicit time-stepping recursion which allows the steady state solution to be obtained rapidly. This integration is performed using a special four-stage Runge–Kutta algorithm which includes adaptive local time stepping. The basic Runge–Kutta algorithm employs a Gauss–Seidel update method with natural ordering and can only be run in non-vector mode on a uniprocessor. In the present work this scheme is restructured in four ways to allow vectorization and enhance concurrency. The restructured four-stage RK scheme is only second-order-accurate, which is immaterial in the present context. However, we exploit this fact to select the RK parameters so that a larger local time step can be taken. Hence this results in a new class of accelerated iterative schemes or 'pseudo-time-iterative' methods.[3] Numerical performance studies are carried out using the restructured algorithms on Cray Y-MP and C-90 computers. In particular, a domain decomposition technique yields an algorithm which achieves multi-gigaflop performance on the Cray machines for reasonable meshes.

In the following sections the background and mathematical formulation of the compressible flow solver are first discussed. This is followed by a presentation of the four-step Runge–Kutta algorithm developed for this analysis. The manner in which vectorization and concurrency are exploited in this algorithm is then discussed. Finally, the performance results and conclusions drawn from the study are given.

2. FORMULATION

Motivated by the success of time-marching schemes in the efficient solution of the compressible Euler equations, researchers have extended this approach to the Navier–Stokes equations. For the incompressible case a difficulty arises since the incompressible continuity equation does not contain a derivative with respect to time.[4] This may be overcome by using either artificial compressibility, which adds a time term to the continuity equation,[5] an implicit ADI scheme with upwind differencing, which circumvents the need for a continuity equation time derivative,[6] the introduction of the pressure Poisson equation, which replaces the continuity equation,[7] or a suitable non-primitive variable formulation, which automatically satisfies the continuity equation.[8]

The formulation used in the present treatment is a combination of a non-primitive variable and pressure Poisson equation approach, extended to compressible flow[9] to allow solution over a range of Mach numbers. Here a compressible streamfunction–vorticity formulation is utilized to calculate the velocity field, a compressible pressure Poisson equation is used to calculate the pressure and an energy equation is used to update the density. The derivation of the compressible flow formulation is found to be a natural extension of the usual incompressible formulation. Owing to the simplicity of the extension, it is possible to calculate incompressible flow solutions in two ways using the compressible code: either by setting the incident Mach number $M \ll 1$ or by first initializing density to unity and solving the streamfunction–vorticity equations for the velocity field, followed by a postprocess solution of the pressure Poisson equation to calculate the pressure field.

The values of density, velocity and pressure within the flow field are governed by the continuity, momentum and energy equations. For two-dimensional, steady, compressible, viscous flow with no heat addition or body forces the continuity, momentum and energy equations may be written in Cartesian co-ordinates in non-dimensional form as

$$(\rho u)_x + (\rho v)_y = 0, \tag{1}$$

$$\rho u u_x + \rho v u_y = -p_x + \frac{1}{Re}[\tfrac{2}{3}(2u_x - v_y)_x + (u_y + v_x)_y], \tag{2}$$

$$\rho u v_x + \rho v v_y = -p_y + \frac{1}{Re}[(v_x + u_y)_x + \tfrac{2}{3}(2v_y - u_x)_y], \tag{3}$$

$$H = \frac{\gamma}{\gamma - 1}\frac{p}{\rho} + \tfrac{1}{2}V^2 = \text{constant enthalpy}, \tag{4}$$

where $V^2 = u^2 + v^2$ and the Reynolds number is defined as $Re = \rho_\infty U_\infty L/\mu_\infty$.

The dimensional and non-dimensional quantities are related through the expressions

$$x = \frac{\bar{x}}{L}, \quad y = \frac{\bar{y}}{L}, \quad p = \frac{\bar{p}}{\rho_\infty U_\infty^2}, \quad u = \frac{\bar{u}}{U_\infty}, \quad v = \frac{\bar{v}}{U_\infty}, \quad \rho = \frac{\bar{\rho}}{\rho_\infty} \quad H = \frac{\bar{H}}{U_\infty^2},$$

where the barred terms represent dimensional quantities and U_∞ and ρ_∞ are the freestream velocity and density respectively. The characteristic length of the flow is L. It is assumed that the viscosity of the flow is constant throughout; thus $\bar{\mu} = \mu_\infty$.

By taking the curl of the momentum equations, the pressure can be eliminated to obtain an equation involving vorticity, velocity and density. Then the definition of vorticity $\omega = v_x - u_y$ and the continuity equation are used to obtain the vorticity transport equation

$$-\rho \mathbf{V}\cdot\nabla\omega + \tfrac{1}{2}(V^2)_x\rho_y - \tfrac{1}{2}(V^2)_y\rho_x + \frac{1}{Re}\nabla^2\omega = 0. \tag{5}$$

Introducing the streamfunction ψ with $\rho u = \psi_y$ and $\rho v = -\psi_x$ into $\omega = v_x - u_y$ gives

$$\omega + \left(\frac{\psi_x}{\rho}\right)_x + \left(\frac{\psi_y}{\rho}\right)_y = 0, \tag{6}$$

which completes the standard streamfunction–vorticity derivation.

Next, taking the divergence of the momentum equation, the viscous term can be eliminated to obtain an equation relating pressure to vorticity, velocity and density. To see this, first set $\omega = v_x - u_y$ in (2) and (3) to obtain the following form of the momentum equations:

$$P'_x = \rho v\omega + \tfrac{1}{2}\rho_x V^2 - \frac{1}{Re}\omega_y, \tag{7}$$

$$P'_y = -\rho u\omega + \tfrac{1}{2}\rho_y V^2 + \frac{1}{Re}\omega_x, \tag{8}$$

where P'; is a modified pressure defined as

$$P' = p + \tfrac{1}{2}\rho V^2 - \frac{4}{3}\frac{1}{Re}(\nabla\cdot\mathbf{V}). \tag{9}$$

Now, differentiating (7) with respect to x and (8) with respect to y and adding, one obtains the pressure Poisson equation

$$\nabla^2 P' = \sigma, \tag{10}$$

where

$$\sigma = (\rho v\omega + \tfrac{1}{2}\rho_x V^2)_x + (-\rho u\omega + \tfrac{1}{2}\rho_y V^2)_y. \tag{11}$$

3. ARTIFICIAL TRANSIENT SCHEME

For simplicity and convenience in promoting the central concepts, let us first consider the incompressible case in which the streamfunction, vorticity and velocity fields, but not the pressure field, are calculated. For this incompressible case ρ is set equal to unity and the problem reduces to solving (5) and (6) subject to boundary conditions on ψ and ω.

In the present work we develop artificial transient algorithms to iterate by time stepping to a steady state.[3] To achieve this form, artificial transient terms are added to the vorticity transport and streamfunction equations. For example, the term ω_t is added to the vorticity transport equation (5) to create the transient convection–diffusion equation

$$\omega_t = -\rho \mathbf{V}\cdot\nabla\omega + \tfrac{1}{2}(V^2)_x\rho_y - \tfrac{1}{2}(V^2)_y\rho_x + \frac{1}{Re}\nabla^2\omega. \tag{12}$$

Similarly, the term ψ_t is added to the streamfunction equation (6) to produce the transient diffusion equation

$$\psi_t = \left(\frac{\psi_x}{\rho}\right)_x + \left(\frac{\psi_y}{\rho}\right)_y + \omega. \tag{13}$$

The spatial derivatives in (12) and (13) may be discretized using finite differences, finite volumes or finite elements to obtain a semidiscrete system of the form

$$\frac{d\mathbf{q}}{dt} = \mathbf{r}(\mathbf{q}), \tag{14}$$

where $\mathbf{q}$ is the solution vector containing the discrete values of ω and ψ, and $\mathbf{r}$ is the residual vector consisting of the difference functions created by the discretization of the spatial derivatives.

Note that at a steady state solution of the artificial transient forms of the vorticity transport and streamfunction equations, $d\mathbf{q}/dt = 0$, so $\mathbf{r}(\mathbf{q}) = 0$ and we have a solution to (5) and (6). Hence it is usual to interpret $\mathbf{r}$ as a residual whose value has to be zero in order to obtain a solution. This is consistent also with the usual definition of residual in iterative solutions to the steady state discrete system.[3]

To integrate (14), any number of ODE system integrators can be used.[10,11] In this study we develop a special form of explicit four-step Runge–Kutta integration algorithm with accelerated convergence properties for $\mathbf{r}(\mathbf{q}) \to 0$. We then show that it is well suited to efficient vector–parallel solution.

4. RUNGE–KUTTA RECURSION

Consider the intial value problem

$$\frac{d\mathbf{q}}{dt} = \mathbf{f}(t, \mathbf{q}), \qquad \mathbf{q}(t_0) = \mathbf{q}_0, \tag{15}$$

where $\mathbf{q}_0$ is an initial vector (starting value or 'iterate'). The basis for all Runge–Kutta methods is to express the solution $\mathbf{q}$ at some time t_{n+1} as

$$\mathbf{q}_{n+1} = \mathbf{q}_n + \sum_{i=1}^{s} w_i \mathbf{k}_i, \tag{16}$$

with

$$\mathbf{k}_i = h_n \mathbf{f}\left(t_n + \alpha_i h_n,\ \mathbf{q}_n + \sum_{j=1}^{i-1} \beta_{ij}\mathbf{k}_j\right), \tag{17}$$

where $h_n = t_{n+1} - t_n$, $\alpha_1 = 0$ and s, w_i, α_i and β_{ij} are specific constants. These parameters are usually chosen so that the coefficients of the increasing powers of h_n in a Taylor series expansion of the right- and left-hand sides of (16) agree exactly up to a desired order.[11]

In this study we utilize a four-stage (i.e. $s = 4$) Runge–Kutta (RK) method similar to that of Jameson[12] but exploit the stability properties of the associated RK families in selecting parameters. It is computationally convenient for our purposes (since stability rather than accuracy is at issue) to restrict the class of RK-4 formulae to a sequential four-step scheme. That is, we set $w_3 = w_2 = w_1 = 0$, $w_4 = 1$ and $\beta_{31} = \beta_{41} = \beta_{42} = 0$. Equations (16) and (17) then become

$$\mathbf{q}_{n+1} = \mathbf{q}_n + w_4 \mathbf{k}_4, \tag{18}$$

where $\mathbf{k}_4$ is computed as the fourth step in the sequence

$$\begin{aligned} \mathbf{k}_1 &= h_n \mathbf{f}(t_n, \ \mathbf{q}_n), \\ \mathbf{k}_2 &= h_n \mathbf{f}(t_n + \alpha_2 h_n, \ \mathbf{q}_n + \beta_{21}\mathbf{k}_1), \\ \mathbf{k}_3 &= h_n \mathbf{f}(t_n + \alpha_3 h_n, \ \mathbf{q}_n + \beta_{32}\mathbf{k}_2), \\ \mathbf{k}_4 &= h_n \mathbf{f}(t_n + \alpha_4 h_n, \ \mathbf{q}_n + \beta_{43}\mathbf{k}_3). \end{aligned} \tag{19}$$

This implies sequential dependence of $\mathbf{k}_i$ on $\mathbf{k}_{i-1}$ so that only the most recent vector need be saved. It also emphasizes the vector form of the calculations.

From the structure in (18) and (19) we may similarly define intermediate values of the solution vector $\mathbf{q}^m$ such that the four-stage algorithm can be written as

$$\begin{aligned} \mathbf{q}^1 &= \mathbf{q}^0 + \beta_{21} h_n \mathbf{f}(t_n, \ \mathbf{q}^0), \\ \mathbf{q}^2 &= \mathbf{q}^0 + \beta_{32} h_n \mathbf{f}(t_n + \alpha_2 h_n, \ \mathbf{q}^1), \\ \mathbf{q}^3 &= \mathbf{q}^0 + \beta_{43} h_n \mathbf{f}(t_n + \alpha_3 h_n, \ \mathbf{q}^2), \\ \mathbf{q}^4 &= \mathbf{q}^0 + h_n \mathbf{f}(t_n + \alpha_4 h_n, \ \mathbf{q}^3), \end{aligned} \tag{20}$$

where $\mathbf{q}^0 = \mathbf{q}_n$ and $\mathbf{q}_{n+1} = \mathbf{q}^4$. Note that now $\mathbf{q}_{n+1}$ is assembled in the vector $\mathbf{q}^m$ as m increases from $m = 0$ to 4. Here it is seen that only the old solution vector $\mathbf{q}^0$ and the assembling vector $\mathbf{q}^m$ need be stored.

This simplified form of the RK algorithm is achieved at a loss of accuracy. That is, the scheme is no longer $O(h^4)$ for this restrictive choice of parameters. In fact, to produce a second-order approximation, the values of α_4 and β_{43} must be $\alpha_4 = \beta_{43} = \frac{1}{2}$, but the choice of α_3, β_{32}, α_2 and β_{21} remains open. Hence we are free to make this choice to enhance the stability properties for the problem. For example, setting $\alpha_3 = \beta_{32} = \frac{1}{3}$ and $\alpha_2 = \beta_{21} = \frac{1}{4}$ will produce the same stability domain as fourth-order four-stage methods (although this choice does not produce a fourth-order method as noted above). Other choices will produce different stability domains that can be tailored to the eigenspectrum of the problem. In this way we are able to develop RK variations that are not necessarily time-accurate but converge to the steady state solution much faster. For example, we may use the Chebyshev-based[3] coefficients $\beta_{21} = 0{\cdot}1$, $\beta_{32} = 0{\cdot}2124$ and $\beta_{43} = 0{\cdot}4265$ to accelerate the scheme.

Applying (20) to the semidiscrete system (14) results in the algorithm

$$\begin{aligned} \mathbf{q}^0 &= \mathbf{q}_n, \\ \mathbf{q}^m &= \mathbf{q}^0 + \varepsilon_m h_n \mathbf{r}(\mathbf{q}^{m-1}) \quad \text{for } m = 1, 2, 3, 4, \\ \mathbf{q}_{n+1} &= \mathbf{q}^4, \end{aligned} \tag{21}$$

where $\varepsilon_m = \beta_{21}$, β_{32}, β_{43}, 1 for $m = 1$, 2, 3, 4, respectively. Note that the boundary conditions are applied after each calculation of $\mathbf{q}^m$. The residual $\mathbf{r} = \mathbf{f}$ has been introduced to emphasize the

equivalence of the procedure to a residual-based iterative method for iterating an initial iterate to the steady solution. This algorithm is self-starting from an initial guess for **q** and may be applied until the residual **r**(**q**) is reduced below a given tolerance. The maximum allowable 'time' step h_n for stability at the current 'time' level is recalculated after every 'time' step.

4.1. Local relaxation

A stability analysis of the associated linear system determines the maximum step size h_n allowable, i.e. $h_n\lambda_{\max}$ should lie in the associated stability domain. In the application of interest here the properties of the system vary dramatically in space—a region in the viscous flow near a body will have larger flow gradients than the far field. Since the objective is to damp the residuals and accelerate iteration to the steady state, the variation in local stiffness can be accommodated by carrying out a local stability analysis at each grid point and determining an estimate of a local step h_n^{ij} at grid point (i, j). This is equivalent to replacing h_n in (21) by the diagonal matrix **H** with diagonal entries h_n^{ij}. Alternatively we can write $h_n^{ij} = \omega_{ij}h_n$, where ω_{ij} is a local extrapolation parameter or overrelaxation parameter that varies for each point (i, j) in the discretization. Note that for the linear problem on a uniform grid $\omega_{ij} = 1$ at all points.

Remark. This point extrapolation method is now a point iterative residual method and not an RK integration method. If the pointwise overrelaxation is removed ($\omega_{i,j} = 1$), then the scheme reverts to an RK integration scheme. If h_n satisfies the stability condition and accuracy requirements, then time-accurate solutions to real transient problems can also be computed (although this is not relevant for the artificial transient formulation constructed here).

5. VECTOR AND PARALLEL ASPECTS

In order to increase the execution speed of a computer program, it is best to first isolate those portions of the code where the majority of the execution time is spent. The programmer can then focus on those parts where reprogramming will have the greatest effect. Utilities for this type of performance analysis are available on most computer systems. For example, on a Cray UNICOS system the utilities FLOWTRACE, PERFTRACE and PROF can be used.[13]

Through such a performance analysis of the compressible code it was found that the Runge–Kutta solver used for the vorticity transport and streamfunction equations as well as the line successive overrelaxation solver used for the pressure Poisson equation constitute the most computationally intensive parts of the code. In order to limit the scope of the current investigation, we first consider the incompressible viscous flow problem. Accordingly, it is necessary to optimize the performance of the Runge–Kutta solver. Calculations of the streamfunction, vorticity and velocity fields for incompressible flow as well as performance statistics are considered in the results given later.

The computationally intensive nature of the Runge–Kutta solver can be discerned from (21), where it is observed that in order to advance the streamfunction–vorticity solution by one time step, four residual evaluations over the entire grid along with four boundary condition calculations must be performed. Implementation of vectorization and concurrency for the boundary condition calculation of the Runge–Kutta solver is straightforward and will not be discussed further, as the Fortran compilers produced efficient vectorized and parallel code in this area with only minor modifications. The residual calculation, however, required extensive modification and analysis and is therefore the focus of the following discussion.

In the following subsection, four vector–concurrent methods are described which are used as modifications in the previous baseline Runge–Kutta residual calculation. First, however, it is necessary

to briefly describe the way in which vector and/or parallel programs are generated on the representative parallel–vector computers used in this study.

Our preliminary studies were carried out on an eight-processor Alliant FX/8 system, with subsequent large-scale applications on the Cray Y-MP and C-90 systems. As discussed in Section 7, the preliminary Alliant studies were performed to gauge the relative effectiveness of different solution algorithms in terms of convergence rate, complexity and parallel speed-up rather than in terms of flop rate. Once the best algorithm was found, the Cray systems were used to achieve high flop rates while maintaining good convergence rates using reasonable meshes.

On the Cray Y-MP, up to eight vector processors can be utilized to increase execution speed. On the C-90, 16 processors are available. Each processor of the Y-MP has a 6 ns clock and a single vector pipe and allows the chaining of addition and multiplication operations, giving the Y-MP a peak performance of 2·67 Gflops. The C-90 has a 4·167 ns cycle time and the processors have dual vector pipes, each of which allows chaining, delivering a 15·36 Gflop peak performance rate.

The Alliant FX/FORTRAN compiler, by default, produces vector–parallel programs. To obtain easily reproducible results in this study, these default compiler settings were used whenever possible. However, it was often necessary to specify with compiler directives which do-loops were to be run concurrently and which were to vectorize, or to force vectorization and concurrency on a portion of code where the compiler identified possible data dependencies.

Vectorization is automatic on the standard Cray UNICOS compiler, cft77, while parallel code generation is not. (This compiler is used on both the Y-MP and C-90.) Instead, when performing parallel processing on a Cray system, the user has the options of macrotasking, microtasking or autotasking. Macrotasking requires inserting explicit calls to a special Fortran-callable library of synchronization routines. It tends to be cumbersome and in most cases does not yield the best results. Microtasking, which was developed subsequently, consists of placing CMIC$ microtasking directives in an existing code which is then read by a preprocessing program named 'fmp'. Preprocessing by fmp creates a new Fortran source which is then compiled using cft77. Since microtasking uses directives rather than macrotasking's cumbersome subroutine calls, microtasking is much simpler to use. In addition, microtasking uses special parallel processing features of the Cray Y-MP and C-90, thus producing much improved overall speed-up results as compared with macrotasking. Autotasking involves a Fortran preprocessor named 'fpp' which automatically analyses a Fortran code for parallelism, placing an enriched set of microtasking directives in the original Fortran code. The new code is then passed through fmp to cft77. Like fmp, fpp also reads directives. These CFPP$ autotasking directives can be placed in the original code by the programmer to aid and guide fpp. The enriched set of microtasking directives used by autotasking, which include simplified data-scoping techniques and the use of parallel regions, can lead to parallel performance which is superior to the older microtasking directives. The fpp–fmp–cft77 compilation process can be performed somewhat transparently using the Cray program cf77, which will run the preprocessors and compiler for the user.

In the current study a combination of autotasking and microtasking was used to achieve gigaflop performance in the compressible Navier–Stokes flow solver NSFLOW for moderate meshes on the Y-MP and multi-gigaflop rates on the C-90. The Fortran code was first modified with autotasking directives and preprocessed with fpp. This placed into the code the majority of microtasking directives needed for concurrent operation. Some of these microtasking directives were then modified and supplemented and the code processed with fmp. The final result was a highly optimized Fortran code which was then compiled. An example illustrating this approach is provided by the boundary condition treatment in the code: fpp was able to recognize this section as a parallel region and distinguish the region by surrounding the code with CMIC$ PARALLEL–CMIC$ END PARALLEL microtasking directives; however, it was necessary to use the CMIC$ CASE–CMIC$ CASE–...–CMIC$ END

CASE directives to break the code into blocks which could be executed in parallel. In addition, the full extent of parallel regions was not always recognized by fpp and had to be extended manually. Another example was in the RK residual calculation when domain decomposition was used. Even with directives, fpp refused to ignore the parallel dependencies and would not parallelize the calculation. This finally had to be done manually with microtasking directives.

On shared memory multiprocessor systems the multiple processors can be used in a variety of ways. Five strategies are used in the current study. In the first approach an innermost do-loop can be run concurrently over multiple processors in scalar mode. Secondly, to increase execution speed, the innermost do-loop can be run concurrently over multiple processors in vector mode. A third and usually faster method is to allow an innermost do-loop to run in vector mode while an outermost do-loop, if present, runs concurrently. This third method is typically the most desirable and is implemented by default by the Alliant's FX/FORTRAN compiler and Cray's fpp preprocessor whenever possible. Breaking from the do-loop-based concurrency, both the FX/FORTRAN compiler and fpp preprocessor allow subroutines to be run concurrently, without vectorization. This is the fourth method used. The fifth and final method is simply to run subroutines concurrently, utilizing vectorization in the subroutines whenever possible. Note that the fourth and fifth methods will not necessarily lead to performance gains over the first three methods owing to overhead associated with subroutine execution.

Differences in the implementation of concurrent algorithms occur between systems, e.g. systems differ in the manner in which a do-loop is distributed among processors for parallel execution. Systems such as the Alliant utilize what is termed stride-based parallelism. In this method the original do-loop is first distributed to all processors. Then, during execution, the stride of the loops is set equal to the number of processors. The starting points of the loops on each processor are reset such that each processor operates on separate portions of the original loop. By doing this, the compiler attempts to evenly distribute the work performed in the loop, even if some sections of the loop have a larger workload than others. The default method used for creating a concurrent do-loop on the Y-MP and C-90 is to split a do-loop into equal-sized single-stride chunks, with the number of chunks being equal to the number of processors. During concurrent operation, each chunk is run simultaneously on a separate processor.

5.1. RK update strategies

With a basic understanding of the options available it is now possible to consider the schemes used to allow the Runge–Kutta algorithm to run in a vector–concurrent manner. The methods developed in the present work are motivated by point and block update techniques for iterative solvers. First the five-point stencil and iteration definitions are presented. The Gauss–Seidel update method used in the baseline Runge–Kutta method is then discussed. Finally, the Jacobi, modified Gauss–Seidel, red–black and domain decomposition alternative RK schemes are given.

The Runge–Kutta residual calculation is performed in the computational plane over a rectangular finite difference grid labelled using natural ordering, with a corresponding central finite difference 'five-point stencil' (Figure 1(a)). Let us suppose that the solution calculation is progressing from point to point in the i-direction, with the calculation or update advancing once in the j-direction only after all the grid points on an $i=$ constant line have been transversed. We will define the action of the solver in moving from one grid point to the next as one grid step and each series of calculations in which values at all grid points along a single $i=$ constant line are updated as an i-direction sweep. Since the solution update is progressing in the positive i- and j-directions (natural ordering), when the value at point (i, j) is to be computed, the point $(i-1, j)$ has just been updated during the previous grid step. The value at the point $(i, j-1)$ was calculated in the current RK step during the previous i-direction sweep when

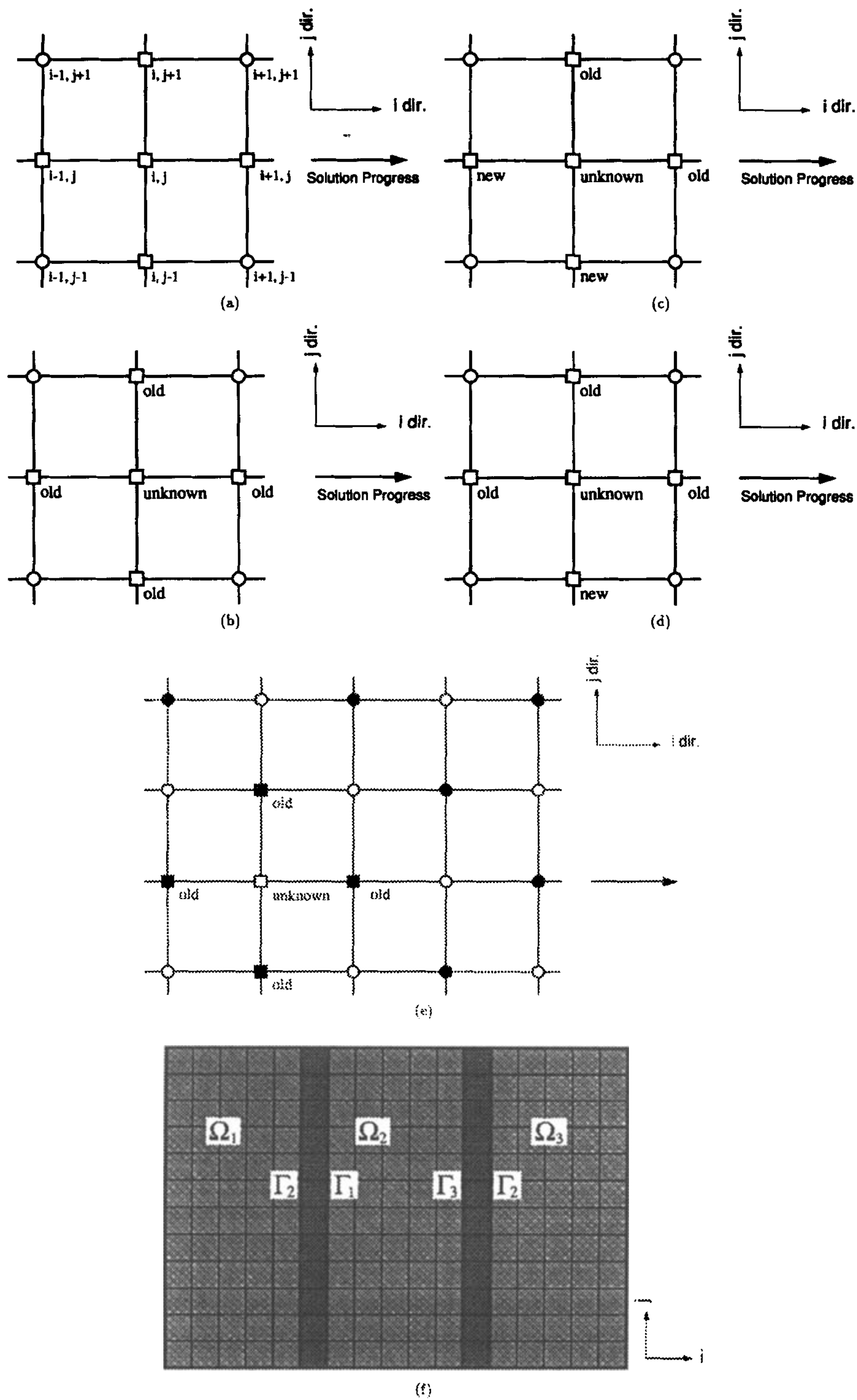

Figure 1. The central difference five-point stencil (a), typical grid point structures used in the Jacobi RK (b), Gauss–Seidel RK (c), modified Gauss–Seidel RK (d) and red (white)–black RK update methods (e) and a typical partitioning of the computational plane used with the domain decomposition update method (f)

the solver was traversing the $i-1$ line. The values at the points $(i+1, j)$ and $(i, j+1)$ were updated during the previous Runge–Kutta step. Let us label grid points with updated values from the previous RK step as 'old' and grid points with values from the current RK step as 'new'. When computing a value for a variable at a grid point (i, j), it is possible to utilize either old or new values from the points $(i-1, j)$ and $(i, j-1)$. The choice of new values may create data dependencies which can preclude the use of vectorization and/or concurrency and will affect the convergence of the algorithm. The selection of which data to use will now be addressed.

Jacobi RK. In the Jacobi RK update shown in Figure 1(b), only old grid values from the previous RK step are used to update the unknown value. This implies that there are no data dependencies in either vector, parallel or vector–parallel computations. In terms of the RK algorithm given in (21), the equation for the calculation of the $\mathbf{q}^m$ entry corresponding to the grid point (i, j) can be written for the Jacobi RK update method as

$$q^m_{i,j} = q^0_{i,j} + \varepsilon_m h_n r(q^{m-1}_{i+1,j},\ q^{m-1}_{i-1,j},\ q^{m-1}_{i,j},\ q^{m-1}_{i,j+1},\ q^{m-1}_{i,j-1}). \tag{22}$$

Using this approach allows each i-direction sweep to be run in vector mode on a uniprocessor machine or in concurrent mode on a multiprocessor non-vector machine. On a vector–parallel shared memory machine the i-direction sweeps can be run either in vector–parallel mode or in vector mode with the sweeps distributed among processors. However, the main drawback with the Jacobi RK update is that the use of four old values degrades convergence rates considerably. In addition, to store both old and new grid values, twice the storage is required as compared with more efficient update methods. Finally, there is a small overhead time required to replace the old grid values with new grid values in memory between RK steps. These ideas are consistent with those encountered in Jacobi iteration for linear systems, but here enter as part of the RK scheme.

Gauss–Seidel RK. If the unknown grid point value is updated using the two old and two new values as shown in Figure 1(c), the method is termed a Gauss–Seidel RK update. In the RK algorithm this update procedure takes the form

$$q^m_{i,j} = q^0_{i,j} + \varepsilon_m h_n r(q^{m-1}_{i+1,j},\ q^{m}_{i-1,j},\ q^{m-1}_{i,j},\ q^{m-1}_{i,j+1},\ q^{m}_{i,j-1}). \tag{23}$$

The Gauss–Seidel RK update gives good convergence results, as the maximum amount of current data is used when calculating the new values. In addition, the Gauss–Seidel RK update requires the least amount of storage, as both old and new grid values at one point need not be saved. For these two reasons the Gauss–Seidel approach is used as the baseline Runge–Kutta solver for comparison purposes. Unfortunately, the use of a Gauss–Seidel RK update in conjunction with natural ordering precludes the use of vectorization and/or concurrency. This is caused by data dependencies associated with the use of the new values from the current RK step, precisely in the same sense as for Gauss–Seidel iteration used for linear systems. (Note that diagonal grid point reordering can circumvent this problem.)

Modified Gauss–Seidel RK. A simple hybrid of the Gauss–Seidel RK and Jacobi RK update methods is shown in Figure 1(d). This will be referred to as a modified Gauss–Seidel RK update and, as can be seen, the unknown value is updated using one new value from the current RK step and the three old values from the previous RK step. In terms of the RK algorithm the modified Gauss–Seidel RK update is written as

$$q^m_{i,j} = q^0_{i,j} + \varepsilon_m h_n r(q^{m-1}_{i+1,j},\ q^{m-1}_{i-1,j},\ q^{m-1}_{i,j},\ q^{m-1}_{i,j+1},\ q^{m}_{i,j-1}). \tag{24}$$

The individual i-direction sweeps of the modified Gauss–Seidel RK approach vectorize completely on a single-processor machine and parallelize completely on a multiprocessor non-vector machine. On a vector–parallel machine, performance is lost over a Jacobi RK update, since the i-direction sweeps cannot be run simultaneously on separate processors. Instead, each sweep can be individually run in vector–concurrent mode, thus giving better performance than pure vectorization alone. As might be expected, the convergence behaviour of the modified Gauss–Seidel RK update scheme falls between the Jacobi RK and Gauss–Seidel RK methods. The extra memory requirement is minimized, as only one line of grid points in the i-direction need be saved, though the overhead swap time is still present.

Red–black RK. The red–black RK update scheme,[14] shown in Figure 1(e), vectorizes and parallelizes in the same manner as the Jacobi RK update, requires no extra storage, has little overhead and converges almost as well as Gauss–Seidel RK. In this method the red values are first updated using the black values calculated during the previous RK step. Then, using these newly calculated red values, updated black values are calculated, thus completing the current RK step. In reference to the RK update algorithm the red–black RK scheme is written in two steps: first, for all red points in $\mathbf{q}^m$,

$$q^m_{i,j_{\text{red}}} = q^0_{i,j} + \varepsilon_m h_n r(q^{m-1}_{i+1,j_{\text{black}}},\ q^{m-1}_{i-1,j_{\text{black}}},\ q^{m-1}_{i,j_{\text{black}}},\ q^{m-1}_{i,j+1_{\text{black}}},\ q^{m-1}_{i,j-1_{\text{black}}}); \tag{25}$$

then, for all black points in $\mathbf{q}^m$,

$$q^m_{i,j_{\text{black}}} = q^0_{i,j} + \varepsilon_m h_n r(q^m_{i+1,j_{\text{red}}},\ q^m_{i-1,j_{\text{red}}},\ q^{m-1}_{i,j_{\text{red}}},\ q^m_{i,j+1_{\text{red}}},\ q^m_{i,j-1_{\text{red}}}). \tag{26}$$

Another approach for updating the black values is to use the red values from the previous RK step. This, however, would yield exactly the Jacobi RK method and its associated slower convergence rate and larger storage requirement. An overhead is associated with this method which is incurred by the logic needed to drive the red and black grid calculations.

Domain decomposition. To parallelize the Gauss–Seidel RK method while still retaining performance, or to better utilize the available concurrency of the modified Gauss–Seidel RK method, a parallel Schwarz domain decomposition method[15–17] can be used. Here the computational domain is first divided into a series of overlapping subdomains Ω_i and overlap boundaries Γ_i as shown in Figure 1(f). Then, to obtain concurrency, the subdomains are distributed among available processors. To achieve proper load balancing, the field is split such that, to the largest degree possible, the domains are distributed evenly among processors, the same number of grid points is assigned to each processor and the domains are of similar shape. During the solution process an update method is used to obtain an interior solution on each individual subdomain. If vectorization is not desired, Gauss–Seidel RK can be used to obtain the best convergence rate. If vectorization is to be used, the modified Gauss–Seidel RK update method can be utilized. Jacobi RK or red–black RK could also be used, but the former's poor convergence and the logic required to drive the latter on subdomains make it inadvisable.

The implementation of domain decomposition as shown in Figure 1(f) is straightforward on the Cray systems, since do-loops which are to be run concurrently are by default broken into chunks. For example, if during the RK residual calculation the do-loop which increments the grid index i is made the outer do-loop, the grid will be broken into subdomains as in Figure 1(f) by default. On systems that use stride-based parallelism this is not possible. Instead, the portion of the code in which the residual is calculated is placed in a recursive subroutine: during concurrent operation this recursive subroutine is distributed to all processors for execution. The call to the recursive subroutine is written such that each processor operates only on the appropriate portion of the grid and domain decomposition is obtained.

With reference to boundary conditions the outer boundary is updated in the usual manner. For the overlap boundaries Γ_i the interior solution on each individual subdomain is used as the overlap

boundary data in adjacent subdomains. This is done automatically on the shared memory machines utilized in the present work. It is possible for memory conflicts to occur, though this was not found to be a problem. In addition, owing to asynchronous operation of the separate processors, it is not clear whether the information used for the overlap boundary data is from the current or previous RK step. Hence the scheme is really an asynchronous parallel update method in this sense.

Remarks. Note that in the above discussion it was assumed that as the solution calculation progressed over the grid, an i-direction sweep was performed before an advance was made in the j-direction. Obviously this is not essential to the discussion and all statements made will hold if sweeps are made in the j-direction before an advance is made in the i-direction. Alternating direction schemes are also admissible.

Clearly the process of updating a grid point value using surrounding grid point values is one that is encountered in many iterative techniques for sparse finite difference, finite volume or finite element equation solvers as well as other algorithms. Thus the concepts underlying the above schemes are applicable to many types of numerical solution procedures and the results following are indicative of performance characteristics in general.

6. CASCADE FLOW PROBLEM

As a representative test problem we consider steady incompressible viscous flow into a semi-infinite cascade of thin plates indicated by the periodic pair of plates in Figure 2. Symmetry is used to reduce the domain to the upper half of the flow field in the figure. For this problem the Reynolds number is 500 and the discretized domain is the region $-2{\cdot}5 \leqslant x/h \leqslant 5$, $0 \leqslant y \leqslant 1$. The vertical grid lines are clustered about the leading edge co-ordinate of the cascade ($x = 0$), while the horizontal grid lines are clustered towards the upper boundary ($y = 1$). The clustering is achieved using a geometric series in the x-direction and a sine function in the y-direction. For a typical computation a 150×51 grid is utilized with a minimum increment between grid lines of order 10^{-3}.

In the following three figures the scale in the vertical (y) direction has been amplified by a factor of five for clearer presentation of results. The boundary conditions required for the solution of the

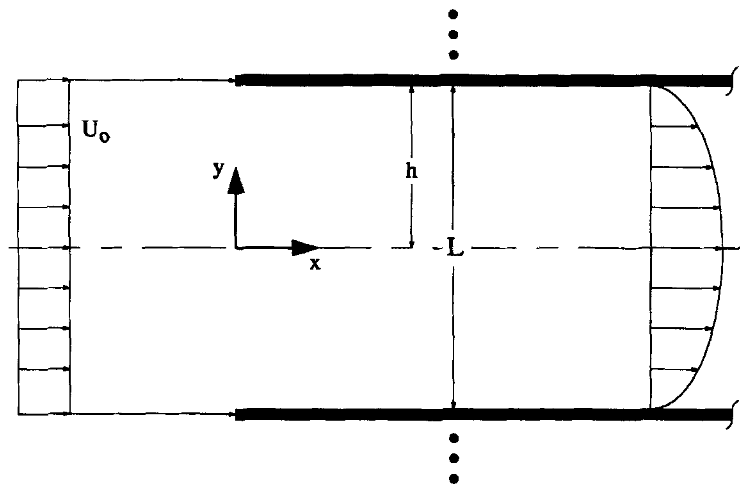

Figure 2. Periodic geometry of semi-infinite cascade of thin plates

streamfunction and vorticity equations are given as values of the streamfunction, vorticity and velocity. The value of vorticity at the walls and the exit as well as the values of the streamfunction at the exit are recalculated after every step of the Runge–Kutta four-stage method.

An example of the type of flow calculated by the solver is given by the streamfunction, vorticity and pressure coefficient contours shown in Figure 3. In Figure 3(a) the initially parallel streamlines turn towards the centreline of the cascade as the plate is approached. This behaviour is caused by the

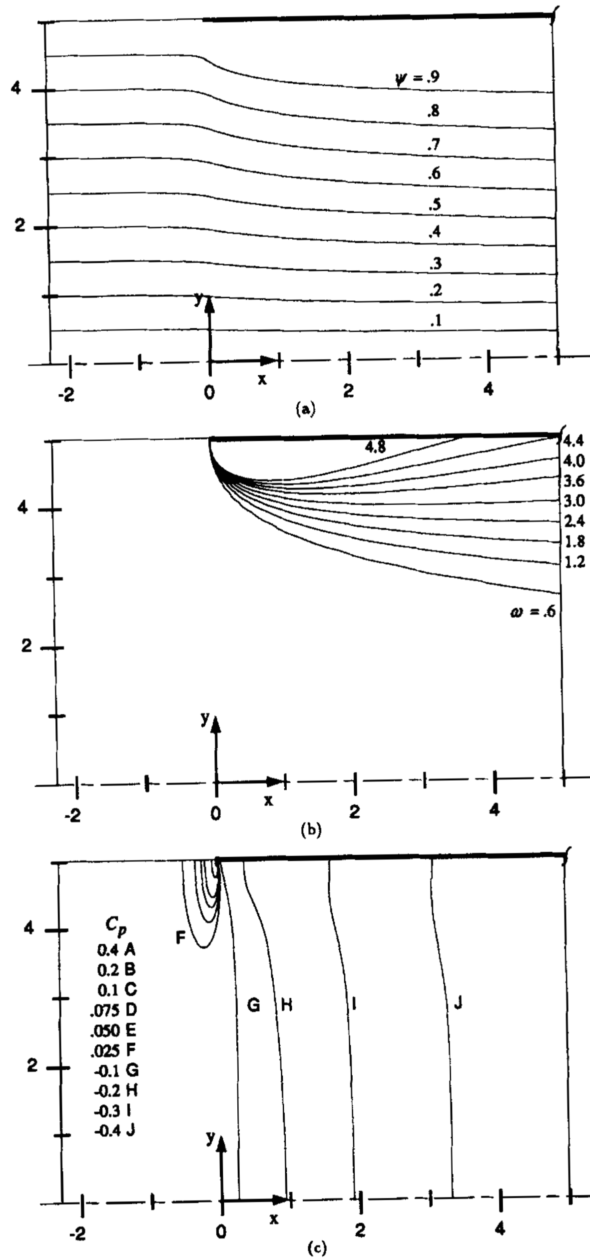

Figure 3. Streamfunction (a), vorticity (b) and C_p (c) contours for incompressible flow through entrance of cascade of thin plates at $Re = 500$. $x:y = 1:5$

growing boundary layer, so that the effective channel section resembles that for a converging nozzle which accelerates the subsonic flow. Vorticity in the flow is generated at the plate's leading edge and swept downstream by the flow as seen in Figure 3(b). For reference the pressure coefficient (C_p) contours calculated by solving the pressure Poisson equation are shown in Figure 3(c). These pressure contours in Figure 3(c) clearly show the high pressure gradient at the leading edge of the thin plate and the spread of the contour lines caused by the growing boundary layer. From the contours along the wall we see that the familiar boundary layer theory approximation of $\partial p/\partial y = 0$ inside the boundary layer is validated, except near the leading edge, as expected.

7. PERFORMANCE RESULTS

In this section the results obtained through the utilization of vectorization and concurrency with the Runge–Kutta solver are presented for the cascade test problem. First, for each update method the iterations needed to achieve convergence and the execution time required are presented. Next, from the Cray Y-MP and C-90 studies, results in the form of gigaflop rate versus number of processors used and gigaflop rate versus problem size are given. Finally, the reduction in wall clock time achieved by using domain decomposition is contrasted with the corresponding increase in iterations needed to achieve convergence.

The four update methods developed in Section 5.1 as alternatives to the Gauss–Seidel RK method were first run in three modes: scalar uniprocessor, four-processor concurrent and four-processor vector–concurrent. As a baseline for calculating the relative effectiveness of each method, the Gauss–Seidel RK update method was used. All calculations were performed until a converged solution was achieved based on a tolerance of 10^{-8} for the steady state relative residual. The comparative performance of these computations for all the methods is presented in Table I and Figure 4. The Jacobi RK update method clearly has the worst performance, requiring 77% more iterations than the baseline Gauss–Seidel RK update. The modified Gauss–Seidel RK, red–black RK and domain decomposition update methods are seen to be equal, within a few percentage points, with the modified Gauss–Seidel RK having the best performance with only a 17% degradation in convergence behaviour over Gauss–Seidel RK. Note that the iteration count for all methods except domain decomposition is independent of concurrency and vectorization. For domain decomposition the scalar uniprocessor and scalar concurrent runs utilized the Gauss–Seidel RK update, with the scalar concurrent runs showing a degradation in convergence due to the use of subdomains. In order to allow vectorization, the vector–concurrent domain decomposition computation employs modified Gauss–Seidel RK, causing further convergence degradation as discussed in Section 5.1. The iteration value for domain decomposition given in Table I is for the vector–concurrent run as it is the highest of the three examined.

The execution time advantages achieved by using concurrency and vector–concurrency are clearly seen on studying the CPU results shown in Figure 4. When comparing concurrent and vector–concurrent results with the scalar result for each update method, we see that the execution time reductions are quite large. On average the concurrent mode achieved a 73% reduction in CPU time over scalar execution, while the vector–concurrent mode achieved an 84% reduction, with the Jacobi RK update giving the best vector–concurrent speed-up. The percentage reductions are much less when compared with the CPU time for Gauss–Seidel RK. The average concurrent mode reduction is only 64%, while the average vector–concurrent mode reduction is only 79%, with Jacobi RK giving the worst speed-up. From these results an important observation can be made: speed-up and performance gains which are achieved by increased calculation rates may be misleading, since the actual CPU performance may be reduced by the requirement to do more iterations or other calculations. Therefore comparing an algorithm's vector–concurrent performance against its scalar single-processor

Table I. Iterations required for convergence—Alliant

Update method	Gauss–Seidel RK	Jacobi RK	Modified Gauss–Seidel RK	Red–black RK	Domain decomposition
No. of iterations	1175	2083	1375	1401	1424
Increase over Gauss–Seidel	–	77%	17%	19%	21%

performance is not a true measure of algorithm effectiveness. Performance should also be measured against that of the most efficient scalar single-processor algorithm to obtain a more viable measure of parallel enhancement.

Finally, from the CPU times given in Figure 4 it is seen that of the four vector concurrent methods the domain decomposition update method provides a solution in the shortest CPU time. The method is seen to give a reduction in execution time of 71% over the Gauss–Seidel RK method when concurrency is used and 82% when both vectorization and concurrency are used. Note that although the scalar uniprocessor domain decomposition calculation is equivalent to the baseline Gauss–Seidel RK calculation (since there is only one domain when one processor is used), the CPU time given in Figure 4 is slightly higher owing to the overhead of the subroutine calls needed to implement domain decomposition.

7.1. Cray Y-MP and C-90 results

The purpose of the calculations on the Cray Y-MP was to create a suitable algorithm and program which can, without sacrificing convergence performance, execute at a sustained rate of over 1 Gflop on reasonable meshes. On the C-90 the goal was to achieve one-half machine speed (7·76 Gflops), again retaining convergence performance and using a reasonable mesh size. This level of performance will facilitate non-linear analysis and design studies where numerous solves may be needed. Domain decomposition was chosen for the update strategy for two reasons. First, as seen previously in Figure 4, it provided a solution in the shortest CPU time. Secondly, the default manner in which the Cray's fpp

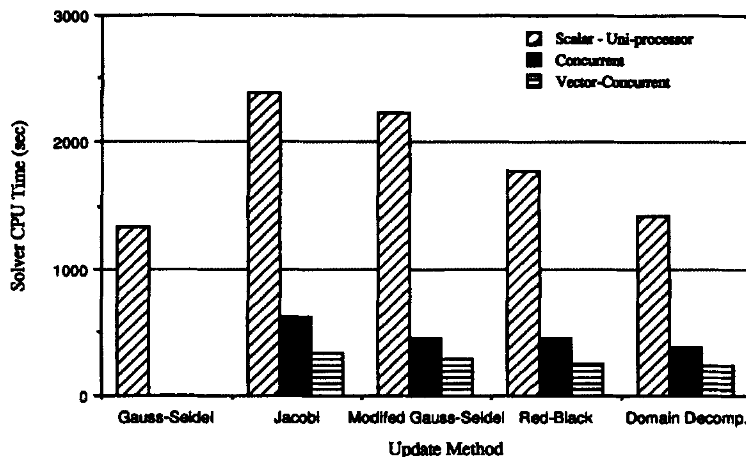

Figure 4. Total solver CPU time needed for convergence—Alliant

Fortran preprocessor creates parallel code is ideally suited for domain decomposition, requiring no additional logic to be implemented (as would be needed for red–black RK). Furthermore, this approach allows the use of an efficient parallel outer loop with vectorized inner loop. This optimization method and modified Gauss–Seidel RK cannot be utilized together unless domain decomposition is used.

Table II contains sustained execution rates and speed-up results obtained on the Y-MP using a 150×51 grid. Using eight processors, a sustained execution rate of 1·1 Gflops is obtained. Table III contains analogous results for the C-90 using a 256×256 grid. Here the sustained execution rate on 16 processors is 7·76 Gflops.

From the speed-up data an equivalent estimate of parallelism, the percentage parallelism, can be inferred by means of Amdahl's law.[18,19] The percentage parallelism refers to the percentage of the total execution time during which calculations are being done in parallel. Using the speed-up values in Tables II and III, average values of 96% and 99% parallelism are obtained respectively. A plot of the speed-up factors from Table II as a function of the active number of processors is shown in Figure 5. As expected, when the number of processors is progressively increased and the problem size remains fixed, the execution time of the parallel sections of code decreases until the execution time of the serial portions becomes more significant, causing the speed-up to degrade. The sublinear behaviour is also seen in Figure 6, where speed-up results for all runs made on the C-90 are given. Conversely, the speed-up, and hence the percentage parallelism, increases with problem size. This is clearly seen in Figure 7, where the effect of problem size on percentage parallelism is shown for all grid sizes using both eight and 16 processors. Here a high degree of parallelism is reached quickly when eight processors are employed, but more slowly when 16 are used, as larger problem sizes are needed to provide sufficient amounts of parallel work for all available processors. The upward trend in percentage parallelism with problem size indicates that the parallel overhead in the problem increases at a slower rate than the problem size.

The effect of increasing problem size on the execution rate of a parallel program when the number of processors is fixed is displayed in Figure 8 for the Y-MP and Figure 9 for the C-90. It is seen that the performance rate initially increases quickly with problem size, then levels off asymptotically. Here the maximum sustained rates are 1·322 Gflops for the Y-MP, 4·36 Gflops for the C-90 using eight processors and 8·15 Gflops for the C-90 using 16 processors. The increase in performance with problem size is due to the fact that the serial portion of the code which cannot be parallelized does not grow at the same rate as the problem size. This can be justified in the following way. When dividing a

Table II. Megaflop rates as a function of processors. 150×51 nodes—Cray Y-MP

No. of CPUs	1	2	3	4	5	6	7	8
Megaflops	177	334	479	624	752	881	1030	1100
Speed-up	1·0	1·9	2·7	3·5	4·2	5·0	5·8	6·2

Table III. Megaflop rates as a function of processors. 256×256 nodes—Cray C-90

No. of CPUs	1	8	16
Megaflops	558	4200	7760
Speed-up	1·0	7·53	13·92

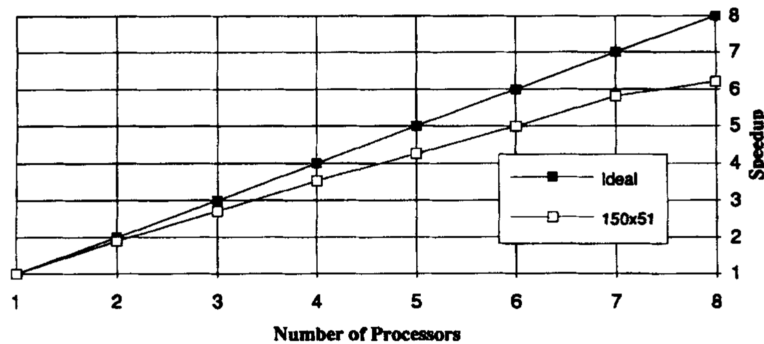

Figure 5. Speed-up obtained versus processors used. 150 × 51 nodes—Cray Y-MP

program into tasks for parallel execution, let the execution time E be defined as the length of time required to execute a program segment on a single processor. This time can be divided into a serial portion s which cannot be parallelized and a parallel portion p which can (i.e. $s+p=E$). Doing this, the effect of the serial portion on speed-up S can be modelled as

$$S=\frac{E}{s+p/N}, \tag{27}$$

where N is the number of parallel tasks. Note that if we normalize by the scalar execution time by setting $E=1$, s and p become the serial and parallel percentages respectively and we obtain the parallel processing version of Amdahl's law. If the problem size is fixed and we increase N, the speed-up approachs $1/s$. If instead the problem size is increased while both N and the serial and parallel fractions remain constant, then no increase in speed-up is realized. However, if as the problem size increases the serial percentage drops while the number of parallel tasks remains constant, the speed-up increases asymptotically towards N as the execution time E increases.

This is what occurs in the current problem, as can be seen in Figure 10, where the 16-processor speed-up is plotted as a function of problem size for the C-90. Since the serial portion of the code does not go to zero for large grid sizes, the ideal speed-ups of eight on the Y-MP, or approximately 1·4 Gflops, and 16 on the C-90, or approximately 8·9 Gflops, are not obtained. However, the serial fraction at the largest grid sizes is extremely low (06% for the C-90) and maximum performance is nearly obtained. With the serial fraction so low the machine is performing the calculations nearly as

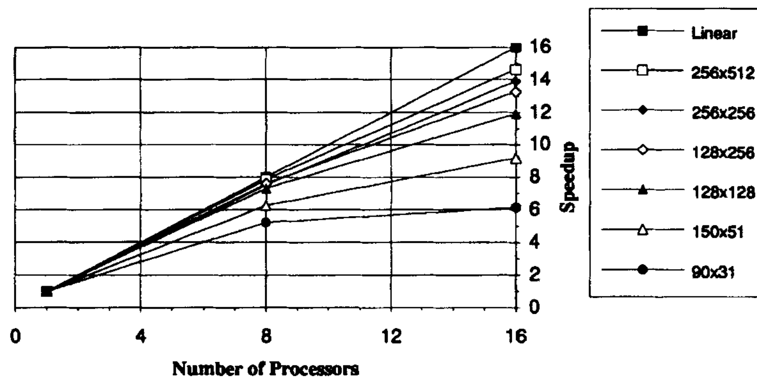

Figure 6. Speed-up obtained versus processors used. All grid sizes—Cray C-90

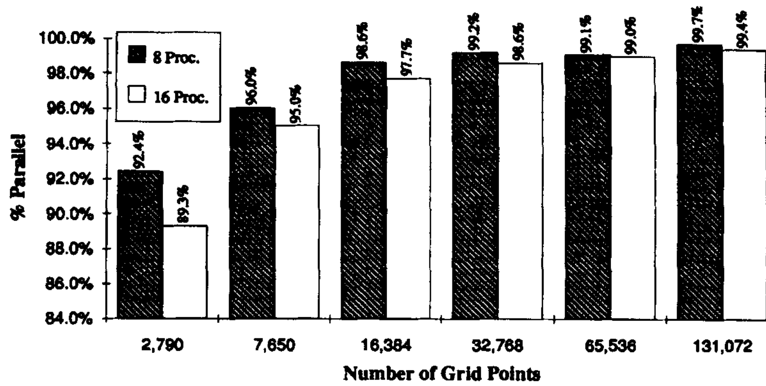

Figure 7. Percentage parallelism as a function of grid size. All grid sizes—Cray C-90

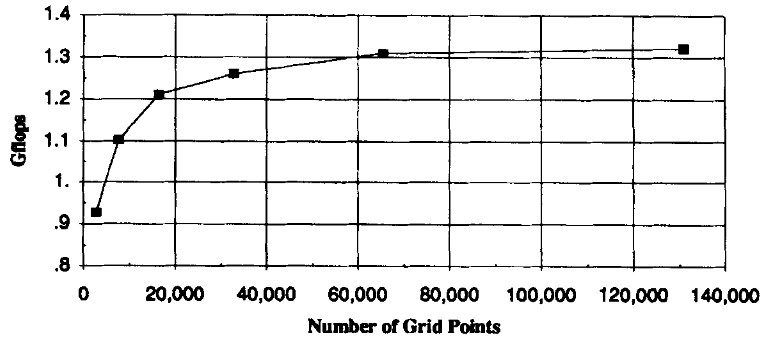

Figure 8. Eight-processor gigaflop rate versus grid size—Cray Y-MP

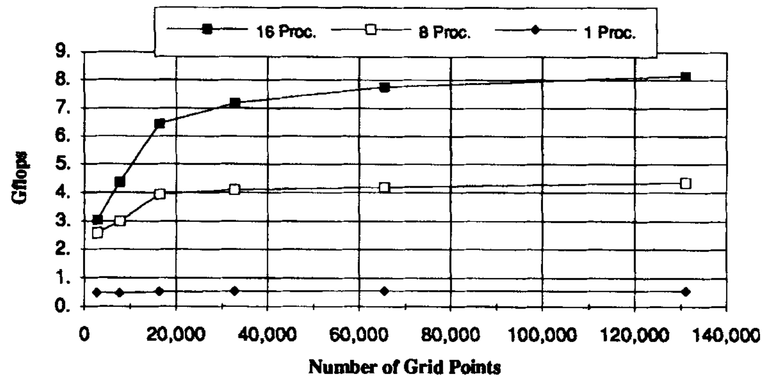

Figure 9. One-, eight- and 16-processor gigaflop rate versus grid size—Cray C-90

fast as code-dependent factors such as memory access and vectorization efficiency will allow. As this occurs, increases in performance can only be obtained through the use of additional processors.

The effect of domain decomposition on both convergence rate and wall clock time as a function of the number of processors is compared in Figure 11. Here a fixed grid size of 150×51 nodes was used and the number of domains was equal to the number of available processors. As mentioned in Section 5.1, when implementing the domain decomposition strategy, various factors may cause degradation of the convergence rate. One such factor is the use of the solution in one subdomain as boundary data in an adjacent subdomain. When this is done, it is possible for old data (referring to the definitions of 'old' and 'new' data given in Section 5.1) to be used in a calculation instead of new data which would have been available had domain decomposition not been used. Another factor is simply the change in the manner in which the update procedure traverses the grid caused by the use of subdomains. Such effects are common when implicit iteration techniques such as line successive over relaxation (LSOR) are used. For example, in a fluid flow problem, when the flow is predominantly in a single direction, convergence is considerably faster when implicit updates are performed along grid points which lie in the flow direction.

In Figure 11 it is seen that the number of iterations needed to converge increases with the number of processors used, with a 5% increase occurring when eight processors are used. In contrast, the wall clock time needed to converge decreases as the number of processors increases, with an 80% decrease in wall clock time obtained with eight processors despite the iteration increase.

The step size in the method is determined from a simplified stability estimate that in turn depends on the mesh size (since the extreme eigenvalues depend on the mesh). If the grid is coarsened, then the step size can be increased and convergence to the steady state is accelerated. Spatial accuracy on the coarse grid will of course be inferior. This coarse grid solution can be used as an iterate for the next fine grid calculation and so on through a succession of nested grids. One can similarly apply multigrid acceleration to the procedure.

The previous example was computed on a sequence of grids with mesh sizes 20×8, 40×15, 80×29 and 160×57. The convergence history for this case is shown in Figure 12. A full solution on the fine grid, shown in Figure 13, required 5645 iterations to converge compared with 2659 for the nested grids. In addition, the coarse grid solutions require substantially less CPU time per iteration as compared with the fine grid. An approximation to the CPU time measure is found by comparing the sum of the times the grid points are 'visited' for each method. The nested grid has $14{\cdot}16 \times 10^6$ visits while the non-nested calculation requires $51{\cdot}4 \times 10^6$ visits. Therefore using nested grids leads to an overall reduction of 72%.

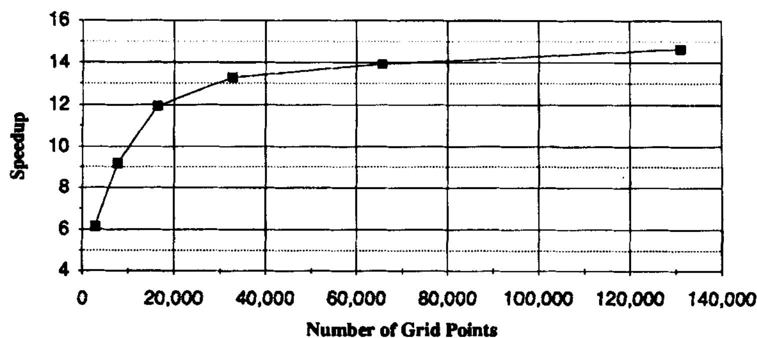

Figure 10. Sixteen-processor speed-up versus grid size—Cray C-90

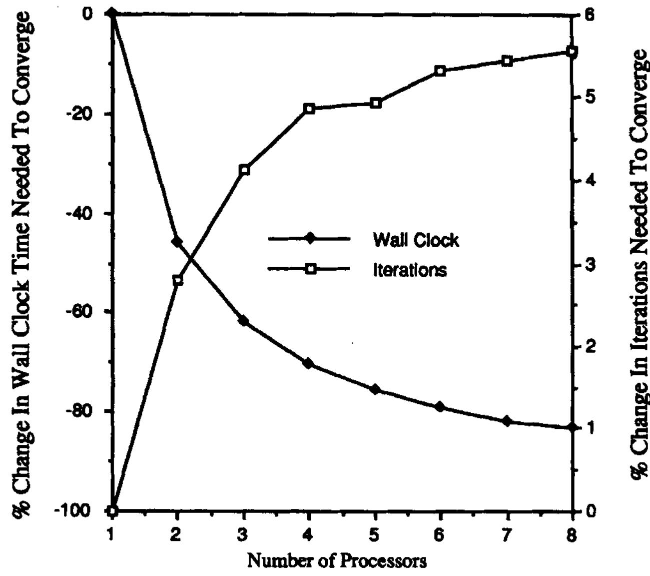

Figure 11. Percentage change in iterations and wall clock time needed to converge when compared with a single-processor run. Domain decomposition using modified Gauss–Seidel RK. Fixed grid size (150 × 51)—Cray Y-MP

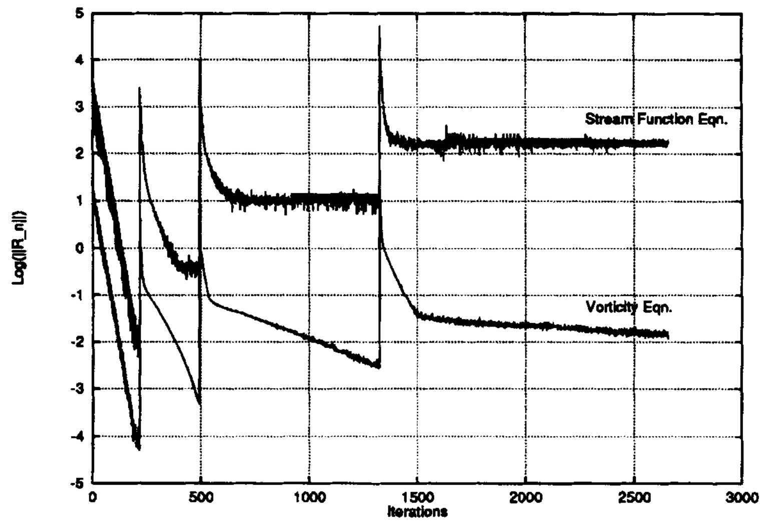

Figure 12. Convergence history for nested grids: 20 × 8, 40 × 15, 80 × 29, 160 × 57

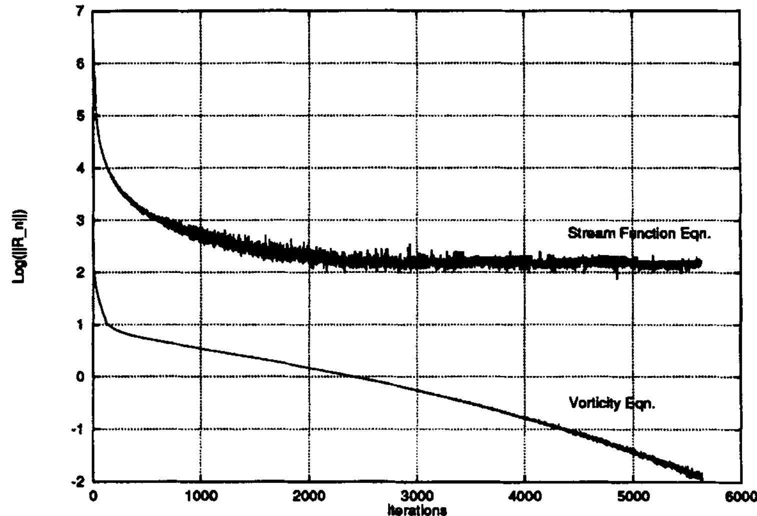

Figure 13. Non-nested convergence history for 160 × 57 grid

8. CONCLUDING REMARKS

In this study, four procedures—Jacobi update, a hybrid of the Jacobi and Gauss–Seidel methods, red–black ordering and domain decomposition—are used to restructure a four-stage Runge–Kutta scheme, permitting the vector–parallel solution of compressible and incompressible flows. The RK scheme is accelerated by parameter selection and local time stepping to yield a fast time-iterative type of recursion. Using performance results obtained from the solution of a representative incompressible viscous flow problem to compare the procedures, it is found that a combination of the domain decomposition method and the hybrid Jacobi/Gauss–Seidel method produces a scheme which converges in the shortest CPU time and which is easily implemented using the Cray Fortran compiler. The resulting Navier–Stokes solver is highly parallel and is capable of a sustained performance on a reasonable grid in excess of 1 Gflop on an eight-processor Cray Y-MP and exceeding 8 Gflops on a 16-processor Cray C-90.

In the performance studies it is found that computational optimization of a program through the use of vectorization and concurrency can produce large gains in performance when the execution time of a single algorithm is considered. However, when optimized results are compared with scalar uniprocessor results of another algorithm, the large gains may be significantly reduced. As expected, the effects of residual scalar code are found to reduce parallel performance gains as the number of processors is increased. Finally, by increasing the problem size, it is possible to achieve impressive performance rates on a grid which is unreasonably large and hence obtain flop rates that may not be representative for engineering applications. Because of these factors, the coupling of a vector–parallel technique and a solution algorithm must be performed judiciously. When this is done, extremely high performance rates can still be achieved on grids of realistic size as demonstrated here. Other strategies such as nested mesh refinement and multigrid ideas can be combined with the present strategy to considerable advantage.

In future phases of the research we plan to utilize the present high-performance Navier–Stokes solver within an optimization strategy for shape modification and design enhancement. Inherent to optimization is the requirement to make numerous solves. Thus the amount of wall clock time required to obtain a design depends heavily on the rate at which the flow simulation calculations can be performed. The

present vector–parallel solver will give us the high rates necessary to obtain final design results efficiently. We are also currently exploring the extension of the scheme to distributed parallel systems.

ACKNOWLEDGEMENTS

This research has been supported in part by The Texas Advanced Research Project, DOE (Grant DE FG05-87ER25048), ARPA (Grant DABT63-92-C-0024) and Cray Research, Inc.

REFERENCES

1. P. J. Denning, 'The science of computing–parallel computation', *Am. Sci.*, **73**, 322–323 (1985).
2. M. J. Flynn, 'Some computer organizations and their effectiveness', *IEEE Trans. Comput.*, **C-21**, 948–960 (1972).
3. A. A. Lorber, G. F. Carey and W. D. Joubert, 'ODE recursions and iterative solvers for linear equations', *SIAM J. Sci. Comput.*, in press.
4. D. A. Anderson, J. C. Tannehill and R. H. Pletcher, *Computational Fluid Mechanics and Heat Transfer*, Hemisphere, New York, 1984.
5. A. J. Chorin, 'A numerical method for solving incompressible viscous flow problem', *J. Comput. Phys.*, **2**, 12–26 (1967).
6. C. L. Merkle and M. Athavale, 'Time-accurate unsteady incompressible flow algorithms based on artificial compressibility', *AIAA Paper 87-1137*, 1987.
7. F. H. Harlow and J. E. Welch, 'Numerical calculation of time-dependent viscous incompressible flow of fluid with free surface', *Phys. Fluids*, **8**, 2182–2189 (1965).
8. P. J. Roache, *Computational Fluid Dynamics*, Hermosa, Albuquerque, NM, 1982.
9. A. A. Lorber, 'Numerical solution of a streamfunction vorticity formulation for compressible and incompressible flows', *M.S. Thesis*, Department of Aerospace Engineering, Penn State University, University Park, PA, 1989.
10. C. W. Gear, *Numerical Initial Value Problems in Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1971.
11. A. Ralston and P. Rabinowitz, *A First Course in Numerical Analysis*, 2nd edn, McGraw-Hill, New York, 1987.
12. A. Jameson, 'Numerical solutions of the Euler equations by finite volume methods using Runge–Kutta time-stepping schemes', *AIAA Paper 81-1259*, 1981.
13. K. Dowd, *High Performance Computing*, O'Reily, Sebastopol, CA, 1993.
14. D. M. Young and L. A. Hageman, *Applied Iterative Methods*, Academic, New York, 1981.
15. G. F. Carey, 'Parallel sub-domain and element-by-element techniques', in G. F. Carey (ed.), *Parallel Supercomputing: Methods, Algorithms and Applications*, Wiley, Chichester, 1989, pp. 57–75.
16. G. Rodrique and S. Shah, 'Pseudo-boundary conditions to accelerate parallel Schwarz methods', in G. F. Carey (ed.), *Parallel Supercomputing: Methods, Algorithms and Applications*, Wiley, Chichester, 1989, pp. 77–88.
17. T. F. Chan, R. Glowinski, J. Periaux and O. Widlund (eds.), *Domain Decomposition Methods for Partial Differential Equations*, SIAM, Philadelphia, PA, 1989.
18. G. M. Amdahl, 'Validity of the single processor approach to achieving large scale computing capabilities', *Am. Fed. Info. Process. Soc. Conf. Proc.*, **30**, 483–485 (1967).
19. G. M. Johnson, 'Parallel processing in fluid dynamics', in O. Baysal (ed.), *Applications of Parallel Processing in Fluid Mechanics*, ASME, New York, 1987, pp. 14–17.