Implementation of a Rotary-Wing Navier-Stokes Solver on a Massively Parallel Computer

Brian E. Wake* and T. Alan Egolf†
United Technologies Research Center, East Hartford, Connecticut 06108

An unsteady, compressible, three-dimensional, implicit Navier-Stokes solver (NSR3D) for helicopter and propeller applications has been implemented using FORTRAN with 8X array extensions on the massively parallel connection machine (CM-2). In this paper, the modifications to the original algorithm necessary to overcome communication bottlenecks and achieve reasonable computational efficiency on the CM-2 are described. The modified implicit solver achieves better than twice the speed of a CRAY-2 processor on a 16384 processor CM-2. The CM-2 and FORTRAN 8X array extensions, including coding examples, are briefly described. Some programming issues for difficult problems such as solving the linear systems, the boundary conditions, and the dissipation switching are discussed. Results for a selected application are also provided.

Introduction

FOR rotary-wing applications, a detailed analysis of the flowfield requires a three-dimensional solver that includes viscous and transonic compressibility effects. In the tip region of helicopter blades, a tip vortex is formed through viscous mechanisms. Adequate prediction of the vortex development and shedding requires an unsteady three-dimensional Navier-Stokes solver. For Propfan applications, the leading-edge vortex is a fluid phenomenon of interest. The prediction of these phenomena on a routine daily basis will require orders-of-magnitude improvement in computer performance.

The processor speeds of vector supercomputers seem to be approaching a plateau and, as a result, interest has grown in parallel computers as a possible means of achieving this performance. Massively parallel computers of the SIMD (Single Instruction, Multiple Data path) type are one family of computer architectures that may provide orders-of-magnitude improvement in computational performance over today's supercomputers. The current connection machine (CM-2) with 65536 processors, described later, is the preeminent example of such a machine.

Although there is a significant amount of research to be done in the area of parallel algorithm development, utilizing the capabilities of a massively parallel computer, such as the CM-2, is not as difficult as perceived. It may appear to be extremely difficult, if not impossible, to efficiently implement an implicit flow solver on a massively parallel computer with such nonparallel features as matrix inversions, realistic boundary conditions, dissipation switching, and an eddy-viscosity model. However, as this paper demonstrates, better than CRAY-2 speeds have been realized in solving the Navier-Stokes equations on the CM-2 using a familiar engineering computer language (FORTRAN), with only a few significant modifications to the original algorithm.

Although no standard yet exists for FORTRAN 8X, a compiler was developed for use on the CM-2 that supports the proposed FORTRAN 8X array extensions. For the purpose of brevity, the term FORTRAN as used on the CM-2 will mean FORTRAN with 8X array extensions. An overview of scientific applications under development on the CM-2 by these authors and their co-workers is provided in Ref. 1. For other examples of computational fluid dynamics (CFD) on the CM-2, see Refs. 2–6.

This paper summarizes the results of a research effort to implement a three-dimensional Navier-Stokes solver on the CM-2 at CRAY-like speeds with a minimal number of algorithmic changes. Toward this goal, a mature and well-vectorized Navier-Stokes solver for rotary-wing applications (NSR3D) was recoded using the 8X array extensions with initially no fundamental changes to the algorithm. After the virtually direct conversion on the CM-2 was achieved, significant computation inefficiencies were noted and resolved. The remaining sections describe in more detail the CM-2, the FORTRAN 8X array extensions, and the algorithm modifications (including example code fragments) that enabled such impressive speeds to be obtained on the CM-2. Some of this work was originally described in detail in Ref. 7. A significant enhancement to the linear system solver has recently been implemented. The corresponding new timings, along with some applications of the solver, were presented in Ref. 8.

Connection Machine

The connection machine model CM-2 is an integrated system of hardware and software for data parallel computing[9,10] that associates one processor with each data element. The hardware consists of a SIMD parallel processing unit containing up to 65,536 physical processors (64 k) in blocks of 8 k (where "k" stands for 1024). Each processor has its own local memory of 2048 32-bit words (expandable to 8192 32-bit words). The system can have one to four front-end computers. The CM-2 system software is designed to utilize existing programming languages and environments as much as possible and parallel versions of LISP, C, and FORTRAN are available. The front-end operating systems, UNIX or LISP, remain largely unchanged. The high-level assembly-like language of the CM-2 is PARIS and is the target language of the high-level language compilers. The CM-2 used for this study has 16 k processors (one quarter of a CM-2) and is front-ended by a DEC 6320 computer. An I/O system supports parallel mass storage and high-speed graphic display devices. Parallel data structures are spread across the data processors, with a single element stored in each processor's memory. When the number of parallel data elements exceeds the number of physical processors (the normal case), the hardware operates in virtual processor mode, effectively presenting the user with a larger number of processors, each with a correspondingly smaller memory. The ratio of virtual to physical processors is known as the virtual-processor ratio (vp ratio). A 64 k processor CM-2 can perform at an aggregate peak of about 4 GFlops

Received May 19, 1989; revision received Feb. 1, 1990. Copyright © 1990 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved.

*Research Engineer. Member AIAA.
†Senior Principal Engineer. Member AIAA.

using single-precision hardware with the high-level parallel language, PARIS, the target language of the FORTRAN compiler. (The speed for double precision is halved using double-precision hardware.) Even higher performance is possible by microcoding these floating-point processors and special cases have exceeded 20 GFlops.

The CM-2 system provides two forms of interprocessor communication. The general mechanism, the router, allows any processor to communicate in parallel with any other processor. The average time to send data through the router is equivalent to 60 integer-add operations. A more structured, local communication mechanism, the NEWS grid, allows processors to pass data in parallel in a regular multidimensional pattern, and communication with adjacent processors is about 6 integer-add operations. There is also a special power-of-two NEWS grid that makes optimal use of the underlying communication hardware, when the distance between communicating processors is an integer power of two.

CM FORTRAN

Connection machine FORTRAN (CM FORTRAN) consists of the incorporation of proposed FORTRAN 8X array extensions[11] to the ANSI FORTRAN 77 standard. The most important enhancement is the treatment of entire arrays as objects. These array extensions are logical features that map fine-grain parallel problems naturally onto the underlying data parallel hardware of the CM-2. In practice, the 8X array extensions have proven to be very convenient for numerically intensive codes. One disadvantage of the 8X array extensions is that temporary variables become temporary arrays; hence, on the CM-2, the memory requirements are larger than on traditional supercomputers.

The best way to explain and acquire the "flavor" of these array extensions is by code fragment examples. Some of the more fundamental operations can be categorized as elemental, data reduction, and complex operations. An example that demonstrates some of the operations is given below:

```
PARAMETER (N=128,M=64)

REAL R(N),A(N,M),B(N,M),C(N,M),D(N,M)

•

C = A + B

D = ALOG(C)

•

S = SQRT( SUM( R * R ) ) / FLOAT( N )

RM = MAXVAL( ABS( R ) )
```

In the first segment of code, each element of C will contain the sum of the corresponding elements of A and B, and D contains the logarithm of C. In FORTRAN 8X, A, B, C, and D can be any set of conforming scalars, vectors, or multidimensional arrays. From this code segment, the reader may have correctly inferred that all mathematical operations and traditional FORTRAN intrinsic functions now work on complete arrays on an element-by-element (elemental) basis. These arrays are stored as one element for every (virtual) processor. In the second segment of the example, the reduction operations to compute the sum of all elements and the maximum value of an array are demonstrated. Here R*R produces the elemental square of the elements of R, and the function SUM then computes the sum of these squares (a reduction operation). The resultant S is the root mean square of all elements of R, while the scalar RM contains the maximum element of the elemental absolute values of R. Many additional reduction operations, including more complex operations such as matrix multiplication, are available in FORTRAN 8X; see Ref. 11 for more details.

An important language feature of FORTRAN 8X is the ability to move data within arrays, and hence between processors on the CM-2. Since, in many cases, calculation of values on all elements of an array structure may not be of interest, new concepts for control of the calculation on arrays have been introduced in FORTRAN 8X. The only two discussed here are the WHERE statement and array sections along with one type of FORTRAN array-movement function.

```
  PARAMETER (L=128,M=64)

  REAL F(L,M), RES(L,M)

  LOGICAL INTERIOR(L,M)

  •

C Set mask at interior nodes to true.

  INTERIOR                  = .FALSE.

  INTERIOR(2:L-1,2:M-1)  = .TRUE.

  WHERE (INTERIOR)

C Calculate residual at the interior nodes.

   RES = CSHIFT(F,1,-1) + CSHIFT(F,1,+1)

  1  + CSHIFT(F,2,-1) + CSHIFT(F,2,+1) - 4.0 * F

  ELSEWHERE

C Set residual to zero at the grid edges.

   RES = 0.0

  ENDWHERE
```

Here the CSHIFT function is the circular shift of the data in the specified array (the first argument) along a specified axis of the array (the second argument) for a specified displacement (the last argument). This particular example calculates the residual (RES) of a five-point stencil for a simple Laplace formulation on the interior of a uniform two-dimensional grid. Note that the circular shift wraps along the specified axis the last value in the array to the first, or vice versa, depending on the specified direction. The use of CSHIFT(F,2,-1) is equivalent to the use of F(I,J-1) embedded inside a two-level DO loop over all I and J values of the array F (without the wrap-around feature).

To avoid the calculation of the residual using the above stencil at the boundaries, a program-defined mask, INTERIOR, is used as an argument to the WHERE block. The mask is a logical array of bits, each bit associated with a single processor, whose context (true or false) determines whether or not the result of the operation of the processor is actually used. The array section construct (colon notation appearing in the array indexing) is used to set a logical bit (mask) stored in the logical array, INTERIOR, to a true value at the interior of the multidimensional grid.

From these simple examples, it should be evident that the FORTRAN 8X array extensions reduce the amount of coding in many instances, and hence the likelihood for programming errors.

Navier-Stokes Solver

The Navier-Stokes solver, NSR3D, is an unsteady three-dimensional Navier-Stokes solver that was originally developed for helicopter applications at Georgia Tech.[12] Over the past few years, the code has undergone further enhancements and has been modified to perform Propfan and fixed-wing calculations as well. The numerical procedure is the alternating direction implicit (ADI) scheme originally developed by Beam and Warming[13] and enhanced for arbitrary coordinate systems by Pulliam and Steger.[14] The spanwise terms, which usually do not drive the stability of the scheme, may be treated explicitly. This is referred to as the hybrid algorithm, developed by Rizk and Chausee.[15] The diagonalized form of the left-hand side is used as originally described by Pulliam and Chausee,[16] and viscous terms and boundary conditions are treated explicitly.

The code treats the boundary conditions explicitly for flexibility and can be used for C-H, O-H, or H-H grid topologies. For turbulence, the algebraic Baldwin-Lomax eddy-viscosity model is used.[17] The CRAY version of the code is mature and very well-vectorized. The equations and details of the original "vector-version" of the algorithms are given in Ref. 12. The details of the diagonalized form of the equations are given in Refs. 7 and 16.

Resulting Systems of Linear Equations

Each sweep of the ADI scheme represents inversions along a coordinate line. During each sweep, a block tridiagonal system is inverted for every point in the other two dimensions. Let ξ, η, and ζ represent the three curvilinear coordinates. In the first sweep (ξ sweep), there are finite differences in ξ only. Thus, a tridiagonal system of equations is formed for which ξ is the inversion direction. Furthermore, there is a tridiagonal equation along ξ for every η and ζ. Similarly, the second sweep (η sweep) involves matrix inversions for all ξ and ζ, whereas the third sweep (ζ sweep) is inverted for all ξ and η. Note that for the hybrid algorithm, there are only two sweeps since the spanwise direction is treated explicitly in time (i.e., no linear system to solve).

Diagonalization of the Left-Hand Side

The Navier-Stokes equations consist of five equations for five unknowns at each grid point. As a result, the linear equations are 5×5-block tridiagonal systems. Inversion of a block tridiagonal system is much more costly than a scalar one and requires more memory. Pulliam and Chausee[16] have described a method of diagonalizing the left-hand side of the equations by reducing the 5×5 blocks to diagonal matrices. Diagonalization decreases the matrix inversion cost dramatically. The cost of diagonalization, although significant, is greatly offset by the reduced inversion cost.

Boundary Conditions

For simplicity, and to give a flexible solution procedure, all boundary conditions are applied explicitly. That is, Δq^{n+1} is taken as zero along all of the boundaries. Then after the ADI procedure has been performed, the boundary values are updated.

At the far-field boundaries, quantities are either fixed to freestream, or extrapolated from the interior, depending on the nature of the flow at the boundary (subsonic vs supersonic, and inflow vs outflow). The characteristic velocities of the Euler equations determine the number of quantities that should be extrapolated from the interior. At the surface of the blade, the no-slip conditions are applied, while the density and pressure are found by specifying their normal derivative to be zero. At the inboard station of the blade, one-sided difference is used to compute the convection terms and the spanwise stress term is turned-off.

The C-grid produced by the sheared parabolic transformation leads to a cut that originates at the trailing edge, and ends at the downstream boundary. Along this cut, the flow properties are averaged from above and below. For the O-grid, periodic conditions are applied behind the trailing edge.

Basic Algorithm Considerations

Development of efficient algorithms on massively parallel machines is still in the research phase, but this does not imply that programming on such a machine is a difficult task. As with vector machines, there are a few straightforward guidelines to follow to achieve efficient use of the computer. Here, the basic principles for developing codes on vector and massively parallel machines, the CM-2 in particular, are summarized and briefly related to issues associated with the Navier-Stokes solver. For the remainder of this paper, the terminology, massively parallel computer, is meant to refer particularly to the CM-2, and generally to massively parallel SIMD computers with enough processors so that each processor can be associated with a grid node in the problem.

Vector Machine

Efficient coding on a traditional vector machine can be conceptually reduced to several simple rules. Basically, these rules require a code to have long DO loops without dependencies, GOTOs, subroutine CALLs, or IF statements.

For Navier-Stokes applications, calculation of the right-hand side of the equations vectorizes naturally since it is a nonrecursive explicit computation. On the other hand, the solution of the linear system of equations is a more involved issue. The lower-upper (LU) decomposition of a single matrix does not vectorize since there are dependencies in the elimination and back substitution. But when employed in the ADI procedure for a three-dimensional problem, an independent system of tridiagonals is solved for every point in the other two dimensions. Thus, for each sweep of the ADI scheme, using LU decomposition, one can say that the linear-system solution algorithm is vectorizable and parallelizable in two dimensions and serial in the inversion direction.

Massively Parallel Machine

Just as there are general rules in vectorizing a code, there are a few important facts to keep in mind when coding for a massively parallel machine. First, performing operations for one point, one line, one plane, or the entire grid, all take the same amount of time. Thus, one should avoid, if possible, treating grid points, lines, or planes individually because doing so will result in many processors sitting idle. A direct consequence of this fact is that the problem under consideration must be large enough to utilize the entire machine. Second, short-range communication is slow relative to the floating-point operations (6 adds) and long-range communication is very slow (60 adds). Therefore, processor communication, especially long-range, must be minimized. With these simple facts, we can define two general rules for efficient programming on massively parallel computers:

1) Try to use *all* processors simultaneously.
2) Try to minimize processor communication.

Obviously, idle processors and communication cannot be avoided entirely for most problems. For an implicit Navier-Stokes solver, there are several instances where these guidelines necessitate very careful coding or algorithmic modifications. Treatment of the boundaries, which correspond to grid lines and planes, will be a violator of the first rule. Linear-system solvers for implicit schemes require a large amount of communication since elements of the matrix must be shifted to other processors to perform the matrix elimination. As a result, large communication costs are difficult to avoid in implicit schemes. Furthermore, LU decomposition is serial in the inversion direction, precluding efficient utilization of the machine.

Thus, from the point of view of implementing an implicit Navier-Stokes solver on the CM-2, other schemes must be considered to reduce communications, at the cost of increased floating-point operations, to achieve high computational efficiency. The following sections will discuss the present implementation of NSR3D on the CM-2 in view of these considerations.

Parallel Implementation

When implementing a code on the CM-2, the first step is to determine which calculations (if any) are to be done on the front-end computer. Presently, the front-end machines that are available for the CM-2 do not approach the computational speed of the CM-2 nor the CRAY. A rough estimate of the performance of the current front-end (DEC VAX 6320) is 1/30th the speed of the CRAY-2. Improper utilization of the front end can destroy any chance of good overall performance.

The purpose of the front-end machine is to serve as a code development site, as a driver of the CM-2, and to perform calculations that would take longer on the CM-2. In addition to floating-point performance issues on the front end, another consideration is the time required to transfer data between the front-end and the CM-2. For most supercomputers with front ends, such as the CRAY, all computations are performed on the supercomputer. On the other hand, with the CM-2, portions of a code may run on the CM-2 while others execute on the front end. Thus, the CM-2 system represents a hybrid computer in which nonparallel (or not fully parallelizable) pieces of the code may be executed on the scalar machine (the front end), while the massively parallel portions run on the CM-2. It is up to the code developer to understand the balance and tradeoff issues on the CM-2 system to produce the fastest running algorithms.

A real-world Navier-Stokes code can be coarsely divided to three phases of execution: initialization/preprocessing phase, iteration or time-stepping phase, and closing and postprocessing phase. The initialization phase obviously belongs on the front end, since it is I/O intensive, generally does not perform fine-grain concurrency tasks, and does not influence the overall run time of the total solver iteration process. After the initialization has been completed, the iterations are performed. Upon completion of the solution iteration process, the solution variables must be transferred back to the front end for postprocessing. For the same reasons, the routines in this phase of execution are well-suited for the front-end computer.

The transmission of data to and from the CM-2 and the front end is relatively fast but it still takes a measurable percentage of the preprocessing and postprocessing time. If performed before and after the solution analysis only, the transmission time is not substantial relative to the time it takes to perform the thousands of iterations required for convergence. However, to develop an efficient computer analysis on the CM-2, it was expedient to clearly isolate the data flows required by the solver from the front end to avoid unnecessary communication between the front end and CM-2 processors.

Some portions of the Navier-Stokes solver, i.e., within the time-stepping loop, are not massively parallel by nature. The best example of this are the boundary conditions. Thus, one may consider performing these calculations on the front end. To date, with the current front ends, this has not been the case. There are scenarios in which front-end calculations within the marching loop may be the best way to go, particularly if faster front ends become available.

Right-Hand Side

Evaluation of the right-hand side of the linear equations includes the convection and stress terms (the spatial portion of the governing equations), the explicit artificial dissipation, and the turbulence model. As mentioned previously, the right-hand side runs very efficiently in parallel since this is an explicit computation. However, there are a few places where careful coding is required, including special differencing at some of the computational boundaries, switching of the dissipation model to second-order, and computing the eddy viscosity.

Convection and Stress Terms

Computation of the convection terms involves evaluating the flux vectors (***E***, ***F***, and ***G***) at all grid-point locations and performing a central difference at all of the interior nodes. One special case is the treatment of the inboard spanwise boundary ($j = 1$) where one-sided differencing is used.

Evaluating the flux vectors at every node is very efficient since all processors will be utilized. In FORTRAN 8X, evaluation of ***F***, for example (for the equations see Refs. 7, 12, or 14), can be coded:

```
C Evaluate the spanwise flux vector everywhere.

      F1 = Q1*U*JACI

      F2 = (Q2*U + ETAX*P)*JACI

      F3 = (Q3*U + ETAY*P)*JACI

      F4 = (Q4*U + ETAZ*P)*JACI

      F5 = ((Q5+P)*U - ETAT*P)*JACI
```

These are of course all three-dimensional real arrays. A mask is not needed since the evaluation can be performed for all nodes. If the actual problem is smaller than the dimensioning of the arrays, calculations will be performed at nodes that are not used, but this does not affect the computation time.

To evaluate the convection term in the spanwise direction, the central difference of ***F*** must be taken for all interior locations and a one-sided difference is taken at the inboard boundary ($j = 1$). The authors believe the best way to do this is to precompute the coefficients of the finite-difference stencil. An example is given below, using FORTRAN 8X.

```
C NOTE: MASK_J1 = TRUE for J=1 only (predefined).

      WHERE( MASK_J1 )

C Use first-order one-sided difference at inboard boundary:

        AP1 = DT

        A00 = -DT

        AM1 = 0.0

      ELSEWHERE

C Use central difference elsewhere.

        AP1 = 0.5 * DT

        A00 = 0.0

        AM1 = -0.5 * DT

      END WHERE

C Now add the spanwise convection terms to RHS:

      RHS1 = RHS1 + AM1 * CSHIFT(F1,2,-1) + A00 * F1
     1   + AP1 * CSHIFT(F1,2,+1)

      •

      RHS5 = RHS5 + AM1 * CSHIFT(F5,2,-1) + A00 * F5
     1   + AP1 * CSHIFT(F5,2,+1)
```

Note that there could also be special treatments at other boundaries or interior points. The point of this example is to show that precomputing the coefficients of a stencil is convenient and an efficient way to code in parallel. The computationally intensive step (i.e., taking the finite difference) is only performed once.

Calculation of the stress terms involves second-order differences. These differences are evaluated about half-points to help prevent odd-even decoupling and produce compact stencils. Unlike the Euler flux vectors ***E***, ***F***, and ***G***, the viscous-flux vectors $\boldsymbol{E}_v$, $\boldsymbol{F}_v$, and $\boldsymbol{G}_v$ require many finite differences in their construction. The stress tensor involves many Cartesian derivatives. To evaluate a Cartesian derivative by the chain rule, three finite differences are required. These finite differences, which represent short-range communications, are not likely to destroy performance. Still, care should be taken to insure that an efficient calculation of the stress terms is used.

As an example, the viscous flux vector in the ζ direction, $\boldsymbol{G}_v$, is evaluated at $(i,j,k\text{-}1/2)$. To construct this term, derivatives in all three directions are required. Derivatives in ζ are simply computed by one-sided differences. However, a term such as u_ξ is given by

$$u_\xi = 0.5^*[(\delta_\xi u)_k + (\delta_\xi u)_{k-1}] \tag{1}$$

which represents the central difference in ξ averaged between k and $k-1$. On the CM-2, time can be saved by evaluating the central difference $\delta_\xi u$ only once. To do this, the central difference is precomputed for all nodes at which it is required and stored in a temporary array. (Since u_ξ is required at all interior nodes, 2 to KMAX − 1, $\delta_\xi u$ is required for $k = 1$,KMAX − 1.) This temporary array is then used to evaluate the average.

At first glance, it may seem reasonable to code Eq. (1) as

```
C NOTE: INSIDE = TRUE for k = 2 to KMAX-1 only.

   WHERE( INSIDE )

C Compute the central difference of U.

   CDIFF = 0.5*(CSHIFT(U,1,+1)-CSHIFT(U,1,-1))

C Average to get value at K-1/2.

   UXI = 0.5*( CDIFF + CSHIFT(CDIFF,3,-1) )

   ENDWHERE
```

but this produces an *error.*

The mask, INSIDE, permits the values of u_ξ, UXI, to be computed at the required points. But there is a problem with the earlier coding. Notice that the temporary, CDIFF, will not be computed at $k = 1$, since it is also under this mask, yet this value will be required to compute $(u_\xi)_{k=2}$. The calculation of CDIFF must be performed outside of this WHERE statement. But had the central difference been explicitly included in the average operation (producing four CSHIFTs), the code would have worked fine. This example illustrates how, when using masks around finite differences and programming for efficiency, one must be very careful that undefined quantities are not used.

Artificial Dissipation

For the explicit dissipation, a blending between second-order and fourth-order dissipation is used.[18] In regions of shocks, second-order dissipation is employed in the ξ direction. Also, in all directions, second-order dissipation is employed adjacent to wall boundaries. Otherwise, standard fourth-order dissipation is used.

By expanding the difference operators into their complete form, the stencil coefficients can be determined as explicit functions of the switching functions described in Ref. 18. Additionally, these coefficients are computed using toggle functions that switch the dissipation to second-order adjacent to the boundaries. On a vector machine, these dissipation terms can be efficiently pieced together by performing a sequence of finite-difference operations. However, on the CM-2, the evaluation of the fourth-order/second-order difference should be grouped into one fully parallel step to use the machine effectively. This is another example where precomputing the coefficients of the stencil is the most efficient method that the authors have found for this problem.

Eddy-Viscosity Model

Coding the eddy-viscosity model is awkward on vector machines as well as massively parallel computers. On the CM-2, the primary source of inefficiency in the eddy-viscosity model is in the calculation of quantities that are defined on only one plane of the grid. Examples of two-dimensional quantities are the skin-friction coefficient, wall stress, and the minimum/maximum of functions along the normalwise direction. A considerable amount of careful coding was required, but no major bottlenecks were observed. In fact, the model performed much better in parallel than anticipated.

Solution of the Linear Systems

Implementation of the solution of the linear system of equations is the most important aspect of this problem on the CM-2. The linear-system solver is the source of a majority of the floating-point work and presents a challenge in the reduction of long-range communications. To utilize the machine effectively, it was found that the diagonalized form of the equations was necessary due to memory limitations (with the recent 4X memory upgrade this restriction has been removed). Hence, the discussion of linear systems that follows pertains primarily to the diagonalized scheme.

Setting Up the Left-Hand Side

Before solving the linear system of equations, the left-hand side (i.e., the system matrix) must be constructed. Construction of the left-hand side includes the calculation of the elements of the diagonalized block tridiagonal matrices, and the calculation of the eigenvector transformation matrices that are used to diagonalize the blocks.

To compute the elements of the tridiagonal matrices, the eigenvalue vectors are first computed at every grid node. Setting up the off diagonals involves short-range shifting of these vectors and a few simple calculations. Also, the implicit dissipation is added to the elements of the tridiagonal matrix. Each of these steps is fully parallelizable with some short-range communication. Since the values are required at all nodes except on the boundaries, this phase of execution uses the processors very efficiently.

Computation of the transformation matrices is very efficient since it involves a significant amount of floating-point work, no communications, and full utilization of the processors. Thus, overall, setting up the left-hand side is a high-performance feature of the code on a massively parallel machine. All of the calculations fully utilize the processors with a minimum of short-range communication, and no long-range communication.

LU Decomposition

The traditional method of solving a tridiagonal system of equations is to use LU decomposition to obtain the solution, sometimes called the Thomas Algorithm. This technique is also valid for block tridiagonal systems of equations. The method is very fast and requires only $\mathcal{O}(n)$ operations for both the forward and back substitutions, where n is the number of unknowns of the tridiagonal system. However, the solution technique is recursive, thus nonparallelizable in the inversion direction.

Parallel Cyclic Reduction

For a massively parallel computer, such as the CM-2, where each processor can be used to represent a grid point in the solution domain, a direct method that uses all processors simultaneously is desired. The method of odd-even cyclic reduction applied to a tridiagonal system of equations has this property. The details of this technique are described in Ref. 19.

Conceptually, the method consists of performing a series of mathematical operations that move the upper and lower diagonals in the matrix away from the main diagonal in the finite sequence of steps (or cycles). Since the mathematical operations used to move the off diagonals are not recursive, all processors can be used simultaneously. Because the number of cycles (N) needed is finite and a function only of the size (n) of the tridiagonal matrices

$$N = 1 + \log_2(n-1)$$

the method is a direct one. Furthermore, the displacement of the upper and lower diagonals away from the main diagonal is proportional to 2^n. This means that the actual implementation on the CM-2 can use the most efficient form of long-range communication possible (power-of-two NEWS).

The advantage of this algorithm on the CM-2 is that all processors are doing floating-point work simultaneously since each row of the tridiagonal matrices is represented by a processor. There is, however, increased work and communication activity associated with this method. This increase in work, as well as communication, is magnified for block tridiagonal systems since the cyclic-reduction method requires more block inversions and movement of data. In addition, there is the need for more memory, which limits the size of problem that can be solved. In the diagonal form, storage is reduced to the diagonal elements (5 vs 25) and the matrix inversions of the 5×5 blocks are reduced to five scalar divisions. Even with the increased floating-point work and communication, cyclic reduction for the diagonal form is much more efficient than LU decomposition on the CM-2 because all processors are used simultaneously.

There is an additional side effect of this algorithm that can be exploited under certain circumstances. If the system of equations is diagonally dominant, then the off-diagonal elements reduce in magnitude quadratically relative to the diagonal elements at each cycle of the algorithm. As a result, it may not be necessary to perform all of the cycles of the reduction to obtain an acceptable solution since the elements on the off diagonals become negligibly small. This can have a dramatic impact on performance since the last few cycles of the method require the largest shifts of data and hence long-range communications time.

When solving the Navier-Stokes/Euler equations with the ADI procedure, the linear systems are not strictly diagonally dominant. However, they are sufficiently close so that this feature can be used to an advantage. Jespersen and Levit[6] have investigated this feature in detail for steady and unsteady two-dimensional viscous flow problems. They found that by terminating the cyclic reduction at a specified ratio of the maximum off-diagonal element to the diagonal element, ϵ, the long-range communications and total CPU cost could be dramatically reduced without destroying the time accuracy. Recently, this feature, referred to as "cyclic reduction with early cutoff" or just "early cutoff," was implemented into NSR3D. Inclusion of this feature requires a negligible amount of additional CPU yet can eliminate almost all of the long-range communication requirements of the linear system solver as will be shown in the performance discussion. The net result is an implicit flow solver that demonstrates a comparable efficiency (i.e., relative performance to the CRAY) on the CM-2 as explicit methods.

Boundary Conditions

As mentioned previously, the boundary conditions are a source of computational inefficiencies on the CM-2 because many processors will be sitting idle. These inefficiencies can be minimized by grouping the computations together as much as possible. Most of the communications that occur are short range (extrapolation of quantities), but averaging quantities across the C-grid is a potentially severe bottleneck in the solver.

To apply the boundary conditions, masks are preset, which define the boundary of interest. For example, there is a mask for the surface of the blade:

```
C Mask for the blade surface (C-grid)

  MASK_SURF = .FALSE.

  MASK_SURF(ITEL:ITEU,1:JTIP,1) = .TRUE.
```

where ITEL, ITEU, and JTIP denote the lower trailing edge, upper trailing edge, and tip locations, respectively. Similarly, there is a mask that defines the cut in the grid. This cut occurs at $k=1$, beyond the tip ($j>$JTIP) for all i, and behind the trailing edge ($i<$ITEL or $i>$ITEU) for $j \leq$ JTIP. Note that one mask, call it MASK_CUT, should be used to define the grid-cut boundary, although using two may seem logical. The use of two masks will double the amount of time to apply this condition. Predefined masks are a convenient way to treat the boundary conditions and, if employed properly, they can help reduce the execution time.

Each far-field boundary must be treated separately because extrapolation occurs from different directions depending on the boundary. The best one can do is to group the calculations of similar form under a mask that defines all far-field boundaries. For example, after the various extrapolations are computed at the far-field boundaries, the total energy is computed from the equation of state. This can be performed for all far-field boundaries at once under a single mask (ALL_FF):

```
  WHERE( ALL_FF )

    Q5 = P/(GAMMA-1.0) + 0.5 * ( Q2*Q2 + Q3*Q3
  1
     + Q4*Q4 )/Q1

  END WHERE
```

The quantities on the right side were previously computed under individual masks for the different far-field boundaries. To compute these other quantities, the normal Mach number to the far-field boundaries must be computed to determine which quantities are fixed or extrapolated. The calculation of this Mach number can also be performed for all far-field boundaries at once. However, different components of metrics (ξ, η, or ζ) are required, depending on which computational plane represents the boundary. Thus, before the normal Mach number is computed, dummy metrics are set to the appropriate metric values using individual masks.

At the surface of the blade, which corresponds to the MASK_SURF mask, the velocity relative to the blade is set to zero, and density and pressure are extrapolated from the interior. This is a very simple condition that involves a few short-range shifts of data.

Averaging the quantities across the cut in the C-grid is a more difficult problem. Recall that this averaging is employed behind the trailing edge as well as beyond the tip of the blade. For purposes of illustration, applying this condition beyond the tip is given here in serial code:

```
C Average quantities across the cut in the grid.

  DO J = JTIP+1, JMAX

  DO I = 1, IMAX

    I1 = I + (IMAX - 2*I + 1)

    Q1(I,J,1) = 0.5*( Q1(I1,J,2) + Q1(I,J,2) )
    Q5(I,J,1) = 0.5*( Q5(I1,J,2) + Q5(I,J,2) )
  ENDDO
  ENDDO
```

The index I1 represents the grid node that is across the cut from the *I*th. It was coded here such that the shift in I is evident, (IMAX − 2*I + 1). The data must be shifted by a variable amount. One way to do this, *the wrong way,* is to have a DO loop (in I) that performs the CSHIFT operation. This could be done in parallel (i.e., for all J), but the I loop would be serial. Initially, this condition was coded using this method and it consumed 60% of the computer time! The fact that such a simple operation can dominate the time of a three-dimensional Navier-Stokes solver verifies the potential for communication bottlenecks, especially for the inexperienced CM-2 programmer.

To perform this in parallel, vector-valued subscripts are used, as demonstrated in the next example for averaging the density:

```
  DO I = 1, IMAX
    I1(I) = IMAX − I + 1
  ENDDO
  Q1_DUM(I1,1:JMAX,1:KMAX) = Q1
  WHERE( MASK_CUT )
    Q1 = 0.5*CSHIFT( (Q1 + Q1_DUM), 3, +1 )
  END WHERE
```

In this example, I1 is now a vector-valued subscript and would be precomputed. The CSHIFT is used to grab the data just off of the cut ($k = 2$). Performing this boundary condition in parallel for I reduces the computation time by a factor of 1/IMAX.

Computational Performance

In this section, various performance statistics of the CM-2 version of NSR3D are compared with the statistics of the highly vectorized version run on a single-processor of a CRAY-2. The CM-2 cases were run on a 16 k CM-2 (one-quarter of a full CM-2). The following timings and performance profiles are for a single time step of the solver and include the Baldwin-Lomax turbulence model. There is also some additional overhead on the front end associated with initialization and termination of the code; these times will also be addressed. It is important to note that the CRAY-2 has 64-bit precision while the CM-2 is a 32-bit precision machine. If 64-bit precision is required, the CM-2 performance with double-precision hardware would be halved. To date, it has been found that 32-bit precision is adequate for the NSR3D code.

To utilize the CM-2 to its maximum efficiency, the problem must be dimensionalized to match the number of virtual processors available (8 k times an integral power of two: 8 k, 16 k, 32 k, etc.). For example, a 200 k grid will run just as long as a 256 k grid. Furthermore, the higher the vp ratio the better, since the average time for a long-range communication is reduced and there is some gain in arithmetic operations on the floating-point hardware. The CRAY-2 times are for the most efficient implementation on the CRAY, which is the LU decomposition for the solution of the systems of tridiagonal equations.

Nondiagonal Form

Originally, using an earlier version of the compiler, the nondiagonal form of the solver was implemented on the CM-2. Because the memory requirements were so large, timings for the nondiagonal form were for a 32 k grid-node problem (vp ratio = 2). By using this unrealistically small grid, the CRAY-2 performance will be poor relative to an appropriately sized Navier-Stokes grid because of the short vector lengths.

It was found that the cyclic reduction without cutoff performed better than the LU decomposition but neither approached the speed of the CRAY-2 for this problem (cyclic reduction was 35% CRAY-2 speed). On the latest compiler, based on the improvement in the performance of the diagonal form and other codes, this would increase to roughly 65% CRAY-2 speed. Implementation of the cutoff feature would improve the performance even more (at least by a factor of two based on the timings which will be discussed later). However, the memory requirements are so great for the nondiagonal form that Navier-Stokes solutions for very large grid sizes cannot be achieved on the present 16 k CM-2. With the recent memory upgrade, a 256 k grid could be solved without diagonalization but this still does not offer the potential of the CRAY-2 (well over a 512 k can be run on the CRAY-2). For these reasons, the nondiagonal form has not been pursued further. As a final note, it should be mentioned that, given a full machine with the memory upgrade, the time-conservative nondiagonal form could be exercised for a 1024 k grid.

Diagonal Form

Using the diagonal form of the equations requires less memory, as a result the vp ratio can be as high as 64, allowing 1024 k grid on one-quarter of the present 16 k CM-2. However, to date, solutions for only a 256 k grid have been obtained since the larger memory is a recent upgrade. The performance for a 256 k grid is summarized in Table 1.

Diagonalization decreases the time on the CRAY-2 by about a factor of two, but the relative performance of the CM-2 still improves considerably. The fully implicit scheme is equal to the CRAY-2, and the hybrid scheme is 16% better than that for the CRAY-2. For most grids, the hybrid scheme is preferred whether execution is on a vector machine or the CM-2. The CM-2 relative performance is better for the hybrid scheme due to the reduced long-range communications.

From the above comparisons, it is evident that the LU decomposition cannot compete with the cyclic reduction on the CM-2. Hence, the cyclic reduction appears to be the method of preference for this type of problem on the present-day CM-2.

Recently, the early cutoff feature was implemented into the cyclic reduction scheme, its performance for several values of ϵ are given in Table 2.

Note that $\epsilon = 0$ corresponds to solving the system of equations exactly, using all cycles of the cyclic reduction. In the last column of this table, the relative difference in the residual after 20 iterations is given. The small difference between the

Table 1 Relative performance of cyclic reduction and LU decomposition

	Full ADI scheme		Hybrid scheme	
	Sec.	Relative performance	Sec.	Relative performance
CRAY-2 (LU)	7.12	1.00	6.30	1.00
CM-2 cyclic ($\epsilon = 0$)	7.10	1.00	5.45	1.16
CM-2 (LU)	15.19	0.46	13.10	0.48

Table 2 Relative performance of cyclic reduction with early cutoff

	Full ADI	Hybrid	
	Relative performance	Relative performance	Relative error
CRAY-2 (LU)	1.00	1.00	0
CM-2 ($\epsilon = 0$)	1.00	1.16	10^{-6}
CM-2 ($\epsilon = .001$)	2.27	2.42	5×10^{-6}
CM-2 ($\epsilon = .1$)	2.41	2.56	8×10^{-4}

CRAY-2 and the $\epsilon = 0$ solution can be attributed to roundoff error. Using a value of $\epsilon = 0.001$ produces essentially the same residual. In the work of Ref. 6, it was found that $\epsilon = 0.001$ produced, for all practical purposes, the same solution, while for $\epsilon = 0.1$, the errors are significant and are likely to taint unsteady solutions slightly, although steady-state solutions will be unaffected. Most of the gain in performance is achieved by using $\epsilon = 0.001$ and since the residual is virtually unchanged, this is the value that is used for the benchmark performance comparisons with the CRAY-2.

The hybrid scheme exhibits slightly better relative performance than the fully implicit scheme. This is due to the reduced communications and floating-point work required for the hybrid scheme (fewer linear systems to solve). Note that as ϵ is increased, the difference between the two schemes relative to the equivalent CRAY code becomes less since the influence of communications is reduced.

Front-End Time

The performance statistics presented earlier are for the time-stepping loop only. Typically, hundreds or several thousand iterations are exercised for a given run. Consequently, the total run time is dominated by the execution within the time-stepping loop. However, since the computations outside the time-stepping loop (which are primarily performed on the front end) do require a measurable amount of time, this aspect of performance must also be addressed.

Execution before the time-stepping loop (pre-iteration computations) consists of reading the input variables, generating or reading the grid, reading the initial solution, computing the metrics, setting the logical masks, and transferring arrays to the CM-2. The postiteration executions include transferring arrays to the front end, creating plotting and output files, and generating the solution file for restart capability. The restart files (or solutions files) are written to the high-speed data vault, saving a considerable amount of I/O time. For a 256 k Navier-Stokes case, ideally 1000 steps are run at a time; in Table 3 the out-of-loop times are presented for the CM-2 and CRAY-2, along with the iteration and total time for a run consisting of 1000 iterations using the hybrid scheme ($\epsilon = 0.001$).

The out-of-loop overhead is four times as fast on the CRAY-2. For a typical run (1000 steps), this does not impact the performance significantly. However, if smaller jobs are continually submitted (200–500 steps), the impact on overall performance becomes larger.

Performance Profile

To identify bottlenecks in the solver, it is useful to determine where the majority of the time is accumulated. A rough performance profile is given for the three major portions of an iteration: 1) set up and solve the linear systems, 2) RHS plus control overhead, 3) boundary conditions. The profiles are given in Table 4 for the hybrid scheme using cyclic reduction with no cutoff ($\epsilon = 0$) and with a cutoff of $\epsilon = 0.001$ for the above 256 k case. These timings differ slightly from those in Ref. 7, due to compiler upgrades and inclusion of the turbulence model in the timings. First consider the CM-2 profile with $\epsilon = 0.0$ (no cutoff). Nearly three-fourths (73.7%) of the time is used solving the linear systems. It was determined that more than two-thirds of this time is due to long-range communications. Therefore, at least 50% of the total run time is the long-range communications of the cyclic reduction procedure. The performance of the boundary conditions is poor compared to the CRAY-2, but it does not dominate the overall time. The degradation of performance associated with the boundary conditions and the long-range communications of the cyclic reduction is recovered by the superior performance of the CM-2 in computing the right-hand side. (It is important to emphasize that this right-hand side includes the Baldwin-Lomax turbulence model.) As future problems of interest become larger (more grid points), the impact of the boundary conditions will be reduced.

The fact that the long-range communications consume at least 50% of the total run time explains the effectiveness of the cyclic cutoff method. In using the early cutoff procedure ($\epsilon = 0.001$), the time for the right-hand side and boundary conditions are not affected since only the linear solver has been changed. The long-range communications have almost been entirely eliminated, reducing the time for the linear-systems portion to 30% of what it was without the early cutoff. For this procedure, the linear systems prove to be just as efficient as the explicit right-hand side (relative speed of 2.8). This demonstrates that the long-range communications no longer drive the efficiency of this linear system solver. The boundary conditions, which now consume 13% of the computer time, reduce the overall performance from 2.8–2.4. However, the net efficiency (relative to the CRAY) of this implicit scheme is approaching that of the explicit portions, and hence explicit schemes in general.

Results

A variety of applications of NSR3D are described in Ref. 8 to demonstrate the present capabilities of the solver. These applications include propeller, helicopter, Propfan, and fixed-wing configurations. Here, Euler and Navier-Stokes solutions are presented for a conventional propeller geometry. During the validation phase of this research effort, comparisons with the vectorized version of the code were made. Based on a detailed comparison of all components of the residual history and final pressure distributions, it was determined that the results from the CM-2 version of the analysis are identical to that of the vectorized version.

Euler and Navier-Stokes solutions for the NACA 16-series propeller geometry shown in Fig. 1 have been obtained. A large amount of pressure data is available for this two-bladed geometry and is given in Ref. 20. For the case presented here, the axial freestream Mach number was 0.60 and the rotational-tip Mach number was 0.92 (a net freestream Mach number of 1.10 at the tip). Pressure distributions are given in Figs. 2 and 3 for two radial locations. C-grids were used for these calculations: 128 k nodes (128 streamwise, 32 spanwise, 32 normalwise) for the Euler calculations, and 256 k nodes

Table 3 Overall performance (1000 steps of hybrid scheme, $\epsilon = 0.001$)

	CM-2, s	CRAY-2, s	CM-2 relative performance
Iteration loop	2600	6300	2.42
Outside of loop	120	30	0.25
Total for hybrid run	2720	6330	2.33

Table 4 Performance profile of hybrid scheme

	CM-2 cyclic ($\epsilon = 0$)			CM-2 cyclic ($\epsilon = 0.001$)			CRAY-2 (LU)	
Category	Sec.	Percent time	Relative speed	Sec.	Percent time	Relative speed	Sec.	Percent time
1	4.01	73.7	0.81	1.17	45.0	2.76	3.23	51.3
2	1.09	24.0	2.80	1.09	41.9	2.80	3.05	48.4
3	0.34	6.3	0.06	0.34	13.1	0.06	0.02	0.3
Total	5.44		1.16	2.60		2.42	6.30	

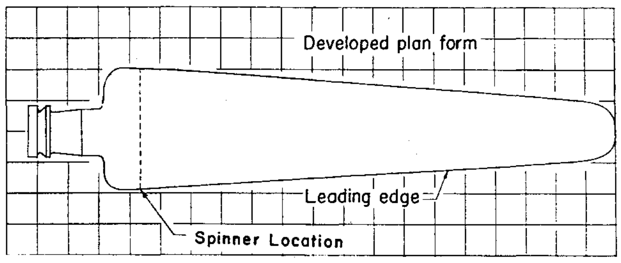

Fig. 1 Planform geometry of NACA propeller blade of Ref. 20.

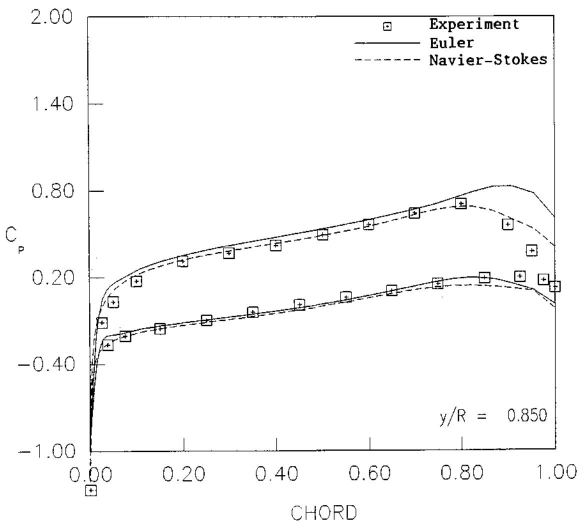

Fig. 2 Pressure distributions compared with experiment for NACA propeller blade.

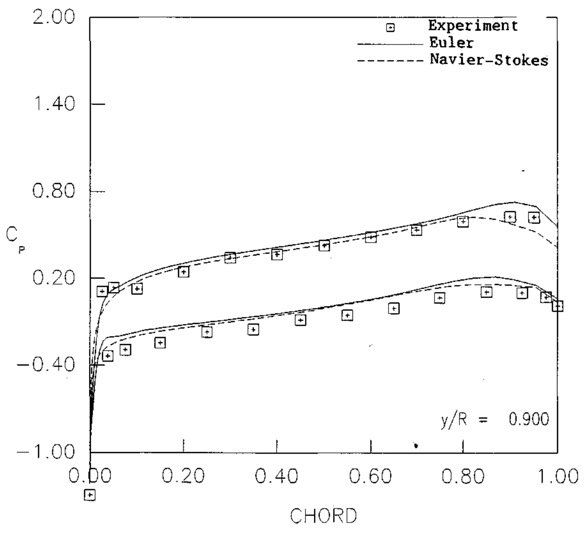

Fig. 3 Pressure distributions compared with experiment for NACA propeller blade.

(128 streamwise, 32 spanwise, 64 normalwise) for the Navier-Stokes case. For the Navier-Stokes analysis, the normal spacing at the blade surface was 0.0001 chords and the Reynolds number was two million. At these radial stations the overall agreement with experiment is good, and the viscous solutions are an improvement over the Euler results. Further inboard there is a strong shock near the trailing edge, which is not well resolved using the present C-grid. It has been found that using an O-grid (for Euler calculations) improves the shock capturing greatly due to the fine streamwise spacing near the trailing edge. Navier-Stokes calculations using the O-grid and a larger C-grid are planned.

Concluding Remarks

An implicit Navier-Stokes solver has been developed for use on the massively parallel connection machine. This code, NSR3D, now achieves better than twice the performance of a single-processor CRAY-2 on one-quarter of a machine. The solver was coded using FORTRAN with the 8X array extensions. The authors have found this language extension to be very well-suited for the data-level parallelism of the CM-2 architecture and application to numerically intensive programs such as a Navier-Stokes solver. Although achieving good computational performance may require modifications to the fundamental algorithms, utilizing the machine with FORTRAN 8X array extensions is not really any more difficult from a programming point of view than developing a well-vectorized code.

An application of the code has been demonstrated for a propeller. Presently, the code has reached a level of maturity at which flow features can be examined and numerical fine-tuning can be performed to improve correlation with experiment. For helicopter applications, the inclusion of the wake effects still requires validation. Also, to resolve the details of vortical flow features, larger grid densities are desired. The memory has recently been upgraded to four times the previous CM-2 memory, enabling grids with 1024 k nodes to be solved on a quarter machine. To enhance to stability of the solver, efficient schemes for pentadiagonal systems and tridiagonal systems with fringes are under development.

In conclusion, the authors feel that the practical and efficient use of massively parallel computers, like the CM-2, has been demonstrated by the implementation of NSR3D using FORTRAN 8X array extensions for implicit CFD algorithms. The knowledge gained by developing the NSR3D code (and other codes under development in other disciplines) on the CM-2 is providing the foundation necessary to rapidly utilize the next generation of massively parallel computers.

Acknowledgments

The CRAY-2 runs were performed on the National Aerodynamic Simulator (NAS) facility at the NASA Ames Research Center.

References

[1]Egolf, T. A., "Scientific Applications of the Connection Machine at the United Technologies Research Center," The NAS Connection Machine Conference, NASA Ames Research Center, Moffett Field, CA, Sept. 12-14, 1988.

[2]Long, L. N., Khan, M. M. S., and Sharp, H. T., "A Massively Parallel Three-Dimensional Euler/Navier-Stokes Method," AIAA Paper 89-1937, June 1989.

[3]Egolf, T. A., "Helicopter Free-Wake Analysis on a Massively Parallel Computer," The International Symposium on Boundary-Element Methods, East Hartford, CT, Oct. 2-4, 1989.

[4]Agarwal, R. K., "Development of a Navier-Stokes Code on a Connection Machine," AIAA Paper 89-1938, June 1989.

[5]Shapiro, R. A., "Finite-Element Algorithms for the Euler Equations on the Connection Machine," The International Seminar on Supercomputing in Fluid Flow, Lowell, MA, Oct. 3-5, 1989.

[6]Jespersen, D., and Levit, C., "A Computational Fluid Dynamics Algorithm on Massively Parallel Computer," AIAA Paper 89-1936, June 1989.

[7]Wake, B. E., and Egolf, T. A., "Implementation of a Rotary-Wing Three-Dimensional Navier-Stokes Solver on a Massively Parallel Computer," AIAA Paper 89-1939, June 1989.

[8]Wake, B. E., and Egolf, T. A., "Applications of a Rotary-Wing Viscous Flow Solver on a Massively Parallel Computer," AIAA Paper 89-0334, Jan. 1990.

[9]Hillis, W. D., "The Connection Machine," *Scientific American,* Vol. 256, No. 6, June 1987, pp. 108-115.

[10]Connection Machine Model CM-2 Technical Summary, Thinking Machines Technical Rept. HA87 – 4, Thinking Machine Corp., April 1987.

[11]Metcalf, M., and Reid, J., *FORTRAN 8X Explained,* Clarendon Press, Oxford, 1987.

[12]Wake, B. E., and Sankar, L. N., "Solution of the Navier-Stokes Equations for the Flow about a Rotor Blade," *American Helicopter*

Society Journal, Vol. 34, No. 2, April 1989, pp. 13–23.

[13]Beam, R., and Warming, R. F., "An Implicit Factored Scheme for the Compressible Navier-Stokes Equations," *AIAA Journal,* Vol. 16, April 1978, pp. 393–402.

[14]Pulliam, T. H., and Steger, J. L., "Implicit Finite-Difference Simulations of Three-Dimensional Compressible Flow," *AIAA Journal,* Vol. 18, No. 2, 1980, pp. 159–167.

[15]Rizk, Y. M., and Chausee, D. S., "Three-Dimensional Viscous-Flow Computations Using a Directionally Hybrid Implicit-Explicit Procedure," AIAA Paper 83-1910, 1983.

[16]Pulliam, T. H., and Chausee, D. S., "A Diagonal Form of an Implicit Approximate-Factorization Algorithm," *Journal of Computational Physics,* Vol. 39, 1981, pp. 347–363.

[17]Baldwin, B. S., and Lomax, H., "Thin-Layer Approximation and Algebraic Model for Separated Turbulent Flows," AIAA Paper 78-257, June 1978.

[18]Jameson, A. J., Schmidt, W., and Turkel, E., "Numerical Solutions of the Euler Equations by Finite-Volume Method Using Runge-Kutta Time-Stepping Schemes," AIAA Paper 81-1259, 1981.

[19]Stone, H. S., "Parallel Tridiagonal Equation Solvers," *ACM Transactions on Mathematical Software,* Vol. 1, No. 4, Dec. 1975, pp. 289–307.

[20]Evans, A. J., "Propeller Section Aerodynamic Characteristics as Determined by Measuring the Section Surface Pressures on a NACA 10-(3)(08)-03 Propeller under Operating Conditions," NACA RM L50H03, Nov. 1950.