NASA Technical Memorandum 102806

Multiprocessing on Supercomputers for Computational Aerodynamics

Maurice Yarrow and Unmeel B. Mehta

May 1990

NASA
National Aeronautics and
Space Administration

NASA Technical Memorandum 102806

Multiprocessing on Supercomputers for Computational Aerodynamics

Maurice Yarrow, Sterling Federal Systems, Inc., Palo Alto, California
Unmeel B. Mehta, Ames Research Center, Moffett Field, California

May 1990

National Aeronautics and
Space Administration

Ames Research Center
Moffett Field, California 94035-1000

MULTIPROCESSING ON SUPERCOMPUTERS FOR COMPUTATIONAL AERODYNAMICS

Maurice Yarrow
Sterling Federal Systems, Inc., Sterling Software
and
Unmeel B. Mehta
NASA Ames Research Center

ABSTRACT

Very little use is made of multiple processors available on current supercomputers (computers with a theoretical peak performance capability equal to 100 MFLOPS or more) in computational aerodynamics to significantly improve turnaround time. The productivity of a computer user is directly related to this turnaround time. In a timesharing environment, the improvement in this speed is achieved when multiple processors are used efficiently to execute an algorithm. We apply the concept of multiple instructions and multiple data (MIMD) through multitasking via a strategy which requires relatively minor modifications to an existing code for a single processor. Essentially, this approach maps the available memory to multiple processors, exploiting the C-Fortran-Unix interface. The existing single processor code is mapped without the need for developing a new algorithm. The procedure for building a code utilizing this approach is automated with the Unix stream editor. As a demonstration of this approach, a Multiple Processor Multiple Grid (MPMG) code is developed. It is capable of using nine processors, and can be easily extended to a larger number of processors. This code solves the three-dimensional, Reynolds averaged, thin-layer and slender-layer Navier-Stokes equations with an implicit, approximately factored and diagonalized method. The solver is applied to a generic oblique-wing aircraft problem on a four processor Cray-2 computer, using one process for data management and non-parallel computations and three processes for pseudo-time advance on three different grid systems. These grid systems are overlapped. A tricubic interpolation scheme is developed to increase the accuracy of the grid coupling. For the oblique-wing aircraft problem, a speedup of two in elapsed (turnaround) time is observed in a saturated timesharing environment.

INTRODUCTION

Current supercomputer architectures employ a few high-performance processors to provide a significant increase in speed (defined as throughput) over the speed of single-processor

supercomputers of the past. With massively parallel supercomputer architectures, the speed of supercomputers may be increased by at most two orders of magnitude. Specifically, a factor of 100 from a highly parallel computer architecture can be expected. This limitation is explained by Amdahl's law[1], which imposes a stiff penalty for the last few percent of non-parallelizable code in an otherwise parallelizable program. (Note that Buzbee and Sharp[2] have credited Ware[3] for this model.) This model fails to account for additional instructions required when multiprocessing an otherwise single processor group of instructions. Taking into account this increase in instructions for multiprocessing, the Buzbee and Sharp model suggests that the maximum speedup is less than the number of processors, even if the computation under consideration can be completely put into a parallel form. In the future, supercomputers with between 16 and 64 processors are likely to be available. Yet there has been very little use made of multiple processors available on current supercomputers in computational aerodynamics to significantly enhance the productivity of the user. The objective of the presented research is to map an existing algorithm onto a multiprocessor supercomputer to demonstrate the advantages of using multiple processors.

The motivation of this research is as follows. In computational aerodynamics, the need to study flow fields around more realistic and complex three dimensional geometries has led to a few promising computational techniques, all of which make a substantial demand on supercomputing resources, namely cpu time and memory. The computing power required is roughly proportional to the number of grid points (or finite volumes or finite elements) into which the complete flow field is discretized. In a timesharing environment, jobs will be resident for long times, as measured by the wall clock. Total wall-clock (elapsed) time is proportional to the total number of grid points; this time can be substantially reduced by using a number of processors simultaneously.

We will now offer some perspective on various techniques which have been used to parallelize computer codes. First, applications on computers other than supercomputers are considered, and then those on supercomputers are discussed.

Liewer et al.[4] have implemented a plasma particle-in-cell simulation code on a Hypercube 64 processor machine by using each processor to calculate, for a single particle, the updated particle positions, velocities, and particle contributions to charge and current density. Comparisons with Cray X-MP/48 *single* processor times for the most computation intensive section of the code indicate a speedup in the elapsed time by a factor of two. Much effort is also being applied at this time to parallelize flow solving methods onto massively parallel architectures, such as the Connection Machine. Lin[5] has implemented a particle-in-cell simulation of wave particle interactions using the massively parallel processor (MPP) which consists of 16,384 processors. Tuccillo[6] has used the Connection Machine (CM-2) for numerical weather prediction. Jespersen and Levit[7] have used the Connection Machine for a two-dimensional finite-difference algorithm solving the thin-layer Navier-Stokes equations. On such computers, however, uniquely system-specific instructions and algorithms are required to achieve speeds comparable to that possible when multiple processors of supercomputers are utilized. Creating these codes thus requires a considerable investment in programming time. Codes typically implemented on vector supercomputers need to be programmatically redesigned.

On supercomputers such as the Cray Y-MP and the Cray-2, multitasking, microtasking, and autotasking are available. Multitasking is a mechanism for multiprocessing at a subroutine level. The implementation from Fortran is via such calls as TSKSTART and TSKWAIT. With these calls, parallel execution of Fortran subroutines may be initiated and synchronized. This implementation of multiprocessing is well suited for MIMD architectures. Microtasking is a mechanism for multiprocessing of DO loops. It parallelizes the execution of DO loops most typically immediately outside the innermost vectorizing loops. It is invoked with compile directives such as DO GLOBAL in the Fortran source. The precompiler "premult" will then add appropriate parallelizing system calls to the Fortran source. Autotasking is essentially microtasking that is automatically performed by the Cray dependency analyzer FPP. It is activated by a compile switch and is thus the simplest multiprocessing option to use. Autotasking is well suited for SIMD-appropriate algorithms (single instruction, multiple data).

Smith and Miller[8] have calculated galactic collisions by multitasking the motion of four groups of stars (each group of which is arranged in 256 blocks, with each block containing 1024 stars) onto four Cray-2 processors. The motion of stars in time is influenced by a potential field, which must be updated between timesteps based on the latest position of particles. Taylor and Bauschlicher[9,10] have multitasked the work required for generation of full configuration interaction wave functions in computational chemistry problems by subdividing over inner products large matrix multiplies onto four Cray-2 processors. Andrich et al[11] have multitasked a general circulation model of the ocean by applying vertical and horizontal operators by "slab" (plane) on separate processors of a Cray-2. Chevrin[12] has simultaneously multitasked and microtasked the NCAR Community Climate Model on a Cray X-MP/48 and has achieved speedups (decrease of elapsed time) of up to 3.7. In this case, vertical slabs within the model constituted independent computational elements and were multitasked at the subroutine level onto separate processors. It seems there are no archived publications using multiprocessing for computational aerodynamics.

There are two approaches to speedup on supercomputers: global (coarse-grain) parallelization and local (medium and fine-grain) parallelization. The latter is the approach taken in the references just cited. In the present work, we have achieved a speedup using a global coarse-grain implementation and making use of the available MIMD architectures, with basically standard algorithms. The reason for not using microtasking and autotasking is explained below.

Microtasking and autotasking are not efficient for any but the most ideal algorithms on the Cray Y-MP and Cray-2 architectures, because the synchronization between processes under microtasking results in the loss of "synchronization wait" time. For real world algorithms, this results in additional cpu time consumed. This cpu penalty is highly system-load dependent and seems to be proportional to this load. Microtasked processes waiting and idle at a synchronization point continue to accumulate user cpu time.

The productivity of the user depends not only on the efficiency of the code he or she develops but also on the efficiency of the computer resources he or she is going to use. The concept of MIMD through multiprocessing allows efficient use of these resources. This is

explained as follows. First, consider a worst case. If a single process job accesses all of the available central memory, all processors except one will be idle since there is no memory available for jobs queued for the other processors. This causes considerable inefficiency in system resources and in throughput. A possible solution is to let a user request m/p of the total memory resources, where m is the number of processors requested by the user and p is the total number of processors available. Second, multiprocessing allows completion of jobs sooner, freeing the system for use by other users and providing a considerable benefit in terms of enhanced productivity to the user community at large. Third, the shortening of residence time for a job reduces vulnerability to system crashes.

In the following sections, the multitasking implementation is first discussed, and a general outline of the method is given. A simple example of multitasking is then provided to give some basis for the subsequent detailed look at the multitasking implementation in the MPMG code. A discussion is included of some relevant memory management coding details. The Unix stream editor, which is used to automate code editing tasks, is then briefly discussed. Next follows a description of the governing equations, followed by a development of the tricubic interpolation scheme used for grid coupling, and discussions of numerical experiments.

MULTITASKING APPROACH

Multitasking on a supercomputer with multiple processors requires that a computational task be subdivided into independent tasks, which are then run concurrently. This technique results in a decrease in wall clock time for tasks so subdivided, but obviously not in a decrease in the total number of floating point operations required. Nevertheless, the gain in terms of productivity will still be considerable for the user.

There are many possible strategies for multitasking fluid flow solvers, but here we will describe only the MPMG solver implementation, and the requirements that led to its particular multitasking strategy. This strategy is different from the strategies discussed in the Introduction. Although the multitasking approach is discussed below in the context of the Cray-2 computer, it is applicable in principle to other such supercomputers.

The Cray-2 permits multitasking via two different software mechanisms. Since the Cray-2 operating system, Unicos, is a Unix implementation, processes may be multitasked out of the C programming language, using the system calls "fork", "exec", "wait", etc. These multitasking mechanisms, however, are relatively low level and lack the required spectrum of capabilities such as simple mechanisms for synchronization of tasks. The Cray-2-Unicos-Fortran implementation, on the other hand, provides a Multitasking Library, callable from Fortran, which offers a wide spectrum of multitasking capabilities. It guarantees that, when available, separate processors will run separate tasks (processes). One still competes with other users for processors, but tasks will benefit from true concurrency. Therefore, multiprocessing is done via the Fortran multitasking library calls.

Various requirements for MPMG dictate aspects of the design of this code. The MPMG code must be able to access very large amounts of the Cray-2 memory. As work space of

approximately 38 words per grid node is required, and since this code may be used for up to 9 grid systems, with an average of, possibly, one-third-million nodes each, it should be able to successfully request 125 million words of memory, or half of the entire Cray-2 memory. Therefore, a consistent and reliable mechanism for memory allocation is to force the loader to give the main program all of the work space memory and then to allocate subsets of this memory to the separate multitasked solvers.

Flexibility in memory allocation mechanisms and the ability to develop data types which would mimic and map to such Fortran memory types as `COMMON` blocks or subroutine parameter lists, made the use of the C programming language attractive for the main driver. Data types (structures) in C also are well suited to parallel programming techniques, as will be demonstrated. Additionally, C has the ability to call functions (or Fortran subroutines) by their addresses. This makes it possible to invoke the routines being multitasked by cycling through an array of function addresses. The advantage to such an approach is that multitasked subroutines are invoked by grid number, i.e., indicially. Such an approach is not possible in Fortran.

The decision to multitask the solver at the grid level (one flow solver copy for each grid), rather than within the grid level (finer grain level) is influenced by several factors. First, the choice of multitasking the solver at the grid level seemed a "natural" organizational level. Data and work spaces related to a given grid could easily be kept distinct from those of other grids. Second, multiple embedded grid schemes ordinarily advance a single independent time step on each constituent grid, and then update all dependencies between grids (coupling the grids explicitly, by interpolation). Third, the purpose of parallel processing is to speed up the execution of individual programs. The same program with different inputs and different grids can be run on a number of processes equal to the number of grids plus one. Fourth, multitasking should be transparent to the user so that any modifications to the physics and to the numerics can be easily made. This at once makes multitasking at the grid level simpler. However, when grid sizes differ in the number of nodes, multitasking results in the completion of a time step on a smaller grid before that of the larger. The advantage gained by multitasking is thus correspondingly diminished.

On the Cray-2, binding of C and Fortran object codes is simple and robust: the codes are simply individually compiled and then linked together. The first cardinal rule that must be observed when passing data between C and Fortran routines is that the C argument list contain only pointers (addresses) to the respective Fortran variables. This is normally true when the data being passed is in an array, in which case, placing the C array name in the argument list suffices to make the array's address visible to the called Fortran routine. (This is standard in Fortran-to-Fortran calls.) When a scalar variable is being passed from C to Fortran, a pointer variable containing the address of the scalar must be placed in the C argument list (called by address). On the Fortran side, the actual scalar variable is placed in the parameter list and subsequently receives the scalar value at its address.

The second cardinal rule which must be observed when binding C and Fortran codes on the Cray-2 is that all names of Fortran subroutines called from C and all names of C functions called from Fortran must be capitalized, since the Cray-2 Fortran compiler only

really recognizes capital letters in symbol names.

MULTIPROCESSING MULTIPLE GRID CODE

The code is divided into three main computational units. The first is a C main driver, the second is the multitaskable Fortran solvers, one for each grid, and the third is a small group of Fortran utility routines (Fig. 1). The Fortran utility routines include the interpolation and update procedures, some memory allocation procedures, the interpolation file read-in routine, etc.

The C main is preceded by a section of data structure declarations. These roughly fall into two types: those that map to Fortran `COMMON` blocks, and those that map to Fortran subroutine argument lists. In the main program, data initialization is followed by the actual time step loop, including grid interpolation and dependency update procedures, and the multitasking of the flow solvers for a time step. Following the time step loop is the termination sequence, including output of the restart file and user requested flow data.

The multitaskable flow solvers are each a collection of Fortran routines based on the core flow solver. When multitasking an existing Fortran code, there are two possible choices: either a single copy of the code can be multitasked m times, or m unique copies of the code can be produced and individually multitasked. Both of these choices have advantages and disadvantages, and these will be compared. The authors chose the latter method, which is made simpler, in part, by using the Unix stream editor to produce m unique flow solver copies.

MULTITASKING BASICS AND A SIMPLE EXAMPLE

The Cray-2 multitasking library is exceptionally easy to use, especially when called entirely from Fortran routines. The basic principles for utilizing this library are explained with an example. Appendix A is a small sample code "`MHEAT`" which solves the steady state 2-D heat equation by successive-over-relaxation on a grid of 100 by 200. This grid is partitioned into two grids of 100 by 100 each. The outer boundary is set to 200 degrees, the interior to 0 degrees at the outset. The intermediate boundary separating the right and left domains starts at 0 degrees. The routine "`SEIDEL`" is multitasked, one copy for the left domain, and one for the right. Given the latest boundary conditions for these domains, the parallel execution of the two copies of `SEIDEL` solves both domains simultaneously for a steady state solution (that is, until the iteration-to-iteration change drops below 0.0001 degrees) and then return to the main program. The centerline boundary between the two domains is then updated by averaging temperatures immediately on its left and right. This process is repeated and eventually, the entire domain converges to 200 degrees.

Note first the integer arrays `PROC1` and `PROC2`. These are required by the system multitasking routines and are used by them to identify the individual processes. The initialization of their first element is a necessary formality as is the external declaration of the routine `SEIDEL`. Next, note the call to the multitasking library routine `TSKTUNE`. This indicates to

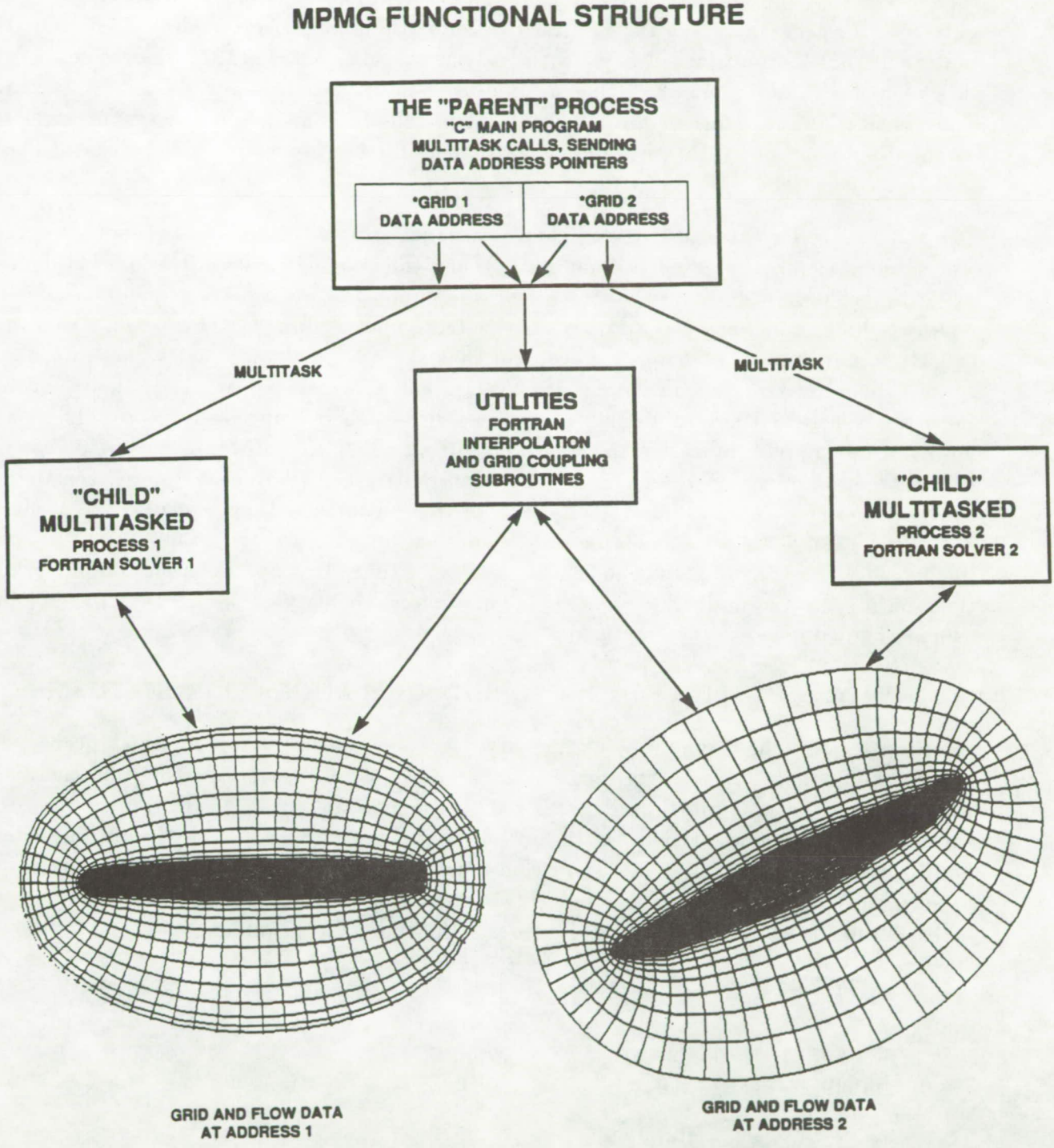

Fig. 1 Multiprocessing multiple grids.

the system that the maximum number of CPU's on which tasks may simultaneously compute is 4. In general, this is the natural choice for the four processor Cray-2. Finally, the routine `SEIDEL` is multitasked twice with the library calls to `TSKSTART`, which contains the task identifier, and the name of the routine being multitasked, followed by its arguments. The multitasking invocations are followed by two calls to `TSKWAIT`, which synchronizes the completion of the tasks, that is, guarantees that no further processing will occur until both tasks have completed.

The work space for the grid is the single dimensional array `A(20000)` in the main program. The starting element for the left domain `A(1)` and the starting element for the right domain `A(9901)` are passed down to the two tasks, whereupon the work space is redimensioned to be a two-dimensional array of appropriate size (e.g., idim x jdim/2). This permits each task to have a unique portion of work space, and thus there is no chance that values in the grid space will be overwritten inadvertently. Scalars and dimensioned arrays that are local to the multitasked routine are unique; there is no risk of their being overwritten. There are, however, no common blocks in the multitasked routine. The multitasking library does not provide a mechanism by which common data in multitasked routines is unique: scalars and arrays in commons are available to all copies of the routine which are executing. A declared data type `TASK COMMON` is available in the multitasking library. This `COMMON` is protected so that only one copy of a routine can access it at a time. But upon completion of work by that routine, another multitasked copy may now access the very same memory, thus possibly overwriting values.

MPMG MULTITASKING AND MEMORY ACCESS MECHANISMS

In order to overcome common block memory access uniqueness problems which occur when multitasking a subroutine containing common blocks, a technique different from the above (single copy of routine multitasked) was used in the MPMG code. As previously indicated, the base flow solver was replicated to form a unique copy for each grid. Subroutine names had to be made unique, of course, and common block names were also made unique within a given copy of the solver. These modifications were performed by the Unix "sed" stream editor script that builds the separate copies of the code. For example,

```
COMMON/BASE/A,B,C
```

becomes

```
COMMON/BASE1/A,B,C
```

everywhere in solver no. 1 and

```
COMMON/BASE2/A,B,C
```

for solver no. 2, and so on. The variables `A,B,C`, etc. are now unique to the individual copy of the numbered routine-common block combination in which they appear.

At the initialization of the MPMG program, all common blocks are made available to the C main code. This is done by calling a special set of Fortran utility routines `FXTERN1`,

FXTERN2, etc., one for each flow solver copy. In each of these were the (uniquely named) common blocks appropriate for the solver. (See Appendix B, for a sample.) The address of the start of each common block is then passed back to the C main code by calling the C function CXTERN. CXTERN places these addresses in the equivalent (indicially referenced by mesh number) pointer-to-structure element. (See Appendix C.) This has the effect of placing a "template", which is the structure, over (the memory starting at the address of the first variable in) each common block. The Fortran common blocks are now available to the C main via the arrays of pointers-to-structures.

As previously indicated, large amounts of Cray-2 memory are requested for flow solver work spaces. This memory is used for storing the grids, the flow data, intermediate calculations, etc. During the initialization process, the C main program calls a Fortran utility routine named FALLOC (Appendix D). As in the above mechanisms, FALLOC itself calls a C function CMNADDRS, and passes ALLMEM, which is the address of the block of memory requested in the common block WORKMEM. In CMNADDRS, the address of this (generally very large) block of memory is passed to a globally declared pointer-to-float, thus making the memory available to the C main. Subsequently, memory is allocated to work spaces by a simple C utility function (Appendix E), which sets the work space pointers to appropriate addresses within this contiguous memory block.

A convenient method for passing down to Fortran routines argument lists containing the addresses of the work spaces for a given grid is to build arrays of structures whose fields are pointers. For example, given

```
      SUBROUTINE GRID (JMAX,KMAX,LMAX,X,Y,Z)
      DIMENSION X(JMAX,KMAX,LMAX),
     &          Y(JMAX,KMAX,LMAX),
     &          Z(JMAX,KMAX,LMAX)
       .
       .
       .
      RETURN
      END
```

then an appropriate array of structures for mapping to this parameter list would be

```
struct s_grid
    {
    int *p_jmax;
    int *p_kmax;
    int *p_lmax;
    float *p_x;
    float *p_y;
    float *p_z;
    }    s_grid[NUMBER_OF_GRIDS];
```

There is one such structure for each grid. The pointers in this structure may be initialized

following the memory allocation procedure. This is done by the routine `init_structs`:

```
void init_structs (mesh_number)
    int mesh_number;
    {
    s_grid[mesh_number].p_jmax = p_jmax[mesh_number];
    s_grid[mesh_number].p_kmax = p_kmax[mesh_number];
    s_grid[mesh_number].p_lmax = p_lmax[mesh_number];
    s_grid[mesh_number].p_x = p_x[mesh_number];
    s_grid[mesh_number].p_y = p_y[mesh_number];
    s_grid[mesh_number].p_z = p_z[mesh_number];
    }
```

Thereafter, Fortran subroutines may be called from C in a very compact fashion. Referring to the above Fortran subroutine `GRID`, the C statement

```
GRID (s_grid[mesh_number]);
```

has the effect of placing all fields contained in the "mesh_number" element of the structure `s_grid` onto the subroutine argument call list, and since these individual fields are really addresses, the called Fortran subroutine receives only the appropriate addresses, as required.

Lastly, it is important to ensure Fortran functions may be called by address from C. If the C array `grid[NUMBER_OF_GRIDS]` is of type array of pointers-to-functions returning void, and if its individual elements have been loaded with the addresses of the Fortran subroutines `GRID1`, `GRID2`, and so on, then the statement

```
for (mesh_number=0;
     mesh_number<NUMBER_OF_GRIDS;
     mesh_number++)
    (*grid[mesh_number])(s_grid[mesh_number]);
```

will invoke these routines successively, passing to them the addresses for appropriate parameters which are being kept in the structure `s_grid`. Note that loading the addresses of Fortran subroutines into a pointer-to-function can be accomplished by first declaring the Fortran subroutines to be of type void, as follows:

```
void GRID1 ();
void GRID2 ();
.
.
.
```

The pointer-to-function returning void is declared by

```
void (*grid[NUMBER_OF_GRIDS])();
```

Finally, these pointers are given the Fortran subroutine addresses with

```
grid[0] = GRID1;
grid[1] = GRID2;
.
.
.
```

Multitasking these routines from C, followed by task completion synchronization, is now simple:

```
for_all_mesh TSKSTART (proc[mesh_number],
                       ,grid[mesh_number]
                       ,s_grid[mesh_number]);
for_all_mesh TSKWAIT (proc[mesh]);
```

where, for convenience, `for_all_mesh` has been previously defined by:

```
#define  for_all_mesh \
         for (mesh_number=0; \
              mesh_number<NUMBER_OF_GRIDS; \
              mesh_number++).
```

AUTOMATING LARGE SCALE EDITING WITH SED

The MPMG code building process merits some attention. Changes to the code required by the number of grids, the grid dimensions, etc., and also algorithmic changes to the flow solver core are not actually made to the base code, but rather, to a Unix shell script and Unix "sed" stream editor[13] script. This makes version control and testing of algorithmic changes and enhancements easier to monitor. Modifications, which may be quite global in nature, can be enabled or disabled simply by enabling or disabling portions of the controlling sed script.

The Unix stream editor sed is a very convenient tool for automating the MPMG text manipulating needs. Sed takes advantage of the "regular expression" capabilities of Unix. These versatile wild cards are without equal for generalizing text editing commands. Sed also accepts string variables passed into it, which may then be incorporated into the target text produced. Sed's ability to memorize portions of a line of text which match specified patterns or patterns specified with regular expressions makes it tremendously powerful.

Sed is employed to automate the task of producing unique copies of the base flow solver. Sed also makes all algorithmic changes to the flow solver, for example, those required by the interpolation mechanisms, and those necessary to account for differing grid sizes, etc. Thus, it is really never necessary to change the base flow solver, but instead, changes are incorporated into the sed script and the Unix shell script that drives sed.

GOVERNING EQUATIONS

The governing equations are the thin-layer approximation to the Navier-Stokes equations,

which are expressed as

$$\partial_\tau \widehat{Q} + \partial_\xi \widehat{E} + \partial_\eta \widehat{F} + \partial_\zeta \widehat{G} = Re^{-1}\partial_\zeta \widehat{S}$$

where

$$\widehat{Q} = J^{-1}\begin{bmatrix} \rho \\ \rho u \\ \rho v \\ \rho w \\ e \end{bmatrix}, \qquad \widehat{E} = J^{-1}\begin{bmatrix} \rho U \\ \rho u U + \xi_x p \\ \rho v U + \xi_y p \\ \rho w U + \xi_z p \\ (e+p)U - \xi_t p \end{bmatrix},$$

$$\widehat{F} = J^{-1}\begin{bmatrix} \rho V \\ \rho u V + \eta_x p \\ \rho v V + \eta_y p \\ \rho w V + \eta_z p \\ (e+p)V - \eta_t p \end{bmatrix}, \quad \widehat{G} = J^{-1}\begin{bmatrix} \rho W \\ \rho u W + \zeta_x p \\ \rho v W + \zeta_y p \\ \rho w W + \zeta_z p \\ (e+p)W - \zeta_t p \end{bmatrix},$$

$$\widehat{S} = J^{-1}\begin{bmatrix} 0 \\ \mu_{eff}\phi u_\zeta + (\mu_{eff}/3)\varphi\zeta_x \\ \mu_{eff}\phi v_\zeta + (\mu_{eff}/3)\varphi\zeta_y \\ \mu_{eff}\phi w_\zeta + (\mu_{eff}/3)\varphi\zeta_z \\ \{\phi[0.5\mu_{eff}(u^2+v^2+w^2)_\zeta \\ +\kappa Pr^{-1}(\gamma-1)(a^2)_\zeta] \\ +(\mu_{eff}/3)(\zeta_x u + \zeta_y v + \zeta_z w)\times\varphi\} \end{bmatrix}$$

with

$$\begin{aligned} \phi &= (\zeta_x^2 + \zeta_y^2 + \zeta_z^2) \\ \varphi &= (\zeta_x u_\zeta + \zeta_y v_\zeta + \zeta_z w_\zeta) \\ \mu_{eff} &= \mu_{lam} + \mu_{tur} \end{aligned}$$

When appropriate these thin-layer equations are replaced by the slender-layer Navier-Stokes equations, in which viscous terms are retained in two directions.

NUMERICAL PROCEDURE

The numerical procedure is discussed in two parts, the first of which outlines the implicit procedure used to solve the governing equations, and the second of which explains the methodology for coupling the solutions on the various grid systems. The former is outlined briefly, as the procedure is relatively well known. The latter is discussed in detail, as the procedure for applying tricubic interpolation to overset curvilinear grid schemes is new.

Implicit Scheme

The governing equations are solved with an implicit approximately factored scheme[14], following linearization in time. This scheme is modified to include nonlinear artificial dissipation terms suggested by Jameson et al.[15] and to accommodate a hole recognition mechanism required for overlapping grid systems. This mechanism is implemented with i_b for

"iblanking."[16] Consequently, the numerical scheme is the following for the thin-layer equations:

$$\begin{aligned} & T_\xi(I + i_b h\delta_\xi \Lambda_\xi^n + i_b \mathcal{D}_{\xi j}) \\ & N_{\xi\eta}(I + i_b h\delta_\eta \Lambda_\eta^n + i_b \mathcal{D}_{\eta k}) \\ & N_{\eta\zeta}(I + i_b h\delta_\zeta \Lambda_\zeta^n + i_b \mathcal{D}_{\zeta l} \\ & -i_b h Re^{-1}\delta_\zeta J^{-1}\hat{\Lambda}_M^n)T_\zeta^{-1}\Delta\hat{Q}^n \\ = i_b h(\delta_\xi \hat{E}^n + \delta_\eta \hat{F}^n + \delta_\zeta \hat{G}^n - Re^{-1}\delta_\zeta \hat{S}^n) & \\ & +i_b(\mathcal{D}_{\xi j} + \mathcal{D}_{\eta k} + \mathcal{D}_{\zeta l})\hat{Q}^n \end{aligned}$$

The nonlinear artificial dissipation operator, $\mathcal{D}$, is

$$\mathcal{D}_{\xi j} = \nabla_\xi(\sigma_{j+1}J_{j+1}^{-1} + \sigma_j J_j^{-1})(\epsilon_j^{(2)}\Delta_\xi J_j - \epsilon_j^{(4)}\Delta_\xi \nabla_\xi \Delta_\xi J_j),$$

where Δ and ∇ are forward and backward difference operators, respectively, σ_j are scaling coefficients obtained from the spectral radii of the flux Jacobians, and ϵ_j are the second and fourth order dissipation coefficients. The Λ's are the eigenvalue matrices of the flux Jacobians, the T's are the eigenvector matrices, and $N_{\xi\eta} = T_\xi^{-1}T_\eta$. The δ's are the spatial difference operators applied to the eigenvalue matrices, h is the timestep, and J_j's are the Jacobians of the coordinate transformation. The above equation describes the complete differencing scheme at an interior point of the grids. Boundary points, in the far field and at solid walls, and at interior hole boundaries require special care to insure that fourth order operators in the above expression are converted automatically to one sided or second order differences. These strategies are automated by the use of the iblank switch i_b at each node to signal whether to solve for the dependent variables at that node. Note that when two grids overlap, the presence of a solid body associated with one of the grids creates a hole in the interior of the other grid.

Tricubic Patch Interpolation Scheme

Linear interpolation mechanisms have been successful and easy to use for coupling the solutions on different grid systems. Their chief advantages are that they require a small number of data points (bi-linear requires data at the four corners of a two-dimensional cell, and trilinear requires data at the 8 corners of a three-dimensional cell), thus making their formulation relatively simple, and they have the interesting (and in some cases desirable) property that the value of a function at a linearly-interpolated point can never exceed the value at any surrounding corner point. This is, essentially, a monotonicity condition, and is a safe choice in the presence of shocks. The chief disadvantage of linear interpolation schemes is that they are only first-order accurate, and thus are a poor choice of interpolant when describing smooth but nonlinear functions. Cubic interpolation, by contrast, preserves the curvature present in the interpolated functions and is thus more accurate.

The problem of interpolation may be stated as follows: Given a function (typically flow data) defined in some domain at the nodes of a curvilinear coordinate system, find the function value at a given arbitrary point or set of points in this domain. Interpolation may be

performed in the physical domain in which the function values are at non-equispaced (x, y, z) locations of a curvilinear coordinate system. However, interpolation in this domain is inefficient in terms of storage and computational effort compared with that in the computational domain (ξ, η, ζ). In this domain the function values are known at the equispaced knots of a rectilinear grid, which allows a simplification in the interpolation procedure. All that is necessary is to transform the given (x, y, z) set of points to their (ξ, η, ζ) equivalents.

The procedure used to find the (ξ, η, ζ) interpolation point is as follows: a multivariate Newton-Raphson iterative procedure is used to invert the given (x, y, z) position to the corresponding (ξ, η, ζ) position. This also requires an interpolation procedure to obtain the initial guess and subsequent improved guesses.

Interpolants in curvilinear coordinate systems

First, the cell containing the point to interpolate is found via a search algorithm. Then for convenience the cell is translated so that one corner is located at the origin of (ξ, η, ζ) space. That is, calling the point we seek to interpolate (x^*, y^*, z^*), (ξ^*, η^*, ζ^*) will be an element of the unit cube in (ξ, η, ζ) space.

Next, the following iterative procedure is used. Since (x, y, z) is available at all (ξ, η, ζ) node points in computational space, form the three position function interpolants

$$x \simeq X(\xi, \eta, \zeta)$$

$$y \simeq Y(\xi, \eta, \zeta)$$

$$z \simeq Z(\xi, \eta, \zeta)$$

Then, invert this set of equations at the given (x^*, y^*, z^*) for the respective (ξ^*, η^*, ζ^*) using multivariate (three-dimensional) Newton-Raphson iteration. Starting with an initial guess

$$(\xi^o, \eta^o, \zeta^o),$$

solve the above system for (ξ^*, η^*, ζ^*) using the iteration

$$\begin{bmatrix} \Delta\xi^{n+1} \\ \Delta\eta^{n+1} \\ \Delta\zeta^{n+1} \end{bmatrix} := \left[\frac{\partial(X, Y, Z)}{\partial(\xi, \eta, \zeta)}\right]^{-1} \cdot \begin{bmatrix} \Delta x^n \\ \Delta y^n \\ \Delta z^n \end{bmatrix}$$

with

$$\begin{bmatrix} \Delta x^n \\ \Delta y^n \\ \Delta z^n \end{bmatrix} := \begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} - \begin{bmatrix} X(\xi^n, \eta^n, \zeta^n) \\ Y(\xi^n, \eta^n, \zeta^n) \\ Z(\xi^n, \eta^n, \zeta^n) \end{bmatrix}$$

which will be used to produce

$$\begin{bmatrix} \xi^{n+1} \\ \eta^{n+1} \\ \zeta^{n+1} \end{bmatrix} := \begin{bmatrix} \xi^n \\ \eta^n \\ \zeta^n \end{bmatrix} + \begin{bmatrix} \Delta\xi^{n+1} \\ \Delta\eta^{n+1} \\ \Delta\zeta^{n+1} \end{bmatrix}$$

Assuming a 1-1 mapping, the inverse of the Jacobian exists[16,17] and so it is possible to solve for the $(\Delta\xi, \Delta\eta, \Delta\zeta)$ vector. In general (assuming a "close enough" initial guess) the iteration will converge quadratically[18,19] to the desired (ξ^*, η^*, ζ^*), i.e., we will have (ξ^*, η^*, ζ^*) such that

$$\begin{bmatrix} x^* \\ y^* \\ z^* \end{bmatrix} = \begin{bmatrix} X(\xi^*, \eta^*, \zeta^*) \\ Y(\xi^*, \eta^*, \zeta^*) \\ Z(\xi^*, \eta^*, \zeta^*) \end{bmatrix}$$

Note that the assumption of a 1-1 mapping is violated in practice typically at topological singularities (such as polar axes of ellipsoidal grids) of the coordinate transformation. At such points, the Jacobian will be (theoretically) zero, but more likely (computationally) extremely small. If the Jacobian is not identically zero, it will frequently still be possible to continue the iteration to a successful conclusion. More information may be determined by finding the singular value decomposition (SVD: $J = UDV^t$ where $D = diag(\Lambda, 0, ...0) = diag(\lambda_1, ...\lambda_k, 0, ..., 0)$, λ_i being the singular values of J, $k = rank(J)$) of the Jacobian matrix. Singular values of zero will reveal, in a stable manner, the rank of the Jacobian matrix[20]. These may indicate that in at least one of the coordinate directions ξ, η, or ζ, a change in coordinate does not produce any change in x, y, or z. This will be true, for example, if the current position is on a singular line. The SVD may be then followed by singular value damping or singular value truncation[21], followed by construction of the "pseudo-inverse" $(J^+ = V\Lambda^{-1}U^t)$. The effect of small or zero singular values is to make the iteration unstable since it is the reciprocals of these singular values that are used in the formation of the pseudo-inverse. Singular value damping or truncation methods serve to stabilize the iteration by limiting the effect of these small or zero singular values. We will show later how the elements of the required Jacobian matrix may be obtained.

The Tricubic Interpolation

In the above procedure for finding the (ξ, η, ζ) position of a given (x, y, z) point, no mention was made of which particular interpolation method will be used. Any interpolation mechanism could be inserted. We have in fact started with trilinear interpolation coefficients (the (ξ^*, η^*, ζ^*) offsets) and have used them as the initial guess for a Newton-Raphson iteration using tricubic interpolation for the base interpolants.

The four coefficients of a one-dimensional cubic polynomial may be uniquely specified given two adjacent function values and the two derivatives of the function at those points. These two derivatives are themselves calculated from discrete data by central differences, and thus four function values are required. In three dimensions, 4^3 function values are required to produce a combination of 64 values of the function, its first derivatives, second, and third mixed partial derivatives. These in turn are placed in a $4 \times 4 \times 4$ tensor which is then multiplied by the three respective cubic blending polynomials, one each for x, y, and z. These cubic blending polynomials are themselves functions of the tricubic interpolation coefficients, that is, the (ξ^*, η^*, ζ^*) offsets. A locally defined cubic polynomial interpolant approximation to the discrete function is thus produced. Note that this interpolant is C^1 continuous across adjacent cells.

For each cell in 3-D containing an interpolation point, sixty-four function values will be required. That is, if the bottom, left-hand, forward corner of the cell in question has index (i,j,k), then values at all points $\{i-1,i,i+1,i+2\}\times\{j-1,j,j+1,j+2\}\times\{k-1,k,k+1,k+2\}$ are used. These are differenced as needed: first partials at each corner of the interpolation cell (24), second mixed partials at each corner (24), and third mixed partials at each corner (8). These 56 partial derivatives, along with the 8 corner function values are placed into a $4\times4\times4$ tensor B according to the arrangement of Stanton et al[22]:

$$B_1=\begin{bmatrix} f(0,0,0) & f(0,1,0) & f_\eta(0,0,0) & f_\eta(0,1,0) \\ f(1,0,0) & f(1,1,0) & f_\eta(1,0,0) & f_\eta(1,1,0) \\ f_\xi(0,0,0) & f_\xi(0,1,0) & f_{\xi,\eta}(0,0,0) & f_{\xi,\eta}(0,1,0) \\ f_\xi(0,0,0) & f_\xi(1,1,0) & f_{\xi,\eta}(1,0,0) & f_{\xi,\eta}(1,1,0) \end{bmatrix}$$

$$B_2=\begin{bmatrix} f(0,0,1) & f(0,1,1) & f_\eta(0,0,1) & f_\eta(0,1,1) \\ f(1,0,1) & f(1,1,1) & f_\eta(1,0,1) & f_\eta(1,1,1) \\ f_\xi(0,0,1) & f_\xi(0,1,1) & f_{\xi,\eta}(0,0,1) & f_{\xi,\eta}(0,1,1) \\ f_\xi(1,0,1) & f_\xi(1,1,1) & f_{\xi,\eta}(1,0,1) & f_{\xi,\eta}(1,1,1) \end{bmatrix}$$

$$B_3=\begin{bmatrix} f_\zeta(0,0,0) & f_\zeta(0,1,0) & f_{\eta,\zeta}(0,0,0) & f_{\eta,\zeta}(0,1,0) \\ f_\zeta(1,0,0) & f_\zeta(1,1,0) & f_{\eta,\zeta}(1,0,0) & f_{\eta,\zeta}(1,1,0) \\ f_{\xi,\zeta}(0,0,0) & f_{\xi,\zeta}(0,1,0) & f_{\xi,\eta,\zeta}(0,0,0) & f_{\xi,\eta,\zeta}(0,1,0) \\ f_{\xi,\zeta}(1,0,0) & f_{\xi,\zeta}(1,1,0) & f_{\xi,\eta,\zeta}(1,0,0) & f_{\xi,\eta,\zeta}(1,1,0) \end{bmatrix}$$

$$B_4=\begin{bmatrix} f_\zeta(0,0,1) & f_\zeta(0,1,1) & f_{\eta,\zeta}(0,0,1) & f_{\eta,\zeta}(0,1,1) \\ f_\zeta(1,0,1) & f_\zeta(1,1,1) & f_{\eta,\zeta}(1,0,1) & f_{\eta,\zeta}(1,1,1) \\ f_{\xi,\zeta}(0,0,1) & f_{\xi,\zeta}(0,1,1) & f_{\xi,\eta,\zeta}(0,0,1) & f_{\xi,\eta,\zeta}(0,1,1) \\ f_{\xi,\zeta}(1,0,1) & f_{\xi,\zeta}(1,1,1) & f_{\xi,\eta,\zeta}(1,0,1) & f_{\xi,\eta,\zeta}(1,1,1) \end{bmatrix}$$

where f is the function being interpolated, evaluated at the indicated corner of the unit cube.

The cubic blending polynomials are given by

$$\begin{aligned} \alpha_1(\xi) &= 2\xi^3-3\xi^2+1 \\ \alpha_2(\xi) &= -2\xi^3+3\xi^2 \\ \alpha_3(\xi) &= \xi^3-2\xi^2+\xi \\ \alpha_4(\xi) &= \xi^3-\xi^2 \end{aligned}$$

The function f interpolant can now be given by

$$f(\xi,\eta,\zeta)\simeq\sum_{l,m,n}\alpha_l(\xi)\alpha_m(\eta)\alpha_n(\zeta)B_{lmn}$$

We may now see that the partial derivatives of the curvilinear coordinate system position functions X, Y, and Z, which are required in the Jacobian matrix for the Newton-Raphson iteration, may be easily formed. First, analytically differentiate the expression immediately above with respect to ξ, η, and ζ, and then evaluate the resulting partials for the position functions X, Y, and Z at the current (ξ,η,ζ) estimate within the Newton-Raphson process.

That is, the partial derivatives of the position function interpolants are interrogated at the current (ξ, η, ζ) Newton-Raphson estimate.

Note that interpolation of discontinuities requires the use of trilinear rather than tricubic interpolation to insure monotonic behavior of the interpolated values.

NUMERICAL EXPERIMENTS

The Reynolds averaged, Navier-Stokes equations with the Baldwin-Lomax turbulence model are solved to determine flows past a generic oblique-wing aircraft (Fig. 2). Two and three overlapping grid systems are used, respectively, for a wing-fuselage combination and a wing-pedestal-fuselage combination. The flow is at a Mach number of 1.4, a Reynolds number of 4,000,000, and an angle of attack of 14°. The wing is positioned as follows: sweep angle = 65°; bank angle = 5°, with the leading tip banked down; pitched down by 7.8°; wing pivot point is at 39.0% of the mid-chord; the wing attachment point is at 60.0% of the length of the fuselage from the nose. The solutions are communicated from one grid system to the neighboring one by interpolation. The tricubic interpolation scheme is applied for two overlapping grids; and the trilinear interpolation scheme is used for three grids. The wing and the fuselage flowfields are computed using the thin-layer approximation; and the pedestal flowfield is computed using the slender-layer approximation.

The complexity of computations and flow fields is indicated by presenting sample results (Figs. 3-6). The effect of trilinear and tricubic interpolations on convergence to a steady state is discussed, and a comparison of the accuracy of the two methods demonstrates that the latter is more accurate. Timing results for the MPMG code are then presented to demonstrate the speedup achieved in the elapsed (wall-clock) time by multiprocessing.

Tricubic Patch Application

Figures 3 and 4, respectively, show the overlapping wing and fuselage grid systems and surface flow patterns. Solutions are obtained at each time step separately on these grid systems, and are communicated between them by the tricubic interpolation scheme. The smoothness in the solution across the boundary of one grid system into another is higher for the tricubic interpolation than for the trilinear. This smoothness affects the convergence to steady state. With the trilinear scheme, the residues of the discrete governing equations cannot be reduced beyond about four orders of magnitude, but with the tricubic scheme, these residues are reduced more than six orders of magnitude. In these experiments all other parameters were kept the same.

The L_1, L_2, and L_∞ norms of error and relative error for a known nonlinear function are less for the tricubic than for the trilinear interpolation. Numerous tests were devised to test and compare the accuracy of the tricubic and trilinear interpolation schemes. All of these tests resulted in the same conclusion. We describe here a test which is particularly revealing. First, a solution was obtained for the fuselage grid ($95 \times 83 \times 50$) alone using the MPMG code. The resulting gradients were strong at the nose and tail of the fuselage. At each point in the grid, the solution was interpolated from alternate surrounding points. Both trilinear

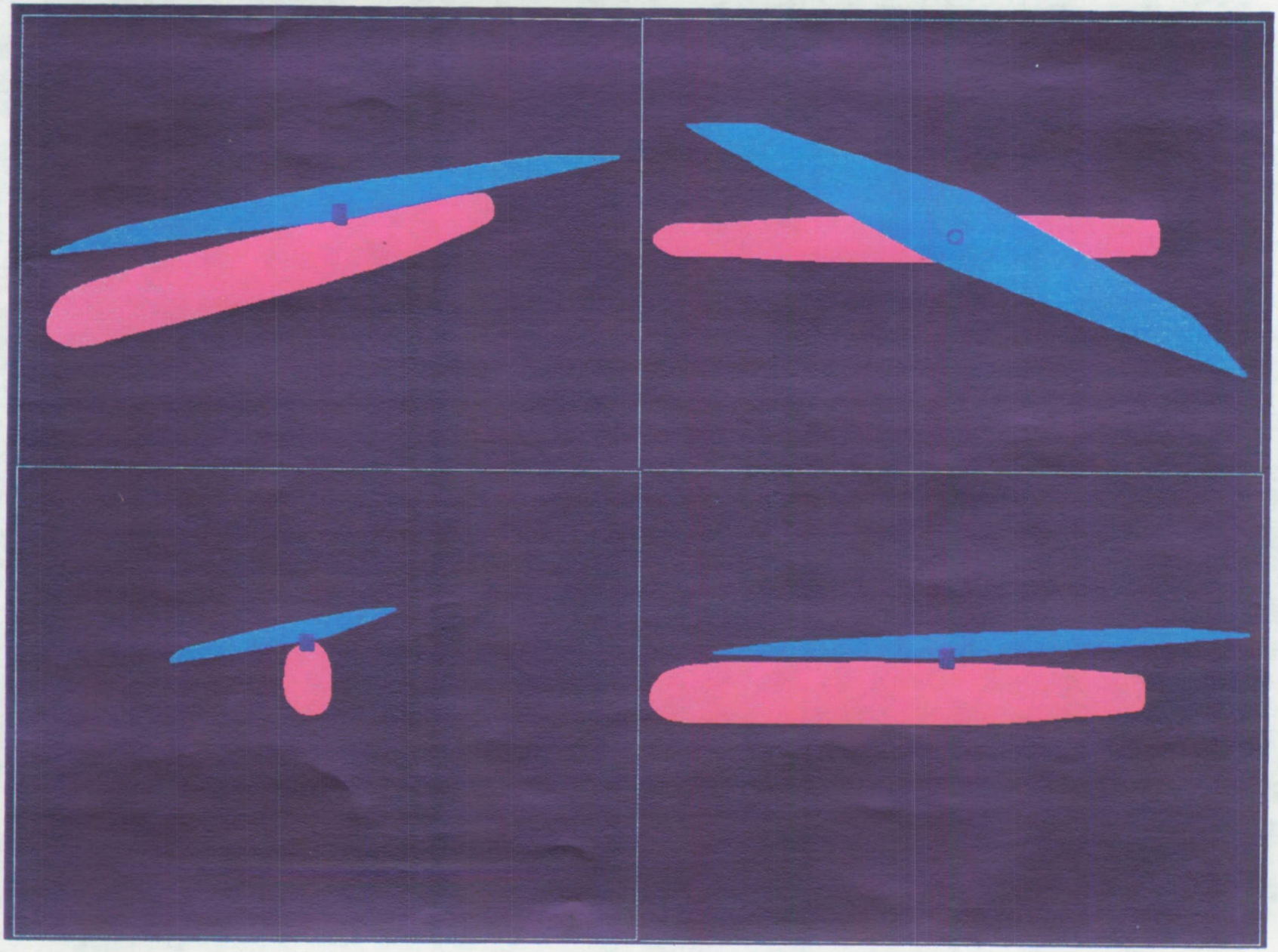

Figure 2: Four views of a generic oblique-wing aircraft.

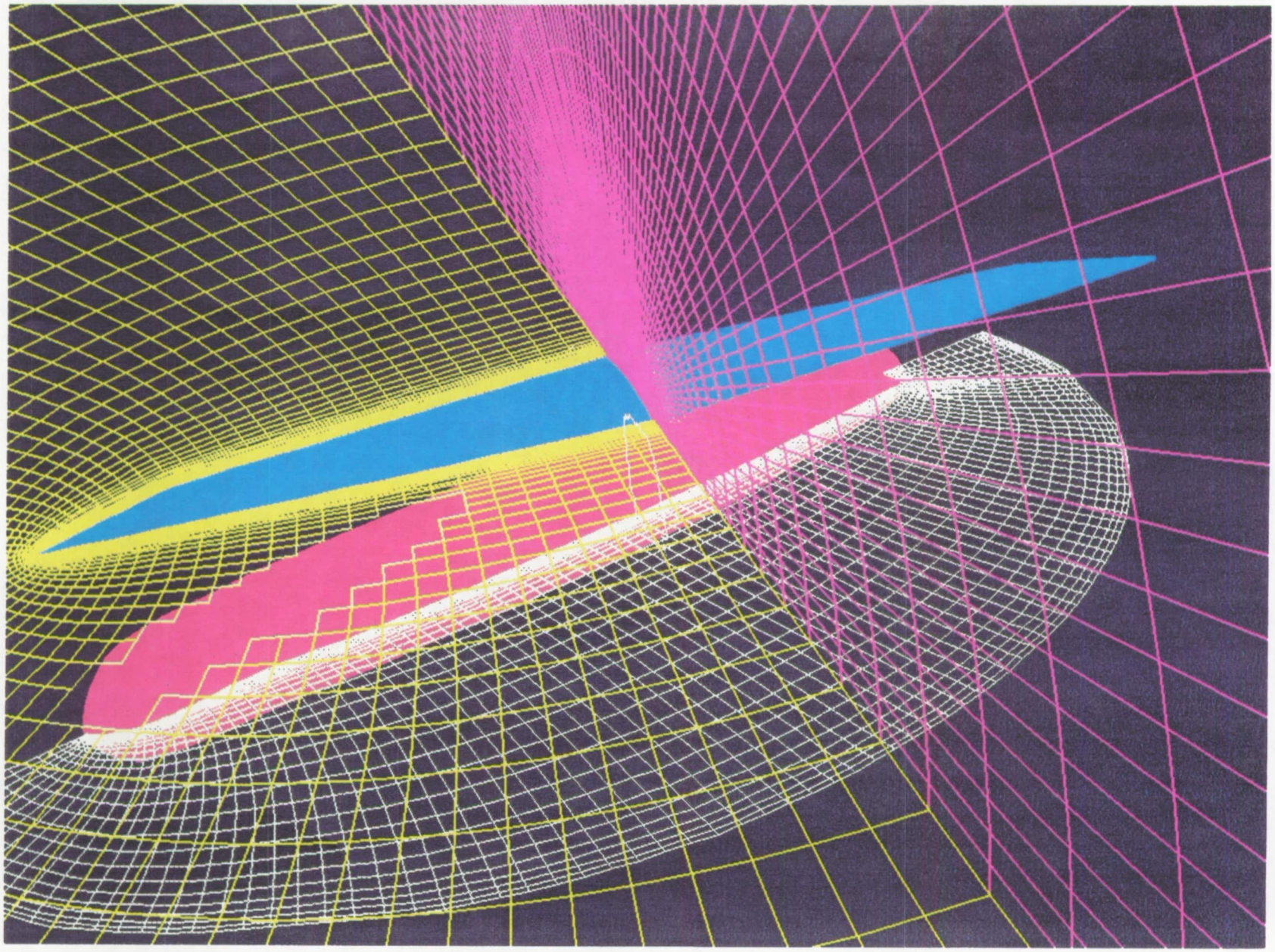

Figure 3: Two grid systems with holes, one around the wing and the other around the fuselage.

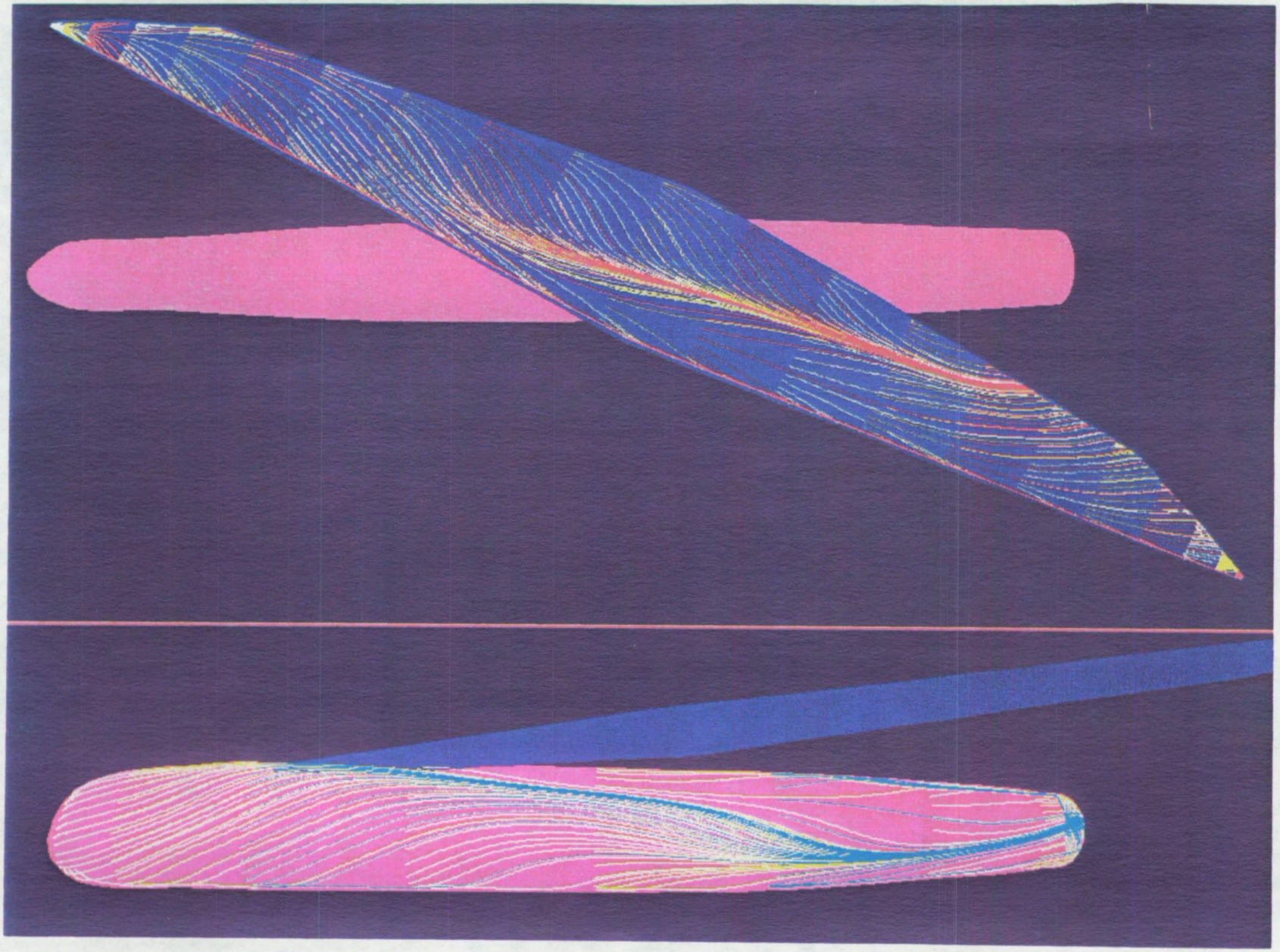

Figure 4: Surface flow patterns on a wing-fuselage combination.

and tricubic interpolation were used in this manner and the results compared to the original solution at this point. (Note that the values used in this test are Q variables, not scaled by transformation Jacobian.) Virtually every measure of error is smaller for the tricubic scheme than for the trilinear scheme (Appendix G). This is, of course, no surprise. In general, the error is reduced by between one third and one half. The errors showed no particular correlation with position in the grid. The tricubic interpolation, therefore, can be advantageous for complex computational aerodynamics problems requiring multiple grids. The obvious disadvantage of using tricubic interpolation is that it requires roughly eight times more computational effort than the trilinear interpolation. Multiprocessing the interpolations between different grid systems would reduce the elapsed time for such computations.

MPMG Application

The MPMG code is applied to solve flow past the generic oblique-wing aircraft. The flow conditions are those considered above. Three overlapping grid systems are used, one for each of the following parts of the aircraft: wing (389,500 grid-points), fuselage (229,190 grid-points), and pedestal, the wing support between the wing and the fuselage (223,860 grid-points). The governing equations are solved concurrently on these separate grids by multitasking as three concurrent processes. The results are then updated on the overlapping grid regions by interpolation in the fourth process. Some details of the grid systems are shown in Fig. 5. A sample of the result is presented in Fig. 6 showing particle traces around the pedestal.

The averaged cpu seconds per minute from numerous MPMG runs of a three-grid problem, both in single-process mode (no multitasking) and multi-process mode (the multitaskable portion of the computation proceeding as three parallel processes), are presented in Table 1. The average effective speedup is 2.1. This factor is consistent with total elapsed time trials made for this case. When considering these results, it is important to remember that the three-processor values include some time spent executing in a serial only mode on a single processor. Appendix F gives timings of each of these runs. Variations in timing values are due to variations in machine load conditions. The timing information is presented in the form of cpu seconds consumed per wall-clock (elapsed) minute.

Processes	CPU seconds/minute
1	34
3	71

Table 1. Increase in cpu with multiprocessing

Under the following conditions, the number of cpu seconds per minute which a job will obtain while multitasking is roughly equal to the number of processes requested times the number of seconds per minute which a single process job will get. First, the machine is saturated, that is, the number of processes exceeds the number of processors. This implies that there is no idle compute time. Second, all experiments performed, regardless of the number of processes, are run at the same priority. Third, the total number of all user jobs

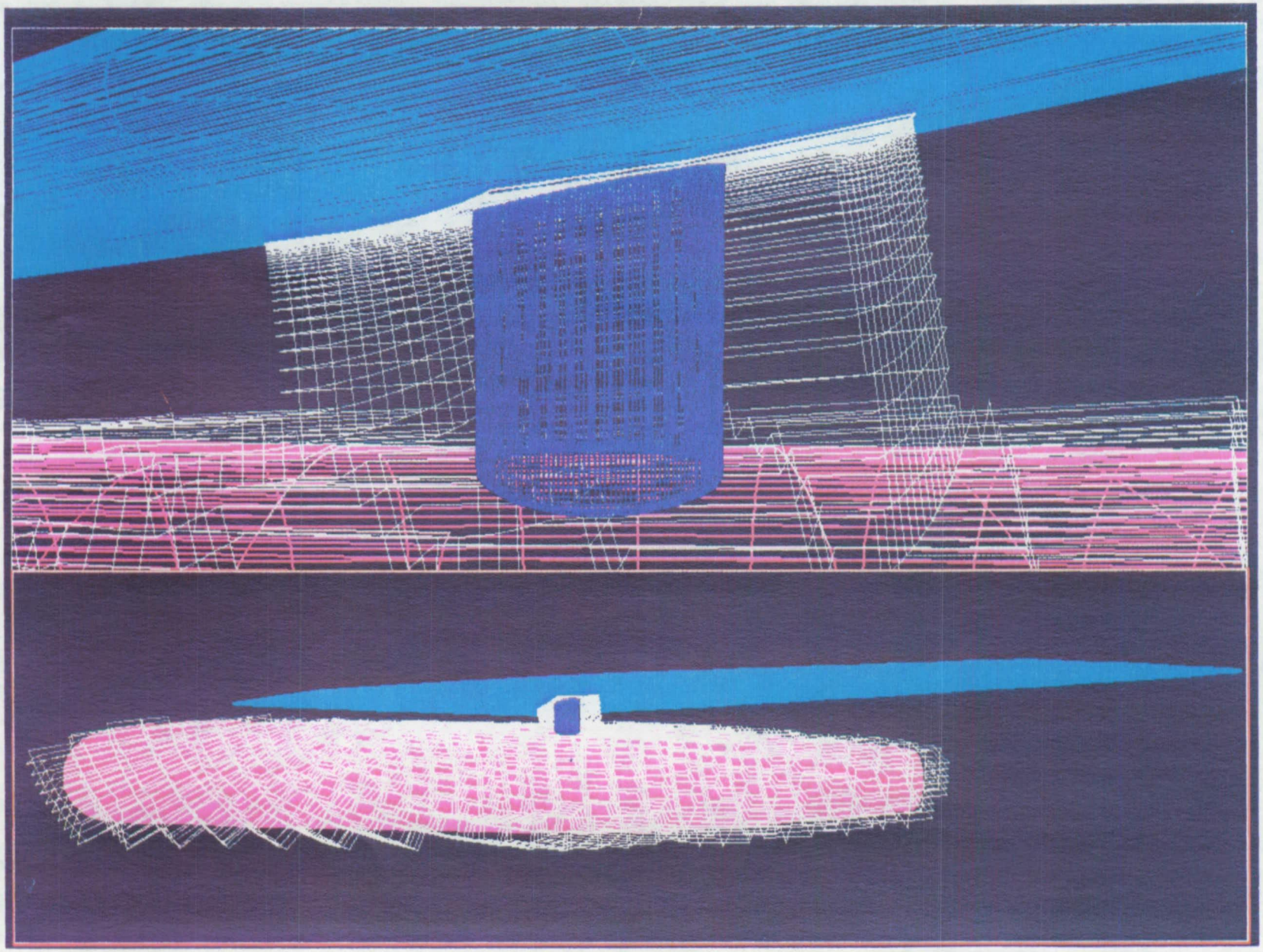

Figure 5: Some details of three grid systems with holes around the oblique-wing aircraft.

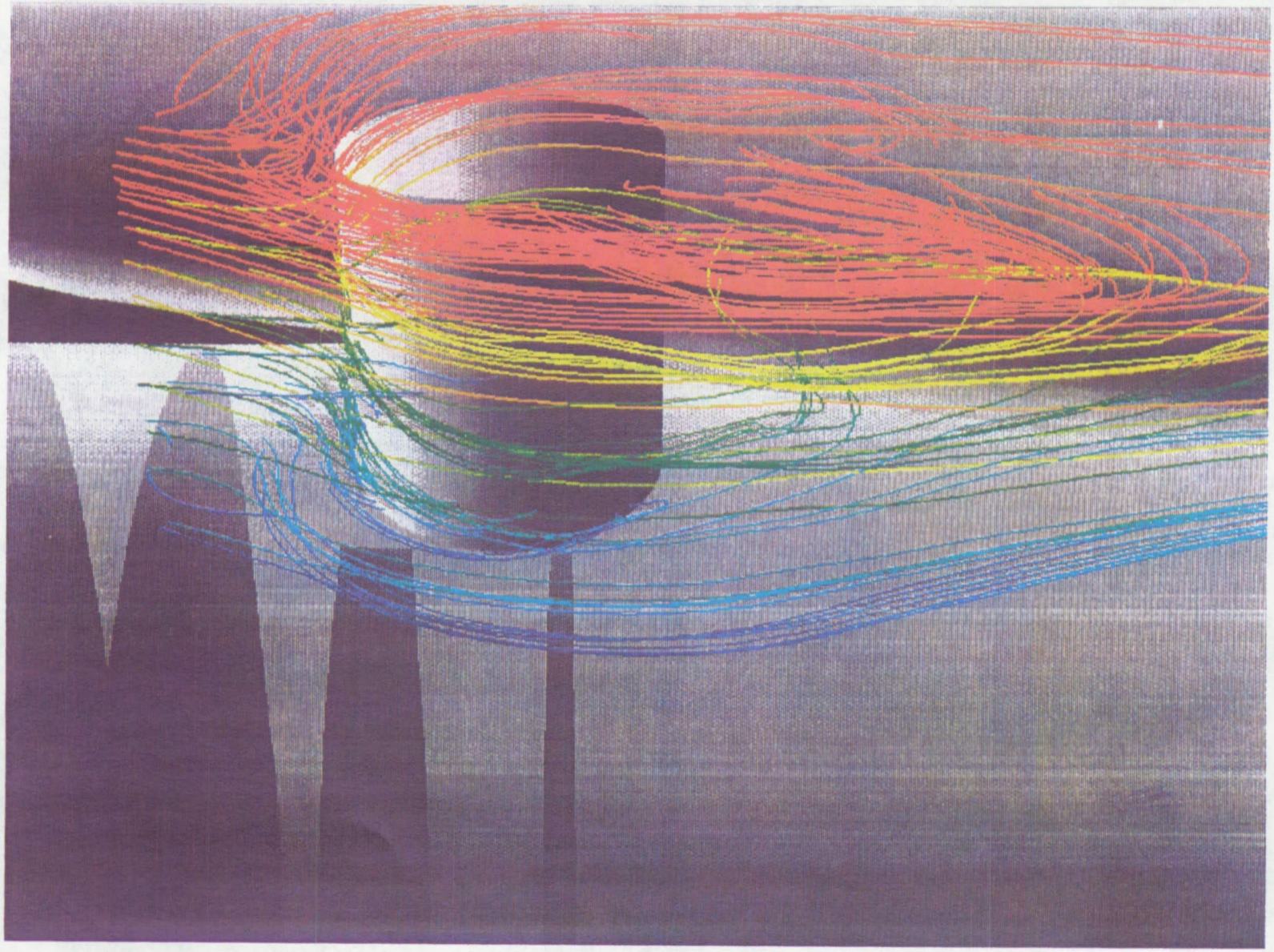

Figure 6: Particle traces around the pedestal.

is considerably larger than the number of processes on which the flow code is multitasked.

In order to estimate the fraction of the flow solver code which was multitaskable, the above averaged timing results were applied in the following manner. The code is divided into two portions. On the first portion, computation proceeds serially, that is, in one process only. On the second portion, computation proceeds in parallel, in this case, in three concurrent processes. It is the extent of this section which is to be determined.

Consider a simple example in which 75% of the code is parallelizable onto three processes. For this job, the 25% nonparallelizable portion will complete in 25% of the total elapsed time required for the entire job to run as a single process. The parallelizable 75%, run on three processes, will complete three time more rapidly than it would have if run as a single process. Therefore, the total parallelizable version will complete in 50% (that is, 25% plus one third of 75%) of the elapsed time required for the nonparallelizable version. This is Amdahl's law.

In the present case, we know the elapsed times for the single processor and three processor flow code computations (subject to some statistical uncertainty), and from these we seek the fraction of the code which is serially executed only (the fraction which is effectively multitasked will then be 1 minus the serial fraction) This is the inverse of Amdahl's law:

$$f = \frac{mr - 1}{(m - 1)}.$$

Here f is the fraction of the code that cannot be multitasked, m is the number of concurrent processes in the multitasked part, and r is the ratio of the multitasked to nonmultitasked elapsed times.

This formula assumes that the m multitaskable processes all perform the same amount of work and thus, when started simultaneously, will all finish at the same time. This is not necessarily the case for the MPMG code, and, in fact, is not true for the test cases described in this report, where the amount of work in the multitaskable processes is proportional to the different grid sizes, i.e., the number of points in each grid. To find the fraction of code which is executed serially only, we therefore use the following modified formula:

$$f_s = \frac{-s_m + r\sum_{i=1}^{m} s_i}{\sum_{i=1}^{m-1} s_i}$$

where r is, as above, the ratio of multitasked to serial-only elapsed times, and the s_i's are the sizes or relative sizes of the multitaskable pieces of work, with the mth being the largest. Therefore, based on the measured elapsed times (Table 1) and setting s_i equal to the various grid sizes, we conclude that about 3.1% of the total computation was executed in serial mode only. We have used here for the ratio r of multitasked to nonmultitasked elapsed times the ratio of the values given in Table 1 for cpu seconds per minute of elapsed time, which is $1/2.1 = 0.48$. Note that the overhead for multitasking in the MPMG code was found to be negligible.

One can estimate the decrease in multitasking efficiency due to the differences in grid sizes. By using Amdahl's law with the assumption that $f = 3.1\%$ we find that the resultant ideal ratio of multitasked to serial elapsed times will be about 0.35, rather than the value of 0.48 achieved with the differing mesh sizes.

Multitasking at the relatively coarse grain level in the MPMG code was found to be quite efficient on the Cray-2 computer, in that the overhead incurred by the multitasking library was very small. By contrast, the autotasking facility (automatic Cray microtasking library invocation) provided on the Cray-2 and Cray Y-MP computers is relatively inefficient, based on the first author's experiments with autotasking 3-D Navier-Stokes, implicit flow solvers. Those experiments showed that, though elapsed time was decreased, a penalty in total cpu time consumed of up to forty percent was incurred. This penalty varies depending on system load, job priority, and other factors and is caused by details in the autotasking/microtasking library implementation.

CONCLUDING REMARKS

1) Although multiprocessing is essential for significantly improving the productivity of computational aerodynamicists, it is hardly ever used in computational aerodynamics.

2) Judicious use of multiprocessing allows efficient use of computer-system resources.

3) An approach is presented and demonstrated that multitasks existing supercomputer Fortran programs with relative ease using "C" for the main program.

4) A significant improvement in turnaround time is demonstrated and the theoretical basis for it is explained.

5) Efficient use of future multiheaded supercomputers will typically require multiprocessing flow solvers.

6) Tricubic interpolations can reduce error in grid coupling mechanisms compared to trilinear interpolations.

APPENDIX A

MULTITASKING IN THE ABSENCE OF COMMON BLOCKS

```
C     =============
      PROGRAM MHEAT
C     =============
      EXTERNAL SEIDEL
      INTEGER PROC1(2), PROC2(2)
      DIMENSION A(20000)

      CALL TSKTUNE (6HMAXCPU,4)
      PROC1(1) = 2
      PROC2(1) = 2
      IDIM = 100
      JDIM = 200
      CALL INITIA (A,IDIM,JDIM)
      ITCNT = 0
      ALOWDT = .0001
      DUMYLG = 100000.
      IL2R = 0
      IR2L = 1
 1000 CONTINUE
      CALL TSKSTART (PROC1,SEIDEL,IDIM,JDIM/2,ACHNG1,ALOWDT,
     &               ITS1,IR2L,A(1))
      CALL TSKSTART (PROC2,SEIDEL,IDIM,JDIM/2+1,ACHNG2,ALOWDT,
     &               ITS2,IL2R,A(9901))
      CALL TSKWAIT (PROC2)
      CALL TSKWAIT (PROC1)
      CALL AVCNTR (IDIM,A(9801))
      CALL SEIDEL (IDIM,JDIM/2,ACHNG1,DUMYLG,IDUM,IR2L,A(1))
      CALL SEIDEL (IDIM,JDIM/2+1,ACHNG2,DUMYLG,IDUM,IL2R,A(9901))
      AMOST = AMAX1(ACHNG1,ACHNG2)
      ITCNT = ITCNT + 1
      WRITE(*,9000) ITCNT,ITS1,ACHNG1,ITS2,ACHNG2
 9000 FORMAT(1X,I5,2(5X,I5,2X,E13.7))
      IF (AMOST .GT. ALOWDT) GO TO 1000
      CALL OUTPRN (A,IDIM,JDIM)
      END
C     =================
      SUBROUTINE SEIDEL (IDIM,JDIM,TCHANG,ALOWDT,ITS,IDIREC,T)
C     =================
      DIMENSION T(IDIM,JDIM)
      DATA  OMEGA / 1.93 /

      IF (IDIREC .EQ. 0) THEN
```

```
         JSTART = 2
         JEND   = JDIM-1
         JINC   = 1
      ELSE
         JSTART = JDIM-1
         JEND   = 2
         JINC   = -1
      ENDIF
      ITS = 0
1000  CONTINUE
         TCHANG = -1E40
         DO 100 J=JSTART,JEND,JINC
         DO 100 I=2,IDIM-1
            TPREV = T(I,J)
            T(I,J) = (T(I-1,J) + T(I+1,J) + T(I,J-1) + T(I,J+1))/4.
            T(I,J) = TPREV + OMEGA*(T(I,J) - TPREV)
            TDIFF = ABS(TPREV-T(I,J))
            IF (TDIFF .GT. TCHANG) THEN
               TCHANG = TDIFF
               ICHNG = I
               JCHNG = J
               TMPCHG = T(I,J)
            ENDIF
  100    CONTINUE
         ITS = ITS + 1
         IF (TCHANG .GT. ALOWDT) GO TO 1000
      RETURN
      END
C     =================
      SUBROUTINE AVCNTR (IDIM,T)
C     =================
      DIMENSION T(IDIM,3)

      DO 100 I=2,IDIM-1
         T(I,2) = (T(I,1)+T(I,3))/2.
  100 CONTINUE
      RETURN
      END
C     =================
      SUBROUTINE INITIA (T,IDIM,JDIM)
C     =================
      DIMENSION T(IDIM,JDIM)

      DO 50 J=2,JDIM-1
      DO 50 I=2,IDIM-1
```

```
         T(I,J) = 0.
   50 CONTINUE
      DO 100 I=1,IDIM
         T(I,1)    = 200.
         T(I,JDIM) = 200.
  100 CONTINUE
      DO 200 J=2,JDIM-1
         T(1,J)    = 200.
         T(IDIM,J) = 200.
  200 CONTINUE
      RETURN
      END
C     ================
      SUBROUTINE OUTPRN (A,IDIM,JDIM)
C     ================
      DIMENSION A(IDIM,JDIM)

      DO 100 I=1,100
         WRITE(*,*) I,A(I,50),A(I,100),A(I,150)
  100 CONTINUE
      RETURN
      END
```

APPENDIX B

MAKING FORTRAN COMMON BLOCK ADDRESSES AVAILABLE TO MAIN C PROGRAM

```
C
      SUBROUTINE FXTERN1
C
      COMMON/COMBLK1/ A,B,C
      COMMON/COMBLK2/ X(1000),Y(1000),Z(1000)
      COMMON/COMBLK3/ M(20),Q,R,S

      MESHNUM = 1
      CALL CXTERN (MESHNUM,A,X,M)
      RETURN
      END
```

APPENDIX C

OBTAINING ADDRESSES OF FORTRAN COMMON BLOCKS IN C

```
struct s_comblk1
```

```
          /* global declarations */
   {
   float a;
   float b;
   float c;
   };

struct s_comblk1 *ps_comblk1[NUMBER_OF_GRIDS];

          /* Pointer to common comblk1 */
          /* NUMBER_OF_GRIDS defined as */
          /* total number of grids */

struct s_comblk2
   {
   float x[1000];
   float y[1000];
   float z[1000];
   };

struct s_comblk2 *ps_comblk2[NUMBER_OF_GRIDS];

struct s_comblk3
   {
   int   m[20];
   float q;
   float r;
   float s;
   };

struct s_comblk3 *ps_comblk3[NUMBER_OF_GRIDS];

void  CXTERN (mesh_number
              ,plocal_comblk1
              ,plocal_comblk2
              ,plocal_comblk3)
          /* Note:  mesh_number is pointer to int */
          /* since it is passed from Fortran routine */
          /* and therefore must be an address; */
          /* same for all these pointers */
   int *mesh_number;
   int *plocal_comblk1;
   int *plocal_comblk2;
   int *plocal_comblk3;
{
   extern struct s_comblk1    *ps_comblk1[];
   extern struct s_comblk2    *ps_comblk2[];
```

```
    extern struct s_comblk3    *ps_comblk3[];
           /* typecast and assign address to */
           /* approp pointers to structures */
           /* These pointers to structures are */
           /* global and thus available to main */
    ps_comblk1[*mesh_number] = (struct s_comblk1 *) plocal_comblk1;
    ps_comblk2[*mesh_number] = (struct s_comblk2 *) plocal_comblk2;
    ps_comblk3[*mesh_number] = (struct s_comblk3 *) plocal_comblk3;
}
```

APPENDIX D

MAKING ADDRESS OF BULK COMPUTATIONAL WORK SPACE AVAILABLE TO C MAIN PROGRAM

```
C
      SUBROUTINE FALLOC
C
C  Before compilation, totalmem must be replaced with the
C        following number: 37*(total num of nodes from all grids)
C        + 3*(sum of all jdimc + sum of all kdimc).
C        A slightly excessive overestimate would be
C        38*(total num of nodes from all grids)
C  ALLMEM(38), which appears here is replaced by
C        ALLMEM(38*(+i1*j1*k1+i2*j2*k2+...)),
C        the substitution being made by sed script falloc.sed.
C  CMNADDRSS, i.e., common address
C  In this example, the grid sizes are: 92*82*50, 65*82*43,  35*164*39

      COMMON /WORKMEM/ ALLMEM(38*(95*82*50+65*82*43+35*164*39))

      CALL CMNADDRS (ALLMEM)
      RETURN
      END
```

APPENDIX E

WORK SPACE MEMORY ALLOCATOR

```
int     *p_maxj[NUMBER_OF_GRIDS]
           /* global declarations */
        ,*p_maxk[NUMBER_OF_GRIDS]
        ,*p_maxl[NUMBER_OF_GRIDS]
float   *p_x[NUMBER_OF_GRIDS]
        ,*p_y[NUMBER_OF_GRIDS]
```

```
        ,*p_z[NUMBER_OF_GRIDS]

void    givmem (mesh_number,jmax,kmax,lmax)
   int mesh_number
      ,jmax
      ,kmax
      ,lmax;
{
   int maxdim = jmax * kmax * lmax;
   p_maxj[mesh_number] = (int *) word_alloc ( 1 );
   p_maxk[mesh_number] = (int *) word_alloc ( 1 );
   p_maxl[mesh_number] = (int *) word_alloc ( 1 );
   p_x[mesh_number] = (float *) word_alloc (maxdim);
   p_y[mesh_number] = (float *) word_alloc (maxdim);
   p_z[mesh_number] = (float *) word_alloc (maxdim);
}

float  *word_alloc (numwords)
          /* memory given to solver arrays */
   int numwords;
{
   extern float  *p_allmem;
   static int     n_already_given = 0;
   int            i;
   i = n_already_given;
   n_already_given += numwords;
   return (p_allmem + i);
}
```

APPENDIX F

CPU TIMING TRIALS

Cpu Seconds per Minute

1 Processor	3 Processors
29	57
33	66
32	65
33	58
32	64
30	62
32	56
33	52
33	46
36	50
33	52
34	51
34	56
34	65
35	55
35	65
35	64
36	65
34	74
35	71
34	63
40	115
32	125
31	113
35	108
31	76
37	79
34	77
38	73
40	77
	80
	72
	79
	79
	77

APPENDIX G

A COMPARISON OF INTERPOLATION ERRORS

RMS errors:
cub: 8.557E-3, 6.395E-3, 4.539E-3, 5.482E-3, 2.109E-2
lin: 1.400E-2, 9.904E-3, 9.054E-3, 1.139E-2, 3.468E-2

average abs(error):
cub: 3.520E-3, 2.503E-3, 2.118E-3, 2.485E-3, 9.077E-3
lin: 6.936E-3, 4.894E-3, 4.883E-3, 5.817E-3, 1.752E-2

stand dev abs(error):
cub: 7.800E-3, 5.884E-3, 4.014E-3, 4.887E-3, 1.904E-2
lin: 1.216E-2, 8.611E-3, 7.625E-3, 9.793E-3, 2.993E-2

L-1 norm of errors:
cub: 57.617, 40.975, 34.670, 40.675, 148.556
lin: 113.520, 80.105, 79.911, 95.195, 286.834

L-2 norm of errors:
cub: 1.094, 0.818, 0.580, 0.701, 2.698
lin: 1.791, 1.267, 1.158, 1.457, 4.437

L-inf norm of errors
cub: 6.781E-2, 0.104, 3.921E-2, 5.700E-2, 0.173
lin: 1.000E-1, 0.113, 6.256E-2, 1.062E-1, 0.261

L-1 norm of relative errors:
cub: 54.767, 49.739, 6879.075, 832.340, 53.174
lin: 114.164, 124.852, 10341.191, 1540.516, 108.532

L-2 norm of relative errors:
cub: 0.969, 2.777, 545.013, 149.249, 0.933
lin: 1.782, 11.316, 853.034, 217.192, 1.787

L-inf norm of relative errors:
cub: 7.652E-2, 1.252, 230.220, 105.581, 6.915E-2
lin: 9.610E-2, 6.741, 342.255, 100.158, 0.126

average abs(relative errors):
cub: 3.346E-3, 3.039E-3, 0.420, 5.086E-2, 3.249E-3
lin: 6.976E-3, 7.629E-3, 0.631, 9.413E-2, 6.631E-3

stand dev abs(relative errors):
cub: 6.802E-3, 2.149E-2, 4.239, 1.165, 6.533E-3
lin: 1.206E-2, 8.813E-2, 6.638, 1.695, 1.230E-2

REFERENCES

[1] Cray-2 Multitasking Programmer's Manual, Cray Research, Inc., No. SM-2026, March, 1986, pp. 2-14.

[2] Buzbee, B. L., and Sharp, D. H., "Perspectives on Supercomputing," *Science*, Vol. 227, 1985, pp. 591-597.

[3] Ware, W.: "The Ultimate Computer," *IEEE Spectrum*, Mar. 1972, pp. 84-91.

[4] Liewer, P. C., Zimmerman, B. A., Decyk, V. K., and Dawson, J. M., "Application of Hypercube computers to Plasma Particle-in-cell Simulation Codes," Proceedings of Fourth Int'l Conf. on Supercomputing, Vol. II, 1989.

[5] Lin, C. S., "Particle-in-cell Simulations of Wave Particle Interactions using the Massively Parallel Processor," Proceedings of Fourth Int'l Conf. on Supercomputing, Vol. II, 1989.

[6] Tuccillo, J. J., "Numerical Weather Prediction on the Connection Machine," Proceedings of Fourth Int'l Conf. on Supercomputing, Vol. II, 1989.

[7] Jespersen, D. C., and Levit, C., "A Computational Fluid Dynamics Algorithm on a Massively Parallel Computer," AIAA 9th Computational Fluid Dynamics Conference, June, 1989, Buffalo, N.Y.

[8] Smith, B. F. and Miller, R. M., "A Computational Approach to Galaxy Formation and Evolution, Proceedings of Second Int'l Conf. on Supercomputing, Vol. II, 1987.

[9] Taylor, P. R., and Bauschlicher, Jr., C. W., "Computational Chemistry on the Cray-2," Proceedings of Second Int'l Conf. on Supercomputing, Vol. II, 1987.

[10] Taylor, P. R., and Bauschlicher, Jr., C. W., "Strategies for obtaining the maximum performance from current supercomputers," Theoretica Chimica Acta, 1987, Vol. 71, pp. 105-115.

[11] Andrich, P., Delecluse, P., Levy, C., and Madec, B., "A Multitasked General Circulation Model of the Ocean," Proc. of Fourth Int'l Symp. on Science and Engineering on Cray Supercomputers, Minneapolis, Minn., Oct., 1988.

[12] Chevrin, R. M., Do Global, "A Climate Modeling Imperative as Well as a Microtasking Directive," Proc. of Fourth Int'l Symp. on Science and Engineering on Cray Supercomputers, Minneapolis, Minn., Oct., 1988.

[13] Unix Programmers Manual, Vol. 2, Bell Labs, 1979.

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

[15] Jameson, A., Schmidt, W., and Turkel, E., "Numerical Solutions of the Euler Equa-

tions by Finite Volume Methods Using Runge-Kutta Time-Stepping Schemes," AIAA 14th Fluid and Plasma Dynamics Conference, AIAA Paper 81-1259, Palo Alto, Ca., 1981.

[16] Benek, J.A., Steger, J.L., Dougherty, F.C., and Buning, P.G., Chimera: "A Grid-Embedding Technique," AEDC Report TR-85-64, Arnold Eng. Dev. Ctr., Tenn., Apr., 1986.

[17] Sokolnikoff, I.S., *Advanced Calculus*, Chap. XII, McGraw-Hill, 1939.

[18] Carnahan, B., Luther, H.A., Wilkes, J.O., *Applied Numerical Meth.*, J. Wiley, Inc., 1969.

[19] Young, D.M., Gregory, R.T., *A Survey of Numerical Mathematics*, Addison-Wesley, 1972.

[20] Buning, P.G., PLOT3D computer program, CELL3 Routine, NASA Ames Rsrch Ctr.

[21] Marquardt, D.W., "Generalized Inverses, Ridge Regression, Biased Linear Estimation, and Nonlinear Estimation," Technometrics, Vol. 12, pp. 591-612, 1970.

[22] Stanton, E.L., Crain, L.M., Neu, T.F., "A Parametric Cubic Modelling System For General Solids of Composite Material," Int. J. Num. Meth. in Eng., Vol. 11, pp. 653-670.

NASA
National Aeronautics and Space Administration

Report Documentation Page

1. Report No.	2. Government Accession No.	3. Recipient's Catalog No.
NASA TM-102806		

4. Title and Subtitle
Multiprocessing on Supercomputers for Computational Aerodynamics

5. Report Date
May 1990

6. Performing Organization Code

7. Author(s)
Maurice Yarrow (Sterling Federal Systems, Inc., Palo Alto, CA)
and Unmeel B. Mehta

8. Performing Organization Report No.
A-90121

10. Work Unit No.
505-60

9. Performing Organization Name and Address
Ames Research Center
Moffett Field, CA 94035-1000

11. Contract or Grant No.

13. Type of Report and Period Covered
Technical Memorandum

12. Sponsoring Agency Name and Address
National Aeronautics and Space Administration
Washington, DC 20546-0001

14. Sponsoring Agency Code

15. Supplementary Notes
This Technical Memorandum is a revised version of AIAA Paper 90-0337 presented at the 28th Aerospace Sciences Meeting, Reno, Nevada, January 8-11, 1990.
Point of Contact: Unmeel B. Mehta, Ames Research Center, MS 202A-1
Moffett Field, CA 94035-1000 (415) 604-5548 or FTS 464-5548

16. Abstract
Very little use is made of multiple processors available on current supercomputers (computers with a theoretical peak performance capability equal to 100 MFLOPs or more) in computational aerodynamics to significantly improve turnaround time. The productivity of a computer user is directly related to this turnaround time. In a time-sharing environment, the improvement in this speed is achieved when multiple processors are used efficiently to execute an algorithm. We apply the concept of multiple instructions and multiple data (MIMD) through multitasking via a strategy which requires relatively minor modifications to an existing code for a single processor. Essentially, this approach maps the available memory to multiple processors, exploiting the C-Fortran-Unix interface. The existing single processor code is mapped without the need for developing a new algorithm. The procedure for building a code utilizing this approach is automated with the Unix stream editor. As a demonstration of this approach, a Multiple Processor Multiple Grid (MPMG) code is developed. It is capable of using nine processors, and can be easily extended to a larger number of processors. This code solves the three-dimensional, Reynolds averaged, thin-layer and slender-layer Navier-Stokes equations with an implicit, approximately factored and diagonalized method. The solver is applied to a generic oblique-wing aircraft problem on a four processor Cray-2 computer. A tricubic interpolation scheme is developed to increase the accuracy of coupling of overlapped grids. For the oblique-wing aircraft problem, a speedup of two in elapsed (turnaround) time is observed in a saturated time-sharing environment.

17. Key Words (Suggested by Author(s))	18. Distribution Statement
Computational aerodynamics, Supercomputers Multiprocessing, Computational fluid dynamics Multitasking, Tricubic interpolation	Unclassified-Unlimited Subject Category – 02

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of Pages	22. Price
Unclassified	Unclassified	36	A03