AIAA-94-2265

AN EFFICIENT HIGHER ORDER ACCURATE PARALLEL ALGORITHM FOR AEROACOUSTIC APPLICATIONS

Thomas S. Chyczewski and Lyle N. Long

Department of Aerospace Engineering
The Pennsylvania State University

25th AIAA Fluid Dynamics Conference

June 20-23, 1994 / Colorado Springs, CO

For permission to copy or republish, contact the American Institute of Aeronautics and Astronautics
370 L'Enfant Promenade, S.W., Washington, D.C. 20024

AN EFFICIENT HIGHER ORDER ACCURATE PARALLEL ALGORITHM FOR AEROACOUSTIC APPLICATIONS

Thomas S. Chyczewski*

Lyle N. Long†

Aerospace Engineering, The Pennsylvania State University

Abstract

In this paper the problems associated with achieving good performance from a Computational Aeroacoustics (CAA) code on the CM-5 (in an entirely data parallel approach) are addressed. The CAA algorithm requires solving the full 3-D Navier Stokes equations using high order spatial and temporal differencing and nonreflecting boundary conditions, among other things, to preserve the integrity of the acoustic wave solution. The spatial differencing is accomplished with sixth order central differences. A fourth order Runge-Kutta scheme is used for time advancement. Each grid point needs data from eighteen neighboring points to perform the spatial differencing. Near the boundaries biased stencils are required to maintain high order accuracy. These large, varied stencils present two problems from a parallel processing point of view. One is the large amount of data that has to be communicated and the associated communication time. The other is the large number of stencils required to maintain high accuracy near the boundaries. Specialized treatment of each of the different stencils would provide for a very poorly load balanced (and consequently inefficient) code. The communication time overhead is significantly decreased by organizing the data into blocks and communicating on a block basis instead of an element basis. The specialized boundary treatment problem is all but eliminated by using generalized global stencil arrays so that for a given physical boundary condition all points can be evaluated in parallel. Results indicate a significant performance improvement over a code that doesn't use the methodologies proposed here. These results are presented in the form of a detailed timing breakdown per iteration.

*Graduate Assistant, Student Member, AIAA

†Associate Professor, Senior Member, AIAA

Copyright 1994 by Chyczewski and Long. Published by the American Institute of Aeronautics and Astronautics, Inc. with permission

1 Introduction

The development of jet noise prediction tools has received much attention since the Federal Aviation Administration began regulating the amount of noise produced by aircraft. These tools have become an invaluable component of the development of the High Speed Civil Transport (HSCT) and are also important in military applications.

In the past, the practical prediction strategies consisted of Lighthills acoustic analogy [1, 2, 3] and analytical approaches [4, 5, 6]. While these methods give great insight into the noise generation process, there are limitations to the problems they can solve and they require a substantial amount of empirical data.

With the advent of parallel processors and the promise of teraflop performance, an alternative method is becoming more attractive. This approach uses the full Navier-Stokes equations to directly calculate the noise generation and the consequent propagation of sound to the far field (see [7] for an example of a Navier Stokes code coupled with a moving Kirchoff surface). It contrasts with the acoustic analogy approach that uses an inhomogeneous wave equation as the governing equation with the right hand side being the modelled noise source. The direct simulation approach requires significant computer resources since the algorithm

must contain all of the physics associated with the noise generation process.

Unfortunately, along with the introduction of parallel processors, other issues are introduced. The primary issue is how to get the best performance from a parallel processor for the problem at hand. To answer this question, one must determine if the work load is evenly distributed across the processors and if the overhead time due to communication is minimized.

For message passing machines, good load balancing is often achieved when using a domain decomposition parallelization strategy. These machines allow multiple instructions to be performed on multiple data (MIMD). This allows, for example, one processor to perform inflow boundary conditions while another processor does outflow conditions (see [8] for example). On data parallel machines, load balancing is more challenging since there is quite often only a single instruction working at a given time (SIMD). Therefore, inflow boundaries and outflow boundaries must be computed sequentially. An exception is found in [9] where the properties of the upwind scheme are used to reduce the special attention required by the boundary to simple array assignments.

The problem of reducing the communication time can be approached in many ways. Some researchers have significantly reduced the amount of communication at the expense of increasing the number of floating point operations (see [10] for example). This tradeoff is beneficial since as the number of processors increases, the time to perform the floating point operations decreases linearly while the efficiency at which data is communicated deteriorates. On data parallel machines, such as the CM-5, the communication time can be reduced by a clever distribution of the data across the processors.

To get great performance from a data parallel code that predicts noise is a significant challenge. Critics of the data parallel approach claim that good performance is only achievable for very simple problems. In this paper, some approaches are presented that significantly improve the performance of a data parallel code that tackles a nontrivial problem.

Following this section, the numerical issues that impose the parallel difficulties are discussed. This is followed by a discussion of the methods used to decrease the cost of these difficulties. Then some numerical and timing results are presented and discussed. Finally, some conclusions are drawn.

2 Numerical Algorithm

The long term objective of this research is to develop an efficient algorithm that directly computes the noise generated from turbulent jets. To perform such direct calculations, a successful Computational Aeroacoustics (CAA) algorithm must address some difficult issues which significantly increase the number of floating point operations and the communication on parallel processors. Among these issues is the requirement of a time accurate account of the turbulence. Since it is the unsteady nature of turbulence that generates the noise in jets, statistical models that modify the mean quantities are not appropriate. Instead, a time accurate approach, such as a Large Eddy Simulation (LES) method, is required. Although not yet implemented, LES will eventually be incorporated in the code.

Another issue that must be addressed is numerical dispersion. The numerical algorithm must propagate all signals at the sound speed even under strenuous grid point per wavelength conditions. This is accomplished by using sixth order central spatial differencing to calculate the fluxes and a fourth order Runge-Kutta to integrate in time. For a 3-D calculation, the sixth order flux evaluation requires information from 18 neighboring points (see figure 1). This significantly increases the floating point operations and the communication. There are no significant parallel issues to be addressed concerning the Runge-Kutta time integration. This segment of the code requires no communication and is very efficient. The only drawback is that it requires 4 stages to complete a timestep.

The algorithm must also be capable of allowing waves to exit the computational domain without reflections that could alter or contaminate the interior wave solution. Boundary conditions developed by Tam [11] are adopted here. They consist of a system of equations that represent the asymptotic solution of the linearized Euler equations. These equations are applied at the three ghost cell layers on the outer boundary of the domain that are required for the sixth order differencing. The stencil used to discretize the boundary equations is optimized for each grid point. Figure 2 shows a typical stencil near a corner of the grid. The num-

ber of optimized stencils is on the order of 100 for a 3-D problem. If each boundary were to be treated sequentially, very poor performance would be achieved.

The code also incorporates the artificial selective damping function developed by Tam [12]. This function damps the high wavenumber waves that are spuriously generated by the central differencing. It is appropriate for CAA calculations since it is optimized to provide a large amount of damping to high wavenumber waves while leaving low wavenumber waves relatively unaffected. It uses a 19 point stencil (for the 3-D problem) with optimized coefficients to provide the desired damping.

Also implemented in the code are solid wall boundary conditions (both viscous and inviscid). They are applied using a single layer of ghost cells and are consequently susceptible to the inefficiencies found at the far field. There is an additional complexity associated with the nature of the boundary condition. The typical boundary condition requires the pressure derivative normal to the wall set equal to zero. For the most general situation, the normal to the wall can be along any of the curvilinear coordinate directions or anywhere in between. For an implementation on a parallel computer, it is desirable to perform all calculations simultaneously without regard to the direction of the normal or the curvilinear coordinate along which the wall lies.

The typical timestep consists first of building up the flux from contributions of interior inviscid, viscous, boundary inviscid and artificial dissipation. Then the solution is updated in a 4 stage Runge-Kutta fashion. Figure 3 shows the algorithm flow chart. The first steps are to update the wall variables and to calculate the derivatives of the primitive variables and store them for use in evaluating the boundary inviscid fluxes and the viscous shear stresses. Then the shear stresses are calculated and the flux build up begins. Finally the solution is updated.

3 Implementation

The code was optimized to run efficiently using a data parallel approach on the CM-5. It is written in CMFortran, which is very similar to High Performance Fortran (HPF) [13]. To get good performance two issues must be addressed and resolved. One is to minimize the communication and the other is to do as much as possible in parallel.

3.1 Communication Considerations

On a data parallel machine, the compiler does its best to give each element of an array its own processor. When there are more array elements than processors, the compiler generates 'virtual processors' by splitting a number of physical processors so each element gets its 'own' processor. When a derivative is evaluated, information that is needed from neighboring grid points is obtained by communication (the 'cshift' command in CMFortran). In a 1-D problem using sixth order differencing, the seven point stencil needs information from 6 neighbors. In pseudo CMFortran the derivative evaluation would look like :

$$\frac{\partial q}{\partial x} = \frac{1}{\Delta x}\sum_{i=-3}^{3} c_i \;\; cshift(q,1,i) \qquad (1)$$

Where cshift(q,1,i) means shift array q along axis 1 (there is only one axis for this 1-D case), i positions.

As a result, there are 12 instances of an array value moving across a processor (cshift(q,1,-3) is 3 instances since the data is moving from processor '-3' to processor '0'). One way of reducing this expense is to telescope the data to processor '0'. For example, the 1-D code would look like:

$$S_{0} = q$$
$$S_{-1} = cshift(S_{0},1,-1)$$
$$S_{-2} = cshift(S_{-1},1,-1)$$
$$S_{-3} = cshift(S_{-2},1,-1)$$
$$S_{+1} = cshift(S_{0},1,+1)$$
$$S_{+2} = cshift(S_{+1},1,+1)$$
$$S_{+3} = cshift(S_{+2},1,+1)$$

$$\frac{\partial q}{\partial x} = \frac{1}{\Delta x}\sum_{i=-3}^{3} c_i S_i \qquad (2)$$

This reduces the 12 instances to 6. This approach was found to decrease the communication time by about 40 %.

Another approach is to organize the data in blocks. This reduces communication even further to 2 instances by moving larger amounts of data with each call. Again consider a 1-D problem for simplicity. Using the standard approach, the q array would be dimensioned as

real q(Jmax)

and through the compiler directive

layout q(:news)

would be spread across the processors as described before. Using the block approach, the above is replaced with

real q(3,$\frac{Jmax}{3}$)

and

layout q(:serial,:news)

The layout directive tells the compiler that for each j, q(1:3,j) is to be stored on a single processor. In other words, the first dimension of q is stacked on a single processor and the second dimension is spread out across the processors. Now the derivative looks like:

```
L = cshift(q,2, -1)
R = cshift(q,2,+1)
```

$$\frac{\partial q(1,:)}{\partial x} = \frac{1}{\Delta x} * (c_{-3}L(1,:)+$$

$$c_{-2}L(2,:) + c_{-1}L(3,:) + c_0 q(1,:)+$$

$$c_{+1}q(2,:) + c_{+2}q(3,:) + c_{+3}R(1,:))$$

$$\frac{\partial q(2,:)}{\partial x} = \frac{1}{\Delta x} * (c_{-3}L(2,:)+$$

$$c_{-2}L(3,:) + c_{-1}q(1,:) + c_0 q(2,:)+$$

$$c_{+1}q(3,:) + c_{+2}R(1,:) + c_{+3}R(2,:))$$

$$\frac{\partial q(3,:)}{\partial x} = \frac{1}{\Delta x} * (c_{-3}L(3,:)+$$

$$c_{-2}q(1,:) + c_{-1}q(2,:) + c_0 q(3,:)+$$

$$c_{+1}R(1,:) + c_{+2}R(2,:) + c_{+3}R(3,:)) \quad (3)$$

The code has now become more complicated since each of the 3 elements of the first dimension of q must be evaluated on a separate line. It also has the disadvantage of performing 3 sequential operations. The advantage is that there are only two communication calls.

3.2 Parallel Considerations

The ideal parallel algorithm would require no communication and would keep all processors busy all of the time, ie, it would be well load balanced. Although most, if not all, scientific applications require communication, its expense can be reduced by methods described in the previous section. In a data parallel approach, good load balancing is difficult to obtain since there is essentially one thread of instructions that all processors are executing (Thinking Machines is working towards eliminating this problem by providing a 'global/local' execution model [14]).

In high order accuracy schemes, the load balancing problem becomes significant when one considers the number of stencils that are required. For example, consider a 1-D problem that solves the wave equation with nonreflecting boundary conditions to sixth order accuracy. Figure 4 illustrates what the grid might look like. Using a standard approach, there are 7 instructions that must be executed sequentially to determine the flux. First, the interior flux has to be determined (j=4 to Jmax-3). Then each of the 6 ghost cells (1 to 3 and Jmax-2 to Jmax) must be evaluated using the boundary condition. Each of these points has a different stencil, starting with a fully forward stencil at j = 1 and ending with a fully backwards stencil at j = Jmax. This results in 7 instructions (1 interior and 6 boundary) that are executed sequentially.

The present approach reduces these 7 sequential steps to 2 using global stencil arrays. The seven point stencil is replaced by a 13 point stencil that provides backwards, biased backwards, central, biased forward and forward stencils automatically where needed. A similar approach was introduced in [15]. In that work, the biasing (or lack of biasing) is not modified by the stencil arrays. Instead, the stencil is identical everywhere, but the stencil coefficients are locally modified to adapt to solving either the interior or boundary equation.

As an example of how the stencil arrays are implemented here, consider the 1-D problem. The stencil arrays are illustrated in figure 5. The derivative, using the telescoping communication approach for simplicity, looks like

$S_0 = q$
$S_{-1} = cshift(S_0, 1, -1)$
$S_{-2} = cshift(S_{-1}, 1, -1)$

$$S_{-3} = cshift(S_{-2}, 1, -1)$$
$$S_{-4} = cshift(S_{-3}, 1, -1)$$
$$S_{-5} = cshift(S_{-4}, 1, -1)$$
$$S_{-6} = cshift(S_{-5}, 1, -1)$$
$$S_{+1} = cshift(S_{0}, 1, +1)$$
$$S_{+2} = cshift(S_{+1}, 1, +1)$$
$$S_{+3} = cshift(S_{+2}, 1, +1)$$
$$S_{+4} = cshift(S_{+3}, 1, +1)$$
$$S_{+5} = cshift(S_{+4}, 1, +1)$$
$$S_{+6} = cshift(S_{+5}, 1, +1)$$

$$\frac{\partial q}{\partial x} = \frac{1}{\Delta x} \sum_{i=-6}^{6} a_i S_i \tag{4}$$

Consider j=1, which requires a fully forward stencil. a(-6 to -1) are set to zero since they correspond to points behind j=1. a(0 to 6) are assigned values to provide a sixth order accurate forward derivative. For points 4 to Jmax-3, a(-6 to -4 and 4 to 6) are set to zero and a(-3 to 3) provide a sixth order central stencil. Although it requires additional communication and memory, the time saved by performing everything in 2 steps is quite significant. On a 32 processor CM-5, the extra memory requirements of the global stencil arrays reduces the largest problem size from a 108x108x108 grid to a 96x96x96 grid. This is considered to be an acceptable penalty for the types of problems that will be investigated in the future.

The global stencil array approach becomes more significant when a 3-D problem is considered. The number of stencils is on the order of 100 and this method still performs everything in 2 steps (provided all locations have the same physical boundary conditions).

When the block communication approach is used, the communication expense is not significantly increased since it still requires only 2 communication calls, but the size of the block increases from 3x3x3 to 6x6x6.

The global stencil arrays also significantly improves the performance of the solid wall boundary conditions. Consider, for example, the case of calculating the flow from a rectangular jet. If the walls are included in the calculation they provide eight surfaces for which a wall boundary condition is imposed (if the wall is infinitely thin). Using the stencil arrays and defining arrays that contain normals to the wall for the surface points, all eight of these surfaces can be computed in parallel. They even require less time to compute than the far field boundary conditions since the wall condition is identical at all locations.

Improving performance is not the only thing to be considered when developing an idea for a parallel algorithm. An issue that should also be addressed is how the implementation of these ideas affects the legibility of the code. If the code becomes very complicated, then it may affect debugging time as well as make it difficult to distribute it to other users. Of the two ideas presented here, one increases the complexity of the code while the other makes it very easy to read. As mentioned earlier, introducing the block layout significantly increases the number of lines of code required to take a derivative. This is due to the fact that there are three instructions that have to be executed sequentially(for the 3x3x3 block case illustrated in equation 3). This admittedly has made the code more difficult to debug, but not prohibitively so. On the other hand, introducing the stencil arrays significantly decreases the complexity of the code. With the arrays in place, the advantages of the CM Fortran syntax can be fully exploited making the code easy to read. In fact, this advantage alone can be considered significant enough, without considering the performance advantages.

4 Results and Discussion

Most of the results in this section were obtained from the 128 processor CM-5 located at the National Aerodynamic Simulation Lab (NAS). The peak speed of each processor is 128 MFLOPS. For most of each day the machine is partitioned into three smaller machines: two 32 processor partitions and one 64 processor partition. In this configuration each of the partitions is timeshared. Therefore the 32 and 64 processor results presented here were obtained from a timeshared environment and may not be precise. They are presented since they do faithfully represent the trends associated with increasing problem size and machine size. The 128 processor results were obtained from the machine when it was configured in a single 128 processor partition. In this configuration the code can be run in a dedicated mode.

The code was compiled using the CMF version 2.1 compiler. A variety of compiler flags were used to get better performance. This includes the optimizer (-O), the vector units(-vu (de-

fault)), nopadding (-nopad) and noaxisreordering (-noaxis). The NAS CM5 is configured with 4 vector units per processing node. When using the vector units the code treats each vector unit as a processing node. The processing nodes themselves provide communication and operating system services. The nopadding flag forces the compiler to distribute the arrays in quanta of 1 element. The default is 8 elements (the size of a vector register). This could leave many vector units without a piece of the array and provide for a poorly load balanced code. This is particularly true when the data is organized in blocks. The noaxis flag also affects the way the arrays are distributed across the processors and was found to have a positive affect on performance.

Some results from a 64 processor CM-5E are also presented. This upgraded version of the CM-5 has a processor peak speed of 160 MFLOPS.

All times presented here are CM busy times that are output from the CM timer routines.

Figures 6 through 11 illustrates some preliminary results from a test problem that is being used to validate the code. Figure 6 shows a grid for a cylinder that has 75 points in the radial direction and 120 around the cylinder. For this 2-D problem, a symmetry condition is applied in the third dimension of the 3-D code. A Gaussian pressure pulse is placed one cylinder radius away from the leading edge of the cylinder as an initial condition. This is illustrated in figure 7 which shows pressure contours. For this test case, the viscous terms are turned off resulting in the Euler equations being solved. The CFL number used in these calculations is 0.07. As time progresses, this acoustic pulse propagates at the speed of sound and interacts with the cylinder. Figures 8 and 9 show pressure contours at different times in the simulation. In figure 8, the wave front has reached the cylinder and is partially reflected. In figure 9, the wave front has travelled completely around the cylinder. The lower amplitude reflected wave is approaching the left boundary by this time. The contours on the leeside surface of the cylinder are not physical and are likely due to insufficient artificial dissipation. Results for this test case are also presented in figures 10 and 11. The horizontal axis in these figures is a ray that begins at the surface of the cylinder ($r = -1$), goes through the origin of the source ($r = -2$) and terminates at the computational domain boundary ($r = -6$). Figure 10 contains results for times before the right running wave from the initial pulse has reached the cylinder surface. Figure 11 contains results for times after the right running wave is reflected into a left running wave. Again, some dissipation problems are evident at the wall in figure 11.

The rest of this section dissects a timestep and evaluates overall performance.

All of the times presented here were calculated by running the code for 10 timesteps and dividing the results by 10. The code was run on different occasions to evaluate repeatability of the results, which proved to be quite good. The test case used to determine the codes performance incorporates all of the features of the algorithm; 3-D, far field boundary conditions, viscous terms, artificial viscosity, and the solid wall boundary condition. Five faces of the computational domain are far field boundaries while the sixth is a solid wall. Identical times will be obtained independent of the wall geometry, although the FLOP performance will be affected.

The first part of the performance analysis determines which portions of the timestep require the most time. In figure 12 the time spent in each segment of the timestep is plotted as a function of the problem size for the 64 processor machine. 'Runge' is the time spent updating the solution, checking convergence, etc. 'Prim deriv' is the evaluation of derivatives of primitive variables used in boundary conditions and viscous fluxes. 'Boundary' is the calculation of the boundary conditions. 'Influx', 'Visflux' and 'Avisflux' are the calculations of the inviscid, viscous and artificial viscous fluxes respectively. As may be expected, 'Runge' performs very well. There are very few operations and little communication. Aside from the wall boundary condition, all other segments of the code require about the same amount of time. Ideally, one would like a relatively small amount of time spent on the boundary conditions since they are applied to a small portion of the grid. Also, since these boundary conditions are functions of the derivatives of the primitive variables and the time spent evaluating these derivatives is accounted for in a different segment, the boundary segment should be fast. This is not the case. It appears as though all of the logic associated with determining whether each face is an inflow or an outflow and then sequentially performing each of these boundary conditions requires significant time.

The value of the stencil arrays is emphasized by the performance of the wall boundary condition.

This is seen in figure 12 and is further illustrated in figure 13 which shows the GFLOPS achieved as a function of the problem size for the 64 processor machine. Even though a wall (or multiple walls) of any arbitrary shape, can be handled by these generalized boundary conditions, the performance of this segment of the code is such that the overall algorithm performance is only slightly compromised.

Figure 14 shows how the calculation time per grid point varies as a function of problem size and machine size. This time is determined by putting timers around all segments of the code except cshifts. The calculation time is observed to scale very well with machine size, as one might expect. As the machine size doubles, the calculation time approximately halves. There is also a desirable trend with problem size. Generally, there is a slight decrease in the time per grid point as the number of grid points increases.

Similar trends are also observed in figure 15 which shows the communication time. This time is determined by putting timers around cshifts. The 32 processor case scales the best with problem size. The slopes level off as the number of processors increases. Like the calcuations time, the communication time is halved as the number of processors doubles from 64 to 128. This is not true for all of 32 processor cases when comparing them to the 64 processor cases. For the smaller problem sizes, the scaling is approximately linear. But as the problem size grows, the 32 processor times quickly approach the 64 processor times. This is a consequence of the superior scaling of the 32 processor case with problem size.

Figure 16 shows the GFLOP performance. Overall, the trends are quite good. The performance scales approximately linearly with machine size. It also improves as the number of grid points increases. For each of the cases, the code gets 7.5% of the peak of the machine. This seemingly low number is actually quite good for a problem of this complexity. It is more than 8 times faster than the original implementation which did not use the strategies presented here. Even though the algorithm employs a block data layout, 50% of the time is spent communicating. If this time could be eliminated, then the percent peak achieved would double to 15%, which is about what the better CFD codes are getting on the CM.

Also presented in figures 14 through 16 is a data point taken from a 64 processor CM-5E that coincides with the largest 64 processor CM-5 case. The calculation and communication time decrease while the GFLOP performance increases by about 27%.

In all of the results presented so far, the problem sizes were selected so that the data can be evenly distributed across the vector units. There are six data points for each machine size. They correspond to precisely 1, 4, 7, 10, 13, and 16 blocks per vector unit. Figure 17 shows the effect of selecting problem sizes that can't be equally distributed across the vector units for the 64 processor case. This data is the same as the data presented in figure 16 for the 64 processor case, with the addition of two data points. When there are an integer number of blocks per vector unit, ie each vector unit has the same number of blocks, the performance is good. When the blocks cannot be distributed evenly across the vector units, ie, noninteger values of blocks per vector unit, inefficiencies arise due to poor load balancing. Some vector units will have more blocks than others. Since each block is 6x6x6, this means some vector units have 216 more points to work on than others. This performance deterioration imposes a constraint on the grid size in order to take advantage of the block layout. For the intent of this code, this constraint is not that much of an issue.

5 Conclusions

In this paper, some methods of improving the performance of a CAA code on the CM-5 are presented. One method is to use global stencil arrays to perform the differencing. This addresses the load balancing problem of parallel machines and is quite effective. It allows each processor to evaluate the high order derivative at any location in the interior and on the boundary simultaneously. Load balancing inefficiencies can also be introduced if the data cannot be evenly distributed across the processors. Results indicate a more than 20% decrease in performance if each processor does not have the same number of blocks.

Other methods are introduced to reduce the overhead due to communication. One approach is to perform the communication in increments and telescope the results to the node. The other is to arrange the data into blocks and perform communication on a block basis instead of an element basis. Results for the stencil arrays/blocks approach indi-

cate significant improvement over a baseline code that doesn't use these methods.

6 Acknowledgements

This work was funded by the NASA Graduate Student Research Program and by NASA Grant Nos. NAG-1-1470 and NAG-1-1367. Computer resources were provided by the National Aerodynamic Simulation Lab and by Thinking Machines Corporation.

References

[1] M.J. Lighthill, "Jet Noise", *AIAA Journal*, Vol. 1, no. 7, July 1963.

[2] W. Bailly, W. Bechara, P. Lafon, S. Candel, "Jet Noise Predictions Using a K-Epsilon Turbulence Model", AIAA 93-4412, 1993.

[3] C. Berman, G. Gordon, G. Karniadakis, S. Orszag, "Jet Turbulence Noise Computations", AIAA 93-4365, 1993.

[4] P.J. Morris, T.R.S. Bhat, G. Chen, "A Linear Shock Cell Model for Jets of Arbitrary Exit Geometry", *J. Sound and Vibration*, vol. 132(2), pp 199-211, 1989

[5] C.K.W. Tam, "Stochastic Model Theory of Broadband Shock Associated Noise from Supersonic Jets", *J. Sound and Vibration*, vol. 116, no. 2, July 22, 1987, pp. 265-302

[6] C.K.W. Tam, N.N. Reddy, "A Prediction Method for Broadband Shock Associated Noise from Supersonic Rectangular Jets", AIAA 93-4387, 1993.

[7] Y. Ozyoruk, L. Long, "A Navier-Stokes /Kirchoff Method for Noise Radiation from Ducted Fans," AIAA 94-0462, 1994

[8] T. Chyczewski, F. Marconi, R. Pelz, "Solution of the Euler and Navier Stokes Equations on Parallel Processors using a Transposed/Thomas ADI Algorithm," AIAA 93-3310, 1993

[9] Z. Weinberg, L. Long, "A Massively Parallel Solution of the Three Dimensional Navier-Stokes Equations on Unstructured, Adaptive Grids,", AIAA-94-0760, 1994

[10] E. Curchitser, R. Pelz, F. Marconi, "Solution of the Euler and Navier Stokes Equations on MIMD Multiprocessors Using Cyclic Reduction," AIAA 92-0561, 1992

[11] C.K.W. Tam, J.C. Webb, "Dispersion-Relation-Preserving Finite Difference Schemes for Computational Acoustics", *J. Comp. Phys.*, Aug. 1993

[12] C.K.W. Tam, Shen,H., "Direct Computations of Nonlinear Acoustic Pulses using High Order Finite Difference Schemes",AIAA 93-4325, 1993

[13] High Performance Fortran Forum. High Performance Fortran Language Specification Version 1.0. Scientific Programming, 2((1-2)):1-170, Spring and Summer 1993.

[14] CM Fortran Programming Guide, Version 2.1, January 1994, Thinking Machines Corporation.

[15] J. Myczkowski, M. Bromley, D. McCowan, "Extremely Fast Finite Difference Techniques for the Connection Machine", AIAA 91-0436

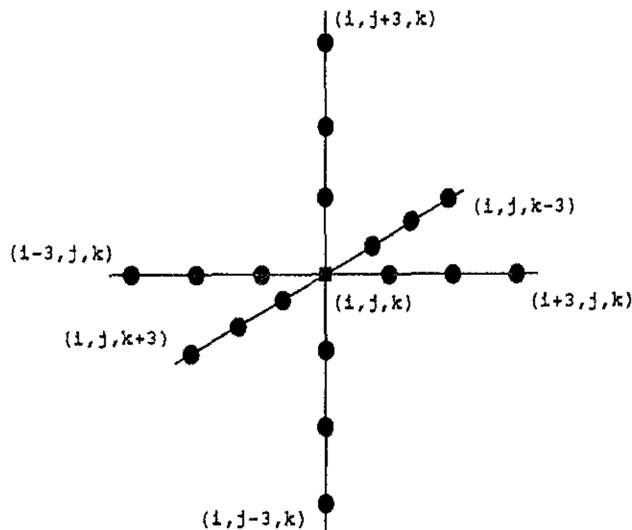

Figure 1: 3-dimensional 6th order finite difference stencil

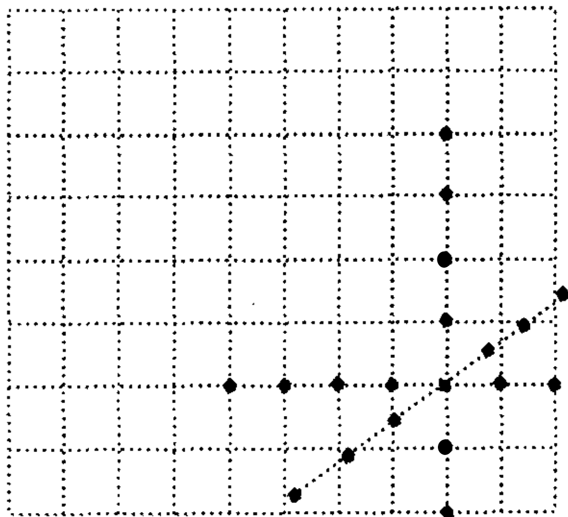

Figure 2: Typical 6th order stencil near a boundary

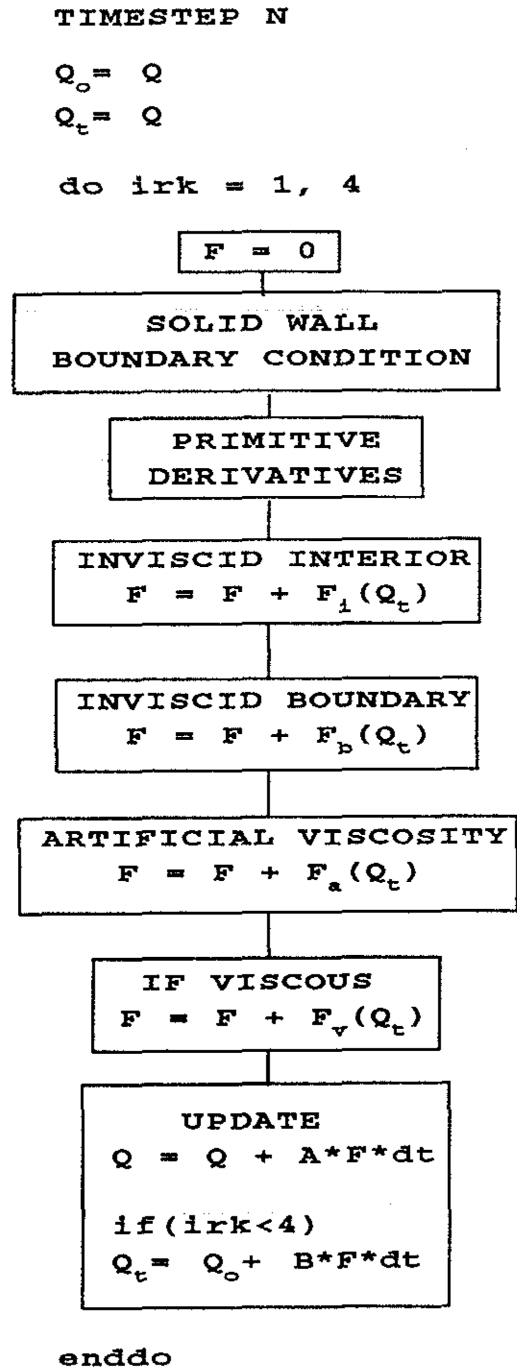

Figure 3: Algorithm flow chart

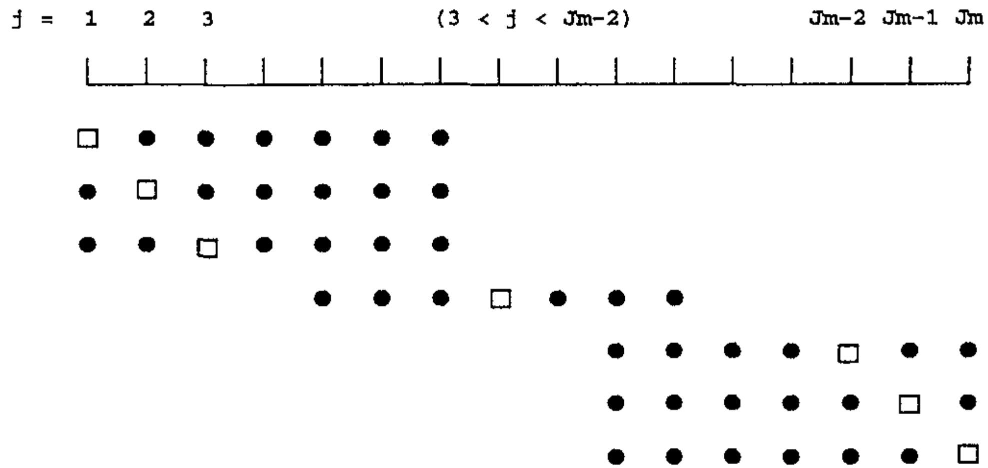

Figure 4: The 7 stencils of a 6th order 1-dimensional problem

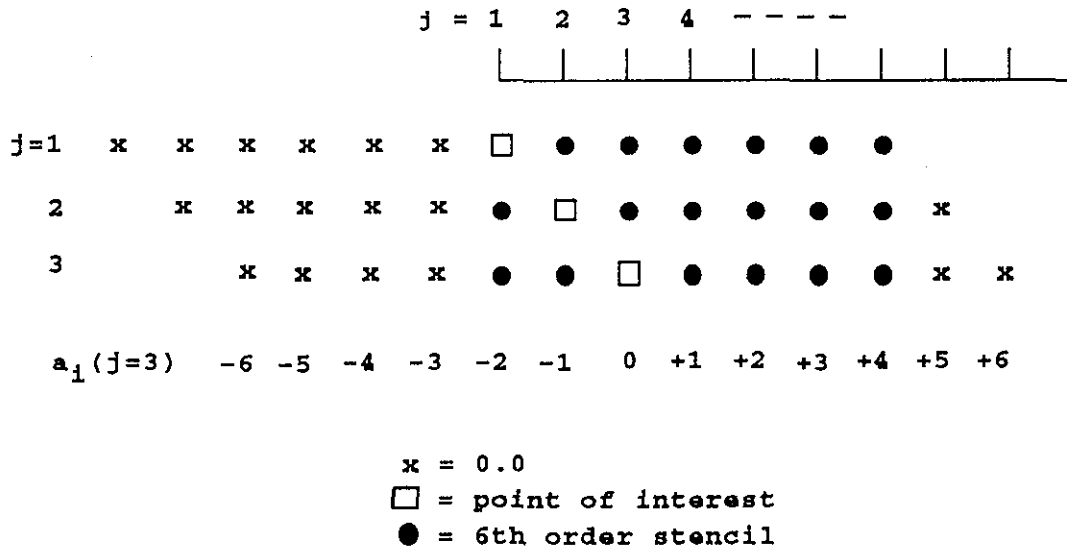

Figure 5: Global stencil arrays

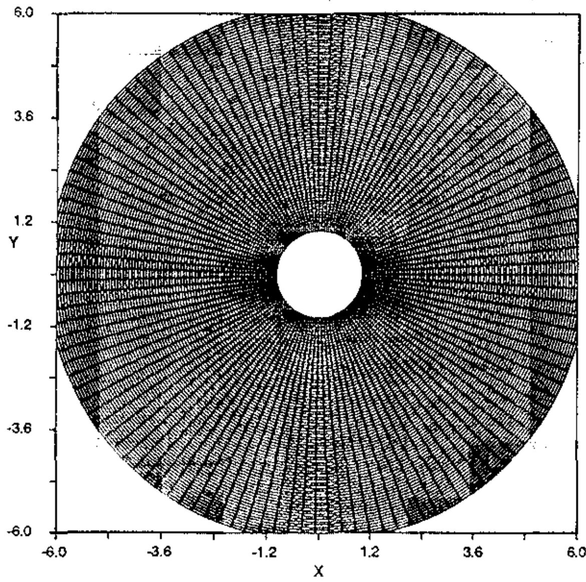

Figure 6: 120 by 75 point grid used for cylinder scattering problem

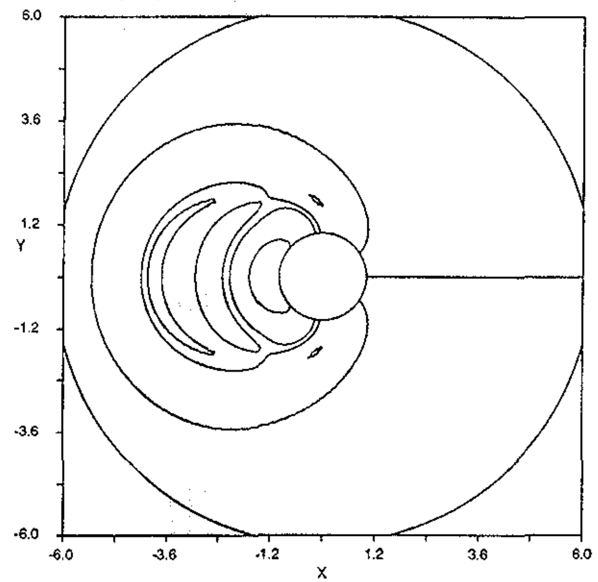

Figure 8: Pressure contours after 700 timesteps

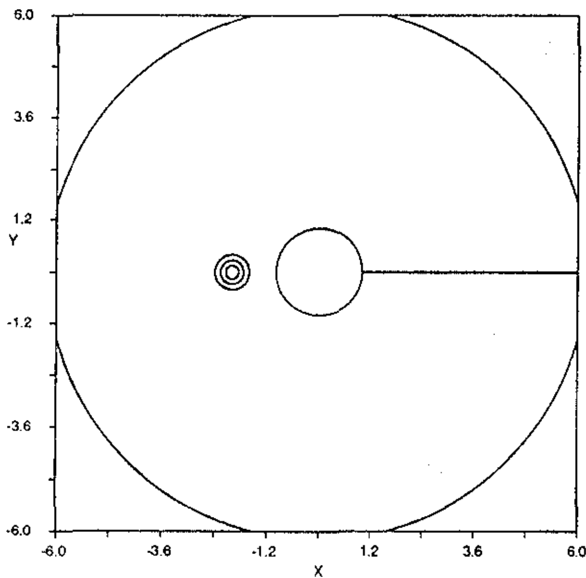

Figure 7: Pressure contours of the initial condition

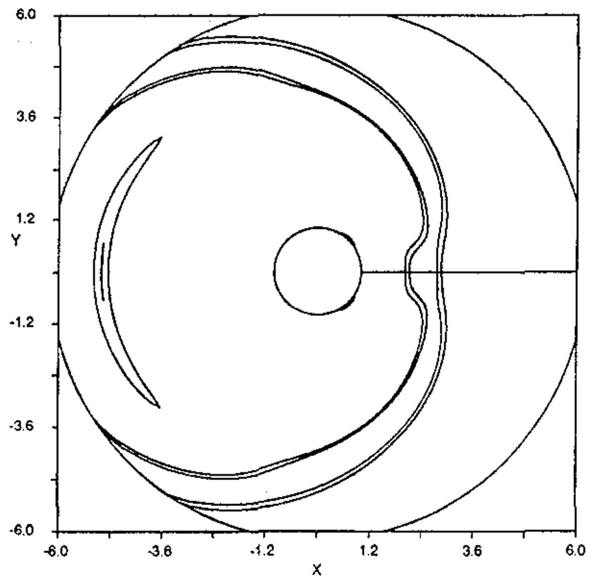

Figure 9: Pressure contours after 1400 timesteps

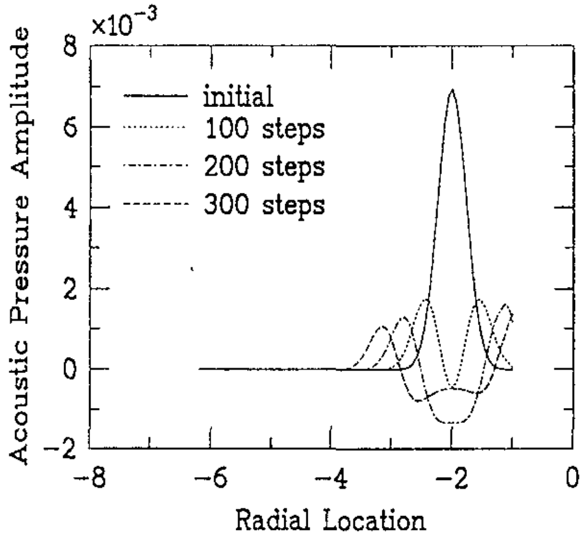

Figure 10: Solution history before reflected wave leaves the wall

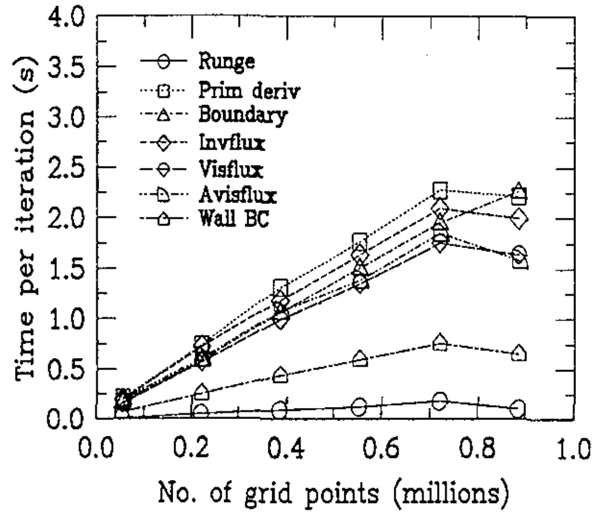

Figure 12: Analysis of the performance of each code segment for the 64 processor case.

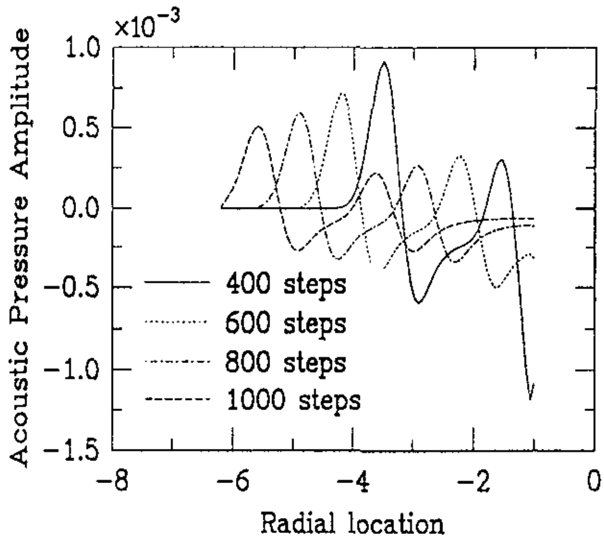

Figure 11: Solution history after reflected wave leaves the wall

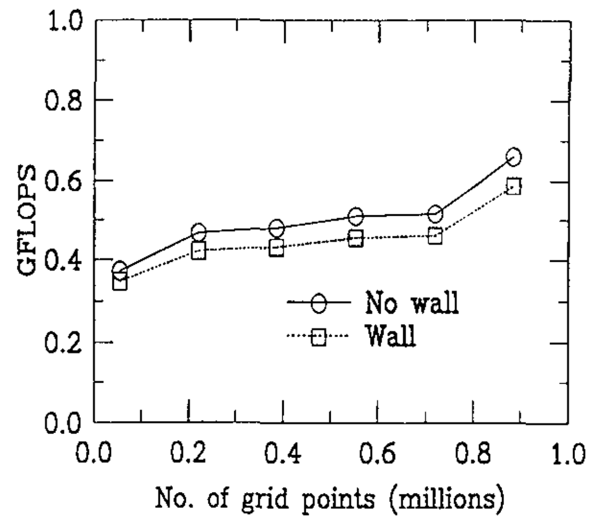

Figure 13: Effect of wall boundary condition on code performance

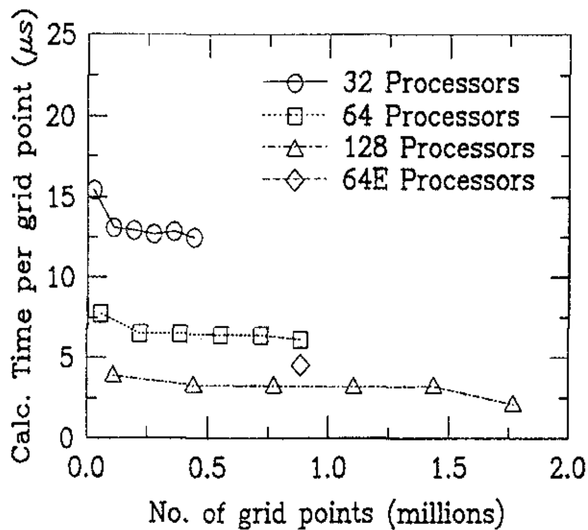

Figure 14: Calculation time per timestep per grid point

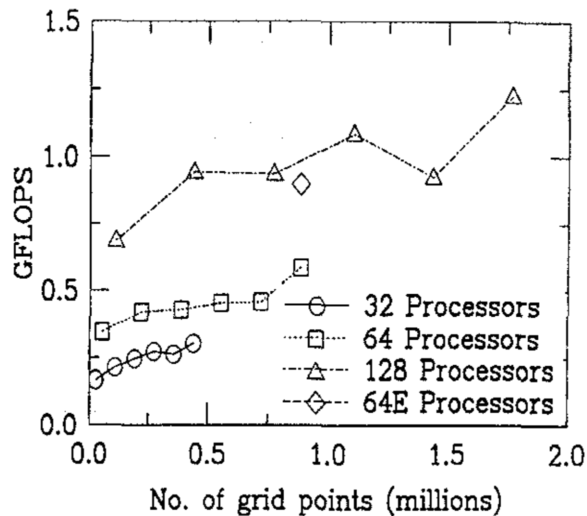

Figure 16: Overall algorithm performance

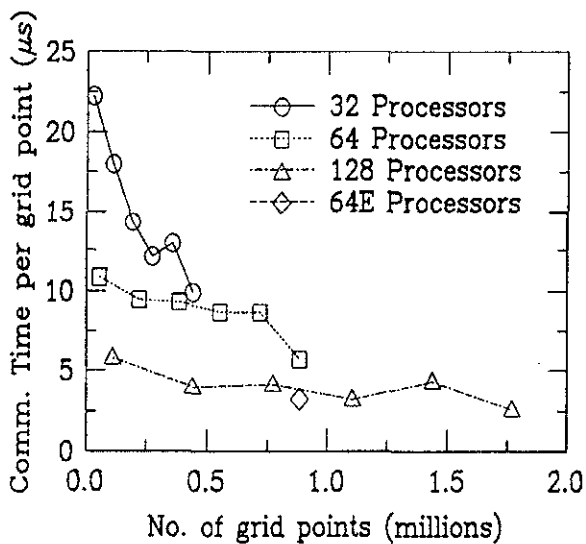

Figure 15: Communication time per timestep per grid point

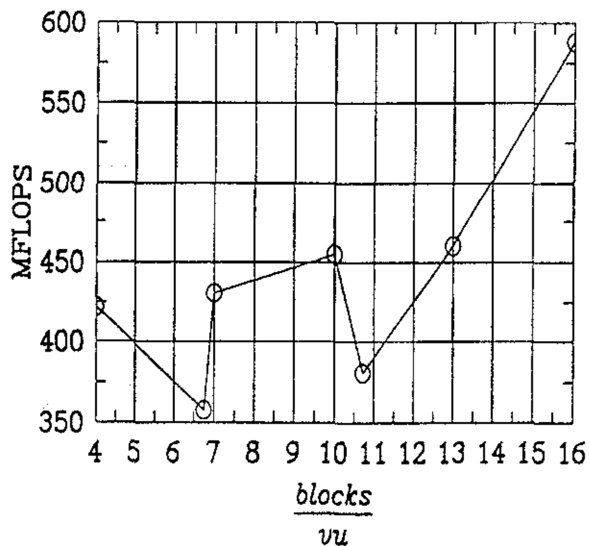

Figure 17: Effect of nonoptimal problem sizes