AIAA Journal
Vol. 32, No. 1, January 1994

Generic Parallel Adaptive-Grid Navier-Stokes Algorithm

Y. Kallinderis* and A. Vidwans†
University of Texas at Austin, Austin, Texas 78712

A parallel adaptive-grid Navier-Stokes algorithm based on generic primitives has been developed. The parallel primitives are general for the class of explicit finite-volume Navier-Stokes numerical schemes. Furthermore, they allowed relatively simple implementation of the algorithm on two different parallel systems; an eight-processor Cray Y-MP and the Connection Machine CM-2. A novel data structure for the adaptive grid allowed efficient parallel refinement/coarsening of the mesh. Substantial speeds compared to the corresponding sequential algorithm were realized on both systems.

I. Introduction

COMPUTATIONAL fluid dynamics (CFD) has advanced rapidly over the last two decades, and it is recognized as a valuable tool for engineering design. However, numerical simulation of viscous flowfields remains very expensive even with use of current vector computers. Advances in numerical algorithms are not expected to reduce the cost of those computations to the extent that they can routinely be applied for design.

Vector computers accelerated computations by one or two orders of magnitude compared to scalar machines, which is not sufficient for efficient large-scale flow simulations. Another approach to computer architectures has been employment of a number of processors that work in parallel executing the same job. Parallel computing appears to be a promising approach for future design applications of CFD.

Development of CFD applications on state-of-the-art parallel machines currently requires considerable effort on the part of the user to understand the intricacies of the underlying architecture and fine tune the application to match them. Most of the effort has to be duplicated when the same application has to be ported to another machine. All of this inefficiency can be eliminated to a large extent by allowing the user to design the application in a machine-independent fashion using general primitives.

Several parallel algorithms have been developed in the past. The architectures that have been employed include the Cray Y-MP/8,[1,2] as well as the Connection Machine CM-2.[3-5] Many of the applications to date have dealt with structured meshes, in which data is stored in a regular manner. Unstructured grid solvers have also been developed.[4,6,7] A large number of the parallel CFD codes have been developed for a specific architecture, and portability of the algorithms has not been considered.

Adaptive algorithms have become quite popular in CFD. They provide flexibility to adjust the grid during the solution procedure without intervention by the user.[8-10] Those adaptive grid algorithms have been developed for sequential execution and have reached a level of maturity. However, the area of parallel adaptive algorithms is relatively unexplored.[1]

The present work develops a parallel adaptive algorithm that employs generic primitives. Those parallel primitives are general for the class of explicit finite-volume Navier-Stokes numerical schemes. Furthermore, they allow relatively simple implementation of the algorithm on two different parallel architectures. Parallel adaptive grid refinement/coarsening is attained via a special data structure that was developed in the present work.

The adaptive algorithm applies local grid refinement/coarsening during the solution process to resolve local flow features efficiently. Creation as well as updating of the data structure after each grid adaptation is done in parallel. The finite-volume algorithm uses explicit time marching and central differencing type of spatial discretization. Second- and fourth-order artificial dissipation is added for stability and for capturing of shock waves. Parallel execution on an eight-processor Cray Y-MP, as well as on the Connection Machine CM-2 yielded substantial speedups compared to the corresponding sequential algorithm.

In the following sections, the adaptive-grid Navier-Stokes method and the data structure are described first. Then, the generic parallel primitives are presented. Finally, results obtained with parallel execution on the Cray Y-MP/8, as well as on the CM-2 are presented and discussed.

II. Adaptive Finite-Volume Algorithm

The Navier-Stokes system may be written in Cartesian two-dimensional conservation form as:

$$\frac{\partial U}{\partial t} + \frac{\partial F}{\partial x} + \frac{\partial G}{\partial y} = \frac{\partial R}{\partial x} + \frac{\partial S}{\partial y} \tag{1}$$

It consists of four equations expressing conservation of mass, momentum in the x and y directions, and energy of the fluid. In Eq. (1) U is the state vector comprising the variables of density, the two components of momentum and the total energy. The vectors F and G represent the convective terms. Finally, R and S are the vectors corresponding to the diffusion terms.

The two-dimensional Navier-Stokes equations are integrated over an area S

$$\iint_S \frac{\partial U}{\partial t}\,\mathrm{d}S + \iint_S \left(\frac{\partial F}{\partial x} + \frac{\partial G}{\partial y}\right)\mathrm{d}S = \iint_S \left(\frac{\partial R}{\partial x} + \frac{\partial S}{\partial y}\right) \tag{2}$$

The surface integrals are evaluated by using Green's theorem.

$$\frac{\partial}{\partial t}\iint_S U\,\mathrm{d}S + \oint_{\partial S} (F\,\mathrm{d}y - G\,\mathrm{d}x) = \oint_{\partial S} (R\,\mathrm{d}y - S\,\mathrm{d}x) \tag{3}$$

Received Sept. 23, 1993; revision received March 29, 1993; accepted for publication March 29, 1993. Copyright © 1993 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved.

*Assistant Professor, Department of Aerospace Engineering and Engineering Mechanics. Member AIAA.

†Graduate Research Assistant, Department of Electrical and Computer Engineering.

The flow domain consists of smaller areas (cells) that are defined through the coordinates of their vertices. Equation (3) is considered for each one of the cells

$$\frac{\partial}{\partial t}\iint_{\text{cell area}} U\,\mathrm{d}S + \oint_{\text{cell edges}} (F\,\mathrm{d}y - G\,\mathrm{d}x) = \oint_{\text{cell edges}} (R\,\mathrm{d}y - S\,\mathrm{d}x) \tag{4}$$

The first term expresses the total change with time of the state variables, whereas the second term is the net fluxes of the same quantities through the edges of the cell. The term on the right-hand side represents viscous fluxes through the edges. The net flux through the cell-edges is termed the residual. Details of the specific finite-volume scheme considered in the present work are given in Refs. 8 and 11. The standard second- and fourth-order smoothing operators are employed for stability and for capturing of shock waves.

A. Adaptive Local Grid Refinement/Coarsening

Grid adaptation consists of adjusting the grid spacing so that the numerical error is relatively small and equally distributed throughout the solution domain. This is accomplished by increasing grid resolution locally in regions in which flow features exist. Initial coarse grid cells are divided by inserting additional points in between the initial points, thus creating a local embedded grid. Directional division of a cell along one of its directions is also employed in cases of flow features that have the same direction. Several levels of such finer grids are allowable, and they can be limited to those regions of the domain in which important features exist. Conversely, excessive resolution is removed by deleting grid points locally over regions in which the solution does not vary appreciably. It should be noted that the control volumes for the numerical integration are the locally finest cells.

The sequence of steps for the adaptive algorithm is as follows:

1) Solve the Navier-Stokes equations on an initial coarse grid.

2) Monitor the residual error until it falls below a prespecified value; detect the main flow features, such as shear layers and shock waves; refine the grid locally if the detection parameter exceeds the threshold for division, or remove grids if it is less than the threshold for deletion.

3) Continue computing on the updated adaptive grid.

4) Repeat steps 2 and 3 until steady state has been reached, or up to a specified time level for unsteady simulations.

Details of the adaptive algorithm considered in the present work can be found in Refs. 8, 9, and 12.

B. Data Structure for the Adaptive Grid

The computational grid consists of two principal entities: the individual grid points and the composite cells consisting of these points. The data structure for the grid must allow efficient implementation of the following operations comprising the adaptive algorithm: 1) flow of data from the grid points (nodes) associated with a cell to that cell, 2) flow of data from a cell to its constituent nodes, and 3) modification of the grid under local adaptation. The first operation corresponds to evaluation of the residual at the cell centers. The second operation corresponds to distribution of the residual to the cell corners. The third operation divides a cell into "children." Successive divisions create a tree structure for every cell of the original unadapted grid. Some of these operations may involve the deletion of a group of children cells in order to recover the parent cell.

The original unrefined grid is represented as a list of cells, a list of grid points, and a mapping, which defines an association between an individual cell and its corner nodes. This is implemented as an array of cells, an array of nodes, and four pointer arrays each containing one of the cells' four corner nodes. Consider the cells and nodes lists

$$C = \{1, 2, \ldots, n\text{cels}\}, \qquad N = \{1, 2, \ldots, n\text{nodes}\}$$

The mapping of cells to their respective first corner node is

$$M_1 = [M_1(1), \ldots, M_1(n\text{cels})]$$

where $M_1(i)$ is the first corner of cell $i \in N$. Similar mappings exist for the remaining corners of the cells. When a cell gets divided, additions have to be made to the cell array as well as to the node array to accommodate the additional cells and nodes. Furthermore, all of those children cells should be accessible efficiently given the parent cell. Additional data structures for this are a parent pointer array and a level array. The former indicates the parent of every cell, whereas the latter shows the level of embedding of every cell. The list of parent cells is

$$P = \{P(1), \ldots, P(n\text{cels})\}$$

where $P(i)$ is the parent of cell i and $P(i) \in \{0\}UC\,\forall\, i \in C$. The cells of the original grid that remain undivided are not considered as parent cells. The level of embedding of each cell is indicated by the list:

$$L = \{L(1), \ldots, L(n\text{cels})\}$$

where $L(i)$ is the level of embedding of cell i and $L(i) \in \{0, 1, \ldots, n\text{lev}\}$, with $n$lev being the highest level of embedding within the grid.

An example of a two-level embedded grid is shown in Fig. 1. Cells 1–4 form the original, unadapted grid. Cell 3 is divided into four children numbered 5–8 whereas cell 7 further gets divided into cells 9–12. The adaptation tree corresponding to this embedding grid example is illustrated in Fig. 2a. The initial grid is labeled as level 0, and the embedded portions are labeled according to the number of embeddings.

Correct functioning of the algorithm without regard to efficiency does not impose any particular restriction on the way in which the cells and nodes are inserted into the lists. However,

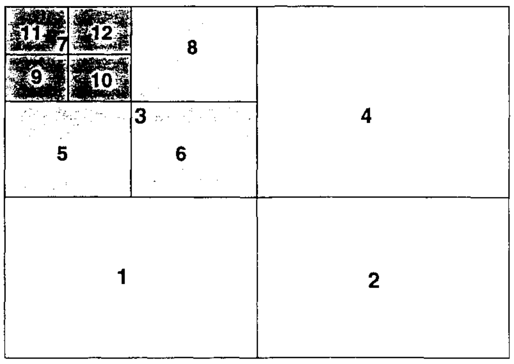

Fig. 1 Example of a two-level embedded grid.

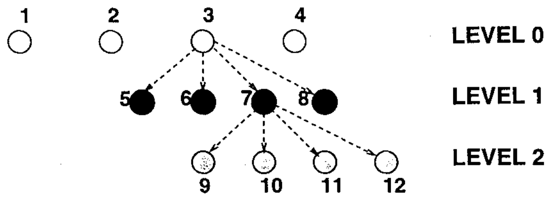

Fig. 2a Adaptation tree of cells corresponding to the two-level embedded grid.

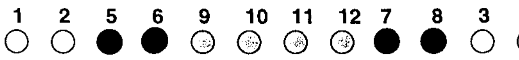

Fig. 2b Corresponding postorder listing of cells.

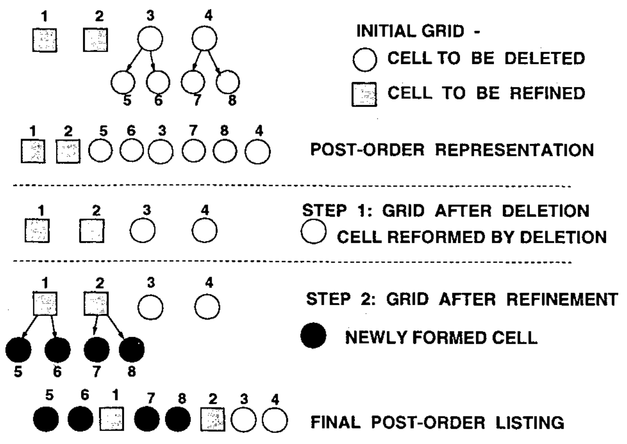

Fig. 3 Example of ordering of cells during the adaptation process.

parallel cell division and deletion warrant special attention to the manner in which cells are inserted and deleted from the adaptation tree.

C. Postorder Representation for Parallel Adaptation

In the present work, postorder traversal of a tree is defined as a traversal starting from the lowest level 0 such that the children of a given element are listed before that element itself is listed. The postorder representation corresponding to the aforementioned two-level embedded grid is shown in Fig. 2b. Cells 5–8, which are children of cell 3, are listed before their parent. Similarly, cells 9–12 are listed just before their parent cell 7.

A fundamental property of this postorder listing is that the postorder listing for a grid modified by adaptation can be obtained by localized operations on the postorder listing for the original grid. This enables parallelization of the entire adaptation process. For example, a cell to be divided into four children only needs to insert these children to its immediate left in the cell array. Similarly, every cell to be deleted requires access to only a few locations to its left in the array. As a consequence, all such cells can carry out the adaptation process at the same time.

An example of the adaptation process is illustrated in Fig. 3. The initial grid consists of four original cells, two of which (3 and 4) have been divided into two children each. These children are marked for deletion, whereas cells 1 and 2 are marked for division into two children each. Since both cells 3 and 4 have their children in contiguous locations immediately to their left in the postorder listing, the unrefinement process can take place in parallel for these two cells (step 1). In the division process, the children for cells 1 and 2 are introduced to their immediate left. Therefore, this operation also can be performed entirely in parallel (step 2).

IV. Generic Primitives for Parallel Algorithm

This section defines certain fundamental primitives of the parallel adaptive algorithm on which the Navier-Stokes implementation is based. Architecture specific details are restricted to these primitives so that the overall algorithm can be expressed in a completely machine-independent fashion thus making the algorithm portable across the two architectures.

A. Gather

A gather operation is defined as parallel read operation by a set of destination entities from a set of source entities. A gathering pattern specifies, for each destination entity, the corresponding source entity from which data is to be received. Consider $S(1 : n\text{sources})$ to be a source array, $\boldsymbol{D}(1 : n\text{dest})$ to be a destination array, and $\boldsymbol{M}(1 : n\text{dest})$ is a mapping array containing numbers from 1 to $n$sources, then $\boldsymbol{D}(\boldsymbol{M}) = \boldsymbol{S}$ "gathers" the array $\boldsymbol{S}$ into the array $\boldsymbol{D}$ such that for every i from 1 to $n$dest, $\boldsymbol{D}(i)$ receives $S[\boldsymbol{M}(i)]$.

In the parallel finite-volume algorithm, this primitive is used to gather data about the current solution from the corners of a cell to the cell itself to calculate the residual. This simply consists of an indirect addressing via the mesh connectivity array. An example of this operation in Fortran 90 is

```
F(1,1:ICELTOT) = U[2,ICELNOD(1:ICELTOT)]
```

where F is the array containing the first element of the flux vector $\boldsymbol{F}$, U is the array of the state variables, and ICELNOD is the connectivity array.

B. Elemental Operations

These operations are inherently parallel with no communication involved. Examples of such operations are cell-based operations which can be executed at each cell independently of other cells. Thus at a conceptual level, a processor can be assigned to a cell and full parallelism can be realized. This will be termed as the *cell-per-processor paradigm* of parallel computation.

The computation of the residual $\boldsymbol{R}$ by calculating the balance of the fluxes across the cell edges is a main elemental operation of the finite-volume scheme. The vector $\boldsymbol{R}_i$ corresponding to cell i is obtained by using the values of the solution $\boldsymbol{U}_n$ and grid coordinates X_n, Y_n, which are stored at the cell corners n, as follows:

$$\boldsymbol{R}_i = \sum_{e=1}^{4} [(\boldsymbol{F}-\boldsymbol{R})_e \Delta Y_e - (\boldsymbol{G}-\boldsymbol{S})_e \Delta X_e] \qquad (5)$$

where $(\boldsymbol{F}-\boldsymbol{R})_e = \frac{1}{2}[(\boldsymbol{F}-\boldsymbol{R})_{n1} + (\boldsymbol{F}-\boldsymbol{R})_{n2}]$ and $(\boldsymbol{G}-\boldsymbol{S})_e = \frac{1}{2}[(\boldsymbol{G}-\boldsymbol{S})_{n1} + (\boldsymbol{G}-\boldsymbol{S})_{n2}]$. Also, $\Delta X_e = X_{n2} - X_{n1}$, $\Delta Y_e = Y_{n2} - Y_{n1}$. The subscripts n_1 and n_2 denote values at the two end points of the cell edge e.

C. Scatter

A scatter operation is defined as parallel write operation from a set of source entities to a set of destination entities. A scattering pattern specifies, for each source entity, a corresponding destination entity to which data is to be sent. Consider $\boldsymbol{S}(1 : n\text{sources})$ to be a source array, $\boldsymbol{D}(1 : n\text{dest})$ to be a destination array, and $\boldsymbol{M}(1 : n\text{sources})$ is a mapping array containing numbers from 1 to $n$dest, then $\boldsymbol{D}(\boldsymbol{M}) = \boldsymbol{S}$ "scatters" the array $\boldsymbol{S}$ into the array $\boldsymbol{D}$ such that for every i from 1 to $n$sources, $\boldsymbol{D}[\boldsymbol{M}(i)]$ receives $\boldsymbol{S}(i)$.

In the context of the parallel finite-volume algorithm, data is scattered from a cell to its four corners. This is done through the mappings from the cells to their corners. The main scatter operation of the algorithm is distribution of the residual $\boldsymbol{R}_i$ to the cell corners n, in order to compute the change in time of the solution vector $(\delta \boldsymbol{U}_n)$. The distribution to one of the corners is given by the following formula[8,11]

$$(\delta \boldsymbol{U})_n = \tfrac{1}{4}\{\boldsymbol{R}_i + \Delta f + \Delta g\} \qquad (6)$$

where $\Delta f \equiv \Delta t/S(\Delta \boldsymbol{F} \Delta y^l - \Delta \boldsymbol{G} \Delta x^l)$, and $\Delta g \equiv \Delta t/S(\Delta \boldsymbol{G} \Delta x^m - \Delta \boldsymbol{F} \Delta y^m)$, with $\Delta \boldsymbol{F} \equiv (\partial \boldsymbol{F}/\partial \boldsymbol{U})\boldsymbol{R}$, $\Delta \boldsymbol{G} \equiv (\partial \boldsymbol{G}/\partial \boldsymbol{U})\boldsymbol{R}$. The cell-metric terms Δx^l, Δy^l, Δx^m, and Δy^m denote the cell dimensions along the l and m cell coordinates.

D. Global Operations

A global operation is a reduction operation that converts a vector of data values into a single scalar quantity. Examples of this kind of operation are evaluation of the maximum and rms residual errors over the entire domain for convergence test, as well as calculation of the average and maximum flow gradients for detection of the flow features that require local grid adaptation.

A typical algorithm for implementing such an operation is to build a binary tree with the vector of values as the leaves and perform the operation on pairs of values while traversing the tree from the leaves to the root.[13] Figure 4 illustrates the parallel binary tree algorithm. The vector 1, 2, 3, 4 forms the input to be summed up. The sum is obtained at the root in log(4) = 2 steps. Intermediate levels hold partial sums of elements of the vector.

E. Prefix

The prefix operation, defined for associative binary operators,[14] computes a destination array $D(1 : n\text{size})$ from a source array $S(1 : n\text{size})$ such that for all i from 1 to $n$size,

$$D(i) = \sum_{j=1}^{i} S(j) \tag{7}$$

This operation is used to rearrange the listing of cells during the first phase of the parallel adaptive algorithm where space is created within the tree for new cells. Specifically, every cell to be divided is marked with the number of cells it gets divided into and all other cells are marked 0. Then a prefix is performed on this array and the result indicates the number of locations through which a particular cell has to move to get into correct position in the rearranged array. Thus, if Flag(1 : $n$cells) indicates the number of children that each cell is to be divided, then

$$\text{Incr}(i) = \sum_{j=1}^{i} \text{Flag}(j) \tag{8}$$

indicates the number of locations that a particular cell has to be shifted in order to properly accommodate all of the newly created cells.

The prefix operation is implemented in parallel by constructing a cluster of binary trees with the results of the prefix operation appearing at the root of each of the trees in the forest. An example of the parallel prefix operation is illustrated in Fig. 5. The four inputs to the prefix operation (3, 8, 12, 34) form the lowest level of the binary tree. The operation takes log(n) steps with $n = 4$ in this case. In step i, every data element is combined with the data element which is 2**i places to its left if such an element exists. Thus in step 0, input 3 is added to 8, 8 is added to 12, and 12 is added to 34 to get the partial sum vector (3, 11, 20, 46). In step 1, 3 is now added to 20, and 11 is added to 46 to get the final result as (3, 11, 23, 57).

F. Monotone Rout

This primitive is used immediately after the prefix operation in the adaptation process to physically rearrange the cell array so as to create extra spaces for new cells and delete spaces corresponding to the cells which are marked for unrefinement. The monotone rout or permutation is a special kind of scatter operation where the relative order of elements to be reordered is not changed by the routing process.[15] In the context of the parallel adaptive algorithm, the gaps resulting from the monotone rout are filled by new cells formed in the division process. During the deletion step, the monotone rout acts as a parallel compaction operation wherein the gaps corresponding to deleted cells are filled up by cells to their right in the listing.

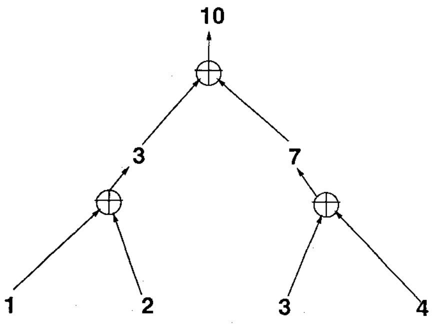

Fig. 4 Example of a global operation: calculation of the sum of a given set of numbers using the binary tree parallel algorithm.

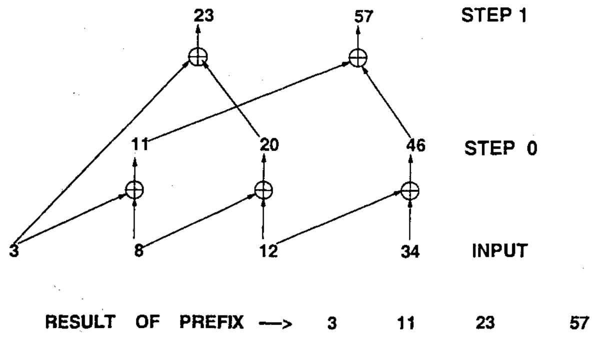

Fig. 5 Example of the parallel prefix operation: calculation of the prefix values for an input set of four numbers.

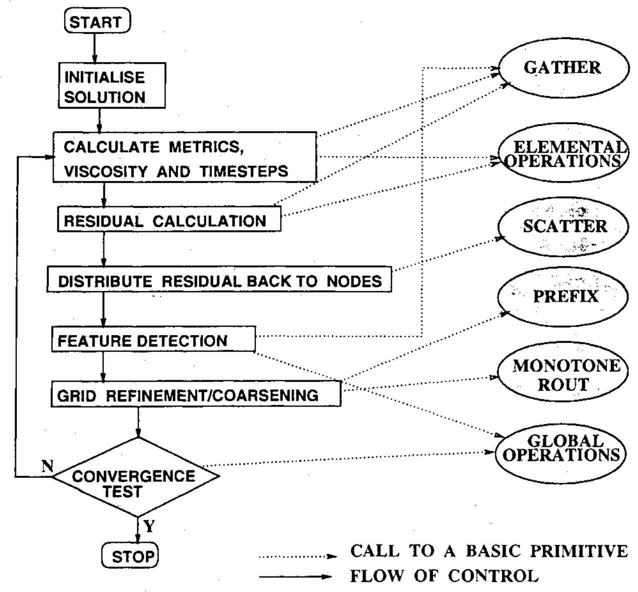

Fig. 6 Flowchart for the overall algorithm with calls to generic primitives.

The primitives are employed by the parallel adaptive algorithm in a manner illustrated in the flowchart of Fig. 6.

V. Implementation on a Shared Memory Multiple Instruction Multiple Data Architecture

A. Architecture Details

The Cray Y-MP is a shared memory multiple instruction multiple data (MIMD) architecture consisting of eight high-performance vector processors sharing a common, global memory. All data is held within the shared memory effectively eliminating any kind of communication among the processors extraneous to the main storage. However, the CPU-memory transfer of data is pipelined and has a bandwidth limitation. Although every processor is equipped with vector registers, the user has some responsibility of ensuring that this bandwidth does not become a bottleneck. The following additional features are important to obtain good performance:

1. Stripmining

Parallelization on the Cray Y-MP is achieved through splitting of DO-loops containing no data dependencies among the eight processors of the architecture. Each processor receives vector(s) of smaller length which are then concurrently processed in standard vectorized mode depending on the size of the original vector to be processed. This distribution of workload is done dynamically at runtime.

This method of parallelization provides the user with a programming interface in which it is relatively easy to incorpo-

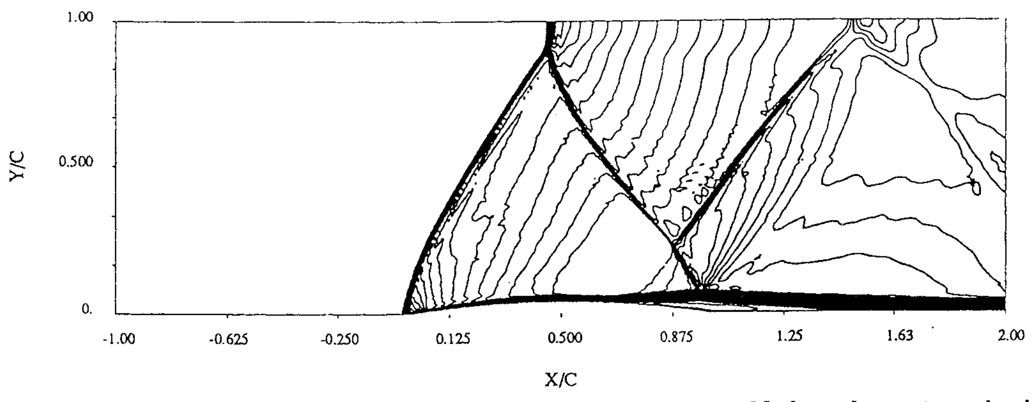

Fig. 7 Parallel simulation on the Cray Y-MP/8 of viscous flow through channel with a bump; Mach number contours showing interaction of shock wave with boundary layer.

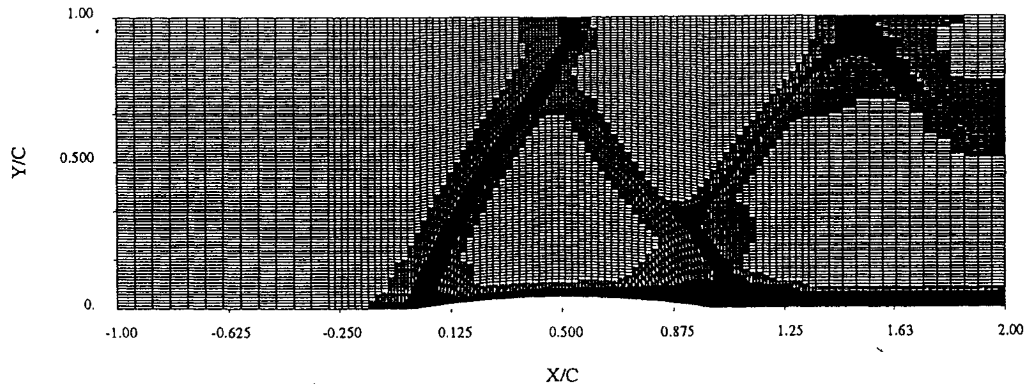

Fig. 8 Parallel creation on the Cray Y-MP/8 of two-level adapted grid for viscous flow through channel with a bump; local refinement of the shock and the boundary-layer regions.

rate the cell-per-processor paradigm of the parallel adaptive algorithm. Any DO-loop which has no data dependencies appears to have been processed completely in parallel because the inner details of stripmining are totally hidden. Furthermore, since the distribution of work is performed at run time, the load gets dynamically balanced between the available processors without any special efforts from the user. This also implies that introduction of additional processors causes a near-proportional reduction in execution time. There is some distortion due to system and other overheads.

2. Indirect Vectorization

A feature of the Y-MP architecture that is important in the context of unstructured grid algorithms is its ability to vectorize indirect pointer references instead of the actual vector. This ensures that the DO loop whose index is used to dereference pointers can also be executed in vector-concurrent mode.

In view of these features, the strategy for parallelization on the Cray Y-MP is to ensure that all of the single DO loops and inner loops in nested DO loops are executed in vector-concurrent mode.

3. Prefix Optimization

The prefix operation on an input size N is implemented using a nested loop structure with the outer loop going over the log(N) steps and the two inner loops performing the partial addition steps. The inner loops are fully parallelizable, whereas the outer loop is not. Therefore, it is clear that minimizing the number of prefix operations performed enhances performance. Several prefix operations with the same input size were combined into the same nested loop structure. This results in the outer loop being executed just once for more than one prefix operation. The inner loops are made bigger and hence the loop startup overhead is distributed across a larger workload. A typical DO loop for combined prefix is

```
DO ISTEP = 0 , ILOGN
        IJUMP = 2**ISTEP
    DO NOD = IJUMP + 1 , NODTOT
        C1(NOD) = B1(NOD) + B1(NOD - IJUMP)
        C2(NOD) = B2(NOD) + B2(NOD - IJUMP)
    ENDDO
    DO NOD = 1 , NODTOT
        B1(NOD) = C1(NOD)
        B2(NOD) = C2(NOD)
    ENDDO
ENDDO
```

The preceding piece of code combines two prefix operations having the same input length into a single DO-loop structure. ILOGN is the logarithm of the input size N to the base 2 and denotes the number of steps in the prefix operation. IJUMP denotes the distance between two elements being combined at each step. B1 and B2 are the two input arrays, which also hold the successive partial sums during the operation. C1 and C2 are the corresponding temporary arrays. The first inner loop performs the actual addition at each step and the second loop copies back the partial results into the original arrays B1 and B2. This technique can be used to combine several such prefix operations.

B. Code Performance

The case of parallel simulation of supersonic viscous flow ($M_\infty = 1.4$) through a channel with a bump is considered. Figure 7 illustrates the flowfield in terms of Mach number contours. The computed field includes shock waves and expansion fans, as well as a shear layer formed at the region of the lower boundary. Two levels of embedding were employed by the parallel adaptive algorithm. Figure 8 shows the two-

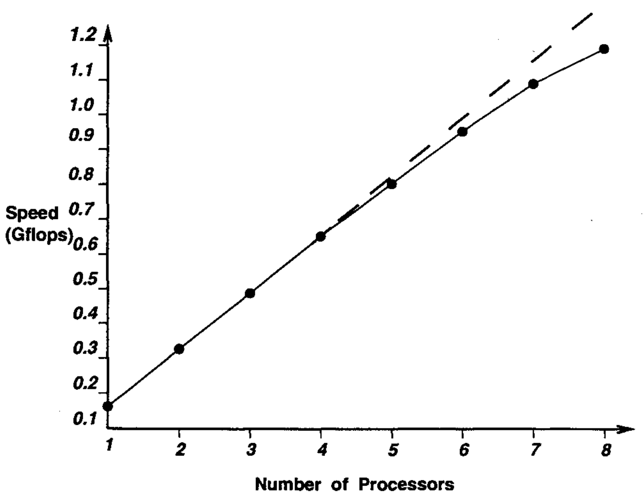

Fig. 9 Parallel execution speed of Navier-Stokes finite-volume algorithm vs number of processors on the Cray Y-MP/8, 60 K grid points; dashed line corresponds to linear increase in speed.

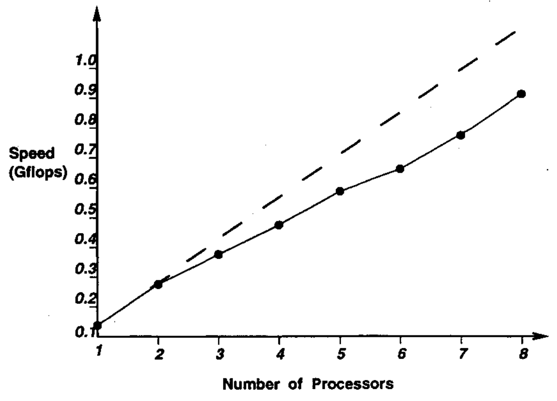

Fig. 10 Parallel execution speed of combined solver/grid adapter vs number of processors on the Cray Y-MP/8, 60 K point initial grid being adapted at every timestep; dashed line shows corresponding linear increase in speed.

level adapted grid. It is observed that additional resolution is automatically placed by the adaptive algorithm within the local regions of the flow features, which are primarily shock waves and boundary layer.

The stand-alone solver (without the adapter) was executed with a grid consisting of 60 K points. It was found to give a maximum speed of approximately 160 Mflops on a single CPU of an eight-processor Cray and a speed of about 1.2 Gflops on all of the eight processors. Whereas the speed in the former case is entirely due to the vectorization capabilities of a single CPU, the speedup obtained over that in the latter case is due to the parallelism offered by multiple such CPUs. It should be noted that further increase in the grid size does not result in significant increase in performance, since both the number of processors, and length of the vector pipes are fixed. Figure 9 illustrates the attained speed as a function of the number of processors that were utilized in parallel. A speed of approximately 1.165 Gflops was obtained.

The solver coupled with the adapter was executed with an initial grid consisting of 60 K points and a group of 4000 cells were refined and unrefined at each time step to measure performance of the algorithm. The measured speed was 140 Mflops with one processor, whereas the speed with all eight processors approached 0.95 Gflops. The total execution time was 1.86 s per time step. Parallel execution speeds are shown in Fig. 10 as a function of number of processors from one to eight. It should be noted that the increase in speed is not in exact proportion to the number of processors employed due to the additional overhead of partitioning the work among the processors. Furthermore, overheads extraneous to the execution of the algorithm, such as operating system interrupts and network overheads, caused fluctuations in the speeds obtained with different numbers of processors.

VI. Implementation on a Single Instruction Multiple Data Architecture

A. Architecture Details

The CM-2 has a single instruction multiple data (SIMD) architecture with 64 K processors. There is no globally shared memory, and every processor has its own local memory. Under the "slicewise" model of parallel computation, these 64 K processors are organized into 2 K groups of 32 processors. These groups are called processing elements (PEs). The programming interface is a dialect of Fortran-90 with additional constructs for CM-specific operations. In every array operation, the individual array elements are allocated 1 per PE and wrapped around if necessary. Every PE executes the same instructions on its set of data inputs. This paradigm exactly matches the one cell-per-processor model with every PE simulating as many virtual PEs as its data inputs. Additional features important for obtaining good performance are given in the following.

1. Load Balancing

The allocation of elements to PEs is done at runtime since the compiler is not aware of the number of PEs that will be allocated to the program for execution. This entails a runtime overhead of system calls to do the job but has an advantageous effect of dynamically balancing the workload among the available PEs in a "greedy" fashion, i.e., balancing of load as soon as an imbalance is created. However, this dynamic allocation is costly in terms of the system overhead. Consequently, a large input size results in more effective amortization of this overhead over the execution time. The amount of memory available on each one of the PEs allowed larger grid sizes compared to the Cray that was used.

2. General Routing

The gather and scatter operations and the associated communication costs prove to be the most important factors affecting performance on the CM-2. These operations are done at the machine level by a router or a hardware component that routs data from a set of sources to a set of destinations based on a gathering/scattering pattern. This router can handle any kind of reordering pattern and is consequently costly in terms of execution time. Therefore, it is essential to minimize the number of such routs used in the implementation. This was achieved through combining the implementation of the convective and diffusive terms in the finite-volume scheme into one large elemental block containing no communication operations with one gather operation preceding and one scatter operation following this block.

Further reduction in communication costs can be obtained by using the fact that the routing pattern, i.e., the paths of all of the elements being moved, is the same for every gather/scatter operation as long as the grid does not change. The CMSSL library routines for gather and scatter take advantage of this fact and can reduce the execution time by almost a factor of two. However, these routines are not effective in case of rapidly varying grids, since the time expended in computing the routing information is relatively large.

3. Additional Communication

Some of the constructs of the programming interface have additional communication costs. Array sections are constructs to extract a subset of the elements in an array for selective processing. This operation has the overhead of redistributing the section over the available PEs and can significantly affect performance. Therefore, minimum use of these constructs is necessary. A special case of an array section is a construct

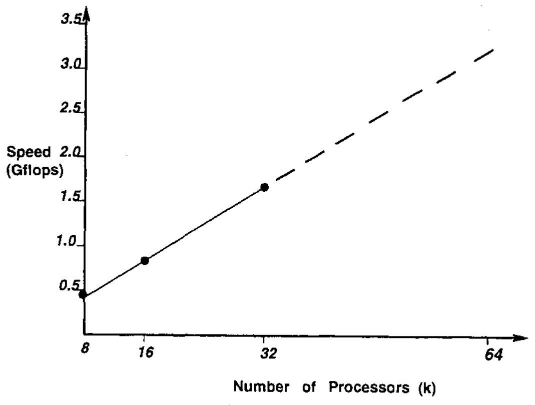

Fig. 11 Parallel execution speed of Navier-Stokes finite-volume algorithm vs number of processors on the Connection Machine CM-2, 100 K grid points; dashed line corresponds to linear increase in speed.

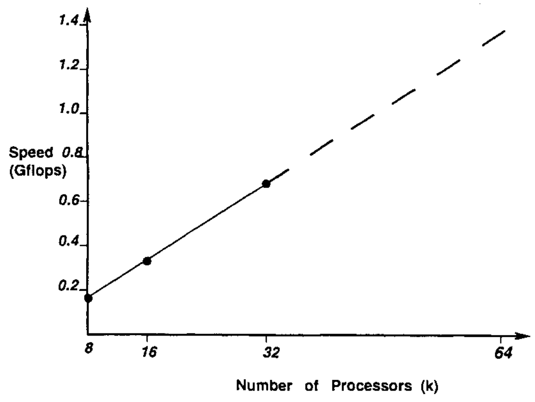

Fig. 12 Parallel execution speed of combined solver/grid adapter vs number of processors on the Connection Machine CM-2, 100 K point initial grid being adapted at every time step (dashed line corresponds to linear increase in speed).

which accesses a subset of the dimensions of a multidimensional array. This, too, has a significant cost and has to be minimized. Fixed-size dimensions from multidimensional arrays were eliminated, and the DO loops that looped over them were unrolled.

B. Code Performance

The parallel adaptive finite-volume algorithm was programmed in CM-Fortran, and fully implemented on 32 K processors of a CM-2. The solver only and the combined solver/adapter were executed in parallel.

The Mflops speeds for the parallel adaptive algorithm on the CM-2 are calculated using a special procedure that takes into account the number of CM processors utilized, the size of the input grid, and the actual CM busy time during the execution. The CM compiler generates statistics that indicate the number of floating point instructions and the number of clock cycles associated with every block of CM Fortran code. Substitution of the cycle and flop counts into the following formula yields the peak Mflops rate for that section of the code:

$$\text{Mflops} = \frac{4 * (\text{no. of processor}) * \text{flop count}}{32 * \text{cycle count} * \text{cycle time}} \tag{9}$$

where cycle time is 150 ns. This formula assumes an assignment of one array to every PE, i.e., the cycle count indicates the number of cycles required for a PE to process one array element. The method used here to calculate the Mflops values

Table 1 Execution timings of different primitives on the CM-2, 100 K point initial grid being adapted at every time step

	Processors		
Operation	8 K, s	16 K, s	32 K, s
Gather	0.1100	0.5570	0.0256
Scatter	0.1600	0.083	0.0418
Elemental operations: convective	0.1448	0.0729	0.0352
Elemental operations: diffusive	0.2910	0.1460	0.0739
Artificial damping	0.9510	0.4850	0.2370
Adapter: cell division	5.180	2.610	1.310
Adapter: cell deletion	2.570	1.290	0.650
Total	9.4068	4.795	2.351

for the actual execution of the algorithm involves a modified version of formula (9)

$$\text{Mflops} = \frac{\text{loop count} * 4 * (\text{no. of processor}) * \text{flop count}}{32 * (\text{actual execution time})} \tag{10}$$

where loop count is the number of array elements allocated to a PE during execution and is given by: loop count = (input size)/(number of PEs). The actual execution time is obtained by selectively timing the section of code involved using the CM timer routines. The modified formula replaces the CM cycle time with the actual execution time of the algorithm on a given number of PEs. The loop count serves as a correction factor that takes into account the input grid size and the actual number of PEs employed during execution. This formula is more appropriate, since it takes into account the adaptation process for calculation of the speeds. The execution time of the adaptive part of the algorithm is added to the total execution time in the denominator for calculating the speed of the solver-adapter combination.

The performance of the stand-alone solver without adaptation was evaluated first. A computational grid consisting of 100 K grid points was employed. Figure 11 shows the scalability of speedup obtained. It is observed that speed increased in almost exactly the same proportion as the number of processors used. This illustrates one of the advantages of the fine-grained SIMD parallelism which is not there in a shared memory MIMD architecture due to restrictions of hardware technology. Although it is infeasible to have more than 16 PEs operating under a shared memory environment, the CM-2 system can be relatively easily scaled up to higher numbers of processing elements, making it ideal for applications manifesting a very fine granularity of parallelism. Thus, whereas a performance of close to 1.8 Gflops was obtained when all of the 32 K processors were employed, this speed can be expected to scale almost linearly up to a maximum of 3.5 Gflops for a CM-2 consisting of 64 K processors.

To measure the performance of the integrated solver-adapter, a group of 4000 grid cells was repeatedly divided and deleted. This form of adaptation was carried out at every time step to simulate an unsteady calculation and to assess performance of the algorithm under the most demanding conditions of rapidly varying grid. Figure 12 shows the execution speed of the combined solver and grid adapter. It is observed that adaptation causes an appreciable reduction in speed on the CM-2 as compared to the Cray. The principal reason for this is the fact that the adaptive algorithm has extra overheads associated with division/deletion in the form of communication from the cells to nodes and vice versa. This communication arises primarily when the state vector at new grid points is to be obtained by interpolation between the two neighboring grid points. Every cell that gets divided has to gather the two interpolating state vectors and scatter the interpolated vectors to the newly created grid point. Thus, the cell division process is notably more expensive than the deletion process. This is not an important factor on the Cray since there is no communication among the processors at all. However, it causes a degradation in performance on the CM-2 because such communication has to be handled using generic routing. Furthermore, the CMSSL software is not effective in this case of rapidly varying grid.

Table 1 shows the execution timings per time step of the various parallel primitives on the Connection Machine for the same test case. It is seen that the timings scale down in almost exact proportion to the number of PEs employed to execute the program, illustrating the scalability of the algorithm on the SIMD architecture. The routine for artificial damping includes additional gather and scatter operations for the fourth-order smoothing, which cause an increase in execution time.

VII. Conclusions

The developed parallel adaptive grid Navier-Stokes algorithm yielded substantially higher execution speeds compared to the corresponding sequential algorithm. Furthermore, portability was demonstrated by implementing the algorithm on two different systems: the Cray Y-MP/8 and the Connection Machine CM-2.

The developed generic primitives corresponded to each one of the major operations of a typical adaptive-grid finite-volume algorithm. Those primitives allowed relatively simple implementation on two different parallel architectures. A novel data structure using special ordering of the elements of the embedding tree enabled parallel adaptation of the mesh. Thus the cell-per-processor, fine-grained paradigm of the generic algorithm was supported by the programming interfaces of both architectures. As a consequence, implementation of the same parallel adaptive algorithm on the two systems was relatively simple.

Speeds of over 1 Gflop were attained on the Cray Y-MP/ 8. Speed of 1.6 Gflops was attained on 32 K processors of the CM-2. The speed was found scalable after executing on 8 K, 16 K, and 32 K processors. Comparing the performance of the generic algorithm on the two architectures under consideration, it was seen that the CM-2 outperforms the Cray for very large input sizes primarily due to its raw processing power and the available memory. However, for small input grid sizes, the startup overhead on the CM-2 tends to drastically cut down the performance, whereas the Cray performance drops only marginally. Adaptation caused a substantial reduction in speed on the CM-2 as compared to the Cray. The communication overhead associated with division/deletion of grid cells had to be handled using generic routing, since the CMSSL software was not effective in the considered case of rapidly varying grid.

The near-portability of the implementation of the algorithm across the two radically different architectures and the fact that the major differences in these implementations are in the handling of basic primitives suggests that algorithms for similar problems can be expressed in a higher level parallel language that is essentially independent of either of these two architectures. Such a language would have the basic primitives as fundamental constructs, and the user would encode the algorithm in this higher level language. The program would then be translated by a compiler into code specific to the underlying parallel architecture.

Acknowledgments

This work was supported by National Science Foundation Grant ASC-9111540, and monitored by M. Patrick. It was also supported by NSF Grant ECS-9023770, and monitored by G. Lea. This support is gratefully acknowledged. Computing time on the Cray Y-MP/8 was provided by the Center for High Performance Computing of the University of Texas at Austin. Computing time on the CM-2 was provided by the Thinking Machines Corp. and by the National Center for SuperComputing Ap-plications (NCSA) at the University of Illinois at Urbana-Champaign.

References

[1]Kallinderis, Y., and Vidwans, A., "Parallel Adaptive Navier-Stokes Algorithm Development and Implementation on the Cray Y-MP," *Proceedings of Parallel CFD '92 Conference* (Rutgers Univ.), May 1992, pp. 241–252.

[2]Shankar, V., "A Gigaflop Performance Algorithm for Solving Maxwell's Equations of Electromagnetics," *Proceedings of the 10th AIAA CFD Conference,* AIAA, New York, pp. 584-590; AIAA Paper 91-1578, June 1991.

[3]Agarwal, R. K., "Development of a Navier-Stokes Code on a Connection Machine," *Proceedings of the 9th AIAA CFD Conference,* AIAA, New York, pp. 103–108; AIAA Paper 89-1938, June 1989.

[4]Hammond, S., and Barth, T. J., "Efficient Massively Parallel Euler Solver for Two-Dimensional Unstructured Grids," *AIAA Journal,* Vol. 30, No. 4, 1992, pp. 947–952.

[5]Long, L. N., Khan, M. M., and Sharp, H. T., "A Massively Parallel Three-Dimensional Euler Method," *Proceedings of the 9th AIAA CFD Conference,* AIAA, New York, pp. 89–102; AIAA Paper 89-1937, June 1989.

[6]Das, R., Mavriplis, D. J., Saltz, J., Gupta, S., and Ponnusamy, R., "The Design and Implementation of a Parallel Unstructured Euler Solver Using Software Primitives," AIAA Paper 92-0562, Jan. 1992.

[7]Venkatakrishnan, V., Simon, H. D., and Barth, T. J., "A MIMD Implementation of a Parallel Solver for Unstructured Grids," *Journal of Supercomputing,* Vol. 6, 1992, p. 117–137.

[8]Kallinderis, Y., and Baron, J. R., "Adaptation Methods for a New Navier-Stokes Algorithm," *AIAA Journal,* Vol. 27, No. 1, 1989, pp. 37-43.

[9]Kallinderis, Y., and Baron, J. R., "A New Adaptive Algorithm for Turbulent Flows," *Computers and Fluids Journal,* Vol. 21, No. 1, 1992, pp. 77–96.

[10]Oden, J. T., Strouboulis, T., and Devloo, P., "Adaptive Finite-Element Methods for High-Speed Compressible Flows," *International Journal for Numerical Methods in Fluids,* Vol. 2, No. 7, 1987, pp. 1211–1228.

[11]Kallinderis, Y., "A Finite-Volume Navier-Stokes Algorithm for Adaptive Grids," *International Journal for Numerical Methods in Fluids,* Vol. 15, No. 2, 1992, pp. 193–217.

[12]Kallinderis, Y., "Adaptation Methods for Viscous Flows," Ph.D. Thesis, Dept. of Aeronautics and Astronautics, Massachusetts Inst. of Technology, Boston, MA, CFDL-TR-89-5, May 1989.

[13]Leighton, F. T., *Introduction to Parallel Algorithms and Architectures,* Morgan Kaufman, San Mateo, CA, 1992, Chap. 5.

[14]Kruskal, C. P., Rudolph, L., and Snir, M., "The Power of Parallel Prefix," *IEEE Transactions on Computers,* Vol. 34, 1984, pp. 965–968.

[15]Nassimi, D., and Sahni, S., "Parallel Permutation and Sorting Algorithms and a New Generalized Connection Network," *Journal of the Association of Computing Machinery,* Vol. 29, No. 6, 1982, pp. 642-667.