AIAA JOURNAL
Vol. 32, No. 9, September 1994

Unified Parallel Algorithm for Grid Adaptation on a Multiple-Instruction Multiple-Data Architecture

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

A unified parallel algorithm for grid adaptation by local refinement/coarsening is presented. It is designed to be independent from the type of the grid. This is achieved by employing a generic data template that can be configured to capture the data structures for any computational grid regardless of structure and dimensionality. Furthermore, the algorithm itself is specified in terms of generic parallel primitives that are completely independent of the underlying parallel architecture. The unified parallel algorithm is employed for dynamic adaptation of three-dimensional unstructured tetrahedral grids on a partitioned memory multiple-instruction multiple-data architecture. Performance results are presented for the Intel iPSC/860.

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 three-dimensional flowfields remains very expensive even with use of current vector supercomputers.

Vector computers have accelerated computations only by one or two orders of magnitude compared with scalar machines. An evolving approach in computer architectures is the design of scalable massively parallel computers, wherein a number of processors are involved in executing different portions of the job. Parallel computing appears to be a promising approach for future design applications of CFD.

State-of-the-art parallel architectures can be broadly classified into single-instruction multiple-data (SIMD), shared memory multiple-instruction multiple-data (MIMD), and partitioned memory MIMD architectures. Shared memory MIMD architectures such as the Cray Y-MP are multiprocessor extensions of pipelined vector processors with additional facilities to enable effective utilization of the available multiple processors. On the other hand, SIMD architectures such as the CM-2 are based on the "lockstep" paradigm of parallel computing wherein a large number of processors execute the same instructions on local data. Partitioned memory MIMD architectures provide the user with the flexibility to allocate data as well as each individual processor, thereby enabling fine tuning of the application to the underlying architecture.

Adaptive grid algorithms are employed extensively in CFD. They provide flexibility to adjust the grid during the solution procedure without intervention by the user. A popular method divides initial coarse grid cells, thus creating locally embedded grids. 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 cells locally over regions in which the solution does not vary appreciably. Several such algorithms for two-dimensional grids have been developed.[1,3,4,7] Furthermore, adaptive local refinement/coarsening of unstructured tetrahedral grids has been developed and implemented for complex, three-dimensional geometry flow simulations.[6,8] Different types of grid topology have been employed within the same domain to resolve the various types of flow features.[2]

Very little work has been done on development of efficient parallel algorithms for grid adaptation.[1,5,11] Furthermore, these algorithms are typically designed with a specific architecture in mind.[1] As a result, they have inherent assumptions about the underlying architecture and are not suited for implementation across radically different parallel systems. Extra effort is thus needed when an application is to be ported to multiple architectures. In addition, the data structure employed by these algorithms for representation of the computational grid is critically dependent on the topology and dimensionality of that grid. This further affects the flexibility of such algorithms vis-a-vis the range of problems to which they can be applied.

There is thus a need to develop architecture-independent algorithms for grid adaptation with sustained high performance across a wide variety of parallel machines. It is also imperative that the data structure for such algorithms be decoupled from the specific topology and dimensionality of the computational grid. A generic parallel adaptive algorithm for solution of the Navier-Stokes equations on a two-dimensional quadrilateral mesh, which is portable across SIMD and shared memory MIMD architectures, has been developed in Ref. 5. The main idea is to express the algorithm in terms of generic parallel "primitives" that are independent of the underlying system. The architecture-specific details are encapsulated in the implementation of these primitives on that architecture.

The current work presents a more "unified" approach to parallel grid adaptation. It consists of a unified parallel algorithm for grid adaptation that uses generic parallel primitives similar to those presented in Ref. 5. The present algorithm, however, encompasses partitioned memory MIMD architectures in addition to the two discussed in Ref. 5. It is efficiently implemented on the Intel iPSC/860 and manifests high performance and scalability. Also, the data structure it employs is in the form of a "data template" that can be configured to capture any computational grid regardless of dimensionality and topology.

In the following sections, the grid adaptation method is described for a three-dimensional unstructured tetrahedral grid. Then the unified algorithm, its data structures, and the generic parallel primitives are described. Next, parallel implementation on a partitioned memory MIMD architecture is presented. The performance of this algorithm is evaluated on the Intel iPSC/860.

II. Tetrahedral Grid Adaptation

We consider a three-dimensional unstructured grid made up of tetrahedral cells. Adaptive embedding introduces finer cells in those regions of the field that need to be resolved, while simultaneously removing cells from previously embedded regions that do not require extra resolution. The adaptive grid method considered in this work is discussed in detail in Ref. 6. We present here an overview of the same.

Received June 24, 1993; revision received March 26, 1994; accepted for publication April 1, 1994. Copyright © 1994 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved.

*Graduate Research Assistant; currently at Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801.

†Associate Professor, Department of Aerospace Engineering and Engineering Mechanics. Member AIAA.

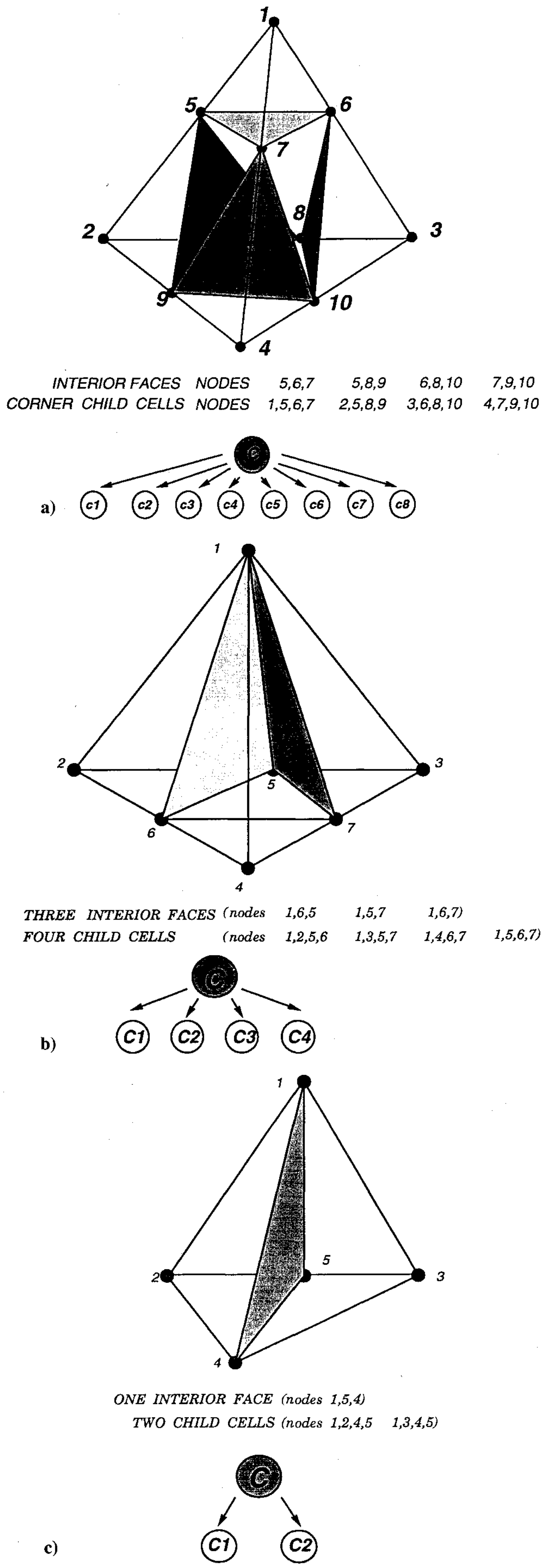

1
2
3
4
5
6
7
8
9
10
INTERIOR FACES NODES 5,6,7 5,8,9 6,8,10 7,9,10
CORNER CHILD CELLS NODES 1,5,6,7 2,5,8,9 3,6,8,10 4,7,9,10
a)
c1 c2 c3 c4 c5 c6 c7 c8
THREE INTERIOR FACES (nodes 1,6,5 1,5,7 1,6,7)
FOUR CHILD CELLS (nodes 1,2,5,6 1,3,5,7 1,4,6,7 1,5,6,7)
b)
C1 C2 C3 C4
ONE INTERIOR FACE (nodes 1,5,4)
TWO CHILD CELLS (nodes 1,2,4,5 1,3,4,5)
c)
C1 C2

Fig. 1 Different types of tetrahedral cell division: a) isotropic division into eight children cells, b) directional division into four children cells, and c) directional division into two children cells.

Grid embedding essentially involves dividing a grid cell by inserting new nodes. The new nodes are introduced in the middle of the edges of the tetrahedra to be refined. Each edge is divided into two edges and each face into four faces after introducing three edges in the interior of the divided face.

Following division of all four faces of the tetrahedron, the midface edges on the faces of the original cell are connected, as shown in Fig. 1a, to form four faces that are interior to the original cell, namely, the faces constituted by the nodes 5-6-7, 5-8-9, 6-8-10, and 7-9-10. This results in four corner children cells, as shown in Fig. 1a, namely, the cells constituted by the nodes 1-5-6-7, 2-5-8-9, 3-6-8-10, and 4-7-9-10. The interior octahedron, which is constituted by the nodes 5-6-7-8-9-10, is divided into four tetrahedra by the proper choice of the shortest diagonal of the octahedron. This scheme thus results in the formation of eight children cells.

Interface cells are formed at the borders between different embedded grids, and they contain additional points in the middle of some of their edges. Two special methods of refinement are used to eliminate such nodes.[6] One method involves the division of only one face of a tetrahedral cell into four children. The cell itself is then divided into four children as shown in Fig. 1b. Face 2-3-4 is divided into four faces, each corresponding to one of the children cells. The second method divides only one out of the six original edges, and the cell is henceforth divided into two children as shown in Fig. 1c, where face 2-3-4 is divided into two faces.

An important aspect of grid adaptation is the deletion of previously divided tetrahedra. Flowfields with evolving flow features are very common, and an adaptive scheme, which eliminates resolution at a region in which additional nodes are not needed anymore, is essential. In principle, the unrefinement process is the inverse of the refinement scheme. The midedge nodes on the "parent" cell edges are removed, resulting in deletion of the corresponding children edges, faces, and cells. Cells that are to be deleted are eliminated along with their sibling cells that were formed from the same parent cell. In this way, the parent cells are recovered after the deletion of their children cells.

The grid-adaptive algorithm, in summary, consists of the following steps: 1) a feature detector identifies the cells to be refined, as well as the cells to be deleted, 2) the cells that are flagged for deletion are removed, and 3) the cells that are flagged for refinement are divided.

III. Generic Data Representation for Adaptive Grids

We employ a generic data representation for computational grids that is independent of grid topology and dimensionality. This is based on the observation that the components of a grid data structure that determine the geometry and dimensionality of that grid are completely irrelevant as far as parallel grid adaptation is concerned. As a result, the data structure for a specific computational grid can be obtained from the generic representation by introducing certain modifications that have no effect on the efficiency of the parallel algorithm using that data structure. It is thus sufficient to express the algorithm as operating on the generic representation.

We introduce a special notation based on lists for describing the generic data representation. A list is defined as an ordered set of N elements where N can be any integer. It should be noted that there is no restriction on what those elements can be. Thus, a list can have other lists as its elements. The "cardinality" or "length" $|L|$ of a list L is defined as the number of elements in the list. The position of an entity in a list will be referred to as the id of that element and is an integer between one and the cardinality of that list.

A. Generic Template for Grid Data Structures

The generic data representation is in the form of a data "template" that holds only the skeleton of a typical grid data structure. The details regarding the topology and dimensionality are left undefined. The utility of this template arises from the fact that it is possible to completely separate those aspects of a grid data structure that define its topology and dimensionality from those that affect the efficiency of the parallel algorithm that uses it. The generic template retains precisely those aspects that are relevant from the perspective of parallel grid adaptation. It consists of the following components.

1) A list N for grid points or nodes. Each element of N is itself a list that holds various "attributes" associated with a particular grid point. For example, a possible set of attributes could consist of three real numbers corresponding to the x, y, and z coordinates of the grid point.

2) A list E for edges, where each element of E is a list of attributes corresponding to that edge. A typical set of attributes would consist of two integers indicating the ids of the two nodes in N that are joined by that edge.

3) A list F for faces, where each element would be a list of attributes for the faces. These attributes would typically list the nodes or edges that constitute that face.

4) A list C of cells, where a cell would be a collection of attributes such as the faces, edges, or nodes forming it.

This template derives its flexibility to represent any computational grid specifically from the attributes in each of its constituent lists. For instance, consider a two-dimensional unstructured grid consisting of triangles. The data structure for this can be derived from the template as follows. Each element of the list N holds the x and y coordinates of a given node in the grid. An element of the list E holds two integers that correspond to the ids of the two nodes forming that edge. Similarly, F holds either the three nodes forming a face (triangle), or the three edges, or both. The list C is omitted altogether. However, regardless of the specific topology of the grid, the structure of the template as a collection of lists interconnected through the ids of entities remains the same.

In the data template defined earlier, no restriction has been imposed on how the entities in a given list are ordered with respect to each other. It was found in Ref. 5 that, from the point of view of efficient parallel grid adaptation, it is essential to impose certain rules for addition and removal of entities from the lists during the refinement/coarsening process. For instance, when a grid gets refined, one or more cells get divided into multiple children. This results in the formation of "adaptation trees," one for each original grid cell as shown in Fig. 2a. The insertion of these new cells into C has to be done in a manner most efficient from the point of view of parallelization. Furthermore, the resulting list of cells should have a structure that allows efficient deletion of these children cells if the need arises later. Since refinement and coarsening are both "local" operations, it is apparent that the most efficient way of insertion would be to insert new children corresponding to a particular cell in its immediate vicinity in the lists. This would, in turn, allow efficient coarsening since a cell to be coarsened would find all of its children next to it in the cell listing.

Because of the reasons just described, the new cells to be created are inserted in such a manner that the final listing of cells corresponds exactly to the postorder listings of all of the adaptation trees placed contiguously one after the other. This special ordering is illustrated in Fig. 2b. It is seen that the newly introduced children of a cell in the original grid are to the immediate left of that cell. This creates a "local access mechanism" for a parent cell that needs to access its children cells for coarsening.

Similarly, when a face gets divided into multiple faces, all of its children are inserted in contiguous fashion on its immediate left in F. The same technique is also applied for division of edges into multiple edges.

B. Data Structure for a Tetrahedral Grid

The data structure for a three-dimensional unstructured grid consisting of tetrahedra follows the aforementioned template. Relevant attributes are assigned to the lists N, E, F, and C as follows:

1) The attributes for a node are its x, y, and z coordinates. An element of the list N is thus a triplet of real numbers $<x, y, z>$.

2) Following are the attributes for an edge:

The ids n_1 and n_2 of the two nodes that it connects.

A "parent id" e_{par} which is the id of its parent edge if it exists. It is 0 otherwise.

A "coarseness indicator" e_{cr} that is an integer having a value of 1 if the edge is a coarse edge and 0 otherwise.

An edge is thus represented by the list $<n_1, n_2, e_{par}, e_{cr}>$.

3) The attributes for a face are as follows:

The ids of the three nodes n_1, n_2, and n_3 and the three edges e_1, e_2, and e_3 that form the face.

A parent id f_{par} for the face similar to that for an edge.

A coarseness indicator for the face f_{cr}.

A face is thus represented by the list $<n_1, n_2, n_3, e_1, e_2, e_3, f_{par}, f_{cr}>$.

4) The attributes for a tetrahedral cell are as follows:

The ids for its four corner nodes n_1, n_2, n_3, and n_4; six edges e_1, e_2, e_3, e_4, e_5, and e_6; and four bounding faces f_1, f_2, f_3, and f_4.

A parent id c_{par} and a coarseness indicator c_{cr} similar to those of a face or an edge.

An integer c_{lev} corresponding to its "embedding level." A cell in the original, unadapted grid is said to have embedding level 0. A cell formed after n adaptations of a level 0 cell is said to have embedding level n.

A cell is thus a list $<n_1, n_2, n_3, n_4, e_1, e_2, e_3, e_4, e_5, e_6, f_1, f_2, f_3, f_4, c_{par}, c_{cr}, c_{lev}>$.

It can be seen that, in the preceding interpretation of the generic template, the attributes that are introduced define only the local relationships or "mappings" between different entities in terms of their ids. There is no attempt to define where that entity lies in three-dimensional space or how it is oriented in relation to other entities in the grid. This property of the data structure is specifically introduced from the considerations of the adaptive method described earlier, which is essentially localized to either a cell or a group of cells.

IV. Generic Parallel Primitives of the Unified Algorithm

This section presents generic parallel primitives that form the backbone of the unified algorithm. All of the architecture-specific details of the algorithm are restricted to the implementation of these primitives on that architecture, whereas the primitives themselves remain system independent. This provides a "screen" between those aspects of grid adaptation that are common to implementations on all systems and those that critically depend on the particular machine.

We first define certain operations on lists and their symbols. These are used to describe the generic primitives as well as the unified algorithm itself in a compact, precise manner.

A. Operations on Lists Pertaining to Adaptation

The following operations on lists are useful from the point of view of grid adaptation:

1) Reference: Given a list L, the reference operator ( ) accesses a particular element of that list. Thus, $L(i)$ is the ith element of list L. It should be noted that this operator can be applied any number of times as long as it is applied to a list. Thus if the ith element of

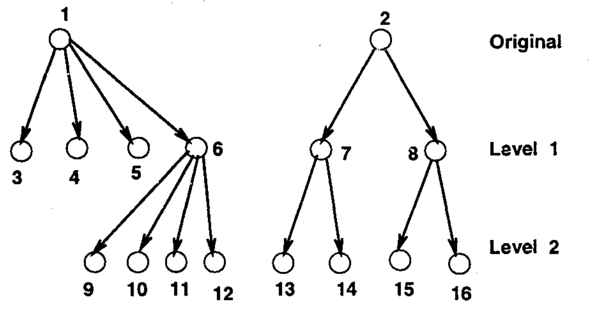


Fig. 2a Formation of adaptation trees of cells. One tree can be formed for each cell of the original unadapted grid.

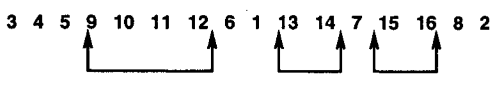


Fig. 2b Postorder representation.

list L is itself a list, then $L(i)(j)$ accesses the jth element of the list $L(i)$. For example, in the data structure for the three-dimensional tetrahedral grid, $C(1)$ accesses the sublist corresponding to the first cell in the cell list C, whereas $C(1)(2)$ accesses the second attribute of the same cell.

2) Extraction: The extraction operator [ ] extracts a list from the elements of another list. If L is a list whose elements are themselves lists, then $L[1]$ "extracts" a list of cardinality $|L|$ from L whose elements are the first elements of each element of L. An example of this operation is shown next. List L consists of three sublists, each containing three elements. The list $L[1]$ is thus formed of the first elements (12, 37, and 102) of each of the three sublists:

List L: <<12, 187>, <37, 91>, <102, 17>>

List $L[1]$: <12, 37, 102>

In the case of the data structure for a three-dimensional unstructured tetrahedral grid, application of the extraction operator to the list of cells C extracts a particular attribute of all of the cells into another list. For example, $C[1]$ contains the ids of the first nodes of all cells in C.

3) Permutation: The permutation operation reorders the elements of a list according to the contents of another list. Thus, if L and M are two lists, then $L\{M\}$ is a list of cardinality $|M|$, and the ith element of $L\{M\}$ is the $M(i)$th element of list L. It should be noted that in this case elements of list M can only be integers from 1 to $|L|$. An example of the permutation operation is shown next. The three elements of list $L\{M\}$ are the fourth, third, and sixth elements of list L, respectively:

List L: $<A, B, C, D, E, F>$

List M: <4, 3, 6>

List $L\{M\}$: $<D, C, F>$

In the case of the three-dimensional tetrahedral grid, the node list N can be permuted according to the list of first nodes of all cells $C[1]$. Thus, $N\{C[1]\}$ is a list of cardinality $C[1]$ containing elements of N permuted according to their ids in $C[1]$.

B. Gather Primitive

The gather primitive copies a given list (L) into another list (M) based on the contents of a third list (N). The list N contains a permutation of the elements of L. This operation can be written in list notation as $M = L\{N\}$. Basically, it uses the permutation operation as a right-hand-side value of an assignment. Thus, the permuted values of the list L are copied into M whereas L itself is unchanged.

In the context of the three-dimensional unstructured tetrahedral grid, one use of the gather operation is the "collection" of the x coordinate of the first node of all of the cells into a list C_x^1. This is easily achieved by assigning $C_x^1 = N\{C[1]\}[1]$. Recall that $C[1]$ is the list of ids of all first nodes and that $N\{C[1]\}$ is the list of all first nodes. The rightmost extraction operator simply extracts the x coordinate out of the permuted list of nodes.

C. Scatter Primitive

The scatter primitive is the dual of the gather primitive and can be represented in list notation as $L\{N\} = M$. This operation uses the permutation operation defined earlier on the left-hand side of an assignment statement. Thus, the elements of L get permuted and are then assigned elements of M on a one-to-one basis. The list M is unchanged, whereas the elements of list L involved in the permutation get new values.

In the case of the three-dimensional unstructured tetrahedral grid, one use of the scatter operation is the "marking" of the first nodes of all of the cells with a specific integer (int) by creating a list F of cardinality $|N|$ that contains that integer in a particular position if and only if the corresponding element in N is the first node of some cell. This is easily achieved by assigning $F\{C[1]\} = \text{int}$.

D. Elemental Operations

This primitive performs elementwise computation on one or more lists. A given operation is performed on corresponding elements of each list to get an element of the new list. An example of this primitive can be written in list notation as $L_3 = L_1 + L_2$, where L_1, L_2, and L_3 all have the same cardinality. The preceding example creates a list L_3 wherein each element is obtained by adding the two corresponding elements in the other two lists.

In case of the three-dimensional unstructured tetrahedral grid, one instance where the elemental operations primitive is used is when all elements of a list are to be multiplied by a scalar. For instance, in the unified algorithm, the list C_f, which is a list of flags over all cells, has to be multiplied by n, where n is the number of cells into which a cell gets divided. This is an example of the elemental operations primitive and is expressed as $C_f = n * C_f$.

E. Prefix Primitive

The prefix primitive performs a prefix operation on a given list of numbers. For each element in the initial list, its prefix is obtained by summing together all of the elements on its left, including itself.[9] A list L is said to be a prefix of list M [$L = \text{prefix}(M)$] if and only if

$$L(i) = \sum_{j=1}^{j=i} M(j), \forall i, 1 \le i \le |M|$$

In the unified algorithm, the prefix primitive is used to calculate the new positions of entities in a list after refinement/coarsening. For instance, in the division case, the input to the prefix operation is a list of integers of the same length as the list of entities with an integer in position i holding the number of entities to be added as a result of division of the ith entity of the original entity list. For an entity not to be divided, this integer is 0. After the prefix operation, the ith integer indicates the number of positions through which the ith entity has to move to reach its correct position in the entity list after division.

F. Monotone Rout Primitive

The monotone rout is a special kind of scatter operation where the relative order of the elements being scattered does not change.[10] Its representation in list notation as well as algorithmic form is exactly the same as the scatter operation. What makes the monotone rout unique is the fact that if list L is "routed into" list M based on a routing pattern in list N, then the relative order of any two elements in L remains the same after the routing.

The monotone rout is used to rearrange an entity list L after refinement/coarsening during the adaptation process. The scattering pattern in this case is the output of the prefix operation with i added to the integer in the ith position, with i being an integer from 1 to $|L|$.

V. Unified Parallel Algorithm

The unified parallel algorithm for grid adaptation can be described in terms of the generic parallel primitives and the data template defined earlier. It employs a "hierarchical approach" wherein entities are divided and undivided in ascending and descending order of dimensionality, respectively. In other words, edges, faces, and cells are divided in that order during refinement, whereas unrefinement is performed in the reverse order. The division/deletion procedure is essentially the same for all entities algorithmically, with the only differences being in the number of new entities added and the assignment of attributes to the newly created entities. For instance, during division of cells in a three-dimensional tetrahedral grid, eight new cells are created for each cell divided, whereas during division of edges, only two new edges are created for each edge to be divided. As a result, this procedure can be described in an entity-independent fashion, and the overall algorithm can then be expressed as repeated applications of this procedure to various entity lists.

A. Entity Division/Deletion Procedure

The procedure takes in as inputs an integer n and two lists I and M, where I is the list of entities and M is a list of integers of the same cardinality as list I. An element in M has a value of 1 if the corresponding element in I is to be divided, -1 if it is to be deleted, and 0 if neither. The integer n specifies how many children entities are formed when an entity in I gets refined or how many are deleted when it is coarsened. A positive value of n denotes division, whereas a negative value denotes deletion. The steps in the procedure are as follows:

1) $M_p = n * M$. This step multiplies all elements of the list M by n and creates a new list M_p. The list M_p now holds the number of new entities that are to be introduced or removed for each entity. Thus, M_p contains 0 if the corresponding entity is neither to be divided nor deleted and n otherwise.

2) $M_p = \text{prefix}(M_p)$. The list M_p is assigned the result of performing a prefix operation on it. It now holds for each entity the "correction" that has to be applied to its position in the entity list as a result of the division/deletion.

3) $M_p = M_p + I_{\text{pos}}$, where I_{pos} is simply a list of integers containing the integer i in its ith position. As a result of this operation, the list M_p now contains the new position for the ith entity in position i, with i being an integer from 1 to $|L|$.

4) $I_1 = \text{monotone\_rout}\ (I, M_p)$. This step creates the new entity array I_1 by calling the monotone_rout primitive with the scattering pattern M_p.

5) Set up the attributes for the newly created entities. This step is performed only in case of division. The particulars of this step depend on the type of entities in the entity array as well as their attributes. For example, in the case of the three-dimensional unstructured grid, division of the list of cells would be followed by a step to fill in the cell-to-face, cell-to-edge, and cell-to-node mappings for each newly created cell.

It can be seen that this procedure uses only the generic primitives defined earlier as its building blocks.

B. Hierarchical Adaptation Method

The grid adaptation algorithm consists of a refinement step and an unrefinement step. During the refinement step, entities that are marked for division are divided by application of the division/deletion procedure described earlier. The input to this step is a list of "flags" of cardinality equal to that of the highest dimensional entity list, i.e., cells in the case of a three-dimensional tetrahedral grid. An entity is flagged 1 if it is to be divided and 0 otherwise. The refinement is performed as follows:

1) The flags for the highest entity list are scattered to all entities of lower dimensions. Thus, in the case of a three-dimensional tetrahedral grid, a cell to be divided flags all of its constituent faces and edges as candidates for division.

2) The division/deletion procedure is applied to all entity lists starting from the list corresponding to the entity lowest in dimensionality. This ensures that when an entity list of a particular dimensionality is being divided, all lists of lower dimensionality are already divided, thereby enabling the division of higher entities. In the case of a tetrahedral grid, the edges are divided first, followed by the faces and cells in that order.

In the unrefinement step, the division/deletion procedure is applied to the entity lists in the reverse order. The input is a list of flags similar to that in the case of refinement, except that an entity to be deleted is marked -1. The overall unrefinement is performed as follows:

1) The highest entity list scatters the flags to all lower entities. This step is exactly the same as the first step in refinement.

2) Those entities in the highest entity list that are not to be deleted now scatter their flags, which are all zeros, to all lower entities. This is termed as the "constraint imposition step."

3) The division/deletion procedure is now applied to all entity lists in descending order of dimensionality. Thus, in the case of the three-dimensional tetrahedral grid, the cells are deleted first, followed by the faces and edges.

VI. Implementation on a Partitioned Memory MIMD Architecture

A generic, parallel adaptive Navier-Stokes algorithm has been developed[5] for shared memory MIMD (Cray Y-MP) and SIMD (CM-2) architectures. The grid is a two-dimensional quadrilateral grid that is structured in two dimensions initially but is rendered unstructured by adaptation. The adaptive Navier-Stokes algorithm itself is based on generic parallel primitives that are similar to those used by the unified algorithm. In this section, we demonstrate the universal nature of the unified algorithm by presenting its implementation and performance results on a partitioned memory MIMD architecture, the Intel iPSC/860.

A partitioned memory MIMD architecture typically consists of a collection of multiple processors connected together by a high-speed interconnection network. Each processor has the freedom of executing its own set of instructions on its data. There is no notion of shared memory, and the only way that processors can interact is through the connecting network. The Intel iPSC/860 is an example of such an architecture. It has 128 processors, each of which is an Intel i860 microprocessor with a clock rate of 40 MHz, 8 Mbytes of memory, and a direct connect module. A node has a peak performance of 60 Mflops in 64-bit arithmetic. A bidirectional hypercube interconnect facilitates communication across the nodes.

The same user program is executed on all of the processors, each with its own set of data. Coordination among processors is achieved through message passing for which "send" and "receive" primitives are provided. These can be either synchronous or asynchronous depending on the requirement of the algorithm. The programming paradigm is essentially that of any ordinary sequential machine. That is, the actual structure of any program written for the iPSC has basically a sequential form with additional calls to the message passing routines for synchronization among processors.

We consider the problem of local refinement/coarsening of a three-dimensional unstructured tetrahedral grid. The grid is partitioned among the processors in such a way that any cell belongs to exactly one processor, whereas faces, edges, and nodes can be shared by more than one processor. The partitions are assigned one each to all of the processors. Details regarding the partitioning algorithm are discussed in Ref. 11.

The data structure for the grid is derived from the generic template. It consists of all of the components described earlier and some additional information regarding the entities that are shared with other processors. Each processor maintains this data structure for its portion of the computational grid. The additional components of a processor's data structure are as follows:

1) Considering the total number of processors to be P, $P-1$ pairs of integers are added as attributes to the attribute lists of all faces, edges, and nodes. For a given entity, there is one pair for each other processor in the system. The first integer of the pair is the processor number, and the second integer gives the id of that entity on that processor. For an entity that belongs to only one processor, the second integer is -1 for all pairs. Thus the attributes for a node on processor 0 for a system consisting of four processors are $<x, y, z, <0, -1>, <1, i_1>, <2, i_2>, <3, i_3>>$, where i_1, i_2, and i_3 are "images" or ids of that node on processors 1, 2, and 3, respectively.

2) Three lists F_{adj}, E_{adj}, and N_{adj} that hold information regarding processors that share entities with the processor are employed. The list F_{adj} is a list of integers denoting the numbers of all processors that share at least one face with this processor. Similarly, E_{adj} and N_{adj} contain the numbers of processors sharing at least one edge and node, respectively.

The division part of the algorithm takes in as input the data structure for the computational grid and a list F_c of integers of cardinality equal to that of the list of cells. A cell is to be divided if the corresponding element in F_c is 1, and it is to be deleted if it is -1. The cell is unchanged by adaptation otherwise. Additional lists F_f and F_e are used to hold similar "flags" for the faces and edges, respectively.

1) For each cell i, where $F_c(i) = 1$,
   a) Set $F_f(C[j]) = 1$, for $j = 11, 14$;
   b) Set $F_e(C[j]) = 1$, for $j = 5, 10$.
2) Call procedure divide (E, F_e).
3) Call procedure divide (F, F_f).
4) Call procedure divide (C, C_f).

Recall from Sec. III.B that attributes 11–14 for a cell in the three-dimensional tetrahedral grid data structure hold the ids of its faces, whereas attributes 5–10 hold the edges.

The preceding steps illustrate the importance of the hierarchical approach to adaptive refinement employed by the unified algorithm. The edges, which are dimensionally the lowest, are divided first so that higher entities can be divided without consideration to the fact whether their constituent edges have been refined. The faces are divided next, followed by the cells. The division procedure defined in Sec. V.A is invoked repeatedly to perform the bulk of the work.

The inputs to the deletion algorithm are the grid data structures, a list of flags for the cells an element that contains a -1 if the corresponding cell is to be deleted and a 0 otherwise. Additional lists are used to hold similar flags for the faces and edges as well. The steps in the algorithm are as follows:

```
1) For each cell $i$, where $F_c(i) = -1$,
    a) Set $F_f(C[j]) = -1$, for $j = 11, 14$;
    b) Set $F_e(C[j]) = -1$, for $j = 5, 10$.
2) For each cell $i$, where $F_c(i) = 0$,
    a) Set $F_f(C[j]) = 0$, for $j = 11, 14$;
    b) Set $F_e(C[j]) = 0$, for $j = 5, 10$.
3) Call procedure delete $(E, F_e)$.
4) Call procedure delete $(F, F_f)$.
5) Call procedure delete $(C, C_f)$.
```

The first scatter step just listed can be called an "intention to delete" step where every cell that is to be deleted marks all of its associated entities with a -1 flag. The second scatter step is the constraint imposition step wherein all cells that are to be retained ensure that none of their entities are deleted by the deletion module. Thereafter, the deletion module is called once each for cells, faces, and edges.

Some additions to the preceding manifestation of the unified algorithm are required for synchronization across partition boundaries. During division, after the faces and edges of cells to be divided are marked with flags as per the unified algorithm, each processor performs a "logical OR" operation for each marked face/edge that is shared with other processors. This basically involves exchanging information regarding that entity with all processors that share it. This is because if one processor divides its copy of the shared entity, then all processors sharing that entity have to divide their own copies as well.

The deletion part of the adaptive algorithm needs only one enhancement to the unified version. This is done in the constraint imposition step when cells that are not marked for deletion impose constraints against the deletion of their faces and edges. A logical OR operation, exactly the same as the one just described, is performed on all shared faces and edges. This is because even if one processor does not delete its copy of a shared face or node, then no other processor can delete its copy of the same entity.

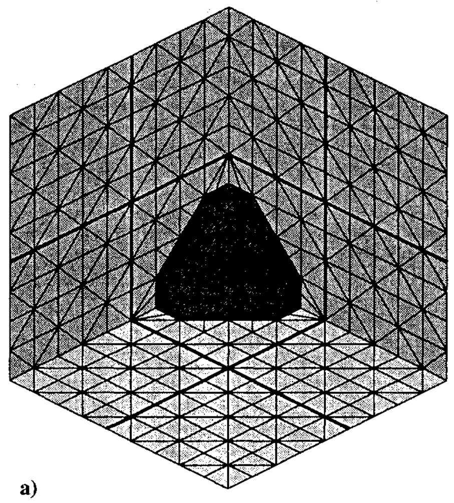
Figure 3 shows an application of the preceding algorithm for parallel adaptation of a three-dimensional channel grid with a moving blast wave front. The wave starts from the origin and propagates away from it, continuously growing in size. The adaptive algorithm monitors the position of the wave and places locally finer grids in its vicinity. Figure 3b shows the adapted grid with the wave front having moved away from its previous position of Fig. 3a. It can be seen that the adaptation in the vicinity of the previous position is removed, whereas additional cells are introduced around the new position of the wave. The amount of adaptation is seen to increase in proportion to the size of the wave.

VII. Performance Results on the iPSC/860

We present performance results on the Intel iPSC/860 for a three-dimensional channel grid consisting of tetrahedra. The grid is adapted once by dividing a certain number of cells within each processor partition. These are then immediately deleted. The total execution time for this step is measured on each processor. The total time for execution of the algorithm is taken to be the maximum of all of the individual processor timings. The different parameters to be considered are 1) the number of processors into which the original grid is divided, 2) the partitioning method used, and 3) the number of cells divided/deleted within each partition.

We consider two different methods for grid partitioning for performance evaluation of the parallel adaptive algorithm. The "strip" partitioning method partitions the original grid among processors by using cutting planes along one of the three coordinate axes. The "all-round" partitioning method partitions the initial grid using cutting planes along all three coordinate axes. Details regarding these methods are discussed in Ref. 11. These two methods differ significantly in terms of communication costs, and as a result, it is insightful to evaluate the performance of the adaptive algorithm for these two partitioning techniques. Figure 4 shows the table of timings for the all-round partitioning case. The rows correspond to the number of processors involved, whereas the columns denote the number of cells that were divided in each case. Figure 5 shows timings for the same instances of the algorithm wherein the grid is initially divided among processors using the strip partitioning method. Two trends can be observed vis-a-vis the execution times for both forms of grid partitioning. The total execution time reduces as additional processors are introduced in the system, with the number of cells being adapted remaining the same. This implies that the communication overhead caused by additional processors does not overwhelm the efficiency of the overall algorithm. The second trend is observed in case of increasing number of cells being adapted for a given number of processors. The execution time increases with increase in the amount of adaptation. This is to be expected as a larger amount of adaptation implies additional work per processor in the system.

Figure 6 graphically illustrates the variation in execution time with the number of processors, with the number of cells being adapted held fixed at 200. The dashed line corresponds to a strip


a)

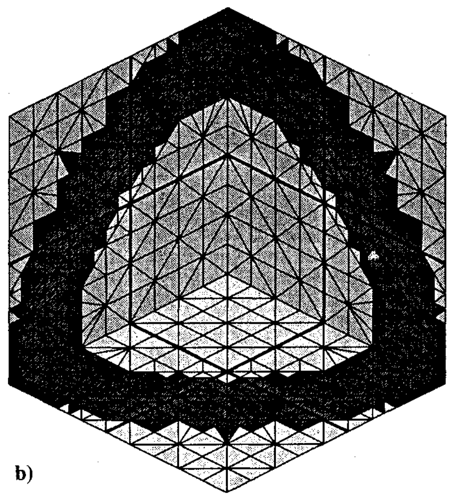

b)

Fig. 3 Illustration of the three-dimensional grid adaptation process involving both division and deletion of tetrahedra. A blast wave is shown moving through a channel grid. Dark area shows adaptation around the position of the wave: a) initial position of the wave and b) position of the wave as it is about to move out of the channel.

P \ C	1	50	100	150	200
1	1.048	1.126	1.182	1.219	1.248
2	0.889	0.961	1.002	1.060	1.102
4	0.592	0.639	0.755	0.835	0.896
8	0.459	0.589	0.682	0.771	0.804
16	0.334	0.458	0.586	0.667	0.747
32	0.247	0.364	0.410	0.536	0.622

Fig. 4 Timing results in seconds on the iPSC/860 for one adaptation of a three-dimensional channel grid with all-round partitioning being used. Rows correspond to partitioning of the grid into a given number of processors (P), whereas columns correspond to different number of cells being adapted (C).

P \ C	1	50	100	150	200
1	1.048	1.126	1.182	1.219	1.248
2	0.889	0.961	1.002	1.060	1.102
4	0.514	0.592	0.642	0.686	0.735
8	0.300	0.389	0.449	0.501	0.549
16	0.192	0.277	0.339	0.415	0.477
32	0.157	0.213	0.262	0.319	0.361

Fig. 5 Timing results in seconds on the iPSC/860 for one adaptation of a three-dimensional channel grid with strip partitioning being used. Rows correspond to partitioning of the grid into a given number of processors (P), whereas columns correspond to different number of cells being adapted (C).

partitioning, whereas the solid line corresponds to an all-round partitioning. The scale on the x axis is logarithmic with base 2. It can be seen that strip partitioning results in lesser execution time as compared with the time for the corresponding instance with all-round partitioning. Furthermore, this discrepancy increases with an increase in the number of processors involved in the execution. This is due to the startup overhead of messages during interprocessor communication. In the case of the strip partitioning, a processor only needs to communicate with at most two other processors, whereas in the case of the all-round partitioning, the number of adjacent processors can be as high as eight. Since the communication takes place across interpartition boundaries that are essentially two dimensional, the total communication overhead is dominated by the startup overhead of each message rather than the actual length of the message. As a result, this overhead is directly proportional to the number of messages being sent during adaptation. The number of messages in turn depends on the number of adjacent processors with which a given processor needs to communicate. Consequently, the communication overhead tends to increase with an increase in the number of processors in the all-round partitioning case, leading to the discrepancy in execution times.

We define the speedup S obtained in the case of P processors as the ratio of the execution time for the algorithm on one processor to the execution time on P processors, all other parameters remaining the same. Scalability of the algorithm can then be defined as variation of the speedup with the number of processors involved in the execution. However, this definition of scalability is valid only if the overall input size remains the same as P increases. In the case of the adaptive algorithm, the lengths of the various lists in the grid data structure can be taken as the input size. Clearly, this varies due to addition and deletion of entities during adaptation. As a result, a modified definition of scalability is needed that can allow accurate appraisal of the performance of the adaptive algorithm with an increasing number of processors.

We define scalability as the variation in speedup of the parallel adaptive algorithm when the amount of adaptation per processor is "negligible" in comparison with the original input size. For our purposes, negligible adaptation is taken to be division and deletion of only one cell per processor in the original grid. Division of only one cell can result in introduction of at most 8 extra cells, 24 new faces, and 12 additional edges and grid points each. If the size of the original grid is sufficiently large, this increase in input size is negligible.

Figure 7 depicts the scalability of the parallel adaptive algorithm. Speedup is plotted on the y axis and the number of processors is plotted on the x axis with a logarithmic scale of base 2. Curve A corresponds to a strip partitioning, whereas curve B corresponds to an all-round partitioning of the initial domain. Ideal or linear speedup is shown by the dashed line. It is interesting to note the slow rise of the curves from $P=1$ to $P=2$ and the steep rise thereafter. This is because the one-processor case involves no communication at all. This shows that the communication overhead is a significant part of the total execution time for $P>1$.

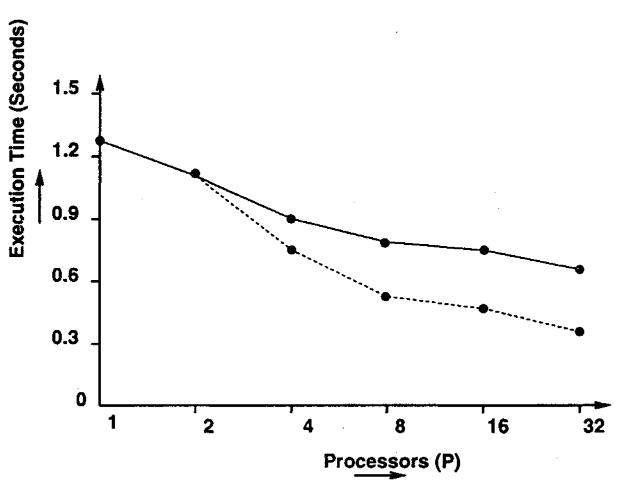


Fig. 6 Reduction in execution times with increasing number of processors for one adaptation of a three-dimensional channel grid wherein 200 cells are divided and deleted; ---: strip partitioning of initial grid and ——: all-round partitioning of initial grid (the scale on the x axis is logarithmic).

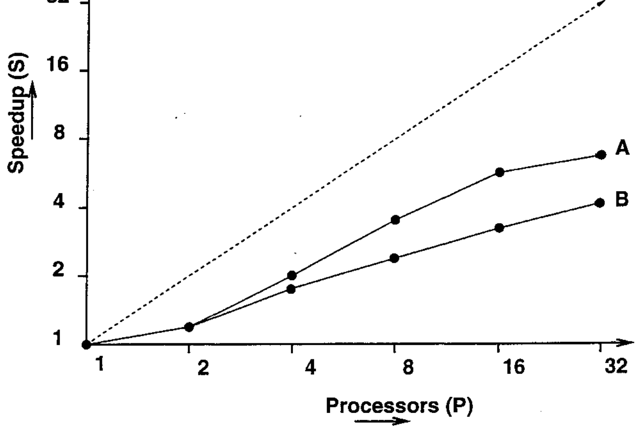


Fig. 7 Scalability results on the iPSC/860 for one adaptation of the channel grid wherein only one cell on each processor is divided and deleted; curve A: strip partitioning, curve B: all-round partitioning, and ---: ideal speedup (the scale on both axes is logarithmic).

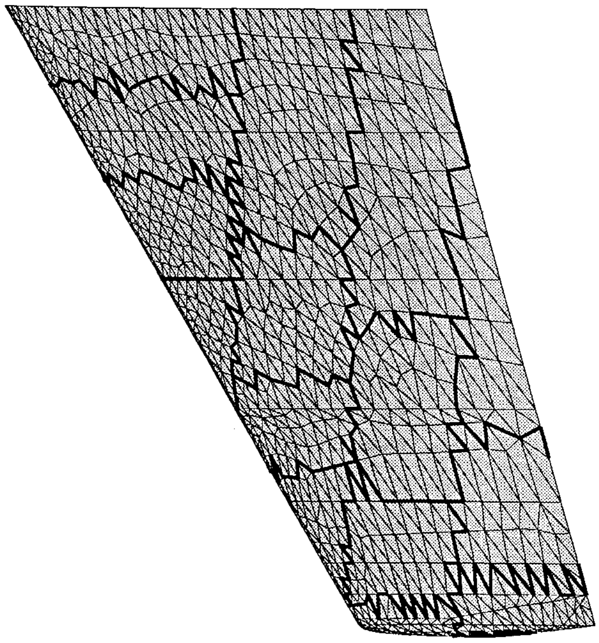


Fig. 8 Surface plot of the original grid for an ONERA M6 wing consisting of 35,008 tetrahedral cells partitioned among 128 processors of the iPSC/860. Thick lines denote interpartition boundaries.

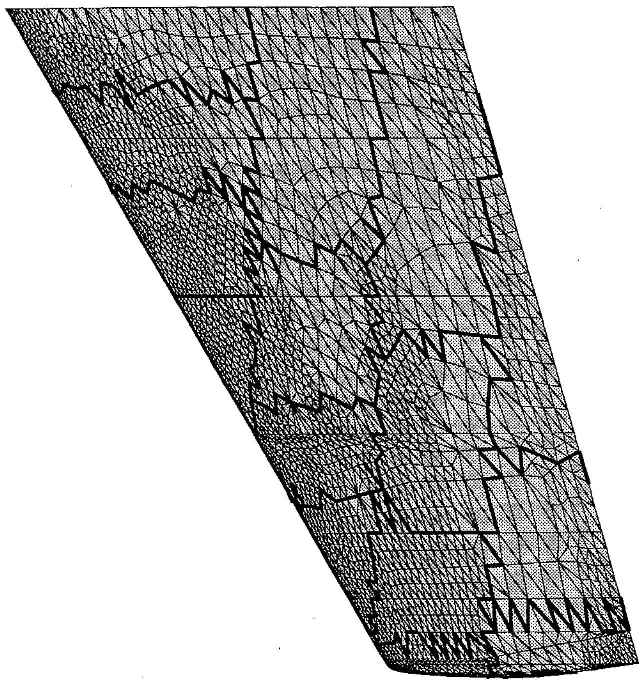


Fig. 9 Surface plot of a one-level adapted grid for the ONERA M6 wing consisting of 103,580 cells obtained in a total execution time of 1.65 s on a 128-processor iPSC/860. Thick lines denote interpartition boundaries.

The case of adapting the grid around a wing in transonic flow is considered next. Figure 8 shows the surface plot corresponding to an initial, unadapted grid for the ONERA M6 wing. The grid consists of 35,008 tetrahedral cells that are partitioned among all 128 processors of the iPSC/860. The problem could not fit in fewer than 128 processors due to the memory limitation of 8 Mbytes per processor of the iPSC/860. As a consequence, scalability cannot be tested on this case. Nevertheless, this is considered a severe test of performance of the parallel adaptive algorithm. The all-round partitioning method employs four cutting planes along each of the x and z axes, as well as eight cutting planes along the y axis. The view for the surface plot is taken along the y axis. Transonic flow of Mach number 0.84 involves two shock waves on the upper surface of the wing: a fore shock close to the leading edge and an aft shock that forms a lambda system with the former.[6] The parallel adaptive algorithm is employed to perform one step of adaptation on this initial grid. A total of 9000 cells from the initial grid were divided, resulting in an adapted grid consisting of 103,580 cells. Figure 9 shows the surface plot for the one-level adapted grid. A local embedded grid has been placed by the algorithm in the regions of the leading edge and shock waves.[6] The adapted grid was obtained in a total execution time of 1.65 s.

VIII. Conclusions

A unified parallel algorithm for grid adaptation by local refinement/coarsening was developed. Creation of a generic template for the data structure enabled the algorithm to be independent of the dimensionality and topology of the computational grid. Generic parallel primitives were defined as building blocks of the parallel algorithm. This is a major step toward achieving portability of the algorithm to different parallel architectures.

The algorithm was employed to carry out adaptation of a three-dimensional unstructured tetrahedral grid on a partitioned memory MIMD machine. Scalability of speedup was evaluated on the Intel iPSC/860 for a three-dimensional channel grid. Reduction of execution time with an increasing number of processors depends to a small extent on the method used for initial partitioning of the computational grid.

The parallel adaptive algorithm was also employed to obtain a one-level adapted tetrahedral grid for the ONERA M6 wing in a total execution time of 1.65 s on all 128 processors of the iPSC/860.

Acknowledgments

This work was supported by Defense Advanced Research Projects Agency Grant DABT 63-92-0042. Computing time on the Intel iPSC/860 was provided by the NAS Division of NASA Ames Research Center, Moffett Field, CA.

References

[1]Berger, M. J., and Saltzman, J., "AMR on the CM-2," Research Inst. for Advanced Computer Science, NASA Ames Research Center, TR 92.16, Aug. 1992.

[2]Ward, S., and Kallinderis, Y., "Hybrid Prismatic/Tetrahedral Grid Generation for Complex 3-D Geometries," AIAA Paper 93-0669, Jan. 1993.

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

[4]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.

[5]Kallinderis, Y., and Vidwans, A., "Generic Parallel Adaptive-Grid Navier-Stokes Algorithm," *AIAA Journal*, Vol. 32, No. 1, 1994, pp. 54–61.

[6]Kallinderis, Y., and Vijayan, P., "Adaptive Refinement/Coarsening Scheme for Three-Dimensional Unstructured Meshes," *AIAA Journal*, Vol. 31, No. 8, 1993, pp. 1440–1447.

[7]Lohner, R., "The Efficient Simulation of Strongly Unsteady Flows by the Finite-Element Method," AIAA Paper 87-0555, Jan. 1987.

[8]Lohner, R., and Baum, J., "Numerical Simulation of Shock Interaction with Complex Geometry Three-Dimensional Structures Using a New Adaptive H-Refinement Scheme on Unstructured Grids," AIAA Paper 90-0700, Jan. 1990.

[9]Kruskal, C. P., Rudolph, L., and Snir, M., "The Power of Parallel Prefix," *IEEE Transactions on Computers*, Vol. 34, No. 3, 1984, pp. 965–968.

[10]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.

[11]Vidwans, A., Kallinderis, Y., and Venkatakrishnan, V., "Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids," *AIAA Journal*, Vol. 32, No. 3, 1994, pp. 497–505.