AIAA JOURNAL
Vol. 32, No. 3, March 1994

Parallel Dynamic Load-Balancing Algorithm for Three-Dimensional Adaptive Unstructured Grids

A. Vidwans* and Y. Kallinderis†
University of Texas at Austin, Austin, Texas 78712
and
V. Venkatakrishnan‡
Institute for Computer Applications in Sciences and Engineering, Hampton, Virginia 23681

Adaptive local grid refinement/coarsening results in unequal distribution of work load among the processors of a parallel system. A novel method for balancing the load in cases of dynamically changing tetrahedral grids is developed. The approach employs local exchange of cells among processors to redistribute the load equally. An important part of the load-balancing algorithm is the method employed by a processor to determine which cells within its subdomain are to be exchanged. Two such methods are presented and compared. The strategy for load balancing is based on the *divide-and-conquer* approach that leads to an efficient parallel algorithm. This method is implemented on a distributed-memory multiple instruction multiple data system.

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 the 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 extensions of pipelined vector processors with additional facilities to enable effective utilization of the available multiple processors. However, the scalability of such architectures is severely limited due to the inherent bottleneck of the common, shared memory. On the other hand, SIMD architectures such as the CM-2 are based on the "lockstep" paradigm of parallel computation wherein a large number of processors execute the same instructions on local data. Although this enhances scalability, the overhead associated with communication among these processors can sometimes prove to be a major bottleneck.[1]

Partitioned memory MIMD architectures provide a good compromise between the other two types of systems. The user has the flexibility to allocate data as well as tasks to each individual processor, thereby enabling fine tuning of the application to the underlying architecture. There is a price to be paid, however, in terms of additional user responsibility to coordinate the processors by exchanging the relevant information via message passing. In the case of an adaptive grid algorithm, it is also necessary to insure uniform distribution of the work load among processors. The work load on a processor is characterized by the number of grid cells allocated to it. Local grid adaptation results in the creation of new grid cells and hence potential unequal distribution of the work load among the processors. The regions where new cells will appear are generally not known a priori, which implies that the dynamic load distribution is unpredictable. As a consequence, a dynamic load-balancing algorithm is required. Furthermore, such an algorithm has to be as efficient as possible; otherwise it could itself be a source of degradation in overall performance.

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 local 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.[2,3] Furthermore, adaptive local refinement/coarsening of unstructured tetrahedral grids has been developed and implemented for complex, three-dimensional geometry flow simulations.[4,5]

Relatively few parallel solvers for unstructured grids have been developed.[6-9] These typically employ static grids, which are partitioned and allocated to processors a priori by the user. Algorithms for load balancing in the case of two-dimensional grids that do not change (also referred to as partitioning algorithms) have been presented in Refs. 9–11. Two approaches are simulated annealing and recursive bisection. Simulated annealing changes the work distribution randomly among processors.[11,12] This method requires iterations to achieve optimal balance of the load, which can be expensive. Eigenvalue recursive bisection[9,11,13,14] requires solution of eigenvalue problems and is also quite expensive for dynamic load balancing. Orthogonal recursive bisection uses cutting planes to partition the computational grid based on centroidal coordinates of the cells. This approach is the least expensive among the aforementioned methods.

An important factor affecting the communication costs in the case of a computational grid partitioned among several processors is the cumulative length of all interpartition boundaries. This is usually defined as the total number of grid entities of a particular type such as grid points that are on any interpartition boundary. It is therefore important that the

Received March 13, 1993; revision received July 19, 1993; accepted for publication July 22, 1993. Copyright © 1993 by the American Institute of Aeronautics and Astronautics, Inc. All rights reserved.

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

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

‡Research Scientist, MS 132C, NASA Langley Research Center. Member AIAA.

grid-partitioning algorithm minimizes the cumulative boundary length.

For the aforementioned reasons, the present work utilizes orthogonal recursive bisection for the initial partitioning of the computational grid. Depending on the orientation of the cutting planes, a partitioning can be either a "strip" or an "all-round" partitioning.[9] The strip partitioning involves cutting the original domain along one of the three coordinate axes. The all-round partitioning involves cutting planes along more than one coordinate axis.

An approach to load balancing in the case of dynamically varying computational grids is to repartition the computational domain from scratch using the algorithm employed for the initial partitioning of the grid. However, all of the aforementioned algorithms are highly expensive computationally and also are not amenable for efficient parallelization, thus making them unsuitable for a situation where the grid is adapted frequently. As a consequence, a parallel dynamic load balancer is required, which can be efficiently executed as often as needed without causing a bottleneck on the overall performance of the system.

This work develops a parallel algorithm for balancing the work load among processors dynamically during an adaptive tetrahedral grid simulation. This is achieved by local migration of cells from one processor to the neighboring ones to distribute the load equally. An essential part of local migration is the method used by a processor to determine which cells within its partition are to be migrated. This directly affects the shape and structure of the interpartition boundaries after migration. Two approaches to this problem are presented and compared. The overall load-balancing strategy is based on the *divide-and-conquer* technique, which results in an efficient, deterministic parallel algorithm. The method is implemented on the Intel iPSC/860.

In the following sections, the grid-partitioning algorithm is presented. Then, the load-balancing strategy and the local migration algorithm are described. The effectiveness of the algorithm is evaluated via test cases, including an adapted tetrahedral grid for transonic flow around a wing.

II. Original Grid Partitioning

The computational grid is divided into as many subgrids as processors using a partitioning algorithm. This partitioning is done in such a way that nodes, edges, and faces can be shared among multiple processors, but cells are entirely within a particular processor. The grid-partitioning algorithm consists of the following steps: 1) coordinate-based grouping of cells and 2) splitting of the grid data structures.

A. Coordinate-Based Grouping

The cells in the grid are divided into groups based on their centroidal coordinates by cutting planes that are provided as input. Each processor is assigned a partition. By suitably setting the orientation of the cutting planes, one can realize any arbitrary partitioning of the initial grid. In the present work, the following two instances of grid partitioning are considered:

1) "Strip" partitioning: In this kind of partitioning, the cutting planes are all along one of the three coordinate axes. This partitioning typically results in longer interpartition boundaries, but a given processor has at most two neighboring processors with which to communicate.

2) "All-round" partitioning: In this kind of partitioning, the cutting planes are aligned along more than one coordinate axis. This results in interpartition boundaries that are shorter than those resulting from the strip partitioning, but a given partition can have an arbitrary number of neighboring partitions.

B. Splitting of Grid Data Structures

The three-dimensional unstructured grid is uniquely defined as a collection of the following sets:

1) A set of "nodes" or points in three-dimensional space, N, a member of which is defined by its x, y, z coordinates.

2) A set of "edges" E, where each edge is uniquely specified by a pair of nodes from the set N and is said to "connect" the two nodes. Further, two edges are said to be *adjacent* if and only if they have exactly one node in common. An edge is geometrically represented by a straight line joining the two grid points in three-dimensional space.

3) A set of "faces" F, where each face is uniquely specified by a triplet of edges from the set E where each edge in the triplet is adjacent to the other two. Clearly, a face is a triangle in three-dimensional space and can also be uniquely defined by the three grid points forming the triangle. A face is said to be adjacent to another face if and only if they have one and only one edge in common.

4) A set of "cells" C, where each cell is uniquely specified by a quadruplet of faces from the set F such that every face in the quadruplet has exactly one edge in common with the other three faces. A cell can thus be represented as a tetrahedron in three-dimensional space. This implies that a cell can also be uniquely specified either by its six edges or its four corner nodes.

Cells, faces, edges, and nodes are collectively referred to as "entities." The data structures defining the grid are split into P subsets, where P is the number of processors in the system. Each processor is assigned the subsets that completely specify the subdomain allocated to it.

Within a subdomain, an entity of a particular type (cell/face/edge/node) is assigned a unique integer identification number (ID) between 1 and the total number of entities of that type in that subdomain. This simplifies the storage mechanism since the correlation between entities of different types can now be expressed in terms of these IDs. Thus an edge is a pair of integers (n_1, n_2) where n_1 and n_2 are IDs of two nodes within the subdomain. These IDs are completely local to a subdomain. There is no notion of global numbering of entities across all processors. This is of particular significance in the case of a dynamically varying computational grid. In a global numbering scheme, all processors have to communicate with each other whenever a global number is to be assigned to a newly created entity, such as a face or a cell. This entails additional communication overhead and can prove to be a bottleneck. Furthermore, the algorithms for dynamic updating of global numbers are cumbersome and are not amenable to parallelization.

An entity that is shared between several processors has multiple copies of itself, one on each processor that shares it. Furthermore, for every shared entity within its partition, a

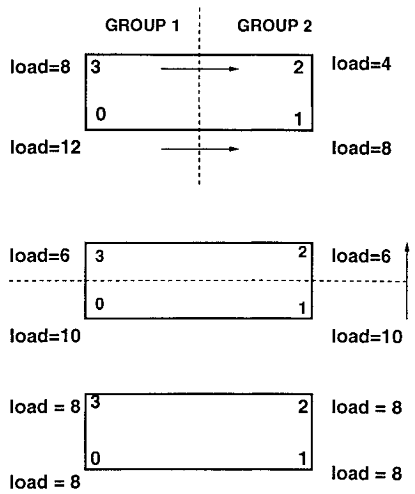


Fig. 1 Schematic of the divide-and-conquer method for balancing the load among four processors: a) initial load distribution, b) load distribution after step 1, and c) load distribution after step 2.

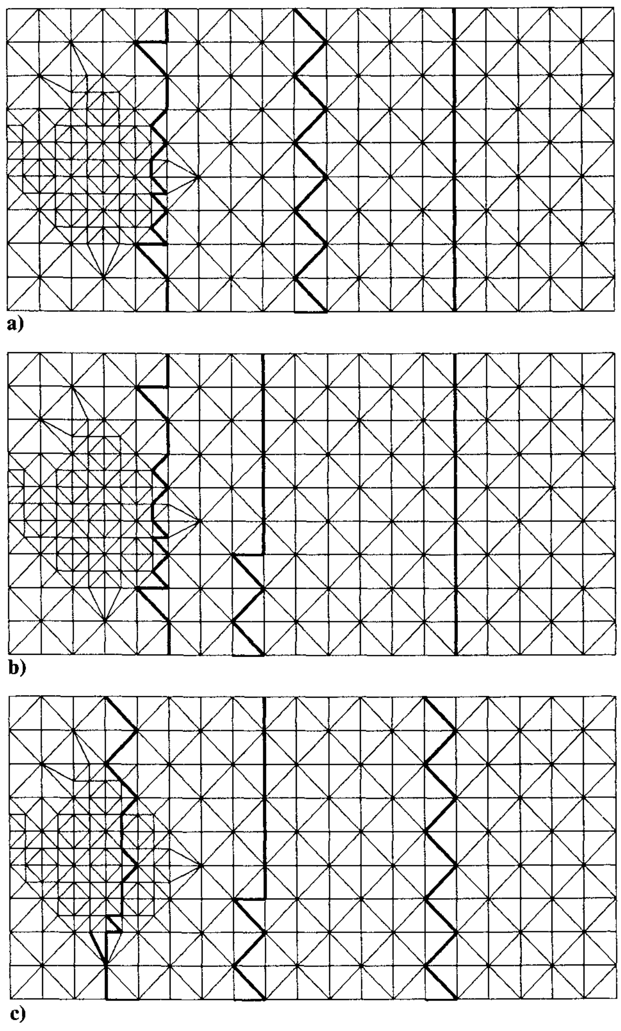


Fig. 2 Illustration of the divide-and-conquer method for load balancing in case of a strip partitioning: a) original strip partitioning, b) partitions after step 1, and c) final balanced partitions.

processor maintains a list of processors that share it, as well as the ID of that entity on each of those processors. This enables two processors that share a given entity to communicate with each other when data are to be transferred through that entity.

Processors that share at least one face with a given processor are said to be face adjacent to that processor. It is useful to define a partition communication graph (PCG) to represent face adjacency of partitions in the system. A PCG is obtained by having one vertex for every partition and an edge between two vertices if and only if they share at least one face between them.

III. Parallel Dynamic Load Balancing

The initial grid-partitioning algorithm generates subdomains with an equal number of grid cells in each of them. The grid is adapted dynamically, which creates an imbalance as new cells are introduced within processor partitions. The problem of eliminating this imbalance consists of two independent subproblems. One concerns the identification of the processors who need to exchange cells with their face-adjacent neighbors along with the number of cells to be exchanged. This is termed as the higher or global level load-balancing strategy and is entirely independent of *how* the cells are actually exchanged. The second problem concerns the actual modus operandi of the exchange of cells between any two face-adjacent processors, including the updating of the pertinent data structures. In the present work, algorithms for solution of the aforementioned subproblems are based on the following fundamental concepts.

1) Divide and conquer: The high-level load-balancing strategy is based on the divide-and-conquer technique wherein the global problem involving all of the processors in the system is efficiently split into two similar, independent problems, each of which involves only half the total number of processors in the system. These two problems are then recursively solved in exactly the same fashion. The recursion is terminated when the problem size reduces to two, in which case the processors simply balance the sum of their individual loads by exchanging grid cells across their common boundary.

2) Local migration: The method for the actual exchange of cells between face-adjacent processors is based on the concept of local migration. A cell that belongs to a given processor P_1 and has at least one face on the interpartition boundary with another processor P_2, can be sent to P_2 by deleting it from the grid data structure for P_1 and adding it to the grid data structure for processor P_2. The previous process can also be looked upon as *movement* of the interpartition boundary toward processor P_1. Several such cells can be migrated together. This provides a mechanism for exchange of cells between face-adjacent processors and can be used for dynamic redistribution of the work load.

Each of the previous two approaches has inherent parallelism that effectively contributes to the performance of the overall dynamic load-balancing algorithm. The divide-and-conquer technique exhibits a "macroparallelism" because of the fact that a single global problem is successively reduced to two "less global" problems that are essentially independent of each other. Further, these two problems are assigned to different groups of processors and can be solved fully in parallel without any additional communication overhead. On the other hand, the idea of local migration is local by definition. Because of this, multiple cell exchanges between different processor pairs can proceed simultaneously. This can be termed as "microparallelism" or parallelism at a lower level.

The efficiency of the parallel dynamic load-balancing algorithm is a direct result of the previous two types of parallelisms inherent in the algorithms for global load-balancing strategy and for exchange of cells among face-adjacent processors. The two concepts are now discussed in more detail.

A. Global Load-Balancing Strategy: Divide and Cconquer

The top-down divide-and-conquer method consists of $\log_2 P$ steps with P being the number of processors. The number of steps remains the same irrespective of the amount of imbalance and its distribution across the processors. The algorithm is thus deterministic, and an analysis of its efficiency in terms of execution can be made.

The technique is applied by dividing the processors in the system into two disjoint groups based on their processor IDs. The cumulative load on the two groups is then balanced by migrating cells from one group to the other. The group with the larger cumulative number of cells is called the *sender*

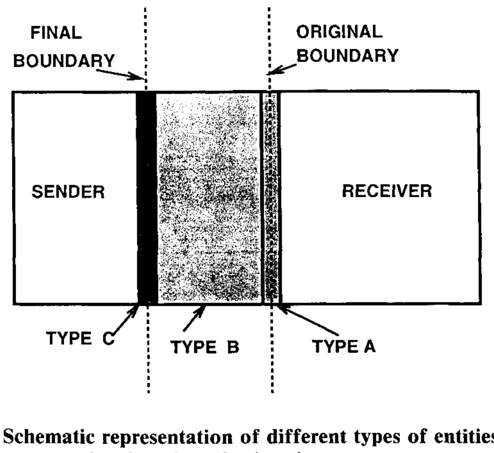


Fig. 3 Schematic representation of different types of entities on the sender processor involved in cell migration.

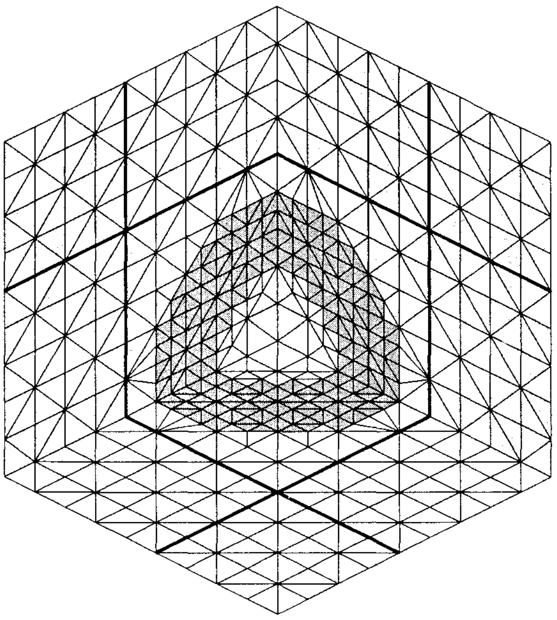


Fig. 4 Surface plot of the channel grid showing adaptation in the vicinity of a spherical surface, as well as the initial interpartition boundaries.

group, whereas the other is called the *receiver group*. Processors in each group that are face-adjacent to at least one processor in the other group are involved in the actual migration of cells and are called *candidate processors*. A candidate progressor in the sender group migrates cells to its face-adjacent candidate processor in the receiver group.

The number of cells migrated by each candidate processor in the sender gorup is given by the following relation:

$$Mig_i = \left(\frac{N_i}{N_{\text{total}}}\right) * Mig_{\text{total}}$$

where Mig_i is the number of cells to be migrated by the ith candidate processor in the sender group (P_i), N_i is the total number of cells on processor P_i, Mig_{total} is the total number of cells to be migrated, and N_{total} is the cumulative number of cells on all candidate processors of the sender group.

As a result of this step, exactly half of the total system load is allocated to each one of these groups. The global load-balancing problem is now reduced to two similar, mutually independent problems. Each of these subproblems is then further divided into two smaller problems by applying the same technique that was applied in the case of the initial global problem. Subsequent steps apply the same technique to successively smaller subgroups. The algorithm ends when all subgroups have exactly one processor. At this point, the load on the system is uniformly distributed across all of the processors.

Initially the size of the group is P and gets divided by two at each step of the algorithm. Therefore, the algorithm takes a total of $\log_2 P$ steps to complete. Figure 1 shows a schematic of the divide-and-conquer technique for a four-processor case. The processors are numbered 0, 1, 2, and 3 and are represented by the PCG defined earlier. A side of the rectangle represents face adjacency of the two partitions it joins. The load on a processor is represented by integers assigned to corners of the rectangles. Step 1 assigns processors 0 and 3 to the sender group and 1 and 2 to the receiver group. The cumulative load on these groups is then balanced by migrating cells from processor 0 to processor 1 and processor 3 to processor 2. Two cells are migrated by each sender processor, which allocates exactly half the total load in the system to each of the two groups. Step 2 then balances the load *within* each group by migrating cells from processor 0 to processor 3 and processor 1 to processor 2. In this case, the load is balanced in a total of $\log_2 P = 2$ steps. It can be noted that all of the local cell migrations that are performed during a single step are performed entirely in parallel.

Figure 2 illustrates the load-balancing algorithm as applied to a "strip" partitioning case of a three-dimensional channel grid discretized with tetrahedra. The grid is cut into four strips along the x axis. The load on the partitions is unbalanced due to adaptation around a source located within the leftmost partition. Figure 2 shows cross sections of the grid corresponding to each step of the load-balancing process. Step 1 of the algorithm migrates cells across the middle partition so as to balance the load across the two halves of the grid defined by it. This causes the middle partition to move to the left, whereas the other two partitions are unchanged. Step 2 migrates cells across the other two partitions in parallel. This results in the load being equally distributed across all four partitions.

B. Local Migration

The local migration algorithm is invoked at every step of the divide-and-conquer process following determination of the candidate processors in the sender and receiver groups, as well as the number of cells to be exchanged between each pair. During this phase, all candidate processors in the sender group send the required number of cells to their respective receiver processors in parallel. Additional overhead is incurred to adjust inter-partition data structures so as to accurately reflect the change in the assignment of cells to partitions. This overhead is primarily due to the fact that all of the processors neighboring a sender processor have to update their data structures as well. This needs to be done to reflect the fact that some entities have been migrated from the sender to the corresponding receiver and are hence no longer shared with the sender processor. The algorithm consists of two steps.

1) Cell-designation step: The sender processor decides which cells within its partition are to be migrated to the receiver processor. This will be termed cell designation in the following. The designated cells have to be physically contiguous so as to maintain a well-defined interprocessor boundary after migration. Also, the number of designated cells must equal the number of cells to be migrated.

2) The sender processor deletes the designated cells and the appropriate faces, edges, and nodes from its partition data structure. It then composes a message consisting of information about the deleted cells and sends it over to the receiver processor. The receiver adds the cells, faces, edges, and nodes to its partition. Both the sender and the receiver then update the interpartition data structures to reflect the change in the interpartition boundary.

Cell-Designation Strategy

The method employed by the sender processor to decide which cells to migrate has important implications with regard to the length and nature of the communication boundary between the two processors after migration. Two different techniques are considered here.

The *grid-connectivity-based* cell-designation strategy starts with cells that have at least one face on the initial interprocessor boundary. It then designates cells that have at least one face in common with previously designated cells. This is continued until the required number of cells has been designated. The algorithm consists of the following steps:

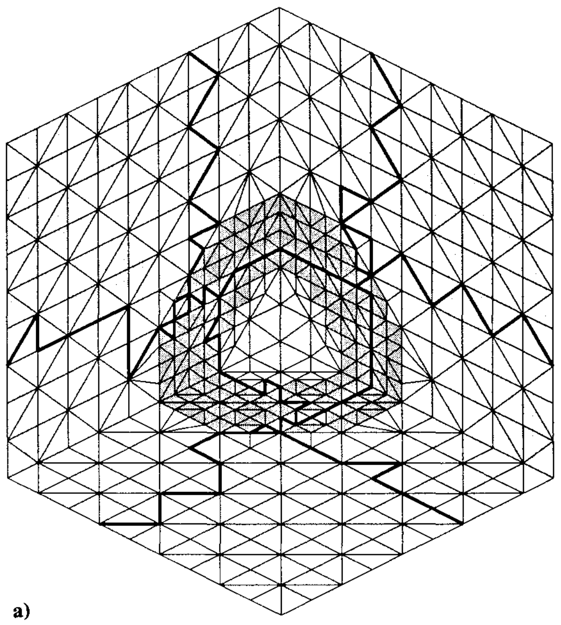
1) Mark all faces that are on the interprocessor boundary.

2) For every cell, if not designated already, designate it if at least one of its faces is marked. Mark all of the faces of this cell.

3) Repeat step 2 until the required number of cells is designated.

The designation of cells is done in "layers" starting from the outermost layer of cells. This layer contains those that have at least one face on the interpartition boundary between the two processors. The distance through which the boundary moves in on the sender side is determined by the size of the cells as well as the connectivity of the unstructured grid. In adapted regions, several layers of cells occupy a very small physical space, whereas coarser cells occupy a relatively larger volume. However, since there is no logical distinction between an adapted and a coarse cell, the connectivity-based designation algorithm is unable to adjust to this difference in size, thus leading to jagged and long boundaries. Furthermore, since there is no spatial orientation of cells in an unstructured grid, the designation algorithm has no control over the shape or structure of the interpartition boundaries. This can lead to a partition belonging to a single processor being physically split into two disjoint subpartitions, especially in the wake of repeated cell migrations.

The *coordinate-based* designation algorithm seeks to alleviate the problems with the *connectivity-based* algorithm by designating cells based on spatial regions instead of grid layers. Thus, all cells that have their centroids within a particular region in three-dimensional space are marked for migration. This region is typically defined to be adjoining the interprocessor boundary. However, the number of cells within a given region is not known a priori. As a result, the width of this region has to be determined by a trial-and-error approach so that it contains the required number of cells. Although this results in extra overhead, the significant advantage offered by this approach is that the length and form of the interprocessor boundary can be maintained even under migration. This is very important, especially when multiple migrations are to be effected in a single execution.



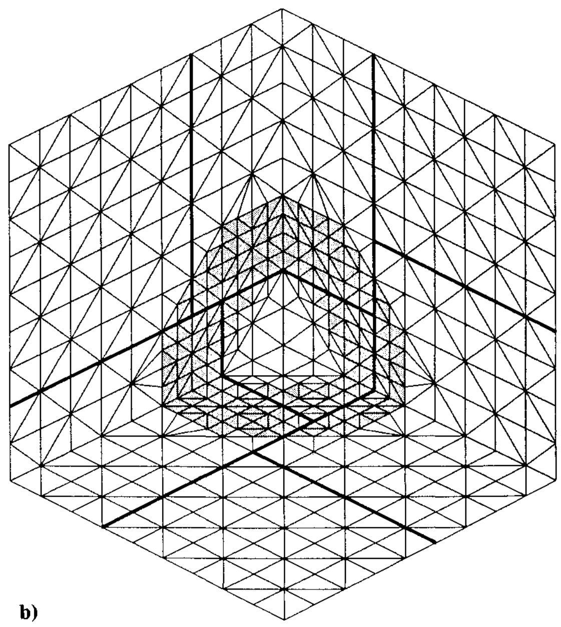


Fig. 5 Surface plot of channel grid showing interpartition boundaries after load balancing using a) the connectivity-based cell-designation algorithm and b) the coordinate-based cell-designation algorithm.

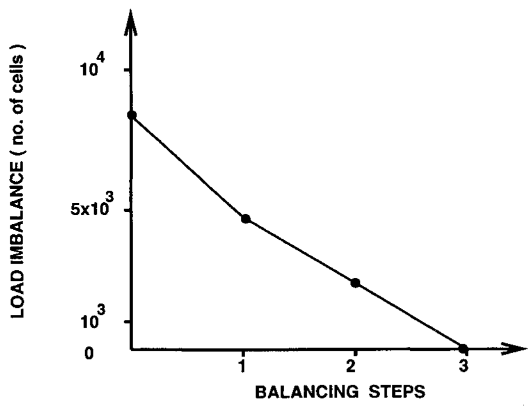


Fig. 6 Decrease in imbalance in the system after each step of the divide-and-conquer algorithm. Imbalance is defined as the difference in the number of cells between the most and least heavily loaded processors (case of adapted grid for the spherical wave).

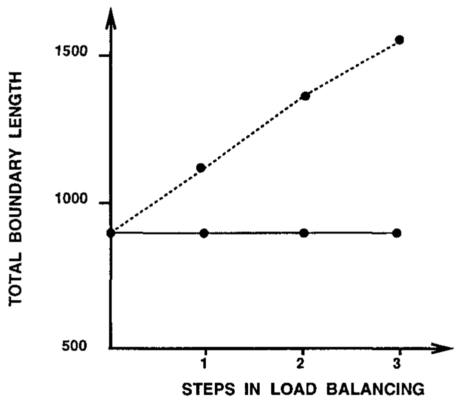


Fig. 7 Comparison of the length of interpartition boundaries after successive migrations using connectivity-based (dotted line) and coordinate-based (solid line) cell-designation algorithms (case of adapted grid for the spherical wave).

Cell Migration Algorithm

Once the cells to be migrated have been designated, the sender processor has to delete these cells from its partition along with the appropriate faces, edges, and nodes and send a message to the receiver processor, which then adds the cells to its partition. Both processors also update the data structures pertaining to the interprocessor boundary.

The entities on the sender processor can be classified as follows.

Type A entities are those that are currently on the boundary between the sender and the receiver and that will be exclusive to the receiver after the migration.

Type B entities are those that are currently exclusive to the sender processor and that will be exclusive to the receiver processor after the migration.

Type C entities are those that are currently exclusive to the sender but will be shared by the sender and the receiver after the migration.

Figure 3 illustrates the three kinds of entities on the sender processor. The sender algorithm consists of the following steps:

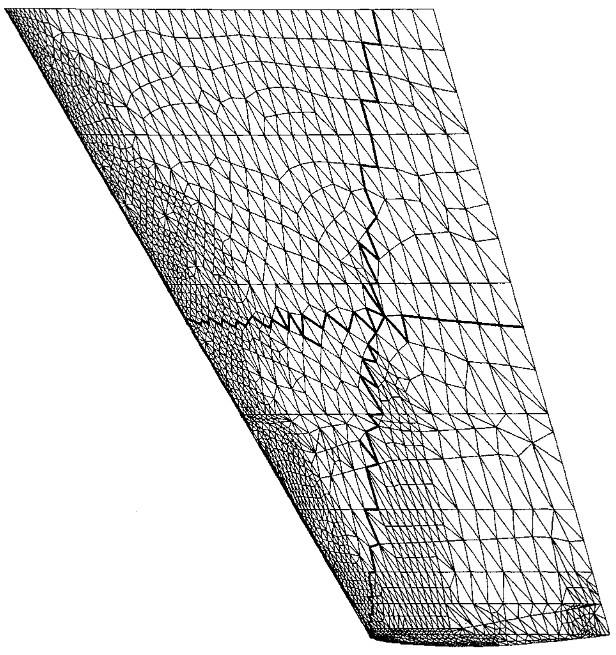


Fig. 8 Surface plot showing the adapted ONERA M6 wing grid before load balancing. Thick lines denote partition boundaries.

1) Delete all type A entities from the grid data structure.

2) Delete all type B entities from the grid data structure.

3) Update the interpartition boundary to incorporate all type C entities.

4) Send all type A, B, and C entities to the receiver.

The receiver algorithm consists of the following steps.

1) Update the interpartition data structure for all type A entities that are no longer to be shared with the sender.

2) Add the type B entities sent by the sender.

3) Add all type C entities to the partition and update the interpartition data structures.

IV. Applications

A. Adapted Grid for a Spherical Wave

We illustrate the process of cell migration in three dimensions through an example of a spherical wave propagating in a three-dimensional channel. The channel is discretized with tetrahedral elements. Grid adaptation is applied to resolve the surface of the wave. Figure 4 shows one level of a locally embedded grid on three of the six walls of the channel. The grid is divided into eight partitions by cutting it into two equal parts along each of the three x, y, and z directions. Adaptation within the vicinity of the surface of the wave causes a load imbalance among the processors assigned to the eight partitions. This load imbalance is eliminated using the parallel dynamic load-balancing algorithm.

Figure 5a shows the channel after the balancing process in which the connectivity-based cell-designation algorithm has been used. It is observed that the resulting interpartition boundaries are "jagged," non-uniform, and longer than the initial regular boundaries. Furthermore, the movement of the boundary in regions of adaptation is less than its movement in regions of little adaptation. This also causes a "skewness" in the boundary leading to its irregular shape. Figure 5b shows the channel after the balancing process in which the coordinate-based cell-designation algorithm has been used. It can be seen that the interpartition boundaries retain their original regular shape and structure. Also, there is no appreciable increase in their overall length.

Figure 6 shows the variation of the total imbalance over all of the processors with the steps in the load-balancing algorithm. The imbalance is calculated as the difference between the number of cells on the most and least loaded processors. It is observed that the total imbalance in the system is eliminated after exactly $\log_2 P = 3$ steps. Figure 7 shows variation of the cumulative length of interprocessor boundaries with the steps in the balancing process. The cumulative length is defined as the total number of shared faces on all processors. The broken line depicts the length for successive boundaries when grid-connectivity-based cell designation is used, whereas the solid line shows the successive lengths in the case of a coordinate-based cell-designation algorithm. It is observed that the overall interpartition boundary length increases in the former case, whereas it remains the same in the case of the latter.

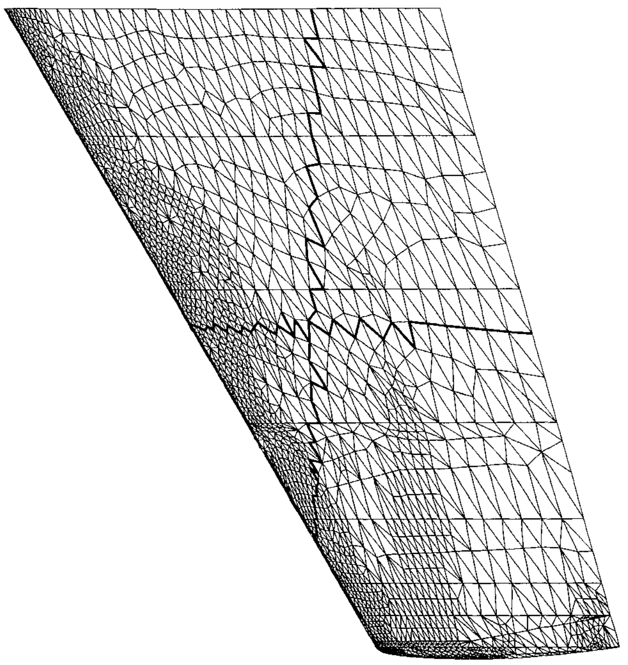


Fig. 9 Surface plot of the ONERA M6 wing grid partitions after the first step of the load-balancing process (coordinate-based cell designation).

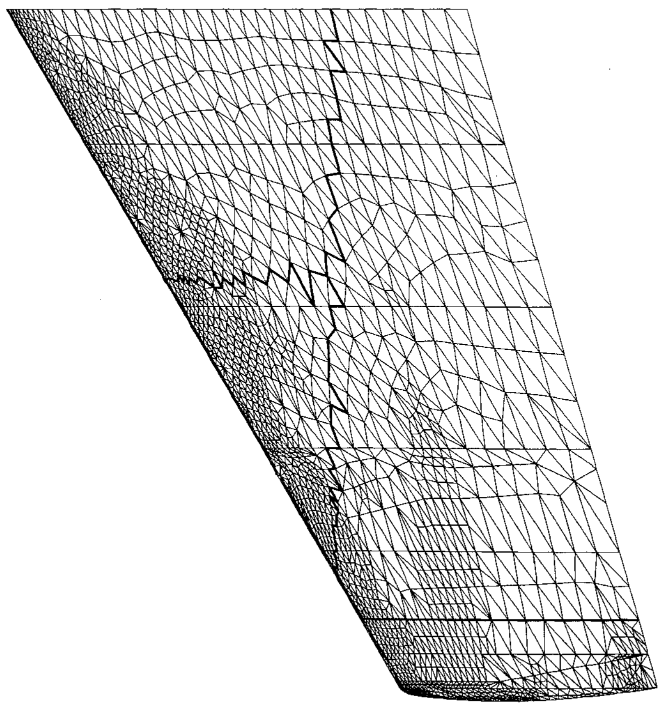


Fig. 10 Surface plot showing the ONERA M6 wing grid partitions after the load balancing is completed (coordinate-based cell designation).

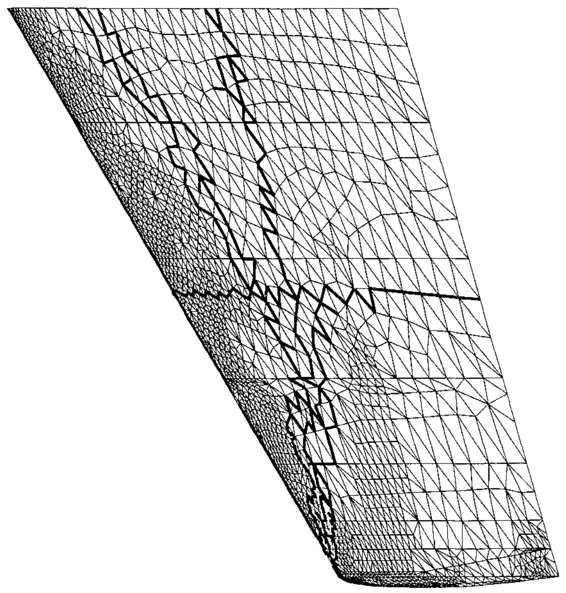


Fig. 11 Surface plot showing the ONERA M6 wing grid partitions after the first step of load balancing using the connectivity-based cell-designation algorithm.

B. Adapted Grid for ONERA M6 Wing

The adapted grid corresponding to transonic flow over an ONERA M6 wing is considered. Figure 8 shows a surface plot of the wing triangulation. It has been adapted once, and locally finer grids are placed at the regions of the leading edge, the tip, and shock waves. Initially, the grid is partitioned among four processors with interpartition wing surface boundaries indicated by the thick lines. Grid adaptation results in a load imbalance among the four processors. The parallel dynamic load-balancing algorithm eliminates this imbalance by rearranging the interpartition boundaries in a manner so as to equalize the number of cells in each partition. Figure 9 shows the surface plot after one step of the load-balancing process. At this stage, the load has been balanced across the "vertical" partition boundary along the span of the wing. It is observed that the interpartition boundary has moved to the left. In the second step, the load is balanced within the two halves on either side of the "vertical" partition. The surface plot for the balanced grid is shown in Fig. 10. It can be seen that the partition boundaries move toward the adapted zones to counteract the imbalance. It should also be noted that the length of each boundary does not change appreciably due to the balancing. This is due to the fact that the coordinate-based cell-designation algorithm is used during the migration.

Figure 11 shows the surface plot of the wing when the connectivity-based cell-designation algorithm is used. It can be observed that the algorithm causes the interprocessor boundary to "jump" several times, creating the illusion of multiple interpartition boundaries. Because of the unstructured nature of the grid, the intersection of the interpartition boundary and the wing surface does not follow the surface of the wing but skips some cells that have at least one face on the wing's surface. This illustrates the principal disadvantage of the connectivity-based cell-designation algorithm. It is unable to adapt to the unstructured nature of the grid and hence leads to jagged and poorly formed interprocessor boundaries.

V. Parallel Implementation on the Intel iPSC/860—Timing Results

The Intel iPSC/860 is a multiple instruction multiple data stream (MIMD) parallel computer with 128 processors. Each node is composed of an Intel i860 microprocessor with a clock rate of 40 MHz, 8 Mbytes of memory, and a direct connect module (DCM) that handles communication. 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 parallel dynamic load-balancing algorithm uses asynchronous sends and receives to minimize the overhead of the intermediate synchronization steps. This is principally due to the fact that asynchronous messages get buffered, which enables the receiver to proceed with its computation until the time the message is really required.

The migration algorithm is implemented using asynchronous sends and receives as are the messages for updating the shared data structures. The migration algorithm itself is slightly uneven in terms of work load due to the fact that processors not involved in the migration have a lesser work load than the actual senders and receivers. This is, however, countered by the fact that all processors demarcated as senders during a single step can execute the algorithm completely in parallel. Furthermore, the local migration algorithm has been designed in such a way that, for any number of cells to be migrated from one processor to another, only one message has to be sent by the sending processor to the receiving processor. Thus each individual migration is quite cost efficient because the high startup overhead associated with a message is incurred only once per migration.

We consider a channel grid partitioned into strips with one strip being assigned to each processor. Figure 12 compares the execution times for three different load-balancing algorithms for different numbers of processors P. Algorithm A is a sequential algorithm in which all of the interprocessor cell migrations at each step of the divide-and-conquer algorithm are done sequentially one after the other. Algorithm B is the developed parallel load-balancing technique where all cell migrations in a given step of the divide-and-conquer approach are carried out in parallel. Algorithm C is a fictitious "ideal" parallel algorithm. This algorithm takes the same amount of time for load balancing irrespective of the number of processors used. It can be seen that the parallel load-balancing algorithm, although not optimal, exhibits a significant improvement over the sequential algorithm. The curve for algorithm B corresponds to actual timings for the load balancer obtained on the iPSC system. The timings for algorithms A and C are obtained by estimation based on the timings for algorithm B.

The difference in performance of algorithms A and B can be explained in terms of the time each one takes to execute a step of the divide-and-conquer approach. Since algorithm A exe-

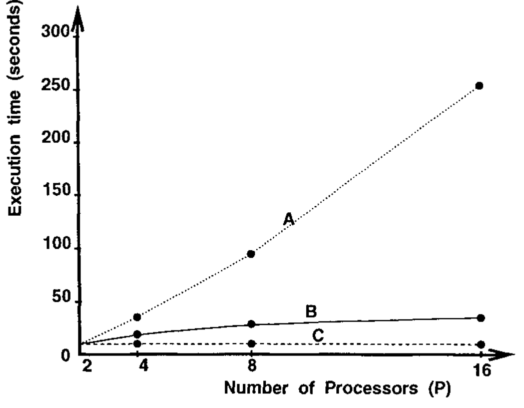


Fig. 12 Variation of execution time for the parallel load-balancing algorithm with the number of processors in the system. Comparison with sequential and ideal parallel executions (A: sequential, B: present, C: ideal).

a)

Phase \ # of P	2	4	8	16	32
Cell Designation	1.63	2.10	2.79	2.83	3.02
Cell Migration	5.68	5.52	5.79	6.52	7.11

b)

Phase \ # of P	2	4	8	16	32
Cell Designation	1.73	2.22	1.91	2.34	2.59
Cell Migration	6.76	8.41	11.49	14.53	17.97

Fig. 13 Tables of timings (in seconds) for the cell-designation and cell-migration algorithms for different number of processors in the system for the channel grid: a) corresponds to a strip partitioning and b) corresponds to an all-round partitioning.

cutes all cell migrations at every step sequentially, the time taken for a step in this case is the sum of the time for all of the cell migrations. Now, the kth step of the divide-and-conquer procedure consists of $2^{(k-1)}$ cell migrations. Hence, if T is the average time per migration, then the total time for algorithm A can be written as

$$\begin{aligned} T_{\text{total}} &= T_1 + T_2 + \cdots + T_{\log_2 P} \\ &= 1(T) + 2(T) + 4(T) + \cdots + p/2(T) \\ &= (P-1)T \end{aligned}$$

Thus the execution time for the sequential algorithm exhibits a linear variation with the number of processors. The parallel load-balancing algorithm, however, executes all of the migrations at a particular step in parallel. Hence, the execution time for this algorithm can be written as

$$\begin{aligned} T_{\text{total}} &= T + T + T + \cdots + T \quad (\log_2 P \text{ times}) \\ &= (\log_2 P)T \end{aligned}$$

Thus, the execution time for the parallel load-balancing algorithm exhibits a logarithmic variation with the number of processors in the system. This difference in performance is particularly significant from the point of view of scalability. The parallel load-balancing algorithm exhibits only logarithmic degradation in performance with increasing processors as compared with the linear degradation of the sequential algorithm.

Figure 13 gives timings in seconds for the cell-designation and cell-migration phases of the local migration algorithm for a channel grid for different numbers of processors. Figure 13a provides timings corresponding to the strip partitioning, whereas Fig. 13b corresponds to an all-round partitioning of the channel along all three coordinate axes. For a given number of processors P the timings correspond to the last step of the divide-and-conquer algorithm. During this step, exactly $P/2$ processors simultaneously exchange cells with the other $P/2$ processors in the system. The timing for such a step is the maximum time taken by any processor to execute that step.

It can be seen that in the case of the strip partitioning there is only a small increase in the execution time per step with an increasing number of processors. This illustrates the degree of parallelism in the local migration algorithm. An increase in the number of processors that exchange cells does not significantly increase the execution time of a load-balancing step. The slight increase is due to the fact that the partitions are assigned to processors in a random fashion and not according to the underlying hypercube architecture. This causes some contention for communication links as the number of processors involved in the migration process is increased.

In the case of the all-round partitioning, there is an extra overhead associated with updating the data structures on processors that are adjacent to the sender and receiver processors in a given local migration step. This overhead increases in direct proportion to the number of processors involved in the migration of cells. As a result, there is a bigger increase in execution times as additional processors get involved in the load-balancing process.

Timings for the cell-designation step do not manifest significant variation due to the fact that the cell-designation algorithm does not involve any communication among processors and is thus executed entirely in parallel by all sender processors.

VI. Summary

A new approach to dynamic load balancing in cases of adaptive, three-dimensional unstructured grids has been developed. The algorithm is based on the concepts of divide-and-conquer and local migration, which makes it eminently suitable for parallelization on a partitioned memory MIMD system.

The initial computational domain is partitioned among the available processors by a partitioning algorithm based on the orthogonal recursive bisection method. Different ways of partitioning have been presented and compared.

Local migration includes the subproblem of cell designation wherein a processor decides which of the grid cells within its partition are to be exchanged during the migration process. Two approaches for solving this problem have been examined. Coordinate-based cell designation has been found to maintain the shape and size of the initial interpartition boundaries after migration.

The algorithms have been implemented on an Intel iPSC/860 system. Scalability of the parallel dynamic balancing algorithm has been illustrated through comparison with the corresponding sequential algorithm and through timing of the different phases of the balancing process.

Acknowledgments

This work was supported by National Science Foundation through Grant ASC-9111540 and monitored by M. Patrick. It was also supported by DARPA Grant DABT 63-92-0042 and monitored by Gil Weigand. Computing time on the iPSC/860 was provided by the NAS Division of the NASA Ames Research Center.

References

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

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

[3]Kallinderis, Y. G., 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 Vijayan, P., "An Adaptive Refinement-Coarsening Scheme for 3-D Unstructured Meshes," *AIAA Journal,* Vol. 31, No. 8, 1993, pp. 1440–1447.

[5]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.

[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]Fox, G. C., *Numerical Algorithms for Modern Parallel Computers,* edited by M. Schultz, Springer-Verlag, Berlin, Germany, 1988.

[8]Hammond, S., and Barth, T. J., "Efficient Massively Parallel Euler Solver for 2-D Unstructured Grids," *AIAA Journal,* Vol. 30, No. 4, 1992, pp. 947–952.

[9]Venkatakrishnan, V., "A MIMD Implementation of a Parallel Euler Solver for Unstructured Grids," *Journal of Supercomputing,* Vol. 6, No. 1, 1992, pp. 117–137.

[10]Nicol, D. M., and Saltz, J., "Principles for Problem Aggregation and Assignment in Medium Scale Multiprocessors," ICASE Rept. 87-39, Hampton, VA, Sept. 1987.

[11]Williams, R. D., "Performance of Dynamic Load Balancing Algorithms for Unstructured Mesh Calculations," California Inst. of Technology, Concurrent Computation Rept. C3P 913, Pasadena, CA, June 1990.

[12]Otten, R. H. J. M., and Van Ginneken, L. P. P. P., *The Annealing Algorithm,* Kluwer Academic, Boston, MA, 1989.

[13]Barnes, E. R., "An Algorithm for Partitioning the Nodes of a Graph," *SIAM Journal of Alg. Disc. Meth.,* Vol. 3, March 1982, p. 541.

[14]Pothen, A., Simon, H. D., and Liou, K.-P., "Partitioning Sparse Matrices with Eigenvectors of Graphs," *SIAM Journal of Mathematical Anal. Applications,* Vol. 11, April 1990, pp. 430–452.