JOURNAL OF COMPUTATIONAL PHYSICS 91, 94–109 (1990)

Efficient Data Structures for Adaptive Remeshing with the FEM

H. H. DANNELONGUE AND P. A. TANGUY

Department of Chemical Engineering & CERSIM, Université Laval, Québec, Canada G1K 7P4

Received October 25, 1988; revised August 8, 1989

The meshing and interpolation problems encountered during adaptive remeshing on unstructured finite element grids are analyzed. Both problems are solved independently in 2D and 3D with cost efficient data structures issued from recent advances in computational geometry. Implementation algorithms and timings are presented in 2D. The performance of commonly used schemes is also reviewed. © 1990 Academic Press, Inc.

1. INTRODUCTION

The accurate solution of industrial fluid flow problems with the finite element method (FEM) requires large amounts of memory space and computer time. The accuracy of the solution can be improved without increasing the computational cost by concentrating the elements where they are most needed [1]. This is termed refinement when a subdivision of an initially coarse mesh is performed, while is it called remeshing when an entirely new mesh is generated. In the optimal variation of this scheme, called adaptive, the mesh spacing itself is computed. The memory space required for the adaptive remeshing of a problem is related to the number of elements in the grid and is much smaller than the total size of the finite element equation system. The memory constraint for adaptive procedures is therefore only second to the execution time constraint. Hereafter emphasis is put on reducing the execution time.

Once the finite element problem is solved on the initial user defined mesh, an error estimate is computed at each node. The error level to be reached is chosen, and the local mesh size required to achieve it is computed at each node using the error estimate [2]. The new mesh is then built, and an initial solution is needed at the new nodes to restart the solution procedure if it is iterative. The solution on the initial mesh is an adequate guess for this purpose, provided it is interpolated on the new mesh. Therefore, both a fast meshing algorithm and a fast interpolation scheme between meshes are needed for the implementation of adaptive schemes. For steady flows, only one or two remeshing iterations lead to convergence using a local error indicator, and global remeshing is suitable [3]. For time dependent or front tracking problems, truly local refinement or patch remeshing is best suited [4]. The

0021-9991/90 $3.00
Copyright © 1990 by Academic Press, Inc.
All rights of reproduction in any form reserved.

chosen algorithms and the associated data structures should therefore allow for remeshing of the total domain or part of it.

Data structures devised to answer both the meshing and the interpolation problem have been presented in the finite element literature. They are either modifications of classical structures [2, 3], or original structures, as reviewed in [4]. These dual-purpose structures are based on the partition of space in nested square cells.

The novelty of the present paper is to propose to decouple the meshing and interpolation solutions in order to use data structures of proven performance directly issued from recent advances in computational geometry in 2D and 3D [5–10]. The performance of these structures is governed by the number of points in the mesh rather than by grid point spacing. It should be noted that the full benefit of these structures can only be obtained with a programming language that supports true dynamic allocation and recursion [11]. The present work was carried out using the C language.

2. Current Adaptive Structures

2.1. *Data Structures and Optimality*

The most elementary one-dimensional data structure is a linear list, in which all points, or numbers, are stored sequentially. Storing N numbers in such a sequential list takes up minimal memory space, i.e., N, and searching for an item also costs $O(N)$. It is possible to improve the search performance by using a more complex structure, the classical binary tree (Fig. 1a). This tree allows for insertion, deletion, or search of any of its elements in time $O(\log_2 N)$, and total storage is still only $O(N)$. In what follows, $\log N$ is used to mean $\log_2 N$, since the order is defined to a multiplicative constant. The computational cost of the tree search, or time performance, is equal to the distance between the root and the farthest leaf of the tree, also called the depth of the tree. Thus, the cost of search in a binary tree varies with the order of insertion of the points (Fig. 1b), degenerating into a worst-case performance of $O(N)$ when points are inserted in order. The tree is then merely a linear list of depth N. Therefore, the true criterion of a structure's suitability is its worst-case performance, not the average one, and time complexities given in this paper are worst-case bounds. The worst-case performance of the simple binary tree can be made logarithmic by balancing it, turning it into an AVL tree [11]. No other structure can do better, and the balanced binary tree is the optimal structure for one-dimensional search problems, both for time and space.

In 2D and 3D, $O(\log N)$ in time and $O(N)$ in space have not been achieved simultaneously to this day, and the best performance in time is often achieved at the cost of redundancy in storage. Optimality is not always defined, and solutions have to be compared to present day best cases rather than absolute yardsticks. In any case, the chosen solutions should be of known performance. Furthermore,

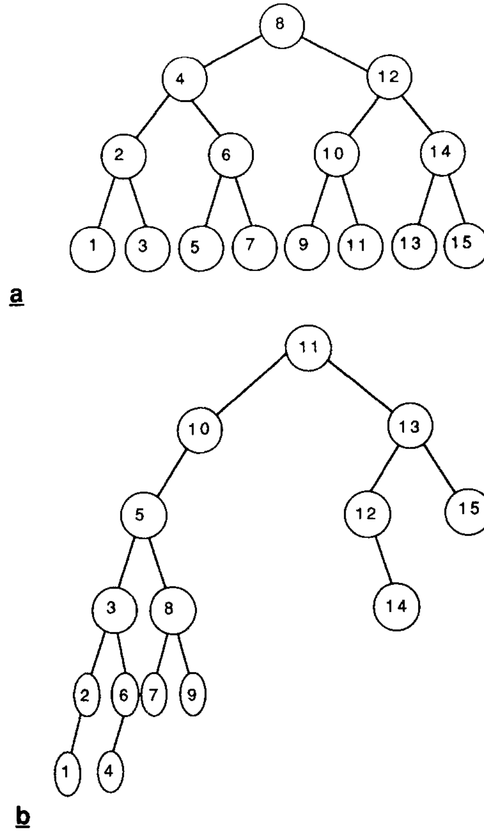

FIG. 1. (a) Balanced binary tree of depth 3 obtained by insertion of 15 data points in optimal order: 8, 4, 12, 2, 6, 10, 14, 1, 3, 5, 7, 9, 11, 13, 15; (b) Binary tree of greater depth obtained by insertion of the same data points in random order: 11, 10, 13, 5, 12, 15, 3, 8, 14, 2, 6, 7, 9, 1, 4.

intricate structures that are mere embodiments of algorithms should be avoided and a compromise is to be found between programming complexity and execution time.

A second restriction to keep in mind when evaluating structures is that the use of recursion and pointer arithmetics in order to ensure asymptotic optimality leads to computational overhead. This in fact deteriorates the performance of the algorithms for small data sets, as shown in Fig. 2, where an efficient sorting algorithm, shellsort, is compared to a more sophisticated one, quicksort, that uses recursion [11]. It can be seen that while the advantage of the recursive solution is obvious for N larger than 100, there is no gain to speak of for small N.

2.2. *Meshing*

The FEM yields an approximate solution of a set of differential equations on a bounded domain. The solution is computed on a discrete set of points. These points

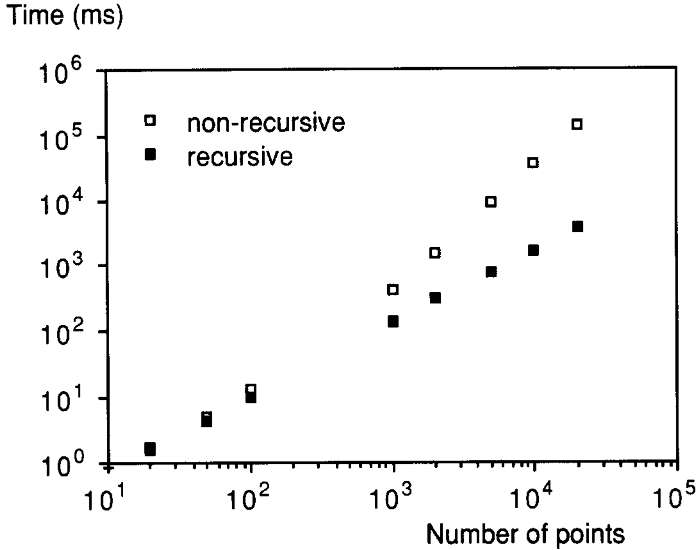

FIG. 2. The gain in execution time obtained by the use of recursion for sorting numbers only becomes significant for large data sets.

form a mesh of elements, called the discretization of the domain. The elements must have a good aspect ratio in order to produce an accurate FEM solution [12]. In the case of triangular elements, this is equivalent to stating that the smallest angle of all triangles must be bounded away from zero.

The meshing of complex geometrical features is best carried out using triangles in 2D and tetrahedra in 3D. Being d-simplices, they can fill any polygon in 2D or polyhedron in 3D, are always convex and never degenerate. Their properties with respect to the convergence of the finite element method are well known [12], and they also generate minimum matrix bandwidth. For these reasons, the present work is based on a triangular mesh generator.

The classical method of mesh generation is in two steps: generation of a set of points covering the area to be meshed, then valid triangulation of the domain using these points. In 2D, the N points are usually evenly distributed on a rectangular lattice, and their computation time is $O(1)$. Total execution time of order $O(N \log N)$ has been obtained for the Delaunay triangulation method [13]. The resulting triangulation and the associated tesselation define the zone of influence of each point. This would be an ideal framework for searching for the nearest neighbor of all points [5]. But since the aspect ratios of the individual triangles are not checked during generation, Laplace smoothing [14] is necessary before the mesh can be used for finite element computations, and the essential tesselation property is lost before being used. There is therefore no practical advantage to having a tesselation associated with a finite element mesh.

In 3D, the points are also usually evenly distributed on a rectangular lattice. While a regular triangulation can be obtained in 2D from such a network, it cannot in 3D. Indeed, it is impossible to build a basic cube from several identical tetrahedra. This is a fact well known to crystallographers, who call the basic cube a Bravais lattice [see [8], for instance]. Tetrahedra do not stack up to form a repetitive pattern in 3D like triangles do in 2D. A common scheme implemented in current solid modeling sofware is to divide each basic cube formed by eight points of the network in five tetrahedra, the center one being twice larger than the four others (Fig. 3), that is, not a regular mesh, and the local error is still bound by the largest element.

Furthermore, the points have to be unevenly distributed when a finer mesh is needed close to singularities to keep the error within reasonable bounds. The generation of the set of points itself can be costly in the case of such graded meshes, thus adding another drawback to the two-step method. Lastly, this type of triangulation does not allow for directionality, a key property for front-tracking problems.

A more recent approach to mesh generation is the advancing front method [16, 17]. It can be used for convex or concave polygons, with multiple boundaries, and directionality can be included. In this method, a "greedy" algorithm is used. Points are created one at a time in order to build a new triangle with optimal shape, i.e., equilateral for uniform meshing. Possible intersections with existing edges and point inclusions are checked locally to guarantee a valid triangulation. In order to do so, existing active front points within a given neighborhood of the newly created point have to be found.

Neighbor search in 2D is presently carried out using the quad-tree, introduced to the finite element literature by [18] and used for advancing front mesh generation by [2]. The quad-tree is a way to store a rectangular lattice of $n \times m$ evenly spaced

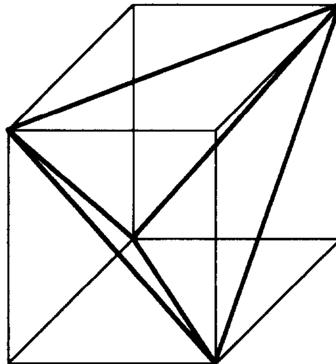

FIG. 3. The classical decomposition of a cube in five tetrahedra leads to a large central tetrahedron, represented with thick lines, surrounded by four small ones.

points. These points are the ends of $(n-1) \times (m-1)$ adjacent segments, which partition the plane in $(n-1) \times (m-1)$ identical cells. The quad-tree was originally used to measure the union of rectangles for VLSI network design. The depth of a quad-tree in the cells of which N random points are stored depends on the minimum spacing between points, rather than on the number of points. In the case of graded meshes, points are unevenly distributed, so the structure is not balanced. Its depth is a logarithmic function of the minimum mesh size, far greater than $O(\log N)$, and the quad-tree does not solve the neighbor query optimally. Furthermore, the search is performed on a quad-tree eventually containing the whole mesh, when in fact the points of interest are only those of the advancing front itself, a small dynamic subset of the mesh. Performances in $O(\log N)$ reported for mesh generation [2] are fortuitous and can be explained by the fact that the structure never contains all N points during meshing. Efficient implementation of the advancing front method requires a specific data structure to tackle this problem.

2.3. *Interpolation*

Interpolation techniques all require the knowledge of the location of the new point in the elements of the initial mesh. The problem is one of point location, on a static structure of N points.

Several data structures containing the points of the initial mesh have been used for this problem. They are based on the partitioning of space in identical square cells and are reviewed in [4]. In addition, the use of the quad-tree was proposed for interpolation [2]. This family of structures containing the mesh points does not directly solve the point location problem. Linked lists have to be grafted to connect neighboring elements, and their computation and storage are inefficient. Costly multiple tree traversal is required. When these structures are created during mesh generation, they have to support insertion and deletion as well, which adds to their complexity. While cost of $(\log_4 N)$ is claimed for interpolation on an unstructured grid using the quad-tree [2], the actual cost is much larger due to the tree's depth and its inadequacy to point location query. The performance of the other structures is similar, since they are based on the same principle, and while they certainly represent a major improvement over linear search, better results can be achieved for the same programming complexity.

An altogether different approach is suggested in [19]. The initial nodal field is first interpolated onto a much finer regular square mesh. Interpolation of any new point is then performed at very low cost on this intermediate mesh rather than on the initial mesh. The cost of interpolation of the initial field on the fine mesh is large, since it is performed with a straightforward algorithm. In fact, the cost of this scheme is superior to the cost of plain interpolation as long as the cardinality of the new finite element mesh is smaller than the number of points in the fine mesh. Since this one needs to be very fine to avoid errors due to the double interpolation, this algorithm is ill-suited to our purpose.

3. Overall Adaptive Strategy

3.1. *Divide and Conquer*

The idea governing our strategy is to improve on the above results by solving meshing and interpolation as two-independent problems. While meshing, the searched set is a dynamic list, the advancing front. On the other hand, interpolation can be identified as a point location problem on a static list. Using a divide and conquer approach, we will present specialized structures to solve each problem separately. Their implementation is a straightforward application of bisection and does not require the handling of extra pointers.

Both data structures are not needed simultaneously, and neither is needed during resolution of the finite element problem, when memory space is in high demand. The advancing front dynamic list is only needed during meshing, and it is empty once the domain is fully meshed. During solution of the finite element problem itself, the N points of the mesh and corresponding elements can be stored in a linear array, minimizing storage. Once the problem is solved and the new adapted mesh is completed, the static data structure containing the N points for the interpolation can be built. It is then discarded before the next iteration, once again saving storage.

3.2. *Range Search*

The meshing neighbor search is a search in a fixed radius disk centered on a point. This is part of a wider class of problems called range search. The radial search can be relaxed to range search in a product of one-dimensional segments, called orthogonal range search. The cost of the latter is lowest of the two [5], and the problem encountered in mesh generation is best treated as an orthogonal range search.

In one dimension, range search on a static set of N points can be accomplished optimally in time $O(\log N)$ using a balanced binary tree in which the points themselves are stored. The tree is traversed twice, once searching for the left end of the range, once for the right end. Points situated between the two paths are then gathered. In d dimensions, a naïve decision tree would lead to the erroneous lower time bound of $O(\log N)$, but the proven bounds are higher. This can be shown qualitatively in 2D using Fig. 4. In a time of $O(\log N)$, one can only tell which points are included in the union of two segments using one binary tree per axis. But the answer to the range search problem is the list of points belonging to the intersection of both segments, a different problem altogether. The extension of the binary tree to d dimensions, the range tree [6], solves this query directly in $O(\log N)^d)$. A range tree is a succession of nested binary trees. Each node of the main tree, corresponding to the x axis, is itself organized as a binary tree, called sub-tree and corresponding to the y axis. In 3D, every node of the sub-tree itself is a tree corresponding to the z axis.

The dynamic multi-dimensional problem is more complex, since the nested struc-

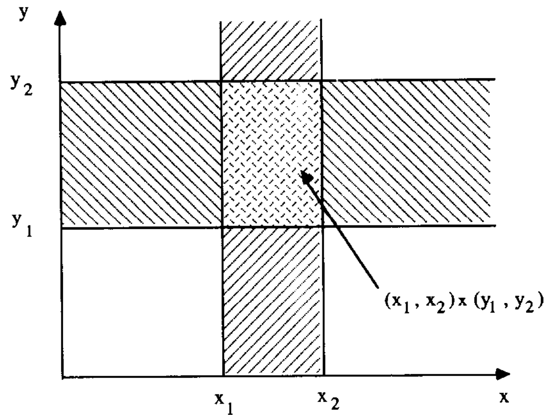

FIG. 4. The union of two segments can be found in $2 \times \log N$, but finding their intersection (cross hatched area) is more costly.

tures must support deletion and insertion. The proven lower worst-case bound for dynamic range search structures is $O(\log N)^d)$, but the structure itself is unwieldy [8]. Another method also claims the same performance [7], but the implementation is again far from simple according to its author.

The dynamic structures associated with optimal performance being so complex, let us recall the warning of Section 2.1 and estimate the size of the lists to be handled. For a large problem comprising 2000 mesh points in a square domain in 2D, the maximum size of the advancing front is typically 300. This is large enough to justify the use of binary structures and to rule out linear lists. On the other hand, if a multi-dimensional tree is used, the number of points stored in each node of the main tree is 87 at the most, and 34 on the average. The large computational effort necessary to update the sub-trees would not be worthwhile for such small lists. Therefore, simpler structures are needed and it was decided to use two independent balanced binary trees, one for each dimension. Each tree can be easily balanced after insertion or deletion, leading to a logarithmic worst-case bound. This combination does not yield a direct answer to the range search, as seen earlier, but rather two lists of coordinates. To answer the range search query, the intersection of the x and y lists is then obtained as follows: the two short lists are first sorted using the sorting routines described above, and their common points are found by going down the list once.

3.3. *Point Location*

The range search problem was solved using a structure containing the points of the mesh themselves. The point location problem associated with interpolation can only be solved efficiently with a structure containing a description of the elements rather than the points themselves [9]. This duality between both problems further

justifies our divide and conquer approach. Recall that we are only interested in the static problem here.

In one dimension, the dual problem is solved directly using a structure in which segments are stored [10]. A skeleton binary tree is first built with the $(N-1)$ segments of unit length formed by N points [Fig. 5a]. Each segment is then stored in the nodes that it fully contains (Fig. 5b). A segment is not stored in a node if it is already stored in the corresponding parent node. The point location performance of such a one dimensional static segment tree is $O(\log N)$. This is true even for unevenly distributed points, in contrast with the results presented in Section 2. It is due to the fact that bisection is not used here to partition the real space in equal halves, but rather to divide the point set in two halves of same cardinality. This is analogous to using the median rather than the mean of a data set for statistical analysis: the crucial step in building balanced binary structures is to divide the data set in two halves at each level, independently of the coordinates of the data points. The segment tree is used here with real numbers, and a mapping between the real space and the integer segment $[1, N]$ is necessary. The coordinates are first ordered and then identified by their position number in the ordered list.

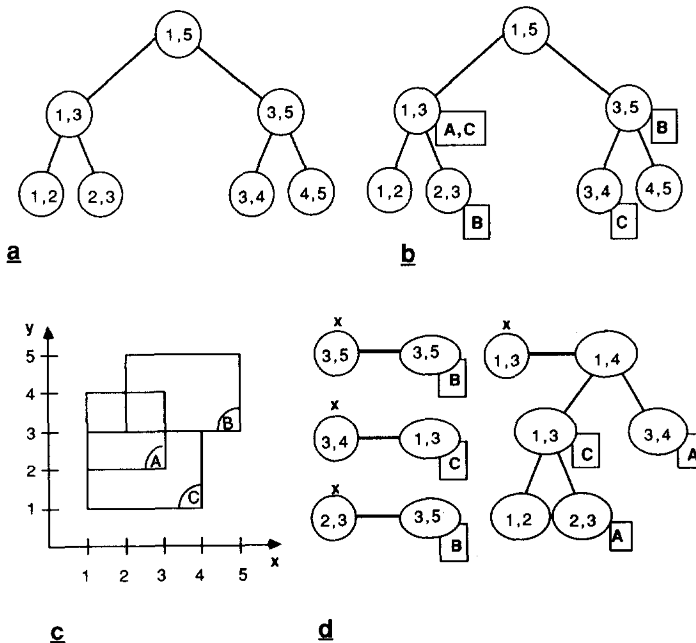

FIG. 5. (a) Skeleton segment tree defined by five points; (b) Skeleton tree loaded with $A=[1,3]$, $B=[2,5]$, $C=[1,4]$; (c) Rectangles $A=[1,3]\times[2,4]$, $B=[2,5]\times[3,5]$, $C=[1,4]\times[1,3]$ generating the above main segment tree by mapping on the x axis; (d) Loaded secondary trees for the y projection of these rectangles; there is one tree per non-empty node of the main tree. Nodes of the x tree are represented as circles, nodes of the y tree as ovals.

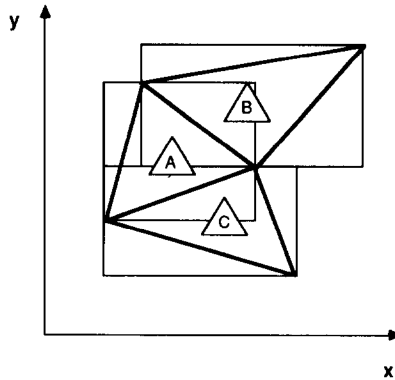

FIG. 6. Triangles corresponding to the rectangles of Fig. 5.

A segment turns into a product of segments in higher dimensions, and the one-dimensional concept can be broadened. The extension of the segment tree to d dimensions has performance of $O(\log N)^d$ [9]. It can be viewed as a segment tree of segment trees. In 2D, the projection of the rectangles on the x axis (Fig. 5c) are stored in the main tree (Fig. 5). Each non-empty node of this main tree contains a list of the corresponding projections on the y axis arranged as a segment tree (Fig. 5d). The total building time for this static structure is $O(N \log N)^d)$. In the case of regular meshes, the mapping from real to integer space removes double values and reduces the size of the set from which the skeleton tree is built, further improving the performance of the structure below its theoretical bound.

The performance of the tree can be improved to $O((\log N)^{d-1})$ by storing more information in the nodes [9]. This is the present day best case for point location, but represents again an intricate modification according to its author. In order to remain with a straightforward and well documented structure [5], the unmodified segment tree was chosen here as an efficient answer to point location queries.

The two-dimensional segment tree is designed to store rectangles with sides parallel to the axes. To use it with unstructured grids, each triangular element is first replaced by the smallest rectilinear rectangle that contains it. Figure 6 shows a set of three triangles associated with the rectangles and trees of Fig. 5. The ordered list of the projection of rectangle vertices on the x axis is mapped onto $[1, N]$ in order to build the main skeleton tree. For each main node, the projection on the y-axis of the Nx rectangle vertices associated with the elements stored in that node is mapped onto $[1, Nx]$ in order to build the secondary tree.

4. MESHING RESULTS

While the data structures chosen above are appropriate for solving both 2D and 3D problems, the implementation presented in this section is strictly 2D.

Implementation in 3D does not require any changes as far as the data structures are concerned.

Advancing front mesh generation has been detailed in [2, 16, 17]. It has to be presented again here in order to highlight the features of the present implementation. The advancing front is a list of n faces, n being small compared to N. Each face is an ordered pair of vertices. The order defines the direction of the inner normal to the face, i.e., the inside of the domain to be meshed. Each face is stored in the two binary trees using the coordinates of its mid-point. Each node of the first tree contains the x coordinate of the mid-point, the key to the ordering of that tree. It also contains the face number, the number of the first vertex of the face, and the memory address pointing to the next nodes (Fig. 7). The second tree contains the corresponding information for the y axis. For a given orthogonal range, a search is conducted on each tree to collect the faces mid-points belonging to each one-dimensional range. This yields points contained in the union of one-dimensional segments. The intersection of the two lists has then to be computed. This post-processing is done on a small subset of the n faces, and the extra cost to the logarithmic performance of the structure is small if the lists are first sorted with an optimal routine (see [11], for instance). The upper bound for a single range search performed during meshing depends on the mesh parameters and topology of the domain. It increases of an unknown but small amount as N increases.

The list of vertices and edges defining the initial boundary can be entered manually or provided by a solid modeler for complex geometries. This boundary may enclose only part of the total domain for partial remeshing. The mesh size is given at each vertex. The number and the length of segments along each edge are interpolated from both end values, thus building the initial generation front. For graded meshes, the mesh size in the interior of the domain is controlled by the vertex mesh size and by additional control points. These allow for refinement in the middle of the domain required for complex flow patterns.

The triangulation proper starts with the first face in the initial front, i.e., the root of the face tree. The coordinates of the point that would form an optimal triangle with that face as a base are computed. This is called the ideal point. Using the binary face trees described above, faces and points located in an orthogonal range

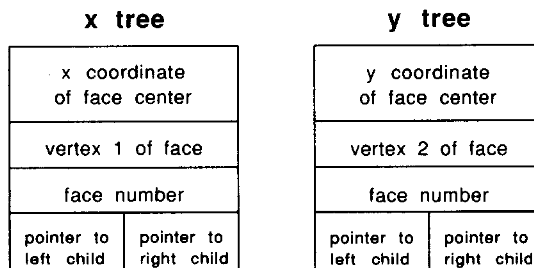

FIG. 7. Nodes of the x face binary tree and y face binary tree used for range searching.

centered on the ideal point are found. The range size is user defined. The points in the range are ordered by increasing distance from the ideal point, and all points beyond a maximum distance are discarded. This distance defines the acceptable deviation from the ideal triangle. In other words, using any of the points in the reduced list as an apex would lead to a triangle with an acceptable aspect ratio. The first point of the reduced list is used to create a new triangle. If this generates no intersection with the advancing front or no inclusion of front points, this triangle is added to the mesh and the advancing front is updated. If not, the remaining points of the reduced list are tested in turn, ending with the ideal point. If no suitable point was found in the list, a new face is chosen in the range and the procedure is restarted.

Once a new triangle is created and added to the connectivity list, the process continues with the first remaining active face in the range. Thus there is no time spent searching for the next face [2], and the resulting meshing pattern is a spiral if the boundaries have been entered in cyclic order in the trees. However, when a face connecting two opposed boundaries is created, the domain to be meshed is naturally divided in two closed sub-domains. The location of that bridge edge is stored and will be used to restart the algorithm once no active face is found in the vicinity of the last created point.

The above algorithm is termed a greedy algorithm since it sacrifices global optimality for local optimality. To check this tendency, the Boolean inclusion criterion was replaced by a more stringent one. To determine if an existing active point is within the triangle to be built, the sine of the angle between each oriented

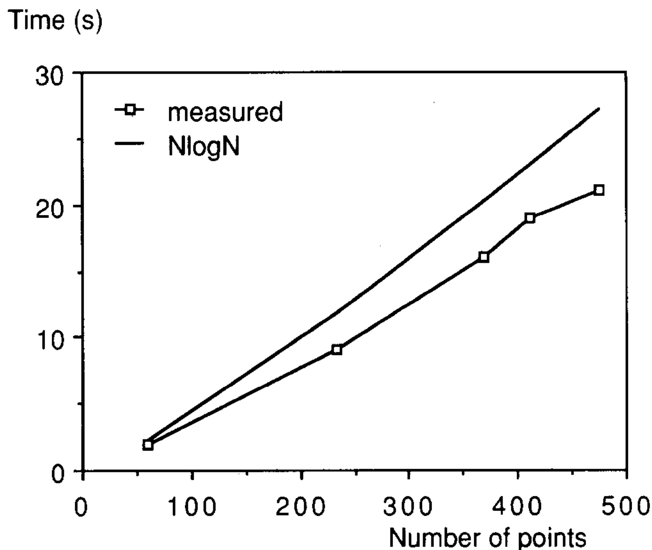

FIG. 8. Execution time of 2D meshing algorithm. Implementation on a 16 MHz 80386 microcomputer in C language.

face of the triangle and the vector joining the origin of that face with the point to be checked is computed. A positive sine means that the point is inside the triangle. Here, the sine is compared to a user-defined negative tolerance instead of zero. This allows for rejection of triangles that would lead to neighbors with very acute angles and increase the local error bound of the solution. With this scheme, there is no need for overall Laplace smoothing of the mesh.

In order to demonstrate the performance of the binary structure used, the meshing time for a sequence of meshes of increasing cardinality is plotted in Fig. 8. It can be seen that total meshing time is roughly linear in N. The scale of the $N \log N$ curve is arbitrary, and it is only presented for visual comparison.

5. Interpolation Results

The implementation presented in this section is again strictly 2D. Implementation in 3D does not require any changes as far as the data structures are concerned.

The static list of points on which the location query has to be answered represents the entire mesh, and is stored in a segment tree built as described above. Given the coordinates of a new point to locate in the existing mesh, the main tree is then just traversed once using bisection as follows. The x coordinate of the new point is compared with the mid-point of each node to determine the direction of branching. The address of all the main nodes thus traversed are gathered. Recall that the node values are integers, and the comparison is actually made with the mapping of these integer values onto real space. The secondary tree of each traversed main node is then visited in the same way using the y coordinate of the new point, and the contents of all nonempty secondary nodes is gathered. This is the list of rectangles containing the new point. It then remains to verify inclusion of the points in the element associated with each rectangle. Only two rectangles are gathered typically, leading with very little algebra to the triangle in which the new point falls.

The average time of execution for the location of a single point in a mesh was chosen as representative of the performance of the algorithm. Graded meshes were generated in a 4:1 contraction (Fig. 9), a geometry of practical interest in polymer engineering [1]. Location queries were performed on that structure for 10,000 regularly distributed points, leading to the average execution times plotted on Fig. 10. A logarithmic least square fit of the average execution time shows that the average performance is in $O(\log N)$, and therefore below the theoretical worst-case bound of $O(\log N)^2$ for segment trees.

The maximum time for the location of a single point in the same conditions is also plotted on Fig. 11. Precise comparison between theoretical worst-case performance in $O(\log N)^2$ and measured maximum proved inconclusive, and no poly-logarithmic fit was drawn.

In order to further demonstrate the excellent location performance of the segment tree on graded meshes, a straightforward location algorithm was written and run to

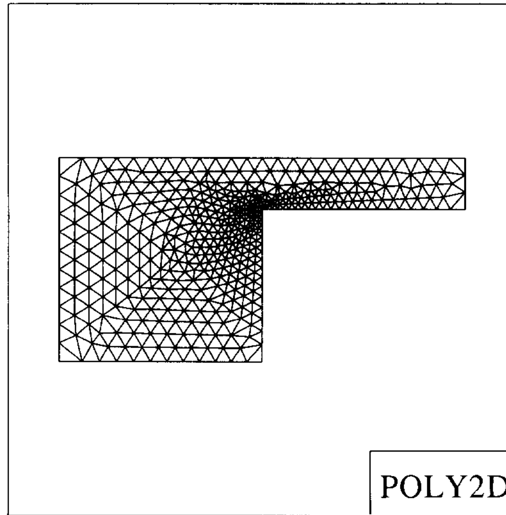

FIG. 9. Typical graded mesh in a 4:1 contraction.

locate 20,000 points in an 800 element mesh, a test case presented in [19]. The speedup achieved with the segment tree is 94, which is to be compared with the speedup of 3 obtained in [19].

The segment tree is rebuilt between iterations, and the building cost itself should be checked against theoretical values (Fig. 11). The theoretical worst-case bound is

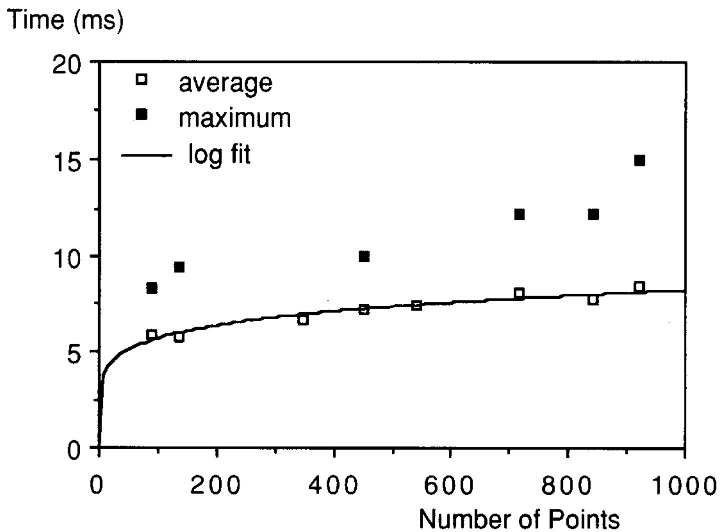

FIG. 10. Maximum execution time, average execution time and corresponding logarithmic fit for the location of a point in a graded mesh stored in a 2D segment tree. Implementation on an Apollo DN 3000 in C language.

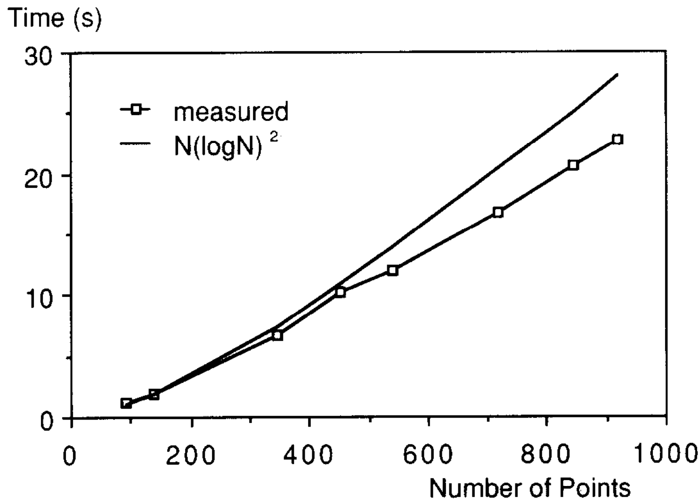

FIG. 11. Execution time for building a 2D segment tree and storing a graded mesh in it. Implementation on an Apollo DN 3000 in C language.

represented by a curve of arbitrary scale for visual comparison. It can be seen that building time is roughly linear in N and below the theoretical bound. Inclusion of the building time in the above comparison only reduces the computed speedup of 10%.

6. CONCLUSION

The meshing and interpolation problems encountered in adaptive remeshing with unstructured grids are very different in nature. They were solved separately using specific data structures based on bisection. These structures are straightforward to program and yet their performance on graded meshes is close to that of present day best algorithms. The structures can also be used to solve the geometrical problems associated with other numerical methods such as the method of characteristics.

REFERENCES

1. M. P. ROBICHAUD AND P. A. TANGUY, *Polym. Eng. Sci.* **29**, 488 (1988).
2. R. L. LÖHNER, *Commun. Appl. Num. Methods* **4**, 123 (1988).
3. W. RHEINBOLDT, *Int. J. Num. Methods Eng.* **17**, 649 (1981).
4. R. E. EWING, in *Accuracy Estimates and Adaptive Refinement in Finite Element Computations*, edited by I. Babuska *et al.* (Wiley, Chichester, 1986), p. 299.
5. K. MELHORN, *Multi-Dimensional Searching and Computational Geometry*, Vol. 3 (Springer-Verlag, Berlin, 1984).
6. J. L. BENTLEY AND H. A. MAURER, *Acta Inform.* **13**, 155 (1980).

7. B. Chazelle, in *Proceedings, IEEE 24th Symposium on Foundations of Computer Sciences*, 1983, p. 122.
8. M. L. Fredman, *J. Assoc. Comput. Mach.* **28**, 696 (1981).
9. V. K. Vaishnavi, *IEEE Trans. Comput.* **C-31**, 22 (1982).
10. J. L. Bentley and D. Wood, *IEEE Trans. Comput.* **C-29**, 571 (1980).
11. N. Wirth, *Algorithm + Data Structures = Programs*, (Prentice–Hall, Englewood Cliffs, NJ, 1976).
12. P. A. Raviart and J. M. Thomas, *Lecture Notes in Mathematics*, Vol. 606, (Springer-Verlag, Berlin, 1977), p. 292.
13. P. J. Green and R. Sibson, *Comput. J.* **21**, No. 2, 168 (1978).
14. J. C. Cavendish, *Int. J. Num. Methods Eng.* **8**, 679 (1974).
15. C. Kittel, *Introduction to Solid State Physics* (Wiley, New York, 1986).
16. S. H. Lo, *Int. J. Num. Methods Eng.* **21**, 1403 (1985).
17. J. Peraire, M. Vahdati, K. Morgan, and O. C. Zienkiewicz, *J. Comput. Phys.* **72**, 449 (1987).
18. M. A. Yerry and M. S. Shephard, *Int. J. Num. Methods Eng.* **20**, 1965 (1984).
19. D. Seldner and T. Westermann, *J. Comput. Phys.* **79**, 1 (1988).