International Journal of Computational Geometry & Applications
© World Scientific Publishing Company

PARALLEL CONSTRUCTION OF QUADTREES AND QUALITY TRIANGULATIONS

MARSHALL BERN* DAVID EPPSTEIN†
SHANG-HUA TENG‡

Received 12 August 1996
Revised 18 February 1997
Communicated by M. T. Goodrich

ABSTRACT

We describe efficient PRAM algorithms for constructing unbalanced quadtrees, balanced quadtrees, and quadtree-based finite element meshes. Our algorithms take time $O(\log n)$ for point set input and $O(\log n \log k)$ time for planar straight-line graphs, using $O(n + k/\log n)$ processors, where n measures input size and k output size.

1. Introduction

A crucial preprocessing step for the finite element method is mesh generation, and the most general and versatile type of two-dimensional mesh is an unstructured triangular mesh. Such a mesh is simply a triangulation of the input domain (e.g., a polygon), along with some extra vertices, called *Steiner points*. Not all triangulations, however, serve equally well; numerical and discretization error depend on the *quality* of the triangulation, meaning the shapes and sizes of triangles. A typical quality guarantee gives a lower bound on the minimum angle in the triangulation.

Baker et al.[1] first proved the existence of quality triangulations for arbitrary polygonal domains; their grid-based algorithm produces a triangulation with all angles between 14° and 90°. Chew[2] also bounded all angles away from 0° using incremental constrained Delaunay triangulation. Both of these algorithms, however, produce triangulations in which all triangles are approximately the size of the smallest input feature; hence, many more triangles than necessary may be generated, slowing down both the mesh generation and finite element procedures. Bern et al.[3] used adaptive spatial subdivision, namely quadtrees, to achieve guaranteed quality with a small number of triangles (within a constant factor of the opti-

*Xerox Palo Alto Research Center, 3333 Coyote Hill Rd., Palo Alto, CA 94304.
†Department of Information and Computer Science, University of California, Irvine, CA 92717.
Supported by NSF grant CCR-9258355. Work performed in part while visiting Xerox Palo Alto Research Center. E-mail: eppstein@ics.uci.edu
‡Department of Mathematics, Massachusetts Inst. of Technology, Cambridge, MA 02139. Work performed in part while at Xerox Palo Alto Research Center.

mal number). Modifications yield other desirable properties, such as small total length[4] and no obtuse triangles.[5] Mitchell and Vavasis generalized this approach to three dimensions.[6] Subsequently, Ruppert[7] built on Chew's more elegant incremental method to achieve theoretical guarantees on number and quality of triangles similar to those of the quadtree meshing methods of Bern et al. For more mesh generation theory, see our survey.[8]

In this paper, we give parallel algorithms for mesh generation. We use a theoretical model of parallel computation, the PRAM, but our main contribution is to reduce much of the work of mesh generation to a sorting algorithm, which can be performed efficiently on most or all parallel computers.

The finite element method is often performed on parallel computers, but parallel mesh generation is less common. (We are unaware of any theoretical papers on the subject and only a few practical papers, for example, see Refs. [9,10,11].) In some applications, a single mesh is generated and used many times; in this case the time for mesh construction is not critical and a relatively slow, sequential algorithm would suffice. In other applications, especially when the physics or geometry of the problem changes with time, a mesh is used once and then discarded or modified. Here, a parallel mesh generator would offer considerable speed-up over a sequential generator.

We parallelize the quadtree-based methods of Bern et al.[3] The grid-based method of Baker et al.[1] and a grid-based modification of Chew's algorithm[2] both parallelize easily, but as mentioned above these may produce too many triangles. It is currently unknown whether Ruppert's method[7] has an efficient parallel version.

A *quadtree*[12] is a recursive partition of a region of the plane into axis-aligned squares. One square, the *root*, covers the entire region. A square can be divided into four *child* squares, by splitting it with horizontal and vertical line segments through its center. The collection of squares then forms a tree, with smaller squares at lower levels of the tree.

It may seem that quadtrees are easy to construct in parallel, a layer a time. In practice this idea may work well, but it does not provide an asymptotically efficient algorithm because the quadtree may have depth proportional to its total size. We use the following strategy instead. We first find a "framework", a tree of quadtree squares such that every internal node has at least two nonempty children. Our construction for this framework is based on sorting the input according to an interleaved bit ordering known as the "linear quadtree".[13] This tree guides the computation of the complete quadtree. We then *balance* the quadtree so that no square is adjacent to a square more than twice its side length. For polygonal inputs, we further refine and rebalance the quadtree so that edges are well separated. Finally, we perform local "warping" as in Bern et al.[3] or Melissaratos and Souvaine,[5] to construct a guaranteed-quality triangulation. Figure 1 shows a mesh computed by a variant of our sequential algorithm.

Since the preliminary conference version of this paper appeared, similar ideas to ours have also found application in problems other than mesh generation. Callahan[14] gives parallel algorithms for certain clustering problems, which he applies to nearest

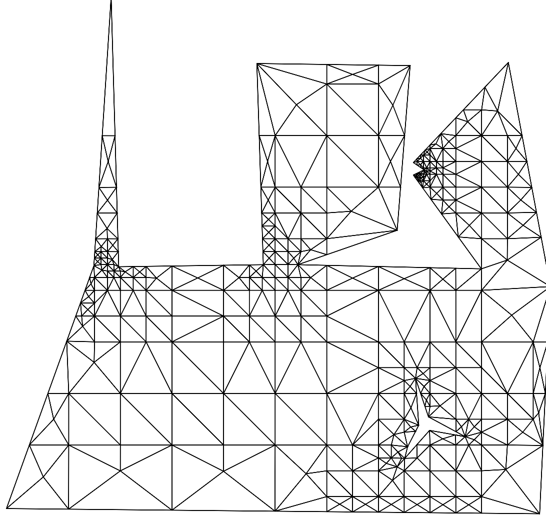

Fig. 1. A mesh derived from a quadtree (courtesy Scott Mitchell).

neighbor searching; his technique is based on the construction of a *fair split tree*. The quadtree framework we develop below is a special case of such a tree, and our methods provide a simpler alternate to his (however Callahan does not make our assumption of integer input). Arya et al.[15] use a form of centroid decomposition in quadtree-like structures, very similar to our algorithm for partitioning input PSLG edges into quadtree squares, as a key part of their approximate nearest neighbor data structure. Mount and Arya have applied the same idea in other approximate range searching problems.[16] More recently, Arora[17] used a variant of quadtrees in his breakthrough $(1 + \epsilon)$-approximation for the Euclidean traveling salesman problem and related problems.

1.1. Computational Model

We assume that the input is a point set or planar straight-line graph, with n vertices. The coordinates of the points or vertices are fixed-point binary fractions, strictly between 0 and 1, that can be stored in a single machine word. We assume the ability to perform simple arithmetic and Boolean operations on such words in constant time per operation, including bit shift operations. Finally, we assume the ability to detect the highest order nonzero bit in the binary representation of such a number in constant time.

These assumptions are similar to a model in which the input coordinates are machine-word integers, as in the work of Fredman and Willard.[18,19] Our description in terms of fixed point fractions is somewhat more convenient for our application, but equivalent in expressive power. If we were to use a real number model instead—as is more usual in computational geometry—we would incur a slight expense in time. The key operation of finding the highest bit can be simulated by binary search

in time $O(\log\log R)$ per operation, where R is the ratio between the largest and smallest numbers tested by the algorithm. For our purposes $R = O(2^k)$ where k is the output size, so the penalty for weakening the model is a factor of $O(\log k)$ time.

Our parallel algorithms use a somewhat nonstandard version of the EREW PRAM model of exclusive access to shared memory.[20] Since our processor bound depends on k, the size of the output triangulation, which is not known in advance, we need a mechanism for allocating additional processors within the course of the computation. We assume that a single allocation step, in which each of the N processors already allocated asks to be replaced by some number k_p of additional processors, can be performed in $O(\log N)$ time. It is assumed that each new processor will receive $O(1)$ words of initialization information provided by the requesting processor p, together with its position in the list of k_p new processors requested by p. The $O(\log N)$ time bound avoids misusing this feature to broadcast information quickly in order to escape the exclusive read restrictions of the model.

1.2. New results

We describe EREW PRAM algorithms to perform the following tasks.

- We construct a balanced or unbalanced quadtree for an arbitrary point set in time $O(\log n)$ with $O(n + k/\log n)$ processors.
- We triangulate a point set with total edge length $O(1)$ times the minimum and all angles bounded between 36° and 80°, using a total number of points within a constant factor of optimal, in time $O(\log n)$ using $O(n + k/\log n)$ processors.
- We triangulate a point set with total edge length $O(1)$ times the minimum and all angles less than 90°, using $O(n)$ Steiner points, in time $O(\log n)$ using $O(n)$ processors.
- We triangulate a planar straight-line graph (PSLG) with all angles except those of the input bounded away from zero, using a total number of triangles within a constant factor of optimal, in time $O(\log k \log n)$ using $O(n+k/\log n)$ processors.

Our last algorithm can also be used to produce a guaranteed-quality triangulation of a polygon, but the number of triangles in this case may not be within a constant of optimal. If the polygon wraps around and nearly touches itself from the outside (as in Figure 1), our algorithm uses some triangles with size approximately the size of this outside "feature", which may be unnecessarily small.

Our results can also be used to fill a gap in our earlier paper,[3] noted by Mitchell and Vavasis.[6] In that paper we gave sequential algorithms for triangulation of point sets, polygons, and planar straight-line graphs, all based on quadtrees. In the conference version of the paper we claimed running times of $O(n \log n + k)$ in all cases, but gave a proof only for the case of point sets. There turn out to be some complications in achieving this bound for polygons and PSLGs. (A straightforward

implementation achieves $O(k \log n)$.) However, the ideas given here also improve sequential quadtree-based triangulation, yielding the first guaranteed-quality triangulation algorithms with the optimal running time of $O(n \log n + k)$. This last result holds also in the more standard real number model of computation.

2. Unbalanced Quadtrees

We first describe how to generate a quadtree for a set of points in the plane. In the resulting quadtree, there are no restrictions on the sizes of adjacent squares, but no leaf square may contain more than one point. The root square of the quadtree will be the semi-open unit square $[0,1)^2$. Thus the corners of quadtree squares have coordinates representable in our fixed-point format. Bern et al.[3] described a sequential algorithm using $O(k + n \log n)$ operations (where k denotes the output size); here we give a parallel algorithm with the same asymptotic bound on the total work but with $O(\log n)$ running time.

The sides of all squares in any quadtree with the given root have lengths of the form 2^{-i}, and for any quadtree square of side length 2^{-i} the coordinates of all four corners are integral multiples of 2^{-i}. We say that a square with bottom left corner (x, y) and size 2^{-i} contains point (x', y') if $x \leq x' < x + 2^{-i}$ and $y \leq y' < y + 2^{-i}$. Given two input points (x, y) and (x', y'), we define their *derived square* to be the smallest square with the above restrictions on side length and corner coordinates that contains both points. The size of this square can be found by comparing the high order bits of $x \oplus x'$ and $y \oplus y'$, and the coordinates of its corners can be found by masking off lower order bits from the coordinates of x and y.

Given a point (x, y), where x and y are k-bit fixed point fractions, we define the *shuffle* $Sh(x, y)$ to be the $2k$-bit fixed point fraction formed by alternately taking the bits of x and y from most significant to least significant, the x bit before the y bit. Since it may not be easy to compute $Sh(x, y)$ explicitly in our model of computation, we represent this value implicitly by the pair (x, y). Any two such pairs can be compared in constant time by using arithmetic and high bit operations separately on x and y.

The first step of our algorithm will be to sort all the input points by the values of their shuffled coordinates. This can be done in $O(\log n)$ time with $O(n)$ EREW processors.[21] From now on we assume that the points occur in this sorted order.

Lemma 1 *The set of points in any square of a quadtree rooted at $[0,1)^2$ form a contiguous interval in the sorted order.*

Proof. The points in a square of size 2^{-i} have the same i most significant shuffled-coordinate bits, and any pair of points with those same bits shares a square of that size. If a point (x, y) is outside the given square, one of its first i bits must differ. If that bit is zero, (x, y) will appear before all points in the square. If the bit is one, (x, y) will appear after all points in the square. Hence it is impossible for (x, y) to appear before some points and after some others in the sorted order. □

Figure 2 illustrates a quadtree with the contiguous intervals defined by Lemma 1.

Lemma 2 *If more than one child of quadtree square s contains at least one input point, then s is the derived square for some two adjacent points in the sorted order.*

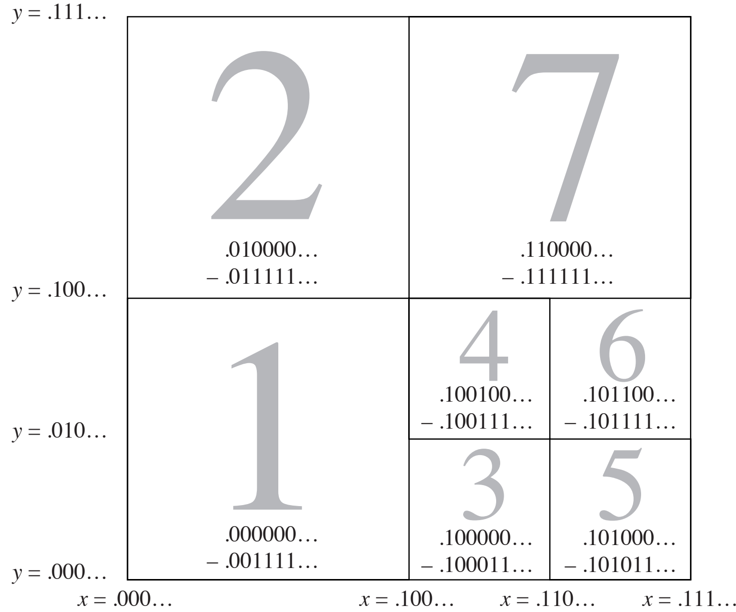

Fig. 2. The contiguous intervals of binary numbers formed by the shuffled coordinates of points in each quadtree square, and the sorted order of those intervals.

Proof. By Lemma 1, the points in s form an interval, which can be divided into two to four smaller intervals corresponding to the children of s. Then s is the derived square for any pair of points in two different smaller intervals. □

Consider the desired unbalanced quadtree as an abstract rooted tree. If we remove all leaves of this tree that do not contain input points, and contract all remaining paths of nodes having one child each, we obtain a tree T_F in which all internal nodes have degree two or more. We call T_F the *framework* and use it to construct the quadtree.

By Lemma 2, the nodes of T_F exactly correspond to the derived squares for adjacent points in the sorted order, and the structure of T_F corresponds to the nesting of intervals induced by the derived squares, as in Lemma 1. So for each adjacent pair of points, we compute their derived square and note its side length. (Some squares may be derived in as many as three ways, but we can eliminate this problem later.)

Lemma 3 *Let s be a quadtree square derived from a pair of points p_i, p_{i+1} in the sorted order. Then the parent of s can be derived from a pair p_j, p_{j+1} such that for any k with $j < k < i$ or $i < k < j$, the derived square of p_k, p_{k+1} has size no larger than s.*

Proof. If s is the first child of its parent, we let p_j be the last point of the sorted order that is contained in s; otherwise we let p_{j+1} be the first point contained in s. Then exactly one of p_j, p_{j+1} is in s, and the other is also in the parent of s, so the

parent of s is derived from p_j, p_{k+1}. For all k between i and j, both p_k and p_{k+1} are in s, so the derived square must either be s or one of its descendants. □

By Lemma 3 we can compute the nesting of squares in T_F by computing the sizes of squares derived from adjacent pairs of points, and for each derived pair finding the nearest pairs to the left and to the right in the sorted order that give a larger size. The smaller of the two resulting squares must be the desired parent. This is an all-nearest-larger-values computation, taking $O(\log n)$ time with $O(n/\log n)$ work.[22]

Once the framework T_F is computed, we construct the quadtree T_Q. Each edge in T_F corresponds to a path of perhaps many squares in T_Q, with the number of squares determined by the relative sizes of T_F squares. So we perform a processor allocation step in which each framework edge e requests $O(p_e/\log n)$ processors, where p_e is e's number of squares. Now all remaining squares (leaves and children of path squares), can be constructed in $O(\log n)$ time. The total number of processors is $O(n + k/\log n)$ where k is the complexity of the resulting (unbalanced) quadtree.

Theorem 1 *Given n input points, we can compute a quadtree with k squares, in which each point is alone in its square, in time $O(\log n)$ using $O(n+k/\log n)$ EREW processors.*

3. Balancing the Quadtree

Most quadtree-based mesh generation algorithms[3,4,5] impose a *balance condition*: no leaf square is adjacent to another leaf square smaller than half its size. (Some variants of these algorithms impose stronger conditions involving bounds on the ratio of side lengths between leaf squares separated by a constant number of other squares; the techniques given here generalize to these stronger conditions with the same asymptotic performance.) These algorithms also need *cross pointers* between squares of the same size sharing a common side. Sequentially, the balance condition can be enforced top down, in time linear in the quadtree complexity. The cross pointers can also be found top down. We present an alternate parallel algorithm that uses $O(k + n \log n)$ work and takes $O(\log n)$ parallel time. Unlike the top-down algorithm we do not create a balanced quadtree with the minimum possible number of squares, however the number of squares will be within a small constant factor of optimal.

We proceed in two stages, starting from the unbalanced quadtree T_Q of Section 2. In the first stage, we produce a tree of squares T'_Q in which some non-leaf squares may have fewer than four child squares. Tree T'_Q, however, will satisfy the balance condition above; it will also include cross pointers.

The balance condition is ensured if, for each square s, three other squares (not necessarily leaves) exist: the two squares adjacent to both s and its parent, twice as large as s, and the square s' sharing a corner with both s and its parent. If s' is contained in the grandparent of s, it has size twice that of s; otherwise it is four times as large as s. See Figure 3 for the latter case. (If any of these squares protrudes from $[0, 1)^2$, then an exception is granted.) The parents of these forced squares will also be forced to exist either by the same rule applied to the parent of

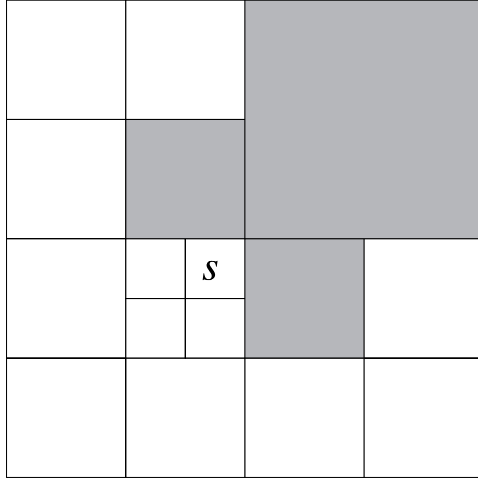

Fig. 3. The three shaded squares are forced by the balance condition for square s.

s, or because they are themselves ancestors of s. The three forced squares do not force other new squares in an unbounded chain of requirements; the nine squares forced by these three either already exist as ancestors of s or are already forced by the parent of s. (We do not require the siblings of the forced squares to be created, as this could result in such an unbounded chain. This is why we allow a square to have fewer than four children.)

How do we add forced squares in parallel all over T_Q? The following simple method will suffice, although it uses more total work than we desire. We simply build a large array, in which we store four squares for each already-existing square: the square itself and the three larger squares forced by the balance condition. Then we sort this array and eliminate duplicates. The squares can be grouped by size, so that (within a single size class) only $O(n)$ squares need to be sorted, and the sorting step takes only $O(\log n)$ time and $O(k \log n)$ total work. Each square can appear only $O(1)$ times in the sorted list, so duplicate elimination takes only constant time. The same sorting idea can be used to create cross-links between squares.

In the second stage of the algorithm, once we have constructed T'_Q, we turn it back into a quadtree simply by splitting all squares, leaves as well as internal nodes without all four children. This preserves the balance condition and restores the required number of children per parent. Due to this extra split, the resulting balanced quadtree T_B may have up to four times as many squares as the quadtree produced by the sequential balancing algorithm. The cross pointers for a square in T_B can be found in constant time by using the cross pointers for the square's parent constructed as part of T'_Q.

The following theorem shows how to combine the "sorting algorithm" just described with a top-down algorithm in order to achieve an optimal parallel work bound, and to reduce the increase in the number of squares compared to the se-

quential algorithm.

Theorem 2 *An unbalanced quadtree can be balanced and cross-linked in time $O(\log n)$ using $O(n + k/\log n)$ processors.*

Proof. Consider the following top-down method. Assume a virtual processor per square of the unbalanced quadtree. This processor waits until all the balancing and cross-linking is done for its parent's level. Then it creates nearby squares to enforce the balance condition, coordinates with neighboring squares so that no new square is created multiple times, and adds all necessary cross-links, in $O(1)$ time. It also tests whether its square contains the (at most one) input point contained in its parent. This results in an algorithm which has time t bounded by the number of levels in the quadtree, and total work $O(k)$. By Brent's lemma,[23] only $O(k/t)$ actual processors are required to simulate all virtual processors in time $O(t)$.

To combine the two algorithms, we apply the sorting algorithm only at certain levels. We choose a set of levels, equally spaced and $\ell = \lceil \log_2 n \rceil$ apart. Once the spacing is fixed there are ℓ ways of making this choice, each giving rise to a set of levels disjoint from other such sets, so for some such set there are a total of only $O(k/\log n)$ squares. The appropriate choice of set can be determined from the framework tree. Once we have added the required squares in this set of levels, we apply the parallelization of the top-down algorithm in $O(\log n)$ time and $O(k)$ total work. The scheduling required for Brent's lemma can be performed using information on the number of squares per level, again computed from the $O(n)$-size framework tree. $\square$

4. Locating Points in Quadtree Squares

The algorithm described in the previous section produces a balanced quadtree T_B defined on the input set of n points, however the information describing for each point the square containing it, available in the unbalanced quadtree, has become lost in the transformations. We next describe how to recover this information.

As in the previous section, we find a set S of $O(k/\log n)$ squares spaced $O(\log n)$ levels apart in T_B. We then partition S into blocks of $O(n)$ squares. Within each block we perform the following computation. We are given $O(n)$ squares s_j in the block, and we also use in each block a list of the n input points p_i. (The set of $O(k/n \log n)$ such lists may be constructed with $O(k)$ work and $O(\log n)$ time by assigning an additional $k/n \log n$ processors per point at the time we performed the first processor allocation step.)

We first find for each square s_i the bottom left corner $b(s_i)$ and the centerpoint $c(s_i)$, and compute a framework F (that is, a compressed quadtree such as the one above) for the set consisting of the union of the point sets p_i, $b(s_j)$, and $c(s_j)$. Then F contains each input square s_j since s_j has two nonempty children (namely those containing $b(s_j)$ and $c(s_j)$. Further each input point p_i is at a leaf of F.

We now follow step by step from the squares in S down through T_B, as in the top-down part of the algorithm in the previous section. Each square s visited in this top-down process is a descendant of a square in some particular block, and hence has a smallest containing square in the corresponding framework F. We simply

compute this containing square from the parent of s. Thus all such containing squares can be found in time $O(\log n)$ and work $O(k)$.

Lemma 4 *Let s be a square of T_B without any children in T_B, and having smallest containing square s' in framework F. Then s' has $O(1)$ descendants in F, and any input point contained in s can be found in $O(1)$ time from these descendants.*

Proof. The only points in s that can have been involved in the construction of F are $b(s)$, $c(s)$, and at most a single input point p_i. Therefore s' can have at most four descendants, one of which is the leaf of F corresponding to p_i. □

Corollary 1 *We can find the correspondence between input points and squares of the balanced quadtree of Theorem 2 in time $O(\log n)$ using $O(n+k/\log n)$ processors.*

5. Mesh Generation for Point Sets

The balanced quadtree computed in the last section can be modified by a set of "warping" steps to give a triangulation of the input point set, with no angles smaller than about 20°, as in Bern et al.[3] These warping steps are local, involving only $O(1)$ squares each, and hence can obviously be performed in constant parallel time with optimal work. In this section, we go on to solve two slightly harder problems: (1) approximate minimum-weight no-small-angle triangulation, and (2) approximate minimum-weight nonobtuse triangulation.

Eppstein[4] showed how to sequentially compute triangulations of point sets with these guarantees: all angles between 36° and 80°, total edge length within a constant factor of the minimum, and total number of triangles within a constant of the minimum for any angle-bounded triangulation. The algorithm again uses local warping, trivial to parallelize, but the quadtree must also satisfy some stronger conditions than the ones given directly by Theorems 1 and 2:

- The coordinate axes of the quadtree must be rotated so that the diameter, of length d, connecting the farthest pair of points, is horizontal.

- A row of equal-size smaller squares that contain the input is cut from the root square. These squares have side length proportional to $\max\{d', d/n\}$, where d' is the maximum distance of any point from the diameter.

- The quadtree square must satisfy a *strong balance condition* in which no quadtree square s can be adjacent to both a smaller square and a larger square, unless the adjacencies are directly across s from each other.

- The points must be *well-separated*, meaning that for some specified constant c, if a point is in a square with side ℓ then its nearest neighbor must be at least distance $c\ell$ away. (In Bern et al.[3], $c = 2\sqrt{2}$.)

We relax these requirements somewhat to simplify our parallel algorithm. Rather than computing the exact diameter, it suffices to find some line segment with length within a factor of, say, .95 of the diameter, and rotate the points so that line segment forms an angle of $O(1/n)$ with the horizontal axis. Such a line segment can

be found by projecting the point set onto $O(1)$ different axes, taking the extrema of each projected point set, and choosing the pair forming the longest segment. The rotation can be performed in our integer model by treating our inputs as complex numbers with integer coordinates, and multiplying by another such number chosen appropriately. The result will be a set of points with the correct orientation but scaled by a factor of $O(n)$, which corresponds to using $O(\log n)$ additional bits to represent each coordinate value.

We scale and translate the rotated point set to fit in the rectangle $[1/4, 3/4) \times [2^{-i}, 2^{-i+1})$, where i is the smallest integer with $2^{-i} \geq 1/n$ for which this is possible. The row of squares will then have side lengths equal to 2^{-i+1}. These steps may all be performed in $O(\log n)$ time with $O(n/\log n)$ processors.

To enforce the strong balance condition, we first balance the quadtree as before, and subdivide each square one further time. After this subdivision, each square that is the smallest of a triple of squares violating the strong balance condition can only be adjacent to same-size or larger squares. We then subdivide again, this time only subdividing squares having a smaller neighbor. In any bad triple, the two larger squares are subdivided, and no new bad triple can be formed in this final subdivision, so the result is a strongly balanced quadtree.

It remains to ensure the separation of points. We first compute *near neighbors* (approximate nearest neighbors), using the balanced quadtree constructed by Theorem 2. We simply examine the $O(1)$ squares around each square containing an input point. If all those squares are empty, we need not find a near neighbor for the input point; otherwise we take the near neighbor to be any point in a neighboring square. We can now add notations to the framework tree of Section 2 specifying the desired size of the quadtree square containing a point with a near neighbor; the size is, say, one-eighth of the distance to the near neighbor. A suitable quadtree can then be computed as in Theorems 1 and 2.

Theorem 3 *Given a set of n points in the plane, we can compute a triangulation with total edge length $O(1)$ times the minimum possible in which all angles measure between $36°$ and $80°$, in time $O(\log n)$ using $O(n + k/\log n)$ EREW processors. The output size k is $O(m + n)$, where m is the minimum number of Steiner points required to triangulate the input point set with no angle smaller than $36°$ (or any other fixed constant).*

We now consider the second problem: nonobtuse triangulation. Sequential quadtree-based methods can triangulate a point set with all acute angles and only $O(n)$ Steiner points.[3] Moreover, the total edge length can be made

to approximate that of the minimum-weight triangulation.[4] The following theorem extends this result to the parallel case.

The idea in Bern et al.[3] is to locate *clusters* of points, triangulate these clusters recursively, and then treat them as single points in the outer quadtree. A cluster is a set of closely spaced points far from all other points, that lies in a long path of quadtree squares, or equivalently, in a square of the framework tree whose parent square is much larger. The sequential algorithm proceeds top-down, recognizing clusters whenever the required sequence of subdivisions is seen. This algorithm

does not work in parallel since the quadtree may have as many as $\Omega(n)$ levels.

Our parallel algorithm uses the framework tree to identify *potential clusters*, that is, parent-child pairs of sufficiently large size difference. The algorithm then builds a collection of balanced quadtrees in parallel, one for each potential cluster, using the methods of Theorems 1 and 2. Small potential clusters within a larger one are represented by single points in this computation.

A true cluster is a potential cluster that is also well-separated from its neighbors. If the quadtree square containing a potential cluster's representative point is only a constant factor larger than the potential cluster itself, we merge the potential cluster with the quadtree containing it. Otherwise, we check that containing square and $O(1)$ neighbors and ancestors. These contain $O(1)$ points and clusters, some of which may be too close to the given cluster. If so, they merge with it to form a true cluster. Isolated points may also need to merge to form small clusters. At most four clusters or points may join in any merger, and this can only happen if all four are near a shared corner of their squares. Each merged group may then split immediately into its component clusters, but this can be tested in constant time.

Once the true clustering structure is known, the final balanced quadtrees may be constructed as before. The resulting system of quadtrees is triangulated using local rules.

Theorem 4 *A set of n input points can be triangulated with $O(n)$ triangles, all angles less than $90°$, and total length $O(1)$ times the minimum possible, in time $O(\log n)$ using $O(n)$ EREW processors.*

6. Mesh Generation for Planar Straight-line Graphs

For most finite-element mesh generation applications, the input is not a point set but rather a polygonal region. We discuss here the most general input, a planar straight-line graph (PSLG). Our triangulation algorithm handles simple polygons as a special case, but the output complexity may be larger than necessary due to input features that are near to each other in Euclidean distance but far in geodesic distance.

Several new complications arise with PSLG input. First, we must modify the input to eliminate any pre-existing acute (below $90°$) corners,[3] and this should be done without introducing new points or edges too close to existing ones. Second, we must subdivide the edges of the PSLG where they cross the sides of quadtree squares. Third, we require that vertices not only be well-separated from other vertices, but also from other edges, and that edges be well-separated from each other; this means that each (piece of an original) edge is contained in a sufficiently small square that no vertex or other edge passes nearby (less than a specified constant—such as 3—times the square's side length).

To accomplish these goals, we take the following approach. We compute vertex-to-vertex and vertex-to-edge approximate nearest neighbors using an initial balanced quadtree T_B. This information enables us to cut off the acute corners and build a second quadtree T'_B in which vertices are well-separated from edges and other vertices. We subdivide the edges into pieces in this quadtree, and then split

the squares of the quadtree to create a third quadtree T''_B in which each square contains only $O(1)$ pieces of edges. We can then—finally—compute approximate nearest neighbor information for pieces of edges, which we use to construct the final quadtree T^*_B, in which everything is well-separated.

We now flesh out the steps mentioned above. Approximate vertex-vertex nearest neighbors proceeds as in Section 5. The following lemma shows how to find a nearby non-incident edge for each vertex of the planar straight-line graph.

Lemma 5 *If the nearest edge e to a vertex v has distance d from v, then either some vertex v' is within distance $\sqrt{2}d$ from v, or e is visible to v through an angle of at least* $90°$.

Proof. If neither endpoint of e is within distance $\sqrt{2}d$, then e covers an angle of $90°$ within which it is entirely within distance $\sqrt{2}d$. No other edge can entirely obscure e in that angle, or it would cross the segment of length d between p and e, and hence be closer than e. If an edge partially obscures e in that angle, it has an endpoint within distance $\sqrt{2}d$. □

So for each vertex of the PSLG, we need only determine which edge is visible along each of the four axis directions, and choose the nearest of these four edges. This horizontal and vertical ray-shooting problem is exactly that solved by the *trapezoidal decomposition*, which can be constructed in time $O(\log n)$ using $O(n)$ CREW processors;[24] this algorithm can be simulated on an EREW machine with logarithmic slowdown.[20]

Now we cut off acute angles around each vertex at a distance proportional to the vertex's nearest neighbor as in Bern et al.[3] and Ruppert,[7] so that cut-off triangles do not contain other parts of the input, two such triangles on the same edge match up, and the new cutting edges are not unnecessarily short. We can ignore the cut-off triangles for the remainder of the algorithm, simply reattaching them (with appropriately subdivided cutting edges) after the final warping step. For the remainder of the algorithm, we can assume that all angles measure at least $90°$, and consequently each vertex has bounded degree.

We can now compute a quadtree T'_B in which each vertex lies in a square of size proportional to the nearer of its nearest neighbors (vertex and edge). Vertices will be well-separated from other vertices and edges, but two PSLG edges may yet cross the same square. We subdivide the PSLG edges into pieces contained in each quadtree square so that we will be able to discover these problems and correct them. Sequentially, we could simply walk along each edge, keeping track of its crossings with each square, but in a parallel algorithm we must do this differently.

We apply the accelerated centroid technique[25] to T'_B. This produces a binary tree T_C, the *accelerated centroid tree*, which has logarithmic height and linear size; each subtree of T_C corresponds to a subtree of T'_B. To find the subdivisions of a given PSLG edge, we test it against T_C's root node r. Node r corresponds to a square in the quadtree T'_B, and we simply test whether the edge misses the square, is contained in the square, or crosses the square's boundary. If one of the first two cases occurs, we continue recursively to the left or right child of r. If the third case occurs, we split the segment in pieces and continue in parallel for each piece

to the appropriate child of r. We process all edges in parallel, one level of T_C at a time. Each level can be handled in $O(\log k)$ time by $O(k/\log k)$ EREW processors. Between levels we reallocate processors and split the lists of edges being tested, resulting in time $O(\log k)$ and linear work.

This edge-subdivision step of our algorithm takes $O(\log^2 k)$ time and a total of $O(k \log k)$ work. But in each of these bounds, a factor of $O(\log k)$ can be reduced to $O(\log n)$ by the following trick: partition the quadtree T'_B into $O(k/n)$ subtrees, each of $O(n)$ squares, and perform the edge partition algorithm described above in parallel for each edge in each subtree. The number of levels in each centroid decomposition tree is thus reduced from $O(\log k)$ to $O(\log n)$, but $O(\log k)$ time per level is still required for processor reallocation.

We now look at the arrangement of edges in a single square of quadtree T'_B. We assert that each cell in this arrangement has only $O(1)$ complexity. If the cell is not convex, then there is a vertex of the PSLG on its boundary, and the bound on angles then implies the assertion. If the cell is convex, PSLG edges on its boundary must either meet at a nearby PSLG vertex or be nearly parallel. The fact that vertices are well-separated in T'_B then implies the assertion.

We consider in parallel each pair of edge pieces that bound a common cell. By the assertion above, there are only $O(k)$ such pairs over all of T'_B. For each pair, we determine the further quadtree subdivision that would be necessary to separate the edges in that square. We combine and balance all such subdivisions with sorting (as in Section 3) to produce a new balanced quadtree T''_B in time $O(\log n)$ and work $O(k \log n)$. We again use a centroid tree to determine the sub-pieces of edge pieces; each quadtree square now intersects only $O(1)$ sub-pieces.

Finally, for each square we search $O(1)$ nearby squares to determine how much further splitting is necessary. We again subdivide and rebalance to construct a last quadtree T^*_B, in which all PSLG edges are finally well-separated. Local warping then offers the following result. Bern et al.[3] achieve an angle bound of 18° sequentially; the same specific bound can be achieved in parallel, only with a constant factor more triangles.

Theorem 5 *Given a planar straight-line graph with n vertices, we can compute a triangulation with k triangles, in which all new angles are bounded away from 0°, in time $O(\log k \log n)$ using $O(n + k/\log n)$ EREW processors. The output size k is $O(n+m)$, where $m = m(\epsilon)$ is the minimum number of Steiner points required to triangulate the given PSLG with no new angle smaller than fixed constant ϵ.*

7. Sequential Algorithms

As noted in the introduction, our results fill a gap in our earlier paper.[3] In that paper we gave sequential algorithms for triangulation of point sets, polygons, and planar straight-line graphs, all based on quadtrees. However we were unable to prove our claimed $O(n \log n + k)$ time bound for polygons and PSLG's. As we now see, this can be achieved using ideas of the previous section.

Theorem 6 *Given a planar straight-line graph with n vertices, we can compute a triangulation with k triangles, in which all new angles are bounded away from 0°,*

in time $O(k + n \log n)$.

Proof. We follow the same basic outline of the previous section: we first construct a quadtree on the input vertices, and use this to find approximate nearest neighbors to each vertex. We use this information to guide the construction of a second quadtree in which all vertices are well-separated. We then subdivide the input edges into portions contained within a single quadtree square; unlike the parallel case we can simply walk from one end of each edge to the other, keeping track of the crossed squares. Finally, as in the previous section, we construct cells of constant complexity within each quadtree square, and determine the necessary quadtree subdivision separately within each cell. Unlike the parallel case, we can glue the subdivided cells together in linear time by walking sequentially along their boundaries. □

Note that the only steps in which our parallel algorithm requires the input to be integer-valued are in the construction of quadtree frameworks for point sets, however sequentially we can construct these in $O(n \log n)$ time even for real-valued input, as shown in Bern et al.[3] Therefore the sequential algorithm above works also for real-valued point coordinates.

8. Conclusions

We have given a theoretical study of parallel two-dimensional mesh generation, and also demonstrated the application of similar techniques to sequential mesh generation algorithms. We believe this area deserves further research, both theoretical and practical.

The triangulations we construct are within a constant factor of the optimal complexity, but it might be of some practical interest to improve the constant factors. In particular, our balancing method wastes a factor of four, and further constant factors are lost to achieve the strong balance condition. Is it possible to compute the minimum balanced quadtree for a set of n points in time $O(\log n)$? Another interesting problem that we are leaving open is the parallel computation of approximate geodesic nearest neighbors. Efficient algorithms for this problem (both vertex-vertex and vertex-edge) would extend our PSLG methods to simple polygons with a stronger bound on the total number of triangles.

References

1. B. Baker, E. Grosse, and C. Rafferty. Nonobtuse triangulation of polygons. *Disc. Comput. Geom.* 3:147–168, 1988.
2. L. P. Chew. Guaranteed-quality triangular meshes. Tech. Rep. TR-89-983, Cornell Univ., Dept. of Computer Science, 1989.
3. M. Bern, D. Eppstein, and J. R. Gilbert. Provably good mesh generation. *J. Comp. Sys. Sci.* 48:384–409, 1994.
4. D. Eppstein. Approximating the minimum weight Steiner triangulation. *Disc. Comput. Geom.* 11:163–191, 1994.
5. E. A. Melissaratos and D. L. Souvaine. Coping with inconsistencies: A new approach to produce quality triangulations of polygonal domains with holes. *8th ACM Symp.*

Comp. Geom., pp. 202–211, 1992.

6. S. A. Mitchell and S. A. Vavasis. Quality mesh generation in three dimensions. *8th ACM Symp. Comp. Geom.*, pp. 212–221, 1992.
7. J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. *4th ACM-SIAM Symp. Discrete Algorithms*, pp. 83–92, 1993.
8. M. Bern and D. Eppstein. Mesh generation and optimal triangulation. *Computing in Euclidean Geometry*, 2nd edition, pp. 47–123. World Scientific, Lecture Notes Ser. Comput. 4, 1995.
9. M. T. Jones and P. E. Plassman. Parallel algorithms for the adaptive refinement and partitioning of unstructured meshes. *Scalable High-Performance Computing Conf.*, pp. 478–485, 1994.
10. R. Lohner, J. Camberos, and M. Merriam. Parallel unstructured grid generation. *Comp. Meth. Appl. Mech. Eng.* 95:343–357, 1992.
11. R. D. Williams. Adaptive parallel meshes with complex geometry. Tech. Rep. CRPC-91-2, Cal. Tech., Ctr. for Research on Parallel Computation, 1991.
12. H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys* 16:188–260, 1984.
13. I. Gargantini. An effective way to represent quadtrees. *Commun. ACM* 25:905–910, 1982.
14. P. B. Callahan. Optimal parallel all-nearest-neighbors using the well-separated pair decomposition. *34th IEEE Symp. Found. Comp. Sci.*, pp. 332–340, 1993.
15. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbor searching. *5th ACM-SIAM Symp. Discrete Algorithms*, pp. 573–582, 1994.
16. D. M. Mount and S. Arya. Approximate range searching. *11th ACM Symp. Comp. Geom.*, pp. 172–181, 1995.
17. S. Arora. Nearly linear time approximation schemes for Euclidean TSP and other geometric problems. *38th IEEE Symp. Found. Comp. Sci.*, pp. 554–563, 1997.
18. M. L. Fredman and D. E. Willard. Surpassing the information-theoretic bound with fusion trees. *J. Comp. Sys. Sci.* 47:424–436, 1993.
19. D. E. Willard. Applications of the fusion tree method to computational geometry and searching. *3rd ACM-SIAM Symp. Discrete Algorithms*, pp. 286–295, 1992.
20. D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comp. Sci.* 3:233–283, 1988.
21. R. Cole. Parallel merge sort. *SIAM J. Comput.* 17:770–785, 1988.
22. O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. *21st ACM Symp. Theory of Computing*, pp. 309–319, 1989.
23. R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM* 21:201–206, 1974.
24. M. J. Atallah, R. Cole, and M. T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Comput.* 18:499–532, 1989.
25. R. Cole and U. Vishkin. Optimal parallel algorithms for expression tree evaluation and list ranking. *3rd Aegean Worksh. Comput.* Springer-Verlag, Lecture Notes in Comp. Sci. 319, 1988.