Parallel Construction of Quadtrees and Quality Triangulations

Marshall Bern* David Eppstein† Shang-Hua Teng‡

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. [2] 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 [7] 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. [4] used adaptive spatial subdivision, namely quadtrees, to achieve guaranteed quality with a small number of triangles (within a constant factor of the optimal number). Modifications yield other desirable properties, such as small total length [10] and no obtuse triangles [13]. Mitchell and Vavasis

*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.

‡Department of Mathematics, Massachusetts Inst. of Technology, Cambridge, MA 02139. Work performed in part while at Xerox Palo Alto Research Center.

generalized this approach to three dimensions [14]. Subsequently, Ruppert [15] built on Chew's more elegant algorithm to achieve the same theoretical guarantee on the number of triangles. For more mesh generation theory, see our survey [5].

In this paper, we give parallel algorithms for mesh generation. 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 [18].) 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. Then a parallel mesh generator would offer considerable speed-up over a sequential generator.

We parallelize the quadtree-based methods of Bern et al. [4]. The grid-based method of Baker et al. [2] and a grid-based modification of Chew's algorithm [7] both parallelize easily, but as mentioned above these may produce too many triangles. It is currently unknown whether Ruppert's method [15] has an efficient parallel version.

A *quadtree* [16] 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. 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 [4] or [13], to construct a guaranteed-quality triangulation. Figure 1 shows a mesh computed by a variant of our sequential algorithm.

1.1 Input assumptions

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 [12, 17]. Our

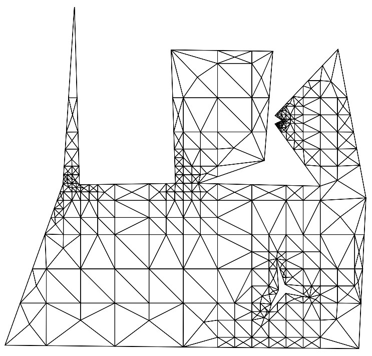

Figure 1. A mesh derived from a quadtree (courtesy Scott Mitchell).

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 the EREW PRAM model of exclusive access to shared memory [11]. 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 of additional processors, can be performed in $O(\log N)$ time.

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 produces 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 [4], noted by Mitchell and Vavasis [14]. In that paper we gave sequential algorithms for triangulation of point sets, polygons, and planar straight-line graphs, all based on quadtrees. We claimed running times of $O(n \log n + k)$ in all cases, but gave a proof only the case of point sets. There turned 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 $O(n \log n + k)$.

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.

The sides of all squares in the quadtree have lengths of the form 2^{-i}, and for any square of side length 2^{-i} the coordinates of all four corners are integral multiples of 2^{-i}. We define a square with bottom left corner (x, y) and size 2^{-i} to contain a 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 such square containing 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.

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. We represent $Sh(x, y)$ implicitly by the pair (x, y). We can compare two numbers $Sh(x, y)$ and $Sh(x', y')$ in this implicit representation in constant time, 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 [8]. 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. ∎

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

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.) The nesting of intervals is computed by finding, for each derived square, the first larger one to its right and to its left in the sorted order. This is an all-nearest-larger-values computation, taking $O(\log n)$ time with $O(n/\log n)$ work [3].

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.* ∎

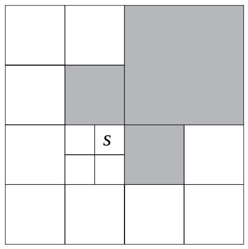

Figure 2. The squares forced by square s.

3 Balancing the quadtree

The quadtree-based mesh generation algorithms of [4, 10, 13] 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; the techniques given here generalize.) These algorithms also need *cross pointers* between squares of the same size sharing a common side.

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 sharing a corner with both s and its parent, four times as large as s. See Figure 2. (If any of these squares protrudes from $[0,1)^2$, then an exception is granted.) The parents of these forced squares exist either by the same rule applied to the parent of 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 or are forced by the parent of s. (We do not require the siblings of the forced squares, 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? For each side length in T_Q, we compute a list of squares (say top left corners) containing: all current squares, all squares of that size that we must create, and all squares that should be cross-linked if they exist or are created. Then we simply sort these lists using shuffled or unshuffled coordinates. Each list contains $O(1)$ copies of any top left corner; these determine all actions necessary to produce T'_Q (creating squares and adding cross pointers). Each quadtree level has at most $O(n)$ squares, so this algorithm takes only $O(\log n)$ time, but requires $O(k)$ processors and hence $O(k \log n)$ total work.

In the second stage, we turn T'_Q back into a quadtree by simply 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. Cross pointers for T_B can be found recursively by examining squares' siblings as well as the children of squares cross-linked to their parents.

The two stages together give us the balanced quadtree T_B, but we may no longer know which squares contain which input points. To obtain this important information, we group the squares of T_B into blocks of $O(n)$ squares each and then process each block separately. Within a block, we compute a framework (that is, a compressed quadtree as above) for the input points along with the bottom left corners of the new squares. We find the squares of this framework that contain one input point, and from them determine the nearest input point above and to the right of each T_B square corner. For each T_B square, we can then test its closest above-right point to see whether it lies within the square. These steps also take $O(\log n)$ time and $O(k \log n)$ work.

The following theorem shows how to combine the "sorting algorithm" just described with a top-down algorithm in order to achieve optimal efficiency.

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 [6], 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. ∎

4 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 [4]. 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 [10] 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 points must be *well-separated* from each other, 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 [4], $c = 2\sqrt{2}$.)

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.

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 k triangles, total 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 $m = m(\epsilon)$ is the minimum number of Steiner points required to triangulate the input point set with no angle smaller than fixed constant ϵ.* ∎

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 [4]. Moreover, the total edge length can be made to approximate that of the minimum-weight triangulation [10]. The following theorem extends this result to the parallel case. To save space, we omit the algorithm and proof; these will appear in the journal version of this paper.

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.*

5 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 [4], 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 4. The following lemma (easy geometric proof omitted) shows how to find a nearby non-incident edge for each vertex of the planar straight-line graph.

Lemma 3. *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 $90°$.*

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 [1]; this algorithm can be simulated on an EREW machine with logarithmic slowdown [11].

Now we cut off acute angles around each vertex at a distance proportional to the vertex's nearest neighbor as in [4, 15], 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 as in [4, 15] 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. In a sequential algorithm, we could subdivide simply by walking from one end of each edge to the other, keeping track of the crossed squares, but in a parallel algorithm we must do this differently.

We apply the accelerated centroid technique [9] 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. [4] 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.* ∎

6 Conclusions

We have given a theoretical study of parallel two-dimensional mesh generation. 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; 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] M.J. Atallah, R. Cole, and M.T. Goodrich. Cascading divide-and-conquer: a technique for designing parallel algorithms. *SIAM J. Comput.* 18 (1989) 499–532.

[2] B. Baker, E. Grosse, and C. Rafferty. Nonobtuse triangulation of polygons. *Discrete Comput. Geom.* 3 (1988) 147–168.

[3] O. Berkman, D. Breslauer, Z. Galil, B. Schieber, and U. Vishkin. Highly parallelizable problems. *21st Symp. Theory of Computing* (1989) 309–319.

[4] M. Bern, D. Eppstein, and J.R. Gilbert. Provably good mesh generation. *31st Symp. Found. Comput. Sci.* (1990) 231–241. To appear in *J. Comp. Sys. Sci.*

[5] M. Bern and D. Eppstein. Mesh generation and optimal triangulation. In *Euclidean Geometry and the Computer*, World Scientific, 1992.

[6] R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM* 21 (1974) 201–206.

[7] L.P. Chew. Guaranteed-quality triangular meshes. TR-89-983, Cornell, 1989.

[8] R. Cole. Parallel merge sort. *SIAM J. Comput.* 17 (1988) 770–785.

[9] R. Cole and U. Vishkin. Optimal parallel algorithms for expression tree evaluation and list ranking. *3rd Aegean Workshop on Computing*, Springer LNCS 319 (1988).

[10] D. Eppstein. Approximating the minimum weight triangulation. *3rd Symp. Discrete Algorithms* (1992) 48–57. To appear in *Discrete Comput. Geom.*

[11] D. Eppstein and Z. Galil. Parallel algorithmic techniques for combinatorial computation. *Ann. Rev. Comput. Sci.* 3 (1988) 233–283.

[12] M.L. Fredman and D.E. Willard. Blasting through the information-theoretic barrier with fusion trees. *22nd Symp. Theory of Computing* (1990) 1–7.

[13] E.A. Melissaratos and D.L. Souvaine. Coping with inconsistencies: A new approach to produce quality triangulations of polygonal domains with holes. *8th Symp. Comput. Geom.* (1992) 202–211.

[14] S.A. Mitchell and S.A. Vavasis. Quality mesh generation in three dimensions. *8th Symp. Comput. Geom.* (1992) 212–221.

[15] J. Ruppert. A new and simple algorithm for quality 2-dimensional mesh generation. *4th Symp. Discrete Algorithms* (1993) 83–92.

[16] H. Samet. The quadtree and related hierarchical data structures. *Computing Surveys* 16 (1984) 188-260.

[17] D.E. Willard. Applications of the fusion tree method to computational geometry and searching. *3rd Symp. Discrete Algorithms* (1992) 286–295.

[18] R.D. Williams. Adaptive parallel meshes with complex geometry. Tech. Report CRPC-91-2, Center for Research on Parallel Computation, Cal. Tech.