Parallelism in Dynamic Well-Spaced Point Sets

Umut A. Acar
Max-Planck Institute for Software Systems
Kaiserslautern, Germany
umut@mpi-sws.org

Andrew Cotter
Toyota Technological Institute
Chicago, IL
cotter@ttic.edu

Benoît Hudson
Autodesk, Inc.
Montreal, QC, Canada
benoit.hudson@autodesk.com

Duru Türkoğlu
Dept. of Computer Science
University of Chicago
Chicago, IL
duru@cs.uchicago.edu

ABSTRACT

Parallel algorithms and dynamic algorithms possess an interesting duality property: compared to sequential algorithms, parallel algorithms improve run-time while preserving work, while dynamic algorithms improve work but typically offer no parallelism. Although they are often considered separately, parallel and dynamic algorithms employ similar design techniques. They both identify parts of the computation that are independent of each other. This suggests that dynamic algorithms could be parallelized to improve work efficiency while preserving fast parallel run-time.

In this paper, we parallelize a dynamic algorithm for well-spaced point sets, an important problem related to mesh refinement in computational geometry. Our parallel dynamic algorithm computes a well-spaced superset of a dynamically changing set of points, allowing arbitrary dynamic modifications to the input set. On an EREW PRAM, our algorithm processes batches of k modifications such as insertions and deletions in $O(k \log \Delta)$ total work and in $O(\log \Delta)$ parallel time using k processors, where Δ is the geometric spread of the data, while ensuring that the output is always within a constant factor of the optimal size. EREW PRAM model is quite different from actual hardware such as modern multiprocessors. We therefore describe techniques for implementing our algorithm on modern multi-core computers and provide a prototype implementation. Our empirical evaluation shows that our algorithm can be practical, yielding a large degree of parallelism and good speedups.

Categories and Subject Descriptors

F.2.2 [**Analysis of Algorithms and Problem Complexity**]: Nonnumerical Algorithms and Problems—*Geometrical problems and computations*

SPAA'11, June 4–6, 2011, San Jose, California, USA.

General Terms

Algorithms, Theory, Performance, Experimentation

Keywords

Well-spaced point sets, Voronoi diagrams, mesh refinement, parallel batch dynamic updates, self-adjusting computation, multithreading

1. INTRODUCTION

In many applications, algorithms are repeatedly invoked on sequences of data that are related to or derived from each other. For example, in computer aided design, one might create an incrementally evolving model which undergoes relatively small changes at each iteration. After adjusting the design, the user may invoke features that send the design as input to an algorithm that requires expensive computations, e.g., a geometric algorithm such as a mesh generator. To take advantage of the resulting similarity between the inputs, researchers have developed so called *dynamic algorithms* that update the output by performing significantly less work than a complete re-computation. Except for a few (e.g, [19, 25]), dynamic algorithms are typically sequential (see [11, 14, 13] for some surveys) and allow the input to undergo only a single modification, e.g., insertion or deletion, at a time.

The interaction between parallel and dynamic algorithms is not well understood, but appears to be strong. For example, they are duals in terms of their effect on work and run-time: when compared to sequential algorithms, parallel algorithms improve parallel run-time but not the work (total computation), whereas dynamic algorithms improve work by only updating parts of the output affected by a change, but offer no parallelism. The design and analysis of parallel and dynamic algorithms are also deeply related. Both parallel and dynamic algorithms identify independent parts of the computation in order to achieve either a high degree of parallelism or efficient dynamic updates when the input is modified. These relationships suggest that dynamic and parallel algorithms may be designed to improve both work and parallel run-time. Such algorithms accept arbitrarily changing input data sets (as opposed to permitting single, unitary changes to the input), and respond to them in both work-optimal and parallel-time-optimal fashion by

performing only the amount of work necessary to update the previous output in accordance with the modified input.

In this paper, we present parallel construction and parallel dynamic update algorithms for the *well-spaced point-sets problem*. Given an input set of points N, the problem requires computing the asymptotically smallest *well-spaced* superset $\mathsf{M} \supset \mathsf{N}$ by inserting additional so called *Steiner points*, to ensure that the Voronoi cells of the final Voronoi diagram all have bounded aspect ratio (Section 2). Well-spaced point sets are closely related to mesh generation, an important problem in computational geometry [10, 6, 21, 27, 16], where the goal is to cover a domain with simplices (i.e. triangles in 2D and tetrahedra in 3D) such that all simplices have good quality, in the sense that their face and dihedral angles are bounded away from 0° and 180°. Well-spaced point sets directly yield quality meshes in 2D, and can be used to obtain quality meshes in 3D with the help of a sliver removal algorithm [9].

Our construction algorithm builds a well-spaced superset of a given point set, and our dynamic algorithm updates it as the input is modified. Both return well-spaced point sets that are *size-optimal*, in the sense that their size is within a constant factor of the size of the smallest well-spaced superset. Our dynamic update algorithm allows arbitrary modifications to the input, e.g., single or batch insertions, or deletions, or their combinations. We present our algorithms in the EREW PRAM model, where no concurrent memory accesses take place. Our construction algorithm matches the best prior parallel off-line algorithms [17] in efficiency: it builds a superset of a given set of n points in $O(n \log \Delta)$ work and $O(\log \Delta)$ parallel time using n processors, where Δ is the *geometric spread*, defined as the ratio of the diameter to the closest pair distance of the input set. Our update algorithm performs a batch of k updates (insertions and deletions) in $O(k \log \Delta)$ work and $O(\log \Delta)$ parallel time using k processors. These bounds assume that vertex coordinates fit inside a machine word, and that common arithmetic operations on these words require only constant time.

The starting point for our algorithms is the aforementioned similarity between the design of dynamic and parallel algorithms: that the very independence exploited by a dynamic algorithm may be utilized to extract parallelism. Indeed, we start with a previously proposed dynamic algorithm for well-spaced point sets [4] and parallelize it for the EREW PRAM model (Section 4). Parallelization of the dynamic algorithm involves some major modifications to the quadtree data structure used for point location (Section 3) and careful exploitation of locality properties of the geometric operations used by the dynamic algorithm. This yields a parallel dynamic update algorithm that accepts arbitrary modifications to the input and updates the output correctly and efficiently. To support such parallel dynamic updates, we utilize a *computation graph* to represent the dependencies in the computation, allowing quick identification and re-use of those tasks which are unaffected by a modification. We show that it is possible to provide lock-free mutually exclusive access to the computation graph by taking advantage of certain locality properties (Section 5).

The approach of developing a construction algorithm and then providing a dynamic update algorithm based on change propagation is inspired by recent advances on self-adjusting computation (e.g., [2, 15, 20]). In self-adjusting computation, programs can respond automatically to modifications

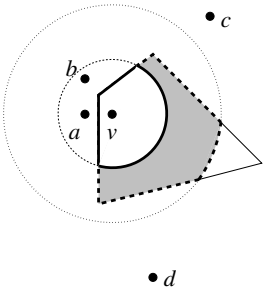

Figure 1: $\mathsf{M} = \{a, b, c, d, v\}$ $\mathrm{NN}_{\mathsf{M}}(v) = |va|$. Thick solid and dashed boundaries display $\mathrm{Vor}^{\rho}_{\mathsf{M}}(v)$ and $\mathrm{Vor}^{\beta}_{\mathsf{M}}(v)$. Shaded region is the (ρ, β) picking region. c and d are β-clipped but not ρ-clipped Voronoi neighbors of v.

to their data by invoking a general-purpose change propagation algorithm [1]. The data structures required by change propagation can be constructed automatically by observing execution. Our computation graphs are abstract representations of these data structures. Similarly our dynamic update algorithm is an adaptation of the change propagation algorithm for the problem of well-spaced point sets. Nearly all of that work, however, consider sequential computations. The results in this paper show that they can effectively be parallelized in this particular problem. Previous work applied these techniques effectively to other computational geometry problems such as kinetic three-dimensional convex hulls [3] and sequential dynamic and kinetic well-spaced point sets [4, 5]. Our approach can also be viewed as a dynamization technique, which has been used effectively for a relatively broad range of algorithms (e.g., [26, 8, 12, 24]).

Since our algorithms are based on the EREW PRAM model, and since our bounds are asymptotic, it is not directly clear if our algorithms yield the desired efficiency in practice, e.g., on modern multi-core computers with a modest number of processors. For example, in the EREW PRAM model, processors execute in lockstep, emulation of which in multi-core systems would require global synchronization. Similarly, in our EREW algorithms, exclusivity of memory accesses is ensured at the cost of a large amount of sequentialization, although much less suffices in practice, largely because concurrent reads are permissible. We therefore present practical adaptations of our construction and dynamic update algorithms to multi-core systems (Section 6). We also provide prototype implementations. Our experiments with 2D and 3D synthetic and real-world data sets give some evidence that the proposed techniques work effectively in practice (Section 7).

2. PRELIMINARIES

Our algorithms maintain a well-spaced superset M of a set of input points N that can be dynamically modified by insertion/deletion of a set of points N^*. Without loss of generality, we scale and shift the point set N such that $B = [0, 1]^d$ becomes a bounding box. We use the term *point* to refer to any point in B and *vertex* to refer to the input and output points. For any vertex set $\mathsf{M} \subset B$, the *Voronoi cell* of v in M, written $\mathrm{Vor}_{\mathsf{M}}(v)$, consists of points $x \in B$ such that for all $u \in \mathsf{M}, |vx| \leq |ux|$. The *nearest-neighbor distance of* v, written $\mathrm{NN}_{\mathsf{M}}(v)$, is the distance from v to the nearest other vertex in M. Following Talmor [27], a vertex is *ρ-well-spaced* if its Voronoi cell is contained within the ball of radius $\rho\,\mathrm{NN}_{\mathsf{M}}(v)$ centered at v; M is ρ-well-spaced if every vertex in M is ρ-well-spaced. The *β-clipped Voronoi cell* of v, written $\mathrm{Vor}^{\beta}_{\mathsf{M}}(v)$, is the intersection of $\mathrm{Vor}_{\mathsf{M}}(v)$ with the ball of radius $\beta\,\mathrm{NN}_{\mathsf{M}}(v)$ centered at v [18]. For any $\beta > \rho$, we define the (ρ, β) *picking region* of v, written $\mathrm{Vor}^{(\rho,\beta)}_{\mathsf{M}}(v)$, as

$\mathrm{Vor}^{\beta}_{\mathsf{M}}(v) \setminus \mathrm{Vor}^{\rho}_{\mathsf{M}}(v)$, the region of the Voronoi cell bounded by concentric balls of radius $\rho\,\mathrm{NN}_{\mathsf{M}}(v)$ and $\beta\,\mathrm{NN}_{\mathsf{M}}(v)$. A vertex u is a *(β-clipped) Voronoi neighbor* of v if the (β-clipped) Voronoi cell of v contains a point equidistant from v and u. Figure 1 illustrates these definitions.

3. QUADTREE

To permit the rapid calculation of such things as nearest neighbors and clipped Voronoi cells, we use a point location data structure based on the balanced quadtree of Bern, Eppstein, and Gilbert [6]. In two dimensions, under the assumptions that every vertex coordinate is a b-bit integer and that arithmetic and bitwise operations on b-bit integers require constant time, earlier work [7] has shown how to construct a balanced quadtree on a set of n vertices in $O(n \log n)$ work and $O(\log n)$ parallel time on an EREW PRAM, assuming that $b \in O(\log n)$. We extend this construction to arbitrary dimensions, and describe how k *dynamic vertices* may be inserted into (or deleted from) an existing quadtree in $O(k \log \Delta)$ work and $O(\log \Delta)$ parallel time, with the assumption that $b \in O(\log \Delta)$, which states that we do not use much higher arithmetic precision than is necessary to distinguish the input points.

We define a d-dimensional balanced quadtree to be the *minimal* quadtree which satisfies the following properties:

- Crowding: every leaf node of the quadtree contains at most one vertex, and if it does, none of its neighbors contains a vertex.
- Grading: all neighbors of any internal node must exist in the quadtree.

Here, we define the *neighbors* of a quadtree node to be the nodes in each of the $3^d - 1$ cardinal and diagonal directions, at the same level. To support fast traversals and access, a quadtree node keeps pointers to its parent, children, and neighbors. Additionally, every leaf node, which we refer to as a *square*, contains a pointer to an input vertex it may contain, and a list of Steiner vertices. In a balanced quadtree satisfying these properties, the set of squares partitions the space defined by the quadtree in such a way that two adjacent squares (i.e, two squares that share a common border) are either neighbors, or at consecutive levels.

Our quadtree supports the `QTClippedVoronoi`(v, β) function, which returns the β-clipped Voronoi cell of v in $O(1)$ time [18] and the `QTInsert` and `QTDelete` operations for modifying the vertex set. Each takes as input a balanced quadtree on a set of vertices N, and a set N^* of k *dynamic vertices* to either insert into, or delete from, this quadtree. `QTInsert` and `QTDelete` return the set of leaf nodes of the original quadtree that are internal nodes of the resulting quadtree and the leaf nodes that are removed from the original quadtree respectively. Our main result about this data structure, which we give without proof due to space limitations, states that the `QTInsert` and `QTDelete` functions may be effectively parallelized.

THEOREM 3.1. *Given a set N^* of k vertices and a quadtree $\mathcal{Q}$, the functions `QTInsert` and `QTDelete` update the quadtree by performing $O(k \log \Delta)$ work in $O(\log \Delta)$ parallel time on an EREW PRAM.*

The `QTInsert` and `QTDelete` functions start by constructing, in Bern et al.'s terminology [7], a "framework" (a hypothetical quadtree) on N^*, the primary purpose of which is to define an "ownership" relation between dynamic vertices and quadtree nodes such that every node containing a dynamic vertex is "owned" by exactly one dynamic vertex. The functions then insert/delete the vertices N^* into/from the quadtree and finish by enforcing the crowding, grading and minimality properties in several passes, each of which is performed in parallel.

To repair the quadtree, we use k processors, each of which is identified with a dynamic vertex $p \in \mathsf{N}^*$. As we prove in lemmas 3.2 and 3.3, insertion or deletion of a vertex p only affects a local neighborhood of the quadtree nodes which contain p. Each processor is responsible for repairing the portion of the quadtree affected by its vertex, potentially in all b levels. The ownership relation defined by the framework, and a careful ordering on parallel operations defined by a coloring scheme, ensure exclusive accesses to memory, and that each affected node will be repaired by only one processor. By ensuring that there are a constant number of colors, and permitting no two operations of different colors to execute concurrently, these passes satisfy the requirements of an EREW PRAM model, with only a constant factor overhead in runtime.

Framework construction. The construction of the framework, which is performed at the start of every `QTInsert` and `QTDelete` operation, is very similar to the unbalanced quadtree construction of Bern et al. (section 2 of [7]).

We begin by sorting the dynamic vertices N^* in Morton's Z-order [23, 7]. Using this sorted list, we will define an auxiliary data structure for each vertex which contains the information needed by the framework.

We say that a dynamic vertex "owns" a quadtree square if it is the first dynamic vertex (in the sorted array N^*) which is contained within the boundaries of this square. Every dynamic vertex owns at least one square, but a vertex may own multiple squares–in particular, if it owns a square at level ℓ, then it owns a smaller square at every level $\ell' > \ell$. Similarly, we say that one dynamic vertex p is a "parent" of p' if p is the owner of the parent square (in the quadtree) of the minimum-level square owned by p'.

The framework keeps track of this parent-child relationship between vertices. For convenience, we will treat this data structure as a set of fields vertices themselves, although its lifetime is only that of the insertion or deletion operation, not of the quadtree itself:

- Level: Level (in the quadtree) of the minimum-level square owned by this vertex.
- Parent: Index of the parent vertex in the sorted list N^*.
- Nodes: Array of pointers, indexed by level, to the quadtree squares owned by this vertex. Initialized to *null*.

The level of the ith dynamic vertex is the number of high-order bits of the coordinates of $\mathsf{N}[i]$ and $\mathsf{N}[i-1]$ which are identical, plus one. The first vertex in the sorted list is at level 0. Once the levels have been found, the *parent* fields may be populated. One may verify that the parent index i of a vertex at index j is the largest $i < j$ such that $\mathsf{N}[i].level < \mathsf{N}[j].level$.

The final step in this construction is to "link up" this framework to the quadtree proper by populating the *nodes* fields. This is accomplished using a set of parallel processes, one for each dynamic vertex, which execute in lockstep.

Each iterates through the levels of the quadtree in a top-down manner (from the root towards the leaves), and finds the quadtree node owned by its dynamic vertex at the current level by inspecting the children (in the quadtree) of the node owned by its *parent* dynamic vertex at the previous level. If a `QTInsert` operation is being performed, then the new dynamic vertices are inserted into the appropriate leaf nodes once they are encountered, and these leaves split as necessary in order to ensure that each leaf node contains at most one vertex. If a `QTDelete` operation is being performed, then the dynamic vertices are removed from these leaf nodes, but no quadtree nodes are merged–minimality will be enforced later.

During this top-down pass, we use a coloring scheme (described in detail below) to ensure that concurrent memory accesses cannot occur. During a `QTDelete` operation, two processes could access the same memory location only if they simultaneously work on nodes which share a parent. The fact that `QTSplit` is called during a `QTInsert` operation slightly complicates things, since we must update neighbor pointers when nodes are split, making it necessary for us to choose a coloring scheme which guarantees that no two processes may concurrently work on nodes whose parents share a neighbor.

Crowding and Grading Passes. The sole purpose of the framework is to make it possible to iterate, in parallel, over the quadtree nodes owned by dynamic vertices. With the framework in place, we repair the quadtree in a series of top-down and bottom-up passes over the quadtree. During these passes, a distinct process is associated with each dynamic vertex. These processes then, in lockstep, iterate through first the levels, and then the colors. In parallel, each performs a local constant-time operation on a node which it owns at the current level, if it is of the current color.

The coloring scheme is of vital importance to ensuring that concurrent memory accesses cannot occur during a pass over the quadtree. We assign a color to each quadtree node by taking each coordinate of the node modulo κ. Two nodes at the same level which are of the same color will be at least $\kappa - 1$ squares away from each other. The following two lemmas show that a small constant κ (7, in fact) suffices, when enforcing the crowding and grading properties.

LEMMA 3.2. *Let φ be a node that is must be split due to crowding. Then there is a vertex $p \in \mathsf{N}^*$ that lies either inside φ or one of its neighbors.*

PROOF. Follows immediately from the definition of the crowding property. □

LEMMA 3.3. *(Lemma 1 of Moore [22]) Let φ be a node that is split due to grading. Then a descendant of one of its neighbors must have been split due to crowding.*

Due to space limitations, we omit details on the passes which are performed in order to enforce the crowding, grading and minimality properties after some number of insertions or deletions. Briefly, we proceed by first enforcing the crowding property in a single top-down pass. We then perform a bottom-up pass in which it is determined which nodes must be split or merged in order to satisfy the grading property and minimality, and finally a top-down pass in which these splits/merges are actually performed.

4. PARALLEL ALGORITHM

Given a set of vertices N, we can construct a ρ-well-spaced superset of N by repeatedly "filling" the vertices until the set becomes ρ-well-spaced. To fill a vertex v, we apply a *fill* operation to v that inserts Steiner vertices within the (ρ, β) picking region of v (Figure 1), making v ρ-well-spaced. This approach, while correct, is not efficient because we may fill vertices many times.

To create an efficient parallel algorithm, we first notice that the Steiner vertices inserted when filling a vertex are always at least ρ times the nearest neighbor distance from the vertex, and therefore that inserting a Steiner vertex does not affect the well-spacedness of those vertices with nearest neighbor distances less than ρ times that of the vertex being filled. We can thus partition the vertices into groups, called *ranks*, such that vertices with nearest neighbor distances within a factor of ρ of each other are in the same rank. The vertices will then be filled in rank order. We observe that the vertices in each rank need not be filled sequentially, because filling a vertex only affects a local neighborhood. This allows us to partition the vertices in each rank into a constant number of *colors*, in such a way that we can fill vertices of the same color in parallel, while sequentially ordering the vertices of different colors. Each processor maintains an independent queue of pending operations, and fills the vertices sequentially in rank and color order. We show that the ranks and colors can be carefully picked to ensure that all memory reads and writes are exclusive.

This partial ordering takes advantage of independence of operations. We make this independence concrete by recording dependencies between vertices in a computation graph. In order to handle batch dynamic updates to the input, we employ a change-propagation mechanism that updates parts of the computation affected by dynamic modifications.

Construction Algorithm. The construction algorithm (Figure 2) revolves around two operations, `Fill` and `Dispatch`. As was briefly described above, the `Fill` operation inserts Steiner vertices to make a vertex ρ-well-spaced. A `Dispatch` operation computes the rank of a vertex, which we define as the base ρ logarithm of its nearest neighbor distance, and keeps it up to date as Steiner vertices are inserted. We say both operations *act on* a vertex (the first argument). The unique dispatch operation acting on a vertex v runs before the fill operations acting on v, and schedules fill operations for the vertex and its β-clipped Voronoi neighbors at the current rank, ensuring correct ordering of fill operations. We define *time* as a triple consisting of a rank, a flag indicating a dispatch (`D`) or fill (`F`), and a color.

Given a set of input vertices N, we assign one processor to each vertex $p \in \mathsf{N}$, each of which locally maintains an operation array Ω indexed by time, and executes `ParallelWS` in order to construct a ρ-well-spaced superset of its input N. `ParallelWS` starts by constructing a quadtree in parallel by inserting vertices into an empty quadtree. It then enqueues a dispatch operation for its assigned vertex p and proceeds by, at each rank, iterating through each color, executing the dispatch operations for that color, and then doing the same for the rank's fill operations. Both sets of operations use κ_O colors; we discuss the choice of κ_O in detail later in this section. The dispatch and fill operations also modify the computation graph. Consider a dispatch or a fill operation acting on a vertex v at time t, represented by the node (v, t); this operation inserts edges $((v, t), (w, t_w))$ as described in

```
ParallelWS (p, N) =
  QTInsert(p, N, nil)
  t_p ← Enqueue(p, |square of p|, D, Ω)
  Add edge (p, 0) ⟶ (p, t_p)
  for r = rank of t_p to ⌊log_ρ √d⌋ do
    for c = 1 to κ_O^d do
      t ← (r, D, c)
      for each v ∈ Ω[t] do Dispatch(v, t, Ω)
    for c = 1 to κ_O^d do
      t ← (r, F, c)
      for each v ∈ Ω[t] do Fill(v, t, Ω)

Dispatch (v, t, Ω) =
  (u, CV) ← QTClippedVoronoi(v, β, t)
  t_v ← Enqueue(v, |vu|, F, Ω)
  Add edge (v, t) ⟶ (v, t_v)
  for each CV-neighbor w of v do
    t_w ← Enqueue(w, |wv|, F, Ω)
    Add edge (v, t) ⟶ (w, t_w)

Fill (v, t, Ω) =
  (u, CV) ← QTClippedVoronoi(v, β, t)
  while v is not ρ-well-spaced do
    Pick w ∈ CV s.t.  |vw| ≥ ρ · |vu|
    Insert w as a Steiner vertex
    t_w ← Enqueue(w, |wv|, D, Ω)
    Add edge (v, t) ⟶ (w, t_w)
    Update CV with w

Enqueue (v, nnv, T, Ω) =
  r_v ← ⌊log_ρ nnv⌋ ,  c_v ← color(v, r_v)
  t_v ← (r_v, T, c_v)
  if ∄ edge · ⟶ (v, t_v) then
    Ω[t_v] ← Ω[t_v] ∪ {v}
  return t_v
```

Figure 2: The pseudo-code of the parallel algorithm.

the pseudo-code. Here t_w is the time of the (potential) operation scheduled to act on w. In order to keep track of dependencies through the quadtree, a `QTClippedVoronoi` call executed by the operation acting on v records the node (v, t) in every square s that it accesses.

Dynamic Update Algorithm. We describe our parallel algorithm (pseudo-code in Figure 3) for updating the well-spaced output after a batch insertion/deletion of vertices into/from the input. Given a set of dynamic vertices N^* to be inserted or deleted, we assign one processor to each vertex $p \in \mathsf{N}^*$, each of which participates in the dynamic update by executing `Insert`/`Delete` and then `Propagate`. Each processor locally maintains three arrays of operations $\Omega^{\ominus}, \Omega^{\oplus}, \Omega^{\otimes}$ that hold (respectively) the obsolete, fresh, and inconsistent operations, which are (respectively) to be deleted, executed, and re-executed. The `Insert` and `Delete` functions take a set of vertices N^* along with the quadtree $\mathcal{Q}$, and insert/delete N^* into/from $\mathcal{Q}$, receiving a set of obsolete squares $\Sigma^{\ominus}$. Next, each enqueues the dispatch operation for the vertex p assigned to its processor into the obsolete or fresh operation arrays, and proceeds to execute `Propagate`. This function starts by inserting the readers of the obsolete squares into the array of inconsistent operations by iterating through each quadtree depth, using κ_S colors (details later in this section) in order to ensure that the operation queues of different processors are disjoint. It then enqueues an obsolete dispatch operation for each vertex contained in an obsolete square, and enqueues a fresh dispatch operation for the same vertex at the new square. After the initializations, `Propagate` proceeds in time order. At each rank, it starts by fixing the dispatch operations. Iterating through colors, `Fix` undoes the operations in the obsolete and inconsistent arrays, and updates the inconsistent operation array by removing the obsolete operations. It then finishes fixing the dispatch operations by performing those in the inconsistent and fresh arrays. Next, `Propagate` fixes the fill operations by having `Fix` undo and perform them in a similar fashion. For fill operations, `Fix` also marks readers of those squares whose Steiner vertex lists have changed to be inconsistent.

During the update, the dispatch and fill operations, as well as the undo operations, all maintain the computation graph. Undos remove the edges originating from the vertex on which these operations act. In order to propagate and repair the effects of inconsistencies that arise while inserting or removing Steiner vertices, we rely on the `MarkReaders` operation. `MarkReaders` marks dispatch and fill operations that are scheduled in the future, and whose clipped Voronoi computations access the inconsistent square, to be inconsistent themselves.

Coloring for Dynamism and Parallelism. To update the output when the input point set changes, our dynamic algorithm identifies the operations made inconsistent by the changes, and re-executes them. When an operation is re-executed, it can make another operation inconsistent by inserting a Steiner vertex. For efficient updates, it is therefore crucial that such dependency chains be short–of no more than logarithmic length. Since there are logarithmically many ranks, it suffices to ensure that the dependencies between operations in the same rank are of constant length. At any given rank r, both dispatches and fills access only the quadtree squares within a ball of radius $O(\rho^r)$. Their modifications to the computation graph are local as well, since they insert edges only towards vertices within this ball. This allows us to partition the work at a given rank into a constant number of color classes in such a way that dependencies occur only between the operations of different colors, guaranteeing efficient dynamic updates as well as exclusive reads and writes: operations at the same color class are independent, in that they neither access the same quadtree squares nor access the same nodes in the computation graph.

More formally, we define two squares of the same size to be *related* if there is a dispatch or fill operation that accesses both of them. We show that there exists a constant number of colors (κ_S^d) coloring squares of the same size such that no two squares are related if they have the same color (Lemma 5.2). This ensures that in the initialization of the dynamic update, different processors do not simultaneously mark operations reading an obsolete square to be inconsistent. We say that two dispatch or fill operations executed at the same rank *interfere* if the squares accessed by the operations are related. We disallow interference by using a *coloring scheme* that partitions the space based on a param-

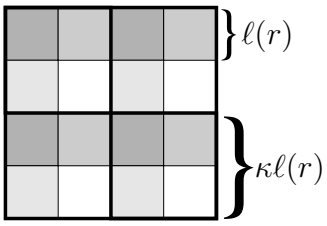

Figure 4: Illustration of a coloring scheme ($\kappa = 2$).

```
Fix (r, T) =
  for c = 1 to κ_O^d do
    t ← (r, T, c)
    for each v ∈ Ω^⊖[t] do Undo(v, t)
    Remove unflagged vertices from Ω^⊗[t]
    for each v ∈ Ω^⊗[t] do Undo(v, t)
    for each v ∈ Ω^⊕[t] ∪ Ω^⊗[t] do
      if T = D then Dispatch(v, t, Ω^⊕)
      if T = F then
        Fill(v, t, Ω^⊕)
        for each inserted Steiner w do
          MarkReaders(square of w, t)

Undo (v, t) =
  Unflag v at time t
  for each edge (v, t) ⟶ (w, t_w) do
    Remove edge (v, t) ⟶ (w, t_w)
    if ∄ edge · ⟶ (w, t_w) then
      Ω^⊖[t_w] ← Ω^⊖[t_w] ∪ {w}
    if t = (·, F, ·) then
      MarkReaders(square of w, t)
      Delete w from its square

MarkReaders (s, t) =
  for each v reading s at time t' > t do
    if v at time t' is not flagged then
      Flag v at time t'
      Ω^⊗[t'] ← Ω^⊗[t'] ∪ {v}

Insert (p, N*, Q) =
  Ω^⊖, Ω^⊕, Ω^⊗ ← ∅
  Σ^⊖ ← QTInsert(p, N*, Q)
  t_p ← Enqueue(p, |square of p|, D, Ω^⊕)
  Add edge (p, 0) ⟶ (p, t_p)

Delete (p, N*, Q) =
  Ω^⊖, Ω^⊕, Ω^⊗ ← ∅
  Σ^⊖ ← QTDelete(p, N*, Q)
  Remove edge (p, 0) ⟶ (p, ·)
  Enqueue(p, |square of p|, D, Ω^⊖)

Propagate (p, Σ^⊖) =
  for each quadtree depth ℓ do
    for c = 1 to κ_S^d do
      for each s ∈ Σ^⊖ at depth ℓ do
        if color of s is c then
          MarkReaders(s, 0)
  for each s ∈ Σ^⊖ and v ≠ p ∈ s do
    if v is an input vertex then
      Remove edge (v, 0) ⟶ (v, ·)
      Enqueue(v, |s|, D, Ω^⊖)
      t_v ← Enqueue(v, |square of v|, D, Ω^⊕)
      Add edge (v, 0) ⟶ (v, t_v)
  r_min ← min rank in Ω^⊖ ∪ Ω^⊕ ∪ Ω^⊗
  for r = r_min to ⌊log_ρ √d⌋ do
    Fix(r, D), Fix(r, F)
```

Figure 3: The pseudo-code of the parallel dynamic update algorithm.

eter κ_O and a real valued function $\ell(r)$ defined on ranks. At each rank r, we partition the space into d-dimensional hypercubes or *tiles* with side length $\ell(r)$. We color tiles such that they are colored periodically in each dimension with period κ_O, using κ_O^d *colors* in total. An operation at rank r that acts on a vertex v has color $c \in \{1, 2, \ldots, \kappa_O^d\}$ if v lies in a c colored tile. Figure 4 illustrates a coloring scheme in 2D. By choosing $\ell(r)$ small enough and κ_O large enough, we prove that two operations at the same rank do not interfere with each other if they have the same color (Lemma 5.3).

5. PARALLEL WORK AND DEPTH

We show that the work is efficiently distributed among the processors, of which there is one for each input vertex (construction), or each dynamic vertex being inserted or deleted (dynamic update). First, we prove that our parallel algorithms can be implemented on an EREW PRAM, i.e., the operations executed at any time step perform only exclusive reads and writes (Lemma 5.4). Then, we prove that a processor performs operations only on the vertices that are relatively close to its input vertex (Lemma 5.5). Taking advantage of this property, we prove that each processor spends $O(1)$ time at each time-step.

The design of the parallel algorithms we present in Section 4 is inspired by the algorithms developed by Acar et al. [4]. Some of their results are useful in proving the results we state here. In particular, they prove that the nearest neighbor distance of a vertex at rank r is bounded below by $\Omega(\rho^r)$ and that any operation acting on that vertex accesses a region within a ball of radius at most $O(\rho^r)$. These bounds allow us to show the existence of a constant size coloring scheme and that our algorithms are suitable for the EREW PRAM model. For construction and dynamic updates, the theorems on correctness follow similarly.

THEOREM 5.1. *The superset of points that* `ParallelWS` *constructs and that* `Insert` *and* `Delete` *maintains are ρ-well-spaced and size-optimal.*

PROOF. The major difference between our algorithms and the algorithms of Acar et al. is the order in which the vertices are processed, due to difference in the coloring schemes. This does not affect the correctness of our algorithms, so the correctness theorems of Acar et al. [4] continue to apply. □

LEMMA 5.2. *We can color the squares of a given size with $\kappa_S^d \in O(1)$ colors in such a way that two squares are not related if they have the same color.*

PROOF. Hudson and Türkoğlu prove that once the vertices the nearest neighbors of which are within distance $O(\rho^r)$ are ρ-well-spaced the clipped Voronoi cell computations only read squares s that have side length $|s| \in \Omega(\rho^r)$ and within distance $O(\rho^r)$ of v [18]. The balanced condition on the quadtree ensures that these squares have side lengths of size $O(\rho^r)$, thus, $\Theta(\rho^r)$. In other words, squares of a certain size ℓ can be read by operations at constantly many different ranks; let the maximum of those be r. Since any operation at rank r reads squares within a hypercube whose size (side length) is $O(\rho^r)$, in every dimension, a constant number of squares of size ℓ cover this hypercube. Setting κ_S to be this constant completes our proof. □

LEMMA 5.3. *There exists a coloring scheme with $\ell(r) \in \Omega(\rho^r)$ and $\kappa_O \in O(1)$ such that any two dispatch or fill operations executed at the same time step do not interfere.*

PROOF. Consider an operation *op* acting on v at rank r. Consider another operation op' acting on a vertex w at rank r' that reads a common square. Assuming that the set of squares op' reads is $\mathcal{S}$, we would like to show that there is no vertex $v' \neq v$ at the same time step as v that reads a square from $\mathcal{S}$. The arguments used in the proof of Lemma 5.2 show us $|ws| \in O(\rho^{r'}) = O(\rho^r)$ and $|vs| \in O(\rho^r)$. Using the triangle inequality, we get $|wv| \leq \alpha\rho^r$ for some constant α. Acar et al. proves that the nearest neighbor distance of v is bounded from below by $\Omega(\rho^r)$ [4], let $\ell(r)$ be this bound. To ensure independence, a coloring scheme with $(\kappa_O - 1)\ell(r) > 2\alpha\rho^r$ suffices because any vertex v' that could interfere with v has to be within $2\alpha\rho^r$ distance of v. Since $\ell(r) \in \Omega(\rho^r)$, there exists a coloring parameter $\kappa_O \in O(1)$ that satisfies the above inequality. □

LEMMA 5.4. *The `ParallelWS`, `Insert` and `Delete` functions perform exclusive reads and exclusive writes at every parallel step.*

PROOF. The `Insert` and `Delete` functions start by modifying the quadtree. Theorem 3.1 ensures that the quadtree modifications obey the restrictions of the EREW PRAM model. These functions then call `Propagate`. The initial loop iterates over each quadtree depth ℓ (or size) and square color c and marks the readers of the squares (at depth ℓ and color c) of every processor. Since the square lists $\Sigma^{\ominus}$ of each processor are disjoint, Lemma 5.2 ensures that two squares at the same depth and color are not related, guaranteeing exclusive memory accesses. The next loop satisfies the EREW conditions by disjointness of the squares.

We are left to prove that the main loop performs operations in an exclusive manner. We show this in two parts: first we prove that two distinct operations executing at the same time do not interfere with each other, second that multiple processors do not try to execute the same operation. For the first part, we observe that fill, dispatch, and undo operations only read squares visited by its `QTClippedVoronoi` call and may only enqueue operations that read a common square. The definition of operation interference captures this observation: two independent operations cannot read or write into the same memory locations. Lemma 5.3 shows the existence of a coloring scheme that enables us to process only independent operations at each time step.

The second part is more technical. In order to ensure that no two distinct processors enqueue the same operation into their schedule, we check the existence of edges in the computation graph. Consider an operation *op* acting on w at time t_w that is already scheduled in $\Omega^{\oplus}$. We guarantee that no other processor schedules the same vertex at the same time by checking incoming edges onto (w, t_w). If one exists, we do not schedule. For undo operations, we follow a similar pattern, and do not schedule an operation acting on w at time t_w to be undone until all edges towards (w, t_w) are cleared. Hence, obsolete and fresh operation arrays on all processors contain at most one copy of any vertex. For the inconsistent lists we ensure the same property using flags: the only function that enqueues into inconsistent lists makes sure that the same vertex is not scheduled into two different inconsistent lists. These arguments complete our proof. □

LEMMA 5.5. *Given the operation schedule Ω associated with an input vertex p, each vertex v scheduled in Ω at rank r satisfies $|vp| \in O(\rho^r)$.*

PROOF. We prove our claim using induction on the order our algorithm enqueues vertices into the schedule Ω. For each rank r, we show that there exists a constant α such that for every vertex v scheduled at rank r, we have $|vp| < \alpha\rho^r$. Initially, a vertex v scheduled in Ω either lies inside a square in $\Sigma^{\ominus}$ or there is an operation acting on it reading a square from $\Sigma^{\ominus}$. Using the Lemmas 3.2 and 3.3 and the fact that each operation at rank r reads squares within $O(\rho^r)$ distance, we prove the base case, that $|vp| \in O(\rho^r)$. For the inductive step, we assume that any vertex v at rank $r' < r$ is within $\alpha\rho^{r'}$ distance of p. By the locality of the operations, we know that any vertex scheduled for a dispatch or a fill operation at rank r is within distance $O(\rho^r)$ of a vertex in Ω at an earlier rank $r' < r$ or at rank r but an earlier color. Since there is a constant number of colors, any vertex v is within distance $O(\rho^r)$ of another vertex w at rank $r' < r$; let α' be the constant in the asymptotic notation. By inductive hypothesis, we have $|wp| < \alpha\rho^{r'}$. Setting $\alpha = \rho\alpha'/(\rho - 1)$, and using the triangle inequality, we prove that $|vp| < \alpha\rho^r$. □

THEOREM 5.6. *Given a set of k vertices N^*, `Insert` and `Delete` update the previous input N to N' by inserting or deleting N^*, and update the previous output to a size-optimal ρ-well-spaced superset of N' in $O(k \log \Delta)$ work and $O(\log \Delta)$ parallel time on an EREW PRAM.*

PROOF. Theorem 3.1 shows that the quadtree can be modified in $O(\log \Delta)$ parallel time using k processors on an EREW PRAM. Lemma 5.4 shows that the `Insert` and `Delete` algorithms can be implemented on an EREW PRAM without extra overhead. For the processor associated with a vertex $p \in \mathsf{N}^*$, Lemma 5.5 shows that any vertex $v \in \Omega$ at rank r is of distance $O(\rho^r)$ away from p. Acar et al. prove a lower bound on the nearest neighbor distance of a vertex v at rank r: $\mathrm{NN}_{\mathsf{M}}(v) \in \Omega(\rho^r)$ [4]. Thus, a packing argument bounds the number of the vertices in Ω at rank r by a constant. Furthermore, Acar et al. prove that each vertex can be processed in $O(1)$ time. The fact that there are $O(\log \Delta)$ ranks proves our claim on parallel runtime. Observing that there are k processors completes the proof. □

6. A PRACTICAL ALGORITHM

The algorithm that we present in Section 4 assumes an execution model in which processors execute each instruction in lockstep; in practice, this requires global synchronization after every instruction. The algorithm also relies on a large number of colors to ensure exclusive accesses to memory. Although we prove (section 5) the existence of a constant number of colors guaranteeing independence, thus ensuring a high degree of parallelism, we believe the constants may be too large for many practical input sizes. It is therefore not clear if the EREW algorithm can be implemented efficiently on actual hardware, such as contemporary multi-core computers, on which asymptotic benefits may not be realized.

In this section, we describe some key ingredients of a practical implementation of our algorithms on contemporary multi-core machines. We identified the modifications that would be most helpful in attaining such an algorithm by implementing a sequential version of our EREW algorithm, profiling it extensively, and developing an implementation

that is tailored not to the asymptotic case, but instead to commodity multi-core machines, and to real-world datasets sampled from the literature. Broadly, the modifications consist of a simplification that permits sequentializing much of the algorithm, elimination of EREW requirements in favor of judicious use of locks, and establishment of a trade-off between number of colors and the parallelism, enabling us to use far fewer colors.

Sequential Operations and Locks. Our EREW algorithm prevents all concurrent reads and writes. On multi-core machines, concurrent reads cause no problems, although concurrent writes must be prevented. This can be accomplished through the use of locks, especially when data is accessed by a small number of processors, each of which holds the lock briefly. Simple experiments with a single-threaded prototype implementation indicate that upwards of 70% of program runtime is spent performing geometric computations in 2D, and significantly more in 3D. The remaining runtime is split between quadtree construction, operation queues, the computation graph, and other bookkeeping tasks. Hence, our primary goal must be to effectively distribute `Dispatch` and `Fill` operations, which perform these geometric operations, across processors. Unless there are a very large number of processors, the vital portions of the minor components of the algorithm may be protected with locks, or even performed sequentially. The `QTInsert` and `QTDelete` quadtree operations fit into this category, since although their theoretical cost is asymptotically significant, it may be, in practical terms, neglected. These operations may therefore be performed sequentially. The computation graph is also inexpensive enough to maintain that it may be protected by a single global lock.

There is little practical benefit to explicitly partitioning pending operations across processors. Instead, it is simplest to use a global priority queue for the `Dispatch` and `Fill` operations, protected by a lock. In addition to simplifying the algorithm, this approach postpones the decision of on which processor each operation will be scheduled from when it is created, to when it is performed, potentially resulting in a better-balanced workload.

The lists of Steiner vertices which are maintained on the leaves of the quadtree may be accessed concurrently by multiple `Fill` operations, and must therefore be protected with locks. The locality of operations (lemma 5.5) implies that only "nearby" operations could potentially access the same list, and therefore that there will be little contention. Our experiments confirm this intuition.

Coloring Scheme. The EREW algorithm uses colors to ensure exclusive memory accesses. Since `Dispatch` operations write only to the computation graph, which can be protected with a lock, we can perform all dispatches at each rank in parallel–no coloring scheme is necessary. In contrast, since `Fill` operations insert Steiner vertices, the order in which they are executed will affect their results, implying that they cannot be executed entirely in parallel. We therefore continue to use a coloring scheme to ensure correctness.

The coloring scheme we propose is a variant of that described in section 4. In both schemes, the goal is to identify independent operations so that they may be executed in parallel, although the definition of "independent" is subtly different. For the EREW algorithm, we ensure that no two processes access related quadtree squares. In practice, the computation graph and the quadtree squares are protected by locks, so it is sufficient to guarantee that two `Fill` operations, executing in parallel, cannot affect each others' Steiner vertex choices. Specifically, if there are two concurrently-executing `Fill` operations acting on vertices u and v at rank r, then that acting on u may not insert a Steiner vertex which would be a β-clipped Voronoi neighbor of v (and vice-versa). Because any Steiner vertex inserted by the operation acting on u will be inserted within a radius of $\beta\rho^{r+1}$, and all β-clipped Voronoi neighbors of v must be within $2\beta\rho^{r+1}$, it suffices to ensure that concurrently-executing `Fill` operations are more than $3\beta\rho^{r+1}$ apart.

As in section 4, we partition space into a grid of tiles of side length $\ell(r)$. The "color" of each tile will be determined by taking its coordinates modulo κ_O, and the "color coordinate" by taking the integer quotients with κ_O. The "color" and "color coordinate" of an operation are those of the tile containing the vertex on which the operation acts. While, for the EREW algorithm, $\ell(r)$ was chosen to be small enough that each tile could contain at most one operation, we will, for reasons which will be explained shortly, here relax this restriction. Instead, if there are multiple operations scheduled at the same color and color coordinate (i.e. within the same tile), then they will be executed sequentially by the same processor. As before, operations of different colors will be executed sequentially, and operations of the same color *but different color coordinates* will be executed in parallel. To ensure independence, we must choose $(\kappa_O - 1)\ell(r) > 3\beta\rho^{r+1}$.

The operation queues are prioritized by (in order), rank, color and color coordinate. At each rank and color, each processor locks the queue, and takes possession of the top operations of the same color coordinate. After releasing the lock, it then performs these operations sequentially. This approach comes at a cost, in terms of parallelism: the larger $\ell(r)$ is, the more steps must be taken while sequentially executing all of the operations within one tile. Conversely, the larger κ_O is, the more colors there will be, and the more sequential steps will be performed while iterating over the possible colors. Experimentally, we have found that the latter consideration wins out. When finding well-spaced supersets of 2D and 3D uniformly random datasets of 10000 points, the parallel depth of the computation increases monotonically with κ_O for all $\kappa_O > 3$. For this reason, in our implementation, we fix $\kappa_O = 3$.

7. EXPERIMENTS

We have implemented two versions of the proposed algorithms (Section 6), one sequential and one parallel, and performed an experimental analysis using both synthetic and real datasets with parameters $\rho = \sqrt{2}$, and $\beta = 2$ in 2D or $\beta = 2\sqrt{2}/\sqrt{3}$ in 3D. Our experiments confirm our asymptotic bounds and give strong evidence that they can be realized efficiently in practice.

The Implementations. The sequential implementation, while computing the well-spaced superset of the input set, also calculates the work and depth of the computation. Although experiments with the sequential implementation confirm that our asymptotic bounds apply to real-world data, they give no indication as to what the constant factors may be. To resolve this, we have completed a multi-threaded implementation[1] along the lines of that described in Section 6.

[1] `http://nagoya.uchicago.edu/~cotter/projects/dynamic_wsp`

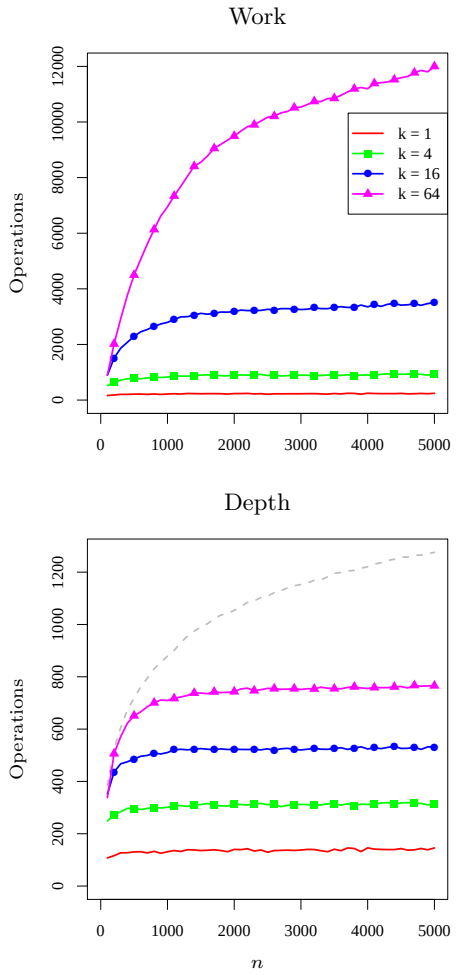

Figure 5: Measured work and depth versus input size n, for two sets of k changes. The dashed curve in the depth plot is twice the depth of a from-scratch run, and is an upper bound on the depth due only to `Dispatch` and `Fill` operations, neglecting undos.

Work and Depth. The *work* and *depth* numbers reported by the sequential implementation are the numbers of operations executed in total, and along the maximum length path in the computation. Alternatively, they measure the single processor and multiprocessor (on an idealized infinitely-parallel machine) times, respectively. Figure 5 shows the work and depth which result from erasing k random vertices from an existing well-spaced point set of size n, updating the set of Steiner vertices, then inserting k vertices, and updating again. We average over 16 runs on each of 16 different size-n input vertex sets (for a total of 256 runs per plotted point). All vertices have floating-point coordinates, and are chosen uniformly at random from the unit square, in two dimensions. The plot on the left shows that the total work required for these two sets of size-k changes is roughly proportional to our $O(k \log \Delta)$ bound (Theorem 5.6). The right hand plot shows that the depths resulting from these changes do not appear to meet our expected $O(\log \Delta)$ upper bound–there appears to be a k dependence. The reason for this is that our upper bound is tight only when operations are scheduled at all ranks and colors. In practice, the proportion of ranks and colors which are occupied decreases as k decreases, resulting in observed depths outperforming the upper bound by increasing amounts for smaller k. The relative magnitudes of the work and depth numbers show that there is significant parallelism in this algorithm even with the relatively small inputs considered.

Table 1 shows results of computations of well-spaced supersets of a number of well-known 2D and 3D datasets, all of which are significantly larger than the synthetic datasets of the previous experiments. "Simulated" work and depth numbers are calculated as before. On these real-world datasets, there appears to be significant available parallelism–the minimum work-to-depth ratio is 60 (on the Stanford bunny).

Timings and Speedups. Using our parallel implementation, we have measured the actual run-times for finding well-spaced supersets of each of these real datasets, which are also reported in Table 1. Our testing machine has an Intel Core i5 750 CPU, 8G of memory, and runs Ubuntu 10.04. This is a four-core processor.

In the first set of experiments, we calculated a well-spaced point set for each dataset, from scratch, for each number of threads, and averaged the wall-clock times over 10 runs. Each "Speedups" column reports the ratio of the average time taken by the t-thread runs to that of the 1-thread runs.

Our second set of experiments were designed to test the multi-threaded performance of localized dynamic changes. In these experiments, we first created a well-spaced point set on all of the points of a dataset *except* for those within a random ball of radius 0.01 in 2D or 0.1 in 3D. The "k" column in Table 1 contains the number of points in each of these balls. We then measured the time required to insert the "missing" points and update the well-spaced point set on each number of threads, averaged over 10 runs. The reported speedups are the ratios of these average runtimes to that required to find a well-spaced point set from scratch on a single thread. They therefore measure the performance improvement due *both* to dynamism and parallelism. The performance improvement due *only* to dynamism is the 1-thread speedup, while the parallel speedups of the dynamic algorithm may be recovered by taking the ratios of the k-thread and 1-thread speedups. The tested changes are fairly large, because our focus is on the performance impact of parallelism. There is an appealing synergy between dynamism and parallelism when viewed from a performance standpoint: dynamic updates are most efficient when the change set is small and the dataset is two dimensional, whereas parallel speedups are most pronounced when the change set is large and the dataset is three dimensional.

8. CONCLUSION

We presented a parallel and dynamic algorithm for well-spaced point sets, a fundamental problem directly related to meshing in 2D and 3D. The algorithm combines the best characteristics of parallel and dynamic algorithms: as a parallel algorithm, it delivers fast response on parallel computers, and as a dynamic algorithm, it does so by performing minimally necessary work by taking advantage of the similarity between inputs. When the input is modified by k insertions and/or deletions the algorithm performs $O(k \log \Delta)$ work and requires $O(\log \Delta)$ time using k processors on an EREW PRAM. We also presented an adaptation of the algorithm for modern multi-core systems and experimental results suggesting that the algorithm can be made practical.

Table 1: Simulated work and depth numbers, and actual speedups observed, on real-world data. The "Speedups" columns show the parallel speedups observed when calculating well-spaced supersets from scratch. The "Dynamic Speedups" columns show the speedups observed when performing k localized dynamic insertions relative to the time for calculating a well-spaced superset from scratch using a single thread.

Application			Simulated		Speedups		Dynamic Speedups			
Data Set	d	n	Work	Depth	2 cores	4 cores	k	1 core	2 cores	4 cores
New Zealand	2	18595	194838	1650	1.7×	2.9×	463	49×	74×	98×
Cape Cod	2	20930	170588	1330	1.7×	2.9×	83	144×	188×	200×
Lake Superior	2	33487	318484	1622	1.7×	2.8×	296	102×	150×	189×
SF Bay	2	85910	681506	2000	1.7×	2.7×	322	196×	299×	408×
Stanford Bunny	3	35947	289860	4779	1.9×	3.6×	657	9.3×	16×	26×
Armadillo	3	172974	1214280	7539	1.9×	3.6×	4242	14×	25×	42×

9. REFERENCES

[1] U. A. Acar. *Self-Adjusting Computation*. PhD thesis, Department of Computer Science, Carnegie Mellon University, May 2005.

[2] U. A. Acar, G. E. Blelloch, M. Blume, and K. Tangwongsan. An experimental analysis of self-adjusting computation. In *Programming Language Design and Implementation*, 2006.

[3] U. A. Acar, G. E. Blelloch, K. Tangwongsan, and D. Türkoğlu. Robust kinetic convex hulls in 3D. In *European Symposium on Algorithms*, September 2008.

[4] U. A. Acar, A. Cotter, B. Hudson, and D. Türkoğlu. Dynamic well-spaced point sets. In *SCG '10: the 26th Annual Symposium on Computational Geometry*, 2010.

[5] U. A. Acar, B. Hudson, and D. Türkoğlu. Kinetic mesh refinement in 2d. In *SCG '11: the 27th Annual Symposium on Computational Geometry*, 2011.

[6] M. Bern, D. Eppstein, and J. R. Gilbert. Provably Good Mesh Generation. *J. Computer and System Sciences*, 48(3):384–409, 1994.

[7] M. W. Bern, D. Eppstein, and S.-H. Teng. Parallel construction of quadtrees and quality triangulations. *International Journal of Computational Geometry and Applications*, 9(6):517–532, 1999.

[8] J.-D. Boissonnat, O. Devillers, R. Schott, M. Teillaud, and M. Yvinec. Applications of random sampling to on-line algorithms in computational geometry. *Discrete Computional Geometry*, 8:51–71, 1992.

[9] S.-W. Cheng, T. K. Dey, H. Edelsbrunner, M. A. Facello, and S.-H. Teng. Silver exudation. *J. ACM*, 47(5):883–904, 2000.

[10] L. P. Chew. Guaranteed-quality triangular meshes. Technical Report TR-89-983, Department of Computer Science, Cornell University, 1989.

[11] Y.-J. Chiang and R. Tamassia. Dynamic algorithms in computational geometry. *Proceedings of the IEEE*, 80(9):1412–1434, 1992.

[12] K. L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Computational Geometry Theory and Application*, 3:185–212, 1993.

[13] C. Demetrescu, I. Finocchi, and G. Italiano. *Handbook on Data Structures and Applications*, chapter 36: Dynamic Graphs. 2005.

[14] L. Guibas. Modeling motion. In J. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, pages 1117–1134. Chapman and Hall/CRC, 2nd edition, 2004.

[15] M. A. Hammer, U. A. Acar, and Y. Chen. CEAL: A C-based language for self-adjusting computation. In *Programming Language Design and Implementation*, June 2009.

[16] B. Hudson, G. Miller, and T. Phillips. Sparse voronoi refinement. In *Proceedings of the 2006 International Meshing Roundtable*, 2006.

[17] B. Hudson, G. Miller, and T. Phillips. Sparse Parallel Delaunay Mesh Refinement. In *SPAA*, 2007.

[18] B. Hudson and D. Türkoğlu. An efficient query structure for mesh refinement. In *Canadian Conference on Computational Geometry*, 2008.

[19] H. Jung and K. Mehlhorn. Parallel algorithms for computing maximal independent sets in trees and for updating minimum spanning trees. *Inf. Process. Lett.*, 27:227–236, April 1988.

[20] R. Ley-Wild, U. A. Acar, and M. Fluet. A cost semantics for self-adjusting computation. In *Principles of Programming Languages*, 2009.

[21] G. L. Miller, D. Talmor, S.-H. Teng, N. Walkington, and H. Wang. Control Volume Meshes Using Sphere Packing: Generation, Refinement and Coarsening. In *Fifth Intl. Meshing Roundtable*, pages 47–61, 1996.

[22] D. Moore. The cost of balancing generalized quadtrees. In *SMA '95: symposium on Solid modeling and app.*, pages 305–312, New York, NY, USA, 1995. ACM.

[23] G. M. Morton. A computer oriented geodetic data base; and a new technique in file sequencing. Technical report, IBM, Ottowa, CA, 1966.

[24] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms*. Prentice Hall, 1994.

[25] S. Pawagi and O. Kaser. Optimal parallel algorithms for multiple updates of minimum spanning trees. *Algorithmica*, 9:357–381, 1993.

[26] O. Schwarzkopf. Dynamic maintenance of geometric structures made easy. In *32nd Annual Symposium on Foundations of Computer Science*, pages 197–206, October 1991.

[27] D. Talmor. *Well-Spaced Points for Numerical Methods*. PhD thesis, Carnegie Mellon University, August 1997. CMU-CS-97-164.