Parallelism in Randomized Incremental Algorithms

Guy E. Blelloch
Carnegie Mellon University
guyb@cs.cmu.edu

Yan Gu
Carnegie Mellon University
yan.gu@cs.cmu.edu

Julian Shun
UC Berkeley
jshun@eecs.berkeley.edu

Yihan Sun
Carnegie Mellon University
yihans@cs.cmu.edu

ABSTRACT

In this paper we show that many sequential randomized incremental algorithms are in fact parallel. We consider several random incremental algorithms including algorithms for comparison sorting and Delaunay triangulation; linear programming, closest pair, and smallest enclosing disk in constant dimensions; as well as least-element lists and strongly connected components on graphs.

We analyze the dependence between iterations in an algorithm, and show that the dependence structure is shallow for all of the algorithms, implying high parallelism. We identify three types of dependences found in the algorithms studied and present a framework for analyzing each type of algorithm. Using the framework gives work-efficient polylogarithmic-depth parallel algorithms for most of the problems that we study. Some of these algorithms are straightforward (e.g., sorting and linear programming), while others are more novel and require more effort to obtain the desired bounds (e.g., Delaunay triangulation and strongly connected components). The most surprising of these results is for planar Delaunay triangulation for which the incremental approach is by far the most commonly used in practice, but for which it was not previously known whether it is theoretically efficient in parallel.

1. INTRODUCTION

The randomized incremental approach has been an extremely useful paradigm for generating simple and efficient algorithms for a variety of problems. There have been dozens of papers on the topic (e.g., see the surveys [57, 48]). Much of the early work was in the context of computational geometry, but the approach has more recently been applied to graph algorithms [20, 23]. The main idea is to insert elements one-by-one in random order while maintaining a desired structure. The random order ensures that the insertions are somehow spread out, and worst-case behaviors are unlikely.

The incremental process would appear sequential since it is iterative, but in practice incremental algorithms are widely used in parallel implementations by allowing some iterations to start in parallel and using some form of locking to avoid conflicts. Many parallel implementations for Delaunay triangulation and convex hull, for example, are based on the randomized incremental approach [17, 26, 18, 46, 5, 33, 15, 50, 59]. In theory, however, after 25 years, there are still no known bounds for parallel Delaunay triangulation using the incremental approach, nor for many other problems.

In this paper we show that the incremental approach for Delaunay and many other problem is actually parallel, at least with the right incremental algorithms, and leads to work-efficient polylogarithmic-depth (time) algorithms for the problems. The results are based on analyzing the dependence graph. This technique has recently been used to analyze the parallelism available in a variety of sequential algorithms, including the simple greedy algorithm for maximal independent set [7], the Knuth shuffle for random permutation [60], greedy graph coloring [40], and correlation clustering [49]. The advantage of this method is that one can use standard sequential algorithms with modest change to make them parallel, often leading to very simple parallel solutions. It has also been shown experimentally that this approach leads to quite practical parallel algorithms [6], and to deterministic parallelism [12, 6].

The contributions of the paper can be summarized as follows.

1. We describe a framework for analyzing parallelism in randomized incremental algorithms, and give general bounds on the depth of algorithms with certain dependence probabilities (Section 2).
2. We show that randomly ordered insertion into a binary search tree is inherently parallel, leading to an almost trivial comparison sorting algorithm taking $O(\log n)$ depth and $O(n \log n)$ work (i.e., n processors), both with high probability on the priority-write CRCW PRAM (Section 3). Surprisingly, we know of no previous description and analysis of this parallel algorithm.
3. We propose a new randomized incremental algorithm for planar Delaunay triangulation, and then describe a simple way to parallelize it (Section 4). The algorithm takes $O(\log^2 n)$ depth with high probability, and $O(n \log n)$ work (i.e., $n/\log n$ processors) in expectation, on the CRCW PRAM. It would seem to be by far the simplest work-efficient parallel Delaunay triangulation algorithm.
4. We show that classic sequential randomized incremental algorithms for constant-dimensional linear programming, closest pair, and smallest enclosing disk can be parallelized (Section 5). This leads to very simple linear-work and polylogarithmic-depth randomized parallel algorithms for all three problems.
5. For two incremental graph algorithms, instead of forcing the parallel version to abide by all dependences, we allow some dependences to be violated, but show that this does not asymptotically increase work, or change the result. We use this approach for a sequential algorithm for computing least-element lists, leading to an efficient parallel implementation (Section 6.1). Least-element lists have applications for probabilistic tree embeddings on graph metrics [30, 10], and estimating neighborhood sizes in

SPAA'16, July 11–13, 2016, Pacific Grove, California, USA.

DOI: http://dx.doi.org/10.1145/2935764.2935766

Problem	Work (expected)	Depth (with high probability)
Comparison sorting (Section 3)	$O(n \log n)$	$O(\log n)$
Planar Delaunay triangulation (Section 4)	$O(n \log n)$	$O(\log^2 n)$
2D linear programming (Section 5.1)	$O(n)$	$O(\log n)$
2D closest pair (Section 5.2)	$O(n)$	$O(\log n \log^* n)$
Smallest enclosing disk (Section 5.3)	$O(n)$	$O(\log^2 n)$
Least-element lists (Section 6.1)	$O(W_{SP}(n,m) \log n)$	$O(D_{SP}(n,m) \log n)$
Strongly connected components (Section 6.2)	$O(W_R(n,m) \log n)$	$O(D_R(n,m) \log n)$

Table 1: Work and depth bounds for our parallel randomized incremental algorithms. $W_{SP}(n,m)$ and $D_{SP}(n,m)$ denote the work and depth, respectively, of a single-source shortest paths algorithm. $W_R(n,m)$ and $D_R(n,m)$ denote the work and depth, respectively, of performing a reachability query.

graphs [21]. We also use the approach to parallelize an elegant algorithm for strongly connected components [23] (Section 6.2).

Other than the graph algorithms, which call subroutines that are known to be hard to parallelize, all of our solutions are work-efficient and run in polylogarithmic depth (time). The bounds for all of our parallel randomized incremental algorithms can be found in Table 1.

Preliminaries

We analyze parallel algorithms in the work-depth paradigm [42]. An algorithm proceeds in a sequence of D (depth) rounds, with round i doing w_i work in parallel. The total work is therefore $W = \sum_{i=1}^{D} w_i$. We account for the cost of allocating processors and compaction in our depth. Therefore the bounds on a PRAM with P processors is $O(W/P + D)$ time [14]. We use the concurrent-read and concurrent-writes (CRCW) PRAM model. By default, we assume the arbitrary-write CRCW model, but when stated use the priority-write model. We say $O(f(n))$ ***with high probability (w.h.p.)*** to indicate $O(kf(n))$ with probability at least $1 - 1/n^k$.

2. ITERATION DEPENDENCES

An ***iterative algorithm*** is an algorithm that runs in a sequence of ***steps*** (iterations) in order. When applied to a particular input, we refer to the computation as an ***iterative computation***. Each step i of an iterative computation does some work $W(i)$, and has some depth $D(i)$ (the steps themselves can be parallel). Step j is said to ***depend*** on step $i < j$ if the computation of step j is affected by the computation of step i. The particular dependences, or even the number of steps, can be a function of the input, and can be modeled as a directed acyclic graph (DAG)—the steps ($I = 1, \ldots, n$) are vertices and dependences between them are arcs (directed edges).

DEFINITION 1 (ITERATION DEPENDENCE GRAPH [60]). *An **iteration dependence graph** for an iterative computation is a (directed acyclic) graph $G(I, E)$ such that if every step $i \in I$ runs after all predecessor steps in G have completed, then every step will do the same computation as in the sequential order.*

We are interested in the depth (longest directed path) of iteration dependence graphs since shallow dependence graphs imply high parallelism—at least if the dependences can be determined online, and depth of each step $D(i)$ can be appropriately bounded. We refer to the depth of the DAG as the ***iteration depth***, and denote it as $D(G)$. In this paper, we are interested in probabilistic bounds on the iteration depth over random input orders.

An ***incremental algorithm*** is an iterative algorithm that maintains some property over elements while ***inserting*** a new element on each step. We will use $E = \{e_1, \ldots, e_n\}$ to indicate the insertion order of n elements. A ***randomized incremental algorithm*** is an incremental algorithm in which the elements are added in a uniformly random order. In randomized incremental algorithms, the presence of a dependence arc between steps i and j will have a probability p_{ij} based on all possible orders (each of the $n!$ orders is a primitive event in the sample space). We are interested in upper bounds on these probabilities, which we will refer to as $\hat{p}_{ij}$. A subtle point is that the exact probabilities p_{ij} are sometimes not independent (e.g., along a path), but the upper bounds $\hat{p}_{ij}$ are, allowing them to be multiplied. We will use backwards analysis [57]—we consider "removing" randomly selected elements one at a time from the end, noting that the analysis of elements $1, \ldots, i$ does not depend on the elements $j > i$.

In this paper, we consider three types of incremental algorithms, which we refer to as Type 1, 2, and 3, for lack of better names.

Type 1 Algorithms. In these algorithms we analyze the dependence depth by considering all possible paths in the iteration dependence graph and taking a union bound over the probability of each. We describe two algorithms of this type—sorting by insertion into a binary search tree, and incremental planar Delaunay triangulation. In the algorithms (and indeed in just about all incremental algorithms) inserting an element j between two elements $i < j$ and $k > j$ will never add a dependence between i and k (although it might remove one). The property means that we only need to consider the dependence between positions i and $i+1$ when calculating an upper bound on the probability $\hat{p}_{ij}$ ($j > i$). In particular, for all $j \geq i+1$ we use $\hat{p}_{i(i+1)} \geq \hat{p}_{ij} \geq p_{ij}$. We use the following lemma.

LEMMA 2.1. *Consider an iteration dependence graph G of n iterations with $\hat{p}_{ij} = f(i) \geq 1/n$, independent along any path, then*

$$\Pr(D(G) \geq l) < n\left(\frac{e\sum_{i=1}^{n} f(i)}{l}\right)^l$$

PROOF. Consider a path of length $l-1$, and let $K \subseteq \{1, \ldots, n\}$ be the vertices on the path ($|K| = l$). We have that the probability of the path existing is upper bounded by:

$$P(K) = n \prod_{k \in K} f(k) \,.$$

The multiplicative factor of n is needed to account for the fact that the last element of Kdoes not contribute to the probability of the path and can be as small as $1/n$. We can now take the union bound over all possible paths of length $l-1$, giving:

$$\Pr(D(G) > l-1) \leq X(G) = \sum_{K \subseteq \{1,\ldots,n\}, |K|=l} P(K)$$

If $f(i)$ is a constant with value $\hat{p}$ we have:

$$X(G) = \binom{n}{l} n\hat{p}^l < n\left(\frac{en\hat{p}}{l}\right)^l = n\left(\frac{e\sum_{i=1}^{n} f(i)}{l}\right)^l$$

where we use the inequality $\binom{n}{m} < \left(\frac{en}{m}\right)^m$.

We now show that unequal (non-constant) probabilities that maintain the same sum $\sum_{i=1}^{n} f(i)$ will only reduce $X(G)$ and hence the upper bound on $\Pr(D(G) \geq l)$. Therefore the probability is maximized by the equation above. Consider two locations i and j such that $f(i) \neq f(j)$. We show that changing these probabilities both to $\hat{p}_m = (f(i) + f(j))/2$ will increase $X(G)$. A path will either go through i but not j, j but not i, neither or both. Clearly the ones through neither will not affect the total sum. For every path through just i there is a path through just j going through the same set of other vertices. If P_r is the product of probabilities of the other vertices in one of these pairs of paths then the contribution to the union bound of both is $f(i)P_r + f(j)P_r = 2\hat{p}_m P_r$. The contribution from these paths is therefore not changed by changing $f(i)$ and $f(j)$ to $\hat{p}_m$. However, the contribution from paths going through both will increase since the old product is $f(i)f(j)P_r$ while the new one is $\hat{p}_m^2 P_r$, which has to be at least as large. □

COROLLARY 2.2. *Consider an iteration dependence graph G of n iterations with $\hat{p}_{ij} \leq c/i$, independent along any path. Then for any $k \geq 2ce^2$ we have*

$$\Pr(D(G) > k \ln n) \in O(1/n^{k-1}) .$$

PROOF. Plugging into Lemma 2.1 gives:

$$\begin{aligned} \Pr(D(G) \geq k \ln n) &< n\left(\frac{ec\sum_{i=1}^{n} 1/i}{k \ln n}\right)^{k \ln n} \\ &< n\left(\frac{ec(1+\ln n)}{k \ln n}\right)^{k \ln n} \\ &\leq n(1/e)^{k \ln n} = 1/n^{k-1} \quad \square \end{aligned}$$

To apply the previous lemma or corollary requires showing independence of the upper $\hat{p}_{ij}$ along every path. For sorting this is easy. For Delaunay triangulation the probabilities are not independent among the iterations corresponding to points, but we divide the iterations into sub-iterations, corresponding to the creation of triangles, for which they are independent.

The Type 1 algorithms that we describe can be parallelized by running a sequence of rounds. Each round checks all remaining steps to see if their dependences have been satisfied and runs the steps if so. The algorithms require at most $O(n)$ work per round to check violations. By Theorem 2.2, the number of rounds will be $O(\log n)$ w.h.p. The total expected work is therefore $O(n \log n)$ for the checks, plus the work for the steps, which is the same as for the sequential variants—$O(n \log n)$ in expectation. The total work is therefore $O(n \log n)$ in expectation.

Type 1 incremental algorithms can be implemented in two ways: one completely online, only seeing a new element at the start of each step, and the other offline, keeping track of all elements from the beginning. In the first case, a structure based on the history of all updates can be built during the algorithm that allows us to efficiently locate the "position" of a new element (e.g., [38]), and in the second case the position of each uninserted element is kept up-to-date on every step (e.g., [19]). The bounds on work are typically the same in either case. Our incremental sort uses an online style algorithm, and the Delaunay triangulation uses an offline one.

Type 2 Algorithms. Here we describe a class of incremental algorithms, called Type 2 algorithms, that have a special structure. The iteration dependence graph for these algorithms is formed as follows: each step j independently has probability at most c/j of being a ***special step*** for some constant c; each special step j has dependence arcs to all steps $i < j$; and all non-special steps have one dependence arc to the closest earlier special step. For Type 2 algorithms, when a special step i is processed, it will check all previous steps, requiring $O(i)$ work and $d(i)$ depth, and when a non-special step is processed it does $O(1)$ work. It can be shown that in expectation each step takes $O(1)$ work, so this means that sequential implementations of Type 2 algorithms take $O(n)$ work in expectation.

THEOREM 2.3. *A Type 2 incremental algorithm has an iteration dependence depth of $O(\log n)$ w.h.p., and can be implemented to run in $O(n)$ expected work and $O(d(n) \log n)$ depth w.h.p., where $d(n)$ is the depth of processing a special step.*

PROOF. Since the probabilities are independent and the expectation is $\sum_{j=1}^{n} c/j = O(\log n)$, using a Chernoff bound, it is easy to show that the number of special steps is $O(\log n)$ w.h.p. With this bound, we can show that the iteration dependence depth, or the length of the longest path in the dependence graph, is $O(\log n)$ w.h.p. by noticing that there cannot be two consecutive non-special steps in a path (i.e., in the worst case every other vertex in the path is a special step, and there are only $O(\log n)$ of them w.h.p.).

We now show how parallel linear-work implementations can be obtained. A parallel implementation needs to execute the special steps one-by-one, and for each special step it can do its computation in parallel. For the non-special steps whose closest earlier special step has been executed, their computation can all be done in parallel. To maintain work-efficiency, we cannot afford to keep all unfinished steps active on each round. Instead, we start with a constant number of the earliest steps on the first round and on each round geometrically increase the number of steps processed, similar to the prefix methods described in earlier work on parallelizing iterative algorithms [7].

Without loss of generality, assume $n = 2^k$ for some integer k. We can process batched steps in $k + 1$ rounds—the first round runs the first task and the i-th round ($i > 1$) runs the second half of the first 2^{i-1} tasks (we refer to each batch as a ***prefix***). Each time a prefix is processed it checks all steps, finds the earliest unfinished special step, applies the computation associated with that step, and marks that special step and all earlier steps as finished. Since each step takes $O(1)$ work in expectation, each time a prefix is processed, it applies some computation with work $O(2^{i-2})$ and depth $d(n)$. The maximum/minimum of n elements can be computed in $O(n)$ work and $O(1)$ depth w.h.p. on an arbitrary CRCW PRAM [66], so finding the earliest special step can be done in $O(2^{i-2})$ work and $O(1)$ depth w.h.p. Marking steps as finished can be done in the same bounds. The number of times a prefix needs to be executed is equal to the number of special steps in the prefix, which for any prefix k is bounded by $\sum_{i=2^{k-2}}^{2^{k-1}-1} c/i = O(1)$ in expectation. Therefore, the work per prefix is $O(2^{i-2})$ in expectation, and summed over all rounds is $O(1) + \sum_{i=2}^{\log n} O(2^{i-2}) = O(n)$ in expectation. Summed across all prefixes, the total number of times they are processed is equal to the iteration dependence depth, which is $O(\log n)$ w.h.p., and so we have an overall depth of $O(d(n) \log n)$ w.h.p. □

Type 3 Algorithms. In the third type of incremental algorithms, instead of fully abiding by the dependence arcs and bounding the iteration depth, we allow for violations of the dependence arcs, and hence allow computations to differ from the sequential ordering possibly doing some extra work. However, we bound the extra work and show how to resolve the conflicts so the results are the same as in the sequential algorithm.

Consider a set of elements S. We assume that each element $x \in S$ defines a total ordering $<_x$ on all S. This ordering can be the same for each $x \in S$, or different. For example, in sorting the

total ordering would be the order of the keys and the same for all $x \in S$. We say the computation has ***separating dependences*** if the following condition is satisfied.

DEFINITION 2 (SEPARATING DEPENDENCES). *For any three elements $a, b, c \in S$, if $a <_c b <_c c$ or $c <_c b <_c a$, then c can only depend on a if a is inserted first among the three.*

In other words, if b separates a from c in the total ordering for c, and runs first, it will separate the dependence between a and c (also if c runs before a, of course, there is no dependence from a to c). Again using sorting as an example, if we insert b into a BST first (or use it as a pivot in quicksort), it will separate a from c and they will never be compared (each comparison corresponds to a dependence).

LEMMA 2.4. *In a randomized incremental algorithm that has separating dependences, $\hat{p}_{ij} = 2/i$ is an upper bound on p_{ij}.*

PROOF. Consider the total ordering $<_j$. The elements are inserted in random order, and so the probability that i is the nearest element before j among the first i is at most $1/i$. If it is not the nearest, it has already been separated from j by an earlier insertion. Similarly for the nearest element after j, giving a total probability of $2/i$. □

COROLLARY 2.5. *The number of dependences in a randomized incremental algorithm with separating dependences is $O(n \log n)$ in expectation.*[1]

This comes simply from the sum $\sum_{j=2}^{n} \sum_{i=1}^{j-1} p_{ij}$ which is bounded by $2n \ln n$. This leads to yet another proof that quicksort, or randomized insertion into a binary search tree, does $O(n \log n)$ comparisons in expectation. This is not the standard proof based on $p_{ij} = 2/(j - i + 1)$ being the probability that the i'th and j'th smallest elements are compared. Here the p_{ij} represent the probability that the i'th and j'th elements in the random order are compared.

In this paper we introduce graph algorithms that have separating dependences with respect to insertion of the vertices, and there is a dependence from vertex i to vertex j if a search from i (e.g., shortest path or reachability) visits j.

To allow for parallelism we permit iterations to run concurrently in rounds. This means that we might not separate elements that were separated in the sequential order (for example, if i separated j from k in the sequential order, but we run i and j concurrently, then they might both have a dependence to k). Also, for each algorithm we describe a way to combine results from steps that run in parallel so they give an identical result as the sequential order. If all elements are randomly permuted and the rounds are of geometrically increasing size starting with constant size, as with Type 2 algorithms, the approach wastes at most a constant factor in extra dependences (extra visited vertices in our graph algorithms).

THEOREM 2.6. *A randomized incremental algorithm with separating dependences can run in $O(\log n)$ parallel rounds over the iterations and every element will have $O(\log n)$ incoming dependence arcs in expectation (for a total of $O(n \log n)$).*

PROOF. As in the proof of Theorem 2.3, assume without loss of generality $n = 2^k - 1$ for some integer k, and we process batched steps in k rounds where the batch size increases by powers of 2 $(1, 2, 4, 8, \ldots)$. Clearly the number of rounds is $O(\log n)$. Consider the number of incoming arcs to the last element x to be inserted since it is the worst case in expectation. We can consider all elements at each round happening at the very beginning of the round, since in a parallel execution no other elements in the round will separate any elements in the round from x (although all previous rounds can). For round i, the beginning of the round is at position 2^{i-1}. Therefore by Lemma 2.4, the probability from any element in round i is $2/2^{i-1}$, and there are 2^{i-1} such elements giving 2 as the expected number of incoming arcs to x from elements in round i. When summed across the $\log n$ rounds we get $2 \log n$, which gives us the $O(\log n)$ bound on incoming arcs as claimed. □

As a side remark we note that by batching we have increased the number of incoming arcs over the sequential algorithm by a factor of about $(2 \log_2 n)/(2 \ln n) = \log_2 e \approx 1.44$.

[1] Also true w.h.p.

Algorithm 1: INCREMENTALSORT

Input: A sequence $K = \{k_1, \ldots, k_n\}$ of keys.
Output: A binary search tree over the keys in K.

```
1  // *P reads indirectly through the pointer P.
2  // The check on Line 8 is only needed for the parallel version.
3  Root ← a pointer to a new empty location
4  for i ← 1 to n do
5      N ← newNode(k_i)
6      P ← Root
7      while true do
8          if *P = null then
9              write N into the location pointed to by P
10             if *P = N then
11                 break
12         if N.key < *P.key then
13             P ← pointer to *P.left
14         else
15             P ← pointer to *P.right
16 return Root
```

3. COMPARISON SORTING

The first algorithm that we consider is sorting by incrementally inserting into a binary search tree (BST) with no rebalancing (w.l.o.g. we assume that no two keys are equal). It is well-known that for a random insertion order, this takes $O(n \log n)$ expected time. We apply our approach to show that the sequential incremental algorithm is also efficient in parallel. Algorithm 1 gives pseudocode that works either sequentially or in parallel. A step is one iteration of the **for** loop on Line 2. For the parallel version, the **for** loop should be interpreted as a **parallel for**, and the assignment on Line 7 should be considered a priority-write—i.e., all writes happen synchronously across the n iterations, and when there are writes to the same location, the smallest value gets written. The sequential version does not need the check on Line 8 since it is always true.

The dependence between iterations in the algorithm is in the check if $^*P$ is empty in Line 6. This means that iteration j depends on $i < j$ if and only if the node for i is on the path to j. This leads to the following lemma.

LEMMA 3.1. *For keys in random order,* INCREMENTALSORT *iteration j depends on iteration $i < j$ with probability at most $2/i$, and this upper bound is independent of all choices for $k > i$.*

PROOF. The proof follows the standard analysis (e.g. [57]). We consider the probability for steps i and $i + 1$ (and hence an upper bound for all $j > i$). Since it was inserted last, node i is a leaf in the BST when $i + 1$ is inserted. Node i will therefore only be on the path to $i + 1$ if they are neighbors in **sorted**$(1, \ldots, i + 1)$.

Node $i+1$ has at most two neighbors, each which is added on step i with probability $1/i$ (independent of all choices of $k > i$), giving $\hat{p}_i = 2/i$. □

Along with Corollary 2.2 this implies the following.

COROLLARY 3.2. *Insertion of n keys into a binary search tree in random order has iteration dependence depth $O(\log n)$ w.h.p.*

We note that since iterations only depend on the path to the key, the transitive reduction of the iteration dependence graph is simply the BST itself. In general, e.g. Delaunay triangulation in the next section, the dependence structure is not a tree.

THEOREM 3.3. *The parallel version of* INCREMENTALSORT *generates the same tree as the sequential version, and for a random order of n keys runs in $O(n \log n)$ work and $O(\log n)$ depth w.h.p. on a priority-write CRCW PRAM.*

PROOF. They generate the same tree since whenever there is a dependence, the earliest step wins. The number of rounds of the while loop is bounded by the iteration depth ($O(\log n)$ w.h.p.) since for each iteration, each round checks a new dependence (i.e. each round traverses one level of the iteration dependence graph). Since each round takes constant depth on the priority-write CRCW PRAM with n processors, this gives the required bounds. □

Note that this gives a much simpler work-optimal logarithmic-depth algorithm for comparison sorting than Cole's mergesort algorithm [22], although it is on a stronger model (priority-write CRCW instead of EREW) and is randomized.

4. PLANAR DELAUNAY TRIANGULATION

A Delaunay triangulation (DT) in the plane is a triangulation of a set of points P such that no point in P is inside the circumcircle of any triangle (the circle defined by the triangle's three corner points). We say a point ***encroaches*** on a triangle if it is in the triangle's circumcircle, and will assume for simplicity that the points are in general position (no three points on a line or four points on a circle). Delaunay triangulation can be solved sequentially in optimal $O(n \log n)$ work. There are also several work-efficient parallel algorithms that run in polylogarithmic depth [54, 3, 11], but they are all quite complicated.

The widely-used incremental Delaunay algorithms, due to their simplicity, date back to the 1970s [36]. They are based on the rip-and-tent idea: for each point p in order, rip out the triangles p encroaches on and tent over the resulting cavity with a ring of triangles centered at p. The algorithms differ in how the encroached triangles are found, and how they are ripped and tented. Clarkson and Shor [19] first showed that randomized incremental 3D convex hull is efficient, running in $O(n \log n)$ time in expectation, which by reduction implies the same results for DT. Guibas et al. (GKS) showed a simpler direct incremental algorithm for DT [38] with the same bounds, and this has become the standard version described in textbooks [48, 24, 29] and often used in practice. The GKS algorithm uses a history of triangle updates to locate the triangle t that p is in. It then searches out for all other encroached triangles. The algorithm, however, is inherently sequential since for certain inputs and certain points in the input, the search from t will likely have depth $\Theta(n)$, and hence a single step can take linear depth.

Our goal is to use an incremental DT algorithm for which the steps themselves can be parallelized. For this purpose we use an offline variant of an algorithm by Boissonnat and Teillaud [13]. We show that the iteration depth is $O(\log n)$ w.h.p. although this requires analyzing substeps. We further show that each step can be parallelized leading to a simple parallel algorithm with $O(n \log n)$ work in expectation and $O(\log^2 n)$ depth w.h.p.

Algorithm 2: INCREMENTALDT

Input: A sequence $V = \{v_1, \ldots, v_n\}$ of points in the plane.
Output: DT(V).
Maintains: A set of triangles M, and for each $t \in M$, the points that encroach on it, $E(t)$.

```
1  t_b ← a sufficiently large bounding triangle
2  E(t_b) ← V
3  M ← {t_b}
4  for i ← 1 to n do
5  |  foreach triangle t ∈ M with v_i ∈ E(t) do
6  |  |  REPLACETRIANGLE(M,t,v_i)
7  return M

8  function REPLACETRIANGLE(M,t,v)
9  |  foreach edge (u,w) ∈ t (three of them) do
10 |  |  if (u,w) is a boundary of v's encroached region then
11 |  |  |  t_o ← the other triangle sharing (u,w)
12 |  |  |  t' ← (u,w,v)
13 |  |  |  E(t') ← {v' ∈ E(t) ∪ E(t_o) | INCIRCLE(v',t')}
14 |  |  |  M ← M ∪ {t'}
15 |  M ← M \ {t}
```

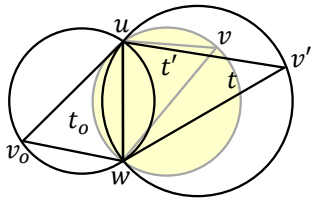

Figure 1: An illustration of the procedure of REPLACETRIANGLE. For each edge (u, w) that is a boundary of v's encroached triangle t, we find the triangle t_o on the other side of (u, w), generate the new triangle t', and recompute the encroaching set $E(t')$. Notice that the new (colored) circumcircle for t' (the encroaching region for t') can only contain points that are in the circumcircles of t and t_o.

Our variant is described in Algorithm 2. For each triangle $t \in M$ it maintains the set of uninserted points that encroach on t, denoted as $E(t)$. On each step i the algorithm selects the triangles that point i encroach on (all already known), removes these triangles and replaces them with new ones (see Figure 1). All work on uninserted points is done in determining $E(t')$ for each new triangle t', and for each new triangle only requires going through two existing sets, $E(t)$ and $E(t')$. This justified by Fact 4.1 [13]. Determining which triangles encroach on a point can be implemented by keeping a mapping of points to encroached triangles.

FACT 4.1. *When adding a triangle $t' = (u, w, v)$ for a new point v, and for the two old triangles t and t_o that shared the edge (u, w), we have $E(t) \cap E(t_o) \subseteq E(t') \subseteq E(t) \cup E(t_o)$.*

PROOF. Let $t = (u, w, v')$ be the triangle being removed and $t_o = (w, u, v_o)$ be the other triangle sharing the edge (u, w). The new point v must be in the circumcircle of t since it is removing it, but cannot be in the circumcircle of t_o since then it would be removing t_o as well and (u, w) would not be a boundary. The circumcircle of t' therefore must be contained in the union of the circumcircles of t and t_o, and must contain the intersection (see Figure 1). □

A time bound for INCREMENTALDT of $O(n \log n)$ follows from the analysis of Boissonnat and Teillaud [13], and more indirectly from Clarkson and Shor [19]. However for completeness and to show precise (within constant factor) bounds we include a bound on the number of INCIRCLE tests here. We note that due to Fact 4.1, the INCIRCLE test is not required for points that appear in both $E(t_o)$ and $E(t)$.

THEOREM 4.2. INCREMENTALDT *on n points in random order does at most $24n \ln n + O(n)$* INCIRCLE *tests in expectation.*

PROOF. On step i, for each point at $j > i$ we consider the boundary of the region j encroaches on. We define each of the boundary edges by its two endpoints (u, w) along with the (up to) two points sharing a triangle with (u, w). Note that in REPLACETRIANGLE a point is only tested for encroachment on the triangle (u, w, v) if its boundary (u, w, v_o, v') is being deleted and replaced with (u, w, v_o, v). We can therefore charge every comparison to the creation of a boundary of a point, and spend it when deleted.

Consider steps i and $i + 1$ (recall we can use $i + 1$ as a surrogate for any $j > i$). By Euler's formula, the average degree of a node in a planar graph is at most 6. Therefore, since $i + 1$ is selected uniformly at random (among $1, \ldots, i + 1$), its expected boundary size will be at most 6. Each boundary involves up to 4 points from $1, \ldots, i$, so the probability that the random point removed on step i is one of them is at most $4/i$. Therefore, the total expected number of boundaries of $i + 1$ (and hence any $j > i$) added on step i is at most $6 \times 4/i = 24/i$. If C is the number of in-circle tests, this gives:

$$\mathbb{E}[C] \leq 3n + \sum_{j=2}^{n} \sum_{i=1}^{j} 24/i \leq 24n \ln n + O(n)$$

where the $3n$ term comes from having to charge for the creation of the initial bounding triangle. □

Parallel Version and Analysis

We now consider the dependence depth of our algorithm. One approach to parallelizing the algorithm is to on each parallel round have every uninserted point check if its dependences are satisfied, and insert itself if so. It turns out that two points i and $i + 1$ are dependent if and only if immediately before either is added, their encroached regions overlap by at least an edge. Unfortunately this means that the probabilities of the dependence arcs (i, j) and (j, k) are not independent. In particular if j has a large encroached region, this increases both p_{ij} and p_{jk}. For example, consider a wagon wheel—$(n - 1)$ points nearly on a circle, and a single point at the hub. When the hub point is inserted at j, it will have dependence arcs from all previous points, and to all future points.

We therefore consider a more fine-grained dependence structure that relaxes the dependences. The observation is that not all triangles added by a point need to be added on the same round. In particular, REPLACETRIANGLE only depends on the triangle it is replacing and the three neighbors. We therefore can run REPLACETRIANGLE(M, t, v) as long as among the points encroaching on t and the three neighbors of t, there is no earlier point than v. An equal point is fine, since that would be the same point.

Algorithm 3 describes such a parallel variant. Since the triangles for a given point can be added on different rounds, the mesh is not necessarily self consistent after each round. We therefore assume that if a neighboring triangle (i.e., t_1, t_2 or t_3) is already deleted it can be ignored, and if not yet added, then REPLACETRIANGLE cannot proceed until added. A hash table mapping pairs of vertices representing edges to their up to two adjacent triangles can be used to find neighboring triangles. We assume that there is a synchronization point before Line 8.

Algorithm 3: PARINCREMENTALDT

Input: A sequence $V = \{v_1, \ldots, v_n\}$ of points in the plane.
Output: DT(V).
Maintains: $E(t)$, the points that encroach on each triangle t.

```
1 t_b ← a sufficiently large bounding triangle
2 E(t_b) ← V
3 M ← {t_b}
4 while E(t) ≠ ∅ for any t ∈ M do
5 |   parallel foreach triangle t ∈ M do
6 |   |   Let t_1, t_2, t_3 be the three neighboring triangles
7 |   |   if min(E(t)) ≤ min(E(t_1) ∪ E(t_2) ∪ E(t_3)) then
8 |   |   |   REPLACETRIANGLE(M, t, min(E(t)))
9 return M
```

Note that there is a one-to-one correspondence between the calls to REPLACETRIANGLE in the sequential and parallel algorithm—i.e., they are the "same" algorithm but just with a different ordering. We believe that this parallel version is even simpler than the sequential version since it does not require a mapping from points to encroached triangles.

For a sequence of points V, let $G_T(V) = (T, E)$ be the dependence graph defined by PARINCREMENTALDT(V) in the following way. The vertices T corresponds to triangles created by the algorithm, and for each call to REPLACETRIANGLE(M, t, v_i) we place an arc from triangle t and its three neighbors (t_1, t_2 and t_3) to each of the one, two, or three triangles created by REPLACETRIANGLE. Note that we can associate each triangle with the point v_i that created it. This is an iteration dependence graph over all iterations, including subiterations that create triangles.

THEOREM 4.3. *For points V in random order, $D(G_T(V)) = O(\log n)$ w.h.p.*

PROOF. For a sequence of points V let $T(V, i)$ be the set of triangles created by point v_i, and let $E_{t,t'}(V)$ be the indicator variable for a dependence arc from triangle t to t' given V. Let $\hat{p}_{ij}$ be an upper bound for the total probability that triangles created by v_i have an arc to any single triangle created by v_j (uniformly random over all permutations of the input). More precisely:

$$\hat{p}_{ij} \geq \frac{1}{|V|!} \left(\sum_{V' \in \text{perms}(V)} \left(\max_{t \in T(V', j)} \left(\sum_{t' \in T(V', i)} E_{t,t'}(V') \right) \right) \right).$$

Consider a path going through a triangle created by each point $K \subseteq \{1, \ldots, n\}$. If the $\hat{p}_{ij}$ are independent along any path, then the probability of such a path is bounded by the product of the $\hat{p}_{ij}$ along the path and the expectation on the number of triangles on the last point (which is 6 and independent of the $\hat{p}_{ij}$). For $\hat{p}_{ij} = f(i) \geq 1/n$ this gives a total bounded by $6n \prod_{k \in K} f(k)$. We can now apply the proof of Lemma 2.1, where the $\hat{p}_{ij}$ are interpreted in this new way, and the probability along a path $K \subseteq \{1, \ldots, n\}$ is interpreted as the probability that any path exists involving triangles created by those points. As in the proof of Lemma 2.1, the union bound over all possible subsets K of length l gives an overall upper bound on the probability of any path of length $l - 1$:

$$\Pr(D(G_T(V)) > l - 1) \leq \sum_{K \subseteq \{1, \ldots, n\}, |K| = l} 6n \prod_{k \in K} f(k)$$

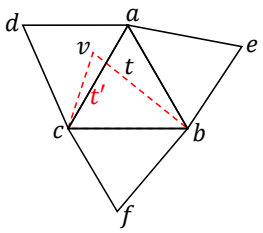

Figure 2: The dependence of t' on four previously created triangles.

and assuming that $\hat{p}_{ij} = f(i) = O(1/i)$ gives $O(\log n)$ depth w.h.p. (the argument about equal probabilities being the worst case still holds).

We are therefore left with showing that the $\hat{p}_{ij} = O(1/i)$ is a valid upper bound, and that this bound is independent along any path (allowing us to multiply them). As usual we consider steps (points) i and $j = i+1$, and hence an upper bound for any $j > i$. We consider one triangle t' created at $i+1$. Every triangle t' depends on 4 triangles—the t that was sacrificed for it in REPLACETRIANGLE, and its three neighbors (see Figure 2). These 4 triangles have six corners in total, any one of which could be the point i. Three of those points (a, b and c in the figure) would create three triangles that t' depends on. The other three (d, e and f in the figure) only create one triangle t' depends on. Therefore given that the point i is selected uniformly at random from i points the total probability that triangles at i have an arc to t' (a triangle at $i+1$) is bounded by $\hat{p}_{ij} = (3 \cdot 3 + 3 \cdot 1)/i = 12/i$.

The probabilities are independent since $\hat{p}_{ij} = 12/i$ does not depend on the point j, or indeed any of the points selected in positions $(i+1), \ldots, n$. For example, conditioned on the center of the wagon wheel being at j, $\hat{p}_{ij} = 12/i$ is still an upper bound. With $\hat{p}_{ij} = O(1/i)$, and independence of the $\hat{p}_{ij}$ along paths, we can apply our variant of Lemma 2.1 (described above), and Corollary 2.2 for the result. □

THEOREM 4.4. PARINCREMENTALDT *(Algorithm 3) runs in* $O(n \log n)$ *work in expectation and* $O(\log^2 n)$ *depth w.h.p. on the CRCW PRAM.*

PROOF. The number of rounds of PARINCREMENTALDT is $D(G_T(V))$ since the iteration dependence graph is defined by the algorithm. Each round has depth $O(\log n)$ for merging the encroached sets and load balancing, for an overall depth of $O(\log^2 n)$ (w.h.p.). Assuming each triangle maintains its minimum index, checking if a triangle is safe to process takes constant work. Since there are be at most $O(n)$ triangles on any round (true even though the mesh is not necessarily consistent), each round does at most $O(n)$ work to check all the triangles, for a total of $O(n \log n)$ work across rounds (w.h.p.). The rest of the work is no more than the sequential version, which is $O(n \log n)$ in expectation. □

5. LINEAR-WORK ALGORITHMS

In this section, we study several problems from low-dimensional computational geometry that have linear-work randomized incremental algorithms. These algorithms fall into the Type 2 category of algorithms defined in Section 2, and their iteration depth is polylogarithmic w.h.p. To obtain linear-work parallel algorithms, we process the steps in prefixes, as described in Section 2. For simplicity, we describe the algorithms for these problems in two dimensions, and and briefly note how they can be extended to any fixed number of dimensions.

5.1 Linear Programming

Constant-dimensional linear programming (LP) has received significant attention in the computational geometry literature, and several parallel algorithms for the problem have been developed [16, 25, 35, 28, 1, 34, 58, 2]. We consider linear programming in two dimensions. We assume that the constraints are given in general position and the solution is either infeasible or bounded. We note that these assumptions can be removed without affecting the asymptotic cost of the algorithm [56]. The standard randomized incremental algorithm [56] adds the constraints one-by-one in a random order, while maintaining the optimum point at any time. If a newly added constraint causes the optimum to no longer be feasible (a tight constraint), we find a new feasible optimum point on the line corresponding to the newly added constraint by solving a one-dimension linear program, i.e., taking the minimum or maximum of the set of intersection points of other earlier constraints with the line. If no feasible point is found, then the algorithm reports the problem as infeasible.

The iteration dependence graph is defined with the constraints as steps, and fits in the framework of Type 2 algorithms from Section 2. The steps corresponding to inserting a tight constraint are the special steps. Special steps depend on all earlier steps because when a tight constraint executes, it needs to look at all earlier constraints. Non-special steps depend on the closest earlier special step i because it must wait for step i to execute before executing itself to retain the sequential execution (we can ignore all of the earlier constraints since i will depend on them). Using backwards analysis, a step j has a probability of at most $2/j$ of being a special step because the optimum is defined by at most two constraints and the constraints are in a randomized order. Furthermore, the probabilities are independent among different steps. This gives us a dependence depth of $O(\log n)$ w.h.p. as discussed in Section 2.

As described in the proof of Theorem 2.3, our parallel algorithm executes the steps in prefixes. Each time a prefix is processed, it checks all of the constraints and finds the earliest one that causes the current optimum to be infeasible using line-side tests. The check per step takes $O(1)$ work and processing a violating constraint at step i takes $O(i)$ work and $O(1)$ depth w.h.p. to solve the one-dimensional linear program which involves minimum/maximum operations. Applying Theorem 2.3 with $d(n) = O(1)$ gives the following theorem.

THEOREM 5.1. *The randomized incremental algorithm for 2D linear programming can be parallelized to run in* $O(n)$ *work in expectation and* $O(\log n)$ *depth w.h.p. on an arbitrary-CRCW PRAM.*

We note that the algorithm can be extended to the case where the dimension d is greater than two by having a randomized incremental d-dimensional LP algorithm recursively call a randomized incremental algorithm for solving $(d-1)$-dimensional LPs. This increases the iteration dependence depth (and hence the depth of the algorithm) to $O(d! \log^{d-1} n)$ w.h.p. and increases the expected work to $O(d!n)$. We note that we can generate the random permutation only once and reuse it for the sub-problems. Although we lose independence, the expectation is not affected, and since there are only a constant (a function of d) number of sub-problems with high probability, the high probability bound for the depth is not affected.

5.2 Closest Pair

The ***closest pair*** problem takes as input a set of points in the plane and returns the pair of points with the smallest distance between each other. We assume that no pair of points have the same distance. A well-known expected linear-work algorithm [51, 44, 32, 39] works by maintaining a grid and inserting the points into the grid in a

random order. The grid partitions the plane into regions of size $r \times r$ where each non-empty region stores the points inside the region and r is the distance of the closest pair so far (initialized to the distance between the first two points). It is maintained using a hash table. Whenever a new point is inserted, one can check the region the point belongs in and the eight adjacency regions to see whether the new value of r has decreased, and if so, the grid is rebuilt with the new value of r. The check takes $O(1)$ work as each region can contain at most nine points, otherwise the grid would have been rebuilt earlier. Therefore insertion takes $O(1)$ work, and rebuilding the grid takes $O(i)$ work where i is the number of points inserted so far. Using backwards analysis, one can show that point i has probability at most $2/i$ of causing the value of r to decrease, so the expected work is $\sum_{i=1}^{n} O(i) \cdot (2/i) = O(n)$.

This is a Type 2 algorithm, and the iteration dependence graph is similar to that of linear programming. The special steps are the ones that cause the grid to be rebuilt, and the dependence depth is $O(\log n)$ w.h.p. Rebuilding the grid involves hashing, and can be done in parallel in $O(i)$ work and $O(\log^* i)$ depth w.h.p. for a set of i points [31]. We also assume that the points in each region are stored in a hash table, to enable efficient parallel insertion and lookup in linear work and $O(\log^* i)$ depth. To obtain a linear-work parallel algorithm, we again execute the algorithm in prefixes. Applying Theorem 2.3 with $d(n) = O(\log^* n)$ gives the following theorem.

THEOREM 5.2. *The randomized incremental algorithm for closest pair can be parallelized to run in $O(n)$ work in expectation and $O(\log n \log^* n)$ depth w.h.p. on an arbitrary-CRCW PRAM.*

We note that the algorithm can be extended to d dimensions where the depth is $O(\log d \log n \log^* n)$ w.h.p. and expected work is $O(c_d n)$ where c_d is some constant that depends on d.

5.3 Smallest Enclosing Disk

The ***smallest enclosing disk*** problem takes as input a set of points in two dimensions and returns the smallest disk that contains all of the points. We assume that no four points lie on a circle. Linear-work algorithms for this problem have been described [47, 67], and in this section we will study Welzl's randomized incremental algorithm [67]. The algorithm inserts the points one-by-one in a random order, and maintains the smallest enclosing disk so far (initialized to the smallest disk defined by the first two points). Let v_i be the point inserted on the i'th iteration. If an inserted point v_i lies outside the current disk, then a new smallest enclosing disk is computed. We know that v_i must be on the smallest enclosing disk. We first define the smallest disk containing v_1 and v_i, and scan through v_2 to v_{i-1}, checking if any are outside the disk (call this procedure **Update1**). Whenever v_j ($j < i$) is outside the disk, we update the disk by defining the disk containing v_i and v_j and scanning through v_1 to v_{j-1} to find the third point on the boundary of the disk (call this procedure **Update2**). **Update2** takes $O(j)$ work, and **Update1** takes $O(i)$ work plus the work for calling **Update2**. With the points given in a random order, the probability that the j'th iteration of **Update1** calls **Update2** is at most $2/j$ by a backwards analysis argument, so the expected work of **Update1** is $O(i) + \sum_{j=1}^{i}(2/j) \cdot O(j) = O(i)$. The probability that **Update1** is called when the i'th point is inserted is at most $3/i$ using a backwards analysis argument, so the expected work of this algorithm is $\sum_{i=1}^{n}(3/i) \cdot O(i) = O(n)$.

This is another Type 2 algorithm whose iteration dependence graph is similar to that of linear programming and closest pair. The points are the steps, and the special steps are the ones that cause **Update1** to be called, which for step i has at most $3/i$ probability of happening. The dependence depth is again $O(\log n)$ w.h.p. as discussed in Section 2.

Our work-efficient parallel algorithm again uses prefixes, both when inserting the points, and on every call to **Update1**. We repeatedly find the earliest point that is outside the current disk by checking all points in the prefix with an in-circle test and taking the minimum among the ones that are outside. **Update1** is work-efficient and makes $O(\log n)$ calls to **Update2** w.h.p., where each call takes $O(1)$ depth w.h.p. as it does in-circle tests and takes a maximum. As in the sequential algorithm, each step takes $O(1)$ work in expectation. Applying Theorem 2.3 with $d(n) = O(\log n)$ w.h.p. (the depth of a executing a step and calling **Update1**) gives the following theorem.

THEOREM 5.3. *The randomized incremental algorithm for smallest enclosing disk can be parallelized to run in $O(n)$ work in expectation and $O(\log^2 n)$ depth w.h.p. on an arbitrary-CRCW PRAM.*

The algorithm can be extended to d dimension, with $O(d! \log^d n)$ depth w.h.p., and $O(c_d n)$ expected work for some constant c_d that depends on d. Again, we can use the same randomized order for all sub-problems.

6. ITERATIVE GRAPH ALGORITHMS

In this section we study two sequential graph algorithms that can be viewed as offline versions of randomized incremental algorithms. We show that the algorithms are Type 3 algorithms, as described in Section 2, and also that steps executing in parallel can be combined efficiently. This gives us simple parallel algorithms for the problems. The algorithms use single-source shortest paths and reachability as (black-box) subroutines, which is the dominating cost. Our algorithms are within a logarithmic factor in work and depth of a single call to these subroutines on the input graph.

6.1 Least-Element Lists

The concept of Least-Element lists (LE-lists) for a graph (either unweighted or with non-negative weights) was first proposed by Cohen [20] for estimating the neighborhood sizes of vertices. The idea has subsequently been used in many applications related to estimating the influence of vertices in a network (e.g., [21, 27] among many others), and generating probabilistic tree embeddings of a graph [43, 8], which itself is a useful component in a number of network optimization problems and in constructing distance oracles [8]. For $d(u, v)$ being the shortest path from u to v in G, we have:

DEFINITION 3 (LE-LIST). *Given a graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$, the* **LE-lists** *are*

$$L(v_i) = \left\{ v_j \in V \mid d(v_i, v_j) < \min_{k=1}^{j-1} d(v_i, v_k) \right\}$$

sorted by $d(v_i, v_j)$.

In other words, a vertex u is in vertex v's LE-list if and only if there are no earlier vertices (than u) that are closer to v. Often one stores with each vertex v_j in $L(v_i)$ the distance of $d(v_i, v_j)$.

Algorithm 4 provides a sequential iterative (incremental) construction of the LE-lists, where the i'th step is the i'th iteration of the for-loop. The set S captures all vertices that are closer to the i'th vertex than earlier vertices (the previous closest distance is stored in $\delta(\cdot)$). Line 3 involves computing S with a single-source shortest paths (SSSP) algorithm (e.g., Dijkstra's algorithm for weighted graphs and BFS for unweighted graphs, or other algorithms [62, 65, 45, 9] with more work but less depth). Note that the only minor

Algorithm 4: The iterative LE-lists construction [20]

Input: A graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$
Output: The LE-lists $L(\cdot)$ of G

```
Set δ(v) ← +∞ and L(v) ← ∅ for all v ∈ V.
for i ← 1 to n do
    Let S = {u ∈ V | d(v_i, u) < δ(u)}
    for u ∈ S do
        δ(u) ← d(v_i, u)
        concatenate ⟨v_i, d(v_i, u)⟩ to the end of L(u)
return L(·)
```

change to these algorithms is to remove the initialization of the tentative distances before we run SSSP, and instead use the $\delta(\cdot)$ values from previous steps in Algorithm 4. Thus the search will only explore S and its outgoing edges. Cohen [20] showed that if the vertices are in random order, then each LE-list has size $O(\log n)$ w.h.p., and that using Dijkstra, with distances initialized with $\delta(\cdot)$, the algorithm runs in $O(\log n(m + n \log n))$ time. Here we show that the algorithm is in fact parallel, and in particular a Type 3 incremental algorithm.

In this paper, we treat the shortest paths algorithm as a black box that computes the set S in depth $D_{SP}(n', m')$ and work $W_{SP}(n', m')$, where $n' = |S|$ and m' is the sum of the degrees of S. We also assume the cost functions are convex, i.e. $W_{SP}(n_1, m_1) + W_{SP}(n_2, m_2) \leq W_{SP}(n, m)$ for $n_1 + n_2 \leq n$ and $m_1 + m_2 \leq m$, which holds for all existing shortest paths algorithms.

For the separating dependences: v_j depends on v_i if and only if $v_i \in L(v_j)$, (i.e., was searched by v_i), and we use the total orderings $i <_k j$ if $d(v_k, v_i) < d(v_k, v_j)$. This gives:

LEMMA 6.1. *Algorithm 4 has a separating dependence for the dependences and orderings $<_v$ defined above.*

PROOF. By Definition 2, we need to show that for any three vertices $v_a, v_b, v_c \in V$, if $v_a <_c v_b <_c v_c$ or $v_c <_c v_b <_c v_a$, then v_c can only be visited on v_a's step if v_a's step is the first among the three.

Clearly the statement holds if v_c's step is the earliest among the three. We now consider the case when v_b's step is the first among the three. Since $d(v_c, v_c) = 0$, $v_a <_c v_b <_c v_c$ cannot happen, so we only need to consider the case $v_c <_c v_b <_c v_a$. Since $d(v_c, v_b) < d(v_c, v_a)$ and $b < a$, based on the definition of the LE-lists, $v_a \notin L(v_c)$. As a result, v_c can only be visited in v_a's step if v_a's step is first among the three. □

To parallelize the algorithm, we execute prefixes of vertices in batches that are geometrically increasing in size. Multiple searches will be executing in parallel in a round, so some dependences will be violated. We discuss how to fix these violations in a post-processing phase, which effectively combines the steps so we can apply Theorem 2.6 and Lemma 6.1 to obtain the following theorem.

THEOREM 6.2. *The LE-lists of a graph with the vertices in random order can be constructed in $O(W_{SP}(n, m) \log n)$ expected work and $O(D_{SP}(n, m) \log n)$ depth w.h.p. on the CRCW PRAM.*

PROOF. First we bound the cost of the algorithm excluding the post-processing step. Note that here visiting each vertex is no longer the same cost but based on the number of outgoing edges. However, since each vertex is visited no more than $O(\log n)$ times, the cost can be averaged. Assuming random input order, we can apply Theorem 2.6 and Lemma 6.1 with the convexity of the work complexity, and this is within the claimed bounds.

Since we execute multiple searches in parallel, there may be entries in the LE-lists that would not have been generated in the sequential algorithm. These extra entries violate the property that entries have to be strictly increasing in distance. We fix this with a post-processing phase, which effectively combines the steps executing in parallel. If the elements in the list are sorted by order of the vertices, we can sequentially filter out the extra entries in the list by finding the ones that are out of order by their distance (distances should be in reverse order). The size of the LE-lists can be shown to be $O(\log n)$ w.h.p. [20, 57], so the overall cost of the scan is $O(n \log n)$ work and $O(\log n)$ depth w.h.p.

We now show that we can efficiently sort the list. Without loss of generality, assume $n = 2^k$ for some integer k. Since the order of vertices in lists can only be violated inside a round, we only need to sort the entries within rounds. Consider the last vertex v_n (which is the worst case) and let X_i be the indicator variable for the event that v_i is in the LE-list of v_n. Thus $\mathbb{E}[X_i] = 1/i$ when $i < n$ and $\mathbb{E}[X_i] = 1$ when $i = n$. Note that two random variables X_i and X_j are independent. We analyze the cost of sorting the LE-list of v_n in a round, which executes a prefix of the steps. Assume that we use a loose upper bound of quadratic work for sorting and let $s = 2^{k-2}$ and $t = 2^{k-1}$. The expected work in round 1 is clearly $O(1)$ and in round $k > 1$ is:

$$\mathbb{E}\left[(\sum_{j=s+1}^{t} X_j)^2\right] = \mathbb{E}\left[\sum_{j=s+1}^{t} X_j^2\right] + 2 \cdot \mathbb{E}\left[\sum_{j=s+1}^{t} \sum_{j'=j+1}^{t} X_j X_{j'}\right]$$
$$< s \cdot \frac{1}{s} + 2 \cdot \frac{1}{s^2} \cdot s(s-1) = O(1)$$

Since there are $O(\log n)$ rounds, the expected work to sort this LE-list is $O(\log n) \cdot O(1) = O(\log n)$. The only exception is X_n, which is 1 with probability 1 instead of $1/n$. In the worst case, this adds at most $O(\log n)$ to the work to compare it to all elements in the list. Thus the expected work to sort one LE-list in the worst case is $O(\log n)$, and is $O(n \log n)$ when summed over all lists. The depth for sorting is $O(\log n)$ [22], which is within the claimed bounds. □

6.2 Strongly Connected Components

Given a directed unweighted graph $G = (V, E)$, a ***strongly connected component*** (SCC) is a maximal set of vertices $C \subseteq V$ such that for every pair of vertices u and v in C, there are directed paths both from u to v and from v to u. Tarjan's algorithm [63] finds all strongly connected components of a graph using a single pass of depth-first search (DFS) in $O(|V| + |E|)$ work. However, DFS is generally considered to be hard to parallelize [53], so a divide-and-conquer SCC algorithm [23] is usually used in parallel settings [41, 4, 61, 64].

The basic idea of the divide-and-conquer algorithm is similar to quicksort. It applies forward and backward reachability queries for a specific "pivot" vertex v, which partitions the remaining vertices into four subsets of the graph, based on whether it is forward reachable from v, backward reachable, both, or neither. The subset of vertices reachable from both directions form a strongly connected component, and the algorithm is applied recursively to the three remaining subsets. Coppersmith et al. [23] show that if the vertex v is selected uniformly at random, then the algorithm sequentially runs in $O(m \log n)$ work in expectation.

Although divide-and-conquer is generally good for parallelism, a problem with this algorithm is that the divide-and-conquer tree can be very unbalanced. The issue is that if the input graph is very sparse such that most of the reachability searches only visit a few vertices, then most of the vertices will fall into the subset of un-

Algorithm 5: The sequential iterative SCC algorithm

Input: A directed graph $G = (V, E)$ with $V = \{v_1, \ldots, v_n\}$.
Output: The set of strongly connected components of G.

1 $\mathcal{V} \leftarrow \{\{v_1, v_2, \ldots, v_n\}\}$ (Initial Partition)
2 $S_{scc} \leftarrow \{\}$
3 **for** $i \leftarrow 1$ *to* n **do**
4 Let $S \in \mathcal{V}$ be the subgraph that contains v_i
5 **if** $S = \varnothing$ **then** go to the next iteration;
6 $R^+ \leftarrow$ FORWARD-REACHABILITY(S, v_i)
7 $R^- \leftarrow$ BACKWARD-REACHABILITY(S, v_i)
8 $V_{scc} \leftarrow R^+ \cap R^-$
9 $\mathcal{V} \leftarrow \mathcal{V} \backslash \{S\} \cup \{R^+ \backslash V_{scc}, R^- \backslash V_{scc}, S \backslash (R^+ \cup R^-)\}$
10 $S_{scc} \leftarrow S_{scc} \cup \{V_{scc}\}$
11 **return** S_{scc}

reachable vertices from v, creating unbalanced partitions with $\Theta(n)$ recursion depth. Schudy describes a technique to better balance the partitions [55], however the approach requires a factor of $O(\log n)$ extra work compared to the original algorithm and the depth is equal to the depth of $O(\log^2 n)$ reachability queries. Tomkins et al. describe another parallel approach [64], although the analysis is quite complicated.[2]

The divide-and-conquer algorithm [23] can also be viewed as an incremental algorithm as described in Algorithm 5. A random ordering is equivalent to picking the pivots at random. This is the algorithm we will analyze as a Type 3 algorithm. Our analysis is significantly simpler than those of [55, 64], and furthermore the work of our algorithm matches that of the sequential algorithm.

As in previous work on parallel SCC algorithms, we treat the algorithm for performing reachability queries as a black box with $W_R(n, m)$ work and $D_R(n, m)$ depth, where n are the number of reachable vertices and m is the sum of their degrees. It can be implemented using a variety of algorithms with strong theoretical bounds [62, 65] or simply with a breadth-first search for low-diameter graphs. We also assume convexity on the work $W_R(n, m)$, which holds for existing reachability algorithms.

We first show that the algorithm has separating dependences. Here a dependence from i to j corresponds to a forward or backward reachability search from i visiting j (Lines 6 and 7 in Algorithm 5). Let $T = (t_1, t_2, \ldots, t_n)$ be an arbitrary topological order of components in the given graph G, with vertices of the same component arbitrarily ordered within the component. T is not constructed explicitly, but only used in analysis. To define the total order for vertex v_i, i.e., $<_{v_i}$, we take all vertices of T that are forward or backward reachable from v_i (including v_i itself) and put them at the beginning of the ordering (maintaining their relative order), and put the unreachable vertices after them. Given this ordering we have the following lemma.

LEMMA 6.3. *Algorithm 5 has a separating dependence for the dependences and orderings $<_v$ defined above.*

PROOF. By Definition 2, we need to show that for any three vertices $v_a, v_b, v_c \in V$, if $v_a <_c v_b <_c v_c$ or $v_c <_c v_b <_c v_a$, v_c can only be reached (forward or backward) in v_a's step if v_a's step is the first among the three.

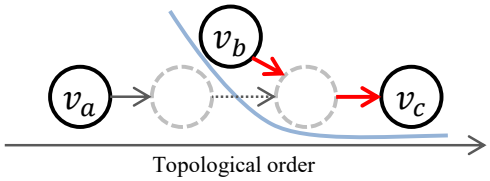

Figure 3: An illustration for the proof of Lemma 6.3.

Clearly the statement is true if v_c is earliest. We now consider the case when v_b's step is first among the three vertices. We give the argument for the forward direction, and the backward direction is true by symmetry. If v_b is in the same SCC as either v_a or v_c, then in v_b's step either v_a or v_c is marked in one SCC and removed from the subgraph set $\mathcal{V}$. Otherwise, since v_b and v_a are not in the same SCC, when $v_c <_c v_b <_c v_a$, v_c cannot be forward reachable in v_a's step, and when $v_a <_c v_b <_c v_c$, v_a cannot be forward reachable from v_b's step. In the second case, after v_b's step, the forward reachability search from v_b reaches v_c but not v_a, so v_a and v_c fall into different components in $\mathcal{V}$ (shown in Figure 3). As a result, v_c is also not reachable in v_a's step.

In conclusion, v_c can only be reached (forward or backward) in v_a's step if v_a's step is first among the three. □

Note that this separating dependence implies that the sequential algorithm on a random ordering does $O(m \log n)$ work since each vertex v_j is visited $\sum_{i=1}^{j} 2/i = O(\log n)$ times in expectation using Lemma 2.4, and the expectation is independent of the degrees. Since $W_R(n, m) = O(m)$ sequentially, e.g., using BFS, this gives $O(m \log n)$ work.

We now consider a parallel version based on rounds. To implement the parallel version, we need a way to efficiently combine the steps that run in the same round. First, there might be multiple vertices (sources) that obtain the same SCC in the same round and we want to avoid having duplicates in the final output. Second, since multiple reachability queries are performed in parallel, we want them to partition the subgraph set $\mathcal{V}$ consistently. The solution for combining steps is have all steps do a priority-write into a data field associated with the vertices that they visit during the reachability queries, and synchronize after the reachability queries.

To partition the graph, all edges with endpoints that hold different data values or differ in whether they are forward or backward reachable are removed (marked) from the graph, and this disconnects the graph into appropriate subgraphs. Also, all nodes that are reachable from the sources in both directions, which can be checked by their data fields, form SCCs and are removed (marked) from graph. All of these extra steps take linear work and constant depth. Finally, after all $O(\log n)$ rounds are complete, we group the vertices to form SCCs based on the vertex labels in data fields, which requires linear work and $O(\log n)$ depth [52, 37]. Note that this implementation yields exactly the same intermediate state as the sequential algorithm after each round, making it deterministic.

THEOREM 6.4. *For a random order of the input vertices, the incremental SCC algorithm does $O(W_R(n, m) \log n)$ expected work and has $O(D_R(n, m) \log n)$ depth on a priority-write CRCW PRAM.*

PROOF. The overall extra work is $O(m \log n)$ and no more than the work for executing the reachability queries. The depth for the additional operations is constant in each round and $O(\log n)$ at the end of the algorithm. Therefore if the input vertices are randomly

[2] Tomkins et al. [64] claim that their algorithm takes the same amount of work as the sequential algorithm, however it seems that there are errors in their analysis. For example, the goal of the analysis is to show that in each round their algorithm visits $O(n)$ vertices in expectation, which they claimed to imply visiting $O(m)$ edges in expectation. This is not generally true since the vertices do not necessarily have the same probabilities of being visited. Other than this, their work contains many interesting ideas that motivated us to look at this problem.

permuted, we can apply Theorem 2.6 with Lemma 6.3 and the convexity of the work cost to bound the expected work and depth of the algorithm. □

7. CONCLUSION

In this paper, we have analyzed the dependence structure in a collection of known randomized incremental algorithms (or slight variants) and shown that there is inherently high parallelism in all of the algorithms. The approach leads to particularly simple parallel algorithms for the problems—only marginally more complicated (if at all) than some of the very simplest efficient sequential algorithms that are known for the problems. Furthermore the approach allows us to borrow much of the analysis already done for the sequential versions (e.g., with regard to total work and correctness). We presented three general classes of algorithms, and tools and general theorems that are useful for multiple algorithms within each class. We expect that there are many other algorithms that can be analyzed with these tools and theorems.

Acknowledgments

This research was supported in part by NSF grants CCF-1314590 and CCF-1533858, the Intel Science and Technology Center for Cloud Computing, and the Miller Institute for Basic Research in Science at UC Berkeley.

8. REFERENCES

[1] M. Ajtai and N. Megiddo. A deterministic poly($\log \log n$)-time n-processor algorithm for linear programming in fixed dimension. In *ACM Symposium on Theory of Computing (STOC)*, pages 327–338, 1992.

[2] N. Alon and N. Megiddo. Parallel linear programming in fixed dimension almost surely in constant time. *J. ACM (JACM)*, 41(2):422–434, Mar. 1994.

[3] M. Atallah and M. Goodrich. Deterministic parallel computational geometry. In *Synthesis of Parallel Algorithms*, pages 497–536. Morgan Kaufmann, 1993.

[4] J. Barnat, P. Bauch, L. Brim, and M. Ceska. Computing strongly connected components in parallel on CUDA. In *International Parallel & Distributed Processing Symposium (IPDPS)*, pages 544–555, 2011.

[5] D. K. Blandford, G. E. Blelloch, and C. Kadow. Engineering a compact parallel Delaunay algorithm in 3D. In *ACM Symposium on Computational Geometry (SoCG)*, pages 292–300, 2006.

[6] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, and J. Shun. Internally deterministic algorithms can be fast. In *Principles and Practice of Parallel Programming (PPoPP)*, pages 181–192, 2012.

[7] G. E. Blelloch, J. T. Fineman, and J. Shun. Greedy sequential maximal independent set and matching are parallel on average. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 308–317, 2012.

[8] G. E. Blelloch, Y. Gu, and Y. Sun. Efficient construction of probabilistic tree embeddings. *arXiv preprint: 1605.04651*, 2016.

[9] G. E. Blelloch, Y. Gu, Y. Sun, and K. Tangwongsan. Parallel shortest-paths using radius stepping. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[10] G. E. Blelloch, A. Gupta, and K. Tangwongsan. Parallel probabilistic tree embeddings, k-median, and buy-at-bulk network design. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 205–213, 2012.

[11] G. E. Blelloch, J. C. Hardwick, G. L. Miller, and D. Talmor. Design and implementation of a practical parallel Delaunay algorithm. *Algorithmica*, 24(3-4):243–269, 1999.

[12] R. L. Bocchino, V. S. Adve, S. V. Adve, and M. Snir. Parallel programming must be deterministic by default. In *Usenix HotPar*, 2009.

[13] J.-D. Boissonnat and M. Teillaud. On the randomized construction of the delaunay tree. *Theoretical Computer Science*, 112(2):339–354, 1993.

[14] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM (JACM)*, 21(2):201–206, 1974.

[15] C. Cao Minh, J. Chung, C. Kozyrakis, and K. Olukotun. STAMP: Stanford transactional applications for multi-processing. In *IEEE International Symposium on Workload Characterization (IISWC)*, 2008.

[16] D. Z. Chen and J. Xu. Two-variable linear programming in parallel. *Computational Geometry*, 21(3):155 – 165, 2002.

[17] P. Cignoni, C. Montani, R. Perego, and R. Scopigno. Parallel 3D delaunay triangulation. *Computer Graphics Forum*, 12(3):129–142, 1993.

[18] M. Cintra, D. R. Llanos, and B. Palop. International conference on computational science and its applications. In *Speculative Parallelization of a Randomized Incremental Convex Hull Algorithm*, pages 188–197, 2004.

[19] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Discrete & Computational Geometry*, 4(5):387–421, 1989.

[20] E. Cohen. Size-estimation framework with applications to transitive closure and reachability. *Journal of Computer and System Sciences*, 55(3):441–453, 1997.

[21] E. Cohen and H. Kaplan. Efficient estimation algorithms for neighborhood variance and other moments. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 157–166, 2004.

[22] R. Cole. Parallel merge sort. *SIAM J. Comput.*, 17(4):770–785, 1988.

[23] D. Coppersmith, L. Fleischer, B. Hendrickson, and A. Pinar. A divide-and-conquer algorithm for identifying strongly connected components. Technical Report RC23744, IBM, 2003.

[24] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag, 2008.

[25] X. Deng. An optimal parallel algorithm for linear programming in the plane. *Information Processing Letters*, 35(4):213 – 217, 1990.

[26] P. Diaz, D. R. Llanos, and B. Palop. Parallelizing 2D-convex hulls on clusters: Sorting matters. *Jornadas De Paralelismo*, 2004.

[27] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha. Scalable influence estimation in continuous-time diffusion networks. In *Advances in Neural Information Processing Systems (NIPS)*, pages 3147–3155, 2013.

[28] M. Dyer. A parallel algorithm for linear programming in fixed dimension. In *Symposium on Computational Geometry (SoCG)*, pages 345–349, 1995.

[29] H. Edelsbrunner. *Geometry and Topology for Mesh Generation*. Cambridge University Press, 2006.

[30] J. Fakcharoenphol, S. Rao, and K. Talwar. A tight bound on approximating arbitrary metrics by tree metrics. *Journal of Computer and System Sciences (JCSS)*, 69(3):485–497, 2004.

[31] J. Gil, Y. Matias, and U. Vishkin. Towards a theory of nearly constant time parallel algorithms. In *Foundations of Computer Science (FOCS)*, pages 698–710, 1991.

[32] M. Golin, R. Raman, C. Schwarz, and M. Smid. Simple randomized algorithms for closest pair problems. *Nordic J. of Computing*, 2(1):3–27, Mar. 1995.

[33] A. Gonzalez-Escribano, D. R. Llanos, D. Orden, and B. Palop. Parallelization alternatives and their performance for the convex hull problem. *Applied Mathematical Modelling*, 30(7):563 – 577, 2006.

[34] M. T. Goodrich. Fixed-dimensional parallel linear programming via relative ϵ-approximations. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 132–141, 1996.

[35] M. T. Goodrich and E. A. Ramos. Bounded-independence derandomization of geometric partitioning with applications to parallel fixed-dimensional linear programming. *Discrete & Computational Geometry*, 18(4):397–420, 1997.

[36] P. J. Green and R. Sibson. Computing Dirichlet tessellations in the plane. *The Computer Journal*, 21(2):168–173, 1978.

[37] Y. Gu, J. Shun, Y. Sun, and G. E. Blelloch. A top-down parallel semisort. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 24–34, 2015.

[38] L. J. Guibas, D. E. Knuth, and M. Sharir. Randomized incremental construction of Delaunay and Voronoi diagrams. *Algorithmica*, 7(4):381–413, 1992.

[39] S. Har-peled. *Geometric Approximation Algorithms*. American Mathematical Society, 2011.

[40] W. Hasenplaugh, T. Kaler, T. B. Schardl, and C. E. Leiserson. Ordering heuristics for parallel graph coloring. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 166–177, 2014.

[41] S. Hong, N. C. Rodia, and K. Olukotun. On fast parallel detection of strongly connected components (SCC) in small-world graphs. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC)*, pages 1–11, 2013.

[42] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.

[43] M. Khan, F. Kuhn, D. Malkhi, G. Pandurangan, and K. Talwar. Efficient distributed approximation algorithms via probabilistic tree embeddings. *Distributed Computing*, 25(3):189–205, 2012.

[44] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, 1995.

[45] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.

[46] D. R. Llanos, D. Orden, and B. Palop. Meseta: A new scheduling strategy for speculative parallelization of randomized incremental algorithms. *International Conference on Parallel Processing Workshops*, pages 121–128, 2005.

[47] N. Megiddo. Linear-time algorithms for linear programming in R^3 and related problems. *SIAM Journal on Computing*, 1983.

[48] K. Mulmuley. *Computational geometry - an introduction through randomized algorithms*. Prentice Hall, 1994.

[49] X. Pan, D. Papailiopoulos, S. Oymak, B. Recht, K. Ramchandran, and M. I. Jordan. Parallel correlation clustering on big graphs. In *Advances in Neural Information Processing Systems (NIPS)*, 2015.

[50] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. Méndez-Lojo, D. Prountzos, and X. Sui. The tao of parallelism in algorithms. In *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*, 2011.

[51] M. O. Rabin. Probabilistic algorithms. *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39, 1976.

[52] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.

[53] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.

[54] J. H. Reif and S. Sen. Optimal randomized parallel algorithms for computational geometry. *Algorithmica*, 7(1-6):91–117, 1992.

[55] W. Schudy. Finding strongly connected components in parallel using $O(\log^2 n)$ reachability queries. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 146–151, 2008.

[56] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6(3):423–434, 1991.

[57] R. Seidel. Backwards analysis of randomized geometric algorithms. In *New Trends in Discrete and Computational Geometry*, pages 37–67. 1993.

[58] S. Sen. A deterministic $poly(\log \log n)$ time optimal CRCW PRAM algorithm for linear programming in fixed dimensions. Technical report, Department of Computer Science, University of Newcastle, 1995.

[59] J. Shun, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, A. Kyrola, H. V. Simhadri, and K. Tangwongsan. Brief announcement: the Problem Based Benchmark Suite. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 68–70, 2012.

[60] J. Shun, Y. Gu, G. Blelloch, J. Fineman, and P. Gibbons. Sequential random permutation, list contraction and tree contraction are highly parallel. In *ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pages 431–448, 2015.

[61] G. M. Slota, S. Rajamanickam, and K. Madduri. BFS and Coloring-based parallel algorithms for strongly connected components and related problems. In *International Parallel and Distributed Processing Symposium (IPDPS)*, 2014.

[62] T. H. Spencer. Time-work tradeoffs for parallel algorithms. *Journal of the ACM (JACM)*, 44(5):742–778, 1997.

[63] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

[64] D. Tomkins, T. Smith, N. M. Amato, and L. Rauchwerger. Efficient, reachability-based, parallel algorithms for finding strongly connected components. Technical report, Texas A&M University, 2015.

[65] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.

[66] U. Vishkin. Thinking in Parallel: Some Basic Data-Parallel Algorithms and Techniques. 2010.

[67] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, 1991.