Parallelism in Randomized Incremental Algorithms*

GUY E. BLELLOCH, Carnegie Mellon University
YAN GU, University of California, Riverside
JULIAN SHUN, MIT CSAIL
YIHAN SUN, University of California, Riverside

In this paper, we show that many sequential randomized incremental algorithms are in fact parallel. We consider algorithms for several problems including Delaunay triangulation, linear programming, closest pair, smallest enclosing disk, least-element lists, and strongly connected components.

We analyze the dependences between iterations in an algorithm, and show that the dependence structure is shallow with high probability, or that by violating some dependences, the structure is shallow and the work is not increased significantly. We identify three types of algorithms based on their dependences and present a framework for analyzing each type. Using the framework gives work-efficient polylogarithmic-depth parallel algorithms for most of the problems that we study.

This paper shows the first incremental Delaunay triangulation algorithm with optimal work and polylogarithmic depth. This result is important since most implementations of parallel Delaunay triangulation use the incremental approach. Our results also improve bounds on strongly connected components and least-element lists, and significantly simplify parallel algorithms for several problems.

CCS Concepts: • **Theory of computation → Parallel algorithms**; **Shared memory algorithms**; *Graph algorithms analysis*; *Sorting and searching*; **Computational geometry**;

ACM Reference Format:
Guy E. Blelloch, Yan Gu, Julian Shun, and Yihan Sun. 0. Parallelism in Randomized Incremental Algorithms. *J. ACM* 0, 0, Article 0 (0), 27 pages. https://doi.org/?.?

1 INTRODUCTION

The randomized incremental approach has been a very useful paradigm for generating simple and efficient algorithms for a variety of problems. There have been many dozens of papers on the topic (e.g., see the surveys [63, 72]). Much of the early work was in the context of computational geometry, but the approach has also been applied to graph algorithms [28, 32]. 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 appears 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

*A preliminary version of this work appeared in the Proceedings of the ACM Symposium on Parallelism in Algorithms and Architectures (SPAA), 2016 [13].

Authors' addresses: Guy E. Blelloch, Carnegie Mellon University, guyb@cs.cmu.edu; Yan Gu, University of California, Riverside, ygu@cs.ucr.edu; Julian Shun, MIT CSAIL, jshun@mit.edu; Yihan Sun, University of California, Riverside, yihans@cs.ucr.edu.

0004-5411/0/0-ART0 $15.00
https://doi.org/?.?

Problem	Work	Depth	Type
Comparison sorting (Section 3)	$O(n \log n)$	$O(\log n)$	1
d-dimensional Delaunay triangulation (Section 4)	$O(n \log n + n^{\lceil d/2 \rceil})\dagger$	$O(\log n \log^* n)$	1
2D linear programming (Section 5.1)	$O(n)\dagger$	$O(\log n)$	2
d-dimensional closest pair (Section 5.2)	$O(n)\dagger$	$O(\log n \log^* n)$	2
Smallest enclosing disk (Section 5.3)	$O(n)\dagger$	$O(\log^2 n)$	2
Least-element lists (Section 6.1)	$O(W_{SP}(n, m) \log n)\dagger$	$O(D_{SP}(n, m) \log n)$	3
Strongly connected components (Section 6.2)	$O(W_R(n, m) \log n)\dagger$	$O(D_R(n, m) \log n)$	3

Table 1. Work and depth bounds for our parallel randomized incremental algorithms. Here we assume d is a constant. $W_{SP}(n, m)$ 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. Bounds marked with a † are expected bounds, and the rest are high probability bounds. All bounds are for the arbitrary CRCW PRAM, except for comparison sorting that requires the priority-write CRCW PRAM. In all cases, the work is the same as the sequential incremental algorithm, since the algorithms are effectively equivalent beyond either reordering (Type 1 or 2) or some redundancy (Type 3). We assume here that $D_{SP}(n, m)$ and $D_R(n, m)$ are $\Omega(\log n)$.

Delaunay triangulation and convex hull, for example, are based on the randomized incremental approach [8, 23, 25, 26, 37, 46, 61, 65, 74]. In theory, however, 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 triangulation, and many other problems, is indeed parallel and leads to work-efficient polylogarithmic-depth (time) algorithms for the problems. The results are based on analyzing the dependence graph (more accurately the distribution of dependence graphs over the random order). 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 [12], the Knuth shuffle for random permutation and list/tree contraction [75], greedy graph coloring [54], and correlation clustering [64]. 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 the incremental approach leads to practical parallel algorithms [11], and to deterministic parallelism [11, 20]. Subsequent to the publication of the conference version of this paper [13], Alistarh et al. [2, 3] have shown that several parallel incremental algorithms have strong theoretical guarantees using relaxed schedulers. Furthermore, we have recently followed up on the work presented in this paper, and have shown that the randomized incremental algorithms for convex hull, half-space intersection, and finding the intersection of a set of unit circles have logarithmic dependence depth *whp*, which leads to work-optimal polylogarithmic-depth algorithms for these problems [15]. We have also shown that some algorithms in this paper can be write-efficient [14], which is motivated by the recent NVRAM technologies that writes to main memory are more costly than reads [7, 9, 10].

The contributions of the paper are summarized as follows.

1. We describe a framework for analyzing parallelism in randomized incremental algorithms. We consider three types of dependences (Type 1, 2, and 3), and give general bounds on the depth of algorithms with for each type (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). We know of no previous description and analysis of this parallel algorithm.

3. We show that an offline variant of Boissonnat and Teillaud's [21] randomized incremental algorithm for Delaunay triangulation in d dimensions has dependence depth $O(d \log n)$ with high probability (Section 4). We then describe a way to parallelize the algorithm, which leads to a parallel version with $O(d \log n \log^* n)$ depth with high probability, and $O(n \log n + n^{\lceil d/2 \rceil})$ work in expectation, on the CRCW PRAM. This is the first incremental construction of Delaunay triangulation with optimal work and polylogarithmic depth. This problem has been open for 30 years, and is important since most implementations of parallel Delaunay triangulation use the incremental approach, but none of them have polylogarithmic depth bounds. Surprisingly, our algorithm is very simple.
4. We show that classic sequential randomized incremental algorithms for constant-dimensional linear programming, closest pair, and smallest enclosing disk have shallow dependence depth (Section 5). This leads to very simple linear-work and polylogarithmic-depth randomized parallel algorithms for all three problems.
5. We show that by relaxing dependences (i.e., allowing some to be violated), two random incremental graph algorithms have (reasonably) shallow dependence depth. The relaxation increases the work, but only by a constant factor. We apply the approach to generate efficient parallel versions of Cohen's algorithm [28] for least-element lists (Section 6.1) and Coppersmith et al.'s algorithm [32] for strongly connected components (SCC, Section 6.2). In both cases we improve on the previous best bounds for the problems. Least-element lists have applications to tree embeddings on graph metrics [18, 42], and estimating neighborhood sizes in graphs [29]. Coppersmith et al.'s SCC algorithm [32] is widely used in practice [6, 55, 76, 79]. In this paper, we analyze the parallelism of this algorithm, which had been a long-standing open question. This algorithm was later implemented and shown experimentally to be practical by Dhulipala et al. [36].

Other than the graph algorithms, which call subroutines that are known to be hard to efficiently parallelize (reachability and shortest paths), 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 [56]. 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 [22]. 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 (whp)*** to indicate $O(kf(n))$ with probability at least $1 - 1/n^k$. For the graph algorithms, we use m to indicate the number of edges and n the number of vertices, and assume that $m \geq n - 1$.

2 ITERATION DEPENDENCES

An ***iterative algorithm*** is an algorithm that runs in a sequence of ***iterations*** (steps) in order. When applied to a particular input, we refer to the computation as an ***iterative computation***. Iteration j is said to ***depend*** on iteration $i < j$ if the computation of iteration j is affected by the computation of iteration i. The particular dependences, or even the number of iterations, can be a function of the input, and can be modeled as a directed acyclic graph (DAG)—the iterations

($I = 1, \ldots, n$) are vertices and dependences between them are arcs (directed edges from lower index to higher index).

Definition 2.1 (Iteration Dependence Graph [75]). An ***iteration dependence graph*** for an iterative computation is a (directed acyclic) graph $G(I, E)$ such that if every iteration $i \in I$ runs after all predecessor iterations in G have completed, then every iteration 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 the depth of each iteration can be appropriately bounded. We refer to the depth of the DAG as the ***iteration dependence depth***, and denote it as $D(G)$.

In general there can be sub-iterations nested within each iteration of an algorithm. In this case we can consider the dependences between these sub-iterations instead of the top-level iterations (i.e., a dependence from the sub-iteration in one iteration to the sub-iteration in either the same or a different iteration). The iteration dependence graph is defined analogously—dependence arcs go from earlier sub-iterations to latersub-iterations, either in the same or different top-level iterations. In this paper, we only consider one such algorithm, Delaunay triangulation, where the main iterations are over the points, and the sub-iterations are for each triangle created by adding the point.

An *incremental algorithm* takes a sequence of elements (or objects) E, and iteratively inserts them one at a time while maintaining some properties over the elements. A ***randomized incremental algorithm*** is an incremental algorithm in which the elements are added in a uniformly random order—each permutation is equally likely. In this paper, we are interested in deriving probability bounds over the iteration dependence depth. We consider three types of randomized incremental algorithms, which we refer to as Type 1, 2, and 3, for lack of better names.

2.1 Type 1 Algorithms

In these algorithms we show the probability bounds on iteration dependence depth by considering all possible paths of dependences. By bounding the probability of each path, and bounding the number of possible paths, the union bound can be used to bound the probability that any path is long. We use backwards analysis [72] to analyze the length and number of paths.

We say that an incremental algorithm has *k-bounded dependences* if for any input E and element $x \in E$ inserted last, x directly depends on at most k other elements—i.e., once those up to k other elements are inserted, x can be inserted. For example, consider sorting by inserting into a binary search tree (BST) based on a random order. For any key v inserted last, once the previous and next keys in sorted order, v_p and v_n, have been inserted, we can immediate insert v. In particular the key v will either be the right child of v_p or the left child of v_n depending on which of the two was inserted later. More subtly, the search path for v will also be the same once v_p and v_n are both inserted (more discussion in Section 3). Inserting into a BST therefore has 2-bounded dependences.

If the iterations in an incremental algorithm are nested, then we consider the pairs of an element along with each of its sub-iterations. We say that an incremental algorithm has *k-bounded nested dependences* if for any input E, any element $x \in E$ inserted last, and any sub-iteration s for x, (x, s) directly depends on at most k possible previous element–sub-iteration pairs. We say "possible" here since the sub-iterations might differ depending on the order of the previous elements. For example, consider Delaunay triangulation in d dimensions. Inserting an element (point) x will run sub-iterations adding a set of triangles (d-simplices). As shown in Section 4, each new triangle (sub-iteration) will depend on at most two previous triangles. Each of these previous triangles could have been added by a sub-iteration of any of its $(d + 1)$ corner points, whichever was inserted last.

Hence there are $2(d+1)$ possible element–sub-iteration pairs that the sub-iteration for x could depend on, and Delaunay triangulation therefore has $2(d+1)$-bounded nested dependences.

For a given insertion order of all elements, a *tail* is an element (or element–sub-iteration pair for the nested case) that no other element (or element-sub-iteration pair) depends on. The *tail count* is the number of possible tails over all orderings. Inserting n keys into a BST, for example, has a tail count of n since every key can be a tail. For Delaunay triangulation every final triangle can be involved in a tail, and each one created by any of its corners, depending on which corner is last. Therefore the tail count is at most $(d+1)$ times the number of final triangles.

We are now interested in the length of dependence paths for incremental algorithms with k bounded dependences, either nested or not, and how that limits the iteration dependence depth.

Theorem 2.2. *Consider a randomized incremental algorithm on n elements with k-bounded (nested) dependences, and for which the tail count is bounded by cn^b, for some constants b and c. Over the distribution of iteration dependence graphs G, and for all $\sigma \geq ke^2$:*

$$\Pr(D(G) \geq \sigma H_n) < cn^{-(\sigma-b)}$$

where $H_n = \sum_{i=1}^{n} 1/i$.

Proof. We use backwards analysis by considering removing elements one-by-one from the last iteration. We analyze a specific dependence path to a tail and then take a union bound over all possible paths. We use the term *point* to mean an element for the non-nested case, or an element–sub-iteration pair for the nested case.

To consider the single specific path, we start at iteration n working backwards, and let i be the iteration number. On each iteration, x_i will correspond to a particular point, and e_i to its corresponding element ($e_i = x_i$ for the non-nested case). Consider one of the possible cn^b tails. It corresponds to a single point, which we set as x_n. The probability of the event "element e_i is inserted at iteration $i = n$" is $1/n$ since all permutations are equally likely. If e_i is inserted at i, then the iteration i is on the dependence path for that tail. We then arbitrarily choose one of the k points that x_i depends on and set x_{i-1} to that point. The fact there are k of them will be handled in the union bound. If e_i is not inserted at i, then $x_{i-1} = x_i$. Now we move back to $i = n-1$, and the probability e_i is inserted at i is again $1/i$. This is true whether $x_n = x_{n-1}$ or not—in both cases we are looking for a single element out of i possible elements. By repeating this process until $i = 1$, the probability that element e_i is inserted at iteration i is always $1/i$ for all i. Each time e_i is inserted at i, we extend the dependence path by 1 and make another arbitrary choice among up to k predecessors, setting x_{i-1} with our choice, otherwise it is not extended.

Let L be a random variable corresponding to the total number of dependences on the path we are considering. We therefore have that $E[L] = \sum_{i=1}^{n} \frac{1}{i} = H_n$. Furthermore, each event (e_i inserted at i) is independent since each element is chosen at random from the remaining i elements. Using a Chernoff bounds, we obtain:

$$P[L \geq \sigma E[l]] < \left(\frac{e^{\sigma-1}}{\sigma^\sigma}\right)^{E[l]} < \left(\frac{e}{\sigma}\right)^{\sigma E[l]} = \left(\frac{e}{\sigma}\right)^{\sigma H_n}.$$

We now take a union bound over the cn^b possible tails and the at most k^l possible choices we make for a predecessor for a dependence path of length l. Note that any path longer than l must have a length l path as a prefix, and so we need not consider the longer paths when taking the union bound. For $\sigma \geq ke^2$ and $l = \sigma H_n$, we therefore have:

$$\begin{aligned} P[D(G) \geq \sigma H_n] &\leq cn^b k^{\sigma H_n} \cdot P[L \geq \sigma H_n] \\ &< cn^b k^{\sigma H_n} \left(\frac{e}{\sigma}\right)^{\sigma H_n} \\ &= cn^b \left(\frac{ke}{\sigma}\right)^{\sigma H_n} \\ &\leq cn^b \left(\frac{1}{e}\right)^{(\ln n)\sigma} \\ &= cn^{-(\sigma - b)} . \end{aligned}$$

□

The Type 1 algorithms that we describe can be parallelized by running a sequence of rounds. Each round checks all remaining iterations to see if their dependences have been satisfied and runs the iterations if so. They can be implemented in two ways: one completely online, only seeing a new element at the start of each iteration, 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., [51]), and in the second case the position of each uninserted element is kept up-to-date on every iteration (e.g., [27]). 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.

2.2 Type 2 Algorithms

Type 2 incremental algorithms have a special structure. The iteration dependence graph for these algorithms is formed as follows: each iteration j is either a *special iteration* or a *regular iteration* (depending on insertion order and the particular element). Each special iteration j depends on all iterations $i < j$, and each regular iteration depends on the closest earlier special iteration. The first iteration is special. Furthermore, the probability of being a special iteration is upper bounded by c/j for some constant c and is independent of the choices for iterations $j + 1$ to n. For Type 2 algorithms, when a special iteration i is processed, it will check all previous iterations, requiring $O(i)$ work and depth denoted as $d(i)$, and when a non-special iteration is processed it does $O(1)$ work.

Theorem 2.3. *A Type 2 incremental algorithm has an iteration dependence depth of* $O(\log n)$ whp*, and can be implemented to run in* $O(n)$ *expected work and* $O(d(n) \log n)$ *depth* whp*, where* $d(n)$ *an upper bound on the depth of processing a special iteration as a function of* n*.*

Proof. Since the probability of a special iteration is bounded by c/j independently of future iterations, the expected number of special iterations is $\sum_{j=1}^{n} c/j = O(\log n)$, and using a Chernoff bound, the number of special iterations is $O(\log n)$ *whp*. By construction, there cannot be more than two consecutive regular iterations in a path of the iteration dependence graph, and so the iteration dependence depth is at most twice the number of special iterations and hence $O(\log n)$ *whp*.

We now show how parallel linear-work implementations can be obtained. A parallel implementation needs to execute the special iterations one-by-one, and for each special iteration it can do its computation in parallel. For the non-special iterations whose closest earlier special iteration has been executed, their computation can all be done in parallel. To maintain work-efficiency, we cannot afford to keep all unfinished iterations active on each round. Instead, we start with a constant number of the earliest iterations on the first round and on each round geometrically

ALGORITHM 1: Type 2 Algorithm

Input: Iterations $[0, \dots, n)$.

```
run special iteration 0
j ← 1
for r ← 2 to log_2 n do
    while j < 2^{r-1} do
        parallel foreach k ∈ {j, ..., 2^{r-1} − 1} do
            F[k] ← check if iteration k is special
        l ← minimum true index in F, or 2^{r-1} if none
        parallel foreach k ∈ {j, ..., l − 1} do
            run regular iteration k
        if l < 2^{r-1} then
            run special iteration l
        j ← l
```

increase the number of iterations processed, similar to the prefix methods described in earlier work on parallelizing iterative algorithms [12].

Pseudocode is given in Algorithm 1. Without loss of generality, assume $n = 2^k$ for some integer k. We refer to the outer **for** loop as rounds, and the inner **while** loop as sub-rounds. Each round r processes iterations $2^{r-2}, \dots, 2^{r-1} - 1$, which we refer to as a *prefix*. The variable j at the start of each sub-round indicates that all iterations before j are done, and all iterations at or after j are not. Each sub-round finds the first unfinished special iteration l within the round, if any. It then runs all regular iterations up to l (all of their dependences are satisfied). Finally, if a special iteration was found, that special iteration is run (all of its dependences are satisfied). Finding the first unfinished special iteration requires computing a minimum, which can be done in $O(2^r)$ work and $O(1)$ depth *whp* on an arbitrary CRCW PRAM [81]. Running all regular iterations also requires $O(2^r)$ work and $O(1)$ depth, and running the special iteration requires $O(2^r)$ work and $O(d(n))$ depth. The number of sub-rounds within a round is one more than the number of special iterations 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 in round r is $O(2^{r-1})$ in expectation, and summed over all rounds is $O(1) + \sum_{r=2}^{\log n} O(2^{r-1}) = O(n)$ in expectation. The number of sub-rounds is bounded by the number of special iterations plus $\log_2 n$, and each sub-round has depth $O(d(n))$, and so the total depth is $O(d(n) \log n)$ *whp*. □

2.3 Type 3 Algorithms

In the third type of incremental algorithms it is safe to run iterations in parallel, but this can require extra work with respect to the sequential algorithm. In these algorithms an iteration can "separate" future iterations. A simple example, again, is insertion into a binary search tree, where the first key inserted separates keys less than it from ones greater than it. The idea is to then process the iterations in rounds of increasing powers of two, as in the Type 2 case. However in this case, every iteration in a round will run as if it is at the beginning of the round ignoring conflicts within the round, and then resolving the conflicts at the end of the round. We apply this approach to two graph problems: least-element (LE) lists and strongly connected components (SCC).

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 $x \in S$. For a DAG, the ordering could be a topological sort, and possibly different for each vertex since topological sorts are not unique.

ALGORITHM 2: Type 3 Algorithm

Input: Iterations $[0, \ldots, n)$.

```
run iteration 0
for r ← 1 to log_2 n do
    parallel foreach k ∈ {2^{r−1}, ..., 2^r − 1} do
        Run iteration k as if at iteration 2^{r−1}, i.e., using the final state from the previous round
    parallel foreach k ∈ {2^{r−1}, ..., 2^r − 1} do
        Combine state such that earlier k have higher priority
        (Final state should be the same as if run sequentially up to 2^r − 1)
```

For both our applications, LE-lists and SCC, the orderings can be different for each element. The distinct orders is the innovative aspect of our analysis.

Definition 2.4 (separating dependences). An incremental algorithm has ***separating dependences*** if for all input S: (1) it has total orderings $<_x, x \in S$, and (2) 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). Let $d_{(i,j)}$ be the event that there is a dependence from iteration i to iteration j, and $p(d_{(i,j)})$ be its probability over all insertion orders.

Lemma 2.5. *In a randomized incremental algorithm that has separating dependences, we have that $p(d_{(i,j)})) \leq 2/i$ for $1 \leq i < j \leq n$.*

Proof. Consider the total ordering $<_j$. Among the elements inserted in the first i iterations, at most two of them are the closest (by $<_j$) to the element inserted at iteration j (at most one on each side). There will be a dependence from iteration i to j only if the element selected on iteration i is one of these two—otherwise iterations before i would have separated i from j. Since all of the first i elements are equally likely to be selected on iteration i, the probability is at most $2/i$. □

Corollary 2.6. *The number of dependences in a randomized incremental algorithm with separating dependences is $O(n \log n)$ in expectation (also true* whp *as given in Corollary 2.9).*

This comes simply from the sum $\sum_{j=2}^{n} \sum_{i=1}^{j-1} p(d_{(i,j)})$ which is upper bounded by $2n \ln n$. This leads, for example, to a 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 [33]. 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 the processing order 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, as shown in Algorithm 2. This means that we might not separate iterations that were separated in the sequential order. For example, if $a <_c b <_c c$ and in the sequential order a is inserted first then b and then c, then b will separate a from c, avoiding a dependence between them. However in the round-based parallel order, a could be in one round, and then b and c in another later round. In this case b will not separate a from c allowing for a dependence from a to c. We therefore have to consider c as running at the start of the round (position 2^{r-1}) in determining the probability $p(d_{a,c})$. This can cause additional dependences, and hence work, but as argued in the theorem below, the number of dependences is within a constant factor. The second parallel loop is needed to combine results from the iterations that are run in parallel. The technique here depends on the algorithm, but is reasonably simple for the algorithms we consider for LE-Lists and SCC.

As an example, consider applying the approach to insertion into a binary search tree. On each round r, 2^{r-1} keys are already inserted into a BST and in parallel we try to insert the next 2^{r-1} keys. In the first loop all new keys will search the tree for where they belong. Many will fall into their own leaf and be happy, but there will be some conflicts in which multiple keys fall into the same leaf. The second loop would resolve these conflicts. This is a different parallel algorithm than the Type 1 algorithm described in Section 3.

We say that iteration a has a left (right) dependence to a later iteration b if b depends on a and $a <_b b$ ($b <_b a$). This definition is used to bound the total number of dependences of a specific iteration as follows.

Lemma 2.7. *When applying Algorithm 2 to an incremental algorithm with separating dependences, let $p_{rj}(l), j \geq 2^r$ be the probability that l iterations in round r have a left dependence to iteration j. Then for all r and j, we have $p_{rj}(l) \leq 2^{-l}$.*

Proof. Clearly $p_{rj}(0) \leq 2^{-0} = 1$. The probability that among iterations $\{0, \ldots, 2^r - 1\}$, the closest iteration to j based on $<_j$ appears among $\{2^{r-1}, \ldots, 2^r - 1\}$ is $1/2$ (since elements are in random order). Therefore $p_{rj}(1) \leq 1/2$. Now given that the first is closest, the probability that the second is closest out of the remaining iterations in $\{2^{r-1}, \ldots, 2^r - 1\}$ is $(2^{r-1} - 1)/(2^r - 1) < 1/2$. Hence, the probability for $l = 2$ is less than $1/4$. This repeats so $p_{rj}(l) < 2^{-l}$ for $l > 1$, giving our bound. □

We can make the symmetric argument about dependences on the right. Importantly the expected number of dependences from a round to a later element is constant, and the probability that the number of dependences is large is low.

Theorem 2.8. *A randomized incremental algorithm with separating dependences can run in $O(\log n)$ parallel rounds over the iterations and every iteration will have $O(\log n)$ incoming dependences* whp *(for a total of $O(n \log n)$* whp*).*

Proof. We just consider left dependences, the right ones will just double the count. For fixed j the upper bounds on the probabilities p_{rj} are independent across the rounds r. This is because working backwards each round picks a random set of $1/2$ the remaining elements. The round that iteration j belongs to contains fewer dependences than previous rounds, and the rounds later have no dependences to iteration j. Therefore, $p_{rj}(l) \leq 2^{-l}$ holds for all rounds even when $j < 2^r$.

For a set of independent random variables X_i with exponential distribution $X_i \sim Exp(a)$, the sum $X = \sum X_i$ satisfies the following tail bounds [57]:

$$P(X > \sigma \mathbb{E}[X]) \leq \sigma e^{-a\mathbb{E}[X](\sigma - 1 - \ln \sigma)}$$

For $\log_2 n$ rounds, $\mathbb{E}[X] = (\log_2 n)/a$ and $P(X > (\sigma/a) \log_2 n) \leq \sigma n^{-(\sigma-1-\ln \sigma)}$ which satisfies the high probability condition. Since by Lemma 2.7 our distributions are sub-exponential, the tails are no larger. □

Corollary 2.9. *A sequential randomized incremental algorithm with separating dependences will have $O(\log n)$ incoming dependences per iteration* whp.

ALGORITHM 3: IncrementalSort

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

```
// *P reads the value from pointed to by P.
// P → x reads field x via pointer P.
// The check on Line 8 is only needed for the parallel version.
1  Root ← a pointer to a new a location containing a null pointer
2  for i ← 1 to n do
3      N ← newNode(k_i)
4      P ← Root
5      while true do
6          if *P = null then
7              write N into the location pointed to by P
8              if *P = N then
9                  break // write succeeded and iteration i is done, always true sequentially
10         if N → key < (*P) → key then
11             P ← pointer to (*P) → left
12         else
13             P ← pointer to (*P) → right
14 return Root
```

Proof. As discussed above, every dependence in the sequential order will also appear in the parallel order so the number of dependences is at most as many. □

Theorem 2.8 does not explicitly give the work and depth for an algorithm since it will depend on the costs of running each iteration. These will be given for the particular algorithms in Section 6.

3 COMPARISON SORTING (TYPE 1)

We first consider how to use our framework for sorting by incrementally inserting into a binary search tree (BST) with no re-balancing. For simplicity, we assume that no two keys are equal. It is well-known that for a random insertion order, inserting into a BST takes $O(n \log n)$ time (comparisons) in expectation, and even with high probability. We apply our Type 1 approach to show that the sequential incremental algorithm is also efficient in parallel. Algorithm 3 gives pseudocode that works either sequentially or in parallel. An iteration is one round 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 earliest iteration 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. The only important dependence is the last one on the path, since all the others are subsumed by the last one (i.e., they do not appear in the transitive reduction of the dependence graph).

Lemma 3.1. *Insertion of n keys into a binary search tree in random order has iteration dependence depth $O(\log n)$ whp.*

Proof. When inserting an element at iteration i (removing in backwards analysis), there are at most two keys it can directly depend on, the previous and the next in sorted order (it is at most since there might not be a previous or next key). This is among the keys from iterations 1 to $i - 1$.

Therefore there is a 2-bounded dependence for all iterations. Every key can be a tail (a leaf in the final binary search tree), so the tail count is n. Using Theorem 2.2 we therefore have that the iteration depth is bounded by σH_n for $\sigma > 2e^2$ with probability at most $n^{-\sigma+1}$. □

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., in Delaunay triangulation in the next section, the dependence structure is not a tree.

Theorem 3.2. *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* whp *on a priority-write CRCW PRAM.*

Proof. They generate the same tree since whenever there is a dependence, the earliest iteration wins. The number of rounds of the while loop is bounded by the iteration dependence depth ($O(\log n)$ *whp*) 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 [30], although it is on a stronger model (priority-write CRCW instead of EREW) and is randomized.

4 DELAUNAY TRIANGULATION (TYPE 1)

A Delaunay triangulation (DT) in d dimensions is a triangulation of a set of points P in $\mathbb{R}^d$ such that no point in P is inside the *circumsphere* of any triangle (the sphere defined by the triangle's $d + 1$ corners). Here we will use *triangle* to mean a d-simplex defined by $d + 1$ corner points, and use *face* to mean a $d - 1$ simplex with d corner points. We say a point ***encroaches*** on a triangle if it is in the triangle's circumsphere, and will assume for simplicity that the points are in general position, i.e., no $k \leq d + 1$ points on a $(k - 2)$-dimensional hyperplane, or $k \leq d + 2$ points on a $(k - 2)$-dimensional sphere. Delaunay triangulation for $d = 2$ can be solved sequentially in optimal $O(n \log n + n^{\lceil d/2 \rceil})$ work. There are also several work-efficient (or near work-efficient) parallel algorithms for $d = 2$ that run in polylogarithmic depth [19, 31, 69], and at least one for higher dimension [5], but they are all complicated. We assume that d is a constant.

The widely-used and simple incremental Delaunay algorithms date back to the 1970s [49]. 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 triangles from p to each boundary face of the cavity. The algorithms differ in how the encroached triangles are found, and how they are ripped and tented. Clarkson and Shor [27] first showed that randomized incremental convex hull is efficient, running with work $O(n \log n + n^{\lfloor d/2 \rfloor})$ in expectation. These results imply optimal $O(n \log n + n^{\lceil d/2 \rceil})$ work for DT.

Guibas et al. [51] (GKS) showed a simpler direct randomized incremental algorithm for 2d DT with optimal expected time bounds, and this has become the standard version described in textbooks [34, 40, 63] and often used in practice. The GKS algorithm uses a history of triangle updates to locate a triangle t that a new point p encroaches. It then searches out for all other encroached triangles flipping pairs of triangles as it goes. Edelsbrunner and Shah [41] generalized the GKS method to work in arbitrary dimension with optimal work, again in expectation. The algorithms, however, are 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 iteration can take linear depth.

ALGORITHM 4: IncrementalDT

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

$t_b \leftarrow$ a sufficiently large bounding triangle
$E(t_b) \leftarrow V$
$M \leftarrow \{t_b\}$
for $i \leftarrow 1$ *to* n **do**
  Let $R \leftarrow \{t \in M \mid v_i \in E(t)\}$
  foreach face f on the boundary of R **do**
    $(t, t_o) \leftarrow$ the two triangles incident on f, with t on the v_i side
    ReplaceBoundary(t_o, f, t, v_i)
return M

function ReplaceBoundary(t_0, f, t, v)
  $t' \leftarrow$ a new triangle consisting of f and v
  $E(t') \leftarrow \{v' \in E(t) \cup E(t_o) \mid \textsc{InCircle}(v', t')\}$
  detach t from face f in M
  add t' to M

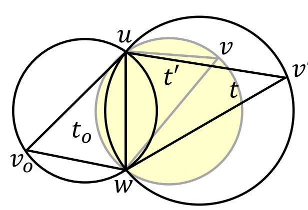

Fig. 1. An illustration of the procedure of ReplaceBoundary(f, v) on a new point v in two dimensions. In this case, the boundary face f is (u, w). The function will detach t and replace it with t'. The new triangle t' only depends on t_o and t. In support of Fact 4.1, we note that since v encroaches on t but not t_o, it must be in the larger black ball but not the smaller one. Therefore, the yellow ball must be contained in the union of the two black balls and must contain the intersection of them.

Boissonnat and Teillaud [21] (BT) consider a somewhat different but equally simple direct random incremental algorithm for DT that does optimal work in expectation for arbitrary dimension. Instead of using the history to locate a single triangle that a point p encroaches and then searching out from it for the rest, it locates all encroached triangles directly using the history. It therefore does not suffer the inherent sequential bottleneck of GKS.

Our result. Here we show that an offline variant of the BT algorithm has iteration dependence depth $O(\log n)$ *whp*. We further show that the iterations can be parallelized leading to a very simple parallel algorithm doing no more work than the sequential version (i.e., optimal), and with dependence depth $O(\log n)$ depth *whp*.

Our sequential variant of BT is described in Algorithm 4. For each triangle $t \in M$, the algorithm maintains the set of uninserted points that encroach on t, denoted as $E(t)$. On each iteration i, the algorithm identifies the boundary of the region that point i encroaches on, and for each face f of that region it detaches the triangle on the inside and replaces it with a new triangle t' consisting of f and v. All work on uninserted points is done in determining $E(t')$, which only requires going through two existing sets, $E(t)$ and $E(t')$. This is justified by Fact 4.1. Determining the boundary of the region can be implemented efficiently by maintaining a mapping from each point to the simplices it encroaches, and checking those simplices.

Fact 4.1 ([21]). *Given two d-simplices t and t_o that share a face f, and a point v that encroaches on t but not t_o, then for $t' = (f, v)$ we have $E(t) \cap E(t_o) \subseteq E(t') \subseteq E(t) \cup E(t_o)$.*

This fact is proven in [21], and an illustration of it is given in Figure 1. A work bound for IncrementalDT of $O(n \log n + n^{\lceil d/2 \rceil})$ follows from the analysis of Boissonnat and Teillaud [21]. Later we show a more precise bound on the number of InCircle tests for $d = 2$, giving an upper bound on the constant factor for the dominant term.

In the following discussion, given a set of points $V' \subseteq V$, the notation $\min(V')$ indicates the earliest point in V' by insertion order, and that comparing two points, as in $v < v'$, compares their position in the insertion order.

Dependence Depth and A Parallel Version

We now consider the dependence depth of the algorithm. One approach is to consider dependences among the outer iterations (adding each point). Unfortunately it seems difficult to prove a logarithmic bound on dependence depth for such an approach. The problem is that although a point will encroach on a constant number of triangles and associated points in expectation, in some cases it could encroach on up to a linear number. Hence it does not have a k-bounded dependence. It seems that although expectation is good enough for the work bound, it does not suffice for the depth bound since we need to consider maximum depth over multiple paths.

We therefore consider a more fine-grained dependence structure based on the triangles (sub-iterations) instead of points (top level iterations). The observation is that not all triangles added by a point v need to be added on the same round. This will allows us to show that Algorithm 4 has k bounded nested dependences. We will make use of the following Lemma.

Lemma 4.2. *In Algorithm 4 the function (sub iteration) ReplaceBoundary(t_o, f, t, v) will be applied if and only if t_o and t share the face f at some point in the algorithm, and $v = \min(E(t))$ with $v < min(E(t_o))$.*

Proof. Firstly, v must be later than the points defining t_0 otherwise t is detached before t_0 is created and the two never share a face. Once t and t_o are created the only points that can remove them are points that encroach on the triangles. Since v is the earliest such point, and only encroaches on t, when running the iteration that inserts v, the triangles t_o and t will still be there, and ReplaceBoundary(t_o, f, t, v) will be applied. This is the only case when ReplaceBoundary will be applied. □

We can now define a dependence graph $G_T(V) = (T, E)$ based on Lemma 4.2. The vertices T corresponds to triangles created by Algorithm 4 (each sub-iteration), and for each call to ReplaceBoundary(t_o, f, t, v) we include an arc from each of t_o and t to the new triangle it creates t'.

Theorem 4.3. *Algorithm 4 has iteration dependence depth $O(d \log n)$* whp *over the random orders of V, i.e. $D(G_T(V)) = O(d \log n)$* whp.

Proof. This follows from Theorem 2.2. In particular the algorithm has a $2(d+1)$-bounded nested dependence. Each creation of a triangle (sub-iteration) by a point v depends on at most two previous triangles (by Lemma 4.2, each of which depends on at most $d + 1$ points (the corners of a triangle). Therefore adding the triangle for point v depends on $2 \cdot (d + 1)$ possible previous sub-iterations. It

ALGORITHM 5: ParIncrementalDT

Input: A sequence $V = \{v_1, \ldots, v_n\}$ of points in $\mathbb{R}^d$.
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 (t_o, t) sharing a face f ∈ M, s.t. min(E(t)) < min(E(t_o)) do
6         ReplaceBoundary(t_o, f, t, min(E(t)))
7 return M
```

is important to note that in a given run there will only be dependences to the two triangles, but the definition of k-bounded dependence requires we consider all possible dependences to sub-iterations and associated elements (points). The tail count (the number of possible sub-iterations that ended a dependence chain) is bounded by the number of triangles in the result, $O(n^{\lceil d/2 \rceil})$, times the number of points that could have generated each triangle, at most $(d+1)$, giving a total of $O(dn^{\lceil d/2 \rceil})$. Plugging into Theorem 2.2, gives a dependence depth of

$$\Pr(D(G) \geq \sigma H_n) < dn^{-(\sigma-(d+1)/2)}$$

for $\sigma \geq 2(d+1)e^2$, satisfying the bounds. □

Since we consider d to be constant, the d can be dropped from the bounds, but it might be useful to understand the dependence on d, so we left it in for this Lemma (i.e., not assuming d is constant).

Algorithm 5 describes a parallel variant of Algorithm 4 based on the dependence structure. On each round the parallel algorithm applies ReplaceBoundary($t_o, f, t, \min(E(t))$ to all faces that satisfy the conditions of Lemma 4.2—t and t_o are present, and $\min(E(t)) < \min(E(t_o))$. The subroutine ReplaceBoundary is unchanged. Because of Lemma 4.2, the parallel variant will make exactly the same calls to ReplaceBoundary as the sequential variant, just in a different order. We note that since the triangles for a given point can be added on different rounds, the triangulation is not necessarily self consistent after each round. Importantly, a face might only have one adjacent triangle. In that case the face cannot proceed until it receives the second triangle (or is the boundary of the DT). Also the faces of a triangle can be detached on different rounds. This does not affect the algorithm—once all boundary faces of a point have been replaced, the old interior will be fully detached from the new triangulation.

To implement the algorithm one can maintain three data structures: (1) the set of triangles that have been created, each with the set of points that encroach on it, (2) a hash-map that maps faces to their up to two neighboring triangles, and (3) the set of faces that satisfy the condition on line Line 5, which we refer to as the active faces. The hash-map is indexed on the d corners of a face in some canonical order. Each round goes over all the active faces in parallel, and runs ReplaceBoundary. This involves first looking up the neighboring triangles, running the InCircle tests across their points in parallel, and filtering out the ones that return true. The algorithm also finds the minimum indexed such point. Then the new triangle is added to the triangle set, the $d+1$ faces of the new triangle are updated in the hash-map (some might be new), and the subset of them that satisfy Line 5 are added to the set of active faces.

Most steps are easily parallelizable. Applying and filtering on the InCircle tests, and allocating the new active faces for each ReplaceBoundary, can use processor allocation and compaction.

This can be done approximately—i.e., into a constant factor larger set of locations. On the CRCW PRAM the approximate version can be be implemented work efficiently in $O(\log^* n)$ depth *whp* [44]. On the CRCW PRAM the hash table operations and the minimum can also can be done work efficiently in $O(\log^* n)$ depth *whp* [44, 52].

Theorem 4.4. *ParIncrementalDT (Algorithm 5) runs in* $O(\log n \log^* n)$ *depth* whp*, and with work* $O(n \log n + n^{\lceil d/2 \rceil})$ *in expectation, 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 dependences in the algorithm. Each round has depth $O(\log^* n)$ *whp* as described above, so the overall depth is as stated. The work of the algorithm is the same as the sequential work [21] since the calls to ReplaceBoundary are the same, and all steps are work-efficient. We assume d is constant. □

Work bound for $d = 2$

Putting constants into the proof of the work bounds in GKS [51], and appearing in some textbooks [34], gives an upper bound of $36n \ln n + O(n)$ on the expected number of InCircle tests for 2-d Delaunay triangulation. The argument, very roughly, is that every point encroaches on 4 triangles in expectation on each iteration, and each triangle has 3 points. Now each of these triangles can involve one, two, or three InCircle tests for an encroaching point when it is removed (depending on how many of its edges are on the boundary of the encroached region). This gives an upper bound of an expected $3 \times 4 \times 3/i$ per iteration i leading to the $36n \ln n + O(n)$ upper bound.

Here we tighten the bounds. In the proof we take advantage that, due to Fact 4.1, the InCircle test is not required for points that appear in both $E(t_o)$ and $E(t)$ since they will always appear in $E(t')$. We know of no previous work that gives this bound.

Theorem 4.5. *IncrementalDT for* $d = 2$ *and on* n *points in random order does at most* $24n \ln n + O(n)$ *InCircle tests in expectation.*

Proof. We denote the point added at iteration i as x_i. For an iteration j we consider the history of iterations $i < j$, and we are interested in the ones that do InCircle tests on x_j. For each such iteration we consider the boundary of the region that x_j encroaches immediately after iteration i. We will bound the number of InCircle tests on point x_j based on the changes to this boundary over the iterations. We define each face of the boundary by its two endpoints (u, w) along with the (up to) two points sharing a triangle with (u, w), which we denote as the four tuple $(u, w; v_l, v_r)$, and refer to as a *winged edge*. For example in Figure 1 the winged edge $(u, w; v_o, v)$ corresponds to the edge (u, w) after adding v.

In ReplaceBoundary a point is only tested for encroachment (an InCircle test) if its boundary winged edge $(u, w; v_l, v_r)$ is being deleted, and replaced with another. This is because a point only needs to be tested if it encroaches on one side (one wing) and not the other. It seems to be messy to keep track of the deletions, however, so instead we keep track of additions of these boundaries. We can then charge each deletion against the addition—i.e., we do the InCircle test on the deletion, but "pay" for it earlier on the addition. This means we have to include some charge for the initial additions at the start of the algorithm. This is 3 per point, one for each edge of the bounding triangle. However, when we add x_j it has at least 3 boundaries we don't have to pay for, so the net additional tests needed for this accounting method is at most zero. Let Y_{ij} be the random variable specifying the number of boundaries for point x_j that iteration i adds (i.e., winged edges that include x_i, and are on the boundary of x_j's encroached region when added). The total number of InCircle tests C

is then bounded by

$$C \leq \sum_{j=2}^{n} \sum_{i=1}^{j-1} Y_{ij} \ .$$

To analyze the expectation $E[Y_{ij}]$ we note that we can consider point x_j as immediately following iteration i (since no other point $x_k, k > i$ has been added yet). All points $x_1, \ldots, x_i, x_j$ are equally likely to be selected as x_j, so the expected number of boundaries for x_j is at most 6 (due to the fact that planar graphs can have average degree at most 6). Each boundary winged edge has 4 points that could create it, any of which could be at position i. Therefore $E[Y_{ij}]$, is upper bounded by the at most 6 boundaries in expectation, times the at most four points (worst case) and divided by the i possible points $x_1, \ldots, x_i$, each equally likely. This gives $E[Y_{ij}] \leq 6 \times 4/i = 24/i$, leading to the claimed result:

$$\mathbb{E}[C] \leq \sum_{j=2}^{n} \sum_{i=1}^{j-1} E[Y_{ij}] = \sum_{j=2}^{n} \sum_{i=1}^{j-1} \frac{24}{i} \leq 24n \ln n + O(n).$$

□

5 LINEAR-WORK ALGORITHMS (TYPE 2)

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 2.2, and their iteration depth is polylogarithmic *whp*. To obtain linear-work parallel algorithms, we process the iterations in prefixes, as described in Section 2.2. 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 [1, 4, 24, 35, 39, 47, 48, 73]. 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 [71]. Seidel's [71] elegant and very simple randomized incremental algorithm 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 iterations, and fits in the framework of Type 2 algorithms from Section 2.2. The iterations corresponding to inserting a tight constraint are the special iterations. Special iterations depend on all earlier iterations because when a tight constraint executes, it needs to look at all earlier constraints. Non-special iterations depend on the closest earlier special iteration i because it must wait for iteration 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 iteration j has a probability of at most $2/j$ of being a special iteration because the optimum is defined by at most two constraints and the constraints are added in a randomized order. Furthermore, the probabilities (event of being a special iteration) are independent among different iterations.

As described in the proof of Theorem 2.3, our parallel algorithm executes the iterations 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 iteration takes $O(1)$ work and processing a violating constraint at iteration i takes $O(i)$ work and $O(1)$ depth *whp* 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. *Seidel's randomized incremental algorithm for 2D linear programming has iteration dependence depth $O(\log n)$ and can be parallelized to run in $O(n)$ work in expectation and $O(\log n)$ depth* whp *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. For any constant dimension d this increases the iteration dependence depth (and hence the depth of the algorithm) to $O(\log^{d-1} n)$ *whp*. The work remains $O(n)$ as in the sequential algorithm [71]. We note that although the work is optimal in n, its depth is not as good as the best parallel algorithms [4], but is very much simpler.

5.2 Closest Pair

The ***closest pair*** problem takes as input a set of points in Euclidean space in d dimensions 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 [45, 53, 59, 66] works by maintaining a grid and inserting the points into the grid in a random order. If r is the distance of the closest pair seen so far (initialized to the distance between the first two points), the grid partitions space into cubic regions of length r in each dimension, and where each non-empty region stores the points inside the region. The grid is maintained using a hash table so that only non-empty regions need to be represented. Each region contains at most $O(1)$ points since the points have distance at least r from each other. Whenever a new point is inserted, one can check the region the point belongs to and the $3^d - 1$ adjacent regions to see whether there are any points in those regions that are closer than r (any point closer than distance r must be in those regions). If so, r is updated with the distance to the nearest point, and the grid has to be rebuilt by inserting all points into a new hash table based on the new r. The check takes $O(1)$ work and the rebuild takes $O(i)$ work (*whp* because of hashing). 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 $\sum_{i=1}^{n}(O(1) + O(i) \cdot (2/i)) = O(n)$ in expectation.

This is a Type 2 algorithm, and the iteration dependence graph is similar to that of linear programming, with a dependence depth is $O(\log n)$ *whp*. To obtain a linear-work parallel algorithm, we again execute the algorithm in prefixes. The special iterations are the ones that cause the grid to be rebuilt. This happens if a point in the prefix has a distance closer than r to a point already inserted in a previous round, or to another point that appears earlier in the prefix. To find the special steps, we insert all points in the prefix to a copy of the hash table, and for each point in the current prefix, we check the $3^d - 1$ adjacent regions of its region to see if it forms a closer pair. Inserting into a parallel hash table takes $O(i)$ work and $O(\log^* i)$ depth *whp* for a set of i points [44].

We now argue that when checking a region, a point only does $O(1)$ work in expectation. A region will have $O(1)$ points from previous rounds, and can accomodate $O(1)$ points from the current prefix without causing a pair with distance closer than r to be formed. If a closer pair is formed, then we must account for the work of checking the newly added points in the region. Consider the first z closest pairs among the points in this region up to and including this prefix. Since the points are given in a random order, with $1/4$ probability, both points of a pair have already been inserted prior to this round, and with 3/4 probability, at least one point is a newly inserted point in this round. Although the pairs are not independent, there must be at least $\sqrt{z}$ independently chosen points among z pairs. Therefore, the probability that none of the z closest pairs involve points prior to this round (which means a closer pair is found with a point in the current prefix) is at most $(3/4)^{\sqrt{z}}$. If a closer pair is found, we will incur $O(z)$ additional work for checking these extra points in the region. Therefore, the expected work for checking a region is $O(1+\sum_{z=0}^{n^2} z \cdot (3/4)^{\sqrt{z}}) = O(1)$ for each point in the prefix. Each point in the current prefix checks $O(1)$ regions, and so the total work in checking is linear in the size of the prefix. The checks can all be done in parallel in $O(1)$ depth.

Finally, we take the minimum unfinished special iterations (if there are any), and rebuild the hash table for all points up to and including the point corresponding to that iteration.

Applying Theorem 2.3 with $d(n) = O(\log^* n)$ gives the following theorem.

Theorem 5.2. *The randomized incremental algorithm for closest pair in constant d dimensions and for n points can be parallelized to run in $O(n)$ work in expectation and $O(\log n \log^* n)$ depth* whp *on an arbitrary-CRCW PRAM.*

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 [62, 82], and in this section we will study Welzl's randomized incremental algorithm [82]. 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 iterations, and the special iterations are the ones that cause **Update1** to be called, which for iteration i has at most $3/i$ probability of happening. The dependence depth is again $O(\log n)$ *whp* as discussed in Section 2.2.

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** *whp*, where each call takes $O(1)$ depth *whp* as it does in-circle tests and takes a maximum. As in the sequential algorithm, each iteration takes $O(1)$ work in expectation. Applying Theorem 2.3 with $d(n) = O(\log n)$ *whp* (the depth of a executing a iteration 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* whp *on an arbitrary-CRCW PRAM.*

ALGORITHM 6: The iterative LE-lists construction [28]

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

1 Set $\delta(v) \leftarrow +\infty$ and $L(v) \leftarrow \varnothing$ for all $v \in V$
2 **for** $i \leftarrow 1$ to n **do**
3   Let $S = \{u \in V \mid d(v_i, u) < \delta(u)\}$
4   **for** $u \in S$ **do**
5         $\delta(u) \leftarrow d(v_i, u)$
6   append $\langle v_i, d(v_i, u)\rangle$ to the end of $L(u)$
7 **return** $L(\cdot)$

In a similar way to linear programming, i.e., recursing on a lower dimension, the algorithm can be extended to constant d dimension, with $O(\log^d n)$ depth *whp*, and $O(n)$ expected work.

6 ITERATIVE GRAPH ALGORITHMS (TYPE 3)

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 2.3, and also that iterations 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 [28] 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., [29, 38] among many others), and generating probabilistic tree embeddings of a graph [16, 58] which itself is a useful component in a number of network optimization problems and in constructing distance oracles [16, 18]. For $d(u, v)$ being the shortest path from u to v in G, we have:

Definition 6.1 (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 6 provides a sequential iterative (incremental) construction of the LE-lists, where the i'th iteration 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 [17, 60, 77, 80] with more work but less depth). We note that the only minor change to these algorithms is to drop the initialization of the tentative distances before we run SSSP, and instead use the $\delta(\cdot)$ values from previous iterations in Algorithm 6. Thus the search will only explore S and its outgoing edges. Cohen [28] showed that if the vertices are in random order, then each LE-list has size $O(\log n)$ *whp*, and that using Dijkstra with distances initialized with $\delta(\cdot)$, the algorithm runs in $O((m + n\log n)\log n)$ time.

Parallel version. To parallelize the algorithm we use the general approach of Type 3 algorithms as described in Section 2.3 and in particular Algorithm 2. 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 assume the cost functions are concave, 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. We also assume independent shortest path computations can run in parallel (i.e., they do not interfere with each other's state). We assume that the output of each shortest path computation from a source is a set of source-target-distance triples, one for each target that is visited in Line 3.

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 6.2. *Algorithm 6 has a separating dependence for the dependences and orderings $<_v$ defined above.*

Proof. By Definition 2.4, 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 iteration if v_a's iteration is the first among the three.

Clearly the statement holds if v_c's iteration is the earliest among the three. We now consider the case when v_b's iteration 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 iteration if v_a's iteration is first among the three. □

As required by Line 6 of Algorithm 2, we need to combine the results from the iterations in a round—the sets of source-target-distance triples. For LE-lists we need to collect the contributions to each LE-list, remove extra entries, and write the minimum distance to each vertex for the next round. There are extra entries since running an iteration early could find a path not found by the strict sequential order. Collecting the contributions to each LE-list can be done with a semisort on the targets. The elements corresponding to each target can then be sorted based on the iteration number of the source vertex. In the sequential order, distances can only decrease with increasing source iteration index. Therefore, if any of the distances increase with increasing source index, it correspond to an extra entries and the entry is removed. Finally, the remaining elements are appended to the end of the appropriate LE-lists and the minimum distance is written to each vertex. At the end of each round, the state corresponds exactly to the sequential state if all the iterations up to the end of the round had been done one at a time incrementally. This leads to the following theorem:

Theorem 6.3. *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 $O((D_{SP}(n, m) + \log n)\log n)$ depth* whp *on the CRCW PRAM.*

Proof. First we bound the cost of the algorithm excluding the post-processing step. Because of the separating dependences in Algorithm 6 shown in Lemma 6.2, Theorem 2.8 indicates that each vertex is visited no more than $O(\log n)$ times in all iterations *whp*, assuming a random input order of the vertices. Namely, at most $O(\log n)$ searches visit each vertex and its neighbors. Since we assume concavity of the search cost, the overall work for all searches is $O(\log n)$ times $W_{SP}(n, m)$, the cost of the first search that visits all vertices.

The combining after each round requires a semisort on the target vertex, a sort on the source vertex within each target, and a pass to remove extra entries. The semisort can be done in randomized linear work and logarithmic depth [50, 67]. We now show that we can efficiently sort by source. As shown in the proof of Lemma 2.7, the probability that we need to sort l elements for a vertex in one round (i.e. the number of incoming dependences) is bounded by 2^{-l}. Assume that we use a loose upper bound of quadratic work for sorting. The expected work for sorting the elements for each vertex in one round is $\sum 2^{-l} \cdot l^2$ for $l \geq 1$, which solves to $O(1)$. Filtering out is clearly linear in the number of dependences. Thus the total work for combining is $O(m \log n)$ in expectation.

The depth for the combining step of each round is bounded by the cost of the semisort, the sort, and the filter, which can all be done within $O(\log n)$ depth *whp*. When including the depth of each round for the shortest path algorithm ($D_{\text{SP}}(n, m)$) this gives the stated bounds since there are $O(\log n)$ rounds.

□

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 [78] 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 [68], and so a divide-and-conquer SCC algorithm [32] is usually used in parallel settings [6, 55, 76, 79].

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 the vertex is (1) only forward from v, (2) only backward reachable, (3) both forward and backwards reachable, or (4) neither forward no backward reachable. The subset (3) 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. [32] show that if the vertex v is selected uniformly at random, then the algorithm sequentially runs in $O(m \log n)$ work in expectation.[1]

Although divide-and-conquer is generally good for parallelism, the challenge in this algorithm is that the divide-and-conquer tree can be very unbalanced. For example, if the input graph is very sparse such that most of the reachability searches only visit a few vertices, then most of the vertices can fall into the subset of unreachable vertices from v, creating unbalanced partitions with $\Theta(n)$ recursion depth. Schudy [70] describes a technique to better balance the partitions, which can bound the depth of the algorithm to be $O(\log^2 n)$ reachability queries. Unfortunately, his approach requires a factor of $O(\log n)$ extra work compared to the original algorithm. Tomkins et al. [79] describe another parallel approach, although the analysis is quite complicated.[2]

The divide-and-conquer algorithm [32] can also be viewed as an incremental algorithm, as given in Algorithm 7. This is effectively the same relationship as between quicksort and insertion into a BST in random order. More specifically the operations in the incremental algorithms are equivalent to the operations in the divide and conquer algorithm within reordering, where operations are

[1] We assume $m \geq n$.

[2] Tomkins et al. [79] claim that their algorithm takes the same amount of work as the sequential algorithm, but 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.

ALGORITHM 7: 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 induced subgraph of the partition containing 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     $V^- \leftarrow R^- \backslash V_{scc}$
10     $V^+ \leftarrow R^+ \backslash V_{scc}$
11     $V^O \leftarrow S \backslash (R^+ \cup R^-)$
12     $\mathcal{V} \leftarrow (\mathcal{V} \backslash \{S\}) \cup \{V^+, V^-, V^O\}$
13     $S_{scc} \leftarrow S_{scc} \cup \{V_{scc}\}$
14 **return** S_{scc}

comparisons in sorting, or found reachabilities from v_i to v_j for SCC. This assumes the divide-and-conquer algorithms maintain order among elements when partitioning, and picks the first element of a sequence as the pivot. Since the input is in random order, this is equivalent in terms of the probability distribution of possible calls to picking random elements as pivots. We can analyze Algorithm 7 as a Type 3 algorithm, using the general theorem shown in Section 2.3. Our analysis is significantly simpler than those of [70, 79], and the asymptotic 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 [43, 77, 80] or simply with a breadth-first search for low-diameter graphs. We also assume concavity on the work $W_R(n, m)$, which holds for existing reachability algorithms, and that independent reachability computations can run in parallel.

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 7). Let $T = (t_1, t_2, \ldots, t_n)$ be an arbitrary topological order of components in the given graph G, in which vertices of the same component are 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 6.4. *Algorithm 7 has a separating dependence for the dependences and orderings $<_v$ defined above.*

PROOF. By Definition 2.4, 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 iteration if v_a's iteration is the first among the three.

Clearly the statement is true if v_c is earliest. We now consider the case when v_b's iteration is first among the three vertices. We give the argument for the forward direction, and the backward

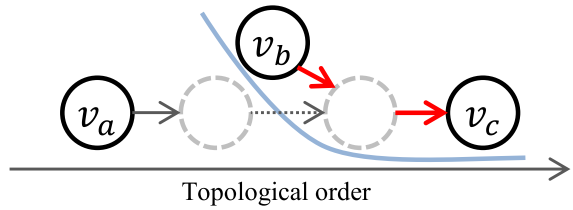

Fig. 2. An illustration for the proof of Lemma 6.4.

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 iteration, either v_a or v_c is marked in one SCC that v_b is in, 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 iteration, and when $v_a <_c v_b <_c v_c$, v_a cannot be forward reachable from v_b's iteration. In the second case, after v_b's iteration, the forward reachability search from v_b reaches v_c but not v_a, and so v_a and v_c fall into different components in $\mathcal{V}$ (shown in Figure 2). As a result, v_c is also not reachable in v_a's iteration.

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

This separating dependence implies that the sequential algorithm on a random ordering visits $O(m \log n)$ edges *whp* since by Corollary 2.9 each vertex v_j, and hence each edge, is visited no more than $O(\log n)$ times *whp*. Since $W_R(n, m) = O(m)$ sequentially, e.g., using BFS, this algorithm does $O(m \log n)$ work *whp*.

We now consider the parallel version. We use the general approach of Type 3 algorithms as described in Section 2.3 and in particular Algorithm 2. The iterations we run in parallel for each round (consisting of increasing power of two) are the same as in Algorithm 7. Within the round we run the iterations independently (as in the assumptions) and then combine the results. The results can be combined as follows. Each iteration identifies the SCC its vertex belongs in (V_{SCC}) and the edges that partition the four subsets (V_{SCC}, V^+, V^-, V^O). We call these edges the cut edges. We take the union of all the SCCs. Note that an SCC will be found by multiple searches when there is more than one vertex from the round in the same SCC. Giving a unique label can be implemented, for example, by initializing the SCC identifier with infinity, and writing the index of each search to the vertices in V_{SCC} using write with minimum (priority write). Now all iterations use their cut edges to cut all the corresponding edges in the graph. Again edges could be cut multiple times.

Observation 6.1. *Running a contiguous sequence of steps of Algorithm 7 independently, and in parallel, and then combining them as described above will find the same SCCs and cut all all the edges that processing the iterations sequentially would. Furthermore, it will only cut edges that do not join two vertices in the same SCC.*

The parallel round will find the same SCCs since it will search from the same vertices each which will find its own SCC. It will cut all the edges cut by the sequential since (a) the boundaries of all the SCCs will be cut, and (b) each independent search will search at least as many reachable vertices as the sequential search (in both directions), never search anything in S^O (since these are not reachable in the graph), and therefore cut at least as many edges from S^- to S^O or from S^O to S^+. Note that the parallel version might cut additional edges, but these will never cut an SCC since if any vertex in an SCC is reached they all will be and therefore edges between them will not appear in the cut.

Theorem 6.5. *For a random order of the input vertices, the incremental SCC algorithm does $O(W_R(n,m)\log n)$ expected work and has $O((D_R(n,m)+\log n)\log n)$ depth* whp *on the CRCW PRAM.*

Proof. Since we have a separating dependence (Lemma 6.4) we can apply Theorem 2.8 to bound the number of times a vertex is visited across all rounds by $O(\log n)$ *whp*. Therefore, all edges are also visited $O(\log n)$ times *whp*. Since we assume the work cost for the reachability queries is a concave function, and the largest search is upper bounded by m vertices and n edges, the total work across all reachability queries is bounded by $O(W_R(n,m)\log n)$ *whp*. Furthermore, since there are $O(\log n)$ rounds each of which is parallel, the total depth across reachability searches is bounded by $O(D_R(n,m)\log n)$ depth *whp*.

Beyond the reachability queries, the rest of the work is in combining the results from the searches. All the combining work is proportional to the number of vertices and edges visited and hence across the $O(\log n)$ rounds is bounded by $O(m\log n)$ *whp*. In particular all that is needed is some priority writes to mark the SCCs, and some concurrent writes to cut the edges. The priority writes can be implemented with in linear work and $O(\log n)$ depth using semisorting. All other operations are easily with $O(\log n)$ depth, for a total of $O(\log n)$ depth per round. This gives the overall bounds. □

Acquiring the same intermediate states as the sequential algorithm. The partitioning of the vertex sets in the previously discussed parallel algorithm is more eager than the sequential algorithm. If desired, the same intermediate states as generated by the sequential algorithm can be retrieved in the parallel version. Let's consider the following case in one parallel round: vertices x and y can reach z is the forward direction, and the priorities are ordered x, y then z. Sequentially, the search from y will reach z iff y is reachable from x, otherwise x's search will disconnect (separate) y and z before y's search. Our goal is to label every vertex with the last search in the round that visited it if run sequentially. We just consider the forward direction, and the backward direction can be done symmetrically.

To generate these labels we run the iterations in parallel as before. For all iterations and for all vertices in V^+ for the iteration we create a source-visited pair and semisort by visited—i.e. for each vertex we collect the indices of all searches that have it in their V^+ set. Now for that vertex we sort by index, and set the current label i to the earliest. We now iterate over the rest in increasing order and for each search index j, if j is reachable from i (easy to look up in V^+ for i, e.g., using hashing) then set i (the current label) to j otherwise leave it. Whenever we do not change the label, this corresponds to a visit that happened in the parallel algorithm that would not happen in the sequential one (i would have separated them). After generating the final label, we cut any edge that has different labels on its two endpoints. Since each vertex is visited $O(\log n)$ times *whp*, the sorting and scanning has low depth. The analysis of this approach would be similar to Theorem 6.3.

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 developed for the sequential versions (e.g., with regard to total work and correctness). We have presented three general types of dependences of algorithms, and tools and general theorems that are useful for multiple algorithms within each type. 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. We thank Laxman Dhulipala for catching a mistake in the SCC algorithm, and Yiqiu Wang and Shangdi Yu for catching a mistake in the closest pair algorithm in the original conference version of this paper. We have fixed these mistakes in this version of the paper.

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] D. Alistarh, T. Brown, J. Kopinsky, and G. Nadiradze. Relaxed schedulers can efficiently parallelize iterative algorithms. In *ACM Symposium on Principles of Distributed Computing (PODC)*, page 377–386, 2018.

[3] D. Alistarh, G. Nadiradze, and N. Koval. Efficiency guarantees for parallel incremental algorithms under relaxed schedulers. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, page 145–154, 2019.

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

[5] N. M. Amato, M. T. Goodrich, and E. A. Ramos. Parallel algorithms for higher-dimensional convex hulls. In *Symposium on Foundations of Computer Science (FOCS)*, pages 683–694, 1994.

[6] 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.

[7] N. Ben-David, G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, C. McGuffey, and J. Shun. Parallel algorithms for asymmetric read-write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2016.

[8] 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.

[9] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Sorting with asymmetric read and write costs. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2015.

[10] G. E. Blelloch, J. T. Fineman, P. B. Gibbons, Y. Gu, and J. Shun. Efficient algorithms with asymmetric read and write costs. In *European Symposium on Algorithms (ESA)*, 2016.

[11] 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.

[12] 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.

[13] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallelism in randomized incremental algorithms. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 467–478, 2016.

[14] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Parallel write-efficient algorithms and data structures for computational geometry. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2018.

[15] G. E. Blelloch, Y. Gu, J. Shun, and Y. Sun. Randomized incremental convex hull is highly parallel. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, 2020.

[16] G. E. Blelloch, Y. Gu, and Y. Sun. Efficient construction of probabilistic tree embeddings. In *International Colloquium on Automata, Languages and Programming (ICALP)*, 2017.

[17] 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.

[18] 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.

[19] 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.

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

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

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

[23] 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.

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

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

[26] 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.

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

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

[29] 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.

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

[31] R. Cole, M. T. Goodrich, and C. Ó. Dúnlaing. A nearly optimal deterministic parallel voronoi diagram algorithm. *Algorithmica*, 16(6):569–617, Dec 1996.

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

[33] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms (3. ed.).* MIT Press, 2009.

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

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

[36] L. Dhulipala, G. E. Blelloch, and J. Shun. Theoretically efficient parallel graph algorithms can be fast and scalable. In *ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 393–404, 2018.

[37] P. Dıaz, D. R. Llanos, and B. Palop. Parallelizing 2D-convex hulls on clusters: Sorting matters. *Proc. XV Jornadas de Paralelismo, Almerıa, Spain*, pages 247–252, 2004.

[38] 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.

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

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

[41] H. Edelsbrunner and N. R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15(3):223–241, 1996.

[42] 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.

[43] J. T. Fineman. Nearly work-efficient parallel algorithm for digraph reachability. In *ACM Symposium on Theory of Computing (STOC)*, pages 457–470, 2018.

[44] 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.

[45] 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.

[46] 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.

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

[48] 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.

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

[50] 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.

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

[52] T. Hagerup. Fast parallel generation of random permutations. In *International Colloquium on Automata, Languages and Programming*, pages 405–416. Springer, 1991.

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

[54] 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.

[55] 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.
[56] J. Jaja. *Introduction to Parallel Algorithms*. Addison-Wesley Professional, 1992.
[57] S. Janson et al. Tail bounds for sums of geometric and exponential variables. *Statistics & Probability Letters*, 135(C):1–6, 2018.
[58] 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.
[59] S. Khuller and Y. Matias. A simple randomized sieve algorithm for the closest-pair problem. *Information and Computation*, 118(1):34–37, 1995.
[60] P. N. Klein and S. Subramanian. A randomized parallel algorithm for single-source shortest paths. *Journal of Algorithms*, 25(2):205–220, 1997.
[61] 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.
[62] N. Megiddo. Linear-time algorithms for linear programming in rˆ3 and related problems. *SIAM journal on computing*, 12(4):759–776, 1983.
[63] K. Mulmuley. *Computational geometry - an introduction through randomized algorithms*. Prentice Hall, 1994.
[64] 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.
[65] 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.
[66] M. O. Rabin. Probabilistic algorithms. *Algorithms and Complexity: New Directions and Recent Results*, pages 21–39, 1976.
[67] S. Rajasekaran and J. H. Reif. Optimal and sublogarithmic time randomized parallel sorting algorithms. *SIAM J. Comput.*, 18(3):594–607, 1989.
[68] J. H. Reif. Depth-first search is inherently sequential. *Information Processing Letters*, 20(5):229–234, 1985.
[69] J. H. Reif and S. Sen. Optimal parallel randomized algorithms for three-dimensional convex hulls and related problems. *SIAM J. Comput.*, 21(3):466–485, 1992.
[70] 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.
[71] R. Seidel. Small-dimensional linear programming and convex hulls made easy. *Discrete & Computational Geometry*, 6(3):423–434, 1991.
[72] R. Seidel. Backwards analysis of randomized geometric algorithms. In *New Trends in Discrete and Computational Geometry*, pages 37–67. 1993.
[73] 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.
[74] 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.
[75] 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.
[76] 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.
[77] T. H. Spencer. Time-work tradeoffs for parallel algorithms. *Journal of the ACM (JACM)*, 44(5):742–778, 1997.
[78] R. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.
[79] 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.
[80] J. D. Ullman and M. Yannakakis. High-probability parallel transitive-closure algorithms. *SIAM Journal on Computing*, 20(1):100–125, 1991.
[81] U. Vishkin. Thinking in parallel: Some basic data-parallel algorithms and techniques, 2010. Class notes from a course on parallel algorithms at UMD.
[82] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, 1991.