Dynamic Constant Time Parallel Graph Algorithms with Sub-Linear Work

Jonas Schmidt
TU Dortmund University, Germany

Thomas Schwentick
TU Dortmund University, Germany

Abstract

The paper proposes dynamic parallel algorithms for connectivity and bipartiteness of undirected graphs that require constant time and $\mathcal{O}(n^{1/2+\epsilon})$ work on the CRCW PRAM model. The work of these algorithms almost matches the work of the $\mathcal{O}(\log n)$ time algorithm for connectivity by Kopelowitz et al. (2018) on the EREW PRAM model and the time of the sequential algorithm for bipartiteness by Eppstein et al. (1997). In particular, we show that the sparsification technique, which has been used in both mentioned papers, can in principle also be used for constant time algorithms in the CRCW PRAM model, despite the logarithmic depth of sparsification trees.

2012 ACM Subject Classification Theory of computation → Dynamic graph algorithms; Theory of computation → Parallel algorithms

Keywords and phrases Dynamic parallel algorithms, Undirected connectivity, Bipartiteness

Digital Object Identifier 10.4230/LIPIcs.MFCS.2023.80

Related Version *Full Version*: `http://arxiv.org/abs/2307.10107`

Acknowledgements We are grateful to Jens Keppeler and Christopher Spinrath for careful proof reading.

1 Introduction

There has been a lot of research on dynamic algorithms for graph problems.[1] Usually, the setting is that graphs can be changed by edge insertions or deletions and that there are query operations that allow to check whether the graph has certain properties. Most of this research has been about sequential algorithms and the goal has been to find algorithms that are as fast as possible. Some algorithms use randomisation, others are deterministic, sometimes the time bounds are worst-case bounds per change or query operation and sometimes they are amortised bounds.

There has been also some research on dynamic *parallel* graph algorithms. Many of these algorithms use the EREW PRAM model[2] and try to achieve logarithmic or polylogarithmic running time, while being work-efficient or even work-optimal. That is, the overall work of all processors should be (almost) the same as for the best sequential algorithm.[3]

There is an entirely separate line of work that studied the maintenance of graph (and other) properties in a setting that was inspired by Database Theory. It is often called *Dynamic Complexity* in Database Theory. In the setting of Dynamic Complexity, dynamic algorithms are called *dynamic programs* and they are not specified in an "algorithmic fashion"

[1] Below we will give pointers to literature. For the beginning of the introduction, we try to keep the story simple.

[2] In an EREW PRAM, parallel processors can use shared memory, but at each moment, each memory cell can be accessed by only one processor. EREW stands for *exclusive-read/exclusive-write*.

[3] We note that in our context of constant-time parallel algorithms work is within a constant factor of the number of processors.

but rather by logical formulas. As a classical example from [13], to maintain reachability information between pairs of nodes in a directed acyclic graph, a dynamic program can use an auxiliary relation T that is intended to store the transitive closure of the graph. The program can then be specified by two formulas that specify how the new version T' of T is defined after the insertion or deletion of an edge (u, v):

Insertion: $T'(x,y) \stackrel{\text{def}}{=} T(x,y) \vee \big(T(x,u) \wedge T(v,y)\big)$. After inserting (u,v) there is a path from x to y if such a path already existed or if there was a path from x to u and from v to y.

Deletion: $T'(x,y) \stackrel{\text{def}}{=} T(x,y) \wedge \Big(E(x,y) \vee \neg T(x,u) \vee \neg T(v,y) \vee$

$\exists u', v' \Big((u' \neq u \vee v' \neq v) \wedge T(x,u') \wedge E(u',v') \wedge T(v',y) \wedge T(u',u) \wedge \neg T(v',u) \Big) \Big)$.

This formula is slightly more complicated. In the main case, the nodes u', v' are chosen such that neither the path from x to u' nor the path from v' to y relies on the edge (u,v). In the former case this is thanks to $T(u',u)$ (since if $T(x,u')$ involved (u,v), the graph were not be acyclic) and in the latter case it is thanks to $\neg T(v',u)$.

As in the example, the underlying logic is usually first-order logic, since it corresponds to the main (theoretical) query language for relational databases, the relational algebra, which in turn corresponds to the core of SQL. The class of problems or queries that can be maintained in this way is usually called DynFO.

Dynamic Complexity has existed quite separated from the world of dynamic algorithms, but there is a direct link that connects the two areas: it follows from a fundamental result[4] from Immerman [8, Theorem 1.1] that dynamic programs can be translated into parallel programs that run in *constant time* on suitable versions of CRCW PRAMs[5] with polynomially many processors. And vice versa.

Dynamic Complexity has focussed on the question whether a graph property can be maintained at all by first-order logic (or fragments thereof), but did not care about the work efficiency of the parallel algorithms that are obtained from translating the update formulas. It turns out that this automatic translation often does not yield very efficient parallel algorithms.

As an example, the parallel dynamic algorithm that is obtained by direct translation of the above formulas, has work $\mathcal{O}(n^4)$ for deletions, since it would consist of two nested loops for x and y and two more for u and v. This is far from being work-efficient.[6] The translation of the dynamic program for Connectivity in undirected graphs from [13] even yields a work bound of $\mathcal{O}(n^5)$. We will show that this work bound can be improved considerably.

This paper is part of an effort to bridge the gap between Dynamic Complexity and (parallel) Dynamic Algorithms by developing algorithms that run in constant time on CRCW PRAMs and are as work efficient as possible. It presents constant-time dynamic parallel algorithms for Connectivity and Bipartiteness in undirected graphs. In the *arbitrary* CRCW model, the algorithms require work at most $n^{\frac{1}{2}}$ polylog(n). In the *common* CRCW model, the algorithms can be instantiated, for each constant $\epsilon > 0$, such that they obey a work bound of $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$, where n is the number of nodes in the graph.

---

[4] Immerman's result is not about dynamic programs, but each formula of a dynamic program can be translated separately.

[5] In a CRCW PRAM more than one processor can read a memory cell, at the same time. Even more than one processor can write into the same cell, but there has to be a strategy that deals with conflicts. This will be explained later in the text.

[6] We have to admit that this paper does not present a better algorithm for directed reachability. We chose that problem only as an example, since its formulas are relatively easy to understand.

The algorithm for Connectivity follows the parallel EREW PRAM algorithm of Kopelowitz et al. [12], which in turn was based on a sequential algorithm by Fredrickson [5] and its sparsification by Eppstein et al. [4]. Thus, the work of our algorithm almost matches the work bound $\mathcal{O}(n^{\frac{1}{2}})$ of the parallel algorithm of [12] and the worst-case runtime of [4]. However, it does not match the runtime of the recent breakthrough algorithm by Chuzhoy et al. [1].

The main technical challenge here is to make the sparsification and the tree-like data structure of [12] work in constant time, despite their use of trees of logarithmic depth. For sparsification, this means updating all logarithmically many nodes along the path from the changed leave to the root of a tree of logarithmic height in parallel constant time although in classical sparsification the change in the leave is propagated from one node to the other along the path. For handling the tree-like data structure of [12] in constant parallel time, data is stored differently by switching from lists to arrays and it is shown (in full version of this paper) that balanced search trees ((a, b)-trees to be precise) of logarithmic height are maintainable in constant parallel time. In the classical algorithm for, e.g., splitting an (a, b)-tree tree into two separate trees, the tree is first split into logarithmically many smaller trees and then the two new trees are built by merging logarithmically many of those smaller trees back together. Both steps are done sequentially in $\mathcal{O}(\log n)$ time by splitting one of those smaller trees at a time and merging only two of the smaller trees at a time, but for our purpose have to be done in constant parallel time.

The algorithm for bipartiteness almost matches the runtime of the bipartiteness algorithm of Eppstein et al. [4]. It is based on the observation that a graph is bipartite if and only if its distance-2 graph has twice as many connected components as the graph itself. The algorithm therefore basically maintains two spanning trees, for the graph and its distance-2 graph. Here, the main technical challenge is to show that the same sparsification approach as for connectivity also works for bipartiteness.

Structure of the paper. We introduce some basic concepts about CRCW PRAMs in Section 2. The algorithm for connectivity is presented in Section 3. The algorithm for bipartiteness is given in Section 4.

Related work. Some related work has already been mentioned above. Dynamic Complexity has started by the work of Patnaik and Immerman [13] and Dong and Su [3]. For a recent survey on the dynamic complexity of Reachability in directed and undirected graphs, we refer to [14]. For a recent survey on dynamic graph algorithms, we refer to [6].

Of course, the PRAM model is not the only parallel computation model for parallel algorithms. Parallel dynamic algorithms for the MPC model can be found, e.g., in [9].

2 Preliminaries

For natural numbers $i \leq j$, we write $[i, j]$ for the set $\{i, \ldots, j\}$. We only deal with undirected graphs and denote an undirected edge between two vertices u and v by (u, v).

Dynamic algorithmic problems. In this paper, we view a dynamic (algorithmic) problem basically as the interface of a data type: that is, there is a collection of operations by which some object can be initialised, changed, and queried. A *dynamic algorithm* is then a collection of algorithms, one for each operation. We consider two main dynamic problems in this paper, Connectivity and Bipartiteness.

The algorithmic problem Connectivity maintains an undirected graph G and has the following operations.

- `Init`(G, n) yields an initial graph G with n nodes, that are initially deactivated but without edges;
- `ActivateNode`(G, v) yields an identifier for a new node of G;
- `DeactivateNode`(G, v) deactivates the node v from G. The node u must be isolated;
- `InsertEdge`(G, u, v) inserts edge (u, v) to G;
- `DeleteEdge`(G, u, v) deletes edge (u, v) from G;
- `Connected`(G, u, v) returns true if u and v are in the same connected component, otherwise false.
- #`Components`(G) yields the number of connected components of G on the activated nodes.

Bipartiteness has almost the same operations, but instead of `Connected` and #`Components` it has a query operation `Bipartite`(G) which yields true if the graph G is bipartite.

Throughout this paper we only consider the effort for change and query operations, but disregard the effort for the initialisation of a graph. We also note that the number n of nodes can not grow. The nodes are represented by numbers in $\{1, \ldots, n\}$.

Parallel Random Access Machines (PRAMs). A *parallel random access machine* (PRAM) consists of a number of processors that work in parallel and use a shared memory.[7] The memory is comprised of memory cells which can be accessed by a processor in $\mathcal{O}(1)$ time. Furthermore, we assume that simple arithmetic and bitwise operations, including addition, can be done in $\mathcal{O}(1)$ time by a processor. The work of a PRAM computation is the sum of the number of all computation steps of all processors made during the computation. We define the space s required by a PRAM computation as the maximal index of any memory cell accessed during the computation.

We use the Concurrent-Read Concurrent-Write model (CRCW PRAM), i.e., processors are allowed to read and write concurrently from and to the same memory location. More precisely, we will consider two different versions of CRCW PRAMs.

- In the *arbitrary* model, if multiple processors concurrently write to the same memory location, one of them, "arbitrarily", succeeds;
- In the slightly weaker *common* model, concurrent write into the same memory location, is only allowed if all processors write the same value.

The two models will yield slightly different work bounds for our dynamic algorithms for Connectivity and Bipartiteness: in the arbitrary model, the work will be at most $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$, whereas in the common model, we will have algorithms with work $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$, for every $\epsilon > 0$. Here, $\tilde{\mathcal{O}}(f(n))$ allows an additional polylogarithmic factor with $f(n)$.

We refer to [10] for more details on PRAMs and to [15, Section 2.2.3] for a discussion of alternative space measures.

For simplicity, we assume that even if the number n of nodes of the input graph grows, a number in the range $[0, n]$ can still be stored in one memory cell. This assumption is justified, since addition of larger numbers N can still be done in constant time and polylogarithmic work on a CRCW PRAM.

The following lemma exhibits a simple CRCW PRAM algorithm in the common model that will be used as a sub-algorithm. It also illustrates the frequent use of arrays in PRAM algorithms. It will mainly be used as a tie-breaker, if one of several objects has to be chosen, and it will therefore not be needed in the context of the arbitrary model. The lemma was shown in a slightly more general form in [11, Proposition 5.4].

[7] Some content of this paragraph is copied from [11].

▶ **Lemma 2.1** ([11, Proposition 5.4]). *Let A be an array of size n over a finite alphabet Σ. The minimum/maximum value of A can be computed in constant parallel time on a common CRCW PRAM with work $\mathcal{O}(n^{1+\epsilon})$ for any $\epsilon > 0$.*

Proof sketch. We only describe how the minimum can be computed, since finding the maximum value is completely analogous. A naïve approach is to assign one processor to each pair i, j of positions in the array. Whenever $A[i] < A[j]$ or $A[i] = A[j]$ and $i < j$, then a 1 is written into $B[j]$, where B is an auxiliary array of size n, in which all entries are initially set to zero. Afterwards, one processor is assigned to each cell of B and the processor assigned to the only cell $B[i]$ with value 0 outputs $A[i]$ as the minimum. However, this algorithm requires $\mathcal{O}(n^2)$ work. The (standard) idea to reduce the work to $\mathcal{O}(n^{1+\epsilon})$ is to first compute the minimum of subarrays of A of size n^ϵ. This requires time $\mathcal{O}(n^{2\epsilon})$, for each of the $n^{1-\epsilon}$ subarrays, resulting in work $\mathcal{O}(n^{1+\epsilon})$. The minimal values of the sub-arrays can then be stored in an array of size $n^{1-\epsilon}$ whose minimum can be computed recursively. Since the number of recursion rounds is bounded by the constant $\lceil \frac{1}{\epsilon} \rceil$, the overall work is $\mathcal{O}(n^{1+\epsilon})$. ◀

3 Connectivity

In this section, we present the main result of this paper and (most of) its proof.

▶ **Theorem 3.1.** *There are dynamic parallel constant-time algorithms for* Connectivity *with the following work bounds per change or query operation.*

- $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ *work on the* arbitrary *CRCW PRAM model.*
- $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ *work on the* common *CRCW PRAM model, for every $\epsilon > 0$.*

As usual, the algorithm basically maintains a spanning forest and the graph G is connected if and only if #`Components`(G) yields 1.

In fact, we will consider the data type SpanningForest as an extension of Connectivity with the following additional operation.

- `TreeEdge`(G, u, v) returns true if (u, v) is a tree edge, otherwise false.

The proof is along the lines of [12] and is split into the same three main steps. For each step, we need to show that it can be done in constant parallel time on a CRCW PRAM, as opposed to $\mathcal{O}(\log n)$ on an EREW PRAM. This strengthening comes with an additional work factor of polylog(m) or polylog(n) on an *arbitrary* CRCW PRAM and m^ϵ or n^ϵ on a *common* CRCW PRAM.

We first show that, for graphs of maximum degree 3, SpanningForest can be maintained with work $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ and $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ per operation, depending on the PRAM model. Then we show that the case of graphs of unbounded degree can be reduced to the case of graphs with degree bound three. Finally, we show that, with the help of sparsification, both bounds from above are translatable to be in n instead of m.

More precisely, we show the following three results.

▶ **Proposition 3.2.** *There are dynamic parallel constant time algorithms for the special case of* SpanningForest*, where the maximum degree of the graph never exceeds 3 with the following work bounds per change or query operation.*

- $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ *on the* arbitrary *CRCW PRAM model.*
- $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ *on the* common *CRCW PRAM model, for every $\epsilon > 0$.*

▶ **Proposition 3.3.** *If* SpanningForest *can be maintained in parallel constant time on a CRCW PRAM with the work bounds of Proposition 3.2 per change or query operation, for any $\epsilon > 0$, for graphs with maximum degree 3, it can be maintained with the same bounds for general graphs with the provision that they never have more than cn edges, for some constant c.*

▶ **Proposition 3.4.** *If* SpanningForest *can be maintained in parallel constant time on a CRCW PRAM with the work bounds of Proposition 3.2 per change or query operation, then it can also be maintained with $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ work per change or query operation on the common CRCW model and with $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ work per change or query operation on the arbitrary CRCW model.*

Proposition 3.2 will be shown in the next two subsections. Proposition 3.3 and Proposition 3.4 will be shown in Subsection 3.2.

3.1 Maintaining a spanning forest for bounded degree graphs

As mentioned before, our algorithm closely follows [12] and therefore uses a similar data structure. Some modifications are required though, to achieve constant parallel update and query time while keeping almost the same amount of work. The data structure maintains an Euler tour, for each spanning tree in a spanning forest of the graph. More precisely, it maintains, for each spanning tree, a cyclic list of tree edges that visits each tree edge once in either direction.

We first concentrate on the change operations `InsertEdge`(G, u, v) and `DeleteEdge`(G, u, v) and the query operations.

The algorithm does not need to change the Euler tour, if a new edge is inserted which connects two nodes of the same spanning tree or if a non-tree edge is deleted. If an edge e between two different spanning trees is inserted, the algorithm can just merge the two Euler tours. If an edge e of a spanning tree is deleted, the algorithm first splits the Euler tour at both occurrences of e and then tries to find a *replacement edge* that connects the two sub-trees resulting from the deletion. The search for a replacement edge is actually the most critical part of the algorithm, since trying out all edges of the graph would yield linear work.

Towards a more efficient algorithm, we follow the same two-tiered approach as [12]: each Euler tour is chopped into chunks of about $\sqrt{m}$ edges, which are represented as *arrays of edges*. The underlying idea is that after a change operation the necessary updates can be divided into low-level manipulations inside only a few chunks and high-level manipulations on the level of sequences of chunks. Each kind of manipulation should cause not much more than $\mathcal{O}(\sqrt{m})$ work.

Furthermore, it will maintain information about non-tree edges between different chunks, ultimately allowing to find a replacement edge with work close to $\mathcal{O}(\sqrt{m})$.

We fix a number K that will be roughly $\sqrt{m}$ later on and enforce that chunks contain between $\frac{K}{2}$ and K edges, with the exception of at most one chunk per spanning tree. We denote the number of chunks by J which is in $\mathcal{O}(\frac{m}{K})$.

For the lower tier, i.e., creating and removing chunks and changing their content and additional information, the edge arrays representing the chunks are stored together in one *master array* $\mathcal{M}$ with $\mathcal{O}(\sqrt{m})$ slots of sub-arrays of length K. The slots will contain some additional information to be specified later. The order of chunks in $\mathcal{M}$ can be arbitrary and $\mathcal{M}$ might contain empty slots from deleted chunks. By $\mathcal{M}(i)$ we refer to the chunk that is stored in the i-th slot of the master array. Some entries in $\mathcal{M}$ might be unused or deactivated. For a chunk C, we refer by C also to the entry in $\mathcal{M}$ for this chunk.

We say that two chunks C and C' are *linked*, if there is a non-tree edge (u, v) in G such that u occurs in C and v in C'. With each chunk C of edges, we associate a *link vector* B_C, which is a bit array of length J that reflects which chunks are linked with C. More precisely, $B_C(i) = 1$ if C and $\mathcal{M}(i)$ are linked. Here all slots in $\mathcal{M}$ are relevant, even the unused or deactivated ones (but they will inevitably yield the bit 0).

The higher tier, which is responsible for maintaining the order of the chunks and information about sequences of chunks, maintains, for each Euler tour a *chunk array*: this is an array of pointers to chunks such that the concatenation of all edge lists in the order induced by the array represents the Euler tour. Furthermore, it maintains information about links between sequences of chunks in a sufficiently work-efficient way.

The algorithm uses a data type ChunkArrays whose operations can be split into two groups. The first group consists of the following operations, which only access chunks and their link arrays, but do not directly refer to chunk arrays. In both groups of operations, i, j and k are always indices, C, C_1 and C_2 are chunk (pointers), A, A_1 and A_2 are (pointers to) chunk arrays, B is a bit vector and E is an edge array.

- `SetChunk`(i, C, E) activates a new chunk C in $\mathcal{M}(i)$ and stores the edge array E in C;
- `Deactivate`(C) deactivates chunk C in $\mathcal{M}$;
- `Link`(C_1, C_2) and `Unlink`(C_1, C_2) allow to mark chunks C_1 and C_2 as linked or unlinked;
- `BulkSetLinks`(C, B) replaces the link vector of chunk C by the bit vector B and changes the bit that refers to C in all other chunks C' according to B. More precisely, if $C = \mathcal{M}(i)$ then, for each j, the i-th bit in the link vector of $\mathcal{M}(j)$ is set to $B(j)$.

We note that `SetChunk` and `Deactivate` do not automatically change any link vectors.

The operations of the other group are as follows. They explicitly refer to chunk arrays.

- `InsertChunk`(A, i, C) inserts (a pointer to) chunk C at position i of chunk array A, moving each entry, from i on, by one to the right;
- `DeleteChunk`(A, i) deletes the chunk pointer at position i of chunk array A, moving each entry, from $i + 1$ on, one to the left;
- `Concatenate`(A_1, A_2) concatenates the chunk array A_2 to the end of A_1;
- `Split`(A, i) splits A into two arrays A_1 and A_2 getting intervals $[1, i]$ and $[i + 1, max(A)]$ and yields (pointers to) A_1 and A_2;
- `Reorder`(A, i, j, k) moves the chunks of positions $j, \ldots, k$ to position $i < j$. That is, the chunks are ordered as $1, \ldots, i - 1, j, \ldots, k, i + 1, \ldots, j - 1, k + 1, \ldots, m$, where m is the size of A;
- `Query`(A, i, j, k, ℓ) yields an arbitrary pair (C, C') of linked chunks where C is from $[i, j]$ and C' is from $[k, \ell]$.

The algorithm will maintain the invariant that each chunk that is present in $\mathcal{M}$ occurs in at most one chunk array. Each chunk in the master array contains a back pointer to its chunk pointer in its chunk array, and these entries are maintained by the above operations.

The following lemma is shown in the full version of this paper.

▶ **Lemma 3.5.** *There is a dynamic parallel constant-time algorithm for* ChunkArrays *on an* arbitrary *CRCW PRAM that supports all operations with $\mathcal{O}(J \operatorname{polylog} J)$ work.*

Furthermore, for each $\epsilon > 0$, there is a parallel constant-time dynamic algorithm for ChunkArrays *on a* common *CRCW PRAM that supports* `Query` *with $\mathcal{O}(J^{1+\epsilon})$ work and all other operations with $\mathcal{O}(J \operatorname{polylog} J)$ work.*

The implementation of ChunkArrays uses (a, b)-trees [7], which are trees of logarithmic height in which inner nodes have between a and b children, and support insertion and deletion of leaves as well as split and join of trees. In the ChunkArrays for each chunk array A one $(2, 6)$-tree is maintained, that has the link arrays of the chunks of A at its leaves, in the order of A. The inner vertices of the tree store link arrays that are the bitwise disjunction of the link arrays of the leaves below them.

Given Lemma 3.5, we can now show Proposition 3.2, stating that SpanningForest can be maintained in parallel constant time with $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ work per operation on an *arbitrary* CRCW PRAM and, for every $\epsilon > 0$, with work $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ work per operation, on a *common* CRCW PRAM, if the maximum degree of the graph never exceeds 3. The degree bound mainly helps by bounding the number of edges incident to one chunk by $\mathcal{O}(K)$.

Proof (of Proposition 3.2). We first describe the data structure and then how it is maintained for the different change operations.

The algorithm maintains a master array and a chunk array as described above. It uses them to maintain a spanning tree and a corresponding Euler tour for each connected component of the graph. It uses $K = J = \sqrt{m}$.

Furthermore, the algorithm maintains an array with all nodes of the graph, and three additional entries for the up to three neighbours of each node, representing the edges. Additionally, there are pointers to the at most six appearances of a node in edges of Euler tours in the master array. The algorithm thereby implicitly maintains pointers from each edge to its at most two appearances in the Euler tours. Finally, a counter for the number of connected components is maintained.

The two query operations `Connected` and `#Components` can be answered in constant sequential time using the pointers from each node to occurences in an Euler tour or the maintained counter, respectively.

For the change operation `InsertEdge`(u, v), we consider two different cases: (1) the insertion of a new edge (u, v) where u and v are in the same connected component and (2) the insertion of a new edge (u, v) where u and v are in different connected components.

The algorithm first identifies through the master array two chunk arrays A_u and A_v in which u and v reside. If $A_u = A_v$, we are in case (1) and it suffices to mark C_u and C_v as linked by `Link`(C_u, C_v) for all C_u and C_v that contain u and v respectively. All those chunks can be found using the maintained pointers from nodes to their appearances in chunks.

If A_u and A_v are different, we are in case (2) and (u, v) newly connects the two spanning trees T_u and T_v, yielding a new spanning tree T. To this end, the two Euler tours represented by A_u and A_v need to be joined. Let the tour of A_u consist of two paths P_1, P_2, where P_1 ends in u and P_2 starts in u. Note that the last node of P_2 is the same as the first node of P_1. Let the two paths Q_1, Q_2 be defined analogously for A_v and v. Then the combined Euler tour will be $P_1, (u, v), Q_2, Q_1, (v, u), P_2$.

The algorithm first joins the two chunk arrays by `Concatenate`(A_u, A_v). It splits the edge sequence of C_u into a sequence E_u^1 that ends in u and the remaining sequence E_u^2 that starts in u. The position where C_u needs to be split can be found using the maintained edge pointers. E_u^1 remains in C_u and for E_u^2 a new chunk C'_u is reserved in $\mathcal{M}$ and inserted in A_u, next to C_u. Similarly, the content of C_v is split into C_v and a new chunk C'_v. Then (u, v) is added to C_u and (v, u) to C_v.

The algorithm then restructures A_u by copying the sub-arrays corresponding to Q_2 and $(Q_1, (v, u))$ to their correct places by two calls to `Reorder`. This also moves P_2 to its right place. If any of the four modified chunks has fewer than $\frac{K}{2}$ edges, it is combined with a neighbour chunk in A_u: if possible, the two chunks are joined or otherwise each gets at least $\frac{K}{2}$ edges to fulfil the invariant. This completes the restructuring of A_u.

It remains to update the link information between chunks. To this end, the algorithm first computes the link vectors for C_u, C'_u, C_v and C'_v. This can be done by initializing the vector with $\vec{0}$ and then scanning all at most $3K$ edges of the respective chunk. The four resulting link vectors are then set by `BulkSetLinks`. The operation `BulkSetLinks` takes also care of the modifications in the link vectors of all other chunks. Finally, the connected component counter is decreased by 1.

For the change operation `DeleteEdge`(u, v), we consider again two different cases: (1) the deletion of an edge (u, v) that is not in any spanning tree and (2) the deletion of a spanning tree edge (u, v). If there is no pointer from (u, v) to an occurence in any Euler tour, we are in case (1) and (u, v) was not a spanning tree edge. The algorithm then checks whether the chunk pairs linked by (u, v) are still linked without (u, v) by scanning all, thanks to the degree bound at most $3K$ many edges of one of the two chunks per pair. If not, it marks the two chunks as unlinked with `Unlink`.

If there are pointers from (u, v) to occurences in an Euler tour, we are in case (2) and (u, v) was a spanning tree edge. Let $P_1, (u, v), P_2, (v, u), P_3$ be the decomposition of the Euler tour of its tree T. By two applications of `Query` the algorithm checks whether there are any links between P_1 and P_2 or P_3 and P_2, respectively.

If there are no such edges then it first reorders the chunk array, so that P_1 and P_3 are consecutive, moving P_2 towards the end, and then splits it into the two parts P_1, P_3 and P_2.

If there is such an edge, let us assume there are chunks C_1 in P_1 and C_2 in P_2 that are linked. The algorithm inspects all edges in C_1 in parallel and tests, whether their partner edge is in C_2. It then either picks an arbitrary edge (if the PRAM model supports that) or computes the minimum edge with this property. Let the chosen replacement edge be (w_1, w_2). Decomposing P_1 into P_1', P_1'', separated at w_1, and P_2 into P_2', P_2'', separated at w_2, the new Euler cycle is $P_1', (w_1, w_2), P_2'', P_2', (w_2, w_1), P_1'', P_3$. It can be constructed in T by splitting C_1 and C_2 into two chunks, with the help of two newly inserted chunks, reordering the array, and repairing small chunks, similarly to the above case of inserting a new tree edge.

The work of the algorithm is dominated by finding a replacement edge. It requires two initial calls to `Query` of the ChunkArrays data type requiring $\mathcal{O}(J \operatorname{polylog} J)$ or $\mathcal{O}(J^{1+\epsilon})$ work, depending on the PRAM model. Then it requires work $\mathcal{O}(K)$ to identify all at most $3K$ possible replacement edges. In the arbitrary model the choice of the actual edge is immediate. In the common model, it might take work $\mathcal{O}(K^{1+\epsilon})$. Apart from that, the algorithm applies a constant number of calls to operations of ChunkArrays that all require $\mathcal{O}(J \operatorname{polylog} J)$ work. By Lemma 3.5 and the choice of J and K we get the desired work bounds.

The operations `ActivateNode` and `DeactivateNode` can be easily implemented in constant sequential time. ◀

3.2 From bounded to unbounded degree and from m to n

We first show Proposition 3.3 which alllows us to conclude that Proposition 3.2 can be lifted to graphs without a degree restriction.

Proof sketch (of Proposition 3.3). Like for [12] our algorithm uses the well known graph reduction already used by [5]. To maintain connectivity for an unrestricted graph G, the algorithm maintains a graph G' of degree at most 3, which is connected if and only if G is connected. The number of nodes of this graph is initialised as cn, where c is as in the statement of the proposition. The idea of the reduction is to replace each node v of G of degree $d > 3$ in G' by a cycle of length d and to connect each node of the cycle to one node adjacent to v.

More formally, G' has two nodes, denoted as $n(u, v)$ and $n(v, u)$, for each (undirected) edge (u, v) of G and one node, denoted v, for each isolated node v of G.

For each non-isolated node u of G, the nodes of the form $n(u, v)$ are connected in some cyclic order. To this end, the algorithm maintains a doubly linked list of nodes of G', for each node u of G.

An insertion of a new edge (u, v) into G translates into activating two new nodes $n(u, v)$ and $n(v, u)$ in G', to connect them with each other and to insert them at an arbitrary position into the cycle of u and v respectively. Together, this yields 2 node additions, 2 edge deletions and 5 edge insertions.

A deletion of an edge (u, v) boils down to the reverse operations: 5 edge deletions, 2 edge insertions and 2 node deactivations.

And obviously, a connectivity query towards G can just be translated into a connectivity query towards G'.

Altogether, each operation for G can be translated into a constant number of operations for G'. The number of nodes of G' is linear in the number of edges plus the number of (isolated) nodes of G. Therefore, the work bounds $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ and $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ for G' translates into a work bound $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ and $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ for G. ◀

The final step towards Theorem 3.1 is to show that the work bounds $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ and $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ for maintaining SpanningForest can be replaced by $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ and $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$, thus showing Proposition 3.4. We use the sparsification technique of [4] which has also been used in [12] to maintain Connectivity in a parallel setting.

In a nutshell, the approach is to maintain a so-called *sparsification tree* $\mathcal{S}$, that is a rooted tree of logarithmic depth in n, each node u of which represents a certain subgraph G_u of G and carries additional structure. The root represents the whole graph, each leaf represents a subgraph consisting of (at most) one edge, and the graph of each inner node is basically the union of the edge sets of the graphs of its children. The crucial idea is that $\mathcal{S}$ has an additional *base graph* B_u, for each tree node, which has a subset of the edges of G_u of linear size, and has the same connected components (viewed as sets of nodes) as G_u. Furthermore, the algorithm maintains a spanning forest F_u of B_u (and thus for G_u), for each tree node u, using the algorithm with work bound $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ or $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ depending on the PRAM model. It has the invariant that, for each inner node u, B_u consists of all edges of the spanning forests F_v, for all children v of u. As this number will be a constant (in fact: 4), the invariant guarantees the linear number of edges of B_u.

We will see that each change operation can be basically handled by triggering change operations along one path of the tree. Since the base graph of a node at level i has at most $\frac{cn}{2^i}$ many edges, for some constant c, and, the overall work can be bounded by $\tilde{\mathcal{O}}(n^{\frac{1}{2}} \log n) = \tilde{\mathcal{O}}(n^{\frac{1}{2}})$ and $\mathcal{O}(n^{\frac{1}{2}+\epsilon'} \log n) = \mathcal{O}(n^{\frac{1}{2}+\epsilon})$, if ϵ' is chosen appropriately.[8]

We next describe the underlying tree structure of $\mathcal{S}$ in more detail. It relies[9] on a *node partition tree* $\mathcal{N}(G)$: it is a binary tree, in which each node is a set U of nodes of G. The root is the set V of all nodes and, for each inner node U with children U_1, U_2, U is the disjoint union of U_1 and U_2 and the sizes of U_1 and U_2 differ by at most 1. The leaves are the singleton sets. Clearly this tree has depth at most $\log(n) + 1$. We emphasise that the partitions are independent of the edge set of G, they do not need to partition the graph into meaningful clusters.

The edge set of G and $\mathcal{N}(G)$ determine the structure of the sparsification tree $\mathcal{S}$ and its graphs G_u as follows. For each level i of $\mathcal{N}(G)$, $\mathcal{S}$ has one node G_u, for each pair (V_1, V_2) of nodes of $\mathcal{N}(G)$ of level i. Here, $V_1 = V_2$ is allowed. The node set of G_u is $V_1 \cup V_2$ and

[8] In fact, using the sizes of the base graphs along a path and the "well-behavedness" of $n^{\frac{1}{2}+\epsilon}$, one actually gets a $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ bound, directly.

[9] We remark that for our algorithm it is actually not important how the (potential) edges of the graph G are exactly partitioned in the sparsification tree, as long as it has constant branching, logarithmic depth and the correspondence between edge sets of nodes and of their children. The definition with the help of the node partition tree is just one concrete way of doing it.

the edges are the edges of G that connect a node from V_1 with a node from V_2. If $V_1 = V_2$, then G_u is thus just the subgraph of G induced by V_1. If U_1 is the parent of V_1 and U_2 the parent of V_2 in $\mathcal{N}(G)$, then the node corresponding to (U_1, U_2) is the parent of (V_1, V_2) in $\mathcal{S}$. For a leaf u, G_u either has one edge or has no edges, if the edge for the pair (x, y) of nodes corresponding to u is not present in G.

The base graphs B_u and the spanning forests F_u can be chosen in any way that is consistent with the invariant that, for each inner node u, B_u consists of all edges of the spanning forests F_v, for all children v of u.

Although our presentation slightly differs from [4, 12], the node partition tree and the sparsification tree are basically the same as there.

Before we present the proof of Proposition 3.4, we state some helpful observations about $\mathcal{S}$.

(1) Each edge (x, y) of G occurs exactly in the graphs G_u along the paths from the leaf with (x, y) to the root.

(2) If two nodes x, y are in the same connected component in G_u, this also holds in all G_v, where v is an ancestor of u.

(3) If an edge (x, y) occurs in some spanning forest F_u, then it occurs in all spanning forests F_v on the path from u to the leaf containing (x, y).

Proof sketch (of Proposition 3.4). The algorithm maintains a sparsification tree $\mathcal{S}$ for the graph G. For each node u of $\mathcal{S}$ it maintains a spanning forest F_u for B_u (and implicitly, for G_u) with the help of the algorithm for SpanningForest from Proposition 3.3, with $c = 4$.

If an edge (x, y) is inserted to G, the algorithm checks, for each node u on the path π from the leaf for (x, y) to the root, whether x and y are in the same connected component of T_u. From Observation (2) it follows that the nodes u, for which this is *not* the case constitute some initial segment of π. For all these nodes u, (x, y) is added to B_u and T_u. Furthermore, it is added to B_v of the parent v of the last node of π.

The deletion of an edge (x, y) is slightly more complicated. For all nodes u on the path from the leaf v with (x, y) to the root, the algorithm tests in parallel, whether (x, y) occurs in F_u. Thanks to Observation (3), the nodes v, for which this test is positive form an initial segment π' of π up to some node w. For each of these nodes, the algorithm computes a replacement edge for (x, y), if such exists. Thanks to Observation (2), a replacement edge that works for some F_v is also a replacement edge for all nodes on π' above v. In particular, all edges v, for which F_v has a replacement edge form an upper segment of π' and the replacement edge for the lowest F_v can be used for all of them. Therefore, after doing the initial test and computing a replacement edge for each forest, constant time and work $\mathcal{O}((\log n)^2)$ suffice to determine the lowest node w and its replacement edge e. Since $(\log n)^2 = \mathcal{O}(\text{polylog}\, n)$ and $(\log n)^2 = \mathcal{O}(n^\epsilon)$, for each $\epsilon > 0$, this work can be neglected. Afterwards, for each node u of π' above w, (x, y) is deleted from B_u and F_u and instead e is added. In the base graph of the parent of w, edge (x, y) is deleted and e added.

As already explained above, the algorithm applies at most a logarithmic (in n) number of times an operation of the algorithm underlying Proposition 3.3, for a base graph, i.e., a graph with $\mathcal{O}(n)$ edges. The desired work bound $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ for arbitrary CRCW PRAMs is thus immediate and by choosing in Proposition 3.3, any $\epsilon' < \epsilon$ instead of the given ϵ, we can establish the desired work bound $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ for common CRCW PRAMs. ◀

4 Bipartiteness

In this section, we show that the work bound established for Connectivity in Section 3 also holds for Bipartiteness. In fact, the algorithm will rely on the algorithm of Proposition 3.3.

▶ **Theorem 4.1.** *There are dynamic parallel constant-time algorithms for* Bipartiteness *with the following work bounds per change or query operation.*

- $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ *work on the* arbitrary *CRCW PRAM model.*
- $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ *work on the* common *CRCW PRAM model, for every* $\epsilon > 0$.

The result follows from an analogous series of statements, as for Connectivity (or SpanningForest, for that matter).

▶ **Proposition 4.2.** *There are dynamic parallel constant time algorithms for the special case of* Bipartiteness*, where the maximum degree of the graph never exceeds 3 with the following work bounds per change or query operation.*

- $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ *on the* arbitrary *CRCW PRAM model.*
- $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ *on the* common *CRCW PRAM model, for every* $\epsilon > 0$.

▶ **Proposition 4.3.** *If* Bipartiteness *can be maintained in parallel constant time on a CRCW PRAM with the work bounds of Proposition 4.2 per change or query operation, for any* $\epsilon > 0$*, for graphs with maximum degree 3, it can be maintained with the same bounds for general graphs with the provision that they never have more than* cn *edges, for some constant* c*.*

▶ **Proposition 4.4.** *If* Bipartiteness *can be maintained in parallel constant time on a CRCW PRAM with the work bounds of Proposition 4.2 per change or query operation, then it can also be maintained with* $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ *work per change or query operation on the common CRCW model and with* $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ *work per change or query operation on the arbitrary CRCW model.*

For an undirected graph $G = (V, E)$, we write $G^{(2)}$ for the graph[10] $(V, E^{(2)})$, where a pair (u, v) of nodes is in $E^{(2)}$, if they are connected by a path of length exactly 2 in G.

Bipartiteness of a graph G can be characterised in the following way by the numbers of connected components of G and $G^{(2)}$.

▶ **Lemma 4.5.** *An undirected graph* G *is bipartite, if and only if the number of connected components of* $G^{(2)}$ *is twice the number of connected components of* G*.*

Proof. It suffices to show that a connected graph G is bipartite if and only if $G^{(2)}$ has 2 connected components.

Let us assume first that G is bipartite and let the nodes of G be coloured with black or yellow such that no two nodes of the same color are connected by an edge. Clearly, each pair of nodes of the same color is connected by a path of even length in G and is therefore in the same connected component in $G^{(2)}$.

Towards a contradiction, let us assume that $G^{(2)}$ is connected. Then there must be a yellow node u and a black node v that are connected by an edge in $G^{(2)}$. Therefore, there must be a node w, such that (u, w) and (w, v) are edges in G. But w can neither be black nor yellow, the desired contradiction.

Let us now assume that G is not bipartite and let C be a cycle of G of odd length. Then all pairs of nodes of C are connected by paths of even length and therefore all nodes of C are in the same connected component of $G^{(2)}$. But clearly, each other node of G is connected by a path of even length to *some* node of C and thus $G^{(2)}$ is connected. ◀

[10] $G^{(2)}$ should not be confused with the square G^2 of G, where edges are induced by paths of length *at most* 2.

With the help of Lemma 4.5, it is now easy to find an algorithm for Bipartiteness for graphs of degree at most 3.

Proof (of Proposition 4.2). To maintain bipartiteness for a graph G of maximum degree 3, the algorithm maintains two instances of SpanningForest, one for G, and one for $G^{(2)}$. It answers that G is bipartite, whenever the number of connected components of $G^{(2)}$ is twice the number of connected components of G.

An edge insertion in G results in at most 6 edge insertions in $G^{(2)}$, and likewise for edge deletions. Furthermore, the number of edges of $G^{(2)}$ is at most $3m$, if m is the number of edges of G. Therefore, Bipartiteness can be maintained in parallel constant time with work $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ or rather $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$, thanks to Proposition 3.3. ◀

Next we lift the bound to graphs of unbounded degree with the help of a bipartiteness preserving reduction.

Proof (of Proposition 4.3). To maintain bipartiteness of graph G, the algorithm again maintains bipartiteness for a graph G' of maximal degree 3, such that G is bipartite if and only if G' is bipartite. The graph G' results from G by applying the following replacement step, consecutively to all (original) nodes of G.

A node u of degree $d > 1$ is replaced by a cycle $u_1, u'_1, u_2, \cdots, u'_d, u_1$ with $2d$ nodes. Each node u_i is connected to a neighbour of u by an edge. It is easy to see that any path that connects two neighbours of u and uses intermediate nodes of the new cycle has even length. The construction therefore preserves bipartiteness. Furthermore, each node in G' has degree at most 3 and the number of edges of G' is at most 6 times the number of edges of G. Finally, each edge insertion or deletion in G triggers at most 5 edge insertions or deletions in G'. ◀

The final step from work $\tilde{\mathcal{O}}(m^{\frac{1}{2}})$ to $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ and work $\mathcal{O}(m^{\frac{1}{2}+\epsilon})$ to $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ again uses sparsification. In fact, it uses the same kind of sparsification tree as the proof of Proposition 3.4. The crucial observation is that if a graph G is not bipartite, it has a base graph in its sparsification tree that is not bipartite.

▶ **Lemma 4.6.** *Let G be an undirected graph and $\mathcal{S}$ a sparsification tree for G. Then G is bipartite if and only if all base graphs in $\mathcal{S}$ are bipartite.*

Proof. Since each base graph of $\mathcal{S}$ is a subgraph of G, the "only if" implication is trivial.

To show the "if" implication, let G be a non-bipartite graph. Since the graph G_r for the root r of $\mathcal{S}$ is non-bipartite, but all graphs G_v for leaves v of $\mathcal{S}$ are bipartite, there must be a node u, such that G_u is non-bipartite, but all graphs G_w, for children w of u, are bipartite. We claim that the base graph B_u is non-bipartite.

Indeed, let C be some cycle of odd length in G_u. By construction of $\mathcal{S}$, each edge (x, y) of C occurs in some graph G_w, where w is a child of u. Therefore, x and y are in the same connected component of G_w and there must be a path between x and y in the spanning forest F_w. Since G_w is bipartite and there is an edge between x and y, the length of this path must be odd. By definition, all edges of this path are in B_u. Since this holds for every edge of C, there exists a closed path in B_u, consisting of an odd number of paths of odd length. This implies that B_u has a cycle of odd length and is therefore not bipartite. ◀

Now we are prepared to give the proof of Proposition 4.4 and thus complete the proof of Theorem 4.1.

Proof (of Proposition 4.4). Just like for Proposition 3.4, the algorithm maintains a sparsification tree $\mathcal{S}$ for the graph G. For each node u of $\mathcal{S}$ it maintains whether B_u is bipartite with the algorithm resulting from Proposition 4.2 and Proposition 4.3. This is possible with work bounds $\tilde{\mathcal{O}}(n^{\frac{1}{2}})$ and $\mathcal{O}(n^{\frac{1}{2}+\epsilon})$ per change operation, just as for Proposition 3.4.

On top of that, the algorithm maintains, for each node u of $\mathcal{S}$, a flag, signalling whether all base graphs in the tree induced by u are bipartite. These flags can be maintained in a straightforward fashion with work $\mathcal{O}(\log n)$. The bipartiteness status of G can then be inferred from the flag of the root of $\mathcal{S}$, thanks to Lemma 4.6. ◀

5 Conclusion

This paper was motivated by the goal to find graph problems whose sublinear sequential dynamic complexity carries over to sublinear work of a dynamic parallel constant time algorithm. In future work it has to be seen whether the faster algorithm from [1] can be translated equally well. Another challenge is to find a dynamic parallel constant time algorithm for the reachability problem in directed graphs. The upper work bound of the algorithm stemming from [2] is roughly $\mathcal{O}(n^{12})$. Another interesting question is whether the algorithm for Bipartiteness can be adapted so that it also yields a 2-colouring of the graph.

References

1 Julia Chuzhoy, Yu Gao, Jason Li, Danupon Nanongkai, Richard Peng, and Thatchaphol Saranurak. A deterministic algorithm for balanced cut with applications to dynamic connectivity, flows, and beyond. In Sandy Irani, editor, *61st IEEE Annual Symposium on Foundations of Computer Science (FOCS)*, pages 1158–1167, 2020. doi:10.1109/FOCS46700.2020.00111.

2 Samir Datta, Raghav Kulkarni, Anish Mukherjee, Thomas Schwentick, and Thomas Zeume. Reachability is in DynFO. *J. ACM*, 65(5):33:1–33:24, 2018. doi:10.1145/3212685.

3 Guozhu Dong and Jianwen Su. First-order incremental evaluation of datalog queries. In *Database Programming Languages (DBPL-4), Proceedings of the Fourth International Workshop on Database Programming Languages - Object Models and Languages*, pages 295–308, 1993.

4 David Eppstein, Zvi Galil, Giuseppe F. Italiano, and Amnon Nissenzweig. Sparsification - a technique for speeding up dynamic graph algorithms. *J. ACM*, 44(5):669–696, 1997.

5 Greg N. Frederickson. Data structures for on-line updating of minimum spanning trees, with applications. *SIAM J. Comput.*, 14(4):781–798, 1985. doi:10.1137/0214055.

6 Kathrin Hanauer, Monika Henzinger, and Christian Schulz. Recent advances in fully dynamic graph algorithms (invited talk). In James Aspnes and Othon Michail, editors, *1st Symposium on Algorithmic Foundations of Dynamic Networks (SAND)*, pages 1:1–1:47, 2022. doi:10.4230/LIPIcs.SAND.2022.1.

7 Scott Huddleston and Kurt Mehlhorn. A new data structure for representing sorted lists. *Acta Informatica*, 17:157–184, 1982. doi:10.1007/BF00288968.

8 N. Immerman. Expressibility and parallel complexity. *SIAM J. Comput.*, 18:625–638, 1989.

9 Giuseppe F. Italiano, Silvio Lattanzi, Vahab S. Mirrokni, and Nikos Parotsidis. Dynamic algorithms for the massively parallel computation model. In Christian Scheideler and Petra Berenbrink, editors, *The 31st ACM on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 49–58, 2019. doi:10.1145/3323165.3323202.

10 Joseph F. JáJá. *An Introduction to Parallel Algorithms*. Addison-Wesley, 1992.

11 Jens Keppeler, Thomas Schwentick, and Christopher Spinrath. Work-efficient query evaluation with prams. In Floris Geerts and Brecht Vandevoort, editors, *26th International Conference on Database Theory (ICDT)*, pages 16:1–16:20, 2023. doi:10.4230/LIPIcs.ICDT.2023.16.

12 Tsvi Kopelowitz, Ely Porat, and Yair Rosenmutter. Improved worst-case deterministic parallel dynamic minimum spanning forest. In Christian Scheideler and Jeremy T. Fineman, editors, *Proceedings of the 30th on Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 333–341, 2018. doi:10.1145/3210377.3210403.

13 Sushant Patnaik and Neil Immerman. Dyn-FO: A Parallel, Dynamic Complexity Class. *J. Comput. Syst. Sci.*, 55(2):199–209, 1997. doi:10.1006/jcss.1997.1520.

14 Thomas Schwentick, Nils Vortmeier, and Thomas Zeume. Sketches of dynamic complexity. *SIGMOD Rec.*, 49(2):18–29, 2020. doi:10.1145/3442322.3442325.

15 Peter van Emde Boas. Machine models and simulation. In Jan van Leeuwen, editor, *Handbook of Theoretical Computer Science, Volume A: Algorithms and Complexity*, pages 1–66. Elsevier and MIT Press, 1990.