Wait-Free Concurrent Graph Objects with Dynamic Traversals

Nikolaos D. Kallimanis[1] and Eleni Kanellou[2]

1 **FORTH-ICS, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece**
nkallima@ics.forth.gr

2 **FORTH-ICS, Foundation for Research and Technology – Hellas (FORTH), Institute of Computer Science (ICS), Heraklion, Greece, and University of Rennes 1, Rennes, France**
kanellou@ics.forth.gr

Abstract

Graphs are versatile data structures that allow the implementation of a variety of applications, such as computer-aided design and manufacturing, video gaming, or scientific simulations. However, although data structures such as queues, stacks, and trees have been widely studied and implemented in the concurrent context, multi-process applications that rely on graphs still largely use a sequential implementation where accesses are synchronized through the use of global locks or partitioning, thus imposing serious performance bottlenecks. In this paper we introduce an innovative concurrent graph model that provides addition and removal of any edge of the graph, as well as atomic traversals of a part (or the entirety) of the graph. We further present `Dense`, a concurrent graph implementation that aims at mitigating the two aforementioned implementation drawbacks. `Dense` achieves wait-freedom by relying on helping and provides the inbuilt capability of performing a partial snapshot on a dynamically determined subset of the graph.

1998 ACM Subject Classification F.1.2 Modes of Computation

Keywords and phrases graph, shared memory, concurrent data structure, snapshot

Digital Object Identifier 10.4230/LIPIcs.OPODIS.2015.27

1 Introduction

The irrevocable paradigm shift towards multi-core hardware has been accompanied by developments in concurrent data structure design. Data structures that are at the heart of many applications built for the sequential setting have been ported into the concurrent domain. Such data structures include trees (e.g. [2, 6, 10, 12]), stacks (e.g. [5, 14, 17, 23]) or queues (e.g. [16, 23, 25, 34]).

However, numerous are also the applications that rely on data structures with a more complex or irregular morphology. An example of such a data structure is the *graph*. A graph consists of a set V of *vertices* and a set E of *edges*, which are pairs of type (x, y), where x and y are elements of V. Each edge may additionally be associated with a value w, referred to as the *weight* of the edge, out of some set W. Vertices can be used to represent many types of entities, from simple or complex data structures to tasks, functions or processes, while the edges can flexibly express several types of relations. Graphs are essential building blocks for applications in robotics (e.g. [9]), machine learning (e.g. [36]), automated design of digital circuits (e.g. [24]), task scheduling in operating systems (e.g. [27]), garbage collection (e.g. [39]), and video-game design (e.g. [7]) to name a few.

Thus, in the multi-core era, applications that rely on graphs are also important in the concurrent context, where they can be used either in message-passing or in shared memory settings. Notable shared memory examples include Computer-Aided Manufacturing (CAM) applications, scientific simulation tools, or video games, where a virtual world is represented as a graph. Vertices of the graph model objects in the virtual world, while edges represent the relationships and interactions of those objects [3]. As these relationships and interactions are unlikely to be static, this implies that processes executing the application need to update the structure and connectivity of the graph frequently. However, concurrent applications that rely on graphs still mostly use sequential implementations and synchronize the access of multiple processes to them with the help of a global lock or by partitioning the vertices into disjoint sets that can then individually be accessed by different processes [37].

If a global lock is used, accesses become expensive and the lock poses a global bottleneck for overall performance. Furthermore, even if fine-grained locking is employed on subsets of the graph, the liveness of the application is restricted, as it becomes blocking by the reliance on locks. By relying on graph partitioning instead, the interaction of processes for the purpose of synchronizing accesses to the graph can be minimized, as different processes access distinct partitions. Such an implementation can offer a higher degree of parallelism to applications that are suitable for exploiting it. However, such an application then either has to rely on a static, predefined partitioning, or it has to perform re-partitioning in order to accommodate dynamicity, in which case synchronization among all processes may be necessary and performance is likely to suffer.

These issues may be aggravated when the operation to be performed affects the entire graph or a subset of it, as is the case with iterations, searches, or partial traversals. In those cases it is important to provide a consistent view of the graph or some subgraph for the operation, but without blocking or impeding concurrent updates on it. This possibility is offered by atomic snapshots [1]. A snapshot is a concurrent object that consists of a collection of components and which provides two operations: `Update`, which modifies the value of one of the components, and `Scan`, which returns a consistent view of the values of all components. To implement a graph, a one-to-one correspondence is established between graph edges and snapshot components. Modifications on the graph structure are performed by using the `Update` operation of the snapshot, while reads are done by using `Scan`.

When only a subset of the graph has to be accessed, the overhead that a snapshot incurs can be avoided by using partial snapshot implementations, such as [4, 22]. There, `Scan` takes as input argument the subset of components on which the snapshot is to be performed. However, this precludes applications, in which the subgraph that should be accessed is not known a priori, e.g. when there is the need to perform a random traversal on some part of the graph, where the next edge to be accessed is determined dynamically.

In this paper we are concerned with the problem of providing such a dynamic traversal for graphs. A generalized solution consists in using *transactional memory* (TM) [20, 35]. This paradigm offers the *transaction* abstraction. A transaction encapsulates one or several read and update operations and attempts to apply them atomically. If this is possible, the transaction *commits* making the effects of the operations visible. Otherwise, it *aborts* discarding any change. We want to avoid the commit/abort semantics inherent in this paradigm, the performance overheads that this paradigm can entail, as well as well-known progress limitations [8]. For this reason, we define a novel concurrent graph object. It supports the addition, removal, and modification of edges of the graph, as well as the atomic walk of a dynamically-determined subset of the graph edges, such that a consistent view of it may be obtained. The model we propose for the dynamic traversals is reminiscent of TM:

Dynamic traversals resemble read-only transactions, while modifications of the edges of the graph behave like mini-transactions that encapsulate a single update on a single transactional object. Contrary to common transactional semantics, however, our model does not need, and so does not include the possibility to abort.

We further present `Dense`, a adjacency matrix concurrent graph implementation based on the proposed model. `Dense` provides the linearizable operation `Update`, which is used to add or remove edges of the graph or to modify existing edge weights. The *step complexity* (i.e., the total number of accesses in the shared base objects) is $O(k)$, where k is the number of active processes (i.e. `Dense` is an adaptive algorithm). Dynamic traversals are implemented as a composite operation, reminiscent of read-only transactions. For this purpose, `Dense` provides `DynamicTraverse`, another linearizable operation that is used to initiate a dynamic traversal, and a matching `EndTraverse` operation which terminates it. An auxiliary routine, `Read`, is used for obtaining edge weights. Instances of `Read` that are enclosed by an instance of `DynamicTraverse` and a matching instance of `EndTraverse`, return a consistent view for the weights of the edges they read. As such, dynamic traversals appear to be atomic and can be considered as a virtual operation that is linearizable.

`Dense` operations are *wait-free*, i.e. an operation by a process that does not fail terminates in a finite number of steps in any execution. Wait-freedom is achieved by employing light-weight *helping* using a mechanism reminiscent of the one presented in [19]. Instances of operations that are concurrent with instances of `Update` help them carry out edge modifications. Operations are aware of concurrent active dynamic traversals and ensure that those dynamic traversals can return a consistent view by storing old edge versions for them (in the worst case, `Dense` keeps n different versions, one for each process, on a given edge of the graph). The step complexity of `DynamicTraverse` and `Read` is $O(k)$ and $O(1)$, respectively. `Dense` uses m `LL/SC` base objects (one for each edge), one atomic *Add* base object of n bits, and an additional `LL/SC` object. `Dense` is of theoretical interest since the size of `LL/SC` objects is big. We further present `S-Dense`, a practical version of `Dense`, with step complexity $O(k^2)$, which uses small base objects.

The rest of the paper is organized as follows. Section 2 summarizes relevant literature. Section 3 provides our view of the underlying system. Section 4 gives an overview of the defining characteristics of the proposed implementation, while Section 5 provides a detailed description of it and sketches out the proof of the correctness argument. Finally, Section 6 contains a discussion of the presented results and of prospects for future work.

2 Related Work

A great body of work on the concurrent implementation of graph algorithms tackles common graph-related issues (e.g. [11, 29, 33]) and focuses either on parallelizing existing sequential algorithms or on providing concurrency through the use of locks on well-known sequential algorithms. Then, liveness guarantees are rather relaxed, as most of these implementations are blocking. In contrast, we are interested in the graph as a general-purpose, concurrent data structure and are especially concerned with providing wait-freedom and linearizability.

Work on concurrent data structures has been devoted to commonly-used ones, such as queues, stacks, or trees, with the focus on providing interesting progress properties – initially simply by avoiding locks (e.g. [28, 34]), and recently a step further, by proposing wait-free implementations. Notably, [25] uses helping via an announce array in order to make a wait-free version out of the `CAS`-based lock-free queue of [28]. Together with a "fast path, slow path" methodology [38], previously used for the implementation of a wait-free linked list

out of well-known lock-free design [15], this method is proposed as a generalized methodology of designing wait-free concurrent data structures, given a lock-free implementation [26]. Our method is "stand-alone", providing wait-freedom without requiring a lock-free design as base.

Recently, techniques that provide *iterators* of concurrent data structures have been proposed. An iterator parses a data structure in order to obtain a consistent view. In [31], a methodology is proposed for enhancing lock-free or wait-free set-based data structures with a `CAS`-based implementation of a wait-free iterator. It entails reporting data structure updates to any active snapshot, so that they can be taken into account, depending on the order of linearization. In [32], update and read operations on a trie are aware of an ongoing iterator, and copy – and thus, effectively rebuild – the parts of the trie that they access, leaving intact the albeit obsolete version that the iterator is parsing. The complexity is divided among updates and reads, while the snapshot occurs in constant time. In [30] a theoretical framework for defining the consistency of iterators is proposed and used in a case study that equips the well-known lock-free concurrent queue of [28] with a wait-free iterator that is linearizable according to the provided framework.

We, however, are interested in a partial view, which, furthermore, is dynamically defined. Thus, we want to avoid the overhead that is induced by iterating over the entire data structure. Arguably, the implementation in [32] does not induce it, having a constant-time snapshot. However, to achieve that, it must employ either double compare, single swap (`DCSS`) primitives, or a custom-made, `CAS`-like software primitive,unlike our method, which simply relies on `LL/SC`. Moreover, those works take advantage of the structural regularity of the underlying data structure. In contrast, a graph usually has irregular characteristics. Our work is more akin to partial snapshots, such as [4, 22], as we use an adjacency matrix to represent the graph. However, partial snapshots are more restrictive than our model as they require a priori knowledge of the component subset to be scanned.

The required dynamicity can be provided by using transactional memory to access a graph. Indeed the dynamic traversal provided by our model resembles a read-only transaction. However, efficient TM algorithms commonly rely on locks, while even obstruction-free or non-blocking ones commonly burden reads and updates with the processing overhead necessary for conflict-detection and resolution (cf. with [13] for a survey on TM algorithmic techniques). We wish to avoid these issues, as well as the commit/abort semantics inherent in TM, but unusual for data structures. The recent impossibility result in [8] further implies that, even if commit/abort semantics are included in our model, the TM progress property equivalent to wait-freedom cannot be achieved.

3 Model

System model. We assume an asynchronous, shared memory system of n processes, which communicate by accessing *base objects*. A base object O has a state and provides a set of *primitives*, used by processes in order to access, i.e. read and/or modify, the state of O. We use the following base objects: A read/write object O has a state that takes values out of some set S. It provides the primitives $read(O)$, which returns the state of O, and $write(O, v)$, $v \in S$, which sets the state of O to v. An *Add* object O has a state that takes values out of some set of integers S. It provides the primitives $read(O)$, which returns the state of O, and $add(O, v)$, $v \in S$, which adds the value v to the state of O. An `LL/SC` object O has a state that takes values out of some set S. It provides the primitives $\texttt{LL}(O)$ and $\texttt{SC}(O, v)$, $v \in S$. $\texttt{LL}(O)$ returns the current state of O. $\texttt{SC}(O, v)$, executed by a process p_u, $u \in \{1, 2, \ldots, n\}$, must follow an execution of $\texttt{LL}(O)$ also by p_u. It changes the state of O to v if O has not

changed since the last execution of $\texttt{LL}(O)$ by p_u. The concurrent implementation of a data structure also has a *state*, stored in shared base objects. It provides algorithms for the operations provided by the data structure, which processes may use in order to access or modify its state. A process executes an operation by issuing an *invocation* for it and an operation terminates by returning a *response* to the process.

Executions. At any point in time, the system is characterized by a *configuration* C, which is a vector that contains the state of each process in the system and the state of each base object. We denote by C_0 the initial configuration of the system. A *step* ϕ is either the execution of a primitive by some process, or the issuing of an operation invocation by some process, or the response of some operation to some process.

A (possibly infinite) sequence $C_0, \phi_1, C_1, \ldots, C_{i-1}, \phi_i, C_i, \ldots$, of alternating configurations (C_k) and steps (ϕ_k), starting from the initial configuration C_0, where for each $k \geq 0$, C_{k+1} results from applying step ϕ_{k+1} to configuration C_k, is referred to as an *execution*. A subsequence of an execution α in the form $C_i, \phi_{i+1}, C_{i+1}, \ldots, C_j, \phi_{j+1}, C_{j+1}$, of alternating configurations and steps, starting from some configuration C_k, $k > 0$, is referred to as an *execution interval* of α.

If some configuration C occurs before some configuration C', $C \neq C'$, in an execution α, then we say that C *precedes* C' in α and denote it as $C < C'$. Conversely, we say that C' *follows* C in α. Precedence among a step ϕ_i and a step ϕ_j, or precedence among a step ϕ_i and a configuration C_j is defined in a similar fashion and denoted by the same operation $<$.

Let α_1 and α_2 be two execution intervals of some execution α. If the last configuration of α_1 precedes or is the same with the first configuration in α_2, then we say that α_1 precedes α_2 and denote it $\alpha_1 < \alpha_2$. In that case we also say that α_2 follows α_1. If neither $\alpha_1 < \alpha_2$ nor $\alpha_2 < \alpha_1$ are the case, then we say that α_1 and α_2 *overlap*.

Given the instance of some operation *op* for which the invocation and response steps are included in α, we define α_{op}, the execution interval of *op*, as that subsequence of α which starts with the configuration in which *op* is invoked and ends with the configuration that results from the response of *op*. If there are no two operation instances op_1, op_2 in α for which the execution intervals overlap, then we say that α is a *sequential* execution, or that operations in α are executed *sequentially*.

Concurrent graph. A *graph* $G = \langle V, E \rangle$ is composed of V, a (finite) set of elements referred to as *vertices*, and E, a set of pairs of vertices, referred to as the *edges* between them. Each edge $e_{i,j} \in E$ has a weight w_{ij}, that takes values out of some set W. A graph supports several abstract operations, well-known in literature, such as operations for adding vertices or edges, deleting vertices or edges, modifying attributes of vertices or edges, returning specific subsets of the graph vertices or edges, etc. A *concurrent graph* is a graph that can be accessed concurrently, through those types of operations, by n processes.

We propose the *dynamic traversal* (henceforth referred to as *d-traversal* for brevity) as a concurrent graph operation exhibiting the following characteristics: (i) starts from a vertex v of the graph, (ii) visits a sequence of vertices that is not necessarily known at the point that the traversal initiates, (iii) the sequence of visits may be decided while the visiting is taking place; (iv) the dynamic traversal returns a *consistent view* of the weights of all the edges that it has traversed, i.e., all the returned values have co-existed at some point in time.

We further propose a concurrent graph implementation. The graph is represented as an $m \times m$ adjacency matrix, for some positive integer m, and it allows the addition, and removal of edges, the modification of edge weights by providing an `Update` operation. $\texttt{Update}(i, j, w)$,

where i, j are indices of vertices in V and where w is in $W \cup \{\perp\}$, modifies the graph as follows: Assume that $e_{i,j} \in E$. If $w = \perp$, then the edge is removed. Otherwise, its weight is changed to w. If $e_{i,j} \notin E$, then it is inserted in E with weight w. The implementation supports the *d-traversal* as a composite operation, consisting of the following ones:

- `DynamicTraverse`, which is used to mark the beginning of a *d-traversal* of the graph.
- `EndTraverse`, which is used to mark the end of a *d-traversal* of the graph.
- `Read`(i, j), where i, j are indices of vertices in V. It returns a weight for edge e_{ij},, if $e_{i,j} \in E$, and $\perp$ if $e_{i,j} \notin E$.

`Read` is only used in *d-traversals*, as part of a sequence of `Read` operations. A *d-traversal* by process p_u consists in an instance bt of a `DynamicTraverse` operation, followed by a sequence of instances of `Read`, followed in turn by an instance et of an `EndTraverse` operation. No other operation is invoked between bt and et. The execution interval of the *d-traversal* starts in the configuration in which p_u invokes bt and ends in the configuration resulting from the response of et. For correctness argumentation, we consider *d-traversal* as a virtual operation that is invoked at the point that `DynamicTraverse` is invoked and responds at the matching `EndTraverse` (if any).

Correctness criteria. We consider *linearizability* [21] as correctness criterion for the graph operations. An execution α is *linearizable* if it is possible to assign a *linearization point* inside the execution interval of each operation in α (`Update` and *d-traversal* operations), so that the result of these operations is the same as it would be, if they had been performed sequentially in the order dictated by their linearization points. A *d-traversal* operation is considered linearizable if it is possible to assign a linearization point in its execution interval so the result of the `Read` operations enclosed in it, is the same as it would be, if all the `Read` operations had been performed at the linearization point in the sequential execution (the order of `Read` is imposed by the invocation steps). Roughly speaking, we consider that the entire sequence of `Read` operations enclosed in a dynamic traversal have a linearization point inside the execution interval of the *d-traversal*, such that the `Read` return the weights that the traversed edges had in the configuration in which the linearization point is placed.

Progress criteria. In this work we consider that processes that participate in an execution α may suffer from *crash failures*, i.e. we consider that a process may unexpectedly stop taking steps in α after some configuration. In this context, we provide a graph implementation with operations that satisfy *wait-freedom* [18]. A data structure implementation is wait-free if in any execution, each process finishes the execution of every operation it initiates within a finite number of steps independently of the speed or the failure of other processes.

4 Main Ideas

Our implementation provides linearizable, wait-free operations and linearizable *d-traversals* by using light-weight helping. To achieve it, each `Update` or `DynamicTraverse` operation is first *announced* by a process, subsequently *agreed* by all processes, and then it can be *applied* by some process – not necessarily the one that invoked it – and finally, terminate and return a response. Processes provide very light-weight assistance to each other when applying operations. In the case of `DynamicTraverse` operations, helping consists in storing an integer number for it, while in the case of `Update`, helping consists in applying the update on the edge. In order to coordinate how announced operations are executed among multiple

processes, the processes collaborate to alternate between two types of phases, namely `AGREE` and `APPLY`. An announce array of dimension n, an n-bit bitvector *Add* object, and two sets of n bits each, namely *ann* and *done*, are used for this coordination.

To announce an operation, a process p_u, $1 \leq u \leq n$, starts by writing the operation type and arguments (the *operation information* or op-info for brevity) in the u-th element of the announce array. Notice that p_u is the only process that may write to this element while all processes in the system can read it. Subsequently, p_u flips the value of the u-th bit of the bitvector. Other processes can now also help the operation of p_u.

The `AGREE` phase is used by processes in order detect which op-info in the announce array corresponds to a pending operation: p_u has a pending operation if the u-th bit of the bitvector is not equal to $done[u]$. In this phase, processes essentially "agree" on a set of operations that they will attempt to apply on the graph in the following `APPLY` phase. Then, the `APPLY` phase that follows is used by processes for attempting to apply those pending operations. As a result, operations are applied to the graph in batches. When an announced operation is carried out by some process, we say that it is *applied*. Otherwise, it is *pending*. An applied operation can return a response to the process that invoked it. The status of an operation, i.e. whether it has been already applied or not, is reflected in the values of $ann[u]$ and $done[u]$: An invariant in our implementation is that when $ann[u] = done[u]$, the latest agreed operation by p_u has been applied, while when $ann[u] \neq done[u]$, it is pending. A process which completes the actions associated with a phase, attempts to flip it.

A shared integer *seq* acts as global version counter and is also used to make the wait-free partial traversals possible. Each time the phase changes from `AGREE` to `APPLY`, *seq* is incremented. Apart from their weight, edges of the graph also have a version number as a further attribute. In each configuration, the version of an edge is the value of *seq* at the configuration in which the edge was last updated. Each *d-traversal* that is initiated, is also assigned – either by the process that executes it or by some other, helping process – the value that *seq* has at the configuration in which its `DynamicTraverse` is applied. This value is referred to as the *d-traversal read value* and is visible to all processes.

Before a process p_u applies an `Update` on edge $e_{i,j}$, it also detects whether a *d-traversal* by some other process p_l, $1 \leq u \leq n$, is active. If the *read version* of p_l's *d-traversal* is lower than the current version of $e_{i,j}$, p_u stores the weight and version of $e_{i,j}$ for p_l – as *previous weight* and *previous version* for p_l – and updates them only afterwards. For this, each edge contains a vector *prev* of dimension n, where element l corresponds to p_l and contains the previous weight and previous version that p_l may need during its *d-traversal*.

When p_l uses `Read` to collect the weight-version pair of an edge that it accesses during a *d-traversal*, it compares the edge's version with the *read version* of the *d-traversal*. If it is greater, p_l collects the previous weight and previous version of the edge. Otherwise it uses the current weight and value. This way, all the `Read` that are enclosed between the `DynamicTraverse` and the matching `EndTraverse` operation of p_l's *d-traversal*, return mutually consistent values and the *d-traversal* forms a consistent view.

5 `Dense`, a Concurrent Graph Implementation

In the following, `Dense`, an algorithm for our proposed concurrent graph implementation, is presented. `Dense` is so named because it is mostly suitable for dense graphs, i.e. graphs with high connectivity, in which case the allocated adjacency matrix is sufficiently exploited.

Listing 1 shows the data structures used by `Dense` (initial values are indicated on lines 20–23). Operation information is stored in a structure of type *AnnStruct*. This structure

```
1  type OpType = {DynamicTraverse, Update, Noop}; //operation types

2  struct AnnStruct // the data type of the announce array elements
3    OpType op; // the  announced operation
4    int i, j; // if OpType=Update, e_{i,j} has to be updated
5    int value; // weight to be assigned to e_{i,j} if OpType=Update
6  };

7  struct StateStruct // data type for storing the graph's state
8    int seq; // the sequence number, used as a version counter
9    boolean phase; // current phase of execution, Announce or Apply
10   int ann[1..n]; // used as n-bit vector
11   int done[1..n]; // used as n-bit vector
12   int rvals[1..n]; // read value for each process
13 };

14 struct EdgeStruct { // the data type of a graph edge
15       ⟨ weightval, int ⟩ prev[1..n]; // one element per process
16       int seq; // current version of the edge
17       weightval w; // current weight of the edge
18 };

19 shared int BitVector; // used as n-bit vector
20 shared AnnStruct Announce[1..n] = {⟨⟨Noop, 0, 0, 0⟩, ..., ⟨Noop, 0, 0, 0⟩⟩};
21 shared StateStruct ST = ⟨0, AGREE, ⟨0, ..., 0⟩, ⟨0, ..., 0⟩, ⟨0, ..., 0⟩⟩;
                            // operations status and phase indicator
22 shared EdgeStruct Edges[1..m][1..m] = {⟨⟨0, 0⟩, 0, 0⟩, ..., ⟨⟨0, 0⟩, 0, 0⟩⟩};
                          // adjacency matrix representing the graph
23 private int toggle_u = 2^u; // a variable per process, u ∈ {1, ..., n}
```

Listing 1 `Dense`: Data structures for a concurrent graph object suitable for dense graphs.

consists of four fields, namely: (i) *op*, of type *OpType*, which represents operations provided by `Dense` (i.e., `DynamicTraverse`, `Update`, and the void operation `Noop`); (ii) i and j which identify the edge on which an `Update` operation is to be applied (if $op =$ `Update`); and (iii) *value*, an integer representing the value that an `Update` operation has to apply to the weight of the edge specified by fields i and j (if $op =$ `Update`).

The status of operations on the graph is indicated by ST, an `LL/SC` object of type *StateStruct* consisting of: (i) *seq*, an integer which serves as global version counter. It is incremented each time a process successfully switches the execution phase from `AGREE` to `APPLY`; (ii) *phase*, a boolean variable which indicates whether the execution of `Dense` is in an `AGREE` or an `APPLY` phase at any given moment; (iii) $ann[1..n]$, an array implemented as n-bit integer, where $ann[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and whose value is toggled each time an operation by p_u is agreed; (iv) $done[1..n]$, an array implemented as n-bit integer, where $done[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and whose value is set equal to $ann[u]$ each time an operation by p_u is applied to the graph; and (v) $rvals[1..n]$, an array of n elements, where t $rvals[u]$ corresponds to process p_u, $u \in \{1, 2, \ldots, n\}$, and which stores the value of *seq* that p_u uses as *read version*, in case it is performing a *d-traversal*.

We represent the graph G with *Edges*, an adjacency matrix, i.e. a two-dimensional array, where each element (i, j) of the array represents edge between vertices i and j, $i, j \leq m$. Graph edges, i.e. adjacency matrix elements, are `LL/SC` objects of type *EdgeStruct*. This type is a record of three fields: (i) *prev*, an array of n elements (one for each process),

```
24 void Update(int i, int j, int value) { //for process p_u, u ∈ {1,...,n}
25   BTU(Update, int i, int j, int value)
26 }

27 void DynamicTraverse() { //for process p_u, u ∈ {1,...,n}
28   BTU(DynamicTraverse, ⊥, ⊥, ⊥);
29 }

30 void EndTraverse() { //for process p_u, u ∈ {1,...,n}
31   ;
32 }

33 int Read(int i, int j) { //for process p_u, u ∈ {1,...,n}
34   EdgeStruct edge;
35   int val, int seq, int rval;
36   edge = Edges[i][j];
37   rval = ST.rvals[u];

38   if (edge.seq > rval) {
39     ⟨val, seq⟩ = edge.prev[u];
40   }
41   else val = edge.w;
42   return val;
43 }
```

Listing 2 `Dense`: Operations `Update`, `DynamicTraverse`, `EndTraverse`, and `Read`.

where each element is a pair $< w, seq >$ of integers. Whenever an update operation modifies the weight of an edge, it stores the current weight and version in $prev[u]$ if process p_u is performing a *d-traversal* on the graph using as read value, stored in $ST.rvals[u]$, a value that is larger than the current version of the edge; (ii) seq, an integer which stores the current version of the edge; (iii) w, of type $weightval$, which stores the current weight of the edge - recall that if this value is $\perp$, the corresponding edge does not exist.

Recall that `Dense` implements the helping mechanism, where any process p_u that invokes an operation also attempts to apply pending operations by other processes. Operation information is stored by processes in $Announce[1..n]$, an announce array of n elements, where each element $Announce[u]$, $u \in \{1, 2, \ldots, n\}$, is of type $AnnStruct$ and can be written to only by process p_u, but can be read by all processes. The announcing of an operation is complemented by the use of $BitVector$, shared vector of n bits (represented as a n-bit integer) where bit u corresponds to process p_u. In order to indicate a pending operations, each time p_u announces operation information in $Announce[u]$, it flips the u-th bit of $BitVector$. It does so with the aid of a local, persistent variable, $toggle_u$, with initial value 2^u. After p_u announces an operation, it inverts the value of $toggle_u$.

Pseudocode Description. Pseudocode for the operations of the graph that are described in Section 3 is presented in Listing 2. Operations `Update` and `DynamicTraverse` require that the processes that execute them, assist each other. In order to do this, they both invoke auxiliary routine `BTU` (these initials stand for "*B*egin a *T*raversal or *U*pdate"). `BTU` implements the phase alternation and is further detailed below. We say that an execution of `Dense` is in `AGREE` or `APPLY` phase during those execution intervals in which $ST.phase =$ `AGREE`, or $ST.phase =$ `APPLY`, respectively. Notice that `Read` is independent of the phases. Instances of

`Read` are only invoked by a process following the execution of a `DynamicTraverse` operation by the same process. They rely on `Update` operations to store possibly useful old edge versions for them in the *prev* arrays of each modified edge.

The `DynamicTraverse` operation that initiates some *d-traversal* d, obtains as *read version* the current value v of $ST.seq$ (this happens when either the process that initiated d or some other process helps to apply this `DynamicTraverse` operation while executing line 74). An instance r of `Read` that is invoked by process p_u on edge $e_{i,j}$ and that is included in d, must check whether the version of $e_{i,j}$ is greater than v (line 38). If this is the case, then $e_{i,j}$ was updated after d started. However, in `Dense`, *d-traversals* must not be aware of concurrent edge updates and have to return values that the edge weights had before the *d-traversal* initiated. For this reason, r must return a previous weight of $e_{i,j}$, and finds this in $e_{i,j}.prev[u]$ (line 39). If the version of $e_{i,j}$ is less than v, then r returns $e_{i,j}$'s current weight (line 41). Notice that although the instances of `Read` that are included in a *d-traversal* are not aware of concurrent `Update` instances (i.e. instances whose execution intervals overlap with that of the *d-traversal*), those `Update` instances become aware of *d-traversals* and store the necessary old edge weights for them when they modify edges the graph.

Listing 2 presents `BTU`, which is at the heart of the `Dense` implementation. It is invoked by `Update` specifying as arguments the operation type, integers i and j, which identify the edge to be modified, and integer *value*, which specifies the weight to be written to this edge. When `BTU` is invoked by `DynamicTraverse`, then only the operation type is specified as argument, while the remaining three are $\perp$, as they are not required for the *d-traversal*.

An instance of `BTU` by p_u first writes the operation information into element u of the announce array (line 48) and then sets the value of the u-th bit of $BitVector$ (line 49), using the current value of local persistent variable $toggle_u$ bit. It then flips $toggle_u$ (line 50) in order to prepare its value for the next execution of an operation by p_u. The algorithm implements this practice in order to provide a previously mentioned invariant: by comparing $ST.ann[u]$ and $ST.done[u]$, a process is able to detect whether the latest agreed operation by p_u has already been applied or not. Notice that the contents of $BitVector$ are copied into $ST.ann$ by each process that successfully executes an `AGREE` phase of `Dense` (lines 53, 56, 80), while they are copied into $ST.done$ by a process that successfully executes an `APPLY` phase of `Dense` (lines 53, 77, 80). Therefore, each operation by p_u must correspond to a different $BitVector[u]$ value than the previous one.

`BTU` carries out any light-weight helping in addition to the execution of the operation that invoked it. To do this, it iterates via a `for` loop (lines 51-81). An iteration of this `for` loop consists in locally copying ST (line 52), and then attempting to perform the actions that are required by the phase indicated in $ST.phase$. Once these actions have been performed, `BTU` attempts to change the phase by executing the `SC` of line 80. If this `SC` is successful, we say that `BTU` (or, abusing terminology, the process or the operation that invoked it) successfully executed the phase. The execution of this primitive may fail if some instance of `BTU`, executed by a process other than p_u, has already performed the current phase and advanced the execution to the next phase. When executing the `for` loop (lines 51-81), `BTU` proceeds as follows, depending on the phase it performs:

- `AGREE` phase (lines 55-58). This phase updates the status record ST with the newly announced operations, so that all processes can agree on them. So, `BTU` first records this status locally on st, before using an `SC` instruction in order to attempt to update it globally on ST. In order to set st, `BTU` collects information from the $BitVector$ regarding newly announced and therefore possibly pending operations. It does so by copying the contents of $BitVector$ into $st.ann$ (line 56). Notice that for a process $p_l, 1 \leq l \leq n$

```
44 void BTU(OpType op, int i, int j, int value) { //for p_u, u ∈ {1,...,n}
45   StateStruct st;
46   int lbv, opi, opj;
47   EdgeStruct e;

48   Announce[u] = ⟨op, i, j, value⟩;
49   Add(BitVector, toggle_u);
50   toggle_u = - toggle_u;

51   for (i=0; i < 4; i++) {
52     st = LL(ST);
53     lbv = BitVector;

54     if (lbv[u] == st.done[u]) break;

55     if (st.phase == AGREE) {  // AGREE Phase
56       st.ann[1..n] = lbv[1..n];
57       st.seq = st.seq + 1;
58       st.phase = APPLY;
59     } else {  // APPLY Phase
60       for (r = 1; r ≤ n; r++) {
           // at most k shared memory accesses, k=active processes
61         if (st.ann[r] ≠ st.done[r]) {
62           if (Announce[r].op == Update) {
63             opi = Announce[r].i;
64             opj = Announce[r].j;
65             e = LL(Edges[opi][opj]);
66             if (e.seq < st.seq) {
67               for (k = 1; k ≤ n; k++) {
68                 if (e.seq < st.rvals[k]) e.prev[k] = ⟨e.w, e.seq⟩;
69               }
70               e.w = Announce[r].value;
71               e.seq = st.seq;
72               SC(Edges[opi][opj], e);
73             } // if (e.seq < st.seq)
74           } else st.rvals[r] = st.seq;
75         } // if (st.ann[r] ≠ st.done[r])
76       } // for (r = 1; r ≤ n; r++)
77       st.done[1..n] = lbv[1..n];
78       st.phase = AGREE;
79     }
80     SC(ST, st);
81   }
82 }
```

Listing 3 Dense: BTU auxiliary routine.

that has a newly announced operation, the invariant $st.ann[u] \neq st.done[u]$ must hold. Therefore, a successful assignment of st to ST (through the execution of the SC of line 80) creates the inequality between $ST.ann[u]$ and $ST.done[u]$ and makes all processes "agree" that p_u has a newly announced operation which has not been applied yet. Once the information regarding pending operation for each process has been copied into st, BTU increments seq, the global version counter in st (line 57) and changes the $phase$ field of st from AGREE to APPLY.

- `APPLY` phase (lines 59–78). This phase applies any pending agreed `Update` operation on the edges of the graph, and assigns *read version* to any pending agreed `DynamicTraverse` operation. For this, `BTU` uses st again, and for each process p_u (line 60) it checks whether such a pending operation exists (line 61), in which case it holds that $st.ann[u] \neq st.done[u]$. Consider the case of a pending `Update` operation by p_u on edge $e_{i,j}$. Since multiple processes may be executing an operation on $e_{i,j}$, these modifications must be synchronized in order to safeguard correctness. For this reason, $e_{i,j}$ is copied locally into e using `LL` (line 65). If the current version number of $e_{i,j}$, $e.seq$ is greater than $st.seq$ then the specific `Update` operation has already been applied, namely by some process other than p_u, that has also changed the state. However, if this is not the case, the modification of $e_{i,j}$ is carried out. Before setting the new value for the weight (line 70) and version (line 71) of $e_{i,j}$, a comparison of the current version of $e_{i,j}$ and all *read versions* stored in $st.rvals$ is performed (lines 67–68). If the current version of $e_{i,j}$ is less than the *read version* for some process p_r, $1leqr \leq n$, then the condition $e.seq<st.rvals[r]$ is true. This means that a concurrent *d-traversal* by process p_r might be in progress. In order to guarantee that an eventual such *d-traversal* can read mutually consistent values, the current values of $e_{i,j}$'s weight and version are stored in $e.prev[r]$. There, instances of `Read` on $e_{i,j}$ that are included in a *d-traversal*, can later find it if necessary. `BTU` then attempts to finalize the update of $e_{i,j}$ by using `SC` to copy e into $e_{i,j}$ (line 72). Whether the `SC` on the edge is successful or not, the operation is considered applied.
  If p_u's pending operation is a `DynamicTraverse`, the *read version* must simply be set. This is first recorded in $st.rvals[u]$ (line 74) and is eventually stored in $ST.rvals[u]$ (line 80) by the process that successfully executes the phase. Recall that it is used by a concurrent `Update` operation in order to judge whether to discard the current value of the edge that it is updating or whether to keep it for the ongoing *d-traversal* of p_u. If the assignment of line 74 followed by a successful `SC` on ST is executed more than once for a given `DynamicTraverse` instance or for the *d-traversal* that it initiated, then the consistency of the `Read` instances of the *d-traversal* could be compromised. An eventual bad scenario would happen if `Read` instances that are invoked before the second execution of those lines and `Read` instances that are invoked after the second execution would use a different *read version* when reading edges.
  Thus, at the end of an `APPLY` phase, the *done* bits in st are set equal to the corresponding *ann* bits (line 77). Then, `BTU` attempts to change the phase from `APPLY` back to `AGREE` (line 78) by switching the *phase* field of st, which is reflected on ST if the `SC` instruction of line 80 is successful.

Notice that an instance of `BTU` may be slow and end up performing the actions associated with a phase while the execution has already progressed to the next phase. Notice also that in the worst case, an instance of `BTU` has to perform four iterations of the `for` loop before the operation that invoked it is applied. Such a worst-case scenario is the following: Let I_{btu} be an instance of `BTU` that executes the first iteration of the `for` loop during an `AGREE` phase and let p_l be the process that successfully flips the phase to `APPLY` by executing the `SC` on ST of line 80. Consider however that the execution of line 49 by I_{btu} occurs after p_l executes the `LL` of line 52, which corresponds to the successful `SC` on ST. This means that in the following `APPLY` phase, the operation that invoked I_{btu} will not be executed. In the worst case, all other processes are slow and the process that invoked I_{btu} must perform the actions associated with the `APPLY` phase itself, during the second iteration of the `for` loop, as well as the actions required by the following `AGREE` phase, during the third iteration of its `for` loop. During this `AGREE` phase, the *Add* on $BitVector$ by I_{btu} is guaranteed to be observed

by the process that performs the successful SC on ST and changes the phase to APPLY. Here again, in the worst case, all other processes are once more slower than the process which invoked I_{btu}, and thus it is I_{btu} that performs the actions associated with the APPLY phase, in its fourth iteration of the for loop. This time, however, the operation that invoked it is guaranteed to have been applied.

However, in the common case, the operation may be applied earlier, by some other, helping process. The condition that signals this is expressed on line 54 and is checked at each iteration of the for loop. It consists in verifying whether the toggle bit for p_l, the process executing BTU, in shared array $BitVector$ has the same value as the corresponding bit in the $ST.done$ array. If that is the case, the operation executed by BTU is considered applied and the iteration of the for loop terminates as well.

S-Dense. We provide S-Dense, a variation on Dense, implemented with smaller registers. S-Dense is meant to conform to current machine architectures restrictions. The main difference with Dense is the implementation of the $EdgeStruct$, which no longer contains a large register $prev$. Instead, a 3-dimensional array $prev$, external to the $EdgeStruct$ structure, is used. A process that updates some edge $e_{i,j}$ records locally the weight w_{old} and version seq_{old} that it read in $e_{i,j}$. After performing the SC on $e_{i,j}$ (as on line 72 of Dense), the process uses the LL/SC primitive to attempt to write w_{old} and seq_{old} into the $prev$ position they have to be recorded for some other traversing process. Notice that the status LL/SC object ST can also be implemented with smaller registers, by applying the technique that is used in the P-Sim algorithm that is presented in [14]. A detailed description of S-Dense will be presented in the full version of the paper.

5.1 Proof of Correctness Sketch

Due to space constraints, we present a sketch of the correctness argument for our algorithm. Consider an execution α of Dense. Denote by $SC_1^{ST}, SC_2^{ST}, \ldots$ the sequence of successful SC of line 80 on ST in α and by $LL_1^{ST}, LL_2^{ST}, \ldots$ the sequence of matching LL on ST. We first prove that the phases of Dense indeed oscillate between AGREE and APPLY.

▶ **Corollary 1.** *Any SC_k^{ST} such that $k \mod 2 = 1$ changes $ST.phase$ from AGREE to APPLY. Any SC_k^{ST} such that $k \mod 2 = 0$ changes $ST.phase$ from APPLY to AGREE.*

Let op be an instance of some operation, invoked by process p_u, $1 \leq u \leq n$. When line 49 is executed for op, we say that op is *announced*. Let op be an operation that is announced in some configuration C. Let p_l, $1 \leq l \leq n$ where possibly $l \neq u$, be a process which executes a successful SC on ST after C, such that the value written to $BitVector[u]$ at C is copied into $ST.toggles[u]$. If this occurs, we say that op has been *agreed*. Let op be a DynamicTraverse operation that is agreed in some configuration C'. Let p_l, where possibly $l \neq u$, be a process which executes a successful SC on ST after C', such that the value written to $BitVector[u]$ at C is now also copied into $ST.done[u]$. If this occurs, we say that op has been *applied*. We say that an agreed Update operation op, with edge $e_{i,j}$ as parameter, has been applied, if some process p_l successfully executes the SC of line 72 on $e_{i,j}$ with the parameter v of op. We assign the linearization points to operations in the configuration in which they are applied.

We prove that each operation is agreed in its execution interval and use this to prove that it is also applied exactly once during its execution interval, as well as the following lemmas:

▶ **Lemma 2.** *Let OP be any instance of either DynamicTraverse or Update. The linearization point of OP is included in its execution interval.*

Then, we prove that instances of `Read` enclosed in a *d-traversal*, read edge values that are mutually consistent.

▶ **Lemma 3.** *Consider an instance R of* `Read` *with arguments i and j, executed by p_u and let r be the executed by R on line 36. Let DT be the last instance of* `DynamicTraverse` *executed by p_u before R. Then, R returns as the weight for edge $e_{i,j}$ the value v, which is the weight written to $e_{i,j}$ by U, where U is the last instance of* `Update` *with arguments i, j, v, that was linearized before the linearization point of DT.*

We prove then that *d-traversals* have a linearization point inside their execution interval and use all the above to prove the following theorem.

▶ **Theorem 4.** *`Dense` is a wait-free linearizable concurrent graph implementation with $O(k)$ step complexity, where k is the number of active processes.*

6 Discussion

We have introduced a concurrent graph and provided an implementation, which supports wait-free operations, which include updates to the graph edges and the possibility of performing dynamically defined partial traversals. Operations implemented by `Dense` access and affect edges of a graph. Thus, the algorithm is designed with the implicit assumption of a fixed or at least, maximal number of possible vertices out of a specific vertex set. `Dense` operations are oblivious to the values or possible other attributes of those vertices. Indeed, there are many applications that are concerned with the connectivity of a graph only and need only access graph edges. Examples include garbage-collection – where objects are represented by graph nodes, while references to them are represented by graph edges – and graph-based video game navigation – where the edges of a graph represent walkable surfaces between obstacles, represented in turn by graph nodes. Nevertheless, an interesting line of future work is to extend the update and traversal capabilities of `Dense` to also provide information about the state or attributes of the visited vertices. `Dense` takes an irregular data structure and uses a regularized representation of it, in order to provide dynamic traversals. An interesting question concerns whether the helping mechanism employed by `Dense` can be used as a generalized traversal technique. It would be interesting to explore what other irregular or regular data structures (trees, lists, queues, etc) can benefit from it.

Acknowledgements. The authors would further like to thank Prof. Panagiota Fatourou for the useful comments and fruitful discussion that she provided. We thankfully acknowledge the support of the ARISTEIA Action of the Operational Programme Education and Lifelong Learning which is co-funded by the European Social Fund (ESF) and National Resources through the GreenVM project, and the support of the European Commission under the 7th Framework Programs through the EuroServer (FP7-ICT-610456) and HiPEAC3 (FP7-ICT-287759) projects.

References

1 Yehuda Afek, Hagit Attiya, Danny Dolev, Eli Gafni, Michael Merritt, and Nir Shavit. Atomic snapshots of shared memory. *J. ACM*, 40(4):873–890, September 1993. doi:10.1145/153724.153741.

2 Marcos Kawazoe Aguilera, Wojciech M. Golab, and Mehul A. Shah. A practical scalable distributed b-tree. *PVLDB*, 1(1):598–609, 2008.

3 Zeyad Abd Algfoor, Mohd Shahrizal Sunar, and Hoshang Kolivand. A comprehensive study on pathfinding techniques for robotics and video games. *International Journal of Computer Games Technology*, 2015.

4 Hagit Attiya, Rachid Guerraoui, and Eric Ruppert. Partial snapshot objects. In *Proceedings of the 20th Annual Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 336–343, NY, USA, 2008. ACM.

5 Gal Bar-Nissan, Danny Hendler, and Adi Suissa. A dynamic elimination-combining stack algorithm. *CoRR*, abs/1106.6304, 2011.

6 Trevor Brown, Faith Ellen, and Eric Ruppert. A general technique for non-blocking trees. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 329–342, NY, USA, 2014. ACM. doi:10.1145/2555243.2555267.

7 Vadim Bulitko, Yngvi Bjornsson, Nathan R. Sturtevant, and Ramon Lawrence. Real-time heuristic search for pathfinding in video games. In *Artificial Intelligence for Computer Games*, pages 1–30. Springer New York, 2011. doi:10.1007/978-1-4419-8188-2_1.

8 Victor Bushkov, Rachid Guerraoui, and Michal Kapalka. On the liveness of transactional memory. In *Proceedings of the 31st Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 9–18, NY, USA, 2012. ACM.

9 Joseph Carsten, Arturo Rankin, Dave Ferguson, and Anthony Stentz. Global planning on the mars exploration rovers: Software integration and surface testing. *J. Field Robot.*, 26(4):337–357, April 2009. doi:10.1002/rob.v26:4.

10 Bapi Chatterjee, Nhan Nguyen, and Philippas Tsigas. Efficient lock-free binary search trees. In *Proceedings of the 33rd Annual ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 322–331, NY, USA, 2014. ACM. doi:10.1145/2611462.2611500.

11 Guojing Cong, Sreedhar B. Kodali, Sriram Krishnamoorthy, Doug Lea, Vijay A. Saraswat, and Tong Wen. Solving large, irregular graph problems using adaptive work-stealing. In *37th International Conference on Parallel Processing (ICPP)*, pages 536–545, 2008.

12 Faith Ellen, Panagiota Fatourou, Eric Ruppert, and Franck van Breugel. Non-blocking binary search trees. In *Proceedings of the 29th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, pages 131–140, NY, USA, 2010. ACM. doi:10.1145/1835698.1835736.

13 Panagiota Fatourou, Mykhailo Iaremko, Eleni Kanellou, and Eleftherios Kosmas. Algorithmic techniques in stm design. In *Transactional Memory. Foundations, Algorithms, Tools, and Applications*, volume 8913, pages 101–126. Springer, 2015. doi:10.1007/978-3-319-14720-8_5.

14 Panagiota Fatourou and Nikolaos D. Kallimanis. Highly-efficient wait-free synchronization. *Theory of Computing Systems*, pages 1–46, 2013. doi:10.1007/s00224-013-9491-y.

15 Timothy L. Harris. A pragmatic implementation of non-blocking linked-lists. In *Proceedings of the 15th International Conference on Distributed Computing (DISC)*, pages 300–314, London, UK, 2001. Springer-Verlag. URL: http://dl.acm.org/citation.cfm?id=645958.676105.

16 Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Scalable flat-combining based synchronous queues. In *Distributed Computing*, volume 6343, pages 79–93. Springer, 2010.

17 Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *Proceedings of the 16th Annual ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, pages 206–215. ACM, 2004.

18 Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13(1):124–149, January 1991. doi:10.1145/114005.102808.

19 Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15(5):745–770, 1993.

20 Maurice Herlihy and J. Eliot B. Moss. Transactional memory: Architectural support for lock-free data structures. *SIGARCH Comput. Archit. News*, 21(2):289–300, May 1993.

21 Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.

22 Damien Imbs and Michel Raynal. Help when needed, but no more: Efficient read/write partial snapshot. In *Distributed Computing*, volume 5805, pages 142–156. Springer Berlin Heidelberg, 2009.

23 Prasad Jayanti and Srdjan Petrovic. Logarithmic-time single deleter, multiple inserter wait-free queues and stacks. In *Proceedings of the 25th International Conference on Foundations of Software Technology and Theoretical Computer Science (FSTTCS)*, pages 408–419. Springer-Verlag, 2005. doi:10.1007/11590156_33.

24 Frank M. Johannes. Partitioning of vlsi circuits and systems. In *Proceedings of the 33rd Annual Design Automation Conference*, DAC'96, pages 83–87, NY, USA, 1996. ACM.

25 Alex Kogan and Erez Petrank. Wait-free queues with multiple enqueuers and dequeuers. In *Proceedings of the 16th ACM Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 223–234, NY, USA, 2011. ACM. doi:10.1145/1941553.1941585.

26 Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. *SIGPLAN Not.*, 47(8):141–150, February 2012. doi:10.1145/2370036.2145835.

27 Yu-Kwong Kwok and Ishfaq Ahmad. Benchmarking and comparison of the task graph scheduling algorithms. *J. Parallel Distrib. Comput.*, 59(3):381–422, December 1999. doi:10.1006/jpdc.1999.1578.

28 Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the 15th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 267–275, NY, USA, 1996. ACM. doi:10.1145/248052.248106.

29 Donald Nguyen and Keshav Pingali. Synthesizing concurrent schedulers for irregular algorithms. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 333–344, 2011. doi:10.1145/1950365.1950404.

30 Yiannis Nikolakopoulos, Anders Gidenstam, Marina Papatriantafilou, and Philippas Tsigas. A consistency framework for iteration operations in concurrent data structures. In *2015 IEEE International Parallel and Distributed Processing Symposium, IPDPS 2015, Hyderabad, India, May 25-29, 2015*, pages 239–248, 2015. doi:10.1109/IPDPS.2015.84.

31 Erez Petrank and Shahar Timnat. Lock-free data-structure iterators. In *Distributed Computing*, volume 8205, pages 224–238. Springer Berlin Heidelberg, 2013. doi:10.1007/978-3-642-41527-2_16.

32 Aleksandar Prokopec, Nathan G. Bronson, Phil Bagwell, and Martin Odersky. Concurrent tries with efficient non-blocking snapshots. *SIGPLAN Not.*, 47(8):151–160, Feb 2012. doi:10.1145/2370036.2145836.

33 Dimitrios Prountzos, Roman Manevich, and Keshav Pingali. Elixir: A system for synthesizing concurrent graph programs. *SIGPLAN Not.*, 47(10):375–394, October 2012. doi:10.1145/2398857.2384644.

34 William N. Scherer III, Doug Lea, and Michael L. Scott. Scalable synchronous queues. In *Proceedings of the 11th ACM Symposium on Principles and Practice of Parallel Programming (PPOPP)*, NY, USA, 2006. ACM.

35 Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the 14th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pages 204–213, NY, USA, 1995. ACM.

36 Nino Shervashidze, Pascal Schweitzer, Erik Jan van Leeuwen, Kurt Mehlhorn, and Karsten M. Borgwardt. Weisfeiler-lehman graph kernels. *J. Mach. Learn. Res.*, 12:2539–2561, November 2011. URL: `http://dl.acm.org/citation.cfm?id=1953048.2078187`.

37 S. Taylor, J.R. Watts, M.A. Rieffel, and M.E. Palmer. The concurrent graph: basic technology for irregular problems. *Parallel Distributed Technology: Systems Applications, IEEE*, 4(2):15–25, Summer 1996. `doi:10.1109/88.494601`.

38 Shahar Timnat, Anastasia Braginsky, Alex Kogan, and Erez Petrank. Wait-free linked-lists. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP)*, pages 309–310, NY, USA, 2012. ACM. `doi:10.1145/2145816.2145869`.

39 Chung Yung, Jheng-Jyun Syu, and Shiang-Yu Yang. A graph-based algorithm of mostly incremental garbage collection for active object systems. In *International Computer Symposium (ICS)*, pages 988–996, 2010. `doi:10.1109/COMPSYM.2010.5685367`.