Sequential Proximity

Towards Provably Scalable Concurrent Search Algorithms

Karolos Antoniadis(✉), Rachid Guerraoui, Julien Stainer, and Vasileios Trigonakis

École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland
{karolos.antoniadis,rachid.guerraoui,julien.stainer, vasileios.trigonakis}@epfl.ch

Abstract. Establishing the scalability of a concurrent algorithm a priori, before implementing and evaluating it on a concrete multi-core platform, seems difficult, if not impossible. In the context of search data structures however, according to all practical work of the past decade, algorithms that scale share a common characteristic: They all resemble standard sequential implementations for their respective data structure type and strive to minimize the number of synchronization operations.

In this paper, we present *sequential proximity*, a theoretical framework to determine whether a concurrent search algorithm is close to its sequential counterpart. With sequential proximity we take the first step towards a theory of scalability for concurrent search algorithms.

1 Introduction

Concurrent search data structures (CSDSs), such as linked lists and skip lists, are fundamental building blocks of modern software, ranging from operating systems, such as the Linux kernel [15], to key-value stores, such as RocksDB [6]. A vast amount of work has been dedicated to the development of *correct* and *scalable* CSDS algorithms [3–5,7–10,14,17].

To establish the correctness of such algorithms, several formal tools are available. For instance, *linearizability* [12] helps determine the *safety* of CSDS algorithms. Similarly, in terms of *liveness*, we can prove whether a CSDS algorithm is *lock-free* or *wait-free* [11].

In contrast, no formal tool is available for establishing the scalability of a CSDS algorithm, namely that the algorithm delivers better performance when the number of threads accessing the data structure increases. A non-scalable CSDS that resides in an application's critical path eventually becomes a performance bottleneck that needs to be replaced by an alternative design. Ideally, we would like to be able to prove that an algorithm is scalable without the need to evaluate the algorithm on every single workload and multi-core platform.

Defining a formal theory of scalability is an onerous task, since such a theory would need to take into account a multitude of different architectures, diverse set

This work has been supported in part by the European ERC Grant 339539 - AOC.

A. El Abbadi and B. Garbinato (Eds.): NETYS 2017, LNCS 10299, pp. 405–420, 2017.
DOI: 10.1007/978-3-319-59647-1_30

of workloads, etc. In this work, we follow an indirect approach: Instead of formalizing scalability, we create a formal framework that captures when a CSDS is similar to its respective sequential search data structure. Our work is based on the vast amount of prior practical work that points to a single direction for achieving scalability: Strip down synchronization (i.e., every construct that induces coordination of concurrent threads), which is a major impediment to scalability. To achieve minimal synchronization, all existing patterns for designing concurrent data structures do, directly or indirectly, promote concurrent designs that are close to their sequential counterparts: concrete CSDS algorithms [10,13], RCU [17], RLU [16], OPTIK [8], ASCY [4], etc.

Comparing a CSDS and a sequential search data structure in a formal way is challenging (e.g., how to compare the number of stores or where stores are issued between a CSDS and its respective sequential counterpart, etc.) In this paper, we tackle this challenge by introducing *sequential proximity (SP)*, a theoretical framework composed of ten formal properties that can be used to establish whether a CSDS algorithm is *close* to a reference sequential counterpart. SP can be viewed as a first step towards formalizing the scalability of CSDS algorithms.

Sequential Proximity: Overview. Our ten SP properties (Table 1) are defined with respect to the three basic operations of a CSDS: search, insert, and delete, for retrieving, adding, and removing an element from a set, respectively.

SP_{1-4} concern search operations. In a sequential design, search operations (i) are read-only, (ii) do not block, (iii) do not restart, and (iv) do not allocate any memory. SP_{1-4} enforce the exact same behavior as (i)–(iv) for concurrent search operations. SP_{2-5} concern parsing the set before performing an update (i.e., insert or delete). Essentially, the *parse* phase of an update operation traverses the set to find the node(s) to be modified. In a sequential data structure, parsing is identical to searching, hence searching and parsing share SP_{2-4}. SP_5 replaces SP_1 for parsing, to capture the fact that concurrent designs (e.g., [7,9]) might retain some minimal helping strategy in order to "clean-up" the data structure. SP_6 concerns both insertions and deletions. In a sequential design, no writes are issued if the operation is unsuccessful (e.g., a deletion does not find the target element in the set). SP_6 enforces the same behavior for concurrent algorithms: An unsuccessful update cannot perform any stores or atomic operations after parsing. SP_7

Table 1. The ten commandments of SP.

	$SP_{\#}$	Name	search	insert	delete
traversal	SP_1	Read-only	✓		
	SP_2	Non-blocking	✓	✓	✓
	SP_3	No back-step	✓	✓	✓
	SP_4	No allocation	✓	✓	✓
	SP_5	Read-clean		✓	✓
modification	SP_6	Read-only unsuccessful		✓	✓
	SP_7	Conflict restart		✓	✓
	SP_8	Number of stores		✓	✓
	SP_9	Region of stores		✓	✓
	SP_{10}	No allocation			✓

restricts the ability of an update operation to restart due to concurrency. SP_7 does not have any correspondence in sequential algorithms, as the latter never restart. Intuitively, an update in a CSDS can only restart when a concurrent update of another thread modifies the same nodes as the current update. SP_8 and SP_9 restrain the amount of synchronization allowed when modifying the structure during insertions and deletions. We define the maximum number of shared memory stores (or atomic operations) and the locations of these stores in a concurrent design with respect to the sequential counterpart per data structure. Finally, SP_{10} captures the fact that deleting an element from a set should not allocate memory.

A CSDS algorithm is said to be *sequentially proximal* if it satisfies SP_{1-4} for search, SP_{2-9} for insert, and SP_{2-10} for delete operations.

Overall, we believe that SP can be used in guiding the design of scalable CSDS algorithms, detecting whether a CSDS algorithm is likely to scale, and optimizing existing CSDS designs by "fixing" one or more SP properties.

Roadmap. The rest of the paper is organized as follows. In Sect. 2, we recall background notions on CSDSs and describe the machinery we use to formulate the SP properties. We describe the SP properties in Sect. 3. We conclude the paper of SP in Sect. 4. Due to space limitations, we defer the reader to the technical report [1] for the precise definitions of some parts of our vocabulary, proofs of relations between SP properties and classic progress conditions, proofs that two known concurrent linked lists are sequentially proximal, related work, as well as concrete examples of the applicability of SP.

2 Preliminaries

In this section, we define sequential and concurrent search data structures and we introduce the formalism used to define our ten SP properties.

2.1 Search Data Structures

A *search data structure* (SDS) corresponds to a set of elements and operations for retrieving, storing, and removing elements. The main operations of a SDS are the search, insert, and delete operations. In this work, we consider linked lists, hash tables, skip lists, and binary search trees, which are all widely-used SDSs. Queues and stacks are not SDSs as they do not provide search operations.

The insert and delete operations are *update operations* used for inserting and removing elements, respectively. An update operation can be divided into two phases: *parse* and *modify*. For instance, an insertion in a sorted linked list first looks for the position where the element has to be inserted. The actual insertion can then happen during a modify phase. The typical flow of an update operation in a SDS is depicted in Fig. 1. The parse phase takes place first and returns a boolean value which indicates whether it can be followed by a modification. If the

returned value is true (e.g., deleting an element that exists), the modification can be attempted. Otherwise, if the returned value is false (e.g., deleting an element that does not exist), the parse phase did not find a valid state to apply the subsequent modification. After a successful parse phase, the modify phase takes place (which always returns true in sequential SDSs).

Sequential Specification. The sequential specification of a SDS, denoted $Spec_{SDS}$, can be constructed using the notion of a set. At the beginning of a history of $Spec_{SDS}$ the set is empty, thus every search operation returns false. If an insert operation is called and the element is not in the set, the element is inserted into the set and true is returned. Otherwise, the set remains unchanged and false is returned. If a delete operation is called for an element that belongs to the set, the element is removed from the set and true is returned. Otherwise the set remains unchanged and false is returned.

Concurrent Search Data Structures (CSDSs). In CSDSs, the modify phase of update operations can return two values other than true, namely false and restart. These two additional transitions appear as dashed lines in Fig. 1.

On the one hand, a modification can return false either due to concurrency (e.g., the element was concurrently deleted by another process), or because the algorithm enters the modify phase, although the operation cannot be completed. On the other hand, a modification might return restart due to conflicting concurrency (i.e., another process modifies the same vicinity of the structure).

The sequential and concurrent SDSs that we consider in this work are implementations of $Spec_{SDS}$. We assume they have been proven correct in their respective environments (i.e., when used by one process for sequential and by several for CSDSs). We consider that fulfilling $Spec_{SDS}$ in a concurrent context means ensuring *linearizability* [12].

2.2 Language

To describe CSDS algorithms, we consider a formal language [2] that we extend to capture specific characteristics of CSDSs. We present here a quick overview of its classic features and a more detailed description of the additions we introduce to capture the notions needed to define sequential proximity.

Shared Memory Locations. These are the unit of memory, accessible by every process, on which read and write instructions operate atomically.

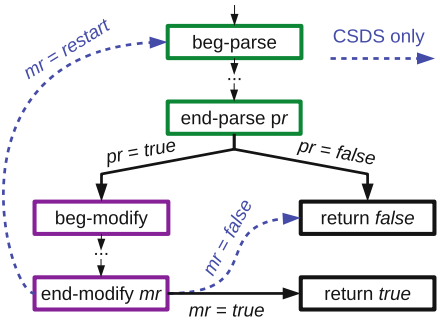

Fig. 1. Flow diagram of an update operation. The transitions in dashed lines are only feasible in concurrent SDSs.

Local and Global Instructions. Each process executes a sequential program (of a Turing-complete language) augmented with instructions to interact with the shared memory. The language uses a standard syntax and semantics for boolean and numerical literals, variables, and expressions. It also features pointers, conditionals expressions, and branching (labels and goto instructions).

Each process maintains a *state* (set of local variables and execution context) and executes elementary *local* or *global* instructions. Shared memory allocations, and any instruction that takes as operand a shared memory location, are considered global instructions. There are six types of global instructions: allocate, read, write, compare-and-swap, try-lock, and unlock. A read(l) instruction retrieves the content of the shared memory location l and a write(v, l) writes the value of the local variable v to shared memory location l.

Compare-and-swap. In one atomic step, a compare-and-swap(l, old, new) instruction reads the content v of the shared memory location l, and, if $v = old$, it writes value new in l. In any case, compare-and-swap returns v.

Try-lock and Unlock. In one atomic step, the try-lock(l) instruction tests if the value v contained in the shared memory location l is true, and, in this case, it writes false in l. In any case, try-lock returns v. The try-lock instruction can be used to implement a traditional blocking lock operation by repeatedly executing try-lock until it returns true. The unlock(l) instruction writes true in l.

Allocate. allocate takes a list of local variables as argument and fills each variable with the address of a newly allocated shared memory location. Note that the use of allocate is closely related to the notion of *node*, defined below, that plays an important role in the definition of the SP properties.

Operations Delimiters. To capture the implementation of CSDSs, additional dummy statements are introduced to delimit the beginning and the end of search, insert, and delete operations. For update operations (i.e., insert and delete), additional statements are used to localize the beginning and the end of the parse and modify phases: beg-parse, end-parse, beg-modify, and end-modify. The statement end-parse returns a boolean indicating if the update is possible (i.e., the target value is not already present in the set for insert operations, or is present for delete operations). The statement end-modify returns true, false, or restart, indicating respectively that the operation succeeded, failed, or has to be restarted. For $op \in \{\text{search}, \text{insert}, \text{delete}\}$ the dummy statement entry op v (resp. exit op b) denotes the beginning (resp. end) of an operation of type op on the data structure (resp. returning a boolean b, indicating success or failure).

States, Transitions, and Executions. A *program state* σ is a tuple $(pc, locals, globals)$ where pc associates to each process the current value of its program counter, $locals$ associates values to the local variables of each process, and $globals$ to shared memory locations. The *transition function* TF associates to a state σ and a process p the program state σ' reached after p executes its next instruction in state σ. A triple (σ, p, σ') s.t. $TF(\sigma, p) = \sigma'$ is called a *transition*.

An *execution* is a sequence of transitions $t_0, t_1, \ldots$ s.t. $\forall i \geq 0, t_i = (\sigma_i, p_{j_i}, \sigma_{i+1})$, where $p_{j_i} \in \{p_0, p_1, \ldots\}$. Furthermore, σ_0 designates the *initial*

state in which each process is about to execute its first instruction and all the local variables and shared memory locations are uninitialized.

Histories. A *history* is a sequence of tuples (p, st) where st is an entry or exit statement and p is a process. To any execution π, we associate history $hs(\pi)$ defined as the subsequence of the transitions of π corresponding to entry and exit statements, labelled by the processes taking them.

Given a history H, we denote by $H|_p$ the history formed by the subsequence of the tuples of H taken by p. Statements $s = (p,$ entry $op\ v)$ and $s' = (p',$ exit $op'\ b)$ of a history H are said *matching* if $p = p'$, $op = op'$, s precedes s' in H, and if there is no $(p,$ exit $op\ b')$ statement in H between s and s'. An entry statement of a history H that has no matching exit in H is said *pending*. A history H is said *sequential* if $H = en_0, ex_0, en_1, ex_1, \ldots$ where for all $i \geq 0$, en_i and ex_i are matching entry and exit statements. A sequential history that does not end with a pending entry statement is said to be a *complete sequential* history. A history H is *well-formed* if for each process p, $H|_p$ is sequential.

Consider any execution π s.t. $hs(\pi)$ is a well-formed history, and t_{en} a transition of π corresponding to an entry statement executed by process p. We define $opTrans(t_{en}, \pi)$ as the subsequence of π formed by the transitions of p from t_{en} to the next transition t_{ex} corresponding to an exit statement by p. If the operation entered in t_{en} is pending in $hs(\pi)|_p$, there is no such transition t_{ex} and $opTrans(t_{en}, \pi)$ is defined as the sequence of transition taken by p in π starting from t_{en}.

Parse-modify Patterns. For an execution π s.t. $hs(\pi)$ is well-formed, let us consider a transition t_{en} taken by process p that corresponds to an entry $op\ v$ statement with $op \in \{$insert, delete$\}$ and let t_{ex} be the matching exit transition. We defer for the moment the case of t_{en} corresponding to a pending entry statement in $hs(\pi)$. We say that the operation entered in t_{en} *follows a parse-modify pattern* if it follows the flow illustrated by Fig. 1. Formally, if we consider $pm(opTrans(t_{en}, \pi))$ the subsequence of transitions of $opTrans(t_{en}, \pi)$ corresponding to beg-parse, end-parse, beg-modify and end-modify statements, then (a) $pm(opTrans(t_{en}, \pi))$ starts with a beg-parse statement, (b) each beg-parse is immediately followed by an end-parse, (c) an end-parse returning true is immediately followed by a beg-modify statement, (d) if an end-parse or end-modify statement returns false, it is the last transition of $pm(opTrans(t_{en}, \pi))$ and t_{ex} returns false, (e) a beg-modify is immediately followed by an end-modify statement, (f) if an end-modify statement returns true, it is the last transition of $pm(opTrans(t_{en}, \pi))$ and t_{ex} returns true, and (g) an end-modify statement returning restart is immediately followed by a beg-parse statement.

If the transition t_{en} is pending in $hs(\pi)|_p$, we consider that $opTrans(t_{en}, \pi)$ follows a parse-modify pattern if π can be extended to an execution in which t_{en} has a corresponding t_{ex} statement and the (now complete) operation entered in t_{en} follows a parse-modify pattern.

Consider an entry transition t_{en} of an execution π s.t. $opTrans(t_{en}, \pi)$ follows a parse-modify pattern. We define the integer $numberOfParsePhases(t_{en}, \pi)$ (resp. $numberOfModifyPhases(t_{en}, \pi)$) as the number of transitions

corresponding to beg-parse (resp. beg-modify) statements in $opTrans(t_{en}, \pi)$. We also define the sequence $parsePhase(t_{en}, \pi, k)$ (resp. $modifyPhase(t_{en}, \pi, k)$), for any k in $1, \ldots, numberOfParsePhases(t_{en}, \pi)$ (resp. $1, \ldots, numberOfModifyPhases(t_{en}, \pi)$), as the subsequence of $opTrans(t_{en}, \pi)$ starting at the k-th beg-parse (resp. beg-modify) statement and ending at the next following end-parse (resp. end-modify) statement (or at the end of $opTrans(t_{en}, \pi)$ if there is no such statement).

Positions of Global Transitions. We say that an execution π s.t. $hs(\pi)$ is well-formed has *no global transition outside operations* if each global transition of π belongs to some $opTrans(t_{en}, \pi)$ with t_{en} an entry transition of π.

Similarly, we state that an execution π has *no global update transition outside parse and modify phases* if, for any entry transition t_{en} of an insert or delete operation, any global transition of $opTrans(t_{en}, \pi)$ belongs to either the set $parsePhase(t_{en}, \pi, k)$ (for some k in $1, \ldots, numberOfParsePhases(t_{en}, \pi)$) or $modifyPhase(t_{en}, \pi, k')$ (for some k' in $1, \ldots, numberOfModifyPhases(t_{en}, \pi)$).

Well-formed Executions. An execution π is *well-formed* if it verifies: (a) $hs(\pi)$ is a well-formed history, (b) transitions never read uninitialized variables, (c) for any transition t_{en} corresponding to an entry *op* v statement with *op* $\in$ {insert, delete}, the operation entered in t_{en} follows a parse-modify pattern, (d) π has no global transition outside operations, and (e) π has no global update transition outside parse and modify phases.

A program *Prog* is said *well-formed* if all the executions it allows are well-formed. The set of all the executions allowed by *Prog* is denoted $[\![Prog]\!]$. The remaining of the paper considers only well-formed programs.

2.3 Nodes and Allocation Sets

Nodes and Shared Memory Management. We assume that a SDS implementation provides the notion of *node* that captures the set of shared memory locations that are allocated and freed/unlinked together. It is assumed that one allocate statement allocates a list of shared memory locations corresponding to exactly one node. For example, in an external tree, a single operation can allocate shared memory locations logically corresponding to an internal node and to a leaf. The SP properties rely on that a separate allocate instruction is used for each of these two nodes.

To capture this relation between nodes and allocate instructions, we define, for any execution π and any transition t_{al} corresponding to an allocate instruction, the set $NodeAlloc(t_{al}, \pi)$ of the memory locations it reserves.

Memory reclamation is orthogonal to designing correct CSDSs and is typically handled by an external garbage collector. For clarity reasons, we do not consider memory reclamation in our model: Once a node is unlinked from the data structure (becomes *unreachable*, see below), the corresponding shared memory area is never reused.

Read and Written Locations. For any execution π and any transition t of π, we denote by $wloc(t)$ (resp. $rloc(t)$) the set that contains the shared memory location written (resp. read) by the instruction corresponding to t. If t corresponds to a local instruction, a global read, or an allocate instruction, then $wloc(t) = \emptyset$. If t corresponds to a write(v, l), try-lock(l), unlock(l), or a compare-and-swap(l, old, new) global instruction, then $wloc(t) = \{l\}$. Similarly, $rloc(t) = \emptyset$ if the instruction executed during t is a local instruction, a global write, or an allocate instruction, while $rloc(t) = \{l\}$ if it is a read(l), try-lock(l), or a compare-and-swap(l, old, new). By an abuse of terminology, we will refer to instructions issued by a transition t s.t. $wloc(t) \neq \emptyset$ as write instructions.

For each transition t_{en} of π that corresponds to a process p executing an entry *op* v statement, we define the set $WrittenLoc(t_{en}, \pi)$ (resp. $ReadLoc(t_{en}, \pi)$) of shared memory locations written (resp. read) during the operation started at t_{en} as follows:

$$WrittenLoc(t_{en}, \pi) = \bigcup_{t \in opTrans(t_{en}, \pi)} wloc(t)$$

$$ReadLoc(t_{en}, \pi) = \bigcup_{t \in opTrans(t_{en}, \pi)} rloc(t).$$

Writing to Nodes Allocated by Others. Consider a well-formed execution π and any entry transition t_{en} corresponding to an entry *op* v statement by a process p. Let S be a subsequence of $opTrans(t_{en}, \pi)$, and let us denote by $w(S)$ the subsequence of transitions of S corresponding to global write instructions. We define $opAlloc(t_{en}, \pi)$ as the set of shared memory locations allocated by p during the operation starting by t_{en}. Formally:

$$opAlloc(t_{en}, \pi) = \bigcup_{t \in al(opTrans(t_{en}, \pi))} NodeAlloc(t, \pi),$$

where $al(opTrans(t_{en}, \pi))$ is the subsequence of $opTrans(t_{en}, \pi)$ transitions that issue allocate instructions.

We now define the set $OtherNodeWrites(S, t_{en}, \pi)$ of the transitions of S writing into shared memory locations that have not been allocated by p during the operation it started at t_{en}. Formally, $OtherNodeWrites(S, t_{en}, \pi)$ is the maximal subset of $w(S)$ such that:

$$opAlloc(t_{en}, \pi) \cap \left(\bigcup_{t \in OtherNodeWrites(S, t_{en}, \pi)} wloc(t) \right) = \emptyset.$$

2.4 Solo Executions, Relative Nodes, and Reachability

Capturing the idea that a CSDS issues stores in a similar region as a respective sequential one is challenging: It is difficult to define what a "similar region" is.

To overcome this challenge, we define the notions of sequential freedom and solo executions and then introduce the concept of relative nodes. We then show how relative nodes can be used to construct sets of read and written nodes. Finally, we define the notion of reachability and the set of nodes that are freed during an operation.

Sequential Freedom. An execution π is in a *steady state*, if there is no entry statement pending in $hs(\pi)$. A program $Prog$ is *sequentially free* if, starting from any steady state, an operation taking steps alone terminates.

Solo Execution. A *solo execution* by a program $Prog$ of a history $S \in Spec_{SDS}$ corresponds to the execution of each operation of S by $Prog$ in a solo (i.e., running the operation alone with no real concurrency) manner. Formally, consider a complete sequential history $S = en_0,\ ex_0,\ en_1,\ ex_1, \ldots, en_n, ex_n$ s.t. $S \in Spec_{SDS}$. Let Σ be the sequence $p_{j_0}, p_{j_1}, ..., p_{j_n}$ of process identifiers that execute operations $en_0, en_1, ..., en_n$, respectively (a process identifier might appear several times). We call *solo execution* of history S by program $Prog$, and denote by $se(S, Prog, \Sigma)$, the execution of $Prog$ in which p_{j_0} executes alone the transitions of the operation entered in en_0 and exited in ex_0, then followed by p_{j_1} executing alone the operation entered in en_1, etc.

Relative Nodes. A *relative node* corresponds to a pair $(a, b) \in \mathbb{N} \times \mathbb{N}$ in an execution π, if there is a transition t_{en} in π that corresponds to an entry statement s.t. this entry statement appears in the a-th position in $hs(\pi)$ and the sequence $al(opTrans(t_{en}, \pi_S))$ contains at least b elements. For example, if there exists a relative node $(5, 2)$ in an execution π, then this node has a "one-to-one" correspondence with the second allocate statement that was executed during the fifth operation.

Using relative nodes, we abstract away from memory locations and instead of comparing writes, we can compare the nodes where those writes are issued to. This abstraction allows us to compare writes (by comparing nodes) between a CSDS and a sequential SDS in order to capture property SP_9. We use relative nodes only on solo executions. We assume that in any solo execution of a given sequential history S, the operations of the CSDS and those of the respective SDS *allocate the same nodes and in the same order.*

Given an execution π, we define $rel(a, b, \pi)$ for $a, b \in \mathbb{N}$ to be a transition t_{al} of π. If $rel(a, b, \pi) = t_{al}$, this means that there is an entry statement in the a-th position of history $hs(\pi)$ that has a corresponding transition t_{en} in π and there are at least b elements in $al(opTrans(t_{en}, \pi))$ issuing an allocate instruction with t_{al} being the b-th such transition. If there exist no such a and b, then $rel(a, b, \pi) = \perp$.

Read and Written Nodes. For defining the read and written nodes of an operation we first define the set S which contains all the relative nodes of an execution π.

$$S = \{(a, b) \in \mathbb{N} \times \mathbb{N} : rel(a, b, \pi) \neq \perp\}$$

We can now define the sets of read and written nodes that contain relative nodes. $ReadNodes(t_{en}, \pi)$ is the set of pairs $(a, b) \in S$ satisfying:

$$ReadLoc(t_{en}, \pi) \cap NodeAlloc(rel(a, b, \pi), \pi) \neq \emptyset.$$

Similarly, $WrittenNodes(t_{en}, \pi)$ is the set of pairs $(a, b) \in S$ that satisfy:

$$WrittenLoc(t_{en}, \pi) \cap NodeAlloc(rel(a, b, \pi), \pi) \neq \emptyset.$$

These sets are used in defining property SP_9 in Sect. 3.

Reachability and the Root Pointer. Consider an execution π and a transition t of π such that, after t, a pointer pt points to a shared memory location l. Since π is well-formed, l was allocated by an allocate statement. Let t_{al} be the corresponding transition in π. l satisfies $l \in NodeAlloc(t_{al}, \pi)$. We define $reachable(pt, 1)_t$ as the set $NodeAlloc(t_{al}, \pi)$. For a set of shared memory locations M, we denote by $pointers(M)$ the locations of M that corresponds to pointers.[1] We define recursively for any $x > 0$:

$$reachable(pt, x+1)_t = \bigcup_{pt' \in pointers(reachable(pt,x)_t)} reachable(pt', 1)_t.$$

Intuitively, $reachable(pt, x)_t$ captures the set of shared memory locations that are reachable from pt by following a path traversing at most x nodes. Those locations are reachable immediately after transition t has been executed in π but before the transition succeeding t in π has been executed. We additionally define $reachable(pt, \infty)_t = \bigcup_{x>0} reachable(pt, x)_t$ the set of all shared memory locations accessible from pt.

We assume that each data structure provides an init operation that is executed before any other operation. The init operation, as the name implies, is used for initializing the data structure. For example, for a linked list, init could allocate the head and tail of the list to simplify the execution of the upcoming operations. We denote with *root* (and call it *root pointer*) any pointer that points to a memory location that was allocated during the first allocate statement of init. For instance, init for linked list has to first allocate the head node, so the root pointer points to this head node.

Reachable and Freed Nodes. In a sequential setting, freed nodes are the ones removed by a delete operation. In order to define freed nodes, we first have to define the nodes that are reachable from a pointer pt. Using *reachable*, for a transition $t \in \pi$ we define $ReachableNodes(pt, \pi)_t$ as the set of pairs $(a, b) \in S$ satisfying:

$$NodeAlloc(rel(a, b, \pi), \pi) \cap reachable(pt, \infty)_t \neq \emptyset,$$

[1] Locations containing pointers could be differentiated from other locations if they contain a pointer type. This could be easily done by for example marking the last bit of the value residing in such a location.

where S is the set of relative nodes defined earlier. $ReachableNodes$ includes the nodes that contain at least one location reachable from pt just after transition t.

For a tuple (t_{en}, t_{ex}) in a sequential history $hs(\pi)$, we define $FreedNodes(t_{en}, \pi) = InitialNodes \setminus FinalNodes$, where $InitialNodes = ReachableNodes(root, \pi)_{t_{en}}$ and $FinalNodes = ReachableNodes(root, \pi)_{t_{ex}}$.

The above definition captures the idea that freed nodes are the nodes that were reachable from a root pointer at the beginning of the operation, but are not anymore reachable at the end. Note that the definition of $FreedNodes$ makes sense only for solo executions and is helpful when restricting the number (SP_8), as well as the region of stores (SP_9).

Logical Deletion. Many CSDSs [9,10] perform deletions in two steps: (i) mark the node to be deleted, and (ii) do the actual deletion (i.e., physical removal).

In the technical report [1], we formally define logical deletions. Additionally, we define when a transition is a *cleaning-up* store, meaning a transition that physically removes a marked node from the data structure. Intuitively, a cleaning-up store is defined as a transition that after it is performed in an execution, makes a reachable node of the data structure to be unreachable (based on *reachable*).

3 Sequential Proximity (SP)

In this section, we define the ten SP properties. The first five properties describe characteristics of traversals: search operations and parse phases. The last five describe modifications due to update operations.

3.1 Traversals

Traversals correspond to search operations or parse phases of update operations. More precisely, for an entry transition t_{en} in execution π, we define $traversals(t_{en}, \pi)$. If t_{en} is a search entry transition (i.e., t_{en} executes an entry *search* v statement), then $traversals(t_{en}, \pi)$ corresponds to $\{opTrans(t_{en}, \pi)\}$. If t_{en} is an update entry transition (i.e., t_{en} executes an entry *op* v statement where *op* $\in$ {insert, delete}), then $traversals(t_{en}, \pi)$ corresponds to $\{parsePhase(t_{en}, \pi, k), 1 \leq k \leq n\}$ where $n = numberOfParsePhases(t_{en}, \pi)$.

SP_1: Read-only Traversal. No global memory is written during traversals.

Definition 1 (SP_1). *A program Prog has op read-only traversals if for each* entry *op transition t_{en} in $\pi \in [\![Prog]\!]$, there is no transition executing a* write *instruction in any sequence of $traversals(t_{en}, \pi)$.*

SP_2: Non-blocking Traversal. Traversals must not block (e.g., do not wait for a lock to be released). To define this property, we first define the notion of a *non-blocking* process. Intuitively, a process is non-blocking if there is a constant

n such that no global memory location is read more than n times. Also, in every n steps that the process takes, at least one global memory location is read.

In detail, we say that a process p is *n steps non-blocking in* $tr(p) = t_1, t_2, \ldots, t_e$, where $tr(p)$ is a contiguous subsequence of $opTrans(t_{en}, \pi)$ with an entry transition t_{en} taken by process p in execution π, if $\exists n \in \mathbb{N}$ s.t.:

- no more than n transitions from $tr(p)$ execute a global read instruction to the same memory location;
- for all $r \in \{1, 2, \ldots, e\}$, consider $k = \lfloor r/n \rfloor$ s.t. $(k+1) \cdot n \leq e$, then there is a transition that issues a global read in the sequence of transitions: $t_{k \cdot n+1}, \ldots, t_{(k+1) \cdot n}$.

Definition 2 ($\mathbf{SP}_2$). *A program Prog has op non-blocking traversals if there exists an $n \in \mathbb{N}$ such that: For every entry op transition t_{en} taken by a process p in execution $\pi \in [\![Prog]\!]$, p is n steps non-blocking in every sequence of $traversals(t_{en}, \pi)$.*

$\mathbf{SP}_3$: No Back-step Traversal. Only forward progress is allowed in traversals: When moving from a node a to a b during traversal, node a is never visited again.

For this property, we first define the notion of no back-steps. More precisely, consider a contiguous subsequence $tr(p)$ of $opTrans(t_{en}, \pi)$ where t_{en} is an entry transition taken by a process p in π. We say that process p has *no back-steps* if, for any pair of transitions $t_r, t_{r'}$ appearing in this order in $tr(p)$ with $rloc(t_r) = rloc(t_{r'}) = \{\ell\}$ and $\ell \in NodeAlloc(t_{al}, \pi)$, every transition t taken between t_r and $t_{r'}$ in $tr(p)$ verifies $rloc(t) \subseteq NodeAlloc(t_{al}, \pi)$.

Definition 3 ($\mathbf{SP}_3$). *A program Prog has op no back-step traversals if for every entry op transition t_{en} taken by a process p in $\pi \in [\![Prog]\!]$, in every sequence $trav$ in $traversals(t_{en}, \pi)$, process p has no back-steps in $trav$.*

$\mathbf{SP}_4$: No allocation Traversal. Traversals do not allocate any memory.

Definition 4 ($\mathbf{SP}_4$). *A program Prog has op no allocation traversals if for every entry op transition t_{en} in $\pi \in [\![Prog]\!]$, there is no transition executing an allocate instruction in any sequence of $traversals(t_{en}, \pi)$.*

$\mathbf{SP}_5$: Read-clean Traversal. Traversals might issue stores only for cleaning-up purposes.

Definition 5 ($\mathbf{SP}_5$). *A program Prog has op read-clean traversals if for every entry op transition t_{en} in $\pi \in [\![Prog]\!]$, if a transition t_w executes a write instruction in a sequence of $traversals(t_{en}, \pi)$, t_w is a cleaning-up store.*

3.2 Modifications

For an update entry transition t_{en} in π, we define $modifications(t_{en}, \pi)$ to be the set of sequences $\{modifyPhase(t_{en}, \pi, k), 1 \leq k \leq n\}$ where $n = numberOfModifyPhases(t_{en}, \pi)$.

SP_6: Read-only Unsuccessful Modification. An unsuccessful operation (e.g., trying to insert an element that is already present) does not issue any write in a solo execution.

Definition 6 (SP_6). *A program Prog has op read-only unsuccessful modifications, if, for any complete sequential history $S \in Spec_{SDS}$ and any sequence of processes P, the solo execution $\pi = se(S, Prog, P)$ verifies that: For every* entry *op transition t_{en} in π that has a matching* exit *op* false *statement in $hs(\pi)$, it is the case that $modifications(t_{en}, \pi) = \emptyset$.*

SP_7: Conflict Restart Modification. The modify phase of an update operation can restart if there is a conflict with a concurrent operation. This type of conflict corresponds to the modification of similar nodes by concurrent operations. To capture when concurrent operations are allowed to conflict and restart, we check if such a conflict exists in the underlying sequential data structure.

We first introduce some auxiliary definitions. Two entry transitions t_{en_0} and t_{en_1} are said *conflict-free* in a solo execution π, if $(WrittenNodes(t_{en_0}, \pi) \cup FreedNodes(t_{en_0}, \pi)) \cap (WrittenNodes(t_{en_1}, \pi) \cup FreedNodes(t_{en_1}, \pi)) = \emptyset$. An entry transition t_{en} is called *restart-free* in an execution π, if $opTrans(t_{en}, \pi)$ does not contain an end-modify transition with a restart result. Given an execution π and two operations op_1 and op_2, we say that an execution π' is an *extension of π by op_1 and op_2*, if π is a prefix of π' followed by the transitions of the operations op_1 and op_2 executed by two processes (possibly concurrently) until their corresponding exit transitions.

Consider two programs $Prog_S$ and $Prog_C$ and $S' = S, en_0, ex_0, en_1, ex_1$ a complete sequential history, where S is a history and for every $i \in \{0, 1\}$, en_i corresponds to an entry statement and ex_i is its matching exit statement. Let us consider the following notations:

- $\pi_S = se(S', Prog_S, P_S)$ and $\pi_C = se(S, Prog_C, P_C)$, for P_S and P_C any sequences of processes;
- t_{en_0} and t_{en_1} the transitions corresponding to the entry statements en_0 and en_1 in π_S.

The triple $t = (S', Prog_S, Prog_C)$ is said to be a *valid restart triple* if t_{en_0} and t_{en_1} are not conflict-free in π_S or if, for any extension $\pi_{C'}$ of π_C by operations en_0 and en_1, the transitions corresponding to the entry statements en_0 and en_1 in $\pi_{C'}$ are restart-free.

Definition 7 (SP_7). *A program $Prog_C$ has valid conflict restart modifications, with respect to a sequential search data structure $Prog_S$, if for all complete sequential histories S with at least four tuples, triple $(S, Prog_S, Prog_C)$ is valid restart triple.*

SP_8: Number of Stores per Modification. SP_8 defines the number of stores allowed per modification. SP_8 depends on a respective sequential SDS and on whether the operations of the concurrent algorithm are blocking or not. The distinction between blocking and non-blocking is made due to the fact that a

Table 2. SP_9: Upper bounds on the number of writes (i.e., stores, lock acquisitions, and CAS operations).

insert	$\lvert CASOps(modi)\rvert + \lvert AcquiredLocks(modi)\rvert \leq MaxOtherNodeWrites(\mathsf{insert})$
	$\lvert OtherNodeWrites(modi,\pi)\rvert \leq MaxOtherNodeWrites(\mathsf{insert})$, for lock-based.
	$\lvert OtherNodeWrites(modi,\pi)\rvert = 0$, for non-blocking.
delete	$\lvert CASOps(modi)\rvert + \lvert AcquiredLocks(modi)\rvert$ $\leq MaxOtherNodeWrites(\mathsf{delete}) + MaxFreedNodes(\mathsf{delete})$
	$\lvert OtherNodeWrites(modi,\pi)\rvert$ $\leq MaxOtherNodeWrites(\mathsf{delete}) + MaxFreedNodes(\mathsf{delete})$, for lock-based.
	$\lvert OtherNodeWrites(modi,\pi)\rvert = 0$, for non-blocking.

lock-based algorithm needs to acquire a lock and then issue its modification store. In contrast a non-blocking algorithm applies its modification simultaneously with a compare-and-swap statement.

Definition 8 (SP_8). *A program $Prog_C$ has a sequential number of stores per modification, with respect to a sequential search data structure $Prog_S$, if the number of stores per modify phase is bounded by the maximum number of sequential writes and freed nodes, as defined in Table 2. Specifically, the upper bounds of Table 2 hold for all modi $\in modifications(t_{en}, \pi)$ where t_{en} is an update* entry *op transition in $\pi \in [\![Prog]\!]$. $CASOps(S)$ corresponds to the set of transitions that execute a* compare-and-swap *instruction in the sequence of transitions S. $AcquiredLocks(S)$ corresponds to the transitions from S that successfully acquired a lock (i.e., transitions that executed a* try-lock *statement that returned* true*). $MaxFreedNodes(typ)$ is defined as the maximum number of freed nodes during the sequential execution of an operation of type typ. $MaxOtherNodeWrites(typ)$ is defined as the maximum number of writes issued during the sequential execution of an operation op of type typ to nodes that were not allocated by operation op. The number of stores are constrained depending on whether the CSDS operation is blocking or not.*

SP_9: Region of Stores per Modification. The following property restricts the nodes that an operation writes during a modification, with respect to a sequential SDS.

We first define the written nodes during all the modify phases of an operation. To do this, we define all the memory locations that were written during all the modify phases:

$$WrittenMLoc(t_{en}\pi) = \bigcup_{t \in modi \,:\, modi \in modifications(t_{en},\pi)} wloc(t)$$

$WrittenMNodes(t_{en}, \pi)$ is the set of pairs $(a, b) \in \mathbb{N} \times \mathbb{N}$ s.t. $rel(a, b, \pi) \neq \perp$ that satisfies:

$$WrittenMLoc(t_{en}, \pi) \cap NodeAlloc(rel(a, b, \pi), \pi) \neq \emptyset.$$

Definition 9 (SP_9). *A program $Prog_C$ has a valid region of stores per modification with respect to a sequential search data structure $Prog_S$ if it writes to similar nodes as $Prog_S$ during modifications. Formally, for every complete sequential history S and any sequence of processes P_C and P_S, consider the solo executions $\pi_C = se(S, Prog_C, P_C)$ and $\pi_S = se(S, Prog_S, P_S)$. Since $hs(\pi_C) = hs(\pi_S)$, for every* entry *transition t_{en} in π_C, there is a corresponding* entry *transition $t_{en'}$ in π_S. SP_9 is satisfied*[2] *if the following holds for every update transition t_{en} in π_C: If t_{en} executes*

- *an* insert *statement, then $WrittenMNodes(t_{en}, \pi_C) = WrittenNodes(t_{en'}, \pi_S)$;*
- *a* delete *statement, then $WrittenMNodes(t_{en}, \pi_C) \subseteq WrittenNodes(t_{en'}, \pi_S) \cup FreedNodes(t_{en'}, \pi_S)$.*

SP_{10}: No Allocation Modification. No memory is allocated during modifications.

Definition 10 (SP_{10}). *A program $Prog$ has op no allocation modifications if for every* entry *op transition t_{en} in $\pi \in [\![Prog]\!]$ there is no transition executing an* allocate *instruction for any sequence in $modifications(t_{en}, \pi)$.*

Definition 11: Sequential Proximity

A concurrent search data structure $Prog_C$ is called ***sequentially proximal*** if it satisfies SP_{1-4} for search, SP_{2-9} for insert, and SP_{2-10} for delete operations.

4 Concluding Remarks

In this paper, we defined sequential proximity (SP), a formalization that captures the closeness of concurrent search data structures (CSDSs) and their sequential counterparts. Based on prior work, we argued that sequentially-proximal algorithms, namely algorithms which follow SP, are scalable. As a result, we claim that SP is the first step towards a formal theory for proving that a CSDS algorithm is likely to be scalable. We believe that from a practitioner's point of view, adherence to the SP properties can lead to scalable implementations and help avoid commonly introduced bottlenecks in CSDSs.

[2] For randomized data structures, such as skip lists [18], we assume that the underlying random number generator produces the exact same sequences of numbers for both $Prog_S$ and $Prog_C$.

References

1. Antoniadis, K., Guerraoui, R., Stainer, J., Trigonakis, V.: Sequential proximity: towards provably scalable concurrent search algorithms. Technical report, EPFL (2017)
2. Attiya, H., Guerraoui, R., Hendler, D., Kuznetsov, P. Michael, M.M., Vechev, M.T.: Laws of order: expensive synchronization in concurrent algorithms cannot be eliminated. In: POPL (2011)
3. Bronson, N.G., Casper, J., Chafi, H., Olukotun, K.: A Practical Concurrent Binary Search Tree. In: PPopp (2010)
4. David, T., Guerraoui, R., Trigonakis, V., Concurrency, A.: The secret to scaling concurrent search data structures. In: ASPLOS (2015)
5. Ellen, F., Fatourou, P., Ruppert, E., van Breugel, F.: Non-blocking binary search trees. In: PODC (2010)
6. Facebook: RocksDB. http://rocksdb.org
7. Fraser, K.: Practical lock-freedom. Ph.D. thesis, University of Cambridge (2004)
8. Guerraoui, R., Trigonakis, V.: Optimistic concurrency with OPTIK. In: PPopp (2016)
9. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: Welch, J. (ed.) DISC 2001. LNCS, vol. 2180, pp. 300–314. Springer, Heidelberg (2001). doi:10.1007/3-540-45414-4_21
10. Heller, S., Herlihy, M., Luchangco, V., Moir, M., Scherer, W.N., Shavit, N.: A lazy concurrent list-based set algorithm. In: Anderson, J.H., Prencipe, G., Wattenhofer, R. (eds.) OPODIS 2005. LNCS, vol. 3974, pp. 3–16. Springer, Heidelberg (2006). doi:10.1007/11795490_3
11. Herlihy, M.: Wait-free synchronization. In: TOPLAS (1991)
12. Herlihy, M., Wing, J.: Linearizability: a correctness condition for concurrent objects. In: TOPLAS (1990)
13. Herlihy, M., Lev, Y., Luchangco, V., Shavit, N.: A simple optimistic skiplist algorithm. In: Prencipe, G., Zaks, S. (eds.) SIROCCO 2007. LNCS, vol. 4474, pp. 124–138. Springer, Heidelberg (2007). doi:10.1007/978-3-540-72951-8_11
14. Howley, S.V., Jones, J.: A non-blocking internal binary search tree. In: SPAA (2012)
15. Linux Kernel: Linux Kernel. https://www.kernel.org
16. Matveev, A., Shavit, N., Felber, P., Marlier, P.: Read-log-update: a lightweight synchronization mechanism for concurrent programming. In: SOSP (2015)
17. McKenney, P.E., Slingwine, J.D.: Read-copy update: using execution history to solve concurrency problems. In: PDCS (1998)
18. Pugh, W., Lists, S.: A probabilistic alternative to balanced trees. In: CACM (1990)