Contention in Structured Concurrency: Provably Efficient Dynamic Non-Zero Indicators for Nested Parallelism

Umut A. Acar[*†] Naama Ben-David[*] Mike Rainey[†]
[*]Carnegie Mellon University, USA [†]Inria, France
umut@cs.cmu.edu, nbendavi@cs.cmu.edu, mike.rainey@inria.fr

Abstract

Over the past two decades, many concurrent data structures have been designed and implemented. Nearly all such work analyzes concurrent data structures empirically, omitting asymptotic bounds on their efficiency, partly because of the complexity of the analysis needed, and partly because of the difficulty of obtaining relevant asymptotic bounds: when the analysis takes into account important practical factors, such as contention, it is difficult or even impossible to prove desirable bounds.

In this paper, we show that considering structured concurrency or relaxed concurrency models can enable establishing strong bounds, also for contention. To this end, we first present a dynamic relaxed counter data structure that indicates the non-zero status of the counter. Our data structure extends a recently proposed data structure, called SNZI, allowing our structure to grow dynamically in response to the increasing degree of concurrency in the system.

Using the dynamic SNZI data structure, we then present a concurrent data structure for series-parallel directed acyclic graphs (sp-dags), a key data structure widely used in the implementation of modern parallel programming languages. The key component of sp-dags is an *in-counter* data structure that is an instance of our dynamic SNZI. We analyze the efficiency of our concurrent sp-dags and in-counter data structures under nested-parallel computing paradigm. This paradigm offers a structured model for concurrency. Under this model, we prove that our data structures require amortized $O(1)$ shared memory steps, including contention. We present an implementation and an experimental evaluation that suggests that the sp-dags data structure is practical and can perform well in practice.

1. Introduction

Non-blocking data structures are commonly used in practical concurrent systems because such data structures allow multiple processes to access the structure concurrently while guaranteeing that the whole system will make progress. As discussed by the survey of Moir and Shavit (38), many non-blocking data structures have been designed and implemented for various problems, such as shared counters (22; 42), non-zero indicator counters (14), stacks (45), queues (37), doubly ended queues (7), and binary search trees (9; 13).

The theoretical efficiency of non-blocking data structures, however, has received relatively little attention. One reason for this lack of attention is that in a non-blocking data structure, some processes may get stuck, leading to seemingly infinite execution times. Another reason is that the common shared-memory models, under which concurrent data structures are analyzed, usually do not account for an important practical concern: the effects of memory contention. These effects can be significant especially as the degree of concurrency increases (12; 38; 43).

We are interested in the design and analysis of provably efficient non-blocking data structures in realistic models of concurrency that account for the cost of contention. This goal represents a major challenge, because contention often factors into the running time as a linear additive factor (e.g., (13; 18; 16; 39)). A series of papers have also established linear-time lower bounds (27; 30; 16) when contention is taken into account by using models that allow only one process to access a memory object (11; 12; 5). For example, Fich, Hendler, and Shavit (16) show that, when accounting for contention, a number of concurrent data structures, including counters, stacks, and queues, require $\Omega(n)$ time, where n is the number of processes/threads in the system, a.k.a., the *degree* of concurrency.

In this paper, we show that a relaxation of the general concurrency model can unlock the design and analysis of provably efficient concurrent data structures, accounting also for contention. As a relaxation, we consider a structured concurrent programming paradigm, nested parallelism, where concurrency is created under the control of programming primitives, such as fork-join and async-finish. Nested

parallelism is broadly available in a number of modern programming languages and language extensions, such as OpenMP, Cilk (19), Fork/Join Java (34), Habanero Java (28), TPL (35), TBB (29), X10 (10), parallel ML (17), and parallel Haskell (32). In these systems, nested-parallel programs can be expressed by using a primitives such as **fork–join** (a.k.a., **spawn–sync**), and **async–finish**. When executed, a nested parallel program can create many threads, leading to a high degree of concurrency, but in a structured fashion: threads may be created by other threads via a **async** (or **fork**) operation and can also terminate and synchronize at known **join** (or **finish**) points. The **fork–join** primitives allow only two threads to synchronize at a **join**. In contrast, with **async–finish** primitives, any number of threads can terminate and synchronize at a single **finish** point. This structured nature of thread creation, termination, and synchronization is the key difference between nested parallelism and general concurrency.

Our starting point is prior work on relaxed concurrent counters, also called *indicators*, that indicate whether a counter is positive or not. Indicators contrast with exact counters as, for example, those considered by Herlihy, Shavit, and Waarts (27). In prior work, Ellen et al. (14) present the *SNZI* (read "snazzy"), short for Scalable Non-Zero Indicator, data structure, prove it to be linearizable, and evaluate it empirically. They do not, however, provide any analytical upper bounds. Indeed, SNZI data structure could experience contention as multiple operations climb up the SNZI tree. In this paper, we present an extension to the SNZI data structure to allow it to grow dynamically at run time, perhaps in response to increasing degree of concurrency and thus increasing potential for contention (Section 2).

We then consider an application of our dynamic SNZI data structure to nested parallel computations, specifically for representing series-parallel directed acyclic graphs or *sp-dags* with unbounded in-degree vertices, which arise for example with parallel loops and **async–finish** primitives. Variants of dag data structures are used broadly in the implementation of modern parallel programming systems (19; 34; 28; 35; 29; 10; 17; 32). These languages and systems typically represent the computation as a dag where each vertex corresponds to a fine-grained thread and schedule the vertices of the dag over the processors to minimize completion time. For efficiency reasons, it is important for the dag data structure to be non-blocking and low-contention, because a nested parallel program can create a dynamically varying set of fine-grained threads, consisting of anywhere from several threads to millions or more, depending on the input size.

The crux of the sp-dags problem is determining when a vertex in the dag becomes *ready*, i.e., all of its dependencies have been executed. Such readiness detection requires a concurrent dependency counter, which we call *in-counter*, that counts the number of incoming dependencies for each thread. When a dependency between a thread u and v is created, the in-counter of v is incremented; when u terminates its in-counter is decremented; when its in-counter reaches zero,vertex v becomes ready and can be executed. We show that in-counters can be implemented efficiently by using our dynamic SNZI data structure (Section 3). We prove that by managing carefully the growth of our dynamic SNZI data structure, we can guarantee that all operations require amortized $O(1)$ time, including also contention (Section 4).

Finally, we present an implementation and perform an experimental evaluation in comparison to prior work (Section 5). Our results offer empirical evidence for the practicality of our proposed techniques, showing that our approach can perform well in practice.

Placed in the context of previously proven lower bounds for non-blocking data structures, the key to our result is the structured concurrency model that we consider. Our contributions are as follows.

- Dynamic SNZI: an extension to the original SNZI data structure that allows it to grow varying numbers of threads.
- Non-blocking low-contention sp-dags: a non-blocking data structure for series-parallel dags that can be used to represent nested-parallel computations provably efficiently and with low contention.
- Analysis: proofs that our series-parallel dag data structure experiences low contention under a nested parallel programming model.
- Evaluation: implementation and empirical evaluation that shows that our data structure is practical and that it can perform well in practice.

Due to space restrictions, we present some proof details and more experiments in the full version of the paper, which is available as a technical report (1).

1.1 Model

We consider a standard model of asynchronous shared memory where threads can communicate through taking atomic read-modify-write steps such as read, write and compare-and-swap (CAS). Threads can use these primitives to implement high-level *operations*. We define the *contention* experienced by an operation as the maximum number of non-trivial, shared memory steps that concurrently access the same memory location over all executions. A *non-trivial step* is one that might change the value of the memory location to which it is applied, and a *non-trivial operation* is an operation that takes at least one non-trivial step.

Our definition of contention is similar to *stalls* as defined by Fich et al. in (16). Their definition, inspired by Dwork et al.'s work on contention models (12), considers a model in which each non-trivial memory step on a memory location must operate in isolation, and the rest of the operations stall.

For our bounds, we consider computations that are structured in a nested-parallel fashion, where threads execute independently in parallel or serially and synchronize at their termination points. A series-parallel computation starts with a single root thread and constructs, as it executes, a series-parallel directed acyclic graph (sp-dag) of vertices where each vertex represents a thread of control, and edges represent dependencies between threads.

2. Dynamic SNZI

Concurrent counters are an important data structure used in many algorithms but can be challenging to implement efficiently when accounting for contention, as shown by the linear (in the degree of concurrency) lower bound of Fich et al (16). Observing that the full strength of a counter is unnecessary in many applications, Ellen et al proposed *non-zero indicators* as relaxed counters that allow querying not the exact value of the counter but its sign or *non-zero status* (14). Their non-zero indicator data structure, called SNZI, achieves low-contention by using a tree of *SNZI nodes*, each of which can be used to increment or decrement the counter. Determining the non-zero status of the counter only requires accessing the root of the tree. The tree is updated by increment and decrement operations. These operations are filtered on the way up to the root so that few updates reach the root.

The SNZI data structure is flexible. Each SNZI node interacts with its parent and its children, and will retain its correctness regardless of how many children it has. Thus, a SNZI tree can have any shape but it can be difficult to know the optimal shape of the tree a priori. In this section, we present a simple extension to SNZI that allows the tree to be grown dynamically, and show that our extension preserves correctness. In Section 3, we demonstrate how to use dynamic SNZI in a specific application—namely, to keep track of dependencies in an sp-dag efficiently.

Figure 1 illustrates the interface for the SNZI data structure and pseudo-code for the SNZI node **struct** `snzi_node`, the basic building block of the data structure from (14). A SNZI node contains a counter c, a version number v, an array of children (two in our example), and a parent. The value at the counter indicates the *surplus* of arrivals with respect to the departures. The `arrive` and `depart` operations increment and decrement the value of the counter at the specified SNZI node a respectively, and `query`, which must be called at the root of the tree, indicates whether the number of `arrive` operations in the whole tree is greater than number of `depart` operations since creation of the tree.

To ensure efficiency and low contention, the SNZI implementation carefully controls the propagation of an arrival or departure at a node up the tree. The basic idea is to propagate a change to a node's parent only if the surplus at that node flips from zero to positive or vice versa. More specifically, an `arrive` at node a is called on a's parent if and only if a had surplus 0 at the beginning of the operation. Similarly, a `depart` at a node a is recursively called on a's parent if and only if a's surplus became 0 due to this `depart`. We say that node *a has surplus due to its child* if there was an `arrive` operation on a that started in a's child's subtree.

```
1  type snzi_node =
2  struct {
3    (* counter *)
4    c : ℕ ∪ {½}; initially 0
5    (* version number *)
6    v : ℕ; initially 0
7    children : array of 2 snzi nodes; initially [null, null]
8    parent : snzi_node
9  }
10
11 void fun snzi_arrive (snzi_node a) = ...
12 void fun snzi_depart (snzi_node a) = ...
13 bool fun snzi_query (snzi_node a) = ...
```

Figure 1: Partial pseudocode for the SNZI data structure; full description can be found in original SNZI paper (14).

The correctness and linearizability of the SNZI data structure as proven relies on two important invariants (14): (1) a node has surplus due to its child if and only if its child has surplus, and (2) surplus due to a child is never negative. Together, these properties guarantee that the root has surplus if and only if there have been more `arrive` than `depart` operations on the tree.

The SNZI data structure does not specify the shape of the tree that should be used, instead allowing the application to determine its structure. For example, if the counter is being used by p threads concurrently, then we may use a perfectly balanced tree with p nodes, each of which is assigned to a thread to perform its `arrive` and `depart` operations. This approach, however, only supports a *static* SNZI tree, i.e., unchanging over time. While a static SNZI tree may be sufficient for some computations, where a small numbers of coarse-grained threads are created, it is not sufficient for nested parallel computations, where the number of fine-grained threads vary dynamically over a wide range from just a few to very large numbers, such as millions or even billions, depending on the input size. The problem is that, with a fixed-sized tree, it is impossible to ensure low contention when the number of threads increase because many threads would have to share the same SNZI node. Conversely, it is impossible to ensure efficiency when the number of threads is small, because there may be many more nodes than necessary.

To support re-sizing the tree dynamically, we extend the SNZI data structure with a simple operation called `grow`, whose code is shown in Figure 2. The operation takes a SNZI node as argument, determines whether to extend the node (if the node has no children), and returns the (possibly newly created) children of the node. If the node already

```
1  (snzi_node, snzi_node)
2  fun grow (a: snzi_node, p: probability) =
3    heads ← flip(p) (* flip a p−biased coin. *)
4    if heads then
5      left ← new snzi_node(0)
6      right ← new snzi_node(0)
7      CAS(a.children, null, [left, right])
8    children ← a.children
9    if children = null then
10     return (a, a)
11   return children
```

Figure 2: Pseudo-code for the probabilistic SNZI `grow`.

has children, pointers to them are returned, but they are left unchanged. If the node does not have children, then they are locally allocated, and then the process tries to atomically link them to the tree. They are initialized with surplus 0 so that their addition to the tree does not affect the surplus of their parent. The `grow` operation also takes a probability p, which dictates how likely it is to create new children for the node. The operation only attempts to create new children if the node does not already have children and it flipped heads in a coin toss with probability p for heads. The idea is that this probabilistic growing allows an application to control the rate at which a SNZI tree grows without changing the protocol for some of the threads using it. Such probabilistic growing is useful when one wants a balance between low contention and frequent memory allocation. The right probability threshold to use, however, may depend not only on the application, but also on the system.

If node a does not have any children at the end of a `grow` operation, we have the operation return two pointers to a itself. This return value is convenient for the in-counter application that we present in the rest of the paper. Other applications using dynamic SNZI may return anything else in that case.

The `grow` operation may be called at any time on any SNZI node. To keep the flexible structure of the SNZI data structure, we do not specify when to use the `grow` operation on the tree, instead leaving it to the application to specify how to do so. For example, in Section 3.3, we show how to use the `grow` operation to implement a low-contention dependency counter for dags. It is easy to see that the operation does not break the linearizability of the SNZI structure; it is completely independent from the *count*, *version number*, and *parent* fields of nodes already in the tree, which are the only fields that affect the correctness and linearizability.

We note a key property of the `grow` operation. We would like to ensure that, given a probability p, when `grow` is called on a childless node, only $1/p$ such calls will return no children in expectation, regardless of the timing of the calls. That is, even if all calls to `grow` are concurrent, we want only $1/p$ of them to return no children. This property is ensured by having the coin flip happen *before* reading the value of the *children* array to be returned. This guarantees that any adversary that does not know the result of local coin flips cannot cause more than $1/p$ calls to return no children in expectation.

Dynamically shrinking the SNZI tree is more difficult, because we need to be sure that no operation could access a deallocated node. In general, one may easily and safely delete a SNZI node from the tree if the following two conditions hold: (1) its surplus is 0, and (2) no thread other than the one deleting the node can reach it. The second condition is much trickier to ensure. In Section 4 we show how to deallocate nodes in our dependency counter data structure.

3. Series-Parallel Dags

A nested-parallel program can be represented as a series-parallel dag, or an sp-dag, of threads, where vertices are pieces of computation, and edges represent dependencies between them. In fact, to execute such a program, modern programming languages construct an sp-dag and schedule it on parallel hardware. We present a provably low-contention sp-dag data structure that can be used to execute nested-parallel programs. After defining sp-dags, we present in Section 3.1 a data structure for sp-dags by assuming an *in-counter* data structure that enables keeping track of the dependencies of a dag vertex (thread). We then present our data structure for in-counters in Section 3.3.

A *serial-parallel dag*, or an *sp-dag* for short, is a directed-acyclic graph (dag) that has a unique *source* vertex, which has indegree 0, and a unique *terminal* vertex, which has out-degree 0 and that can be defined iteratively. In the base case, an sp-dag consists of a source vertex u, a terminal vertex v, and the edge (u, v). In the iterative case, an sp-dag $G = (V, E)$ with source s and terminal t is defined by serial or parallel composition of two disjoint sp-dags G_1 and G_2, where $G_1 = (V_1, E_1)$ with source s_1 and terminal t_1 and $G_2 = (V_2, E_2)$ with source s_2 and terminal t_2, and $V_1 \cap V_2 = \emptyset$:

- **serial composition:** $s = s_1$, $t = t_2$, $V = V_1 \cup V_2$, $E = E_1 \cup E_2 \cup (t_1, s_2)$.
- **parallel composition:** $s, t \notin (V_1 \cup V_2)$, $V = V_1 \cup V_2 \cup \{s, t\}$, and $E = E_1 \cup E_2 \cup \{(s, s_1), (s, s_2), (t_1, t), (t_2, t)\}$.

For any vertex v in an sp-dag, there is a path from s to t that passes through v. We thus define the *finish vertex* of v as the vertex u which is the closest proper descendant of v such that every path from v to the terminal t passes through u.

3.1 The sp-dag data structure

Figure 3 shows our data structure for sp-dags. We represent an sp-dag as a graph G consisting of a set of vertices V and a set of edges E. The data structure assigns to each vertex of the dag an in-counter data structure, which counts the number of (unsatisfied) dependencies of the vertex. As the

code for a dag vertex executes, it may dynamically insert and delete dependencies by using several *handles*. These handles can be thought of, in the abstract, as in-counters, but they are implemented as pointers to specific parts of an in-counter data structure.

More precisely, a `vertex` consists of

- a query handle (`query`), an increment handle (`inc`), and a decrement handle (`dec`),
- a flag `first_dec` indicating whether the first or the second decrement handle should be used,
- a flag `dead` indicating whether vertex has been executed,
- the finish vertex `fin` of the vertex, and
- a `body`, which is a piece of code that will be run when the scheduler executes the vertex.

The sp-dag uses an in-counter data structure `Incounter`, whose implementation is described in the Section 3.3. The in-counter data structure supplies the following operations:

- `make`: takes an integer n, creates an in-counter with n as the initial count and returns a handle to the in-counter;
- `increment`: takes a vertex v, increments its in-counter, and returns two decrement and two increment handles;
- `decrement`: takes a vertex v and decrements its in-counter;
- `is_zero`: takes a vertex v and returns true iff that in-counter of v is zero.

Inspired by recent work on formulating a calculus for expressing a broad range of parallel computations (3), our dag data structure provides several operations to construct and schedule dags dynamically at run time. The operation `new_vertex` creates a vertex. It takes as arguments a finish vertex u, an increment handle i, a decrement handle d, and a number n indicating the number of dependencies it starts with. It allocates a fresh vertex, sets its `dead` and `first_dec` flags to `false`, and sets the `body` for the vertex to be a dummy placeholder. It then creates an in-counter and a handle to it. The operation returns the new vertex along with the handle to its in-counter. The handle becomes the query handle. The operation `make` creates an sp-dag consisting of a root u and its finish vertex z and returns the root u. As can be seen in the function `make`, the increment and decrement handles of a vertex v always belong to the in-counter of v's finish vertex.

A vertex u and its finish vertex z can be thought as a computation in which u executes the code specified by its `body` and then returns to z. This constraint implies that z serially depends on u because z can only be executed after u. As it executes, a vertex u can use the functions `chain` and **spawn** to "nest" another sequential or parallel computation (respectively) within the current computation.

The `chain` operation, which corresponds to serial composition of sp-dags, nests a sequential computation within the current computation. When a vertex u calls `chain`, the call creates two vertices v and w such that w serially depends on v.

```
module Incounter = ... (* As defined in Figure 5 *)
module Dag =
G = (V, E)
type handle = Incounter.handle
struct {
  handle query, inc, dec[2];
  boolean first_dec, dead;
  vertex fin;
  (void → void) body;
} vertex;

(vertex, handle)
fun new_vertex (vertex u, handle i, handle d, int n) =
  V ← V ∪ {v}, v ∉ V
  (v.firstDec, v.dead) ← (false, false)
  v.body ← { return }
  v.query ← Incounter.make (n)
  (v.fin, v.inc, v.dec)  ←  (u, i, d)
  return (v, v.query)

(vertex,vertex)
fun make ()
  V ← {v}
  E ← ∅
  (v.firstDec, v.dead) ← (false, false)
  v.body ← { return }
  v.query ← Incounter.make (1)
  u ← new_vertex (u, v.query, v.query, 0)
  return (u, v)

(vertex, vertex)
fun chain (vertex u) =
  (w, h) ← new_vertex (u.fin, u.inc, u.dec, 1)
  (v, h) ← new_vertex (w, h, [h, h], ∅)
  E ← E ∪ {(u, v)}
  u.dead ← true
  return (v, w)

(vertex, vertex)
fun spawn (vertex u) =
  (d, i, j) ← Incounter.increment (u.fin)
  (v, _) ← new_vertex (u.fin,  i, d, ∅)
  (w, _) ← new_vertex (u.fin, j, d, ∅)
  E ← E ∪ {(u, v), (u, w)}
  u.dead ← true
  return (v, w)

void
fun signal (u: vertex) =
  Incounter.decrement (u.fin)
  E ← E ∪ {(u,u.fin)}
```

Figure 3: The pseudo-code for our sp-dag data structure.

Furthermore, v serially depends on u, and z serially depends on w. After calling `chain`, u terminates and thus dies. The initial in-counters of v and w are set to 0 and 1 respectively (to indicate that v is ready for execution, but w is waiting on

one other vertex). The edge (u, v) is inserted to the dag to indicate a satisfied dependency between u and v.

The **spawn** operation, which corresponds to parallel composition of sp-dags, nests a parallel computation with the current computation. When a vertex u calls **spawn**, the call creates two vertices, v and w, which depend serially on u, but are themselves parallel. The operation increments the in-counter of u's finish node, to indicate a serial dependency between u's finish vertex and the new vertices v and w. Even though two new vertices are created, the in-counter is incremented once, because one of the vertices can be thought of as a continuation of u. The `increment` operation returns two new increment handles i and j and a decrement-handle pair d. The **spawn** operation then creates the two vertices v and w using the two increment handles, one for each, and the decrement-handle pair d as a shared argument. Assigning each of v and w their own separate increment handles enables controlling contention at the in-counter of u's finish node. As described in Section 3.3, sharing of the decrement handles by the new vertices (v, w) is critical to the efficiency. As with the `chain` operation, **spawn** must be the last operation performed by u. It therefore completes by creating the edges from u to v and w, and marking u dead.

The operation `signal` indicates the completion of its argument vertex u by decrementing the in-counter for its finish vertex v, and inserting the edge (u, v).

In terms of efficiency, apart from calls to `Incounter`, the operations of sp-dags involve several simple, constant time, operations. The asymptotic complexity of the sp-dags is thus determined by the complexity of the in-counter data structure.

3.2 An Example

Although simple, the sp-dag data structure can be used to create any nested-parallel computation. As an example, we consider the classic parallel Fibonacci program, which computes n^{th} Fibonacci number by recursively and in parallel computing the $(n-1)^{th}$ and $(n-2)^{nd}$ numbers. Figure 4 illustrates the code for this example based on the sp-dag data structure and a `Scheduler` module that implements a standard parallel scheduler, such as work stealing (8; 2). The function `fib_main` initializes the system by calling the function `Scheduler.initialize` and then creates the root and the final vertex of the dag. It then sets the body of the root to run `fib`, passing the arguments n and *result*. The vertices *root* and *final* are then given to the scheduler to be executed. The scheduler returns when the computation is finished, which leads to the return of the result.

The function `fib` implements the parallel Fibonacci function. The function start by creating a chain (u, v), passing `this_vertex`, which is the variable indicating the currently executing vertex, as the parent to the root of the chain. The body of u is set to spawn the two recursive calls and the

```
fib (n: int, dest: int ref) =
  if n <= 1 then
    dest ← n
  else
    res1 = alloc ()
    res2 = alloc ()
    (u, v) ← Dag.chain (this_vertex)
    u.body ← lambda (). {
                (w1, w2) = Dag.spawn (this_vertex)
                w1.body ← lambda ().fib (n − 1, res1)
                w2.body ← lambda ().fib (n − 2, res2)
                Scheduler.add (w1, w2)
              }
    v.body ← lambda (). ∗dest ← ∗res1 + ∗res2
    Scheduler.add (u, v)

fib_main (n: int) =
   int result;
   Scheduler.initialize ()
   (root, final) ← Dag.make ()
   root.body ← lambda ().fib (d, &result)
   Scheduler.add (root, final)
   return result
```

Figure 4: Fibonnacci example. The parallel recursive function at the top and the main function, where execution starts, at the bottom.

body of v is set to compute the final result by summing the return values of the recursive calls. Note that when spawning vertices the body of u uses `this_vertex` as a parent for the vertices created. Since the body is executed when u is executed, at the point of execution `this_vertex` is equal to u.

3.3 Incounters

We present the in-counter: a time and space efficient low-contention data structure for keeping track of pending dependencies in sp-dags. When a new dependency for vertex v is created in the dag, its in-counter is incremented. When a dependency vertex in the dag terminates, v's in-counter is decremented.

Our goal is to ensure that the increment and decrement operations are quick; that is, that they access few memory locations and encounter little contention. These goals could seem contradictory at first—if operations access few memory locations then they would conflict often. Our in-counter data structure circumvents this issue by ensuring that each operation accesses few memory locations that are mostly *disjoint* from those of others.

The in-counter is fundamentally a dynamic SNZI tree. To ensure disjointness of memory accesses, the data structure assigns different *handles* to different dag vertices. These handles are pointers to SNZI nodes within the in-counter, dictating where in the tree an operation should begin. Thus,

```
1  module Incounter =
2  type handle = snzi_node (* A handle is a snzi node *)
3
4  (* Growth probability: architecture specific constant. *)
5  p  ← ...
6
7  (* Make a new counter with surplus n. *)
8  handle
9  fun make (n) = snzi_make (n)
10
11 (* Auxiliary function: select decrement handle. *)
12 handle
13 fun claim_dec (vertex u) =
14   if CAS (u.firstDec, false, true) then u.dec[0]
15   else u.dec[1]
16
17 (* Increment u's target dependency counter. *)
18 (handle[2], handle, handle)
19 fun increment (vertex u) =
20   (a, b) ← grow (u.inc, p)
21   (i1,i2) ← (a, b)
22   if u is a left child then  d2 ← a
23   else  d2 ← b
24   snzi_arrive (d2)
25   d1 ← claim_dec (u)
26   return [[d1,d2],i1,i2]
27
28 (* Decrement u's counter. *)
29 void
30 fun decrement (vertex u) =
31    d ← claim_dec (u)
32    snzi_depart (d)
33
34 (* Check that u's counter is zero. *)
35 boolean
36 fun is_zero (u) = return snzi_isZero(u.query)
```

Figure 5: The pseudo-code for the in-counter data structure.

if two threads have different handles, their operations will access different memory locations.

The in-counter data structure ensures the invariant that only the leaves have a surplus of zero. This allow us to exploit an important property of SNZI: operations complete when they visit a node with positive surplus, effectively stopping propagation of the change up the tree. Specifically, an increment that starts at a leaf completes quickly when it visits the parent. To maintain this invariant, we have to be careful about which SNZI nodes are decremented.

Figure 5 shows the pseudo-code for the in-counter data structure. The in-counter's interface is similar to the original SNZI data structure: the operation `make` creates an instance of the data structure and returns a handle to it. The operation `increment` is meant to be used when a dag vertex spawns. It takes in a dag vertex, and uses its increment handle to increment the SNZI node's surplus. Before doing so, it calls the `grow` operation. Intuitively, this growth request notifies the tree of possible contention in the future and gives it a chance to grow to accommodate a higher load. The `increment` operation returns handles to other nodes in the SNZI tree, that is, two decrement handles and two increment handles, indicating where the newly spawned children of the dag vertex should perform their increment or decrement. The operation `decrement` is meant to be used when a dag vertex signals the end of its computation. It takes in dag vertex, and uses its decrement handle to decrement the surplus of a previously incremented node.

Since the `increment` operation is the only thing that can cause growth in the SNZI tree, the structure of the tree is determined by the operations performed on the in-counter structure. The tree is therefore not necessarily balanced. However, as we establish in Section 4, the operations' running time does not depend on the depth of the tree. In fact, we show that the operations complete in constant amortized time and also lead to constant amortized contention.

To understand the implementation, it is helpful to consider the way the sp-dag structure uses the in-counter operations. The `increment` operation is called by a dag vertex when the vertex calls the **spawn** operation, which uses the vertex's increment handle to determine the node in the SNZI tree, where an arrive operation will start. To find the place, the `increment` operation first uses a `grow` operation to check whether the SNZI node pointed at by the handle has children. Additionally, the operation creates new children if necessary (line 20). Intuitively, we create new SNZI nodes to reduce contention by using more space. Thus, if the increment handle has children, we should make use of them and start our `arrive` operation there. This is exactly what the `increment` operation does (line 24). Note that, in case the `grow` operation does not return new children, we simply have the `arrive` operation start at the increment handle. The `increment` operation returns two increment handles and two decrement handles; one of each for each of the dag vertex's new children.

The in-counter algorithm makes use of an important SNZI property: decrements at the top of the tree do not affect any node below them. Furthermore, if there is a depart operation at SNZI node a, it cannot cause a to phase change as long there is surplus anywhere in a's subtree. These observations lead to the following intuition: to minimize phase changes in the SNZI tree, priority should be given to decrementing nodes closer to the root.

Thus, each dag vertex stores two decrement handles rather than just one. These two decrement handles are shared with the vertex's sibling, and they are ordered: the first handle always points to a SNZI node that is higher in the tree than the second. Decrement handle is needed by the `decrement` operation (line 31) and the `increment` operation. Recall that an `increment` always returns two decrement handles. One of these decrement handles always points to the SNZI node on which the `increment` operation started its `arrive`. However, the other one is inherited from the parent

dag vertex (the one that invoked the `increment`). To preserve the invariant, the incrementing vertex needs to claim a decrement handle (line 25). The two decrement handles returned are always ordered as follows: first, the one inherited from the parent, and, then, the one pointing at the freshly incremented SNZI node. In this way, we guarantee that the first decrement handle in any pair points higher in the tree than its counterpart. When a decrement handle is needed, the two dag vertices determine which handle to use through a test-and-set. The first of the two to make use of a decrement handle will take the first handle, thus ensuring that higher SNZI nodes are decremented earlier. We rely on this property to prove our bounds in Lemma 4.6.

Note that, in an `increment` operation, the decrement handle is claimed only *after* the `arrive` has completed. This key invariant helps to ensure that phase changes rarely occur in the SNZI tree, thus leading to fast operations.

4. Correctness and Analysis

We first prove that our in-counter data structure is linearizable and then establish its time and space efficiency. Due to lack of space, we only present the proof for the case where the probability used for the `grow` operation is 1, i.e., the SNZI tree grows on every `increment` operation. The proof for different growth probabilities is similar (and leads to contention that depends on the probability given) but more intricate, and is therefore left for the journal version of this paper.

We say that the in-counter is correct if any `is_zero` operation at the root of the tree correctly indicates whether there is a surplus of `increment` operations over `decrement` operations. To prove correctness, we establish a symmetry with the original SNZI data structure. Note that each `increment`, `decrement`, and `is_zero` operation calls the corresponding SNZI operation (i.e., `arrive`, `depart`, `query` respectively) exactly once. We define the linearization points of each operation as that of the corresponding SNZI operation.

Since the correctness condition for the in-counter and for SNZI are the same, we have the following observation.

Observation 1. *An execution of the in-counter is linearizable if the corresponding SNZI execution is linearizable.*

At this point, we recall that any SNZI run is linearizable if there are never more departs than arrives on any node, i.e., surplus at any node is never negative (14). We show that this invariant indeed holds for any valid execution of the in-counter data structure, and therefore that any valid execution is linearizable. We define a valid execution as follows.

Definition 1. *An in-counter execution is* valid *if any handle passed as argument to the* `decrement` *operation was returned by a prior* `increment` *operation. Furthermore, that handle is passed to at most one* `decrement` *operation.*

Lemma 4.1. *Any valid in-counter execution is linearizable.*

Proof. Observe that every `increment` operation generates one new decrement handle, and that handle points to the node at which this `increment`'s arrive operation was called. Thus, in valid executions, any `depart` on the underlying SNZI data structure (which is always called by the `decrement` operation) has a corresponding `arrive` at an earlier point in time. Therefore, there are never more departs than arrives at any SNZI node, and the corresponding SNZI execution is linearizable. From Observation 1, the in-counter execution is linearizable as well. □

It is easy to see that any execution of the in-counter that only accesses the object as prescribed by the sp-dag algorithm is valid. From this observation, the main theorem follows immediately.

Theorem 4.2. *Any execution of the in-counter through sp-dag accesses is linearizable.*

4.1 Running Time

To prove that the in-counter is efficient, we analyze shared-memory steps and contention separately: we first show that every operation takes an amortized constant number of shared memory steps, and then we show that each shared memory location experiences constant amortized contention at any time. In our analysis, we do not consider `is_zero`, since it does not do any non-trivial shared memory steps.

For the analysis, consider an execution on an sp-dag, starting with a dag, consisting of a single root vertex and its corresponding finish vertex. Further, observe that this finish vertex has a single SNZI node as the root of its in-counter. Note that all shared memory steps in the execution are on the in-counter. We begin by making an important observation.

Lemma 4.3. *There exist at most one increment and one decrement handle pointing to any given SNZI node.*

To prove this lemma, note that every `increment` operation creates new children for the SNZI node whose handle is used, and thus does not produce another handle to that node. A simple induction, starting with the fact that the root node only has one handle pointing to it, yields the result.

We now want to show that every operation on the in-counter performs an amortized constant number of shared memory steps. First, note that the only calls in the in-counter operations that might result in more than a constant number of steps are their corresponding SNZI operations. The SNZI operations do a constant number of shared memory steps per node they reach, but they can be recursively called up an arbitrarily long path in the tree, as long as nodes in that path phase change due to the operation. That is, frequent phase changing in the SNZI nodes will cause operations on the data structure to be slower. In fact, we show that on average, the length of the path traversed by an `arrive` or `depart` operation is constant, when those SNZI operations are called only through the in-counter operations.

We say that a dag vertex is *live* if it is not marked dead and a handle is live if it is owned by a live vertex. We have the following lemma about live vertices.

Lemma 4.4. *If dag vertex v is live, then it has not used* `claim_dec` *to claim a decrement handle.*

Keeping track of live vertices in the dag is important in the analysis, since by Lemma 4.4, these are the vertices that can use their handles and potentially cause more contention.

To show that operations on SNZI nodes are fast, we begin by considering executions in which only increment operations are allowed, and no decrements happen on the in-counter. Our goal is to show that the in-counter's SNZI tree is relatively saturated, so that `arrive` operations that are called within a `increment` only access a constant number of nodes. After we establish that increment operations are fast when decrements are not allowed, we bring back the decrement operations and see that they cannot slow down the increments.

Lemma 4.5. *Without any decrement operations, when there are no* `increments` *mid-execution, only leaves of the in-counter's SNZI tree can have surplus 0.*

Proof. The proof is by induction on the number of increment operations called on the in-counter. The tree is initialized with one node with surplus 1. Thus, when there have been 0 increments, no node has surplus 0. Assume that the lemma holds for in-counters that have had up to k increments on them. Consider an in-counter that has had k increments, and consider a new `increment` operation invoked on node a in the tree. If a is not a leaf, the `grow` operation does not create new children for a, and the tree does not grow. Thus, the lemma still holds. If a is a leaf, the `increment` operation creates two children for a and starts an arrive operation on one of the children. As a consequence of the SNZI invariant, we know that the surplus of a increased by 1. After the `increment`, regardless of what its surplus was before this operation, a cannot have surplus 0, and the lemma still holds. □

By Lemma 4.5, we know that, ignoring any effect that decrements may have, the tree has surplus in all non-leaf nodes. Recall that an `arrive` operation that reaches a node with positive surplus will terminate at that node. Thus, any `arrive` operation invoked on this tree will only climb up a path until it reaches the first node that was not a leaf before this `increment` started. That is, since each `increment` only expands the tree by at most one level, any `arrive` operation will reach at most 3 nodes.

We now bring back the `decrement` operations. The danger with decrements is that they could potentially cause non-leaf nodes to phase change back to 0 surplus, meaning that an `arrive` operation in that node's subtree might have to traverse a long path up the SNZI tree. Our main lemma shows that traversal of a long path can never happen; if a decrement causes a vertex's surplus to go back to 0, then no subsequent increment operation will start in that node's subtree. We present the proof in the full version.

Lemma 4.6. *If an in-counter node a at any point had a surplus and then phase changed to* 0 *surplus, then no live vertices in the dag point anywhere in its subtree.*

Note that Lemma 4.5 and Lemma 4.6 immediately give us the following important property.

Corollary 4.7. *No* `increment` *operation can invoke more than 3 arrive operations on the SNZI tree.*

Proof. We already saw that by Lemma 4.5, if no decrements happen, then each `increment` operation can invoke at most 3 `arrive` operations.

Now consider decrements as well. Note that if a non-leaf node has surplus 0 and there is no `increment` mid-operation at that node, then it must have previously had surplus (again by Lemma 4.5). By Lemma 4.6, no new `increment` operations can happen on that node's subtree. Thus, even allowing for decrement operations, no `increment` operation can invoke more than 3 arrives on the SNZI tree. □

To finish the proof, we amortize the number of `departs` invoked by `decrements` against the `arrives` invoked by `increments`. Recall that the number of `departs` invoked on a SNZI tree is at most the number of `arrives` invoked on it. We get the following theorem:

Theorem 4.8. *Any operation performed on the in-counter itself calls an amortized $O(1)$ operations on the SNZI tree.*

We now move on to analyzing the amount of contention a process experiences when executing an operation on the in-counter. Note that there have been several models suggested to account for contention, as discussed in Section 6. In our contention analysis, we say that an operation contends with a shared memory step if that step is non-trivial, and it could, in any execution, be concurrent with the operation at the same memory location. Our results are valid in any contention model in which trivial operations do not affect contention (i.e. such as stalls (16), the CRQW model (21), etc.).

Theorem 4.9. *Any operation executed on the in-counter experiences $O(1)$ amortized contention.*

Proof. We show something stronger: the maximum number of operations that access a single SNZI node over the course of an entire dag computation (other than `is_zero` operations on the root) is constant.

Consider a node a in the SNZI tree. By Lemma 4.3, only one `increment` operation ever starts its `arrive` at a. That `increment` has exactly one corresponding `decrement` whose `depart` starts at a. By the SNZI algorithm, an `arrive` at a's child only propagates up to a if that child's surplus was 0 before the `arrive`. By Lemma 4.6, this situation can only

happen once; if a's child's surplus returns to 0 after being higher, the counter will never be incremented again. Thus, each of a's children can only ever be responsible for at most two operations that access a; the initial `arrive` and the final `depart` from that subtree.

In total, we get a maximum of 6 operations that access a over the course of the computation. Note also that the first `arrive` at a node always strictly precedes any other operation in that node's subtree. Thus, concurrent `arrive` operations are not an issue. Therefore, any operation on the in-counter can be concurrent with at most 4 other operations per SNZI node it accesses. By Theorem 4.8, every operation on the in-counter accesses amortized $O(1)$ nodes. Therefore, any such operation experiences amortized $O(1)$ contention. □

4.2 Space bounds

Shrinking the tree by removing unnecessary nodes is tricky because, in its most general form, an ideal solution likely requires knowing the future of the computation. Knowing where in the tree to shrink requires knowing which nodes are not likey to be incremented in the future. For the specific case of the `grow` probability of 1, we establish a safety property that makes it possible to keep the data structure compact by removing SNZI nodes that correspond to deleted subgraphs of the sp-dag. Due to space constraints, we present the details in the full version (1).

5. Experimental evaluation

We report an empirical comparison between our in-counter algorithm, the simple fetch-and-add counter, and an algorithm using fixed-size SNZI trees. Our implementation consists of a small, C++ a library that uses a state-of-the-art, implementation of a work-stealing scheduler (2). Overall, our results show that the simple, fetch-and-add counter performs well only when there is one core, the fixed-size SNZI trees scale better, but our in-counter algorithm outperforms the others when the core count is two or more.

Implementation. Our implementation of core SNZI operations `snzi_arrive` and `snzi_depart` follows closely the algorithm presented in the original SNZI paper, except for two differences. First, we do not need `snzi_query` because readiness detection is performed via the return result of `snzi_depart`. This method suffices because only the caller of `snzi_depart` can bring the count to zero. Second, our `snzi_depart` returns `true` if the call brought the counter to zero. To control the grain of contention, we use the probabilistic technique presented in Section 2, with probability $p := \frac{1}{25c}$, where c is the number of cores. The idea of using c is to try to keep contention the same as the number of cores increase. We chose the constant factor 25, because it yields good results, but as our Threshold study, described below, shows many other constants also yield good results, e.g., any constant in the range $2.5c \leq p \leq 25c$ yields qualitatively the same results. The implementation of the sp-dags and in-counter data structures (Section 3) build on the SNZI data structure. The sp-dags interface enables writing nested-parallel programs, using various constructs, such as fork-join and async-finish; we use the latter in our experiments.

```
fun fanin_rec(n)
  if n >= 2
    async fanin_rec(n/2)
    async fanin_rec(n/2)

fun fanin(n)
  finish {fanin_rec(n)}
```

Figure 6: Fan-in benchmark.

```
fun indegree2(n)
  if n >= 2
    finish {
      async indegree2(n/2)
      async indegree2(n/2)
    }
```

Figure 7: Indegree-2 benchmark.

We compare our in-counter with an atomic, fetch-and-add counter because the fetch-and-add counter is optimal for very small numbers of cores. For higher number of cores, we designed a different, SNZI-based algorithm that uses a fixed-depth SNZI tree. This algorithm gives us another point of comparison, by offering a data structure that uses the existing state of the art more directly. The fixed-depth SNZI algorithm allocates for each finish block a SNZI tree of $2^{d+1} - 1$ nodes, for a given depth d. To maintain the critical SNZI invariant that the surplus of a SNZI node never becomes negative, the fixed-depth SNZI algorithm ensures that every `snzi_depart` call targets the same SNZI node that was targeted by a matching `snzi_arrive` call. To determine which SNZI node to be targeted by a `snzi_arrive` call, we map DAG vertices to SNZI nodes using a hash function to ensure that operations are spread evenly across the SNZI tree.

Experimental setup. We compiled the code using GCC `-O2 -march=native` (version 5.2). Our machine has four Intel E7-4870 chips and 1Tb of RAM and runs Ubuntu Linux kernel v3.13.0-66-generic. Each chip has ten cores (40 total) and shares a 30Mb L3 cache. Each core runs at 2.4Ghz and has 256Kb of L2 cache and 32Kb of L1 cache. The machine has a non-uniform memory architecture (NUMA), whereby RAM is partitioned into four banks. Pages are, by default in our Linux distribution, allocated to banks according to the first-touch policy, which assigns a freshly allocated page to the bank of the processor that first accesses the page. An alternative policy assigns pages to banks in a round-robin fashion. We determined that, for the purposes of our experiments, the choice of NUMA policy has no significant impact on our main results. The experiment we performed to back this claim is described in more detail in the full paper (1). For all other experiments, we used the round-robin policy.

Benchmarks: `fanin` and `indegree2`. To study scalability, we use the two small benchmarks shown in Figures 6 and 7. The `fanin` benchmark performs n async calls, all of which

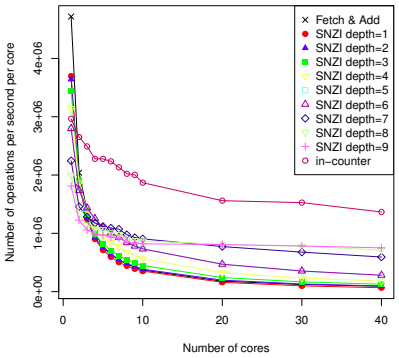

Figure 8: Fanin benchmark with $n = 8$ million, & varying number of processors. Higher is better.

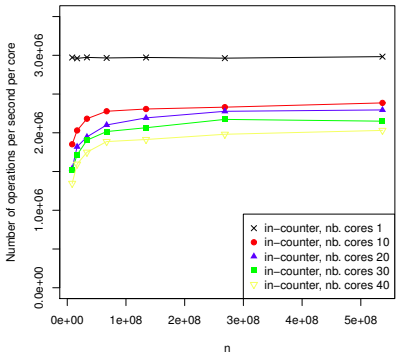

Figure 9: Fanin benchmark, varying the number of operations n. Higher is better.

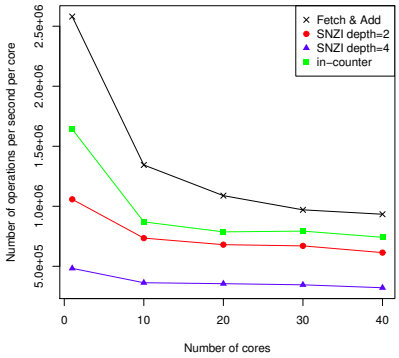

Figure 10: Indegree-2 benchmark, using number of operations $n = 8$ million. Higher is better.

synchronize at a single finish block, making the finish block a potential source of contention. The `fanin` benchmark implements a common pattern, such as a parallel-for, where a number of independent computations are forked to execute in parallel and synchronize at termination. Our second benchmark, `indegree2` implements the same pattern as in `fanin` but by using binary fork join. This program yields a computation dag in which each finish vertex has indegree 2.

Scalability study. Figure 8 shows the scalability of the `fanin` benchmark using different counter data structures. With one core, the Fetch & Add counter performs best because there is no contention, but with more, Fetch & Add gives worst performance for all core counts (this pattern is occluded by the other curves). For more than one core, our in-counter performs the best. The fixed-depth SNZI algorithm scales poorly when depth is small, doing best at the tree depth of 8, where there are enough nodes to deal with contention among 40 cores. Increasing the depth further does not improve the performance, however.

Size-invariance study. Theorem 4.9 shows that the amortized contention of our in-counter is constant. We test this result empirically by measuring the running time of the `fanin` benchmark for different input parameters n. As shown in Figure 9, for all input sizes considered and for all core counts, the throughput is within a factor 2 of the single-core (no-contention) Fetch & Add counter. The throughput suffers a bit for the smaller values of n because, at these values, there is not enough useful parallelism to feed the available cores.

Low-indegree study. Our next experiment examines the overhead imposed by the in-counter by using the `indegree2` benchmark shown in Figure 7. The results, illustrated in Figure 10, show that our in-counter data structure is within a factor 2 of the best performer, the fetch-and-add counter. For SNZI, we only considered small-depths, since larger ones took too long to run. With the fixed-depth SNZI, the benchmark creates many finish blocks and thus many SNZI trees, becoming inefficient for large trees.

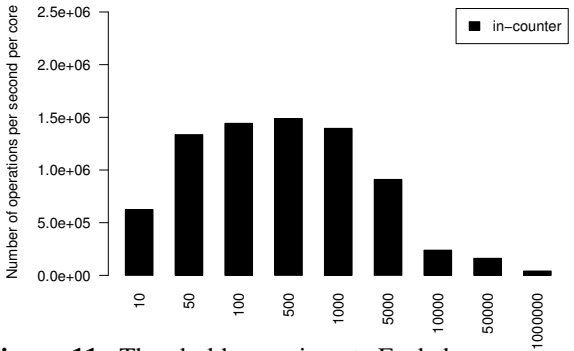

Figure 11: Threshold experiment. Each bar represents a different setting for $p = \frac{1}{threshold}$. All runs were performed using 40 cores. Higher is better.

Threshold study. In Section 2, we presented a technique to control the grain of contention by expanding the SNZI tree dynamically. Here, we report, using the `fanin` benchmark, the results of varying the range of probabilities $p = \frac{1}{threshold}$ for different settings of *threshold*. As shown in Figure 11, essentially any threshold between 50 and 1000 works well. We separately checked that on our test machine using a constant threshold such as 100 or 1000 yields good results when using any number of cores. On a machine, perhaps with many more cores, the best setting might be different or might also depend on the number of cores used.

6. Related Work

Upper bounds for several non-blocking data structures, accounting also for contention, have been proven for several tree- and list-based search structures (13; 18; 39). In these upper bounds, contention factors in as a linear additive term, which is consistent with established lower bounds for many problems in the general concurrency model (27; 30; 15; 16).

In contrast, for relaxed counters, we show that contention for series-parallel programs can be upper bounded by a constant.

Complexity models for analyzing contention and techniques for designing algorithms for reducing contention has also been important subject of study (36; 31; 12; 20; 21) Contention has also been identified as an important practical performance issue in multiprocessor systems. Techniques that reduce contention by detecting and introducing wait periods or using tokens to be passed between threads proved to be essential for reducing detrimental effects of contention (4; 6). Managing contention in software-transactional memory may require contention managers that schedule transactions carefully to minimize aborted transactions (41; 23; 26; 40).

In this paper, we consider non-blocking, or lock-free algorithms. Since our upper bounds are quite good, $O(1)$, our algorithms guarantee that each operation completes quickly in our relaxed concurrency model. Wait-free data structures can guarantee similar properties in the general concurrency model (24; 25). Such data structures were traditionally considered to be impractical, though recent results show that they may be also be practical (33; 44).

7. Conclusion

The design and analysis of provably efficient non-blocking data structures has been a challenging problem. One reason is the complexity of the concurrency models. Another reason is the difficulty of accounting for important practical concerns, such as contention, that can significantly impact performance. In this paper, we show that it is possible to design provably efficient data structures under a slightly more restrictive but still important concurrency model: nested-parallel computations. Our results take advantage of certain structural invariants in nested parallel computations, specifically those that concern the creation, termination, and synchronization of threads. Interesting future research directions include the design and analysis of other concurrent data structures for the nested-parallel concurrency model and consideration of more general, but still restricted, models of concurrency, such as those based on futures.

Acknowledgments

This research is partially supported by the a fellowship from Natural Sciences and Engineering Research Council of Canada (NSERC), and grants from National Science Foundation (CCF-1320563 and CCF-1408940) and European Research Council (grant ERC-2012-StG-308246). We are grateful to Doug Lea for his feedback on the paper.

References

[1] Umut A. Acar, Naama Ben-David, and Mike Rainey. Contention in structured concurrency: Provably efficient dynamic nonzero indicators for nested parallel computation. Technical Report CMU-CS-16-133, Department of Computer Science, Carnegie Mellon University, 2016.

[2] Umut A. Acar, Arthur Charguéraud, and Mike Rainey. Scheduling parallel programs by work stealing with private deques. In *PPoPP '13*, 2013.

[3] Umut A. Acar, Arthur Charguéraud, Mike Rainey, and Filip Sieczkowski. Dag-calculus: A calculus for parallel computation. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP 2016, pages 18–32, 2016.

[4] A. Agarwal and M. Cherian. Adaptive backoff synchronization techniques. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, ISCA '89, pages 396–406, 1989.

[5] James H. Anderson and Yong-Jik Kim. An improved lower bound for the time complexity of mutual exclusion. *Distrib. Comput.*, 15(4):221–253, December 2002.

[6] T. E. Anderson. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. Parallel Distrib. Syst.*, 1(1):6–16, January 1990.

[7] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the tenth annual ACM symposium on Parallel algorithms and architectures*, SPAA '98, pages 119–129. ACM Press, 1998.

[8] Robert D. Blumofe and Charles E. Leiserson. Scheduling multithreaded computations by work stealing. *J. ACM*, 46:720–748, September 1999.

[9] 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 '14, pages 329–342, 2014.

[10] Philippe Charles, Christian Grothoff, Vijay Saraswat, Christopher Donawa, Allan Kielstra, Kemal Ebcioglu, Christoph von Praun, and Vivek Sarkar. X10: an object-oriented approach to non-uniform cluster computing. In *Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, OOPSLA '05, pages 519–538. ACM, 2005.

[11] Robert Cypher. The communication requirements of mutual exclusion. In *Proceedings of the Seventh Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '95, pages 147–156, 1995.

[12] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM (JACM)*, 44(6):779–805, 1997.

[13] Faith Ellen, Panagiota Fatourou, Joanna Helga, and Eric Ruppert. The amortized complexity of non-blocking binary search trees. In *Proceedings of the 2014 ACM Symposium on Principles of Distributed Computing*, PODC '14, pages 332–340, 2014.

[14] Faith Ellen, Yossi Lev, Victor Luchangco, and Mark Moir. Snzi: Scalable nonzero indicators. In *Proceedings of the Twenty-sixth Annual ACM Symposium on Principles of Distributed Computing*, PODC '07, pages 13–22, 2007.

[15] Faith Fich and Eric Ruppert. Hundreds of impossibility results for distributed computing. *Distrib. Comput.*, 16(2-3):121–163, September 2003.

[16] Faith Ellen Fich, Danny Hendler, and Nir Shavit. Linear lower bounds on real-world implementations of concurrent objects. In *Foundations of Computer Science, 2005. FOCS 2005. 46th Annual IEEE Symposium on*, pages 165–173. IEEE, 2005.

[17] Matthew Fluet, Mike Rainey, John Reppy, and Adam Shaw. Implicitly threaded parallelism in Manticore. *Journal of Functional Programming*, 20(5-6):1–40, 2011.

[18] Mikhail Fomitchev and Eric Ruppert. Lock-free linked lists and skip lists. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 50–59, 2004.

[19] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, pages 212–223, 1998.

[20] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran. Efficient low-contention parallel algorithms. In *Proceedings of the sixth annual ACM symposium on Parallel algorithms and architectures*, pages 236–247. ACM, 1994.

[21] Phillip B Gibbons, Yossi Matias, Vijaya Ramachandran, et al. The queue-read queue-write pram model: Accounting for contention in parallel algorithms. *SIAM Journal on Computing*, pages 638–648, 1997.

[22] James R. Goodman, Mary K. Vernon, and Philip J. Woest. Efficient synchronization primitives for large-scale cache-coherent multiprocessors. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, pages 64–75, 1989.

[23] Tim Harris and Keir Fraser. Language support for lightweight transactions. In *Proceedings of the 18th Annual ACM SIGPLAN Conference on Object-oriented Programing, Systems, Languages, and Applications*, OOPSLA '03, pages 388–402, 2003.

[24] Maurice Herlihy. Wait-free synchronization. *ACM Trans. Program. Lang. Syst.*, 13:124–149, January 1991.

[25] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Trans. Program. Lang. Syst.*, 15(5):745–770, November 1993.

[26] Maurice Herlihy, Victor Luchangco, Mark Moir, and William N Scherer III. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 92–101. ACM, 2003.

[27] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.

[28] Shams Mahmood Imam and Vivek Sarkay. Habanero-java library: a java 8 framework for multicore programming. In *2014 International Conference on Principles and Practices of Programming on the Java Platform Virtual Machines, Languages and Tools, PPPJ '14, Cracow, Poland, September 23-26, 2014*, pages 75–86, 2014.

[29] Intel. Intel threading building blocks. 2011. https://www.threadingbuildingblocks.org/.

[30] Prasad Jayanti, King Tan, and Sam Toueg. Time and space lower bounds for nonblocking implementations. *SIAM Journal on Computing*, 30(2):438–456, 2000.

[31] Richard M Karp and Yanjun Zhang. Randomized parallel algorithms for backtrack search and branch-and-bound computation. *Journal of the ACM (JACM)*, 40(3):765–789, 1993.

[32] Gabriele Keller, Manuel M.T. Chakravarty, Roman Leshchinskiy, Simon Peyton Jones, and Ben Lippmeier. Regular, shape-polymorphic, parallel arrays in haskell. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, ICFP '10, pages 261–272, 2010.

[33] Alex Kogan and Erez Petrank. A methodology for creating fast wait-free data structures. In *Proceedings of the 17th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '12, pages 141–150, 2012.

[34] Doug Lea. A java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, 2000.

[35] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. In *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, OOPSLA '09, pages 227–242, 2009.

[36] Pangfeng Liu, William Aiello, and Sandeep Bhatt. An atomic model for message-passing. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures*, pages 154–163. ACM, 1993.

[37] Maged M. Michael and Michael L. Scott. Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. *J. Parallel Distrib. Comput.*, 51(1):1–26, May 1998.

[38] Mark Moir and Nir Shavit. Concurrent data structures. *Handbook of Data Structures and Applications*, pages 47–14, 2007.

[39] Rotem Oshman and Nir Shavit. The skiptrie: Low-depth concurrent search without rebalancing. In *Proceedings of the 2013 ACM Symposium on Principles of Distributed Computing*, PODC '13, pages 23–32, 2013.

[40] William N. Scherer, III and Michael L. Scott. Advanced contention management for dynamic software transactional memory. In *Proceedings of the Twenty-fourth Annual ACM Symposium on Principles of Distributed Computing*, PODC '05, pages 240–248, 2005.

[41] Nir Shavit and Dan Touitou. Software transactional memory. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, PODC '95, pages 204–213, 1995.

[42] Nir Shavit and Asaph Zemach. Combining funnels. *J. Parallel Distrib. Comput.*, 60(11):1355–1387, November 2000.

[43] Julian Shun, Guy E. Blelloch, Jeremy T. Fineman, and Phillip B. Gibbons. Reducing contention through priority updates. In *Proceedings of the Twenty-fifth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '13, pages 152–163, 2013.

[44] Shahar Timnat and Erez Petrank. A practical wait-free simulation for lock-free data structures. In *Proceedings of the 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '14, pages 357–368, 2014.

[45] R Kent Treiber. Systems programming: Coping with parallelism. Technical report, International Business Machines Incorporated, Thomas J. Watson Research Center, 1986.