Contestion in Structured Concurrency: Provably efficient Dynamic Non-Zero Indicators for Nested Parallelism

Umut Acar¹,², Naama Ben-David¹, and Mike Rainey²

¹ Carnegie Mellon University
² INRIA, France
Concurrent Algorithms

- We are interested in designing **non-blocking data structures** in shared memory.
Concurrent Algorithms

- Threads operate asynchronously.
- A lot of work proving correctness and liveness of shared data structures.

**Efficiency:**
- Mostly experimental evaluation
- Relatively little work on establishing bounds
Running Time Bounds: Challenges

- **Asynchrony** allows for too many possibilities.

- Sometimes we can’t do better than “must terminate eventually”, or even “might never terminate”.

- Classic asynchronous model doesn’t account for practical costs like *contention*. 
Contestation

• Hardware *sequentializes accesses* to the same memory location.

• Accessing a busy location takes much longer than an uncontested one.
Contention

• Hardware **sequentializes accesses** to the same memory location.

• Accessing a busy location takes much longer than an uncontested one.
Contestion

- Hardware **sequentializes accesses** to the same memory location.

- Accessing a busy location takes much longer than an uncontested one.

```
thread t

m
```

```
I want to increment m!
```

```
memory location m
```

0
Contention

- Hardware *sequentializes accesses* to the same memory location.
- Accessing a busy location takes much longer than an uncontested one.

![Diagram](image)
Contention

• Hardware **sequentializes accesses** to the same memory location.

• Accessing a busy location takes much longer than an uncontested one.

thread $t$

memory location $m$

I want to increment $m$!
Contention

- Hardware **sequentializes accesses** to the same memory location.

- Accessing a busy location takes much longer than an uncontested one.
Contestation Lower Bound

- Several asynchronous shared memory models that account for contention [DHW’97, FHS’05, HK’08].

- $\Omega(n)$ contention lower bound ($n = \# \text{ of threads}$) for a class of data structures, including counters [FHS’05].
  - Regardless of implementation
  - At least one thread experiences $\Omega(n)$ contention
Running Time Bounds: Challenges 2.0

- Classic asynchronous model doesn’t account for practical costs like *contention*.

- Additional challenge: When we do model it, we have discouraging *lower bounds*.
Our Goal

• **Broadly:** we wish to design *provably efficient* concurrent algorithms

• **Approach:** Consider a *relaxed concurrency model* that is practically relevant

• **Our result:** A provably efficient concurrent dependency counter for *nested parallelism*
Nested Parallelism

- Two computations can be done in parallel if neither one depends on the result of the other.
- Otherwise, they must be done serially.
- Each computation can either fork or terminate.
Nested Parallelism

- Two computations can be done **in parallel** if neither one depends on the result of the other.

- Otherwise, they must be done **serially**.

- Each computation can either **fork** or **terminate**.

```plaintext
AddToAll(A):
    for (i = 0; i < size(A); i++):
        A[i] += 1;
```
Nested Parallelism

• Two computations can be done in parallel if neither one depends on the result of the other.

• Otherwise, they must be done serially.

• Each computation can either fork or terminate.

AddToAll(A):

for (i = 0; i < size(A); i++):
  A[i] += 1;

For-loop can be done in parallel!
Parallel For

size(A) = n

Finish vertex
Parallel For

size(A) = n

Fork

size(A) = n/2

Finish vertex
Parallel For

size(A) = n
size(A) = n/2
size(A) = n/4

Fork
Fork
Fork

Finish vertex
Parallel For

size(A) = n
size(A) = n/2
size(A) = n/4
size(A) = 1

Finish vertex
Parallel For

size(A) = n
size(A) = n/2
size(A) = n/4
size(A) = 1

Fork
Fork
...
Fork
Terminate
Finish vertex
Parallel For

size(A) = n
size(A) = n/2
size(A) = n/4

A
Fork
Fork
Fork

size(A) = 1

Asynchrony

Finish vertex
Parallel For

A

size(A) = n

size(A) = n/2

size(A) = n/4

... size(A) = 1

Fork

Fork

Fork

Terminate

Finish vertex

Asynchrony
Parallel For

A

Asynchrony

Fork

size(A) = n

size(A) = n/2

size(A) = n/4

...
Parallel For

size(A) = n
size(A) = n/2
size(A) = n/4

Fork
Fork
... 
Fork

size(A) = 1

Terminate

Finish vertex

Asynchrony
Parallel For

A

size(A) = n

size(A) = n/2

size(A) = n/4

Finish vertex

Nested Computations
Parallel For

size(A) = n

size(A) = n/2

size(A) = n/4

Fork

Fork

Finish vertex

Finish vertex

Nested Computations
Parallel For

A

size(A) = n

size(A) = n/2

size(A) = n/4

Fork

Fork

Nested Computations

Finish vertex

Finish vertex
Parallel For

size(A) = n
size(A) = n/2
size(A) = n/4

Fork
Fork

Nested Computations

Finish vertex
Finish vertex
Dependency Counters: Desiderata

• Must know if a task is still waiting on dependencies.

• Operations needed (called a non-zero indicator):
  • Increment
  • Decrement
  • Query (Did it reach zero?)

• Must be able to handle concurrent accesses.

• Want it as efficient as possible.
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.

**Idea:** Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.

**Idea:** Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.

Idea: Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.

Idea: Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.

Idea: Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.

**Idea:** Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).
Scalable Non-Zero Indicators [ELLM’07]

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.

Idea: Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).
Idea: Notify your parent of change only if you “phase change” (i.e. surplus changes from 0 to 1 or vice versa).

- Have a tree to filter non-essential updates.
- Each node’s counter indicates whether its subtree has surplus.
Scalable Non-Zero Indicators [ELLMM’07]

- Concurrency makes things more complex
- SNZI is flexible: user can pick any tree size/shape
- Key limitation: the tree is static.
- How do we pick the tree?
SNZI Slow Executions

What makes executions slow?

- Shared memory steps (in this case, accessing many nodes)
- Contention

Small Tree Example

$n$ concurrent operations $\rightarrow \Omega(n)$ contention
SNZI Slow Executions

What makes executions slow?

- Shared memory steps (in this case, accessing many nodes)
- Contention

Large Tree Example

One thread alone traverses long paths.
Can We Make SNZI Dynamic?
Our Work: Dynamic SNZI

• To allow SNZI to grow dynamically without sacrificing its correctness, we add another operation.

```python
def grow(p: probability)
    myChildren = read(children)
    if (myChildren == null && flip(p) == heads)
        then CAS(children, null, new children)
    return read(children)
```
Dependency Counters using Dynamic SNZI

- One dynamic SNZI per finish vertex in the dag.

- Intuitively, each *fork increments* and each *terminate decrements*.

- To control contention, SNZI tree grows to match concurrency level.

- How do we determine where operations begin?
Dependency Counters using Dynamic SNZI

Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.
Dependency Counters using Dynamic SNZI

Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.

Parallel DAG

SNZI tree

Finish task
Dependency Counters using Dynamic SNZI

Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.

Parallel DAG

\[ ptr = c \]

SNZI tree

\[
\begin{array}{c}
\text{a} & \text{1} \\
\text{b} & \text{1} \\
\text{c} & \text{0}
\end{array}
\]

Finish task

\text{Fork!}
Dependency Counters using Dynamic SNZI

Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.
Dependency Counters using Dynamic SNZI

Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.

Parallel DAG

SNZI tree

ptr = c

Fork!
Dependency Counters using Dynamic SNZI

Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.
Dependency Counters using Dynamic SNZI

Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.

Parallel DAG

\[ \text{ptr} = c \]

Fork!

Finish task

SNZI tree

Increment!
Idea: Give each vertex a pointer to where it should start its operation, to avoid having to search for it.
Dependency Counters using Dynamic SNZI

- Actual algorithm a bit more intricate
- Split into **increment** and **decrement** pointers
Running Time Analysis

**Theorem:** Every operation takes amortized $O(1)$ time (shared memory steps and contention).

- **Few pointers to every SNZI node**
- **Once a node reaches 0, it is never accessed again.**

Low contention

Few memory accesses
Experimental Results

![Graph showing the number of operations per second per core versus the number of cores for different SNZI depth configurations. Different markers and colors represent different SNZI depths, with the legend indicating Fetch & Add, SNZI depth=1 to 9, and in-counter.]
Summary

• Our Contributions:
  • Dynamic SNZI
  • Efficient dependency counters

• Future Directions
  • More general provably-efficient concurrent data structures for nested parallelism
  • Design similar data structures for different relaxed concurrency models
Summary

• Our Contributions:
  • Dynamic SNZI
  • Efficient dependency counters

• Future Directions
  • More general provably-efficient concurrent data structures for nested parallelism
  • Design similar data structures for different relaxed concurrency models

Thank you!