Shared Counters and Parallelism

Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit
A Shared Pool

- **Put**
  - Insert item
  - block if full

- **Remove**
  - Remove & return item
  - block if empty

```java
public interface Pool<T> {
    public void put(T x);
    public T remove();
}
```
Simple Locking Implementation

put

put

put
Simple Locking Implementation

Problem: hot-spot contention
Simple Locking Implementation

Problem: hot-spot contention

Problem: sequential bottleneck
Simple Locking Implementation

Problem: sequential bottleneck

Problem: hot-spot contention

Solution: Queue Lock
Simple Locking Implementation

Problem: sequential bottleneck
Solution: Queue

Problem: hot-spot contention
Solution: Lock
Counting Implementation

Art of Multiprocessor Programming
Counting Implementation

Only the counters are sequential
Shared Counter
Shared Counter

No duplication
Shared Counter

No duplication

No Omission
Shared Counter

No duplication

No Omission

Not necessarily linearizable
Shared Counters

- Can we build a shared counter with
  - Low memory contention, and
  - Real parallelism?

- Locking
  - Can use queue locks to reduce contention
  - No help with parallelism issue …
Parallel Counter Approach

How to coordinate access to counters?

- 0, 4, 8.....
- 1, 5, 9.....
- 2, 6, 10.....
- 3, 7 .......

Parallel Counter Approach
A Balancer

Input wires

Output wires
Tokens Traverse Balancers

- Token $i$ enters on any wire
- leaves on wire $i \pmod{2}$
Tokens Traverse Balancers
Tokens Traverse Balancers
Tokens Traverse Balancers
Tokens Traverse Balancers
Quiescent State: all tokens have exited

Arbitrary input distribution

Balanced output distribution
Smoothing Network

1-smooth property
Counting Network

step property
Counting

Step property guarantees no duplication or omissions, how?

Multiple counters distribute load

Counters

0, 4, 8, ...
1, 5, 9, ...
2, 6, ...
3, 7, ...

Art of Multiprocessor Programming
Step property guarantees that in-flight tokens will take missing values

If 5 and 9 are taken before 4 and 8
Counting Networks

• Good for counting number of tokens
• low contention
• no sequential bottleneck
• high throughput
• practical networks depth $\log^2 n$
Counting Network
Counting Network
Counting Network

![Counting Network Diagram]

Art of Multiprocessor Programming
Counting Network
Counting Network
Counting Network
Bitonic[k] Counting Network
Bitonic[k] Counting Network
Bitonic\([k]\) not Linearizable
Bitonic[k] is not Linearizable
Bitonic[\(k\)] is not Linearizable
Bitonic\([k]\) is not Linearizable.
Bitonic[k] is not Linearizable

Problem is:
- Red finished before Yellow started
- Red took 2
- Yellow took 0
But it is “Quiescently Consistent”

Has Step Property in any quiescent State (one in which all tokens have exited)
Shared Memory Implementation

class balancer {
    boolean toggle;
    balancer[] next;

    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
Shared Memory Implementation

class balancer {
  boolean toggle;
  balancer[] next;
}

synchronized boolean flip() {
  boolean oldValue = this.toggle;
  this.toggle = !this.toggle;
  return oldValue;
}
Shared Memory Implementation

class balancer {
    boolean toggle;
    balancer[] next;

    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
Shared Memory Implementation

class balancer {
    boolean toggle;
    balancer[] next;

    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
Shared Memory Implementation

Balancer traverse (Balancer b) {
  while (!b.isLeaf()) {
    boolean toggle = b.flip();
    if (toggle)
      b = b.next[0]
    else
      b = b.next[1]
    return b;
  }
}
Shared Memory Implementation

Balancer traverse (Balancer b) {
    while (!b.isLeaf()) {
        boolean toggle = b.flip();
        if (toggle)
            b = b.next[0]
        else
            b = b.next[1]
        return b;
    }

Stop when we exit the network
Balancer traverse (Balancer b) {
    while (!b.isLeaf()) {
        boolean toggle = b.flip();
        if (toggle) 
            b = b.next[0]
        else
            b = b.next[1]
        return b;
    }
}
Shared Memory Implementation

Balancer traverse (Balancer b) {
    while (!b.isLeaf()) {
        boolean toggle = b.flip();
        if (toggle)
            b = b.next[0];
        else
            b = b.next[1];
        return b;
    }
}
Bitonic[2k] Inductive Structure

Bitonic[2]

Bitonic[2]

Merger[4]

Unfolded Bitonic[8] Network

Merger[8]
Unfolded Bitonic[8] Network
Unfolded Bitonic[8] Network
Bitonic\([k]\) Depth

- Width \(k\)
- Depth is \((\log_2 k)(\log_2 k + 1)/2\)
Proof by Induction

• Base:
  – Bitonic[2] is single balancer
  – has step property by definition

• Step:
  – If Bitonic[k] has step property …
  – So does Bitonic[2k]
Bitonic[2k] Schematic
Need to Prove only Merger[2k]

Induction Hypothesis

Need to prove
Merger\[2k\] Schematic
Merger\([2k]\) Layout
Induction Step

- Bitonic[k] has step property ...
- Assume Merger[k] has step property if it gets 2 inputs with step property of size k/2 and
- prove Merger[2k] has step property
Assume Bitonic[k] and Merger[k] and Prove Merger[2k]

Induction Hypothesis

Need to prove
Proof: Lemma 1

If a sequence has the step property ...
Lemma 1

So does its even subsequence
Lemma 1

Also its odd subsequence
Lemma 2

Even + odd

Diff at most 1

Even + odd

Odd + even
Bitonic[2k] Layout Details

Bitonic[k] → even → Merger[k] → odd → Bitonic[k]
By induction hypothesis

Outputs have step property

Bitonic[k]

Merger[k]

Bitonic[k]

Merger[k]
By Lemma 1

All subsequences have step property
By Lemma 2

Diff at most 1

Art of Multiprocessor Programming
By Induction Hypothesis

Outputs have step property
By Lemma 2

At most one diff

Merger[k]

Merger[k]
Last Row of Balancers

Merger[k]

Merger[k]

Outputs of Merger[k]

Outputs of last layer
Last Row of Balancers

Merger[k]

Merger[k]

Wire i from one merger

Wire i from other merger
Last Row of Balancers

Merger[k]

Merger[k]

Outputs of Merger[k]

Outputs of last layer
Last Row of Balancers

Merger\[k\]

Merger\[k\]
So Counting Networks Count

Merger[k]

Merger[k]

QED
Periodic Network Block
Periodic Network Block
Periodic Network Block
Periodic Network Block
Block[2k] Schematic
Block[2k] Layout
Periodic[8]
Network Depth

• Each block$[k]$ has depth $\log_2 k$
• Need $\log_2 k$ blocks
• Grand total of $(\log_2 k)^2$
Lower Bound on Depth

Theorem: The depth of any width $w$ counting network is at least $\Omega(\log w)$.

Theorem: there exists a counting network of $\Theta(\log w)$ depth.

Unfortunately, proof is non-constructive and constants in the 1000s.
Sequential Theorem

• If a balancing network counts
  – Sequentially, meaning that
  – Tokens traverse one at a time

• Then it counts
  – Even if tokens traverse concurrently
Red First, Blue Second
Blue First, Red Second
Either Way

Same balancer states
Order Doesn’t Matter

Same balancer states

Same output distribution
Index Distribution Benchmark

```java
void indexBench(int iters, int work) {
    while (int i = 0 < iters) {
        i = fetch&inc();
        Thread.sleep(random() % work);
    }
}
```
Performance (Simulated)

Higher is better!

Throughput

Number processors

MCS queue lock
Spin lock

* All graphs taken from Herlihy, Lim, Shavit, copyright ACM.
Performance (Simulated)

Throughput

Number processors

64-leaf combining tree
80-balance counting network

MCS queue lock
Spin lock

Higher is better!
Performance (Simulated)

- 64-leaf combining tree
- 80-balancer counting network

Combining and counting are pretty close

Throughput

Number processors

MCS queue lock
Spin lock
Performance (Simulated)

- 64-leaf combining tree
- 80-balance counting network

But they beat the hell out of the competition!

MCS queue lock
Spin lock

Throughput vs. Number of processors
Saturation and Performance

Undersaturated $P < w \log w$

Saturated $P = w \log w$

Oversaturated $P > w \log w$
Throughput vs. Size

Throughput vs. Number of processors for Bitonic[4], Bitonic[8], and Bitonic[16].
Shared Pool

![Diagram of Shared Pool with 'put' and 'remove' actions]
What About

• Decrement
• Adding arbitrary values
• Other operations
  – Multiplication
  – Vector addition
  – Horoscope casting …
First Step

• Can we decrement as well as increment?
• What goes up, must come down …
Anti-Tokens

[Diagram of Anti-Tokens]
Tokens & Anti-Tokens Cancel
Tokens & Anti-Tokens Cancel

Diagram showing a flow of tokens through a system with cancellation points.
Tokens & Anti-Tokens Cancel
Tokens & Anti-Tokens Cancel

As if nothing happened
Tokens vs Antitokens

- Tokens
  - read balancer
  - flip
  - proceed

- Antitokens
  - flip balancer
  - read
  - proceed
Pumping Lemma

Eventually, after $\Omega$ tokens, network repeats a state

Keep pumping tokens through one wire
Anti-Token Effect

token

anti-token
Observation

- Each anti-token on wire $i$
  - Has same effect as $\Omega-1$ tokens on wire $i$
  - So network still in legal state
- Moreover, network width $w$ divides $\Omega$
  - So $\Omega-1$ tokens
Before Antitoken
Balancer states as if …

Ω-1 is one brick shy of a load
Post Antitoken

Next token shows up here
Implication

• Counting networks with
  – Tokens (+1)
  – Anti-tokens (-1)

• Give
  – Highly concurrent
  – Low contention

• getAndIncrement + getAndDecrement methods

QED
Adding Networks

• Combining trees implement
  – Fetch&add
  – Add any number, not just 1

• What about counting networks?
Fetch-and-add

- Beyond `getAndIncrement + getAndDecrement`
- What about `getAndAdd(x)`?
  - Atomically returns prior value
  - And adds x to value?
- Not to mention
  - `getAndMultiply`
  - `getAndFourierTransform`?
Bad News

• If an adding network
  – Supports $n$ concurrent tokens
• Then every token must traverse
  – At least $n-1$ balancers
  – In sequential executions
Uh-Oh

- Adding network size depends on $n$
  - Like combining trees
  - Unlike counting networks
- High latency
  - Depth linear in $n$
  - Not logarithmic in $w$
Generic Counting Network
First Token

First token would visit green balancers if it runs solo.
Claim

• Look at path of +1 token
• All other +2 tokens must visit some balancer on +1 token’s path
Second Token

Takes 0
Second Token

They can’t both take zero!
If Second avoids First’s Path

- **Second token**
  - Doesn’t observe first
  - First hasn’t run
  - Chooses 0

- **First token**
  - Doesn’t observe second
  - Disjoint paths
  - Chooses 0
If Second avoids First’s Path

• Because +1 token chooses 0
  – It must be ordered first
  – So +2 token ordered second
  – So +2 token should return 1

• Something’s wrong!
Second Token

Halt blue token before first green balancer
Third Token

Takes 0 or 2
Third Token

They can’t both take zero, and they can’t take 0 and 2!

Takes 0

Takes 0 or 2
First, Second, & Third Tokens must be Ordered

• Third (+2) token
  – Did not observe +1 token
  – May have observed earlier +2 token
  – Takes an even number
First, Second, & Third Tokens must be Ordered

• Because +1 token’s path is disjoint
  – It chooses 0
  – Ordered first
  – Rest take odd numbers

• But last token takes an even number

• Something’s wrong!
Third Token

Halt blue token before first green balancer
Continuing in this way

- We can “park” a token
  - In front of a balancer
  - That token #1 will visit
- There are n-1 other tokens
  - Two wires per balancer
  - Path includes n-1 balancers!
Theorem

• In any adding network
  – In sequential executions
  – Tokens traverse at least n-1 balancers

• Same arguments apply to
  – Linearizable counting networks
  – Multiplying networks
  – And others
Shared Pool

Depth $\log^2 w$

Can we do better?
Counting Trees

Single input wire
Counting Trees
Counting Trees
Counting Trees
Counting Trees

Step property in quiescent state
Counting Trees

Interleaved output wires
Inductive Construction

Tree[2k] = Tree_0[k] + Tree_1[k]

At most 1 more token in top wire

Tree[2k] has step property in quiescent state.
Inductive Construction

$$\text{Tree}[2k] = \text{Tree}_0[k] \downarrow \text{Tree}_1[k]$$

- For $k$ even, outputs:
- For $k$ odd, outputs:

Top step sequence has at most one extra on last wire of step.

Tree[2k] has step property in quiescent state.
Implementing Counting Trees
Example
Example
Implementing Counting Trees

Contension

Sequential bottleneck
Diffraction Balancing

If an even number of tokens visit a balancer, the toggle bit remains unchanged!
Diffracting Balancer

Diffracting Tree

Diffracting balancer same as balancer.
Diffracting Tree

Prism

Diffracting Balancer

High load → Lots of Diffraction + Few Toggles
Low load → Low Diffraction + Few Toggles

High Throuhput with Low Contention
Performance

Throughput

Latency

P=Concurrency

MCS

Dtree

Ctree

0
50
100
150
200
250
300

0
50
100
150
200
250
300

0
4000
8000
12000
16000

0
2000
4000
6000
8000
10000
12000
140000
160000
Amdahl’s Law Works

Fine grained parallelism gives great performance benefit

Coarse Grained

25% Shared

75% Unshared

Fine Grained

25% Shared

75% Unshared

Fine grained parallelism gives great performance benefit
But…

- Can we always draw the right conclusions from Amdahl’s law?
- Claim: sometimes the overhead of fine-grained synchronization is so high...that it is better to have a single thread do all the work sequentially in order to avoid it
Software Combining Tree

Tree requires a major coordination effort: multiple CAS operations, cache-misses, etc.
Oyama et. al Mutex

Object lock

Every request involves CAS

CAS()

Apply a, b, c, and d to object

Release lock

Return responses
Flat Combining

- Have single lock holder collect and perform requests of all others
  - Without using CAS operations to coordinate requests
  - With combining of requests (if cost of $k$ batched operations is less than that of $k$ operations in sequence $\rightarrow$ we win)
Most requests do not involve a CAS, in fact, not even a memory barrier
Flat-Combining Pub-List Cleanup

Every combiner increments counter and updates record’s time stamp when returning response.

Cleanup requires no CAS, only reads and writes.

If thread reappears must add itself to pub list.
Fine-Grained Lock-free FIFO Queue

P: Dequeue() => a
Q: Enqueue(d)
Flat Combining FIFO Queue

OK, but can do better...combining: collect all items into a "fat node", enqueue in one step

Sequential FIFO Queue

- **Head**
- **Tail**

**Publication list**

- **null**
- **Enq(b) 12**
- **Enq(b) 54**

**CAS()**

**Deq()**

**Enq(a)**

**Enq(b)**
OK, but can do better…combining: collect all items into a “fat node”, enqueue “Fat Node” easy sequentially but cannot be done in concurrent alg without CAS

Sequential
“Fat Node” FIFO Queue
Linearizable FIFO Queue

SPARC T2 - QUEUE - Throughput
50% ENQ; 50% DEQ

ops / ms

fc
michael scott
Basket
tree
oyama
oyama combin
log sync

thre

MS queue, Oyama, and Log-Synch

Combining tree

Flat Combining
Flat combining with sequential pairing heap plugged in...
Don’t be Afraid of the Big Bad Lock

- Fine grained parallelism comes with an overhead…not always worth the effort.
- Sometimes using a single global lock is a win.
This work is licensed under a Creative Commons Attribution-ShareAlike 2.5 License.

- **You are free:**
  - **to Share** — to copy, distribute and transmit the work
  - **to Remix** — to adapt the work
- **Under the following conditions:**
  - **Attribution.** You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).
  - **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.
- For any reuse or distribution, you must make clear to others the license terms of this work. The best way to do this is with a link to
  - http://creativecommons.org/licenses/by-sa/3.0/.
- Any of the above conditions can be waived if you get permission from the copyright holder.
- Nothing in this license impairs or restricts the author's moral rights.