Shared Counters and Parallelism

Maurice Herlihy
CS176
Fall 2005
A Shared Pool

public interface Pool {
    public void put(Object x);
    public Object remove();
}

Unordered set of objects

- **Put**
  - Inserts object
  - blocks if full

- **Remove**
  - Removes & returns an object
  - blocks if empty
Simple Locking Implementation
Simple Locking Implementation

Problem: hot-spot contention
Simple Locking Implementation

Problem: hot-spot contention

Problem: sequential bottleneck
Simple Locking Implementation

Problem: hot-spot contention

Solution: Queue Lock

Problem: sequential bottleneck
Simple Locking Implementation

Problem: hot-spot contention

Solution: Queue Lock

Solution: sequential bottleneck

Problem: sequential bottleneck

put

put

Shavit
Counting Implementation

(c) 2003-2005 Herlihy and Shavit
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 ...
Software Combining Tree

Contention: All spinning local

Parallelism: Potential $n / \log n$ speedup
Combining Trees
Combining Trees

0

+3

(c) 2003-2005 Herlihy and Shavit
Combining Trees

(c) 2003-2005 Herlihy and Shavit
Combining Trees

Two threads meet, combine sums

(c) 2003-2005 Herlihy and Shavit
Combining Trees

Two threads meet, combine sums

(c) 2003-2005 Herlihy and Shavit
Combining Trees

Combined sum added to root

(c) 2003-2005 Herlihy and Shavit
Combining Trees

Result returned to children

(c) 2003-2005 Herlihy and Shavit
Combining Trees

Results returned to threads

(c) 2003-2005 Herlihy and Shavit
Devil in the Details

• What if
  - threads don’t arrive at the same time?

• Wait for a partner to show up?
  - How long to wait?
  - Waiting times add up …

• Instead
  - Use multi-phase algorithm
  - Try to wait in parallel
Combining Status

```c
enum CStatus{
    IDLE, FIRST, SECOND, DONE, ROOT};
```
Combining Status

```c
enum CStatus{
    IDLE,
    FIRST, SECOND, DONE, ROOT
};
```

Nothing going on
Combining Status

```c
enum CStatus{
    IDLE, FIRST, SECOND, DONE, ROOT;
};
```

1\textsuperscript{st} thread ISO partner for combining, will return soon to check for 2\textsuperscript{nd} thread
Combining Status

```c
enum CStatus{
    IDLE, FIRST, SECOND, DONE, ROOT};
```

2\textsuperscript{nd} thread arrived with value for combining
Combining Status

```
enum CStatus{
    IDLE, FIRST, SECOND, DONE, ROOT);
```

1\textsuperscript{st} thread has completed operation & deposited result for 2\textsuperscript{nd} thread
Combining Status

enum CStatus{
  IDLE, FIRST, SECOND, DONE, ROOT\};

Special case: root node
Node Synchronization

• Short-term
  - Synchronized methods
  - Consistency during method call

• Long-term
  - Boolean locked field
  - Consistency across calls
Phases

• Precombining
  - Set up combining rendez-vous

• Combining
  - Collect and combine operations

• Operation
  - Hand off to higher thread

• Distribution
  - Distribute results to waiting threads
Precombining Phase

Examine status

IDLE
Precombining Phase

0

FIRST

If IDLE, promise to return to look for partner
Precombining Phase

At ROOT, turn back
Precombining Phase

0

FIRST
Precombining Phase

If FIRST, I'm willing to combine, but lock for now
Code

- Tree class
  - In charge of navigation
- Node class
  - Combining state
  - Synchronization state
  - Bookkeeping
Precombining Navigation

Node node = myLeaf;
while (node.precombine()) {
    node = node.parent;
}
Node stop = node;
Precombining Navigation

```java
Node node = myLeaf;
while (node.precombine()) {
    node = node.parent;
}
Node stop = node;
```

Start at leaf
Precombining Navigation

Node node = myLeaf;

while (node.precombine()) {
    node = node.parent;
}

Node stop = node;

Move up while instructed to do so
Precombining Navigation

Node node = myLeaf;
while (node.precombine()) {
    node = node.parent;
}

Node stop = node;

Remember where we stopped
Precombining Node

```java
synchronized boolean precombine() {
    while (locked) wait();

    switch (cStatus) {
        case IDLE: cStatus = CStatus.FIRST;
                    return true;
        case FIRST: locked = true;
                    cStatus = CStatus.SECOND;
                    return false;
        case ROOT: return false;
        default: throw new PanicException()
    }
}
```
Precombining Node

```java
synchronized boolean precombine() {
    while (locked) wait();
    switch (cStatus) {
        case IDLE: cStatus = CStatus.FIRST;
                    return true;
        case FIRST: locked = true;
                    cStatus = CStatus.SECOND;
                    return false;
        case ROOT: return false;
        default: throw new PanicException();
    }
}
```

Short-term synchronization
Synchronization

```java
synchronized boolean precombine() {
    while (locked) wait();
    switch (cStatus) {
        case IDLE: cStatus = CStatus.FIRST;
                   return true;
        case FIRST: locked = true;
                   cStatus = CStatus.SECOND;
                   return false;
        case ROOT: return false;
        default: throw new PanicException();
    }
}
```

Wait while node is locked
Precombining Node

```java
synchronized boolean precombine() {
    while (locked) wait();
    switch (cStatus) {
    case IDLE:
        cStatus = CStatus.FIRST;
        return true;
    case FIRST:
        locked = true;
        cStatus = CStatus.SECOND;
        return false;
    case ROOT:
        return false;
    default:
        throw new PanicException();
    }
}
```

Check combining status
Node was IDLE

```java
synchronized boolean precombine() {
    while (locked) {wait();}
    switch (cStatus) {
    case IDLE: cStatus = CStatus.FIRST;
                return true;
    case FIRST: locked = true;
                cStatus = CStatus.SECOND;
                return false;
    case ROOT: return false;
    default: throw new PanicException();
    }
}
```

I will return to look for combining value
Precombining Node

```java
synchronized boolean precombine() {
    while (locked) {wait();}
    switch (cStatus) {
        case IDLE: cStatus = CStatus.FIRST;
                   return true;
        case FIRST: locked = true;
                   cStatus = CStatus.SECOND;
                   return false;
        case ROOT: return false;
        default: throw new PanicException()
    }
}
```

Continue up the tree
I'm the 2\textsuperscript{nd} Thread

synchronized boolean precombine() {
    while (locked) {wait();}
    switch (cStatus) {
        case IDLE: cStatus = CStatus.FIRST;
            return true;
        case FIRST: locked = true;
            cStatus = CStatus.SECOND;
            return false;
        case ROOT: return false;
        default: throw new PanicException();
    }
}

If 1\textsuperscript{st} thread has promised to return, lock node so it won't leave without me
Precombining Node

```java
synchronized boolean precombine() {
    while (locked) {wait();}
    switch (cStatus) {
    case IDLE:
        cStatus = CStatus.FIRST;
        return true;
    case FIRST:
        locked = true;
        cStatus = CStatus.SECOND;
        return false;
    case ROOT:
        return false;
    default:
        throw new PanicException();
    }
}
```

Prepare to deposit 2nd value
Precombining Node

End of phase 1, don't continue up tree

```java
while (sStatus == SStatus.BUSY)
    wait();

switch (cStatus) {
    case IDLE:
        cStatus = CStatus.FIRST;
        return true;
    case FIRST:
        locked = true;
        cStatus = CStatus.SECOND;
        return false;
    case ROOT:
        return false;
    default:
        throw new PanicException();
}
```
Node is the Root

If root, phase 1 ends, don't continue up tree

```java
switch (cStatus) {
    case IDLE: cStatus = CStatus.FIRST;
               return true;
    case FIRST: locked = true;
               cStatus = CStatus.SECOND;
               return false;
    case ROOT: return false;
    default: throw new PanicException();
}
```
synchronized boolean phase1() {
    while (locked) {wait();}
    switch (cStatus) {
        case IDLE: cStatus = CStatus.FIRST;
                   return true;
        case FIRST: locked = true;
                   cStatus = CStatus.SECOND;
                   return false;
        case ROOT: return false;
        default: throw new PanicException()
    }
}
Combining Phase

1\textsuperscript{st} thread locked out until 2\textsuperscript{nd} provides value

(c) 2003-2005 Herlihy and Shavit
Combining Phase

2\textsuperscript{nd} thread deposits value to be combined, unlocks node, & waits ...

(c) 2003-2005 Herlihy and Shavit
Combining Phase

1\textsuperscript{st} thread moves up the tree with combined value ...

(c) 2003-2005 Herlihy and Shavit
Combining (reloaded)

0

FIRST

2\textsuperscript{nd} thread has not yet deposited value ...

(c) 2003-2005 Herlihy and Shavit
Combining (reloaded)

1st thread is alone, locks out late partner
Combining (reloaded)

Stop at root

FIRST

(c) 2003-2005 Herlihy and Shavit
Combining (reloaded)

FIRST

$0$

$2^{nd}$ thread's phase

1 visit locked out
Combining Navigation

```java
node = myLeaf;
int combined = 1;
while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}
```
Combining Navigation

```java
node = myLeaf;
int combined = 1;
while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}
```

Start at leaf
Combining Navigation

node = myLeaf;

int combined = 1;

while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}

Add 1
Combining Navigation

```java
node = myLeaf;
int combined = 1;
while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}

Revisit nodes visited in phase 1
```
Combining Navigation

```java
node = myLeaf;
int combined = 1;
while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}
```

Accumulate combined values, if any
Combining Navigation

We will re-traverse the path in reverse order ...

```
node = myLeaf;
int combined = 1;
while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}
```
Combining Navigation

```java
node = myLeaf;
int combined = 1;
while (node != stop) {
    combined = node.combine(combined);
    stack.push(node);
    node = node.parent;
}
```

Move up the tree
Combining Phase Node

```java
synchronized int combine(int combined) {
    while (locked) wait();
    locked = true;
    firstValue = combined;
    switch (cStatus) {
        case FIRST:
            return firstValue;
        case SECOND:
            return firstValue + secondValue;
        default: ...
    }
}
```
Combining Phase Node

```java
synchronized int combine(int combined) {
    while (locked) wait();
    locked = true;
    firstValue = combined;
    switch (cStatus) {
    case FIRST:
        return firstValue;
    case SECOND:
        return firstValue + secondValue;
    default: ...
    }
}
```

Wait until node is unlocked
Combining Phase Node

synchronized int combine(int combined) {
    while (locked) wait();
    locked = true;
    firstValue = combined;
    switch (cStatus) {
        case FIRST:
            return firstValue;
        case SECOND:
            return firstValue + secondValue;
        default: ...
    }
}
synchronized int combine(int combined) {
    while (locked) wait();
    locked = true;
    firstValue = combined;
    switch (cStatus) {
        case FIRST:
            return firstValue;
        case SECOND:
            return firstValue + secondValue;
        default: ... 
    }
}
Combining Phase Node

synchronized int combine(int combined) {
    while (locked) wait();
    locked = true;
    firstValue = combined;
    switch (cStatus) {
        case FIRST:
            return firstValue;
        case SECOND:
            return firstValue + secondValue;
        default: ...
    }
}

Check status
synchronized int combine(int combined) {
    while (locked) wait();
    locked = true;
    firstValue = combined;
    switch (cStatus) {
        case FIRST:
            return firstValue;
        case SECOND:
            return firstValue + secondValue;
        default: ...
    }
}
Combining Node

synchronized int combine(int combined) {
    while (locked) wait();
    locked = true;
    firstValue = combined;
    switch (cStatus) {
    case FIRST:
        return firstValue;
    case SECOND:
        return firstValue + secondValue;
    default: ...
    }
}
Operation Phase

Add combined value to root, start back down (phase 4)
Operation Phase (reloaded)

5

SECOND 2

Leave value to be combined ...
Operation Phase (reloaded)

Unlock, and wait ...

(c) 2003-2005 Herlihy and Shavit
Operation Phase Navigation

\[
prior = \text{stop.op(combined)};
\]
Operation Phase Navigation

```cpp
prior = stop.op(combined);
```

Get result of combining
synchronized int op(int combined) {
    switch (cStatus) {
        case ROOT: int oldValue = result;
                    result += combined;
                    return oldValue;
        case SECOND: secondValue = combined;
                        locked = false; notifyAll();
                        while (cStatus != CStatus.DONE) wait();
                        locked = false; notifyAll();
                        cStatus = CStatus.IDLE;
                        return result;
        default: ...
    }
}
synchronized int op(int combined) {
    switch (cStatus) {
        case ROOT: int oldValue = result;
            result += combined;
            return oldValue;
        case SECOND: secondValue = combined;
            locked = false; notifyAll();
            while (cStatus != CStatus.DONE) wait();
            locked = false; notifyAll();
            cStatus = CStatus.IDLE;
            return result;
        default: ... 
    }
}

At Root

Add sum to root, return prior value
synchronized int op(int combined) {
    switch (cStatus) {
    case ROOT: int oldValue = result;
        result += combined;
        return oldValue;
    case SECOND: secondValue = combined;
        locked = false; notifyAll();
        while (cStatus != CStatus.DONE) wait();
        locked = false; notifyAll();
        cStatus = CStatus.IDLE;
        return result;
    default: ...
    }
}
synchronized int op(int combined) {
    switch (cStatus) {
    case ROOT: int oldValue = result;
                result += combined;
                return oldValue;
    case SECOND: secondValue = combined;
                locked = false; notifyAll();
                while (cStatus != CStatus.DONE) wait();
                locked = false; notifyAll();
                cStatus = CStatus.IDLE;
                return result;
    default: ...
synchronized int op(int combined) {
    switch (cStatus) {
        case ROOT: int oldValue = result;
            result += combined;
            return oldValue;
        case SECOND: secondValue = combined,
            locked = false; notifyAll();
            while (cStatus != CStatus.DONE) wait();
            locked = false; notifyAll();
            cStatus = CStatus.IDLE;
            return result;
        default: ...
    }
}
synchronized int op(int combined) {
    switch (cStatus) {
        case ROOT: int oldValue = result;
            result += combined;
            return oldValue;
        case SECOND: secondValue = combined;
            locked = false; notifyAll();
            while (cStatus != CStatus.DONE) wait();
            locked = false; notifyAll();
            cStatus = CStatus.IDLE;
            return result;
        default: ...
    }
}
Distribution Phase

0

SECOND

5

Move down with result

ZZZ
Distribution Phase

Leave result for 2\textsuperscript{nd} thread & lock node
Distribution Phase

Move result back down tree
Distribution Phase

2nd thread awakens, unlocks, takes value
Distribution Phase Navigation

```java
while (!stack.empty()) {
    node = stack.pop();
    node.distribute(prior);
}
return prior;
```
while (!stack.empty()) {
    node = stack.pop();
    node.distribute(prior);
}
return prior;

Traverse path in reverse order
Distribution Phase Navigation

while (!stack.empty()) {
    node = stack.pop();
    node.distribute(prior);
}
return prior;

Distribute results to waiting 2\textsuperscript{nd} threads
Distribution Phase Navigation

while (!stack.empty()) {
    node = stack.pop();
    node.distribute(prior);
}

return prior;

Return result to caller
Distribution Phase

synchronized void distribute(int prior) {
    switch (cStatus) {
    case FIRST:
        cStatus = CStatus.IDLE;
        locked = false; notifyAll();
        return;
    case SECOND:
        result = prior + firstValue;
        cStatus = CStatus.DONE; notifyAll();
        return;
    default: ...
    }
synchronized void distribute(int prior) {
    switch (cStatus) {
    case FIRST:
        cStatus = CStatus.IDLE;
        locked = false; notifyAll();
        return;
    case SECOND:
        result = prior + firstValue;
        cStatus = CStatus.DONE; notifyAll();
        return;
    default: ...
    }
Distribution Phase

synchronized void distribute(int prior) {
   switch (cStatus) {
      case FIRST:
         cStatus = CStatus.IDLE;
         locked = false; notifyAll();
         return;
      case SECOND:
         result = prior + firstValue;
         cStatus = CStatus.DONE; notifyAll();
         return;
      default: ...
   }
}

Notify 2\textsuperscript{nd} thread that result is available
Bad News: High Latency

Log n

+2

+5

+3
Good News: Real Parallelism

+5

+2

+3

1 thread

2 threads

(c) 2003-2005 Herlihy and Shavit
Throughput Puzzles

• **Ideal circumstances**
  - All $n$ threads move together, combine
  - $n$ increments in $O(\log n)$ time

• **Worst circumstances**
  - All $n$ threads slightly skewed, locked out
  - $n$ increments in $O(n \cdot \log n)$ time
Index Distribution Benchmark

```java
void indexBench(int iters, int work) {
    while (int i < iters) {
        i = r.getAndIncrement();
        Thread.sleep(random() % work);
    }
}
```
Index Distribution Benchmark

```java
void indexBench(int iters, int work) {
    while (int i < iters) {
        i = r.getAndIncrement();
        Thread.sleep(random() % work);
    }
}
```

How many iterations
Index Distribution Benchmark

```java
void indexBench(int iters, int work) {
    while (int i < iters) {
        i = r.getAndIncrement();
        Thread.sleep(random() % work);
    }
}
```

Expected time between incrementing counter
Index Distribution Benchmark

```java
void indexBench(int iters, int work) {
    while (int i < iters) {
        i = r.getAndIncrement();
        Thread.sleep(random() % work);
    }
}
```

Take a number
Index Distribution Benchmark

void indexBench(int iters, int work) {
    while (int i < iters) {
        i = r.getAndIncrement();
        Thread.sleep(random() % work);
    }
}

Pretend to work
(more work, less concurrency)
Performance Benchmarks

- **Alewife**
  - NUMA architecture
  - Simulated

- **Throughput:**
  - average number of `inc` operations in 1 million cycle period.

- **Latency:**
  - average number of simulator cycles per `inc` operation.
Performance

Latency:

Throughput:

work = 0
Performance

Latency:

Throughput:

cycles per operation

operations per million cycles

work = 0

(c) 2003-2005 Herlihy and Shavit
The Combining Paradigm

• Implements any RMW operation

• When tree is loaded
  – Takes $2 \log n$ steps
  – for $n$ requests

• Very sensitive to load fluctuations:
  – if the arrival rates drop
  – the combining rates drop
  – overall performance deteriorates!
Combining Load Sensitivity

![Graph showing the relationship between throughput and processors with notice of load fluctuations.](c) 2003-2005 Herlihy and Shavit
Combining Rate vs Work
Better to Wait Longer
Conclusions

• Combining Trees
  - Work well under high contention
  - Sensitive to load fluctuations
  - Can be used for getAndMumble() ops

• Next
  - Counting networks
  - A different approach ...
A Balancer

Input wires

Output wires
Tokens Traverse Balancers

- Token $i$ enters on any wire
- leaves on wire $i \mod \text{fan-out}$
Tokens Traverse Balancers
Tokens Traverse Balancers
Tokens Traverse Balancers
Tokens Traverse Balancers
Tokens Traverse Balancers

Arbitrary input distribution

Balanced output distribution
Smoothing Network

k-smooth property
Counting Network

step property
Counting Networks Count!

- 0, 4, 8.....
- 1, 5, 9.....
- 2, 6, 10.....
- 3, 7 .........
Bitonic[4]
Bitonic[4]
Bitonic[4]
Bitonic[4]
Bitonic[4]
Bitonic[4]
Counting Networks

- Good for counting number of tokens
- low contention
- no sequential bottleneck
- high throughput $\log^2 n$
- practical networks depth
Bitonic[k] is 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
Shared Memory Implementation

class balancer {
    boolean toggle;
    balancer[] next;

    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
class balancer {
    boolean toggle;
    balancer[] next;

    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
class balancer {
    boolean toggle;
    balancer[] next;

    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
class balancer {
    boolean toggle;
    balancer[] next;

    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
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 get to the end
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;
}
Balancer traverse (Balancer b) {
while(!b.isLeaf()) {
    boolean toggle = b.flip();
    if (toggle)
        b = b.next[0]
    else
        b = b.next[1]
    return b;
}
Alternative Implementation:
Message-Passing
Bitonic[2k] Schematic
Bitonic[2k] Layout
Unfolded Bitonic Network
Unfolded Bitonic Network

Merger[8]
Unfolded Bitonic Network
Unfolded Bitonic Network
Unfolded Bitonic Network
Unfolded Bitonic Network

Merger[2]
Merger[2]
Merger[2]
Merger[2]
Bitonic\([k]\) Depth

- **Width** \(k\)
- **Depth** is \((\log_2 k)(\log_2 k + 1)/2\)
Merger[2k]
Merger[2k] Schematic
Merger[2k] Layout
Lemma

If a sequence has the step property ...
Lemma

So does its even subsequence
Lemma

And its odd subsequence
Merger[2k] Schematic
Proof Outline

Outputs from Bitonic[k]        Inputs to Merger[k]

(c) 2003-2005 Herlihy and Shavit
Proof Outline

Inputs to Merger[k]  Outputs of Merger[k]

even  even
odd  odd
odd  even
Proof Outline

Outputs of Merger[k]  Outputs of last layer
Periodic Network Block
Periodic Network Block
Periodic Network Block
Periodic Network Block
Block[2k] Schematic
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 $(\log w)$.

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

Higher is better!

- 64-leaf combining tree
- 80-balancer counting network
- MCS queue lock
- Spin lock

* All graphs taken from Herlihy, Lim, Shavit, copyright ACM.

(c) 2003-2005 Herlihy and Shavit
Performance (Simulated)

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

Combining and counting are pretty close

Throughput vs. Number of processors

MCS queue lock
Spin lock

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

But they beat the hell out of the competition!

Throughput

Number processors

64-leaf combining tree
80-balancer counting network

MCS queue lock
Spin lock

* All graphs taken from Herlihy, Lim, Shavit, copyright ACM.
Saturation and Performance

Undersaturated \( P < w \log w \)

Optimal performance

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

- Put counting network
- Remove counting network
Put/Remove Network

- **Guarantees never:**
  - Put waiting for item, while
  - Get has deposited item

- **Otherwise OK to wait**
  - Put delayed while pool slot is full
  - Get delayed while pool slot is empty
What About

- Decrements
- 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
Tokens & Anti-Tokens Cancel
Tokens & Anti-Tokens Cancel
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 \(-1\) tokens on wire i
  - So network still in legal state

• Moreover, network width \(w\) divides \_
  - So \(-1\) tokens
Before Antitoken

(c) 2003-2005 Herlihy and Shavit
Balancer states as if ...

\[-1\] 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

Takes 0

Takes 0

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 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
Clip Art