Introduction to Multicores and Parallelism

Guest Lecture by
Nir Shavit

Based on Companion slides for “The Art of Multiprocessor Programming”
Moore’s Law

Transistor count still rising

Clock speed flattening sharply
Vanishing from your Desktops: The Uniprocesor
Your Server: The Shared Memory Multiprocessor (SMP)
Your New Server or Desktop: The Multicore Processor (CMP)

All on the same chip

Sun T2000 Niagara
Multicores are everywhere

...Intel is selling 8 core 32 way Core i7 Nehalem processors...

...Oracle is selling 32 core 256 way Nahelem based systems
Why is Kunle Smiling?

Niagara 1
Why do we care?

• Time no longer cures software bloat
  – The “free ride” is over

• When you double your program’s path length
  – You can’t just wait 6 months
  – Your software must somehow exploit twice as much concurrency
Traditional Scaling Process

- **User code**
  - 1.8x
  - 3.6x
  - 7x

- **Time: Moore's law**
  - Traditional Uniprocessor

Art of Multiprocessor Programming
Ideal Scaling Process

Unfortunately, not so simple...
Actual Scaling Process

Speedup

1.8x

2x

2.9x

User code

Multicore

Parallelization and Synchronization require great care...
Sequential Computation

memory

object

object

thread
Concurrent Computation

memory

object

object

threads
Asynchrony

- Sudden unpredictable delays
  - Cache misses (short)
  - Page faults (long)
  - Scheduling quantum used up (really long)
Model Summary

- Multiple *threads*
  - Sometimes called *processes*
- Single shared *memory*
- *Objects* live in memory
- Unpredictable asynchronous delays
Concurrency Jargon

• Hardware
  – Processors

• Software
  – Threads, processes

• Sometimes OK to confuse them, sometimes not.
Parallel Primality Testing

• Challenge
  – Print primes from 1 to $10^{10}$

• Given
  – Ten-processor multiprocessor
  – One thread per processor

• Goal
  – Get ten-fold speedup (or close)
Load Balancing

- Split the work evenly
- Each thread tests range of $10^9$
Procedure for Thread $i$

```java
void primePrint {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*10^9+1, j<(i+1)*10^9; j++) {
        if (isPrime(j))
            print(j);
    }
}
```
Issues

• Higher ranges have fewer primes
• Yet larger numbers harder to test
• Thread workloads
  – Uneven
  – Hard to predict
Issues

• Higher ranges have fewer primes
• Yet larger numbers harder to test
• Thread workloads
  – Uneven
  – Hard to predict
• Need *dynamic* load balancing
Shared Counter

each thread takes a number
Procedure for Thread $i$

```java
int counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < $10^{10}$) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
```
Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < 10^{10}) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}
Where Things Reside

```c
void primePrint {
    int i = ThreadID.get(); // IDs in {0..9}
    for (j = i*10+1, j<(i+1)*10; j++) {
        if (isPrime(j))
            print(j);
    }
}
```
Procedure for Thread $i$

Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < $10^{10}$) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}

Stop when every value taken
Procedure for Thread $i$

Counter counter = new Counter(1);

void primePrint {
    long j = 0;
    while (j < $10^{10}$) {
        j = counter.getAndIncrement();
        if (isPrime(j))
            print(j);
    }
}

Increment & return each new value
Counter Implementation

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```
Counter Implementation

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}
```

OK for single thread, not for concurrent threads.
What It Means

public class Counter {
    private long value;

    public long getAndIncrement() {
        return value++;
    }
}

What It Means

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        long temp = value;
        value = temp + 1;
        return temp;
    }
}
```
Not so good…

Value... 1 2 3 2

read write read write
1 2 2 3

read write
1 2

time
Is this problem inherent?

If we could only glue reads and writes together...
public class Counter {
    private long value;

    public long getAndIncrement() {
        temp = value;
        value = temp + 1;
        return temp;
    }
}
Challenge

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        long temp = value;
        value = temp + 1;
        return temp;
    }
}
```

Make these steps *atomic* (indivisible)
Hardware Solution

public class Counter {
    private long value;

    public long getAndIncrement() {
        long temp = value;
        value = temp + 1;
        return temp;
    }
}

ReadModifyWrite() instruction
In Java™

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp  = value;
            value = temp + 1;
        }
        return temp;
    }
}
In Java™

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp  = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

Mutual Exclusion
Mutual Exclusion or “Alice & Bob want to share a pond”
Alice has a pet
Bob has a pet
The Problem

The pets don't get along
Formalizing the Problem

• Two types of formal properties in asynchronous computation:
  • Safety Properties
    – Nothing bad happens ever
  • Liveness Properties
    – Something good happens eventually
Formalizing our Problem

• Mutual Exclusion
  – Both pets never in pond simultaneously
  – This is a safety property

• No Deadlock
  – if only one wants in, it gets in
  – if both want in, one gets in.
  – This is a liveness property
Simple Protocol

• Idea
  – Just look at the pond

• Gotcha
  – Not atomic
  – Trees obscure the view
Interpretation

• Threads can’t “see” what other threads are doing
• Explicit communication required for coordination
Cell Phone Protocol

• Idea
  – Bob calls Alice (or vice-versa)

• Gotcha
  – Bob takes shower
  – Alice recharges battery
  – Bob out shopping for pet food …
Interpretation

• Message-passing doesn’t work
• Recipient might not be
  – Listening
  – There at all
• Communication must be
  – Persistent (like writing)
  – Not transient (like speaking)
Can Protocol
Bob conveys a bit

Art of Multiprocessor Programming
Bob conveys a bit
Can Protocol

• Idea
  – Cans on Alice’s windowsill
  – Strings lead to Bob’s house
  – Bob pulls strings, knocks over cans

• Gotcha
  – Cans cannot be reused
  – Bob runs out of cans
Interpretation

• Cannot solve mutual exclusion with interrupts
  – Sender sets fixed bit in receiver’s space
  – Receiver resets bit when ready
  – Requires unbounded number of interrupt bits
Flag Protocol
Alice’s Protocol (sort of)
Bob’s Protocol (sort of)
Alice’s Protocol

- Raise flag
- Wait until Bob’s flag is down
- Unleash pet
- Lower flag when pet returns
Bob’s Protocol

• Raise flag
• Wait until Alice’s flag is down
• Unleash pet
• Lower flag when pet returns
Bob’s Protocol (2nd try)

• Raise flag
• While Alice’s flag is up
  – Lower flag
  – Wait for Alice’s flag to go down
  – Raise flag
• Unleash pet
• Lower flag when pet returns
Bob’s Protocol

- Raise flag
- While Alice’s flag is up
  - Lower flag
  - Wait for Alice’s flag to go down
  - Raise flag
- Unleash pet
- Lower flag when pet returns

Bob defers to Alice
The Flag Principle

- Raise the flag
- Look at other’s flag
- Flag Principle:
  - If each raises and looks, then
  - Last to look must see both flags up
Proof of Mutual Exclusion

• Assume both pets in pond
  – Derive a contradiction
  – By reasoning backwards

• Consider the last time Alice and Bob each looked before letting the pets in

• Without loss of generality assume Alice was the last to look…
Alice's last look
Bob's last look

Bob last raised flag

Alice last raised her flag
Alice's last look

Alice must have seen Bob's Flag. A Contradiction

QED
Proof of No Deadlock

• If only one pet wants in, it gets in.
Proof of No Deadlock

- If only one pet wants in, it gets in.
- Deadlock requires both continually trying to get in.
Proof of No Deadlock

• If only one pet wants in, it gets in.
• Deadlock requires both continually trying to get in.
• If Bob sees Alice’s flag, he gives her priority (a gentleman…)

QED
Remarks

• Protocol is *unfair*
  – Bob’s pet might never get in

• Protocol uses *waiting*
  – If Bob is eaten by his pet, Alice’s pet might never get in
Moral of Story

• Mutual Exclusion cannot be solved by
  – transient communication (cell phones)
  – interrupts (cans)

• It can be solved by
  – one-bit shared variables
  – that can be read or written
  – with waiting
The Arbiter Problem (an aside)
Why did we use Mutual Exclusion only on counter?

```java
public class Counter {
    private long value;

    public long getAndIncrement() {
        synchronized {
            temp = value;
            value = temp + 1;
        }
        return temp;
    }
}
```

Synchronized block
Mutual Exclusion Implies Sequential Execution

• We want as much of the code as possible to execute in parallel
• A larger sequential part implies reduced performance
• Amdahl’s law: this relation is not linear…
Amdahl’s Law

\[
\text{Speedup} = \frac{\text{OldExecutionTime}}{\text{NewExecutionTime}}
\]

...of computation given \( n \) CPUs instead of 1
Amdahl’s Law

\[ \text{Speedup} = \frac{1}{1 - p + \frac{p}{n}} \]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]
Amdahl’s Law

\[
\text{Speedup} = \frac{1}{1 - p + \frac{p}{n}}
\]
Amdahl’s Law

Speedup = \frac{1}{1 - p + \frac{p}{n}}

Sequential fraction

Parallel fraction

Number of processors
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 60% concurrent, 40% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = \frac{1}{1 - 0.6 + \frac{0.6}{10}} = 2.17
\]
Example

- Ten processors
- 80\% concurrent, 20\% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 80% concurrent, 20% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 3.57 = \frac{1}{1 - 0.8 + \frac{0.8}{10}}
\]
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 90% concurrent, 10% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 5.26 = \frac{1}{1 - 0.9 + \frac{0.9}{10}}
\]
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?
Example

- Ten processors
- 99% concurrent, 01% sequential
- How close to 10-fold speedup?

\[
\text{Speedup} = 9.17 = \frac{1}{1 - 0.99 + \frac{0.99}{10}}
\]
Back to Real-World Multicore Scaling

User code

Multicore

Not reducing sequential % of code
Why?

Amdahl’s Law:

Pay for N = 8 cores
SequentialPart = 25%

As num cores grows the effect of 25% becomes more acute
2.3/4, 2.9/8, 3.4/16, 3.7/32....
Shared Data Structures

The reason fine grained parallelism has huge performance benefit.
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
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
Simple Locking Implementation

Problem: sequential bottleneck
Simple Locking Implementation

Problem: sequential bottleneck
Counting Implementation

put

remove
Counting Implementation

Only the counters are sequential
Shared Counter
Shared Counter

No duplication
Shared Counter

- No duplication
- No Omission
Can we design a parallel shared counter?

Multiple counters introduce parallelism and distribute load.
Can we design a parallel shared counter?

random

- $\rightarrow 0, 4, 8\ldots$
- $\rightarrow 1, 5, 9\ldots$
- $\rightarrow 2, 6, \ldots$
- $\rightarrow 3, 7 \ldots$
Solution: Counting Networks

counters

0, 4, 8…..
1, 5, 9…..
2, 6, …
3, 7 …
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, ...
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
- no sequential bottleneck
- high throughput
- practical networks depth $\log^2 n$
Counting Network
Counting Network
Counting Network
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)
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;
    }
}
class balancer {
    boolean toggle;
    balancer[] next;
    synchronized boolean flip() {
        boolean oldValue = this.toggle;
        this.toggle = !this.toggle;
        return oldValue;
    }
}
Shared Memory Implementation

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

  boolean toggle;
  balancer[] next;

  getAndComplement

  }
Shared Memory Implementation

```java
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;
    }
    Stop when we exit the network
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

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

Exit on wire
Alternative Implementation: Message-Passing

Art of Multiprocessor Programming
Bitonic[2k] Inductive Structure

Bitonic[2]

Bitonic[2]

Merger[4]
Bitonic[8] Layout

Bitonic[4]

Merger[8]
Unfolded Bitonic[8] Network

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

Merger[2]
Merger[2]
Merger[2]
Merger[2]
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

Odd + even
Bitonic[2k] Layout Details

Bitonic[k]

Merger[k]

Merger[2k]
By induction hypothesis

Outputs have step property
By Lemma 1

All subsequences have step property
By Lemma 2

Diff at most 1
By Induction Hypothesis

Outputs have step property
Remember: by Lemma 2

At most one diff

Merger[k]

Merger[k]
Last Row of Balancers

Merger[k]

Merger[k]
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]
Last Row of Balancers

Merger[k]

Merger[k]

Subsequences always look the same except one token
Last Row of Balancers

Subsequences that look the same before balancers merge evenly after
Extra odd token falls on a separate balancer and will be pushed up
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)

Higher is better!

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

Throughput

Number processors

MCS queue lock
Spin lock
Performance (Simulated)

Combining and counting are pretty close

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

Throughput

Number processors

- MCS queue lock
- Spin lock
Performance (Simulated)

But they beat the hell out of the competition!

MCS queue lock
Spin lock
Saturation and Performance

Undersaturated \( P < w \log w \)

Optimal performance

Saturated \( P = w \log w \)

Oversaturated \( P > w \log w \)
Throughput vs. Size

![Graph showing throughput vs. number of processors for different sizes of Bitonic algorithms: Bitonic[4], Bitonic[8], Bitonic[16].]
Shared Pool

put
remove
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 $\Omega - 1$ 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 ... 

\( \Omega - 1 \) 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
- \texttt{getAndIncrement} + \texttt{getAndDecrement} methods
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 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!
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?

put

remove
Counting Trees

A Tree Balancer:

Single input wire

Step property in quiescent state
Counting Trees

Interleaving of output wires
Lemma: Tree[2k] has step property in quiescent state.

Tree[2k] = y_0 + b + y_1

At most 1 more token in top wire

k even outputs

k odd outputs
Inductive Construction

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

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

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

\[ \text{inc} = \text{follow getAndComplement of toggle-bits} \]
Implementing Counting Trees

problem: **toggle bit**
in balancer..

Contention and
Sequential bottleneck…so what have
we achieved?

To lesser extent in
next balancers
Idea (as in elimination stack): if an even number of tokens pass balancer, the toggle bit remains unchanged!
Lemma: **Diffracting balancer same as balancer.**
Diffracting Tree

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

High Throughput with Low Contention
Performance

Throughput

Latency

P=Concurrency

MCS

Dtree

Ctree
Summary

- Multicores offer great potential for speedup of code execution times.
- Making use of multicores requires reducing size of sequential (mutually exclusive) parts of code.
- Can use modern combinatorial techniques to design highly parallel and distributed solutions.
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.