Concurrency & Parallelism

Nick Sumner
wsumner@sfu.ca
Concurrency & Parallelism
Seeking out performance

- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism
Seeking out performance

- Improving performance can come from tuning
  - Algorithmic complexity
  - Memory access patterns
  - Concurrency
  - Parallelism

- As processor speeds have slowed increasing, much focus has been placed on the last two
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrent & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.
Concurrenty & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

- **Parallelism** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions
Concurrency & Parallelism

- **Concurrency** is the management of multiple tasks at the same time.
  - e.g. Sharing a CPU across multiple processes.

- **Parallelism** is using multiple resources to perform multiple tasks at the same time.
  - e.g. multiple cores for tasks, vector instructions
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.

- The best possible running time is determined by the critical path or span of dependent tasks through the program.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combined.

- The best possible running time is determined by the *critical path* or *span* of dependent tasks through the program.
Using Parallelism

- Large problems can sometimes be split into parallel tasks, and the effects of the parallel tasks combine.

This is too optimistic! Why?

- The best possible running time is determined by the critical path or span of dependent tasks through the program.
Using Parallelism

- There are often more tasks than compute resources
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits

\[
\text{Given } p \text{ processors, } \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty
\]
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits

\[
\text{Given } p\text{ processors, } \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty
\]

- Identifying good opportunities for effective parallelism is open to research
Using Parallelism

- There are often more tasks than compute resources
  - Brent’s Theorem describes the time accounting for limits

\[
\text{Given } p \text{ processors}, \frac{\text{Time}_1}{p} \leq \text{Time}_p \leq \frac{\text{Time}_1}{p} + \text{Time}_\infty
\]

- Identifying good opportunities for effective parallelism is open to research
  - Profiling for tasks to extract
  - Understanding the effect of speeding specific tasks
  - ...
Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others
Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others

- Specifically
  - Deadlock / Livelock
  - Starvation
  - Data races
  - Atomicity violations
  - Order violations
  - ...

Correctness issues

- Parallel & concurrent code is challenging to write
  - Nondeterministic timing
  - Actions of one task may subtly affect others

- Specifically
  - Deadlock / Livelock
  - Starvation
  - Data races
  - Atomicity violations
  - Order violations
  - ...

97% of real world concurrency bugs

[Lu, ASPLOS 2008]
Data Races

- A data race occurs when:
Data Races

A data race occurs when:
1) two threads access the same location
Data Races

• A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
Data Races

- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
  3) at least one access is a write (WAW, WAR, RAR)
Data Races

- A data race occurs when:
  1) two threads access the same location
  2) the accesses are *logically simultaneous*
  3) at least one access is a write *(WAW, WAR, RAR)*
**Data Races**

```
x++
tmp = x
tmp = tmp+1
x = tmp
```

```
tmp \_1 = x
tmp \_1 = tmp \_1 + 1
x = tmp \_1
```

```
tmp \_2 = x
 tmp \_2 = tmp \_2 + 1
x = tmp \_2
```
Data Races

```
x++
tmp = x
tmp = tmp + 1
x = tmp
```

T1  T2

```
tmp_1 = x
tmp_1 = tmp_1 + 1
x = tmp_1
```

```
tmp_2 = x
tmp_2 = tmp_2 + 1
x = tmp_2
```

T1  T2

```
tmp_1 = x
tmp_1 = tmp_1 + 1
x = tmp_1
```

```
tmp_2 = x
tmp_2 = tmp_2 + 1
x = tmp_2
```
Data Races

Synchronization discipline prevents data races.

tmp = x
tmp = tmp + 1
x = tmp

tmp1 = x
tmp1 = tmp1 + 1
x = tmp1

tmp2 = x
tmp2 = tmp2 + 1
x = tmp2

T1 T2

x++
“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless
“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless

- Both programming languages and hardware have memory models that determine what is *really* okay
“Benign” Data Races

- Sometimes a developer will make use of a data race
  - Avoid expensive synchronization
  - The race looks “benign” or harmless

- Both programming languages and hardware have memory models that determine what is really okay
  - A *memory model* determines what values may be read by a given memory access, esp. w.r.t. previous writes

[CACM 2010, PLDI 2018]
“Benign” Data Races

if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;

[Boehm, Hotpar 2011]
“Benign” Data Races

- Threads race on init
- The compiler assumes no races while optimizing

```java
if (!init) {
  lock();
  if (!init) {
    data = create();
    init = true;
  }
  unlock();
}
tmp = data;

[Boehm, Hotpar 2011]
```
“Benign” Data Races

-Threads race on \texttt{init}
-The compiler assumes no races while optimizing

\begin{verbatim}
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
} tmp = data;
\end{verbatim}

\begin{verbatim}
if (!init) {
    lock();
    if (!init) {
        init = true;
        data = create();
    }
    unlock();
} tmp = data;
\end{verbatim}

[Boehm, Hotpar 2011]
“Benign” Data Races

- Threads race on `init`
- The compiler assumes no races while optimizing

```java
if (!init) {
    lock();
    if (!init) {
        data = create();
        init = true;
    }
    unlock();
}
tmp = data;
```

[Boehm, Hotpar 2011]
“Benign” Data Races

local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}

[Boehm, Hotpar 2011]
“Benign” Data Races

- Data race freedom allows extra reads.

```c
local = counter;
if (local > localMax) {
    handler = ...;
}
update = work();
if (local > localMax) {
    handler(update);
}
```

[Boehm, Hotpar 2011]
“Benign” Data Races

- Races can introduce bugs on non-racy variables

\[
c = a + 10 \\
\ldots \\
b = a + 10 \\
c = 1
\]

[Dolan, PLDI 2018]
“Benign” Data Races

- Races can introduce bugs on non-racy variables

```
c = a + 10  
...  
b = a + 10
```

[Dolan, PLDI 2018]
"Benign" Data Races

- Races can jump forward and backward in time

```
[flag = true]
a = 1
f = flag
b = a
c = a
```

[Dolan, PLDI 2018]
“Benign” Data Races

Races can jump forward and backward in time

This can happen in Java when flag is volatile & b is a complex reference

[Dolan, PLDI 2018]
"Benign" Data Races

- Races can jump forward and backward in time

This can happen in Java when flag is volatile & b is a complex reference

2 can be read after 1 even in the same thread!

[Dolan, PLDI 2018]
Happens-Before Ordering

- Memory models are often specified using *Happens-Before* relations.
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: simultaneously)
  - defined behavior occurs when writes & reads are ordered
Happens-Before Ordering

• Memory models are often specified using Happens-Before relations.
  – a partial order over logical time (recall: *simultaneously*)
  – defined behavior occurs when writes & reads are ordered
  – lock/unlock, fork/join constrain order
  – access to volatile variables keeps per variable order
Happens-Before Ordering

- Memory models are often specified using Happens-Before relations.
  - a partial order over logical time (recall: simultaneously)
  - defined behavior occurs when writes & reads are ordered
  - lock/unlock, fork/join constrain order
  - access to volatile variables keeps per variable order

- Happens-Before ordering of a specific execution can be tracked to identify bugs
Happens-Before Ordering

T1  T2

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()
Happens-Before Ordering

T1    T2

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
Happens-Before Ordering

T1  T2

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()

lock()
tmp = x
tmp = tmp+1
x = tmp
unlock()
Happens-Before Ordering

T1  T2

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()  

tmp = x  
tmp = tmp+1  
x = tmp  

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()  

tmp = x  
tmp = tmp+1  
x = tmp
Happens-Before Ordering

T1  T2

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()  

T1  T2

lock()  
tmp = x  
tmp = tmp+1  
x = tmp  
unlock()  

No ordering!  Simultaneous!  Race detected!

tmp = x  
tmp = tmp+1  
x = tmp  

Happens-Before Ordering

- Note, this only detects races in the current execution!
  - *Sound predictive* data race detection can extend it across other executions [PLDI 2017/2018]
Happens-Before Ordering

- Note, this only detects races in the current execution!
  - *Sound predictive* data race detection can extend it across other executions [PLDI 2017/2018]

- Requires careful tracking of dependences
  - Careful construction of logical time using *vector clocks* [JVM 2001, PLDI 2009]
What is the happens-before relation?

Logical Time & Vector Clocks

T1  T2

lock(a)
write x
unlock(a)
write x

write y
lock(a)
read x
unlock(a)
Logical Time & Vector Clocks

What is the happens-before relation?

T1
- lock(a)
- write x
- unlock(a)
- write x

T2
- write y
- lock(a)
- read x
- unlock(a)
- unlock(a)
What is the happens-before relation?

T1
- lock(a)
- write x
- unlock(a)
- write x

T2
- write y
- lock(a)
- read x
- unlock(a)
- unlock(a)
Logical Time & Vector Clocks

What is the happens-before relation?

lock(a)
write x
unlock(a)
write y
lock(a)
read x
unlock(a)
write x
unlock(a)
What is the happens-before relation?

T1
- lock(a)
- write x
- unlock(a)
- write x

T2
- write y
- lock(a)
- read x
- unlock(a)
Logical Time & Vector Clocks

What is the happens-before relation?

lock(a)
write x
unlock(a)
write x

write y
lock(a)
read x
unlock(a)

T1   T2
Logical Time & Vector Clocks

\[ x \mapsto R[0,0] \]
\[ y \mapsto R[0,0] \]
\[ W[0,0] \]

lock(a)
write x
unlock(a)
write x
lock(a)
read x
unlock(a)
write y
lock(a)

T1 ↦ [0, 0]
T2 ↦ [0, 0]

Clocks

Shadow Memory

x ↦ R[0, 0]
y ↦ R[0, 0]
W[0, 0]
Logical Time & Vector Clocks

Logical time and vector clocks are used to model the behavior of concurrent processes. In the diagram, we have two processes, $T_1$ and $T_2$, with logical times represented as $[1,0]$ and $[0,0]$ respectively. The processes involve operations such as locking, writing, and unlocking variables.

- $lock(a)$
- $write(x)$
- $unlock(a)$
- $read(x)$

The shadow memory and clock operations are also shown in the diagram:

- $x \mapsto R[0,0]$ (read)
- $w \mapsto W[0,0]$ (write)

The clocks for the processes are:

- $T_1 \mapsto [1,0]$
- $T_2 \mapsto [0,0]$
Logical Time & Vector Clocks

T1  T2

[1,0]  [0,0]

lock(a)  write x  unlock(a)

write x

write y

lock(a)  read x  unlock(a)

Clocks

T1⇒[1,0]
T2⇒[0,0]

Shadow Memory

x⇒R[0,0]
W[1,0]
y⇒R[0,0]
W[0,0]
Logical Time & Vector Clocks

Clocks
- T1 $\rightarrow [2,0]$
- T2 $\rightarrow [0,0]$

Shadow Memory
- x $\leftrightarrow R[0,0]$
- y $\leftrightarrow R[0,0]$
- a $\leftrightarrow [1,0]$
- W[1,0]
- W[0,0]
Logical Time & Vector Clocks

Clocks

T1 \[\rightarrow [2,0] \]
T2 \[\rightarrow [0,0] \]

Shadow Memory

x \[\mapsto R[0,0] \]
W[1,0]
y \[\mapsto R[0,0] \]
W[0,0]
a \[\mapsto [1,0] \]

sendProgress(T1, a)
advanceLocal(T1)
Logical Time & Vector Clocks

T1  T2

Clocks

Shadow Memory

lock(a)  write x  unlock(a)

write y

lock(a)  read x  unlock(a)

write x

lock(a)  write x  unlock(a)

T1⇒[2, 0]
T2⇒[1, 1]

x⇒R[0, 0]
W[1, 0]
y⇒R[0, 0]
W[0, 0]
a⇒[1, 0]
Logical Time & Vector Clocks

Clocks

T1: \([2,0]\)

T2: \([1,1]\)

Shadow Memory

x: \(R[0,0]\)

w: \(W[1,0]\)

y: \(R[0,0]\)

a: \(W[0,0]\)

receiveProgress(T2, a)

advanceLocal(T2)
Logical Time & Vector Clocks

T1  T2

Clocks

Shadow Memory

lock(a)
write x
unlock(a)

write y
lock(a)
read x
unlock(a)

write x

[1,0] T1\rightarrow[2,0]
T2\rightarrow[1,1]

R[1,1] W[1,0]
y\mapsto R[0,0] W[0,0]

[2,0]
[0,0]
[1,0]
[1,1]
Logical Time & Vector Clocks

Clocks

\[ T_1 \mapsto [2, 0] \]
\[ T_2 \mapsto [1, 2] \]

Shadow Memory

\[ x \mapsto R[1, 1] \]
\[ y \mapsto R[0, 0] \]
\[ w \mapsto W[1, 0] \]
\[ w \mapsto W[0, 0] \]
Logical Time & Vector Clocks

Clocks

T1 $\rightarrow [2,0]$
T2 $\rightarrow [1,2]$

Shadow Memory

$X \mapsto R[1,1]$
$W[2,0]$

$Y \mapsto R[0,0]$
$W[0,0]$

write $x$
lock(a)
write $x$
unlock(a)

write $y$
lock(a)
read $x$
unlock(a)
**Logical Time & Vector Clocks**

- **Clocks**
  - $T_1 \mapsto [2, 0]$
  - $T_2 \mapsto [1, 2]$

- **Shadow Memory**
  - $x \mapsto R[1,1]$
  - $y \mapsto R[0,0]$
  - $W[2,0]$
  - $W[0,0]$

- **Event Sequence**
  - $\text{lock}(a)$
  - $\text{write } x$
  - $\text{unlock}(a)$
  - $\text{write } y$
  - $\text{lock}(a)$
  - $\text{read } x$
  - $\text{unlock}(a)$

- **Inference**
  - $C(R,X) \not\sqsubseteq C(W,X)$
  - $\& C(W,X) \not\sqsubseteq C(R,X) \rightarrow \text{race!}$
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

```
T1  T2

lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)

lock(b)
write x
unlock(b)
```

Shadow Memory

- $y \mapsto a, b$
- $x \mapsto a$
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!
Data Race Detection - Locksets

- Lack of synchronization arises with complex locking
- We can dynamically track the locks guarding an address!

```
T1
lock(a); lock(b)
read y
write x
unlock(b)
write x
unlock(a)

T2
lock(b)
write x
unlock(b)
```

Shadow Memory

```
y↦a, b
x↦{}
```

Note: Both x and y are always protected by locks. x still races.
Lockset based data race detection has many issues

- Synchronization may be fork/join, wait/notify based
- Initialization --> Process in Parallel --> Combine
- Richer parallel designs
Data Race Detection - Locksets

• Lockset based data race detection has many issues
  – Synchronization may be fork/join, wait/notify based
  – Initialization --> Process in Parallel --> Combine
  – Richer parallel designs

• Tends to have many false positives
Order Violations

- Some accesses are wrongly assumed to occur before others

\[ x = \text{new Data} \]

\[ x\rightarrow\text{datum} \]

wait/notify or condition variables can fix these
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

```plaintext
tmp = x
tmp = tmp+1
x = tmp
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

```
int x = ...;
int tmp = ...;

// No race, similar effect!
lock()
    tmp = x
    tmp = tmp + 1
unlock()
lock()
    x = tmp
unlock()
```

```
int x = ...;
int tmp = ...;

// Race!
lock()
    tmp = x
    tmp = tmp + 1
unlock()
lock()
    x = tmp
unlock()
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

What do we really want?

```
tmp = x
lock()
tmp = tmp + 1
unlock()
x = tmp
```

```
lock()
tmp = x
unlock()
tmp = tmp + 1
lock()
x = tmp
unlock()
```
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is *atomic* if it is equivalent to a sequentially executed one.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.
- An execution (or fragment thereof) is **atomic** if it is equivalent to a sequentially executed one.

![Diagram](image-url)
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is *atomic* if it is equivalent to a sequentially executed one.
Atomicity Violations

- Data races are a matter of perspective
  - Fine grained locking doesn’t solve much.

- An execution (or fragment thereof) is **atomic** if it is equivalent to a sequentially executed one.
  - This also takes care of data races
  - Similar to notions from databases (serializability & linearizability)
Atomicity Violations

- How can we find atomicity violations?
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ’75, POPL ‘04]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction \([\text{CACM \text{ '75}, \text{POPL \text{ '04}}]\)]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]

Only some patterns are unserializable. Detect unlikely issues via training.
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

• How can we find atomicity violations (or correctness)?
  – Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  – 2 thread atomicity patterns [Lu ASPLOS ‘06]
  – Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]
How can we find atomicity violations (or correctness)?
- Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
- 2 thread atomicity patterns [Lu ASPLOS ‘06]
- Conflict graphs [PLDI ‘08, RV ‘11]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]

\[
\begin{align*}
a &= x \\
e &= d \\
c &= x \\
d &= c \\
T1 &\rightarrow T2 \\
T2 &\rightarrow T3 \\
\text{Cycles are unserializable!}
\end{align*}
\]
Atomicity Violations

- How can we find atomicity violations (or correctness)?
  - Lipton’s Theory of Reduction [CACM ‘75, POPL ‘04]
  - 2 thread atomicity patterns [Lu ASPLOS ‘06]
  - Conflict graphs [PLDI ‘08, RV ‘11]

- How do we know what regions should be atomic?
Concurrent Test Generation

- What if we don’t already have a buggy execution?
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emption finds most bugs
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
- Careful schedule generation & selection
Concurrent Test Generation

- What if we don’t already have a buggy execution?
- Explore bounded schedules
  - 2 threads and few pre-emptions finds most bugs
- Careful schedule generation & selection
- **Generate API unit tests targeting concurrency**
  - Small enough for exhaustive schedule exploration
Other Directions

- Shepherding toward good behaviors
- Tolerating & avoiding on the fly
- Static analysis
Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard
Summary

- Parallelism is important for modern performance
- Choosing what to parallelize can be hard
- Parallelizing correctly can be very hard

And the hard problems are interesting to study.