Scalability + Correctness

Chris Rossbach + Calvin Lin

CS380p
Outline for Today

- Concurrency & Parallelism Basics
  - Decomposition redux
  - Measuring Parallel Performance
  - Performance Tradeoffs
  - Correctness and Performance

Acknowledgments: some materials in this lecture borrowed from or built on materials from:
- Emmett Witchel, who borrowed them from: Kathryn Mckinley, Ron Rockhold, Tom Anderson, John Carter, Mike Dahlin, Jim Kurose, Hank Levy, Harrick Vin, Thomas Narten, and Emery Berger
- Mark Silberstein, who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta
Review: Game of Life
Review: Game of Life

• Given a 2D Grid:

\[ v_t(i, j) = F(v_{t-1}(\text{of all its neighbors})) \]
Domain decomposition
Domain decomposition

Each CPU gets part of the input
Domain decomposition

Each CPU gets part of the input

- What would a functional decomposition look like?
- Issues/obstacles with this domain decomposition?
Functional decomposition

CPU 0:
\[ \text{tmp}_{i,j} = F(v_{t-1}(\text{neighbors})) \]

CPU 1:
\[ v_t(i,j) = \text{tmp}_{i,j} \]
Functional decomposition

Each CPU gets part of the per-cell work

CPU 0:
\[ \text{tmp}_{i,j} = F(\text{vt-1(neighbors)}) \]

CPU 1:
\[ \text{vt}(i,j) = \text{tmp}_{i,j} \]
Functional decomposition

Each CPU gets part of the per-cell work

CPU 0:
\[ \text{tmp}_{i,j} = F(v_{t-1}(\text{neighbors})) \]

CPU 1:
\[ v_t(i,j) = \text{tmp}_{i,j} \]
Domain decomposition
Domain decomposition

• Each CPU gets part of the input
Domain decomposition

• Each CPU gets part of the input
Domain decomposition

- Each CPU gets part of the input

Issues?
Domain decomposition

- Each CPU gets part of the input

Issues?
- Accessing Data
Domain decomposition

• Each CPU gets part of the input

Issues?
  • Accessing Data
    • Can we access $v(i+1, j)$ from CPU 0
Domain decomposition

• Each CPU gets part of the input

CPU 0

CPU 1

Issues?
• Accessing Data
  • Can we access v(i+1, j) from CPU 0
  • ...as in a “normal” serial program?
  • Shared memory? Distributed?
  • Time to access v(i+1,j) == Time to access v(i-1,j) ?
  • *Scalability vs Latency*
Domain decomposition

• Each CPU gets part of the input

Issues?
• Accessing Data
  • Can we access \(v(i+1, j)\) from CPU 0
  • ...as in a “normal” serial program?
  • Shared memory? Distributed?
• Time to access \(v(i+1,j)\) == Time to access \(v(i-1,j)\) ?
  • Scalability vs Latency
• Control
  • Can we assign one vertex per CPU?
  • Can we assign one vertex per process/logical task?
  • Task Management Overhead
Domain decomposition

• Each CPU gets part of the input

Issues?
• Accessing Data
  • Can we access \( v(i+1, j) \) from CPU 0
  • ...as in a “normal” serial program?
  • Shared memory? Distributed?
  • Time to access \( v(i+1,j) \) == Time to access \( v(i-1,j) \) ?
• Scalability vs Latency
• Control
  • Can we assign one vertex per CPU?
  • Can we assign one vertex per process/logical task?
  • Task Management Overhead
• Load Balance
Domain decomposition

• Each CPU gets part of the input

CPU 0

Issues?

• Accessing Data
  • Can we access $v(i+1, j)$ from CPU 0
  • ...as in a “normal” serial program?
  • Shared memory? Distributed?
  • Time to access $v(i+1,j) =\text{ Time to access } v(i-1,j)$?
  • *Scalability vs Latency*

• Control
  • Can we assign one vertex per CPU?
  • Can we assign one vertex per process/logical task?
  • *Task Management Overhead*

• *Load Balance*

• Correctness
  • order of reads and writes is non-deterministic
  • synchronization is required to enforce the order
  • *locks, semaphores, barriers, conditionals*...
Load Balancing
Load Balancing

• Slowest task determines performance
Load Balancing

• Slowest task determines performance
Load Balancing

- Slowest task determines performance
Granularity
Granularity

\[ G = \frac{\text{Computation}}{\text{Communication}} \]
Granularity

$G = \frac{\text{Computation}}{\text{Communication}}$

- **Fine-grain parallelism**
  - $G$ is small
  - Good load balancing
  - Potentially high overhead
  - Hard to get correct

- **Coarse-grain parallelism**
  - $G$ is large
  - Load balancing is tough
  - Low overhead
  - Easier to get correct
Performance: Amdahl’s law
Performance: Amdahl’s law

• Speedup is bound by serial component
• Split program serial time \( T_{\text{serial}} = 1 \) into
  • Ideally parallelizable portion: \( A \)
    • assuming perfect load balancing, identical speed, no overheads
  • Cannot be parallelized (serial) portion: \( 1 - A \)
  • Parallel time:
    \[
    T_{\text{parallel}} = \frac{A}{\#\text{CPUs}} + (1 - A)
    \]

\[
\text{Speedup}(\#\text{CPUs}) = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{1}{\frac{A}{\#\text{CPUs}} + (1 - A)}
\]
Performance: Amdahl’s law

- Speedup is bound by serial component.

\[
\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}}
\]

\[
\text{Speedup}(\#\text{CPUs}) = \frac{T_{\text{serial}}}{T_{\text{parallel}}} = \frac{1}{A \frac{\#\text{CPUs}}{1 - (1 - A)}}
\]
Amdahl’s law
Amdahl’s law

- **X seconds**
  - my task

- **X/2 seconds**
  - Serial
- **X/2 seconds**
  - Parallelizable
Amdahl’s law

What makes something “serial” vs. parallelizable?
Amdahl’s law

End to end time: $X$ seconds
Amdahl’s law

End to end time: X seconds

2 CPUs

X/2 seconds

Serial

Parallelizable

X/2 seconds

Scalability + Correctness
Amdahl’s law

X/2 seconds

Serial

End to end time: X seconds
Amdahl’s law

End to end time: X seconds

Serial
Parallelizable
Parallelizable

X/2 seconds
X/4 seconds

2 CPUs

Scalability + Correctness
Amdahl’s law

Serial

X/2 seconds

Parallelizable

X/4 seconds

Parallelizable

2 CPUs

Scalability + Correctness
Amdahl’s law

End to end time: \( \frac{X}{2} + \frac{X}{4} = \frac{3}{4}X \) seconds
Amdahl’s law

End to end time: \((X/2 + X/4) = (3/4)X\) seconds

What is the “speedup” in this case?
Amdahl’s law

End to end time: \((X/2 + X/4) = (3/4)X\) seconds

What is the “speedup” in this case?

\[
\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{\frac{5}{2 \text{ CPUs}} + (1 - .5)} = 1.333
\]
Speedup exercise

8 CPUs

$X/4$ seconds

$3 \times X/4$ seconds

Serial

Parallelizable

End to end time: $X$ seconds
Speedup exercise

Serial

X/4 seconds

8 CPUs

End to end time: X seconds
Speedup exercise

\[ \frac{X}{4} \text{ seconds} \]

Serial

8 CPUs

Scalability + Correctness
Speedup exercise

What is the “speedup” in this case?
Speedup exercise

X/4 seconds

| Serial | P | P | P | P | P | P | P | P |

What is the “speedup” in this case?
Speedup exercise

X/4 seconds

Serial

8 CPUs

What is the “speedup” in this case?
Speedup exercise

What is the “speedup” in this case?
Speedup exercise

What is the “speedup” in this case?

\[
\text{Speedup} = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{A \frac{1}{\#CPUs} + (1 - A)} = \frac{1}{\frac{75}{8} + (1 - 0.75)} = 2.91x
\]
Amdahl Action Zone

50% PARALLEL

SPEEDUP vs. NUMBER OF CPUS

CS380P
Scalability + Correctness
13
Amdahl Action Zone

Percentage of parallel work
- 50%
- 75%

SPEEDUP

NUMBER OF CPUS

CS380P
Scalability + Correctness
Amdahl Action Zone

Percentage of parallel work

- 50%
- 75%
- 90%
- 95%
- 99%

SPEEDUP

NUMBER OF CPUS

CS380P  Scalability + Correctness  15
Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson

• $N = \#CPUs$, $S = \text{serial portion} = 1 - A$

• Amdahl's law: $\text{Speedup}(N) = \frac{1}{\frac{A}{N} + S}$
  
  • **Strong scaling**: $\text{Speedup}(N)$ calculated with total work fixed
  • Solve same fixed size problem, #CPUs grows
  • Fixed parallel portion $\rightarrow$ speedup stops increasing

• Gustafson's law: $\text{Speedup}(N) = N + (N-1) \cdot S$
  
  • **Weak scaling**: $\text{Speedup}(N)$ calculated with work-per-CPU fixed
  • Add more CPUs $\rightarrow$ Add more work $\rightarrow$ granularity stays fixed
  • Problem size grows: solve larger problems
  • Consequence: speedup upper bound much greater
Super-linear speedup
Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: *best* serial algorithm
Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: *best* serial algorithm
- Example:
Super-linear speedup

• Possible due to cache
• But usually just poor methodology
• Baseline: *best* serial algorithm
• Example:

  • Efficient bubble sort takes:
    • Parallel 40s
    • Serial 150s
    • \( Speedup = \frac{150}{40} = 3.75 \) ?

  • NO!
    • Serial quicksort runs in 30s
    • \( Speedup = 0.75 \)
Concurrency and Correctness

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}

Thread 2
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}

Answer:
A. 0
B. 1
C. 2
D. More than 2
Schedules/Interleavings

Model of concurrent execution
• Interleave statements from each thread into a single thread
• If any interleaving yields incorrect results, synchronization is needed

Thread 1

```plaintext
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
```

Thread 2

```plaintext
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
```
Schedules/Interleavings

Model of concurrent execution

• Interleave statements from each thread into a single thread
• If any interleaving yields incorrect results, synchronization is needed

If X==0 initially, X == 1 at the end. WRONG result!
Locks fix this with Mutual Exclusion

Mutual exclusion ensures only safe interleavings
  • But it limits concurrency, and hence scalability/performance

```java
void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}
```
Locks fix this with Mutual Exclusion

```java
void increment() {
    lock.acquire();
    int temp = X;
    temp = temp + 1;
    X = temp;
    lock.release();
}
```

Mutual exclusion ensures only safe interleavings

• *But it limits concurrency, and hence scalability/performance*
Correctness conditions

while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
Correctness conditions

while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}

Scalability + Correctness
Correctness conditions

• Safety
  • Only one thread in the critical region

while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
Correctness conditions

- **Safety**
  - Only one thread in the critical region

- **Liveness**
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

```c
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```
Correctness conditions

- **Safety**
  - Only one thread in the critical region

- **Liveness**
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

- **Bounded waiting**
  - A thread that enters the entry section enters the critical section within some bounded number of operations.

```c
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```
Correctness conditions

- **Safety**
  - Only one thread in the critical region

- **Liveness**
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region

- **Bounded waiting**
  - A thread that enters the entry section enters the critical section within some bounded number of operations.
  - If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i's request is granted

```c
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```
Correctness conditions

• Safety
  • Only one thread in the critical region

• Liveness
  • Some thread that enters the entry section eventually enters the critical region
  • Even if other thread takes forever in non-critical region

• Bounded waiting
  • A thread that enters the entry section enters the critical section within some bounded number of operations.
  • If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i’s request is granted

while(1) {
  Entry section
  Critical section
  Exit section
  Non-critical section
}
Correctness conditions

• Safety
  • Only one thread in the critical region

• Liveness
  • Some thread that enters the entry section eventually enters the critical region
  • Even if other thread takes forever in non-critical region

• Bounded waiting
  • A thread that enters the entry section enters the critical section within some bounded number of operations.
  • If a thread is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i’s request is granted

while(1) {
    
    Mutex, spinlock, etc. are ways to implement these

    Critical section

    Non-critical section

}
Let’s talk concurrency control
Let’s talk concurrency control

Consider a hash-table
Let’s talk concurrency control

Consider a hash-table
Let’s talk concurrency control

Consider a hash-table

thread T1

``` java
ht.add();
if(ht.contains())
   ht.del();
```
Let’s talk concurrency control

Consider a hash-table

thread T1

ht.add( );

if(ht.contains( ))
    ht.del( );
Let’s talk concurrency control

Consider a hash-table

thread T1

ht.add( );

if(ht.contains( ))
    ht.del( );
Let’s talk concurrency control

Consider a hash-table

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ht.add(□);</code></td>
<td><code>ht.add(□);</code></td>
</tr>
<tr>
<td><code>if(ht.contains(□))</code></td>
<td><code>if(ht.contains(□))</code></td>
</tr>
<tr>
<td><code>ht.del(□);</code></td>
<td><code>ht.del(□);</code></td>
</tr>
</tbody>
</table>

Scalability + Correctness
Let’s talk concurrency control

Consider a hash-table

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.add( )</td>
<td>ht.add( )</td>
</tr>
<tr>
<td>if(ht.contains( ))</td>
<td>if(ht.contains( ))</td>
</tr>
<tr>
<td></td>
<td>ht.del( )</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Scalability + Correctness
Let’s talk concurrency control

Consider a hash-table

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.add( );</td>
<td>ht.add( );</td>
</tr>
<tr>
<td>if(ht.contains( ))</td>
<td>if(ht.contains( ))</td>
</tr>
<tr>
<td></td>
<td>ht.del( );</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Pessimistic concurrency control: coarse locks

Consider a hash-table

thread T1

ht.add( );

if(ht.contains( ))
    ht.del( );
Pessimistic concurrency control: coarse locks

Consider a hash-table

<table>
<thead>
<tr>
<th></th>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ht.add(□);</td>
<td>ht.add(□);</td>
</tr>
<tr>
<td></td>
<td>if(ht.contains(□))</td>
<td>if(ht.contains(□))</td>
</tr>
<tr>
<td></td>
<td>ht.del(□);</td>
<td>ht.del(□);</td>
</tr>
</tbody>
</table>
Pessimistic concurrency control: coarse locks

Consider a hash-table

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.add( );</td>
<td>ht.add( );</td>
</tr>
<tr>
<td>if(ht.contains( ));</td>
<td>if(ht.contains( ));</td>
</tr>
<tr>
<td>ht.del( );</td>
<td>ht.del( );</td>
</tr>
</tbody>
</table>

Scalability + Correctness
Pessimistic concurrency control: coarse locks

Consider a hash-table:

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.lock();</td>
<td>ht.add(□);</td>
</tr>
<tr>
<td>ht.add(□);</td>
<td></td>
</tr>
<tr>
<td>if(ht.contains(□))</td>
<td>if(ht.contains(□))</td>
</tr>
<tr>
<td>ht.del(□);</td>
<td>ht.del(□);</td>
</tr>
</tbody>
</table>

Scalability + Correctness
Pessimistic concurrency control: coarse locks

Consider a hash-table

```java
thread T1
ht.lock();
ht.add( );
if(ht.contains( ))
    ht.del( );
ht.unlock();
thread T2
ht.add( );
if(ht.contains( ))
    ht.del( );
```
Pessimistic concurrency control: coarse locks

Consider a hash-table

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ht.lock();</code></td>
<td><code>ht.lock();</code></td>
</tr>
<tr>
<td><code>ht.add( );</code></td>
<td><code>ht.add( );</code></td>
</tr>
<tr>
<td>if(ht.contains( ))</td>
<td>if(ht.contains( ))</td>
</tr>
<tr>
<td><code>ht.del( );</code></td>
<td><code>ht.del( );</code></td>
</tr>
<tr>
<td><code>ht.unlock();</code></td>
<td><code>ht.unlock();</code></td>
</tr>
</tbody>
</table>
Consider a hash-table

```
ht.lock();
ht.add();
if (ht.contains())
    ht.del();
ht.unlock();
```

```
ht.lock();
ht.add();
if (ht.contains())
    ht.del();
ht.unlock();
```

<table>
<thead>
<tr>
<th>Thread T1</th>
<th>Thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ht.lock();</code></td>
<td><code>ht.lock();</code></td>
</tr>
<tr>
<td><code>ht.add();</code></td>
<td><code>ht.add();</code></td>
</tr>
<tr>
<td><code>if (ht.contains())</code></td>
<td><code>if (ht.contains())</code></td>
</tr>
<tr>
<td><code>ht.del();</code></td>
<td><code>ht.del();</code></td>
</tr>
<tr>
<td><code>ht.unlock();</code></td>
<td><code>ht.unlock();</code></td>
</tr>
</tbody>
</table>
Consider a hash-table

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td>ht.lock();</td>
<td>ht.lock();</td>
</tr>
<tr>
<td>ht.add( );</td>
<td>ht.add( );</td>
</tr>
<tr>
<td>if(ht.contains( ))</td>
<td>if(ht.contains( ))</td>
</tr>
<tr>
<td>ht.del( );</td>
<td>ht.del( );</td>
</tr>
<tr>
<td>ht.unlock();</td>
<td>ht.unlock();</td>
</tr>
</tbody>
</table>
Pessimistic concurrency control: coarse locks

Consider a hash-table:

```
ht.lock();
ht.add( );
if(ht.contains( ))
    ht.del( );
ht.unlock();
```

```
ht.lock();
ht.add( );
if(ht.contains( ))
    ht.del( );
ht.unlock();
```

Coarse lock:
- Non-conflicting ops serialized
- Low Complexity -- Low Performance
- Scalability + Correctness
Pessimistic concurrency control: fine locks

Consider a hash-table

```
thread T1
figure-out-locks();
lock-them-inorder();
ht.add( );
if(ht.contains( ))
    ht.del( );
unlock-locks();
```

```
thread T2
figure-out-locks();
lock-them-inorder();
ht.add( );
if(ht.contains( ))
    ht.del( );
unlock-locks();
```
Consider a hash-table

<table>
<thead>
<tr>
<th>thread T1</th>
<th>thread T2</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>figure-out-locks();</strong></td>
<td><strong>figure-out-locks();</strong></td>
</tr>
<tr>
<td><strong>lock-them-inorder();</strong></td>
<td><strong>lock-them-inorder();</strong></td>
</tr>
<tr>
<td>ht.add( );</td>
<td>ht.add( );</td>
</tr>
<tr>
<td>if(ht.contains( ))</td>
<td>if(ht.contains( ))</td>
</tr>
<tr>
<td>ht.del( );</td>
<td>ht.del( );</td>
</tr>
<tr>
<td>unlock-locks();</td>
<td>unlock-locks();</td>
</tr>
</tbody>
</table>

Fine-grain lock:
- Non-conflicting parallel
- High Complexity -- High Performance
Why Locks are Hard

• Coarse-grain locks
  • Simple to develop
  • Easy to avoid deadlock
  • Few data races
  • Limited concurrency

• Fine-grain locks
  • Greater concurrency
  • Greater code complexity
  • Potential deadlocks
    • Not composable
  • Potential data races
    • Which lock to lock?
Why Locks are Hard

• Coarse-grain locks
  • Simple to develop
  • Easy to avoid deadlock
  • Few data races
  • Limited concurrency

• Fine-grain locks
  • Greater concurrency
  • Greater code complexity
  • Potential deadlocks
    • Not composable
  • Potential data races
    • Which lock to lock?

// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
Why Locks are Hard

• Coarse-grain locks
  • Simple to develop
  • Easy to avoid deadlock
  • Few data races
  • Limited concurrency

• Fine-grain locks
  • Greater concurrency
  • Greater code complexity
  • Potential deadlocks
    • Not composable
  • Potential data races
    • Which lock to lock?

// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}

Thread 0
move(a, b, key1);

Thread 1
move(b, a, key2);

CS380P
Scalability + Correctness 29
Why Locks are Hard

- Coarse-grain locks
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

- Fine-grain locks
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}

Thread 0
move(a, b, key1);

Thread 1
move(b, a, key2);

DEADLOCK!