Barrier Synchronization

CS 475, Spring 2019
Concurrent & Distributed Systems

With material from Herlihy & Shavit, Art of Multiprocessor Programming
Reducing Latency without lots of $$$

• Approach: use *concurrency*
• Limited by serial section
Matrix Multiplication

void multiply() {
    Worker[][] worker = new Worker[n][n];
    for (int row …)
        for (int col …)
            worker[row][col] = new Worker(row,col);
    for (int row …)
        for (int col …)
            worker[row][col].start();
    for (int row …)
        for (int col …)
            worker[row][col].join();
}
Matrix Addition

\[
\begin{pmatrix}
C_{00} & C_{00} \\
C_{10} & C_{10}
\end{pmatrix}
= \begin{pmatrix}
A_{00} + B_{00} & B_{01} + A_{01} \\
A_{10} + B_{10} & A_{11} + B_{11}
\end{pmatrix}
\]

4 parallel additions
Matrix Addition Task

class AddTask implements Runnable {
    Matrix a, b; // multiply this!
    public void run() {
        if (a.dim == 1) {
            c[0][0] = a[0][0] + b[0][0]; // base case
        } else {
            (partition a, b into half-size matrices a<sub>ij</sub> and b<sub>ij</sub>)
            Future<?> f<sub>00</sub> = exec.submit(add(a<sub>00</sub>,b<sub>00</sub>));
            ...
            Future<?> f<sub>11</sub> = exec.submit(add(a<sub>11</sub>,b<sub>11</sub>));
            f<sub>00</sub>.get(); ...; f<sub>11</sub>.get();
            ...
        }
    }
}
class FibTask implements Callable<Integer> {
    static ExecutorService exec = Executors.newCachedThreadPool();
    int arg;
    public FibTask(int n) {
        arg = n;
    }
    public Integer call() {
        if (arg > 2) {
            Future<Integer> left = exec.submit(new FibTask(arg-1));
            Future<Integer> right = exec.submit(new FibTask(arg-2));
            return left.get() + right.get();
        } else {
            return 1;
        }
    }
}
Arrows Reflect Dependencies

Note inefficiency in this implementation: $\text{fib}(2)$’s result should be computed only once
Fib Critical Path

Critical path length is 8
Addition

• Work is

\[
A_1(n) = 4 A_1(n/2) + \Theta(1) \\
= \Theta(n^2)
\]

Same as double-loop summation
Addition

• Critical Path length is

\[ A_\infty(n) = A_\infty(n/2) + \Theta(1) \]

spawned additions in parallel
Partition, synch, etc
Addition

- Critical Path length is

\[ A_{\infty}(n) = A_{\infty}(n/2) + \Theta(1) = \Theta(\log n) \]
Chaining CompletableFuture

```java
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```
Chaining CompletableFuture

```java
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```
Chaining CompletableFuture

```java
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});

// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```
Chaining CompletableFuture

```java
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```
Chaining CompletableFuture

```java
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Create ANOTHER future that is \textit{chained} to the first
Chaining CompletableFuture

```java
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    return "Jon";
});
// Chain on some more code to run when the future is done
CompletableFuture<String> greetingFuture = whatsYourNameFuture.thenApply(returnValue -> {
    return "Hello, " + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```

Block the main thread for both futures to finish
Today

• Barrier synchronization - the evil (?) twin of mutual exclusion
• Last synchronization topic!
  • Focus very heavily on implementation: how much memory is required, and how much contention is there on those resources?
Simple Video Game

- Prepare frame for display
  - By graphics coprocessor
- “soft real-time” application
  - Need at least 35 frames/second
  - OK to mess up rarely
Simple Video Game

while (true) {
    frame.prepare();
    frame.display();
}
Simple Video Game

• What about overlapping work?
  – 1\textsuperscript{st} thread displays frame
  – 2\textsuperscript{nd} prepares next frame

```java
while (true) {
  frame.prepare();
  frame.display();
}
```
Two-Phase Rendering

```java
while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}
```

```java
while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}
```
Two-Phase Rendering

while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}

while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}

Even phases
Two-Phase Rendering

while (true) {
    if (phase) {
        frame[0].display();
    } else {
        frame[1].display();
    }
    phase = !phase;
}

while (true) {
    if (phase) {
        frame[1].prepare();
    } else {
        frame[0].prepare();
    }
    phase = !phase;
}

odd phases
Synchronization Problems

• How do threads stay in phase?
• Too early?
  – “we render no frame before its time”
• Too late?
  – Recycle memory before frame is displayed
Ideal Parallel Computation

Ideal: All processes/threads move between phases at the same rate - no falling behind/getting ahead
Ideal Parallel Computation

Ideal: All processes/threads move between phases at the same rate - no falling behind/getting ahead
Real-Life Parallel Computation
Actual: Some work is harder than other work, some threads might get less CPU time… we get out of sync
Barrier Synchronization
Barrier Synchronization
Barrier Synchronization

Until every thread has left here

No thread enters here
Why Do We Care?

• Mostly of interest to
  – Scientific & numeric computation
  – Distributed protocols

• Elsewhere
  – Garbage collection
  – Less common in systems programming
  – Still important topic
Duality

• Dual to mutual exclusion
  – Include others, not exclude them
• Same implementation issues
  – Interaction with caches …
    • Invalidation?
    • Local spinning?
Example: Parallel Prefix

Before:

\[
\begin{array}{cccc}
    a & b & c & d \\
\end{array}
\]

After:

\[
\begin{array}{cccc}
    a & a+b & a+b+c & a+b+c+d \\
\end{array}
\]
Parallel Prefix

One thread
Per entry
Parallel Prefix: Phase 1

\[
\begin{array}{cccc}
a & b & c & d \\
\end{array}
\]

\[
\begin{array}{cccc}
a & a+b & b+c & c+d \\
\end{array}
\]
Parallel Prefix: Phase 2

\[ \begin{array}{cccc}
  a & b & c & d \\
  a+b & a+b+c & a+b+c+d \\
\end{array} \]
Parallel Prefix

• N threads can compute
  – Parallel prefix
  – Of N entries
  – In $\log_2 N$ rounds

• What if system is asynchronous?
  – Why we need barriers
class Prefix extends Thread {
    private int[] a;
    private int i;
    private Barrier b;
    public Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}
class Prefix extends Thread {
    private int[] a;
    private int i;
    private Barrier b;
    public Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}

Array of input values
Prefix

class Prefix extends Thread {
    private int[] a;
    private int i;
    private Barrier b;
    public Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}

Thread index
class Prefix extends Thread {
    private int[] a;
    private int i;
    private Barrier b;
    public Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}

Shared barrier
class Prefix extends Thread {
    private int[] a;
    private int i;
    private Barrier b;
    public Prefix(int[] a, Barrier b, int i) {
        a = a;
        b = b;
        i = i;
    }
}

Initialize fields
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        if (i >= d)
            a[i] += sum;
        d = d * 2;
    }
}
Where Do the Barriers Go?

```java
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        d = d * 2;
    }
}
```
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        d = d * 2;
    }
}
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        b.await();
        d = d * 2;
    }
}
public void run() {
    int d = 1, sum = 0;
    while (d < N) {
        if (i >= d)
            sum = a[i-d];
        b.await();
        if (i >= d)
            a[i] += sum;
        b.await();
        d = d * 2;
    }
}
Sidebar: HW2 Part 4 with Barriers
Barrier Implementations

• Cache coherence
  – Spin on locally-cached locations?
  – Spin on statically-defined locations?

• Latency
  – How many steps?

• Symmetry
  – Do all threads do the same thing?
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Number threads not yet arrived
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
Barriers

public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

If I’m last, reset fields for next time
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

Otherwise, wait for everyone else
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}

What’s wrong with this protocol?
Barrier b = new Barrier(n);
while ( mumble() ) {
    work();
    b.await();
}

Repeat

Do work

Synchronize
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n){
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
public class Barrier {
    AtomicInteger count;
    int size;
    public Barrier(int n) {
        count = AtomicInteger(n);
        size = n;
    }
    public void await() {
        if (count.getAndDecrement() == 1) {
            count.set(size);
        } else {
            while (count.get() != 0);
        }
    }
}
Basic Problem

- One thread “wraps around” to start phase 2
- While another thread is still waiting for phase 1
- One solution:
  - Always use two barriers
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    threadSense.set(!mySense)}}}
public class Barrier {
    AtomicInteger count;
    int size;

    boolean sense = false;

    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    threadSense.set(!mySense)}}

Completed odd or even-numbered phase?
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    }
    threadSense.set(!mySense)]]>
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {} 
        }
        threadSense.set(!mySense)}}}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)}}}
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>…

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement() == 1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
        threadSense.set(!mySense)\}
Sense-Reversing Barriers

```java
public class Barrier {
    AtomicInteger count;
    int size;
    boolean sense = false;
    ThreadLocal<boolean> threadSense = new ThreadLocal<boolean>...

    public void await {
        boolean mySense = threadSense.get();
        if (count.getAndDecrement()==1) {
            count.set(size); sense = !mySense
        } else {
            while (sense != mySense) {}
        }
    }

    threadSense.set(!mySense)}
```
Sense Reversing Barriers

• Good parts:
  • Spinning happens on *sense* which doesn’t get modified until all threads are done (no contention from spinning on the count value, which gets modified by each thread)

• Bad parts:
  • Hard to scale to hugely parallel situations — still have all $n$ threads contending for that one *sense* field
Combining Tree Barriers

2-barrier

2-barrier

2-barrier

2-barrier

2-barrier
Combining Tree Barriers

2-barrier

2-barrier

2-barrier
Combining Tree Barrier

```java
class Node {
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()}
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
```
public class Node{
    AtomicInteger count; int size;
    Node parent;
    Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {} 
        }...}}}

Parent barrier in tree
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}
        }
    }
}

Combining Tree Barrier

Am I last?
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {} 
        }
    }
}
Combining Tree Barrier

```java
public class Node{
    AtomicInteger count; int size;
    Node parent; Volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await()
            } else {
                count.set(size);
                sense = mySense
            } else {
                while (sense != mySense) {}
            }
        }
    }
}
```
public class Node{
    AtomicInteger count; int size;
    Node parent; volatile boolean sense;

    public void await() {...
        if (count.getAndDecrement()==1) {
            if (parent != null) {
                parent.await();
            }
            count.set(size);
            sense = mySense
        } else {
            while (sense != mySense) {}{}
        }...}}
public class Node{
  AtomicInteger count; int size;
  Node parent; Volatile boolean sense;

  public void await() {
    if (count.getAndDecrement()==1) {
      if (parent != null) {
        parent.await()
        count.set(size);
        sense = mySense
      } else {
        while (sense != mySense) {}
      }
    }...
  }
}
Combining Tree Barrier

- No sequential bottleneck
  - Parallel getAndDecrement() calls
- Low memory contention
  - Same reason
- Cache behavior
  - Local spinning on bus-based architecture
  - Not so good for non-uniform memory access architectures (NUMA)
    - common for large multiprocessor systems
Remarks

- Everyone spins on sense field
  - Local spinning on bus-based (good)
  - Network hot-spot on distributed architecture (bad)
- Not really scalable
  - Nodes going up and down the tree are not predictable - which (left or right) goes up? Whichever goes first -> possible contention and cache misses
Tournament Tree Barrier

• If tree nodes have fan-in 2
  - Don’t need to call getAndDecrement()
  - Winner chosen statically

• At level i
  - If i-th bit of id is 0, move up
  - Otherwise keep back
Tournament Tree Barriers

[Diagram of a tournament tree with nodes labeled as winners and losers, and arrows indicating the flow from loser to winner.]
Tournament Tree Barriers

All flags blue
Tournament Tree Barriers

Loser thread sets winner's flag
Tournament Tree Barriers

Loser spins on own flag
Tournament Tree Barriers

Winner spins on own flag
Tournament Tree Barriers

Winner sees own flag, moves up, spins
Tournament Tree Barriers

Bingo!
Tournament Tree Barriers

Sense-reversing: next time use blue flags
Remarks

- No need for read-modify-write calls
- Each thread spins on fixed location
  - Good for bus-based architectures
  - Good for NUMA architectures
So...How will we make use of multicores?

Back to Amdahl’s Law:

\[
\text{Speedup} = \frac{1}{(\text{ParallelPart}/N + \text{SequentialPart})}
\]

Pay for N = 8 cores

\[
\text{SequentialPart} = 25\%
\]

\[
\text{Speedup} = \text{only 2.9 times!}
\]

Must parallelize applications on a very fine grain!
Need Fine-Grained Locking

The reason we get only 2.9 speedup

Coarse Grained

Fine Grained

25% Shared

75% Unshared

25% Shared

75% Unshared
Traditional Scaling Process

Speedup

1.8x 3.6x 7x

User code

Traditional
Uniprocessor

Time: Moore’s law
Multicore Scaling Process

As noted, not so simple…
Real-World Scaling Process

Speedup

1.8x

2x

2.9x

User code

Parallelization and Synchronization will require great care…

Multicore
Nothing is free - if we want to take advantage of multicore machines, we need to work hard at designing our algorithms and systems to be amenable to parallelization.

We are at the dawn of a new era…

Still a lot of research and development to be done.

Moving on now: what happens when each CPU is on a different physical machine, connected by a network?
This work is licensed under a Creative Commons Attribution-ShareAlike license

- This work is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/
- You are free to:
  - Share — copy and redistribute the material in any medium or format
  - Adapt — remix, transform, and build upon the material
  - for any purpose, even commercially.
- Under the following terms:
  - Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - ShareAlike — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.