Barrier Synchronization

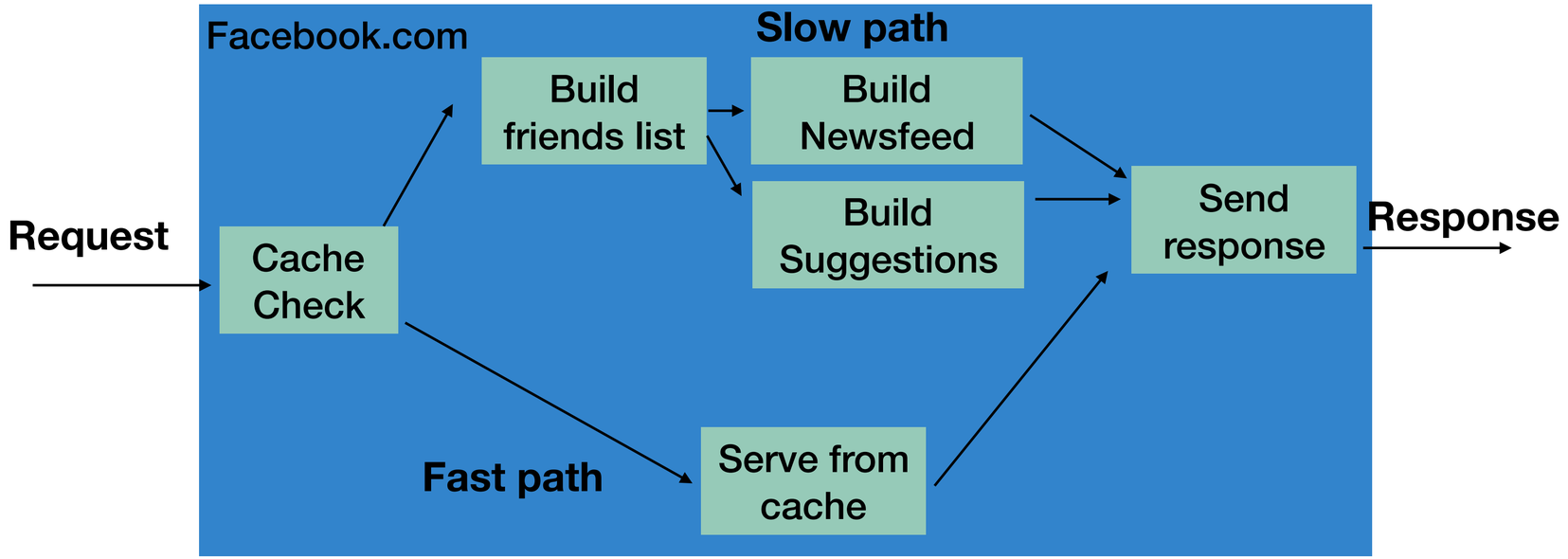
CS 475, Spring 2019
Concurrent & Distributed Systems

GEORGE MASON UNIVERSITY

With material from Herlihy & Shavit, Art of Multiprocessor Programming

Reducing Latency without lots of $$$

- Approach: use **concurrency**
- Limited by serial section



Matrix Multiplication

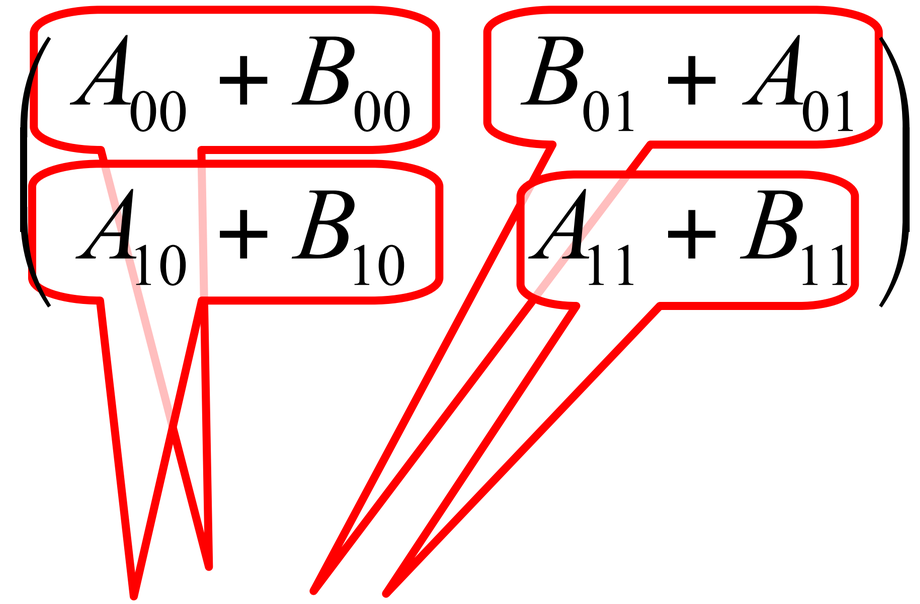
```
void multiply() {
  Worker[][] worker = new Worker[n][n];
  for (int row …)
    for (int col …)
      worker[row][col] = new Worker(row,col);
  for (int row …)
    for (int col …)
      worker[row][col].start();
  for (int row …)
    for (int col …)
      worker[row][col].join();
}
```

Start them

Wait for them to finish

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_ij and b_ij)
     Future<?> f00 = exec.submit(add(a00,b00));
     …
     Future<?> f11 = exec.submit(add(a11,b11));
      f00.get(); …; f11.get();
      …
     }}
```

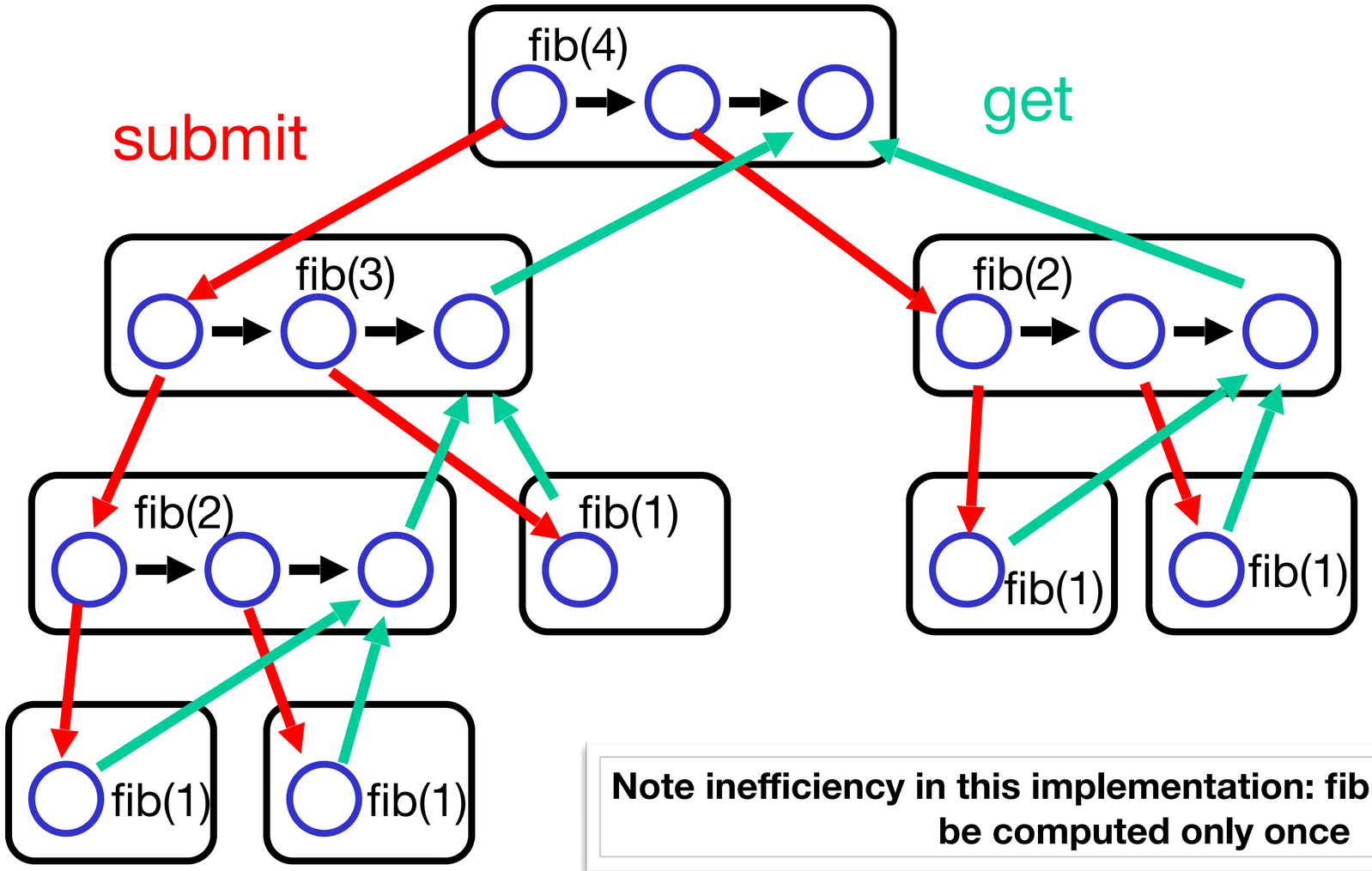
Submit 4 tasks

Multithreaded Fibonacci

```
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;
    }}}
```

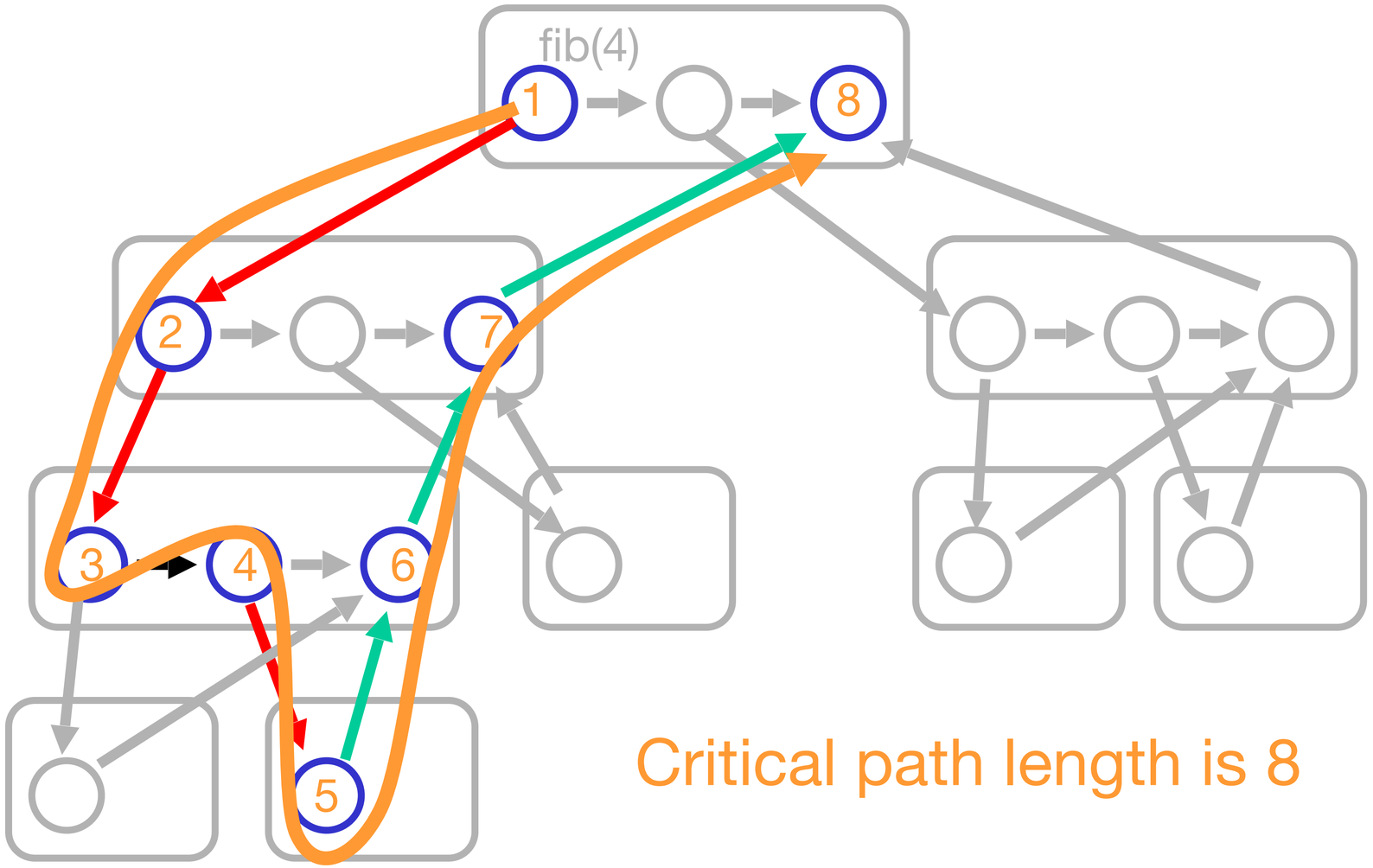
Pick up & combine results

Arrows Reflect Dependencies



Note inefficiency in this implementation: fib(2)'s result should be computed only once

Fib Critical Path



Critical path length is 8

Addition

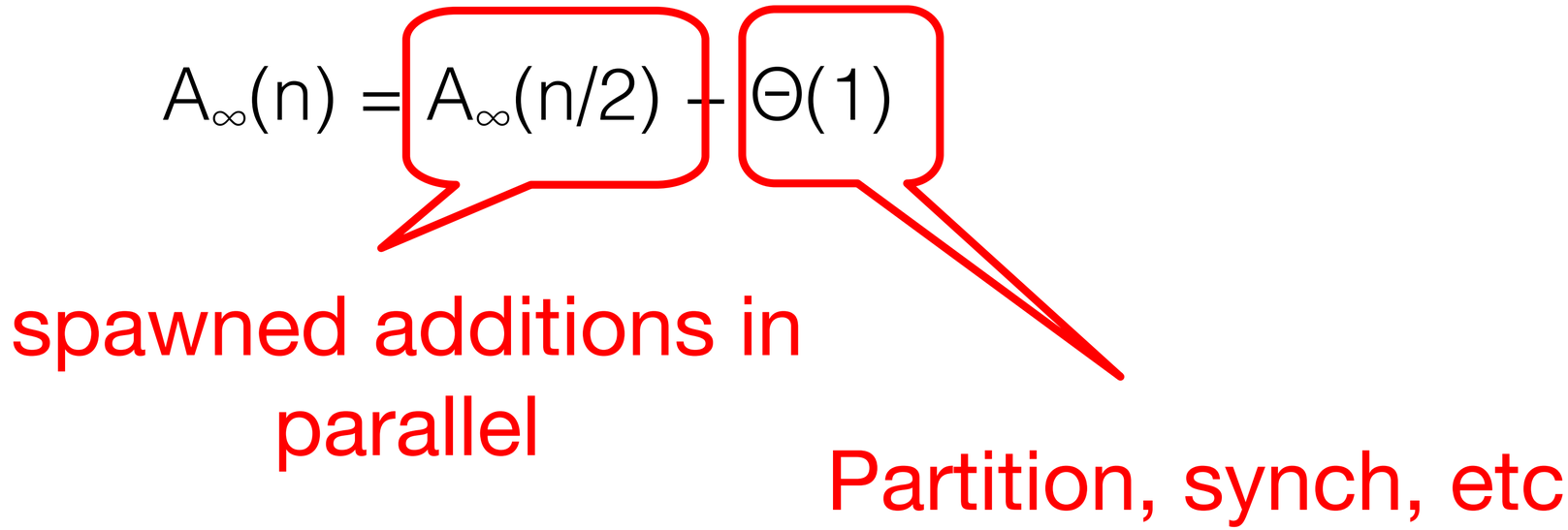
- Work is

$$A_1(n) = 4\,A_1(n/2) + \Theta(1)$$
$$= \Theta(n^2)$$

Same as double-loop summation

Addition

- Critical Path length is



Addition

- Critical Path length is

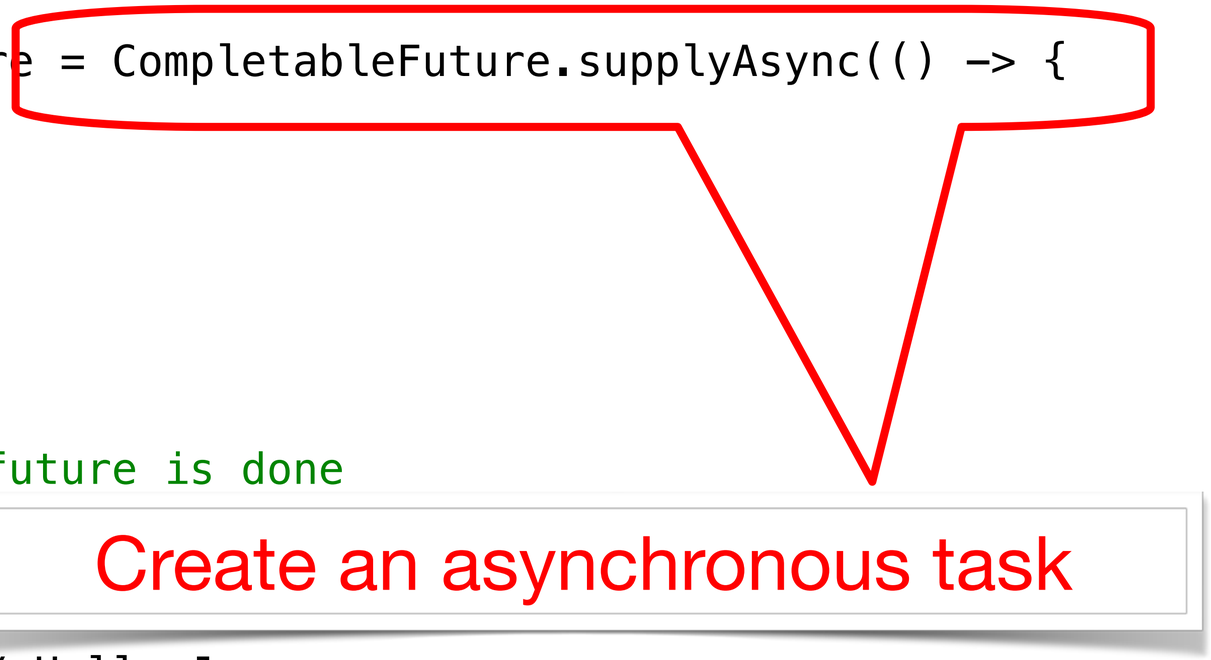
$$A_\infty(n) = A_\infty(n/2) + \Theta(1)$$
$$= \Theta(\log n)$$

Chaining CompletableFuture

```
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

```
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 =
    return “Hello, ” + returnValue;
});
System.out.println(greetingFuture.get()); // Hello Jon
```



Chaining CompletableFuture

Task will return string “Jon” eventually

```
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

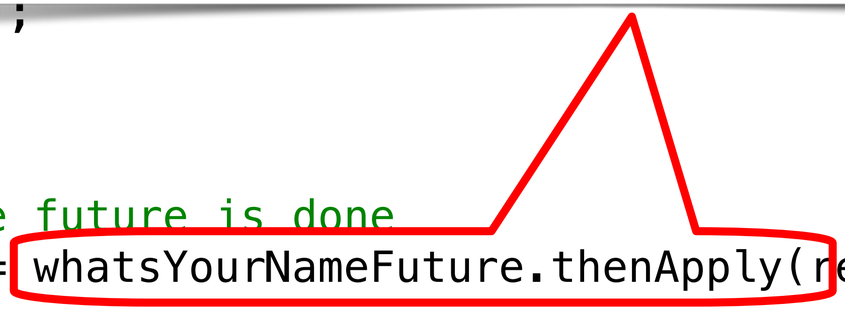
Task will return string “Jon” eventually

```
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

```
CompletableFuture<String> whatsYourName
    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 *chained* to the first



Chaining CompletableFuture

```
CompletableFuture<String> whatsYourNameFuture = CompletableFuture.supplyAsync(() -> {
    try {
        TimeUnit.SECONDS.sleep(1);
    } catch (InterruptedException e) {
        throw new IllegalStateException(e);
    }
    re
});
// 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?
  - 1st thread displays frame
  - 2nd prepares next frame

```
while (true) {
  frame.prepare();
  frame.display();
  }
```

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;
}
```

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;
}
```

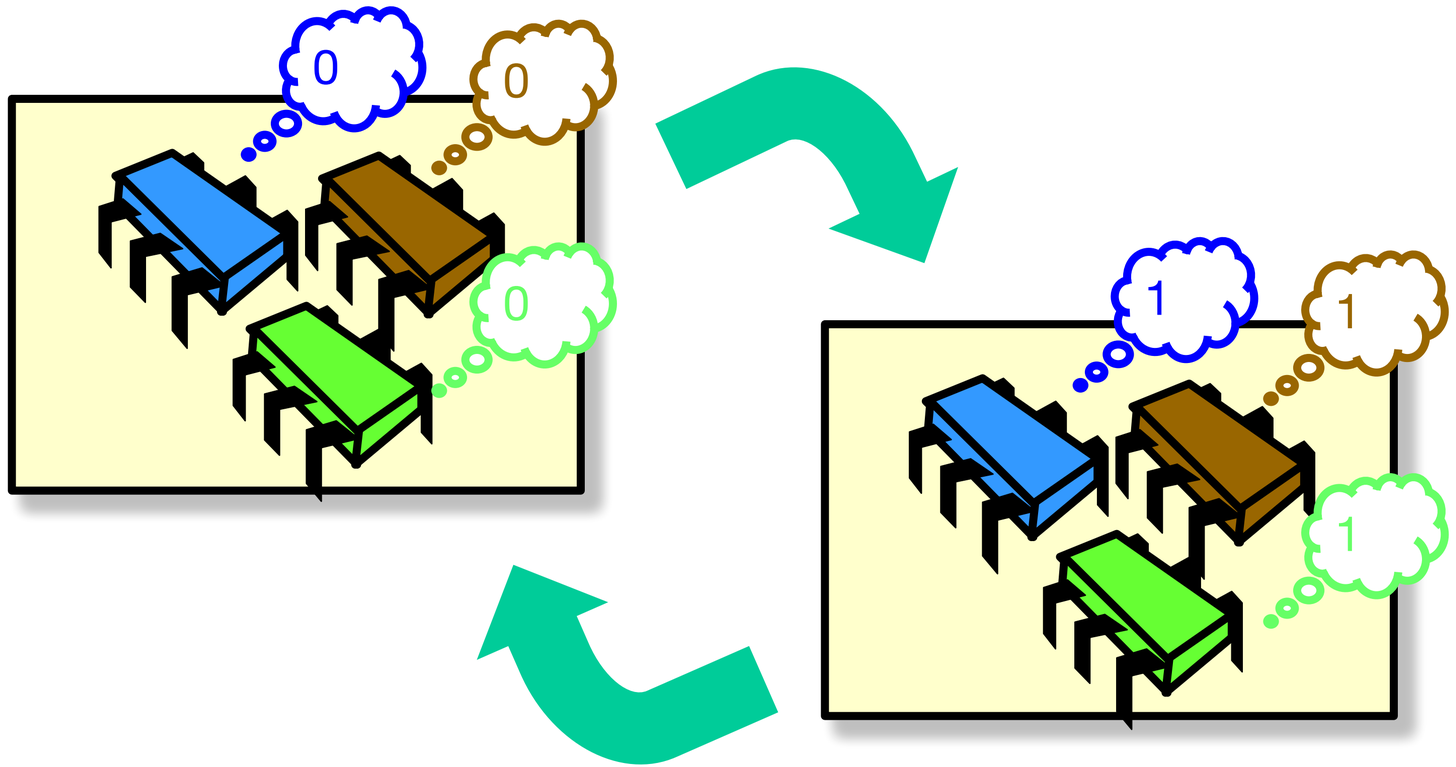
```
while (true) {
  if (phase) {
    frame[1].prepare();
  } else {
    frame[0].prepare();
  }
  phase = !phase;
}
```

odd phases

Synchronization Problems

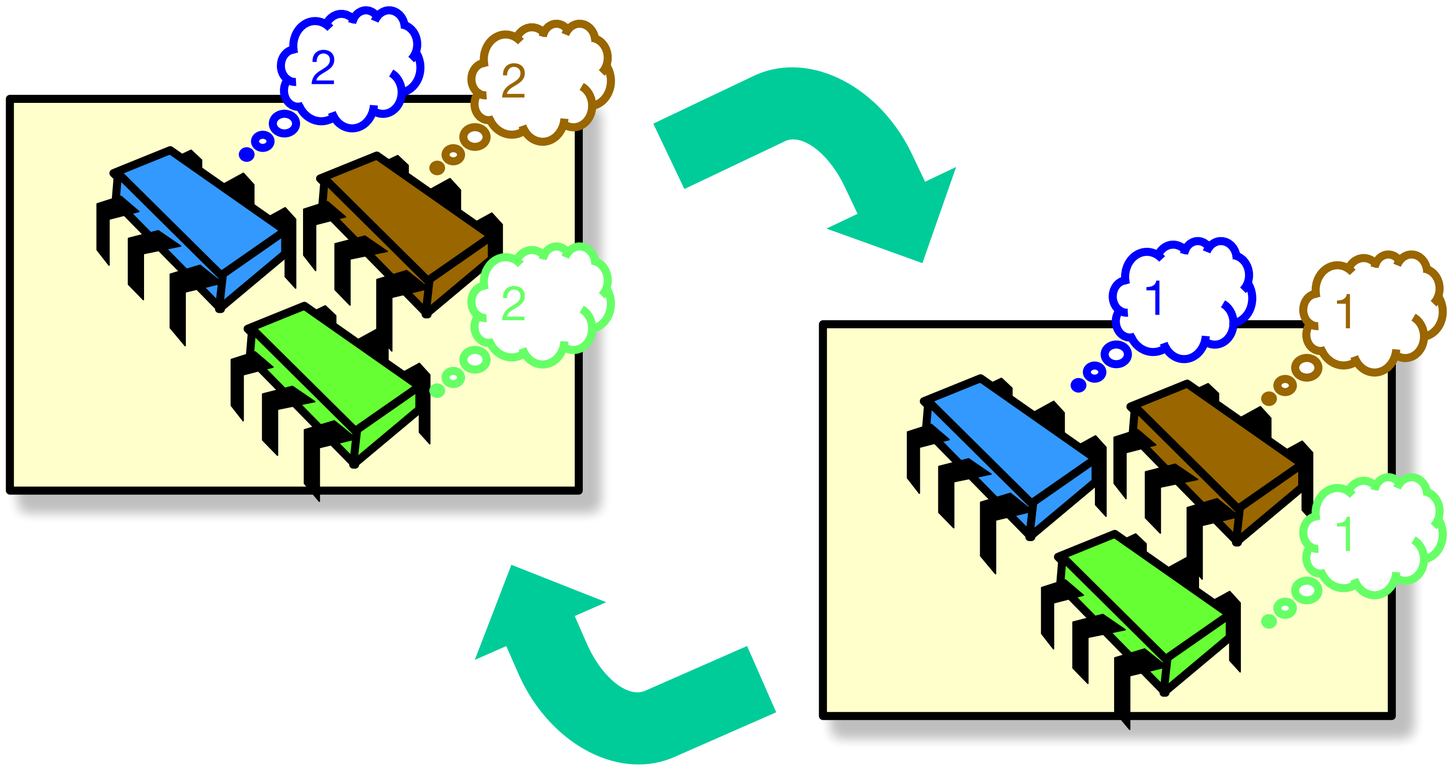
- How do threads stay in phase?
- Too early?
  - “we render no frame before its time”
- Too late?
  - Recycle memory before frame is displayed

Ideal Parallel Computation



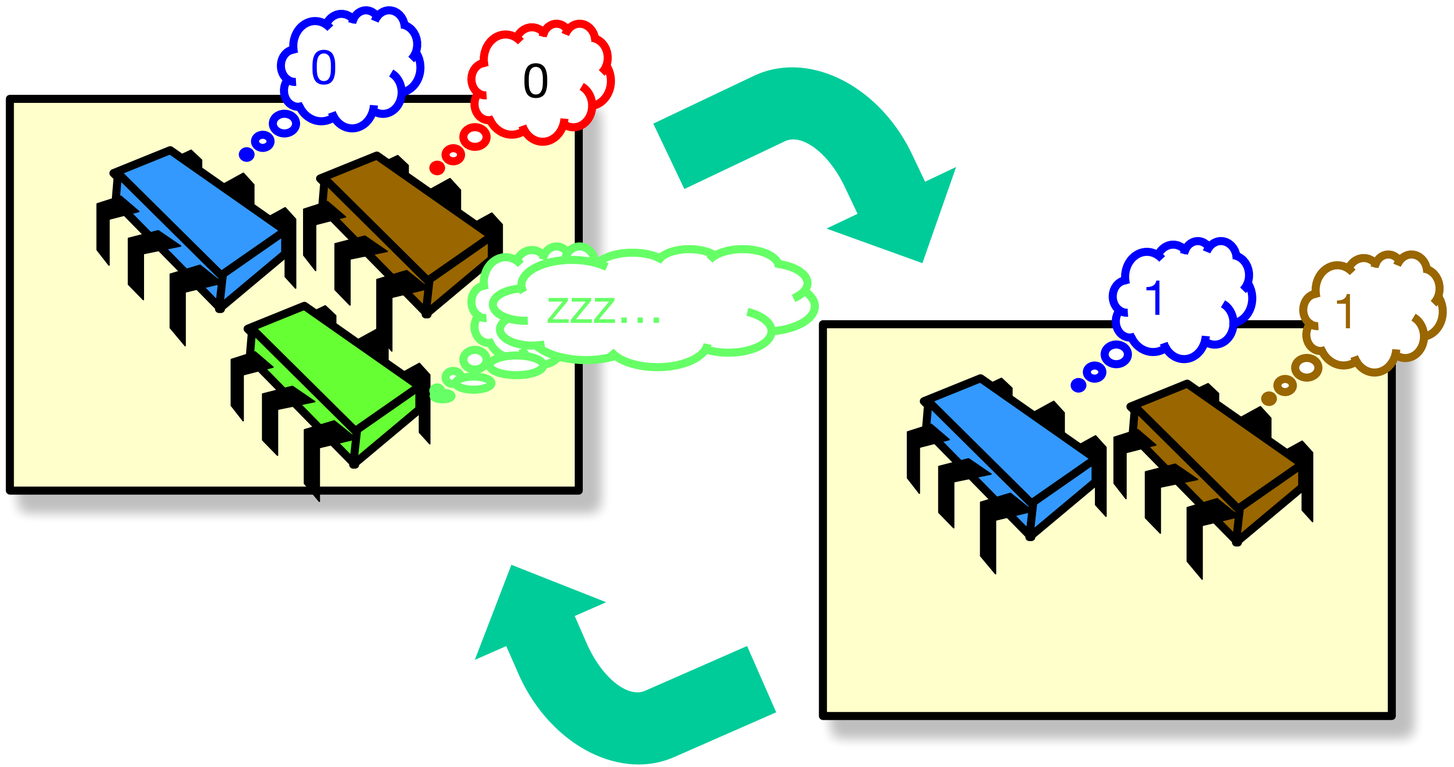
Ideal: All processes/threads move between phases at the same rate - no falling behind/getting ahead

Ideal Parallel Computation

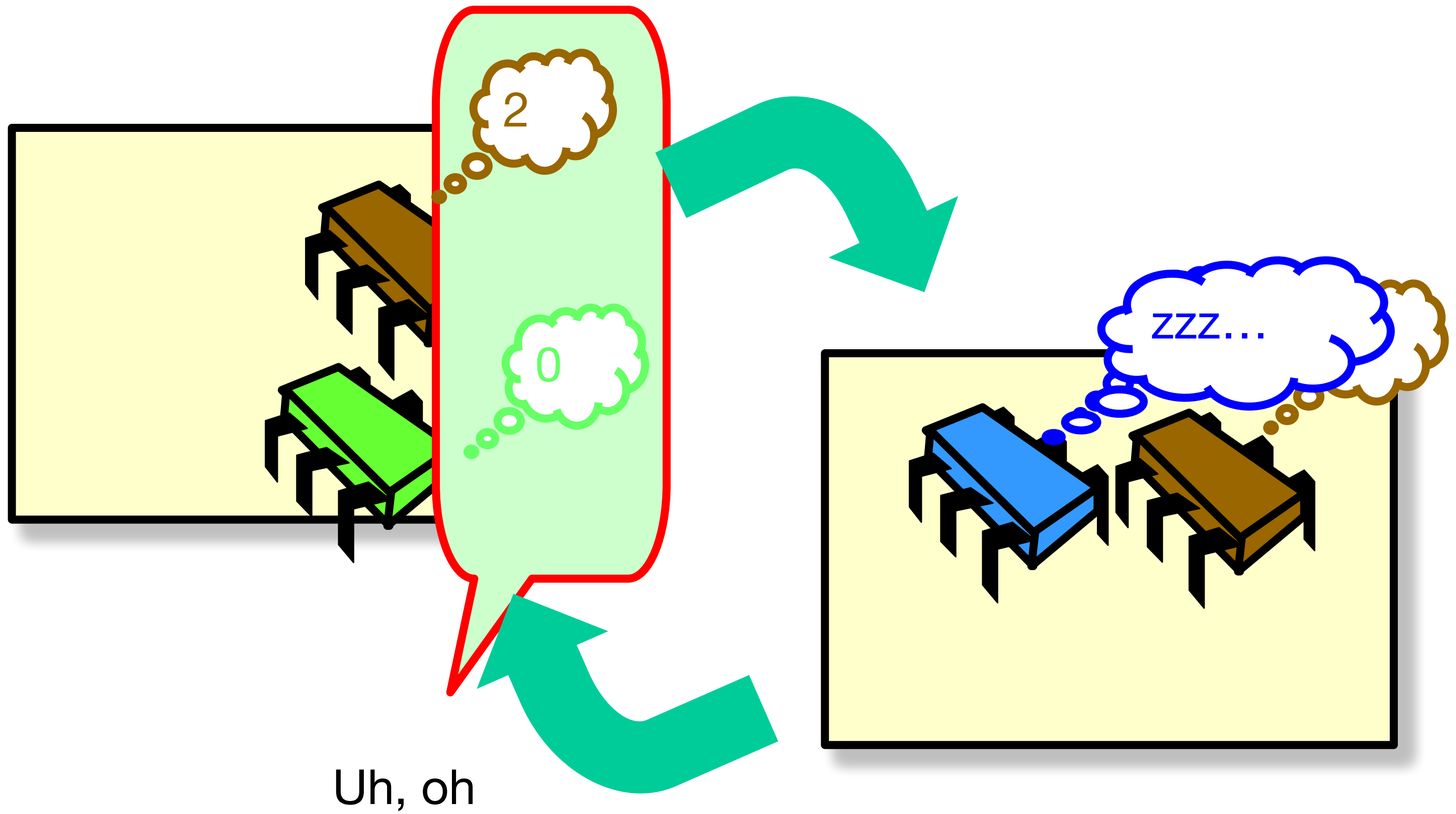


Ideal: All processes/threads move between phases at the same rate - no falling behind/getting ahead

Real-Life Parallel Computation

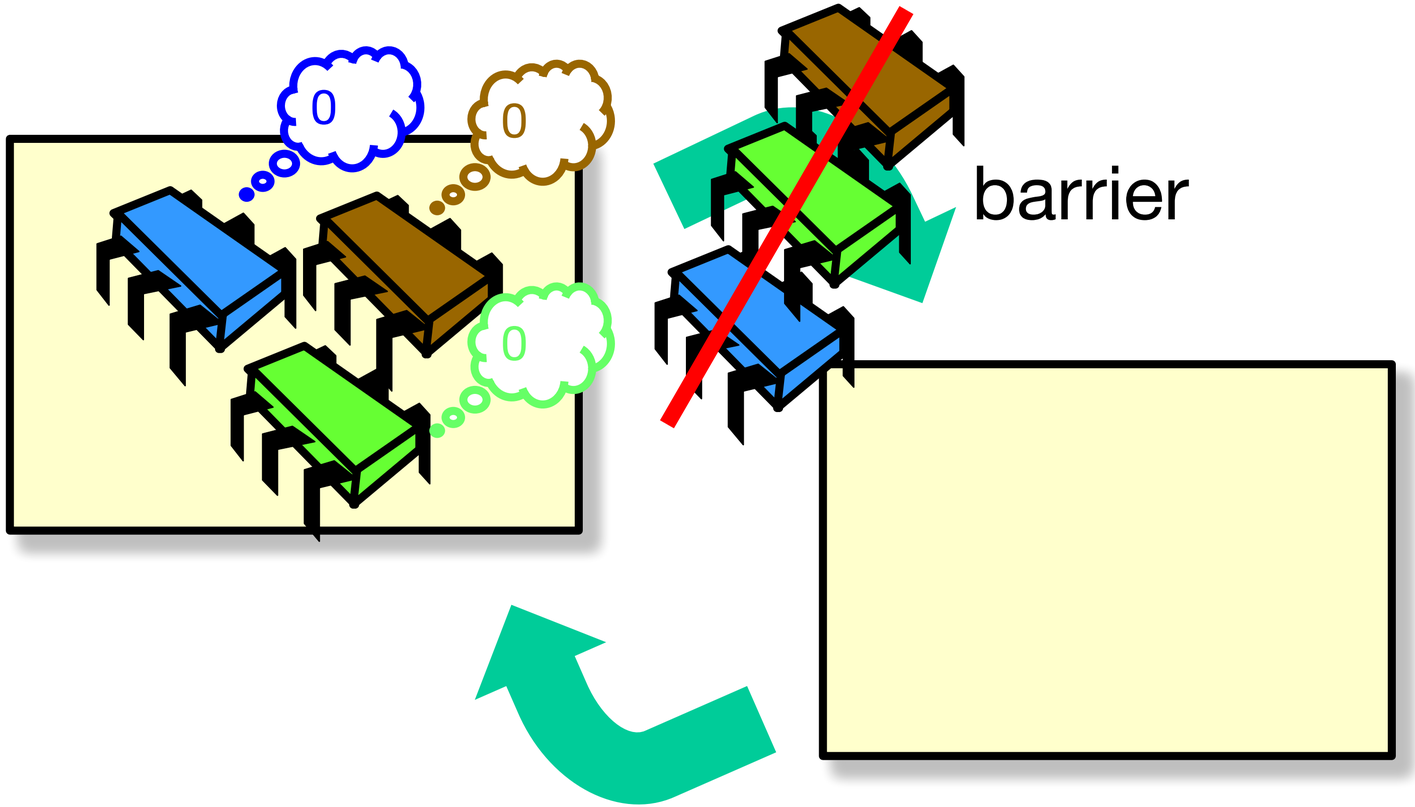


Real-Life Parallel Computation

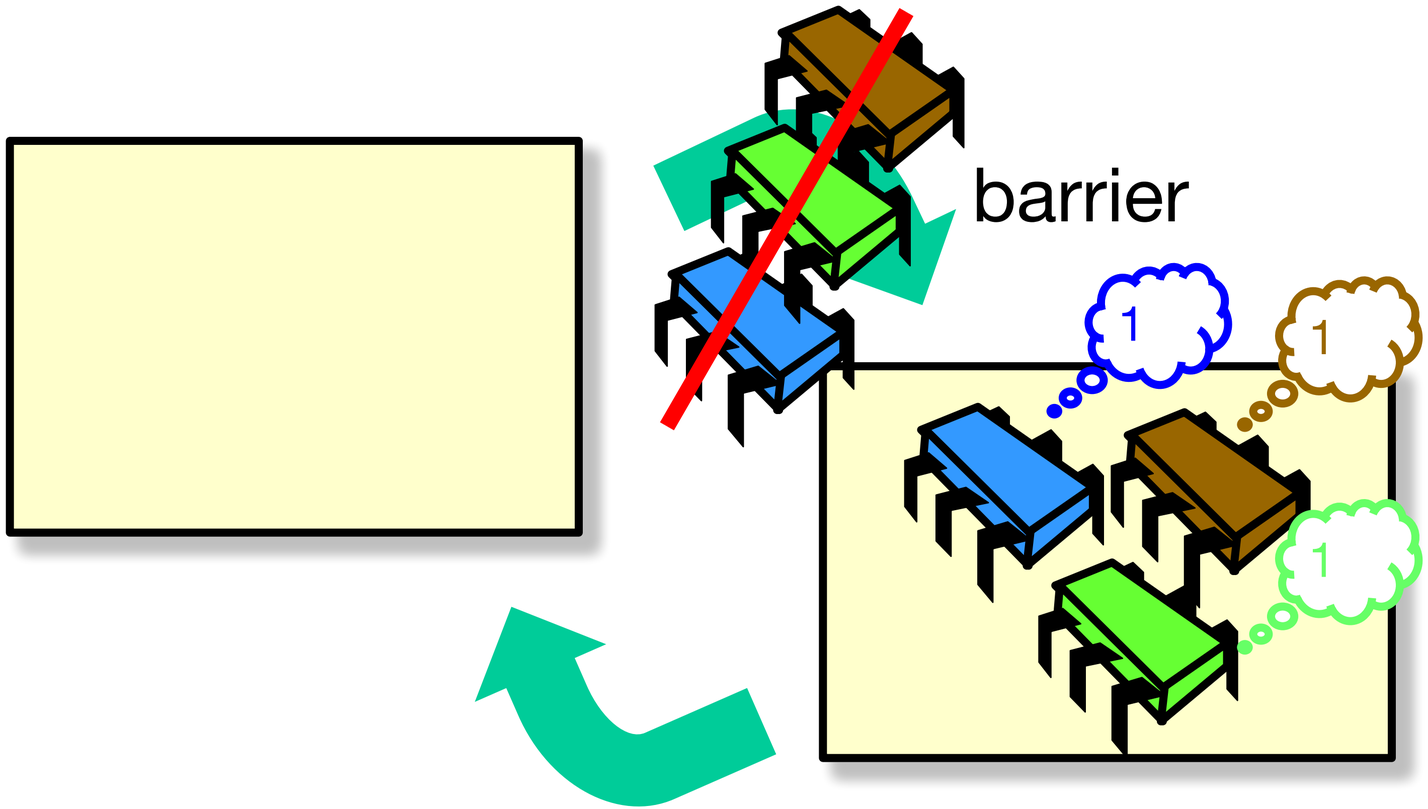


Actual: Some work is harder than other work, some threads might get less CPU time… we get out of sync

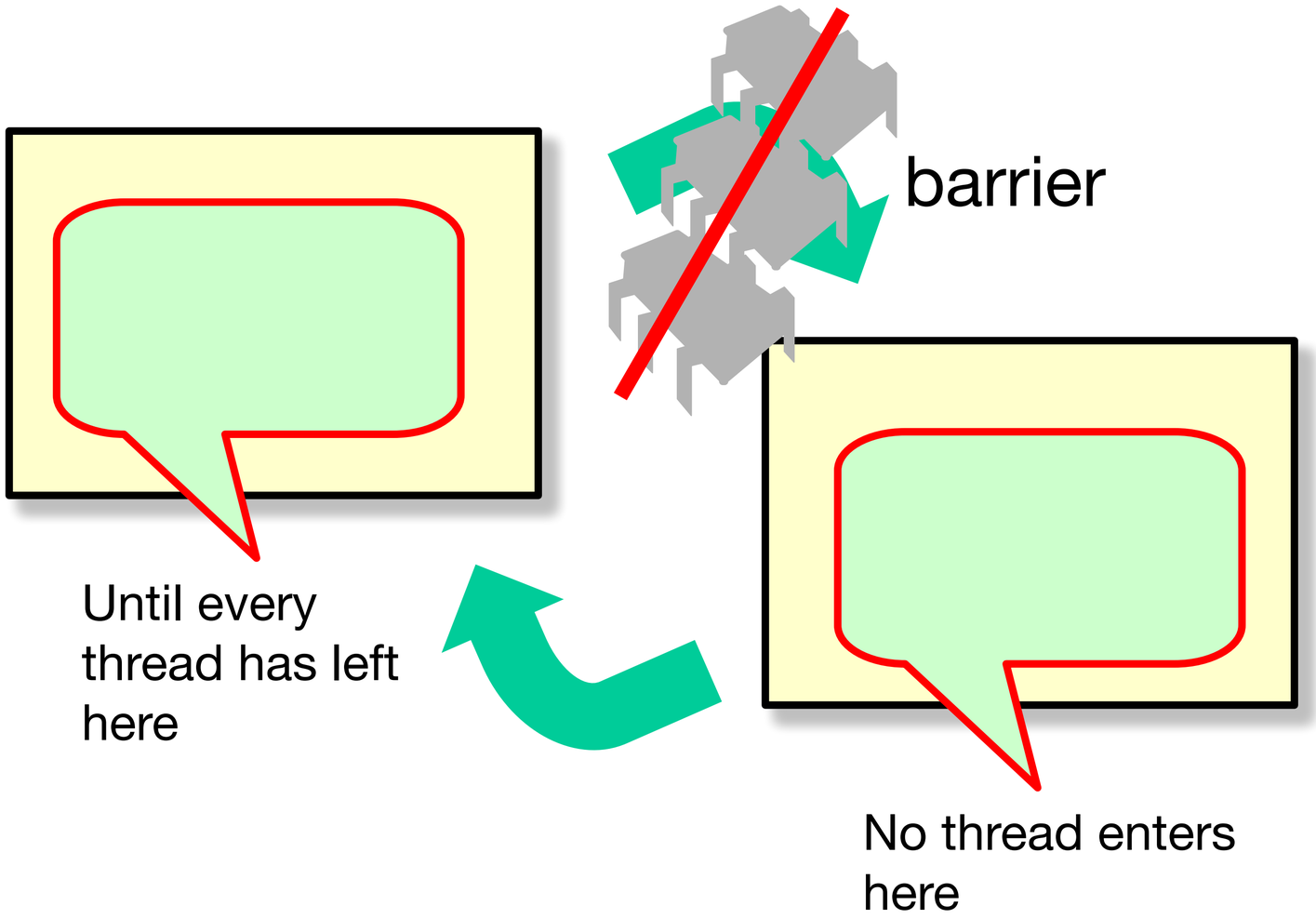
Barrier Synchronization



Barrier Synchronization



Barrier Synchronization



Sidebar: HW2 Part 4 with Barriers

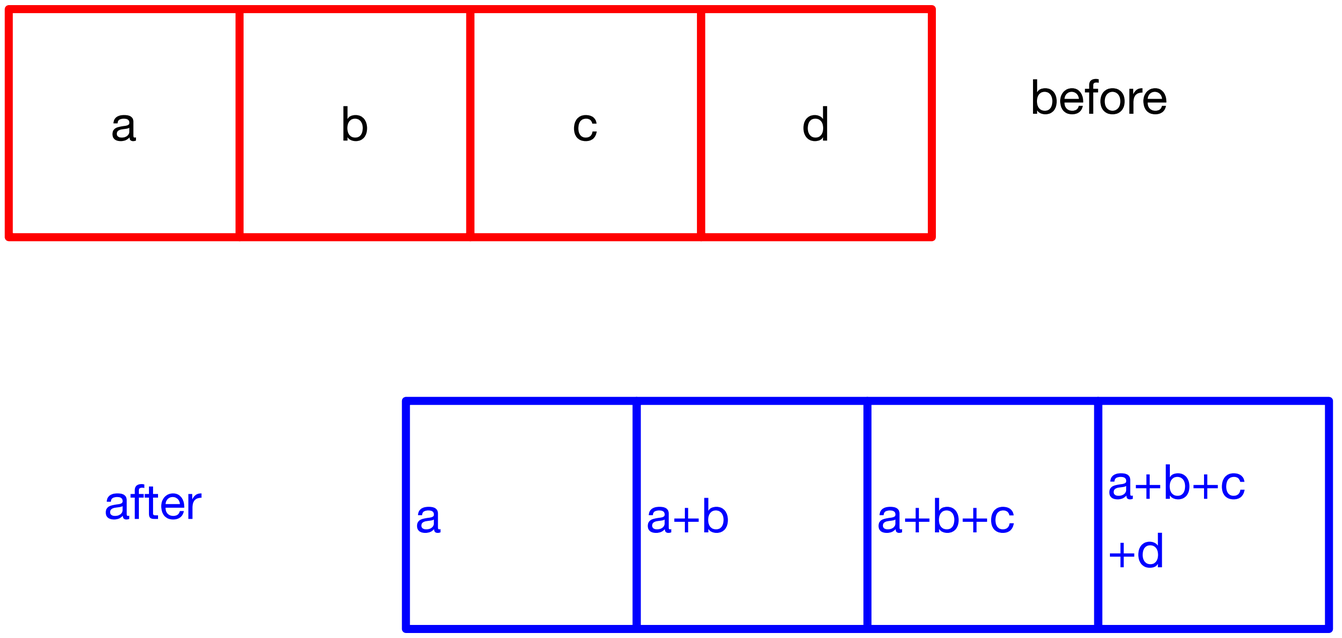
Why Do We Care?

- Mostly of interest to
  - Scientific & numeric computation
  - Distributed protocols
- Elsewhere
  - Garbage collection
  - Less common in systems programming
  - Still important topic

Duality

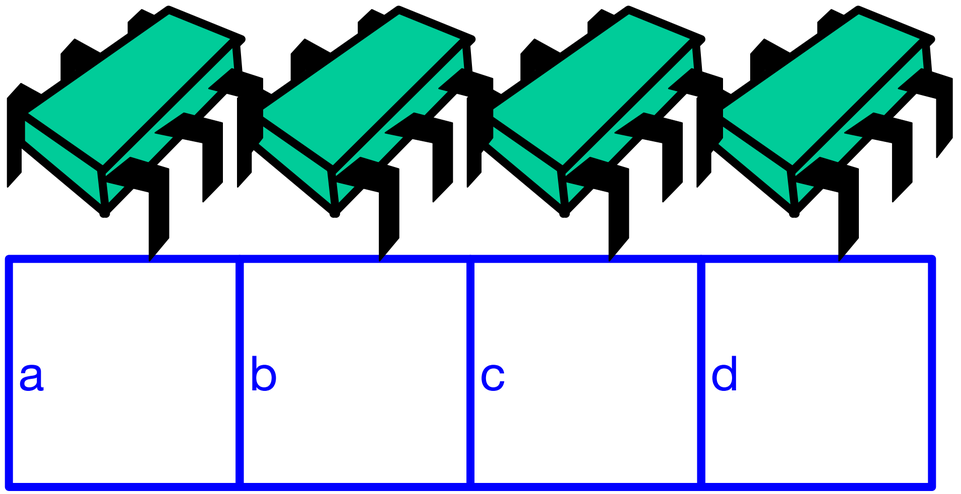
- Dual to mutual exclusion
  - Include others, not exclude them
- Same implementation issues
  - Interaction with caches …
    - Invalidation?
    - Local spinning?

Example: Parallel Prefix

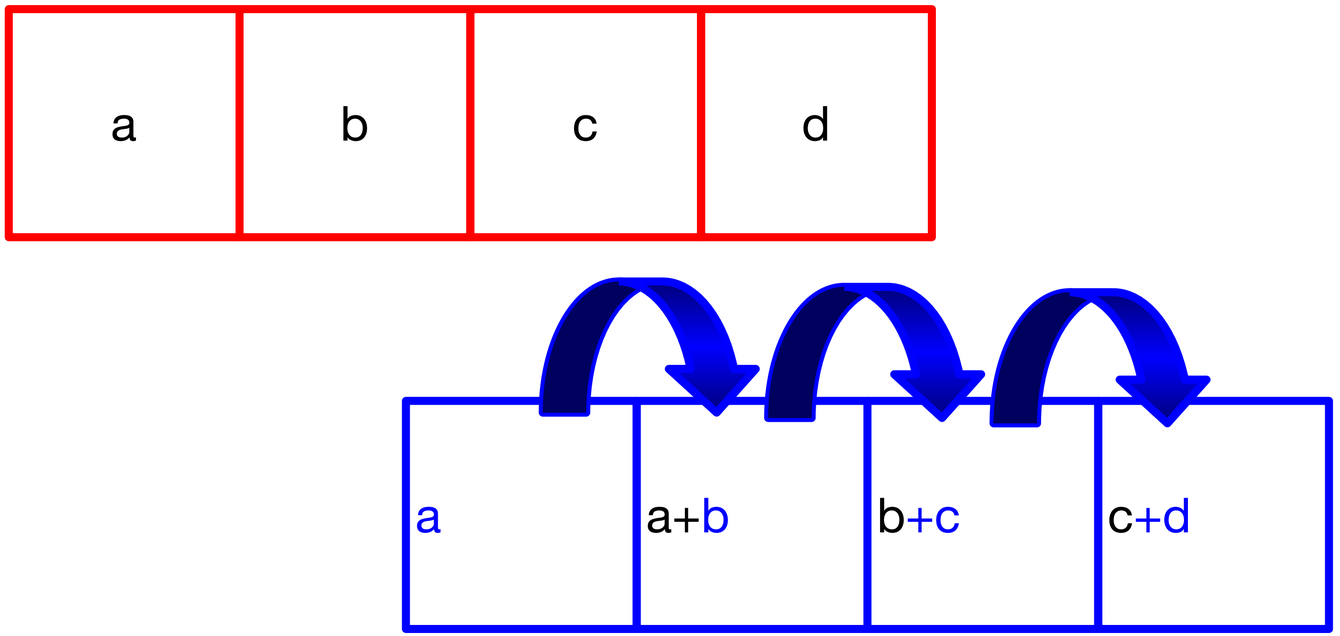


Parallel Prefix

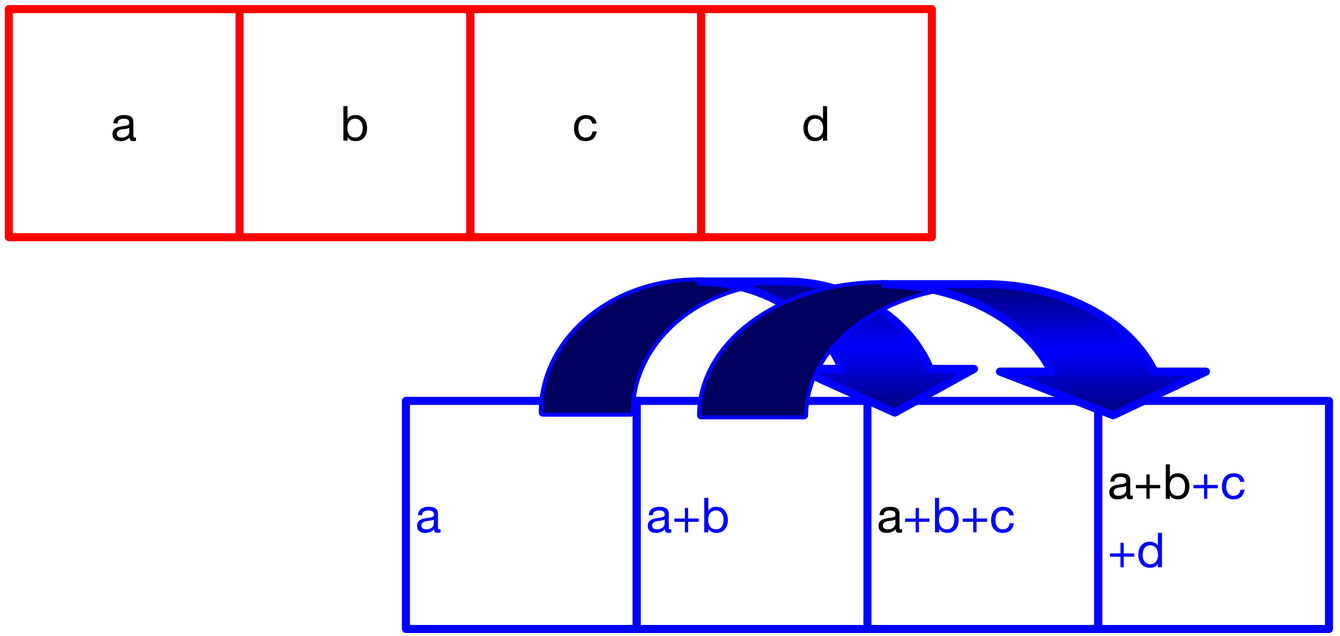
One thread
Per entry



Parallel Prefix: Phase 1



Parallel Prefix: Phase 2



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

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;
 }
```

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;
 }
```

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

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;
 }
```

Shared barrier

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;
 }
```

Initialize fields

Where Do the Barriers Go?

```
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?

```
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;
}}}
```

Where Do the Barriers Go?

```
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;
}}}
```

Make sure everyone reads before anyone writes

Where Do the Barriers Go?

```
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;
}}}
```

Make sure everyone reads before anyone writes

Where Do the Barriers Go?

```
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;
}}}
```

Make sure everyone reads before anyone writes

Make sure everyone writes before anyone reads

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?

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);
 }}}}
```

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);
 }}}}
```

Number threads not yet arrived

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);
 }}}}
```

Number threads participating

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);
 }}}}
```

Initialization

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);
 }}}}
```

Principal method

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);
 }}}}
```

If I'm last, reset fields for next time

Barriers

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);
 }}}}
```

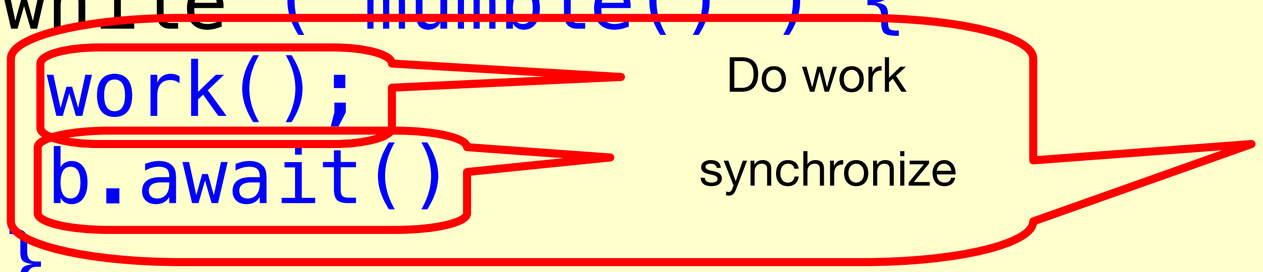
Barriers

```
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);
 }}}}
```

What's wrong with this protocol?

Reuse

```
Barrier b = new Barrier(n);
while ( mumble() ) {
  work();
  b.await()
}
```



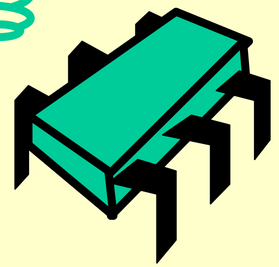
Barriers

```
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);
 }}}}
```

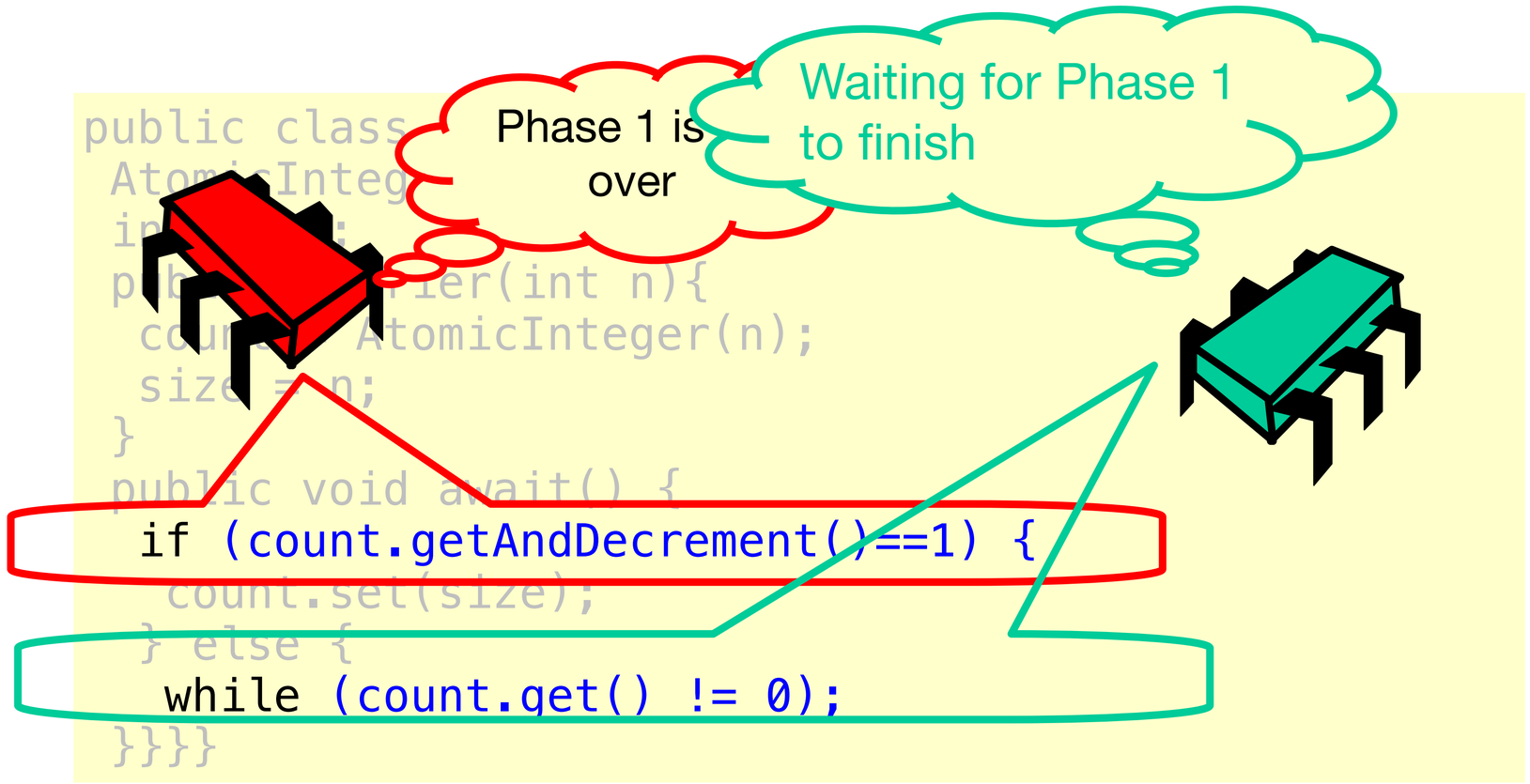
Barriers

```
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);
 }}}}
```

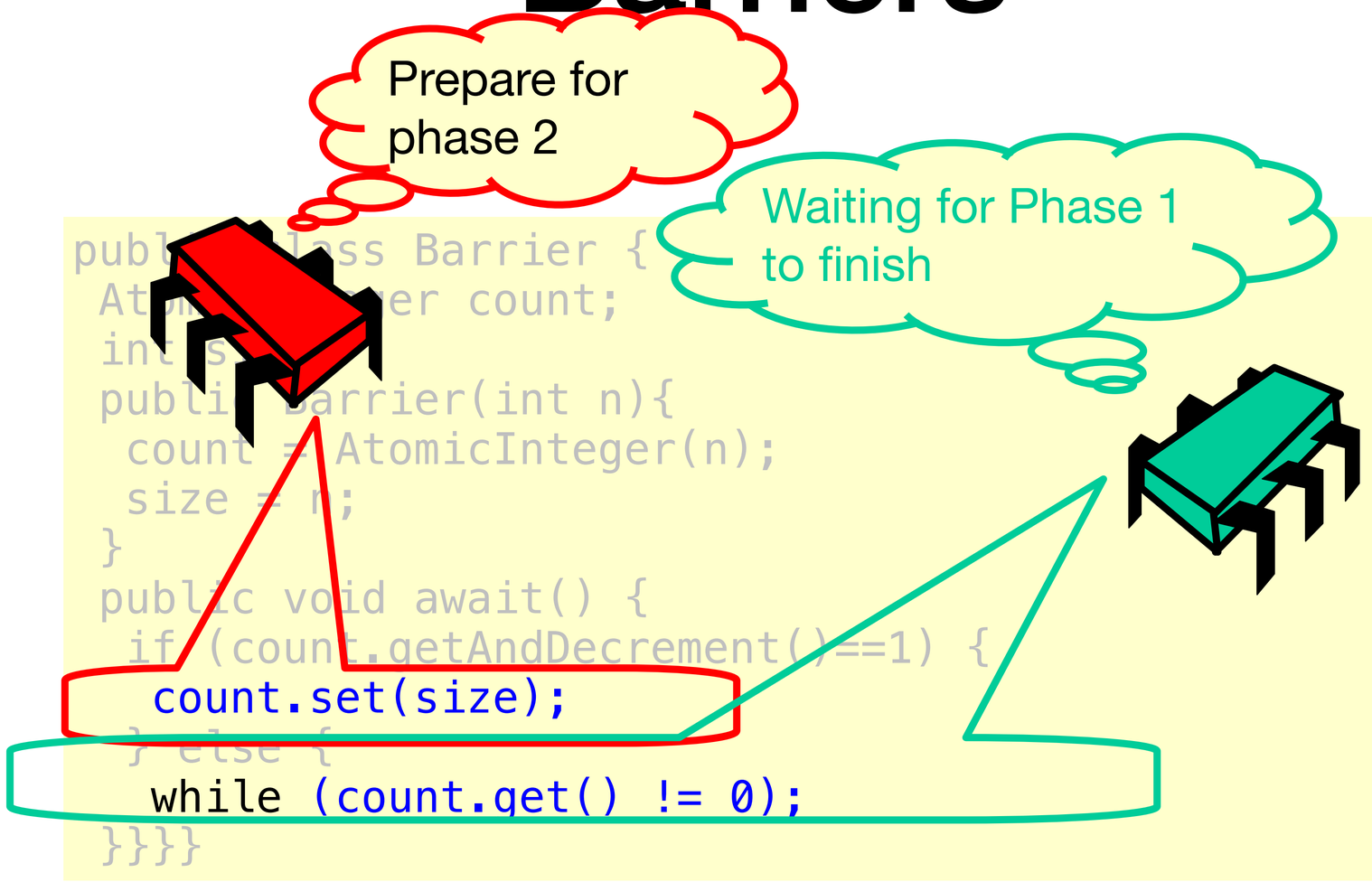
Waiting for Phase 1 to finish



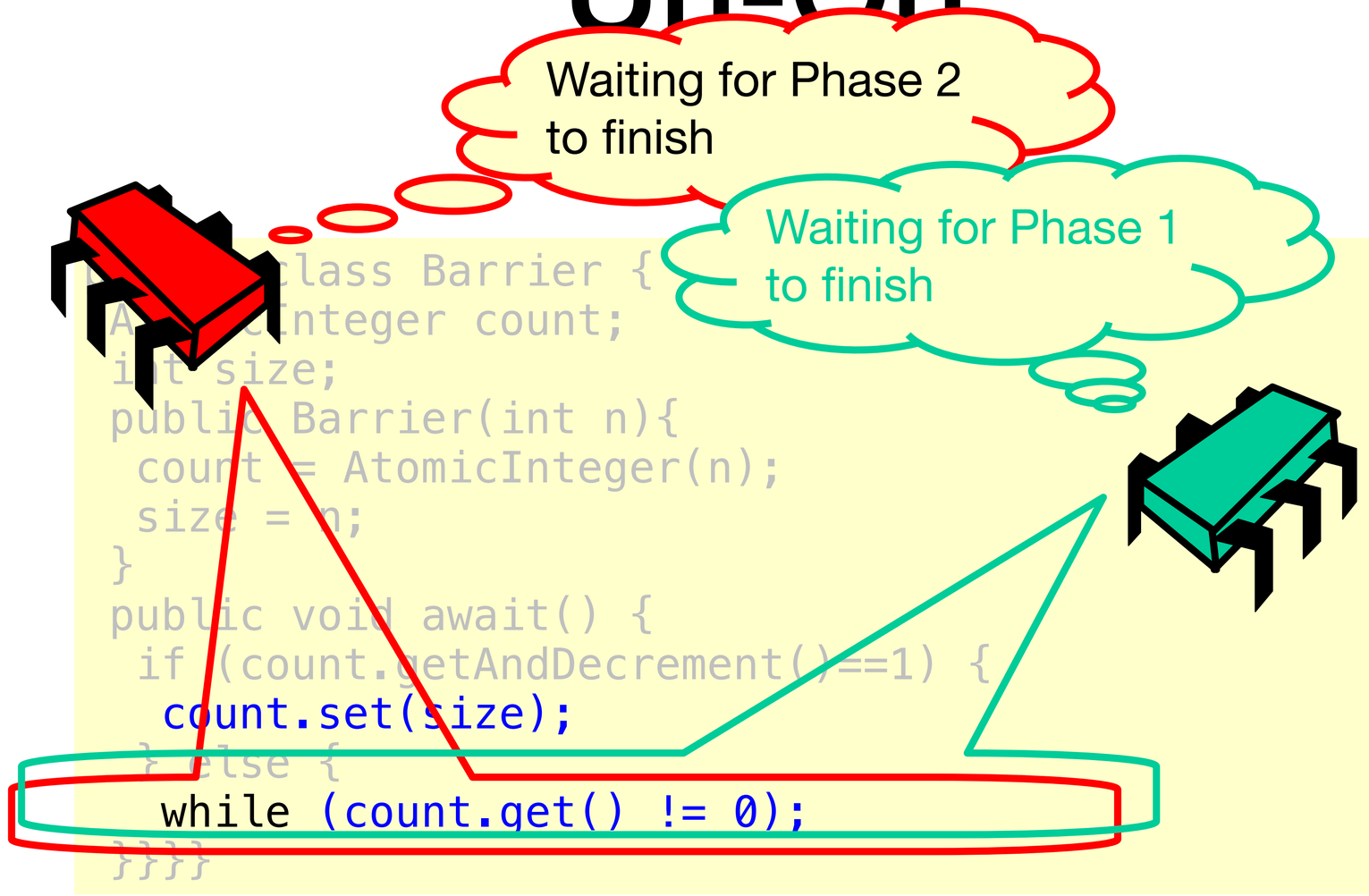
Barriers


Phase 1 is over
Waiting for Phase 1 to finish
public class
AtomicInteg
in
pub ier(int n){
cou AtomicInteger(n);
size = n;
}
public void await() {
if (count.getAndDecrement()==1) {
count.set(size);
} else {
while (count.get() != 0);
}}}}

Barriers


Prepare for phase 2
Waiting for Phase 1 to finish
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);
}}}}

Uh-Oh


Waiting for Phase 2
to finish
Waiting for Phase 1
to finish
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

Sense-Reversing 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)}}}
```

Sense-Reversing 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)}}}
```

Completed odd or even-numbered phase?

Sense-Reversing 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)}}}
```

Store sense for next phase, initialized to true

Sense-Reversing 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)}}}
```

Get new sense determined by last phase

Sense-Reversing 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)}}}
```

If I'm last, reverse sense for next time

Sense-Reversing 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)}}}
```

Otherwise, wait for sense to flip

Sense-Reversing Barriers

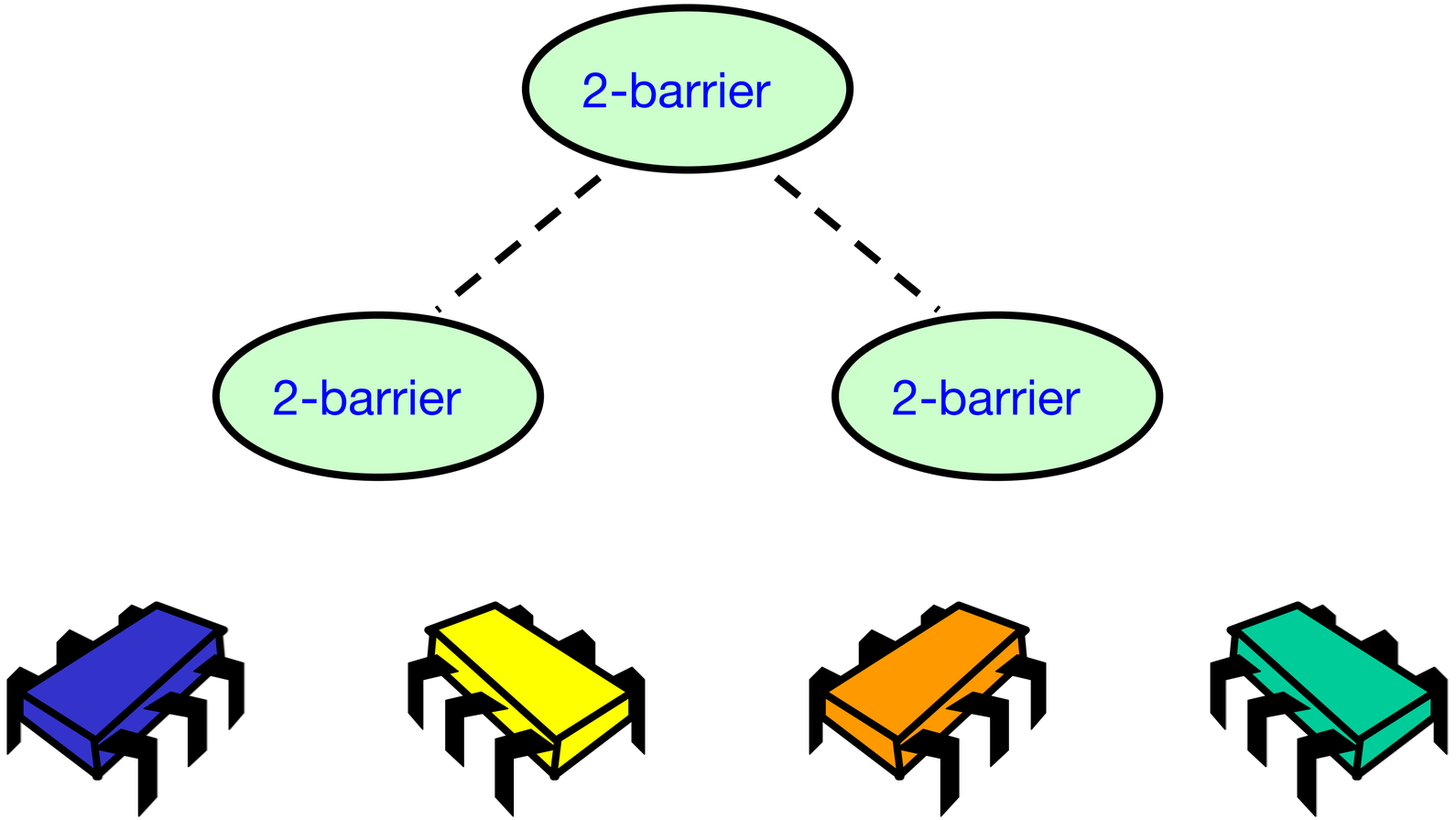
```
public class Barrier {
 AtomicInteger count;
 int size;
 boolean sense = false;
 threadSense = new ThreadLocal<boolean>…

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

Prepare sense for next phase

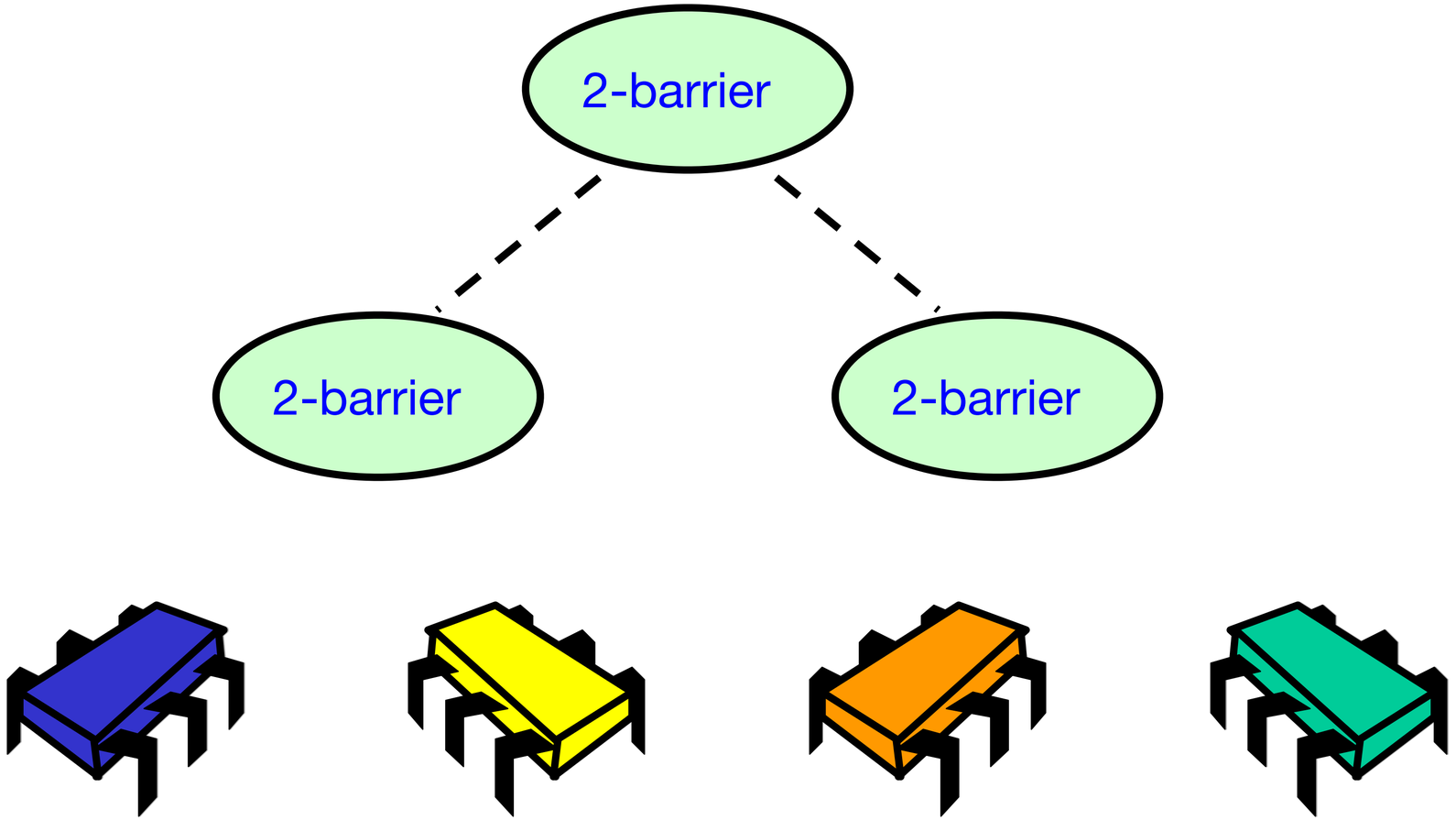
Sense Reversing Barriers

- 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



Combining Tree Barriers



Combining Tree Barrier

```
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

```
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

Combining Tree Barrier

```
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) {}
 }…}}}
```

Am I last?

Combining Tree Barrier

```
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) {}
 }…}}}
```

Proceed to parent barrier

Combining Tree Barrier

```
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) {}
 }…}}}
```

Prepare for next phase

Combining Tree Barrier

```
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) {}
 }…}}}
```

Notify others at this node

Combining Tree Barrier

```
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) {}
 }…}}}
```

I'm not last, so wait for notification

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

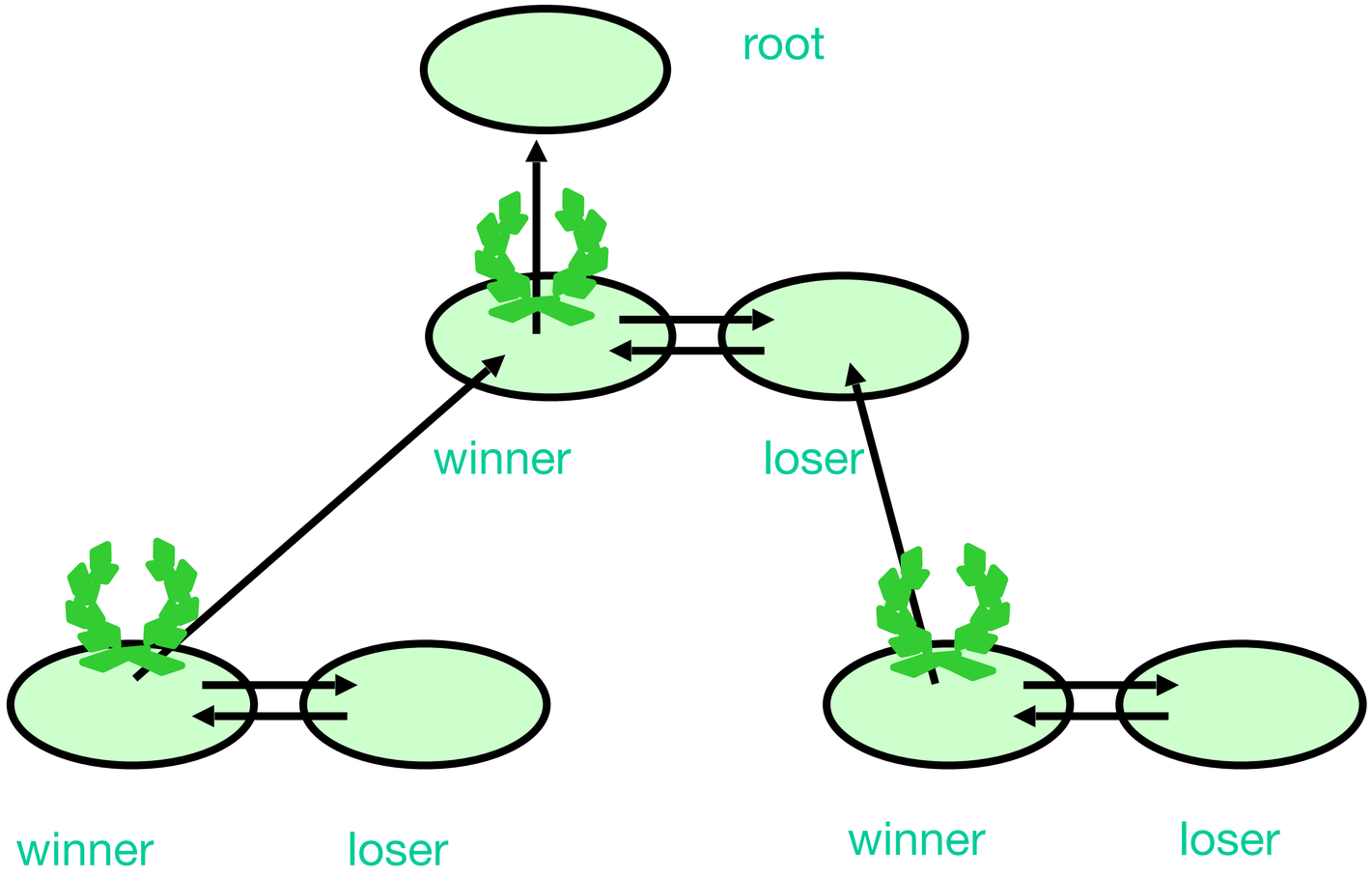
Remarks

- 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

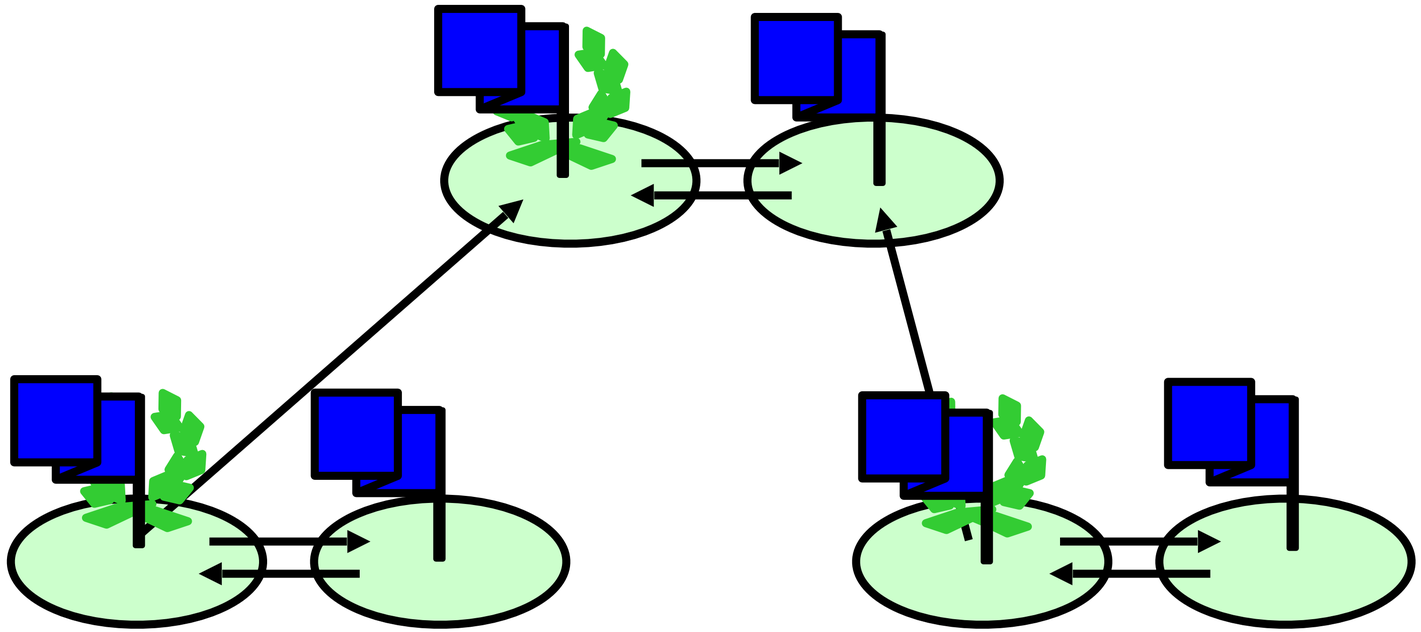
Tournament Tree Barrier

- 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

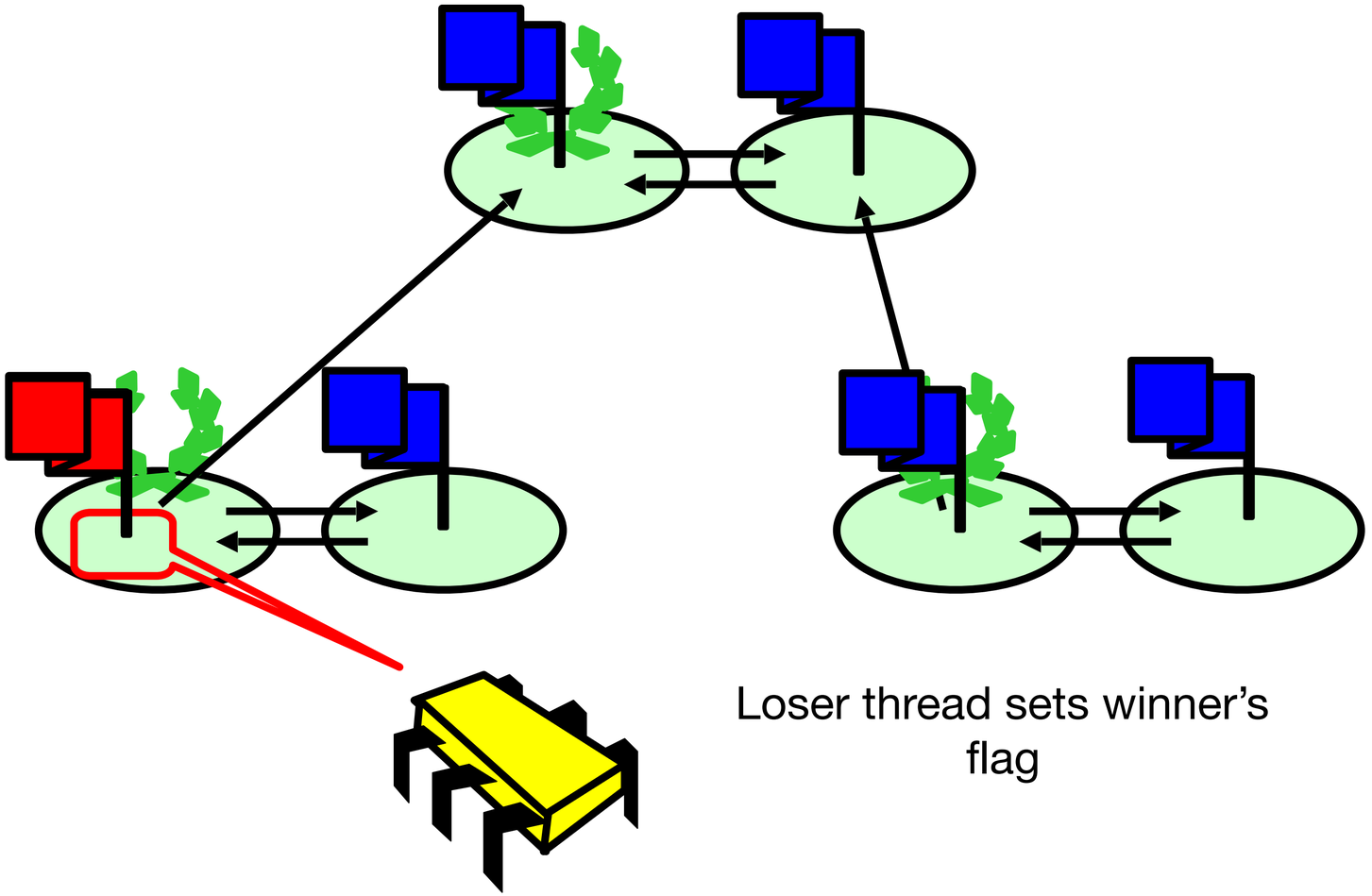


Tournament Tree Barriers



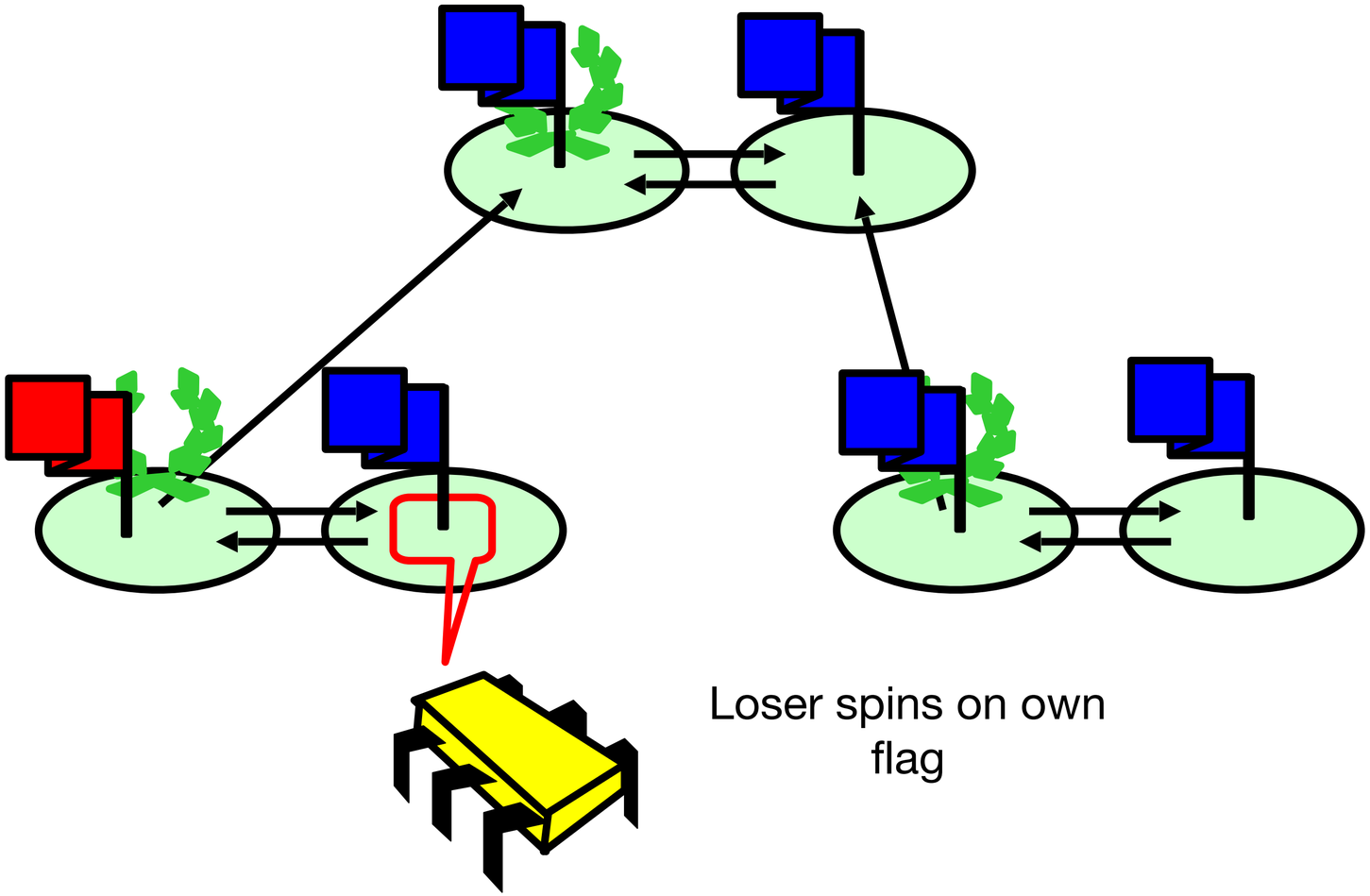
All flags blue

Tournament Tree Barriers



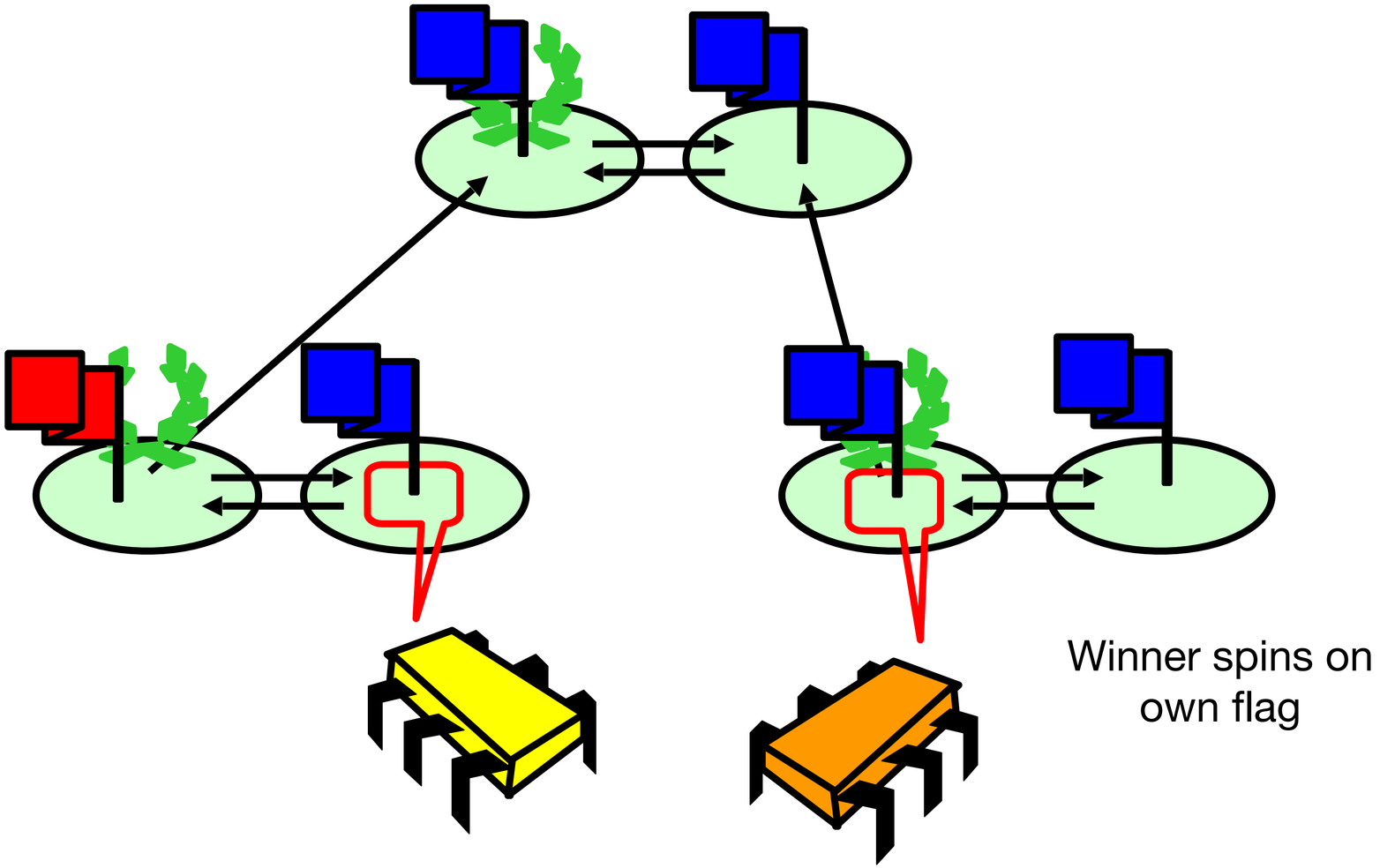
Loser thread sets winner's flag

Tournament Tree Barriers

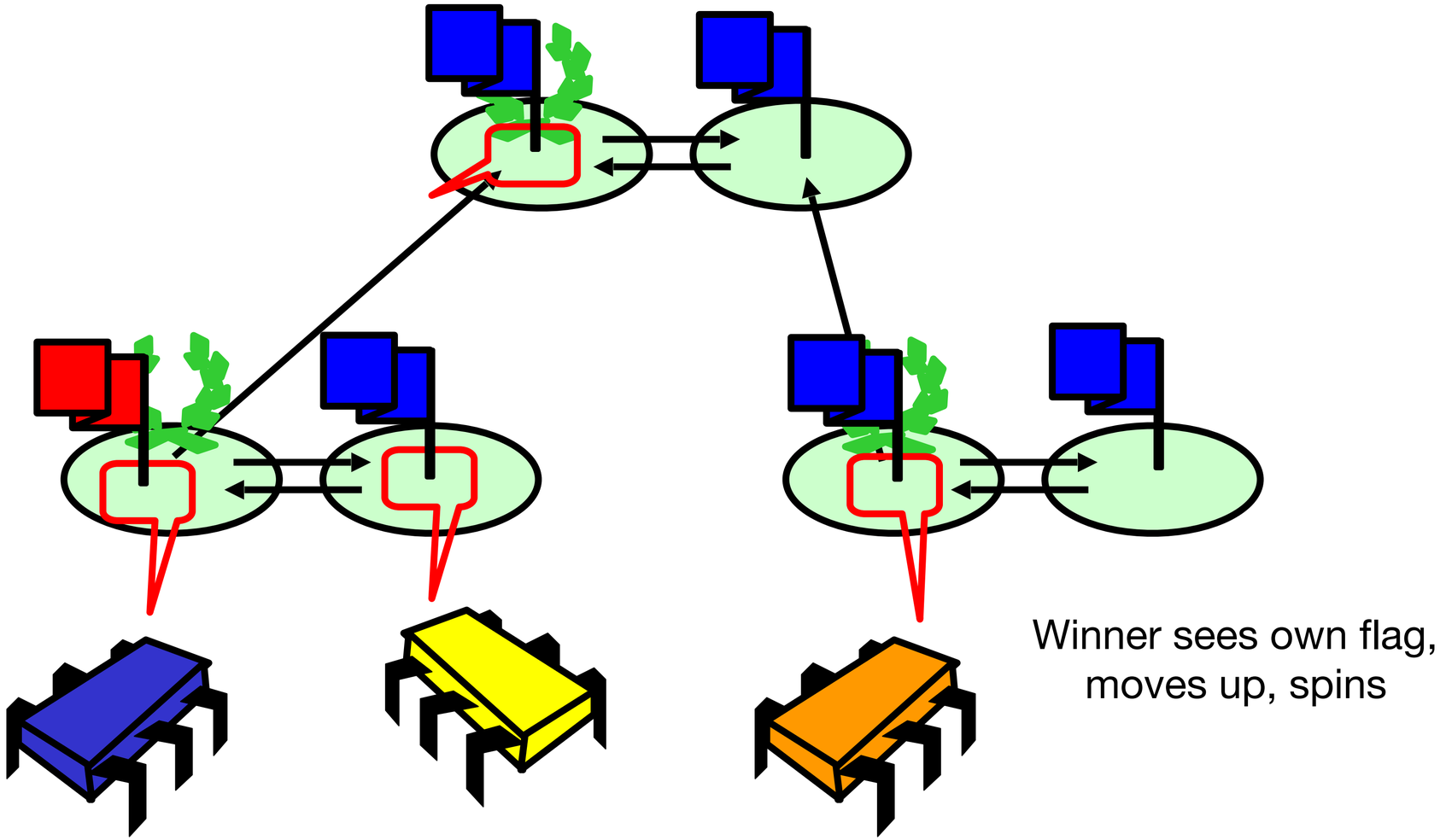


Loser spins on own flag

Tournament Tree Barriers

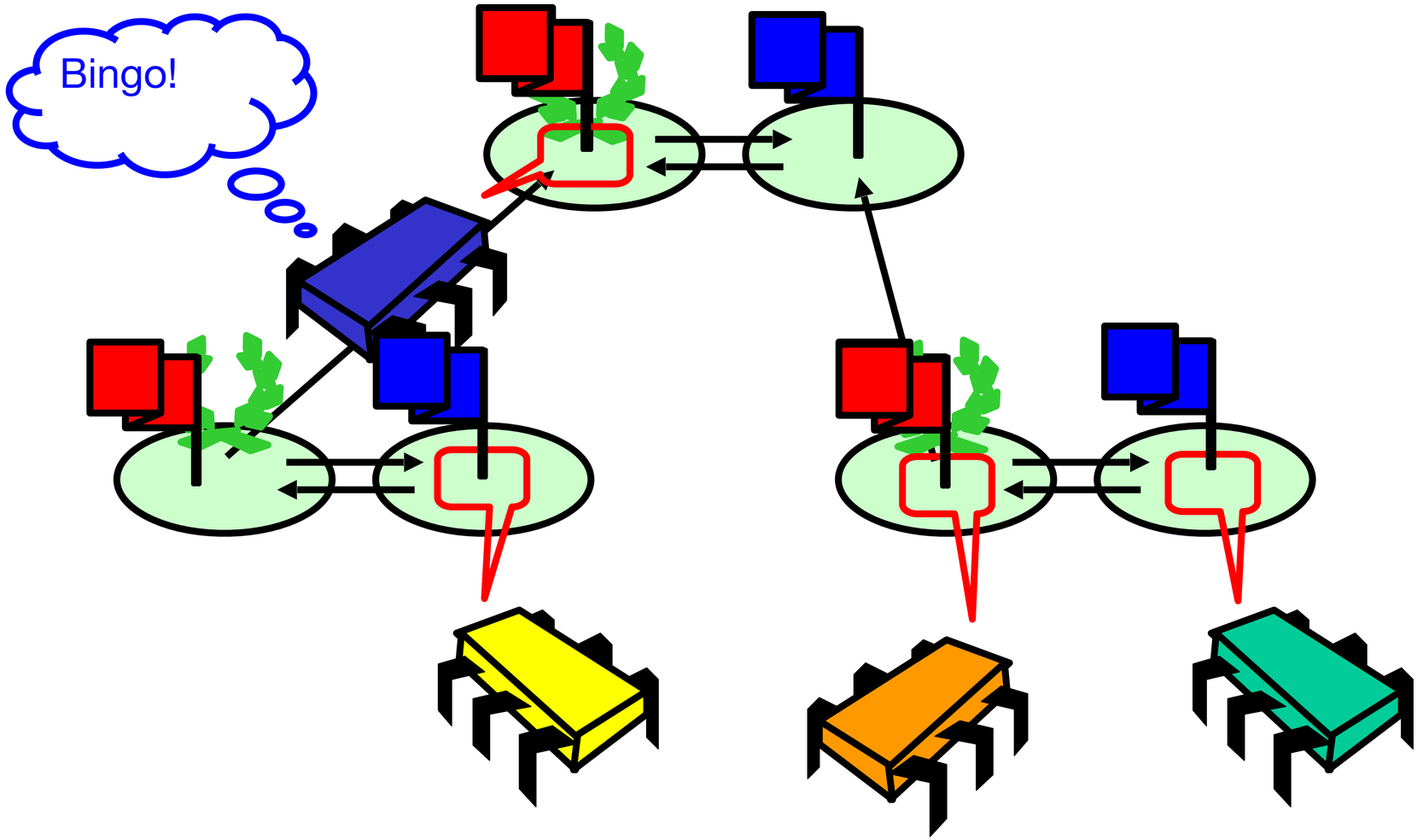


Tournament Tree Barriers

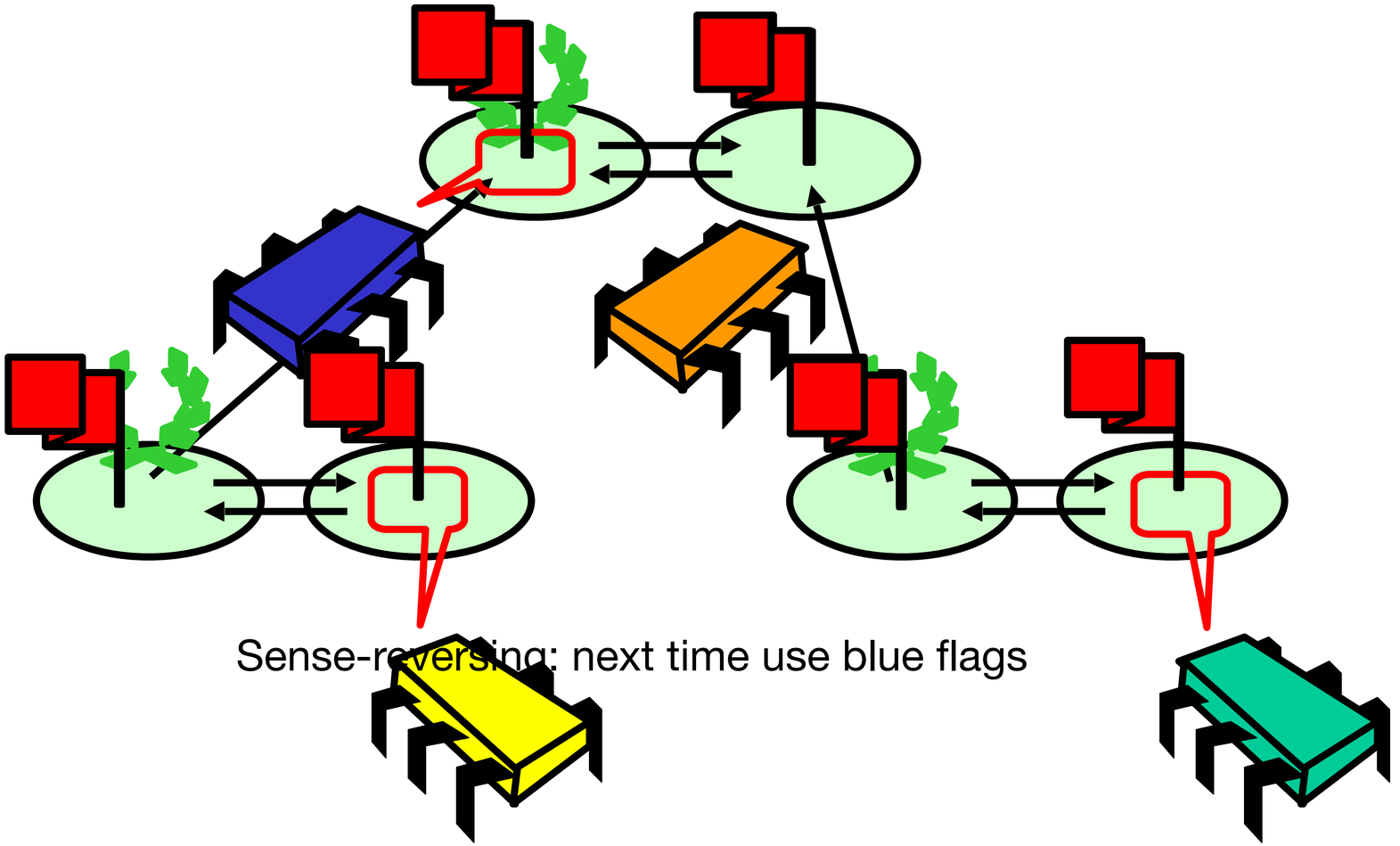


Winner sees own flag, moves up, spins

Tournament Tree Barriers



Tournament Tree Barriers



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:

Speedup = 1/(ParallelPart/N + SequentialPart)

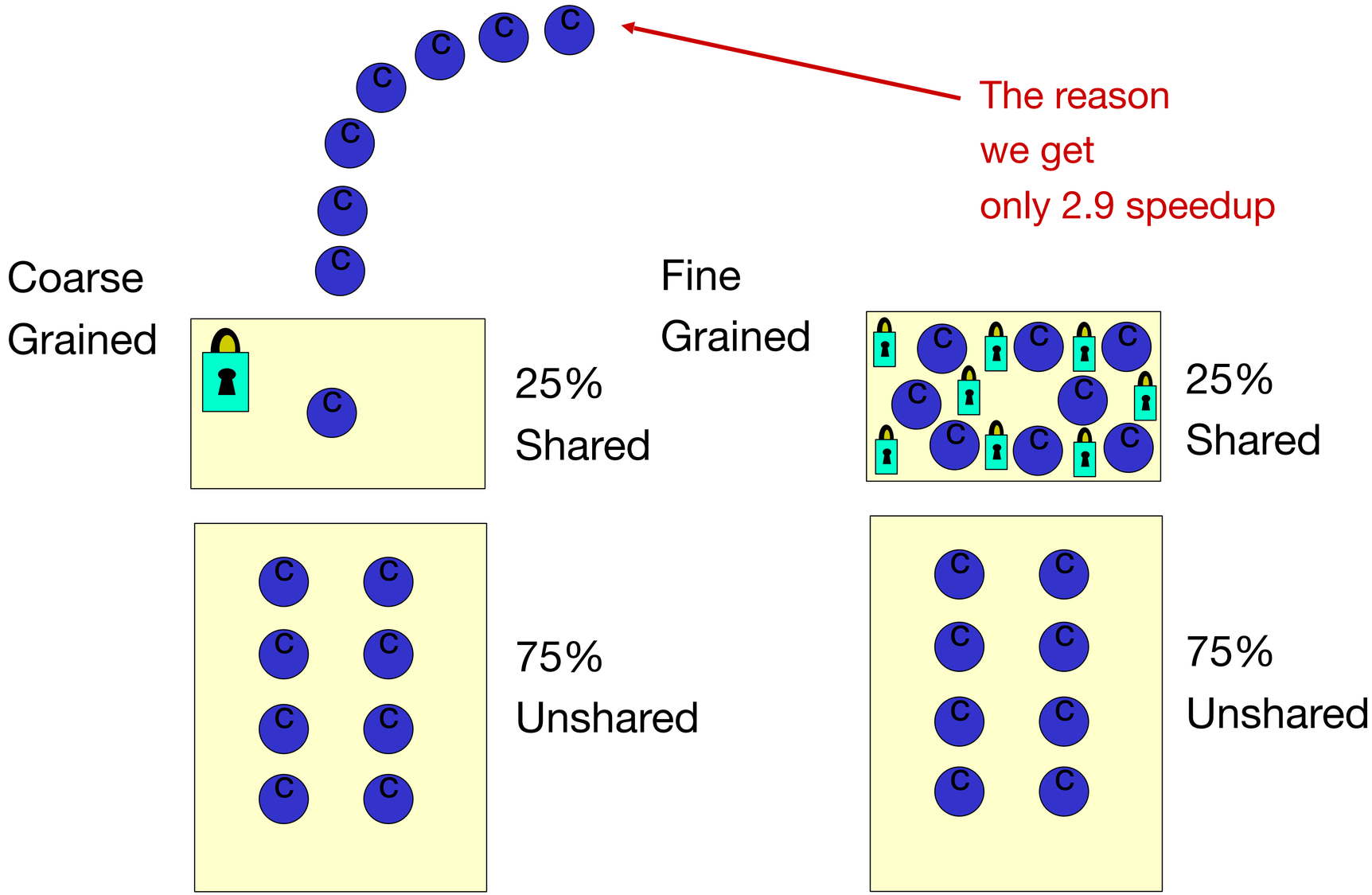
Pay for N = 8 cores

SequentialPart = 25%

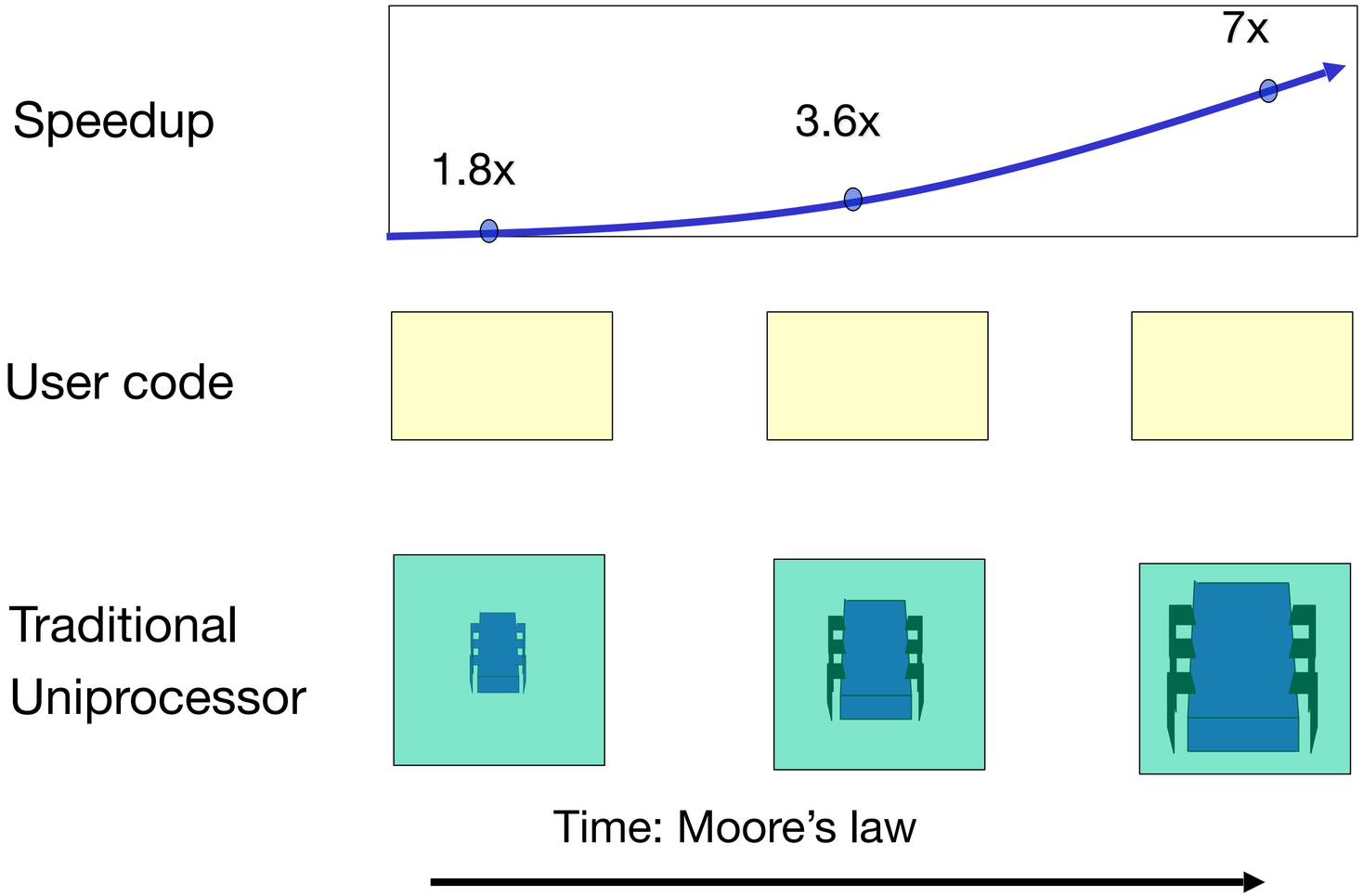
Speedup = only 2.9 times!

Must parallelize applications on a very fine grain!

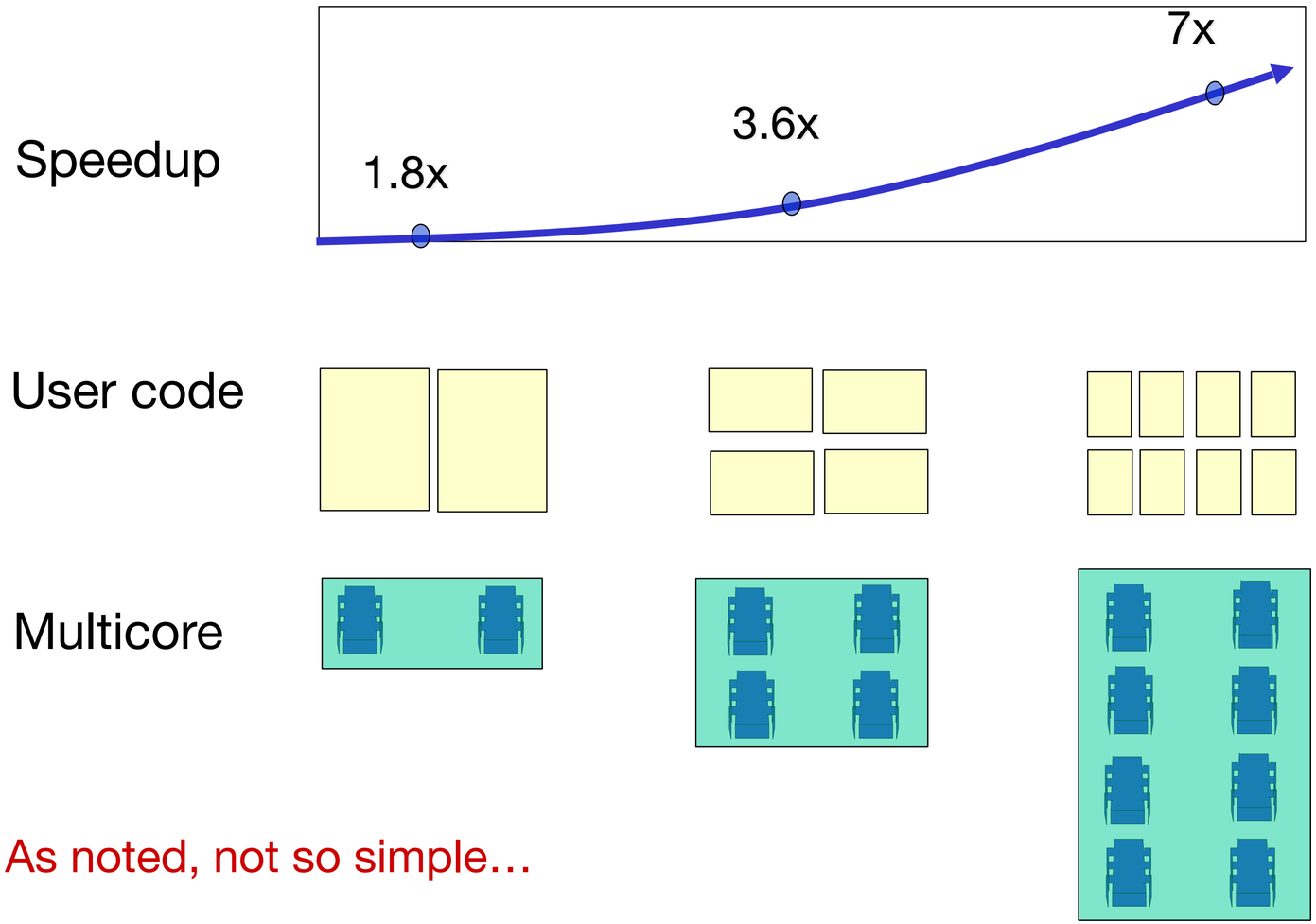
Need Fine-Grained Locking



Traditional Scaling Process

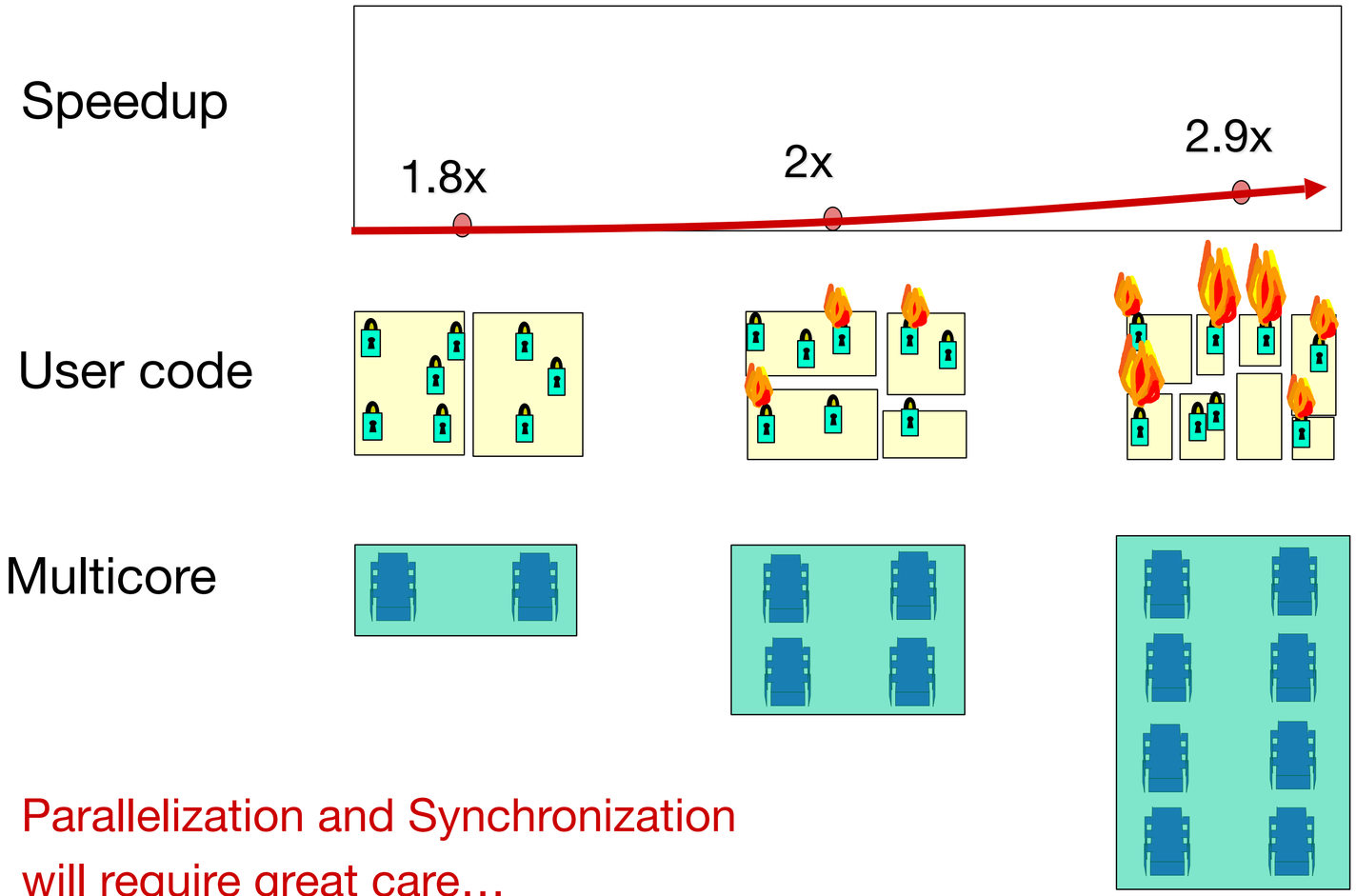


Multicore Scaling Process



As noted, not so simple…

Real-World Scaling Process



Parallelization and Synchronization will require great care…

Multicore Programming

- 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