Barrier Synchronization

Christof Fetzer, TU Dresden

Based on slides by Maurice Herlihy and Nir Shavit

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

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

- What about overlapping work?
  - 1st thread displays frame
  - 2nd prepares next frame

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;
}
```

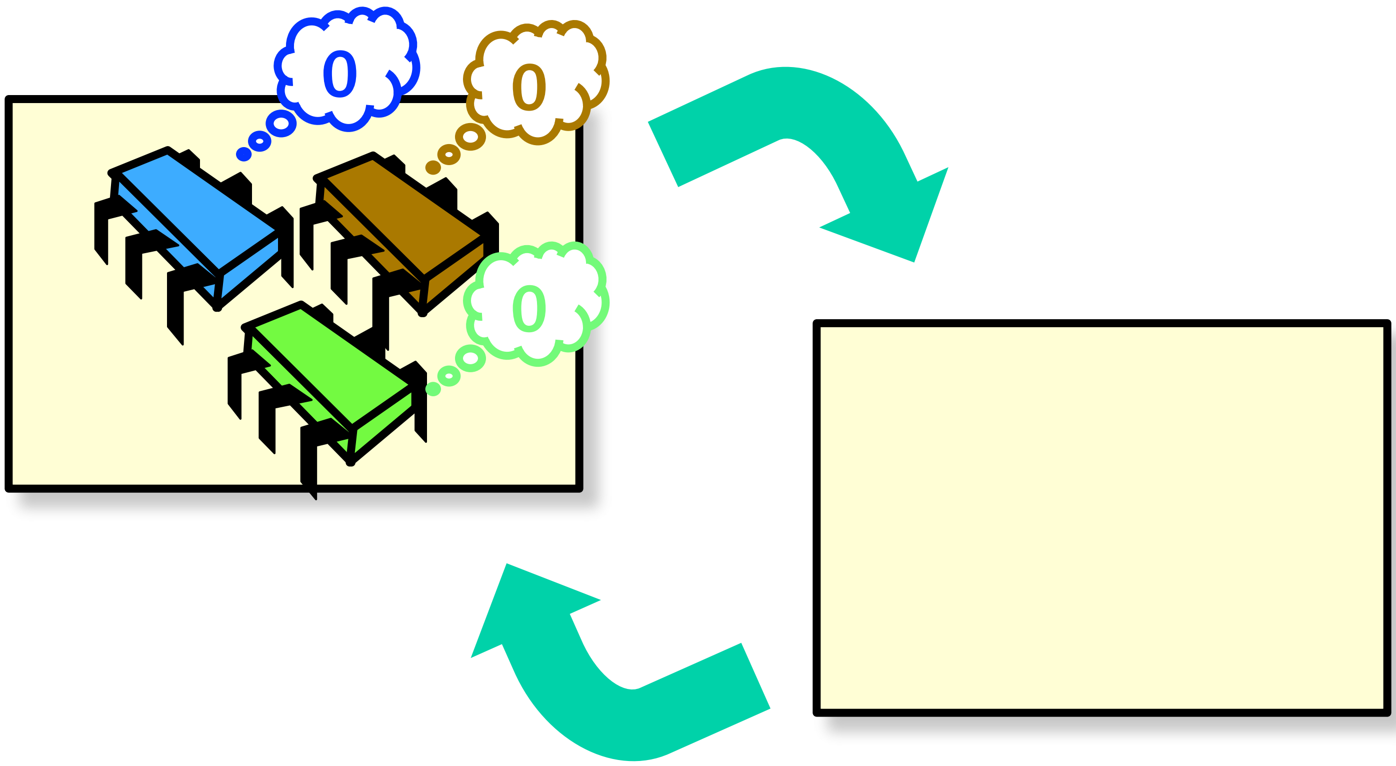
```
while (true) {
  if (phase) {
    frame[1].prepare();
  } else {
    frame[0].prepare();
  }
  phase = !phase;
}
```

odd phases

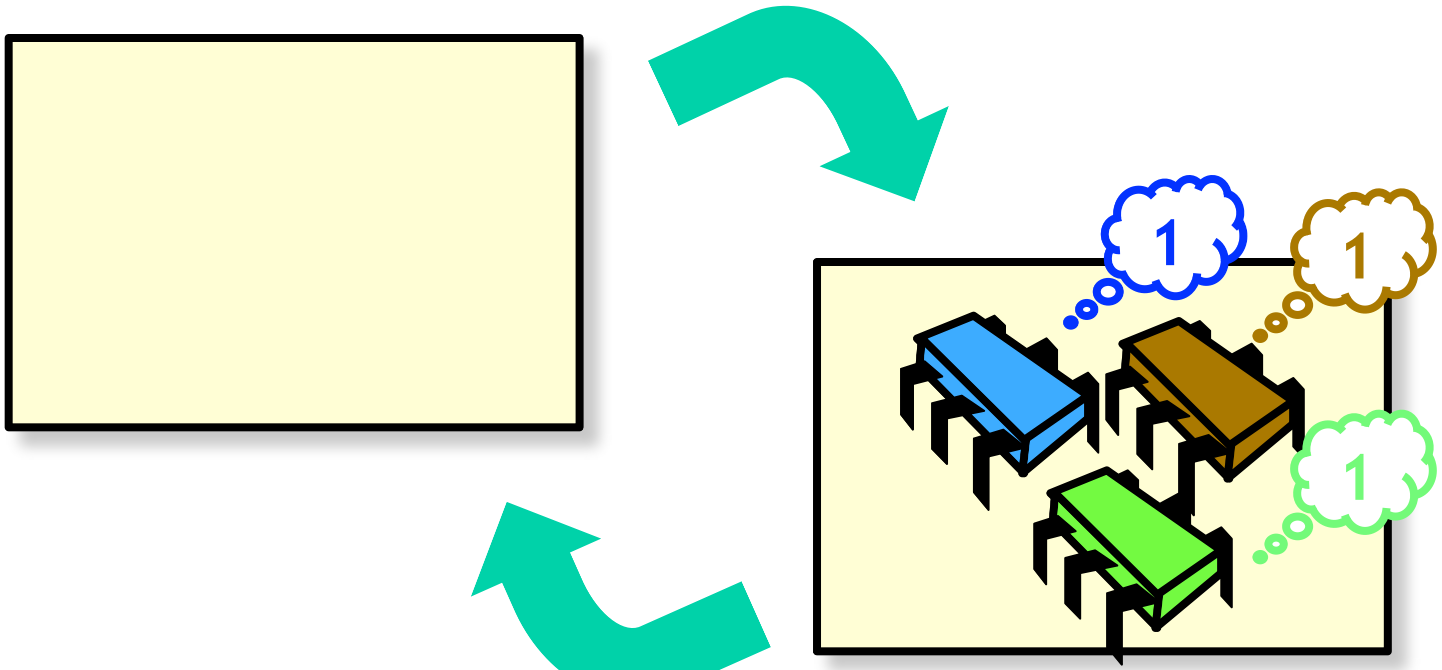
Synchronization Problems

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

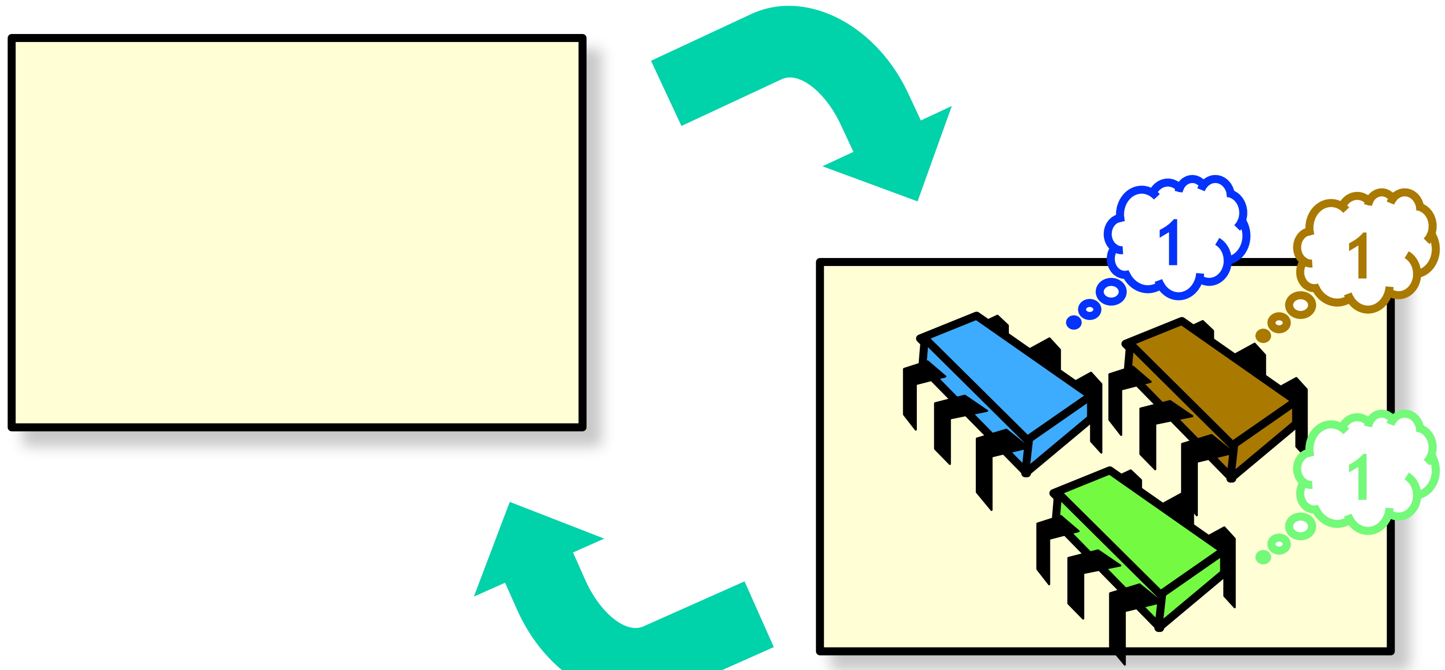
Ideal Parallel Computation



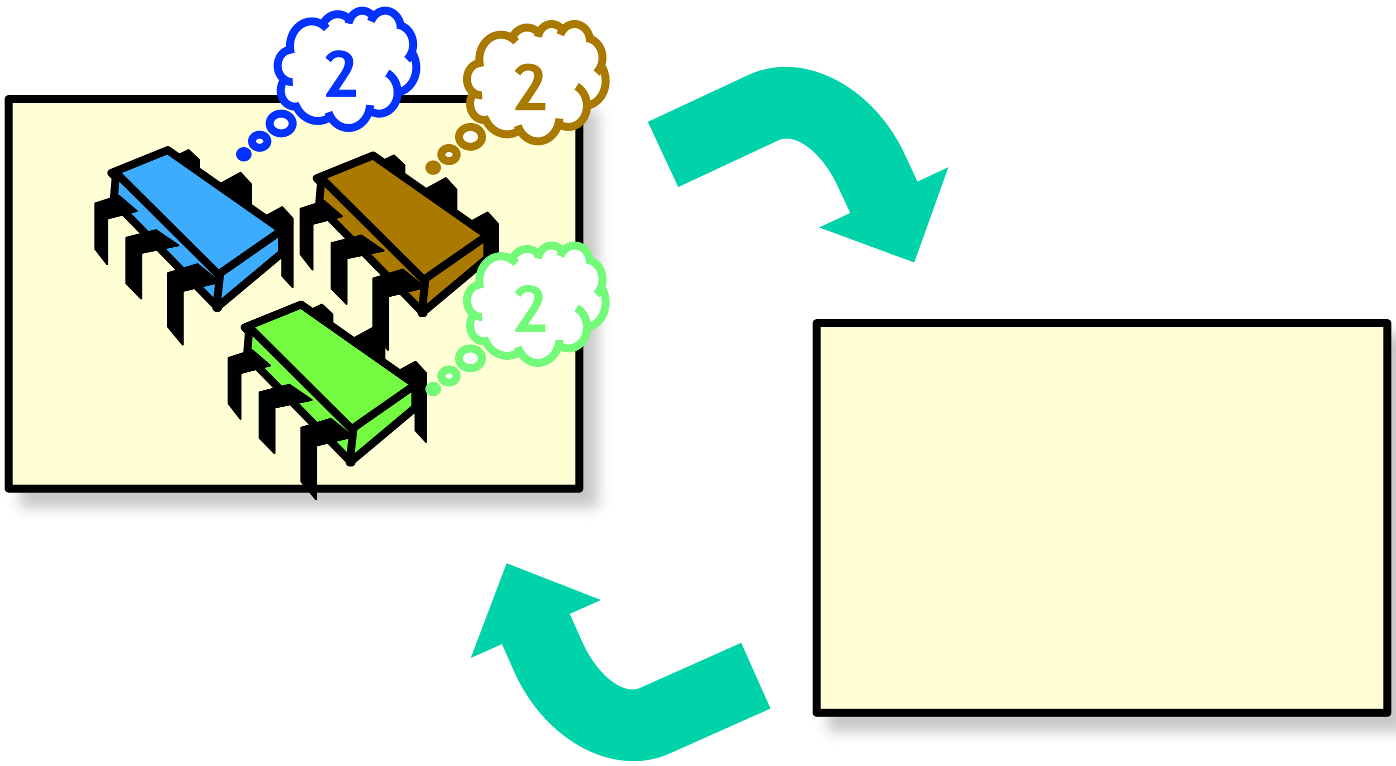
Ideal Parallel Computation



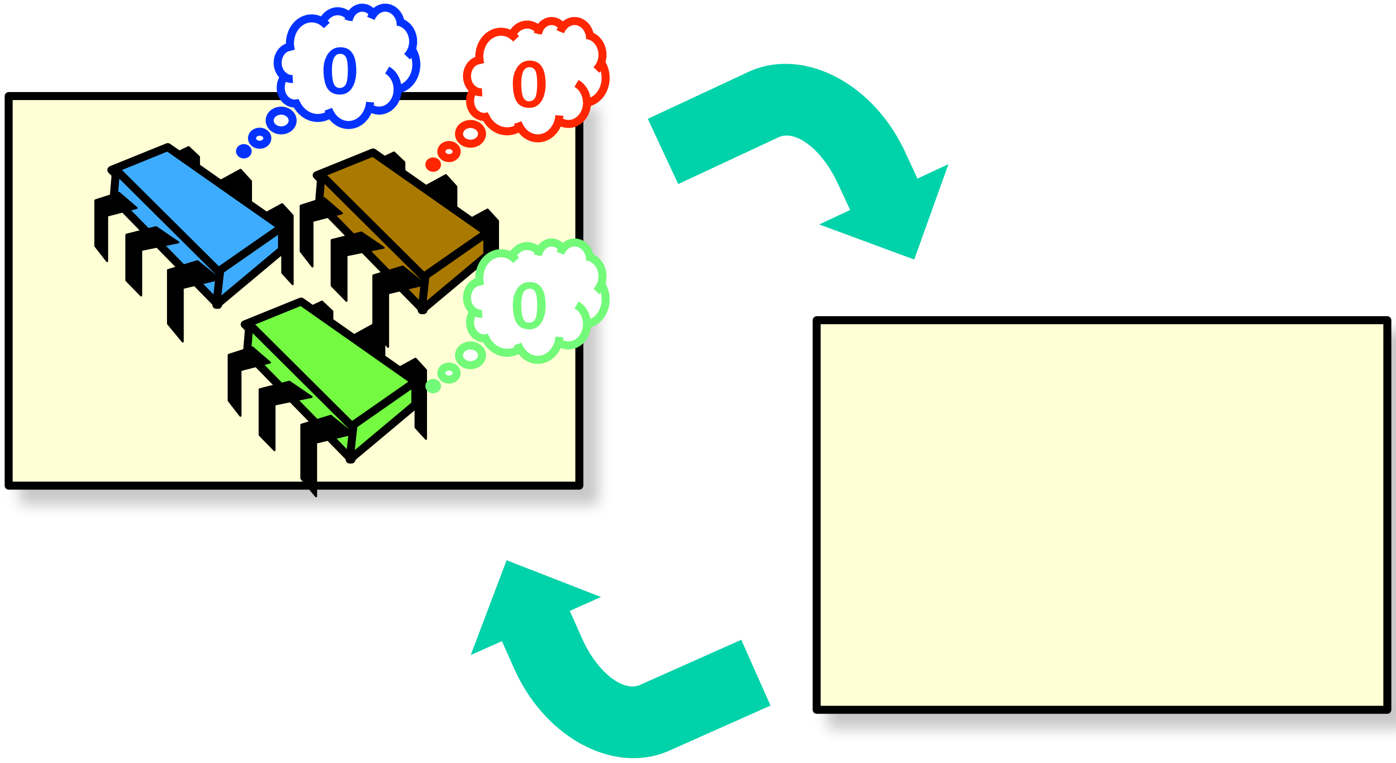
Ideal Parallel Computation



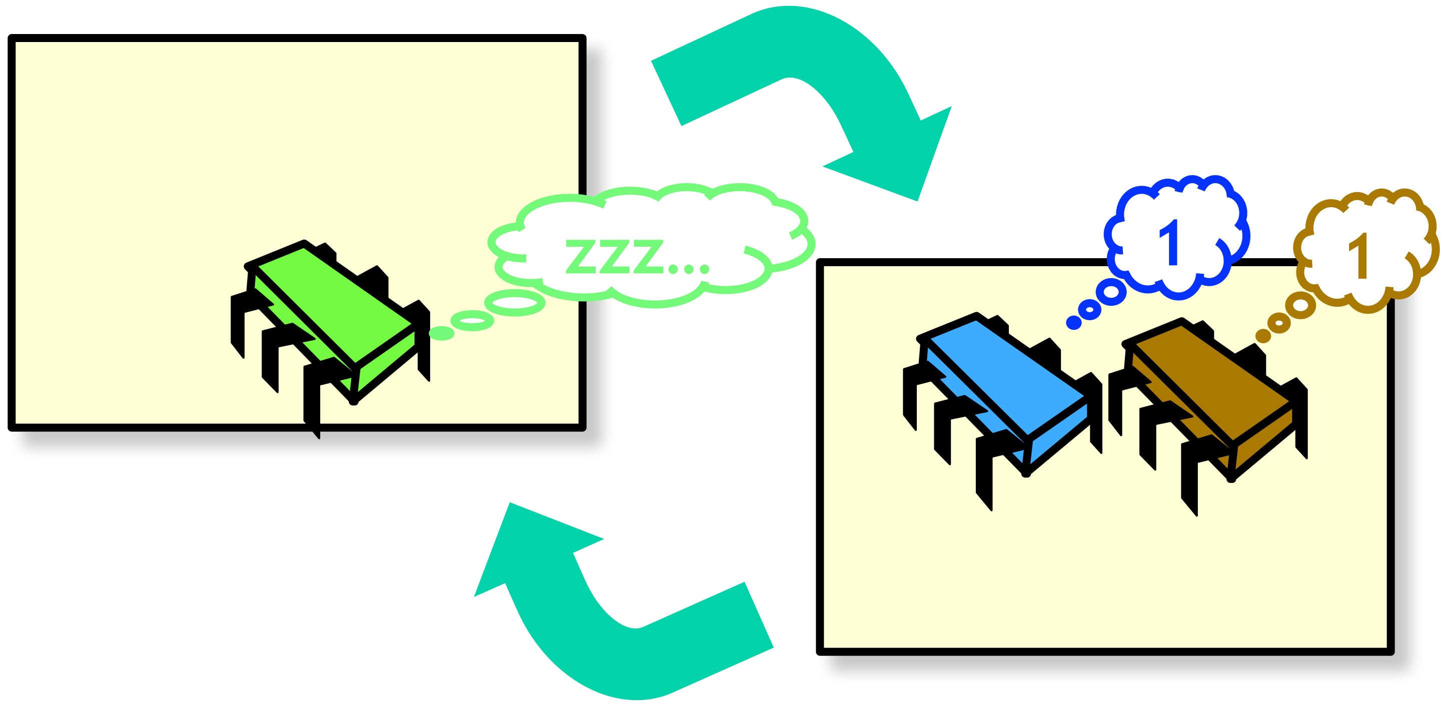
Ideal Parallel Computation



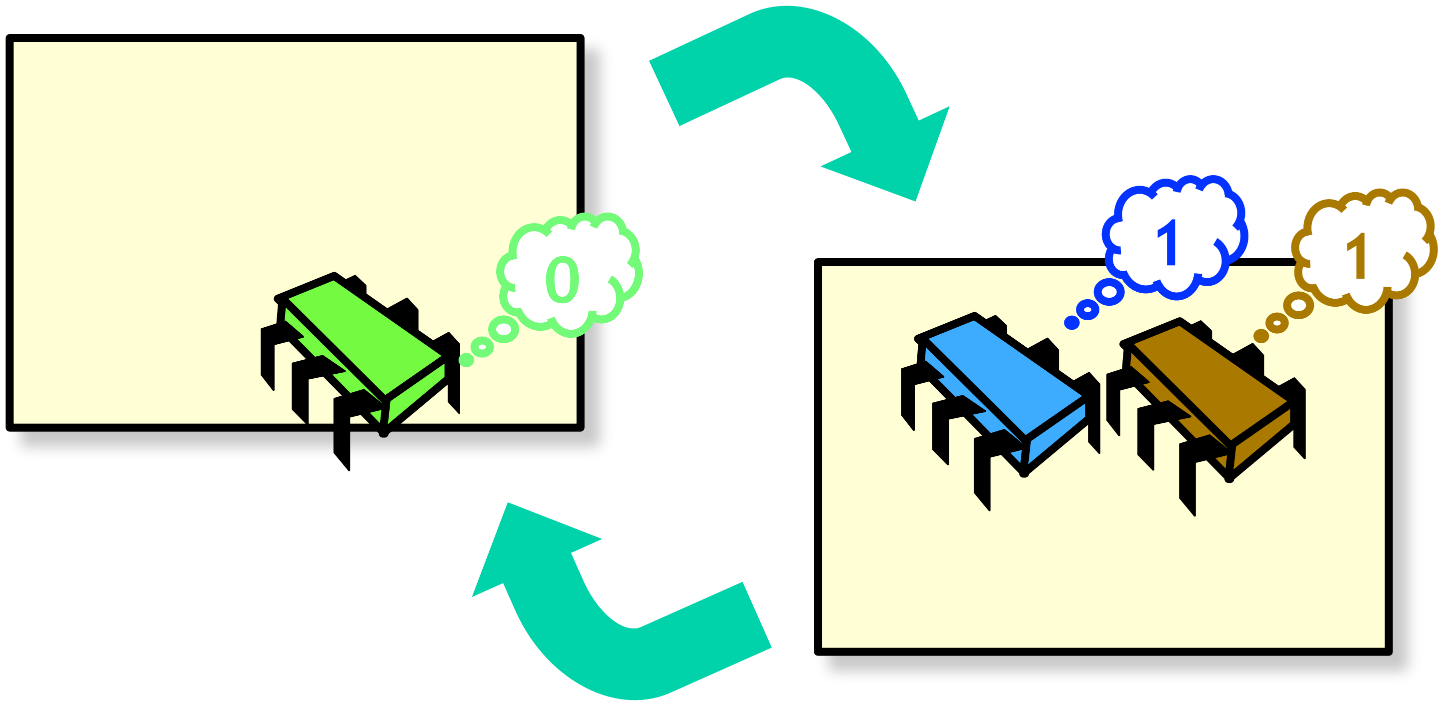
Real-Life Parallel Computation



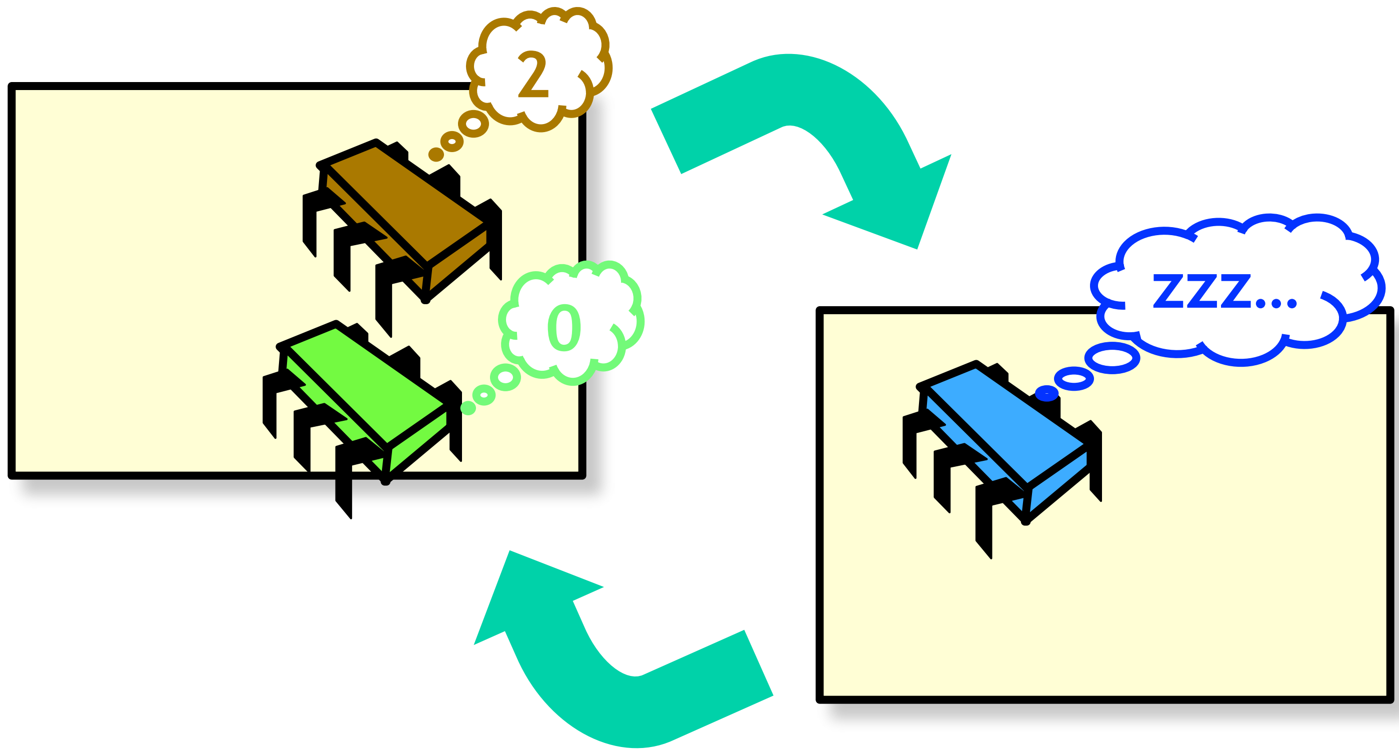
Real-Life Parallel Computation



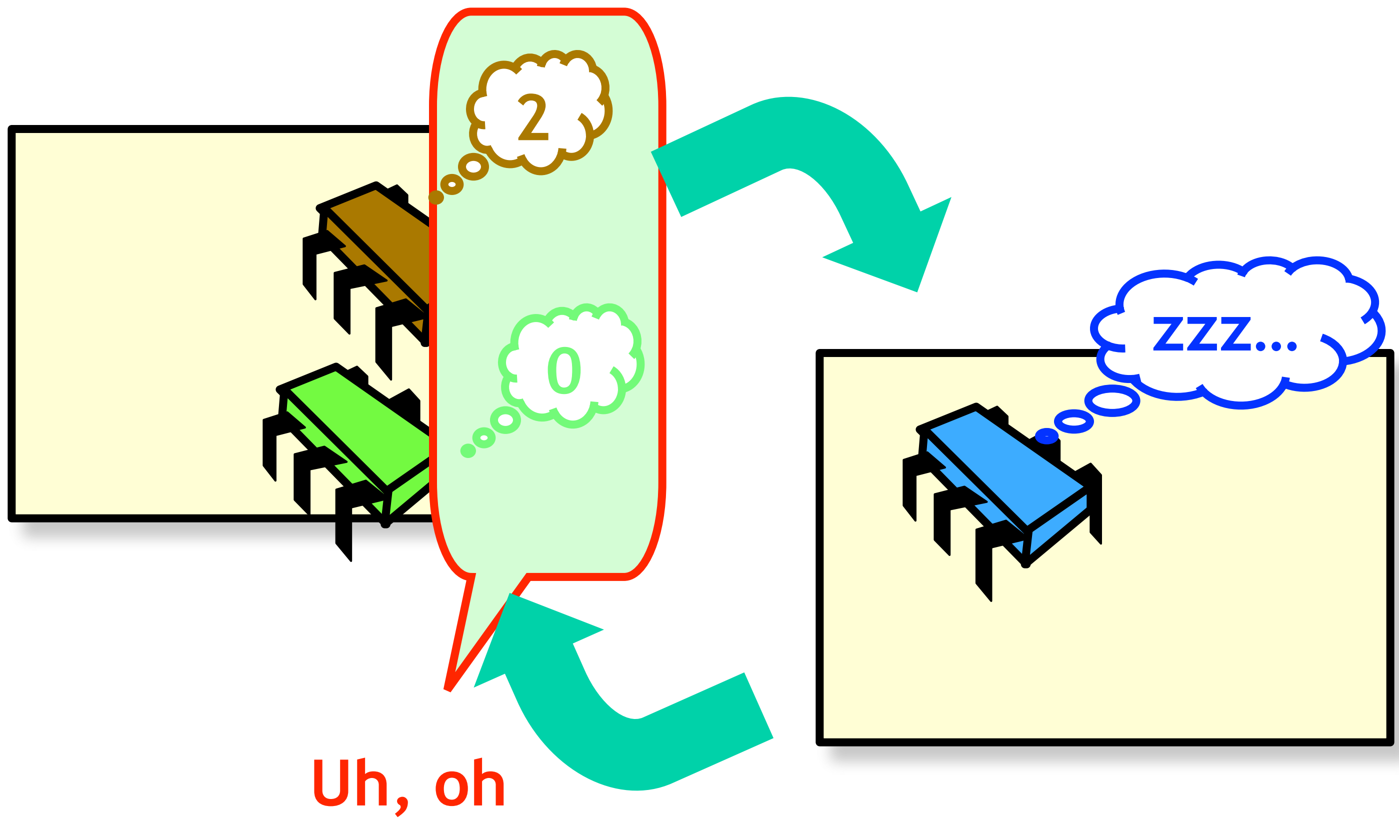
Real-Life Parallel Computation



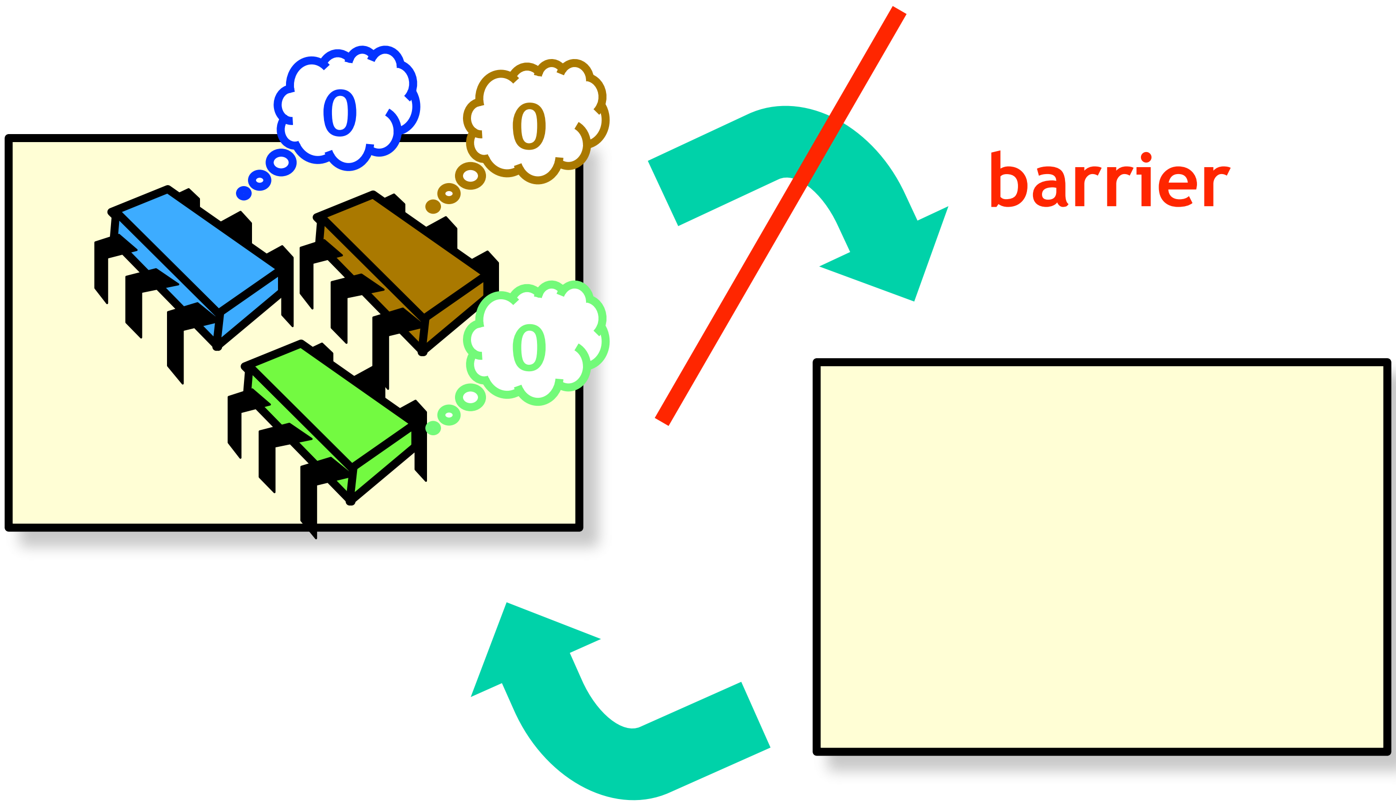
Real-Life Parallel Computation



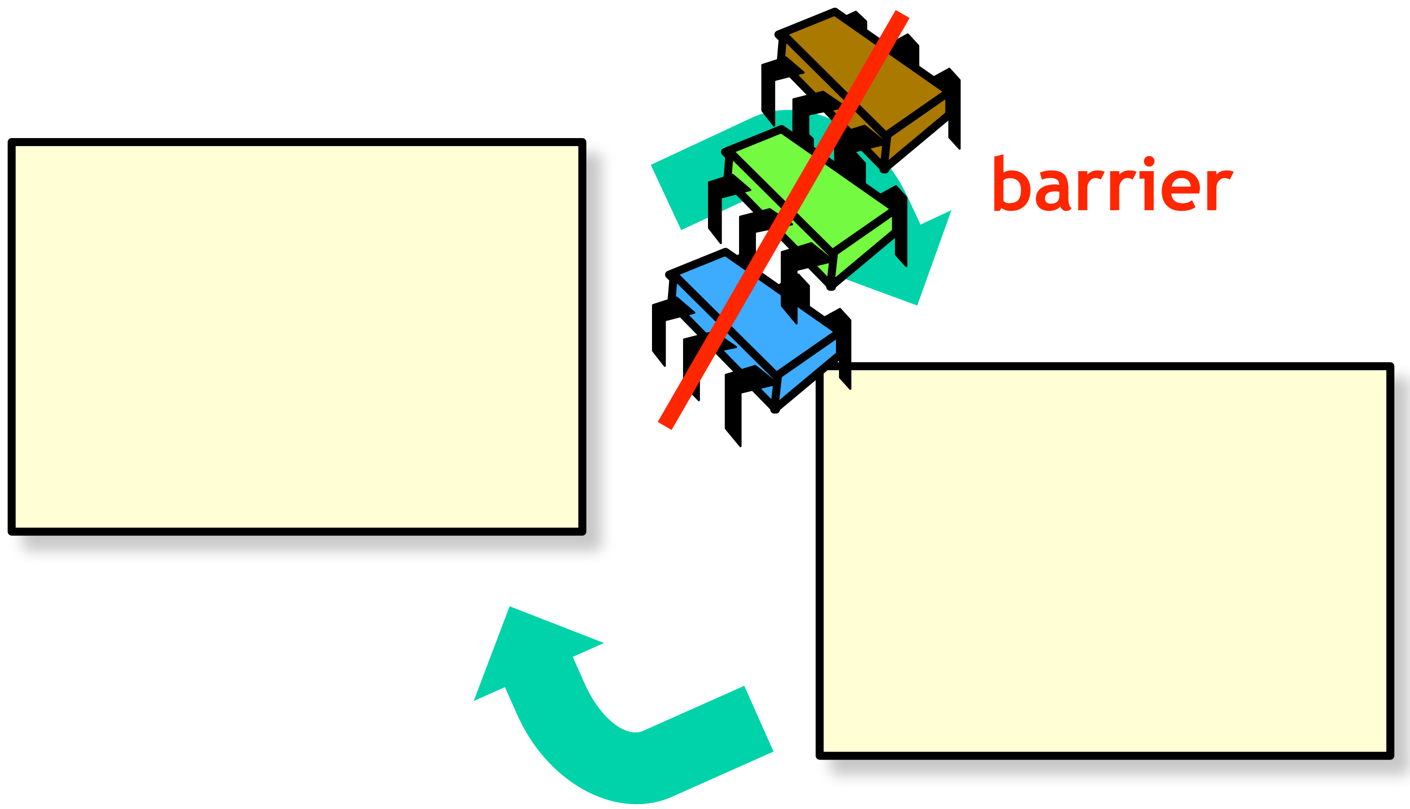
Real-Life Parallel Computation



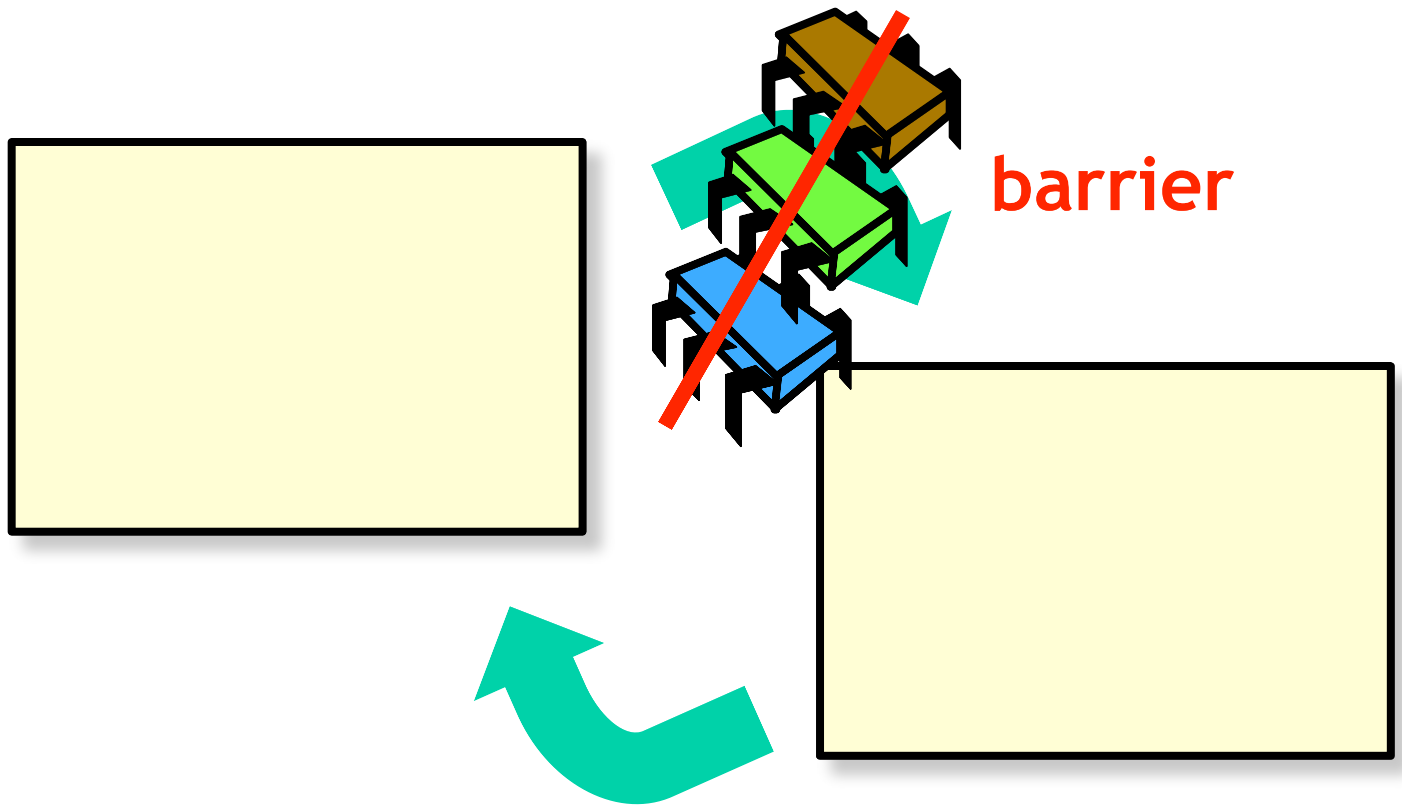
Barrier Synchronization



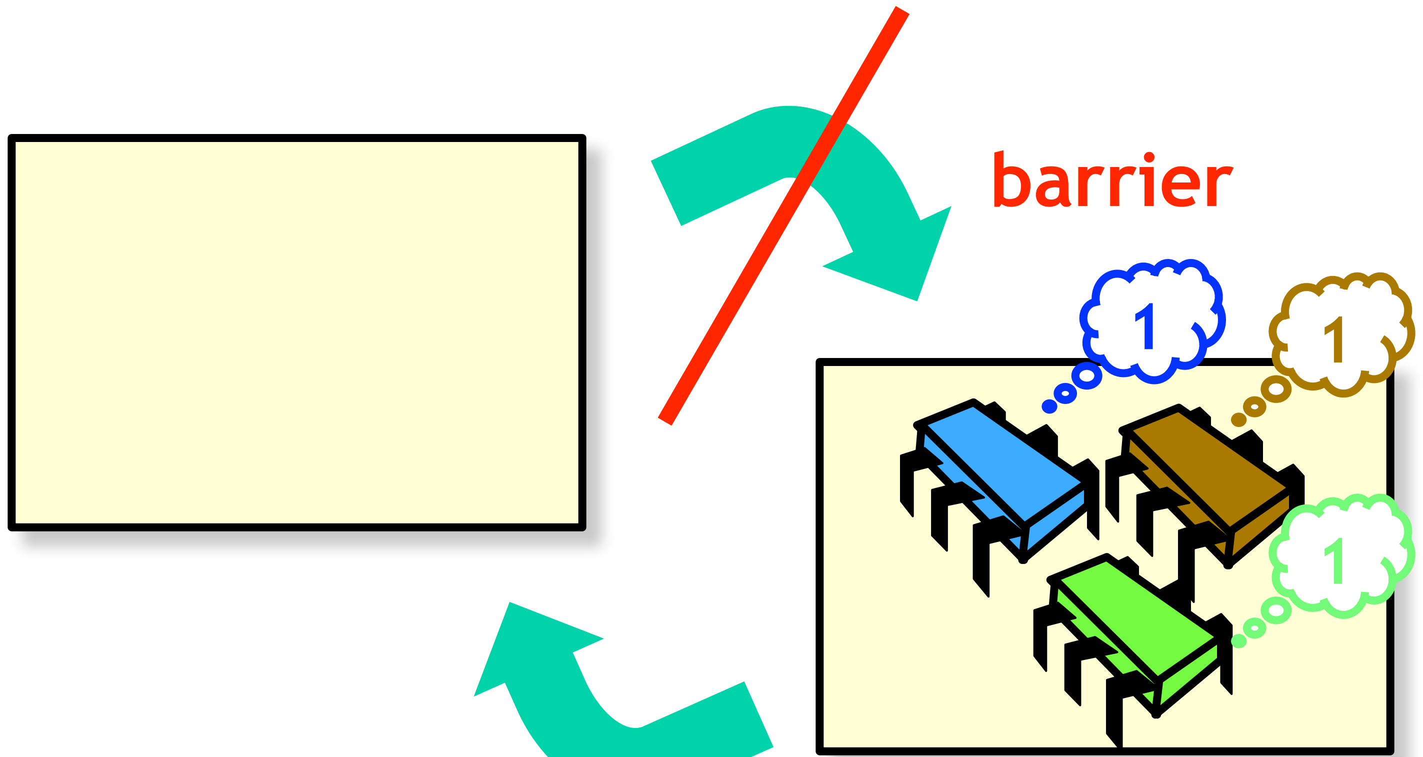
Barrier Synchronization



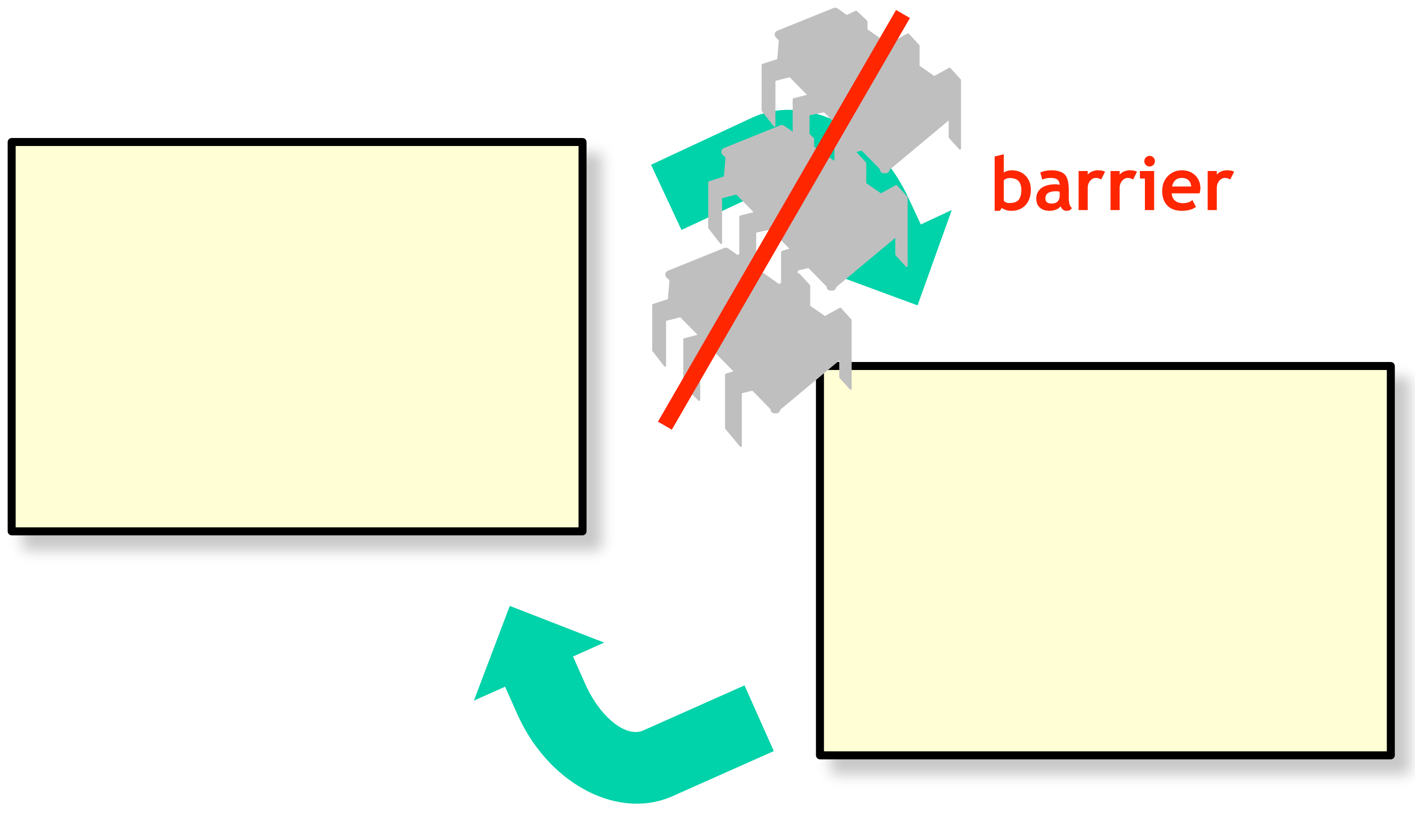
Barrier Synchronization



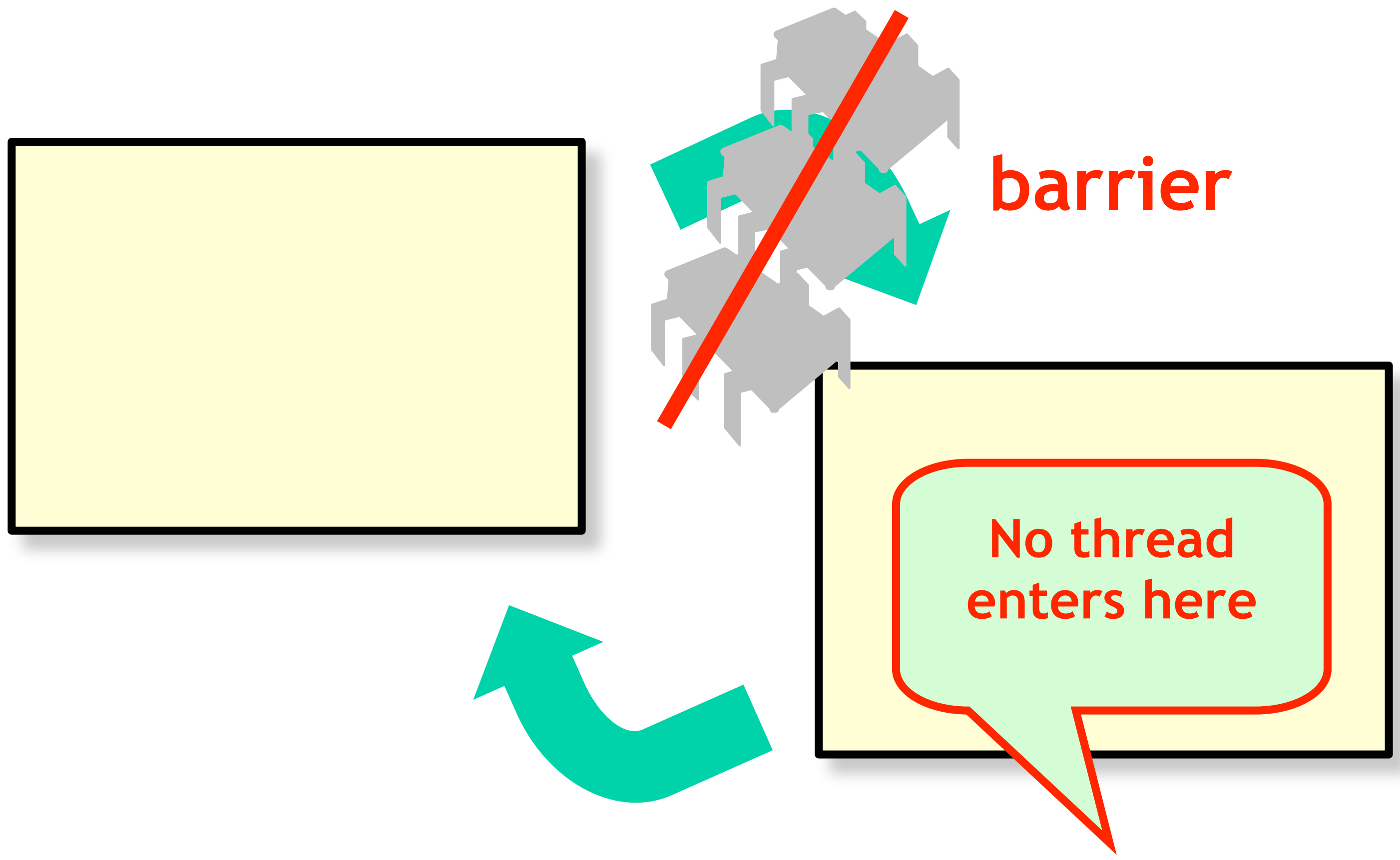
Barrier Synchronization



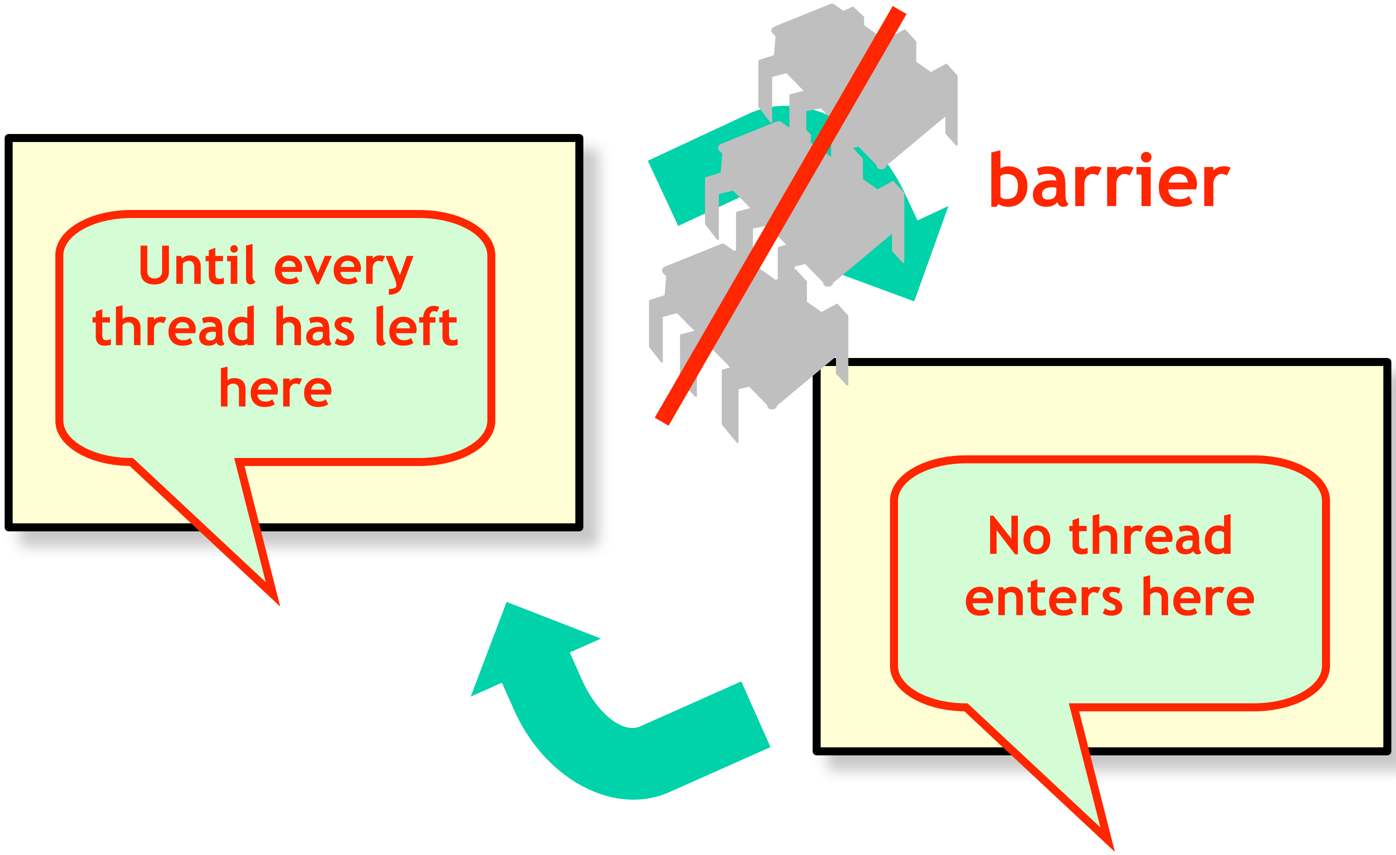
Barrier Synchronization



Barrier Synchronization



Barrier Synchronization



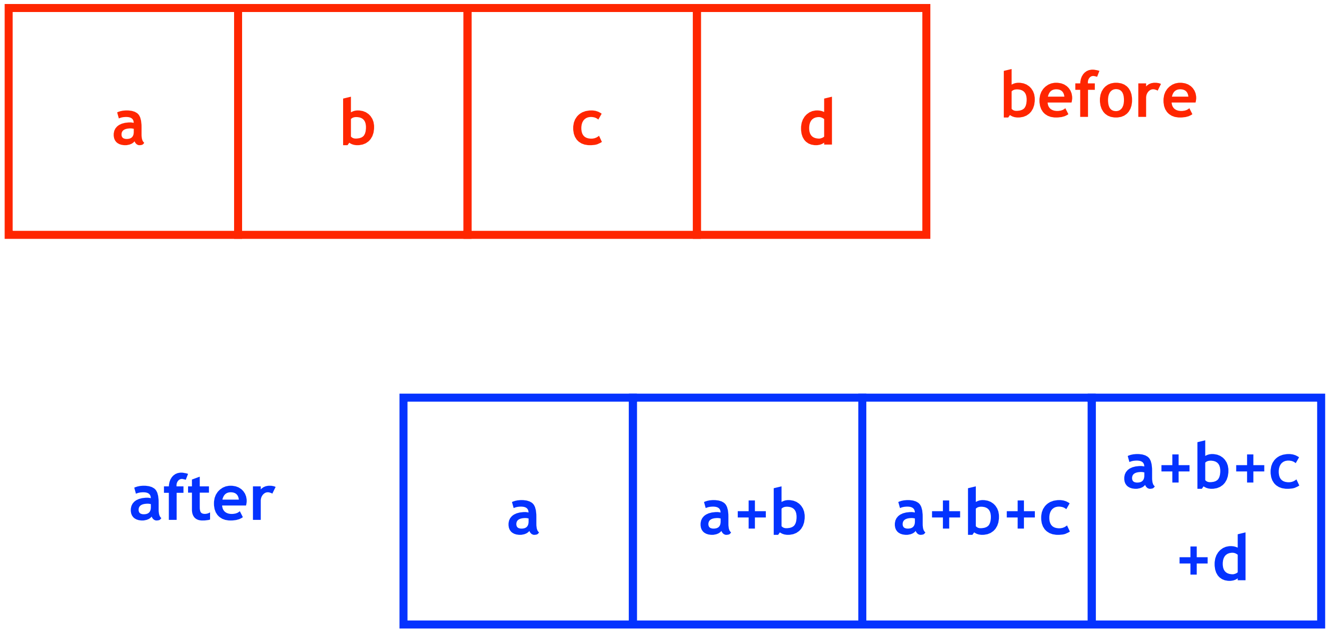
Why Do We Care?

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

Duality

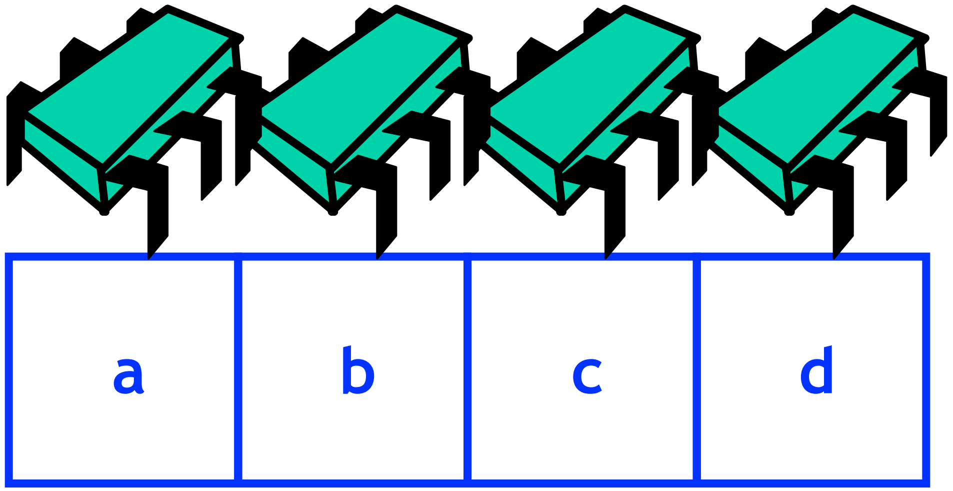
- Dual to mutual exclusion
  - Include others, not exclude them
- Same implementation issues
  - Interaction with caches …
    - Invalidation?
    - Local spinning?

Example: Parallel Prefix

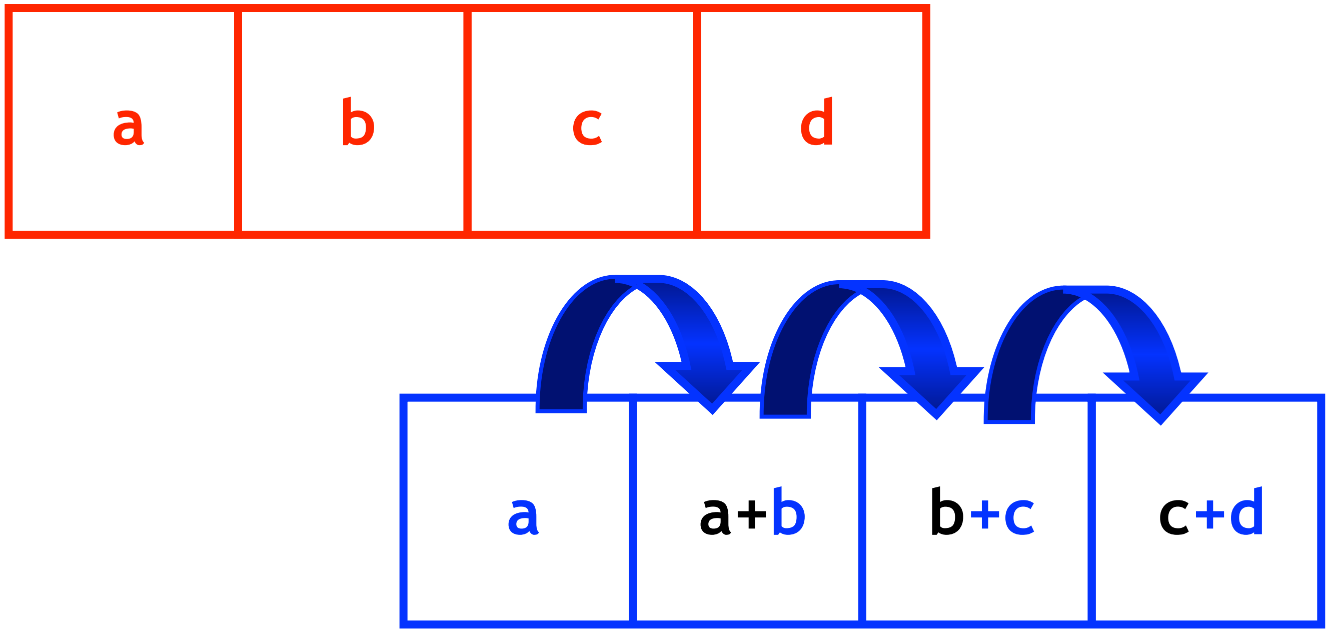


Parallel Prefix

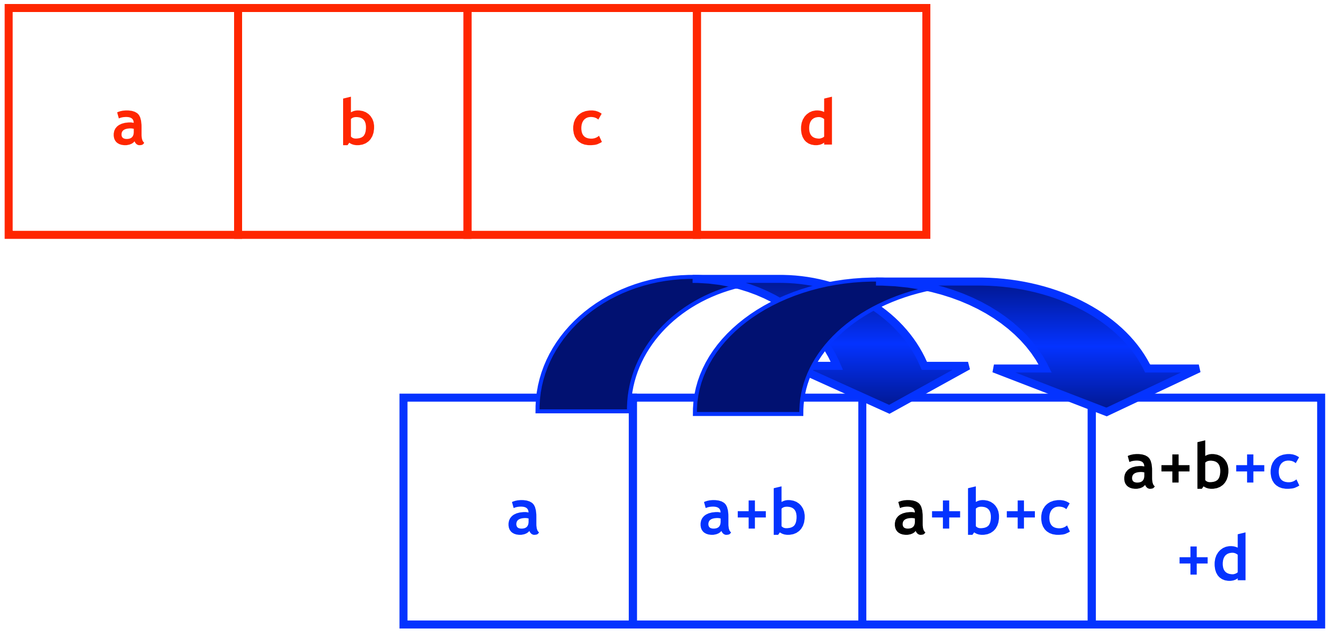
One thread
Per entry



Parallel Prefix: Phase 1



Parallel Prefix: Phase 2



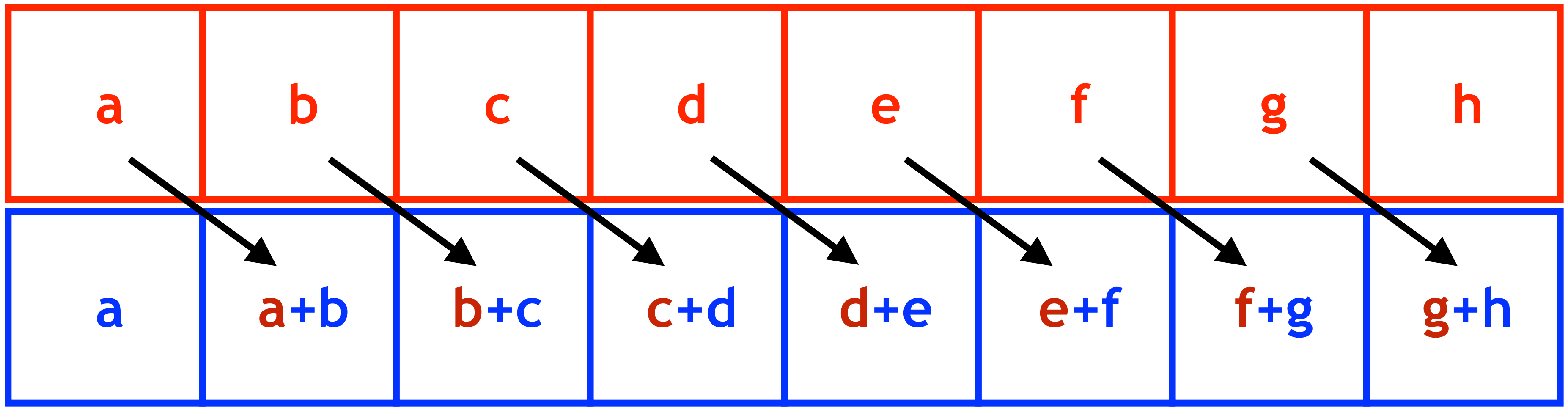
Parallel Prefix: Phase 2

a	b	c	d	e	f	g	h

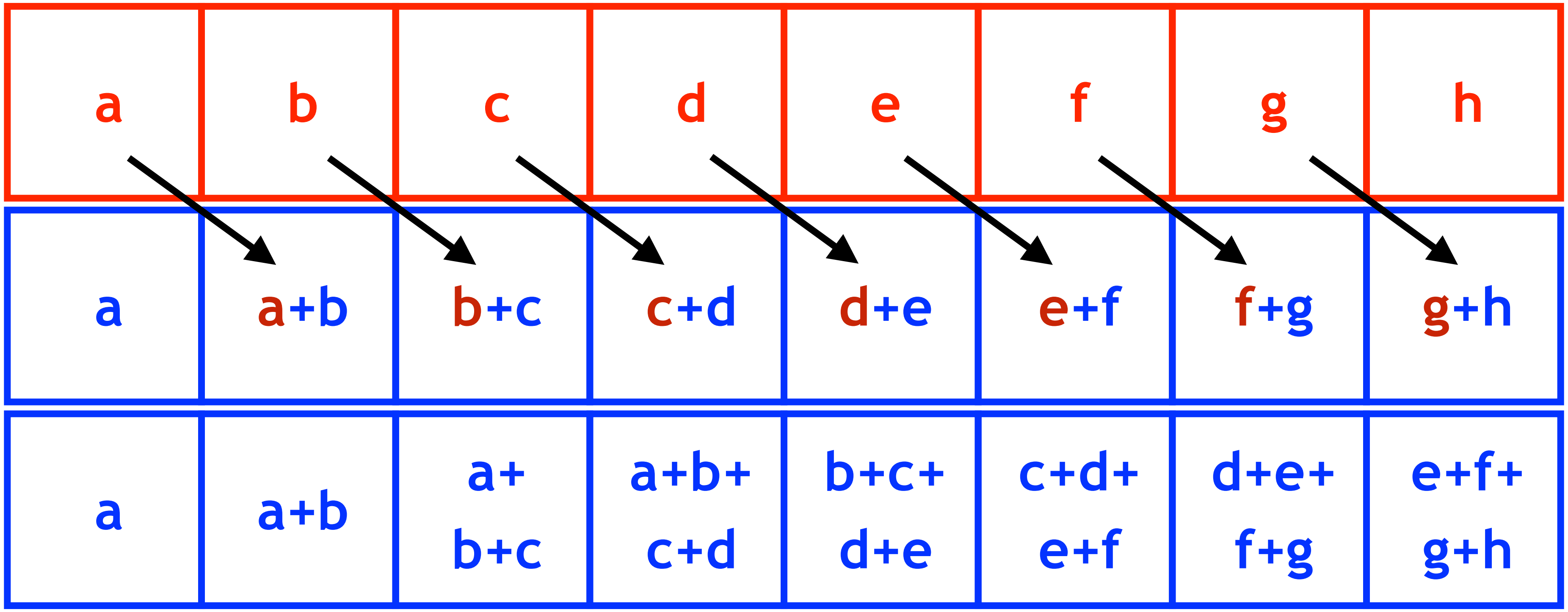
Parallel Prefix: Phase 2

a	b	c	d	e	f	g	h
a	a+b	b+c	c+d	d+e	e+f	f+g	g+h

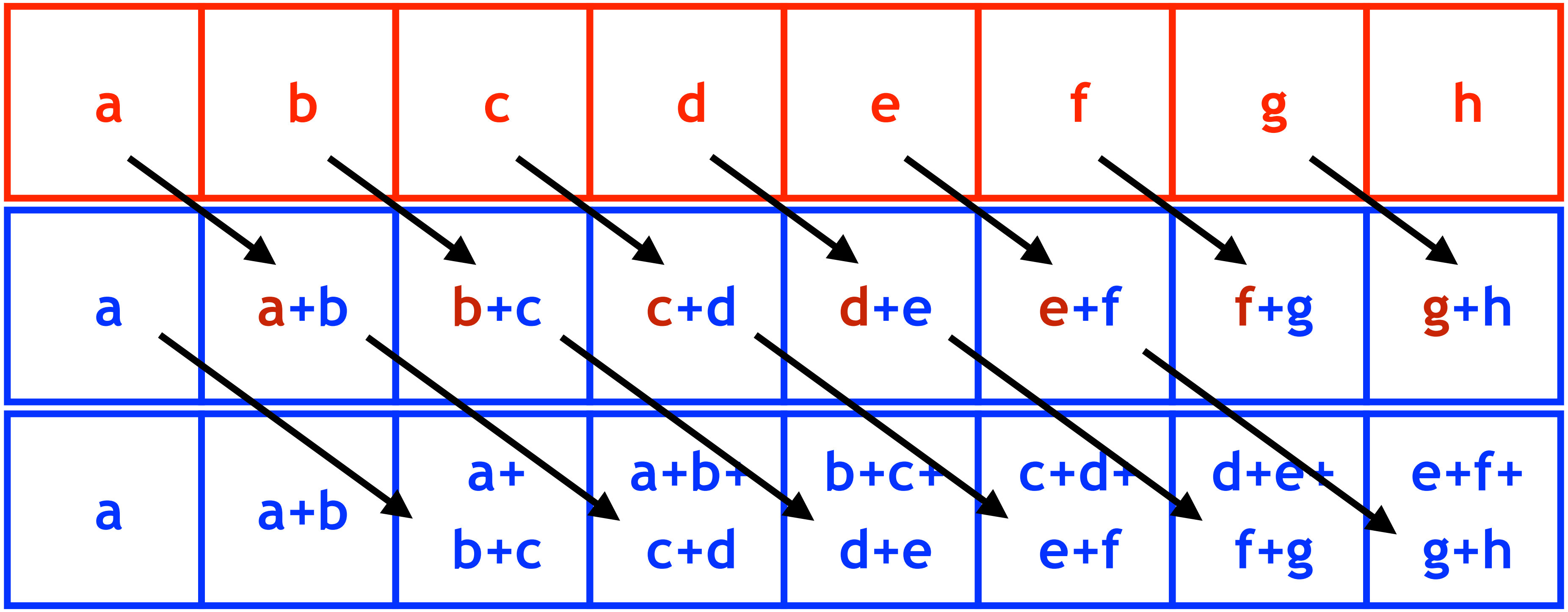
Parallel Prefix: Phase 2



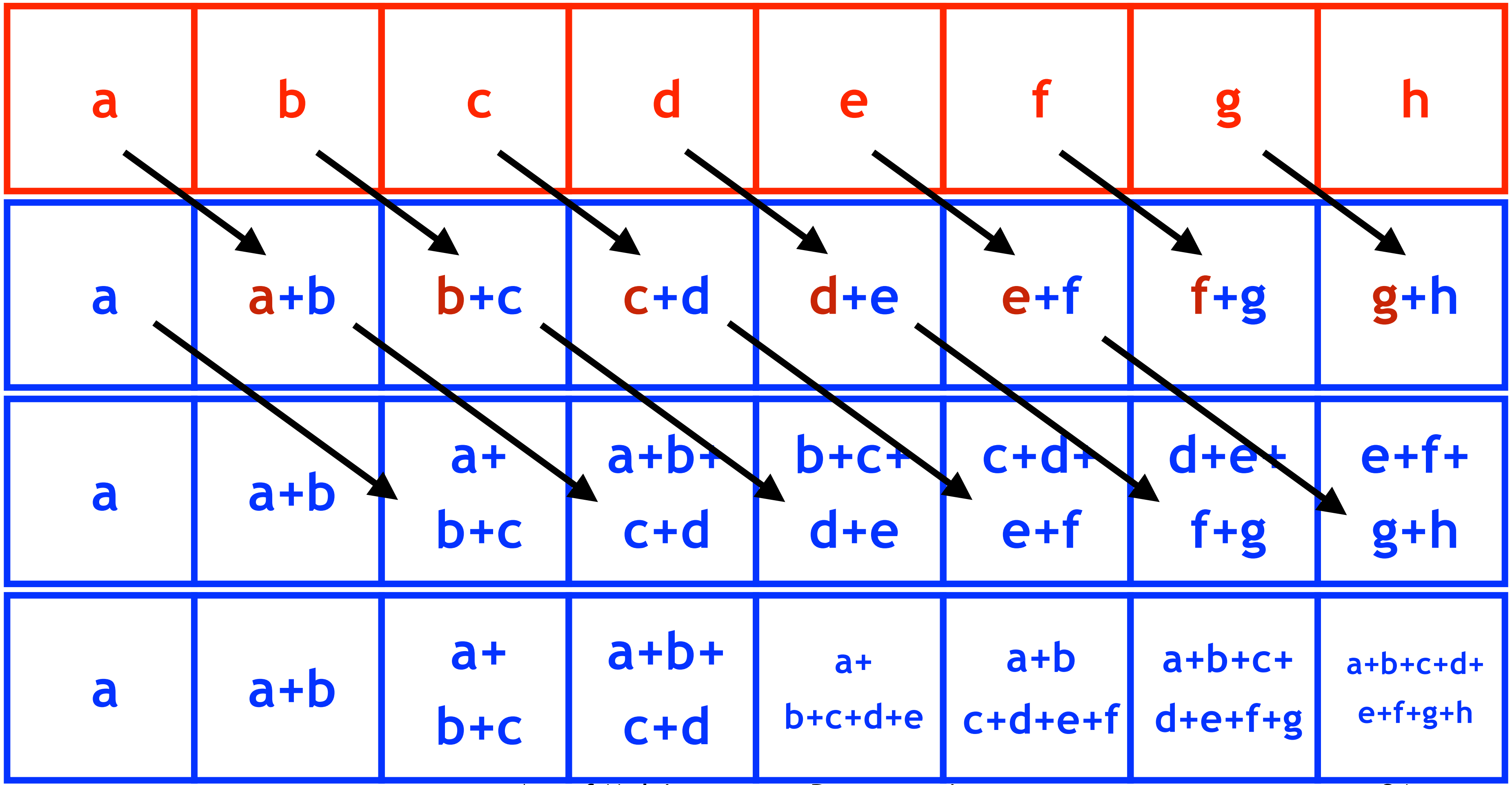
Parallel Prefix: Phase 2



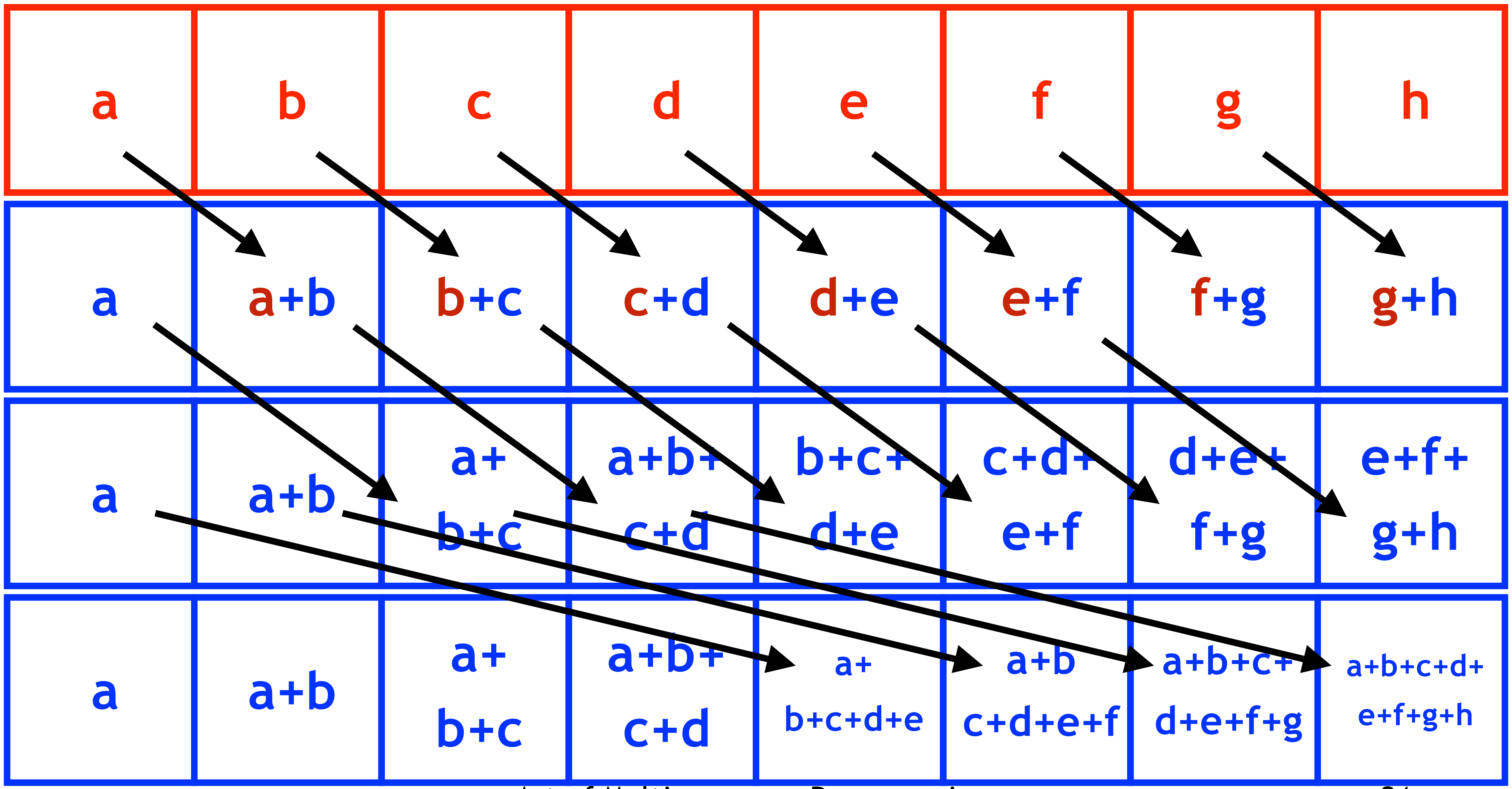
Parallel Prefix: Phase 2



Parallel Prefix: Phase 2



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) {
  this.a = a;
  this.b = b;
  this.i = i;
 }
```

Prefix

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

```
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

40

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()
}
```

Reuse

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

Do work

Reuse

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

Do work

synchronize

Reuse

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

Do work

synchronize

repeat

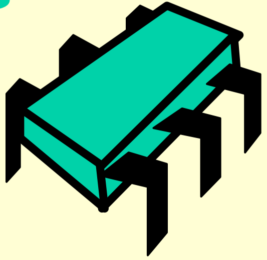
Barriers

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

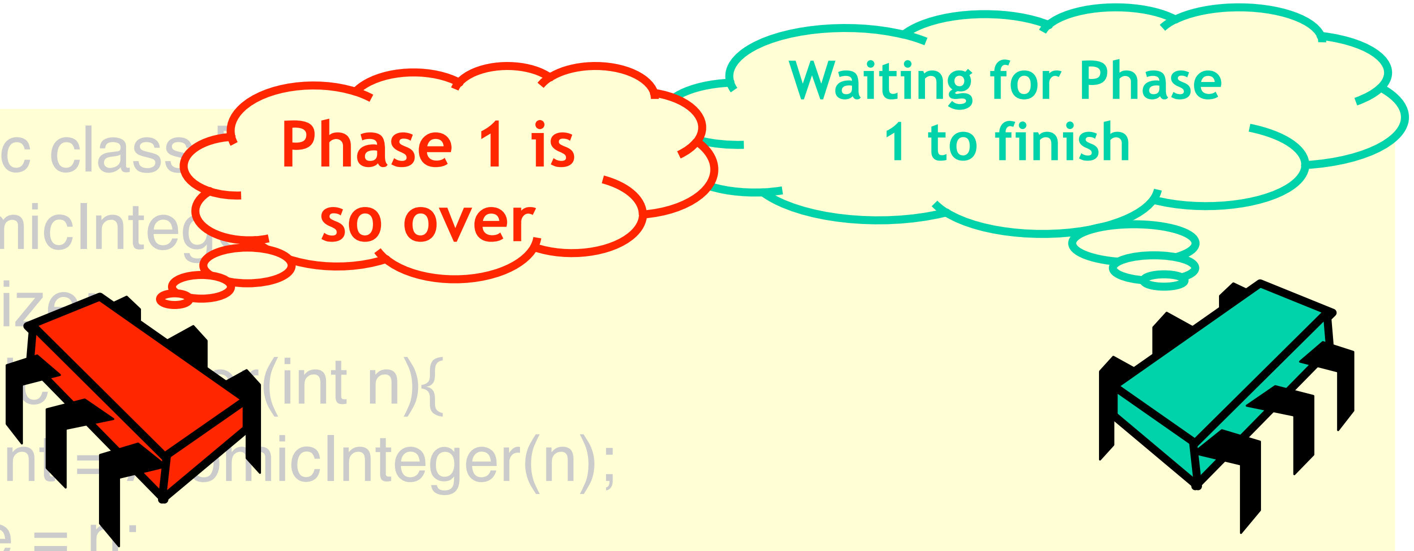
Barriers

```
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

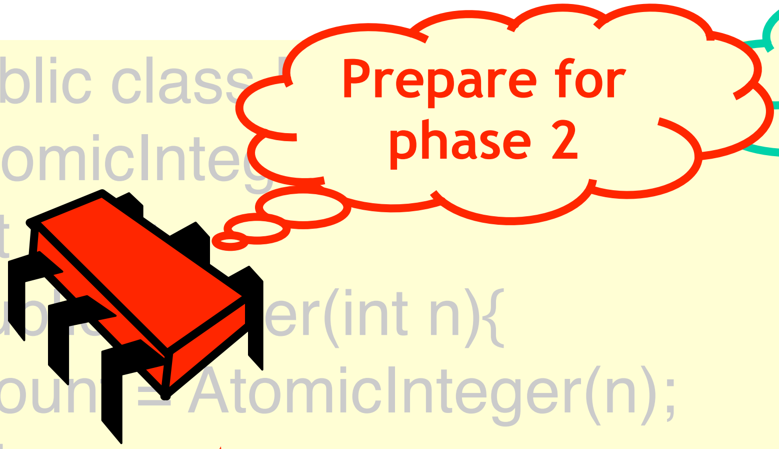


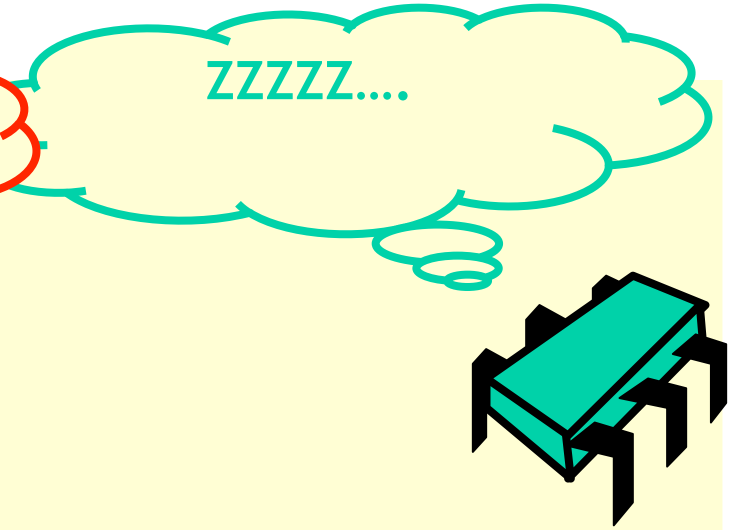
Barriers



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

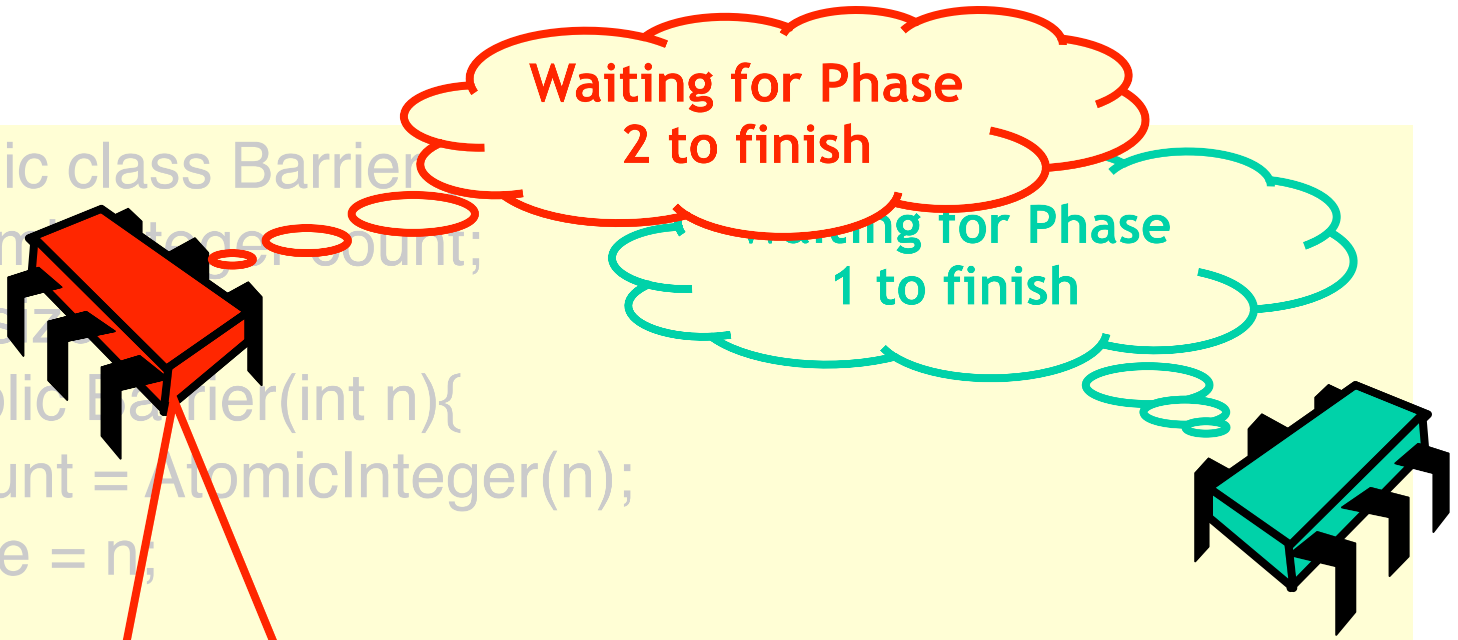
Barriers





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

Uh-Oh



```
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

Sense-Reversing Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 volatile 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;
 volatile 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;
 volatile 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

Sense-Reversing Barriers

```
public class Barrier {
 AtomicInteger count;
 int size;
 volatile 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;
 volatile 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;
 volatile 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

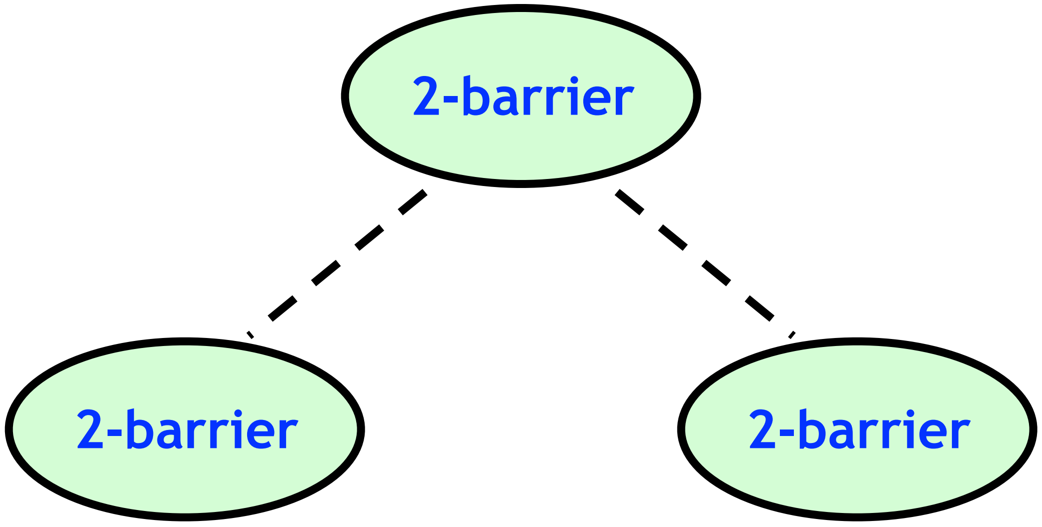
```
public class Barrier {
 AtomicInteger count;
 int size;
 volatile boolean sense = false;
 threadSense = new ThreadLocal<boolean>…

 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

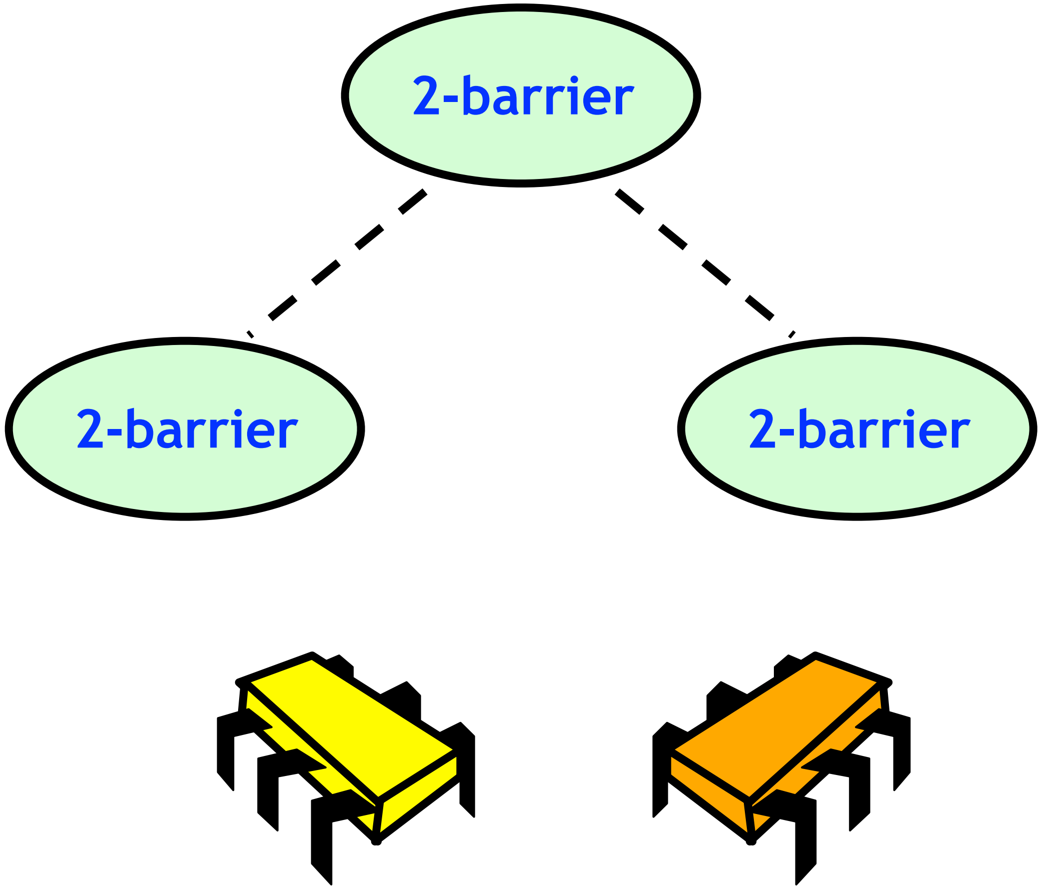
Combining Tree Barriers

- Reduce contention
- Split large barrier into tree of small barriers
- Requests go up the tree and notifications down
- Adds latency

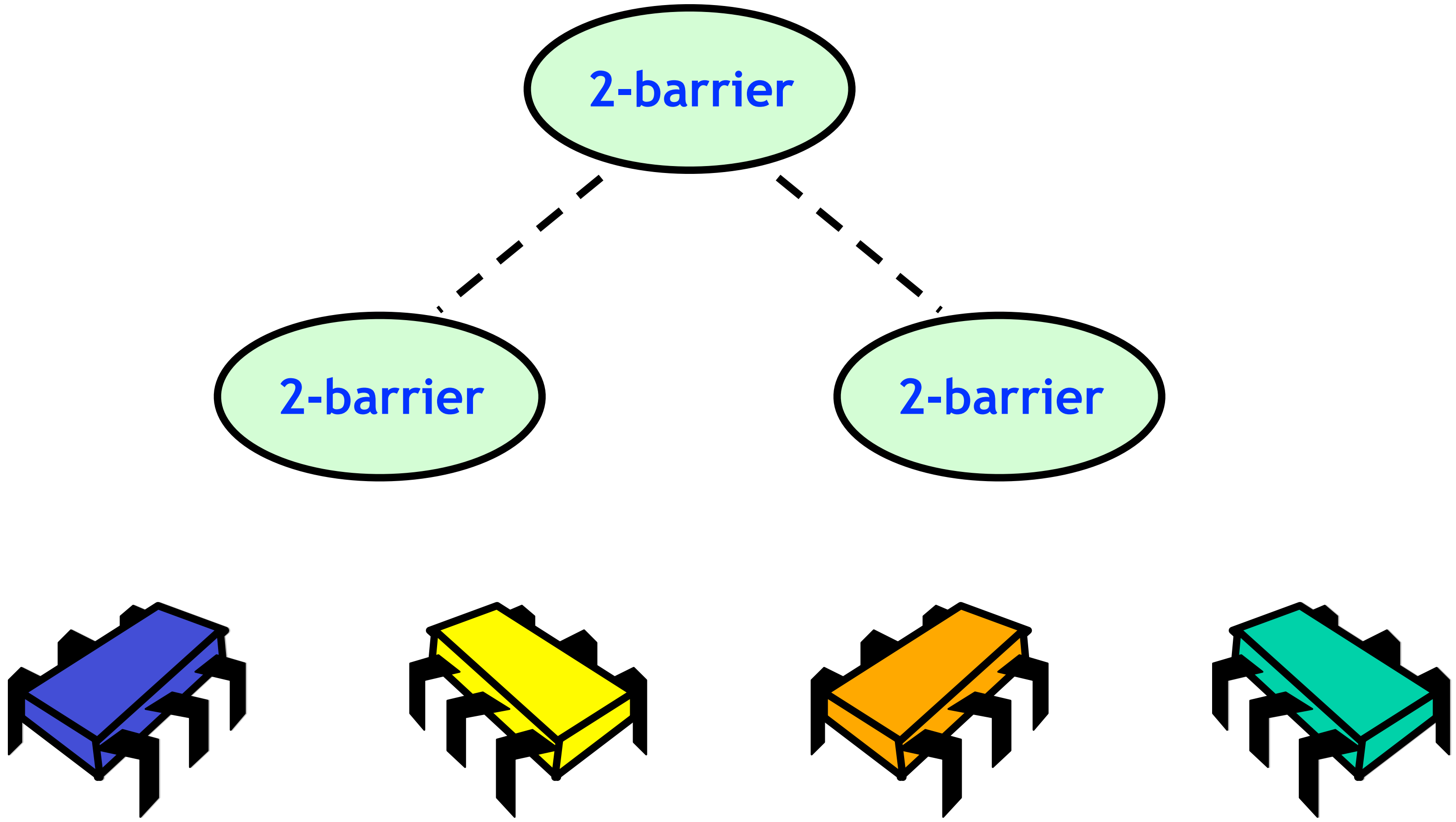
Combining Tree Barriers



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

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

```
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 NUMA

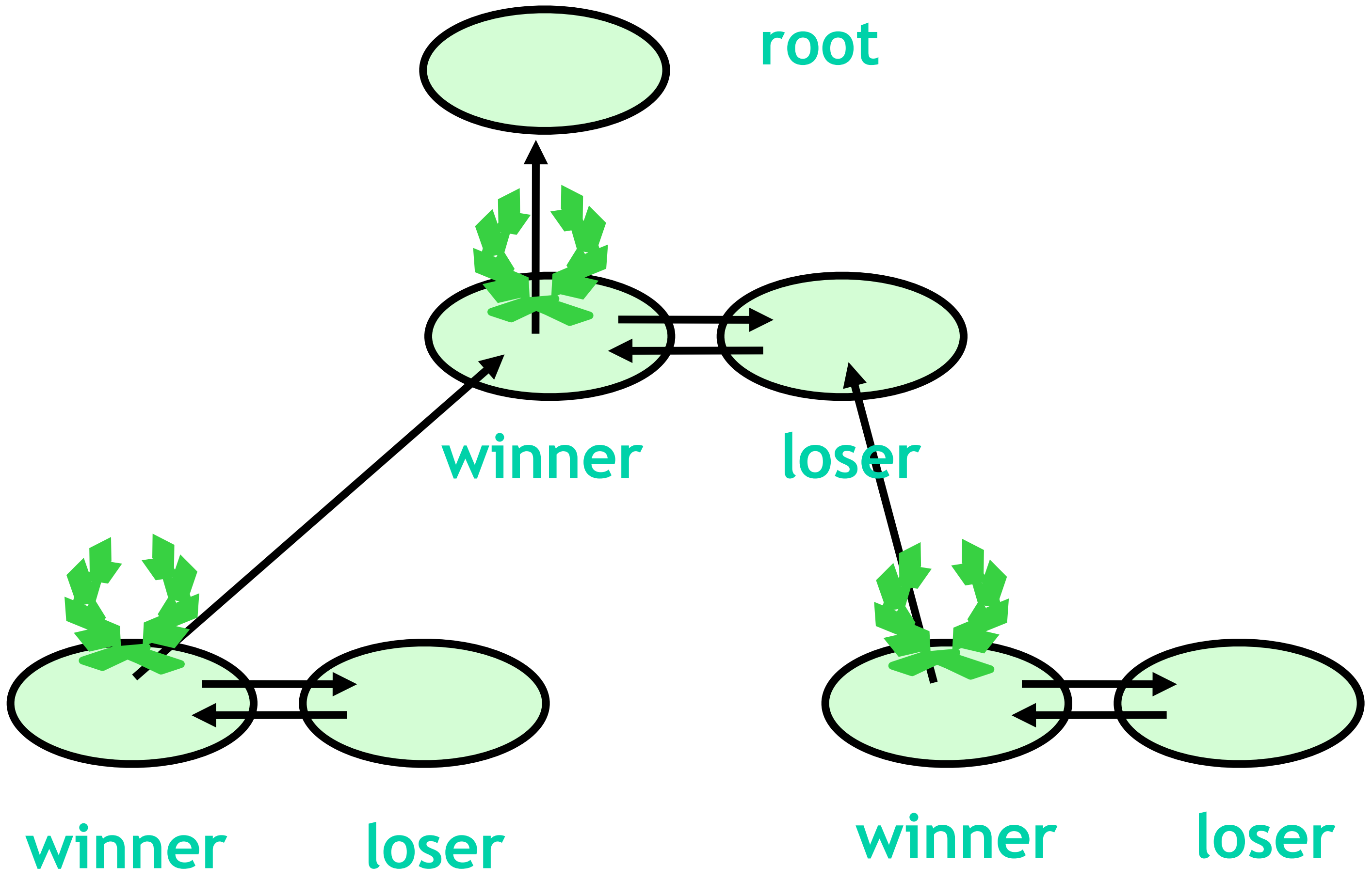
Remarks

- Everyone spins on sense field
  - Local spinning on bus-based (good)
  - Network hot-spot on distributed architecture (bad)
- Not really scalable

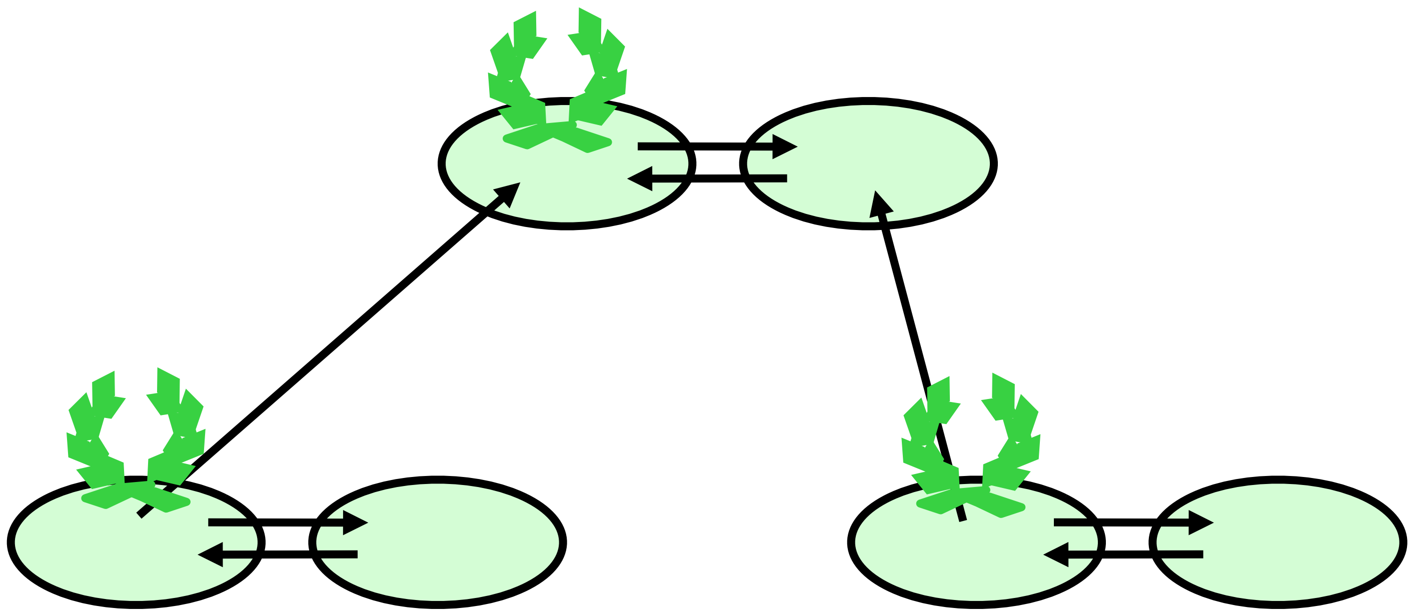
Tournament Tree Barrier

- 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

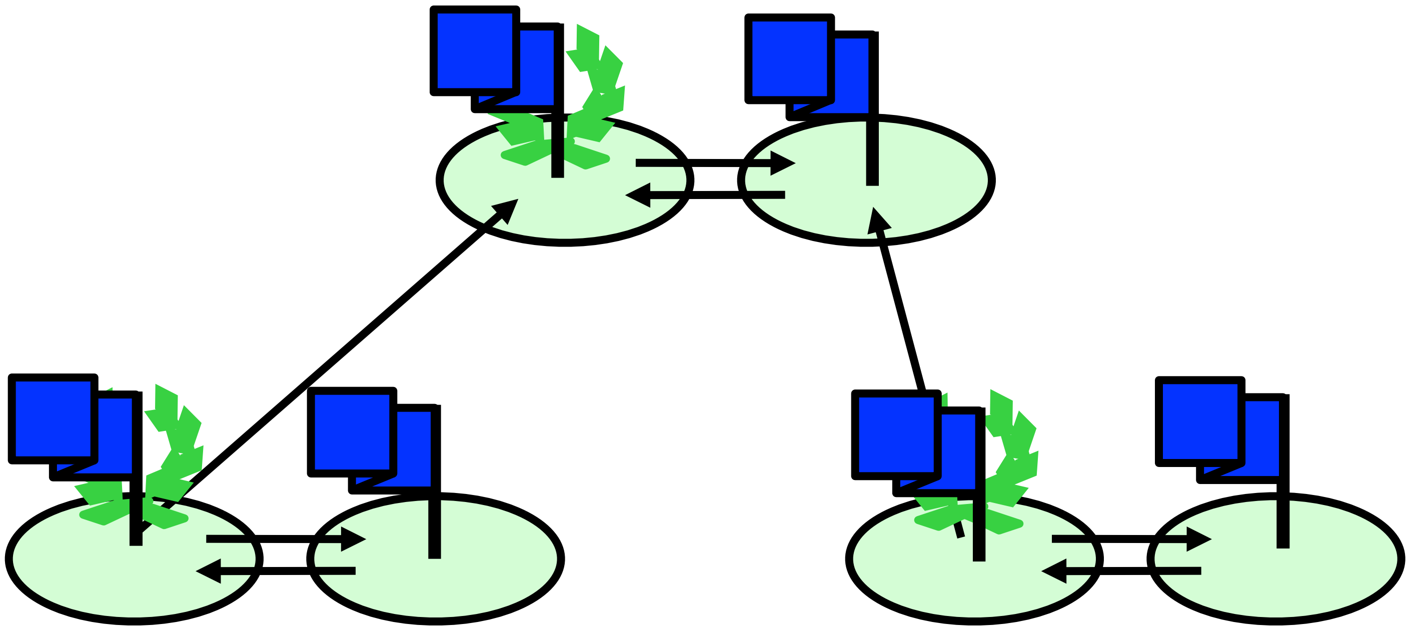
Tournament Tree Barriers



Tournament Tree Barriers

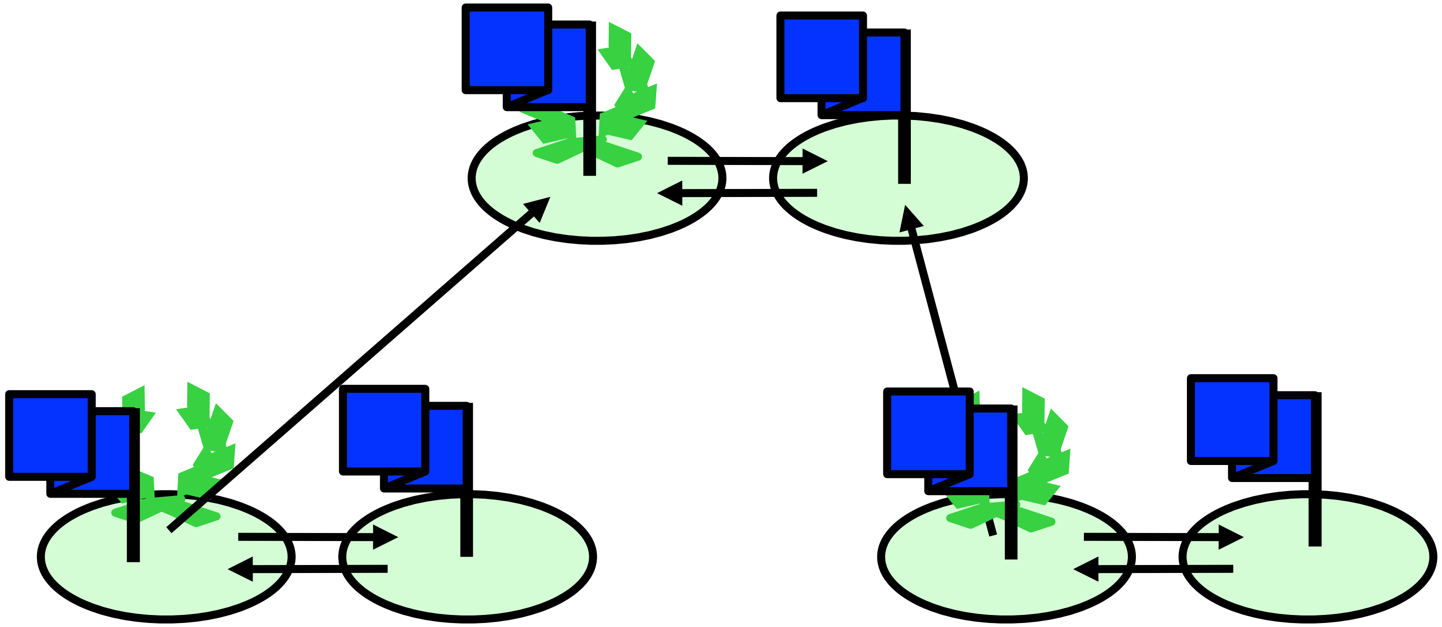


Tournament Tree Barriers

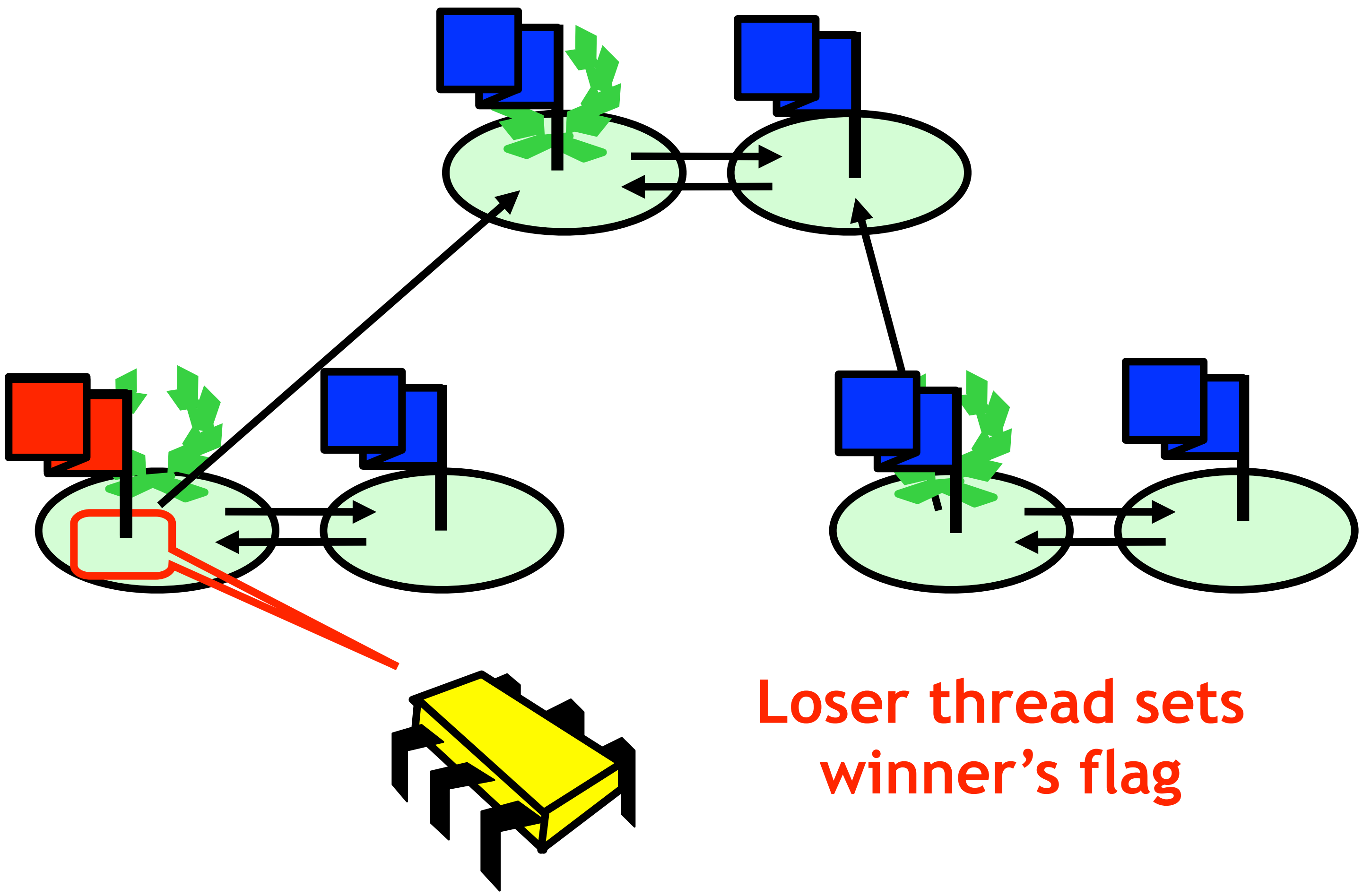


All flags blue

Tournament Tree Barriers

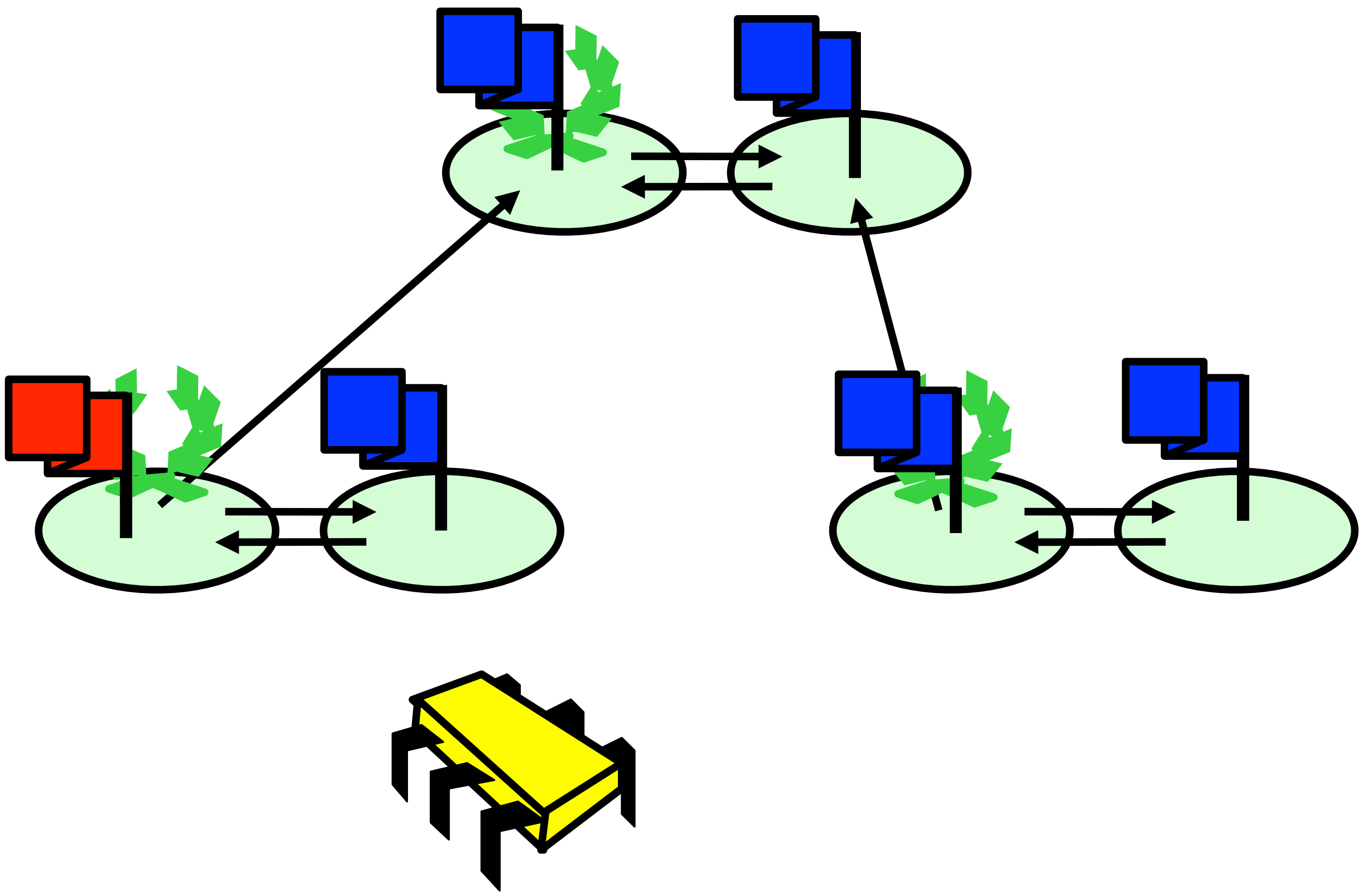


Tournament Tree Barriers

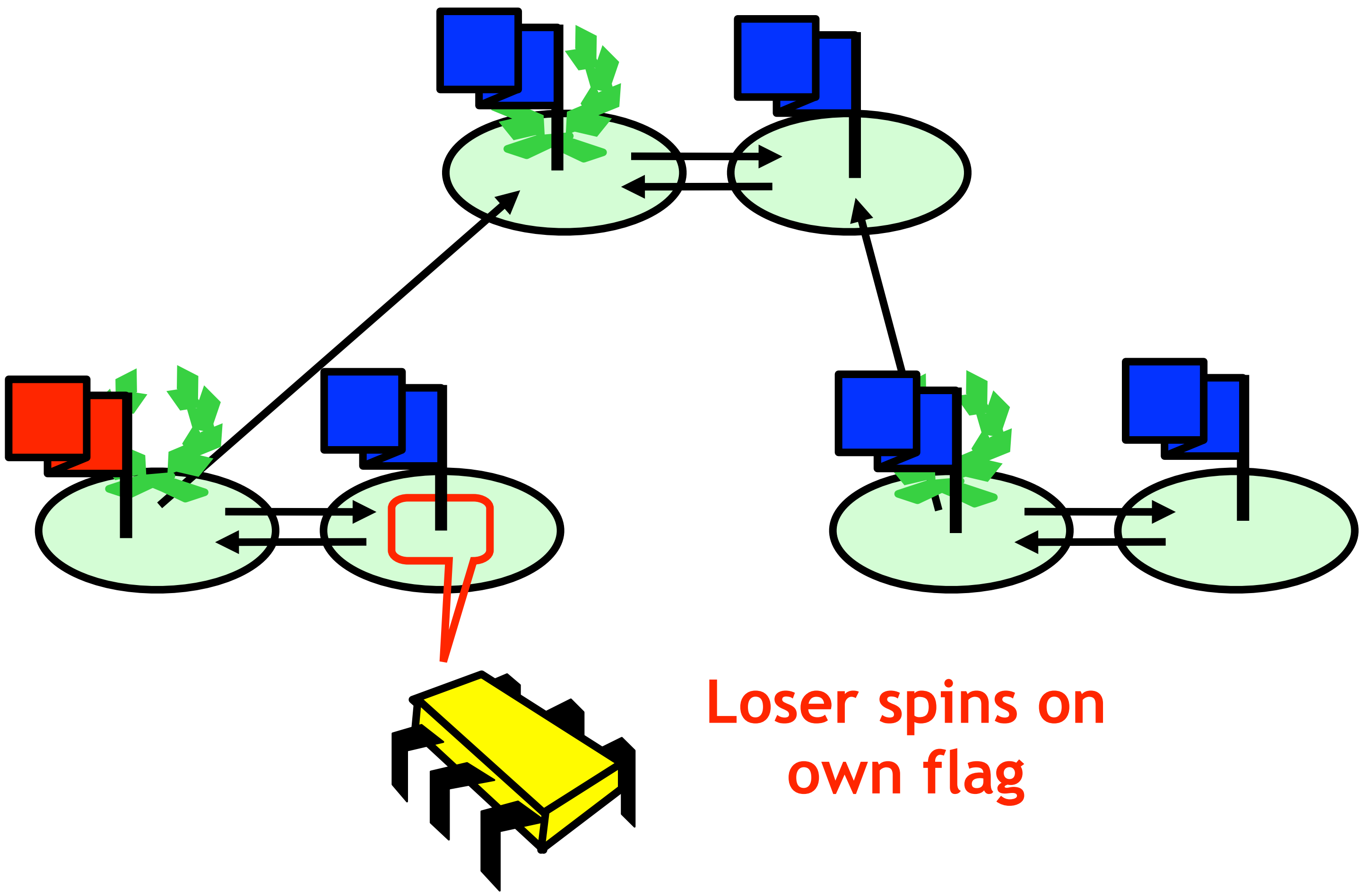


Loser thread sets winner's flag

Tournament Tree Barriers

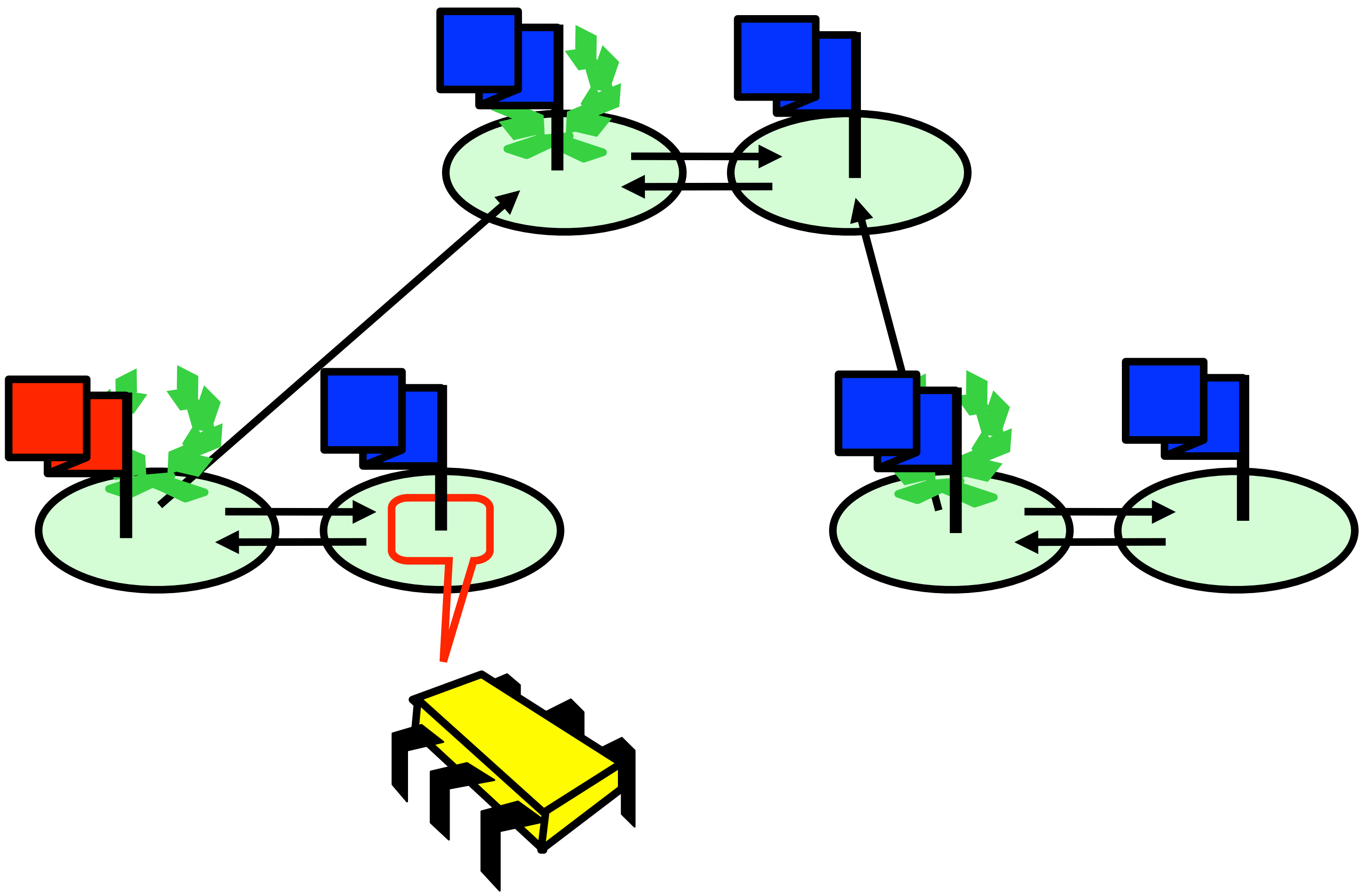


Tournament Tree Barriers

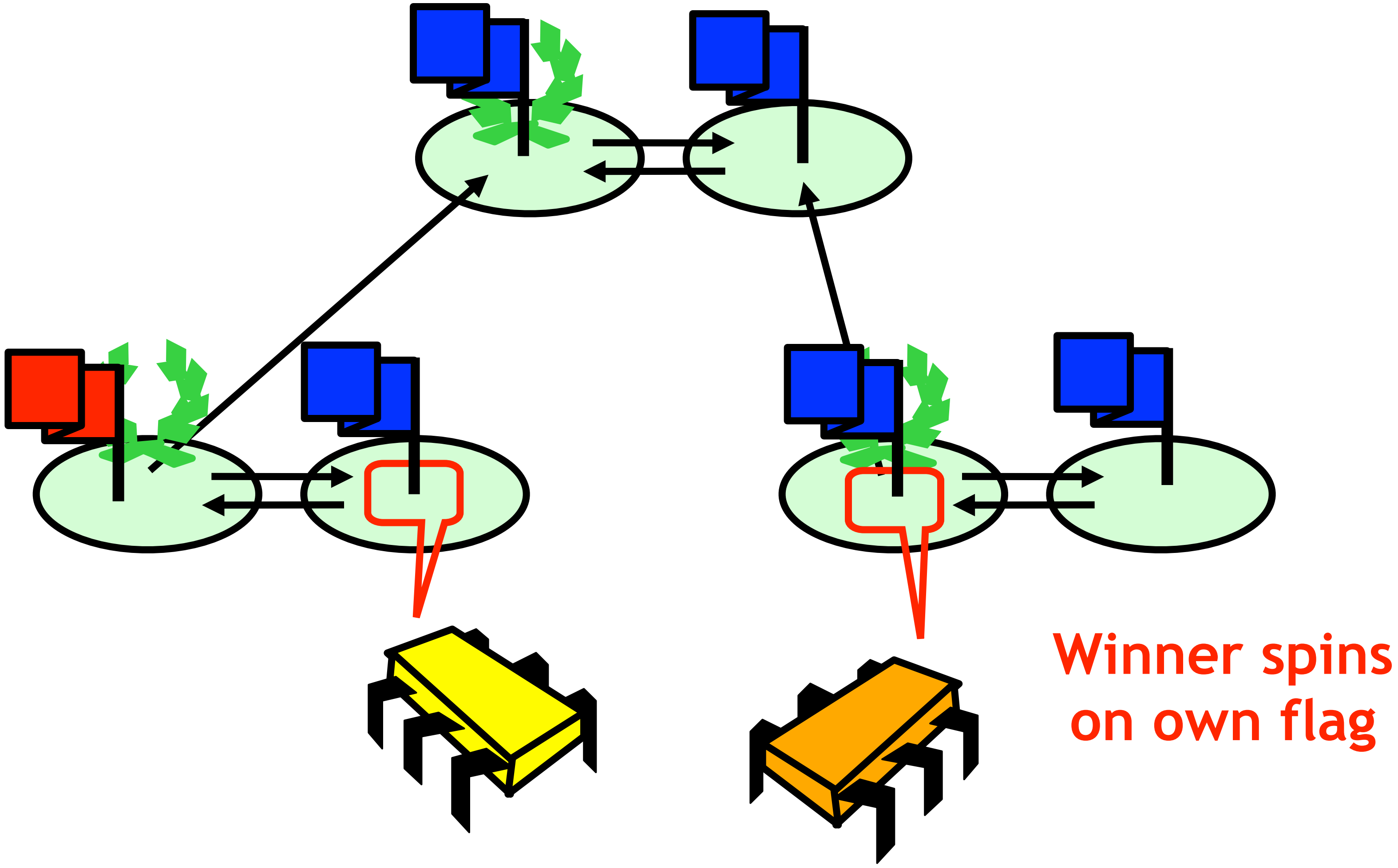


Loser spins on own flag

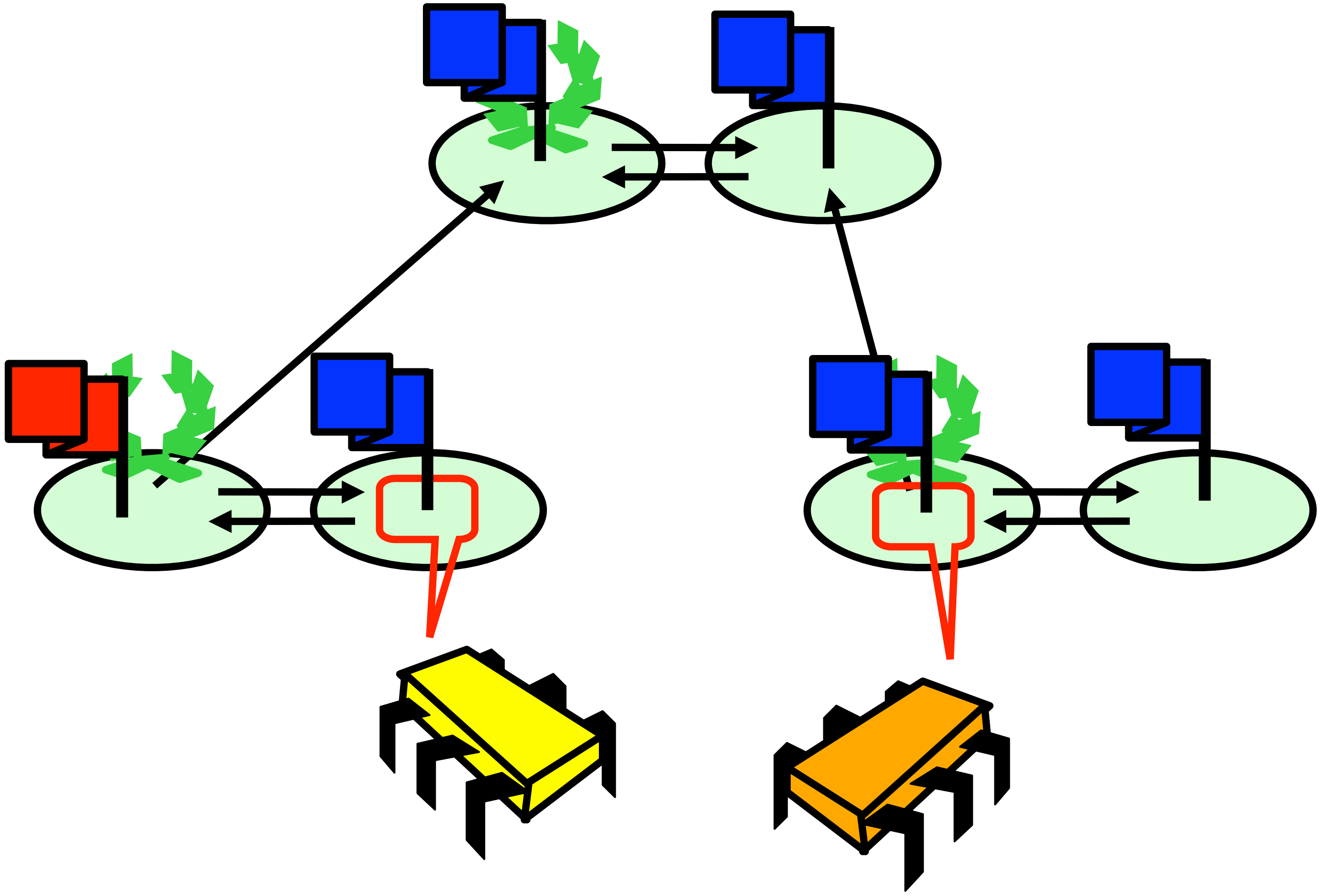
Tournament Tree Barriers



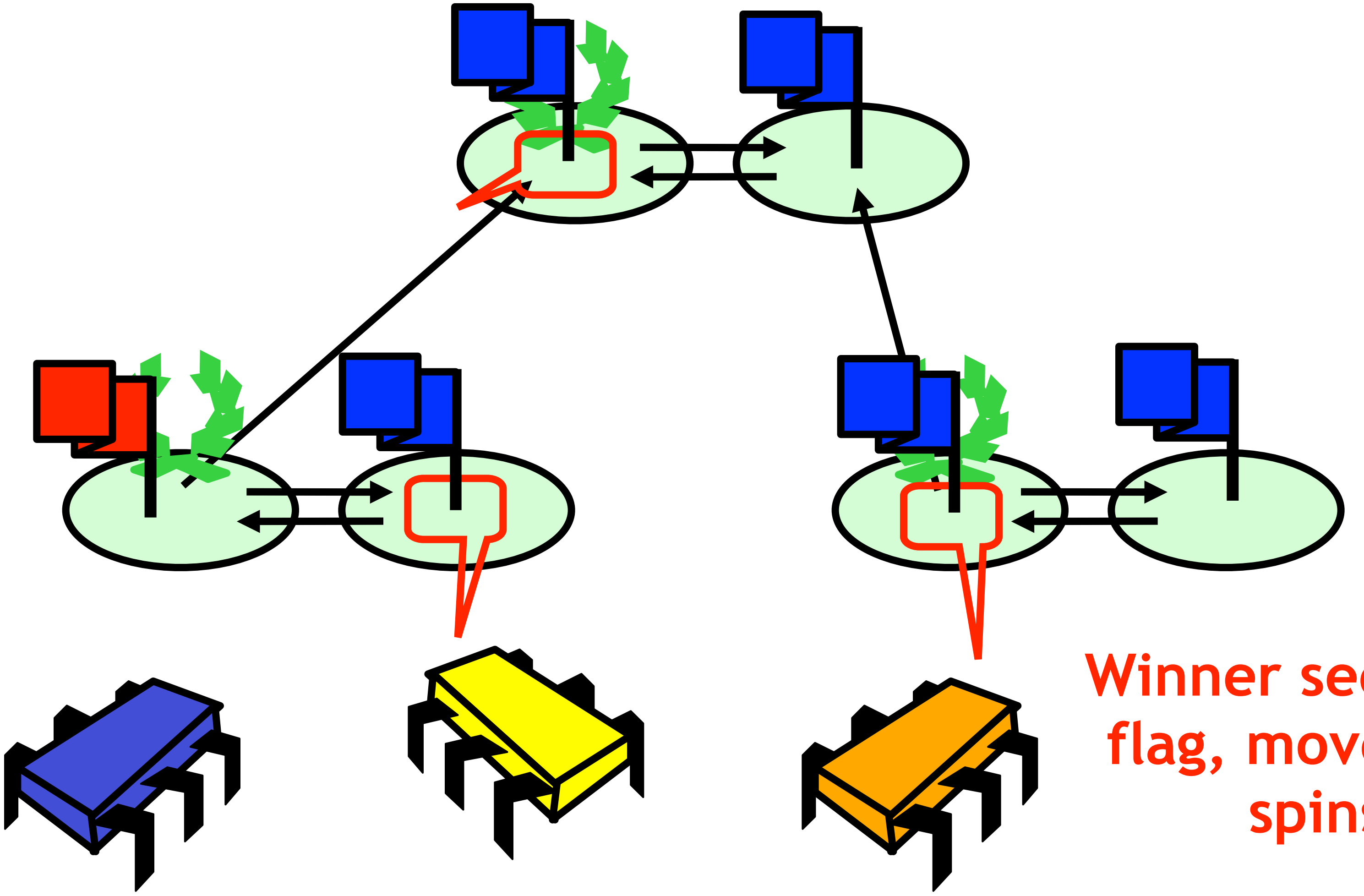
Tournament Tree Barriers



Tournament Tree Barriers

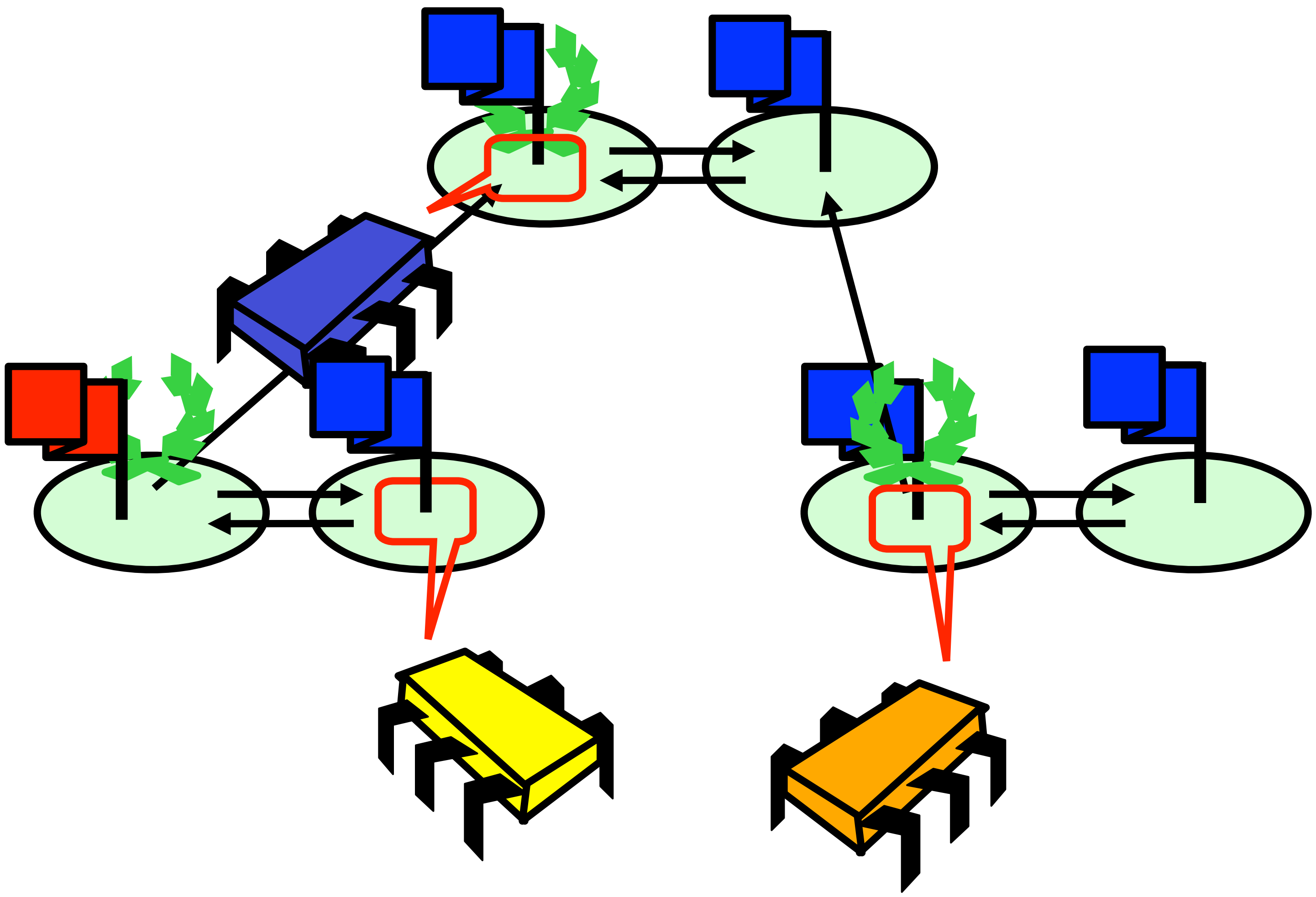


Tournament Tree Barriers

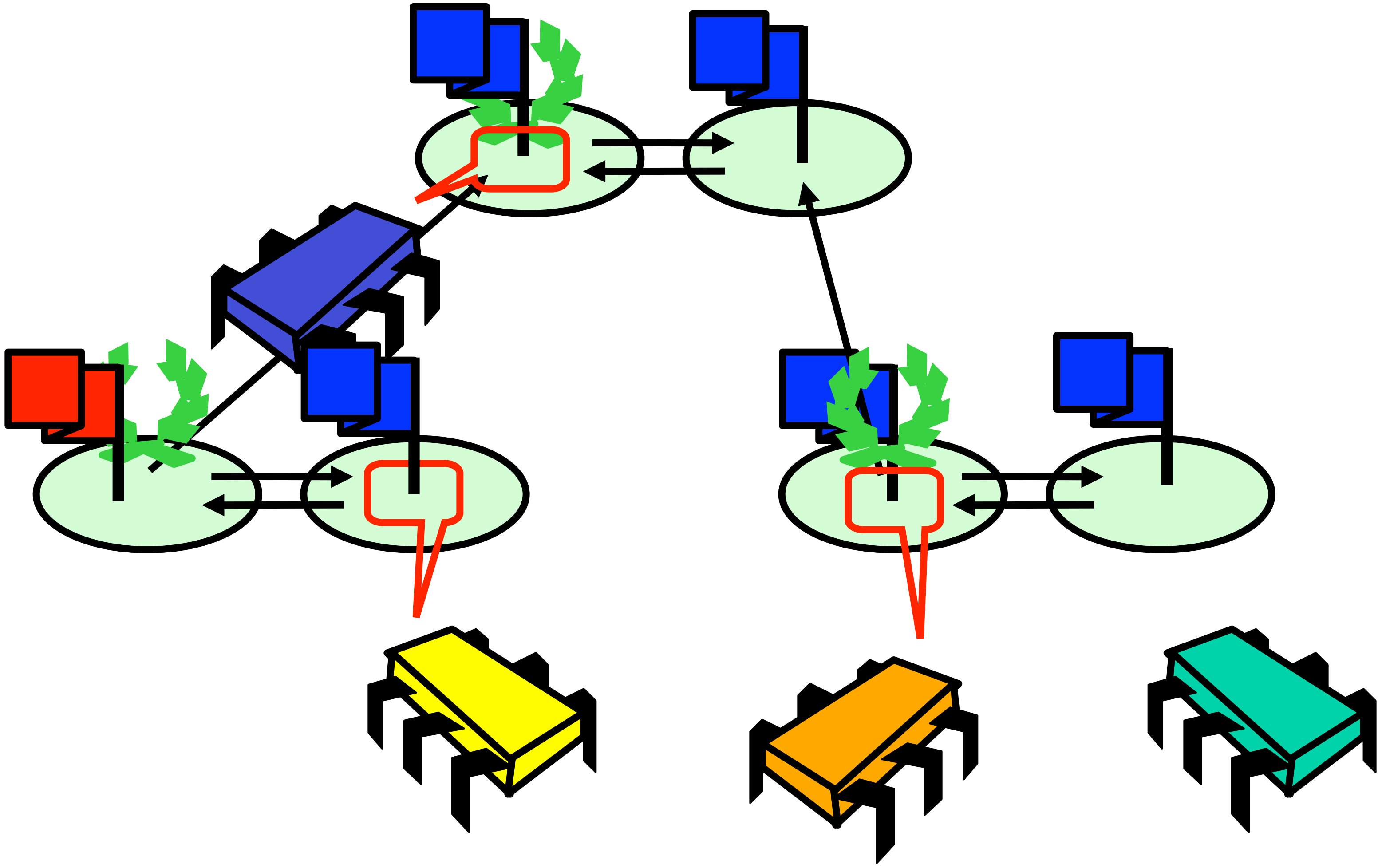


Winner sees own flag, moves up, spins

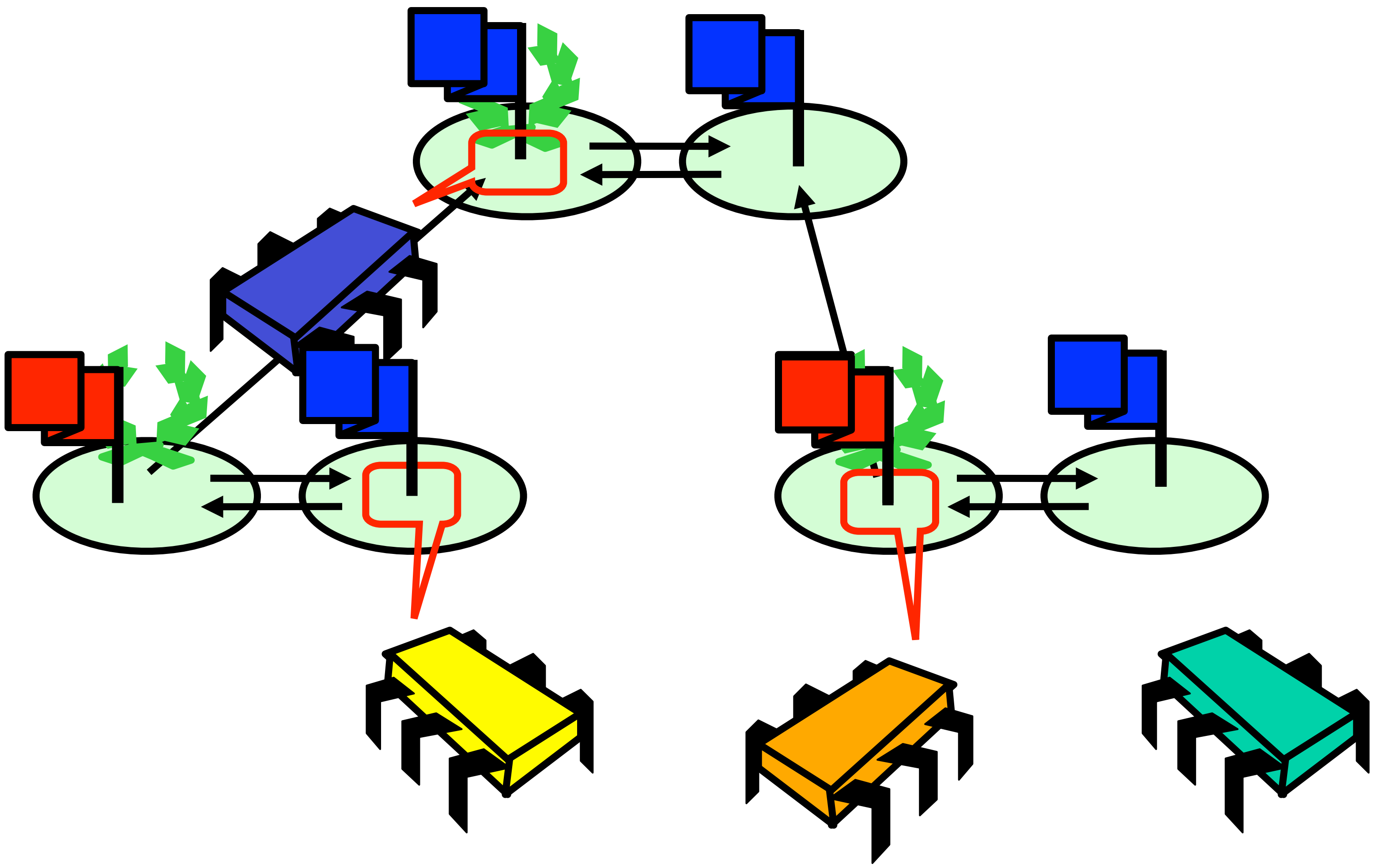
Tournament Tree Barriers



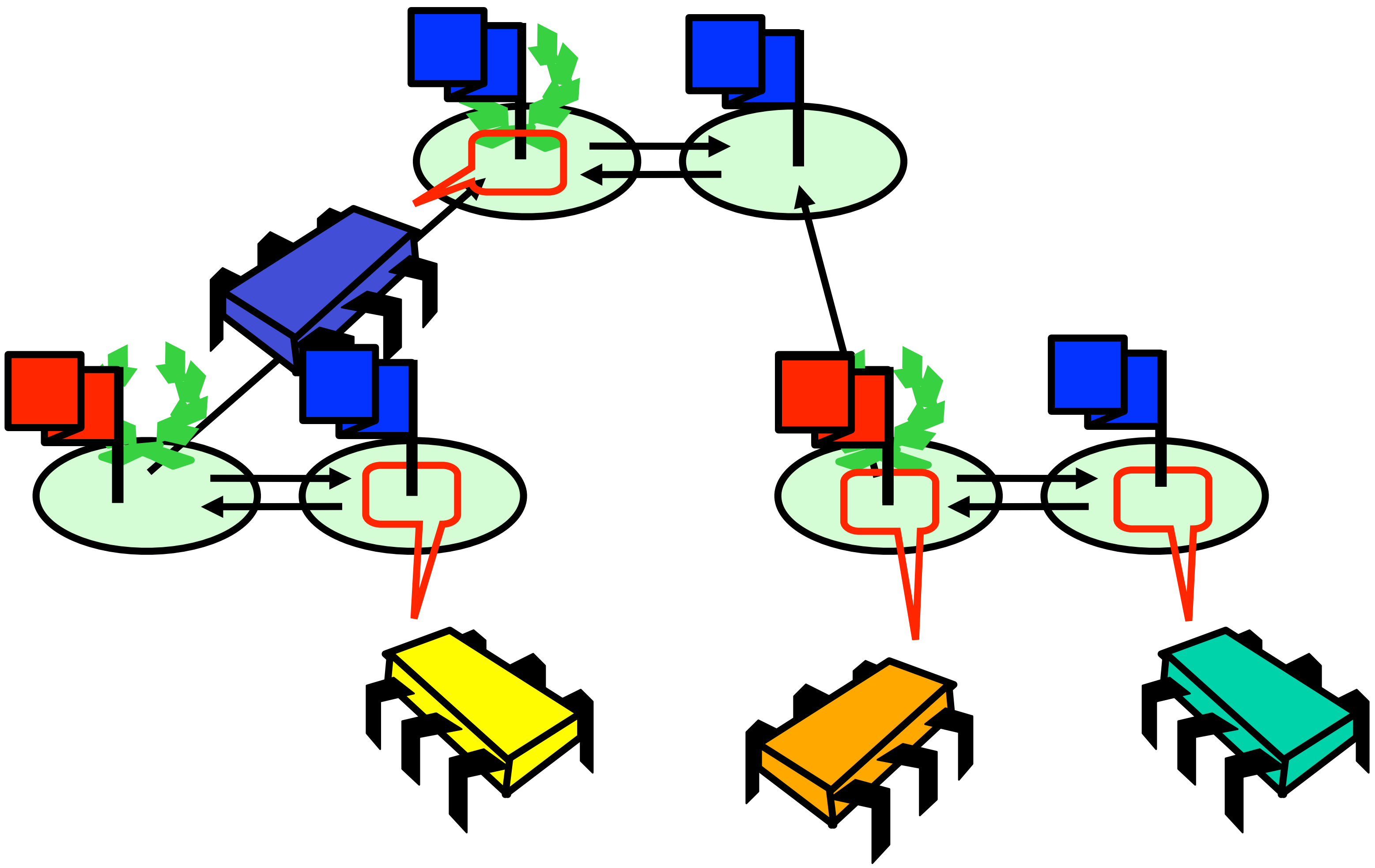
Tournament Tree Barriers



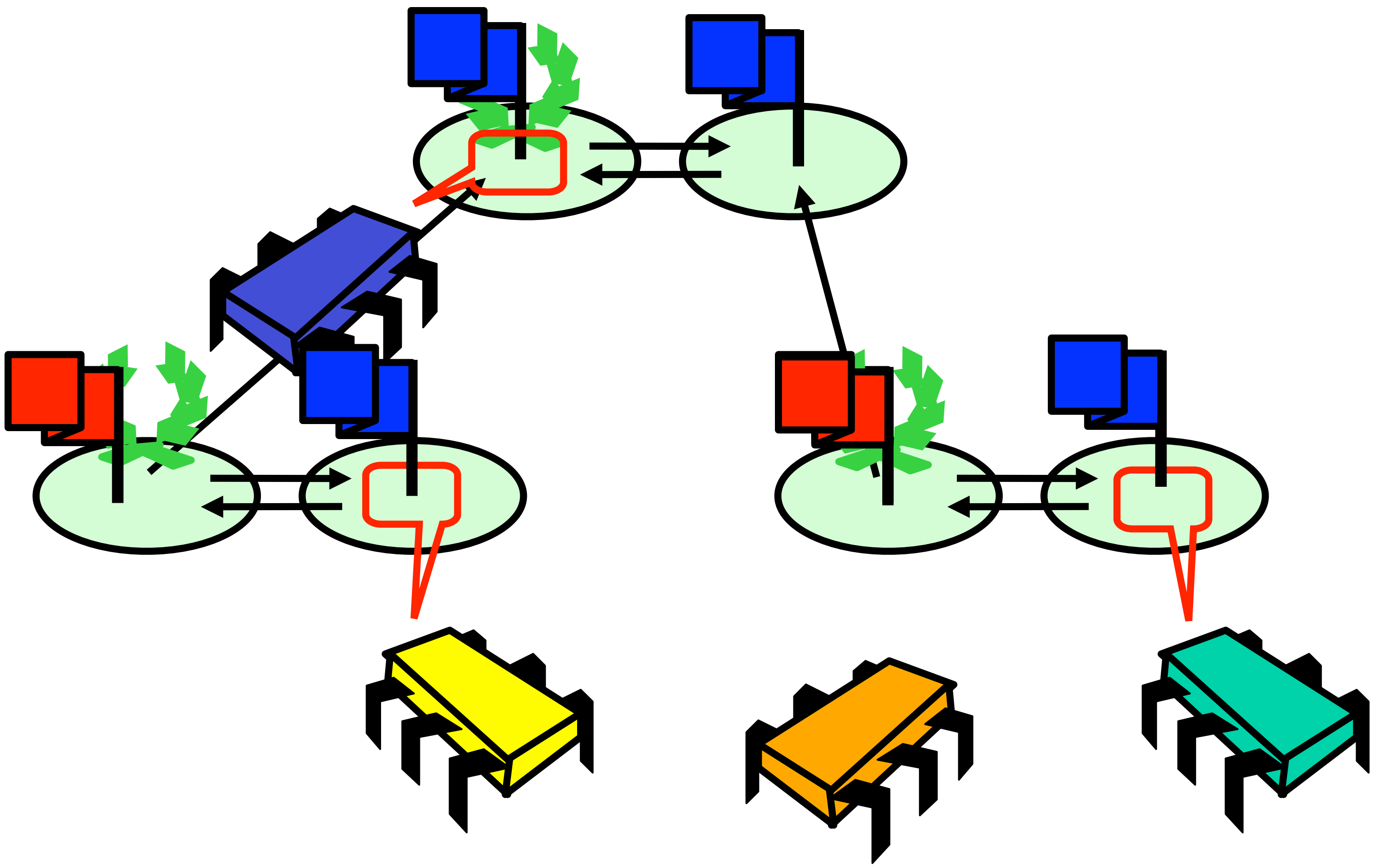
Tournament Tree Barriers



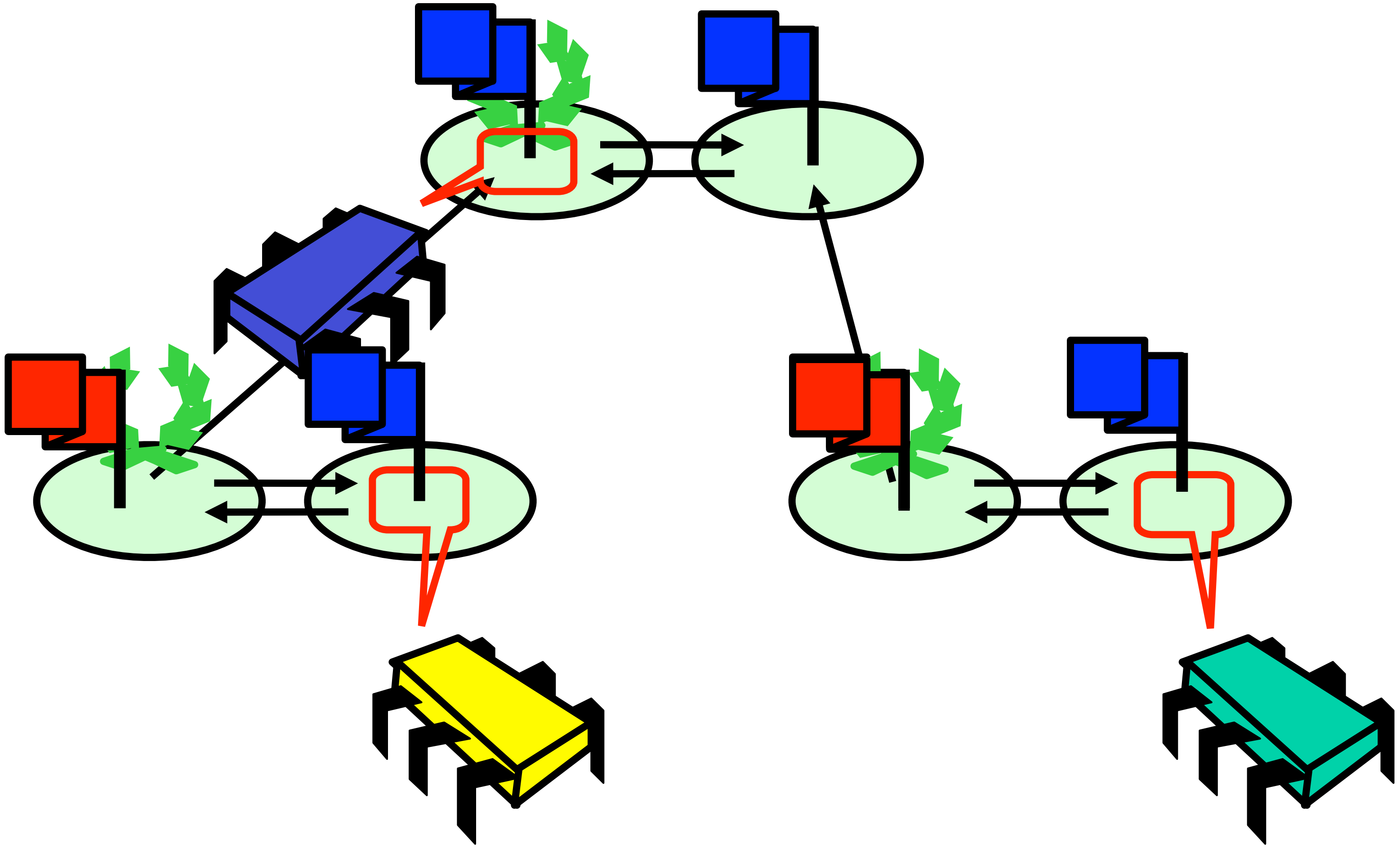
Tournament Tree Barriers



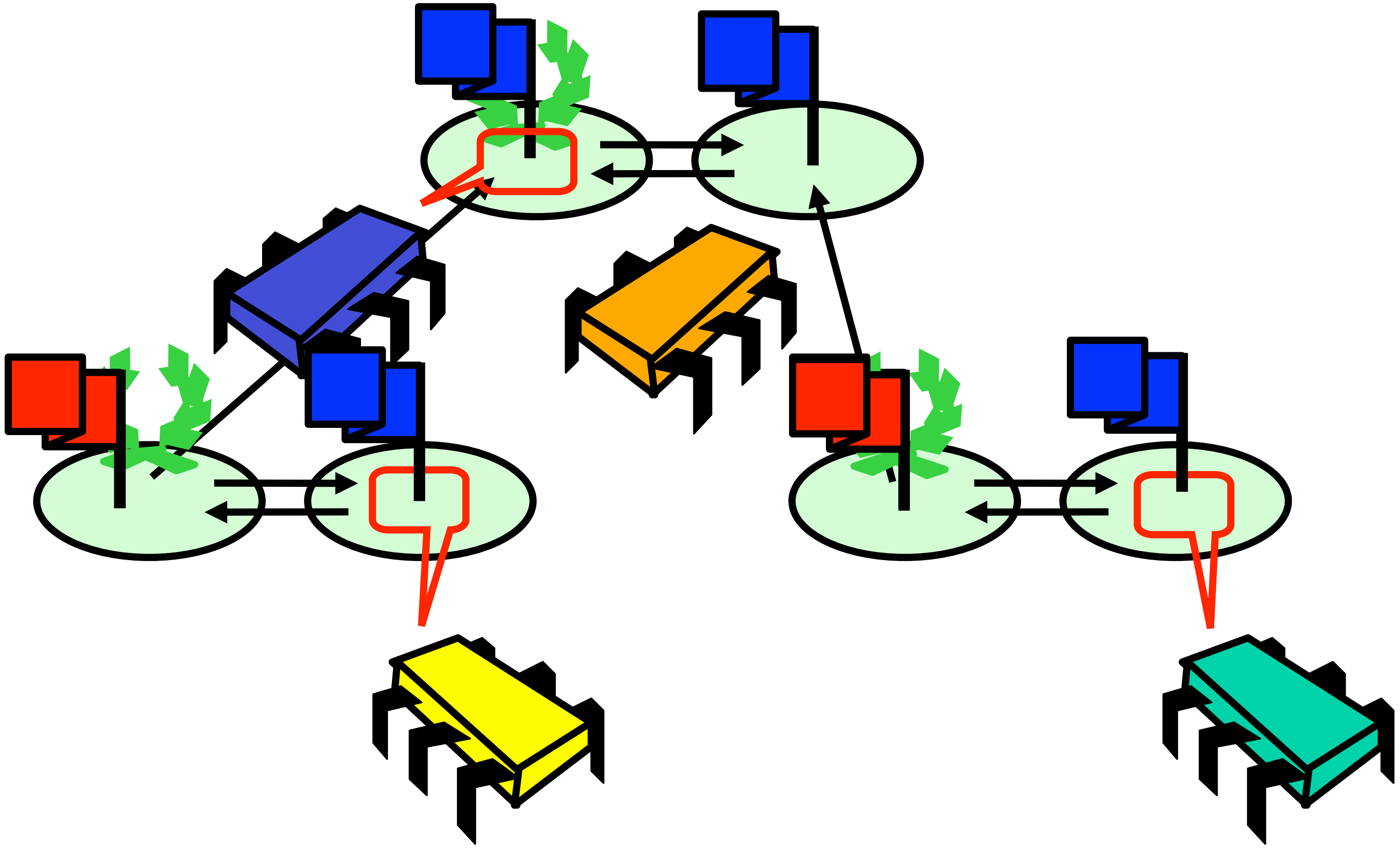
Tournament Tree Barriers



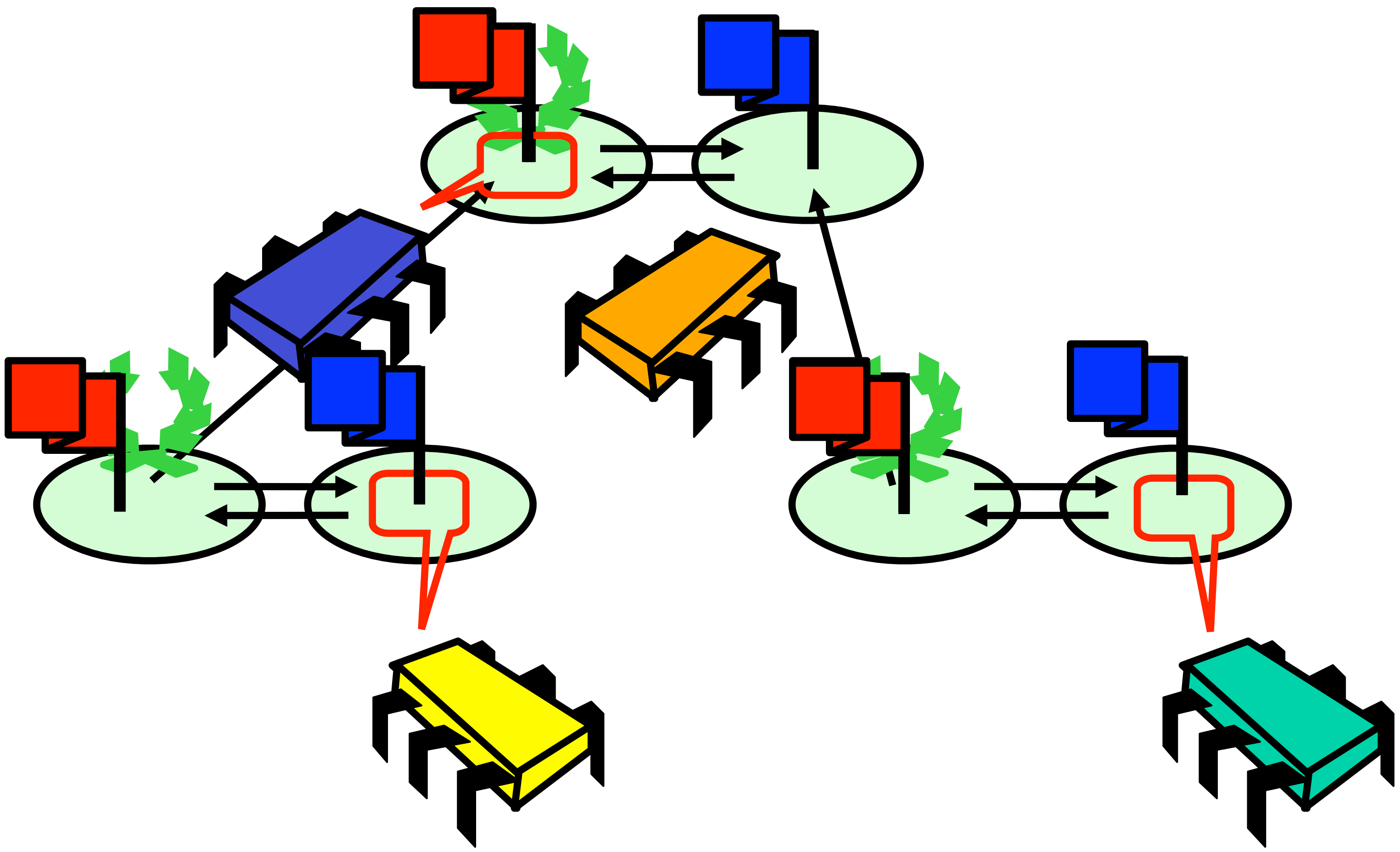
Tournament Tree Barriers



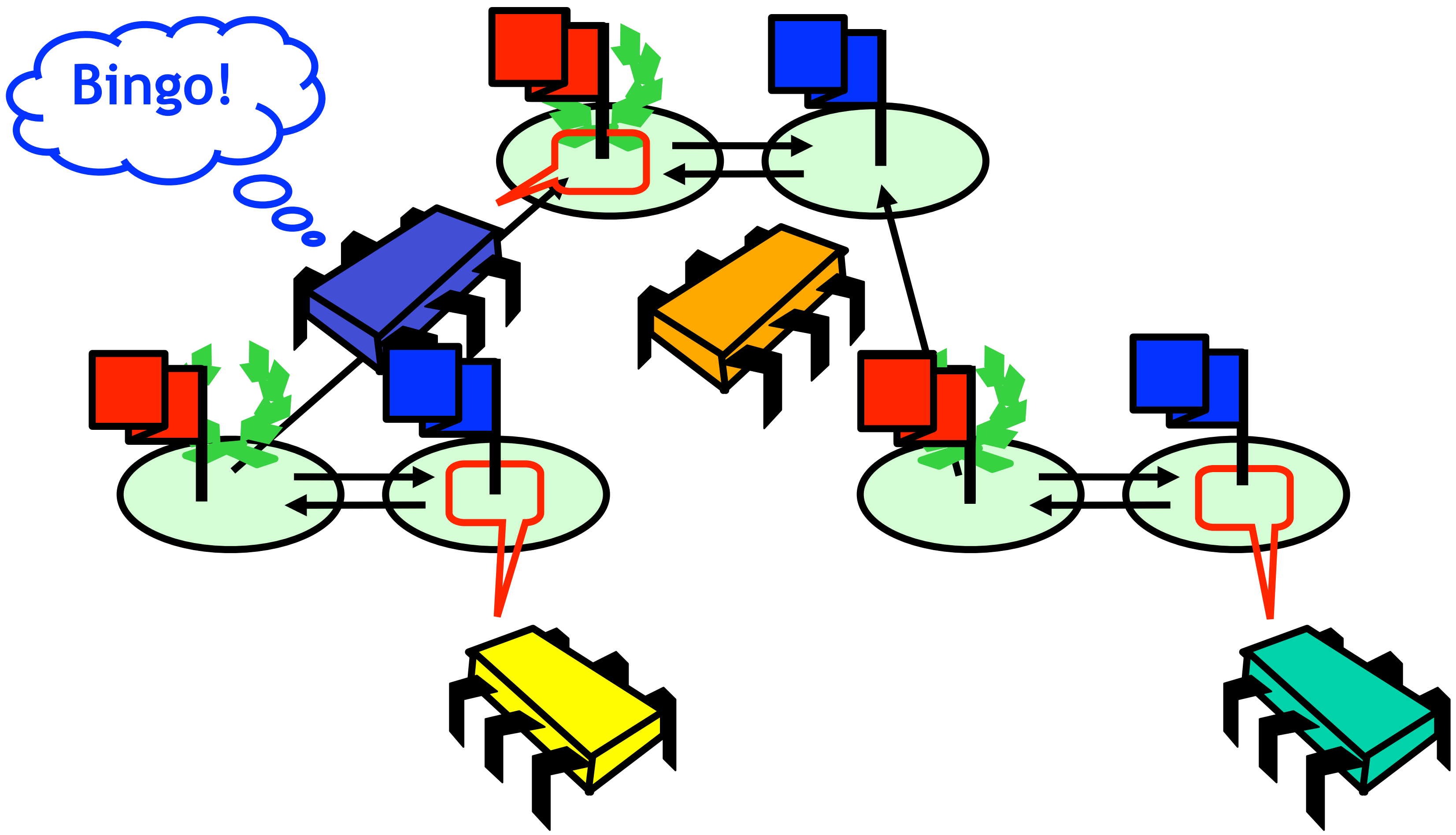
Tournament Tree Barriers



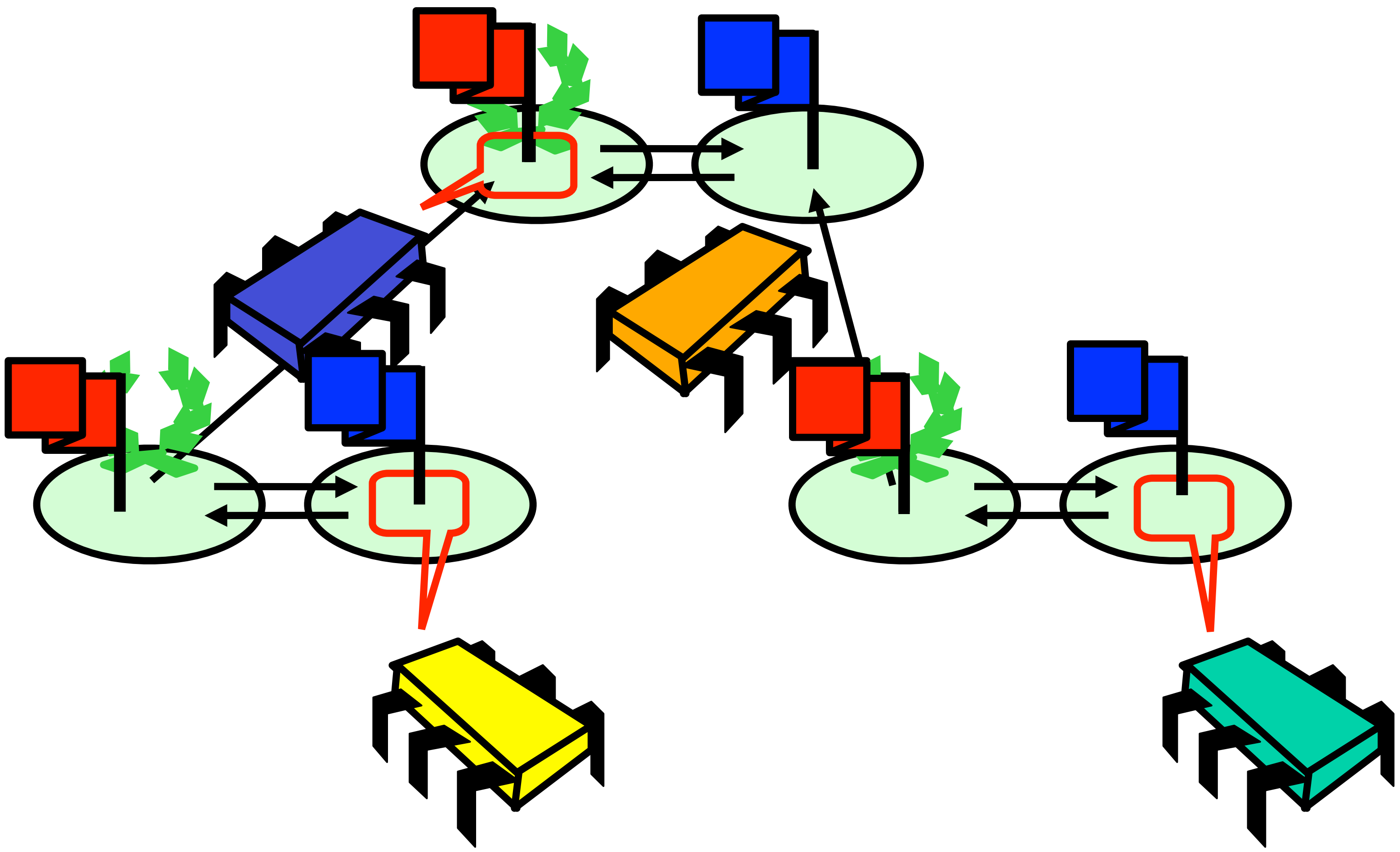
Tournament Tree Barriers



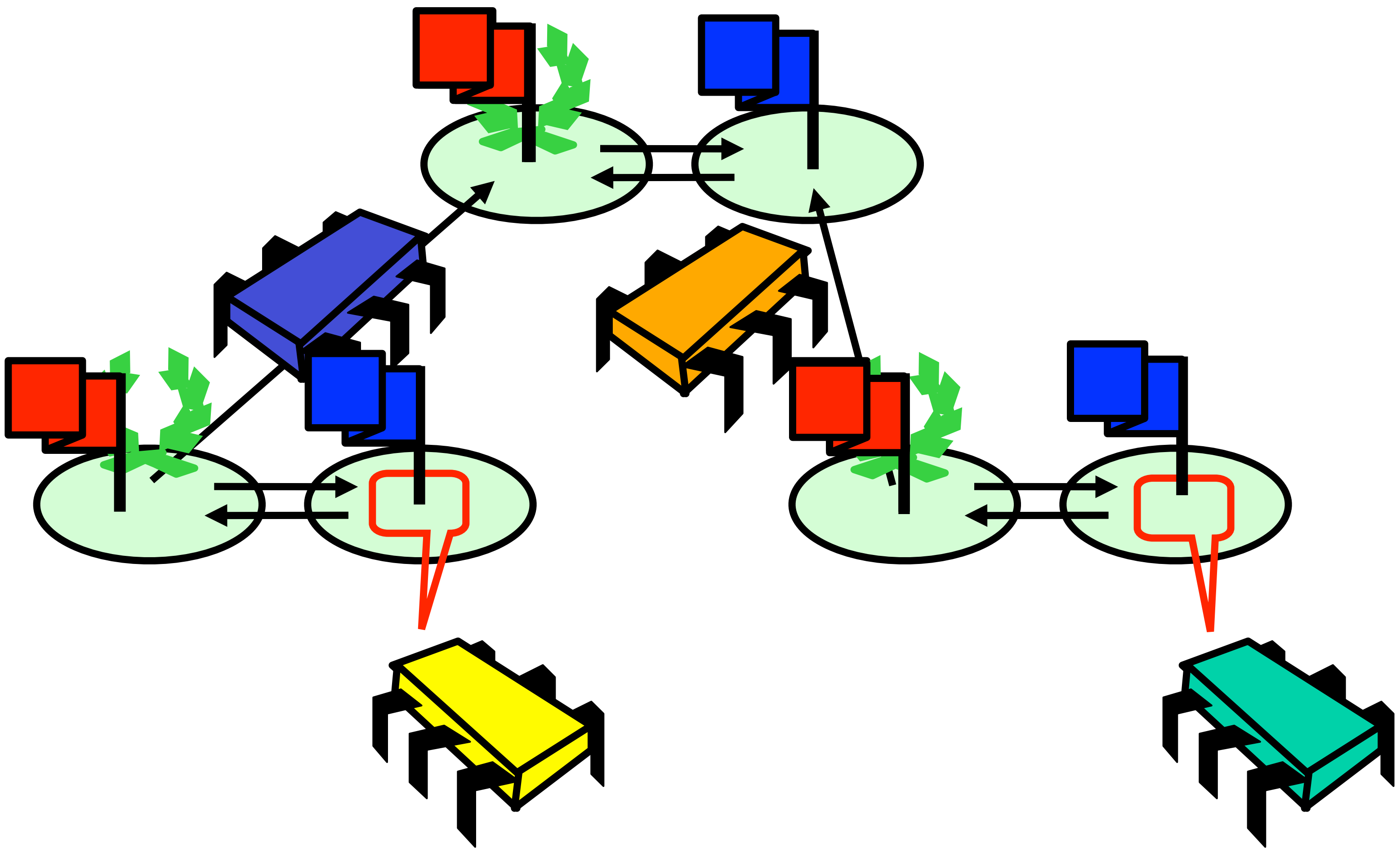
Tournament Tree Barriers



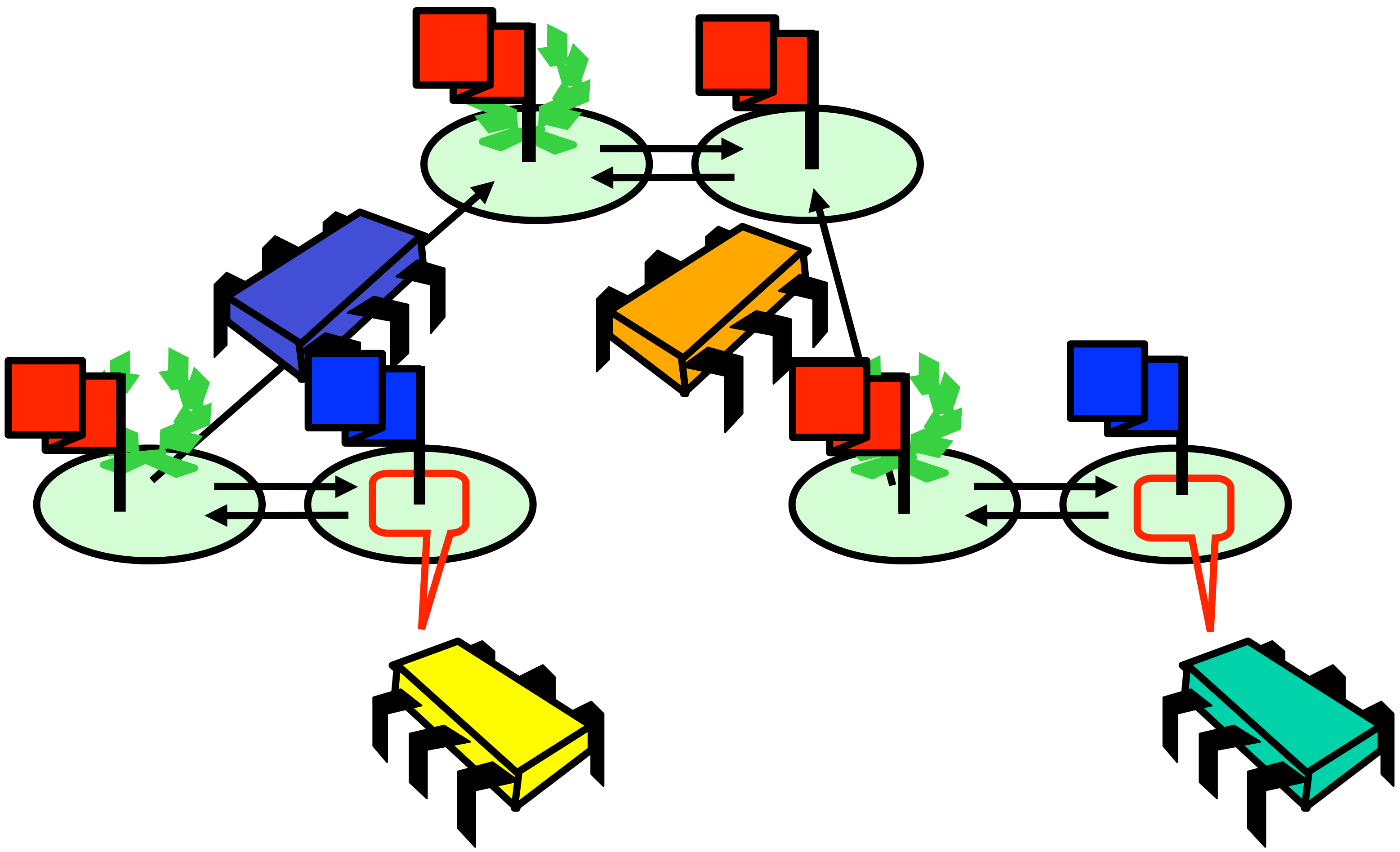
Tournament Tree Barriers



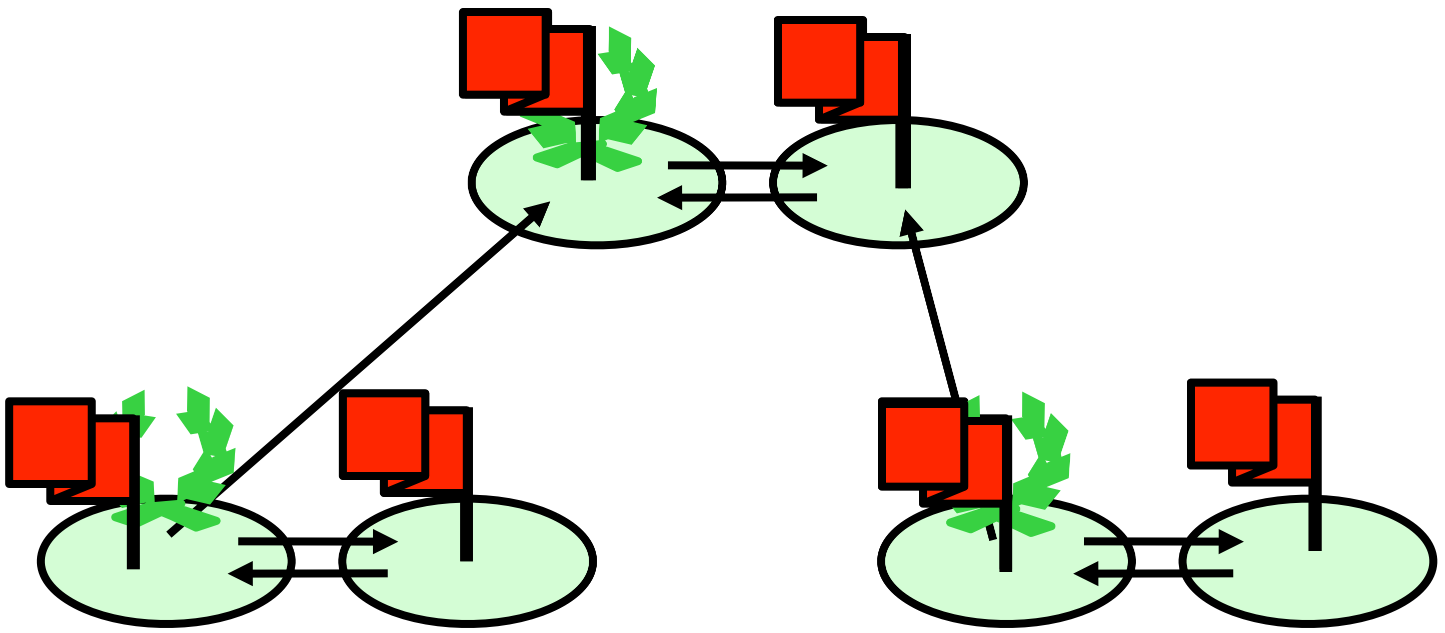
Tournament Tree Barriers



Tournament Tree Barriers



Tournament Tree Barriers



Sense-reversing: next time use blue flags

Tournament Barrier

```
class TBarrier {
  boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```

Tournament Barrier

```
class TBarrier {
  boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```

Notifications delivered here

Tournament Barrier

```
class TBarrier {
  boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```

Other thead at same level

Tournament Barrier

```
class TBarrier {
  boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```

Parent (winner) or null (loser)

Tournament Barrier

```
class TBarrier {
  boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```

Am I the root?

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Tournament Barrier

Current sense

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Le root, c'est moi

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

I am already a winner

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Wait for partner

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Synchronize upstairs

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Inform partner

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Inform partner

Order is important (why?)

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Natural-born loser

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Tell partner I’m here

Tournament Barrier

```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

Wait for notification from partner

Remarks

- No need for read-modify-write calls
- Each thread spins on fixed location
  - Good for bus-based architectures
  - Good for NUMA architectures

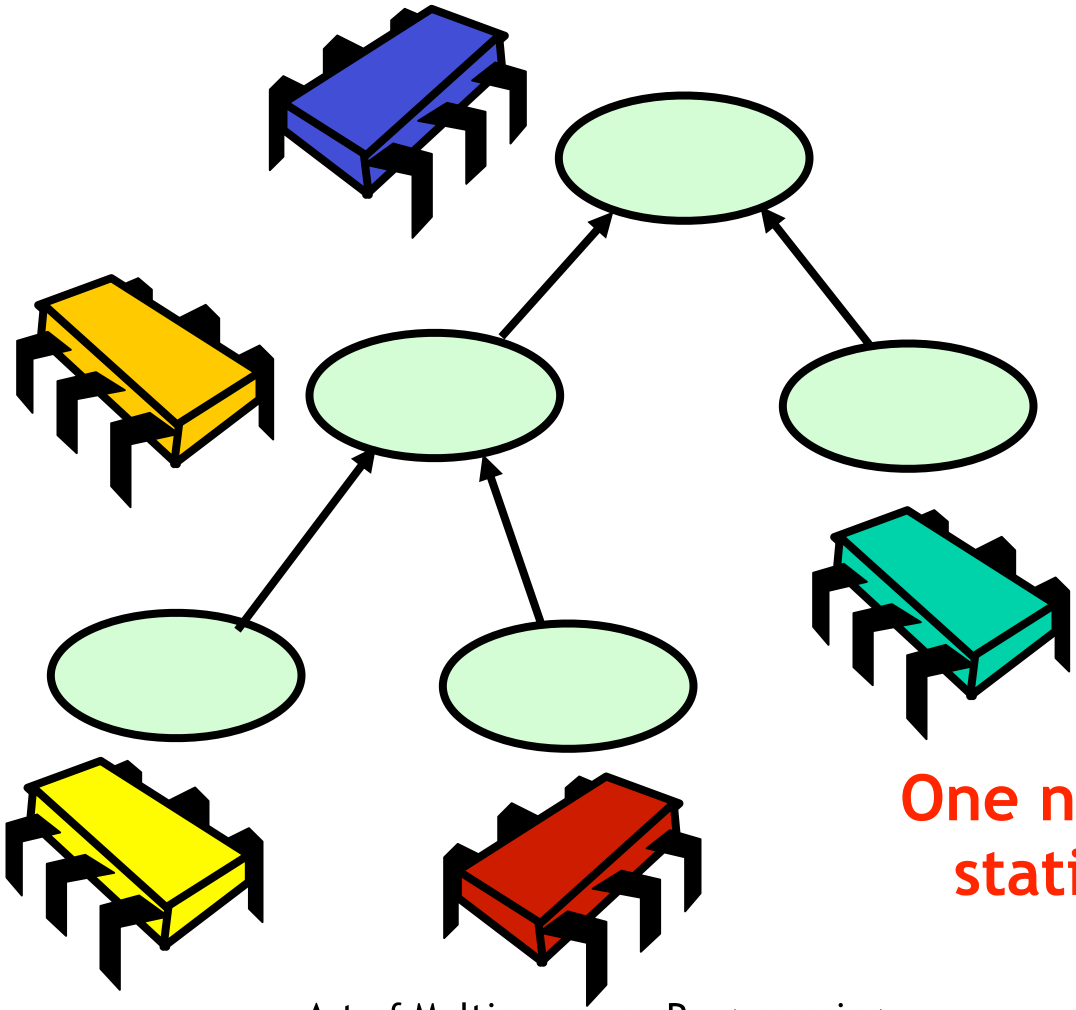
Ideas So Far

- Sense-reversing
  - Reuse without reinitializing
- Combining tree
  - Like counters, locks …
- Tournament tree
  - Optimized combining tree

Which is best for Multicore?

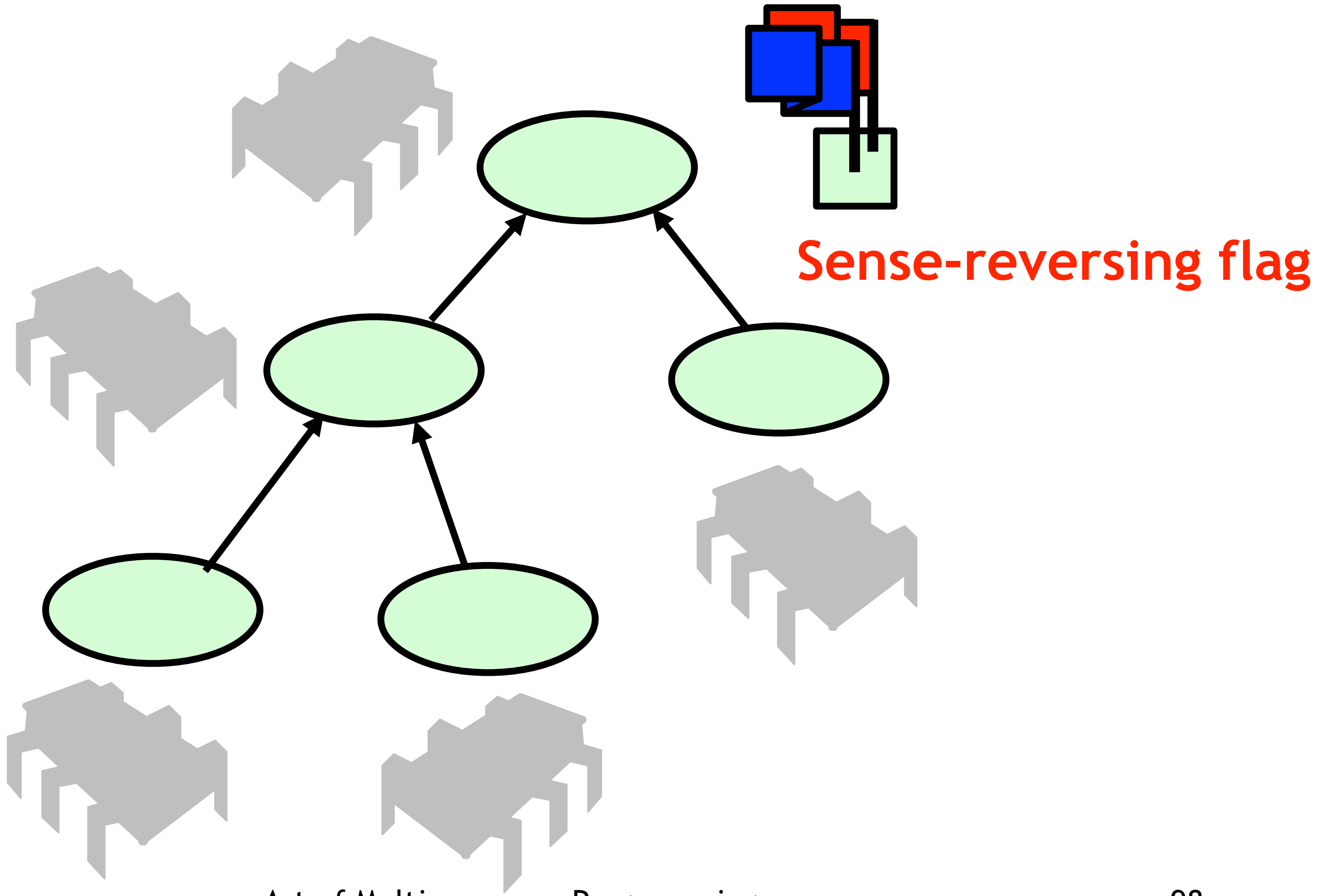
- On a cache coherent multicore chip: perhaps none of the above…
- Here is another (arguably) better algorithm …

Static Tree Barrier

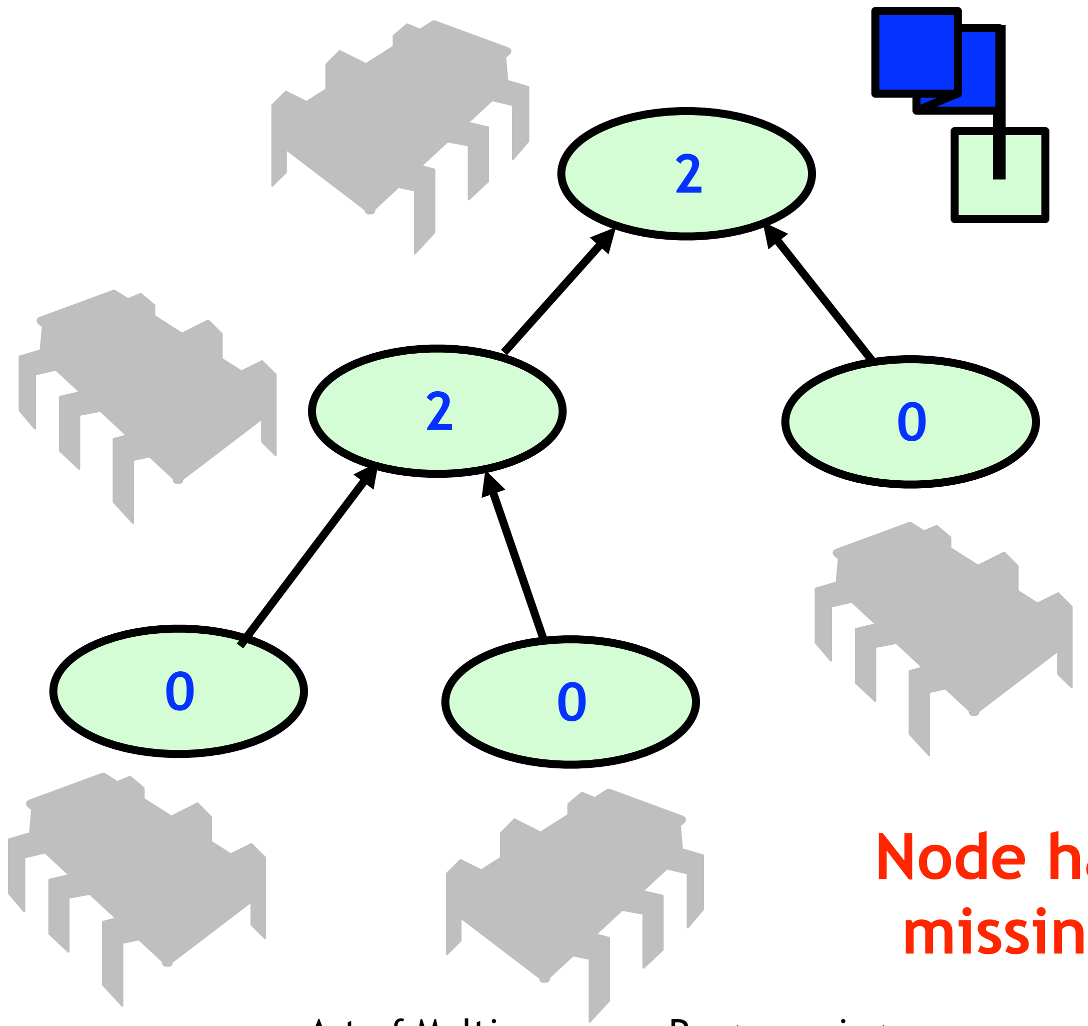


One node per thread, statically assigned

Static Tree Barrier

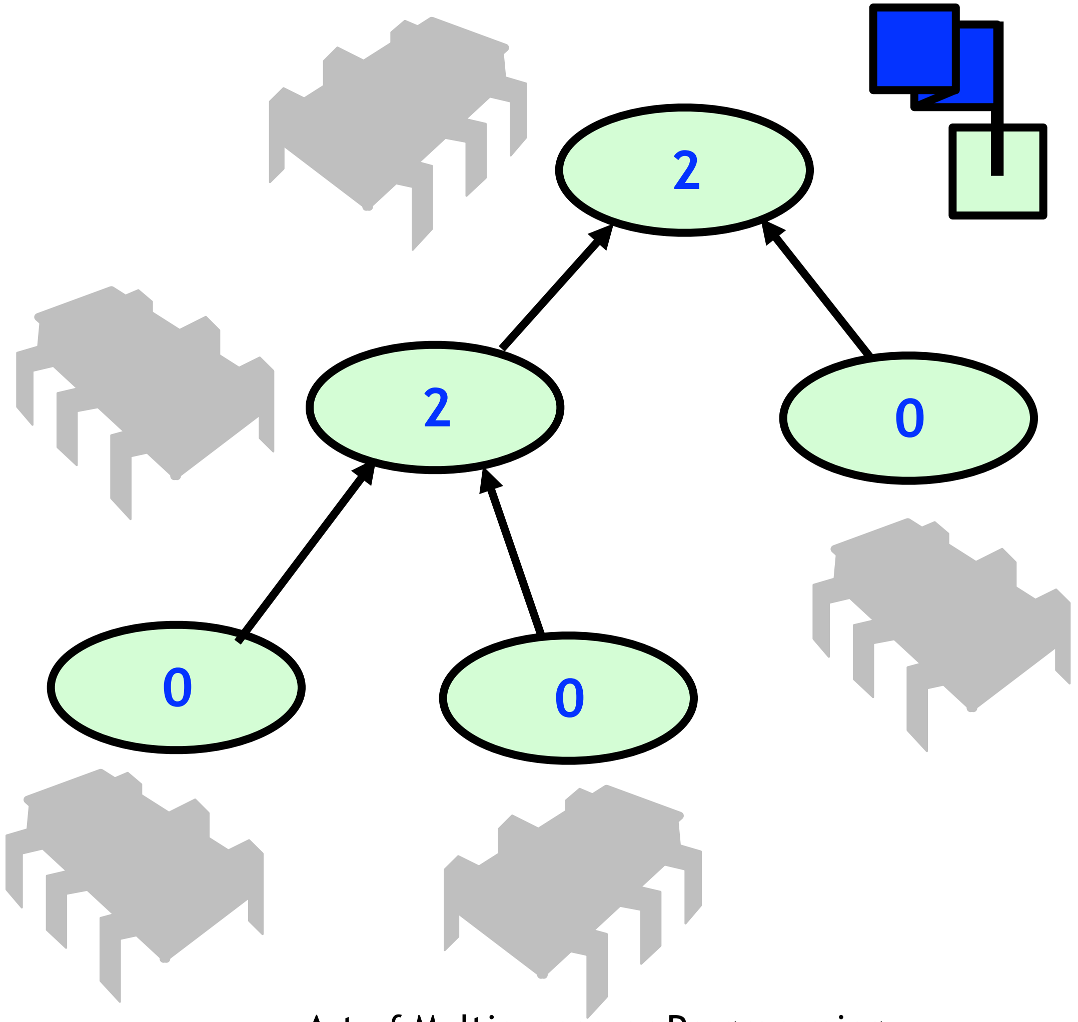


Static Tree Barrier

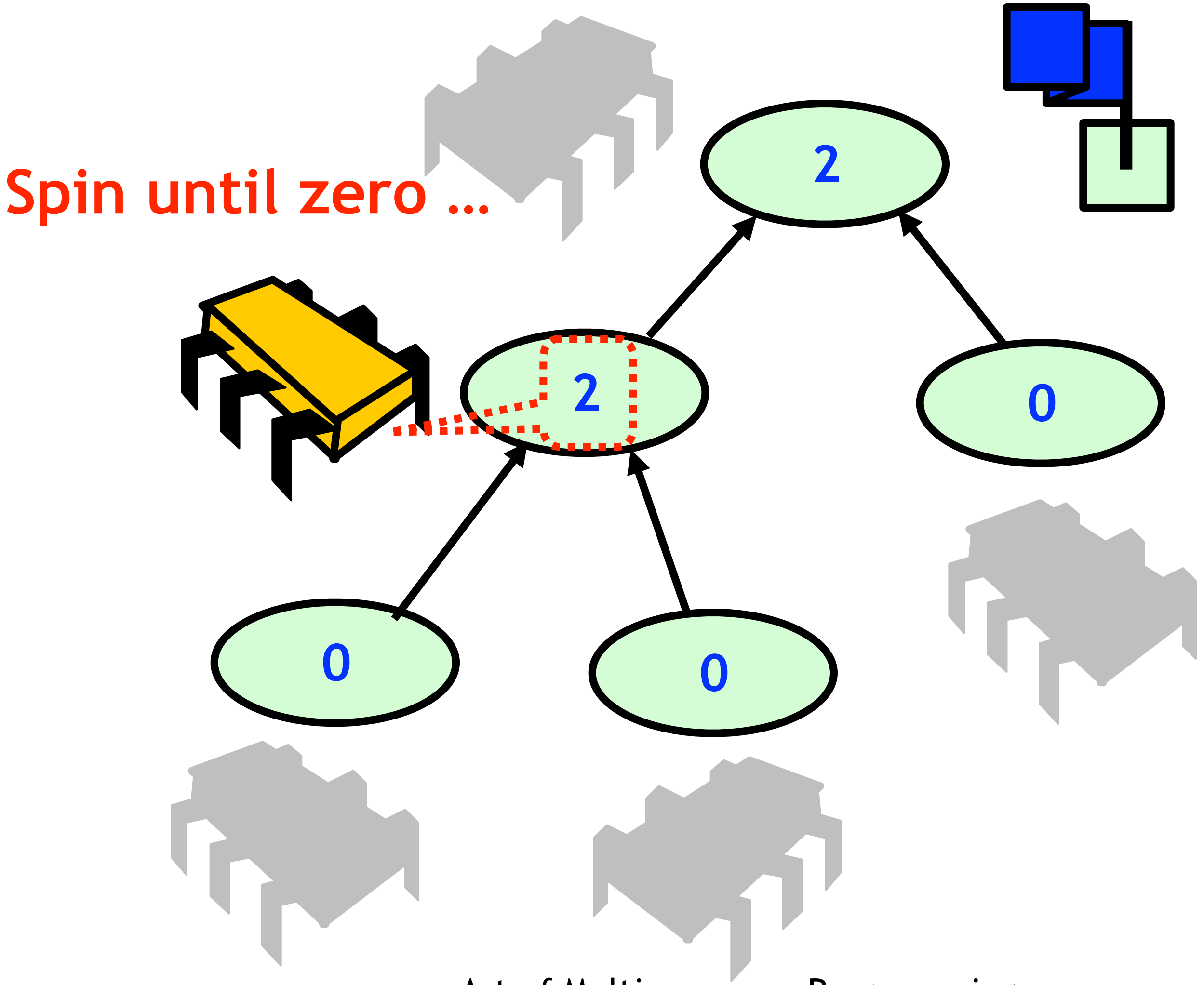


Node has count of missing children

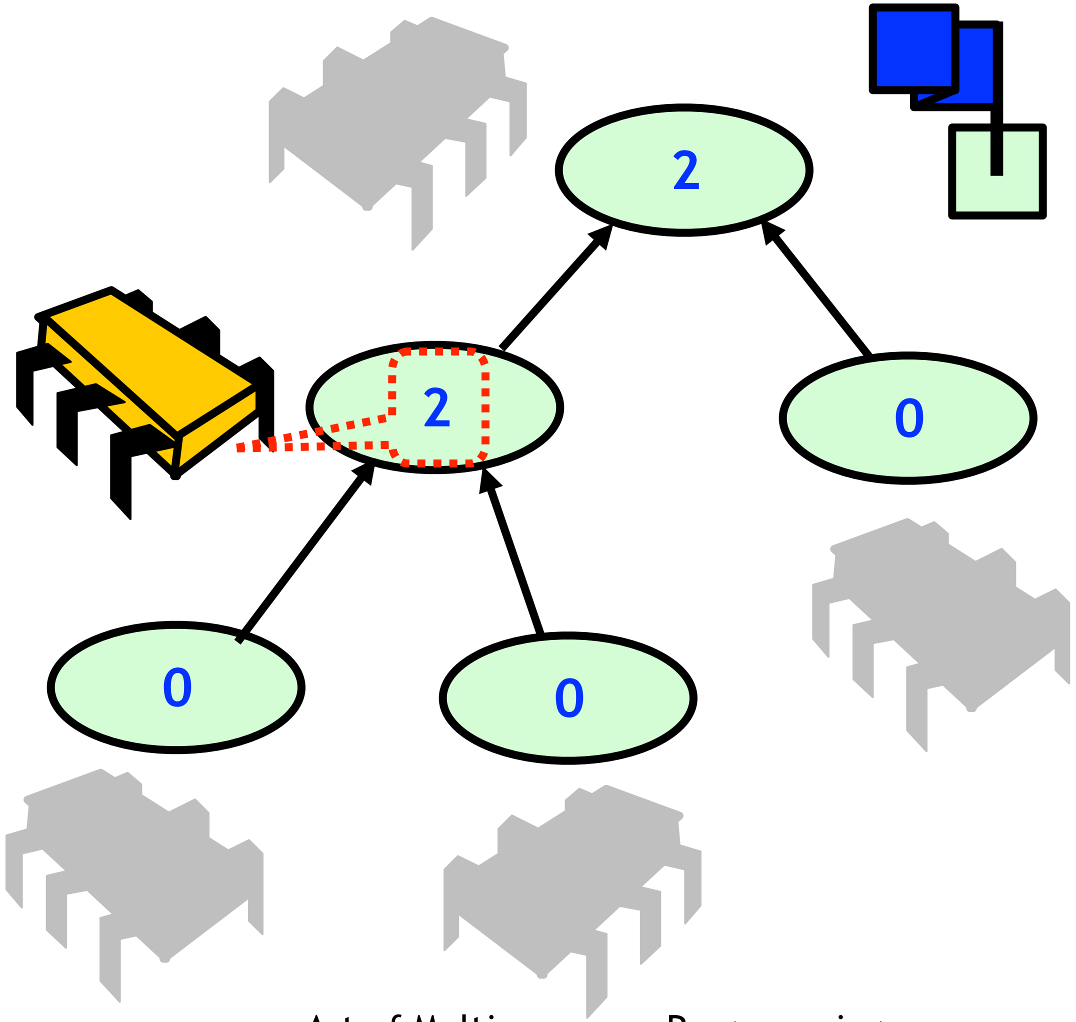
Static Tree Barrier



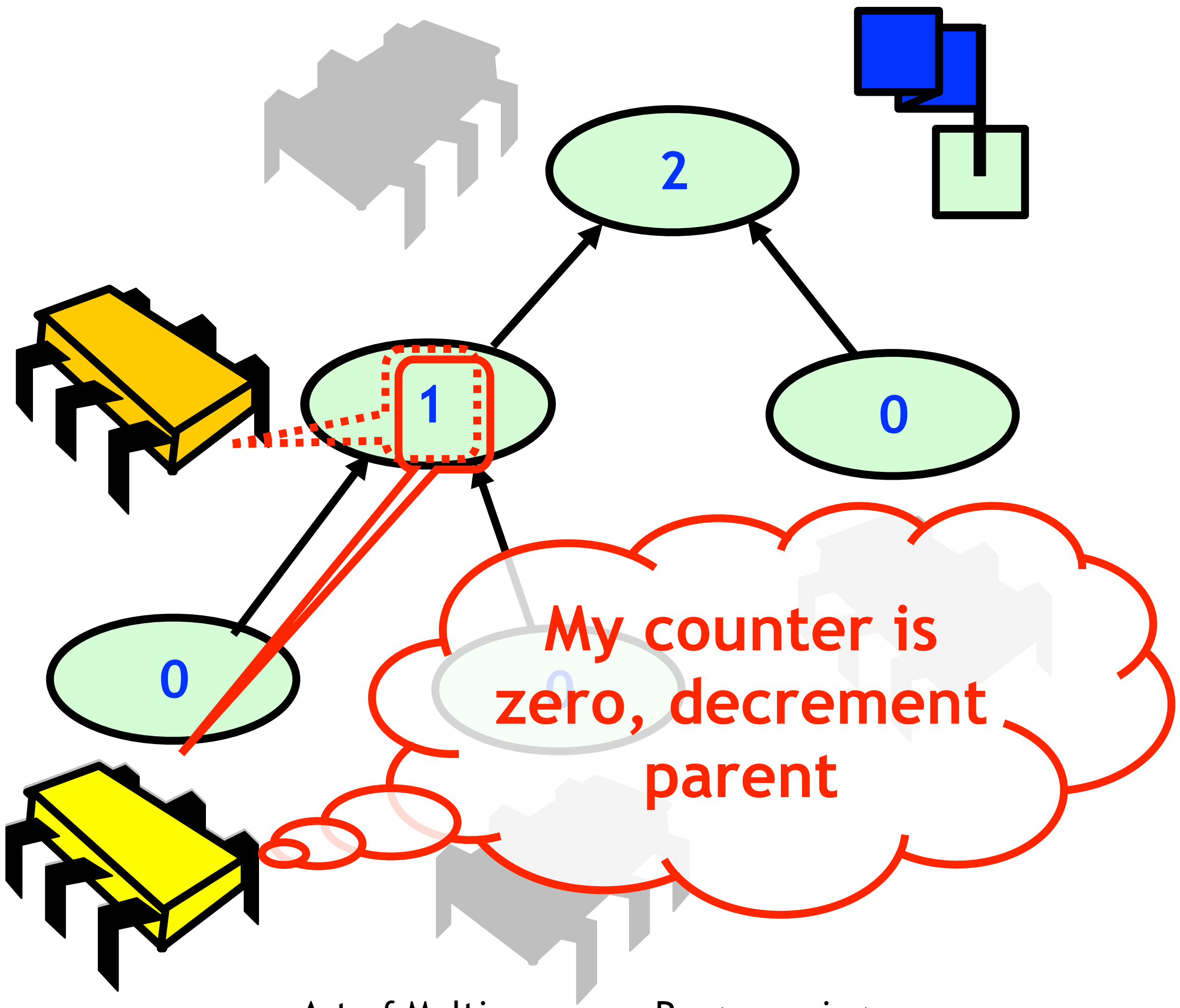
Static Tree Barrier



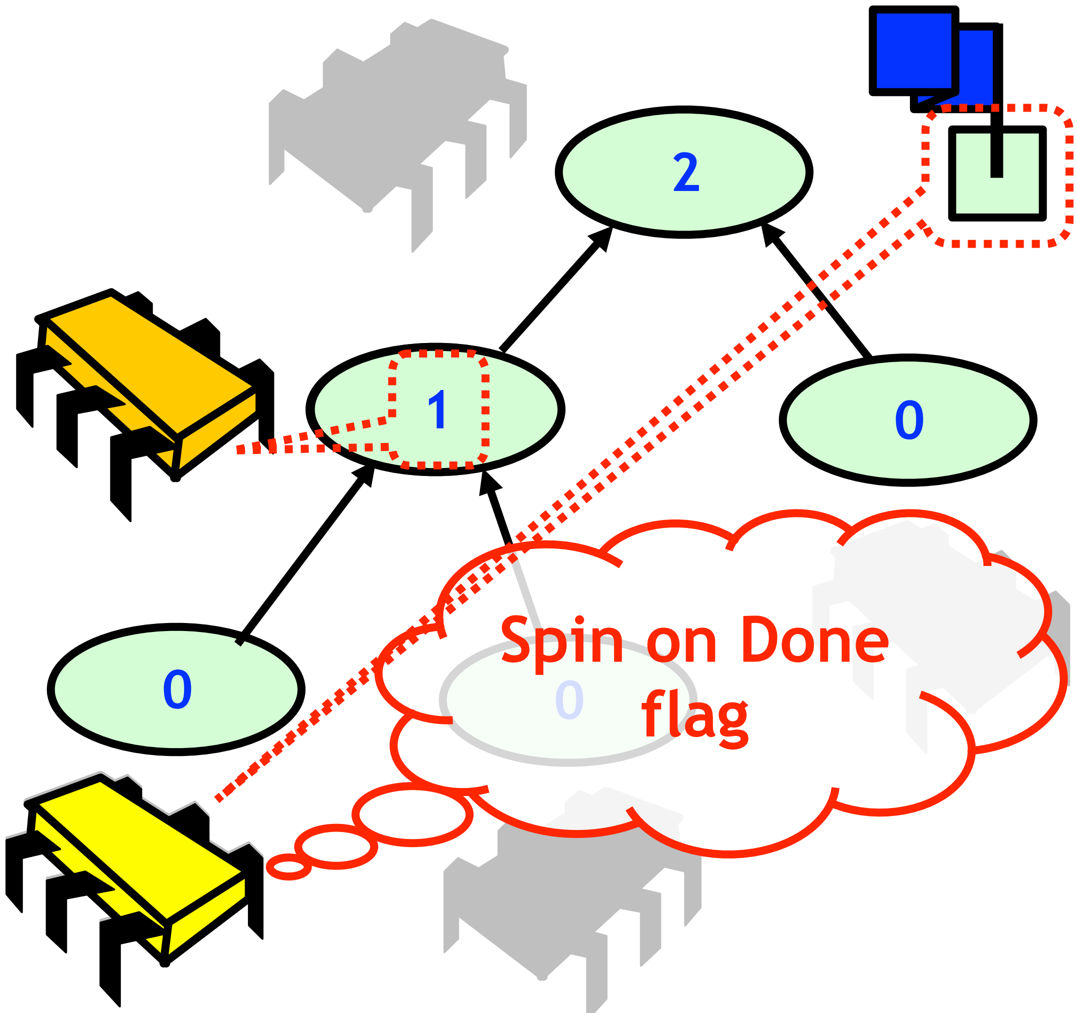
Static Tree Barrier



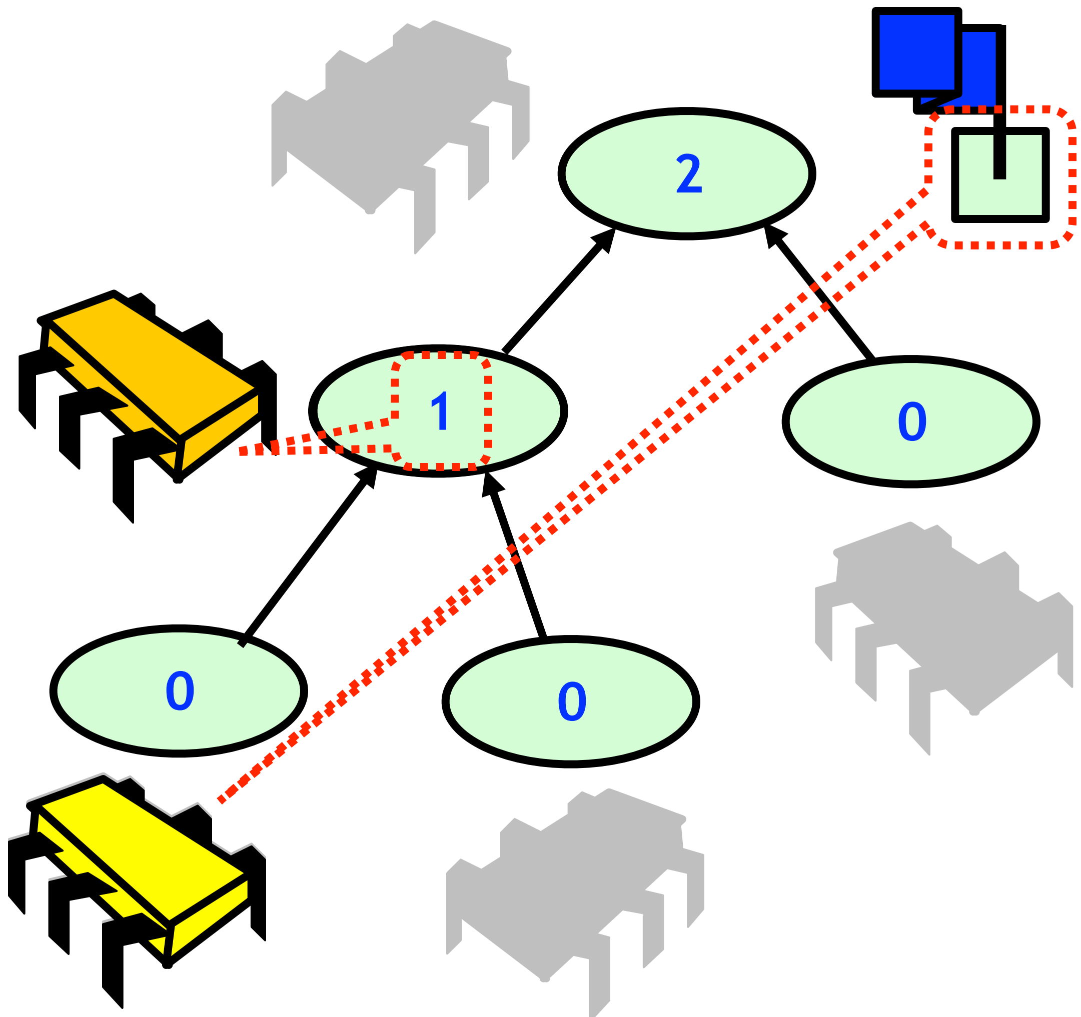
Static Tree Barrier



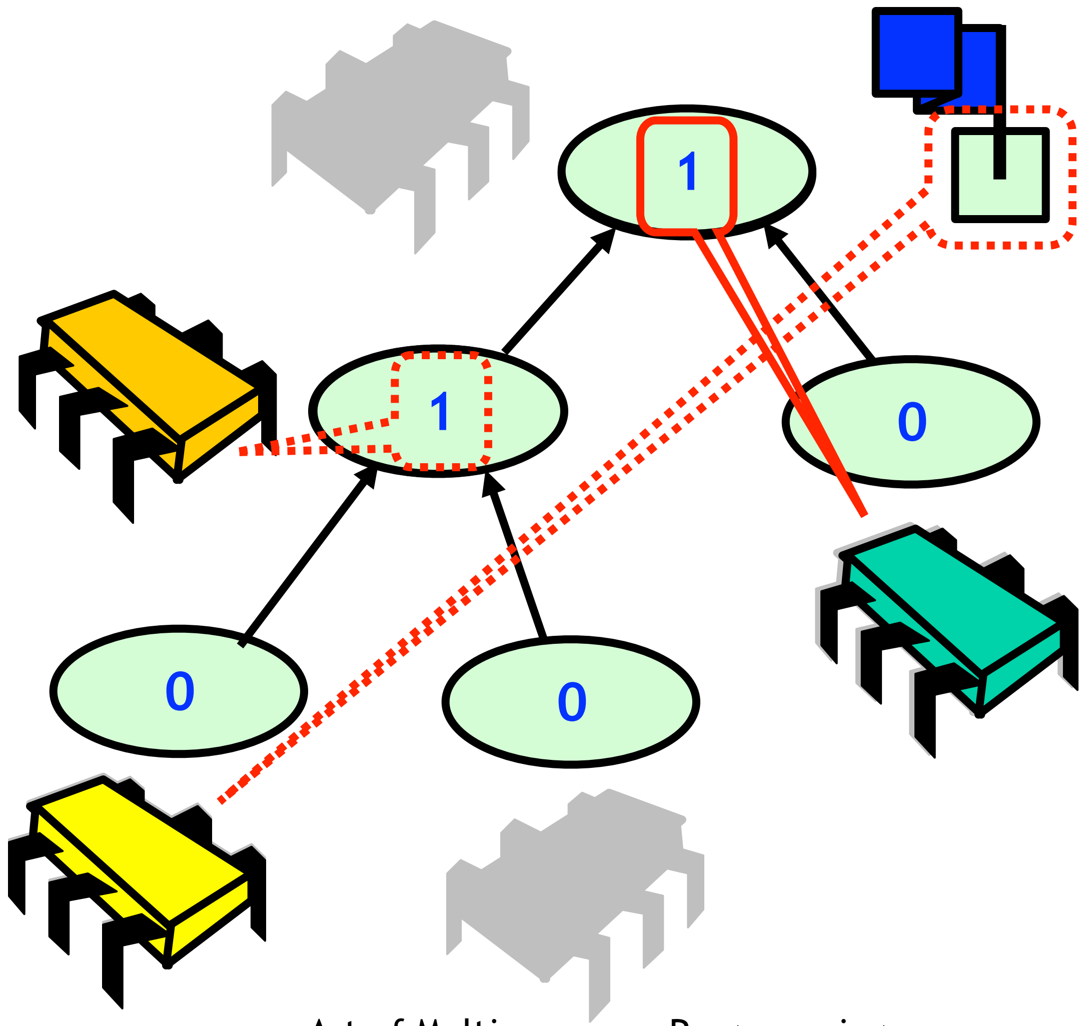
Static Tree Barrier



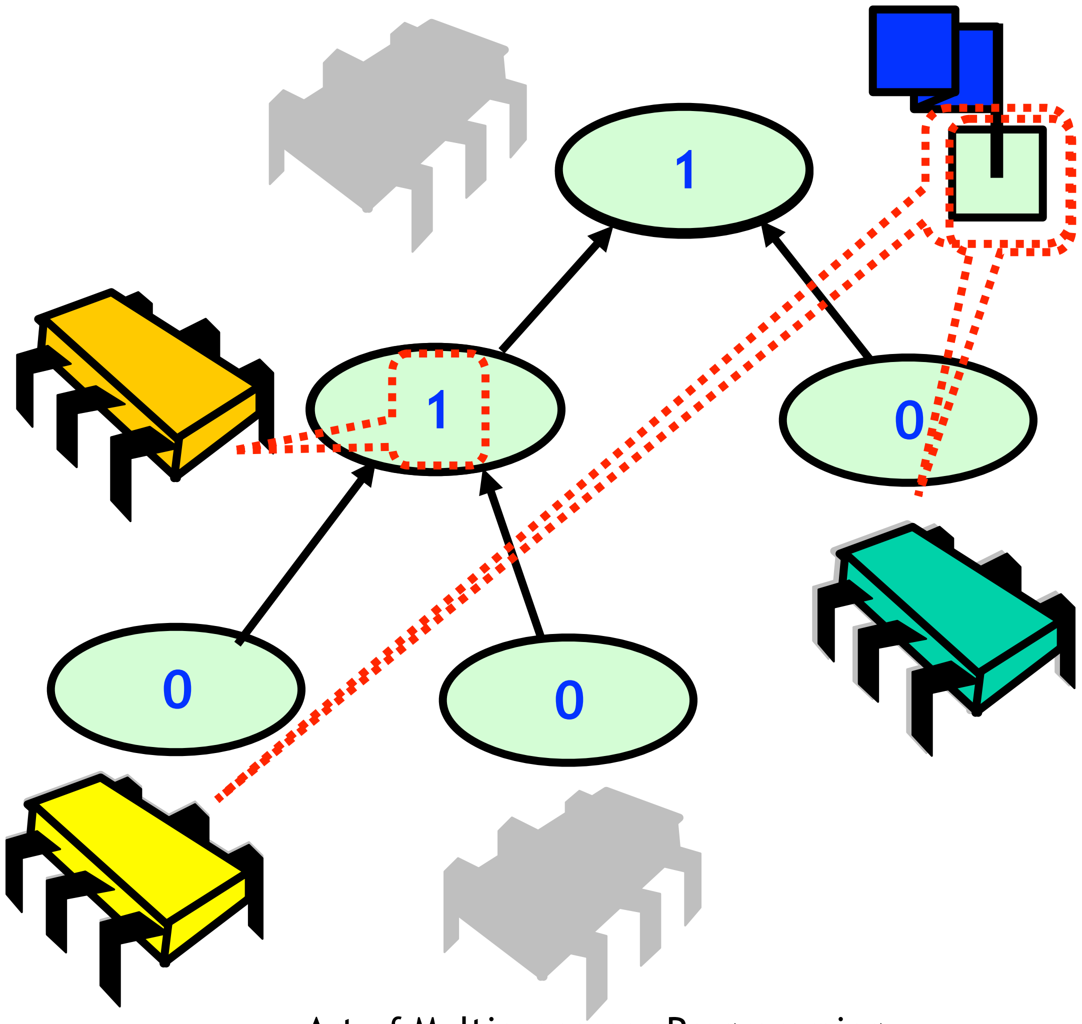
Static Tree Barrier



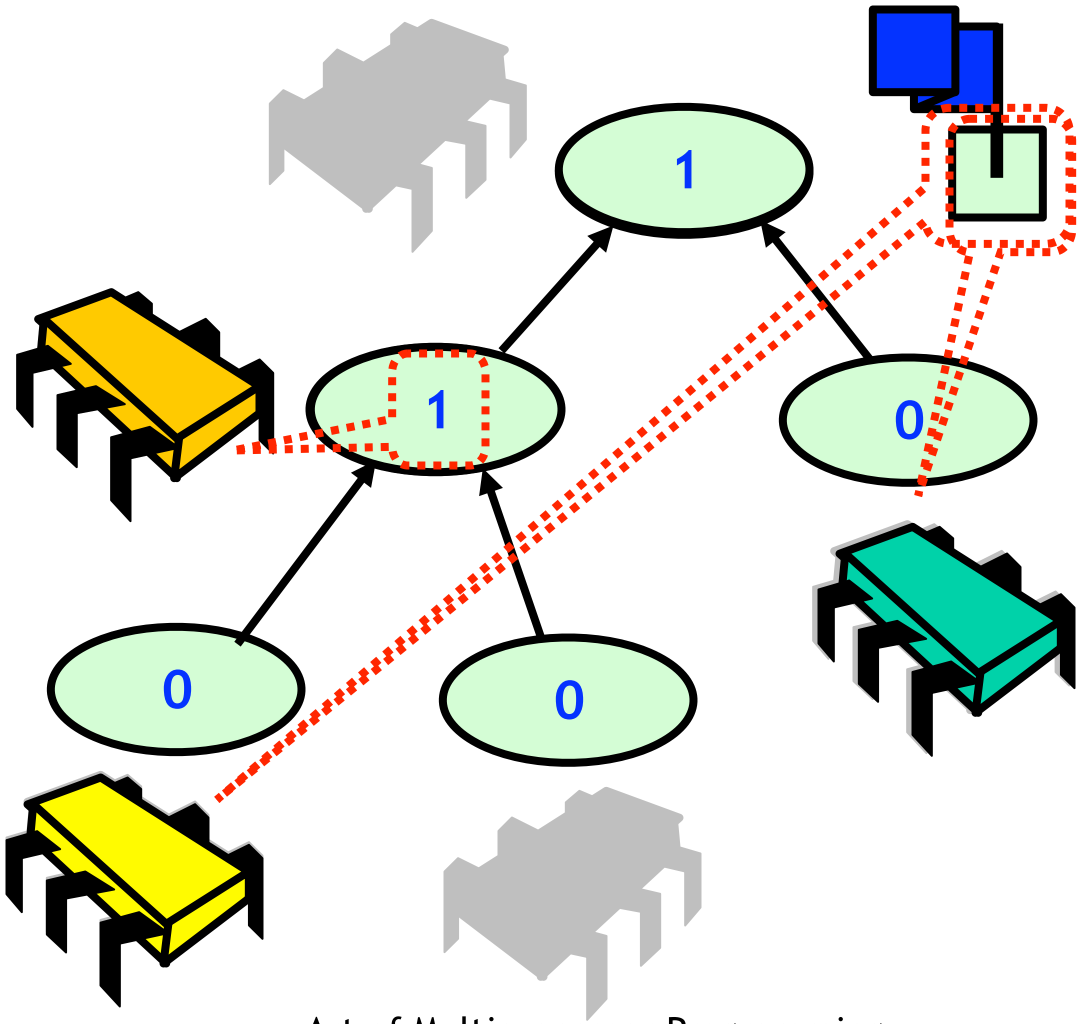
Static Tree Barrier



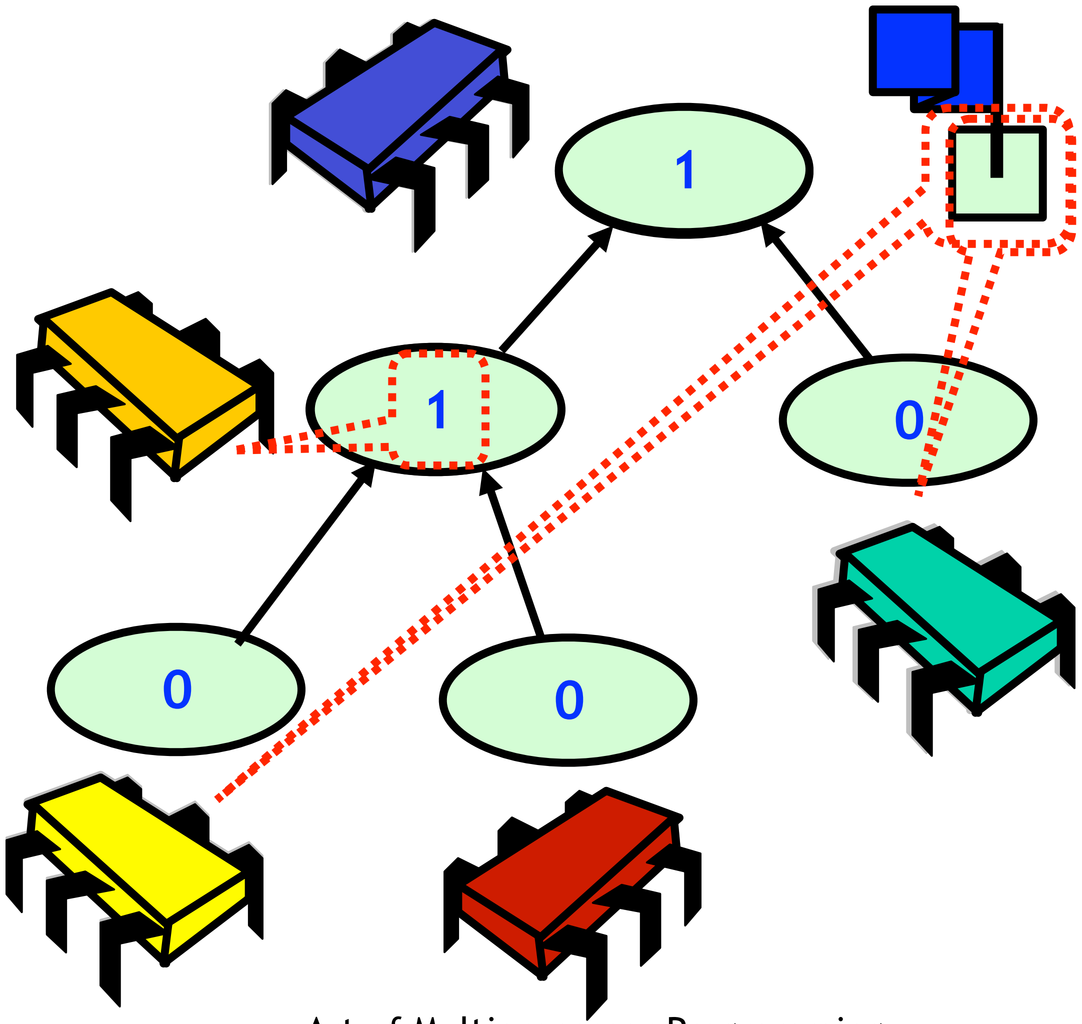
Static Tree Barrier



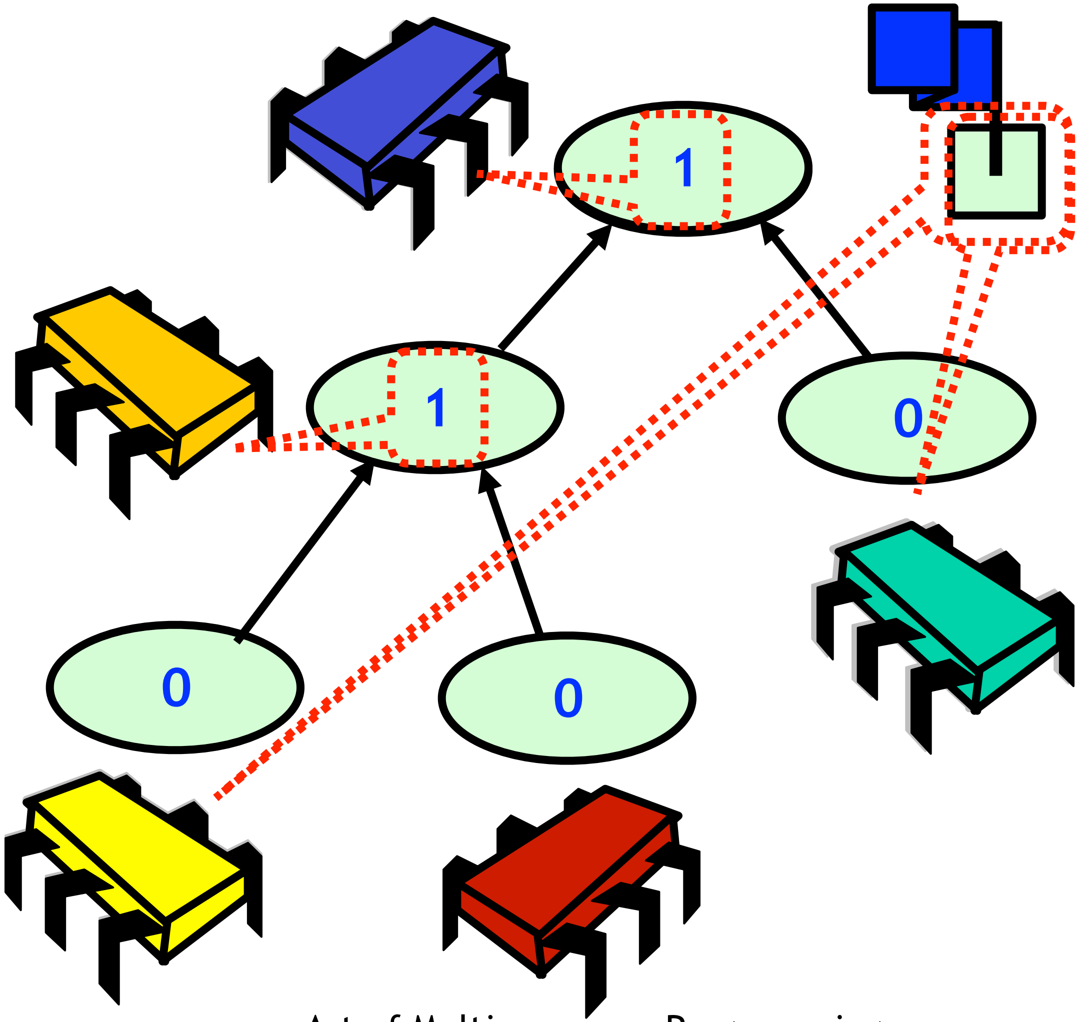
Static Tree Barrier



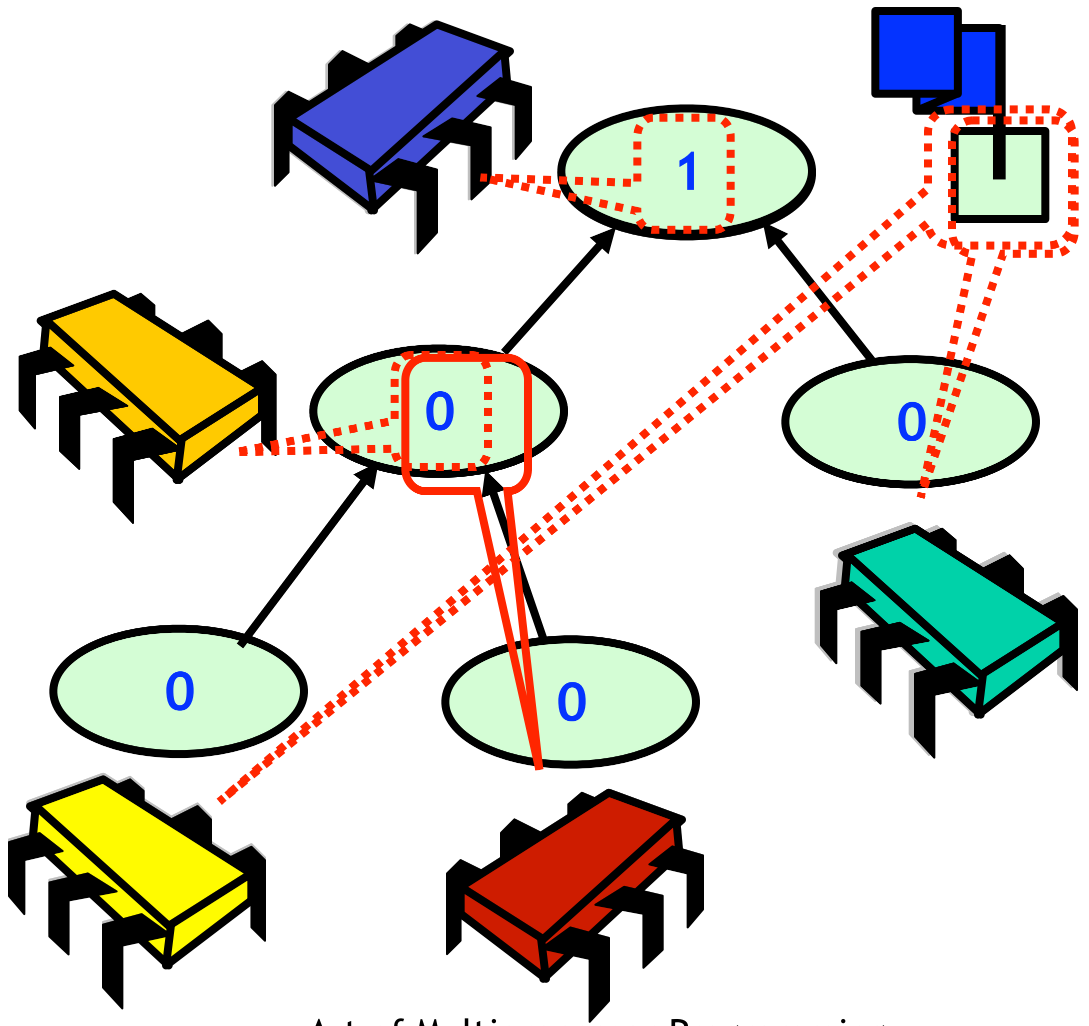
Static Tree Barrier



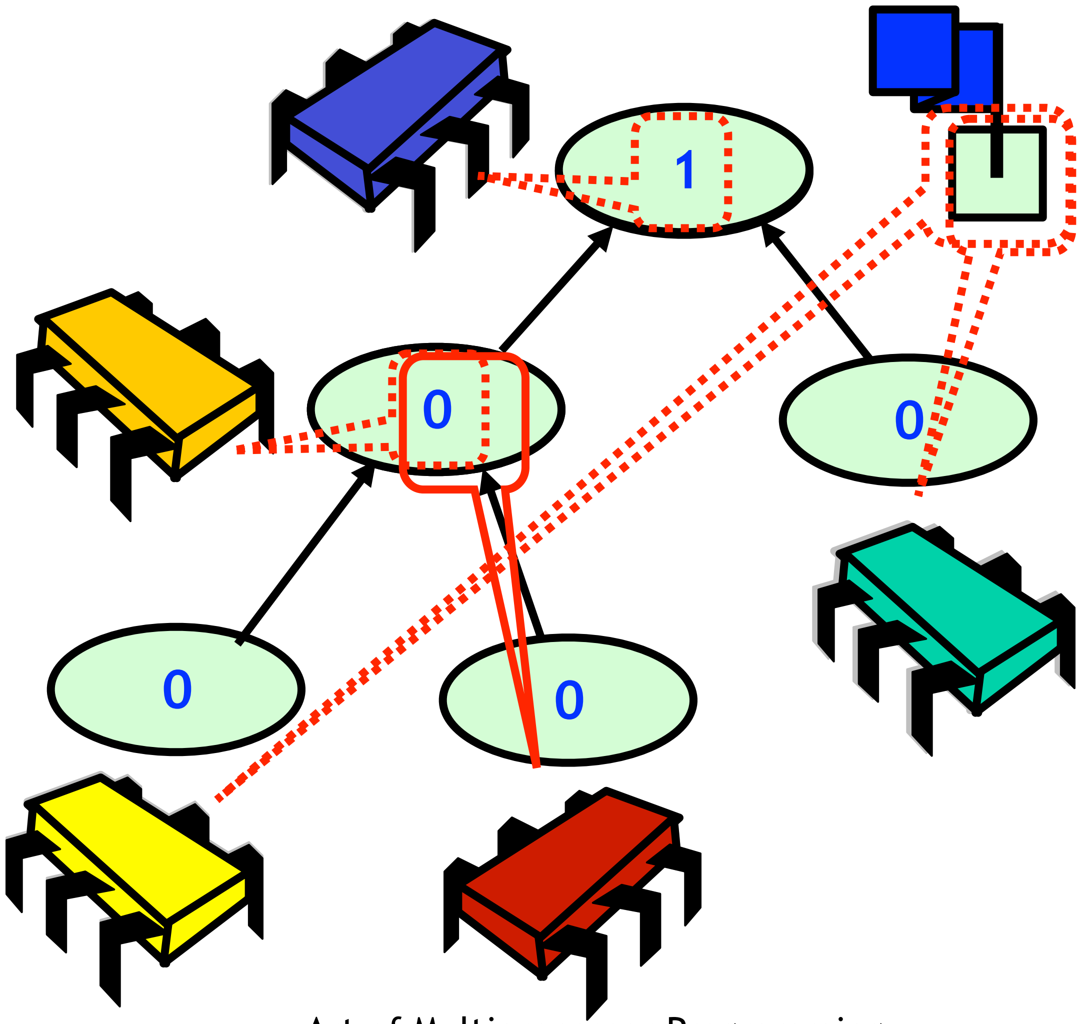
Static Tree Barrier



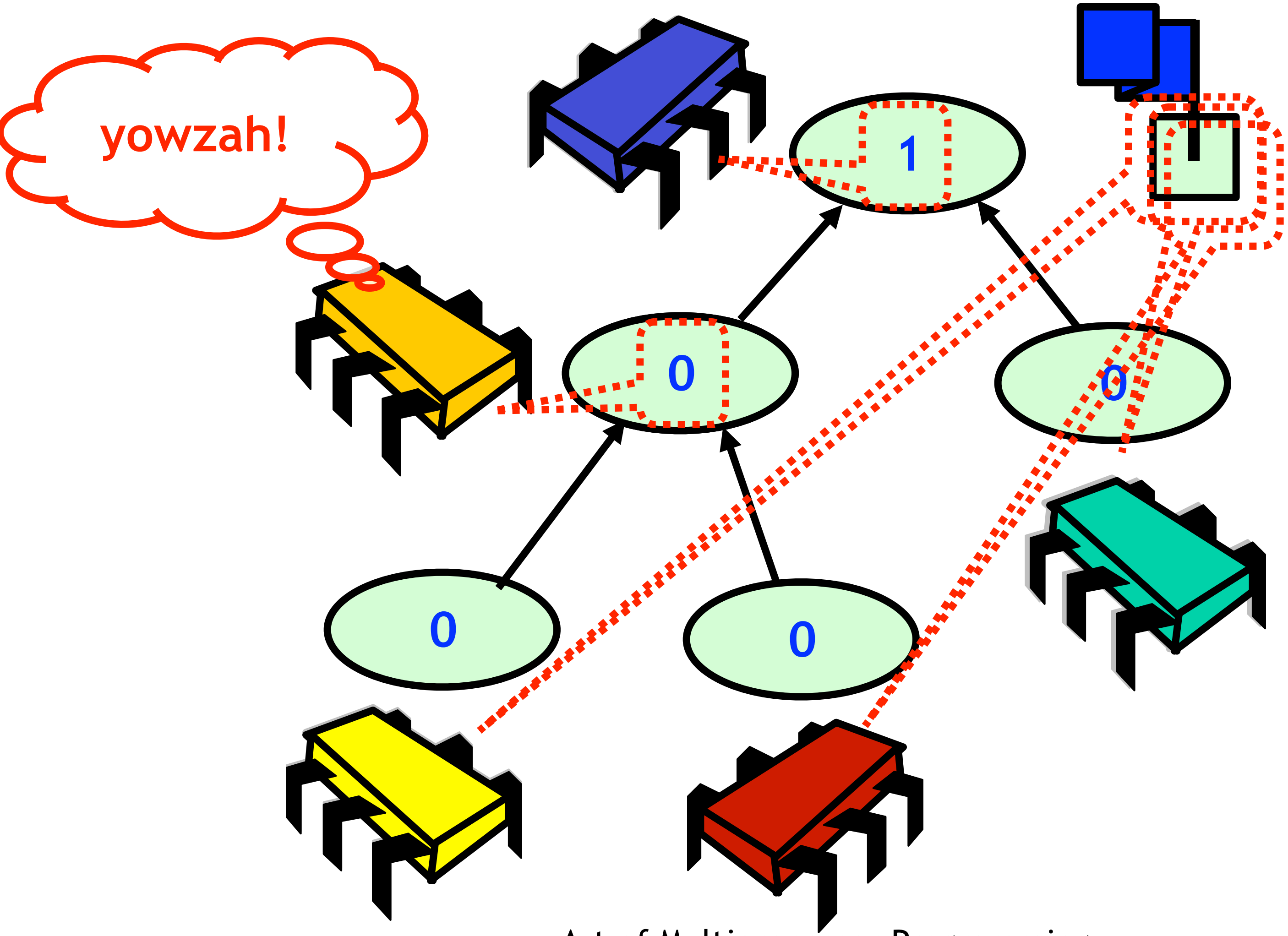
Static Tree Barrier



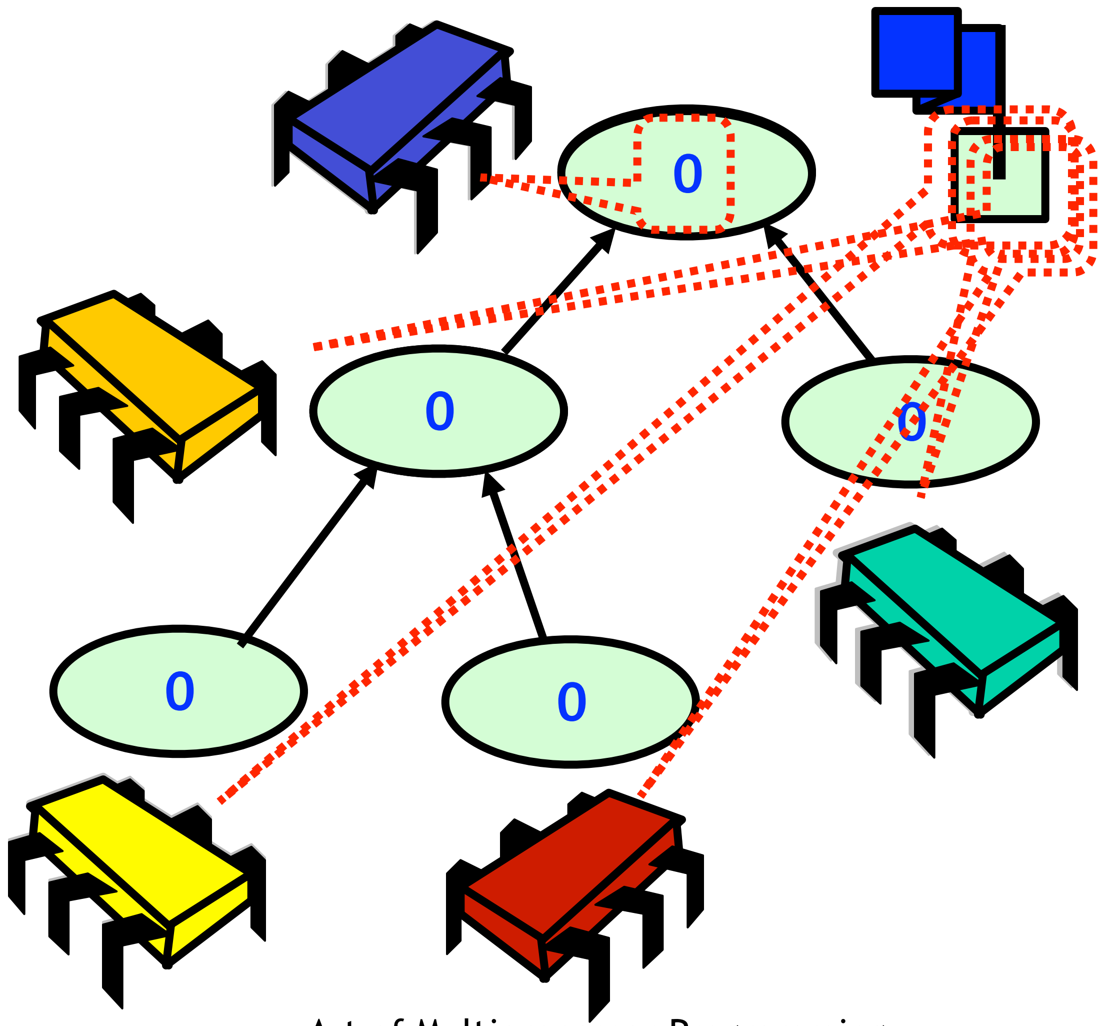
Static Tree Barrier



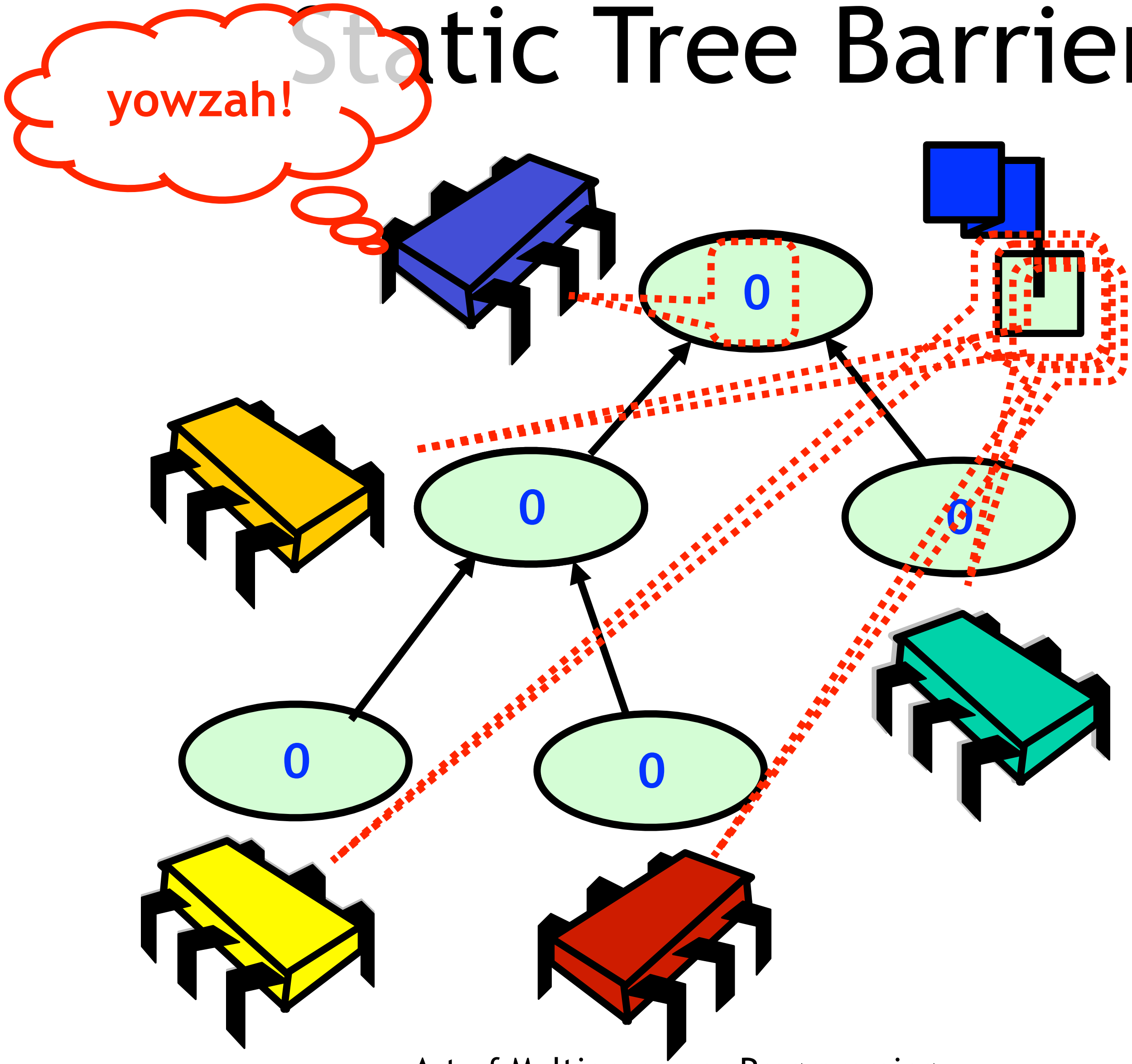
Static Tree Barrier



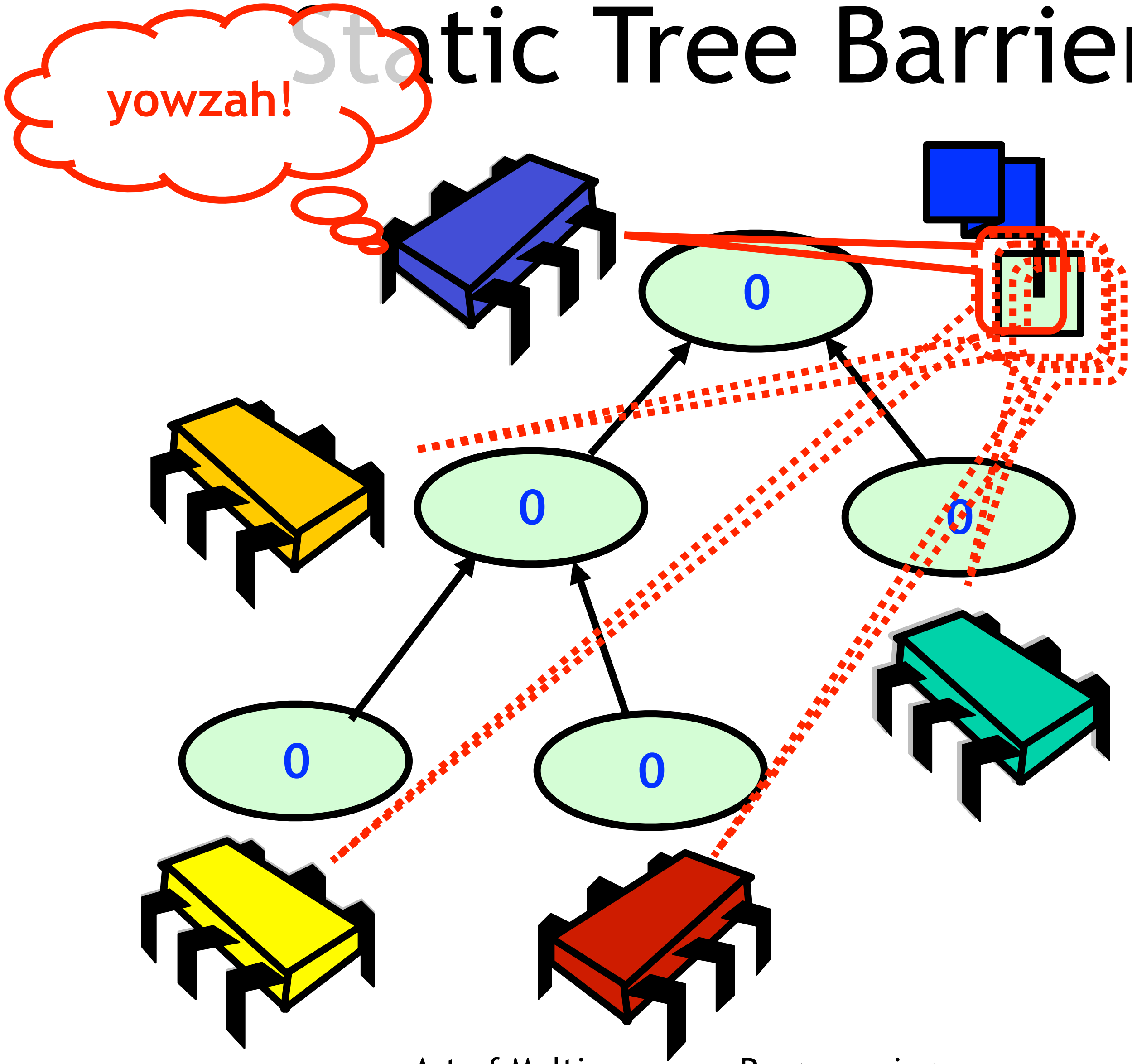
Static Tree Barrier



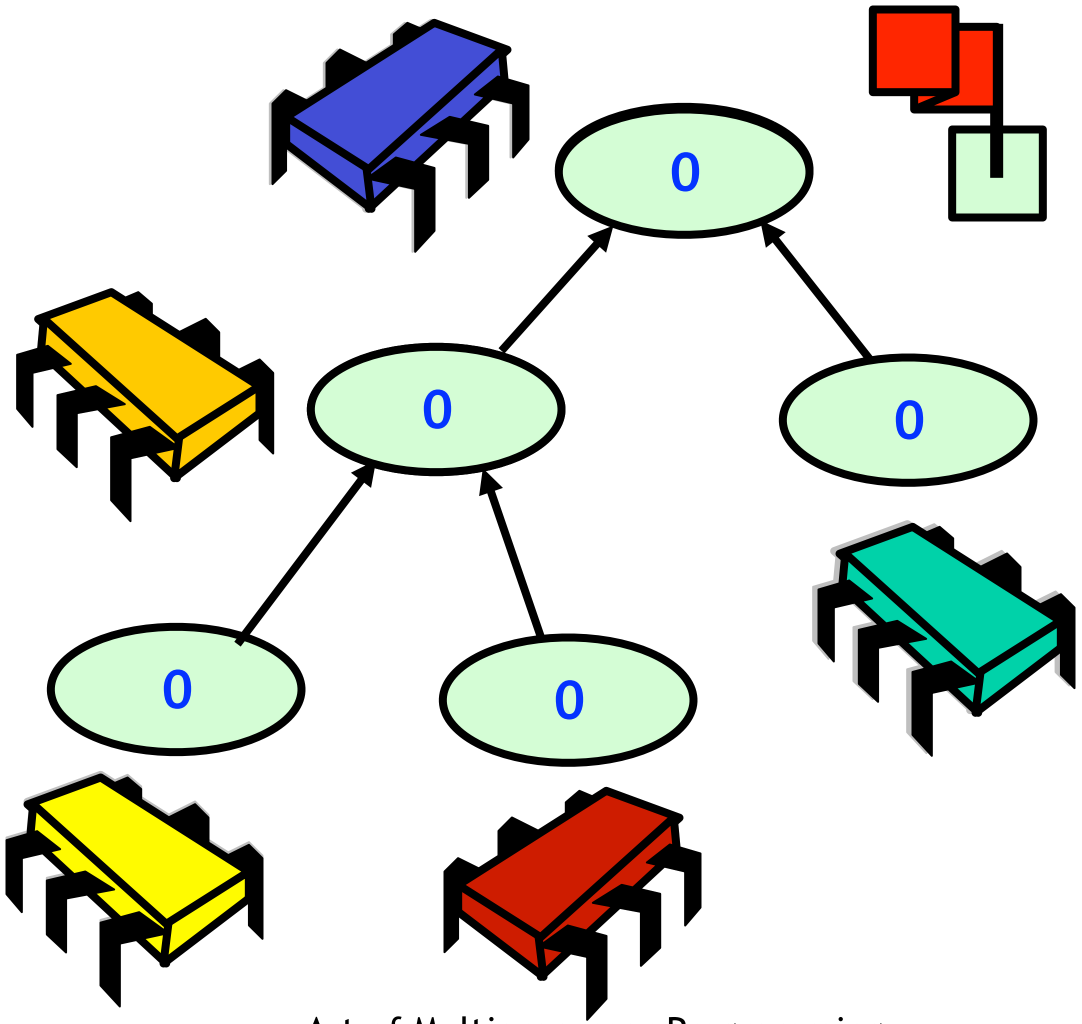
Static Tree Barrier



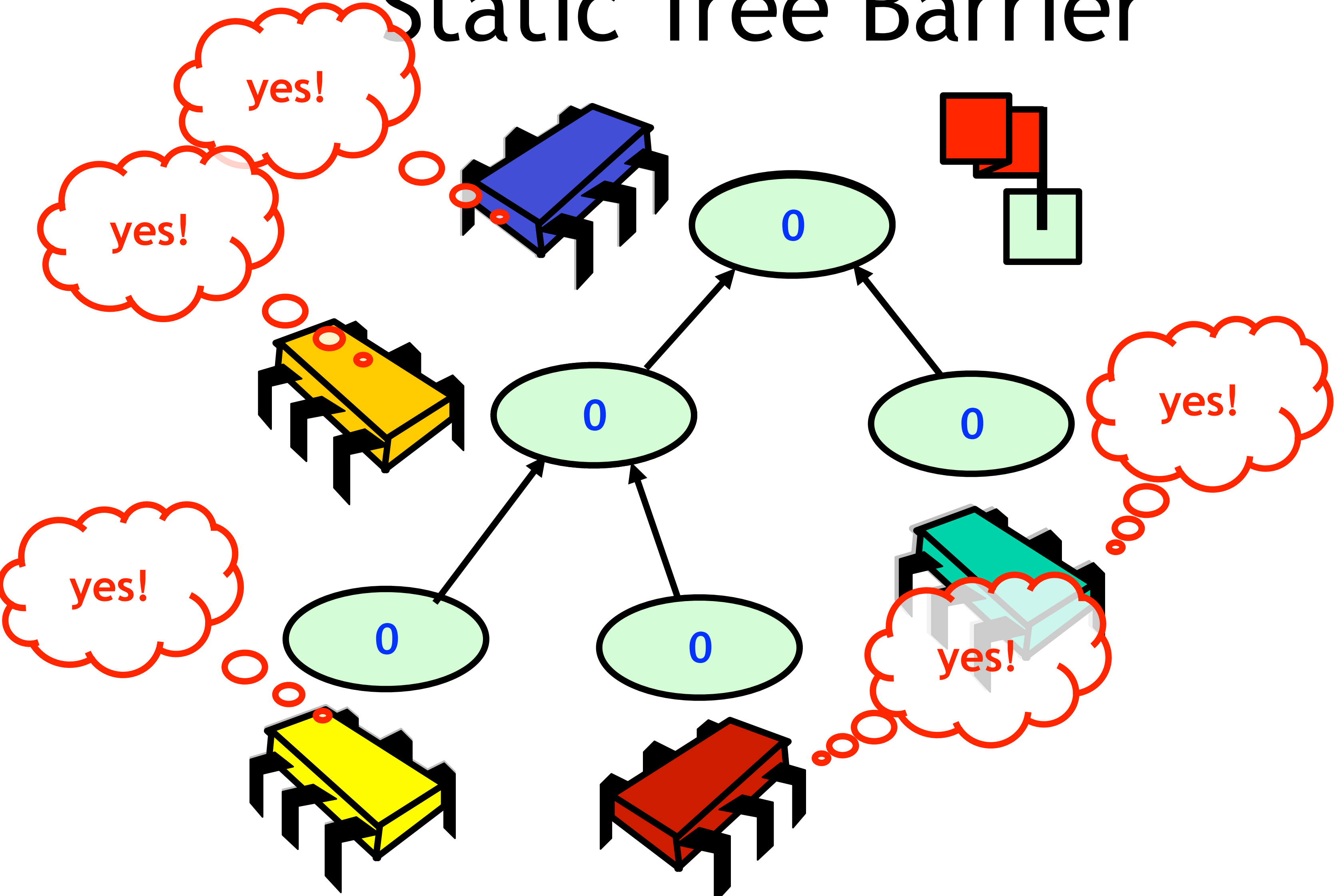
Static Tree Barrier



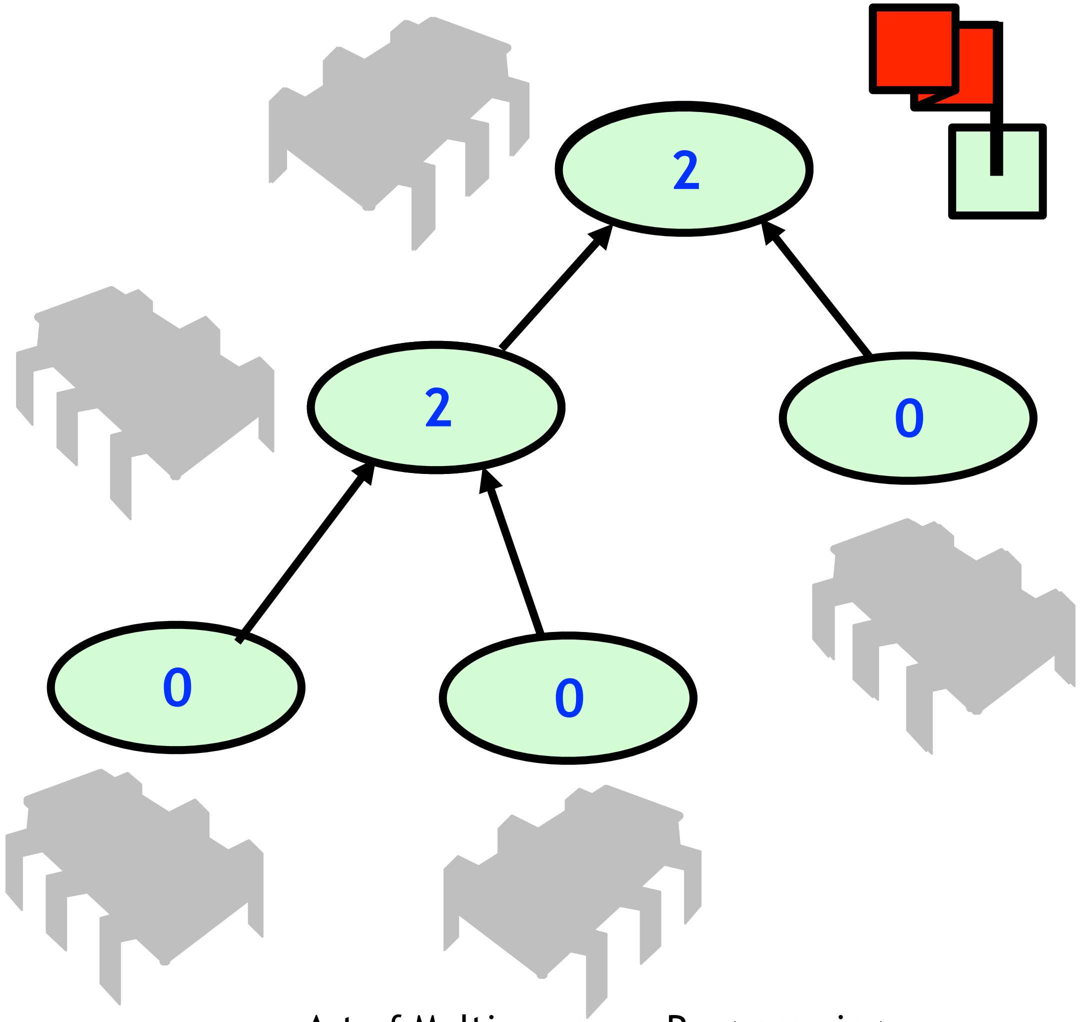
Static Tree Barrier



Static Tree Barrier



Static Tree Barrier



Remarks

- Very little cache traffic
- Minimal space overhead
- On message-passing architecture
  - Send notification & sense down tree