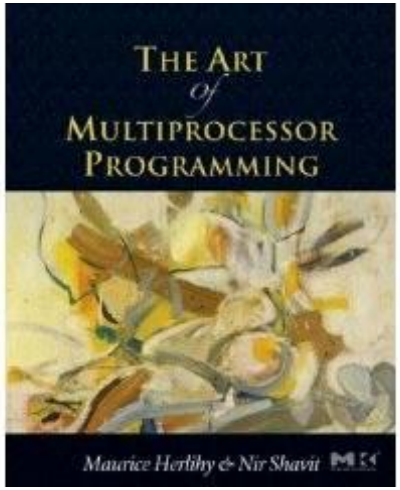
# Barrier Synchronization



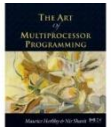

Companion slides for

The Art of Multiprocessor Programming

by Maurice Herlihy & Nir Shavit

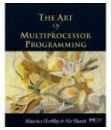
# Simple Video Game

- 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

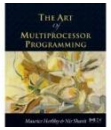
```
while (true) {
  frame.prepare();
  frame.display();
  }
```



# Simple Video Game

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

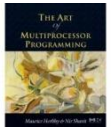
- What about overlapping work?
  - 1st thread displays frame
  - 2nd prepares next frame



# 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;
}
```

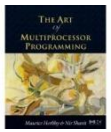


# Two-Phase Rendering

```
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**

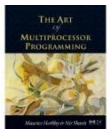


# Two-Phase Rendering

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

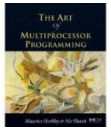
```
while (true) {
  if (phase) {
    frame[1].prepare();
  } else {
    frame[0].prepare();
  }
  phase = !phase;
}
```

**odd phases**

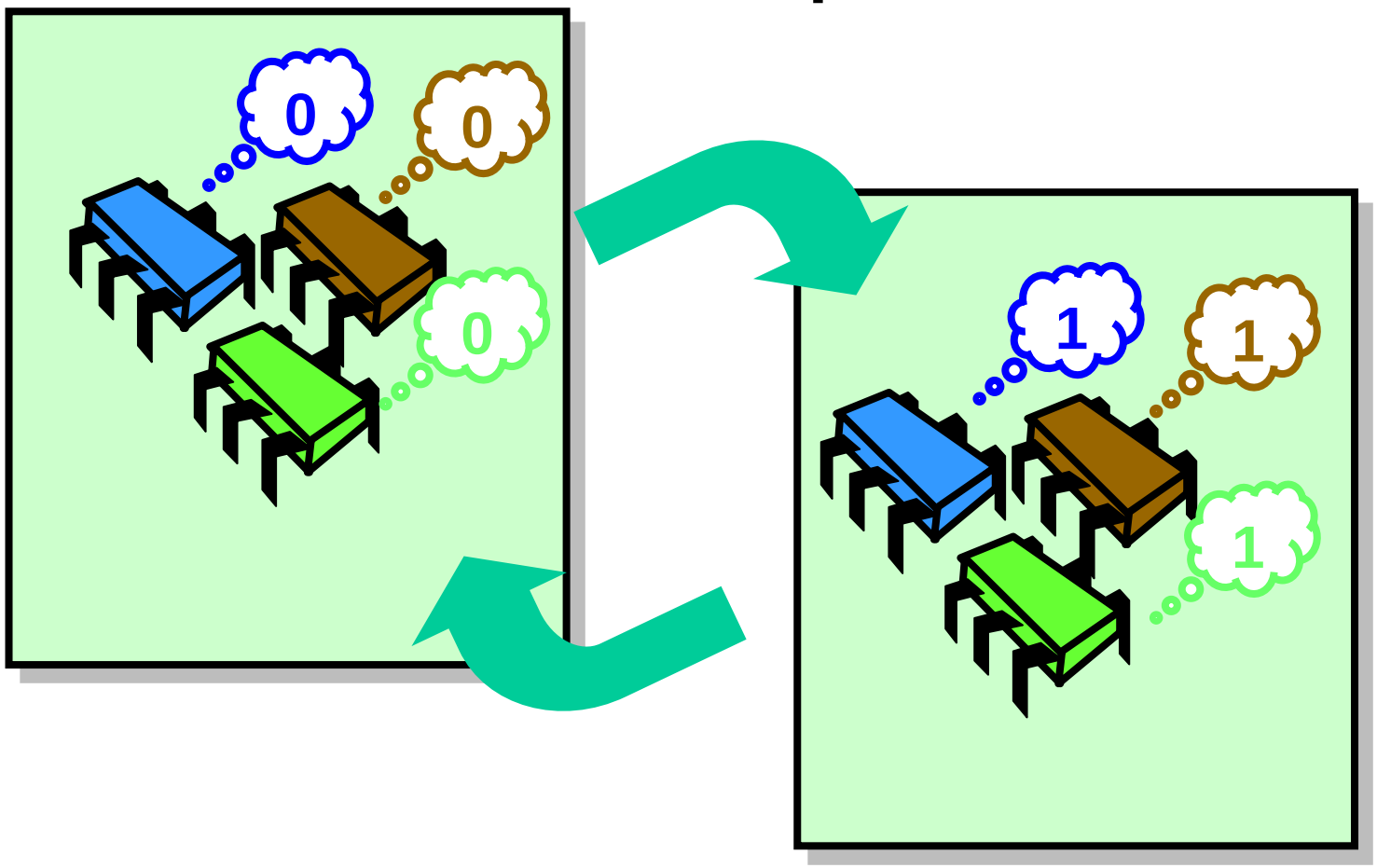


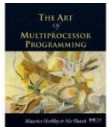
# Synchronization Problems

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

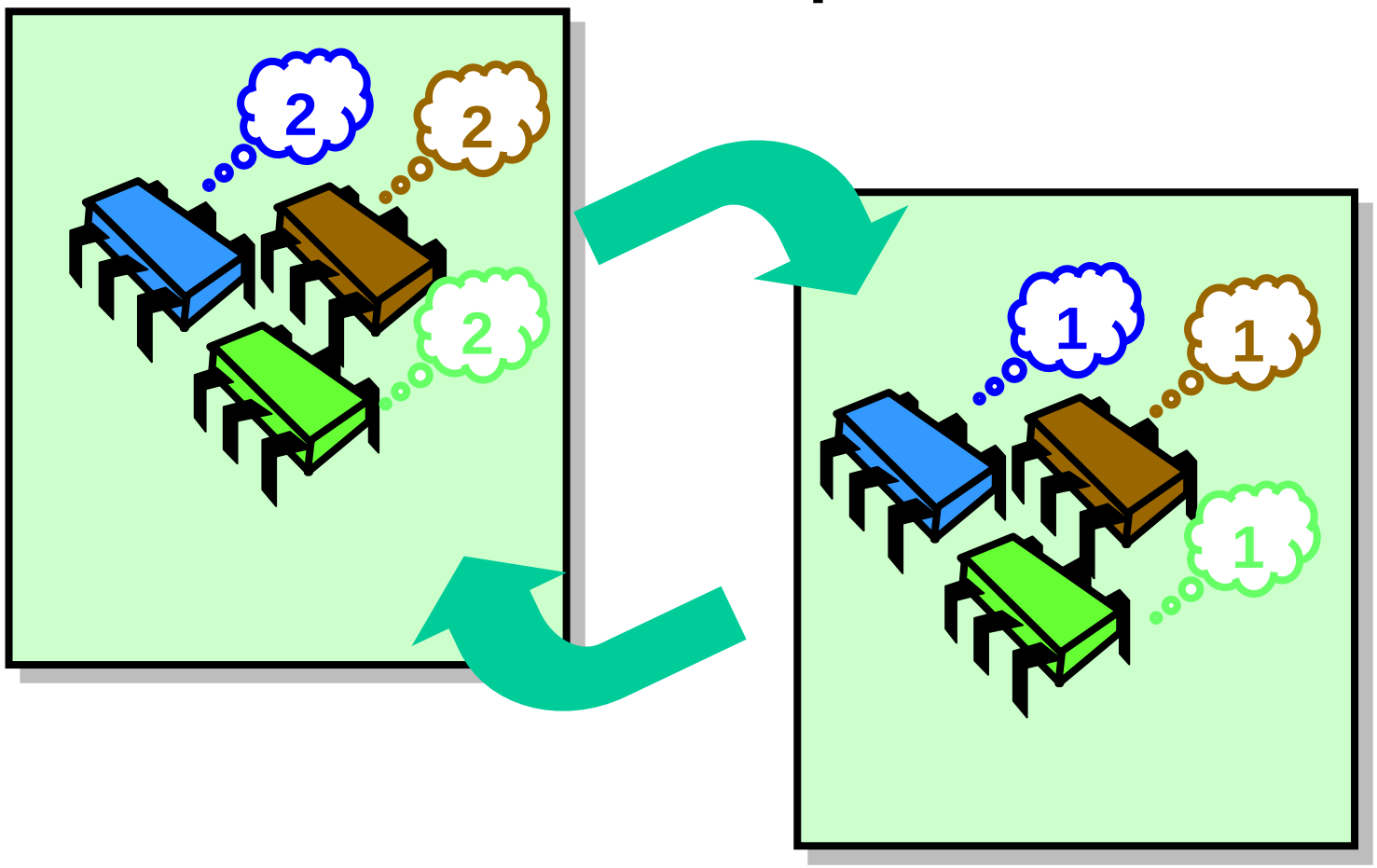


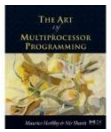
# Ideal Parallel Computation



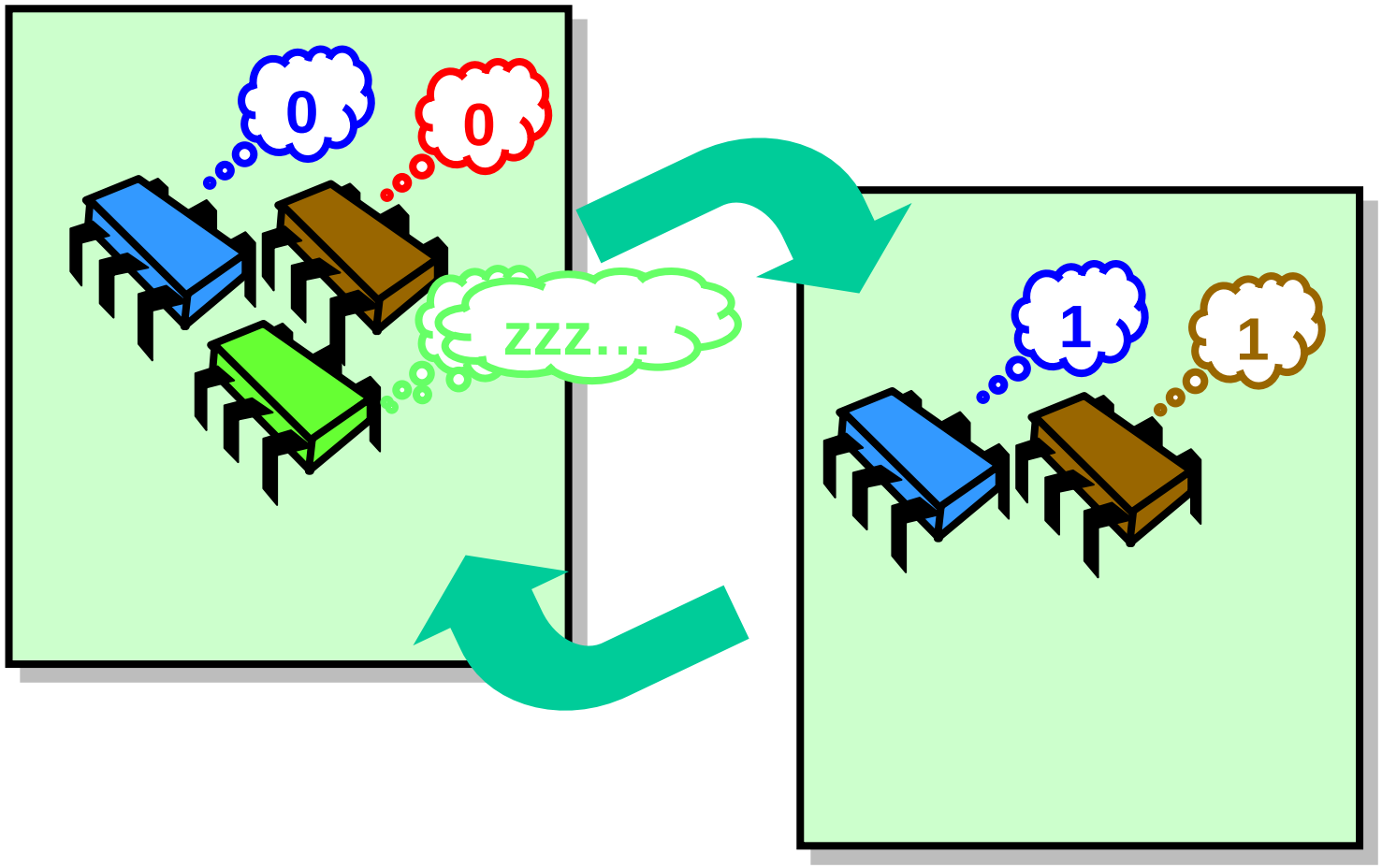



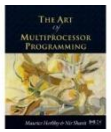
# Ideal Parallel Computation



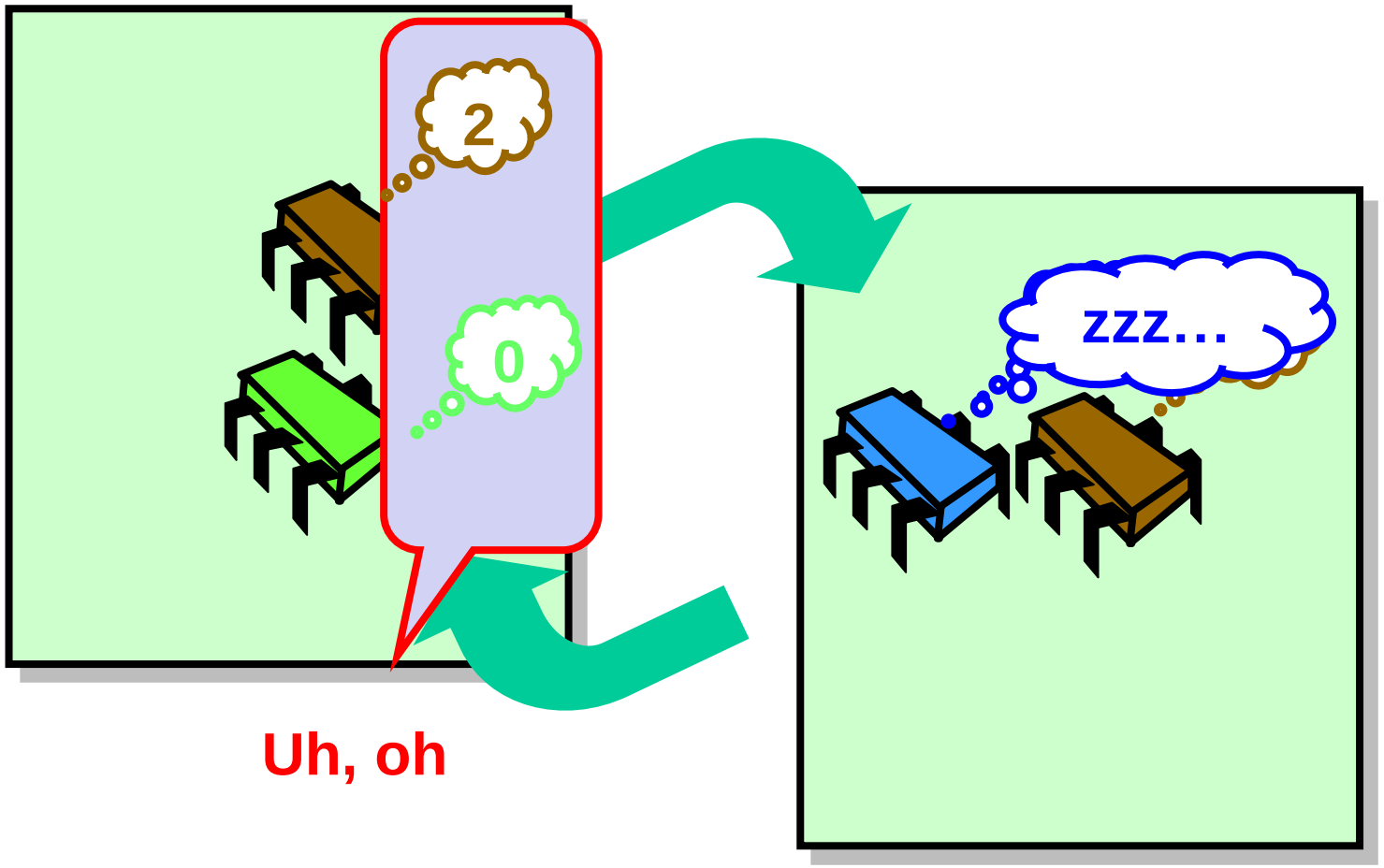



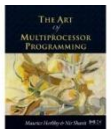
# Real-Life Parallel Computation



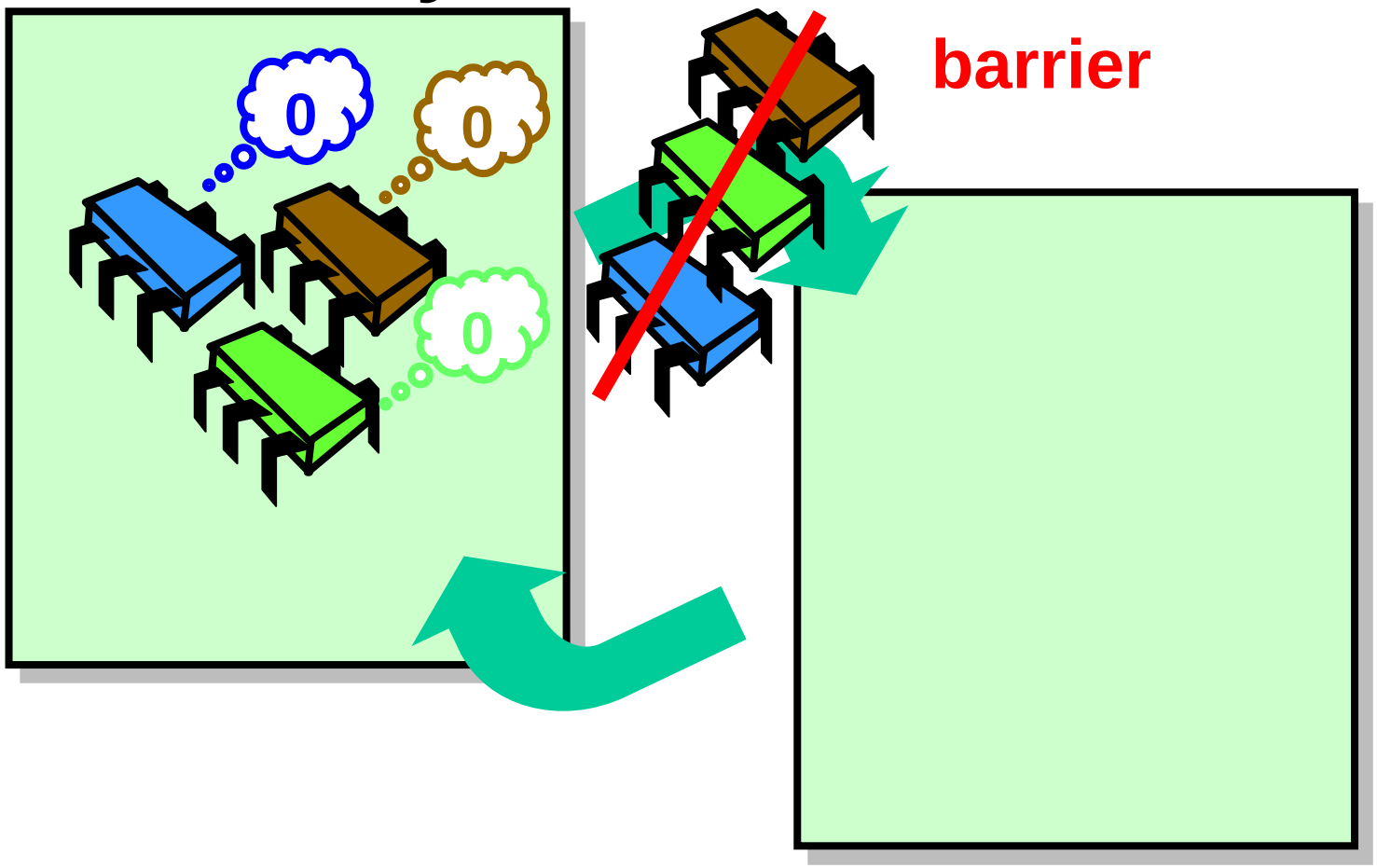



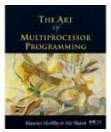
# Real-Life Parallel Computation



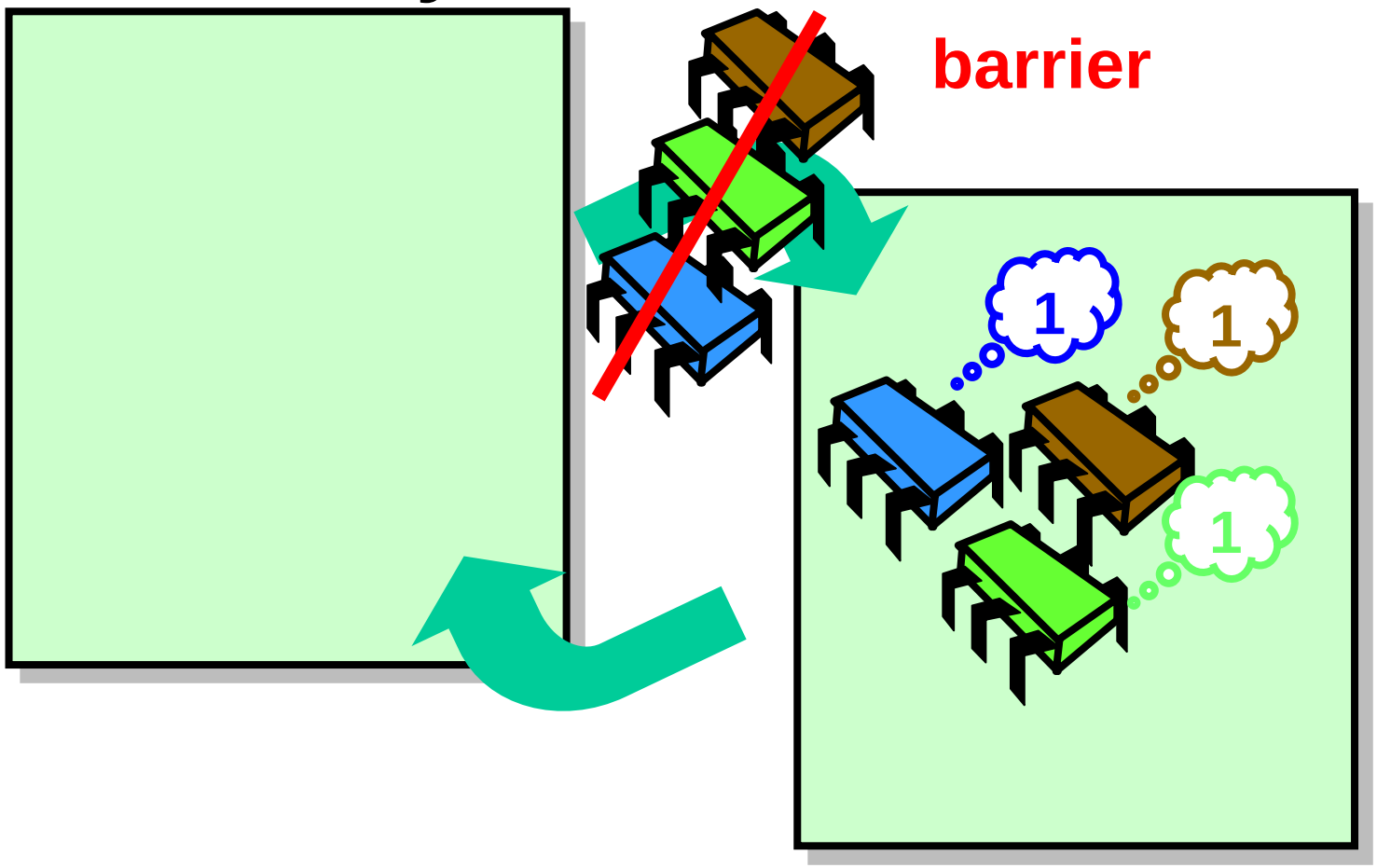



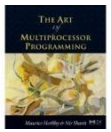
# Barrier Synchronization



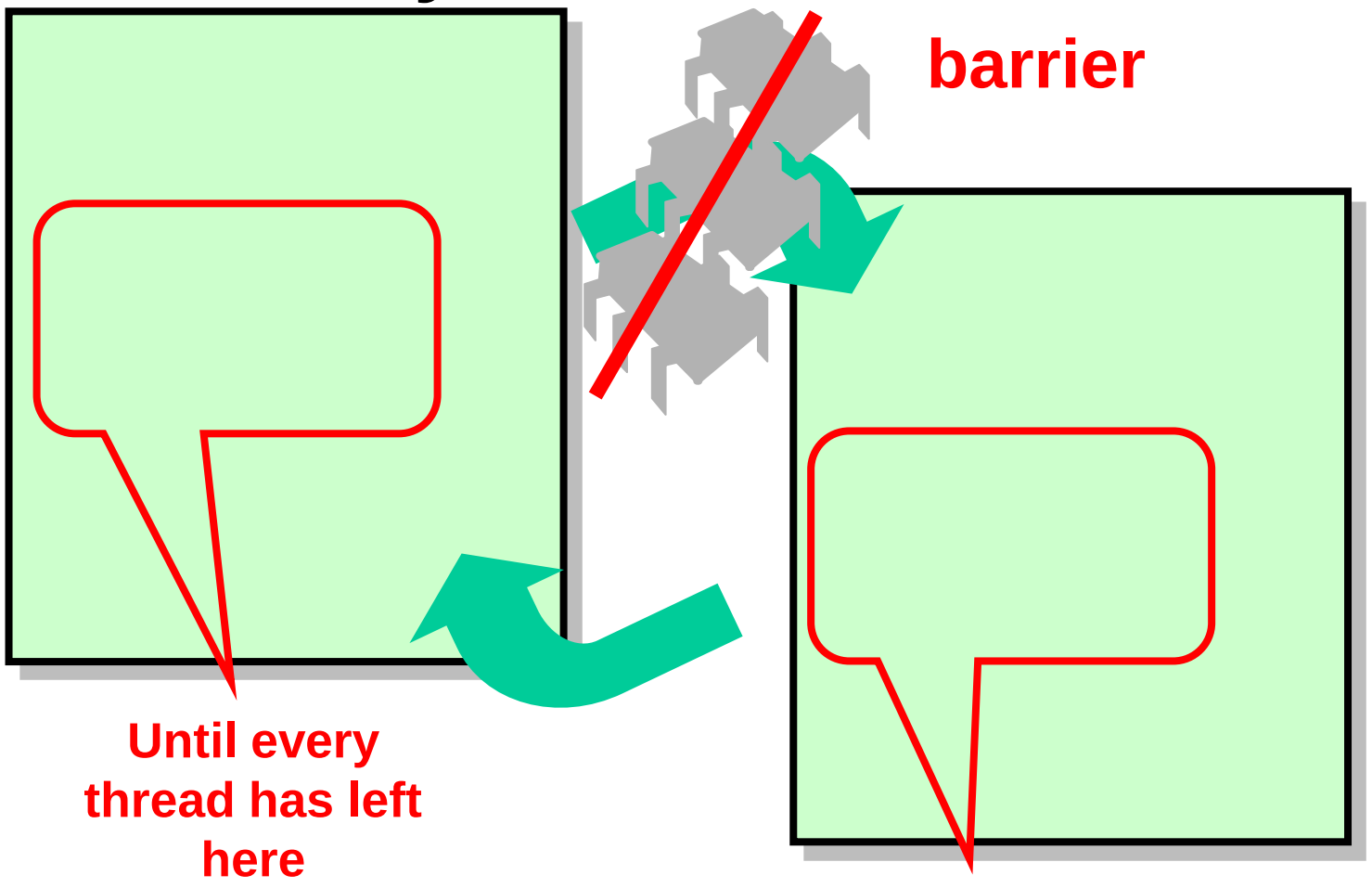



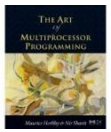
# Barrier Synchronization



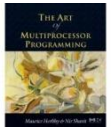
# Barrier Synchronization



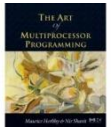
# Why Do We Care?

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

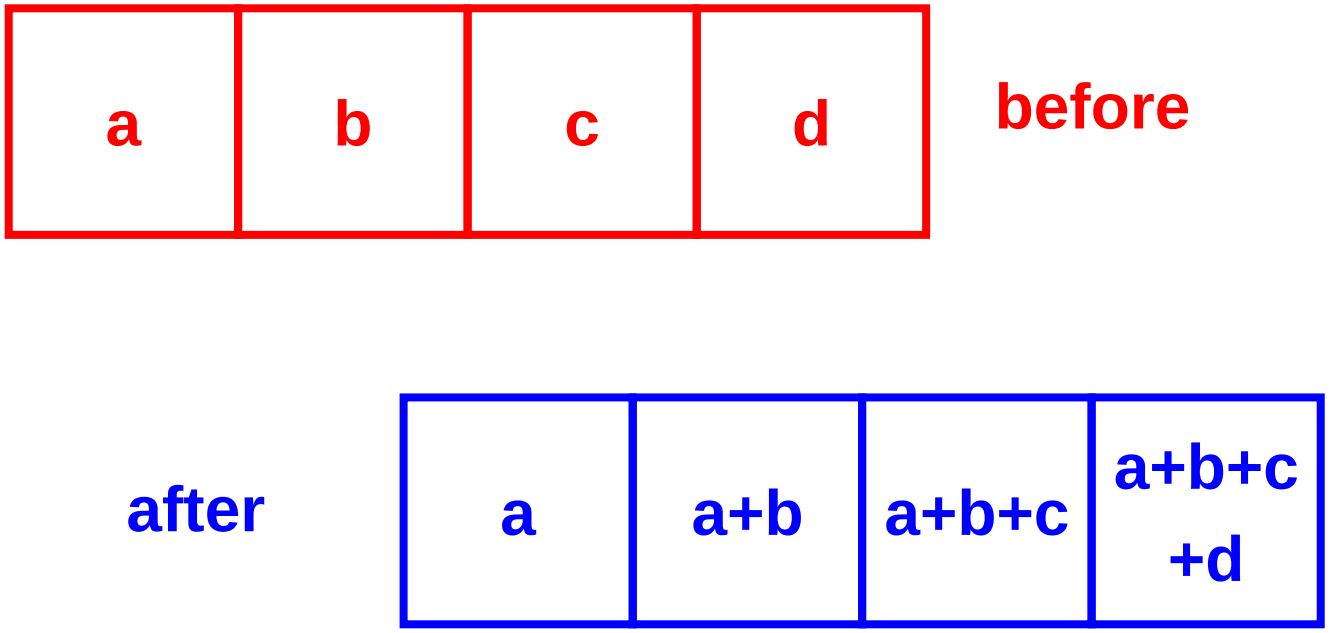


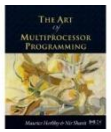
# Duality

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



# Example: Parallel Prefix

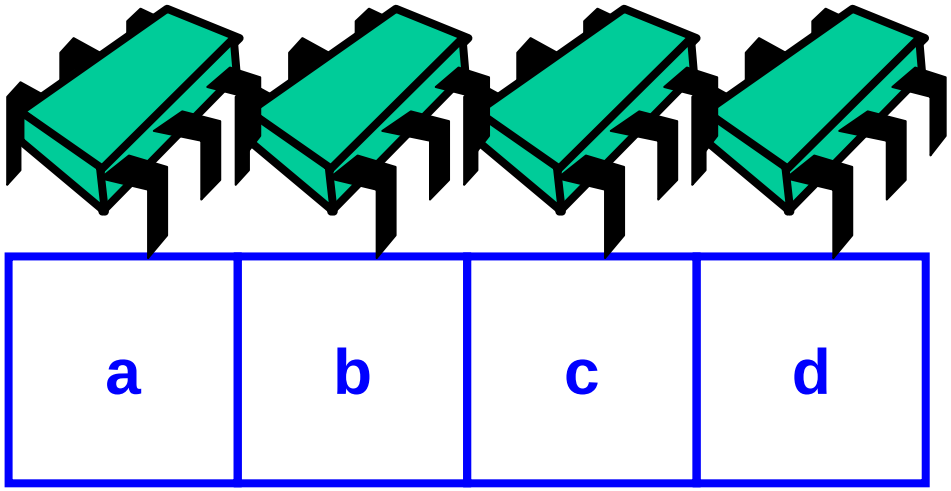


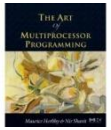



# Parallel Prefix

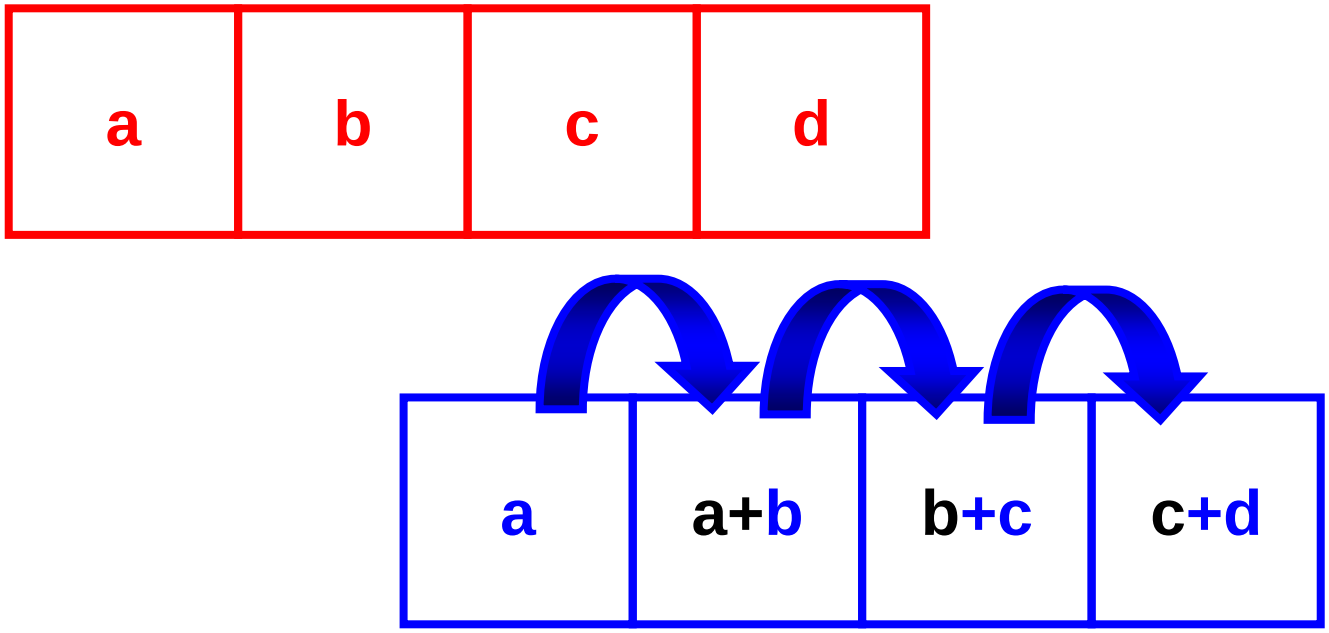
**One thread**

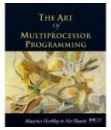
**Per entry**



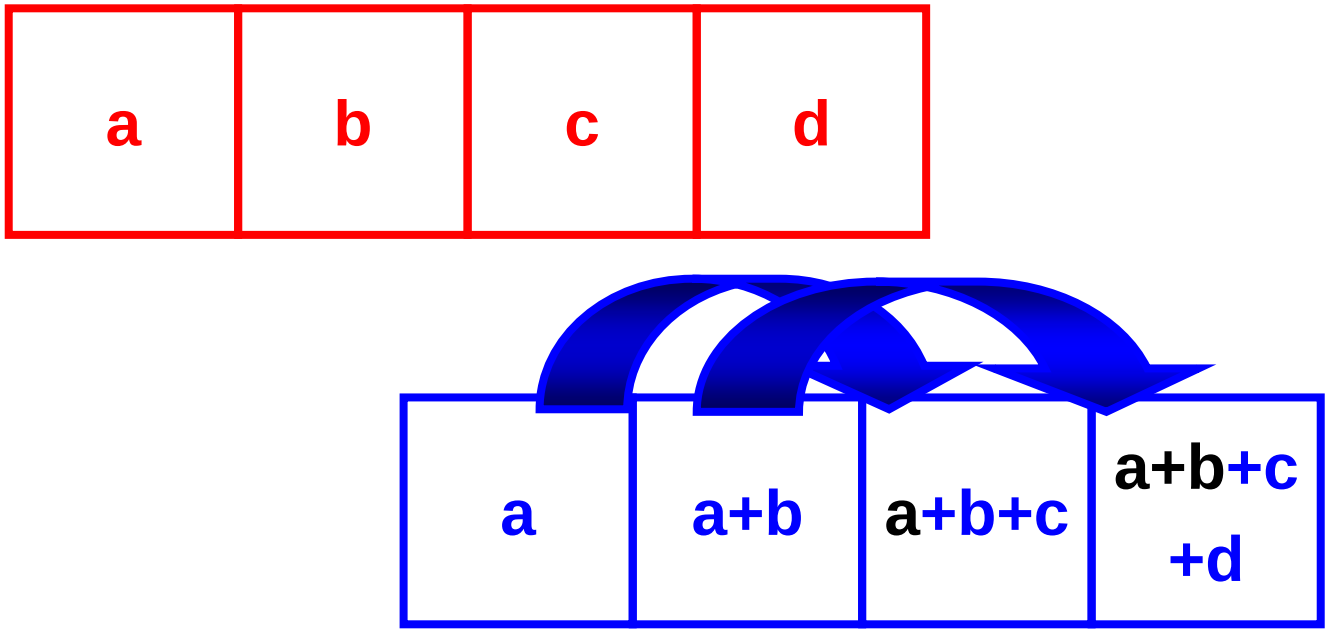



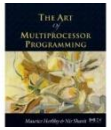
# Parallel Prefix: Phase 1



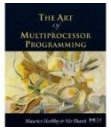
# Parallel Prefix: Phase 2



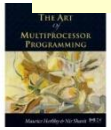
# Parallel Prefix

- N threads can compute
  - Parallel prefix
  - Of N entries
  - In $\log_2 N$ rounds
- What if system is asynchronous?
  - Why we need barriers



# Prefix

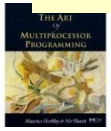
```
class Prefix extends Thread {
 int[] a;
 int i;
 Barrier b;
 void Prefix(int[] a,
             Barrier b, int i) {
  a = a;
  b = b;
  i = i;
 }
```



# Prefix

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

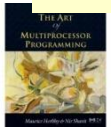
**Array of input values**



# Prefix

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

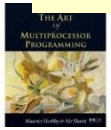
**Thread index**



# Prefix

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

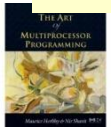
**Shared barrier**



# Prefix

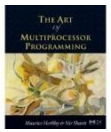
```
class Prefix extends Thread {
 int[] a;
 int i;
 Barrier b;
 void Prefix(int[] a
             Barrier b, int i) {
  a = a;
  b = b;
  i = i;
 }
```

**Initialize fields**



# Where Do the Barriers Go?

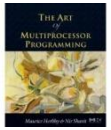
```
public void run() {
 int d = 1, sum = 0;
 while (d < N) {
  if (i >= d)
    sum = a[i-d];
  if (i >= d)
   a[i] += sum;
  d = d * 2;
}}}
```



# 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;
}}}
```

**Make sure everyone reads before anyone writes**

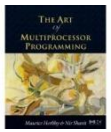


# Where Do the Barriers Go?

```
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;
}}}
```

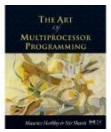
**Make sure everyone reads before anyone writes**

**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 of 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 of 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

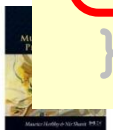
```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n){
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
 }}}}
```

**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**



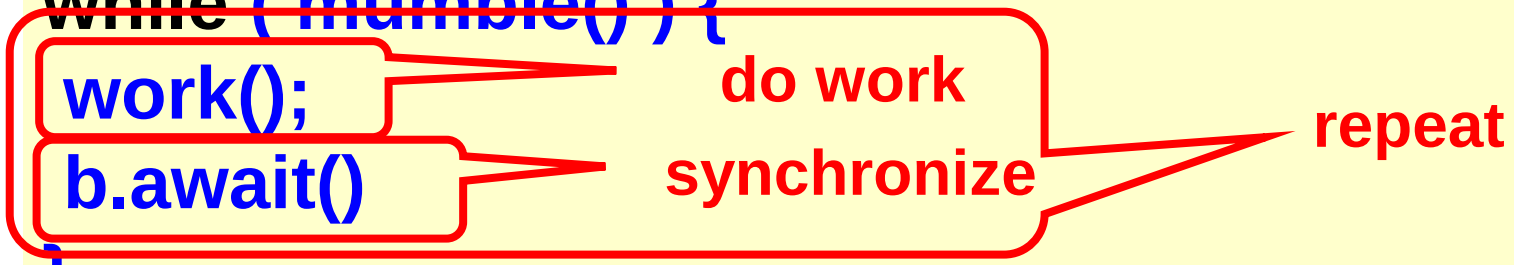
# Barriers

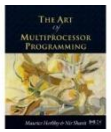
```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n){
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
 }}}}
```

What's wrong with this protocol?

# Reuse

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



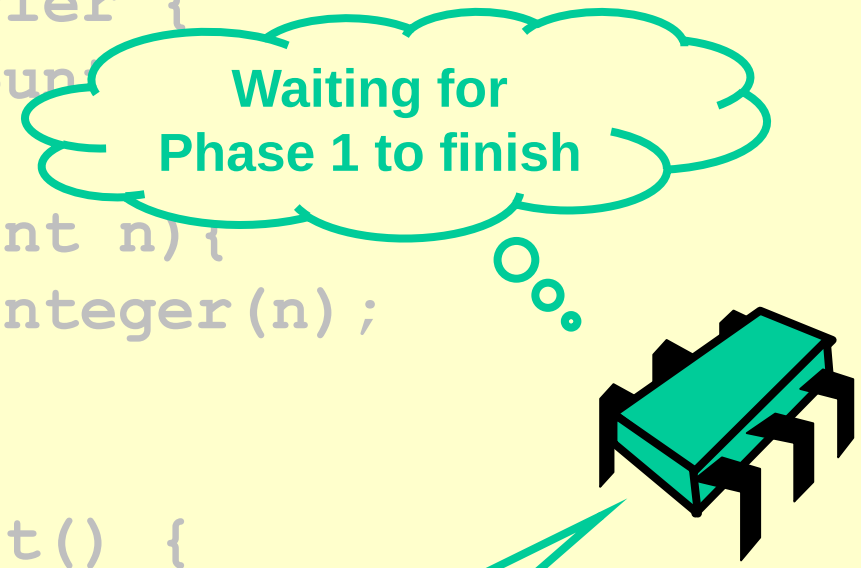



# Barriers

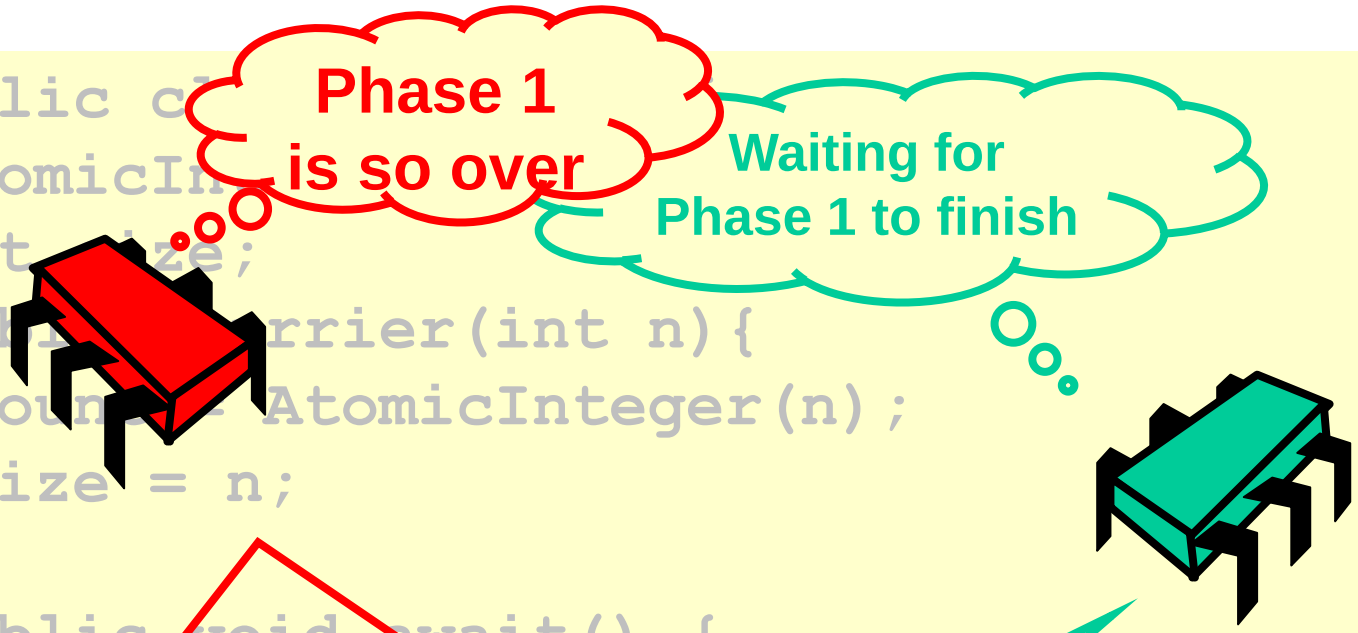
```
public class Barrier {
 AtomicInteger count;
 int size;
 public Barrier(int n){
  count = AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
 }}}}
```

# 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);
 }}}}
```

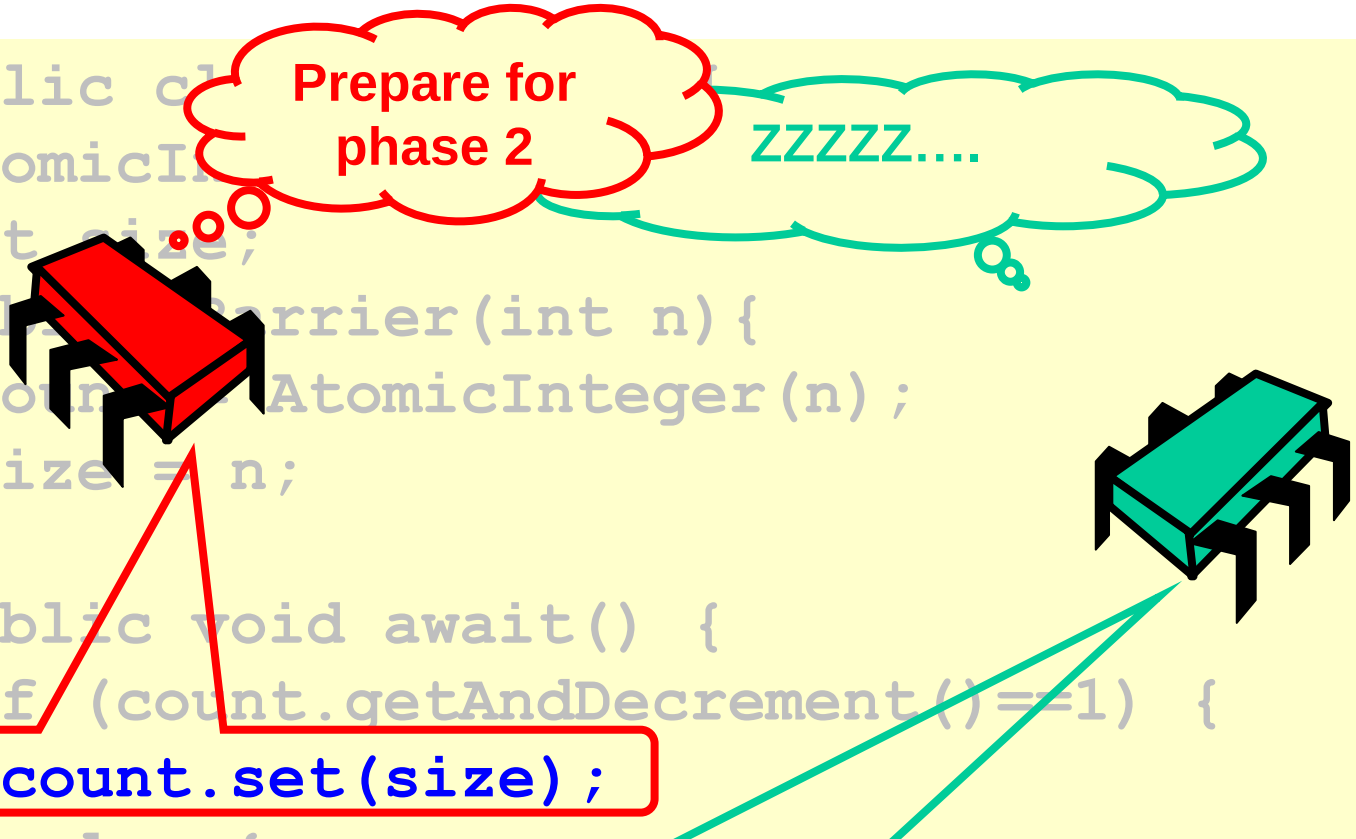



# Barriers



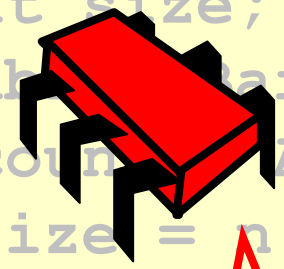

```
public cl
 AtomicIn
 int size;
 pub rrier(int n){
  coun AtomicInteger(n);
  size = n;
 }
 public void await() {
  if (count.getAndDecrement()==1) {
   count.set(size);
  } else {
   while (count.get() != 0);
 }}}}
```

# Barriers



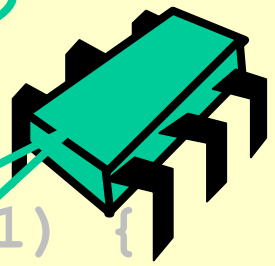
# Uh-Oh

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



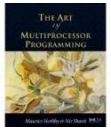
Waiting for
Phase 2 to finish

Waiting for
Phase 1 to finish



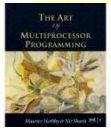
# Basic Problem

- 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

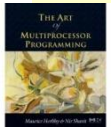
```
public class Barrier {
 AtomicInteger count;
 int size;
 volatile boolean sense = false;
 threadSense = new ThreadLocal<boolean>…

 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

**Completed odd or even-numbered phase?**

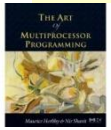
```
public class Barrier {
 AtomicInteger count;
 int size;
 volatile boolean sense = false;
 threadSense = new ThreadLocal<boolean>…

 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)}}}
```

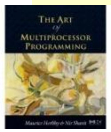
**Store sense for next phase**



# 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)}}}
```

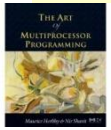
**Get new sense determined by last phase**



# 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)}}}
```

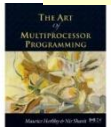
**If I'm last, reverse sense for next time**



# 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)}}}
```

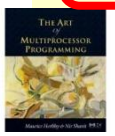
**Otherwise, wait for sense to flip**



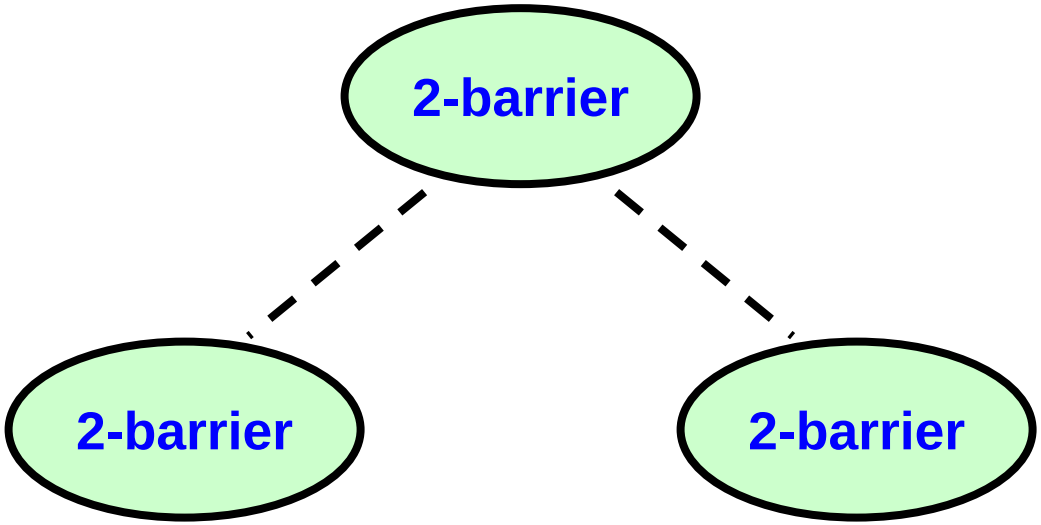
# Sense-Reversing Barriers

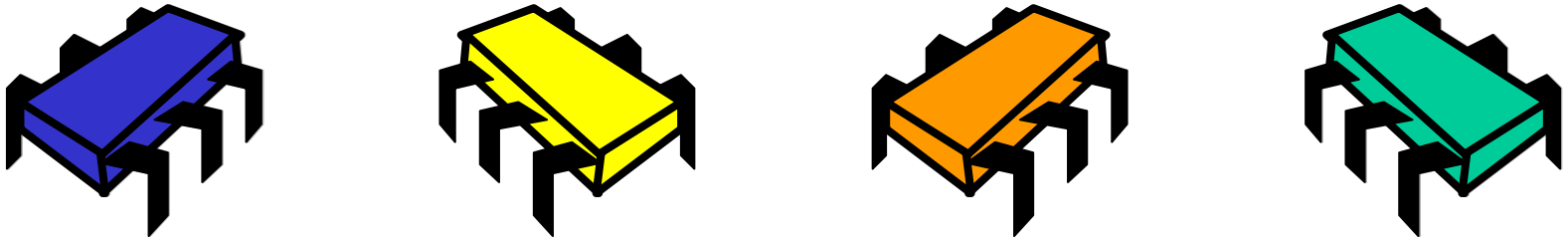
```
public class Barrier {
 AtomicInteger count;
 int size;
 volatile boolean sense = false;
 threadSense = new ThreadLocal<boolean>…

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

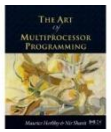
**Prepare sense for next phase**



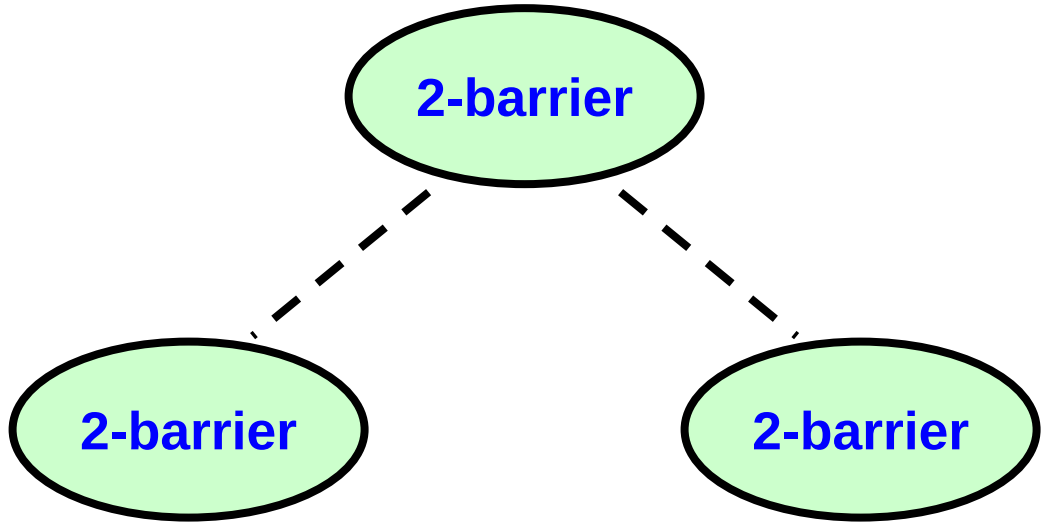
# Combining Tree Barriers

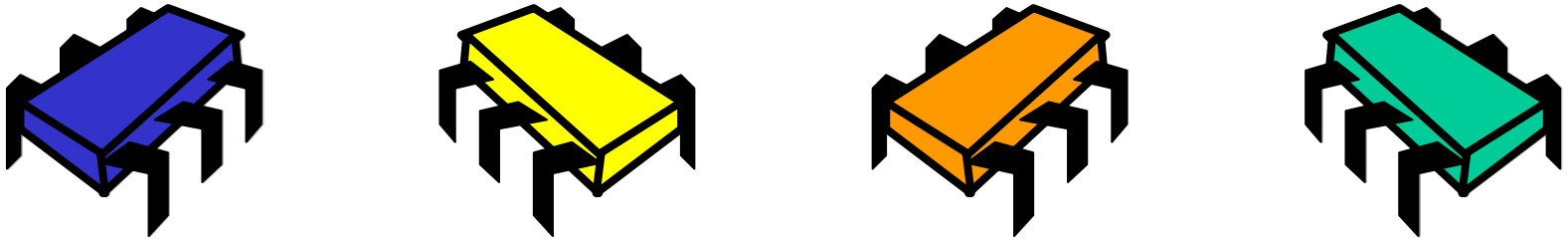


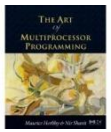





# Combining Tree Barriers

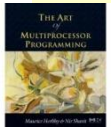




# Combining Tree Barrier

```
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

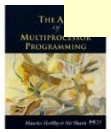
```
public class Node{
 AtomicInteger count; int size;
 Node parent; volatile boolean sense;

 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) {}
 }…}}}
```

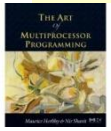
**Proceed to parent barrier**



# 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

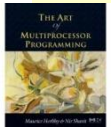
```
public class Node{
 AtomicInteger count; int size;
 Node parent; volatile boolean sense;

 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

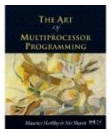
```
public class Node{
 AtomicInteger count; int size;
 Node parent; volatile boolean sense;

 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



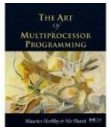
# Combining Tree Barrier

- No sequential bottleneck
  - Parallel getAndDecrement() calls
- Low memory contention
  - Same reason
- Cache behavior
  - Local spinning on bus-based architecture
  - Not so good for NUMA



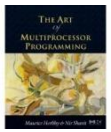
# Remarks

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

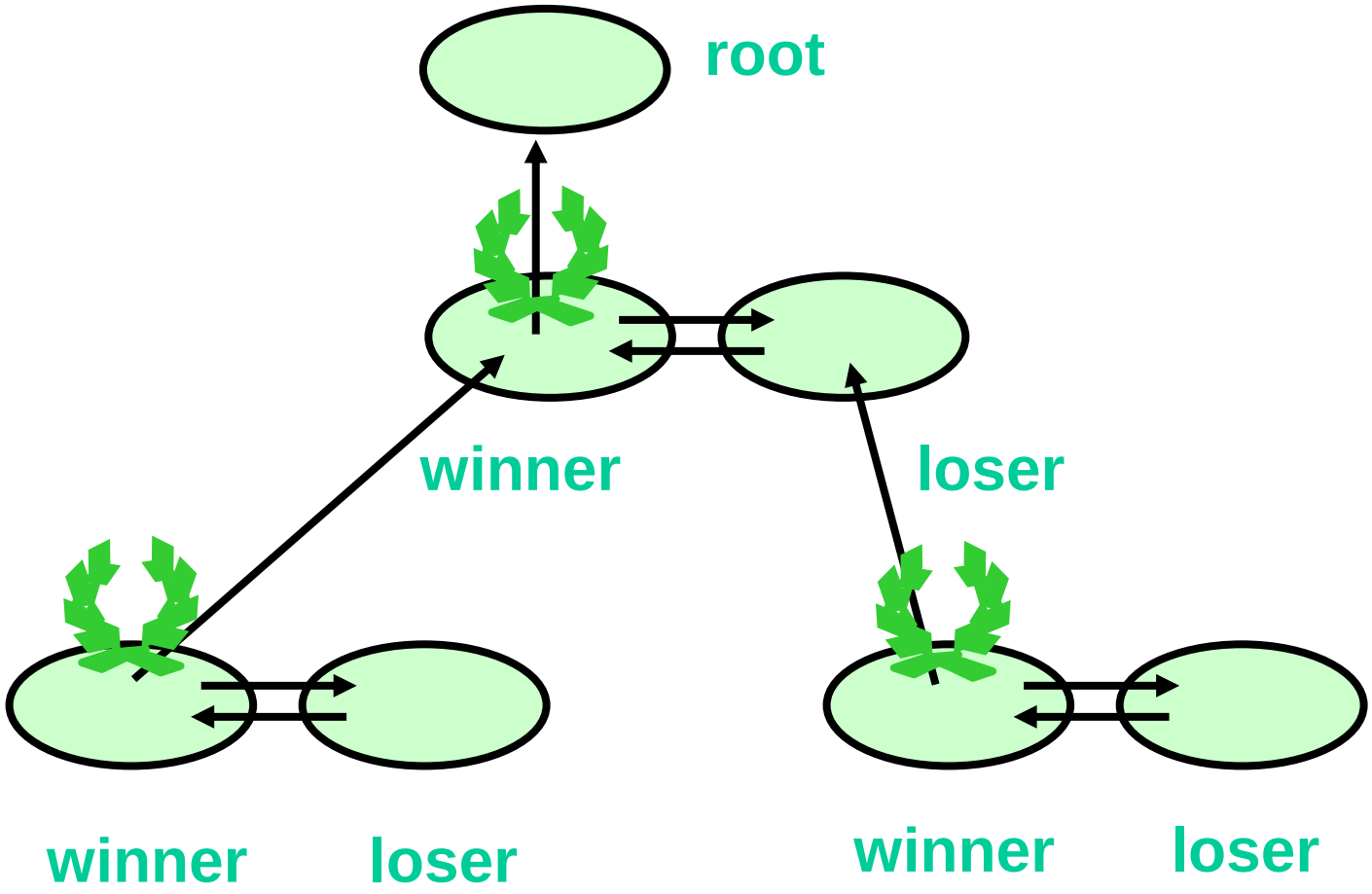


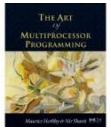
# Tournament Tree Barrier

- 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

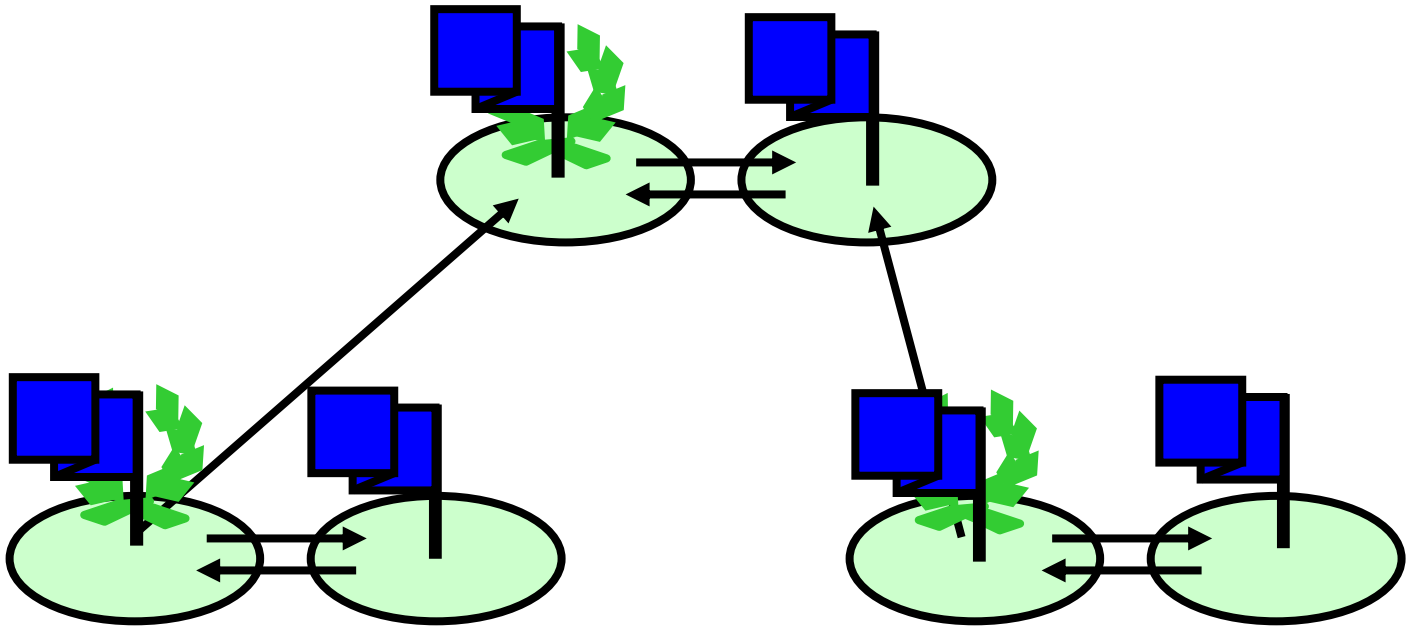


# Tournament Tree Barriers

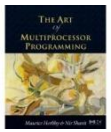


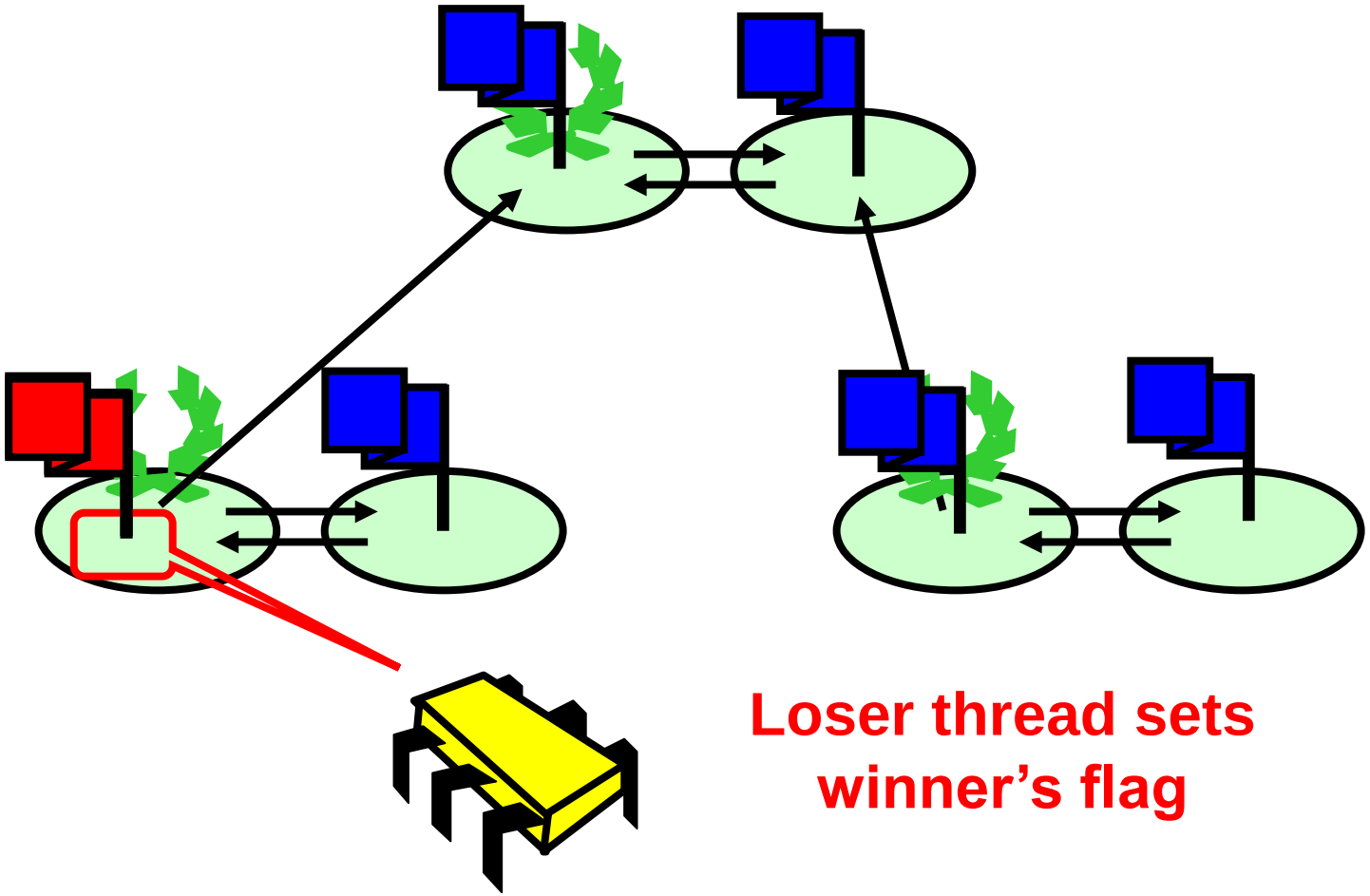
# Tournament Tree Barriers



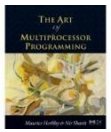
**All flags blue**



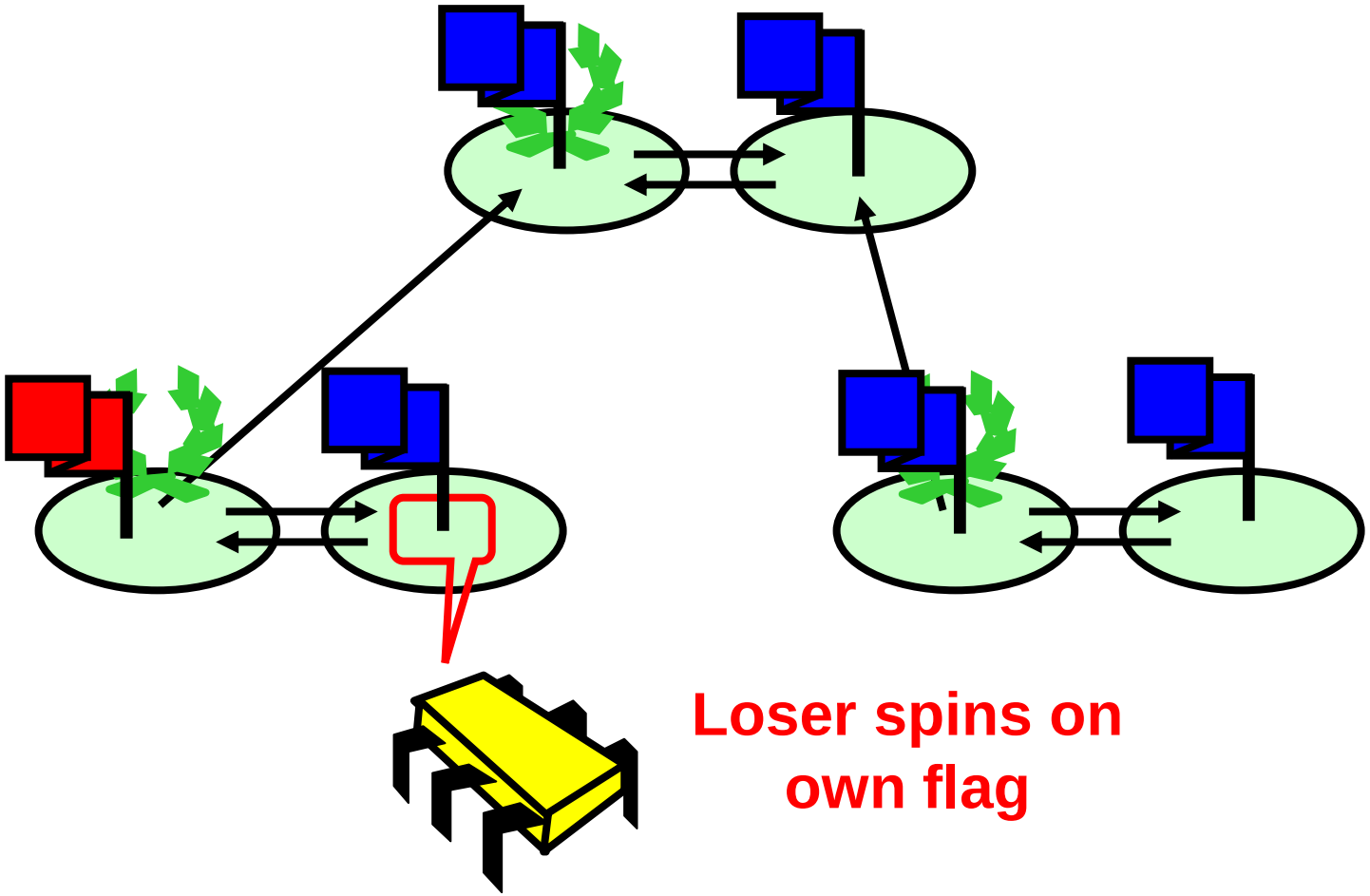
# Tournament Tree Barriers



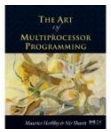
**Loser thread sets winner's flag**



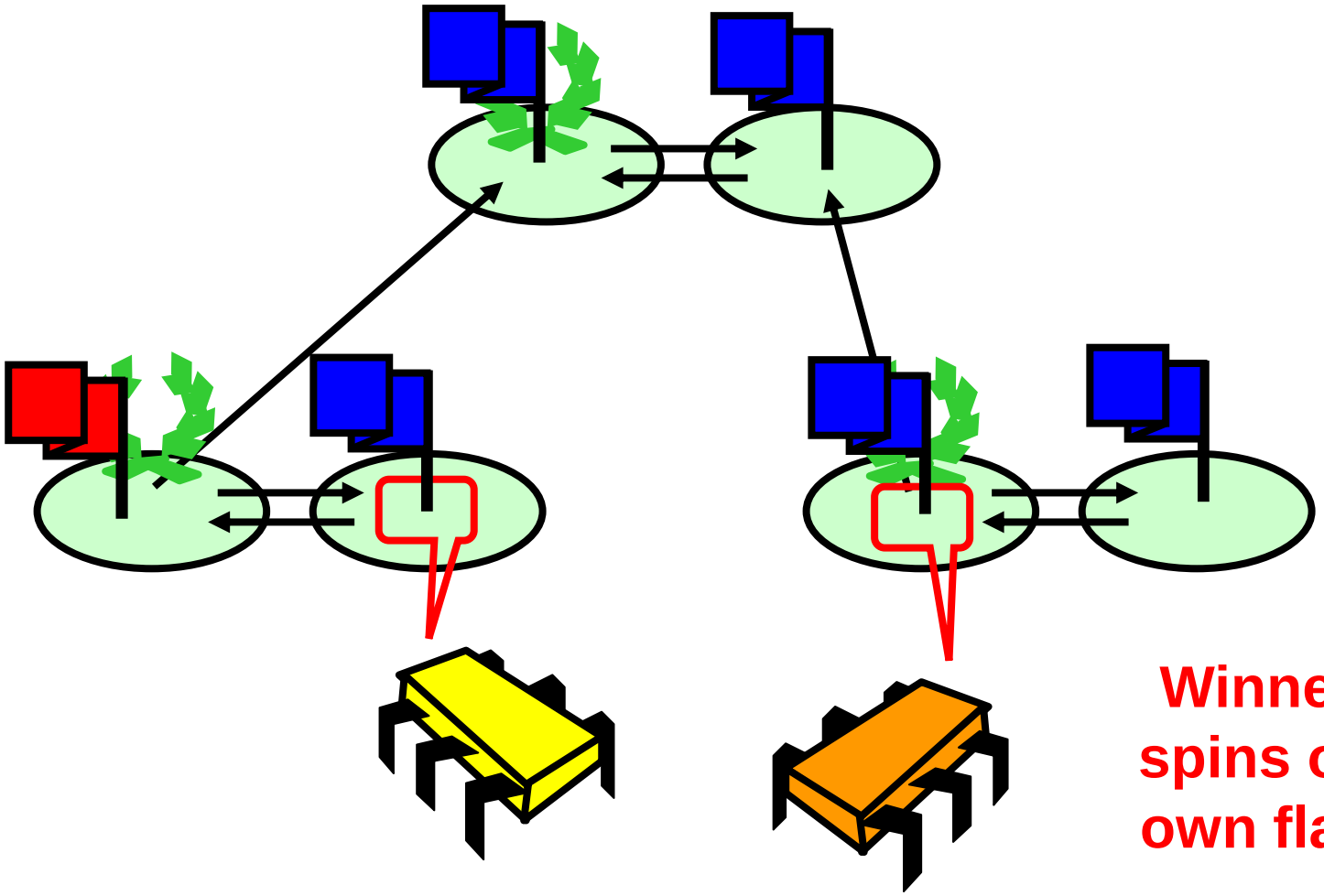
# Tournament Tree Barriers



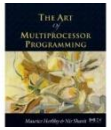
**Loser spins on own flag**



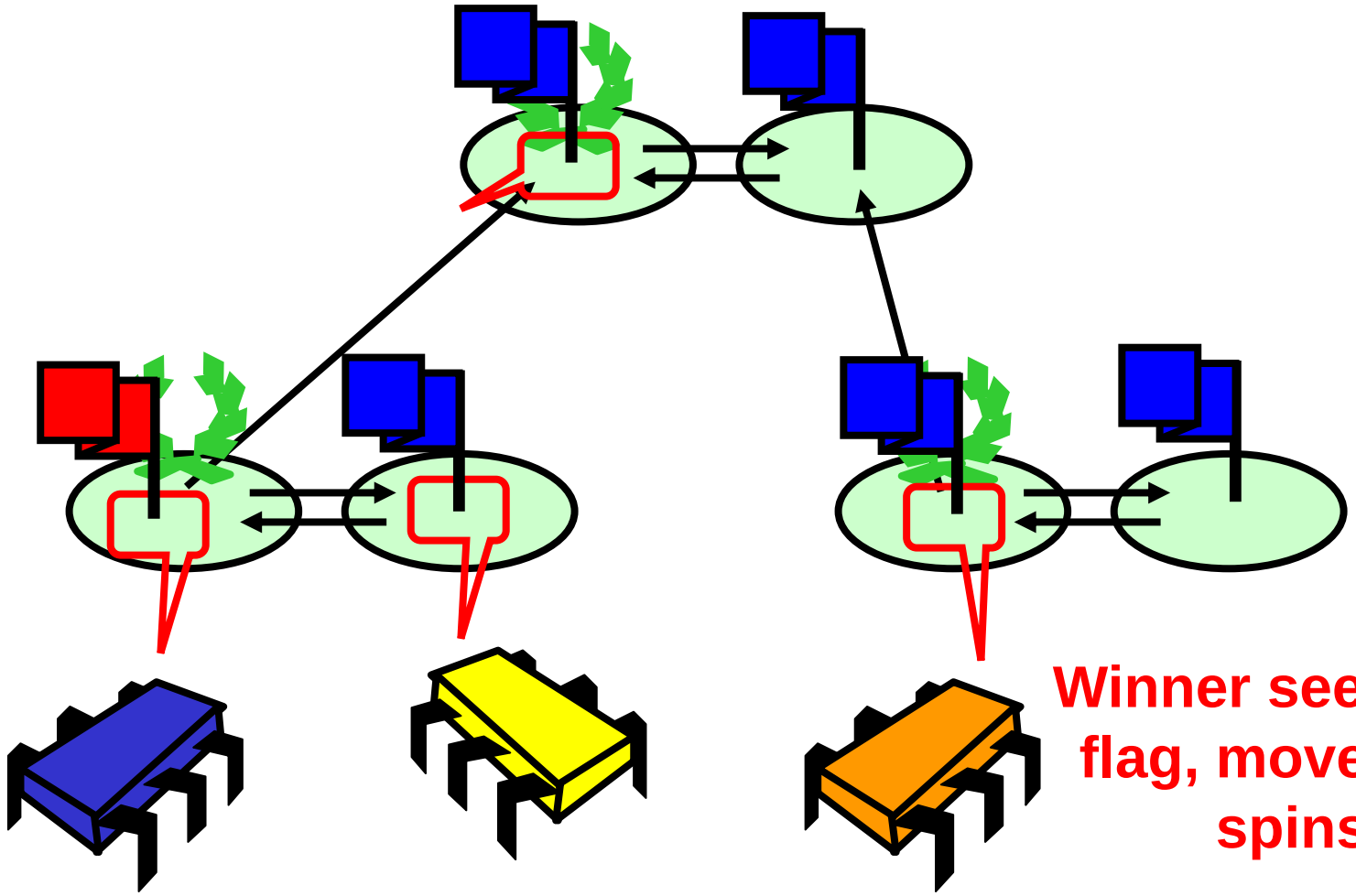
# Tournament Tree Barriers



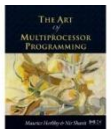
**Winner spins on own flag**



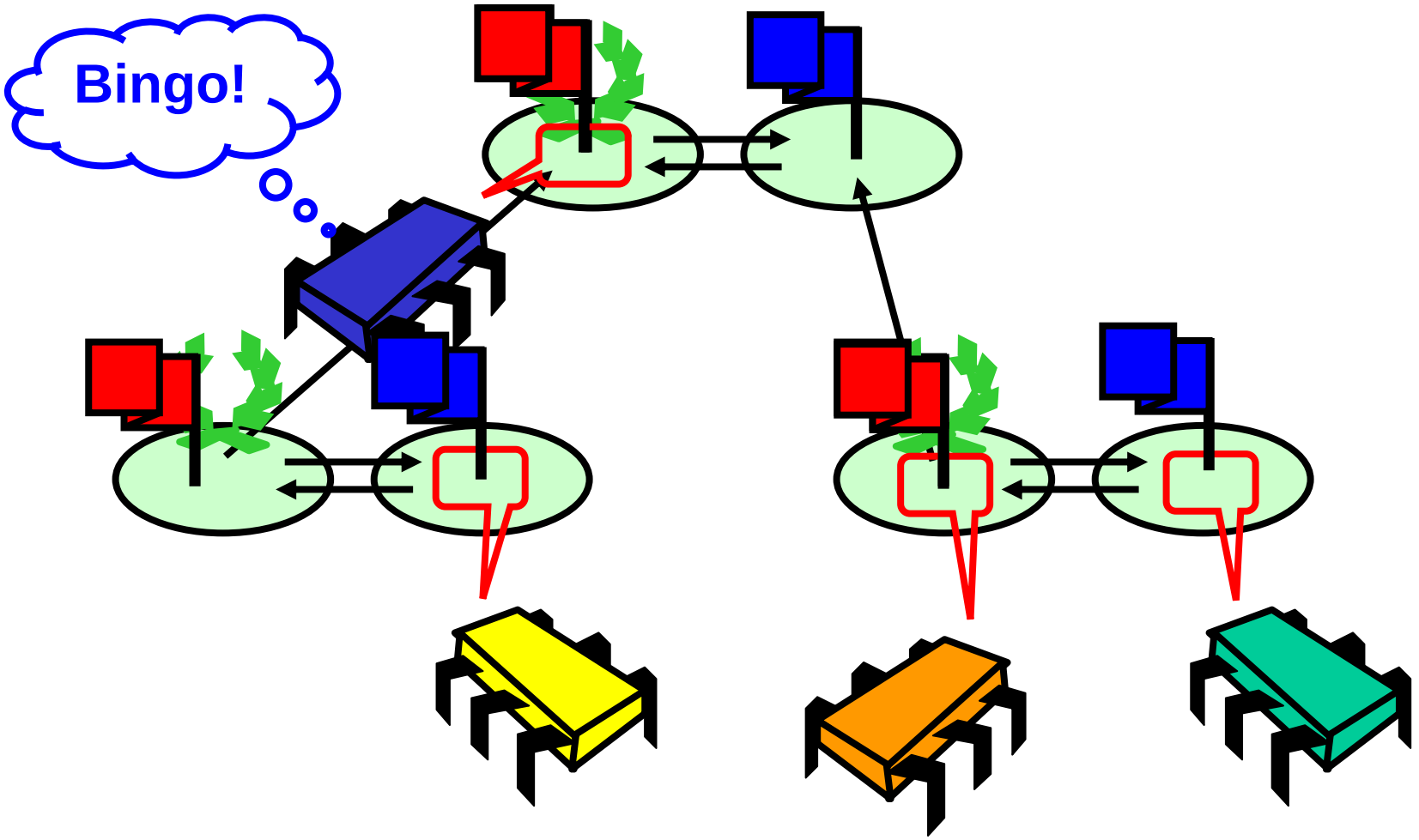
# Tournament Tree Barriers

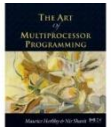


**Winner sees own flag, moves up, spins**

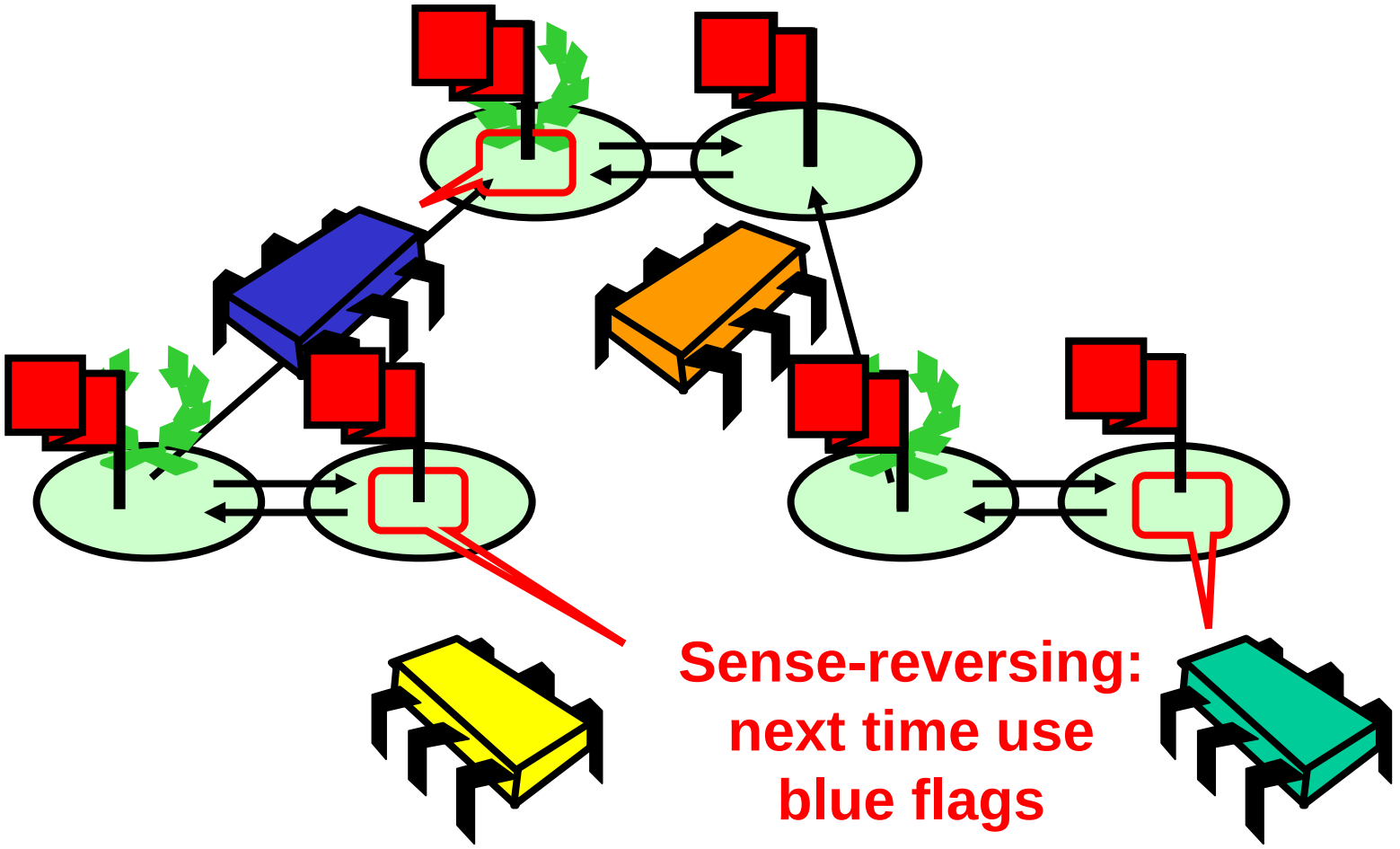


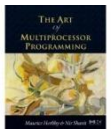
# Tournament Tree Barriers



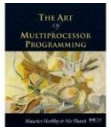
# Tournament Tree Barriers



# Tournament Barrier

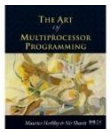
```
class TBarrier {
  volatile boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```



# Tournament Barrier

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

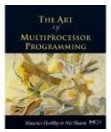
**Notifications delivered here**



# Tournament Barrier

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

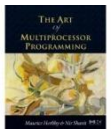
**Other thead at same level**



# Tournament Barrier

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

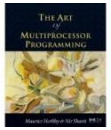
**Parent (winner) or null (loser)**



# Tournament Barrier

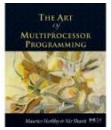
```
class TBarrier {
  volatile boolean flag;
  TBarrier partner;
  TBarrier parent;
  boolean top;
  …
}
```

**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) {};
}}}
```

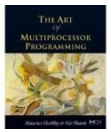


# Tournament Barrier

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

Current sense

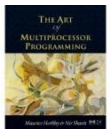
Le root, c'est moi



# 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) {};
}}}
```

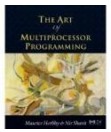
I am already a winner



# 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) {};
}}}
```

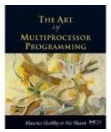
**Wait for partner**



# 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) {};
}}}
```

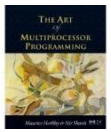
**Synchronize upstairs**



# 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**

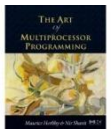


# Tournament Barrier

```
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**

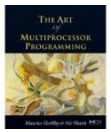
**Order is important (why?)**



# 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) {};
}}}
```

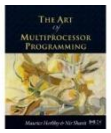
**Natural-born loser**



# 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) {};
}}}
```

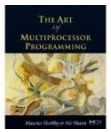
Tell partner I'm here



# Tournament Barrier

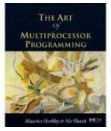
```
void await(boolean mySense) {
 if (top) {
  return;
 } else if (parent != null) {
  while (flag != mySense) {};
  parent.await(mySense);
  partner.flag = mySense;
 } else {
  partner.flag = mySense;
  while (flag != mySense) {};
}}}
```

**Wait for notification from partner**



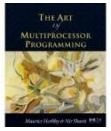
# Remarks

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

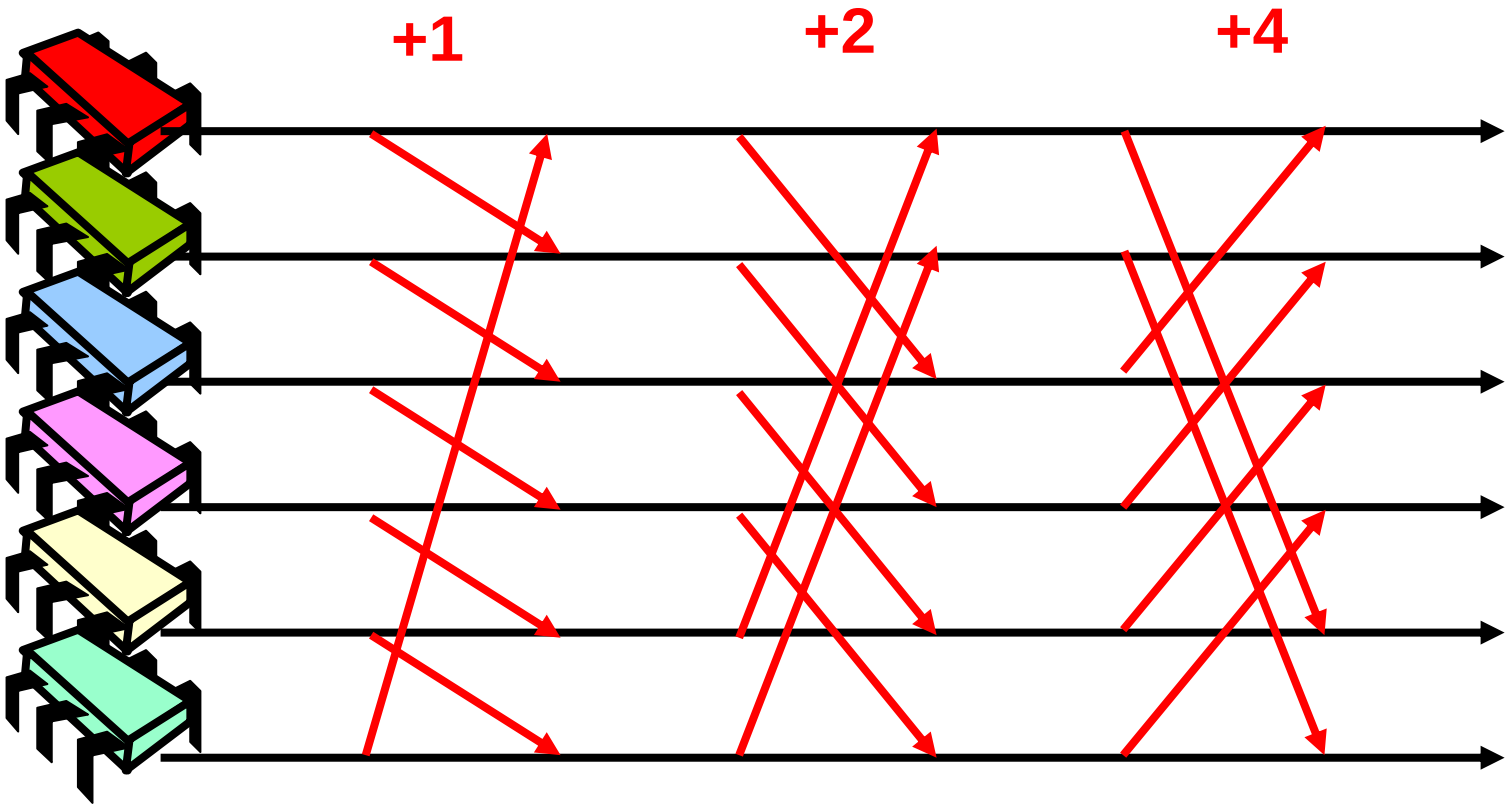


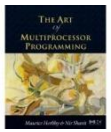
# Dissemination Barrier

- At round i
  - Thread A notifies thread $A+2^i$ (mod n)
- Requires log n rounds



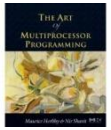
# Dissemination Barrier



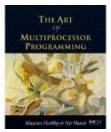
# Remarks

- Elegant
- Good source of homework problems
- Not cache-friendly



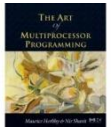
# Ideas So Far

- Sense-reversing
  - Reuse without reinitializing
- Combining tree
  - Like counters, locks …
- Tournament tree
  - Optimized combining tree
- Dissemination barrier
  - Intellectually Pleasing (matter of taste)

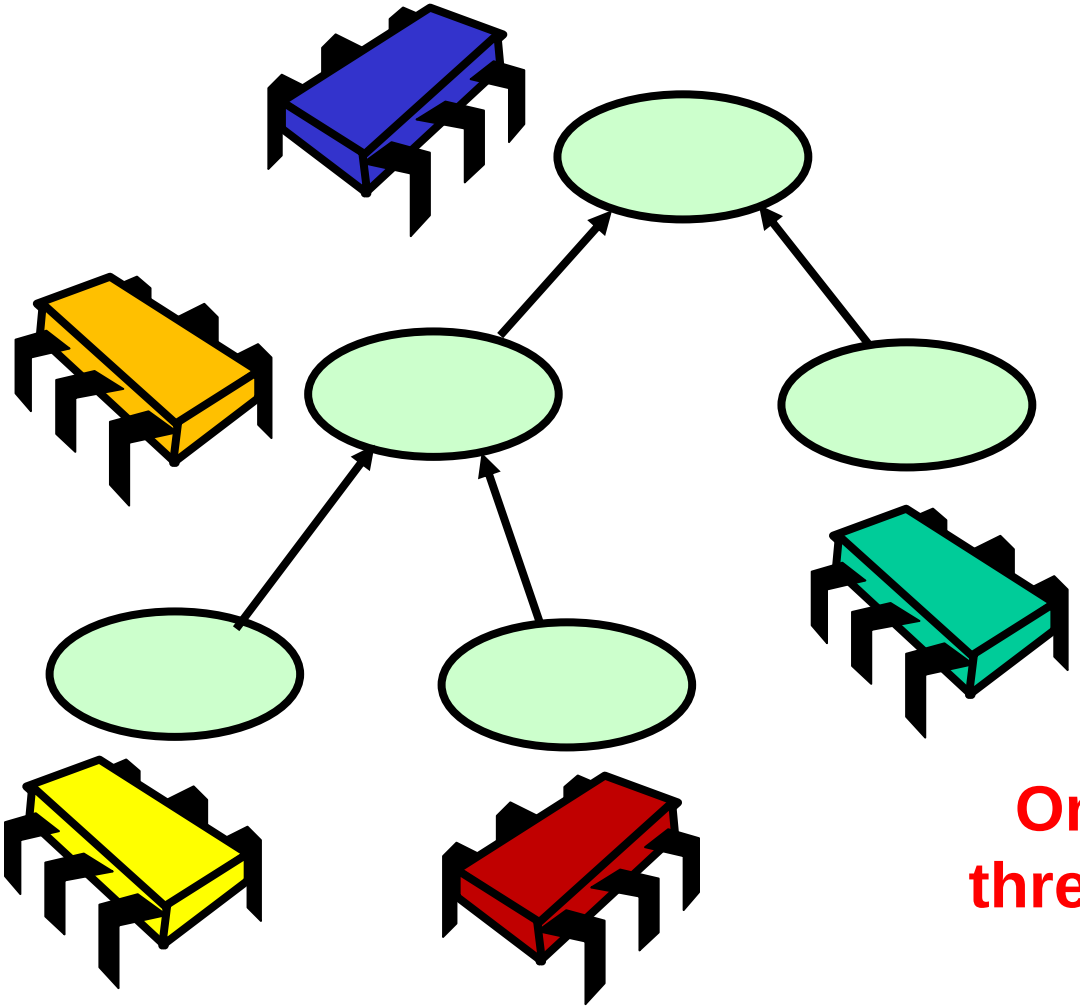


# Which is best for Multicore?

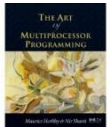
- On a cache coherent multicore chip: perhaps none of the above…
- Here is another (arguably) better algorithm …



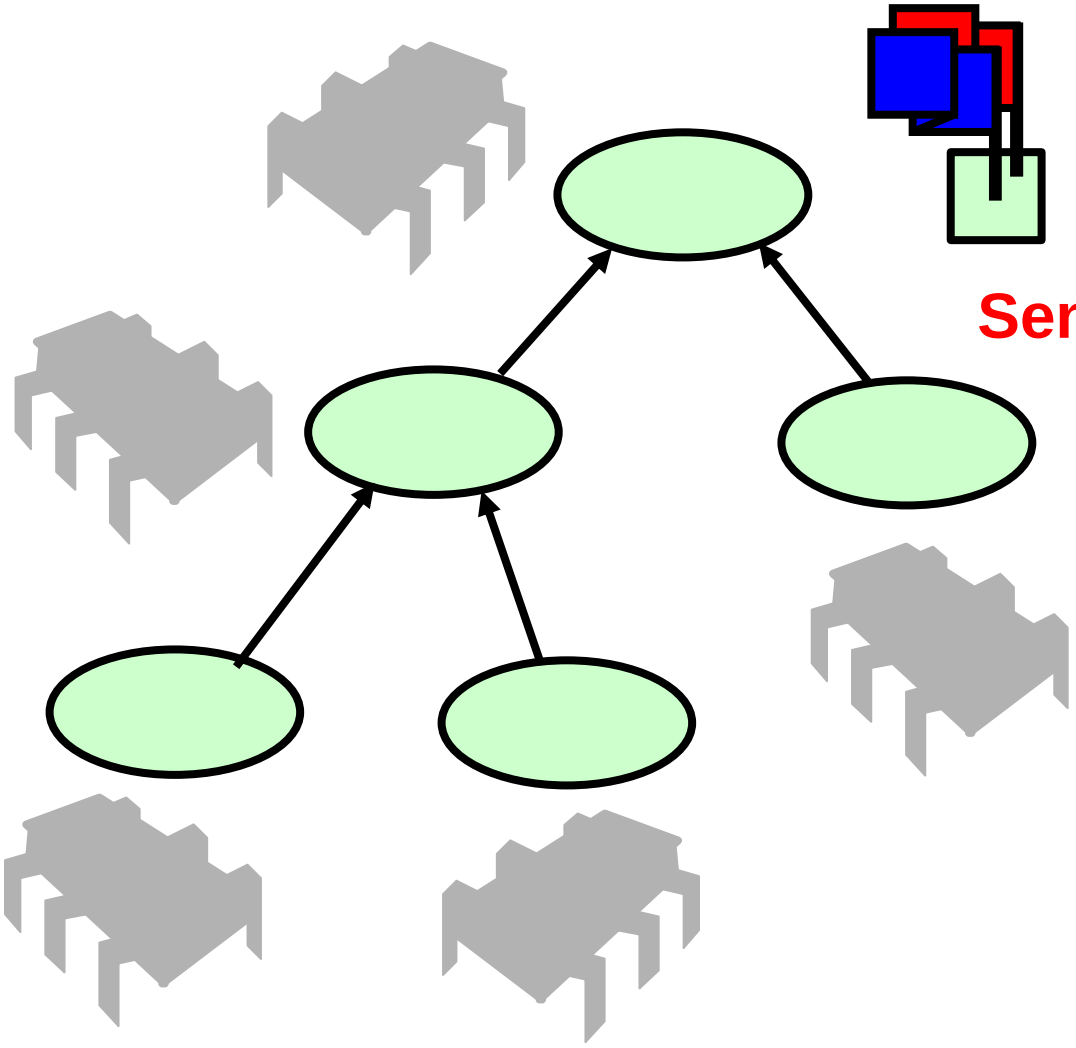
# Static Tree Barrier



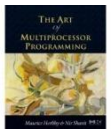
**One node per thread, statically assigned**



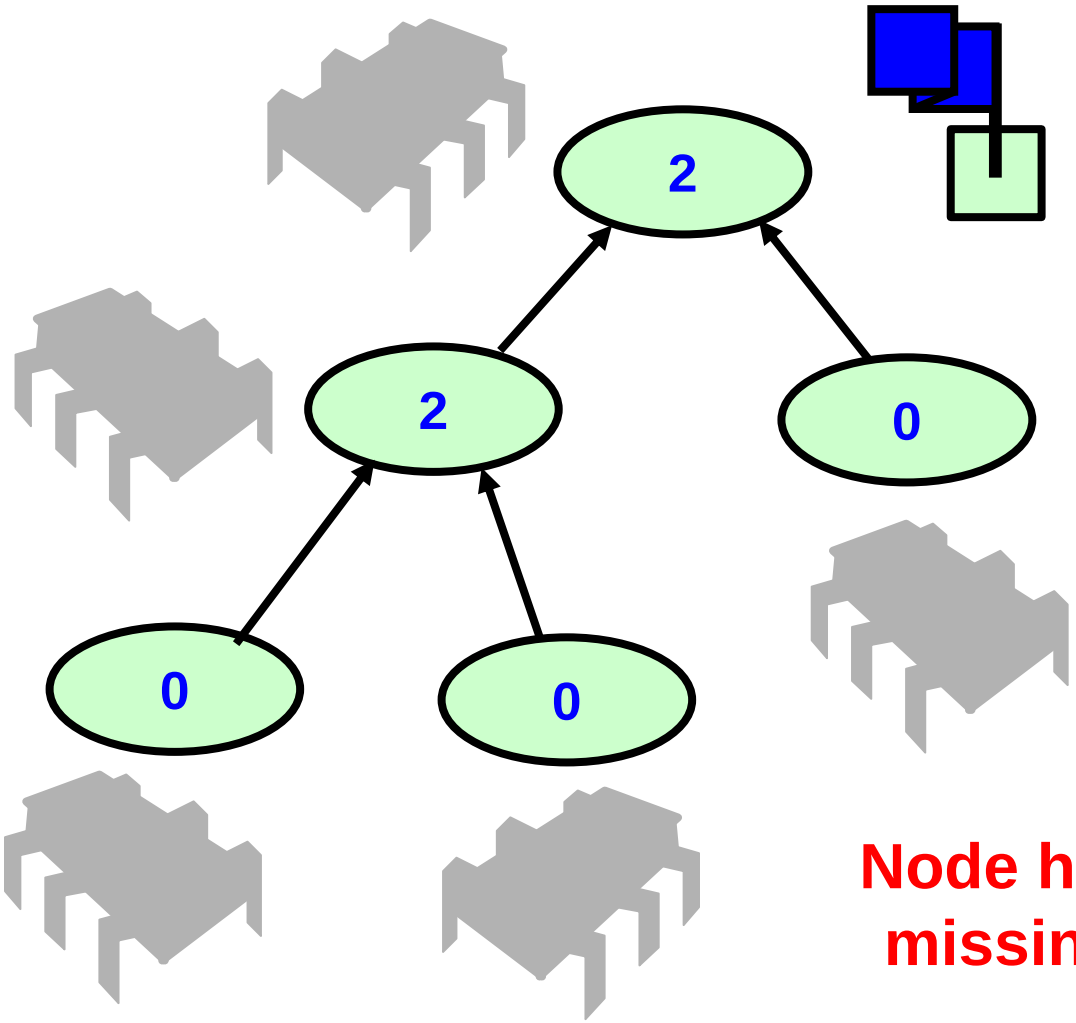
# Static Tree Barrier



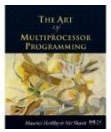
**Sense-reversing flag**



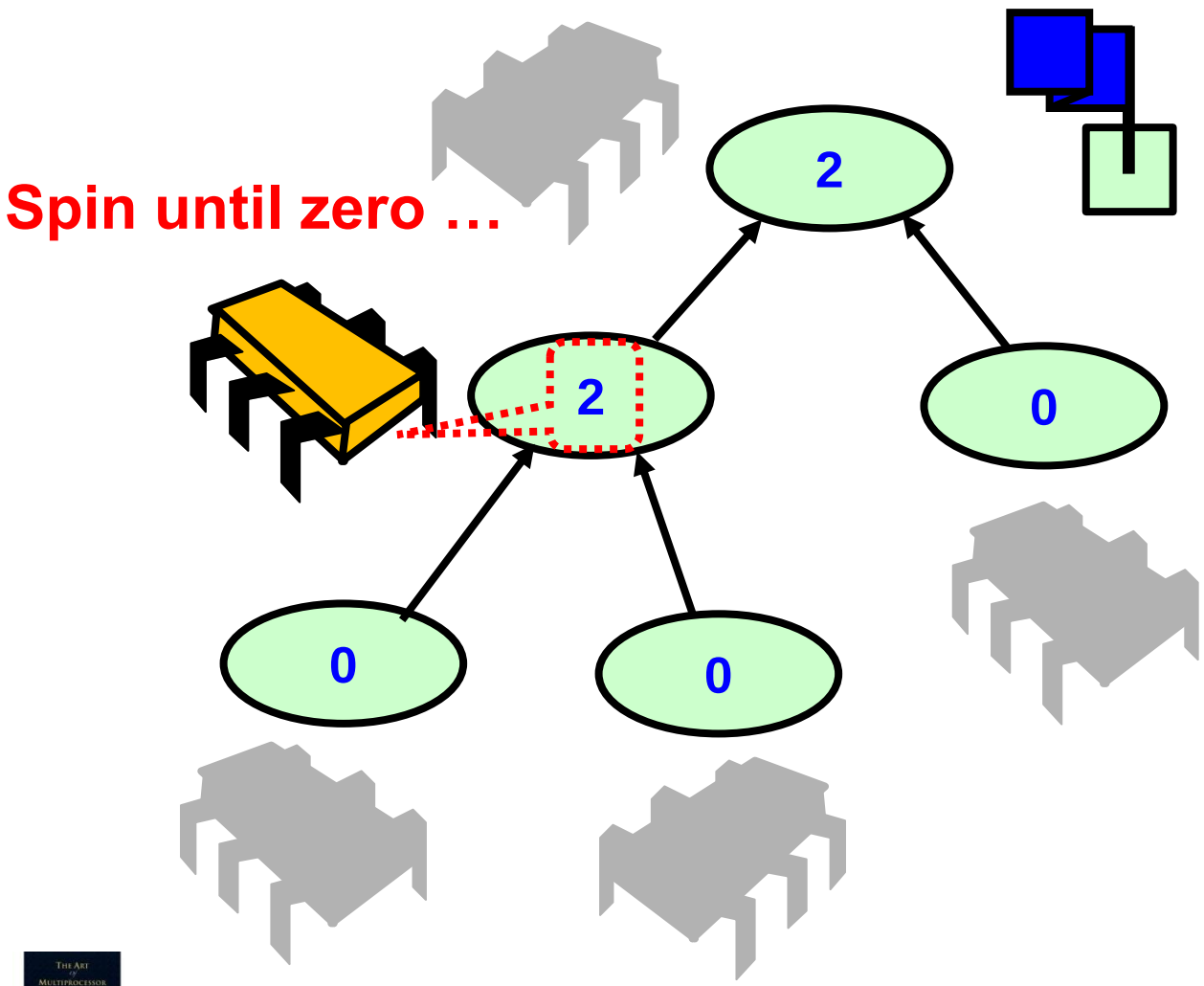
# Static Tree Barrier

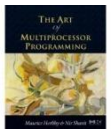



**Node has count of missing children**

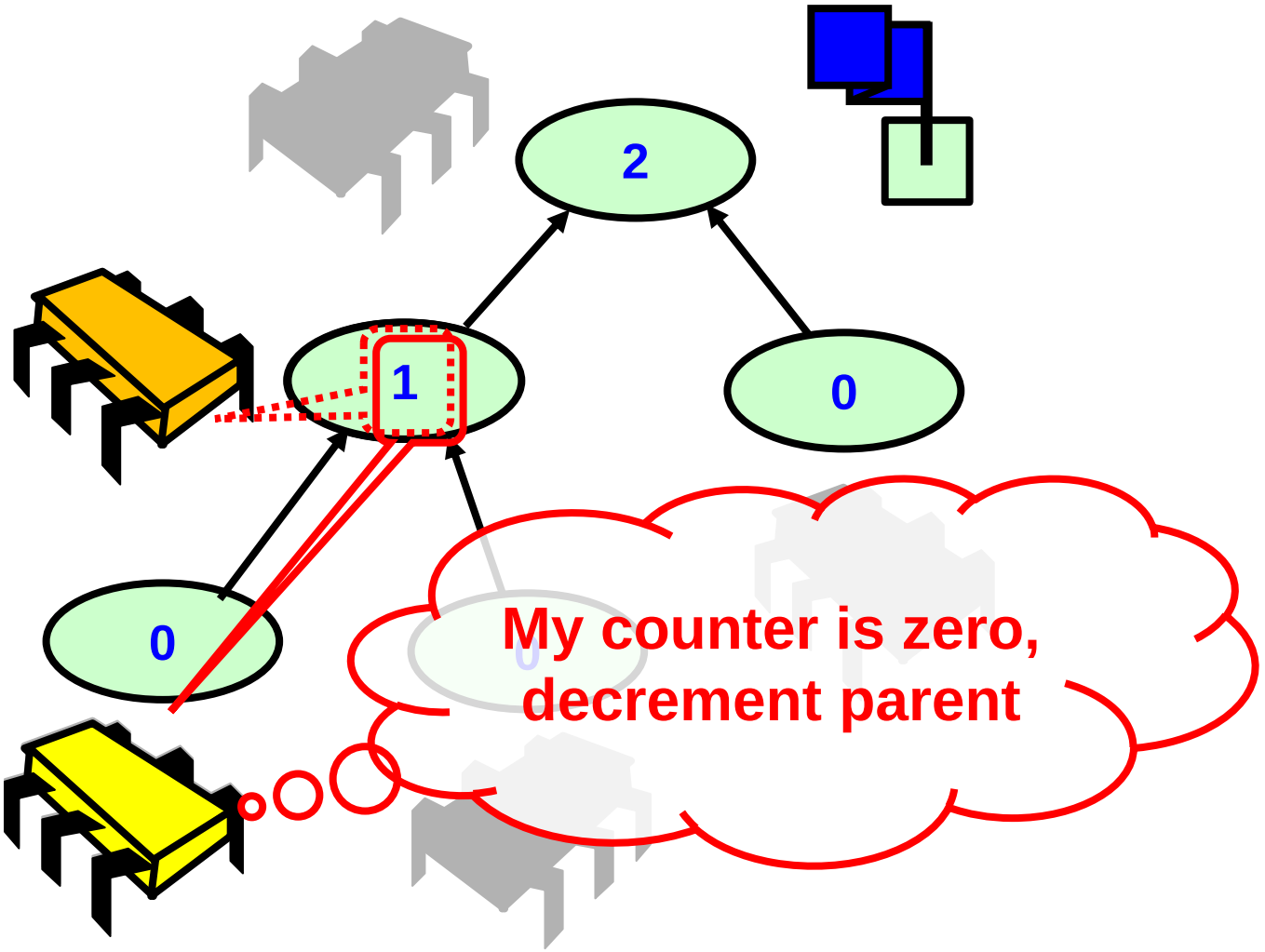


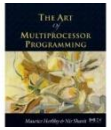
# Static Tree Barrier



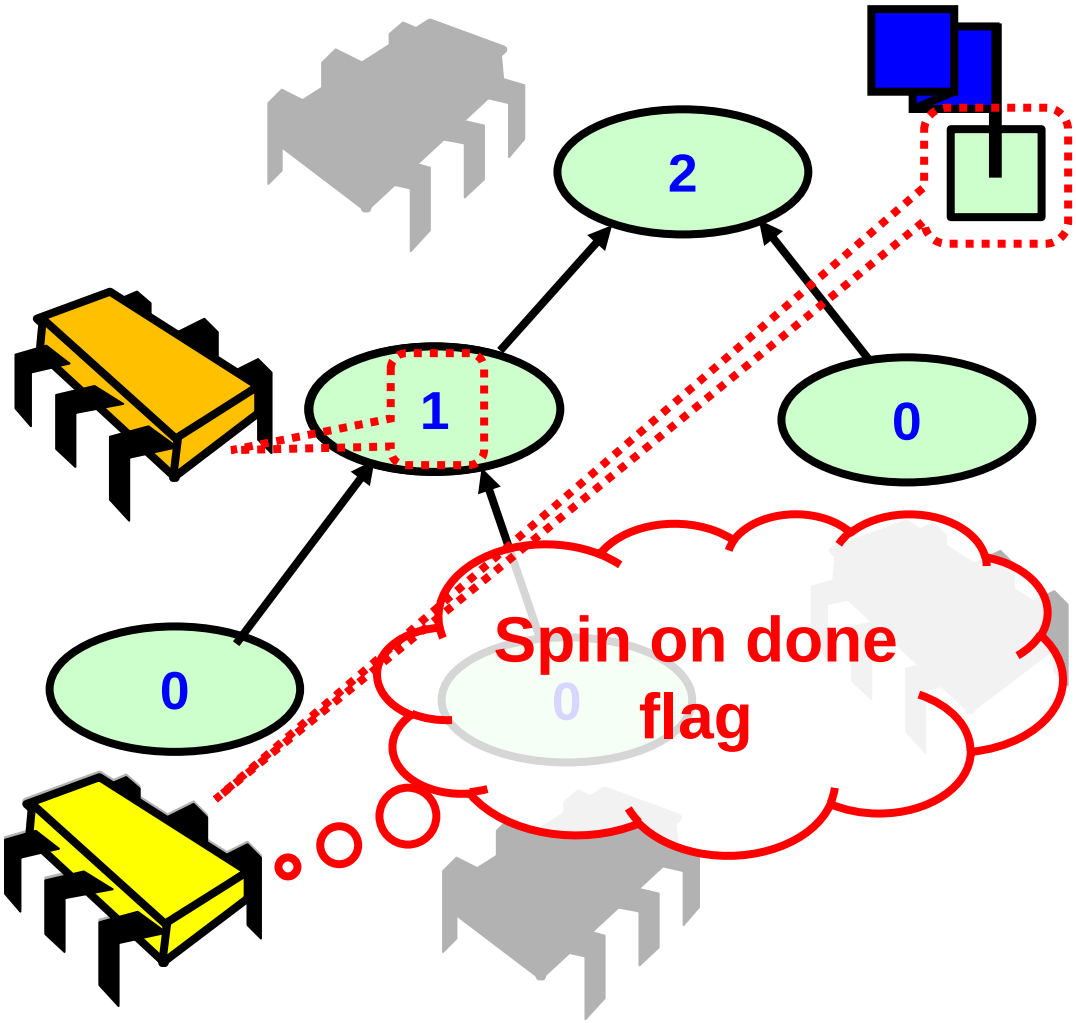



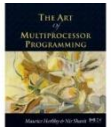
# Static Tree Barrier



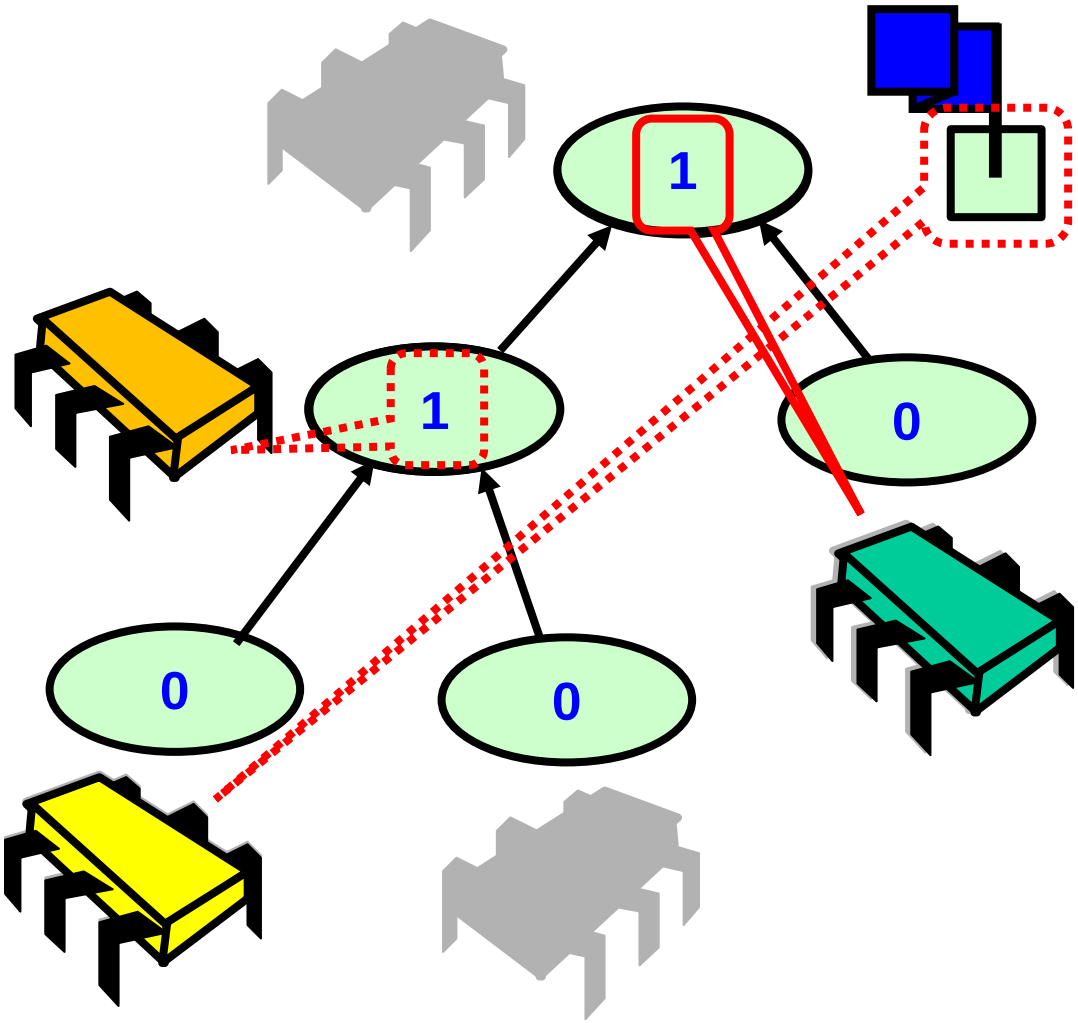



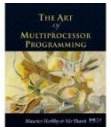
# Static Tree Barrier



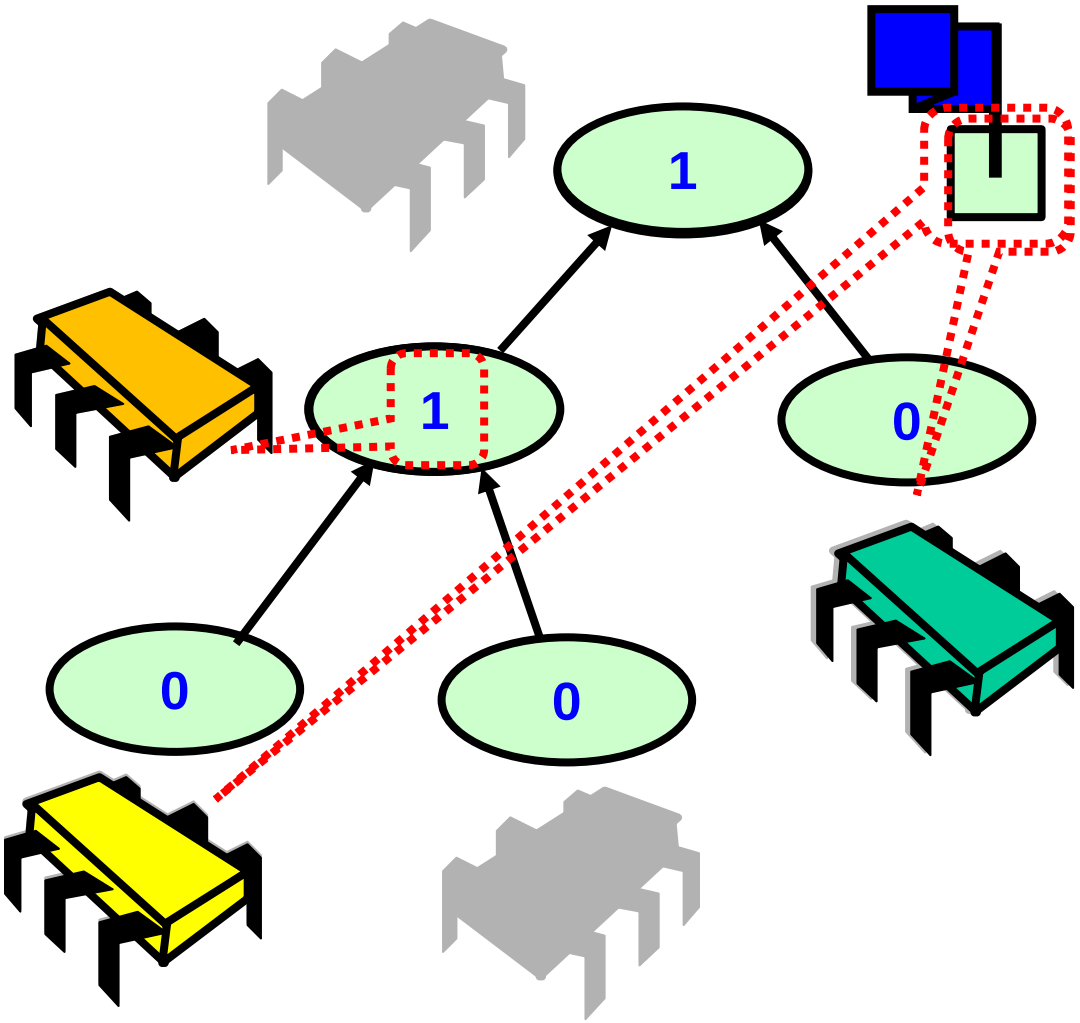



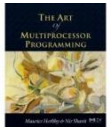
# Static Tree Barrier



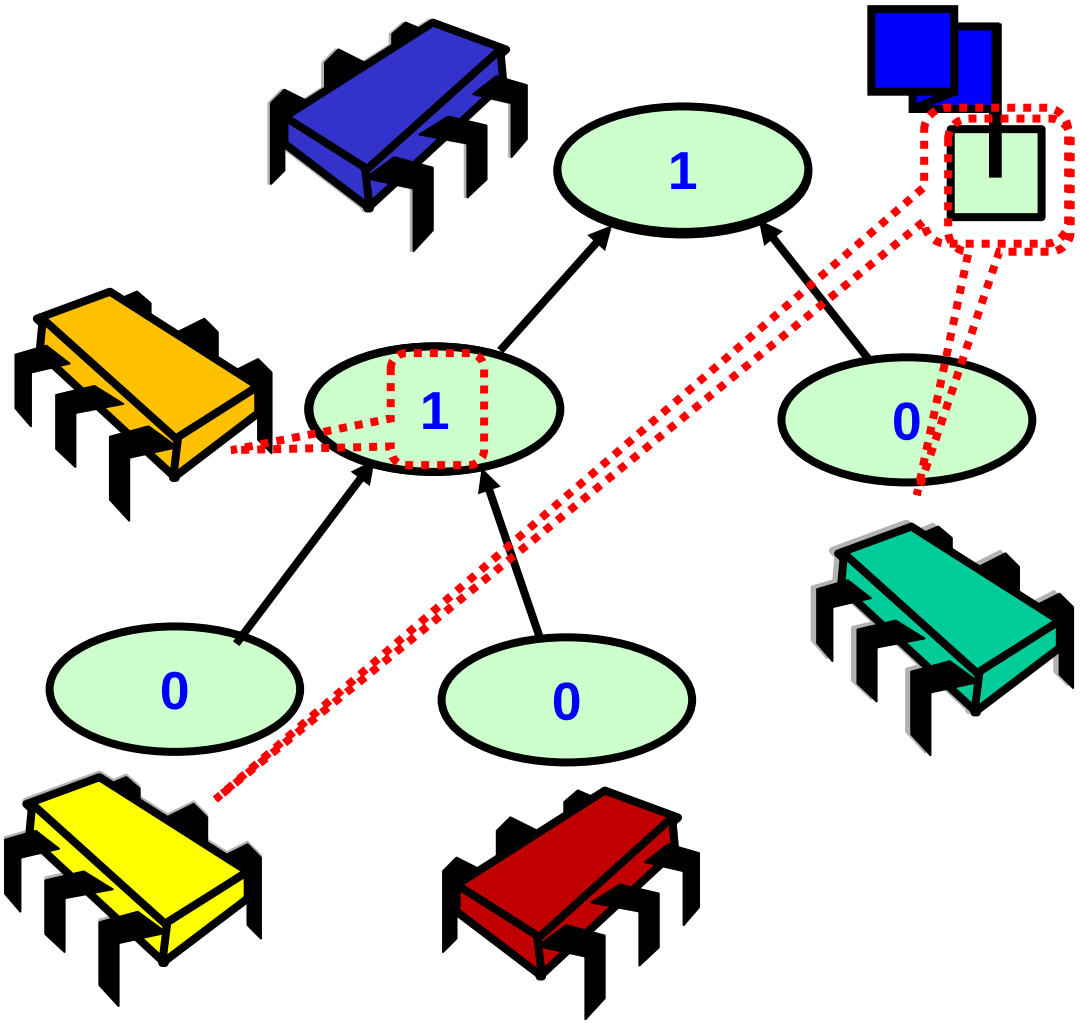



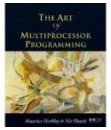
# Static Tree Barrier



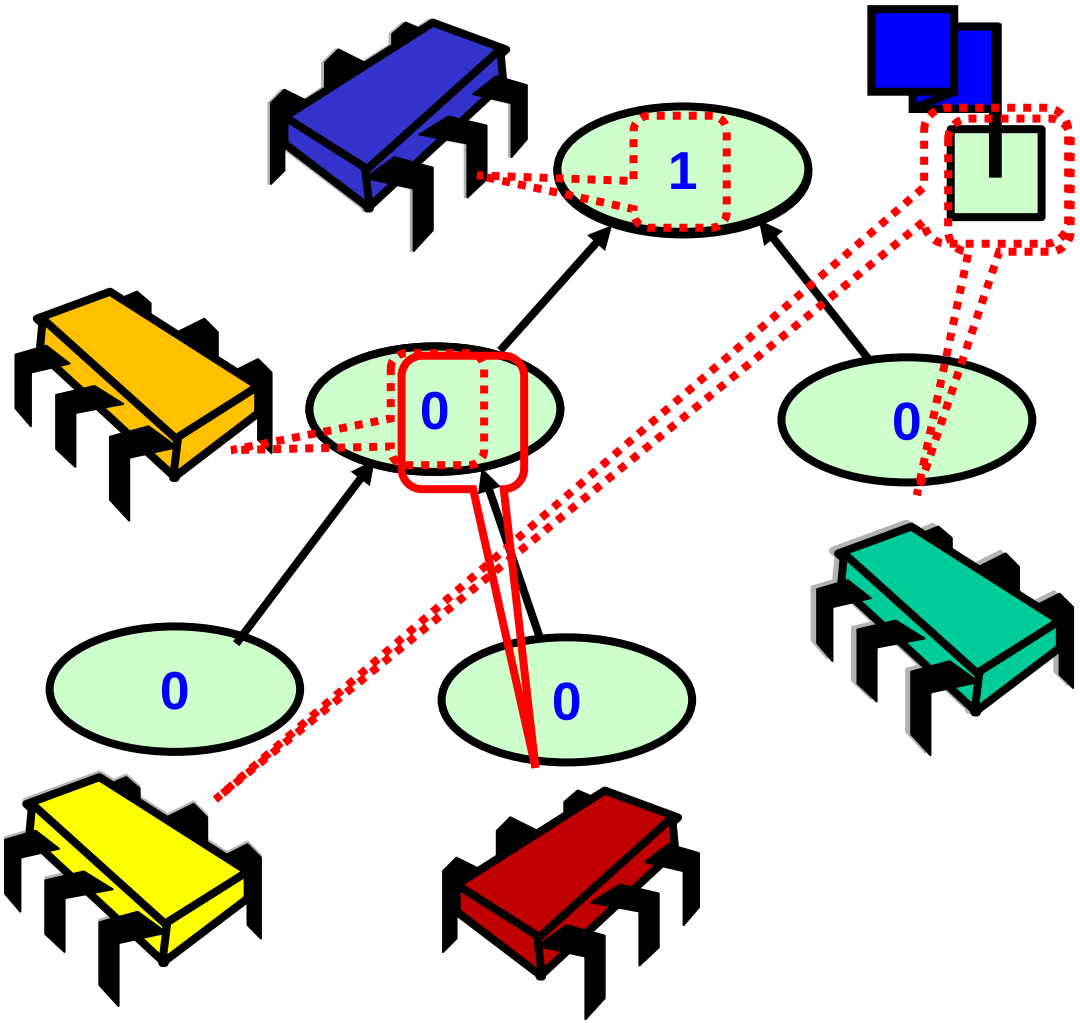



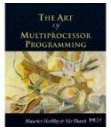
# Static Tree Barrier



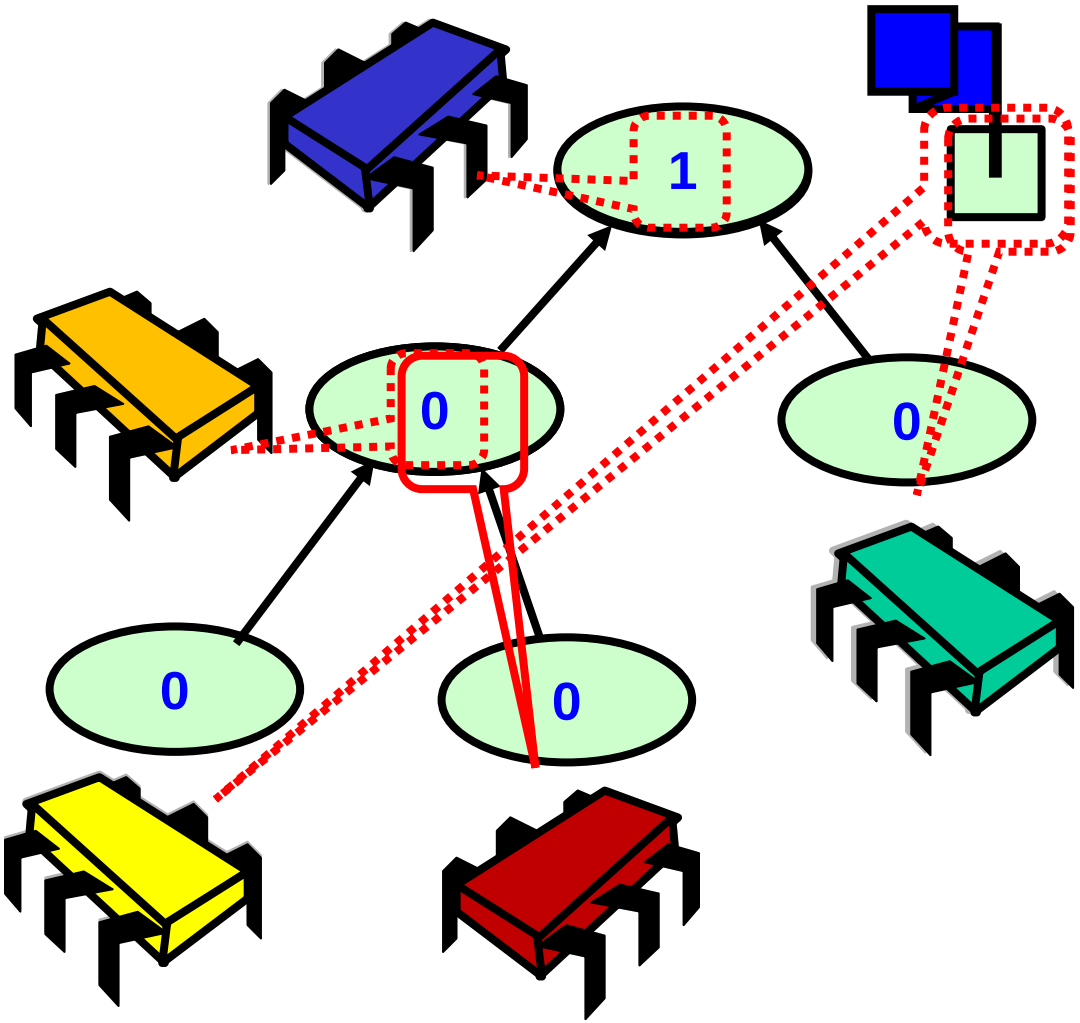



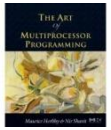
# Static Tree Barrier



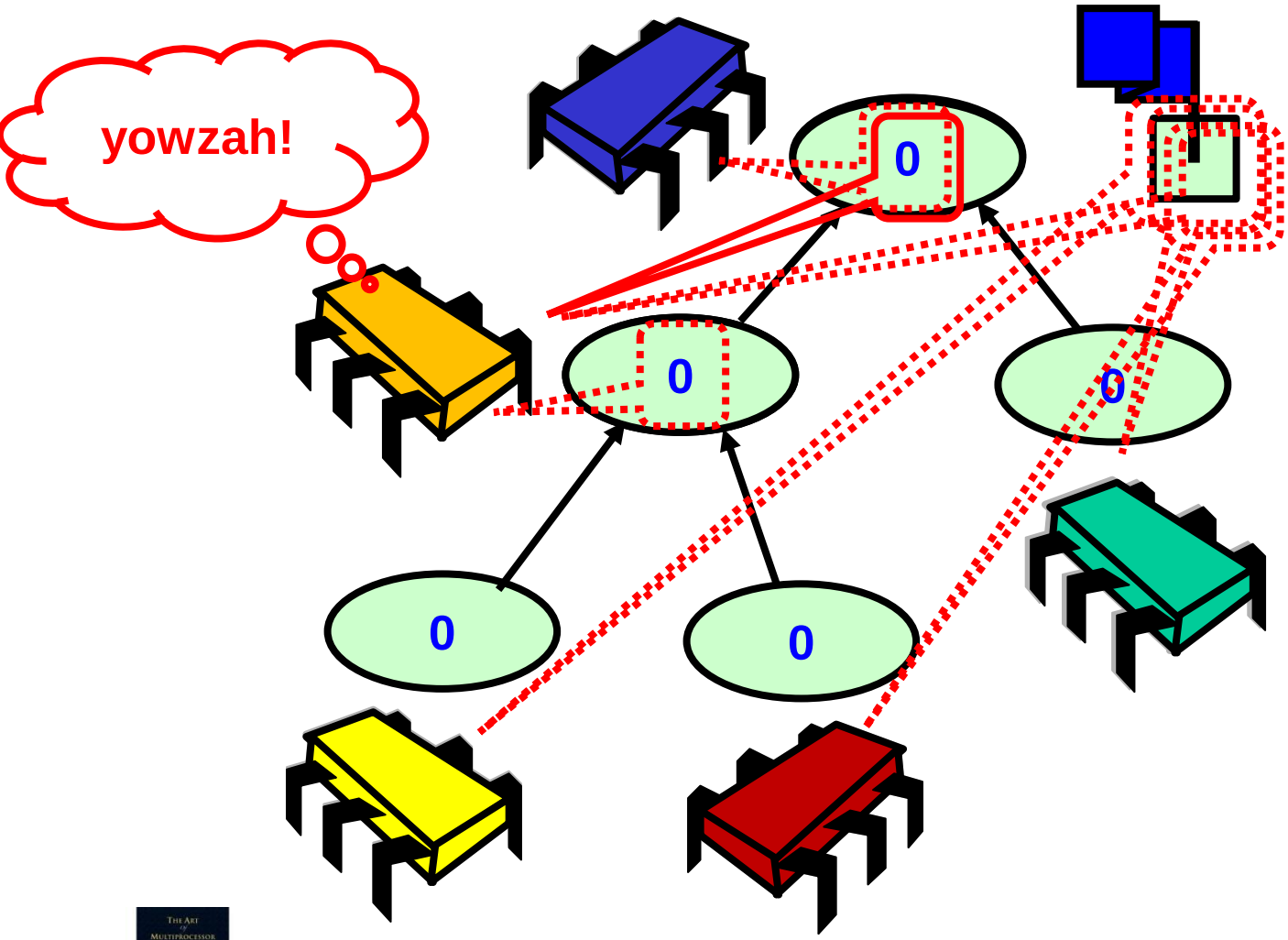



# Static Tree Barrier

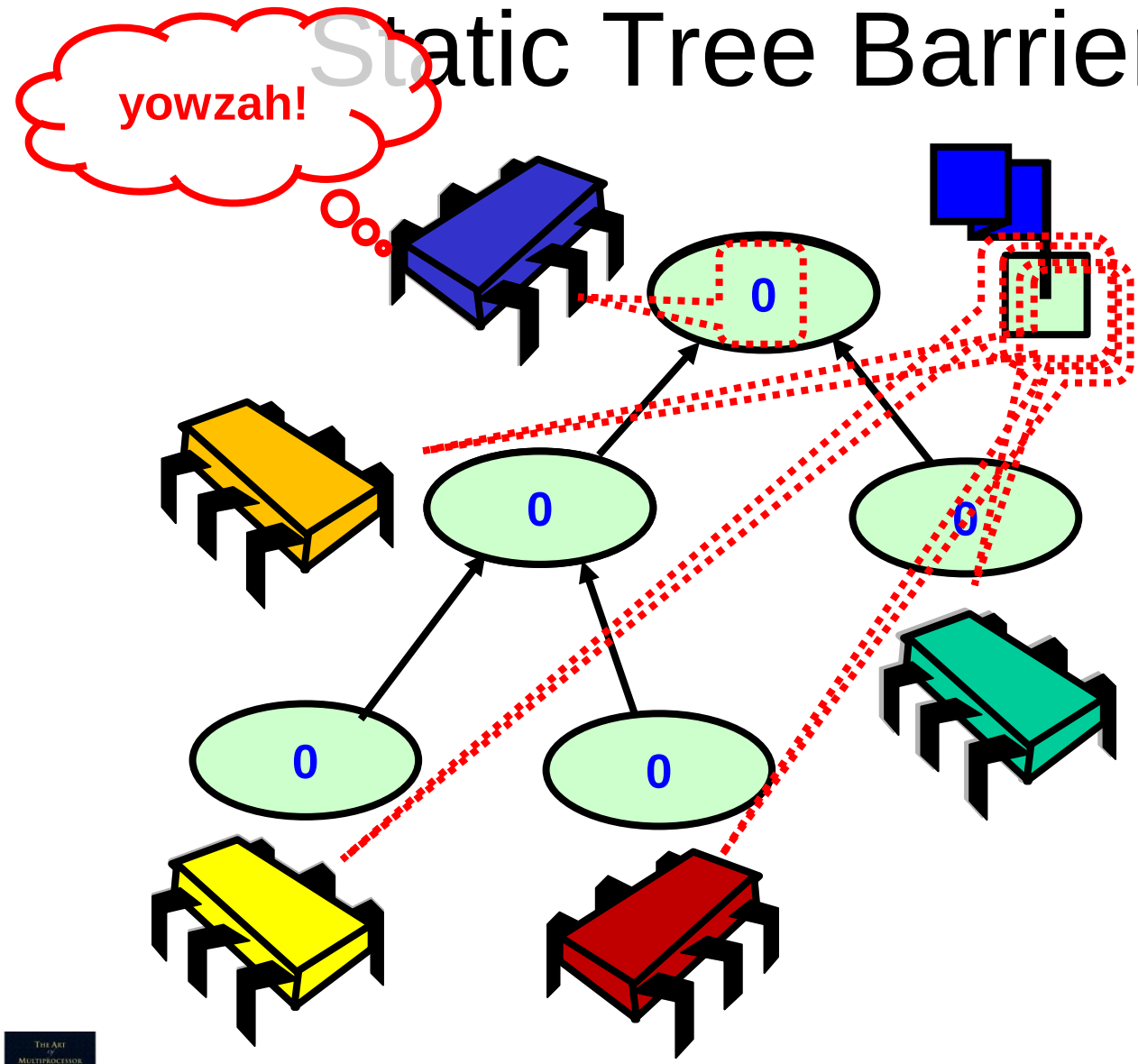


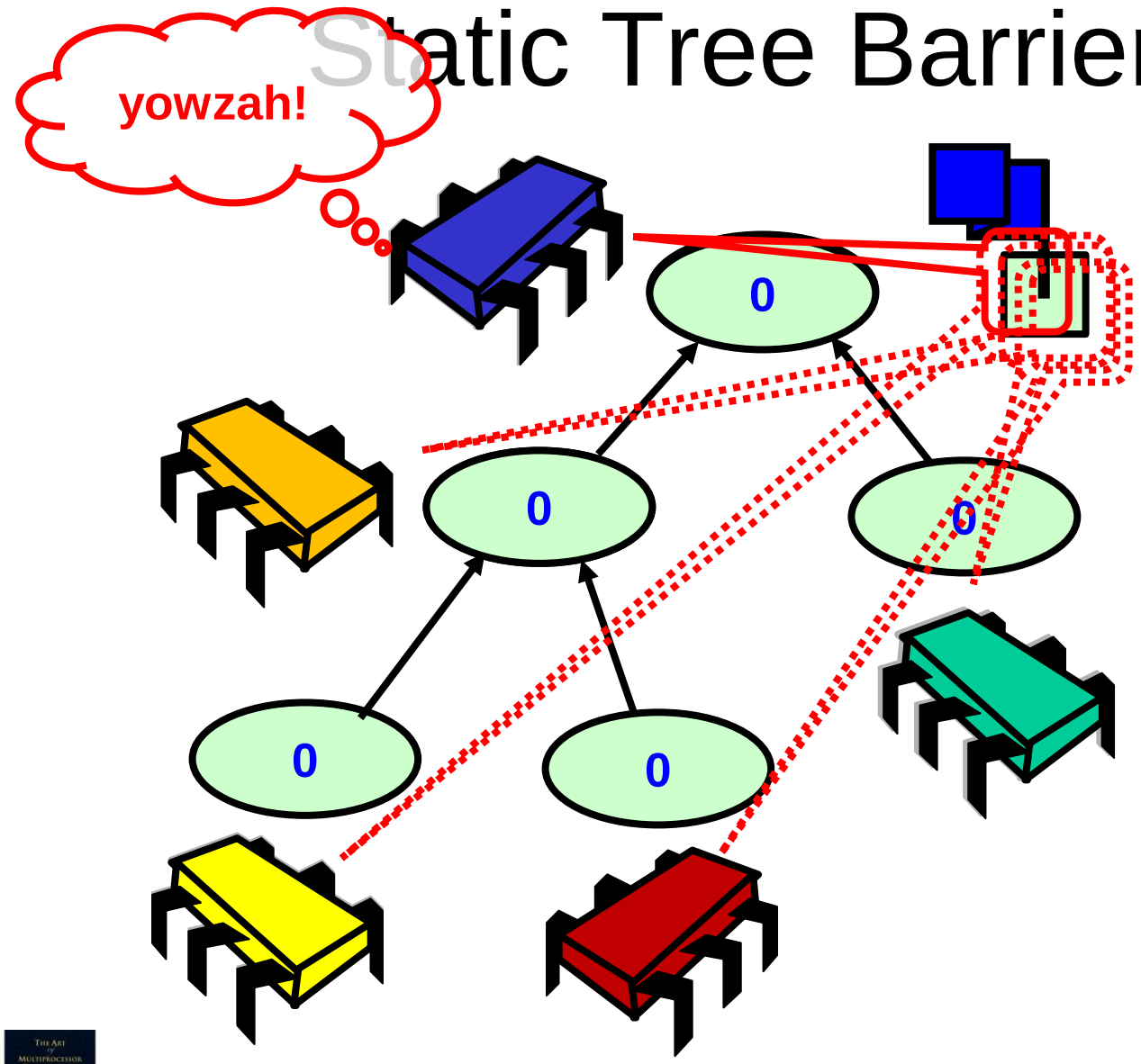
# Static Tree Barrier



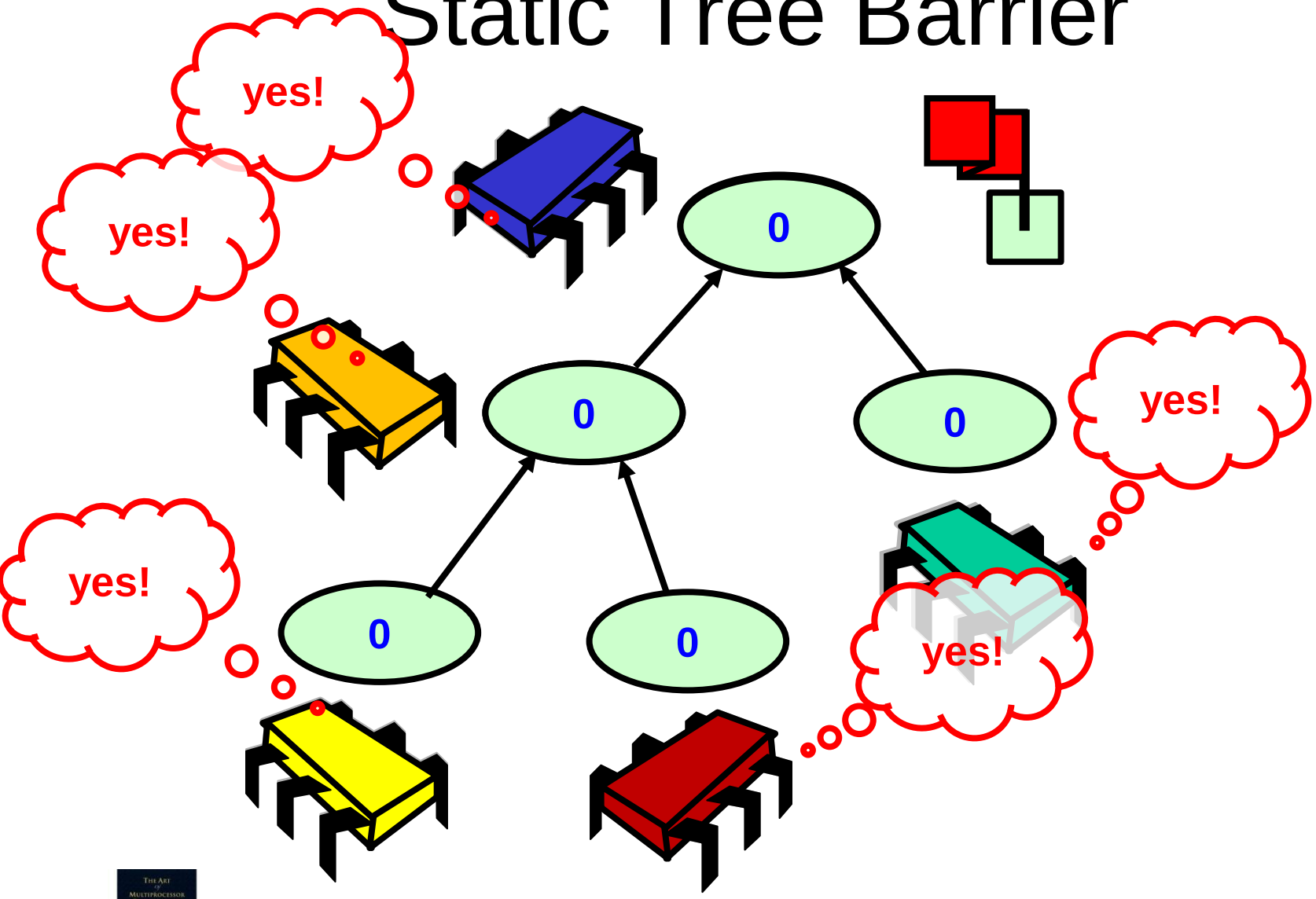
# Static Tree Barrier



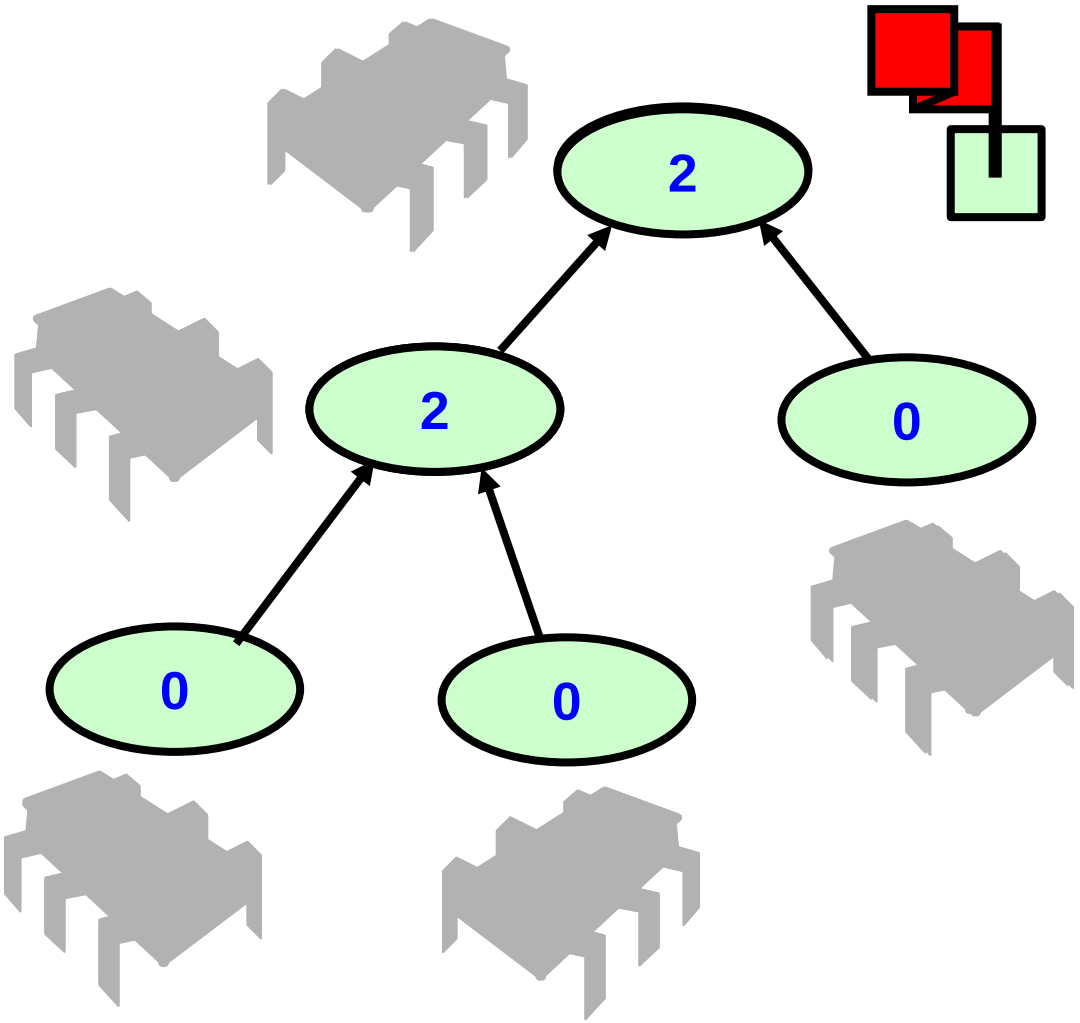
# Static Tree Barrier

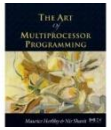


# Static Tree Barrier



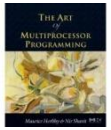
# Static Tree Barrier



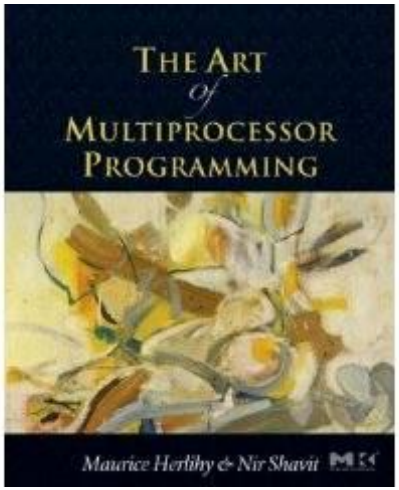



# Remarks

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



# The Nature of Progress*

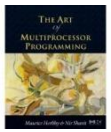



Companion slides for
The Art of Multiprocessor Programming
by Maurice Herlihy & Nir Shavit

# Concurrent Programming

- Many real-word data structures
  - blocking (lock-based) implementations &
  - non-blocking (no locks) implementations
- For example:
  - linked lists, queues, stacks, hash maps,…

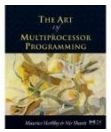
**Does this make sense?**



# Concurrent Programming

- Many data structures combine blocking & non-blocking methods
- Java™ concurrency package
  - skiplists, hash tables, exc
  - on 10 million deskt

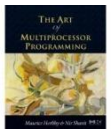
**Can seemingly contradictory conditions co-exist in same alg?…**



# Progress Conditions

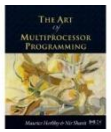
- *Deadlock-free:*
  - Some thread eventually acquires lock.
- *Starvation-free:*
  - Every thread eventually acquires lock.
- *Lock-free:*
  - Some method call returns.
- *Wait-free:*
  - Every method call returns.
- *Obstruction-free:*
  - Every method call ret… if it ex…

**We will show an example shortly**

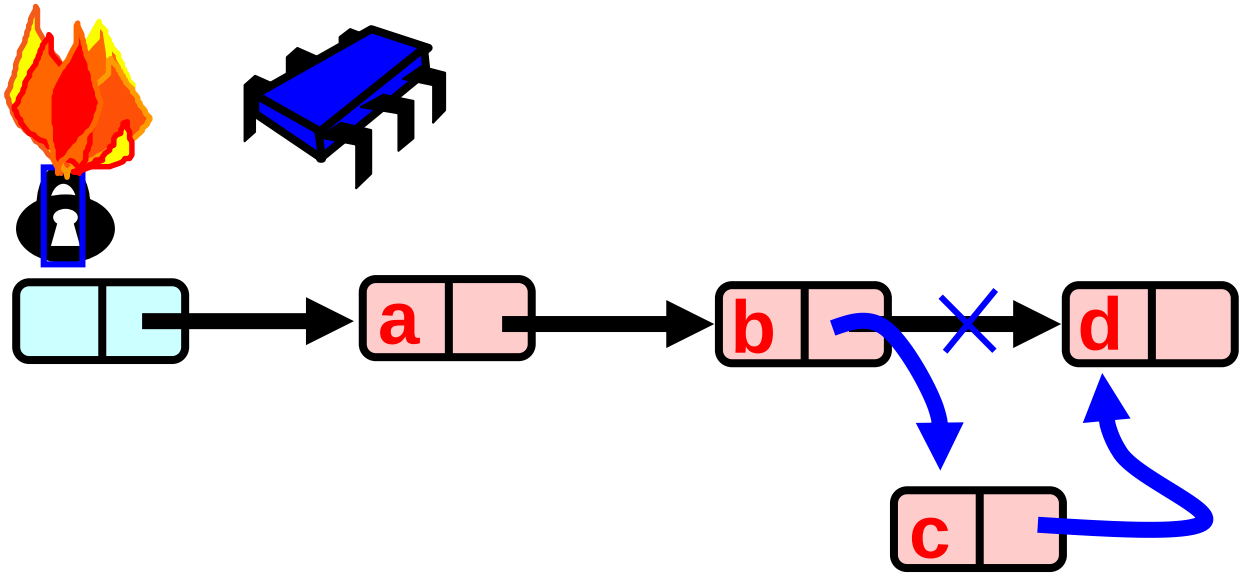


# List-Based Sets

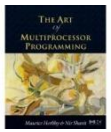
- Unordered collection of elements
- No duplicates
- Methods
  - Add() a new element
  - Remove() an element
  - Contains() if element is present



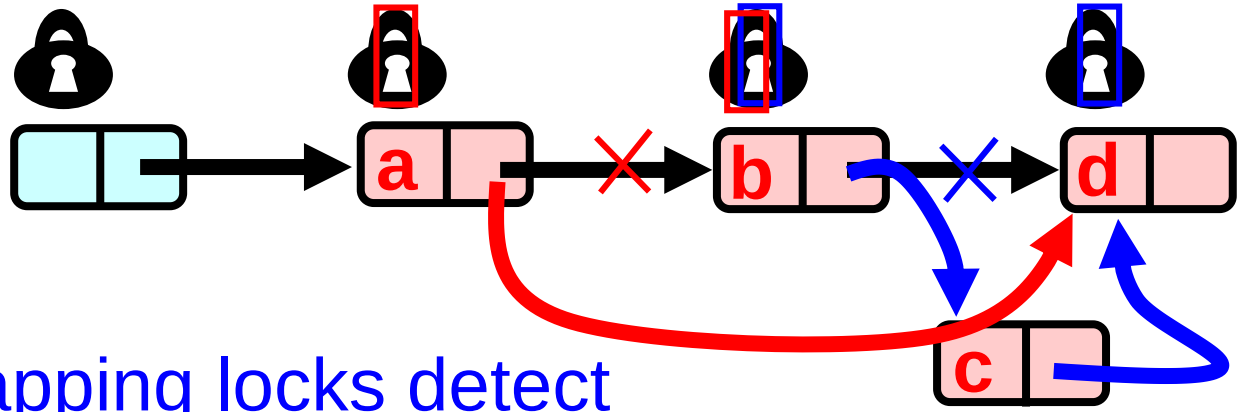
# Coarse Grained Locking




Lock is starvation-free: every attempt to acquire the lock eventually succeeds.

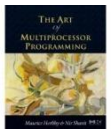


# Fine Grained (Lock Coupling)

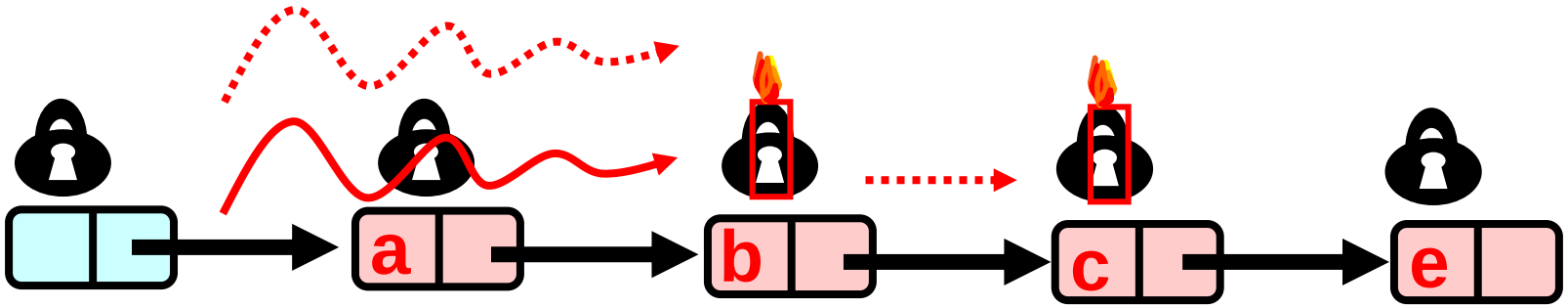



Overlapping locks detect overlapping operations

Deadlock-free: some thread eventually acquires lock.

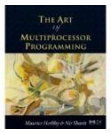


# Optimistic Fine Grained

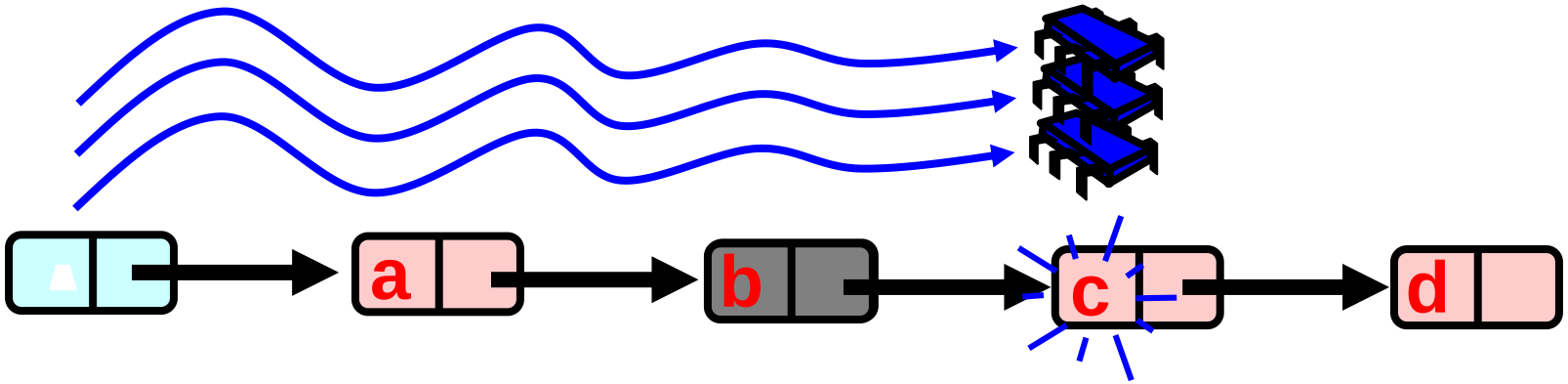



`add()`, `remove()`, `contains()` lock destination nodes in order

Deadlock-free: some thread trying to acquire the locks eventually succeeds.

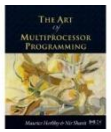


# Obstruction-free contains()



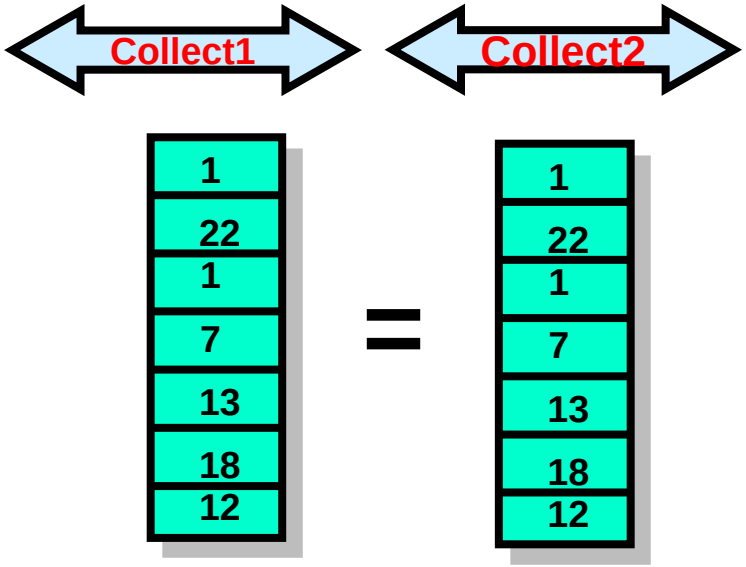

Snapshot: if all nodes traversed twice are the same

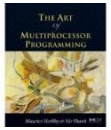
Obstruction-free: the method returns if it executes in isolation for long enough



# The Simple Snapshot is Obstruction-Free

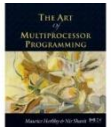
- Put increasing labels on each entry
- Collect twice
- If both agree,
  - We're done
- Otherwise,
  - Try again



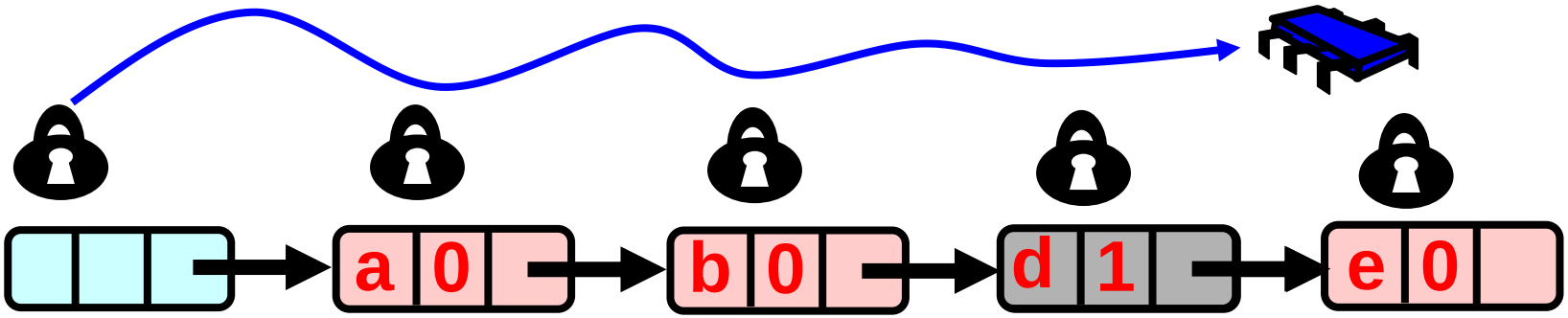



# Obstruction-freedom

- In the simple snapshot alg:
  - The update method is wait-free
  - But scan is obstruction-free
    - Completes if it executes in isolation
    - (no concurrent updates).

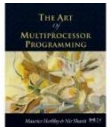


# Wait-free contains()

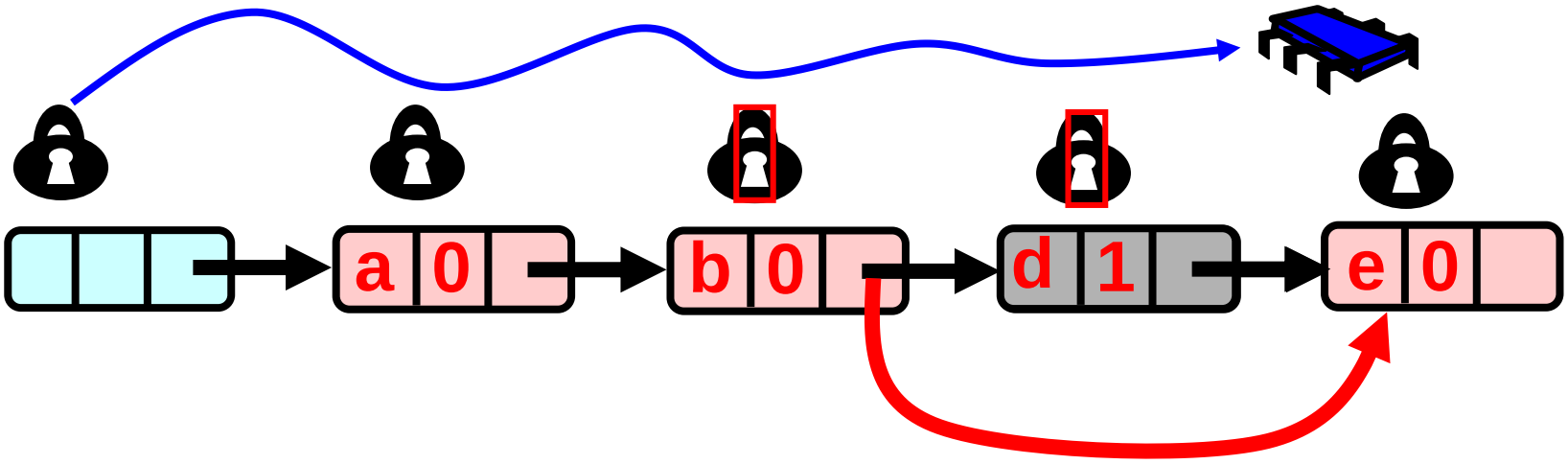



Use mark bit + list ordering

1. Not marked → in the set
2. Marked or missing → not in the set



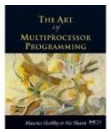
# Lazy List-based Set Alg



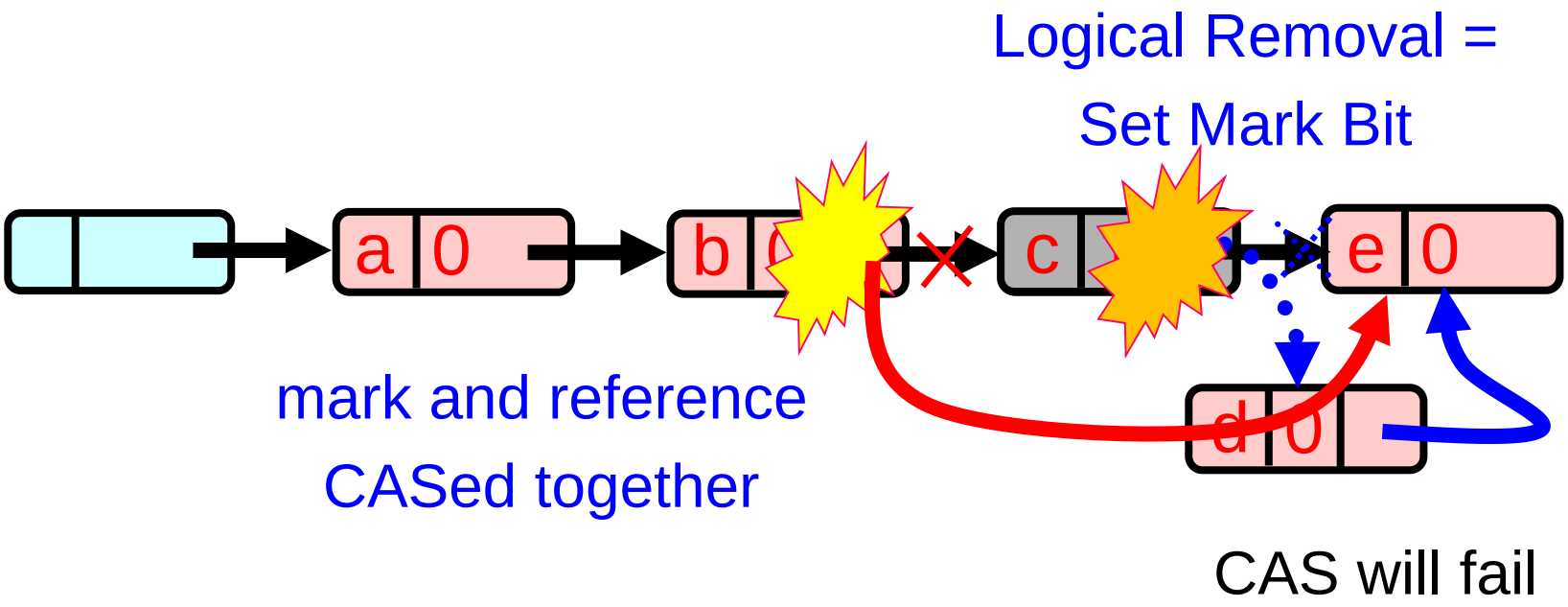

Combine blocking and non-blocking: *deadlock-free*

**`add()`** and **`remove()`** and *wait-free*

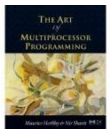
**`contains()`**



# Lock-free List-Based Set

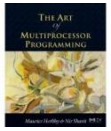



*Lock-free* **`add()`** and **`remove()`** and *wait-free* **`contains()`**



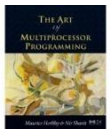
# So how can this make sense?

- Why have methods with different progress conditions?
- Let us try to understand this…



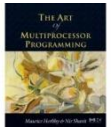
# Progress Conditions

- *Deadlock-free:*
  - Some thread eventually acquires lock.
- *Starvation-free:*
  - Every thread eventually acquires lock.
- *Lock-free:*
  - Some method call returns.
- *Wait-free:*
  - Every method call returns.
- *Obstruction-free:*
  - Every method call returns if it executes in isolation



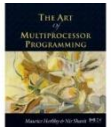
# A "Periodic Table" of Progress Conditions

| | Non-Blocking | | Blocking |
|---|---|---|---|
| **All make progress** | **Wait-free** | **Obstruction-free** | **Starvation-free** |
| **Some make progress** | **Lock-free** | | **Deadlock-free** |



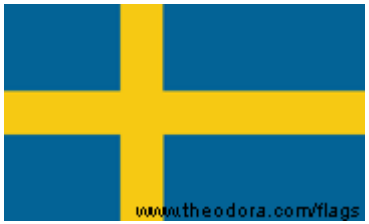
# More Formally

- Standard notion of *abstract object*
- Progress conditions relate to method calls of an object
  - A thread is *active if it takes an infinite* number of concrete (machine level) steps
  - And is *suspended if not.*

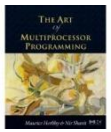


# Maximal vs. Minimal

- Minimal progress
  - some call eventually completes
  - System matters, not individuals
- Maximal progress
  - every call eventually completes.
  - Individuals matter

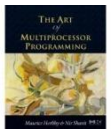

Flags courtesy of www.theodora.com/flags used with permission



# The "Periodic Table" of Progress Conditions

| | Non-Blocking | | Blocking |
|---|---|---|---|
| **Maximal progress** | **Wait-free** | **Obstruction-free** | **Starvation-free** |
| **Minimal progress** | **Lock-free** | | **Deadlock-free** |

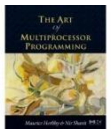




# The Scheduler's Role

Multiprocessor progress properties:

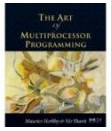
- Are not about the guarantees a method's implementation provides.
- Are about scheduling needed to provide minimal or maximal progress.



# Fair Scheduling

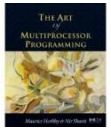
- A history is *fair* if each thread takes an infinite number of steps

- A method implementation is *deadlock-free* if it guarantees minimal progress in every *fair history*.



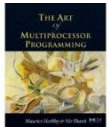
# Starvation Freedom

- A method implementation is *starvation-free* if it guarantees maximal progress in every *fair history*.



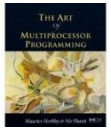
# Dependent Progress

- Dependent progress conditions
  - Do not guarantee minimal progress in every history
- Independent ones do.
- Blocking progress conditions
  - *deadlock-freedom*, *Starvation-freedom*
  - are dependent.



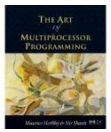
# Non-blocking Independent Conditions

- A lock-free method guarantees
  - minimal progress
  - in every history.
- A wait-free method guarantees
  - maximal progress
  - in every history.



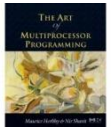
# The "Periodic Table" of Progress Conditions

| | Non-Blocking | | Blocking |
|---|---|---|---|
| **Maximal progress** | **Wait-free** | **Obstruction-free** | **Starvation-free** |
| **Minimal progress** | **Lock-free** | | **Deadlock-free** |
| | **Independent** | | **Dependent** |



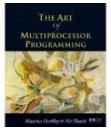
# Uniformly Isolating Schedules

- A history is uniformly isolating if any thread eventually runs by itself for "long enough"
- Modern systems do this with backoff, yield, etc.

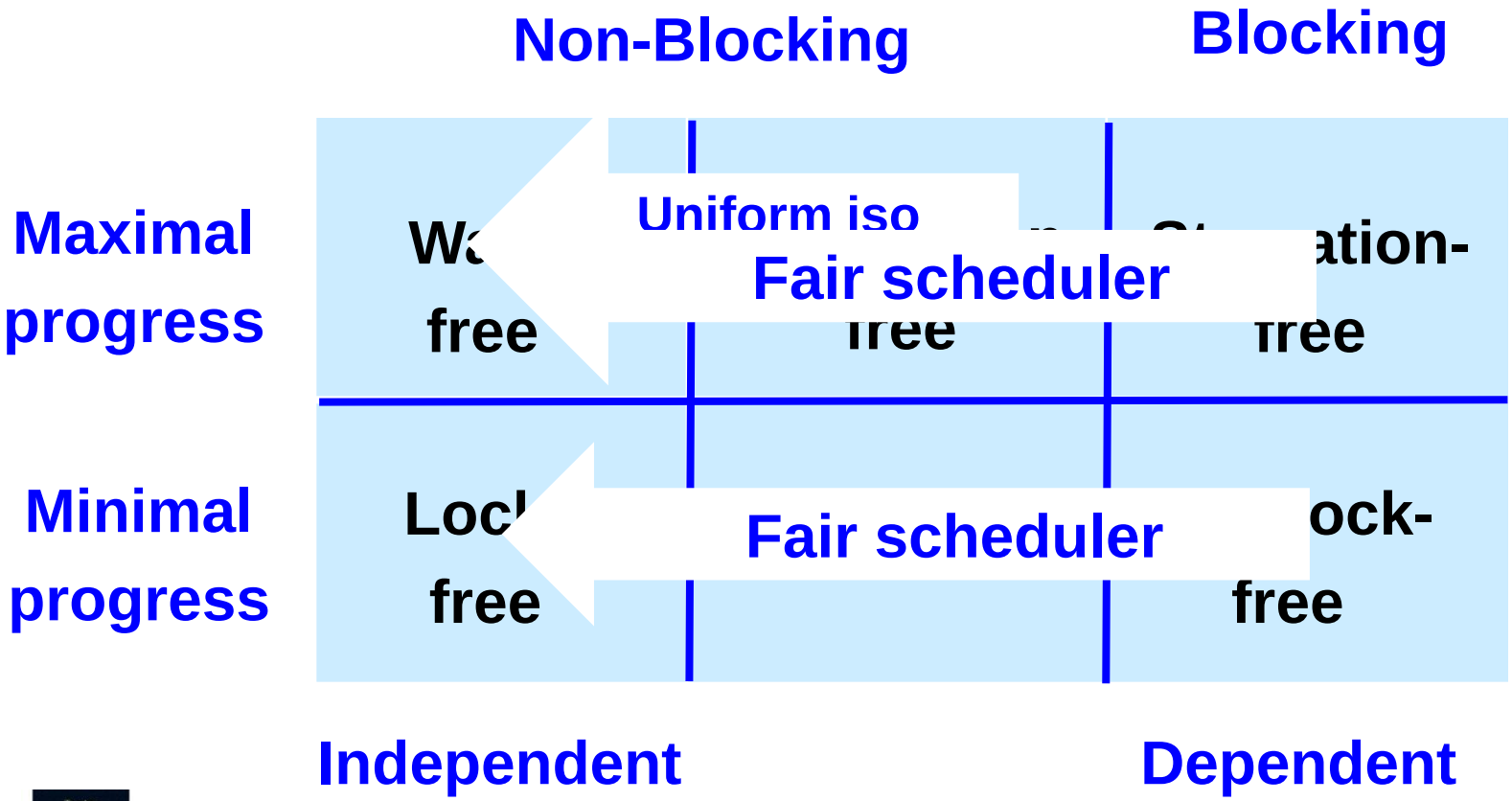


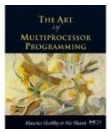
# A Non-blocking Dependent Condition

- A method implementation is obstruction-free if it guarantees
  - maximal progress
  - in every uniformly isolating history.



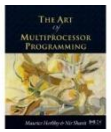
# The "Periodic Table" of Progress Conditions



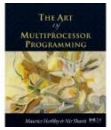
# The "Periodic Table" of Progress Conditions

| | Non-Blocking | | Blocking |
|---|---|---|---|
| Maximal progress | Wait-free | Obstruction-free | Starvation-free |
| Minimal progress | Lock-free | Clash-free | Deadlock-free |
| | Independent | | Dependent |

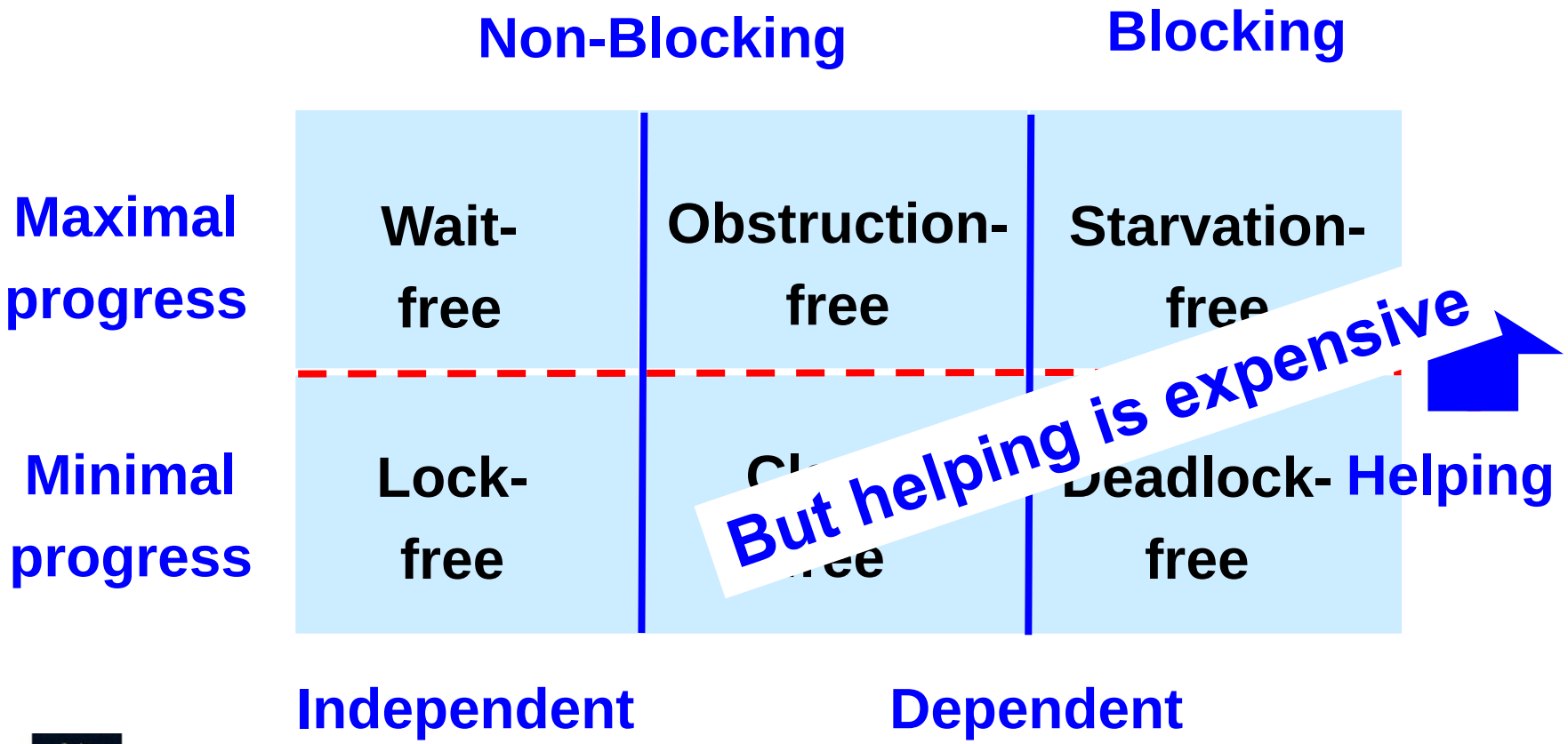


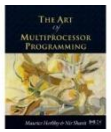
# Clash-Freedom: the "Einsteinium" of Progress

- A method implementation is clash-free if it guarantees
  - minimal progress
  - in every uniformly isolating history.
- Thm: clash-freedom strictly weaker than obstruction-freedom



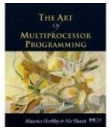
# Getting from Minimal to Maximal



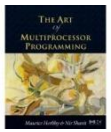
# Maximal Progress Postulate

- Programmers want maximal progress.
- Methods' progress conditions define
  - What we expect from the scheduler
  - For example
    - Don't halt in critical section
    - Let me run in isolation long enough …

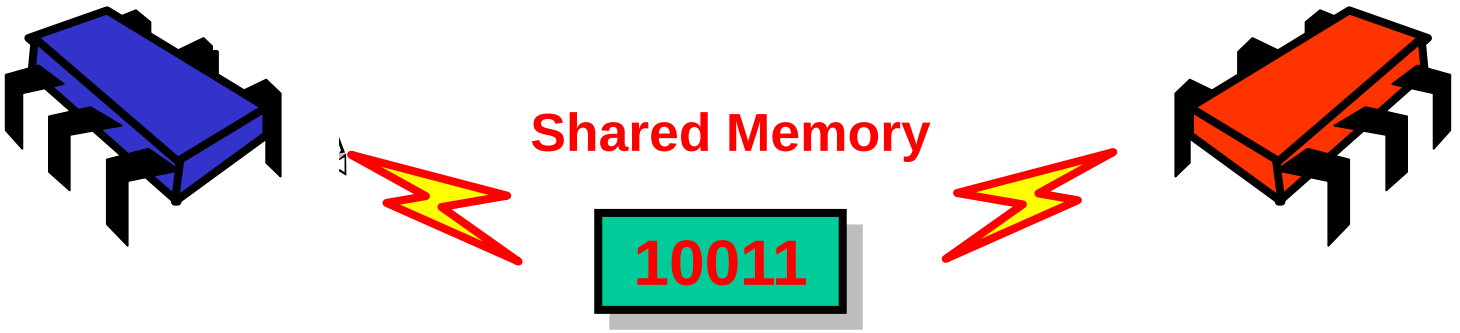


# Why Lock-Free is OK

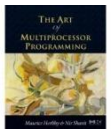
- We all want maximal progress
  - Wait-free
- Yet we often write lock-free or deadlock-free lock-based algorithms
- OK if we expect the scheduler to be benevolent
  - Often true (not always!)



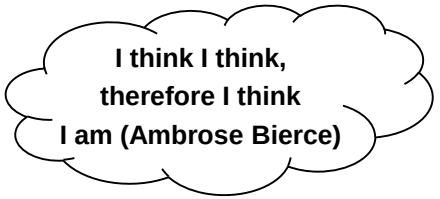
# Shared-Memory Computability



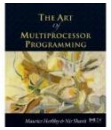

- What is (and is not) concurrently computable
- Wait-free Atomic Registers
- Lock-free/Wait-free Hierarchy
  and Universal Constructions



# Troubling Intellectual Question…

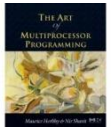



- Why use *non-blocking lock-free and wait-free* conditions when most code uses locks?

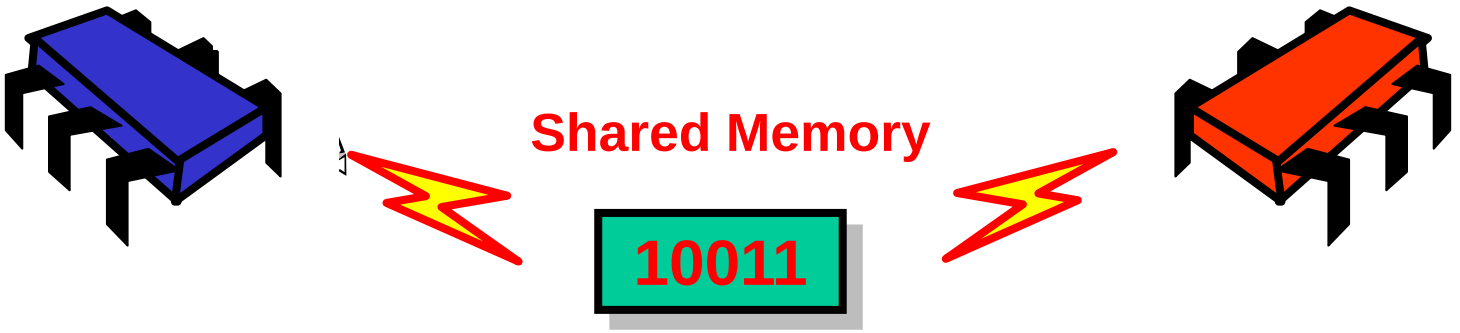


# The Answer

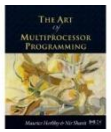
- Not about being non-blocking…
- About being independent!
- Do not rely on the good behavior of the scheduler.



# Shared-Memory Computability




- Independent progress: use *Lock-free* and *Wait-free* Memory Hierarchy and Universal Constructions



# Programmers Expect the Best

- Programmers expect maximal progress.
- Progress conditions define scheduler requirements necessary to achieve it.