

Barrier Synchronization

ch.17

synchronization problems:

- mutual exclusion:
 - keep others out;
 - e.g. access shared data;
 - use locks or Java's "synchronized".
- condition synchronisation
 - action depends on some state;
 - e.g. `buffer.read()` requires `notEmpty()`;
 - condition objects, Java's "wait/notify"

synchronization problems:

- barrier synchronization:
include all others:
everybody ready to move on?
- termination detection
everybody finished?

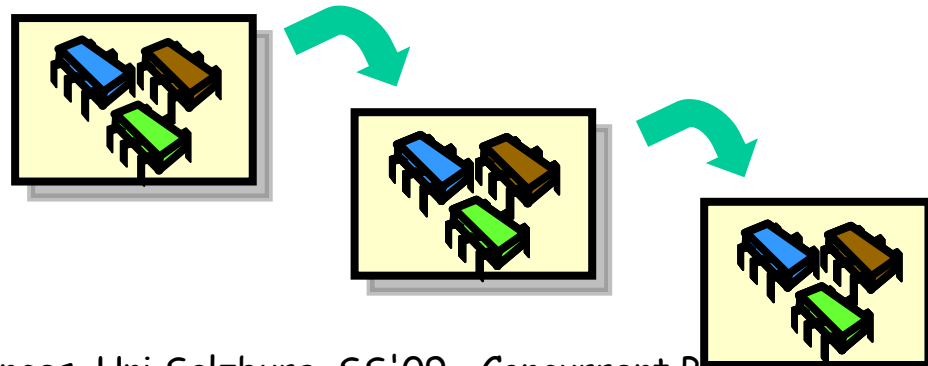
.....

this chapter:

- barrier synchronization
and a few words on termination detection
- like spin-locks chapter:
performance matters,
algo/hardware interaction

barrier synchronization: what is it?

- a multithreaded computation may consist of a **series of phases**
- a thread may enter a **new phase only when all threads are ready to go:**



barrier interface & typical use

```
interface Barrier {  
    public void await() } //may have param's
```

```
Barrier b = new MyKindOfBarrier( ...);
```

```
//thread:
```

```
While ( ... ) { //typical use:  
    doWork( ... ); //do own part of work  
    b.await(); //synchronize with others
```

Example:

display for computer game

- "soft real-time" application,
- need at least 35 frames/second
- OK to mess up rarely.
- iteration of 2 steps:

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

split frame into n parts &
assign work to n threads

```
Barrier b = new SomeBarrier( ... );

Int me = ThreadId.get()
while (true) {
    frame[me].prepare();
    b.await() //synchronize with others
    frame[me].display();
}
```

Example:

parallel discrete event simulation

- model: system behavior = series of events;
- event = update system state
+ deduce future events !
- use threads for:
 - split system into reasonable parts
 - have work done in parallel

event-driven execution

- **eventlist:**
 - list of future events, in ascending time-stamp order;
 - every thread has its own specific list
- **algo:**
 - while (myEventlist not empty)**
 - dequeue & process first event;**

parallel discrete event simulation

- problem:

threads generate events for **other** threads =>
difficult to maintain temporal/causal order

- solution:

there is some lookahead value Δ so that:

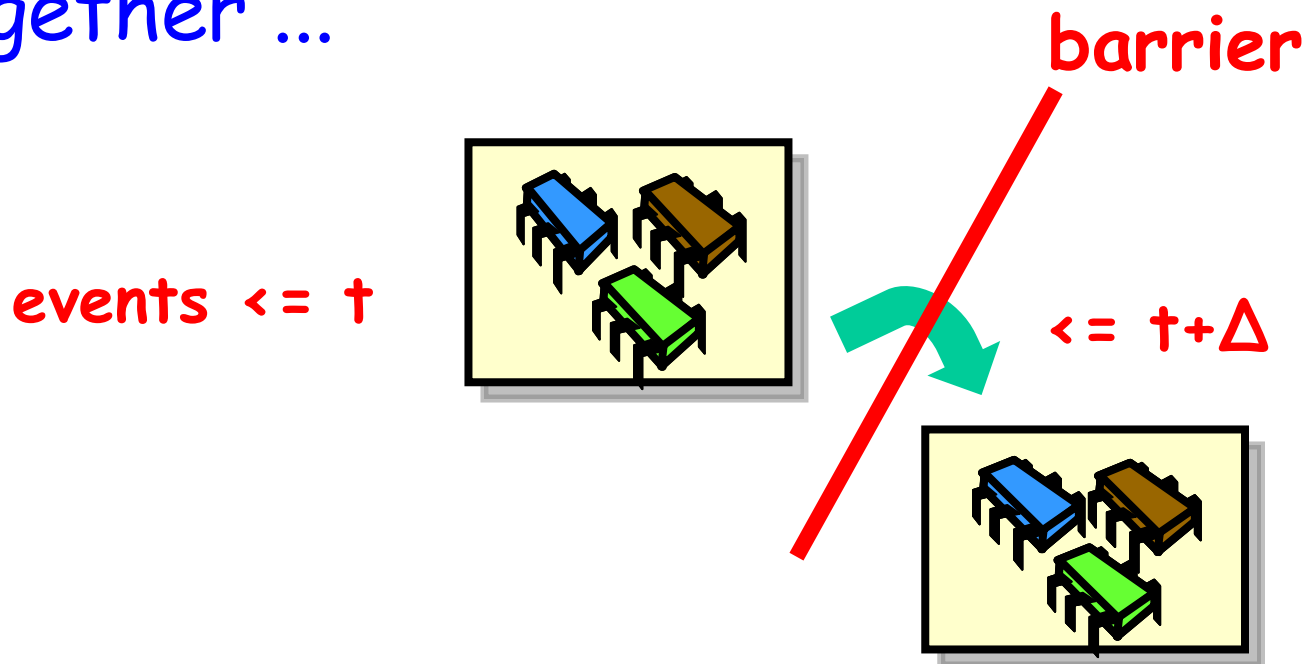
if thread has seen all events up to time t ,

thread can produce events up to **time $\leq t + \Delta$** ;

Δ may vary, as needed by threads

parallel discrete event simulation

threads agree on next safe time horizon and
move together ...



Barrier implementation aspects

- Cache coherence
 - Spin on locally-cached locations?
- Latency
 - How many steps on notification?
- Symmetry
 - Do all threads do the same thing?

Barrier:

simplistic implementation

```
public class SimpleBarrier {
    AtomicInteger count; // # threads not arrived
    int size;           // # participants

    public SimpleBarrier(int n){
        count.set(n); size = n;
    }
    public void await() {
        if (count.getAndDecrement()==1)
            //last arrival; count=0;
            count.set(size); //re-init barrier
        else
            while (count.get() != 0) {} //wait
    }
}
```

SimpleBarrier:

What's wrong with this ?

- iteration is common with barriers:

```
While ( ... ) {  
    doSomething( ... );    //do own part work  
    b.await() }           //synch with others
```

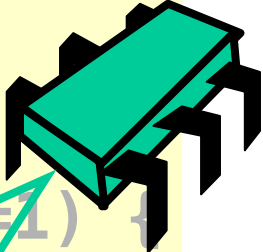
- but then:

last thread to arrive may re-increase
count before waiting threads see
count==0

Barriers

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

Waiting for Phase 1 to finish

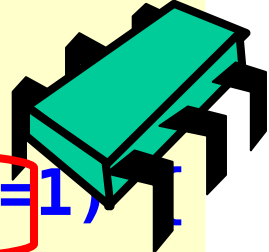
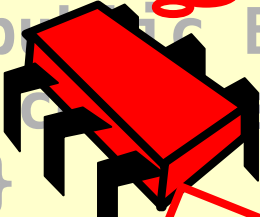


Barriers

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

Phase 1 is so over

waiting for Phase 1 to finish



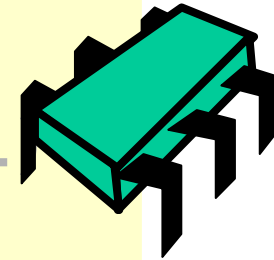
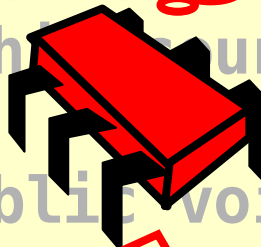
The code snippet shows a barrier implementation. The `await()` method uses `count.getAndDecrement()` to check if it's the last thread to reach the barrier. If so, it resets the count to `size`. Otherwise, it enters a `while` loop that waits until `count.get() != 0`.

Barriers

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

Prepare for phase 2

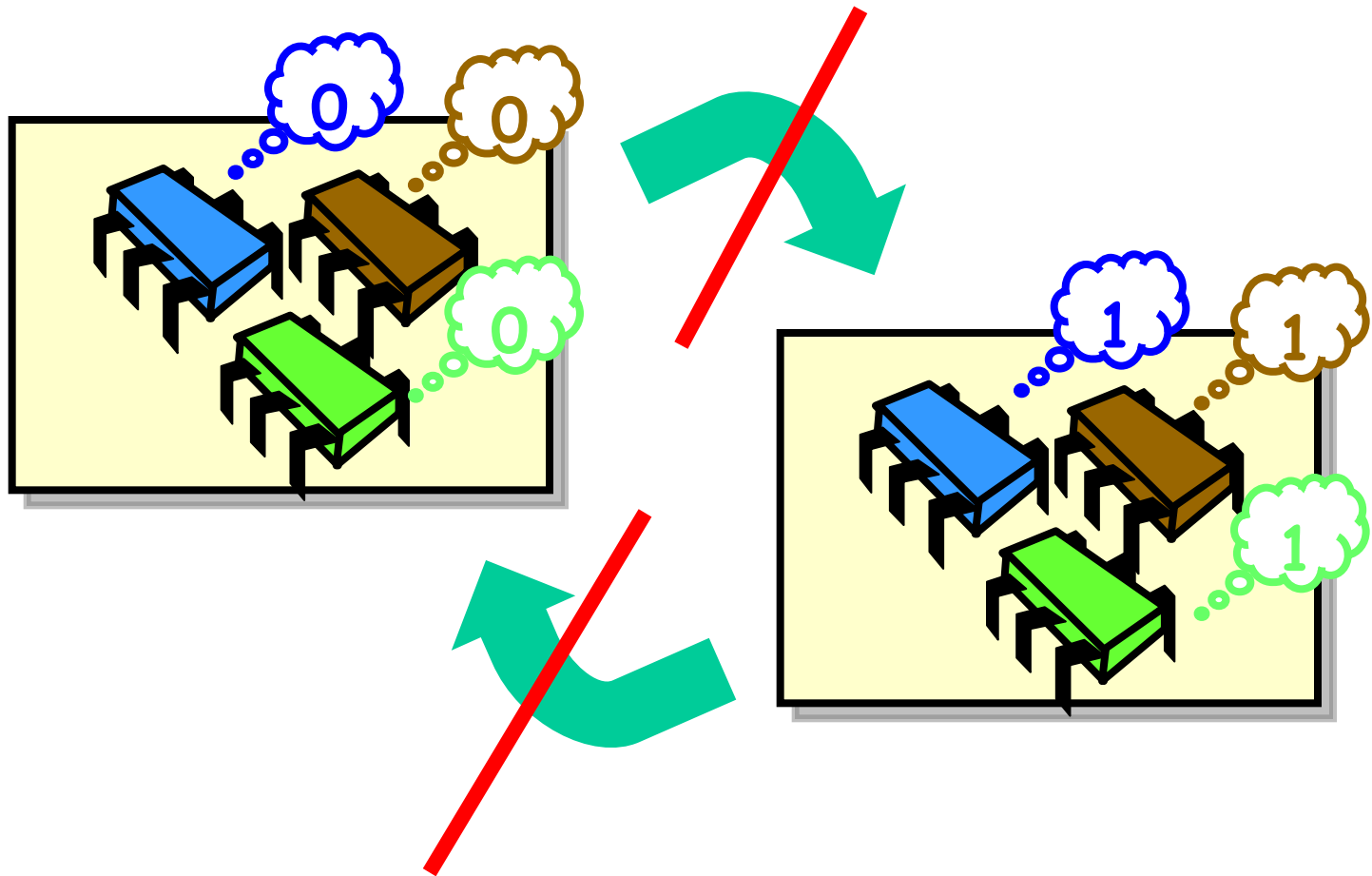
zzzzz....



So...

- one thread “wraps around” to start phase 2
- while another thread is still waiting for phase 1 to finish!
- obvious solution:
use two barrier objects and alternate!

two barrier objects:
correct but expensive ...



more economic idea: Sense-Reversing Barriers

```
public class SenseBarrier {
    AtomicInteger count;
    int size;
    boolean sense = true;    //odd or even numbered phase?

    SenseBarrier(int n){...} //constructor, as before

    public void await(boolean mySense) {
        //invariant: for all threads, mySense != sense
        if (count.getAndDecrement() == 1) {
            count.set(size);
            sense = mySense;}
        else
            while (sense != mySense) {};
        //on return, threads must change their sense !!!!!!!
    }}}
```

Sense-Reversing Barriers

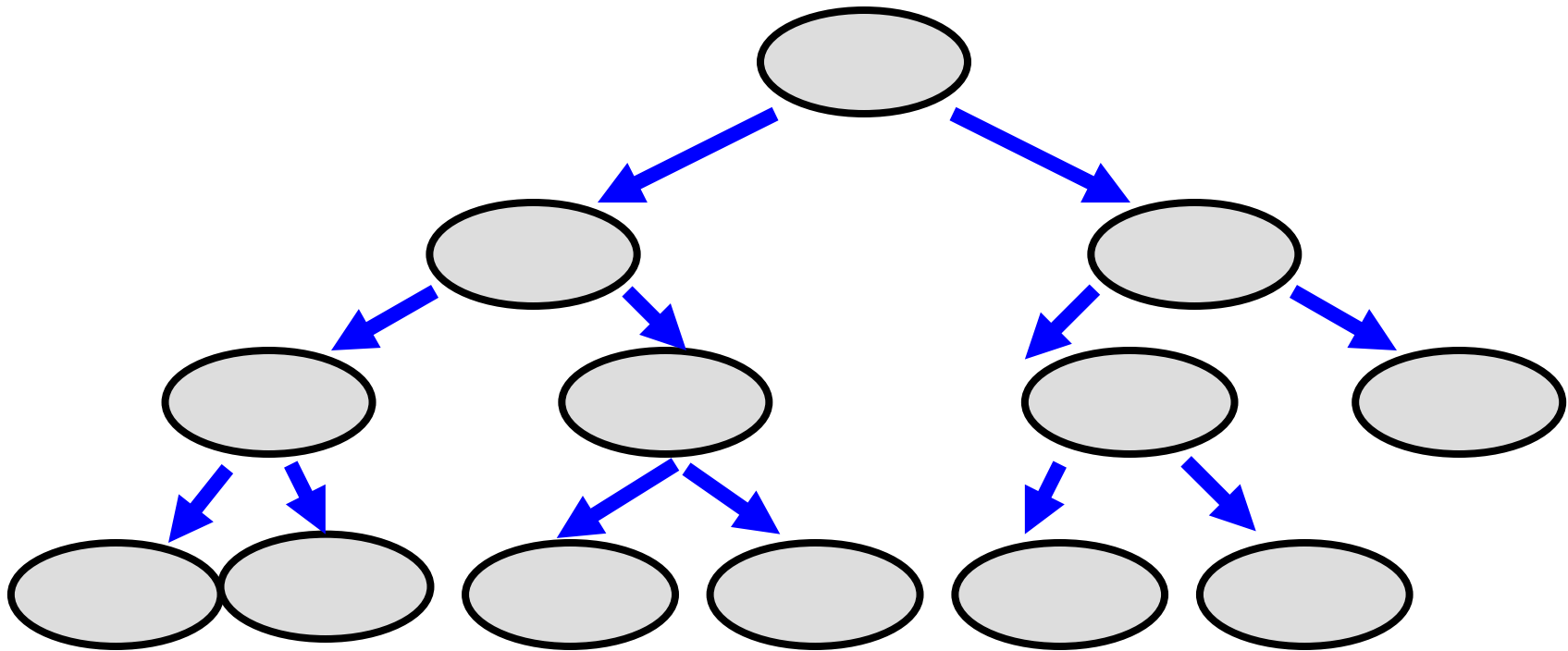
- **spinning is on sense-field:**
 - good for cache coherent architectures: use locally cached copies,
 - only change is when leaving the barrier !
- **possibly contention:**
 - count variable

Tree Barriers

- better for NUMAs, invented for general distributed systems;
- do not rely on single shared variable like "AtomicInteger count"
- reduce communication by organizing threads into a **tree** (e.g. binary)

Tree Barrier

nodes = threads



Tree Barrier algorithm:

- thread t informs parent thread (node) when
 - t itself has reached `await(...)` statement
 - the same is true for t 's children
- root detects completion of barrier and signals next phase

Tree Barrier algorithm:

technique:

on cache-coherent machine, use global
"boolean sense" variable for spinning
otherwise propagate info down tree

$O(\log \#threads)$ steps to go up tree (and
down again, if necessary..)

java.util.concurrent:

- class `CountDownLatch`
for one-time use
- class `CyclicBarrier`
can be re-used after thread release
optional `Runnable` to be executed after all
threads have arrived but before they are
released again

new subject:

Termination Detection

- *not just a special case of "barrier"*:
 - a thread may repeatedly finish its work and then get more from other threads ;
 - so a thread may alternate between active and inactive state;
 - termination is when all activity has died down (so that no thread will be made active again)

Example:

parallel discrete event simulation

- threads process events to modify state and produce future events;
- so a thread may run out of events
 - empty event list => become inactive
- and later get new event notices from other threads
 - non-empty event list => become active again

termination detection, how to:

- use global variable
 - e.g. count on unprocessed events in simulation example
- distributed system
 - organize threads into a tree-structure
 - keep track of messages in transit, i.e. sent but not consumed

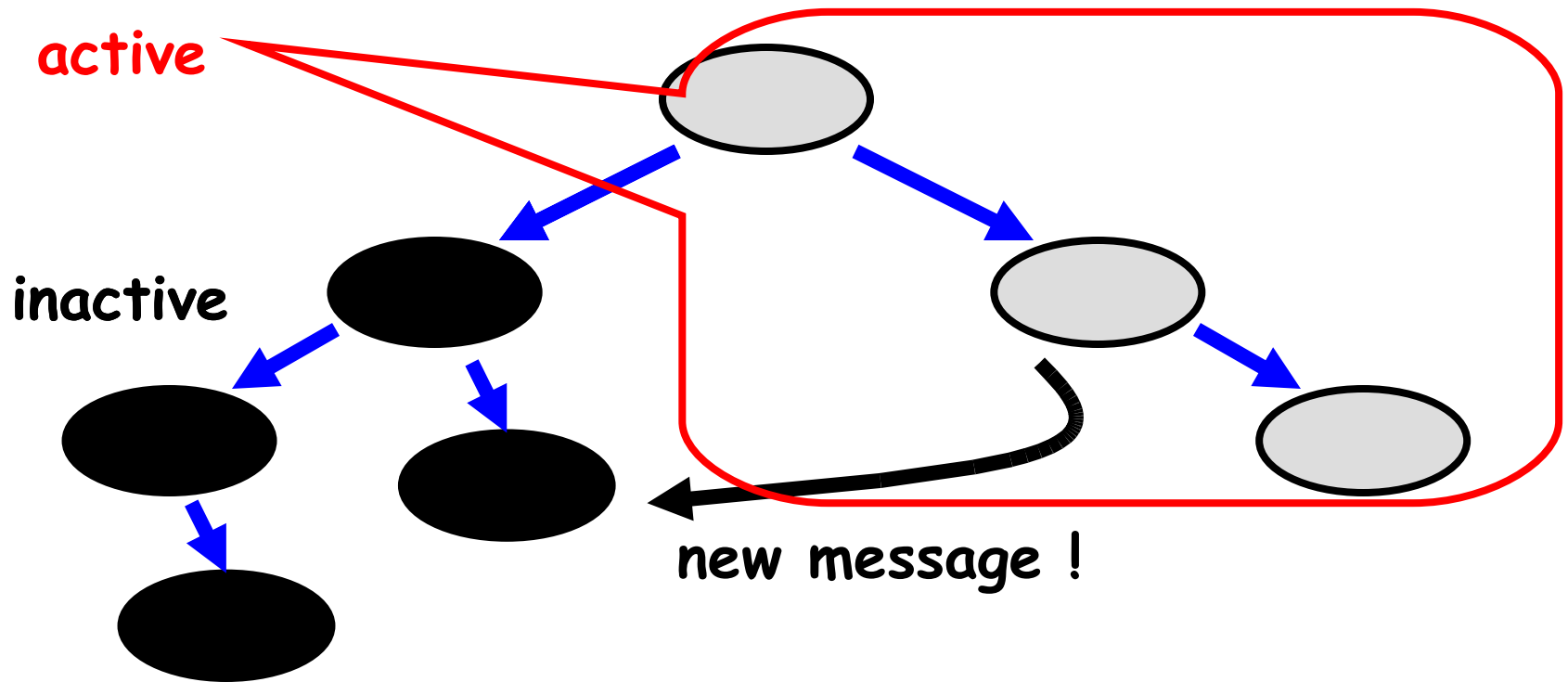
(here, message = piece of work to be done by other thread, e.g. event in simulation)

termination detection, distributed system:

tree of threads:

- **parent node** = thread that **activated** child by sending **first** message to work upon;
- maybe artificial starter node necessary;
- thread terminates if all its children have terminated
- **but:** maybe some message is on the way ..

termination detection, distributed system:



termination detection, distributed system:

- **accounting for messages:**

to become inactive,

sender waits for all its sent messages to
be acknowledged

receiver acknowledges received
messages to their sender, with the
**activating message as last thing at
own termination!**

termination detection, distributed system:

details & proof:

E.W.Dijkstra, C.S.Scholten

"Termination detection for diffusing
computations".

Information Processing Letters 11(1), 1-4, 1980

This lecture uses and adapts Maurice Herlihy's slides for “The Art of Multiprocessor Programming”, M.Herlihy & N.Shavit, 2008, which are available at <http://www.elsevierdirect.com/companion.jsp?ISBN=9780123705914> and licensed under Creative Commons Attribution-ShareAlike 2.5 License. This license shall apply to my adaption also.

You are free:

to **Share** — to copy, distribute and transmit the work

to **Remix** — to adapt the work

Under the following conditions:

Attribution. You must attribute the work to “The Art of Multiprocessor Programming” (but not in any way that suggests that the authors endorse you or your use of the work).

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license.

For any reuse or distribution, you must make clear to others the license terms of this work

. The best way to do this is with a link to

<http://creativecommons.org/licenses/by-sa/3.0/>.

Any of the above conditions can be waived if you get permission from the copyright holder.

Nothing in this license impairs or restricts the author's moral rights.