CSEP 524: Parallel Computation
(week 4)

Brad Chamberlain

Tuesdays 6:30 – 9:20

MGH 231

Pthreads vs. Chapel

Categorizing Pthreads and Chapel (Generated Dynamically in-class)

	C+Pthreads	Chapel
degree of voodoo	less voodoo	more voodoo
useful abstractions	more HW-oriented	more problem-oriented
verbosity	more verbose	less verbose
control of memory	more control due to C	less control (today)
HW independence	less abstracted from HW	more abstracted…
portability	quite good	potentially more portable
libraries	lots of existing library support	almost none currently
opportunities for error	more opportunities due to C and details of sync primitives	less so
notation	library	language
maturity	very mature	much less so
classic concepts	the set of classic concepts	pretty significant departure
completeness	confidence that it's complete	unclear

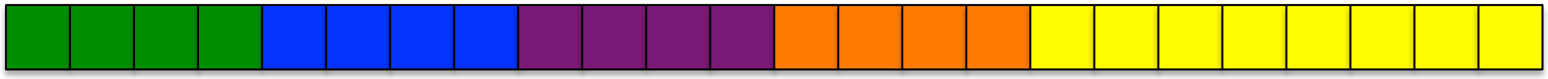
Alternatives to C+Pthreads (Shared Memory Multithreaded Programming)

- Java/C# Threads
- Emerging C/C++ constructs for parallelism
- User-level threading libraries
  + lighter-weight implementation
  – interfaces probably similarly complex
  – some examples:
    - **Qthreads (Sandia):** supports full-empty variables
    - **MassiveThreads (U Tokyo):** pthreads interface w/ user-level impl.
    - **Nanos++ (BSC):** data-driven parallelism
- Intel TBB, Microsoft TPL, [Intel] Cilk, …
- OpenMP (next week)

Unbalanced Block Distributions

Last week's discussion about Block strategies

- A Block strategy I was critical/skeptical of last week:
  - floor(n/p) to the first p-1 tasks; remainder to final task
  - e.g., for n = 24, p = 5, task 5 gets 2x the work as 1-4



  - More generally, how bad is it?
    - the max difference in items it can get is p-1 more than the others
    - as a fraction of overall problem size, that's (p-1)/n more work
    - for small p, as n goes to a very large number, this is insignificant

Impact of Unbalanced Distributions

- Consider a 32 GB, 16-core node
- Fill half the memory with a 1D array of real(64)s:
  - n = 2147483647, p = 16
    - ⇒ floor(n/p) = 134,217,727 items
    - ⇒ remainder = 134,217,742
    - ⇒ (essentially the identical amount of work)

- (analogy between my reaction and my 6-year-old's)

- But, this is not the only way to fill memory…

Impact of Unbalanced Distributions

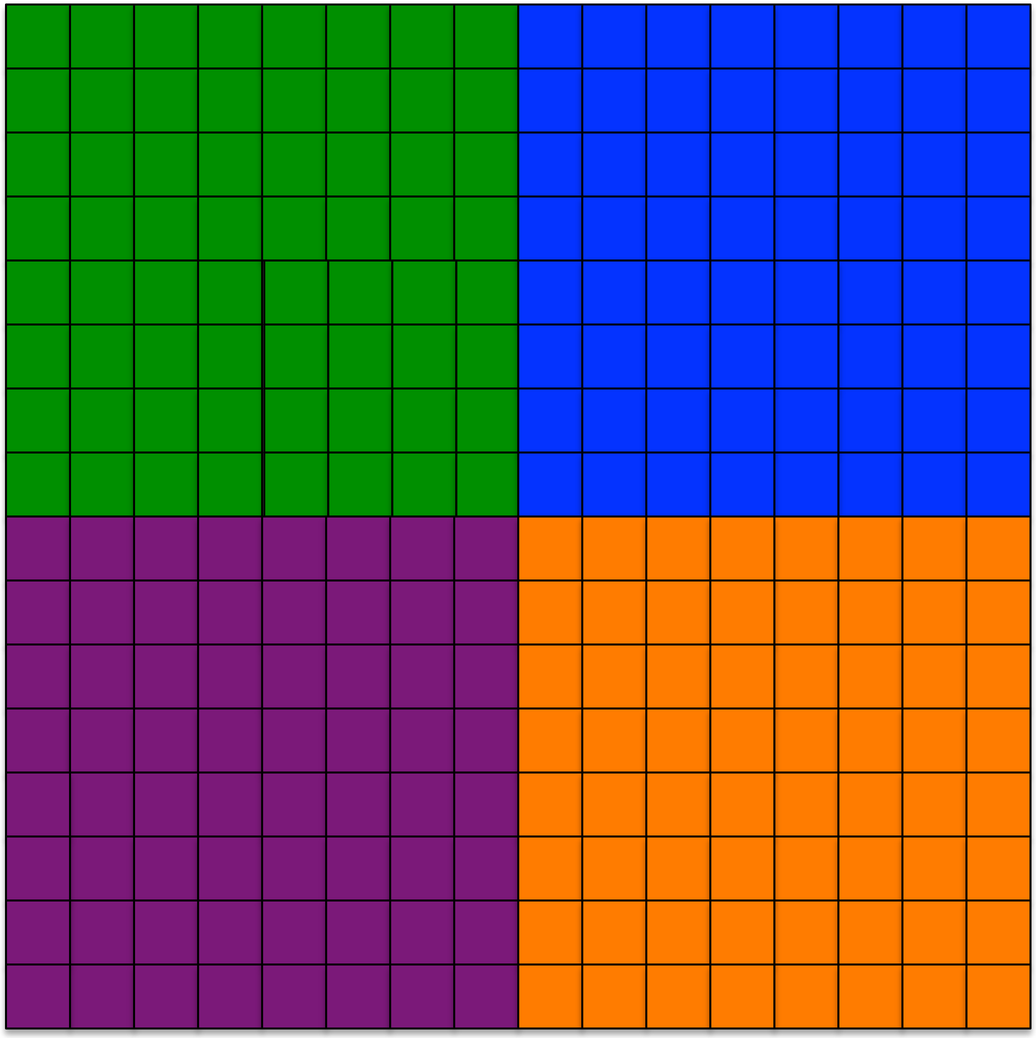
- Consider a 32 GB, 16-core node
- What if we fill mem. using an array of arrays instead...
  - **var** Data: [1..26] [1..82595525] **real**;

...and distributed only the outer array by block?

  - n = 26, p = 16
    - ⇒ floor(n/p) = 1 items
    - ⇒remainder = 10 extra items
    - ⇒ one task gets 11x the amount of work/data

- Now, arguably we should've distributed the work on the inner arrays instead/as well
  - but maybe we couldn't parallelize them or were too lazy or...

Recall Multidimensional Distributions: 2D Block x Block (distributed to 2x2 tasks)



Impact of Unbalanced Distributions

- Consider a 32 GB, 16-core node
- And a 3D array w/ a multidimensional distribution
  - e.g., var Data: [1..1291, 1..1291, 1..1291] real;
  - n = 1291, p = 4 x 4 x 1
    - ⇒ floor(n/p) = 322 x 322 x 1291 = 133,856,044 items
    - ⇒ remainder = 3 x 3 x $\cancel{1}$ 0 extra items = ~~136,467,500~~ 136,361,875 items
    - ⇒ one task gets ~~~2%~~ 1.8% more work/data

Oops – I changed my 'p' dimensions at some point but forgot to update my remainders. The red text fixes my mistake. 1/31

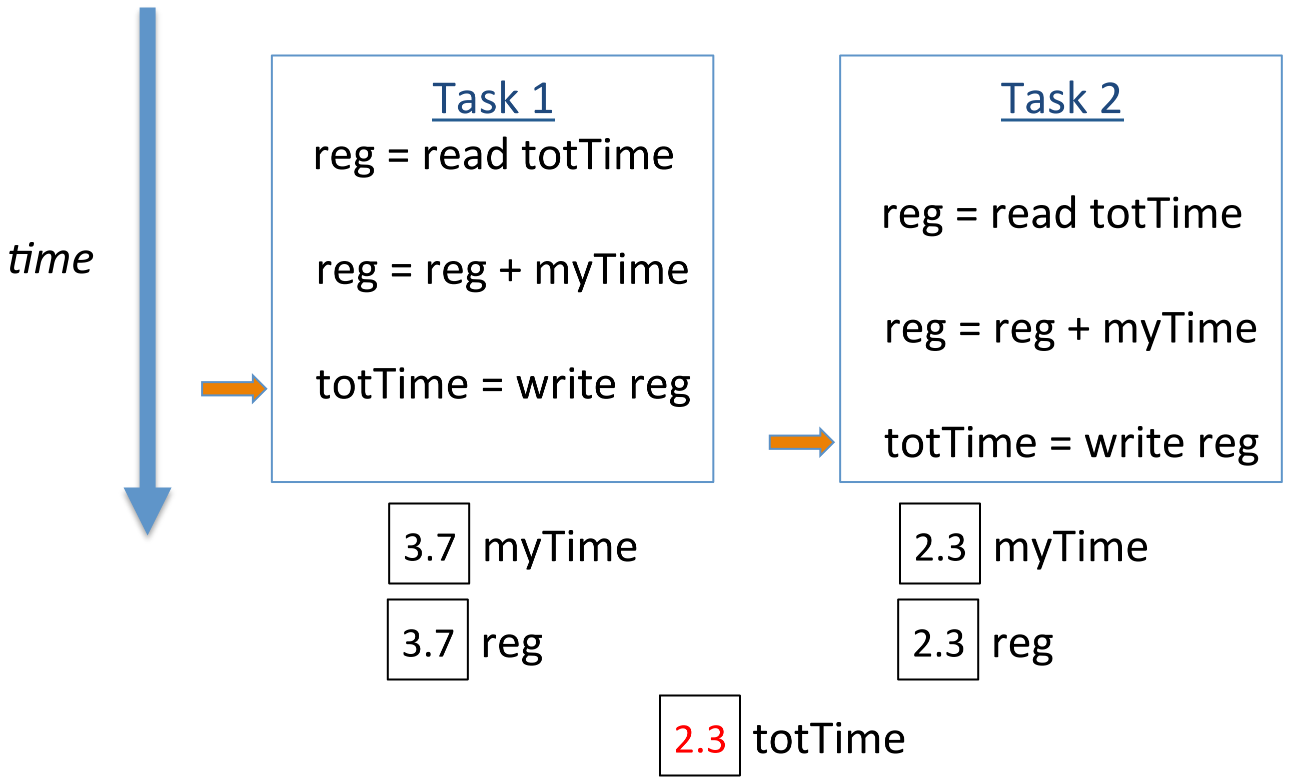
- Distributing across a distributed memory machine where p is 18,688 or 299,008, the remainders and therefore imbalance could be even more significant…

Locks and RRWW Bugs…
Where Were We?

Recap: RRWW Bugs

- The following schedule is problematic:

executing “totTime += myTime;” in parallel…



Task 1
```
reg = read totTime
reg = reg + myTime
totTime = write reg
```

Task 2
```
reg = read totTime
reg = reg + myTime
totTime = write reg
```

Fixing RRWW bugs with locks

Pthreads

```
pthread_mutex_t totTimeMutex;
pthread_mutex_init(&totTimeMutex, NULL);

create tasks
  …
  pthread_mutex_lock(&totTimeMutex);
  totTime += myTime;
  pthread_mutex_unlock(&totTimeMutex);
  …
join tasks

pthread_mutex_destroy(&totTimeMutex);
```

Chapel

```
var totTime$: sync real = 0.0;

coforall tid in 0..#numTasks {
  …
  totTime$ += myTime;
  …
}
```

Pitfalls of Using Locks I

Deadlock: Can occur when grabbing the same locks in different orders

```
cobegin {
  {  // task 1
    const val = lock1$;
    lock2$ += 1;
    lock1$ = val + 1;
  }
  {  // task 2
    const val = lock2$;
    lock1$ += 1;
    lock2$ = val + 1;
  }
}
```

Task 1	Task 2
val = lock1$.readFE();	
	val = lock2$.readFE()
tmp = lock2$.readFE();	
	tmp = lock1$.readFE()
(…blocks…)	
	(…blocks…)
(Zzzzz….)	(Zzzzz….)

Pitfalls of Using Locks I

Deadlock: Similarly trivial example in pthreads

```
{   // task 1
  pthread_mutex_lock(&lock1);
  pthread_mutex_lock(&lock2);
  pthread_mutex_unlock(&lock2);
  pthread_mutex_unlock(&lock1);
}
{   // task 2
  pthread_mutex_lock(&lock2);
  pthread_mutex_lock(&lock1);
  pthread_mutex_unlock(&lock1);
  pthread_mutex_unlock(&lock2);    }
}
```

Pitfalls of Using Locks II

Livelock:

- Tasks are still executing…
- …but not making useful progress

Livelock Example

```
var lock1$, lock2$, lock3$: sync int;   // all start empty
lock1$ = 1;                             // fill lock 1
cobegin {
  {  // task 1
    do {
      var val = lock1$,
          val2 = 0;
      if (lock2$.isFull) { val2 = lock2$; lock3$ = val2 + 1; }
      lock1$ = val + 1;
    } while (val2 == 0);
  }
  {  // task 2
    do {
      var val = lock1$, val2 = 0;
      if (lock3$.isFull) { val2 = lock3$; lock2$ = val2 + 1; }
      lock1$ = val + 1;
    } while (val2 == 0);
}   }
```

Writing Deadlock-Free Lock Code

One technique:

- when requiring multiple locks, take them in a specific order
  - e.g., "always take lock1 before lock2"
- then release them in the opposite order
- ensures that you'll never enter deadlock

- yet this is not a foolproof solution…

Problems with Taking Locks in Sorted Order

1) Sometimes you can't predict the locks you'll need

– e.g., if the locks are determined by dynamic values

```
var A$: [1..numItems] sync real =0.0;// synchronized data array
const i = infile.read(int);          // read an unknown index

const val = A$[i];                   // grab its value & lock

const j = someBigComputation(val);   // compute some other index
const val2 = A$[j];                  // grab its value & lock

A$[j] = val;                         // swap the values,
A$[i] = val2;                        // ...releasing the locks

/* If we can't determine the relative ordering of i and j
   a priori, we can't rely on this code to be safe */
```

Problems with Taking Locks in Sorted Order

2) Sometimes you may not realize what locks you're taking

- e.g., if you're calling into library code that takes locks
  - a good library would presumably document such behavior
  - but will users think deeply enough to take such issues into account?
  - arguably the issue interferes with the black box benefit of libraries

Writing Deadlock-Free Lock Code

3) Use atomic operations

Concept:

- never block
- instead, ensure no other task can see intermediate state
  - analogy to databases…

we'll come back to this later…

Pitfalls of Using Locks III (and of Shared Memory Parallelism more generally)

- Memory consistency model impacts?
  - a problem with library-based locks in traditionally sequential languages
    - e.g., C+Pthreads, as documented in Boehm paper
  - even in designed-to-be-parallel languages, something to be wary of
    - e.g., Java, C#, Chapel, etc.

(Note: C/C++ are evolving to address this by becoming parallel-aware)

Memory Consistency Models (“MCMs”)

Memory Consistency Models in a Nutshell

Memory Consistency Model:

Memory Consistency Models in a Nutshell

Memory Consistency Model: Rules that define how distinct tasks may view concurrent updates to memory.

Strict Consistency

- All reads/writes to memory are viewed in a globally consistent order by all tasks
  - i.e., intuitively, exactly what you would like
  - e.g., imagine all memory ops went over a single wire or were guarded by a single lock
  - by definition, different tasks couldn't have simultaneous contradictory notions of memory
  - but, practically speaking, this is untenable
    - kills parallelism across tasks by making memory a bottleneck
    - also, destroys architectural benefits of multiple memory banks
    - i.e., it could be done, but not without sacrificing performance to the extent that you might as well have stayed in a sequential language

Motivating Example (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2

reg1 = x
y = 1
	reg2 = y
	x = 2

What could reg1 and reg2 hold at this point?

Motivating Example (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	
	reg2 = y
y = 1	
	x = 2

reg1 = 0, reg2 = 0

Motivating Example
(adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	
y = 1	
	reg2 = y
	x = 2

reg1 = 0, reg2 = 1

Motivating Example (adopted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
	reg2 = y
	x = 2
reg1 = x	
y = 1	

reg1 = 2, reg2 = 0

A Weaker Model: Sequential Consistency

- Two parts to the definition:
  - All memory ops within a task complete in program order
  - Across tasks, memory ops are interleaved in a consistent total order
- Intuitively: "An interleaving of the tasks' memops if they were instantaneous"
  - all of the preceding slides obeyed sequential consistency
- Not as ideal as strict, but still comprehensible
- Unfortunately, still untenable in general
  - guaranteeing a consistent, total order on memory ops again implies too much overhead

Difference Between Strict and Sequential

[Clarifying something I stumbled over in lecture after the fact]

In strict, the global total ordering matches that “which actually happened” according to some global clock

In sequential, the global total ordering is consistent, but may not reflect what happened w.r.t. some global clock if all memory ops were instantaneous

- i.e., it takes into account skew in clocks, time required for things to percolate through a system

Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

Sadly, yes – this can occur within most languages/architectures

Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the compiler" explanation:

- Traditionally, a compiler looks at a single task at a time
  (Practically speaking, it can't consider all possible potentially concurrent tasks)
- To a compiler looking at code in isolation, nothing prevents reordering as follows:

Code Snippet 1	Code Snippet 2
y = 1	x = 2
reg1 = x	reg2 = y

(at which point, obvious execution interleavings can yield the reg1 = 2, reg2 = 1 result).

Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

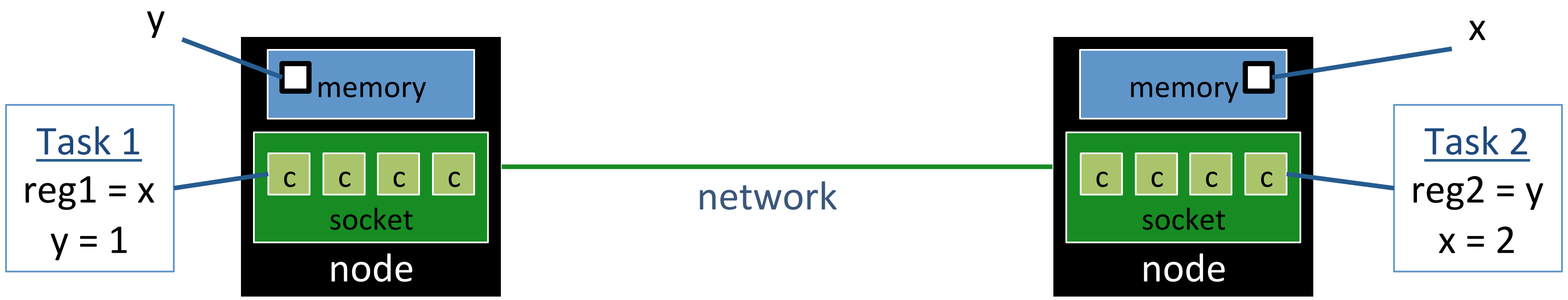
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors typically don't serialize memory ops (particularly on large-scale machines)
- In practice, independent memory ops can have different latencies / be reordered in HW
    - analogous to compiler situation: HW doesn't know about all other tasks



Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

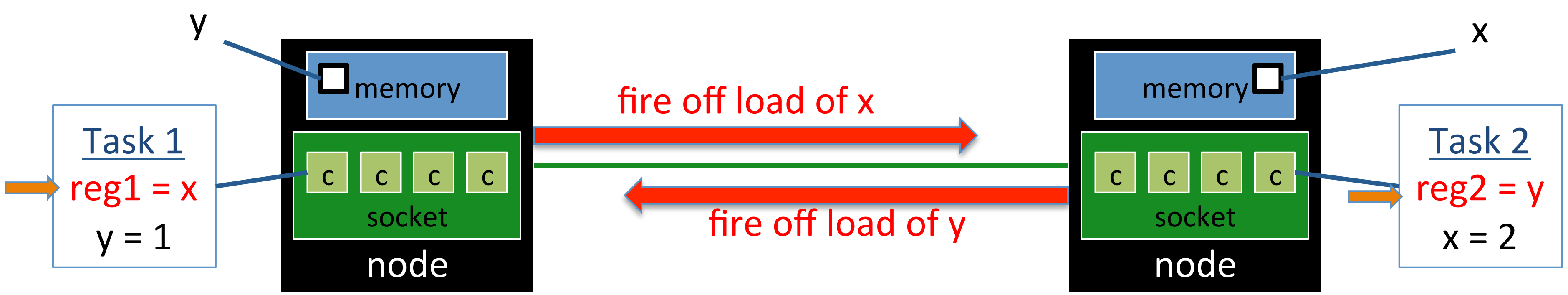
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors typically don't serialize memory ops (particularly on large-scale machines)
- In practice, independent memory ops can have different latencies / be reordered in HW
    - analogous to compiler situation: HW doesn't know about all other tasks



Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

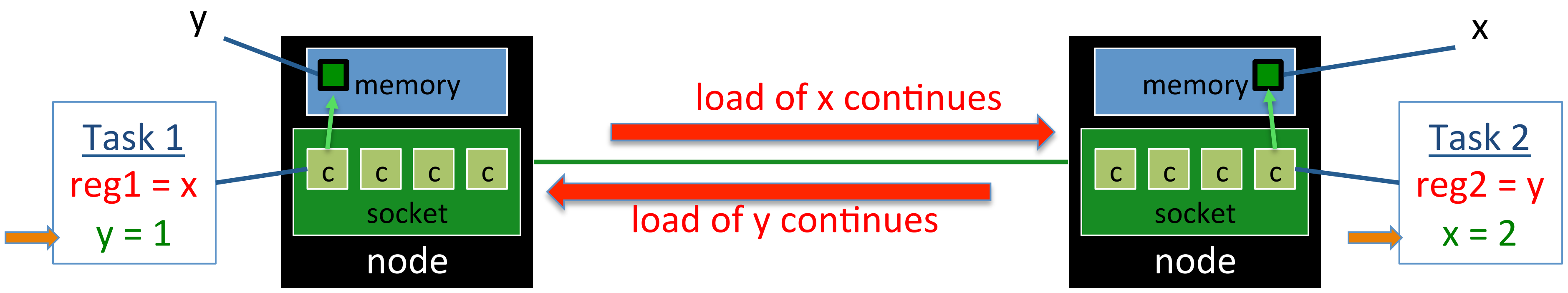
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors typically don't serialize memory ops (particularly on large-scale machines)
- In practice, independent memory ops can have different latencies / be reordered in HW
  - analogous to compiler situation: HW doesn't know about all other tasks



Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

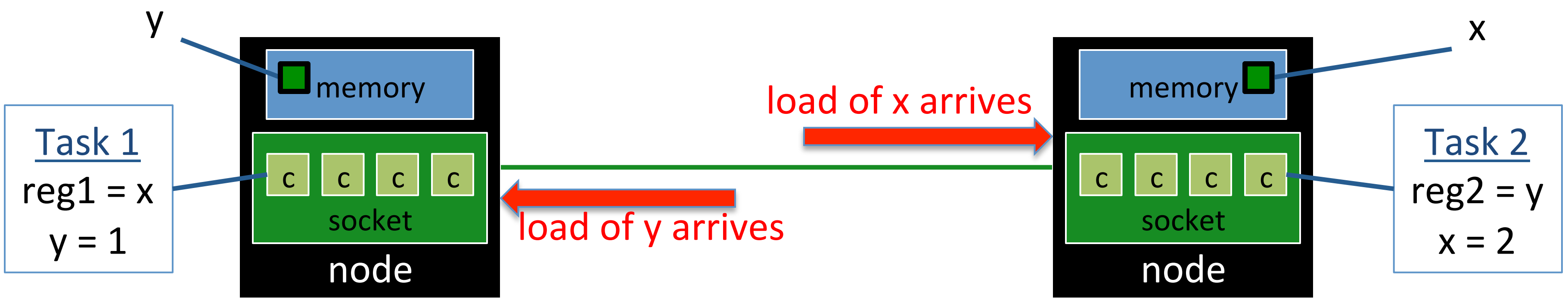
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors typically don't serialize memory ops (particularly on large-scale machines)
- In practice, independent memory ops can have different latencies / be reordered in HW
    - analogous to compiler situation: HW doesn't know about all other tasks



Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

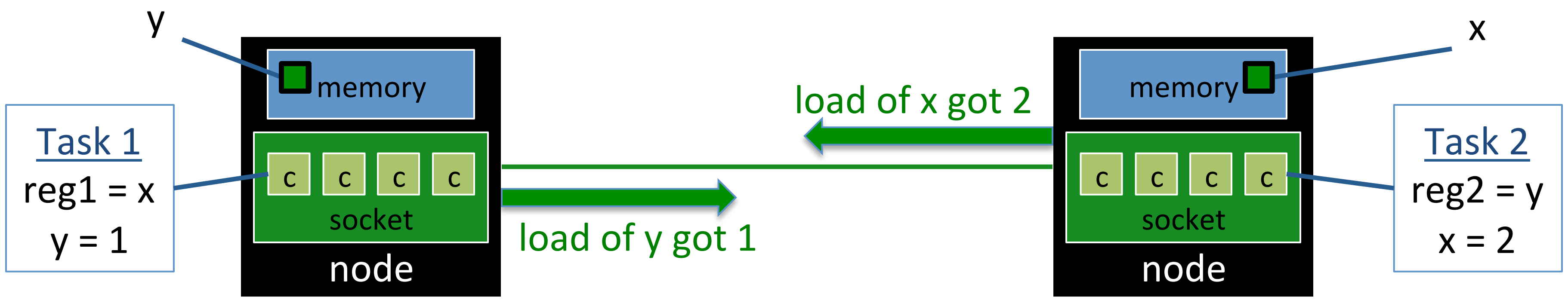
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors typically don't serialize memory ops (particularly on large-scale machines)
- In practice, independent memory ops can have different latencies / be reordered in HW
    - analogous to compiler situation: HW doesn't know about all other tasks



Motivating Example (adapted from *The Java MCM*: Manson, Pugh, Adve)

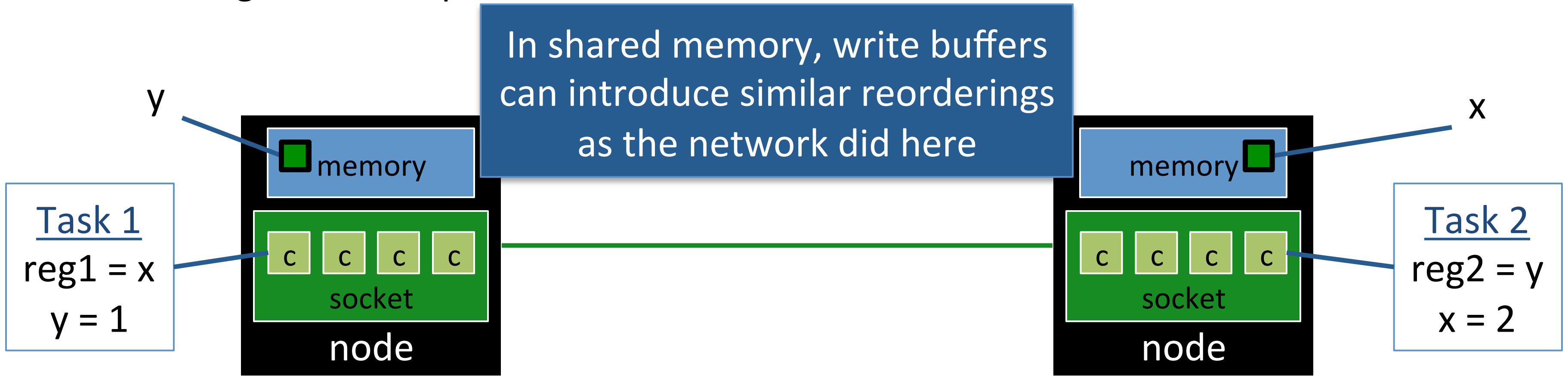
Initially, x == 0, y == 0

Task 1	Task 2
reg1 = x	reg2 = y
y = 1	x = 2

What about reg1 = 2, reg2 = 1 ?

The "blame the hardware" explanation:

- Processors typically don't serialize memory ops (particularly on large-scale machines)
- In practice, independent memory ops can have different latencies / be reordered in HW
  - analogous to compiler situation: HW doesn't know about all other tasks



In shared memory, write buffers can introduce similar reorderings as the network did here

Clarifying the "Inconsistency" in MCMs

[Here is something I stumbled over in lecture and got help clarifying at the happy office hour. Hopefully I've got them right now (he wrote at 12pm)]:

- As stated in lecture, MCMs are about whether or not two tasks have a consistent view of what happened
- In class, I was having trouble connecting the dots between the surprising reorderings as explained and different tasks seeing different things. Try this:
  - pretend you're task 1 (or the programmer who wrote it)
  - you know that according to your program order, you are loading x and then storing y; and that task 2 is loading y then storing x
  - upon getting the value of 2 for x, you reason that "Oh, task 2 must have run before me since I saw the updated x value"
  - meanwhile, task 2 does the symmetric thing: "Oh, task 1 must have run before me since I saw the updated y value"
  - the fact that we each conclude different orderings based on our observations suggests that our view of what happened w.r.t. memory is inconsistent

Relaxed/Weak Consistency Models

- Without getting into detailed definitions…
  - much less intuitive than strict/sequential consistency
  - but much more likely to be implemented/adopted in practice

- Several other models exist as well (beyond the scope of this course)

Possible Attitudes About MCMs

"Memory *what*?"

– Goal for this class: Get you beyond this point; make you aware of this important and complex topic

"It's dumb that we are living with this—let's fix it!"

– How much parallel performance are you willing to sacrifice?

– How much work to make compilers, architectures conform?

"This is complicated, I don't want to think about it!"

– Completely natural, but accept that this is part of life…

– As a parallel programmer, you ignore it at your peril

"I'll just write code without data races, so I'm good"

– True enough, if you can stick to that

MCMs and Data Races

data races:

- Uncoordinated accesses to shared memory by distinct tasks like we've been talking about here
- several memops to same location where 1 or more is a write

Note: Many languages that define memory consistency models do so only for programs without data races

- *i.e.*, "If your program contains a data race, all bets are off"
- Personally, this has always been a little discouraging to me

  (a) it'd be easy to have a data race and never realize it (so "blammo!"?)

  (b) intuitively, if you hit a data race, there are more and less likely things that will occur

  - but, in the defense of languages, how much can they really constrain/ define all future HW and compiler optimizations?

Preventing Data Races

- The key to preventing data races is *synchronization*
  - i.e., coordinate between tasks rather than having them race to memory independently of one another

- The specific synchronization mechanisms available and semantics they impose tend to be language-/compiler-specific

A Common Mechanism: Memory Fences

Memory Fences:

- Intrinsic operations that have specific semantics w.r.t. memory accesses
  - also known as memory barriers, or simply "fences", …
- Sometimes specified in language, other times by compilers
  - e.g., C, being sequential, hadn't defined one, so gcc did
- A typical example of a fence:
  - execution won't proceed until all outstanding loads/stores complete
  - compiler cannot reorder loads/stores across fence operations
- Specifics vary with implementation
- Granularity is an important issue to pay attention to
  - w.r.t. what subset of the hardware?
  - w.r.t. ops from the task or process or program or …?

Memory Consistency in Chapel

Two part story:

1) Traditional variables have a relaxed consistency model

```
var data: [1..size] int,
    flag = false;
cobegin {
  {                                    // task 1
    forall i in 1..size do             // write data
      data[i] = i;
    flag = true;                       // signal data written
  }
  {                                    // task 2
    while (!flag) do ;                 // spin on flag
    writeln("data is:", data);         // unsafe read!
  }
}
```

Memory Consistency in Chapel

Two part story:

2) Operations on sync/single variables imply a fence:

```
var data: [1..size] int,
    flag$: sync bool;
cobegin {
  {                                  // task 1
    forall i in 1..size do           // write data
      data[i] = i;
    flag$ = true;                    // signal data written
  }
  {                                  // task 2
    const flagval = flag$;           // block on flag$
    writeln("data is:", data);       // read will be safe
  }                                  // due to fence
}
```

Memory Consistency Models in Adopted Languages

Java: the first major language to adopt one?

- circa 2005 (?)
- the touchstone of language-based MCMs
- arguably overkill for other languages due to security requirements

C/C++: Playing catch-up – adopted in C11/C++11

- Defined reasonably similarly for both

C#: Seems to have one, I'm not familiar with the history

Resources For Further Reading

Shared Memory Consistency Models: A Tutorial

- Sarita V. Adve ("Queen of MCMs"), Kourosh Gharachorloo
- http://www.hpl.hp.com/techreports/Compaq-DEC/WRL-95-7.pdf

The Java Memory Model

- Jeremy Manson, William Pugh, Sarita Adve
- http://dl.dropbox.com/u/1011627/journal.pdf (paper)
- http://cseweb.ucsd.edu/classes/fa05/cse231/Fish.pdf (slides)
- See also: http://www.cs.umd.edu/~pugh/java/memoryModel/ (resources)

Foundations of the C++ Concurrency Memory Model

- Hans Boehm, Sarita Adve
- http://www.hpl.hp.com/techreports/2008/HPL-2008-56.html
- See also: http://www.hpl.hp.com/personal/Hans_Boehm/c++mm/

The C# Memory Model in Theory and Practice

- http://msdn.microsoft.com/en-us/magazine/jj863136.aspx

Discuss Boehm Paper Here

Performance Impacts of Locks

Fixing RRWW bugs with locks

Pthreads

```
pthread_mutex_t totTimeMutex;
pthread_mutex_init(&totTimeMutex, NULL);

create tasks
  …
  pthread_mutex_lock(&totTimeMutex);
  totTime += myTime;
  pthread_mutex_unlock(&totTimeMutex);
  …
join tasks

pthread_mutex_destroy(&totTimeMutex);
```

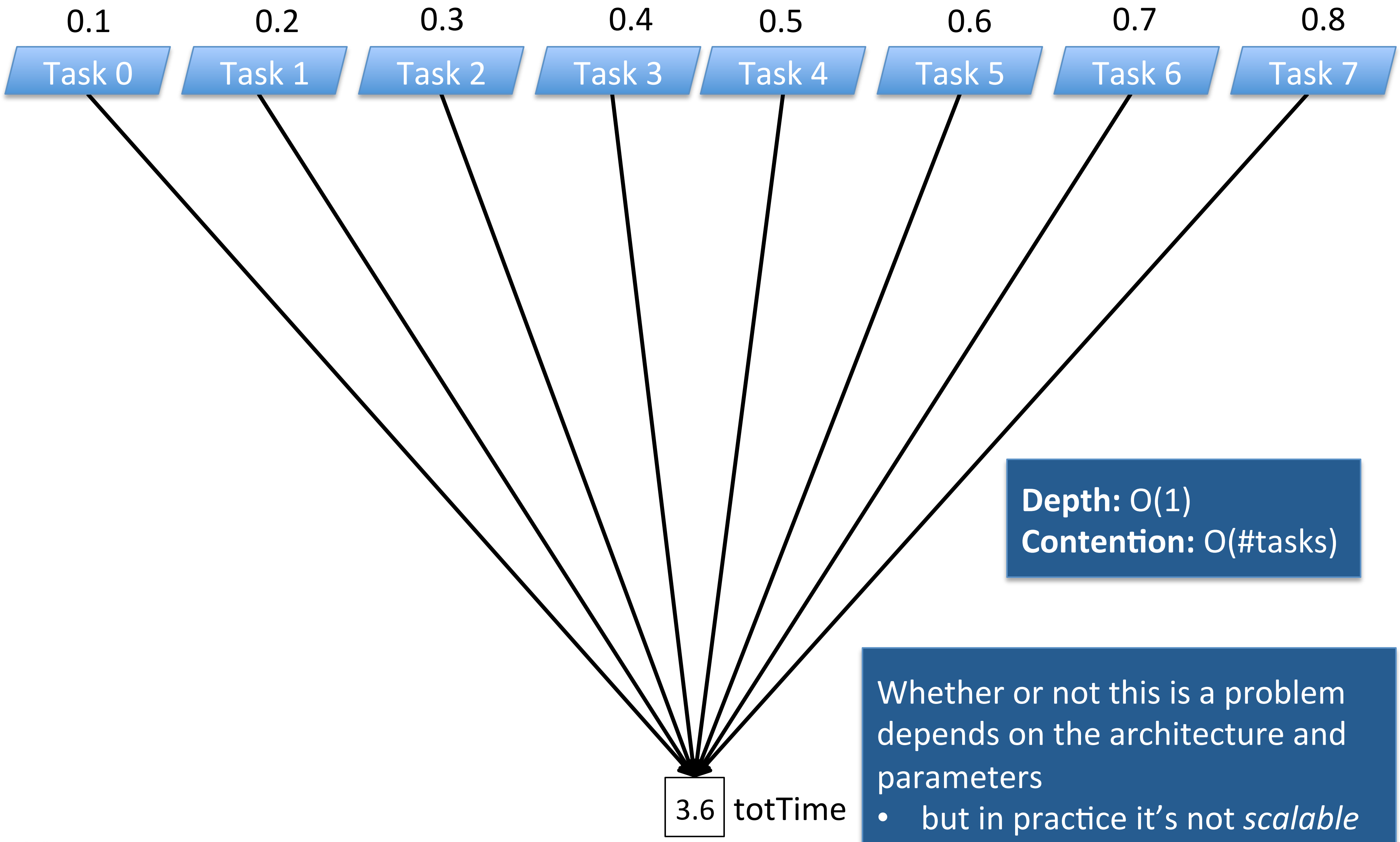
Chapel

```
var totTime$: sync real = 0.0;

coforall tid in 0..#numTasks {
  …
  totTime$ += myTime;
  …
}
```

What's the performance problem with these codes as we increase the number of tasks?

The Problem: totTime becomes a bottleneck

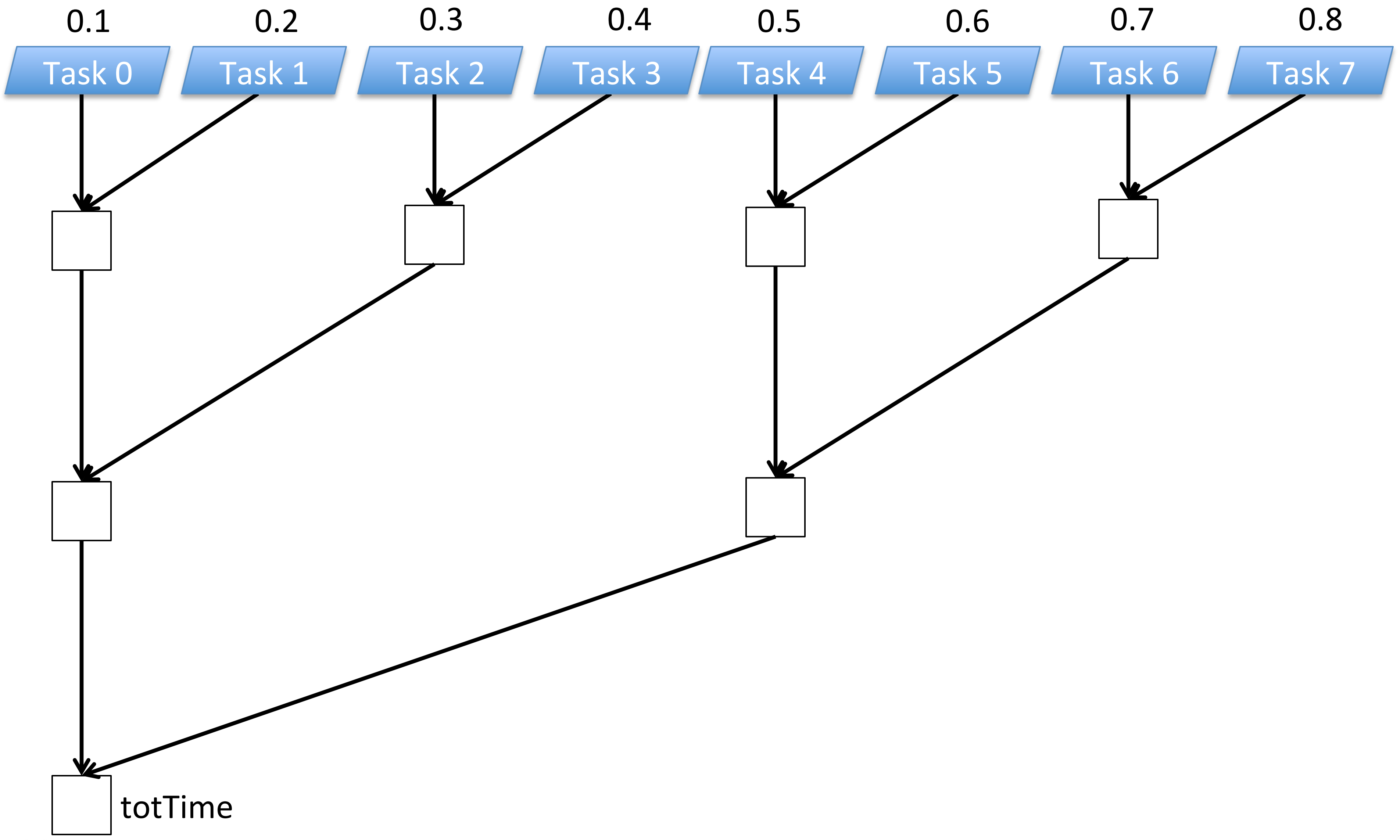


Depth: O(1)
Contention: O(#tasks)

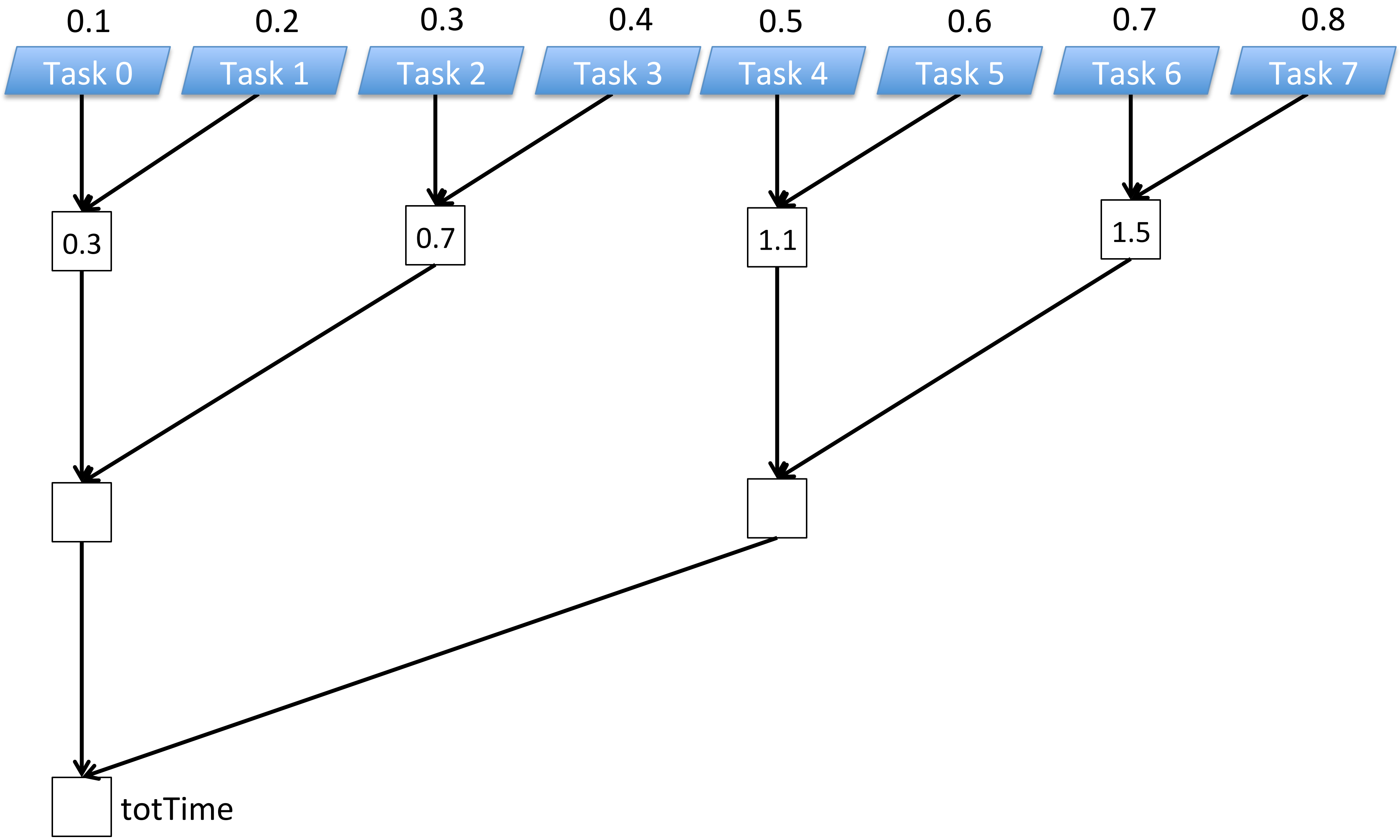
Whether or not this is a problem depends on the architecture and parameters

- but in practice it's not *scalable*

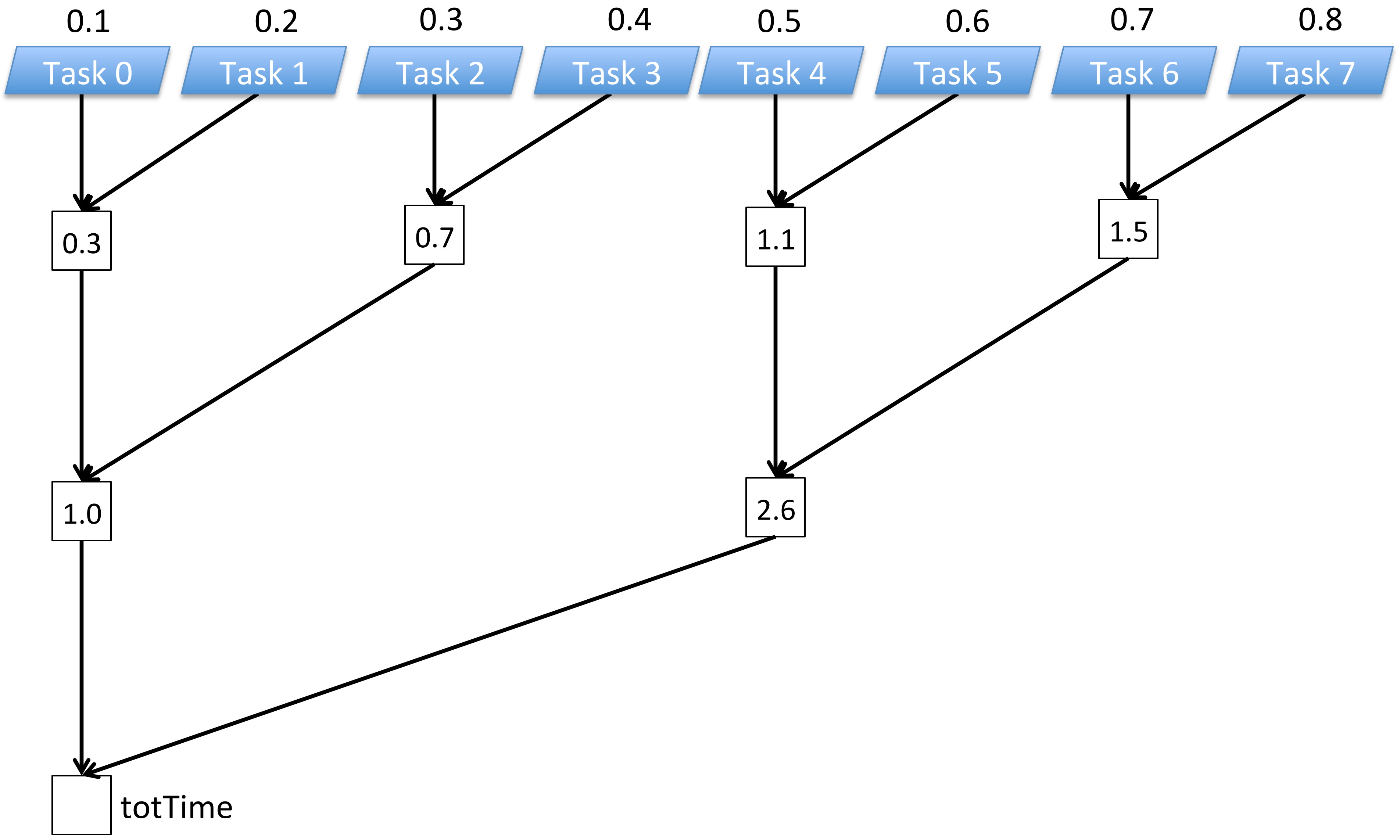
Fix: Use a *Reduction*



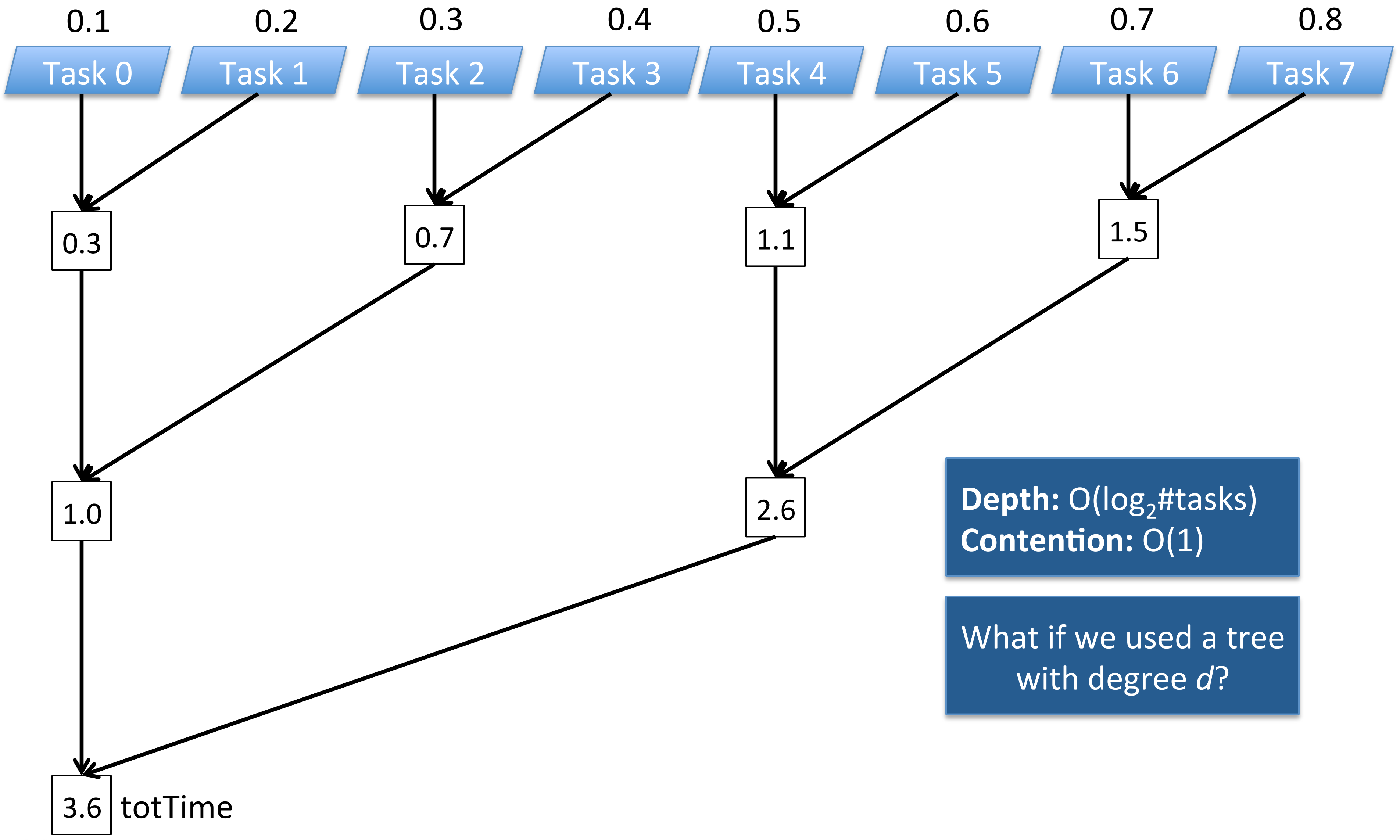
Fix: Use a *Reduction*



Fix: Use a *Reduction*



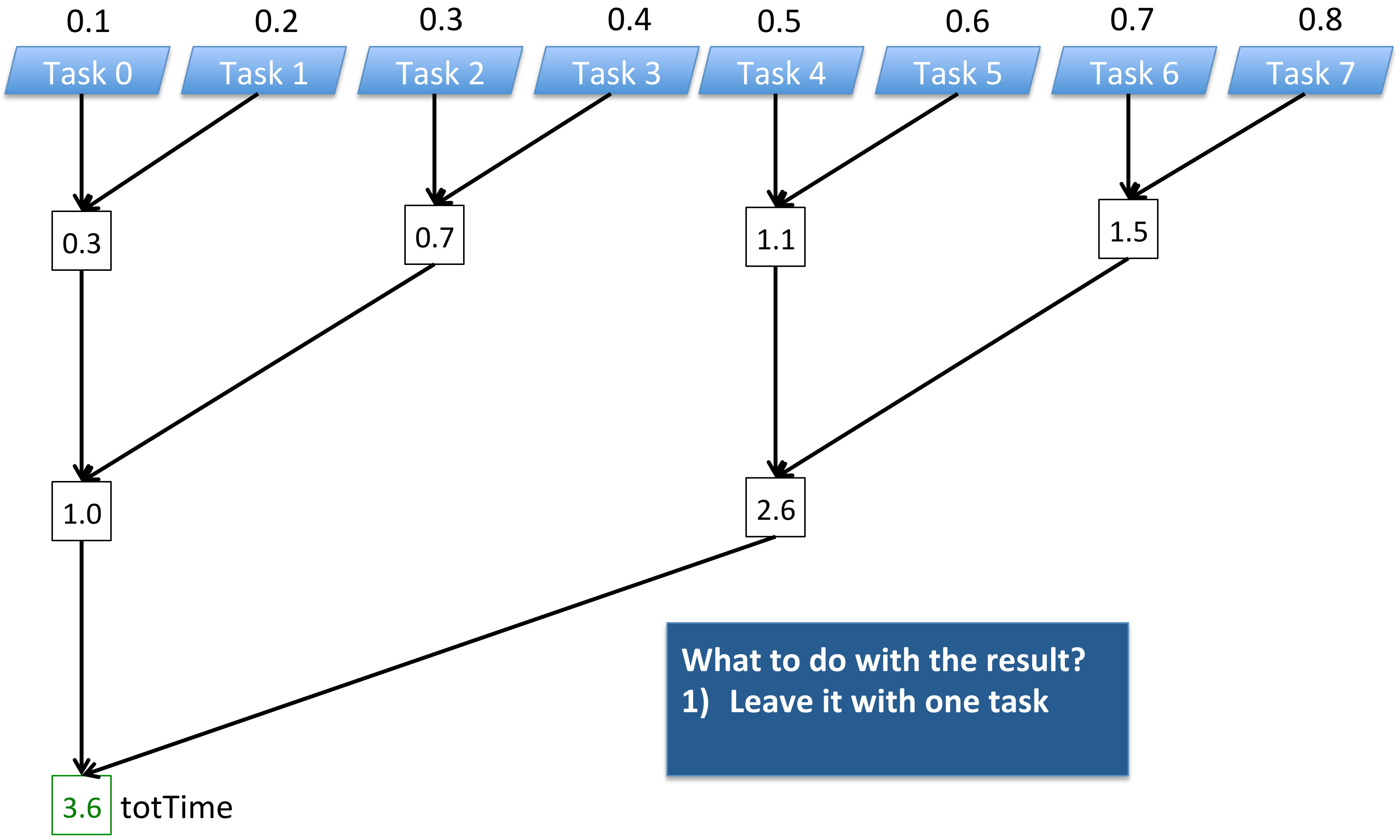
Fix: Use a *Reduction*



Depth: O($\log_2$#tasks)
Contention: O(1)

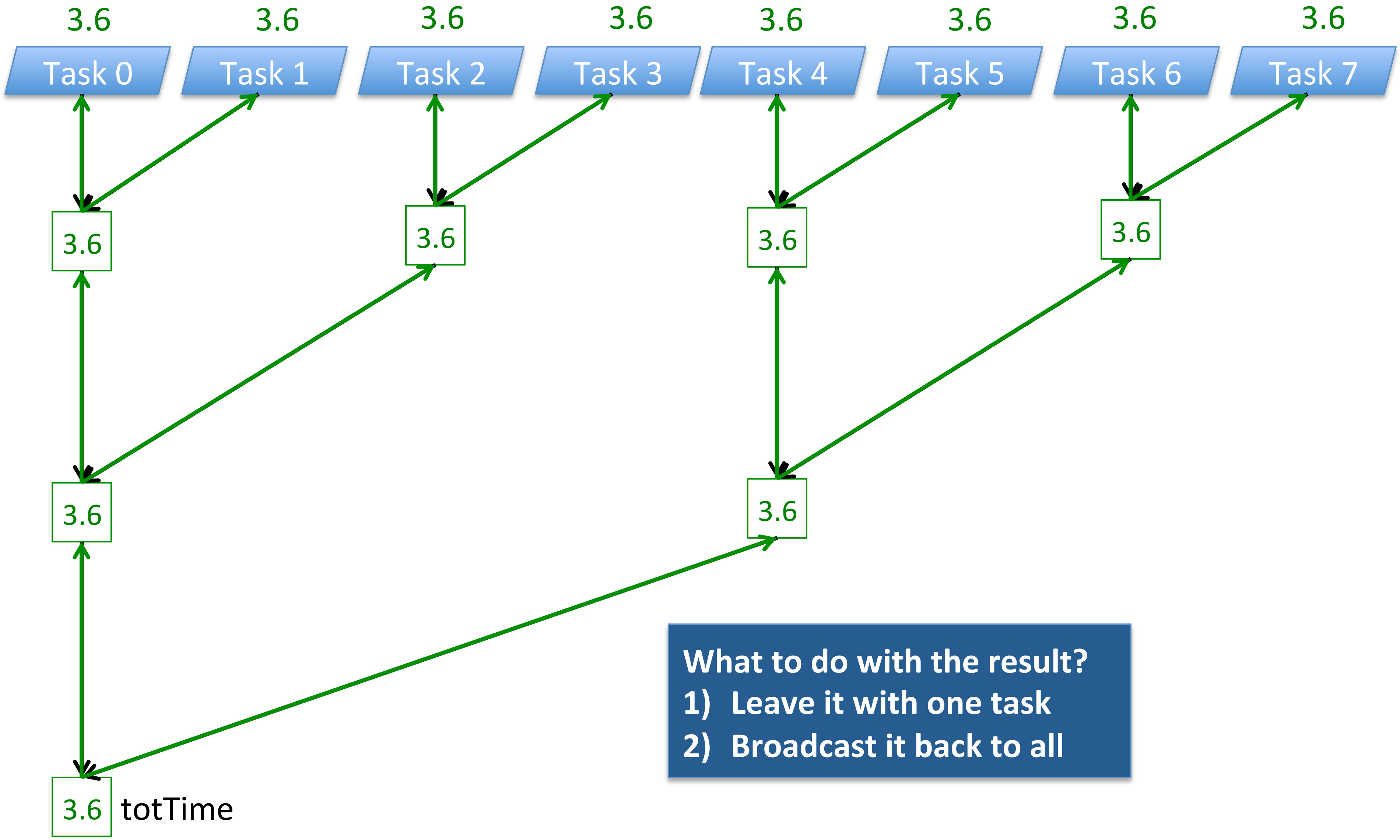
What if we used a tree with degree *d*?

Fix: Use a *Reduction*



What to do with the result?
1) **Leave it with one task**

Fix: Use a *Reduction*



What to do with the result?

1) **Leave it with one task**
2) **Broadcast it back to all**

Reduction Operations

- Typical operations: +, *, &, |, ^, min[loc], max[loc], …
  - *typically* operators that are commutative and associative
- Two main flavors:
  - collective *("members contribute")*

```
create tasks…
  const myContribution = doSomeWork(…);
  const total = sumReduceAll(myContribution);
join tasks…
```

  - global-view *("holistic")*

```
const total = + reduce A;   // sum A's elements
```

Reductions in Chapel (Global-View only today)

- Syntax

```
reduce-expr:
  reduce-op reduce iterator-expr
```

- Semantics
  - Combines argument values using *reduce-op*
  - *Reduce-op* may be built-in or user-defined
- Examples

```
total = + reduce A;
bigDiff = max reduce [i in Inner] abs(A[i]-B[i]);
(minVal, minLoc) = minloc reduce zip(A, D);
```

Fixing RRWW bugs with reductions

Collective Style

```
create tasks
  …
  myTotTime = sumReduce(myTime);
  …
join tasks
```

Interestingly, collective-style reductions are supported by neither Pthreads (to my knowledge) nor Chapel (planned as future work)

Global-View Style

```
var times: [0..#numTasks] real;

coforall tid in 0..#numTasks {
  …
  times[tid] = myTime;
  …
}

const totTime = + reduce times;
```

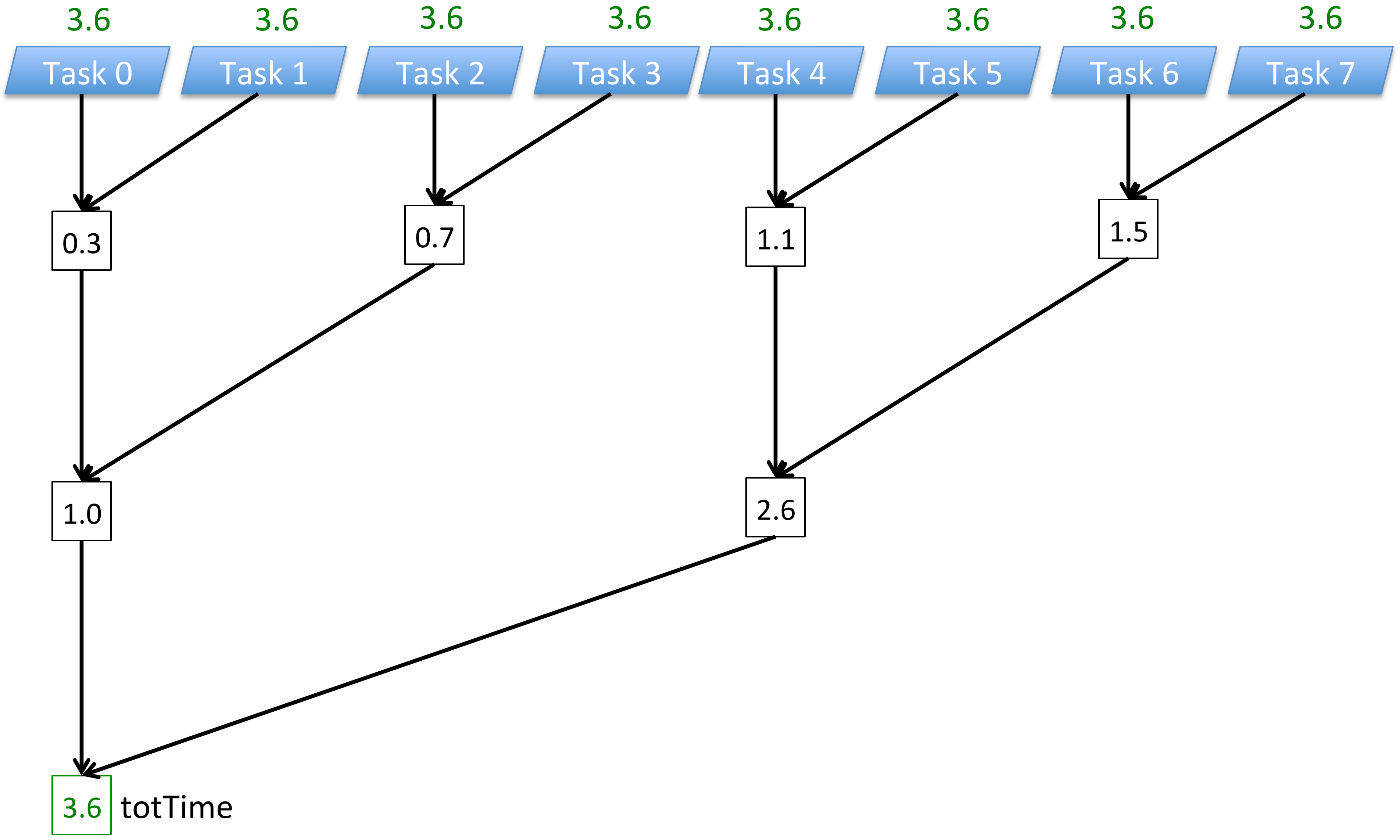
Defining Parallel Reductions

- What's required?
- In the simplest case (result type == state type == input type)
  - An identity element
    - What should we return if we end up reducing nothing?
  - A *combiner* function
    - Combines two input values to create a result value

Defining Parallel Reductions

- What's required?
- In the simplest case (result type == state type == input type)
    - example1: sum reduction
    - example2: min reduction
  - An identity element
    - 0
    - max(type)?
  - A *combiner* function
    - v1 + v2
    - max(v1, v2)

Fix: Use a *Reduction*



Defining Parallel Reductions

- What's required?
- More generally (result type != input type, or state is required)
  - An identity element
    - What should we initialize our state to?
  - An *accumulator* function
    - Combines an input value and a state value, creating a state value
  - A *combiner* function
    - Combines two state values, creates a state value
  - A *result* function
    - Transforms a state value into an answer

Defining Parallel Reductions

- What's required?
- More generally (result type != input type, or state is required)
  - example: "min-max reduction" (find range of values)
    - e.g., min-max reduce of [4, 2, 9, -3, 7, 8] would be -3..9
  - An identity element
    - (max, min)
  - An *accumulator* function
    - (min(newval, state.min), max(newval, state.max))
  - A *combiner* function
    - (min(state1.min, state2.min), max(state1.max, state2.max))
  - A *result* function
    - return state.min..state.max

That’s it for today!

- Questions?