TORSTEN HOEFLER

Parallel Programming

Wait-Free Consensus & Parallel Algorithms Primer

Microsoft announces new supercomputer, lays out vision for future AI work

Jennifer Langston
May 19, 2020

Microsoft has built one of the top five publicly disclosed supercomputers in the world, making new infrastructure available in Azure to train extremely large artificial intelligence models, the company is announcing at its Build developers conference.

Built in collaboration with and exclusively for OpenAI, the supercomputer hosted in Azure was designed specifically to train that company's AI models. It represents a key milestone in a partnership announced last year to jointly create new supercomputing technologies in Azure.

It's also a first step toward making the next generation of very large AI models and the infrastructure needed to train them available as a platform for other organizations and developers to build upon.

"The exciting thing about these models is the breadth of things they're going to enable," said Microsoft Chief Technical Officer Kevin Scott, who said the potential benefits extend far beyond narrow advances in one type of AI model.

Torsten Hoefler @thoefler · May 19
Sabbatical well spent ;).

> **Microsoft** @Microsoft · May 19
> Microsoft's new supercomputer will enable a previously unimaginable #AI software platform to accelerate developer projects, both large and small. 💪
>
> Learn more: msft.it/6006TiKLK #MSBuild

1 · 31

Torsten Hoefler Retweeted

ETH Zürich @ETH · Nov 22, 2019
Zwei Forschungsgruppen der ETH Zürich entwickelten eine Methode, die realitätsnah und effizient elektronische Nanobauteile und deren Eigenschaften simuliert. Gestern erhielten sie für ihre Leistung den Gordon Bell Prize für Supercomputing. ethz.ch/de/news-und-ve... @cscsch @CSatET

2 · 16

Learning goals for today

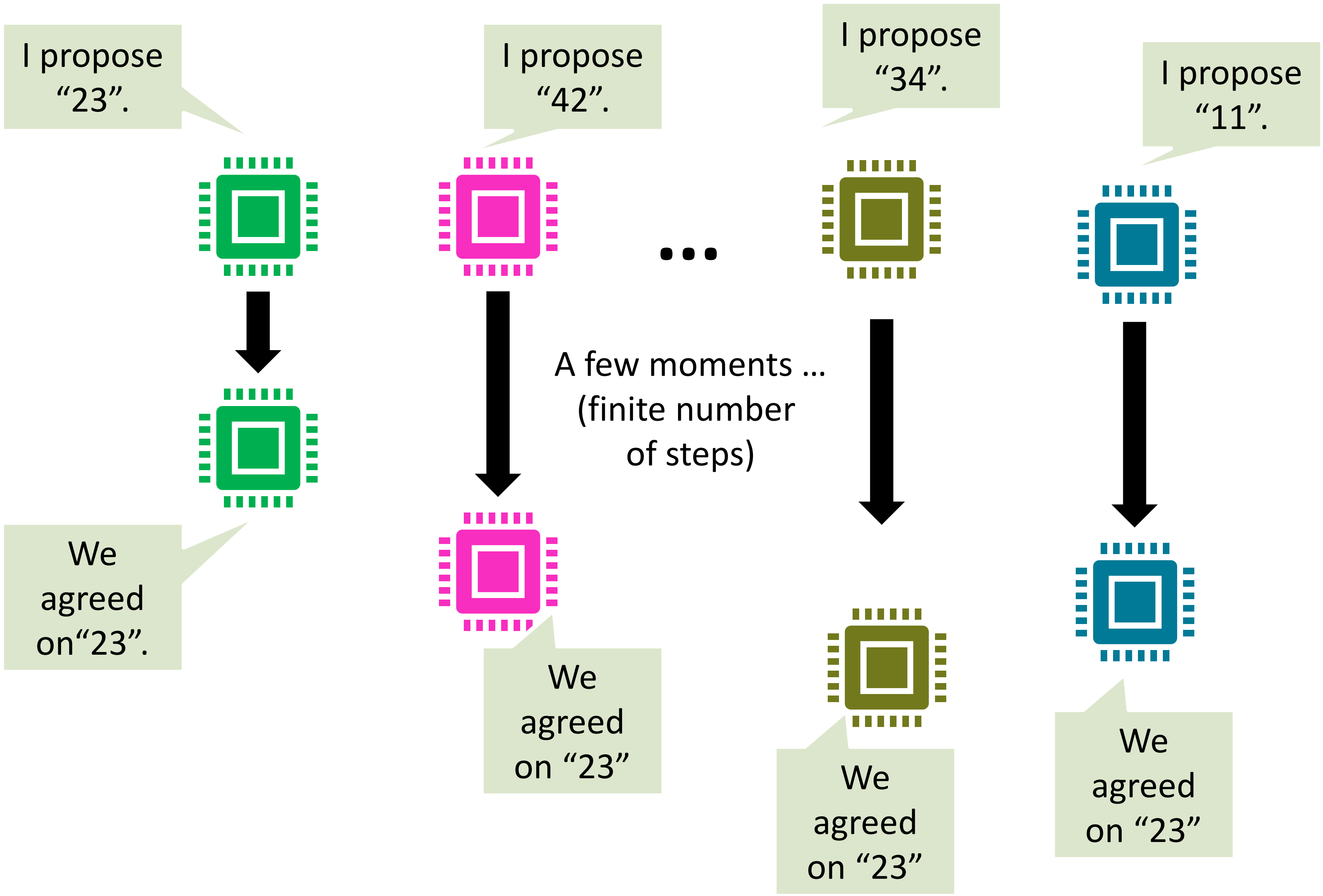
- **Understand one fundamental principle of parallel computing – with an impossibility proof!**
  - Herlihy, Shavit: *“The aforementioned corollary is perhaps one of the most striking impossibility results in Computer Science. It explains why, if we want to implement lockfree concurrent data structures on modern multiprocessors, our hardware must provide primitive synchronization operations other than loads and stores (reads– writes).”*

- **We will proof the impossibility of wait-free consensus with reader/writer registers**
  - Why wait-free – you should know ☺
  - What is the solution: atomic operations (we already covered it)

    They are expensive though! And which operations is still unclear

- **Recall the consensus hierarchy!**
  - Consensus number 1, 2, …, ∞

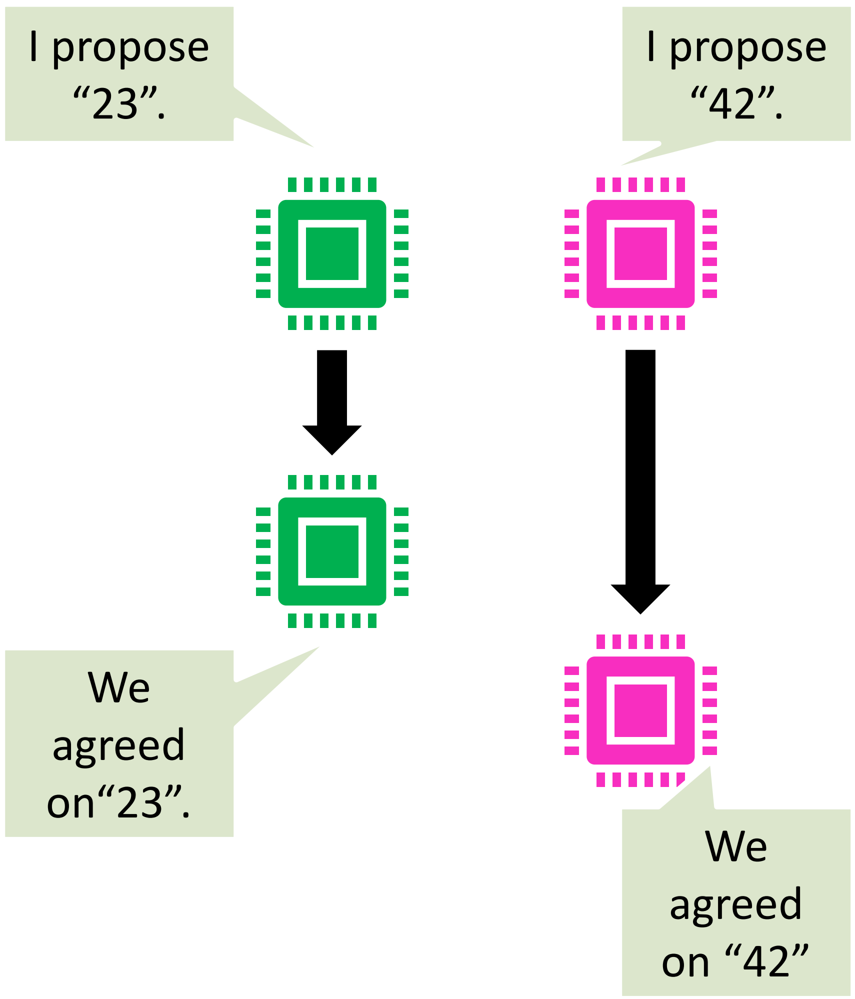
Recap: Wait-free Consensus Protocols



Simplification to two-thread consensus
(it doesn't get simpler than that ☺)

Which other scenarios are allowed?

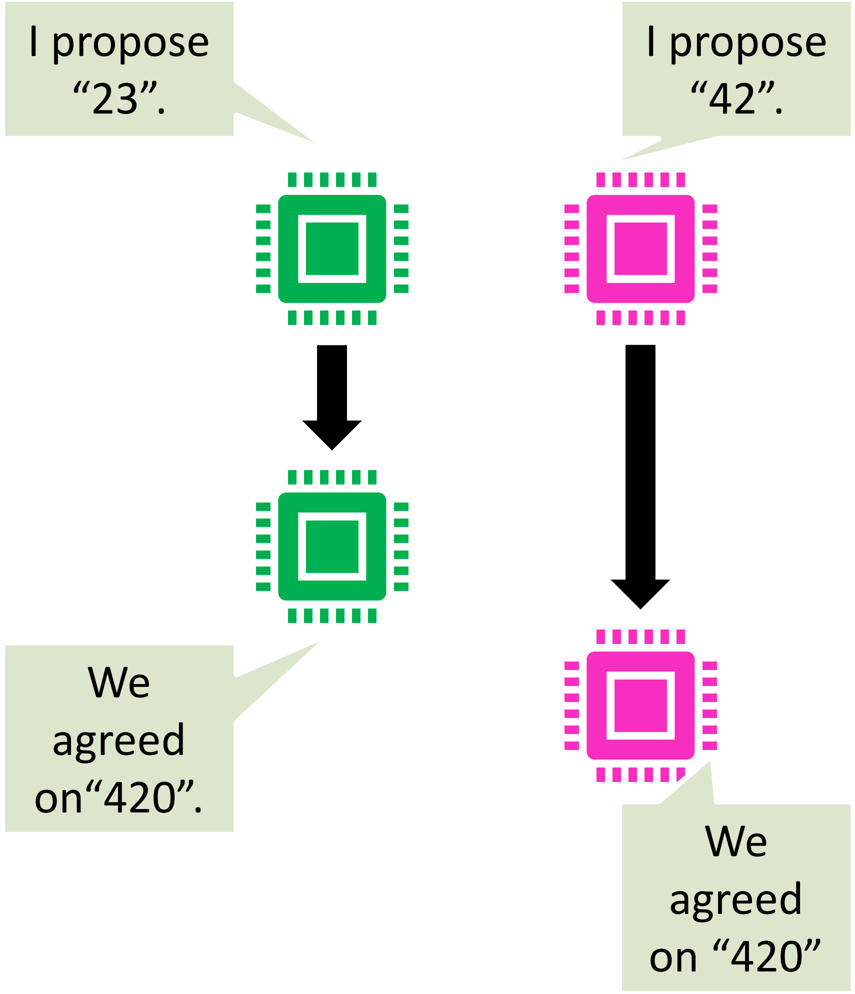
Consistent Result



This is illegal!

Consensus result needs to be consistent: the same on all threads.

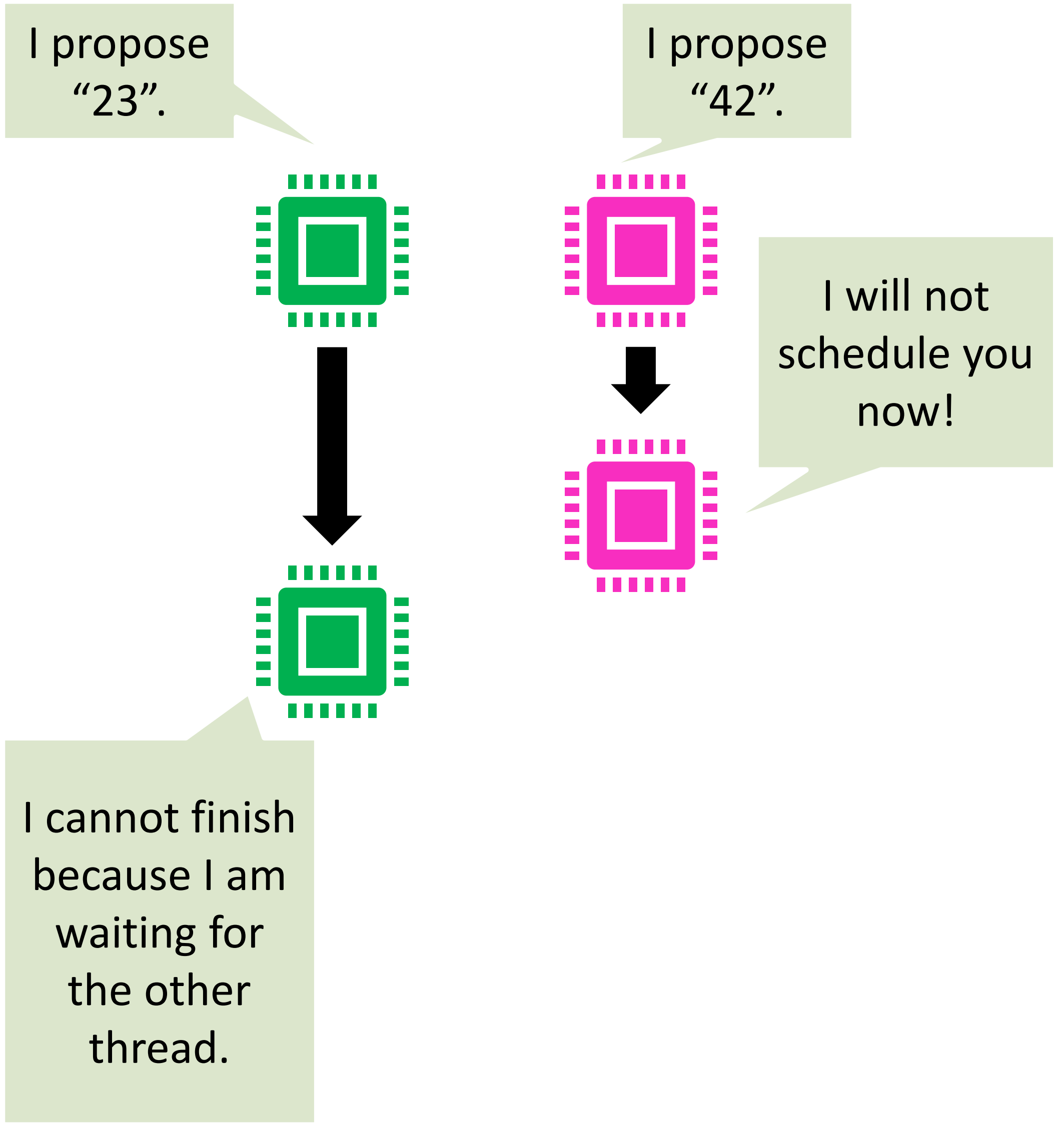
Valid Result



This is illegal!

Consensus result needs to be valid: proposed by some thread.

Wait-Free



This is illegal!

Consensus needs to be wait-free: All threads finish after a finite number of steps, independent of other threads.

Simplification: Binary Consensus

- **Instead of proposing an integer, every thread now proposes either 0 or 1**
- **Equivalent to “normal” consensus for two threads**
  - How can we proof this?
  - If we have int_decide(int) as primitive, we can implement bin_decide(bit)
  - and vice-versa

→

```
bin_decide(bit b) {
  return int_decide(b)
}
```

We can implement binary consensus using integer consensus.

←

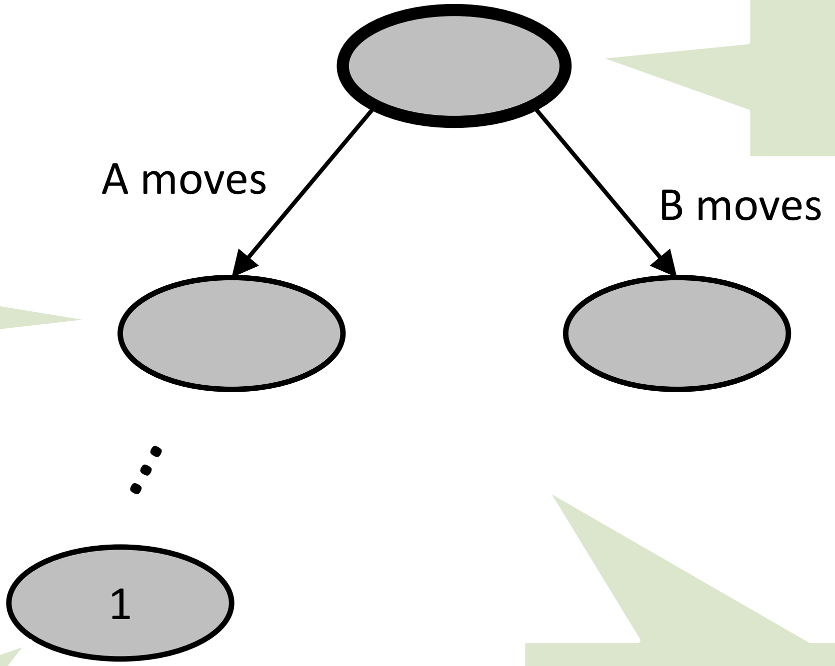
(two threads only)

```
int_decide(int d) {
  propose[id] = d; // shared array
  int winner = bin_decide(id);
  return propose[winner];
}
```

We can implement integer consensus using binary consensus (id in {0,1} and unique).

State Diagrams of Two-thread Consensus Protocols

Initial state, both threads (A and B) have not yet executed the first instruction of the consensus protocol.

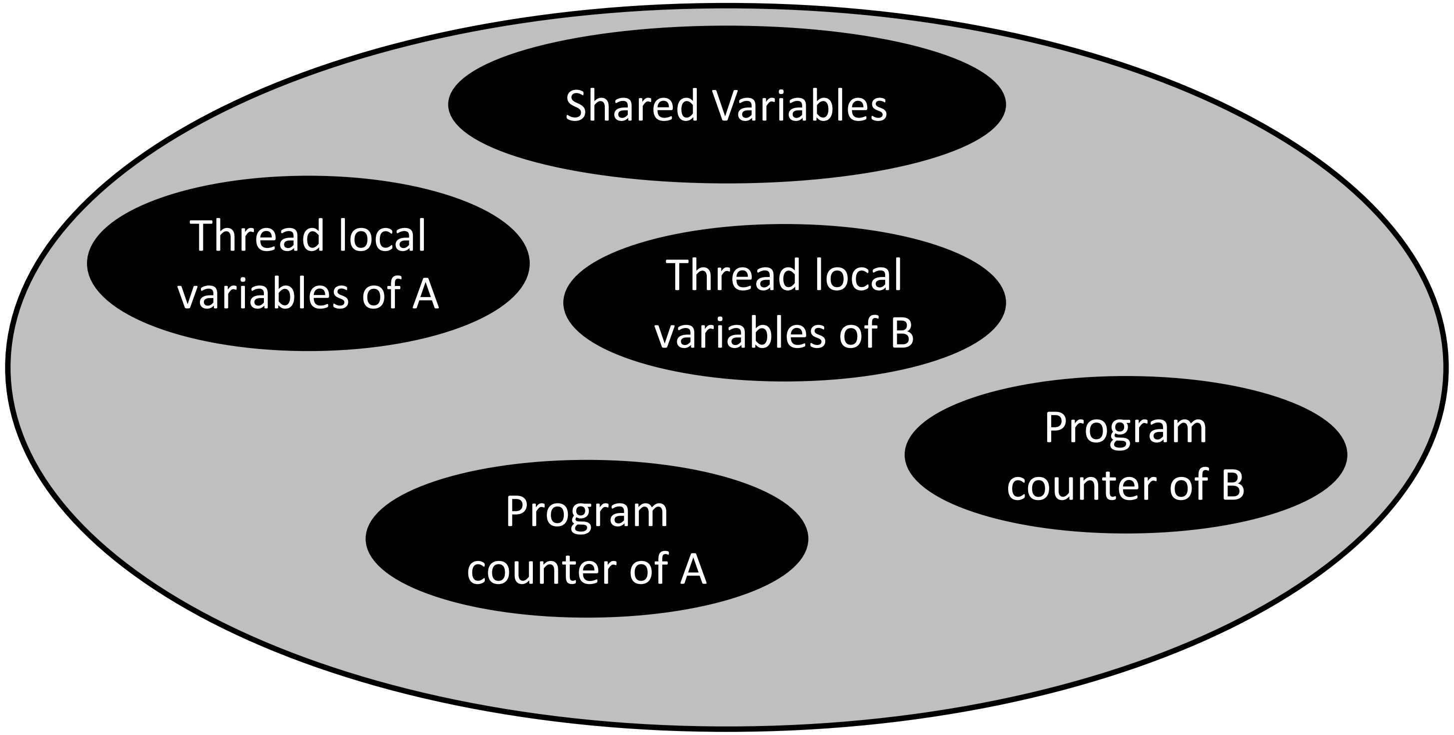


Each state has at most two **successors**: Either A or B execute an instruction.

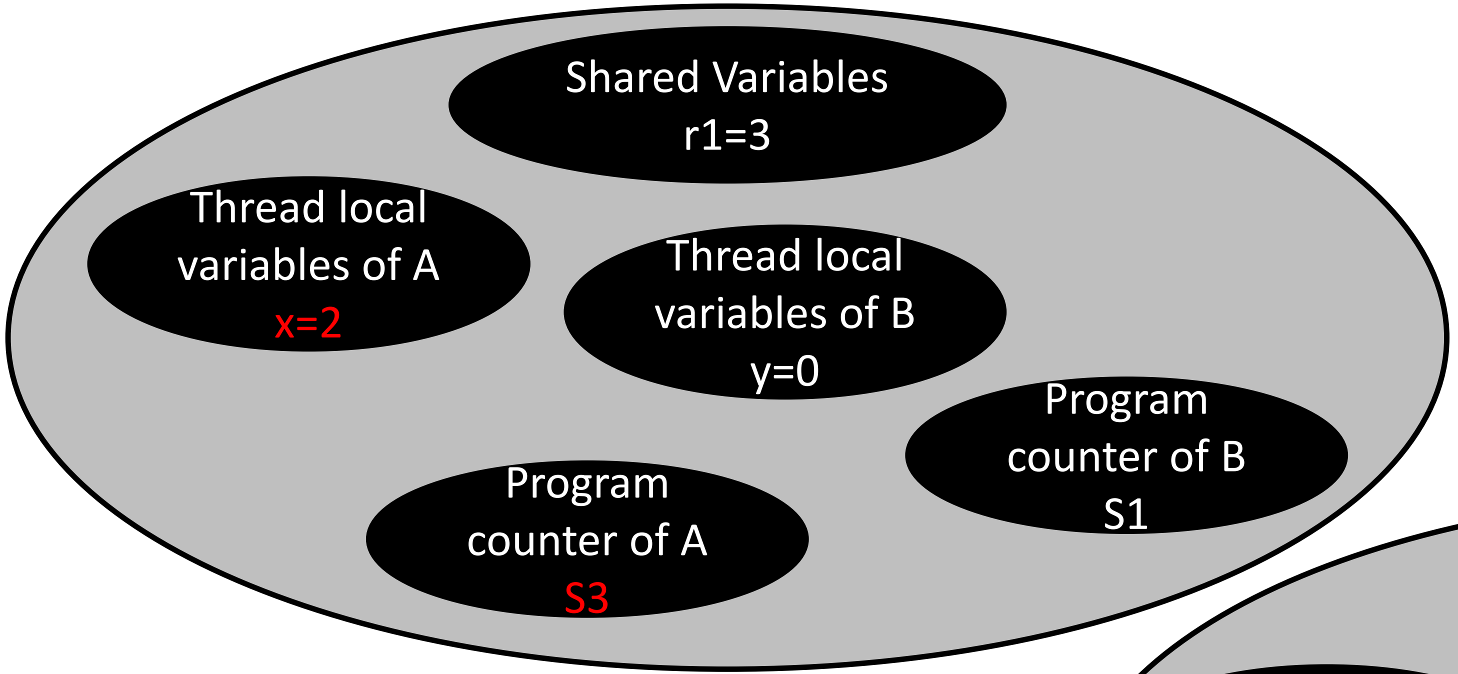
Final state (decision value of any final state reached has to be the same on both threads!)

This tree **must** be finite (because the protocol is wait-free)

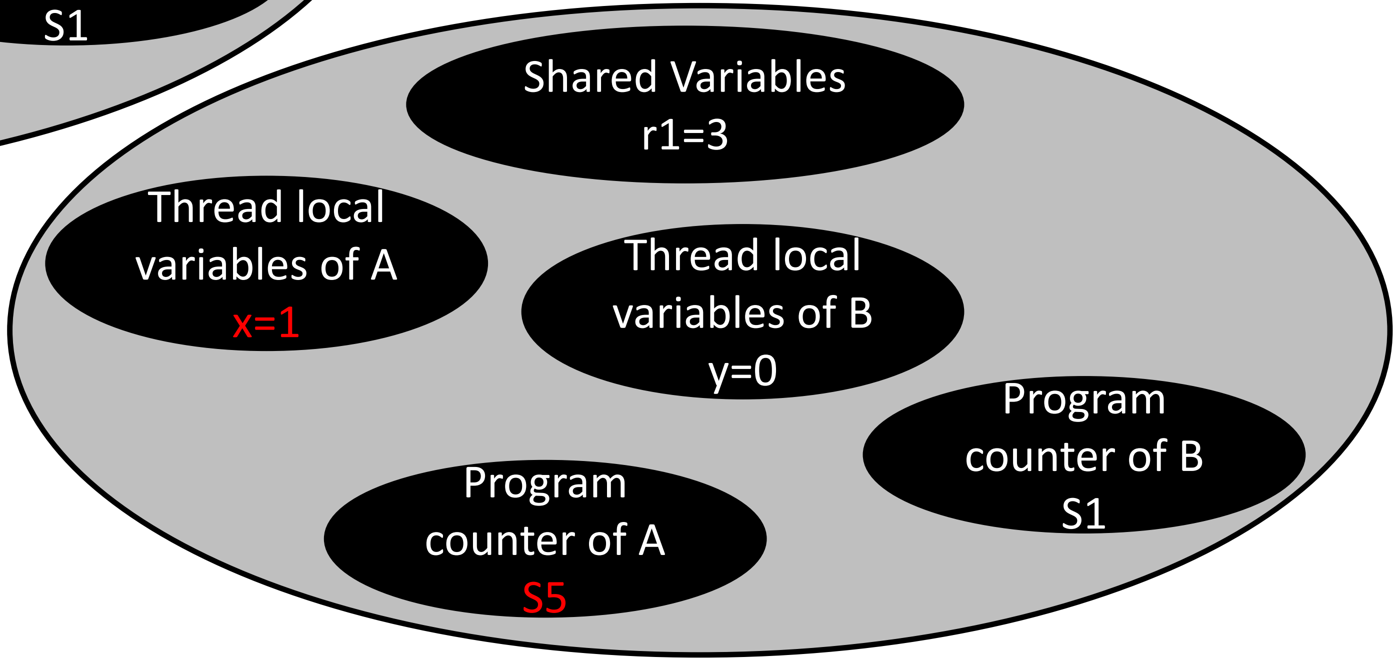
Anatomy of a State (in Two-Thread Consensus)



Anatomy of a State - Example



The states are different, since A has different local variables and program counter values.



Yet from B's perspective they look the same! (Until A writes x into a shared variable!)

The Concept of Valency

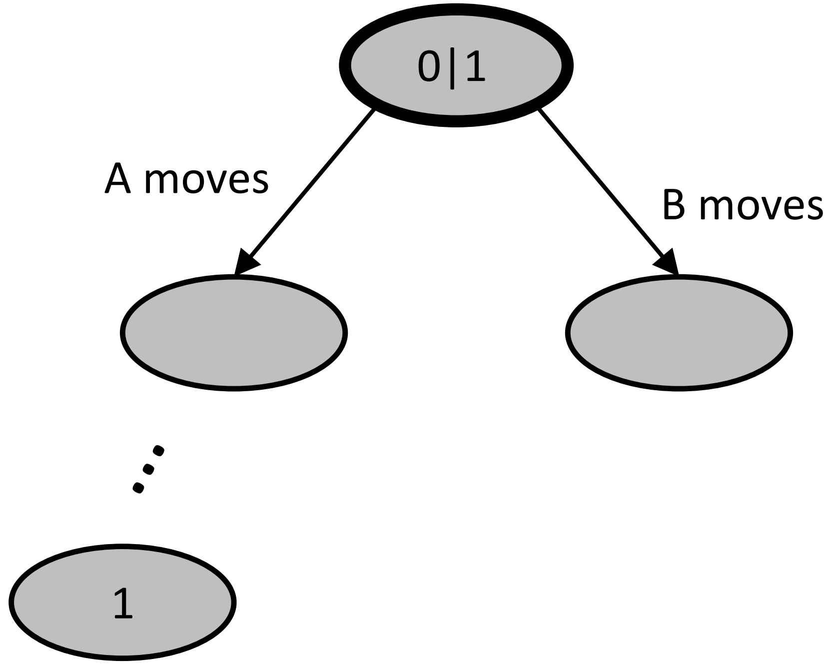
- **In binary two-thread consensus, threads either decide zero (0) or one (1)**
- **At some point during the execution (i.e., a state), each thread will "decide" what to return**
  - We call a state where a thread has decided on one 1-valent and a state where a thread has decided on zero 0-valent
  - Undecided states are called bivalent – decided states are called univalent

- **Lemma 1: The initial state is bivalent**
  - Proof outline:

    Consider initial state with A has input 0 and B has input 1

    If A finished before B starts, we must decide 0 and if B finishes before A starts, we must decide 1 (because is only knows the thread's input!)

    Thus, the initial state must be bivalent!



Critical States in Binary Two-Thread Consensus

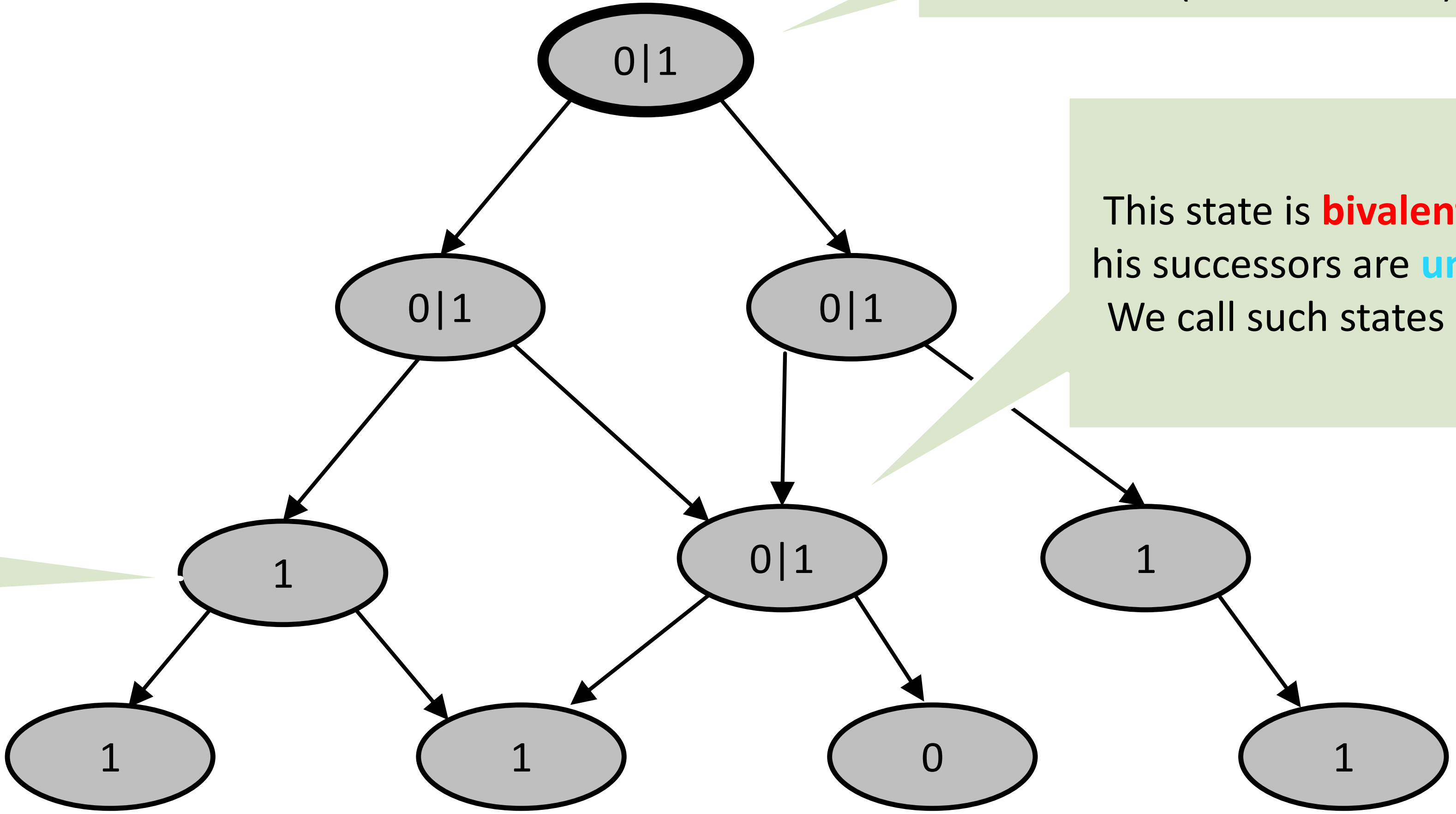
Definition: a (bivalent) state is called **critical**, if both child states are univalent!

There is always at least one **bivalent** state (the start state).

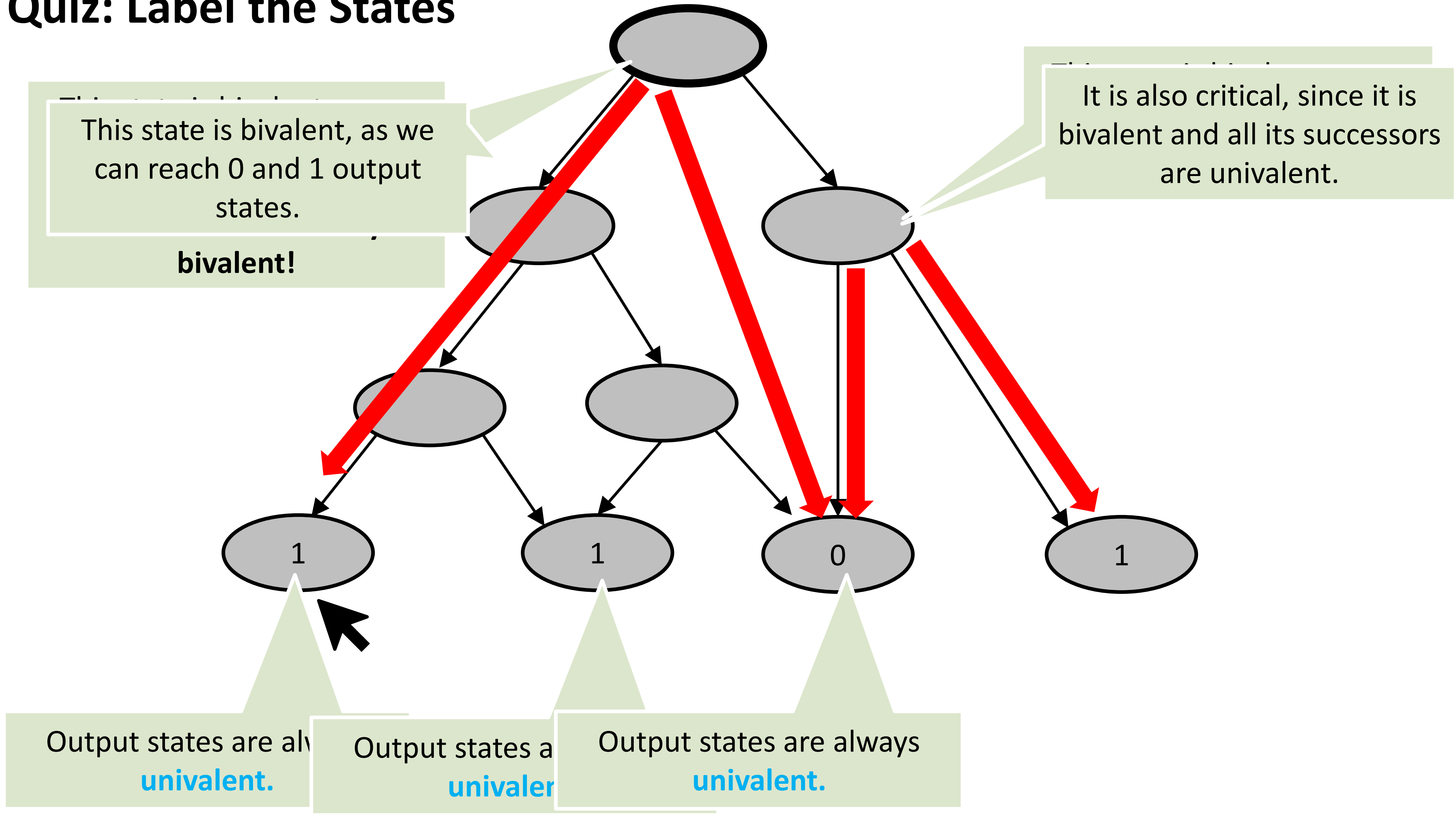
This state is **bivalent** but all his successors are **univalent**. We call such states **critical**.

From this state we only reach states with output 1, so it is also **univalent.**

Output states are always **univalent.**



Quiz: Label the States



Critical State Existence Proof

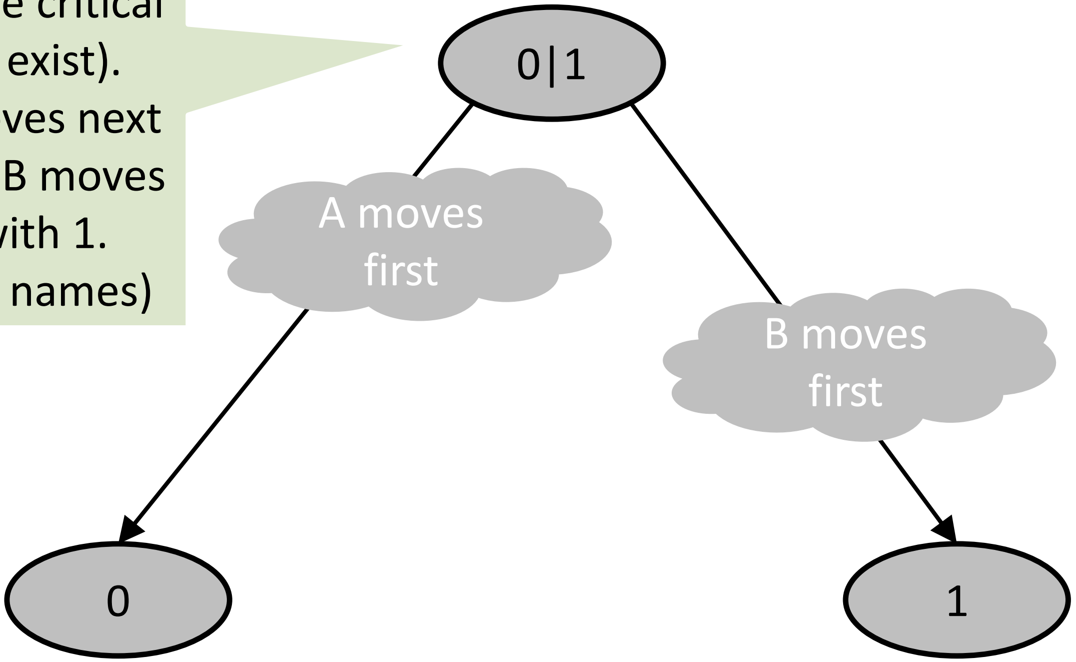
Lemma 2: Every consensus protocol has a critical state.

Proof: From (bivalent) start state, let the threads only move to other bivalent states.

- **If it runs forever the protocol is not wait free.**
- **If it reaches a position where no moves are possible this state is critical.**

Impossibility Proof Setup – Critical State

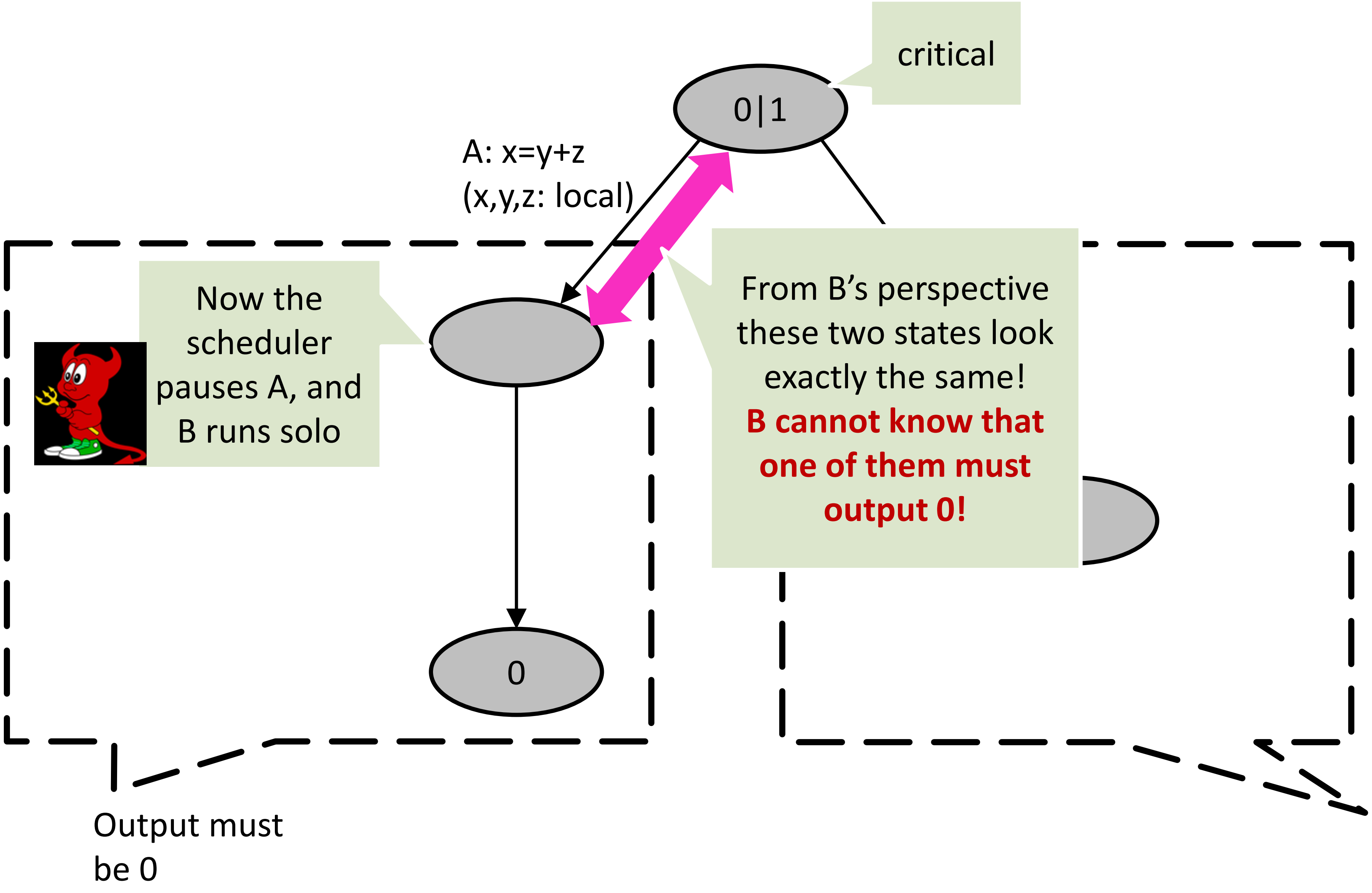
Assume we are in the critical state (which must exist). Assume that if A moves next we end up with 0, if B moves next we end up with 1. (w.l.o.g., can switch names)



So what actions can a thread perform in its "move"?

Either read or write a shared register! – Let's see why.

Impossibility Proof Setup – Possible actions of a thread



So what actions can a thread perform in his "move"?

What happens if A just reads from and writes to local vars?

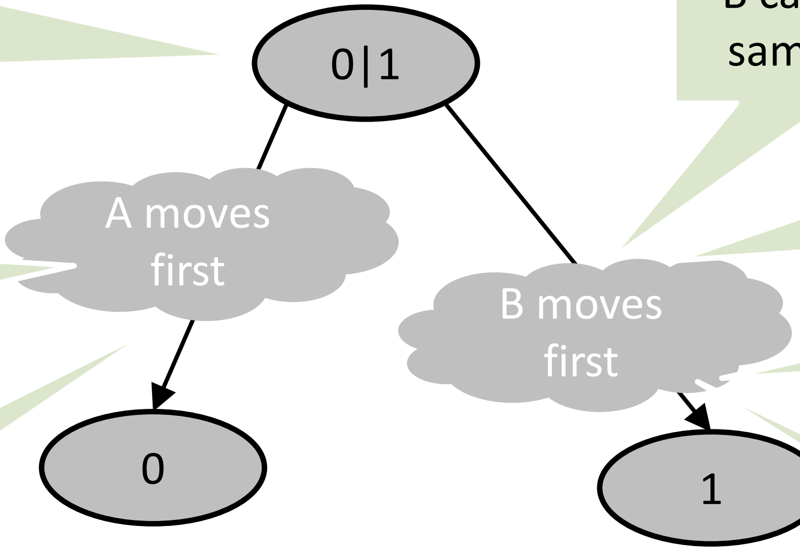
Conclusion: First instruction after critical state must be a read or write of a shared variable!

Impossibility Proof Setup – Possible actions of a thread

We know reading/writing local variables cannot lead out of a critical state – what remains?

A can read a shared variable

A can write a shared variable



B can read the same variable

B can read a different variable

B can write the same variable

B can write a different variable

Many cases… let's make tables

Many Cases to check

		First Action		n	
		A: r1.read()	A: r1.write()	A: r1.write()	A: r2.write()
Second Action	B: r1.read()		?		
	B: r2.read()				
	B: r1.write()				
	B: r2.write()				

Is binary consensus possible for any of those?

Can we simplify somehow?

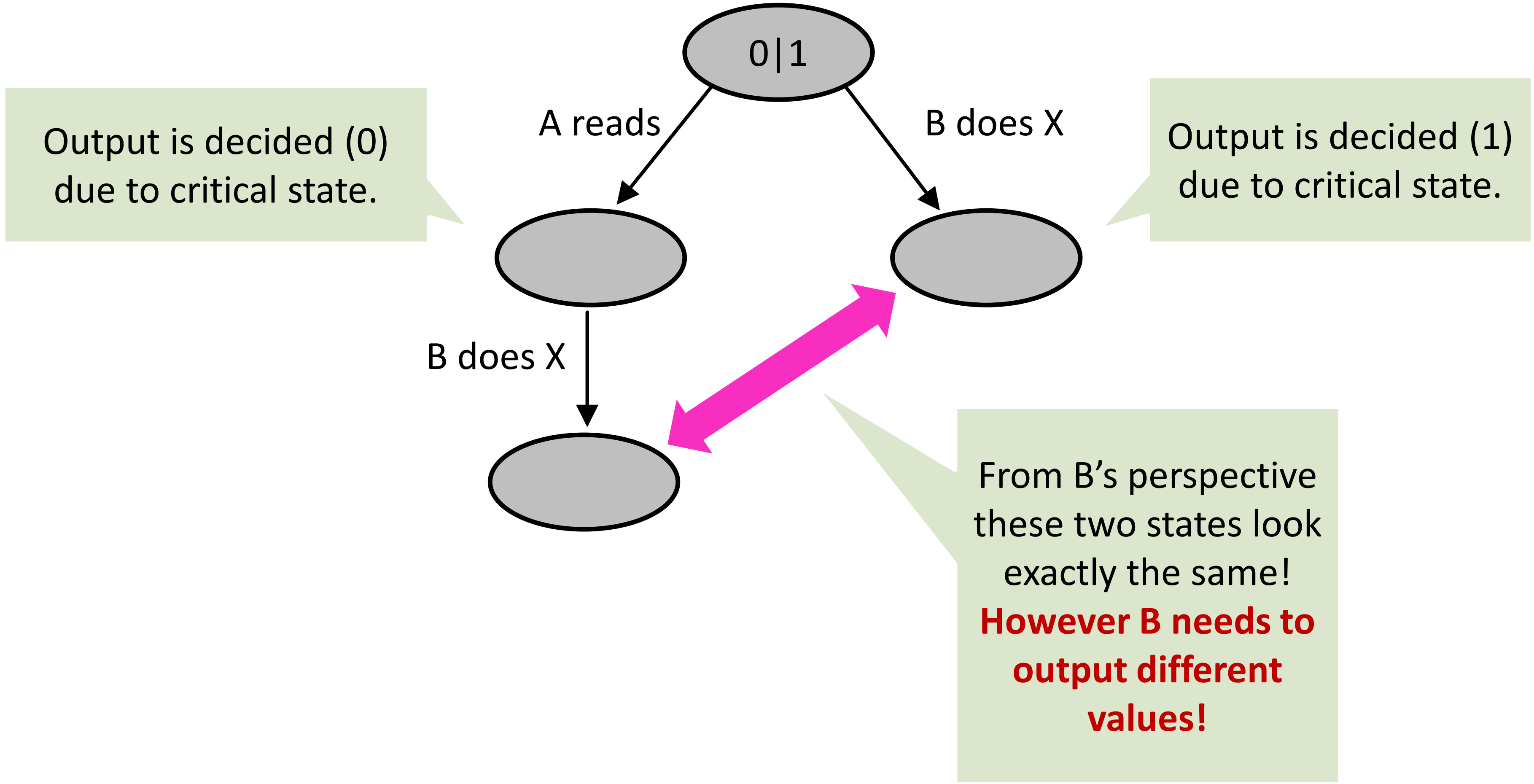
		Second Action			
		A: r1.read()	A: r2.read()	A: r1.write()	A: r2.write()
First Action	B: r1.read()		?		
	B: r2.read()				
	B: r1.write()				
	B: r2.write()				

Managable… Let's look at the cases where A reads

Let's say A always moves first, otherwise, switch names.

Similarly, we can call the register A reads r1 in both cases.

Impossibility Proof Case I: A reads

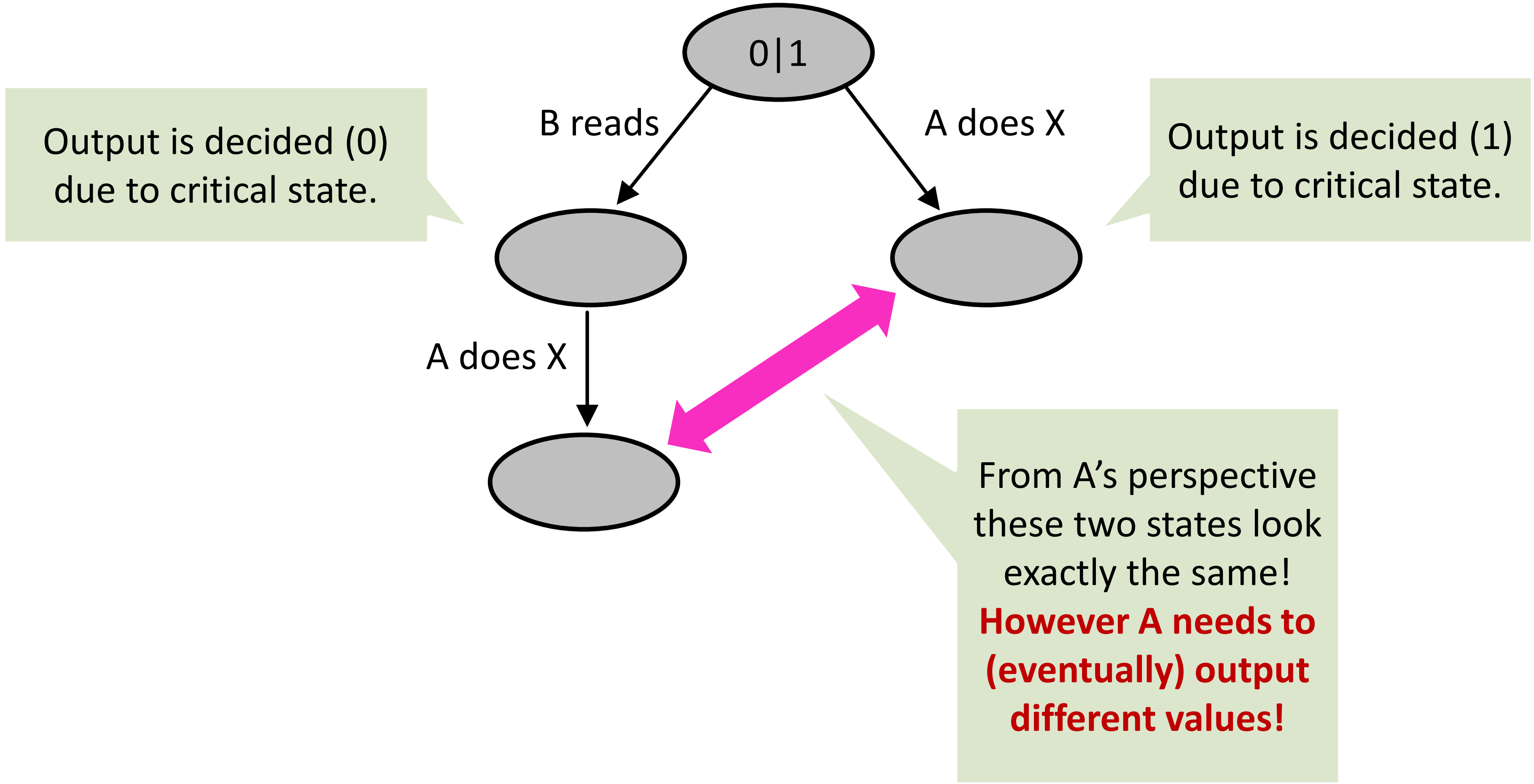


What did we just prove?

		First Action	
		A: r1.read()	A: r1.write()
Second Action	B: r1.read()	No, Case I	?
	B: r2.read()	No, Case I	
	B: r1.write()	No, Case I	
	B: r2.write()	No, Case I	

Is binary consensus possible for any of those?

Impossibility Proof Case I’: B reads

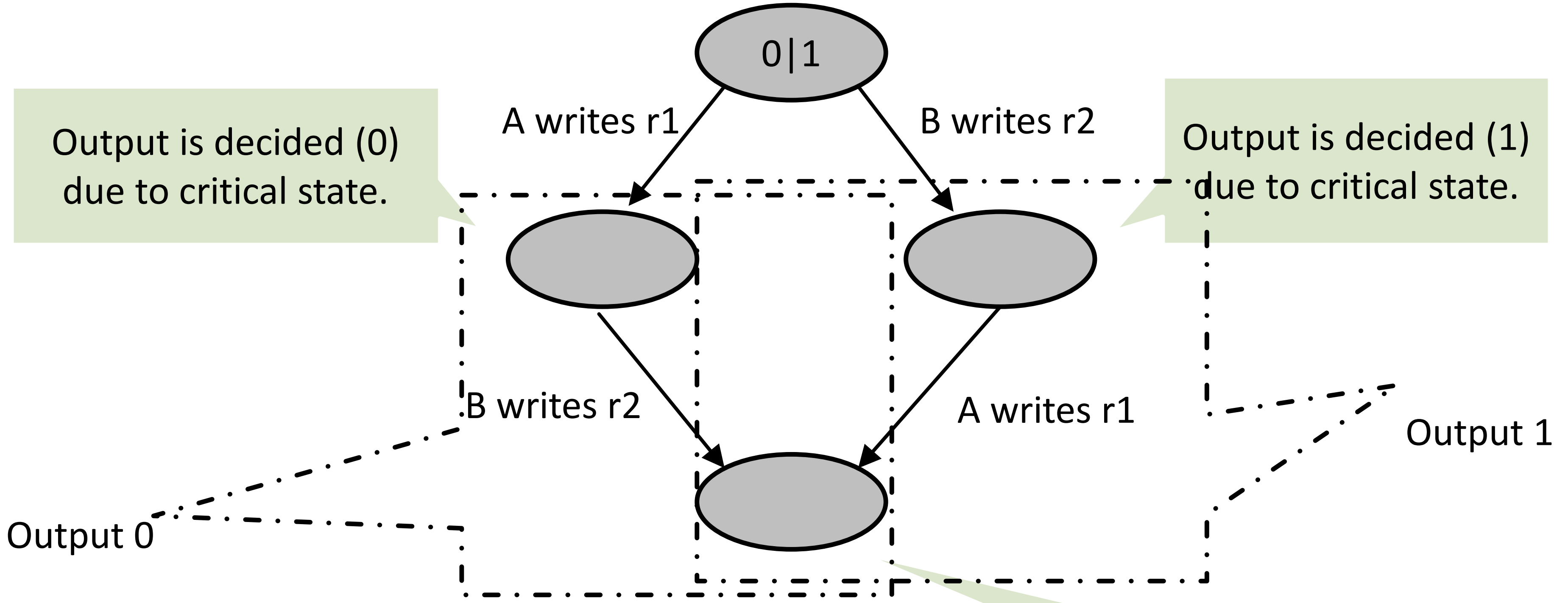


What did we just prove?

		First Action	
		A: r1.read()	A: r1.write()
Second Action	B: r1.read()	No, Case I	No, Case I'
	B: r2.read()	No, Case I	No, Case I'
	B: r1.write()	No, Case I	?
	B: r2.write()	No, Case I	

Is binary consensus possible for any of those?

Impossibility Proof Case II: A and B write to different registers



Output is decided (0) due to critical state.

Output is decided (1) due to critical state.

Exactly the same state!

However it should be outputting 0 / 1 depending on where it was reached from!

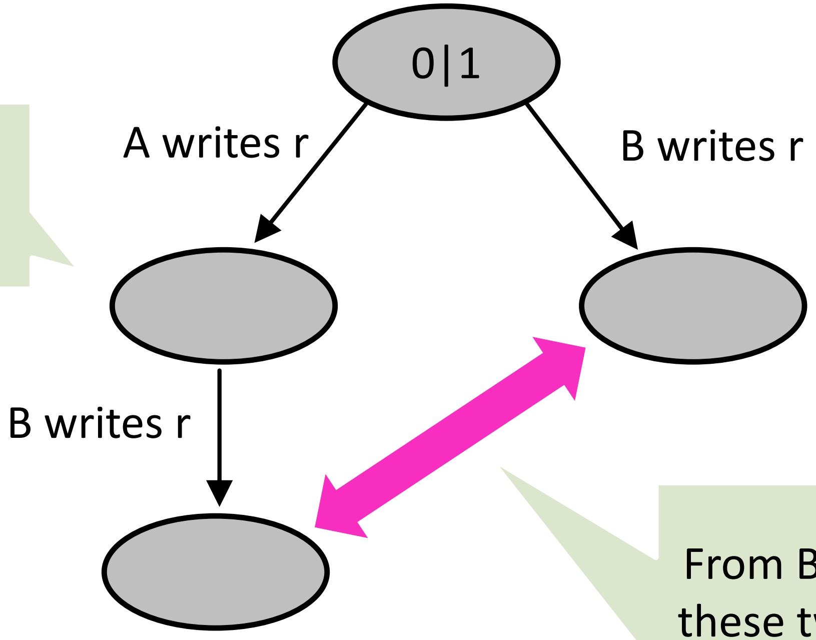
What did we just prove?

		First Action	
		A: r1.read()	A: r1.write()
Second Action	B: r1.read()	No, Case I	No, Case I’
	B: r2.read()	No, Case I	No, Case I’
	B: r1.write()	No, Case I	?
	B: r2.write()	No, Case I	No, Case II

Is binary consensus possible for any of those?

Impossibility Proof Case III: A and B write to the same register

Output is decided (0) due to critical state.



Output is decided (1) due to critical state.

From B's perspective these two states look exactly the same!
However B needs to output different values!

That’s all

		First Action	
		A: r1.read()	A: r1.write()
Second Action	B: r1.read()	No, Case I	No, Case I’
	B: r2.read()	No, Case I	No, Case I’
	B: r1.write()	No, Case I	No, Case III
	B: r2.write()	No, Case I	No, Case II

Is binary consensus possible for any of those?

No

1985, 2.5k citations

Impossibility of Distributed Consensus with One Faulty Process

MICHAEL J. FISCHER

Yale University, New Haven, Connecticut

NANCY A. LYNCH

Massachusetts Institute of Technology, Cambridge, Massachusetts

AND

MICHAEL S. PATERSON

University of Warwick, Coventry, England

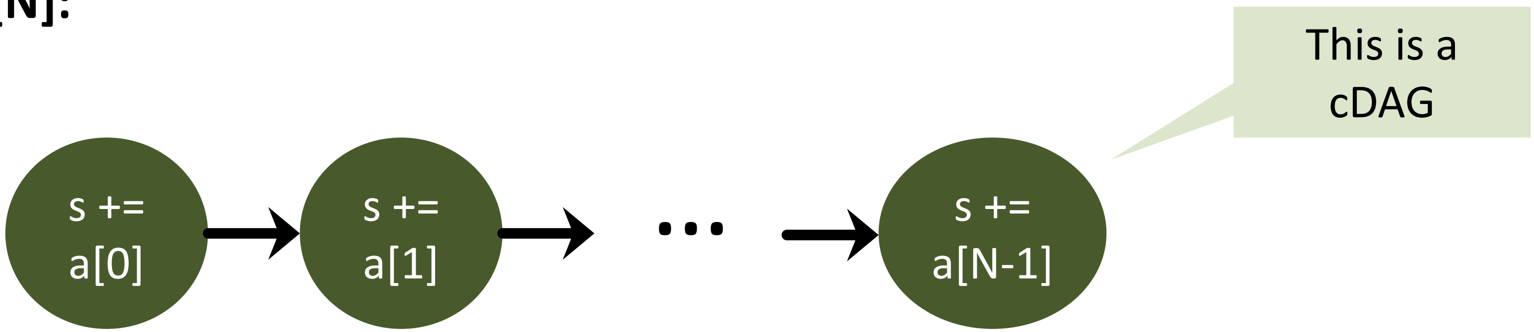
Abstract. The consensus problem involves an asynchronous system of processes, some of which may be

Primer for Parallel Algorithms

- **This lecture is called "parallel programming" – unfortunately, there is no "parallel algorithms" lecture in our curriculum. Sequential algorithms are different and programming without algorithms questionable.**
- **You already heard about work and depth in the first part – I will show you some (simple) and practical algorithms as examples today!**
- **Recall:**
  - Work W – number of operations performed when executing the algorithm (= sequential running time for P=1)
  - Depth D – minimal number of operations for any parallel execution (= parallel running time for P=∞)

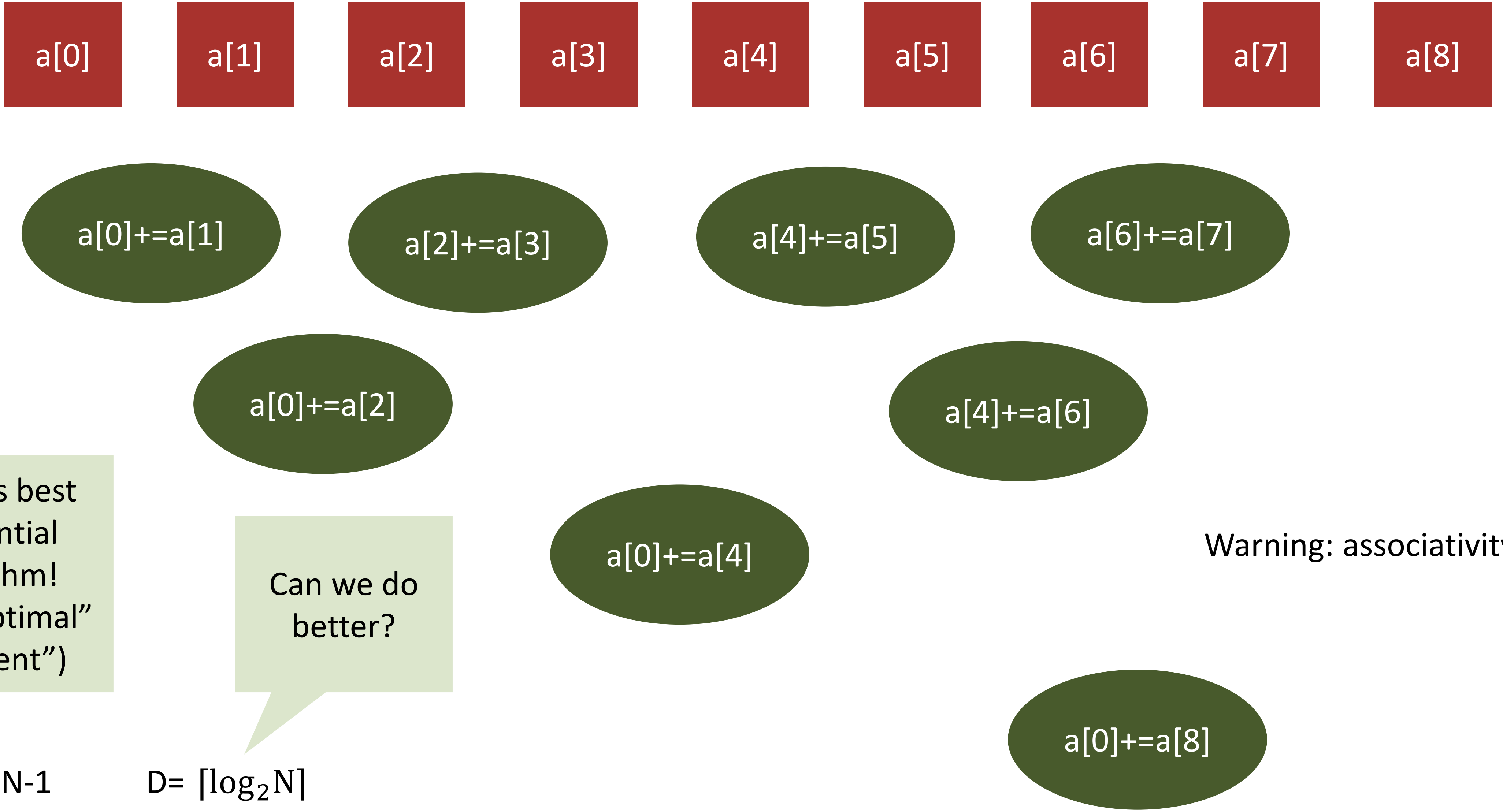
    Depth is also the longest path in the computational DAG (cDAG)
- **Example: summation of array a[N]:**

```
for(int i=1; i<N; ++i) {
  a[0] += a[i];
}
```



W= N-1 D= N-1 Is this a good parallel algorithm?

Parallel Summation ("Reduction")



Same as best sequential algorithm! "work optimal" ("efficient")

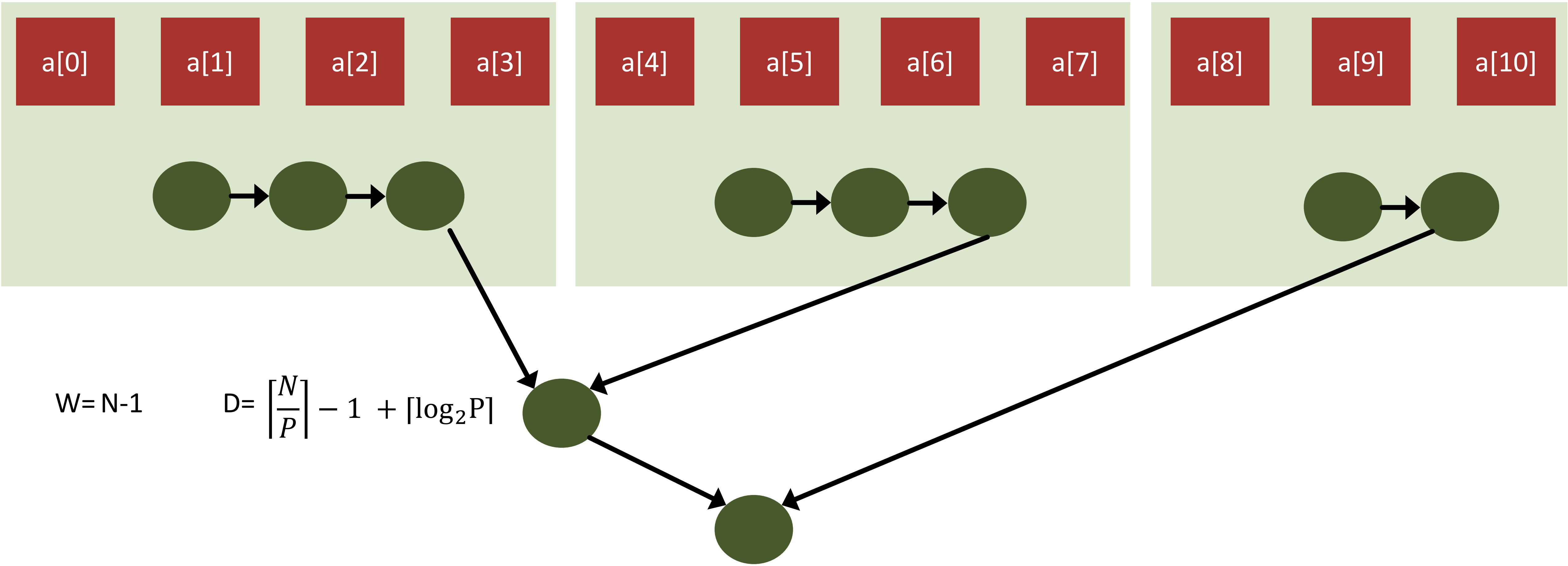
Can we do better?

Warning: associativity required!

W= N-1

D= $\lceil \log_2 N \rceil$

What if N ≫ P (usually the case!)



W= N-1

D= $\left\lceil\frac{N}{P}\right\rceil - 1 + \lceil\log_2 \mathrm{P}\rceil$

Write the code for this (in the exercise) for arbitrary N and P!

Now to something real – Parallel Matrix Multiplication (e.g., Neural Networks)

```
double A[N][K], B[K][M], C[N][M];

for (int i =0; i < N; ++ i)
  for (int j =0; j < M; ++ j) {
    C[i][j] = 0;
    for (int l =0; l < K; ++ l)
      C[i][j] += A[i][l] * B[l][j];
}
```

W= NMK

D= NMK

simple parallel

```
double A[N][K], B[K][M], C[N][M];

parallel for (int i =0; i < N; ++ i)
  parallel for (int j =0; j < M; ++ j) {
    C[i][j] = 0;
    for (int l =0; l < K; ++ l)
      C[i][j] += A[i][l] * B[l][j];
}
```

W= NMK

D= K

Can we do better?
(What if P >> NM?)

```
double A[N][K], B[K][M], C[N][M];
double T[N][M][K]

parallel for (int i =0; i < N; ++ i)
  parallel for (int j =0; j < M; ++ j) {
    parallel for (int l =0; l < K; ++ l)
      T[i][j][k] = A[i][k] * B[k][j];
      C[i][j] = reduce(T[i][j][k])
}
```

W= NMK

D= $\log_2 K$

What is the problem?

```
double A[N][K], B[K][M], C[N][M];
double T[N][M][P]

parallel for (int i =0; i < N; ++ i)
  parallel for (int j =0; j < M; ++ j) {
    parallel for (int r = [0.. P-1]) {
      T[i][j][r] = 0;
      for (int k = r*K/P; k < (r +1) * K / P; k++)
        T[i][j][r] = T[i][j][r] + A[i][k]*B[k][j];
      C[i][j] = reduce(T[i][j][r]) ;
} }
```