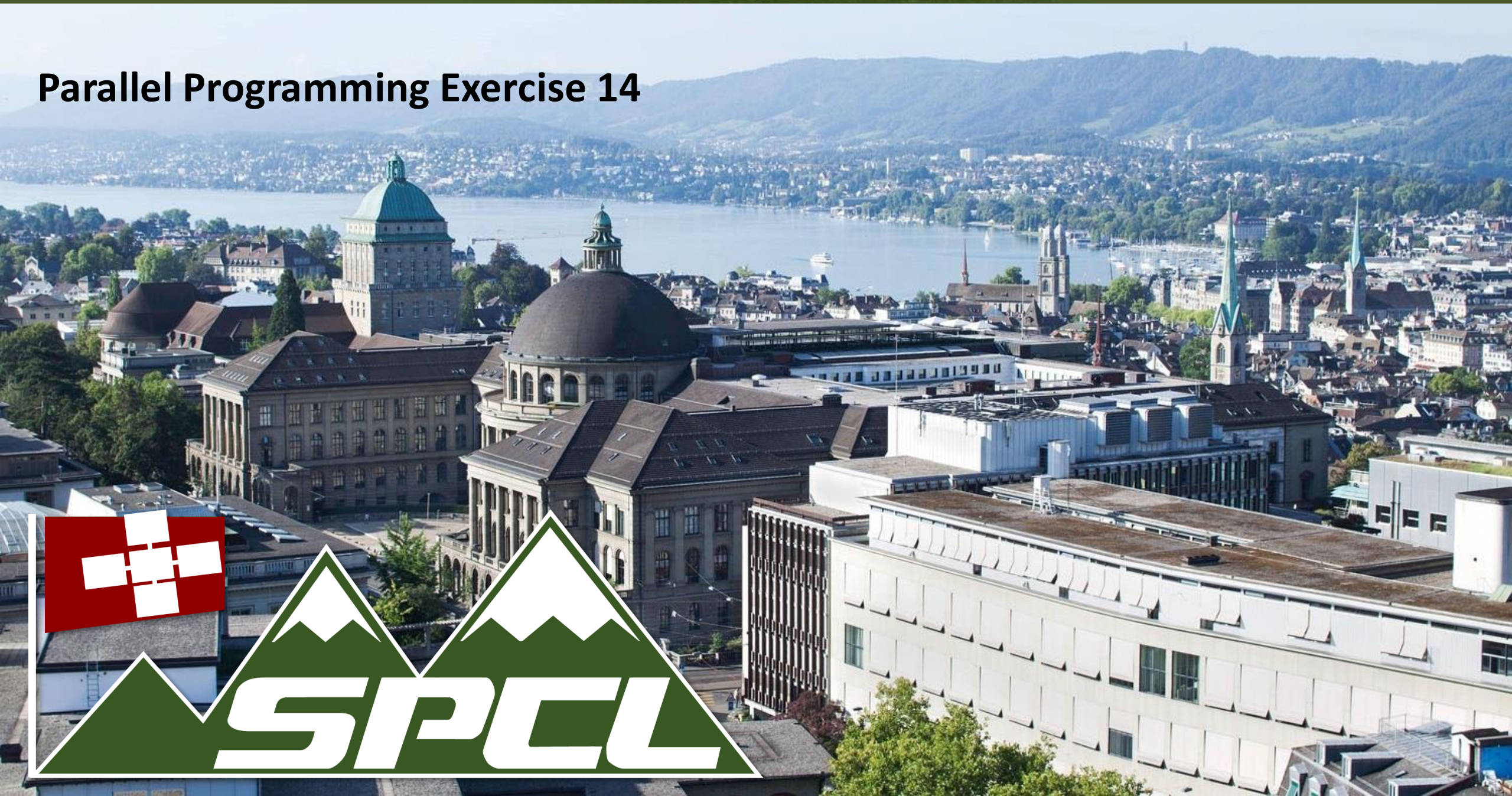
Parallel Programming Exercise 14



Outline

- Post-Discussion: Assignment 13
- Pre-Discussion: Assignment 14
- Theory
- Exam Tasks & Tips

Post-Discussion Assignment 13

Sequential Consistency

For each of the following histories, indicate if they are sequentially consistent or not

registers: r and s
queue: q

```
A:   --|r.write(1)|-------------------------
B:   ------|r.read():0|---------------------
C:   --------------------|r.read():1|------
```

```
A: q.enq(5)
B: q.enq(3)
A: void
B: void
A: q.deq()
B: q.deq()
A: 3
B: 3
```

```
A:   --|s.write(1)|-------------------------
B:   ------|r.read():0|---------------------
C:   --------------------|r.read():1|------
```

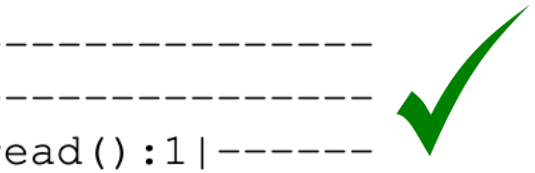
```
A:   --|s.write(1)|-------------------------
B:   ------|r.read():1|---|r.read():0|------
```

Sequential Consistency

For each of the following histories, indicate if they are sequentially consistent or not

registers: r and s
queue: q

```
A:   --|r.write(1)|-------------------------
B:   ------|r.read():0|---------------------
C:   --------------------|r.read():1|------
```



```
A: q.enq(5)
B: q.enq(3)
A: void
B: void
A: q.deq()
B: q.deq()
A: 3
B: 3
```

```
A:   --|s.write(1)|-------------------------
B:   ------|r.read():0|---------------------
C:   --------------------|r.read():1|------
```

```
A:   --|s.write(1)|-------------------------
B:   ------|r.read():1|---|r.read():0|------
```

Linearizability

Infer the object type from the supported operations,
registers are initially zero, stacks/queues initially empty.

```
A: s.push(1)
A: void
B: s.push(2)
B: void
B: s.pop()
A: s.pop()
B: 1
A: 2
```

```
A:    --|s.write(1)|-------------------------
B:    ------|r.read():1|---|r.read():0|------
```

Linearizability

Infer the object type from the supported operations,
registers are initially zero, stacks/queues initially empty.

```
A: s.push(1)
A: void
B: s.push(2)
B: void
B: s.pop()
A: s.pop()
B: 1
A: 2
```

```
A:    --|s.write(1)|-------------------------
B:    ------|r.read():1|---|r.read():0|------
```

Equivalence

For all threads T: H|T = G|T

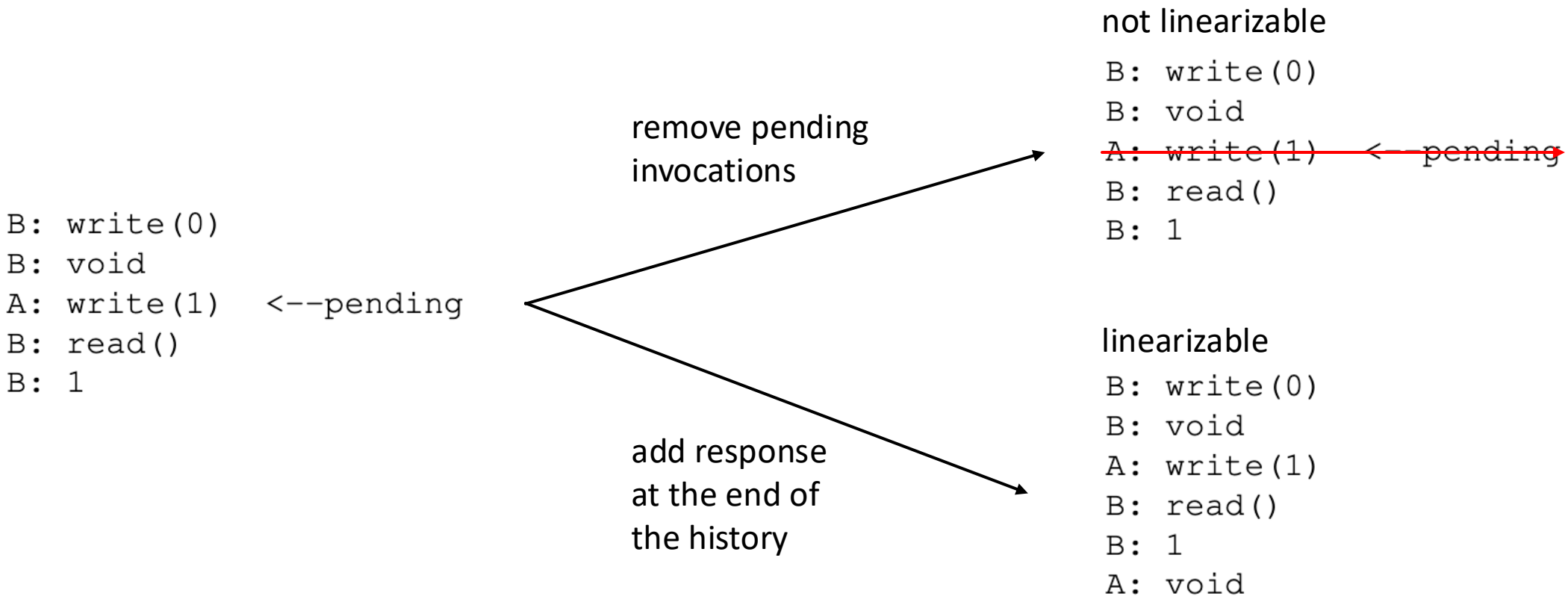
```
A: r.read()
A: 0
C: r.write(1)
C: void
B: r.read()
B: 1
```

is equivalent to

```
B: r.read()
B: 1
C: r.write(1)
C: void
A: r.read()
A: 0
```

Incomplete Histories

When histories are obtained from a program trace, the history might be incomplete.
This can be dealt with in two ways.
Why do we need both ways? Give an example where discarding all pending invocations will lead to a non-linearizable history, but adding a response will lead to a linearizable history.



Sequential Consistency vs Linearizability

```
A: write(1)
A: void
B: read()
B: 0
```

SC but not linearizable

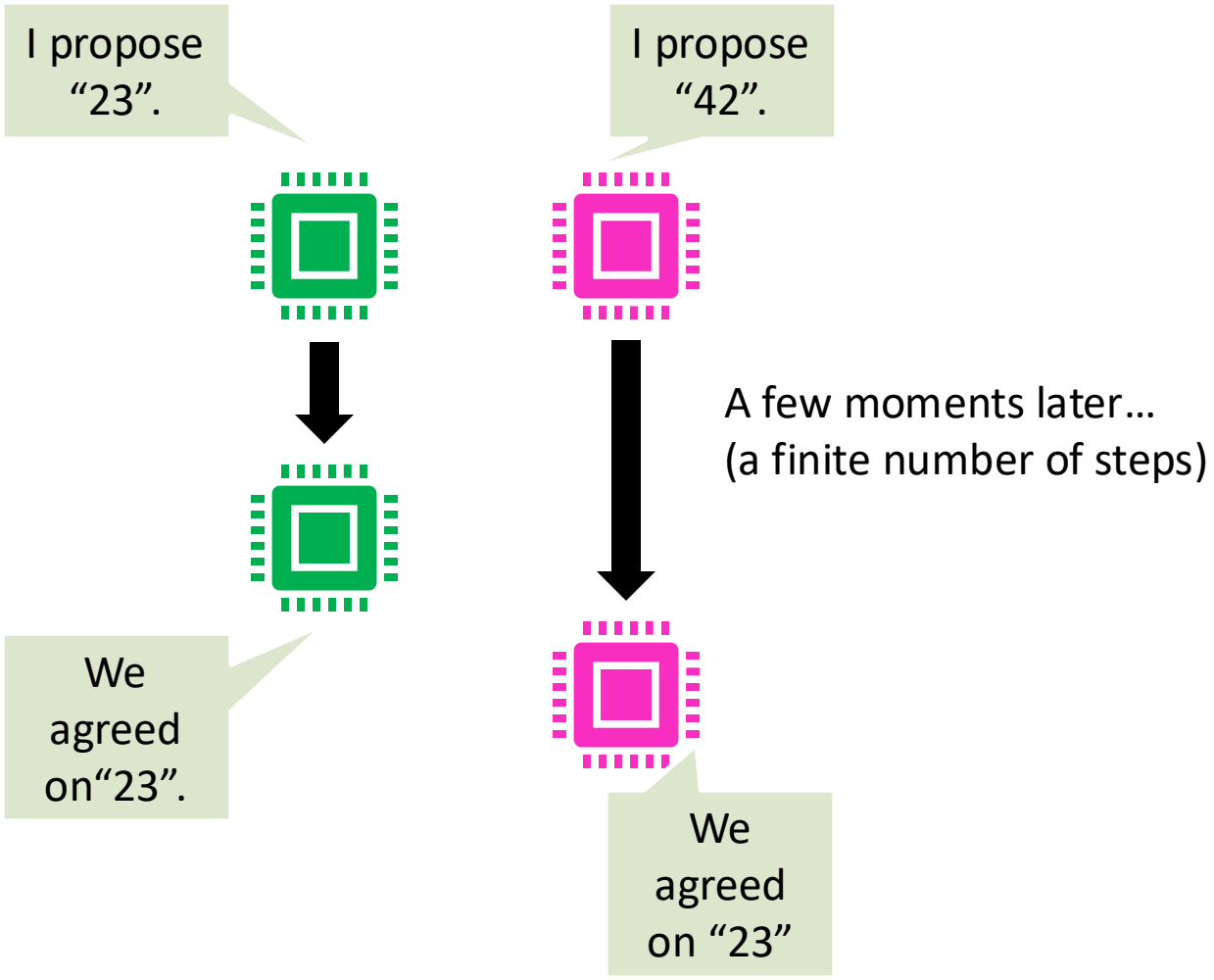
Pre-Discussion Assignment 14

Consensus

- Code snippets: argue about consistency (result same for each thread), validity (result was proposed by a thread) and wait-freedom
- Implement consensus protocol with wait-free FIFO queue
- Equivalence between consensus and binary consensus for two threads

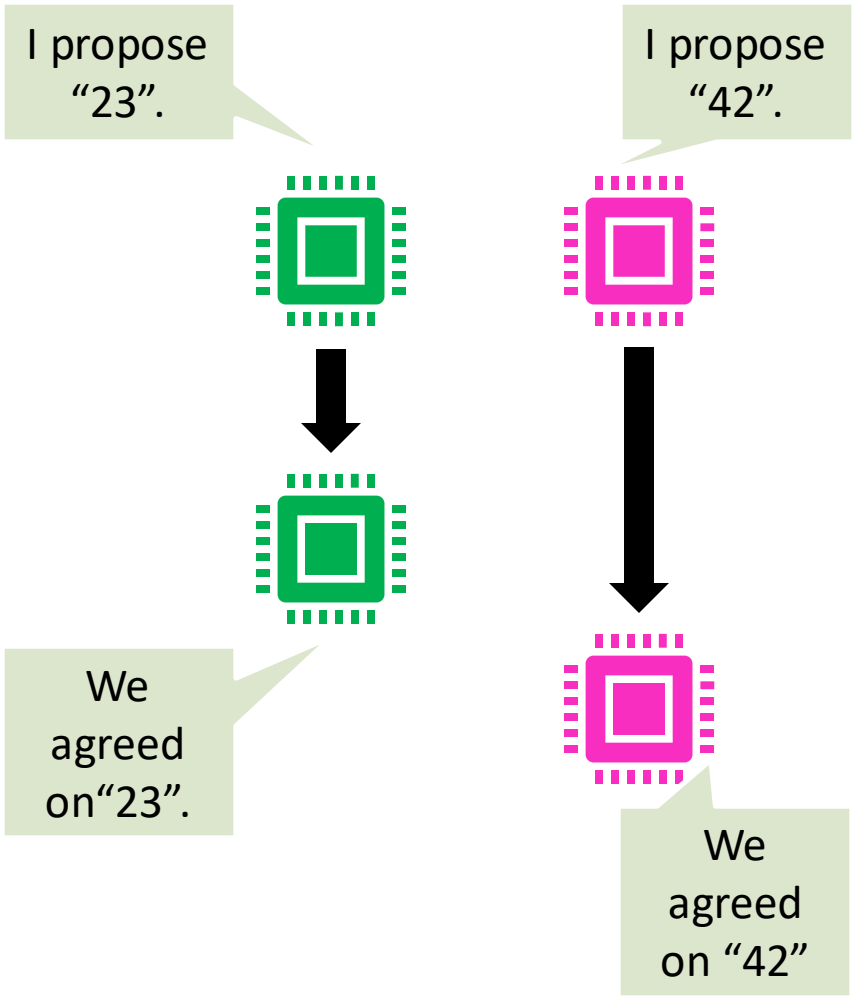
Theory

Recap: Consensus Protocols



Which other scenarios are allowed?

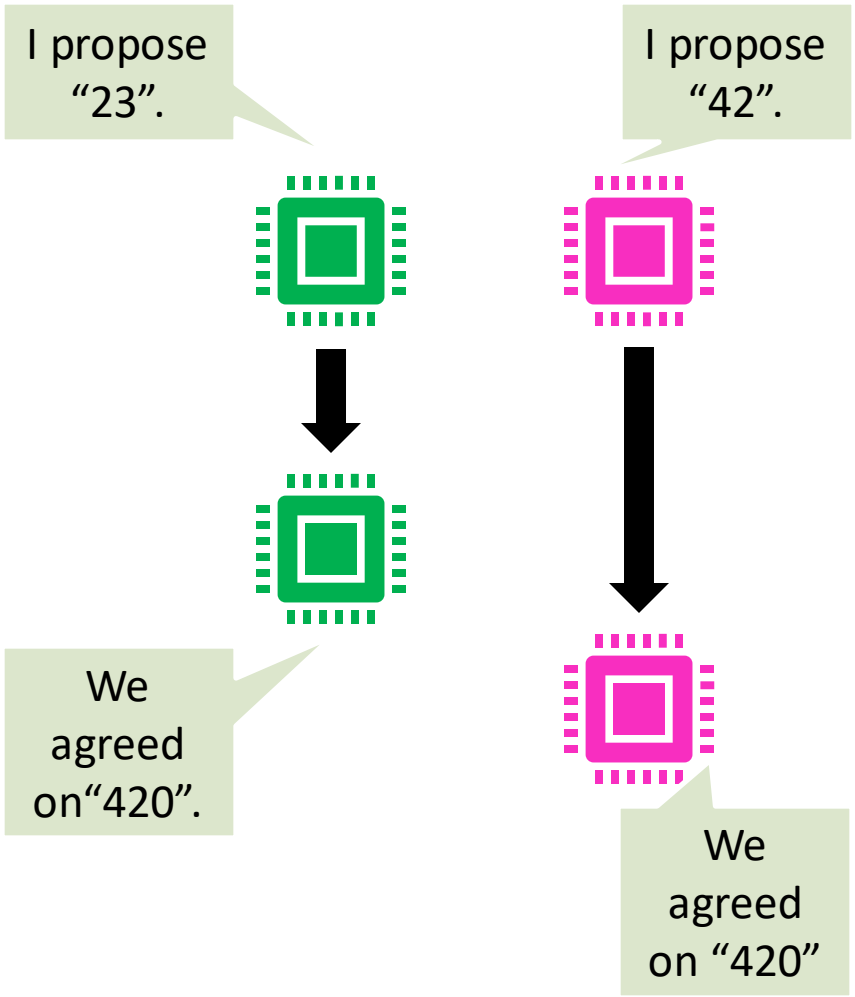
Consistent Result



This is illegal!

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

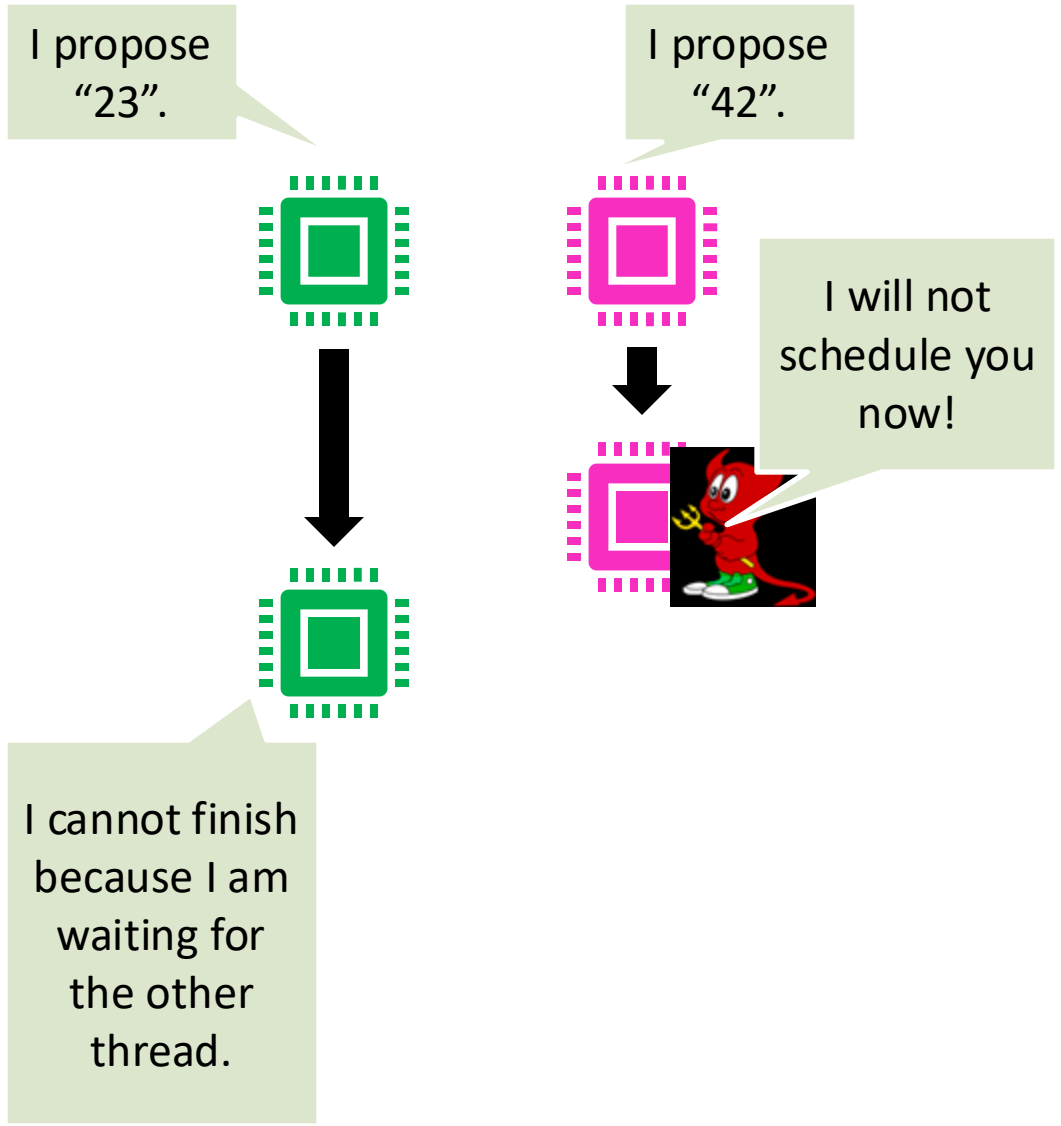
Valid Result



This is illegal!

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

Wait-Free

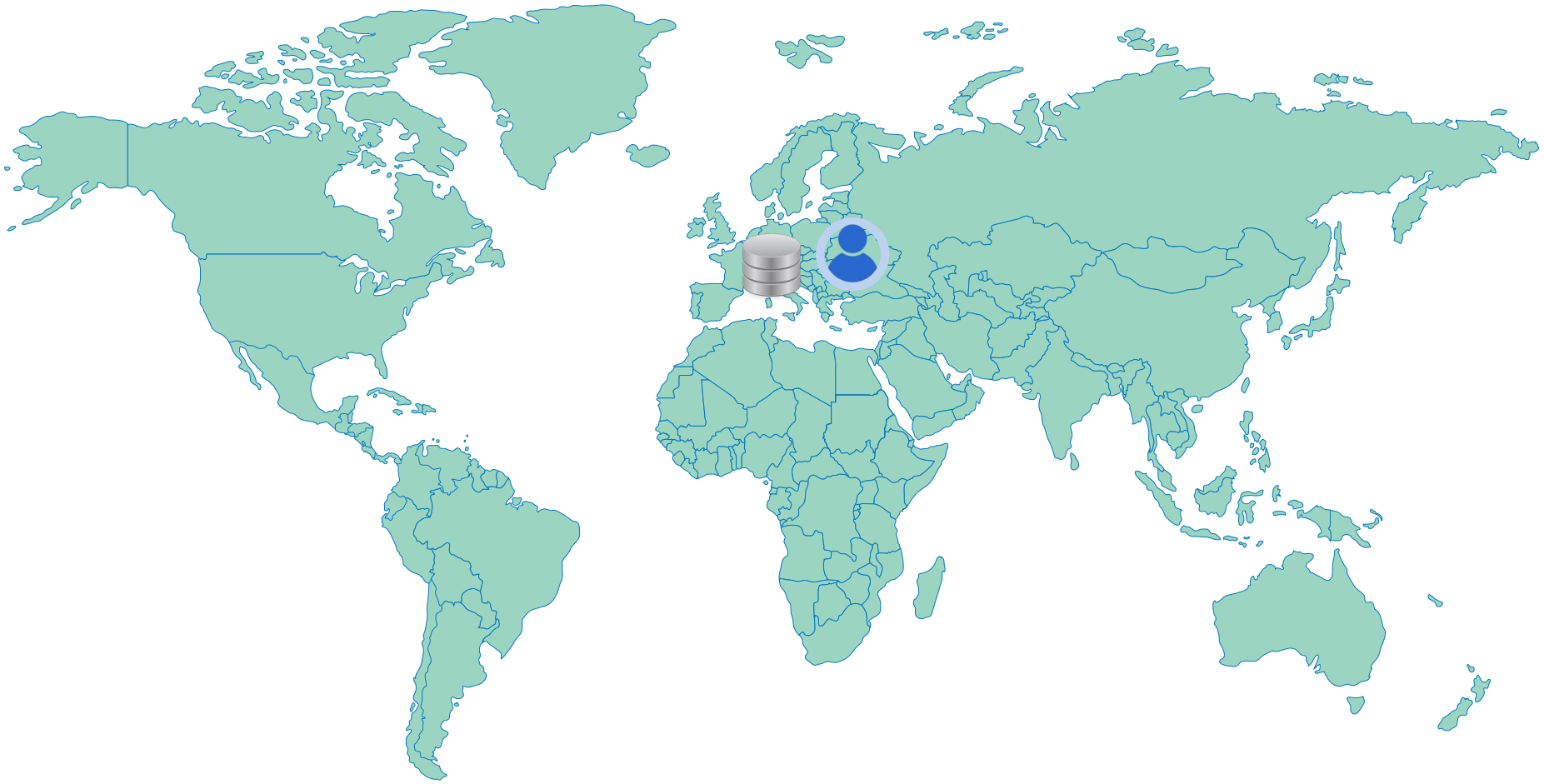


This is illegal!

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

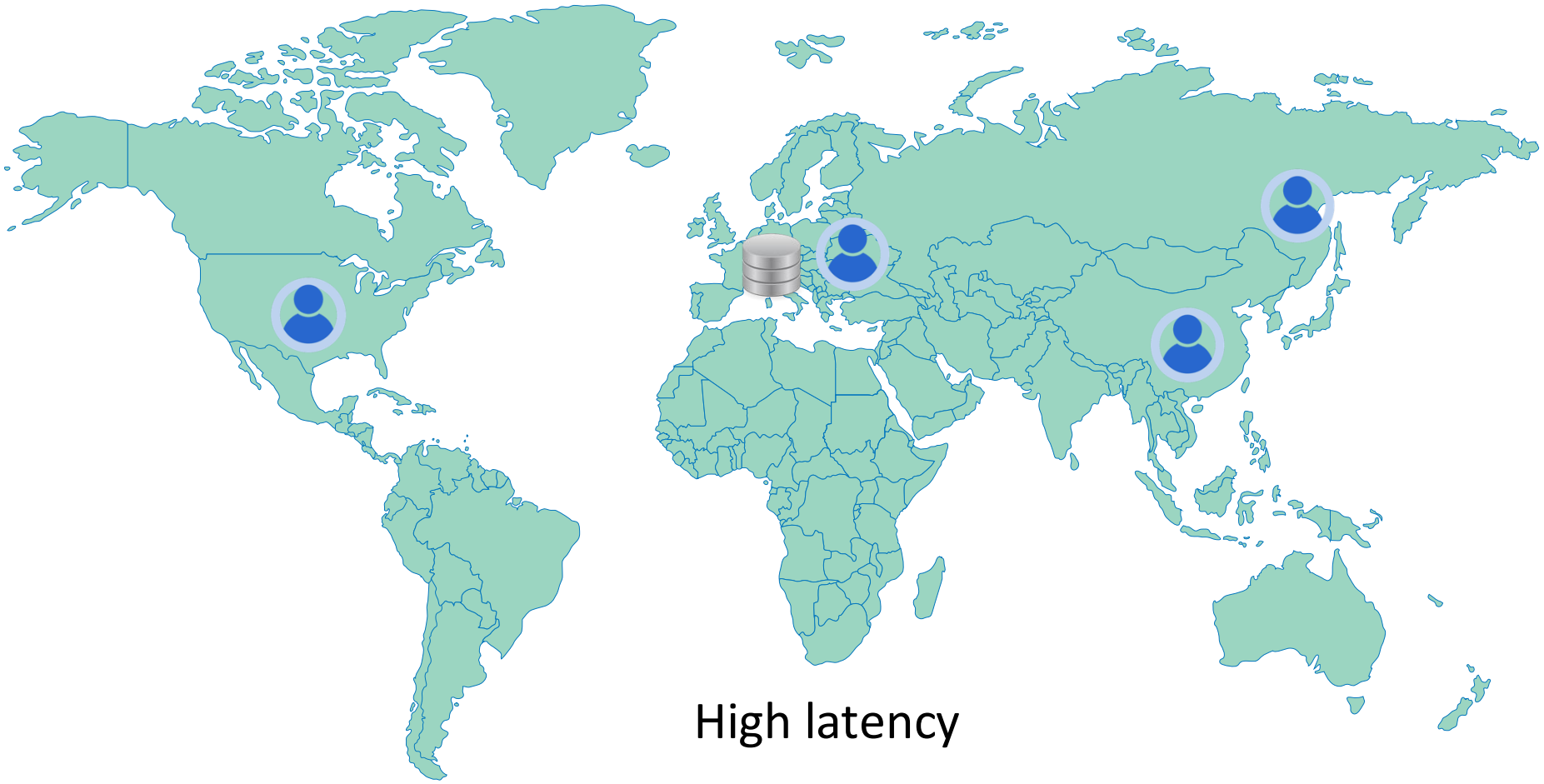
Consensus: Motivation

Example: Databases (of social networks)



Consensus: Motivation

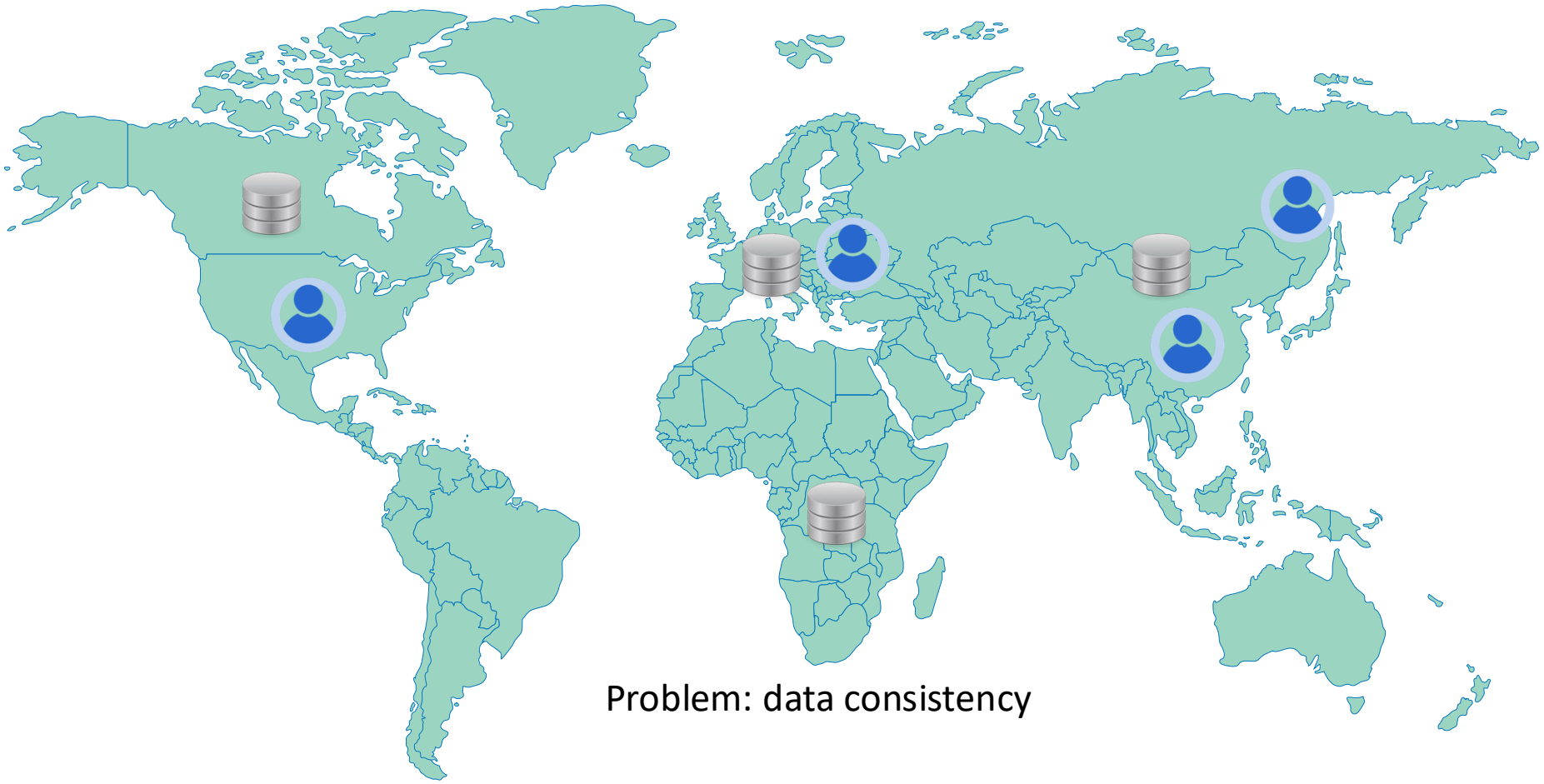
Example: Databases (of social networks)



High latency

Consensus: Motivation

Example: Databases (of social networks)



Problem: data consistency

Consensus Number

„The **consensus number** of a concurrent object is defined to be the maximum number of processes in the system which can reach consensus by the given object in a wait-free implementation. “

atomic read/write registers, mutex: 1

TAS, wait-free queue & stack: 2

CAS: ∞

Note: wait-free queue & stack cannot be implemented with atomic read/write registers

Atomic read/write registers: Consensus Number 1

Proof: There is no wait-free implementation of n-thread consensus (n > 1) with atomic read/write registers.

Proof simplification

Proof that is does not work for 2 threads to show that it does not work for n threads.

Why can we make this assumption?

Assume our consensus protocol is correct (consistent, valid, **wait-free**).

Assume n - 2 threads die / get descheduled.

Since our consensus is **wait-free**, it should still work for the two remaining 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?

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

We can implement binary consensus using normal consensus.

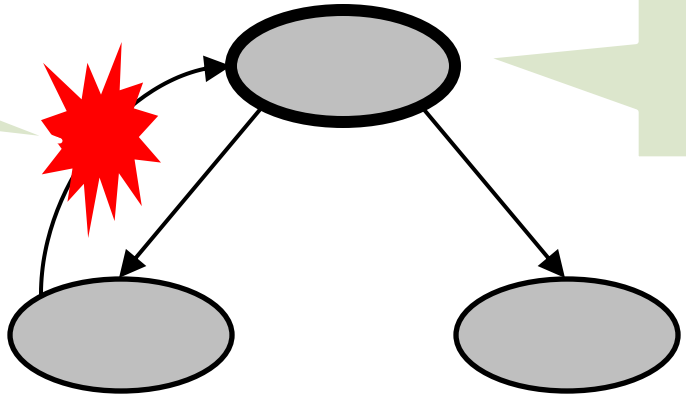
```
int_decide(int d) {
  prop[id] = d; //prop is shared
  other = (id + 1)%2;
  int win = bin_decide(id);
  return prop[win];
}
```

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

State Diagrams of Two-thread Consensus Protocols

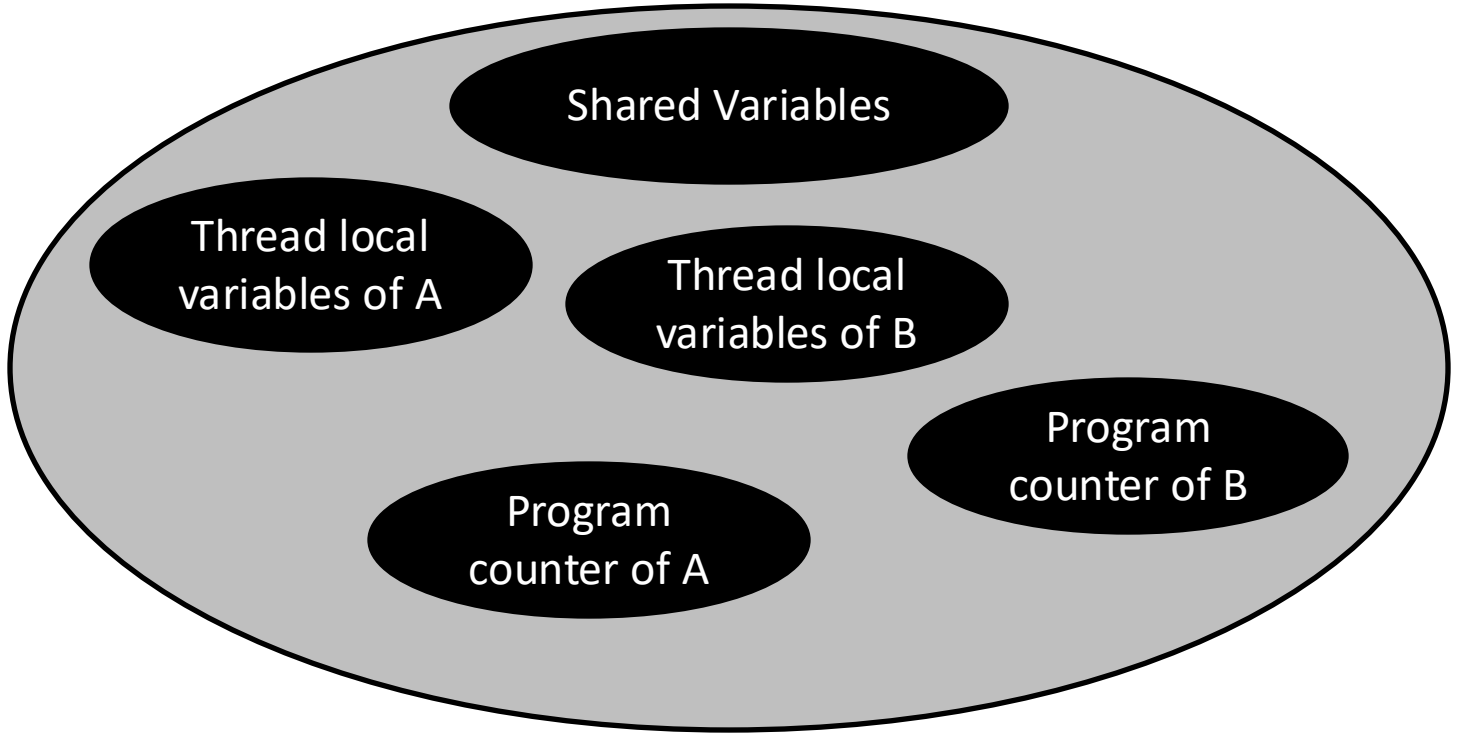
Cycles among states cannot exist in a wait-free algorithm: The state “looks” the same each time we visit, so we are trapped forever in the loop and not wait-free.

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

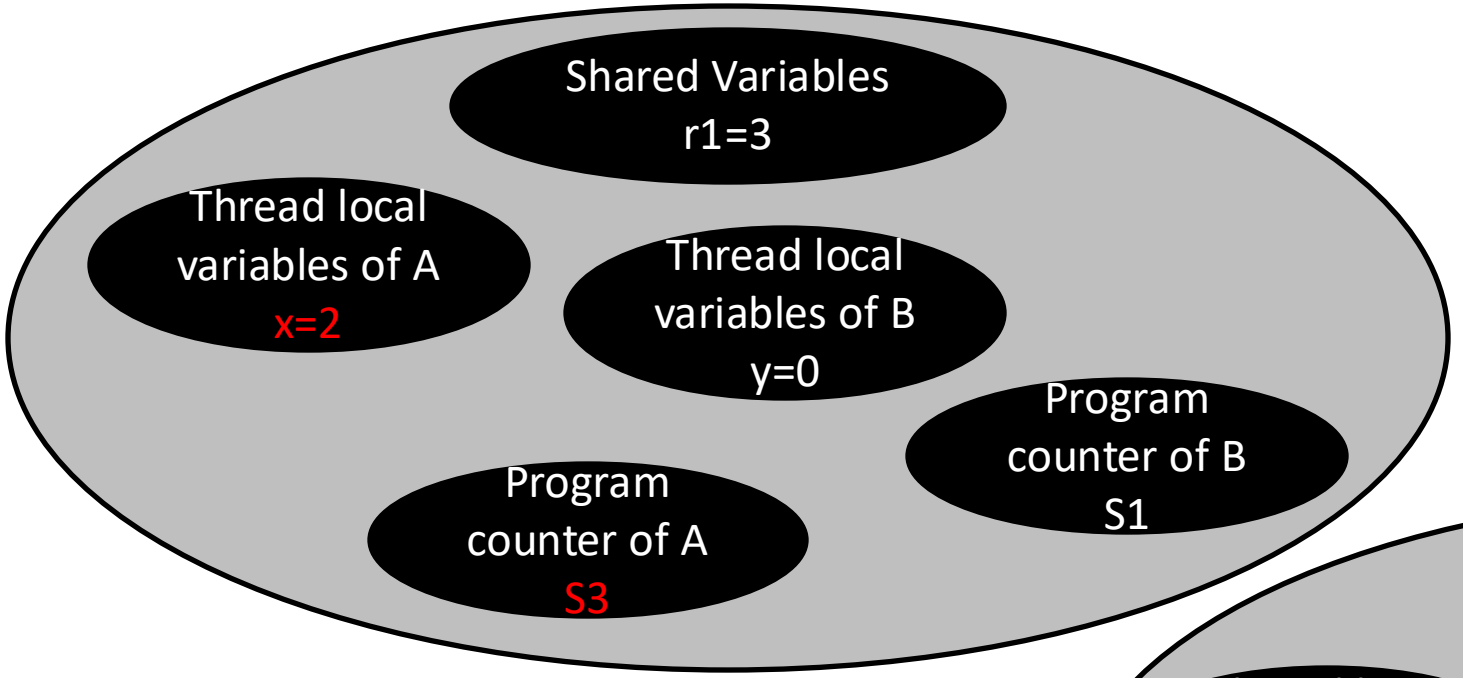


Start state, both threads (A and B) have not yet executed the first instruction of the consensus protocol.

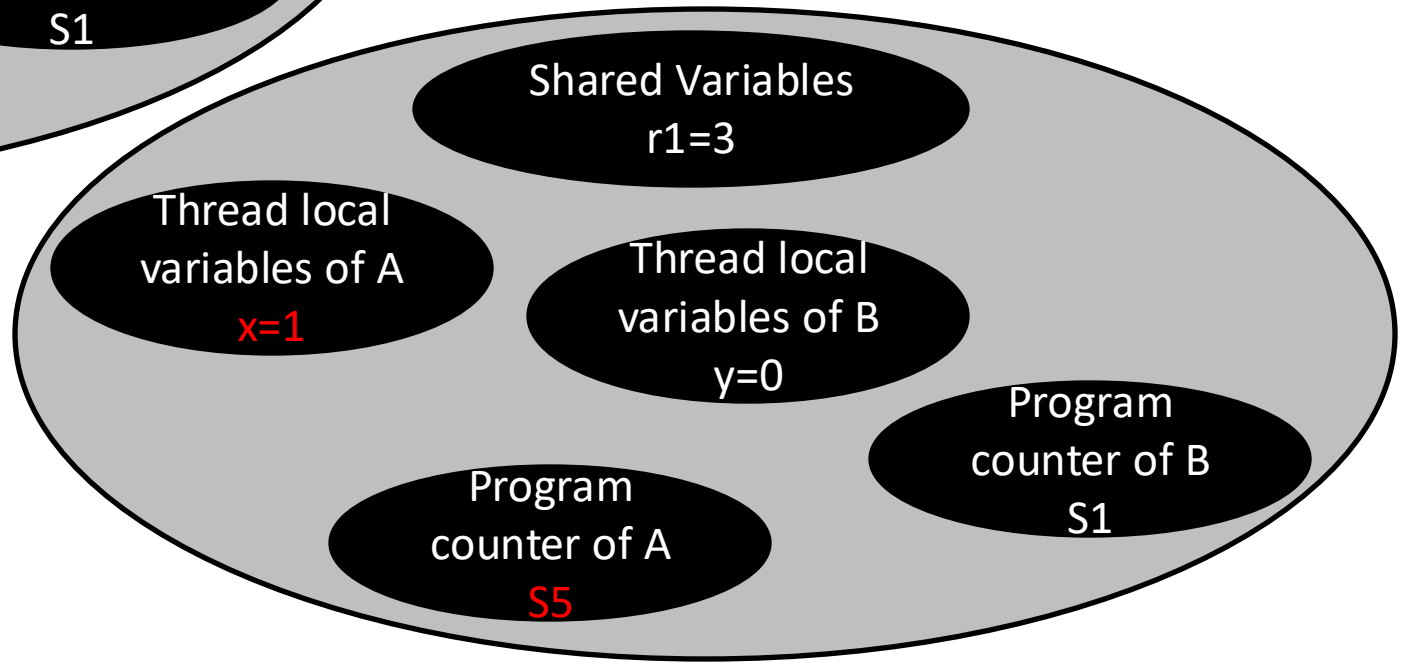
Anatomy of a State (in two-thread consensus)



Anatomy of a State



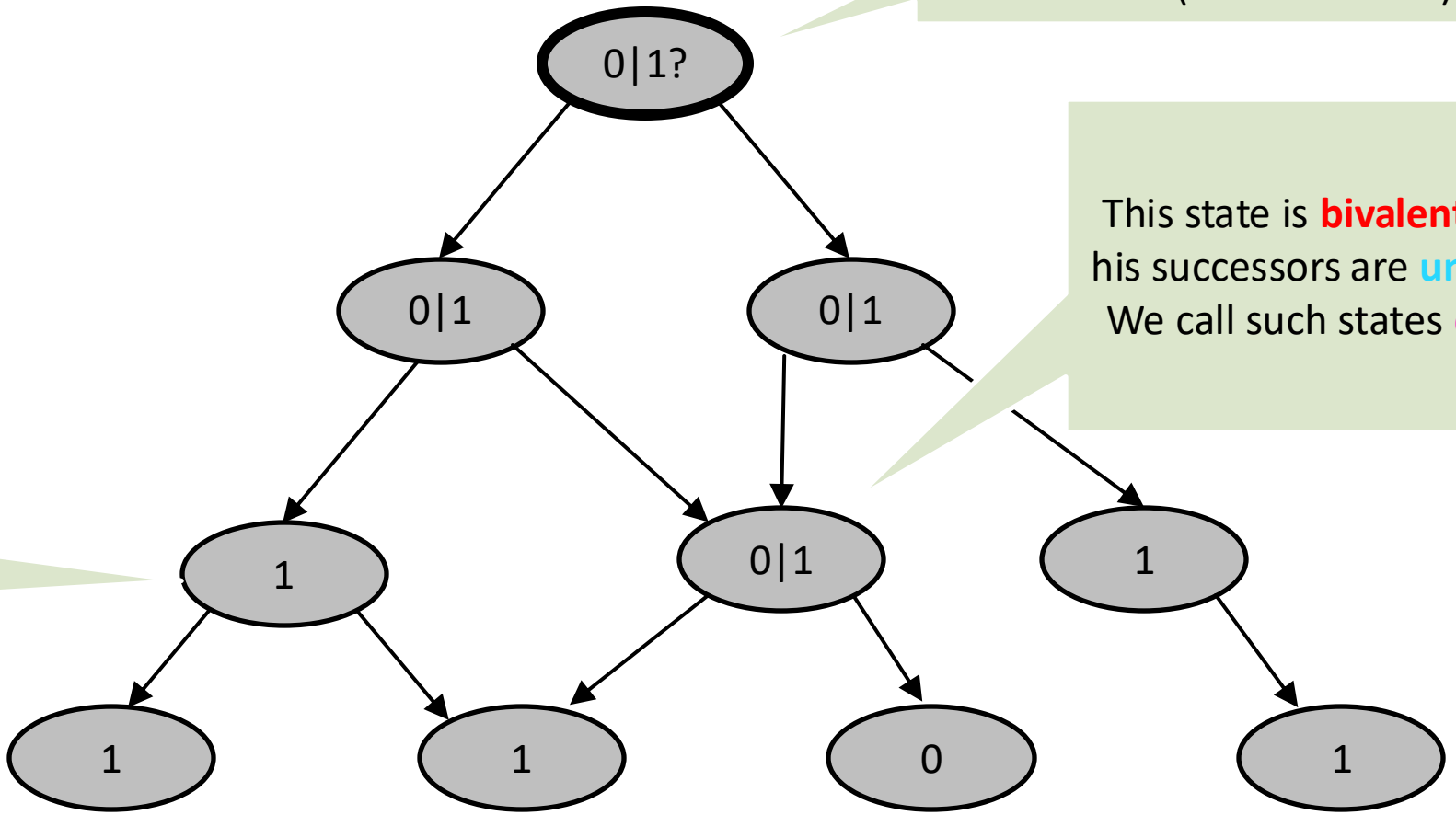
The states are different, since A has different local variables and program counter values.



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

Critical States

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

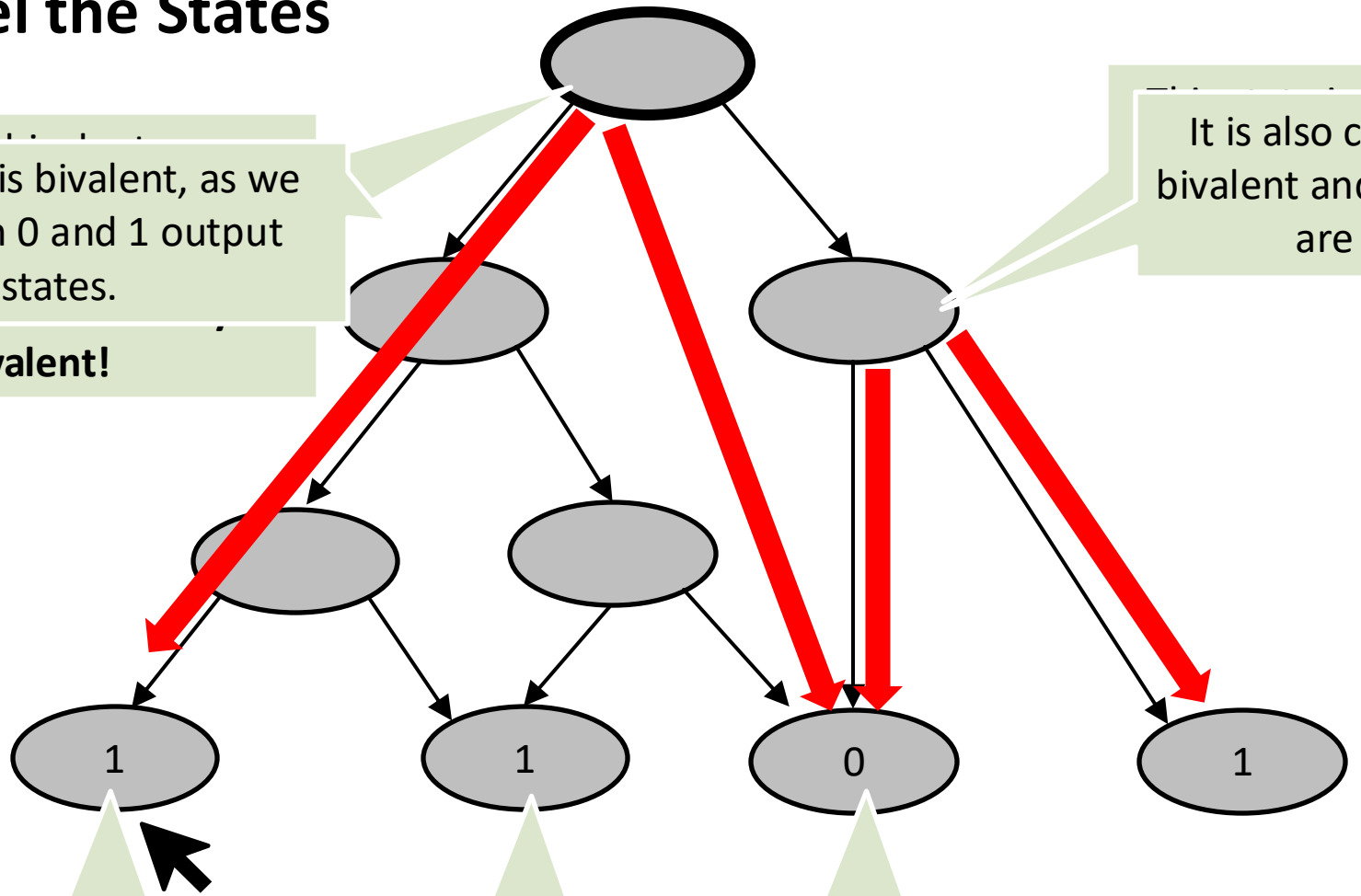


This state is **bivalent** but all his successors are **univalent**. We call such states **critical**.

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

Output states are always **univalent.**

Quiz: Label the States



This state is bivalent, as we can reach 0 and 1 output states.

bivalent!

It is also critical, since it is bivalent and all its successors are univalent.

Output states are alv
univalent.

Output states a
univaler

Output states are always **univalent.**

Critical State Existence Proof

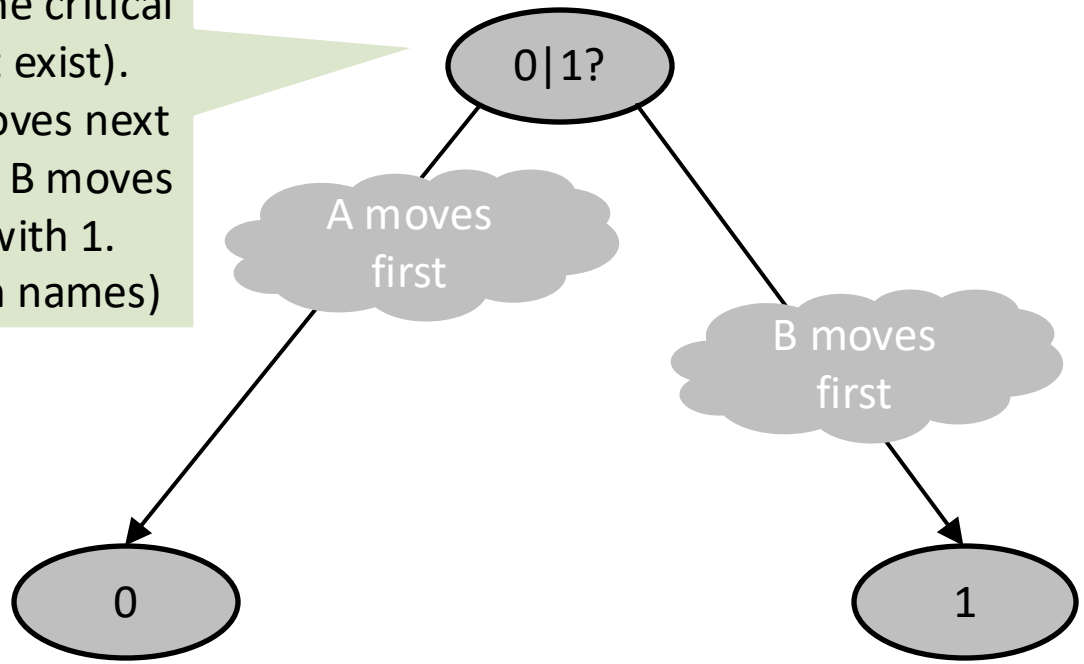
Lemma: Every consensus protocol has a critical state.

Proof: From (bivalent) start state, let the treads 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

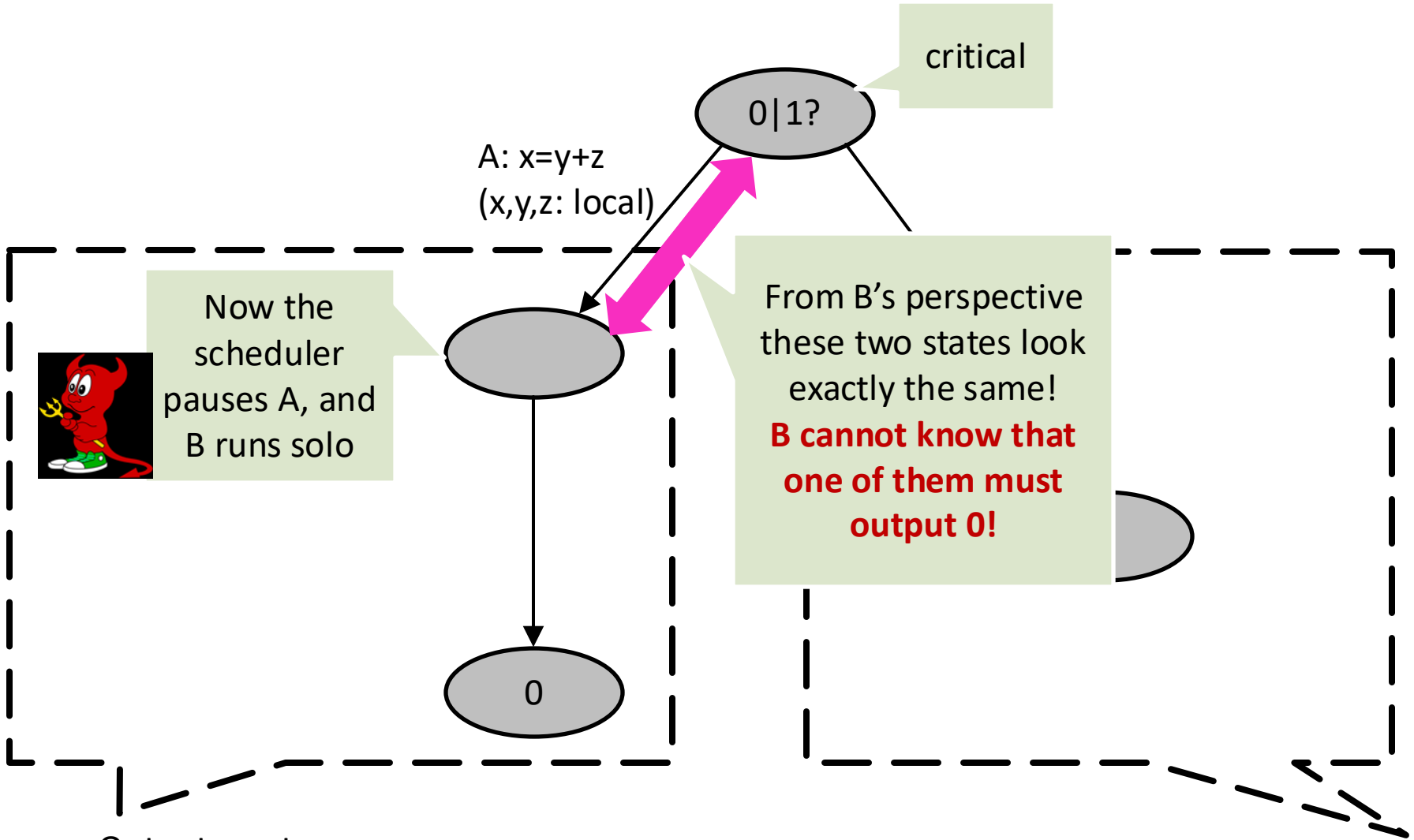
Assume we are in the critical state (which must exist). Assume that if A moves next we end up with 0, if B moves next we end up with 1. (w.l.o.g., can switch names)



So what actions can a thread perform in his “move”?

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

Impossibility Proof Setup – Possible actions of a thread

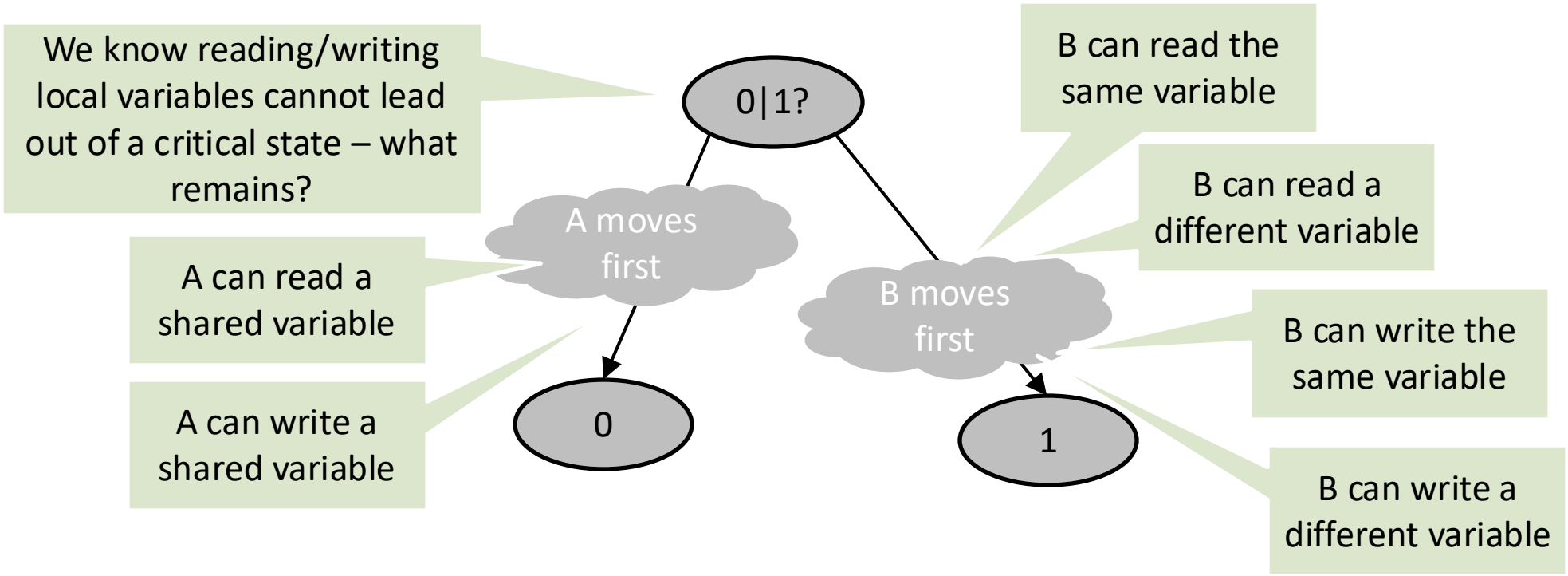


So what actions can a thread perform in his "move"?

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



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?

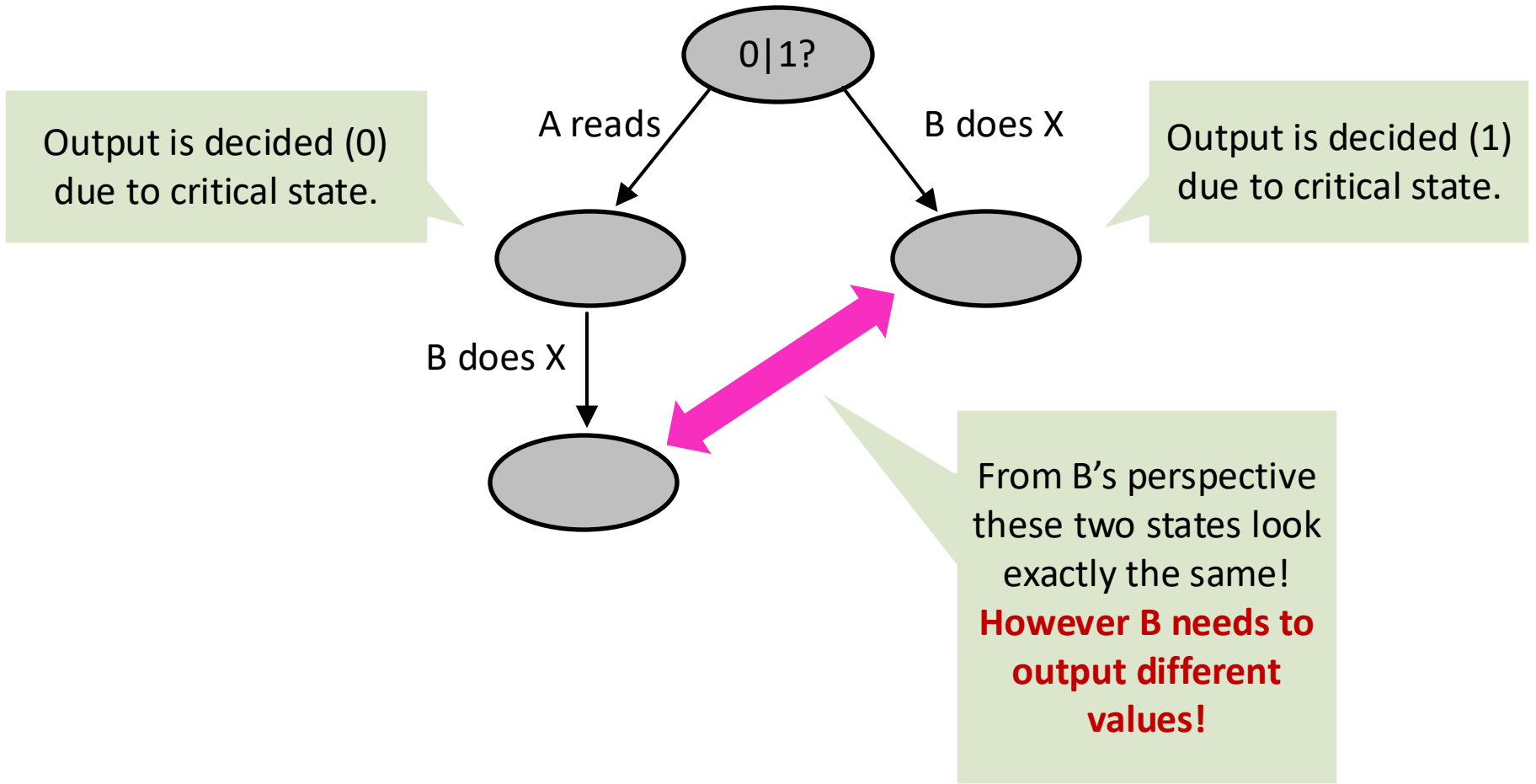
		Second Action			
		A: r1.read()	A: r2.read()	A: r1.write()	A: r2.write()
First Action	B: r1.read()		?		
	B: r2.read()				
	B: r1.write()				
	B: r2.write()				

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

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

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

Impossibility Proof Case I: A reads

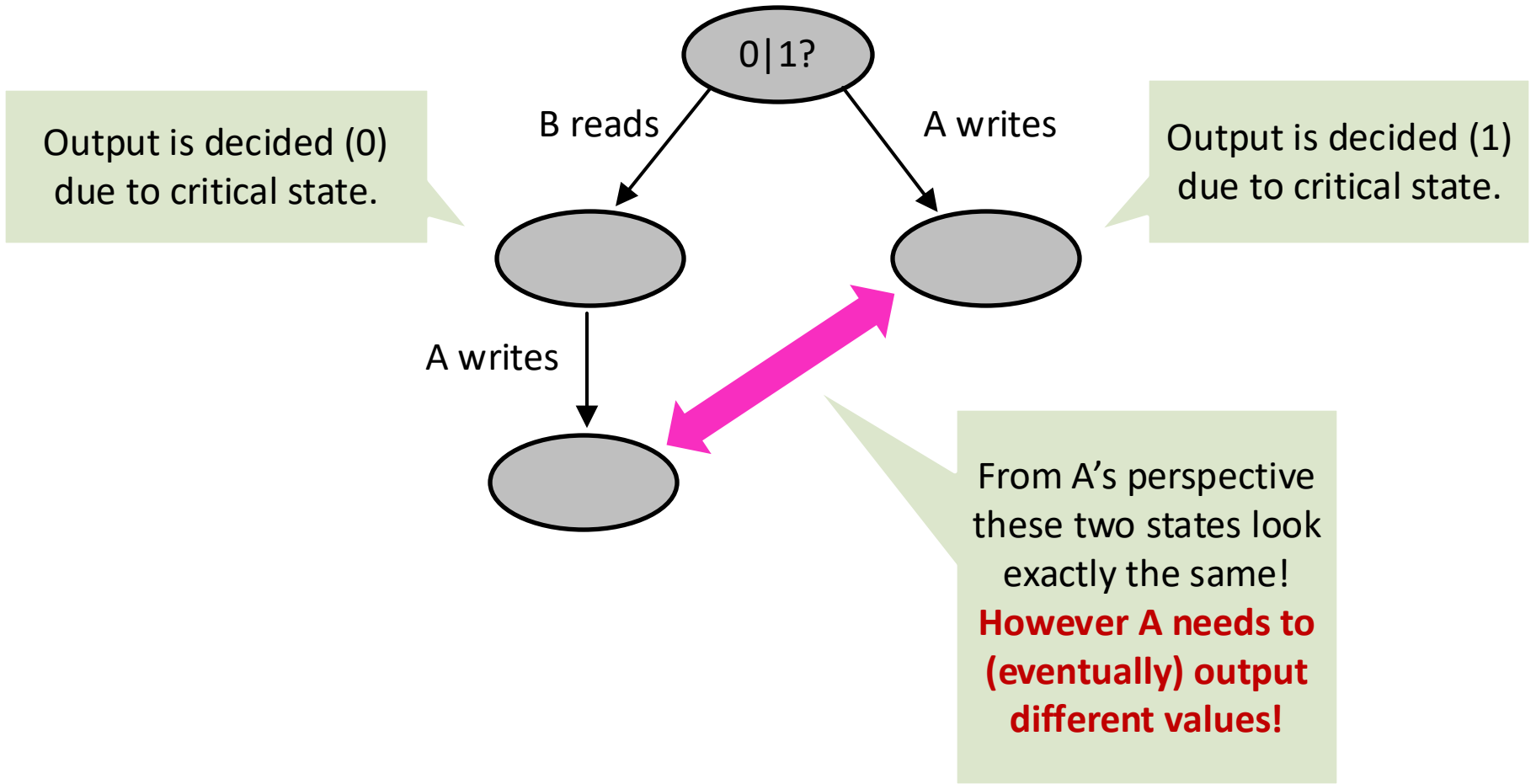


What did we just prove?

		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

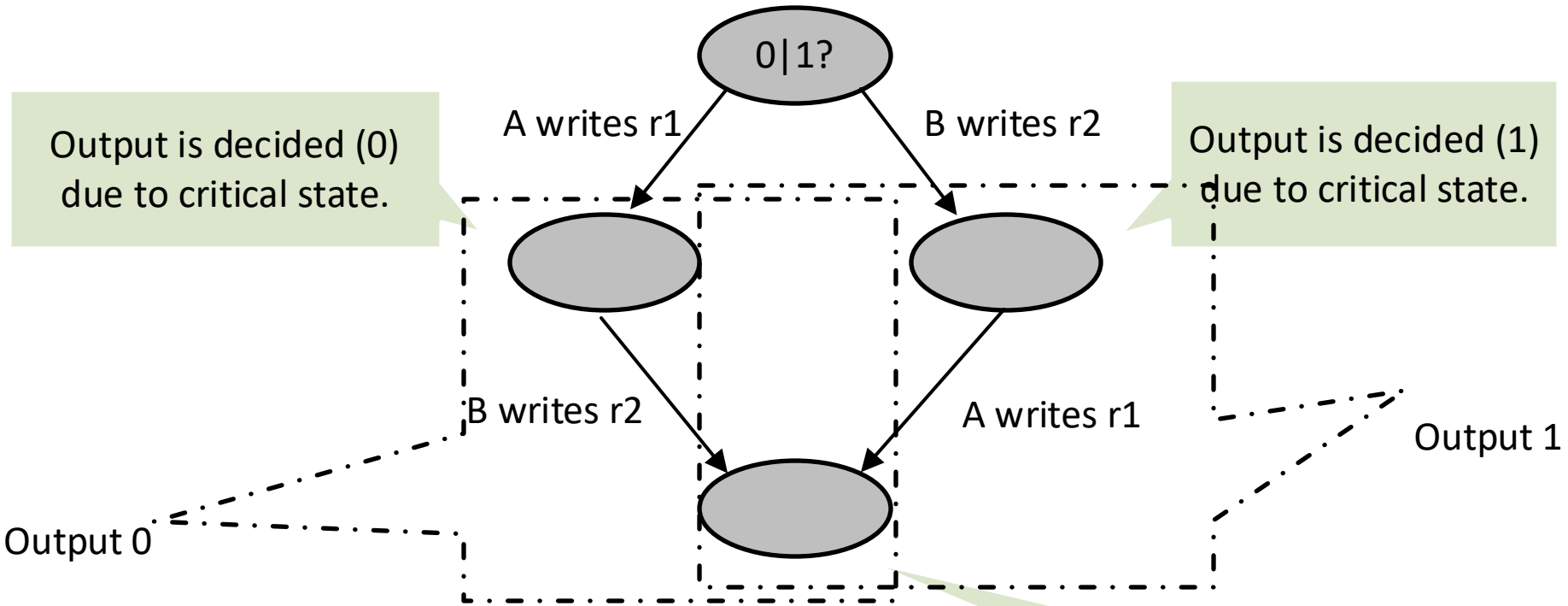


What did we just prove?

		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

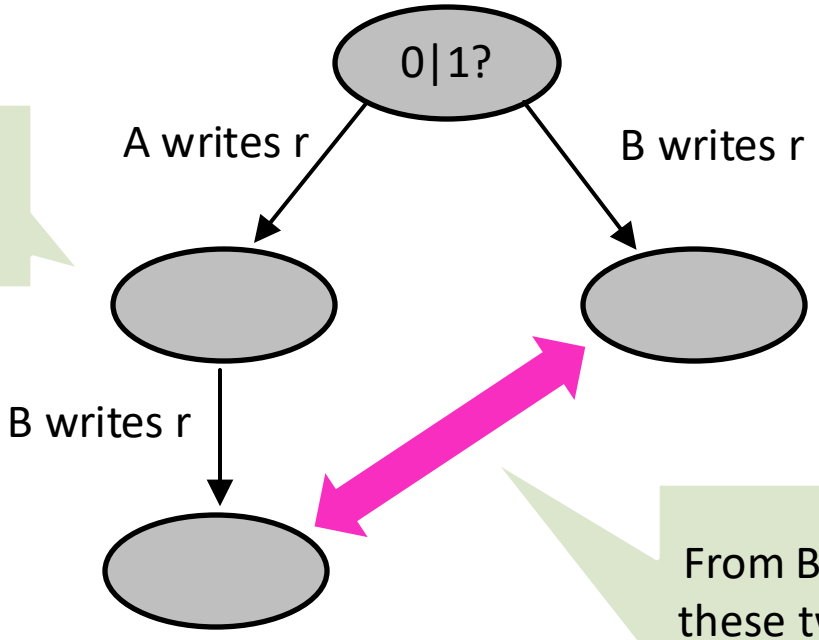


What did we just prove?

		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

What did we prove?

- **Atomic read/write registers have consensus number 1**

Consensus: TAS consensus number

Proof: TAS has consensus number 2

Proof outline

- First proof that TAS has consensus number n >= 2 by construction
- Then proof that TAS has consensus number n < 3 by contradiction

➔ n needs to be 2

Implementing two thread consensus with TAS

- **Assume you have a machine with atomic registers and an atomic test-and-set operation with the following semantics (mem[s] is initially 0):**

```
boolean TAS(memref s) {
  if (mem[s] == 0) {
    mem[s] = 1;
    return true;
  }
  return false;
}
```

- **Implement a wait-free two-process consensus protocol using TAS and atomic registers.**

Implementing two thread consensus - Solution

- **Pseudo-Code for both threads**

```
AtomicIntegerArray proposed = new AtomicIntegerArray(2);
flag = 0;

int decide (int value) {
        int i = ThreadID.get();
        proposed.set(i, value);
        if (TAS(flag)) {
                return value;
        }
        else {
                return proposed.get((i + 1) % 2);
        }
}
```

TAS consensus number: proof outline that n < 3

- **Assume there exist some correct wait-free consensus protocol with TAS for n > 2 threads.**
- **Proof that it does not work for 3 threads (which implies that it does not work for n > 3 threads)**

- **Go over all cases just like in the proof for atomic read/write registers**
- **Difference to proof for atomic read/write registers**
  - Each node can have 3 children
  - Multivalent would be a better word for states that did not decide yet

Consensus: CAS consensus number

Proof: CAS has consensus number ∞

Proof by construction: CAS consensus number ∞

```
class CASConsensus {
   private final int FIRST = -1;
   private AtomicInteger r = new AtomicInteger(FIRST); // supports CAS
   private AtomicIntegerArray proposed; // suffices to be atomic register

   … // constructor (allocate array proposed etc.)

   public Object decide (Object value) {
      int i = ThreadID.get();
      proposed.set(i, value);
      if (r.compareAndSet(FIRST, i)) // I won
         return proposed.get(i); // = value
      else
         return proposed.get(r.get());
   }
}
```

Consensus: What should you definitely know for the exam

- Know about properties: consistent, valid, wait-free
- Bivalent, univalent, critical state
- Argue about properties: why acyclic graph?
- Know consensus numbers
- Argue with consensus numbers

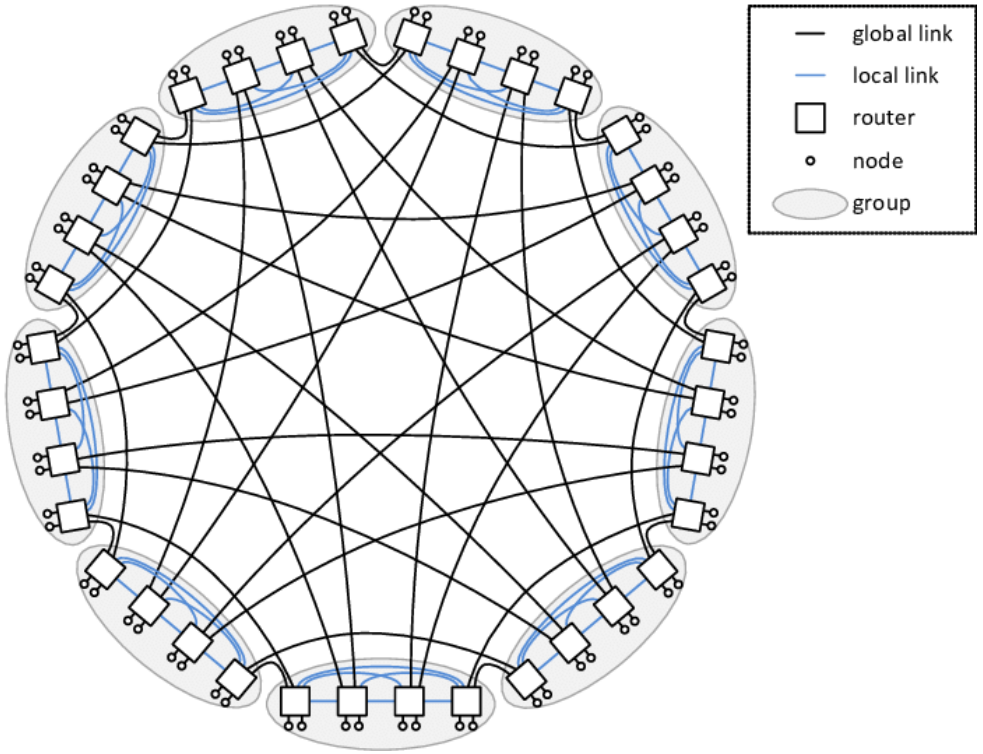
MPI (Message Passing Interface)

MPI: Motivation

- Locking is too slow
- Shared memory is not always viable
- What if we have multiple compute nodes and want to distribute work?

High Performance Computing: Supercomputers

- Connect many different computers (cluster) and let them communicate with each other over high speed interconnects (e.g. Infiniband)
- Example: Dragonfly-Topology



Infiniband demo

```
[eli@sbrinz1 ~]$ ib_read_bw

************************************
* Waiting for client to connect... *
************************************
---------------------------------------------------------------------------------------
                    RDMA_Read BW Test
 Dual-port       : OFF          Device         : mlx5_0
 Number of qps   : 1            Transport type : IB
 Connection type : RC           Using SRQ      : OFF
 PCIe relax order: ON           Lock-free      : OFF
 ibv_wr* API     : ON           Using DDP      : OFF
 CQ Moderation   : 1
 Mtu             : 4096[B]
 Link type       : IB
 Outstand reads  : 16
 rdma_cm QPs     : OFF
 Data ex. method : Ethernet
---------------------------------------------------------------------------------------
 local address: LID 0x03 QPN 0x18bac PSN 0xdfad59 OUT 0x10 RKey 0xb83a00 VAddr 0x00400000529000
 remote address: LID 0x03 QPN 0x18bab PSN 0x6c0903 OUT 0x10 RKey 0xb69600 VAddr 0x00400000529000
---------------------------------------------------------------------------------------
 #bytes     #iterations    BW peak[MiB/sec]    BW average[MiB/sec]   MsgRate[Mpps]
 65536      1000             39185.09            39155.05                     0.626481
---------------------------------------------------------------------------------------
```

MPI

- Message Passing **Interface** defines the semantics
- There are many different MPI implementations: e.g. OpenMPI, MPICH

→ When you build a program using MPI, you can decide which implementation to use

→ Different implementations can lead to different performance, but the semantic is the same as defined by the MPI standard

→ One is also free to choose the number of MPI-ranks & processes

MPI

```
MPI.Init(args);
int rank = MPI.COMM_WORLD.Rank();
int size = MPI.COMM_WORLD.Size();

// CODE...

MPI.Finalize();
```

https://luck-wildflower-a5b.notion.site/pprog-summary-leon-thomm

MPI

```
comm.send(
  Object buf,              // the data object to be sent
  int offset,              // start offset from the data start item
  int count,               // number of items to be sent
  Datatype datatype,       // data type of items
  int dest,                // destination process id
  int tag                  // data id tag
)
```

```
comm.recv(
  Qbject buf,              // the object to write the data to
  int offset,              //
  int count,               // number of items to be receiv
  Datatype datatype,       // data type of items
  int src,                 // souce process id or MPI_ANY_SOURCE
  int tag                  // data id tag MPI_ANY_TAG
)
```

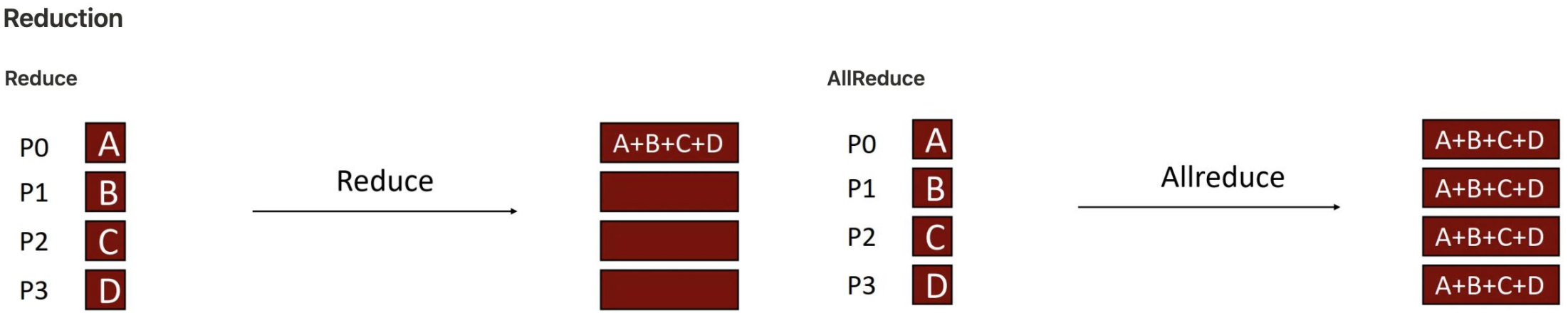
https://luck-wildflower-a5b.notion.site/pprog-summary-leon-thomm

MPI barrier

```java
public void mybarrier() {
    int rank = MPI.COMM_WORLD.Rank();
    int size = MPI.COMM_WORLD.Size();
    boolean[] buf = new boolean[1];

    if (rank == 0) {
       // Wait until every process sent a meaningless boolean
       for (int i = 1; i < size; i++) {
           MPI.COMM_WORLD.Recv(buf, 0, 1, MPI.BOOLEAN, i, 0);
       }

       // All processes are ready

       // Send a signal to all processes so that they can continue
       for (int i = 1; i < size; i++) {
           MPI.COMM_WORLD.Send(buf, 0, 1, MPI.BOOLEAN, i, 0);
       }
    } else {
         MPI.COMM_WORLD.Send(buf, 0, 1, MPI.BOOLEAN, 0, 0); // send meaningless boolean
         MPI.COMM_WORLD.Recv(buf, 0, 1, MPI.BOOLEAN, 0, 0); // wait until process 0 gives signal
    }
 }
```

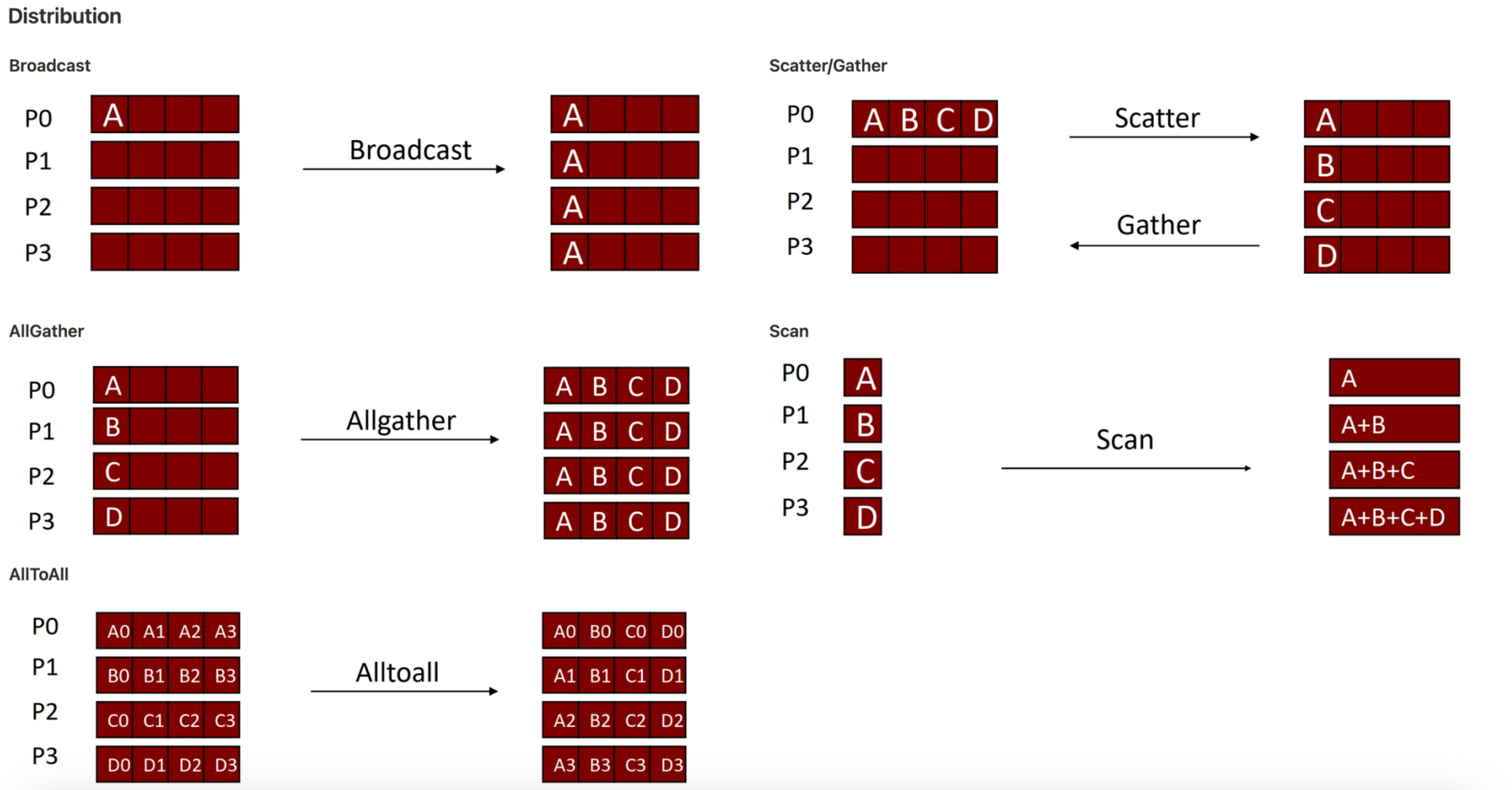
https://luck-wildflower-a5b.notion.site/pprog-summary-leon-thomm

Collective Computation paradigms



https://luck-wildflower-a5b.notion.site/pprog-summary-leon-thomm

Collective Computation paradigms



https://luck-wildflower-a5b.notion.site/pprog-summary-leon-thomm

Exam tasks

Consensus

11. Welche der folgenden Methoden sind korrekte Implementierungen von Wait-free Consensus für die angegebene Zahl N an Threads? Begründen Sie Ihre Antworten.

Which of the following Java Methods are valid implementations of wait-free consensus for the given number N of threads? Justify your answers.

(a) $N = 1$ (2)

```
1 int consensus(int x) {
2       return 0;
3 }
```

(b) $N = 1$ (2)

```
1 int consensus(int x) {
2     return x;
3 }
```

Consensus

11. Welche der folgenden Methoden sind korrekte Implementierungen von Wait-free Consensus für die angegebene Zahl N an Threads? Begründen Sie Ihre Antworten.

Which of the following Java Methods are valid implementations of wait-free consensus for the given number N of threads? Justify your answers.

(a) $N = 1$ (2)

```
1 int consensus(int x) {
2       return 0;
3 }
```

Not valid

(b) $N = 1$ (2)

```
1 int consensus(int x) {
2     return x;
3 }
```

correct

(c) $N = 2$ (2)

```
1 volatile int decision;
2 volatile bool decided = false;
3
4 synchronized int consensus(int x) {
5       if (!decided)
6           decision = x;
7       decided = true;
8       return decision;
9 }
```

(d) $N = 2$ (2)

```
1 AtomicInteger dec = 0;
2 int[] prop = new int[] {0,0};
3
4
5 int consensus(int x) {
6       prop[dec.Get()] = x;
7       t = dec.IncrementAndGet();
8       return prop[t-1];
9 }
```

(c) $N = 2$ (2)

```
1 volatile int decision;
2 volatile bool decided = false;
3
4 synchronized int consensus(int x) {
5       if (!decided)
6           decision = x;
7       decided = true;
8       return decision;
9 }
```

Not wait free

(d) $N = 2$ (2)

```
1 AtomicInteger dec = 0;
2 int[] prop = new int[] {0,0};
3
4
5 int consensus(int x) {
6       prop[dec.Get()] = x;
7       t = dec.IncrementAndGet();
8       return prop[t-1];
9 }
```

Not consistent

Can binary wait-free consensus for two threads be implemented with locks? Explain your answer.

Can binary wait-free consensus for two threads be implemented with locks? Explain your answer.

No, as using locks can lead to threads blocking each other
→ Not wait-free

Can binary wait-free consensus for ~~two~~ one threads be implemented with locks? Explain your answer.

Can binary wait-free consensus for ~~two~~ one threads be implemented with locks? Explain your answer.

Yes

What are the consensus numbers of the following three objects:

a) atomic registers that support only the operations Read() and Write(),

b) atomic registers that also support the TestAndSet() operation, and

c) atomic registers that also support the CompareAndSet() operation?

What are the consensus numbers of the following three objects:

a) atomic registers that support only the operations Read() and Write(), 1

b) atomic registers that also support the TestAndSet() operation, and 2

c) atomic registers that also support the CompareAndSet() operation? ∞

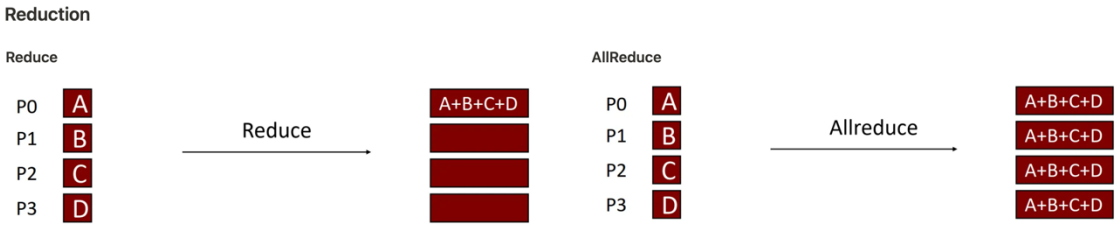
What is the difference between MPI point-to-point and collective operations? Why are collective operations used?

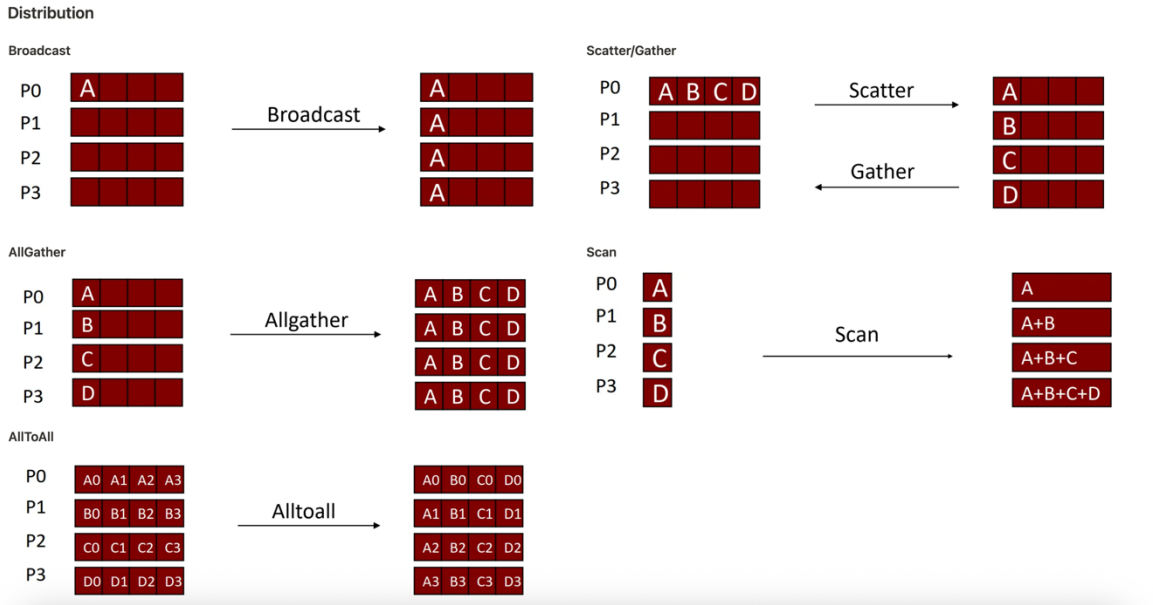
What is the difference between MPI point-to-point and collective operations? Why are collective operations used?

Point-to-Point: One sender to one receiver
Collective operations: see previous slide about „collective computation paradigms“

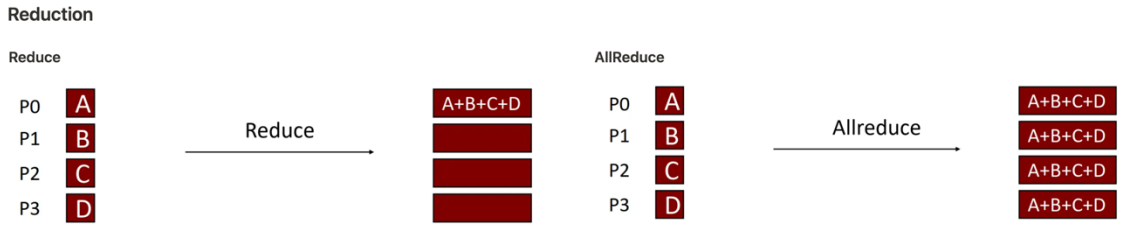
We mainly use collective operations to get scalable performance

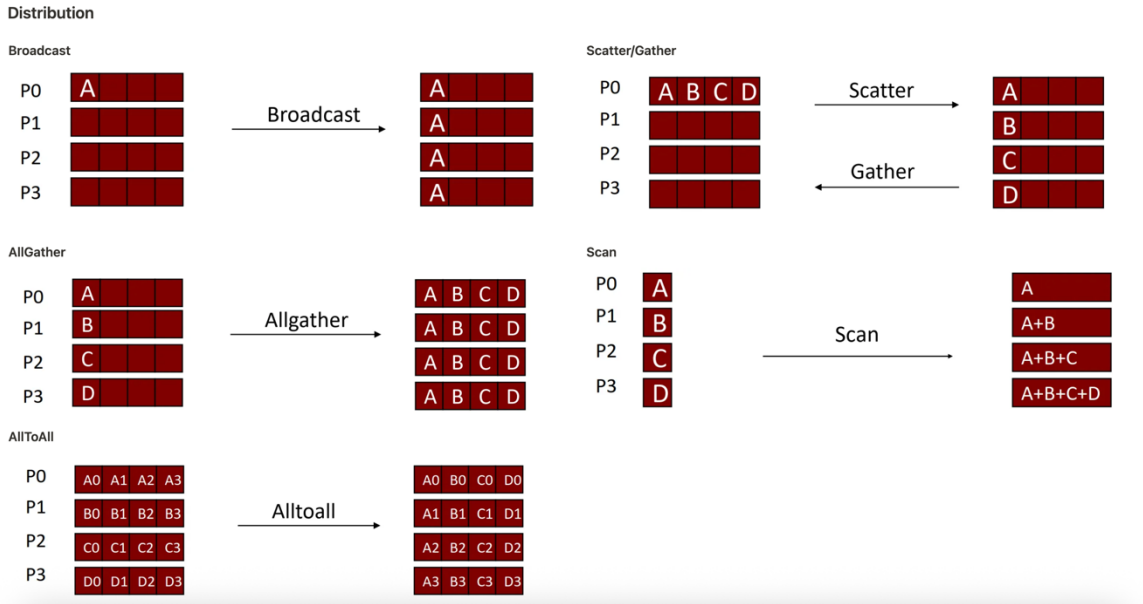
Which two collective operations can be combined to imlement the Allgather operation?





Which two collective operations can be combined to imlement the Allgather operation?





Gather and Broadcast

Ein MPI Programm wird von 8 Prozessen ausgeführt. Jeder Prozess kann zu jeder Zeit maximal an einem Nachrichtenaustausch teilnehmen. Eine Nachricht welche einen double Wert enthält kann innerhalb 1 ms zwischen zwei beliebigen Prozessen ausgetauscht werden. In den unten gezeichneten Schemata (A, B und C) wird jeder Nachrichtenaustausch als gerichtete Kante zwischen Sender und Empfänger dargestellt.

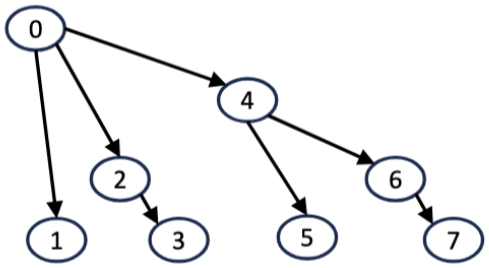
Das Ziel des MPI Programms ist es, einen double Wert von Prozess 0 zu jedem anderen Prozess zu senden. Beantworten Sie für jedes der Schemata unten folgende Fragen: Wird das Ziel erreicht? Was ist die minimale Zeitdauer, bis der letzte Prozess seine Aufgabe erfüllt hat.

An MPI program is executed by 8 processes. ~~Each process can only participate in one communication at a time~~ (either as a sender or a receiver). A message containing a double can be sent between any two processes in 1 ms. In the communication schemes below (A, B, and C) each directed edge represents a message sent from a sender to a receiver process.

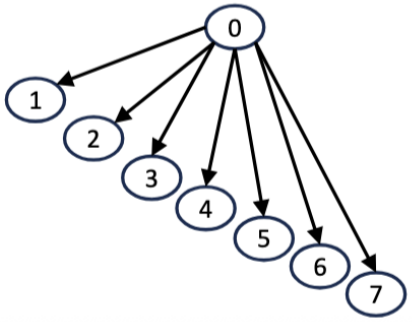
We have point-to-point communication

The goal is to send a double from process 0 to all other processes. For each of the schemes below, answer the following: Do they achieve the goal? What is the minimum time for the last process to finish its task?

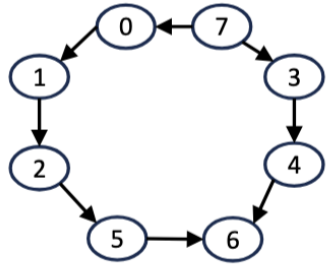
A:



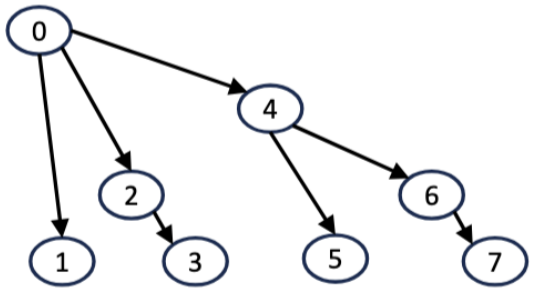
B:



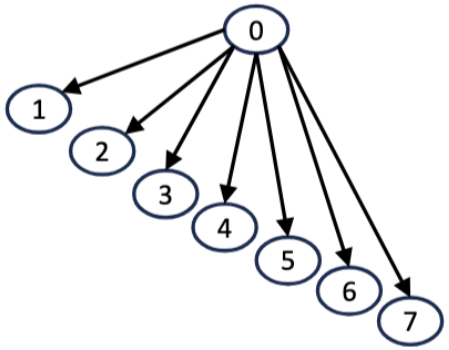
C:



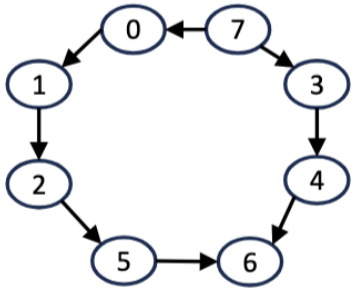
A:



B:



C:



Benjamin Scherling Catalan @bscherling · 10 months ago · edited 10 months ago

7

A: Yes, message passes through all processes. Minimum time = 3ms

B: Yes, all processes recieve message directly from 0. Minimum time = 7ms

C: No, since we start in process 0 we see that process 7 has no in-going edge and therefore can't recieve any message

Advice / Outline for exam preparation

- **Revise topics to get an overview over everything**
- **Practice some „easy points“ tasks until you are comfortable with them (always the same pattern)**
  - Amdahl, Gustafson, Pipelining, Histories, State Diagrams, Fork-Join, etc.
- **Practice harder tasks (some „creativity“ needed)**
  - Wait / notify, questions about properties of locks, barriers, code snippets, etc.
- **Practice „Mixer“ tasks**
  - Very unpredictable
  - Revise theory; try to understand and make connections between different concepts

- **Practice some old exams without a timer first**
- **Identify tasks where you struggle → specifically practice those tasks and skip easy tasks**
- **Try to solve under time contraints when you feel comfortable with the tasks and theory**
- **Try to conclude how much time to invest for each task based on the assigned points**

Good theory overview

- https://luck-wildflower-a5b.notion.site/pprog-summary-leon-thomm
- PVW script
- Summaries on ComSol (https://exams.vis.ethz.ch/)
- The Art of Multithreading book

Viel Glück und Erfolg für die Prüfung!!