

**LECTURE 16**  
**Nondeterministic Parallel  
Programming**

**Charles E. Leiserson**  
**November 8, 2022**



# Determinism

**Definition.** A program is **deterministic** on a given input if every memory location is updated with the same sequence of values in every execution.

- The program always behaves the same way.
- Two different memory locations may be updated in different orders, but each location always sees the same sequence of updates.

**Advantage:** DEBUGGING!

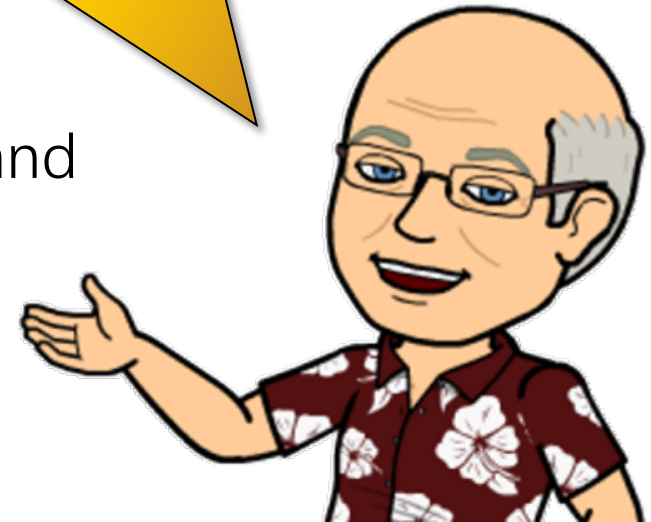
A Cilk program with no determinacy races is deterministic.

- Cilksan can help you avoid nondeterminacy bugs.

# Golden Rule of Parallel Programming

**Never** write nondeterministic  
parallel programs.

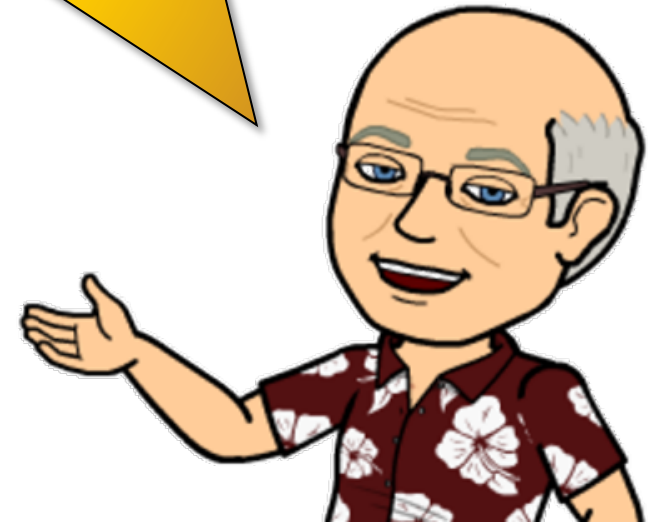
They can exhibit anomalous behaviors, and  
it's hard to debug them.



# Golden Rule of Parallel Programming

N

But a nondeterministic program  
may give me more  
**performance!**



**DANGER**



**ENTER AT YOUR OWN RISK**

# Silver Rule of Parallel Programming

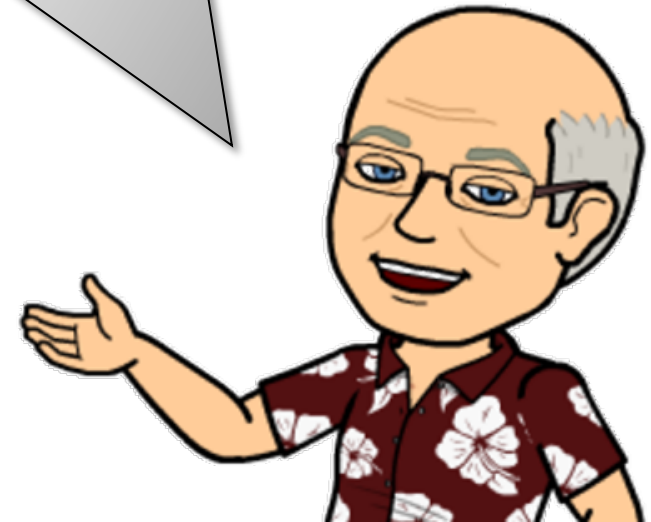
**Never** write nondeterministic  
parallel programs.  
— **but if you must\*** —  
always devise a test strategy  
to manage the nondeterminism!

## Typical test strategies

- Turn off nondeterminism.
- Encapsulate nondeterminism.
- Substitute a deterministic alternative.
- Use analysis tools.

---

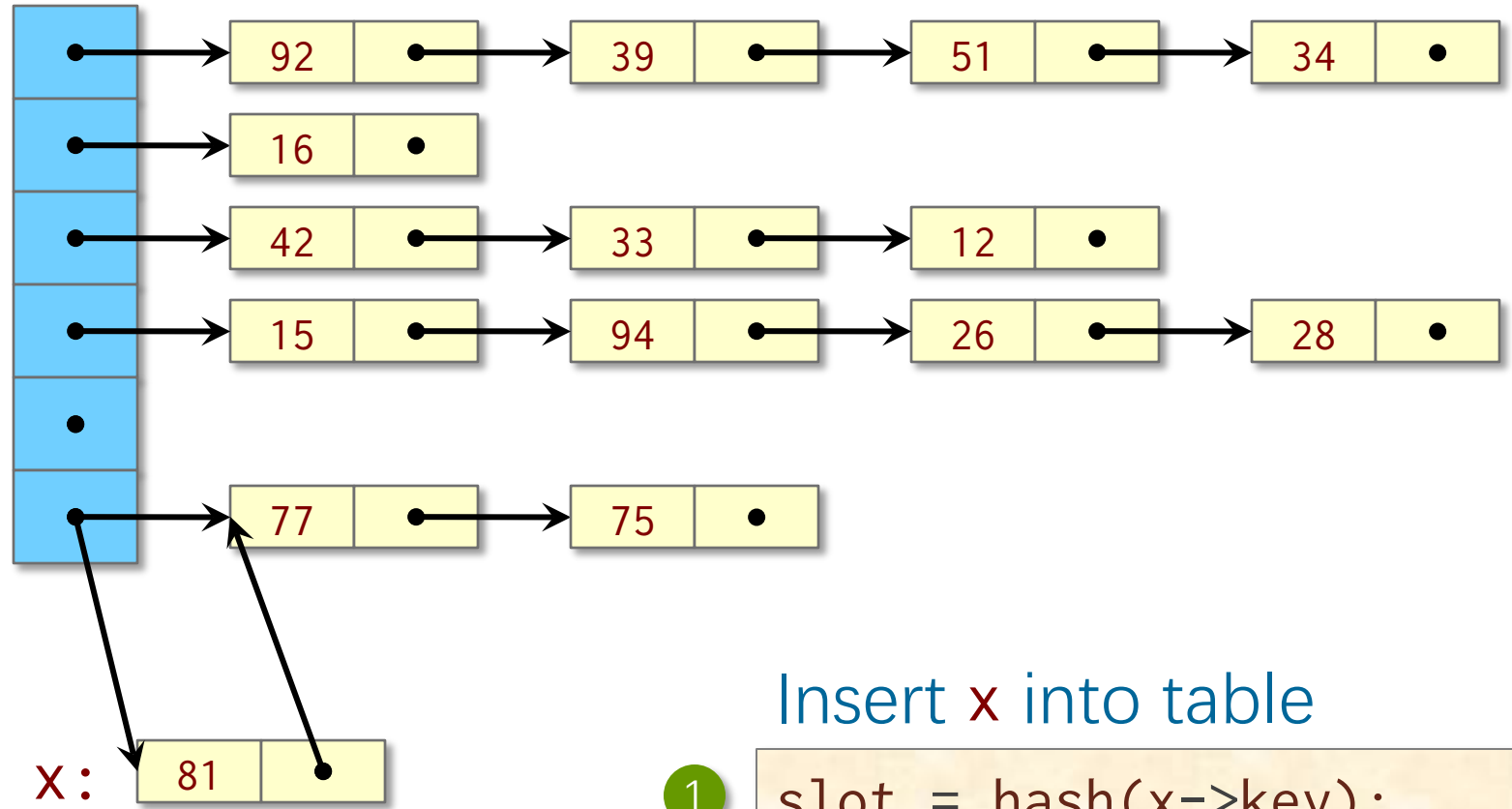
\*E.g., for performance.



# ATOMICITY & MUTUAL EXCLUSION



# Example: Hash Table

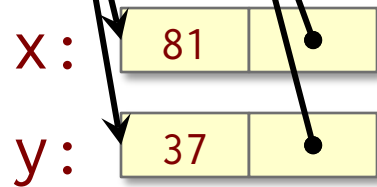
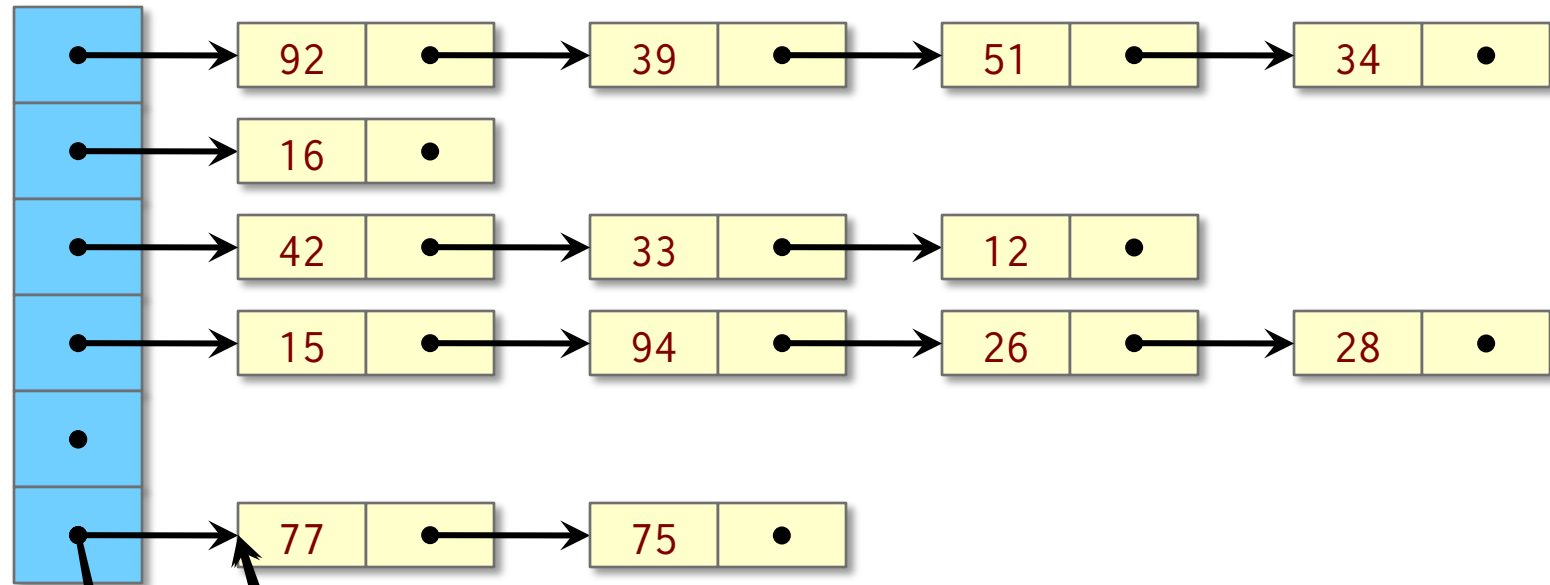


Insert  $x$  into table

```
1 slot = hash(x->key);  
2 x->next = table[slot];  
3 table[slot] = x;
```



# Concurrent Hash Table



**DATA RACE!**

```
1 slot = hash(x->key);  
2 x->next = table[slot];  
6 table[slot] = x;
```

```
3 slot = hash(y->key);  
4 y->next = table[slot];  
5 table[slot] = y;
```

# Atomicity and Mutexes

**Definition.** A sequence of instructions is *atomic* if the rest of the system never views them as partially executed. At any moment, either no instructions in the sequence have executed or all of them have executed.

**Definition.** A *critical section* is a piece of code that accesses a shared data structure which must not be accessed by two or more strands at the same time (*mutual exclusion*).

**Definition.** A *mutex* is an object with `lock()` and `unlock()` functions. An attempt by a strand to lock an already locked mutex causes that strand to *block* (*i.e.*, wait) until the mutex is unlocked.

# Concurrent Hash Table

## Modified hash-table code

- Introduce a mutex **L**.
- Lock **L** before executing the critical section.
- Unlock **L** after executing the critical section.

*DSIUDBM*  
*TFDUPO*

```
slot = hash(x->key);  
lock(&L);  
x->next = table[slot];  
table[slot] = x;  
unlock(&L);
```

## Performance problem

Only one strand can insert into the hash table at a time.

# Concurrent Hash Table II

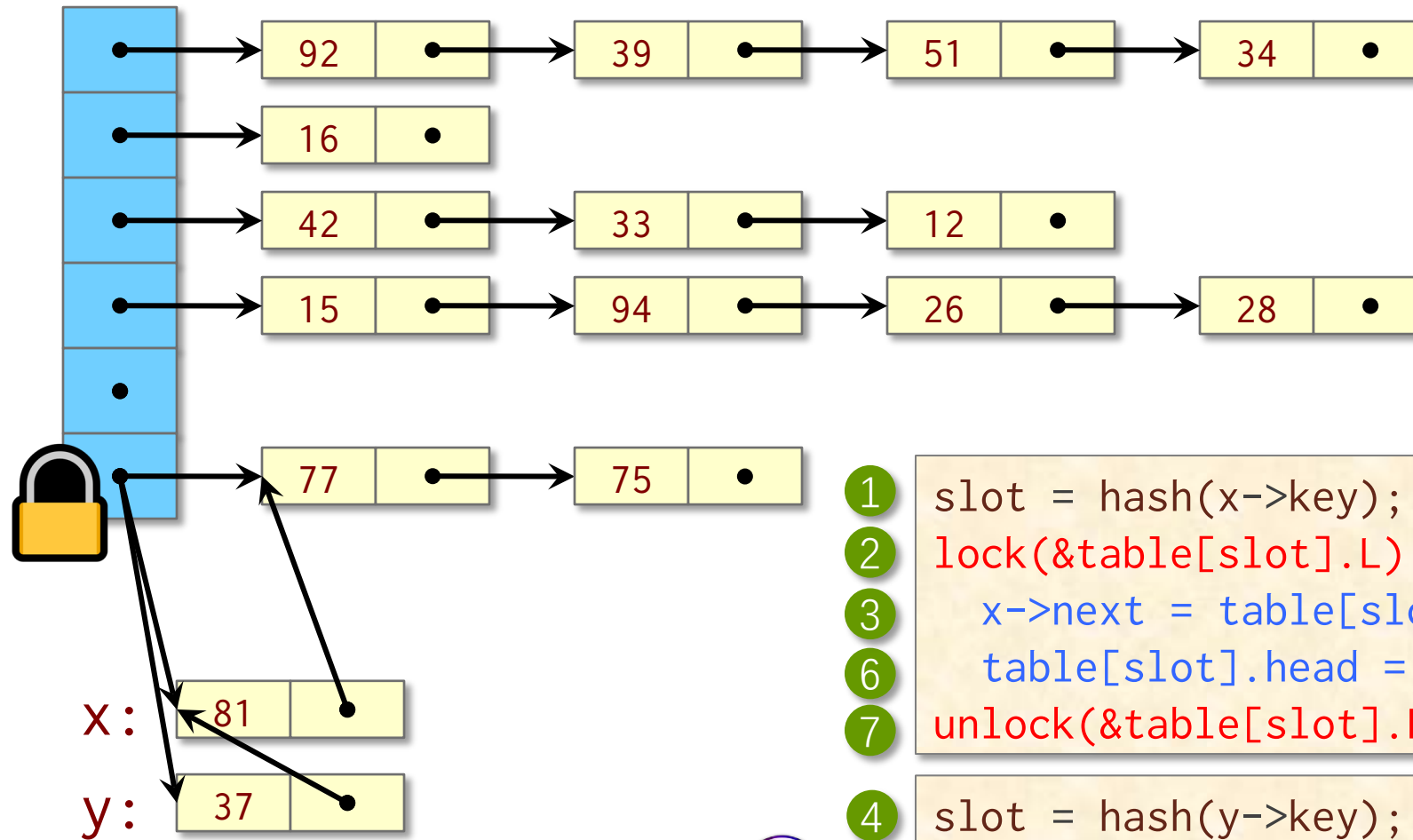
**Idea:** One mutex per slot

- Make each slot a **struct** with a mutex **L** and a pointer **head** to the slot contents.

*critical  
section*

```
slot = hash(x->key);  
lock(&table[slot].L);  
    x->next = table[slot].head;  
    table[slot].head = x;  
unlock(&table[slot].L);
```

# Concurrent Hash Table with Mutexes



Q: Is this table deterministic?

**NO!**

```
1 slot = hash(x->key);  
2 lock(&table[slot].L);  
3   x->next = table[slot].head;  
6   table[slot].head = x;  
7 unlock(&table[slot].L);
```

```
4 slot = hash(y->key);  
5 lock(&table[slot].L);  
8   y->next = table[slot].head;  
9   table[slot].head = y;  
10 unlock(&table[slot].L);
```

# Recall: Determinacy Races

**Definition.** A *determinacy race* occurs when two logically parallel instructions access the same memory location and at least one of the instructions performs a write.

- A program execution with no determinacy races means that the program is deterministic on that input.
- The program always behaves the same on that input, no matter how it is scheduled and executed.
- If a determinacy race exists in an ostensibly deterministic program (e.g., a program with no mutexes), Cilksan guarantees to find such a race.

# Data Races

**Definition.** A *EBUB SDF* occurs when two logically parallel strands *holding no locks in common* access the same memory location and at least one of the strands performs a write.

Although data-race-free programs obey atomicity constraints, they can still be nondeterministic, because *acquiring a lock* can cause a determinacy race with *another lock acquisition*.



**WARNING:** Codes that use locks are nondeterministic by intention, and they invalidate Cilksan's guarantee.

# No Data Races $\neq$ No Bugs

## Example

```
slot = hash(x->key);  
  
lock(&table[slot].L);  
    x->next = table[slot].head;  
unlock(&table[slot].L);  
  
lock(&table[slot].L);  
    table[slot].head = x;  
unlock(&table[slot].L);
```

Nevertheless, the presence of mutexes and the absence of data races at least means that the programmer thought about the issue.



# “Benign” Races

**Example:** Identify the set of digits in an array.

A: 4, 1, 0, 4, 3, 3, 4, 6, 1, 9, 1, 9, 6, 6, 6, 3, 4

```
for (int i = 0; i < 10, ++i) {  
    digits[i] = 0;  
}  
cilk_for (int i = 0; i < N; ++i) {  
    digits[A[i]] = 1; // benign race  
}
```

digits:

1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9



**CAUTION:** This code only works correctly if the hardware writes the array elements atomically (e.g., it may race on byte values for some architectures).

# “Benign” Races

**Example:** Identify the set of digits in an array.

A: 4, 1, 0, 4, 3, 3, 4, 6, 1, 9, 1, 9, 6, 6, 6, 3, 4

```
for (int i = 0; i < 10, ++i) {  
    digits[i] = 0;  
}  
cilk_for (int i = 0; i < N; ++i) {  
    digits[A[i]] = 1; // benign race  
}
```

digits:

1	1	0	1	1	0	1	0	0	1
0	1	2	3	4	5	6	7	8	9

Cilksan allows you to turn off race detection for intentional races, which is dangerous but practical. Better solutions exist, e.g., *fake locks* in Intel’s Cilkscreen (see Intel Cilk Plus Tools User's Guide).

# IMPLEMENTATION OF MUTEXES



# Properties of Mutexes

- **Yielding/spinning**

A yielding mutex returns control to the operating system when it blocks. A spinning mutex consumes processor cycles while blocked.

- **Reentrant/nonreentrant**

A reentrant mutex allows a thread that is already holding a lock to acquire it again. A nonreentrant mutex deadlocks if the thread attempts to reacquire a mutex it already holds.

- **Fair/unfair**

A fair mutex puts blocked threads on a FIFO queue, and the unlock operation unblocks the thread that has been waiting the longest. An unfair mutex lets any blocked thread go next.

# Simple Spinning Mutex

Spin\_Mutex:

```
    cmp 0, mutex ; Check if *mutex is free  
    je Get_Mutex  
    pause ; x86 hack to unconfuse pipeline  
    jmp Spin_Mutex
```

Get\_Mutex:

```
    mov 1, %eax  
    xchg mutex, %eax ; Try to get mutex  
    cmp 0, %eax ; Test if successful  
    jne Spin_Mutex
```

Critical\_Section:

```
    <critical-section code>  
    mov 0, mutex ; Release mutex
```

**Key property:** xchg is an atomic exchange.

# Simple Yielding Mutex

Spin\_Mutex:

```
    cmp 0, mutex ; Check if *mutex is free  
    je Get_Mutex  
    call pthread_yield ; Yield quantum  
    jmp Spin_Mutex
```

Get\_Mutex:

```
    mov 1, %eax  
    xchg mutex, %eax ; Try to get mutex  
    cmp 0, %eax ; Test if successful  
    jne Spin_Mutex
```

Critical\_Section:

```
    <critical-section code>  
    mov 0, mutex ; Release mutex
```

# Competitive Mutex

## Competing goals:

- To claim mutex soon after it is released.
- To behave nicely and waste few cycles.

**IDEA:** Spin for a while, and then yield.

## How long to spin?

As long as a context switch takes. Then, you never wait longer than twice the optimal time.

- If the mutex is released while spinning, optimal.
- If the mutex is released after yield,  $\leq 2 \times$  optimal.

## Randomized algorithm [KMMO94]

A clever randomized algorithm can achieve a competitive ratio of  $e/(e-1) \approx 1.58$ .

# LOCKING ANOMALY: DEADLOCK





# Deadlock

Holding more than one lock at a time can be dangerous:

## Thread 1

```
1 lock(&A);  
2 lock(&B);  
   <critical section>  
unlock(&B);  
unlock(&A);
```

## Thread 2

```
2 lock(&B);  
2 lock(&A);  
   <critical section>  
unlock(&A);  
unlock(&B);
```

The ultimate loss of performance!

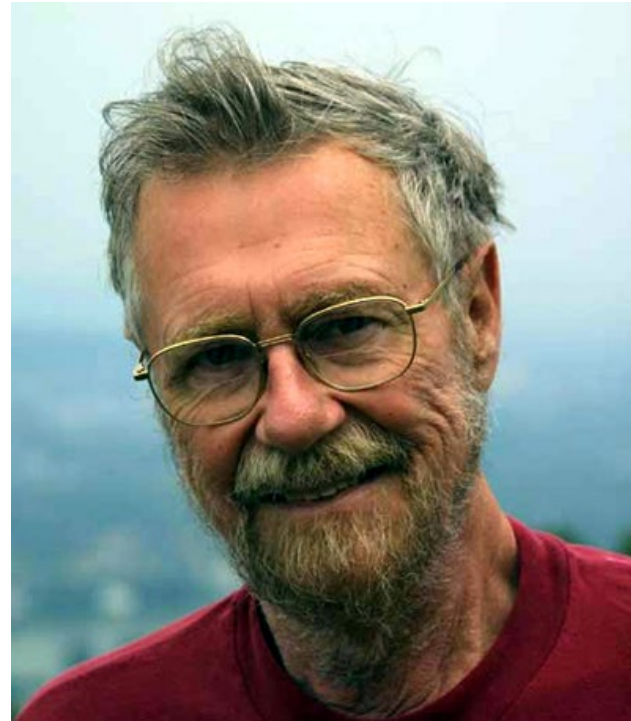
# Conditions for Deadlock

1. **Mutual exclusion** — Each thread claims exclusive control over the resources it holds.
2. **Nonpreemption** — Each thread does not release the resources it holds until it completes its use of them.
3. **Circular waiting** — A cycle of threads exists in which each thread is blocked waiting for resources held by the next thread in the cycle.

# Dining Philosophers



*C.A.R. (Tony) Hoare*



*Edsger Dijkstra*

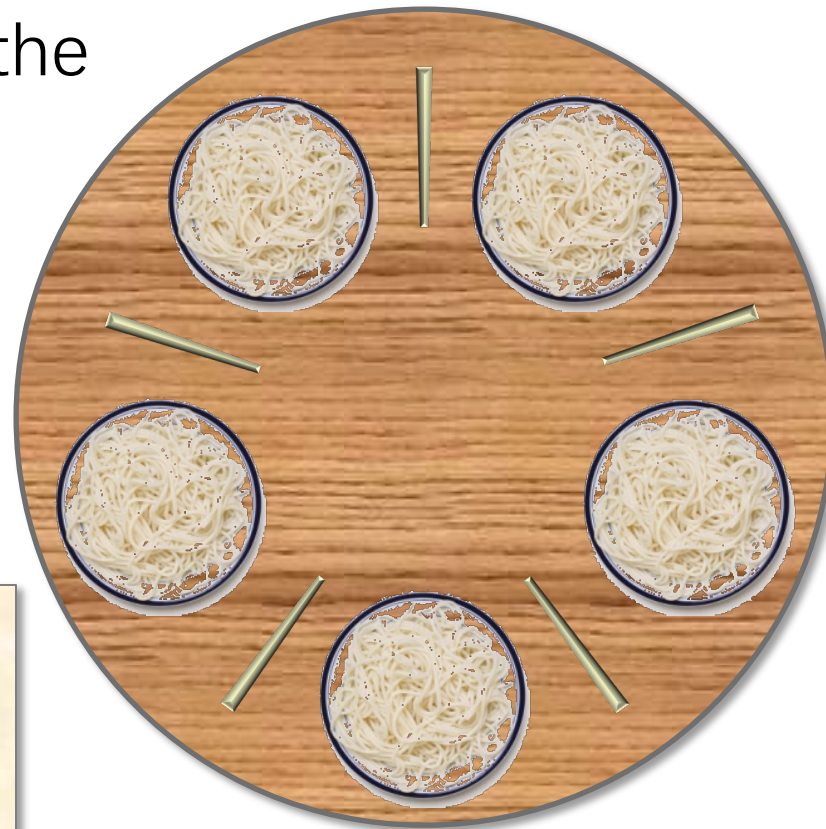
Illustrative story of deadlock told by Charles Antony Richard Hoare based on an examination question by Edsger Dijkstra. The story has been embellished over the years by many retellers.

# Dining Philosophers

Each of  $n$  philosophers needs the two chopsticks on either side of their plate to eat their noodles.

## Philosopher $i$

```
while (1) {  
    think();  
    lock(&chopstick[i].L);  
    lock(&chopstick[(i+1)%n].L);  
    eat();  
    unlock(&chopstick[i].L);  
    unlock(&chopstick[(i+1)%n].L);  
}
```



# ~~Dining~~ Philosophers Starving

Each of  $n$  philosophers needs the two chopsticks on either side of their plate to eat their noodles.



One day they all pick up their left chopsticks simultaneously.

## Philosophers

```
while (1) {  
    think();  
    lock(&chopstick[i].L);  
    lock(&chopstick[(i+1)%n].L);  
    eat();  
    unlock(&chopstick[i].L);  
    unlock(&chopstick[(i+1)%n].L);  
}
```

# Preventing Deadlock

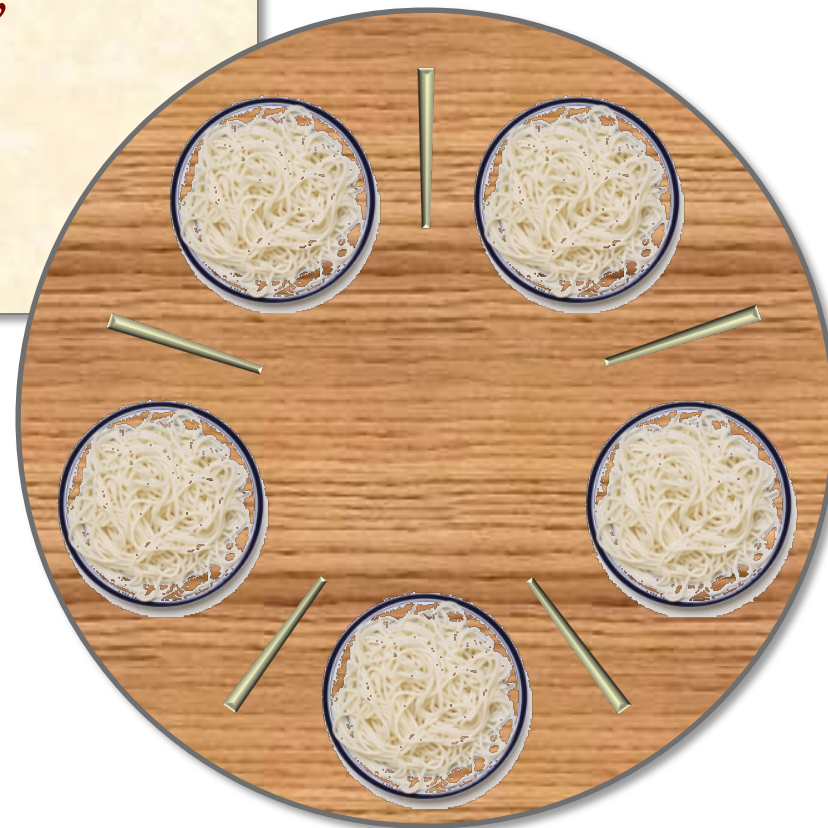
**Theorem.** Assume that we can linearly order the mutexes  $L_1 < L_2 < \dots < L_n$  so that whenever a thread holds a mutex  $L_i$  and attempts to lock another mutex  $L_j$ , we have  $L_i < L_j$ . Then, no deadlock can occur.

*1SPPG* Suppose that a cycle of waiting exists. Consider the thread in the cycle that holds the “largest” mutex  $L_{\max}$  in the ordering, and suppose that it is waiting on a mutex  $L$  held by the next thread in the cycle. Then, we must have  $L_{\max} < L$ . Contradiction. ■

# Dining Philosophers

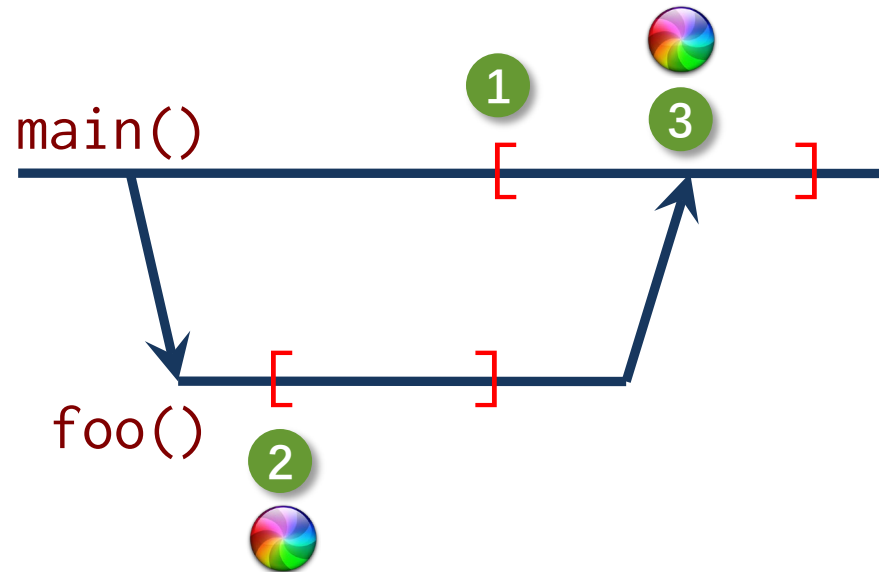
## Philosopher $i$

```
while (1) {  
    think();  
    lock(&chopstick[min(i, (i+1)%n)].L);  
    lock(&chopstick[max(i, (i+1)%n)].L);  
    eat();  
    unlock(&chopstick[i].L);  
    unlock(&chopstick[(i+1)%n].L);  
}
```



# Deadlocking Cilk with just one lock

```
void main() {  
  cilk_scope {  
    cilk_spawn foo();  
    lock(&L); ①  
  } ③  
  unlock(&L);  
}  
  
void foo() {  
  lock(&L); ②  
  unlock(&L);  
}
```



- Don't hold mutexes across joins!
- Hold mutexes only within `cilk_scope`'s.
- As always, try to avoid nondeterministic programming (but that's not always possible).



# LOCKING ANOMALY: CONVOYING



# Convoying

A lock *convoy* occurs when multiple threads of equal priority contend repeatedly for the same lock.

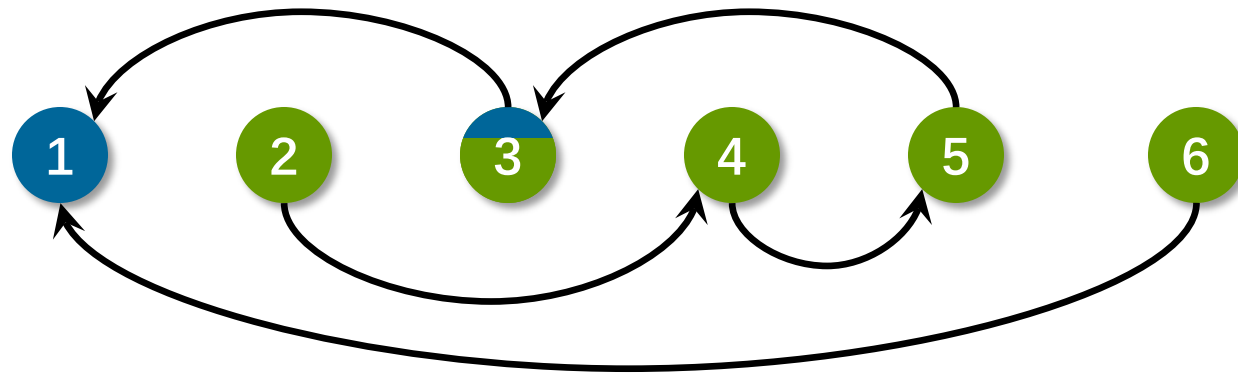
## Example: Performance bug in MIT-Cilk

When random work-stealing, each thief grabs a mutex on its victim's deque:

- If the victim's deque is empty, the thief releases the mutex and tries again at random.
- If the victim's deque contains work, the thief steals the topmost frame and then releases the mutex.

**PROBLEM:** At start-up, most thieves quickly converge on the worker containing the initial strand, creating a *convoy*.

# Performance Bug in MIT-Cilk



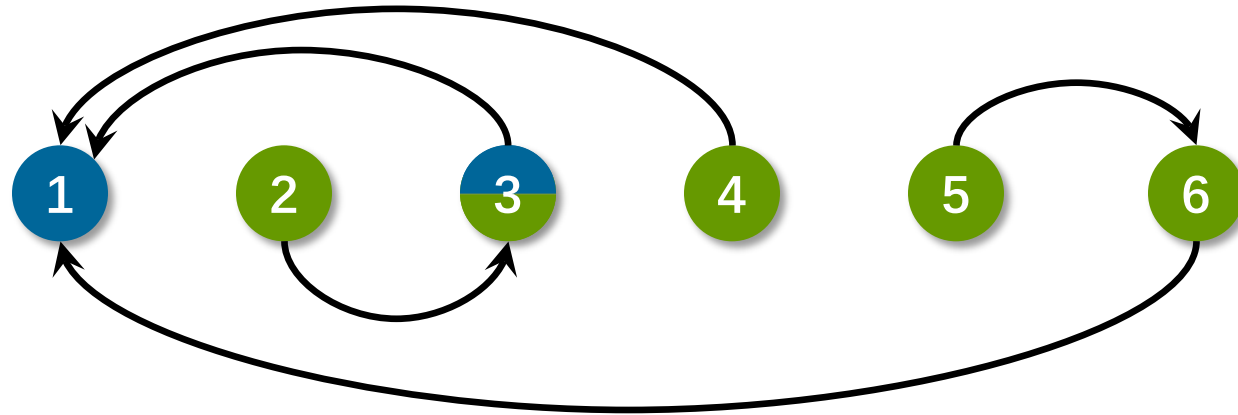
● : busy worker

●<sup>3</sup> : successful steal in progress

● : idle worker

● → ● : dependency from ● onto the lock on 's deque

# Performance Bug in MIT-Cilk



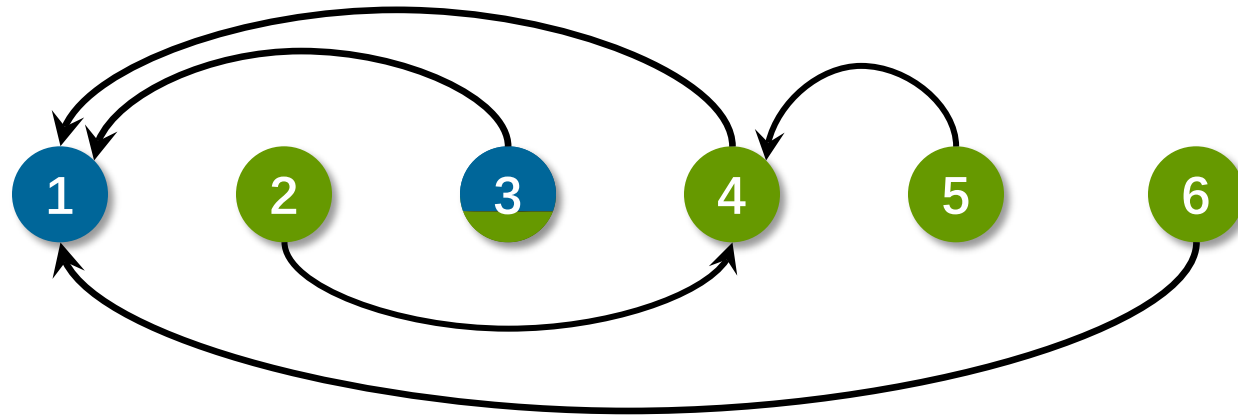
● : busy worker

●<sup>3</sup> : successful steal in progress

● : idle worker

● → ● : dependency from onto the lock on 's deque

# Performance Bug in MIT-Cilk



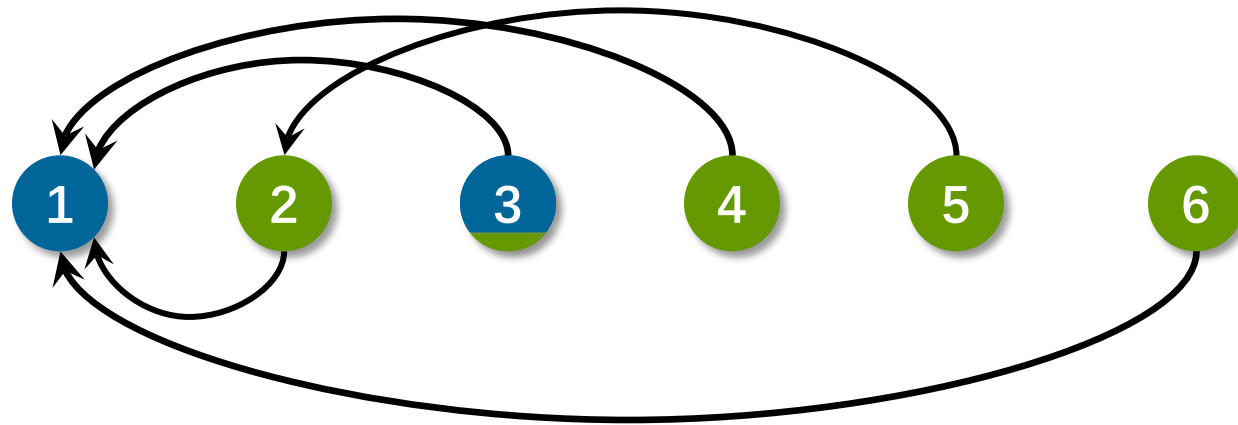
● : busy worker

●<sup>3</sup> : successful steal in progress

● : idle worker

● → ● : dependency from ● onto the lock on 's deque

# Performance Bug in MIT-Cilk



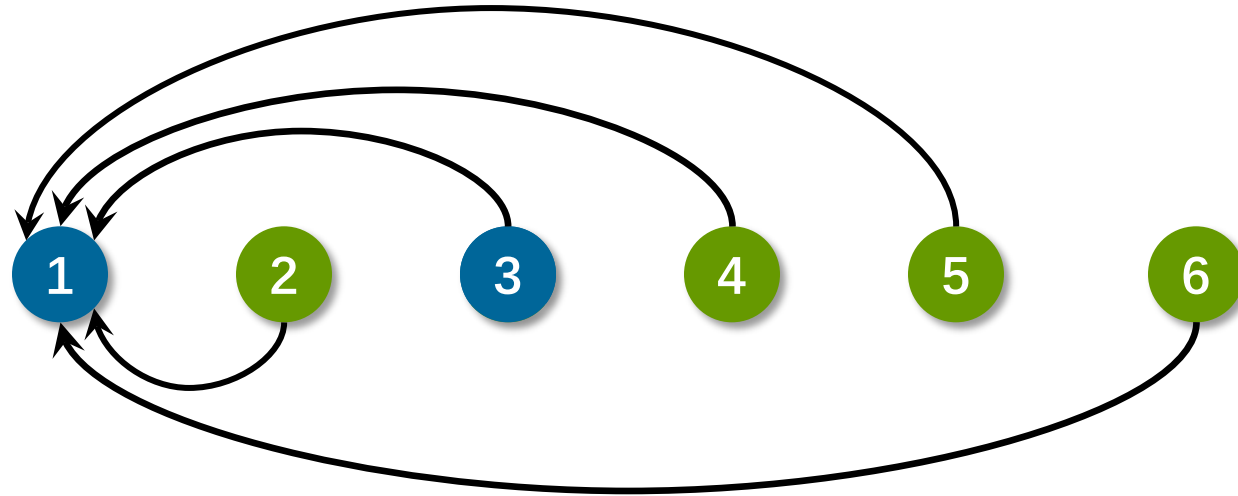
 : busy worker

 : idle worker

 : successful steal in progress

  : dependency from  onto the lock on  's deque

# Performance Bug in MIT-Cilk



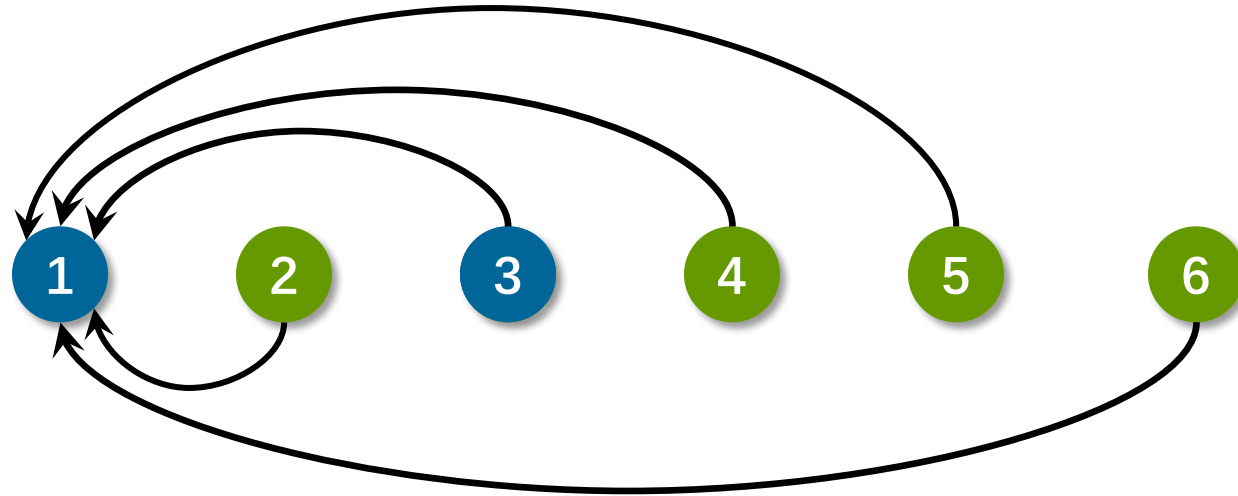
● : busy worker

● : idle worker

●<sup>3</sup> : successful steal in progress

● → ● : dependency from ● onto the lock on 's deque

# Performance Bug in MIT-Cilk



The work now gets distributed slowly as each thief serially obtains Processor 1's mutex.



# Solving the Convoying Problem

Use the nonblocking function `try_lock()`, rather than `lock()`:

- `try_lock()` attempts to acquire the mutex and returns a flag indicating whether it was successful, but it does not block on an unsuccessful attempt.

In Cilk Plus, when a thief fails to acquire a mutex, it simply tries to steal again at random, rather than blocking.

# LOCKING ANOMALY: CONTENTION



# Summing Example

```
int compute(const el_t *v);
const size_t n = 1000000;
extern el_t myArray[n];

int main() {
    int result = 0;
    for (size_t i = 0; i < n; ++i) {
        result += compute(&myArray[i]);
    }
    printf("The result is: %d\n", result);

    return 0;
}
```

# Summing Example in Cilk

```
int compute(const el_t *v);
const size_t n = 1000000;
extern el_t myArray[n];

int main() {
    int result = 0;
    cilk_for (size_t i = 0; i < n; ++i) {
        result += compute(&myArray[i]);
    }
    printf("The result is: %d\n", result);

    return 0;
}
```

Assume  $\Theta(1)$   
work.

Race!

## Work/span theory

$$T_1(n) = \Theta(n)$$

$$T_\infty(n) = \Theta(\lg n)$$

$$T_p(n) = O(n/P + \lg n)$$

# Mutex Solution

```
#include <pthread.h>
int compute(const el_t *v);
const size_t n = 1000000;
extern el_t myArray[n];

int main() {
    int result = 0;
    pthread_spinlock_t slock;
    pthread_spin_init(&slock, 0);
    cilk_for (size_t i = 0; i < n; ++i) {
        pthread_spin_lock(&slock);
        result += compute(&myArray[i]);
        pthread_spin_unlock(&slock);
    }
    printf("The result is: %d\n", result);

    return 0;
}
```

*Lock contention*  
⇒ no parallelism!

## Contention

$$T_1(n) = \Theta(n)$$

$$T_\infty(n) = \Theta(\lg n)$$

$$T_P(n) = \Omega(n)$$

# Scheduling with Mutexes

Greedy scheduler:

$$T_P \leq T_1/P + T_\infty + B ,$$

where  $B$  is the **bondage** — the total time of all critical sections.

---

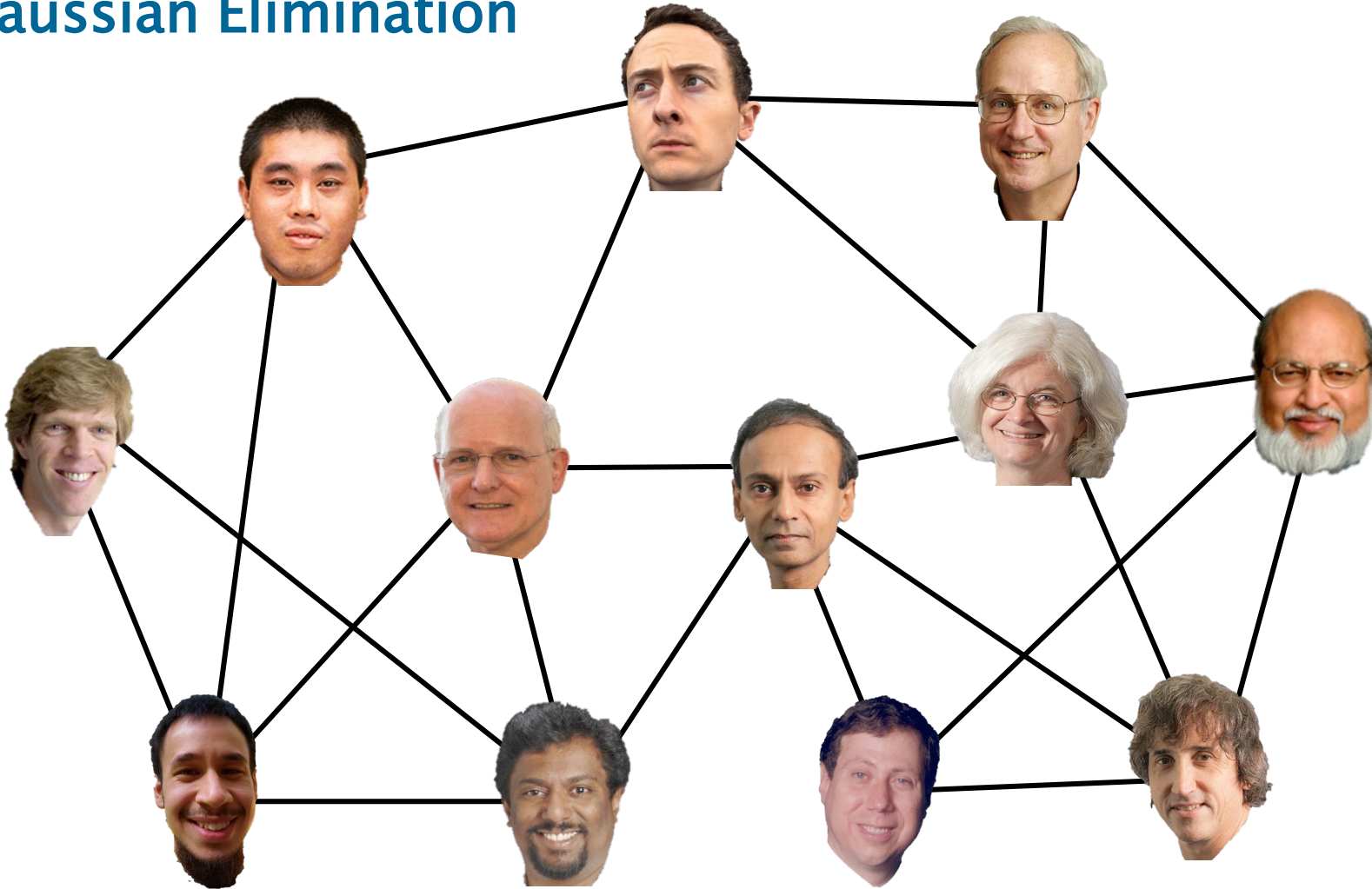
This upper bound is weak, especially if many small mutexes each protect different critical regions. Little is known theoretically about lock contention.

# TRANSACTIONAL MEMORY



# Concurrent Graph Computation

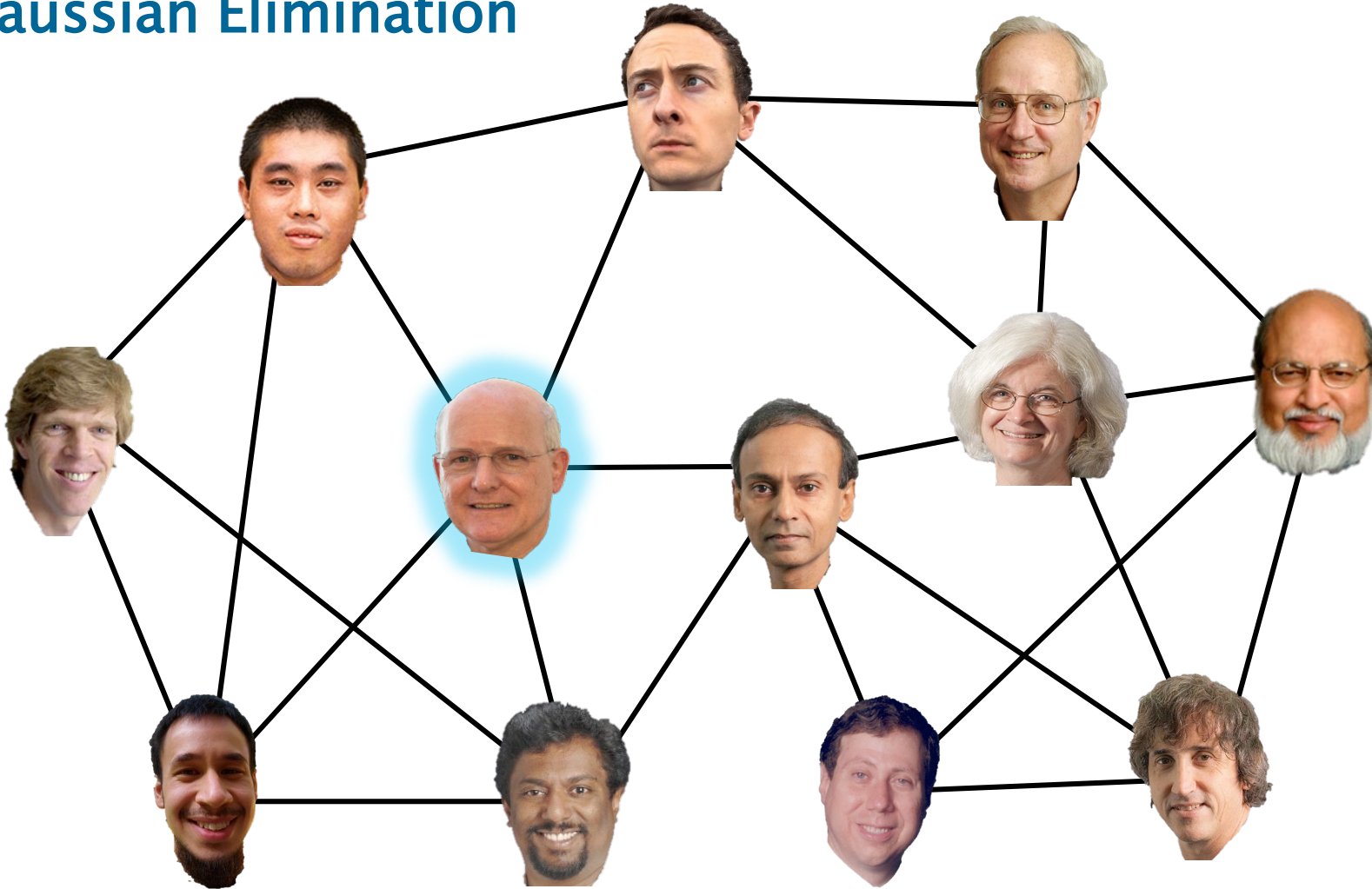
## Gaussian Elimination





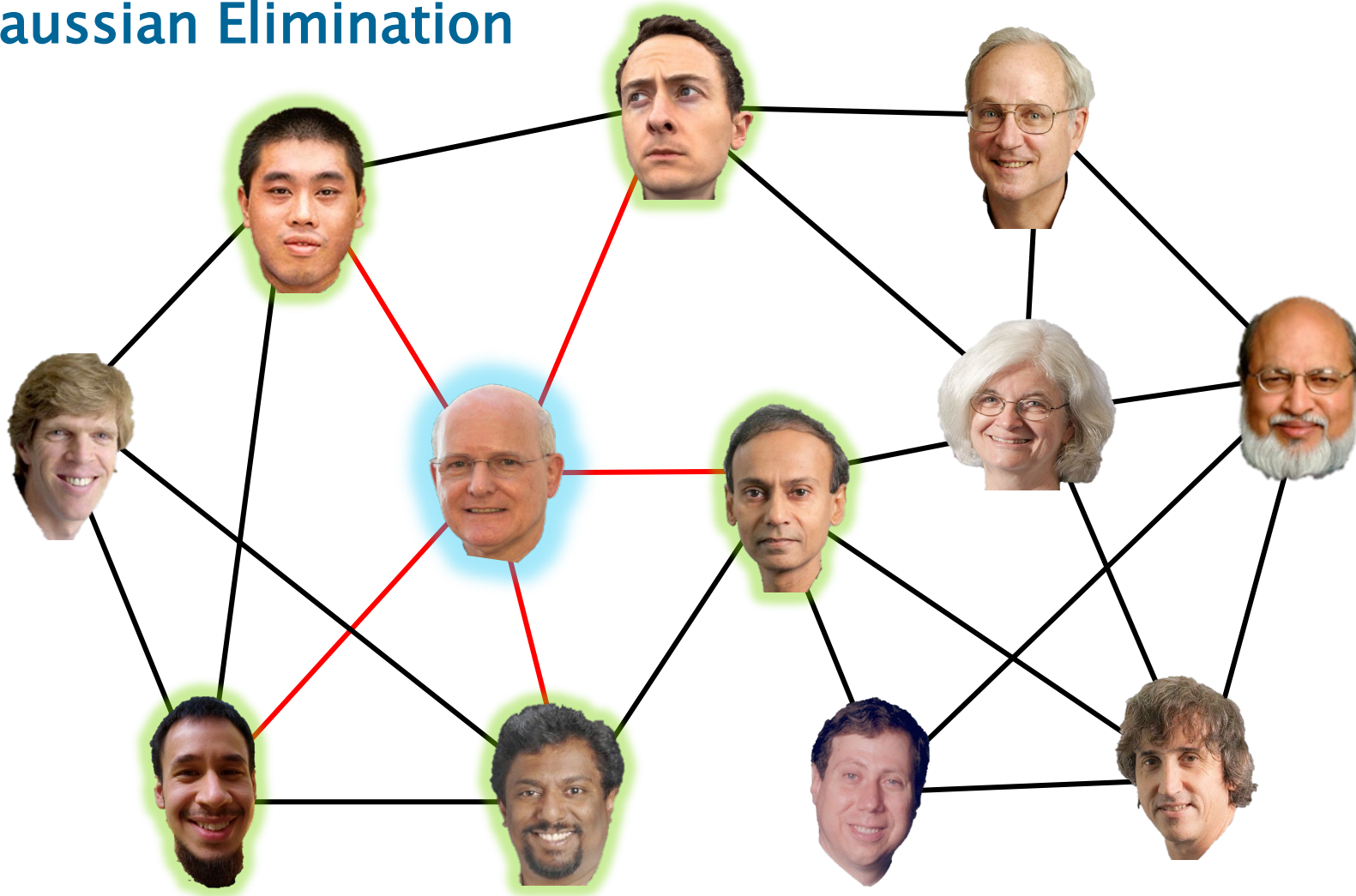
# Concurrent Graph Computation

## Gaussian Elimination



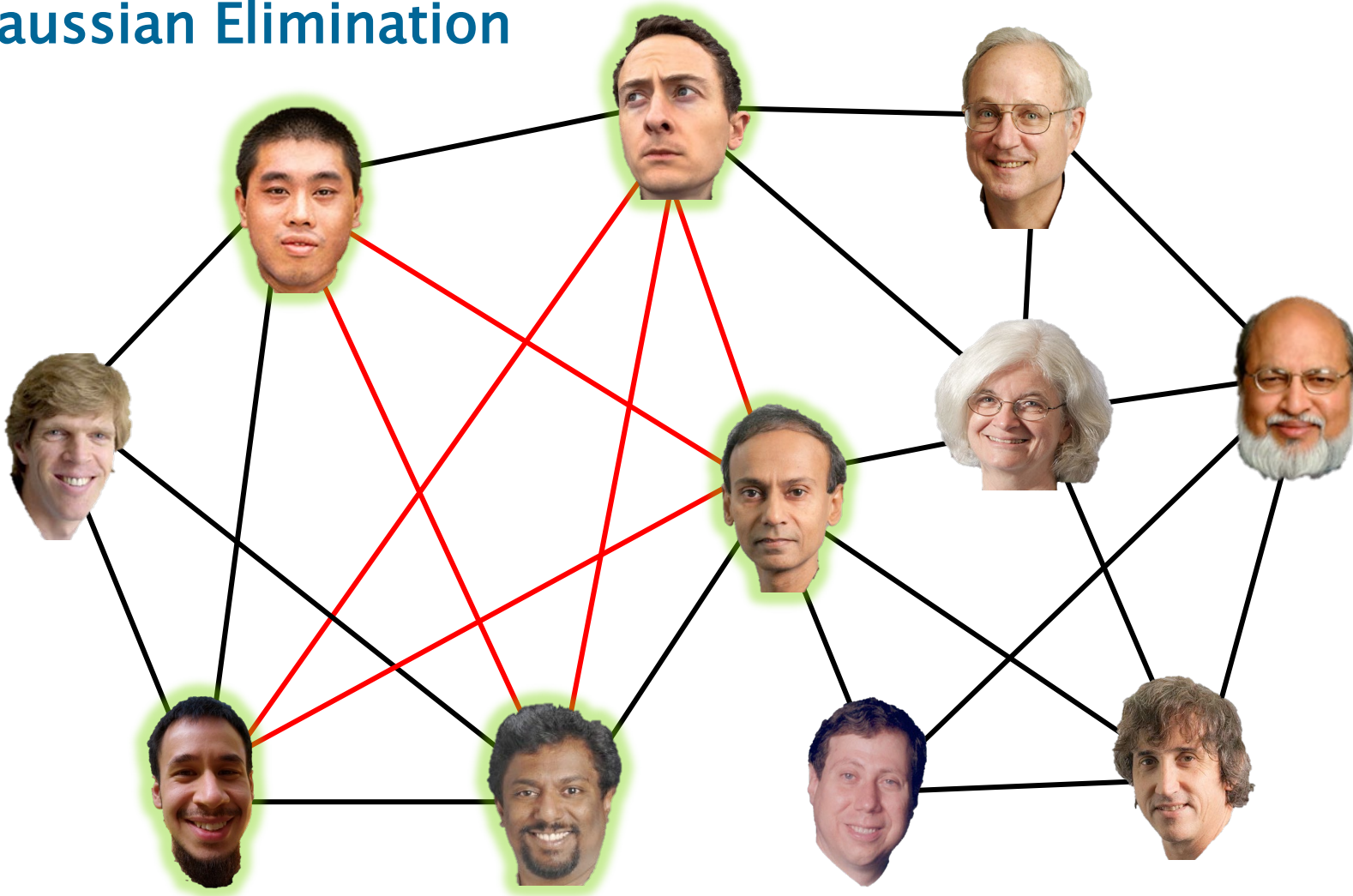
# Concurrent Graph Computation

## Gaussian Elimination



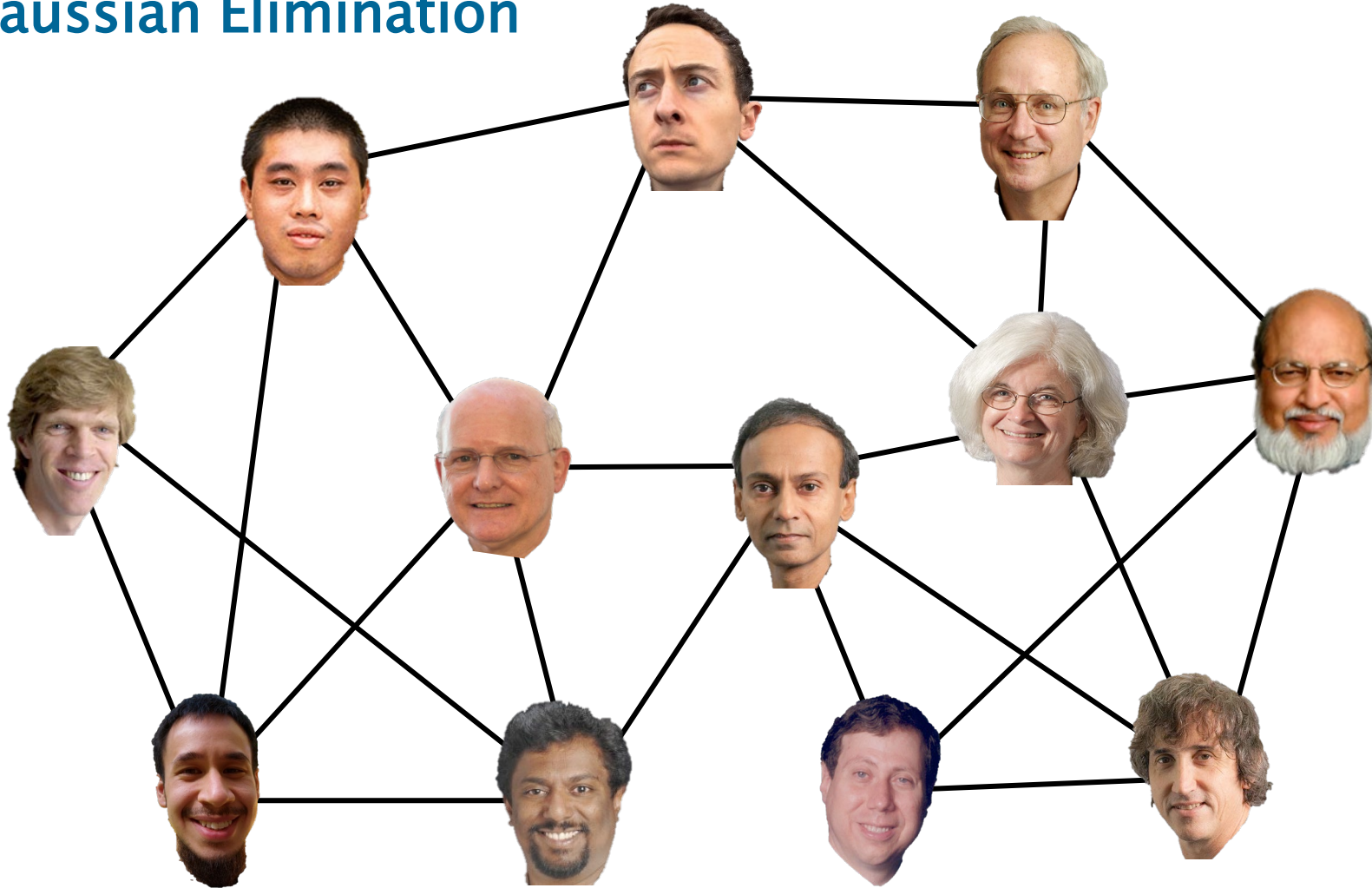
# Concurrent Graph Computation

## Gaussian Elimination



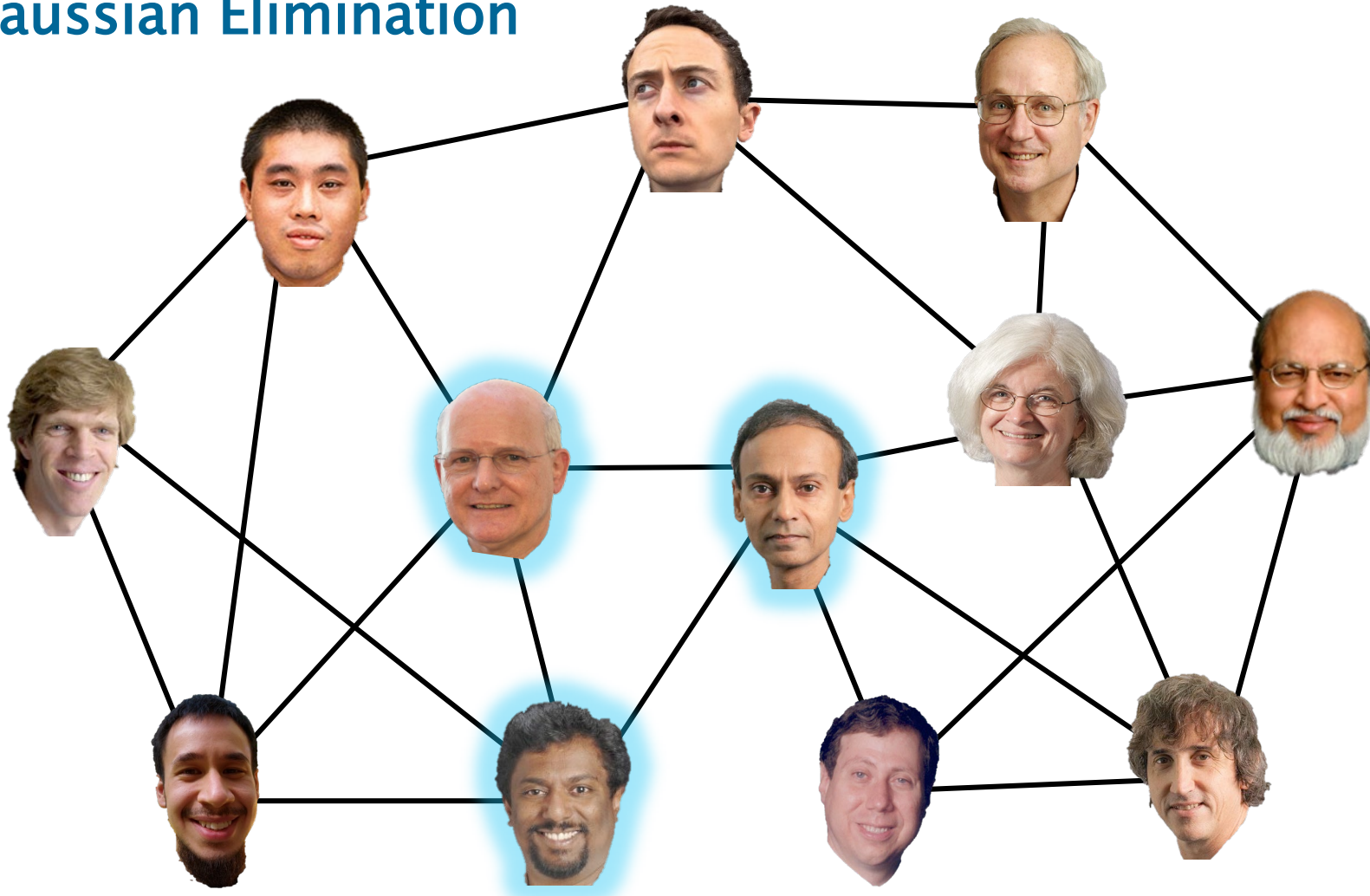
# How to Deal with Concurrency?

## Gaussian Elimination



# How to Deal with Concurrency?

## Gaussian Elimination



# Transactional Memory\*

```
Gaussian_Eliminate(G, v) {  
  atomic {  
    S = neighbors[v];  
    for u ∈ S {  
      E(G) = E(G) - {(u, v)};  
      E(G) = E(G) - {(v, u)};  
    }  
    V(G) = V(G) - {v};  
    for u ∈ S  
      for u' ∈ S - {u}  
        E(G) = E(G) ∪ {(u, u')};  
  }  
}
```

## Atomicity

- On transaction **commit**, all memory updates in the critical region appear to take effect at once.
- On transaction **abort**, none of the memory updates appear to take effect, and the trans-action must be **restarted**.
- A restarted transaction may take a different code path.

# Definitions

## Conflict

When two or more transactions attempt to access the same location of transactional memory concurrently.

## Contention resolution

Deciding which of two conflicting transactions to wait or to abort and restart, and under what conditions.

## Forward progress

Avoiding deadlock, **livelock**, and **starvation**.

## Throughput

Run as many transactions concurrently as possible.

# Algorithm L [L16]

Algorithm L is a remarkably simple algorithm for guaranteeing the forward progress of transactions.

Assume that the transactional-memory system provides mechanisms for

- logging reads and writes,
- aborting and rolling back transactions,
- restarting.

Algorithm L employs a lock-based approach that combines two ideas:

- *finite ownership array* [HF03],
- *release-sort-reacquire* [L95, RFF06].



# Finite Ownership Array

- An array `lock[0..n-1]` of antistarvation (queuing) **mutual-exclusion locks**,\* which support:
  - **ACQUIRE(l)**: Grab lock `l`, blocking until it becomes available.
  - **TRY\_ACQUIRE(l)**: Try to grab lock `l`, and return `true` or `false` to indicate success or failure, respectively.
  - **RELEASE(l)**: Release lock `l`.
- An **owner function**  $h: U \rightarrow \{0, 1, \dots, n-1\}$  mapping the space `U` of memory locations to indexes in `lock`.
- To lock location  $x \in U$ , acquire `lock[h(x)]`.

---

\*For greater generality, one can use **reader/writer locks**.

# Release-Sort-Reacquire

Before accessing a memory location  $x$ , try to acquire  $\text{lock}[h(x)]$  greedily. On conflict (i.e., the lock is already held):

1. Roll back the transaction (without releasing locks).
2. Release all locks with indexes larger than  $h[x]$ .
3. Acquire  $\text{lock}[h(x)]$ , blocking if already held.
4. Reacquire the released locks in sorted order, blocking if already held.
5. Restart the transaction.

# Algorithm L

```
SAFE_ACCESS(x, L)
1  if  $h(x) \notin L$ 
2       $M = \{i \in L : i > h(x)\}$ 
3       $L = L \cup \{h(x)\}$ 
4      if  $M == \emptyset$ 
5          ACQUIRE(lock[h(x)]) // blocking
6      elseif TRY_ACQUIRE(lock[h(x)]) // nonblocking
7          // do nothing
8      else
9          roll back transaction state (without releasing locks)
10         for  $i \in M$ 
11             RELEASE(lock[i])
12         ACQUIRE(lock[h(x)]) // blocking
13         for  $i \in M$  in increasing order
14             ACQUIRE(lock[i]) // blocking
15         restart transaction // does not return
16     access location  $x$ 
```

Owner function.

Set of local lock-indexes.

Locks held with indexes larger than  $h(x)$ .

Global finite ownership array.

Safely access a memory location  $x$  within a transaction having local lock-index set  $L$ .

- At transaction start, the transaction's lock-index set  $L$  is initialized to the empty set:  $L = \emptyset$ .
- When the transaction completes, all locks with indexes in  $L$  are released.

# Forward Progress (1)

Before accessing a memory location  $x$ , try to acquire  $\text{lock}[h(x)]$  greedily. On conflict (i.e., the lock is already held):

1. Roll back the transaction (without releasing locks).
2. Release all locks with indexes larger than  $h[x]$ .
3. Acquire  $\text{lock}[h(x)]$ , blocking if already held.
4. Reacquire the released locks in sorted order, blocking if already held.
5. Restart the transaction.

## No deadlocks

A transaction only **blocks** when waiting for a lock larger than any of the locks it already holds  $\Rightarrow$  no deadly embrace, i.e., no cycle of blocking.

# Forward Progress (2)

Before accessing a memory location  $x$ , try to acquire  $\text{lock}[h(x)]$  greedily. On conflict (i.e., the lock is already held):

1. Roll back the transaction (without releasing locks).
2. Release all locks with indexes larger than  $h[x]$ .
3. Acquire  $\text{lock}[h(x)]$ , blocking if already held.
4. Reacquire the released locks in sorted order, blocking if already held.
5. Restart the transaction.

## No livelocks or starvation

Each time a transaction restarts, it holds at least one more lock than it held the previous time. Thus, a transaction can be attempted at most  $n$  times, where  $n$  is the size of the ownership array.

# Remarks

Properly choosing the length  $n$  of the ownership-array is crucial:

- The smaller  $n$  is, the more the **false contention**.
- The larger  $n$  is, the weaker the forward-progress guarantee.
- If the owner function  $h$  is random, by the **birthday paradox**, the number of “false” conflicts is at most  $1$  if  $n = m^2/2$ , where  $m$  is the total number of shared-memory locations in all concurrently running transactions.

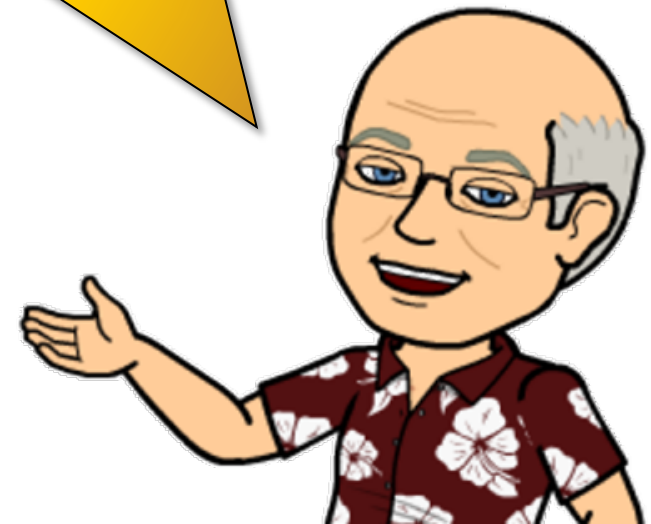
As a practical matter, **timestamp-based algorithms** seem to be the preferred method for guaranteeing forward progress:

- wound-wait and wait-die [RSL78],
- TL2 [DSS06],
- provable bounds [GHP05].

But these algorithms tend to be complex.

# Golden Rule of Parallel Programming

**Never** write nondeterministic  
parallel programs.



# Silver Rule of Parallel Programming

**Never** write nondeterministic  
parallel programs.

— but if you must\* —  
always devise a test strategy  
to manage the nondeterminism!

