Scalability + Correctness

Chris Rossbach + Calvin Lin

CS380p

Outline for Today

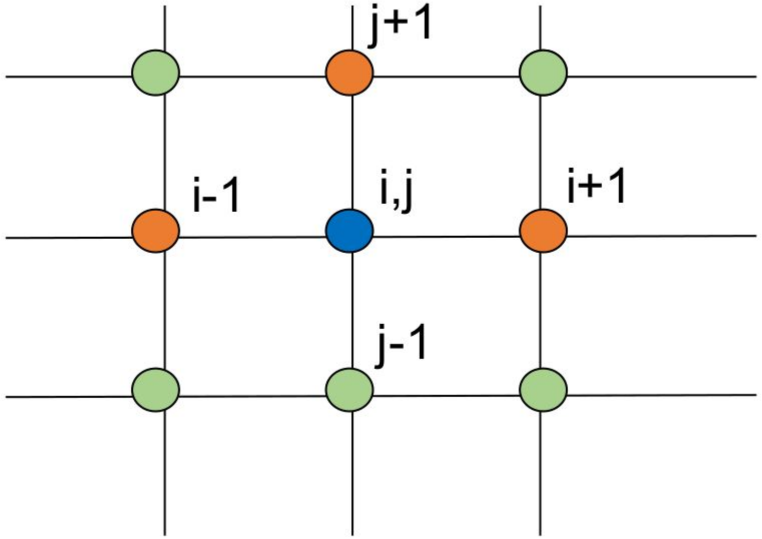
- Concurrency & Parallelism Basics
  - Decomposition redux
  - Measuring Parallel Performance
  - Performance Tradeoffs
  - Correctness and Performance

Acknowledgments: some materials in this lecture borrowed from or built on materials from:

- *Emmett Witchel, who borrowed them from: Kathryn McKinley, Ron Rockhold, Tom Anderson, John Carter, Mike Dahlin, Jim Kurose, Hank Levy, Harrick Vin, Thomas Narten, and Emery Berger*
- *Mark Silberstein, who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta*

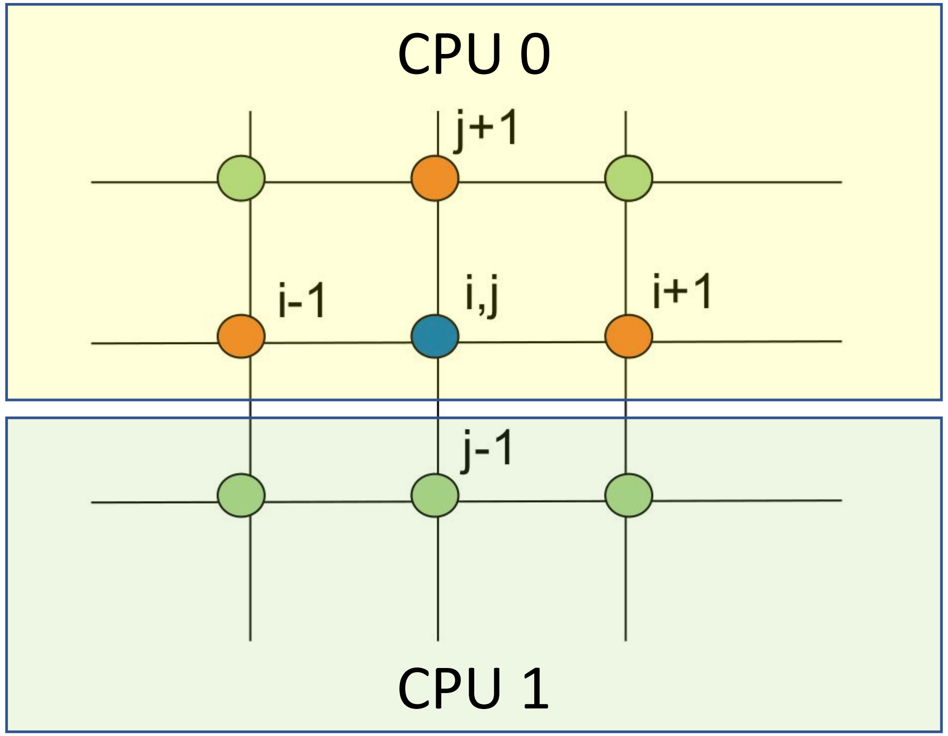
Review: Game of Life

- Given a 2D Grid:
- $v_t(i,j) = F\big(v_{t-1}(of\ all\ its\ neighbors)\big)$



Domain decomposition

Each CPU gets part of the input



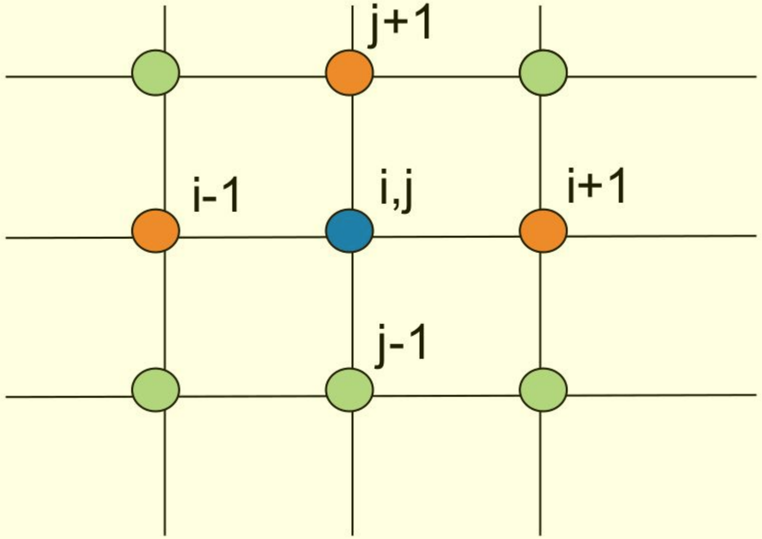
- What would a functional decomposition look like?
- Issues/obstacles with this domain decomposition?

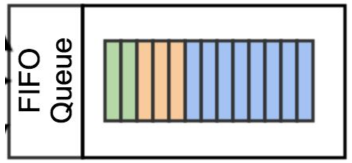
Functional decomposition

Each CPU gets part of the per-cell work

CPU 0:

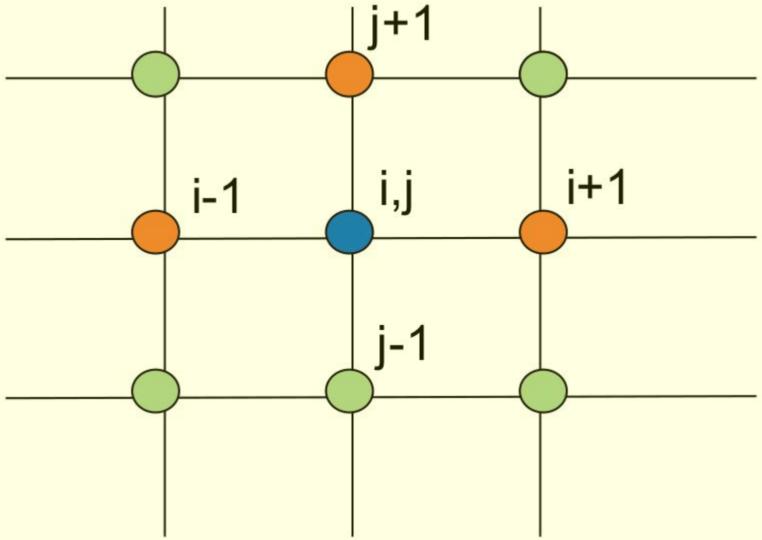
$tmp_{i,j} = F(v_{t-1}(\text{neighbors}))$





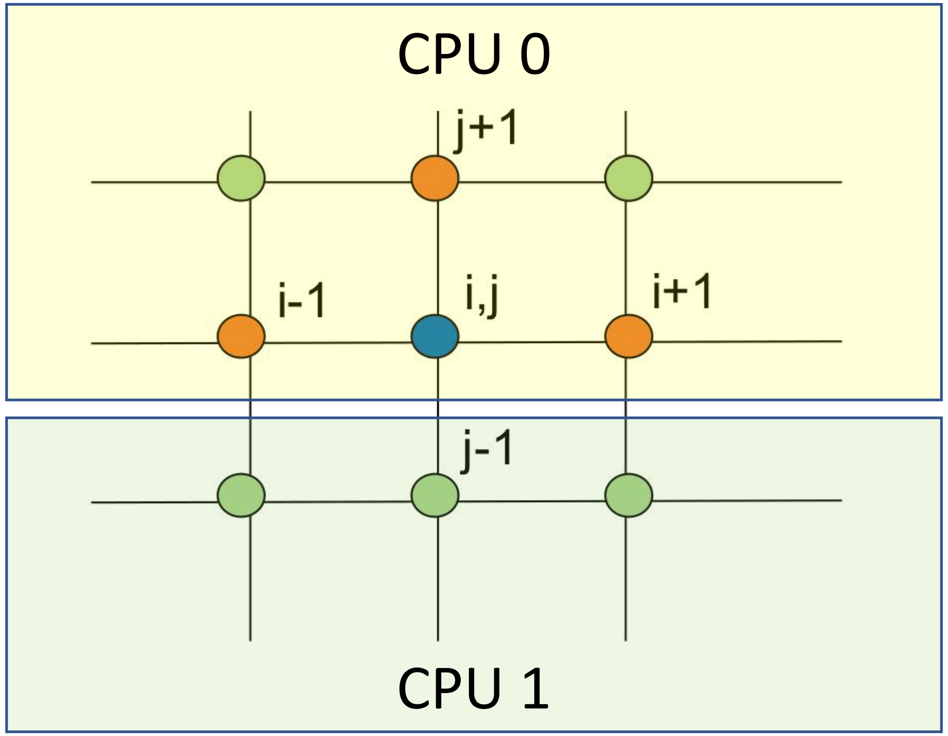
CPU 1:

$v_t(i,j) = tmp_{i,j}$



Domain decomposition

- Each CPU gets part of the input

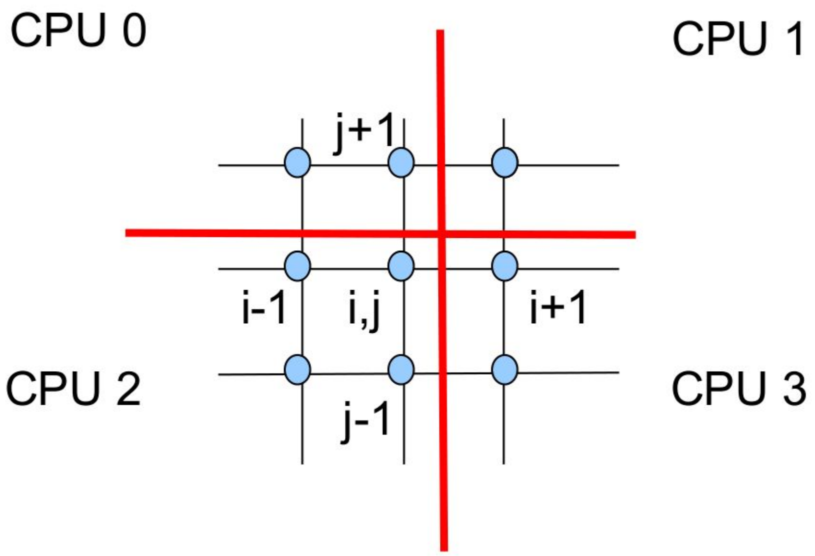


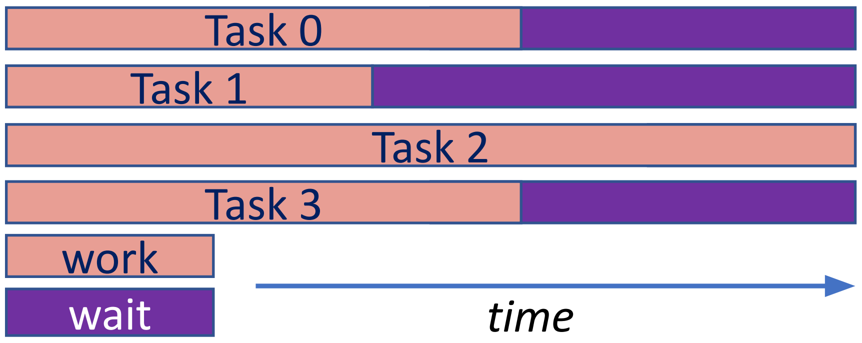
Issues?

- Accessing Data
  - Can we access v(i+1, j) from CPU 0
    - …as in a "normal" serial program?
    - Shared memory? Distributed?
  - Time to access v(i+1,j) == Time to access v(i-1,j) ?
  - *Scalability vs Latency*
- Control
  - Can we assign one vertex per CPU?
  - Can we assign one vertex per process/logical task?
  - *Task Management Overhead*
- *Load Balance*
- Correctness
  - order of reads and writes is non-deterministic
  - synchronization is required to enforce the order
  - *locks, semaphores, barriers, conditionals….*

Load Balancing

- Slowest task determines performance



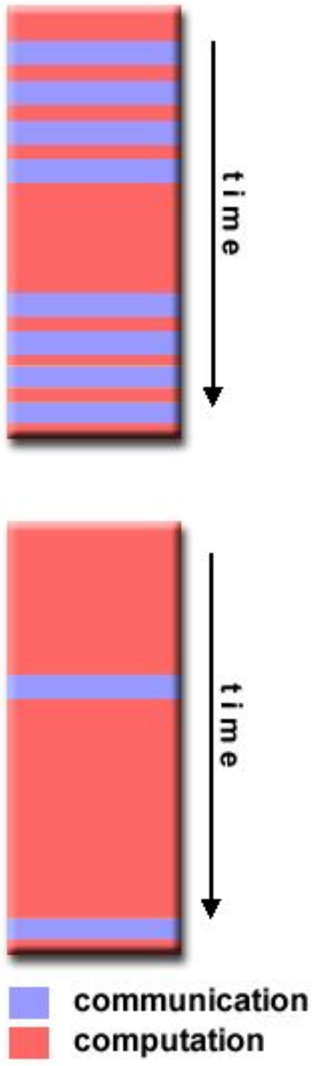


Granularity

$$G = \frac{Computation}{Communication}$$

- Fine-grain parallelism
  - G is small
  - Good load balancing
  - Potentially high overhead
  - Hard to get correct

- Coarse-grain parallelism
  - G is large
  - Load balancing is tough
  - Low overhead
  - Easier to get correct



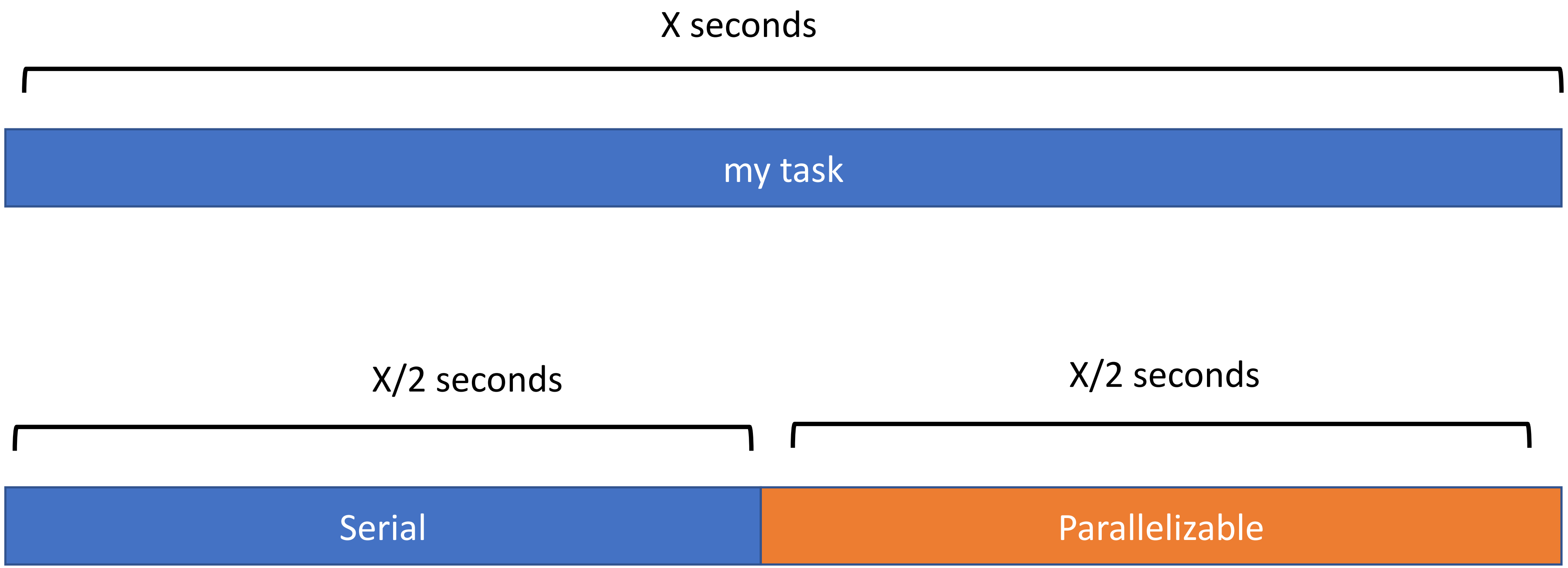
Performance: Amdahl’s law

- Speedup is bound by serial component
- Sp

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}}$$

$$Speedup(\#CPUs) = \frac{T_{serial}}{T_{parallel}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)}$$

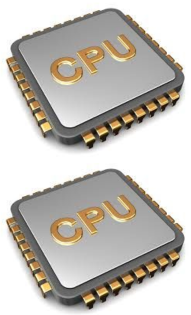
Amdahl’s law

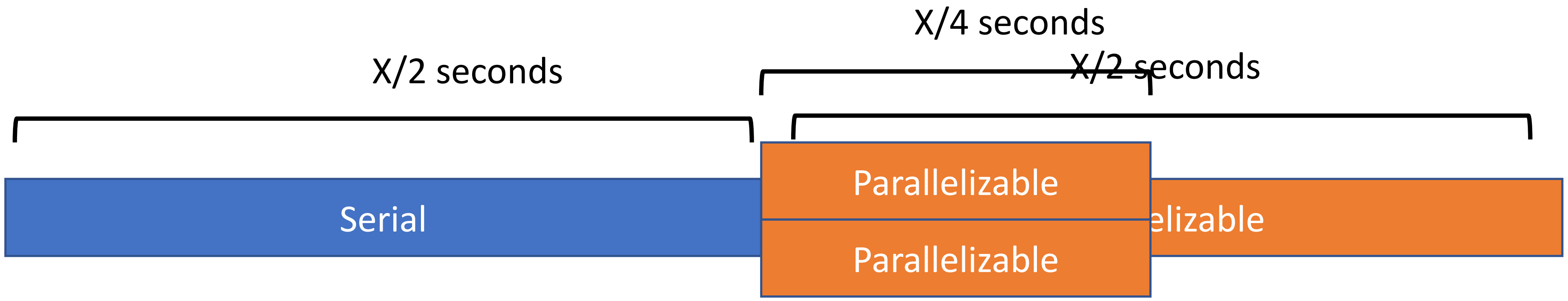


What makes something “serial” vs. parallelizable?

Amdahl's law

2 CPUs





End to end time: X seconds

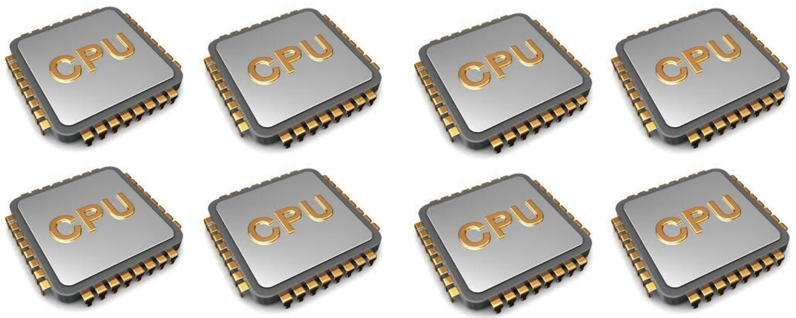
End to end time: (X/2 + X/4) = (3/4)X seconds

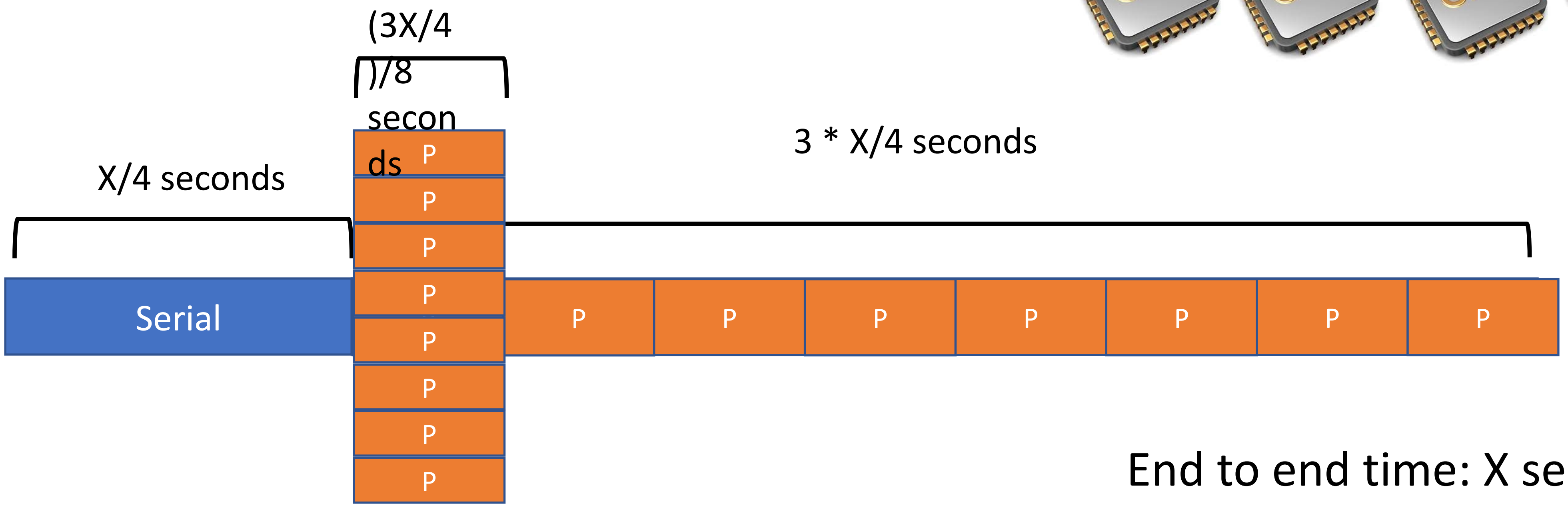
What is the "speedup" in this case?

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1-A)} = \frac{1}{\frac{.5}{2\text{ cpus}} + (1\text{-}.5)} = 1.333$$

Speedup exercise

8 CPUs



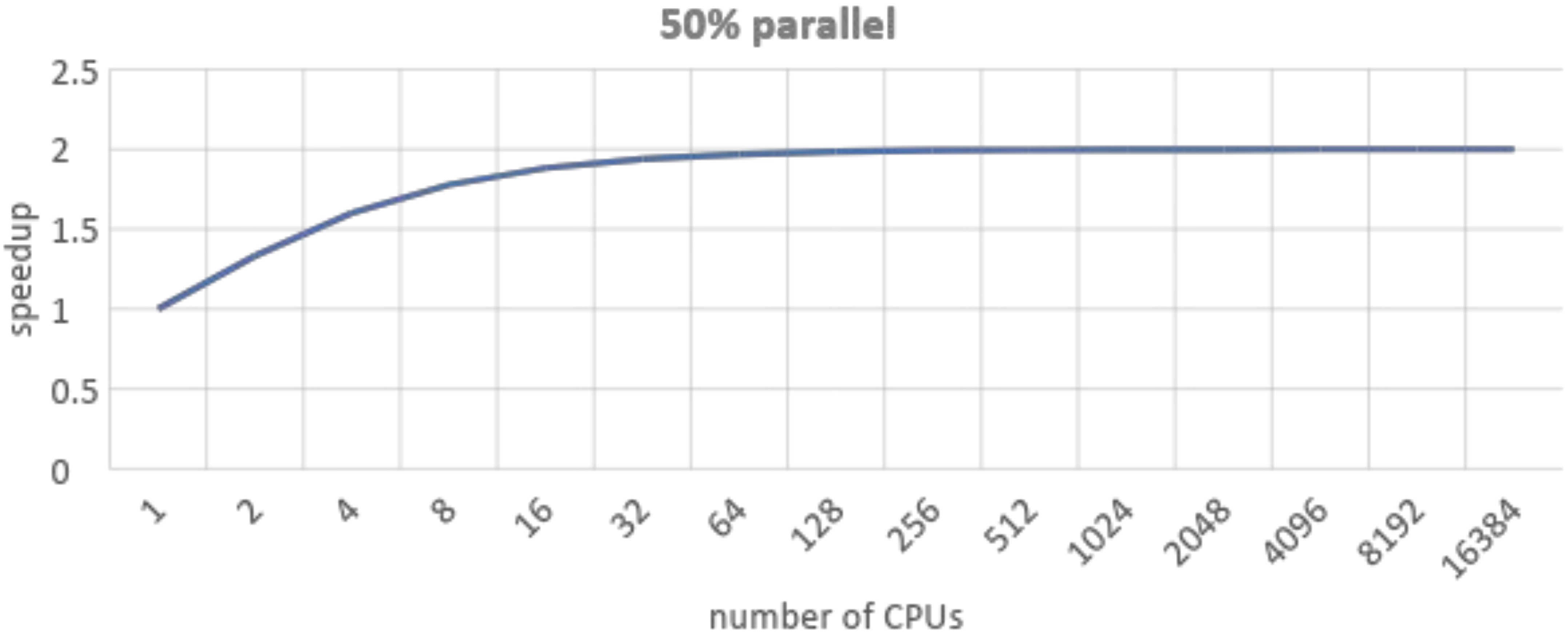


End to end time: X seconds

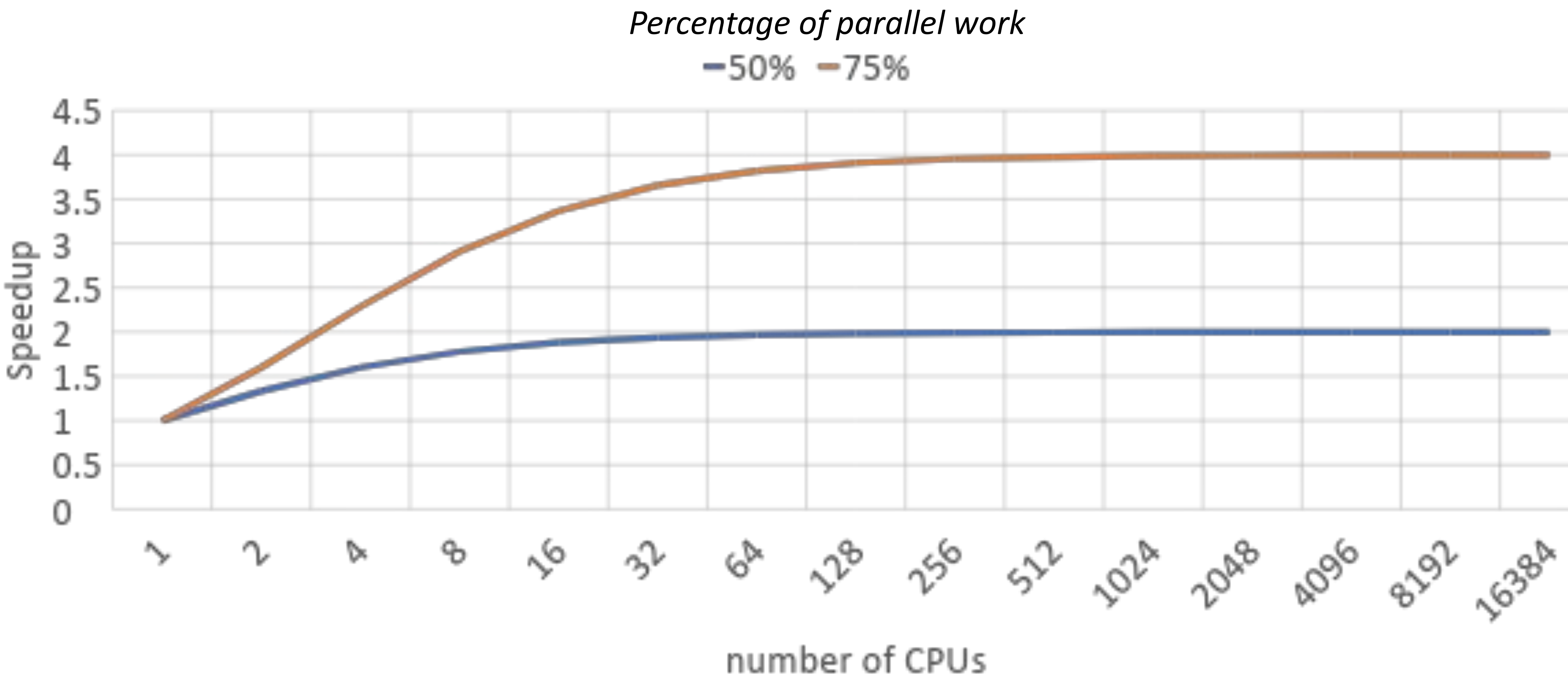
What is the “speedup” in this case?

$$Speedup = \frac{\text{serial run time}}{\text{parallel run time}} = \frac{1}{\frac{A}{\#CPUs} + (1 - A)} = \frac{1}{.75/8 + (1-.75)} = 2.91\text{x}$$

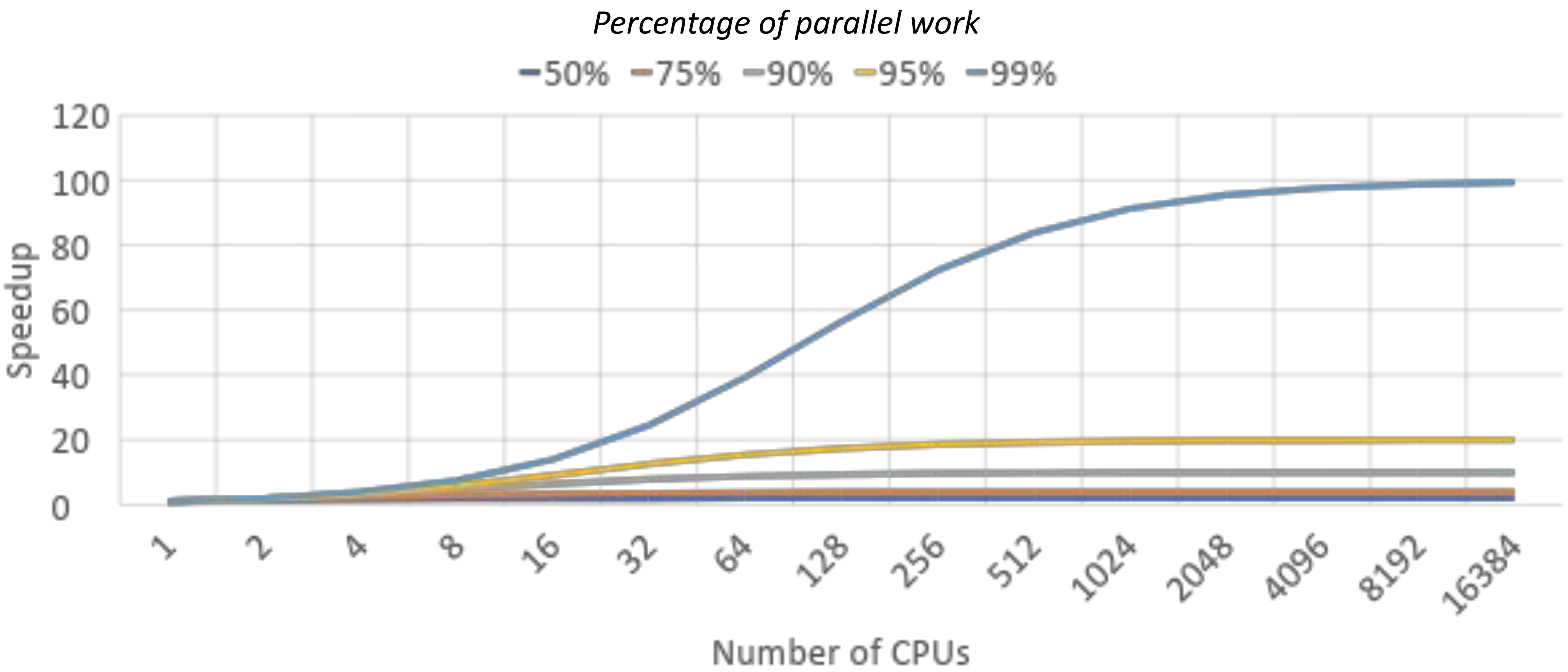
Amdahl Action Zone



Amdahl Action Zone



Amdahl Action Zone

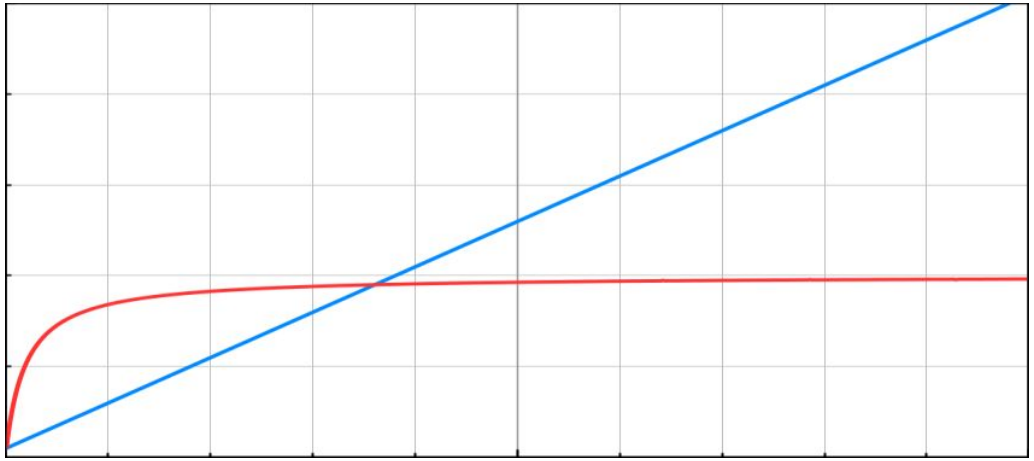


Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson

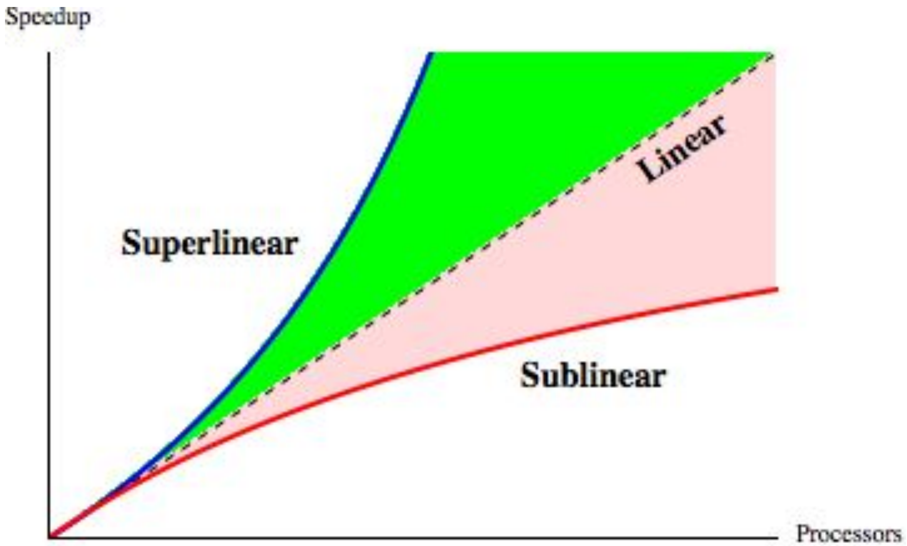
- $N = \#CPUs,\ S = serial\ portion = 1 - A$
- Amdahl's law: $Speedup(N) = \frac{1}{\frac{A}{N}+S}$
    - **Strong scaling:** *Speedup(N)* calculated with total work fixed
    - Solve same fixed size problem, #CPUs grows
    - Fixed parallel portion speedup stops increasing

- Gustafson's law: $Speedup(N) = N + (N{-}1)\cdot S$
    - **Weak scaling:** *Speedup(N)* calculated with work-per-CPU fixed
    - Add more CPUs Add more work granularity stays fixed
    - Problem size grows: solve larger problems
    - Consequence: speedup upper bound much greater



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:
  - Efficient **bubble sort** takes:
    - Parallel 40s
    - Serial 150s
    - $Speedup = \frac{150}{40} = 3.75$?
  - NO!
    - Serial quicksort runs in 30s
    - $\Longrightarrow Speedup = 0.75$



Concurrency and Correctness

If two threads execute this program concurrently, how many different final values of X are there?

Initially, X == 0.

Thread 1

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

Thread 2

```
void increment() {
    int temp = X;
    temp = temp + 1;
    X = temp;
}
```

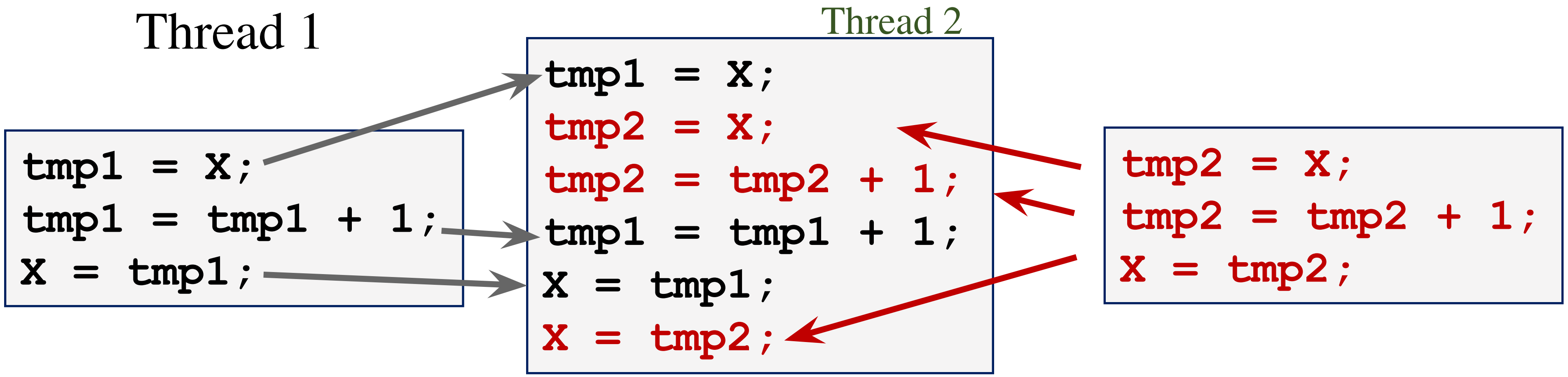
Answer:

- **A. 0**
- **B. 1**
- **C. 2**
- **D. More than 2**

Schedules/Interleavings

Model of concurrent execution

- Interleave statements from each thread into a single thread
- If **any** interleaving yields incorrect results, synchronization is needed



Thread 1

```
tmp1 = X;
tmp1 = tmp1 + 1;
X = tmp1;
```

Thread 2

```
tmp2 = X;
tmp2 = tmp2 + 1;
X = tmp2;
```

```
tmp1 = X;
tmp2 = X;
tmp2 = tmp2 + 1;
tmp1 = tmp1 + 1;
X = tmp1;
X = tmp2;
```

If X==0 initially, X == 1 at the end. WRONG result!

Locks fix this with Mutual Exclusion

```
void increment() {
   lock.acquire();
   int temp = X;
   temp = temp + 1;
   X = temp;
   lock.release();
}
```

Mutual exclusion ensures only safe interleavings

- *But it limits concurrency, and hence scalability/performance*

Is mutual exclusion a good abstraction?

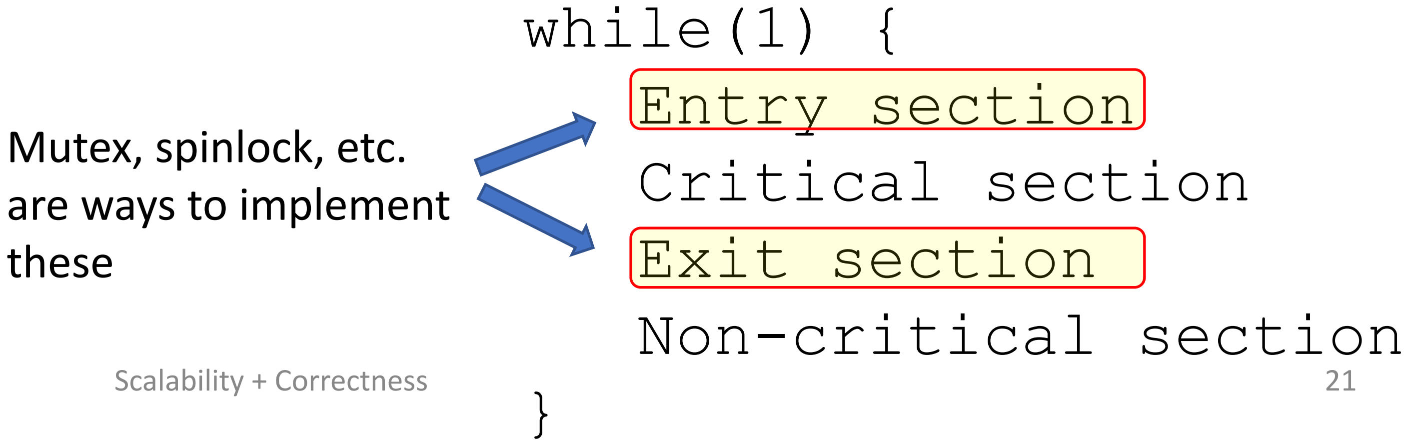
Correctness conditions

- Safety
  - Only one thread in the critical region
- Liveness
  - Some thread that enters the entry section eventually enters the critical region
  - Even if other thread takes forever in non-critical region
- Bounded waiting
  - ~~A thread that enters the entry section enters the critical section within some bounded number of operations.~~
  - *If a thread i is in entry section, then there is a bound on the number of times that other threads are allowed to enter the critical section before thread i's request is granted*

Theorem: Every property is a combination of a safety property and a liveness property.

-Bowen Alpern & Fred Schneider [1985]

https://www.cs.cornell.edu/fbs/publications/defliveness.pdf

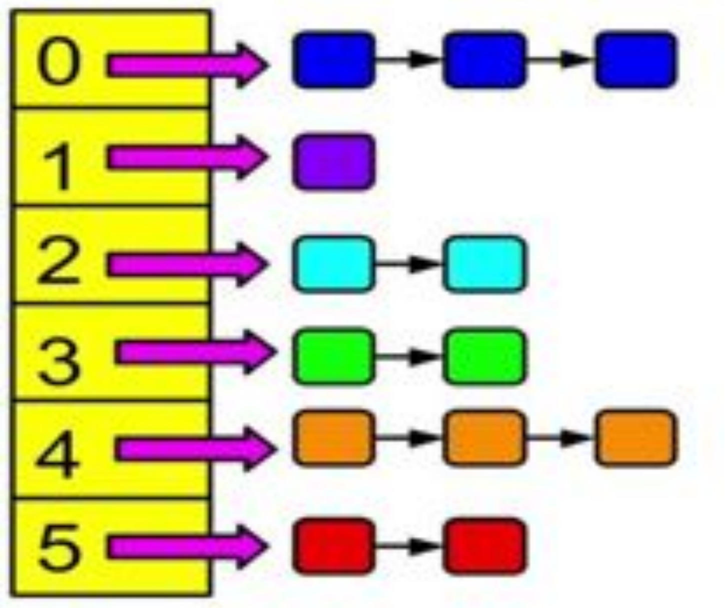


Mutex, spinlock, etc. are ways to implement these

```
while(1) {
    Entry section
    Critical section
    Exit section
    Non-critical section
}
```

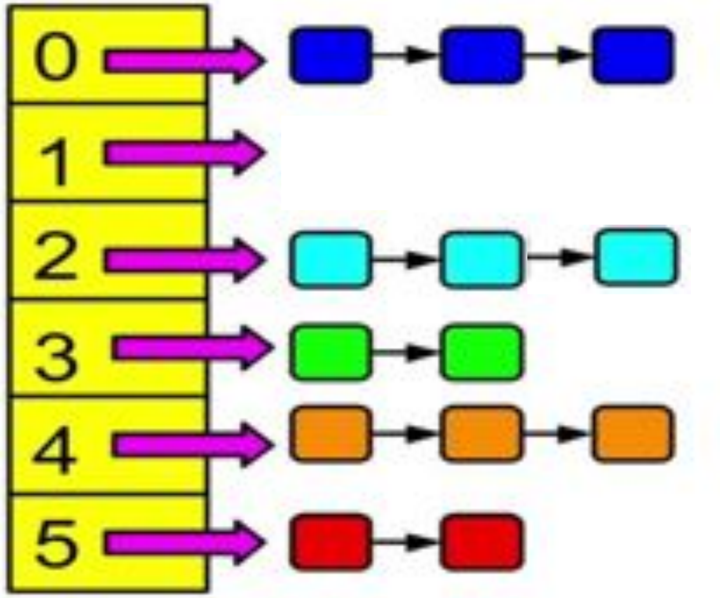
Let’s talk concurrency control

Consider a hash-table



Let’s talk concurrency control

Consider a hash-table

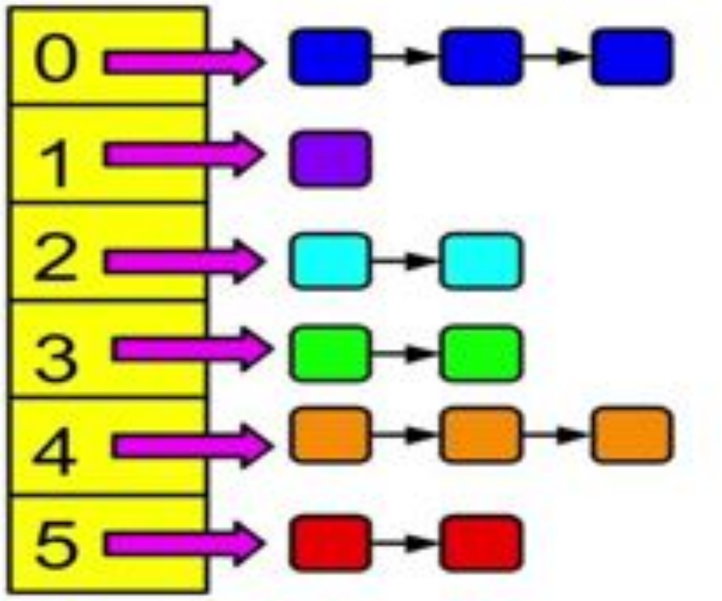


thread T1

```
ht.add(🟦);


if(ht.contains(🟪))
   ht.del(🟪);
```

Let’s talk concurrency control

Consider a hash-table

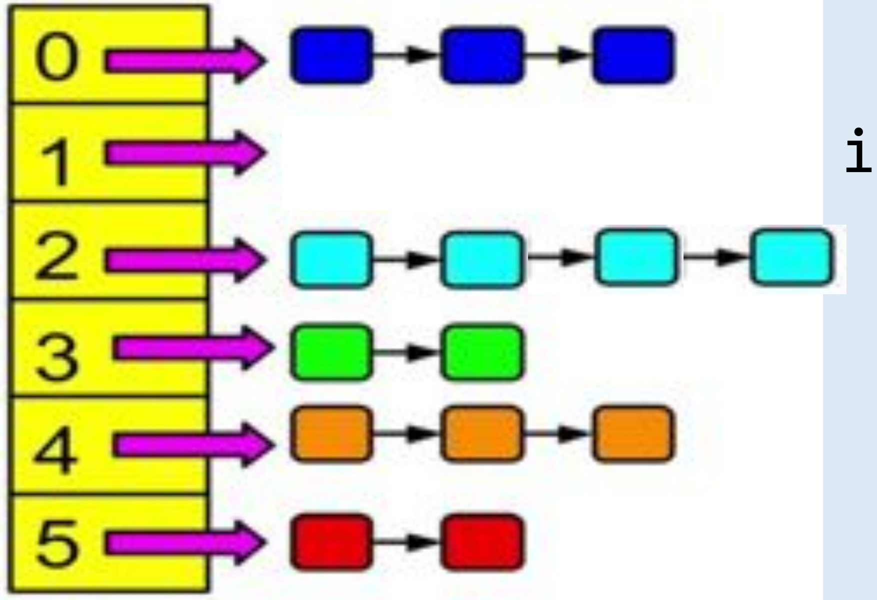


thread T1	thread T2
ht.add(🟦);	ht.add(🟦);

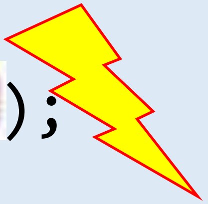
if(ht.contains(🟪))
    ht.del(🟪);
	if(ht.contains(🟪))
	    ht.del(🟪);

Let’s talk concurrency control

Consider a hash-table

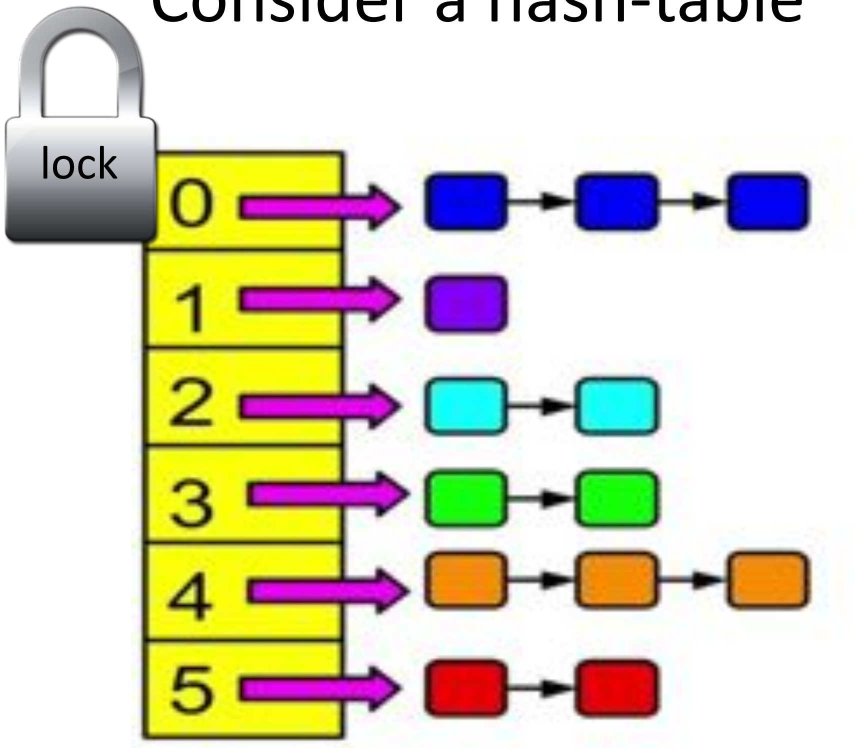


thread T1	thread T2
`ht.add(🟦);`	
	`ht.add(🟦);`
`if(ht.contains(🟪))`	
	`if(ht.contains(🟪))`
	`    ht.del(🟪);`
`    ht.del(🟪);`	



Pessimistic concurrency control: coarse locks

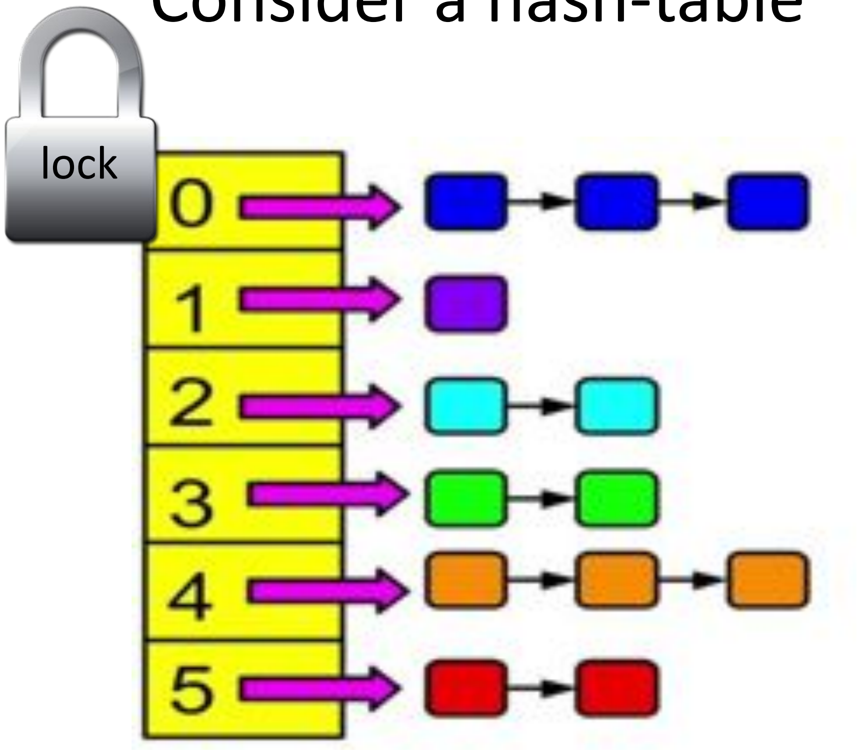
Consider a hash-table



thread T1	thread T2
`ht.lock();`	
`ht.add(🟦);`	
`if(ht.contains(🟪))`	
`    ht.del(🟪);`	
`ht.unlock();`	

Pessimistic concurrency control: coarse locks

Consider a hash-table



thread T1

```
ht.lock();
ht.add(🟦);



if(ht.contains(🟪))
    ht.del(🟪);
ht.unlock();
```

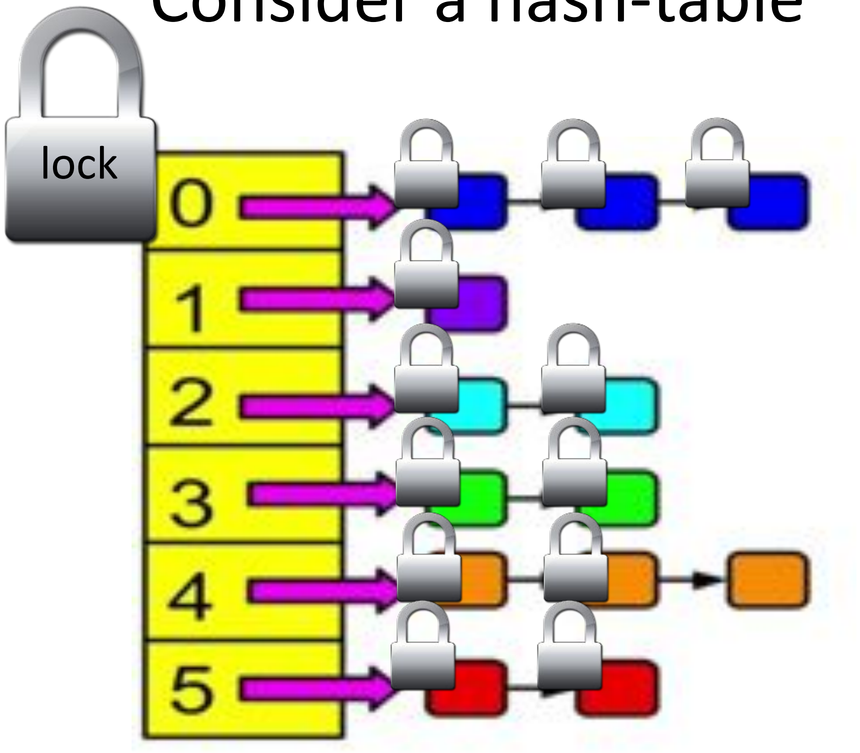
thread T2

```
ht.lock();
ht.add(🟧);



if(ht.contains(🟩))
    ht.del(🟥);
ht.unlock();
```

Coarse lock:
Non-conflicting ops serialized
Low Complexity -- Low Performance

Pessimistic concurrency control: fine locks

Consider a hash-table



thread T1

```
figure-out-locks();
lock-them-inorder();
ht.add(🟦);


if(ht.contains(🟪))
    ht.del(🟪);
unlock-locks();
```

thread T2

```
figure-out-locks();
lock-them-inorder();
ht.add(🟧);


if(ht.contains(🟩))
    ht.del(🟥);
unlock-locks();
```

Fine-grain lock:
Non-conflicting parallel
High Complexity -- High Performance

Why Locks are Hard

- Coarse-grain locks
  - Simple to develop
  - Easy to avoid deadlock
  - Few data races
  - Limited concurrency

```
// WITH FINE-GRAIN LOCKS
void move(T s, T d, Obj key){
  LOCK(s);
  LOCK(d);
  tmp = s.remove(key);
  d.insert(key, tmp);
  UNLOCK(d);
  UNLOCK(s);
}
```

- Fine-grain locks
  - Greater concurrency
  - Greater code complexity
  - Potential deadlocks
    - Not composable
  - Potential data races
    - Which lock to lock?

Thread 0	**Thread 1**
`move(a, b, key1);`	
	`move(b, a, key2);`

DEADLOCK!