Foundations:
Concurrency Concerns
Synchronization Basics

Chris Rossbach

CS378H

Today

- Questions?
- Administrivia
  - You've started Lab 1 right?
- Foundations
  - Parallelism
  - Basic Synchronization
  - Threads/Processes/Fibers, Oh my!
  - Cache coherence (maybe)

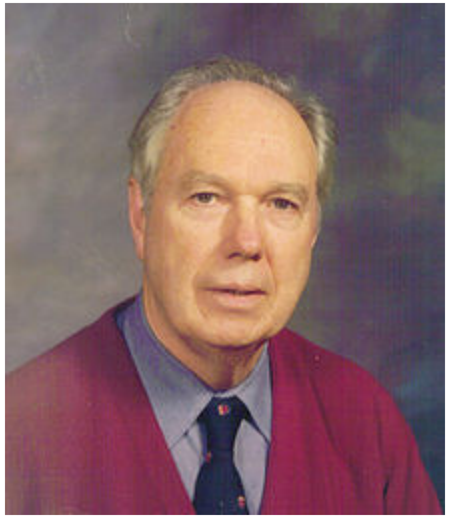
- Acknowledgments: some materials in this lecture borrowed from
  - Emmett Witchel (who borrowed them from: Kathryn McKinley, Ron Rockhold, Tom Anderson, John Carter, Mike Dahlin, Jim Kurose, Hank Levy, Harrick Vin, Thomas Narten, and Emery Berger)
  - Mark Silberstein (who borrowed them from: Blaise Barney, Kunle Olukoton, Gupta)
  - Andy Tannenbaum
  - Don Porter
  - me…
  - Photo source: https://img.devrant.com/devrant/rant/r_10875_uRYQF.jpg

Faux Quiz (answer any 2, 5 min)

- Who was Flynn? Why is her/his taxonomy important?
- How does domain decomposition differ from functional decomposition? Give examples of each.
- Can a SIMD parallel program use functional decomposition? Why/why not?
- What is an RMW instruction? How can they be used to construct synchronization primitives? How can sync primitives be constructed without them?

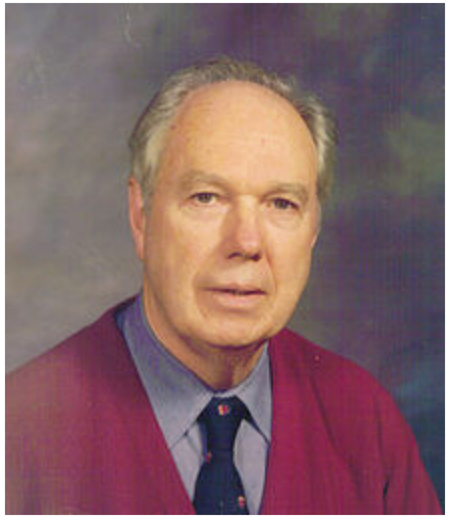
Who is Flynn?

Who is Flynn?



Who is Flynn?

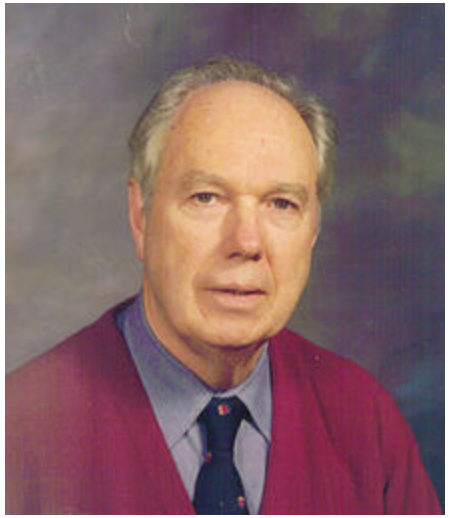
Michael J. Flynn



Who is Flynn?

Michael J. Flynn

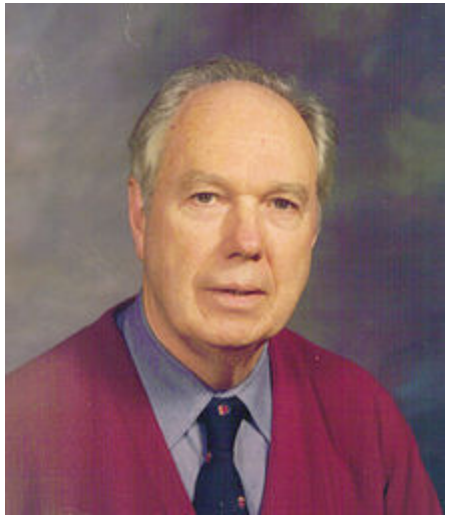
- Emeritus at Stanford



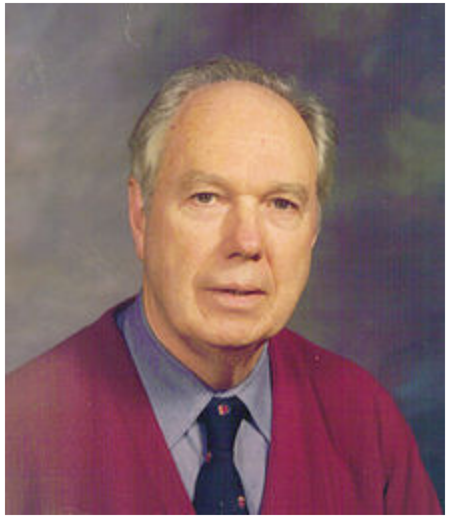
Who is Flynn?

Michael J. Flynn

- Emeritus at Stanford
- Proposed taxonomy in 1966 (!!)



Who is Flynn?



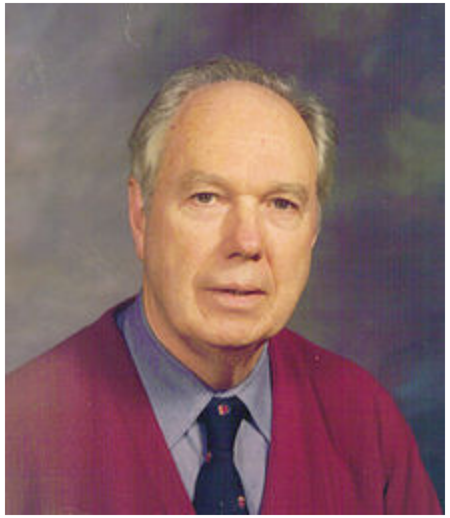
Michael J. Flynn

- Emeritus at Stanford
- Proposed taxonomy in 1966 (!!)
- 30 pages of publication titles

Who is Flynn?

Michael J. Flynn

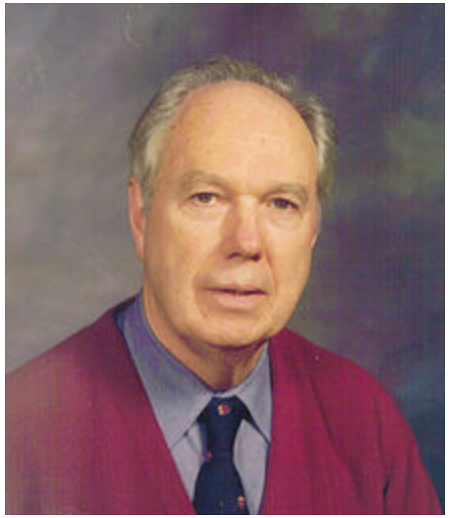
- Emeritus at Stanford
- Proposed taxonomy in 1966 (!!)
- 30 pages of publication titles
- Founding member of SIGARCH



Who is Flynn?

Michael J. Flynn

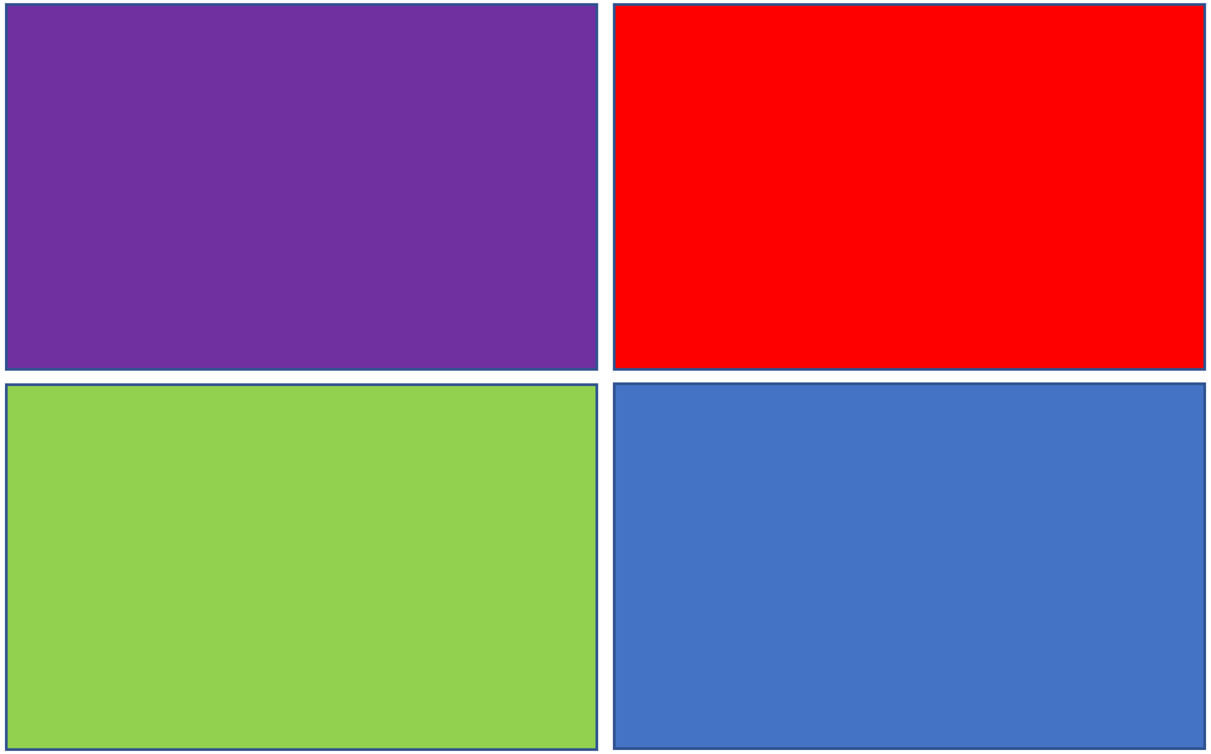
- Emeritus at Stanford
- Proposed taxonomy in 1966 (!!)
- 30 pages of publication titles
- Founding member of SIGARCH



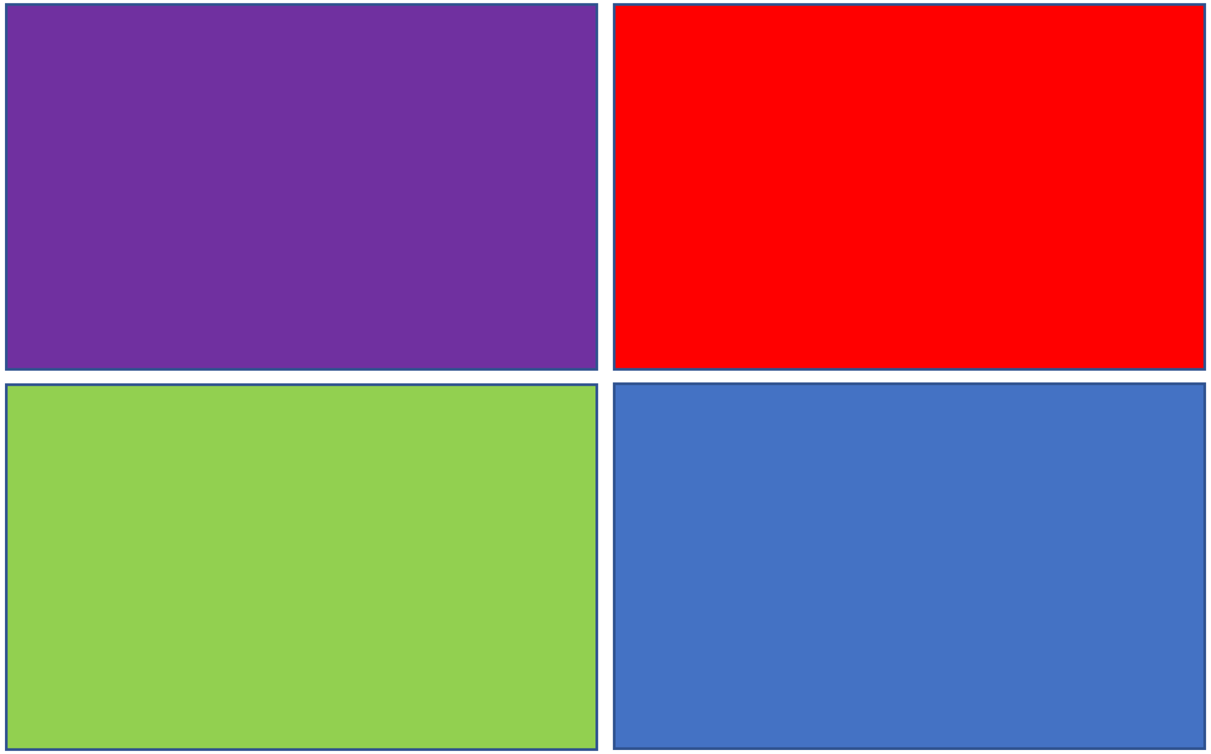
- (Thanks Wikipedia)

Review: Flynn’s Taxonomy

Review: Flynn’s Taxonomy



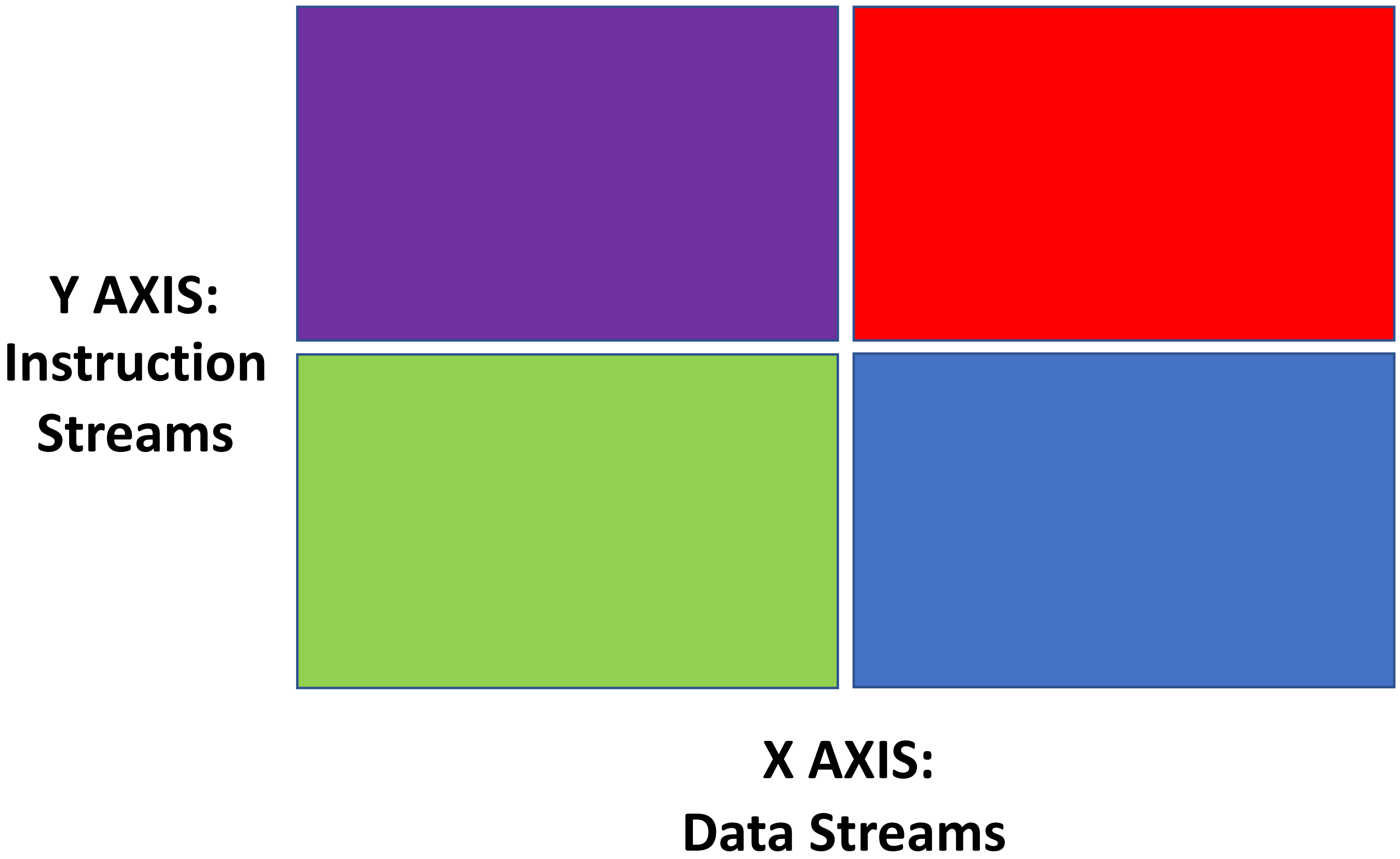
Review: Flynn’s Taxonomy



Y AXIS:

X AXIS:

Review: Flynn’s Taxonomy

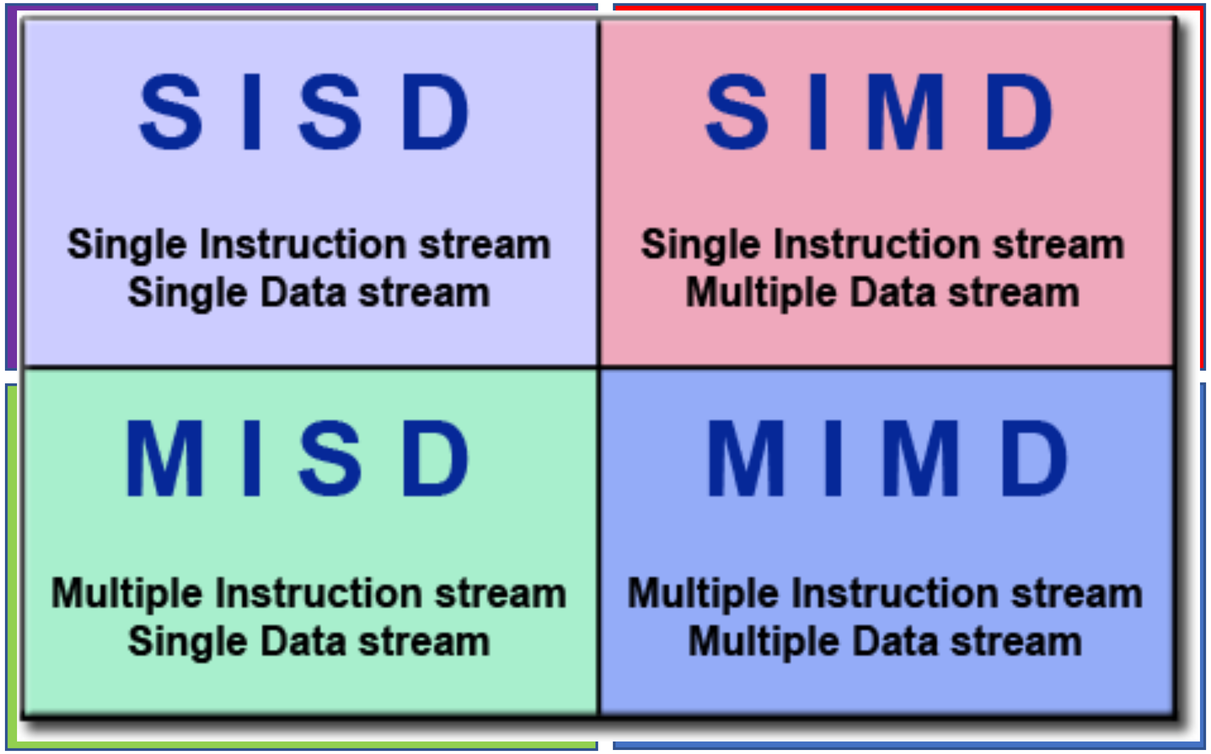


Y AXIS:
Instruction
Streams

X AXIS:
Data Streams

Review: Flynn’s Taxonomy

Y AXIS:
Instruction Streams



SISD Single Instruction stream Single Data stream	**SIMD** Single Instruction stream Multiple Data stream
MISD Multiple Instruction stream Single Data stream	**MIMD** Multiple Instruction stream Multiple Data stream

X AXIS:
Data Streams

Review: Problem Partitioning

Review: Problem Partitioning

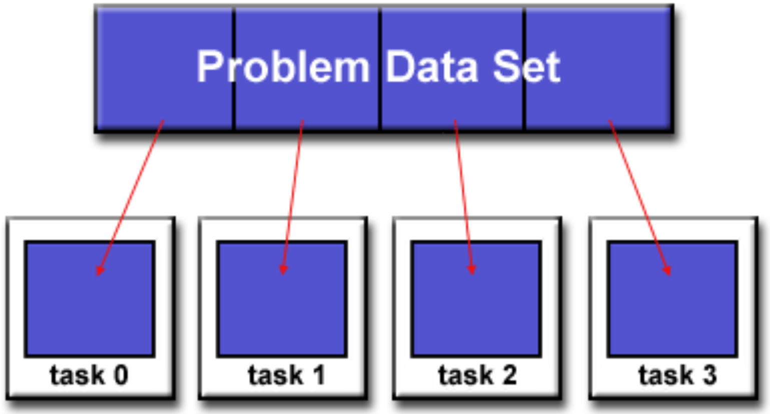
- Domain Decomposition

Review: Problem Partitioning

- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both

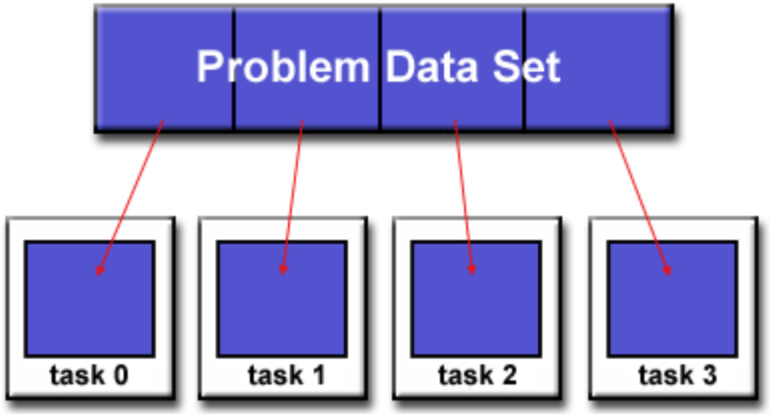
Review: Problem Partitioning

- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both



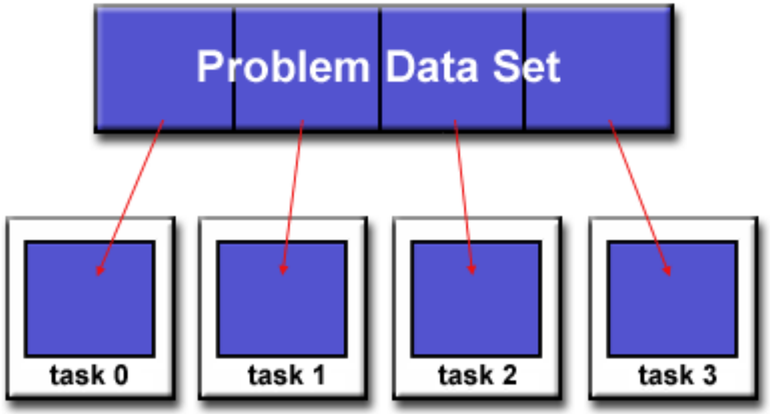
Review: Problem Partitioning

- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both
- Functional Decomposition



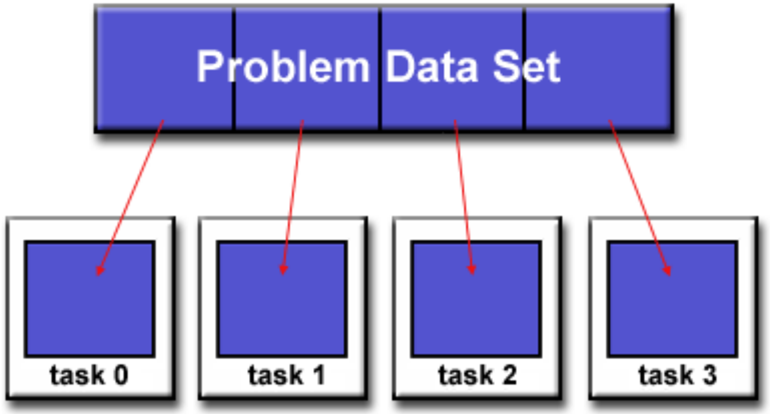
Review: Problem Partitioning

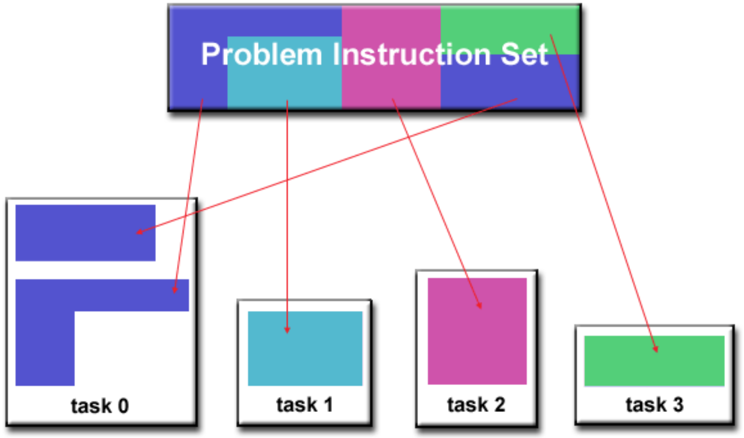
- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both
- Functional Decomposition
  - MPMD
  - Independent Tasks
  - Pipelining



Review: Problem Partitioning

- Domain Decomposition
  - SPMD
  - Input domain
  - Output Domain
  - Both
- Functional Decomposition
  - MPMD
  - Independent Tasks
  - Pipelining





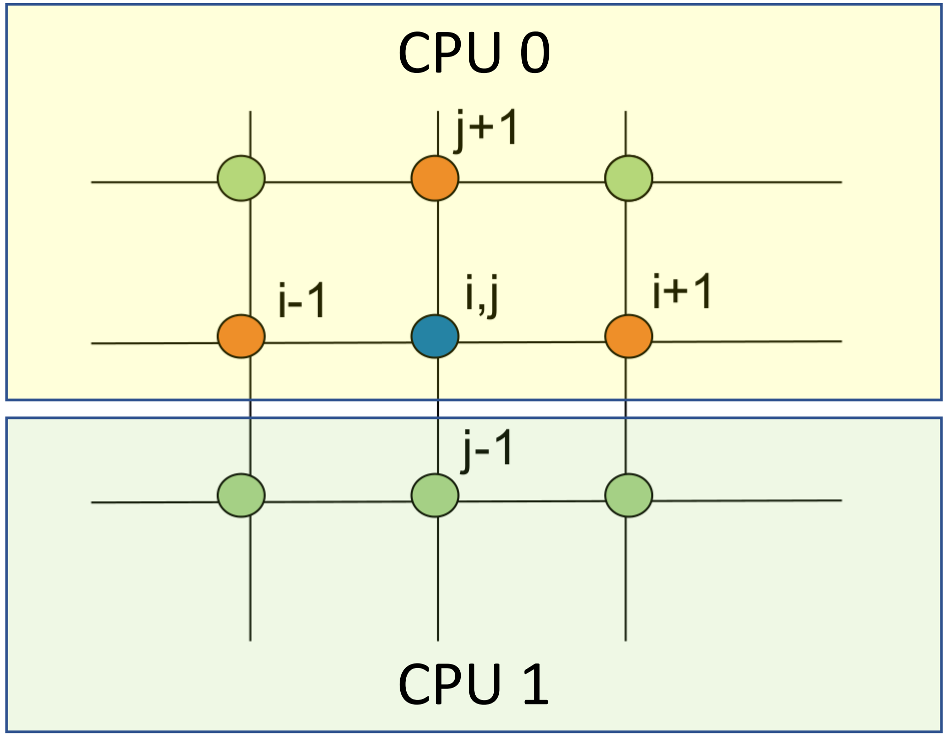
Domain decomposition

Domain decomposition

- Each CPU gets part of the input

Domain decomposition

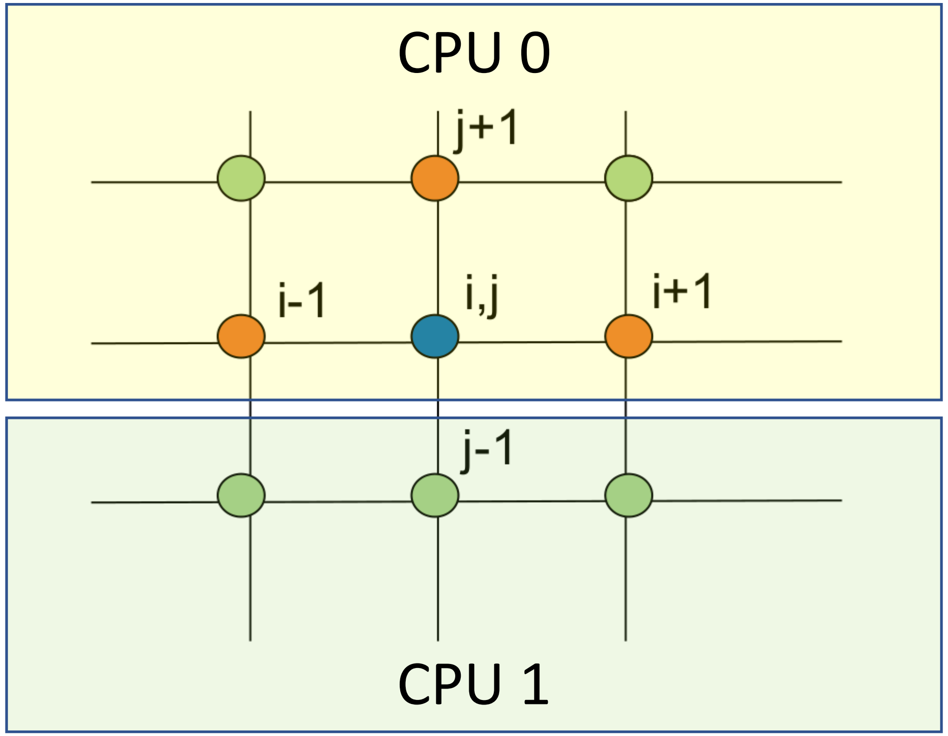
- Each CPU gets part of the input



Domain decomposition

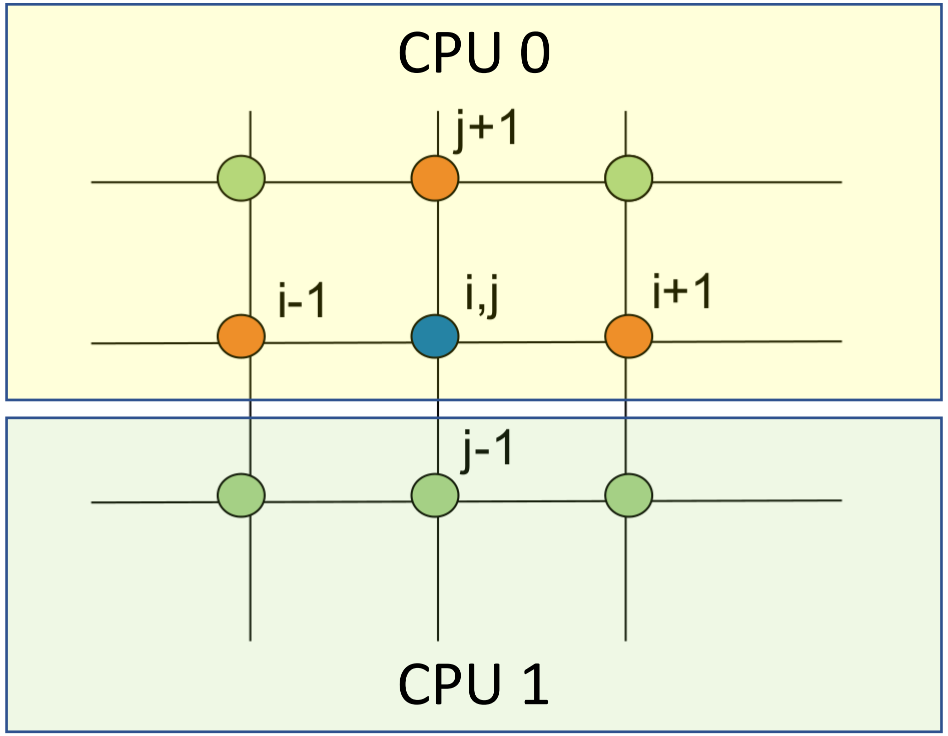
- Each CPU gets part of the input

Issues?



Domain decomposition

- Each CPU gets part of the input

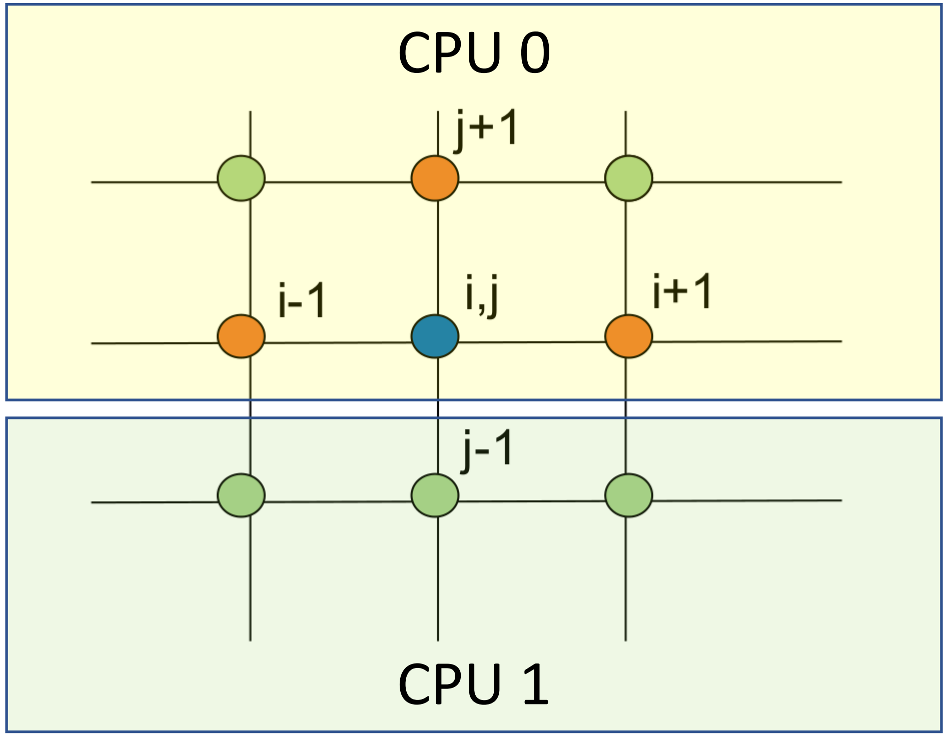


Issues?

- Accessing Data

Domain decomposition

- Each CPU gets part of the input

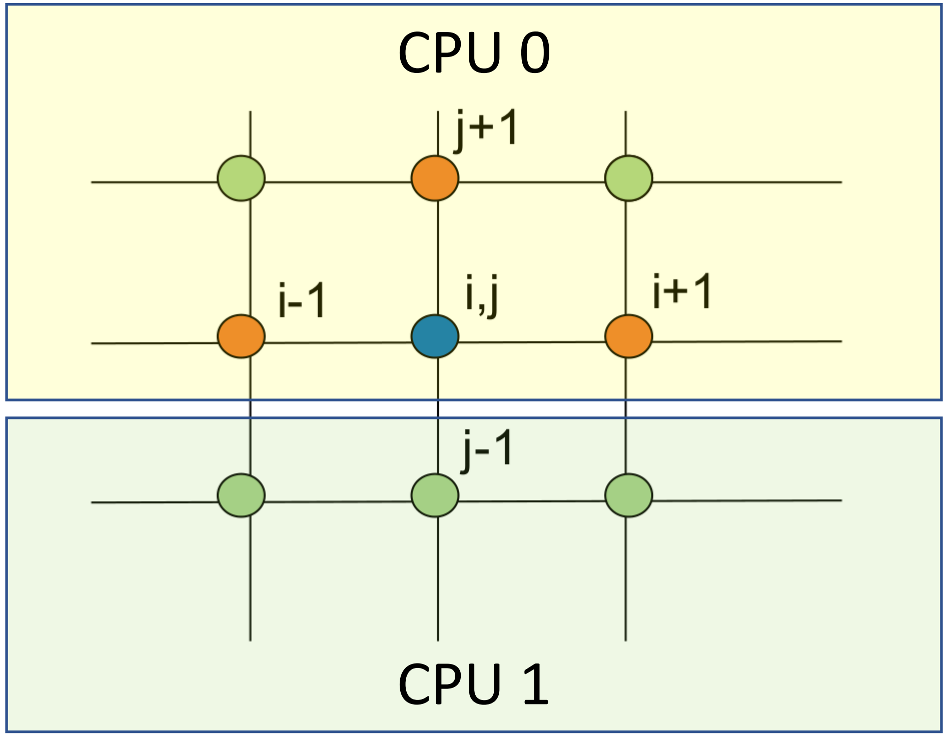


Issues?

- Accessing Data
    - Can we access v(i+1, j) from CPU 0

Domain decomposition

- Each CPU gets part of the input

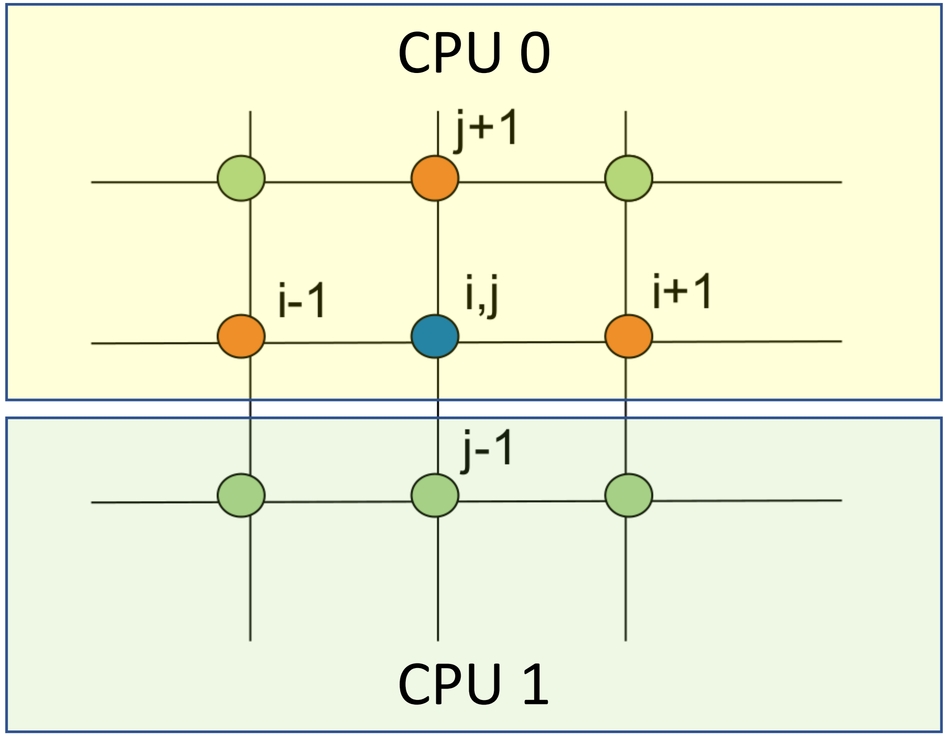


Issues?

- 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*

Domain decomposition

- Each CPU gets part of the input

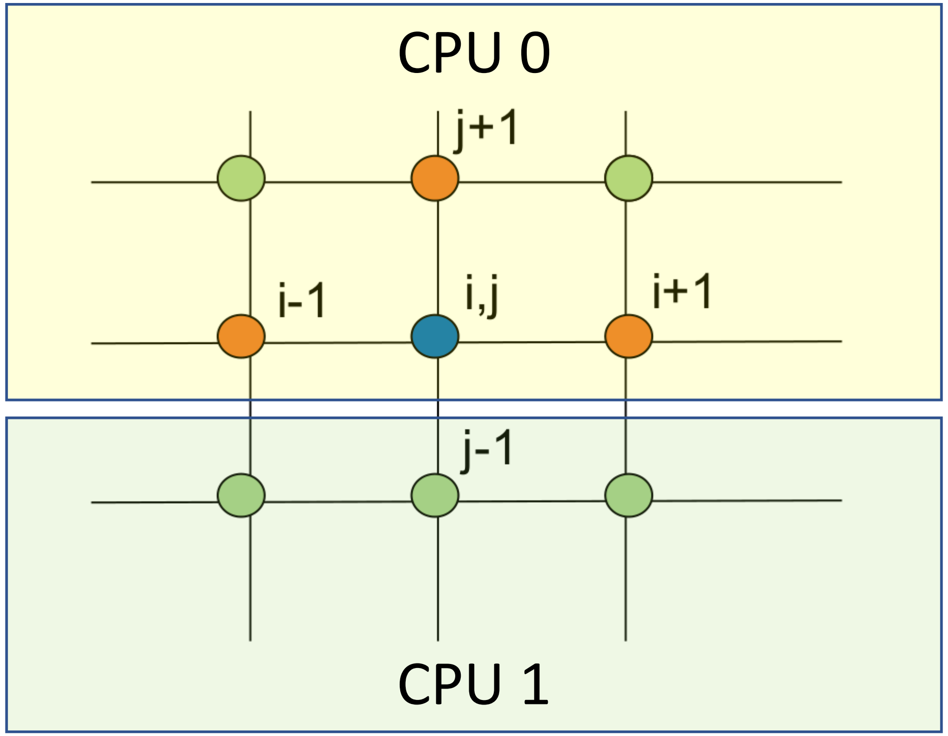


Issues?

- 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*

Domain decomposition

- Each CPU gets part of the input

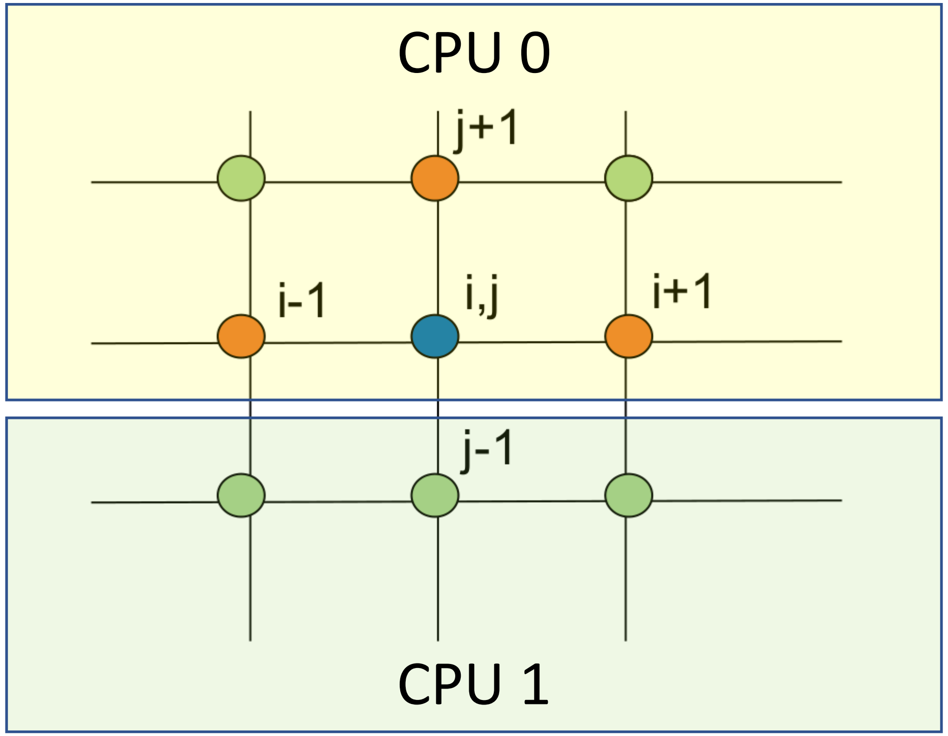


Issues?

- 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*

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….*

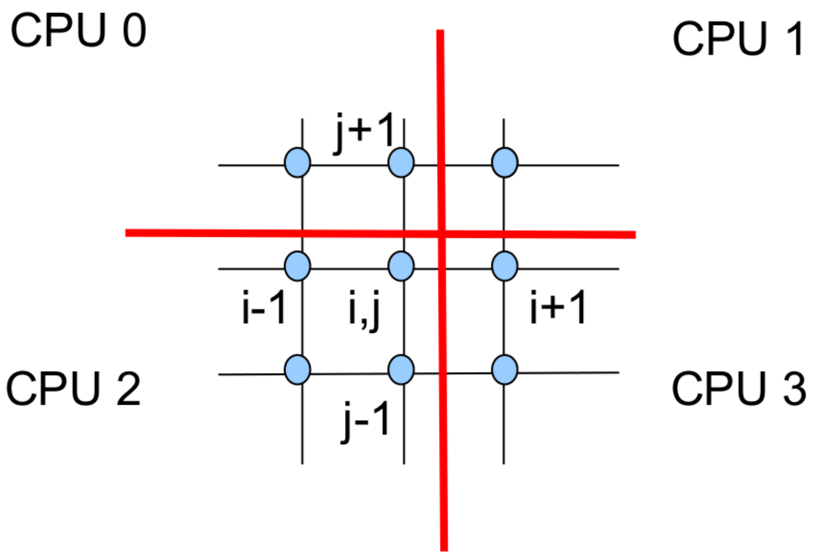
Load Balancing

Load Balancing

- Slowest task determines performance

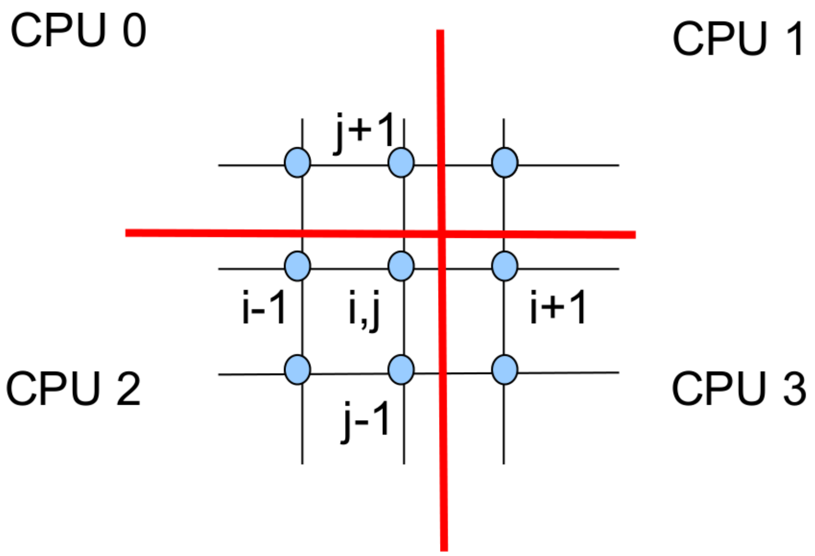
Load Balancing

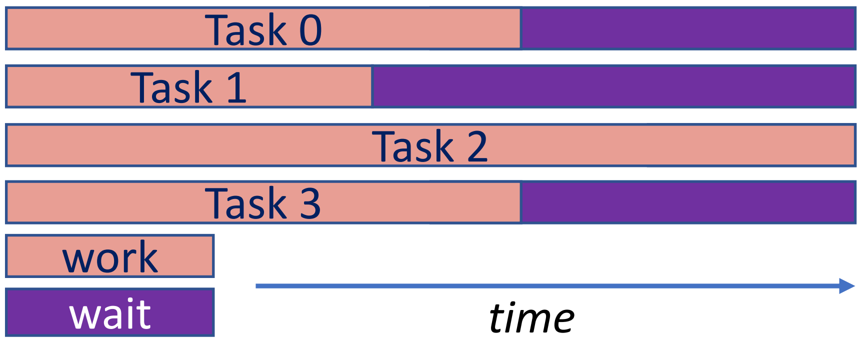
- Slowest task determines performance



Load Balancing

- Slowest task determines performance





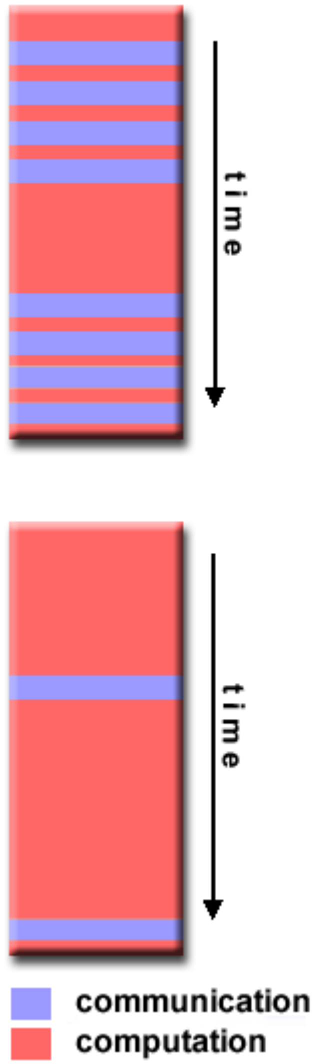
Granularity

Granularity

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

Granularity

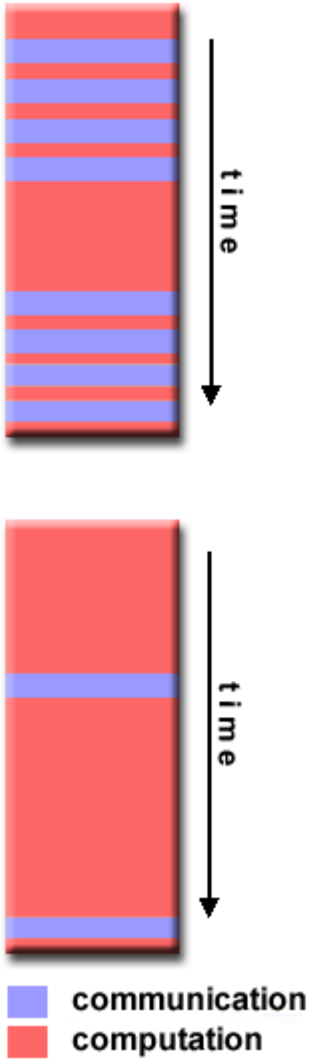
$$G = \frac{Computation}{Communication}$$



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

Performance: Amdahl’s law

- Speedup is bound by serial component
- Split program serial time ($T_{serial} = 1$) into
  - Ideally parallelizable portion: A
    - assuming perfect load balancing, identical speed, no overheads
  - Cannot be parallelized (serial) portion : $1 - A$
  - Parallel time:

$$\mathrm{T}_{\mathrm{parallel}} = \frac{A}{\#CPUs} + (1 - A)$$

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

Performance: Amdahl’s law

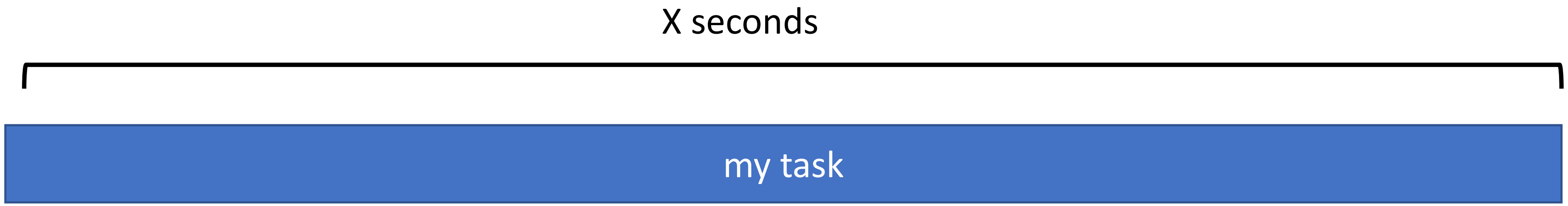
- Speedup is bound by serial component
- Sp

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

$_{parallel}$ $\#CPUs$

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

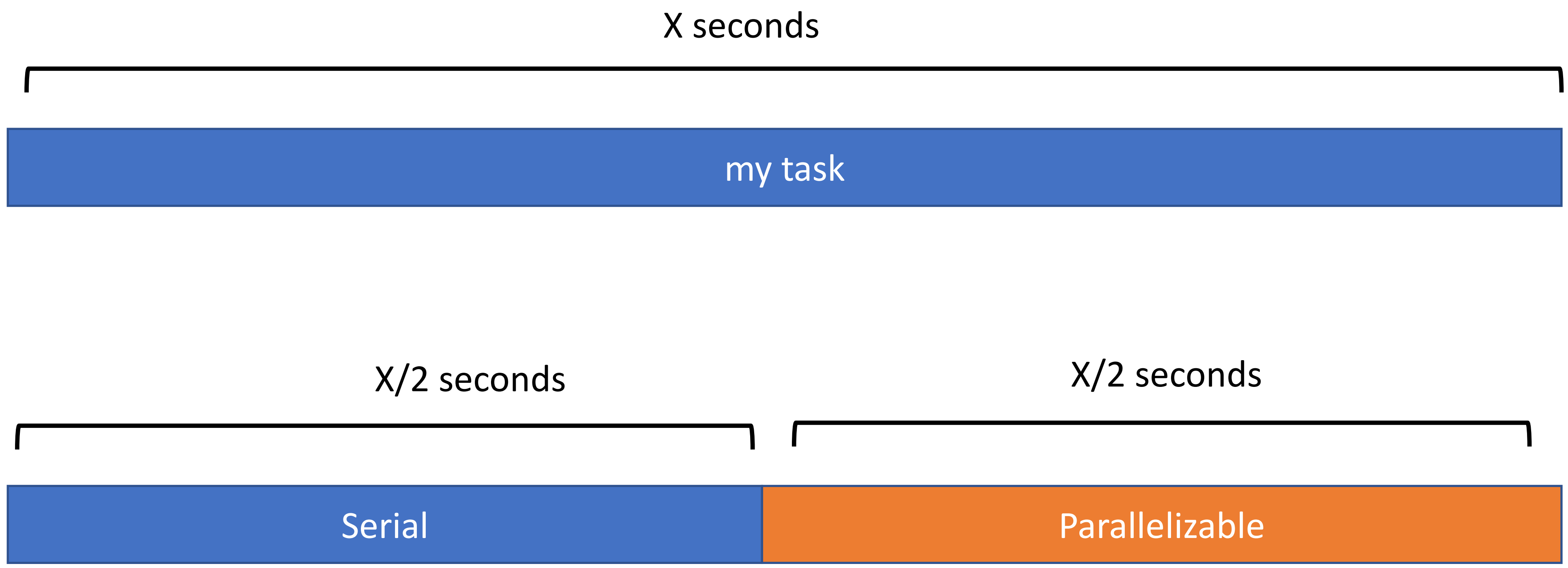
Amdahl’s law



X seconds

my task

Amdahl’s law



X seconds

my task

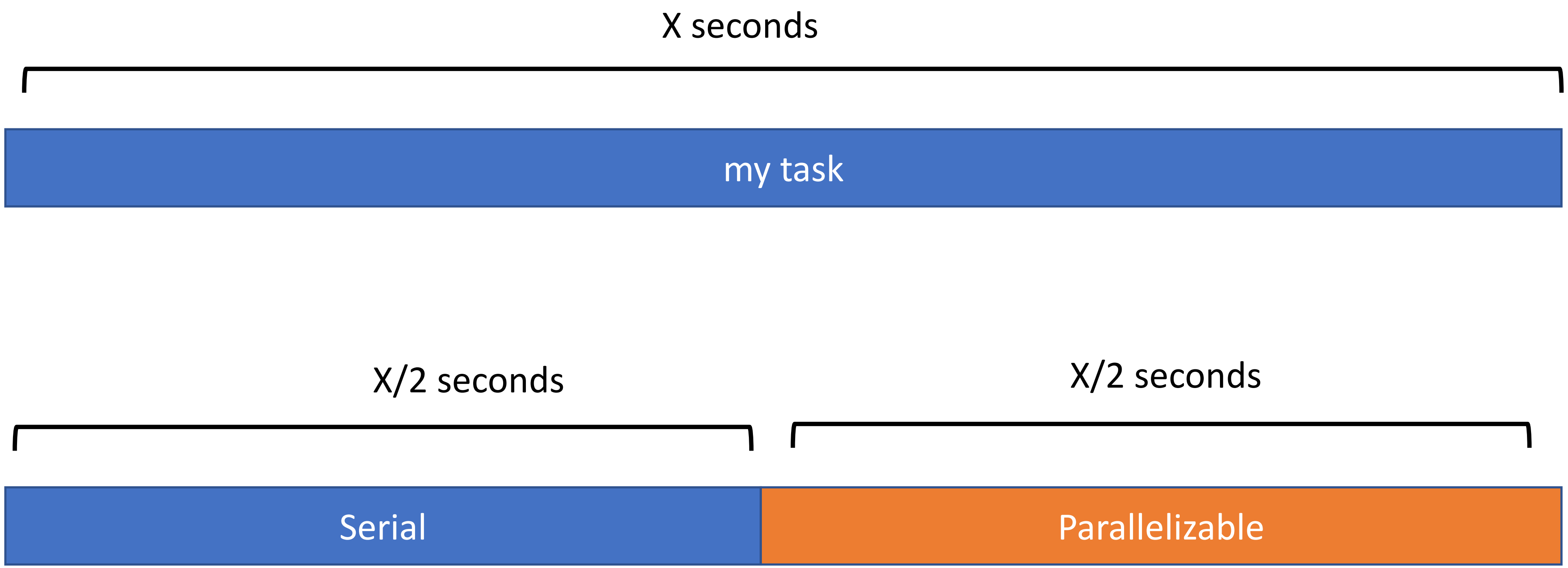
X/2 seconds

X/2 seconds

Serial

Parallelizable

Amdahl’s law



X seconds

my task

X/2 seconds

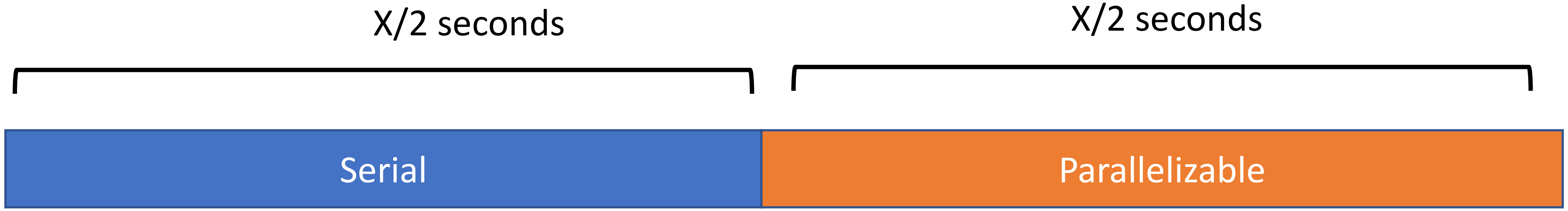
X/2 seconds

Serial

Parallelizable

What makes something “serial” vs. parallelizable?

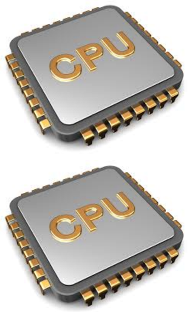
Amdahl’s law

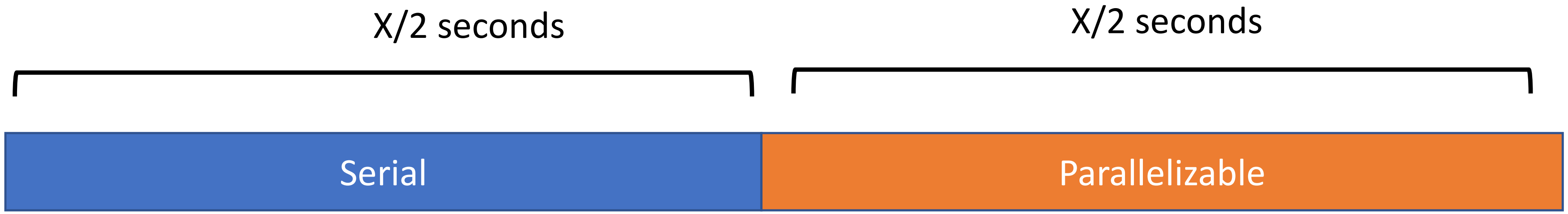


End to end time: X seconds

Amdahl’s law

2 CPUs

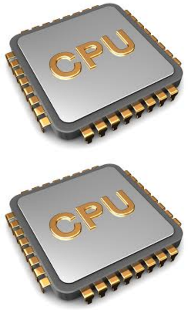


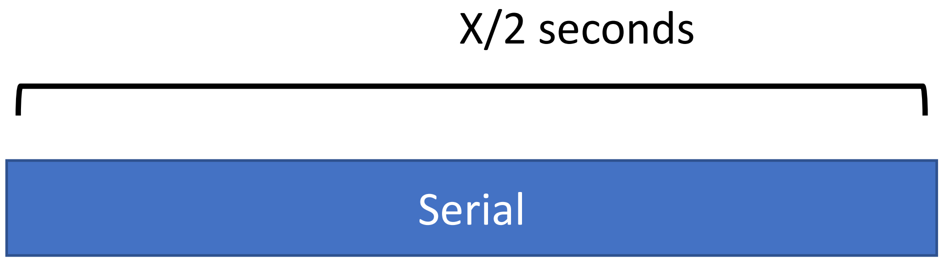


End to end time: X seconds

Amdahl’s law

2 CPUs





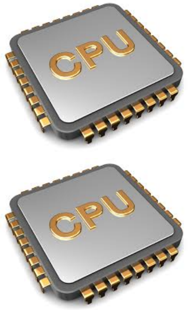
X/2 seconds

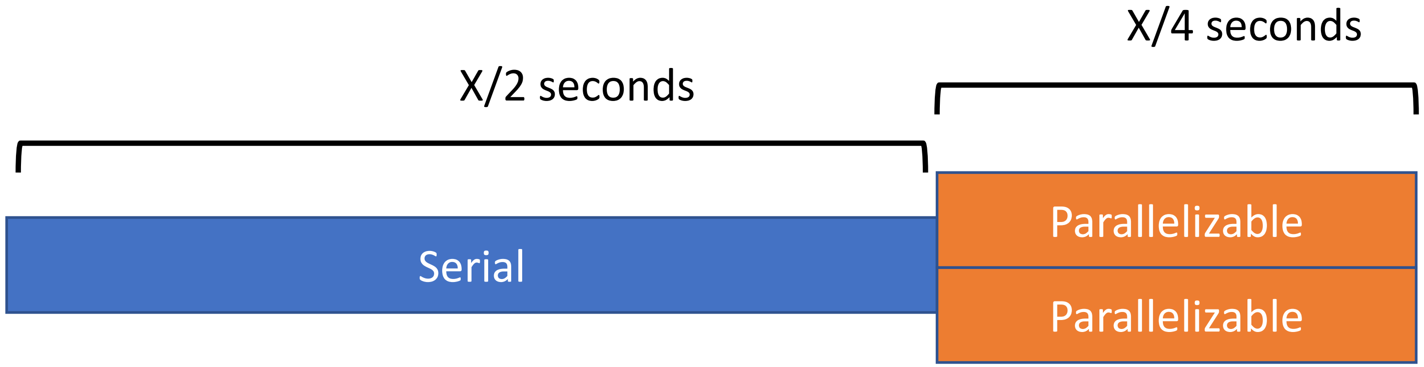
Serial

End to end time: X seconds

Amdahl’s law

2 CPUs

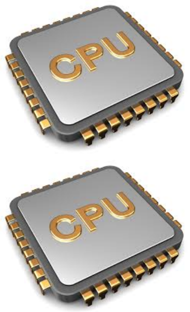


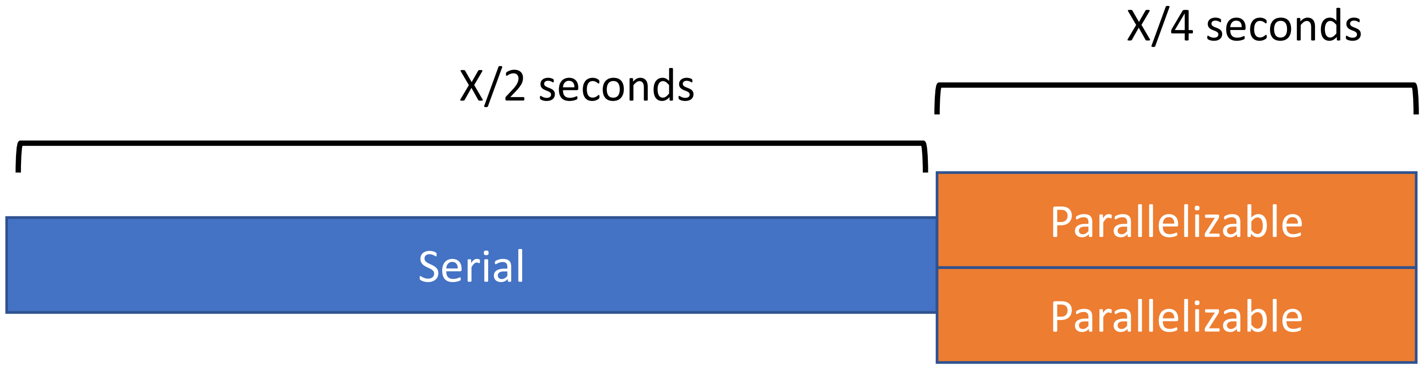


End to end time: X seconds

Amdahl’s law

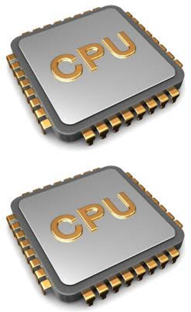
2 CPUs

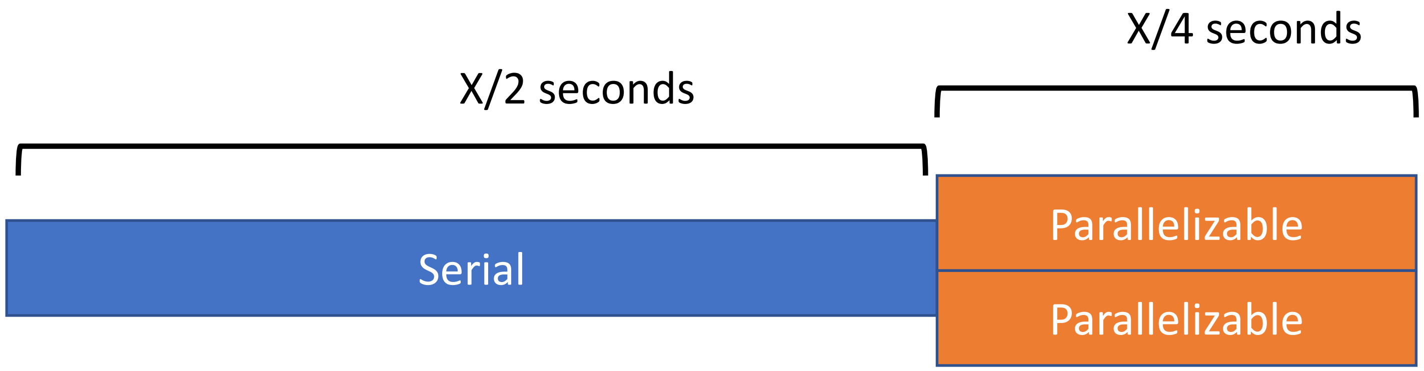




Amdahl’s law

2 CPUs





X/4 seconds

X/2 seconds

Serial

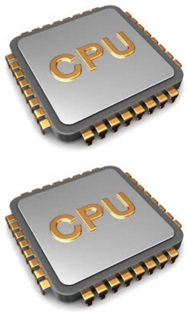
Parallelizable

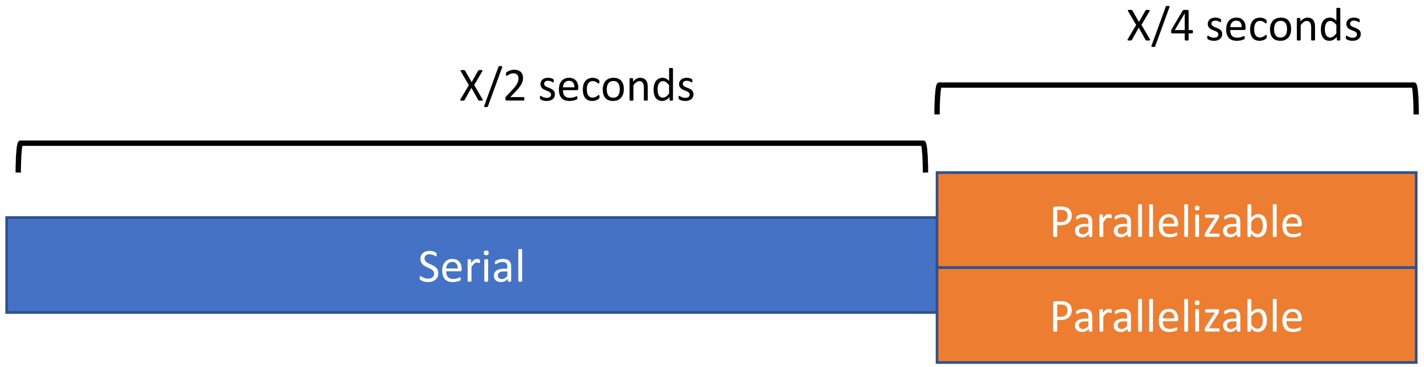
Parallelizable

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

Amdahl’s law

2 CPUs





X/2 seconds

X/4 seconds

Serial

Parallelizable

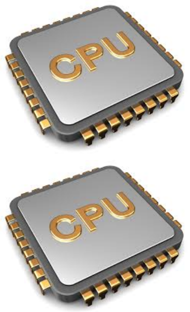
Parallelizable

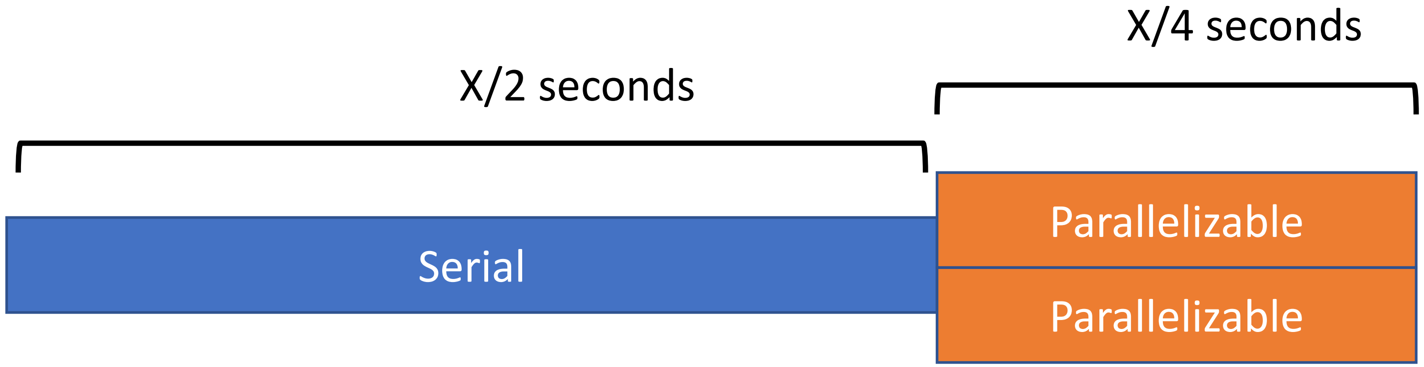
End to end time: (X/2 + X/4) = (3/4)X seconds

What is the “speedup” in this case?

Amdahl’s law

2 CPUs





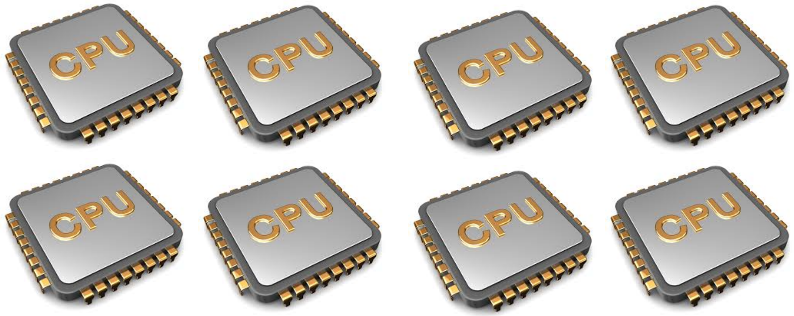
End to end time: (X/2 + X/4) = (3/4)X seconds

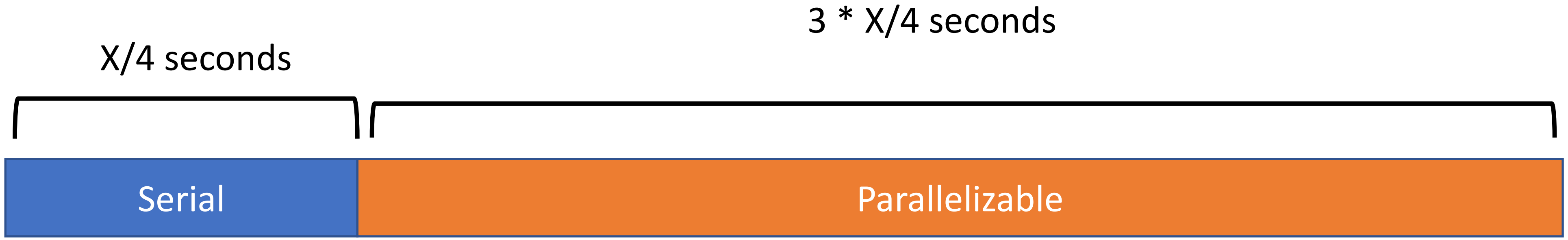
What is the “speedup” in this case?

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

Speedup exercise

8 CPUs





3 * X/4 seconds

X/4 seconds

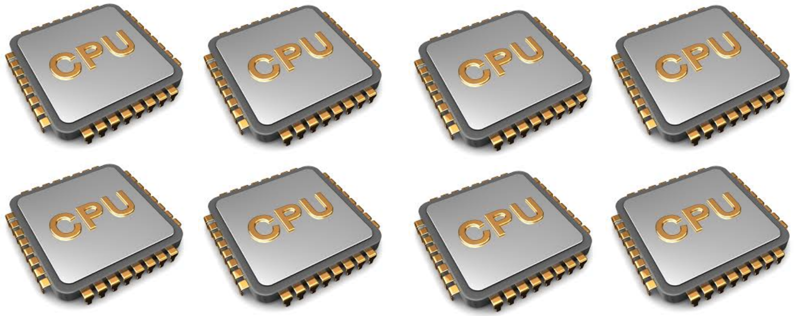
Serial

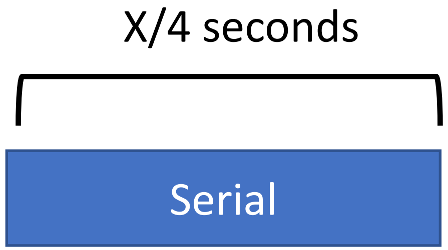
Parallelizable

End to end time: X seconds

Speedup exercise

8 CPUs





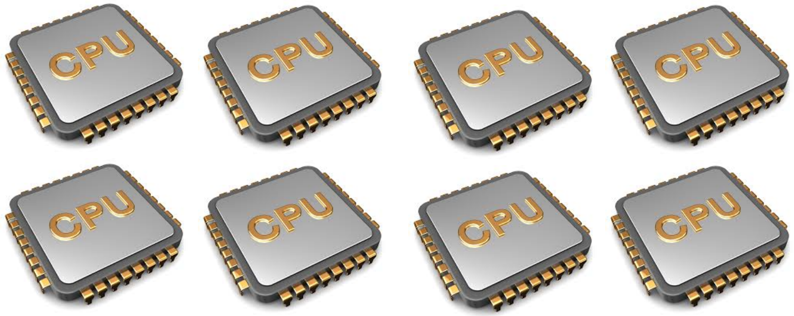
X/4 seconds

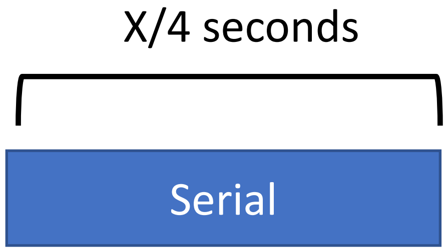
Serial

End to end time: X seconds

Speedup exercise

8 CPUs

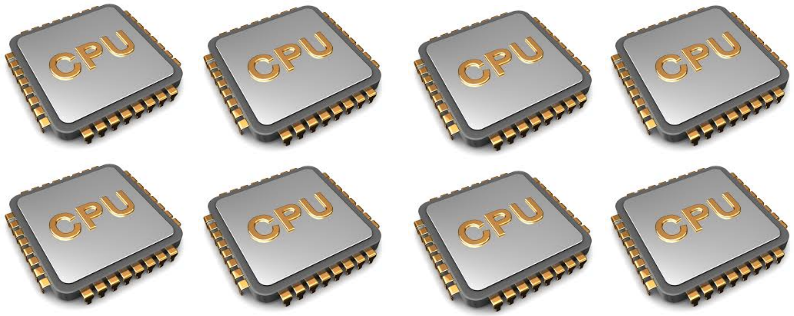




X/4 seconds

Serial

Speedup exercise



8 CPUs

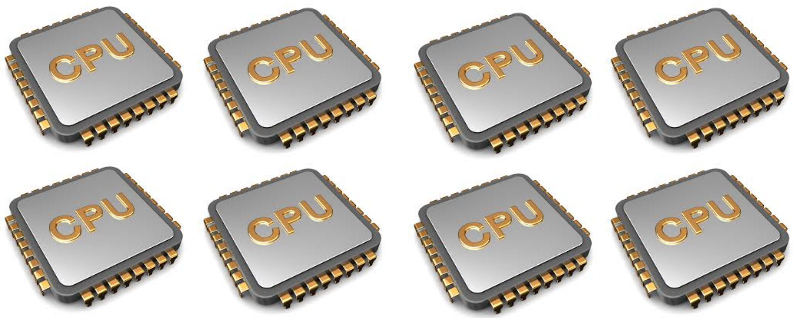
X/4 seconds

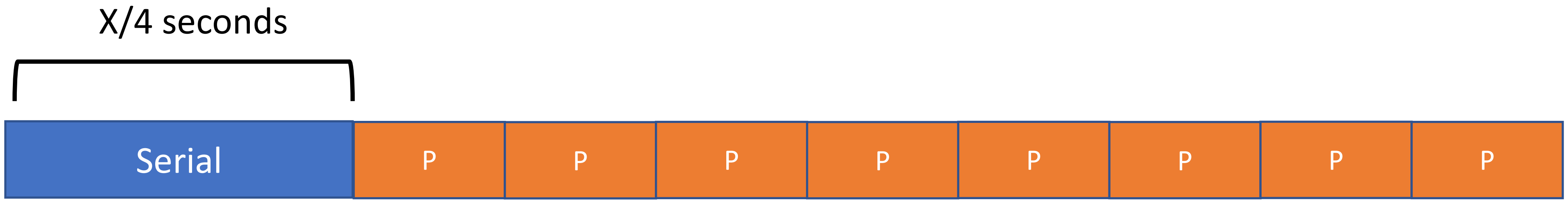
Serial

What is the “speedup” in this case?

Speedup exercise

8 CPUs




X/4 seconds

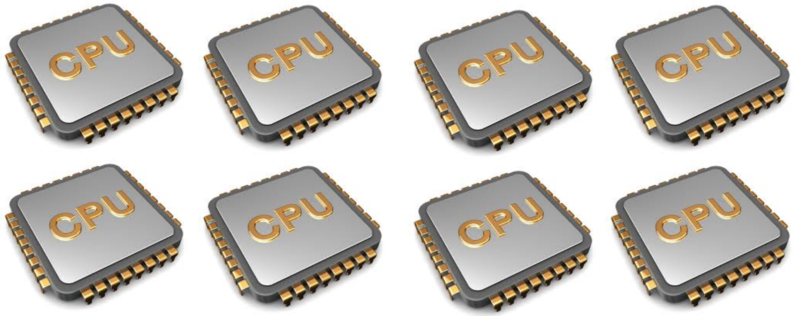
Serial

P P P P P P P P

What is the “speedup” in this case?

Speedup exercise

8 CPUs



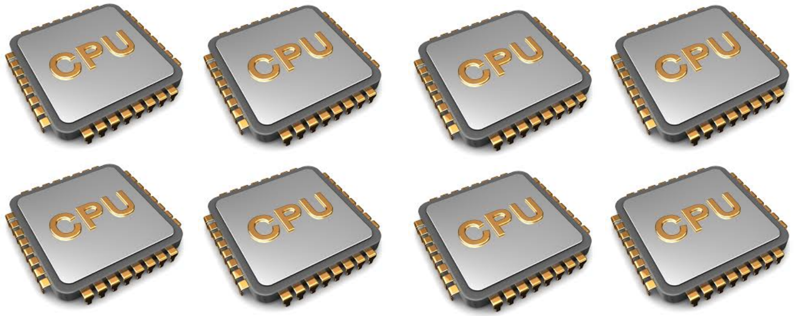
X/4 seconds

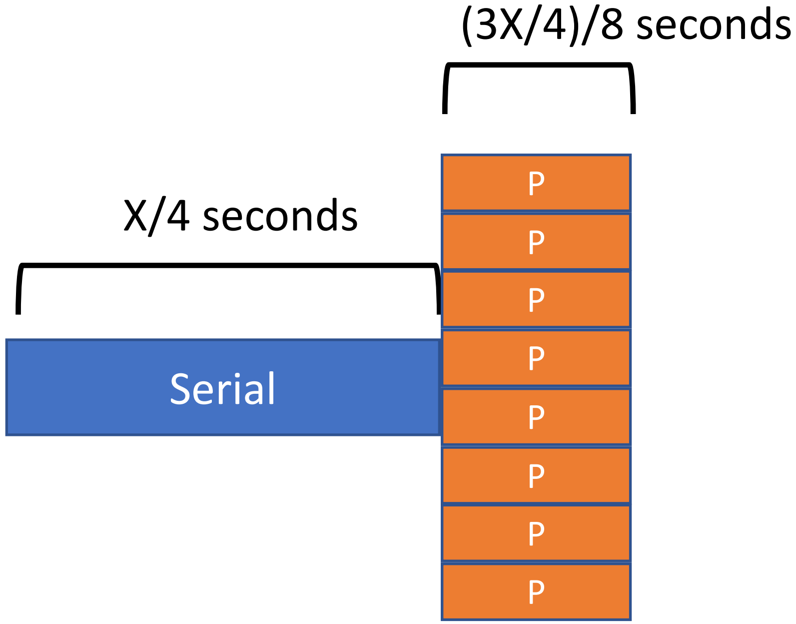
Serial

What is the “speedup” in this case?

Speedup exercise

8 CPUs

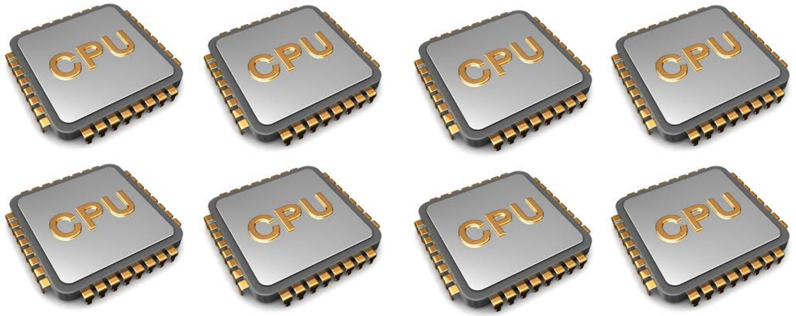


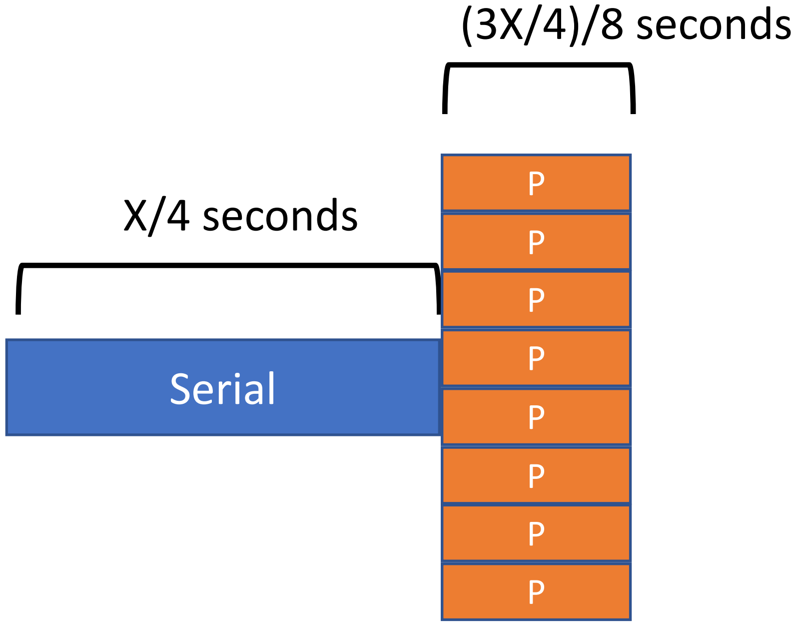


What is the “speedup” in this case?

Speedup exercise

8 CPUs

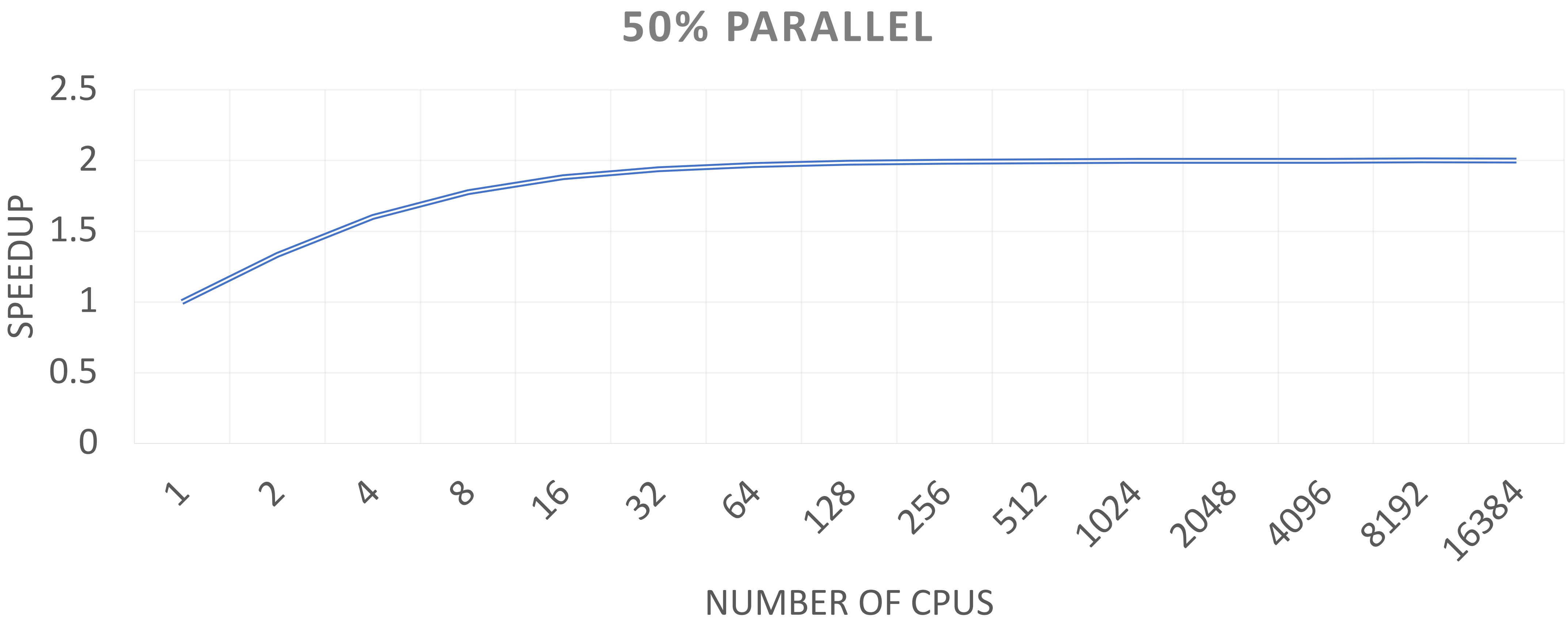




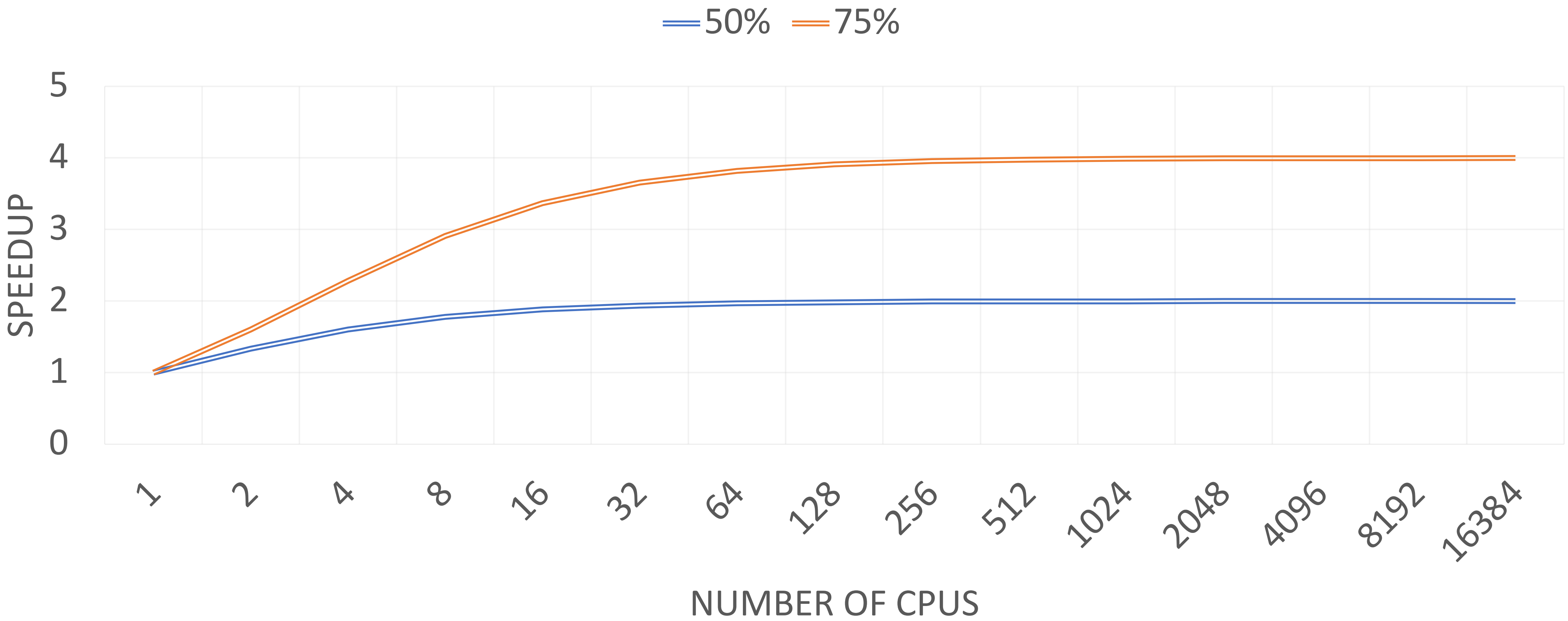
What is the "speedup" in this case?

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

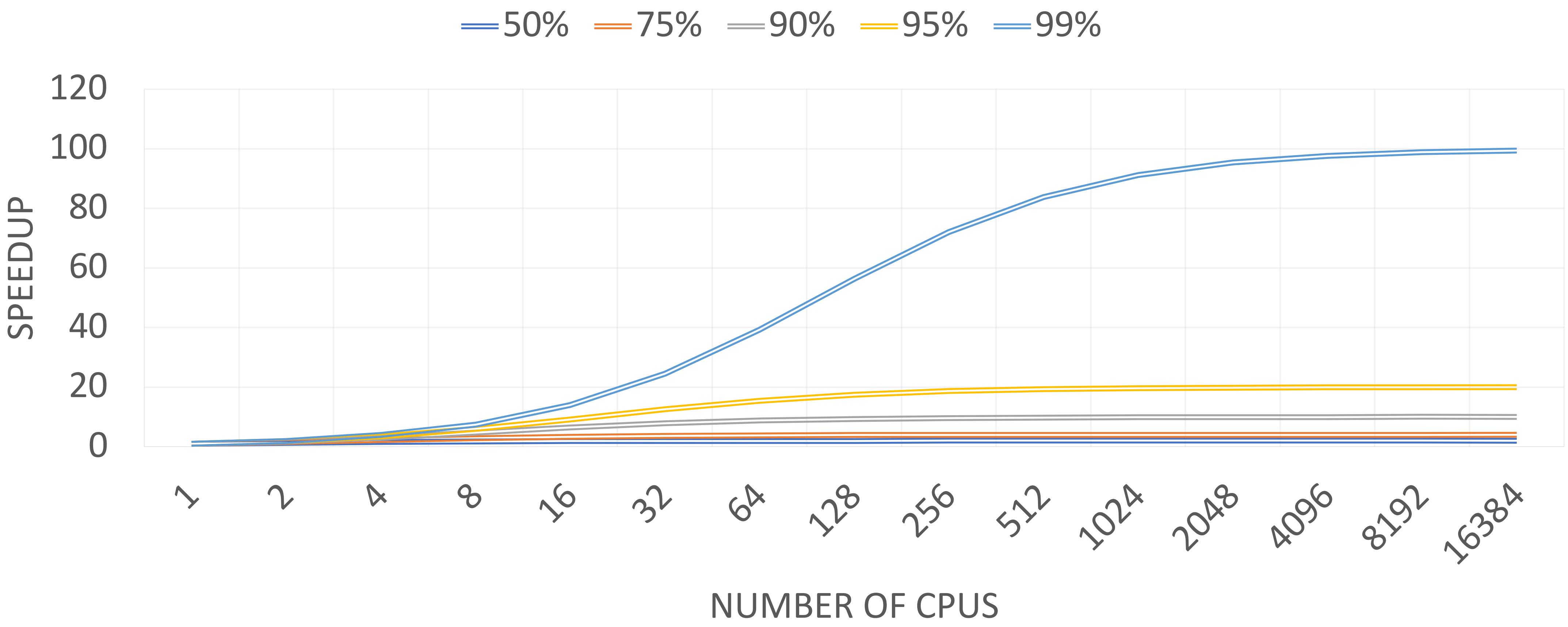
Amdahl Action Zone



Amdahl Action Zone



Amdahl Action Zone



Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson

Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson

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 given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and #CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

Strong Scaling vs Weak Scaling

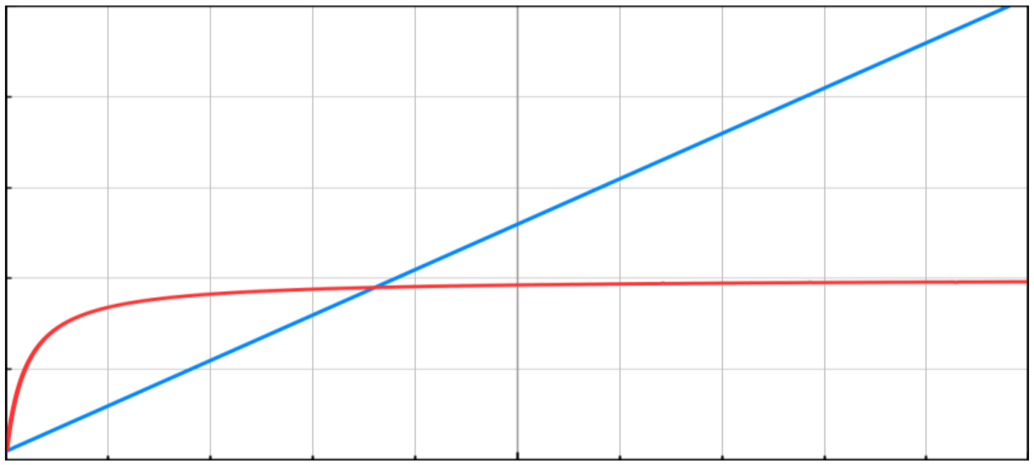
Amdahl vs. Gustafson

- $N = \#CPUs,\ S = serial\ portion = 1 - A$
- Amdahl's law: $Speedup(N) = \frac{1}{\frac{A}{N} + S}$
  - **Strong scaling:** $Speedup(N)$ calculated given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and #CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

- Gustafson's law: *Speedup(N)* = S + (S-1)*N
  - **Weak scaling:** *Speedup(N) calculated given that work per CPU is fixed*
  - Work/CPU fixed when adding more CPUs keeps granularity fixed
  - Problem size grows: solve larger problems
  - **Consequence:** speedup upper bound is much higher

Strong Scaling vs Weak Scaling

Amdahl vs. Gustafson



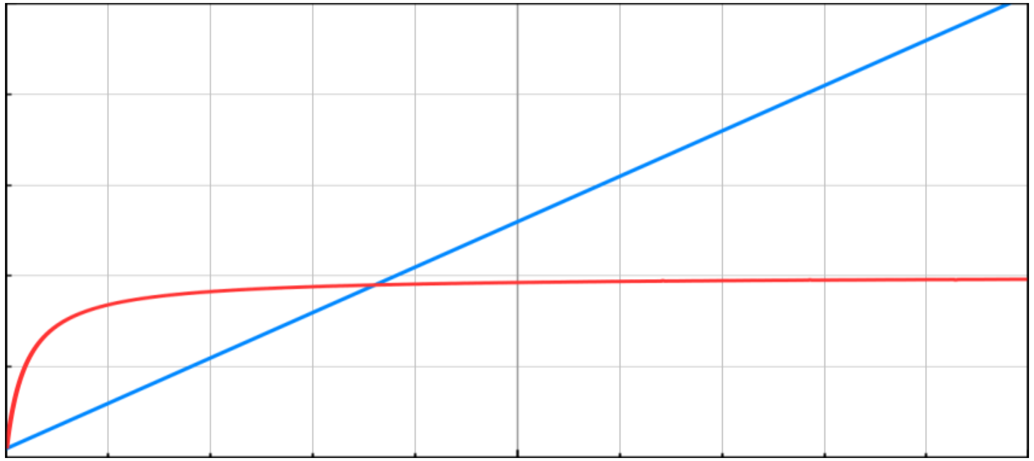
- $N = \#CPUs,\ S = serial\ portion = 1 - A$
- Amdahl's law: $Speedup(N) = \frac{1}{\frac{A}{N}+S}$
  - **Strong scaling:** $Speedup(N)$ calculated given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and #CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

- Gustafson's law: *Speedup(N)* = S + (S-1)*N
  - **Weak scaling:** *Speedup(N) calculated given that work per CPU is fixed*
  - Work/CPU fixed when adding more CPUs keeps granularity fixed
  - Problem size grows: solve larger problems
  - **Consequence:** speedup upper bound is much higher

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 given total amount of work is fixed
  - Solve same problems faster when problem size is fixed and #CPU grows
  - Assuming parallel portion is fixed, speedup soon seizes to increase

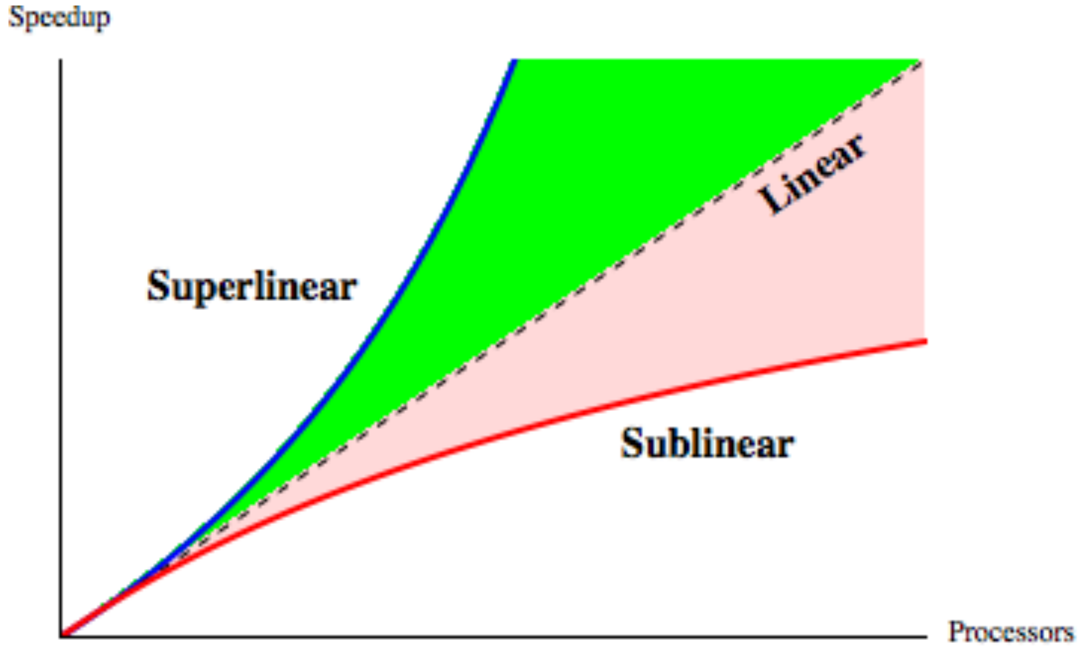


- Gustafson's law: *Speedup(N)* = S + (S-1)*N
  - **Weak scaling:** *Speedup(N) calculated given that work per CPU is fixed*
  - Work/CPU fixed when adding more CPUs keeps granularity fixed
  - Problem size grows: solve larger problems
  - **Consequence:** speedup upper bound is much higher

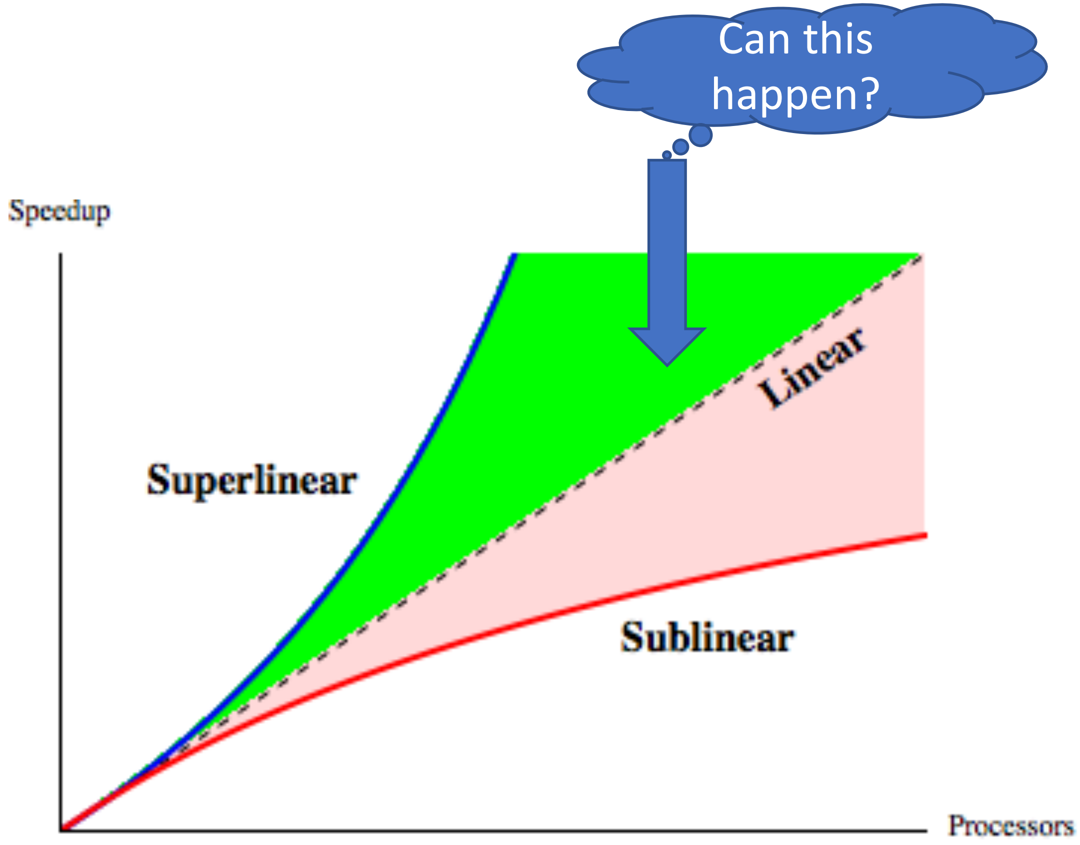
When is Gustavson's law a better metric?
When is Amdahl's law a better metric?

Super-linear speedup

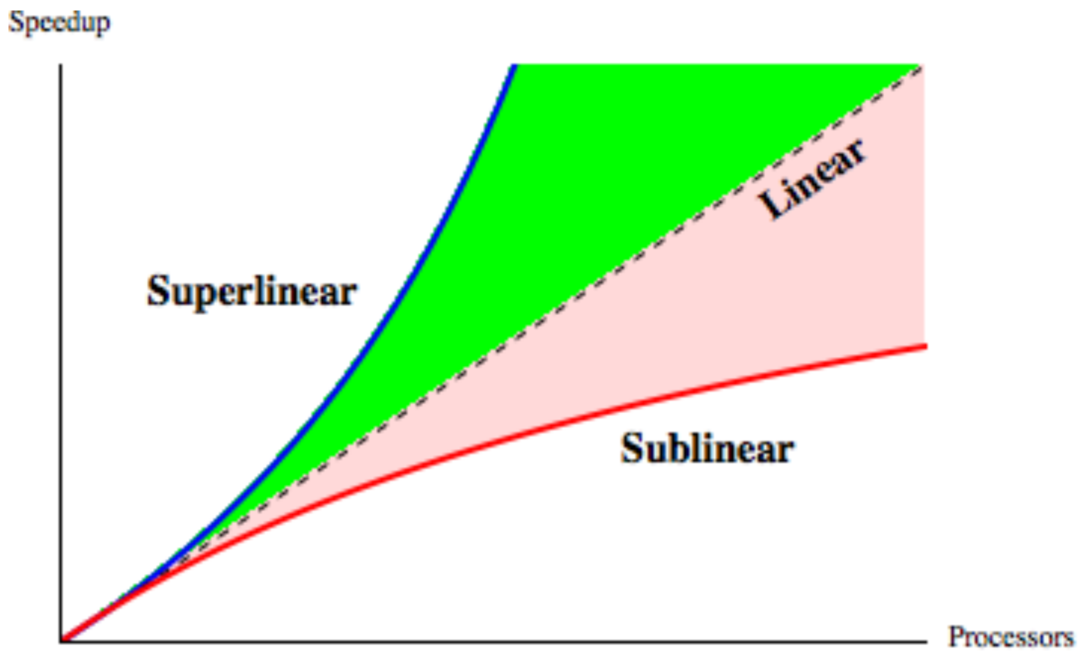
Super-linear speedup



Super-linear speedup

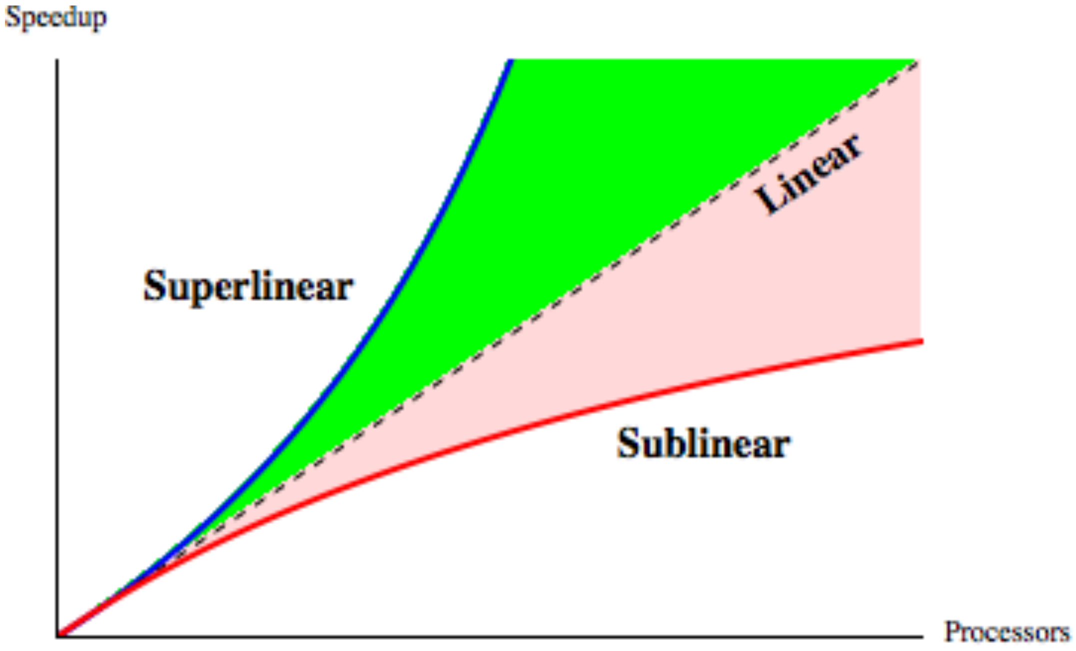


Super-linear speedup



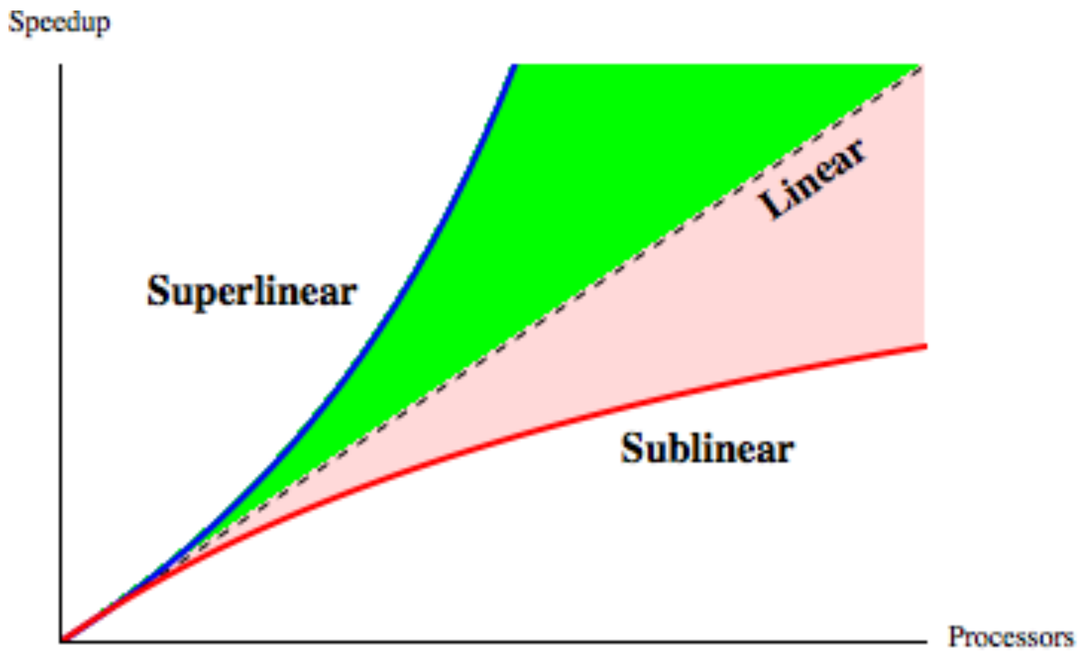
Super-linear speedup

- Possible due to cache



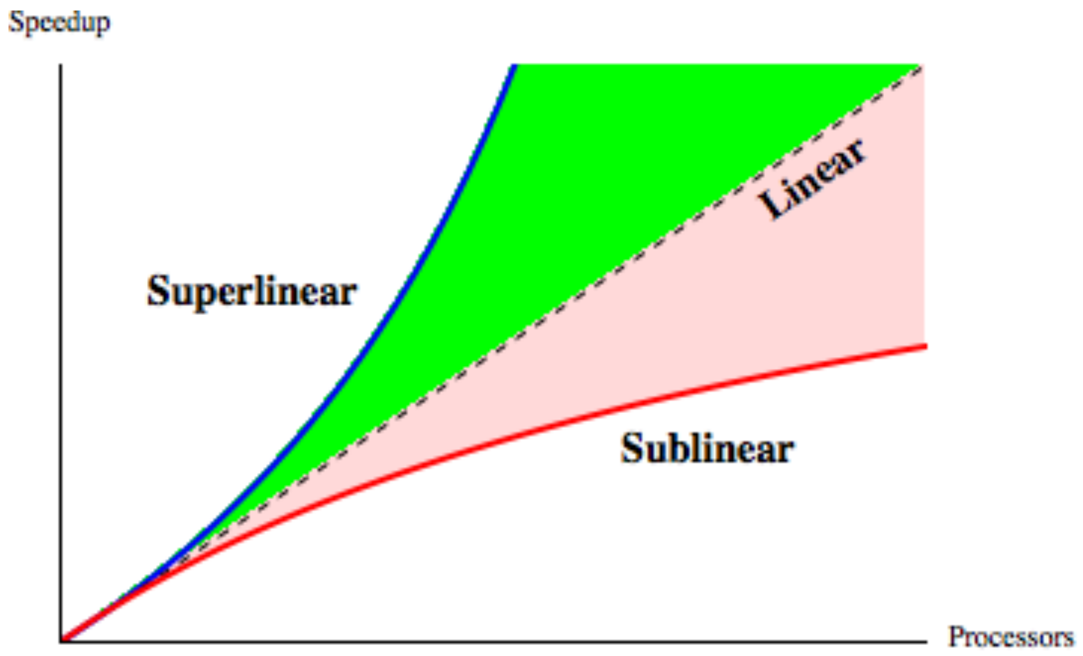
Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm



Super-linear speedup

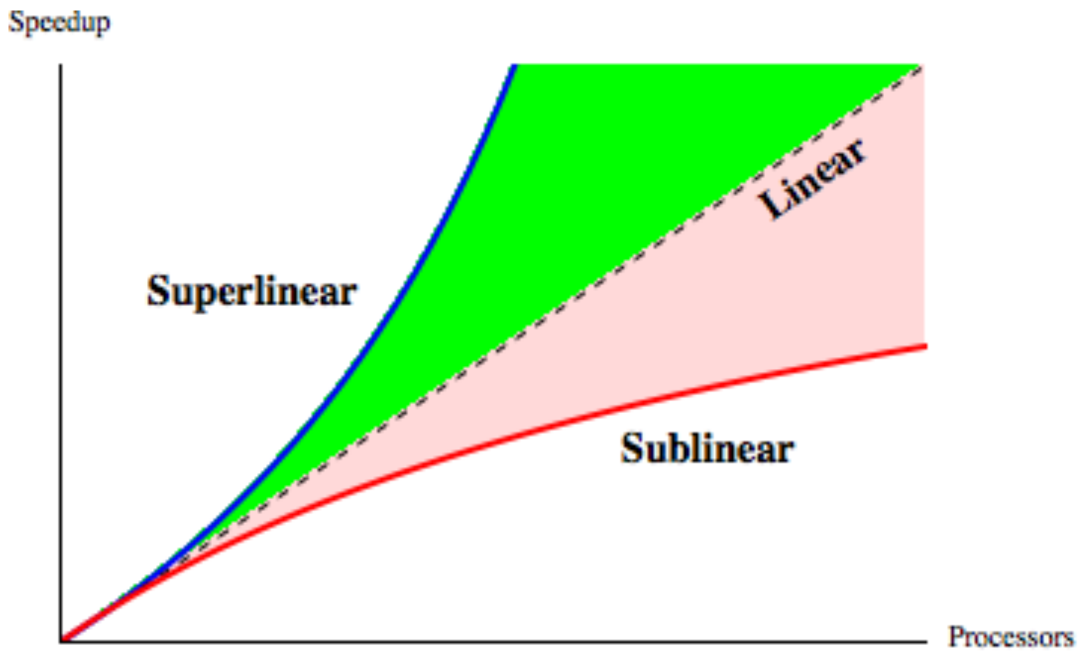
- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

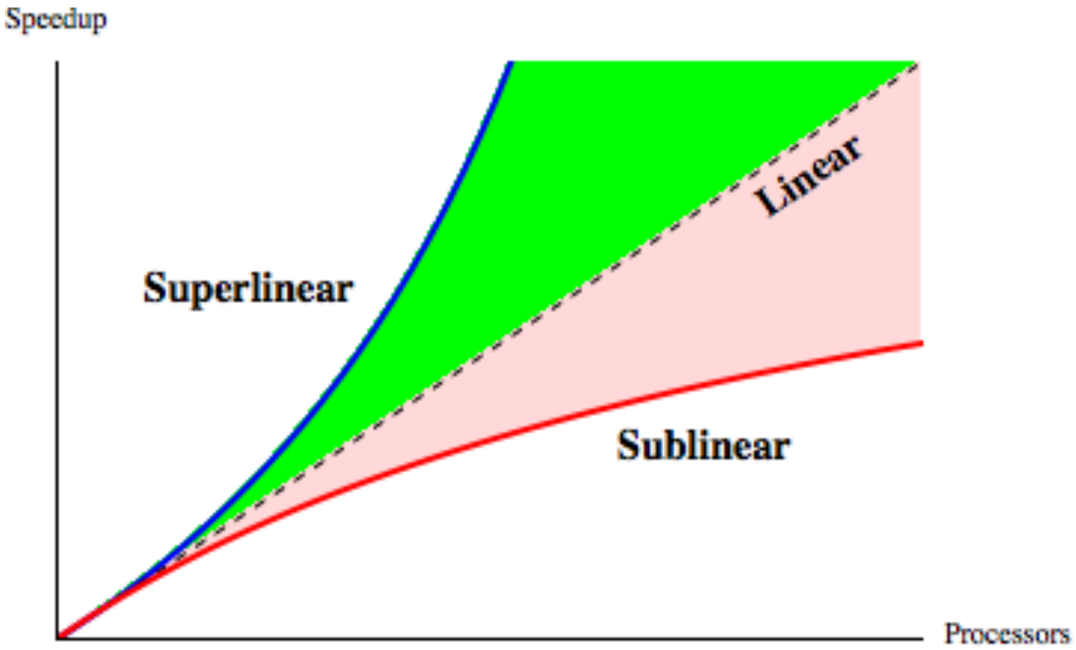
Efficient **bubble sort**



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

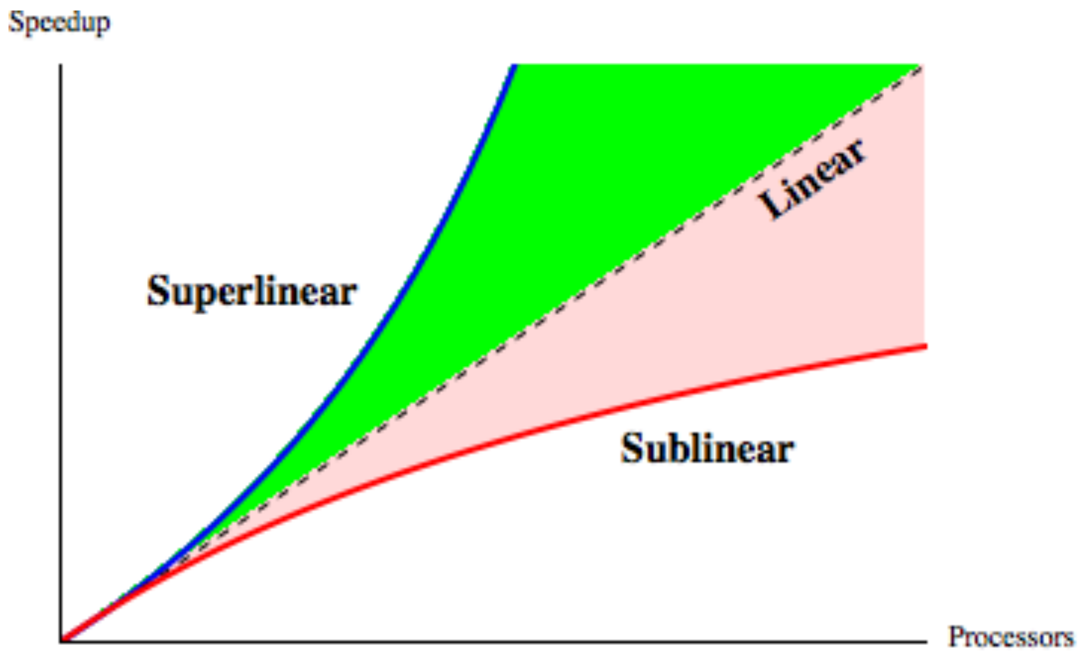
Efficient **bubble sort**
•*Serial: 150s*



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

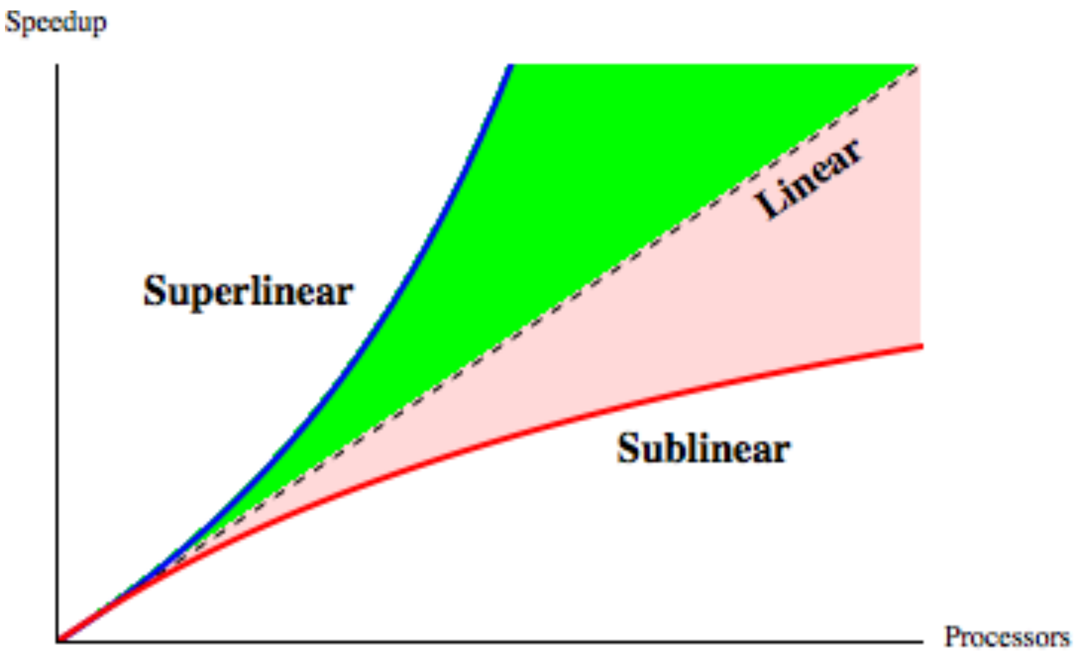
Efficient **bubble sort**
•*Serial: 150s*
•*Parallel 40s*



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

Efficient **bubble sort**
•*Serial: 150s*
•*Parallel 40s*
•*Speedup:*

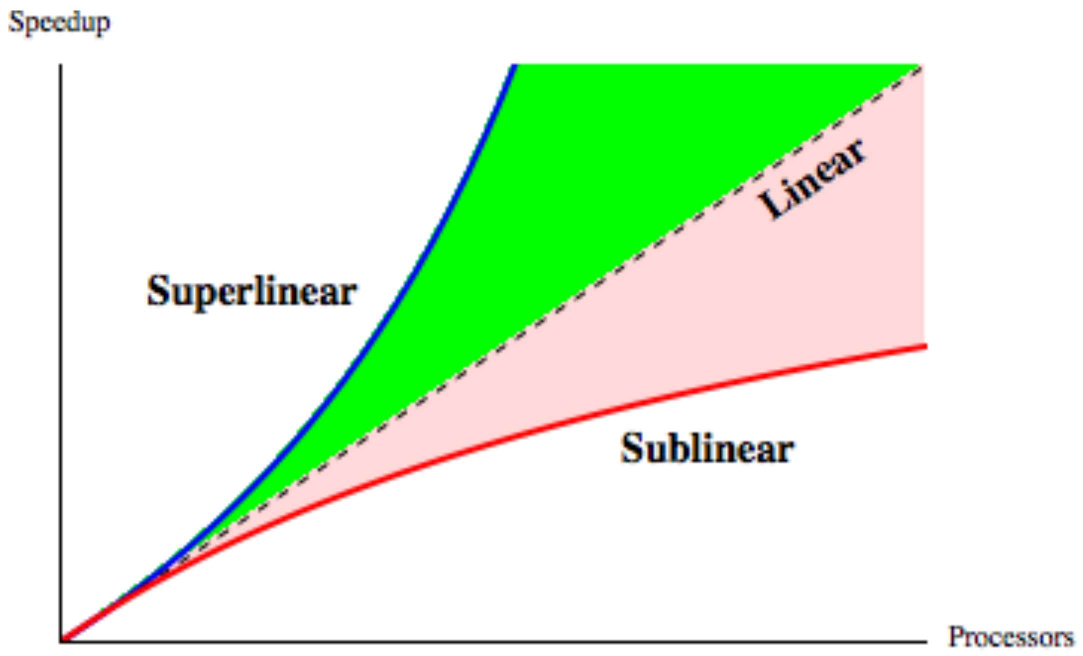


Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

Efficient **bubble sort**
•*Serial: 150s*
•*Parallel 40s*
•*Speedup:*

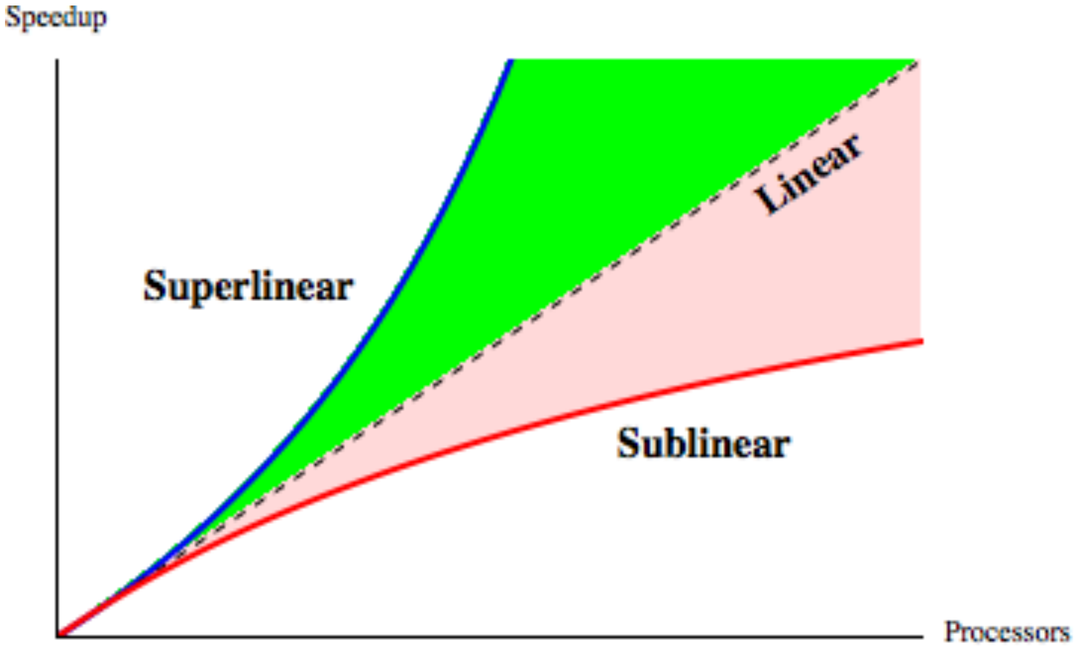
$$\frac{150}{40} = 3.75\ ?$$



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

Efficient **bubble sort**
•*Serial: 150s*
•*Parallel 40s*
•*Speedup:*
NO NO NO! $\frac{150}{40} = 3.75$?



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best** serial algorithm*
- Example:

Efficient **bubble sort**

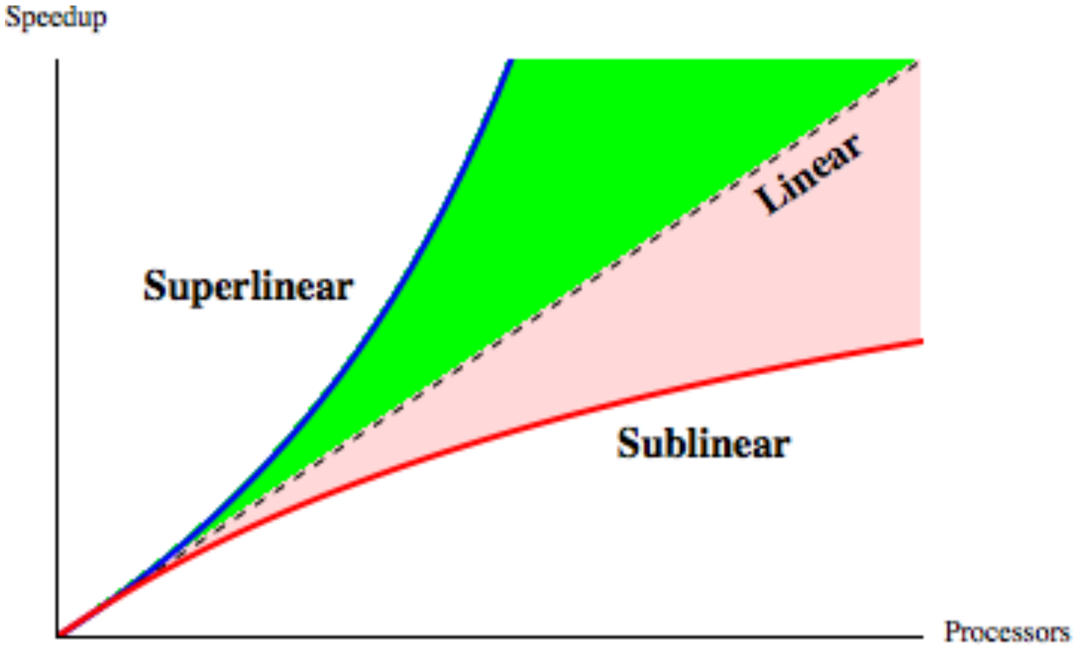
•*Serial: 150s*

•*Parallel 40s*

•*Speedup:*

NO NO NO! $\frac{150}{40} = 3.75$?

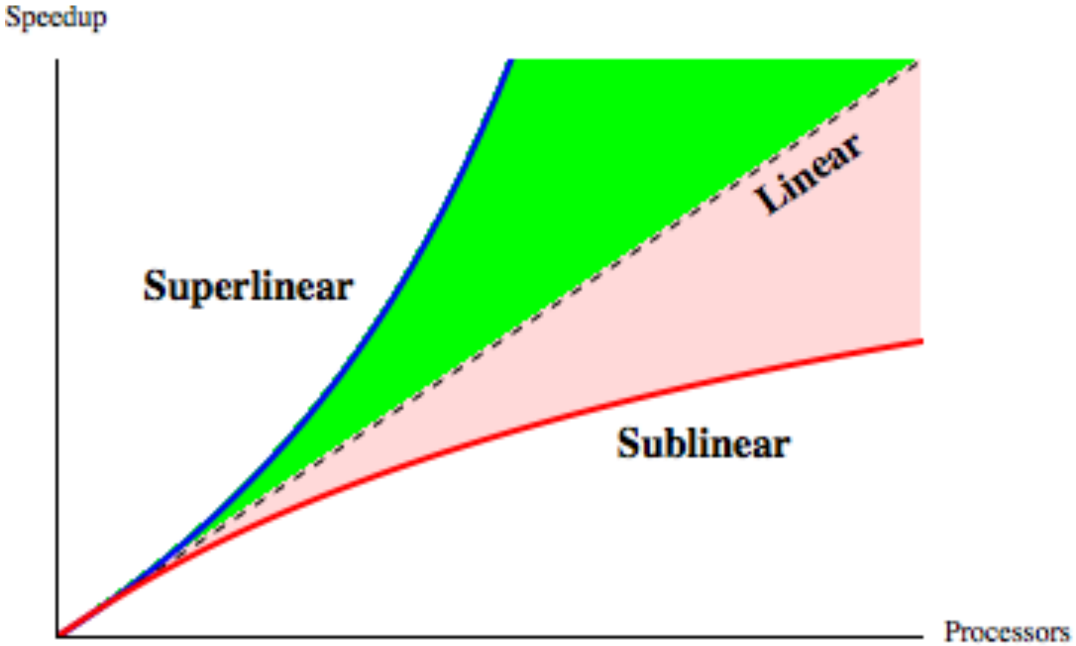
•*Serial quicksort: 30s*



Super-linear speedup

- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

Efficient **bubble sort**
•*Serial: 150s*
•*Parallel 40s*
•*Speedup:*
NO NO NO! $\frac{150}{40} = 3.75$?
•*Serial quicksort: 30s*
•*Speedup = 30/40 = 0.75X*



Super-linear speedup

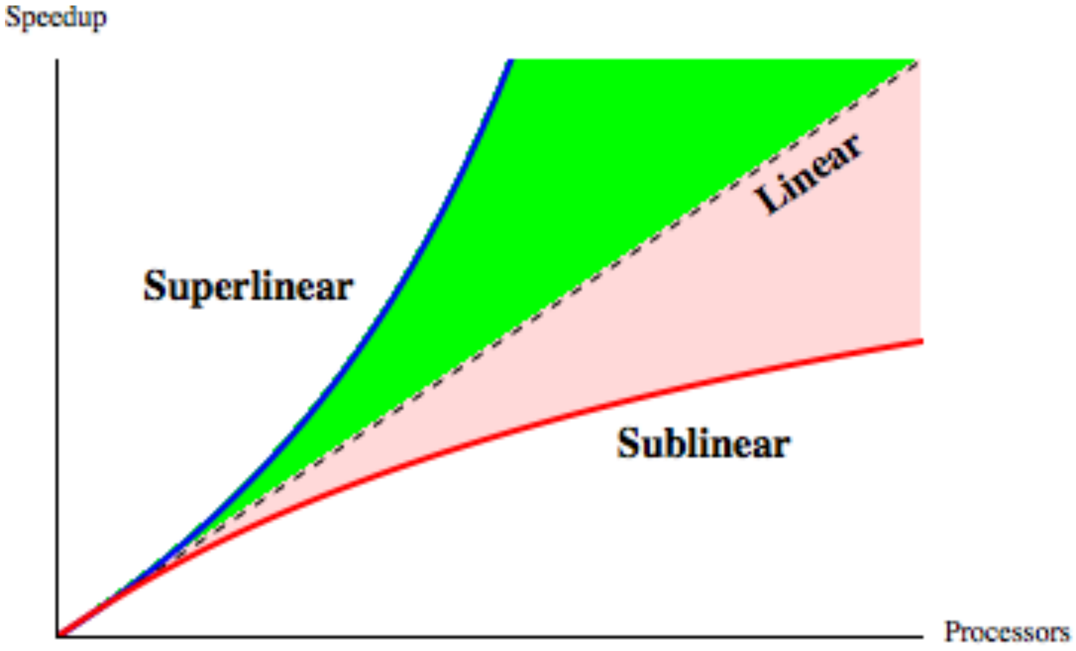
- Possible due to cache
- But usually just poor methodology
- Baseline: ***best*** serial algorithm
- Example:

Efficient **bubble sort**

- *Serial: 150s*
- *Parallel 40s*
- *Speedup:*

NO NO NO! $\frac{150}{40} = 3.75$?

- *Serial quicksort: 30s*
- *Speedup = 30/40 = 0.75X*



Why insist on best serial algorithm as baseline?

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;
```

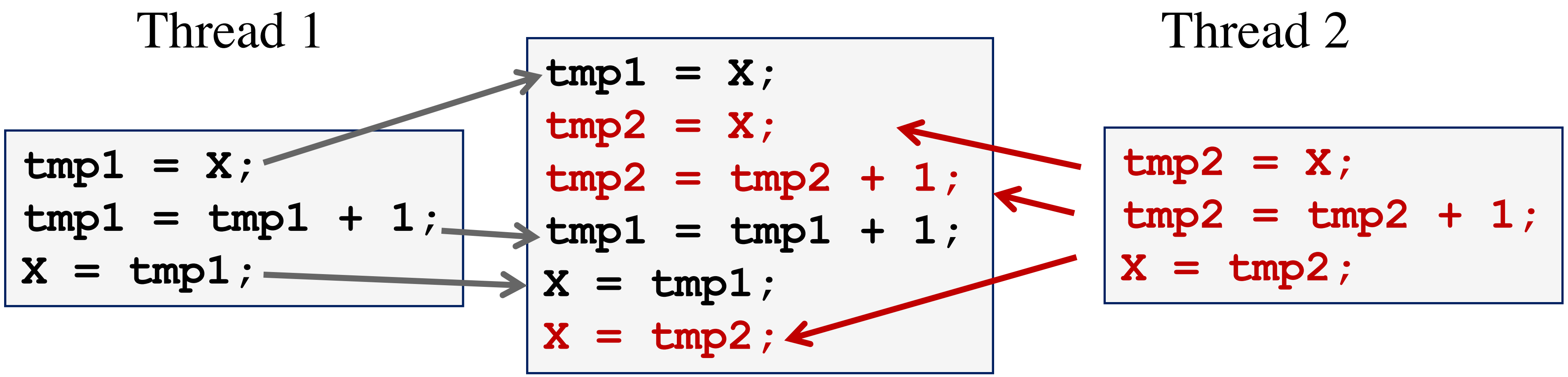
Thread 2

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

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;
```

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

Thread 2

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

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?

Why are Locks “Hard?”

Why are Locks “Hard?”

- Coarse-grain locks

Why are Locks “Hard?”

- Coarse-grain locks

- Fine-grain locks

Why are Locks “Hard?”

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

- Fine-grain locks

Why are Locks “Hard?”

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

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

Why are Locks “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?

Why are Locks “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);`

Why are Locks "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

```
move(a, b, key1);
```

Thread 1

```
move(b, a, key2);
```

DEADLOCK!

Review: correctness conditions

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

Review: correctness conditions

- Safety
    - Only one thread in the critical region

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

Review: 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

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

Review: 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.

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

Review: 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*

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

Review: 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

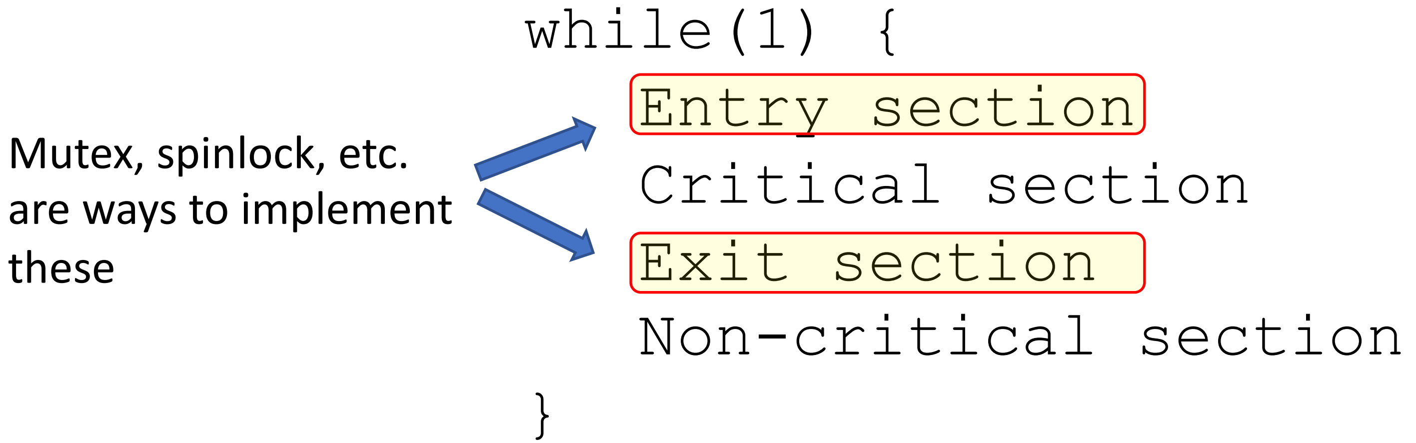
https://www.cs.cornell.edu/fbs/publications/defliveness.pdf

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

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



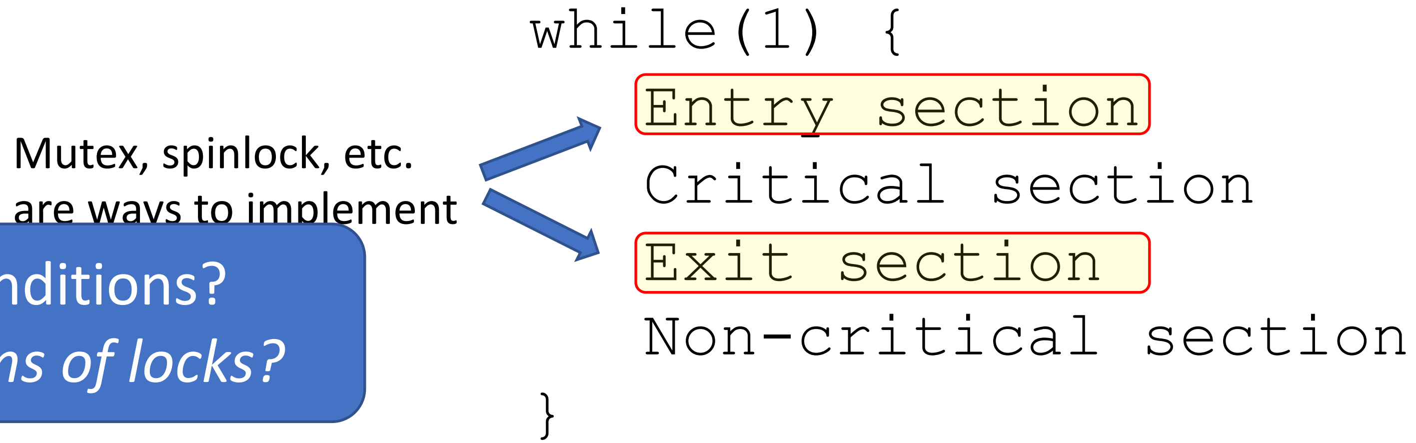
Mutex, spinlock, etc. are ways to implement these

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

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



Mutex, spinlock, etc. are ways to implement

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

Did we get all the important conditions?
Why is correctness defined in terms of locks?

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

```
Lock::Release() {
    *lock = 0;
}
```

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

```
Lock::Release() {
    *lock = 0;
}
```

What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work

Implementing Locks

```
int lock_value = 0;
int* lock = &lock_value;
```

```
Lock::Acquire() {
  while (*lock == 1)
     ; //spin
  *lock = 1;
}
```

```
Lock::Release() {
    *lock = 0;
}
```

Completely and utterly broken.
How can we fix it?

What are the problem(s) with this?

- A. CPU usage
- B. Memory usage
- C. Lock::Acquire() latency
- D. Memory bus usage
- E. Does not work