# AMATH 483/583
# High Performance Scientific Computing

## Lecture 11:
## Shared Memory Parallelism, Correctness, Performance

Xu Tony Liu, PhD

Paul G. Allen School of Computer Science & Engineering

University of Washington

Seattle, WA

# Overview

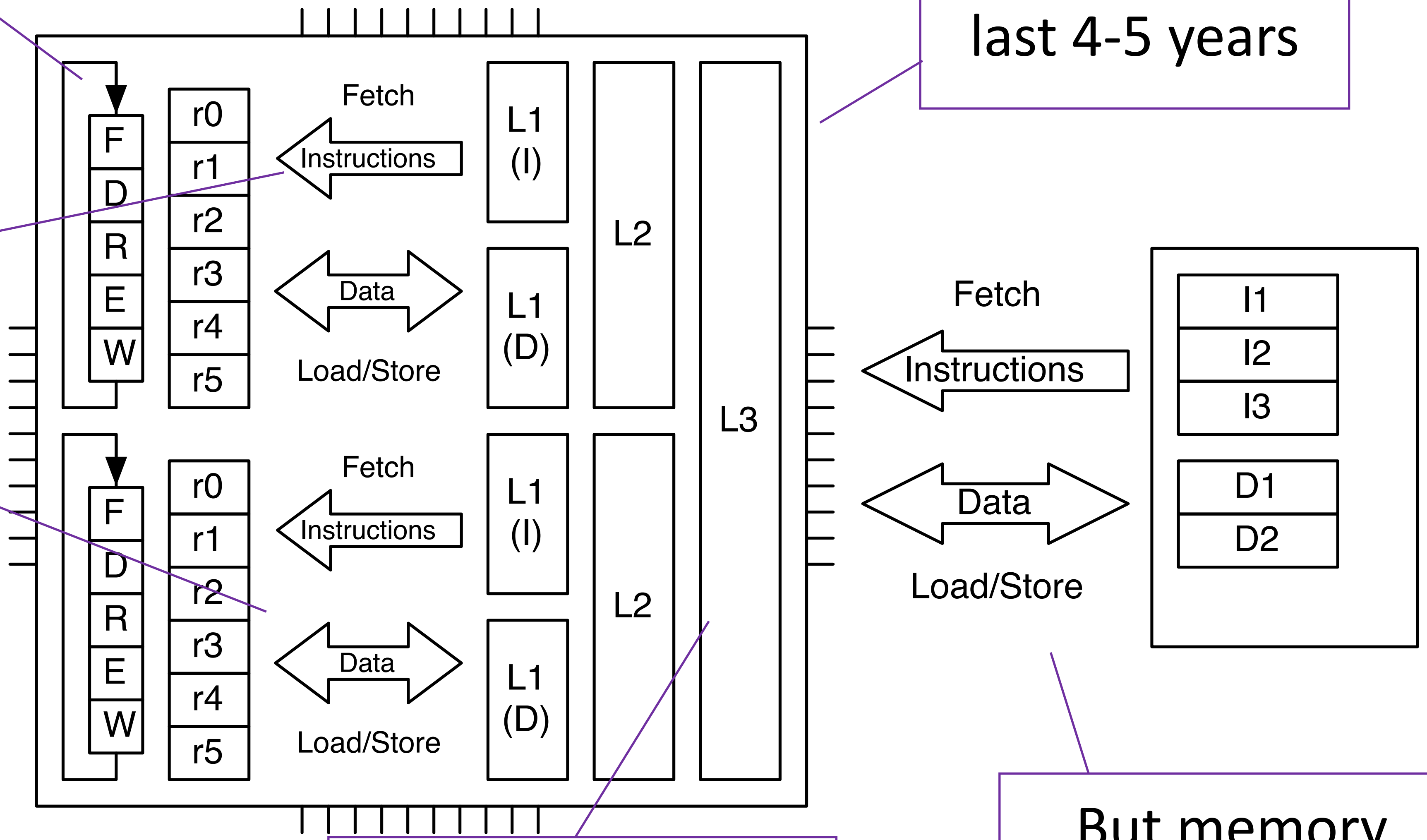
- Review
  - Multiple cores
  - Processes / threads as resource / computation abstraction
  - Parallelization strategies for multiple computations
- Correctness
  - Race condition
  - The critical-section problem
- Performance
  - Amdahl’s law
  - Gustaffson’s law

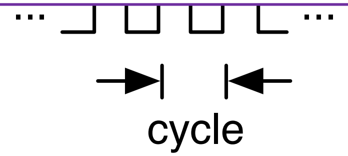
# Multicore Architecture

Core is a FDREW + regs

Each runs its own sequence of instructions

Each can access its own data



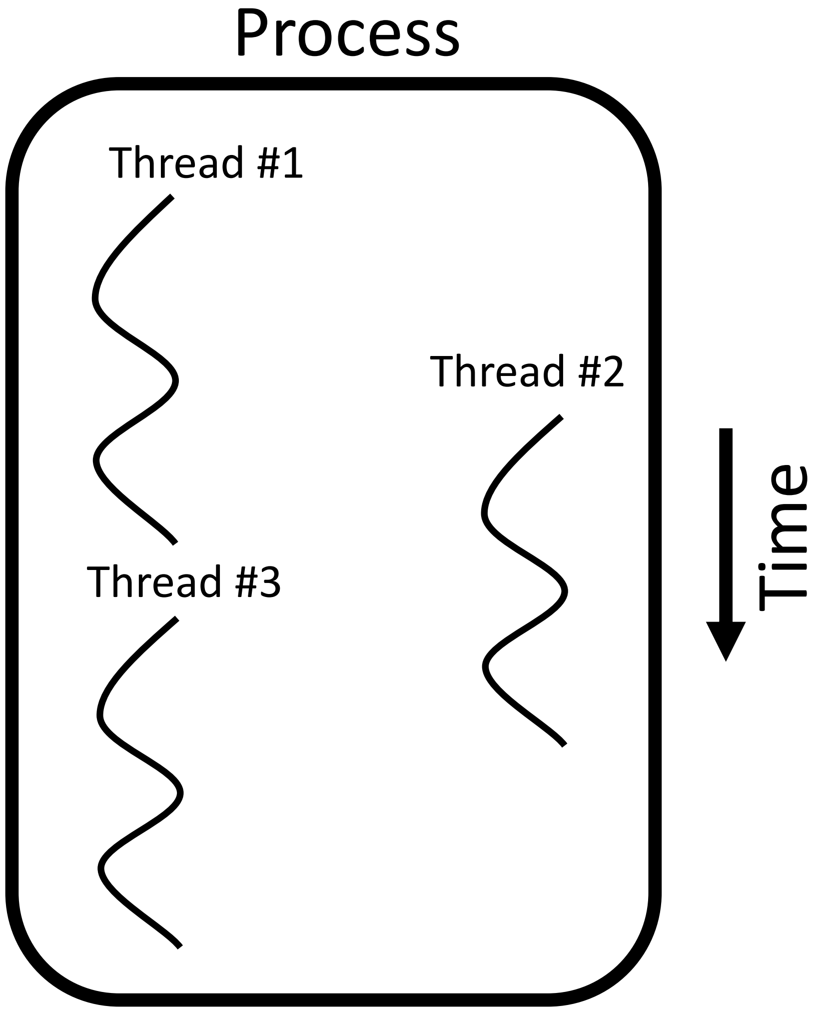

Any CPU in the last 4-5 years

Each has memory hierarchy

But memory might be shared

# Processes and Threads

- A process is an abstraction for a collection of resources to represent a (running) program
  - CPU
  - Memory
  - Address space
- A thread is an abstraction of execution (using the resources within a process)
  - Can share an address space



# Process Abstraction

Stored in Process Control Block (PCB

Set of information about process resources

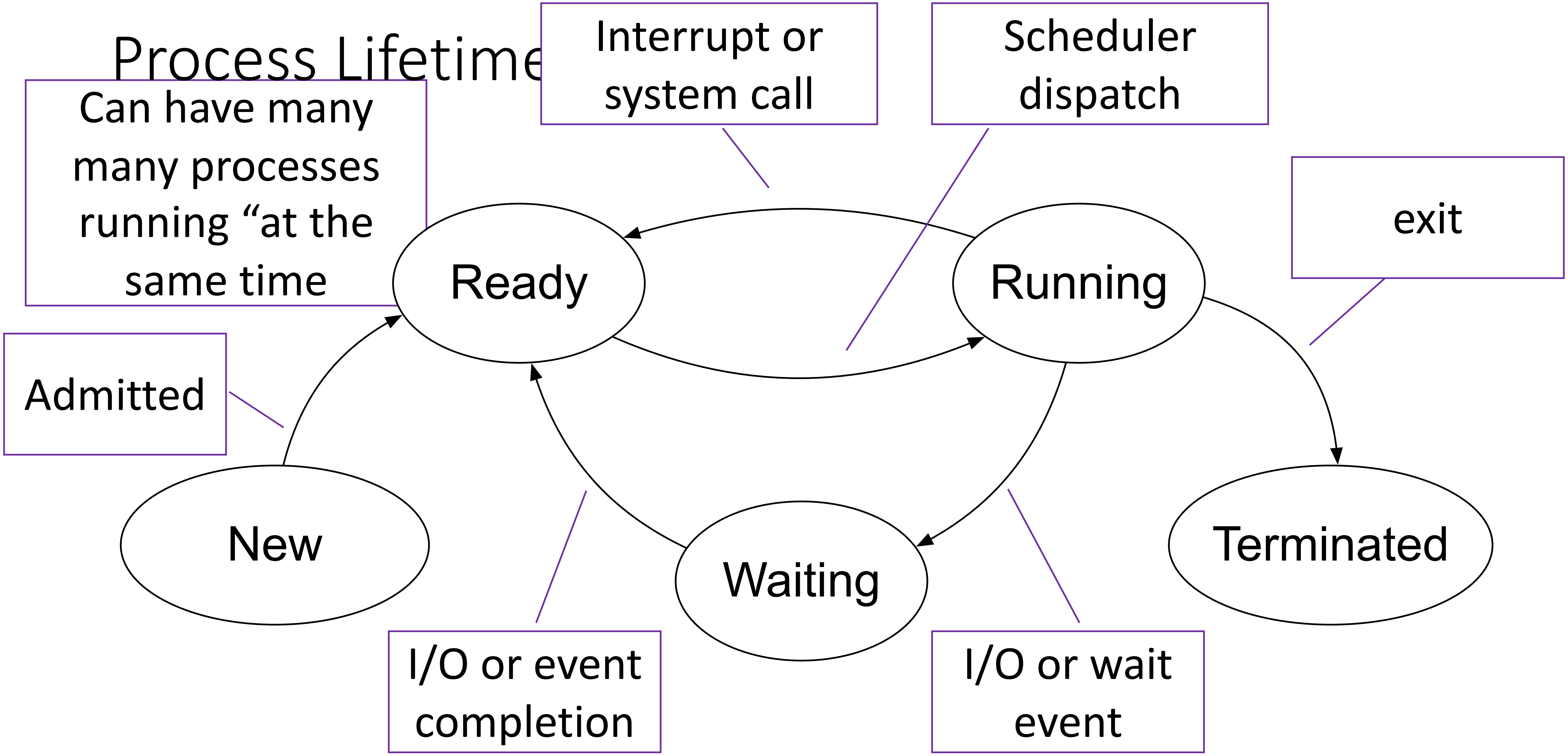
Sufficient to be able to start a process after stopped

Also for accounting / administrative purposes

| **Process management** | **Memory management** | **File management** |
| --- | --- | --- |
| Registers | Pointer to text segment | Root directory |
| Program counter | Pointer to data segment | Working directory |
| Program status word | Pointer to stack segment | File descriptors |
| Stack pointer | | User ID |
| Process state | | Group ID |
| Priority | | |
| Scheduling parameters | | |
| Process ID | | |
| Parent process | | |
| Process group | | |
| Signals | | |
| Time when process started | | |
| CPU time used | | |
| Children's CPU time | | |
| Time of next alarm | | |

What does program counter represent?

# Process Lifetime



# Context Switch

P0 and P1 are running processes

What does this mean?

And this?

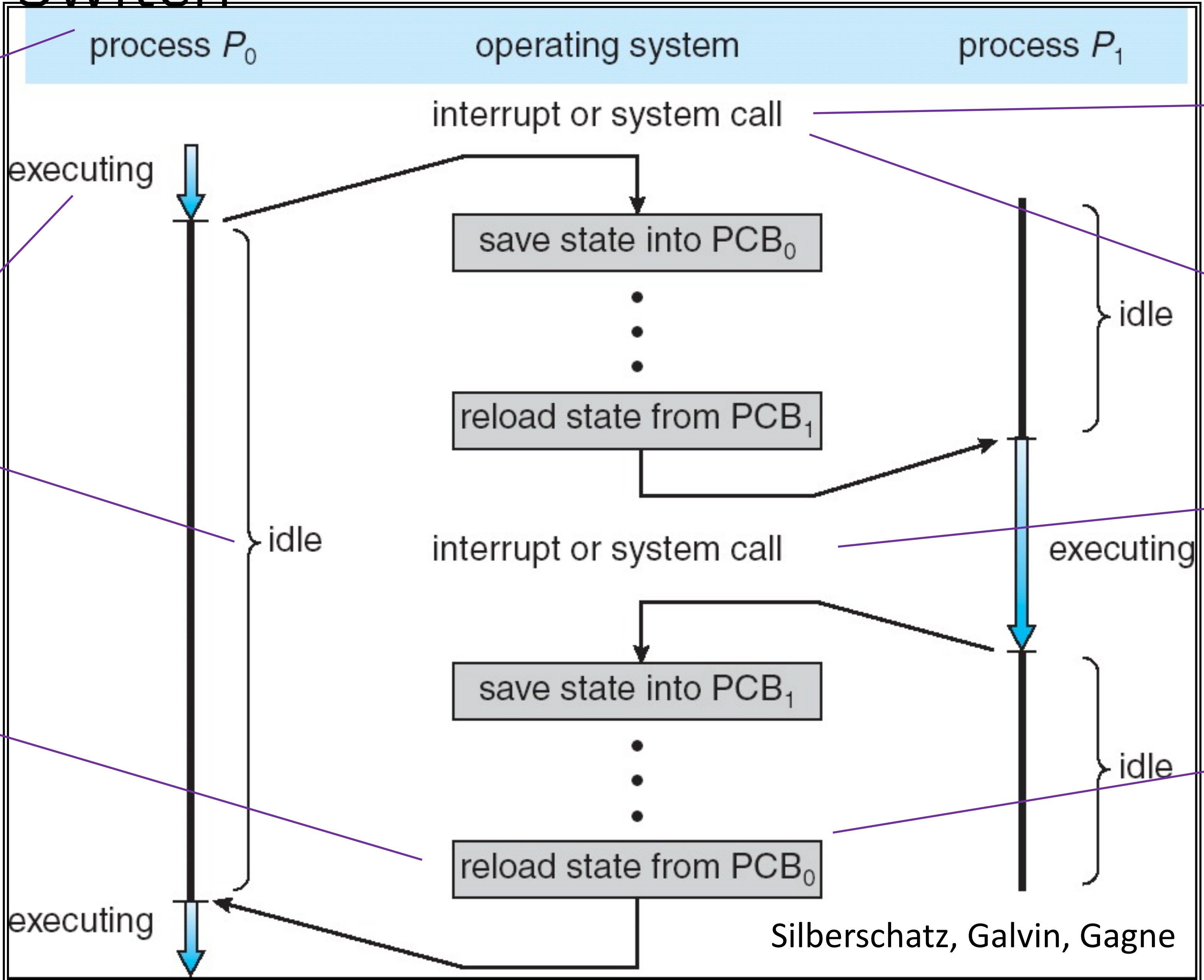
PCB = Process Control Block

External to OS

OS does not do this

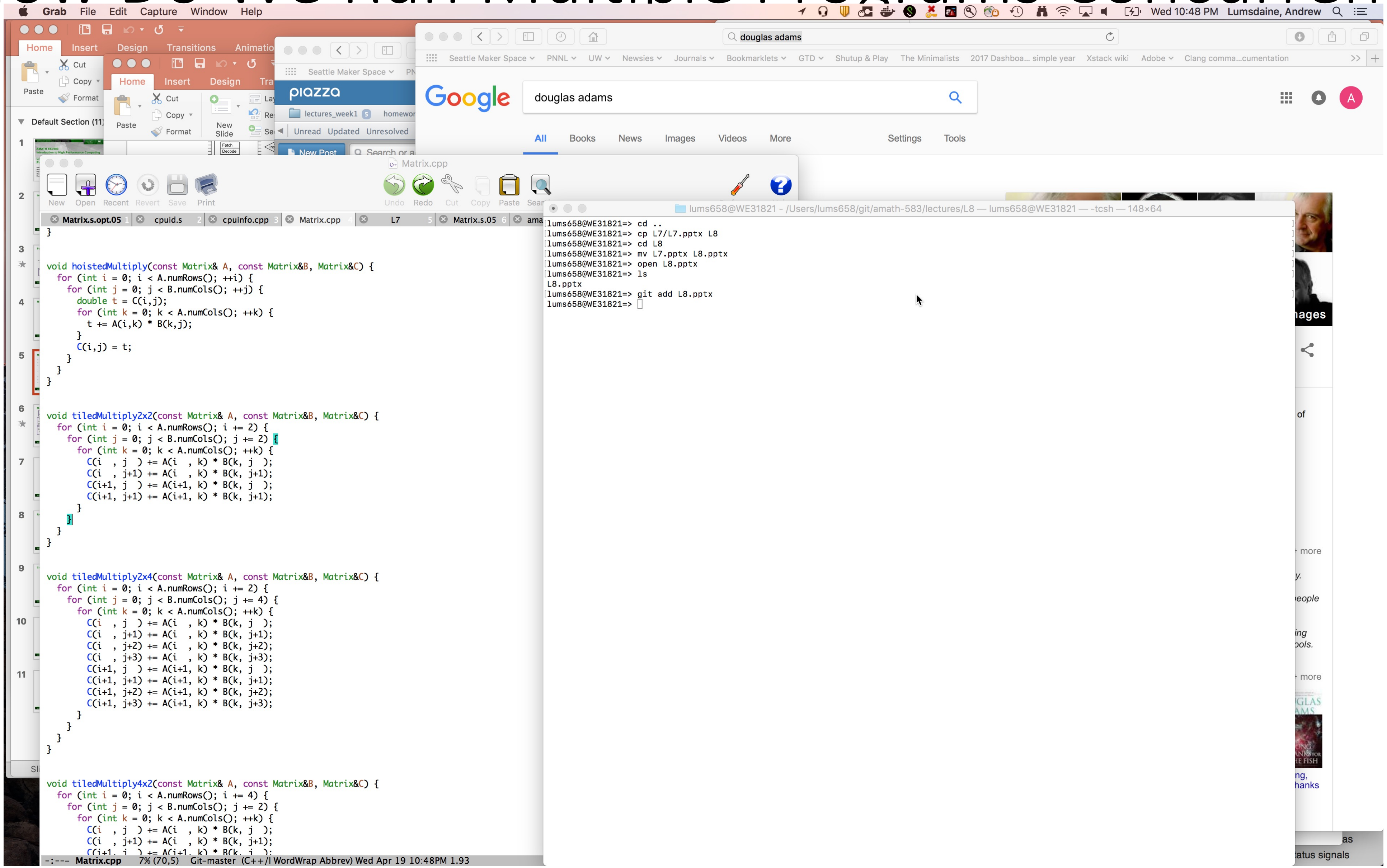
External to OS

Expensive!




Silberschatz, Galvin, Gagne

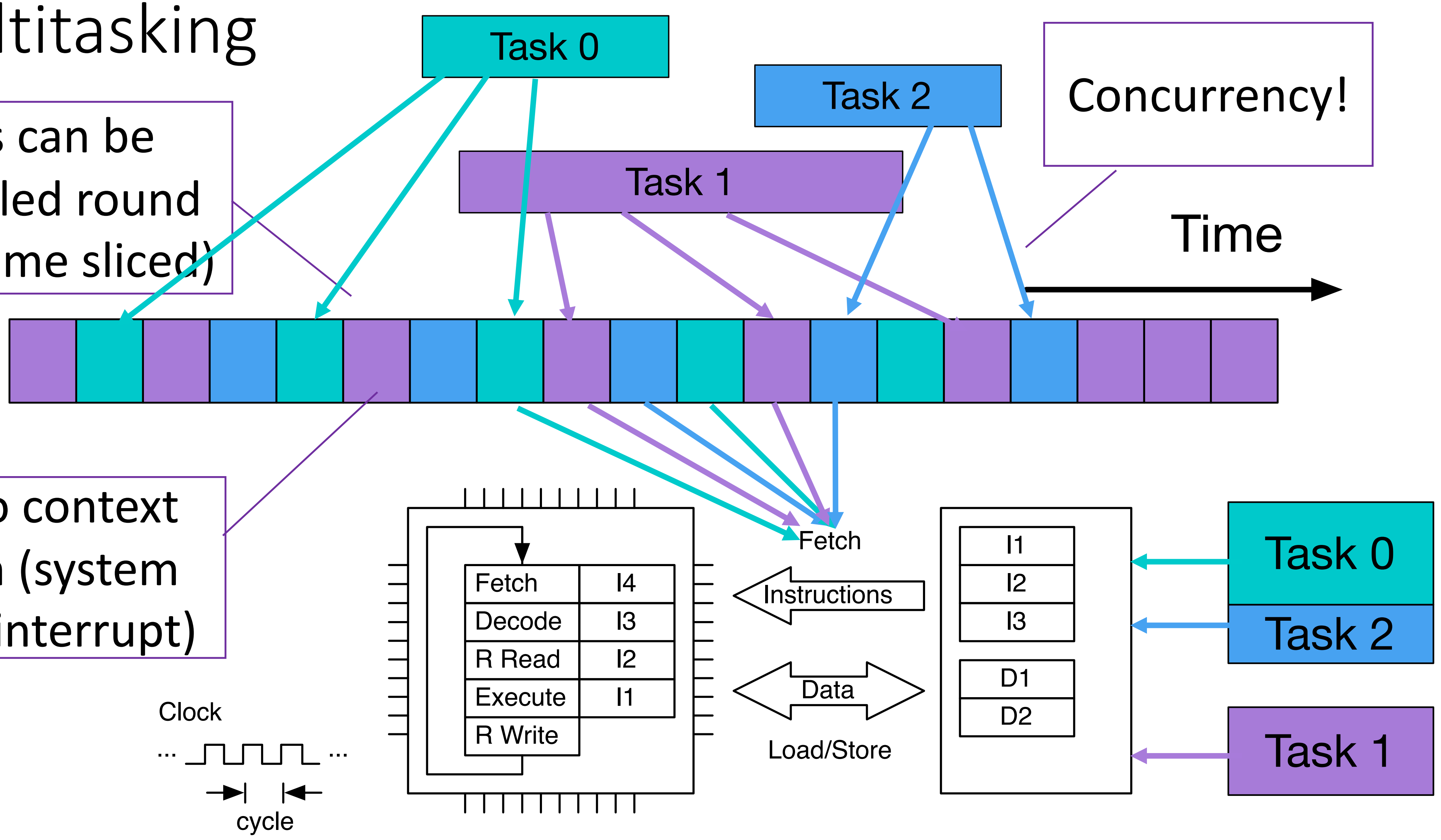
# How Do We Run Multiple Programs Concurrently?



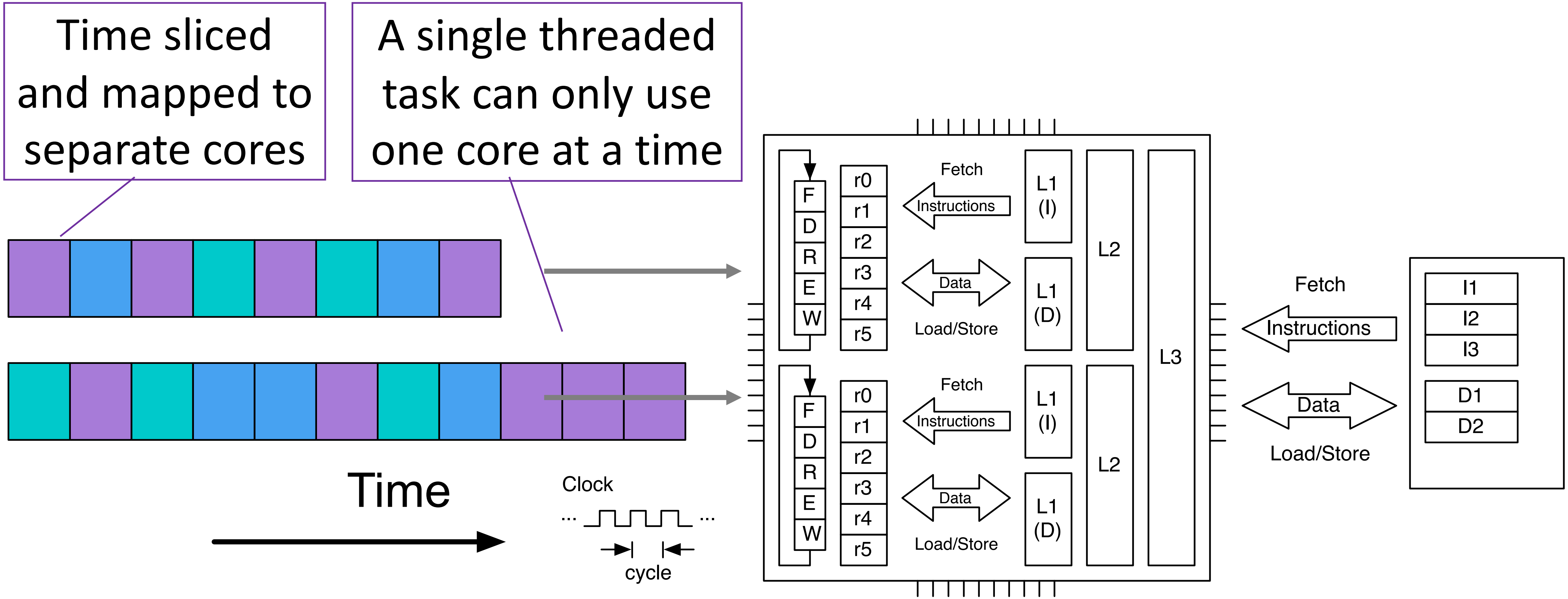
# Multitasking

Tasks can be scheduled round robin (time sliced)

Run to context switch (system call or interrupt)



# Multitasking on Multicore



# Multitasking on Multicore

Parallelism!

What about L1, L2?

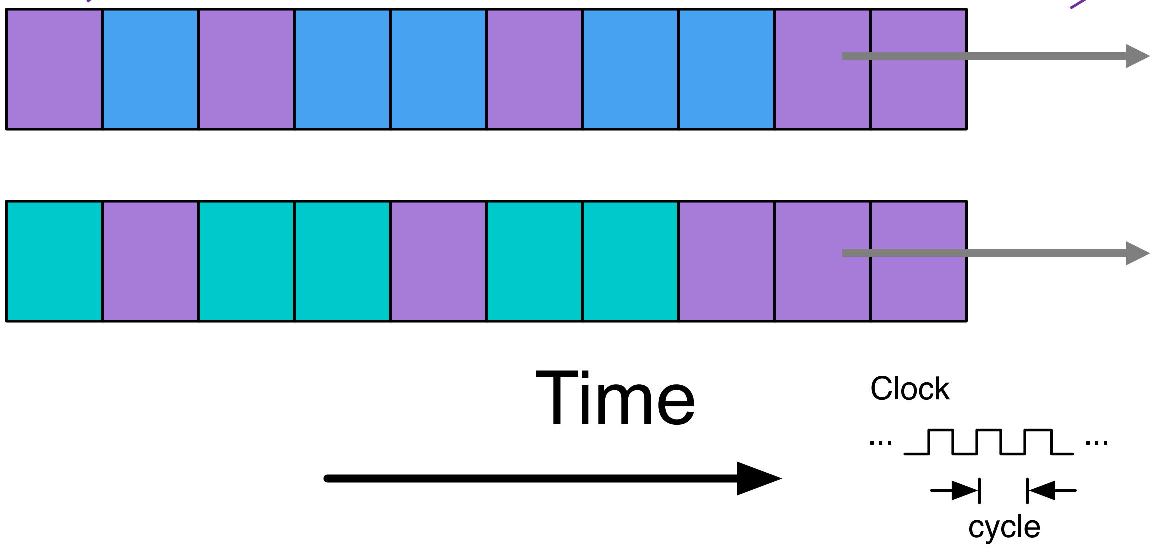
Time sliced and mapped to separate cores

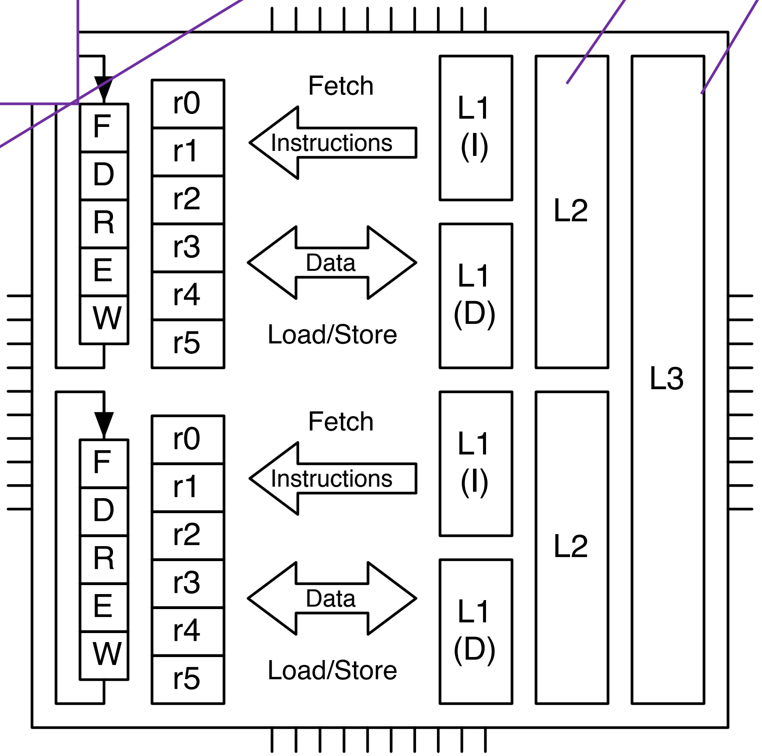
A multithreaded task can use multiple cores at a time

Shorter run time!

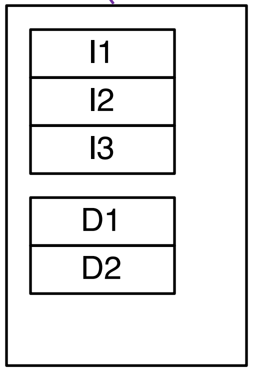
And L3 cache

Threads can share memory




Fetch
Instructions
Data
Load/Store




Access same variables

# Cache Coherence

Hardware managed

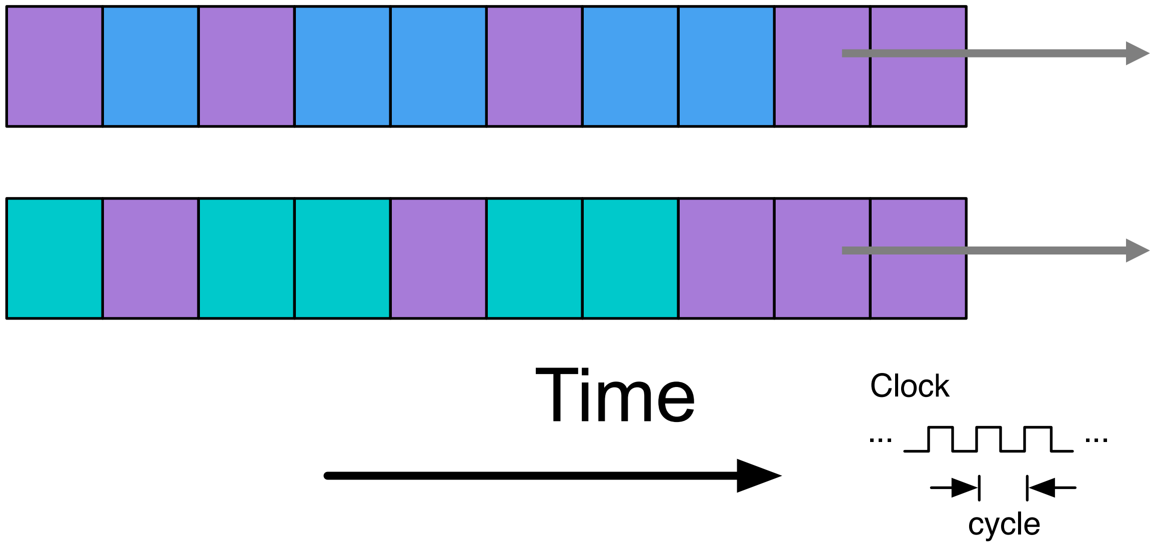
Same variable can be in two different caches

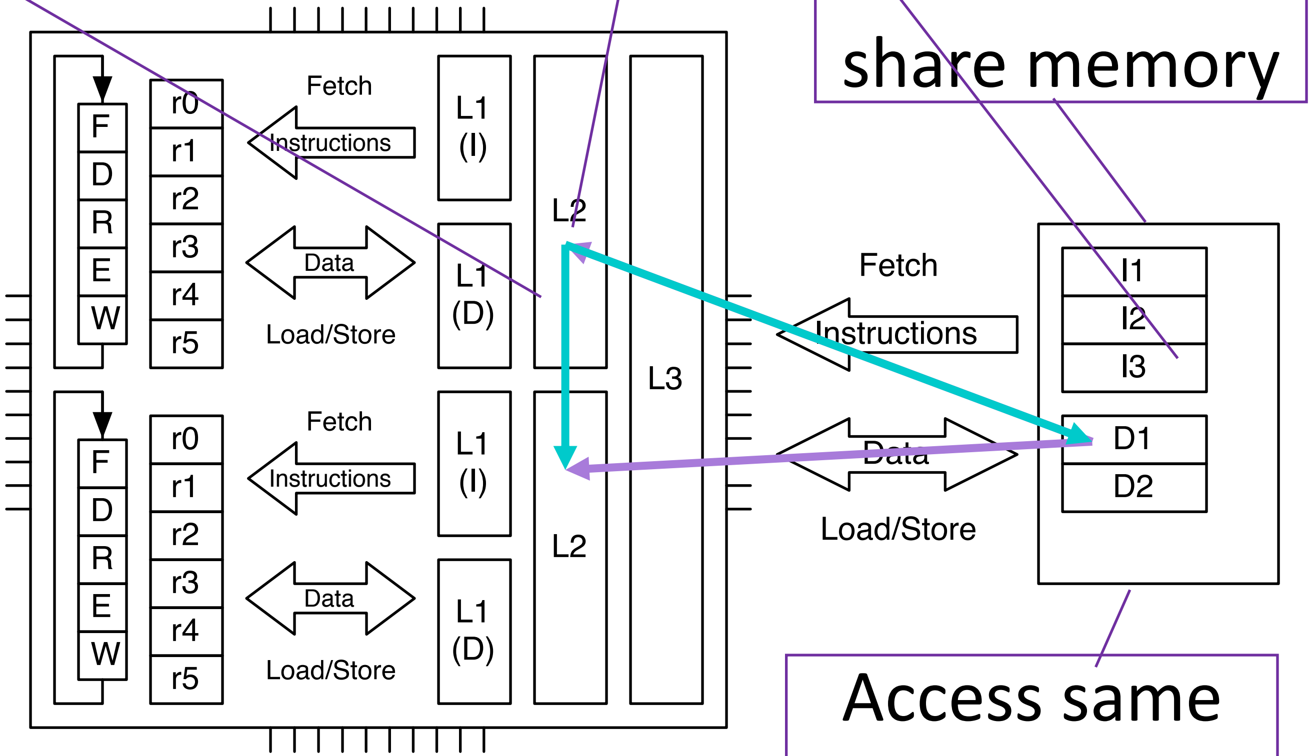
A multithreaded task can use multiple cores at a time

Cache coherence / memory consistency

What if one gets modified?

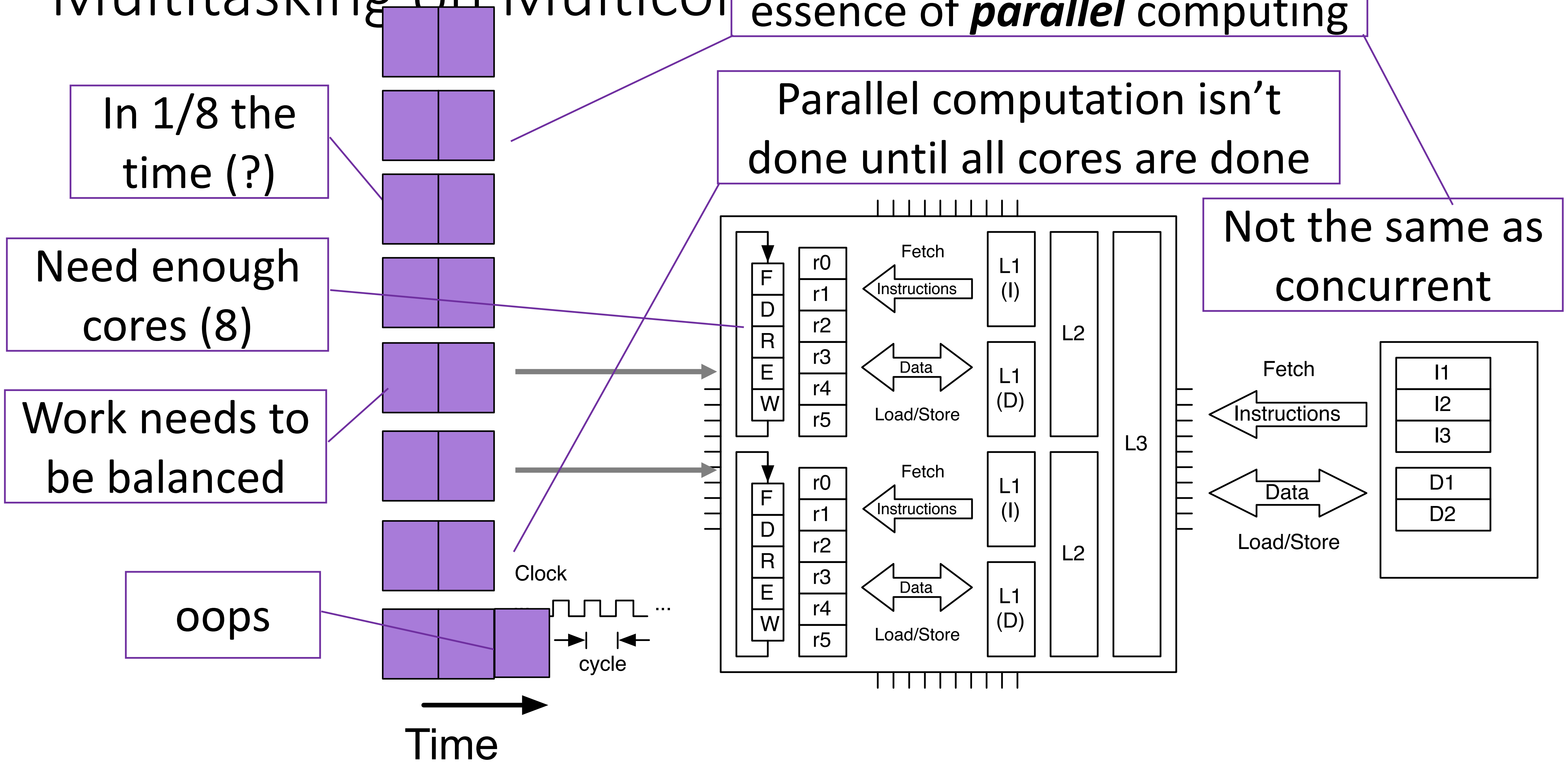
Threads can share memory



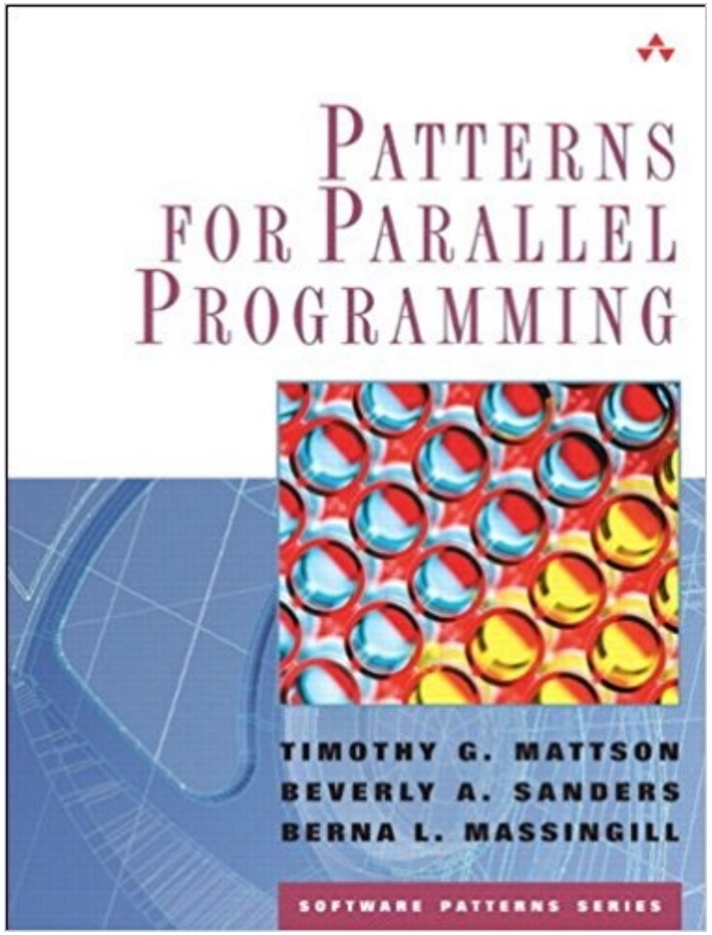




Access same variables

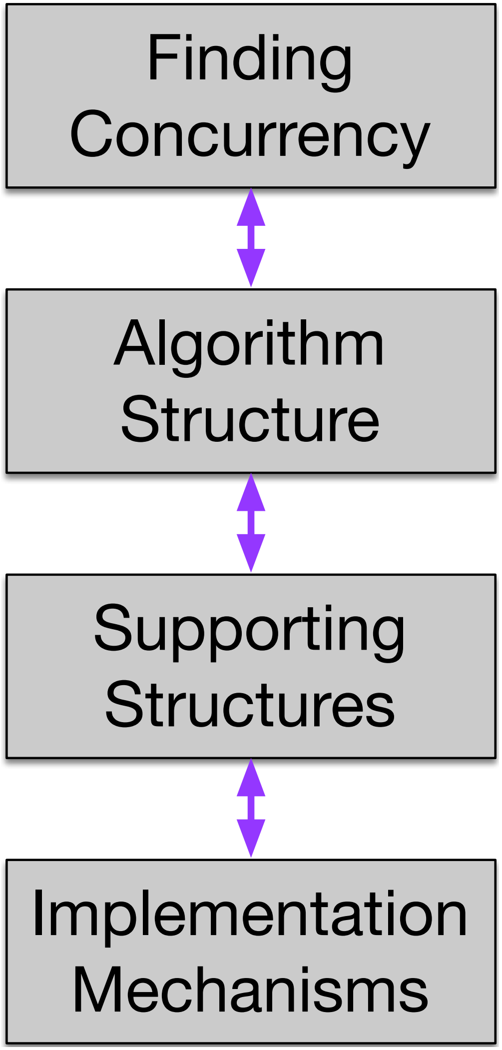
# Multitasking on Multicor



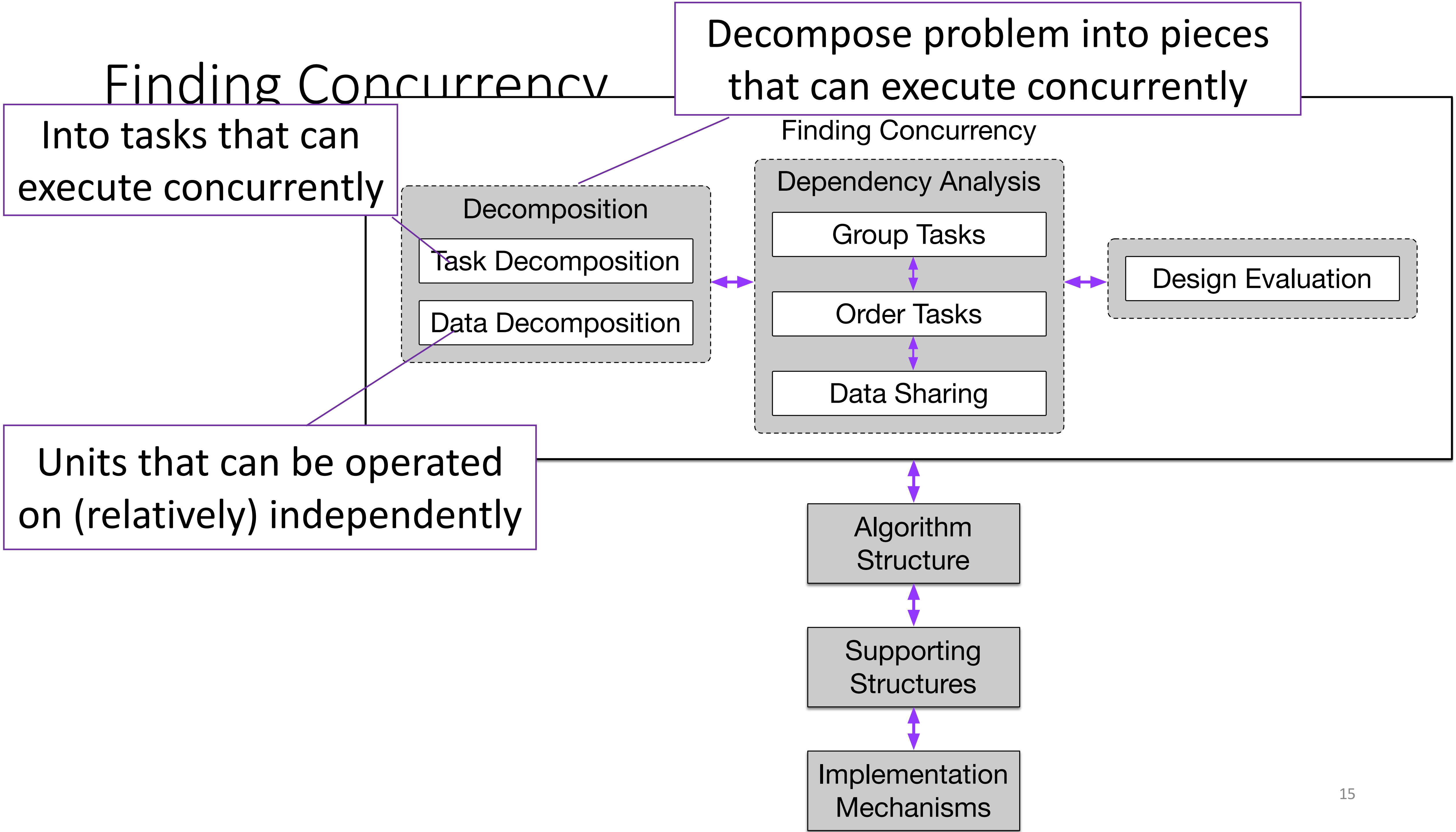
# Parallelization Strategy



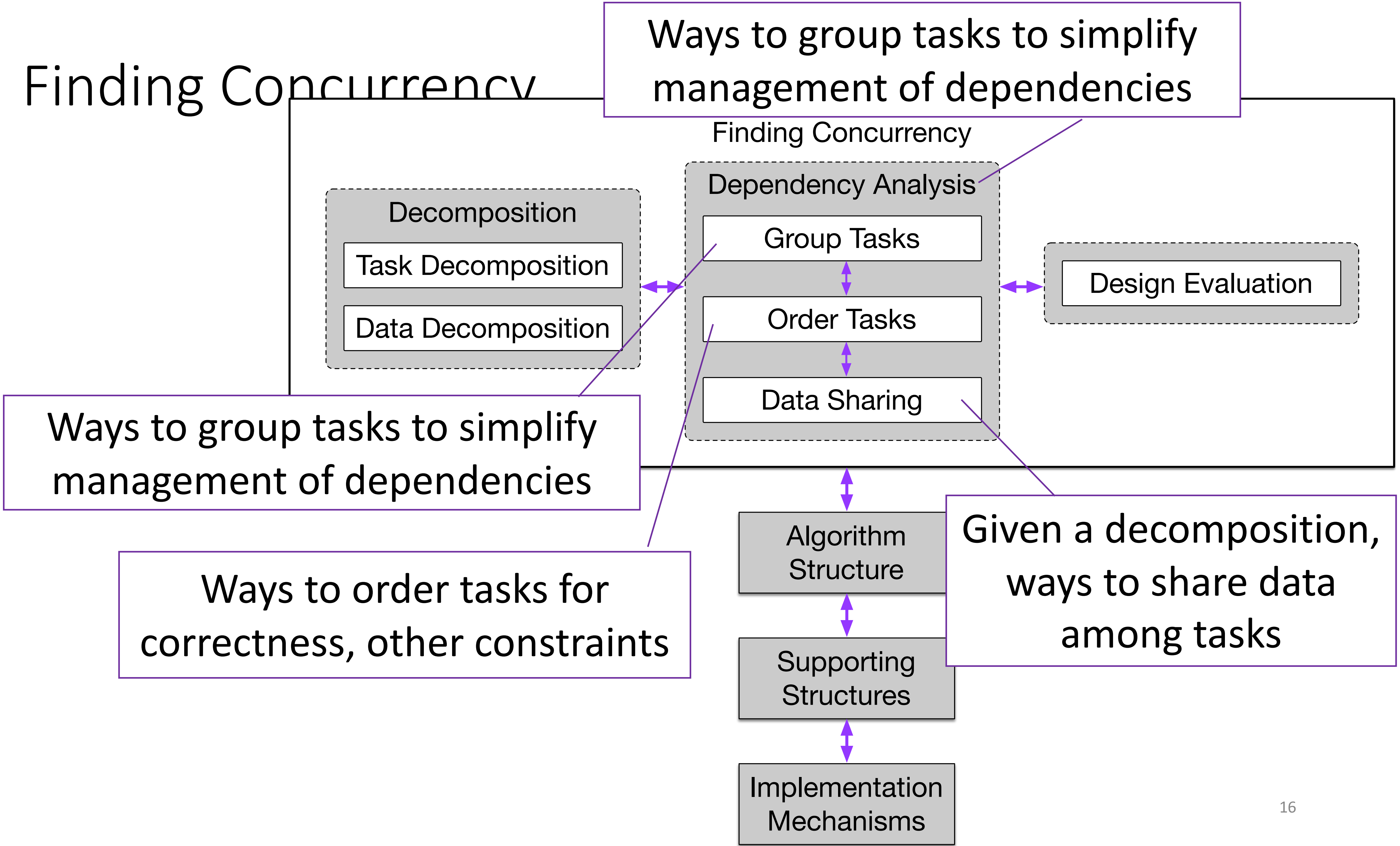

Timothy Mattson, Beverly Sanders, and Berna Massingill. 2004. *Patterns for Parallel Programming*(First ed.). Addison-Wesley Professional.



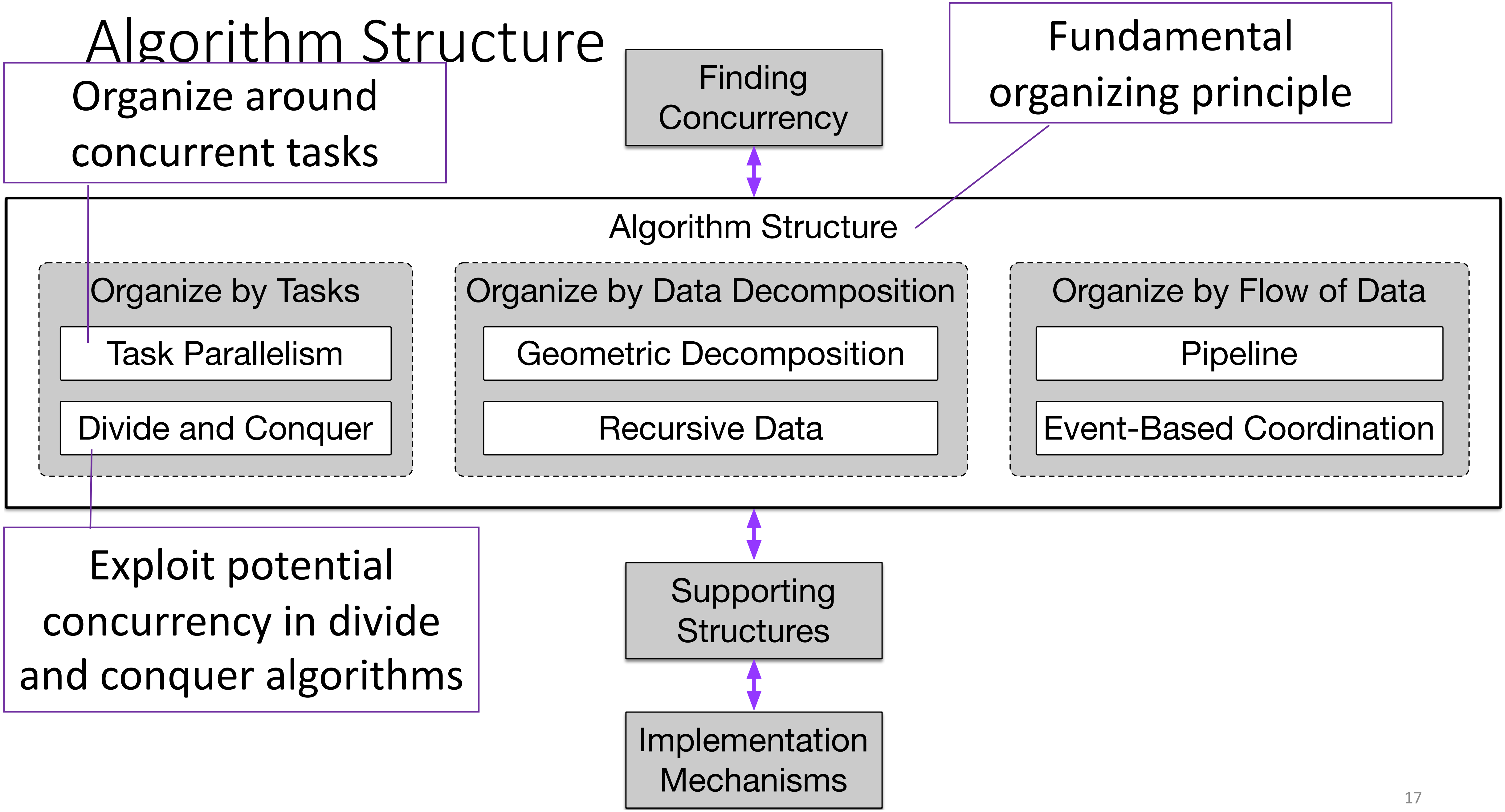
# Finding Concurrency



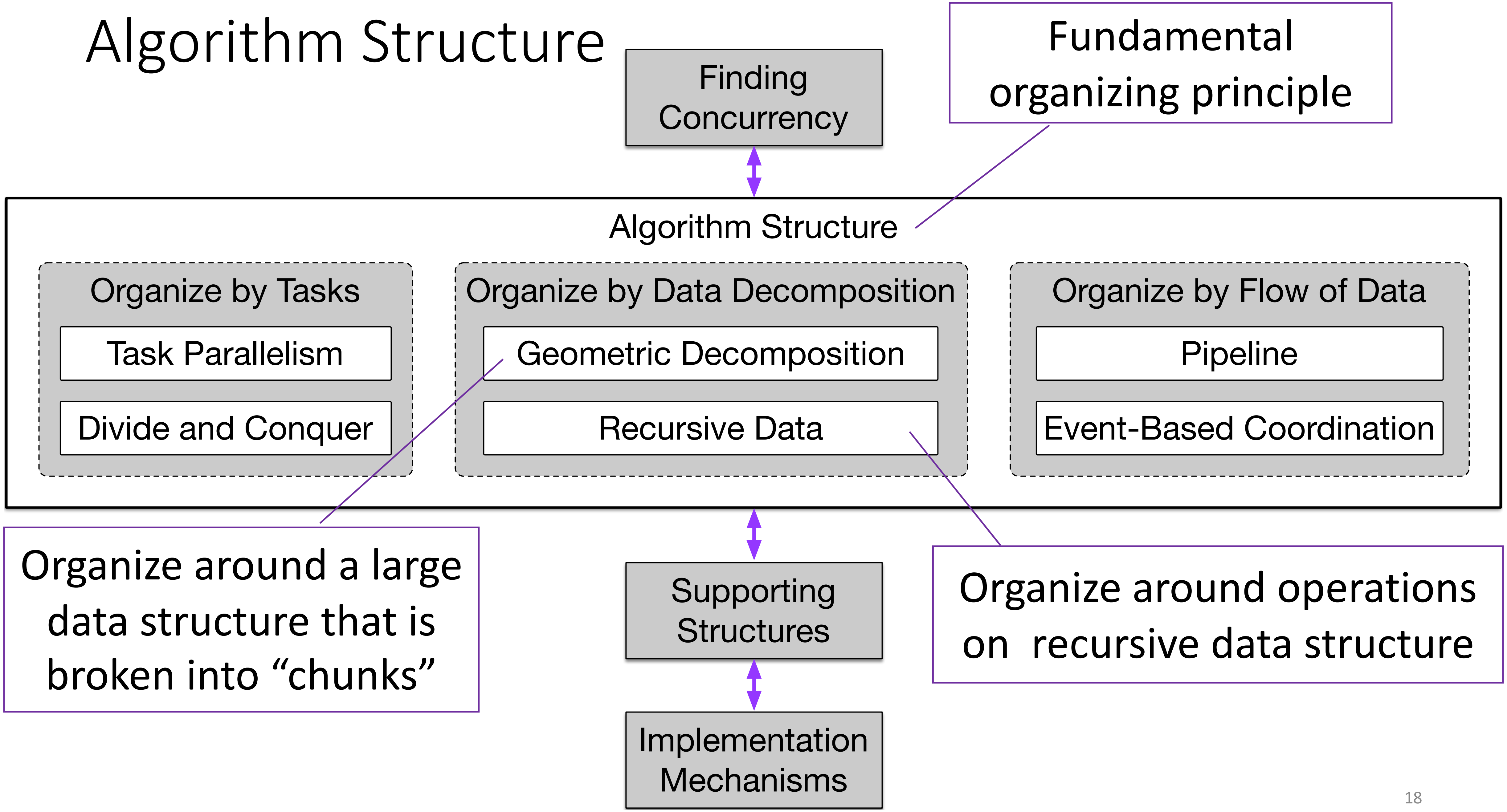
# Finding Concurrency



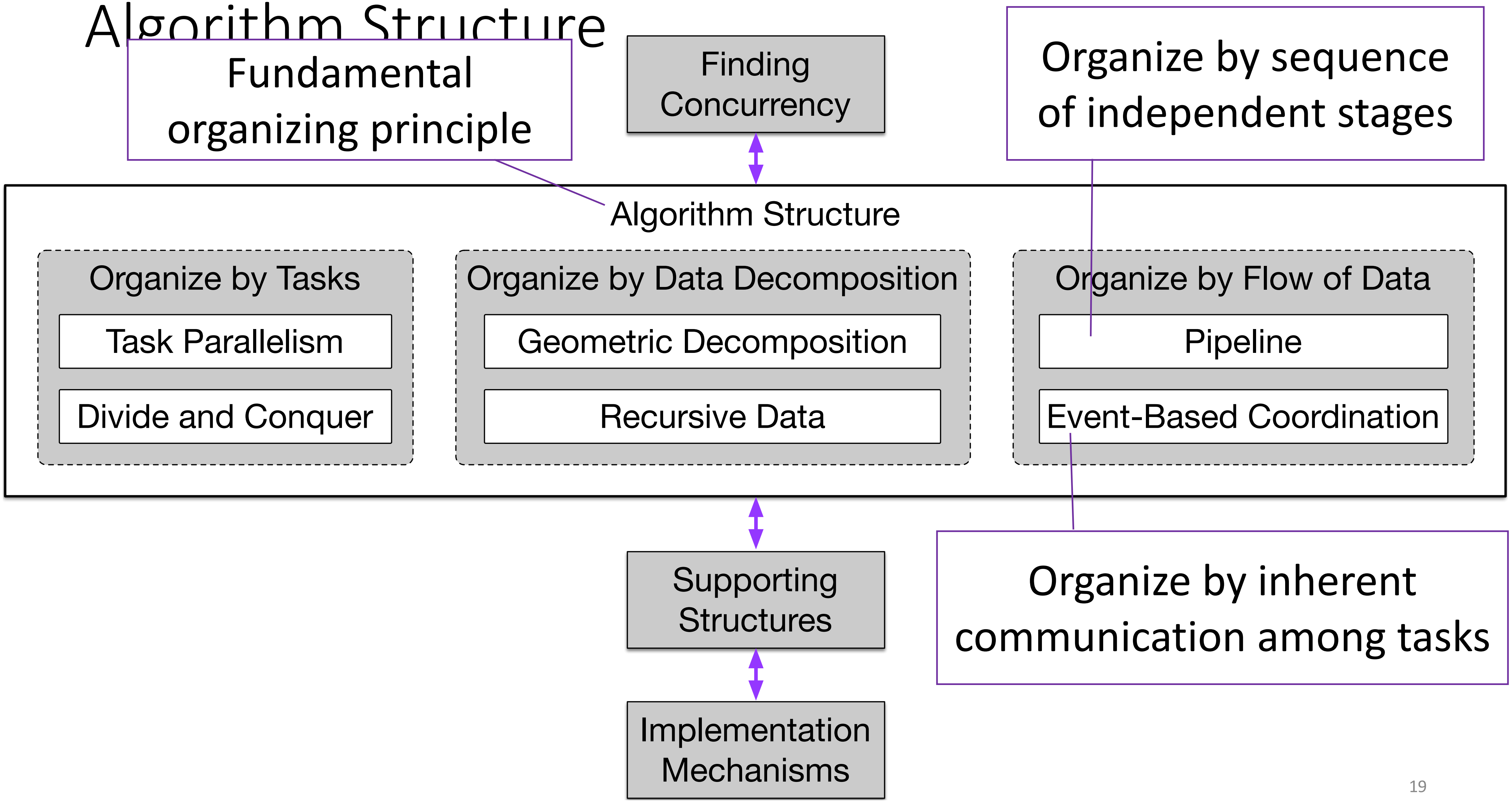
# Algorithm Structure



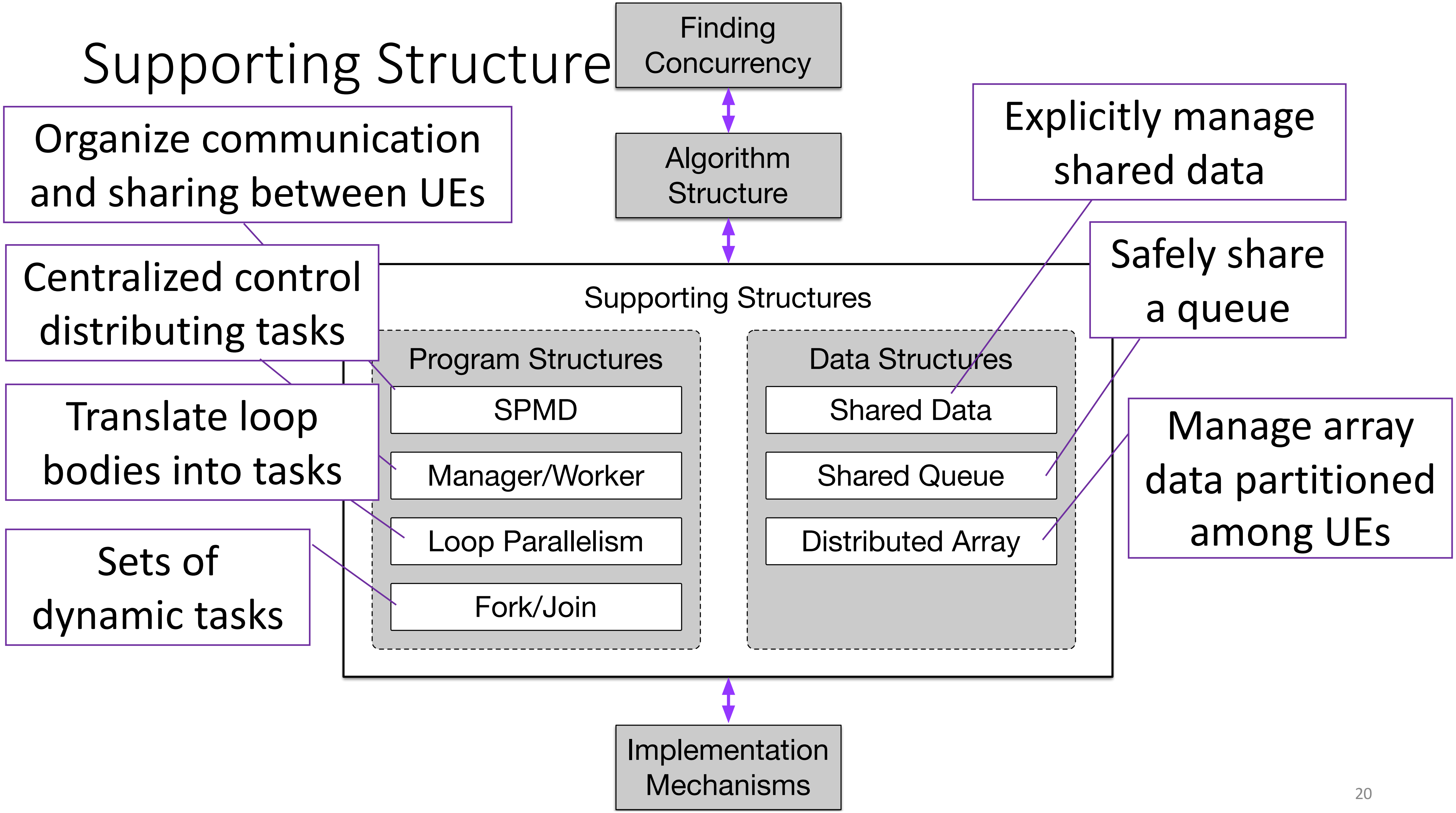
# Algorithm Structure



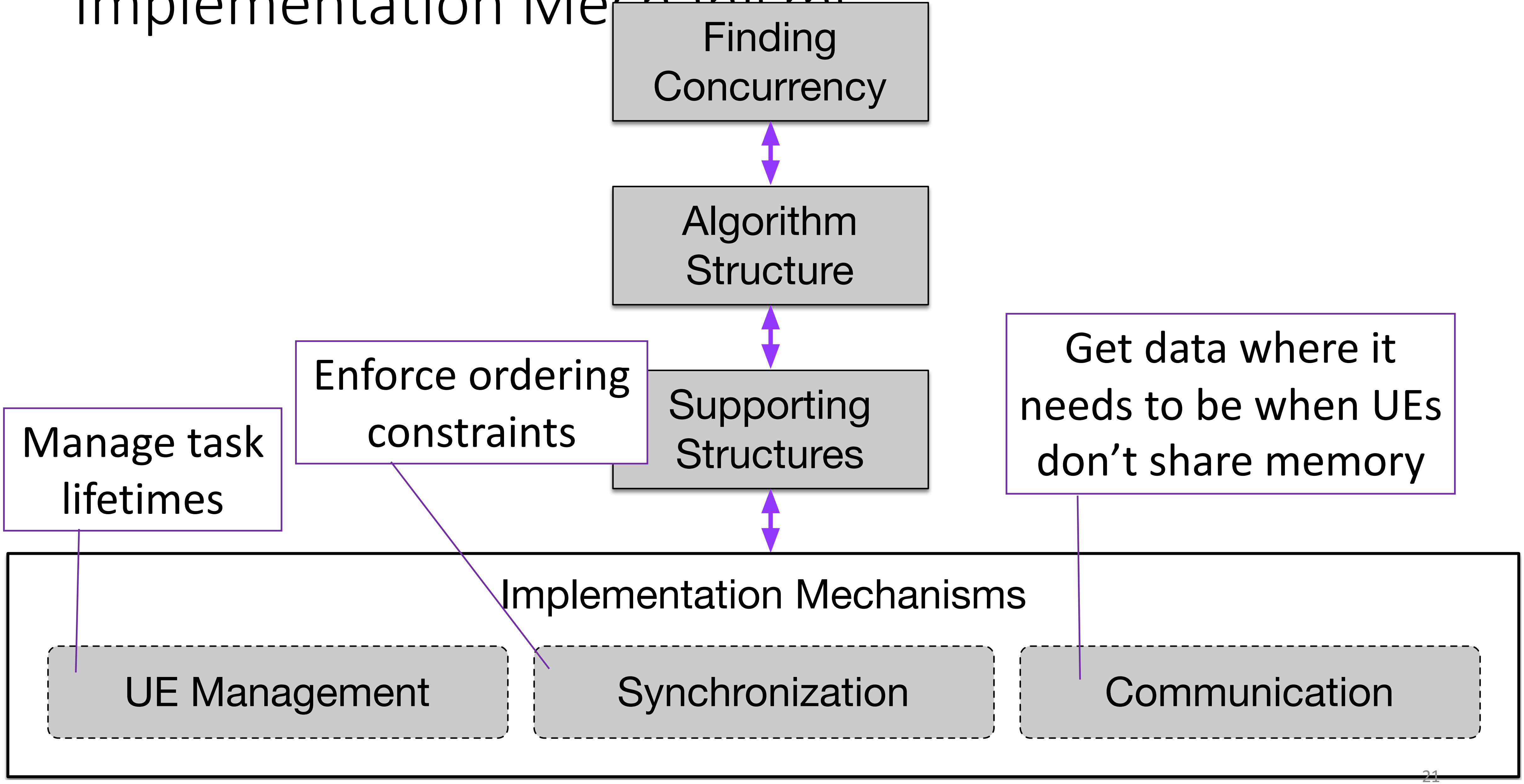
# Algorithm Structure



# Supporting Structure



# Implementation Mechanisms

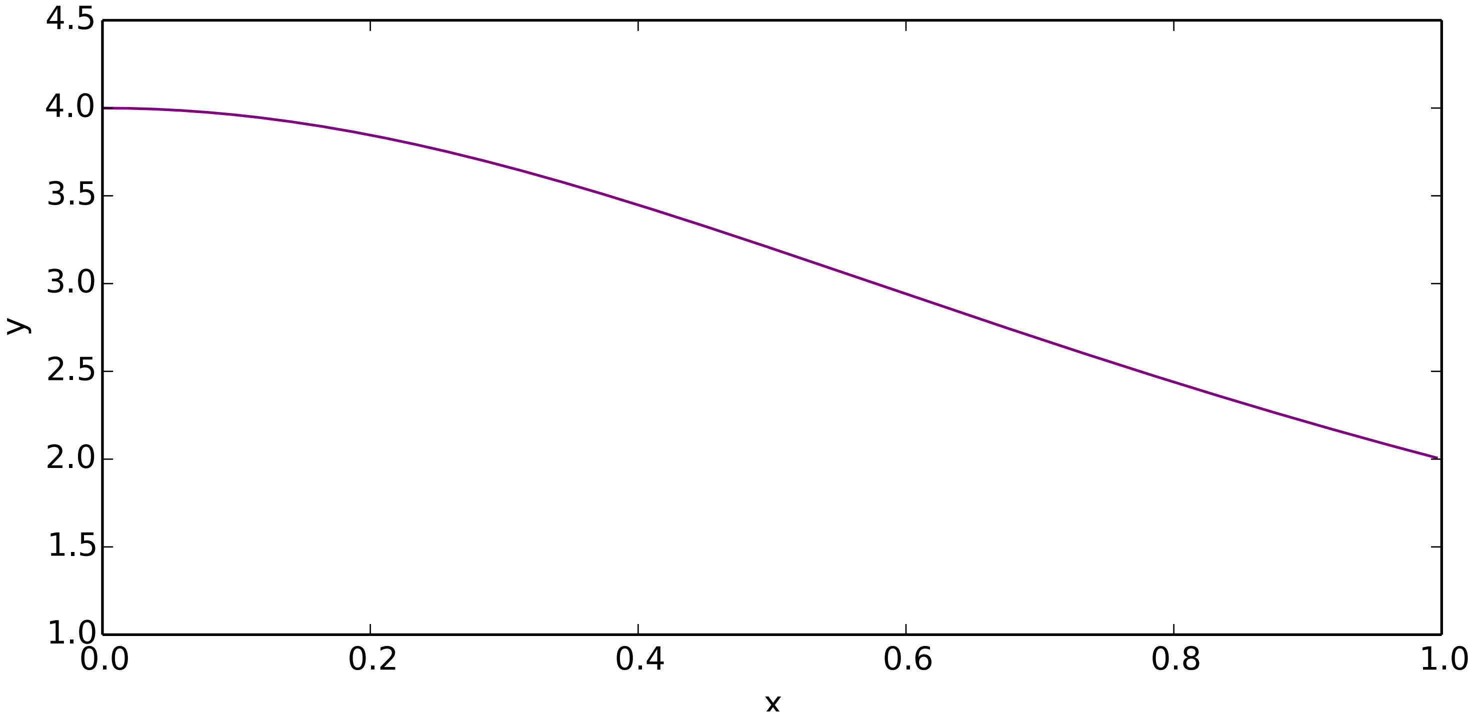


# Example

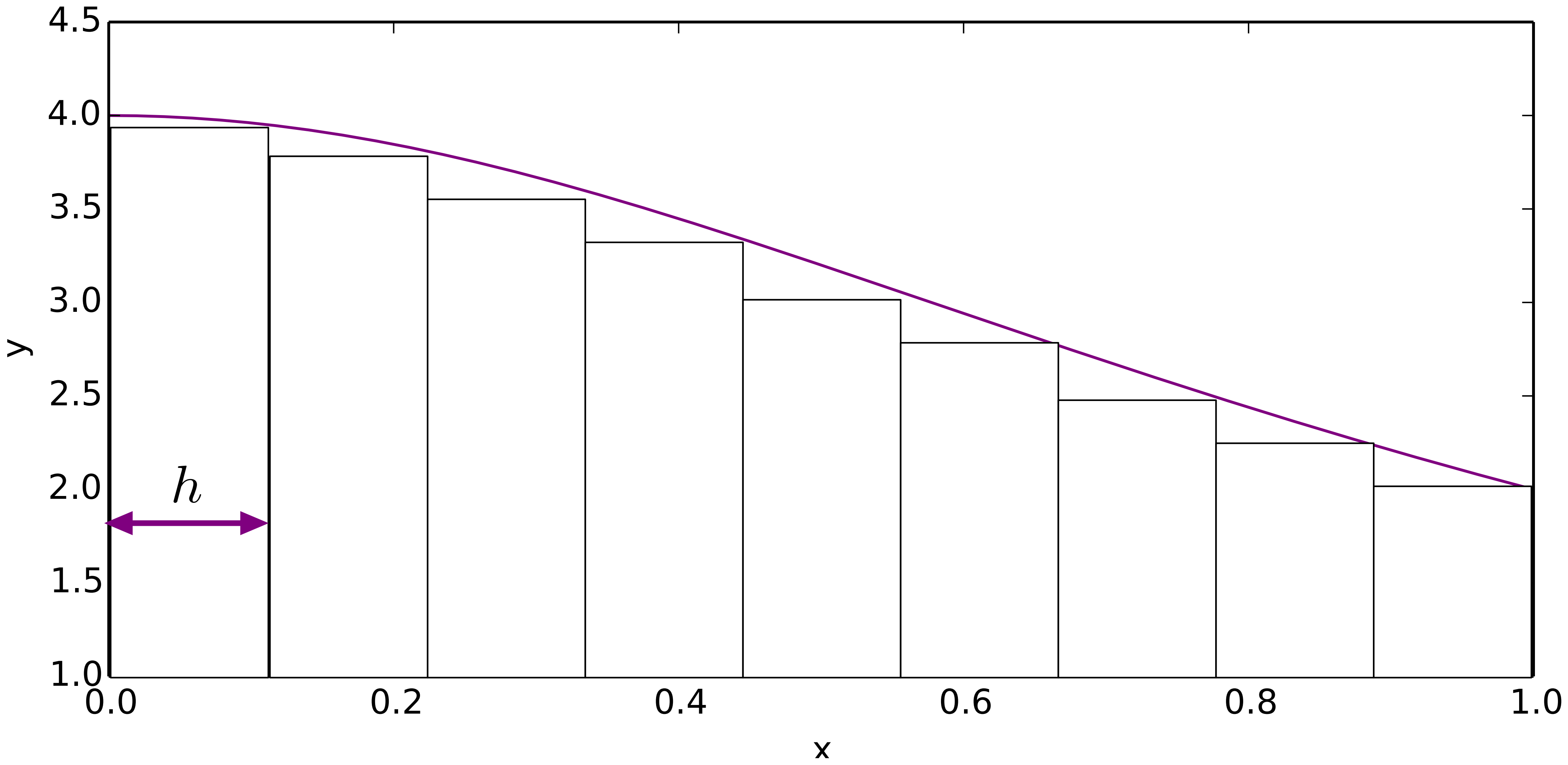
- Find the value of $\pi$

- Using formula

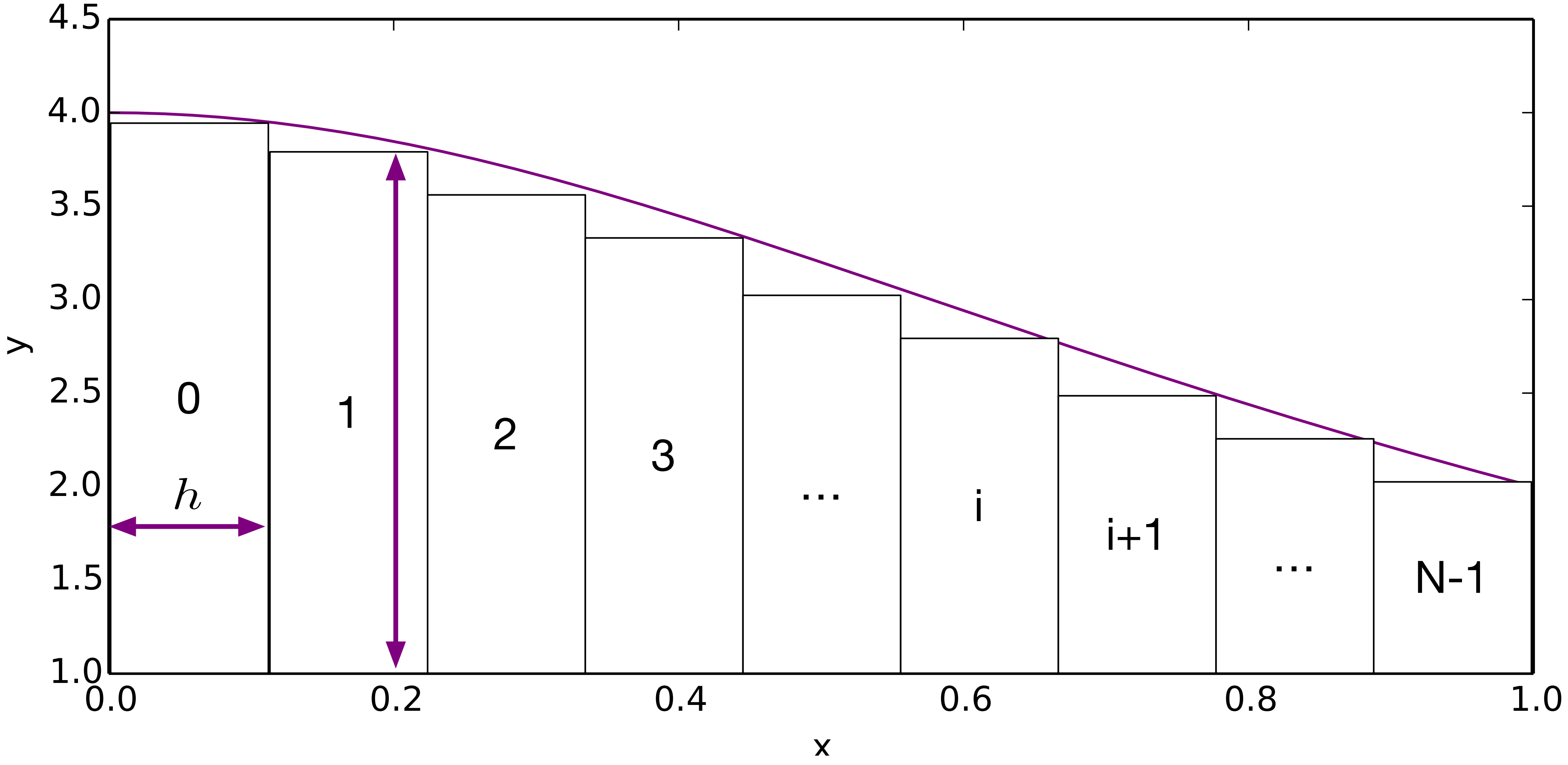
$$\pi = \int_0^1 \frac{4}{1+x^2} dx$$



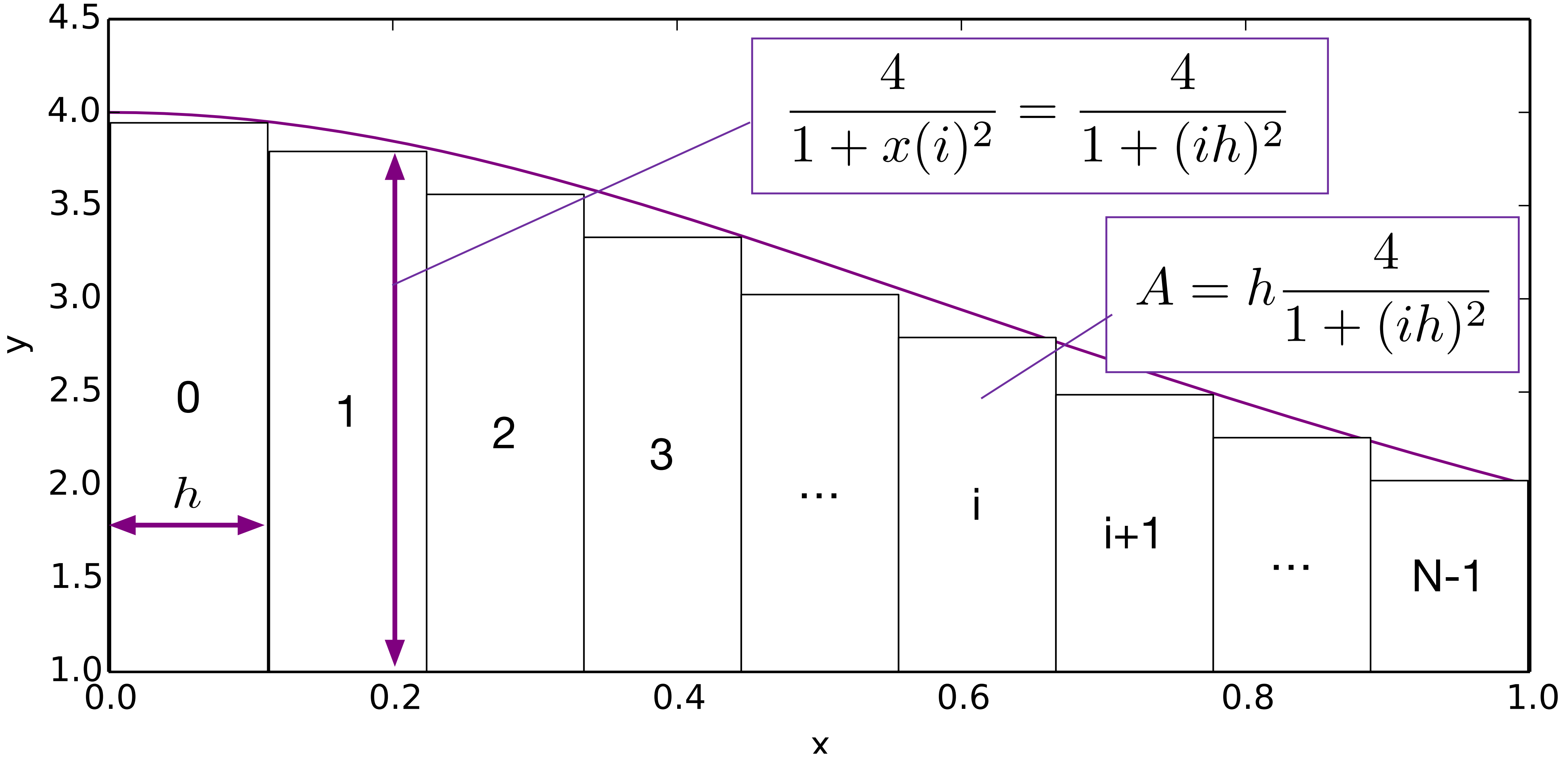
# Discretization



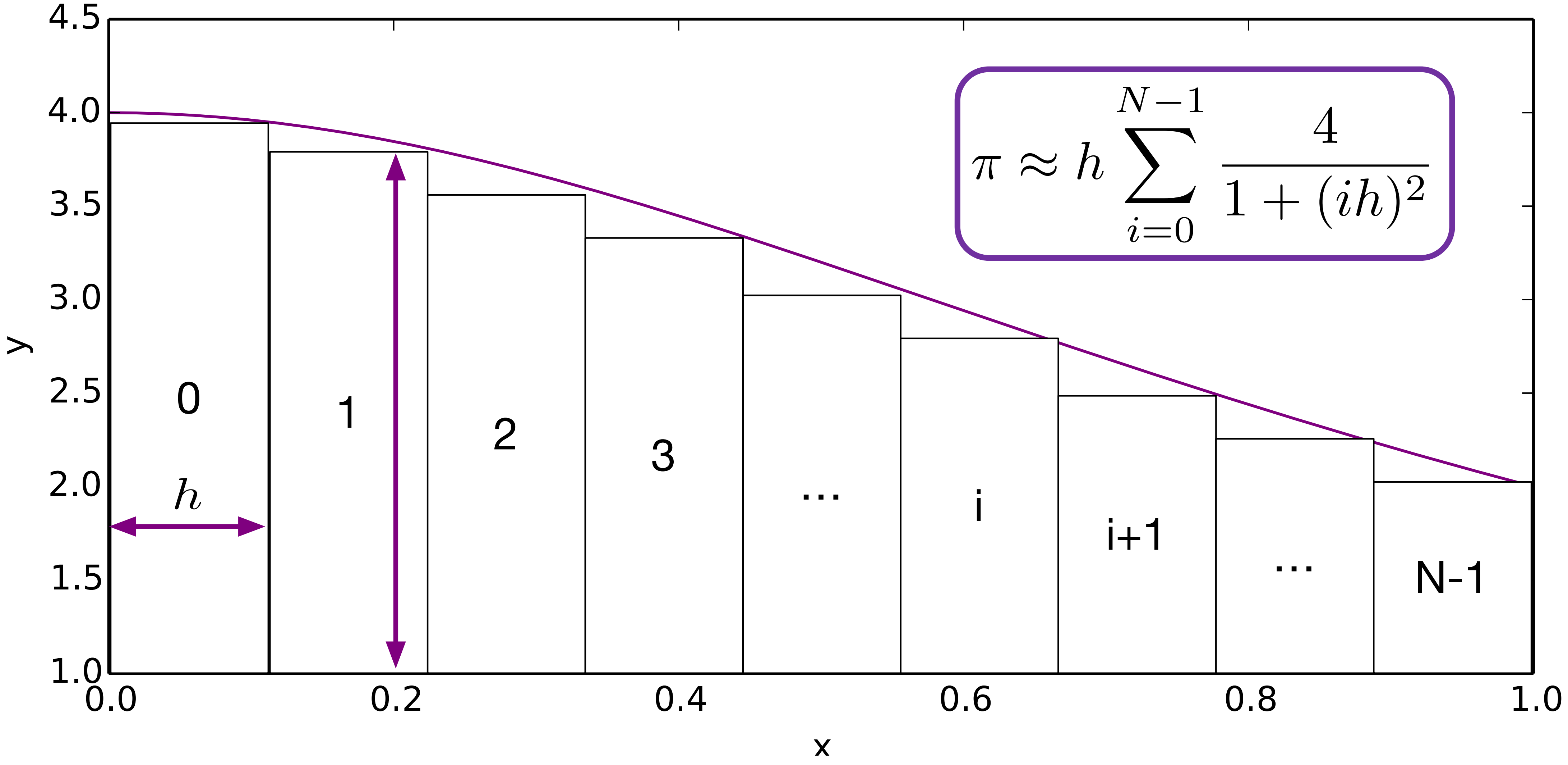
# Numerical Quadrature



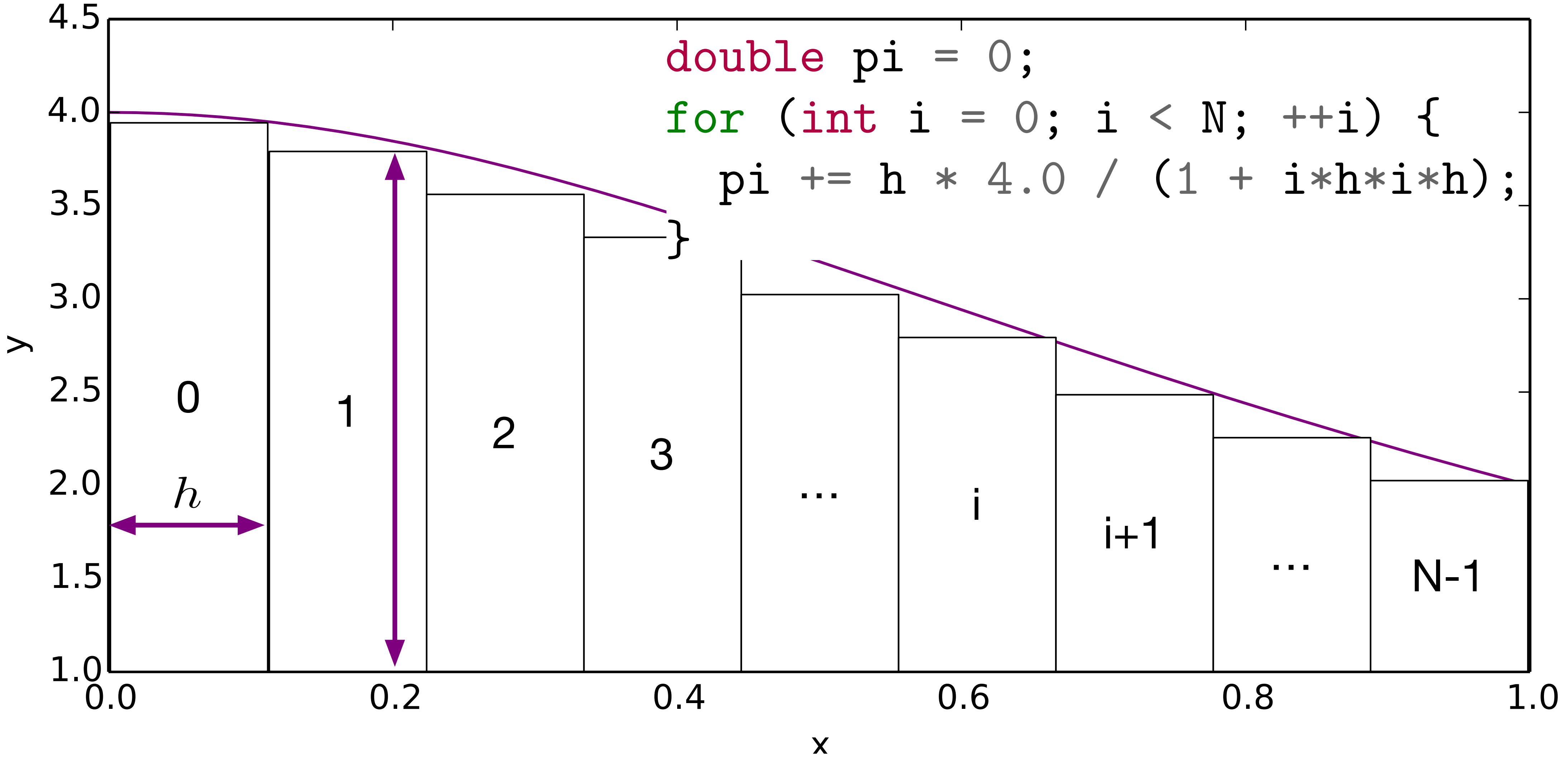
# Numerical Quadrature



# Numerical Quadrature

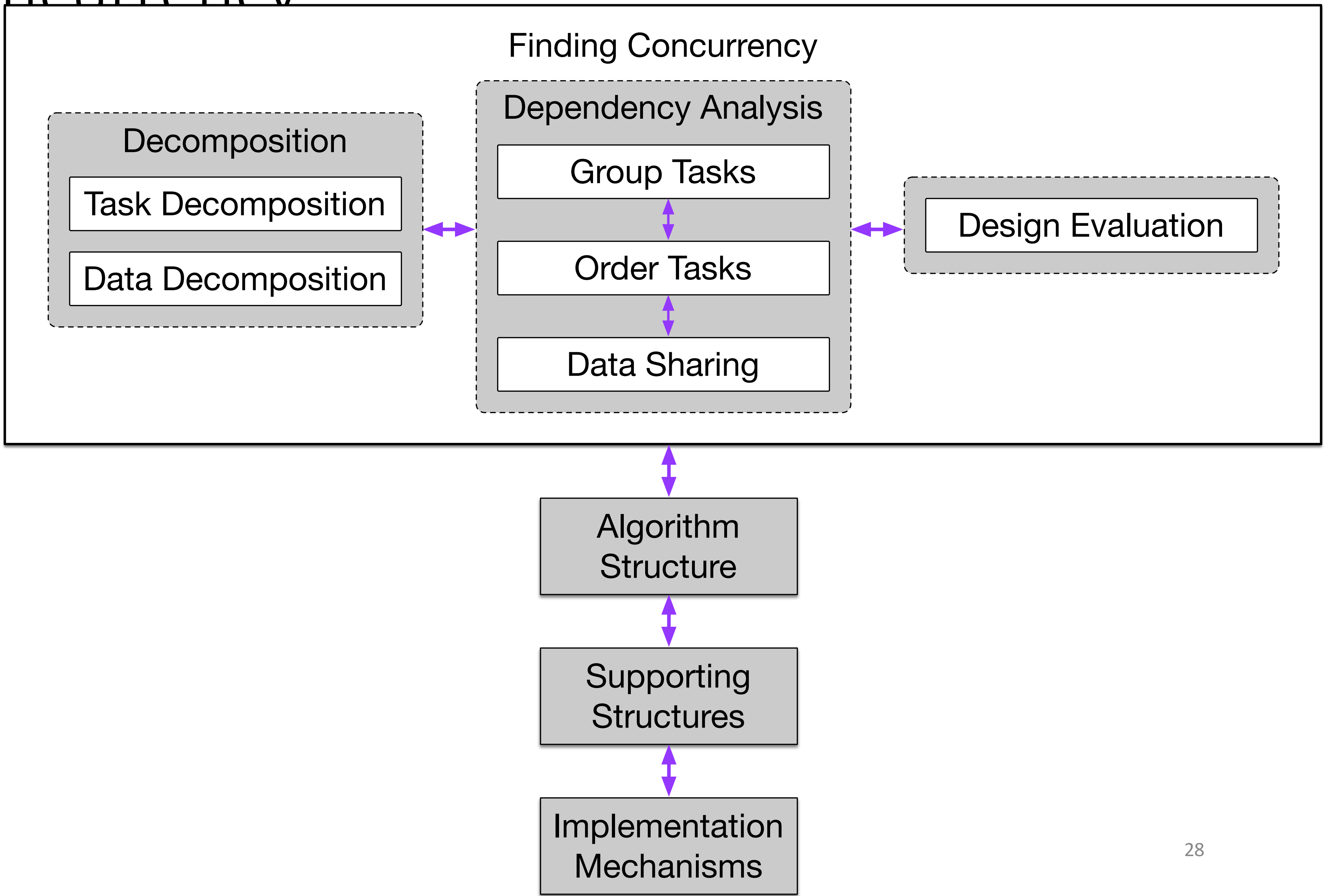


# Numerical Quadrature




```
double pi = 0;
for (int i = 0; i < N; ++i) {
  pi += h * 4.0 / (1 + i*h*i*h);
}
```

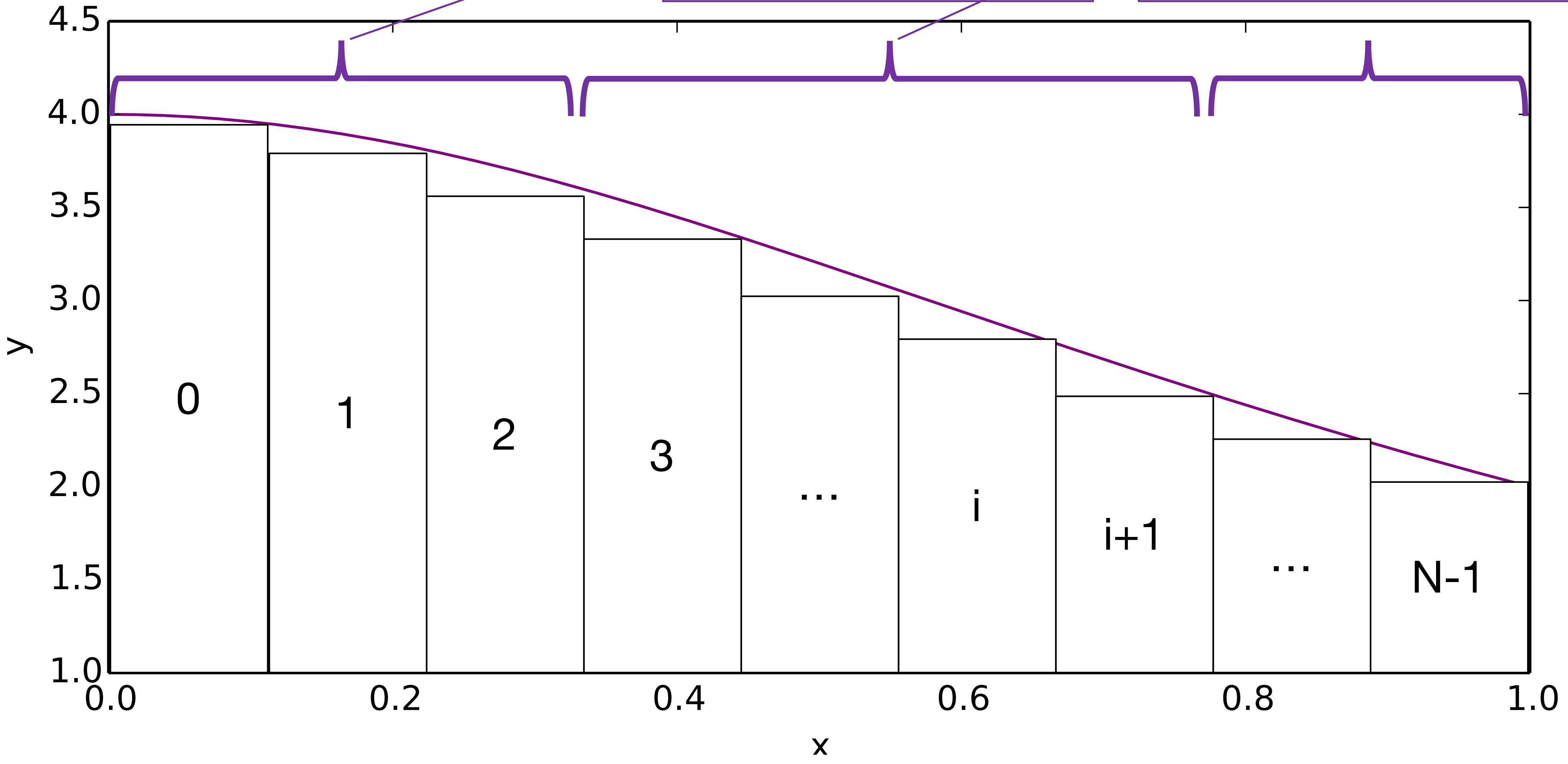
# Finding Concurrency



# Finding Concurrency

Partial sums are all independent

Can be computed concurrently

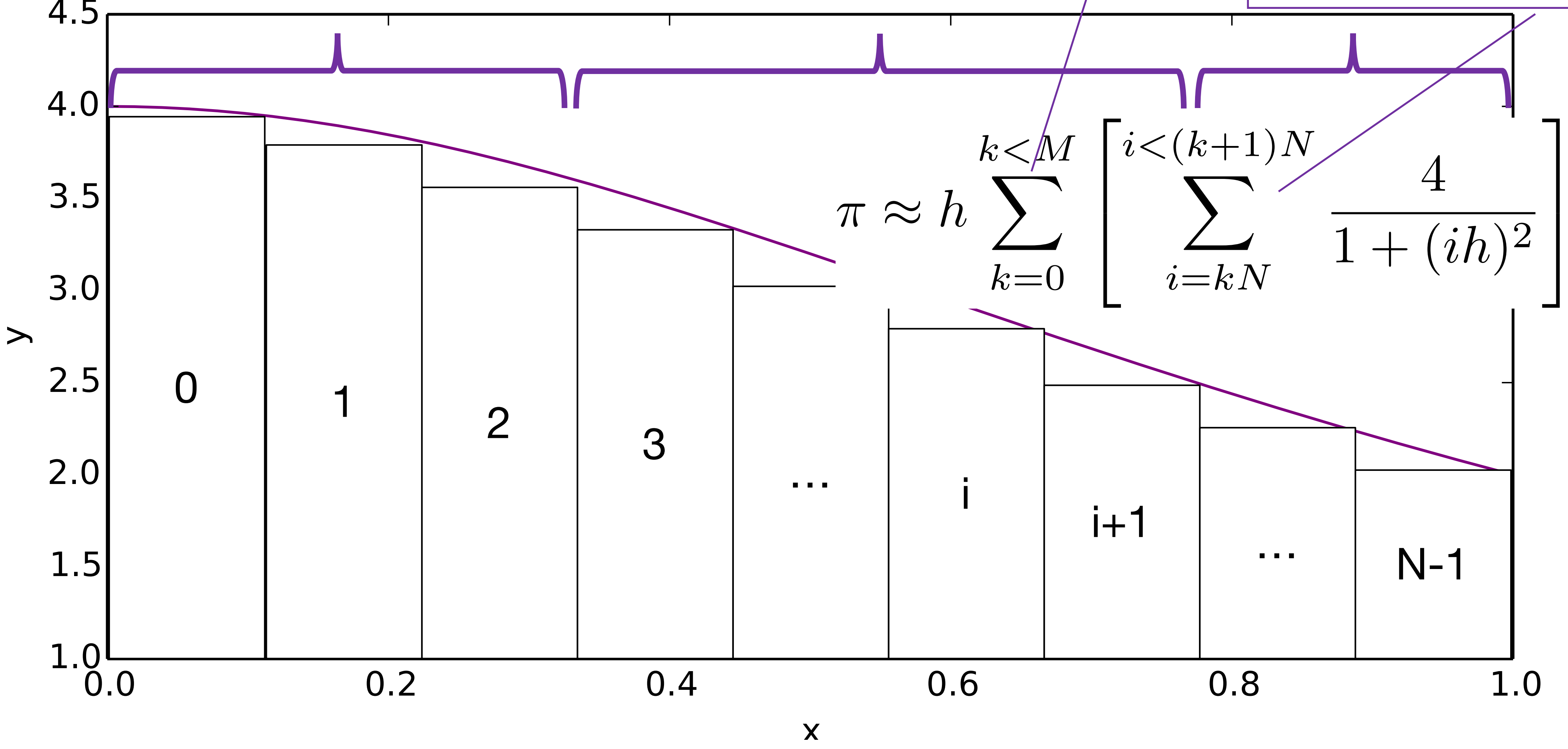


# Finding Concurrency

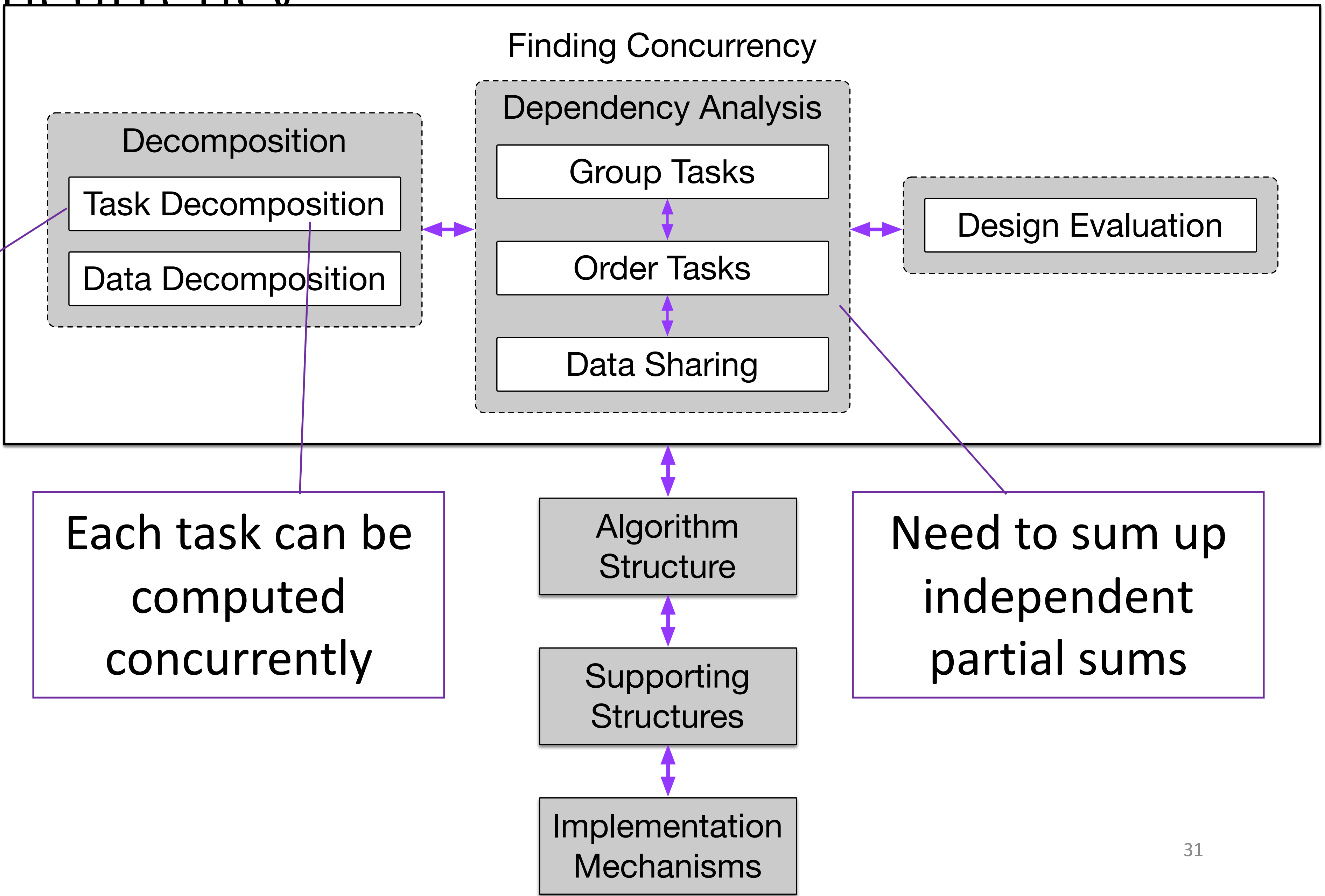
Sum over partial sums

Partial sum

$$\pi \approx h \sum_{k=0}^{k<M} \left[ \sum_{i=kN}^{i<(k+1)N} \frac{4}{1+(ih)^2} \right]$$



# Finding Concurrency

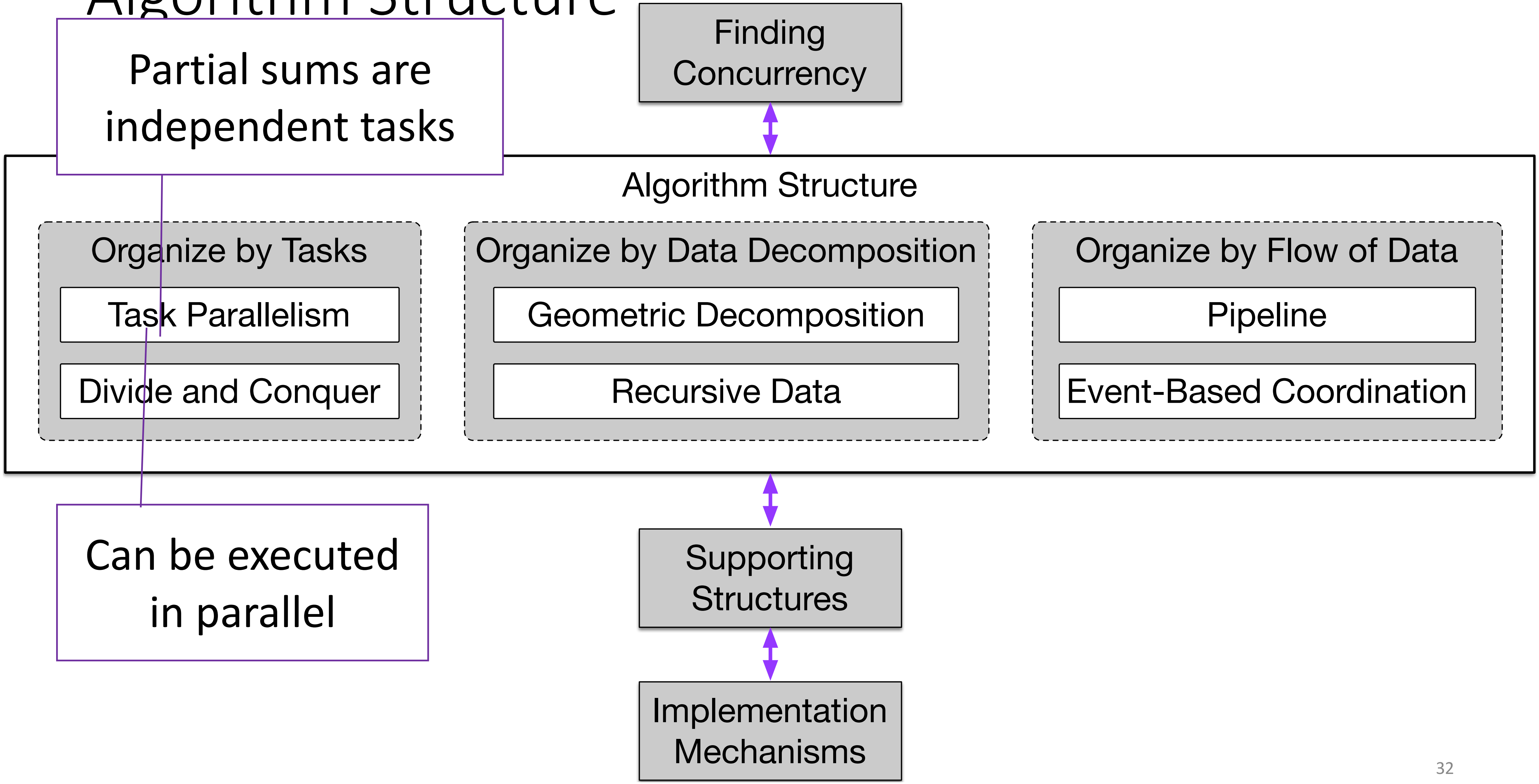



Decompose total sum into a sum of partial sums

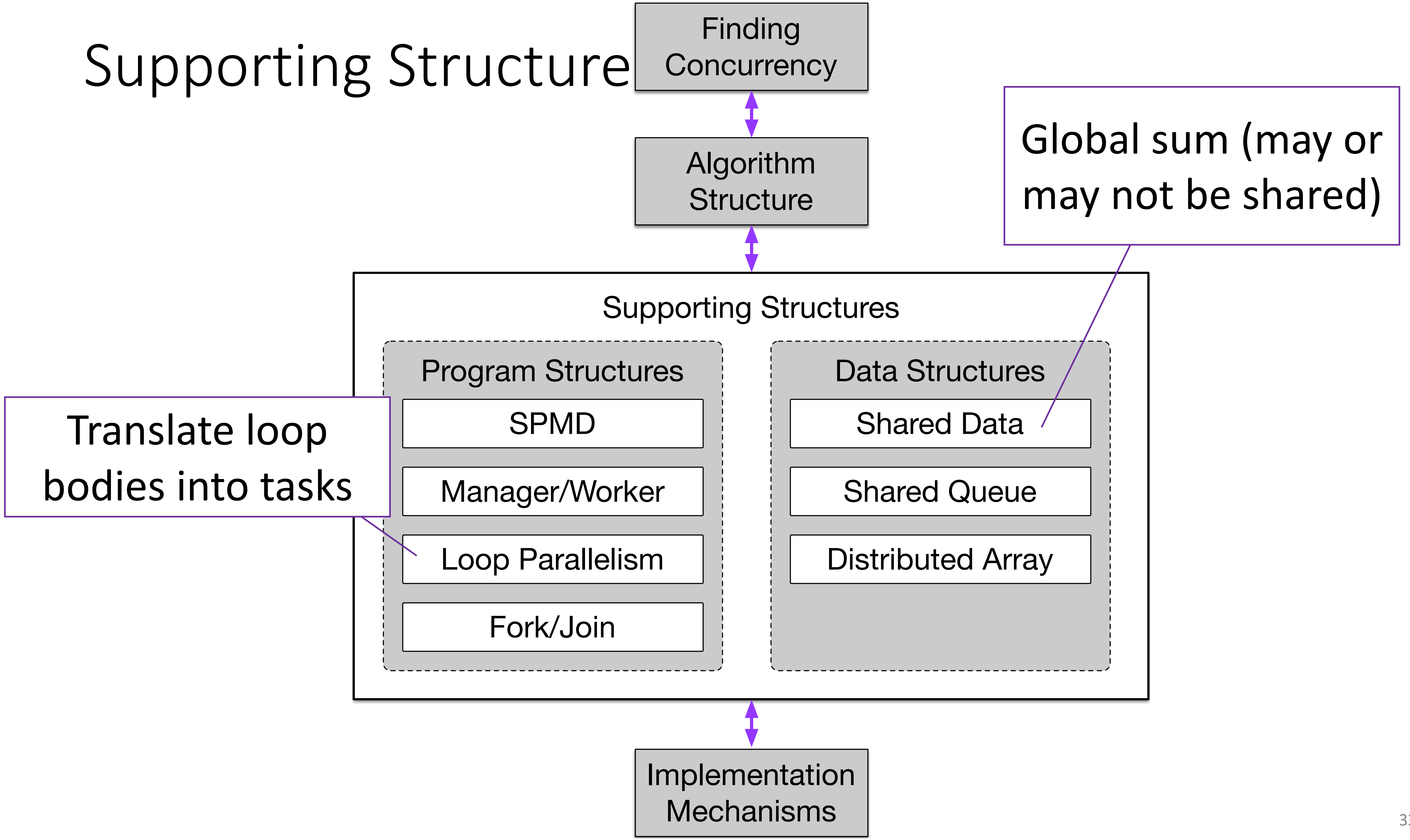
Each task can be computed concurrently

Need to sum up independent partial sums

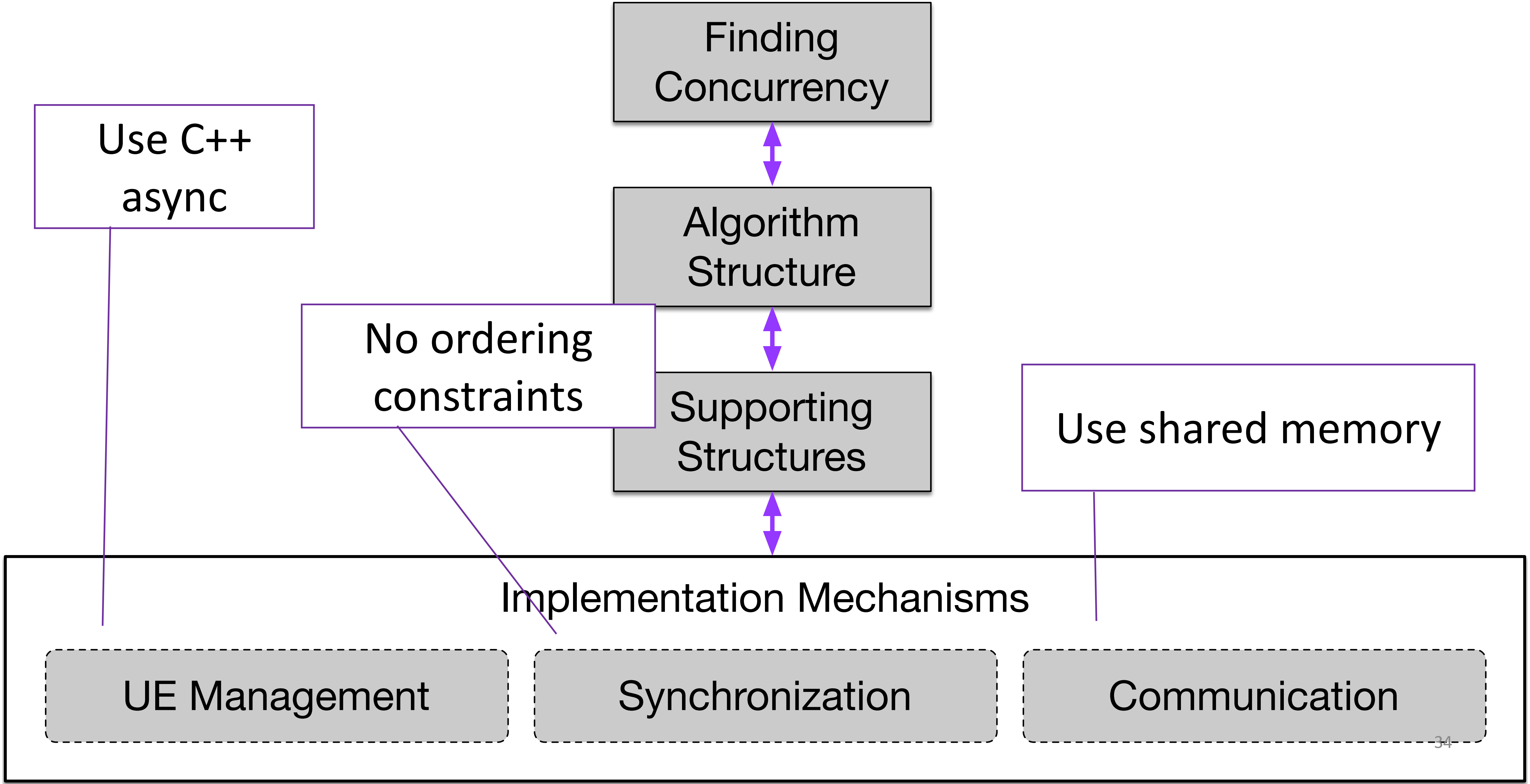
# Algorithm Structure



# Supporting Structure



# Implementation Mechanisms



# Sequential Implementation (Two Nested Loops)

```
double h = 1.0 / (double) intervals;

double pi = 0.0;
for (int k = 0; k < intervals; k += blocksize) {
  double partial_pi = 0.0;
  for (int i = k; i < (k+blocksize); ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h * partial_pi;
}
```

Discretization

For each set of discretized points

Compute partial sum

Accumulate final sum

# Threads vs Tasks

```
void sayHello(int tnum) {
  cout << "Hello World.  I am thread " << tnum << endl;
}

int main() {
  std::thread tid[16];

  for (int i = 0; i < 16; ++i)
    tid[i] = thread (sayHello, i);

  for (int i = 0; i < 16; ++i)
    tid[i].join();

  return 0;
}
```

Task

Launch threads

"fork"

Wait for tasks to finish

"join"

# Threads

Thread returns void

Oops

How do we get partial sums?

How do we update global total?

```
void partial_pi(unsigned long begin, unsigned long end) {
  double partial_pi = 0.0;
  for (unsigned long i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  return partial_pi;
}

int
main(int argc, char *argv[])
{
  double h = 1.0 / (double) intervals;

  double pi = 0.0;
  for (int k = 0; k < intervals; k += blocksize) {

    pi += h * partial_pi;
  }
  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

# Threads

Task

Assign task to thread

```
void partial_pi(unsigned long begin, unsigned long end, double h, double& pi) {
  double partial_pi = 0.0;
  for (unsigned long i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h*partial_pi;
}

int
main(int argc, char *argv[])
{
  std::vector<std::thread> threads;

  double h = 1.0 / (double) intervals;

  double pi = 0.0;
  for (unsigned long k = 0; k < num_blocks; ++k) {
    threads.push_back(std::thread(partial_pi,
        k*blocksize, (k+1)*blocksize, h, std::ref(pi)));
  }

  for (unsigned long k = 0; k < num_blocks; ++k) {
    threads[k].join();
  }
  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

# Threads

Local variable

Shared variable

```
void partial_pi(unsigned long begin, unsigned long end, double h, double& pi) {
  double partial_pi = 0.0;
  for (unsigned long i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  pi += h*partial_pi;
}
```

Update shared variable

# Threads

```
int main(int argc, char *argv[]) {
  double h = 1.0 / (double) intervals;

  std::vector<std::thread> threads;

  double pi = 0.0;
  for (unsigned long k = 0; k < num_blocks; ++k) {
    threads.push_back(
      std::thread(
        partial_pi, k*blocksize, (k+1)*blocksize, h, std::ref(pi)));
  }

  for (unsigned long k = 0; k < num_blocks; ++k) {
    threads[k].join();
  }
  std::cout << "pi is approximately " << pi << std::endl;

  return 0;
}
```

Container for created threads

Thread constructor

Function that will be the task

Arguments to the function

Have to explicitly tag this as a reference

We are invoking std::thread, not partial pi

# Results

```
$ ./thrpi
pi is approximately 3.14159

$ ./thrpi
pi is approximately 3.14159

$ ./thrpi
pi is approximately 2.69322
```

Correct

Correct

Incorrect!

Exactly same program!

What happened?

# Name This Famous Couple

Bonnie Parker

Clyde Barrow

# Bonnie and Clyde Use ATMs

```
int bank_balance = 300;

void withdraw(const string& msg, int amount) {
  int bal = bank_balance;
  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;
  bank_balance = bal - amount;
}

int main() {
  cout << "Starting balance is " << bank_balance << endl;

  thread bonnie(withdraw, "Bonnie", 100);
  thread clyde(withdraw, "Clyde", 100);

  bonnie.join();
  clyde.join();

  cout << "Final bank balance is " << bank_balance << endl;

  return 0;
}
```

# Withdraw Function

```
int bank_balance = 300;

void withdraw(const string& msg, int amount) {
  int bal = bank_balance;
  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;
  bank_balance = bal - amount;
}
```

Get balance

Compute the new balance

Save new balance

# Making Concurrent Withdrawals

```
int main() {
  cout << "Starting balance is " << bank_balance << endl;

  thread bonnie(withdraw, "Bonnie", 100);
  thread clyde(withdraw, "Clyde", 100);

  bonnie.join();
  clyde.join();

  cout << "Final bank balance is " << bank_balance << endl;

  return 0;
}
```

Launch threads

Run withdraw function

Constructor

Wait for completion

# Bonnie and Clyde Use ATMs


```
$ ./a.out
Starting balance is 300
Bonnie withdraws 100
Clyde withdraws 100
```

Is this correct?

# What Happened

Bonnie's thread, bal = 300

```
void withdraw(const string& msg, int amount) {
  int bal = bank_balance;
  string out_s = msg + " withdraws " + to_string(amt) + "\n";
```

Context switch

```
  cout << out_s;
  bank_balance = bal - amount;
}
```

bal is still 300

bank_balance gets 200

Clyde's thread, bal = 300

```
void withdraw(const string& msg, int amount) {
  int bal = bank_balance;
  string out_s = msg + " withdraws " + to_string(amt) + "\n";
```

Context switch

```
  cout << out_s;
  bank_balance = bal - amount;
}
```

bal is still 300

bank_balance gets 200

Profit!

# What Happened: Race Condition

- Final answer depends on instructions from different threads are interleaved with each other
- Often occurs with shared writing of shared data
- Often due to read then update shared data
- What was true at the read is not true at the update

# Critical Section Problem

```
int bank_balance = 300;

void withdraw(const string& msg, int amount) {
    int bal = bank_balance;
    string out_string = msg + " withdraws " + to_string(amount) + "\n";
    cout << out_string;
    bank_balance = bal - amount;
}
```

We want to tell operating system not to run anything else here

When some thread is executing this ***critical section***, no other thread may execute it

# The Critical-Section Problem

- n processes all competing to use some shared data
- Each process has a code segment, called critical section, in which the shared data is accessed.
- Problem – ensure that when one process is executing in its critical section, no other process is allowed to execute in its critical section.
- What do we mean by “execute in its critical section”?

# Solution to Critical-Section Problem

- ***Mutual Exclusion*** - If process Pi is executing in its critical section, then no other processes can be executing in their critical sections
- ***Progress*** - If no process is executing in its critical section and there exist some processes that wish to enter their critical section, then the selection of the processes that will enter the critical section next cannot be postponed indefinitely
- ***Bounded Waiting*** - A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted
  - Assume that each process executes at a nonzero speed
  - No assumption concerning relative speed of the N processes

# Critical Section Problem

```
int bank_balance = 300;

void withdraw(const string& msg, int amount) {
  int bal = bank_balance;
  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;
  bank_balance = bal - amount;
}
```

This is a ***critical section***

Let's just think about mutual exclusion for now

# Critical Section Problem

```
bool lock = false;

int bank_balance = 300;

void withdraw(const string& msg, int amount) {
  while (lock == true)
    ;
  lock = true;

  int bal = bank_balance;
  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;
  bank_balance = bal - amount;

  lock = false;
}
```

Test if another thread is holding the lock

Spin if it is

Fall through when lock == false

Take the lock

Execute critical section

Release lock

# Aside

```
bool lock = false;

int bank_balance = 300;

void withdraw(const string& msg, int amount) {

  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;
  bank_balance -= amount;

}
```

Still a race

# Aside

```
bool lock = false;

int bank_balance = 300;

void withdraw(const string& msg, int amount) {

  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;
  bank_balance = bank_balance - amount;

}
```

Then write

Still a race

Read

Compute

# Critical Section Problem

```
bool lock = false;

int bank_balance = 300;

void withdraw(const string& msg, int amount) {

  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;


  bank_balance = bank_balance - amount;

}
```

Critical section

# Solution (?)

```
bool lock = false;

int bank_balance = 300;

void withdraw(const string& msg, int amount) {

  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;

  while (lock == true)
    ;
  lock = true;

  bank_balance = bank_balance - amount;

  lock = false;
}
```

Test if another thread is holding the lock

Spin if it is

Fall through when lock == false

Take the lock

Execute critical section

Release lock

# Solution (?)

```
bool lock = false;

int bank_balance = 300;

void withdraw(const string& msg, int amount) {

  string out_string = msg + " withdraws " + to_string(amount) + "\n";
  cout << out_string;

  while (lock == true)
    ;
  lock = true;

  bank_balance = bank_balance - amount;

  lock = false;
}
```

Common pattern (when correct)

Test if another thread is holding the lock

Spin if it is

Take the lock

Fall through when lock == false

Lock might be taken between the test and the set

We've traded one critical section problem for another

# Synchronization Hardware

- Many systems provide hardware support for critical section code
- Uniprocessors – could disable interrupts
  - Currently running code would execute without preemption
  - Generally too inefficient on multiprocessor systems
    - Operating systems using this not broadly scalable
- Modern machines provide special ***atomic*** hardware instructions
    - Atomic = non-interruptable
  - Either test memory word and set value
  - Or swap contents of two memory words

# Test and Set

```
bool TestAndSet (bool& target)
{
  bool rv = target;
  target = TRUE;
  return rv:
}
```

```
bool TestAndSet (bool *target)
{
  bool rv = *target;
  *target = TRUE;
  return rv:
}
```

These are the semantics, not the implementation

Implemented in hardware as an invisible instruction

# Compare And Swap

```
void CompareAndSwap (bool& a, bool& b)
{
  bool temp = a;
  a = b;
  b = temp;
}
```

```
void CompareAndSwap (bool *a, bool *b)
{
  bool temp = *a;
  *a = *b;
  *b = temp;
}
```

These are the semantics, not the implementation

Implemented in hardware as an invisible instruction

# Correct Withdraw

```
int bank_balance = 300;
bool lock = false;

void withdraw(const string& msg, int amount) {
  string out_s = msg + " withdraws " + to_string(amt) + "\n";
  cout << out_s;

  while (TestAndSet(lock) == true)
    ;

  bank_balance -= amount;

  lock = false;
}
```

Under what condition will we fall through?

What is the state of the lock?

Spin while the value is true (another thread holds the lock)

Release the lock

# Correct Withdraw

```
int bank_balance = 300;
bool lock = false;

void withdraw(const string& msg, int amount) {
  string out_s = msg + " withdraws " + to_string(amt) + "\n";
  cout << out_s;

  while (TestAndSet(lock) == true)
    ;

  bank_balance -= amount;

  lock = false;
}
```

What is the CPU doing?

How is it affecting other threads?

"Spin lock" (common pattern)

Is this a good programming abstraction?

# Multitasking on Multicor

Nonetheless, this is the essence of ***parallel*** computing

Parallel computation isn't done until all cores are done

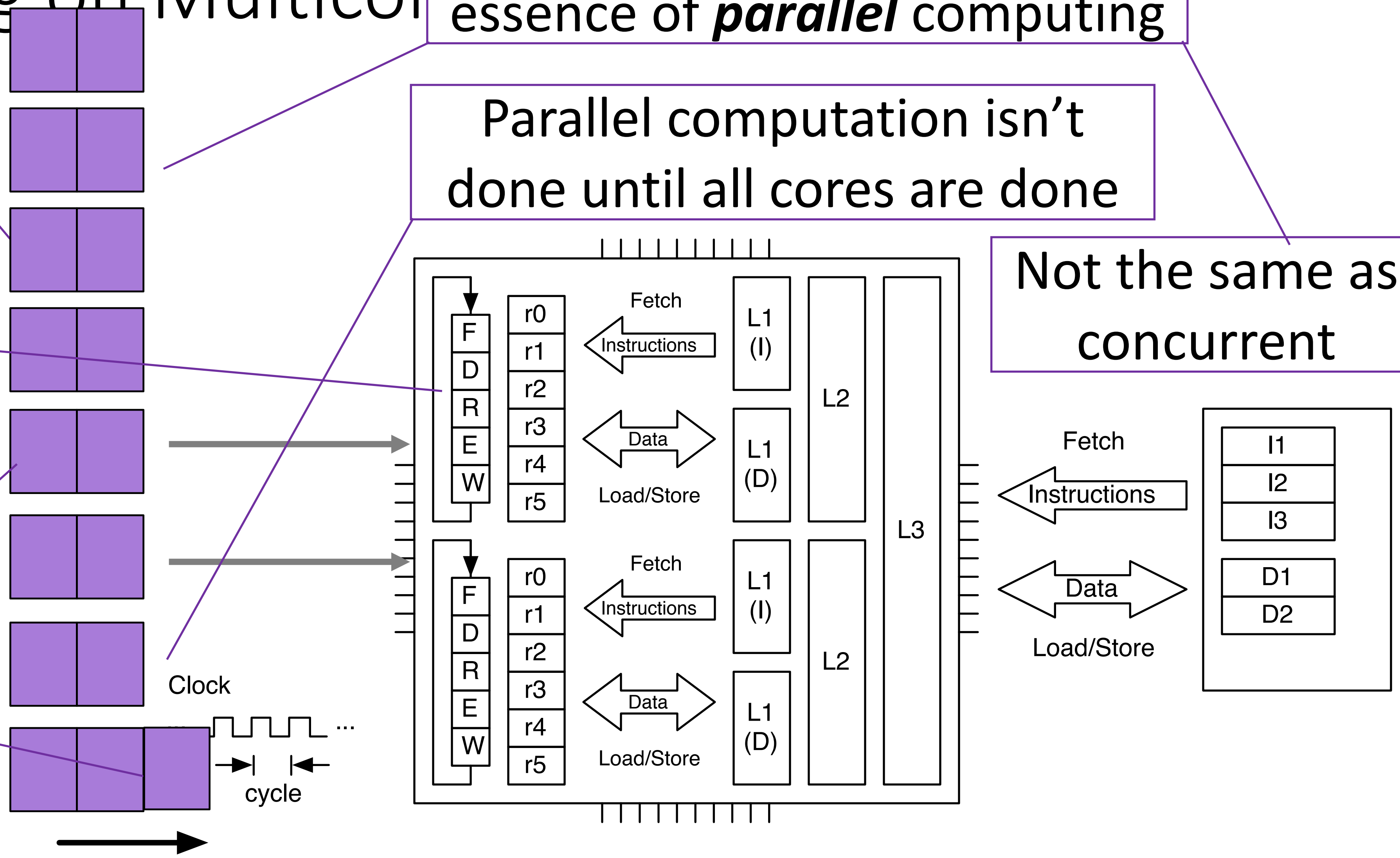
Not the same as concurrent

In 1/8 the time (?)

Need enough cores (8)

Work needs to be balanced

oops



# Numerical Quadrature Task

```
double partial_pi(unsigned long begin, unsigned long end, double h) {
  double partial_pi = 0.0;
  for (unsigned long i = begin; i < end; ++i) {
    partial_pi += 4.0 / (1.0 + (i*h*i*h));
  }
  return partial_pi;
}
```

Nothing remarkable about this function

Nothing remarkable about this function

# Performance

```
$ time ./taskpi 500000000 1
pi is approximately 3.14159
2.006u 0.006s 0:02.01 99.5%
```

CPU time | OS time | Elapsed time | Utilization

```
$ time ./taskpi 500000000 2
pi is approximately 3.14159
1.895u 0.008s 0:00.95 198.9%
```

CPU time | OS time | Elapsed time | Utilization

```
$ time ./taskpi 500000000 4
pi is approximately 3.14159
2.020u 0.007s 0:00.51 396.0%
```

CPU time | OS time | Elapsed time | Utilization

# Performance

```
$ time ./taskpi 500000000 1
pi is approximately 3.
2.006u 0.006s 0:02.01 99.5%

$ time ./taskpi 500000000 2
pi is approximately 3.14159
1.895u 0.008s 0:00.95 198.0%

$ time ./taskpi 500000000 4
pi is approximately 3.14159
2.020u 0.007s 0:00.51 396.
```

```
$ time ./taskpi 500000000 8
pi is approximately 3.14159
3.669u 0.008s 0:00.48 762.5%
```

```
$ time ./taskpi 500000000 16
pi is approximately 3.14159
3.659u 0.009s 0:00.48 760.4%
```

```
$ time ./taskpi 500000000 50000
pi is approximately 3.14159
2.963u 1.194s 0:00.92 451.0%
```

Elapsed time

Utilization

OS time

CPU time

Too many threads

# Parallel Speedup, Parallel Efficiency

***Speedup*** on p processing units

Time to run problem size n on one PU

Time to run problem size n on p PUs

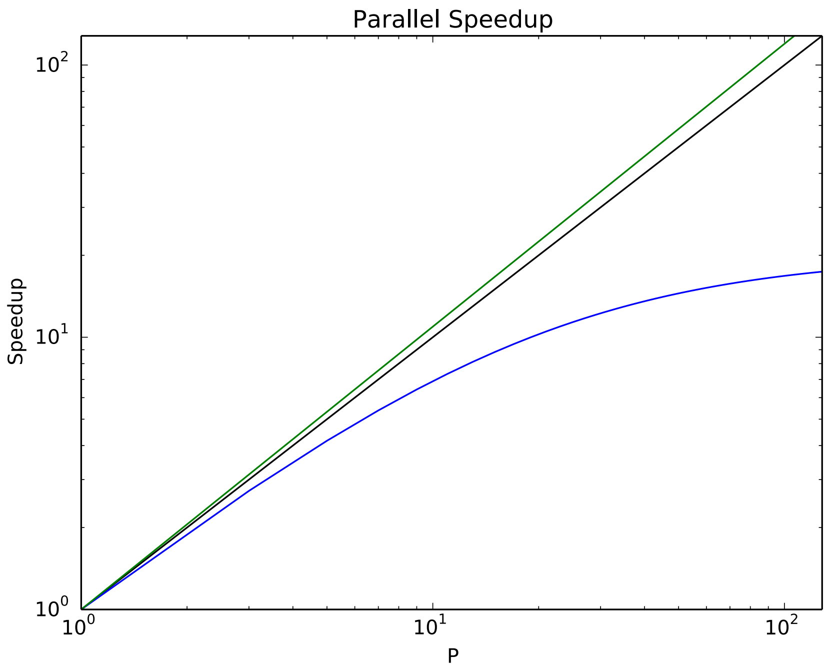
$$S(p) = \frac{T(n,1)}{T(n,p)}$$

***Efficiency*** on p processing units

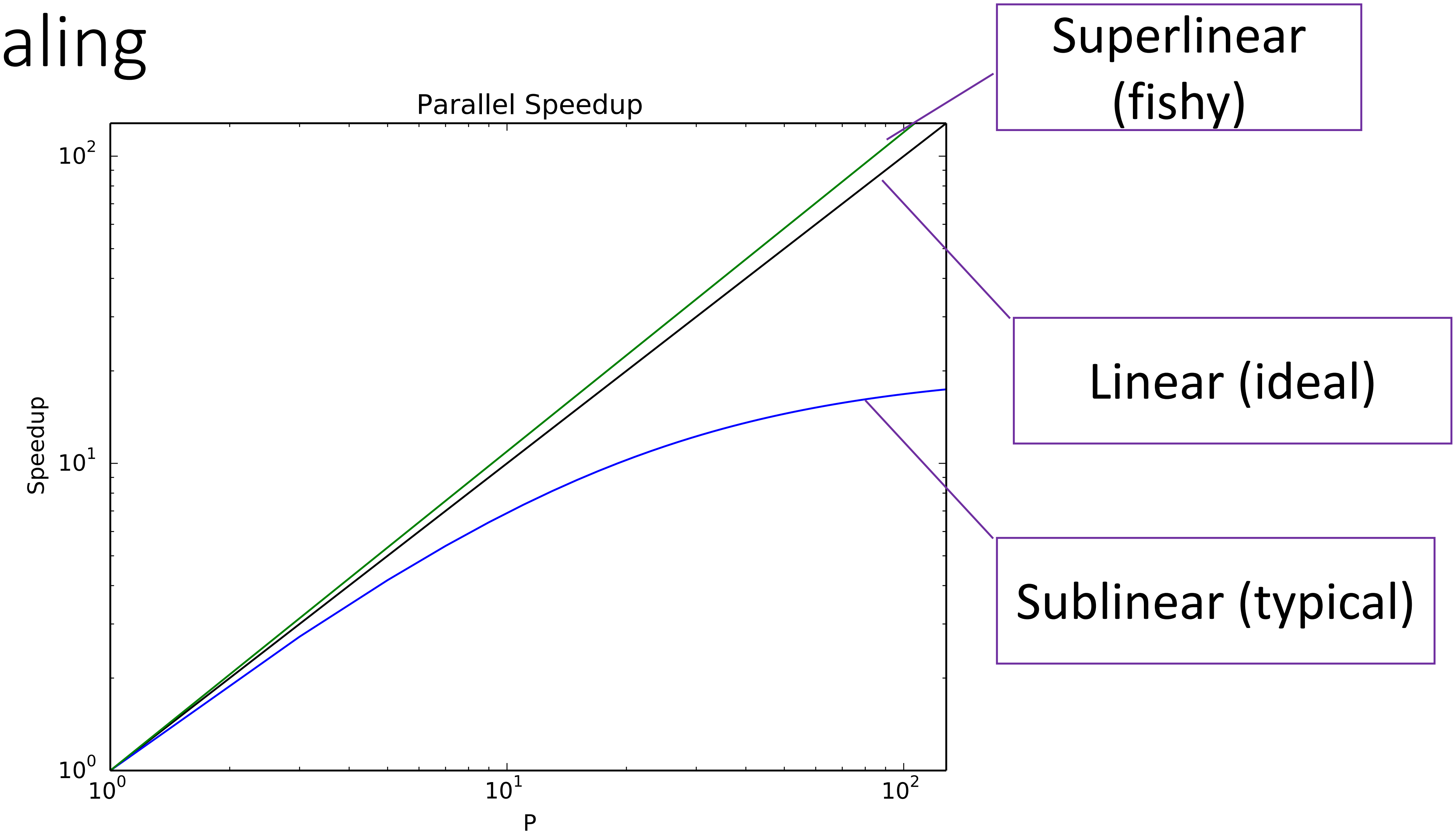
Ideal parallel execution time

Divided by actual parallel execution time

$$E(p) = \frac{T(n,1)/p}{T(n,p)} = \frac{T(n,1)/T(n,p)}{p} = \frac{S(p)}{p}$$



# Scaling


Parallel Speedup
Speedup
P
$10^0$
$10^1$
$10^2$
Superlinear (fishy)
Linear (ideal)
Sublinear (typical)

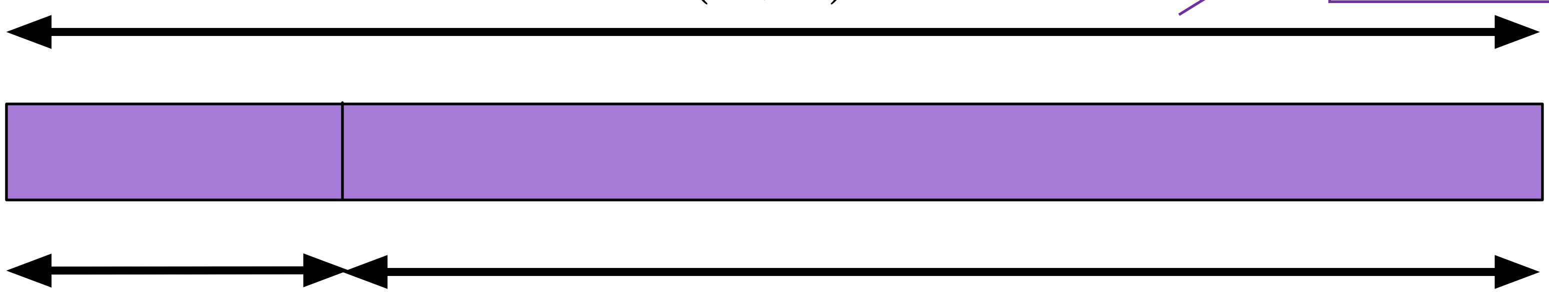
# Name This Famous Person

"Validity of the single processor approach to achieving large-scale computing capabilities," AFIPS Conference Proceedings (30): 483–485, 1967.

Gene Amdahl (1922-2015)

Amdahl's Law

# Limits to Parallelism (Amdahls's Law)




$$T(n,1) = \alpha T(n,1) + (1-\alpha)T(n,1)$$

Inherently sequential

Perfectly parallelizable

$$\begin{aligned} T(n,p) &= \alpha T(n,1) + \tfrac{1}{p}(1-\alpha)T(n,1) \\ &= T(n,1)(\alpha + \tfrac{1}{p}(1-\alpha)) \end{aligned}$$

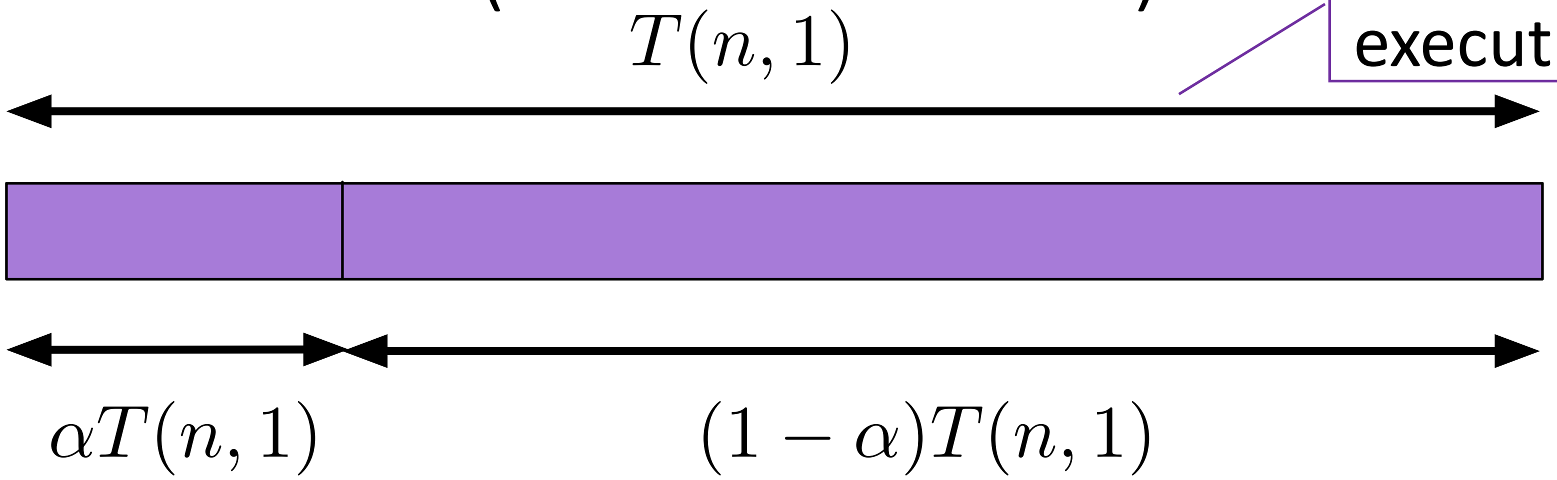
Sequential portion

Ideal speedup (for parallelizable portion)

# Limits to Parallelism (Amdahls's Law)

Sequential execution time

Inherently sequential




Speedup

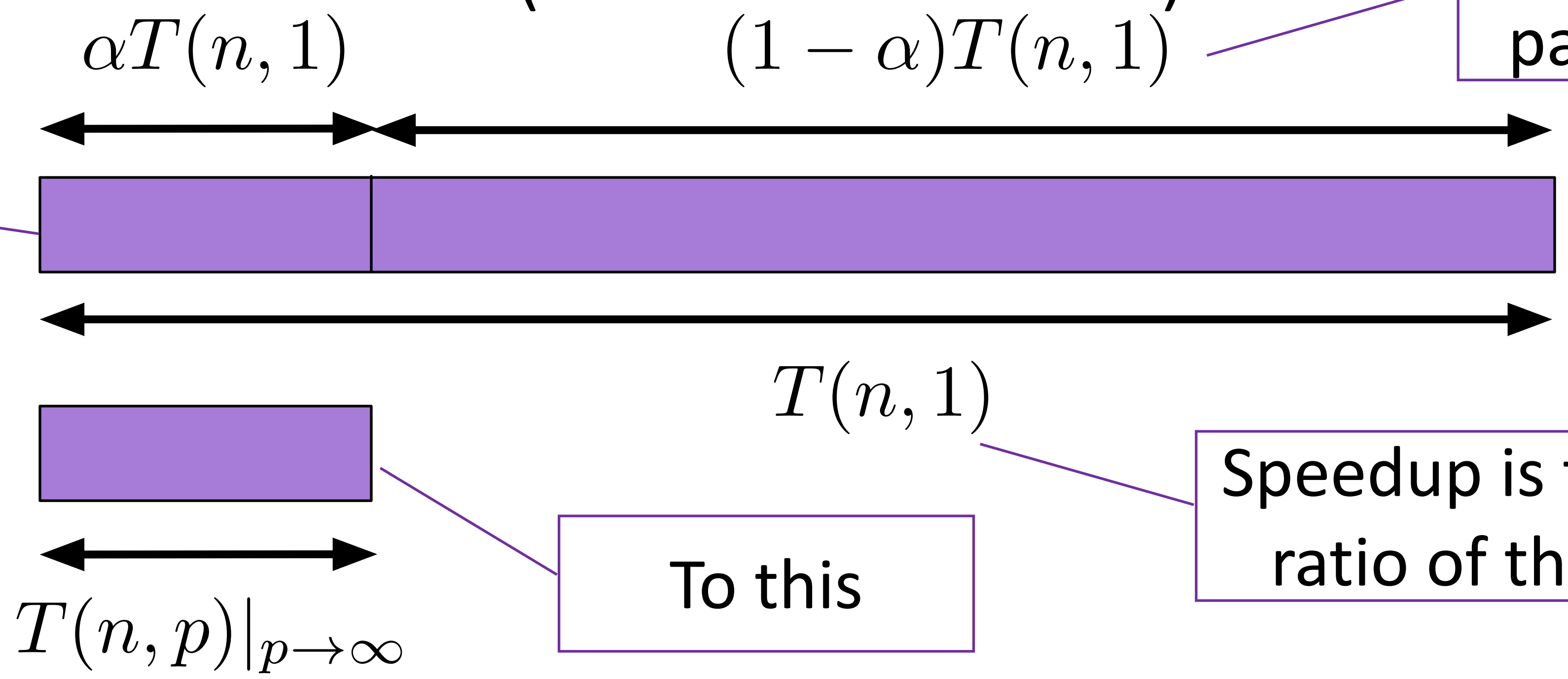
Perfectly parallelizable

$$S(p) = \frac{T(n,1)}{T(n,p)} = \frac{T(n,1)}{T(n,1)[\alpha + \frac{1}{p}(1-\alpha)]}$$

$$= \frac{1}{\alpha + \frac{1}{p}(1-\alpha)} \leq \frac{1}{\alpha}$$

$$\lim_{p\to\infty} S(p) = \frac{1}{\alpha}$$

# Limits to Parallelism (Amdahls's Law)




Inherently sequential

Perfectly parallelizable

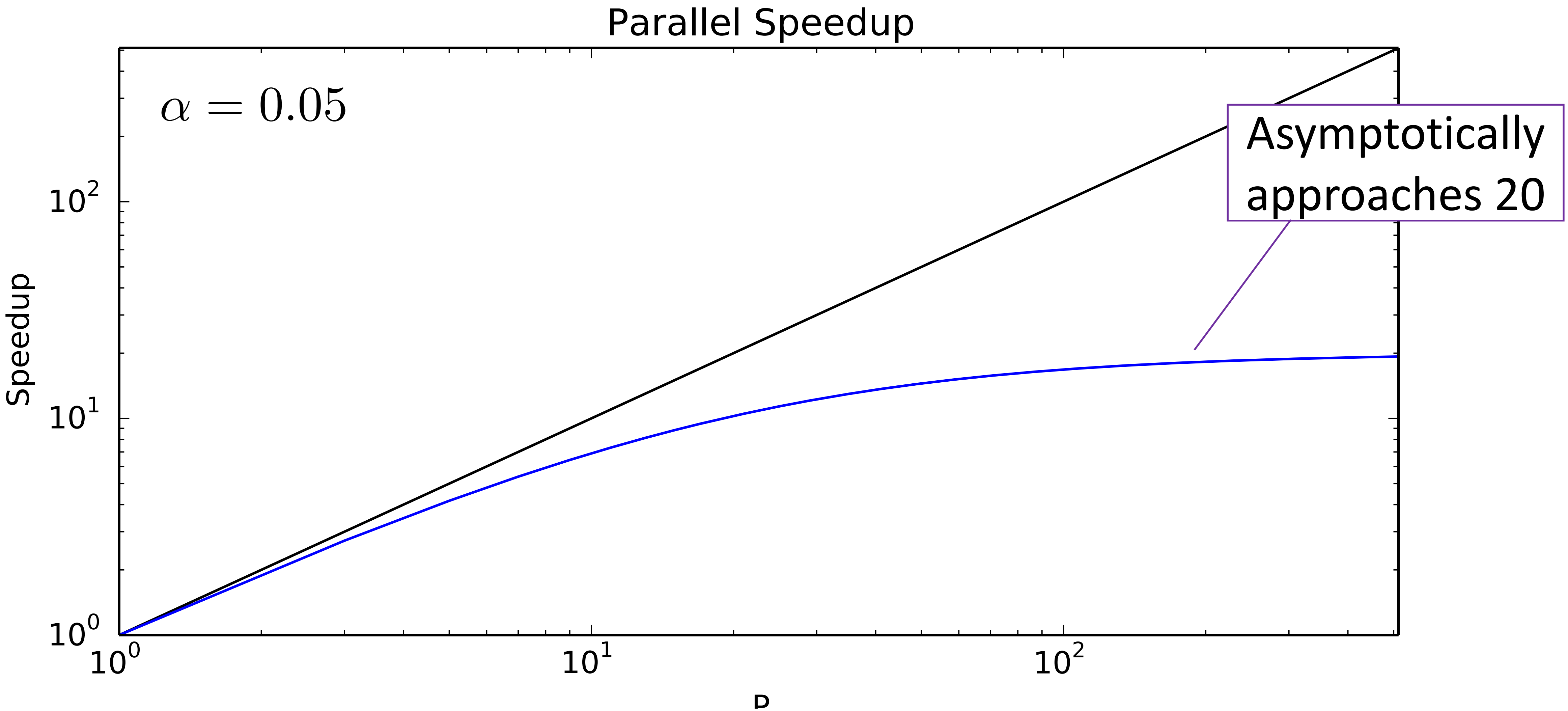
Speedup is the ratio of this

To this

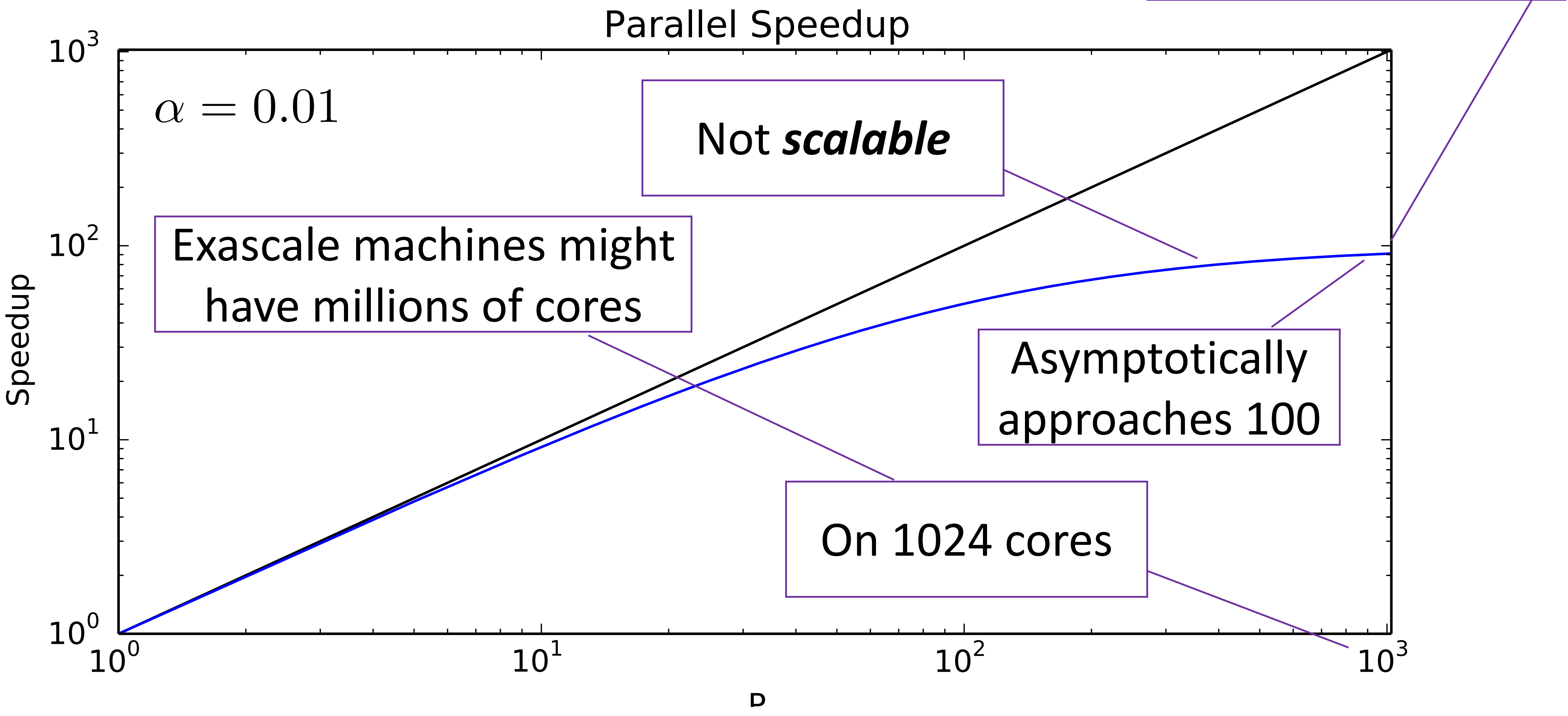
$$S(p) = \frac{T(n,1)}{T(n,p)}$$

$$\lim_{p\to\infty} S(p) = \frac{1}{\alpha}$$

# Limits to Parallelism (Amdahl's Law)

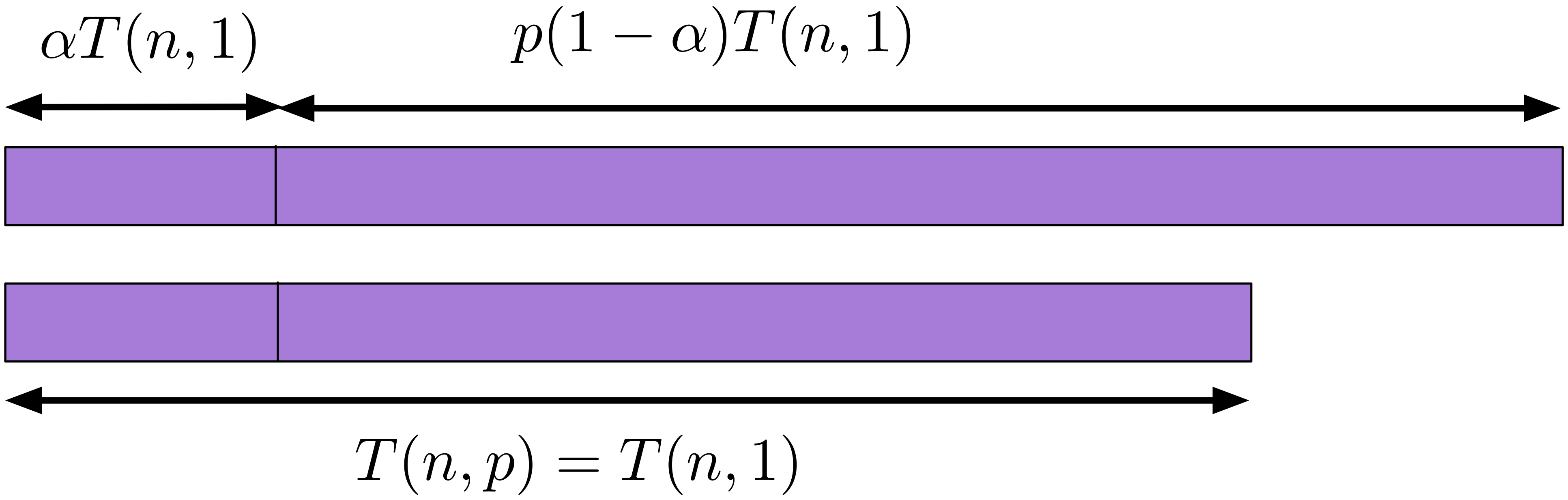


# Limits to Parallelism



# There are no Limits (Gustafson's Law)

- Doing the same problem faster and faster is not how we use parallel computers
- Rather, we solve bigger and more difficult problems
- I.e., the amount of parallelizable work grows



# There are no Limits (Gustafson's Law)

$\alpha T(n,1)$ $\quad$ $p(1-\alpha)T(n,1)$

Perfectly parallelizable

$T(n,p) = T(n,1)$

Parallel performance

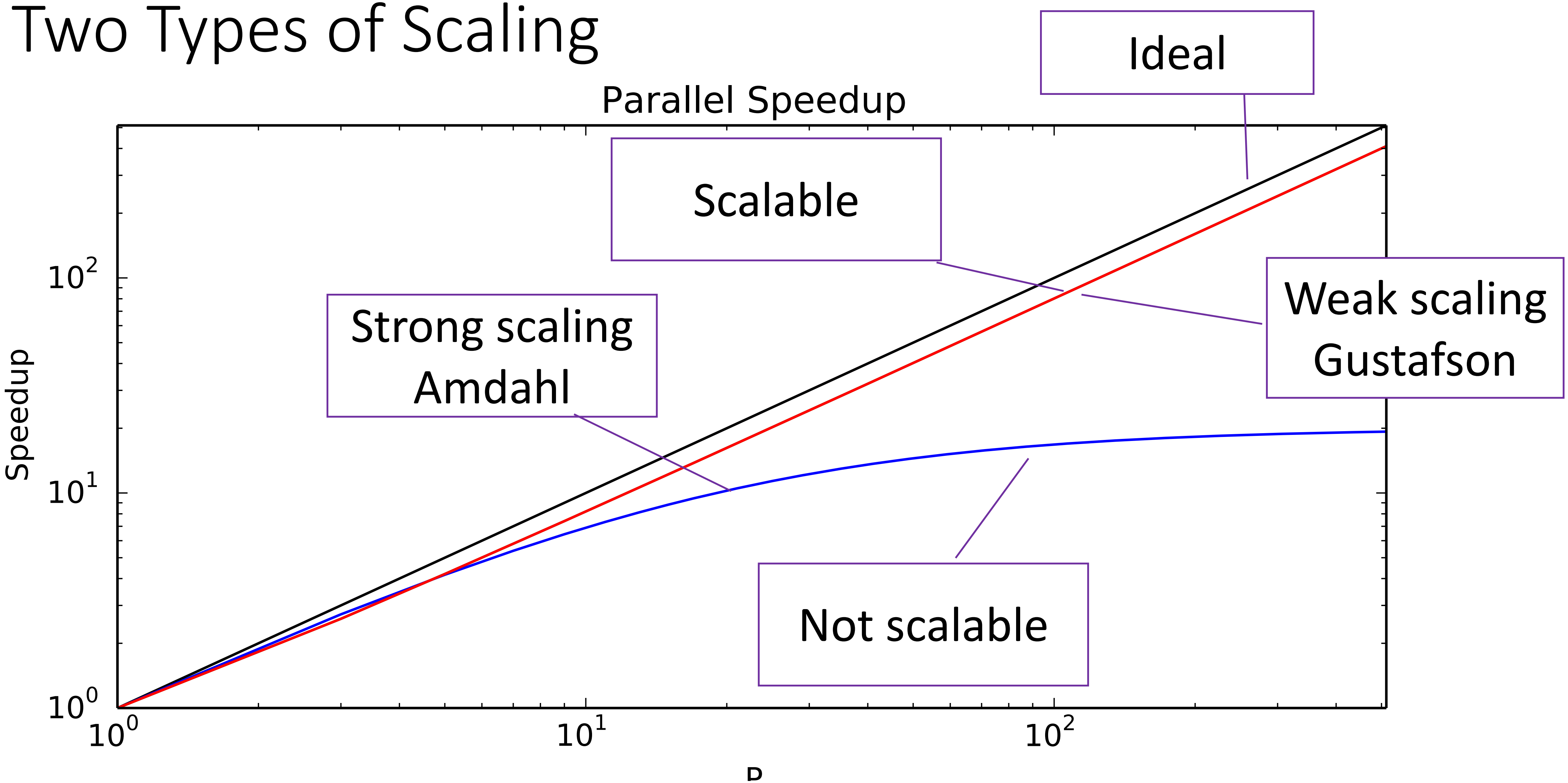
Ratio of non speed up to speed up

$$S(p) = \frac{\alpha T(n,1) + p(1-\alpha)T(n,1)}{T(n,p)} = \frac{\alpha T(n,1) + p(1-\alpha)T(n,1)}{T(n,1)} = \alpha + p(1-\alpha)$$

$$E(p) = \frac{S(p)}{p}$$

$$\lim_{p\to\infty} E(p) = (1-\alpha)$$

# Two Types of Scaling



# Stay Tuned

- C++ threads
- C++ async()
- C++ atomics

# Thank you!

# Creative Commons BY-NC-SA 4.0 License