Introduction to concurrent programming

A motivating example

As simple as counting to two

We illustrate the challenges introduced by concurrent programming on a simple example: a counter modeled by a Java class.

- First, we write a traditional, sequential version
- Then, we introduce concurrency and… run into trouble!

Sequential counter

```java
public class Counter {
  private int counter = 0;

  // increment counter by one
  public void run() {
    int cnt = counter;
    counter = cnt + 1;
  }

  // current value of counter
  public int counter() {
    return counter;
  }
}
```

```java
public class SequentialCount {
  public static
  void main(String[] args) {
    Counter counter = new Counter();
    counter.run(); // increment once
    counter.run(); // increment twice
    // print final value of counter
    System.out.println(
       counter.counter());
  }
}
```

- What is printed by running: `java SequentialCount`?
- May the printed value change in different reruns?

Modeling sequential computation

```
5   public void run() {
6     int cnt = counter;
7     counter = cnt + 1;
8   }
```

```
counter.run(); // first call: steps 1-3
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: $\perp$	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: $\perp$	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Modeling sequential computation

```
5   public void run() {
6     int cnt = counter;
7     counter = cnt + 1;
8   }
```

```
counter.run(); // first call: steps 1-3
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: ⊥	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: ⊥	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Modeling sequential computation

```
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8    }
```

```
counter.run(); // first call: steps 1-3
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: $\perp$	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: $\perp$	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Modeling sequential computation

```
5   public void run() {
6     int cnt = counter;
7     counter = cnt + 1;
8   }
```

```
counter.run(); // first call: steps 1-3
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: ⊥	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: ⊥	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Modeling sequential computation

```
5   public void run() {
6     int cnt = counter;    ●
7     counter = cnt + 1;
8   }
```

```
  counter.run(); // first call: steps 1-3
● counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: ⊥	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: ⊥	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Modeling sequential computation

```java
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8    }
```

```java
counter.run(); // first call: steps 1-3
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: $\perp$	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: $\perp$	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Modeling sequential computation

```java
5   public void run() {
6     int cnt = counter;
7     counter = cnt + 1;
8   }
```

```java
counter.run(); // first call: steps 1-3
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	`cnt`: $\perp$	`counter`: 0
2	pc: 7	`cnt`: 0	`counter`: 0
3	pc: 8	`cnt`: 0	`counter`: 1
4	pc: 6	`cnt`: $\perp$	`counter`: 1
5	pc: 7	`cnt`: 1	`counter`: 1
6	pc: 8	`cnt`: 1	`counter`: 2
7	done		`counter`: 2

Modeling sequential computation

```
5   public void run() {
6     int cnt = counter;
7     counter = cnt + 1;
8   }
```

```
counter.run(); // first call: steps 1-3
counter.run(); // second call: steps 4-6
```

#	LOCAL STATE		OBJECT STATE
1	pc: 6	cnt: $\perp$	counter: 0
2	pc: 7	cnt: 0	counter: 0
3	pc: 8	cnt: 0	counter: 1
4	pc: 6	cnt: $\perp$	counter: 1
5	pc: 7	cnt: 1	counter: 1
6	pc: 8	cnt: 1	counter: 2
7	done		counter: 2

Adding concurrency

Now, we revisit the example by introducing concurrency:

> Each of the two calls to method `run` can be executed in parallel

In Java, this is achieved by using threads. Do not worry about the details of the syntax for now, we will explain it later.

The idea is just that:

- There are two independent execution units (threads) `t` and `u`
- Each execution unit executes `run` on the same `counter` object
- We have no control over the order of execution of `t` and `u`

Concurrent counter

```java
public class CCounter
  extends Counter
  implements Runnable
{
  // threads
  // will execute
  // run()
}
```

```java
public class ConcurrentCount {
  public static void main(String[] args) {
    CCounter counter = new CCounter();
    // threads t and u, sharing counter
    Thread t = new Thread(counter);
    Thread u = new Thread(counter);
    t.start(); // increment once
    u.start(); // increment twice
    try { // wait for t and u to terminate
      t.join(); u.join(); }
    catch (InterruptedException e)
    { System.out.println("Interrupted!"); }
    // print final value of counter
    System.out.println(counter.counter());
} }
```

- What is printed by running: `java ConcurrentCount`?
- May the printed value change in different reruns?

What?!

```
$ javac Counter.java CCounter.java ConcurrentCount.java
$ java ConcurrentCount.java
2
$ java ConcurrentCount.java
2
...
$ java ConcurrentCount.java
1
$ java ConcurrentCount.java
2
```

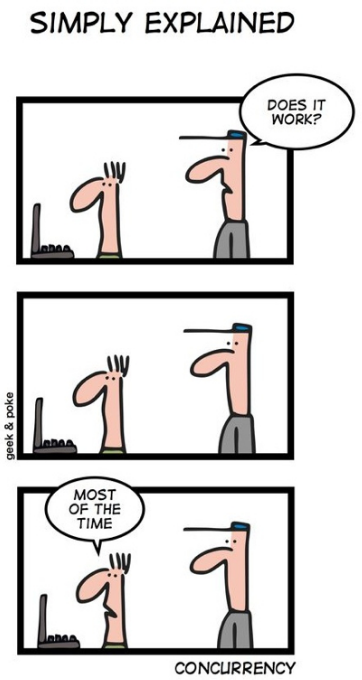
The concurrent version of counter occasionally prints 1 instead of the expected 2. It seems to do so unpredictably.

What?!

```
$ javac Counter.java CCounter.java ConcurrentCount.java
$ java ConcurrentCount.java
2
$ java ConcurrentCount.java
2
...
$ java ConcurrentCount.java
1
$ java ConcurrentCount.java
2
```

The concurrent version of counter occasionally prints 1 instead of the expected 2. It seems to do so unpredictably.

Welcome to concurrent programming!


SIMPLY EXPLAINED
DOES IT WORK?
geek & poke
MOST OF THE TIME
CONCURRENCY

Why concurrency?

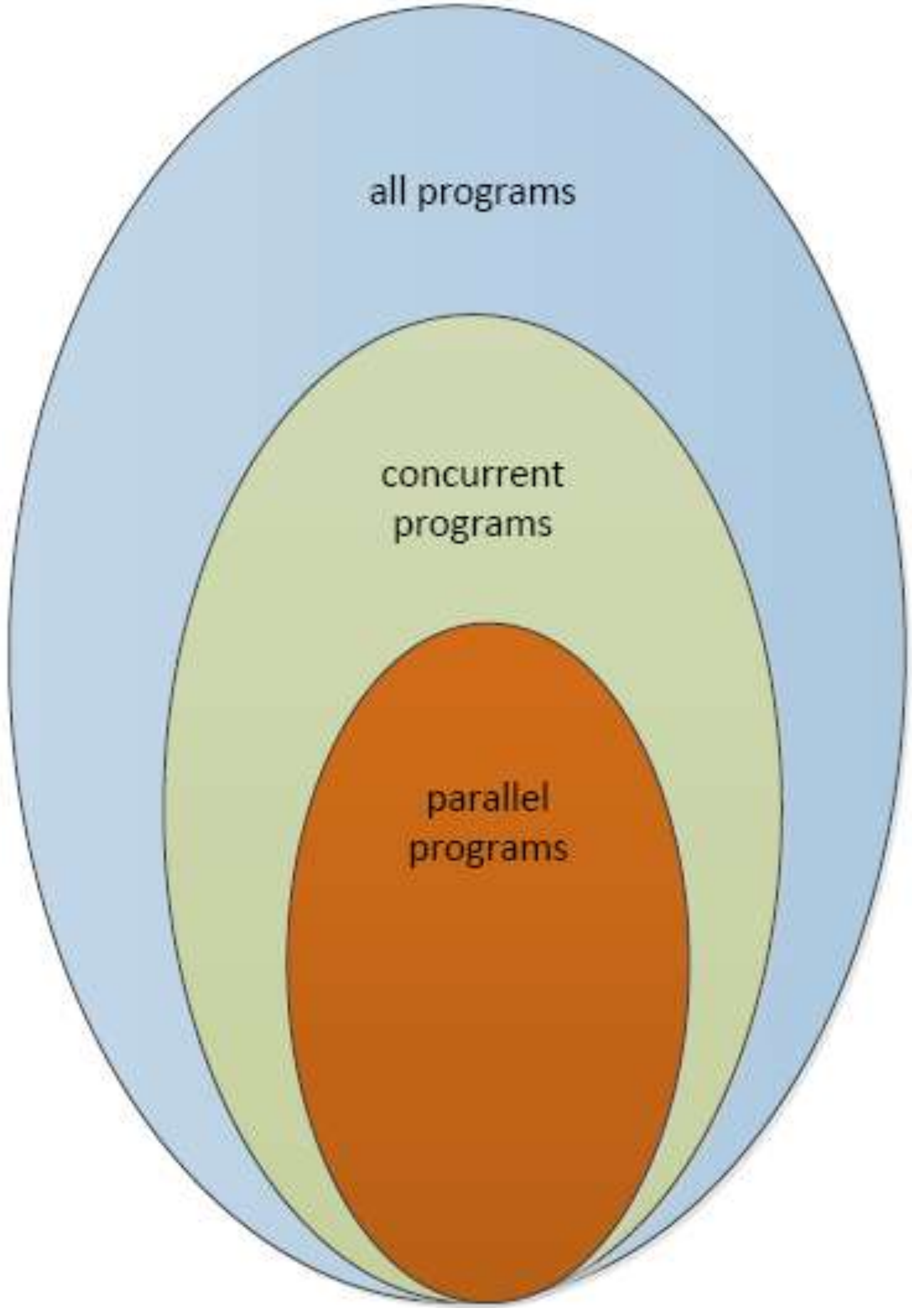
Reasons for using concurrency

Why do we need concurrent programming in the first place?

abstraction: separating different tasks, without worrying about when to execute them (example: download files from two different websites)

responsiveness: providing a responsive user interface, with different tasks executing independently (example: browse the slides while downloading your email)

performance: splitting complex tasks in multiple units, and assign each unit to a different processor (example: compute all prime numbers up to 1 billion)


all programs
concurrent programs
parallel programs

Concurrency vs. parallelism

In this course we will mostly use concurrency and parallelism as synonyms. However, they refer to similar but different concepts:

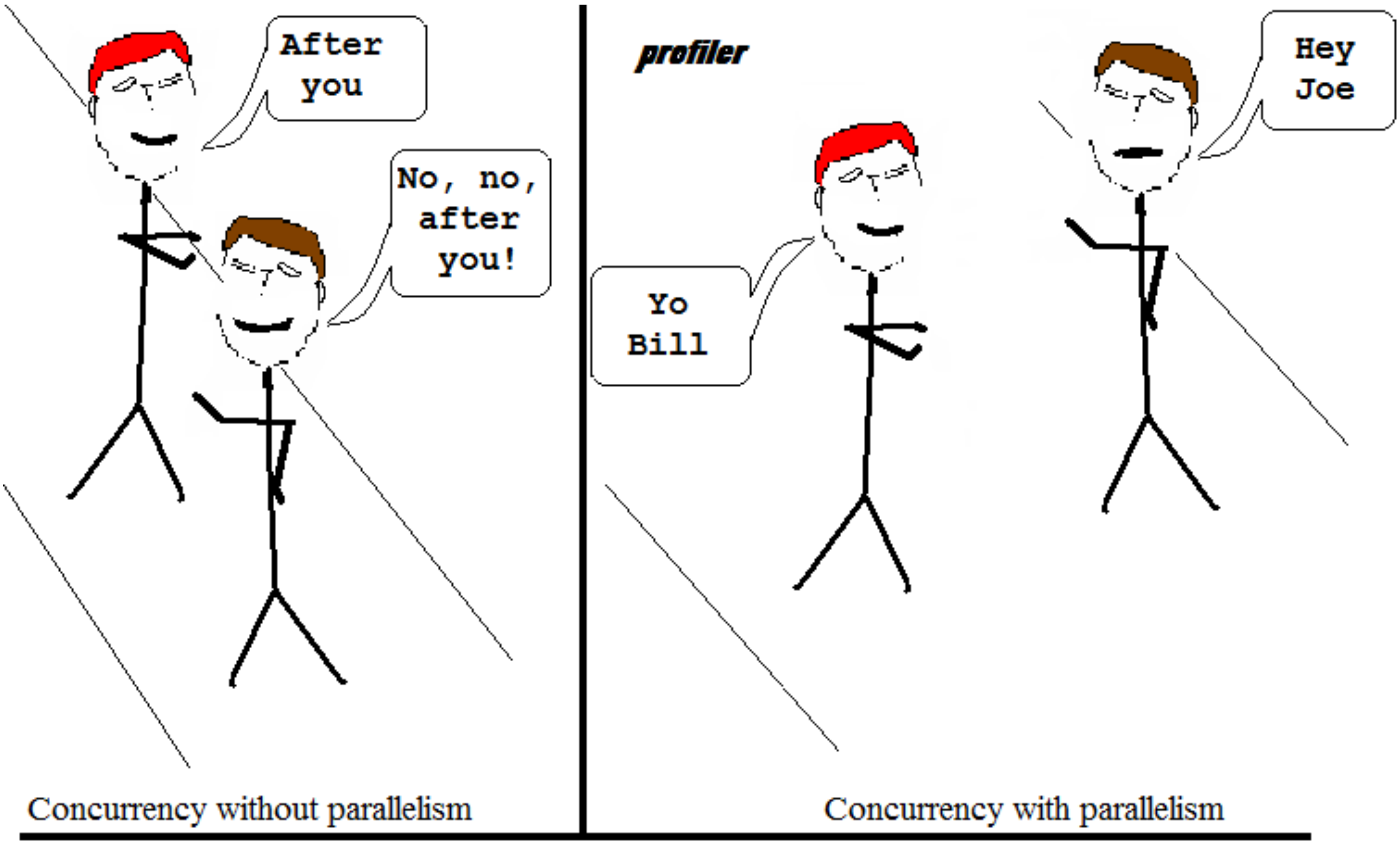
concurrency: nondeterministic composition of independently executing units (logical parallelism)

parallelism: efficient execution of fractions of a complex task on multiple processing units (physical parallelism)

- You can have concurrency without physical parallelism: operating systems running on single-processor single-core systems
- Parallelism is mainly about speeding up computations by taking advantage of redundant hardware

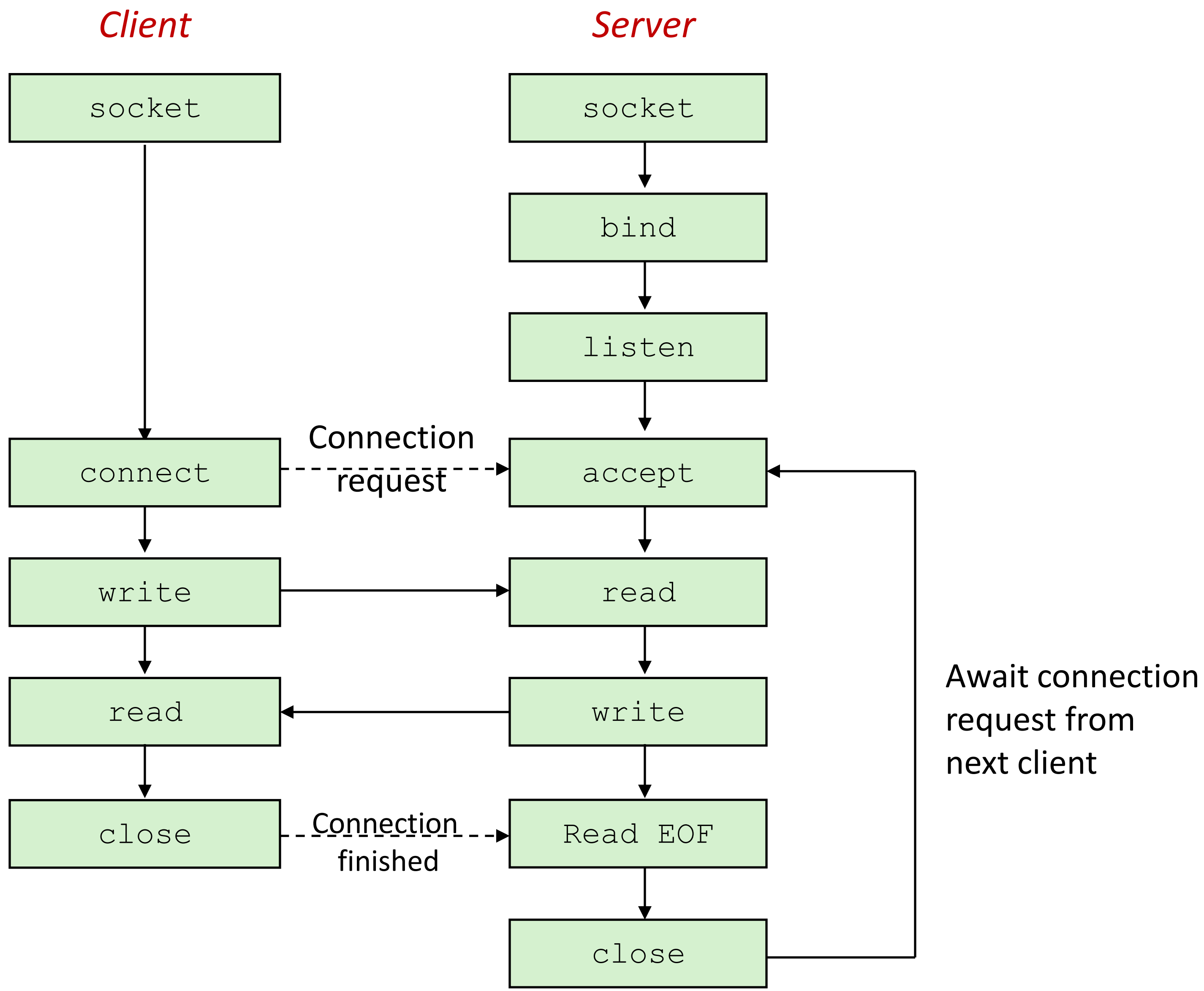
Same Meaning?

- Concurrency: At least two tasks are making progress at the same time frame.
  - Not necessarily at the same time
  - Include techniques like time-slicing
  - Can be implemented on a single processing unit
  - Concept more general than parallelism
- Parallelism: At least two tasks execute *literally* at the same time.
  - Requires hardware with multiple processing units



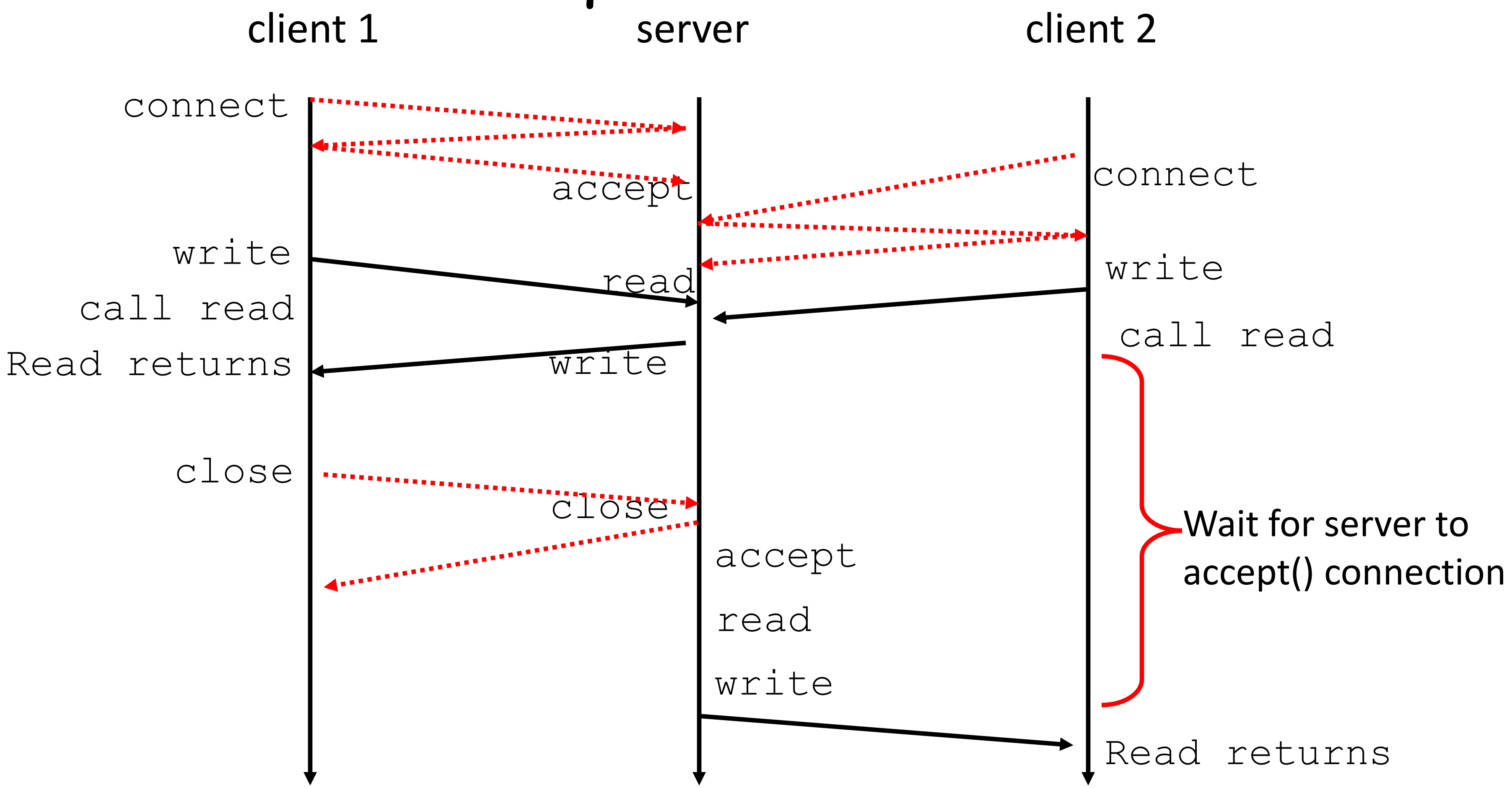
Performance tuning technique number 106: Concurrency vs. Parallelism

Sequential Echo Server



Sequential Echo Server

- Process one request at a time



Why not Sequential Servers?

- Increased latency
  - client2 must wait for client1 to finish before getting served
- Low utilization
  - Server is idle while waiting for client1's requests. It could have served another client during those idle times!
- Solution: implement *concurrent servers*
  - serve multiple clients at the same time

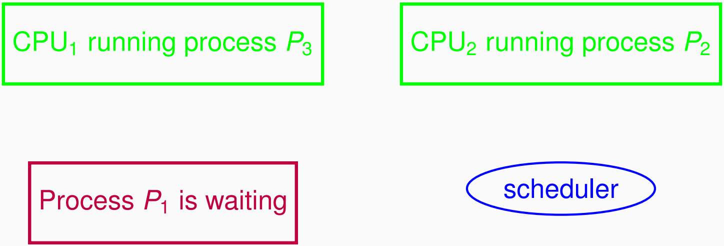
Basic terminology and abstractions

Processes

A process is an independent unit of execution – the abstraction of a running sequential program:

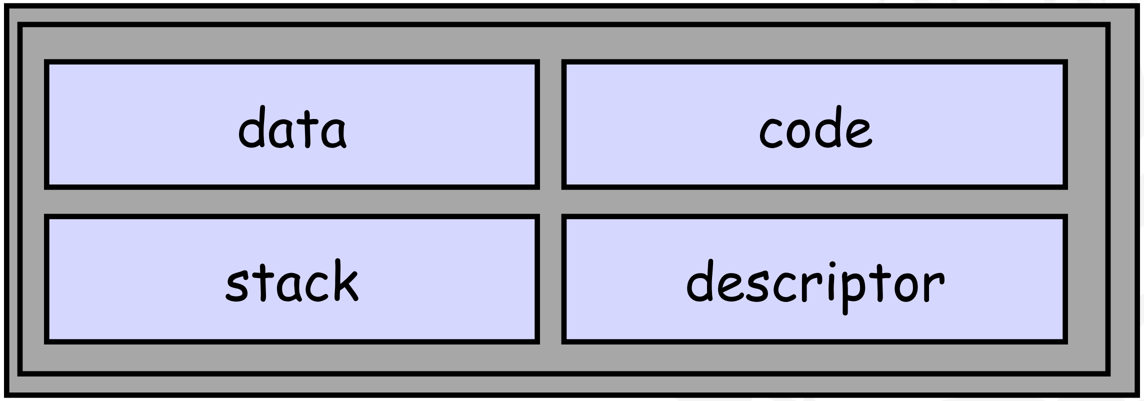
- identifier
- program counter
- memory space

The runtime/operating system schedules processes for execution on the available processors:



One Process

- Process:



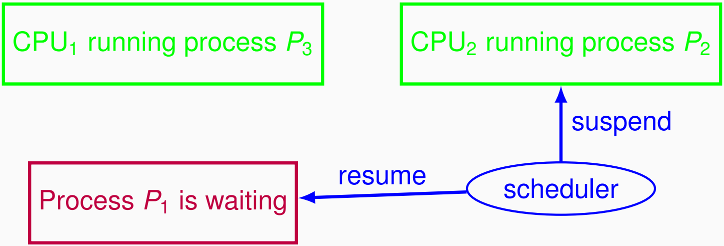
- Data: The heap (global, heap allocated data)
- Code: The program (bytecode)
- Stack: The stack (local data, call stack)
- Descriptor: Program counter, stack pointer, …

Processes

A process is an independent unit of execution – the abstraction of a running sequential program:

- identifier
- program counter
- memory space

The runtime/operating system schedules processes for execution on the available processors:



Processes

A process is an independent unit of execution – the abstraction of a running sequential program:

- identifier
- program counter
- memory space

The runtime/operating system schedules processes for execution on the available processors:

CPU_1 running process P_3

CPU_2 running process P_1

Process P_2 is waiting

scheduler

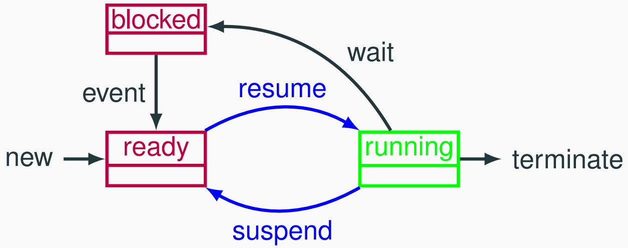
Process states

The scheduler is the system unit in charge of setting process states:

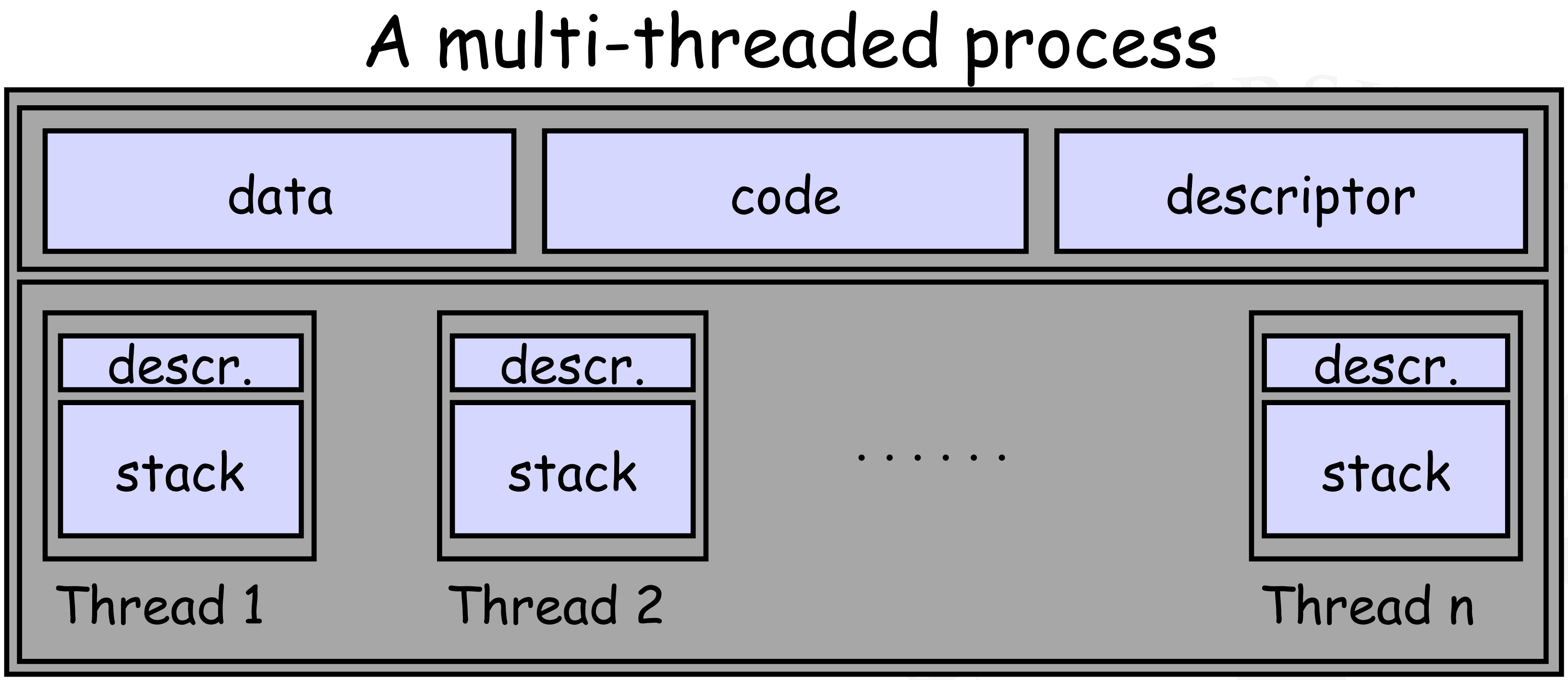
ready: ready to be executed, but not allocated to any CPU

blocked: waiting for an event to happen

running: running on some CPU



Implementing processes - the OS view

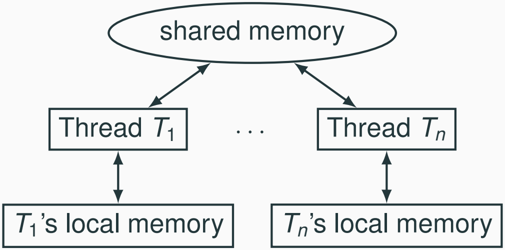


A (heavyweight) process in an operating system is represented by its code, data and the state of the machine registers, given in a descriptor. In order to support multiple (lightweight) **threads of control**, it has multiple stacks, one for each thread.

Threads

A thread is a lightweight process – an independent unit of execution on the same program space:

- identifier
- program counter
- memory
  - local memory, separate for each thread
  - global memory, shared with other threads



In practice, the difference between processes and threads is fuzzy and implementation dependent. Normally in this course:

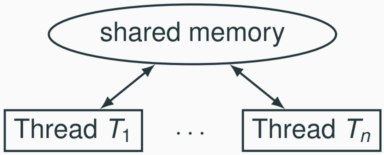
processes: executing units that do not share memory (in Erlang)

threads: executing units that share memory (in Java)

Shared memory vs. message passing

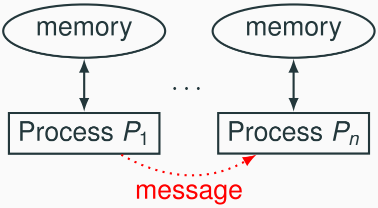
Shared memory models:

- communication by writing to shared memory
- e.g. multi-core systems



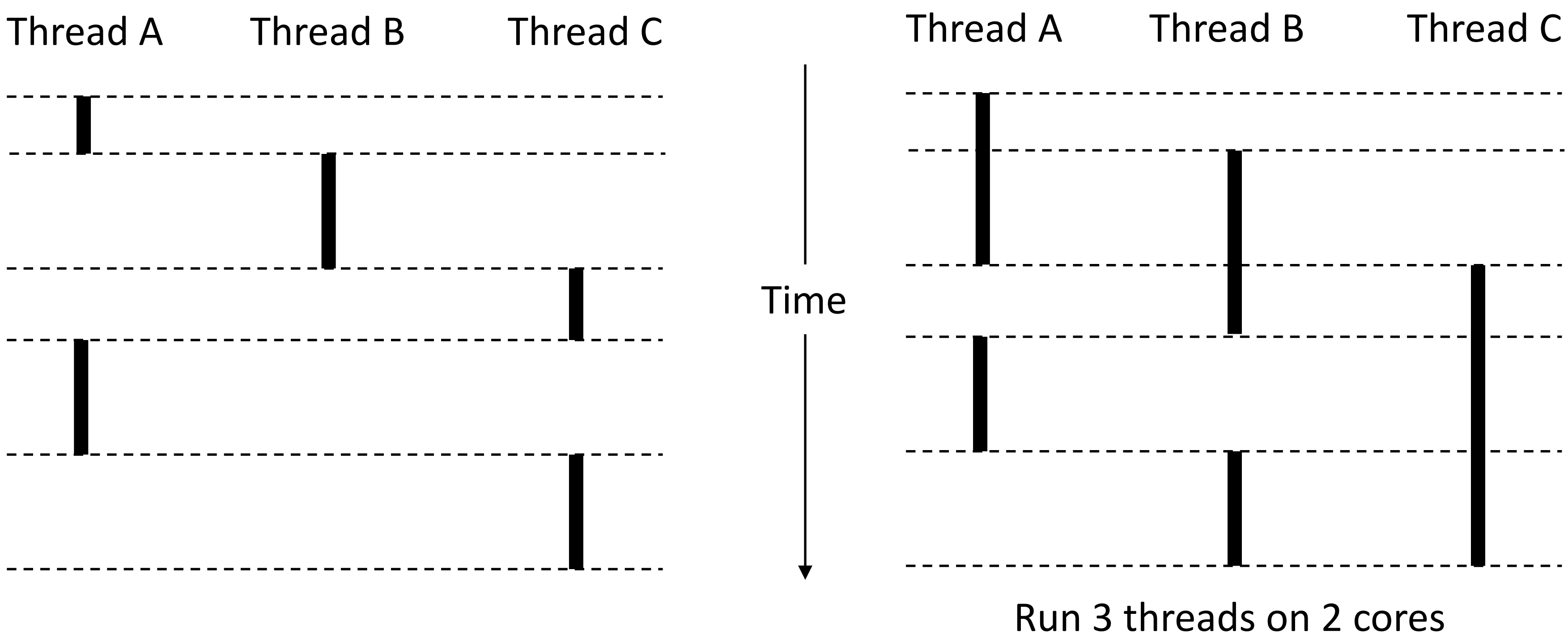
Distributed memory models:

- communication by message passing
- e.g. distributed systems



Thread Execution

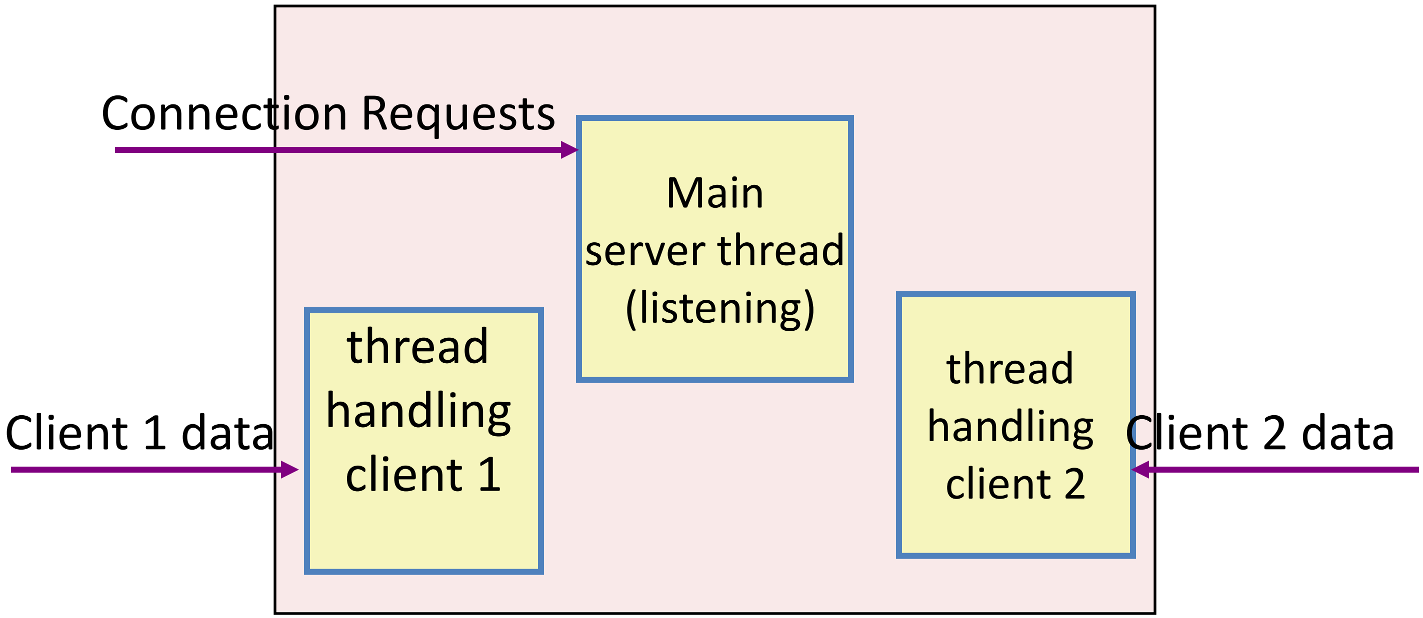
- Single Core Processor
  - Simulate concurrency by time slicing
- Multi-Core rocessor
  - true concurrency



Threads vs. Processes

- How threads and processes are similar
  - Each has its own logical control flow
  - Each can run concurrently with others (possibly on different cores)
  - Each is context switched
- How threads and processes are different
  - Threads share code and some data
    - Processes (typically) do not
  - Threads are less expensive than processes

Threaded Execution Model



– Multiple threads within single process

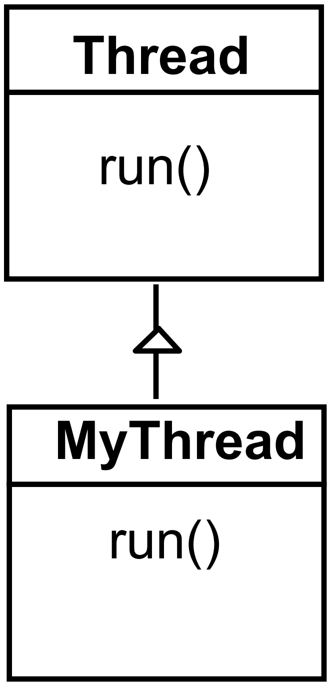
Pros and Cons of Thread-Based Designs

- \+ Easy to share data structures between threads
  - e.g., logging information, file cache.
- \+ Threads are more efficient than processes.
- – Unintentional sharing can introduce subtle race errors!

Java threads

Threads in Java

A Thread class manages a single sequential thread of control. Threads may be created and deleted dynamically.



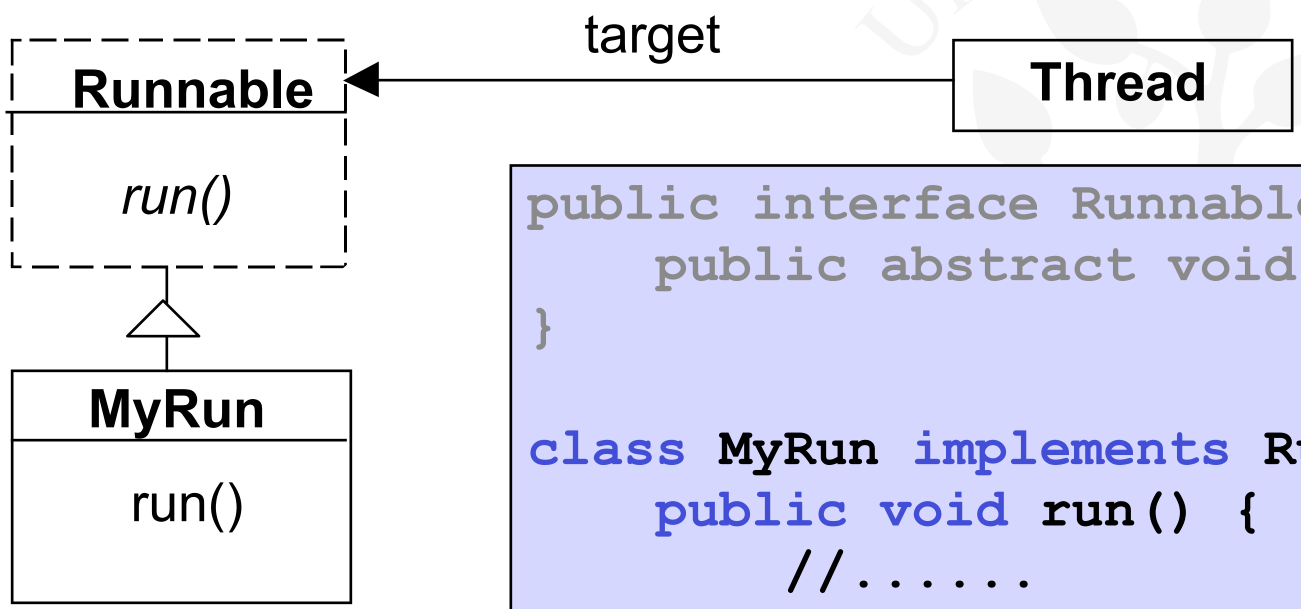
The Thread class executes instructions from its method run(). The actual code executed depends on the implementation provided for run() in a derived class.

```
class MyThread extends Thread {
      public void run() {
            //......
      }
}
```

```
Thread x = new MyThread();
```

Threads in Java (cont'd)

Since Java does not permit multiple inheritance, we often implement the **run()** method in a class not derived from Thread but from the interface Runnable.




```java
public interface Runnable {
    public abstract void run();
}

class MyRun implements Runnable {
    public void run() {
        //......
    }
}
```

```java
Thread x = new Thread(new MyRun());
```

Java threads

Two ways to build multi-threaded programs in Java:

- inherit from class `Thread`, override method `run`
- implement interface `Runnable`, implement method `run`

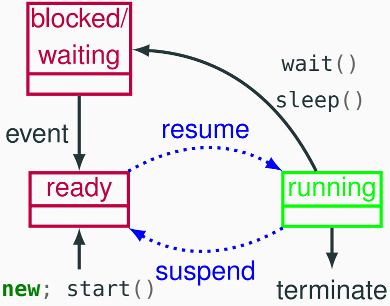
```java
public class CCounter
  implements Runnable
{
   // thread's computation:
   public void run() {
     int cnt = counter;
     counter = cnt + 1;
   }
}
```

```java
CCounter c = new CCounter();

Thread t = new Thread(c);
Thread u = new Thread(c);

t.start();
u.start();
```

States of a Java thread

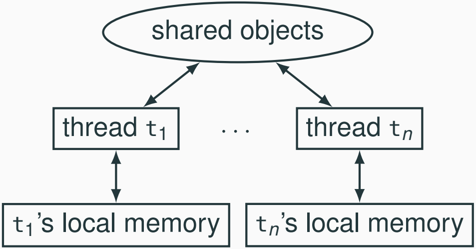


Resuming and suspending is done by the JVM scheduler, outside the program's control

For a thread object `t`:

- `t.start()`: the thread is ready for execution
- `t.sleep(n)`: block the thread for `n` milliseconds (correct timing depends on JVM implementation)
- `t.wait()`: block the thread until an event occurs
- `t.join()`: block the current thread until `t` terminates

Thread execution model



Shared vs. thread-local memory:

- shared objects: the object on which the thread operate, and all reachable objects
- local memory: local variables, and special thread-local attributes

Threads proceed asynchronously, so they have to coordinate with other threads accessing the same shared objects.

One possible execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2    int counter = 0;     // shared object state
3
4    // thread's computation:
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter: 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 1
4	done	pc_u : 6 cnt_u : $\perp$	counter: 1
5	done	pc_u : 7 cnt_u : 1	counter: 1
6	done	pc_u : 8 cnt_u : 1	counter: 2
7	done	done	counter: 2

One possible execution of the concurrent counter

```
1  public class CCounter implements Runnable {
2    int counter = 0;     // shared object state
3
4    // thread's computation:
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	$pc_t : 6$ $cnt_t : \bot$	$pc_u : 6$ $cnt_u : \bot$	counter: 0
2	$pc_t : 7$ $cnt_t : 0$	$pc_u : 6$ $cnt_u : \bot$	counter: 0
3	$pc_t : 8$ $cnt_t : 0$	$pc_u : 6$ $cnt_u : \bot$	counter: 1
4	done	$pc_u : 6$ $cnt_u : \bot$	counter: 1
5	done	$pc_u : 7$ $cnt_u : 1$	counter: 1
6	done	$pc_u : 8$ $cnt_u : 1$	counter: 2
7	done	done	counter: 2

One possible execution of the concurrent counter

```java
1 public class CCounter implements Runnable {
2    int counter = 0;     // shared object state
3
4    // thread's computation:
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8 } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter: 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 1
4	done	pc_u : 6 cnt_u : $\perp$	counter: 1
5	done	pc_u : 7 cnt_u : 1	counter: 1
6	done	pc_u : 8 cnt_u : 1	counter: 2
7	done	done	counter: 2

One possible execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2    int counter = 0;      // shared object state
3
4    // thread's computation:
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8  } }
```

#	t's LOCAL	u's LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter: 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 1
4	done	pc_u : 6 cnt_u : $\perp$	counter: 1
5	done	pc_u : 7 cnt_u : 1	counter: 1
6	done	pc_u : 8 cnt_u : 1	counter: 2
7	done	done	counter: 2

One possible execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2     int counter = 0;     // shared object state
3
4     // thread's computation:
5     public void run() {
6       int cnt = counter;
7       counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 cnt_t : $\bot$	pc_u : 6 cnt_u : $\bot$	counter : 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\bot$	counter : 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\bot$	counter : 1
4	done	pc_u : 6 cnt_u : $\bot$	counter : 1
5	done	pc_u : 7 cnt_u : 1	counter : 1
6	done	pc_u : 8 cnt_u : 1	counter : 2
7	done	done	counter : 2

One possible execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2     int counter = 0;      // shared object state
3
4     // thread's computation:
5     public void run() {
6       int cnt = counter;
7       counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter: 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 1
4	done	pc_u : 6 cnt_u : $\perp$	counter: 1
5	done	pc_u : 7 cnt_u : 1	counter: 1
6	done	pc_u : 8 cnt_u : 1	counter: 2
7	done	done	counter: 2

One possible execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2    int counter = 0;     // shared object state
3
4    // thread's computation:
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 $\mathtt{cnt}_t$: $\perp$	pc_u : 6 $\mathtt{cnt}_u$: $\perp$	`counter`: 0
2	pc_t : 7 $\mathtt{cnt}_t$: 0	pc_u : 6 $\mathtt{cnt}_u$: $\perp$	`counter`: 0
3	pc_t : 8 $\mathtt{cnt}_t$: 0	pc_u : 6 $\mathtt{cnt}_u$: $\perp$	`counter`: 1
4	done	pc_u : 6 $\mathtt{cnt}_u$: $\perp$	`counter`: 1
5	done	pc_u : 7 $\mathtt{cnt}_u$: 1	`counter`: 1
6	done	pc_u : 8 $\mathtt{cnt}_u$: 1	`counter`: 2
7	done	done	`counter`: 2

One possible execution of the concurrent counter

```
1  public class CCounter implements Runnable {
2     int counter = 0;     // shared object state
3
4     // thread's computation:
5     public void run() {
6       int cnt = counter;
7       counter = cnt + 1;
8  } }
```

#	t's LOCAL	u's LOCAL	SHARED
1	pc_t : 6 cnt_t : $\bot$	pc_u : 6 cnt_u : $\bot$	counter : 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\bot$	counter : 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\bot$	counter : 1
4	done	pc_u : 6 cnt_u : $\bot$	counter : 1
5	done	pc_u : 7 cnt_u : 1	counter : 1
6	done	pc_u : 8 cnt_u : 1	counter : 2
7	done	done	counter : 2

One possible execution of the concurrent counter

```
1  public class CCounter implements Runnable {
2      int counter = 0;      // shared object state
3
4      // thread's computation:
5      public void run() {
6        int cnt = counter;
7        counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter: 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 0
3	pc_t : 8 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 1
4	done	pc_u : 6 cnt_u : $\perp$	counter: 1
5	done	pc_u : 7 cnt_u : 1	counter: 1
6	done	pc_u : 8 cnt_u : 1	counter: 2
7	done	done	counter: 2

One alternative execution of the concurrent counter

```
1  public class CCounter implements Runnable {
2     int counter = 0;     // shared object state
3
4     // thread's computation:
5     public void run() {
6       int cnt = counter;
7       counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	$pc_t : 6$ $cnt_t : \perp$	$pc_u : 6$ $cnt_u : \perp$	counter : 0
2	$pc_t : 7$ $cnt_t : 0$	$pc_u : 6$ $cnt_u : \perp$	counter : 0
3	$pc_t : 7$ $cnt_t : 0$	$pc_u : 7$ $cnt_u : 0$	counter : 0
4	$pc_t : 7$ $cnt_t : 0$	$pc_u : 8$ $cnt_u : 0$	counter : 1
5	$pc_t : 8$ $cnt_t : 0$	done	counter : 1
6	done	done	counter : 1

One alternative execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2     int counter = 0;     // shared object state
3
4     // thread's computation:
5     public void run() {
6       int cnt = counter;
7       counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	$\mathrm{pc}_{\mathtt{t}}$: 6 $\mathtt{cnt}_{\mathtt{t}}$: $\perp$	$\mathrm{pc}_{\mathtt{u}}$: 6 $\mathtt{cnt}_{\mathtt{u}}$: $\perp$	`counter`: 0
2	$\mathrm{pc}_{\mathtt{t}}$: 7 $\mathtt{cnt}_{\mathtt{t}}$: 0	$\mathrm{pc}_{\mathtt{u}}$: 6 $\mathtt{cnt}_{\mathtt{u}}$: $\perp$	`counter`: 0
3	$\mathrm{pc}_{\mathtt{t}}$: 7 $\mathtt{cnt}_{\mathtt{t}}$: 0	$\mathrm{pc}_{\mathtt{u}}$: 7 $\mathtt{cnt}_{\mathtt{u}}$: 0	`counter`: 0
4	$\mathrm{pc}_{\mathtt{t}}$: 7 $\mathtt{cnt}_{\mathtt{t}}$: 0	$\mathrm{pc}_{\mathtt{u}}$: 8 $\mathtt{cnt}_{\mathtt{u}}$: 0	`counter`: 1
5	$\mathrm{pc}_{\mathtt{t}}$: 8 $\mathtt{cnt}_{\mathtt{t}}$: 0	done	`counter`: 1
6	done	done	`counter`: 1

One alternative execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2    int counter = 0;     // shared object state
3
4    // thread's computation:
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	$pc_t : 6$ $\mathtt{cnt}_t : \perp$	$pc_u : 6$ $\mathtt{cnt}_u : \perp$	counter : 0
2	$pc_t : 7$ $\mathtt{cnt}_t : 0$	$pc_u : 6$ $\mathtt{cnt}_u : \perp$	counter : 0
3	$pc_t : 7$ $\mathtt{cnt}_t : 0$	$pc_u : 7$ $\mathtt{cnt}_u : 0$	counter : 0
4	$pc_t : 7$ $\mathtt{cnt}_t : 0$	$pc_u : 8$ $\mathtt{cnt}_u : 0$	counter : 1
5	$pc_t : 8$ $\mathtt{cnt}_t : 0$	done	counter : 1
6	done	done	counter : 1

One alternative execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2    int counter = 0;     // shared object state
3
4    // thread's computation:
5    public void run() {
6      int cnt = counter;
7      counter = cnt + 1;
8  } }
```

#	t's LOCAL	u's LOCAL	SHARED
1	pc_t : 6 $\texttt{cnt}_t$: $\perp$	pc_u : 6 $\texttt{cnt}_u$: $\perp$	`counter`: 0
2	pc_t : 7 $\texttt{cnt}_t$: 0	pc_u : 6 $\texttt{cnt}_u$: $\perp$	`counter`: 0
3	pc_t : 7 $\texttt{cnt}_t$: 0	pc_u : 7 $\texttt{cnt}_u$: 0	`counter`: 0
4	pc_t : 7 $\texttt{cnt}_t$: 0	pc_u : 8 $\texttt{cnt}_u$: 0	`counter`: 1
5	pc_t : 8 $\texttt{cnt}_t$: 0	done	`counter`: 1
6	done	done	`counter`: 1

One alternative execution of the concurrent counter

```java
public class CCounter implements Runnable {
   int counter = 0;     // shared object state

   // thread's computation:
   public void run() {
     int cnt = counter;
     counter = cnt + 1;
} }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter: 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter: 0
3	pc_t : 7 cnt_t : 0	pc_u : 7 cnt_u : 0	counter: 0
4	pc_t : 7 cnt_t : 0	pc_u : 8 cnt_u : 0	counter: 1
5	pc_t : 8 cnt_t : 0	done	counter: 1
6	done	done	counter: 1

One alternative execution of the concurrent counter

```
1  public class CCounter implements Runnable {
2     int counter = 0;     // shared object state
3
4     // thread's computation:
5     public void run() {
6       int cnt = counter;
7       counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	$pc_t : 6$ $cnt_t : \perp$	$pc_u : 6$ $cnt_u : \perp$	counter: 0
2	$pc_t : 7$ $cnt_t : 0$	$pc_u : 6$ $cnt_u : \perp$	counter: 0
3	$pc_t : 7$ $cnt_t : 0$	$pc_u : 7$ $cnt_u : 0$	counter: 0
4	$pc_t : 7$ $cnt_t : 0$	$pc_u : 8$ $cnt_u : 0$	counter: 1
5	$pc_t : 8$ $cnt_t : 0$	done	counter: 1
6	done	done	counter: 1

One alternative execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2     int counter = 0;     // shared object state
3
4     // thread's computation:
5     public void run() {
6       int cnt = counter;
7       counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 cnt_t : $\perp$	pc_u : 6 cnt_u : $\perp$	counter : 0
2	pc_t : 7 cnt_t : 0	pc_u : 6 cnt_u : $\perp$	counter : 0
3	pc_t : 7 cnt_t : 0	pc_u : 7 cnt_u : 0	counter : 0
4	pc_t : 7 cnt_t : 0	pc_u : 8 cnt_u : 0	counter : 1
5	pc_t : 8 cnt_t : 0	done	counter : 1
6	done	done	counter : 1

One alternative execution of the concurrent counter

```java
1  public class CCounter implements Runnable {
2      int counter = 0;     // shared object state
3
4      // thread's computation:
5      public void run() {
6        int cnt = counter;
7        counter = cnt + 1;
8  } }
```

#	t'S LOCAL	u'S LOCAL	SHARED
1	pc_t : 6 $\texttt{cnt}_t$: $\perp$	pc_u : 6 $\texttt{cnt}_u$: $\perp$	`counter`: 0
2	pc_t : 7 $\texttt{cnt}_t$: 0	pc_u : 6 $\texttt{cnt}_u$: $\perp$	`counter`: 0
3	pc_t : 7 $\texttt{cnt}_t$: 0	pc_u : 7 $\texttt{cnt}_u$: 0	`counter`: 0
4	pc_t : 7 $\texttt{cnt}_t$: 0	pc_u : 8 $\texttt{cnt}_u$: 0	`counter`: 1
5	pc_t : 8 $\texttt{cnt}_t$: 0	done	`counter`: 1
6	done	done	`counter`: 1

Traces

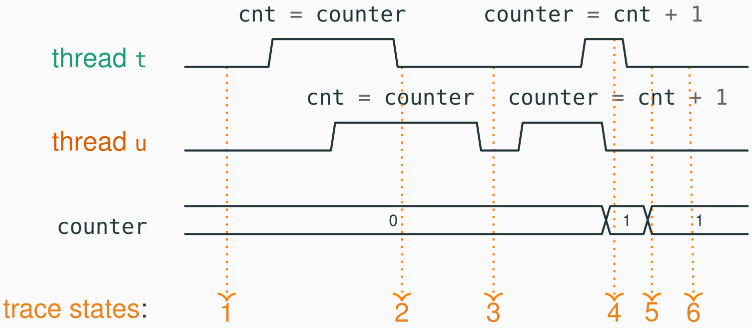
Traces

#	t's LOCAL	u's LOCAL	SHARED
1	$pc_t : 6$ $\mathtt{cnt}_t : \bot$	$pc_u : 6$ $\mathtt{cnt}_u : \bot$	$\mathtt{counter} : 0$
2	$pc_t : 7$ $\mathtt{cnt}_t : 0$	$pc_u : 6$ $\mathtt{cnt}_u : \bot$	$\mathtt{counter} : 0$
3	$pc_t : 7$ $\mathtt{cnt}_t : 0$	$pc_u : 7$ $\mathtt{cnt}_u : 0$	$\mathtt{counter} : 0$
4	$pc_t : 7$ $\mathtt{cnt}_t : 0$	$pc_u : 8$ $\mathtt{cnt}_u : 0$	$\mathtt{counter} : 1$
5	$pc_t : 8$ $\mathtt{cnt}_t : 0$	done	$\mathtt{counter} : 1$
6	done	done	$\mathtt{counter} : 1$

The sequence of states gives an execution trace of the concurrent program. A trace is an abstraction of concrete executions:

- atomic/linearized
- complete
- interleaved

Trace abstractions



atomic/linearized: the effects of each thread appear as if they happened instantaneously, when the trace snapshot is taken, in the thread's sequential order

complete: the trace include all intermediate atomic states

interleaved: the trace is an interleaving of each thread's linear trace (in particular, no simultaneity)

Abstraction of concurrent programs

When convenient, we will use an abstract notation for multi-threaded applications, which is similar to the pseudo-code used in Ben-Ari's book but uses Java syntax.

shared memory

```
int counter = 0;
```

thread t	thread u
`int cnt;`	`int cnt;` local memory
`cnt = counter;`	`cnt = counter;`
`counter = cnt + 1;`	`counter = cnt + 1;`

code

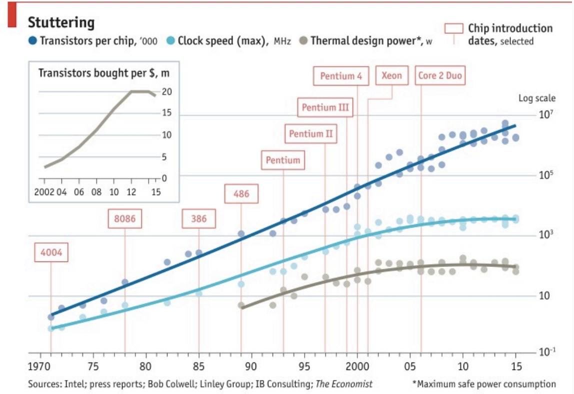
Each line of code includes exactly one instruction that can be executed atomically:

- atomic statement $\simeq$ single read or write to global variable
- precise definition is tricky in Java, but we will learn to avoid pitfalls

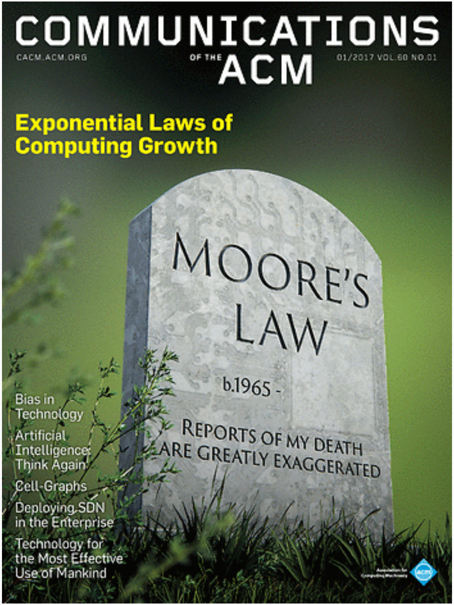
Moore’s law and its end

The spectacular advance of computing in the last 60+ years has been driven by Moore’s law:

The density of transistors in integrated circuits doubles approximately every 2 years



Moore’s Law in January 2017



Concurrency everywhere

The end of Moore's law is having a major impact on the practice of programming:

- before: CPUs get faster without significant architectural changes
  - program as usual, and wait for your program to run faster
  - concurrent programming is a niche skill (for operating systems, databases, high-performance computing)
- now: CPUs do not get faster but add more and more parallel cores
  - program with concurrency in mind, otherwise your programs remain slow
  - concurrent programming is pervasive

Very different systems all require concurrent programming:

- desktop PCs
- smart phones
- video-games consoles
- embedded systems
- the Raspberry Pi
- cloud computing

Amdahl's law: concurrency is no free lunch

We have n processors that can run in parallel. How much speedup can we achieve?

$$\text{speedup} = \frac{\text{sequential execution time}}{\text{parallel execution time}}$$

Amdahl's law shows that the impact of introducing parallelism is limited by the fraction p of a program that can be parallelized:

$$\text{maximum speedup} = \frac{1}{\underbrace{(1-p)}_{\text{sequential part}} + \underbrace{p/n}_{\text{parallel part}}}$$

Amdahl's law: examples

$$\text{maximum speedup} = \frac{1}{\underbrace{(1-p)}_{\text{sequential part}} + \underbrace{p/n}_{\text{parallel part}}}$$

With $n = 10$ processors, how close can we get to a 10x speedup?

% SEQUENTIAL	% PARALLEL	MAX SPEEDUP
20%	80%	3.57
10%	90%	5.26
1%	99%	9.17

With $n = 100$ processors, how close can we get to a 100x speedup?

% SEQUENTIAL	% PARALLEL	MAX SPEEDUP
20%	80%	4.81
10%	90%	9.17
1%	99%	50.25

Creating a parallel program

- **Thought process:**
  1. **Identify work that can be performed in parallel**
  2. **Partition work (and also data associated with the work)**
  3. **Manage data access, communication, and synchronization**

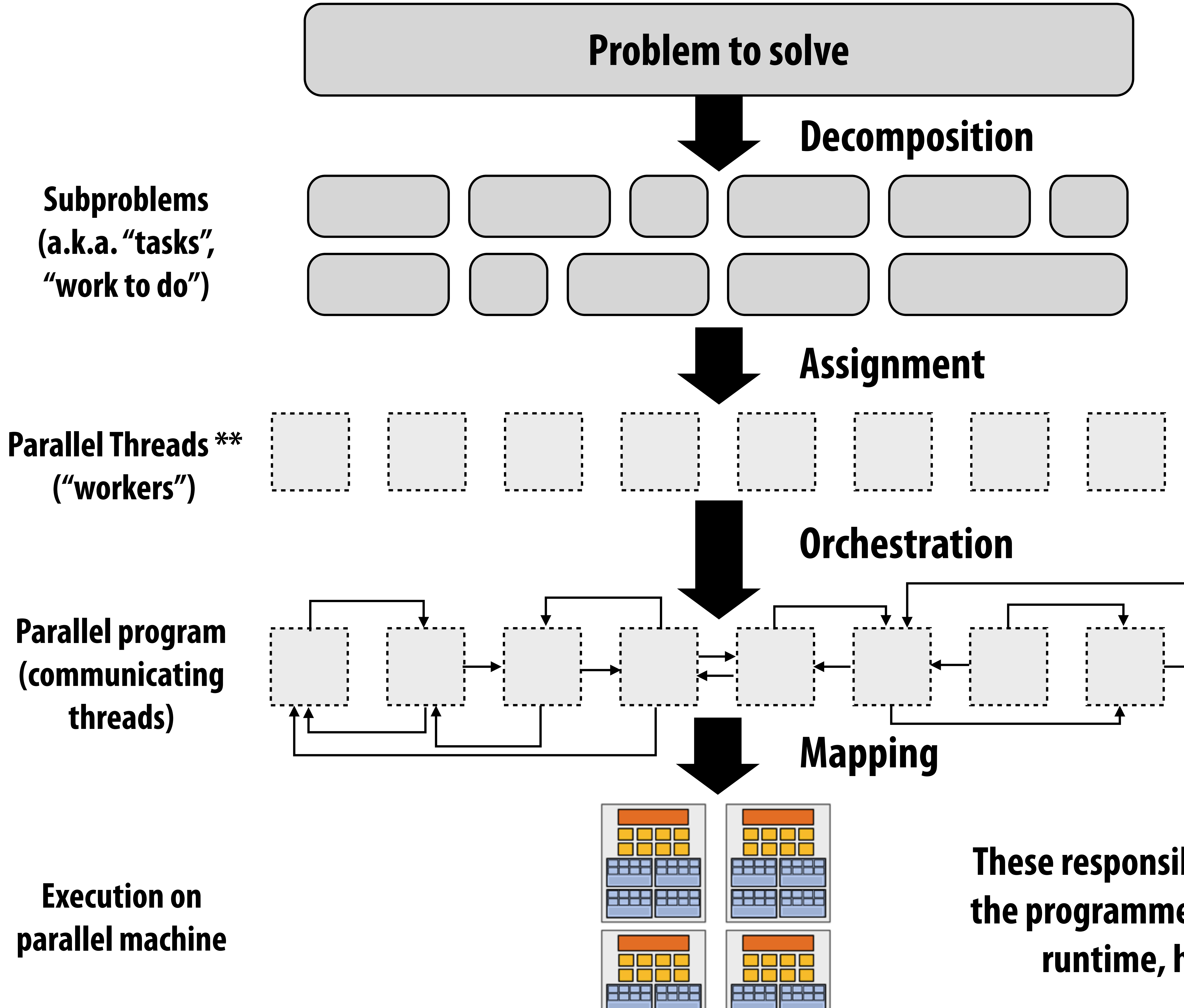
- **Recall one of our main goals is speedup ***

  For a fixed computation:

$$\text{Speedup(P processors)} = \frac{\text{Time (1 processor)}}{\text{Time (P processors)}}$$

* Other goals include high efficiency (cost, area, power, etc.) or working on bigger problems than can fit on one machine

Creating a parallel program



** I had to pick a term

These responsibilities may be assumed by the programmer, by the system (compiler, runtime, hardware), or by both!

Adopted from: Culler, Singh, and Gupta

Decomposition

- Break up problem into tasks that can be carried out in parallel
  - Decomposition need not happen statically
  - New tasks can be identified as program executes

- Main idea: create at least enough tasks to keep all execution units on a machine busy

Key aspect of decomposition: identifying dependencies (or... a lack of dependencies)

Amdahl's Law: dependencies limit maximum speedup due to parallelism

- **You run your favorite sequential program...**

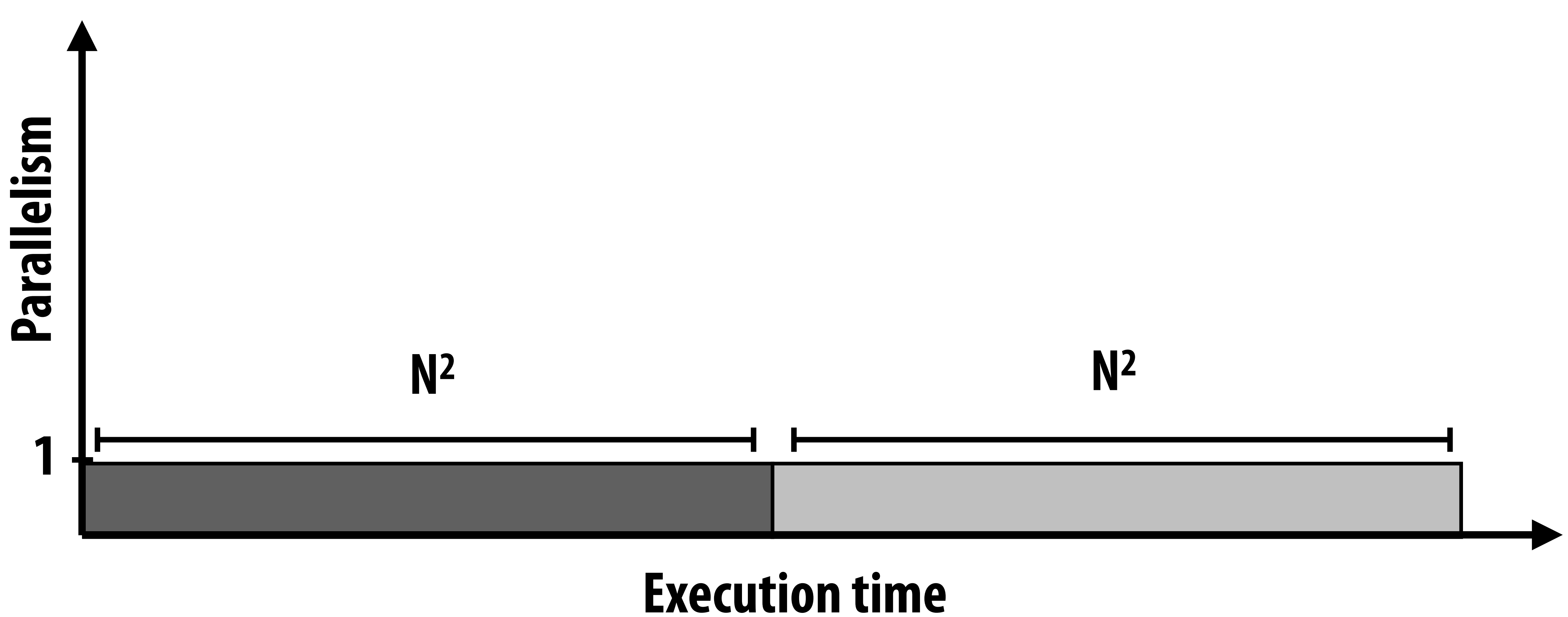
- **Let S = the fraction of sequential execution that is inherently sequential (dependencies prevent parallel execution)**

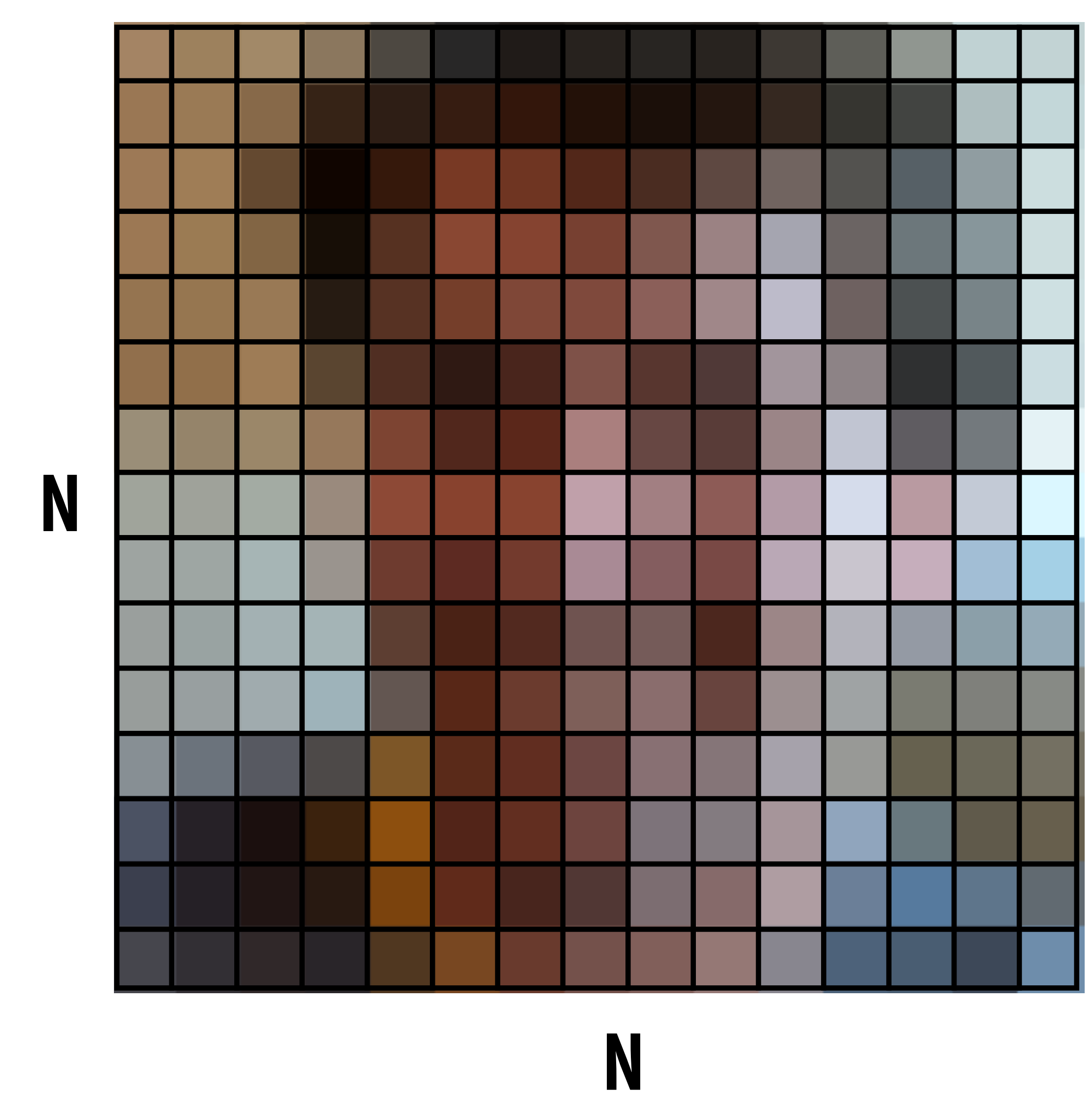
- **Then maximum speedup due to parallel execution $\leq {}^{1}/_{S}$**

A simple example

- Consider a two-step computation on a N x N image
  - Step 1: double brightness of all pixels
    (independent computation on each grid element)
  - Step 2: compute average of all pixel values

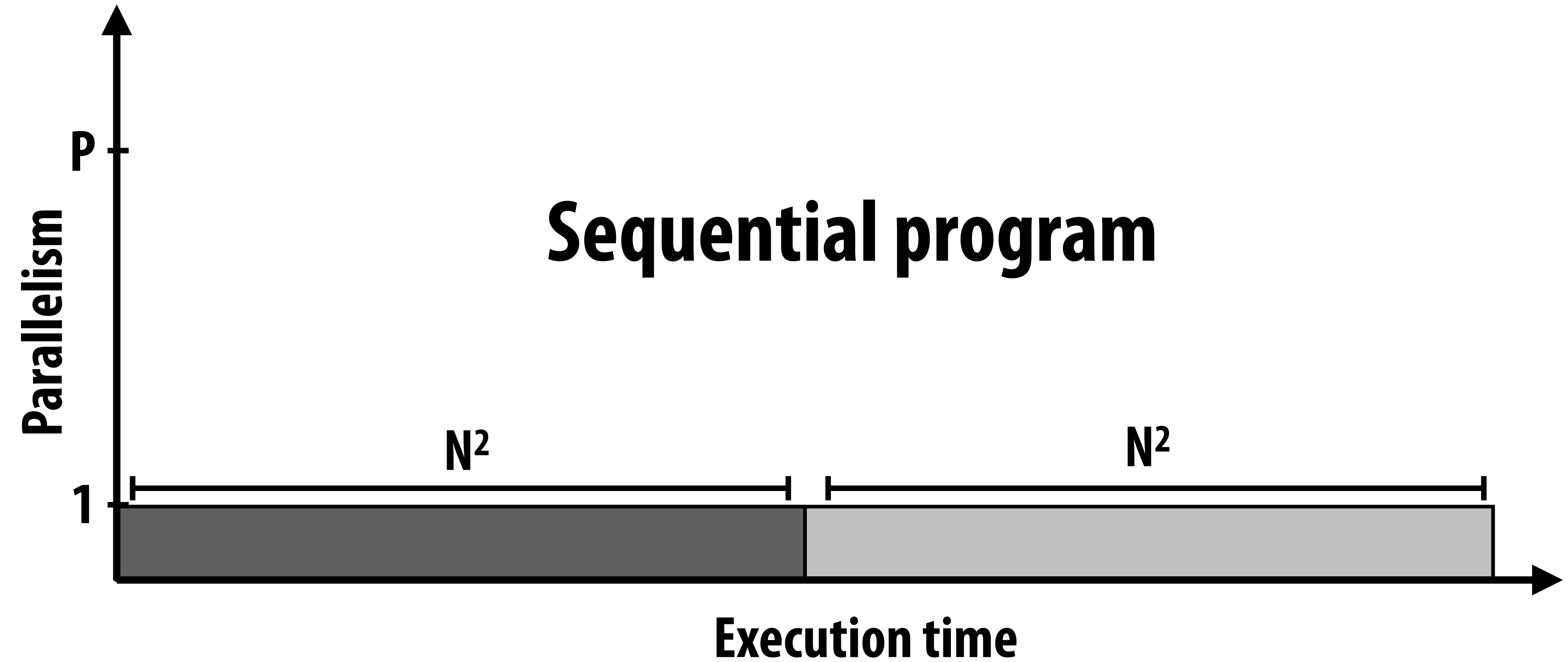
- Sequential implementation of program
  - Both steps take ~ N^2 time, so total time is ~ $2N^2$





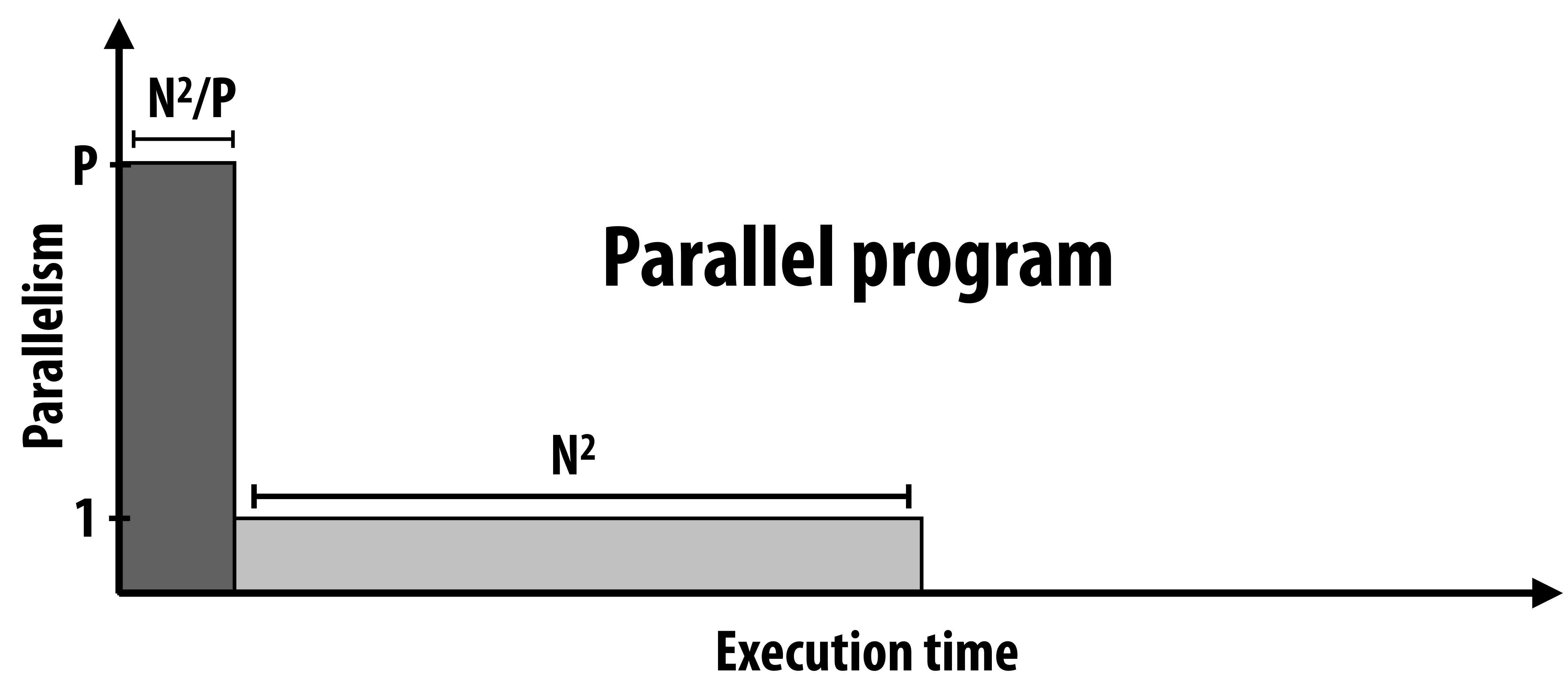
First attempt at parallelism (P processors)

- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: execute serially
    - time for phase 2: N^2



- **Overall performance:**

$$\text{Speedup} \le \frac{2n^2}{\frac{n^2}{p} + n^2}$$



$$\text{Speedup} \le 2$$

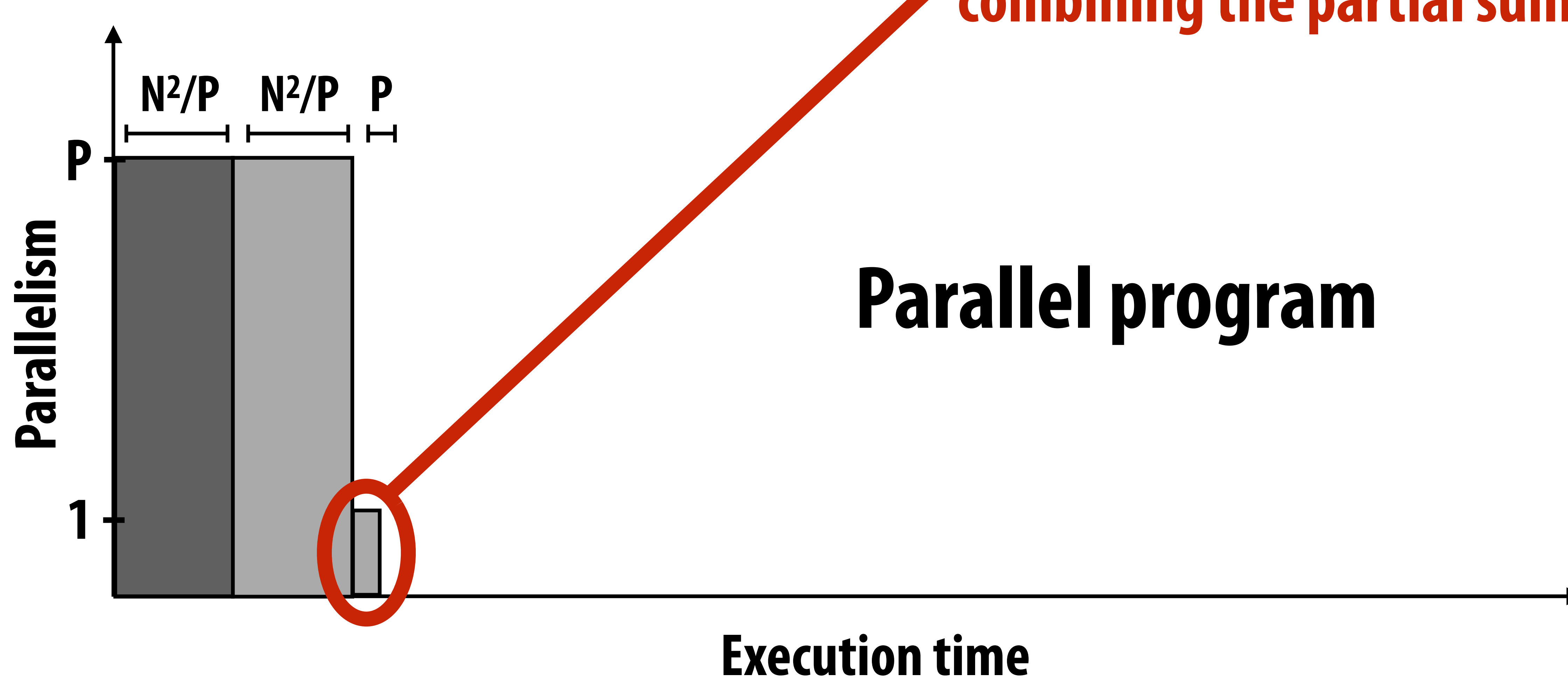
Parallelizing step 2

- **Strategy:**
  - Step 1: execute in parallel
    - time for phase 1: N^2/P
  - Step 2: compute partial sums in parallel, combine results serially
    - time for phase 2: $N^2/P + P$

- **Overall performance:**
  - Speedup $\leq \dfrac{2n^2}{\dfrac{2n^2}{p} + p}$

Note: speedup → P when N >> P

overhead:
combining the partial sums



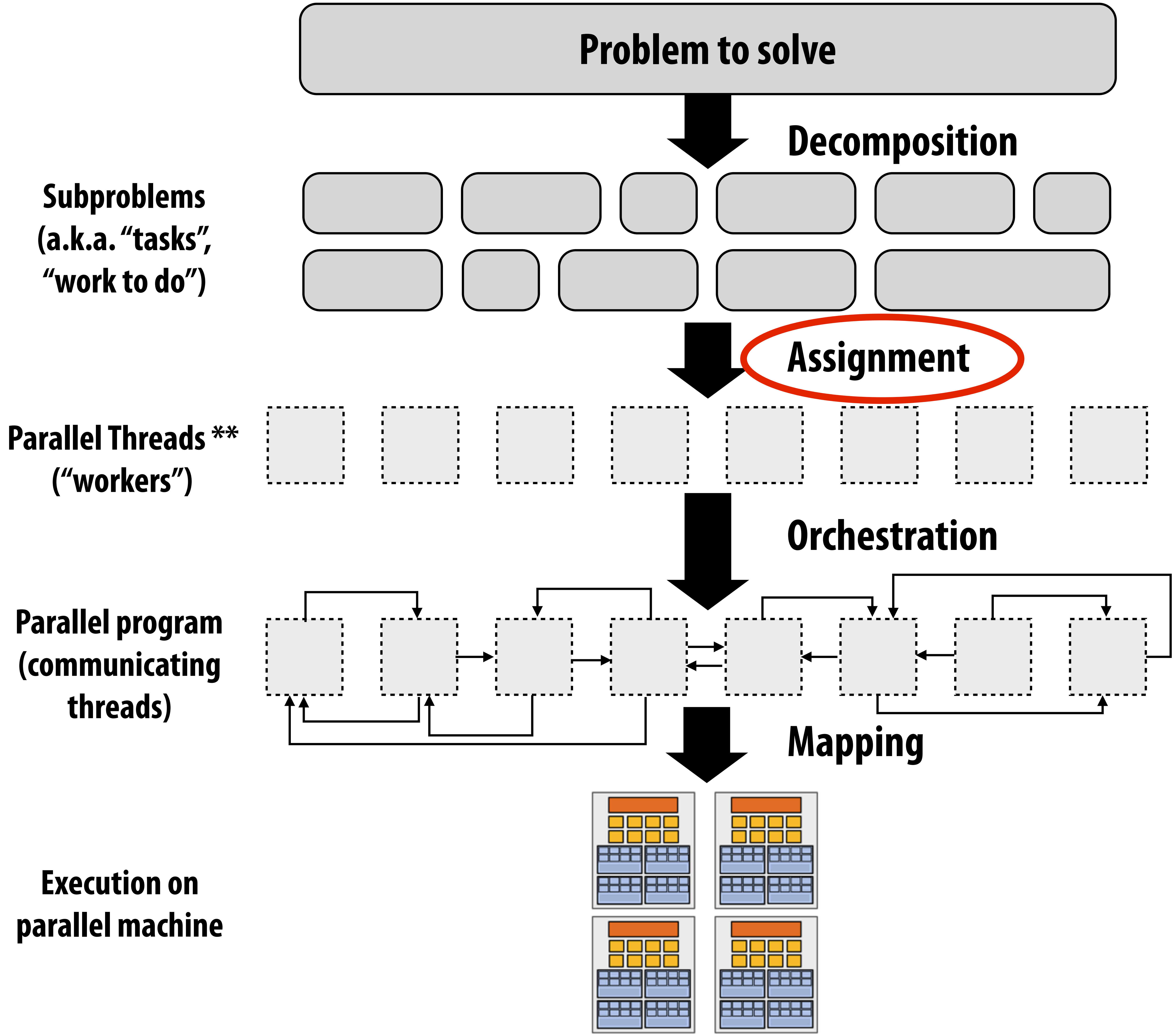
Parallel program

Decomposition

- **Who is responsible for performing decomposition?**
  - In most cases: the programmer

- **Automatic decomposition of sequential programs continues to be a challenging research problem (very difficult in general case)**
  - Compiler must analyze program, identify dependencies
    - What if dependencies are data dependent (not known at compile time)?
  - Researchers have had modest success with simple loop nests
  - The "magic parallelizing compiler" for complex, general-purpose code has not yet been achieved

Assignment



** I had to pick a term

Assignment

** I had to pick a term
(will explain in a second)

- Assigning tasks to threads **
  - Think of "tasks" as things to do
  - Think of threads as "workers"
- Goals: balance workload, reduce communication costs
- Can be performed statically, or dynamically during execution
- While programmer often responsible for decomposition, many languages/runtimes take responsibility for assignment.

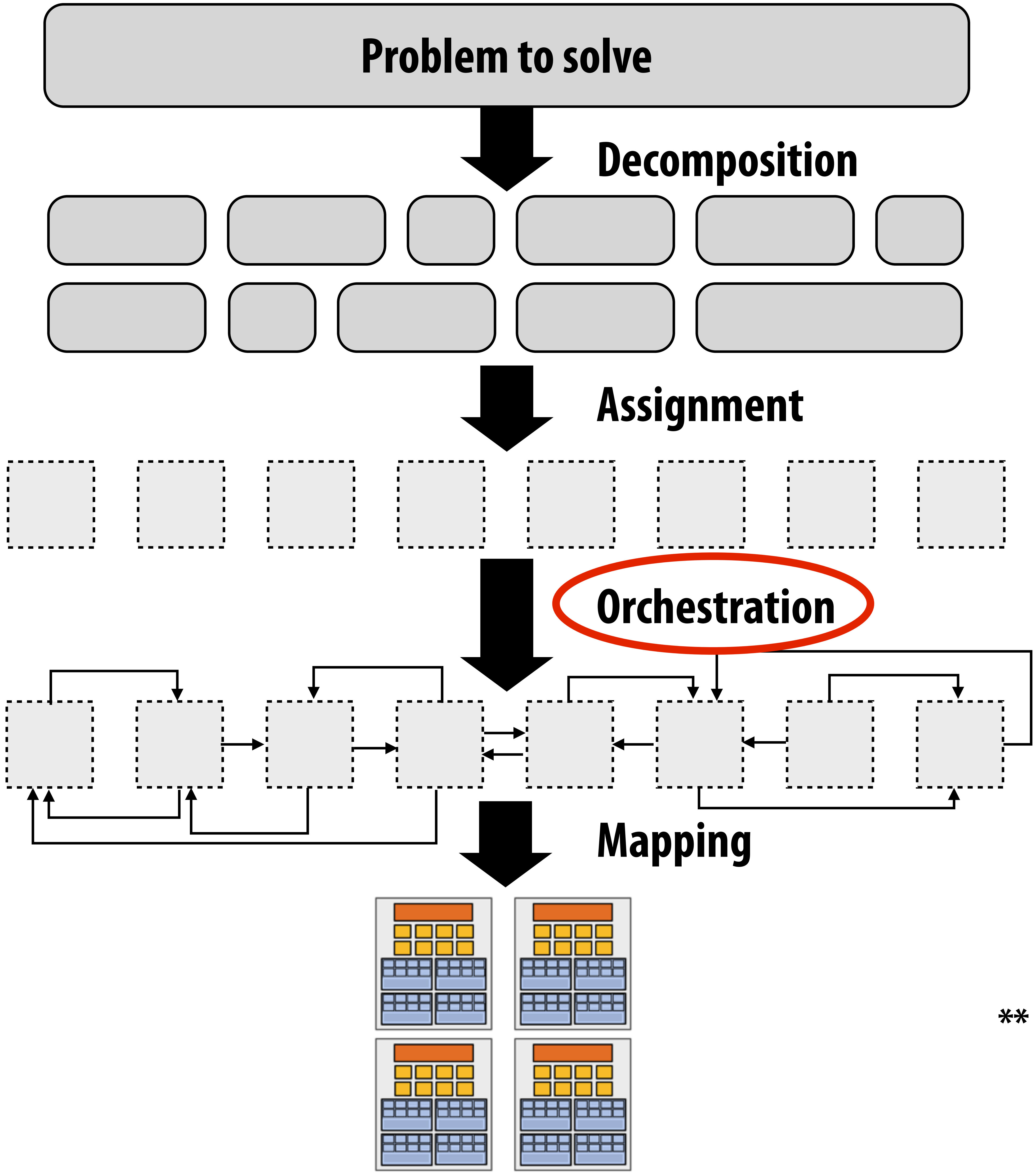
Orchestration

Subproblems
(a.k.a. "tasks",
"work to do")

Parallel Threads **
("workers")

Parallel program
(communicating
threads)

Execution on
parallel machine



** I had to pick a term

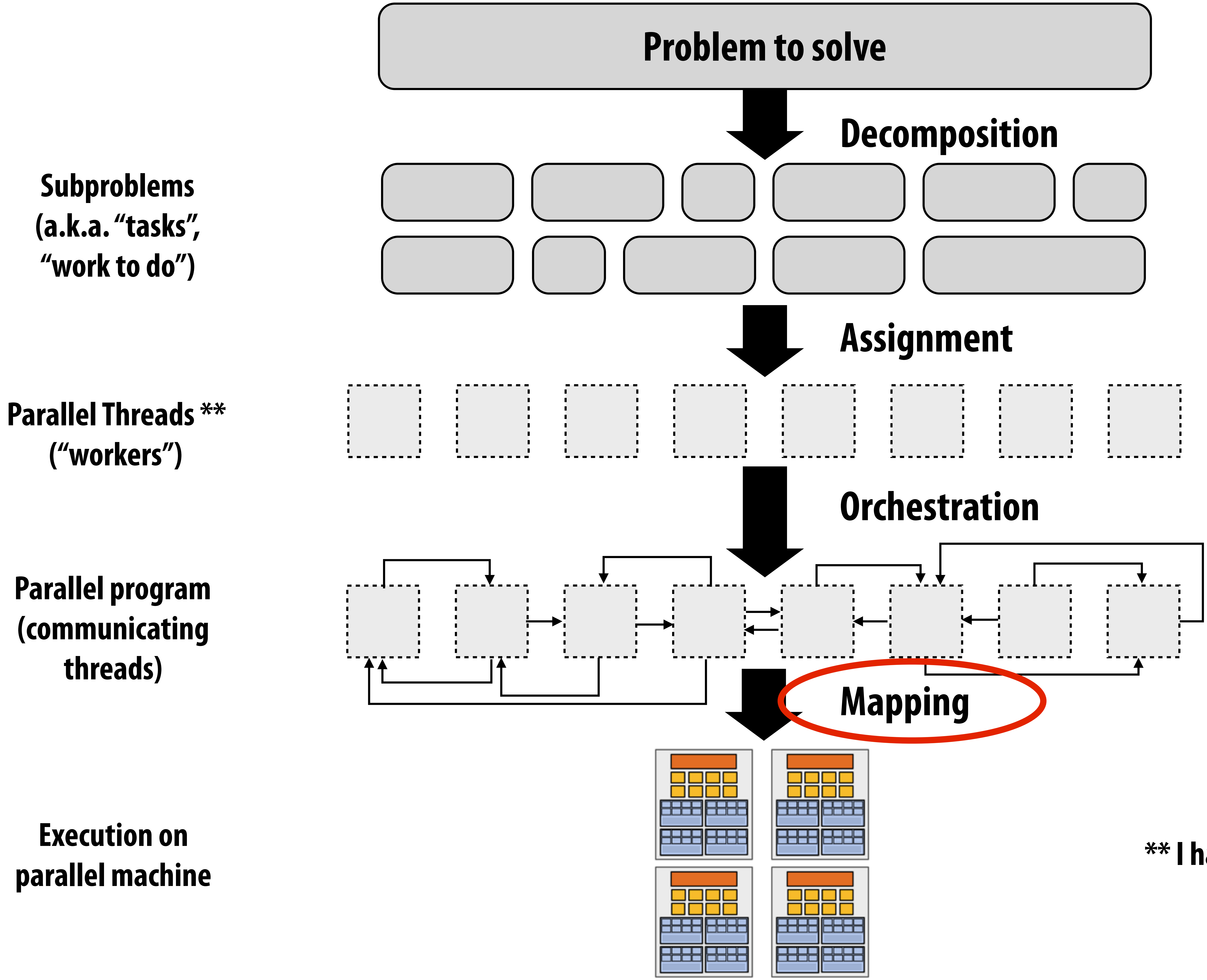
Orchestration

- **Involves:**
  - Structuring communication
  - Adding synchronization to preserve dependencies if necessary
  - Organizing data structures in memory
  - Scheduling tasks

- **Goals: reduce costs of communication/sync, preserve locality of data reference, reduce overhead, etc.**

- **Machine details impact many of these decisions**
  - If synchronization is expensive, might use it more sparsely

Mapping to hardware



** I had to pick a term

Mapping to hardware

- **Mapping "threads" ("workers") to hardware execution units**

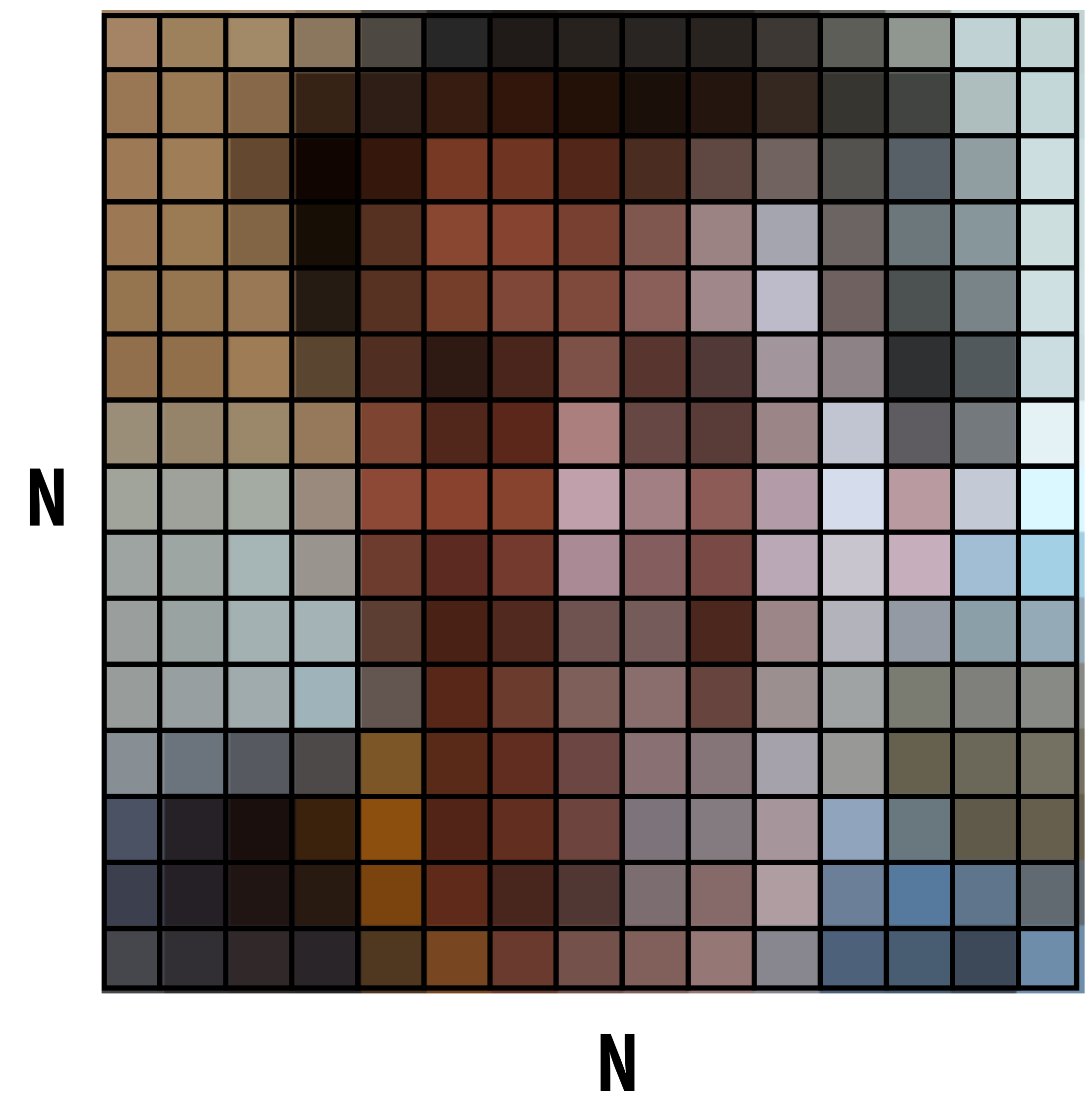
- **Example 1: mapping by the operating system**
  - e.g., map pthread to HW execution context on a CPU core

- **Example 2: mapping by the compiler**
  - Map ISPC program instances to vector instruction lanes

- **Example 3: mapping by the hardware**
  - Map CUDA thread blocks to GPU cores (future lecture)

- **Some interesting mapping decisions:**
  - Place <u>related</u> threads (cooperating threads) on the same processor (maximize locality, data sharing, minimize costs of comm/sync)
  - Place <u>unrelated</u> threads on the same processor (one might be bandwidth limited and another might be compute limited) to use machine more efficiently

Decomposing computation or data?



Often, the reason a problem requires lots of computation (and needs to be parallelized) is that it involves manipulating a lot of data.

I've described the process of parallelizing programs as an act of partitioning <u>computation</u> (work).

Often, it's equally valid to think of <u>partitioning data</u>. (computations go with the data)

But there are many computations where the correspondence between work-to-do ("tasks") and data is less clear. In these cases it's natural to think of partitioning computation.