SUPSI

Introduction to concurrency and parallelism (Part 1)

Massimo Coluzzi
Content realised in collaboration with:
Tiziano Leidi, Fabio Landoni

Parallel and Concurrent Programming
Bachelor in Data Science

In previous episodes...

- We made a summary of the Von Neumann architecture.
- We have seen how memory hardware works and why.
- A CPU copies data from RAM to L3 cache and every core copies data from L3 cache to L2 and L1 cache.

In this lecture

- Sequential Vs Concurrent Executions
- Concurrent Vs Parallel Executions
- The Amdahl's Law

Question

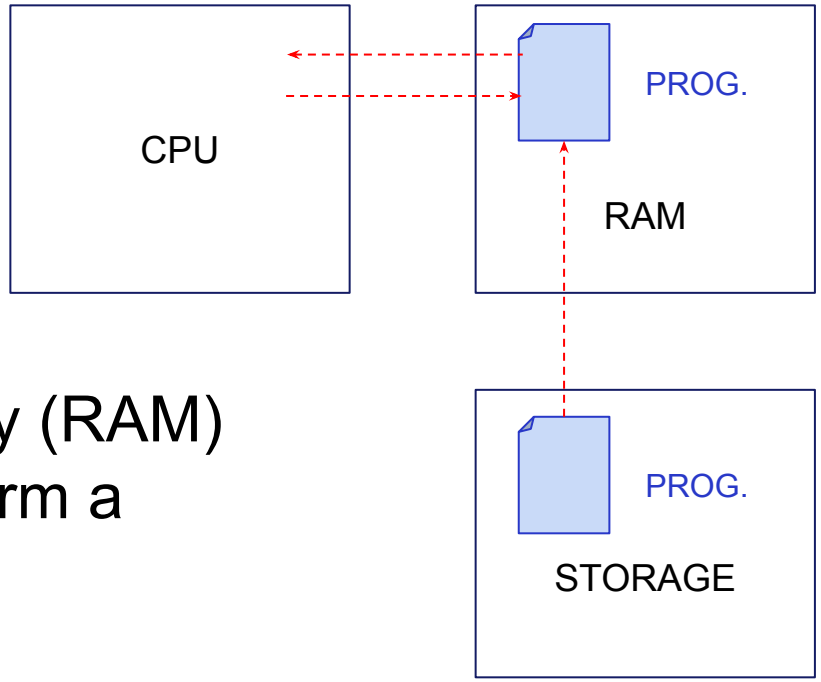
What is a program?

Answer

A program is a set of files stored on disk.

It is made of:

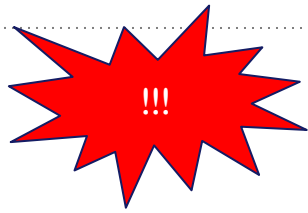
- a set of instructions
- some associated data



A program is loaded into memory (RAM) by the operating system to perform a task.

Question

What is a process?



Answer

A process is the independent instance of a running program.

It represents the execution environment of a program.

It is made of:

- instructions (the code)
- data
- a state
- other resources

- CPU
- Memory
- Address space
- Disk
- Network

Sequential execution

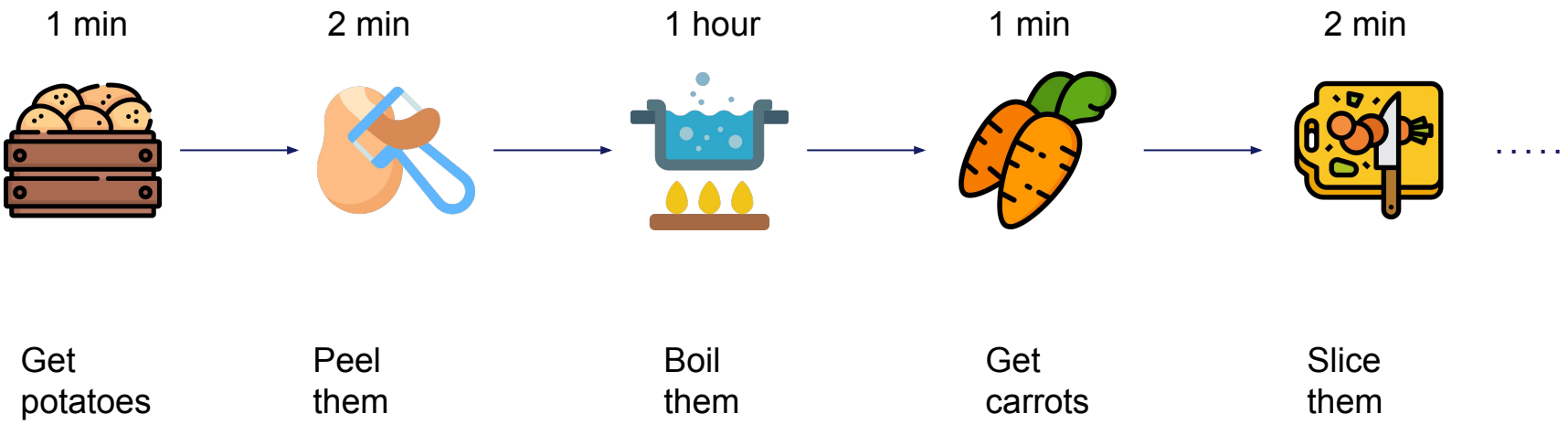
In sequential executions:

The program is broken down into a sequence of instructions that are executed one after another.

The processor executes one instruction at a time.
There is no overlapping.

Sequential execution: example cooking

For example let's have a cook working in the kitchen:

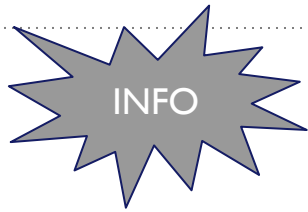


Sequential execution

The time it takes for a sequential program to run is limited by the speed of the processor and how fast it can execute that series of instructions.

New programmers are generally taught to code in this way, because it is easy to understand.

However, this approach has limitations.



Moore's Law

In 1965 Gordon Moore, co-founder of Intel, observed that:

About every two years:

- the number of transistors that can be packed into a given unit of space doubles.
- the processing power of computers doubles.
- the cost halves.

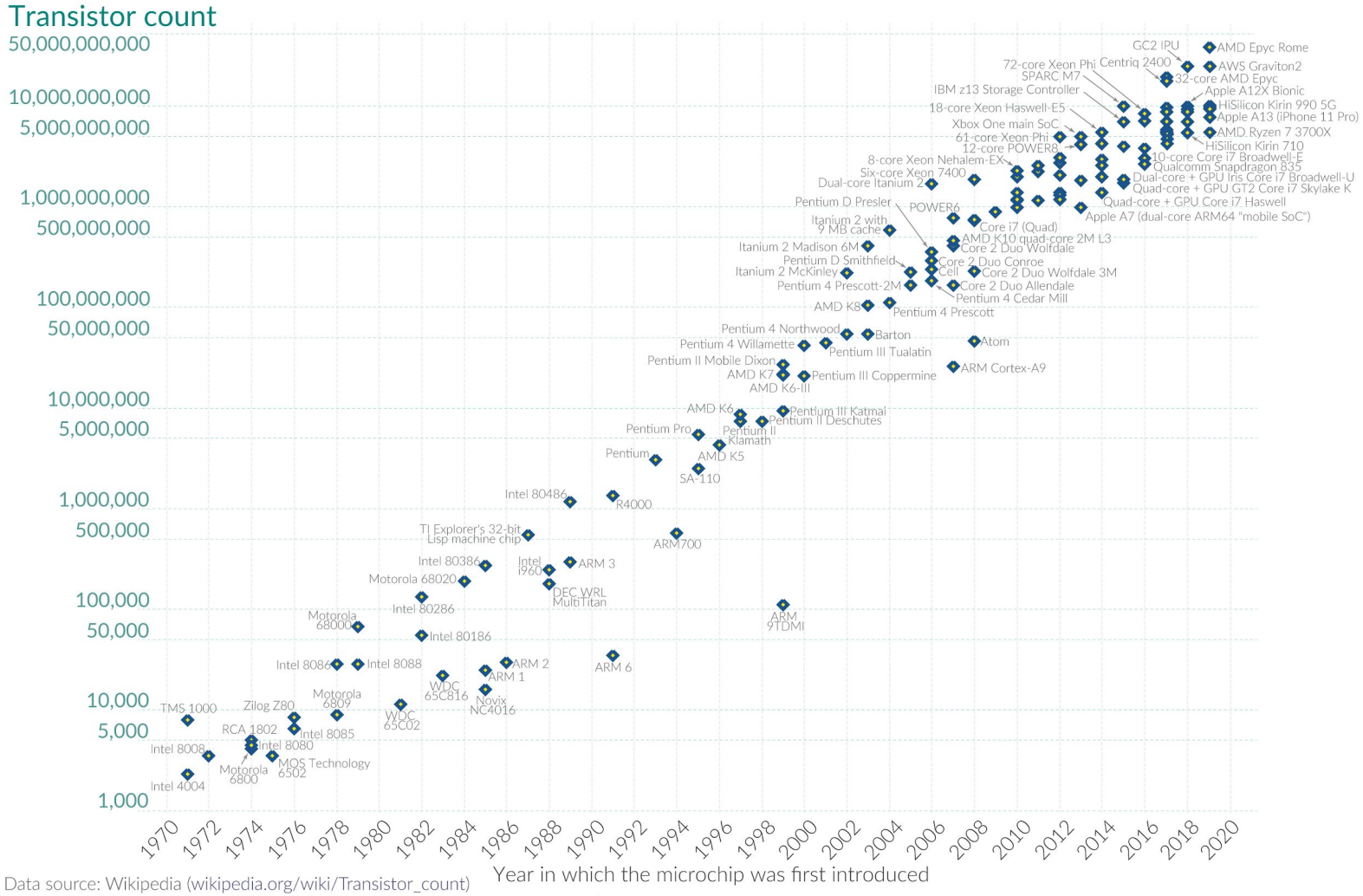
Moore's Law

INFO

Moore's Law: The number of transistors on microchips doubles every two years

Our World in Data

Moore's law describes the empirical regularity that the number of transistors on integrated circuits doubles approximately every two years.
This advancement is important for other aspects of technological progress in computing – such as processing speed or the price of computers.



Data source: Wikipedia (wikipedia.org/wiki/Transistor_count)
OurWorldinData.org – Research and data to make progress against the world's largest problems.

Moore's Law

Exponential growth by Moore's law came to an end more than a decade ago with respect to clock speeds.

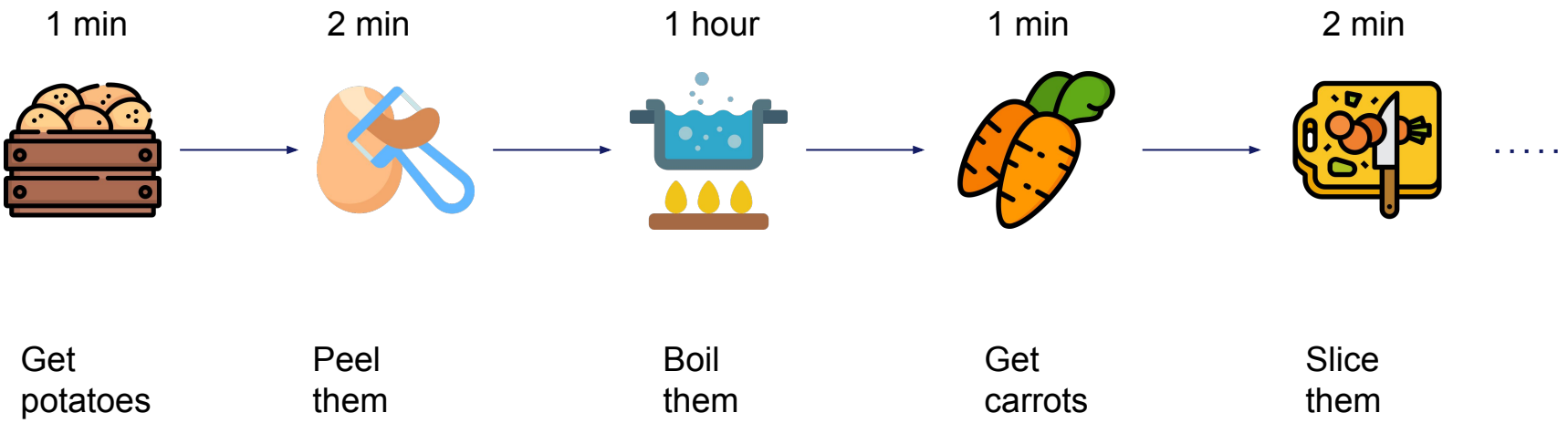
We need alternatives to achieve performance gains.

Technology is moved into multicore processors (more than one CPU on the same machine).

To exploit this processing power, programs can no longer be written with sequential execution in mind.

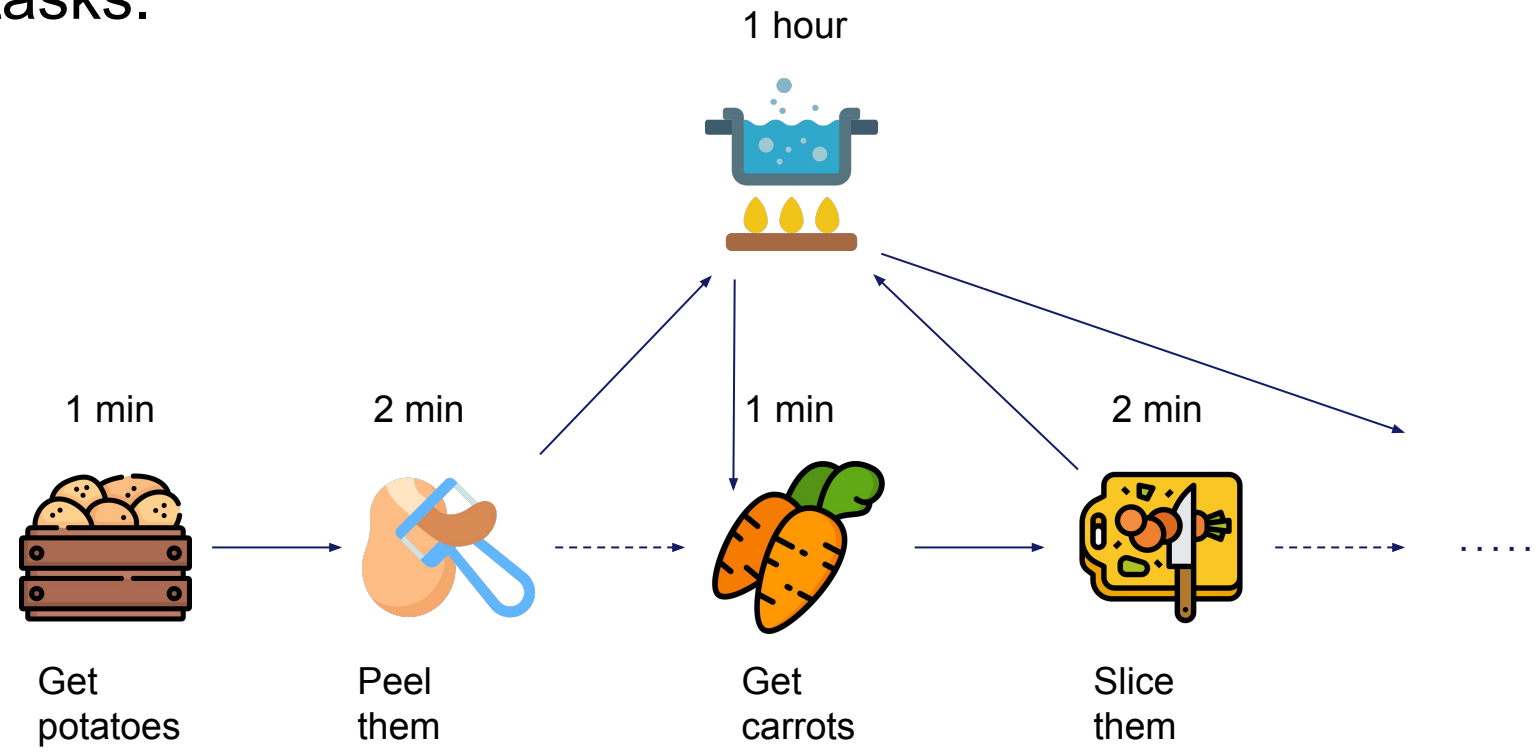
Question

Currently our cook takes 3 hours to complete the meal.
How could we optimize the process?



Answer

No work is required by the cook during boiling. Therefore, while potatoes are boiling the cook can move forward to other tasks.



Asynchronous execution

In the previous example, the cooking hob does the task of boiling and does not require the cook.

In computer science, we can identify the cook with the processor and the cooking hob with the disk.

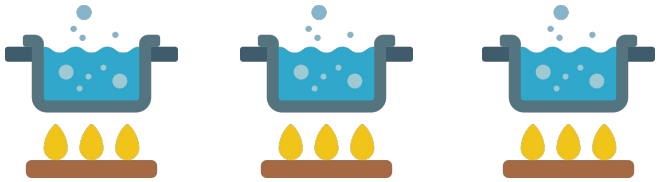
Hence, boiling becomes an I/O operation.

During I/O operations, the processor is not involved and can perform other tasks.

Concurrency

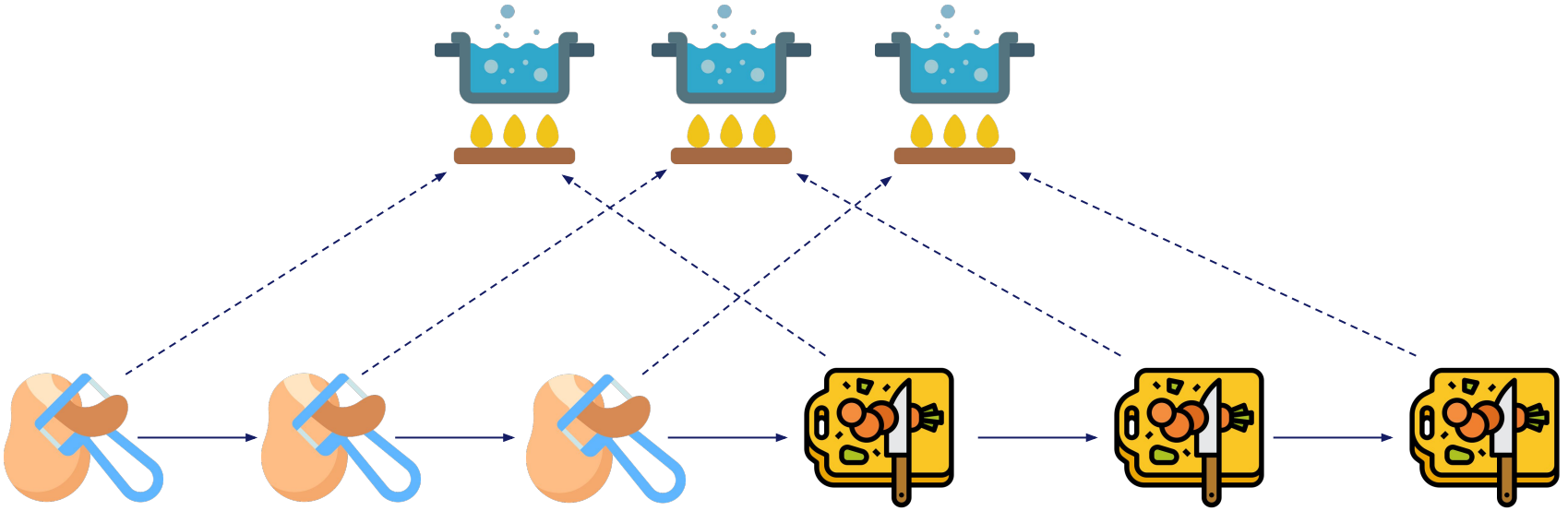
Concurrent execution: example

Our cook needs to boil potatoes and carrots as before.
But this time, we have multiple pots.



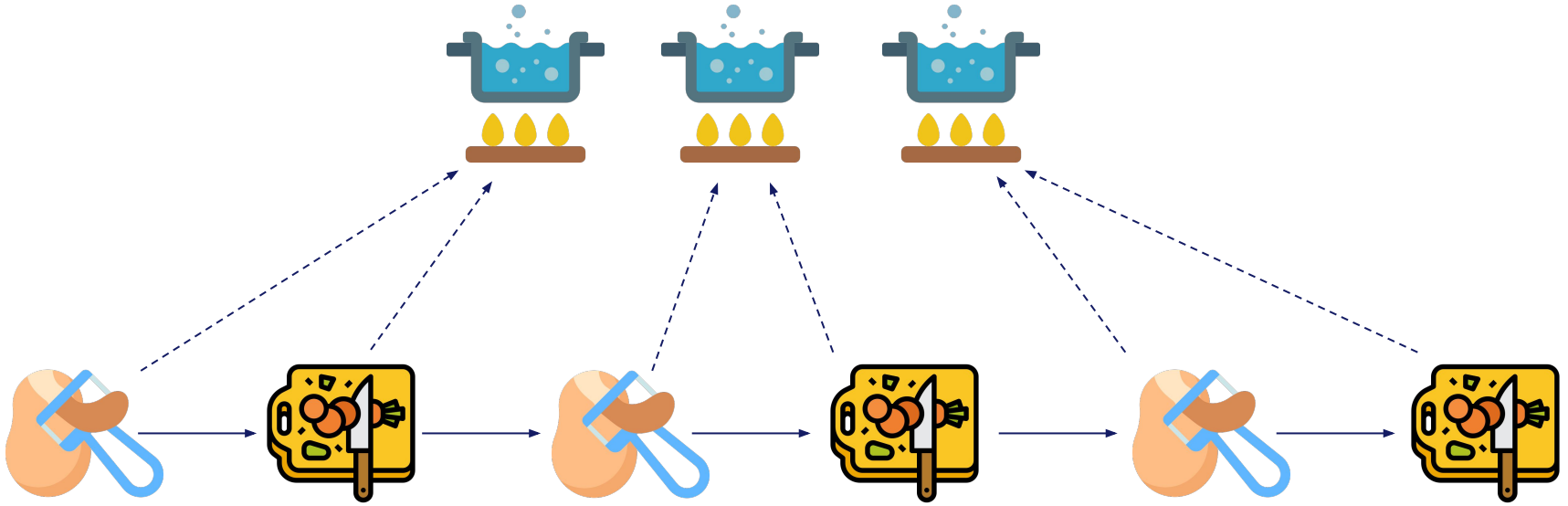
Concurrent execution: example

The cook could peel all the potatoes first and then all the carrots. But this means potatoes will boil longer than carrots.



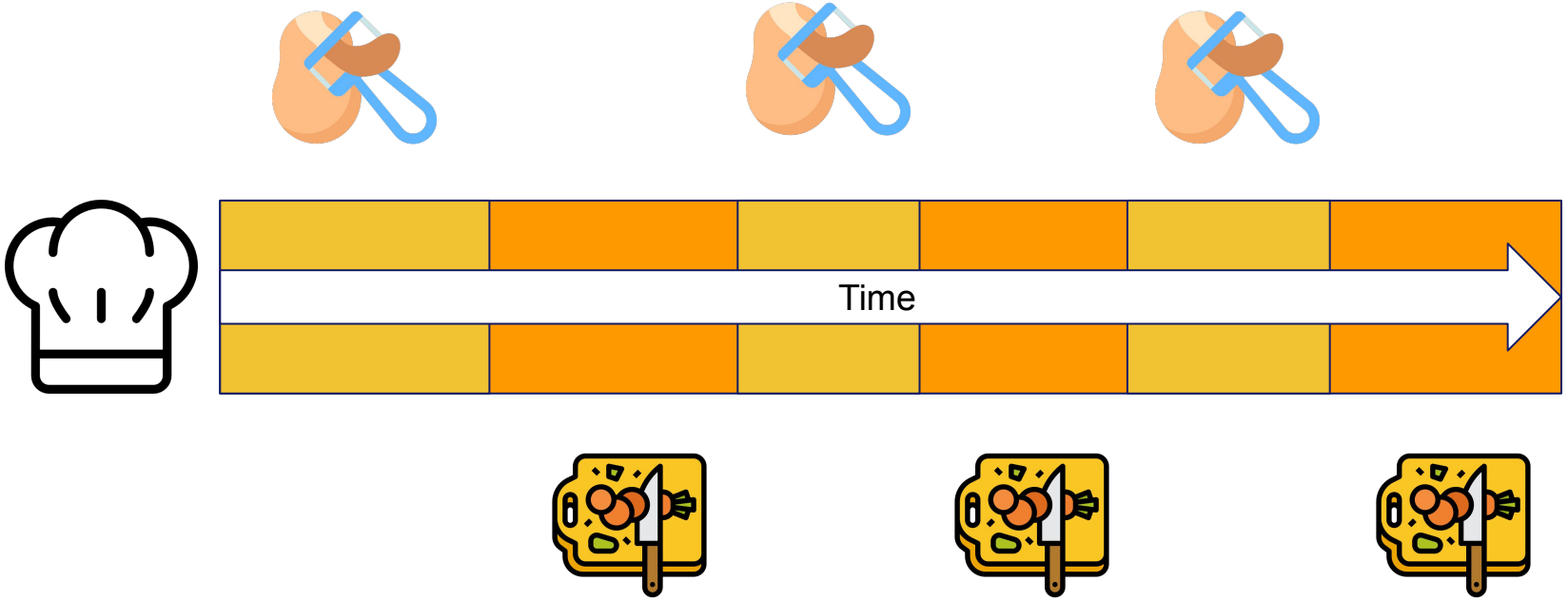
Concurrent execution: example

Alternatively, the cook can alternate his effort between potatoes and carrots.



Concurrent execution: example

In this case, the cook's working time is interleaved between potatoes and carrots.



Concurrency

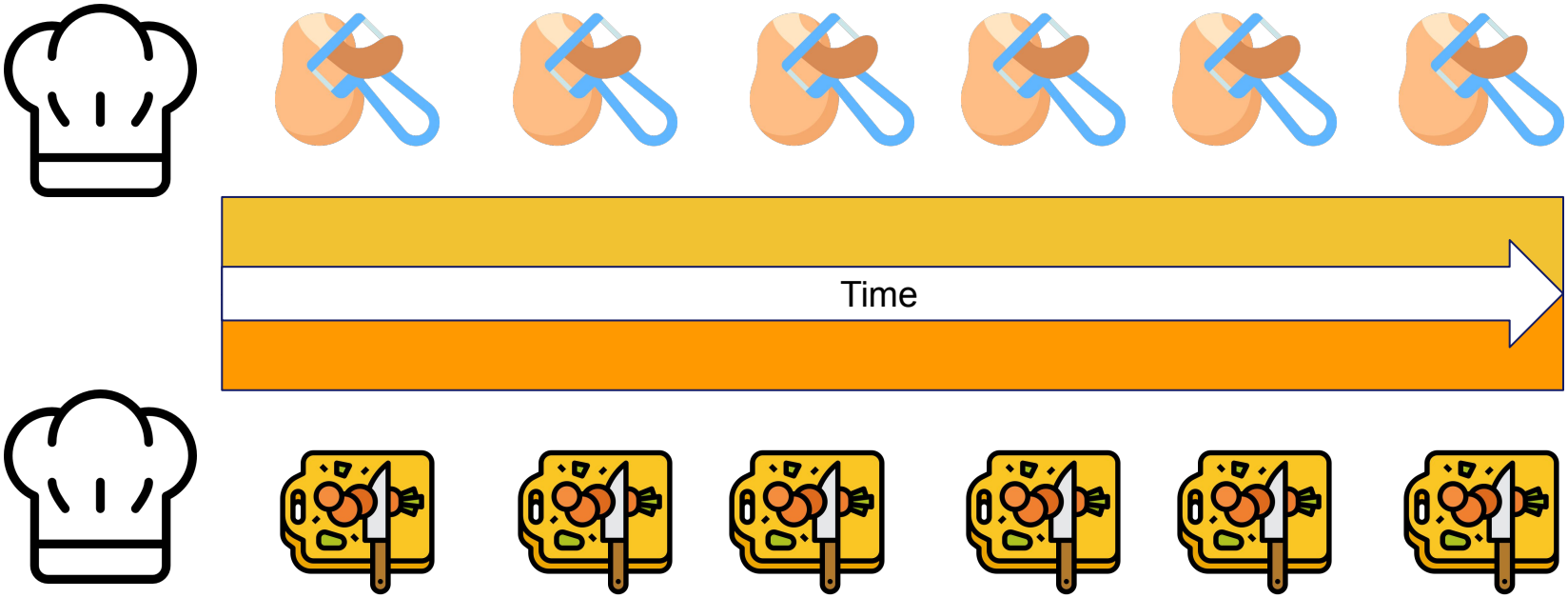
Concurrency is the ability of a program to be broken into parts that can run out of order (or partially out of order) without affecting the end results.

- Two (or more) tasks overlap in time.
- Since there is only a single shared resource (i.e., processor), only one task will actually be executed at any instant.
- If the swap takes turns more frequently, it might create the illusion that they are executing simultaneously.

Parallelism

Parallel execution: example

Let’s pretend to have two cooks. This time, every cook can take care of a different vegetable.



Parallel execution

Parallel execution allows multiple tasks to be executed simultaneously.

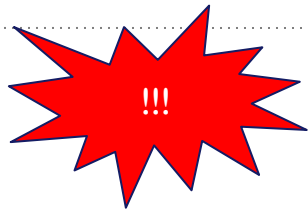
It needs parallel hardware. For example:

- multiple processing cores
- graphics processing units (GPUs)
- computer cluster.

The software environment in which execution is performed also needs to support parallel processing.

Question

Which is the difference between concurrency and parallelism?



Answer

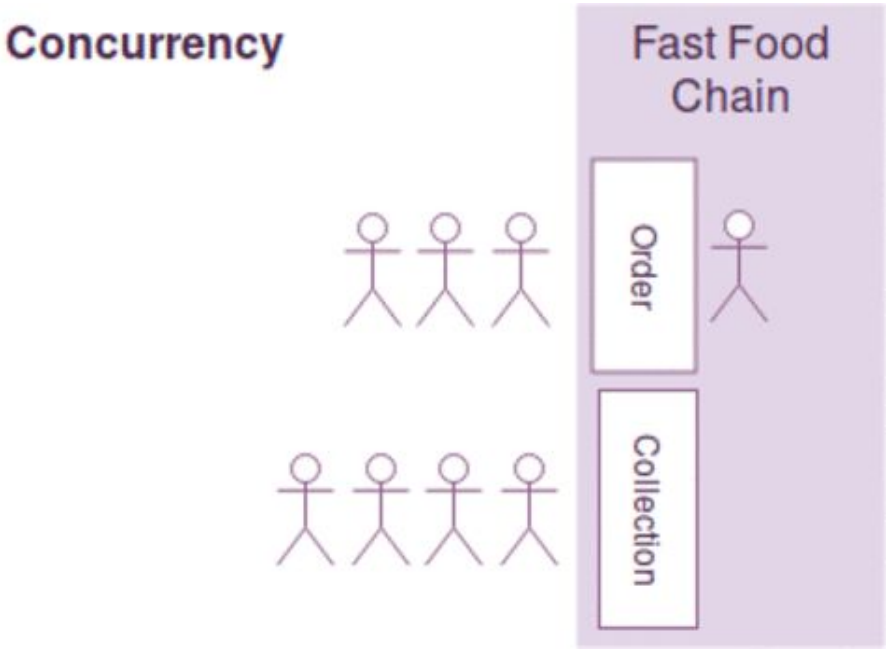
A system is said to be concurrent if it can support two or more actions in progress at the same time.

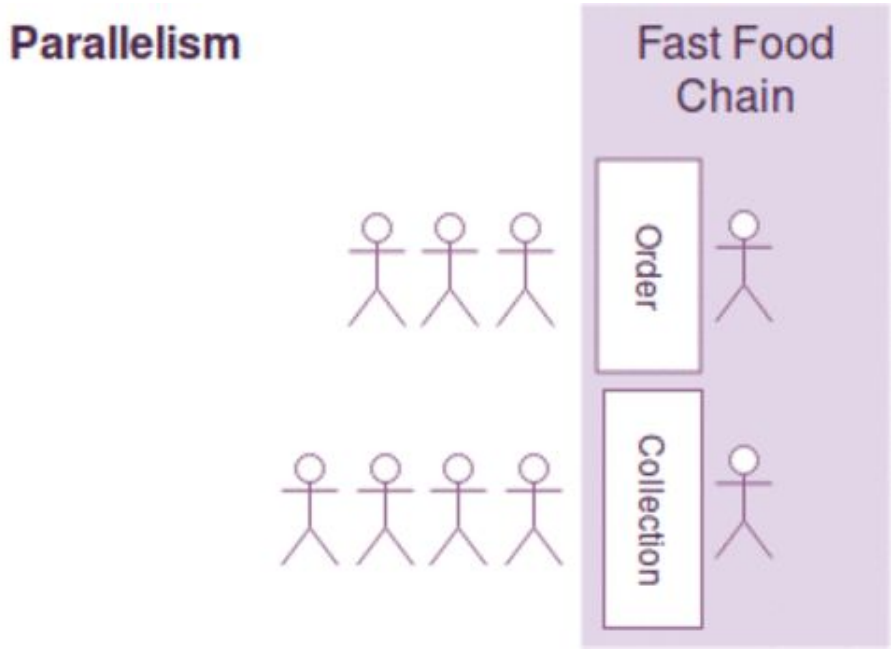
A system is said to be parallel if it can support two or more actions executing simultaneously.

The key concept and difference between these definitions is the word “simultaneously”.

Concurrency is about dealing with lots of things at once.
Parallelism is about doing lots of things at once.

Concurrency vs Parallelism





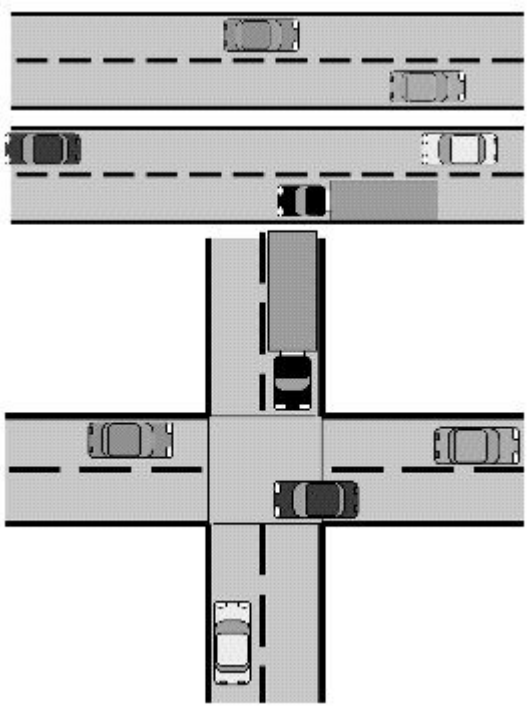
Concurrency vs Parallelism

A problem has to be concurrent in nature before it can be executed in parallel.

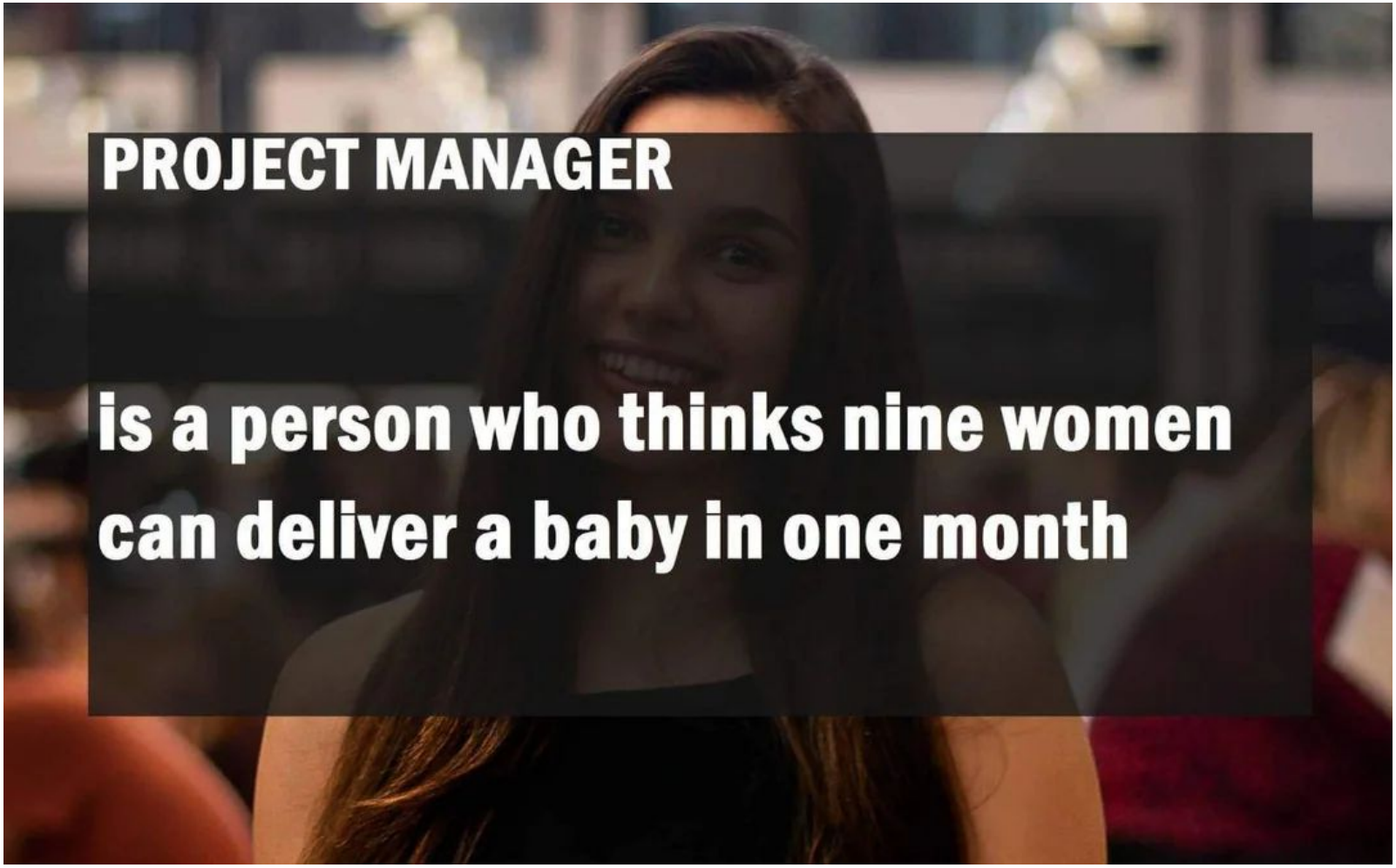
Parallel executions are always concurrent.

Concurrent executions may not be parallel.

Programs may not always benefit from parallel execution with respect to concurrent execution.



Parallel nature of problems



Inherently sequential

Not all programs can be made parallel or concurrent.

A program (or portion of a program) that cannot be executed concurrently or in parallel is called inherently sequential.

Real-world example: pregnancy, the number of women will never reduce the length of gestation.

Programming example: computation of $f^{1000}(3)$, with $f(x)=x^2-x+1$, $f^0(x)=f(x)$, and $f^{n+1}=f(f^n(x))$

Embarrassingly parallel

A program (or portion of a program) is called embarrassingly parallel if it can be divided into different parallel tasks, between which there is little to no dependency or need for communication.

Real-world example: peel a crate of potatoes

Programming example: image processing (GPU working principle)

Concurrent and parallel programs

Every program is composed by:

- at least one inherently sequential portion (usually are more than one),
- the inherently sequential portion can also be 100%,
- portions that can be made asynchronous,
- portions that can be run in random order (concurrently),
- portions that can be run in parallel.

Concurrent and parallel programs: example

In our cooking example:

- peeling a single potato or slicing a single carrot are inherently sequential operations,
- boiling is an asynchronous operation,
- peeling and slicing multiple vegetables can be executed in a different order (concurrently). They can be executed in parallel if there is more than one cook.

Amdahl's Law

Parallelizing software

Converting a sequential program into a concurrent or parallel execution is not a trivial task.

Moreover, concurrency and parallelism are not a silver bullet that can speed up any non-sequential architecture infinitely and unconditionally.

Question

Which is the better?

- Having 4 processors running a given program with 40% of its instructions parallelizable.

- Having 2 processors running a given program with 80% of its instructions parallelizable.

Answer

Answering such question is not trivial.

Gene Myron Amdahl, a mainframe architect at IBM, suggested a formula that defines a theoretical maximum for the ability of systems to scale up.

Amdahl's Law provides a mathematical formula that calculates the potential improvement in speed of a concurrent program by increasing its resources (specifically, the number of available processors).

Amdahl's Law

$$Speedup = \frac{1}{B + \frac{1-B}{j}}$$

$$\text{Speedup} = \frac{\text{time it takes to execute sequentially}}{\text{time it takes to execute in parallel}}$$

where:

- j = number of processors
- B = inherently sequential fraction of the program

Amdahl's Law

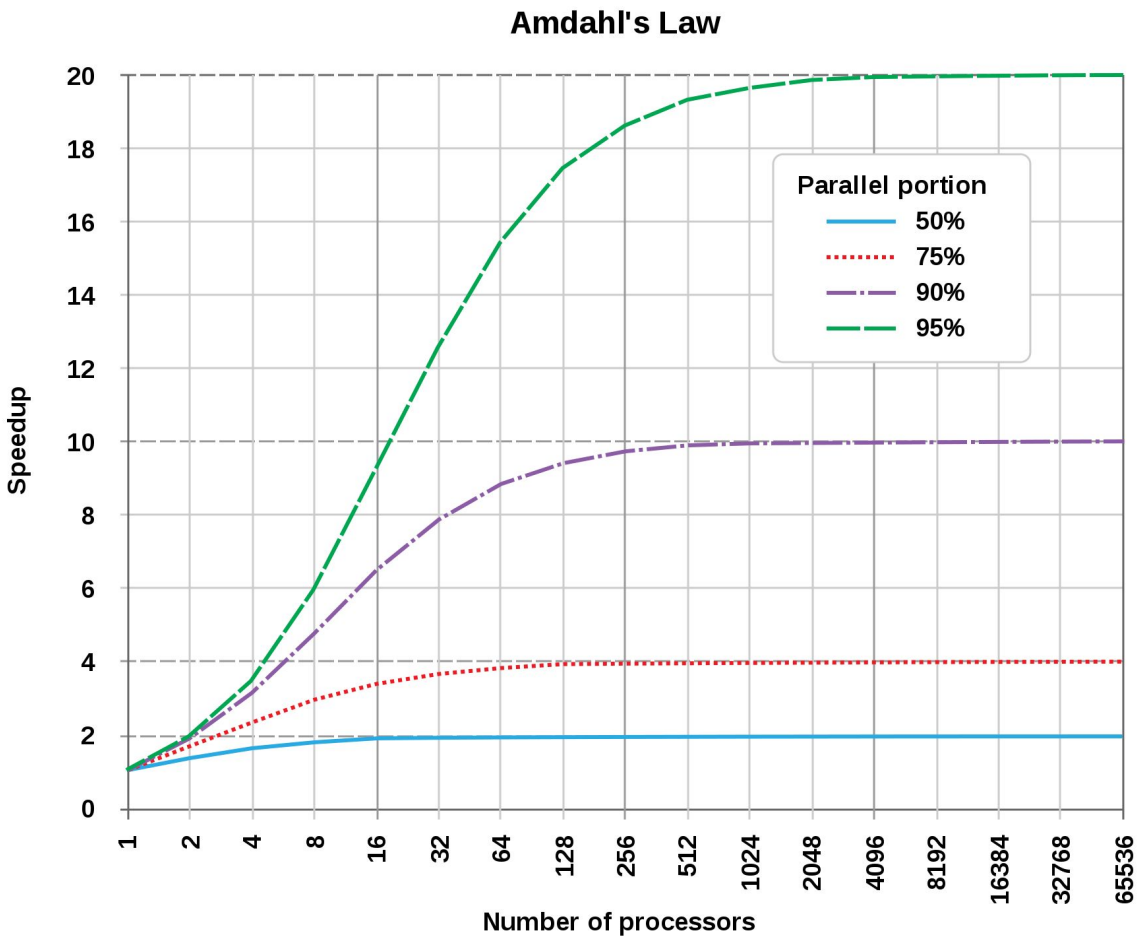
Assuming the resources (processors) to grow up indefinitely, we get:

$$\lim_{j \to \infty} Speedup = \frac{1}{B}$$

Given a program with a inherently sequential portion of B = 10% = 0.1 we get:

Processors	Speedup
1	1
2	1.81
5	3.57
10	5.26
100	9.17
1000	9.91
∞	**10**

Amdahl's Law


Amdahl's Law
Speedup
Number of processors
Parallel portion
50%
75%
90%
95%
0
2
4
6
8
10
12
14
16
18
20
1
2
4
8
16
32
64
128
256
512
1024
2048
4096
8192
16384
32768
65536

Amdahl's Law

Amdahl's Law offers a theoretical estimation of potential speedup.

It makes a number of underlying assumptions and does not take into account some potentially important factors, such as the overhead of parallelism or the speed of memory.

In the end, only actual measurements can precisely answer our questions about how much speedup our concurrent programs will achieve in practice.

Back to previous question…

Which is the better?

- Having 4 processors running a given program with 40% of its instructions parallelizable.

- Having 2 processors running a given program with 80% of its instructions parallelizable.

Answer

Applying Amdahl's Law we have the following:

- Having 4 processors running a given program with 40% of its instructions parallelizable.
  Speedup = `1 / (0.6 + 0.4 / 4) = 1.43`

- Having 2 processors running a given program with 80% of its instructions parallelizable.
  Speedup = `1 / (0.2 + 0.8 / 2) =` `1.67`

Amdahl's Law - Example

What: check if a number is prime

Input: all integer number between 1013 and 1013 + 1000

How: on lecturer laptop using 1, 2, 3, ..., 8 workers

Output:

```
Number of workers: 1. Took: 4.93
seconds. ====================
Number of workers: 2. Took: 2.50
seconds. ====================
Number of workers: 3. Took: 1.71
seconds. ====================
Number of workers: 4. Took: 1.32
seconds.
```

```
Number of workers: 5. Took: 1.47
seconds. ====================
Number of workers: 6. Took: 1.48
seconds. ====================
Number of workers: 7. Took: 1.97
seconds. ====================
Number of workers: 8. Took: 2.31
seconds.
```

Considerable improvements: from 1 to 3 workers. Hardly any speedup was achieved from 3 to 4. It took longer during the next iterations: this is most likely overhead processing.

Summary

- Moore’s Law
- Sequential execution
- Concurrent execution
- Parallel execution
- Amdahl’s law