Parallel Processing in Python

Presented by:
Ong Chin Hwee (@ongchinhwee)

27 August 2019
Python User Group Singapore, Aug 2019 Meetup

About me

- Current role: Data Engineer at ST Engineering
- Background in aerospace engineering and computational modelling
- Experience working on aerospace-related projects in collaboration with academia and industry partners
- Find me if you would like to chat about Industry 4.0 and flight + travel!

A typical data science workflow

1. Define problem objective
2. Data collection and pipeline
3. Data parsing/preprocessing and Exploratory Data Analysis (EDA)
4. Feature engineering
5. Model training
6. Model evaluation
7. Visualization and Reporting
8. Model deployment

What do you think are some of the bottlenecks in a data science project?

Bottlenecks in a data science project

- Lack of data / Poor quality data
- Data Preprocessing
  - The 80/20 data science dilemma
    - In reality, it's closer to 90/10
- The organization itself

Bottlenecks in a data science project

- Data Preprocessing
    - Pandas faces low performance and long runtime issues when dealing with large datasets (> 1 GB)

Bottlenecks in a data science project

- Data Preprocessing
  - Pandas faces low performance and long runtime issues when dealing with large datasets (> 1 GB)
  - Slow loops in Python
    - Loops are run on the interpreter, not compiled (unlike loops in C)

Scope of Talk

I will talk about:

1. What is parallel processing?

Scope of Talk

I will talk about:

1. What is parallel processing?
2. Synchronous vs asynchronous execution

Scope of Talk

I will talk about:

1. What is parallel processing?
2. Synchronous vs asynchronous execution
3. When should you go for parallelism?

Scope of Talk

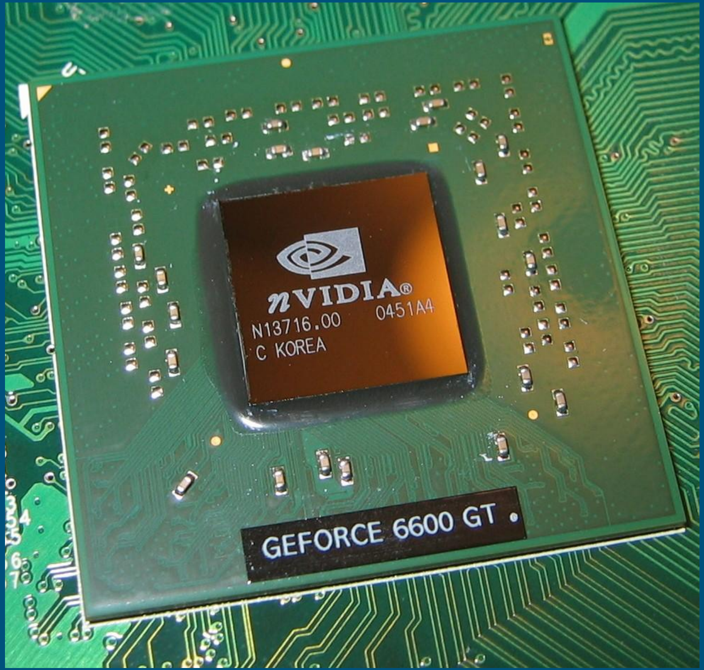
I will talk about:

1. What is parallel processing?
2. Synchronous vs asynchronous execution
3. When should you go for parallelism?
4. Parallel processing in Python

Scope of Talk

I will **NOT** talk about:

1. Parallel computing hardware



Scope of Talk

I will **NOT** talk about:

1. Parallel computing hardware
2. Apache Spark, MapReduce etc.

Scope of Talk

I will **NOT** talk about:

1. Parallel computing hardware
2. Apache Spark, MapReduce etc.
3. The GIL controversy

What is parallel processing?

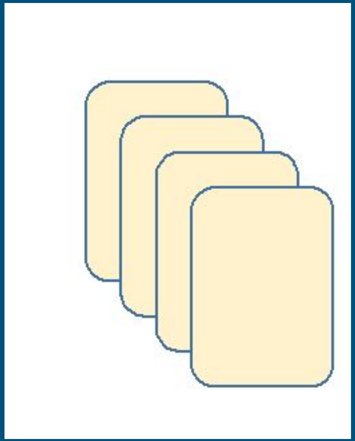
Let’s imagine I own a bakery cafe.

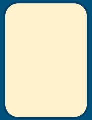
Task 1: Toast 100 slices of bread

Assumptions:
1. I'm using single-slice toasters. (Yes, they actually exist.)
2. Each slice of toast takes 2 minutes to make.
3. No overhead time.

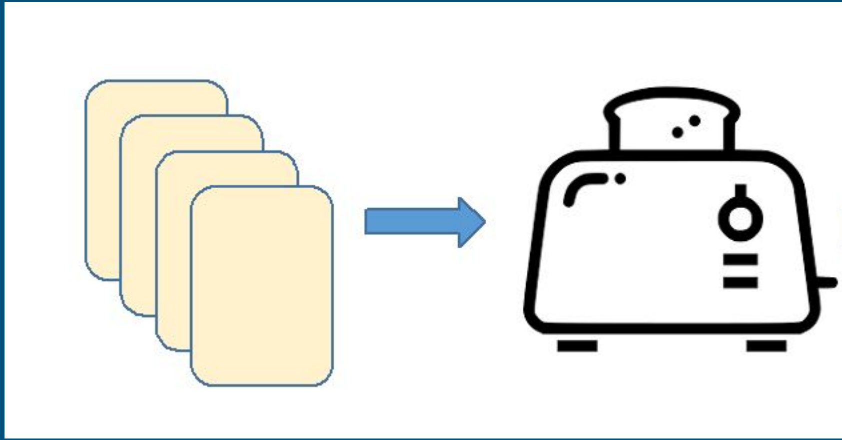
Image taken from: https://www.mitsubishielectric.co.jp/home/breadoven/product/to-st1-t/feature/index.html

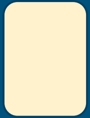
Sequential Processing



 = 25 bread slices

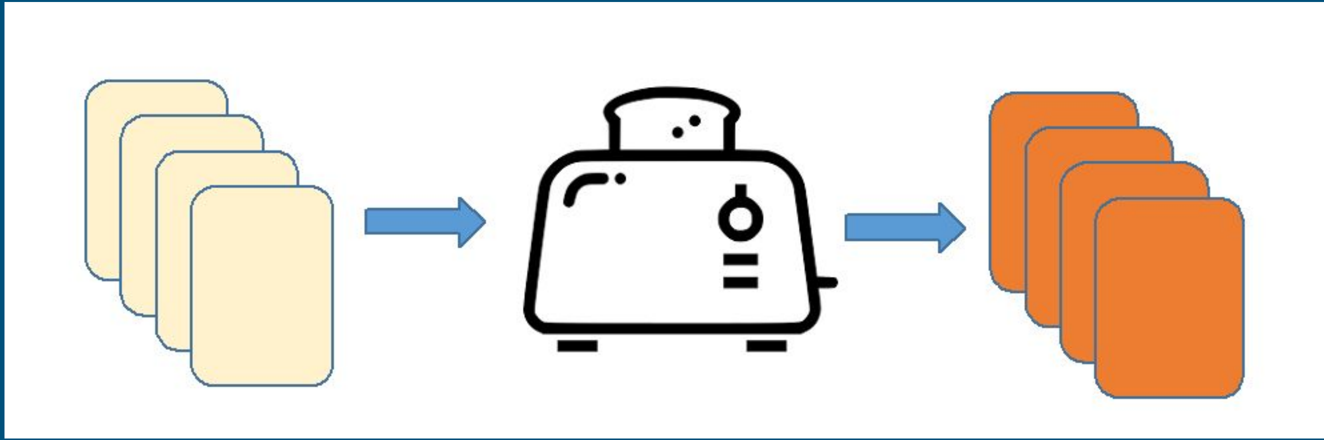
Sequential Processing



 = 25 bread slices

Processor/Worker:
Toaster

Sequential Processing

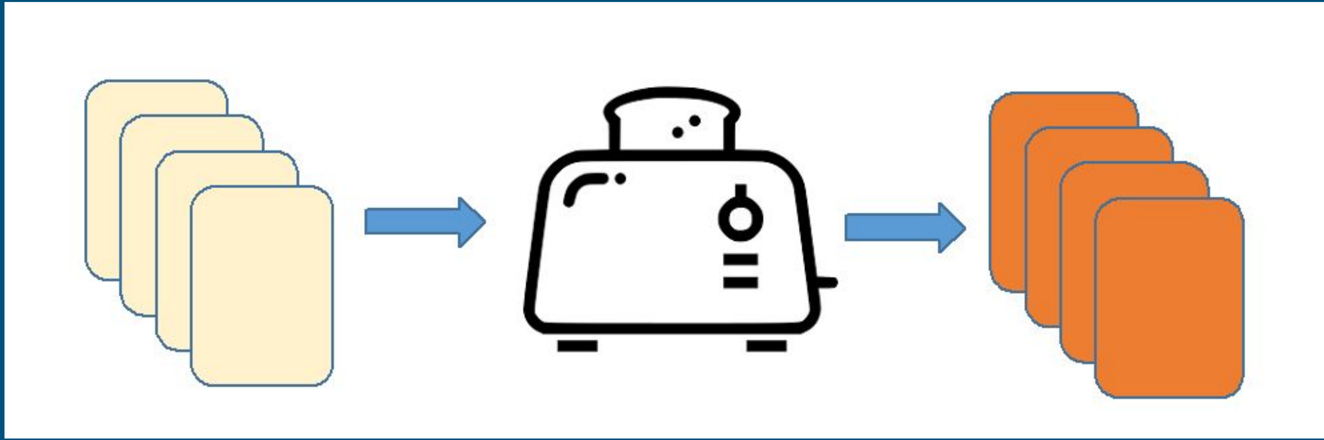


= 25 bread slices

Processor/Worker: Toaster

= 25 toasts

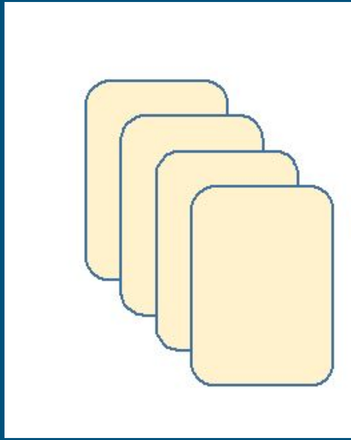
Sequential Processing

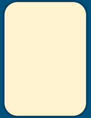


Execution Time = 100 toasts × 2 minutes/toast

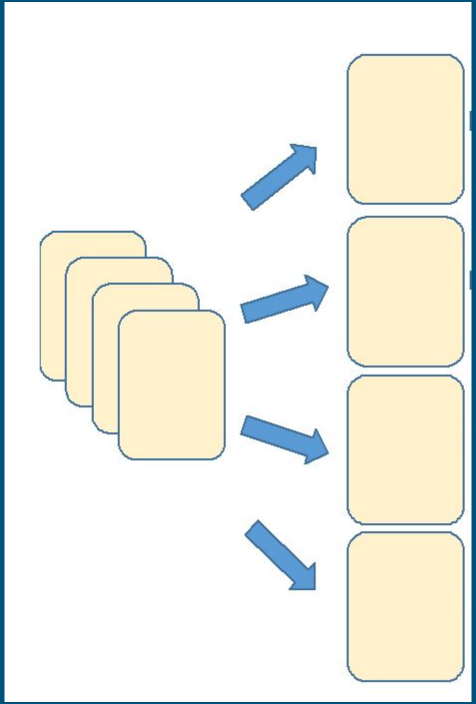
= <u>**200 minutes**</u>

Parallel Processing

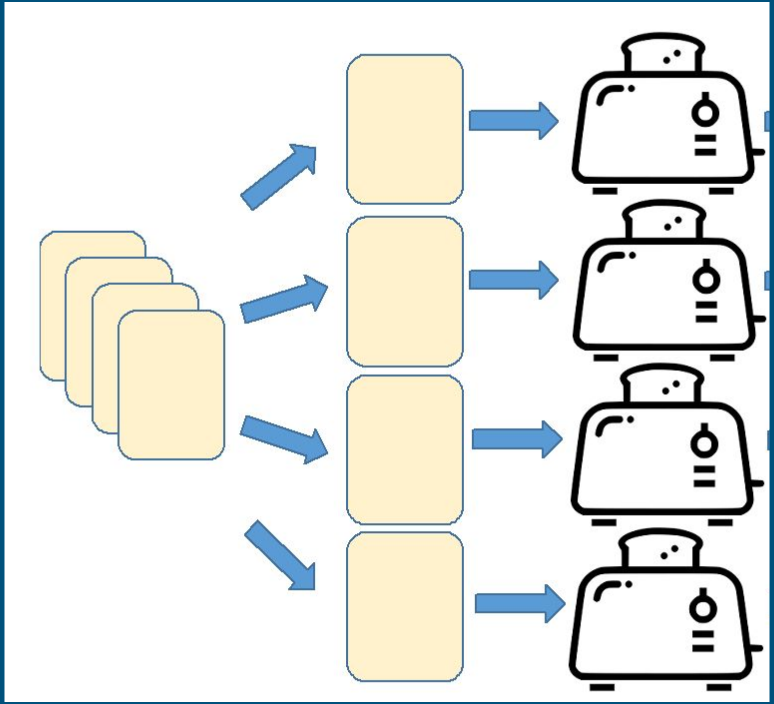


 = 25 bread slices

Parallel Processing

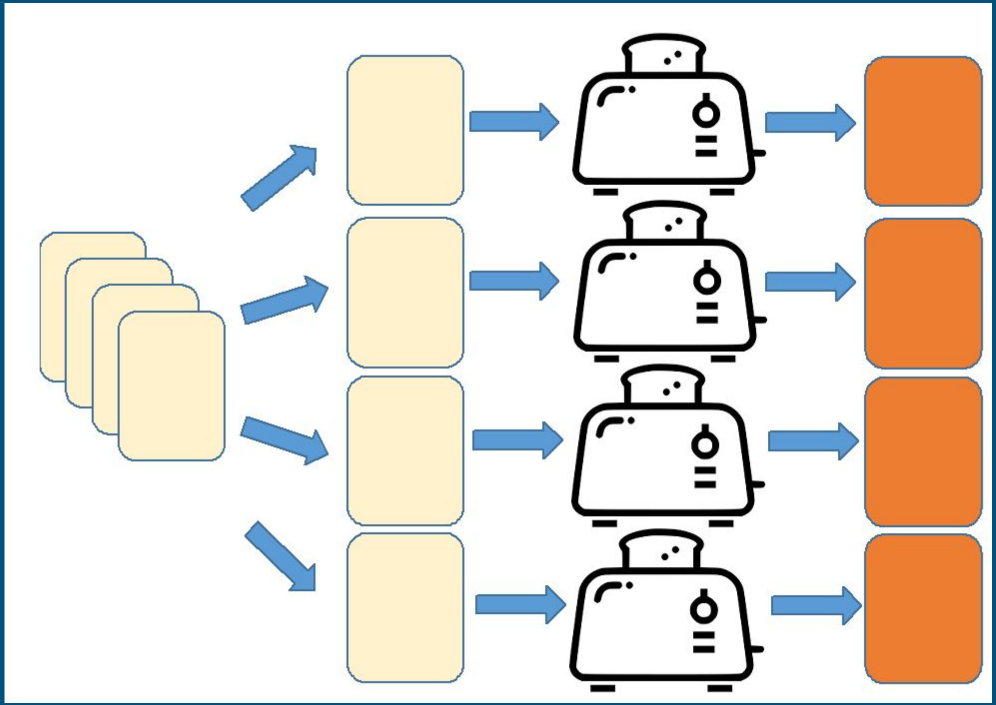


Parallel Processing



Processor (Core):
Toaster

Parallel Processing

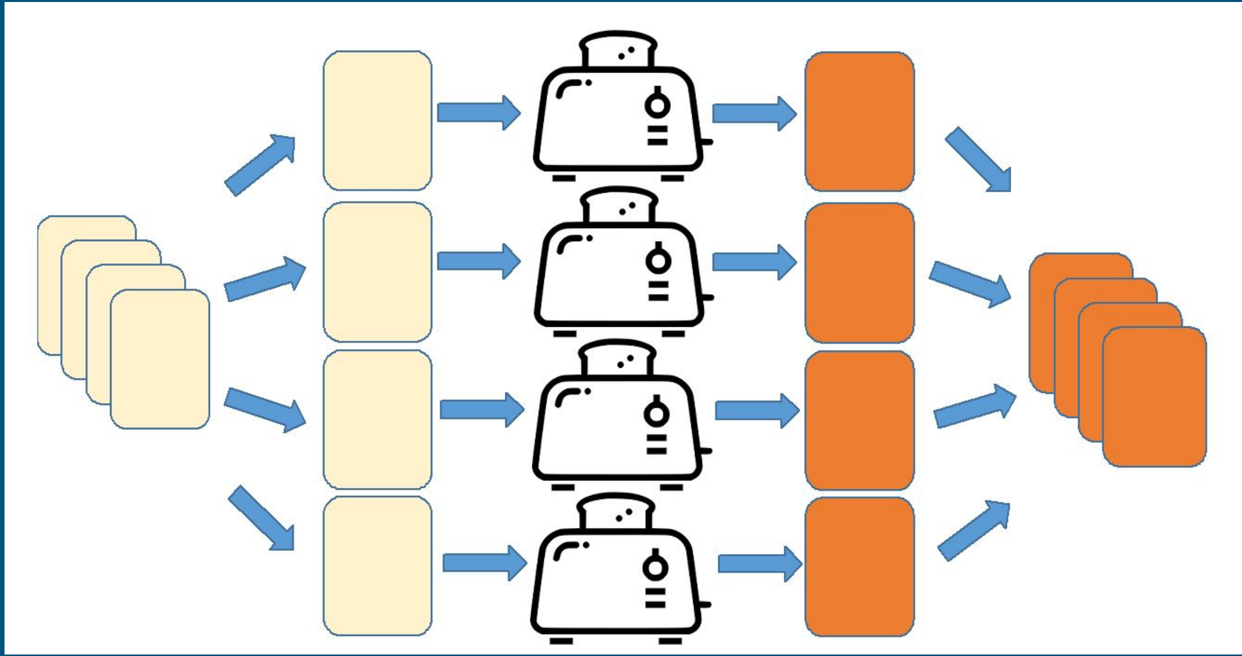


Processor (Core): Toaster

Task is executed using a **pool** of **4 toaster subprocesses**.

Each toasting subprocess runs **in parallel** and **independently** from each other.

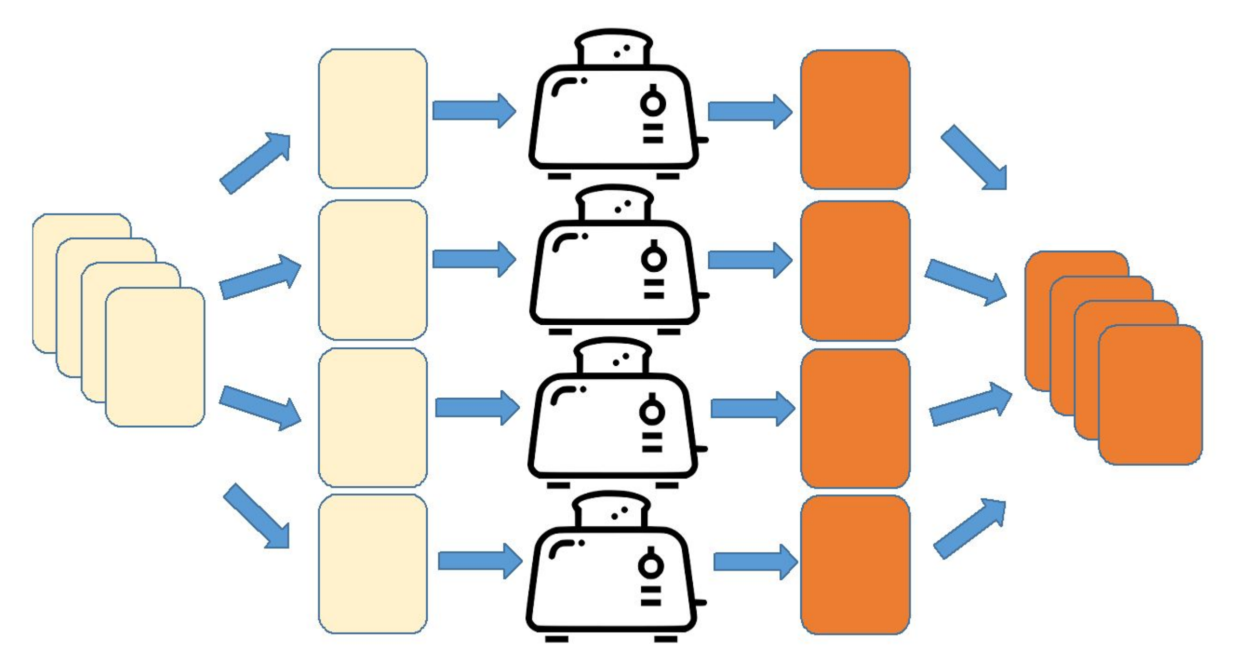
Parallel Processing



Processor (Core): Toaster

Output of each toasting process is **consolidated** and **returned** as an overall output (which may or may not be ordered).

Parallel Processing



Execution Time
= 100 toasts × 2 minutes/toast ÷ 4 toasters
= **<u>50 minutes</u>**

Speedup
= **<u>4 times</u>**

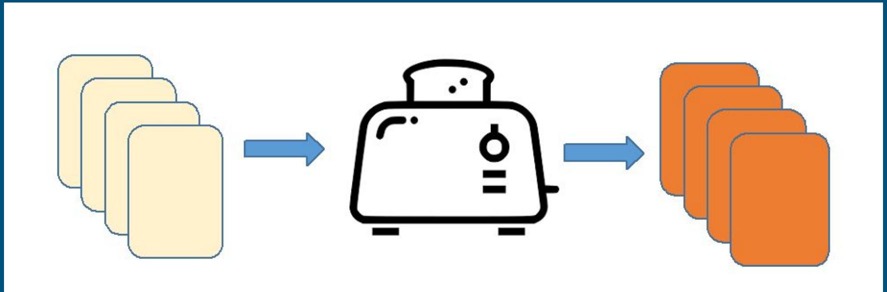
Synchronous vs Asynchronous Execution

Task 2: Brew 100 cups of coffee

Assumptions:
1. I am doing BOTH tasks 1 and 2 (toast + coffee).
2. One coffee maker to make one cup of coffee.
3. Each cup of coffee takes 5 minutes to make.

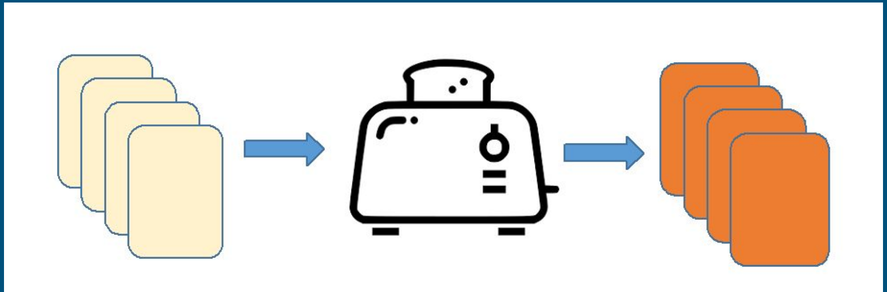
Image taken from: https://www.crateandbarrel.com/breville-barista-espresso-machine/s267619

Synchronous Execution



Process 1: Toast a slice of bread on single-slice toaster
Duration: 2 minutes

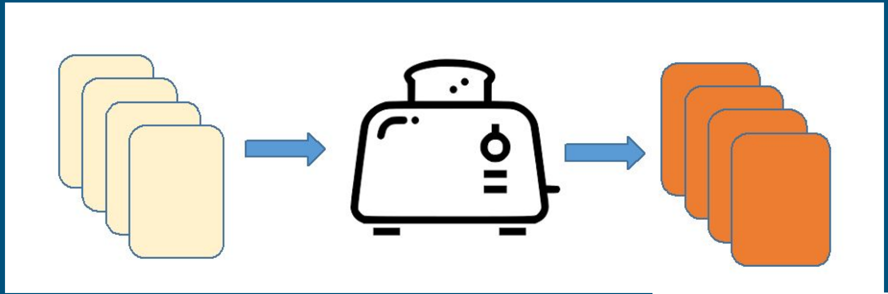
Synchronous Execution



Process 2: Brew a cup of coffee on coffee machine
Duration: 5 minutes

Process 1: Toast a slice of bread on single-slice toaster
Duration: 2 minutes

Synchronous Execution



Process 2: Brew a cup of coffee on coffee machine
Duration: 5 minutes

Process 1: Toast a slice of bread on single-slice toaster
Duration: 2 minutes

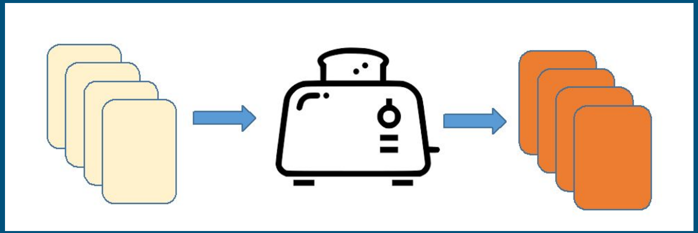
Output: **1 toast + 1 coffee**

Total Execution Time = 2 minutes + 5 minutes = **7 minutes**

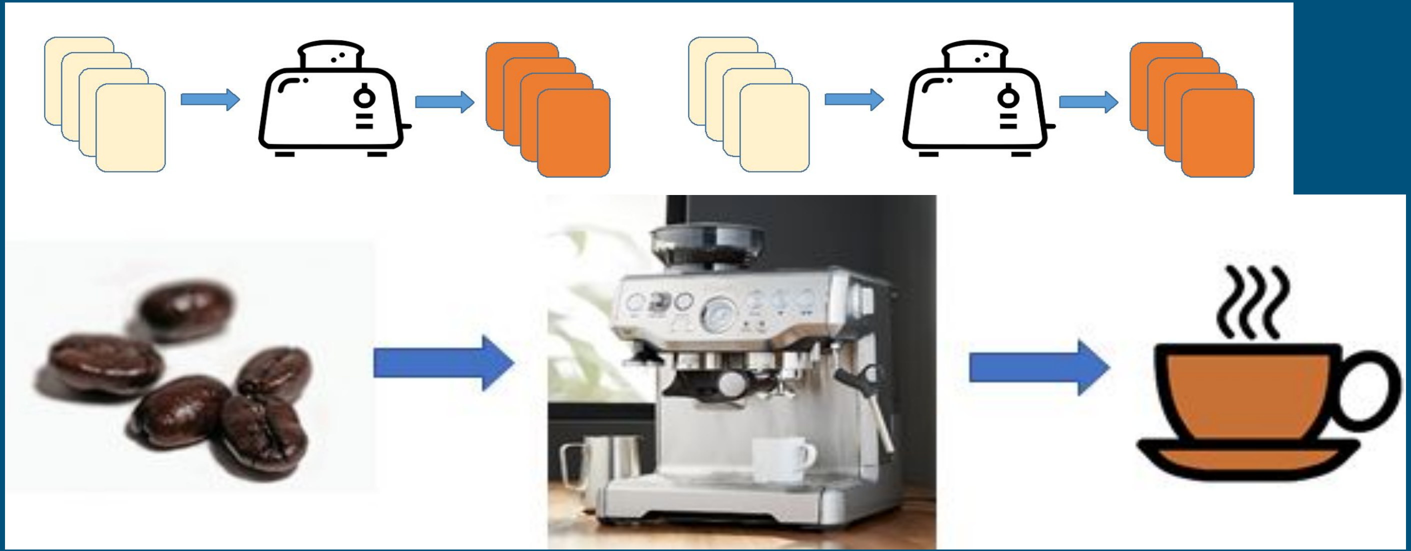
Asynchronous Execution

While brewing coffee:

Make some toasts:



Asynchronous Execution



Output: **2 toasts + 1 coffee** (1 more toast!)
Total Execution Time = **5 minutes**

When is it a good idea to go for parallelism?

When is it a good idea to go for parallelism?

(or, “Is it a good idea to simply buy a 256-core processor and parallelize all your codes?”)

Practical Considerations

- Is your code already optimized?
  - Sometimes, you might need to rethink your approach.
  - Example: Use list comprehensions instead of for-loops for array iterations.

Practical Considerations

- Is your code already optimized?
  - Sometimes, you might need to rethink your approach.
- Problem architecture
  - Nature of problem limits how successful parallelization can be.
  - If your problem consists of processes which depend on each others' outputs, maybe not.

Practical Considerations

- Is your code already optimized?
  - Sometimes, you might need to rethink your approach.
- Problem architecture
  - Nature of problem limits how successful parallelization can be.
- Overhead in parallelism
  - There will always be parts of the work that cannot be parallelized. → **Amdahl's Law**
  - Extra time required for coding and debugging (parallelism vs sequential code)

Amdahl's Law and Parallelism

Amdahl's Law states that the theoretical speedup is defined by the fraction of code ***p*** that can be parallelized:

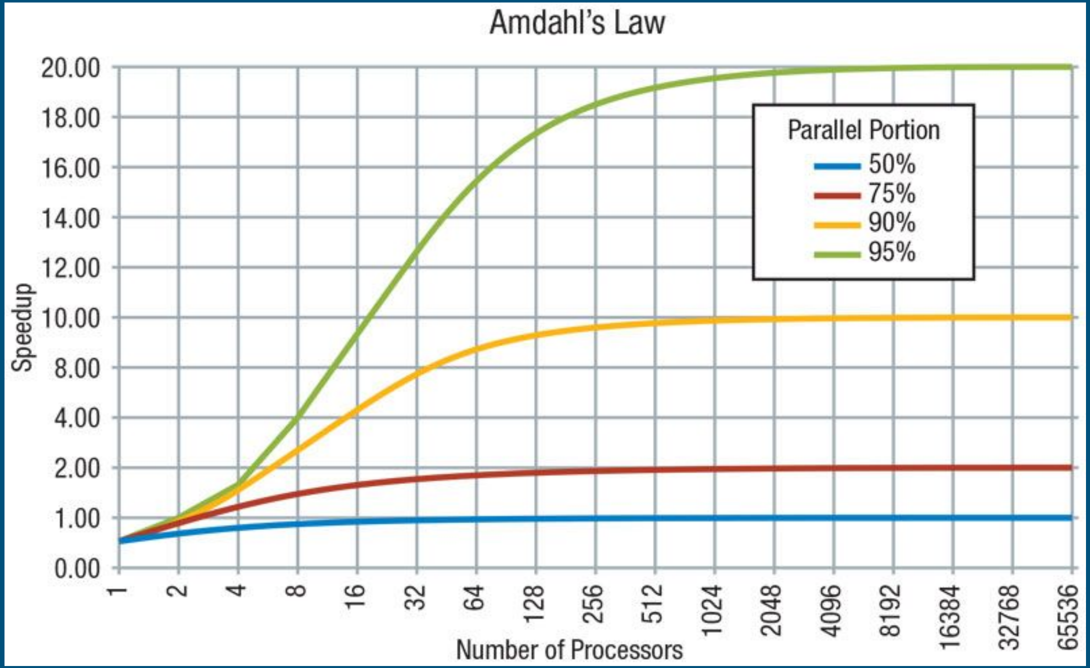
$$S = \frac{1}{(1-p) + \frac{p}{N}}$$

S: Theoretical speedup (theoretical latency)
p: Fraction of the code that can be parallelized
N: Number of processors (cores)

Amdahl's Law and Parallelism

If there are **no parallel parts** (*p* = 0): **<u>Speedup = 0</u>**

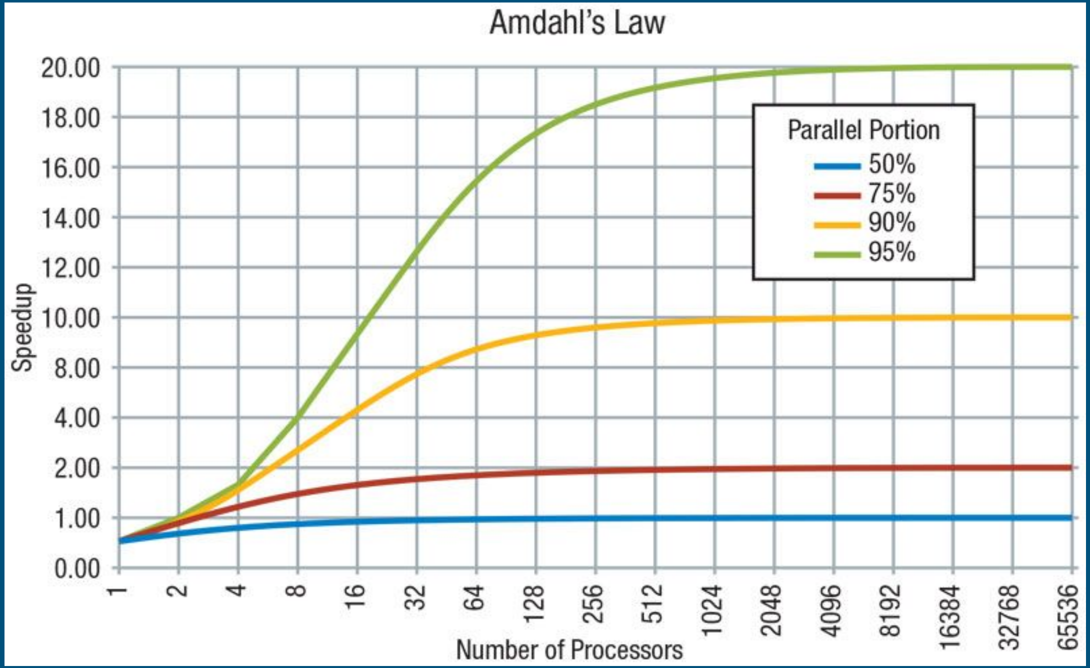
If **all parts are parallel** (*p* = 1):



Amdahl's Law and Parallelism

If there are **no parallel parts** (p = 0): **Speedup = 0**

If **all parts are parallel** (p = 1): **Speedup → ∞**

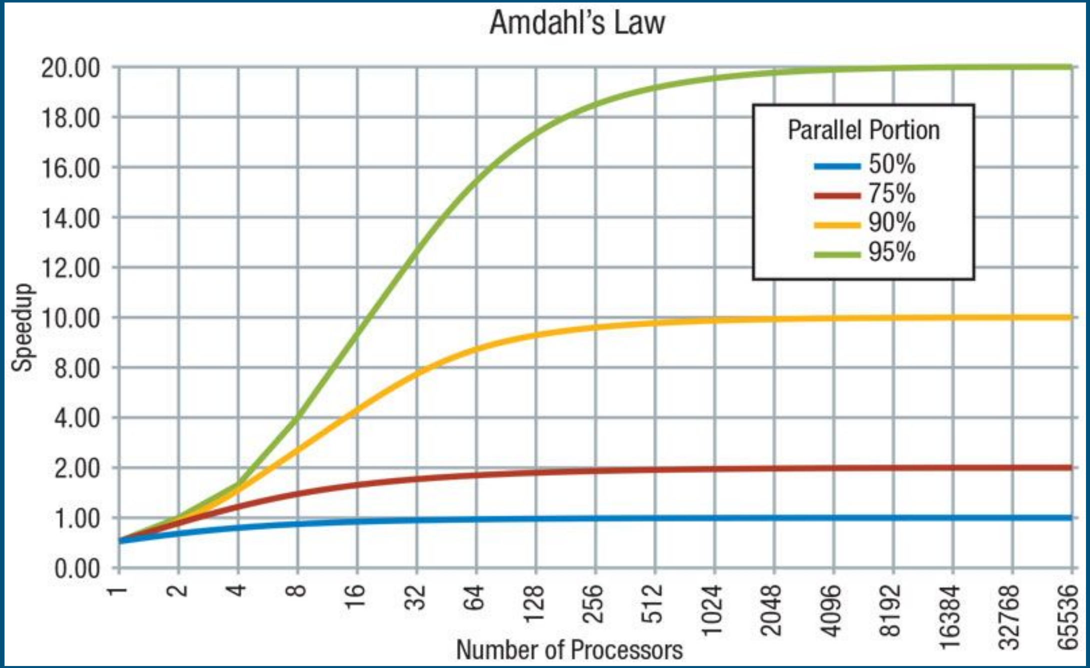


Amdahl's Law and Parallelism

If there are **no parallel parts** (p = 0): **<u>Speedup = 0</u>**

If **all parts are parallel** (p = 1): **<u>Speedup → ∞</u>**

Speedup is <u>limited</u> by **fraction of the work that is not parallelizable** - will not improve **<u>even with infinite number of processors</u>**



Parallel Processing in Python Programming

Why is parallelism so tricky in Python?

Global Interpreter Lock (GIL)

In CPython, the GIL is a **mutex** (mutually exclusive object) that allows **only one thread** to hold control of the Python interpreter **at once**.

Necessary mainly because CPython's memory management is **not fully thread-safe**.

While non-parallel codes can run faster with the GIL, it <u>does not allow parallel thread execution.</u>

Why is parallelism so tricky in Python?

Global Interpreter Lock (GIL)

However, stuff that happens <u>outside the GIL realm</u> (I/O and libraries written on external C codes) can **bypass the GIL.**

How to do Parallel Processing in Python?

Parallel Processing in Python

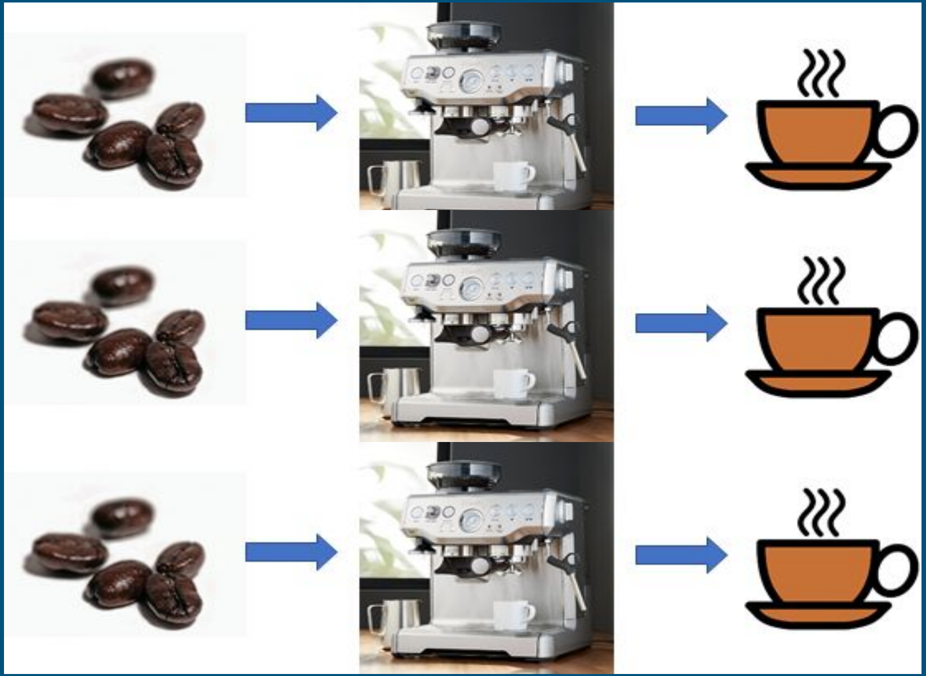
concurrent.futures module

- High-level API for launching **asynchronous parallel tasks**
- Introduced in Python 3.2 as an abstraction layer over **multiprocessing** module
- Two modes of execution:
  - *ThreadPoolExecutor()* for multithreading
  - *ProcessPoolExecutor()* for multiprocessing

Multiprocessing vs Multithreading

Multiprocessing:

System allows executing multiple processes at the same time using multiple processors



Multiprocessing vs Multithreading

Multiprocessing:

System allows executing multiple processes at the same time using multiple processors

Multithreading:

System executes multiple threads of sub-processes at the same time within a single processor

Multiprocessing vs Multithreading

Multiprocessing:

System allows executing multiple processes at the same time using multiple processors

Better option for **processing large volumes of data**

Multithreading:

System executes multiple threads of sub-processes at the same time within a single processor

Best suited for **I/O operations**

ProcessPoolExecutor vs ThreadPoolExecutor

From the Python Standard Library documentation:

For *ProcessPoolExecutor,* this method chops iterables into a number of chunks which it submits to the pool as separate tasks. The (approximate) size of these chunks can be specified by setting chunksize to a positive integer. For very long iterables, using a large value for chunksize can significantly improve performance compared to the default size of 1. With *ThreadPoolExecutor*, chunksize has no effect.

Recap: map() in Python Built-in Functions

map() takes as input:

1. The <u>function</u> that you would like to run, and
2. A <u>list (iterable)</u> where each element of the list is a single input to that function;

and returns an <u>iterator</u> that <u>**yields**</u> the results of the function being applied to every element of the list.

map() in concurrent.futures

Similarly, **executor.map()** takes as input:

1. The function that you would like to run, and
2. A list (iterable) where each element of the list is a single input to that function;

and returns an iterator that **yields** the results of the function being applied to every element of the list.

Practical Implementation

Case: Image Processing

Dataset: Shopee National Data Science Challenge (https://www.kaggle.com/c/ndsc-advanced)

Size: **77.6GB** of image files

Data Quality: Images in the dataset are of **different formats** (some are RGB while others are RGBA) and **different dimensions**

Image Resizing Code

```
from PIL import Image

def image_proc(index):
      '''Convert + resize image'''
      im = Image.open(define_imagepath(index))
      im = im.convert("RGB")
      im_resized = np.array(im.resize((64,64)))

      return im_resized
```

Batch Processing Code

```
def arraypartition_calc(start, batch_size):
    '''Process images in partition/batch'''
    end = start + batch_size
    if end > N:
        end = N

    partition_list = [image_proc(image) for image
in range(start, end)]

    return partition_list
```

Without Parallelism

```
import sys
import time

N = 35000 # size of dataset to be processed
start = 0
batch_size = 1000
partition = int(np.ceil(N/step))
partition_count = 0
imagearray_list = [None] * partition

start_cpu_time = time.clock()
start_wall_time = time.time()
```

Without Parallelism

```
while start < N:

      end = start + batch_size
      if end > N:
            end = N

      imagearray_list[partition_count] =
[arraypartition_calc(image) for image in range(start, end)]

      start += batch_size
      partition_count += 1
```

Without Parallelism

```
while start < N:

	end = start + batch_size
	if end > N:
		end = N

	imagearray_list[partition_count] = 
[arraypartition_calc(image) for image in range(start, end)]

	start += batch_size
	partition_count += 1
```

Without Parallelism

```
while start < N:

      end = start + batch_size
      if end > N:
            end = N

      imagearray_list[partition_count] = 
[arraypartition_calc(image) for image in range(start, end)]

      start += batch_size
      partition_count += 1
```

Execution Speed:
3300 images after 7 hours
= 471.43 images/hr

With Parallelism

```
N = 35000
start = 0
batch_size = 1000
partition, mod = divmod(N, batch_size)

if mod:
    partition_index = [i * batch_size for i in range(start //
batch_size, partition + 1)]
else:
    partition_index = [i * batch_size for i in range(start //
batch_size, partition)]
```

With Parallelism

```
import sys
import time
from concurrent.futures import ProcessPoolExecutor

start_cpu_time = time.clock()
start_wall_time = time.time()

with ProcessPoolExecutor() as executor:
      future = executor.map(arraypartition_calc, partition_index)

imgarray_np = np.array([x for x in future])
```

With Parallelism

Execution Speed:
35000 images after 3.6 hours
= 9722.22 images/hr

```
import sys
import time
from concurrent.futures import ProcessPoolExecutor

start_cpu_time = time.clock()
start_wall_time = time.time()

with ProcessPoolExecutor() as executor:
      future = executor.map(arraypartition_calc, partition_index)

imgarray_np = np.array([x for x in future])
```

With Parallelism

Execution Speed:
35000 images after 3.6 hours
= 9722.22 images/hr

Speedup: 19.4 times

```
import sys
import time
from concurrent.futures import ProcessPoolExecutor

start_cpu_time = time.clock()
start_wall_time = time.time()

with ProcessPoolExecutor() as executor:
      future = executor.map(arraypartition_calc, partition_index)

imgarray_np = np.array([x for x in future])
```

With Parallelism

```
import sys
import time
from concurrent.futures import ProcessPoolExecutor

start_cpu_time = time.clock()
start_wall_time = time.time()

with ProcessPoolExecutor() as executor:
      future = executor.map(arraypartition_calc, partition_index)

imgarray_np = np.array([x for x in future])
```

Extract results from iterator (similar to generator)

Key Takeaways

Parallel Processing in Python

concurrent.futures module:

- Provides **hassle-free, high-level** implementation of parallel processing without delving into parallelization architectures
- Part of **Python Standard Library** since Python 3.2 - no need for additional installations!

Parallel Processing in Python

- Not all processes should be parallelized
  - **Amdahl's Law** on parallelism
  - Extra time required for coding and debugging (parallelism vs sequential code)
  - If the cost of rewriting your code for parallelization outweighs the time savings from parallelizing your code (especially if your process only takes a few hours), maybe you should consider **other ways of optimizing your code** instead.

References

Official Python documentation on concurrent.futures (https://docs.python.org/3/library/concurrent.futures.html)

Built-in Functions - Python 3.7.4 Documentation (https://docs.python.org/3/library/functions.html#map)

First Programs and How to Think in CUDA. (2011). CUDA Application Design and Development, 1–31. doi:10.1016/b978-0-12-388426-8.00001-x

Contact

Ong Chin Hwee
LinkedIn: ongchinhwee
Twitter: @ongchinhwee
https://ongchinhwee.me

Women Who Code Connect Asia 2019

Women Who Code Connect Asia 2019

Date: August 31st 2019 (Saturday)
Time: 8:00 AM – 6:00 PM SGT
Venue: Lifelong Learning Institute

For anyone purchasing using Singapore Dollars:
Member price - $75 USD : $100 SGD
Open Collective link:
https://opencollective.com/wwcodeconnectasia/events/women-who-code-connect-asia-2019-august-31-st-member-tickets-50595ev