Introduction to Programming with Python

Dr. Anatol Wegner

Chair of Machine Learning for Complex Networks
Center for Artificial Intelligence and Data Science (CAIDAS)
Julius-Maximilians-Universität Würzburg
Würzburg, Germany

anatol.wegner@uni-wuerzburg.de

Lecture 10
Parallel Processing

January 24, 2025

Recap

- **Databases:** Organized collections of data for efficient storage, retrieval, and analysis.
- **Relational databases:** Data is organized in tables with rows and columns.
- **SQL:** A powerful language for interacting with relational databases.
- **SQLite:** A lightweight, file-based database system ideal for learning and small projects.
- **Key concepts:** Tables, columns, rows, primary keys, foreign keys, SQL commands (‘SELECT‘, ‘INSERT‘, ‘UPDATE‘, ‘DELETE‘).

Recap

- **Databases:** Organized collections of data for efficient storage, retrieval, and analysis.
- **Relational databases:** Data is organized in tables with rows and columns.
- **SQL:** A powerful language for interacting with relational databases.
- **SQLite:** A lightweight, file-based database system ideal for learning and small projects.
- **Key concepts:** Tables, columns, rows, primary keys, foreign keys, SQL commands (‘SELECT‘, ‘INSERT‘, ‘UPDATE‘, ‘DELETE‘).

Today: Parallel Processing

- Multiprocessing basics
- Parallel processing in Python using the `multiprocessing` module

- **What is Parallel Processing?**
  - Executing multiple parts of a program **simultaneously** on multiple CPU cores.
  - In contrast sequential processing executes instructions one after the other.
- **Why is it Important?**
  - Significantly speed up computationally intensive tasks.
  - Efficient use of modern multi-core processors.
- **Real-World Examples:**
  - Scientific simulations (e.g., weather forecasting, physics simulations).
  - Large-scale data analysis and processing.
  - Machine Learning model training especially Artificial Neural Networks.
  - Image and video processing.

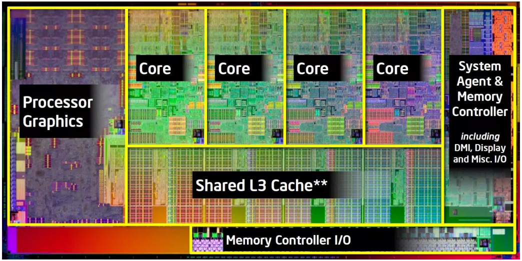
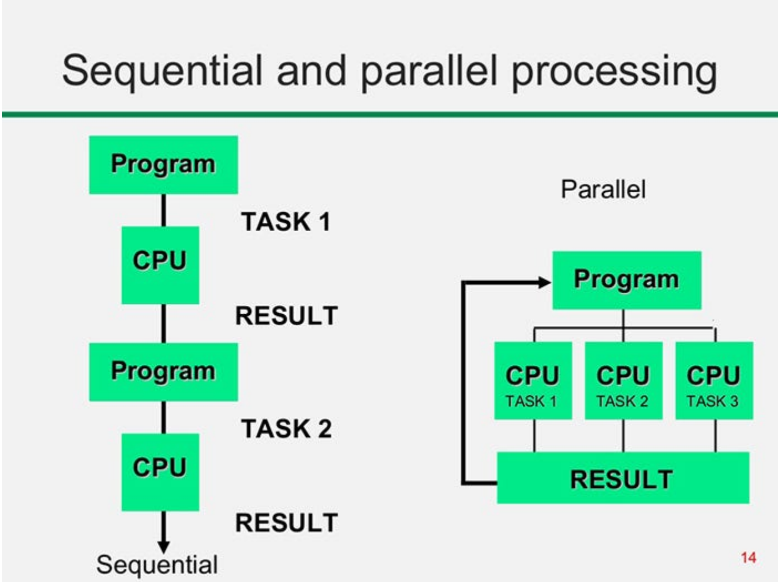


Figure: A multi core CPU



Sequential and parallel processing

Processes and the Operating System

- **What is a Process?**
  - An **independent** execution environment.
  - Has its own memory space, resources (files, open sockets, etc.), and process ID (PID).
  - Managed by the Operating System (OS).
- **Operating System and Process Management:**
  - The OS is responsible for:
    - Creating and terminating processes.
    - Allocating resources to processes.
    - Scheduling processes to run on CPU cores (process scheduling).
  - Processes are isolated from each other. Changes in one process do not affect other processes (by default).

Process Creation

- When you start a program, the OS creates a new process for it.
- The program's code, data, and other resources are loaded into the process's memory space.
- The OS then schedules the process to run on a CPU core.
- In a multi-core system, multiple processes can run **simultaneously**.

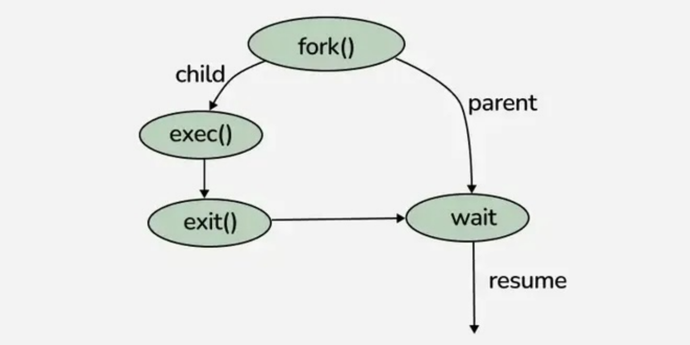


Figure: Simplified Process Creation

Introduction to `multiprocessing` in Python

- The `multiprocessing` module provides tools for creating and managing processes in Python.
- Key components:
  - `multiprocessing.Process`: Represents a process.
  - `process.start()`: Starts the process.
  - `process.join()`: Waits for the process to finish.

```
import multiprocessing
import time
def task(name):
    print(f"Process {name}: Starting")
    time.sleep(1) # Simulate some work
    print(f"Process {name}: Finishing")
if __name__ == "__main__":
    p = multiprocessing.Process(target=task,
args=("My Process",))
    p.start()
    p.join()
    print("Main process finished")
```

- `target`: The function that the process will execute.
- `args`: Arguments passed to the target function (must be a tuple, even for a single argument).
- The `if __name__ == "__main__":` block is crucial to prevent recursive process creation on Windows.

Creating Multiple Processes

```
def task(name):
    print("Process ",name," Starting")
    time.sleep(1) # doing something
    print("Process ",name," Finishing")
if __name__ == "__main__":
    processes = []
    for i in range(3):
        p = multiprocessing.Process(target=task, args=[i])
        processes.append(p)
        p.start()
    for p in processes:
        p.join()
    print("Main process finished")
```

- The `processes` list keeps track of the created processes so we can wait for them to finish using `join()`.
- Each process executes the `task` function independently.

Data Sharing Between Processes

- Processes have separate memory spaces. This means they cannot directly access each other's variables.
- To share data between processes, we need Inter-Process Communication (IPC) mechanisms.
- `multiprocessing.Queue` provides a simple way to exchange data between processes.
- **Using `multiprocessing.Queue`:**
  - Create a `Queue` object: `q = multiprocessing.Queue()`
  - Put data into the queue from a process: `q.put(data)`
  - Get data from the queue in another process (or the main process): `data = q.get()`

Process Communication with Queue

```python
import multiprocessing

def worker(q):
    q.put("Hello from process!")

if __name__ == "__main__":
    q = multiprocessing.Queue()
    p = multiprocessing.Process(target=worker, args=(q,))
    p.start()
    message = q.get()
    p.join()
    print("Message from process: ",message)
```

- The `worker` function puts a message into the queue.
- The main process retrieves the message from the queue using `q.get()`.
- `q.get()` is a **blocking** operation. The main process will wait until data is available in the queue.

```
import multiprocessing
def square(num, q):
    q.put(num * num)
if __name__ == "__main__":
    numbers = [1, 2, 3, 4, 5]
    q = multiprocessing.Queue()
    processes = []
    for num in numbers:
        p = multiprocessing.Process(target=square,
args=(num, q))
        processes.append(p)
        p.start()
    results = []
    for p in processes:
        results.append(q.get()) #retreive results
        p.join()
    print(results)
```

Practice session

Practice Session 1

- Creating Processes with `multiprocessing`
- Using queues in `multiprocessing`

`https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks/-/blob/main/PythonIntroNotebooks/Lecture_10.ipynb`

`multiprocessing.Pool`

- Managing processes individually (starting, joining, handling queues) can become cumbersome, especially for complex parallel tasks.
- `multiprocessing.Pool` provides a simpler interface for managing a pool of worker processes.
- It automatically distributes tasks across the available processes and collects the results.
- Key methods:
  - `pool.map(function, iterable)`: Applies a function to each item in an iterable, distributing the work across the process pool. Returns a list of results in the same order as the input iterable.
  - `pool.apply_async(function, args)`: Applies a function asynchronously. Returns an `AsyncResult` object, which can be used to retrieve the result later using `get()`.

Parallel Calculation with `Pool`

```
import multiprocessing
def square(num):
    return num * num
if __name__ == "__main__":
    pool=multiprocessing.Pool(processes=4)
    numbers = [1, 2, 3, 4, 5]
    results = pool.map(square, numbers)
    print(results)
    pool.close()
```

- The `pool=multiprocessing.Pool(processes=4)` statement creates a process pool.
- `pool.map()` returns results is a list with the results in the same order as the input.
- `pool.close()` closes the pool.

Parallel Calculation with `Pool`

```
import multiprocessing
def square(num):
    return num * num
if __name__ == "__main__":
    pool=multiprocessing.Pool(processes=4)
    numbers = [1, 2, 3, 4, 5]
    async_results = [pool.apply_async(square,
    (num,)) for num in numbers]
    async_results_list = [res.get() for res
    in async_results]
    print(async_results_list)
    pool.close()
```

- `pool=multiprocessing.Pool(processes=4)` creates a process pool.
- `pool.apply_async()` shows how to launch processes asynchronically and retrieve the results later.
- `pool.close()` closes the pool.

Practice session

Practice Session 2

- Parallel calculations with `Pool`

`https://gitlab2.informatik.uni-wuerzburg.de/ml4nets_notebooks/2024_wise_infhaf_notebooks/-/blob/main/PythonIntroNotebooks/Lecture_10.ipynb`

Summary

- **Parallel processing** uses multiple CPU cores to execute parts of a program simultaneously, leading to significant performance improvements for CPU-bound tasks.
- Processes are independent execution environments with their own memory spaces.
- The `multiprocessing` module in Python provides tools for creating and managing processes.
- `multiprocessing.Process` creates new processes.
- `process.start()` starts a process, and `process.join()` waits for it to finish.

Summary

- **Parallel processing** uses multiple CPU cores to execute parts of a program simultaneously, leading to significant performance improvements for CPU-bound tasks.
- Processes are independent execution environments with their own memory spaces.
- The `multiprocessing` module in Python provides tools for creating and managing processes.
- `multiprocessing.Process` creates new processes.
- `process.start()` starts a process, and `process.join()` waits for it to finish.
- `multiprocessing.Queue` is a simple way to share data between processes (Inter-Process Communication).

Summary

- **Parallel processing** uses multiple CPU cores to execute parts of a program simultaneously, leading to significant performance improvements for CPU-bound tasks.
- Processes are independent execution environments with their own memory spaces.
- The `multiprocessing` module in Python provides tools for creating and managing processes.
- `multiprocessing.Process` creates new processes.
- `process.start()` starts a process, and `process.join()` waits for it to finish.
- `multiprocessing.Pool` provides a higher-level interface for managing a pool of worker processes and simplifies common parallel tasks.
  - `pool.map(function, iterable)`: Applies a function to each item in parallel.
  - `pool.apply_async(function, args)`: Applies a function asynchronously.

Self-Study Questions

1. Explain the difference between parallel processing and sequential processing. Give an example of each.
2. What is a process? How does it differ from a regular Python program running in the interpreter?
3. Explain why processes do not share memory by default. What are the implications of this?
4. How can you pass data between processes in Python? Explain the use of `multiprocessing.Queue`.
5. Write a program that uses `multiprocessing` to calculate the sum of squares of a list of numbers in parallel.
6. What is the purpose of `process.join()`? What happens if you don't use it?
7. Explain the benefits of using `multiprocessing.Pool` compared to creating and managing individual processes manually.
8. Write a program that uses `multiprocessing.Pool` and `pool.map()` to process a list of files (e.g., counting the number of lines in each file).
9. Under what circumstances is parallel processing most beneficial?

Additional Resources

- **Python Documentation on `multiprocessing`:** `https://docs.python.org/3/library/multiprocessing.html` (The official documentation)
- **Real Python Tutorial on Parallel Processing:** `https://realpython.com/python-concurrency/`
- **Python Parallel Programming Cookbook by Giancarlo Zaccone** (A more advanced book with in-depth coverage of various parallel programming techniques in Python.)
- **Effective Python by Brett Slatkin (2nd Edition)**