UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Parallel Programming

Prof. Marco Bertini

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Python: multithreading

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# No-GIL threads and parallelism

# From GIL to Free-Threaded Python

- Classic CPython uses the Global Interpreter Lock (GIL):
  - Only one thread executes Python bytecode at a time.
  - We have I/O-bound multithreading, doesn't allow CPU-bound work.
- PEP 703 introduced an optional free-threaded build where the GIL is disabled.
  - Python 3.13: experimental free-threaded builds.
  - Python 3.14: free-threaded build is fully supported and faster (specializing interpreter re-enabled).
    - Use uv to install it

# Installing with uv

- Generic syntax:
  - `uv python install 3.13t` (3.13 free-threaded)
  - `uv python install 3.14+freethreaded`
- For a project:
  - `uv python install 3.14+freethreaded`
  - `uv venv -p 3.14+freethreaded`
  - `source .venv/bin/activate  # or .venv\Scripts\activate on Windows`

# Ensuring GIL is off

Even in free-threaded Python, some C extensions can re-enable the GIL for safety.

- Force the GIL off at runtime:
  - Environment variable:
    - `export PYTHON_GIL=0`
  - Or command-line switch:
    - `python -X gil=0 your_script.py`
  - Or running using uv:
    - `uv run -p 3.14+freethreaded python -X gil=0 your_script.py`

# CPU-bound vs. I/O-bound

- CPU-bound:
  - Runtime dominated by computations.
  - Faster CPU ⇒ faster program.
  - Examples: numerical kernels, cryptography, image processing.
- I/O-bound:
  - Runtime dominated by disk/network/DB access.
  - Faster disk/network ⇒ faster program.
- Before free-threaded Python:
  - Threads mainly useful for I/O-bound tasks (especially. Using Asyncio).
  - Multiprocessing required for CPU-bound parallelism.

# Threads vs Processes in a No-GIL World

- Threads (within one process):
  - Cheap to create; share memory.
  - Excellent for fine-grained parallelism in a free-threaded interpreter.
- Processes:
  - Heavyweight; separate address spaces, separate interpreters.
  - Good for isolation & multi-process scaling.
- With free-threaded CPython 3.14:
  - Threads can now give true CPU parallelism, similar to C/Java, while using the familiar `threading` and `concurrent.futures` APIs.

UNIVERSITÀ
DEGLI STUDI
FIRENZE

# Thread packages

# Three Building Blocks

- `threading`: manual threads and synchronization primitives

- `concurrent.futures`: `ThreadPoolExecutor` and `Future` objects

- `multiprocessing`: process-based pools for parallelism

# Cheat Sheet: Thread vs ThreadPool vs Pool

- `threading.Thread`:
  - Full control over threads and synchronization
  - Good for few long-lived threads and custom architectures
- `ThreadPoolExecutor`:
  - High-level API with futures, ideal for many similar tasks
- `multiprocessing.Pool`:
  - Process-based parallelism, good for isolation and classic CPython

# `threading` Module: Thread Lifecycle

- Create threads with `threading.Thread(target=..., args=...)`
- Start with `.start()`, wait with `.join()`
- Pattern: build list of threads, start all, join all
- In free-threaded Python, these threads can run CPU work in parallel

# `threading`: Shared Data and Locks

- Multiple threads may access shared mutable state
- Use `threading.Lock()` to protect critical sections
- `with lock:` ensures acquire()/release() with exception safety
- Without locks: data races and incorrect results in no-GIL Python

# `threading`: Other Useful Primitives

- `RLock`: re-entrant lock for recursive acquisitions
- `Semaphore`: allow limited number of concurrent threads
- `Event`: simple flag with wait()/set()/clear()
- Condition and Barrier: coordination between groups of threads

# Using Functions with Manual Threads

- Basic idea: each thread calls the same function with different args
- Use a small wrapper to store results in a shared list/dict
- Example pattern:
  - `def worker(name, start, end)`: compute and store result
  - `threading.Thread(target=worker, args=(...))`
- Good when logic naturally lives in a free function

# Example

```
import threading

def count_in_range(name: str, start: int,
   end: int) -> int:
    total = 0
    for x in range(start, end):
        total += x
    print(f"{name}: done")
    return total

def main():
    results = {}

    def worker(name: str, s: int, e: int):
        results[name] =
          count_in_range(name, s, e)

    threads: list[threading.Thread] = []

    for i in range(4):
        s = i * 1_000
        e = (i + 1) * 1_000
        t =
          threading.Thread(target=worker,
            args=(f"t{i}", s, e))
        threads.append(t)
        t.start()

    for t in threads:
        t.join()

    print("Total:", sum(results.values()))

if __name__ == "__main__":
    main()
```

# Example

```
import threading

def count_in_range(name: str, start: int,
   end: int) -> int:
    total = 0
    for x in range(start, end):
        total += x
    print(f"{name}: done")
    return total

def main():
    results = {}

    def worker(name: str, s: int, e: int):
        results[name] =
          count_in_range(name, s, e)

    threads: list[thre
```

```
for i in range(4):
    s = i * 1_000
    e = (i + 1) * 1_000
    t =
      threading.Thread(target=worker,
        args=(f"t{i}", s, e))
    threads.append(t)
    t.start()

for t in threads:
    t.join()

print("Total:", sum(results.values()))

if __name__ == "__main__":
    main()
```

Pattern highlights:
Each thread runs a function with different arguments.
Pure functions are easy to run in threads (few shared variables).
Use a wrapper function (worker) to store results in a shared dict/list.

# Using Classes with Manual Threads

- Encapsulate state + behavior in a class
- Each instance has its own configuration and result field
- Pass bound method as target: `threading.Thread(target=instance.run)`
- Easier to extend with logging, config, error handling

# Example

```
import threading

class RangeCounter:
    def __init__(self, name: str, start:
      int, end: int):
        self.name = name
        self.start = start
        self.end = end
        self.result = 0

    def run(self) -> None:
        total = 0
        for x in range(self.start,
          self.end):
            total += x
        self.result = total
        print(f"{self.name}: done")

def main():
    workers = [
        RangeCounter("t0", 0, 1_000),
        RangeCounter("t1", 1_000, 2_000),
    ]

    threads = [
        threading.Thread(target=w.run)
          # bound method as target
        for w in workers
    ]

    for t in threads:
        t.start()
    for t in threads:
        t.join()

    total = sum(w.result for w in
      workers)
    print("Total:", total)

if __name__ == "__main__":
    main()
```

# Example

```
import threading

class RangeCounter:
    def __init__(self, name: str, start:
      int, end: int):
        self.name = name
        self.start = start
        self.end = end
        self.result = 0

    def run(self) -> None:
        total = 0
        for x in range(self.start,
          self.end):
            total += x
        self.result = total
        print(f"{self.name}: done")

def main():
    workers = [
        RangeCounter("t0", 0, 1_000),
        RangeCounter("t1", 1_000, 2_000),
    ]

    threads = [
        threading.Thread(target=w.run)
          # bound method as target
        for w in workers
    ]

    for t in threads:
        t.start()
    for t in threads:
        t.join()

    total = sum(w.result for w in
      workers)
    print("Total:", total)
```

Key ideas:
You want to encapsulate state + behavior in a class.
Use bound methods (e.g. w.run) as target for Thread.
State (start, end, result) lives inside each instance ⇒ less shared data.

Easy to add more behavior (logging, error handling, configuration).

# High-Level Threading: `concurrent.futures`

- `ThreadPoolExecutor`: manages a pool of worker threads
- `submit(fn, *args)`: schedule work, returns a `Future`
- `map(fn, iterable)`: parallel map-style execution
- `Futures` provide a unified interface for waiting on results

# Example

```
from concurrent.futures import
ThreadPoolExecutor
from primes import count_primes


def task(args: tuple[int, int])
  -> int:
    start, end = args
    return count_primes(start, end)


def main():
    ranges = [(1, 75_000), (75_000,
      150_000), (150_000, 225_000),
      (225_000, 300_000)]
    with
ThreadPoolExecutor(max_workers=4)
as ex:
        results = list(ex.map(task,
          ranges))
    print("Total primes:",
      sum(results))


if __name__ == "__main__":
    main()
```

Key idea: “apply this function to many inputs”.
Prepare a list of independent tasks.
Use `executor.map(fn, iterable)` for a clean “parallel map”.
The pool automatically:
- creates worker threads,
- schedules tasks,
- reuses threads between tasks.

# Futures: What They Represent

- `Future` = handle to a not-yet-computed result
- `fut.result()`: return value or raise captured exception
- `fut.done()`, `fut.exception()`, `fut.add_done_callback(cb)`
- Use `as_completed(futures)` to process tasks as they finish

# Thread Pools: Map-Style Parallelism

- ThreadPoolExecutor is ideal for 'apply this function to many inputs'
- Prepare a list of independent tasks (e.g. ranges)
- Use `executor.map(fn, iterable)` for simple parallel map
- Pool handles thread creation, scheduling, and reuse

# Thread Pools: Futures & `as_completed`

- More control with `submit` + `as_completed`
- Submit tasks:
  `futures = [executor.submit(fn, arg) for arg in items]`
- Iterate `as_completed(futures)` to process tasks as they finish
- Handle per-task exceptions via `fut.result()` inside `try/except`

# Example

```
from concurrent.futures import
ThreadPoolExecutor, as_completed
from primes import count_primes


def main():
    ranges = [(1, 75_000), (75_000,
      150_000), (150_000, 225_000),
      (225_000, 300_000)]

    with
ThreadPoolExecutor(max_workers=4) as ex:
        futures = {
            ex.submit(count_primes,
              start, end): (start, end)
            for (start, end) in ranges
        }

        total = 0

        for fut in
          as_completed(futures):
            start, end = futures[fut]
            try:
                n_primes = fut.result()
                print(f"{start}-{end}:
                  {n_primes}")
                total += n_primes
            except Exception as e:
                print(f"Error in range
                  {start}-{end}: {e}")

        print("Total primes:", total)

if __name__ == "__main__":
    main()
```

# Example

```
from concurrent.futures import
ThreadPoolExecutor, as_completed
from primes import count_primes


def main():
    ranges = [(1, 75_000), (75_000,
      150_000), (150_000, 225_000),
      (225_000, 300_000)]

    with
ThreadPoolExecutor(max_workers=4) as ex:
        futures = {
            ex.submit(count_primes,
              start, end): (start, end)
            for (start, end) in ranges
        }

        total = 0

        for fut in
          as_completed(futures):
            start, end = futures[fut]
            try:
                n_primes = fut.result()
                print(f"{start}-{end}:
                  {n_primes}")
                total += n_primes
            except Exception as e:
                print(f"Error in range
                  {start}-{end}: {e}")

        print("Total primes:", total)


if __name__ == "__main__":
```

For more control when using TreadPool, use `submit` + `as_completed`
Benefits:
Process results as soon as each task finishes.
Handle exceptions per-task (inside try: fut.result()).
Same pattern works with ProcessPoolExecutor if you want processes instead of threads.

# Process Pools vs Thread Pools (Usage Focus)

- ThreadPoolExecutor:
  - Shared memory, low overhead, great on free-threaded Python
  - Use when tasks share in-memory data or objects
  - Use for CPU-bound pure Python when libraries are thread-safe
  - Very low overhead to send arguments/results (no pickling).
- multiprocessing.Pool / ProcessPoolExecutor:
  - Separate processes, pickling overhead: arguments/results are pickled thus must be serializable.
  - Use for isolation or on classic CPython for CPU-bound work

# Process-Based Parallelism: `multiprocessing.Pool`

- Uses multiple worker processes rather than threads
- Each process has its own interpreter and memory space
- `map` / `starmap` run functions across workers
- Adds overhead (startup + pickling) but bypasses GIL on classic CPython

# Expected Performance

- Classic CPython (with GIL):
  - Sequential: baseline
  - Threads / ThreadPool: ~same speed as sequential for CPU-bound
  - Multiprocessing: speed-up at the cost of overhead
- Free-threaded Python 3.14:
  - Threads and thread pools: real multi-core speed-up
  - Multiprocessing: still parallel, sometimes similar or slower than threads

# When Threads Beat Multiprocessing

- Lower overhead: no process startup or pickling
- Shared memory: easy to share large read-only data
- Lower memory footprint: single interpreter, many threads
- Great default for CPU-bound pure Python on free-threaded CPython

# When Multiprocessing Still Wins

- Process isolation: worker crash doesn’t kill main program
- Works on classic CPython where GIL blocks CPU-bound threads
- Useful for libraries not yet compatible with free-threaded Python
- Natural building block for multi-machine / distributed setups

# Race Conditions & Synchronization (Recap)

- Removing the GIL exposes real data races on shared state
  - e.g.: unprotected updates (eg. `counter += 1`) can lose increments
- Locks and other primitives are required for correctness
- Free-threaded Python behaves like C/Java with respect to races

# Measuring Performance Correctly

- Use `time.perf_counter()` for wall-clock timing
- Run each scenario multiple times, consider median
- Avoid mixing heavy I/O with CPU-bound work in benchmarks
- For deeper analysis: timeit, cProfile, line profilers, IDE tools

# Summary & Takeaways

- Python 3.14 free-threaded builds enable real CPU-bound multithreading
- `threading` + `concurrent.futures` give powerful, familiar APIs
- `multiprocessing` remains valuable for isolation and legacy setups
- Same code can run on classic and free-threaded Python, with different speedups

# Books

- Python Parallel Programming Cookbook, G. Zaccone, Packt Publishing, Chapt. 2

- The Python 3 Standard Library by Example, D. Hellmann, Addison Wesley, Chapt. 10