Data Science and Advanced Programming — Lecture 13

High-Performance Computing with Python

Simon Scheidegger
Department of Economics, University of Lausanne, Switzerland

December 8th, 2025 | 12:30 - 16:00 | Internef 263

Lecture Overview

Schedule:

1. Why Parallelism? (45 min)
2. Threading & I/O-Bound (50 min)

   — *Break (15 min)* —
3. Multiprocessing & CPU-Bound (55 min)
4. Finance Applications (45 min)

   — *Break (10 min)* —
5. Projects & Best Practices (40 min)

Format:

- Slides introduce concepts
- Switch to notebooks for practice
- Back to slides for next concept
- Repeat!

What You'll Learn:

- Speed up your Python code
- Threading vs. Multiprocessing
- Real finance applications

Topic 1

Why Parallelism Matters

Motivation & Core Concepts

`01_motivation.ipynb`

The Problem: Your Code is Too Slow

Scenario: You need to price 10,000 exotic options using Monte Carlo simulation

- Each option requires 100,000 simulation paths
- Each simulation takes 0.1 seconds
- Total time: $10,000 \times 0.1 = 1,000$ seconds $\approx$ **17 minutes**

The Reality

Your laptop has 8 CPU cores, but Python is only using **one** of them!

The Goal

Use all cores $\rightarrow$ Reduce time to $\approx$ **2 minutes**

The End of Free Speed

Moore's Law (1965-2005):

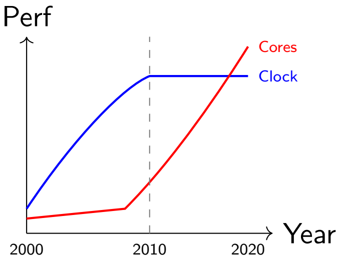
- Transistor count doubles every 2 years
- Clock speed kept increasing
- Your code got faster *automatically*

The Wall (∼2005):

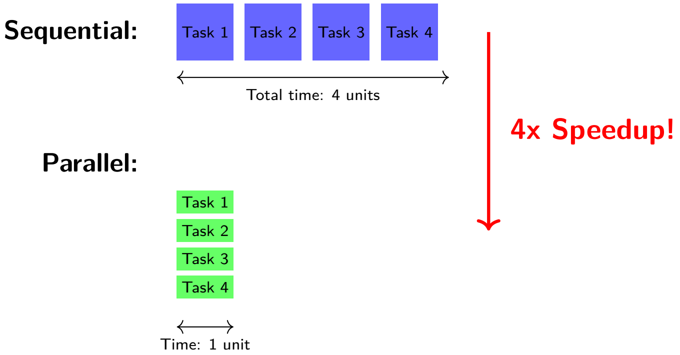
- Power consumption $\propto$ frequency^3
- Clock speeds plateaued at ∼4 GHz

The Solution:

- More cores, not faster cores
- Your laptop: 4–16 cores



Sequential vs. Parallel Execution



Key Insight

If tasks are **independent**, we can run them simultaneously on different cores.

Hands-On Time!

Experience the Speedup

`01_motivation.ipynb`

Sections 1-2: Sequential vs. Parallel Option Pricing

Amdahl's Law: The Limits of Parallelism

Not everything can be parallelized:

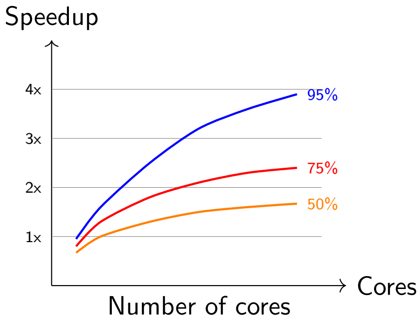
- Reading input data
- Setting up calculations
- Combining final results

Amdahl's Law:

$$\text{Speedup} = \frac{1}{(1-p) + \frac{p}{n}}$$

Where:

- p = parallelizable fraction
- n = number of cores



Takeaway

Even with infinite cores, 50% parallel code gives max 2x speedup!

Hands-On Time!

Explore Amdahl's Law

`01_motivation.ipynb`

Sections 3-4: Scaling experiments and visualization

Why Finance Needs Parallelism

CPU-Intensive Tasks:

- Monte Carlo simulations
- Option pricing (exotic derivatives)
- Portfolio optimization
- Risk calculations (VaR, CVaR)
- Backtesting strategies
- Bootstrap inference

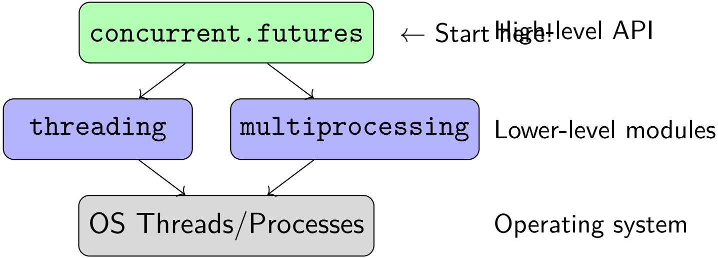
I/O-Intensive Tasks:

- Fetching market data
- Reading multiple data files
- API calls to data providers
- Database queries
- Downloading reports

The Two Types of Waiting

- **CPU-bound:** Waiting for calculations to finish → **Multiprocessing**
- **I/O-bound:** Waiting for data to arrive → **Threading**

Python's Parallel Toolkit



Our Focus: `concurrent.futures`

Clean, simple API that works for both threading and multiprocessing.

First Look: Sequential vs. Parallel

Sequential (what you're used to):

```python
results = []
for item in data:
    result = slow_function(item)
    results.append(result)

```

Parallel (what we'll learn):

```python
from concurrent.futures import ProcessPoolExecutor

with ProcessPoolExecutor() as executor:
    results = list(executor.map(slow_function, data))

```

That's It!

Two extra lines of code can give you 4-8x speedup on multi-core machines.

Hands-On Time!

Complete Topic 1 Exercises

`01_motivation.ipynb`

Section 5: Exercises (put options, volatility grid)

Topic 2

Threading & I/O-Bound Tasks

When Waiting is the Bottleneck

`02_threading_io_bound.ipynb`

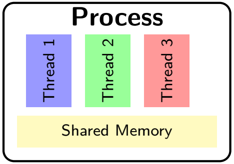
What is a Thread?

Process:

- Independent program, own memory
- Heavy to create, true parallelism

Thread:

- Lives inside a process, shares memory
- Lightweight, concurrent (not parallel*)



***The Python Catch**

Due to the GIL, Python threads don't run truly in parallel for CPU work.

The Global Interpreter Lock (GIL)

What is the GIL?

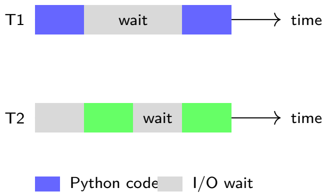
- A mutex in CPython
- Only one thread executes Python bytecode at a time
- Protects memory management

What this means:

- CPU-bound code: threads don't help
- I/O-bound code: threads help a lot!

Why I/O works:

- GIL is released during I/O
- While one thread waits for data...
- ...another can do work!



Key Insight

Threads take turns using the CPU while others wait for I/O.

I/O-Bound vs. CPU-Bound

Characteristic	**I/O-Bound**	**CPU-Bound**
Bottleneck	Waiting for data	Calculations
CPU usage	Low (lots of idle)	High (near 100%)
Solution	Threading	Multiprocessing
Finance Examples		
	Fetching stock prices	Monte Carlo simulation
	Reading CSV files	Portfolio optimization
	API calls	VaR calculations
	Database queries	Option pricing

How to Tell?

Run your code and check CPU usage. If it's low while code runs slowly → I/O-bound.

Hands-On Time!

I/O-Bound Demo

`02_threading_io_bound.ipynb`

Section 1: See threading in action with simulated data fetching

ThreadPoolExecutor: The Simple Way

```python
from concurrent.futures import ThreadPoolExecutor
import time

def fetch_stock_data(ticker):
    """Simulate fetching data (I/O operation)"""
    time.sleep(0.5)  # Simulate network delay
    return {"ticker": ticker, "price": 100.0}

tickers = ["AAPL", "GOOGL", "MSFT", "AMZN"]

# Parallel fetching
with ThreadPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(fetch_stock_data, tickers))

```

Sequential: $4 \times 0.5s = 2.0s$

Parallel: $\approx 0.5s$ (4x faster!)

Pattern 1: executor.map()

Use when: Same function, many inputs, order matters

```python
from concurrent.futures import ThreadPoolExecutor

def process(item):
    return item * 2

items = [1, 2, 3, 4, 5]

with ThreadPoolExecutor(max_workers=4) as executor:
    results = list(executor.map(process, items))  # [2, 4, 6, 8, 10]
```

Key Properties

Results maintain input order • Simple syntax • Good for homogeneous tasks

Pattern 2: executor.submit() + as_completed()

Use when: Want results as they finish (not in order)

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

tickers = ["AAPL", "GOOGL", "MSFT", "AMZN"]

with ThreadPoolExecutor() as executor:
    # Submit tasks
    futures = {executor.submit(fetch_data, t): t
               for t in tickers}

    # Process results as they complete
    for future in as_completed(futures):
        ticker = futures[future]
        result = future.result()
        print(f"{ticker}: got data!")

```

When to use this pattern

Progress feedback, early termination, heterogeneous task times

Hands-On Time!

Practice Both Patterns

`02_threading_io_bound.ipynb`

Section 2-3: executor.map() vs submit()+as_completed()

Handling Exceptions in Threads

```python
from concurrent.futures import ThreadPoolExecutor, as_completed

def risky_fetch(ticker):
    if ticker == "BAD":
        raise ValueError(f"Invalid ticker: {ticker}")
    return {"ticker": ticker, "price": 100.0}

tickers = ["AAPL", "BAD", "MSFT"]

with ThreadPoolExecutor() as executor:
    futures = {executor.submit(risky_fetch, t): t for t in tickers}

    for future in as_completed(futures):
        ticker = futures[future]
        try:
            result = future.result()
            print(f"{ticker}: {result}")
        except Exception as e:
            print(f"{ticker}: ERROR - {e}")
```

Threading: Key Takeaways

When to Use Threading

- Fetching data from multiple sources
- Reading/writing multiple files
- Any task where you're waiting for external resources

When NOT to Use Threading

- Heavy computations (Monte Carlo, optimization)
- Number crunching $\rightarrow$ Use multiprocessing instead!

Best Practices

- Use `ThreadPoolExecutor` (not raw threads)
- Always handle exceptions
- Use context managers (`with` statement)

Hands-On Time!

Complete Threading Exercises

`02_threading_io_bound.ipynb`

Section 4-5: File processing and exercises

Break Time

15 minutes

Next up: Multiprocessing for CPU-bound tasks

Topic 3

Multiprocessing & CPU-Bound Tasks

True Parallelism for Heavy Computation

`03_multiprocessing_cpu_bound.ipynb`

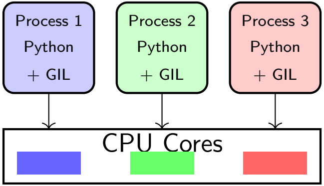
Why Multiprocessing?

The GIL Problem:

- Threads share one GIL
- Only one runs Python at a time
- CPU-bound code doesn't speed up

The Solution:

- Use separate processes
- Each process has its own GIL
- Each process has its own Python interpreter
- True parallel execution!



Result

3 processes = 3 cores working simultaneously

Threads vs. Processes: Trade-offs

Aspect	Threads	Processes
Memory	Shared	Separate (copied)
Creation	Fast	Slower
Communication	Easy (shared vars)	Harder (serialization)
GIL	Blocked	Bypassed
Best for	I/O-bound	CPU-bound
`concurrent.futures`	`ThreadPoolExecutor`	`ProcessPoolExecutor`

Process Overhead

Creating processes is slower and uses more memory. The task must be substantial enough to overcome this overhead.

Hands-On Time!

Threading vs. Multiprocessing

`03_multiprocessing_cpu_bound.ipynb`

Section 1: See why threads fail for CPU-bound tasks

ProcessPoolExecutor: Same API, True Parallelism

```python
from concurrent.futures import ProcessPoolExecutor
import numpy as np

def monte_carlo_pi(n_samples):
    x = np.random.random(n_samples)
    y = np.random.random(n_samples)
    inside = np.sum(x**2 + y**2 <= 1)
    return 4 * inside / n_samples

# Split work across 4 processes
with ProcessPoolExecutor(max_workers=4) as executor:
    estimates = list(executor.map(monte_carlo_pi, [1_000_000] * 4))

pi_estimate = np.mean(estimates)
```

Finance Example: Monte Carlo Option Pricing

```python
import numpy as np
from concurrent.futures import ProcessPoolExecutor

def price_european_call(args):
    S0, K, T, r, sigma, n_paths = args
    Z = np.random.standard_normal(n_paths)
    ST = S0 * np.exp((r - 0.5*sigma**2)*T + sigma*np.sqrt(T)*Z)
    payoffs = np.maximum(ST - K, 0)
    return np.exp(-r * T) * np.mean(payoffs)

params = (100, 100, 1.0, 0.05, 0.2, 250_000)  # S0,K,T,r,sigma,paths

with ProcessPoolExecutor(max_workers=4) as executor:
    prices = list(executor.map(price_european_call, [params]*8))
option_price = np.mean(prices)
```

Hands-On Time!

Monte Carlo in Parallel

`03_multiprocessing_cpu_bound.ipynb`

Section 2: Option pricing with multiprocessing

The Pickling Requirement

Processes have separate memory → data must be serialized (pickled)

What CAN'T be pickled

Lambda functions, nested functions, file handles, DB connections

```python
# This will FAIL
with ProcessPoolExecutor() as executor:
    results = executor.map(lambda x: x**2, [1,2,3])  # Error!

# This WORKS
def square(x):
    return x ** 2

with ProcessPoolExecutor() as executor:
    results = executor.map(square, [1,2,3])  # OK!

```

When Parallelization Hurts: The Overhead Trap

```python
# BAD: Task is too small
def add_one(x):
    return x + 1

# Overhead of creating processes >> computation time
with ProcessPoolExecutor() as executor:
    results = list(executor.map(add_one, range(100)))
# This is SLOWER than sequential!
```

Rule of Thumb

Each task should take at least **10-100ms** to justify process overhead.

Solution: Chunk your work into larger batches.

Hands-On Time!

Chunking and Overhead

`03_multiprocessing_cpu_bound.ipynb`

Section 3: Learn when parallelization helps vs. hurts

Finance Application: Portfolio VaR

Value at Risk (VaR): Maximum expected loss at a confidence level

Monte Carlo VaR requires:

1. Simulate many portfolio return scenarios
2. Sort returns
3. Find the percentile cutoff

Parallelization strategy:

- Split simulations across processes
- Each process generates subset of scenarios
- Combine and calculate VaR at the end

Typical Speedup

4-core machine: 3-4x faster
8-core machine: 6-7x faster

Hands-On Time!

VaR Calculation

`03_multiprocessing_cpu_bound.ipynb`

Section 4: Parallel Value-at-Risk

Multiprocessing: Key Takeaways

When to Use

- Monte Carlo simulations
- Parameter grid searches
- Backtesting strategies
- Heavy numerical computation

Watch Out For

- Overhead for small tasks
- Memory usage (data copied)
- No lambdas (pickling)

Best Practices

Use `ProcessPoolExecutor` • Chunk small tasks • Profile before/after

Topic 4

Real-World Finance Applications

Putting It All Together

`04_finance_applications.ipynb`

Application 1: Parallel Backtesting

The Problem:

- Test a trading strategy with different parameters
- 100 parameter combinations $\times$ 10 years of data
- Sequential: hours of waiting

The Solution:

- Each parameter combination is independent
- Perfect for `ProcessPoolExecutor`
- Distribute across all CPU cores

Example Strategy

Moving average crossover: test all combinations of short (5-50 days) and long (20-200 days) windows

Hands-On Time!

Parallel Backtesting

`04_finance_applications.ipynb`

Section 1: Build a parallel strategy backtester

Application 2: Bootstrap Confidence Intervals

The Problem:

- Estimate uncertainty in Sharpe ratio
- Need 10,000+ bootstrap samples
- Each sample: resample data, calculate statistic

Why it's parallel-friendly:

- Each bootstrap sample is independent
- CPU-bound (resampling + calculations)
- Easy to split: 10,000 samples $\rightarrow$ 2,500 per core

Statistical Rigor

Bootstrap gives you confidence intervals without assuming normality — essential for fat-tailed financial returns.

Hands-On Time!

Bootstrap Analysis

`04_finance_applications.ipynb`

Section 2: Parallel bootstrap for Sharpe ratio CI

Application 3: Correlation Matrix Computation

The Problem:

- 500 assets $\rightarrow$ 124,750 pairwise correlations
- Need rolling correlations over time
- Sequential calculation is slow

Parallelization Approach:

- Split asset pairs across processes
- Or: parallelize across time windows
- Combine results at the end

Note

NumPy already parallelizes some operations internally. Profile first!

Hands-On Time!

Complete Finance Applications

`04_finance_applications.ipynb`

Section 3: Correlation analysis and wrap-up

Break Time

10 minutes

Final session: Best practices and projects

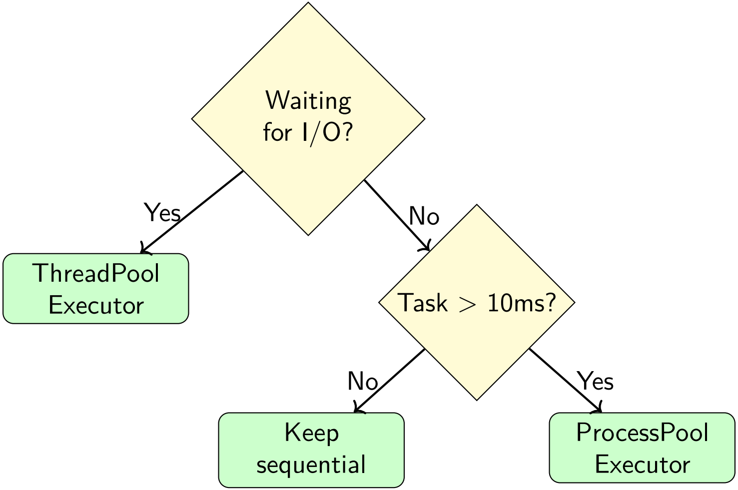
Topic 5

Best Practices & Projects

Writing Robust Parallel Code

`05_project_exercises.ipynb`

Decision Tree: Thread or Process?



Common Pitfall 1: Shared State

The Problem

Multiple processes modifying the same variable causes unexpected behavior.

```python
# WRONG - This doesn't work as expected!
counter = 0

def increment():
    global counter
    counter += 1  # Each process has its OWN copy!

with ProcessPoolExecutor() as executor:
    executor.map(increment, range(100))

print(counter)  # Still 0!

```

Solution

Return values instead of modifying global state. Let the main process aggregate.

Common Pitfall 2: Too Many Workers

```python
# WRONG - More workers than cores
with ProcessPoolExecutor(max_workers=100) as executor:
    results = executor.map(cpu_task, data)
# Context switching overhead kills performance!

# RIGHT - Match workers to cores
import os
n_cores = os.cpu_count()
with ProcessPoolExecutor(max_workers=n_cores) as executor:
    results = executor.map(cpu_task, data)
```

Guidelines

- CPU-bound: workers $\leq$ number of cores
- I/O-bound: workers can exceed cores (2-4x)
- Memory-heavy: reduce workers to avoid swapping

Progress Bars with tqdm

```python
from concurrent.futures import ProcessPoolExecutor, as_completed
from tqdm import tqdm

def slow_task(x):
    # ... some computation
    return x ** 2

items = range(100)

with ProcessPoolExecutor() as executor:
    futures = [executor.submit(slow_task, x) for x in items]

    results = []
    for future in tqdm(as_completed(futures), total=len(items)):
        results.append(future.result())

```

Output

```
100%|============| 100/100 [00:05<00:00, 18.32it/s]
```

Beyond concurrent.futures

When you need more power:

joblib

- Simple API
- Memory mapping
- Good for NumPy
- scikit-learn uses it

Dask

- Parallel DataFrames
- Larger-than-RAM data
- Lazy evaluation
- Scales to clusters

Ray

- Distributed computing
- Actor model
- ML focused
- Production ready

Start Simple

`concurrent.futures` handles 90% of use cases. Only reach for specialized tools when you hit its limits.

Debugging Parallel Code

Parallel bugs are hard to find:

- Non-deterministic behavior
- Errors in worker processes
- Hard to reproduce

Strategies:

1. **Start sequential:** Make sure code works with `max_workers=1`
2. **Catch exceptions:** Always wrap `future.result()` in try/except
3. **Test with small data:** Faster iteration, easier to spot issues
4. **Use logging:** Print statements get mixed up

Golden Rule

If it works with `max_workers=1`, it should work with more. If not, you have a parallelism bug.

Hands-On Time!

Mini-Project Time!

`05_project_exercises.ipynb`

Choose your project and apply what you've learned

Summary: Your Parallel Python Toolkit

concurrent.futures

ThreadPoolExecutor

- I/O-bound tasks
- API calls
- File operations
- Network requests

ProcessPoolExecutor

- CPU-bound tasks
- Monte Carlo
- Optimization
- Backtesting

Key Message

You now have the tools to make your finance code run 4-8x faster. Use them wisely!

Thank You!

Questions?

Key takeaways:

- I/O-bound → `ThreadPoolExecutor`
- CPU-bound → `ProcessPoolExecutor`
- Always profile before and after
- Chunk small tasks to reduce overhead

Happy parallel programming!