High Performance Computing

Lecturer: Bo Li TA: Chen Gao

2026-01-28

Table of contents

0.1 Some Resources

Specifications

- MPI Standard/API Specification
- OpenMP Standard/API Specification

Tutorials

- MPI Tutorial – LLNL
- OpenMP Tutorial – LLNL

Books

- *Introduction to High Performance Computing for Scientists and Engineers*
  Georg Hager, Gerhard Wellein
- *Using Advanced MPI: Modern Features of the Message-Passing Interface*
  (Scientific and Engineering Computation) MIT Press, 2014
  Gropp, W.; Höfler, T.; Thakur, R.; Lusk, E.

0.2 Why Parallel Computing?

High-performance computing (HPC) enables us to:

- **Handle huge data sets**
  - Data management in memory
  - Data management on disk
- **Tackle complex problems**
  - Time-consuming algorithms
  - Data mining
  - Visualization
- **Leverage specialized solutions** for:
  - Processors
  - Networks
  - Storage

0.3 More Reasons for Parallel Computing

- **Transmission speeds:**
  The speed of a serial computer is limited by how fast data can move through hardware. Fundamental limits include:
  - The speed of light (30 cm/ns)
  - The transition limit of copper wire (9 cm/ns) To achieve higher speeds, processing elements must be placed closer together.
- **Limits of miniaturization:**
  While processor technology can place more transistors on a chip, even atomic-scale components will eventually reach their limit on miniaturization.
- **Economic limits:**
  Making a single processor faster becomes increasingly expensive. Using many commodity processors in parallel is often more cost-effective and can provide better performance.

0.4 A way out

- Current computer architectures are increasingly relying upon **hardware level parallelism**.
- Multiple execution units.
- Multi-core.

0.5 From Programming Language to Hardware

- A computer is a "stupid" device; it only understands "on" and "off".
  - These states are represented by 0 and 1 (binary).
- Early programmers communicated directly in 0s and 1s.
- Later, programs were developed to translate from symbolic notation to binary.
  - The first of these was called **assembly language**.
- **Advanced programming languages** improve upon assembly:
  - Allow the programmer to think in a more natural (human-readable) language.
  - Increase the productivity of software development.
  - Improve portability of programs across different hardware platforms.

0.6 Basics: von Neumann Architecture [1]

Virtually all computers have followed this basic design, which is comprised of four main components:

1. **Memory**
   - Read/write, random access memory is used to store both program instructions and data.
   - *Program instructions* are coded data which tell the computer to do something.
   - *Data* is simply information to be used by the program.
2. **Control Unit**
   - Fetches instructions and data from memory
   - Decodes the instructions
   - Sequentially coordinates operations to accomplish the programmed task
3. **Arithmetic Logic Unit (ALU)**
   - Performs basic arithmetic and logical operations
4. **Input/Output (I/O)**
   - Provides the interface between the computer and the human operator (or other systems)

[1] Adapted from LLNL HPC Introduction to Parallel Computing Tutorial

0.7 The End of the "Free Lunch"

For a long time, speeding up computations was considered a "free lunch":

- **Transistor density** increased, making integrated circuits smaller and more efficient.
- **Clock speeds** steadily rose, increasing the number of operations per second (from MHz to GHz).

However, this free lunch has ended in recent years:

- We are now reaching the **limitations of transistor density**.
- **Increasing clock frequency** further requires too much power.

We used to focus only on *floating point operations per second.* Now, we must also consider *floating point operations per Watt.*

1 Parallel Computing

1.1 What is parallel computing?

1.1.1 Serial Computing:

- A problem is broken into a discrete series of instructions.
- Instructions are executed serially one after another.
- Executed on a single processor.
- Only one instruction may execute at any moment in time.

1.1.2 Parallel Computing:

- A problem is broken into a discrete series of instructions that can be solved concurrently.
- Each part is further broken down to a series of instructions.
- Instructions from each part execute simultaneously on different processors.
- An overall control/coordination mechanism is employed.

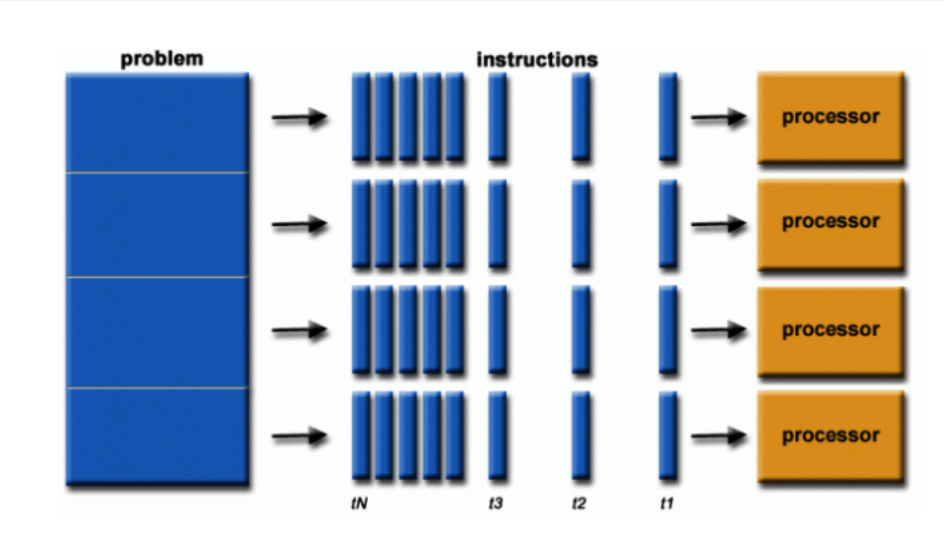

Figure 1: Serial and Parallel Computing

1.2 Speedup, Efficiency & Amdahl's Law

- $T(p, N)$: Time to solve a problem of total size N on p processors.
- **Parallel Speedup:** Let $S(p, N) = \frac{T(1,N)}{T(p,N)}$ be the parallel speedup.
- **Parallel Efficiency:** Let $E(p, N) = \frac{S(p,N)}{p}$ be the parallel efficiency.

. . .

Theorem 1.1. *(Amdahl's Law)*

$$T(p, N) = f \cdot T(1, N) + (1 - f)\frac{T(1, N)}{p}$$

*where: f is the fraction of the computation that is **sequential** (cannot be parallelized), $(1 - f)$ is the fraction that can be parallelized.*

- **Resulting speedup:** $S(p, N) = \frac{1}{f + \frac{1-f}{p}}$
- **Limitation:** As $p \to \infty$, the maximum speedup approaches: $S(p, N) < \frac{1}{f}$

  That is, the speedup is limited by the sequential portion of the code.

1.3 Weak versus strong scaling

Strong scaling: Is defined as how the solution time varies with the number of processors for a fixed total problem size.

Weak scaling: Is defined as how the solution time varies with the number of processors for a fixed problem size per processor.

2 Why Parallelism Matters

2.1 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 path takes **0.1 seconds**
- **Total time:**
  $10{,}000 \times 0.1 = 1{,}000$ seconds $\approx$ **17 minutes**

The Reality:
Your laptop has **14 CPU cores**, but Python is only using **one** of them!

The Goal:
Use all cores $\longrightarrow$ **Reduce time to $\approx$ 1.5 minutes**

2.2 Sequential vs. Parallel Execution

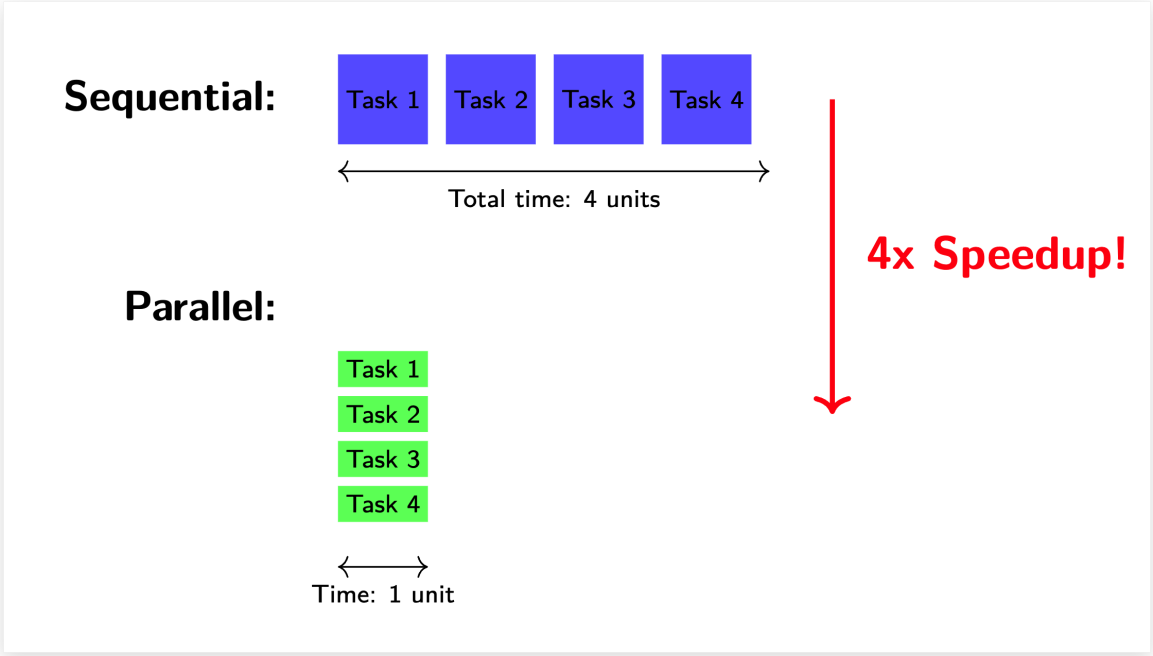

Figure 2: Sequential and Parallel Execution

> **Tip**
>
> **Key Insight:** If tasks are independent, we can run them simultaneously on different cores.

2.3 Why Quantitative Macro Needs Parallelism

- **CPU-Intensive Tasks (Model Solving & Estimation)**
  - **Global Solution Methods:** Value Function Iteration (VFI) and projection methods over large state spaces.
  - **Bayesian Estimation:** Running multiple Markov Chain Monte Carlo (MCMC) chains for parameter estimation.
  - **Simulating Method of Moments (SMM):** Repeatedly solving models to match empirical moments.
- **I/O-Intensive Tasks (Data Pipeline)**

  - **API Data Aggregation:** Fetching thousands of series simultaneously from FRED, World Bank, or IMF APIs.
  - **Large Panel Processing:** Reading census data or administrative tax records from disk.

- **The Two Types of Waiting**
  - **CPU-bound:** Waiting for the model to converge or the grid search to finish ⟶ *Multiprocessing*
  - **I/O-bound:** Waiting for the web scraper or API to return data ⟶ *Threading* or *Async I/O*
- **Python's Parallel Toolkit**

. . .

Layer	Modules/Concepts	Description
High-level	`concurrent.futures`	**Start here!** Clean, simple API for both threading & multiprocessing
Mid-level	`threading`, `multiprocessing`	Lower-level modules to manage OS threads/processes directly
Low-level	OS Threads/Processes	Managed by the operating system

Our Focus:
We'll use `concurrent.futures` — a clean, high-level API that supports both threading and multiprocessing.

2.4 First Look: Sequential vs. Parallel

Sequential (what you're used to):

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

Parallel (what we'll learn):

```
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 **4–8× speedup** on multi-core machines.

3 Threading & I/O-Bound Tasks

3.1 What is a Thread?

- **Process:**
  - An independent program, with its own memory space
  - Heavyweight to create
  - Supports true parallelism
- **Thread:**
  - Lives inside a process, sharing that process's memory
  - Lightweight to create and manage
  - Enables concurrency (but **not** true parallelism in Python*)

. . .

> **Note**
>
> **The Python Catch:** Due to the Global Interpreter Lock (GIL), Python threads **do not** execute bytecode in true parallel for CPU-bound work.

3.2 The Global Interpreter Lock (GIL)

What is the GIL?

- A *mutex* (mutual exclusion lock) implemented in CPython.
- Only **one thread** can execute Python **bytecode** at a time.
- Protects memory management within the Python interpreter.

Implications:

- **CPU-bound code:** Threads *do not* speed up execution (because of GIL).
- **I/O-bound code:** Threads *can* help significantly.

3.3 Why Do Threads Help with I/O-Bound Code?

- The **GIL is released during I/O operations** (like reading files, network calls, waiting for input/output).
- While *one thread* waits for data (e.g., disk or network), *another thread* can use the CPU to continue processing Python code.

. . .

```
# Timeline illustration:
# Time →
# T1:  |---Python code---|.......I/O wait.......|
# T2:                    |---Python code---| ... I/O wait ... |
```

> **Note**
>
> **Key Insight:** Threads take turns using the CPU while others are waiting for I/O, enabling efficient concurrency for I/O-bound workloads.

3.4 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

> **Tip**
>
> **How to Tell?**
> Run your code and check CPU usage. If it's low while code runs slowly → you have an **I/O-bound** problem.

3.5 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"]
```

```python
%%time
# Parallel fetching
with ThreadPoolExecutor() as executor:
    results = list(executor.map(fetch_stock_data, tickers))
```

```
CPU times: user 635 µs, sys: 705 µs, total: 1.34 ms
Wall time: 506 ms
```

```python
%%time
# Sequential fetching
results = [fetch_stock_data(ticker) for ticker in tickers]
```

```
CPU times: user 252 µs, sys: 231 µs, total: 483 µs
Wall time: 2.01 s
```

> **Tip**
>
> **Sequential:** 4 × 0.5s = 2.0s
> **Parallel:** ≈ 0.5s (4x faster!)

3.6 Pattern 1: `executor.map()`

Use when:

- You have the same function to run
- Many inputs
- The order of results matters

. . .

```python
def process(item):
    return item * 2


items = [1, 2, 3, 4, 5]
with ThreadPoolExecutor() as executor:
    results = list(executor.map(process, items))
print(results)
```

```
[2, 4, 6, 8, 10]
```

Key Properties:

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

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

Use when:

- You want results as soon as they finish (not in input order)
- You want progress feedback or early termination
- Tasks have heterogeneous/variable completion times

. . .

```python
from concurrent.futures import as_completed

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

with ThreadPoolExecutor() as executor:
    # Submit tasks
    futures = {executor.submit(fetch_stock_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!")
```

```
AAPL: got data!
GOOGL: got data!
MSFT: got data!
AMZN: got data!
```

3.8 Handling Exceptions in Threads

```python
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}")
```

```
AAPL: {'ticker': 'AAPL', 'price': 100.0}
BAD: ERROR - Invalid ticker: BAD
MSFT: {'ticker': 'MSFT', 'price': 100.0}
```

3.9 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 (e.g., Monte Carlo, optimization)

- Number crunching tasks
  → *Use multiprocessing instead!*

Best Practices

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

4 Multiprocessing & CPU-Bound Tasks

4.1 Why Multiprocessing?

The GIL Problem:

- **Threads share one GIL (Global Interpreter Lock)**
  - Only *one* thread runs Python code at a time.
  - CPU-bound code does **not** speed up with threading.

The Solution: Multiprocessing

- **Use separate processes** (separate memory, separate GIL).
- **The Challenge:** Native Python multiprocessing is notoriously fragile in Jupyter Notebooks (serialization errors!).

The Tool: `joblib`

- The standard for scientific computing (used by scikit-learn).
- **Notebook-friendly:** Works where standard libraries fail.
- **Easy Syntax:** Resembles a simple loop.

4.2 The Syntax: From Loop to Parallel

Sequential Loop (List Comprehension):

```
[slow_function(x) for x in data]
```

Parallel Loop (Joblib):

```
from joblib import Parallel, delayed

# n_jobs=-1 means "use all CPU cores"
Parallel(n_jobs=-1)(delayed(slow_function)(x) for x in data)
```

> **Tip**
>
> **Mental Model:** Think of `delayed(func)(args)` as wrapping your function call in a "package" to be sent to another CPU core.

4.3 Example: Parallel Monte Carlo

Let's estimate using all your laptop's cores.

```
from joblib import Parallel, delayed
import numpy as np


def monte_carlo_pi(n_samples):
    """Simulate points to estimate Pi"""
    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
```

Run in Parallel:

```
%%time

n_tasks = 100
samples_per_task = 1_000_000
results = Parallel(n_jobs=-1)(
    delayed(monte_carlo_pi)(samples_per_task) for _ in range(n_tasks)
)

pi_estimate = np.mean(results)
print(f"Estimated pi: {pi_estimate:.5f}")
```

```
Estimated pi: 3.14168
CPU times: user 35.4 ms, sys: 5.38 ms, total: 40.8 ms
Wall time: 147 ms
```

4.4 When to use what?

Task Type	Examples	Recommended Tool
I/O Bound	Web scraping, API calls, File reading	`ThreadPoolExecutor` (Standard Lib)
CPU Bound	Simulation, Optimization, Estimation	`joblib` (Easier & Robust)

> **Note**
>
> **Pro Tip:** `joblib` can also handle threading!
> Just use: `Parallel(n_jobs=4, prefer="threads")( ... )`

4.5 Debugging Parallel Code

Parallel bugs can be hard to find because of:

- **Non-deterministic behavior**
- **Errors in worker processes**
- **Difficulty reproducing issues**

Strategies for Debugging

1. **Start sequential:** Make sure your code works with `max_workers=1`.
2. **Catch exceptions:** Always wrap `future.result()` (or similar) in `try/except`.
3. **Use logging:** Print statements can get mixed up—prefer proper logging.

> **Tip**
>
> **Golden Rule:**
>
> - If it works with `max_workers=1`, it should work with more.
> - If not, you have a **parallelism bug**.

5 Accelerating Quantitative Macroeconomics: Numba

5.1 What is Numba?

Python is dynamic and flexible, but this comes at a cost: **loops are slow**. Every iteration requires type-checking and memory allocation.

Numba is a **Just-In-Time (JIT) compiler** that translates a subset of Python and NumPy code into fast machine code (using LLVM).

- **`@njit` (nopython mode):** The core decorator. It attempts to compile the decorated function entirely without the Python interpreter. If it fails, it raises an error.
- **`parallel=True`:** Enables automatic parallelization of array operations and explicit parallel loops.
- **`prange`:** "Parallel Range". A replacement for `range` that tells Numba: *"It is safe to run iterations of this loop in any order, on different CPU cores."*

5.2 Scenario: Multi-Sector Aggregate Risk

Imagine an economy with **3 major sectors** (e.g., Manufacturing, Services, Tech). We want to estimate the **"Tail Risk"** (5th percentile outcome) of Aggregate GDP growth.

- We simulate **1 million** stochastic scenarios.
- Each sector has its own productivity trend (μ_i) and volatility (σ_i).

. . .

```
import numpy as np
import time
```

```
def simulate_gdp_tail_risk(n_sims, weights, mu, sigma):
    """
    Simulates aggregate GDP growth scenarios to find the 5th percentile (Tail Risk).
    """
    gdp_growth = np.empty(n_sims)

    # --- The Bottleneck ---
    # Python loops have high overhead for simple math
    for i in range(n_sims):
        sector_shocks = np.random.randn(len(weights))
        sector_growth = sector_shocks * sigma + mu
        gdp_growth[i] = np.dot(weights, sector_growth)
    # ----------------------
```

```
    gdp_growth.sort()
    # Return the 5th percentile (Left-tail risk)
    return gdp_growth[int(n_sims * 0.05)]
```

We now run the simulation.

```
# Parameters
n_sims = int(1e6)
sector_weights = np.array([0.2, 0.5, 0.3])  # Manuf, Services, Tech
mu = np.array([0.01, 0.02, 0.03])  # Trend growth
sigma = np.array([0.05, 0.04, 0.08])  # Volatility

# Timing
start_time = time.time()
risk_result = simulate_gdp_tail_risk(n_sims, sector_weights, mu, sigma)
pure_time = time.time() - start_time

print(f"5% Tail Risk (GDP Growth): {risk_result:.4f}")
print(f"Python Time: {pure_time:.4f} seconds")
```

```
5% Tail Risk (GDP Growth): -0.0331
Python Time: 1.0671 seconds
```

5.3 The Numba Solution (`@njit`)

We add the `@njit` decorator. This compiles the loop into machine code, eliminating Python's interpreter overhead.

```
import numba
```

```
@numba.njit
def simulate_gdp_numba(n_sims, weights, mu, sigma):
    gdp_growth = np.empty(n_sims)
    for i in range(n_sims):
        sector_shocks = np.random.randn(len(weights))
        sector_growth = sector_shocks * sigma + mu
        gdp_growth[i] = np.dot(weights, sector_growth)
```

```
    gdp_growth.sort()
    return gdp_growth[int(n_sims * 0.05)]
```

```
# Warmup (Compile the function)
_ = simulate_gdp_numba(100, sector_weights, mu, sigma)

# Timing
start_time = time.time()
risk_numba = simulate_gdp_numba(n_sims, sector_weights, mu, sigma)
numba_time = time.time() - start_time

print(f"Numba Time: {numba_time:.4f} seconds")
print(f"Speedup: {pure_time / numba_time:.1f}x")
```

```
Numba Time: 0.1632 seconds
Speedup: 6.5x
```

5.4 The Numba Parallel Solution (`prange`)

Now we use all CPU cores.

1. Add `parallel=True` to the decorator.
2. Replace `range` with `numba.prange`.

. . .

```
@numba.njit(parallel=True)
def simulate_gdp_parallel(n_sims, weights, mu, sigma):
    gdp_growth = np.empty(n_sims)

    # prange tells Numba: "You can split this loop across cores"
    for i in numba.prange(n_sims):
        sector_shocks = np.random.randn(len(weights))
        sector_growth = sector_shocks * sigma + mu
        gdp_growth[i] = np.dot(weights, sector_growth)
    gdp_growth.sort()
    return gdp_growth[int(n_sims * 0.05)]
```

```
# Warmup
_ = simulate_gdp_parallel(100, sector_weights, mu, sigma)
start_time = time.time()
risk_parallel = simulate_gdp_parallel(n_sims, sector_weights, mu, sigma)
parallel_time = time.time() - start_time

print(f"Parallel Time: {parallel_time:.4f} seconds")
print(f"Total Speedup vs Python: {pure_time / parallel_time:.1f}x")
```

```
Parallel Time: 0.0742 seconds
Total Speedup vs Python: 14.4x
```