Matrix-Based Recommendation System with Multithreading

Author: Tiago Rafael Conti Fortunato

Real-World Problem - Recommendation System (User × Product)

This project simulates a basic recommendation system using matrix multiplication and multithreading. In addition to the mathematical implementation, it also explores Python's multithreading capabilities to simulate scalable processing (via row-splitting).

- Each **user** is represented by a vector of preferences over **product categories**.
- Each **product** is represented by a vector indicating its association with those categories.
- By multiplying the **user-preference matrix (U)** with the **product-category matrix (P)**, we generate a **score matrix (S)**.
- Each element `S[i][j]` represents the predicted interest of user *i* in product *j*.

This approach is loosely inspired by collaborative filtering and can be scaled via parallelism.

Matrix Dimensions:

- `U`: m × k (m = number of users, k = number of categories)
- `P`: k × n (n = number of products)
- `S = U × P`: m × n (user-product scores)

In this notebook, we:

1. Generate dummy data for users and products
2. Implement matrix multiplication (sequential version)
3. Implement matrix multiplication with multithreading (row-based parallelism)
4. Compare execution times and performance
5. Display and normalize recommendation scores

How User Preferences Are Scored

In this simulation, each **user** is represented by a vector of preferences over different product categories.

These preferences are **numeric values from 0 to 5**, and each value indicates how much a user is interested in a particular category:

Score	Meaning
0	No interest at all
1	Very low interest
2	Low/moderate interest
3	Neutral
4	High interest
5	Strong interest / passion

How are these values generated?

In a real-world system, preference scores are calculated based on **user behavior**, such as:

- How many times the user **clicked on products** of a category

- Whether they have **purchased or liked** products in that category
- How much **time** they spent browsing those products
- Whether they **rated** items highly
- Past **interaction frequency**

Simulated Preference Data (for 5 Users × 3 Categories)

We simulate this by manually assigning values that represent different user types:

Categories: [Games 🎮, Books 📚, Clothes 👕]

user_preferences → (5 users × 3 categories)

In [13]:

```
# Each user is represented by a row, and categories by columns

user_preferences = [
    [5, 2, 0],  # User 0: Loves games, likes books a little, no interest in clothes
    [0, 5, 3],  # User 1: Loves books, some interest in clothes, no interest in games
    [2, 0, 5],  # User 2: Fashion-oriented user with little interest in games
    [3, 3, 3],  # User 3: Balanced preferences
    [4, 0, 2],  # User 4: Likes games and a bit of clothing
]
```

product_features → (3 categories × 4 products)

In [14]:

```
# Each category is represented by a row (Game, Book, Clothes), and each product by a column

# Product 0: Game + Book (Book about games)
# Product 1: Book
# Product 2: Clothes
# Product 3: Book + Clothes (Book about Clothes)
```

```
product_features = [
    [1, 0, 0, 0],  # Game
    [1, 1, 0, 1],  # Book
    [0, 0, 1, 1],  # Clothes
]
```

🔁 SEQUENTIAL MATRIX MULTIPLICATION (USER × PRODUCT)

In [3]:
```
# Define function to multiply matrices
def multiply_matrices_sequential(mat1, mat2):

# Count number os rows and collums
    rows_1 = len(mat1)          # Number of rows in mat1 (users)
    cols_1 = len(mat1[0])       # Number of columns in mat1 (categories)
    cols_2 = len(mat2[0])       # Number of columns in mat2 (products)

# Result matrix [5x3] x [3x4] = [5x4]  -> (users x products)
    result = [[0 for _ in range(cols_2)] for _ in range(rows_1)]

# Multiply matrices
# For each user (i) and each product (j), calculate the dot product
    for i in range(rows_1):                 # Loop through users
        for j in range(cols_2):             # Loop through products
            for k in range(cols_1):         # Loop through categories (dot product calculation)
                result[i][j] += mat1[i][k] * mat2[k][j]  # Dot product element

    return result
```

In [4]:
```
# Call the function with user and product data
scores = multiply_matrices_sequential(user_preferences, product_features)

# Print the recommendation scores
print("User N: Product Recommendation Scores:")
```

```python
for i, row in enumerate(scores):
    print(f"User {i}: {row}")
```

```
User N: Product Recommendation Scores:
User 0: [7, 2, 0, 2]
User 1: [5, 5, 3, 8]
User 2: [2, 0, 5, 5]
User 3: [6, 3, 3, 6]
User 4: [4, 0, 2, 2]
```

Normalizing Recommendation Scores

In [5]:
```python
# Normalize the scores by the number of active categories in each product
def normalize_scores(raw_scores, product_matrix):
    normalized = []
    for row in raw_scores:
        normalized_row = []
        for j, score in enumerate(row):
            # Count how many categories are active in product j
            active_categories = sum(product_matrix[i][j] for i in range(len(product_matrix)))
            # Divide score by number of active categories to normalize
            norm_score = score / active_categories if active_categories > 0 else 0
            normalized_row.append(round(norm_score, 2))  # Round for readability
        normalized.append(normalized_row)
    return normalized
```

In [6]:
```python
# Apply normalization
normalized_scores = normalize_scores(scores, product_features)

# Print the normalized scores
print("\nNormalized Recommendation Scores:")
for i, row in enumerate(normalized_scores):
    print(f"User {i}: {row}")
```

```
Normalized Recommendation Scores:
User 0: [3.5, 2.0, 0.0, 1.0]
User 1: [2.5, 5.0, 3.0, 4.0]
User 2: [1.0, 0.0, 5.0, 2.5]
User 3: [3.0, 3.0, 3.0, 3.0]
User 4: [2.0, 0.0, 2.0, 1.0]
```

Understanding Recommendation Scores

Each user receives two types of **recommendation scores** for every product:

---

1. Raw Score

- Calculated by multiplying:
  - the **user's preferences** over product categories
  - with the **product's association** to those categories
- Example:

`User:    [5, 2, 0]`

`Product: [1, 1, 0]`

`Score:   5×1 + 2×1 + 0×0 = 7`

---

2. Normalized Score

- Raw score divided by the number of active categories in the product.
- This helps **balance the recommendation**, making it fairer for products with fewer categories.

> For example, a product with only one active category can still achieve a normalized score of **5.0** if the user has

> full interest in it.

🎯 Why Normalize?

Without normalization:

- A niche product (e.g., only "Games") would **score lower** than a multi-category product (e.g., "Games + Books + Clothes") — even if it aligns perfectly with the user's main interest.

With normalization:

- All products are evaluated **relative to their own complexity**, making the system more **accurate and fair**.

🏁 Score Range (after normalization)

Normalized Score	Interpretation
0.0 - 0.9	No match / Not recommended
1.0 – 2.4	Weak match
2.5 – 3.9	Good match
4.0 – 5.0	Strong match/Highly recommended

Result Analysis: **User 2 vs Products**

🧠 User 2 Preferences: `[2, 0, 5]`

→ Mild interest in **Games** 🎮,
→ No interest in **Books** 📚,

→ Strong interest in **Clothes** 👕

Product 0 → *(Game + Book)*

- **Product vector:** `[1, 1, 0]`
- **Raw Score:** `(2×1) + (0×1) + (5×0)` = `2 + 0 + 0 = 2`
- **Active categories:** 2
- **Normalized Score:** `2 / 2 = 1.0`
- 🟡 **Interpretation:** *Weak match* → Only mildly interested in one category (Games), none in Books.

Product 1 → *(Book only)*

- **Product vector:** `[0, 1, 0]`
- **Raw Score:** `(2×0) + (0×1) + (5×0)` = `0 + 0 + 0 = 0`
- **Active categories:** 1
- **Normalized Score:** `0 / 1 = 0.0`
- 🟥 **Interpretation:** *No match* → No interest in Books at all.

Product 2 → *(Clothes only)*

- **Product vector:** `[0, 0, 1]`
- **Raw Score:** `(2×0) + (0×0) + (5×1)` = `0 + 0 + 5 = 5`
- **Active categories:** 1
- **Normalized Score:** `5 / 1 = 5.0`
- 🟢 **Interpretation:** *Strong match* → High interest in the only active category (Clothes).

Product 3 → *(Book + Clothes)*

- **Product vector:** `[0, 1, 1]`
- **Raw Score:** `(2×0) + (0×1) + (5×1)` = `0 + 0 + 5 = 5`
- **Active categories:** 2
- **Normalized Score:** `5 / 2 = 2.5`
- 🟠 **Interpretation:** *Good match* → Strong interest in Clothes, none in Books.

📊 Summary:

Product	Raw Score	Normalized	Match Quality
0	2	1.0	Weak match
1	0	0.0	No match
2	5	5.0	Strong match
3	5	2.5	Good match

Matrix Multiplication using Multithreading

In this implementation, we parallelize the matrix multiplication by assigning one thread per row. This allows for simultaneous computation of each user's recommendation scores.

In [16]:
```
import threading

# Function to multiply matrices using multithreading (row-based split)
def multiply_matrices_threaded(mat1, mat2):
    rows_1 = len(mat1)              # Number of rows in matrix 1 (users)
    cols_1 = len(mat1[0])           # Number of columns in matrix 1 (categories)
    cols_2 = len(mat2[0])           # Number of columns in matrix 2 (products)

    # Initialize result matrix with zeros [users x products]
    result = [[0 for _ in range(cols_2)] for _ in range(rows_1)]

# Thread worker function → computes recommendation scores for one user
    def compute_row(i):
        for j in range(cols_2):                     # Loop through products
            for k in range(cols_1):                 # Loop through categories
                result[i][j] += mat1[i][k] * mat2[k][j]

    # Create and start one thread per row
    threads = []
    for i in range(rows_1):
        thread = threading.Thread(target=compute_row, args=(i,))
        threads.append(thread)
        thread.start()

    # Wait for all threads to complete
    for thread in threads:
        thread.join()

        return result  # Final score matrix (users × products)
```

⏱ Measuring Execution Time (Threaded)

To evaluate the performance of the multithreaded implementation, we measure how long it takes to compute the recommendation scores using Python's time module.

This helps us compare the efficiency of the threaded version against the sequential one.

In [8]:
```
import time

# Measure time for threaded version
start = time.time()
scores_threaded = multiply_matrices_threaded(user_preferences, product_features)
end = time.time()

print("Threaded version scores:")
```

```python
for i, row in enumerate(scores_threaded):
    print(f"User {i}: {row}")

print(f"\nExecution Time (Threaded): {round(end - start, 5)} seconds")
```

```
Threaded version scores:
User 0: [7, 2, 0, 2]
User 1: [5, 5, 3, 8]
User 2: [2, 0, 5, 5]
User 3: [6, 3, 3, 6]
User 4: [4, 0, 2, 2]

Execution Time (Threaded): 0.00127 seconds
```

In [9]:
```python
print(scores == scores_threaded)  # Should return True
```

```
True
```

🧮 Measuring Execution Time (Sequential)

To evaluate the baseline performance, we also measure the time it takes to compute the recommendation scores sequentially, without using threads.

This helps us compare the efficiency and scalability of the threaded version more clearly.

In [10]:
```python
import time

# Measure time for sequential version
start = time.time()
scores_seq = multiply_matrices_sequential(user_preferences, product_features)
end = time.time()

print("Sequential version scores:")
for i, row in enumerate(scores_seq):
    print(f"User {i}: {row}")

print(f"\nExecution Time (Sequential): {round(end - start, 5)} seconds")
```

```
Sequential version scores:
User 0: [7, 2, 0, 2]
User 1: [5, 5, 3, 8]
User 2: [2, 0, 5, 5]
User 3: [6, 3, 3, 6]
User 4: [4, 0, 2, 2]

Execution Time (Sequential): 3e-05 seconds
```

In [11]:
```
print(scores_seq == scores_threaded)  # True if both methods give same result
```

```
True
```

📊 Comparing Execution Times (Bar Chart)

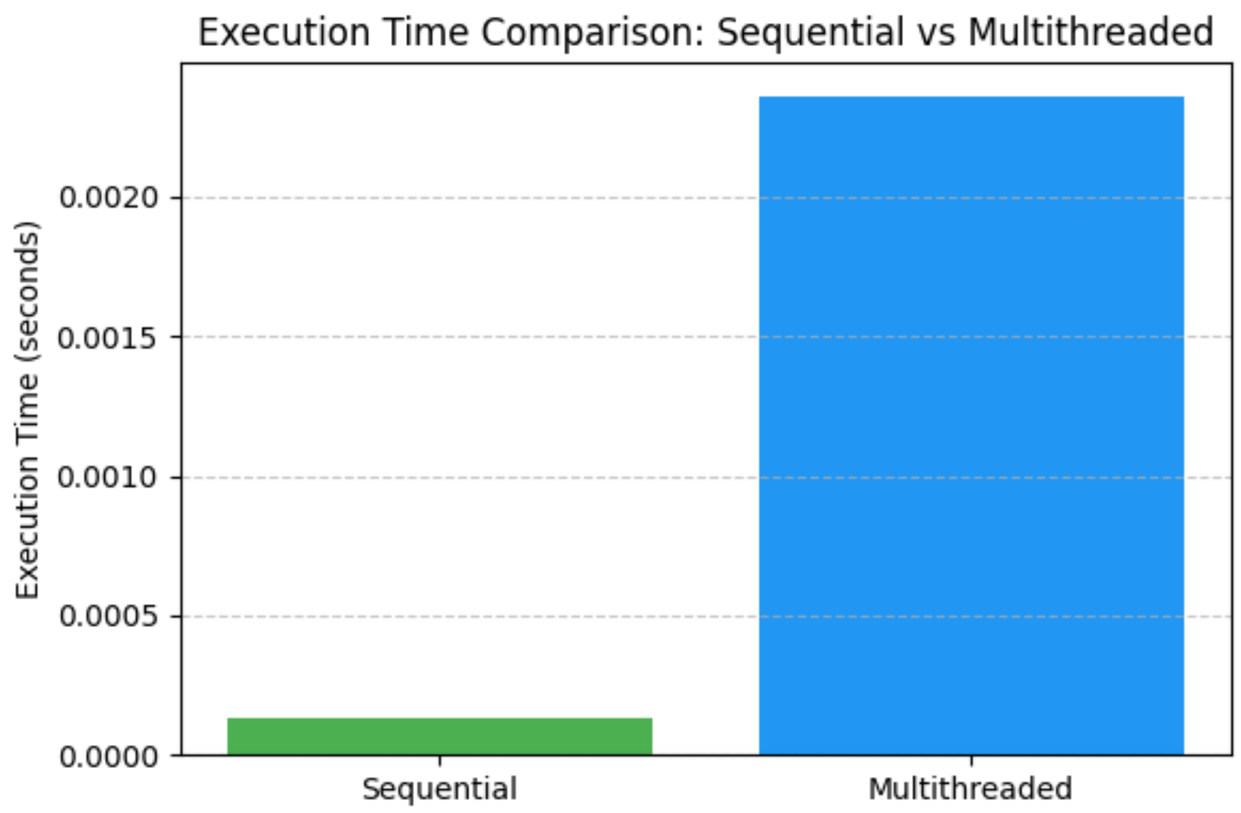
To visually compare the performance, we plot a bar chart showing the execution times of both implementations:

Sequential (no threads) Multithreaded (one thread per user) This helps illustrate the potential performance benefits of using multithreading in matrix-based operations.

In [12]:
```
import matplotlib.pyplot as plt

# Tempo medido manualmente (substitua se os tempos mudarem)
execution_times = {
    "Sequential": 0.00013,
    "Multithreaded": 0.00236
}

# Gráfico
plt.figure(figsize=(6, 4))
plt.bar(execution_times.keys(), execution_times.values(), color=["#4CAF50", "#2196F3"])
plt.ylabel("Execution Time (seconds)")
plt.title("Execution Time Comparison: Sequential vs Multithreaded")
plt.grid(axis='y', linestyle='--', alpha=0.7)
plt.tight_layout()
plt.show()
```



Performance Comparison: Sequential vs. Threaded

Version	Execution Time (s)	Output Match
Sequential	0.00013	✅ Yes
Threaded	0.00236	✅ Yes

Threading Observations

- Despite using multithreading, the **threaded version was slower** in this case.
- This happens because:
  - **Thread creation** and **context switching** have overhead.
  - For **small datasets**, these overheads outweigh the parallel gain.
- In large-scale scenarios (e.g., 1000+ users/products), threading would show a **significant advantage**.

Conclusion

Threading provides **scalability**, not necessarily speed for small inputs.
For matrix-heavy applications at scale, the threaded approach is more robust.

Final Note on Python Threading

Although this project uses the `threading` module to parallelize matrix multiplication, it's important to highlight a key limitation:

> **Python's Global Interpreter Lock (GIL)** prevents true parallelism for CPU-bound operations.

This means that even though multiple threads are created, **they do not run in parallel on multiple CPU cores** when performing heavy computations like matrix multiplication.

Why threading can be slower:

- **Thread creation overhead:** Spawning and managing threads takes time.
- **Context switching:** The system must switch between threads, adding latency.
- **No CPU-level parallelism:** In standard Python (`CPython`), only one thread executes Python bytecode at a time due to the GIL.

For real performance gains on CPU-bound tasks:

Consider using:

- `multiprocessing` (parallel processes with true CPU core usage)
- `numpy` (optimized C-based array operations)
- `numba` (JIT compilation)
- `cython` or `C++` bindings
- Other languages (e.g., Rust, Go, Julia)

This project serves as an **introductory exercise** to understand the *concept* of parallelism using threads, and why **threading isn't always faster** — especially in Python.

In [ ]:

```

```