ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN
KHOA MẠNG MÁY TÍNH VÀ TRUYỀN THÔNG

BÁO CÁO THỰC HÀNH BUỔI 1

Môn: NT538 – GIẢI THUẬT XỬ LÝ SONG SONG VÀ PHÂN BỐ

Giảng viên hướng dẫn : ThS. LÊ MINH KHÁNH HỘI
Lớp : NT538.P21.1

Key của nhóm: q+jRThmNNqESFbG9hWN1NL7R5iaMpxtEhl+D0hHhU/d7

Nhóm thực hiện – Nhóm 6
1. Hồ Hải Dương MSSV: 21520202
2. Trần Ngọc Phương Anh MSSV: 21521839

THỦ ĐỨC – 6/2025

MỤC LỤC NỘI DUNG

BÁO CÁO CHI TIẾT

CHALLENGE 0

A. Mô tả bài toán

Bài toán: Cho một mảng số nguyên A có N phần tử ($N \leq 10^6$).

Yêu cầu: Tính mảng tổng tiền tố (prefix sum) của mảng A bằng phương pháp song song, nhằm tăng tốc độ xử lý khi dữ liệu lớn.

Kết quả: Trả về một mảng mới có cùng độ dài, trong đó phần tử tại vị trí i là tổng các phần tử từ A[0] đến A[i].

Ví dụ minh họa: Input: A = [1, 2, 3, 4]; Output: [1, 3, 6, 10]

B. Ý tưởng và phương pháp song song

B.1. Chia mảng A thành nhiều đoạn nhỏ (chunks)

Để thực hiện song song hóa quá trình tính tổng tiền tố, bước đầu tiên là chia mảng đầu vào A thành nhiều đoạn nhỏ, gọi là chunk. Mỗi chunk bao gồm một số lượng phần tử gần bằng nhau, tùy theo số lượng tiến trình hoặc luồng mà hệ thống có thể xử lý đồng thời.

Giả sử mảng A có N phần tử và hệ thống hỗ trợ P tiến trình (ví dụ: P = số lượng CPU core), ta chia mảng A thành P đoạn như sau:

- Mỗi chunk có độ dài xấp xỉ N / P phần tử.
- Với các phần tử dư ra (nếu N không chia hết cho P), ta có thể phân bổ vào các chunk đầu tiên để cân bằng khối lượng công việc.

Mục tiêu của việc chia chunk là:

- Giảm khối lượng tính toán của mỗi tiến trình.
- Tận dụng hiệu quả tài nguyên xử lý song song của CPU.

- Bảo đảm rằng mỗi chunk có thể được xử lý độc lập ở bước đầu tiên mà không cần thông tin từ các chunk khác.

Ví dụ:

- Nếu mảng A có 1.000.000 phần tử và sử dụng 4 tiến trình, mỗi chunk sẽ gồm khoảng 250.000 phần tử.
- Sau khi chia, ta thu được: chunk1 = A[0:249999], chunk2 = A[250000:499999],...

B.2. Tính prefix sum cục bộ cho từng chunk

Sau khi chia mảng A thành nhiều đoạn nhỏ (chunks) và tính được các giá trị offset tương ứng cho từng chunk, chương trình tiến hành xử lý từng chunk theo hướng song song bằng cách sử dụng multiprocessing.Pool.

- Mỗi tiến trình sẽ nhận một tuple gồm:
- Một đoạn nhỏ chunk của mảng A.

Giá trị offset tương ứng – là tổng các phần tử thuộc các chunk đứng trước.

Bên trong mỗi tiến trình, đoạn mã thực hiện phép cộng dồn (prefix sum) cho toàn bộ phần tử trong chunk, bắt đầu từ offset đã cho, tức là tổng tích lũy khởi đầu không phải là 0 mà là giá trị offset. Nhờ cách làm này, kết quả prefix sum được tính trực tiếp theo đúng vị trí trong toàn mảng A, không cần hậu xử lý gì thêm.

Ví dụ: nếu một chunk là [5, 3, 2] và offset là 10, thì prefix sum tính được sẽ là [15, 18, 20].

Việc xử lý này được thực hiện bằng cách truyền danh sách các (chunk, offset) vào pool.map() để thực thi đồng thời, bảo đảm tận dụng đa luồng hiệu quả.

B.3. Tính tổng từng chunk để xác định offset

Sau khi chia mảng A thành các đoạn nhỏ, bước tiếp theo là tính tổng của từng chunk để xác định offset – đây là phần quan trọng giúp các tiến trình sau đó có thể tính toán chính xác prefix sum cục bộ của mình trong ngữ cảnh toàn cục.

Quá trình được thực hiện như sau:

- Duyệt qua từng chunk và tính tổng tất cả phần tử trong đoạn đó. Kết quả được lưu vào danh sách sums.
- Sau đó, dựa trên danh sách sums, chương trình tính offset cho từng chunk bằng cách cộng dồn các tổng chunk trước đó. Offset tại chunk i sẽ là tổng các chunk 0 đến i-1.
- Danh sách offset này bảo đảm rằng khi mỗi tiến trình xử lý prefix sum cho đoạn của mình, nó sẽ cộng thêm offset đúng để bảo toàn tính đúng đắn của giá trị prefix sum toàn cục.

Giả sử mảng A có 12 phần tử và được chia thành 3 chunk:

- chunk_0 = [1, 2, 3, 4] → tổng là 10
- chunk_1 = [5, 6, 7, 8] → tổng là 26
- chunk_2 = [9, 10, 11, 12] → tổng là 42

Khi đó:

- Offset của chunk_0 là 0
- Offset của chunk_1 là 10
- Offset của chunk_2 là 10 + 26 = 36

Nhờ vậy, các prefix sum cục bộ tính được ở mỗi chunk sau khi cộng thêm offset sẽ phản ánh đúng giá trị thực của prefix sum toàn cục.

B.4. Ghép kết quả lại thành mảng cuối cùng

Sau khi từng tiến trình đã tính toán xong phần prefix sum cục bộ của mình (có cộng thêm offset), bước cuối cùng là tổng hợp toàn bộ các kết quả này lại thành một mảng prefix sum hoàn chỉnh. Cụ thể:

- Mỗi tiến trình trả về một mảng con chứa prefix sum đã được tính toán cho chunk tương ứng.
- Tại tiến trình chính (main process), chương trình duyệt qua tất cả các mảng con này theo đúng thứ tự ban đầu, và nối chúng lại bằng cách dùng lệnh extend() hoặc +=.
- Do thứ tự của các chunk không thay đổi trong quá trình xử lý song song, việc ghép nối này bảo đảm tính đúng đắn và tuần tự của mảng kết quả cuối cùng.

Giả sử sau khi xử lý, ta có 3 mảng con:

- [1, 3, 6, 10]
- [15, 21, 28, 36]
- [45, 55, 66, 78]
- Kết quả sẽ là: [1, 3, 6, 10, 15, 21, 28, 36, 45, 55, 66, 78]. Đây chính là mảng tổng tiền tố (prefix sum) hoàn chỉnh của đầu vào ban đầu.

C. Phân tích các đoạn code quan trọng

C.1. Hàm local_prefix()

Đây là hàm xử lý chính để tính **tổng tiền tố (prefix sum)** cho một đoạn nhỏ (chunk) của mảng, **sau khi đã cộng thêm offset** được tính từ các đoạn trước.

```
3   def local_prefix(args):
4       chunk, offset = args
5       total = offset
6       prefix = []
7       for x in chunk:
8           total += x
9           prefix.append(total)
10      return prefix, total
```

Giải thích:

- chunk: là một đoạn nhỏ của mảng gốc A (đã được chia đều).
- offset: là tổng các phần tử nằm **trước** chunk này, dùng để bảo đảm mỗi phần tử trong chunk tính tổng đúng với vị trí của nó trong toàn bộ mảng.
- total: khởi tạo bằng offset, sau đó cộng dồn các phần tử trong chunk.
- prefix: là mảng lưu tổng tiền tố của chunk sau khi cộng offset.
- Hàm trả về:
    - prefix: danh sách tổng tiền tố đã cộng offset cho chunk này.
    - total: là tổng tất cả phần tử trong chunk sau khi cộng dồn, để sử dụng về sau nếu cần.

Ví dụ minh họa:

Giả sử:

- chunk = [10, 20, 30]
- offset = 100

Quá trình tính toán:

- total = 100 + 10 = 110 → prefix[0] = 110
- total = 110 + 20 = 130 → prefix[1] = 130
- total = 130 + 30 = 160 → prefix[2] = 160

Kết quả trả về: ([110, 130, 160], 160)

C.2. Tính toán và chia mảng đầu vào thành các chunks

Việc chia mảng thành các đoạn nhỏ (chunk) giúp tận dụng song song hóa bằng cách giao từng đoạn cho một tiến trình xử lý riêng biệt.

```
15      num_workers = min(8, multiprocessing.cpu_count(), (n + 99999)//100000)
16      chunk_size = (n + num_workers - 1) // num_workers
17      chunks = [A[i:i+chunk_size] for i in range(0, n, chunk_size)]
```

Giải thích:

- num_workers: số tiến trình sẽ tạo ra. Được tính sao cho:
  - Không vượt quá 8 (tránh tạo quá nhiều tiến trình),
  - Không vượt quá số lõi CPU thực tế (multiprocessing.cpu_count()),
  - Tính toán dựa vào độ lớn mảng (mỗi 100000 phần tử nên có 1 tiến trình).
- chunk_size: kích thước mỗi đoạn nhỏ. Được tính để bảo đảm chia đều mảng A cho các tiến trình.
- chunks: danh sách các đoạn nhỏ đã chia từ mảng A, ví dụ:

  A = [1, 2, 3, 4, 5, 6]

  chunk_size = 2

  => chunks = [[1, 2], [3, 4], [5, 6]]

Việc chia mảng như vậy giúp mỗi tiến trình xử lý nhanh hơn, và các tiến trình có thể chạy **đồng thời** trên nhiều lõi CPU.

C.3. Tính offset cho từng chunk và xác định offset

Sau khi chia mảng thành các đoạn nhỏ (chunks), bước tiếp theo là tính tổng của từng chunk. Tổng này được dùng để xác định **offset** – tức là giá trị cần cộng thêm vào từng chunk để bảo đảm tính đúng tổng tiền tố trên toàn mảng.

```
20      offsets = [0]
21      sums = []
22      for chunk in chunks:
23          sums.append(sum(chunk))
24      for i in range(1, len(sums)):
25          offsets.append(offsets[i-1] + sums[i-1])
```

Giải thích:

- **sums**: chứa tổng của từng chunk. Ví dụ, nếu mảng gốc chia thành 3 chunks:

  chunk 0 = [1, 2, 3] → sum = 6

  chunk 1 = [4, 5] → sum = 9

  chunk 2 = [6, 7] → sum = 13

  => sums = [6, 9, 13]

- **offsets**: mảng này tính toán offset cần cộng thêm cho mỗi chunk. Nó bảo đảm các chunk sau phải cộng thêm tổng của tất cả các chunk trước đó để tạo ra tổng tiền tố liên tục:

  offsets[0] = 0

  offsets[1] = 0 + 6 = 6

  offsets[2] = 6 + 9 = 15

=> offsets = [0, 6, 15]

Vai trò:

Offset giúp chuyển bài toán từ "tính prefix sum của toàn mảng" thành "tính prefix sum của từng phần nhỏ, rồi cộng dồn lại đúng vị trí". Đây là chìa khóa để giữ đúng logic prefix sum trong môi trường song song.

C.4. Xử lý song song từng chunk và ghép kết quả

Sau khi xác định được các offset, ta tiến hành xử lý từng chunk song song để tính tổng tiền tố cục bộ, rồi ghép các kết quả lại thành mảng tổng tiền tố hoàn chỉnh.

```
27      # Gán offset cho từng chunk
28      tasks = list(zip(chunks, offsets))
29
30      # Song song hóa tính prefix từng chunk
31      with multiprocessing.Pool(num_workers) as pool:
32          partial = pool.map(local_prefix, tasks)
33
34      # Ghép kết quả
35      result = []
36      for prefix, _ in partial:
37          result.extend(prefix)
38      return result
```

Giải thích:

- Hàm **local_prefix()** thực hiện việc tính tổng tiền tố cho từng chunk riêng lẻ, bắt đầu từ một giá trị offset đã tính sẵn. Tổng cục bộ được lưu trong danh sách prefix.
- Danh sách tasks chứa các cặp (chunk, offset) tương ứng. Mỗi phần việc như vậy sẽ được giao cho một tiến trình riêng nhờ sử dụng multiprocessing.Pool.

- Sau khi song song xử lý xong, kết quả từ từng chunk được **nối lại theo thứ tự gốc** bằng cách dùng extend() để tạo thành mảng prefix sum cuối cùng.

Vai trò:

Đây là phần cốt lõi để bảo đảm mảng tổng tiền tố được tính nhanh hơn bằng cách tận dụng nhiều lõi CPU. Mỗi tiến trình hoạt động độc lập trên phần dữ liệu riêng, không cần đồng bộ phức tạp, giúp chương trình đạt được hiệu năng cao mà vẫn bảo đảm độ chính xác của kết quả.

D. Tiềm năng tăng tốc và kết luận

Tiềm năng tăng tốc:

Bằng cách chia mảng A thành nhiều đoạn nhỏ và xử lý từng đoạn **song song**, chương trình tận dụng được **nhiều lõi CPU** cùng lúc. Điều này giúp:

- **Giảm đáng kể thời gian tính toán** so với cách làm tuần tự, đặc biệt khi số lượng phần tử N lớn (có thể lên tới hàng triệu).
- Với hệ thống có từ 4 đến 8 lõi CPU, tốc độ xử lý có thể tăng **gấp 3–5 lần**, tùy theo kích thước đầu vào và khả năng phân phối đều công việc.
- Mức tăng tốc đạt được sẽ giảm dần nếu:
    - Số phần tử quá ít (chi phí tạo process cao hơn lợi ích).
    - Số tiến trình quá lớn làm tăng chi phí quản lý.

Kết luận:

Bài toán tính tổng tiền tố (prefix sum) là một ví dụ điển hình có thể áp dụng hiệu quả **tư duy song song hóa**. Bằng cách phân chia dữ liệu thông minh và sử dụng thư viện multiprocessing, nhóm đã triển khai một lời giải **vừa đúng**, **vừa tối ưu**, giúp rút ngắn thời gian xử lý rõ rệt mà không làm ảnh hưởng đến tính đúng đắn của kết quả.

CHALLENGE 1

A. Mô tả bài toán

Bài toán: Cho mảng số nguyên A gồm N phần tử ($N \leq 10^6$) và một số nguyên Q.

Yêu cầu: Tính F(A[i] + 1) mod Q cho từng phần tử A[i], với F(n) là số Fibonacci thứ n (tính theo quy ước F(0) = 1, F(1) = 1).

Kết quả: Trả về một mảng mới B[] với mỗi phần tử B[i] = F(A[i] + 1) % Q.

Ví dụ minh họa:

- Input: A = [0, 1, 3], Q = 1000
- Output: [1, 1, 5] → vì F(1) = 1, F(2) = 1, F(4) = 5

Khó khăn: Với N lớn (tới hàng triệu phần tử), việc tính số Fibonacci theo cách truyền thống sẽ rất chậm. Cần một giải pháp tính Fibonacci hiệu quả và song song để đạt hiệu năng tối ưu.

B. Ý tưởng và phương pháp song song

B.1. Chia mảng giá trị cần tính Fibonacci thành các batch nhỏ

Sau khi đọc dữ liệu từ file input, chương trình lấy các phần tử của mảng A và tăng mỗi giá trị lên 1 để phù hợp với quy ước Fibonacci trong đề bài (tức là F(0) = 1, F(1) = 1,...). Việc cộng thêm 1 là cần thiết vì trong hàm Fibonacci được cài đặt, giá trị đầu tiên F(0) đã được tính là 1, nên để truy xuất đúng vị trí mong muốn thì phải dịch chỉ số lên một đơn vị.

Tiếp theo, toàn bộ danh sách này được chia thành các phần nhỏ (gọi là **batch**) để dễ dàng xử lý song song. Việc chia batch giúp tận dụng nhiều CPU core, tránh việc để một tiến trình đơn xử lý toàn bộ dữ liệu, từ đó tăng hiệu suất tính toán rõ rệt.

Số phần tử trong mỗi batch được tính sao cho phù hợp với số lượng CPU hiện có trên hệ thống. Sử dụng công thức: chunk = max(5000, len(values) // (n_proc * 2))

B.2. Sử dụng thuật toán Fast Doubling để tăng tốc tính Fibonacci

Để tính số Fibonacci thứ n một cách hiệu quả, chương trình sử dụng thuật toán Fast Doubling – một phương pháp đặc biệt giúp rút ngắn thời gian tính toán từ O(n) xuống còn O(log n) bằng cách sử dụng công thức truy hồi đặc biệt cho chỉ số chẵn và lẻ.

Giả sử ta cần tính F(n), thuật toán sẽ dựa trên:

- $F(2k) = F(k) \times [2 \times F(k+1) - F(k)]$
- $F(2k+1) = F(k+1)^2 + F(k)^2$

Tức là từ hai giá trị F(k) và F(k+1), ta có thể suy ra nhanh F(2k) và F(2k+1), tránh việc phải tính dần từng giá trị như cách tuần tự.

Trong chương trình, mỗi số cần tính được xử lý theo bit nhị phân của chỉ số. Điều này giúp giảm số bước tính toán vì thay vì lặp qua từng số từ 0 đến n, ta chỉ cần quét qua từng bit của bin(n).

Ví dụ: Để tính F(10), thay vì phải tính từ F(0) → F(10), ta chỉ cần tính F(5), sau đó áp dụng công thức để lấy F(10), giúp giảm rất nhiều bước. Ngoài ra, phép tính được thực hiện với modulo Q (giá trị lấy từ file input), để giữ cho kết quả không vượt quá giới hạn bộ nhớ và bảo đảm đúng yêu cầu đề bài.

B.3. Sử dụng ProcessPoolExecutor để song song hóa các batch

Do bài toán yêu cầu tính toán hàng loạt số Fibonacci với chỉ số rất lớn, nếu xử lý tuần tự thì sẽ mất rất nhiều thời gian. Vì vậy, chương trình tận dụng thư viện concurrent.futures.ProcessPoolExecutor để thực hiện tính toán song song trên nhiều tiến trình.

Cách thực hiện:

- Dựa vào số lượng lõi CPU (mp.cpu_count()), chương trình xác định số tiến trình phù hợp (thường là 4 hoặc 8).

- Danh sách các chỉ số Fibonacci cần tính được chia thành nhiều batch nhỏ, mỗi batch có kích thước khoảng vài nghìn phần tử. Việc chia nhỏ như vậy giúp:
    - Cân bằng tải giữa các tiến trình.
    - Tránh trường hợp một tiến trình phải xử lý quá lâu trong khi tiến trình khác đã xong.
- Mỗi batch sẽ được gửi đến một tiến trình khác nhau bằng cách sử dụng executor.submit().

Cách hoạt động của ProcessPoolExecutor: Mỗi tiến trình hoạt động độc lập và xử lý 1 batch. Khi tiến trình hoàn thành, kết quả của nó sẽ được gom lại để tạo thành mảng kết quả cuối cùng.

Lợi ích: Rút ngắn thời gian chạy xuống đáng kể, đặc biệt khi số lượng phần tử lớn (ví dụ N lên đến 1 triệu). Không bị "đóng băng" giao diện hay nghẽn I/O vì quá trình xử lý tính toán được thực hiện ở background.

B.4. Gom kết quả từ các tiến trình để tạo danh sách kết quả cuối cùng

Sau khi các tiến trình xử lý xong các batch, kết quả cần được tổng hợp lại để tạo ra mảng kết quả đầu ra hoàn chỉnh.

Cách thực hiện:

- Trong đoạn code:

  for t in tasks:

  result.extend(t.result())

  mỗi biến t đại diện cho một tiến trình đã thực thi xong một batch.
- Phương thức .result() trả về danh sách các số Fibonacci tương ứng với batch đó.

- Hàm extend() được dùng để nối tất cả kết quả của các batch thành một danh sách duy nhất – chính là mảng kết quả cuối cùng cần trả về.

Lý do cần bước này: Mỗi tiến trình chỉ xử lý một phần nhỏ của dữ liệu. Nếu không gom lại, kết quả sẽ bị phân mảnh, không thể sử dụng như một mảng đầu ra liên tục. Bảo đảm giữ nguyên thứ tự xử lý của từng phần tử đầu vào (vì batch chia theo thứ tự ban đầu).

Giả sử ta cần tính Fibonacci cho mảng [4, 10, 20, 50, 100] và chia thành 2 batch:

- Batch 1: [4, 10, 20] → [5, 89, 10946]
- Batch 2: [50, 100] → [12586269025, 573147844013817084101]
- Sau khi gom lại: [5, 89, 10946, 12586269025, 573147844013817084101].

C. Phân tích các đoạn code quan trọng

C.1. Hàm calc_fib_batch(batch, mod)

Đây là hàm cốt lõi của bài toán, có nhiệm vụ tính giá trị **Fibonacci** cho từng phần tử trong một batch nhỏ của mảng đầu vào. Hàm sử dụng **thuật toán Fast Doubling**, giúp giảm độ phức tạp từ O(n) xuống **O(log n)**, rất cần thiết cho việc xử lý những giá trị lớn như F(1.000.000).

Cách hoạt động:

```
4    def calc_fib_batch(batch, mod):
5        output = []
6        for val in batch:
7            a, b = 0, 1
8            # Dùng fast doubling dựa trên bit của val
9            for bit in bin(val)[2:]:
10               temp1 = (a * ((2 * b - a) % mod)) % mod
11               temp2 = (a * a + b * b) % mod
12               if bit == '1':
13                   a, b = temp2, (temp1 + temp2) % mod
14               else:
15                   a, b = temp1, temp2
16           output.append(a)
17       return output
```

Giải thích:

- **val** là chỉ số Fibonacci cần tính.
- Sử dụng biểu diễn nhị phân của val để áp dụng công thức nhân đôi nhanh:
    - F(2k)=F(k)×[2F(k+1)–F(k)]F(2k) = F(k) \times [2F(k+1) – F(k)]F(2k)=F(k)×[2F(k+1)–F(k)]
    - F(2k+1)=F(k+1)2+F(k)2F(2k+1) = F(k+1)^2 + F(k)^2F(2k+1)=F(k+1)2+F(k)2
- Dữ liệu được **modulo Q** tại mỗi bước để tránh tràn số và bảo đảm kết quả hợp lệ.
- Kết quả từng phần tử được lưu vào danh sách output.

Vai trò:

- Nhờ thuật toán này, chương trình có thể tính chính xác và nhanh chóng hàng trăm ngàn số Fibonacci lớn.

- Hàm này còn có thể dễ dàng **parallel hóa** bằng cách truyền các batch nhỏ vào nhiều tiến trình khác nhau.

C.2. Đọc và xử lý input trong hàm MAIN()

Việc đọc và chuẩn bị dữ liệu đầu vào là bước đầu tiên quan trọng trước khi tiến hành xử lý song song. Dữ liệu được đọc từ file "input.txt" theo định dạng đã cho trước.

```
19  def MAIN(input_filename="input.txt"):
20      with open(input_filename) as f:
21          items = list(map(int, f.read().split()))
22      if not items:
23          return []
24      N = items[0]
25      Q = items[1]
26      arr = items[2:2+N]
```

Giải thích:

- **with open(...) as f:**: Mở file đầu vào ở chế độ đọc.
- **f.read().split()**: Đọc toàn bộ nội dung file và tách thành danh sách các chuỗi (các số nguyên).
- **list(map(int, ...))**: Ép tất cả các chuỗi sang số nguyên.
- **N**: Số lượng phần tử trong mảng A.
- **Q**: Modulo cần dùng khi tính Fibonacci.
- **arr**: Danh sách A gồm N số nguyên.

Vai trò:

- Bảo đảm dữ liệu được nạp đúng định dạng và có thể dễ dàng chia nhỏ để xử lý song song.

- Cắt đúng số lượng phần tử đầu vào, loại bỏ dữ liệu thừa hoặc lỗi định dạng.

C.3. Chia batch và song song hóa bằng ProcessPoolExecutor

Sau khi thu được danh sách các chỉ số cần tính Fibonacci, chương trình chia dữ liệu thành các lô nhỏ (batch) để phân phối đều cho các tiến trình xử lý đồng thời.

```
27      values = [x + 1 for x in arr]  # F(0)=1, F(1)=1, ... nên cần +1
28      # Chia batch nhỏ hơn một chút cho load đều CPU
29      n_proc = mp.cpu_count()
30      chunk = max(5000, len(values)//(n_proc*2))
31      batches = [values[i:i+chunk] for i in range(0, len(values), chunk)]
```

Giải thích:

- **values = [x + 1 for x in arr]**: Dịch chuyển chỉ số vì đề cho A[i] → cần tính F(A[i]+1) do F(0) = 1, F(1) = 1.
- **n_proc = mp.cpu_count()**: Xác định số lượng tiến trình tối đa tương ứng với số CPU trên hệ thống.
- **chunk = max(5000, len(values)//(n_proc*2))**: Kích thước mỗi batch bảo đảm không quá nhỏ (ít nhất 5000 phần tử) để tránh overhead nhưng vẫn cân bằng tải giữa các tiến trình.
- **batches = [...]**: Tạo danh sách các batch con từ values.

Vai trò:

- Giảm thiểu chi phí khởi tạo tiến trình bằng cách chia dữ liệu hợp lý.
- Tăng hiệu quả xử lý bằng cách bảo đảm tiến trình nào cũng có việc làm.
- Đây là bước tiền đề giúp hàm calc_fib_batch xử lý song song hiệu quả ở bước tiếp theo.

C.4. Gom kết quả từ các tiến trình

Sau khi đã chia dữ liệu thành các batch nhỏ, chương trình sử dụng ProcessPoolExecutor để xử lý từng batch song song, giúp tăng tốc tính toán.

```
33      with ProcessPoolExecutor(max_workers=n_proc) as executor:
34          tasks = [executor.submit(calc_fib_batch, b, Q) for b in batches]
35          for t in tasks:
36              result.extend(t.result())
37      return result
```

Giải thích:

- **ProcessPoolExecutor(max_workers=n_proc)**: Tạo một pool tiến trình với số lượng tối đa bằng số CPU.
- **executor.submit(...)**: Gửi các batch vào các tiến trình để tính toán bất đồng bộ.
- **calc_fib_batch(b, Q)**: Mỗi tiến trình gọi hàm calc_fib_batch, thực hiện tính Fibonacci modulo Q cho toàn bộ batch b.
- **result.extend(t.result())**: Thu kết quả từ từng tiến trình và ghép lại thành một danh sách kết quả chung.

Vai trò:

- Tận dụng tối đa năng lực tính toán của CPU.
- Giúp giảm đáng kể thời gian xử lý so với thực hiện tuần tự.
- Giải pháp song song hóa hiệu quả và dễ mở rộng.

D. Tiềm năng tăng tốc và kết luận

Tiềm năng tăng tốc:

- **Sử dụng thuật toán Fast Doubling** giúp giảm độ phức tạp từ O(n) xuống **O(log n)** cho mỗi truy vấn, điều này đặc biệt hiệu quả với các số lớn.

- Việc **phân chia dữ liệu đầu vào thành các batch nhỏ** và xử lý song song bằng ProcessPoolExecutor giúp tận dụng **đa lõi CPU**, nhờ đó:
    - Giảm thời gian thực hiện so với cách xử lý tuần tự.
    - Cân bằng tải giữa các tiến trình nhờ chia đều batch.
    - Mỗi tiến trình độc lập, không bị phụ thuộc vào tiến trình khác, tránh tranh chấp tài nguyên.
- Ví dụ: Với 1 triệu phần tử cần tính Fibonacci, nếu thực hiện tuần tự có thể mất vài chục giây. Nhưng nhờ song song hóa, thời gian thực tế giảm xuống chỉ còn vài giây (tuỳ CPU).

Kết luận:

Challenge 1 là một ví dụ điển hình cho việc kết hợp giữa **thuật toán tối ưu** (fast doubling) và **xử lý song song** để đạt hiệu suất cao. Việc sử dụng ProcessPoolExecutor đã giúp chương trình mở rộng hiệu quả trên các máy có nhiều nhân xử lý, đồng thời vẫn bảo đảm tính đúng đắn và dễ quản lý.

CHALLENGE 2

A. Mô tả bài toán

Bài toán: Cho hai ma trận vuông A và B có cùng kích thước n×n (với $n \leq 10^4$).

Yêu cầu: Tính ma trận tích C = A × B bằng cách sử dụng thuật toán song song hóa để cải thiện tốc độ xử lý, đặc biệt khi n lớn.

Kết quả: Đầu ra là ma trận Cij có cùng kích thước n×n, trong đó mỗi phần tử Cij là tổng của tích các phần tử dòng i của A với cột j của B.

Ví dụ minh họa:

- Input:

  A = [[1, 2],
  [3, 4]]

  B = [[5, 6],
  [7, 8]]
- Output:

  C = [[19, 22],
  [43, 50]]

B. Ý tưởng và phương pháp song song

B.1. Phân tích đặc trưng bài toán

Bài toán yêu cầu tính tích hai ma trận vuông A và B có kích thước n × n để thu được ma trận kết quả C. Mỗi phần tử C[i][j] được tính bằng công thức: C[i][j] = A[i][0] * B[0][j] + A[i][1] * B[1][j] + ... + A[i][n-1] * B[n-1][j]

Dễ dàng nhận thấy rằng:

- Mỗi phần tử C[i][j] chỉ phụ thuộc vào dòng thứ i của ma trận A và cột thứ j của ma trận B.
- Do đó, việc tính toán mỗi phần tử là **độc lập với các phần tử còn lại**.
- Đặc biệt, **mỗi dòng của ma trận kết quả C có thể được tính độc lập**, vì không có sự phụ thuộc giữa các dòng.

Nhận định này giúp ta có thể song song hóa quá trình tính toán bằng cách chia các dòng của ma trận A ra cho nhiều tiến trình xử lý cùng lúc.

B.2. Chiến lược song song hóa được chọn

Trong bài toán nhân hai ma trận vuông C=A×B, nhóm đã lựa chọn chiến lược song song hóa dựa trên **phân chia theo dòng (row-wise partitioning)** của ma trận A:

- **Ý tưởng chính**: Mỗi dòng i của ma trận kết quả C được tính độc lập từ dòng i của ma trận A và toàn bộ ma trận B. Vì vậy, ta có thể chia ma trận A thành các nhóm dòng (batches), sau đó giao từng nhóm cho một tiến trình tính toán riêng biệt.
- **Mỗi tiến trình đảm nhiệm một batch dòng A**: Sau khi tính xong các dòng tương ứng của ma trận C, tiến trình trả kết quả về để tổng hợp thành ma trận C cuối cùng.
- **Thư viện sử dụng**: Nhóm dùng ProcessPoolExecutor từ thư viện concurrent.futures để tạo pool tiến trình và thực hiện song song việc tính toán từng batch dòng.
- **Lý do chọn phương pháp này**: Đây là cách đơn giản, dễ hiện thực và phù hợp với kiến trúc song song nhiều lõi hiện nay. Ngoài ra, vì các dòng A độc lập nhau nên không cần đồng bộ hóa phức tạp giữa các tiến trình.

B.3. Lý do sử dụng multiprocessing (thay vì threading)

Trong bài toán nhân hai ma trận lớn, khối lượng tính toán tại mỗi dòng của ma trận kết quả rất cao vì mỗi phần tử cần thực hiện nnn phép nhân và n−1 phép cộng. Toàn bộ ma trận C kích thước n×n yêu cầu O(n3) phép tính.

Do đó, nhóm đã lựa chọn **multiprocessing** (thông qua ProcessPoolExecutor) thay vì threading, vì các lý do sau:

- **Tính chất CPU-bound**: Bài toán chủ yếu tiêu tốn tài nguyên CPU chứ không bị chờ I/O. Trong trường hợp này, threading trong Python bị giới hạn bởi GIL (Global Interpreter Lock), khiến các luồng không thực sự chạy song song trên nhiều nhân CPU. Ngược lại, multiprocessing tạo ra các tiến trình riêng biệt nên tránh được GIL, giúp tận dụng toàn bộ năng lực đa lõi.
- **Hiệu suất cao hơn**: Với các phép tính số học cường độ cao trên mảng lớn, multiprocessing thường cho tốc độ nhanh hơn rõ rệt so với threading.

- **Dễ mở rộng**: Khi chia ma trận thành các nhóm dòng, mỗi tiến trình xử lý độc lập trên một batch, giúp dễ dàng mở rộng theo số lõi CPU và thuận tiện trong việc ghép kết quả.

Vì những lý do trên, multiprocessing là lựa chọn phù hợp cho bài toán này để đạt hiệu năng tối ưu trên các hệ thống đa nhân.

C. Phân tích các đoạn code quan trọng

C.1. Hàm multiply_rows(rows_a, matrix_b)

Hàm multiply_rows đảm nhiệm việc nhân một nhóm các hàng của ma trận A với toàn bộ ma trận B để tạo ra một phần kết quả của ma trận tích C. Cụ thể:

```
3   def multiply_rows(rows_a, matrix_b):
4       n = len(matrix_b)
5       result_rows = []
6       for row in rows_a:
7           result_row = []
8           for j in range(n):
9               s = 0
10              for k in range(n):
11                  s += row[k] * matrix_b[k][j]
12              result_row.append(s)
13          result_rows.append(result_row)
14      return result_rows
```

Giải thích:

- **for row in rows_a:**: Lặp qua từng hàng của ma trận A đã được phân chia.
- **for j in range(n):**: Duyệt qua từng cột của ma trận B.
- **for k in range(n): s += row[k] * matrix_b[k][j]**: Tính tích vô hướng giữa hàng row và cột thứ j của B (đây là định nghĩa chuẩn của nhân ma trận).

- **result_rows.append(result_row)**: Lưu kết quả dòng vừa tính vào danh sách kết quả của batch.

Vai trò:

Hàm này thực hiện phần lõi của thuật toán nhân ma trận. Nó bảo đảm mỗi batch (một tập các hàng) được xử lý độc lập để tận dụng khả năng tính toán song song. Điều này giúp giảm đáng kể thời gian xử lý khi làm việc với các ma trận lớn.

C.2. Cách chia ma trận A thành các batch theo hàng

Việc chia ma trận A thành nhiều batch theo hàng là bước quan trọng để phân phối đều công việc cho các tiến trình. Thay vì xử lý toàn bộ ma trận một cách tuần tự, ta chia từng phần nhỏ hơn để các tiến trình con xử lý song song.

```
19      # Chia đều các hàng của A cho các process
20      chunk_size = (n + max_workers - 1) // max_workers
21      batches = [matrix_a[i:i+chunk_size] for i in range(0, n, chunk_size)]
```

Giải thích:

- **chunk_size = (n + max_workers - 1) // max_workers** → Tính toán số hàng tối đa mỗi batch sẽ chứa, bảo đảm chia đều số lượng công việc giữa các tiến trình → Công thức này giúp xử lý tốt cả khi n không chia hết cho max_workers.
- **batches = [...]** → Tạo danh sách các batch, mỗi batch là một tập con liên tiếp các hàng của ma trận A → Ví dụ: Nếu ma trận A có 8 hàng và dùng 4 tiến trình, mỗi batch sẽ gồm 2 hàng.

Vai trò:

- Mỗi tiến trình sẽ xử lý độc lập một số hàng của A.
- Tránh tình trạng trùng lặp hoặc tranh chấp dữ liệu khi xử lý song song.

- Giữ cho kích thước batch tương đối nhỏ, giúp tận dụng hiệu quả tài nguyên CPU và giảm chi phí thời gian chờ.

C.3. Sử dụng ProcessPoolExecutor để song song hóa xử lý

Sau khi đã chia ma trận A thành các batch theo hàng, nhóm sử dụng **ProcessPoolExecutor** từ thư viện concurrent.futures để phân phối công việc cho các tiến trình khác nhau xử lý song song.

```
24      with ProcessPoolExecutor(max_workers=max_workers) as executor:
25          futures = [executor.submit(multiply_rows, batch, matrix_b) for batch in batches]
26          for f in futures:
27              results.extend(f.result())
28      return results
```

Giải thích:

- **ProcessPoolExecutor**: Dùng để tạo một pool gồm nhiều tiến trình (process), mỗi tiến trình có thể thực hiện xử lý độc lập. Phù hợp cho các tác vụ tính toán nặng như nhân ma trận.
- **executor.submit(...)**: Gửi công việc multiply_rows(batch, matrix_b) tới một tiến trình trong pool. Mỗi batch sẽ được xử lý độc lập.
- **futures**: Danh sách các "tác vụ đang chạy" (future tasks), tương ứng với mỗi batch đã gửi đi.
- **f.result()**: Lấy kết quả từ tiến trình sau khi hoàn tất. Kết quả là các hàng đã được nhân đúng của ma trận C.
- **results.extend(...)**: Nối các hàng kết quả từ các batch lại để tạo thành ma trận C hoàn chỉnh.

Tại sao chọn ProcessPoolExecutor thay vì ThreadPoolExecutor?

- Vì nhân ma trận là tác vụ **CPU-bound** (nặng về tính toán), nên cần tận dụng các **lõi CPU riêng biệt**.

- ThreadPoolExecutor không hiệu quả trong trường hợp này do giới hạn của Global Interpreter Lock (GIL) trong Python.

C.4. Ghép kết quả từ các batch lại thành ma trận C cuối cùng

Sau khi các batch được xử lý song song thông qua ProcessPoolExecutor, mỗi batch trả về một danh sách các dòng kết quả tương ứng với phần của ma trận C. Nhóm sử dụng vòng lặp để **nối các dòng này lại theo đúng thứ tự ban đầu**, tạo thành ma trận C hoàn chỉnh.

```
26          for f in futures:
27              results.extend(f.result())
```

Giải thích:

- **f.result()**: Trả về danh sách các dòng đã tính xong của một batch. Các dòng này vẫn giữ nguyên thứ tự ban đầu trong batch.
- **results.extend(...)**: Dùng để nối kết quả từ từng batch vào danh sách results tổng. Vì các batch đã được tạo theo thứ tự ban đầu của ma trận A, việc nối liên tiếp này sẽ bảo đảm đúng thứ tự các dòng trong ma trận C.
- **Kết quả cuối cùng**: Biến results chứa toàn bộ ma trận C sau khi nhân, được trả về từ hàm MAIN.

D. Tiềm năng tăng tốc và kết luận

Tiềm năng tăng tốc

Việc song song hóa nhân ma trận mang lại tiềm năng tăng tốc rõ rệt, đặc biệt khi kích thước ma trận lớn (ví dụ $n \geq 1000$). Trong phương pháp nhóm sử dụng:

- Các dòng của ma trận A được **chia đều theo batch**, giúp tận dụng nhiều nhân CPU cùng lúc.

- Mỗi tiến trình hoạt động **độc lập** và không có tranh chấp tài nguyên, vì các dòng không phụ thuộc nhau.
- Với 4 tiến trình, lý tưởng có thể giảm thời gian tính toán gần 4 lần so với xử lý tuần tự (tùy thuộc CPU và kích thước dữ liệu).

Tuy nhiên, hiệu suất thực tế còn phụ thuộc vào:

- Số nhân thực tế của CPU.
- Chi phí tạo tiến trình và truyền dữ liệu giữa các tiến trình.
- Kích thước batch đủ lớn để thời gian xử lý > chi phí khởi tạo.

Kết luận

Challenge này là ví dụ điển hình của một bài toán có khả năng **song song hóa tốt** nhờ tính chất độc lập giữa các dòng của ma trận. Việc sử dụng ProcessPoolExecutor để xử lý từng batch dòng A giúp tăng tốc đáng kể, đồng thời dễ dàng mở rộng cho các hệ thống nhiều nhân hơn.

Nhóm đã triển khai thành công giải pháp và kiểm tra đầu ra đúng với kỳ vọng. Đây là cách tiếp cận hiệu quả để giải quyết các bài toán xử lý dữ liệu ma trận quy mô lớn.

CHALLENGE 3

A. Mô tả bài toán

Bài toán: Cho một mảng số nguyên A có N phần tử ($N \leq 10^7$). Nhiệm vụ là sắp xếp mảng A tăng dần.

Yêu cầu: Hiện thực một chương trình song song hóa việc sắp xếp để tăng hiệu suất so với phương pháp tuần tự.

Kết quả: Đầu ra là một mảng được sắp xếp đúng thứ tự từ nhỏ đến lớn.

Ví dụ minh họa: Input: A = [4, 1, 3, 2]; Output: [1, 2, 3, 4]

Thách thức: Dữ liệu rất lớn (có thể tới hàng chục triệu phần tử), khiến thuật toán tuần tự tốn thời gian. Cần chọn chiến lược song song phù hợp để tránh quá tải bộ nhớ và giảm chi phí gộp kết quả.

B. Ý tưởng và phương pháp song song

B.1. Phân mảnh mảng đầu vào thành các đoạn nhỏ (chunks)

Để có thể xử lý song song việc sắp xếp mảng có kích thước lớn, chương trình thực hiện chia mảng đầu vào arrayA thành 4 đoạn nhỏ gần bằng nhau. Việc chia này được thực hiện dựa trên công thức: chunk_size = (n + num_chunks - 1) // num_chunks

Trong đó n là số phần tử của mảng và num_chunks = 4 là số đoạn mong muốn. Công thức này bảo đảm các đoạn sẽ có kích thước gần đều nhau và không bị mất dữ liệu.

Việc phân chia này là bước chuẩn bị rất quan trọng để bảo đảm các đoạn có thể được sắp xếp song song một cách độc lập, từ đó tối ưu hiệu suất xử lý bằng cách tận dụng nhiều lõi CPU.

B.2. Chiến lược song song hóa được chọn

Để tăng tốc quá trình sắp xếp một mảng rất lớn, chương trình sử dụng chiến lược chia để trị (divide and conquer) kết hợp với song song hóa theo từng đoạn. Cụ thể, mảng đầu vào được chia thành 4 phần bằng nhau, sau đó mỗi phần được sắp xếp độc lập bằng một tiến trình riêng biệt. Các tiến trình này chạy song song bằng cách sử dụng ProcessPoolExecutor.

Chiến lược này giúp giảm đáng kể thời gian xử lý so với việc sắp xếp tuần tự toàn bộ mảng, đặc biệt khi mảng có kích thước lớn (hàng triệu phần tử). Sau khi sắp xếp xong từng phần, các mảng con sẽ được gộp lại theo từng cặp để tạo thành mảng kết quả cuối cùng đã được sắp xếp hoàn chỉnh.

B.3. Lý do sử dụng ProcessPoolExecutor

Trong bài toán này, quá trình sắp xếp các phần của mảng là một tác vụ tính toán nặng (CPU-bound). Nếu sử dụng ThreadPoolExecutor, hiệu suất có thể bị giới hạn bởi GIL (Global Interpreter Lock) của Python, khiến các luồng không thực sự chạy song song trên nhiều lõi CPU.

Do đó, ProcessPoolExecutor được chọn để tạo ra các tiến trình riêng biệt, mỗi tiến trình có bộ nhớ và tài nguyên riêng, cho phép tận dụng tối đa nhiều lõi CPU thực sự. Điều này giúp cải thiện đáng kể hiệu suất sắp xếp, đặc biệt là với các mảng lớn.

B.4. Gộp (merge) kết quả

Sau khi mỗi tiến trình hoàn tất việc sắp xếp một phần của mảng (chunk), kết quả là nhiều mảng con đã được sắp xếp (sorted chunks). Tuy nhiên, để tạo thành một mảng duy nhất được sắp xếp hoàn chỉnh, ta cần thực hiện bước hợp nhất (merge).

Giải pháp sử dụng hàm merge_two(a, b) để gộp tuần tự từng cặp mảng con. Cụ thể:

1. Gộp mảng 1 và mảng 2 thành merged_1_2.
2. Gộp merged_1_2 với mảng 3 thành merged_1_2_3.
3. Gộp tiếp với mảng 4 để tạo ra mảng kết quả cuối cùng.

Quá trình này tương đương với kỹ thuật merge trong thuật toán merge sort. Vì số lượng mảng con là nhỏ (4), việc gộp tuần tự đơn giản và hiệu quả, không cần thiết song song hóa bước này.

C. Phân tích các đoạn code quan trọng

C.1. Hàm merge_two(a, b)

Hàm merge_two là một thành phần then chốt trong quá trình hợp nhất các mảng con đã được sắp xếp độc lập trước đó. Mục tiêu của hàm này là **kết hợp hai mảng đã sắp xếp thành một mảng mới, vẫn giữ nguyên thứ tự tăng dần**.

Thuật toán sử dụng kỹ thuật "merge" kinh điển như trong thuật toán Merge Sort – với hai con trỏ i, j trỏ vào đầu hai mảng a và b. Ở mỗi bước, phần tử nhỏ hơn sẽ được chọn và đưa vào mảng merged. Khi một trong hai mảng hết phần tử, phần còn lại của mảng kia sẽ được nối trực tiếp vào kết quả.

```
3   def merge_two(a, b):
4       i, j = 0, 0
5       merged = []
6       while i < len(a) and j < len(b):
7           if a[i] < b[j]:
8               merged.append(a[i])
9               i += 1
10          else:
11              merged.append(b[j])
12              j += 1
13      merged.extend(a[i:])
14      merged.extend(b[j:])
15      return merged
```

Giải thích:

- i, j: là các chỉ số duyệt lần lượt mảng a và b.
- merged: là mảng kết quả.
- while: duyệt đến khi một trong hai mảng hết phần tử.
- merged.extend(...): nối phần còn lại của mảng chưa duyệt hết vào kết quả.
- Hàm trả về mảng merged đã được sắp xếp đúng thứ tự.

Hàm này bảo đảm độ phức tạp thời gian là **O(n + m)**, trong đó n và m là độ dài của hai mảng đầu vào – rất hiệu quả cho việc kết hợp dữ liệu đã được sắp xếp song song.

C.2. Cách chia mảng thành các chunks nhỏ

Để tận dụng khả năng xử lý song song, mảng đầu vào cần được chia nhỏ thành nhiều phần (gọi là **chunks**), mỗi phần có thể được xử lý độc lập. Điều này phù hợp với mô hình song song dữ liệu, trong đó cùng một thao tác (sắp xếp) được áp dụng cho các phần khác nhau của dữ liệu.

Trong bài toán này, mảng đầu vào arr được chia thành **4 chunks**. Việc chia đều các phần tử giúp cân bằng khối lượng công việc giữa các tiến trình.

```
21      num_chunks = 4
22      chunk_size = (n + num_chunks - 1) // num_chunks
23      chunks = [arr[i:i+chunk_size] for i in range(0, n, chunk_size)]
```

Giải thích:

- num_chunks = 4: Số lượng phần cần chia, tương ứng với số tiến trình sẽ dùng.
- chunk_size: Tính kích thước mỗi chunk sao cho chia đều và không bỏ sót phần tử.
- Dòng cuối cùng sử dụng list comprehension để tạo ra danh sách các chunks từ arr.

Việc chia mảng như vậy giúp việc **sắp xếp song song** diễn ra hiệu quả hơn, vì mỗi tiến trình sẽ xử lý một lượng dữ liệu gần bằng nhau, tránh bị thừa hoặc thiếu tải.

C.3. Sử dụng ProcessPoolExecutor để sắp xếp song song

Sau khi chia mảng đầu vào thành các đoạn nhỏ (chunks), mỗi đoạn có thể được sắp xếp độc lập. Để thực hiện điều này song song, chương trình sử dụng ProcessPoolExecutor, một công cụ tiện lợi trong Python để quản lý nhiều tiến trình (processes) chạy song song.

```
24      with ProcessPoolExecutor(max_workers=num_chunks) as executor:
25          sorted_chunks = list(executor.map(sorted, chunks))
```

Giải thích:

- ProcessPoolExecutor(max_workers=num_chunks) khởi tạo một pool gồm 4 tiến trình để thực hiện song song việc sắp xếp các chunk.
- executor.map(sorted, chunks) áp dụng hàm sorted() cho từng chunk một cách song song.
- Kết quả là danh sách sorted_chunks, trong đó mỗi phần tử là một chunk đã được sắp xếp.

Lợi ích:

- **Rút ngắn thời gian xử lý**: Thay vì sắp xếp toàn bộ mảng tuần tự, 4 tiến trình có thể sắp xếp các phần khác nhau cùng lúc.
- **Khai thác tối đa CPU đa nhân**: Đặc biệt hữu ích khi làm việc với mảng có kích thước lớn.

C.4. Gộp tuần tự kết quả từ các chunk đã được sắp xếp

Sau khi các chunk được sắp xếp song song, chúng cần được hợp nhất (merge) thành một mảng cuối cùng có thứ tự tăng dần. Vì mỗi chunk đã được sắp xếp trước, việc gộp lại chỉ còn là bài toán merge nhiều mảng đã sắp.

```
26      # Merge 4 mảng đã sort
27      merged = merge_two(sorted_chunks[0], sorted_chunks[1])
28      merged = merge_two(merged, sorted_chunks[2])
29      merged = merge_two(merged, sorted_chunks[3])
30      return merged
```

Giải thích:

- Hàm merge_two(a, b) thực hiện thao tác merge hai mảng đã sắp xếp a và b thành một mảng mới cũng được sắp xếp.
- Do số lượng chunk là 4, quá trình ghép được thực hiện 3 lần để kết hợp tất cả:
    - Bước 1: merge chunk 0 và chunk 1.

  - Bước 2: merge kết quả với chunk 2.
  - Bước 3: merge kết quả với chunk 3.
- Việc merge này không sử dụng thư viện ngoài, chỉ cần thao tác tuần tự hai con trỏ.

Vai trò:

- **Tối ưu hóa bộ nhớ**: Không cần giữ toàn bộ mảng trong bộ nhớ khi merge từng phần.
- **Bảo đảm thứ tự đúng**: Kết quả cuối cùng là một mảng hoàn chỉnh đã được sắp xếp tăng dần.

D. Tiềm năng tăng tốc và kết luận

Tiềm năng tăng tốc:

- **Tận dụng đa nhân CPU**: Việc chia mảng lớn thành 4 chunk và sắp xếp đồng thời giúp tận dụng được tối đa sức mạnh xử lý của nhiều nhân CPU. Thay vì phải sắp xếp toàn bộ mảng theo cách tuần tự với độ phức tạp O(nlogn), ta có thể xử lý từng phần nhỏ hơn cùng lúc, giảm đáng kể thời gian chạy thực tế.
- **Giảm thời gian nhờ phân mảnh hợp lý**: Sử dụng ProcessPoolExecutor để xử lý các chunk lớn trong các tiến trình riêng biệt cho phép hệ thống sử dụng nhiều core hiệu quả hơn so với threading (vốn bị giới hạn bởi Global Interpreter Lock của Python).
- **Chiến lược merge đơn giản**: Do chỉ có 4 chunk nên quá trình gộp các mảng đã sắp xếp chỉ cần 3 bước, giúp tiết kiệm chi phí merge và giữ cho thuật toán đơn giản, dễ kiểm soát.

Giới hạn:

- **Merge không song song**: Quá trình gộp 4 mảng lại được thực hiện tuần tự, đây là một điểm nghẽn tiềm năng nếu số lượng chunk tăng lên nhiều hơn.
- **Chỉ chia thành 4 phần**: Với mảng cực lớn hoặc hệ thống nhiều hơn 4 core, cần điều chỉnh num_chunks để tận dụng tối đa tài nguyên sẵn có.
- **Không xử lý ngoại lệ hay mảng rỗng rõ ràng trong phần merge**: Cần cải thiện để bảo đảm tính ổn định hơn.

Kết luận:

Bài toán sắp xếp song song được giải quyết hiệu quả bằng cách chia nhỏ mảng đầu vào, sắp xếp từng phần độc lập, sau đó gộp lại. Cách tiếp cận này phù hợp cho xử lý dữ liệu lớn, cho thấy rõ lợi ích của tính toán song song. Việc sử dụng ProcessPoolExecutor giúp giảm thời gian chạy mà không làm phức tạp hóa mã nguồn, đồng thời duy trì được độ chính xác và tính ổn định trong kết quả đầu ra.

CHALLENGE 4

A. Mô tả bài toán

Bài toán: Cho một mảng số nguyên lớn A được lưu trong một file đầu vào. Người dùng cần tìm vị trí đầu tiên của một giá trị nguyên key trong mảng đó.

Yêu cầu: Thiết kế và hiện thực một thuật toán tìm kiếm song song để tăng tốc độ truy xuất trong mảng lớn. Kết quả trả về là chỉ số đầu tiên (index) mà giá trị key xuất hiện trong mảng (nếu có), hoặc -1 nếu không tìm thấy.

Ràng buộc: Mảng có thể rất lớn (lên đến hàng triệu phần tử), nên tìm kiếm tuần tự có thể chậm. Không được sử dụng thư viện bị cấm như sys hoặc heapq. Có thể sử dụng threading hoặc ThreadPoolExecutor.

Kết quả: Với input: array = [10, 20, 30, 40, 50] và key = 30, chương trình sẽ trả về 2 vì 30 xuất hiện ở chỉ số thứ 2. Với input không có giá trị key, chương trình sẽ trả về -1.

Ý nghĩa: Bài toán minh họa rõ ràng việc tăng tốc độ truy xuất dữ liệu lớn thông qua lập trình song song – một trong những ứng dụng quan trọng trong xử lý dữ liệu, tìm kiếm, hệ thống thời gian thực và máy chủ.

B. Ý tưởng và phương pháp song song

Chia mảng đầu vào thành các đoạn nhỏ (chunks)

Sau khi đọc toàn bộ mảng từ file vào biến numbers, mảng được chia thành **các đoạn con** đều nhau tùy theo số lượng luồng (num_threads). Mỗi đoạn sẽ chứa một phần dữ liệu độc lập để xử lý song song, giúp tận dụng hiệu quả các nhân CPU.

Chiến lược song song hóa được chọn:

Chương trình cung cấp **hai lựa chọn** để thực hiện tìm kiếm song song:

- **Cách 1: Dùng threading.Thread thủ công** – các thread được tạo bằng tay và mỗi thread xử lý một đoạn của mảng.
- **Cách 2: Dùng ThreadPoolExecutor** – API tiện lợi hơn để quản lý luồng, giúp dễ dàng mở rộng hoặc kiểm soát số lượng luồng tối đa.

Trong hàm MAIN, chương trình ưu tiên sử dụng phương pháp thứ nhất (parallel_search_with_threads) vì nó kiểm soát rõ ràng và phù hợp với yêu cầu của đề bài.

Lý do sử dụng Thread thay vì Process:

- Vì thao tác tìm kiếm chỉ đọc dữ liệu, không yêu cầu CPU tính toán nặng → phù hợp với **IO-bound task**.

- Dùng threading giúp khởi tạo nhanh hơn, ít chi phí khởi động hơn multiprocessing.
- Ngoài ra, do Python bị giới hạn bởi GIL, threading vẫn là lựa chọn hợp lý cho các bài toán như tìm kiếm tuyến tính.

Tìm vị trí đầu tiên (index nhỏ nhất):

Sau khi tất cả các luồng hoàn thành, kết quả tìm thấy từ từng luồng được lưu vào result_dict. Cuối cùng, chương trình **lọc ra các chỉ số hợp lệ** (khác -1) và trả về **chỉ số nhỏ nhất**, bảo đảm kết quả đúng với yêu cầu "vị trí đầu tiên". Nếu không có kết quả nào được tìm thấy, trả về -1.

C. Phân tích các đoạn code quan trọng

C.1. Đọc dữ liệu từ file

Trong bài toán này, dữ liệu đầu vào là một tệp văn bản chứa nhiều số nguyên. Để xử lý dữ liệu, ta cần viết một hàm chuyên trách đọc toàn bộ các số này và đưa chúng vào một mảng để xử lý tiếp theo.

```
4   def read_numbers_from_file(filename):
5       """Đọc tất cả số nguyên từ file và trả về dưới dạng mảng"""
6       numbers = []
7       try:
8           with open(filename, 'r') as file:
9               for line in file:
10                  line_numbers = line.strip().split()
11                  for num_str in line_numbers:
12                      if num_str:
13                          numbers.append(int(num_str))
14          return numbers
15      except (FileNotFoundError, ValueError):
16          return []
```

Phân tích chi tiết:

- **with open(filename, 'r') as file**: Mở file đầu vào ở chế độ đọc. Sử dụng cú pháp with giúp tự động đóng file sau khi đọc xong, tránh rò rỉ tài nguyên.
- **for line in file**: Đọc từng dòng trong file. Điều này bảo đảm file lớn vẫn có thể xử lý tuần tự từng dòng mà không cần đọc toàn bộ vào bộ nhớ cùng lúc.
- **line.strip().split()**: Loại bỏ khoảng trắng đầu/cuối dòng rồi tách thành các phần tử (các số nguyên dưới dạng chuỗi) theo khoảng trắng.
- **int(num_str)**: Ép kiểu từng phần tử thành số nguyên rồi thêm vào danh sách numbers.
- **Xử lý ngoại lệ**:
    - **FileNotFoundError**: Nếu file không tồn tại.
    - **ValueError**: Nếu có lỗi khi ép kiểu chuỗi thành số nguyên.
    - Nếu gặp lỗi, hàm sẽ trả về danh sách rỗng để không làm chương trình bị crash.

Vai trò: Hàm này đóng vai trò nền tảng cho toàn bộ chương trình vì mọi thao tác tìm kiếm sau đó đều phụ thuộc vào dữ liệu được trích xuất đúng định dạng từ file. Đây là bước đầu tiên và bắt buộc phải chính xác để bảo đảm chương trình hoạt động ổn định.

C.2. Hàm tìm kiếm tuần tự (so sánh với song song)

Đây là một trong hai chiến lược chính được cài đặt để tìm kiếm phần tử trong một mảng số nguyên. Cách tiếp cận này sử dụng thư viện threading để tạo các luồng (threads) và chia công việc tìm kiếm cho nhiều luồng thực hiện song song.

```python
def parallel_search_with_threads(arr, target, num_threads=4):
    """Tìm kiếm song song bằng thread thủ công"""
    if not arr:
        return -1

    n = len(arr)
    chunk_size = n // num_threads
    threads = []
    result_dict = {}

    for i in range(num_threads):
        start_idx = i * chunk_size
        end_idx = n if i == num_threads - 1 else (i + 1) * chunk_size
        thread = threading.Thread(
            target=parallel_search_chunk,
            args=(arr, target, start_idx, end_idx, result_dict, i)
        )
        threads.append(thread)
        thread.start()

    for thread in threads:
        thread.join()

    found_indices = [idx for idx in result_dict.values() if idx != -1]
    return min(found_indices) if found_indices else -1
```

Phân tích chi tiết:

- **Chia mảng thành các phần nhỏ:** Mảng arr được chia thành num_threads phần (mỗi phần gọi là một *chunk*). Mỗi thread phụ trách một đoạn con của mảng.
- **Tạo các luồng:** Sử dụng threading.Thread để tạo từng luồng mới, mỗi luồng thực thi hàm parallel_search_chunk và được truyền các chỉ số đầu và cuối để biết vùng dữ liệu cần tìm.
- **Kết quả lưu vào result_dict:** Vì các luồng chạy đồng thời nên để ghi lại kết quả tìm được từ mỗi luồng, ta dùng một dictionary result_dict với thread_id làm key. Mỗi luồng sẽ lưu vào key của nó giá trị là vị trí tìm thấy hoặc -1.

- **Đồng bộ các luồng:** Dùng thread.join() để bảo đảm chương trình chính chỉ tiếp tục sau khi tất cả các luồng đã kết thúc.
- **Tìm ra kết quả cuối cùng:** Lọc ra các chỉ số hợp lệ (khác -1) rồi trả về chỉ số nhỏ nhất – chính là vị trí đầu tiên tìm thấy phần tử.

Ý nghĩa và vai trò: Hàm này thể hiện cách triển khai song song hóa thủ công bằng luồng trong Python mà không dựa vào thư viện cao cấp. Nó phù hợp với các bài toán nhẹ, đơn giản như tìm kiếm tuyến tính, và đặc biệt hiệu quả khi khối lượng công việc không quá nặng, giúp tận dụng tốt khả năng của CPU.

C.3. Tìm kiếm song song bằng threading.Thread

Hàm này là một phiên bản khác của tìm kiếm song song, sử dụng lớp ThreadPoolExecutor từ thư viện concurrent.futures. Đây là một cách tiếp cận hiện đại, giúp quản lý và sử dụng các luồng (threads) đơn giản và hiệu quả hơn so với việc khởi tạo thủ công bằng threading.Thread.

```python
def parallel_search_with_executor(arr, target, num_workers=4):
    if not arr:
        return -1

    n = len(arr)
    chunk_size = n // num_workers

    def search_chunk(start_idx, end_idx):
        for i in range(start_idx, min(end_idx, n)):
            if arr[i] == target:
                return i
        return -1

    with ThreadPoolExecutor(max_workers=num_workers) as executor:
        futures = []
        for i in range(num_workers):
            start_idx = i * chunk_size
            end_idx = n if i == num_workers - 1 else (i + 1) * chunk_size
            future = executor.submit(search_chunk, start_idx, end_idx)
            futures.append(future)

        results = []
        for future in futures:
            result = future.result()
            if result != -1:
                results.append(result)

        return min(results) if results else -1
```

Phân tích chi tiết:

- **Chia mảng thành các vùng nhỏ:** Tương tự như cách làm ở hàm trước, mảng arr được chia thành các đoạn nhỏ (chunk) dựa trên số lượng num_workers.
- **Định nghĩa hàm con search_chunk:** Đây là hàm tìm kiếm tuyến tính trong một đoạn con của mảng. Nó trả về chỉ số đầu tiên tìm thấy hoặc -1.

- **Sử dụng ThreadPoolExecutor:** Với max_workers là số lượng luồng tối đa, ta khởi tạo một executor. Sau đó, dùng executor.submit(...) để đưa các tác vụ tìm kiếm vào hàng đợi thực thi song song.
- **Thu thập kết quả:** Tập hợp các kết quả từ các Future và lọc ra các chỉ số hợp lệ, trả về chỉ số nhỏ nhất nếu tìm thấy.

Lợi ích của ThreadPoolExecutor:

- **Dễ đọc, dễ bảo trì:** Không cần tự tạo thread, không phải lo quản lý vòng đời của các luồng.
- **Tối ưu hiệu năng:** Executor tự điều phối luồng hiệu quả, thích hợp cho bài toán có khối lượng xử lý nhẹ và I/O-bound như tìm kiếm.
- **Tính mở rộng cao:** Có thể dễ dàng nâng cấp thành các hệ thống xử lý song song phức tạp hơn mà không cần viết lại nhiều code.

C.4. Tìm kiếm song song bằng ThreadPoolExecutor

Hàm MAIN đóng vai trò là điểm khởi đầu của chương trình, có nhiệm vụ đọc dữ liệu đầu vào từ file và thực hiện tìm kiếm giá trị được chỉ định bằng phương pháp song song đã chọn.

```
89  def MAIN(input_file_path, key_value):
90      """Hàm chính của bài tập - tìm kiếm số trong file"""
91      numbers = read_numbers_from_file(input_file_path)
92      if not numbers:
93          return -1
94      return parallel_search_with_threads(numbers, key_value, num_threads=4)
```

Phân tích chi tiết:

- **Đọc dữ liệu đầu vào:** Dữ liệu được đọc từ file thông qua hàm read_numbers_from_file(), trả về danh sách các số nguyên.
- **Xử lý nếu không có dữ liệu:** Nếu file trống hoặc lỗi, hàm trả về -1 ngay để xử lý an toàn.

- **Gọi phương pháp tìm kiếm:** Hàm gọi parallel_search_with_threads(...) với tham số num_threads=4. Trong trường hợp cần thay đổi phương pháp tìm kiếm (ví dụ dùng parallel_search_with_executor()), chỉ cần đổi tên hàm được gọi tại đây.

Cấu trúc tổng thể của chương trình: Được tổ chức rõ ràng theo từng chức năng riêng biệt:

- Đọc dữ liệu từ file: read_numbers_from_file()
- Tìm kiếm tuần tự (so sánh): sequential_search()
- Tìm kiếm song song thủ công: parallel_search_with_threads()
- Tìm kiếm song song dùng executor: parallel_search_with_executor()
- Tìm kiếm song song từng phần: parallel_search_chunk()
- Điểm bắt đầu chạy: MAIN()

D. Tiềm năng tăng tốc và kết luận

Phân tích hiệu suất thực tế:

- **Tìm kiếm tuần tự** có độ phức tạp là **O(n)**, phải duyệt toàn bộ mảng trong trường hợp xấu nhất.
- Trong phương pháp **song song**, mảng được chia cho 4 luồng. Nếu phần tử cần tìm nằm ở đoạn đầu thì có thể trả kết quả nhanh, **giảm thời gian chờ tổng thể**.
- Tuy nhiên, **do bản chất của tuyến tính**, các luồng vẫn phải hoàn tất để xác định đúng chỉ số nhỏ nhất, trừ khi bổ sung cơ chế dừng sớm (early termination).

Giới hạn của mô hình song song:

- Do Python sử dụng GIL, các tác vụ CPU-bound bị giới hạn hiệu quả với threading, tuy nhiên trong bài này (IO-light), threading vẫn phát huy được hiệu quả nhất định.
- Với số luồng lớn hơn, chi phí tạo và quản lý luồng có thể vượt quá lợi ích mang lại, đặc biệt nếu dữ liệu nhỏ.

Kết luận:

- Bài toán tìm kiếm tuyến tính có thể tận dụng **song song hóa** để cải thiện thời gian phản hồi trong một số trường hợp, đặc biệt khi dữ liệu lớn.
- Việc sử dụng threading.Thread hoặc ThreadPoolExecutor đều phù hợp, với lựa chọn tuỳ thuộc vào yêu cầu cụ thể.
- Đây là ví dụ điển hình cho **song song hóa mức độ thấp (fine-grained parallelism)** trong các bài toán đơn giản, giúp sinh viên hiểu được nguyên lý phân chia công việc và đồng bộ hoá kết quả.

HẾT./.