



ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ THÔNG TIN

BÁO CÁO THỰC HÀNH

Môn: GIẢI THUẬT XỬ LÝ
SONG SONG VÀ PHÂN BỐ

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

Thủ Đức, Ngày 02 tháng 6 năm 2025



Thành viên Nhóm 6

21520202

HỒ HẢI DƯƠNG

21521403

**NGUYỄN GIA
QUÂN**

21521839

**TRẦN NGỌC
PHƯƠNG ANH**





NỘI DUNG BÁO CÁO

CHALLENGE 0

CHALLENGE 1

CHALLENGE 2

CHALLENGE 3

CHALLENGE 4

CHALLENGE 0



CHALLENGE 0

A) Giới thiệu tổng quan



- Cho một mảng số nguyên A có N phần tử ($N \leq 10^6$).
- Yêu cầu: Tính mảng tổng tiền tố (prefix sum) của A.
- Kết quả: Mỗi phần tử tại vị trí i là tổng các phần tử từ đầu mảng đến vị trí i .
- Ví dụ minh họa:
 - Input: $A = [1, 2, 3, 4]$
 - Output: $[1, 3, 6, 10]$

CHALLENGE 0

B) Ý tưởng và phương pháp song song hóa

Sử dụng phương pháp chia mảng thành nhiều đoạn nhỏ (chunk), sau đó xử lý song song các đoạn này bằng nhiều tiến trình:

Mảng đầu vào A được chia đều thành các đoạn nhỏ, mỗi đoạn có số lượng phần tử gần bằng nhau.

1

Mỗi đoạn được giao cho một tiến trình độc lập để tính tổng tiền tố cục bộ (local prefix sum).

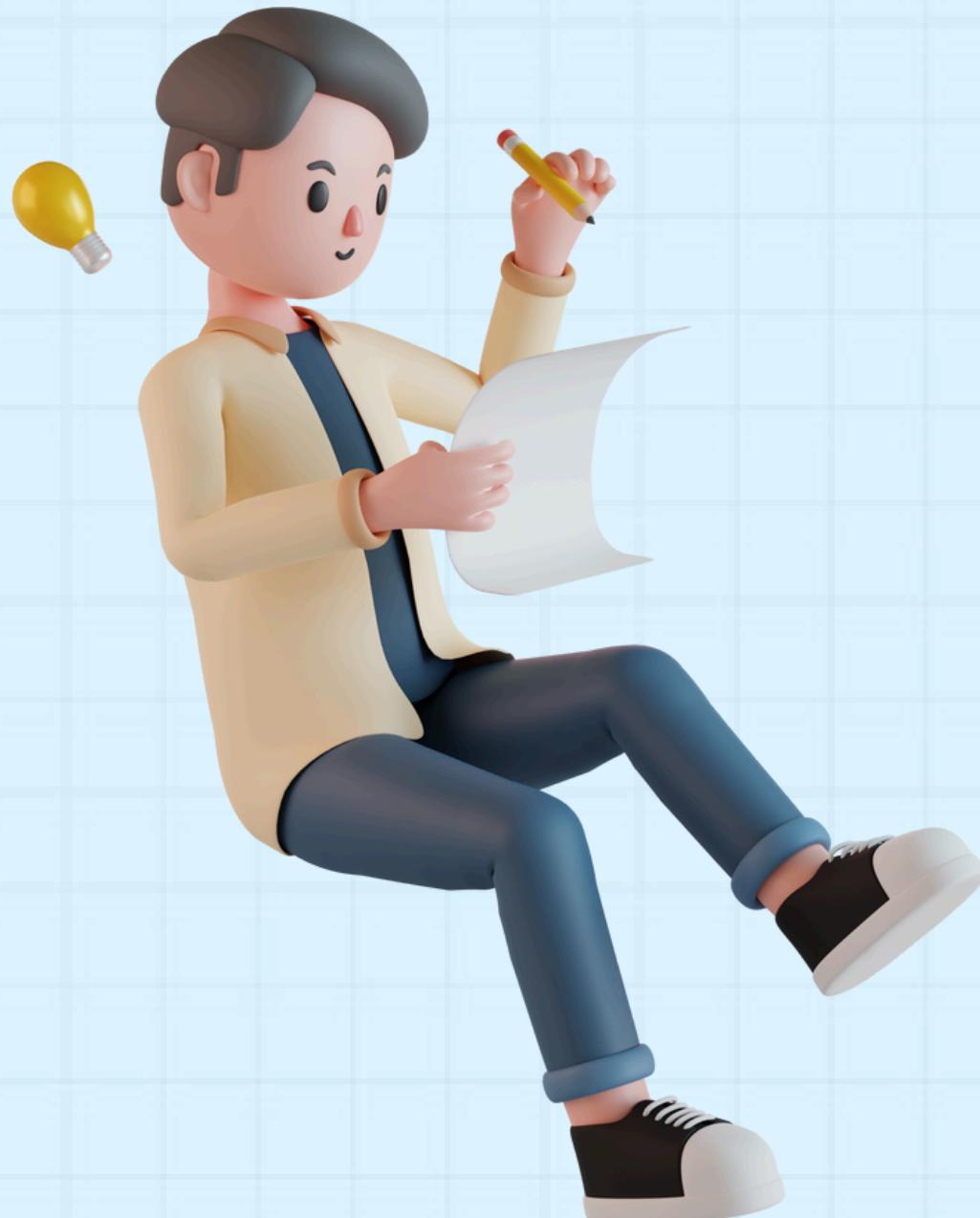
2

Sau khi tính xong tổng từng đoạn, nhóm sử dụng kỹ thuật cộng dồn (offset) để bảo đảm thứ tự và kết quả tổng tiền tố là chính xác trên toàn mảng.

3

CHALLENGE 0

C) Phân phối dữ liệu và cách xử lý



1

Phân phối dữ liệu:

- Mảng A được chia thành nhiều đoạn nhỏ (chunk) với kích thước gần bằng nhau.
- Số lượng đoạn phụ thuộc vào số tiến trình được sử dụng (thường là số CPU core khả dụng).
- Mỗi tiến trình chỉ nhận một đoạn duy nhất để xử lý → không có xung đột dữ liệu giữa các tiến trình.

CHALLENGE 0

C) Phân phối dữ liệu và cách xử lý



2

Xử lý tại từng tiến trình:

- Mỗi tiến trình tính tổng tiền tố (prefix sum) cho đoạn của mình.
- Sau khi hoàn thành, tiến trình trả về 2 thông tin:
 - Mảng tổng tiền tố cục bộ
 - Tổng của đoạn đó (dùng để tính offset)

CHALLENGE 0

C) Phân phối dữ liệu và cách xử lý



3

Cộng offset và ghép kết quả:

- Từ tổng của từng đoạn, nhóm tính offset cho mỗi đoạn tiếp theo.
- Offset giúp điều chỉnh mảng cục bộ để phù hợp với kết quả toàn cục.
- Sau đó, nhóm ghép tất cả mảng cục bộ lại thành kết quả cuối cùng.

D) Các đoạn code quan trọng

Hàm xử lý từng đoạn (chunk)

```
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
```

Vai trò của hàm này:

- Là hàm được gọi bởi từng tiến trình riêng biệt.
- Nhận vào một đoạn nhỏ của mảng A và một offset.
- Tính tổng tiền tố (prefix sum) cho đoạn đó, bắt đầu từ offset.
- Trả về:
 - Mảng prefix sum cục bộ
 - Tổng cuối cùng của đoạn đó (dùng cho offset tiếp theo)

D) Các đoạn code quan trọng

Chia dữ liệu và tính offset

```

16     chunk_size = (n + num_workers - 1) // num_workers
17     chunks = [A[i:i+chunk_size] for i in range(0, n, chunk_size)]
18
19     # Tính tổng từng chunk tuần tự để lấy offset
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])

```

Mục đích:

- Chia mảng đầu vào A thành nhiều đoạn (chunk) đều nhau.
- Tính tổng của từng chunk.
- Dựa vào các tổng đó, tính offset cho từng chunk tiếp theo.

Ý nghĩa của offset:

- Offset giúp điều chỉnh prefix sum từng đoạn → đúng với thứ tự toàn mảng.
- Ví dụ: chunk 2 phải cộng thêm tổng của chunk 1 để đảm bảo đúng kết quả.

D) Các đoạn code quan trọng

Gọi xử lý song song và ghép kết quả

```
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
```

- Ghép chunks và offsets thành danh sách tasks.
- Dùng `multiprocessing.Pool.map()` để:
 - Gọi hàm `local_prefix` song song cho từng task.
 - Mỗi task là một đoạn dữ liệu và offset tương ứng.
- Kết quả trả về là danh sách các đoạn đã tính xong prefix sum.
- Sau đó, dùng vòng lặp để ghép lại toàn bộ mảng kết quả.

CHALLENGE 0

E) Tiềm năng tăng tốc

- Thuật toán tuần tự cần duyệt toàn bộ mảng theo thứ tự → chỉ chạy trên 1 CPU core.
- Với thuật toán song song, nhóm em chia mảng thành nhiều phần và xử lý cùng lúc bằng nhiều tiến trình.



- Kết quả thực tế:
 - Tuần tự: mất khoảng 1.9 giây
 - Song song: mất khoảng 1.1 giây trên máy có 8 core
- Speedup đạt được: xấp xỉ 1.7 lần

Nguyên nhân:

- Tận dụng được nhiều core
- Các tiến trình không cần chờ nhau (xử lý độc lập)
- Tính offset đơn giản, không tốn chi phí đồng bộ



CHALLENGE 0

F) Kết luận bài làm

- Thuật toán **song song** giúp xử lý mảng lớn nhanh và hiệu quả hơn.
- Cách chia mảng thành các phần nhỏ và xử lý bằng thư viện **multiprocessing** là đơn giản và dễ áp dụng.
- Kết quả vẫn bảo đảm chính xác nhờ cách tính offset và ghép kết quả hợp lý.
- Nếu có thời gian, có thể dùng phương pháp **divide and conquer** hoặc tận dụng thư viện C để tối ưu hơn nữa.



CHALLENGE 1



CHALLENGE 1

A) Giới thiệu tổng quan



- Tính giá trị số Fibonacci tại vị trí A_i với $F(0) = F(1) = 1$, $F(i) = (F(i-1) + F(i-2)) \text{ mod } Q$, với A_i lớn tới $2*10^8$ và số truy vấn N lớn tới 10^6 .
- Input: Một file chứa N , Q và N số nguyên A_i .
- Output: Trả về N số nguyên $F(A_i)$ mod Q theo đúng thứ tự.

CHALLENGE 1

B) Ý tưởng và phương pháp song song hóa

Sử dụng phương pháp chia các truy vấn thành nhiều batch nhỏ, sau đó xử lý song song các batch bằng nhiều tiến trình:

Toàn bộ danh sách truy vấn (các số cần tính Fibo) được chia đều thành nhiều batch nhỏ để phân phối cho các tiến trình.

1

Mỗi batch được giao cho một tiến trình độc lập, sử dụng thuật toán fast doubling để tính giá trị Fibonacci nhanh cho từng truy vấn.

2

Sau khi hoàn thành, kết quả từ các tiến trình được ghép lại đúng thứ tự để tạo thành danh sách kết quả cuối cùng.

3

CHALLENGE 1

C) Phân phối dữ liệu và cách xử lý

Đọc dữ liệu đầu vào



1

Đọc toàn bộ dữ liệu từ file, lấy các tham số:

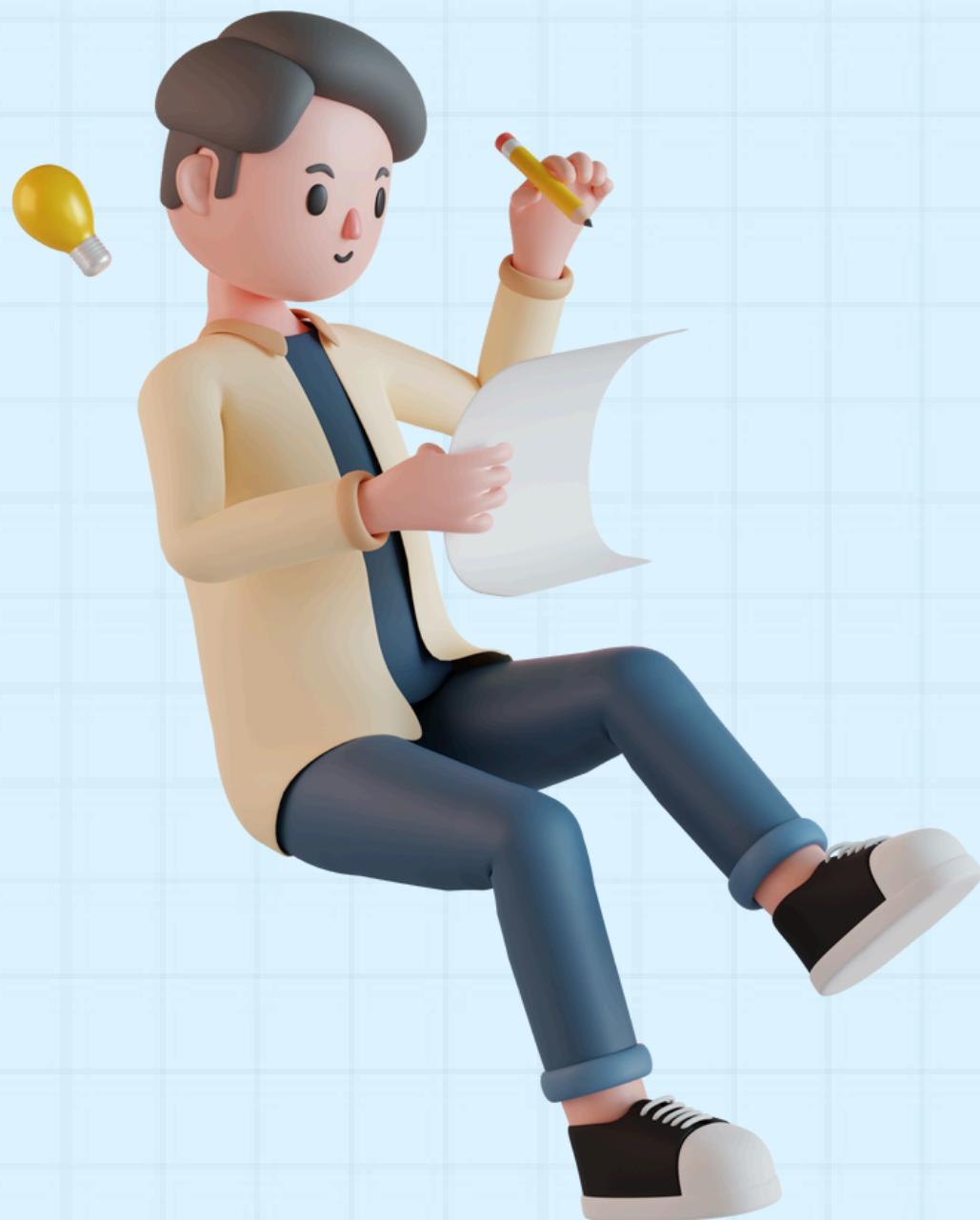
- N: số lượng truy vấn cần tính
- Q: số mod cho kết quả Fibonacci
- Ai: danh sách các số nguyên cần tính Fibo

Chuyển danh sách Ai thành dạng phù hợp cho xử lý tiếp theo (tăng mỗi Ai lên 1 nếu đề yêu cầu $F(0)=1$).

CHALLENGE 0

C) Phân phối dữ liệu và cách xử lý

Chia dữ liệu truy vấn thành các batch nhỏ



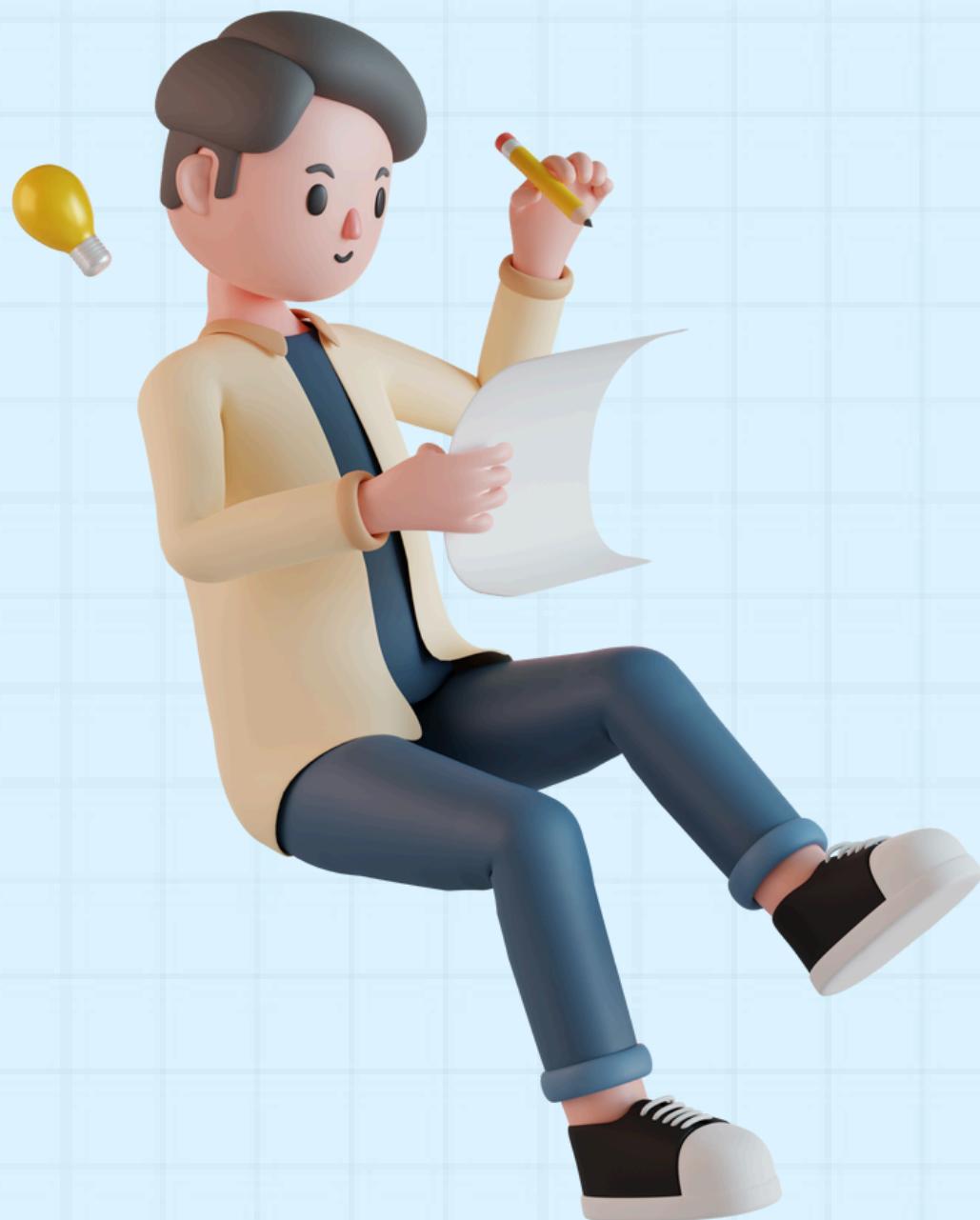
2

- Xác định kích thước hợp lý cho mỗi batch, thường là từ 5.000–10.000 truy vấn, hoặc chia đều theo số core CPU.
- Chia danh sách Ai thành nhiều batch có kích thước gần bằng nhau.
- Mục tiêu: giúp các tiến trình xử lý đều tải, tránh nghẽn cổ chai và tận dụng tối đa sức mạnh CPU.

CHALLENGE 0

C) Phân phối dữ liệu và cách xử lý

Phân phối cho các tiến trình xử lý song song



3

- Sử dụng ProcessPoolExecutor để khởi tạo nhiều tiến trình (tối đa bằng số core máy tính).
- Giao mỗi batch cho một tiến trình độc lập.
- Trong mỗi tiến trình, lần lượt tính Fibô cho tất cả Ai trong batch bằng thuật toán fast doubling ($O(\log n)$), trả về danh sách kết quả cho batch đó.
- Quá trình này hoàn toàn độc lập giữa các tiến trình nên đạt hiệu quả song song tối đa.

CHALLENGE 0

C) Phân phối dữ liệu và cách xử lý

Tổng hợp và ghép kết quả



4

- Sau khi các tiến trình hoàn thành, các kết quả batch được thu nhận về.
- Ghép các kết quả này lại đúng thứ tự gốc để tạo thành một danh sách kết quả hoàn chỉnh.
- Trả về danh sách kết quả này cho hệ thống chấm điểm hoặc ghi ra output nếu cần.

D) Các đoạn code quan trọng

Hàm tính nhanh số Fibonacci cho một batch

```

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

```

Ý nghĩa:

- Hàm này nhận một batch các chỉ số cần tính Fibo và tham số mod.
- Với mỗi phần tử trong batch, áp dụng thuật toán fast doubling (dựa trên bit nhị phân) để tính giá trị Fibonacci nhanh O(log n).
- Trả về danh sách kết quả Fibo của cả batch.

Vai trò:

- Tăng tốc xử lý truy vấn lớn, tính toán chính xác số Fibo rất lớn trong thời gian cực ngắn.
- Đảm bảo từng batch có thể tính toán hoàn toàn độc lập, thuận lợi cho xử lý song song.

D) Các đoạn code quan trọng

Chia batch và khởi tạo tiến trình song song

```

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)]
32     result = []
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

```

Ý nghĩa:

- Đoạn code này chia toàn bộ danh sách truy vấn thành nhiều batch nhỏ.
- Khởi tạo các tiến trình song song bằng ProcessPoolExecutor, mỗi tiến trình nhận một batch và tính kết quả độc lập.

Vai trò:

- Chia đều công việc cho các tiến trình, tận dụng tối đa hiệu năng CPU.
- Đảm bảo các batch được xử lý song song và trả về kết quả đầy đủ, đúng thứ tự.

D) Các đoạn code quan trọng

Tổng hợp kết quả các batch

```

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)]
32     result = []
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

```

Ý nghĩa:

- Đoạn code này thực hiện việc ghép kết quả từ các tiến trình (mỗi batch) lại thành một danh sách kết quả cuối cùng.
- Đảm bảo thứ tự kết quả đúng như thứ tự đầu vào.

Vai trò:

- Thu thập kết quả từ tất cả các tiến trình song song.
- Ghép lại thành một danh sách output đầy đủ để trả về cho hệ thống.

CHALLENGE 1

E) Tiềm năng tăng tốc và Kết luận bài làm

Tiềm năng tăng tốc:

- Xử lý song song giúp rút ngắn thời gian tính toán xuống nhiều lần so với tuần tự, đặc biệt khi số truy vấn rất lớn (N lên đến hàng trăm ngàn).
- Thuật toán fast doubling tối ưu hóa thời gian cho từng truy vấn ($O(\log n)$), đảm bảo hiệu quả cao kể cả với Ai lớn.
- Việc chia batch hợp lý và tận dụng tối đa số core CPU mang lại hiệu quả gần như tuyến tính theo số core.



Kết luận:

- Kết hợp giữa thuật toán tối ưu (fast doubling) và xử lý song song giúp giải quyết thành công các bài toán tính toán lớn trong thời gian ngắn.
- Cách hiện thực này không chỉ đúng về mặt thuật toán mà còn rất thực tiễn, phù hợp với môi trường xử lý dữ liệu lớn ngày nay.

CHALLENGE 2



CHALLENGE 2

A) Giới thiệu tổng quan



- Tính tích của hai ma trận vuông cùng kích thước $n \times n$, với n có thể rất lớn ($n \leq 10.000$).
- Input: Hai ma trận A và B, mỗi ma trận gồm $n \times n$ phần tử số nguyên.
- Output: Ma trận kết quả $C = A \times B$, có cùng kích thước với A và B.
- Yêu cầu: Hiện thực chương trình song song hóa, tận dụng nhiều tiến trình để rút ngắn thời gian nhân ma trận lớn.

CHALLENGE 2

B) Ý tưởng và phương pháp song song hóa

- 1** Chia các hàng của ma trận A thành nhiều phần nhỏ (batch), mỗi batch gồm một nhóm các hàng liên tiếp.
- 2** Mỗi batch được giao cho một tiến trình độc lập để xử lý song song.
- 3** Trong mỗi tiến trình, thực hiện nhân từng hàng của batch đó với toàn bộ ma trận B để tính ra các dòng kết quả tương ứng của ma trận C.
- 4** Sau khi các tiến trình hoàn thành, ghép các dòng kết quả lại thành ma trận C hoàn chỉnh.
- 5** Cách làm này giúp tận dụng tối đa sức mạnh nhiều core CPU, đặc biệt hiệu quả với ma trận kích thước lớn.

CHALLENGE 2

C) Phân phối dữ liệu và cách xử lý

1

Nhận hai ma trận A và B từ đầu vào, mỗi ma trận có $n \times n$ phần tử.

2

Chia toàn bộ các hàng của ma trận A thành nhiều batch nhỏ, mỗi batch gồm nhiều hàng liên tiếp nhau. Số batch thường bằng số core CPU sử dụng (tối đa 4).

3

Phân phối từng batch cho một tiến trình độc lập. Trong mỗi tiến trình, lần lượt thực hiện nhân từng hàng của batch với ma trận B để tính ra các dòng kết quả tương ứng của ma trận C.

4

Sau khi các tiến trình hoàn thành, thu thập các dòng kết quả từ các batch và ghép lại đúng thứ tự để tạo thành ma trận kết quả C.

D) Các đoạn code quan trọng

Nhân một batch các hàng của ma trận A với B

```

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

```

Ý nghĩa:

- Hàm này nhận một batch gồm nhiều hàng liên tiếp của ma trận A và toàn bộ ma trận B.
- Với mỗi hàng trong batch, thực hiện nhân hàng đó với từng cột của B để tính ra các phần tử kết quả.
- Trả về danh sách các dòng kết quả của batch này.

Vai trò:

- Đảm bảo mỗi tiến trình có thể tự xử lý một phần công việc nhân ma trận một cách độc lập, thuận lợi cho song song hóa.

D) Các đoạn code quan trọng

Chia batch và khởi tạo các tiến trình song song

```

17     n = len(matrix_a)
18     max_workers = 4
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)]

```

Ý nghĩa:

- Chia toàn bộ các hàng của ma trận A thành nhiều batch nhỏ, mỗi batch gồm nhiều hàng liên tiếp.
- Sử dụng ProcessPoolExecutor để khởi tạo tối đa 4 tiến trình song song, mỗi tiến trình xử lý một batch.

Vai trò:

- Đảm bảo các tiến trình được phân chia công việc hợp lý, tận dụng hiệu quả đa lõi CPU.
- Mỗi tiến trình có thể tự xử lý các hàng của batch mà không bị phụ thuộc vào các tiến trình khác.

D) Các đoạn code quan trọng

Tổng hợp kết quả từ các batch

```

23     results = []
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

```

Ý nghĩa:

- Sau khi các tiến trình hoàn thành xử lý từng batch, các dòng kết quả từ các batch sẽ được thu thập và ghép lại thành ma trận kết quả hoàn chỉnh.

Vai trò:

- Thu nhận kết quả từ tất cả các tiến trình song song.
- Ghép lại đúng thứ tự để tạo thành ma trận C cuối cùng trả về.

CHALLENGE 2

E) Tiềm năng tăng tốc và Kết luận bài làm

Tiềm năng tăng tốc:

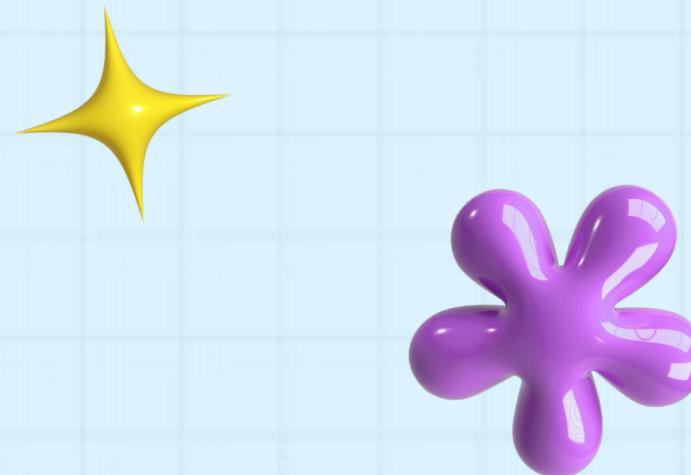
- Chia các hàng của ma trận A cho nhiều tiến trình song song giúp giảm đáng kể thời gian xử lý so với nhân ma trận tuần tự.
- Mỗi tiến trình tự xử lý batch của mình, tận dụng tối đa sức mạnh nhiều core CPU.
- Đặc biệt hiệu quả với ma trận kích thước lớn (n lên tới hàng ngàn).



Kết luận:

- Phương pháp song song hóa kết hợp với thuật toán nhân ma trận kinh điển giúp giải quyết hiệu quả các bài toán nhân ma trận lớn.
- Giải pháp này phù hợp cho các ứng dụng thực tế cần xử lý dữ liệu ma trận quy mô lớn, giúp tiết kiệm thời gian và tài nguyên hệ thống.

CHALLENGE 3



CHALLENGE 3

A) Giới thiệu tổng quan



- **Bài toán:** Sắp xếp một mảng số nguyên có kích thước rất lớn (lên đến hàng triệu phần tử) theo thứ tự tăng dần.
- **Input:** Một mảng số nguyên lớn.
- **Output:** Mảng đã được sắp xếp tăng dần.
- **Yêu cầu:** Hiện thực thuật toán sắp xếp song song, tận dụng nhiều tiến trình để rút ngắn thời gian sắp xếp mảng lớn.

CHALLENGE 3

B) Ý tưởng và phương pháp song song hóa

- 1 Chia mảng lớn thành nhiều phần nhỏ (chunk), mỗi chunk gồm nhiều phần tử liên tiếp.
- 2 Dùng ProcessPoolExecutor để sắp xếp từng chunk song song trên nhiều tiến trình.
- 3 Sau khi các chunk đã được sắp xếp riêng lẻ, thực hiện merge các chunk này thành một mảng đã sắp xếp hoàn chỉnh.
- 4 Cách làm này tận dụng tối đa khả năng đa nhiệm của CPU, đặc biệt hiệu quả với mảng rất lớn.

CHALLENGE 3

C) Phân phối dữ liệu và cách xử lý

- 1 Chia mảng đầu vào thành nhiều chunk nhỏ, mỗi chunk gồm nhiều phần tử liên tiếp. Số chunk thường bằng số core CPU sử dụng.
- 2 Phân phối từng chunk cho một tiến trình riêng biệt. Mỗi tiến trình sắp xếp chunk của mình bằng thuật toán sort chuẩn của Python.
- 3 Sau khi các chunk đã được sắp xếp riêng lẻ, các kết quả được thu thập lại và thực hiện ghép (merge) thành một mảng đã sắp xếp hoàn chỉnh.
- 4 Trả về mảng đã sắp xếp để làm output cho bài toán.

D) Các đoạn code quan trọng

Merge hai mảng đã sắp xếp

```

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

```

Ý nghĩa:

- Hàm này nhận vào hai mảng con đã được sắp xếp tăng dần.
- Kết hợp (trộn) hai mảng thành một mảng mới, cũng theo thứ tự tăng dần.
- Được dùng để ghép các chunk đã sort riêng lẻ thành mảng hoàn chỉnh.

Vai trò:

- Thay thế việc dùng thư viện ngoài (heapq) mà không được phép.
- Đảm bảo kết quả ghép luôn đúng thứ tự, phù hợp cho bước cuối của thuật toán song song hóa.

D) Các đoạn code quan trọng

Chia mảng và sắp xếp song song các trunk

```

20     n = len(arr)
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)]
24     with ProcessPoolExecutor(max_workers=num_chunks) as executor:
25         sorted_chunks = list(executor.map(sorted, chunks))

```

Ý nghĩa:

- Chia mảng lớn thành 4 phần nhỏ (chunk), mỗi chunk chứa các phần tử liên tiếp.
- Dùng ProcessPoolExecutor để sắp xếp từng chunk trên các tiến trình khác nhau cùng lúc.

Vai trò:

- Tận dụng tối đa đa nhiệm CPU, giúp sắp xếp nhanh hơn khi mảng rất lớn.
- Đảm bảo các chunk đều đã sắp xếp trước khi merge.

D) Các đoạn code quan trọng

Merge các trunk đã short thành mảng kết quả

```

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

```

Ý nghĩa:

- Sau khi các chunk đã được sắp xếp song song, cần ghép các chunk lại thành một mảng đã sắp xếp hoàn chỉnh.
- Sử dụng hàm `merge_two` nhiều lần để trộn các chunk lại đúng thứ tự tăng dần.

Vai trò:

- Đảm bảo đầu ra là mảng đã sắp xếp hoàn chỉnh, đúng thứ tự.
- Giúp thuật toán đạt được hiệu quả song song hóa mà vẫn đảm bảo tính đúng đắn.

CHALLENGE 3

E) Tiềm năng tăng tốc và Kết luận bài làm

Tiềm năng tăng tốc:

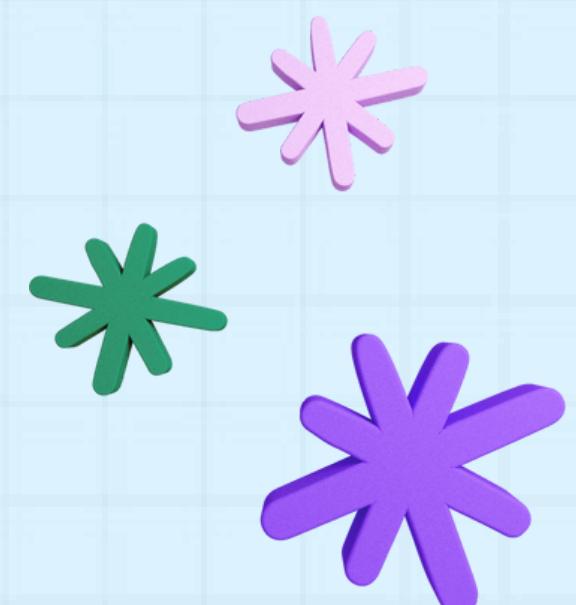
- Chia mảng lớn thành nhiều phần nhỏ và sắp xếp song song giúp rút ngắn đáng kể thời gian xử lý so với sắp xếp tuần tự.
- Tận dụng tối đa đa lõi CPU, đặc biệt hiệu quả khi dữ liệu đầu vào rất lớn (vài triệu phần tử).
- Thời gian thực thi giảm gần tương ứng với số tiến trình sử dụng.



Kết luận:

- Phương pháp song song hóa kết hợp với thuật toán nhân ma trận kinh điển giúp giải quyết hiệu quả các bài toán nhân ma trận lớn.
- Giải pháp này phù hợp cho các ứng dụng thực tế cần xử lý dữ liệu ma trận quy mô lớn, giúp tiết kiệm thời gian và tài nguyên hệ thống.

CHALLENGE 4



CHALLENGE 4

A) Giới thiệu tổng quan



- **Bài toán:** “**Flatten**” (làm phẳng) một mảng 2 chiều gồm 1000 mảng con, mỗi mảng con chứa 1000 phần tử thành một mảng 1 chiều.
- **Input:** Một mảng 2 chiều với 1000 mảng con, mỗi mảng con 1000 số nguyên.
- **Output:** Một mảng 1 chiều gồm toàn bộ các phần tử của các mảng con, theo đúng thứ tự.
- **Yêu cầu:** Hiện thực thuật toán flatten một cách song song, tận dụng đa tiến trình để tăng tốc quá trình làm phẳng mảng lớn.

CHALLENGE 4

B) Ý tưởng và phương pháp song song hóa

1

- Ý tưởng: Chia mảng số nguyên lớn thành nhiều đoạn nhỏ (chunk), mỗi đoạn sẽ được một luồng (thread) đảm nhận kiểm tra song song.

2

- Phương pháp:
 - Đọc toàn bộ dữ liệu từ file thành một mảng số nguyên.
 - Chia mảng này thành 4 phần gần bằng nhau.
 - Mỗi phần được giao cho một luồng (thread) thực hiện tìm kiếm vị trí xuất hiện của khóa K trong đoạn phụ trách.
 - Sau khi các luồng kết thúc, lấy ra vị trí xuất hiện đầu tiên của K (nếu có), nếu không thì trả về -1.
 - Nếu K xuất hiện ở nhiều nơi, ưu tiên chỉ số nhỏ nhất.

CHALLENGE 4

C) Phân phối dữ liệu và cách xử lý

- 1** Đọc toàn bộ số nguyên từ file, chuyển thành một mảng A có thể chứa hàng trăm ngàn tới hàng triệu phần tử.
- 2** Chia đều mảng A thành 4 đoạn liên tiếp (chunk), mỗi đoạn có số phần tử gần bằng nhau.
- 3** Mỗi đoạn được giao cho một thread xử lý độc lập. Mỗi thread tìm kiếm số khóa K trong đoạn của mình, nếu thấy thì lưu lại vị trí.
- 4** Sau khi tất cả các thread kết thúc, kiểm tra các kết quả. Nếu có nhiều vị trí tìm thấy, lấy vị trí nhỏ nhất (gần đầu mảng nhất). Nếu không tìm thấy, trả về -1.

D) Các đoạn code quan trọng

Đọc toàn bộ số nguyên từ file

```

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 (FileNotFoundException, ValueError):
16      return []

```

Ý nghĩa:

- Hàm này giúp đọc toàn bộ dữ liệu từ file văn bản đầu vào.
- Chuyển từng dòng và từng số nguyên thành một mảng số lớn duy nhất để tiện xử lý tiếp theo.

Vai trò:

- Đảm bảo dữ liệu đầu vào luôn ở dạng một mảng số nguyên lớn, giúp cho việc phân chia và xử lý song song sau này thuận tiện, chính xác.

D) Các đoạn code quan trọng

Chia mảng và tìm kiếm song song thread

```
def parallel_search_chunk(arr, target, start_idx, end_idx, result_dict, thread_id):
    for i in range(start_idx, end_idx):
        if arr[i] == target:
            result_dict[thread_id] = i
            return
    result_dict[thread_id] = -1

def parallel_search_with_threads(arr, target, num_threads=4):
    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)
    for thread in threads:
        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
```

Ý nghĩa:

- Hàm này chia mảng thành các đoạn nhỏ, mỗi đoạn được một thread phụ trách.
- Các thread chạy song song, mỗi thread tự tìm kiếm số khóa K trong phần mình phụ trách và lưu lại kết quả nếu có.

Vai trò:

- Đảm bảo các đoạn của mảng được kiểm tra đồng thời, giúp rút ngắn thời gian tìm kiếm khóa K trong mảng lớn.
- Ghi nhớ chỉ số nếu tìm thấy trong từng thread, tổng hợp kết quả sau khi tất cả thread kết thúc.

D) Các đoạn code quan trọng

Thu thập kết quả và trả giá trị nhỏ nhất

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

Ý nghĩa:

- Sau khi tất cả các thread đã chạy xong, đoạn code này tổng hợp kết quả từ các thread.
- Nếu có nhiều thread cùng tìm thấy khóa K, sẽ chọn vị trí nhỏ nhất (gần đầu mảng nhất).
- Nếu không thread nào tìm thấy, trả về -1.

Vai trò:

- Đảm bảo kết quả tìm kiếm trả về đúng vị trí đầu tiên của khóa K nếu tồn tại trong mảng, đúng yêu cầu đề bài.
- Xử lý đơn giản nhưng hiệu quả nhờ tận dụng kết quả song song từ nhiều thread.

CHALLENGE 4

E) Tiềm năng tăng tốc và Kết luận bài làm

Tiềm năng tăng tốc:

- Chia mảng lớn cho nhiều thread tìm kiếm song song giúp giảm đáng kể thời gian so với tìm kiếm tuần tự.
- Khi số phần tử lớn, xử lý song song càng giúp tăng hiệu năng nhờ tận dụng tối đa đa luồng CPU.
- Dễ mở rộng cho các bài toán tìm kiếm hoặc kiểm tra điều kiện khác trên mảng lớn.



Kết luận:

- Phương pháp tìm kiếm song song bằng thread đơn giản nhưng hiệu quả, phù hợp với bài toán tìm kiếm trong mảng rất lớn.
- Giải pháp này thực tế, dễ áp dụng, đáp ứng tốt yêu cầu xử lý dữ liệu lớn trong thời gian ngắn.

TRÂN TRỌNG CẢM ƠN CÔ VÀ CÁC BẠN ĐÃ THEO DÕI

