Báo cáo kết quả thực nghiệm các thuật toán sắp xếp

Nhóm 5 (Phan Hoàng Phước, Nguyễn Xuân Bách)

Ngày 28 tháng 9 năm 2023

Mã Số Sinh Viên: 22521156, 22520093

Mục lục

1 Giới thiệu

Merge Sort là một trong những thuật toán về sắp xếp nổi tiếng và được sử dụng nhiều bởi các lập trình viên. Thuật toán có ưu điểm trong việc có độ phức tạp ổn định và có thể đáp ứng được nhu cầu thực tế trong thời gian cho phép. Bên cạnh đó, người ta đã áp dụng một số phương pháp kỹ thuật tạo ra các biến thể khác để thuật toán có thể chạy nhanh hơn. Một biến thể phổ biến dược dùng rộng rãi là **Parallel Merge Sort** hay *sắp xếp trộn song song*. Bài báo cáo này sẽ so sánh thực nghiệm thời gian chạy giữa **Merge Sort Sequential** hay *sắp xếp trộn tuần tự* và **Parallel Merge Sort**.

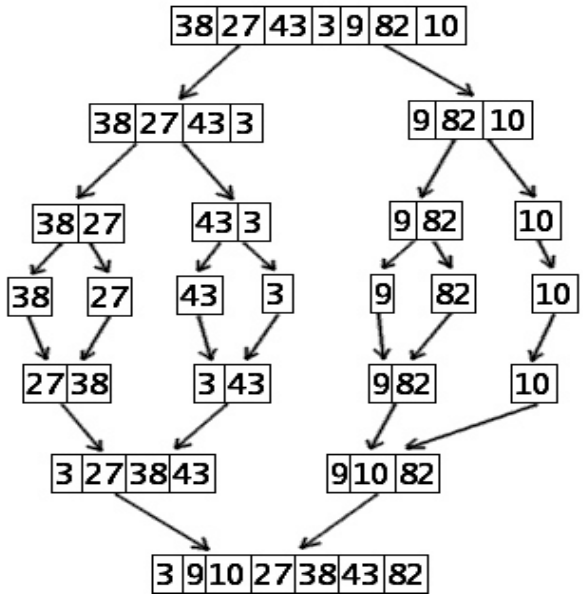
2 Bài toán

Phát biểu: Cho một dãy số gồm có N phần tử. Thiết kế thuật toán song song và sắp xếp lại dãy tăng dần theo thuật toán Merge Sort (sắp xếp trộn).
Input: Một dãy số chứa N phần tử.
Output: Dãy số ban đầu đã được sắp xếp tăng dần.
Yêu cầu: Chương trình được viết bằng ngôn ngữ Python.

3 Sequential Merge Sort

3.1 Khái quát

- Thuật toán áp dụng kỹ thuật chia để trị để chia nhỏ từng đoạn trong dãy hiện tại ra làm hai dãy con có độ dài xấp xỉ nhau và tiếp tục đến khi về trường hợp đơn vị cơ bản không thể chia tiếp được. Sau đó, xử lý trên các bài toán con rồi gộp lại và tiếp tực xử lý.
- Thuật toán Merge Sort được dùng phổ biến vì tính hiệu quả và dễ cài đặt. Tuy nhiên nhược điểm là tốn không gian bộ nhớ.
- Sở dĩ được gọi là **Merge Sort Sequential** vì trong thuật toán này không sử dụng core xử lý để xử lý riêng các mảng con mà thực hiện theo tuần tự các mảng được tách ra. Điều này có thể không tối ưu thời gian chạy dựa trên việc sử dụng hiệu quả các vi xử lý trong máy.



Hình 1: Ví dụ quá trình của thuật toán Merge Sort

3.2 Ý tưởng

- Thuật toán Merge Sort thực hiện theo các bước như sau:

1. Nếu dãy hiện tại có độ dài là 1 thì trở lại hàm trước.
2. Giả sử dãy hiện tại là đoạn $[l, r]$ và vị trí ở giữa là mid. Gọi đệ quy 2 dãy con của dãy hiện tại là $[l, mid]$ và $[mid + 1, r]$.
3. Sau khi sắp xếp 2 hai dãy con, ta gộp chúng lại vào dãy con hiện tại. Để gộp lại, ta làm như sau:
   - Tạo một dãy mới để lưu các phần sắp xếp vào.
   - So sánh 2 phần tử đầu tiên của 2 dãy con, phần tử nào nhỏ hơn thì ta bỏ vào dãy mới đã được tạo.
   - Tiếp tục thực hiện như vậy đến khi hết dãy.
4. Quay về dãy trước và tiếp tục thực hiện đến khi chạy hết chương trình.

- Đánh giá:

- Độ phức tạp: Do thuật toán chia dãy làm 2 liên tục nên sẽ luôn có độ phức tạp ổn định là $\mathcal{O}(N * log(N))$ trong mọi trường hợp.

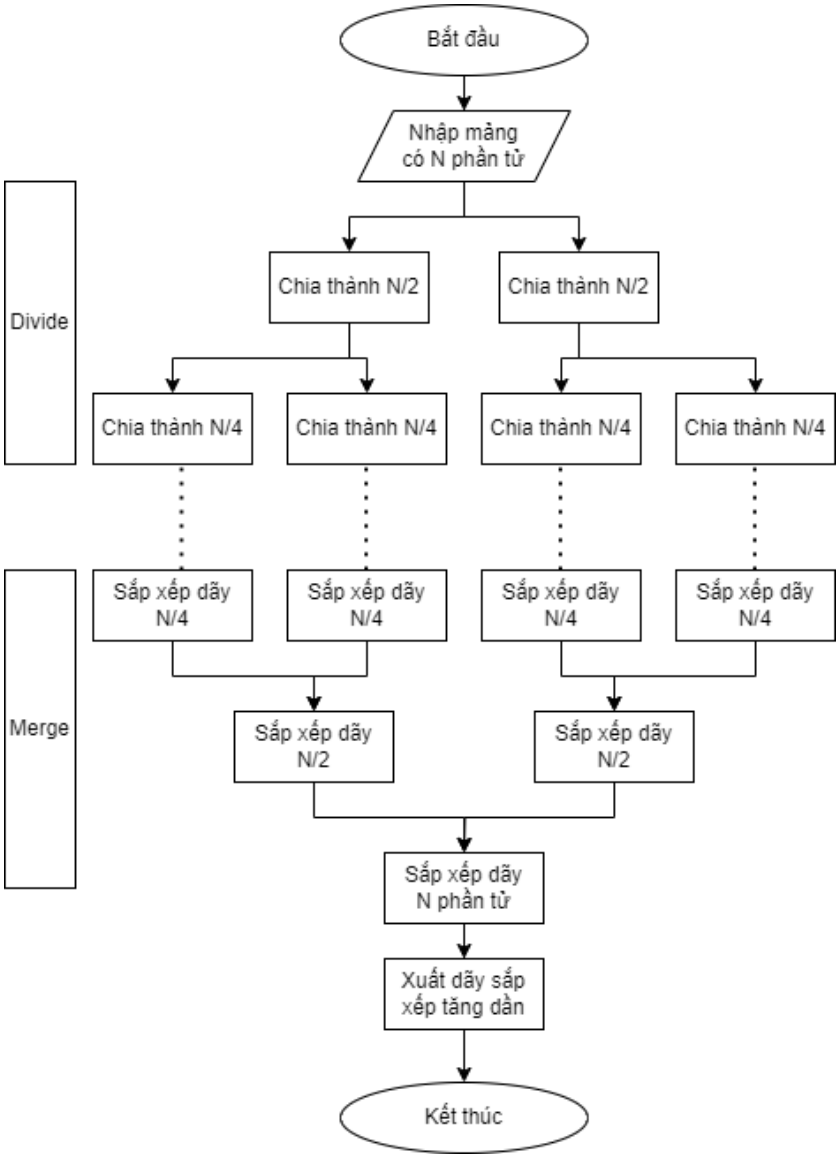
- Bộ nhớ: Do lưu trữ thêm một mảng để lưu lại các phần tử sắp xếp nên sẽ chiếm bộ nhớ là $\mathcal{O}(N)$.

Ưu điểm

- Thuật toán chạy nhanh.
- Ổn định với mọi trường hợp.

Nhược điểm

- Tốn thêm bộ nhớ để tạo mảng lưu trữ.
- Không sử dụng tối ưu các core xử lý trong máy.



Hình 2: Lưu đồ biểu diễn quy trình của thuật toán Merge Sort

3.3 Cài đặt

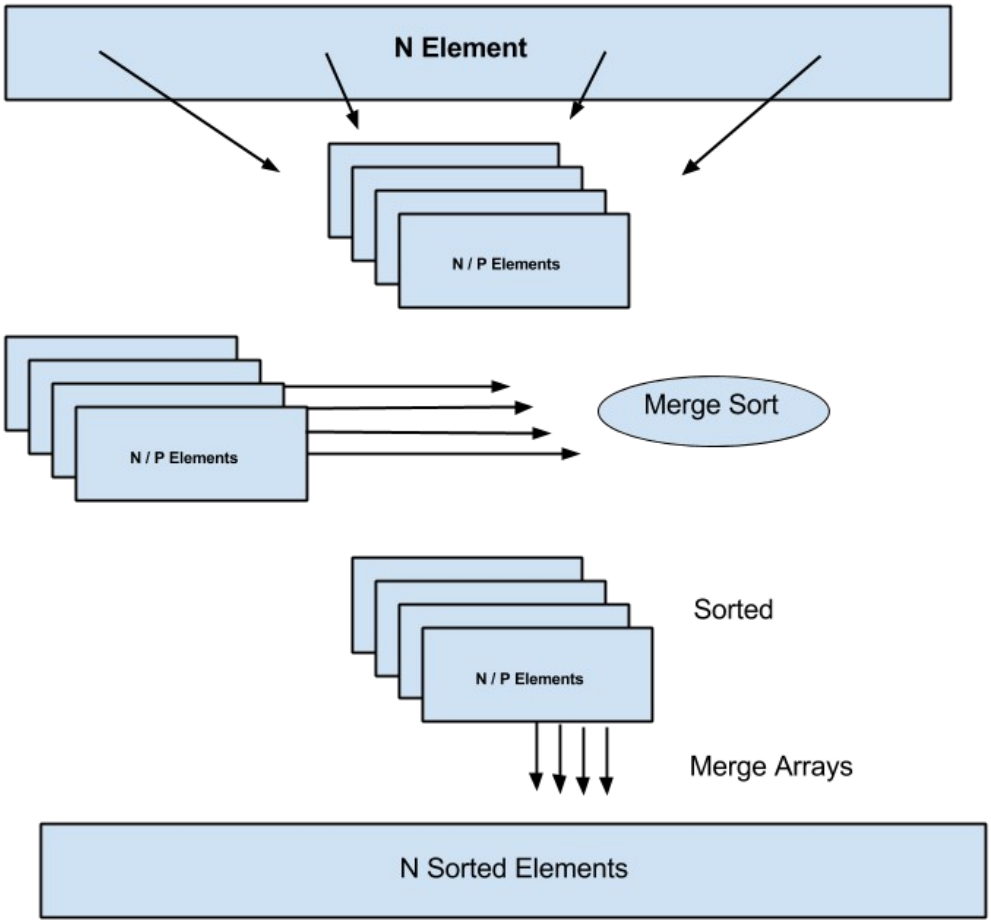
```
def merge(left, right): # hàm trộn hai mảng left và right
    result = []
    i = j = 0
    while i < len(left) and j < len(right):
        if left[i] < right[j]: # đưa phần tử nhỏ hơn
                               # vào mảng kết quả
            result.append(left[i])
            i += 1
        else:
            result.append(right[j])
            j += 1
    result.extend(left[i:]) # đưa tất cả phần tử
                            # còn lại vào mảng kết quả
    result.extend(right[j:])
    return result # trả lại kết quả

def MergeSort(arr): # hàm thực hiện sắp xếp trộn
    if len(arr) <= 1: # nếu độ dài dãy là 1
        return arr    # thì trả lại dãy
    mid = len(arr) // 2 # lấy index ở giữa
    left = MergeSort(arr[:mid]) # gọi đệ quy sort 2 dãy con
    right = MergeSort(arr[mid:])

    return merge(left, right) # trả lại mảng đã sắp xếp
```

4 Parallel Merge Sort

4.1 Khái quát

- Là một biến thể của Merge Sort nhưng được thiết kế để hoạt động trên nhiều tiểu tác vụ (subtasks) chạy đồng thời trên các bộ vi xử lý độc lập.

- Thuật toán sẽ tương tự như Sequential Merge Sort. Độ dày dãy ban đầu sẽ được chia ra thành các dãy con tương ứng với số bộ xử lý có độ dài gần bằng nhau. Các dãy con này sẽ được xử lý song song thông qua nhiều bộ xử lý hoặc luồng. Sau đó từ các dãy con này sẽ được trộn lại thành một dãy đã sắp xếp. Cơ sở của thuật toán là do thuật toán Merge Sort nguyên bản có thể sắp xếp các dãy độc lập nhau rồi trộn nên ta có thể áp dụng ý tưởng đó tương tự với các bộ xử lý.



Hình 3: Mô hình thể hiện quá trình của Parallel Merge Sort

4.2 Ý tưởng

- Thuật toán Parallel Merge Sort thực nhiên như sau:

1. Chia dãy đang có ra thành $numCore$ đoạn với $numCore$ là số lượng bộ xử lý đang dùng. Với mỗi đoạn thứ i chứa các phần tử trong dãy từ $[i.N/numCore, \min((i+1).N/numCore - 1, N)]$.

2. Với mỗi đoạn i sẽ do bộ xử lý thứ i xử lý song song với nhau và sắp xếp dựa trên thuật toán $MergeSort$ thông thường.

3. Sau khi có $N/numCore$ đoạn được sắp xếp, ta trộn các đoạn lại với nhau. Với đoạn thứ i sẽ được trộn với $i+1$, tương tự $i+2$ với $i+3$. Lặp lại bước này cho đến khi còn 1 đoạn duy nhất.

4. Trả lại đoạn kết quả đã sắp xếp.

- Đánh giá:

- Độ phức tạp: Thuật toán ban đầu chia ra $numCore$ đoạn cho các bộ xử lý, mỗi đoạn thực hiện sắp xếp nên tốn độ phức tạp $N/numCore.log(N/numCore)$. Khi trộn các mảng lại, do ta có $numCore$ mảng và mỗi mảng có $N/numCore$ phần tử nên độ phức tạp ở bước trộn lại là $N/numCore.log(numCore)$. Tổng độ phức tạp ta có là:

$$\mathcal{O}\left(\frac{N}{numCore}.log\left(\frac{N}{numCore}\right)+\frac{N}{numCore}.log(numCore)\right).$$
$$=\mathcal{O}(\frac{N}{numCore}.log(N)).$$

- Bộ nhớ: Sử dụng thêm mảng để lưu trữ dãy đã sắp xếp nên độ phức tạp không gian là $\mathcal{O}(N)$.

Ưu điểm

- Thuật toán chạy ổn định.
- Sử dụng tối ưu các core xử lý trong máy.

Nhược điểm

- Tốn thêm bộ nhớ để tạo mảng lưu trữ.
- Mỗi lần chạy phải khởi tạo các core xử lý nên tốn thời gian chạy ban đầu hơn.

4.3 Cài Đặt

```
def ParallelMergeSort(arr, numCore):
# hàm thực hiện thuật toán merge sort song song
# với numCore là số bộ xử lý
    if len(arr) <= 1:
        return arr
    chunks = []
    div, mod = divmod(len(arr), numCore)
    remain = 0
    id = 0
    while id < len(arr): # làm một vòng lặp lấy vị trí
        remain = 0
        if mod > 0:
            remain = 1
            mod -= 1
        chunks.append(arr[id:id + div + remain])
        # chunks[i] lưu lại các phần tử trong dãy thứ i
        id += div + remain
    with multiprocessing.Pool(processes=numCore) as pool:
        sortedChunks = pool.map(MergeSort, chunks)
        # hàm pool để thực hiện MergeSort
        # cho từng bộ xử lý
        while len(sortedChunks) > 1:
            nextChunks = []
            for i in range(0, len(sortedChunks), 2):
            # sắp xếp dãy i và i + 1 bằng phương pháp trộn
                if i + 1 < len(sortedChunks):
                    nextChunks.append(merge(sortedChunks[i]
                                            , sortedChunks[i + 1]))
                else:
                    nextChunks.append(sortedChunks[i])
            sortedChunks = nextChunks
        return sortedChunks[0] # trả lại kết quả cuối cùng
```

Trong phần cài đặt trên có sử dụng một hàm từ thư viện *multiprocessing* là hàm pool. Hàm pool cho phép quản lý một nhóm các tiến trình đồng thời, giúp tận dụng tối đa các lõi xử lý của máy tính. Ở trên, hàm pool giúp thực hiện hàm Merge Sort cho từng bộ xử lý trên máy tính hiện tại.

5 Thực nghiệm và đánh giá

5.1 Cấu hình máy

- Máy được sử dụng trong quá trình chạy code hiện tại có cấu hình như sau:

- Loại CPU: Intel® Core™ i7-11800H 8 cores, 16 threads, 2.3 - 4.6 GHz.
- Số Ram: 8 GB.
- Loại GPU: NVDIA RTX 3050Ti.

5.2 Data

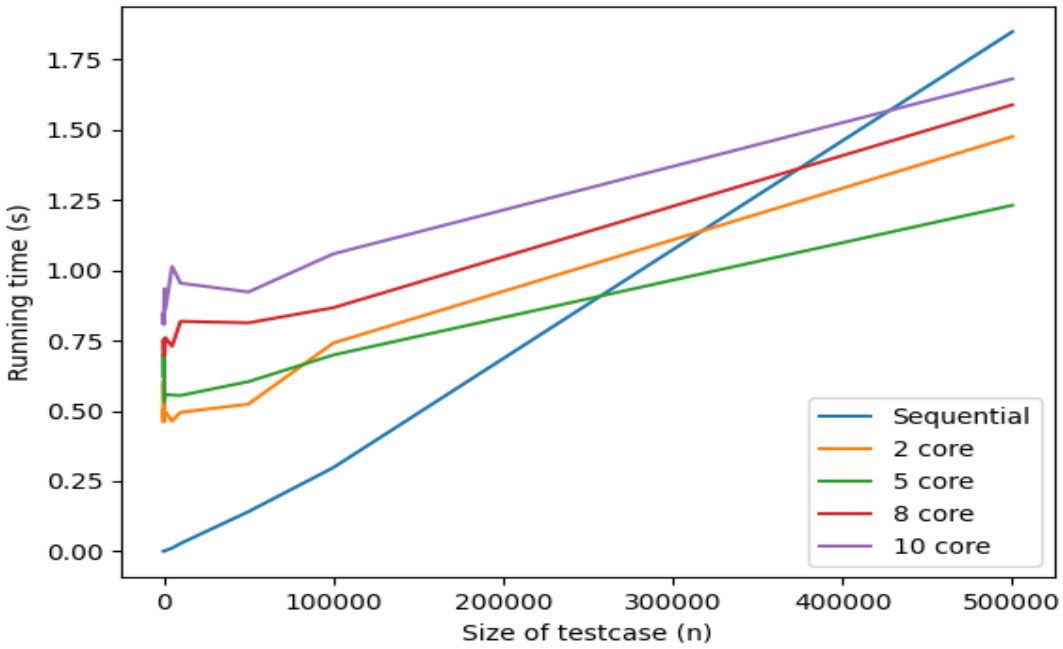
- Code sinh dữ liệu để so sánh như sau:

```
value = 1
nSize = []
for i in range(1, 8):
  value *= 10
  nSize.append(value)
  nSize.append(value * 5)
for i in range(len(nSize)):
    n = nSize[i]
    arr = []
    for i in range(n):
        arr.append(np.random.randint(0, 1000001))
```

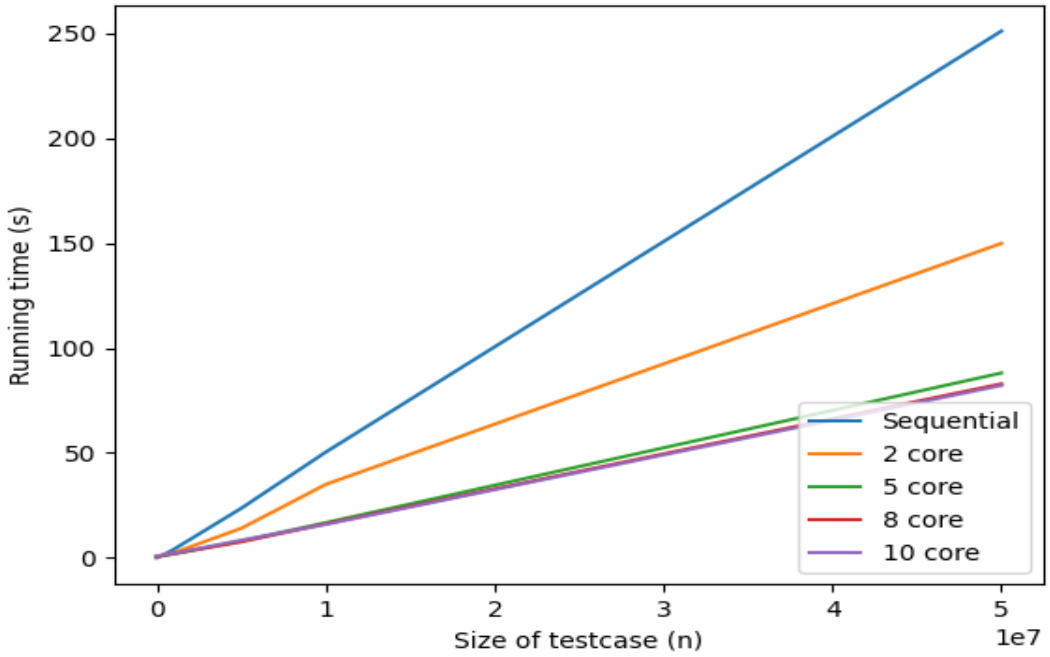
Với đoạn code trên, ta thấy được rằng bộ dữ liệu được sinh ra bao gồm các dãy có số lượng phần tử số nguyên như sau: [10, 50, 100, 500, 1000, 5000, 10000, 50000, 100000, 500000, 1000000, 5000000, 10000000, 50000000].

5.3 Kết quả thực nghiệm

Sau khi thực nghiệm thống kê qua các tập dữ liệu, dựa theo thời gian chạy của code, độ lớn trên tập dữ liệu mà rút ra 2 biểu đồ so sánh:



Hình 4: Biểu đồ thống kê thời gian chạy trên tập dữ liệu có $N \leq 5e5$



Hình 5: Biểu đồ thống kê thời gian chạy trên tập dữ liệu có $5e5 \leq N \leq 5e7$

5.4 Nhận xét

- Trong tập dữ liệu có $N \leq 5e5$, ta nhận thấy rằng Sequential Merge Sort sẽ chạy nhanh hơn so với Parallel Merge Sort. Lý do là vì:

  - Mặc dù thời chạy thuật toán ngắn nhưng việc chia tập dữ liệu thành các phần nhỏ và sắp xếp chúng đồng thời sẽ tạo ra nhiều tiến trình hoặc luồng con. Việc tạo và quản lý các tiến trình hoặc luồng đồng thời có overhead (nguồn tài nguyên được tiêu tốn để tạo và quản lý chúng). Trên các tập dữ liệu nhỏ, overhead này có trở trở thành một yếu tố quan trọng và làm giảm hiệu suất của Parallel Merge Sort.

  - Việc khởi đầu và kết thúc các tiến trình hoặc luồng cũng có thể tốn thời gian đáng kể. Trong Parallel Merge Sort, việc này xảy ra nhiều lần hơn so với Sequential Merge Sort.

- Trong tập dữ liệu có $5e5 \leq N \leq 5e7$, thời gian phát sinh do khởi tạo và quản lý các bộ xử lý không còn đáng kể mà thay vào đó là thời gian của thuật toán. Ta có thể thấy rõ thời gian thực hiện thuật toán của Sequential Merge Sort là chạy lâu nhất vì xử lý theo tuần tự, sau đó là thời gian giảm dần theo độ tăng của số lượng core được dùng.

6 Kết luận

- Việc lựa chọn Sequential Merge Sort và Parallel Merge Sort nên phụ thuộc vào tính chất cụ thể của tập dữ liệu và mục tiêu hiệu suất mong muốn. Đối với các tập dữ liệu nhỏ, việc sử dụng Sequential Merge Sort có thể là một lựa chọn hợp lý, trong khi trên các tập dữ liệu lớn, Parallel Merge Sort có thể mang lại hiệu suất tốt hơn.

7 Tham Khảo

1. Parallel Merge Sort - San José State University
2. Sắp xếp trộn - Wikipedia tiếng Việt