Đại học quốc gia TP.HCM
Trường đại học công nghệ thông tin

Môn học: Phân tích và thiết kế thuật toán

Báo cáo tìm hiểu thuật toán song song

Học viên:
Huỳnh Phạm Đức Lâm – 21521050
Nguyễn Trường Thịnh – 21520110

Giảng viên:
Nguyễn Thanh Sơn

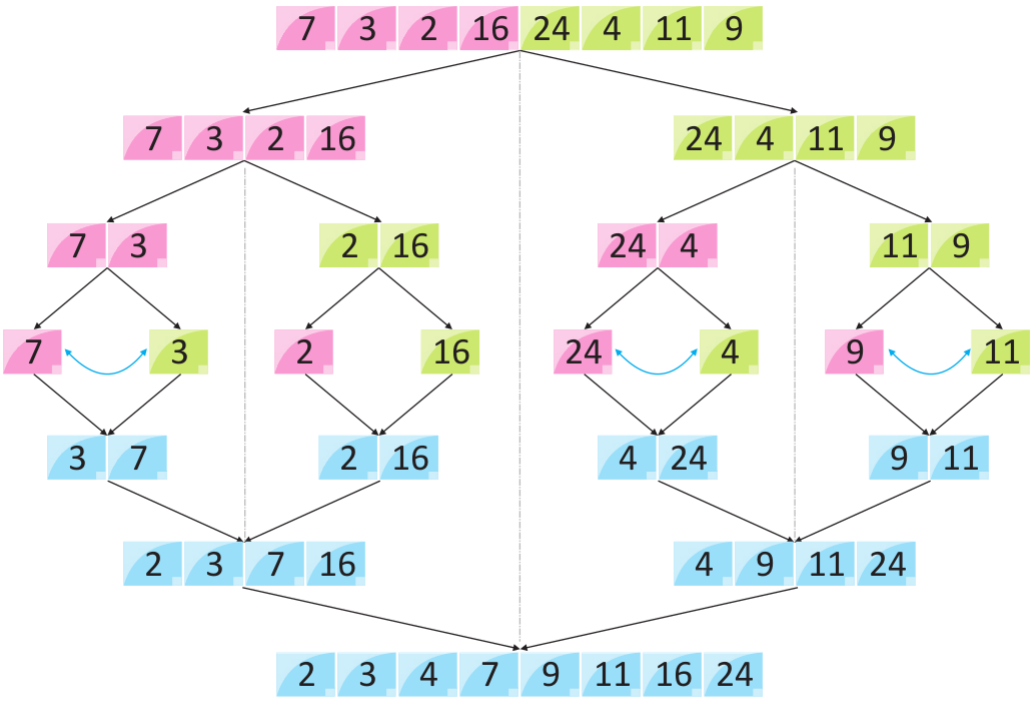
Mục lục

1 Tổng quan

Thiết kế giải thuật theo chiến lược chia để trị tập trung vào việc băm nhỏ bài toán thành những bài toán con có thể giải quyết độc lập, để từ đó tìm được lời giải cho bài toán ban đầu. Việc chia thành những bài toán con độc lập như vậy sẽ giúp cho việc áp dụng xử lí song song dễ dàng, chỉ cần đưa mỗi bài toán con vào một bộ xử lí.

Báo cáo này sẽ tập trung vào thiết kế giải thuật song song cho thuật toán Merge sort, một thuật toán chia để trị.

2 Merge sort



Hình 1: Ví dụ merge sort

Ý tưởng của merge sort là chia mảng ban đầu thành từng phần, sắp xếp từng phần đó và sau đó gộp các phần đã sắp xếp đó lại (merge) để ra được mảng đã sắp xếp hoàn chỉnh.

Merge sort thường được cài đặt theo đệ quy:

```
def merge_sort(data):
    length = len(data)
    if length <= 1:
        return data
    middle = length // 2
```

```
    left = merge_sort(data[:middle])
    right = merge_sort(data[middle:])
    return merge(left, right)
```

Trong đó thao tác merge() ta có thể sử dụng 2 con trỏ để ghép 2 mảng đã sắp xếp với độ phức tạp O(N)

3 Merge sort song song

3.1 Thư viện multiprocessing

Trong python, ta có thể sử dụng thư viện multiprocessing để sử dụng nhiều bộ xử lí. Với việc tạo các pool chứa các CPU, ta có thể sử dụng pool.map() đưa các task vào pool để xử lí đồng bộ.

```
def function(input):
    return result

# input, divided into chunks
input = [chunk1, chunk2, ...]

# create pool with n processes
pool = multiprocessing.Pool(processes=n)

# start worker processes in parallel
# output from each function call
# is appended to results
results = pool.map(function, input)
```

[1]

3.2 Merge sort song song

Để thực hiện song song hóa merge sort, ta sẽ phân chia bộ dữ liệu ban đầu thành n mảng con (với n là số process). Tại sao lại không chia nhỏ hơn nữa? Việc song song hóa thuật toán mục đích chính là để tận dụng tối đa tài nguyên CPU. Với việc chia bộ dữ liệu thành số phần bằng với số bộ xử lí, ta đã phân chia đều công việc cho các bộ xử lí, đã tận dụng được toàn bộ CPU cho đến bước sort các mảng con. Việc phân chia nhỏ hơn nữa sẽ khiến CPU giảm hiệu suất do tốn thêm thời gian để chuyển ngữ cảnh, tốn thêm thời gian để nạp các mảng mới.

Tuy nhiên, khi đến các bước trên cùng (số lượng mảng con $<$ số lượng process), việc sử dụng thuật toán ghép O(n) sẽ chỉ tận dụng được 1 core CPU, không tận dụng hết các tài nguyên đã có.

Giải thuật merge sort song song như sau:

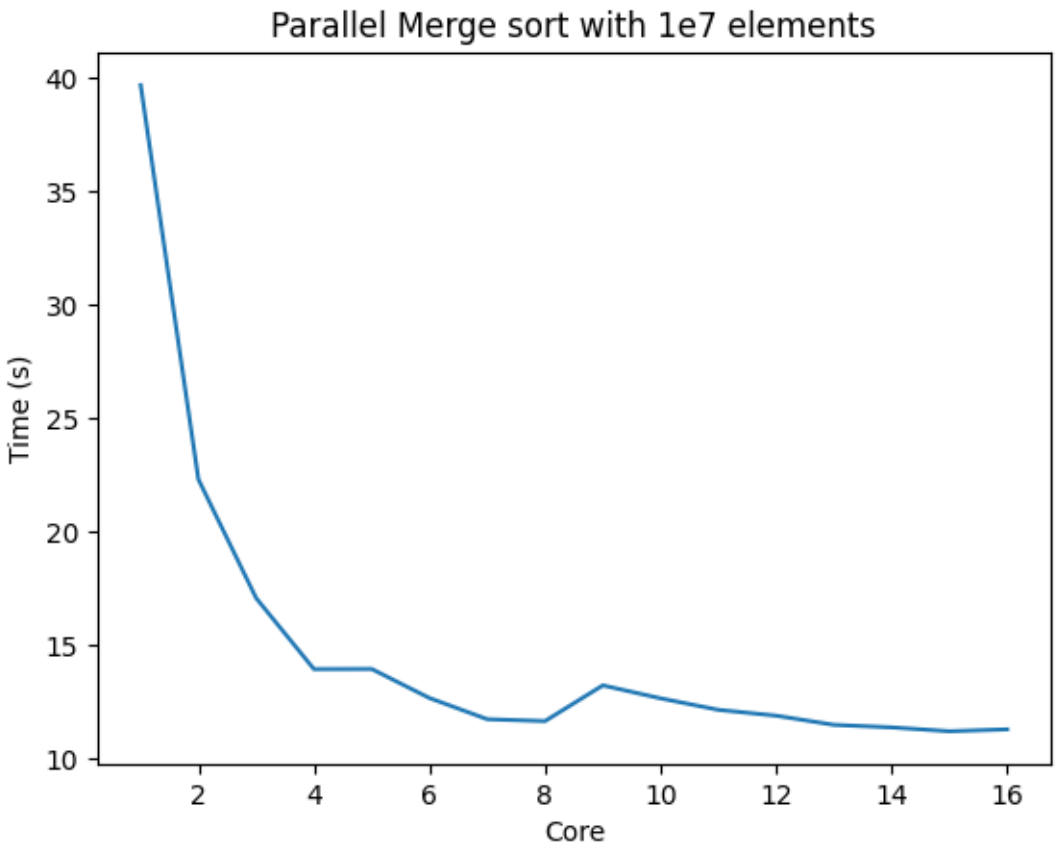
- Chia mảng đã cho thành n phần (n = số process)
- Gọi merge sort trên n phần đó
- Khi số lượng mảng con sắp xếp $>$ 1:

- Nếu số lượng mảng con là lẻ thì ta sẽ để mảng con cuối ra biến khác (Vì ta chỉ quan tâm việc ghép cặp)
- Gọi merge() trên mỗi cặp với các CPU (1 CPU đảm nhận 1 cặp)
- Đưa mảng con còn thừa vào (nếu có).

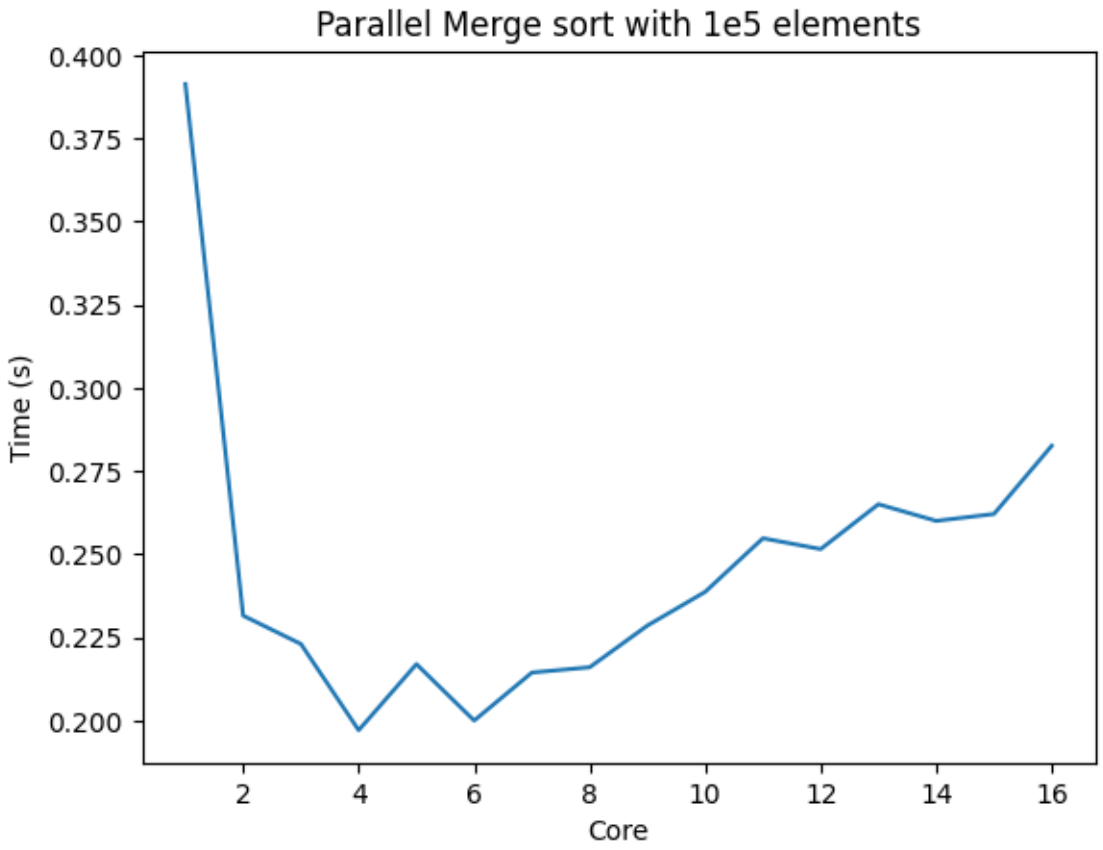
Việc cài đặt trên ngôn ngữ python đã có trên github của nhóm [2]

4 Thử nghiệm

Môi trường thử nghiệm: sử dụng CPU Ryzen 5800HS (8 core, 16 threads, 2.8GHz base clock) trên Windows 11.
Dữ liệu thử nghiệm: 2 bộ (Lớn: 10.000.00 phần tử, Nhỏ: 10.000 phần tử)



Hình 2: Thời gian thực hiện trên mảng 10.000.000 phần tử



Hình 3: Thời gian thực hiện trên mảng 100.000 phần tử

Nhận xét:

- Với bộ dữ liệu lớn, thời gian thực hiện giảm cho đến bội số của 4 (4 core, 8 core, ..) sau đó tăng. Điều này có thể là cho kiến trúc của CPU. Mỗi CCX (Core CompleX) chứa 4 core cùng chia sẻ cache L3. Việc sử dụng 2 CCX không đồng đều có thể khiến tài nguyên bộ nhớ không được chia sẻ nhanh chóng với các core khác.
- Với bộ dữ liệu nhỏ, thời gian thực hiện nhỏ nhất là ở 4 core. Ở mức 8 core, nếu tăng dẫn số core thì thời gian thực hiện tăng lên đáng kể.

5 Tính toán độ phức tạp

Việc cài đặt Merge Sort thông thường sẽ chạy trong O(nlogn) và bộ nhớ O(n). Với giải thuật Merge Sort song song, giả sử ta chạy trên x core cùng lúc:

- Việc chia mảng thành x phần và áp dụng merge sort thông thường trên mỗi phần sẽ mất đpt là: O((n/x)*log(n/x)).
- Để Merge x mảng đã sắp xếp thành một mảng sau cùng ta cần ít nhất

$$\sum_{i=1}^{log(n)} 2^i - 1$$

Tương đương với O(n).

- Vậy tổng độ phức tạp thuật toán là: O(n/x*log(n/x)+n)

Nhận xét: Thuật toán sẽ chạy chậm với bộ dữ liệu nhỏ, nhưng cực kì hiệu quả khi dữ liệu càng lớn.

6 Mở rộng

Việc thiết kế giải thuật song song cho merge sort đến bước chia nhỏ thành số phần bằng số process đã khá ổn. Tuy nhiên, khi thực hiện merge 2 mảng con lớn, ta có thể thực hiện thuật toán chậm hơn nếu chỉ làm việc độc lập nhưng lại tỏ ra nhanh hơn khi làm việc song song.

Đó chính là giải thuật Binary Search, chậm hơn nhiều so với cách xử lí 2 con trỏ. Nhưng khi có nhiều CPU việc tận dụng tính đơn lẻ trong việc tìm kiếm của tìm kiếm nhị phân khiến ta có thể chia nhỏ mảng như sau:

Với mỗi phần tử trong mảng trái, ta sẽ sử dụng tìm kiếm nhị phân trên mảng phải, kết hợp với chỉ số của phần tử đó ở mảng trái thì ta sẽ ra được chỉ số của phần tử đó ở mảng cuối cùng. Vì mỗi bước này là độc lập với nhau, ta có thể đưa các bước tìm kiếm nhị phân này vào nhiều CPU để chúng xử lí song song với nhau.

- Độ phức tạp thuật toán merge 2 con trỏ: O(n)
- Độ phức tạp thuật toán merge tìm kiếm nhị phần O(nlogn)

Tuy độ phức tạp chậm hơn, nhưng nếu số lượng core > log(n) thì phương pháp này sẽ nhanh hơn. Với các CPU server như EPYC, Xeon-W hiện nay, 1 CPU có thể chứa đến 64 core có thể thực hiện merge rất nhanh.

Vì khả năng nhóm có hạn nên việc cài đặt theo thuật toán này nhóm chưa thực hiện được. Ở [3] là code tham khảo C++ có sử dụng giải thuật này.

Tài liệu

[1] Alexandra Yang (2022) Approaches to the Parallelization of Merge Sort in Python
https://arxiv.org/pdf/2211.16479.pdf

[2] https://github.com/Darklul03/CS112/tree/master/MergeParallel

[3] https://github.com/GlenGGG/FastParallelMergeSort/blob/master/omp_mergesort.c