TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI
HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY

Lập trình song song thuật toán Merge Sort trên CPU và GPU

Nhóm 7:
Trần Hoài Nam - 20224412
Nguyễn Hoàng Tân - 20224426
Nguyễn Đăng Khánh - 20224440
Lương Hữu Phúc - 20224452

NỘI DUNG

- Đặt vấn đề
- Merge Sort là gì
- Song song hóa trên CPU bằng OpenMP
- Song song hoá trên GPU bằng Cuda
- Kết quả và đánh giá
- Giữa kì :OpenMP
- Cuối kì :CUDA

TẠI SAO LẠI CẦN SONG SONG HÓA

Merge Sort vốn là thuật toán đệ quy Merge Sort chia nhỏ bài toán → gọi đệ quy trái và phải → mỗi nhánh có thể xử lý độc lập.

Phù hợp với **chia để trị** (Divide & Conquer) Do chia thành 2 phần liên tục → có thể chia cho nhiều luồng xử lý cùng lúc.

Tăng tốc độ sắp xếp Với mảng lớn (triệu phần tử), nếu tuần tự sẽ rất lâu → song song giúp giảm đáng kể thời gian.

Tận dụng CPU đa lõi hiệu quả Mỗi phần của Merge Sort có thể chạy trên 1 core → **tận dụng toàn bộ** tài nguyên hệ thống.

Tối ưu hiệu suất xử lý dữ liệu lớn Merge Sort thường dùng cho dữ liệu lớn → song song hóa giúp xử lý nhanh hơn rất nhiều. chuyển bảng này thành ảnh đi

PHÂN TÍCH TÍNH KHẢ THI

Bản chất thuật toán:

Chia mảng thành 2 nhánh độc lập, dễ song song. **Khả thi cao**

Phân chia công việc:

Nhánh đệ quy song song tốt, merge khó song song.**Khả thi một phần**

Hiệu quả thời gian:

Nhanh hơn với dữ liệu lớn (triệu phần tử). **Khả thi**

Phần cứng:

Cần CPU đa lõi, phổ biến hiện nay. **Khả thi**

Overhead:

Dữ liệu nhỏ có thể chậm hơn do tạo luồng. **Hạn chế**

Merge Sort là gì ?

Sử dụng phương pháp chia để trị (divide to conquer):

- Chia mảng thành 2 nửa nhỏ
- Sắp xếp đệ quy từng nửa
- Sau đó hợp nhất kết quả

Độ phức tạp *O(nlogn)* đảm bảo hiệu suất ổn định ngay cả với mảng lớn

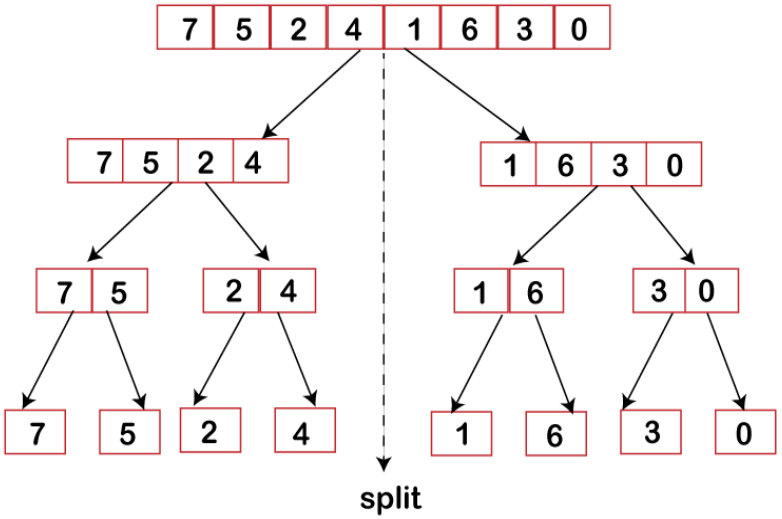
💡 **Nhờ tính chất chia đều khối lượng công việc thực thi nên có thể áp dụng được kỹ thuật song song**

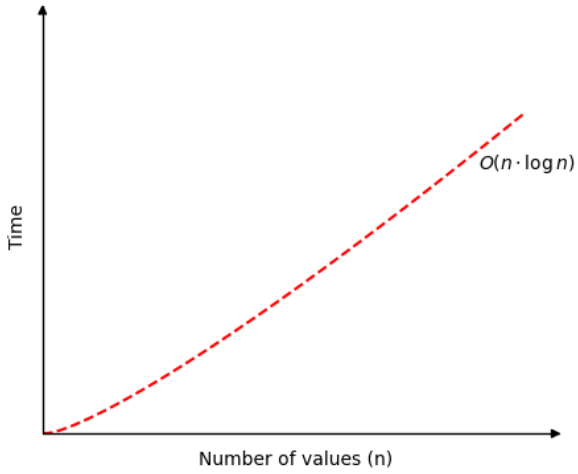
Time Complexity

O(n log n)

Memory Complexity

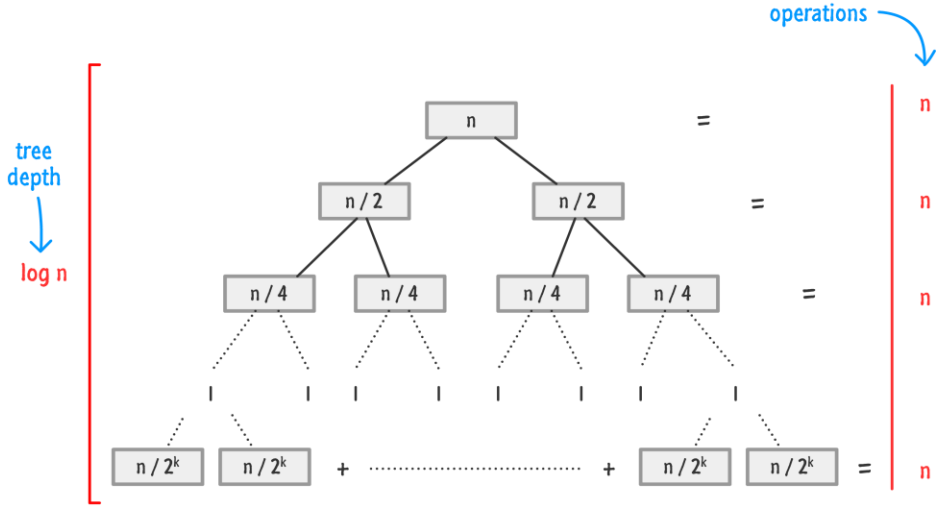
O(n)





Song song hóa trên CPU bằng OpenMP

Thực thi tuần tự:



Khi số phần tử tăng, thời gian xử tăng theo cấp số nhân (gần tuyến tính với n, nhưng độ sâu đệ quy lại tăng $log_2(n)$)

✗ Tuy nhiên chỉ chạy 1 CPU core

⇒ Sử dụng **OpenMP** sẽ tận dụng được sức mạnh của CPU đa nhân

⇒ **Phù hợp để tối ưu bằng song song hóa:**

- Song song trên các hàm đệ quy của thuật toán
- Thực hiện với mảng động n phần tử, các phần tử được sinh ngẫu nhiên
- Khi số phần tử < 10000, nên thực hiện merge sort tuần tự để tránh overhead, tạo ra thread không đáng có, gây chậm chương trình

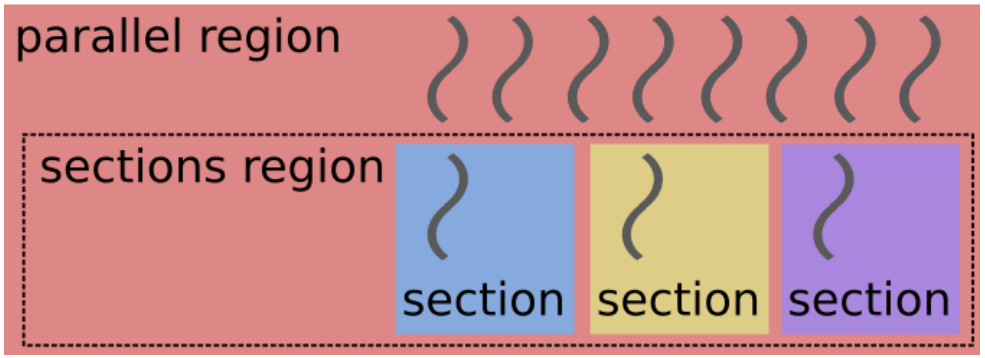
Song song hóa bằng OpenMP Sections

Ý tưởng: Phân chia nhánh trái, nhánh phải thành các sections riêng biệt, sau đó merge lại khi các sections hoàn tất

TH1: Sử dụng chỉ thị: #pragma omp parallel sections

```
#pragma omp parallel sections //T
{
  #pragma omp section //Thread 1
  merge_sort(a, left, mid); //De

  #pragma omp section //Thread 2
  merge_sort(a, mid + 1, right);
}
```



- Đơn giản, dễ triển khai, phù hợp cho các bài toán chia nhánh rõ ràng
- Có thể tận dụng CPU đa nhân

- Hạn chế hiệu suất với số lượng phần tử lớn -> Đệ quy sâu
- Mỗi lần gọi đệ quy lại sinh thêm thread -> thread lồng thread
- Tiêu tốn bộ nhớ, không tận dụng tối đa CPU

Song song hóa bằng OpenMP Task

TH2: Sử dụng chỉ thị: #pragma omp task

```
#pragma omp parallel
{
  #pragma omp single //Dam bao chi
  merge_sort(a, 0, num - 1);
}
```

Hàm main

- **#pragma omp parallel** + **#pragma omp single:** Tạo pool thread, đảm bảo chỉ 1 thread đầu tiên gọi merge_sort(), thread khác chờ để nhận task

```
#pragma omp task shared(a) firstprivate(left, mid)
merge_sort(a, left, mid);

#pragma omp task shared(a) firstprivate(right, mid)
merge_sort(a, mid + 1, right);

#pragma omp taskwait //Cho ca 2 task xong roi moi merge
```

Hàm chứa thành phần đệ quy

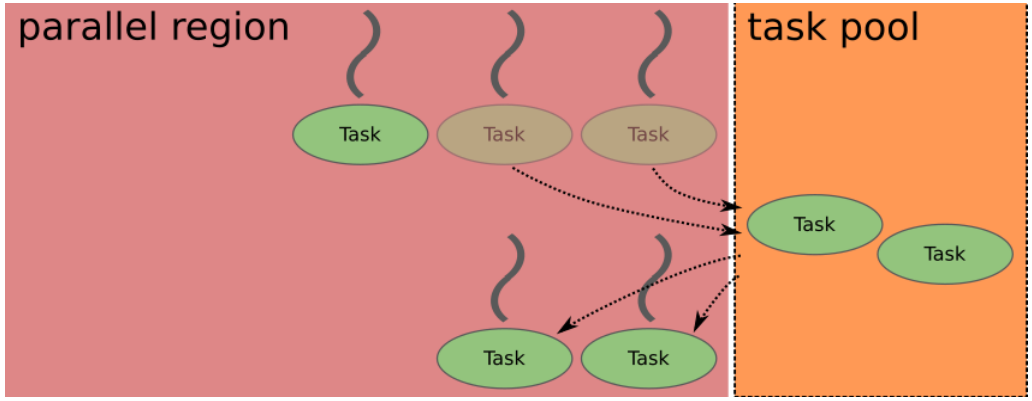
- **#pragma omp task:** tạo 1 nhiệm vụ con cho thread khác xử lý khi có sẵn tài nguyên
- **shared(a):** tất cả các task dùng chung mảng a
- **firstprivate(left, min):** sao chép giá trị left, mid vào biến cục bộ riêng của task
- **#pragma omp taskwait**: chờ cả 2 task hoàn thành rồi merge lại

Song song hóa bằng OpenMP Task

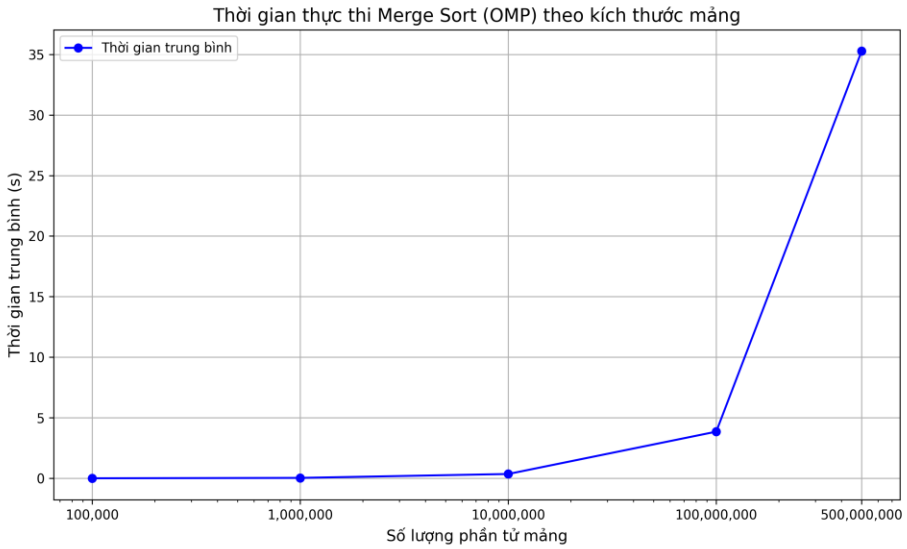
TH2: Sử dụng chỉ thị: #pragma omp task

Quy trình trên máy tính

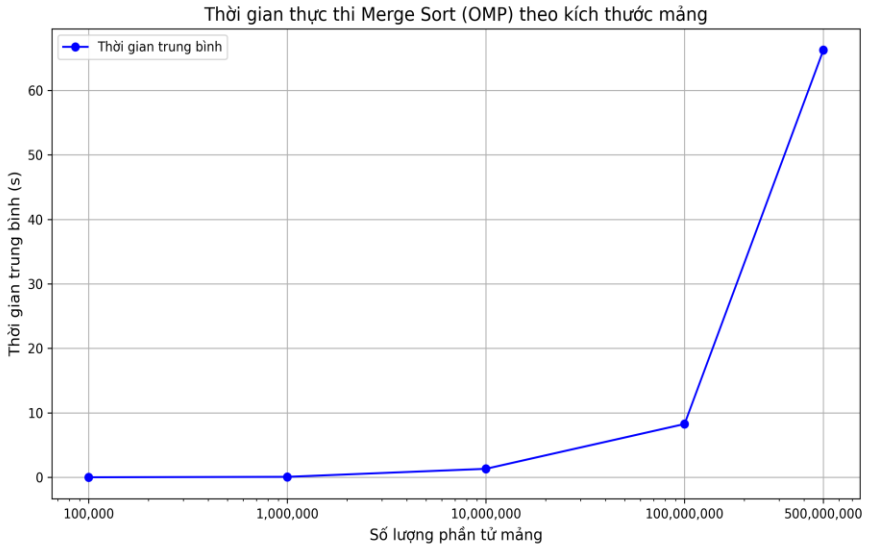
CPU thread	Task
Thread 0	Khởi động merge_sort và tạo task bằng #pragma omp task
Thread 1	Lấy task từ hàng đợi (nếu có) và chạy merge_sort()
Thread 2	Lấy task khác và tiếp tục đệ quy
Thread n	Tiếp tục chia nhỏ công việc



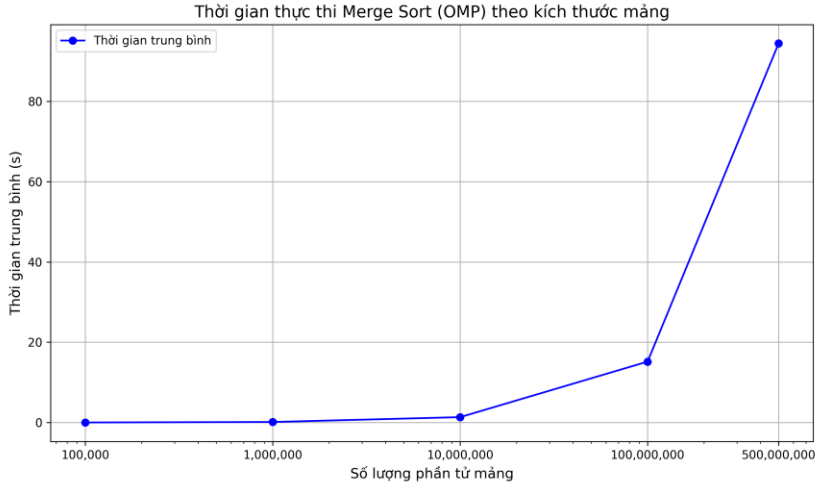
Đánh giá kết quả



Average_time_omp_task

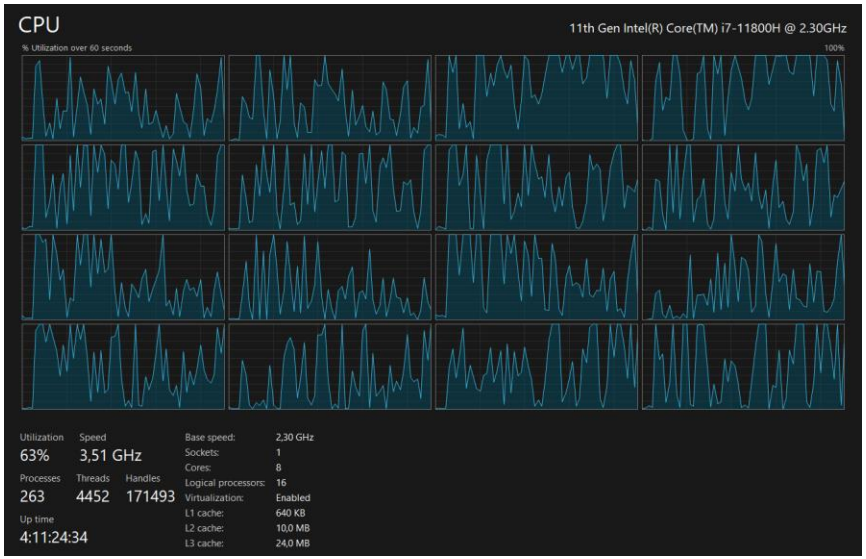


Average_time_sections

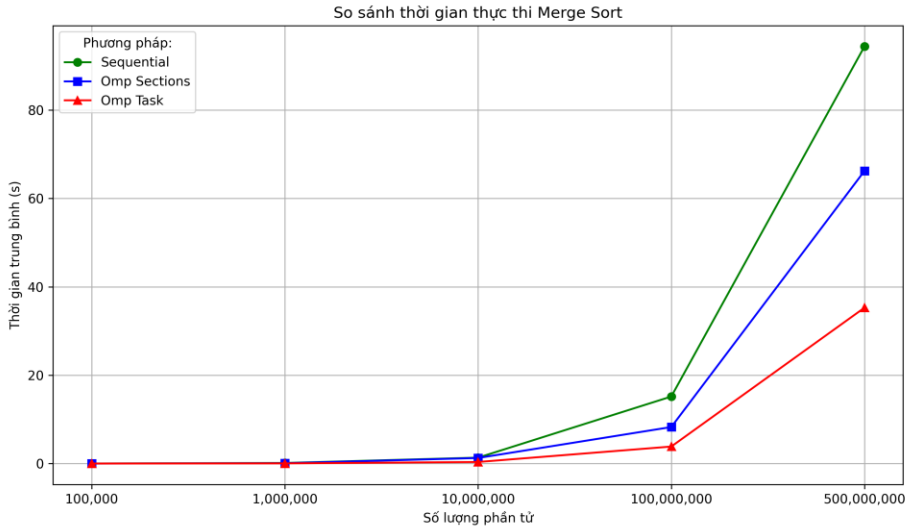


Average_time_sequential

Đánh giá kết quả



CPU khi xử lý song song



So sánh thời gian thực thi

- Đối với mảng từ 100k -> 10 triệu phần tử: sự khác biệt không quá rõ
- Khi mảng từ 100 triệu trở lên:
  - **Omp task**: cho tốc độ xử lý nhanh nhất nhờ khả năng chia nhỏ đệ quy một các linh hoạt
  - **Omp sections**: Cho hiệu quả cao hơn tuần tự nhưng bị giới hạn bởi số section đặt ra
  - **Sequential**: Mất nhiều thời gian nhất do xử lý tuần tự trên 1 thread duy nhất

⇒ ***Sử dụng omp task cho hiệu quả cao nhất do CPU được chia nhiều trên nhiều cores, thể hiện qua biểu đồ sử dụng tài nguyên***

Tính toán kết quả (Phân tích hiệu suất song song)

Các thông số sau cho omp task và omp sections so với chuẩn tuần tự (Merge_sort_sequential):

Tăng tốc (Speedup - S):
Tỷ lệ giữa thời gian thực thi tuần tự và thời gian thực thi song song.

Thời gian thực thi (tính bằng giây):

$$S = \frac{T_{\text{tuần tự}}}{T_{\text{song song}}}$$

Kích thước đầu vào	Merge_sort_sequential	omp task	omp sections
100K	0.0123999914	0.0033333460	0.0081333160
1M	0.1331333478	0.0344666481	0.0733333270
10M	1.3413999716	0.3562666893	1.2609333515
100M	15.1702000300	3.8511333307	8.2645333290
500M	94.4135333538	35.2743333181	66.2394666513

Kết quả sau khi tính toán Speedup:

Kích thước đầu vào	S_omp task	S_omp_sections
100K	3.72	1.52
1M	3.86	1.82
10M	3.76	1.06
100M	3.94	1.84
500M	2.68	1.43

Tính toán kết quả

Hiệu suất ()
Hiệu suất được tính bằng tăng tốc chia cho số luồng:

$$E = \frac{S}{P}$$

Kích thước đầu vào	E_omp task	E_omp sections
100K	0.47	0.19
1M	0.48	0.23
10M	0.47	0.13
100M	0.49	0.23
500M	0.34	0.18

Tính toán kết quả

Khả năng mở rộng (Scalability)

Xu hướng tăng tốc:

- omp task: Tăng tốc đạt đỉnh ở 100M phần tử (3.94) nhưng giảm xuống 2.68 ở 500M, cho thấy chi phí song song hoặc tranh chấp tài nguyên (như băng thông bộ nhớ, tranh chấp cache) tăng lên với đầu vào lớn.
- omp sections: Tăng tốc thấp hơn, đạt đỉnh ở 1M và 100M (1.82–1.84), nhưng giảm mạnh xuống 1.06 ở 10M và 1.43 ở 500M, cho thấy phân phối công việc không hiệu quả.

Xu hướng hiệu suất:

- omp task: Hiệu suất ổn định (~0.47–0.49) đến 100M, nhưng giảm xuống 0.34 ở 500M, phản ánh hạn chế về khả năng mở rộng.
- omp sections: Hiệu suất rất thấp (0.13–0.23), do sử dụng 16 luồng không hiệu quả, có thể vì phân phối công việc không đều hoặc chi phí đồng bộ.

TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY