BỘ GIÁO DỤC VÀ ĐÀO TẠO

TRƯỜNG ĐẠI HỌC PHENIKAA

BÁO CÁO BÀI TẬP LỚN

Lập Trình Song Song

ĐỀ TÀI

Ứng Dụng Giải Thuật Song Song trong Thuật toán Sắp Xếp - Quick Sort

Giảng viên hướng dẫn: PGS.Ts. Lê Văn Vinh

Sinh viên thực hiện:			
	Phạm Hoàng Anh	21011234	K15CNTT1
	Trần Minh Hiếu	21011601	K15CNTT3
	Nguyễn Quang Hệ	21011599	K15CNTT3
	Vi Đăng Quang	21010583	K15CNTT1

Khoa: Công Nghệ Thông Tin

HÀ NỘI, THÁNG 3 2025

Phân công nhiệm vụ

Nhóm gồm có bốn thành viên: Phạm Hoàng Anh, Trần Minh Hiếu, Nguyễn Quang Hệ và Vi Đăng Quang. Dưới đây là bảng phân công nhiệm vụ chi tiết cho dự án này.

Thành viên	Nhiệm vụ	Hoàn thành
Phạm Hoàng Anh	Khái quát cơ sở lý thuyết, Trực quan hóa dữ liệu được tổng hợp, viêt báo cáo	25 %
Trần Minh Hiếu	Tổng hợp và phân tích kết quả của thuật toán Quick Sort khi sử dụng OpenMP, PThreads, MPI, viết báo cáo	25 %
Nguyễn Quang Hệ	Code tuần tự, OpenMP, viết báo cáo	25 %
Vi Đăng Quang	Code Pthreads, MPI, chạy thực nghiệm, viết báo cáo	25 %

Hà Nội, ngày 01 tháng 03 năm 2025

Chữ ký đại diện nhóm

LỜI CẢM ƠN

Được sự hướng dẫn của **PGS.Ts. Lê Văn Vinh** – cán bộ trực tiếp giảng dạy học phần Lập trình song song, với sự nhiệt tình và tâm huyết đã trang bị những kiến thức quý giá, giải đáp thắc mắc giúp chúng em có được những kỹ năng trong việc ứng dụng kỹ thuật vào việc phát triển đề tài **Sắp xếp N số nẫu nhiên theo thứ tự tăng dần và giảm dần bằng thuật toán sắp xếp nhanh (Quicksort) bằng cách sử dụng các kiến thức tuần tự và song song đã học**. Chúng em mong rằng đề tài sau khi thực hiện sẽ được hoàn thiện và phát triển giúp cho việc quản lý cũng như vận hành trong lĩnh vực này sẽ trở nên số hóa, tiện lợi và dễ dàng hơn.

Do hạn chế về mặt thời gian và hiểu biết, đề tài của chúng em có thể còn nhiều thiết sót, rất mong sẽ nhận được sự góp ý, bổ sung từ thầy và các bạn để nhóm chúng em có thể hoàn thiện vốn kiến thức của mình, tạo hành trang vững chắc cho việc phát triển trong tương lai.

Chúng em xin chân thành cảm ơn!

Mục lục

Mục lục hình ảnh

Mục lục bảng

1 Mục đích nghiên cứu

Trong thời đại **dữ liệu lớn (Big Data) và trí tuệ nhân tạo (AI)**, việc xử lý và phân tích dữ liệu với tốc độ nhanh, hiệu suất cao là một yêu cầu tất yếu. Tuy nhiên, với khối lượng dữ liệu khổng lồ, các thuật toán xử lý tuần tự truyền thống không còn đáp ứng được tốc độ mong muốn, dẫn đến việc chậm trễ trong xử lý, làm giảm hiệu quả tính toán và khai thác dữ liệu.

1.1 Lý do quan trọng

Lập trình song song xuất hiện như là giải pháp mạnh mẽ giúp tận dụng tối đa tài nguyên phần cứng, đặc biệt là trên các hệ thống đa nhân (multi-core) hoặc phân tán (distributed computing). Việc áp dụng lập trình song song không chỉ giúp tăng tốc độ xử lý mà còn giúp giảm tải tài nguyên, tối ưu hiệu suất hệ thống. Vì vậy, chúng em quyết định hoàn thành đề tài này một cách thật tâm huyết, thử nghiệm và đánh giá **tác động của lập trình song song lên thuật toán QuickSort**, một trong những thuật toán sắp xếp phổ biến và quan trọng trong khoa học máy tính.

1.2 Phương pháp thực hiện

Để thu hoạch được kết quả của đề tài một cách hiệu quả nhất, chúng em triển khai đề tài theo các bước sau:

1. **Nghiên cứu thuật toán Quick Sort:** Hiểu rõ cơ chế hoạt động của thuật toán, cách chia mảng và phương pháp chọn pivot hiệu quả.
2. **Cài đặt Quick Sort tuần tự:** Viết chương trình thực hiện thuật toán QuickSort theo phương pháp truyền thống để làm cơ sở so sánh.
3. **Ứng dụng song song trên Quick sort:**
    - **Sử dụng OpenMP:** Tận dụng API hỗ trợ lập trình song song trên CPU đa nhân.
    - **Sử dụng PThreads:** Triển khai mô hình luồng (threads) thủ công, quản lý chia công việc bằng cách đồng bộ hóa dữ liệu.
    - **Sử dụng MPI:** Thực hiện thuật toán trong môi trường tính toán phân tán, giao tiếp giữa các tiến trình (processes).

1.3 Mục tiêu

Xuất phát từ những lý do trên, đề tài này chúng em muốn hướng đến các mục tiêu sau:

- Xây dựng chương trình sắp xếp QuickSort theo hai cách: tuần tự và song song.
- Hiện thực thuật toán song song bằng OpenMP, Pthreads và MPI, đánh giá sự khác biệt giữa các mô hình lập trình song song.
- Khảo sát hiệu suất chương trình với số lượng luồng (threads/processors) khác nhau (p = 2, 4, 6, 8, 12) để đánh giá khả năng tăng tốc (speedup) và tính mở rộng (scalability).
- So sánh hiệu quả giữa lý thuyết và thực nghiệm, từ đó đưa ra kết luận về tính khả thi của lập trình song song trong thực tế.

1.4 Cấu trúc báo cáo

Mục 2 trình bày tổng quan về lập trình song song, bao gồm cơ sở lý thuyết, mô hình thực thi song song và cách đánh giá hiệu suất. Mô tả quá trình triển khai thuật toán QuickSort song song bằng OpenMP, Pthreads và MPI, cùng với thiết lập thực nghiệm và các yếu tố kỹ thuật quan trọng.

Mục 3 trình bày kết quả thực nghiệm, phân tích thời gian thực thi, hệ số tăng tốc (speedup) và hiệu suất của từng mô hình song song.

Mục 4 thảo luận về khả năng mở rộng, các hạn chế và đề xuất hướng phát triển trong tương lai để tối ưu hóa thuật toán.

Mục 5 kết luận báo cáo và đề xuất hướng nghiên cứu tiếp theo.

2 Phương pháp nghiên cứu

2.1 Cơ sở lý thuyết

2.1.1 Tổng quan về lập trình song song

Lập trình song song (Parallel Computing) được hiểu là mô hình tính toán trong đó nhiều tác vụ được thực thi đồng thời để tận dụng tối đa tài nguyên phần cứng, từ đó tăng tốc độ xử lý và cải thiện hiệu suất hệ thống. Trong mô hình này, một bài toán lớn

được chia nhỏ thành nhiều phần và thực thi đồng thời trên nhiều luồng xử lý (threads) hoặc bộ xử lý (processors).

Lập trình song song có 4 mô hình chính:

- Song song mức dữ liệu (Data Parallelism): Cùng một tác vụ được thực hiện đồng thời trên nhiều phần dữ liệu.
- Song song mức tác vụ (Task Parallelism): Các tác vụ khác nhau được thực thi đồng thời.
- Song song mức hướng bộ nhớ (Memory Parallelism): Các tiến trình truy xuất bộ nhớ độc lập để tránh tình trạng tắc nghẽn.
- Song song mức hướng pipeline (Pipeline Parallelism): Dữ liệu được xử lý theo từng giai đoạn, tương tự dây chuyền sản xuất.

2.1.2 Cơ chế lập trình song song ở phần cứng

Hiện nay, kiến trúc phần cứng hỗ trợ lập trình song song bao gồm các hệ thống sau:

- Đa lõi (Multi-Core CPUs): Một vi xử lý có nhiều nhân (cores), mỗi nhân có thể thực thi độc lập một luồng (thread). Ví dụ, CPU Intel Core i7 có 8 nhân vật lý, mỗi nhân có thể chạy 2 luồng (HyperThreading) $\rightarrow$ tổng cộng 16 luồng xử lý.
- Đa bộ xử lý (Multi-Processor Systems): Hệ thống có nhiều CPU vật lý, thường xuất hiện trong các máy chủ hiệu năng cao.
- Bộ đồng xử lý (Coprocessors - GPU, FPGA, TPU): Hỗ trợ xử lý song song mạnh mẽ, đặc biệt là trong lĩnh vực AI, Machine Learning và xử lý đồ họa.

Trong thực hành lập trinh song song qua các bài toán đã được thầy giao, chúng em rút ra được vấn đề là khi triển khai lập trình song song trên CPU đa lõi, cần quản lý hiệu quả việc phân chia công việc (workload balancing) để tránh tình trạng một số luồng hoạt động quá tải trong khi các luồng khác rảnh rỗi (load imbalance).

2.1.3 Cơ chế lập trình song song ở mức mã máy

Ở cấp độ mã máy, lập trình song song được thể hiện bằng nhiều cách như:

- Sử dụng lệnh SIMD (Single Instruction Multiple Data): Một lệnh duy nhất xử lý đồng thời nhiều dữ liệu, tận dụng tập lệnh như SSE, AVX trên CPU.

- Sử dụng đa luồng (Multithreading): Một tiến trình (process) có thể tạo ra nhiều luồng (threads), chia sẻ bộ nhớ chung nhưng hoạt động độc lập. Các thư viện như Pthreads, OpenMP hỗ trợ mô hình này.
- Sử dụng mô hình đa tiến trình (Multiprocessing): Chạy song song nhiều tiến trình độc lập, mỗi tiến trình có bộ nhớ riêng. MPI (Message Passing Interface) là một ví dụ tiêu biểu.

Từ những cơ sở lý thuyết trên, trong bài toán đề tài đưa ra, chúng em sẽ áp dụng:

- Pthreads để quản lý luồng độc lập và đồng bộ hóa dữ liệu.
- OpenMP để dễ dàng khai báo và kiểm soát luồng với cú pháp đơn giản.
- MPI để thực thi thuật toán trong môi trường tính toán phân tán.

2.1.4 Cách tính toán hiệu suất trong lập trình song song

Một hệ thống song song không phải lúc nào cũng nhanh hơn hệ thống tuần tự do ảnh hưởng của quá tải giao tiếp (communication overhead) và tắc nghẽn bộ nhớ (memory contention). Chúng em sẽ đánh giá hiệu suất thuật toán thông qua:

- Speedup (Gia tốc): Đo tỷ lệ thời gian thực thi giữa chương trình tuần tự và chương trình song song (với p là số luồng xử lý). Công thức tính gia tốc được tính bằng công thức Eq.1

$$S_p = \frac{T_{Sequential}}{T_{Parallel}} \tag{1}$$

- Hiệu suất sử dụng CPU(Efficiency): Giúp chúng em có thể đánh giá mức độ sử dụng tài nguyên theo số luồng. Công thức tính hiệu suất sử dụng CPU tính bằng công thức Eq.2

$$E_p = \frac{S_p}{p} \tag{2}$$

2.2 Thuật toán sắp xếp

Thuật toán sắp xếp (Sorting Algorithm) là một nhóm thuật toán được sử dụng để sắp xếp các phần tử trong một tập dữ liệu theo một thứ tự nhất định, thường là tăng dần hoặc giảm dần. Sắp xếp dữ liệu là một bước quan trọng trong nhiều ứng dụng tin học, giúp tối ưu hóa quá trình tìm kiếm, quản lý dữ liệu và cải thiện hiệu suất của các thuật toán khác.

Trong bài báo cáo này, chúng em sẽ đi sâu vào tìm hiểu thuật toán Quick Sort, ứng dụng và triển khai lập trình song song nhằm tăng hiệu năng của thuật toán. (Phần 2.2.3)

2.2.1 Một số thuật toán sắp xếp phổ biến

Thuật toán sắp xếp đơn giản:

- Bubble Sort: So sánh từng cặp phần tử liền kề và đổi chỗ nếu chúng không theo đúng thứ tự.
- Selection Sort: Tìm phần tử nhỏ nhất và hoán đổi nó với phần tử đầu tiên, tiếp tục với phần còn lại của danh sách.
- Insertion Sort: Chèn từng phần tử vào vị trí thích hợp trong danh sách đã sắp xếp.

Thuật toán sắp xếp nâng cao:

- Merge Sort: Chia danh sách thành các phần nhỏ hơn, sắp xếp từng phần và sau đó hợp nhất lại theo thứ tự.
- Quick Sort: Chọn một phần tử làm "pivot", phân chia danh sách thành hai phần theo giá trị của pivot, sau đó sắp xếp từng phần.
- Heap Sort: Sử dụng cấu trúc dữ liệu Heap để sắp xếp phần tử theo thứ tự mong muốn.

2.2.2 Ứng dụng của thuật toán sắp xếp

Nhiều thuật toán tìm kiếm như Tìm kiếm nhị phân yêu cầu dữ liệu được sắp xếp trước để đạt hiệu quả cao. Trong học máy, thuật toán sắp xếp hỗ trợ lựa chọn đặc trưng (Feature Selection), đặc biệt trong Feature Engineering. Ví dụ, trong Random Forest, ta có thể sắp xếp các đặc trưng quan trọng theo thứ tự giảm dần để cải thiện hiệu suất mô hình học máy.

2.2.3 Thuật toán Quick Sort

Quick Sort là một thuật toán sắp xếp nổi tiếng, được xây dựng dựa trên phương pháp "chia để trị" (Divide and Conquer). Ý tưởng chính của thuật toán là:

- Chọn một phần tử làm “pivot” (phần tử chốt).

- Phân chia mảng: Sắp xếp lại mảng sao cho tất cả các phần tử nhỏ hơn (hoặc bằng) pivot được đặt bên trái, và các phần tử lớn hơn pivot được đặt bên phải.
- Đệ quy: Sau khi phân chia, áp dụng Quick Sort cho từng nửa mảng (bên trái và bên phải) cho đến khi mảng con có độ dài 0 hoặc 1 (đã được sắp xếp).

Nguyên lý hoạt động:

Chọn Pivot

Có nhiều cách để chọn Pivot:

- Phần tử đầu tiên
- Phần tử cuối cùng
- Phần tử ở giữa
- Chọn ngẫu nhiên

Ví dụ trong khuôn khổ bài báo cáo này, chúng em trình bày chọn **phần tử cuối cùng của mảng làm Pivot**

Quá trình phân vùng

Quá trình phân vùng có nhiệm vụ đưa pivot về vị trí đúng của nó trong mảng đã sắp xếp. Các bước thực hiện như sau:

- Đầu tiên, chúng ta sẽ khởi tạo chỉ số bằng cách đặt biến i = low - 1, với low là chỉ số bắt đầu của mảng hiện hành
- Duyệt chỉ số j từ low đến high - 1 (với high là chỉ số cuối cùng của mảng). Nếu phần tử arr[j] nhỏ hơn hoặc bằng pivot, ta tăng i lên và hoán đổi arr[i] với arr[j]. Điều này đảm bảo các phần tử nhỏ hơn hoặc bằng pivot sẽ được tập hợp ở phần đầu mảng.
- Mục đích của thuật toán là đưa pivot về vị trí chính xác. Sau khi duyệt hết các phần tử, hoán đổi arr[i + 1] với arr[high]. Khi đó, pivot sẽ nằm ở vị trí i+1, với các phần tử bên trái đều nhỏ hơn hoặc bằng nó và bên phải đều lớn hơn nó.

Đệ Quy (Recursion):

Sau khi thực hiện phân vùng, mảng được chia thành hai phần:

- Mảng bên trái từ chỉ số lo đến i (với pivot nằm ở i+1). Mảng bên phải từ chỉ số i+2 đến hi.

- Ta gọi đệ quy Quick Sort cho từng phần này cho đến khi không còn phần mảng nào cần sắp xếp (điều kiện dừng: lo $\geq$ hi).

Ta có thể tóm tắt lại nguyên lý hoạt động của thuật toán như sau:

Thuật toán Quick Sort

Bước 1. Chọn Pivot: Chọn một phần tử trong danh sách làm pivot (thường là phần tử đầu, cuối, giữa hoặc ngẫu nhiên)

Bước 2. Phân chia (Partitioning): Chia danh sách thành hai phần

Bước 2.1. : Phần bên trái chứa các phần tử nhỏ hơn pivot.

Bước 2.2. : Phần bên phải chứa các phần tử lớn hơn pivot.

Bước 3. Đệ quy: Áp dụng QuickSort cho hai phần danh sách bên trái và bên phải.

Bước 4. Kết hợp: Sau khi các phần đã được sắp xếp, danh sách hoàn chỉnh sẽ được kết hợp lại.

Mã giả cho thuật toán Quick Sort:

```
function quickSort(arr, low, high):
    if low < high:
        pivotIndex = partition(arr, low, high)
        quickSort(arr, low, pivotIndex - 1)
        quickSort(arr, pivotIndex + 1, high)

function partition(arr, low, high):
    pivot = arr[high]
    i = low - 1
    for j from low to high - 1:
        if arr[j] <= pivot:
            i = i + 1
            swap(arr[i], arr[j])
    swap(arr[i + 1], arr[hi])
    return i + 1
```

Độ phức tạp của thuật toán Quick Sort có trung bình là $O(n \log n)$. Trong trường hợp tốt nhất là $O(n \log n)$ (nếu pivot chia danh sách thành hai phần bằng nhau). Trong trường hợp xấu nhất là $O(n^2)$ (nếu pivot luôn là phần tử nhỏ nhất hoặc lớn nhất).

3 Kết quả số

Trong Mục 3, chúng tôi/em sẽ trình bày kết quả và phân tích kết quả thực nghiệm thu được. Kết quả được thực nghiệm với các giải thuật tuần tự và song song bằng Quick sort sắp xếp tăng dần. Còn kết quả thực nghiệm giảm dần có trong phần Phụ lục A.3

3.1 Môi trường thực nghiệm

Thí nghiệm được thực hiện trên máy tính cá nhân với các thông số phần cứng như sau:

- **Hệ thống**: Dell Inspiron 16 Plus 7620
- **Bộ xử lý (CPU)**: Intel Core i7-12700H (12th Gen)
- **Bộ nhớ đệm (Cache)**:
  - L1 Cache: 512KiB + 288KiB + 192KiB
  - L2 Cache: 4MiB + 7680KiB
  - L3 Cache: 24MiB + 24MiB
- **Bộ nhớ hệ thống (RAM)**: 40GB SODIMM 4800 MHz
- **Ổ lưu trữ (SSD)**: NVMe Kioxia 1024GB
- **Hệ điều hành**: Ubuntu
- **Trình biên dịch**: GCC
- **Số luồng tối đa**: 20 threads

Chương trình sắp xếp được thực hiện bằng ngôn ngữ lập trình C, sử dụng thuật toán Quicksort để sắp xếp một mảng gồm 30 triệu số nguyên ngẫu nhiên, ngoài ra chúng tôi/em chạy giải thuật tuần tự và song song 100 lần để đưa ra kết quả chính xác nhất. Hiệu năng của thuật toán được đánh giá trên môi trường tuần tự và song song, với các công nghệ sau:

- **Thực hiện tuần tự**: Chạy trên một luồng (single-threaded).
- **Thực hiện song song**: Sử dụng OpenMP, Pthreads và MPI để tối ưu hiệu suất trên các cấu hình đa luồng (lên đến 20 threads).

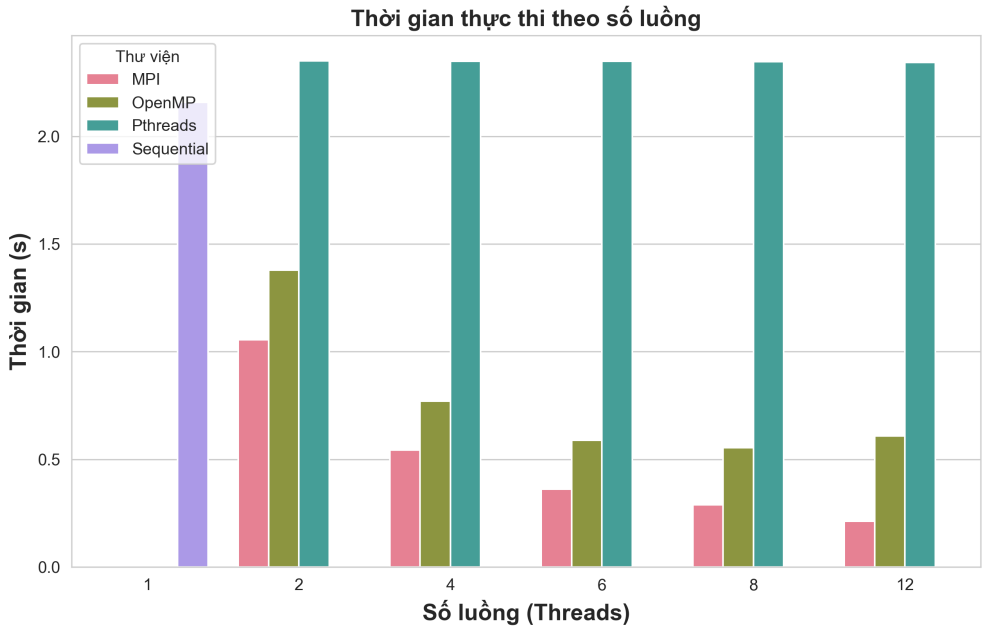
Kết quả thu được sẽ phản ánh tác động của số luồng (threads) và kiến trúc CPU đối với hiệu năng thuật toán.

3.2 Phân tích tính hiệu quả

Trước khi đi vào phân tích, ta cần hiểu ý nghĩa của Speed up và Hiệu suất trong lập trình song song:

- Speed up cho biết chương trình chạy nhanh hơn bao nhiêu lần khi sử dụng tiến trình hoặc luồng
- Hiệu suất (E)thể hiện phần trăm tài nguyên tính toán thực sự được sử dụng một cách hiệu quả. Ví dụ:
  - E $\approx$ 1 (100%), chương trình tận dụng tài nguyên tốt
  - E $\ll$ 1, chương trình có thể gặp hiện tượng tranh chấp tài nguyên hoặc overhead(việc tạo, hủy, quản lý các luồng và tiến trình, hay việc cần dùng mutex để quản lý sử dụng tài nguyên chung) khiến chương trình chạy chậm hơn cả tuần tự.

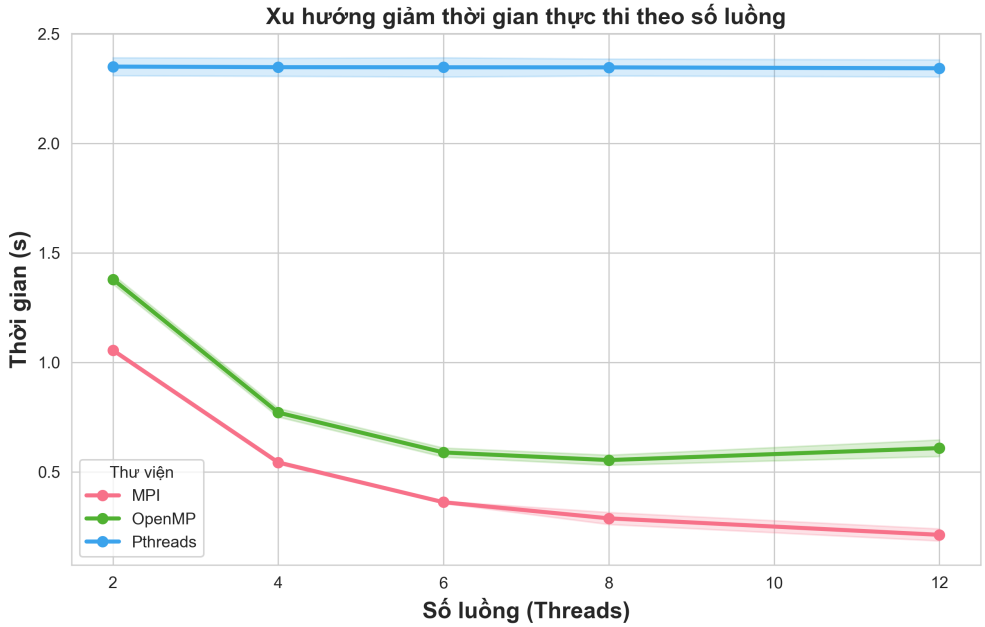
Dưới đây là bảng kết quả thực nghiệm thời gian chạy, hệ số tăng tốc (Speedup) và hiệu suất (Efficiency) của các giải pháp song song so với chạy tuần tự:



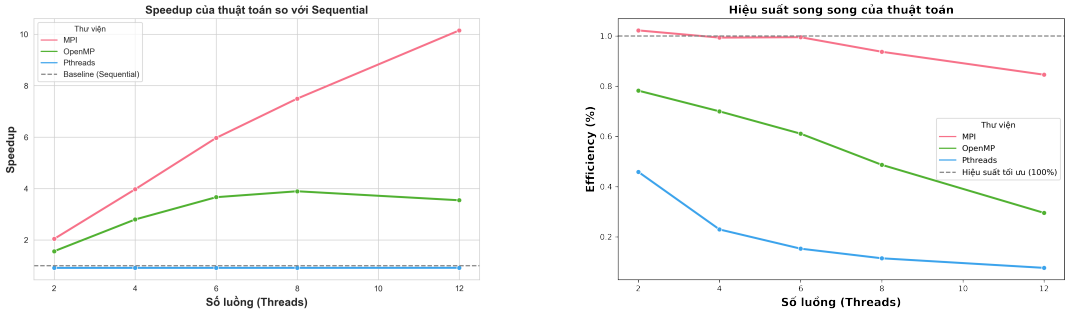
Hình 1: Biểu đồ cột so sánh thời gian thực thi giữa các thư viện

Program	Threads	Time (s)	Speedup	Efficiency
MPI	2	1.055433	2.044393	1.022196
MPI	4	0.542968	3.973935	0.993484
MPI	6	0.361389	5.970626	0.995104
MPI	8	0.287809	7.497039	0.937130
MPI	12	0.212607	10.148868	0.845739
OpenMP	2	1.378867	1.564849	0.782425
OpenMP	4	0.770669	2.799800	0.699950
OpenMP	6	0.588610	3.665788	0.610965
OpenMP	8	0.553807	3.896158	0.487020
OpenMP	12	0.608473	3.546123	0.295510
Pthreads	2	2.350072	0.918150	0.459075
Pthreads	4	2.347409	0.919192	0.229798
Pthreads	6	2.346932	0.919378	0.153230
Pthreads	8	2.346479	0.919556	0.114944
Pthreads	12	2.342451	0.921137	0.076761
Sequential	1	2.157719	1.000000	1.000000

Bảng 1: Kết quả chạy thuật toán Quicksort trên các cài đặt song song khác nhau



Hình 2: So sánh xu hướng thời gian thực thi dựa trên số luồng



(a) Biểu đồ đường so sánh Speedup giữa các thư viện

(b) Biểu đồ đường so sánh hiệu suất giữa các thư viện

Hình 3: Biểu đồ so sánh

Dựa trên kết quả thực nghiệm từ bảng dữ liệu và các biểu đồ trong Hình 1, 2, 3a, 3b, tính hiệu quả của các phương pháp đa luồng (MPI, OpenMP, Pthreads) trong việc tối ưu thuật toán Quicksort cho mảng 30 triệu phần tử được phân tích như sau:

1. Hiệu suất thời gian thực thi (Hình 1)
    - MPI cho thời gian thực thi giảm mạnh từ 1.055s (2 luồng) xuống 0.213s (12 luồng), thể hiện khả năng song song hóa hiệu quả. Xu hướng này phù hợp với biểu đồ đường trong Hình 2, nơi đường MPI dốc xuống rõ rệt.
    - OpenMP có thời gian giảm từ 1.379s (2 luồng) xuống 0.608s (12 luồng), nhưng không ổn định khi số luồng tăng (tăng nhẹ ở 12 luồng), phản ánh qua độ dốc thấp hơn MPI trong Hình 2.
    - Pthreads có thời gian gần như không thay đổi ($\sim$ 2.34s), thậm chí chậm hơn Sequential (2.158s), cho thấy vấn đề trong triển khai hoặc overhead quản lý luồng.
2. Tốc độ tăng tốc (Hình 3a)
    - MPI đạt speedup 10.15x với 12 luồng (Hình 3a), chứng tỏ khả năng mở rộng tuyến tính.
    - OpenMP chỉ đạt speedup 3.55x với 12 luồng, thấp hơn nhiều so với MPI.
    - Pthreads có speedup < 1 ở mọi cấu hình, cho thấy phương pháp này không phù hợp cho bài toán này.
3. Hiệu suất song song (Hình 3b)

- MPI duy trì hiệu suất >84% ngay cả với 12 luồng (Hình 3b), chứng tỏ phân tải công việc đồng đều và tối ưu hóa tốt.
- OpenMP có hiệu suất giảm mạnh từ 78.2% (2 luồng) xuống 29.5% (12 luồng), phản ánh chi phí đồng bộ hóa hoặc contention tài nguyên.
- Pthreads hiệu suất cực thấp (<46%), cho thấy overhead quản lý luồng thủ công là đáng kể.

4 Thảo luận

MPI là lựa chọn tối ưu cho bài toán này, đặc biệt khi số luồng lớn (>8), nhờ kiến trúc phân tán và ít phụ thuộc vào shared memory.

OpenMP phù hợp cho số luồng vừa phải (4-6), nhưng kém hiệu quả khi mở rộng do giới hạn của mô hình shared-memory.

Pthreads không phù hợp trong trường hợp này, cần kiểm tra lại cách phân chia công việc.

MPI thể hiện ưu thế rõ rệt trong xử lý song song quy mô lớn, trong khi OpenMP phù hợp cho các tác vụ đơn giản hơn.

Với những thực nghiệm, ta thấy rõ ràng rằng Pthreads chạy rất chậm, thậm chí còn lâu hơn so với giải thuật tuần tự. Vậy có cách nào để làm tăng hiệu suất, độ hiệu quả của Pthreads không? Câu trả lời là có, dựa trên kiến thức về contention và threshold. Ta có thể chia ngưỡng khi nào dùng song song và tuần tự. Qua Hình 4 và Bảng 2 cho thấy rằng thời gian thực thi của Pthreads tăng lên một cách rõ rệt khi vượt mặt tất cả các thư viện song song với thời gian thực thi sắp xếp trên 3.10^7 là ~ 0.0605s. Code triển khai của Pthreads - cải tiến được trình bày trong phần Phụ lục A.2.

4.1 Tính mở rộng

Chúng em có áp dụng các thư viện trên một số lượng mẫu to hơn 3.10^8 phần tử trong mảng. Bảng 3 cho thấy thời gian thực thi của các giải thuật song song dựa trên số luồng. Ta thấy rằng MPI (triển khai tiến trình) là một giải pháp ổn định song việc áp dụng đúng cách phân chia tài nguyên (contention) và ngưỡng (threshold) số phần tử bằng việc áp dụng tuần tự thay cho song song đã giảm mạnh thời gian thực thi.

4.2 Đề xuất

Cho những ai hứng thú về việc triển khai các giải thuật song song cho các thuật toán sắp xếp, chúng tôi/em có những đề xuất sau đây:

- Luôn nghĩ đến việc phân chia tài nguyên giữa các luồng một cách hợp lý bằng việc sử dụng khóa (ví dụ mutex lock trong Pthread)
- Đưa ra một ngưỡng số phần tử (threshold) để chuyển về tuần tự hợp lý. Dựa trên quan sát, việc áp dụng threshold làm tăng hiệu suất đáng kể thông qua Hình 4.
- Kết hợp MPI với các thư viện MPI và Pthreads, bằng việc tạo tiến trình (MPI) song song với tạo threads để ứng dụng vào bài toán là một hướng đi tuyệt vời (song vì hạn chế về mặt thời gian, chúng tôi/em vẫn chưa hoàn thiện xong phần này. Vì vậy chúng tôi/em sẽ triển khai phần trong tương lai.)

4.3 Hạn chế và hướng đi tương lai

4.3.1 Hạn chế

Mặc dù việc triển khai thuật toán Quicksort song song bằng MPI, OpenMP và Pthreads đã cải thiện đáng kể hiệu suất so với phiên bản tuần tự, vẫn còn một số hạn chế cần được xem xét:

Giảm hiệu suất khi tăng số luồng: Khi số luồng vượt quá một mức nhất định (ví dụ: 8 hoặc 12 luồng), hiệu suất không còn tăng đáng kể, thậm chí có thể giảm do chi phí đồng bộ hóa và tranh chấp tài nguyên.

Quản lý bộ nhớ: Việc tạo và quản lý luồng có thể dẫn đến hiện tượng tắc nghẽn (bottleneck) khi số luồng quá cao. Cần quản lý tài nguyên đặt ra ngưỡng sử dụng tuần tự như trong phần Pthreads cải tiến Hình 4. Cụ thể, qua thực nghiệm chạy giải thuật tuần tự và giải thuật song song OpenMP (2 luồng), ta thấy rằng với 5000 phần tử thì giải thuật tuần tự có thời gian thực thi nhanh hơn so với khi ứng dụng song song (OpenMP, PThreads)

4.3.2 Hướng đi tương lai

Việc phân chia nhiệm vụ cho các luồng là một bài toán khó. Ta cần xem xét kỹ để phân bổ tài nguyên một cách hợp lý. Ngoài ra, còn có những hướng đi như là kết hợp MPI với Pthreads hay OpenMP. Trong tương lai, chúng tôi/em sẽ tiếp tục nghiên cứu các phương pháp tối ưu hơn, đặc biệt là việc kết hợp giữa MPI và OpenMP để tận dụng ưu điểm của cả hai mô hình, đồng thời áp dụng trên các kiến trúc phần cứng khác nhau để đánh giá hiệu suất trên phạm vi rộng hơn.

5 Kết luận

Trong báo cáo này, chúng tôi/em đã nghiên cứu và triển khai thuật toán QuickSort theo hai phương pháp: tuần tự và song song, sử dụng OpenMP, Pthreads và MPI. Kết quả thực nghiệm cho thấy việc áp dụng lập trình song song có thể giúp cải thiện đáng kể hiệu suất thuật toán, tuy nhiên hiệu quả này phụ thuộc vào cách triển khai và số luồng xử lý.

- MPI đạt hiệu suất cao nhất, với speedup gần tuyến tính khi tăng số luồng. Điều này chứng tỏ tính hiệu quả của mô hình phân tán trong việc tối ưu thuật toán sắp xếp trên tập dữ liệu lớn.
- OpenMP hoạt động tốt khi số luồng vừa phải (4-6), nhưng hiệu suất giảm khi số luồng quá lớn do overhead quản lý bộ nhớ chung.
- Pthreads không mang lại hiệu suất mong đợi, thậm chí chậm hơn thuật toán tuần tự trong một số trường hợp. Điều này có thể do overhead quản lý luồng, contention tài nguyên, hoặc chưa tối ưu phân chia công việc. Từ đó, chúng tôi/em đã cải tiến, tối ưu đoạn code của Pthreads đạt được kết quả ngoài mong đợi.
- Thông qua các thí nghiệm, chúng tôi/em nhận thấy rằng để tối ưu lập trình song song, cần xem xét các yếu tố như: lựa chọn mô hình phù hợp, giảm thiểu contention bộ nhớ, và xác định ngưỡng tối ưu để chuyển đổi giữa thuật toán tuần tự và song song, như PThreads - cải tiến.

Tài liệu tham khảo

[1] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Education, 2003.

[2] D. R. Butenhof, *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[3] B. Chapman, G. Jost, and R. V. D. Pas, *Using OpenMP: Portable Shared Memory Parallel Programming*. Cambridge, MA: MIT Press, 2007.

Phụ lục

A1. Mã nguồn C để triển khai thuật toán

A1.1 Triên khai tuần tự

```c
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

Listing 1: Triển khai QuickSort tuần tự

A1.2 Triển khai song song với OpenMP

Mã nguồn dưới đây sử dụng OpenMP để tăng tốc thuật toán QuickSort bằng cách chia nhỏ công việc thành các tác vụ (#pragma omp task) và thực thi chúng song song. Mỗi tác vụ sẽ xử lý một nửa của danh sách sau khi được chia tách bằng partition.

```c
void quickSortParallel(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);

        #pragma omp task
        quickSortParallel(arr, low, pi - 1);

        #pragma omp task
        quickSortParallel(arr, pi + 1, high);
    }
}

void quickSort(int arr[], int low, int high) {
    #pragma omp parallel
    {
        #pragma omp single nowait
        quickSortParallel(arr, low, high);
    }
}
```

Listing 2: Triển khai QuickSort song song với OpenMP

A1.3 Triển khai song song với PThreads

Mã nguồn dưới đây sử dụng Pthreads để thực hiện QuickSort song song. Chương trình sẽ tạo hai luồng mới cho mỗi lần đệ quy, với điều kiện depth < 2.

```
void *quicksort(void *args) {
    QuickSortArgs *arg = (QuickSortArgs *)args;
    int *arr = arg->arr;
    int low = arg->left;
    int high = arg->right;
    int depth = arg->depth;

    if (low < high) {
        int pi = partition(arr, low, high);

        QuickSortArgs leftArgs = {arr, low, pi - 1, depth + 1};
        QuickSortArgs rightArgs = {arr, pi + 1, high, depth + 1};

        if (depth < 2) {
            pthread_t left_thread, right_thread;
            pthread_create(&left_thread, NULL, quicksort, &leftArgs);
            pthread_create(&right_thread, NULL, quicksort, &rightArgs)
   ;

            pthread_join(left_thread, NULL);
            pthread_join(right_thread, NULL);
        } else {
            quicksort(&leftArgs);
            quicksort(&rightArgs);
        }
    }
    return NULL;
}
```

Listing 3: Triển khai QuickSort song song với PThreads

A1.4 Triển khai song song với MPI

Thư viện MPI được triển khai với phương pháp chia nhỏ dữ liệu và thực hiện quicksort cục bộ trên từng tiến trình. Mỗi tiến trình xử lý một phần mảng và sử dụng MPI_Scatter để phân phối dữ liệu và MPI_Gather để gom kết quả.

```
int rank, size;
int *arr = NULL;
int local_N, *local_arr;

MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);

if (rank == 0) {
    arr = (int *)malloc(SIZE * sizeof(int));
    srand(time(NULL));
    for (int i = 0; i < SIZE; i++) {
        arr[i] = rand() % SIZE;
    }
}

local_N = SIZE / size;
local_arr = (int *)malloc(local_N * sizeof(int));

MPI_Scatter(arr, local_N, MPI_INT, local_arr, local_N, MPI_INT, 0,
    MPI_COMM_WORLD);

double start = MPI_Wtime();
quicksort(local_arr, 0, local_N - 1);
double end = MPI_Wtime();

MPI_Gather(local_arr, local_N, MPI_INT, arr, local_N, MPI_INT, 0,
    MPI_COMM_WORLD);
MPI_Finalize();
```

Listing 4: Triển khai QuickSort song song với MPI

A2. Hiệu quả vượt trội của Pthreads (mutex, conditional variables)

Chương trình hiện thực thuật toán QuickSort song song bằng cách sử dụng Pthreads để cải thiện hiệu suất trên bộ xử lý đa luồng

Cấu trúc chương trình

- Quản lý luồng bằng Thread Pool: Các luồng làm việc được tạo trước và duy trì, giúp giảm chi phí khi tạo luồng mới

- Cấu trúc dữ liệu chia sẻ: Sử dụng hàng đợi công việc tast_queue để quản lý các nhiệm vụ cần xử lý
- Đồng bộ hóa bằng mutex và Condition variable: Đảm bảo các luồng không truy cập đồng thời vào tài nguyên chung gây xung đột dữ liệu (tránh contention)

Thuật toán Quick sort

- quicksort sequential: Thực hiện sắp xếp khi kích thước nhỏ để tránh chi phí tạo luồng khi không cần thiết.
- quicksort_parallel: Nếu kích thước mảng nhỏ hơn 10000, chuyển sang Quick sort tuần tự. Nếu không, chia mảng thành hai phần và đẩy vào hàng đợi để các luồng xử lý.

worker_thread

- Mỗi luồng đẩy liên tục lấy công việc từ task_queue, thực hiện sắp xếp và tiếp tục chia nhỏ nếu cần.
- Khi hàng đợi trống và không còn luồng nào đang hoạt động, luồng sẽ tự động kết thúc.

```
void quicksort_parallel(int arr[], int low, int high) {
    if (high - low < 10000) {
        quicksort_sequential(arr, low, high);
        return;
    }

    int pi = partition(arr, low, high);

    QuickSortArgs left_task = {arr, low, pi - 1};
    QuickSortArgs right_task = {arr, pi + 1, high};

    pthread_mutex_lock(&mutex);
    task_queue[task_count++] = left_task;
    task_queue[task_count++] = right_task;
    pthread_cond_signal(&cond);
    pthread_mutex_unlock(&mutex);
}
```

Listing 5: Hàm quicksort_parallel thực hiện QuickSort song song bằng cách kiểm tra kích thước mảng. Nếu nhỏ hơn 10.000 phần tử, thuật toán tuần tự được sử dụng. Ngược lại, mảng được chia nhỏ và các công việc được đưa vào hàng đợi để xử lý song song

```
void *worker_thread(void *arg) {
    while (1) {
        pthread_mutex_lock(&mutex);
        while (task_count == 0) {
            if (active_threads == 0) {
                pthread_mutex_unlock(&mutex);
                pthread_exit(NULL);
            }
            pthread_cond_wait(&cond, &mutex);
        }

        QuickSortArgs task = task_queue[--task_count];
        pthread_mutex_unlock(&mutex);

        if (task.left >= task.right) continue;

        int pi = partition(task.arr, task.left, task.right);

        QuickSortArgs left_task = {task.arr, task.left, pi - 1};
        QuickSortArgs right_task = {task.arr, pi + 1, task.right};

        pthread_mutex_lock(&mutex);
        if (task_count + 2 >= task_capacity) {
            task_capacity *= 2;
            task_queue = realloc(task_queue, task_capacity * sizeof(
    QuickSortArgs));
        }
        task_queue[task_count++] = left_task;
        task_queue[task_count++] = right_task;
        pthread_cond_signal(&cond);
        pthread_mutex_unlock(&mutex);
    }
    return NULL;
}
```

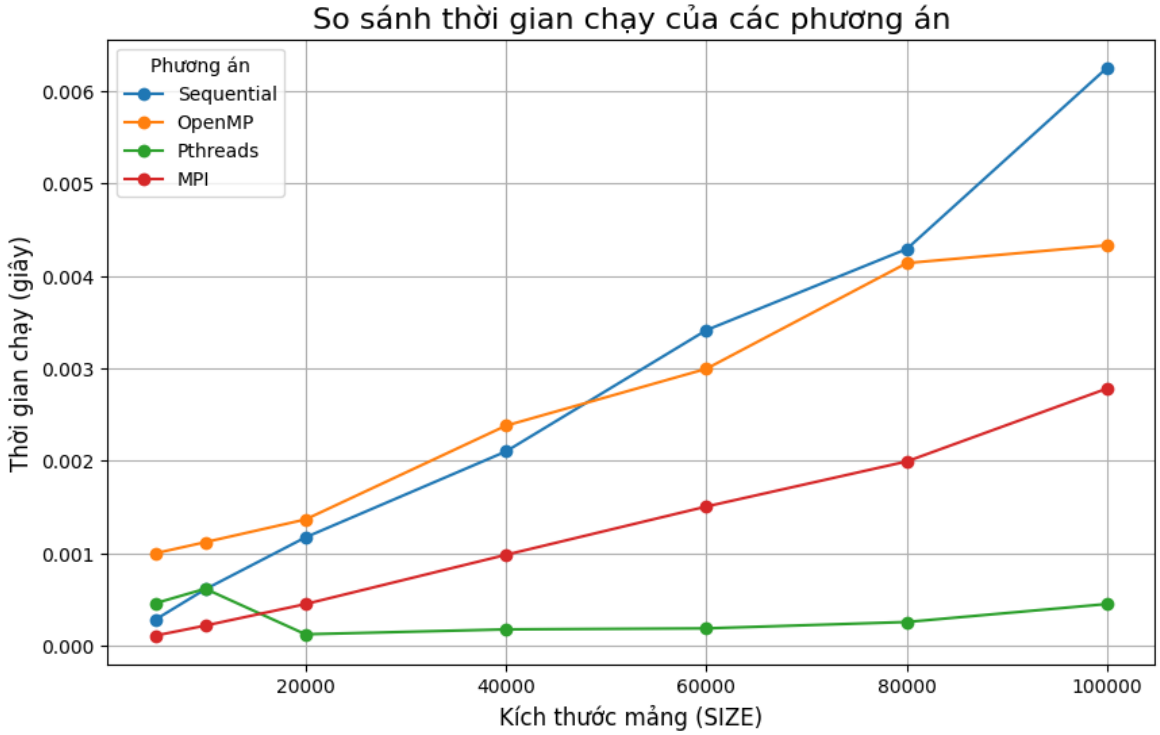
Listing 6: Luồng xử lý chính trong mô hình song song. Hàm liên tục lấy công việc từ hàng đợi, thực hiện phân vùng và đưa các công việc mới vào hàng đợi, đồng thời sử dụng mutex và condition variable để quản lý truy cập

Program	Threads	Time (s)	Speedup	Efficiency
Pthreads	2	0.060582	35.616668	17.808334
Pthreads	4	0.067254	32.083114	8.020778
Pthreads	6	0.064837	33.279115	5.546519
Pthreads	8	0.066228	32.580103	4.072513
Pthreads	12	0.066228	32.580133	2.715011
Sequential	1	2.157719	1.000000	1.000000

Bảng 2: Kết quả chạy thuật toán Pthreads cải tiến với 3.10^7 phần tử trong mảng

Kết quả thực nghiệm cho thấy thuật toán đạt speedup cao khi sử dụng 2 luồng ($\sim$ 35.62x), nhưng khi tăng số luồng lên 4, 6, 8, 12, speedup không tăng đáng kể, trong khi efficiency giảm mạnh. Rõ ràng kết quả của Pthreads sau khi được cải tiến có kết quả vượt trội so với các thư viện khác. Và việc tăng số luồng lên không làm giảm thời gian mà thậm chí làm tăng thời gian thực thi nguyên nhân là do overhead quản lý luồng khi số luồng tăng. Vì vậy, việc tăng số luồng không luôn cải thiện hiệu suất.

A3. Bảng và biểu đồ bổ sung



Hình 4: Ngưỡng mẫu trong mảng khi tuần tự chạy nhanh hơn song song

Hình 4 cho thấy với 5000 phần tử, thuật toán tuần tự chạy nhanh hơn do chi phí tạo và quản lý luồng lớn hơn lợi ích xử lý song song, thậm chi ở 10000 phần tử tuần tự bằng Pthreads, nhanh hơn OpenMP. Điều này nhấn mạnh tầm quan trọng của việc xác định ngưỡng tối ưu để quyết định khi nào nên sử dụng đa luồng, phụ thuộc vào kích thước dữ liệu và đặc điểm phần cứng. Ngoài ra có một số chiến lược phổ biến là dùng độ sâu đệ quy (như ở Listing 3) hoặc kích thước phân vùng để chuyển đổi giữa thuật toán tuần tự và song song. Nếu ngưỡng được chọn hợp lý, hiệu suất có thể cải thiện đáng kể mà không bị ảnh hưởng bởi overhead không cần thiết.

Program	Threads	Time (s)	Speedup	Efficiency
MPI	2	2.600801	1.528679	0.764340
MPI	4	1.585920	2.507555	0.626889
MPI	6	1.098692	3.618011	0.603002
MPI	8	0.856551	4.642117	0.580265
MPI	12	0.580348	6.851928	0.570994
OpenMP	2	3.321953	1.196954	0.598477
OpenMP	4	2.225631	1.786852	0.446713
OpenMP	6	1.742470	2.281068	0.380178
OpenMP	8	1.727225	2.302929	0.287866
OpenMP	12	1.870411	2.126765	0.177230
Pthreads	2	5.663888	0.701981	0.350990
Pthreads	4	5.758324	0.690417	0.172604
Pthreads	6	5.724530	0.694706	0.115784
Pthreads	8	5.726524	0.694451	0.086806
Pthreads	12	5.835334	0.681463	0.056789
Sequential	1	3.976813	1.000000	1.000000

Bảng 3: Kết quả chạy thuật toán Quicksort với 3.10^8 phần tử trong mảng

Khi tăng kích thước dữ liệu từ 3.10^7 lên 3.10^8, thuật toán song song cho thấy hiệu suất cải thiện rõ rệt so với thuật toán tuần tự. MPI đạt speedup tốt nhất, chứng tỏ khả năng chia nhỏ công việc hiệu quả. OpenMP cũng cải thiện nhưng gặp vấn đề khi số luồng cao do overhead chia sẻ bộ nhớ. Pthreads có hiệu suất kém nhất do chi phí quản lý luồng lớn. Kết quả nhấn mạnh tầm quan trọng của việc xác định ngưỡng chuyển đổi giữa thuật toán tuần tự và song song, đặc biệt với dữ liệu nhỏ để tránh overhead không cần thiết.

Program	Threads	Time (s)	Speedup	Efficiency
MPI	2	1.119709	2.061503	1.030752
MPI	4	0.604684	3.817340	0.954335
MPI	6	0.409924	5.631007	0.938501
MPI	8	0.326955	7.059956	0.882494
MPI	12	0.233254	9.895999	0.824667
OpenMP	2	1.532643	1.506081	0.753041
OpenMP	4	0.852007	2.709231	0.677308
OpenMP	6	0.662649	3.483418	0.580570
OpenMP	8	0.616845	3.742082	0.467760
OpenMP	12	0.667834	3.456374	0.288031
Pthreads	2	2.563924	0.900294	0.450147
Pthreads	4	2.566550	0.899373	0.224843
Pthreads	6	2.556135	0.903037	0.150506
Pthreads	8	2.550682	0.904968	0.113121
Pthreads	12	2.544657	0.907110	0.075593
Sequential	1	2.308285	1.000000	1.000000

Bảng 4: Hiệu suất thuật toán Quicksort với các cài đặt song song (sắp xếp giảm dần)

Thời gian tính toán, speed up, hiệu suất sắp xếp giảm dần không khác biệt gì so với thời gian sắp xếp tăng dần.