ĐẠI HỌC QUỐC GIA THÀNH PHỐ HỒ CHÍ MINH
TRƯỜNG ĐẠI HỌC BÁCH KHOA
KHOA KHOA HỌC VÀ KỸ THUẬT MÁY TÍNH

Hệ điều hành

Báo cáo Lab 3

Tính xấp xỉ số Pi dùng Single-threaded, Multi-threaded và Shared Variable Program

---

Giáo viên hướng dẫn:	Hoàng Lê Hải Thanh	
Sinh viên thực hiện:	Mai Xuân Nhựt	2312549
	Nguyễn Thái Sơn	2312968
	Lê Đức Tài	2312995

THÀNH PHỐ HỒ CHÍ MINH, 01/12/2025

Mục lục

1 Approach 1: A Single-Thread program

```
unsigned int base_seed = 123456;
double worker_single_thread(long int npoints, unsigned int seed){
    long inside = 0L;
    for (long int i = 0; i < npoints; i++) {
        unsigned int seed = base_seed + i; // Reset seed cho tung diem
        double x = rand_double(&seed, -1.0 , 1.0);
        double y = rand_double(&seed, -1.0 ,1.0);
        if (x * x + y * y <= 1.0) ++inside;
    }
    double pi = 4.0 * inside / npoints;
    return pi;
}
```

2 Approach 2: A Multi-Thread Program

2.1 Mục tiêu

Ở **Approach 1: Single-Thread**, chúng ta đã xây dựng chương trình đơn luồng để ước lượng giá trị π bằng phương pháp Monte Carlo. Tuy nhiên, với số điểm lớn (ví dụ: $10^7, 10^8, 10^9$), thời gian tính toán trở nên rất lâu do tất cả công việc được thực hiện tuần tự trên một luồng duy nhất.
Trong **Approach 2**, chúng ta áp dụng kỹ thuật **đa luồng (multi-threading)** để:

- Chia nhỏ công việc thành nhiều phần và gán cho các luồng chạy song song.
- Mỗi luồng xử lý một phần số điểm, đếm số điểm rơi vào hình tròn trên **biến cục bộ riêng**.
- Sau khi tất cả luồng hoàn thành, tổng hợp kết quả từ các luồng để tính giá trị π.
- Đo lường **speedup** so với Approach 1 với nhiều lựa chọn số luồng (từ 2 đến 100 hoặc 1000 luồng).

2.2 Thuật toán

Trong phiên bản đa luồng với biến cục bộ:

- Chương trình tạo `nThreads` luồng.
- Mỗi luồng được gán một đoạn chỉ số điểm $[start\_idx, start\_idx + num\_points)$ không trùng nhau:
  - Số điểm mỗi luồng xử lý:
$$\texttt{points\_per\_thread} = \left\lfloor \frac{\texttt{nPoints}}{\texttt{nThreads}} \right\rfloor$$
  - Số điểm dư:
$$\texttt{remaining} = \texttt{nPoints} \bmod \texttt{nThreads}$$
  - Các luồng đầu tiên sẽ nhận thêm 1 điểm từ phần dư:
$$\texttt{num\_points}_i = \begin{cases} \texttt{points\_per\_thread} + 1 & \text{nếu } i < \texttt{remaining}, \\ \texttt{points\_per\_thread} & \text{ngược lại.} \end{cases}$$

– Chỉ số bắt đầu của mỗi luồng:

$$\texttt{start\_idx}_i = \sum_{j=0}^{i-1} \texttt{num\_points}_j$$

- Mỗi luồng sử dụng **biến đếm cục bộ** `points_in_circle` để đếm số điểm rơi vào hình tròn.
- Mỗi điểm được sinh dựa trên một seed phụ thuộc vào chỉ số toàn cục:

$$\texttt{seed} = \texttt{base\_seed} + \texttt{start\_idx} + i$$

- Sau khi tất cả luồng hoàn thành, luồng chính tổng hợp kết quả:

$$\texttt{total\_in\_circle} = \sum_{i=0}^{\texttt{nThreads}-1} \texttt{points\_in\_circle}_i$$

- Giá trị π được ước lượng:

$$\pi \approx 4 \cdot \frac{\texttt{total\_in\_circle}}{\texttt{nPoints}}$$

2.3 Cấu trúc dữ liệu

Mỗi luồng nhận một cấu trúc dữ liệu chứa thông tin cần thiết:

```
typedef struct {
    long long start_idx;      // Chi so diem bat dau
    long long num_points;     // So diem can xu ly
    unsigned int base_seed;   // Seed co so
    long long points_in_circle; // Ket qua
} thread_data_t;
```

2.4 Hàm worker của mỗi luồng

```
void *monte_carlo_worker(void *arg) {
    thread_data_t *data = (thread_data_t *)arg;
    long long count = 0;

    for (long long i = 0; i < data->num_points; i++) {
        unsigned int seed = data->base_seed
                          + data->start_idx + i;
        double x = rand_double(&seed, -1.0, 1.0);
        double y = rand_double(&seed, -1.0, 1.0);
        if (x * x + y * y <= 1.0) count++;
    }

    data->points_in_circle = count;
    return NULL;
}
```

2.5 Ưu điểm của Approach 2

2.5.1 Không cần cơ chế đồng bộ (mutex)

Đây là ưu điểm lớn nhất của Approach 2:

- Mỗi luồng chỉ ghi vào biến cục bộ riêng của mình (`points_in_circle`), được lưu trong cấu trúc `thread_data_t` độc lập.
- Không có tranh chấp tài nguyên (*race condition*) vì các luồng không truy cập cùng một vùng nhớ trong quá trình tính toán.
- Loại bỏ hoàn toàn chi phí của việc lock/unlock mutex trong vòng lặp chính.
- Việc tổng hợp kết quả chỉ diễn ra một lần duy nhất sau khi tất cả luồng hoàn thành, thông qua `pthread_join`.

2.5.2 Hiệu năng cao và khả năng mở rộng tốt

- Tính toán song song thực sự: Các luồng chạy hoàn toàn độc lập, không phụ thuộc lẫn nhau, không chờ đợi tài nguyên chia sẻ.
- Overhead thấp: Không có chi phí đồng bộ liên tục trong vòng lặp. Chi phí duy nhất là:
  - Khởi tạo và hủy luồng (`pthread_create`, `pthread_join`)
  - Context switching giữa các luồng (do hệ điều hành quản lý)
- Khả năng mở rộng tuyến tính: Với số nhân CPU là P, speedup lý tưởng có thể đạt gần P lần.

2.5.3 Đơn giản và dễ bảo trì

- Logic rõ ràng: Quy trình gồm 3 bước đơn giản:
  1. Chia công việc: Phân chia N điểm cho T luồng
  2. Tính toán song song: Mỗi luồng xử lý phần của mình độc lập
  3. Tổng hợp kết quả: Cộng dồn các kết quả cục bộ
- Dễ debug: Không có vấn đề về race condition hay deadlock, các lỗi thường liên quan đến logic phân chia công việc.
- Dễ mở rộng: Có thể dễ dàng thay đổi số luồng mà không cần sửa đổi logic chính.
- Mô hình phổ biến: Đây là mô hình "divide and conquer"song song (parallel reduction), được sử dụng rộng rãi trong lập trình song song.

2.5.4 Tận dụng tối đa phần cứng

- CPU-bound workload: Monte Carlo là bài toán thuần tính toán, không có I/O, rất phù hợp với đa luồng.
- Cache-friendly: Mỗi luồng làm việc với dữ liệu riêng, giảm thiểu cache invalidation và false sharing.
- Load balancing tự động: Nếu phân chia công việc đều, mọi luồng hoàn thành gần cùng lúc, tận dụng tối đa tài nguyên CPU.

3 Approach 3: Shared Variables

3.1 Mục tiêu

Ở 2 cách trước là **Approach 1: Single-Thread** và **Approach 2: Multi-Thread với biến cục bộ**, chúng ta đã tìm hiểu về cách sử dụng biến cục bộ để đếm số điểm rơi vào hình tròn rồi cộng dồn vào tổng khi kết thúc. Trong phần này, chúng ta sẽ khám phá cách tiếp cận thứ ba, đó là **Shared Variables**, trong đó các luồng cập nhật trực tiếp một biến toàn cục dùng chung.
Trong cách 3 yêu cầu:

- Sử dụng biến toàn cục để lưu trữ số điểm rơi vào hình tròn
- Các luồng (thread) **cập nhật trực tiếp** biến toàn cục này mỗi khi luồng thực thi kết thúc.
- Sử dụng cơ chế **mutex locks** để tránh tình trạng **Race Condition** khi nhiều luồng cùng truy cập và cập nhật biến toàn cục.

3.2 Thuật toán

Trong phiên bản đa luồng với biến chia sẻ:

- Chương trình tạo `nThreads` luồng.
- Mỗi luồng được gán một đoạn chỉ số điểm $[start\_idx, end\_idx)$ không trùng nhau:
  - $$\texttt{start\_idx} = \texttt{thread\_id} \cdot \frac{\texttt{nPoints}}{\texttt{nThreads}}$$
  - $$\texttt{end\_idx} = \begin{cases} \texttt{start\_idx} + \frac{\texttt{nPoints}}{\texttt{nThreads}} & \text{nếu không phải luồng cuối,} \\ \texttt{nPoints} & \text{nếu là luồng cuối.} \end{cases}$$
- Mỗi điểm được sinh dựa trên một seed phụ thuộc vào chỉ số toàn cục:
  $$\texttt{seed} = \texttt{base\_seed} + i$$
- Sử dụng mutex locks để tránh race condition khi cập nhật biến toàn cục.

3.3 Tránh race condition bằng mutex

Do `global_count` là biến chia sẻ, nếu nhiều luồng cùng thực hiện:

```
global_count++;
```

mà không có cơ chế đồng bộ, chương trình sẽ gặp *race condition*: các luồng đọc cùng một giá trị cũ, cùng tăng rồi ghi đè lẫn nhau, khiến giá trị cuối cùng của `global_count` bị thiếu hoặc sai. Điều này dẫn tới kết quả xấp xỉ π bị sai lệch.
Để tránh hiện tượng đó, mỗi lần cập nhật `global_count` phải được bảo vệ bởi mutex:

```
pthread_mutex_lock(&lock);
global_count++;
pthread_mutex_unlock(&lock);
```

Như vậy, tại mọi thời điểm chỉ có một luồng được phép vào vùng cập nhật, đảm bảo kết quả cuối cùng là đúng.

3.4 Đề xuất giải pháp

Một nhược điểm của cách tiếp cận 3 là:

- Mỗi lần có điểm rơi vào hình tròn, luồng phải thực hiện thao tác lock và unlock mutex
- Với số điểm lớn lên đến $10^7, 10^8, 10^9$ và nhiều luồng, việc lock/unlock liên tục dẫn đến overhead nặng nề

Để **giảm overhead** mà vẫn giữ ý tưởng sử dụng biến chia sẻ `global_count`, có thể đề xuất giải pháp:

- Mỗi luồng sử dụng một biến đếm cục bộ `local_count`.
- Trong vòng lặp, luồng chỉ tăng `local_count` (không lock mutex liên tục).
- Sau khi hoàn thành tất cả các điểm được gán, luồng mới:

  ```
  pthread_mutex_lock(&lock);
  global_count += local_count;
  pthread_mutex_unlock(&lock);
  ```

- Như vậy, mỗi luồng chỉ lock/unlock **một lần** thay vì hàng triệu lần, giảm đáng kể chi phí đồng bộ.

Cách làm này vẫn giữ đúng yêu cầu "dùng biến chia sẻ `global_count`", đồng thời:

- Tránh race condition nhờ mutex.
- Giảm overhead đồng bộ, giúp Cách 3 tiến gần hơn về hiệu năng so với Cách 2.

4 Điều chỉnh cơ chế sinh số ngẫu nhiên để tăng độ chính xác

Trong bài toán Monte Carlo, chất lượng và cách sử dụng số ngẫu nhiên ảnh hưởng trực tiếp đến:

- Độ **ổn định** của kết quả giữa các lần chạy.
- Độ **chính xác** của giá trị xấp xỉ π khi số điểm `nPoints` tăng lên.

Sử dụng seed cố định và phụ thuộc vào chỉ số toàn cục
Thay vì khởi tạo seed từ thời gian hệ thống (`time(NULL)`), nhóm sử dụng một *seed gốc* cố định `base_seed` và xây dựng seed cho từng điểm dựa trên chỉ số toàn cục i:

```
unsigned int seed = base_seed + (unsigned int)i;
```

Nhờ đó:

- Với mỗi điểm sẽ có 1 seed duy nhất, giúp phân tán tốt hơn các điểm ngẫu nhiên.
- Với cùng giá trị `nPoints`, mỗi chỉ số i luôn tạo ra cùng một cặp (x, y), không phụ thuộc vào số luồng hay cách phân chia công việc.
- Tập các điểm ngẫu nhiên được sinh ra trong phiên bản đơn luồng và đa luồng là như nhau (chỉ khác cách phân chia giữa các luồng).
- Chạy lại chương trình nhiều lần với cùng cấu hình luôn cho cùng một kết quả xấp xỉ π, giúp việc so sánh, đo *speedup* và phân tích trở nên đáng tin cậy hơn.

Nhờ việc "làm mềm" cơ chế ngẫu nhiên theo hướng *deterministic nhưng vẫn đủ phân tán*, nhóm vừa đảm bảo được tính ngẫu nhiên cho thuật toán Monte Carlo, vừa đảm bảo được tính lặp lại và độ chính xác cao cho kết quả thực nghiệm.