Phương pháp 1:

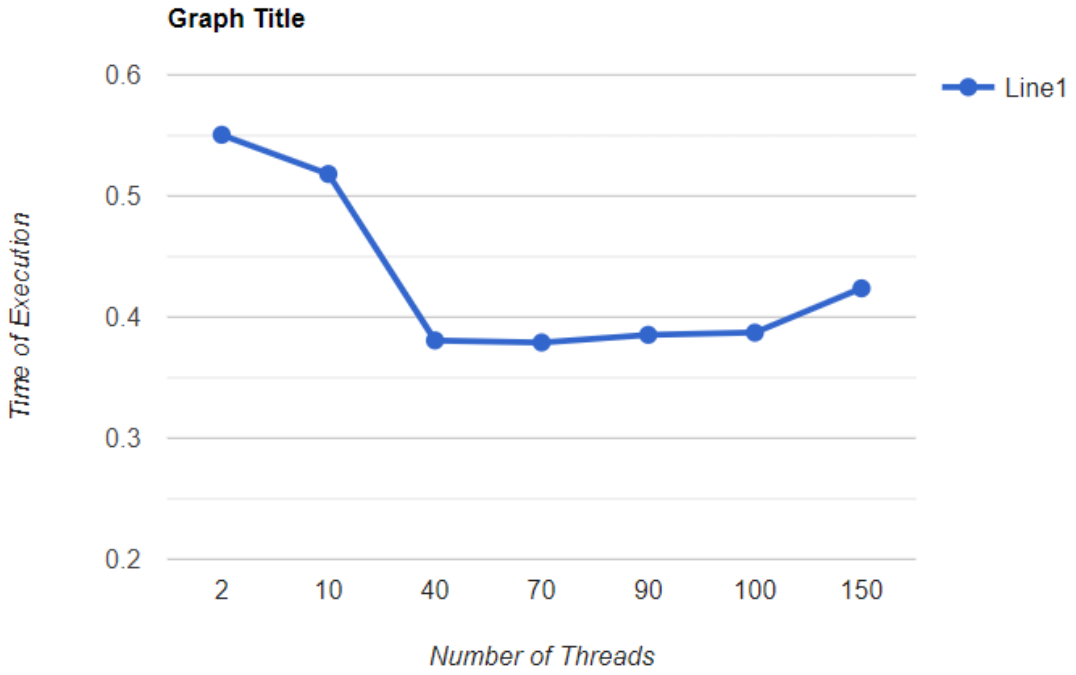
```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>

void circle_point();
static long int Npoint;
static long int count_circle=0;
clock_t start_time, end_time;

int main(){
  Npoint=100000000;

  start_time=clock();
  circle_point();
  double pi=4.0*(double)count_circle/(double)Npoint;
  end_time=clock();

  printf("PI = %17.15f\n",pi);
  printf("Time to compute= %g second\n",(double)(end_time-
start_time)/CLOCKS_PER_SEC);
  return 0;
}

void circle_point(){
  srand(time(NULL));
```

```
  int i;
  for(i=0;i<Npoint;i++){
        double x= (double)rand()/(double)RAND_MAX;
        double y=(double)rand()/(double)RAND_MAX;
        double r= sqrt(x*x+y*y);
        if(r<=1) count_circle+=1;
        }
}
```

Phương pháp 2:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <pthread.h>
#define NUM_THREAD 1000

void* circle_point(void *param);

pthread_t tid[NUM_THREAD];
int count[NUM_THREAD]={0};
clock_t start_time, end_time;
static long int Npoint;
static long int count_circle=0;

int main(){
  Npoint=100000000/NUM_THREAD;

  start_time= clock();
  srand(time(NULL));
  static int i;
  for(i=0; i<NUM_THREAD;i++) {
        pthread_create(&tid[i],NULL,circle_point,&count[i]);
  }

 long int t;
  for(i=0;i<NUM_THREAD;i++){
```

```
        pthread_join(tid[i],NULL);
        count_circle+=count[i];
  }
  double pi=4.0*(double)count_circle/(double)Npoint/(double)NUM_THREAD;
  end_time=clock();

  printf("PI = %17.15f\n",pi);
printf("Time to compute= %g second\n",(double)(end_time-
start_time)/CLOCKS_PER_SEC);
  return 0;
 }

 void* circle_point(void *param){
  int *pcount= (int*)param;
  int i;
  for(i=0; i<Npoint;i++){
        double x= (double)rand()/(double)RAND_MAX;
        double y=(double)rand()/(double)RAND_MAX;
        double r= sqrt(x*x+y*y);
        if(r<=1) *pcount=*pcount+1;
        }
  pthread_exit(0);
 }
```


Graph Title
0.6
0.5
0.4
0.3
0.2
Time of Execution
2
10
40
70
90
100
150
Number of Threads
Line1

Phương pháp 3:

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <math.h>
#include <pthread.h>

#define NUM_THREAD 100

void* circle_point(void *param);

pthread_t tid[NUM_THREAD];
clock_t start_time, end_time;
static long int Npoint;
static long int count_circle=0;

int main(){
  Npoint=10000000/NUM_THREAD;

  start_time= clock();
  srand(time(NULL));
  static int i;
  for(i=0; i<NUM_THREAD;i++) {
        pthread_create(&tid[i],NULL,circle_point,&count_circle);
  }

 long int t;
```

```
  for(i=0;i<NUM_THREAD;i++){
        pthread_join(tid[i],NULL);
  }
  double pi=4.0*(double)count_circle/(double)Npoint/(double)NUM_THREAD;
  end_time=clock();

  printf("PI = %17.15f\n",pi);
  printf("Time to compute= %g second\n",(double)(end_time-
start_time)/CLOCKS_PER_SEC);
  return 0;
 }

void* circle_point(void *param){
  int *pcount= (int*)param;
  int i;
  for(i=0; i<Npoint;i++){
        double x= (double)rand()/(double)RAND_MAX;
        double y=(double)rand()/(double)RAND_MAX;
        double r= sqrt(x*x+y*y);
        if(r<=1) *pcount=*pcount+1;
        }
  pthread_exit(0);
}
```

Nhận xét:

	Approach 1	**Approach 2**	**Approach 3**

Thời gian thực thi	Lớn Tạo ra một lượng lớn các điểm ngẫu nhiên và xử lý với toàn bộ điểm ngẫu nhiên đó trong một thread. Thời gian thực thi của phương pháp này tăng đáng kể với số lượng điểm ngẫu nhiên lớn do t**ính tuần tự** của việc tính toán,	Nhỏ Chia nhỏ công việc và giao cho nhiều luồng khác nhau, mỗi luồng tính toán một phần nhỏ của giá trị pi, từ đó làm giảm thời gian thực thi so với phương pháp đơn luồng.	Trung bình Tương tự như phương pháp Multi-Thread, nhưng sử dụng biến chia sẻ và mutex để đảm bảo tính nhất quán và tránh lỗi cạnh tranh dữ liệu. Thời gian thực thi không bé hơn Multi-Thread vì đặc tính của phương pháp Mutex Lock.
Ưu điểm	Đơn giản và dễ hiện thực	Hiệu suất được cải thiện nhờ vào việc phân chia nhiều luồng	Có thể tăng hiệu suất so với phương pháp Multi-Thread nếu được triển khai chính xác và áp dụng mutex lock hiệu quả.
Nhược điểm	Thời gian thực thi có thể rất lâu nếu số lượng điểm ngẫu nhiên lớn.	Khó hiện thực hơn so với single-thread. Cần phải quản lý và đồng bộ hóa sự chia sẻ dữ liệu giữa các luồng để tránh lỗi cạnh tranh dữ liệu.	Cần quản lý mutex để tránh lỗi cạnh tranh dữ liệu. Phát sinh chi phí quản lý đồng bộ hóa với mutex.

- Phương pháp 2 (Đa luồng) cung cấp hiệu suất tốt hơn so với Phương pháp 1 (Đơn luồng) do tính đồng thời (song song) của tính toán.

- Phương pháp 3 (Biến Chia Sẻ) có thể cải thiện hiệu suất hơn nữa bằng cách giảm thiểu overhead của việc duy trì biến đếm riêng cho mỗi luồng, nhưng đồng thời giới thiệu thêm overhead đồng bộ hóa do sử dụng khóa mutex.

Như vậy có thể nói rằng: việc thực hiện đa luồng và phiên bản cải thiện với biến chia sẻ có thể cải thiện đáng kể hiệu suất trong các mô phỏng Monte Carlo trong việc tính

gần đúng số pi, nhưng lại phát sinh thêm các vấn đề đồng bộ và cần xem xét kỹ lưỡng các cơ chế đồng bộ hóa và tối ưu hóa. là cần thiết để đạt được tính mở rộng và hiệu suất tối ưu.

Propose a solution to reduce the overhead of this technique without removing the shared variable.

1. **Fine-grained locking:**

Thay vì sử dụng một mutex duy nhất để bảo vệ toàn bộ biến chia sẻ, bạn có thể sử dụng nhiều mutex nhỏ hơn để bảo vệ các phần nhỏ của dữ liệu. Ví dụ, nếu biến chia sẻ là một mảng, bạn có thể sử dụng một mutex cho mỗi phần tử của mảng.

2. **Lock-free data structures:**

Sử dụng cấu trúc dữ liệu không cần khóa để lưu trữ dữ liệu chia sẻ. Ví dụ, bạn có thể sử dụng hàng đợi không khóa hoặc vùng nhớ chia sẻ được thiết kế đặc biệt để tránh việc sử dụng mutex và giảm thiểu overhead.

3. **Atomic operations:**

Sử dụng các hoạt động nguyên tử để thực hiện các thay đổi trên biến chia sẻ mà không cần sử dụng mutex. Các hoạt động nguyên tử đảm bảo rằng các hoạt động đọc và ghi được thực hiện nguyên tử, giảm thiểu nguy cơ của race condition.

4. **Thread-local storage:**

Nếu có thể, hãy cân nhắc việc sử dụng bộ nhớ cục bộ của từng luồng để lưu trữ dữ liệu riêng tư của từng luồng mà không cần chia sẻ. Điều này giúp loại bỏ hoặc giảm thiểu sự phụ thuộc vào biến chia sẻ và mutex.

5. **Batch processing:**

Thay vì cập nhật biến chia sẻ sau mỗi lần lặp, bạn có thể tích hợp các cập nhật và chỉ cập nhật biến chia sẻ sau một số lần lặp nhất định. Điều này giảm số lượng lần khóa và mở khóa mutex, giảm thiểu overhead của việc đồng bộ hóa.