# **Các chỉ dẫn của thư viện OpenPM (omp.h)**

# **1. Các lệnh chỉ dẫn**

## **1. #pragma omp parallel**

Chỉ dẫn này tạo ra một vùng song song, trong đó nhiều luồng có thể chạy đồng thời.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  #pragma omp parallel  {  int thread\_id = omp\_get\_thread\_num();  std::cout << "Hello from thread " << thread\_id << std::endl;  }  return 0;  } |

## **2. #pragma omp for**

Chỉ dẫn này chia sẻ một vòng lặp giữa các luồng. Nó phải được sử dụng trong một vùng song song.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  const int n = 4;  int a[n];  #pragma omp parallel for  for (int i = 0; i < n; ++i) {  a[i] = i \* i;  }  // In mảng a  for (int i = 0; i < n; ++i) {  std::cout << a[i] << " ";  }  return 0;  } |
| #include <iostream>  #include <omp.h>  int main() {  const int n = 10;  int a[n];  // Khởi động vùng song song  #pragma omp parallel  {  // Tách lệnh for vào bên trong vùng song song  #pragma omp for  for (int i = 0; i < n; ++i) {  a[i] = i \* i;  }  }  // In mảng a  for (int i = 0; i < n; ++i) {  std::cout << a[i] << " ";  }  return 0;  } |

## **3. #pragma omp parallel for reduction**

Chỉ dẫn này thực hiện phép toán giảm (reduction) trên một biến, tổng hợp kết quả từ tất cả các luồng.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  const int n = 10;  int sum = 0;  #pragma omp parallel for reduction(+:sum)  for (int i = 1; i <= n; ++i) {  sum += i;  }  std::cout << "Sum of first " << n << " natural numbers is " << sum << std::endl;  return 0;  } |
|  |

## **4. #pragma omp sections**

Chỉ dẫn này cho phép định nghĩa các phần riêng biệt trong một vùng song song, cho phép mỗi luồng xử lý một phần riêng.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  #pragma omp parallel  {  #pragma omp sections  {  #pragma omp section  {  std::cout << "Section 1 executed by thread " << omp\_get\_thread\_num() << std::endl;  }  #pragma omp section  {  std::cout << "Section 2 executed by thread " << omp\_get\_thread\_num() << std::endl;  }  }  }  return 0;  } |

## **5. #pragma omp single**

Chỉ dẫn này đảm bảo rằng chỉ có một luồng thực hiện một khối mã nhất định, trong khi các luồng khác chờ.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  #pragma omp parallel  {  #pragma omp single  {  std::cout << "This message is printed by a single thread: " << omp\_get\_thread\_num() << std::endl;  }  }  return 0;  } |

## **6. #pragma omp barrier**

Chỉ dẫn này đồng bộ hóa tất cả các luồng trong một vùng song song, đảm bảo rằng tất cả đều hoàn thành một công việc nhất định trước khi tiếp tục.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  // Đặt số lượng luồng sử dụng  omp\_set\_num\_threads(4); // Sử dụng 4 luồng  #pragma omp parallel  {  std::cout << "Thread " << omp\_get\_thread\_num() << " is at the barrier." << std::endl;  #pragma omp barrier  std::cout << "Thread " << omp\_get\_thread\_num() << " has passed the barrier." << std::endl;  }  return 0;  } |

# **2. Các lệnh chỉ dẫn cho luồng chính và thực hiện đồng bộ hóa**

### **2.1.** master

* Master thread là luồng chính (thread) khởi tạo trong chương trình OpenMP. Khi một đoạn mã được đánh dấu để chạy song song (parallel region), OpenMP sẽ tạo ra nhiều luồng con (worker threads) để thực hiện các công việc song song, nhưng master thread vẫn giữ vai trò kiểm soát và điều phối.
* Trong một parallel region, tất cả các luồng sẽ thực hiện các công việc được phân chia, nhưng có những trường hợp chỉ master thread thực hiện một số đoạn mã cụ thể, thường là những đoạn mã không cần chạy song song hoặc cần được thực thi duy nhất bởi luồng chính.

2. Cấu trúc lệnh master trong OpenMP

|  |
| --- |
| #pragma omp master { // Đoạn mã này chỉ được thực thi bởi master thread } |

Ví dụ:

|  |
| --- |
| #include <omp.h>  #include <iostream>  int main() {  // Bắt đầu vùng song song  #pragma omp parallel  {  int thread\_id = omp\_get\_thread\_num();  // Mỗi thread in ra thông báo, bao gồm cả master thread  std::cout << "Thread " << thread\_id << " is working." << std::endl;  // Chỉ master thread thực thi đoạn mã sau  #pragma omp master  {  std::cout << "Master thread is managing." << std::endl;  }  }  return 0;  } |

### 2.2. **critical**

**critical** là một chỉ thị quan trọng để đảm bảo tính đồng bộ khi nhiều luồng (threads) có thể truy cập và thay đổi cùng một tài nguyên hoặc vùng bộ nhớ chung. Điều này giúp tránh các vấn đề như **race condition** (trạng thái tranh chấp tài nguyên), nơi hai hoặc nhiều luồng cùng cố gắng ghi dữ liệu vào một biến hoặc cấu trúc dữ liệu, dẫn đến kết quả không mong muốn.

* **#pragma omp critical** là một chỉ thị trong OpenMP dùng để giới hạn truy cập vào một đoạn mã cụ thể sao cho chỉ có một luồng được thực thi tại một thời điểm. Khi một luồng vào vùng **critical**, các luồng khác sẽ phải đợi cho đến khi luồng này rời khỏi vùng đó, đảm bảo tính đồng bộ và tránh sự cố tranh chấp tài nguyên.
* Đoạn mã trong vùng **critical** thường là các thao tác mà bạn không muốn thực hiện song song, ví dụ như thay đổi giá trị của một biến chung hoặc ghi dữ liệu vào file.

**Cú pháp của critical trong OpenMP**

#pragma omp critical

{

// Đoạn mã này chỉ được thực thi bởi một luồng tại một thời điểm

}

* Khi một luồng gặp chỉ thị **critical**, nó sẽ kiểm tra xem có luồng nào khác đang ở trong vùng **critical** không. Nếu không, nó sẽ tiến hành thực thi đoạn mã bên trong. Nếu có, nó sẽ đợi cho đến khi vùng đó được giải phóng.

|  |
| --- |
| #include <omp.h>  #include <iostream>  int main() {  int sum = 0;  // Bắt đầu vùng song song với nhiều luồng  #pragma omp parallel  {  int thread\_id = omp\_get\_thread\_num();  // Mỗi thread cố gắng tăng giá trị biến sum  #pragma omp critical  {  sum += thread\_id;  std::cout << "Thread " << thread\_id << " is updating sum. Current sum: " << sum << std::endl;  }  }  std::cout << "Final sum: " << sum << std::endl;  return 0;  } |

### 2.3. atomic

**atomic** là một chỉ thị giúp thực hiện các thao tác trên biến chung một cách an toàn và hiệu quả khi có nhiều luồng (threads) cùng truy cập, nhưng với chi phí thấp hơn so với chỉ thị **critical**. **atomic** thường được sử dụng cho các phép toán đơn giản như tăng, giảm, hoặc gán giá trị và đảm bảo rằng các thao tác này xảy ra một cách **nguyên tử** (atomic), nghĩa là không bị gián đoạn bởi các luồng khác.

 **#pragma omp atomic** bảo đảm rằng một thao tác đơn lẻ trên biến chung được thực hiện một cách nguyên tử, tức là một luồng thực hiện thao tác đó mà không bị ngắt quãng bởi các luồng khác. Điều này giúp tránh các lỗi do nhiều luồng cố gắng cập nhật cùng một biến chung dẫn đến **race condition** (tranh chấp tài nguyên).

 Chỉ thị **atomic** chỉ có hiệu quả với các phép toán đơn giản như tăng/giảm giá trị của biến, gán giá trị, hoặc thực hiện các phép toán cộng/trừ/multiplication trên biến.

|  |
| --- |
| #include <omp.h>  #include <iostream>  int main() {  int sum = 0;  // Bắt đầu vùng song song với nhiều luồng  #pragma omp parallel  {  int thread\_id = omp\_get\_thread\_num();  // Tăng giá trị biến sum một cách nguyên tử  #pragma omp atomic  sum += 1;    std::cout << "Thread " << thread\_id << " incremented sum. Current sum: " << sum << std::endl;  }  std::cout << "Final sum: " << sum << std::endl;  return 0;  } |

# **Một số ví dụ**

## Ví dụ 1

|  |
| --- |
| #include <iostream>  #include <omp.h>  #include <cstdlib>  #include <ctime>  int main() {  // Khởi tạo bộ sinh số ngẫu nhiên  std::srand(std::time(0));  double b = std::rand() % 100 + 1; // Random b từ 1 đến 100  double c = std::rand() % 100 + 1; // Random c từ 1 đến 100  double results[4]; // Mảng để lưu kết quả  omp\_set\_num\_threads(4); // Đặt số lượng luồng  #pragma omp parallel  {  int thread\_id = omp\_get\_thread\_num();  switch (thread\_id) {  case 0:  results[0] = b + c;  std::cout << "Luong 0: a = b + c = " << results[0] << std::endl;  break;  case 1:  results[1] = b \* c;  std::cout << "Luong 1: a = b \* c = " << results[1] << std::endl;  break;  case 2:  results[2] = b / c;  std::cout << "Luong 2: a = b / c = " << results[2] << std::endl;  break;  case 3:  results[3] = b - c;  std::cout << "Luong 3: a = b - c = " << results[3] << std::endl;  break;  }  std::cout << std::endl;  }  return 0;  } |

Ví dụ 2. Tính tổng của dãy số gồm 10000000 số tự nhiên ngẫu nhiên. 4 thread đầu dành cho tính tổng, thread 5 dành cho sắp xếp dãy. Viết chương trình và so sánh thời gian tính toán với thuật toán tính toán tuần tự và thuật toán tính toán song song với OpenMP.

# **C. MỘT SỐ THUẬT TOÁN PHỔ BIẾN SỬ DỤNG OPENMP ĐỂ TẬN DỤNG KHẢ NĂNG XỬ LÝ SONG SONG:**

# **1. Nhân ma trận (Matrix Multiplication)**

* **Mô tả**: Sử dụng OpenMP để phân chia phép nhân ma trận thành các khối nhỏ và thực hiện song song. Ví dụ: nhân ma trận A với ma trận B để tạo ra ma trận C.
* **Ứng dụng**: Được sử dụng rộng rãi trong khoa học máy tính, đồ họa, và các mô hình tính toán khoa học.

### ***Thuật toán Fox (C=AxB)***

Để xác định kích thước khối (block size) tối ưu cho thuật toán Fox khi nhân hai ma trận C(n,n)=A(n,n)×B(n,n)C(n,n) = A(n,n) \times B(n,n)C(n,n)=A(n,n)×B(n,n), ta cần xem xét nhiều yếu tố như:

1. **Kích thước ma trận (nnn)**: Số lượng phần tử trong ma trận càng lớn thì việc chia ma trận thành các khối nhỏ để thực hiện song song càng quan trọng.
2. **Số lượng lõi (cores) và bộ xử lý**: Thuật toán Fox chia ma trận thành các khối và phân phối công việc cho các bộ xử lý. Vì vậy, số lượng khối chia nên phù hợp với số lượng lõi của hệ thống (số lõi vật lý hoặc ảo của CPU).
3. **Bộ nhớ đệm (cache)**: Kích thước khối phù hợp phải tận dụng tối đa bộ nhớ đệm của hệ thống để giảm thời gian truy cập bộ nhớ chính. Các khối ma trận nhỏ hơn thường phù hợp với bộ nhớ đệm và do đó có thể giảm "cache miss."

**Cách xác định kích thước khối tối ưu**

* **Số lõi xử lý (P)**: Nếu hệ thống có PPP lõi xử lý, thì chia n×nn \times nn×n ma trận thành PPP khối để đảm bảo phân phối đều công việc cho các lõi.
* **Kích thước khối (block size)**: Để tối ưu hóa hiệu suất, kích thước khối thường được chọn sao cho mỗi khối vừa với bộ nhớ đệm (cache) của bộ xử lý. Điều này giúp giảm thời gian truy xuất dữ liệu từ bộ nhớ chính.

**Ví dụ về kích thước khối:**

Giả sử hệ thống có 4 lõi xử lý (P = 4) và ma trận có kích thước n=thì:

Sqrt(P) =2

Kích thước khối (block size) sẽ là:

block size=1000/2=500

Tức là mỗi khối sẽ có kích thước 500×500

Nếu hệ thống có nhiều lõi xử lý hơn, ví dụ P=16 (16 cores), thì:

Sqrt(P) =4

Kích thước khối sẽ là:

block size=1000/4=250

Mỗi khối sẽ có kích thước 250×250

**Lời khuyên về block size:**

* **Quá nhỏ**: Nếu kích thước khối quá nhỏ, sẽ có nhiều lần truyền thông tin giữa các khối, tăng chi phí giao tiếp và làm chậm thuật toán.
* **Quá lớn**: Nếu kích thước khối quá lớn, dữ liệu sẽ không phù hợp với bộ nhớ đệm, dẫn đến thời gian truy xuất bộ nhớ cao hơn.

# **2. Thuật toán tìm kiếm song song (Parallel Search)**

* **Mô tả**: Tìm kiếm một phần tử trong một tập dữ liệu lớn có thể được thực hiện bằng cách chia tập dữ liệu thành các phân đoạn và sử dụng nhiều luồng để tìm kiếm đồng thời.
* **Ứng dụng**: Tìm kiếm trong các cơ sở dữ liệu, ứng dụng lập chỉ mục và hệ thống thông tin địa lý.

# **3. Tính tổng dãy số song song (Parallel Reduction)**

* **Mô tả**: Phép cộng dần (reduction) của một dãy số có thể được chia thành nhiều phần để cộng đồng thời trên các luồng khác nhau.
* **Ứng dụng**: Thống kê dữ liệu lớn, tính tổng hoặc giá trị trung bình của các dãy số lớn.

# **4. Tích phân số học song song (Parallel Numerical Integration)**

* **Mô tả**: Tính tích phân của một hàm bằng phương pháp chia nhỏ không gian tích phân và tính toán đồng thời. Các phương pháp như hình thang (Trapezoidal Rule) hoặc phương pháp Simpson thường được sử dụng song song.
* **Ứng dụng**: Tính toán khoa học, vật lý, kỹ thuật.

# **5. Sắp xếp song song (Parallel Sorting)**

* **Mô tả**: Các thuật toán sắp xếp như QuickSort, MergeSort có thể được song song hóa bằng cách chia nhỏ các đoạn mảng và sắp xếp chúng đồng thời.
* **Ứng dụng**: Sắp xếp dữ liệu lớn trong hệ thống quản lý dữ liệu, cơ sở dữ liệu và học máy.

# **6. Phép toán trên vector/mảng (Vectorized Operations)**

* **Mô tả**: Các phép tính như cộng, trừ, nhân, chia trên các vector (hoặc mảng) có thể được thực hiện song song trên nhiều luồng. Ví dụ: cộng từng phần tử của hai mảng lại với nhau.
* **Ứng dụng**: Tính toán ma trận, đồ họa máy tính, xử lý tín hiệu số.

# **7. Chia và trị song song (Parallel Divide and Conquer)**

* **Mô tả**: Thuật toán chia để trị có thể song song hóa bằng cách chia bài toán thành các bài toán con và giải quyết đồng thời. Ví dụ: thuật toán QuickSort có thể chia các mảng thành các phần nhỏ và sắp xếp chúng song song.
* **Ứng dụng**: Xử lý đồ họa, giải phương trình, nén dữ liệu.

# **8. Thuật toán luân phiên Gauss-Seidel song song (Parallel Gauss-Seidel Iteration)**

* **Mô tả**: Sử dụng OpenMP để phân chia các phép lặp Gauss-Seidel nhằm giải hệ phương trình tuyến tính trong khoa học tính toán.
* **Ứng dụng**: Giải hệ phương trình tuyến tính trong các bài toán khoa học và kỹ thuật.

# **9. Đồ thị song song (Parallel Graph Algorithms)**

* **Mô tả**: Các thuật toán đồ thị như Dijkstra, BFS (Breadth-First Search), DFS (Depth-First Search) có thể được song song hóa để xử lý các đồ thị lớn.
* **Ứng dụng**: Phân tích mạng, tìm đường đi ngắn nhất, xử lý đồ thị lớn.

# **10. Phân rã LU song song (Parallel LU Decomposition)**

* **Mô tả**: Phân rã ma trận thành hai ma trận tam giác (LU decomposition) có thể được thực hiện đồng thời bằng cách chia ma trận lớn thành các khối nhỏ và tính toán song song.
* **Ứng dụng**: Giải hệ phương trình tuyến tính trong các bài toán đại số tuyến tính và mô hình khoa học.

# **11. Tính toán Fourier nhanh song song (Parallel Fast Fourier Transform - FFT)**

* **Mô tả**: Thuật toán FFT dùng để tính toán biến đổi Fourier nhanh có thể được song song hóa bằng cách xử lý các thành phần độc lập của tín hiệu.
* **Ứng dụng**: Xử lý tín hiệu số, nén hình ảnh, âm thanh.

# **12. Phép nhân véc-tơ - ma trận song song (Parallel Matrix-Vector Multiplication)**

* **Mô tả**: Phép nhân giữa một ma trận và một véc-tơ có thể được thực hiện song song, bằng cách chia nhỏ ma trận và phân chia cho các luồng xử lý.
* **Ứng dụng**: Xử lý dữ liệu lớn, xử lý tín hiệu, học máy.

# **13. Lọc song song (Parallel Filtering)**

* **Mô tả**: Các phép lọc như lọc trung bình, lọc Gaussian trong xử lý ảnh có thể song song hóa bằng cách chia hình ảnh thành các phần và áp dụng bộ lọc đồng thời.
* **Ứng dụng**: Xử lý ảnh, video, thị giác máy tính.