# **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;  } |

chỉ thị #pragma omp parallel được sử dụng để chỉ định một khối mã sẽ được thực thi song song bởi nhiều luồng. Các **mệnh đề (clauses)** đi kèm với chỉ thị này giúp tinh chỉnh cách các biến và tài nguyên được sử dụng trong các luồng, cũng như kiểm soát số lượng luồng, các điều kiện chạy, và quá trình tổng hợp dữ liệu.

Dưới đây là chi tiết về các mệnh đề phổ biến như if, private, firstprivate, default, shared, copyin, reduction, và num\_threads:

### **1.1. if**

Mệnh đề **if** cho phép kiểm tra một điều kiện và chỉ thực hiện việc song song nếu điều kiện này đúng. Nếu điều kiện là false, khối mã sẽ được thực thi tuần tự bởi luồng chính.

|  |
| --- |
| #pragma omp parallel if(condition)  {  // Mã này chỉ chạy song song nếu "condition" là true, ngược lại chạy tuần tự  } |

### **1.2. private**

Mệnh đề **private** chỉ ra rằng mỗi luồng sẽ có bản sao cục bộ của một biến nhất định. Mỗi bản sao này sẽ độc lập giữa các luồng, và sau khi khối song song kết thúc, các biến cục bộ này sẽ không có giá trị ngoài khối.

int x = 10;

#pragma omp parallel private(x)

{

// Mỗi luồng có bản sao riêng của biến "x"

}

### **1.3. firstprivate**

Mệnh đề **firstprivate** tương tự như **private**, nhưng bản sao cục bộ của biến sẽ được khởi tạo với giá trị của biến gốc (trước khi bước vào vùng song song).

int x = 10;

#pragma omp parallel firstprivate(x)

{

// Mỗi luồng có bản sao riêng của "x", khởi tạo bằng giá trị của "x" bên ngoài là 10

}

### **1.4. default**

Mệnh đề **default** cho phép chỉ định chính sách mặc định cho các biến không được xác định rõ trong các mệnh đề khác. Các tùy chọn cho default bao gồm:

* none: buộc phải khai báo rõ ràng các biến là shared hoặc private.
* shared: tất cả các biến chưa xác định sẽ mặc định được chia sẻ giữa các luồng.
* private: tất cả các biến chưa xác định sẽ mặc định là biến cục bộ cho mỗi luồng.

#pragma omp parallel default(none)

{

// Biến nào không được khai báo rõ ràng sẽ tạo ra lỗi biên dịch

}

### **1.5. shared**

Mệnh đề **shared** chỉ ra rằng một biến cụ thể sẽ được chia sẻ giữa tất cả các luồng. Các luồng sẽ truy cập và cập nhật cùng một giá trị của biến này.

int x = 10;

#pragma omp parallel shared(x)

{

// Mọi luồng truy cập cùng một biến "x"

}

### **1.6. copyin**

Mệnh đề **copyin** cho phép khởi tạo các biến được đánh dấu với từ khóa threadprivate trong các luồng với giá trị của biến gốc từ luồng chính. Điều này hữu ích khi cần sử dụng các biến cục bộ cho các luồng.

#pragma omp threadprivate(x) // Khai báo biến "x" là threadprivate

int x = 20;

#pragma omp parallel copyin(x)

{

// Giá trị của "x" trong các luồng là giá trị của "x" từ luồng chính (20)

}

### **1.7. reduction**

Mệnh đề **reduction** cho phép thực hiện các phép toán trên các biến cục bộ của mỗi luồng và sau đó tổng hợp kết quả lại thành một biến chia sẻ. Các phép toán có thể bao gồm: cộng, trừ, nhân, min, max, v.v.

|  |
| --- |
| int sum = 0;  #pragma omp parallel for reduction(+:sum)  for (int i = 0; i < n; i++) {  sum += a[i];  }  // Sau khối song song, biến "sum" sẽ chứa tổng hợp của các giá trị được tính bởi mỗi luồng |

### **1.8. num\_threads**

Mệnh đề **num\_threads** cho phép xác định rõ số lượng luồng sẽ được sử dụng trong vùng song song. Nếu không được chỉ định, số lượng luồng mặc định sẽ phụ thuộc vào thiết lập môi trường hoặc hệ điều hành.

#pragma omp parallel num\_threads(4)

{ // Đoạn mã này sẽ được thực thi bởi 4 luồng }

## **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;  } |

### **1.1. private**

Mệnh đề **private** chỉ định rằng mỗi luồng sẽ có một bản sao cục bộ của biến nhất định trong mỗi section. Mỗi bản sao này sẽ chỉ có giá trị trong phạm vi section đó, và giá trị của biến sẽ không được giữ lại sau khi vùng song song kết thúc.

int x = 10;

#pragma omp parallel sections private(x)

{

#pragma omp section

{

x = 1;

// Mỗi luồng trong mỗi section có bản sao riêng của "x"

}

#pragma omp section

{

x = 2;

// "x" trong section này là độc lập với section khác

}

}

### **1.2. firstprivate**

Mệnh đề **firstprivate** cũng tạo ra các bản sao cục bộ của biến cho mỗi section, nhưng với một điểm khác biệt: bản sao này được khởi tạo bằng giá trị của biến từ ngoài vùng song song.

int x = 10;

#pragma omp parallel sections firstprivate(x)

{

#pragma omp section

{

// "x" khởi tạo với giá trị từ ngoài (x = 10)

std::cout << "x in section 1: " << x << std::endl;

}

#pragma omp section

{

// "x" cũng khởi tạo với giá trị từ ngoài (x = 10)

std::cout << "x in section 2: " << x << std::endl;

}

}

### **1.3. lastprivate**

Mệnh đề **lastprivate** cho phép giá trị của một biến từ section cuối cùng được sao chép ra ngoài vùng song song. Điều này có nghĩa là giá trị của biến từ section cuối cùng (theo thứ tự hoàn thành) sẽ được bảo toàn sau khi khối song song kết thúc.

int x = 0;

#pragma omp parallel sections lastprivate(x)

{

#pragma omp section

{

x = 1;

}

#pragma omp section

{

x = 2;

}

}

// Sau khi khối song song kết thúc, "x" sẽ có giá trị từ section cuối cùng (có thể là 1 hoặc 2 tùy vào thứ tự)

### **1.4. reduction**

Mệnh đề **reduction** được sử dụng để thực hiện các phép toán tổng hợp trên các biến cục bộ trong mỗi section và sau đó tổng hợp lại giá trị cuối cùng sau khi tất cả các section đã hoàn thành. Điều này thường áp dụng cho các phép toán như cộng, trừ, nhân, min, max, v.v.

int sum = 0;

#pragma omp parallel sections reduction(+:sum)

{

#pragma omp section

{

sum += 10;

}

#pragma omp section

{

sum += 20;

}

}

// Sau khi khối song song kết thúc, "sum" sẽ là 30 (10 + 20)

### **1.5. nowait**

Mệnh đề **nowait** cho phép bỏ qua việc đồng bộ hóa sau khi tất cả các section hoàn thành. Theo mặc định, OpenMP sẽ chờ cho đến khi tất cả các section hoàn thành trước khi tiếp tục thực thi đoạn mã tiếp theo. Với nowait, điều này có thể bị bỏ qua, và chương trình sẽ tiếp tục mà không cần đợi.

#pragma omp parallel sections nowait

{

#pragma omp section

{

// Section 1

}

#pragma omp section

{

// Section 2

}

}

// Không cần đợi tất cả các section hoàn thành trước khi tiếp tục

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

Chỉ dẫn này đảm bảo rằng cho phép một luồng duy nhất thực thi một đoạn mã nhất định, trong khi các luồng khác sẽ chờ tại vị trí này hoặc tiếp tục thực hiện các công việc khác. Điều này hữu ích trong các tình huống khi bạn muốn chỉ một luồng thực thi một đoạn mã cụ thể (ví dụ: khởi tạo dữ liệu chung hoặc ghi log), mà không cần phải đồng bộ hóa toàn bộ nhóm luồng.

|  |
| --- |
| #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;  } |

### **5.1. private(list)**:

**private(list)**: Chỉ định rằng các biến trong list sẽ được tạo riêng cho luồng thực hiện đoạn mã trong chỉ thị single. Mỗi luồng khác sẽ không nhìn thấy hoặc truy cập vào các biến này.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  int x = 20; // Biến x được khởi tạo giá trị ban đầu là 20  #pragma omp parallel  {  #pragma omp single private(x)// Tùy chọn private đảm bảo rằng mỗi luồng sẽ có bản sao riêng biệt của các biến, nhưng không được khởi tạo với giá trị từ luồng cha (giá trị của biến ban đầu không được sao chép).  {  std::cout << "Giá trị của x trong `single` trước khi khởi tạo: " << x << " (luồng " << omp\_get\_thread\_num() << ")" << std::endl;  x = 50; // Khởi tạo giá trị mới cho x trong luồng `single`  std::cout << "Giá trị của x sau khi thay đổi trong `single`: " << x << " (luồng " << omp\_get\_thread\_num() << ")" << std::endl;  }  // Giá trị của x ở các luồng khác vẫn không bị ảnh hưởng bởi thay đổi trong `single`  std::cout << "Giá trị của x ngoài `single`: " << x << " (luồng " << omp\_get\_thread\_num() << ")" << std::endl;  }  return 0;  } |

### **5.2. firstprivate (list)**:

**firstprivate(list)**: Tương tự như private, nhưng giá trị ban đầu của các biến trong list được sao chép từ luồng cha (luồng gọi).

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  int x = 10; // Biến x được khởi tạo giá trị ban đầu là 10  #pragma omp parallel  {  #pragma omp single firstprivate(x)// firstprivate cho phép các biến được khởi tạo bằng giá trị từ luồng cha (luồng ban đầu) và sử dụng biến đó một cách độc lập trong các luồng  {  std::cout << "Giá trị ban đầu của x trong `single`: " << x << " (luồng " << omp\_get\_thread\_num() << ")" << std::endl;  x += 5; // Thay đổi giá trị của x trong luồng này  std::cout << "Giá trị của x sau khi thay đổi trong `single`: " << x << " (luồng " << omp\_get\_thread\_num() << ")" << std::endl;  }  // Giá trị của x ở các luồng khác không bị thay đổi  std::cout << "Giá trị của x ngoài `single`: " << x << " (luồng " << omp\_get\_thread\_num() << ")" << std::endl;  }  return 0;  } |

### **5.3. nowait:**

**nowait**: Loại bỏ rào cản mặc định (barrier) ở cuối chỉ thị single. Khi không có nowait, tất cả các luồng sẽ chờ đến khi luồng thực hiện xong đoạn mã single. Nếu sử dụng nowait, các luồng khác có thể tiếp tục thực hiện ngay lập tức mà không cần đợi luồng đang thực thi single hoàn thành.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  #pragma omp parallel  {  #pragma omp single nowait // sử dụng nowait để các luồng không phải chờ luồng thực thi single hoàn thành.  {  std::cout << "Chỉ một luồng thực hiện khởi tạo, luồng: " << omp\_get\_thread\_num() << std::endl;  }  // Các luồng khác không cần phải đợi sau khi luồng thực hiện `single` kết thúc  std::cout << "Các luồng tiếp tục thực hiện, luồng: " << omp\_get\_thread\_num() << std::endl;  }  return 0;  } |

### **5.4. copyprivate(list):**

**copyprivate(list)**: Biến trong list sẽ được chia sẻ với tất cả các luồng khác sau khi đoạn mã single hoàn thành. Điều này thường được sử dụng để truyền giá trị tính toán bởi một luồng duy nhất cho tất cả các luồng còn lại.

|  |
| --- |
| #include <iostream>  #include <omp.h>  int main() {  int x = 0;  #pragma omp parallel  {  #pragma omp single copyprivate(x)// Dùng firstprivate để chia sẻ giá trị ban đầu của biến cho luồng thực hiện single.  {  x = 200; // Chỉ một luồng khởi tạo giá trị x  std::cout << "Giá trị của x được khởi tạo bởi luồng: " << omp\_get\_thread\_num() << std::endl;  }  // Các luồng khác nhận giá trị của x  std::cout << "Giá trị của x từ luồng: " << omp\_get\_thread\_num() << " là: " << x << 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;  } |

### 2.4. flush

**flush** là một chỉ thị quan trọng giúp đồng bộ hóa bộ nhớ giữa các luồng (threads) trong các ứng dụng song song. Nó đảm bảo rằng tất cả các thay đổi được thực hiện trên các biến chia sẻ giữa các luồng sẽ được ghi lại vào bộ nhớ chính và có thể truy cập được bởi các luồng khác. Điều này đặc biệt hữu ích khi làm việc với bộ nhớ đệm cục bộ của từng luồng.

 **#pragma omp flush** là một chỉ thị được sử dụng để đảm bảo tính nhất quán của dữ liệu giữa các luồng. Khi một luồng thực hiện một chỉ thị **flush**, tất cả các giá trị của các biến chia sẻ (shared variables) mà nó đã sửa đổi sẽ được "đẩy" từ bộ nhớ cục bộ (cache) về bộ nhớ chính, đảm bảo rằng các luồng khác có thể nhìn thấy các thay đổi này ngay lập tức.

 Tương tự, khi một luồng thực hiện **flush**, nó cũng tải lại giá trị của các biến chia sẻ từ bộ nhớ chính vào bộ nhớ cục bộ, đảm bảo rằng luồng đó có được phiên bản mới nhất của các biến mà các luồng khác có thể đã thay đổi.

### 2.5. ordered

chỉ thị **ordered** được sử dụng để đảm bảo rằng một số đoạn mã cụ thể sẽ được thực thi theo thứ tự tuần tự, ngay cả khi các luồng khác đang thực hiện song song. Điều này rất hữu ích khi bạn muốn duy trì một trình tự nhất định cho một số phép toán, chẳng hạn như ghi kết quả ra màn hình, hoặc xử lý dữ liệu mà yêu cầu kết quả phải xuất hiện theo một thứ tự cố định.

**ordered** là một chỉ thị nhằm giới hạn việc thực hiện một đoạn mã nhất định trong một vùng song song, đảm bảo đoạn mã đó được thực thi tuần tự theo thứ tự các lần lặp. Nó chỉ có thể được sử dụng bên trong các vòng lặp **for** hoặc **parallel for** trong OpenMP khi có chỉ định thêm **ordered**.

|  |
| --- |
| #include <omp.h>  #include <iostream>  int main() {  int n = 10;  #pragma omp parallel for ordered  for (int i = 0; i < n; i++) {  // Một đoạn mã song song hóa  std::cout << "Thread " << omp\_get\_thread\_num() << " is working on iteration " << i << std::endl;  // Đảm bảo đoạn mã này được thực thi theo thứ tự  #pragma omp ordered  {  std::cout << "Ordered output for iteration " << i << std::endl;  }  }  return 0;  } |

# **3. *Chỉ dẫn đối với môi trường dữ liệu***

3.1. **threadprivate**

chỉ thị **threadprivate** được sử dụng để khai báo các biến toàn cục hoặc biến tĩnh sao cho mỗi luồng có phiên bản riêng biệt của biến đó. Điều này có nghĩa là mỗi luồng sẽ có bản sao độc lập của biến và chúng sẽ không bị chia sẻ giữa các luồng, giúp tránh được các tình huống xung đột dữ liệu.

 **threadprivate** cho phép biến toàn cục, biến tĩnh hoặc biến tĩnh cục bộ trong một hàm được khai báo là biến riêng biệt cho từng luồng. Mỗi luồng sẽ có một bản sao riêng của biến này, do đó không có sự chia sẻ biến giữa các luồng.

 Thay vì tạo ra các biến cục bộ trong mỗi hàm hoặc vòng lặp, **threadprivate** cho phép các biến toàn cục vẫn có tính chất cục bộ theo từng luồng.

|  |
| --- |
| #include <omp.h>  #include <iostream>  int my\_global\_var; // Biến toàn cục  #pragma omp threadprivate(my\_global\_var) // Đảm bảo mỗi luồng có bản sao riêng của my\_global\_var  int main() {  // Thiết lập môi trường song song  #pragma omp parallel  {  my\_global\_var = omp\_get\_thread\_num(); // Gán giá trị riêng cho biến của mỗi luồng  std::cout << "Thread " << omp\_get\_thread\_num()  << " has my\_global\_var = " << my\_global\_var << std::endl;  }  // Sau khi kết thúc vùng song song, biến vẫn tồn tại và có thể được sử dụng lại  #pragma omp parallel  {  std::cout << "Thread " << omp\_get\_thread\_num()  << " retains my\_global\_var = " << my\_global\_var << 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.

|  |
| --- |
| #include <omp.h>  #include <iostream>  int main() {  int flag = 0;  #pragma omp parallel num\_threads(2)  {  int thread\_id = omp\_get\_thread\_num();  if (thread\_id == 0) {  // Thread 0 cập nhật biến flag  flag = 1;    // Thực hiện flush để đồng bộ biến flag với bộ nhớ chính  #pragma omp flush(flag)    std::cout << "Thread 0 set flag to 1" << std::endl;  } else if (thread\_id == 1) {  int local\_flag = 0;    // Chờ đến khi flag được cập nhật  while (local\_flag == 0) {  #pragma omp flush(flag)  local\_flag = flag;  }    std::cout << "Thread 1 detected flag change" << std::endl;  }  }  return 0;  } |
|  |