**LỜI MỞ ĐẦU**

Ngày nay với sự phát triển của công nghệ thông tin, các máy tính đa nhân, đa lõi (multiple processor) đang dần dần thay thế cho bộ xử lý đơn lõi để xử lý các bài toán có độ phức tạp cao.

Tuy nhiên với lối lập trình truyền thống là lập trình tuần tự thì hầu hết các chương trình ứng dụng đều được lập trình, thiết kế trên bộ xử lý đơn lõi

(single processor). Như vậy sẽ không khai thác hết hiệu năng tính toán mà bộ xử lý đa nhân, đa lõi mang lại đồng thời khó đáp ứng được yêu cầu tính toán của người dùng.

Một thách thức, một yêu cầu đặt ra là làm thế nào để khai thác được hiệu năng tính toán mà bộ xử lý đa nhân, đa lõi mang lại. Không còn cách nào khác là thay vì lập trình, tính toán tuần tự chuyển sang lập trình, tính toán song song.

Lập trình, tính toán song song ra đời nhằm khai thác, phát huy hiệu năng tính toán của bộ xử lý đa lõi, đồng thời giảm thời gian tính toán của các bài toán có khối lượng dữ liệu lớn.

Các công cụ hỗ trợ lập trình, tính toán song song có thể kể đến như: Thư viện MPI (Message Passing Interface), PMV (Parallel Virtual Machine), một số được tích hợp sẵn thành chuẩn trong các ngôn ngữ lập trình như thư viện OpenMP (Open Multiple Processing) trong C/C++, FOTRAN.

Trong khuôn khổ thời gian thực tập kỹ thuật, em đã tìm hiểu và áp dụng lập trình bài toán tính toán song song trên cơ sở sử dụng thư viện mã nguồn mở OpenMP để thực hiện một bài toán đơn giản. Do kiến thức cũng như kỹ năng còn có hạn nên bài toán của em đưa ra chỉ ứng dụng được một phần nhỏ nào của một thư viện lớn như OpenMp.

Em xin chân thành cảm ơn thầy Nguyễn Tiến Thành đã tận tình giúp đỡ và định hướng cho em để em tìm hiều và thực hiện báo cáo thực tập này. Bài làm của em còn nhiều thiếu sót do kiến thức cũng như thời gian có hạn. Em mong nhận được sự đóng góp từ các thầy/cô để em hoàn thiện và tìm hiểu việc tính toán song song trong máy tính được kỹ càng hơn.

Hà nội, ngày 30 tháng 10 năm 2019.

SINH VIÊN

Vũ Văn Cảnh

[**Lời mở đầu**](#_TOC_250076)

[**Mục lục**](#_TOC_250075)

[**CHƯƠNG 1**: Tổng quan về lập trình song song, tính toán song song](#_TOC_250073)

[Định nghĩa](#_TOC_250072)

[Thế nào là lập trình, tính toán song song](#_TOC_250071)

[Tại sao phải lập trình, tính toán song song](#_TOC_250070)

[Sử dụng lập trình, tính toán song song để làm gì](#_TOC_250069)

[So sánh lập trình tính toán tuần tự và lập trình tính toán song song](#_TOC_250068)

[Sự phân chia cấu trúc tính toán song song](#_TOC_250067)

[Phân chia dựa trên quan hệ giữa chỉ thị lệnh và dữ liệu](#_TOC_250066)

[Sự phân chia dựa trên mối quan hệ giữa bộ xử lý và bộ nhớ](#_TOC_250065)

[Các mô hình lập trình song song](#_TOC_250064)

[Mô hình dùng chung bộ nhớ (Shared Memory)](#_TOC_250063)

[Mô hình luồng (Thread)](#_TOC_250062)

[Mô hình truyền thông điệp (Message Passing)](#_TOC_250061)

[Mô hình song song dữ liệu (Data Parallel)](#_TOC_250060)

[Một số vấn đề liên quan đến lập trình và tính toán song song](#_TOC_250059)

[Định luật Amdahl‟s](#_TOC_250058)

[Cân bằng tải](#_TOC_250057)

[Sự bế tắc](#_TOC_250056)

[**CHƯƠNG 2**: Thư viện Mã nguồn mở OpenMP và ứng dụng](#_TOC_250055)

[Tổng quan về OpenMP](#_TOC_250054)

[Giới thiệu](#_TOC_250053)

[Định nghĩa](#_TOC_250052)

[Lịch sử phát triển](#_TOC_250051)

[Mục đích của OpenMP](#_TOC_250050)

[Mô hình lập trình song song trong OpenMP](#_TOC_250049)

[Các chỉ thị biên dịch (Compiler Directive)](#_TOC_250048)

[Khuôn dạng của chỉ thị](#_TOC_250047)

[Phạm vi của chỉ thị](#_TOC_250046)

[Cấu trúc vùng song song](#_TOC_250045)

[Cấu trúc chia sẻ công việc (Work Sharing Construct)](#_TOC_250044)

[Cấu trúc đồng bộ](#_TOC_250043)

[Chỉ thị THREADPRIVATE](#_TOC_250042)

[Các mệnh đề trong OpenMP](#_TOC_250041)

[Mệnh đề PRIVATE](#_TOC_250040)

[Mệnh đề FIRSTPRIVATE](#_TOC_250039)

[Mệnh đề LASTPRIVATE](#_TOC_250038)

[Mệnh đề SHARED](#_TOC_250037)

[Mệnh đề DEFAULT](#_TOC_250036)

[Mệnh đề REDUCTION](#_TOC_250035)

[Mệnh đề COPYIN](#_TOC_250034)

Thư viện Runtime (Runtime Library Routine)

[OMP\_SET\_NUM\_THREADS](#_TOC_250033)

[OMP\_GET\_NUM\_THREADS](#_TOC_250032)

[OMP\_GET\_THREAD\_NUM](#_TOC_250031)

[OMP\_GET\_MAX\_THREADS](#_TOC_250030)

[OMP\_GET\_NUM\_PROCS](#_TOC_250029)

[OMP\_IN\_PARALLEL](#_TOC_250028)

[OMP\_SET\_DYNAMIC](#_TOC_250027)

[OMP\_GET\_DYNAMIC](#_TOC_250026)

[OMP\_SET\_NESTED](#_TOC_250025)

[OMP\_GET\_NESTED](#_TOC_250024)

[OMP\_INIT\_LOCK](#_TOC_250023)

[OMP\_DESTROY\_LOCK](#_TOC_250022)

[OMP\_SET\_LOCK](#_TOC_250021)

[OMP\_UNSET\_LOCK](#_TOC_250020)

[OMP\_TEST\_LOCK](#_TOC_250019)

Các biến môi trường (Enviroment Variables)

[OMP\_SCHEDULE](#_TOC_250018)

[OMP\_NUM\_THREADS](#_TOC_250017)

[OMP\_DYNAMIC](#_TOC_250016)

[OMP\_NESTED](#_TOC_250015)

[**CHƯƠNG 3**: Ứng dụng OpenMP](#_TOC_250014)

[Ứng dụng openMP trong xử lý ảnh](#_TOC_250013)

[Kết luận](#_TOC_250001)

[Tài liệu tham khảo](#_TOC_250000)

**CHƯƠNG 1: Tổng quan về lập trình song song, tính toán song song**

1. **Định nghĩa:**
2. Thế nào là lập trình, tính toán song song

* Tính toán song song là sự thực hiện một cách đồng thời hai hoặc nhiều phép toán, công việc vào một thời điểm, được thực hiện bởi các bộ xử lý khác nhau.

1. Tại sao phải lập trình, tính toán song song

* Theo xu hướng phát triển của công nghệ thông tin, các bộ xử lý đa nhân, đa lõi (multiple processor) đang dần dần thay thế các bộ xử lý đơn lõi (single processor) tuy nhiên với lối lập trình truyền thống (lập trình tuần tự), các câu lệnh, các quá trình xử lý được thực hịên một cách lần lượt, tuần tự như vậy sẽ không phát huy hết công năng, hiệu năng của bộ vi xử lý đa nhân, đa lõi (multiple processor). Lập trình, tính toán song song ra đời như một lời giải cho yêu cầu, thách thức đặt ra là làm thế nào để phát huy công năng, hiệu năng của bộ đa xử lý (multiple processor).
* Trên thực tế, có rất nhiều bài toán với dữ liệu lớn, độ phức tạp tính toán cao mà đòi hỏi thời gian xử lý ngắn, độ chính xác cao. Vd: các bài toán liên quan tới xử lý ảnh, xử lý tín hiệu, dự báo thời tiết, mô phỏng giao thông, mô phỏng sự chuyển động của các phân tử, nguyên tử, mô phỏng bản đồ gen, các bài toán liên quan đến cơ sở dữ liệu và khai thác cơ sở dữ liệu. . . với bộ xử lý đơn lõi thì khó có thể thực hiện và cho kết quả như mong muốn được.
* Lập trình, tính toán song song là lời giải đáp cho bài toán tăng hiệu năng xử lý đồng thời rút ngắn thời gian xử lý tính toán của người dùng.

1. Sử dụng lập trình, tính toán song song để làm gì

* Phát huy công năng, hiệu năng của bộ xử lý đa nhân, đa lõi.
* Giải quyết một số bài toán lớn mà bộ xử lý đơn lõi (single processor) không thực hiện được.
* Tăng hiệu quả tính toán đồng thời giảm thời gian tính toán.
* So sánh lập trình tính toán tuần tự và lập trình tính toán song song.

|  |  |
| --- | --- |
| **Lập trình tính toán tuần tự** | **Lập trình tính toán song song** |
| * Chương trình ứng dụng chạy trên bộ xử lý đơn (single processor). * Các chỉ thị lệnh được bộ xử lý (CPU) thực hiện một cách lần lượt, tuần tự. * Mỗi chỉ thị lệnh chỉ thực thiện trên duy nhất một thành phần dữ liệu. * Lập trình viên chỉ cần đảm bảo viết đúng mã lệnh theo giải thuật chương trình là chương trình có thể dịch, chạy và cho ra kết quả. * Thường được áp dụng đối với các bài toán có dữ liệu nhỏ, độ phức tạp bình thường và thời   gian cho phép. | * Chương trình ứng dụng chạy trên hai hoặc nhiều bộ xử lý. * Các chỉ thị lệnh được các bộ vi xử lý thực hiện một cách song song, đồng thời. * Mỗi chỉ thị lệnh có thể thao tác trên hai hoặc nhiều thành phần dữ liệu khác nhau. * Ngoài việc đảm bảo viết đúng mã lệnh theo giải thuật, lập trình viên còn phải chỉ ra trong chương trình đoạn mã nào được thực hiện song song, đồng thời. * Thường được áp dụng đối với các bài toán có dữ liệu lớn, độ phức tạp cao và thời gian ngắn. |

1. **Sự phân chia cấu trúc tính toán song song**
2. Phân chia dựa trên quan hệ giữa chỉ thị lệnh và dữ liệu

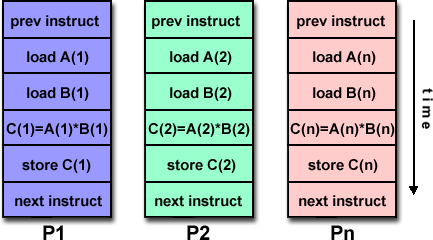
* Dựa vào mối quan hệ giữa chỉ thị lệnh và dữ liệu chia làm các loại :

+ Đơn chỉ thị lệnh, đa dữ liệu SIMD (Single Instruction, Multiple Data).

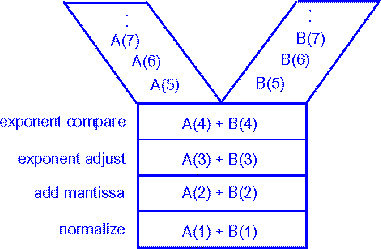
+ Đa chỉ thị lệnh, đơn dữ liệu MISD (Multiple Instruction, Single Data).

+ Đa chỉ thị lệnh, đa dữ liệu MIMD (Multiple Instruction, Multiple Data).

+ Đơn chỉ thị lệnh, đa dữ liệu SIMD (Single Instruction, Multiple Data).



Hình 1.1 Cấu trúc đơn chỉ thị lệnh, đa dữ liệu

* Là một loại của bộ xử lý song song.
* Khi một chỉ thị phát ra, tất cả các quá trình xử lý được thực hiện.
* Mỗi quá trình xử lý sẽ thực hiện trên một thành phần dữ liệu khác nhau của cùng một cấu trúc dữ liệu.
* Được chia làm hai loại:
*  Vector SIMD

Hình 1.2 Mô hình vector SIMD

**+** Một chỉ thị lệnh phát ra, nhiều thao tác bắt đầu cập nhật.

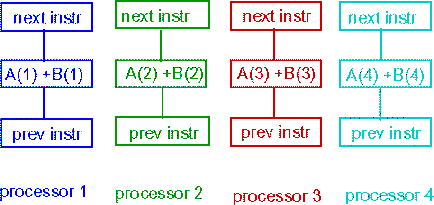
+ Chuẩn tuần tự thao tác, xử lý trên một thành phần dữ liệu, vector SIMD thao tác, xử lý trên vector, nhóm dữ liệu.

* Parallel SIMD

+ Khi một chỉ thị lệnh phát ra, tất cả các bộ vi xử lý thực hiện thao tác trên các dữ liệu khác nhau.

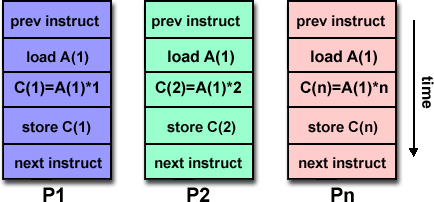
+ Các bộ xử lý chạy đồng bộ trên một nhịp của đồng hồ hệ thống.

+ Người sử dụng không phải chịu trách nhiệm về vấn đề đồng bộ.



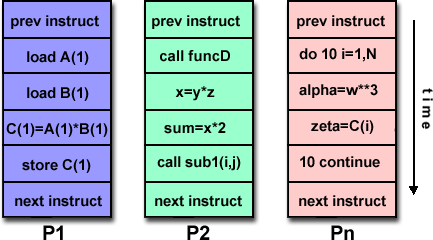
Hình 1.3 Mô hình parallel SIMD

* Đa chỉ thị lệnh, đơn dữ liệu MISD (Multiple Instruction, Single Data).



Hình 1.4 Mô hình đa chỉ thị đơn dữ liệu

* Một dòng dữ liệu được cung cấp cho nhiều quá trình xử lý.
* Mỗi quá trình xử lý sẽ thao tác trên dữ liệu một cách độc lập bằng chỉ thị lệnh khác nhau.
* Chuẩn này thường ít được sử dụng.
* Đa chỉ thị lệnh, đa dữ liệu MIMD (Multiple Instruction, Multiple Data).

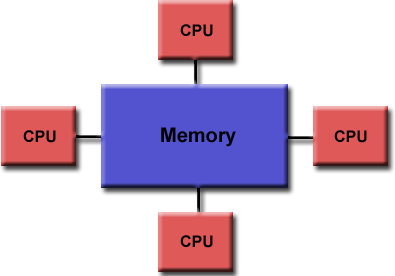


* Đây là cấu trúc phổ biến nhất của máy tính song song
* Cấu trúc này thực hiện dựa trên sự kết nối của nhiều bộ vi xử lý khác nhau.
* Mỗi bộ xử lý sẽ thực thi trên các chỉ thị lệnh khác nhau.
* Mỗi bộ xử lý sẽ thực hiện trên các dòng dữ liệu khác nhau.
* Quá trình thực hiện có thể là đồng bộ hoặc không đồng bộ.
* Thuận lợi:
* Các bộ xử lý có thể thực hiện xử lý một cách đồng thời.
* Mỗi bộ xử lý thực hiện một cách độc lập mà không quan tâm tới bộ xử lý khác đang làm gì.
* Khó khăn:
* Khó khăn trong quá trình đồng bộ và cân bằng tải (Load banacing)
* Khó khăn cho thiết kế chương trình.
* Sự phân chia dựa trên mối quan hệ giữa bộ xử lý và bộ nhớ.
* Dựa trên mối quan hệ giữa bộ xử lý và bộ nhớ được chia làm các loại :

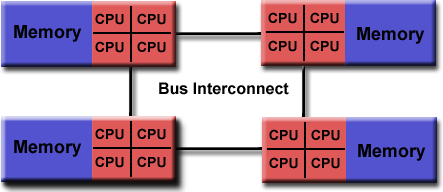
+ Dùng chung bộ nhớ (Shared Memory).

+ Phân bổ bộ nhớ (Distributed Memory).

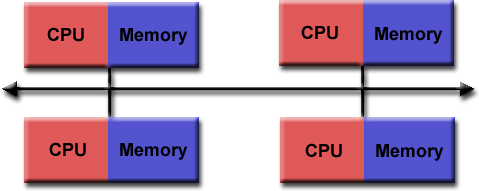
+ Kết hợp cả hai (Hybrid Distributed – Shared Memory).



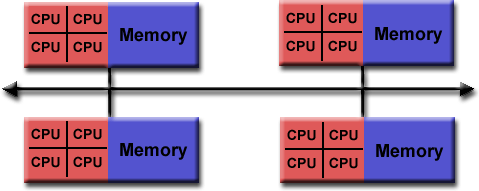
Hình 1.6 Mô hình truy cập bộ nhớ đồng bộ.



Hình 1.7 Mô hình truy cập bộ nhớ không đồng bộ.



Hình 1.8 Mô hình phân bổ bộ nhớ.



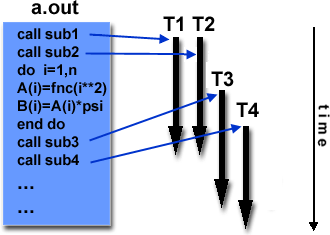
Hình 1.9 Mô hình kết hợp.

* Một số mô hình lập trình song song thường sử dụng là:

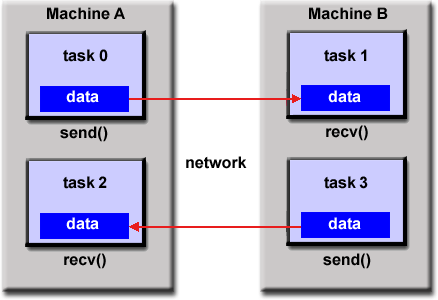
+ Mô hình dùng chung bộ nhớ (Shared Memory)

+ Mô hình luồng (Thread).

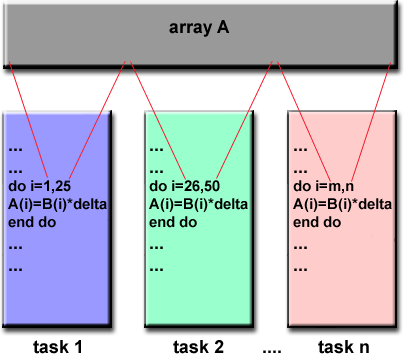
+ Mô hình truyền thông điệp (Message passing).

 + Mô hình song song dữ liệu (Data Parallel).

Hình 1.10 Mô hình luồng.



Hình 1.11 Mô hình truyền thông điệp.

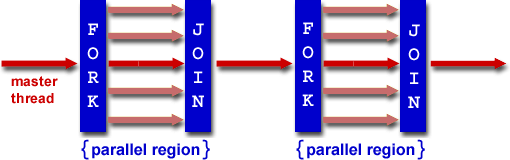


Hình 1. 12 Mô hình song song dữ liệu.

**CHƯƠNG 2: Thư viện Mã nguồn mở OpenMP**

1. **Tổng quan về OpenMP**

* Có rất nhiều công cụ hỗ trợ chúng ta trong lập trình và tính toán song song, một trong những công cụ hỗ trợ hữu hiệu là thư viện mã nguồn mở OpenMP. OpenMP được các nhà phát triển tích hợp thành chuẩn trong các ngôn ngữ lập trình phổ biến như Fortran, C/C++. . . Và hỗ trợ hầu hết các hệ điều hành. Trong khuôn khổ chương trình em xin trình bày các cấu trúc, chỉ thị của OpenMP trong C++.
* OpenMP (Open Multi – Processing) là một giao diện lập trình ứng dụng Application Program Interface (API) được sử dụng để điều khiển các luồng (Thread) dựa trên cấu trúc chia sẻ bộ nhớ chung. Các thành phần của OpenMP gồm:
* Các chỉ thị biên dịch (Compiler Directive).
* Thư viện runtime (Runtime Library Rountines).
* Các biến môi trường (Enviroment Variables).
* Được định nghĩa dựa trên một nhóm phạm trù phần cứng và phần mềm, OpenMP là một thư viện, giúp cho người lập trình đơn giản và mềm dẻo để phát triển chương trình song song chạy trên máy PC hỗ trợ nhiều bộ xử lý.
* Lịch sử phát triển: OpenMP do ARB (Architecture Review Board) một nhóm các nhà phát triển máy tính phát hành với tên API. Phiên bản đầu tiên 1.0 dành cho Fortran được công bố vào tháng 10 năm 1997. Vào tháng 10 năm 1998 C/C ++ tích hợp thành chuẩn của mình. Phiên bản 2.0 được Fortran công bố vào năm 2000 và đến năm 2002 C/C++ cũng tung ra phiên bản 2.0 của mình. Phiên bản 2.5 được cả Fortran và C/C ++ công bố vào năm 2005. Phiên bản 3.0 được công bố vào năm 2008 và nó là phiên bản hiện tại được tích hợp thêm nhiều tính năng mới.
* Mục đích của OpenMP: OpenMP ra đời với mục tiêu cung cấp một chuẩn chung cho rất nhiều kiến trúc và nền tảng phần cứng. Nó là thư viện mã nguồn mở cung cấp rất nhiều các hàm, các chỉ thị giúp cho người lập trình linh động và dễ dàng phát triển ứng dụng song song của mình.
* Mô hình lập trình song song trong OpenMP.
* Mô hình sử dụng để lập trình trong OpenMP là mô hình FORK – JOIN: Trong mô hình này, tất cả các chương trình khi bắt đầu chạy sẽ được xử lý tuần tự bởi luồng chủ (Master Thread) cho đến khi bắt gặp vùng song song.
* Fork: luồng chủ sẽ tạo ra các luồng thực hiện song song. Các đoạn mã song song trong chương trình sẽ được các luồng này thực thi một cách đồng thời.
* Join: Khi các luồng thực thi các đoạn mã trong vùng song song kết thúc chúng sẽ được đồng bộ sau đó công việc lại được thực thi bởi luồng chủ.



Hình 2.1 Mô hình fork - join

1. **Các chỉ thị biên dịch (Compiler Directive)**

* Chỉ thị biên dịch là bắt buộc có đối với mỗi chương trình ứng dụng song song. Chỉ thị biên dịch sẽ báo cho trình biên dịch biết sự bắt đầu của khối mã thực hiện song song.
* Khuôn dạng của chỉ thị.
* Chỉ thị trong OpenMP được cho dưới dạng sau:

*#pragma omp directive- name [clause…] newline*

*#pragma omp*: Đây là yêu cầu bắt buộc đối với mọi chỉ thị trong OpenMP. Chỉ thị này sẽ báo cho chương trình biết bắt đầu của khối mã song song.

*Directive-name*: Tên của chỉ thị, tên của chỉ thị pahỉ xuất hiện sau #pragma và đứng trước bất kỳ mệnh đề nào.

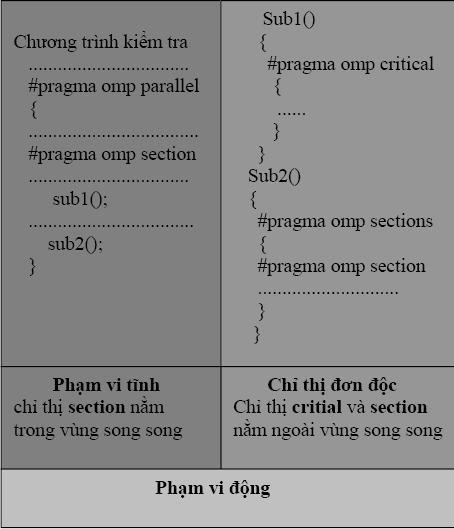
*Clause*: Các chỉ thị này không bắt buộc trong chỉ thị, các chỉ thị này sẽ đưa ra phạm vi hoạt động của các biến đối với các thread.

*newline*: Yêu cầu bắt bộc đối với mỗi cấu trúc chỉ thị. Nó là tập mã lệnh nằm trong khối cấu trúc được bao bọc bởi chỉ thị.

* Ví dụ:
* *#pragma omp parallel shared (a, b) private(i)*
* {
* . . . . . .
* // các khối mã được thực hiện song song.
* . . . .
* }

1. Phạm vi của chỉ thị.

* Phạm vi tĩnh (static extent).
* Chỉ thị đơn độc (ophaned directive).
* Phạm vi động (dymanic extent).



Hình 2.2 Phạm vi của chỉ thị.

1. Cấu trúc vùng song song

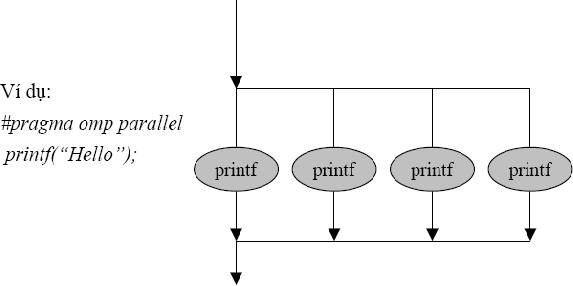
* Một vùng song song là một khối mã mà được thực thi bởi nhiều threads.
* Chúng có khuôn dạng như sau:

*#pragma omp parallel [clause. . . ] newline if (scalar\_expression)*

*private (list) shared (list)*

*default (shared | none) firstprivate (list) reduction (operator: list) copyin (list)*

*num\_threads (integer-expression) structured\_block*



Hình 2.3 Cấu trúc vùng song song.

* Khi một luồng gặp chỉ thị PARALLEL nó sẽ tạo ra một tập các luồng trong đó luồng đầu tiên là luồng chủ của tập các luồng. Luồng chủ cũng là một thành phần của tập các luồng nó có chỉ số là 0, các luồng thứ i sẽ có chỉ số là i-1.
* Khi bắt đầu một vùng song song, đoạn mã nguồn của vùng song song sẽ được sao ra làm nhiều bản để đưa cho các luồng thực hiện một cách song song. Tại vị trí cuối của đoạn mã song song, mặc định sẽ có một điểm đồng bộ để đồng bộ tất cả các luồng, sau điểm đồng bộ này, đoạn mã của chương trình sẽ được thực hiện tuần tự bởi luồng chủ. Vậy một vấn đề đặt ra là có bao nhiêu luồng được thực thi đoạn mã trong vùng song song. Để biết được điều này, OpenMP cung cấp hàm thư viện *omp\_get\_num\_threads()* trả về giá trị là tổng số luồng được thực thi trong vùng song song và *omp\_get\_thread\_num()* trả về chỉ số của luồng hiện tại đang thực thi đoạn mã trong vùng song song.
* Vùng song song lồng (Nested Parallel Region).
* Vùng song song song lồng là vùng song song xuất hiện trong một vùng song song khác. OpenMP cung cấp các hàm thư viện cho phép thực hiện vùng song song lồng *omp\_set\_nested()* và *omp\_get\_nested()* để kiểm tra xem trong đoạn mã thực thi có xuất hiện vùng song song hay không.
* Vùng song song động (Dynamic Parallel Region).
* Bình thường khi một chương trình được chia ra thành các vùng song song thì mặc định các vùng song song đó sẽ được thực hiện bởi các luồng với số lượng bằng nhau. Tuy nhiên OpenMP cho phép chúng ta gán động các luồng thực hiện cho mỗi vùng song song. Để thự hiện được điều này, chúng ta sử dụng hàm thư viện *omp\_set\_dynamic()* hoặc đặt giá trị của biến môi trường OMP\_DYNAMIC là TRUE.
* Cấu trúc chia sẻ công việc (Work Sharing Construct).
* Cấu trúc chia sẻ công việc cho phép người lập trình chia công việc trong vùng song song cho các luồng thực hiện như thế nào. Cấu trúc chia sẻ công việc được thực hiện trong vùng song song. Có ba cấu trúc chia sẻ công việc đó là cấu trúc DO/FOR, cấu trúc SECTIONS và cấu trúc SINGLE.

1. Các loại chỉ thị

* Chỉ thị Do/for:

Chỉ thị DO/FOR chỉ ra rằng các công việc lặp đi lặp lại được cho bởi vòng lặp phải được thực hiện một cách song song. Cấu trúc của chỉ thị này có dạng như sau:

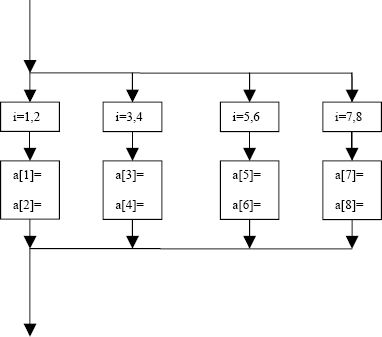
*#pragma omp for [clause. . . ] newline schedule (type [, chunk\_size]) ordered*

*private (list) firstprivate (list) lastprivate (list) shared (list)*

*reduction (operator: list) nowait*

*for( ;. . . . . ; )*

* Ví dụ:
* *#pragma omp parallel*
* *. . . .*
* *#pragma omp for schedule (static, 2) for (int i=1; i<8 ; i++)*
* *a[i]=xxx;*



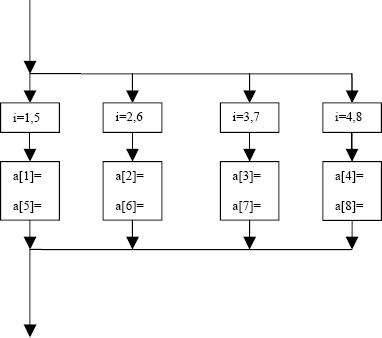
Hình 2.4 Mô tả hoạt động của các luồng thực thi với schedule là static

* DYNAMIC:

Cũng tương tự như STATIC, các công việc lặp đi lặp lại của vòng lặp được chia làm các *chunk\_size* công việc, nhưng khác với STATIC các công việc ở đây được gán động cho các luồng thực hiện.

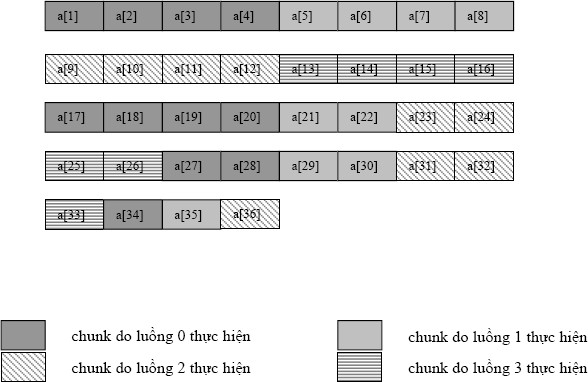
Ví dụ:

* *. . . .*
* *#pragma omp parallel*
* *. . . .*
* *#pragma omp for schedule (dynamic, 1) for (int i=1;i<8 ; i++)*
* *a[i]=xxx;*



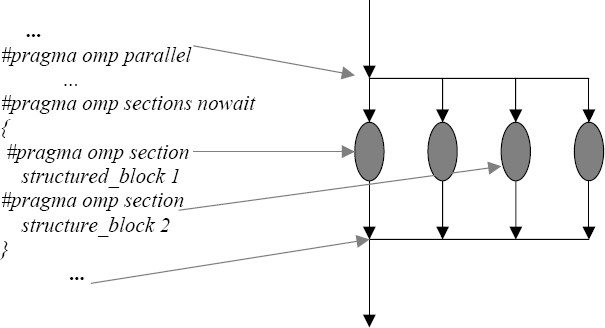
Hình 2.5 Mô tả hoạt động của các luồng thực thi với schedule là dynamic.

* GUIDED:
* Kiểu phân chia này tương tự như kiểu phân chia động, chỉ khác ở chỗ cỡ của mỗi chunk công việc không phải là hằng số mà nó giảm theo hàm mũ qua mỗi lần một luồng thực hiện xong một chunk công việc và chuyển sang thực hiện một chunk công việc mới. khi mà một luồng kết thúc một chunk công việc, nó sẽ chuyển sang một chunk công việc mới. Với *chunk\_size* là 1 thì cỡ của chunk công việc được tính bằng phép chia nguyên số lượng công việc cho số các luồng thực hiện và cỡ này sẽ giảm cho đến 1. Còn nếu *chunk\_size* có giá trị k thì cỡ của chunk công việc sẽ giảm dần cho đến k.
* Ví dụ:
* *#pragma omp parallel*
* *. . . .*
* *#pragma omp for schedule (guided, 1)*
* *for (int i=1;i<37 ; i++) a[i]=xxx;*



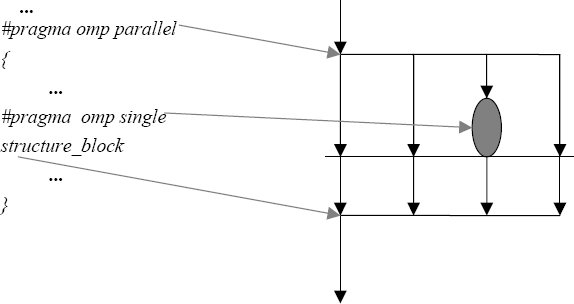
Hình 2.6 Mô tả sự hoạt động của các luồng với schedule là guide.

* RUNTIME
* Khi bắt gặp SCHEDULE(RUNTIME) thì công việc lập lịch bị hoãn lại cho đến khi runtime. Kiểu phân chia và cỡ của các chunk có thể thiết lập tại thời điểm các chunk bằng một biến môi trường có tên là OMP\_SCHEDULE. Nếu biến môi trường này không được thiết lập thì việc lập lịch chia sẻ công việc sẽ được thực hiện mặc định. Khi mà SCHEDULE(RUNTIME) được đưa ra thì *chunk\_size* sẽ không được khởi tạo.
* Mệnh đề ORDERED
* Mệnh đề **NOWAIT**
* Chỉ thị SECTIONS.
* *#pragma omp sections [clause. . . ] newline private (list)*
* *firstprivate (list) lastprivate (list) reduction (operator: list) nowait*
* *{*
* *#pragma omp section newline*
* *structured\_block*
* *#pragma omp section newline*
* *structured\_block*
* *}*



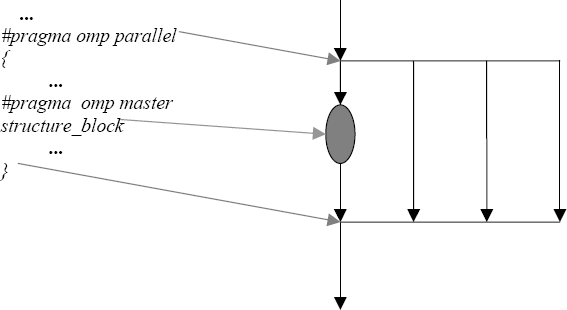
Hình 2.7 Sự hoạt động của các luồng qua chỉ thị sections.

* Một vấn đề đặt ra là có bao nhiêu chỉ thị SECTION cho phù hợp với sự thực thi của các thread, điều gì xảy ra khi số lượng các chỉ thị SECTION lớn hơn hay nhỏ hơn các thread. Khi số lượng chỉ thị SECTION nhỏ hơn các thread, các công việc trong chỉ thị SECTION vẫn được gán cho các thread tuy nhiên sẽ có một số thread không có đoạn mã hay công việc để thực hiện. Khi số lượng chỉ thị SECTION lớn hơn số thread, các đoạn mã hay công việc vẫn được gán cho các threads thực hịên theo kiểu quay vòng giống như mệnh đề schedule(static, *chunk\_size*).
* Chỉ thị SINGLE.
* Chỉ thị SINGLE chỉ ra rằng đoạn mã bao quanh chỉ thị SINGLE chỉ được thực hiện bởi một luồng trong tập các luồng trong vùng song song. Cấu trúc của chỉ thị SINGLE được cho bởi như sau:
* *#pragma omp single [clause. . . ] newline private(list)*
* *firstprivate(list) nowait Structure\_block*
* Các luồng khác mà không thực hiện đoạn mã trong chỉ thị SINGLE sẽ phải đợi cho đến khi luồng thực thi đoạn mã trong chỉ thị SINGLE thực hiện xong đoạn mã của mình mới được thực hiện công việc của mình trừ trường hợp có mệnh đề NOWAIT được đưa ra. Trong chỉ thị SINGLE có hai mệnh đề duy nhất đó là *private* và *firstprivate*.
* Ví dụ:



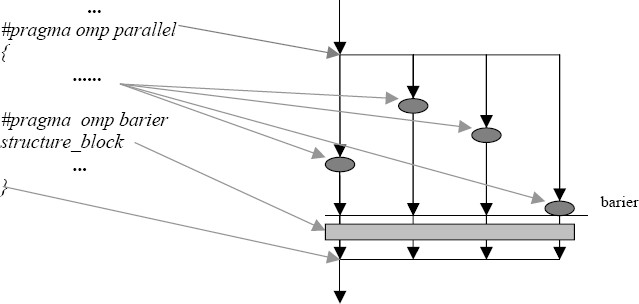
Hình 2.8 Sự hoạt động của các luồng qua chỉ thị single.

* Chỉ thị MASTER.
* Đoạn mã thuộc vùng song song trong chỉ thị MASTER chỉ được thực hiện duy nhết bởi luồng chủ. Cấu trúc của chỉ thị này được cho bởi như sau:
* *#pragma omp master newline struct\_block.*
* Ví dụ:
* Trong chỉ thị này không có bất kỳ chỉ thị nào và các luồng khác ngoài luồng chủ không cấn phải đợi cho đến khi luồng chủ thực hiện xong mới được thực hiện công việc của mình.



Hình 2.9 Sự hoạt động của các luồng qua chỉ thị master.

* Trong chỉ thị này không có bất kỳ chỉ thị nào và các luồng khác ngoài luồng chủ không cấn phải đợi cho đến khi luồng chủ thực hiện xong mới được thực hiện công việc của mình.
* Chỉ thị CRITICAL:
* Với chỉ thị CRITICAL, đoạn mã trong chỉ thị này chỉ được thực hiện bởi một luồng trong một thời điểm. Cấu trúc của chỉ thị cho bởi như sau:
* *#pragma omp critical [name ] newline struct\_block*
* Trong đoạn mã có thể có nhiều chỉ thị CRITICAL. Mỗi chỉ thị CRITICAL khác nhau sẽ có một tên khác nhau để trình biên dịch phân biệt giữa chỉ thị CRITICAL này với chỉ thị CRITICAL khác. Tất cả các chỉ thị CRITICAL không có tên hoặc có tên trùng nhau sẽ được coi như cùng một chỉ thị CRITICAL. Khi một luồng thực hiện công việc cho bởi chỉ thị mà luồng khác cố gắng để thực hiện thì luồng này sẽ bị khoá cho đến khi luồng kia thực hiện xong công việc đó.
* Chỉ thị BARRIER
* Chỉ thị BARRIER chỉ ra một điểm đồng bộ cho các luồng. Khi một luồng hay nhiều luồng bắt gặp chỉ thị BARRIER, chúng sẽ chờ ở đó cho đến khi tất cả các luồng hoàn thành công việc của mình, sau đó tất cả các luồng sẽ thực thi đoạn mã trong chỉ thị BARRIER. Cấu trúc của chỉ thị này cho bởi:
* *#pragma omp barrier newline struct\_block.*



Hình 2.10 Mô tả sự hoạt động của các luồng qua chỉ thị barrier.

* Chỉ thị ATOMIC.
* Trong chỉ thị AUTOMIC các địa chỉ vùng nhớ được cập nhật một cách nguyên tố. Khuôn dạng của chỉ thị này được cho bởi như sau:
* *#pragma omp atomic newline statment\_expression.*
* Chỉ thị này áp dụng trực tiếp một trong các câu lệnh sau: x binop = expr
* x++
* ++x x --
* --x
* x là biến mở rộng, không là cấu trúc hoặc lớp đối tượng. expr là một biểu thức mở rộng không tham chiếu đến biến x binop có thể là: + , \* , - , / , & , ^ , | , >= or <=
* Chỉ thị FLUSH
* Chỉ thị FLUSH được dùng để nhận ra một điểm đồng bộ. Điểm đồng bộ yêu cầu cung cấp một cái nhìn nhất quán về bộ nhớ. Tại thời điểm mà FLUSH xuất hiện, tất cả các biến thread-visiable phải được ghi trở lại bộ nhớ. Khuôn dạng của chỉ thị được cho bởi như sau:
* *#pragma omp flush (list) newline struct\_block.*
* Chú ý rằng danh sách lựa chọn ở đây chứa các biến flush để tránh flush tất cả các biến. Việc thực thi chỉ thị này phải đảm bảo rằng, bất kỳ lần sửa đổi biến thread-visiable lúc trước thì sau thời điểm đồng bộ phải được tất cả các luồng biết đến nó. Có nghĩa là trình biên dịch phải khôi phục từ thanh ghi ra bộ nhớ.
* Chỉ thị FLUSH được bao hàm bởi các chỉ thị sau: BARRIER, CRITICAL, ORDERED, PARALLEL, FOR, SECTIONS, SINGLE. Nhưng nếu có sự xuất hiện của mệnh đề NOWAIT thì chỉ thị FLUSH sẽ không được bao hàm.
* Chỉ thị ORDERED.
* Chỉ thị ORDERED được đưa ra để đảm bảo rằng, các công việc của vòng lặp phải được thực hiện đúng trình tự khi chúng được thực thi tuần tự. Khuôn dạng của chỉ thị được cho bởi như sau:
* *#pragma omp ordered newline struct\_block.*
* Một chỉ thị ORDERED chỉ có thể xuất hiện trong phạm vi động của chỉ thị FOR hoặc PARALLEL FOR trong C/C++. Tại bất cứ thời điểm nào thì chỉ có một luồng thực hiện đoạn mã cho bởi chỉ thị ORDERED. Nếu một vòng lặp chứa chỉ thị này thì nhất định nó phải chứa mệnh đề ORDERED.
* Chỉ thị THREADPRIVATE
* Chỉ thị này dùng để tạo ra các biến có phạm vi toàn cục trong toàn bộ chương trình. Các biến được khai báo trong chỉ thị này sẽ được sử dụng ở nhiều vùng song song khác nhau trong chương trình. Khuôn dạng của chỉ thị được cho bởi:
* *#pragma omp threadprivate(list)*
* Chỉ thị này phải xuất hiện trong phạm vi khai báo biến toàn cục. Các luồng khi sử dụng các biến trong chỉ thị này sẽ tạo ra bản sao của các biến đó để tránh việc sử dụng của biến này ảnh hưởng tới biến khác.
* Các mệnh đề trong OpenMP
* Do OpenMP lập trình trên máy tính chia sẻ bộ nhớ chung nên việc hiểu và sử dụng được phạm vi của các biến trong chương trình là rất quan trọng. OpenMP cung cấp một số mệnh đề giúp người lập trình dễ dàng thiết lập phạm vi các biến trong chương trình để phù hợp. Các mệnh đề bao gồm:
* PRIVATE FIRSTPRIVATE LASTPRIVATE SHARED DEFAULT REDUCTION COPYIN
* Mệnh đề PRIVATE
* Mệnh đề này dùng để khai báo các biến dùng riêng cho mỗi luồng. Mỗi luồng sẽ tạo ra một bản sao của biến trong quá trình thực hiện, sự sử dụng biến của luồng này sẽ không ảnh hưởng tới biến của luồng khác và ngược lại. Khuôn dạng của mệnh đề được cho bởi như sau:
* *private (list)*
* Mệnh đề FIRSTPRIVATE
* Mệnh đề này cũng để khai báo danh sách các biến được sử dụng riêng cho mỗi luồng, danh sách các biến được khởi tạo một giá trị ban đầu. Khuôn dạng của mệnh đề được cho bởi như sau:
* *fistprivate(list)*
* Mệnh đề LASTPRIVATE
* Mệnh đề này cũng dùng để khai báo danh sách các biến sử dụng riêng cho mỗi luồng, tuy nhiên nó khác mệnh đề PRIVATE và FIRSTPRIVATE ở chỗ giá trị cuối cùng của biến được cập nhật là giá trị của biến trong luồng cuối cùng kết thúc công việc. Khuôn dạng của mệnh đề này được khai báo như sau:
* *lastprivate (list)*
* Mệnh đề SHARED
* Mệnh đề này dùng để khai báo danh sách các biến được chia sẻ, dùng chung cho tất cả các luồng. Các biến chia sẻ có cùng vị trí bộ nhớ và các luồng sẽ đọc và ghi trên cùng vị trí ấy, sự thay đổi giá trị của biến của một luồng sẽ được các luồng khác biết đến, tuy nhiên vì các luồng cùng đọc và ghi lên cùng một địa chỉ cho nên có thể dẫn đến sai sót. Người lập trình phải phân bố công việc giữa các luồng sao cho hợp lý để tránh dẫn đến tình trạng sai sót. Khuôn dạng của mệnh đề này được cho bởi như sau:
* *shared (list)*
* Mệnh đề DEFAULT
* Mệnh đề này cho phép người lập trình đưa ra phạm vi PRIVATE, SHARED hoặc NONE cho tất cả các biến thuộc phạm vi của bất kỳ vùng song song nào, và chỉ có mệnh đề DEFAULT mới được đưa ra trong cấu trúc song song. Khuôn dạng của mệnh đề này được khai báo như sau:
* *default(shared | none)*
* Mệnh đề REDUCTION
* Mệnh đề này dùng để thu gọn giá trị của biến. Mỗi bản sao của biến cho bởi danh sách các sẽ được tạo cho mỗi luồng trong quá trình thực thi, tại thời điểm cuối cùng của việc rút gọn, các phép toán rút gọn sẽ áp dụng lên bản sao của mỗi luồng và kết quả của phép rút gọn sẽ được lưu vào biến chia sẻ. Khuôn dạng của mệnh đề này được cho bởi như sau:
* *reduction ( operator: list)*
* Trong đó operator là: x = x op expr
* x = expr op x x binop = expr x ++
* ++ x
* x – , -- x
* x là biến vô hướng trong danh sách các biến.
* expr là một biểu thức vô hướng không tham chiếu đến biến x op là một trong những phép toán: +, -, \*, /, &, ^, |, &&, || binop là một trong những phép toán: +, -, \*, /, &, ^, |
* Mệnh đề COPYIN
* *copyin(list)*
* Thư viện Runtime (Runtime Library Routine).
* OpenMp cung cấp một thư viện với rất nhiều các hàm chức năng bao gồm các truy vấn liên quan tới số lượng, chỉ số, thiết lập các luồng được thực thi trong chương trình và các hàm thiết lập môi trường thực thi giúp người lập trình dễ dàng sử dụng và quản lý chương trình ứng dụng song song của mình. Hầu hết các hàm thư viện chứa trong tệp tiêu đề omp. h, do vậy để sử dụng được các thư viện này, khi khai báo tệp tiêu đề chúng ta phải khai báo **#include <omp. h>**
* OMP\_SET\_NUM\_THREADS
* Hàm thư viện này dùng để thiết lập tổng số luồng thực thi trong vùng song song tiếp theo. Khuôn mẫu của hàm này có dạng.
* *void omp\_set\_num\_threads(int)*
* OMP\_GET\_NUM\_THREADS
* Hàm này trả về giá trị là tổng số luồng được thực thi trong vùng mà nó được gọi. Khuôn mẫu của hàm này có dạng:
* *int omp\_get\_num\_threads(void)*
* Nếu hàm này được gọi trong vùng tuần tự nó sẽ trả về giá trị 1 điều đó có nghĩa là chỉ có một luồng được thực thi. Nếu hàm này được gọi trong vùng song song nó sẽ trả về giá trị là tổng số luồng được thực thi trong vùng song song đó.
* OMP\_GET\_THREAD\_NUM
* Hàm này trả về giá trị là chỉ số của luồng đang thực thi trên đoạn mã mà hàm này được gọi. Chỉ số của luồng bắt đầu từ 0 tới tổng số luồng -1. Khuôn mẫu của hàm này có dạng:
* *int omp\_get\_thread\_num(void)*
* OMP\_GET\_MAX\_THREADS
* Hàm này cũng tương tự như hàm *omp\_get\_num\_threads()* tuy nhiên nó khác hàm *omp\_get\_num\_threads()* ở chỗ nó sẽ trả về giá trị lớn nhất là số luồng có thể tạo ra trong vùng song song. Khuôn mẫu của hàm này được cho bởi:
* *int omp\_get\_max\_threads()*
* OMP\_GET\_NUM\_PROCS
* Hàm này trả về giá trị là số bộ xử lý đang được thực thi của hệ thống.
* Khuôn mẫu của hàm này có dạng:
* *int omp\_get\_num\_procs()*
* Hàm này được gọi trong vùng tuần tự.
* OMP\_IN\_PARALLEL
* Hàm này kiểm tra xem sự thực thi của các luồng có phải là song song hay không. Khuôn mẫu của hàm này có dạng:
* *int omp\_in\_parallel().*
* Hàm này được gọi từ vùng song song và nếu các luồng thực thi đoạn mã song song, hàm sẽ trả về giá trị khác 0. Nếu đoạn mã được thực hiện tuần tự nó sẽ trả về giá trị bằng 0.
* OMP\_SET\_DYNAMIC
* Hàm này cho phép hay không cho phép có sự điều chỉnh động các luồng trong vùng song song. Khuôn mẫu của hàm này có dạng như sau:
* *void omp\_set\_dynamic(int dynamic\_thread)*
* Nếu dynamic\_thread khác 0 có nghĩa là cho phép sự điều chỉnh động các luồng xảy ra có nghĩa là các luồng có thể thực thi hơn một vùng song song. ngược lại không cho phép sự điều chỉnh động các luồng.
* OMP\_GET\_DYNAMIC
* Hàm này dùng để kiểm tra xem có sự điều chỉnh động của các luồng hay không. Khuôn mẫu của hàm này có dạng như sau.
* *int omp\_get\_dynamic()*
* Nếu hàm này trả về giá trị khác 0 nghĩa là có sự điều chỉnh động giữa các luồng, ngược lại không có sự điều chỉnh động giữa các luồng.
* OMP\_SET\_NESTED
* Hàm này cho phép hay không cho phép việc song song lồng. Khuôn mẫu của hàm này có dạng:
* *void omp\_set\_nested(int nested)*
* Hàm này được gọi cả trong vùng tuần tự lẫn song song. Đối số nested trong hàm này là số luồng được phép lồng trong vùng song song. Nếu nested bằng 0 tức là không cho phép sự song song lồng xảy ra, ngược lại nếu đối số của nested khác 0 thì sự thực hiện song song lồng sẽ xảy ra.
* OMP\_GET\_NESTED
* Hàm này dùng để kiểm tra xem có sự song song lồng xảy ra hay không.
* Khuôn mẫu của hàm này có dạng:
* *int omp\_get\_nested()*
* Hàm này bắt buộc phải được gọi trong vùng có đoạn mã song song lồng. Nếu hàm trả về giá trị khác 0 nghĩa là có việc song song lồng xảy ra, ngược lại hàm trả về giá trị bằng 0.
* OMP\_INIT\_LOCK
* Hàm này dùng để thiết lập một khoá thông qua các biến khoá. Khuôn mẫu của hàm này có dạng:
* *void omp\_init\_lock(omp\_lock\_t \*lock)*
* *void omp\_init\_nest\_lock(omp\_nest\_lock\_t \*lock)*
* OMP\_DESTROY\_LOCK
* Hàm này dùng để tách ra các biến khoá từ bất kỳ khoá nào. khuân mẫu của hàm này có dạng như sau:
* *void omp\_destroy\_lock(omp\_lock\_t \*lock)*
* *void omp\_destroy\_nest\_lock(omp\_lock\_t \*lock)*
* OMP\_SET\_LOCK
* Hàm này dùng để bắt buộc sự thực hiện của các luồng phải chờ đợi khi khoá được mở với giả sử rằng các luồng đó được quyền sở hữu khoá đó. Khuôn mẫu của hàm có dạng như sau:
* *void omp\_set\_lock(omp\_set\_t \*lock)*
* *void omp\_set\_nest\_lock(omp\_set\_nest\_t \*lock)*
* OMP\_UNSET\_LOCK
* Hàm này dùng để giải thoát sự thực hiện của các luồng vào khóa. Khuôn mẫu của hàm này có dạng như sau:
* *void omp\_unset\_lock(omp\_unset\_t \*lock)*
* *void omp\_unset\_nest\_lock (omp\_unset\_nest\_t \*lock)*
* OMP\_TEST\_LOCK
* Hàm này được sử dụng để cố gắng thử dặt một khoá. Nếu thành công nó sẽ trả về giá trị khác 0 ngược lại nó trả về giá trị bằng 0. Khuôn mẫu của hàm này có dạng:
* *int omp\_test\_lock(omp\_lock\_t \*lock)*
* *int omp\_test\_nest\_lock(omp\_nest\_t \*lock)*
* Các biến môi trường (Enviroment Variables).
* Ngoài thư viện runtime OpenMP còn cung cấp cho người lập trình rất một số các biến môi trường, giúp người lập trình thuận tiện trong việc điều khiển các đoạn mã song song trong chương trình của mình. Các biến môi trường bao gồm:
* OMP\_SCHEDULE
* Biến này cũng giống như mệnh đề schedule. Dùng để lập lịch sự thực hiện các công việc trong vòng lặp các luồng thực hiện.
* Ví dụ :
* setenv OMP\_SCHEDULE ,,,, static, 4
* OMP\_NUM\_THREADS
* Biến này giống như hàm thư viện omp\_set\_num\_threads(). Dùng để thiết lập số lượng các luồng thực hiện trong vùng song song.
* Ví dụ :
* setenv OMP\_NUM\_THREADS 8
* Thiết lập số lượng luồng thực thi trong vùng song song là 8 luồng.
* OMP\_DYNAMIC
* Biến này dùng để thiết lập sự điều chỉnh động các luồng. Nó nhận hai giá trị TRUE hoặc FALSE, nếu biến này được thiết lập với giá trị TRUE tức là có cho phép sự điều chỉnh động các luồng thực thi trong vùng song song, ngược lại không cho phép sự điều chỉnh động các luồng thực thi trong vùng song song.
* Ví dụ :
* setenv OMP\_DYNAMIC TRUE
* OMP\_NESTED
* Biến này dùng để thiết lập cho phép hay không cho phép vùng song song lồng xảy ra. nó nhận hai giá trị TRUE hoặc FALSE. Nếu biến này được thiết lập với giá trị TRUE tức là có cho phép vùng song song lồng xảy ra, ngược lại không cho phép vùng song song lồng xảy ra.
* Ví dụ: setenv OMP\_NESTED TRUE.

**CHƯƠNG 3: Ứng dụng OpenMP trong xử lý ảnh**

1. **Bài toán đổi tên định dạng 8 nghìn tấm ảnh từ “.jpg” sang “.png”**

* Để thực hiện tính toán song song, em chia việc đổi định dạng ảnh thành 2 section. Mỗi một section sẽ thực hiện việc đổi đuôi 4000 tấm ảnh để thực hiện việc tính toán song song
* Một thực nghiệm em sẽ để thực hiện tính toán tuần tự.
* Phần cứng máy tính: CPU 2 cores, 4 threads, 8Gb Ram
* Source code thực hiện bài toán:

#include "opencv2/highgui/highgui.hpp"

#include "opencv2/features2d/features2d.hpp"

#include <iostream>

#include <vector>

#include <omp.h>

/\*

This is small function to test using openMP library in parallel computing and

not using lib openMP: convert 8k images from .jpg file to .png file

When execute with openMP lib, time to end process is: 2m30s

When execute with no openMP lib, time to end process is: 3m

8k images can be download at: http://academictorrents.com/download/9dea07ba660a722ae1008c4c8afdd303b6f6e53b.torrent

\*/

void convertWithOpenMP()

{

cv::String path("D:\\Dev\\Flickr8k\\Flickr8k\_Dataset\\Flicker8k\_Dataset\\\*.jpg");

std::vector<cv::String> fn;

std::vector<cv::Mat> data;

cv::glob(path, fn, true);

#pragma omp parallel

{

#pragma omp sections

{

#pragma omp section

for (size\_t k = 0; k < fn.size() / 2; ++k)

{

cv::Mat im = cv::imread(fn[k]);

//if (im.empty()) continue; //only proceed if sucsessful

// you probably want to do some preprocessing

//data.push\_back(im);

cv::imwrite("D:\\Dev\\Flickr8k\\out\\" + std::to\_string(k) + ".png", im);

}

#pragma omp section

for (size\_t k = fn.size() / 2; k < fn.size(); ++k)

{

cv::Mat im = cv::imread(fn[k]);

//if (im.empty()) continue; //only proceed if sucsessful

// you probably want to do some preprocessing

//data.push\_back(im);

cv::imwrite("D:\\Dev\\Flickr8k\\out\\" + std::to\_string(k) + ".png", im);

}

}

}

std::cout << "Done" << std::endl;

}

void convertWithoutOpenMP()

{

cv::String path("D:\\Dev\\Flickr8k\\Flickr8k\_Dataset\\Flicker8k\_Dataset\\\*.jpg");

std::vector<cv::String> fn;

std::vector<cv::Mat> data;

cv::glob(path, fn, true);

for (size\_t k = 0; k < fn.size(); ++k)

{

cv::Mat im = cv::imread(fn[k]);

cv::imwrite("D:\\Dev\\Flickr8k\\out\\" + std::to\_string(k) + ".png", im);

}

std::cout << "Done" << std::endl;

}

int main()

{

// with openMp

convertWithOpenMP();

// without openMp

convertWithoutOpenMP();

system("pause");

}

1. Nhận xét

* Khi thực hiện tuần tự, thời gian thực hiện xong là: 3 phút
* Khi thực hiện tính toán song song dùng thư viện OpenMP, thời gian thực hiện là 2 phút 30 giây.
* Bài toán chỉ là một ví dụ nhỏ trong việc thực hiện tính toán song song trên máy tính bởi ứng dụng thư viện mã nguồn mở OpenMP. Với phần cứng máy tính hạn chế, thời gian cải thiện khoảng 30s. Tuy nhiên, giả sử khối lượng tính toán lớn, phức tạp thì việc tính toán song song có thể giúp chúng ta cải thiện đáng kể thời gian thực hiện bài toán.
* Mở rộng trong xử lý ảnh: ta có thể áp dụng để tối ưu một số bài toán liên quan đến detect Object, xử lý tính chất của ảnh. Có thể áp dụng trong việc viết ra các tool auto crop, copy, dán nhãn cho ảnh để làm đầu vào data set trong Deep learning, Machine learning…

**Tài liệu tham khảo**

[1]. Chandra, R., Dagum, L., Kohr, D., Maydan, D., McDonald, J., Menon, R. Parallel Programming in OpenMP. Academic Press. Morgan Kaufmann. 2001.

[2]. Mario Soukup. A Source-to-Source OpenMP Compiler, Master Thesis, Department of Electrical and Computer Engineering. University of Toronto. 2001.

[3]. OpenMP C and C++ Application Program Interface Version 2.0 March 2002

[4]. [http://www.](http://www/) llnl. gov/computing/tutorials/parallel\_comp [5]. [http://www.](http://www/) openmp. org

[6]. [http://www.](http://www/) llnl/computing/tutorials/workshop/openmp/ [7]. [http://www](http://www/) hpcc. unical. it/alarico/LNErbacci2. pdf

[8]. <http://nereida.deicc.ull.es/html/openmp/minnrsota/tutorial/content_op> enmp. html

[9]. [http://wikipedia.](http://wikipedia/) org/wiki/OpenMP.

[10].Phát triển ứng dụng song song với OpenMP - Trịnh Công Quý - Đại Học Quốc Gia Hà Nội.