**LỜI MỞ ĐẦU**

Hà nội, ngày 30 tháng 10 năm 2019.

SINH VIÊN

Vũ Văn Cảnh

Mục lục

[**Lời mở đầu** 1](#_TOC_250076)

[**Mục lục** 2](#_TOC_250075)

[**CHƯƠNG 1**: Tổng quan về lập trình song song, tính toán song song 7](#_TOC_250073)

[Định nghĩa 7](#_TOC_250072)

[Thế nào là lập trình, tính toán song song 7](#_TOC_250071)

[Tại sao phải lập trình, tính toán song song 7](#_TOC_250070)

[Sử dụng lập trình, tính toán song song để làm gì 7](#_TOC_250069)

[So sánh lập trình tính toán tuần tự và lập trình tính toán song song 8](#_TOC_250068)

[Sự phân chia cấu trúc tính toán song song 9](#_TOC_250067)

[Phân chia dựa trên quan hệ giữa chỉ thị lệnh và dữ liệu 9](#_TOC_250066)

[Sự phân chia dựa trên mối quan hệ giữa bộ xử lý và bộ nhớ 12](#_TOC_250065)

[Các mô hình lập trình song song 16](#_TOC_250064)

[Mô hình dùng chung bộ nhớ (Shared Memory) 17](#_TOC_250063)

[Mô hình luồng (Thread) 17](#_TOC_250062)

[Mô hình truyền thông điệp (Message Passing) 18](#_TOC_250061)

[Mô hình song song dữ liệu (Data Parallel) 19](#_TOC_250060)

[Một số vấn đề liên quan đến lập trình và tính toán song song 20](#_TOC_250059)

[Định luật Amdahl‟s 20](#_TOC_250058)

[Cân bằng tải 21](#_TOC_250057)

[Sự bế tắc 22](#_TOC_250056)

[**CHƯƠNG 2**: Thư viện Mã nguồn mở OpenMP và ứng dụng 24](#_TOC_250055)

[Tổng quan về OpenMP 24](#_TOC_250054)

[Giới thiệu 24](#_TOC_250053)

[Định nghĩa 24](#_TOC_250052)

[Lịch sử phát triển 24](#_TOC_250051)

[Mục đích của OpenMP 25](#_TOC_250050)

[Mô hình lập trình song song trong OpenMP 25](#_TOC_250049)

[Các chỉ thị biên dịch (Compiler Directive) 26](#_TOC_250048)

[Khuôn dạng của chỉ thị 26](#_TOC_250047)

[Phạm vi của chỉ thị 27](#_TOC_250046)

[Cấu trúc vùng song song 28](#_TOC_250045)

[Cấu trúc chia sẻ công việc (Work Sharing Construct) 29](#_TOC_250044)

[Cấu trúc đồng bộ 37](#_TOC_250043)

[Chỉ thị THREADPRIVATE 41](#_TOC_250042)

[Các mệnh đề trong OpenMP 41](#_TOC_250041)

[Mệnh đề PRIVATE 41](#_TOC_250040)

[Mệnh đề FIRSTPRIVATE 42](#_TOC_250039)

[Mệnh đề LASTPRIVATE 42](#_TOC_250038)

[Mệnh đề SHARED 42](#_TOC_250037)

[Mệnh đề DEFAULT 42](#_TOC_250036)

[Mệnh đề REDUCTION 43](#_TOC_250035)

[Mệnh đề COPYIN 43](#_TOC_250034)

Thư viện Runtime (Runtime Library Routine) 44

[OMP\_SET\_NUM\_THREADS 44](#_TOC_250033)

[OMP\_GET\_NUM\_THREADS 45](#_TOC_250032)

[OMP\_GET\_THREAD\_NUM 45](#_TOC_250031)

[OMP\_GET\_MAX\_THREADS 45](#_TOC_250030)

[OMP\_GET\_NUM\_PROCS 45](#_TOC_250029)

[OMP\_IN\_PARALLEL 45](#_TOC_250028)

[OMP\_SET\_DYNAMIC 46](#_TOC_250027)

[OMP\_GET\_DYNAMIC 46](#_TOC_250026)

[OMP\_SET\_NESTED 46](#_TOC_250025)

[OMP\_GET\_NESTED 47](#_TOC_250024)

[OMP\_INIT\_LOCK 47](#_TOC_250023)

[OMP\_DESTROY\_LOCK 47](#_TOC_250022)

[OMP\_SET\_LOCK 47](#_TOC_250021)

[OMP\_UNSET\_LOCK 47](#_TOC_250020)

[OMP\_TEST\_LOCK 48](#_TOC_250019)

Các biến môi trường (Enviroment Variables) 48

[OMP\_SCHEDULE 48](#_TOC_250018)

[OMP\_NUM\_THREADS 48](#_TOC_250017)

[OMP\_DYNAMIC 48](#_TOC_250016)

[OMP\_NESTED 49](#_TOC_250015)

[**CHƯƠNG 3**: Ứng dụng OpenMP 50](#_TOC_250014)

[Ứng dụng openMP trong xử lý ảnh 50](#_TOC_250013)

[Kết luận 64](#_TOC_250001)

[Tài liệu tham khảo 65](#_TOC_250000)

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

**Định nghĩa:**

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.

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.

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. |

Sự phân chia cấu trúc tính toán song song.

1.2.1 Phân chia dựa trên quan hệ giữa chỉ thị lệnh và dữ liệu

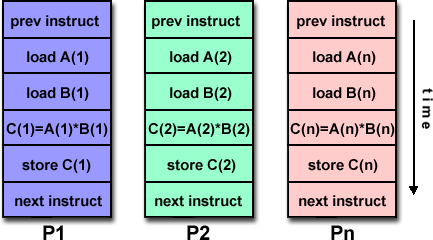
Dựa vào mối quan hệ giữa chỉ thị lệnh và dữ liệu chia làm các loại :

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.

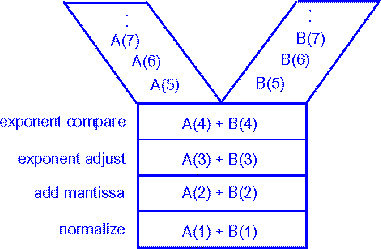
Khi một chỉ thị phát ra, tất cả các quá trình xử lý được thực hiện.

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.

 Parallel SIM.

Vector SIMD

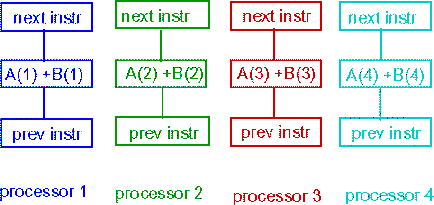


Hình 1.2 Mô hình vector SIMD

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.



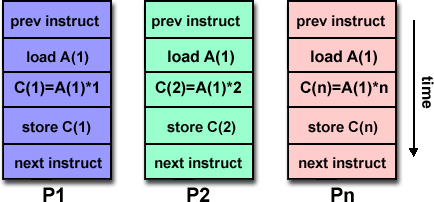
Hình 1.3 Mô hình parallel SIMD

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.

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



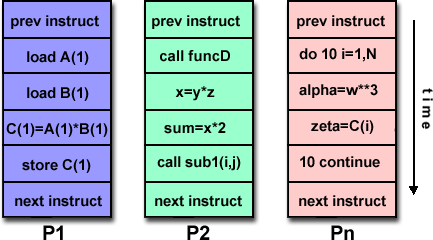
Hình 1.4 Mô hình đa chỉ thị đơn dữ liệu

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).



Hình 1.5 Mô hình đa chỉ thị, đa dữ liệu

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).

Dùng chung bộ nhớ (Shared Memory).

Dùng chung bộ nhớ (Shared memory) là tất cả các bộ xử lý đều có thể truy cập bộ nhớ và gọi là bộ nhớ toàn cục.

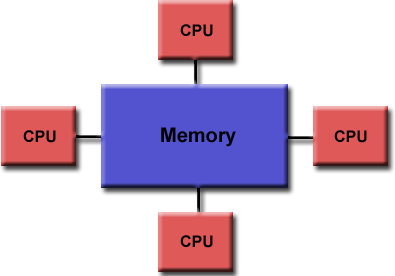
Các bộ xử lý có thể thao tác, hoạt động một cách độc lập nhưng dùng chung tài nguyên bộ nhớ.

Sự thay đổi trong bộ nhớ của một bộ vi xử lý sẽ thông báo cho tất cả các bộ vi xử lý khác biết.

Cấu trúc dùng chung bộ nhớ (Shared memory) được chia làm hai loại.  Truy cập bộ nhớ đồng bộ (Uniform Memory Access).

 Truy cập bộ nhớ không đồng bộ (Non Uniform Memory Access).

Truy cập bộ nhớ đồng bộ (Uniform Memory Access).



Hình 1.6 Mô hình truy cập bộ nhớ đồng bộ.

Thường được gọi là SMP

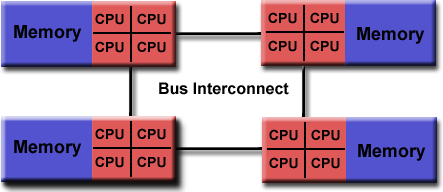
Có các bộ vi xử lý giống nhau.

Bằng nhau về tốc độ truy cập bộ nhớ và thời gian truy cập bộ nhớ.

Thỉnh thoảng còn được gọi là CC – UMA (Cache Coherent UMA).

CC – UMA nghĩa là nếu một vi xử lý cập nhật vị trí trong bộ nhớ dùng chung thì tất cả các bộ vi xử lý khác sẽ biết được thông tin cập nhật.

Truy cập bộ nhớ không đồng bộ (Non Uniform Memory Access)



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

Thường là sự kết nối mạng của hai hay nhiều SMP

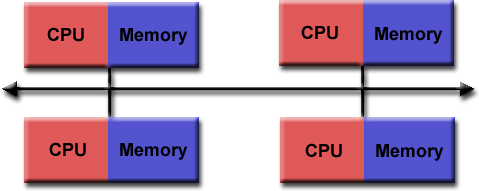
Mỗi SMP không thể truy cập trực tiếp bộ nhớ từ các SMP khác.

Không phải tất cả các bộ xử ký đều có thời gian truy cập tất cả bộ nhớ dùng chung như nhau.

Sự truy cập bộ nhớ thông qua kết nối sẽ chậm hơn.

Các SMP trao đổi thông qua truyền và nhận thông điệp (Message Passing).

Phân bổ bộ nhớ (Distributed Memory).



Hình 1.8 Mô hình phân bổ bộ nhớ.

Phân bổ bộ nhớ (Distributed Memoy) là sự kết nối mạng nhiều bộ nhớ của các bộ vi xử lý.

Mỗi bộ xử lý có bộ nhớ cục bộ riêng, bộ nhớ cục bộ của bộ xử lý này sẽ không có sự truy cập, sử dụng đối với bộ xử lý khác và ngược lại do vậy sẽ không có khái niệm địa chỉ bộ nhớ toàn cục cho các bộ vi xử lý.

Vì mỗi bộ xử lý có bộ nhớ cục bộ riêng nên các bộ xử lý thao tác một cách độc lập. Sự truy cập, thay đổi trên bộ nhớ cục bộ của mỗi bộ xử lý sẽ không làm ảnh hưởng tới các bộ xử lý khác và ngược lại.

Khi một bộ xử lý cần trao đổi với bộ xử lý khác, thông thường công việc của người lập trình sẽ phải định nghĩa rõ khi nào, bao giờ dữ liệu được trao đổi.

Người lập trình sẽ chịu trách nhiệm đồng bộ giữa các công việc.

Thông thường mô hình kết nối giữa các bộ xử lý là mạng Ethernet.

Ưu điểm :

 Số bộ nhớ sẽ cân bằng với số bộ xử lý. Sự tăng lên về số lượng bộ xử lý thì kích cỡ bộ nhớ sẽ tăng lên một cách cân đối.

 Mỗi bộ xử lý truy rất nhanh bộ nhớ của chúng mà không có bất cứ sự can thiệp nào và không có thời gian chờ để truy cập.

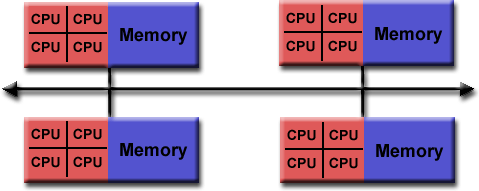
Nhược điểm:

 Người lập trình sẽ phải chịu trách nhiệm trao đổi dữ liệu giữa các bộ xử lý.

 Rất khó cho việc tổ chức dữ liệu đối với kiểu tổ chức bộ nhớ này.

 Là mô hình truy cập bộ nhớ không đồng bộ (Non–Uniform Memory Access)

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



Hình 1.9 Mô hình kết hợp.

Đây là mô hình kết nối rộng và chắc chắn của nhiều máy tính khác nhau.

Mỗi thành phần Shared Memory là một máy tính có cấu trúc SMP. Các bộ xử lý trên một SMP có địa chỉ toàn cục trên chính máy SMP đó.

Thành phần Distributed Memory là mạng kết nối của nhiều SMP. Các SMP biết duy nhất bộ nhớ toàn cục của chúng mà không biết bộ nhớ toàn cục của các SMP khác. Do vậy mô hình này cần có sự trao đổi dữ liệu giữa các SMP khác nhau.

Hiện tại nó là xu hướng phát triển của cấu trúc bộ nhớ trong tính toán song song trong tương lai.

Ưu điểm và nhược điểm: đây là mô hình kết hợp của hai mô hình Shared Memory và Distributed memory nên nó mang những ưu, nhược điểm của cả hai mô hình này

Các mô hình lập trình song song.

**Một số mô hình lập trình song song thƣờng 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).

Mô hình dùng chung bộ nhớ (Shared Memory)

Trong mô hình lập trình dùng chung bộ nhớ, các thao tác, nhiệm vụ sử dụng chung bộ nhớ. Chúng truy cập, đọc, ghi vào bộ nhớ một cách đồng bộ.

Các khoá, cờ hiệu được dùng để điều khiển sự truy cập bộ nhớ.

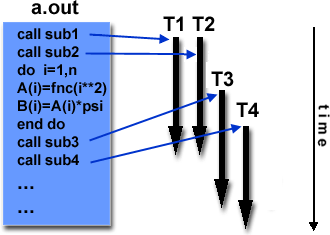
Một điểm thuận lợi trong mô hình lập trình này là không có sự nắm giữ dữ liệu, do vậy không cần phải chỉ rõ sự trao đổi dữ liệu giữa các công việc. Như vậy lập trình viên dễ phát triển ứng dụng của mình hơn.

Một điểm bất lợi trong mô hình lập trình này là rất khó để hiểu và quản lý dữ liệu.

Sự điều khiển dữ liệu rất khó khăn và nằm ngoài tầm của người sử dụng.

Trong mô hình lập trình này, chương trình dịch sẽ chuyển biến trong chương trình thành địa chỉ bộ nhớ và đó là địa chỉ toàn cục.

Mô hình luồng (Thread)



Hình 1.10 Mô hình luồng.

Trong mô hình luồng, một quá trình xử lý có thể thực thi nhiều luồng khác nhau một cách đồng thời.

Luồng (Thread) là một khái niệm dùng để mô tả một chương trình chính có nhiều chương trình, thủ tục con mà khi thực hiện chương trình chính, các chương trình, thủ tục con được thực hiện song song.

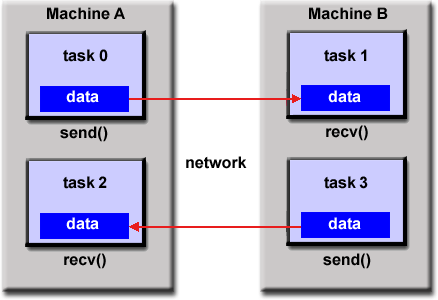
 Khi chương trình chính thực thi, nó thực hiện mốt số bước tuần tự và tạo ra các Thread mà sau đó được thực hiện một cách đồng thời.

 Mỗi Thread có dữ liệu cục bộ nhưng chúng dùng chung tài nguyên của chương trình chính.

 Công việc của mỗi Thread là chương trình, thủ tục con trong chương trình chính. Mỗi Thread có thể thực thi các chương trình, thủ tục con cùng khoảng thời gian với các Thread khác.

 Các Thread trao đổi với nhau thông qua bộ nhớ toàn cục bằng cách cập nhật địa chỉ bộ nhớ toàn cục.

Mô hình truyền thông điệp (Message Passing)



Hình 1.11 Mô hình truyền thông điệp.

Mô hình truyền thông điệp được định nghĩa là :

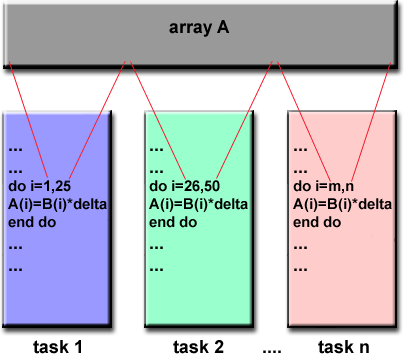
 Đặt quá trình xử lý sử dụng một bộ nhớ cục bộ.

 Các bộ xử lý trao đổi với nhau thông qua việc gửi và nhận các thông điệp.

 Sự di chuyển dữ liệu yêu cầu sự kết hợp thao tác thực hiện của mỗi quá trình xử lý (truyền nhận thông điệp một cách nhịp nhàng).

Lập trình với mô hình truyền thông điệp sẽ liên kết tới các thư viện để quản lý dữ liệu trao đổi giữa các bộ xử lý. Các thư viện này có sẵn trong một số ngôn ngữ lập trình.

Mô hình song song dữ liệu (Data Parallel).



Hình 1. 12 Mô hình song song dữ liệu.

Mô hình song song dữ liệu (Data parallel) được định nghĩa là:

 Mỗi quá trình xử lý công việc được thực hiện trên một thành phần của cấu trúc dữ liệu.

 Thường áp dụng với chương trình nhiều dữ liệu Single Program Multiple Data (SPMD)

 Dữ liệu của chương trình sẽ được chia cho các bộ xử lý.

 Người lập trình không thấy được quá trình trao đổi dữ liệu.

 Thường được xây dựng theo kiểu “on top of ” một kiểu của thư viện Message Passing.

Khi lập trình với mô hình này, người lập trình phải viết chương trình với cấu trúc dữ liệu song song và dịch với chương trình dịch dữ liệu song song (Data parallel Compiler).

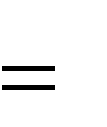
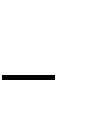
Chương trình dịch sẽ dịch chương trình thành mã chuẩn và gọi tới thư viện Message Passing để chia dữ liệu cho tất cả quá trình xử lý.

Một số vấn đề liên quan đến lập trình và tính toán song song.

1.4.1 Định luật Amdahl’s

Định luật được Amdahl‟s phát biểu vào năm 1967 nhằm đánh giá hiệu năng của việc tính toán song song. Định luật được phát biểu như sau:

Hiệu năng tính toán của chương trình được định nghĩa là phân số của đoạn mã mà được thực thi song song.



1

1 *P*

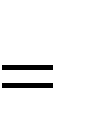
Speedup

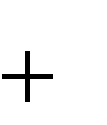
Nếu không có đoạn mã được thực thi song song, P = 0 speedup = 1 (Không cải thiện được tốc độ)

Nếu tất cả các đoạn mã được thực thi song song, P = 1 speedup tăng lên đến vô cùng.

Nếu 50% đoạn mã được thực thi song song, speedup đạt giá trị max. Công thức nêu lên mối quan hệ giữa hiệu năng tính toán với bộ xử lý.

Speedup *1*



*P S*

*N*

Trong đó:

P: là phân số song song.

N: là số bộ xử lý.

S: là phân số tuần tự.

Cân bằng tải.

Thông thường trong quá trình thực hiện xử lý song song, dữ liệu được phân bố trên các bộ nhớ cục bộ của các bộ vi xử lý, khi đó khối lượng công việc cần phải phân phối hợp lý trong suốt quá trình tính toán. Tuy nhiên trong thực tế không phải lúc nào điều này cũng được thực hiện, vì vậy sẽ xảy ra trường hợp một số bộ xử lý thực hiện quá tải trong khi đó một số bộ xử lý lại không thực hiện hết khả năng tính toán của mình. Giải pháp được đặt ra là cân bằng tải động để phân phối công việc cho phù hợp với các bộ xử lý.

Thông thường khi phân phối xong công việc cho các bộ xử lý, quá trình cân bằng tải động sẽ thực hiện theo các bước sau đây.

Giám sát hiệu năng của mỗi bộ xử lý.

Trao đổi thông tin trạng thái giữa các bộ xử lý.

Tính toán và ra quyết định phân phối lại công việc.

Thực hiện chuyển đổi dữ liệu cho các bộ xử lý.

Để thực hiện được điều này có rất nhiều thuật toán được đưa ra tuy nhiên chúng được phân lớp thành các lớp sau:

Cân bằng tải tập trung: Nhằm đưa ra các quyết định có tính chất tổng thể trong việc thực hiện phân phối các công việc cho các bộ xử lý. Các thuật toán trong lớp này sử dụng thông tin hệ thống có tính chất toàn cục để lưu lại trạng thái của các bộ xử lý. Các thông tin này sẽ cho phép thuật toán phân phối lại công việc cho các bộ xử lý một cách dễ dàng. Tuy nhiên khối lượng công việc tăng theo tỷ lệ thuận với số lượng các bộ xử lý do

vậy cần phải có số lượng lớn bộ nhớ trên các bộ xử lý để lưu trữ thông tin. Vì vậy các thuật toán thuộc lớp này ít được sử dụng.

Cân bằng tải phân tán hoàn toàn: Trong chiến lược này mỗi bộ xử lý có một bản sao về thông tin trạng thái của hệ thống. Các bộ xử lý trao đổi thông tin trạng thái với nhau và sử dụng các thông tin này để làm thay đổi một cách cục bộ việc phân chia công việc. tuy nhiên các bộ xử lý chỉ có thông tin cục bộ nên việc cân bằng tải không tốt bằng các thuật toán cân bằng tải tập trung.

Cân bằng tải phân tán một nửa: Các thuật toán này chia các bộ xử lý ra thành từng miền, mỗi miền sử dụng thuật toán cân bằng tải tập trung để phân chia khối lượng công việc cho các bộ xử lý.

Sự bế tắc.

Sự bế tắc xảy ra khi có hơn một hoặc nhiều bộ xử lý cùng sử dụng chung một tài nguyên hệ thống mà không có sự kiểm soát tốt. Sự bế tắc xảy ra trong các hệ điều hành đa nhiệm, các hệ thống đa bộ xử lý và đa máy tính.

Đối với các hệ thống đa máy tính, sự bế tắc phổ biến là bế tắc vùng đệm. Sự bế tắc vùng đệm xảy ra khi một tiến trình đợi một thông điệp mà thông điệp này có thể không bao giờ nhận được khi vùng đệm của hệ thống đã bị đầy.

Các điều kiện gây lên sự bế tắc.

Sự loại trừ lẫn nhau: Mỗi tiến trình có sự độc quyền khi sử dụng tài nguyên riêng của nó.

Không có sự ưu tiên: Mỗi tiến trình không bao giờ giải phóng tài nguyên mà tiến trình đó đang chiếm giữ cho đến khi không còn sử dụng chúng nữa.

Sự chờ đợi tài nguyên: Mỗi tiến trình đang chiếm giữ tài nguyên trong khi lại đợi tiến trình khác giải phóng tài nguyên cho chúng.

Sự chờ đợi giữa các tiến trình: Tiến trình đợi tài nguyên mà tiến trình kế tiếp đang chiếm giữ mà tài nguyên đó không được giải phóng.

Một số cách khắc phục:

Cách thứ nhất ta sử dụng là dò tìm sự bế tắc khi chúng xảy ra và khôi phục lại. Một cách khác là sử dụng các thông tin yêu cầu tài nguyên của các tiến trình để phân phối tài nguyên cho hợp lý tránh rơi vào tình trạng bế tắc. Cách thứ ba là ngăn cấm không để xảy ra đồng thời ba điều kiện cuối trong các điều kiện nảy sinh bế tắc.

CHƯƠNG 2: Thư viện Mã nguồn mở OpenMP

Tổng quan về OpenMP.

Giới thiệu

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++.

Định nghĩa.

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.

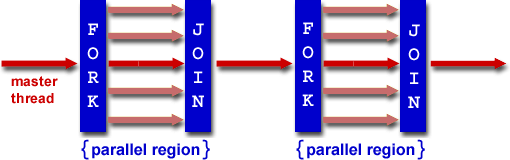
Phiên bản 3. 0 được công bố vào năm 2008 và nó là phiên bản hiện tại được tích hợp thêm nhiều tính năng mới.

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.



Hình 2.1 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.

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.

. . . .

}

Phạm vi của chỉ thị.

Phạm vi tĩnh (static extent).

Phạm vi tĩnh của chỉ thị đựợc tính từ bắt đầu khi khai báo chỉ thị cho đến khi gặp dấu kết thúc của chỉ thị trong vùng song song.

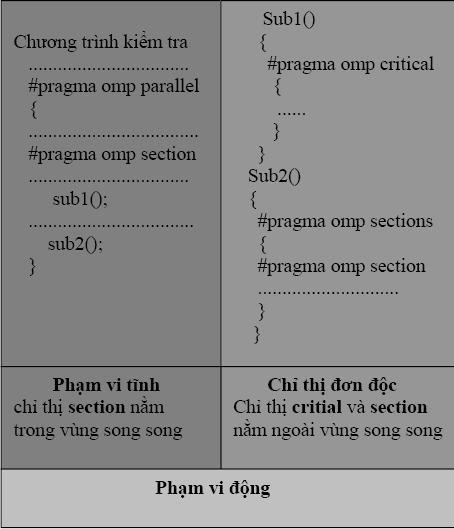
Chỉ thị đơn độc (ophaned directive).

Chỉ thị đơn độc là chỉ thị xuất hiện một cách độc lập so với các chỉ thị khác. Thông thường nó xuất hiện trong các hàm con của chương trình. Chỉ thị đơn độc giúp mở rộng đoạn mã thực hiện song song của chương trình.

Phạm vi động (dymanic extent).

Phạm vi động của chỉ thị bao gồm phạm vi tĩnh của chỉ thị và phạm vi đơn độc của chỉ thị.

Ví dụ:



Hình 2.2 Phạm vi của chỉ thị.

Cấu trúc vùng song song.

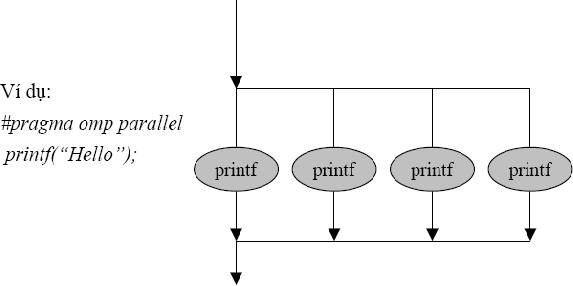
Một vùng song song là một khối mã mà được thực thi bởi nhiều threads.

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.

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( ;. . . . . ; )*

Mệnh đề **SCHEDULE**

Mệnh đề này chỉ ra rằng các công việc lặp đi lặp lại của vòng lặp được thực hiện như thế nào. Có ba kiểu phân chia.

STATIC.

Đối với kiểu phân chia này thì các công việc lặp đi lặp lại của vòng lặp được phân chia một cách tĩnh cho các luồng thực hiện dựa vào biến *chunk\_size*, sau đó sẽ gán cho các luồng thực hiện theo kiểu quay vòng dựa vào chỉ số của các luồng. Nếu biến *chunk\_size* không được chỉ định thì mặc định hệ thống sẽ gán một giá trị là 1.

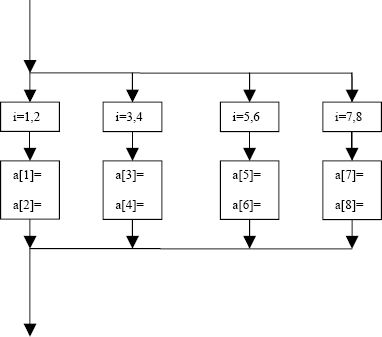
Ví dụ:

*#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:

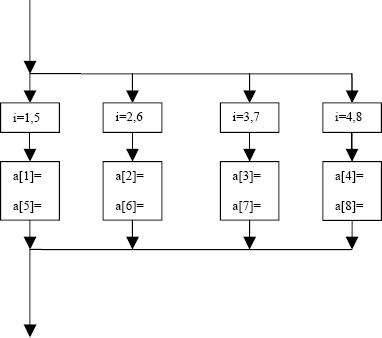
*. . . .*

*#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.

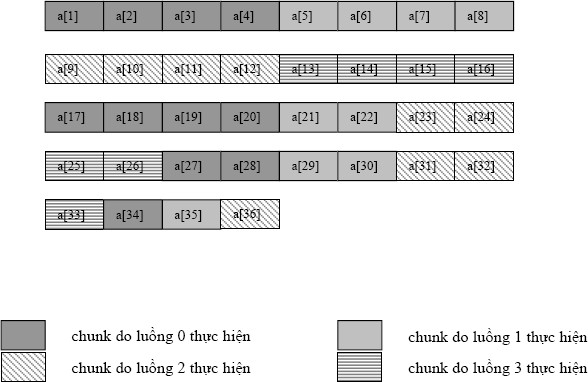
Ví dụ:

*#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 này chỉ xuất hiện khi có chỉ thị ORDERED được bao bọc bởi chỉ thị Do/for.

Mệnh đề **NOWAIT**

Khi xuất hiện mệnh đề này, các luồng thực thi trong đoạn mã song song sẽ không cần phải chờ đợi các luồng khác tại điểm đồng bộ, thực hiện xong công việc của nó mới được thực hiện công việc tiếp theo của mình. Các quá trình thực hiện của các luồng là liên tục hết công việc này đến công việc khác cho tới khi hết mọi công việc được giao trong vùng song song.

Chỉ thị SECTIONS.

Chỉ thị SECTIONS dùng để chia các công việc trong vùng song song cho các luồng thực hiện. Trong cấu trúc của chỉ thị SECTIONS có một hay nhiều chỉ thị SECTION mà mỗi một công việc trong chỉ thị SECTION sẽ được thực hiện bởi một luồng khác nhau. Cấu trúc của chỉ thị SECTION trong C++ cho bởi như sau:

*#pragma omp sections [clause. . . ] newline private (list)*

*firstprivate (list) lastprivate (list) reduction (operator: list) nowait*

*{*

*#pragma omp section newline*

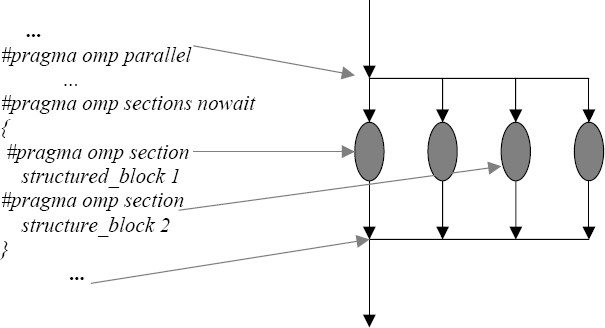
*structured\_block*

*#pragma omp section newline*

*structured\_block*

*}*

Ví dụ:



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.

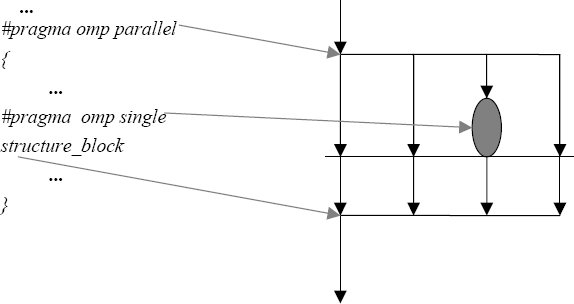
Chỉ thị SINGLE chỉ ra rằng đoạn mã bao quanh chỉ thị SINGLE chỉ được thực hiện bởi một luồng trong tập các luồng trong vùng song song. Cấu trúc của chỉ thị SINGLE được cho bởi như sau:

*#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.

Cấu trúc đồng bộ.

Để nói về cấu trúc này trước tiên ta xét một ví dụ sau. Ví dụ này dùng hai luồng để tăng biến x tại cùng một thời điểm. Biến x lúc đầu mang giá trị 0.

Luồng 1 Luồng 2

increment (x) increment (x)

x = x + 1; x = x + 1; Sự thực thi có thể theo thứ tự sau:

Luồng 1 nạp giá trị của x vào thanh ghi A Luồng 2 nạp giá trị của x vào thanh ghi A Luồng 1 tăng thêm 1 vào thanh ghi A Luồng 2 tăng thêm 1 vào thanh ghi A Luồng 1 lưu thanh ghi A tại vị trí x Luồng 2 lưu thanh ghi A tại vị trí x

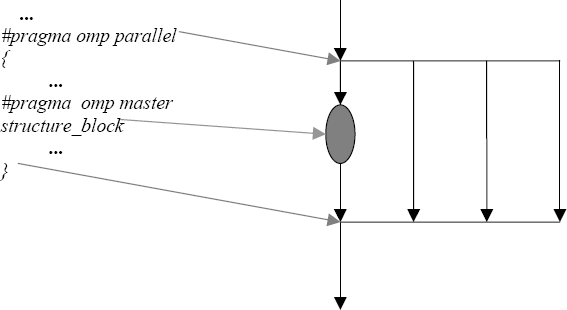
Vậy theo kiểu thực hiện này, sau khi hai luồng thực hiện xong công việc thì kết quả là 1 chứ không phải là 2. Để khắc phục tình trạng này việc tăng biến x phải được đồng bộ giữa hai luồng để đảm bảo kết quả trả về là đúng. OpenMP cung cấp một cấu trúc đồng bộ giúp người lập trình điều khiển sự thực hiện các luồng có liên quan đến nhau như thế nào. Trong cấu trúc đồng bộ có rất nhiều chỉ thị giúp cho việc đồng bộ giữa các luồng.

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:



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*

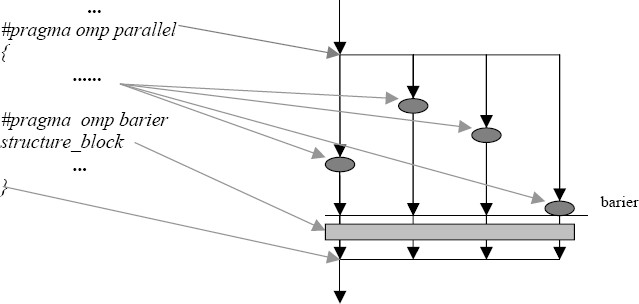
Trong đoạn mã có thể có nhiều chỉ thị CRITICAL. Mỗi chỉ thị CRITICAL khác nhau sẽ có một tên khác nhau để trình biên dịch phân biệt giữa chỉ thị CRITICAL này với chỉ thị CRITICAL khác. Tất cả các chỉ thị CRITICAL không có tên hoặc có tên trùng nhau sẽ được coi như cùng một chỉ thị CRITICAL. Khi một luồng 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.*

Ví dụ:



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.

2.3.6 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

Mệnh đề này dùng để gán giá trị các biến trong chỉ thị THREADPRIVATE cho từng luồng thực thi trong vùng song song. Có nghĩa là

giá trị của biến trong mệnh đề COPYIN của luồng chủ sẽ được dùng làm nguồn. Khi gặp một vùng song song biến nguồn này sẽ được sao cho các luồng thực thi vùng song song đó. Khuôn dạng của mệnh đề được khai báo như sau:

*copyin(list)*

Thƣ 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>**

2.5.1 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)*

Trong đó:

int là một số nguyên, là tổng số luồng cần được tạo để thực hiện vùng song song.

Hàm thư viện này được khai báo trong vùng tuần tự trước vùng mã song song mà vùng mã này có số luồng thực thi cần được tạo lập. Một vấn đề đặt ra là cần phải tạo ra bao nhiêu luồng cho phù hợp với bộ xử lý, số luồng tối đa có thể tạo ra là bao nhiêu. Thông thường nguời lập trình thường tạo số luồng bằng với số bộ xử lý và như vậy mỗi bộ xử lý sẽ thực hiện một luồng khác nhau, tuy nhiên chúng ta có thể tạo số luồng lớn hơn nhiều so với số bộ xử lý nhưng giới hạn không quá 64 luồng.

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 trƣờng (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

3.1

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.