Lời cảm ơn.

Đầu tiên cho em xin phép được gửi lời cảm ơn chân thành và sâu sắc tới thầy, thạc sỹ Ngô Trường Giang, thầy đã tận tình chỉ bảo, hướng dẫn, giúp đỡ em trong suốt quá trình làm tốt nghiệp.

Cho em được gửi lời cảm ơn các thầy cô trong hội đồng phản biện đã chỉ ra cho em những hạn chế, những vấn đề còn thiếu sót của bài báo cáo, qua đó giúp em củng cố, bổ sung để bài báo cáo của mình hoàn thiện hơn.

Cho em được gửi lời cảm ơn tới thầy hiệu trưởng, các thầy cô trong ban lãnh đạo, các thầy cô trong tổ bộ môn Công Nghệ Thông Tin cùng toàn thể quý thầy cô trong trường đã tạo môi trường đào tạo cho em được rèn luyện, học hỏi, tận tình truyền đạt những kiến thức quý báu trong suốt bốn năm học tại trường.

Cuối cùng cho em được gửi tới toàn thể quý thầy cô lời chúc sức khoẻ, thành công và hạnh phúc.

Em xin chân thành cảm ơn quý thầy cô !

Hải Phòng, ngày 26 tháng 06 năm 2009.

SINH VIÊN

Phạm Hải Hưng.

Mục lục

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

(single processor) vì các lý do khác nhau như:

- Tốc độ của bộ xử lý đơn lõi (single processor) đã đến giới hạn.
- Đáp ứng nhu cầu tính toán nhanh của người dùng.
- Giải quyết các bài toán lớn mà với bộ xử lý đơn lõi không đáp ứng được.
- Giảm chi phí đầu tư nhưng vẫn đạt hiệu quả trong tính toán.

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 bài khoá luận em sẽ đi tìm hiểu, áp dụng lập trình, tính toán song song, trên cơ sở sử dụng thư viện OpenMP trong việc giảm thời gian tính toán của bài toán tìm số nguyên tố có số chữ số lớn. Nội dung của bài khoá luận bao gồm:

Chương 1: Tìm hiểu lập trình song song, tính toán song song.

- Chương này giới thiệu một cách tổng quan về lập trình và tính toán song song như sự phân chia cấu trúc tính toán song song, các mô hình lập trình tính toán song song.

Chương 2: Giới thiệu về thư viện mã nguồn mở OpenMP.

- Chương này sẽ đi sâu, nghiên cứu cấu trúc, các thành phần của thư viện mã nguồn mở OpenMP như các chỉ thị biên dịch, các hàm thư viện runtime và các biến môi trường.

Chương 3: Phát biểu, mô tả và cài đặt thực nghiệm bài toán tính giai thừa và bài toán tìm số nguyên tố có số chữ số lớn.

- Chương này sẽ đi vào mô tả, phân tích và cài đặt bài toán tính giai thừa của một số nguyên lớn và bài toán tìm số nguyên tố có số chữ số lớn theo cả hai hướng tuần tự và song song. Từ đó đưa ra sự so sánh, đối chiếu về mặt thời gian của hai bài toán theo hai hướng thực hiện.

Kết luận: Nêu lên những vấn đề đã nghiên cứu và kết quả đạt được, những hạn chế, thiếu sót và phương hướng phát triển trong tương lai.

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

1.1 Định nghĩa:

1.1.1 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.

1.1.2 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 hiện 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.

1.1.3 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.

1.1.4 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.

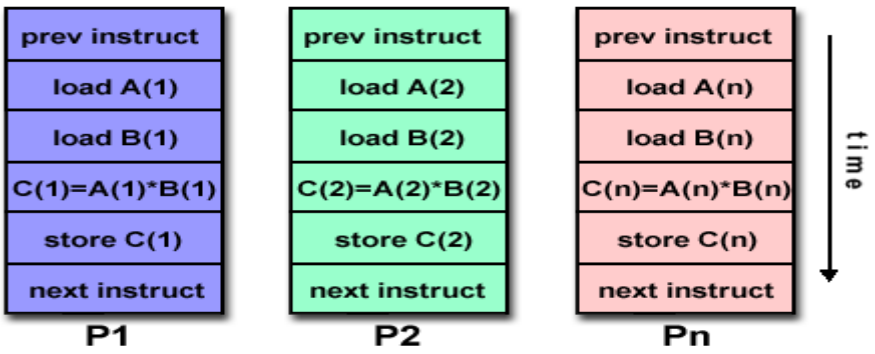
1.2 Sự phân chia cấu trúc tính toán song song.

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).

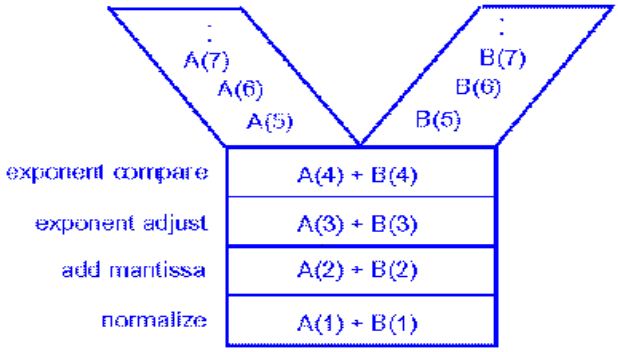
1.2.1.1 Đơn chỉ thị lệnh, đa dữ liệu SIMD (Single Instruction, Multiple Data).



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.

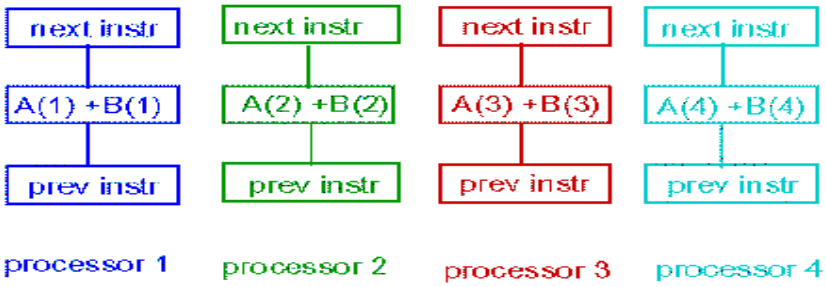
1.2.1.1.1 Vector SIMD



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.

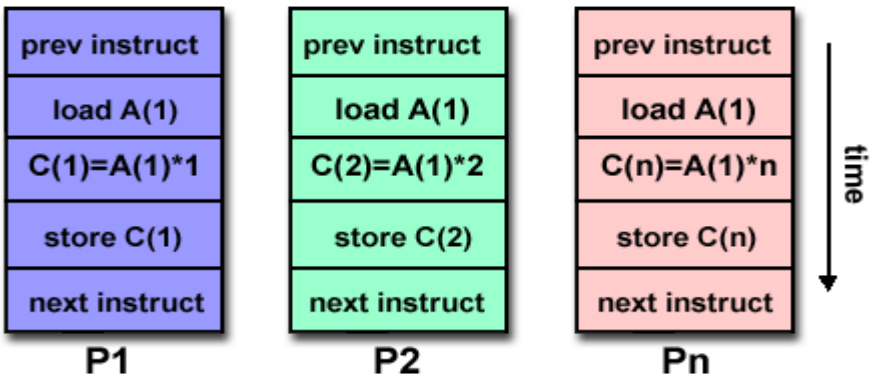
1.2.1.1.2 Parallel SIMD.



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.

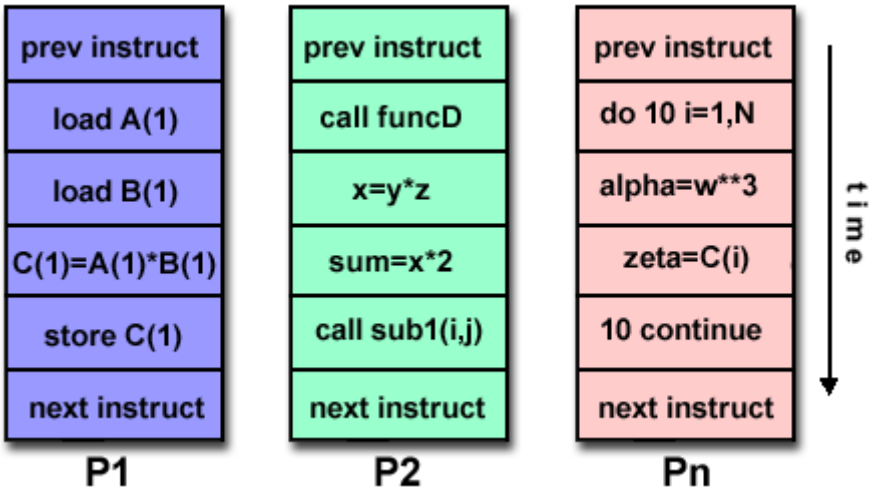
1.2.1.2 Đa chỉ thị lệnh, đơn dữ liệu MISD (Multiple Instruction, Single Data).



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.

1.2.1.3 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.

1.2.2 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 :

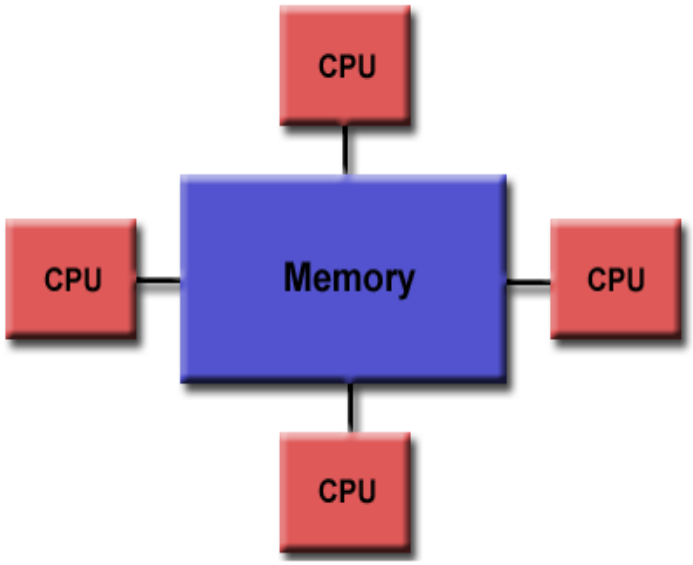
- Dùng chung bộ nhớ (Shared Memory).
- Phân bổ bộ nhớ (Distributed Memory).
- Kết hợp cả hai (Hybrid Distributed – Shared Memory).

1.2.2.1 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).

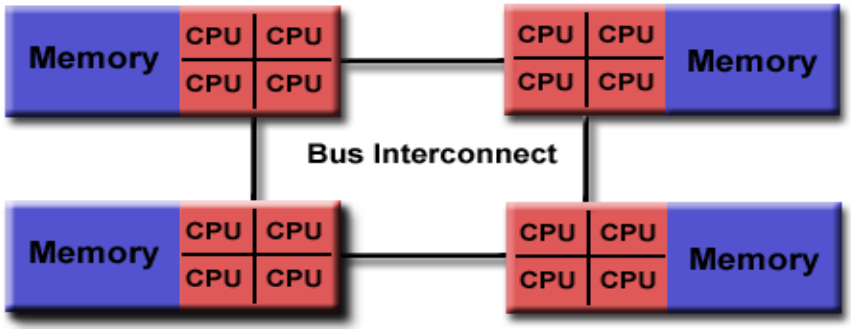
1.2.2.1.1 Truy cập bộ nhớ đồng bộ (Uniform Memory Access).



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.

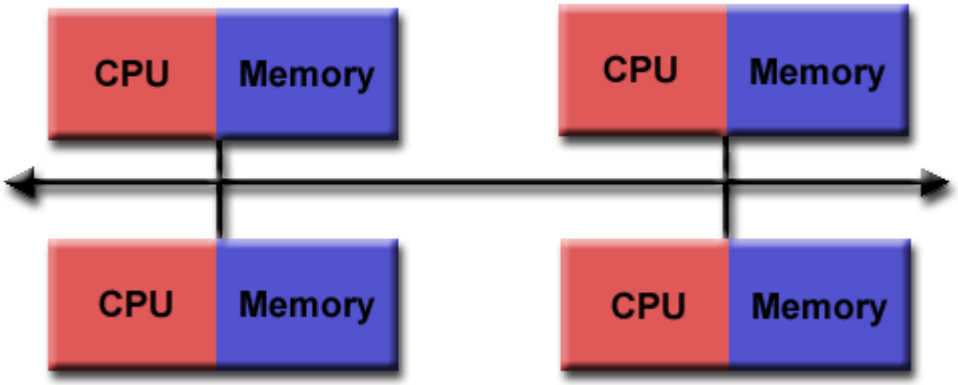
1.2.2.1.2 Truy cập bộ nhớ không đồng bộ (Non Uniform Memory Access)



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).

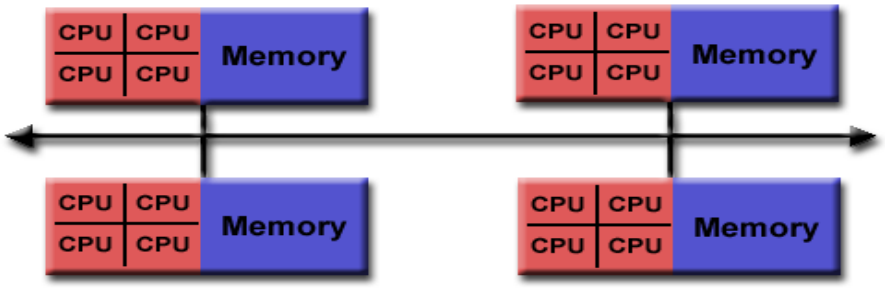
1.2.2.2 Phân bổ bộ nhớ (Distributed Memory).



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)

1.2.2.3 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

1.3 Các mô hình lập trình song song.

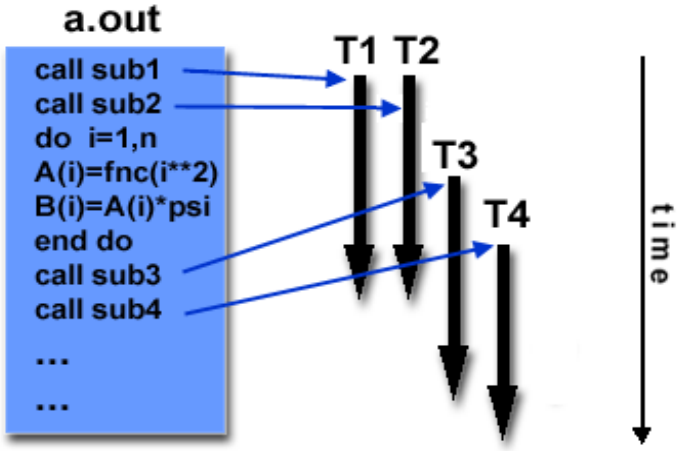
Một số mô hình lập trình song song thường sử dụng là:

- 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).

1.3.1 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.

1.3.2 Mô hình luồng (Thread)

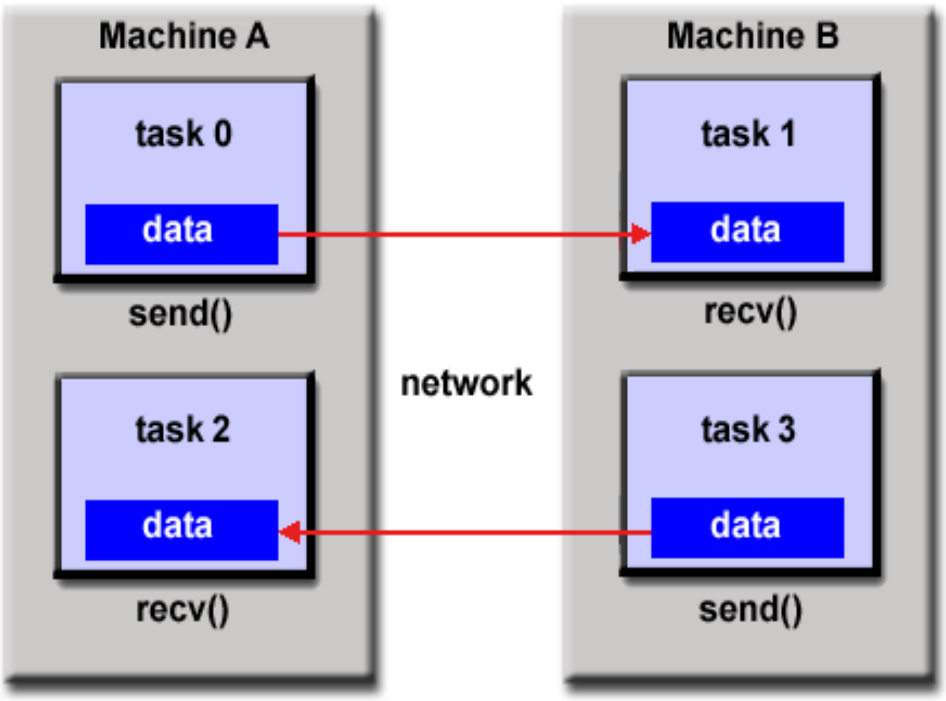


Hình 1.10 Mô hình luồng.

- 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.

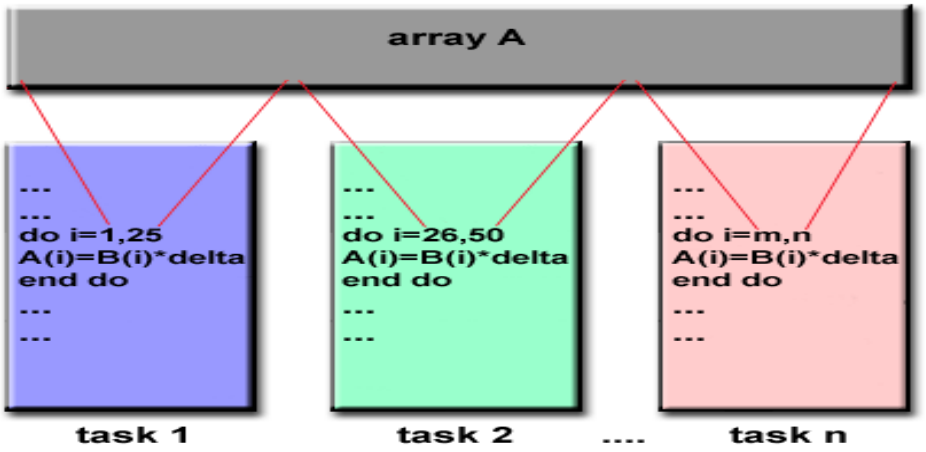
1.3.3 Mô hình truyền thông điệp (Message Passing)



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.

1.3.4 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.

1.4 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.

$$Speedup = \frac{1}{1-P}$$

➢ 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.

$$Speedup = \frac{1}{\frac{P}{N} + S}$$

Trong đó:

- P: là phân số song song.
- N: là số bộ xử lý.
- S: là phân số tuần tự.

1.4.2 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.

1.4.3 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

2.1 Tổng quan về OpenMP.

2.1.1 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++.

2.1.2 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.

2.1.3 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.

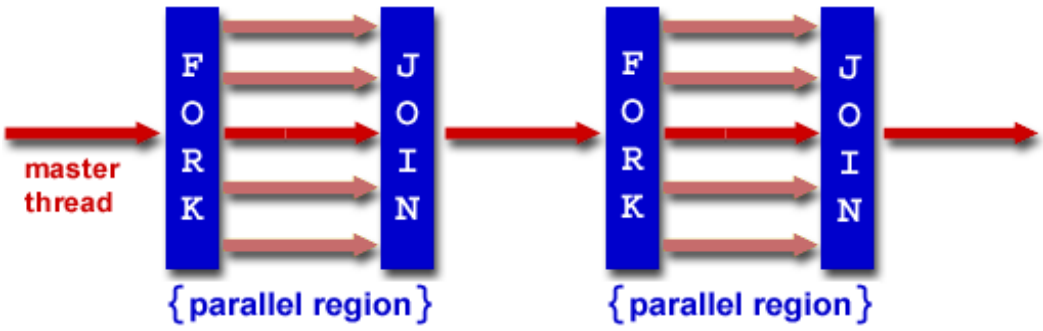
Phiên bản 3. 0 được công bố vào năm 2008 và nó là phiên bản hiện tại được tích hợp thêm nhiều tính năng mới.

2.1.4 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.

2.2 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.

2.3 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.

2.3.1 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.
 . . . .
}
```

2.3.2 Phạm vi của chỉ thị.

Phạm vi tĩnh (static extent).

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

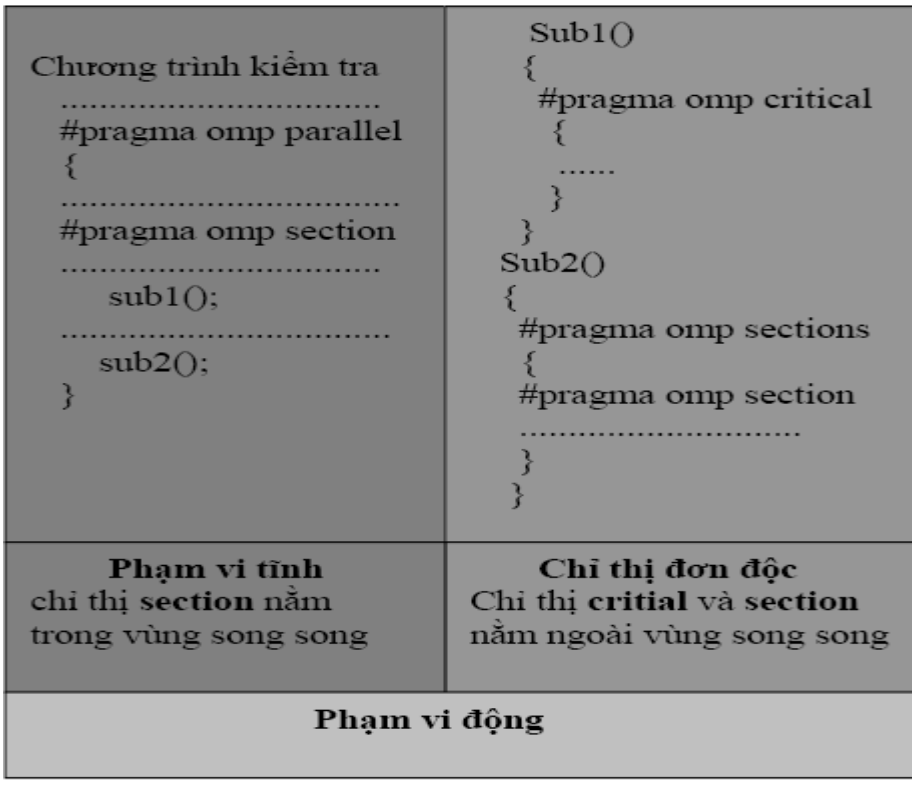
Chỉ thị đơn độc (ophaned directive).

- 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.

2.3.3 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)

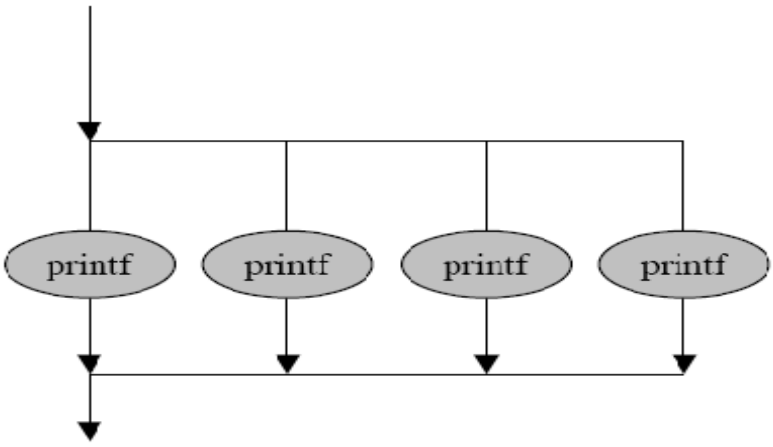
copyin (list)

num_threads (integer-expression)

structured_block

Ví dụ:

```
#pragma omp parallel
 printf("Hello");
```



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.

2.3.3.1 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.

2.3.3.2 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.

2.3.4 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.

2.3.4.1 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:

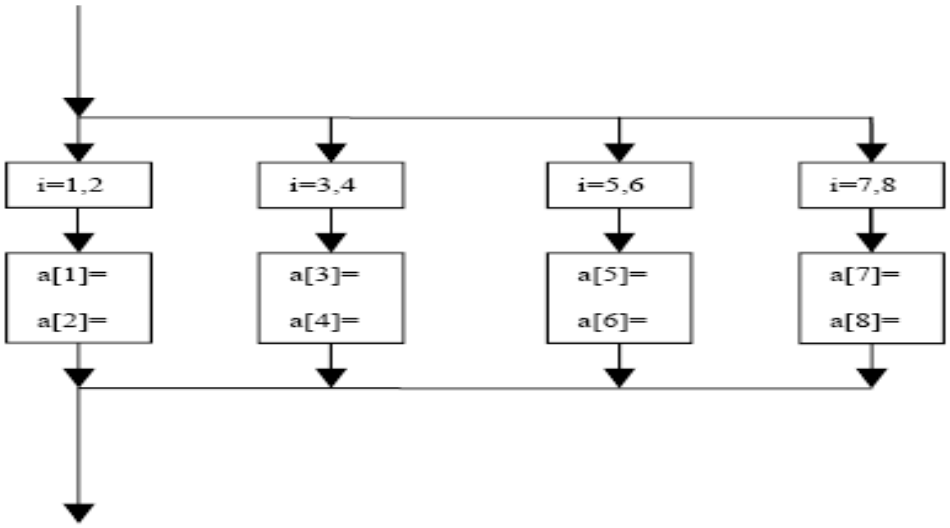
#pragma omp parallel

. . . .

#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.

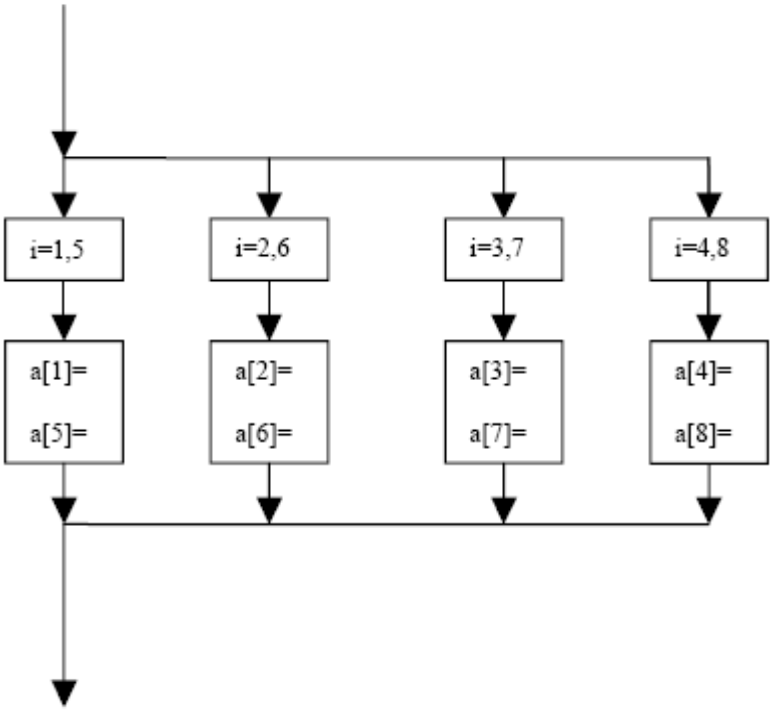
Ví dụ:

. . . .

#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

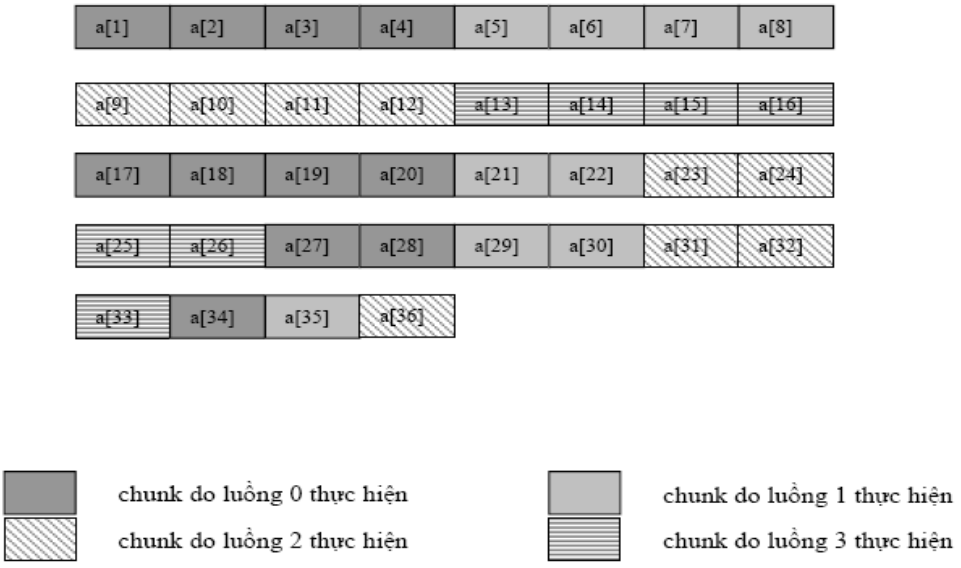
. . . .

#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.

2.3.4.2 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

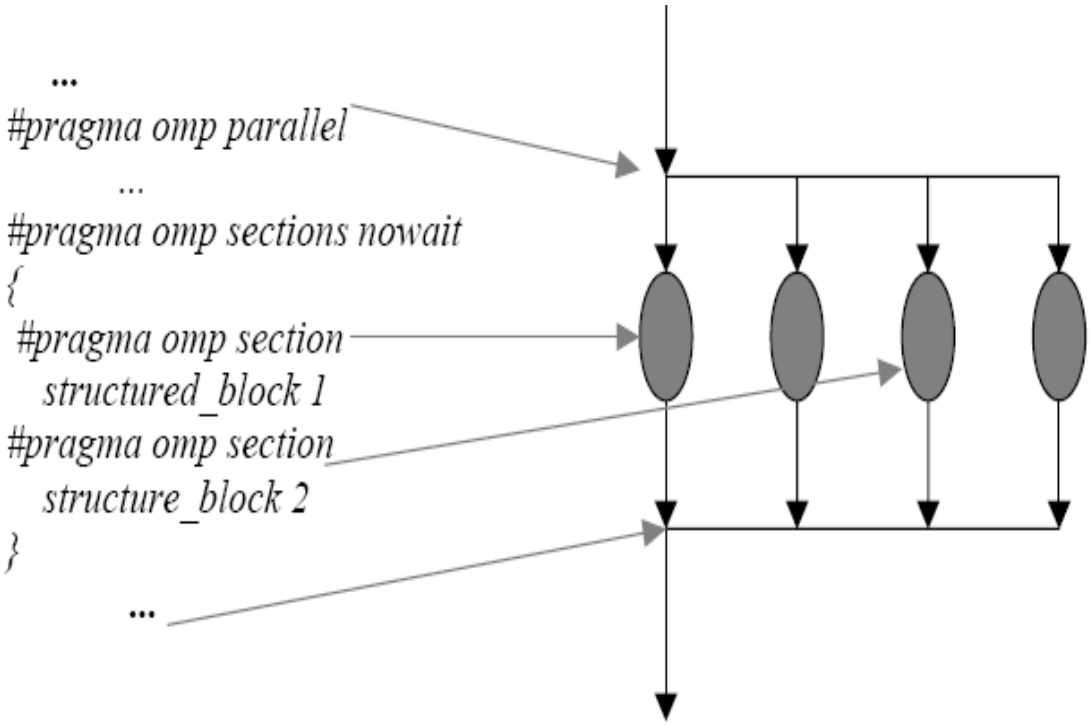
structured_block

#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 hiện theo kiểu quay vòng giống như mệnh đề schedule(static, *chunk_size*).

2.3.4.3 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)

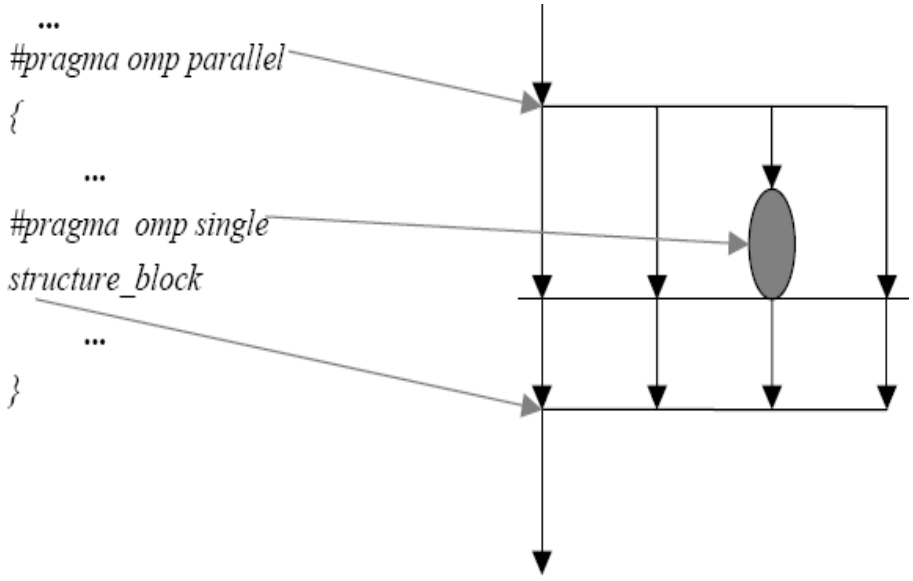
firstprivate(list)

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.

2.3.5 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.

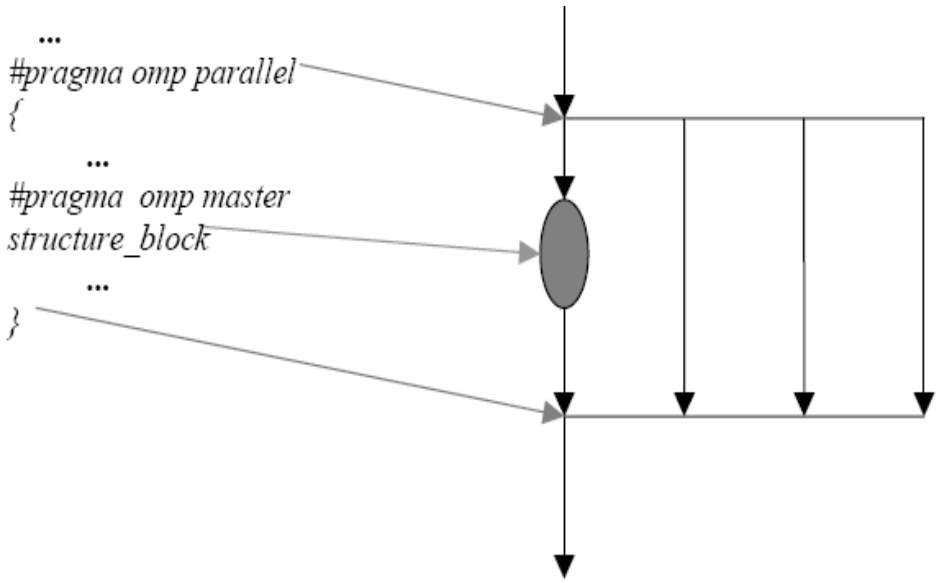
2.3.5.1 Chỉ thị MASTER.

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.

2.3.5.2 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.

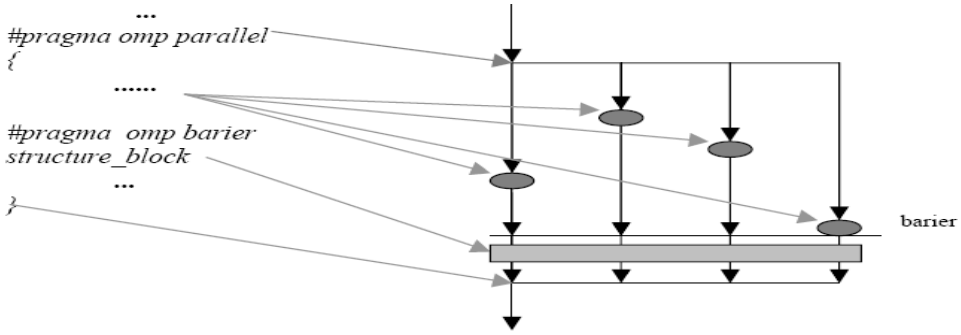
2.3.5.3 Chỉ thị BARRIER

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.

2.3.5.4 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 <=

2.3.5.5 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.

2.3.5.6 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.

2.4 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

2.4.1 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)

2.4.2 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)

2.4.3 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)

2.4.4 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)

2.4.5 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)

2.4.6 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: +, -, *, /, &, ^, |

2.4.7 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)

2.5 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>**

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 người 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.

2.5.2 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.

2.5.3 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)

2.5.4 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()

2.5.5 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.

2.5.6 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.

2.5.7 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.

2.5.8 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.

2.5.9 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.

2.5.10 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.

2.5.11 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)*

2.5.12 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)*

2.5.13 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)*

2.5.14 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)*

2.5.15 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)*

2.6 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:

2.6.1 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’’

2.6.2 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.

2.6.3 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

2.6.4 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: Thực nghiệm

Trên cơ sở tìm hiểu các cấu trúc, các mô hình lập trình tính toán song song, cấu trúc của thư viện OpenMP như các chỉ thị biên dịch, các thư viện RunTime và các biến môi trường. Chương này sẽ áp dụng trong việc giải quyết bài toán tính giai thừa của một số nguyên lớn và bài toán tìm số nguyên tố có n chữ số, sử dụng công cụ là ngôn ngữ lập trình C/C++ trong bộ Visual Studio 2005, có tích hợp sẵn thư viện OpenMP 2. 0.

3.1 Bài toán tính giai thừa của một số nguyên lớn.

Bài toán tính giai thừa của một số nguyên là bài toán hay bắt gặp trong các lĩnh vực khác nhau đặc biệt là trong toán học. Tuy nhiên, trong hầu hết các trường hợp thì bài toán được áp dụng tính toán với số nguyên bé bởi vì thời gian tính toán lớn và tăng lên cùng với giá trị của số nguyên được đem đi tính toán. Với một số nguyên lớn thì thời gian tính toán sẽ rất lâu do vậy việc giảm thời gian tính toán nhưng vẫn đảm bảo đúng kết quả của bài toán trở lên cần thiết. Vận dụng các kiến thức về lập trình và tính toán song song trong việc giảm thời gian tính toán của bài toán, bài toán được phát biểu như sau:

3.1.1 Phát biểu bài toán.

Tính giai thừa của một số nguyên lớn N.

- Với N là bất kỳ và được nhập vào từ bàn phím.

3.1.2 Thuật toán thực hiện.

Có rất nhiều các phương pháp và giải thuật khác nhau để giải quyết bài toán này như sử dụng đệ quy, vòng lặp… Tuy nhiên, không phải giải thuật, phương pháp nào cũng có thể thực hiện song song, hay song song hoá được. Do vậy việc lựa chọn giải thuật để giải quyết bài toán làm sao cho phù hợp, vừa dễ thực hiện tuần tự, vừa có thể thực hiện song song được. Đối với bài toán này, để thuận tiện cho việc song song hoá, chúng ta nên sử dụng vòng lặp.

Khi áp dụng bài toán này cho một số nguyên lớn, kết quả bài toán sẽ rất lớn, do vậy để giải quyết bài toán này, việc đầu tiên chúng ta phải làm là định nghĩa một kiểu dữ liệu mới đủ để lưu trữ kết quả của bài toán.

Sau khi đã có kiểu dữ liệu lớn lưu trữ kết quả bài toán, chúng ta sử dụng vòng lặp để giải quyết bài toán. Các bước thực hiện như sau:

Với số nguyên lớn N nhập vào ta thực hiện như sau:

Bước 1: Gán kết quả bằng 1.

Kq = 1.

Bước 2: Dùng vòng lặp duyệt từ 1 cho đến N.

For i = 1 to N.

Bước 3: với mỗi số i $1 \leq i \leq N$, lấy kết quả nhân với i.

kq = kq * i.

Sau khi vòng lặp kết thúc, kết quả cuối cùng được lưu trữ là kết quả của bài toán.

3.1.3 Song song hoá thuật toán tính giai thừa của một số nguyên lớn.

Song song hoá một thuật toán là phân chia các công việc thực hiện trong thuật toán cho các bộ xử lý thực hiện đồng thời với mục đích là cải thiện tốc độ tính toán cũng như là giảm thời gian tính toán. Việc phân chia các công việc cũng phải làm sao cho phù hợp và đồng đều với các bộ xử lý.

Giả sử hệ thống của chúng ta có bốn bộ xử lý là CPU1, CPU2, CPU3, CPU4, do chúng ta sử dụng vòng lặp để giải quyết bài toán do đó khi song song hoá, chúng ta sẽ chia công việc của vòng lặp ra làm bốn vòng lặp nhỏ và gán các vòng lặp này cho các CPU thực hiện Sau khi các CPU này thực hiện xong công việc, kết quả cuối cùng sẽ được tổng hợp để cho ra kết quả cuối cùng của bài toán. Quá trình song song hoá như sau:

Bước 1: Chia vòng lặp lớn ra bốn vòng lặp nhỏ.

Vòng lặp 1: for i = 1 to N/4.

Vòng lặp 2: for i = N/4 to N/2.

Vòng lặp 3: for i = N/2 to (N/4 + N/2).

Vòng lặp 4: for i = (N/4 + N/2) to N.

Bước 2: Gán bốn vòng lặp này cho bốn CPU thực hiện.

CPU1: for i = 1 to N/4.

kq1 = kq1 * i.

CPU2: for i = N/4 to N/2.

kq2 = kq2 * i.

CPU3: for i = N/2 to (N/4+N/2).

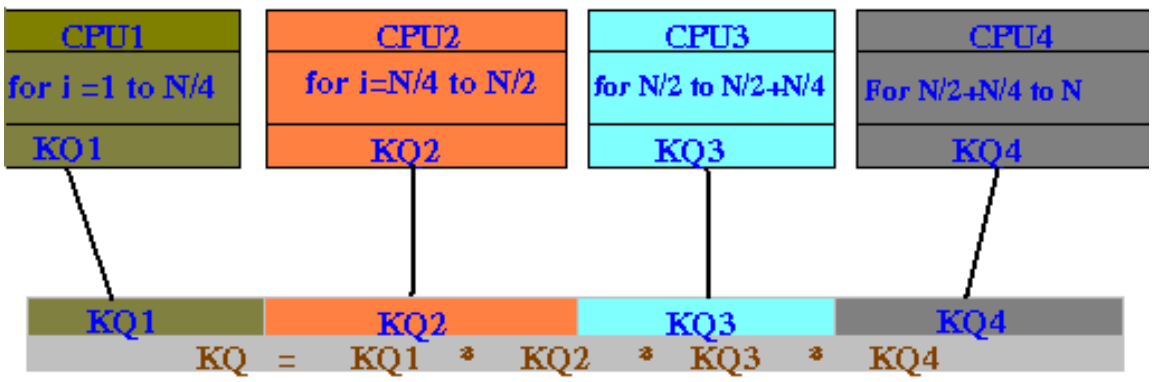
kq3 = kq3 * i.

CPU4: for i = (N/4 + N/2) to N.

kq4 = kq4 * i.

Bước 3: Tổng hợp kết quả bốn CPU để cho ra kết quả cuối cùng.

kq = kq1 * kq2 * kq3 * kq4.



Hình 3.1 Sự phân chia công việc cho bốn bộ xử lý.

3.1.4 Thực hiện song song hoá bằng OpenMP.

Thư viện OpenMP cung cấp các chỉ thị giúp cho người lập trình dễ dàng trong việc song song hoá chương trình của mình. Với giải thuật song song hóa như trên, chúng ta sẽ sử dụng chỉ thị SECTIONS trong OpenMP để chia các công việc cho các bộ xử lý thực hiện. Quá trình song song hóa như sau:

```
# pragma omp section
for i = 1 to N/4.
kq1 = kq1 * i.

# pragma omp section
for i = N/4 to N/2.
kq2 = kq2 * i.

# pragma omp section
for i = N/2 to N/4 + N/2.
kq3 = kq3 * i.

# pragma omp section
for i = N/4 + N/2 to N.
kq1 = kq1 * i.

kq = kq1 * kq2 * kq3 * kq4.
```

Sau khi các CPU thực hiện xong, kết quả của các CPU sẽ được tổng hợp để cho ra kết quả cuối cùng của bài toán.

3.1.5 Kết quả thực nghiệm và nhận xét.

Bài toán được cài đặt bằng ngôn ngữ C/C++, sử dụng chuẩn thư viện OpenMP 2. 0 trong bộ Visual Studio 2005 sau đó được chạy thực nghiệm trên máy tính Pentium R 3. 0GHz, 2 CPU và có hỗ trợ siêu phân luồng Hyper Threading. Sau đây là kết quả thực nghiệm của bài toán.

STT	N Giai thừa	A T/g thực hiện tuần tự (s)	B T/g thực hiện song song (s)	Kích thước N !	So sánh A/B
1	100	4	1	158	4
2	200	19	4	375	4. 75
3	500	180	22	1135	~8. 7
4	1000	1119	124	2568	~9. 1
5	1500	3442	360	4115	~9. 5
6	2000	7249	1638	5736	~4. 9

Bảng 3.1 Kết quả thực nghiệm bài toán tính giai thừa.

Nhận xét:

Qua bảng kết quả thực nghiệm ta thấy, thời gian thực hiện song song giảm đi rất nhiều so với thời gian thực hiện tuần tự, tuy nhiên kết quả so sánh của mỗi lần thực nghiệm đối với một số khác nhau là khác nhau, đó là bởi vì thuật toán của bài toán là đơn giản, dễ thực hiện song song hoá. Do vậy 100% đoạn mã thực hiện tuần tự được thực hiện song song. Theo định luật Amdahl’s thì tốc độ thực hiện tính toán sẽ tăng lên rất nhiều.

3.2 Bài toán tìm số nguyên tố có n chữ số.

3.2.1 Phát biểu bài toán

3.2.1.1 Tổng quan về số nguyên tố.

Định nghĩa số nguyên tố.

- Số nguyên tố là số tự nhiên lớn hơn 1, chỉ có hai ước 1 và chính nó.

Định nghĩa ước và bội.

- Nếu có số tự nhiên A chi hết cho số tự nhiên B thì ta nói A là bội của B, còn B gọi là ước của A.

Do số nguyên tố có tính chất đặc biệt như vậy nên số nguyên tố được áp dụng rộng rãi trong thực tế, đặc biệt là trong các lĩnh vực như mã hoá, an toàn và bảo mật thông tin. Trong lĩnh vực này, hầu hết các hệ mã hoá đều sử dụng số nguyên tố làm khoá để thực hiện các công việc mã hoá. Như vậy tính an toàn, bảo mật của hệ mã càng cao khi số nguyên tố sử dụng trong hệ mã càng lớn. Một bài toán mất rất nhiều thời gian tính toán mà hầu như các hệ mã hoá gặp phải là tìm ra một số nguyên tố lớn có n (n lớn) chữ số. Với các kiến thức đã nghiên cứu về lập trình, tính toán song song, chương này sẽ áp dụng các kiến thức trên trong việc rút ngắn thời gian tính toán trong bài toán tìm số nguyên tố lớn. Bài toán được phát biểu như sau:

Tìm số nguyên tố lớn có n chữ số với n nhập vào từ bàn phím.

3.2.2 Thuật toán thực hiện

Có rất nhiều các thuật toán, phương pháp để tìm số nguyên tố như thuật sàng Eratosthones, thuật toán test Miller Rabin. . . . Tuy nhiên các thuật toán, các phương pháp đều được cài đặt, thực hiện với kiểu dữ liệu cơ bản, nguyên thuỷ của các ngôn ngữ lập trình Ví dụ: int, char, long. . . . (Trong C/C++) tức là người lập trình không phải định nghĩa một kiểu dữ liệu mới mà chỉ việc khai báo và sử dụng. Tuy nhiên các kiểu dữ liệu này thường bé.

Ví dụ:

Kiểu dữ liệu số nguyên lớn nhất trong C/C++ là unsigned long có giá trị 4. 294. 967. 295 (10 chữ số).

-> Số nguyên tố lớn nhất có thể tìm là số nguyên tố có 10 chữ số

Do vậy không đáp ứng yêu cầu bài toán đặt ra. Để giải quyết bài toán này, trước hết chúng ta phải định nghĩa một kiểu dữ liệu số nguyên mới có giá trị lưu trữ lớn hơn 10 chữ số, đa năng hoá các toán tử cho kiểu dữ liệu mới này như:

+, -, *, /, %. . . Sau đó áp dụng giải thuật tìm số nguyên tố cho bài toán này.

Việc lựa chọn giải thuật để giải quyết bài toán sẽ quyết định sự nhanh hay chậm của bài toán và kết quả chính xác của bài toán. Đối với bài toán này, do phải định nghĩa một kiểu dữ liệu mới (lớn) để giải quyết bài toán do đó việc truy cập phần tử có chỉ số lớn (vượt quá phạm vi biểu diễn của C/C++) là khó thực hiện. Vì vậy các thuật toán như thuật sàng Eratosthones. . . là khó thực hiện. Do đó thuật toán thích hợp để giải quyết bài toán này là thuật toán duyệt tuần tự từng phần tử một. Tư tưởng của thuật toán này như sau:

Bước 1: Đối với mỗi phần tử A có n chữ số sẽ làm như sau.

Bước 2: Duyệt từ 2 tới (A -1) để tìm ước của nó.

Bước 3: Kiểm tra nếu A chia hết cho một số B ($2 \leq B \leq (A-1)$).

Kết luận A không phải là số nguyên tố quay lại bước 1 với số tiếp theo sau số A

Nếu A không chia hết cho một số B ($2 \leq B \leq (A-1)$).

Kết luận A là số nguyên tố kết thúc thuật toán.

Nhận xét:

- Nếu A là số có n chữ số thì A lớn nhất là 1. 000. . . (n chữ số 0) -1. A bé nhất là 1. 000. . . (n-1 chữ số 0).
- Ta thấy rằng 2 là số nguyên tố bé nhất và là số chẵn nên tất cả các số nguyên tố lớn hơn 2 đều là số lẻ. Do vậy không cần xét đến các số A có n chữ số là số chẵn.
- Nếu A = B*C thì ước của A là B và C do vậy chỉ cần xét đến B mà không xét đến C. Do đó đối với mỗi số A có n chữ số ta chỉ cần duyệt tìm các ước của nó từ 2 đến A/2.
- Vì mỗi số nguyên dương N đều có ước không quá căn bậc 2 của N do vậy chỉ cần tìm các ước của các số A có n chữ số từ 2 đến căn bậc 2 của A.
- Vậy thuật toán được rút gọn lại thành như sau:
- Giả sử số X = 1. 000. . . (n chữ số 0), số Y = 1. 000. . . (n-1 chữ số 0), số sqrtA là căn bậc hai số học của A.
- **Bước 1**: Duyệt lần lượt từng số X > A ≥ Y (A là số lẻ).
- Với mỗi số A làm như sau:
- **Bước 2**: Duyệt các số B từ 2 đến sqrtA (2 ≤ B ≤ sqrtA).
- **Bước 3**: Kiểm tra.
- Nếu A chia hết cho m ột số B (2 ≤ B ≤ sqrtA).
- Kết luận A không phải là số nguyên tố. Quay lại bước 1 với số sau A là số lẻ (A = A - 2).
- Nếu A không chia hết cho mọi số B (2 ≤ B ≤ sqrtA).
- Kết luận A là số nguyên tố.

Vậy thuật toán sẽ được cài đặt như sau:

- Với số nguyên tố A cần tìm là số có n chữ số (n nhập vào từ bàn phím). Giả sử các số:
- BigInt ctren là số 1. 000. . . (n chữ số 0).
- BigInt cduoi là số 1. 000. . . (n-1 chữ số 0).
- BigInt sqrtA là căn bậc hai số học của A. Vì A lớn nhất là ctren -1 do đó số căn bậc hai lớn nhất sẽ là 1. 000. . . (n/2 chữ số 0).
- Vậy sqrtA = 1. 000. . . (n/2 chữ số 0).
- Chúng ta sử dụng hai vòng lặp for để cài đặt thuật toán.

```
for( A = ctren -1 ; A ≥ cduoi ; A = A - 2) // A lẻ -> A-2 cũng là số lẻ.
{
 for( B = 2 ; B < sqrtA ; B ++ )
 {
  ĐK: nếu A chia hết cho B ;
  break ; // thoát khỏi vòng lặp thứ 2
 }
ĐK: nếu B bằng sqrtA // A không có ước trong khoảng 2 -> sqrtA.
KL: A là số nguyên tố.
break ; // Thoát khỏi vòng lặp thứ 1.
}
```

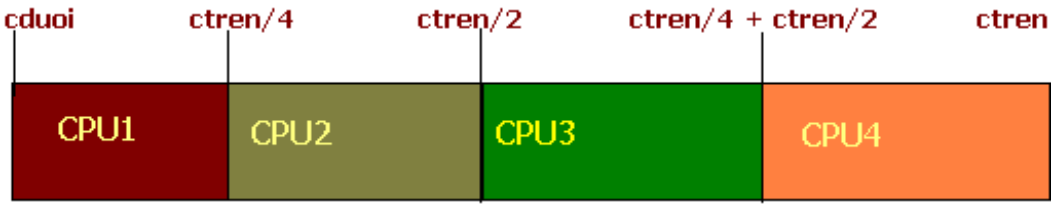
3.2.3 Song Song hoá thuật toán tìm số nguyên tố có n chữ số.

Cũng như thuật toán tuần tự. Sự đúng đắn và tối ưu của thuật toán song song cũng quyết định thời gian tính toán nhanh hay chậm và kết quả tính toán của thuật toán. Ngoài ra chúng ta còn phải chú ý đến các vấn đề xung quanh việc song song hoá thuật toán để tránh xảy ra các trường hợp nghịch lý về song song.

Giả sử chúng ta có 4 bộ xử lý là CPU1, CPU2, CPU3, CPU4 do vậy chúng ta sẽ chia công việc được thực hiện ra làm 4 phần để gán cho các CPU thực hiện và kết quả của các bộ xử lý được tổng hợp để cho ra kết quả cuối cùng.

Nếu chúng ta thực hiện song song hóa vòng for thứ nhất, tức là các công việc của vòng for thứ nhất sẽ được chia đều cho cả 4 bộ xử lý.

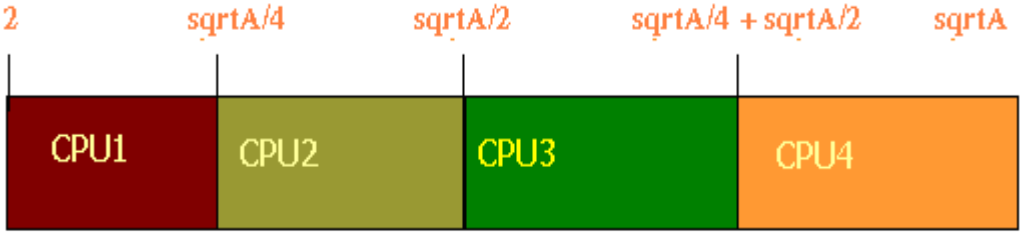
Giả sử các công việc được chia đều như sau:



Hình 3.2 Sự phân chia các công việc vòng for thứ nhất cho 4 bộ xử lý.

Công việc của cả 4 CPU sẽ dừng lại khi bất kỳ một CPU nào trong số 4 CPU tìm ra một số nguyên tố. Như vậy kết quả tìm kiếm của 3 CPU còn lại sẽ bị huỷ. Vậy chúng ta đã chia các công việc đều cho cả 4 CPU cùng thực hiện nhưng kết quả sử dụng cuối cùng lại là kết quả của 1 CPU, nên sự khai thác hoạt động của 3 CPU còn lại là không đúng với mục đích sử dụng -> không đạt hiệu quả trong tính toán song song.

Nếu chúng ta song song hoá vòng for thứ 2 các công việc của vòng for thứ 2 sẽ được phân đều cho 4 CPU. Giả sử các công việc được chia đều như sau:



Hình 3.3 Sự phân chia các công việc vòng for thứ hai cho 4 bộ xử lý.

Khi 1 CPU tìm thấy một ước của số A có n chữ số (tức số A không phải là số nguyên tố), CPU này sẽ thực hiện cập nhật tại địa chỉ bộ nhớ. 3 CPU khác sẽ thấy được thông báo này và sự thực hiện của cả 4 CPU sẽ bị dừng ngay lập tức. Trong trường hợp này, kết quả của 1 CPU được sử dụng còn kết quả 3 CPU khác cũng không được sử dụng. Tuy nhiên miền tìm kiếm của mỗi CPU trong vòng for thứ 2 này là nhỏ do vậy sự lãng phí là không đáng kể. Ngược lại trong trường hợp số A là số nguyên tố, kết quả của cả 4 CPU sẽ được tổng hợp để cho kết quả cuối cùng. Như vậy đã khai thác đúng hoạt động của cả 4 CPU vào đúng mục đích sử dụng.

Để song song hoá vòng for thứ 2 ta sẽ chia đều khoảng giá trị từ 2 – sqrtA thành 4 đoạn nhỏ và gán từng đoạn cho mỗi bộ xử lý. Với mỗi đoạn nhỏ với dd (điểm đầu), dc(điểm cuối), và B là một giá trị thuộc đoạn này thực hiện như sau.

Bước 1: Duyệt các giá trị từ dd đến dc.

Bước 2: Lấy kết quả là thương của phép chia A cho B

Nếu kết quả bằng 0

Gán cờ bằng 1

Gán B bằng dc

Nếu kết quả khác 0, quay lại bước 1 với giá trị B = B+1.

Bước 3: Nếu cờ khác 0 // CPU khác đã tìm ra một ước

Gán B bằng dc.

3.2.4 Thực hiện song song hoá bằng OmpenMP.

Thuật toán sử dụng các chỉ thị SECTION và CRITICAL để thực hiện song song hoá. Chúng ta sử dụng một biến làm cờ báo hiệu cho các CPU khi 1 CPU tìm thấy một ước số của A. Khi một CPU tìm thấy một ước của số A chúng ta sẽ dùng chỉ thị CRITICAL cập nhật lại giá trị của cờ. Quá trình thực hiện như sau:

#pragma omp section //Đoạn mã 1 gán cho CPU 1.

for B = 2 to sqrtA/4.

DK: Nếu kết quả phép chia A/B bằng 0 // *B là 1 ước của A*

#pragma omp critical.

Đặt cờ bằng 1.

Gán B = sqrtA/4 // dùng để thoát vòng lặp.

DK: Nếu cờ khác 0

gán B = sqrtA/4

#pragma omp section //Đoạn mã 2 gán cho CPU 2.

for B = sqrtA/4 to sqrtA/2.

DK: Nếu kết quả phép chia A/B bằng 0 // *B là 1 ước của A*

#pragma omp critical

Đặt cờ bằng 1.

Gán B = sqrtA/2 // dùng để thoát vòng lặp.

DK: Nếu cờ khác 0

gán B = sqrtA/2

#pragma omp section //Đoạn mã 3 gán cho CPU 3.

for B = sqrt4 to (sqrtA/2 +sqrtA/4).

DK: Nếu kết quả phép chia A/B bằng 0 // *B là 1 ước của A*

#pragma omp critical

Đặt cờ bằng 1.

Gán B = sqrtA/2 + sqrtA/4 // *dùng để thoát vòng lặp.*

DK: Nếu cờ khác 0

Gán B = sqrtA/2 + sqrtA/4

#pragma omp section //Đoạn mã 4 gán cho CPU 4.

for (sqrtA/2 + sqrtA/4) to sqrtA

DK: Nếu kết quả phép chia A/B bằng 0 // *B là 1 ước của A*

#pragma omp critical

Đặt cờ bằng 1.

Gán B = sqrtA // dùng để thoát vòng lặp.

DK: Nếu cờ khác 0

Gán B = sqrtA.

3.2.5 Kết quả thực nghiệm và nhận xét

Bài toán sử dụng công cụ là ngôn ngữ C/C++, trong bộ VisualStudio 2005 có tích hợp sẵn thư viện OpenMP chuẩn 2. 0 để cài đặt. Sau đây là kết quả thử nghiệm của bài toán trên máy tính Pentium R 3. 0 GHz, 2 CPU và có hỗ trợ siêu phân luồng Hyper Threading.

STT	N	A T/g thực hiện tuần tự(s)	B T/g thực hiện song song(s)	Kết quả thực hiện	So sánh A/B.
1	6	4	2	9999983	2
2	8	22	13	99999989	~2

3	10	2275	983	9999999967	2. 3
4	11	4440	2460	99999999977	~2
5	12	13735	271954	999999999989	~2

Bảng 3.2 Kết quả thực nghiệm bài toán tìm số nguyên tố có n chữ số.

Nhận xét:

Qua bảng kết quả thực hiện trên cho ta thấy, kết quả về mặt thời gian tính toán của thuật toán thực hiện theo hướng song song giảm đi bằng 1/2 so với thời gian tính toán của thuật toán theo hướng tuần tự. Do thuật toán được song song hoá để phân chia công việc cho 4 bộ xử lý tuy nhiên bài toán được chạy thực nghiệm trên bộ máy tính có 2 bộ xử lý vật lý và hỗ trợ công nghệ siêu phân luồng. Như vậy về mặt logic đảm bảo mỗi bộ xử lý thực hiện các công việc khác nhau nhưng về mặt vật lý thì một bộ xử lý vẫn phải thực hiện đồng thời hai công việc, do đó trong quá trình thực hiện các bộ xử lý vật lý vẫn phải mất thời gian đọc để và ghi dữ liệu lên bộ nhớ khi chuyển đổi giữa các công việc. Hơn nữa trình biên dịch cũng phải mất một thời gian để tạo các threads và huỷ bỏ các threads. Vì vậy thời gian tính toán không thể giảm tới mức tối đa có thể.

Kết luận

Trong khuôn khổ của bài khoá luận, em đã đi nghiên cứu, tìm hiểu về lập trình và tính toán song song, tìm hiểu một công cụ hỗ trợ cho việc lập trình và tính toán song song là thư viện OpenMP. Trên cơ sở những kiến thức đã nghiên cứu, tìm hiểu áp dụng trong bài toán tìm số nguyên tố lớn có n chữ số và bài toán tính giai thừa của số nguyên lớn.

Bài toán được cài đặt theo hai hướng tuần tự và cài đặt để chạy trên máy tính có hai xử lý vật lý và có hỗ trợ công nghệ siêu phân luồng. Tuy bài toán còn bé và chưa có ứng dụng lớn trong thực tiễn nhưng kết quả của bài toán đã nói lên sự khác biệt rõ ràng về thời gian tính toán giữa thuật toán chạy theo hướng tuần tự và thuật toán chạy theo hướng song song góp phần làm cơ sở cho những nghiên cứu về lập trình và tính toán song song sau này.

Tuy nhiên có một vấn đề cần lưu ý khi trong tính toán song song là không phải giải thuật nào cũng có thể song song hoá được. Do vậy trước khi song hóa một giải thuật người lập trình cần chú ý để chọn ra mô hình và phương pháp sao cho bài toán đạt hiệu quả cao đồng thời tránh được nghịch lý trong quá trình song song hoá.

Với sự phát triển của công nghệ thông tin, xu hướng lập trình tính toán song song không chỉ đơn thuần là áp dụng trên một máy đơn có bộ xử lý đa lõi mà còn tiến đến áp dụng đối với nhiều máy tính khác nhau hay mạng máy tính để giải quyết một số bài toán phức tạp trong thực tế.

Trong quá trình nghiên cứu, tìm hiểu, ngoài những kiến thức đã tìm hiểu và nghiên cứu được còn một số những hạn chế, thiếu sót mà em chưa thể nghiên cứu và cập nhật kịp thời. Kính mong các thầy cô chỉ bảo và giúp đỡ để em hoàn thành bài khoá luận này.

Em xin chân thành cảm ơn!

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. llnl. gov/computing/tutorials/parallel_comp
[5]. http://www. openmp. org
[6]. http://www. llnl/computing/tutorials/workshop/openmp/
[7]. http://www hpcc. unical. it/alarico/LNErbacci2. pdf
[8]. http://nereida.deicc.ull.es/html/openmp/minnrsota/tutorial/content_openmp. html
[9]. http://wikipedia. org/wiki/OpenMP.
[10]. Phát triển ứng dụng song song với OpenMP - Trịnh Công Quý - Đại Học Quốc Gia Hà Nội.