ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Trịnh Công Quý

PHÁT TRIỂN ỨNG DỤNG SONG SONG VỚI OPENMP

KHÓA LUẬN TỐT NGHIỆP HỆ ĐẠI HỌC CHÍNH QUY

Ngành: Tin học

HÀ NỘI-2005

ĐẠI HỌC QUỐC GIA HÀ NỘI
TRƯỜNG ĐẠI HỌC CÔNG NGHỆ

Trịnh Công Quý

PHÁT TRIỂN ỨNG DỤNG SONG SONG VỚI OPENMP

KHÓA LUẬN TỐT NGHIỆP HỆ ĐẠI HỌC CHÍNH QUY

Ngành: Tin học

Cán bộ hướng dẫn: TS Nguyễn Hải Châu

HÀ NỘI-2005

Lời cảm ơn

Điều đầu tiên cho tôi gửi lời cảm ơn sâu sắc đến TS. Nguyễn Hải Châu người đã hướng dẫn chỉ bảo tôi trong suốt quá trình thực hiện đề tài. Thầy đã cho tôi những lời khuyên bổ ích, dạy cách viết báo cáo - một kỹ năng không thể thiếu đối với một nhà nghiên cứu. Khóa luận sẽ không hoàn thiện nếu thiếu sự hướng dẫn của thầy từ việc nghiên cứu lý thuyết đến thực nghiệm và hoàn thành khóa luận.

Tôi xin cảm ơn các thầy trong bộ môn *Các Hệ Thống Thông Tin* đã giúp đỡ tôi trang thiết bị máy móc trong quá trình thực nghiệm. Tôi cũng xin cảm ơn đến tập thể lớp K46CC đã có những đóng góp quý báu cho khóa luận cũng như trong quá trình học tập.

Và cuối cùng tôi xin cảm ơn đến gia đình, bạn bè những người luôn quan tâm cổ vũ động viên tôi trong suốt thời gian học tập và làm khóa luận.

Hà nội:tháng 6 năm 2005

Trịnh Công Quý

Tóm tắt nội dung

Ngày nay tính toán song song ra đời với sự thực thi đồng thời của nhiều tài nguyên máy tính giúp giải quyết các bài toán đòi hỏi giới hạn về thời gian xử lý và với dữ liệu lớn như bài toán dự báo thời tiết, bài toán mô phỏng tai nạn giao thông ... Và đã có rất nhiều chuẩn hỗ trợ cho cho việc lập trình song song như MPI (Message Passing Interface) hỗ trợ lập trình song song trên mô hình bộ nhớ phân tán, OpenMP (Open MultiProcesing) hỗ trợ lập trình song song trên mô hình chia sẻ bộ nhớ chung, Pthread hỗ trợ lập trình luồng ...

Trong khuôn khổ của khóa luận văn này chúng tôi đi vào nguyên cứu chi tiết chuẩn OpenMP và ứng dụng của OpenMP vào việc song song hóa bài toán tính lực tương tác giữa các hạt trong hệ mô phỏng N-body.

MỤC LỤC

Bảng các chữ viết tắt

Chữ viết tắt	Tiếng Việt	Tiếng Anh
API	Giao diện lập trình ứng dụng	Application Program Interface
BXL	Bộ xử lý	
MIMD	Đa lệnh đa dữ liệu	Multiple instruction multiple data
MPI	Giao diện truyền thông điệp	Message Passing Interface
OPENMP		Open MultiProcessing
SIMD	Đơn lệnh đa dữ liệu	Single instruction multiple data
SMP	Đa xử lý đối xứng	Symmetric MultiProcesor
UMA	Truy cập bộ nhớ một cách thống nhất	Uniform Access Memory

Mở đầu

Ngày nay sự phát triển của công nghệ được thách thức bởi lớp bài toán lớn cần giải quyết trong nhiều lĩnh vực của đời sống xã hội như dự báo thời tiết, khai phá dữ liệu, xử lý ảnh, mô phỏng tai nạn xe hơi, tự động hóa... Lớp bài toán này vừa đòi hỏi đáp ứng thời gian thực vừa yêu cầu xử lý trên khối dữ liệu lớn. Để giải quyết bài toán này đòi hỏi các bộ xử lý có hiệu năng cao.

Xử lý song song ra đời với mục đích làm tăng khả năng tính toán của máy tính bằng cách kết hợp nhiều bộ xử lý tham gia đồng thời vào quá trình xử lý thay với việc sử dụng các máy tính chuyên biệt đắt tiền.

Với sự phát triển cua kiến trúc máy tính và mạng máy tính cho thấy rằng trong tương lai cho thấy xử lý song song không những được thực hiện trên những siêu máy tính mà có thể được thực hiện trên các trạm làm việc, máy tính cá nhân, mạng máy tính. Nhưng hầu hết các thuật toán ngày nay đều là những thuật toán tuần tự. Cho nên cần xây dựng những thuật toán, cấu trúc dữ liệu cho phép xử lý một cách song song.

Xử lý song song giúp giải quyết hiệu quả rất nhiều bài toán lớn đặc biệt là bài toán mô phỏng N-body. Đó là một bài toán mô phỏng chuyển động của các body trong hệ mô phỏng N-body do lực tương tác giữa giữa các body.Việc song song hóa bài toán trên là rất hợp lý vì một hệ N-body có rất nhiều các body nên việc tính lực tương tác giữa các body tốn rất nhiều thời gian.

Trong khuôn khổ của khóa luận. Áp dụng xử lý song song vào việc giảm thời gian tính lực tương tác giữa các body trong hệ mô phỏng N-body. Luận văn gồm ba chương.

Chương 1: Là chương giới thiệu tổng quan về lập tính toán song song. Chương này đề cập đến các vấn đề như các kiến trúc của máy tính song song, các mô hình lập trình song song, và các vấn đề liên quan đến hiệu năng của lập trình song song như định luật amdahl’s, bế tắc và cân bằng tải.

Chương 2: Là chương giới thiệu về OpenMP. Chương này tập trung nghiên cứu chi tiết các thành phần cuac OpenMP. Bao gồm các chỉ thị biên dịch, các hàm thư viện và các biến môi trường.

Chương 3: Là chương mô tả và cài đặt bài toán N-body. Chương này mô tả sơ qua bài toán N-body. Thuật toán tính lực tương tác lên các body trong hệ, và ba cách song song hóa giai đoạn tính lực tương tác giữa các body.

Kết luận: Nêu lên những vấn đề, kết quả đã đạt được. Chỉ ra sự khác biết giữa các chiến lược song song và hướng phát triển trong tương lai.

Chương 1 Tổng quan về tính toán song song

1.1 Tính toán song song

1.1.1.Tính toán song song là gì

Như chúng ta đã thấy các phần mềm phổ biến ngày nay hầu hết đều được viết trên cơ sở của tính toán tuần tự. Các phần mềm này thường được thực hiện trên một máy tính đơn với duy nhất một bộ xử lý. Vấn đề ở đây được giải quyết thông qua một chuỗi các lệnh tuần tự được thực hiện bởi một bộ xử lý. Tại một thời điểm chỉ có một lệnh được thực hiện.

Tính toán song song ra đời là một sự cải tiến của tính toán tuần tự. Nó là sự giải quyết vấn đề dựa trên sự thực thi đồng thời của nhiều tài nguyên máy tính . Tài nguyên máy tính đây bao gồm:

- ♣ Một máy tính đơn với nhiều bộ xử lý
- ♣ Nhiều máy tính nối lại với nhau thành một mạng máy tính
- ♣ Kết hợp cả hai loại trên

Tính toán song song thường được dùng để giải quyết các vấn đề hết sức phức tạp yêu cầu thời gian tính toán lớn hoặc làm việc với khối dữ liệu lớn như các bài toán dự báo thời tiết, mô phỏng tai nạn xe hơi, xây dựng các mô hình thương mại và các vấn đề khoa học như khai phá dữ liệu , trí tuệ nhân tạo, an toàn dữ liệu…

1.1.2 Tại sao phải tính toán song song

Việc tính toán song song là rất cần thiết. Ngoài hai nguyên nhân chính là nó được dùng để tính toán các bài toán yêu cầu thời gian tính toán lớn và khối lượng dữ liệu lớn còn có các nguyên nhân khác như để sử dụng tài nguyên của các máy khác trong một mạng LAN hoặc thông qua mạng internet, có thể sử dụng nhiều tài nguyên tính toán nhỏ kết hợp lại tạo nên một siêu máy tính. Do giới hạn về không gian lưu trữ của bộ nhớ trên một máy đơn để giải quyết một vấn đề lớn việc sử dụng nhiều bộ nhớ trên nhiều máy tính là rất hữu hiệu trong trường hợp này.

Giới hạn của tính toán tuần tự bao gồm cả hai nguyên nhân thực tế và nguyên nhân vật lý. Để xây dựng nên một máy tính tuần tự tốc độ cao gặp rất nhiều hạn chế

♣ Về tốc độ truyền dữ liệu: Tốc độ truyền của máy tính tuần tự phụ thuộc trực tiếp vào sự di chuyển dữ liệu trong phần cứng. Cho nên việc tăng tốc độ thực hiện phải chủ yếu căn cứ vào các yếu tố tính toán.

♣ Về kích cỡ: Công nghệ chế tạo bộ xử lý cho phép gắn nhiều bóng bán dẫn trên một con chip. Tuy nhiên việc làm này sẽ làm tăng kích thước của bộ xử lý

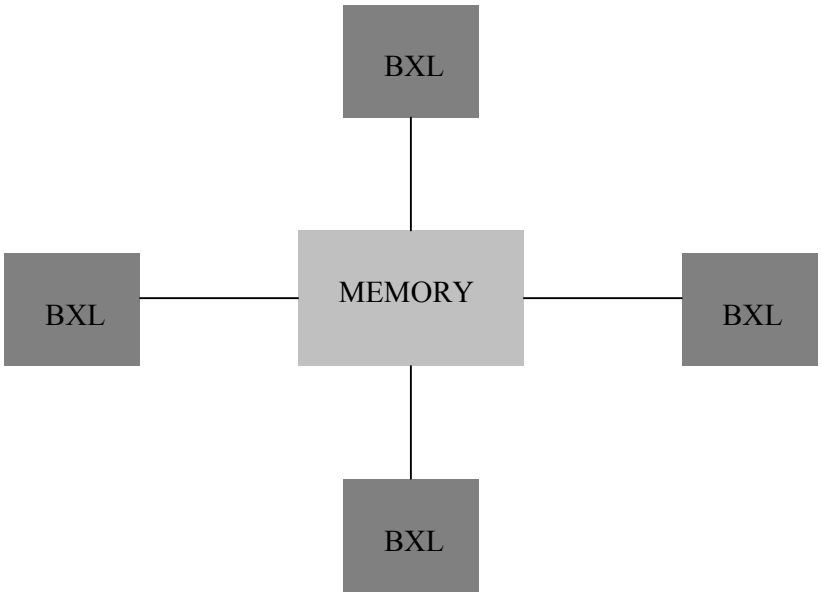
♣ Về thương mại: Việc tạo ra một bộ xử lý tốc độ xử lý cao là rất tốn kém. Sử dụng nhiều bộ xử lý nhỏ đạt hiệu quả tương tự mà lại ít tốn kém hơn

1.2 Phân loại máy tính song song

1.2.1 Phân loại dựa trên sự tương tác giữa các BXL

Một trong những khía cạnh quan trọng của máy tinh song song là cơ chế trao đổi thông tin giữa các BXL.Có ba kiến trúc phổ biến nhất là kiến trúc chia sẻ bộ nhớ chung (shared memory) kiến trúc bộ nhớ phân tán (distributed memory) và kiến trúc bộ nhớ lai(hybrit distributed-shared memory)

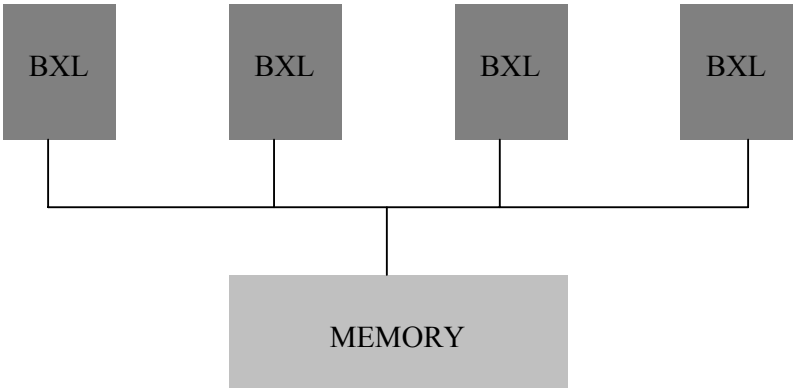
a.Chia sẻ bộ nhớ chung



Hình 1.1: Máy tính song song chia sẻ bộ nhớ chung

Máy tính loại này sử dụng bộ nhớ chia sẻ toàn cục (global shared memory) mà tất cả các BXL đều có thể truy cập đến. Một BXL này có thể trao đổi thông tin với một BXL khác bằng cách ghi vào bộ nhớ toàn cục và BXL thứ hai sẽ đọc dữ liệu tại cùng vị trí đó trong bộ nhớ. Điều này cho phép trao đổi thông tin giữa các BXL.Tuy nhiên dẫn đến một vấn đề là đồng thời có nhiều BXL cùng truy cập tới cùng một vị trí trong bộ nhớ toàn cục. Máy tính loại này có hai loại chính dựa trên thời gian truy cập bộ nhớ

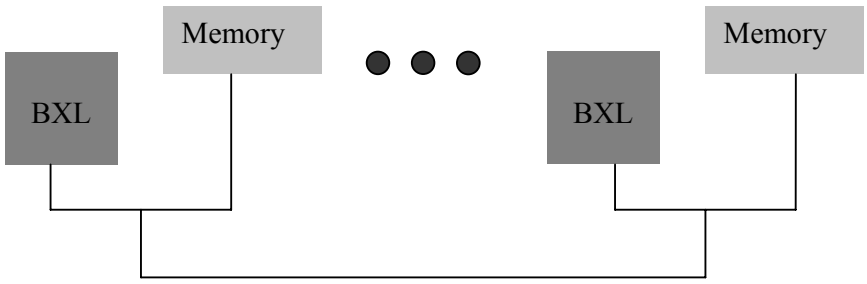
Thứ nhất là máy tính truy cập đồng bộ (UMA). Là loại máy tính với các BXL giống nhau. Tất cả các BXL đều có thể truy cập bộ nhớ đồng thời và thông qua một BUS dùng chung.



Hình 1.2: Máy tính Uniform Access Memory(UMA)

Máy tính loại này có loại gọi là Cache coheren-UMA (CC-UMA). Cache coheren ở đây có nghĩa là khi một BXL cập nhật một vị trí trong bộ nhớ thì tất cả các BXL khác đều nhận biết được sự cập nhật đấy

Thứ hai là máy tính truy cập không đồng bộ (NUMA) .Với máy tính loại này có một đường vật lý nối hai hay nhiều SMP lại với nhau.



Hình 1.3: Máy tính Nun-Uniform Access Memory (NUMA)

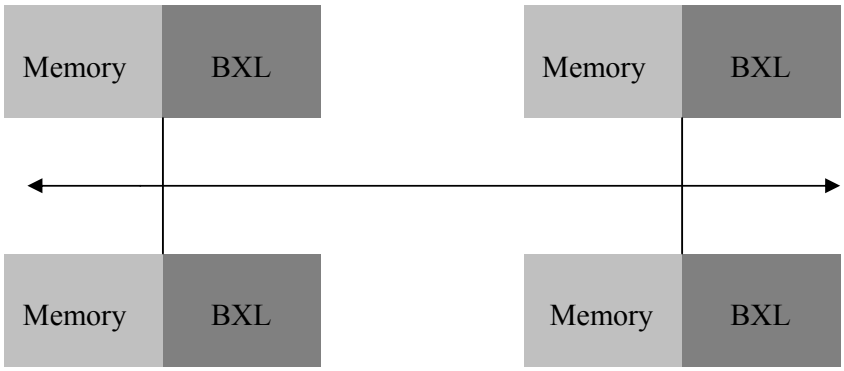
Mỗi một SMP lại có thể truy cập tới bộ nhớ của SMP khác, tuy nhiên với kiến trúc kiểu này thì tất cả các BXL không thể truy cập cùng một lúc tới các bộ nhớ và với việc kết nối các SMP bằng đường vật lý nên thời gian truy cập bộ nhớ chậm

Máy tính chia sẻ bộ nhớ chung có thuận lợi là giúp cho người lập trình thuận tiện khi viết các chương trình song song. Dữ liệu chia sẻ giữa các nhiệm vụ đảm bảo cả hai tiêu chuẩn nhanh và đồng thời. Tuy nhiên máy tính loại này có một số khó khăn là rất khó mở rộng số lượng các BXL vì việc thêm các BXL về phương diện hình học có thể làm tăng các đường kết nối giữa bộ nhớ toàn cục và các BXL. Đối với hệ thống Cache coheren thì làm tăng sự chuyền thông giữa cache và thiết bị quản lý bộ nhớ. Với

máy tính loại này người lập trình phải chịu trách nhiệm đồng bộ chương trình để đảm bảo tính đúng đắn của dữ liệu dùng chung.

b. Bộ nhớ phân tán

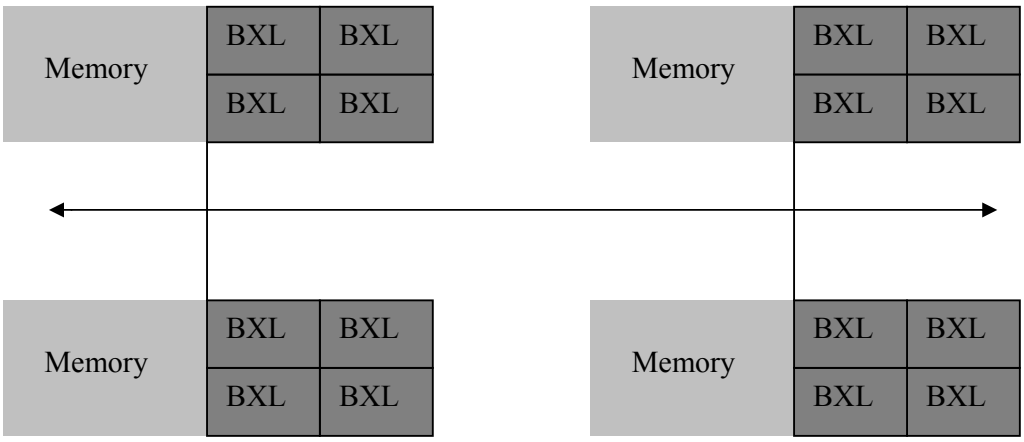
Ngược với máy tính chia sẻ bộ nhớ chung là máy tính với bộ nhớ phân tán trong đó không tồn tại bộ nhớ chia sẻ chung mà mỗi BXL có bộ nhớ cục bộ riêng của chúng. Trong máy tính song song có bộ nhớ phân tán các BXL liên lạc với nhau bằng các thông điệp (message) qua một mạng liên kết (interconnection network) gồm các liên kết truyền thông trực tiếp giữa một số cặp BXL.Một trong những lựa chọn quan trọng trong thiết kế lúc đó sẽ là các cặp BXL nào được nối với nhau.Tốc độ liên lạc là tối ưu khi các BXL được nối trực tiếp với nhau.Tuy nhiên điều này thường là không khả thi do số lượng các liên kết là quá lớn giẫn đến việc tăng giá thành của hệ thống. Cách thứ hai được sử dụng là các bộ sử lý liên lạc thông qua một BUS chia sẻ. Điều này dẫn đến việc độ trễ cao khi số lượng BXL lớn dẫn đến vấn đề tranh chấp BUS



Hình 1.4: Máy tính có bộ nhớ phân tán

c.Máy tính với bộ nhớ lai

Hầu hết các máy tính nhanh và lớn ngày nay đều xây dựng dựa trên sự kết hợp giữa kiến trúc chia sẻ bộ nhớ chung và bộ nhớ phân tán. Sự kết hợp đó tạo nên một máy tính với tên gọi máy tính có bộ nhớ lai



Hình 1.5: Máy tính bộ nhớ lai

Các thành phần chia sẻ bộ nhớ chung trong máy tính bộ nhớ lai thường là các máy CC-SMP. Các BXL trong thành phần chia sẻ bộ nhớ chung có thể truy cập bộ nhớ toàn cục riêng của thành phần đó. Thành phần bộ nhớ phân tán được biết như là một mạng các SMP.Các SMP chỉ có thể truy cập đến bộ nhớ toàn cục trong thành phần chia sẻ bộ nhớ phân tán của chúng chứ không truy cập được bộ nhớ của các thành phần chia sẻ bộ nhớ chung khác. Cái mạng kết nối được xây dựng để chuyển dữ liệu từ SMP này đến SMP khác

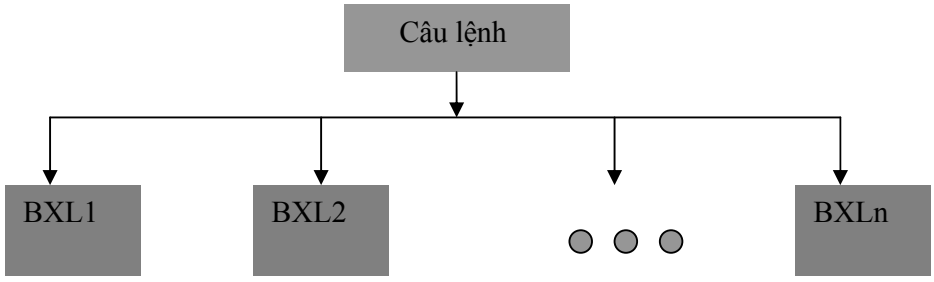
1.2.2 Phân loại dựa trên cơ chế điều khiển chung

Phần lớn các máy tính song song thường có một cơ chế điều khiển chung song vấn đề đặt ra ở đây là các hoạt động của máy tính được điều khiển ở mức độ nào. Xem việc điều khiển theo hai khía cạnh khác nhau. Khía cạnh thứ nhất: Cơ chế điều khiển chung chỉ được sử dụng để nạp chương trình và dữ liệu vào các BXL còn sau đó các BXL hoạt động độc lập. Khía cạnh thứ hai: Cơ chế điều khiển được sử dụng để hướng dẫn các BXL các công việc phải làm tại mỗi bước. Giữa hai khía cạnh này là những cơ chế điều khiển trung gian. Hai loại cơ chế điều khiển phổ biến nhất là.

a.Hệ thống đơn lệnh đa dữ liệu (SIMD)

Các máy tính vector thuộc vào loại này. Mỗi máy tính vector có thể thực hiện một dòng lệnh. Tuy nhiên nó có nhiều BXL số học khác nhau mà mỗi BXL này có khả năng nạp và xử lý dữ liệu riêng của nó. Bởi vậy trong bất kỳ thời điểm nào một thao

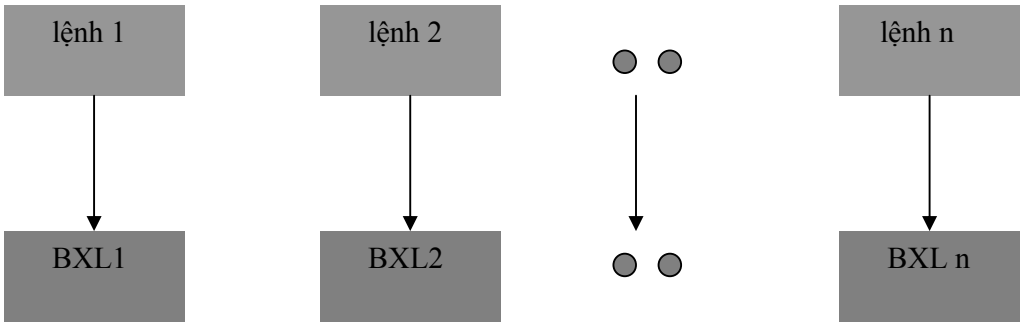
tác luôn ở cùng trạng thái thực thi trên nhiều đơn vị xử lý mà mỗi trong số chúng có thể xử lý dữ liệu riêng rẽ.



Hình 1.6: Hệ thống đơn lệnh đa dữ liệu(SIMD)

b.Hệ thống đa lệnh đa dữ liệu (MIMD)

Phần lớn các máy tính đa xử lý hiện nay đều thuộc vào loại này. Trong các máy tính loại này nhiều dòng lệnh có thể thực hiện cùng một và mỗi dòng lệnh có thể xử lý dữ liệu riêng biệt. Các máy tính loại này ban đầu có rất ít tương tác giữa các BXL. Song hiện nay phần lớn các máy tính đều được thiết kế cho phép tương tác giữa các BXL được thực hiện một cách hiệu quả. Có thể liệt kê một số máy tính loại này như: Symmetry, TC2000, nCUBE2, Paragon XP/S và Connection Machine CM-5.



Hình 1.7: Hệ thống đa lệnh đa dữ liệu(MIMD)

1.3 Các mô hình lập trình song song

1.3.1 Tổng quan về mô hình lập trình song song

Việc đưa ra một mô hình máy tính chung cho việc lập trình giúp cho việc thiết kế giải thuật giải thuật trở nên đơn giản hơn. Lập trình song song đưa thêm những khó khăn mới vào mô hình lập trình tuần tự. Nếu chương trình được thực hiện ở mức thấp nhất thì không những số lệnh thực hiện là rất lớn mà nó còn phải quản lý trực tiếp quá

trình thực hiện song song của hàng nghìn BXL và kết hợp hàng triệu tương tác liên BXL. Bởi vậy khả năng trừu tượng và tính toán module là các đặc tính rất quan trọng trong lập trình song song

Vậy mức độ trừu tượng nào sẽ phù hợp với lập trình song song. Các mô hình này cần cho phép đánh giá cụ thể về khả năng thực hiện đồng thời cũng như tính cục bộ để cho phép phát triển các chương trình có tính modul và có khả năng mở rộng.Và mô hình đó phải phù hợp với kiến trúc của máy tính song song.Các mô hình thông dụng bao gồm

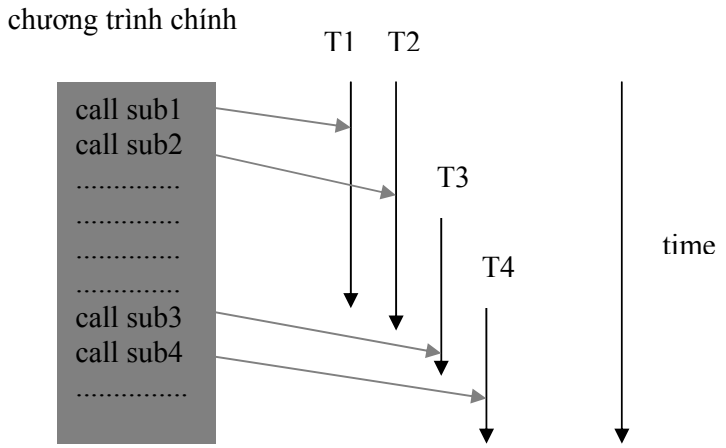
- ♣ Mô hình chia sẻ bộ nhớ chung
- ♣ Mô hình luồng
- ♣ Mô hình truyền thông điệp
- ♣ Mô hình song song dữ liệu
- ♣ Mô hình lai

1.3.2 Mô hình chia sẻ bộ nhớ chung

Trong mô hình chia sẻ bộ nhớ chung các nhiệm vụ cùng chia sẻ một không gian địa chỉ chung có thể được truy cập đọc ghi theo phương thức không đồng bộ.Các cơ chế khác nhau như khóa (locks) và semaphore được điều khiển để truy cập đến bộ nhớ toàn cục.Xét theo quan điểm của lập trình viên thì ưu điểm của mô hình này là không có khái niệm sở hữu dữ liệu. Nghĩa là không phải chỉ định rõ ràng quá trình truyền dữ liệu giữa nhiệm vụ gửi và nhiệm vụ nhận dữ liệu. Tính chất này giúp cho phát triển các chương trình đơn giản hơn. Tuy nhiên khi đó việc hiểu và đảm bảo tính cục bộ trở nên khó khăn và cũng được chú ý nhiều nhất trong kiến trúc chia sẻ bộ nhớ chung. Việc viết các chương trình xác định cũng trở nên khó khăn

1.3.3. Mô hình luồng

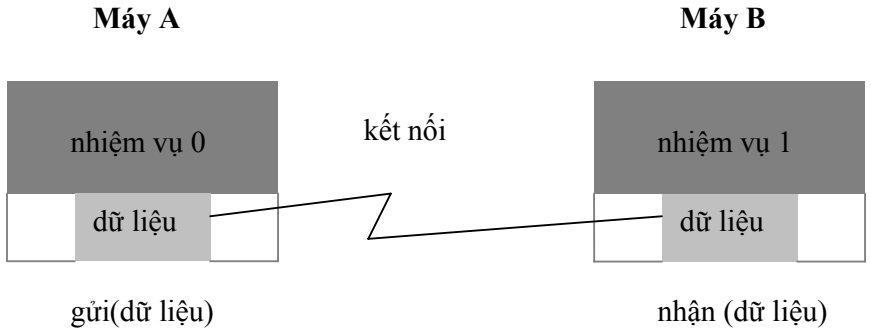
Trong mô hình luồng chương trình chính được chia thành các nhiệm vụ. Mỗi nhiệm vụ được thực hiện bởi các luồng một cách đồng thời. Mỗi một luồng có dữ liệu riêng của nó và chia sẻ dữ liệu toàn cục của chương trình chính. Các nhiệm vụ đưa cho mỗi luồng là các thủ tục con của chương trình chính. Và bất kì luồng nào cũng có thể thực hiện bất kì thủ tục con nào tại cùng thời điểm với các luồng khác.Trong mô hình luồng các luồng kết nối với nhau thông qua bộ nhớ toàn cục với việc kết nối này thì chương trình phải được xây dựng một cách đồng bộ để tránh cùng một lúc có nhiều luồng cùng cập nhập một vị trí trong bộ nhớ toàn cục



Hình 1.8: Mô hình luồng

1.3.4 Mô hình truyền thông điệp

Trong mô hình truyền thông điệp chương trình song song được chia thành các nhiệm vụ. Mỗi nhiệm vụ sử dụng bộ nhớ cục bộ của nó. Các nhiệm vị này có thể được cư trú trên các máy vật lý giống nhau kết nối với nhau qua mạng với số lượng tùy ý

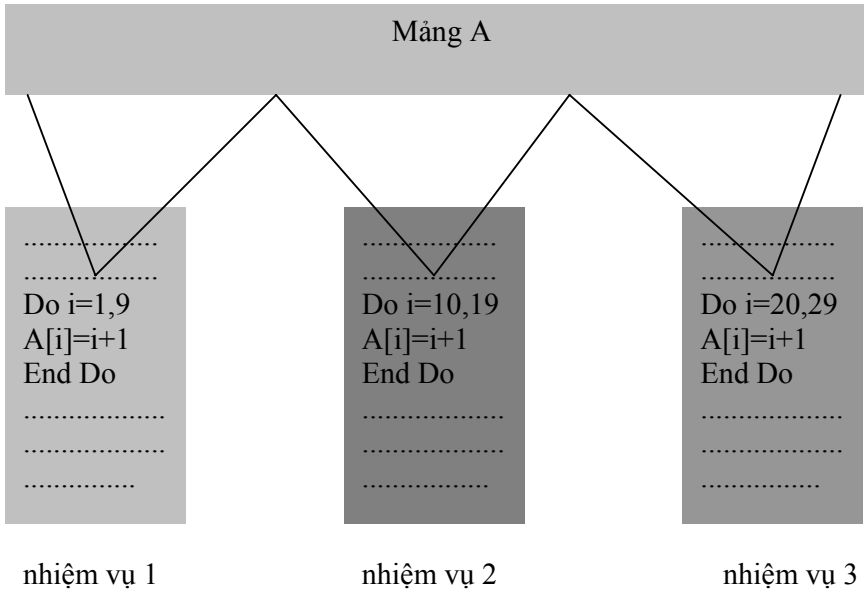


Hình 1.9: Mô hình truyền thông điệp

Các nhiệm vụ trao đổi dữ liệu với nhau qua hai phương thức gửi và nhận thông điệp. Xét trên khía cạnh lập trình thì các thông điệp chứa trong một thư viện thông điệp.Thư viện này phải được gắn vào mã nguồn của chương trình song song.MPI là một thư viện ngày nay được dùng rất phổ biến

1.3.5. Mô hình song song dữ liệu

Mô hình lập trình song song dữ liệu giúp lập trình các chương trình song song được thực hiện trên một tập dữ liệu lớn. Tập dữ liệu ở đây thường được xắp xếp theo một cấu trúc nhất định như là mảng hoặc theo khối



Hình 1.10: Mô hình lập trình song song dữ liệu

Với mô hình này thì các nhiệm vụ của chương trình làm việc với cùng một cấu trúc dữ liệu. Tuy nhiên mỗi nhiệm vụ sẽ làm việc trên từng phân vùng khác nhau của dữ liệu và các nhiệm vụ phải thực hiện các thao tác giống nhau.

Trong kiến trúc chia sẻ bộ nhớ chung thì tất cả các nhiệm vụ truy cập vào cấu trục dữ liệu thông qua bộ nhớ toàn cục. Còn đối với kiến trúc bộ nhớ phân tán thì dữ liệu được chia ra và lưu trữ trên các bộ nhớ cục bộ của các BXL

1.3.6. Mô hình lai

Mô hình lai là sự kết hợp của hai hay nhiều mô hình lập trình song song kết hợp lại với nhau

Hiện nay thì mô hình lai phổ biến nhất là mô hình kết hợp giữa mô hình truyền thông điệp với mô hình luồng hoặc với mô hình chia sẻ bộ nhớ chung. Một mô hình lai khác nữa là sự kết hợp giữa mô hình song song dữ liệu với mô hình truyền thông điệp. Mô hình dạng này rất thuận tiện vì mô hình song song dữ liệu trên kiến trúc bộ nhớ

phân tán sử dụng message passing để trao đổi dữ liệu giữa các nhiệm vụ một cách trong suốt đối với lập trình viên song song

1.4 Hiệu năng của tính toán song song

Trong phần này chúng ta sẽ trình bày một số vấn đề liên quan đến hiệu năng của tính toán song song bao gồm: khả năng tăng tốc độ tính toán,cân bằng tải (Load balancing) và sự bế tắc (Deadlock)

1.4.1 Định luật Amdahl's

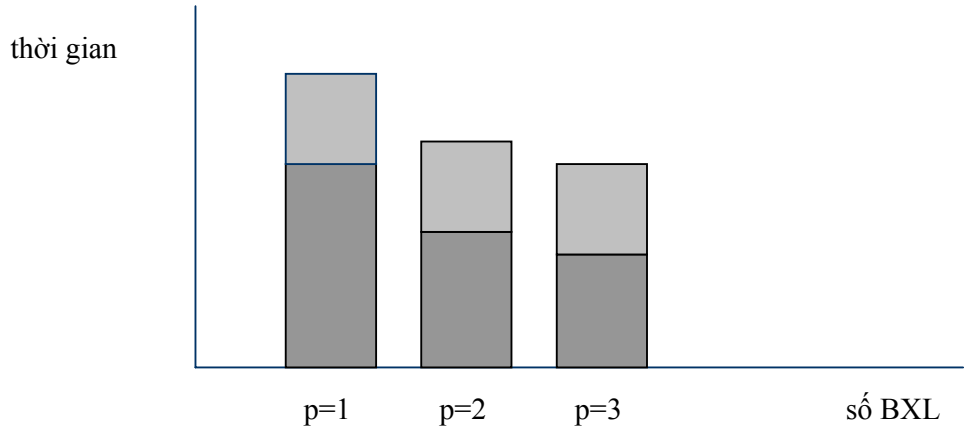
Trong nhiều ứng dụng thực tế đòi hỏi thời gian thực,vấn đề cần giải quyết có kính thước cố định,do đó khối lượng công việc phải làm cũng thường xác định được trước. Định luật do Amdahl phát biểu năm 1967 nhằm đánh giá hiệu năng của việc tính toán cho các bài toán thuộc loại này.

Khi tăng số lượng BXL trong máy tính song song, khối lượng công việc được được phân phối cho nhiều BXL thực hiện. Mục tiêu chính là tìm được kết quả của bài toán nhanh nhất có thể hay nói một cách khác là giảm đến mức tối đa thời gian tính toán.

Định luật Amdahl: Gọi f là phần nhỏ của thao tác tính toán trong quá trình tính toán phải thực hiện một cách tuần tự, $0 \leq f \leq 1$. Tốc độ tối đa S có thể đạt được bằng cách sử dụng máy tính song song với p BXL được cho bởi công thức

$$S \leq \frac{1}{f + (1- f)p}$$

Thời gian cho phần việc xử lý song song của ứng dụng sẽ dảm dần đến 0 khi ta tăng số lượng BXL. Thời gian cho việc xử lý tuần tự luôn là hằng số



Hình 1.11: Sự phụ thuộc thời gian vào số lượng BXL của đinh luật Amlahl

1.4.2 Cân bằng tải

Ta giả sử rằng nếu dữ liệu được phân tán trên các bộ nhớ địa phương của các BXL. Khi đó khối lượng công việc của các BXL cần phải được phân phối hợp lý trong suất quá trình tính toán. Trong nhiều trường hợp , giả sử này là đúng tuy nhiên trên thực tế điều này không phải lúc nào cũng thực hiện được. Giải pháp được đưa ra ở đây là cân bằng tải động nhằm mục đích làm thay đổi sự phân phối khối lượng công việc giữa các BXL trong quá trình thực hiện tính toán

Thông thường sau khi phân phối khối lượng công việc cho mỗi BXL, quá trình cân bằng tải động thực hiện theo bốn bước cơ bạn dưới đây Giám sát hiệu năng của mỗi BXL, trao đổi thông tin trạng thái giữa các BXL, tính toán và ra quết định phân phối lại khối lượng công việc và cuối cùng là thực hiện việc chuyển đổi dữ liệu thật sự

Để thực hiện điều này có rất nhiều thuật toán được thực hiện để cân bằng tải động được đề xuất. Theo kết quả Znstietal phân lớp các thuật toán này theo chiến lược tập trung, phân tán hòa toàn (Fully distributed) và phân tán một nửa(Semi – distributed)

a.Các thuật toán cân bằng tải tập trung

Nhằm đưa ra quyết định có tính chất tổng thể trong việc phân phối lại khối lượng công việc cần thực hiện cho các BXL. Một vài thuật toán trong lớp này sử dụng thông tin hệ thống có tính chất toàn cục để lưu trạng thái của các máy riêng biệt trong hệ thống. Thông tin này sẽ cho phép thuật toán phân phối công việc cho các BXL một cách dễ dàng. Tuy nhiên khối lượng công việc tăng theo tỉ lệ thuận với số lượng các

BXL, do đó đòi hỏi khối lượng lớn bộ nhớ trên một BXL để lưu trữ thông tin trạng thái. Vì vậy các thuật toán thuộc lớp này không được áp dụng một cách rông rãi .

b.Các thuật toán cân bằng tải phân tán hoàn toàn

Trong chiến lược này, mỗi BXL có một bản sao về thông tin trạng thái của hệ thống . Các BXL 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 BXL chỉ có thông tin trạng thái cục bộ nên việc cân bằng tải không tốt bằng các thuật toán cân bằng tải tập trung

c.Các thuật toán cân bằng tải phân tán một nửa

Các thuật toán thuộc lớp này chia các BXL thành từng miền. Trong mỗi miền sử dụng thuật toán cân bằng tải tập trung để phân phối công việc cho các BXL thuộc miền đó.

d. Sự bế tắc(Deadlock)

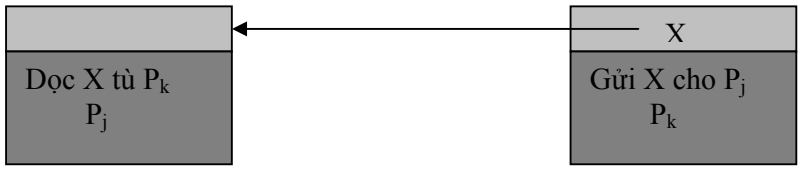
Các tiến trình bị rơi vào tình trạng bế tắc nếu mỗi tiến trình đó nắm giữ tài nguyên mà một vài tiến trình khác yêu cầu sử dụng nó để xử lý.

Lý do tồn tại sự bế tắc là do nhiều tiến trình cùng sử dụng tài nguyên chung mà không có sự kiểm soát tốt. Sự bế tắc tồn tại trong các hệ điều hành đa nhiệm, cũng như các hệ thông đa BXL và đa máy tính

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

Xem xét hệ thống đa máy tính với các BXL xử lý không đồng bộ . BXL P_i gửi thông điệp cho BXL P_j không kết nối cho tới khi có thao tác thông điệp đó. Mặt khác BXL P_i gửi thông điệp cho BXL P_j nội dung của thông điệp được lưu trong vùng đệm của hệ thống cho đến khi BXL P_j nhận và đọc thông điệp. Giả sử rằng trong cùng một thời điểm có nhiều BXL cùng gửi thông điệp đến BXL P_j và điều này sẽ làm cho vùng đệm bị đầy. Việc gửi thông điệp tiếp theo chỉ được thực hiện khi BXL P_j đọc một hay nhiều thông điệp

Giả sử BXL P_k là một trong những BXL có khả năng gửi thông điệp đến BXL P_j. Nếu BXL P_j cố gắng đọc thông điệp do BXL P_k gửi đến nó sẽ bị kết khối cho đến khi nội dung thông điệp có trong vùng đệms. Rõ ràng BXL P_k bị kết khối cho tới khi BXL P_j loại bỏ một hay nhiều thông điệp từ vùng đệm như vậy BXL P_j và P_k rơi vào bế tắc.



Hình 1.12: P_k kết khối để gửi X cho P_j vì vùng đệm P_j bị đầy nên P_j không thể nhận được X . P_k và P_j rơi vào bế tắc

Bốn điều kiện gây nên bế tắc

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

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

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

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

Một số cách khắc phục sự bế tắc

Cách thứ nhất ta sử dụng là cố gắng dò tìm sự bế tắc khi chúng sẩy ra và khôi phục lại. Một cách khác để tránh sự bế tắc thông qua sử dụng các thông tin yêu cầu tài nguyên của các tiến trình để điều khiển sự phân phối để khi tiếp tục phân phối các tài nguyên không là nguyên nhân để các tiến trình rơi vào bế tắc. Cách thứ ba là ngăn cấm không để xảy ra đồng thời ba điều kiện cuối trong bốn điều kiện này sinh bế tắc

Chương 2: Lập trình song song với OpenMP

2.1. Giới thiệu về OpenMP

2.1.1. Khái niệm cơ bản về OpenMP

OpenMP là một giao diện lập trình ứng dụng (API) được sử dụng để điều khiển các luồng trên cấu trúc chia sẻ bộ nhớ chung. Thành phần của OpenMP bao gồm :

1. Các chỉ thị biên dịch (Compiler Directives)

2. Các thư viện runtime (Runtime Library Routines)

3. Các biến môi trường (Emviroment Variables) .

Các chỉ thị biên dịch, các thư viện runtime và các biến môi trường này được sử dụng để lập trình song song với hai ngôn ngữ Fortran và C/C++. OpenMP là một chuẩn bộ nhớ chia sẻ hỗ trợ bởi nhiều nền phần cứng và phần mềm như là DEC, Intel, IBM, SGI, Numerical Algorithms Group. Hơn thế nữa OpenMP còn rất khả chuyển và có thể thực thi trên cả môi trường UNIX và Windows NT

2.1.2. Lịch sử của OpenMP

Ngay từ trước thập kỷ 90. Các nhà cung cấp các máy tính chia sẻ bộ nhớ đã đưa ra các sản phẩm hỗ trợ sự đồng bộ và các chỉ thị cơ bản. Để lập trình các chương trình song song trên kiến trúc dạng này thì ngôn ngữ Fortran được sử dụng với rất nhiều tiện dụng. Người sử dụng có thể làm giảm thời gian thực hiện các chương trình Fortran bằng cách thực hiện các vòng lặp theo cách song song. Trong trường hợp này trình biên dịch phải chịu trách nhiệm song song hóa một cách tự động các vòng lặp thông qua các BXL SMP. Tuy nhiên mỗi một nhà cung cấp lại sử dụng những phương thức và sự thực thi khác nhau phụ thuộc vào các nền tảng phần cứng và kiến trúc riêng của họ

Để đưa ra một chuẩn hỗ trợ việc lập trình song song trên kiến trúc chia sẻ bộ nhớ thì năm 1994 chuẩn ANSI X3H5 ra đời. Nhưng nó không tồn tại được lâu vì trong thời gian này các máy tính bộ nhớ phân tán trở nên rất phổ biến. Chuẩn OpenMP được bắt đưa ra vào mùa xuân năm 1997 để thay thế chuẩn ANSI X3H5. Trong thời gian này thì các máy tính chia sẻ bộ nhớ rất thịnh hành.

Bên cạnh đó Pthread cũng được đưa ra nhưng Pthread không có tính mở rộng, không có các chỉ thị biên dịch. Pthread không hỗ trợ song song tốt, người lập trình rất khó thực thiện việc song song hóa nhờ vào Pthread. Với Pthread người lập trình phải

quan tâm nhiều đến các chi tiết ở mức thấp. Và OpenMP được thiết kế để giảm bới những nhược điểm của Pthread.

2.1.3. Mục đích và ứng dụng của OpenMP

OpenMP ra đời với mục tiêu cung cấp một chuẩn chung cho rất nhiều kiến trúc và nền tảng phần cứng. Nó thiết lập một tập các chỉ thị biên dịch hỗ trợ việc lập trình song song trên máy tính chia sẻ bộ nhớ chung. Một mức song song chính thường được thực thi với ba đến bốn chỉ thị. OpenMP ra đời giúp cho việc lập trình song song một cách dễ dàng nó cung cấp khả năng song song hóa chương trình tuần tự mà không dùng đến thư viện thông điệp v.v...

Có thể sử dụng OpenMP để giải quết các vấn đề giới hạn về thời gian như bài toán dự báo thời tiết, và để mô phỏng các vấn đề thực tế như bài toán mô phỏng tai nạn xe hơi, giải quyết các bài toán khoa học yêu cầu khối lượng tính toán lớn như bài toán mô phỏng N-Body, dự báo thời tiết …

2.2. Mô hình lập trình song song OpenMP

2.2.1. Song song hóa dựa trên cơ chế luồng (Thread based parallelism)

Trong mô hình trên chương trình xử lý trên bộ nhớ toàn cục bao gồm nhiều luồng thực thi đồng thời. OpenMP dựa vào sự tồn tại của nhiều luồng trên một mô hình lập trình chia sẻ bộ nhớ chung.

2.2.2. Mô hình song song hiện (Explicit Parallelism)

Mô hình trên là một mô hình lập trình không tự động. Người lập trình có quyền điều khiển việc song song hóa một cách độc lập

2.2.3. Mô hình Fork-Join

Trong các mô hình trên thì OpenMP sử dụng mô hình Fork-Join để thực thi công việc song song

Hình 2.1 Mô hình Fork-Join

Trong mô hình này tất cả các chương trình song song đều bắt đầu với việc xử lý đơn bởi một luồng chủ (master thread). Luồng chủ này sẽ thực thi một cách tuần tự cho tới khi bắt gặp vùng song song (parallel region) đầu tiên .

FORK: Có nghĩa là luồng chủ sau đó sẽ tạo ra một tập các luồng song song. Và sau đó đoạn mã trong vùng song song được thực thi song song bởi tập luồng song song vừa tạo ra

JOIN: Khi mà tập luồng song song đã hoàn thành đoạn mã trong vùng song song chúng sẽ được đồng bộ và kết thúc rồi sau đó công việc lại được thực hiện bởi luồng chủ

2.3. Các chỉ thị trong OpenMP

2.3.1. Khuôn dạng chỉ thị trong OpenMP

Chỉ thị trong OpenMP được cho dưới dạng sau

 # pragma omp directive-name [clause...] newline

• *# pragma omp*: Yêu cầu bắt buộc đối với mọi chỉ thị OpenMP C/C++

• *directive-name*: Là tên của chỉ thị phải xuất hiện sau ***#pragma omp*** và đứng trước bất kì mệnh đề nào

• *[clause...]*: Các mệnh đề này không bắt buộc trong chỉ thị

• *newlin* : Yêu cầu bắt buộc với mỗi chỉ thị nó là tập mã lệnh nằm trong khối cấu trúc được bao bọc bởi chỉ thị

Ví dụ:

#pragma omp parallel default (shared) private (beta,pi)

2.3.2. Phạm vi của chỉ thị

a. Phạm vi tĩnh (Static Extent)

Đó là những đoạn mã nguyên bản trong phạm vi từ đầu đến cuối khối cấu trúc cho sau mỗi chỉ thị. Phạm vi tĩnh của chỉ thị không mở rộng đến các thủ tục và các tệp chứa mã.

b. Chỉ thị đơn độc (Orphaned Directive)

Chỉ thị đơn độc là chỉ thị xuất hiện độc lập với chỉ thị khác. Nó tồn tại ở ngoài phạm vi tĩnh của chỉ thị khác. Chỉ thị đơn độc mở rộng với các thử tục và các tệp mã nguồn

c. Phạm vi động (Dynamic Extent)

Phạm vi động của chỉ thị bao gồm phạm vi tĩnh của của chỉ thị và phạm vi của các chỉ thị mồ côi

Ví dụ:

<table>
<tr>
<td>

```
Chương trình kiểm tra
  .....
  #pragma omp parallel
  {
  .....
  #pragma omp section
  .....
     sub1();
  .....
    sub2();
  }
```

</td>
<td>

```
  Sub1()
  {
   #pragma omp critical
    {
     .....
    }
  }
 Sub2()
 {
  #pragma omp sections
  {
  #pragma omp section
  .....
  }
  }
```

</td>
</tr>
<tr>
<td>Phạm vi tĩnh
chỉ thị section nằm trong vùng song song</td>
<td>Chỉ thị đơn độc
Chỉ thị critial và section nằm ngoài vùng song song</td>
</tr>
<tr>
<td colspan="2">Phạm vi động</td>
</tr>
</table>

2.3.3. Cấu trúc vùng song song

Một vùng song song là một khối mã nguồn được thực thi bởi nhiều luồng . Trong C/C++ một vùng song song có định dạng như sau:

#pragma omp parallel [clause...] newline

if (scalar_expression)

private (list)

shared (list)

default (shared | none)

firstprivate (list)

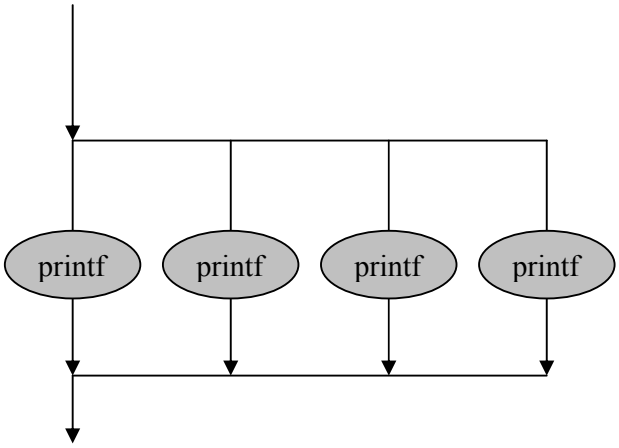
reduction (operator : list)

copyin (list)

structured_block

Ví dụ:

#pragma omp parallel
printf("Hello");



Hình 2.2: Sự thực thi đồng thời của các luồng trong cấu trúc vùng song song

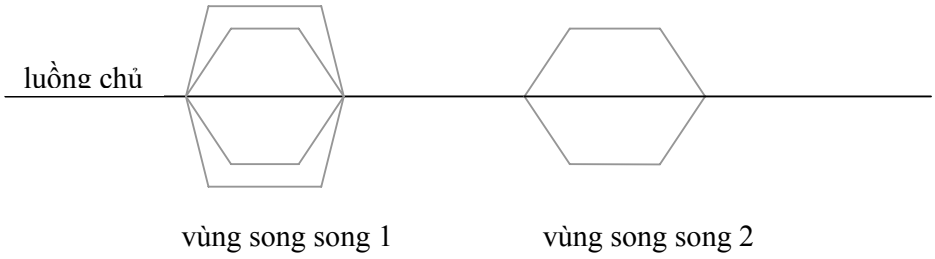
Khi mà một luồng gặp chỉ thị PARALLEL thì nó sẽ tạo ra một tập các luồng và luồng ban đầu sẽ là luồng chủ của tập các luồng đó. Luồng chủ ở đây cũng là một thành viên trong tập các luồng đó và là luồng số 0

Để bắt đầu thực hiện một vùng song song thì đoạn mã nguồn trong vùng song song được sao ra những bản giống nhau đưa cho mỗi luồng thực hiện một cách song song. Đợi cho đến khi tất cả các luồng đều thực hiện song công việc của mình thì luồng chủ sẽ thực hiện công việc tuần tự còn lại ngoài vùng song song đó. Vậy câu hỏi đặt ra ở đây là có bao nhiêu luồng để thực hiện đoạn mã song song trong vùng song

song. Để biết được điều này người ta dùng hàm thư viện OMP_NUM_THREAD(), và để biết được số thứ tự của mỗi luồng ta dùng hàm OMP_GET_THREAD_NUM() ...Lưu ý số thứ tự của các luồng nằm trong khoảng từ 0 đến số thứ tự của luồng chủ trừ đi 1. Cũng từ khái niệm vùng song song xuất hiện khái niệm vùng song song lồng và khái niệm luồng động

Vùng song song lồng (Nested Parallel Region): Có nghĩa là trong một vùng song song con xuất hiên các vùng song song nhỏ khác

Luồng động (Dynamic Thread). Theo mặc định thì khi một chương trình được chia ra thành nhiều vùng song song thì các vùng song song đó sẽ được thực hiện bởi các luồng với số lượng bằng nhau. Điều này có thể thay đổi bằng cách cho phép hệ thống gán động số lượng các luồng thực hiện cho mỗi vùng song song . Chúng ta có hai cách thức để gán động các luồng thứ nhất là dùng hàm thư viện omp_set_dynamic() và thứ hai là dùng biến môi trường OMP_DYNAMIC.



Hình 2.3: Cấu trúc phân chia luồng động

2.3.4. Cấu trúc chia sẻ công việc

Cấu trúc chia sẻ công việc dùng để chia việc thực hiện công việc trong vùng song song cho các luồng trong tập các luồng thực hiện công việc cho bởi vùng song song. Cấu trúc chia sẻ công việc phải được bao bọc bởi một vùng song song để có thể thực hiện song song và cấu trúc này có thể được thực hiện bởi tất cả các luồng trong tập các luồng hoặc chỉ một số luồng trong tập các luồng thực thi vùng song song. Có ba loại 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 (interations) cho bởi vòng lặp phải được các luồng thực hiện một cách song song. Chỉ thị for trong C/C++ được cho dưới dạng sau

#pragma omp for [*clause*...] *newline*

schedule (*type* [,*chunk_size*])

ordered

private (*list*)

firstprivate (*list*)

lastprivate (*list*)

shared (*list*)

reduction (*operator : list*)

nowait

for_loop

Mệnh đề SCHEDULE

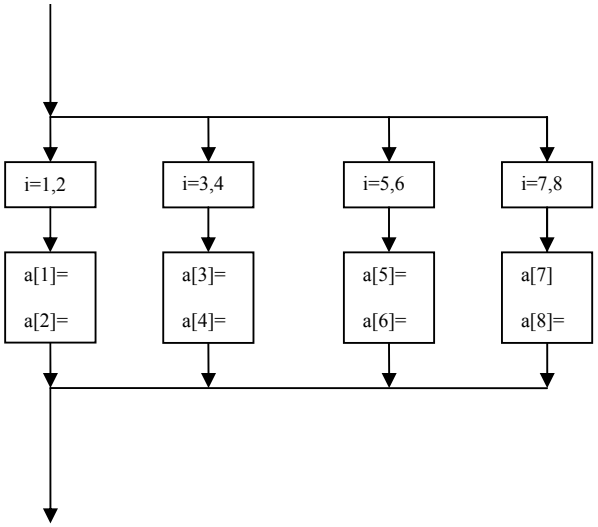
Mệnh đề này chỉ ra rằng các công việc lặp đi lặp lại (interations) của vòng lặp được phân chia cho các luồng thực hiện như thế nào. Có ba kiểu phân chia

STATIC

Đối với kiểu phân chia này thi các công việc của vòng lặp đi lặp lại của vòng lặp được phân chia dựa theo giá trị của biến *chunk_size* thành các chunk công việc liên tiếp (mỗi chunk công việc ở đây bao gồm chunk_size các công việc lặp đi lặp lại) và gán tĩnh chunk công việc này cho các luồng thực hiện theo kiểu quay vòng dựa trên thứ tự của số hiệu mỗi luồng. Nếu biến chunk không được chỉ định thì các công việc này sẽ được phân chia lần lượt cho các luồng.

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 bốn luồng thực thi tính a[1],a[2],...,a[8]

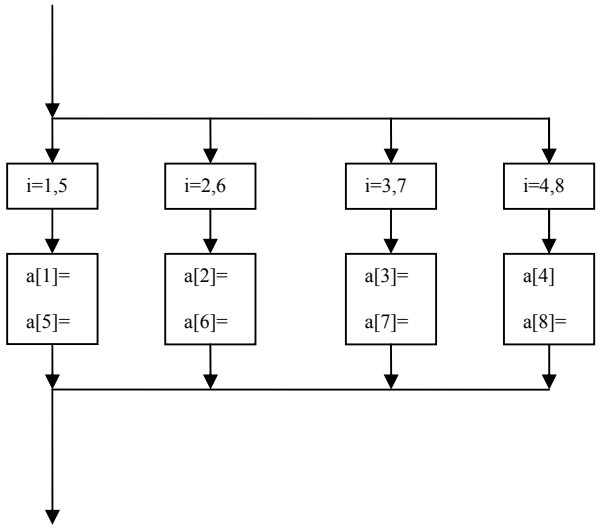
DYNAMIC

Đối với việc phân chia động thì các công việc lặp đi lặp lại của vòng lặp được phân chia thành một chuỗi các chunk. Mỗi chunk ở đây là một tập chunk_size công việc. Các chunk này sẽ được gán động cho mỗi luồng. Các luồng sau khi kết thúc một chunk công việc sẽ đợi để nhận chunk công việc cho đến khi không còn chunk công việc nào được gán. Lưu ý rằng chunk công việc cuối cùng có thể có số lượng công

việc nhỏ hơn *chunk_size*. Nếu biến *chunk_size* không được chỉ ra thì giá trị mặc định của nó là một

Ví dụ

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



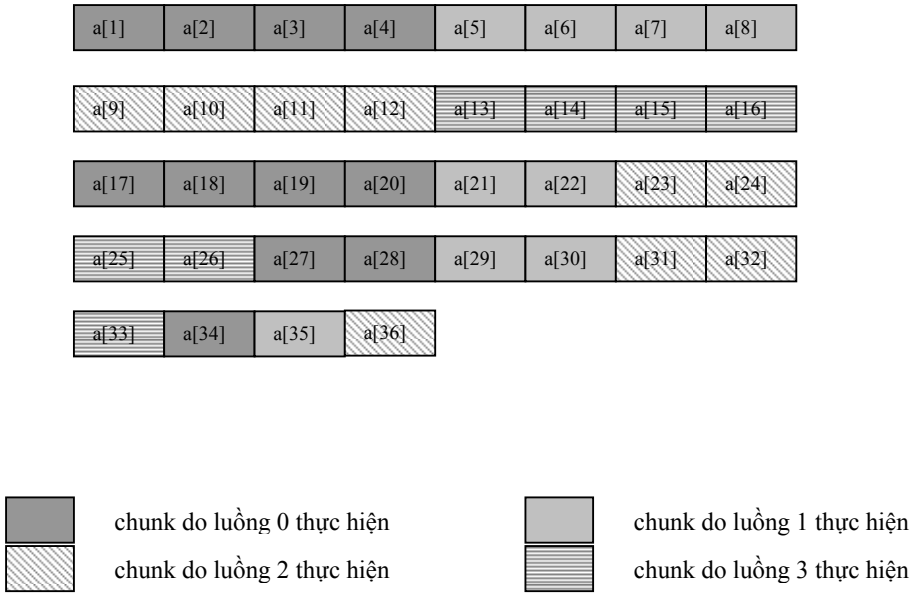
Hình 2.5: Mô tả hoạt động của bốn luồng thực thi tính a[1],a[2],...,a[8]

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 đi theo hàm mũ qua mỗi lần một luồng thực hiện song một chunk công việc và bắt đầu thực hiện một chunk công việc mới. Khi mà một luồng kết thúc một chunk công việc nó sẽ được gán động sang một chunk khác. Với *chunk_size* là 1 thì cỡ của chunk công việc được tính băng phép chia nguyên số lượng công việc cho số các luồng thực hiện và cỡ này sẽ giảm dần cho đến 1. Còn nếu *chunk_size* có giá trị k thì cỡ của chunk công việc sẽ giảm dần cho đến k. Chú ý cỡ của chunk cuối cùng có thể nhỏ hơn k. Khi mà giá trị của chunk_size không được khởi tạo thì giá trị mặc định của nó là 1

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 thực hiện của các luồng trong kiểu lập lịch guided với 36 bước lặp

RUNTIME

Khi mà bắt gặp *schedule (runtime)* thì công việc lập lịch bị hoãn lại cho tới khi *runtime*. Kiểu phân chia và cỡ của các chunk có thể được thiết lập tại thời điểm runtime bằng một biến môi trường có tên gọi 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* không được khởi tạo

• Mệnh đề ORDERED

Mệnh đề này chỉ được xuất hiện khi có chỉ thị ORDERED được bao bọc bởi chỉ thị DO/for

• Mệnh đề NOWAIT

Với mệnh đề này thì tất cả các luồng không cần đồng bộ tại điểm cuối cùng của vòng lặp song song. Các luồng sẽ xử lý trực tiếp đoạn mã lệnh cho tiếp sau vòng lặp. Các mệnh đề còn lại sẽ được thảo luận ở phần sau

2.3.4.2. Chỉ thị SECTIONS

Chỉ thị này dùng để chỉ ra các phần mã trong vùng song song chia cho các luồng thực hiện. Trong phạm vi của chỉ thị SECTIONS có các chỉ thị SECTION. Mỗi một SECTION sẽ được thực hiện bởi một luồng trong tập các luồng và các SECTION khác nhau sẽ được thực hiện bởi các luồng khác nhau. Trong C/C++ chi thị SECTIONS được cho dưới dạng sau

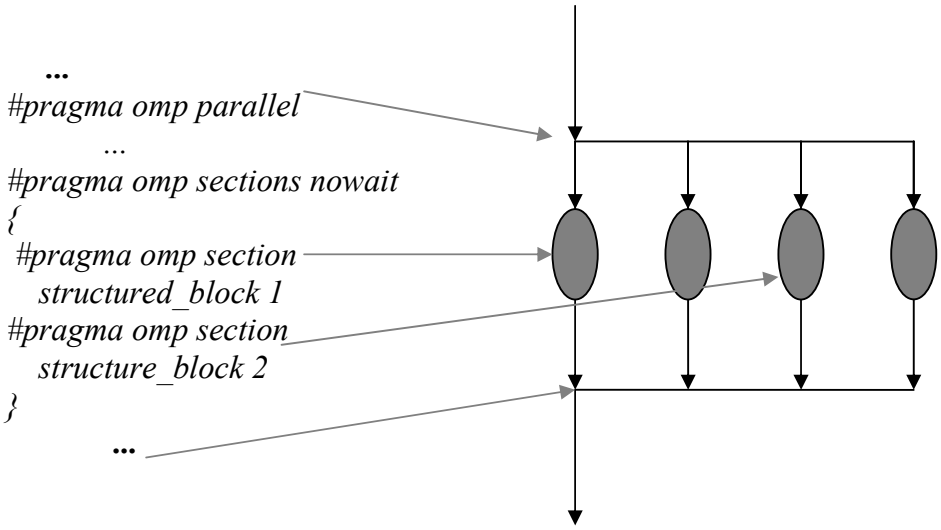
```
#pragma omp sections  [clause...] newline
                      private(list)
                      firstprivate(list)
                      lastprivate(list)
                      reduction(operator:list)
                      nowait
{
#pragma omp section  newline

     structured_block

#pragma omp section  newline

     structured_block
}
```

Ví dụ



Hình 2.7: Mô tả sự thực hiện của các luồng với chỉ thị section

2.3.4.3. Chỉ thị SINGLE

Mệnh đề SINGLE chỉ ra rằng đoạn mã bao quanh chỉ thị chỉ được thực hiện bởi một luồng trong tập các luồng. Trong C/C++ chỉ thị SINGLE được cho dưới dạng sau

#pragma omp sections [*clause...*] *newline*

private(*list*)

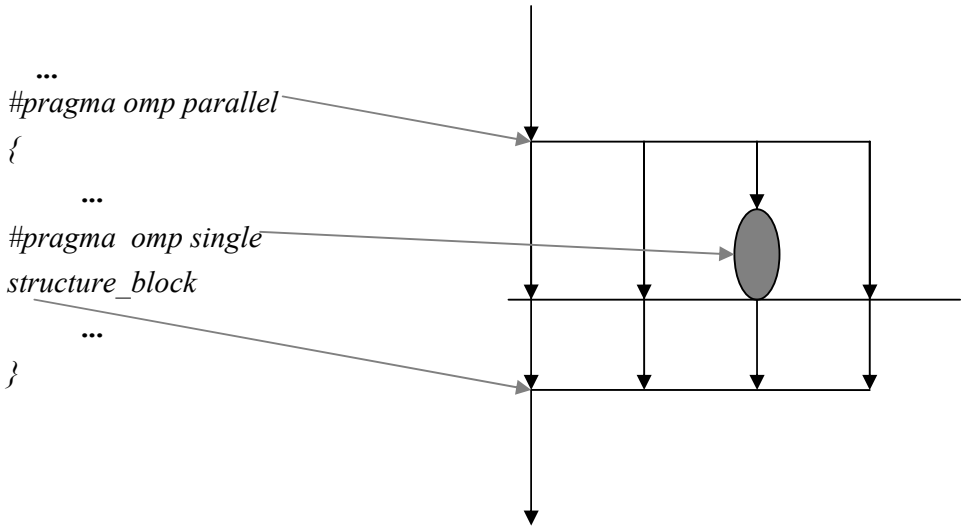
firstprivate(*list*)

nowait

Structure_block

Các luồng khác mà không thực thi đoạn mã trong chỉ thị SINGLE sẽ phải đợi đến khi luồng thực thi đoạn mã trong chỉ thị kết thúc mới được thực hiện các công việc ngoài chỉ thị SINGLE nếu không có mệnh đề NOWAIT được đưa ra. Lưu ý trong chỉ thị SINGLE chỉ có hai mệnh đề là private và firstprivate

Ví dụ



Hình 2.8: Mô tả sự thực hiện của các luồng
với chỉ thị single

2.3.5. Cấu trúc đồng bộ

Để nói về cấu trúc nay trước tiên ta giới thiệu một ví dụ đơn giản. Ví dụ này dùng hai luồng để thực hiện việc tăng giá trị của biến x tại cùng một thời điểm. Biến x ban 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 như sau

1. Luồng 1 nạp giá trị của x vào thanh ghi A
2. Luồng 2 nạp giá trị của x vào thanh ghi A
3. Luồng 1 thêm 1 vào thanh ghi A
4. Luồng 2 thêm 1 vào thanh ghi A
5. Luông 1 lưu thanh ghi A tại vị trí x

6. 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 của x là 1 chứ không phải là 2 như ta mong đợi. Và để tránh việc này sẩy ra việc tăng biến x phải được đồng bộ giữa hai luồng để đảm bảo rằng kết quả trả về là đúng. OpenMP cung cấp một cấu trúc đồng bộ giúp điều khiển sự thực hiện của các luồng liên quan đến nhau như thế nào. Trong cấu trúc đồng bộ có rất nhiều chỉ thị giúp cho việc đồng bộ chương trình sau đây là các chỉ thị đồng bộ đó.

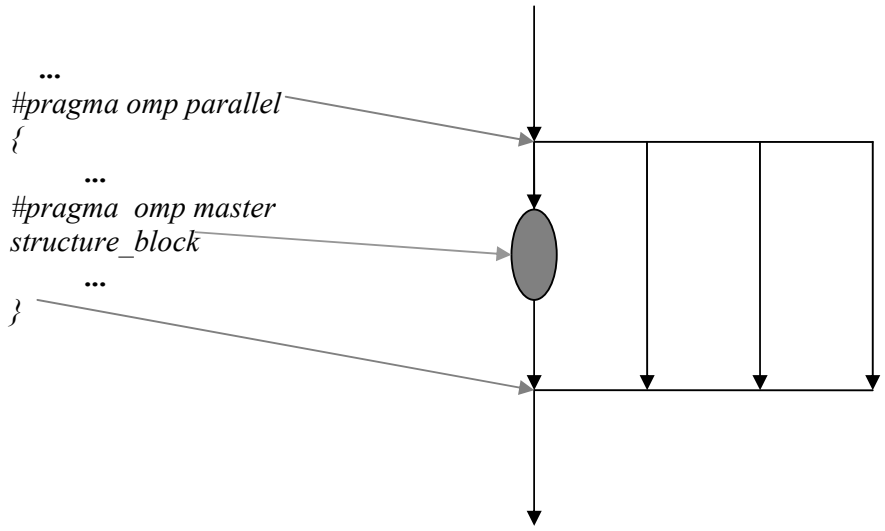
2.3.5.1. Chỉ thị MASTER

Trong chỉ thị MASTER đoạn mã bao quanh chỉ thị chỉ được thực hiện bởi luồng chủ trong tập các luồng . Trong C/C++ chỉ thị được cho dưới dạng sau

#pragma omp master *newline*

structure_block

Ví dụ



Hình 2.9: Mô tả sự thực hiện của các luồng với chỉ thị master

Trong chỉ thị loại này không có bất cứ mệnh đề nào và các luồng khác không cần chờ đến khi luồng chủ thực hiện xong công việc cho bởi chỉ thị master mới được thực hiện công việc của mình

2.3.5.2. Chỉ thị CRITICAL

Với chỉ thị CRITICAL thì vùng mã được cho bởi chỉ thị tại một thời điểm chỉ được thực hiện bởi một luồng. Trong C/C++ chit thị được cho dưới dạng sau

#pragma omp critical [*name*] *newline*

structure_block

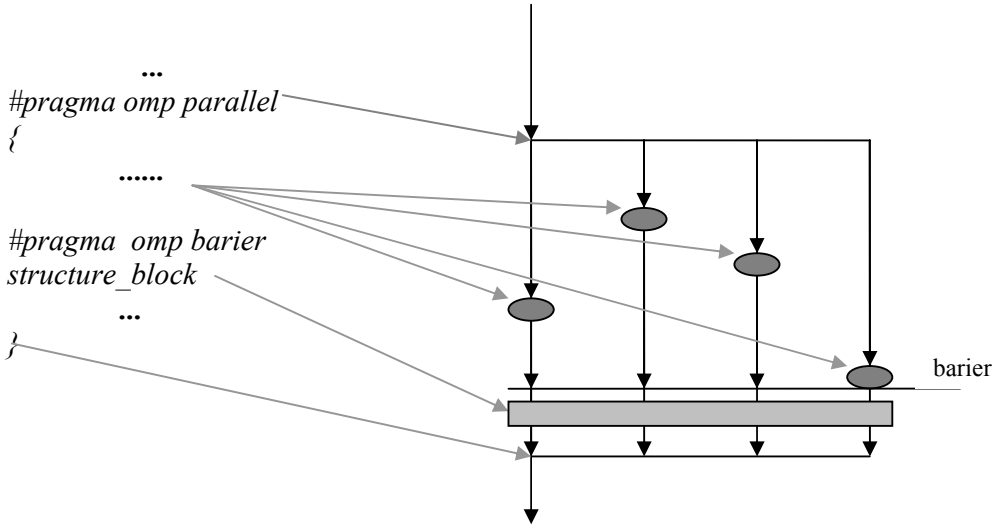
Ta lưu ý rằng nếu có một luồng đăng thực hiện công việc cho bởi chỉ thị mà có một luồng khác cố gắng đòi thực hiện cong việc đó thì nó sẽ bị khóa cho đến khi luồng kia thực hiện xong công việc đó. Một chú ý nữa là có thể tồn tại nhiều chỉ thị CRITIAL với các tên khác nhau trong một vùng song song. Tên của chỉ thị được nhận dạng một cách toàn cục, tất cả các vùng CRITIAL với tên giống nhau được coi như là cùng một vùng. Tất cả vùng CRITIAL không có tên cúng được coi như cùng một vùng

2.3.5.3. Chỉ thị BARRIER

Chỉ thị BARRIER dùng để đòng bộ tất cả các luồng trong tập các luồng. Khi bắt gặp chỉ thị BARRER thì mỗi một luồng sẽ chờ đợi tại thời điểm đấy (thời điểm bắt gặp chỉ thị BARRRIER) cho đến khi tất cả các luồng còn lại đều bắt gặp chỉ thị BARRIER. Và sau đó tất cả các luồng sẽ thực hiện đoạn mã cho bởi thỉ thị BARRIER. Trong C/C++ chỉ thị BARRIER được cho dưới dạng sau

#pragma omp barrier *newline*

structure_block



Hình 2.10: Mô tả sự thực hiện của các luồng với chỉ thị barrier

2.3.5.4. Chỉ thị ATOMIC

Trong chỉ thị ATOMIC các địa chỉ vùng nhớ được cập nhập một cách nguyên tố hơn là việc dùng nhiều luồng cố gắng ghi lên nó. Trong C/C++ chỉ thị này được cho dưới dạng sau

#pragma omp atomic *newline*

statemens_expression

Chỉ thị này chỉ áp dụng trực tiếp một trong các lệnh sau

x binop = expr

x++

++x

x- -

- - x

x là biến mở rộng

expr là một biểu thức mở rộng không tham chiếu đến x

binop là một trong +,*,- , / , & , ^ , | , ≥ or ≤

Chú ý rằng chỉ có phép nạp và lưu trữ biến x mới là nguyên tố

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 chỉ thị FLUSH xuất hiện tất cả các biến thread-visiable phải được ghi trở lại bộ nhớ. Trong C/C++ chỉ thị này được cho dưới dạng sau

#pragma omp flush (*list*) *newline*

Chú ý rằng danh sách lựa chọn ở đây chứa các biến cần flush để tránh việc flush tất cả các biến. Việc thực thi chỉ thị này phải đảm bảo rằng bất kì lần sửa đổi biến thread-visible lúc trước thì sau thời điểm đồng bộ thì nó được tất cả các luồng đều biết đến nó. Có nghĩa là trình biên dịch phải khôi phục các giá trị 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 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 theo thứ tự khi chúng được thực thi tuần tự. Trong C/C++ chỉ thị được cho dưới dạng sau

#pragma omp ordered *newline*

structure_block

Một chỉ thị ORDERED chỉ có thể xuất hiện trong phạm vi động của chỉ thị for hoặc parallel for trong C/C++. Và tại bất cứ thời điểm nào thì chỉ có một luồng thực hiện đoạn mã cho bởi chỉ thị ORDERED. Nếu một vòng lặp có chỉ thị này thì nhất định nó phải chứa mệnh đề ORDERED

2.3.6. Chỉ thị THREADPRIVATE

Chỉ thị này được dùng để tạo ra các biến có phạm vi toàn cục trong một file để các biến đó có thể được sử dụng ở nhiều vùng song song trong một file chương trình và chúng được bảo vệ bởi mỗi luồng. Trong C/C++ chỉ thị được cho dưới dạng sau

#pragma omp threadpivate (*list*)

Chú ý rằng trong chương trình chỉ thị này phải xuất hiện sau dòng lệnh khai báo các biến toàn cục. Mỗi một luồng sau đó sẽ tạo ra một bản sao của biến đó để mà việc thay đổi biến thuộc luồng nay không ảnh hưởng tới biến đó thuộc luồng khác

Ví dụ

```
#include <omp.h>
int alpha[10], beta[10], i;
#pragma omp threadprivate(alpha)
main ()  {
/* mở một luồng động */
  omp_set_dynamic(0);
/* vùng song song một */
#pragma omp parallel private(i,beta)
  for (i=0; i < 10; i++)
     alpha[i] = beta[i] = i;
/* vùng song song hai */
#pragma omp parallel
  printf("alpha[3]= %d and beta[3]= %d\n",alpha[3],beta[3]);
```

2.3. Các mệnh đề trong OpenMP

Vì 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. Phạm vi của các biến ở đây bao gồm hai phạm vi toàn cục và phạm vi bảo vệ. Các biến toàn cục bao gồm các biến tĩnh và biến file toàn cục còn các biến bảo vệ bao gồm biến chỉ số trong vòng lặp, biến trong thủ tục được gọi từ vùng song song. Các mệnh đề về phạm vi dữ liệu bao gồm các mệnh đề sau

•PRIVATE

•FIRSTPRIVATE

•LASTPRIVATE

•SHARED

•DEFAULT

•REDUCTION

•COPYIN

Các mệnh đề về phạm vi dữ liệu này được sử dụng với một vài chỉ thị

(PARALLEL, FOR, SECTIONS) để điều khiển phạm vi các biến trong các chỉ thị đó. Sau đây ta sẽ đi vào chi tiết từng mệnh đề

2.4.1. Mệnh đề PRIVATE

Mệnh đề này dùng để khai báo các biến trong danh sách các biến dùng riêng cho mỗi luồng . Mỗi luồng sẽ sử dụng một bản sao của biến PRIVATE và có quền sử dụng độc lập đối với biến đó . Trong C/C++ nó được khai báo như sau

private (*list*)

2.4.2. Mệnh đề FIRSTPRIVATE

Mệnh đề này cũng dùng để khai báo một danh sách các biến dùng riêng cho mỗi luồng giống như ở mệnh đề PRIVATE. Nhưng nó khác mệnh đề PRIVATE ở chỗ các bản sao của mỗi biến dùng cho mỗi luồng được khởi tạo một giá trị ban đầu trước vùng song song hoặc cấu trúc chia sẻ công việc. Trong C/C++ mệnh đê trên được khai báo như sau

firstprivate (*list*)

2.4.3. Mệnh đề LASTPRIVATE

Mệnh đề này cũng được dùng để khai báo một danh sách các biến dùng riêng cho mỗi luồng như ở mệnh đề PRIVATE. Nhưng nó khác mệnh đề PRIVATE ở chỗ giá trị của biến chính là giá trị của biến dùng riêng của luồng thực hiện công việc cuối cùng của vòng lặp hoặc section cuối cùng trong chỉ thị sections. Trong C/C++ mệnh đề trên được khai báo như sau

#pragma omp lastprivate (*list*)

2.3.4. Mệnh đề SHARED

Mệnh đề này dùng để khai báo các biến trong danh sách các biến được chia sẻ cho tất cả các luồng trong tập các luồng. Các biến chia sẻ chỉ có một vị trí trong bộ nhớ và các luồng sẽ đọc và ghi trên cùng một vị trí đấy. Việc các luồng cùng đọc và ghi lên cùng một vị trí trên bộ nhớ rất có thể dẫn đến sai xót trong chương trình nên người lập trình phải chịu trách nhiệm phân bố công việc cho mỗi luồng một cách hợp lý (ví dụ như thông qua chỉ thị CRITIAL). Trong C/C++ mệnh đề trên được khai báo như sau

shared (*list*)

2.3.5.Mệnh đề DEFAULT

Mệnh đề này cho phép người lập trình đưa ra phạm vi PRIVATE, SHARED, hoặc NODE cho tất cả các biến thuộc vàophạm vi của bất kỳ vùng song song nào. Và chỉ có chỉ thị DEFAULT mới được đưa ra trong cấu trúc vùng song song. Trong C/C++ chỉ thị này được khai báo như sau

default (shared |none)

2.3.6. Mệnh đề REDUCTION

Mệnh đề này được dùng để thu g ọn các biến có ở trong danh sách các biến. Một bản sao của mỗi biến cho bởi danh sách các biến sẽ được tạo ra cho mỗi luồng. Tại thời điểm cuối cùng của việc thu gọn thì các phép toán thu gọn sẽ được áp dụng nên các bản sao dùng riêng của mỗi luồng. Và kết quả cuối cùng được lưu vào biến chia sẻ toàn cục. Trong C/C++ mệnh đề trên được khai báo như sau

reduction (*operator: list*)

Chý ý các biến trong danh sách phải là các biến vô hướng. Chúng không thể là các biến kiểu mảng hoặc kiểu có cấu trúc và chúng phải được khai báo là biến chia sẻ.

Các thao tác thu gọn thì không áp dụng được với các số thực. Mệnh đề REDUCTION được sử dụng trong vùng song song hoặc cấu trúc chia sẻ công việc với các biến thu gọn thì trong cấu trúc và vùng song song này chỉ được sử dụng các dòng lệnh có dạng như sau

x = x op expr
x = expr op x (exceptsubtraction)
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 x

op là một trong những phép toán +, *, -, /, &, ^, |, &&, ||

binop là một trong những phép toán +, *, -, /, &, ^, |

2.3.7. Mệnh đề COPYIN

Mệnh đề này dùng để gán giá trị của biến THREADPRIVATE cho từng luồng trong tập các luồng thực thi một vùng song song. Có nghĩa là giá trị của biến THREADPRIVATE được khai báo trong mệnh đề COPYIN của luồng chủ sẽ được dùng làm nguồn. Khi gặp một vùng song song thì biến nguồn này sẽ được sao cho các luồng thực thi vùng song song đó. Lưu ý rằng các biến khai báo trong mệnh đề COPYIN là các biến THREADPRIVATE. Trong C/C++ mệnh đề trên được khai báo như nhau

Copyin (*list*)

2.5. Thư viện Run-Time

OpenMP cung cấp một thư viện với rất nhiều các hàm chức năng bao gồm các truy vấn liên quan đến số lượng và chỉ số các luồng, thiết lập số lượng các luồng sử dụng, semaphores, và các hàm thiết lập môi trường thực thi. Trong C/C++ để có thể sử dụng các hàm trên thì phải đính vào file thư viện *omp.h*. Sau đây ta đi vào chi tiết các từng hàm thư viện một

2.5.1. OMP_SET_NUM_THREADS

Hàm thư viện này dùng để thiết lập số lượng các luồng để thực hiện vùng song song tiếp sau. Trong C/C++ hàm này đươc cho dưới dạng sau

include <omp.h>

Void omp_set_num_threads(*int num_threads*)

Hàm này phải được khai báo trong vùng tuần tự và khi đến vùng song song gần ngay nó thì vùng song song đó sẽ được thực hiện với số lượng các luồng mà nó đưa ra. Có một cách thiết lập số lượng các luồng khác là dùng biến môi trường OMP_NUM_THREADS

2.5.2. OMP_GET_NUM_THREADS

Hàm này được gọi từ vùng song. Khi được gọi nó sẽ trả về số lượng các luồng thực hiện vùng song song đó. Trong C/C++ nó được cho dưới dạng sau

include <*omp.h*>

int omp_get_num_thread ( )

Nếu hàm này được gọi ra từ vùng tuần tụ thì nó sẽ trả lại kết qyả là 1 .

2.5.3. OMP_GET_MAX_THREADS

Hàm này trả về giá trị lợn nhất trong các giá trị trả về của các hàm OMP_GET_NUM_THREADS. Trong C/C++ nó được cho dưới dạng sau

include <*omp.h*>

int omp_get_max_threads ( )

Hàm này có thể gọi cả ở vùng tuần tự lẫn vùng song song

2.5.4. OMP_GET_THREAD_NUM

Hàm này trả về chỉ số của luồng đang thực hiện . Chỉ số này nằm trong khoảng từ 0 đến OMP_GET_NUM_THREADS – 1. Và luồng chủ luân mang chỉ số 0 . Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

int omp_get_thread_num ()

Nếu nó được gọi từ vùng tuần tự thì kết quả trả về là 0

2.5.4. OMP_GET_NUM_PROCS

Hàm này trả về số lượng các bộ sử lý thực thi chương trinh tại thời điểm nó được gọi. Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

int omp_get_num_procs ()

2.5.5. OMP_IN_PARALLEL

Hàm này được dùng để kiểm tra xem vùng mã chứa nó được thực hiện song song hay tuần tự. Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

int omp_in_parallel ()

Nếu vùng thực hiện là vùng song song thì nó sẽ trả về một giá trị khác 0. Còn nếu là vùng tuần tự hoặc trong phạm vi động của vùng song song nó sẽ trả về 0

2.5.7. OMP_SET_DYNAMIC

Hàm này dùng để cho phép hoặc không cho phép sự điều chỉnh động của các luồng thực thi trong vùng song song. Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

void omp_set_dynamic (*int dynamic_thread*)

Nếu dynamic_thread khác 0 thì điều chỉnh động sảy ra nghĩa là các luồng có thể thực thi hơn một vùng song song. Hàm này có thể thay thế bằng việc sử dụng biến môi trường OMP_DYNAMIC và nó phải được gọi từ vùng tuần tự

2.5.8. OMP_GET_DYNAMIC

Hàm này dùng để kiểm tra xem có sự điều chỉnh động hay không. Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

int omp_get_dynamic ()

Hàm này sẽ trả về giá trị khác 0 nếu trong chương trình có sự điều chỉnh luồng động còn nếu không thì hàm sẽ trả về giá trị 0

2.5.9. OMP_SET_NESTED

Hàm này được sử dụng để cho phép hay không cho phép việc song song lồng . Trong C/C++ Hàm này được cho dưới dạng sau

include <*omp.h*>

void omp_set_nested (int *nested*)

Nếu nested mà khác 0 có nghĩa là việc song song lồng có thể xẩy ra còn nếu nested khác 0 thì việc song song lồng không thể xẩy ra. Một cách mặc định thì không cho phép song song lồng. Nếu không sử dụng hàm này thì có thể sử dụng thông qua biến môi trường OMP_NESTED

2.5.10. OMP_GET_NESTED

Hàm này được sử dụng để nhận biết xem có sự song song lồng sẩy ra không. Trong C/C++ nó được cho dưới dạng sau

include <*omp.h*>

int omp_get_nested ()

Hàm sẽ trả về giá trị khác 0 nếu không tồn tại song song lồng. Trong trường hợp ngược lại hàm sẽ trả về giá trị 0

2.5.11. OMP_INIT_LOCK

Hàm này dùng để thiết lập một khóa thông qua các biến khóa. Trong C/C++ được cho dưới dạng sau

include <*omp.h*>

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 tách ra các biến khóa từ bất kì khóa nào. Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

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 được dùng để bắt buộc sự thực hiện của các luồng phải chờ đợi khi khóa được mở. Với giả sử rằng các luồng đó được quền sở hữu khóa đó. Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

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 được sử dụng để giải thoát sự thực hiện của các luồng vào khóa. Trong C/C++ hàm này được cho dưới dạng sau

include <*omp.h*>

void omp_unset_lock (*omp_unset_t * lock*)

void omp_unset_nest_lock (*omp_unset_nest_lock * lock*)

2.5.15. OMP_TEST_LOCK

Hàm này được dùng để cố gắng thử đặt một khóa. Nếu đặt thành công thì nó sẽ trả về giá trị khác không ngược lại nó sẽ trả về 0. Trong C/C++ hàm này được cho dưới dạng sau

include <omp.h>

int omp_test_lock (omp_lock_t * lock)

int omp_test_nest_lock (omp_nest_t * lock)

2.6. Các biến môi trường trong OpenMP

Các biến môi trường được dùng để điều khiển sự thực hiện đoạn mã song song. Bao gồm các biến môi trường sau

2.6.1. OMP_SCHEDULE

Biến này chỉ được sử dụng trong chỉ thị có kiểu lập lịch RUNTIME như chỉ thị for và parallel for. Giá trị của biến này để xác định xem các công việc trong vòng lặp được lập lịch trên các BXL như thế nào

Ví dụ :

setenv OMP_SCHEDULE “guided , 4”

setenv OMP_SCHEDULE “dynamic”

2.6.2. OMP_NUM_THREADS

Hàm này dùng để thiết lập số lượng lớn nhất các luồng được sử dụng

Ví dụ :

setenv OMP_NUM_THREADS 8

2.6.3. OMP_DYNAMIC

Biến này cho phép hay không cho phép sự điều chỉnh động cho các luông thực thi các vùng song song. Nó nhận hai giá trị TRUE hoặc FALSE

Ví dụ

setenv OMP_DYNAMIC TRUE

2.6.3. OMP_NESTED

Biến này dùng để cho phép hay không cho phép việc song song lồng sẩy ra. Nó nhận hai giá trị TRUE hoặc FALSE .

Ví dụ :

setenv OMP_NESTED TRUE

2.7. Trình biên dịch OpenMP

Trình biên dịch được dùng để biên dịch chương trình OpenMP được dùng trong khóa luận là trình biên dịch Intel C++ compiler for Linux. Chương trình dịch này được download từ địa chỉ https://premier.intel.com/FileDownLoads. aspx.

Cách biên dịch chương trình cho ứng dụng OpenMP :

icc –openmp chuongtrinhnguon –o filechay

Khi chạy chương trình ứng dụng OpenMP

./filechay

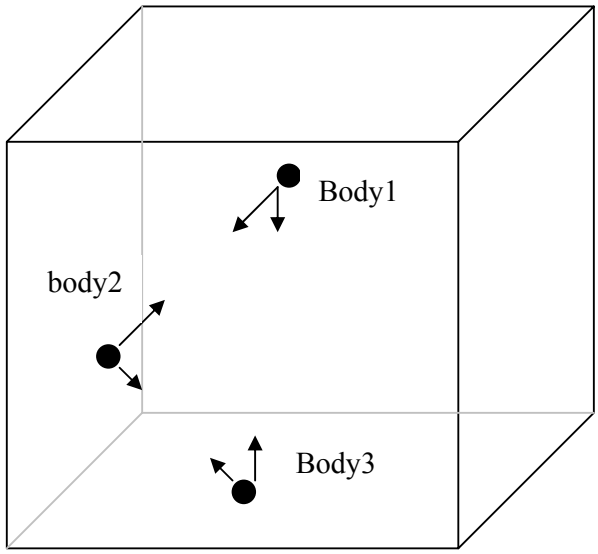
Chương 3: Bài toán mô phỏng N-Body

Chương này đi vào miêu tả bài toán N-body song song hóa thuật toán tính lực tương tác trực tiếp giữa các body qua ba cách và qua đó rút ra kết luận về hiệu quả của chương trình song song và sự khác nhau giữa các chiến lược song song hóa

1.1. Giới thiệu chung về bài toán mô phỏng N-body

Bài toán mô phỏng N-body là bài toán cho việc mô phỏng chuyển động của một tập các body dưới tác dụng của lực hấp dẫn hoặc các lực khác. Từ một trạng thái ban đầu cho trước, dưới tác dụng của lực tương tác giữa các body nên gia tốc và vận tốc của các body được thay đổi liên tục nên chúng sẽ chuyển động đến vị trí khác. Cứ như vậy vị trí của các body được thay đổi liên tục. Để thực hiện việc mô phỏng chuyển động của các body trong hệ chúng ta phải xác định và cập nhập các vị trí của các body tại bất kì thời điểm nào trong xuất thời gian mô phỏng

1.2. Mô tả bài toán N-body



● Chỉ các body

→ Lực tác dụng lên từng body

body1 chịu lực tương tác với body2 và body3
body2 chịu lực tương tác với body1 và body3
body3 chịu lực tương tác với body1 và body2

Hình 3.1 Mô tả hệ gồm 3 body

Trong hệ N-body trên lực tương tác giữa các body là lực hấp dẫn giữa các body.

$$F = \frac{Gm_1m_2}{r^2_{12}}$$

m_1 và m_2 là khối lượng hai body
r_{12} là khoảng cách giữa hai body
G là hằng số hấp dẫn

Tổng lực tác dụng lên mỗi body là tổng hợp của tất cả các lực hấp dẫn tác dụng nên nó

F = ma m là khối lượng của body, a là gia tốc của body

Với tổng lực F làm cho mỗi body di chuyển một đoạn đường có chiều dài là x với

vận tốc v và gia tốc a

x x là độ dài đoạn đường dịch chuyển của body

v = x' v là vận tốc dịch chuyển của body

a = v' = x" a là gia tốc chuyển động của Body

♣ Trạng thái của hệ thống tại thời điểm T

$n_1(x_1,T)$, $n_2(x_2,T)$, $n_3(x_3,T)$ .....$n_N(x_N,T)$

Trong đó ni (i = 1,2,3,..,N) là các body, xi là các thông tin trạng thái của ni tại thời điểm T. Các thông tin trạng thái ở đây bao gồm vị trí, vận tốc và khối lượng của body

1.3. Các bước trong quy trình giải bài toán mô phỏng N-body

Với mô tả bài toán mô phỏng N-body thì đã có rất nhiều phương pháp được đưa ra để xác định thông tin trạng thái của các body trong hệ mô phỏng tại một thời điểm bất kỳ. Tuy nhiên tất cả các bước đều có những đặc điểm chung sau

(1)Biểu diễn hệ thống

(2)Tính toán năng lượng và lực tương tác giữa các body trong hệ

(3)Tính và cập nhập trạng thái của các body

(4)Quá trình được lặp đi lặp lại tại bước thứ hai

Tất cả các phương pháp được đưa ra đều dưới một mục đính là giảm thời gian tính lực bởi vì trong cả quá trình mô phỏng thì giai đoạn tính lực là giai đoạn chiếm chủ yếu thời gian. Phần cài đặt của khóa luận sẽ đi vào việc song song hóa giai đoạn trên bằng việc sử dụng OpenMP.

Có rất nhiều thuật toán để làm giảm thời gian của giai đoạn tính lực. Khóa luận sẽ cài đặt với thuật toán song song bởi vì thuật toán này sẽ rất hiệu quả với hệ mô phỏng với số lượng lớn các body. Dưới đây là tóm tắt thuật toán tính lực trực tiếp tuần tự. Song song hóa thuật toán

• Khái quát chương trình áp dụng thuật toán tính lực trực tiếp tuần tự gồm những bước chính sau

(1)Đọc dữ liệu từ file hoặc từ tham số chương trình

(2)Tính năng lượng khởi tạo

(3)Cập nhập vị trí các body

(4)Tính lực tương tác giữa các body để xác định vị trí mới

(5)Lặp lại bước ba

Việc tính lực giữa các body được mô tả ngắn gọn như sau

```
for(int i=0 ;i<N ;i++)
{
  for(int j=0;j<N;j++)
   {
     Tính lực tương tác giữa body i và body j;
     Cộng dồn lực;
   }
}
```

Cộng dồn lực ở đây có nghĩa là cộng tất cả N-1 lực tương tác của N-1 body tác dụng lên body i

• Song song hóa thuật toán tính lực trực tiếp tuần tự :

Để song song hóa thuật toán tính lực trực tiếp tuần tự ta có thể song song theo các cách sau :

(1) Song song hóa vòng for đầu tiên

Việc song song này được mô tả ngắn gọn như sau

```
#pragma omp parallel  default(shared)  private(i,j)
{
 #pragma omp for schedule(static)  /* N là số Body*/
      for(int i = 0 ; i < N ; i ++) {
        for(int j = 0 ; j<N ; j ++) {
          Tính lực tương tác giữa body i và body j;
          Cộng dồn lực;
         }
       }
  }
```

Với việc song song hóa kiểu như trên thì việc tính lực của các body được phân bố cho các BXL như sau.

x = N/n,y = N%n

BXL0 N/n + 1

BXL1 N/n + 1

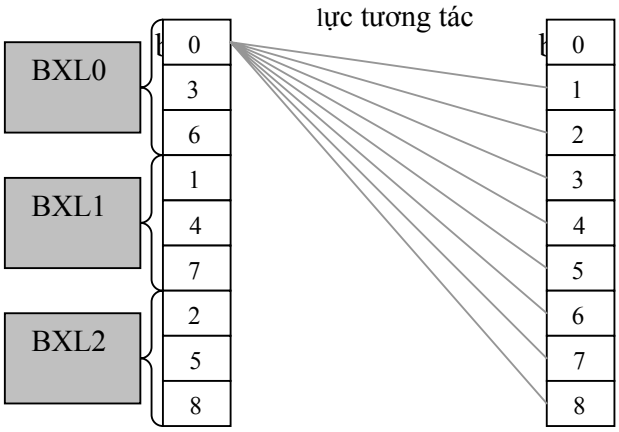
BXL2 N/n + 1

.....

BXLy-1 N/n nếu y>0

.....

BXLn N/n



Hình 3.2: Mô tả thuật toán song song vòng for thứ nhất với n=3, N=9

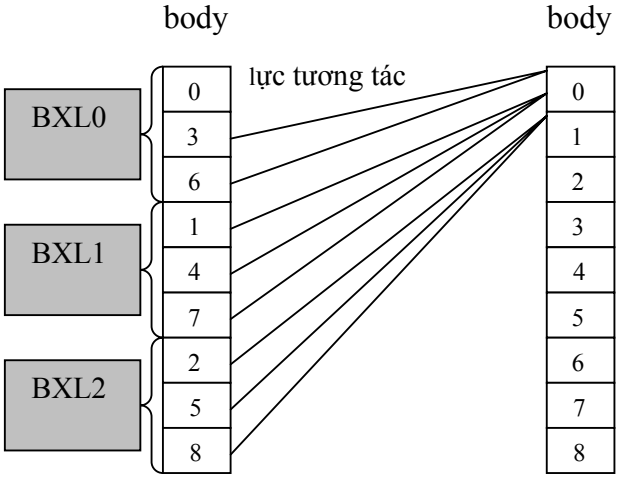
Đối với mỗi body thuộc một BXL việc tính lực tương tác của nó với N-1 body còn lại và việc cộng dồn các lực đó lại sẽ được thực hiện trên BXL đảm nhiệm tính lực tác dụng lên body đó. Khác với MPI là để tính lực tương tác lên mỗi body thì BXL phải gửi thông điệp yêu cầu thông của các Body trên các BXL khác. Còn trong OpenMP làm việc trên các dữ liệu chia sẻ nên các BXL hoàn toàn biết thông tin của các Body nằm trên BXL khác mà không cần phải dùng thao tác gì

(2) Song song hóa vòng for thứ hai

Việc song song hóa này được mô tả ngắn gọn như sau

```
for (int i = 0 ; i < N ; i ++){
 #pragma omp prarallel shared(i,N) {
  #pragma omp for schedule(static)  redution(+; force(i))
/* N là số Body, force(i) là tổng lực tác dụng lên Body i*/
   for (int j = 0 ; j < N ; j ++) {
      Tính lực tương tác giữa body i và body j;
      Cộng dồn lực
   }
       }
}
```

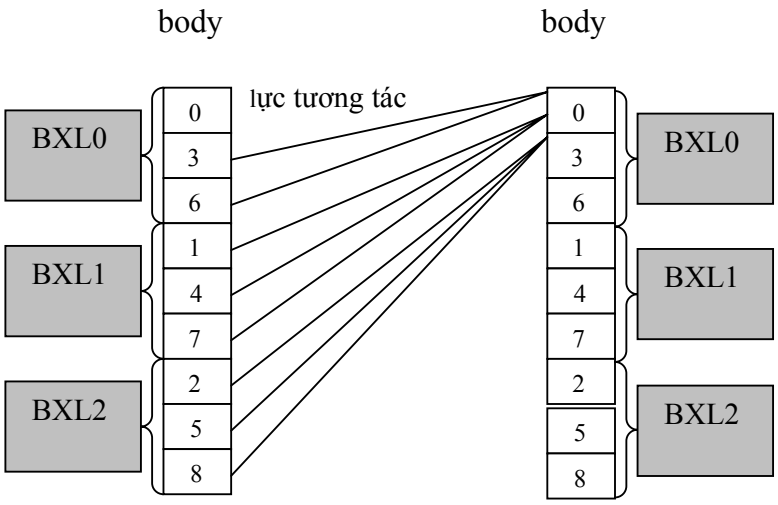
Với việc song song hóa như trên thì việc tính tổng các lực tương tác tác dụng lên từng body không phải được thực hiện trên mỗi một BXL. Mà việc tính các lực tương tác đó chia đều cho mỗi BXL và tổng hợp kết quả ngay trên BXL đó sau đó tổng lực tác dụng lên body được tổng hợp từ các kết quả tổng hợp trên từng BXL



Hình 3.3 Mô tả thuật toán song song vòng for thứ hai với n=3, N=9

(3) Song song hóa cả hai vòng for

```
# pragma omp parallel default(shared)
 {
      #pragma omp for schedule(static)
       for (int i = 0 ; i < N ; i ++) {
          omp_set_nested(1); // cho phép song song lồng
          #pragma omp parallel
           {
            #pragma omp shared(i,N) schedule(static)  reduction(+ : force(i) )
         /*N là số các body, force(i) là tổng lực tác dụng lên body i */
            for(int j = 0 ; j < N ; j ++)
              Tính lực tương tác giữa body i và body j;
              Cộng dồn lực;
           }
         }
      }
 }
```



Hình 3.4: Mô tả thuật toán song song hai vòng for với n=3, n=9

Với việc song song hóa hai vòng for như trên thì việc tính tổng lực lên các body được thực hiện trên từng BXL. Nhưng việc tính các lực tương tác lên một body không phải do một BXL chứa body đấy thực hiện mà thực hiện trên tất cả các BXL sau đó tổng hợp các lực đấy lại thành tổng lực tác dụng lên body đấy

Tuy nhiên với ba cách song song hóa trên thì cách thứ nhất hiệu quả hơn hai cách sau vì khi Khi tạo ra nhiều vùng hoặc luồng song song thì hệ đh mất thời gian tạo luồng, do đó nếu thời gian giảm đi do song song hóa nhỏ hơn thời gian tạo luồng thì cách song song hóa đó là không có lợi. Khi đó chương trình song song có thể chạy chậm hơn tuần tự. Bởi vậy cần lựa chọn các vùng song song để có lợi nhất về hiệu

năng. Trong cách song song thứ nhất thì chỉ cần một lần để tạo các luồng nên không tốn nhiều thời gian, con đối với hai cách còn lại thì số lần mà nhân hệ điều hành tạo ra các luồng bằng với số lượng các body. Với bài toán mô phỏng N-body thì số lượng các body la rất lớn nó nên đến hàng triệu hạt cho nên thuật toán song song hóa hàm tính lực được cài đặt theo cách thứ nhất có nghĩa là song song hóa vòng for đầu tiên

1.4. Kết quả thực nghiệm

Dưới đây là các kết quả thực nghiệm thu được khi chạy chương trình mô tả bài toán mô phỏng N-body của chương trình song song và chương trình tuần tự. Cả hai chương trình trên được cài đặt trên một máy có hai BXL vật lý có hỗ trợ công nghệ siêu phân luồng cho nên máy tính đó xem như có 4 BXL logic. Nhưng công nghệ siêu phân luồng chỉ hỗ trợ các bài toán vào ra và lập trình luồng, còn đối với bài toán N-body này ta coi như máy đó có 2 BXL. Kết quả về sự phụ thuộc thời gian và số các body trong hai chương trình song song và tuần tự cài đặt trên một máy với 2 BXL được cho dưới bốn bảng sau.

Bảng 3.1: Mô tả sự phụ thuộc thời gian tính lực tương tác và số lượng các body trong chương trình tuần tự

Số body(k)	Thời gian(t)
2	0,31336
4	1.25624
8	5.03610
16	20.37267
32	81.74523
64	327.89952

Bảng 3.2: Mô tả sự phụ thuộc thời gian tính lực tương tácvà số lượng các body trong chương trình song song vòng for thứ nhất

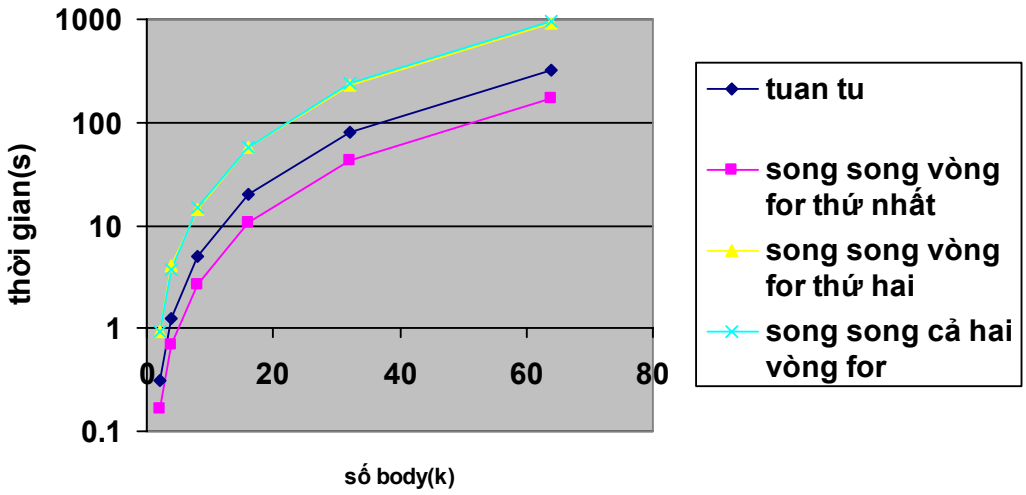
Số body(k)	Thời gian(t)
2	0.16629
4	0.68406
8	2.66889
16	10.72424
32	42.92527
64	173.31044

Bảng 3.3: Mô tả sự phụ thuộc thời gian tính lực tương tác và số lượng các body trong chương trình song song vòng for thứ hai

Số body(k)	Thời gian(t)
2	0.92078
4	4.11586
8	14.22576
16	57.03371
32	230.36806
64	926.65886

Bảng 3.4 Mô tả sự phụ thuộc thời gian tính lực tương tác và số lượng các body trong chương trình song song hai vòng for

Số body(k)	Thời gian(t)
2	0.93970
4	3.78118
8	15.00492
16	58.47619
32	235.08394
64	940.89745



Hình 3.1: Biểu đồ mô tả sự phụ thuộc giữa số lượng body và thời gian xác định vị trí mới của các body trong cả các chương trình song song và tuần tự.

1.4.1. Đánh giá, nhận xét

Qua kết quả thống kê ở trên ta thấy. Thời gian của chương trình song song trên 2 BXL giảm được gần một nửa so với chương trình tuần tự trên 1 BXL vì công việc được chia cho 2 BXL thực hiện đồng thời. Sở dĩ thời gian không thể giảm đi đúng một nửa là vì sự thiếu đồng bộ của hai BXL và nhân của hệ điều hành mất một phần thời gian để thiết lập một vùng song song khi bắt gặp một cấu trúc song song

Kết luận

Trong khuôn khổ của khóa luận này, chúng tôi nghiên cứu xử lý song song trên máy tính chia sẻ bộ nhớ với OpenMP và ứng dụng trong bài toán mô phỏng N-Body.

Việc tổng kết cơ sở lý thuyết về xử lý song song là cơ sở cho nghiên cứu sau này của chúng tôi.

Định hướng của khóa luận về phát triển các úng dụng song song với OpenMP. Đi sâu vào nghiên cứu các cấu trúc, chỉ thị, các hàm thư viện và các biến môi trường trong OpenMP nhằm mục đính phân phối các nhiệm vụ một cách hợp lý cho các BXL để song song hiệu quả một chương trình tuần tự trên máy tính chia sẻ bộ nhớ.

Trong thử nghiệm chúng tôi đã so sánh chiến lược song song hóa trên một ứng dụng bài toán mô phỏng N-body và đã thu được kết quả như mong đợi. Với việc song song hóa đã làm giảm thời gian tính toán so với chương trình tuần tự . Thời gian giảm được xấp xỉ hai lần với hai BXL. Nhưng việc song song hóa không phải trong trường hợp nào cũng hiệu quả về mặt thời gian. Khóa luận cũng cho thấy với các chiến lược song song hóa khác nhau cho ta những hiệu quả về thời gian khác nhau. Nếu không song song hóa một cách hợp lý th ì có thể xẩy ra nghịch lý về song song có nghĩa là

thời gian thực hiện chương trình song song lớn hơn thời gian thực hiện chương trình tuần tự.

Hướng phát triển trong tương lai

Hướng phát triển tiếp theo của khóa luận không chỉ dừng lại ở việc phát triển các ứng dụng với OpenMP trên cấu trúc chia sẻ bộ nhớ chung mà còn nghiên cứu xử lý song song trên cấu trúc bộ nhớ phân tán và trong cấu trúc bộ nhớ lai.vv. Trong quá trình nghiên cứu sau còn xem xét đến cả vấn đề hiệu suất của các BXL trong qúa trình phân phối công việc. Và các úng dụng được song song hóa sẽ là các ứng dụng có kích cỡ lớn hơn, thuật toán phức tạp hơn.

Tài liệu tham khảo

Tiếng Việt

[1] Nguyễn Việt Anh. Xử lý song song trên PVM và ứng dụng trong bài toán bảo mật thông tin. Luận văn thac sĩ. Hà nội. 2003)

Tiếng Anh

[1] Rohit Chandra, Leonardo Dagum, Dave Kohr, Dror Maydan, Jeff McDonald, Ramesh Menon. Parallel Programming in OpenMP

[2] http://www.llnl.gov/computing/tutorials/parallel_comp

[3] http://www.openmp.org

[4] http://www.llnl/computing/tutorials/workshop/openmp/

[5] http://www hpcc.unical.it/alarico/LNErbacci2.pdf

[6] http://nereida.deicc.ull.es/html/openmp/minnrsota/tutorial/content_openmp.html

[7] http://www.schlitt.net/xstar/n-body.pdf

[8] http://www.amara.com/papers/n-body.html