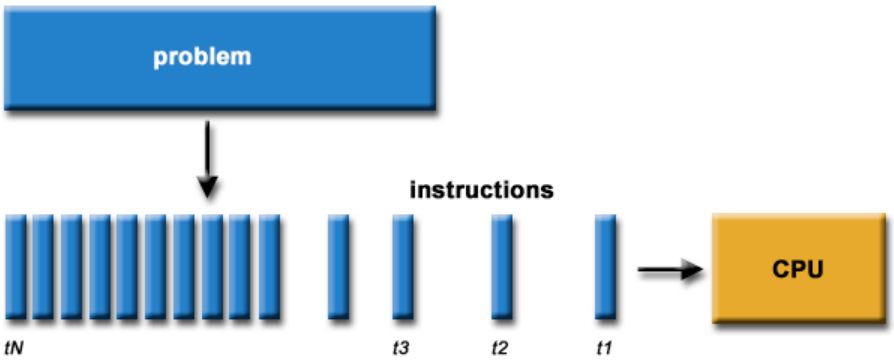
Mục Lục

Chương I – Giới thiệu về tính toán song song 2
I. Giới thiệu chung 2
II. Kiến trúc máy tính 3
III. Phân loại máy tính song song 4
Chương II – Thiết kế thuật toán song song 10
I. Phân chia dữ liệu 10
II. Giao tiếp giữa các tác vụ 12
III. Đồng bộ 13
IV. Phụ thuộc dữ liệu 17
V. Ví dụ 18
VI. Các bước tạo một chương trình song song 19
Chương 3 : Các mô hình lập trình song song 22
I. Giới thiệu 22
II. Lập trình chia sẻ bộ nhớ dựa vào tiến trình 23
III. Lập trình Thread 25
IV. Lập trình Message passing trong Distributed Memmory 25
V. Các mô hình khác 26
Chương IV - PTHREAD POSIX (Portable Operating System Interface) 27
I. Giới thiệu về Thread 27
II. Cơ bản về Pthread 28
Chương 5 : Lập trình MPI 40
I. Giới thiệu về MPI 40
II. Xây dựng phương thức Send và Receive 40
III. Lập trình MPI 43
IV. Truyền thông tập hợp trong MPI 46

Chương I – Giới thiệu về tính toán song song

I. Giới thiệu chung

- Theo định luật Moore : " Số lượng transistor trên mỗi một inch vuôn sẽ tăng gấp đôi sau mỗi năm". Khi số lượng transistor tăng thì tốc độ tính toán của CPU tăng theo, tuy nhiên tản nhiệt không theo kịp tốc độ của CPU.
- Trước đây, phần mềm được viết theo kiểu tuần tự (serial), khi đó chương trình sẽ được chia thành nhiều câu lệnh riêng biệt (instruction) , mỗi lệnh chỉ chạy được trên 1 CPU, tại một thời điểm…



Hình 1.1. – Tính toán tuần tự

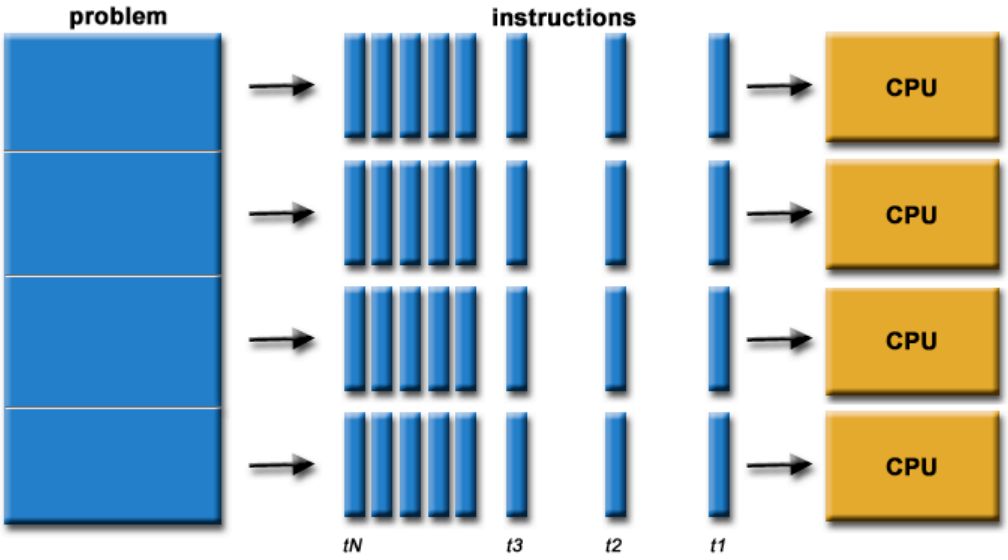
- Vậy tính toán song song là gì?

  Tại một thời điểm, vấn đề sẽ được giải quyết nhờ vào nhiều **tài nguyên tính toán** khác nhau.

  Chương trình được chia thành nhiều phần mà có thể giải quyết đơn lẻ, và từng phần sẽ được xử lý bởi các CPU khác nhau.

  Khái niệm :

  Xử lý song song là quá trình xử lý gồm nhiều tiến trình được kích hoạt đồng thời và cùng tham gia giải quyết một bài toàn, nói chung xử lý song song được thực hiện trên những hệ thống đa bộ xử lý.



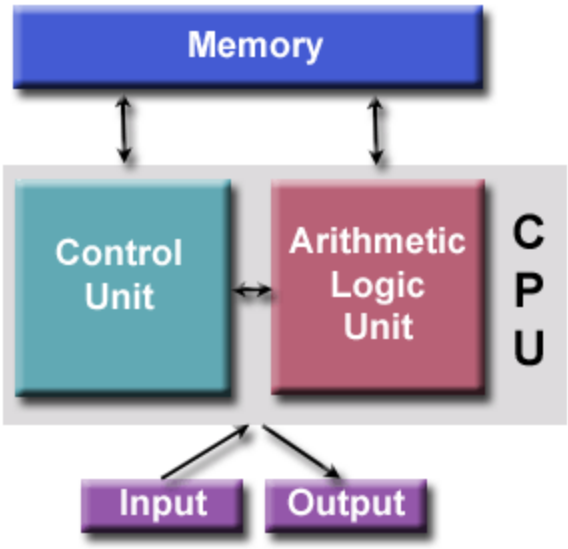
Hình 1.2 – Tính toán song song

- Tài nguyên ở đây là :
  - Một máy tính có nhiều bộ vi xử lý (processors)
  - Nhiều máy tính được kết nối với nhau
  - Hỗn hợp của 2 thành phần trên.
- Vậy nguyên nhân phải sử dụng tính toán song song:
  - Tiết kiệm thời gian và chi phí: Rõ ràng vấn đề được xử lý bằng tính toán song song sẽ nhanh hơn, và xây dựng một hệ thống song song sẽ rẻ hơn so với xây dựng nhiều hệ thống đơn lẻ.
  - Xử lý các vấn đề phức tạp: Xử lý ảnh 3D, dự báo thời tiết, …
  - Khả năng xử lý đồng thời : Có thể xử lý nhiều vấn đề cùng một lúc.
  - ….
- Xử lý song song liên quan trực tiếp đến: kiến trúc máy tính, phần mềm hệ thống (hệ điều hành), thuật toán và ngôn ngữ lập trình, v.v.
- Hệ thống tính song song: là một tập các BXL (thường là cùng một loại) kết nối với nhau theo một kiến trúc nào đó để có thể hợp tác với nhau trong hoạt động và trao đổi dữ liệu được với nhau.

II. Kiến trúc máy tính

Thành phần chính bao gồm :

- Bộ nhớ chính (main memmory)
- Đơn vị điều khiển (CU – Control unit)
- Đơn vị tính toán số học và logic (ALU - Arithmetic Logic Unit)
- Đơn vị vào/ra (Input / Output)



Hình 1.3 – Kiến trúc máy tính

Bộ nhớ chính là bộ nhớ truy cập ngẫu nhiên (RAM – Random access memory) được sử dụng để lưu dữ liệu (data) và các câu lệnh (instructions) của chương trình. Trong đó, các câu lệnh là dữ liệu được mã hóa (code) lệnh để cho CPU biết phải làm gì. Dữ liệu (data) là các thông tin được sử dụng bởi chương trình.

Khối CU có nhiệm vụ lấy dữ liệu hoặc câu lệnh từ bộ nhớ, sau đó giải mã các câu lệnh và thực hiện.

ALU thực hiện tính toán các phép số học.

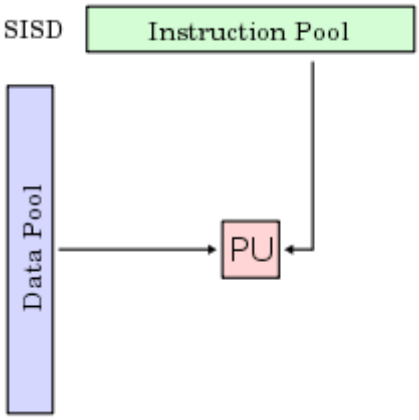
III. Phân loại máy tính song song

Theo Flynn phân loại máy tính song song thành 4 loại sau:

- SISD : Single Instruction, Single Data
- SIMD : Single Instruction, Multiple Data
- MISD : Multiple Instruction, Single Data
- MIMD : Multiple Instruction, Multiple Data

a. SISD - Single Instruction, Single Data

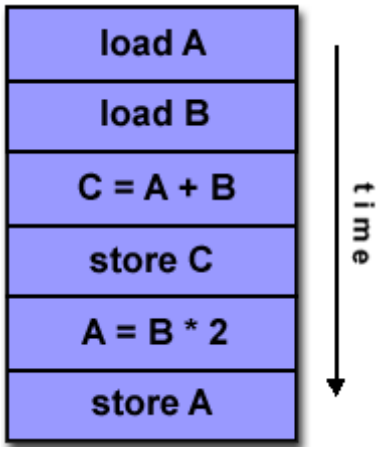
Máy tính loại SISD chỉ có một CPU, ở mỗi thời điểm thực hiện một lệnh và chỉ đọc, ghi một mục dữ liệu. Tất cả các máy tính SISD chỉ có một thanh ghi register được gọi là bộ đếm chương trình (program counter) được sử dụng để nạp địa chỉ của lệnh tiếp theo và kết quả là thực hiện theo một thứ tự xác định của các câu lệnh.



Hình 1.4 – Mô hình SISD

PU – Processing Unit

Mô hình SISD còn được gọi là SPSD (Simple Program Simple Data), đơn chương trình và đơn dữ liệu. Đây chính là mô hình máy tính kiểu von Neumann.

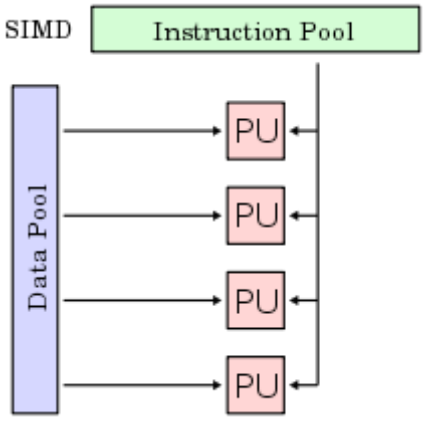


Hình 1.5 – Quá trình xử lý của một SISD

Ví dụ các máy kiểu SISD: Các máy mainframe cũ, các máy PC hiện nay.

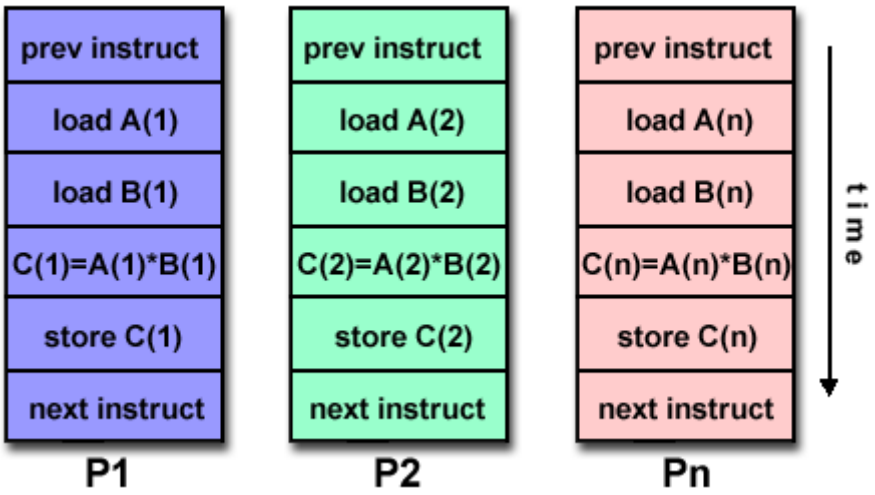
b. SIMD - Single Instruction, Multiple Data

Máy tính loại SIMD có một đơn vị điều khiển để điều khiển nhiều đơn vị xử lý (nhiều hơn một đơn vị) thực hiện theo một luồng các câu lệnh. CU phát sinh tín hiệu điều khiển tới tất cả các phần tử xử lý, những BXL này cùng thực hiện một phép toán trên các mục dữ liệu khác nhau, nghĩa là mỗi BXL có luồng dữ liệu riêng.



Hình 1.5 – Mô hình SIMD

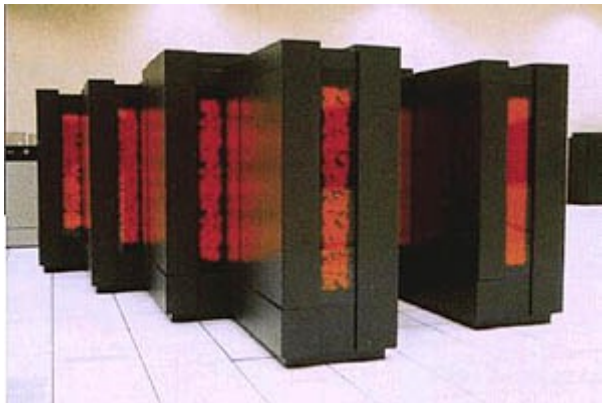
Mô hình SIMD còn được gọi là SPMD, đơn chương trình và đa dữ liệu



Hình 1.6 – Quá trình xử lý của một SIMD

Ví dụ về một số máy kiểu SIMD:

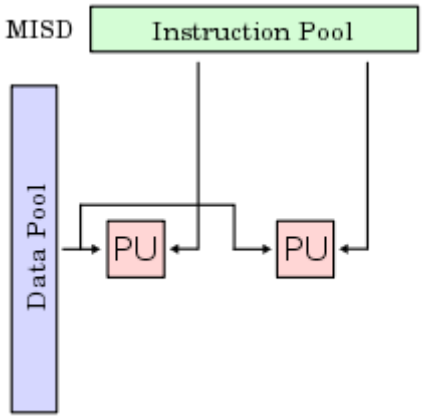
Hình 1.7 – Máy Cray X – MP



Hình 1.8 – Máy Thinking Machines CM-2

c. MISD

Máy tính loại MISD là ngược lại với SIMD. Máy tính MISD có thể thực hiện nhiều chương trình (nhiều lệnh) trên cùng một mục dữ liệu, nên còn được gọi là MPSD (đa chương trình, đơn dữ liệu).



Hình 1.9 – Mô hình MISD

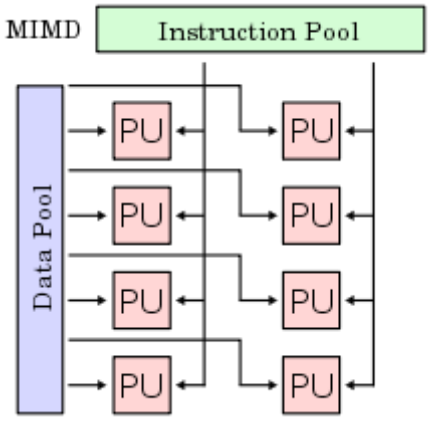
Trong thực tế, MISD rất khó sản xuất, theo một số tác giả, máy tính có kiến trúc của đường ống (piped-machine) là một kiểu của MISD.

d. MIMD – Multiple instruction multiple data

Máy tính loại MIMD còn gọi là đa BXL, trong đó mỗi BXL có thể thực hiện những luồng lệnh (chương trình) khác nhau trên các luồng dữ liệu riêng.

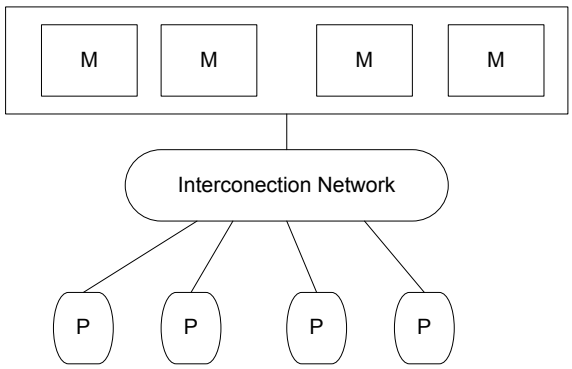
Hầu hết các hệ thống MIMD đều có bộ nhớ riêng và cũng có thể truy cập vào được bộ nhớ chung (global) khi cần, do vậy giảm thiểu được thời gian trao đổi dữ liệu giữa các BXL trong hệ thống.

Đây là kiến trúc phức tạp nhất, nhưng nó là mô hình hỗ trợ xử lý song song cao nhất và đã có nhiều máy tính được sản xuất theo kiến trúc này, ví dụ: BBN Butterfly, Alliant FX, iSPC của Intel, v.v.

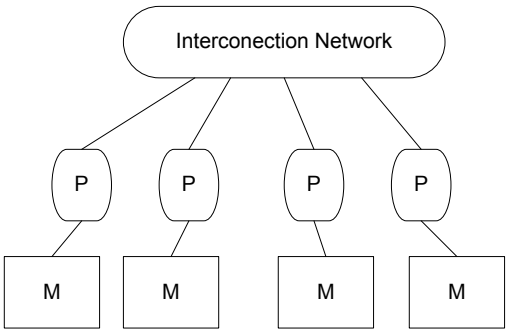


Hình 1.10 – Mô hình MIMD

MIMD được chia ra thành 2 loại : shared memory system và message passing system (hay distributed memory)



Hình 1.11 – Shared Memmory MIMD



Hình 1.12 – Message passing MIMD

Trong shared memory system, các bộ vi xử lý sẽ dùng chung một nhớ (ở đây là bộ nhớ cache), Các processor nối với bộ nhớ qua Interconection Network. Mỗi bộ vi xử lý sẽ có khả năng đọc/ ghi hay tốc độ truy xuất tới bộ nhớ là như nhau, vì vậy mà nó còn đc gọi là SMP (symmetric multiprocessor). Với mô hình này, giá thành sản xuất sẽ rất rẻ, và phát triển phần trên nó sẽ dễ hơn rất nhiều so với mô hình Message passing.

Trong mô hình Message passing, mỗi một bộ vi xử lý sẽ có một bố nhớ riêng (được gọi là node), các bộ vi xử lý được nối với nhau qua Interconection Network . Do không có bộ nhớ dùng chung, vì vậy muốn trao đổi dữ liệu giữa các nốt (node) sẽ dùng tới các bản tin (Message passing). Cụ thể là sẽ bằng một cặp bản tin Send/Receive, và sẽ được viết trong các của ứng dụng.

So sánh giữa hai mô hình, thì Message Passing MIMD hay Distributed – Memmory sẽ có khả năng mở rộng số lượng processor, Shared memory sẽ không thể, tuy nhiên shared memory systems có giá thành rẻ và phát triển ứng dụng trên nó dễ dàng hơn.

Chương II – Thiết kế thuật toán song song

Bước đầu tiên trong quá trình thiết kế chương trình song song là phải thật hiểu được vấn đề của bài toán. Nếu chương trình là tuần tự thì bắt buộc chúng ta phải hiểu toàn bộ code của chương trình. Khi bắt tay vào giải quyết bài toán, chúng ta phải xác định được phần nào có thể chuyển sang song song được, phần nào không thể chuyển được.

Ví dụ 1:

Tính tổng từ 1..100.

Giải quyết vấn đề :

Chúng ta có thể chia thành 2 phần : tính tổng từ 1..50, và 51..100.

Ví dụ 2:

Tính tổng dãy Fibonaci (1,1,2,3,5,8,13,21,...) bằng công thức

F(k + 2) = F(k + 1) + F(k)

Giải quyết vấn đề:

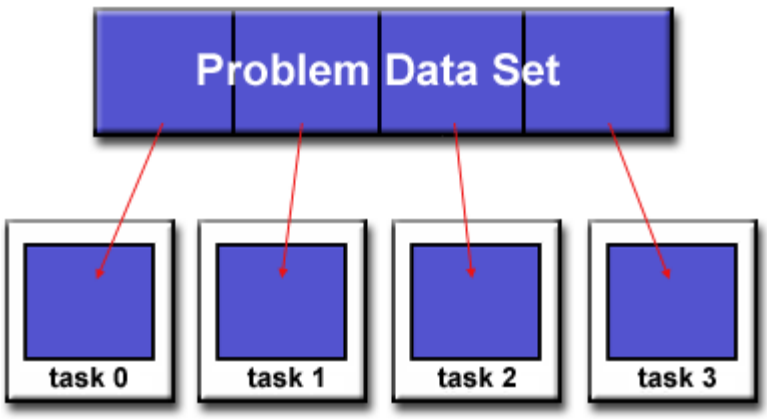
Bài toàn trên không thể chuyển sang song song được, vì theo công thức Fibonaci, muốn tính được số k + 2 thì cần biết k+1 và k, do vậy có sự lệ thuộc lẫn nhau, Chính vì thế không thể chuyển sang song song được.

I. Phân chia dữ liệu

Chúng ta sẽ chia dữ liệu thành các đoạn riêng biệt để các tác vụ làm việc với các dữ liệu đó. Đây được gọi là phân hủy hay phân chia dữ liệu.

Có hai loại phân chia là : phân chia theo miền và phân chia theo chức năng.

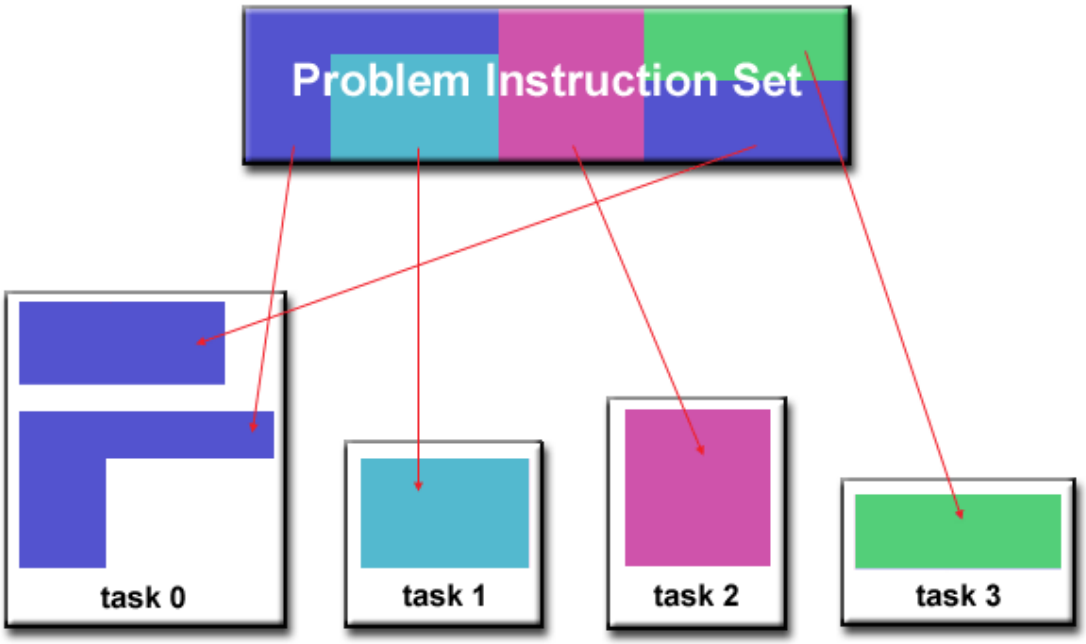
a. Phân chia theo miền



Hình 2.1 – Phân chia theo miền

Trong phần này, dữ liệu được phân chia thành các phần, mỗi phần ứng với một tác vụ nào đó.

b. Phân chia theo chức năng

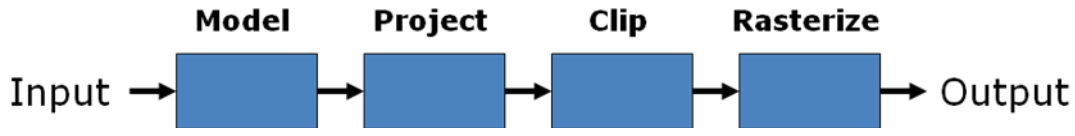


Hình 2.2 – Phân chia theo chức năng

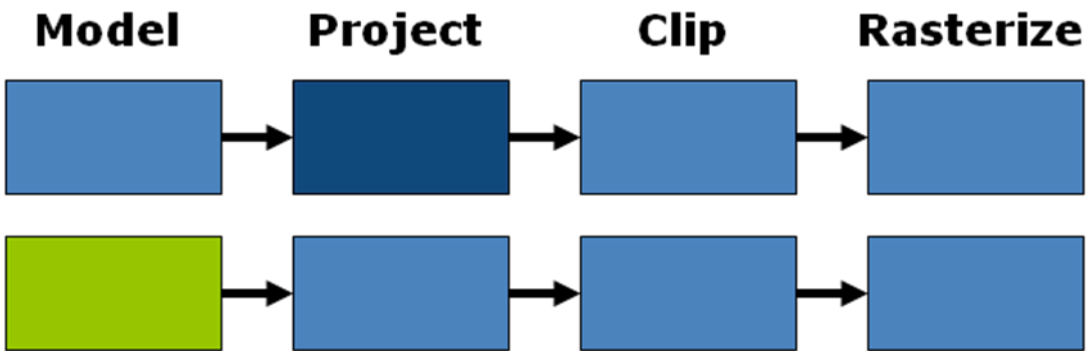
Nếu chương trình được chia thành nhiều chức năng (hàm) khác nhau, các hàm không có sự phụ thuộc lẫn nhau, chúng ta sẽ chia các hàm cho các tác vụ để thực hiện.

Đặc biệt, trong phân chia theo chức năng có một kiểu là đường ống (pipelening).

Ví dụ : Xử lý đồ họa 3D trong máy tính theo 4 bước sau

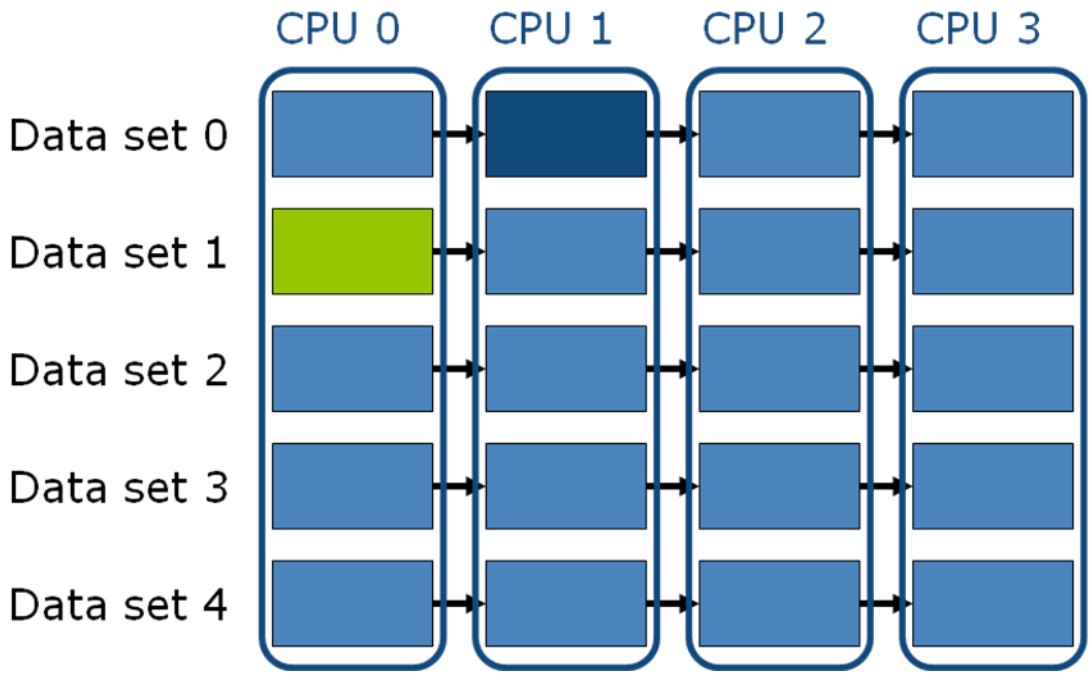


Nếu thực hiện tuần tự thì sẽ phải xử lý tuần tự theo từng bước một, nhưng nếu chuyển sang song song theo pipeling:



Tác vụ 1 sẽ xử lý 1 phần dữ liệu của Model, sau đó xử lý tiếp 1 phần dữ liệu của Project trong khi đó tác vụ 2 sẽ xử lý tiếp phần dữ liệu còn lại của Model.

Như hình 2.3, dữ liệu được thực hiện trong 4 bước, có 4 CPU tương đương với 4 tác vụ riêng biệt. Khi đó mỗi bước dữ liệu được chia thành 5 miền dữ liệu (data set 0 đến data set 4).



Hình 2.3 – Ví dụ về xử lý dữ liệu bằng pipeling

II. Giao tiếp giữa các tác vụ

Khi nào không cần giao tiếp giữa các tác vụ :

Khi mà các tác vụ đang chạy song song mà không cần chia sẻ dữ liệu.

Ví dụ : Trong quá trình xử lý ảnh, chương trình thực hiện việc chuyển

các điểm ảnh màu đen và trắng thành một màu khác. Dữ liệu của bức ảnh có thể phân chia theo miền để thực hiện song song, và các tác vụ song song này chạy độc lập với nhau.

Khi nào cần giao tiếp giữa các tác vụ :

Thực tế, phần lớn các tác vụ song song yêu cần trao đổi dữ liệu với nhau. Ví dụ : Trong quá trình xử lý ảnh 3D, khi tính toán một pixel nào đó, nó cần biết thông tin của các điểm ảnh bên cạnh, do vậy khi các tác vụ chạy song song, cần chia sẻ thông tin lẫn nhau.

Các tham số cần quan tâm khi cần giao tiếp giữa các tác vụ:

- Độ trễ và băng thông
- Đồng bộ và không đồng bộ
- Phạm vi của giao tiếp
- …

III. Đồng bộ

a. Barrier

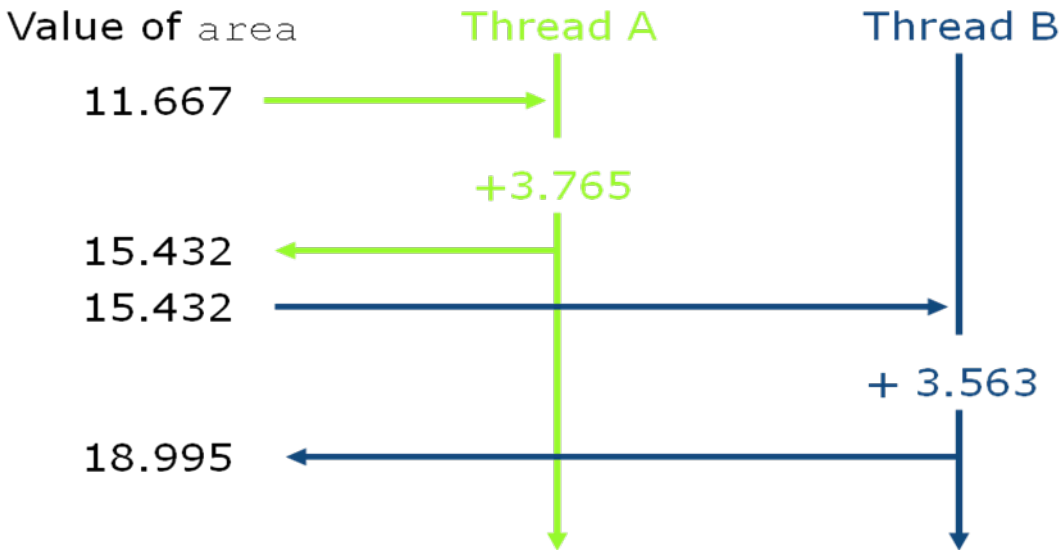
Barrier là phương pháp dùng để đồng bộ các tác vụ với nhau. Các tác vụ thực hiện nhiệm vụ của mình, khi gặp barrier sẽ dừng lại (block) , và đợi đến khi tác vụ cuối cùng gặp barrier.

Ví dụ 1:

```
double area, pi, x;
int i, n;
...
area = 0.0;
for (i = 0; i < n; i++) {
   x = (i + 0.5)/n;
   area += 4.0/(1.0 + x*x);
}
pi = area / n;
```

What happens when we make the `for` loop parallel?

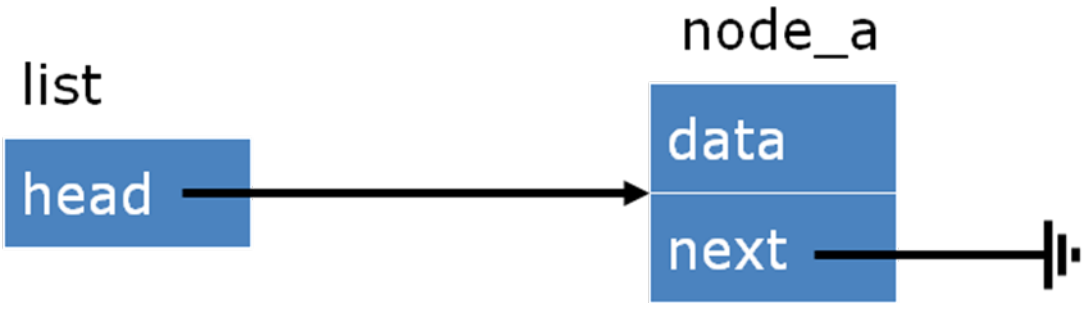
Với bài toán trên, khi phân chia song song, chúng ta sẽ dùng phân chia theo miền, tuy nhiên biến "area" sẽ bị tính toán sai.



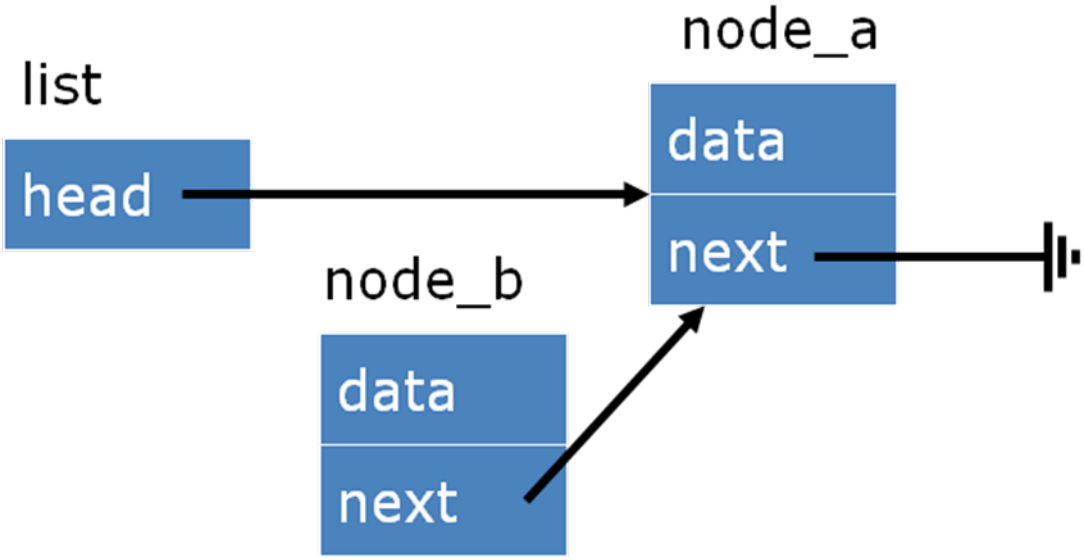
Ví dụ 2:

```
struct Node {
   struct Node *next;
   int data;   };

struct List {
   struct Node *head; }

void AddHead (struct List *list,
              struct Node *node) {
   node->next = list->head;
   list->head = node;
}
```

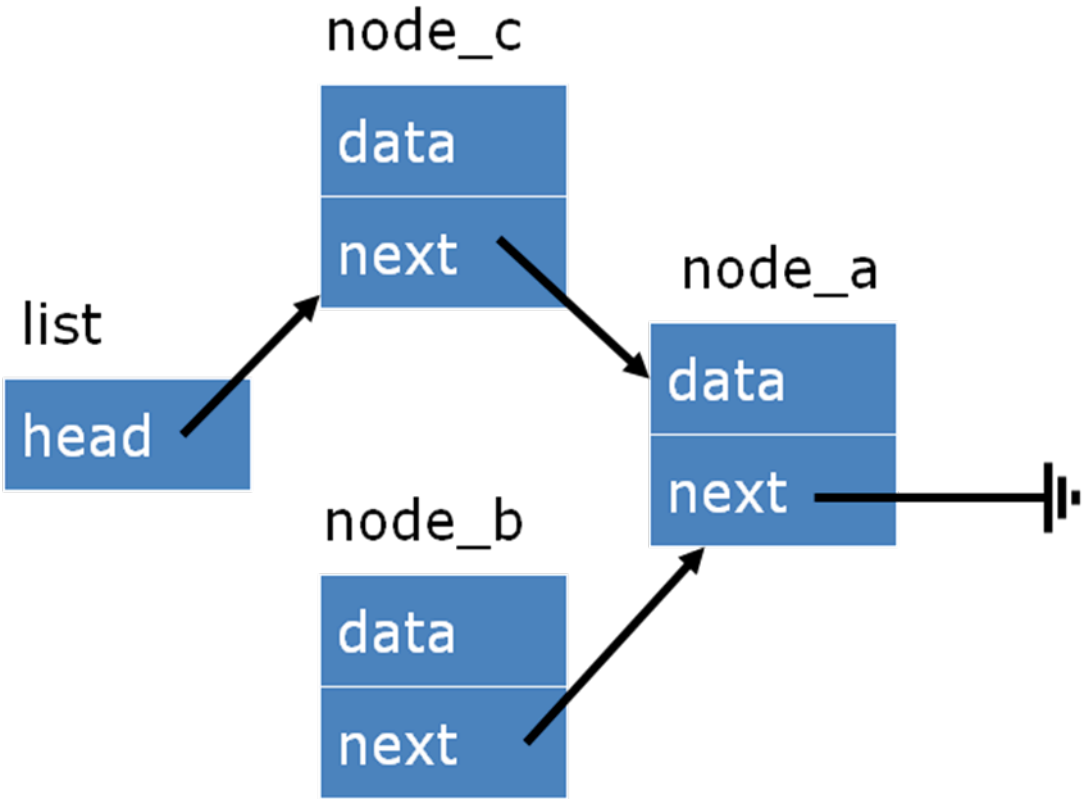
Giả sử ban đầu chúng ta có danh sách sau :



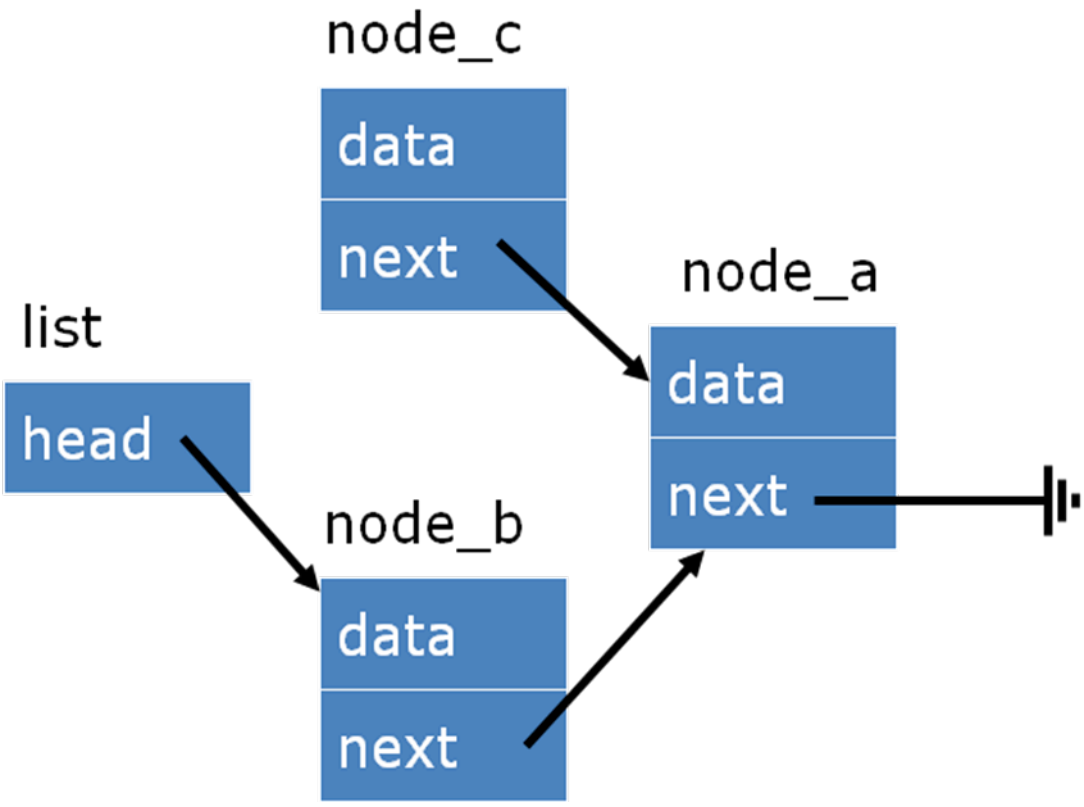
Tác vụ 1 chạy hàm AddHead :



Tác vụ 2 chạy hàm AddHead :



Tác vụ 1 sau khi chạy hàm AddHead :

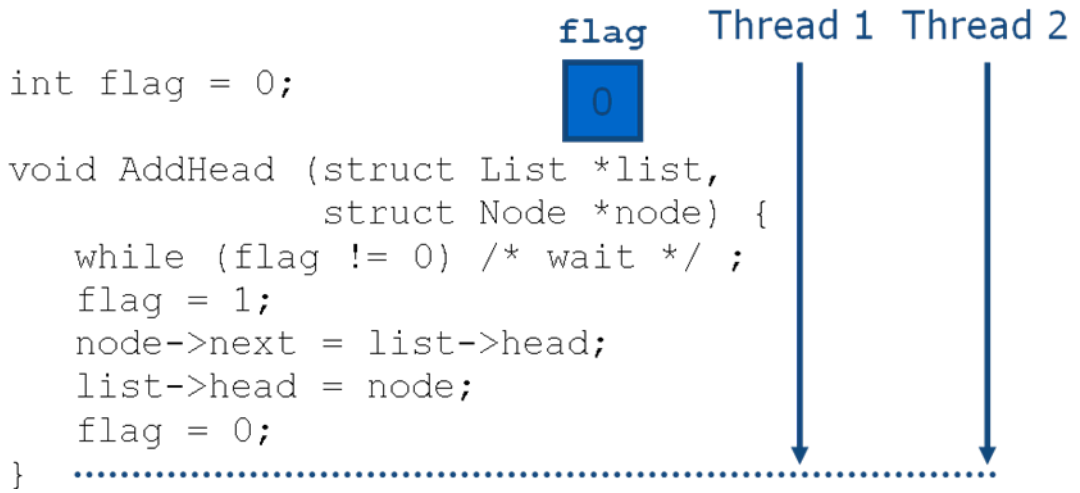


Như vậy chúng ta thấy, sau khi các tác vụ chạy xong hàm AddHead thì danh sách kết nối đã mất liên kết.

Giải quyết vấn đề:

```
int flag = 0;

void AddHead (struct List *list,
              struct Node *node) {
   while (flag != 0) /* wait */ ;
   flag = 1;
   node->next = list->head;
   list->head = node;
   flag = 0;
}
```

Chúng ta sử dụng 1 biến flag để đồng bộ các tác vụ với nhau :



b. Lock/Semaphore

Được dùng để bảo vệ vùng dữ liệu toàn bục hoặc một đoạn code nào đó, chỉ cho phép một tác vụ tác động lên mà thôi.

Khi tác vụ nào đó đến vùng dữ liệu hoặc đoạn code đó, "lock" sẽ được thiết lập, sau khi thực hiện xong, "lock" sẽ được bỏ. Trong quá trình thiết lập khóa, các tác vụ khác phải chờ.

IV. Phụ thuộc dữ liệu

Sự phụ thuộc là trong chương trình có sự quan hệ giữa các câu lệnh khi chúng ta thay đổi thứ tự các câu lệnh đó sẽ ảnh hưởng tới kết quả cuối cùng của chương trình.

Sự phụ thuộc dữ liệu là các nhiệm vụ khác nhau sử dụng cùng một vị trí lưu trữ nhiều lần.

Sự phụ thuộc rất quan trọng trong việc chuyển từ tuần tự sang song song.

Ví dụ :

```
For i:=1 to n do
     a[i+1] = a[i] *2;
```

Ở đây có sự phụ thuộc, muốn tính a[i+1] thì phải biết a[i] và a[i-1]. Vì vậy muốn chuyển sang song song là rất khó.

Nếu giả sử, tác vụ 2 có a[i] và tác vụ 1 có a[i-1], để tính toán giá trị chính xác của a[i] :

- Trong kiến trúc bộ nhớ phân tán: tác vụ 2 phải có giá trị a[i-1] của tác vụ 1 sau khi tác vụ 1 kết thúc.
- Trong kiến trúc bộ nhớ chia sẻ: Tác vụ 2 phải đọc giá trị a[i-1], sau khi tác vụ 1 cập nhật thành công giá trị đó.

Chú ý về việc xử lý phụ thuộc thuộc:

- Trong kiến trúc bộ nhớ phân tán: giao tiếp giữa các tác vụ cần phải đồng bộ.
- Trong kiến trúc bộ nhớ chia sẻ : Sự đồng bộ về đọc và ghi dữ liệu giữa các tác vụ.

V. Ví dụ

Ví dụ 1 : Nhân ma trận.

Hãy phân chia miền để thực hiện nhân ma trận 2x2

$$\begin{pmatrix} A_{1,1} & A_{1,2} \\ A_{2,1} & A_{2,2} \end{pmatrix} \cdot \begin{pmatrix} B_{1,1} & B_{1,2} \\ B_{2,1} & B_{2,2} \end{pmatrix} \rightarrow \begin{pmatrix} C_{1,1} & C_{1,2} \\ C_{2,1} & C_{2,2} \end{pmatrix}$$

Chúng ta có :

$$\begin{aligned} C_{1,1} &= A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ C_{1,2} &= A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ C_{2,1} &= A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ C_{2,2} &= A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Vậy nếu phân chia thành 4 tác vụ thì :

$$\begin{aligned} &\text{Task 1: } C_{1,1} = A_{1,1}B_{1,1} + A_{1,2}B_{2,1} \\ &\text{Task 2: } C_{1,2} = A_{1,1}B_{1,2} + A_{1,2}B_{2,2} \\ &\text{Task 3: } C_{2,1} = A_{2,1}B_{1,1} + A_{2,2}B_{2,1} \\ &\text{Task 4: } C_{2,2} = A_{2,1}B_{1,2} + A_{2,2}B_{2,2} \end{aligned}$$

Mỗi tác vụ sẽ thực hiện tính toán một $C_{i,j}$.

Ví dụ 2:

Cho dãy số : 4,2,7,8,5,1,3,6. Hãy sắp xếp tăng dần.

Ở đây ta thấy n = 8.

Chúng ta sẽ sử dụng thuật toán pipeling.

Giả thiết dữ liệu được lưu ở những tiến trình chẵn là B và ở những tiến trình lẻ là A. Thuật toán song song theo hình ống được mô tả trong MPI như sau :

Tiến trình chẵn	

Pi, i = 0, 2, 4, ..., n-2
recv(&A, Pi+1);
send(&B, Pi+1);
if(A > B) B = A;
	Pi, i = 1, 3, 5, ..., n-3
	send(&A, Pi-1);
	recv(&B, Pi-1);
	if(A > B) A = B;

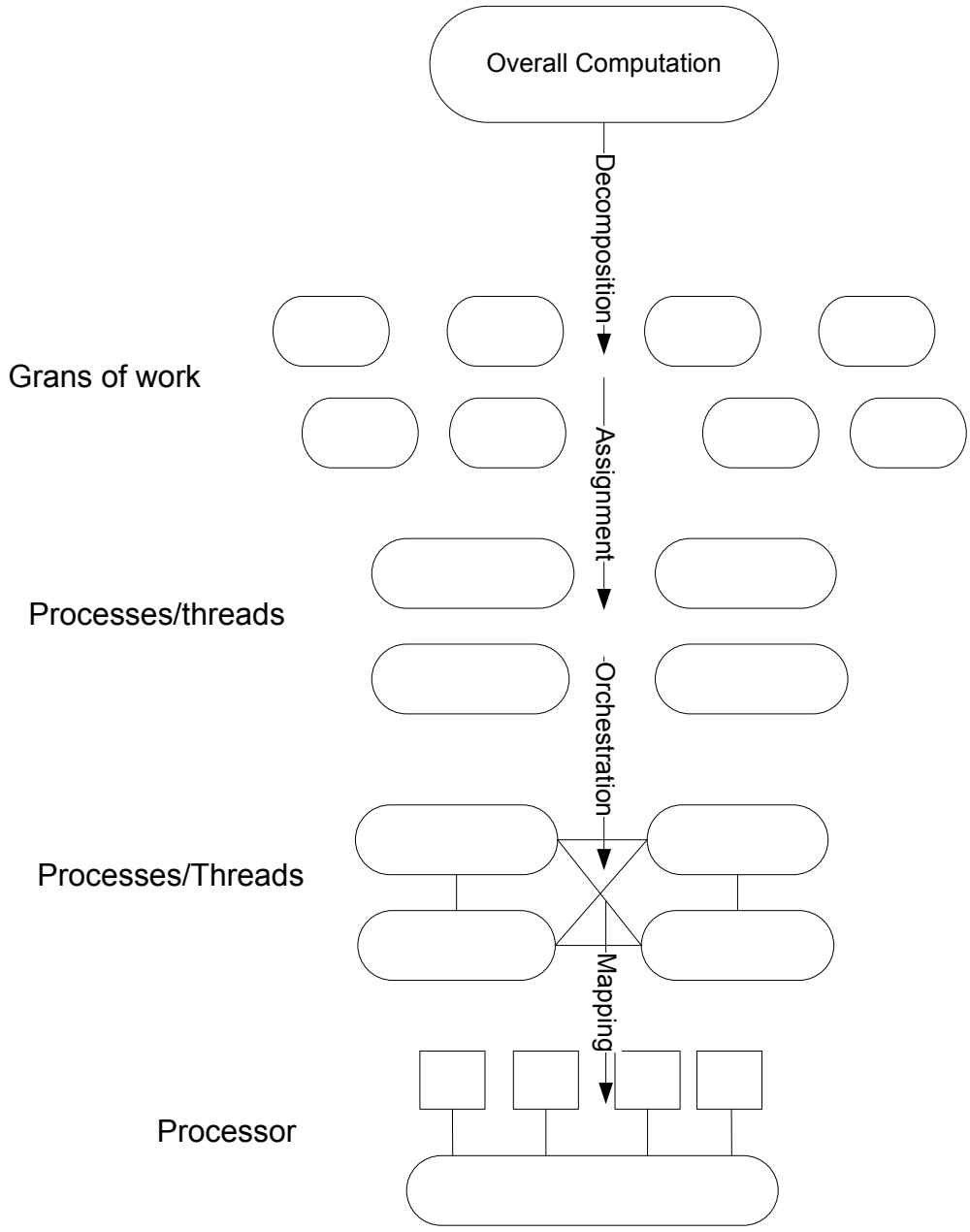
Tiến trình lẻ	

Pi, i = 1, 3, 5, ..., n-3
send(&A, Pi+1);
recv(&B, Pi+1);
if(A > B) A = B;
	Pi, i = 0, 2, 4, ..., n-2
	recv(&A, Pi-1);
	send(&B, Pi-1);
	if(A > B) B = A;

VI. Các bước tạo một chương trình song song

Để tạo môt chương trình song song chúng ta có 4 bước :

- Decomposition : Phân chia dữ liệu
- Assigment : Đưa dữ liệu vào tiến trình (process) hoặc luồng (Thread)
- Orchestration : Đồng bộ giữa các process hoặc thread với nhau
- Mapping : Đưa các process hoặc thread cho các vi xử lý



Hình 2.4 Các bước tạo một chương trình song song

a. Decompositition

Phân tích bài toán, phân chia bài toàn thành các vấn đề nhỏ.

Xác định các vấn đề nào có thể chạy song song hoặc phụ thuộc vào nhau.

Hãy sử dụng các phương pháp đã học để phân chia bài toán, phân chia dữ liệu.

b. Assignment

Xác định cơ chế để chúng ta chia bài toán cho các threads :

- Phân chia chức năng:

Chúng ta sẽ gán mỗi chức năng cho mỗi thread, tuy nhiên, chúng ta phải xác định sự phụ thuộc giữa các chức năng đó với nhau. Khi nói đến phân chia theo chức năng chúng ta sẽ liên tưởng tới phương pháp pipeling.

- Phân chia theo miền/ dữ liệu

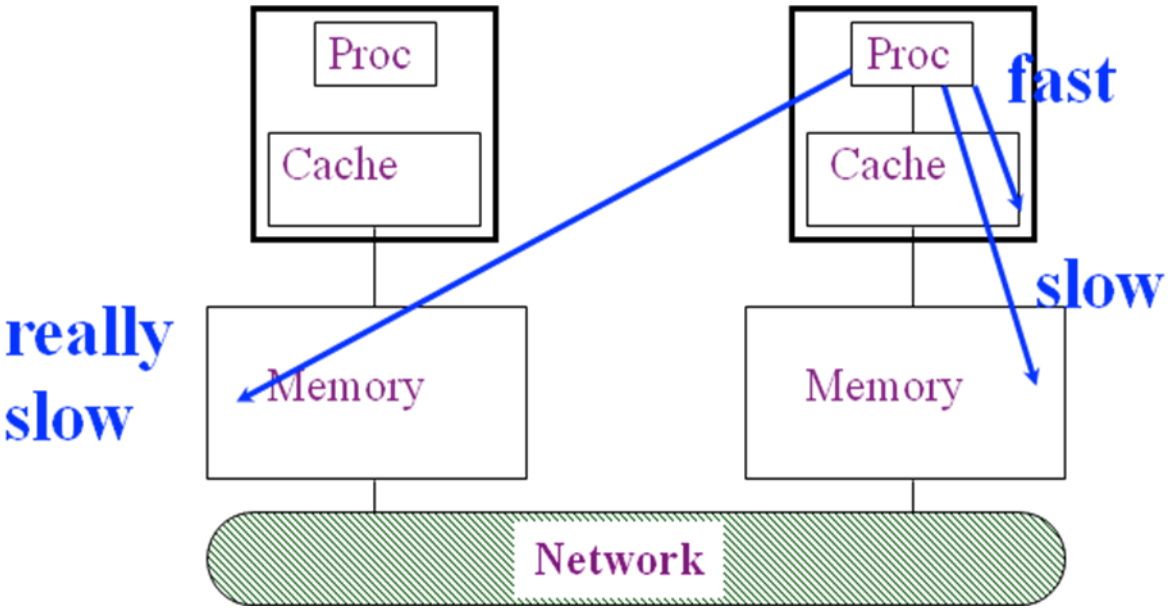
Phân chia theo miền/ dữ liệu là cách phân chia đơn giản và dễ hiểu nhất. Sau khi phân chia, chúng ta gán cho mỗi một miền một thread hoặc process để thực thi. Đơn giản nhất là trong các vòng lặp, vi xử lý j sẽ xử lý dữ liệu từ j*n/p đến (j+1)*n/p-1. Chúng ta sẽ xem kỹ hơn trong ví dụ.

c. Orchestration

Các Threads/processes thường hay chia sẻ dữ liệu, đồng bộ với nhau, vì vậy chúng ta phải điều phối hoạt động của chúng, bởi vì các dữ liệu thường có liên quan đến nhau. Chúng ta phải phân tích kỹ từ bước tìm hiểu bài toán, bởi vì không phải là chuyển sang song song sẽ tốt hơn là tuần tự. Ở đây chúng ta sẽ dùng các phương pháp đồng bộ để điều phối hoạt động của các Threads/Processes

d. Mapping

Đây là bước gán Threads/Processes vào các vi xử lý. Ở đây chúng ta quan tâm là có bao nhiêu Threads/Processes, việc truy cập tới ô nhớ chia sẻ. Các vi xử lý truy cập tới bộ nhớ cache (private memmory) rất nhanh, nhưng khi bộ nhớ khác qua kênh network sẽ rất chậm.



Hình 2.5 – Tốc độ VXL truy cập các bộ nhớ

Thông thường trong quá trình lập trình song song thì chúng ta không hay quan tâm tới việc các threads/processes sẽ do vi xử lý nào đảm nhận, việc phân cho VXL là của hệ điều hành đảm nhận.

e. Ví dụ

Tính tổng sau : S = f(A[1]) + … + f(A[n])

- Decomposition
  - Nếu số VXL p > n thì mỗi vi xử lý sẽ xử lý mỗi f(A[j]) →tính tổng S
  - Nếu số VXL p < n → phân chia theo miền/dữ liệu.
- Assignment
  - thread k tính : Sk = f(A[k*n/p]) + … + f(A[(k+1)*n/p-1])
  - thread 1 tính : S = s1+ … + sp
  - Các thread chia sẻ s với các thread khác.
- Orchestration
  - Cho các Threads chạy
  - Thực hiện đồng bộ các Threads với nhau để truy cập vào biến S
- Mapping
  - VXL j sẽ chạy thread j

Chương 3 : Các mô hình lập trình song song

I. Giới thiệu

Như chúng ta đã biết, có 2 loại kiến trúc máy tính song song theo bộ nhớ :

- Bộ nhớ phân tán (distributed memory)
- Bộ nhớ chia sẻ (Shared memmory)

Các công cụ lập trình tương ứng

	Shared memmory	Distributed Memmory
Công cụ hệ thống	Threads (Pthread)	Socket
Công cụ chuyên biệt	OpenMP	MPI

	Pthread	PVM Globus Toolkit 4

II. Lập trình chia sẻ bộ nhớ dựa vào tiến trình

Yêu cầu là bài toán phải thiết kế thành các tiến trình để có thể xử lý song song, tuy nhiên khi các tiến trình kết thúc, nó sẽ tự động giải phóng bộ nhớ và các thiết bị mà nó đã chiếm giữ, vì vậy khi hủy bỏ phải đảm bảo không ảnh hưởng tới các tiến trình khác.

Tạo N tiến trình :

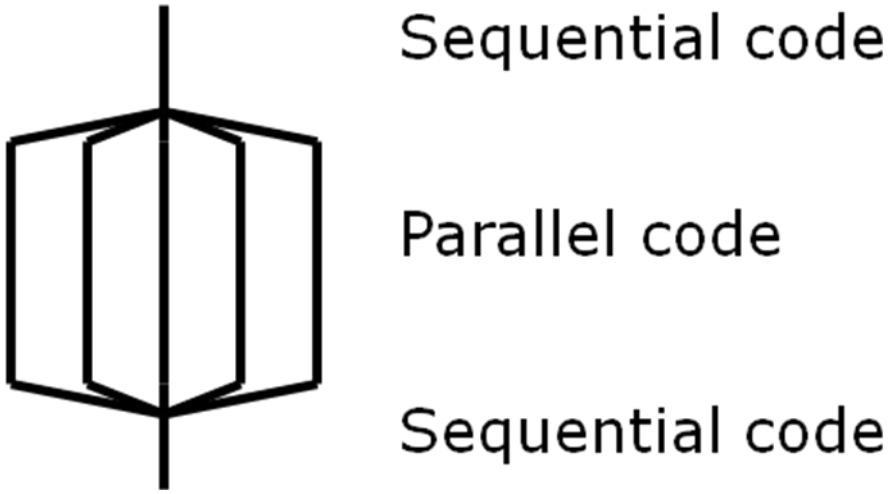
id = create_process(N);

Khi đó sẽ có N+1 tiến trình được tạo ra (1 tiến trình chủ - main());

Phân công nhiệm vụ :

```
 id = create_process(N);
switch(id)
{
     Case 1 : do  Task1 .. ; break;
     Case 2 : do Task2….; break;
     ….
     Case N : do TaskN…;break;
}
join_process(N,0);
```

Câu lệnh join_process(N,0), khi đó các process sẽ phải chờ nhau, đến khi process cuối cùng kết thúc thì process chính mới thực hiện câu lệnh tiếp theo.



Hình 3.1 – Mô hình xử lý song song theo tiến trình

Ví dụ: Cho trước một đoạn chương trình sau và chuyển sang song song:

```
for(i = 0; i < N; i++){
C[i] = A[i] + B[i];
}
```

Giả sử ta có M tiến trình. Chúng ta có thể chia N phần tử thành M phần (thường ta giả thiết N chia hết cho M) và gán từng phần đó cho mỗi tiến trình. Chu trình trên có thể viết thành:

```
for(j = id * N/M; j < (id+1)*N/M; j++){
C[j] = A[j] + B[j];
}
```

Trong đó, *id* là số hiệu của tiến trình, chạy từ 0 đến M-1. Tiến trình thứ *i* xử lý N/M phần tử liên tiếp kể từ *i*N/M+1*..*

Hoặc ta có thể cho phép các tiến trình truy cập xen kẽ vào các phần tử của mảng như sau:

Tiến trình **Pi** bắt đầu từ phần tử thứ *i,* sau đó bỏ qua M phần tử để xử lý phần tử tiếp theo, nghĩa là nó truy cập đến i, i+M, i+2M, v.v. Khi đó được viết như sau:

```
for(j = id; j < N; j+=M){
  C[j] = A[j] + B[j];}
```

III. Lập trình Thread

Các luồng của một tiến trình có thể chia sẻ với nhau về không gian địa chỉ chương trình, các đoạn dữ liệu và môi trường xử lý, đồng thời cũng có vùng dữ liệu riêng để thao tác.

Các tiến trình và các luồng trong hệ thống song song cần phải được đồng bộ, song việc *đồng bộ giữa các luồng được thực hiện hiệu quả hơn đối với các tiến trình.* Đồng bộ giữa các tiến trình đòi hỏi tốn thời gian hoạt động của hệ thống, trong khi đối với các luồng thì việc đồng bộ chủ yếu tập trung vào sự truy cập các biến chung (global) của chương trình.

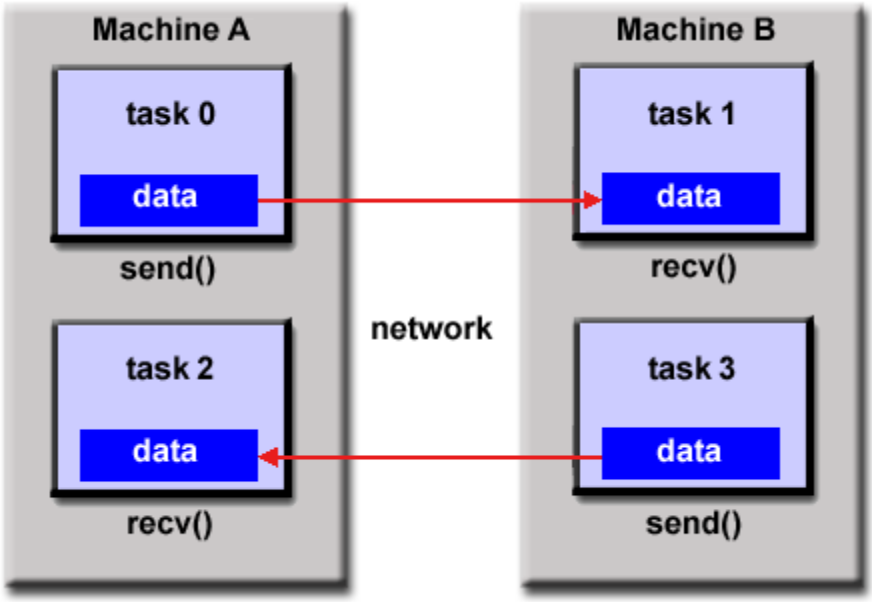
Nhiều hệ điều hành hiện nay hỗ trợ đa luồng như: SUN Solaris, Window NT, Windows 2000, OS/2, v.v. Hiện nay đã có một chuẩn cho việc lập trình song song dựa trên các luồng đó là *Pthread của IEEE Portable Operating System Interface, POSIX.*

IV.Lập trình Message passing trong Distributed Memmory

Mô hình Message Passing

Mô hình Message Passing có những đặc tính sau:

- Mô hình Message Passing được dành cho kiến trúc chia sẻ bộ nhớ, nên các máy tính có những bộ nhớ riêng, các tác vụ nằm trên một máy tính chỉ có thể truy cập vào bộ nhớ riêng này.
- Các tác vụ trên hai máy tính khác nhau muốn chia sẻ dữ liệu sẽ sử dụng : send message và receive message.
- Khi sử dụng hàm send và receive phải phù hợp với nhau.



Hình 3.2 – Mô hình lập trình Message Passing

Lập trình MP cũng giống như lập trình mạng, khi các các tác vụ giao tiếp với nhau bằng message, khi được nhận nó sẽ được lưu trữ trong bộ đệm, khác với trong kiến trúc Shared Memmory là chứa trong hàng đợi.

Lập trình MP khó hơn lập trình theo luồng hay tiến trình trong Shared Memmory. Và đặc biệt là khi một chương trình được viết bằng MP có thể chạy tốt trên kiến trúc Shared Memmory, và nó có thể chạy rất nhanh.

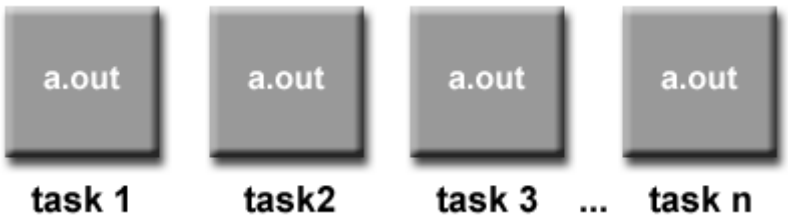
V. Các mô hình khác

a. Hybrid

Trong mô hình này, có nhiều hơn hai mô hình lập trình gộp lại. Thường thì mô hình hybrid sẽ kết hợp mô hình Message Passing với mô hình Thread hoặc Mô hình Shared Memmory(Open MP). Mô hình này sẽ cải thiện môi trường phần cứng cho mo hình Shared Memmory.

b. Single Program Multiple Data(SPMD)

Mô hình này là mô hình lập trình bậc cao mà nó được xây dựng dựa trên các mô hình đã có.



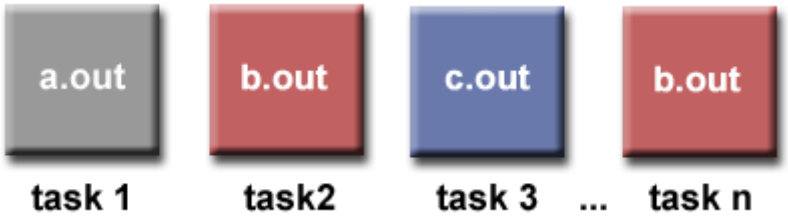
Mô hình SPMD

Một chương trình có thể được thực hiện bởi đồng thời ác tác vụ. Tại bất kỳ một thời điểm nào, các tác vụ có thể thực thi cùng cùng một câu lệnh hoặc các câu lệnh khác nhau nhưng của cùng một chương trình.

Các nhiệm vụ có thể sử dụng dữ liệu khác nhau.

c. Multiple Program Multiple Data (MPMD)

Giống SPMD, MPMD cũng là một mô hình lập trình bậc cao.



Mô hình MPMD

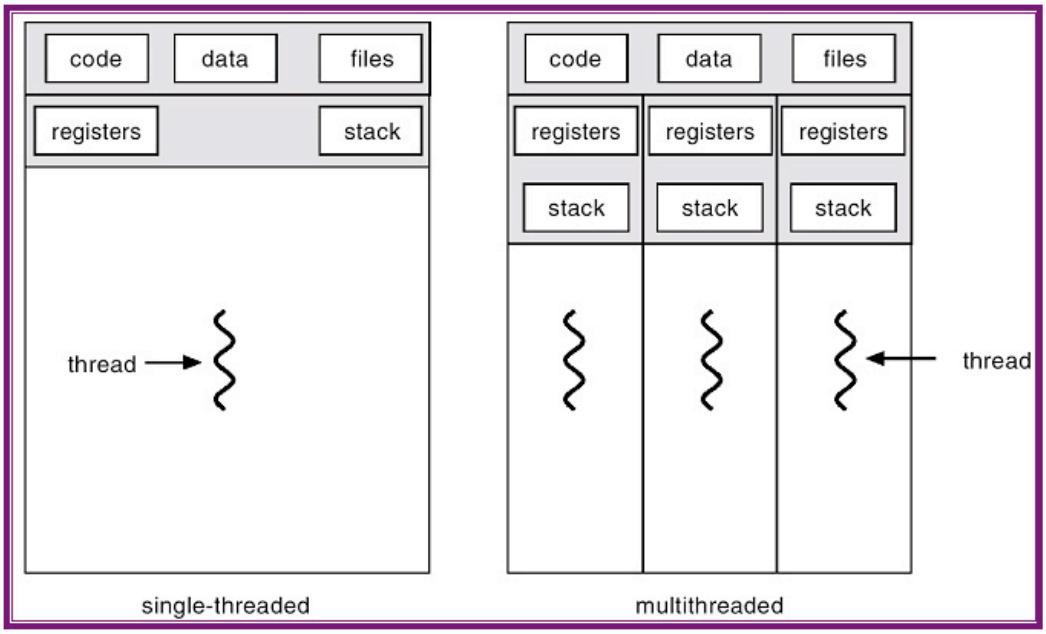
Ứng dụng MPMD chia thành nhiều đối tượng có thể thực thi được, khi ứng dụng đang chạy song song, mỗi tác vụ có thể thực thi cùng đối tượng hoặc khác đối tượng với tác vụ khác. Và mỗi tác vụ có thể sử dụng dữ liệu khác nhau.

Chương IV - PTHREAD POSIX (Portable Operating System Interface)

I. Giới thiệu về Thread

Theo định nghĩa thì một thread là một luồng các câu lệnh độc lập được sắp lịch sẵn để được thực thi bởi hệ điều hành. Thread là một bán tiến trình, nó có stack, code để chạy. Không giống như process, thread có thể chia sẻ bộ nhớ cho thread khác.

Trong một Process có thể có nhiều Thread, chúng chia sẻ cùng một bộ nhớ, và truy cập vào biến toàn cục, bộ nhớ heap, và có cùng một miêu tả chung. Các Thread có thể chạy song song.



Hình 4.1 – Miêu tả Thread trong Process

Vậy lợi ích của Thread là :

- Trong ứng dụng được lập trình theo thread, thì ứng dụng có thể chạy được ở cả máy tính tuần tự và máy song song.
- Vấn đề chính của các chương trình là độ trễ khi truy cập vào I/O, bộ nhớ chính, và giao tiếp giữa các chương trình. Nếu chương trình lập trình theo thread, thì rõ ràng vấn đề độ trễ sẽ không còn.
- Lập trình theo process (tiến trình) thì vấn đề overhead luôn làm các nhà lập trình đau đầu, tuy nhiên nếu lập trình theo thread thì vấn đề overhead không cần quan tâm nhiều lắm.
- Dễ dàng lập trình và debug.

II. Cơ bản về Pthread

a. Tạo và kết thúc thread

Prototype của hàm tạo thread :

```
#include <pthread.h>
   int
   pthread_create (
       pthread_t   *thread_handle,
       const pthread_attr_t   *attribute,
       void *   (*thread_function)(void *),
    void    *arg);
```

thread: Một định danh ID duy nhất cho thread mới

attribute: Lưu trữ các tham số của thread mới, mặc định là Null

thread_function: Địa chỉ của chương trình con mà chúng ra cần chạy trong thread mới này

arg: Các tham số của chương trình con, mặc định là NULL

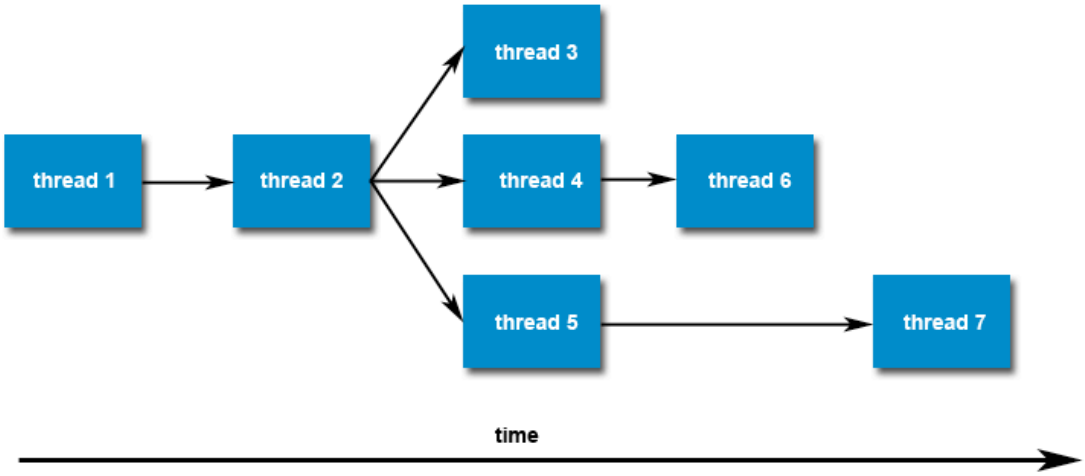
- Kết thúc thread :

```
pthread_exit( void *ptr);
```

- Gộp các thread:

```
int pthread_join (
    pthread_t thread,void **ptr);
```

Chú ý rằng, trong các thread có thể tạo thêm các thread khác.



Hình 4.2 – Ví dụ về tạo thread trong chương trình

```
#include <stdio.h>        // standard I/O
#include <pthread.h>      //các hàm pthread
void* PrintHello(void* data)
{
    int my_data = (int)data;        // Nhận giá trị từ thread
    pthread_detach(pthread_self());
    printf("Hello from new thread - got %d\n", my_data);
    pthread_exit(NULL);     /* kết thúc thread */
}
int main(int argc, char* argv[])
{
    int        rc;          // giá trị trả về khi tạo thread
    pthread_t  thread_id;          /* thread's ID */
    int        t         = 11;  // truyền dữ liệu vào thread
    /* tạo thread và in ra chữ 'PrintHello' */
    rc = pthread_create(&thread_id, NULL, PrintHello,
(void*)t);
    if(rc)                  /* nếu tạo thread lỗi */
    {
        printf("\n ERROR: return code from pthread_create is
%d \n", rc);
        exit(1);    }
    printf("\n Created new thread (%d) ... \n", thread_id);

    pthread_exit(NULL);            /* kết thúc thread */
}
```

Trong chương trình trên, hàm main() có khai báo biến thread_id với kiểu là pthread_t. Đây là một số nguyên để định danh trong hệ thống. Khi chúng ta gọi hàm pthreate_create(), thread sẽ được tạo ra. Chú ý rằng, tham số thứ 2 trong hàm pthread_create(), tham số này được sử dụng để thiết lập các thuộc tính cho thread, chúng ta truyền vào là một con trỏ NULL, đây là giá trị mặc định, vậy khi đó, nếu hàm pthread_create() tạo thread thành công thì giá trị trả về sẽ là 0.

Khi thread được tạo thành công, chương trình sẽ có 2 thread, thread chính là hàm main(). Hai thread này sẽ chạy song song với nhau.

a. Đồng bộ giữa các thread

Giả sử chúng ta có hai thread làm việc song song với nhau, và các thread này làm việc trên cùng một dữ liệu, và chúng có tính toán và làm thay đổi dữ liệu đó, khi đó nếu chúng ta không đồng bộ giữa các thread này thì rất dễ làm dử liệu bị sai.

Chúng ta xét ví dụ sau :

```
// Mỗi thread sẽ cập nhật giá trị best_cost
   if (my_cost < best_cost)
        best_cost = my_cost;
```

Giả sử Thread 1 có giá trị my_cost = 75, thread 2 có my_cost = 50, và best_cost là dữ liệu chung và có giá trị = 100.

Vậy chúng ta mong muốn giá trị best_cost = 50, tuy nhiên nếu thread 1 thực hiện lệnh gán sau cùng thì giá trị best_cost = 75.

Trong API lập trình thread có cung cấp cho chúng ta đồng bộ giữa các thread bằng ***mutex-locks***.

Mutex-locks có hai trạng thái : locked và unlocked. Tại bất cứ một thời điểm nào chỉ cho phép một thread có thể khóa mutex-locks.

Trong API có một số hàm để xử lý mutex-lock. Hàm pthread_mutex_lock sử dụng để khóa mutex-lock. Prototype của hàm pthread_mutex_lock :

```
int
   pthread_mutex_lock (
       pthread_mutex_t *mutex_lock);
```

Nếu hàm được gọi thành công thì giá trị trả về là 0, ngược lại có lỗi xảy ra (xảy ra trong trường hợp deadlocks).

Khi hàm được gọi nó sẽ thiết lập khóa cho mutex_lock. Nếu mutex-lock đã khóa trước đó thì thread gọi hàm sẽ bị khóa và chờ mutex-lock mở khóa, ngược lại mutex-lock sẽ bị khóa thành công.

Trong trường hợp muốn mở khóa mutex-lock, chúng ta sẽ dùng hàm pthread_mutex_unlock. Prototype của hàm :

```
int
   pthread_mutex_unlock (
       pthread_mutex_t *mutex_lock);
```

Tuy nhiên chúng ta cần khởi tạo một mutex-lock, trong API có hàm khởi tạo là pthread_mutex_init, mặc định mutex-lock khởi tạo là trạng thái unlocked.

```
int
   pthread_mutex_init (
       pthread_mutex_t *mutex_lock,
       const pthread_mutexattr_t *lock_attr);
```

Ví dụ về mutex-lock:

Tìm số nhỏ nhất trong một danh sách các số nguyên

```
#include <pthread.h>
   void *find_min(void *list_ptr);
   pthread_mutex_t minimum_value_lock;
   int minimum_value, partial_list_size;
   main() {
       minimum_value = MIN_INT;
       pthread_init();
      //Khởi tạo khóa
      pthread_mutex_init(&minimum_value_lock, NULL);


  }
  void *find_min(void *list_ptr) {
      int *partial_list_pointer, my_min, i;
      my_min = MIN_INT;
      partial_list_pointer = (int *) list_ptr;
      for (i = 0; i < partial_list_size; i++)
          if (partial_list_pointer[i] < my_min)
              my_min = partial_list_pointer[i];

      /* Thiết lập trạng thái locked vì có sự thay đổi giá trị với
biến dùng chung*/
      pthread_mutex_lock(&minimum_value_lock);
      if (my_min < minimum_value)
          minimum_value = my_min;
      /* Mở khóa */
      pthread_mutex_unlock(&minimum_value_lock);
      pthread_exit(0);
  }
```

Trong ví dụ trên, biến minimum_value được bảo vệ bởi khóa minimum_value_lock. Khi đó, các thread sẽ phải thực thi pthread_mutex_lock để có thể truy cập vào biến minimum_value, tại một thời điểm, chỉ có một thread giữ khóa thì mới có thể truy cập vào biến dùng chung.

b. Các thuộc tính của thread

Hàm pthread_attr_init cho phép chúng ta tạo các thuộc tính cho thread.

```
int
   pthread_attr_init (
    pthread_attr_t *attr);
```

Hàm này sẽ tạo các thuộc tính cho atrr các giá trị mặc định. Nếu thực hiện thành công hàm trả về giá trị 0. Muốn xóa biến attr chúng ta dùng hàm pthread_atrr_destroy :

```
int
   pthread_attr_destroy (
    pthread_attr_t *attr);
```

Hàm trả về 0 nếu thực hiện thành công.

Ngoài ra, nếu chúng ta muốn thiết lập các thuộc tính riêng lẻ, APT cung cấp cho chúng ta các hàm sau : pthread_attr_setdetachstate, pthread_attr_setguardsize_np, pthread_attr_setstacksize, pthread_attr_setinheritsched, pthread_attr_setschedpolicy, và pthread_attr_setschedparam.

c. Normal-Mutex

Trong Pthread API cung cấp cho chúng ta 3 loại khóa. Tất cả các khóa đều có hai chức năng là khóa (locking) và mở khóa (unlocking). Một loại mutex-lock là normal-mutex. Loại này chỉ cho phép một thread được phép khóa tại bất kỳ một thời điểm nào đang chạy. Tuy nhiên nếu thread này mà gọi tới hàm khóa lần thứ 2 thì sẽ bị deadlock.

Ví dụ :

Tìm phần tử trong cây nhị phân. Để đảm bảo các thread không làm thay đổi cây nhị phân, trong quá trình tìm kiếm chúng ta sử dụng hàm tree_lock để khóa.

```
search_tree(void *tree_ptr)
{
       struct node *node_pointer;
       node_pointer = (struct node *) tree_ptr;
       pthread_mutex_lock(&tree_lock);
       if (is_search_node(node_pointer) == 1) {

           print_node(node_pointer);
           pthread_mutex_unlock(&tree_lock);
          return(1);
      }
      else {
          if (tree_ptr -> left != NULL)
              search_tree((void *) tree_ptr -> left);
          if (tree_ptr -> right != NULL)
              search_tree((void *) tree_ptr -> right);
      }
      printf("Search unsuccessful\n");
      pthread_mutex_unlock(&tree_lock);
 }
```

d. Tháo gỡ threads – Detach threads

Trong Pthread API cung cấp hàm pthread_detach dùng để giải phóng bộ nhớ của thread khi nó kết thúc.

int pthread_detach(thread_t tid);

Ví dụ :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
double X[10][10];
double Y[10][10];
struct DATA
{
      double x;
      int i;
      int j;
} ;
void *thread_f(void *arg)
{
      DATA* a = (DATA*)arg;
      X[a->i][a->j] = f(a->x);
      return NULL;
}
void main()
{
      Int rc;
      pthread_t thread;
      for (int i=0;i<10; ++i)
      {
            for (int j=0; j<10; ++j)
            {
                  Y[i][j]=i+j;
            }
      }
      DATA *arg;
      for ( i=0;i<10; ++i)
      {
            for (int j=0; j<10; ++j)
            {
                  arg = new DATA;
                  arg->i = i; arg->j = j; arg->x = Y[i][j];
                  pthread_create(&thread, NULL, thread_f,
            (void *)arg);
            // chia thành các thread
            // giải phóng thread
            //rc = pthread_detach(thread);
            //printf ("Trang thai % d \n",rc);
            }
      }
}
```

e. Các ví dụ

Ví dụ về truyền tham số khi tạo ra thread :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUM_THREADS    8
char *messages[NUM_THREADS];
struct thread_data
{
   int  thread_id;
   int  sum;
   char *message;
};
struct thread_data thread_data_array[NUM_THREADS];
void *PrintHello(void *threadarg)
{
   int taskid, sum;
   char *hello_msg;
   struct thread_data *my_data;

   sleep(1);
   my_data = (struct thread_data *) threadarg;
   taskid = my_data->thread_id;
   sum = my_data->sum;
   hello_msg = my_data->message;
   printf("Thread %d: %s  Sum=%d\n", taskid, hello_msg, sum);
   pthread_exit(NULL);
}
int main(int argc, char *argv[])
{
pthread_t threads[NUM_THREADS];
int *taskids[NUM_THREADS];
int rc, t, sum;

sum=0;
messages[0] = "English: Hello World!";
messages[1] = "French: Bonjour, le monde!";
messages[2] = "Spanish: Hola al mundo";
messages[3] = "Klingon: Nuq neH!";
messages[4] = "German: Guten Tag, Welt!";
messages[5] = "Russian: Zdravstvytye, mir!";
messages[6] = "Japan: Sekai e konnichiwa!";
messages[7] = "Latin: Orbis, te saluto!";

for(t=0;t<NUM_THREADS;t++) {
  sum = sum + t;
  thread_data_array[t].thread_id = t;
  thread_data_array[t].sum = sum;
  thread_data_array[t].message = messages[t];
  printf("Creating thread %d\n", t);
  rc = pthread_create(&threads[t], NULL, PrintHello, (void *)
       &thread_data_array[t]);
  if (rc) {
    printf("ERROR; return code from pthread_create() is %d\n",
rc);
    exit(-1);
    }
  }
pthread_exit(NULL);
}
```

Kết quả :

```
Creating thread 0
Creating thread 1
Creating thread 2
Creating thread 3
Creating thread 4
Creating thread 5
Creating thread 6
Creating thread 7
Thread 0: English: Hello World!  Sum=0
Thread 1: French: Bonjour, le monde!  Sum=1
Thread 2: Spanish: Hola al mundo  Sum=3
Thread 3: Klingon: Nuq neH!  Sum=6
Thread 4: German: Guten Tag, Welt!  Sum=10
Thread 5: Russian: Zdravstvytye, mir!  Sum=15
Thread 6: Japan: Sekai e konnichiwa!  Sum=21
Thread 7: Latin: Orbis, te saluto!  Sum=28
```

Ví dụ về sử dụng mutex-lock :

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
typedef struct
 {
   double      *a;
   double      *b;
   double     sum;
   int     veclen;
 } DOTDATA;
/* Define globally accessible variables and a mutex */
#define NUMTHRDS 4
#define VECLEN 100
   DOTDATA dotstr;
   pthread_t callThd[NUMTHRDS];
   pthread_mutex_t mutexsum;
void *dotprod(void *arg)
{
/* Define and use local variables for convenience */
   int i, start, end, len ;
   long offset;
   double mysum, *x, *y;
   offset = (long)arg;

   len = dotstr.veclen;
   start = offset*len;
   end   = start + len;
   x = dotstr.a;
   y = dotstr.b;
   mysum = 0;
   for (i=start; i<end ; i++)
    {
      mysum += (x[i] * y[i]);
    }
   pthread_mutex_lock (&mutexsum);
   dotstr.sum += mysum;
   pthread_mutex_unlock (&mutexsum);
   pthread_exit((void*) 0);
}
```

```
int main (int argc, char *argv[])
{
long i;
double *a, *b;
void *status;
pthread_attr_t attr;

/* Assign storage and initialize values */

a = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));
b = (double*) malloc (NUMTHRDS*VECLEN*sizeof(double));

for (i=0; i<VECLEN*NUMTHRDS; i++) {
  a[i]=1;
  b[i]=a[i];
  }

dotstr.veclen = VECLEN;
dotstr.a = a;
dotstr.b = b;
dotstr.sum=0;

pthread_mutex_init(&mutexsum, NULL);

/* Create threads to perform the dotproduct  */
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_JOINABLE);

for(i=0;i<NUMTHRDS;i++)
  {
  /* Each thread works on a different set of data.
   * The offset is specified by 'i'. The size of
   * the data for each thread is indicated by VECLEN.
   */
   pthread_create(&callThd[i], &attr, dotprod, (void *)i);
   }

pthread_attr_destroy(&attr);
/* Wait on the other threads */

for(i=0;i<NUMTHRDS;i++) {
  pthread_join(callThd[i], &status);
  }
/* After joining, print out the results and cleanup */

printf ("Sum =  %f \n", dotstr.sum);
free (a);
free (b);
pthread_mutex_destroy(&mutexsum);
pthread_exit(NULL);
}
```

Chương 5 : Lập trình MPI

I. Giới thiệu về MPI

MPI (Message passing Interface) là thư viện lập trình cho máy tính song song dựa vào phương thức gửi nhận tin. MPI có thể được viết bằng C/C++ và Fortran.

MPI được lập trình cho các máy tính có kiến trúc bộ nhớ phân tán, tuy nhiên nó cũng có thể áp dụng cho vài kiến trúc chia sẻ bộ nhớ ví dụ như kiến trúc SMP/NUMA.

II. Xây dựng phương thức Send và Receive

Trong lập trình Message passing, thì send và receive là hai phương thức cơ bản nhất:

```
send(void *sendbuf, int nelems, int dest)
receive(void *recvbuf, int nelems, int source)
```

Trong đó, sendbuf là con trỏ chứa dữ liệu để gửi đi, recvbuf là con trỏ chứa dữ liệu nhận về, nelems là số lượng(đơn vị) để gửi và nhận. dest là định danh của tiến trình nhận dữ liệu, source là định danh của tiến trình gửi dữ liệu.

Chúng ta xét ví dụ sau:

P0	P1

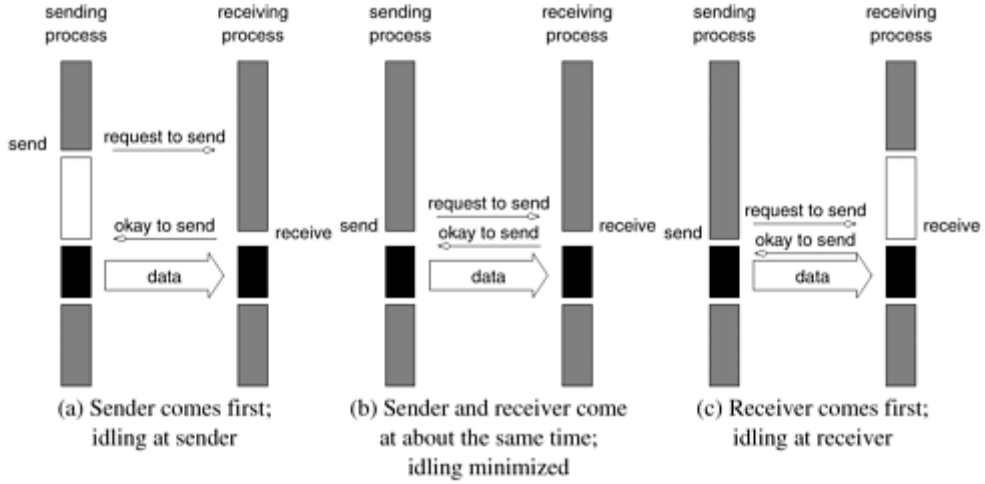
a = 100;
send(&a,1,1);
a = 0;
	Receive(&a,1,0);
	Prinf(“%d”,a);

Trong ví dụ trên, tiến trình P0 gửi giá trị a cho tiến trình P1. Tuy nhiên chú ý rằng trong tiến trình P0, sau khi gửi giá trị a = 100 cho P1 thì lập tức P0 thay đổi giá trị a = 0. Vậy giá trị mà P1 nhận được là giá trị tại thời điểm P0 gửi đi và giá trị đó là 100.

a. Cơ chế Send/Receive không bộ đệm

Trong trường hợp này, hàm send của tiến trình gửi sẽ không kết thúc đến khi mà chưa gặp hàm receive của tiến trình nhận phù hợp. Ở đây, các tiến trình yêu cầu phải tiến hành bắt tay giữa tiến trình gửi và tiến trình nhận. Tiến trình gửi sẽ gửi yêu cầu tới tiến trình nhận, khi đó, tiến trình nhận sẽ

gửi đáp ứng lại cho tiến trình gửi. Vì vậy sẽ không có các bộ nhớ đệm cả bên gửi lẫn bên nhận.



Hình 5.1 – Cơ chế gửi nhận không cần bộ đệm

Trong hình 5.1 chúng ta thấy rằng, trường hợp (a) khi bên gửi và nhận không cùng một lúc thì sẽ gây ra lãng phí về thời gian. Trong trường hợp (c), bên nhận chờ bên gửi.

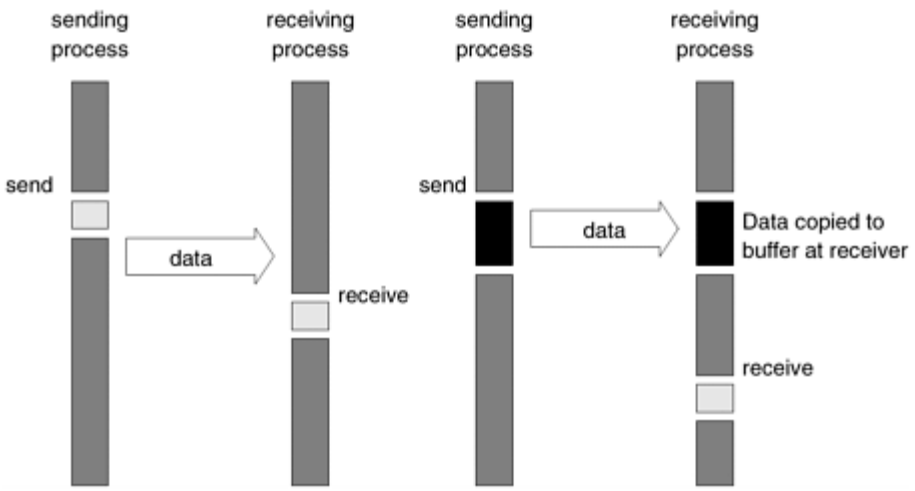
Trong phần này chúng ta phải lưu ý về trường hợp deadlock. Đó là hai tiến trình đều yêu cầu gửi cùng một lúc. Chúng ta xét ví dụ sau:

P0	P1
send(&a,1,1); Receive(&a,1,0);	Send(&a,1,0); Receive(&a,1,0);

Khi P0 yêu cầu gửi sang P1, tuy nhiên P1 lúc này cũng yêu cầu gửi sang P0, khi đó cả 2 tiến trình đều chờ nhau để đáp ứng yêu cầu gửi đó, do đó sẽ dẫn đến trường hợp Deadlock.

b. Cơ chế Send/Receive có bộ đệm

Trong cơ chế này, bên gửi sẽ được cấp phát một bộ nhớ đệm để gửi tin, bên gửi sẽ sao chép dữ liệu vào bộ đệm, và sau đó tiếp tục làm việc khác. Cũng hoàn toàn tương tự với bên nhận, chúng được đưa vào bộ nhớ có địa chỉ xác định, khi tiến trình bên nhận có phương thức receive(), nó sẽ check trong bộ nhớ đệm.



Hình 5.2 – Cơ chế Send/Recevi có bộ đệm

Ví dụ :

```
            P0
for(i:=0 ; i<1000; i++)
{
      produce(&a);
      send(&a, 1, 1);
}
```

```
            P1
for(i:=0; i<1000; i++)
{
      receive(&a, 1, 0);
      consume(&a);
}
```

Trong ví dụ trên ta thấy rằng, tiến trình P0 sẽ sinh ra 1000 giá trị a và gửi sang cho tiến trình P1. Nếu tiến trình P1 không xử lý kịp, khi đó, các giá trị a sẽ được lưu trong bộ đệm của P1 để chờ để xử lý. Vậy nếu bộ đệm của P1 bị đầy thì tiến trình P0 sẽ phải dừng lại để chờ cho đến khi bộ đệm của P1 xử lý hết.

Chú ý trong trường hợp này có thể xảy ra deadlock:

P0	P1
receive(&a, 1, 1);	receive(&a, 1, 0);
send(&b, 1, 1);	send(&b, 1, 0);

Trong ví dụ trên, cả hai Process đều chờ nhau để lấy dữ liệu từ bộ đệm ra để xử lý.

III. Lập trình MPI

Trong thư viện MPI có tới 125 thủ thục, tuy nhiên chúng ta thường chỉ cần dùng tới 6 thủ tục sau:

MPI_Init	Khởi tạo MPI
MPI_Finalize	Kết thúc MPI
MPI_Comm_size	Xác định số lượng tiến trình
MPI_Comm_rank	Xác định tên nhãn của tiến trình
MPI_Send	Gửi một tin
MPI_Recv	Nhận một bản tin

a. Bắt đầu và kết thúc MPI

MPI_Init được dùng để khởi tạo một môi trường MPI, chúng ta gọi MPI_Init một lần trong suốt quá trình chạy chương trình. MPI_Finalize được gọi khi kết thúc, nó sẽ thực hiện dọn dẹp các tác vụ đã kết thúc. Sẽ không được phép gọi MPI_Finalize nếu như chưa gọi MPI_Init.

```
Int MPI_Init(int *argc, char ***argv)
Int MPI_Finalize()
```

Tham số argc và argv của hàm MPI_Init được nhận thông qua các câu lệnh (command-line) của chương trình C. Nếu hai hàm MPI_Init và MPI_Finalize thực hiện thành công thì nó sẽ trả về giá trị là MPI_SUCCESS.

b. Communicators

Mục đích dùng MPI chính là truyền tin giữa các tiến trình với nhau thông qua communication domain. Các thông tin về communication domain được lưu trong MPI_Comm mà được gọi là communicators. Các communicators được sử dụng như các tham số trong các thủ tục(hàm) MPI.

Các tiến trình muốn truyền thông cho nhau với nhiều cách khác nhau thông qua communication domain, vì vậy với mỗi cách sẽ tương ứng với một communicator. MPI định nghĩa ra một communicators mặc định là : MPI_COMM_WORD.

c. Lấy thông tin

Hàm MPI_Comm_size và MPI_Comm_rank được sử dụng để xác định số lượng tiến trình và tên nhãn của tiến trình.

int MPI_Comm_size(MPI_Comm comm, int *size)

int MPI_Comm_rank(MPI_Comm comm, int *rank)

Hàm MPI_Comm_size sẽ trả lại cho biến size về số lượng của tiến trình mà có communicator là comm. Như vậy, nếu như mỗi một bộ vi xử lý xử lý một tiến trình thì size chính là số lượng processor trong máy tính.

Mọi tiến trình của một communicator có một định danh là **rank.** Rank của mộ tiến trình là một số nguyên từ 0 tới kích thước tủa communicator -1.

Ví dụ :

```
#include <mpi.h>

   main(int argc, char *argv[])
   {
     int npes, myrank;

     MPI_Init(&argc, &argv);
     MPI_Comm_size(MPI_COMM_WORLD, &npes);
     MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
     printf("From process %d out of %d, Hello World!\n",
            myrank, npes);
     MPI_Finalize();
   }
```

d. Gửi và nhận tin

Hàm cơ bản để gửi và nhận tin trong MPI là : MPI_Send và MPI_Receive.

int MPI_Send(void *buf, int count, MPI_Datatype datatype,

int dest, int tag, MPI_Comm comm)

int MPI_Recv(void *buf, int count, MPI_Datatype datatype,

int source, int tag, MPI_Comm comm, MPI_Status *status)

MPI_Send gửi dữ liệu được lưu trong một con trỏ là buf. Buffer này chứa kiểu dữ liệu được chỉ ra trong biến datatype. Dung lượng mà buffer được chỉ ra ở biến count. Trong môi trường C có loại kiểu nào thì trong MPI có loại kiểu đó, tuy nhiên trong MPI có thêm hai kiểu mà trong C không có, đó là : MPI_BYTE và MPI_PACKED.

MPI_BYTE tương ứng với 1 byte (8 bit) và MPI_PACKED tương ứng là một tập hợp các dữ liệu mà được tạo bởi bằng cách đóng gói các dữ liệu không liên tiếp nhau.

Trong hàm MPI_Send, dest là định danh của tiến trình nhận, và mỗi bản tin có một biến integer đó là tag, nó được dùng để phân biệt với các bản tin khác, giá trị của nó từ 0 – MPI_TAG_UB (= 32.767).

Trong hàm MPI_Recv, biến source là định danh của tiến trình gửi tin. Biến tag được chỉ ra trong hàm gửi của tiến trình gửi. Nếu source có giá trị là MPI_ANY_SOURCE thì khi đó bất kỳ một tiến trình nào cũng là nguồn của bản tin.

MPI Datatype	C Datatype
MPI_CHAR	signed char
MPI_SHORT	signed short in
MPI_INT	signed int
MPI_LONG	signed long int
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_BYTE	
MPI_PACKED	

Trong hàm MPI_Recv, sau khi bản tin được nhận, biến status được lưu thông tin về hàm MPI_Recv.

```
typedef struct MPI_Status {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
    };
```

MPI_SOURCE và MPI_TAG thì chúng ta đã nói ở trên. MPI_ERROR lưu mã lỗi của bản tin nhận.

Ví dụ : Tác vụ 0 sẽ ping tác vụ 1 :

```
#include "mpi.h"
#include <stdio.h>
int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, dest, source, rc, count, tag=1;
char inmsg, outmsg='x';
MPI_Status Stat;
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank == 0) {
  dest = 1;
  source = 1;
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&Stat);
  }
else if (rank == 1) {
  dest = 0;
  source = 0;
  rc = MPI_Recv(&inmsg, 1, MPI_CHAR, source, tag, MPI_COMM_WORLD,
&Stat);
  rc = MPI_Send(&outmsg, 1, MPI_CHAR, dest, tag, MPI_COMM_WORLD);
  }
rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
printf("Task %d: Received %d char(s) from task %d with tag %d
\n",
       rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

MPI_Finalize();
}
```

IV.Truyền thông tập hợp trong MPI

Truyền thông tập hợp bao gồm tất cả các tiến trình trong phạm vi của một communicator. Mặc định thì tất cả các process đều thuộc communicator MPI_COMM_WORLD.

a. Các loại hoạt động tập hợp

- Synchronization
- Data Movement : Broadcast, scatter/gather, all to all
- Collective computation : Một thành viên của nhóm thu thập dữ liệu từ thành viên khác và thực hiện thao tác trên dữ liệu đó (min, max, add, multiply,…)

b. Các thủ tục

- MPI_Barrier

Barrier dùng để đồng bộ giữa các tác vụ với nhau. Trong MPI sử dụng hàm MPI_Barrier:

int MPI_Barrier(MPI_Comm comm)

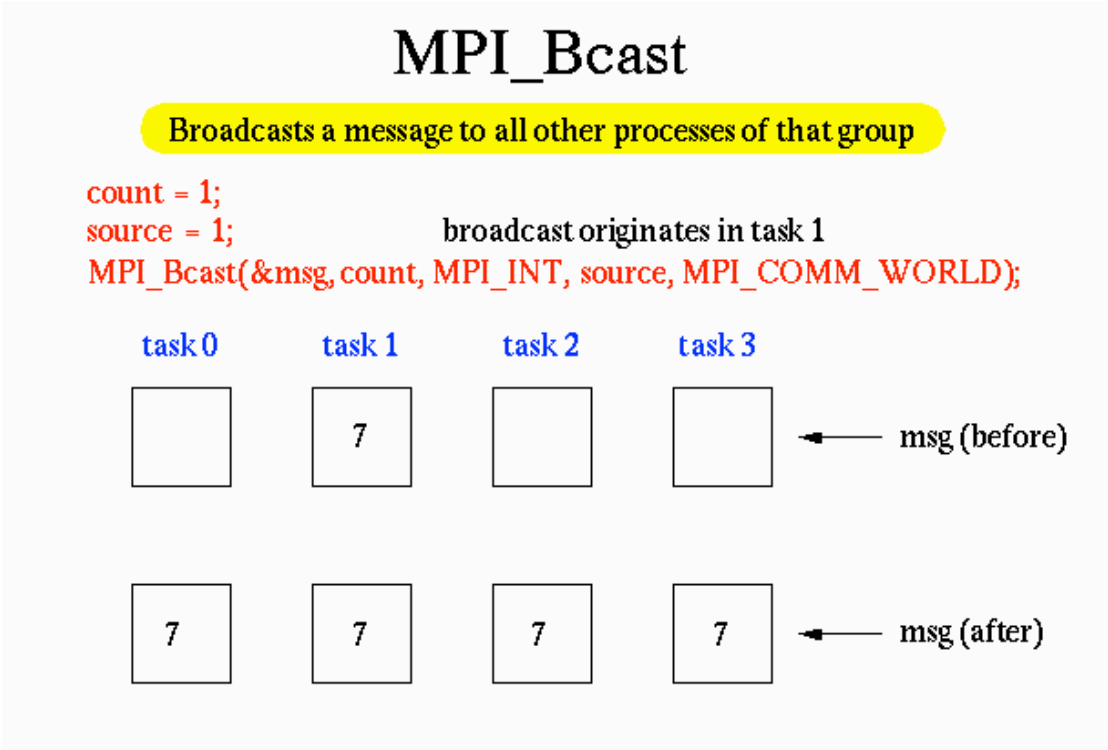
Trong hàm trên chỉ có một tham số là một communicator. Như vậy, trong một communicator, tất cả các tác vụ đều phải dừng lại cho đến khi tất cả đều gọi đến MPI_Barrier.

- MPI_Bcast

Gửi tin nhắn tới tất cả các process trong nhóm

MPI_Bcast (&buffer,count,datatype,root,comm)

MPI_BCAST (buffer,count,datatype,root,comm,ierr)



Hình 5.3 – Broadcast

- MPI_Scatter

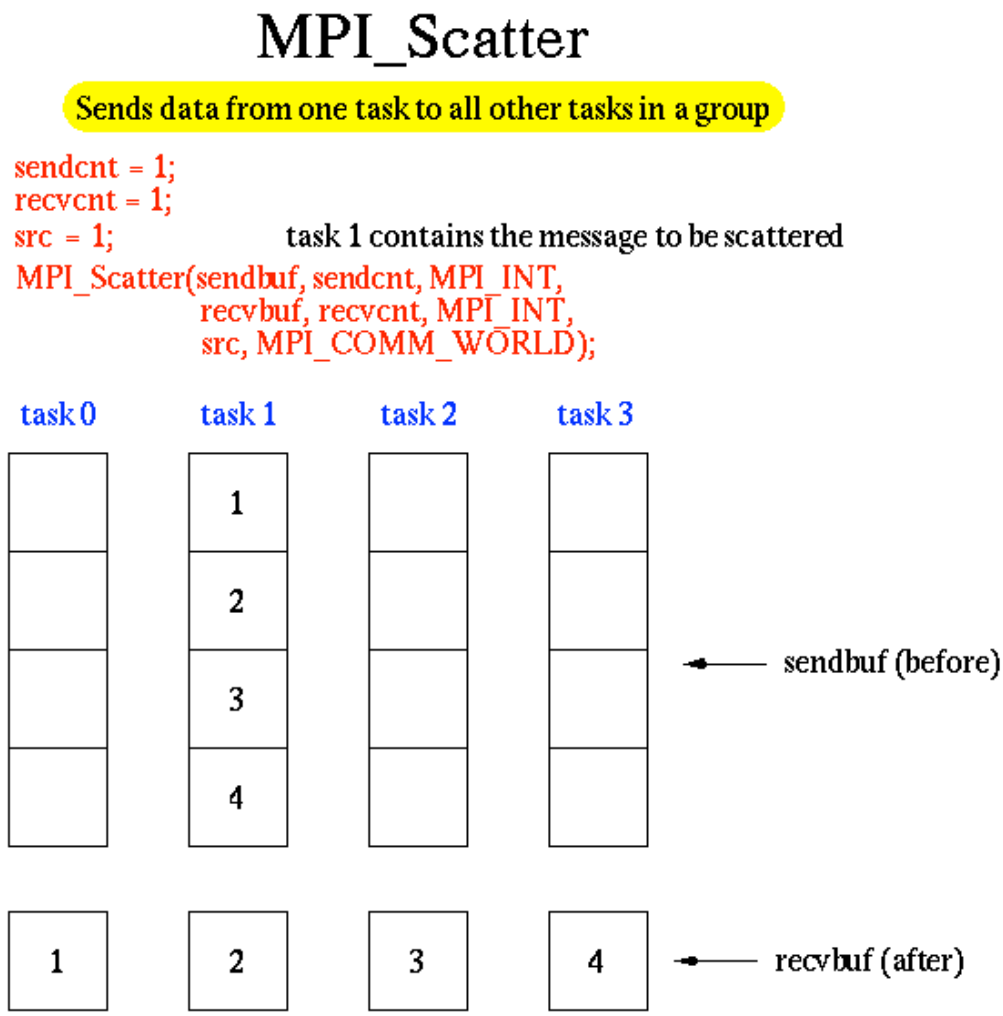
Phát tán một tin nhắn từ một tác vụ tới tất cả các tác vụ khác

MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,

recvcnt,recvtype,root,comm)

MPI_SCATTER (sendbuf,sendcnt,sendtype,recvbuf,

recvcnt,recvtype,root,comm,ierr)



Hình 5.4 – Scatter

- MPI_Gather

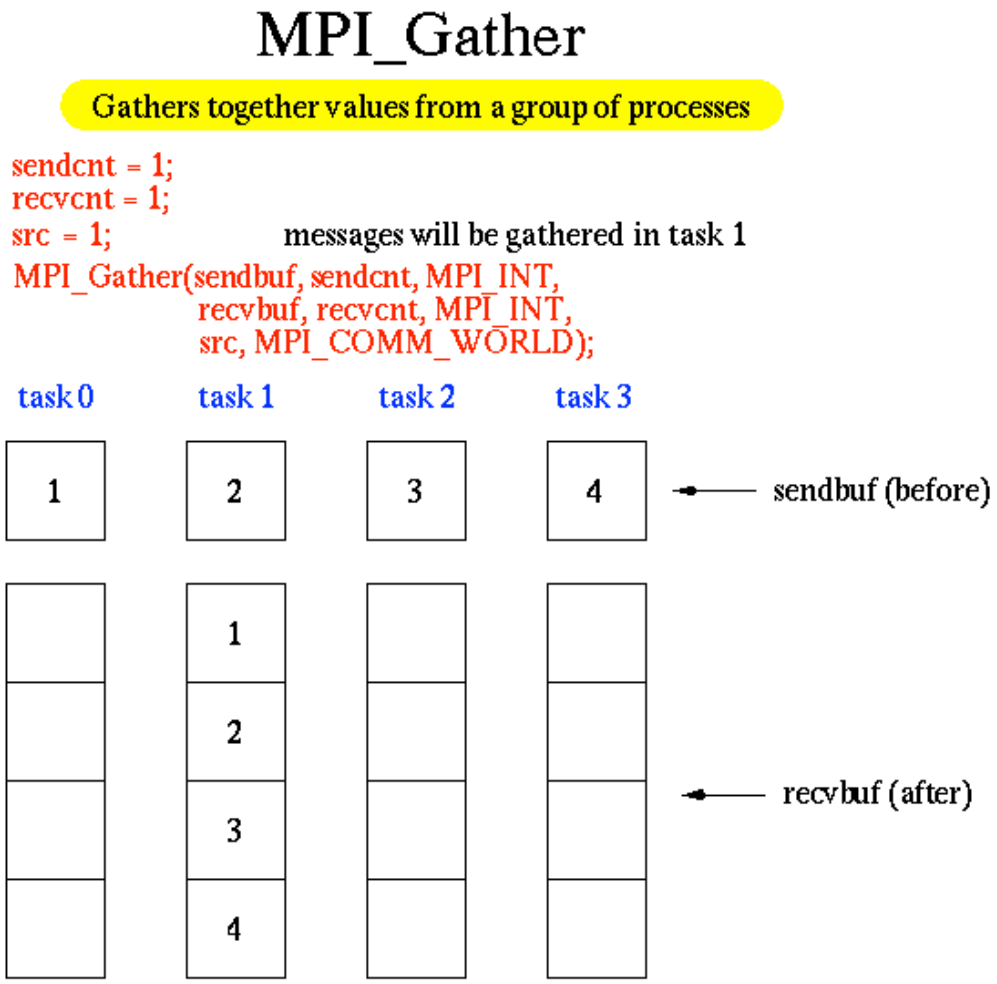
Tập hợp tin nhắn từ các tác vụ trong nhóm tới một tác vụ.

MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,

recvcount,recvtype,root,comm)

MPI_GATHER (sendbuf,sendcnt,sendtype,recvbuf,

recvcount,recvtype,root,comm,ierr)



Hình 5.4 – Gather

- MPI_Allgather

  MPI_Allgather (&sendbuf,sendcount,sendtype,&recvbuf,
  recvcount,recvtype,comm)

  MPI_ALLGATHER (sendbuf,sendcount,sendtype,recvbuf,
  recvcount,recvtype,comm,info)

- MPI_Reduce

Áp dụng một phép toán thu gọn lên các tác vụ trong nhóm và đưa kết quả tới một tác vụ khác.

MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)

MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)

Các phép toán :

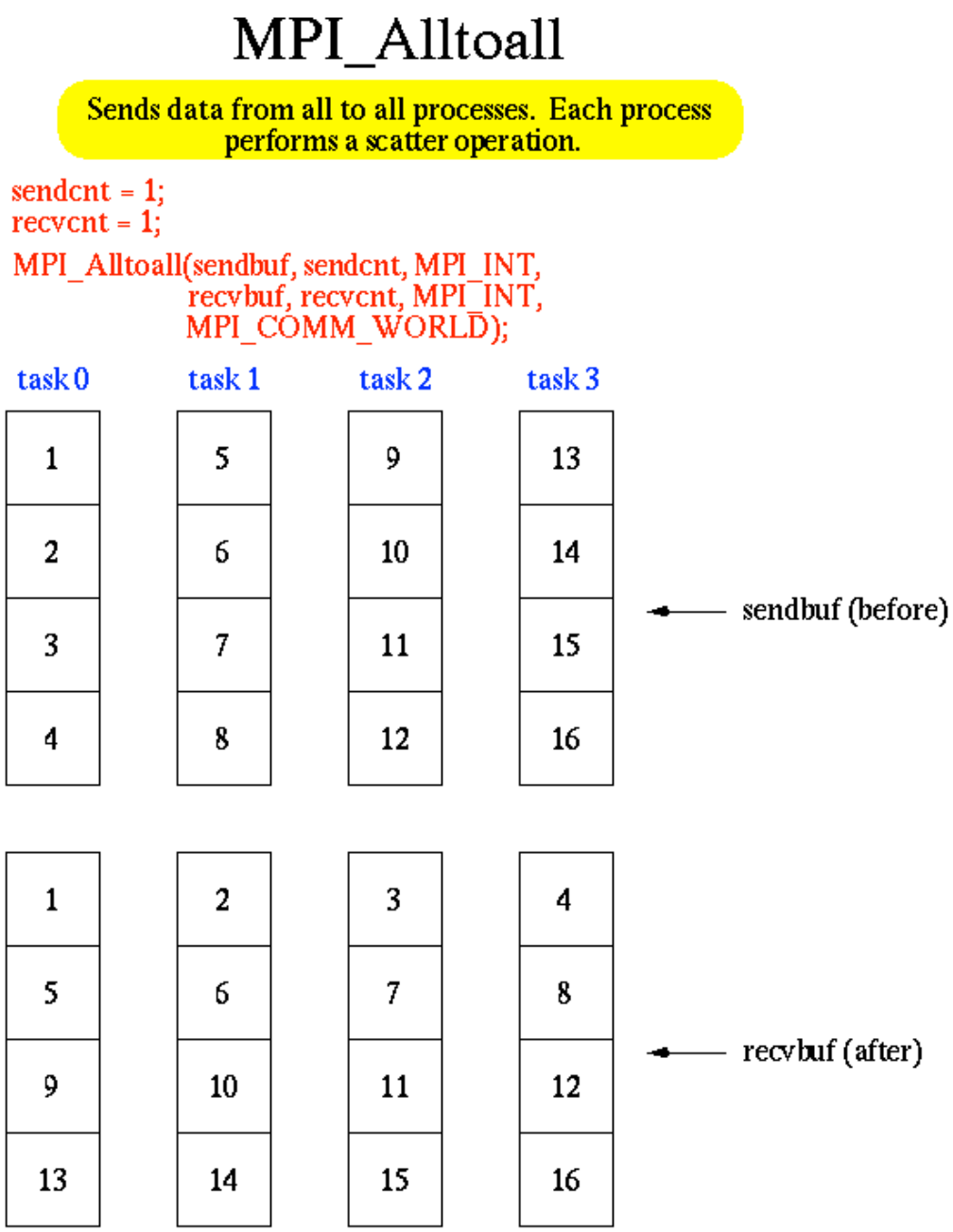
MPI Reduction Operation		C Data types
MPI_MAX	Maximum	Integer,float
MPI_MIN	Minimum	Integer,float
MPI_SUM	Sum	Integer,float
MPI_PROD	Product	Integer,float
MPI_LAND	Logical AND	Integer
MPI_BAND	Bit-wise AND	Integer,MPI_BYTE
MPI_LOR	Logical OR	Integer
MPI_BOR	Bit-wise OR	Integer,MPI_BYTE
MPI_LXOR	Logical XOR	Integer
MPI_BXOR	Bit-wise XOR	Integer,MPI_BYTE
MPI_MAXLOC	Max value and location	Float, double and long double
MPI_MINLOC	Min value and location	Float, double and long double

- MPI_Alltoall

Mỗi tác vụ trong nhóm thực hiện một phép toán phân tán, gửi một bản tin tới các nhóm theo thứ tự.

MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf, recvcnt,recvtype,comm)

MPI_ALLTOALL (sendbuf,sendcount,sendtype,recvbuf, recvcnt,recvtype,comm,ierr)



Hình 5.5 – All to all

Ví dụ : Thực hiện phân tán theo các dòng của một mảng

```
#include "mpi.h"
#include <stdio.h>
#define SIZE 4

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, sendcount, recvcount, source;
float sendbuf[SIZE][SIZE] = {
  {1.0, 2.0, 3.0, 4.0},
  {5.0, 6.0, 7.0, 8.0},
  {9.0, 10.0, 11.0, 12.0},
  {13.0, 14.0, 15.0, 16.0}  };
float recvbuf[SIZE];

MPI_Init(&argc,&argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);

if (numtasks == SIZE) {
  source = 1;
  sendcount = SIZE;
  recvcount = SIZE;
  MPI_Scatter(sendbuf,sendcount,MPI_FLOAT,recvbuf,recvcount,
              MPI_FLOAT,source,MPI_COMM_WORLD);

  printf("rank= %d  Results: %f %f %f %f\n",rank,recvbuf[0],
         recvbuf[1],recvbuf[2],recvbuf[3]);
  }
else
  printf("Must specify %d processors. Terminating.\n",SIZE);

MPI_Finalize();
}
```

Ví dụ : Nhân ma trận một chiều

Theo dòng :

```
RowMatrixVectorMultiply(int n, double *a, double *b, double *x,
                            MPI_Comm comm)
    {
      int i, j;
      int nlocal;
      double *fb;
     int npes, myrank;
     MPI_Status status;

     MPI_Comm_size(comm, &npes);
     MPI_Comm_rank(comm, &myrank);
     fb = (double *)malloc(n*sizeof(double));

     nlocal = n/npes;

     MPI_Allgather(b, nlocal, MPI_DOUBLE, fb, nlocal, MPI_DOUBLE,
         comm);

     for (i=0; i<nlocal; i++) {
       x[i] = 0.0;
       for (j=0; j<n; j++)
         x[i] += a[i*n+j]*fb[j];
     }

     free(fb);
  }
```

Nhân theo cột :

```
   ColMatrixVectorMultiply(int n, double *a, double *b,
double *x,
                           MPI_Comm comm)
   {
     int i, j;
     int nlocal;
     double *px;
     double *fx;
     int npes, myrank;
     MPI_Status status;
     MPI_Comm_size(comm, &npes);
     MPI_Comm_rank(comm, &myrank);

     nlocal = n/npes;

     px = (double *)malloc(n*sizeof(double));
     fx = (double *)malloc(n*sizeof(double));

     for (i=0; i<n; i++) {
       px[i] = 0.0;
       for (j=0; j<nlocal; j++)
         px[i] += a[i*nlocal+j]*b[j];
     }

     MPI_Reduce(px, fx, n, MPI_DOUBLE, MPI_SUM, 0, comm);

     MPI_Scatter(fx, nlocal, MPI_DOUBLE, x, nlocal,
MPI_DOUBLE, 0,
         comm);

     free(px); free(fx);
   }
```