TÍNH TOÁN SONG SONG
(Parallel computing)

TS. Ngô Văn Thanh,
Viện Vật lý.

Chuyên ngành : Công nghệ thông tin.
http://iop.vast.ac.vn/~nvthanh/cours/parcomp/

Chương 3: Lập trình song song

3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp (message passing)

- 3.1.1 Trao đổi thông điệp như một mô hình lập trình.
- 3.1.2 Cơ chế trao đổi thông điệp.
- 3.1.3 Tiếp cận đến một ngôn ngữ cho lập trình song song.

3.2 Thư viện giao diện trao đổi thông điệp (Message Passing Interface – MPI)

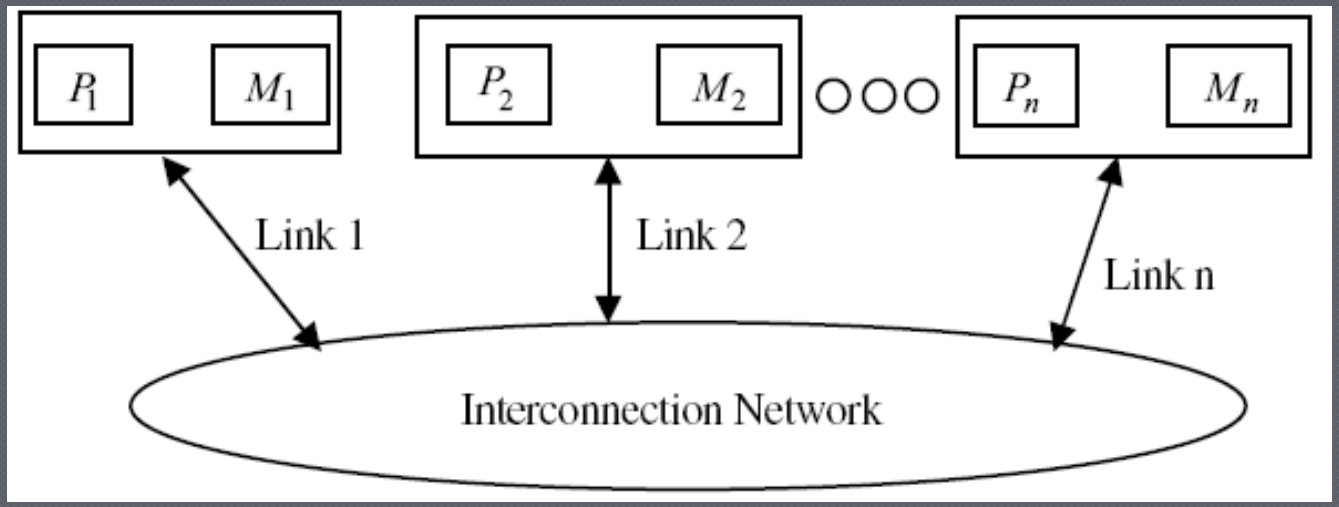
- 3.2.1 Giới thiệu về MPI.
- 3.2.2 Lập trình song song bằng ngôn ngữ C và thư viện MPI.
- 3.2.3 Một số kỹ thuật truyền thông: broadcast, scatter, gather, blocking message passing...

3.3 Máy ảo song song (Parallel Virtual Machine-PVM).

3.4 Thiết kế và xây dựng một chương trình (giải một bài toán (NP-complete) sử dụng MPI và C.

3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp (message passing)

- Phương pháp Message-passing : là phương ra đời sớm nhất và được ứng dụng rộng rãi trong kỹ thuật lập trình song song.
- Dùng để trao đổi thông tin và truyền dữ liệu giữa các processors thông qua cặp lệnh send/receive. Không cần sử dụng bộ nhớ dùng chung.
- Mỗi một node có một processor và một bộ nhớ riêng. Các message được gửi và nhận giữa các node thông qua mạng cục bộ.
- Các nodes truyền thông tin cho nhau thông qua các kết nối (link) và được gọi là *kênh ngoài* (external channels).



3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

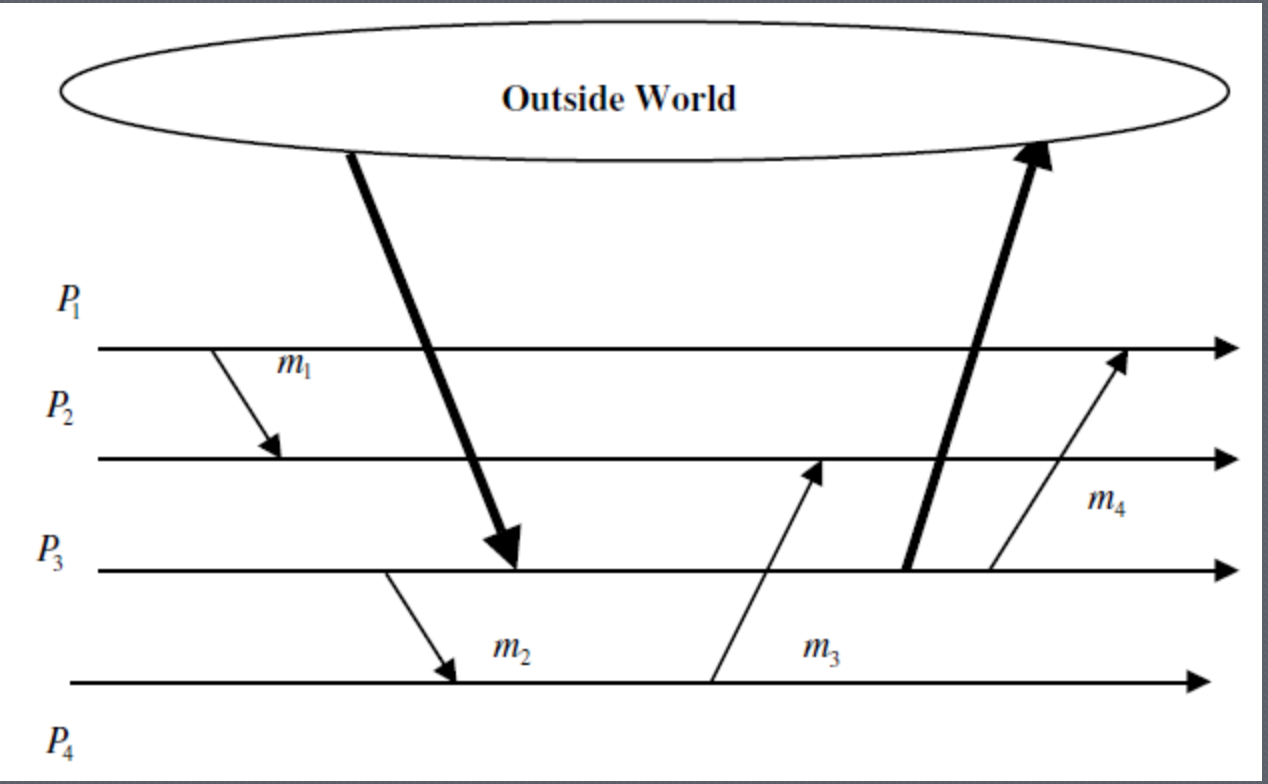
- Các chương trình ứng dụng được chia thành nhiều chu trình, các chu trình được thực hiện đồng thời trên các processors.
- Kiểu chia sẻ thời gian: tổng số các chu trình nhiều hơn số processor.
- Các chu trình chạy trên cùng một processor có thể trao đổi thông tin cho nhau bằng các *kênh trong* (internal channels).
- Các chu trình chạy trên các processor khác nhau có thể trao đổi thông tin thông qua các kênh ngoài.
- Một message có thể là một lệnh, một dữ liệu, hoặc tín hiệu ngắt.
- Chú ý : Dữ liệu trao đổi giữa các processor không thể dùng chung (shared) mà chúng chỉ là bản copy dữ liệu.
- Hạt chu trình (process granularity): là kích thước của một chu trình, được định nghĩa bởi tỷ số giữa thời gian thực hiện chu trình và thời gian truyền thông tin:

$$process\ granularity = (computation\ time)/(communication\ time)$$

- Ưu điểm:
  - Kiểu trao đổi dữ liệu không đòi hỏi cấu trúc đồng bộ của dữ liệu.
  - Có thể dễ dàng thay đổi số lượng các processors.
  - Mỗi một node có thể thực hiện đồng thời nhiều chu trình khác nhau.

3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

- Ví dụ hệ message passing có 4 nodes.
  - m_i là các message trao đổi giữa các processor.
  - Các mũi tên thể hiện hướng trao đổi message giữa hai processors.
  - Hệ message passing có thể tương tác với thế giới bên ngoài (hệ ngoài) cũng phải thông qua các quá trình nhận và gửi các message.

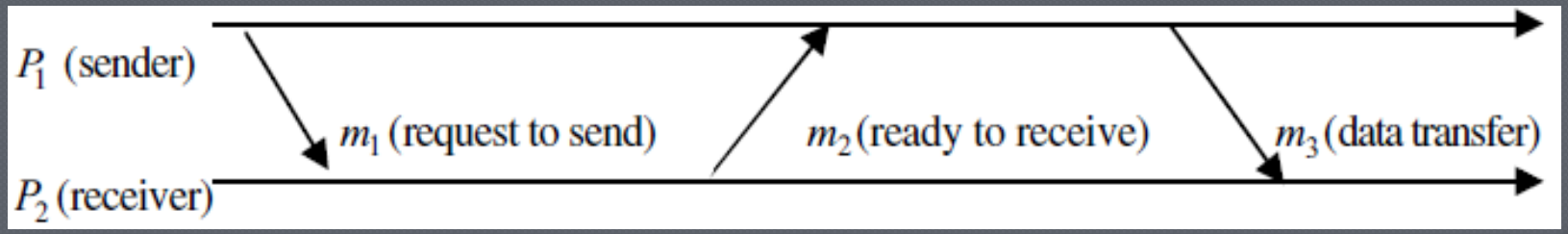


3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

3.1.1 Trao đổi thông điệp như một mô hình lập trình.

- Cấu trúc message passing sử dụng các lệnh mà nó cho phép các chu trình truyền thông tin cho nhau: *send*, *receive*, *broadcast* và *barrier*.
  - Lệnh *send* : lấy dữ liệu từ vùng nhớ đệm (buffer memory) và gửi nó đến một node nào đó.
  - Lệnh *receive* : cho phép nhận một message từ một node khác gửi đến, message này được lưu lại trên một vùng nhớ đệm riêng.

- Mô hình lập trình cơ bản:
  - Kiểu Blocking:
    - Kiểu blocking: Các yêu cầu send từ một processor và yêu cầu receive từ một processor khác đều bị khóa. Dữ liệu được phép chuyển đi khi và chỉ khi node sender đã nhận được trả lời yêu cầu nhận từ node receiver.
    - Kiểu blocking cần phải có 3 bước: Bước 1) gửi yêu cầu truyền dữ liệu đến node nhận. Bước 2) node nhận lưu yêu cầu đó lại và gửi một message trả lời. Bước 3) node gửi bắt đầu gửi dữ liệu đi sau khi đã nhận được trả lời từ node nhận.

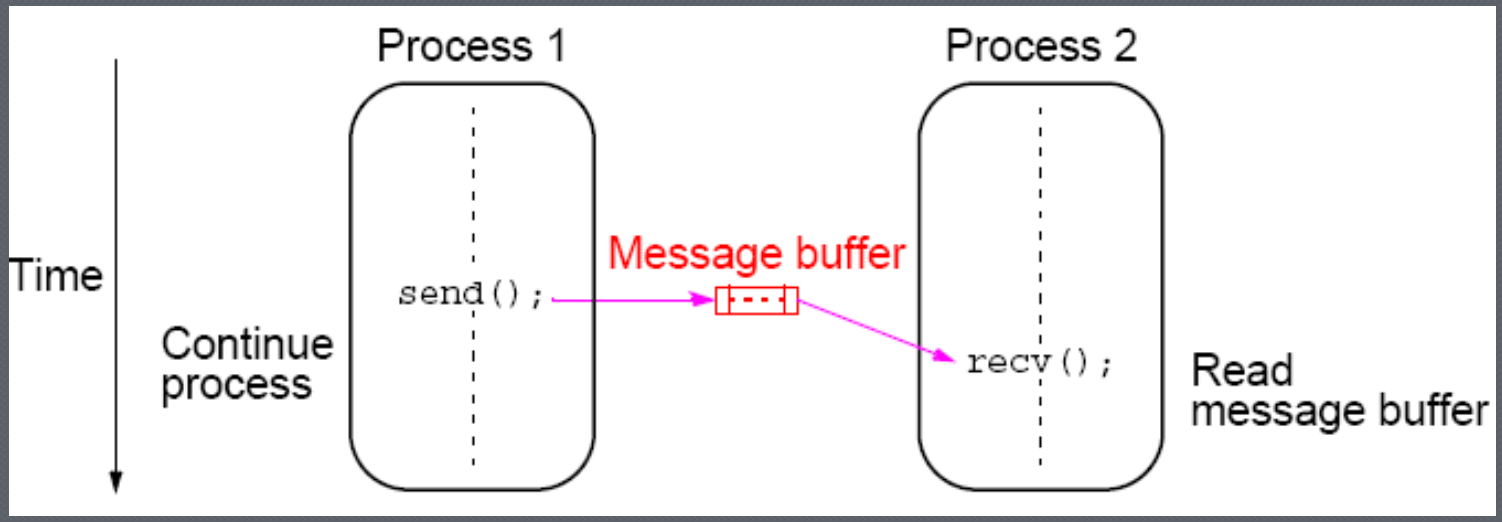


3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

- Ưu điểm: đơn giản, cả hai nodes sender và receiver không cần sử dụng bộ nhớ đệm.
- Nhược điểm: cả hai nodes sender và receiver đều bị khóa (blocked) trong suốt quá trình thực hiện gói send/receive. Trong quá trình này, các processor không hoạt động (trạng thái nghỉ). Không thể thực hiện đồng thời cả việc truyền thông tin và việc tính toán.

➢ Kiểu nonblocking:

- Node sender gửi message trực tiếp cho node receiver mà không phải chờ thông tin trả lời. Mọi dữ liệu được lưu lại trên vùng nhớ đệm và sẽ được truyền đi khi cổng kết nối giữa hai node đã mở.
- Nhược điểm: dễ bị tràn bộ nhớ đệm nếu như các node receiver xử lý không kịp các thông tin gửi từ node sender.



3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

- Ví dụ: tính

$$y = (a + b) \times (c + d)$$

  - Tính trên một processor phải thực hiện qua 8 bước.
  - Tính trên hai processor phải thực hiện qua 7 bước.

Bước tính	Công việc
1	Đọc a
2	Tính a+b
3	Lưu kết quả
4	Đọc c
5	Tính c+d
6	Tính (a+b)*(c+d)
7	Ghi kết quả
8	Kết thúc

Bước tính	Công việc trên P1	Công việc trên P2
1	Đọc a	Đọc c
2	Tính a+b	Tính c+d
3	Gửi kquả cho P2	Lưu kquả
4	Kết thúc	Nhận kquả từ P1
5		Tính (a+b)*(c+d)
6		Ghi kết quả
7		Kết thúc

3.1.2 Cơ chế trao đổi thông điệp.

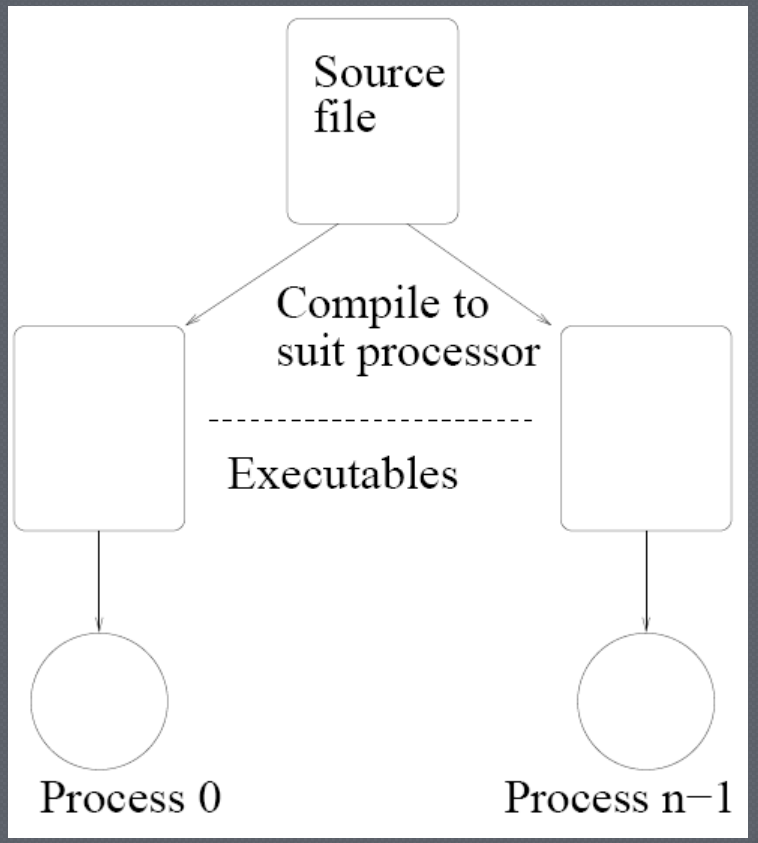
- Định tuyến mạng trong message passing.
  - Được sử dụng cho các message để chọn đường dẫn trên các kênh mạng.
  - Kỹ thuật định tuyến dùng để tìm ra tất cả các đường dẫn khả dĩ để một message có thể đi đến đích, sau đó chọn ra một đường dẫn tốt nhất.
  - Có hai kiểu định tuyến:
    - Định tuyến trung tâm: Tất cả các đường dẫn được thiết lập đầy đủ trước khi gửi message. Kỹ thuật này cần phải xác định được trạng thái nghỉ của tất cả các node trong mạng.
    - Định tuyến phân tán: Mỗi một node tự chọn cho mình các kênh để chuyển tiếp một message đến node khác. Kỹ thuật này chỉ cần biết trạng thái của các node bên cạnh.
  - Định tuyến cho Broadcasting and Multicasting.
    - Broadcast: một node gửi thông điệp cho tất cả các node khác. Nó được ứng dụng để phân phát dữ liệu từ một node đến các node khác.
    - Multicast: một node gửi thông điệp chỉ cho một số node đã chọn, kỹ thuật này được ứng dụng trong các thuật toán tìm kiếm trên hệ multiprocessor.

3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

- Chuyển mạch trong message passing.
  - Được sử dụng để di chuyển dữ liệu từ kênh vào sang kênh ra.
  - Các kiểu chuyển mạch:
    - Store-and-forward: truyền dữ liệu theo kiểu tuần tự, mục đích là để đảm bảo cân bằng tải động cho quá trình truyền message qua mạng.
      - Packet-switched : mỗi một message được chia thành nhiều gói nhỏ (packet) có cùng kích thước. Mỗi một node cần phải có vùng nhớ đệm đủ lớn để lưu giữ packet này trước khi chuyển chúng đi. Mỗi một packet cần phải được dán nhãn để kết nối với nhau sau khi đã truyền xong.
      - Virtual cut-through: packet chỉ lưu trữ trên các node trung gian nếu như node kế tiếp đang còn bận. Nếu node kế tiếp trên đường truyền không bị bận thì nó sẽ gửi luôn packet đi mà không cần phải nhận đầy đủ packet từ node trước nó.
    - Circuit-switching: Các liên kết trên đường truyền dữ liệu từ node nguồn sang node đích được khép kín, không cần sử dụng bộ nhớ đệm trên mỗi node. Sau khi dữ liệu đã được truyền xong, các liên kết này sẽ được giải phóng để sử dụng cho các message khác. Kiểu chuyển mạch này được ứng dụng trong việc truyền dữ liệu có dung lượng lớn do thời gian trễ bé. Đây là một kiểu cân bằng tải tĩnh.

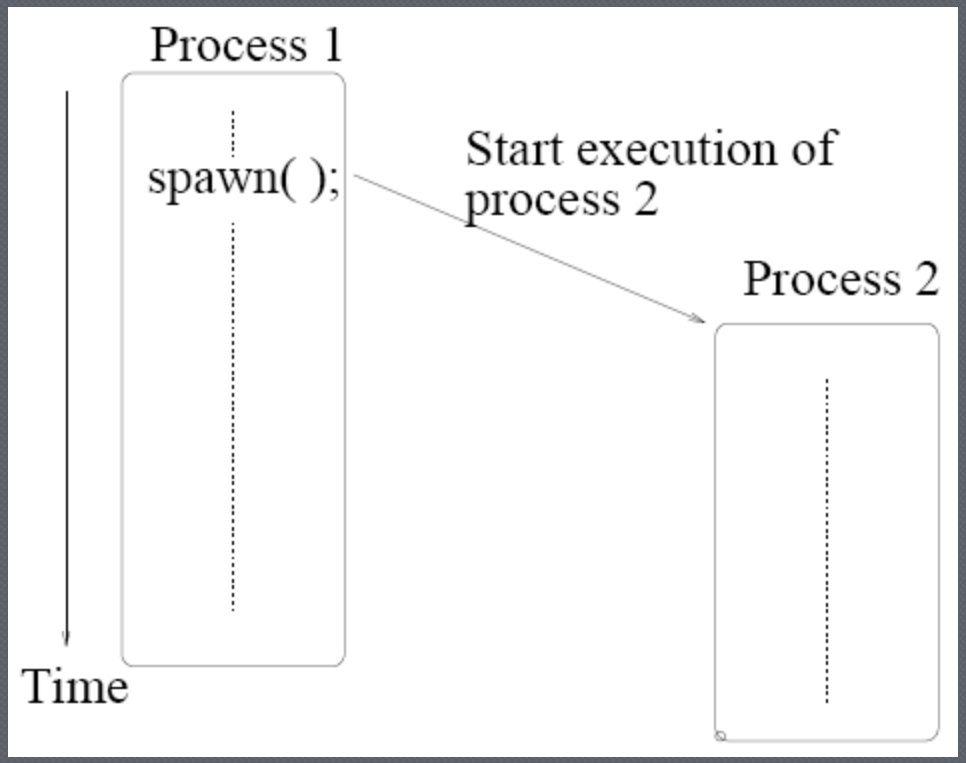
3.1.3 Tiếp cận đến một ngôn ngữ cho lập trình song song.

- Mô hình SPMD (Single Program Multiple Data).
  - Các chu trình được viết chung trong một chương trình.
  - Trong chương trình có các câu lệnh điều khiển để phân phát các phần khác nhau cho các processor.
  - Các chu trình trong chương trình là chu trình tĩnh.
  - Đây là cơ sở cho sự ra đời thư viện MPI (message passing interface).



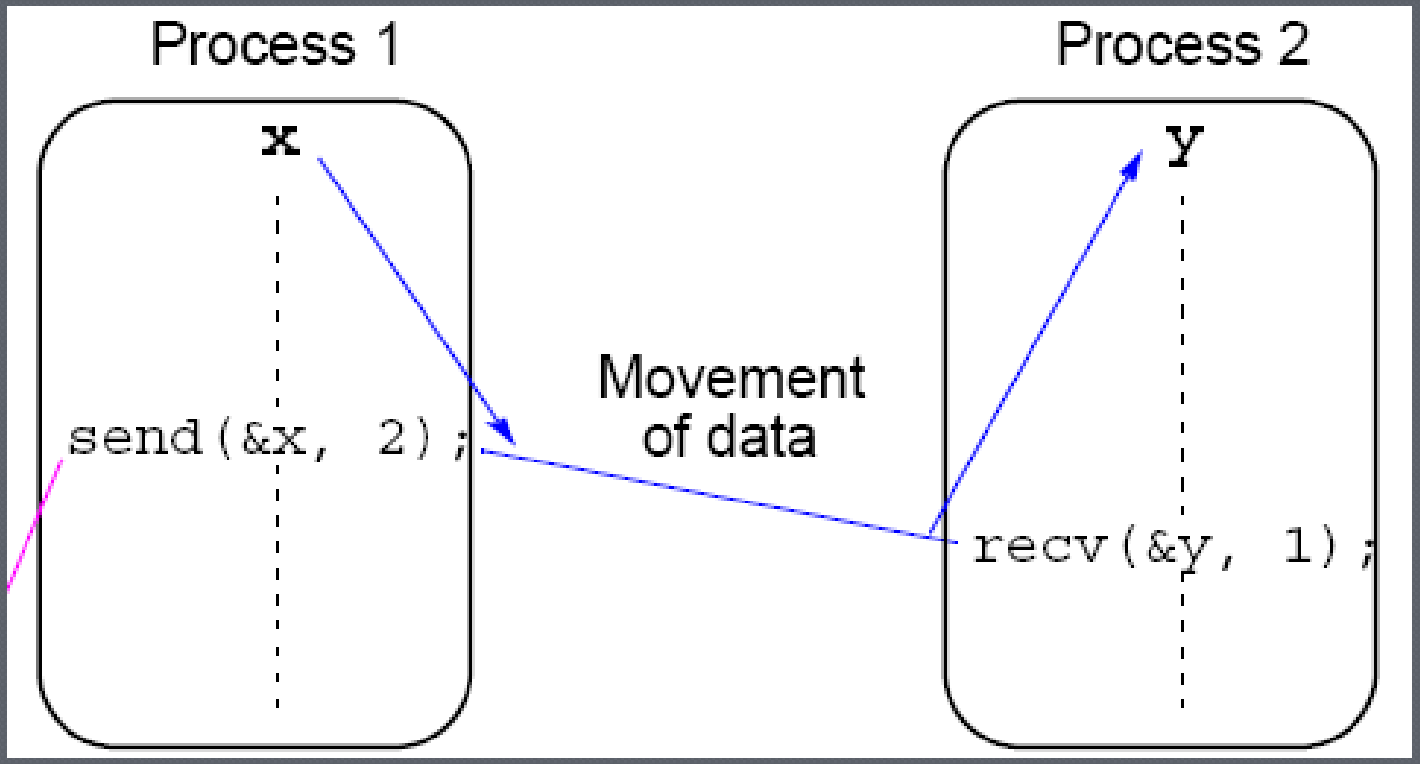
3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

- Mô hình MPMD (Multiple Program Multiple Data).
  - Các chương trình tách biệt được viết riêng cho từng chu trình.
  - Sử dụng phương pháp master-slave.
  - Một processor thực hiện các chu trình master, các chu trình khác (các chu trình slave) sẽ được khởi tạo từ chu trình master trong quá trình chạy.
  - Các chu trình là chu trình động.
  - Đây là cơ sở cho sự ra đời của bộ thư viện PVM (parallel virtual machine).



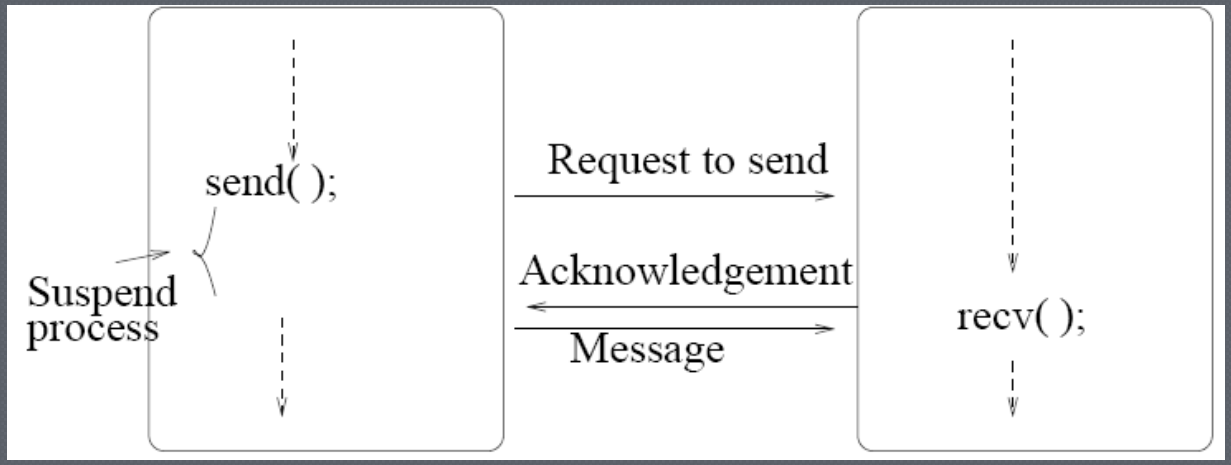
3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

- Các thủ tục cơ bản theo kiểu point-to-point: send và receive .
  - Các thủ tục thường kết thúc khi mà message đã được truyền xong.
  - Thủ tục send đồng bộ: Chờ thông tin chấp nhận từ chu trình nhận trước khi gửi message.
  - Thủ tục receive đồng bộ: chờ cho đến khi message đã đến.

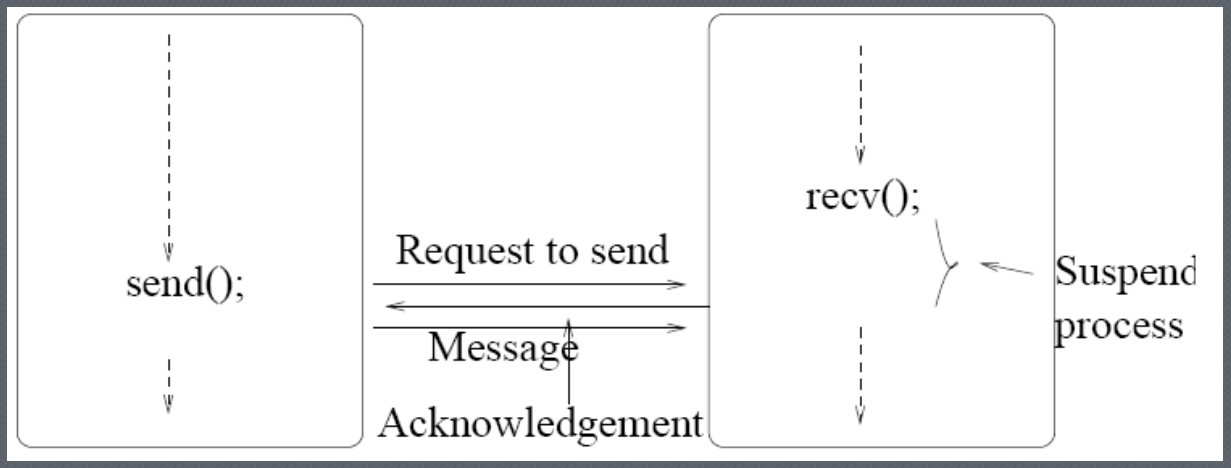


3.1 Cơ bản về giao tiếp bằng phương pháp trao đổi thông điệp

- send và receive đồng bộ :
  - send xuất hiện trước receive.



  - receive xuất hiện trước send.



3.2 Thư viện giao diện trao đổi thông điệp (Message Passing Interface – MPI)

3.2.1 Giới thiệu về MPI.

- MPI là một bộ thư viện hỗ trợ cho việc lập trình kiểu message passing.
- Thư viện MPI bao gồm các thủ tục truyền tin kiểu point-to-point , và các toán hạng chuyển dữ liệu, tính toán và đồng bộ hóa.
- MPI(1) chỉ làm việc trên các chu trình tĩnh, tất cả các chu trình cần phải được định nghĩa trước khi thực hiện và chúng sẽ được thực hiện đồng thời.
- MPI-2 là phiên bản nâng cấp của MPI, có thêm các chức năng có thể đáp ứng cho các chu trình động, kiểu server-client…
- Trong một chương trình ứng dụng, lập trình viên đưa thêm một số lệnh điều khiển link đến các hàm/thủ tục của bộ thư viện MPI. Mỗi một tác vụ trong chương trình được phân hạng (rank) hay đánh số bằng các số nguyên từ 0 đến $n - 1$. n là tổng số các tác vụ.
- Các tác vụ MPI dựa trên các rank đó để phân loại message gửi và nhận, sau đó áp dụng các toán hạng phù hợp để thực hiện các tác vụ.
- Các tác vụ MPI có thể chạy đồng thời trên cùng một processor hoặc trên các processor khác nhau.

3.2 Thư viện giao diện trao đổi thông điệp

3.2.2 Lập trình song song bằng ngôn ngữ C và thư viện MPI.

- Communicator
  - Communicator là môi trường truyền thông tin (communication context) cho nhóm các tác vụ. Để truy cập đến một communicator, các biến cần phải được khai báo kiểu : `MPI_COMM`
  - Khi chương trình MPI bắt đầu chạy thì tất cả các tác vụ sẽ được liên kết đến một communicator toàn cục (`MPI_COMM_WORLD`).
- Nhóm tác vụ: `MPI_Group`
  - Các tác vụ trong MPI có thể được chia thành các nhóm, mỗi nhóm được gán nhãn (đặt tên). Các toán hạng củranka MPI chỉ làm việc với các thành viên trong nhóm.
  - Các thành viên trong nhóm được nhận dạng nhờ vào hạng của nó (rank).
  - MPI cho phép tạo ra những nhóm mới mà các thành viên của nó là tập hợp của các thành viên trong cùng một nhóm hoặc từ các nhóm khác nhau.
- Conmunicator ngầm định: `MPI_COMM_WORLD`
  - `MPI_COMM_WORLD`: Được khởi tạo ngay khi lệnh `MPI_Init()` được gọi. Tham số này được dùng chung trong tất cả các chu trình, nó giữ nguyên không thay đổi trong suốt quá trình thực hiện tác vụ.

3.2 Thư viện giao diện trao đổi thông điệp

- Lệnh `MPI Init()`: Bắt đầu thực hiện các thủ tục MPI.
- Lệnh `MPI Finalize()`: Kết thúc các thủ tục MPI.
  - Ví dụ:

```
main (int argc, char *argv[])
{
    MPI Init(&argc,&argv);
    MPI Comm rank(MPI COMM WORLD,&myrank);
    if (myrank == 0)
          master();     /* master code */
    else
          salve();      /* slave code */
    …
    MPI Finalize();
}
```

3.2 Thư viện giao diện trao đổi thông điệp

- Hạng tác vụ (task rank): `MPI_Comm_rank()`
  - `MPI_Comm_rank()` : trả lại chỉ số rank của tác vụ.
    - Cú pháp:

```
MPI_Comm communicator;      /* communicator handle */
int my_rank;            /* the rank of the calling task */
MPI_Comm_rank(communicator, &my_rank);
```

- Các thủ tục liên quan đến nhóm của communicator.
  - `MPI_Comm_group()`: tạo một nhóm mới từ các nhóm đã có.
    - Cú pháp:

```
MPI_Comm communicator;      /*communicator handle */
MPI_Group corresponding_group;    /*group handle */
MPI_Comm_group(communicator, &corresponding_group)
```

  - `MPI_Comm_size()` : trả lại kích thước của nhóm (tổng số các tác vụ).
    - Cú pháp:

```
MPI_Comm communicator;      /*communicator handle */
int number_of_tasks;
MPI_Comm_size(communicator, &number_of_tasks)
```

3.2 Thư viện giao diện trao đổi thông điệp

- Ví dụ: chương trình có 5 tác vụ T0,T1,T2,T3,T4, có các rank tương ứng là 0,1,2,3,4. Ban đầu cả 5 tác vụ đều được tham chiếu lên communicator `MPI_COMM_WORLD`.
  - Giả sử tác vụ T3 thực hiện lệch gọi:

    ```
    MPI_Comm_rank(MPI_COMM_WORLD, &me);
    ```

    - Biến `me` được gán giá trị là 3.

    ```
    MPI_Comm_size(MPI_COMM_WORLD, &n)
    ```

    - Biến `n` có giá trị là 5.
  - Để tạo một nhóm bao gồm tất cả các tác vụ trong chương trình:

    ```
    MPI_Comm_group(MPI_COMM_WORLD, &world_group)
    ```

- Các thủ tục tạo mới communicator.
  - Tạo bản sao communicator (duplicate)

    ```
    MPI_Comm_dup(oldcomm, &newcomm)
    ```

  - Tạo mới một communicator tương ứng với một nhóm của communicator cũ.

    ```
    MPI_Comm_create(oldcomm, group, &newcomm)
    ```

  - Tạo một communicator tương ứng với một nhóm con được tách ra từ nhóm cũ.

    ```
    MPI_Comm_split(oldcomm, split_key, rank_key, &newcomm)
    ```

3.2 Thư viện giao diện trao đổi thông điệp

- Ví dụ: chương trình có 5 tác vụ T0,T1,T2,T3,T4, có các rank tương ứng là 0,1,2,3,4. Ban đầu chỉ có một nhóm tên là "small_group" với hai phần tử là T0 và T1.
  - Thủ tục tạo communicator mới cho nhóm đó:

```
MPI_Comm_create(MPI_COMM_WORLD, small_group, &small_comm)
```

  - Tách các tác vụ thành hai nhóm, đặt hai giá trị `split_key = 8` và `5`.

T0 gọi thủ tục với `x = 8` và `me = 0`

```
MPI_Comm_split(MPI_COMM_WORLD, x, me, &newcomm)
```

T1 gọi thủ tục với `x = 5` và `me = 1`

```
MPI_Comm_split(MPI_COMM_WORLD, y, me, &newcomm)
```

T2 gọi thủ tục với `x = 8` và `me = 2`

```
MPI_Comm_split(MPI_COMM_WORLD, x, me, &newcomm)
```

T3 gọi thủ tục với `x = 5` và `me = 3`

```
MPI_Comm_split(MPI_COMM_WORLD, y, me, &newcomm)
```

T4 gọi thủ tục với `x = MPI_UNDEFINED` và `me = 4`

```
MPI_Comm_split(MPI_COMM_WORLD,MPI_UNDEFINED,me,&newcomm)
```

  - Kết quả là có hai nhóm {T0,T2} và {T1,T3}. T4 không thuộc nhóm nào.

3.2 Thư viện giao diện trao đổi thông điệp

- Truyền thông tin giữa các tác vụ.
  - Lệnh `send()`: sender sẽ bị khóa cho đến khi message đã được sao chép đầy đủ lên bộ đệm nhận.

    `MPI_Send(buf,count,data_type,dest,tag,commu)`
    - `buf`: địa chỉ của bộ đệm gửi; `count`: số phần tử cần gửi
    - `data_type`: kiểu dữ liệu; `dest`: rank của chu trình nhận
    - `tag` : nhãn của message; `commu`: communication.
  - Lệnh `receive()`: receiver cũng bị khóa cho đến khi message đã được nhận từ bộ đệm.

    `MPI_Recv(buf,count,data_type,source,tag,commu,&status)`
    - `source`: rank của chu trình gửi; `status`: cho biết kết quả của việc nhận message có thành công hay không?
    - Lệnh send và receive phải có cùng tham số `commu`.
  - Lệnh `Isend()/Irecv()`: sender và receiver không bị khóa.

    `MPI_ISend(buf,count,data_type,dest,tag,commu,&request)`

    `MPI_IRecv(buf,count,data_type,source,tag,commu,&request)`
    - `request` dùng để kiểm tra thủ tục send/receive đã hoàn thành hay chưa.

3.2 Thư viện giao diện trao đổi thông điệp

- Lệnh kiểm tra: kiểm tra trạng thái kết thúc của thủ tục `Isend/Irecv`.

```
MPI_Test(request, &flag, &status)
```

  - `request`: tên biến yêu cầu đã dùng trong các lệnh `Isend/Irecv`.
  - `flag`: là biến logic, có giá trị `TRUE` nếu như quá trình truyền tin đã xong.
  - `status`: thông tin bổ sung về trạng thái của thủ tục `Isend/Irecv`.
- Lệnh chờ: Yêu cầu chờ cho đến khi việc truyền tin đã hoàn thành.

```
MPI_Wait(request, &status)
```

- Lệnh kiểm tra và lệnh chờ cho nhiều `request`.

```
MPI_Testall(count, array_of_requests, &flag, &array_of_
Statuses)
```

  - Trả lại giá trị `TRUE` nếu tất cả các `requests` đã hoàn thành.

```
MPI_Testany(count, array_of_requests, &flag, &status)
```

  - Trả lại giá trị `TRUE` nếu một trong số các `requests` đã hoàn thành.

```
MPI_Waitall(count, array_of_requests, &array_of_statuses)
```

  - Chờ cho đến khi tất cả các `requests` đã hoàn thành.

```
MPI_Waitany(count, array_of_requests, &status)
```

  - Chờ cho đến khi một trong số các `requests` đã hoàn thành.

3.2 Thư viện giao diện trao đổi thông điệp

- Lệnh đồng bộ hóa (rào chắn).

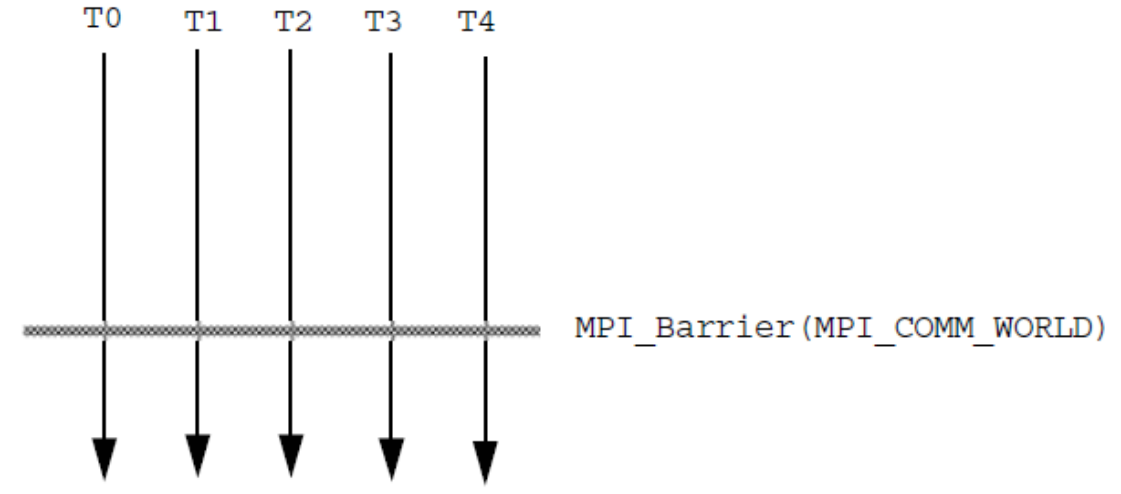
```
MPI_Barrier(communicator)
```

  - Tác vụ tại barrier phải chờ cho đến khi tất cả các tác vụ khác trên cùng một communicator đã hoàn thành.

- Ví dụ: chương trình có 5 tác vụ T0,T1,T2,T3,T4, có các rank tương ứng là 0,1,2,3,4. Ban đầu cả 5 tác vụ đều được tham chiếu lên communicator `MPI_COMM_WORLD`. Sử dụng lệnh:

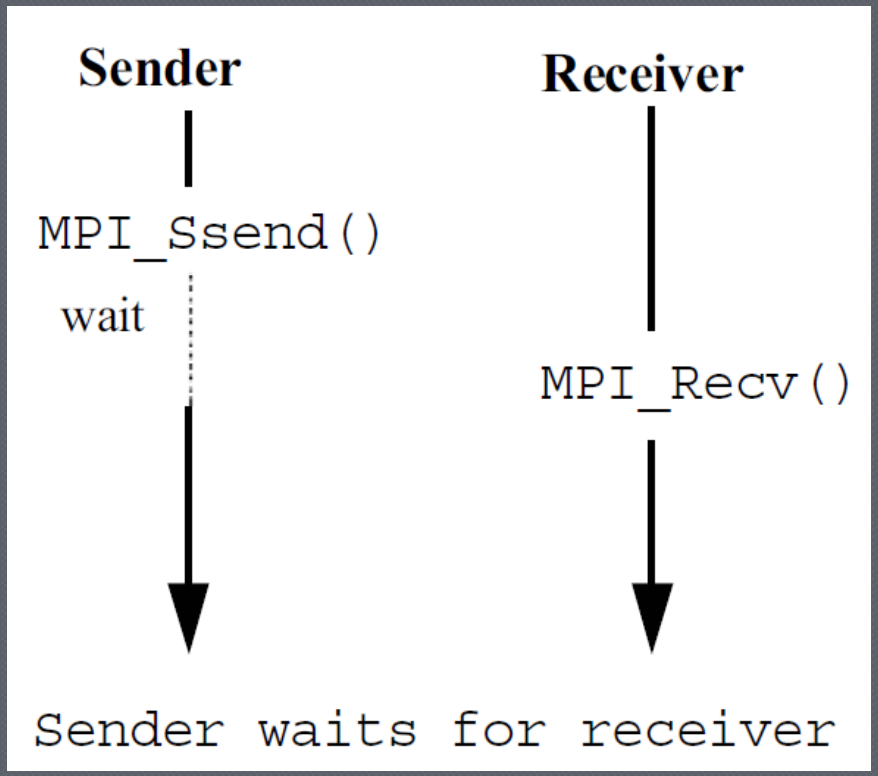
```
MPI_Barrier(MPI_COMM_WORLD)
```

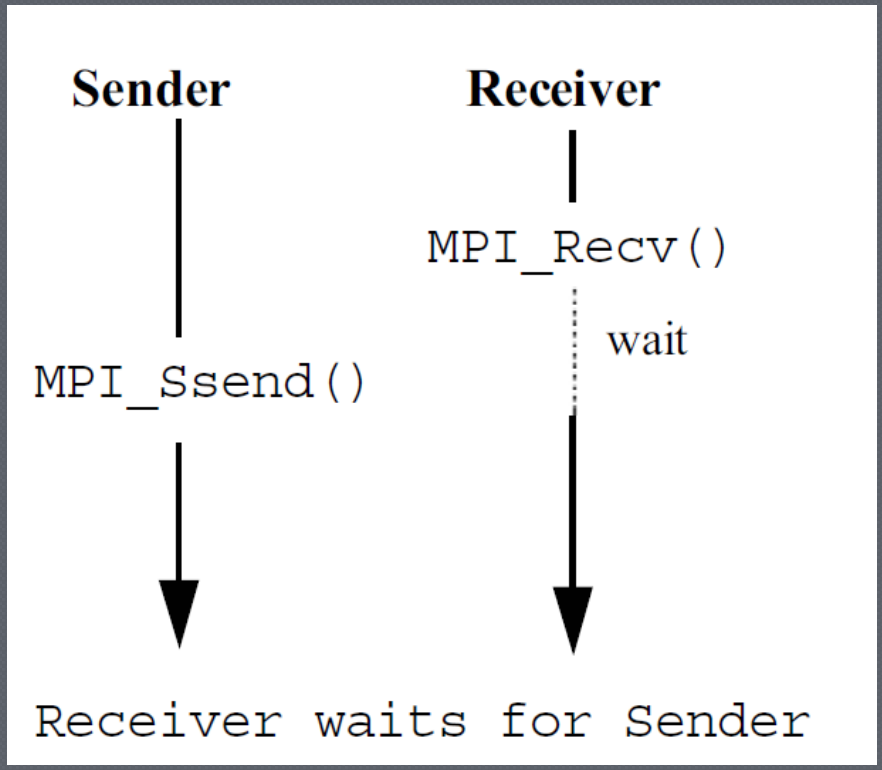
  - Yêu cầu các tác vụ phải chờ tại barrier cho đến khi tất cả các tác vụ đều đến được barrier.



3.2 Thư viện giao diện trao đổi thông điệp

- Lệnh `Ssend/Srecv`: gửi và nhận đồng bộ.
  - Lệnh `Ssend` sẽ chờ cho đến khi thông tin đã được nhận. Lệnh `Srecv` sẽ chờ cho đến khi thông tin đã được gửi.
  - Cả hai chu trình nhận và gửi đều bị block.





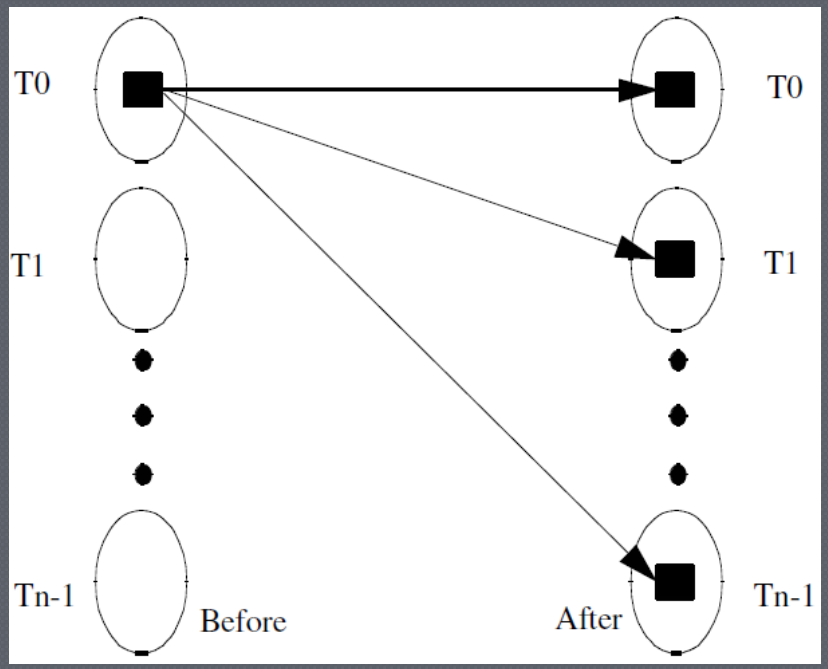
3.2 Thư viện giao diện trao đổi thông điệp

3.2.3 Một số kỹ thuật truyền thông: broadcast, scatter, gather, blocking message passing...

- Broadcast:

```
MPI_Bcast(buf, n, data_type, root, communicator)
```

  - Lệnh gửi bản sao của một buffer có kích thước là `n` từ một tác vụ `root` đến tất cả các tác vụ khác trong cùng communicator.



3.2 Thư viện giao diện trao đổi thông điệp

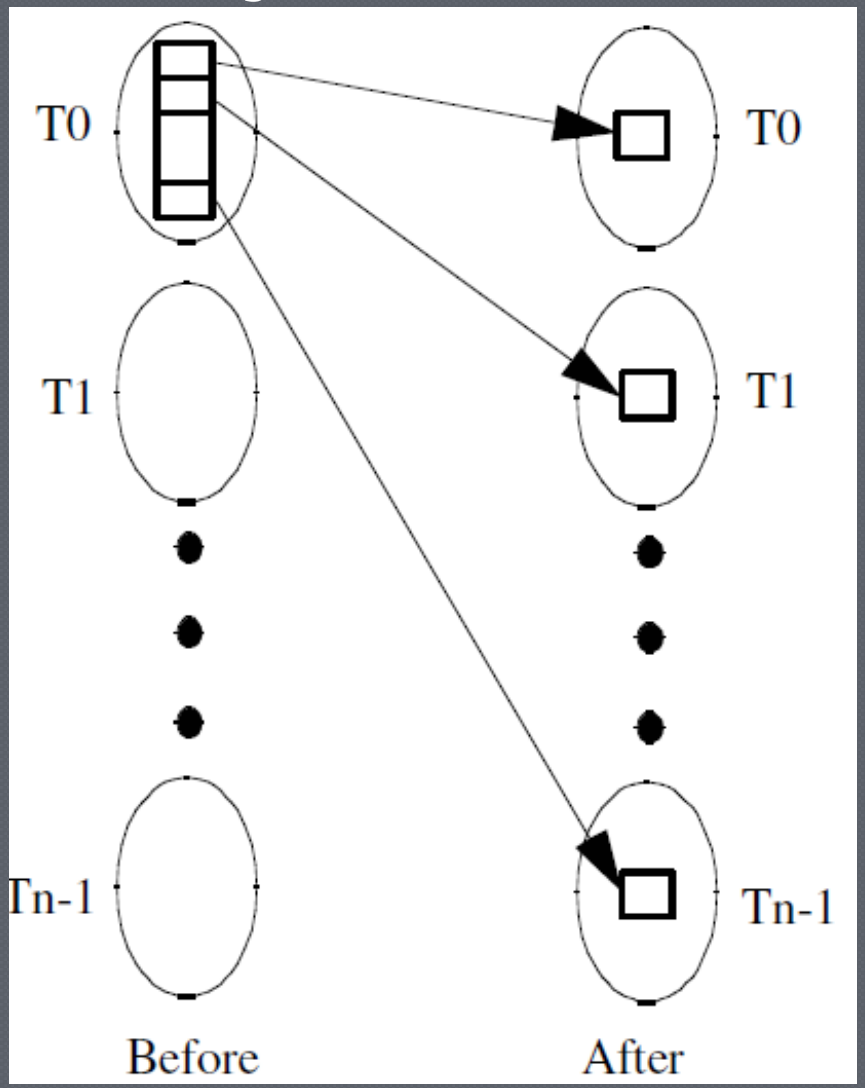
- scatter/gather:

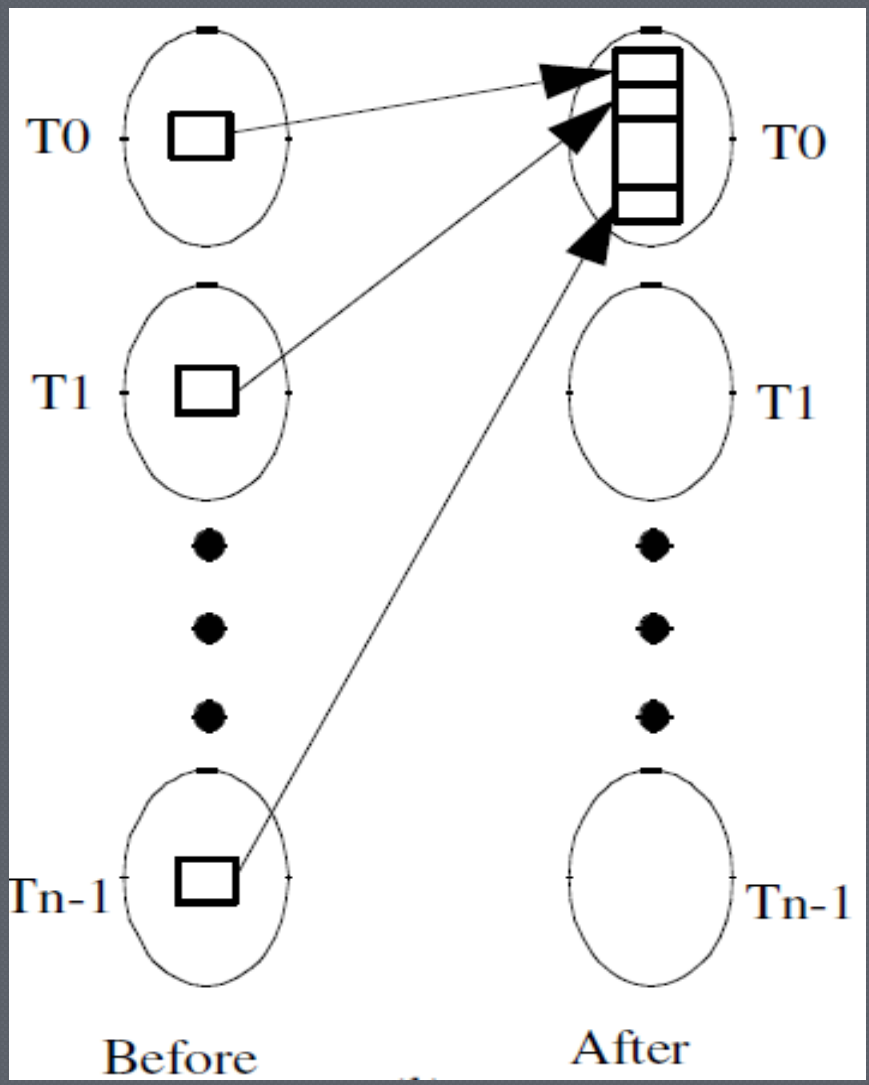
```
MPI_Scatter(sbuf,n,stype,rbuf,m,rtype,rt,communicator)
MPI_Gather(sbuf,n,stype,rbuf,m,rtype,rt,communicator)
```

  - Lệnh `Scatter`: phân phát một buffer lên tất cả các tác vụ khác. Buffer được chia thành `n` phần tử.
  - Lệnh `Gather`: tạo mới một buffer riêng cho mình từ các mảnh dữ liệu gộp lại.
    - `sbuf` : địa chỉ của buffer gửi.
    - `n` : số các phần tử gửi đến cho mỗi tác vụ (trường hợp scatter) hoặc số các phần tử trong buffer gửi (trường hợp gather).
    - `stype`: kiểu dữ liệu của các buffer gửi.
    - `rbuf` : địa chỉ của buffer nhận.
    - `m` : số phần tử dữ liệu trong buffer nhận (trường hợp scatter) hoặc số phần tử đã nhận từ mỗi một tác vụ gửi (trường hợp gather).
    - `rtype` : kiểu dữ liệu của các buffer nhận.
    - `rt` : rank của tác vụ gửi (trường hợp scatter) hoặc rank của tác vụ nhận (trường hợp gather).

3.2 Thư viện giao diện trao đổi thông điệp

- scatter/gather:





3.2 Thư viện giao diện trao đổi thông điệp

- Lệnh `Reduce()`:

  `MPI_Reduce(sbuf, rbuf, n, data_type, op, rt, communicator)`

  - `sbuf` : Địa chỉ của buffer gửi.
  - `rbuf` : Địa chỉ của buffer nhận.
  - `n` : số phần tử dữ liệu trong buffer gửi.
  - `data_type`: kiểu dữ liệu của buffer gửi.
  - `op` : phép toán rút gọn.
  - `rt` : rank của tác vụ gốc.

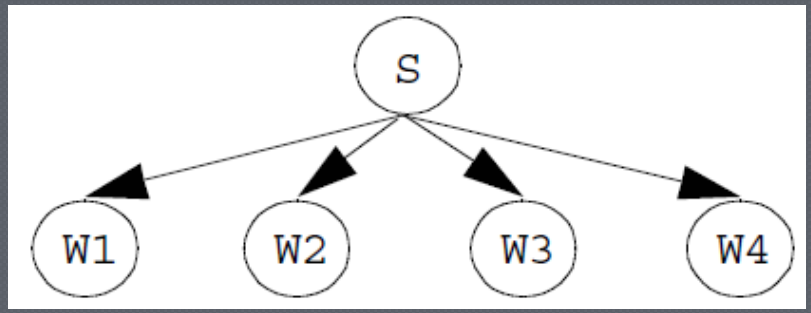
  - Các phép toán rút gọn:
    - `MPI_SUM` : phép tính tổng
    - `MPI_PROD` : phép nhân
    - `MPI_MIN` : tìm cực tiểu
    - `MPI_MAX` : tìm cực đại
    - `MPI_LAND` : Logic AND.
    - `MPI_LOR` : Logic OR.
  - Kết quả cuối cùng được trả về cho tác vụ gốc.

3.3 Máy ảo song song (Parallel Virtual Machine-PVM).

- PVM: là một tập hợp các hệ máy tính khác nhau được kết nối qua mạng và được điều khiển bởi một máy tính đơn trong hệ parallel.
- Mỗi một node máy tính trong mạng gọi là ***host***, các host có thể có một processor hoặc nhiều processor, host cũng có thể là một cluster được cài đặt phần mềm PVM.
- Hệ PVM bao gồm hai phần:
  - Bộ thư viện các hàm/thủ tục PVM.
  - Một chương trình daemon được cài trên tất cả các node trong hệ máy ảo.
- Một chương trình ứng dụng PVM được kết hợp một số các chương trình riêng lẻ, mỗi một chương trình đó được viết tương ứng cho một hoặc nhiều chu trình trong chương trình parallel. Các chương trình này được dịch (compile) để chạy cho mỗi một host. Các file chạy được đặt trên các host khác nhau.
- Một chương trình đặc biệt được gọi là tác vụ khởi đầu (initiating task) được khởi động bằng tay trên một host nào đó.
- initiating task sẽ kích hoạt tự động tất cả các tác vụ trên các host khác.
- Các tác vụ giống nhau có thể chạy trên các khoảng dữ liệu khác nhau, đây là mô hình Single Program Multiple Data (SPMD).

3.3 Máy ảo song song

- Trong trường hợp các tác vụ thực hiện các chức năng khác nhau, đây là mô hình Multiple Program Multiple Data (MPMD).
- Các chương trình có thể thực hiện theo các cấu trúc khác nhau mà không cần phải sửa đổi file nguồn, chỉ cần copy từ cấu trúc này sang cấu trúc khác rồi dịch và chạy chương trình.
- Cấu trúc chương trình PVM:
  - Cấu trúc phổ dụng nhất là cấu trúc hình sao (star). Node chính giữa gọi là supervisor (master), các node còn lại gọi là Workers (slaves).
    - Trong mô hình này, node master thực hiện initiating task sau đó kích hoạt tất cả các tác vụ khác trên các node slave.

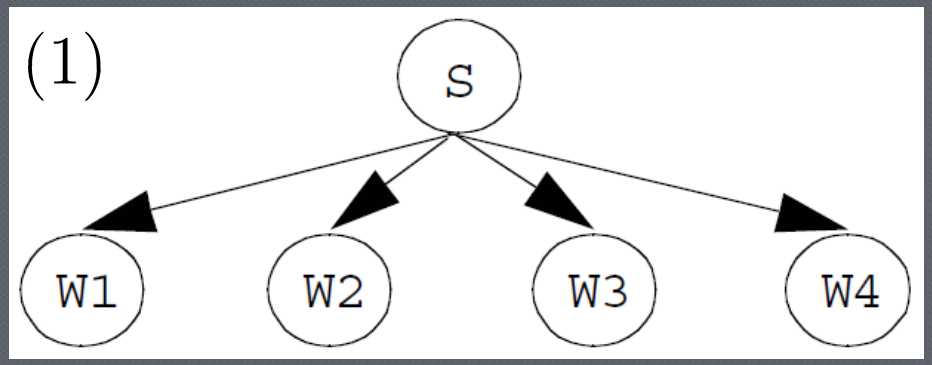


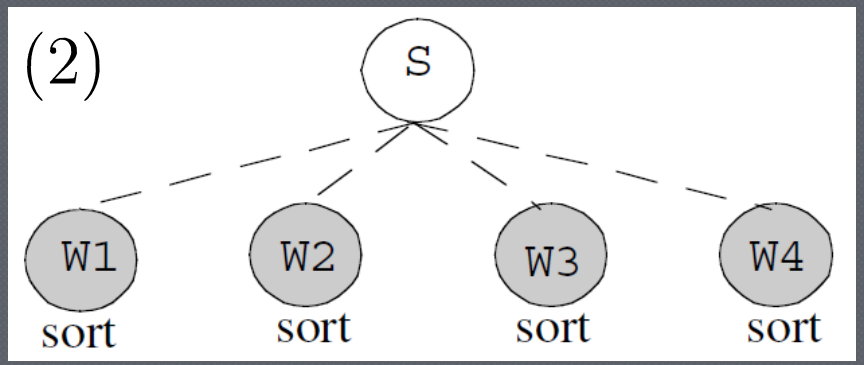
  - Cấu trúc cây (tree) - hierarchy: node master trên cùng thực hiện initiating task gọi là node gốc (root). Các node slave nằm trên các nhánh và chia thành các bậc khác nhau (level).

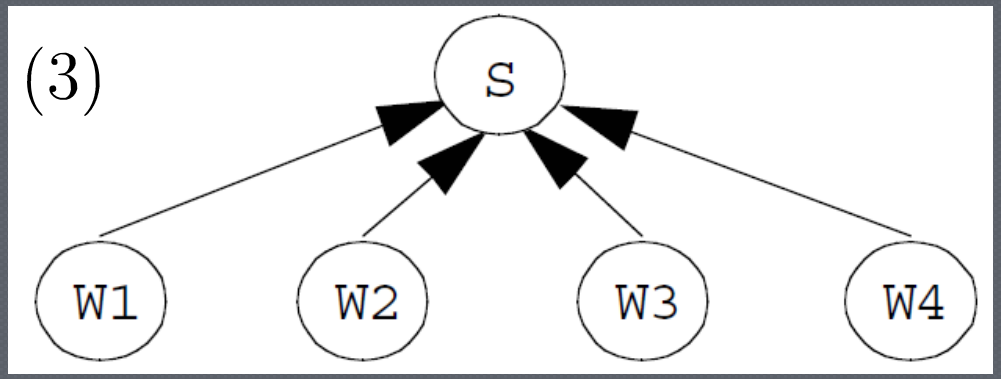
3.3 Máy ảo song song

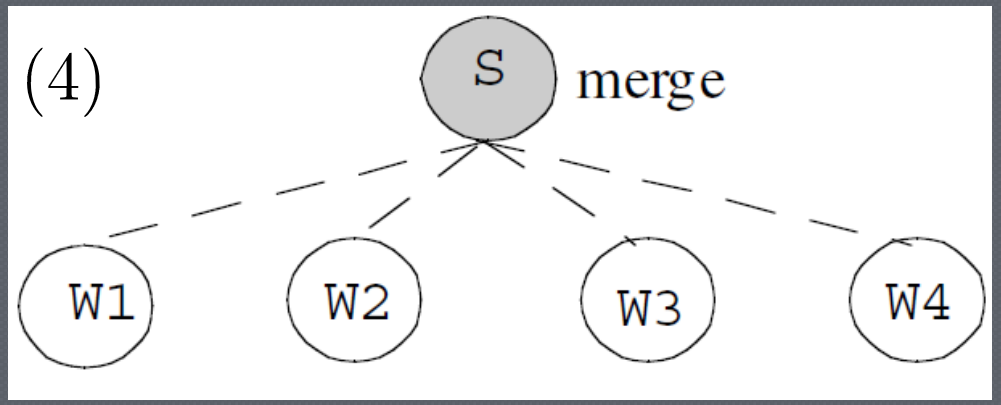
- Cấu trúc Supervisor–Workers hay Master-Slaves.
  - Là trường hợp đặc biệt của cấu trúc cây mà nó chỉ có một mức. Một master và nhiều slaves.
    - Node master được kích hoạt bằng tay tác vụ "initiating", nó tương tác trực tiếp với người sử dụng.
    - Node master kích hoạt các node slave, gán các công việc cho các node slave này, cuối cùng là gom kết quả từ các node slave về node master.
  - Node slave thực hiện công việc tính toán, các node này có thể hoạt động một cách độc lập, hoặc phụ thuộc lẫn nhau.
  - Nếu như node slave hoạt động phụ thuộc lẫn nhau, nó có thể trao đổi thông tin trực tiếp với nhau để hoàn thành công việc, sau đó gửi kết quả về master.
  - Ví dụ: bài toán sắp xếp các phần tử trong mảng.
    - Node master chia mảng thành các phần con có số phần tử bằng nhau.
    - Mỗi một phần con được gán cho một node slave.
    - Các node slave sẽ thực hiện việc sắp xếp một cách độc lập các phần tử trong phần con của mảng. Sau đó gửi kết quả về node master.
    - Cuối cùng node master tập hợp các phần con đã được sắp xếp từ các node slave, trộn các chuỗi lại với nhau thành một chuỗi hoàn chỉnh.

3.3 Máy ảo song song



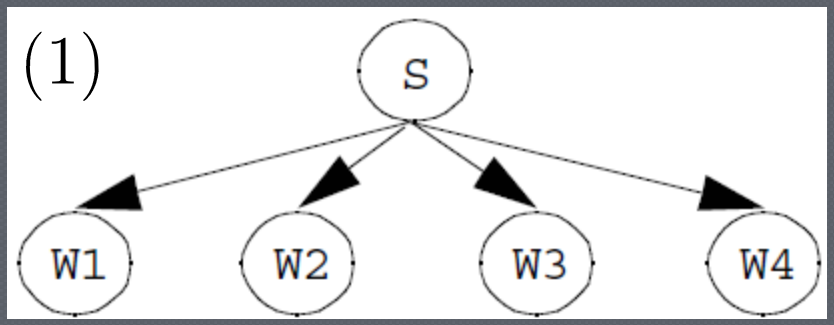


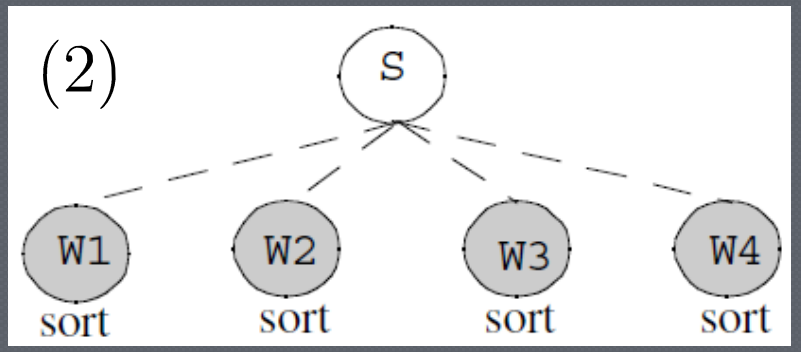


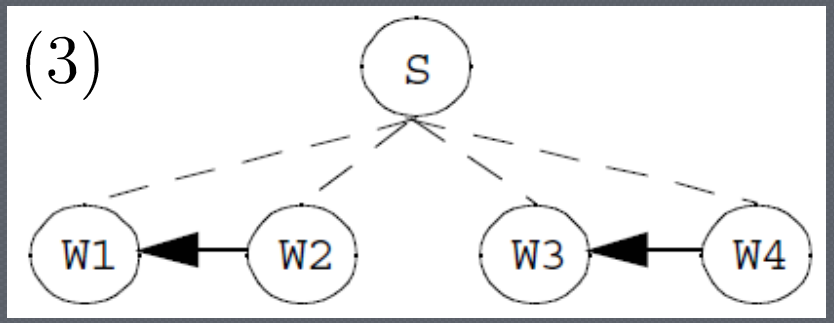


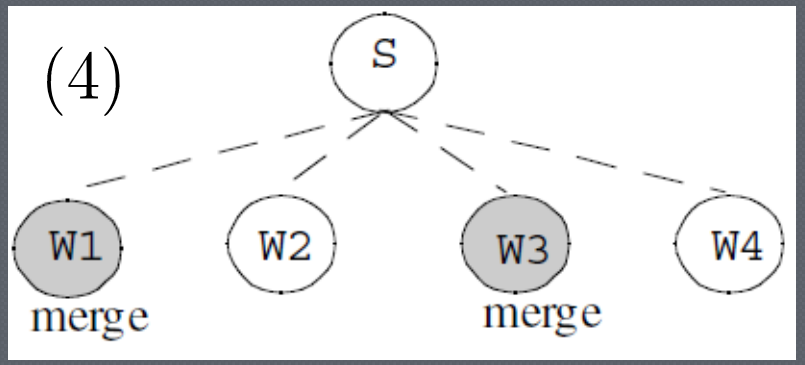
3.3 Máy ảo song song

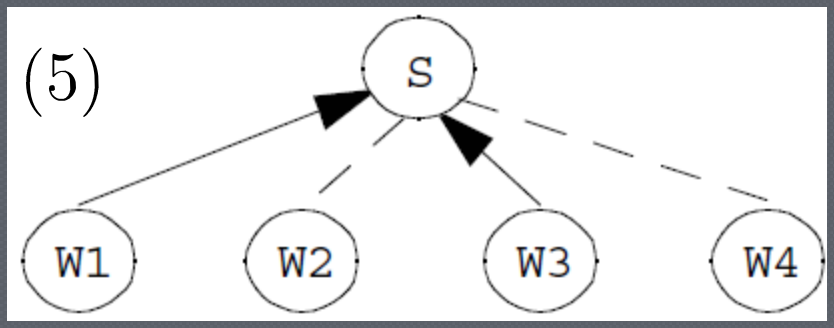
- Cách trộn các chuỗi trước khi gửi về node master:

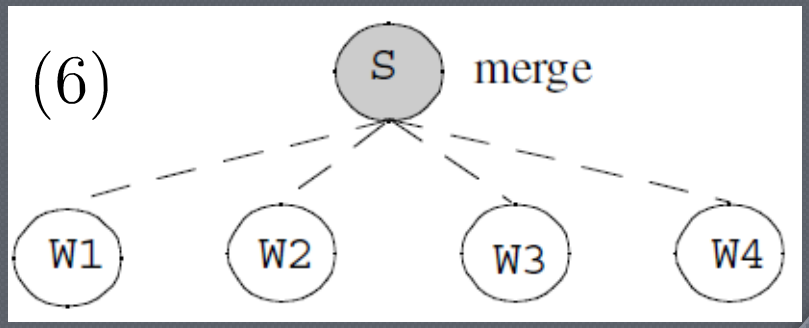






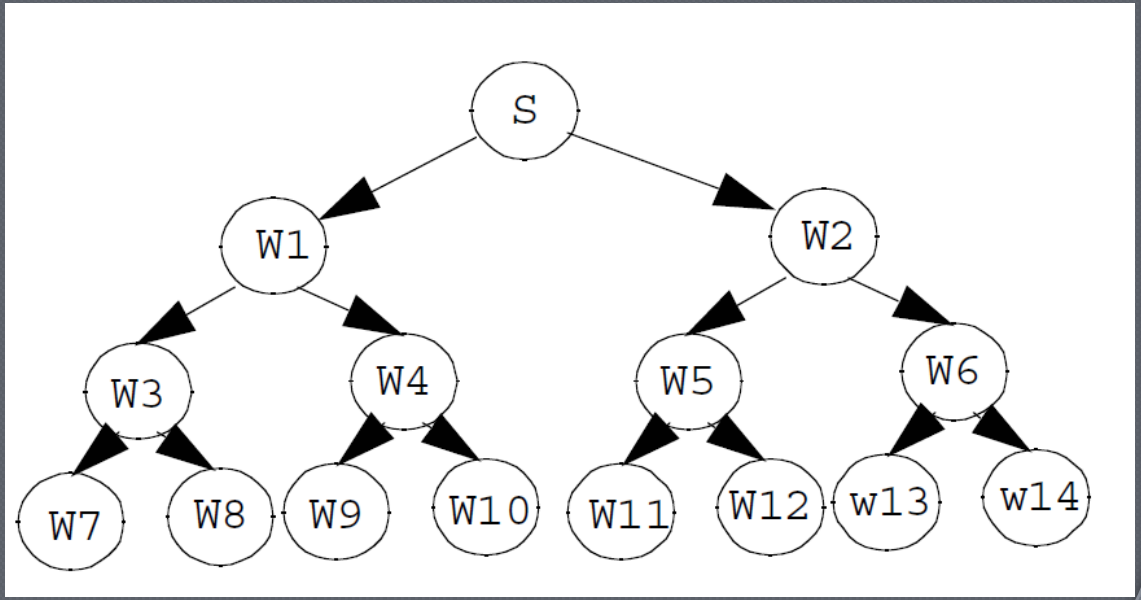






3.3 Máy ảo song song

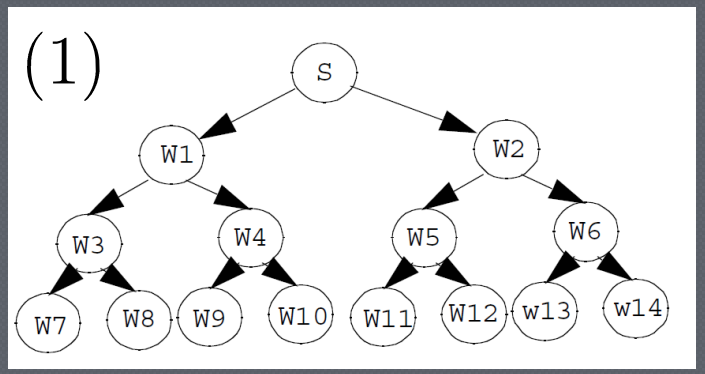
- Cấu trúc cây Hierarchy.
  - Sự khác biệt giữa cấu trúc hình sao và cấu trúc cây đó là mỗi một node slave có thể đóng vai trò là một node master thứ cấp, nó có thể tạo mới các node slave thứ cấp.
  - Node master thực hiện tác vụ initiating gọi là node bậc một hay node gốc, nó tạo ra các node slave kế tiếp bậc hai. Các node bậc 2 tạo ra các node bậc 3…
  - Các "lá" của cây là các node có bậc thấp nhất.

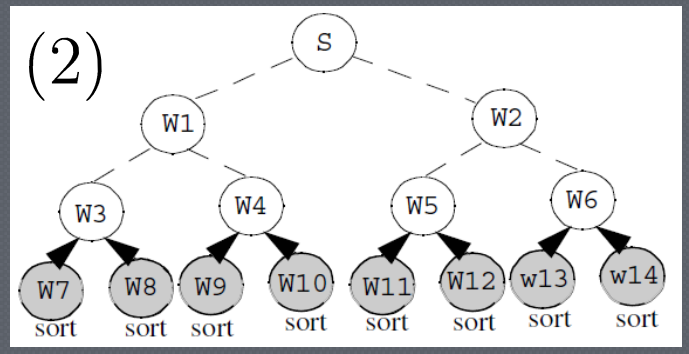


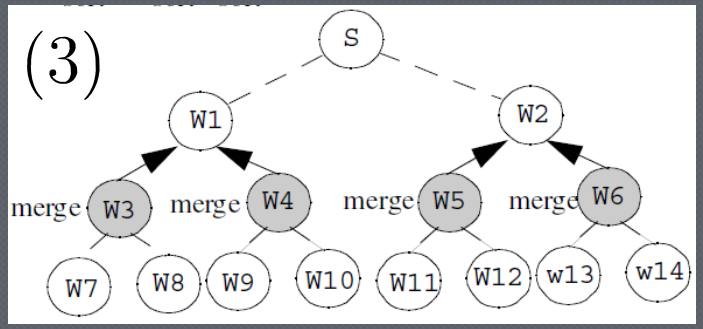
- S: bậc 1 (gốc).
- W1, W2: bậc 2.
- W3,W4,W5,W6: bậc 3.
- W7, W8 … : các lá.

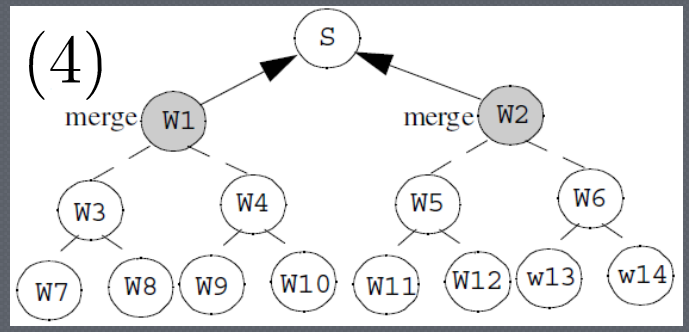
3.3 Máy ảo song song

- Ví dụ bài toán sắp xếp: Mỗi một node chia mảng con thành 2 phần.

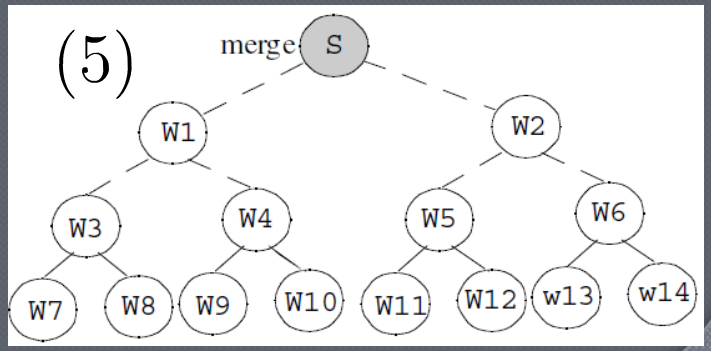








- S chia mảng thành 2 phần và gửi cho W1,W2
- W1 chia phần mảng con thành 2 phần và gửi cho W3, W4
- W3 chia phần mảng con thành 2 phần và gửi cho W7, W8…



3.3 Máy ảo song song

- Tạo tác vụ.
  - Tác vụ trong PVM có thể được tạo bằng tay, hoặc sinh ra từ các tác vụ khác.
  - Tác vụ initiating được tạo ra từ đầu trong chương trình để thực hiện trên node master, nó là một tác vụ tĩnh . Các tác vụ khác được tạo ra trong quá trình chạy chương trình từ các tác vụ khác được gọi là tác vụ động.
  - Hàm `pvm_spawn()`: tạo một tác vụ động.
    - Tác vụ mẹ (parent): tác vụ gọi hàm `pvm_spawn()`.
    - Tác vụ con (child): được tạo ra từ hàm `pvm_spawn()`.
  - Điều kiện thực hiện lệnh `pvm_spawn()`.
    - Xác định node để thực hiện tác vụ con.
    - Đường dẫn đến file chạy trên máy đó.
    - Số bản sao của tác vụ con được tạo ra.
    - Mảng các tham số cho các tác vụ con.
  - Để nhận dạng các tác vụ, người ta gán nhãn hay gán chỉ số cho từng tác vụ. Nhãn hay chỉ số của tác vụ là một số nguyên duy nhất: task ID hay TID.

3.3 Máy ảo song song

- Các hàm liên quan đến chỉ số tác vụ.
  - Hàm `pvm_myid()`: cho biết chỉ số của chính nó (của chính tác vụ gọi hàm).
    - Ví dụ: `mytid = pvm_mytid();`
  - Hàm `pvm_spawn( …, …, …, …,… ,&tid)`: hàm tạo các tác vụ con, hàm trả lại một mảng "`tid`" chứa chỉ số của các tác vụ con vừa được tạo ra.
  - Hàm `pvm_parent()`: cho biết chỉ số của tác vụ mẹ
    - Ví dụ: `my_parent_tid = pvm_parent();`
  - Hàm `pvm_tidtohost(id)`: cho biết chỉ số của host mà chương trình daemon đang chạy. Ví dụ: `daemon_tid = pvm_tidtohost(id);`
- Tạo tác vụ động:
  - Hàm `pvm_spawn()`:

```
num = pvm_spawn(Child,Arguments,Flag,Where,HowMany,&Tids);
```

    - `Child`: Tên file chạy (file phải được đặt trên host mà nó thực hiện).
    - `Arguments`: mảng các tham số chương trình
    - `Flag`: bằng 0 thì các host chạy sẽ được chọn tự động bởi chương trình PVM. Bằng 1 nếu tác vụ mới sẽ chạy trên host được định nghĩa ở tham số `Where`.
    - `Where`: tên host để thực hiện các tác vụ mới.
    - `HowMany`: số tác vụ con được tạo mới.
    - `Tids`: mảng chỉ số các tác vụ con.

VD: `n1 = pvm_spawn("/user/rewini/worker", 0, 1, "homer", 2, &tid1);`

3.3 Máy ảo song song

- Nhóm tác vụ.
  - Các tác vụ có thể gia nhập hoặc rời khỏi nhóm. Mỗi một tác vụ có thể gia nhập nhiều nhóm khác nhau.
  - Mỗi một tác vụ trong một nhóm được đánh số thứ tự bắt đầu từ số 0.
    - Hàm `pvm_joingroup()`: Đưa tác vụ gia nhập nhóm "`group_name`". Hàm trả lại chỉ số của tác vụ trong nhóm.

```
i = pvm_joingroup(group_name);
```

    - Hàm `pvm_lvroup()`: Tách tác vụ ra khỏi nhóm "`group_name`": Hàm trả lại giá trị âm nếu như có lỗi (error).

```
info = pvm_lvgroup(group_name);
```

    - Hàm `pvm_gsize()`: Số các tác vụ trong nhóm "`group_name`":

```
nt = pvm_gsize(group_name);
```

    - Hàm `pvm_gettid()`: Cho biết TID của một tác vụ trong nhóm khi biết số thứ tự của tác vụ đó:

```
tid = pvm_gettid("slave",1);
```

    - Hàm `pvm_getinst()`: Cho biết số thứ tự trong nhóm của một tác vụ khi biết TID của tác vụ đó:

```
inst = pvm_getinst("slave",100)
```

3.3 Máy ảo song song

- Ví dụ: Có các tác vụ :T0, T1, T2, và T3 với TID của các tác vụ này là 200, 100, 300, và 400.
  - Tác vụ T0,T1,T2 gọi các hàm:

```
i1 = pvm_joingroup("slave");          /* i1 = 0 */
i2 = pvm_joingroup("slave");          /* i2 = 1 */
i3 = pvm_joingroup("slave");          /* i3 = 2 */
```

  - Tác vụ T1 gọi hàm rời khỏi nhóm, chỉ số "1" của nhóm sẽ được gán cho lệnh `pvm_joingroup` lần sau.

```
info = pvm_lvgroup("slave")
```

  - Lệnh `size=pvm_gsize("slave")` sẽ trả lại giá trị kích thước nhóm là 2.
  - Tác vụ T3 gọi hàm gia nhập nhóm

```
i4 = pvm_joingroup("slave");          /* i4 = 1 */
```

  - Tác vụ T1 gọi hàm gia nhập nhóm

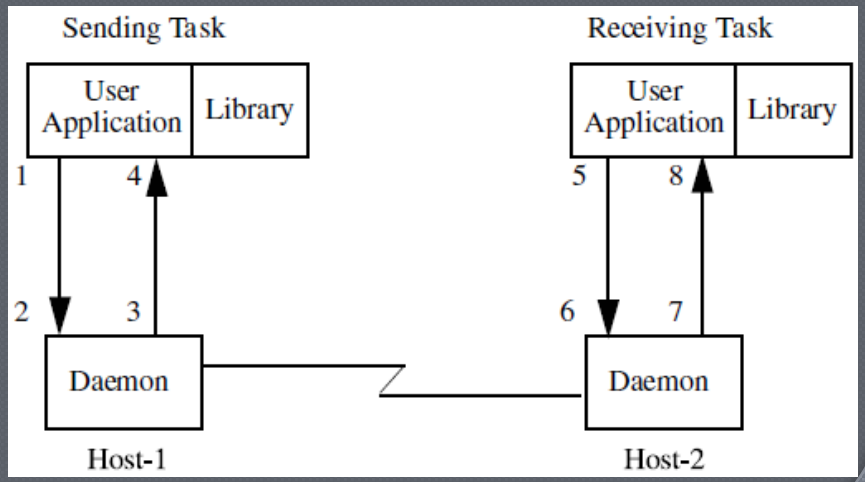
```
i5 = pvm_joingroup("slave");          /* i5 = 3 */
```

  - Lệnh `tid = pvm_gettid("slave",1)` sẽ trả lại giá trị TID =400 của T3.
  - Lệnh `inst = pvm_getinst("slave",100)` sẽ trả lại chỉ số nhóm bằng 3 của tác vụ T1 có TID = 100.

3.3 Máy ảo song song

- Trao đổi thông tin giữa các tác vụ.
  - Trao đổi thông tin giữa các tác vụ trong hệ PVM cũng sử dụng kiểu message passing. Sử dụng cặp lệnh: send/receive thông qua chương trình daemon đang chạy trên mỗi node.
  - Daemon xác định điểm đích để message gửi tới.
    - Nếu message được chuyển đến một tác vụ nội bộ (trên cùng một node với daemon), daemon sẽ chuyển trực tiếp message đó.
    - Nếu message được chuyển đến một tác vụ trên một node khác, daemon sẽ gửi message đó cho daemon của node nhận thông qua mạng.
    - Một message có thể được gửi đến một hoặc nhiều node nhận.
    - Message có thể nhận theo kiểu blocking hoặc nonblocking.

- Lệnh send được thực hiện tại điểm 1.
- Message chuyển đến daemon tại điểm 2
- 3-4: điều khiển trở về cho người sử dụng khi đã chuyển dữ liệu đến 2.
- 5-6: là lệnh nhận từ node receiver
- 7-8 : điều khiển trở về cho người sử dụng.



3.3 Máy ảo song song

- Một tác vụ sender cần phải thực hiện 3 bước:
  - Chuẩn bị buffer gửi.
  - Message phải được nén lại trong buffer này.
  - Message được gửi đầy đủ đến các node nhận.
- Một tác vụ receiver cần phải thực hiện 2 bước:
  - Message đã được nhận đầy đủ.
  - Các phần tử của message đã nhận phải được mở gói trên buffer nhận.

Tạo buffer cho message:

- Lệnh `pvm_initsend()` : tạo một buffer, hàm trả lại ID của buffer đó.

```
bufid = pvm_initsend(encoding_option);
```

  tham số `encoding_option`: có giá trị ngầm định bằng 0, dữ liệu được mã hóa. Bằng 1 thì dữ liệu không được mã hóa.
- Lệnh `pvm_mkbuf()`: Tạo buffer, được sử dụng có hiệu quả cao khi trong chương trình cần phải sử dụng nhiều buffer cho nhiều message.

```
bufid = pvm_mkbuf(encoding_option);
```

- Phiên bản PVM 3: chỉ cho phép một buffer gửi và một buffer nhận cùng hoạt động tại một thời điểm.
  - Lệnh đặt active: `pvm_setsbuf(bufid)` và `pvm_setrbuf(bufid)`.
    hàm trả lại chỉ số ID của buffer và ghi lại trạng thái của buffer trước đó.
  - Lệnh lấy chỉ số ID của buffer: `pvm_getsbuf()` và `pvm_getrbuf()`.

3.3 Máy ảo song song

- Nén dữ liệu (Data packing): `pvm_pk*()`.
  - Nén một mảng dữ liệu để đưa vào buffer gửi.
  - Các lệnh nén có 3 đối số:
    - Biến con trỏ mảng trỏ tới vị trí của phần tử đầu tiên trong mảng.
    - Số các phần tử trong mảng sẽ được gói.
    - Bước nhảy số phần tử trong mảng.
  - Các hàm nén dữ liệu có tên tương ứng với kiểu dữ liệu như số nguyên, số thực, chuỗi ký tự…
    - `pvm_pkint(), pvm_pkfloat(), pvm_pkstr() …`
- Gửi message:
  - Quá trình gửi message là quá trình không đồng bộ, tức là nó không chờ cho đến khi tác vụ nhận hoàn thành.
  - Sau khi buffer được khởi tạo, quá trình gói dữ liệu đã thực hiện xong thì dữ liệu ở trạng thái sẵn sàng (active) được gửi đi.
  - Lệnh gửi đến một node nhận có Task ID là "`tid`":

```
info = pvm_send(tid, tag);
```

    - Tham số "`tag`" là nhãn của message, hàm trả lại giá trị âm nếu bị lỗi.

3.3 Máy ảo song song

- Lệnh gửi đến nhiều node nhận:

```
info = pvm_mcast(tids,n,tag);
```

  - Tham số `n` là số các tác vụ nhận.
- Lệnh gửi đến nhóm:

```
info = pvm_bcast(group_name, tag);
```

  - Tham số `n` là số các tác vụ nhận.
- Lệnh nén và gửi:

```
info = pvm_psend(tid, tag, my_array, n, int)
```

  - Nén một mảng tên là `my_array` gồm `n` số nguyên vào một message có nhãn là `tag`, sau đó gửi đến TID.

- Nhận message:
  - Nhận kiểu blocking: chờ cho đến khi message đã được nhận.

```
bufid = pvm_recv(tid, tag)
```

    - Nếu chọn `tid = -1` hoặc `tag = -1` thì sẽ nhận bất kỳ message nào đó vừa được gửi đến.
  - Nhận kiểu nonblocking: không phải chờ cho đến khi message đã được nhận

```
bufid = pvm_nrecv(tid, tag)
```

    - Nếu message chưa được gửi đến thì `bufid` có giá trị bằng 0.

3.3 Máy ảo song song

- Nhận kiểu timeout: không phải chờ cho đến khi message đã được nhận

```
bufid = pvm_trecv(tid, tag, timeout)
```

  - Hàm nhận sẽ bị khóa trong một khoảng thời gian `timeout` để chờ message được gửi đến. Nếu message không đến thì `bufid=0`.
- Lệnh nhận và giải nén: `pvm_precv()`

```
info = pvm_precv(tid, tag, my_array, len, datatype,
       &src,&atag,&alen)
```

  - Task ID của node gửi, nhãn của message và độ dài của message được gửi đến được gán vào các biến `src,atag,alen`.
- Giải nén dữ liệu:
  - Các lệnh giải nén hoàn toàn tương tự với các lệnh nén, tên hàm có thêm chữ "u": `pvm_upkint(), pvm_upkfloat(), pvm_upkstr()` …
  - Ví dụ:
    - Lệnh giải nén một chuỗi ký tự.

```
info = pvm_upkstr(string)
```

    - Lệnh giải nén một mảng gồm `n` phần tử, bước nhảy giữa các phần tử s = 1.

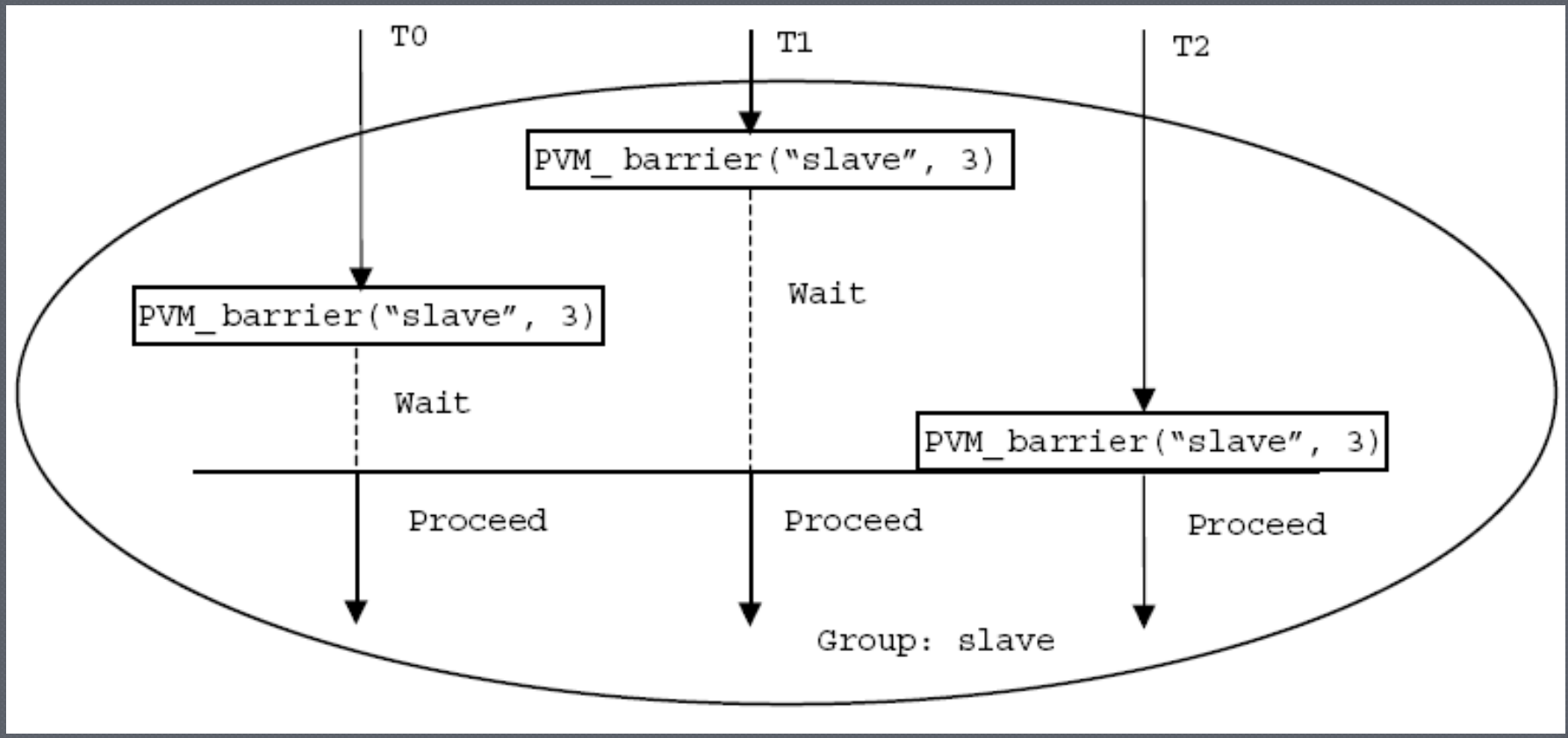
```
info = pvm_upkint(my_array, n, 1)
```

3.3 Máy ảo song song

- Đồng bộ hóa các tác vụ:
  - Lệnh `pvm_barrier()`: Tất cả các tác vụ đều phải chờ tại barrier.

```
info = pvm_barrier(group_name, ntasks)
```

    - `ntasks` là số các thành viên trong nhóm thực hiện lệnh barrier.



3.4 Thiết kế và xây dựng một chương trình

3.4 Thiết kế và xây dựng một chương trình (giải một bài toán (NP-complete) sử dụng MPI và C.

https://computing.llnl.gov/tutorials/mpi/

Practical MPI Programming:

http://www.redbooks.ibm.com/redbooks/pdfs/sg245380.pdf

- Đoạn chương trình kiểm tra hệ thống: `MPI_SUCCESS`

```
#include "mpi.h"
#include <stdio.h>
int main(argc,argv)
int argc;
   char *argv[]; {
   int numtasks, rank, rc;
   rc = MPI_Init(&argc,&argv);
   if (rc != MPI_SUCCESS) {
           printf (" Loi khoi dong MPI. Ket thuc.\n");
           MPI_Abort(MPI_COMM_WORLD, rc);
  }
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);     /** số processor */
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);         /** rank hiện thời */
   printf ("Number of tasks= %d My rank= %d\n", numtasks,rank);
   /*** bắt đầu thực hiện phần chính của chương trình ***/
   MPI_Finalize();
}
```

3.4 Thiết kế và xây dựng một chương trình

- Đoạn chương trình kiểu blocking: `send/recv`

```
#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();
}
```

3.4 Thiết kế và xây dựng một chương trình

- Đoạn chương trình kiểu non-blocking: `Isend/Irecv`

```
#include "mpi.h"
#include <stdio.h>

int main(argc,argv)
int argc;
char *argv[];  {
int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
MPI_Request reqs[4];
MPI_Status stats[4];
MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
prev = rank-1;
next = rank+1;
if (rank == 0)  prev = numtasks - 1;
if (rank == (numtasks - 1)) next = 0;

MPI_Irecv(&buf[0], 1, MPI_INT, prev, tag1, MPI_COMM_WORLD, &reqs[0]);
MPI_Irecv(&buf[1], 1, MPI_INT, next, tag2, MPI_COMM_WORLD, &reqs[1]);

MPI_Isend(&rank, 1, MPI_INT, prev, tag2, MPI_COMM_WORLD, &reqs[2]);
MPI_Isend(&rank, 1, MPI_INT, next, tag1, MPI_COMM_WORLD, &reqs[3]);

      /**  Thực hiện công việc */

MPI_Waitall(4, reqs, stats);
MPI_Finalize();
}
```

3.4 Thiết kế và xây dựng một chương trình

- Bài toán tính tích hai ma trận : C = A*B

$$c_{ik} = \sum_j (a_{ij} \cdot b_{jk})$$

  - Chương trình viết dạng tuần tự chạy trên một processor

```
#include <stdio.h>
#include "mpi.h"
#define nra 62                  /* Số hàng của ma trận A */
#define nca 15                  /* Số cột của ma trận A */
#define ncb 7

main(int argc, char **argv) {
int i,j,k;
double a[nra][nca], b[nca][ncb], c[nra][ncb];
    for (i=0; i<nra; i++)
       for (j=0; j<nca; j++)
         a[i][j]= i+j;
    for (i=0; i<nca; i++)
       for (j=0; j<ncb; j++)
         b[i][j]= i*j;
```

3.4 Thiết kế và xây dựng một chương trình

```
 /* Tính c */

for (k=0; k<ncb; k++)
      for (i=0; i<nra; i++) {
            c[i][k] = 0.0;
            for (j=0; j<nca; j++)
                  c[i][k] = c[i][k] + a[i][j]*b[j][k];
      }

 /* In ket qua */

   printf(”Ket qua tich hai ma tran\n");
   for (i=0; i<nra; i++) {
   printf("\n");
   for (j=0; j<ncb; j++)
       printf("%6.2f   ", c[i][j]);
      }
    printf ("\n");
    }
}
```

3.4 Thiết kế và xây dựng một chương trình

- Chương trình viết dạng parallel sử dụng MPI

```
#include <stdio.h>
#include "mpi.h"
#define nra 62              /* Số hàng của ma trận A */
#define nca 15              /* Số cột của ma trận A */
#define ncb 7
#define MASTER 0
#define FROM_MASTER 1   /* Kiểu message */
#define FROM_WORKER 2
MPI_Status status;
main(int argc, char **argv) {
    numtasks,     /* tổng số tác vụ */
    taskid,       /* chỉ số task */
    numworkers,   /* số task slave */
    source,       /* task id của message nguồn */
    dest,         /* task id của message đíchn */
    nbytes,       /* số byte trong một message */
    mtype,        /* kiểu dữ liệu của message */
    intsize,      /* kích thước của số nguyên theo bytes */
    dbsize,       /* kích thước của số thực theo bytes */
    rows,         /* hàng của ma trận A */
```

3.4 Thiết kế và xây dựng một chương trình

```
    averow, extra, offset,        /* các biến phụ */
    i, j, k,                      /* chỉ số chạy */
    count;
   double a[nra][nca], b[nca][ncb], c[nra][ncb];
   intsize = sizeof(int);
   dbsize = sizeof(double);
/*************************************/
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &taskid);
  MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
  numworkers = numtasks-1;
/************* master task **************/
  if (taskid == MASTER) {
     printf("Number of worker tasks = %d\n",numworkers);
     for (i=0; i<nra; i++)
        for (j=0; j<nca; j++)
          a[i][j]= i+j;
     for (i=0; i<nca; i++)
        for (j=0; j<ncb; j++)
          b[i][j]= i*j;
```

3.4 Thiết kế và xây dựng một chương trình

```
/* send matrix data to the worker tasks */
averow = nra/numworkers;
extra = nra%numworkers;
offset = 0;
mtype = FROM_MASTER;
for (dest=1; dest<=numworkers; dest++) {
  rows = (dest <= extra) ? averow+1 : averow;
  printf("   sending %d rows to task %d\n",rows,dest);
  MPI_Send(&offset, 1, MPI_INT, dest, mtype,
           MPI_COMM_WORLD);
  MPI_Send(&rows, 1, MPI_INT, dest, mtype,
           MPI_COMM_WORLD);
  count = rows*nca;
  MPI_Send(&a[offset][0], count, MPI_DOUBLE, dest, mtype,
           MPI_COMM_WORLD);
  count = nca*ncb;
  MPI_Send(&b, count, MPI_DOUBLE, dest, mtype,
           MPI_COMM_WORLD);
  offset = offset + rows;
  }
```

3.4 Thiết kế và xây dựng một chương trình

```
/* wait for results from all worker tasks */
mtype = FROM_WORKER;
for (i=1; i<=numworkers; i++){
  source = i;
  MPI_Recv(&offset, 1, MPI_INT, source, mtype,
           MPI_COMM_WORLD, &status);
  MPI_Recv(&rows, 1, MPI_INT, source, mtype,
           MPI_COMM_WORLD, &status);
  count = rows*ncb;
  MPI_Recv(&c[offset][0], count, MPI_DOUBLE, source, mtype,
           MPI_COMM_WORLD, &status);
}
/* In ket qua */
  printf("Here is the result matrix\n");
  for (i=0; i<nra; i++) {
     printf("\n");
     for (j=0; j<ncb; j++)
        printf("%6.2f   ", c[i][j]);
  }
  printf ("\n");
}  /* end of master section */
```

3.4 Thiết kế và xây dựng một chương trình

```
/************** worker task ****************/
if (taskid > MASTER) {
  mtype = FROM_MASTER;
  source = MASTER;
  printf ("Master =%d, mtype=%d\n", source, mtype);
  MPI_Recv(&offset, 1, MPI_INT, source, mtype,
           MPI_COMM_WORLD, &status);
  printf ("offset =%d\n", offset);
  MPI_Recv(&rows, 1, MPI_INT, source, mtype,
           MPI_COMM_WORLD, &status);
  printf ("row =%d\n", rows);
  count = rows*nca;
  MPI_Recv(&a, count, MPI_DOUBLE, source, mtype,
           MPI_COMM_WORLD, &status);
  printf ("a[0][0] =%e\n", a[0][0]);
  count = nca*ncb;
  MPI_Recv(&b, count, MPI_DOUBLE, source, mtype,
      MPI_COMM_WORLD, &status);
  printf ("b =\n");
```

3.4 Thiết kế và xây dựng một chương trình

```
   for (k=0; k < ncb; k++)
     for (i=0; i < rows; i++) {
       c[i][k] = 0.0;
       for (j=0; j < nca; j++)
         c[i][k] = c[i][k] + a[i][j] * b[j][k];
       }
   mtype = FROM_WORKER;
   printf ("after computing\n");
   MPI_Send(&offset, 1, MPI_INT, MASTER, mtype,
            MPI_COMM_WORLD);
   MPI_Send(&rows, 1, MPI_INT, MASTER, mtype,
            MPI_COMM_WORLD);
   MPI_Send(&c, rows*ncb, MPI_DOUBLE, MASTER, mtype,
            MPI_COMM_WORLD);
   printf ("after send\n");
 }
 /* end of worker */
   MPI_Finalize();
}
/* end of main */
```

3.4 Thiết kế và xây dựng một chương trình

- Ví dụ tính tổng của một vector $A_i = A_i \cdot \sum_j (A_j)$

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
main(int argc, char **argv){
     int rank, size, myn, i, N;
     double *vector, *myvec, sum, mysum, total;
     MPI_Init(&argc, &argv );
     MPI_Comm_rank(MPI_COMM_WORLD, &rank);
     MPI_Comm_size(MPI_COMM_WORLD, &size);
if (rank == 0) {
     printf("Enter the vector length : ");
     scanf("%d", &N);
     vector = (double *)malloc(sizeof(double) * N);
     for (i = 0, sum = 0; i < N; i++)
            vector[i] = 1.0;
     myn = N / size;
}
MPI_Bcast(&myn,1,MPI_INT,0,MPI_COMM_WORLD);
```

3.4 Thiết kế và xây dựng một chương trình

```
  myvec = (double *)malloc(sizeof(double)*myn);

  MPI_Scatter(vector, myn, MPI_DOUBLE, myvec, myn,
              MPI_DOUBLE, 0, MPI_COMM_WORLD );

  for (i = 0, mysum = 0; i < myn; i++)
      mysum += myvec[i];

  MPI_Allreduce(&mysum, &total, 1, MPI_DOUBLE, MPI_SUM,
              MPI_COMM_WORLD );

  for (i = 0; i < myn; i++)
      myvec[i] *= total;

  MPI_Gather(myvec, myn, MPI_DOUBLE, vector, myn, MPI_DOUBLE,
              0, MPI_COMM_WORLD );
  if (rank == 0)
      for (i = 0; i < N; i++)
              printf("[%d] %f\n", rank, vector[i]);
  MPI_Finalize();
  return 0;
}
```