TRƯỜNG ĐẠI HỌC BÁCH KHOA HÀ NỘI

VIỆN CÔNG NGHỆ THÔNG TIN VÀ TRUYỀN THÔNG

——————— * ———————

BÁO CÁO HỌC PHẦN:

PROJECT II

ĐỀ TÀI: TÌM HIỂU VỀ MÔ HÌNH LẬP TRÌNH SONG SONG MPI VÀ CÁC THUẬT TOÁN TRONG GIAO TIẾP TẬP THỂ.

Sinh viên thực hiện : **Lê Đình Mạnh** - 20162644

Lớp : **CNTT2.02**

Giáo viên hướng dẫn : **TS. Phạm Đăng Hải**

Hà Nội, tháng 4 năm 2019

Mục lục

I. Giới thiệu về MPI:

1. Mở đầu:

Thông thường hiện nay, hầu hết các chương trình tính toán đều được thiết kế để chạy trên 1 lõi (singlecore), đó là cách tính toán tuần tự, không tận dụng hết được tài nguyên trên các hệ thống máy tính (cluster) hoặc các cpu đa lõi (multicore). Vậy nên chúng ta cần phải tiến hành song song hóa các chương trình này để tận dụng được tối đa các tài nguyên đó. Ưu điểm của lập trình song song là thực hiện nhiều tác vụ cùng một lúc. Việc lập trình song song có thể được tiến hành thông qua việc sử dụng các hàm trong thư viện (vd: omp.h hay mpi.h, …)
Hiện nay có rất nhiều mô hình lập trình song song như mô hình đa luồng (multithread), truyền thông điệp (message passing), song song dữ liệu (data parallel), lai (hybrid),... Các loại mô hình này được phân chia dựa theo hai tiêu chí là tương tác giữa các tiến trình và cách xử lý bài toán.
Theo tiêu chí thứ nhất thì chúng ta có 2 loại mô hình song song chủ yếu là mô hình chia sẻ bộ nhớ (shared memory) hoặc truyền thông điệp (message passing).
Theo tiêu chí thứ hai, chúng ta cũng có 2 loại mô hình là song song hóa tác vụ (task parallelism) và song song hóa dữ liệu (data parallelism).

- Với mô hình bộ nhớ chia sẻ, tất cả các xử lý đều truy cập một dữ liệu chung thông qua vùng nhớ dùng chung.

- Với mô hình truyền thông điệp thì mỗi xử lý đều có riêng bộ nhớ cục bộ của nó, các xử lý trao đổi dữ liệu với nhau thông qua hai phương thức gửi và nhận thông điệp.
- Song song tác vụ là phương thức phân chia các tác vụ khác nhau đến các nút tính toán khác nhau, dữ liệu được sử dụng bởi các tác vụ có thể hoàn toàn giống nhau.
- Song song dữ liệu là phương thức phân phối dữ liệu tới các nút tính toán khác nhau để được xử lý đồng thời, các tác vụ tại các nút tính toán có thể hoàn toàn giống nhau.

Mô hình truyền thông điệp là một trong những mô hình được sử dụng rộng rãi nhất trong tính toán song song hiện nay. Nó thường được áp dụng trong những hệ thống phân tán (distributed system). Các đặc trưng mô hình này:

- Các bộ xử lý (processor) sử dụng vùng nhớ cục bộ riêng của chúng trong suốt quá trình tính toán.
- Nhiều bộ xử lý có thể cùng sử dụng một tài nguyên vật lý.
- Các bộ xử lý trao đổi dữ liệu bằng cách gửi nhận các thông điệp.
- Việc truyền dữ liệu thường yêu cầu thao tác điều phối thực hiện bởi mỗi bộ xử lý. Ví dụ, một thao tác gửi ở một luồng thì phải ứng với một thao tác nhận ở bộ xử lý khác.

2. Lịch sử ra đời của MPI:

Mô hình truyền thông điệp là một trong những mô hình lâu đời nhất và được ứng dụng rộng rãi nhất trong lập trình song song. Thời kì trước đây, có rất nhiều thư viện cho mô hình này mà giữa chúng chỉ khác biệt 1 vài tính năng. Vì vậy đến 1992, tác giả của những thư viện này đã tham dự hội nghị Supercomputing để

thống nhất 1 giao diện chuẩn thực hiện việc truyền thông điệp - the Message Passing Interface.

Năm 1994, một giao diện tiêu chuẩn hoàn chỉnh được định nghĩa (MPI-1). Và chỉ mất 1 năm nữa, bản cài đặt MPI đầu tiên cũng được đưa ra, từ đó MPI đã được áp dụng một cách rộng rãi, hiện nay 2 bản cài đặt được biết đến nhiều hơn cả là OPENMPI và MPICH. Trong bài báo cáo này em sử dụng bản MPICH2 cho ngôn ngữ C/C++.

3. Cài đặt cụm máy tính MPI trong mạng LAN

Ta có thể chạy chương trình MPI trên máy tính cá nhân hoặc kết nối các máy tính tạo thành 1 cụm (cluster).

Sau đây, em sẽ trình bày cách kết nối các máy tính (dùng hệ điều hành Ubuntu) trong mạng LAN tạo thành 1 cụm (cluster) để chạy chương trình song song với MPI:

Trước hết, tất cả các máy đều phải cài đặt MPICH2 thông qua lệnh:

```
$ sudo apt-get install mpich2
```

Ta giả sử sẽ có 1 máy tính đóng vai trò là master và các máy còn lại đóng vai trò là slave.

Các bước thực hiện như sau:

- Bước 1: Cấu hình hosts file

  Vì chúng ta cần kết nối các máy tính lại với nhau cho nên việc nhập địa chỉ IP thường xuyên sẽ rất bất tiện cho nên thay vào đó thì ta có thể đặt tên các nút tham gia vào cluster. Hosts file được sử dụng trong hệ điều hành để ánh xạ hostname và địa chỉ IP. Dùng lệnh sudo gedit /etc/hosts để sửa file /etc/hosts như sau:

192.168.1.10 master

192.168.1.11 slave1

cột bên trái là địa chỉ IP của các nút còn cột bên phải là hostname của các nút đó. Lưu ý, với nút master thì ta cần phải lưu ánh xạ của tất cả các nút slave khác và chính nó còn trong file /etc/hosts chỉ cần lưu trữ ánh xạ của nút master và chính nó mà thôi.

- Bước 2: Tạo user mới:

  Ta cần tạo 1 user truy cập chung cho tất cả các nút.

  $ sudo adduser mpiuser

- Bước 3: Cài đặt SSH:

  Các nút trong cluster sẽ giao tiếp qua SSH và chia sẻ dữ liệu qua NFS. Cài đặt SSH:

  $ sudo apt-get install openssh-server

  Sau đó, ta truy cập vào tài khoản vừa tạo:

  $ su – mpiuser

  Giờ ta có thể đăng nhập vào các máy khác dựa vào lệnh

  ssh hostname

  sau đó nhập mật khẩu của máy tính tương ứng với hostname đó. Để đăng nhập được dễ dàng hơn, chúng ta sẽ sinh khóa và copy chúng đến các nút khác :

  $ ssh-keygen -t rsa

  tiếp theo ta copy khóa vừa sinh đến các nút khác trong cluster:

  $ ssh-copy-id slave

  Thực hiện những bước trên với mỗi nút slave và master.

- Bước 4: Cài đặt NFS

Thông qua NFS master nút có thể chia sẻ 1 thư mục thông qua NFS, và các slave nút có thể mounts để trao đổi dữ liệu:

NFS-Server:

Cài đặt những package sau:

```
$ sudo apt-get install nfs-kernel-server
```

Tạo 1 thư mục có tên cloud mà chúng ta muốn chia sẻ trên mạng để export thhư mục cloud ta cần tạo cổng truy cập trong file /etc/exports bằng cách thêm dòng

/home/mpiuser/cloud* (rw,sync,no_root_squash,no_subtree_check)

Nếu cần thiết hãy khởi động lại NFS server

$ sudo service nfs-kernel-server restart

Với NFS-Client

Cài đặt package sau:

$ sudo apt-get install nfs-common

Và tạo thư mục cùng tên cloud ở các nút slave

$ mkdir cloud

và thêm vào file /etc/fstab:

master:/home/mpiuser/cloud /home/mpiuser/cloud nfs

rồi chạy lệnh

sudo mount –a

Quay lại với NFS-SERVER để phân quyền cho tài khoản mpiuser:

chown mpiuser /home/mpiuser/cloud

- Bước 5: Chạy chương trình MPI

Ta copy file chứa mã nguồn chương trình cần thực thi vào trong thư mục cloud vừa tạo ở bước trên và biên dịch chương trình này bằng lệnh:

$ mpicc -o name_of_exec_file name_of_source_file

Để chạy chương trình này chỉ trên máy của bạn:

$ mpirun -np 3 ./name_of_ exec_file

Để chạy chương trình với cụm:

$ mpirun -np 3 -hosts master,slave1,slave2 ./name_of_exec_file

II. MPI

1. Cấu trúc 1 chương trình MPI

Một chương trình song song MPI thường chứa nhiều hơn 1 tác vụ được thực hiện bởi các bộ xử lý (processor) khác nhau. Mỗi bộ xử lý được phân biệt với nhau bởi chỉ số rank, với rank là 1 số nguyên từ 0 đến N-1 với N là tổng số tiến trình tham gia vào chạy chương trình. Với các chương trình chạy theo cơ chế master/slave thì trong hệ thống thường có 1 bộ xử lý master đóng vai trò quản lý các slave khác, master thường có rank 0 còn slave thì có rank từ 1 đến N-1.

Cấu trúc chương trình song song MPI:

Cũng giống như các chương trình song song của các mô hình khác đều có phần code tuần tự và phần code song song, phần code song song của chương trình MPI bắt đầu hàm khởi tạo môi trường MPI cho đến hàm kết thúc môi trường MPI.
MPI là một mô hình sử dụng bộ nhớ phân tán, các bộ xử lý tham gia vào chương trình MPI đều biên dịch cùng 1 file mã nguồn tuy nhiên phụ thuộc vào rank của bộ xử lý nó sẽ thực thi chương trình theo cách khác nhau.

2. Các lệnh cơ bản trong MPI:

MPI gồm 129 lệnh. Tuy nhiên khi mới làm quen với MPI chỉ cần tới từ 6-24 lệnh. Tất cả chúng chỉ ra cách mà các tiến trình trao đổi dữ liệu hay chính là sự gửi và nhận thông điệp.

Những cơ bản dưới đây được sử dụng để xây dựng hầu hết các chương trình MPI:

- Trước hết, tất cả các chương trình MPI đều phải include đến thư viện mpi.h
- MPI_INIT: tất cả các chương trình MPI đều phải gọi lệnh này để khởi tạo môi trường.
- MPI_COMM_SIZE: để lấy số lượng bộ xử lý đang chạy.
- MPI_COMM_RANK: để xác định rank của bộ xử lý đang chạy chương trình.
- MPI_SEND và MPI_RECV: nhóm các lệnh để gửi và nhận thông điệp trong MPI.
- MPI_FINALIZE: lệnh báo hiệu đóng môi trường MPI, tuy nhiên các tác vụ chạy song song đang được thực thi vẫn được tiếp tục. Tất cả các lệnh MPI được gọi sau MPI_FINALIZE đều không có hiệu lực và báo lỗi.

- int MPI_Init(&argc, &argv) : khởi tạo môi trường MPI
- int MPI_Comm_size(MPI_Comm comm, int *size) :
  - Tham số comm: chỉ ra nhóm giao tiếp mà bộ xử lý gọi lệnh này tham gia vào.

    MPI_COMM_WORLD là hằng được định nghĩa trước chỉ nhóm giao tiếp toàn cục chứa tất cả các bộ xử lý.
  - size: lệnh này sẽ trả về số lượng bộ xử lý của nhóm giao tiếp vào biến này.

- int MPI_Comm_rank(MPI_Comm comm, int *rank):
    - comm: chỉ ra nhóm giao tiếp mà bộ xử lý gọi lệnh này tham gia vào.
    - rank: lệnh trả về giá trị rank của bộ xử lý trong nhóm giao tiếp.
- int MPI_Send, MPI_Recv la lệnh gửi và nhận dữ liệu trong MPI sẽ được trình bày cụ thể hơn sau phần sau.
- int MPI_Finalize(): lệnh đóng môi trường MPI.

3. Kiểu dữ liệu trong MPI.

Mỗi kiểu dữ liệu cơ bản trong C/C++ đều có tương ứng MPI, mỗi MPI_Datatype được dùng trong lệnh MPI trong chương trình của C/C++:

Kiểu dữ liệu MPI	Kiểu dữ liệu C/C++
MPI_CHAR	signed char
MPI_SHORT	signed short int
MPI_INT	signed int
MPI_LONG	signed long int
…	…
MPI_FLOAT	float
MPI_BYTE	
MPI_PACKED	

4. Communicator: nhóm giao tiếp

Là một nhóm các bộ xử lý hoặc toàn bộ các bộ xử lý (MPI_COMM_WORLD) mà trong đó việc thực hiện các lệnh MPI được xác định cho nhóm giao tiếp đó. Người mới làm quen với MPI để đơn giản có thể sử dụng MPI_COMM_WORLD là một nhóm giao tiếp toàn cục tác động đến tất cả các bộ xử lý trong chương trình.

Một bộ xử lý có thể tham gia vào nhiều nhóm giao tiếp khác nhau và tại mỗi nhóm giao tiếp nó được gán 1 rank khác nhau và có vai trò khác nhau trong thực hiện công việc.

5. Giao tiếp điểm điểm, giao tiếp tập thể trong MPI.

5.1. Giao tiếp point to point:

Đây là cơ chế giao tiếp giữa từng cặp bộ xử lý với nhau, trong đó 1 bộ xử lý thực hiện công việc gửi thông điệp và bộ xử lý còn lại sẽ nhận thông điệp tương ứng đó. Trong cơ chế này có nhiều kiểu giao tiếp với nhau, như:

- Blocking: các lệnh gửi/nhận dữ liệu sẽ kết thúc khi việc gửi/nhận dữ liệu hoàn tất.
- Non-blocking: các lệnh gửi/nhận dữ liệu sẽ kết thúc ngay mà không cần quan tâm đến việc dữ liệu có được gửi đi hay nhận về thành công chưa. Việc dữ liệu có thực sự được gửi đi hay nhận về sẽ được kiểm tra bằng cách lệnh khác trong MPI.
- Synchronization: gửi và nhận dữ liệu đồng bộ, quá trình gửi dữ liệu chỉ có thể được kết thúc khi quá trình nhận dữ liệu bắt đầu. Tại 1 thời điểm chỉ thiết lập 1 kênh cho quá trình gửi và nhận, các quá trình gửi và nhận khác phải chờ đợi cho đến khi kênh này được giải phóng mới được phép thực hiện.

Sau đây em sẽ trình bày về cơ chế blocking và non-blocking.

a. Blocking (bị chặn):

Để hiểu rõ về cơ chế blocking thì ta đi tìm hiểu về 2 hàm gửi và nhận dữ liệu trong cơ chế này là MPI_Send và MPI_Recv.

Cơ chế của việc gửi và nhận trong Blocking như sau:

Đầu tiên, bộ xử lý A quyết định gửi 1 thông điệp cho bộ xử lý B. bộ xử lý A đóng gói toàn bộ dữ liệu được gửi vào bộ đệm cho bộ xử lý B. Những bộ đệm này giống như là phong bì thư kể từ khi dữ liệu được đóng gói vào 1 thông điệp đơn trước khi chuyển (giống như cách mà lá thư được đóng gói vào bì thư trước khi đưa tới bưu điện).

Sau khi dữ liệu được đóng gói vào bộ đệm, thiết bị giao tiếp (thường là 1 mạng) chịu trách nhiệm điều hướng thông điệp tới địa chỉ thích hợp. Địa chỉ của thông điệp được định nghĩa là rank của bộ xử lý.

Mặc dù thông điệp được điều hướng tới B, bộ xử lý B vẫn phải xác nhận rằng nó muốn nhận dữ liệu từ A. Mỗi lần thực hiện điều này, dữ liệu mới được chuyển thành công. Bộ xử lý A được xác nhận rằng dữ liệu đã được chuyển và có thể trở lại thực hiện các lệnh khác, cần phải chú ý điều này vì trong nhiều trường hợp có thể xảy ra trường hợp 1 bộ xử lý nào đó không hoàn thành việc gửi, nhận dữ liệu khiến cho chương trình bị treo hoặc có thể gây ra hiện tượng các bộ xử lý chờ đợi vòng tròn hay tắc nghẽn (deadlock).

Trường hợp mà A cần gửi nhiều loại thông điệp đến B.

MPI sử dụng nhãn (tag) để phân biệt thông điệp.

B chỉ yêu cầu những thông điệp với nhãn xác định, các thông điệp với nhãn khác sẽ được đệm bởi mạng cho đến khi B sẵn sàng cho chúng. Chú ý nếu các thông điệp gửi đến với cùng nhãn giống nhau thì chương trình sẽ phân biệt theo thứ tự của thông điệp gửi đến.

Cú pháp của MPI_Send:

MPI_Send(void *data, int count, MPI_Datatype datatype, int destination, int tag, MPI_Comm communicator)

- data: địa chỉ vùng đệm dữ liệu cần gửi.
- count: số lượng phần tử muốn gửi đi.
- datatype: kiểu dữ liệu của mỗi phần tử gửi đi.
- destination: rank của tiến trình cần gửi đến.
- tag: nhãn của thông điệp.
- communication: nhóm giao tiếp.

Cú pháp MPI_Recv:

MPI_Recv(void *data, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm communicator, MPI_Status * status)

- data: địa chỉ vùng đệm nhận dữ liệu.
- count: số phần tử cần được nhận.
- datatype: kiểu dữ liệu của mỗi phần tử được nhận.
- source:

rank của bộ xử lý gửi thông điệp đến, hoặc ta có thể sử dụng MPI_ANY_SOURCE khi không rõ hoặc không quan tâm đến thông điệp đến từ bộ xử lý nào.

- tag: nhãn của thông điệp hoặc sử dụng MPI_ANY_TAG khi không quan tâm đến nhãn gửi đến.
- communicator: nhóm giao tiếp.
- status: trạng thái của thông điệp lưu trữ thông tin về rank của bộ xử lý gửi thông điệp đến (status.MPI_SOURCE), nhãn của thông điệp (status. MPI_TAG) và độ dài thông điệp gửi đến.

Ta cũng có thể sử dụng MPI_STATUS_IGNORE để bỏ qua không cần quan tâm đến trạng thái của quá trình trao đổi dữ liệu.

Thực tế số lượng phần tử ở vùng đệm có thể nhỏ hơn số lượng trong khai báo hàm MPI_Recv. Để kiểm tra xem số lượng bên gửi gửi là bao nhiêu để tranh khai báo dư thừa ta chạy những lệnh này trước khi gọi tới MPI_Recv .

MPI_Probe(int source, int tag, MPI_Comm comm, MPI_Status *status);

các thông số của thông điệp sẽ được điền vào biến status

MPI_Get_count(&status, MPI_INT, &number);

lệnh này sẽ trả về số lượng phần tử được gửi đến vào biến number.

Nhìn qua thì thấy điều này có vẻ không cần thiết khi mà ta hoàn toàn có thể gửi thêm 1 thông điệp nữa về số lượng cho bên nhận, nhưng trong thực tế thì chi phí cho việc gửi thông điệp gây tốn kém hơn. Nguyên tắc chung trong cơ chế giao tiếp của MPI:

- Cố gắng nhóm càng nhiều dữ liệu trong 1 lần giao tiếp càng tốt. Gửi M bytes mỗi lần trong N lần giao tiếp sẽ tốn kém hơn gửi 1 lần với N*M bytes.
- Cố gắng gửi chính xác số lượng dữ liệu lưu trữ trong bộ đệm tránh gửi nhiều hơn gây lãng phí.

b. Non-blocking (không bị chặn):

Cơ chế non-blocking gắn với 2 hàm gửi và nhận thông điệp: MPI_Isend và MPI_Irecv.

Lệnh để gửi dữ liệu trong cơ chế non-blocking là MPI_Isend được định nghĩa như sau:

int MPI_Isend(void * data, int count, MPI_Datatype datatype, int dest, int tag, MPI_Communicator comm, MPI_Request * request);

- Các tham số khác đều giống như trong hàm MPI_Send ngoài tham số ứng với kiểu dữ liệu MPI_Request. Hiểu 1 cách đơn giản là khi gọi tới hàm MPI_Isend thì bộ xử lý sẽ chuẩn bị 1 request.
  Request này sẽ được thực thi khi cả 2 bộ xử lý gửi và nhận cùng sẵn sàn đồng bộ để gửi và nhận dữ liệu. Lệnh này thực tế không gửi đi bất kỳ thứ gì đến bộ xử lý nhận nó, nó chỉ chuẩn bị cho việc gửi đi.
- Mỗi lần request được chuẩn bị, nó cần thiết phải được hoàn thành. Có 2 cách để hoàn thành 1 request: là "wait" (đợi) và "test" (kiểm tra).

Và tương tự, lệnh nhận trong cơ chế non-blocking là MPI_Irecv:

```
int Irecv(void *data, int count, MPI_Datatype datatype, int source, int tag,
    MPI_Communicator comm, MPI_Request *request);
```

- Giống như MPI_Isend, MPI_Irecv cũng trả về 1 đối tượng MPI_Request và request này cần được hoàn thành cũng bởi waiting hoặc testing.

Waiting:

Waiting buộc bộ xử lý chuyển sang chế độ blocking. Bộ xử lí gửi dữ liệu đơn giản sẽ phải chờ cho đến khi request kết thúc. Nếu cho bộ xử lí của bạn wait ngay sau khi gọi lệnh MPI_Isend thì lúc này việc gửi đi sẽ giống hoàn toàn với việc gọi hàm MPI_Send.

```
int MPI_Wait(MPI_Request *request, MPI_Status *status);
```

MPI_Wait chỉ đợi sự hoàn thành của request được chỉ định trước. Ngay sau khi request được hoàn thành thì trạng thái của quá trình trao đổi thông điệp sẽ được trả vào biến status.

Ta cũng có thể sử dụng MPI_STATUS_IGNORE nếu không quan tâm đến trạng thái của quá trình trao đổi thông điệp này.

Testing:

Testing có một chút sự khác biệt. Như trình bày ở trên wait sẽ chặn, khóa bộ xử lý cho đến khi request được hoàn thành, còn testing sẽ kiểm tra request có thể được hoàn thành hay không. Nếu có thể, request này sẽ tự động hoàn thành và dữ liệu sẽ được gửi đi thành công.

int MPI_Test(MPI_Request *request, int *flag, MPI_Status* status);

Các tham số về request và status trong MPI_Test cũng giống như trong MPI_Wait, ta chỉ cần nhớ rằng trong bất kỳ trường hợp nào thì bộ xử lý vẫn tiếp tục thực hiện các lệnh tiếp theo sau MPI_Test. Biến flag sẽ chỉ ra request chỉ định có được hoàn thành trong khi thực hiện test hay không. Nếu flag !=0 nghĩa là request được hoàn thành.

Ta có thể kết hợp cả cơ chế giao tiếp blocking và non-blocking một bộ xử lý gửi với MPI_Isend, 1 bộ xử lý khác nhận với MPI_Recv. Sử dụng hàm MPI_Iprobe để thăm dò trước xem có thông điệp nào gửi cho bộ xử lý gọi lệnh này hay không và ghi lại thông tin của thông điệp đó.

MPI_Iprobe(int source, int tag, MPI_Comm comm, int *flag, MPI_Status * status)

- source: rank của tiến trình gửi thông điệp đến,
  ta cũng có thể sử dụng MPI_ANY_SOURCE để chỉ định thăm dò bất kỳ thông điệp đến từ bộ xử lý nào.
- tag: nhãn của thông điệp, có thể sử dụng MPI_ANY_TAG.
- flag: trả về giá trị khác 0 nếu có thông điệp với source, tag, communicator chỉ định gửi đến.

Ví dụ: Giả sử bộ xử lý root (rank =0) đang thực hiện 1 số công việc nào đó và tại nó sẽ ngừng lại khi nhận được tín hiệu từ bộ xử lý rank =1:

```
int stop = 0;
if (rank == 1) {
        /*Thực hiện một số công việc*/
        for (int i = 0; i< 1000; i++){}
        stop = 1;
        MPI_Isend(&stop, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &request);
}
if (rank == 0) {
        int so_vong_lap = 0;
        int flag = 0;
        while (flag != 0){
                /* Thực hiện công việc nào đó tại rank = 0 */
                for (int i =0; i<10; i++){}
                so_vong_lap++;
                MPI_Iprobe(1, 0, MPI_COMM_WORLD, &flag, &status);
        }
        MPI_Recv(&stop, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &status);
        cout<<"Rank 0 thuc hien duoc "<<so_vong_lap<<endl;
}
MPI_Wait(&request, &status);
```

5.2.Giao tiếp tập thể (Collective Communications)

Trong phần trước em đã trình bày về cách gửi và nhận dữ liệu từ 1 bộ xử lý đến một bộ xử lý chỉ định khác.

Tuy nhiên có những bài toán mà yêu cầu sự giao tiếp đến tất cả các bộ xử lý khác trong nhóm giao tiếp, lúc này ta cần sử dụng các lệnh trong giao tiếp tập thể để chương trình trở nên đơn giản và hiệu quả hơn.

Các kiểu giao tiếp trong cơ chế này:

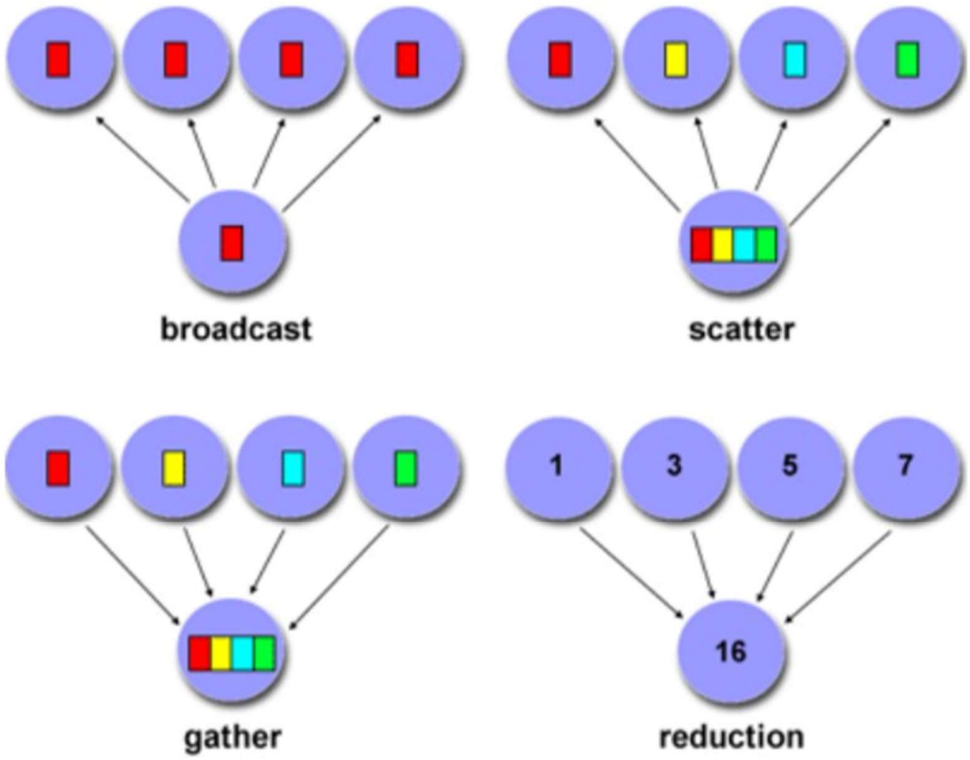
- Broadcast:

  1 bộ xử lý gửi thông điệp giống nhau đến tất cả các bộ xử lý khác.
- Reduction:

  1 bộ xử lý nhận dữ liệu và thực hiện thu gọn dữ liệu bằng các hoạt động: cộng, tìm max, tìm min,... của các dữ liệu nhận về.
- Scatter:

  1 bộ xử lý phân chia dữ liệu thành các phần rồi gửi từng phần đó đến cho các bộ xử lý.
- Gather:

  1 bộ xử lý lắp ghép các phần dữ liệu nhận được từ các bộ xử lý khác.



a, Broadcasting:

Khi một bộ xử lý thực hiện Broadcast, nó sẽ gửi cùng 1 dữ liệu đến tất cả các bộ xử lý khác trong nhóm giao tiếp. Một trong những công dụng chính của broadcast là để gửi input của người dùng trong chương trình song song, hoặc gửi tham số cấu hình đến tất cả các bộ xử lý khác.

Trong MPI, Broadcasting có thể được thực hiện bằng lệnh MPI_Bcast:

```
MPI_Bcast(void *data, int count, MPI_Datatype datatype, int root,
          MPI_Comm comm)
```

Với root là rank của bộ xử lí gửi dữ liệu đi.

Mặc dù bộ xử lý root và bộ xử lí nhận làm các công việc khác nhau nhưng tất cả chúng đều gọi đến hàm MPI_Bcast, biến dữ liệu được gửi đến tất cả các bộ xử lí.

b, MPI_Scatter:

Cũng giống như MPI_Bcast tuy nhiên MPI_Bcast gửi cùng dữ liệu đến các bộ xử lý khác còn MPI_Scatter thì gửi từng phần của mảng chỉ định đến các bộ xử lý khác:

Trong hình minh họa trên thì MPI_Bcast lấy 1 phần tử tại bộ xử lí root và gửi tới tất cả các bộ xử lý khác. MPI_Scatter thì lấy 1 dãy các phần tử và phân tán phần tử này theo thứ tự rank. MPI_Scatter sẽ gửi phần tử thích hợp đến bộ đệm nhận của bộ xử lý nhận:

MPI_Scatter (void *data, int send_count, MPI_Datatype datatype, void * recv_data, int recv_count, MPI_Datatype recv_datatype, int root MPI_Comm comm)

- send_data: là mảng các dữ liệu cần gửi từ bộ xử lý root.
- send_count: là số phần tử gửi cho mỗi bộ xử lí
- recv_data: là mảng nhận dữ liệu của bộ xử lý nhận.
- recv_count: số phần tử là mỗi bộ xử lí cần được nhận về.

c, MPI_Gather:

Ngược với MPI_Scatter, MPI_Gather sẽ lấy các dữ liệu từ bộ xử lý khác và tập trung lại ở bộ xử lý root. Kiểu giao tiếp này sẽ hữu ích cho các thuật toán sắp xếp, tìm kiếm:

MPI_Gather (void *send_data,
int send_count, MPI_Datatype send_datatype,
void *recv_data, int recv_count, MPI_Datatype recv_datatype,

int root, MPI_Comm comm)

- Trong MPI_Gather bộ xử lý root chỉ cần ghi rõ bộ đệm nhận hợp lí còn các bộ xử lý khác có thể đặt NULL cho recv_data.

  Tham số recv_count là số phần tử nhận được ở mỗi bộ xử chứ không phải là số phần tử tổng cộng nhận từ các bộ xử lí.

d, Reduction:

Giống với MPI_Gather, MPI_Reduce lấy 1 mảng các phần tử đầu vào ở mỗi bộ xử lý và trả về 1 mảng các phần tử đầu ra tới bộ xử lý root. Phần tử đầu ra chưa kết quả đã được "giảm tải". Lệnh MPI_Reduce có nguyên mẫu như sau:

MPI_reduce(void * send_data, void * recv_data, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)

- send_data: là mảng các phần tử kiểu datatype mà mỗi bộ xử lý muốn " giảm tải".
- recv_data: là mảng chứa kết quả đã được giảm tải.
- op: tham số tùy chọn chỉ hoạt động "giảm tải" mà ta muốn áp dụng với dữ liệu. Một số tùy chọn như sau:
  - MPI_MAX: trả về phần tử lớn nhất.
  - MPI_MIN: trả về phần tử nhỉ nhất.
  - MPI_SUM: trả về tổng của các phần tử.
  - MPI_PROD: trả về tích các phần tử.
  - MPI_LAND: thực hiện phép logic AND giữa các phần tử .
  - MPI_LOR: thực hiện phép toàn logic OR giữa các phần tử.

Ví dụ:

Trong hình minh họa trên: tại mỗi bộ xử lý đều có 1 mảng gồm 2 phần tử, tại mảng kết quả ở rank 0 cũng sẽ gồm 2 phần tử và mỗi phần tử sẽ có giá trị bằng tổng các phần tử ở các mảng thành phần cùng vị trí với nó.

III. Một số thuật toán trong giao tiếp tập thể:

1. Thuật toán trong Allgather:

- Thuật toán Ring:

Dữ liệu ở mỗi bộ xử lý sẽ được trao đổi theo vòng. Tại bước thứ nhất thì bộ xử lý i sẽ gửi dữ liệu của nó đến cho bộ xử lý i+1 và nhận dữ liệu từ bộ xử lý i-1. Tại bước tiếp theo các bộ xử lý sẽ trao đổi dữ liệu vừa nhận được ở bước trước với các bộ xử lý lân cận.

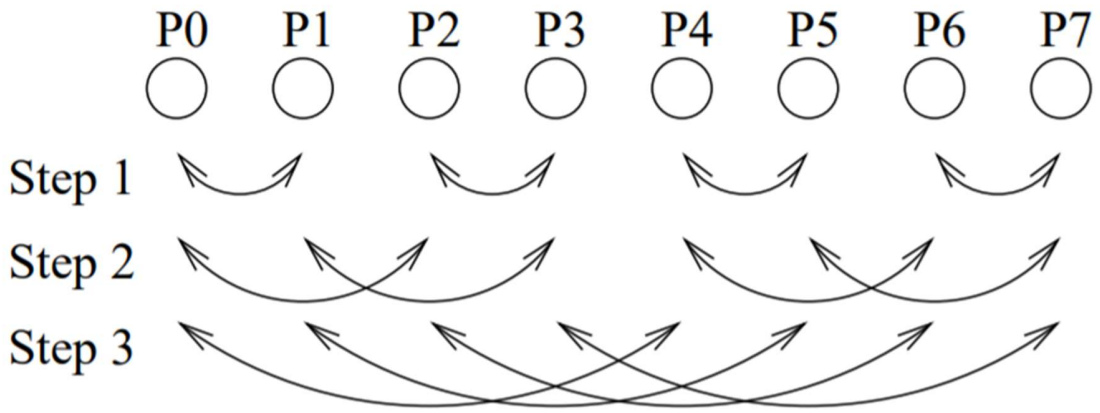
Nếu có p bộ xử lý thì thuật toán này sẽ tiến hành p bước với n (tính theo byte) là tổng số dữ liệu thu thập ở mỗi bộ xử lý khi đó mỗi bước sẽ trao đổi n/p lượng dữ liệu. Thời gian của thuật toán là

$T = (p-1)* \alpha + (p-1)*n*\beta/p$

Với α thời gian trễ truyền tin, β là thời gian trao đổi trên mỗi byte dữ liệu.

Ta có thể giảm thiểu còn lg(p) bước với thuật toán recursive doubling (đệ quy nhân đôi) dưới đây.

- Thuật toán Recursive doubling :



Giả sử có 8 bộ xử lý. Tại bước thứ nhất thì các bộ xử lý với khoảng cách 1 sẽ trao đổi dữ liệu với nhau, tại bước thứ 2 thì các bộ xử lý với khoảng cách 2 sẽ trao đổi dữ liệu với nhau, tại bước thứ 3 thì các bộ xử lý với khoảng cách 4 sẽ trao đổi dữ liệu với nhau. Tại bước đầu tiên các bộ xử lý trao đổi n/p lượng dữ liệu, ở bước 2 là 2n/p lượng dữ liệu,… tại bước cuối là $2^{lg(p)-1}$ *n/p lượng dữ liệu

Vì vậy thời gian của thuật toán là

$T = lg(p)* \alpha + (p-1)*n* \beta/p$

Demo thuật toán :

```
for (int i = 0; i < log(size) / log(2); i++)
{
    int begin;
    // *receive = *value;
    int v = pow(2, i);
    if ((rank % (v << 1)) < v && rank + v < size)
    {
        MPI_Send(value, v, MPI_INT, rank + v, 1, MPI_COMM_WORLD);
        int *receive = new int[v];
        MPI_Recv(receive, v, MPI_INT, rank + v, 1, MPI_COMM_WORLD, &status);
        add(value, size, receive, v);
        free(receive);
    }
    else if ((rank % (v << 1)) >= v)
    {
        MPI_Send(value, v, MPI_INT, rank - v, 1, MPI_COMM_WORLD);
        int *receive = new int[v];
        MPI_Recv(receive, v, MPI_INT, rank - v, 1, MPI_COMM_WORLD, &status);
        add(value, size, receive, v);
        free(receive);
    }
}
```

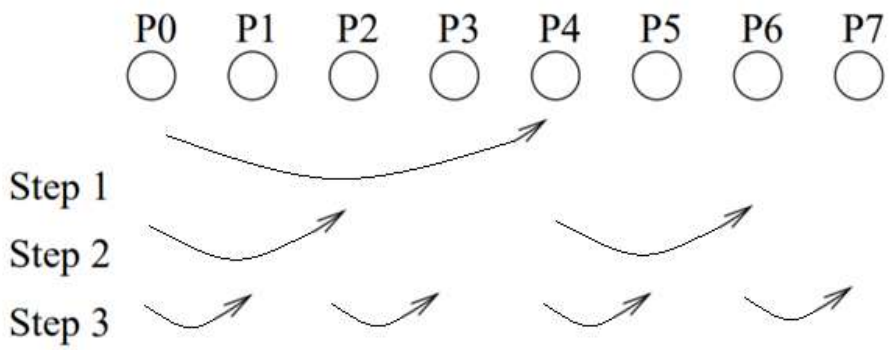
Thuật toán chạy với bộ xử lý là lũy thừa của 2.

Tại bước lặp i thì bộ xử lý có $rank\%2^{i-1} < 2^i$ sẽ tiến hành trao đổi dữ liệu với $rank + 2^i$. Còn $rank\%2^{i-1} > 2^i$ sẽ trao đổi dữ liệu với rank – v . Hàm add() sẽ thêm lượng dữ liệu trao đổi vào mảng lưu trữ.

❖ Nhận xét :

Thuật toán Recursive doubling có số bước lặp nhỏ hơn thuật toán Ring tuy nhiên thuật toán Recursive doubling có sự trao đổi dữ liệu giữa các bộ xử lý có khoảng cách xa (Thuật toán Ring chỉ trao đổi dữ liệu với các bộ xử lý lân cận) và lượng dữ liệu trao đổi mỗi lần tăng lên, do đó thời gian của thuật toán Recursive doubling phụ thuộc vào kích thước dữ liệu trao đổi và cấu hình topology của Cluster.

2.Thuật toán trong Scatter:



Recursive halving algorithm:

Mảng dữ liệu gửi từ rank 0 đến các rank còn lại. Tại bước thứ nhất bộ xử lý gốc (rank =0) gửi đến bộ xử lý cách nó 4 khoảng cách, bước thứ 2, bộ xử lý 0 và 4 gửi dữ liệu đến bộ xử lý cách nó 2 khoảng cách, bước thứ 3 bộ xử lý 0, 2, 4, 6 gửi dữ liệu đến bộ xử lý cách nó 1 khoảng cách.

Thời gian của thuật toán:

$T = \lg(p)* \alpha + (p-1)*n* \beta/p$

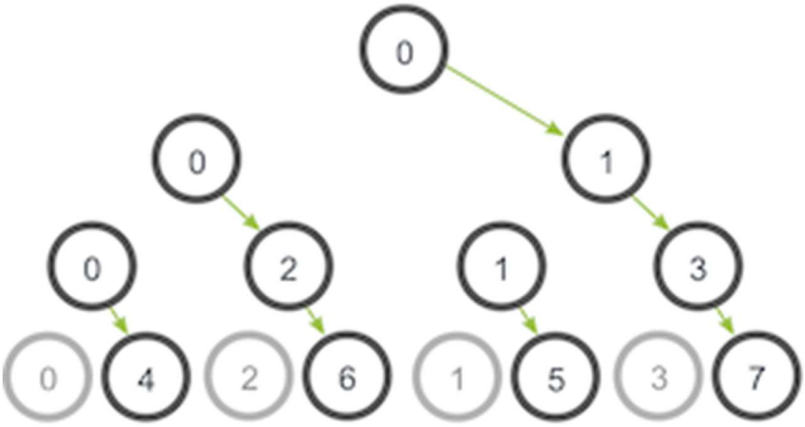
Demo thuật toán:

```
for (int i = 0; i < log(size) / log(2); i++)
{
    int v = pow(2, i);
    if (rank % (size/v) == 0)
    {
        MPI_Send(value + size/(2*v), size/(2*v), MPI_INT, (rank + size/(2*v)), 0, MPI_COMM_WORLD);
    }
    else if (rank % (size/v) == size/(2*v))
    {
        MPI_Recv(value, size/(2*v), MPI_INT, (rank - size/(2*v)), 0, MPI_COMM_WORLD, &status);
    }
}
```

Tại bước lặp thứ i bộ xử lý có rank%(p/2^i) = 0 sẽ gửi dữ liệu đến rank + 2^i.

3.Thuật toán trong Broadcast:

Ta sử dụng thuật toán Binary tree:



Tương tự thuật toán cũng chỉ cần lg(p) bước lặp. Thời gian của thuật toán là:

T = lg(p)* α + (p-1)*n* β

Demo thuật toán:

```
for (int i = 0; i < log(size) / log(2); i++)
{
  int v = pow(2, i);
  if ((rank < v) && (rank + v < size))
  {
    MPI_Send(&value, 1, MPI_INT, (v + rank), 0, MPI_COMM_WORLD);
  }
  else if (rank < (v << 1))
  {
    MPI_Recv(&value, 1, MPI_INT, (rank - v), 0, MPI_COMM_WORLD, &status);
  }
}
```

Tại vòng lặp thứ i các nút có rank nhỏ hơn 2^i sẽ thực hiện chuyển dữ liệu đến cho nút có rank + 2^i. Còn 2^i < rank < 2^{i+1} sẽ tiến hành nhận dữ liệu.

Ngoài ra với gói tin lớn ta có thể Scatter sau đó Gather dữ liệu lại.

Thời gian tính toán: (Gather sử dụng thuật toán Ring)

$T = (\lg(p) + p-1)* \alpha + 2*(p-1)*n* \beta/p$

IV.Tài liệu tham khảo :

- https://mpitutorial.com/tutorials/
- https://www.mcs.anl.gov/~thakur/papers/mpi-coll.pdf