Cơ bản lập trình song song MPI cho C/C++

Đặng Nguyên Phương
dnphuong1984@gmail.com

Ngày 23 tháng 11 năm 2013

Mục lục

1 Mở đầu

Thông thường hiện nay, hầu hết các chương trình tính toán đều được thiết kế để chạy trên một lõi (*single core*), đó là cách tính toán tuần tự (*serial computation*). Để có thể chạy được chương trình một cách hiệu quả trên các hệ thống máy tính (*cluster*) hoặc các cpu đa lõi (*multi-core*), chúng ta cần phải tiến hành song song hóa chương trình đó. Ưu điểm của việc tính toán song song (*parallel computation*) chính là khả năng xử lý nhiều tác vụ cùng một lúc. Việc lập trình song song có thể được thực hiện thông qua việc sử dụng các hàm thư viện (vd: *mpi.h*) hoặc các đặc tính đã được tích hợp trong các chương trình biên dịch song song dữ liệu, chẳng hạn như OpenMP trong các trình biên dịch fortran F90, F95.

Công việc lập trình song song bao gồm việc thiết kế, lập trình các chương trình máy tính song song sao cho nó chạy được trên các hệ thống máy tính song song. Hay có nghĩa là song song hoá các chương trình tuần tự nhằm giải quyết một vấn đề lớn hoặc làm giảm thời gian thực thi hoặc cả hai. Lập trình song song tập trung vào việc phân chia bài toán tổng thể ra thành các công việc con nhỏ hơn rồi định vị các công việc đó đến từng bộ xử lý (*processor*) và đồng bộ các công việc để nhận được kết quả cuối cùng. Nguyên tắc quan trọng nhất ở đây chính là tính đồng thời hoặc xử lý nhiều tác vụ (*task*) hay tiến trình (*process*) cùng một lúc. Do đó, trước khi lập trình song song ta cần phải biết được rằng bài toán có thể được song song hoá hay không (có thể dựa trên dữ liệu hay chức năng của bài toán). Có hai hướng chính trong việc tiếp cận lập trình song song:

- Song song hoá ngầm định (*implicit parallelism*): bộ biên dịch hay một vài chương trình khác tự động phân chia công việc đến các bộ xử lý.
- Song song hoá bằng tay (*explicit parallelism*): người lập trình phải tự phân chia chương trình của mình đế nó có thể được thực thi song song.

Ngoài ra trong lập trình song song, người lập trình cũng cần phải tính đến yếu tố cân bằng tải (*load balancing*) trong hệ thống. Phải làm cho các bộ xử lý thực hiện số công việc như nhau, nếu có một bộ xử lý có tải quá lớn thì cần phải di chuyển công việc đến bộ xử lý có tải nhỏ hơn.

Một mô hình lập trình song song là một tập hợp các kỹ thuật phần mềm để thể hiện các giải thuật song song và đưa vào ứng dụng trong hệ thống song song. Mô hình này bao gồm các ứng dụng, ngôn ngữ, bộ biên dịch, thư viện, hệ thống truyền thông và vào/ra song song. Trong thực tế, chưa có một máy tính song song nào cũng như cách phân chia công việc cho các bộ xử lý nào có thể áp dụng hiệu quả cho mọi bài toán. Do đó, người lập trình phải lựa chọn chính xác một mô hình song song hoặc pha trộn giữa các mô hình với nhau để phát triển các ứng dụng song song trên một hệ thống cụ thể.

Hiện nay có rất nhiều mô hình lập trình song song như mô hình đa luồng (*multi-threads*), 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 (*process interaction*) và cách thức xử lý bài toán (*problem decomposition*). Theo tiêu chí thứ nhất, chúng ta có 2 loại mô hình song song chủ yếu là mô hình dùng bộ nhớ chia sẻ (*shared memory*) hoặc truyền thông điệp (*message passing*). Theo tiêu chí thứ hai, chúng ta cũng có hai 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 một vùng nhớ dùng chung.
- Với mô hình truyền thông điệp thì mỗi xử lý đều có riêng một 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 khac 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 cho các hệ thống phân tán (*distributed system*). Các đặc trưng của mô hình này là:

- Các luồng (*thread*) sử dụng vùng nhớ cục bộ riêng của chúng trong suốt quá trình tính toán.
- Nhiều luồng có thể cùng sử dụng một tài nguyên vật lý.
- Các luồng 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 luồng. Ví dụ, một thao tác gửi ở một luồng thì phải ứng với một thao tác nhận ở luồng khác.

Tài liệu này được xây dựng với mục đích cung cấp các kiến thức cơ bản bước đầu nhằm tìm hiểu khả năng viết một chương trình song song bằng ngôn ngữ lập trình C/C++ theo cơ chế trao đổi thông điệp sử dụng các thư viện theo chuẩn MPI. Mục đích là nhắm tới việc thực thi các chương trình C/C++ trên máy tính đa lõi hoặc hệ thống cụm máy tính (*computer cluster*) giúp nâng cao hiệu năng tính toán. Trong tài liệu này, thư viện MPICH2 được sử dụng để biên dịch các chương trình C/C++ trên hệ điều hành Linux.

2 MPI

2.1 Giới thiệu

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. Hai bộ công cụ phổ biến nhất cho lập trình song song theo mô hình này là PVM (*Parallel Virtual Machine*) và MPI (*Message Passing Interface*). Các bộ công cụ này cung cấp các hàm dùng cho việc trao đổi thông tin giữa các tiến trình tính toán trong hệ thống máy tính song song.

MPI (*Message Passing Interface*) là một chuẩn mô tả các đặc điểm và cú pháp của một thư viện lập trình song song, được đưa ra vào năm 1994 bởi MPIF (*Message Passing Interface Forum*), và được nâng cấp lên chuẩn MPI-2 từ năm 2001. Có rất nhiều các thư viện dựa trên chuẩn MPI này chẳng hạn như MPICH, OpenMPI hay LAM/MPI.

MPICH2 là một thư viện miễn phí bao gồm các hàm theo chuẩn MPI dùng cho lập trình song song theo phương thức truyền thông điệp, được thiết kế cho nhiều ngôn ngữ lập trình khác nhau (C++, Fortran, Python,...) và có thể sử dụng trên nhiều loại hệ điều hành (Windows, Linux, MacOS,...).

2.2 Cài đặt MPICH2

Gói MPICH2 có thể được cài đặt trên tất cả các máy tính thông qua lệnh sau

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

Sau khi đã cài đặt thành công MPICH2, ta cần phải cấu hình trước khi chạy song song. Trong trường hợp phiên bản được sử dụng là 1.2.x trở về trước thì trình quản lý thực thi mặc định sẽ là MPD, còn từ 1.3.x trở về sau thì trình quản lý sẽ là Hydra. Cách thức cấu hình dành cho 2 trình quản lý sẽ như sau:

MPD Tạo 2 file *mpd.hosts* và *.mpd.conf* trong thư mục chủ (vd: `/home/phuong`). Trong đó, file *mpd.hosts* sẽ chứa tên của các máy con trong hệ thống, ví dụ như

```
master
node1
node2
node3
```

Còn đối với file *.mpd.conf*, ta cần phải thiết lập quyền truy cập cho file này thông qua lệnh

```
$ chmod 600 .mpd.conf
```

Sau đó mở file và thêm dòng sau vào trong file

```
secretword=random_text_here
```

Để khởi động MPD, gõ lệnh sau trên máy chủ

```
$ mpdboot -n <N>
```

với N là số máy có trong hệ thống.

Hydra tương tự như với MPD nhưng đơn giản hơn, ta chỉ cần tạo duy nhất 1 file có tên *hosts* tại thư mục `/home/phuong` chứa tên của tất cả các máy con trong hệ thống

```
master
node1
node2
node3
```

2.3 Biên dịch và thực thi chương trình với MPICH2

Biên dịch Để biên dịch một chương trình ứng dụng với MPICH2, ta có thể sử dụng một trong các trình biên dịch sau

Ngôn ngữ	Trình biên dịch
C	`mpicc`
C++	`mpicxx`, `mpic++`, `mpiCC`
Fortran	`mpif77`, `mpif90`, `mpifort`

Ví dụ như ta muốn biên dịch một chương trình ứng dụng viết bằng ngôn ngữ C/C++, ta có thể gõ lệnh sau

```
mpicc -o helloworld helloworld.c
```

Trong đó, `helloworld.c` là file chứa mã nguồn của chương trình, tùy chỉnh `-o` cho ta xác định trước tên của file ứng dụng được biên dịch ra, trong trường hợp này là file `helloworld`.

Thực thi Trong trường hợp phiên bản MPICH2 sử dụng trình quản lý MPD, trước khi thực thi chương trình ta cần gọi MPD qua lệnh `mpdboot` như đã đề cập đến ở trên hoặc

```
mpd &
```

Chương trình đã được biên dịch bằng MPI có thể được thực thi bằng cách sử dụng lệnh

```
mpirun -np <N> <tenchuongtrinh>
```

hoặc

```
mpiexec -n <N> <tenchuongtrinh>
```

Trong đó `N` là số tác vụ song song cần chạy và `tenchuongtrinh` là tên của chương trình ứng dụng cần thực thi.

Ví dụ:

```
$ mpd &
$ mpirun -np 8 helloworld
```

3 Cấu trúc cơ bản của một chương trình MPI

3.1 Cấu trúc chương trình

Cấu trúc cơ bản của một chương trình MPI như sau:

Khai báo các header, biến, prototype,...

Bắt đầu chương trình

. . .

<đoạn code tuần tự>

. . .

Khởi động môi trường MPI

. . .

<đoạn code cần thực hiện song song>

. . .

Kết thúc môi trường MPI

. . .

<đoạn code tuần tự>

. . .

Kết thúc chương trình

3.2 Các khái niệm cơ bản

Một chương trình song song MPI thường chứa nhiều hơn một tác vụ (*task*) hay còn gọi là tiến trình (*process*) thực thi. Mỗi tác vụ (tiến trình) được phân biệt với nhau bởi chỉ số tác vụ (được gọi là *rank* hay *task ID*). Chỉ số này là một số nguyên từ 0 đến (N−1) với N là tổng số tác vụ MPI tham gia chạy chương trình. Đối với các chương trình chạy theo cơ chế *master/slave* thì trong hệ thống thường có một tác vụ chủ (*master*) điều khiển các tác vụ khác được gọi là tác vụ con (*slave*), tác vụ chủ này thường có chỉ số là 0 còn các tác vụ con có chỉ số từ 1 đến (N−1).

Tập hợp của các tác vụ MPI cùng chạy một chương trình được gọi là một nhóm (*group*). Và tập hợp của các tác vụ trong cùng một nhóm mà có thể trao đổi thông tin với nhau được gọi là một *communicator*. Khi bắt đầu chương trình, *communicator* mà bao gồm tất cả các tác vụ thực thi được mặc định là `MPI_COMM_WORLD`.

Các tác vụ trong MPI trao đổi với nhau thông qua việc gửi/nhận các thông điệp (*message*). Mỗi thông điệp đều chứa hai thành phần bao gồm dữ liệu (*data*) và *header*, mỗi *header* bao gồm:

- Chỉ số của tác vụ gửi
- Chỉ số của tác vụ nhận
- Nhãn (*tag*) của thông điệp
- Chỉ số của communicator

3.3 Ví dụ "Hello world"

Để bước đầu làm quen với việc viết một chương trình MPI, ta sẽ bắt đầu với một ví dụ đơn giản, đó là viết chương trình "Hello world" bằng ngôn ngữ C. Các bước thực hiện như sau:

- Bước đầu tiên, ta sẽ tạo một file có tên *hello.c* và mở file này bằng các chương trình soạn thảo văn bản dạng text (vd: *gedit*, *emacs*, *vim*,...)
- Khai báo tên chương trình và thêm header MPI

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

int main (int argc, char** argv)
{

}
```

Cần lưu ý rằng header MPI (`mpi.h`) cần phải được thêm vào trong file để có thể gọi được các lệnh MPI.

- Khai báo môi trường MPI

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

int main (int argc, char** argv)
{
  MPI_Init (&argc, &argv);
}
```

Lệnh `MPI_Init` khởi tạo môi trường MPI để thực hiện tác vụ song song, lệnh này sẽ trả về một giá trị nguyên trong quá trình khởi tạo môi trường.

- Gọi lệnh quản lý số tác vụ song song

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

int main (int argc, char** argv)
{
  int ntasks;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
}
```

Lệnh `MPI_Comm_size` trả về giá trị số lượng tác vụ song song vào trong biến `ntasks`. Tham số `MPI_COMM_WORLD` là tham số chỉ communicator toàn cục, có giá trị là một hằng số nguyên.

- Gọi lệnh xác định chỉ số của tác vụ

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

int main (int argc, char** argv)
{
  int ntasks, mytask;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
```

```
    MPI_Comm_rank (MPI_COMM_WORLD, &mytask);
  }
```

Lệnh `MPI_Comm_rank` trả về chỉ số (*rank*) của tác vụ vào trong biến `mytask`, chỉ số này có giá trị từ 0 đến `ntasks-1`, và được sử dụng để nhận biết tác vụ khi điều khiển gửi/nhận thông tin.

- Thực hiện lệnh xuất ra màn hình

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

int main (int argc, char** argv)
{
  int ntasks, mytask;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
  MPI_Comm_rank (MPI_COMM_WORLD, &mytask);
  printf( "Hello world from task %d of %d \n", mytask, ntasks);
}
```

- Kết thúc môi trường MPI

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

int main (int argc, char** argv)
{
  int ntasks, mytask;

  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &ntasks);
  MPI_Comm_rank (MPI_COMM_WORLD, &mytask);
  printf( "Hello world from task %d of %d \n", mytask, ntasks);
  MPI_Finalize();
  return 0;
}
```

Lệnh `MPI_Finalize` 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 bị báo lỗi.

Ngoài ra, ta cũng có thể viết lại chương trình "Hello world" này theo ngôn ngữ C++, ta sẽ tạo một file mới *hello.cc* có nội dung như sau

```
#include <iostream>
#include <mpi.h>

int main (int argc, char** argv)
{
  int ntasks, mytask;

  MPI::Init(argc, argv);
  ntasks = MPI::COMM_WORLD.Get_size();
  mytask = MPI::COMM_WORLD.Get_rank();
  std::cout << "Hello world from task " << mytask << " of " << ntasks <<
     std::endl;
  MPI::Finalize();
  return 0;
```

```
}
```

Lưu ý rằng cách thức sử dụng lệnh MPI trong C và C++ khác nhau ở hai điểm chính sau đây:

- Các hàm trong C++ được sử dụng với không gian tên (*namespace*) MPI.
- Các tham số (*argument*) được sử dụng trong các hàm C++ là tham chiếu (*reference*) thay vì là con trỏ (*pointer*) như trong các hàm C. Ví dụ như các tham số `argc` và `argv` của hàm `MPI_Init` trong C được sử dụng với dấu `&` phía trước, còn trong C++ thì không.

3.4 Ví dụ truyền thông điệp

Trong ví dụ "Hello world" ta đã làm quen với 4 lệnh cơ bản của MPI. Trong thực tế, rất nhiều chương trình song song MPI có thể được xây dựng chỉ với 6 lệnh cơ bản, ngoài 4 lệnh vừa kể trên ta còn sử dụng thêm hai lệnh nữa là `MPI_Send` để gửi thông điệp và `MPI_Recv` để nhận thông điệp giữa các tác vụ với nhau. Cấu trúc của hai lệnh này trong C như sau:

```
MPI_Send (&buffer,count,datatype,destination,tag,communicator)

MPI_Recv (&buffer,count,datatype,source,tag,communicator,&status)
```

Trong đó

`buffer`	mảng dữ liệu cần chuyển/nhận
`count`	số phần từ trong mảng
`datatype`	kiểu dữ liệu (vd: `MPI_INT`, `MPI_FLOAT`,...)
`destination`	chỉ số của tác vụ đích (bên trong communicator)
`source`	chỉ số của tác vụ nguồn (bên trong communicator)
`tag`	nhãn của thông điệp (dạng số nguyên)
`communicator`	tập hợp các tác vụ
`status`	trạng thái của thông điệp
`ierror`	mã số lỗi

Để có thể hiểu rõ hơn về cách sử dụng hai lệnh này, ta sẽ xem ví dụ vòng lặp (*fixed_ loop*) sau. Trong ví dụ này, `MPI_Send` được sử dụng để gửi đi số vòng lặp đã hoàn thành từ mỗi tác vụ con (có chỉ số từ 1 đến N−1) đến tác vụ chủ (chỉ số là 0). Lệnh `MPI_Recv` được gọi N−1 lần ở tác vụ chủ để nhận N−1 thông tin được gửi từ N−1 tác vụ con.

Các bước khai báo đầu tiên cũng tương tự như trong ví dụ "Hello world"

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

int main(int argc, char **argv)
{
   int i, rank, ntasks, count, start, stop, nloops, total_nloops;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

}
```

Giả sử chúng ta muốn thực hiện vòng lặp 1000 lần, do đó số lần lặp của mỗi tác vụ sẽ bằng `1000/ntasks` với `ntasks` là tổng số tác vụ. Chúng ta sẽ sử dụng chỉ số của mỗi tác vụ để đánh dấu phân khúc lặp của mỗi tác vụ

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

int main(int argc, char **argv)
{
   int i, rank, ntasks, count, start, stop, nloops, total_nloops;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

   count = 1000 / ntasks;
   start = rank * count;
   stop = start + count;

   nloops = 0;
   for (i=start; i<stop; ++i) {
      ++nloops;
   }

   printf("Task %d performed %d iterations of the loop.\n",
        rank, nloops);

}
```

Trong đó `count` là số lần lặp của mỗi tác vụ, các tác vụ có chỉ số `rank` sẽ thực hiện lần lặp thứ `rank*count` đến lần lặp thứ `rank*count+count-1`. Biến `nloops` sẽ đếm số lần thực hiện vòng lặp của tác vụ có chỉ số `rank` và xuất ra màn hình.
Trong trường hợp tác vụ đang thực hiện không phải là tác vụ chủ (`rank` khác 0), tác vụ này sẽ gửi kết quả `nloops` cho tác vụ chủ.

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

int main(int argc, char **argv)
{
   int i, rank, ntasks, count, start, stop, nloops, total_nloops;

   MPI_Init(&argc, &argv);

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

   count = 1000 / ntasks;
   start = rank * count;
   stop = start + count;

   nloops = 0;
   for (i=start; i<stop; ++i) {
      ++nloops;
   }

   printf("Task %d performed %d iterations of the loop.\n",
        rank, nloops);

   if (rank != 0) {
      MPI_Send( &nloops, 1, MPI_INT, 0, 0, MPI_COMM_WORLD );
```

```
    }

}
```

Trong trường hợp tác vụ này là tác vụ chủ, nó sẽ nhận giá trị `nloops` từ các tác vụ con gửi và cộng dồn lại

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

int main(int argc, char **argv)
{
    int i, rank, ntasks, count, start, stop, nloops, total_nloops;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

    count = 1000 / ntasks;
    start = rank * count;
    stop = start + count;

    nloops = 0;
    for (i=start; i<stop; ++i) {
       ++nloops;
    }

    printf("Task %d performed %d iterations of the loop.\n",
         rank, nloops);

    if (rank != 0) {
       MPI_Send( &nloops, 1, MPI_INT, 0, 0, MPI_COMM_WORLD );
    } else {
       total_nloops = nloops;
       for (i=1; i<ntasks; ++i) {
          MPI_Recv( &nloops, 1, MPI_INT, i, 0, MPI_COMM_WORLD, 0 );
          total_nloops += nloops;
       }
    }

}
```

Tác vụ chủ sẽ chạy tiếp để đủ số 1000 vòng lặp trong trường hợp có dư ra một số vòng lặp sau khi chia 1000 cho tổng số tác vụ.

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

int main(int argc, char **argv)
{
    int i, rank, ntasks, count, start, stop, nloops, total_nloops;

    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

    count = 1000 / ntasks;
    start = rank * count;
```

```
      stop = start + count;

      nloops = 0;
      for (i=start; i<stop; ++i) {
         ++nloops;
      }

      printf("Task %d performed %d iterations of the loop.\n",
           rank, nloops);

      if (rank != 0) {
         MPI_Send( &nloops, 1, MPI_INT, 0, 0, MPI_COMM_WORLD );
      } else {
         total_nloops = nloops;
         for (i=1; i<ntasks; ++i) {
            MPI_Recv( &nloops, 1, MPI_INT, i, 0, MPI_COMM_WORLD, 0 );
            total_nloops += nloops;
         }
         nloops = 0;
         for (i=total_nloops; i<1000; ++i) {
            ++nloops;
         }
      }

   }
```

Cuối cùng là xuất ra kết quả và kết thúc môi trường MPI, cũng như kết thúc chương trình.

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

   int main(int argc, char **argv)
   {
      int i, rank, ntasks, count, start, stop, nloops, total_nloops;

      MPI_Init(&argc, &argv);

      MPI_Comm_rank(MPI_COMM_WORLD, &rank);
      MPI_Comm_size(MPI_COMM_WORLD, &ntasks);

      count = 1000 / ntasks;
      start = rank * count;
      stop = start + count;

      nloops = 0;
      for (i=start; i<stop; ++i) {
         ++nloops;
      }

      printf("Task %d performed %d iterations of the loop.\n",
           rank, nloops);

      if (rank != 0) {
         MPI_Send( &nloops, 1, MPI_INT, 0, 0, MPI_COMM_WORLD );
      } else {
         total_nloops = nloops;
         for (i=1; i<ntasks; ++i) {
            MPI_Recv( &nloops, 1, MPI_INT, i, 0, MPI_COMM_WORLD, 0 );
            total_nloops += nloops;
```

```
        }
        nloops = 0;
        for (i=total_nloops; i<1000; ++i) {
           ++nloops;
        }

        printf("Task 0 performed the remaining %d iterations of the loop
              \n", nloops);
    }

    MPI_Finalize();
    return 0;
}
```

Chương trình vòng lặp cũng có thể được viết lại theo ngôn ngữ C++ như sau

```
#include <iostream>
#include <mpi.h>

int main(int argc, char **argv)
{
    MPI::Init(argc, argv);

    int rank = MPI::COMM_WORLD.Get_rank();
    int ntasks = MPI::COMM_WORLD.Get_size();

    int count = 1000 / ntasks;
    int start = rank * count;
    int stop = start + count;
    int nloops = 0;

    for (int i=start; i<stop; ++i) {
       ++nloops;
    }

    std::cout << "Task " << rank << " performed " << nloops
            << " iterations of the loop.\n";

    if (rank != 0) {
       MPI::COMM_WORLD.Send( &nloops, 1, MPI_INT, 0, 0 );
    } else {
       int total_nloops = nloops;
       for (int i=1; i<ntasks; ++i) {
          MPI::COMM_WORLD.Recv( &nloops, 1, MPI_INT, i, 0 );
          total_nloops += nloops;
       }
       nloops = 0;
       for (int i=total_nloops; i<1000; ++i) {
          ++nloops;
       }

       std::cout << "Task 0 performed the remaining " << nloops
                 << " iterations of the loop\n";
    }
    MPI::Finalize();
    return 0;
}
```

4 Các lệnh MPI

4.1 Các lệnh quản lý môi trường MPI

Các lệnh này có nhiệm vụ thiết lập môi trường cho các lệnh thực thi MPI, truy vấn chỉ số của tác vụ, các thư viện MPI,...

MPI_Init khởi động môi trường MPI

```
MPI_Init (&argc,&argv)
Init (argc,argv)
```

MPI_Comm_size trả về tổng số tác vụ MPI đang được thực hiện trong communicator (chẳng hạn như trong `MPI_COMM_WORLD`)

```
MPI_Comm_size (comm,&size)
Comm::Get_size()
```

MPI_Comm_rank trả về chỉ số của tác vụ (`rank`). Ban đầu mỗi tác vụ sẽ được gán cho một số nguyên từ 0 đến (N−1) với N là tổng số tác vụ trong communicator `MPI_COMM_WORLD`.

```
MPI_Comm_rank (comm,&rank)
Comm::Get_rank()
```

MPI_Abort kết thúc tất cả các tiến trình MPI

```
MPI_Abort (comm,errorcode)
Comm::Abort(errorcode)
```

MPI_Get_processor_name trả về tên của bộ xử lý

```
MPI_Get_processor_name (&name,&resultlength)
Get_processor_name(&name,resultlen)
```

MPI_Initialized trả về giá trị 1 nếu `MPI_Init()` đã được gọi, 0 trong trường hợp ngược lại

```
MPI_Initialized (&flag)
Initialized (&flag)
```

MPI_Wtime trả về thời gian chạy (tính theo giây) của bộ xử lý

```
MPI_Wtime ()
Wtime ()
```

MPI_Wtick trả về độ phân giải thời gian (tính theo giây) của `MPI_Wtime()`

```
MPI_Wtick ()
Wtick ()
```

MPI_Finalize kết thúc môi trường MPI

```
MPI_Finalize ()
Finalize ()
```

Ví dụ:

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

int main(int argc, char **argv)
{
   int  numtasks, rank, len, rc;
   char hostname[MPI_MAX_PROCESSOR_NAME];

   rc = MPI_Init(&argc,&argv);
   if (rc != MPI_SUCCESS) {
      printf ("Error starting MPI program. Terminating.\n");
      MPI_Abort(MPI_COMM_WORLD, rc);
   }

   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);
   MPI_Get_processor_name(hostname, &len);
   printf ("Number of tasks= %d My rank= %d Running on %s\n", numtasks,
      rank,hostname);

   /*******  do some work *******/

   MPI_Finalize();
}
```

4.2 Các kiểu dữ liệu

Một số kiểu dữ liệu cơ bản của MPI được liệt kê trong bảng sau

Tên	Kiểu dữ liệu
MPI_CHAR	signed character
MPI_WCHAR	wide character
MPI_SHORT	signed short
MPI_INT	signed int
MPI_LONG	signed long
MPI_LONG_LONG	signed long long
MPI_UNSIGNED_CHAR	unsigned character
MPI_UNSIGNED_SHORT	unsigned short
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double

Tên	Kiểu dữ liệu
MPI_C_COMPLEX	float_Complex
MPI_C_DOUBLE_COMPLEX	double_Complex
MPI_C_BOOL	bool
MPI_INT8_T	int8_t
MPI_INT16_T	int16_t
MPI_INT32_T	int32_t
MPI_INT64_T	int64_t
MPI_UINT8_T	uint8_t
MPI_UINT16_T	uint16_t
MPI_UINT32_T	uint32_t
MPI_UINT64_T	uint64_t
MPI_BYTE	byte
MPI_PACKED	data packed

Ngoài ra người dùng còn có thể tự tạo ra các cấu trúc dữ liệu riêng cho mình dựa trên các kiểu dữ liệu cơ bản này. Các kiểu dữ liệu có cấu trúc do người dùng tự định nghĩa được gọi là *derived data types*. Các lệnh định nghĩa cấu trúc dữ liệu mới bao gồm:

MPI_Type_contiguous tạo ra kiểu dữ liệu mới bằng cách lặp `count` lần kiểu dữ liệu cũ.

```
MPI_Type_contiguous (count,oldtype,&newtype)
Datatype::Create_contiguous(count)
```

MPI_Type_vector tương tự như *contigous* nhưng có các phân đoạn (`stride`) cố định, kiểu dữ liệu mới được hình thành bằng cách lặp một dãy các khối (*block*) của kiểu dữ liệu cũ có kích thước bằng nhau tại các vị trí có tính tuần hoàn.

```
MPI_Type_vector (count,blocklength,stride,oldtype,&newtype)
Datatype::Create_vector(count,blocklength,stride)
```

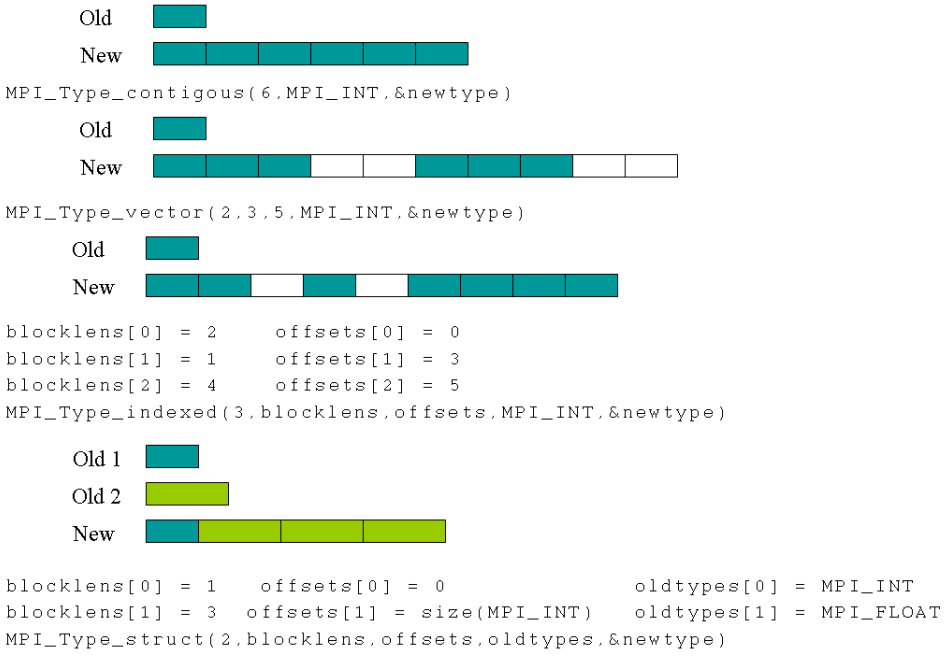
MPI_Type_indexed kiểu dữ liệu mới được hình thành bằng cách tạo một dãy các khối của kiểu dữ liệu cũ, mỗi khối có thể chứa số lượng các bản sao kiểu dữ liệu cũ khác nhau.

```
MPI_Type_indexed (count,blocklens[],offsets[],oldtype,&newtype)
Datatype::Create_hindexed(count,blocklens[],offsets[])
```

MPI_Type_struct tương tự như trên nhưng mỗi khối có thể được tạo thành bởi các kiểu dữ liệu cũ khác nhau.

```
MPI_Type_struct (count,blocklens[],offsets[],oldtypes,&newtype)
Datatype::Create_struct(count, blocklens[],offsets[],oldtypes[])
```

Hình 1 trình bày một số ví dụ cho các cách tạo cấu trúc dữ liệu mới. Trong trường hợp truyền các cấu trúc dữ liệu không cùng kiểu, ta có thể sử dụng các lệnh `MPI_Packed` và `MPI_Unpacked` để đóng gói dữ liệu trước khi gửi.



Hình 1: Ví dụ các cách tạo cấu trúc dữ liệu mới

MPI_Type_extent trả về kích thước (tính theo byte) của kiểu dữ liệu

```
MPI_Type_extent (datatype,&extent)
Datatype::Get_extent(lb,extent)
```

MPI_Type_commit đưa kiểu dữ liệu mới định nghĩa vào trong hệ thống

```
MPI_Type_commit (&datatype)
Datatype::Commit()
```

MPI_Type_free bỏ kiểu dữ liệu

```
MPI_Type_free (&datatype)
Datatype::Free()
```

Ví dụ: *tạo kiểu dữ liệu vector*

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

int main(int argc, char **argv)
{
    int numtasks, rank, source=0, dest, tag=1, i;
    float a[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 b[SIZE];

    MPI_Status stat;
    MPI_Datatype columntype;

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

    MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
    MPI_Type_commit(&columntype);

    if (numtasks == SIZE) {
       if (rank == 0) {
          for (i=0; i<numtasks; i++)
             MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
       }
       MPI_Recv(b, SIZE, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
       printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f\n",
              rank,b[0],b[1],b[2],b[3]);
    } else
       printf("Must specify %d processors. Terminating.\n",SIZE);

    MPI_Type_free(&columntype);
    MPI_Finalize();
}
```

4.3 Các cơ chế truyền thông điệp

Các cơ chế giao tiếp trong MPI gồm có:

Point-to-point cơ chế giao tiếp điểm-điểm, đây là cơ chế giao tiếp giữa từng cặp tác vụ với nhau, trong đó 1 tác vụ thực hiện công việc gửi thông điệp và tác vụ còn lại có nhiệm vụ nhận thông điệp tương ứng đó. Thông điệp được phân biệt bởi chỉ số của tác vụ và nhãn (*tag*) của thông điệp. Trong cơ chế này có nhiều kiểu giao tiếp với nhau, chẳng hạn như

- *Blocking*: các lệnh gửi/nhận dữ liệu sẽ kết thúc khi việc gửi/nhận dữ liệu hoàn tất.

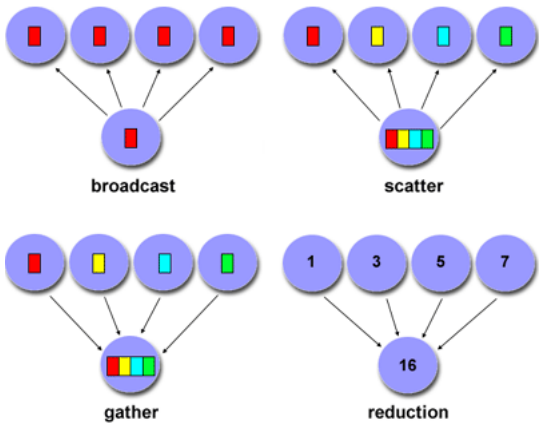
- *Non-blocking*: các lệnh gửi/nhận dữ liệu sẽ kết thúc ngay mà quan tâm đến việc dữ liệu đã thực sự được hoàn toàn gửi đi hoặc nhận về hay chưa. Việc dữ liệu đã thực sự được gửi đi hay nhận về sẽ được kiểm tra các lệnh khác trong thư viện MPI.
- *Synchronous*: gửi dữ liệu đồng bộ, quá trình gửi dữ liệu chỉ có thể được kết thúc khi quá trình nhận dữ liệu được bắt đầu.
- *Buffer*: một vùng nhớ đệm sẽ được tạo ra để chứa dữ liệu trước khi được gửi đi, người dùng có thể ghi đè lên vùng bộ nhớ chứa dữ liệu mà không sợ làm mất dữ liệu chuẩn bị gửi.
- *Ready*: quá trình gửi dữ liệu chỉ có thể được bắt đầu khi quá trình nhận dữ liệu đã sẵn sàng.

Bảng dưới đây tổng hợp các chế độ giao tiếp điểm-điểm và các lệnh gửi/nhận thông điệp tương ứng, thông tin chi tiết về các lệnh này sẽ được trình bày ở những phần sau

Chế độ	Điều kiện kết thúc	Blocking	Non-blocking
Send	Thông điệp đã được gửi	`MPI_Send`	`MPI_Isend`
Receive	Thông điệp đã được nhận	`MPI_Recv`	`MPI_Irecv`
Synchronous send	Khi quá trình nhận bắt đầu	`MPI_Ssend`	`MPI_Issend`
Buffer send	Luôn kết thúc, không quan tâm quá trình nhận đã bắt đầu hay chưa	`MPI_Bsend`	`MPI_Ibsend`
Ready send	Luôn kết thúc, không quan tâm quá trình nhận đã kết thúc hay chưa	`MPI_Rsend`	`MPI_Irsend`

Collective communication cơ chế giao tiếp tập thể, liên quan tới tất cả các tác vụ nằm trong phạm vi của communicator, các kiểu giao tiếp trong cơ chế này (xem Hình 2) gồm có

- *Broadcast*: dữ liệu giống nhau được gửi từ tác vụ gốc (*root*) đến tất cả các tác vụ khác trong communicator.
- *Scatter*: các dữ liệu khác nhau được gửi từ tác vụ gốc đến tất cả các tác vụ khác trong communicator.
- *Gather*: các dữ liệu khác nhau được thu thập bởi tác vụ gốc từ tất cả các tác vụ khác trong communicator.
- *Reduce*: phương thức này cho phép ta có thể thu thập dữ liệu từ mỗi tác vụ, rút gọn dữ liệu, lưu trữ dữ liệu vào trong một tác vụ gốc hoặc trong tất cả các tác vụ.



Hình 2: Minh họa các kiểu giao tiếp theo cơ chế tập thể

4.4 Các lệnh truyền thông điệp blocking

Một số lệnh thông dụng cho chế độ truyền thông điệp blocking gồm có:

MPI_Send gửi các thông tin cơ bản

```
MPI_Send (&buf,count,datatype,dest,tag,comm)
Comm::Send(&buf,count,datatype,dest,tag)
```

MPI_Recv nhận các thông tin cơ bản

```
MPI_Recv (&buf,count,datatype,source,tag,comm,&status)
Comm::Recv(&buf,count,datatype,source,tag,status)
```

MPI_Ssend gửi đồng bộ thông tin, lệnh này sẽ chờ cho đến khi thông tin đã được nhận (thông tin được gửi sẽ bị giữ lại cho đến khi bộ đệm của tác vụ gửi được giải phóng để có thể sử dụng lại và tác vụ đích (*destination process*) bắt đầu nhận thông tin)

```
MPI_Ssend (&buf,count,datatype,dest,tag,comm)
Comm::Ssend(&buf,count,datatype,dest,tag)
```

MPI_Bsend tạo một bộ nhớ đệm (*buffer*) mà dữ liệu được lưu vào cho đến khi được gửi đi, lệnh này sẽ kết thúc khi hoàn tất việc lưu dữ liệu vào bộ nhớ đệm.

```
MPI_Bsend (&buf,count,datatype,dest,tag,comm)
Comm::Bsend(&buf,count,datatype,dest,tag)
```

MPI_Buffer_attach cấp phát dung lượng bộ nhớ đệm cho thông tin được sử dụng bởi lệnh `MPI_Bsend()`

```
MPI_Buffer_attach (&buffer,size)
Attach_buffer(&buffer,size)
```

MPI_Buffer_detach bỏ cấp phát dung lượng bộ nhớ đệm cho thông tin được sử dụng bởi lệnh `MPI_Bsend()`

```
MPI_Buffer_detach (&buffer,size)
Detach_buffer(&buffer,size)
```

MPI_Rsend gửi thông tin theo chế độ *ready*, chỉ nên sử dụng khi người lập trình chắc chắn rằng quá trình nhận thông tin đã sẵn sàng.

```
MPI_Rsend (&buf,count,datatype,dest,tag,comm)
Comm::Rsend(&buf,count,datatype,dest,tag)
```

MPI_Sendrecv gửi thông tin đi và sẵn sàng cho việc nhận thông tin từ tác vụ khác

```
MPI_Sendrecv (&sendbuf,sendcount,sendtype,dest,sendtag,
              &recvbuf,recvcount,recvtype,source,recvtag,comm,&status)
Comm::Sendrecv(&sendbuf,sendcount,sendtype,dest,sendtag,
              &recvbuf,recvcount,recvtype,source,recvtag,status)
```

MPI_Wait chờ cho đến khi các tác vụ gửi và nhận thông tin đã hoàn thành

```
MPI_Wait (&request,&status)
Request::Wait(status)
```

MPI_Probe kiểm tra tính blocking của thông tin

```
MPI_Probe (source,tag,comm,&status)
Comm::Probe(source,tag,status)
```

Ví dụ:

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

int main(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();
}
```

4.5 Các lệnh truyền thông điệp non-blocking

Một số lệnh thông dụng cho chế độ truyền thông điệp non-blocking gồm có:

MPI_Isend gửi thông điệp non-blocking, xác định một khu vực của bộ nhớ thực hiện nhiệm vụ như là một bộ đệm gửi thông tin.

```
MPI_Isend (&buf,count,datatype,dest,tag,comm,&request)
Request Comm::Isend(&buf,count,datatype,dest,tag)
```

MPI_Irecv nhận thông điệp non-blocking, xác định một khu vực của bộ nhớ thực hiện nhiệm vụ như là một bộ đệm nhận thông tin.

```
MPI_Irecv (&buf,count,datatype,source,tag,comm,&request)
Request Comm::Irecv(&buf,count,datatype,source,tag)
```

MPI_Issend gửi thông điệp non-blocking đồng bộ (*synchronous*).

```
MPI_Issend (&buf,count,datatype,dest,tag,comm,&request)
Request Comm::Issend(&buf,count,datatype,dest,tag)
```

MPI_Ibsend gửi thông điệp non-blocking theo cơ chế *buffer*.

```
MPI_Ibsend (&buf,count,datatype,dest,tag,comm,&request)
Request Comm::Ibsend(&buf,count,datatype,dest,tag)
```

MPI_Irsend gửi thông điệp non-blocking theo cơ chế *ready*.

```
MPI_Irsend (&buf,count,datatype,dest,tag,comm,&request)
Request Comm::Irsend(&buf,count,datatype,dest,tag)
```

MPI_Test kiểm tra trạng thái kết thúc của các lệnh gửi và nhận thông điệp non-blocking `Isend()`, `Irecv()`. Tham số `request` là tên biến yêu cầu đã được dùng trong các lệnh gửi và nhận thông điệp, tham số `flag` sẽ trả về giá trị 1 nếu thao tác hoàn thành và giá trị 0 trong trường hợp ngược lại.

```
MPI_Test (&request,&flag,&status)
Request::Test(status)
```

MPI_Iprobe kiểm tra tính non-blocking của thông điệp

```
MPI_Iprobe (source,tag,comm,&flag,&status)
Comm::Iprobe(source,tag,status)
```

Ví dụ:

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

int main(int argc, char **argv)
{
    int numtasks, rank, next, prev, buf[2], tag1=1, tag2=2;
    MPI_Request reqs[4];
    MPI_Status stats[2];

    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]);

    {  /*do some work*/  }

    MPI_Waitall(4, reqs, stats);

    MPI_Finalize();
```

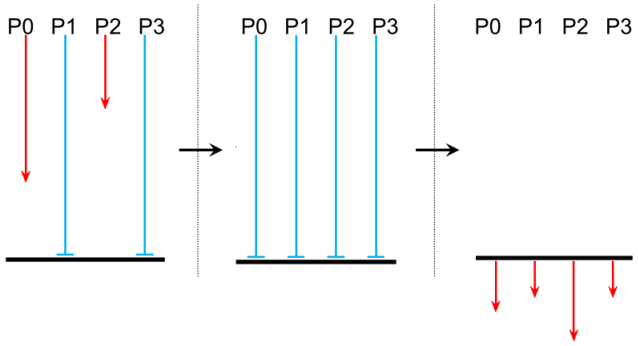
```
}
```

4.6 Các lệnh truyền thông tập thể

Một số lệnh thông dụng cho cho cơ chế truyền thông tập thể gồm có:

MPI_Barrier lệnh đồng bộ hóa (rào chắn), tác vụ tại rào chắn (*barrier*) phải chờ cho đến khi tất cả các tác vụ khác trên cùng một communicator đều hoàn thành (xem Hình 3).

```
MPI_Barrier (comm)
Intracomm::Barrier()
```



Hình 3: Minh họa lệnh rào chắn

MPI_Bcast gửi bản sao của bộ đệm có kích thước `count` từ tác vụ `root` đến tất cả các tiến trình khác trong cùng một communicator.

```
MPI_Bcast (&buffer,count,datatype,root,comm)
Intracomm::Bcast(&buffer,count,datatype,root)
```

MPI_Scatter phân phát giá trị bộ đệm lên tất cả các tác vụ khác, bộ đệm được chia thành `sendcnt` phần.

```
MPI_Scatter (&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)
Intracomm::Scatter(&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root)
```

MPI_Gather tạo mới một giá trị bộ đệm riêng cho mình từ các mảnh dữ liệu gộp lại.

```
MPI_Gather (&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root,comm)
Intracomm::Gather(&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype,root)
```

MPI_Allgather tương tự như `MPI_GATHER` nhưng sao chép bộ đệm mới cho tất cả các tác vụ.

```
MPI_Allgather (&sendbuf,sendcnt,sendtype,&recvbuf,recvcount,recvtype,comm)
Intracomm::Allgather(&sendbuf,sendcnt,sendtype,&recvbuf,recvcnt,recvtype)
```

MPI_Reduce áp dụng các toán tử rút gọn (tham số `op`) cho tất cả các tác vụ và lưu kết quả vào một tác vụ duy nhất.

```
MPI_Reduce (&sendbuf,&recvbuf,count,datatype,op,root,comm)
Intracomm::Reduce(&sendbuf,&recvbuf,count,datatype,op,root)
```

Các toán tử rút gọn gồm có: `MPI_MAX` (cực đại), `MPI_MIN` (cực tiểu), `MPI_SUM` (tổng), `MPI_PROD` (tích), `MPI_LAND` (toán tử AND logic), `MPI_BAND` (toán tử AND bitwise), `MPI_LOR` (toán tử OR

logic), `MPI_BOR` (toán tử OR bitwise), `MPI_LXOR` (toán tử XOR logic), `MPI_BXOR` (toán tử XOR bitwise), `MPI_MAXLOC` (giá trị cực đại và vị trí), `MPI_MINLOC` (giá trị cực tiểu và vị trí).

MPI_Allreduce tương tự như `MPI_Reduce` nhưng lưu kết quả vào tất cả các tác vụ.

```
MPI_Allreduce (&sendbuf,&recvbuf,count,datatype,op,comm)
Intracomm::Allreduce(&sendbuf,&recvbuf,count,datatype,op)
```

MPI_Reduce_scatter tương đương với việc áp dụng lệnh `MPI_Reduce` rồi tới lệnh `MPI_Scatter`.

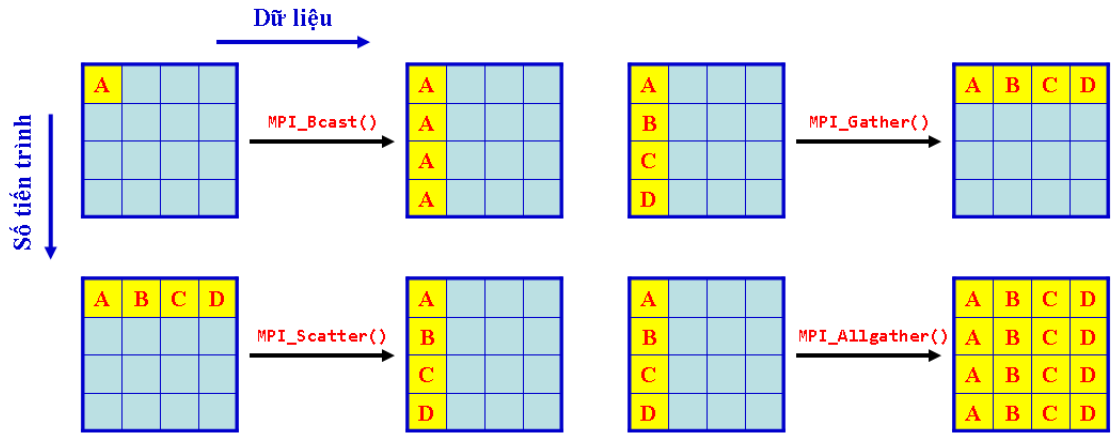
```
MPI_Reduce_scatter (&sendbuf,&recvbuf,recvcount,datatype,op,comm)
Intracomm::Reduce_scatter(&sendbuf,&recvbuf,recvcount[], datatype,op)
```

MPI_Alltoall tương đương với việc áp dụng lệnh `MPI_Scatter` rồi tới lệnh `MPI_Gather`.

```
MPI_Alltoall (&sendbuf,sendcount,sendtype,&recvbuf,recvcnt,recvtype,comm)
Intracomm::Alltoall(&sendbuf,sendcount,sendtype,&recvbuf,recvcnt,recvtype)
```

MPI_Scan kiểm tra việc thực hiện toán tử rút gọn của các tác vụ.

```
MPI_Scan (&sendbuf,&recvbuf,count,datatype,op,comm)
Intracomm::Scan(&sendbuf,&recvbuf,count,datatype,op)
```



Hình 4: Minh họa một số lệnh giao tiếp tập thể

Ví dụ:

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

int main(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();
}
```

5 Một số ví dụ

5.1 Ví dụ tính số π

Trong ví dụ này ta sẽ tiến hành lập trình song song cho phép tính số π. Giá trị của số π có thể được xác định qua công thức tích phân

$$\pi = \int_0^1 f(x)dx \text{ , với } f(x) = \frac{4}{(1+x^2)} \tag{1}$$

Tích phân này có thể được xấp xỉ theo giải tích số như sau

$$\pi = \frac{1}{n}\sum_{i=1}^{n} f(x_i) \text{ , với } x_i = \frac{(i-\frac{1}{2})}{n} \tag{2}$$

Công thức xấp xỉ trên có thể dễ dàng xây dựng với C

```
h   = 1.0 / (double) n;
sum = 0.0;
for (i = 1; i <= n; i++) {
   x = h * ((double)i - 0.5);
   sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;
```

Trong ví dụ này, ta sẽ sử dụng cơ chế truyền thông tập thể. Dựa vào công thức phía trên ta có thể dễ dàng nhận ra rằng chỉ có một tham số duy nhất, đó là `n`, do đó ta sẽ truyền tham số này cho tất cả các tác vụ trong hệ thống thông qua lệnh `MPI_Bcast`

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Mỗi tác vụ con sẽ thực thi công thức xấp xỉ `n/numtasks` lần, với `numtasks` là tổng số các tác vụ, do đó vòng lặp sẽ được chỉnh sửa lại

```
h   = 1.0 / (double) n;
sum = 0.0;
for (i = myid + 1; i <= n; i += numtasks) {
   x = h * ((double)i - 0.5);
   sum += (4.0 / (1.0 + x*x));
}
mypi = h * sum;
```

với `myid` là chỉ số của tác vụ đang thực thi.

Kết quả chạy từ các tác vụ sẽ được tính tổng và lưu ở tác vụ chủ thông qua lệnh `MPI_Reduce`

```
MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

Đoạn code chương trình sau khi thêm phần MPI

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

int main(int argc, char **argv)
{
    int n, myid, numprocs, i;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    h   = 1.0 / (double) n;
    sum = 0.0;
    for (i = myid + 1; i <= n; i += numprocs) {
        x = h * ((double)i - 0.5);
        sum += (4.0 / (1.0 + x*x));
    }
    mypi = h * sum;
    MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);

    MPI_Finalize();
    return 0;
}
```

Chương trình tính giá trị số π hoàn chỉnh như sau

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

int main(int argc, char **argv)
{
    int n, myid, numprocs, i;
    double PI25DT = 3.14159265358979323846264338;
    double mypi, pi, h, sum, x;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    while (1) {
        if (myid == 0) {
            printf("Enter the number of intervals: (0 quits) ");
            scanf("%d",&n);
        }
        MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
        if (n == 0)
```

```
            break;
        else {
            h   = 1.0 / (double) n;
            sum = 0.0;
            for (i = myid + 1; i <= n; i += numprocs) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;
            MPI_Reduce(&mypi, &pi, 1, MPI_DOUBLE, MPI_SUM, 0,
                MPI_COMM_WORLD);
            if (myid == 0)
                printf("pi is approximately %.16f, Error is %.16f\n", pi,
                    fabs(pi - PI25DT));
        }
    }
    MPI_Finalize();
    return 0;
}
```

Chương trình tính số π cũng có thể được viết lại theo ngôn ngữ C++ như sau

```
#include <math.h>
#include <mpi.h>

int main(int argc, char **argv)
{
    int n, rank, size, i;
    double PI25DT = 3.14159265358979323846264 3;
    double mypi, pi, h, sum, x;

    MPI::Init(argc, argv);
    size = MPI::COMM_WORLD.Get_size();
    rank = MPI::COMM_WORLD.Get_rank();

    while (1) {
        if (rank == 0) {
        cout << "Enter the number of intervals: (0 quits)" << endl;
        cin >> n;
        }

        MPI::COMM_WORLD.Bcast(&n, 1, MPI::INT, 0);
        if (n==0)
            break;
        else {
            h = 1.0 / (double) n;
            sum = 0.0;
            for (i = rank + 1; i <= n; i += size) {
                x = h * ((double)i - 0.5);
                sum += (4.0 / (1.0 + x*x));
            }
            mypi = h * sum;

            MPI::COMM_WORLD.Reduce(&mypi, &pi, 1, MPI::DOUBLE,
                  MPI::SUM, 0);
            if (rank == 0)
                cout << "pi is approximately " << pi << ", Error is " <<
                    fabs(pi - PI25DT) << endl;
```

```
        }
    }
    MPI::Finalize();
    return 0;
}
```

5.2 Ví dụ nhân ma trận

Trong ví dụ này ta sẽ xây dựng một chương trình tính tích của hai ma trận bằng phương pháp tính toán song song. Giả sử ta có ma trận `C` là tích của hai ma trận `A` và `B`, ta có thể khai báo trong C như sau

```
#define NRA 62
#define NCA 15
#define NCB 7

double  a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];
```

trong đó `NRA`, `NCA` và `NCB` lần lượt là số cột và số dòng của ma trận `A`, số dòng của ma trận `B`.

Điểm đặc biệt trong ví dụ này là ta sẽ chia các tác vụ (tiến trình) ra làm hai loại: tác vụ chính và tác vụ con, mỗi loại tác vụ sẽ thực hiện những công việc khác nhau. Để thuận tiện trong việc ghi nhớ, ta có thể khai báo các tham số biểu diễn cho chỉ số của tác vụ chính (`MASTER`), và nhãn (*tag*) để đánh dấu rằng dữ liệu được gửi là từ tác vụ chính (`FROM_MASTER`) hay tác vụ con (`FROM_WORKER`).

```
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2
```

Các công việc của tác vụ chính gồm có

- Khởi tạo các ma trận `A` và `B`

```
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;
```

- Gửi dữ liệu cho các tác vụ con. Trong ví dụ này số cột của ma trận `B` sẽ được chia thành `numworkers` phần tương tứng với số lượng tác vụ con.

```
/* 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 offset=%d\n",rows,dest,
                        offset);
   MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
   MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
   MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
                        MPI_COMM_WORLD);
   MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
```

```
      offset = offset + rows;
   }
```

- Nhận dữ liệu từ tác vụ con

```
   /* Receive results from 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);
      MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
               MPI_COMM_WORLD, &status);
      printf("Received results from task %d\n",source);
   }
```

- In kết quả

```
   /* Print results */
   printf("******************************************************\n");
   printf("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******************************************************\n");
   printf ("Done.\n");
```

Các công việc của tác vụ con gồm có

- Nhận dữ liệu từ tác vụ chính

```
   /* Receive matrix data from master task */
   mtype = FROM_MASTER;
   MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype,MPI_COMM_WORLD,&status);
   MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
   MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &
       status);
   MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &
       status);
```

- Tính toán nhân ma trận

```
   /* Do matrix multiply */
   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];
      }
```

- Trả kết quả về cho tác vụ chính

```
   /* Send results back to master task */
   mtype = FROM_WORKER;
   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);
```

Chương trình nhân hai ma trận hoàn chỉnh như sau

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

#define NRA 62
#define NCA 15
#define NCB 7
#define MASTER 0
#define FROM_MASTER 1
#define FROM_WORKER 2

int main(int argc, char **argv)
{
   int numtasks, taskid, numworkers, source, dest, mtype, rows, averow,
      extra, offset, i, j, k, rc;
   double  a[NRA][NCA], b[NCA][NCB], c[NRA][NCB];
   MPI_Status status;

   MPI_Init(&argc,&argv);
   MPI_Comm_rank(MPI_COMM_WORLD,&taskid);
   MPI_Comm_size(MPI_COMM_WORLD,&numtasks);
   if (numtasks < 2 ) {
      printf("Need at least two MPI tasks. Quitting...\n");
      MPI_Abort(MPI_COMM_WORLD, rc);
      exit(1);
   }
   numworkers = numtasks-1;

/************************ master task ******************************/
if (taskid == MASTER) {
   printf("mpi_mm has started with %d tasks.\n",numtasks);
   printf("Initializing arrays...\n");
   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;

   /* 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 offset=%d\n",rows,dest,offset);
      MPI_Send(&offset, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
      MPI_Send(&rows, 1, MPI_INT, dest, mtype, MPI_COMM_WORLD);
      MPI_Send(&a[offset][0], rows*NCA, MPI_DOUBLE, dest, mtype,
                MPI_COMM_WORLD);
      MPI_Send(&b, NCA*NCB, MPI_DOUBLE, dest, mtype, MPI_COMM_WORLD);
      offset = offset + rows;
```

```
      }

      /* Receive results from 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);
         MPI_Recv(&c[offset][0], rows*NCB, MPI_DOUBLE, source, mtype,
                  MPI_COMM_WORLD, &status);
         printf("Received results from task %d\n",source);
      }

      /* Print results */
      printf("******************************************************\n");
      printf("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******************************************************\n");
      printf ("Done.\n");
   }

   /************************ worker task ******************************/
   if (taskid > MASTER) {
      /* Receive matrix data from master task */
      mtype = FROM_MASTER;
      MPI_Recv(&offset, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD,&status);
      MPI_Recv(&rows, 1, MPI_INT, MASTER, mtype, MPI_COMM_WORLD, &status);
      MPI_Recv(&a, rows*NCA, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &
          status);
      MPI_Recv(&b, NCA*NCB, MPI_DOUBLE, MASTER, mtype, MPI_COMM_WORLD, &
          status);

      /* Do matrix multiply */
      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];
         }

      /* Send results back to master task */
      mtype = FROM_WORKER;
      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);
   }
   MPI_Finalize();
}
```

Tài liệu

[1] William Gropp et al, *MPICH2 User's Guide Version 1.0.6*, Mathematics and Computer Science Division, Argonne National Laboratory, 2007.

[2] Serrano Pereira, *Building a simple Beowulf cluster with Ubuntu*
http://byobu.info/article/Building_a_simple_Beowulf_cluster_with_Ubuntu/

[3] Blaise Barney, *Message Passing Interface (MPI)*
https://computing.llnl.gov/tutorials/mpi/

[4] Paul Burton, *An Introduction to MPI Programming*
http://www.ecmwf.int/services/computing/training/material/hpcf/Intro_MPI_Programming.pdf

[5] Stefano Cozzini, *MPI tutorial*, Democritos/ICTP course in "Tools for computational physics", 2005
http://www.democritos.it/events/computational_physics/lecture_stefano4.pdf

[6] Ngô Văn Thanh, *Tính toán song song*
http://iop.vast.ac.vn/~nvthanh/cours/parcomp/

[7] https://www.surfsara.nl/systems/shared/mpi/mpi-intro

[8] http://chryswoods.com/book/export/html/117

[9] http://beige.ucs.indiana.edu/B673/node150.html

[10] http://www.cs.indiana.edu/classes/b673/notes/mpi1.html

[11] http://geco.mines.edu/workshop/class2/examples/mpi/index.html

[12] http://www.mcs.anl.gov/research/projects/mpi/usingmpi/examples/simplempi/main.htm