

IF3230 Sistem Paralel dan Terdistribusi

Praktikum 1 Eksplorasi OpenMPI

A. Persiapan Praktikum

1. Koneksi SSH

Login dengan ssh ke 167.205.35.25 dengan username <NIM> dan password “guest”. Buat pasangan public/private key dengan menjalankan perintah berikut, kosongkan (tekan *enter*) pada setiap prompt yang diberikan

```
> ssh-keygen -t dsa
```

Copy file public key sebagai authorized keys

```
> cp ~/.ssh/id_dsa.pub ~/.ssh/authorized_keys
```

Agar tidak memerlukan password setiap login, jalankan perintah

```
> eval `ssh-agent`
```

```
> ssh-add ~/.ssh/id_dsa
```

Untuk menambahkan tiap host sebagai known host, jalankan perintah berikut untuk tiap ip address host, yaitu 167.205.35.26 - 167.205.35.34 (kecuali 167.205.35.27)

```
> ssh-keyscan -H 167.205.35.26 >> ~/.ssh/known_hosts
```

```
> ssh-keyscan -H 167.205.35.28 >> ~/.ssh/known_hosts
```

```
...
```

```
> ssh-keyscan -H 167.205.35.34 >> ~/.ssh/known_hosts
```

2. Konfigurasi PATH

Lakukan konfigurasi path dengan menjalankan langkah berikut.

```
> nano ~/.bashrc
```

Tambahkan 2 line berikut pada bagian bawah dari file, dan simpan file tersebut.

```
export PATH="$PATH:/opt/openmpi/bin"
export LD_LIBRARY_PATH="$LD_LIBRARY_PATH:/opt/openmpi/lib"
```

Kemudian, jalankan perintah berikut untuk memasukkan *path* yang telah ditambah pada bashrc ke shell yang sedang bekerja.

```
> . ~/.bashrc
```

3. Hello MPI

Pada 167.205.35.25 buat file *mpi_hostfile* yang menyimpan daftar seluruh host yang akan digunakan, yaitu 167.205.35.26 - 167.205.35.34 (kecuali 167.205.35.27)

```
#daftar host
localhost
167.205.35.26
167.205.35.28
167.205.35.29
167.205.35.30
167.205.35.31
167.205.35.32
167.205.35.33
167.205.35.34
```

Buat file hellompi.c

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

int main(int argc, char *argv[]) {
    int numtasks, rank;
    char processor_name[MPI_MAX_PROCESSOR_NAME];
    int name_len;
    MPI_Status Stat;
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numtasks);
    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Get_processor_name(processor_name, &name_len);
    printf("Hello from processor %s, task %d of %d, \n",
           processor_name, rank, numtasks);
    MPI_Finalize();
}
```

Lakukan kompilasi dengan menjalankan perintah

```
> mpicc <program-name> -o <executable-name>
```

Jalankan program dengan perintah

```
> mpirun -np <jumlah-proses> --hostfile mpi_hostfile <executable-name>
```

Tambahkan `--bind-to core:overload-allowed` jika terdapat permasalahan untuk jumlah proses yang besar.

B. Eksplorasi OpenMPI

Pada praktikum kali ini, peserta praktikum diharuskan untuk mengerjakan ulang (membuat dan menjalankan program) 5 eksperimen yang merupakan fitur utama dari OpenMPI. Petunjuk dan *source code* dari masing-masing program yang akan dibuat dapat dilihat pada sub-bab berikut. Pada praktikum ini peserta disarankan untuk memodifikasi kode dengan syarat program yang dibuat memiliki hasil akhir yang serupa dengan kode program yang telah ditampilkan.

Materi pembahasan ini akan digunakan pula pada praktikum berikutnya sehingga diharapkan peserta mengerti materi pembahasan dan tidak hanya menjalankan program.

a. Send and Receive

Sending & receiving adalah dua konsep fundamental dari MPI. Pada dasarnya fungsi-fungsi yang ada pada MPI dapat diimplementasikan menggunakan kedua fungsi ini. Berikut adalah program yang menjalankan send dan receive serta prototipe prosedur dari kedua fungsi tersebut.

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

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

```
// Copyright www.computing.llnl.gov  
#include "mpi.h"  
#include <stdio.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);
    }

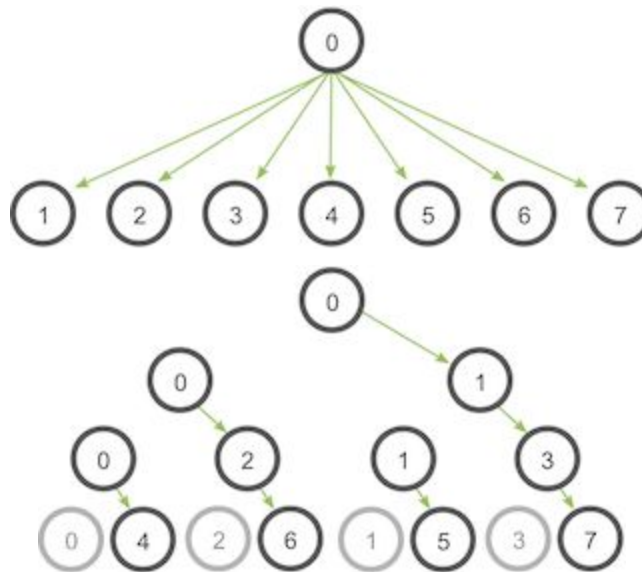
    rc = MPI_Get_count(&Stat, MPI_CHAR, &count);
    printf("Task %d: Received %d char(s) from task %d with tag %d \n",
           rank, count, Stat.MPI_SOURCE, Stat.MPI_TAG);

    MPI_Finalize();
}

```

b. Broadcast

Broadcast adalah suatu teknik komunikasi dengan sifat yang kolektif. Pada saat suatu proses melakukan broadcast, proses tersebut akan mengirimkan data yang sama kepada semua proses lainnya. Kegunaan dari broadcast ini adalah untuk mengirimkan nilai secara cepat kepada seluruh proses yang ada dibandingkan dengan melakukan Send ke seluruh proses. Keunggulan dari Broadcast dibandingkan dengan Send adalah, Broadcast menggunakan lebih dari jalur komunikasi dengan bentuk seperti pohon sehingga memaksimalkan utilisasi jaringan yang ada. Ilustrasi dari komunikasi pada Broadcast dapat dilihat pada dua gambar dibawah.



Berikut adalah prototipe prosedur serta program yang menjalankan fungsi Broadcast dibandingkan dengan fungsi Send, serta perbandingannya menggunakan timing.

```

MPI_Bcast (
    void* data,
    int count,

```

```
MPI_Datatype datatype,  
int root,  
MPI_Comm communicator)
```

```
// Copyright 2011 www.mpitutorial.com  
// Comparison of MPI_Bcast with the my_bcast function  
  
#include <stdio.h>  
#include <stdlib.h>  
#include <mpi.h>  
#include <assert.h>  
  
void my_bcast(void* data, int count, MPI_Datatype datatype, int root,  
             MPI_Comm communicator) {  
    int world_rank;  
    MPI_Comm_rank(communicator, &world_rank);  
    int world_size;  
    MPI_Comm_size(communicator, &world_size);  
  
    if (world_rank == root) {  
        // If we are the root process, send our data to everyone  
        int i;  
        for (i = 0; i < world_size; i++) {  
            if (i != world_rank) {  
                MPI_Send(data, count, datatype, i, 0, communicator);  
            }  
        }  
    } else {  
        // If we are a receiver process, receive the data from the root  
        MPI_Recv(data, count, datatype, root, 0, communicator, MPI_STATUS_IGNORE);  
    }  
}  
  
int main(int argc, char** argv) {  
    if (argc != 3) {  
        fprintf(stderr, "Usage: compare_bcast num_elements num_trials\n");  
        exit(1);  
    }  
  
    int num_elements = atoi(argv[1]);  
    int num_trials = atoi(argv[2]);  
  
    MPI_Init(NULL, NULL);  
  
    int world_rank;  
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);  
  
    double total_my_bcast_time = 0.0;  
    double total_mpi_bcast_time = 0.0;  
    int i;  
    int* data = (int*)malloc(sizeof(int) * num_elements);  
    assert(data != NULL);  
  
    for (i = 0; i < num_trials; i++) {  
        // Time my_bcast  
        // Synchronize before starting timing
```

```

MPI_Barrier(MPI_COMM_WORLD);
total_my_bcast_time -= MPI_Wtime();
my_bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
// Synchronize again before obtaining final time
MPI_Barrier(MPI_COMM_WORLD);
total_my_bcast_time += MPI_Wtime();

// Time MPI_Bcast
MPI_Barrier(MPI_COMM_WORLD);
total_mpi_bcast_time -= MPI_Wtime();
MPI_Bcast(data, num_elements, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Barrier(MPI_COMM_WORLD);
total_mpi_bcast_time += MPI_Wtime();
}

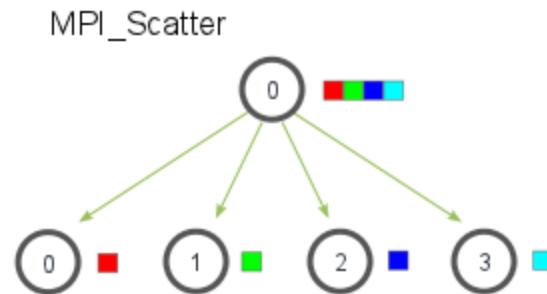
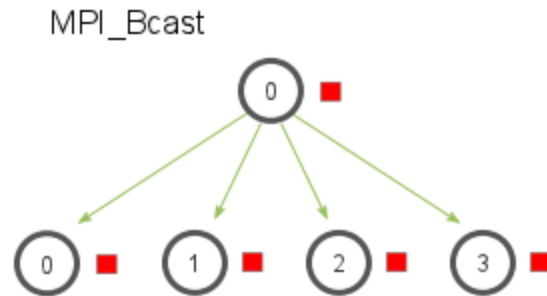
// Print off timing information
if (world_rank == 0) {
    printf("Data size = %d, Trials = %d\n", num_elements * (int)sizeof(int),
           num_trials);
    printf("Avg my_bcast time = %lf\n", total_my_bcast_time / num_trials);
    printf("Avg MPI_Bcast time = %lf\n", total_mpi_bcast_time / num_trials);
}

MPI_Finalize();
}

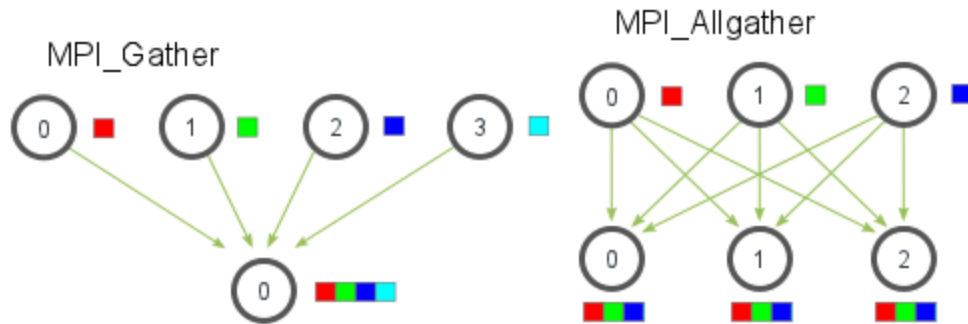
```

c. Scatter - Gather

Scatter adalah operasi penyebaran data. Data disebar dari satu task sumber ke setiap task lainnya pada grup. *Perbedaan* broadcast dengan scatter adalah pada broadcast data yang dikirim ke task lain berupa seluruh data, sedangkan pada scatter data yang dikirim ke task lain dibagi-bagi sehingga setiap task mendapat bagian data yang unik. Ilustrasi perbedaan scatter dan broadcast dapat dilihat di gambar di bawah ini.



Gather adalah operasi pengumpulan data. Data dikumpulkan dari setiap task dari grup ke satu task tujuan. Gather adalah operasi yang merupakan kebalikan dari Scatter. Selain gather, terdapat juga operasi allgather. Operasi allgather mengumpulkan data dari setiap task ke semua task yang terdapat di dalam grup. Perbedaan gather dengan allgather dapat dilihat pada gambar di bawah ini.



Scatter dilakukan dengan memanggil prosedur MPI_Scatter, sedangkan gather dilakukan dengan memanggil prosedur MPI_Gather. Prototipe prosedur MPI_Scatter dan MPI_Gather adalah sebagai berikut.

<pre> MPI_Scatter(void* send_data, int send_count, MPI_Datatype send_datatype, void* recv_data, int recv_count, MPI_Datatype recv_datatype, int root, MPI_Comm communicator) </pre>	<pre> 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 communicator) </pre>
--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Berikut adalah contoh program yang memanfaatkan scatter dan gather untuk mendapatkan nilai rata-rata dari data acak secara paralel.

```
// Copyright 2012 www.mpitutorial.com

// Program yang menghitung rata-rata dari array secara paralel menggunakan
// Scatter dan Gather.
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <mpi.h>
#include <assert.h>

float *create_rand_nums(int num_elements) {
    float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
    assert(rand_nums != NULL);
    int i;
    for (i = 0; i < num_elements; i++) {
        rand_nums[i] = (rand() / (float)RAND_MAX);
    }
    return rand_nums;
}

float compute_avg(float *array, int num_elements) {
    float sum = 0.f;
    int i;
    for (i = 0; i < num_elements; i++) {
        sum += array[i];
    }
    return sum / num_elements;
}

int main(int argc, char** argv) {
    if (argc != 2) {
        fprintf(stderr, "Usage: avg num_elements_per_proc\n");
        exit(1);
    }

    int num_elements_per_proc = atoi(argv[1]);
    srand(time(NULL));

    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    float *rand_nums = NULL;
    if (world_rank == 0) {
        rand_nums = create_rand_nums(num_elements_per_proc * world_size);
    }

    float *sub_rand_nums = (float *)malloc(sizeof(float) *
num_elements_per_proc);
    assert(sub_rand_nums != NULL);
```



```

    MPI_Scatter(rand_nums, num_elements_per_proc, MPI_FLOAT, sub_rand_nums,
num_elements_per_proc, MPI_FLOAT, 0, MPI_COMM_WORLD);

    float sub_avg = compute_avg(sub_rand_nums, num_elements_per_proc);

    float *sub_avgs = NULL;
    if (world_rank == 0) {
        sub_avgs = (float *)malloc(sizeof(float) * world_size);
        assert(sub_avgs != NULL);
    }
    MPI_Gather(&sub_avg, 1, MPI_FLOAT, sub_avgs, 1, MPI_FLOAT, 0,
MPI_COMM_WORLD);

    if (world_rank == 0) {
        float avg = compute_avg(sub_avgs, world_size);
        printf("Avg of all elements is %f\n", avg);
    }

    if (world_rank == 0) {
        free(rand_nums);
        free(sub_avgs);
    }
    free(sub_rand_nums);

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}

```

d. Reduce

Reduce adalah operasi pengumpulan data. Data dari setiap task dikirim ke satu task root. Task root akan memiliki nilai yang merupakan hasil perhitungan tertentu terhadap setiap data yang diterima oleh root. Selain terdapat reduce terdapat juga allreduce. Perbedaan reduce dengan allreduce adalah pada reduce hanya terdapat satu task yang memperoleh hasil perhitungan, sedangkan pada allreduce setiap task dalam grup memperoleh hasil perhitungan.

Reduce dilakukan dengan memanggil prosedur MPI_Reduce. Prototipe prosedur MPI_Reduce terdapat di bawah. Parameter *op* adalah penentu perhitungan apa yang akan dilakukan pada data yang ada. Op dapat bernilai MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_LOR, MPI_BAND, MPI_BOR, MPI_MAXLOC, atau MPI_MINLOC. Arti setiap operator dapat dilihat pada dokumentasi MPI yang tersedia online.

```

MPI_Reduce(
    void* send_data,
    void* recv_data,
    int count,
    MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm communicator)

```

```

// Copyright 2013 www.mpitutorial.com
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
#include <assert.h>

float *create_rand_nums(int num_elements) {
    float *rand_nums = (float *)malloc(sizeof(float) * num_elements);
    assert(rand_nums != NULL);
    int i;
    for (i = 0; i < num_elements; i++) {
        rand_nums[i] = (rand() / (float)RAND_MAX);
    }
    return rand_nums;
}

int main(int argc, char** argv) {
    if (argc != 2) {
        fprintf(stderr, "Usage: avg num_elements_per_proc\n");
        exit(1);
    }

    int num_elements_per_proc = atoi(argv[1]);

    MPI_Init(NULL, NULL);

    int world_rank;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    int world_size;
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    srand(time(NULL)*world_rank);
    float *rand_nums = NULL;
    rand_nums = create_rand_nums(num_elements_per_proc);

    float local_sum = 0;
    int i;
    for (i = 0; i < num_elements_per_proc; i++) {
        local_sum += rand_nums[i];
    }

    printf("Local sum for process %d - %f, avg = %f\n", world_rank, local_sum,
        local_sum / num_elements_per_proc);

    float global_sum;
    MPI_Reduce(&local_sum, &global_sum, 1, MPI_FLOAT, MPI_SUM, 0,
        MPI_COMM_WORLD);

    if (world_rank == 0) {
        printf("Total sum = %f, avg = %f\n", global_sum, global_sum /
            (world_size * num_elements_per_proc));
    }

    // Clean up
    free(rand_nums);

    MPI_Barrier(MPI_COMM_WORLD);
    MPI_Finalize();
}

```

e. Communicator dan Group

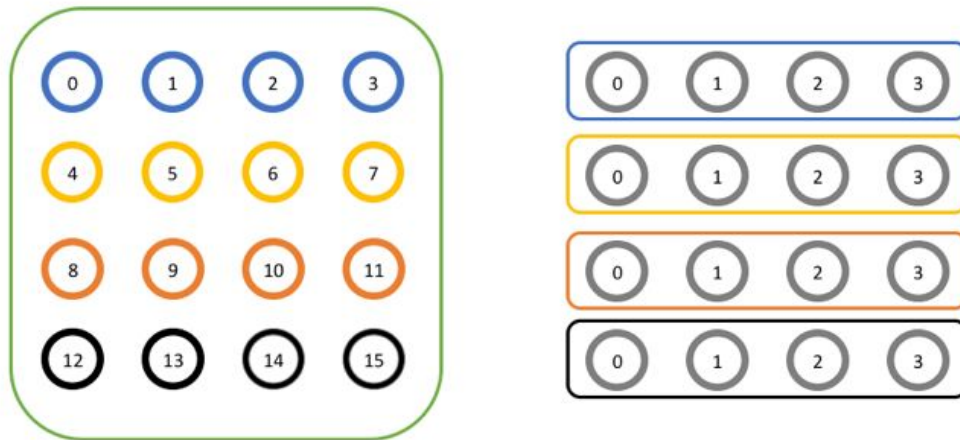
Communicator merupakan objek yang dipakai untuk melakukan komunikasi untuk prosedur yang melibatkan communicator seperti broadcast, scatter, gather, dan reduce. MPI_COMM_WORLD adalah communicator yang didefinisikan secara otomatis oleh MPI ketika kita menjalankan program MPI. Communicator tersebut dipakai untuk melakukan komunikasi kepada semua task yang ada. Sebuah task dapat memperoleh rank dan size task tersebut relatif terhadap communicator. Communicator secara internal mengandung id dan group. Id dipakai oleh MPI untuk membedakan satu communicator dengan yang lainnya. Sedangkan group adalah himpunan dari task yang terlibat dalam communicator tersebut. Sama seperti himpunan dalam matematika, group memiliki operasi union, intersection, dll.

Kadang-kadang kita tidak ingin melakukan komunikasi dengan semua task yang ada, melainkan hanya sebagian task. Untuk itu kita memerlukan communicator untuk sebagian task. Untuk menciptakan communicator yang baru, kita dapat menggunakan MPI_Comm_split, MPI_Comm_dup, atau MPI_Comm_create_group. Berikut adalah prototipe prosedur disebutkan sebelumnya.

<pre>MPI_Comm_split(MPI_Comm comm, int color, int key, MPI_Comm* newcomm)</pre>	<pre>MPI_Comm_dup (MPI_Comm comm, MPI_Comm *comm_out)</pre>	<pre>MPI_Comm_create_group(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm* newcomm))</pre>
------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------	-------------------------------------------------------------------------------------------------------------------------------------

MPI_Comm_split memecah communicator yang sudah ada menjadi communicator yang baru. Communicator yang terbentuk adalah communicator yang mewakili task-task dengan color yang diberikan. Untuk lebih jelasnya dapat dilihat di gambar di bawah ini. Pada contoh tersebut, setiap task dalam MPI_COMM_WORLD memberikan color world_rank/4 dan memberikan key world_rank. Communicator yang tercipta pada task dengan world_rank 0 adalah communicator untuk task dengan world_rank 0, 1, 2, dan 3. Communicator yang tercipta pada task dengan world_rank 15 adalah communicator untuk task dengan world rank 12, 13, 14, dan 15.

Split a Large Communicator Into Smaller Communicators



`MPI_Comm_dup` melakukan duplikasi communicator. Duplikasi diperlukan ketika kita ingin menjalankan perhitungan yang berbeda pada task yang sama. `MPI_Comm_create_group` dipakai untuk membuat communicator dengan group yang ada. Berikut adalah contoh pembuatan communicator menggunakan group.

```
// Copyright 2015 www.mpitutorial.com
#include <stdlib.h>
#include <stdio.h>
#include <mpi.h>

int main(int argc, char **argv) {
    MPI_Init(NULL, NULL);

    int world_rank, world_size;
    MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &world_size);

    MPI_Group world_group;
    MPI_Comm_group(MPI_COMM_WORLD, &world_group);

    int n = 7;
    const int ranks[7] = {1, 2, 3, 5, 7, 11, 13};

    MPI_Group prime_group;
    MPI_Group_incl(world_group, 7, ranks, &prime_group);

    MPI_Comm prime_comm;
    MPI_Comm_create_group(MPI_COMM_WORLD, prime_group, 0, &prime_comm);

    int prime_rank = -1, prime_size = -1;
    if (MPI_COMM_NULL != prime_comm) {
        MPI_Comm_rank(prime_comm, &prime_rank);
        MPI_Comm_size(prime_comm, &prime_size);
    }
}
```

```
printf("WORLD RANK/SIZE: %d/%d --- PRIME RANK/SIZE: %d/%d\n", world_rank,
world_size, prime_rank, prime_size);

MPI_Group_free(&world_group);
MPI_Group_free(&prime_group);

if (MPI_COMM_NULL != prime_comm) {
    MPI_Comm_free(&prime_comm);
}

MPI_Finalize();
}
```

C. Pengumpulan

- Lakukan commit setiap kali anda selesai mengerjakan satu program
- Lakukan commit setiap kali anda selesai mengerjakan satu program
- Soal yang wajib di kumpulkan dan di commit adalah program broadcast dan scatter-gather
- Lakukan pull request kepada kode anda di akhir praktikum untuk penilaian
- Segala bentuk kecurangan akan mendapatkan konsekuensi