CPSC 4770/6770

Distributed and Cluster Computing

Lecture 8: MPI Datatypes, Communicators and Groups

MPI Datatypes

- MPI provides basic datatypes
  - MPI_INT, MPI_LONG, MPI_CHAR, etc
- MPI also provides derived datatypes
  - Allows you to define your own data structures based upon sequences of the MPI primitive data types
    - Contiguous
    - Vector
    - Indexed
    - Struct
  - New datatypes must be committed before they are used
    - int **MPI_Type_commit**(MPI_Datatype *datatype): Commits new datatype to the system
  - Deallocate a specified datatype object
    - int MPI_Type_free(MPI_Datatype *datatype): Frees the datatype

Goal: Pack related data together to reduce total messages!

Contiguous Derived Data Type

- int **MPI_Type_contiguous**(int count, MPI_Datatype old_type, MPI_Datatype *newtype): Produces a new data type by making *count* copies of an existing data type
- For example,

```
MPI_Type_contiguous(count, MPI_CHAR, &mytype);
MPI_Type_commit(&mytype);
MPI_Send(data, 1, mytype, 1, 99, MPI_COMM_WORLD);
MPI_Type_free(&mytype);
```

is equivalent to:

```
MPI_Send(data, count, MPI_CHAR, 1, 99, MPI_COMM_WORLD);
```

contiguous.c

Create a data type representing a row of an array and distribute different rows to all processes

count = 4;
MPI_Type_contiguous(count, MPI_FLOAT, &rowtype);

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

a[4][4]

MPI_Send(&a[2][0], 1, rowtype, dest, tag, comm);

9.0	10.0	11.0	12.0

1 element of rowtype

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

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 rowtype;   // required variable

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

   // create contiguous derived data type
   MPI_Type_contiguous(SIZE, MPI_FLOAT, &rowtype);
   MPI_Type_commit(&rowtype);

   if (numtasks == SIZE) {
      // task 0 sends one element of rowtype to all tasks
      if (rank == 0) {
         for (i=0; i<numtasks; i++)
           MPI_Send(&a[i][0], 1, rowtype, i, tag, MPI_COMM_WORLD);
         }

      // all tasks receive rowtype data from task 0
      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);

   // free datatype when done using it
   MPI_Type_free(&rowtype);
   MPI_Finalize();
}
```

```
[jin6@node0888 datatype]$ mpirun -np 4 contiguous
rank= 0   b= 1.0 2.0 3.0 4.0
rank= 1   b= 5.0 6.0 7.0 8.0
rank= 2   b= 9.0 10.0 11.0 12.0
rank= 3   b= 13.0 14.0 15.0 16.0
```

Vector Derived Data Type

- int **MPI_Type_vector**(int count, int blocklen, int stride, MPI_Datatype old_type, MPI_Datatype *newtype): Creates a vector (strided) datatype
  - count: number of blocks
  - blocklen: number of elements in each block
  - stride: number of elements between start of each block
- It is similar to contiguous, but allows for regular gaps (stride) in the displacements

vector.c

Create a data type representing a column of an array and distribute different columns to all processes

count = 4; blocklength = 1; stride = 4;
MPI_Type_vector(count, blocklength, stride, MPI_FLOAT, &columntype);

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

a[4][4]

MPI_Send(&a[0][1], 1, columntype, dest, tag, comm);

2.0	6.0	10.0	14.0

1 element of columntype

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

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;   // required variable


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

   // create vector derived data type
   MPI_Type_vector(SIZE, 1, SIZE, MPI_FLOAT, &columntype);
   MPI_Type_commit(&columntype);

   if (numtasks == SIZE) {
      // task 0 sends one element of columntype to all tasks
      if (rank == 0) {
         for (i=0; i<numtasks; i++)
            MPI_Send(&a[0][i], 1, columntype, i, tag, MPI_COMM_WORLD);
         }

      // all tasks receive columntype data from task 0
      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);

   // free datatype when done using it
   MPI_Type_free(&columntype);
   MPI_Finalize();
}
```

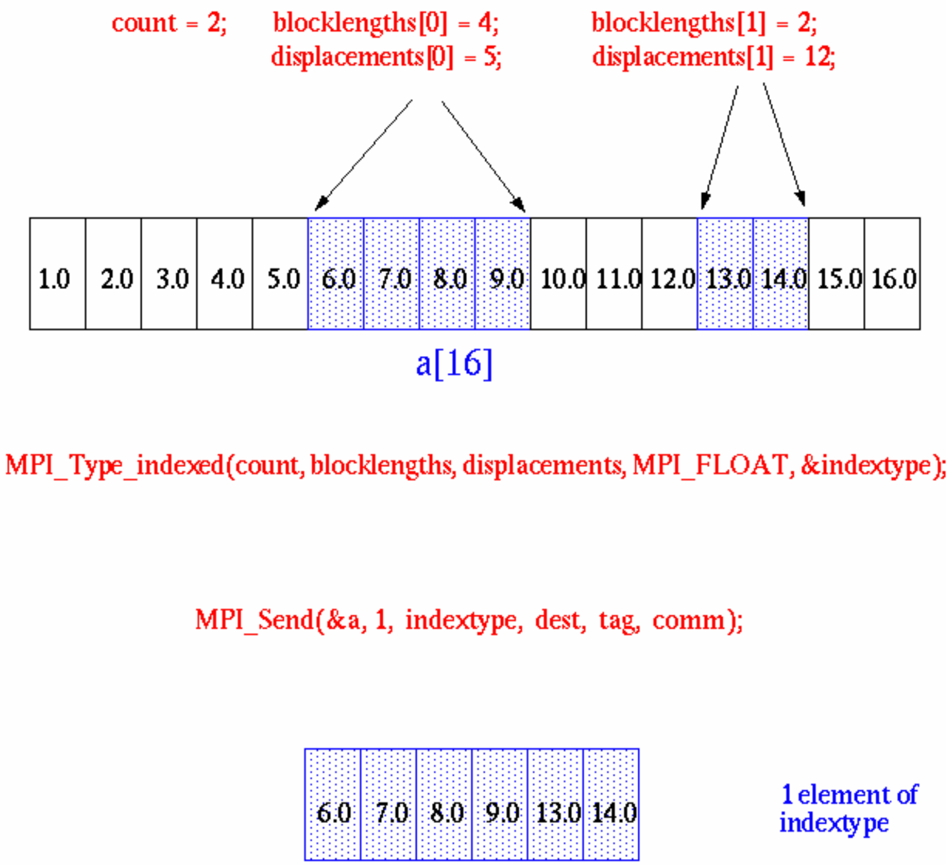
```
[jin6@node0888 datatype]$ mpirun -np 4 vector
rank= 0  b= 1.0 5.0 9.0 13.0
rank= 2  b= 3.0 7.0 11.0 15.0
rank= 3  b= 4.0 8.0 12.0 16.0
rank= 1  b= 2.0 6.0 10.0 14.0
```

Indexed Derived Data Type

- int **MPI_Type_indexed**(int count, int blocklens[], int indices[], MPI_Datatype old_type, MPI_Datatype *newtype): Creates an indexed datatype
  - count: number of blocks
  - blocklens: number of elements in each block
  - indices: displacement of each block in multiples of old_type

index.c

Create a datatype by extracting variable portions of an array and distribute to all tasks




```c
#include "mpi.h"
#include <stdio.h>
#define NELEMENTS 6

main(int argc, char *argv[])  {
   int numtasks, rank, source=0, dest, tag=1, i;
   int blocklengths[2], displacements[2];
   float a[16] =
     {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[NELEMENTS];

   MPI_Status stat;
   MPI_Datatype indextype;   // required variable

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

   blocklengths[0] = 4;
   blocklengths[1] = 2;
   displacements[0] = 5;
   displacements[1] = 12;

   // create indexed derived data type
   MPI_Type_indexed(2, blocklengths, displacements, MPI_FLOAT, &indextype);
   MPI_Type_commit(&indextype);

   if (rank == 0) {
     for (i=0; i<numtasks; i++)
      // task 0 sends one element of indextype to all tasks
        MPI_Send(a, 1, indextype, i, tag, MPI_COMM_WORLD);
     }

   // all tasks receive indextype data from task 0
   MPI_Recv(b, NELEMENTS, MPI_FLOAT, source, tag, MPI_COMM_WORLD, &stat);
   printf("rank= %d  b= %3.1f %3.1f %3.1f %3.1f %3.1f %3.1f\n",
          rank,b[0],b[1],b[2],b[3],b[4],b[5]);

   // free datatype when done using it
   MPI_Type_free(&indextype);
   MPI_Finalize();
}
```

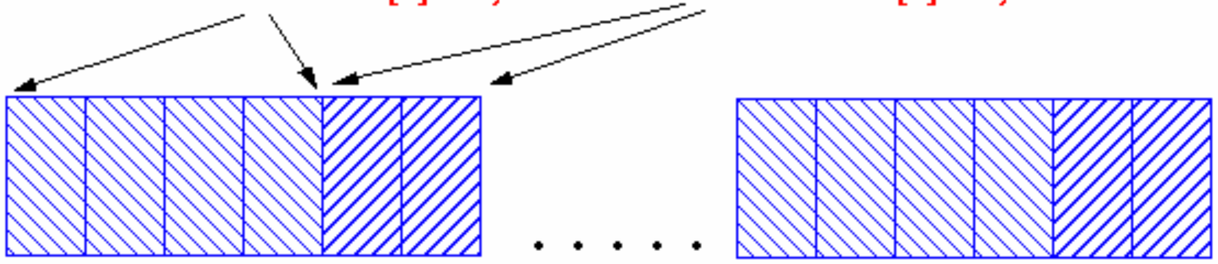
```
[jin6@node0888 datatype]$ mpirun -np 4 index
rank= 2  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 3  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 0  b= 6.0 7.0 8.0 9.0 13.0 14.0
rank= 1  b= 6.0 7.0 8.0 9.0 13.0 14.0
```

Struct Derived Data Type

- int **MPI_Type_struct**(int count, int blocklens[], MPI_Aint indices[], MPI_Datatype old_types[], MPI_Datatype *newtype): Creates a struct datatype
  - count: number of blocks
  - blocklens: number of elements in each block
  - indices: byte displacement of each block
  - old_types: type of elements in each block
- int **MPI_Type_extent**(MPI_Datatype datatype, MPI_Aint *extent): Returns the extent of a datatype

MPI_Type_struct Example

```
typedef struct { float x, y, z, velocity; int n, type; } Particle;
Particle particles[NELEM];

MPI_Type_extent(MPI_FLOAT, &extent);

count = 2; oldtypes[0] = MPI_FLOAT;     oldtypes[1] = MPI_INT
           offsets[0] = 0;              offsets[1] = 4 * extent;
           blockcounts[0] = 4;          blockcounts[1] = 2;
```



particles[NELEM]

```
MPI_Type_struct(count, blockcounts, offsets, oldtypes, &particletype);

MPI_Send(particles, NELEM, particletype, dest, tag, comm);
```

Sends entire (NELEM) array of particles, each particle being comprised four floats and two integers.

struct.c

Create a datatype by extracting variable portions of an array and distribute to all tasks

```c
#include "mpi.h"
#include <stdio.h>
#define NELEM 25

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

   typedef struct {
     float x, y, z;
     float velocity;
     int  n, type;
     }          Particle;
   Particle     p[NELEM], particles[NELEM];
   MPI_Datatype particletype, oldtypes[2];   // required variables
   int          blockcounts[2];

   // MPI_Aint type used to be consistent with syntax of
   // MPI_Type_extent routine
   MPI_Aint    offsets[2], extent;

   MPI_Status stat;

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

   // setup description of the 4 MPI_FLOAT fields x, y, z, velocity
   offsets[0] = 0;
   oldtypes[0] = MPI_FLOAT;
   blockcounts[0] = 4;

   // setup description of the 2 MPI_INT fields n, type
   // need to first figure offset by getting size of MPI_FLOAT
   MPI_Type_extent(MPI_FLOAT, &extent);
   offsets[1] = 4 * extent;
   oldtypes[1] = MPI_INT;
   blockcounts[1] = 2;

   // define structured type and commit it
   MPI_Type_struct(2, blockcounts, offsets, oldtypes, &particletype);
   MPI_Type_commit(&particletype);

   // task 0 initializes the particle array and then sends it to each task
   if (rank == 0) {
     for (i=0; i<NELEM; i++) {
        particles[i].x = i * 1.0;
        particles[i].y = i * -1.0;
        particles[i].z = i * 1.0;
        particles[i].velocity = 0.25;
        particles[i].n = i;
        particles[i].type = i % 2;
        }
     for (i=0; i<numtasks; i++)
        MPI_Send(particles, NELEM, particletype, i, tag, MPI_COMM_WORLD);
     }

   // all tasks receive particletype data
   MPI_Recv(p, NELEM, particletype, source, tag, MPI_COMM_WORLD, &stat);

   printf("rank= %d   %3.2f %3.2f %3.2f %3.2f %d %d\n", rank,p[3].x,
        p[3].y,p[3].z,p[3].velocity,p[3].n,p[3].type);

   // free datatype when done using it
   MPI_Type_free(&particletype);
   MPI_Finalize();
}
```

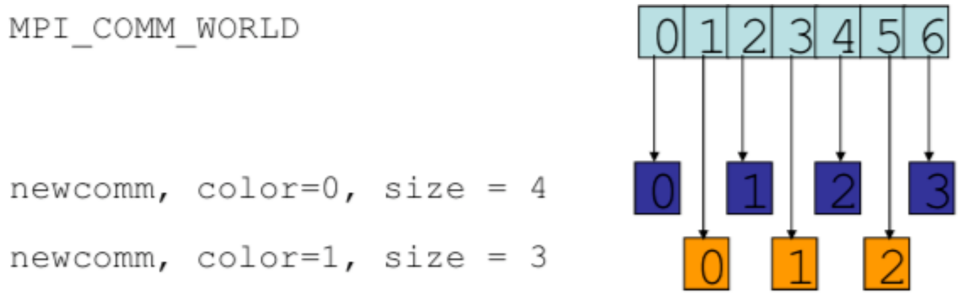
```
[jin6@node0888 datatype]$ mpirun -np 4 struct
rank= 2   3.00 -3.00 3.00 0.25 3 1
rank= 3   3.00 -3.00 3.00 0.25 3 1
rank= 0   3.00 -3.00 3.00 0.25 3 1
rank= 1   3.00 -3.00 3.00 0.25 3 1
```

Overview of Communicators

- An MPI communicator (MPI_Comm) is an object containing a group of processes that may communicate with each other
  - Predefined communicators
    - MPI_COMM_WORLD: contains all processes started with mpirun/mpiexec
    - MPI_COMM_SELF: contains just the local process itself, size is always 1
  - Create new communicators
    - Splitting the original communicator into n-parts
    - Creating subgroups of the original communicator
    - Re-ordering of processes based on topology information
    - Spawn new processes
    - Connect two applications and merge their communicators

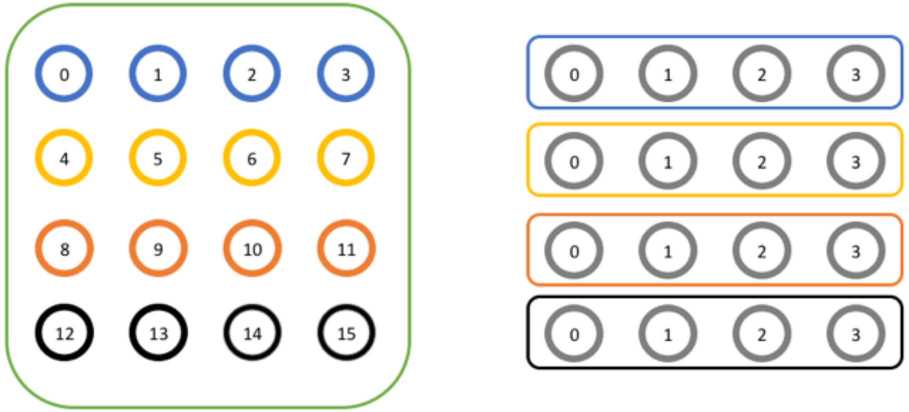
Splitting a Communicator

- int **MPI_Comm_split**(MPI_Comm comm, int color, int key, MPI_Comm *newcomm): Creates new communicators based on colors and keys
  - All processes having the same color will be in the same subcommunicator
    - A process can just be part of one of the generated communicators
    - A process cannot "see" the other communicators
    - A process cannot "see" how many communicators have been created
    - If a process shall not be part of any of the resulting communicators
      - Set *color* to MPI_UNDEFINED (newcomm will be MPI_COMM_NULL)
  - Order processes with the same color according to the *key* value



MPI_Comm_split Example

Split a Large Communicator Into Smaller Communicators




```
WORLD RANK/SIZE: 0/16    ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 1/16    ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 2/16    ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 3/16    ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 4/16    ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 5/16    ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 6/16    ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 7/16    ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 8/16    ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 9/16    ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 10/16   ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 11/16   ROW RANK/SIZE: 3/4
WORLD RANK/SIZE: 12/16   ROW RANK/SIZE: 0/4
WORLD RANK/SIZE: 13/16   ROW RANK/SIZE: 1/4
WORLD RANK/SIZE: 14/16   ROW RANK/SIZE: 2/4
WORLD RANK/SIZE: 15/16   ROW RANK/SIZE: 3/4
```

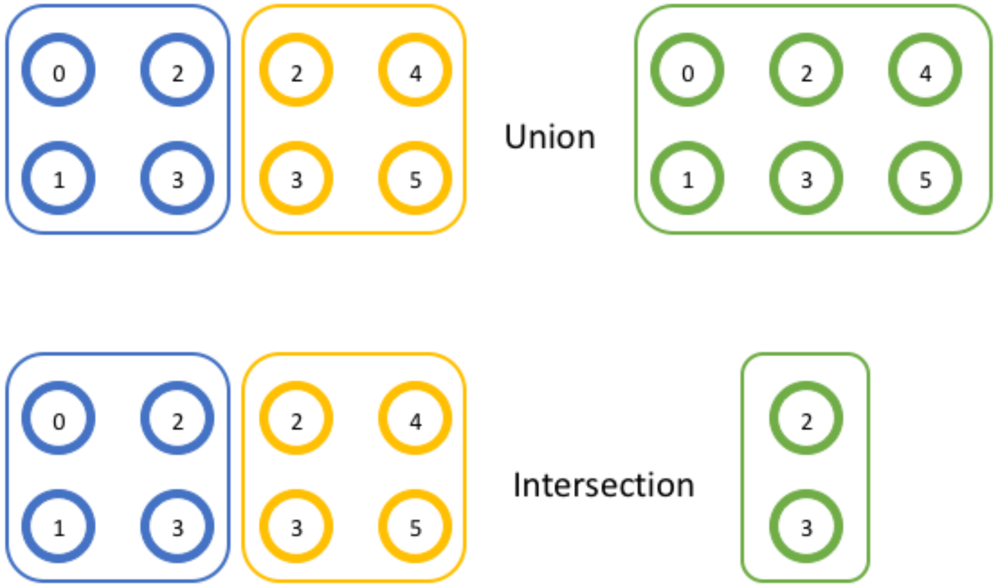
```c
// Get the rank and size in the original communicator
int world_rank, world_size;
MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
MPI_Comm_size(MPI_COMM_WORLD, &world_size);

int color = world_rank / 4; // Determine color based on row

// Split the communicator based on the color and use the
// original rank for ordering
MPI_Comm row_comm;
MPI_Comm_split(MPI_COMM_WORLD, color, world_rank, &row_comm);

int row_rank, row_size;
MPI_Comm_rank(row_comm, &row_rank);
MPI_Comm_size(row_comm, &row_size);

printf("WORLD RANK/SIZE: %d/%d \t ROW RANK/SIZE: %d/%d\n",
        world_rank, world_size, row_rank, row_size);

MPI_Comm_free(&row_comm);
```

Other Communicator Creation Functions

- int **MPI_Comm_dup**(MPI_Comm comm, MPI_Comm *newcomm): Duplicates an existing communicator with all its cached information
- int **MPI_Comm_create**(MPI_Comm comm, MPI_Group group, MPI_Comm *newcomm): Creates a new communicator
  - group: a subset of the group of comm
  - Be aware of the difference between MPI_Comm_create_group function (stay tuned)

Overview of Groups

- An MPI Group (MPI_Group) is the object describing the list of process forming a logical entity
  - A group has a size
    - MPI_Group_size
  - Every process in the group has a unique rank between 0 and (size of group-1)
    - MPI_Group_rank
  - A group is a local object, and can not be used for any communication



Using MPI Groups

- int **MPI_Comm_group**(MPI_Comm comm, MPI_Group *group): Accesses the group associated with given communicator
- int **MPI_Group_union**(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup): Produces a group by combining two groups
- int **MPI_Group_intersection**(MPI_Group group1, MPI_Group group2, MPI_Group *newgroup): Produces a group as the intersection of two existing groups
- int **MPI_Comm_create_group**(MPI_Comm comm, MPI_Group group, int tag, MPI_Comm * newcomm): Creates a new communicator
- int **MPI_Group_incl**(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup): Produces a group by reordering an existing group and taking only listed members
- int **MPI_Group_excl**(MPI_Group group, int n, const int ranks[], MPI_Group *newgroup): Produces a group by reordering an existing group and taking only unlisted members

incl.c

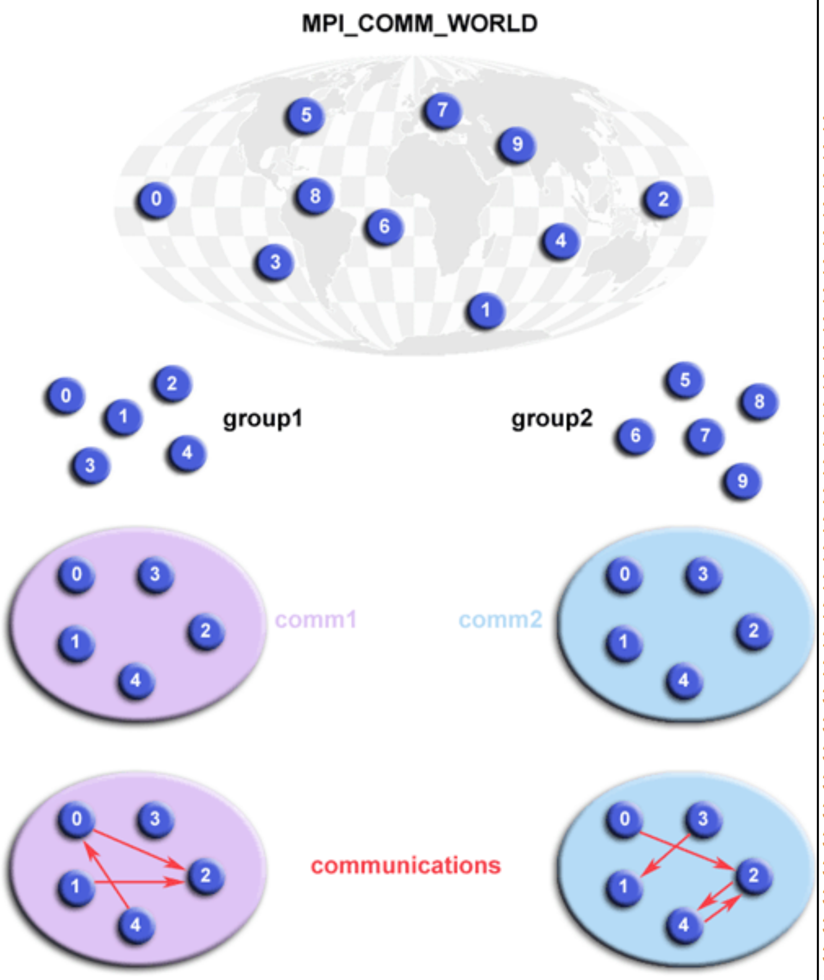
Creating a communicator which contains the prime ranks from MPI_COMM_WORLD

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

int main(int argc, char *argv[]) {
  // Get the rank and size in the original communicator
  int world_rank, world_size;

  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &world_rank);
  MPI_Comm_size(MPI_COMM_WORLD, &world_size);

  // Get the group of processes in MPI_COMM_WORLD
  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};

  // Construct a group containing all of the prime ranks in world_group
  MPI_Group prime_group;
  MPI_Group_incl(world_group, 7, ranks, &prime_group);

  // Create a new communicator based on the 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 this rank isn't in the new communicator, it will be
  // MPI_COMM_NULL. Using MPI_COMM_NULL for MPI_Comm_rank or
  // MPI_Comm_size is erroneous
  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 \t PRIME RANK/SIZE: %d/%d\n",
        world_rank, world_size, prime_rank, prime_size);
    MPI_Comm_free(&prime_comm);
  }

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

  MPI_Finalize();
}
```

```
[jin6@node1684 08-datatype-communicator-group]$ mpirun -np 16 incl
WORLD RANK/SIZE: 1/16    PRIME RANK/SIZE: 0/7
WORLD RANK/SIZE: 2/16    PRIME RANK/SIZE: 1/7
WORLD RANK/SIZE: 3/16    PRIME RANK/SIZE: 2/7
WORLD RANK/SIZE: 5/16    PRIME RANK/SIZE: 3/7
WORLD RANK/SIZE: 7/16    PRIME RANK/SIZE: 4/7
WORLD RANK/SIZE: 11/16   PRIME RANK/SIZE: 5/7
WORLD RANK/SIZE: 13/16   PRIME RANK/SIZE: 6/7
```

Freeing groups and communicators

group.c




```c
#include "mpi.h"
#include <stdlib.h>
#include <stdio.h>
#define NPROCS 10

main(int argc, char *argv[])  {
   int rank, new_rank, sendbuf, recvbuf, numtasks,
       ranks1[5]={0,1,2,3,4}, ranks2[5]={5,6,7,8,9};
   MPI_Group orig_group, new_group;   // required variables
   MPI_Comm new_comm;   // required variable

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

   if (numtasks != NPROCS) {
     printf("Must specify MP_PROCS= %d. Terminating.\n",NPROCS);
     MPI_Finalize();
     exit(0);
   }

   sendbuf = rank;

   // extract the original group handle
   MPI_Comm_group(MPI_COMM_WORLD, &orig_group);

   //  divide tasks into two distinct groups based upon rank
   if (rank < NPROCS/2) {
     MPI_Group_incl(orig_group, NPROCS/2, ranks1, &new_group);
   } else {
     MPI_Group_incl(orig_group, NPROCS/2, ranks2, &new_group);
   }

   // create new new communicator and then perform collective communications
   MPI_Comm_create(MPI_COMM_WORLD, new_group, &new_comm);
   MPI_Allreduce(&sendbuf, &recvbuf, 1, MPI_INT, MPI_SUM, new_comm);

   // get rank in new group
   MPI_Group_rank (new_group, &new_rank);
   printf("rank= %d newrank= %d recvbuf= %d\n",rank,new_rank,recvbuf);

   MPI_Finalize();
}
```

```
[jin6@node1555 group]$ mpicc group.c -o group
[jin6@node1555 group]$ mpirun -np 10 group
rank= 0 newrank= 0 recvbuf= 10
rank= 1 newrank= 1 recvbuf= 10
rank= 2 newrank= 2 recvbuf= 10
rank= 3 newrank= 3 recvbuf= 10
rank= 4 newrank= 4 recvbuf= 10
rank= 5 newrank= 0 recvbuf= 35
rank= 6 newrank= 1 recvbuf= 35
rank= 7 newrank= 2 recvbuf= 35
rank= 8 newrank= 3 recvbuf= 35
rank= 9 newrank= 4 recvbuf= 35
```