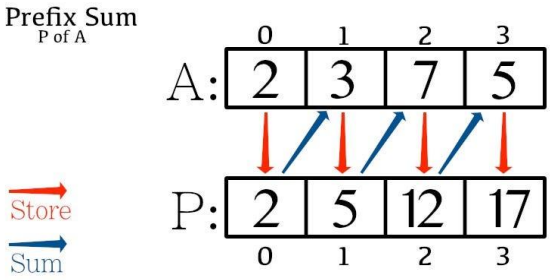
Lab#9: Collective Communication and Computation Operations

Prefix:

Prefix sum is the cumulative sum of all the previous occurred elements with the current element.



MPI_Scan is an inclusive scan: it performs a prefix reduction across all MPI processes in the given communicator. In other words, each MPI process receives the result of the reduction operation on the values passed by that MPI process and all MPI processes with a lower rank. MPI_Scan is a collective operation; it must be called by all MPI processes in the communicator concerned.

int MPI_Scan(const void* send_buffer, void* receive_buffer, int count, MPI_Datatype datatype, MPI_Op operation, MPI_Comm communicator);

Example#1:Prefix Sum

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
 int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);

   // Get my rank
   int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);
    // Get the sum of all ranks up to mine and print it
   int total;
   MPI_Scan(&my_rank, &total, 1, MPI_INT, MPI_SUM, MPI_COMM_WORLD);
   printf("[MPI process %d] Total = %d.\n", my_rank, total);
    MPI_Finalize();
    return EXIT_SUCCESS;
}
```

Task#01: Determine the output of Example#01.

Vector Variants of Scatter, Gather, Allgather

MPI provides a vector variant of the scatter operation, called MPI_Scatterv, that allows different amounts of data to be sent to different processes.

int MPI_Scatterv(void *sendbuf, int *sendcounts, int *displs, MPI_Datatype senddatatype, void *recvbuf, int recvcount,MPI_Datatype recvdatatype, int source, MPI_Comm comm)

As we can see, the parameter sendcount has been replaced by the array sendcounts that determines the number of elements to be sent to each process. In particular, the target process sends sendcounts[i] elements to process *i*. Also, the array displs is used to determine where in sendbuf these elements will be sent from. In particular, if sendbuf is of the same type is senddatatype, the data sent to process *i* start at location displs[i] of array sendbuf . Both the sendcounts and displs arrays are of size equal to the number of processes in the communicator. Note that by appropriately setting the displs array we can use MPI_Scatterv to send overlapping regions of sendbuf .

The vector variants of the MPI_Gather and MPI_Allgather operations are provided by the functions MPI_Gatherv and MPI_Allgatherv , respectively.
int MPI_Gatherv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvdatatype, int target, MPI_Comm comm)

int MPI_Allgatherv(void *sendbuf, int sendcount, MPI_Datatype senddatatype, void *recvbuf, int *recvcounts, int *displs, MPI_Datatype recvdatatype, MPI_Comm comm)

These functions allow a different number of data elements to be sent by each process by replacing the recvcount parameter with the array recvcounts. The amount of data sent by process *i* is equal to recvcounts[i] . Note that the size of recvcounts is equal to the size of the communicator comm. The array parameter displs, which is also of the same size, is used to determine where in recvbuf the data sent by each process will be stored. In particular, the data sent by process *i* are stored in recvbuf starting at location displs[i] . Note that, as opposed to the non-vector variants, the sendcount parameter can be different for different processes.

MPI_Alltoall

MPI_Alltoall is a combination of MPI_Scatter and MPI_Gather. That is, every process has a buffer containing elements that will be scattered across all processes, as well as a buffer in which store elements that will be gathered from all other processes. MPI_Alltoall is a collective operation; all processes in the communicator must invoke this routine.

Example#2:

```
#include <stdio.h>
#include <stdlib.h>
#include <mpi.h>
int main(int argc, char* argv[])
{
   MPI_Init(&argc, &argv);

   // Get number of processes and check that 3 processes are used
   int size;
   MPI_Comm_size(MPI_COMM_WORLD, &size);
   if(size != 3)
   {
      printf("This application is meant to be run with 3 MPI processes.\n");
      MPI_Abort(MPI_COMM_WORLD, EXIT_FAILURE);
   }

   // Get my rank
   int my_rank;
   MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

   // Define my value
   int my_values[3];
   for(int i = 0; i < 3; i++)
   {
      my_values[i] = my_rank * 300 + i * 100;
   }
   printf("Process %d, my values = %d, %d, %d.\n", my_rank, my_values[0],
my_values[1], my_values[2]);

   int buffer_recv[3];
   MPI_Alltoall(&my_values, 1, MPI_INT, buffer_recv, 1, MPI_INT,
MPI_COMM_WORLD);
   printf("Values collected on process %d: %d, %d, %d.\n", my_rank, buffer_recv[0],
buffer_recv[1], buffer_recv[2]);

   MPI_Finalize();

   return EXIT_SUCCESS;
}
```

Task#02: Determine the output of Example#02.