Message Passing Interface (MPI)

Exercise 1: Code the implementation of `MPI_Reduce` using synchronous `MPI_send` and `MPI_recv` primitives.

Exercise 2: Code the implementation of `MPI_Broadcast` using synchronous `MPI_send` and `MPI_recv` primitives.

Exercise 3: Complete the following MPI code that computes the minimum value in a vector of natural (unsigned) values.

```
int MPI_Bcast(void* buffer, int count, MPI_Datatype datatype,
              int root, MPI_Comm comm);
int MPI_Scatter(void* sendbuf, int sendcount, MPI_Datatype sendtype,
                void* recvbuf, int recvcount, MPI_Datatype recvtype,
                int root, MPI_Comm comm);
int MPI_Gather(void* sendbuf, int sendcount, MPI_Datatypesendtype,
               void* recvbuf, int recvcount, MPI_Datatyperecvtype,
               int root, MPI_Comm comm );

int main (int argc, char *argv[])
{
  int *V;
  int total_proc;                 // total number of processes
  int rank;                       // rank of each process
  long long int n_per_proc;       // elements per process
  long long int i, n;

  MPI_Status status;

  // Initialization of MPI environment
  MPI_Init (&argc, &argv);
  MPI_Comm_size (MPI_COMM_WORLD, &total_proc);
  MPI_Comm_rank (MPI_COMM_WORLD,&rank);

  unsigned int min[total_proc];


  if (rank == MASTER) {

    V = (int *) malloc(sizeof(unsigned int)*n);

    MPI_Bcast (&n, _____________________________________________________);

    MPI_Scatter(V, _____________________________________________________);

    unsigned int min = 0;
    for(i=0;i<n_per_proc;i++)
      if (min>V[i]) min = V[i];

    MPI_Gather(_________________________________________________________);

  } else { // Non-master tasks

    MPI_Bcast (_________________________________________________________);

    n_per_proc = n/total_proc;

    V = (int *) malloc(sizeof(int)*n_per_proc);
```

```
    MPI_Scatter(________________________________________________________);

    unsigned int min = 0;
    for(i=0;i<n_per_proc;i++)
      if (min>V[i]) min = V[i];

    MPI_Gather(_________________________________________________________);
  }

  MPI_Finalize();

  return 0;
}
```

Exercise 4: Code the implementation of `MPI_Gatherv` using synchronous `MPI_send` and `MPI_recv` primitives.

Exercise 5: Code the implementation of `MPI_Scatterv` using synchronous `MPI_send` and `MPI_recv` primitives.

Exercise 6: Send and receive a number of elements, unknown until communication will happen.

Given the following C++ code, study the execution of the MPI primitives and understand at which points a process will be blocked or not and what is the role of each MPI primitive. Also, check at which points memory allocation happens to support the communication actions. It might be useful to remember the `MPI_status` definition:

```
struct MPI_Struct {
  int MPI_SOURCE;
  int MPI_TAG;
  int MPI_ERROR;
  int _cancelled;
  size_t _ucount;
};
```

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

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

  int size, rank;
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);

  if (rank == 0) {
    // Process 0 is sending a random number (between 10 and 25) of integers to
    // process 1
    int n_items = rand() % 16 + 10; // BAD way of doing random.
    std::cout << "Process 0, random count gives us " << n_items << " ints to send."
              << std::endl;
```

```
    // Allocation and initialisation of the buffer
    int *send_buf = new int[n_items];
    for (int i=0; i < n_items; ++i)
      send_buf[i] = i*i;

    std::cout << "Process 0, sending : ";
    for (int i=0; i < n_items; ++i)
      std::cout << send_buf[i] << " ";
    std::cout << std::endl;

    // Blocking send
    MPI_Send(send_buf, n_items, MPI_INT, 1, 0, MPI_COMM_WORLD);

    // Deallocation
    delete [] send_buf;
  }
  else {
    // Probing the reception of messages
    MPI_Status status;
    MPI_Probe(0, 0, MPI_COMM_WORLD, &status);

    // From the probed status we get the number of elements to receive
    int n_items;
    MPI_Get_count(&status, MPI_INT, &n_items);

    std::cout << "Process 1, probing tells us message will have "
              << n_items << " ints." << std::endl;

    // Allocating and receiving
    int *recv_buf = new int[n_items];

    MPI_Recv(recv_buf, n_items, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    std::cout << "Process 1, buffer received : ";
    for (int i=0; i < n_items; ++i)
      std::cout << recv_buf[i] << " ";
    std::cout << std::endl;

    delete [] recv_buf;
  }
  MPI_Finalize();
  return 0;
}
```

Exercise 7: User-defined types in MPI, the case of vectors.

One very simple form of a custom datatype is the simple repetition of a primitive type of data. For instance, suppose an application that operates over points in a 3d reference frame, then we would like to manipulate a `Point` structure with three `double` in it. We can achieve this very simply using the MPI_Type_contiguous function. Its prototype is :

```
int MPI_Type_contiguous(int count, MPI_Datatype old_type, MPI_Datatype *new_type);
```

So if we want to create a vector datatype, we can easily do :

```
MPI_Datatype dt_point; MPI_Type_contiguous(3, MPI_DOUBLE, &dt_point);
```

We are not entirely done here, we need to **commit** the datatype. The commit operation allows MPI to generate a formal description of the buffers you will be sending and receiving. This is a mandatory operation. If you don't commit but still use your new datatype in communications, you are most likely to end up with invalid datatype errors. You can commit by simply calling `MPI_Type_commit`.

```
MPI_Type_commit(&dt_point);
```

Study and understand the following C++ code where custom datatypes are used.

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

struct Point {
  double x, y, z;
};

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

  int rank, size;
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  MPI_Datatype dt_point;
  MPI_Type_contiguous(3, MPI_DOUBLE, &dt_point);
  MPI_Type_commit(&dt_point);

  constexpr int n_points = 10;
  Point data[n_points];

  // Process 0 sends the data
  if (rank == 0) {
    for (int i=0; i < n_points; ++i) {
      data[i].x = (double)i;
      data[i].y = (double)-i;
      data[i].z = (double) i * i;
```

```
    }

    MPI_Send(data, n_points, dt_point, 1, 0, MPI_COMM_WORLD);
  }
  else { // Process 1 receives
    MPI_Recv(data, n_points, dt_point, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

    // Printing
    for (int i=0; i < n_points; ++i) {
      std::cout << "Point #" << i << " : (" << data[i].x << "; "
                << data[i].y << "; " << data[i].z << ")"
        << std::endl;
    }
  }

  MPI_Finalize();
}
```

Exercise 8: Code the implementation of `MPI_Gather` using synchronous `MPI_send` and `MPI_recv` primitives.

Exercise 9: Code the implementation of `MPI_Scatter` using synchronous `MPI_send` and `MPI_recv` primitives.