CHAPTER 3

DISTRIBUTED MEMORY PROGRAMMING WITH MPI

Feel free to contact me for more information:

email: miguelsrochajr@gmail.com
LinkedIn: http://linkedin.com/in/miguelrochajr
YouTube: https://www.youtube.com/channel/UCYViw5AtRsUHMyoB_zljaIA

Question 3.1

What happens in the greetings program if, instead of *strlen(greeting)* for the length of the message being sent by the processes 1,2,..., *comm_sz*-1? What happens if we use MAX_STRINGS instead of *strlen(greeeting)+1 ?* Can you explain these results?

The strlen(greeting)+1 is the number of characters in the message plus one characther for the '\0' char. Which, in C, represents the end of a string. If we use only strlen(greeting) we would be sending the String message without the '\0' charachter. This could lead to plenty of problems. The most dangerous of them is segmentation fault. Since we are seding the string "without an end", an implementation that keeps parsing the string up to an '\0' char would never stop and probably attempt to access a memory location that does not exist.

In the case we use MAX_STRING, we have some small differences. Despite our output being the same, the MPI_Send will send all the 100 locations to the receiver. In the case of our greeting string, all of it will be sent and not only the actual text. Since we are also sending the \0, the output will be the same since printf looks for it to end the string output.

Question 3.2

Modify the trapezoidal rule so that it will correctly estimate the integral even if *comm_sz* doesn't evenly divide n. (You can assume that n>= *comm_sz*)

Answer:

First let's analyse what happens when we have comm_sz=4 and n=15 assuming both are integers.

$$quotient = n/comm\_sz = 3$$

If we distribute 3 trapezoids through each of the 4 processes, the total of trapezoids would be 12 and 3 will be missing. So, to fix that, I am going to assign one more trapezoid for those which the rank is less than *comm_sz*. It is important also to note that the *local_a* and *local_b* are going to suffer a little modification also. See the listing 1.

```
quotient = n/comm_sz;
if (my_rank < comm_sz){
        local_n = quotient +1;
        local_a = a + my_rank*local_n*h;
        local_b = local_a + local_n*h;
} else { … }
```

Listing 1

Furthermore, not all ranks are going to fall into the if statement. The last ones will have a little different approach. First, the amount of trapezoids they get is equal to the quotient. Second, the calculation of their *local_a* and *local_b* are a little different. The following code shows the approached used for it.

```
{...} else {
        local_n = quotient;
        local_a = a + my_rank*local_n*h + remainder*h;
        local_b = local_a + local_n*h;
}
```

Listing 2

When we use $a + my\_rank * local\_n * h$ we will obtain a result that will fall somewhere into the range of the other processes. This is due to the difference of the local_n. Thus, since we already have assigned an amount of *remainder* processes with *quotient+1*, we must do this "value shift" so that the *local_a* of the process which my_rank >= comm_sz by adding the amount of trapezoids assigned to the previous processes times the size of these trapezoids. The complete code can be found below at listing 3.

```
quotient = n/comm_sz;
remainder = n%commsz;
if (my_rank < comm_sz){
        local_n = quotient +1;
        local_a = a + my_rank*local_n*h;
        local_b = local_a + local_n*h;
} else {
        local_n = quotient;
        local_a = a + my_rank*local_n*h + remainder*h;
        local_b = local_a + local_n*h;
}
```

Listing 3

Question 3.3

Determine which of the variables in the trapezoidal rule program are local and which are global.

The global variables are those which its value is significant to all processes. Local variables are those which their values are significant only to the process that is using them. So, of the trapezoidal rule program (file Question3.c), they are:

Global:

- comm_sz
- n
- a, b and h

Local:

- my_rank
- local_n
- local_int
- local_a, local_b
- total_int
- source

Question 3.4

Modify the program that just prints a line of output from each process (*mpi_output.c*) so that the output is printed in process rank order: process 0s output first, then process 1s, and so on.

To do so, I'll use the same implementation as the mpi_hello.c. Since we have pretty much the same layout for the string printed by the processes, each process will send their string to process 0. Then the process which rank is zero will, first, display its owns string and then, in a for loop, receive from each other process in sequence to print out to the screen, in sequence. The full code is shown on listing 4.

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

const int MAX_STRING = 100;

int main(void) {

  int my_rank, comm_sz;
  char my_question[MAX_STRING];

  MPI_Init(NULL, NULL);
  MPI_Comm_size(MPI_COMM_WORLD, &comm_sz);
  MPI_Comm_rank(MPI_COMM_WORLD, &my_rank);

  if (my_rank != 0) {
     /* Create message */
     sprintf(my_question, "Process %d of %d > Does anyone have a toothpick?\n",
           my_rank, comm_sz);
     /* Send message to process 0 */
     MPI_Send(my_question, strlen(my_question)+1, MPI_CHAR, 0, 0,
           MPI_COMM_WORLD);
  } else {
     /* Print my message */
     printf("Process %d of %d > Does anyone have a toothpick? \n", my_rank, comm_sz);
     for (int q = 1; q < comm_sz; q++) {
        /* Receive message from process q */
        MPI_Recv(my_question, MAX_STRING, MPI_CHAR, q,
           0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        /* Print message from process q */
        printf("%s", my_question);
     }
  }
  MPI_Finalize();
  return 0;
}
```

Listing 4

Question 3.5

In a Binary tree, there is a unique shortest path from each node to the root. The length of the path is often called the depth of the node. A binary tree in which every nonleaf has two children is called a full binary tree, and a fully binary tree in which every leaf has the same depth is sometimes called a complete binary tree. See Figure 3.14. Use the principle of mathematical induction to prove that if T is a complete binary tree with n leaves, then the depth of the leaves is $log_2(n)$.

Answer:

Assuming that the complete binary tree we are referring here is the one which all non-leaves have two children nodes, we start.

The basis case here is a tree T with 1 one. Since this node is the root but also a leaf, we have a depth of zero.

$$\text{Basis: } n = 1,\ depth(1) = log_2 1 = 0$$

Now, T is a complete binary tree with m leaves which $m > 1$. At this moment, let's consider the parents of the leaves of T. If $d = depth(n)$ in which n is a leaf of T, $d - 1$ is the depth of a leaf's parent. Thus, all the parents have the same depth, $d - 1$.

Note that if we remove the laves from T, we would obtain a tree S that is also a complete binary tree as shown on figure 1.

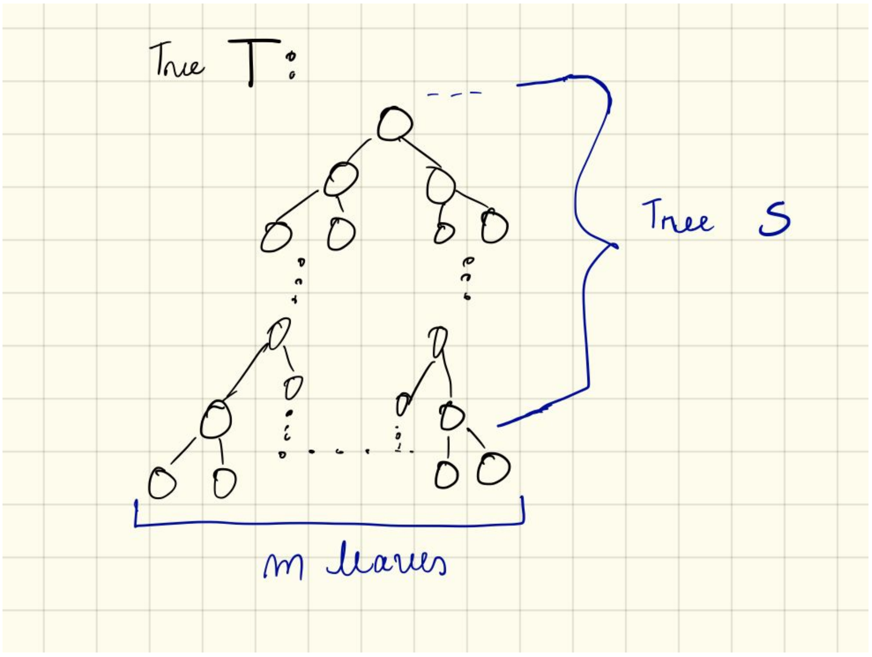


Figure 1: Tree S and tree T

Note also that since S is T without the leaves, S has $n/2$ leaves. So, our induction hypothesis lead us to:

$$I.H.:\ depth(n/2) = log_2(n/2)$$

Then, looking at the tree T, the depth of all its leaves should be the following.

$$log_2(n/2) + 1 = 1 + log_2 n - log_2 2 = log_2 n + 1 - 1 = log_2 n$$

Question 3.6

Suppose $comm\_sz = 4$ and suppose that **x** is a vector with $n = 14$ componentes.

a. How would the components of **x** be distributed among the processes in a program that used a block distribution?
b. How would the components of **x** be distributed among the processes in a program that used a cyclic distribution?
c. How would the components of **x** be distributed among the processes in a program that used a block-cyclic distribution with blocksize $b = 2$?

Answer:

With these assumptions, we have $n/comm\_sz$ values for each process. In this case, $14/4 = 3.5$ values for each.

The most important thing we must keep in mind is the load balance. In order to attempt the best balance, the last $n\%comm\_sz$ processes will receive $(int)(n/comm\_sz)$ values from the vector **x**.

So, in this case, we have the following result.

$$n\%comm\_sz = 14\%3 = 2.$$

The last two will receive $(int)(14/3) = 3\ values$

The rest will receive $ceil(n/comm\_sz) = ceil(3.5) = 4\ values$

Thus, using this method we have the following answers.

a. For a block distribution:

Process 0	x0	x1	x2	x3
Process 1	x4	x5	x6	x7
Process 2	x8	x9	x10	
Process 3	x11	x12	x13	

b. For a cyclic distribution:

Process 0	x0	x4	x8	x12
Process 1	x1	x5	x9	x13
Process 2	x2	x6	x10	
Process 3	x3	x7	x11	

c. For a block-cyclic distribution with blocksize = 2:

Process 0	x0	x1	x8	x9
Process 1	x2	x3	x10	x11
Process 2	x4	x5	x12	
Process 3	x6	x7	x13	

Question 3.7

What do the various MPI collective functions do if the communicator contains a single process?

Answer

The call would result the only core to "send it to itself". Its value will only be copied to the result. For example, in the case of sum of the values in a N dimension vector, we would have the cde as in listing 5:

```
double local_x[N], sum[N];
{...}
MPI_Reduce (loca_x, sum, N, MPI_Double, MPI_SUM, 0, MPI_COMM_WORLD);
```

Listing 5

Since there is no other process to sum its values, we will have sum=local_x, an output just as a copy of the input.

Question 3.8

Suppose $comm\_sz = 8$ and $n = 16$.

a. Draw a diagram that shows how *MPI_Scatter* can be implemented using tree-structured communication with $comm\_sz$ processes when process 0 needs to distribute an array containing n elements.
b. Draw a diagram that shows how *MPI_Scatter* can be implemented using tree-structured communication when an *n*-element array that has been distributed among $comm\_sz$ processes needs to be gathered onto process 0.

Answer:

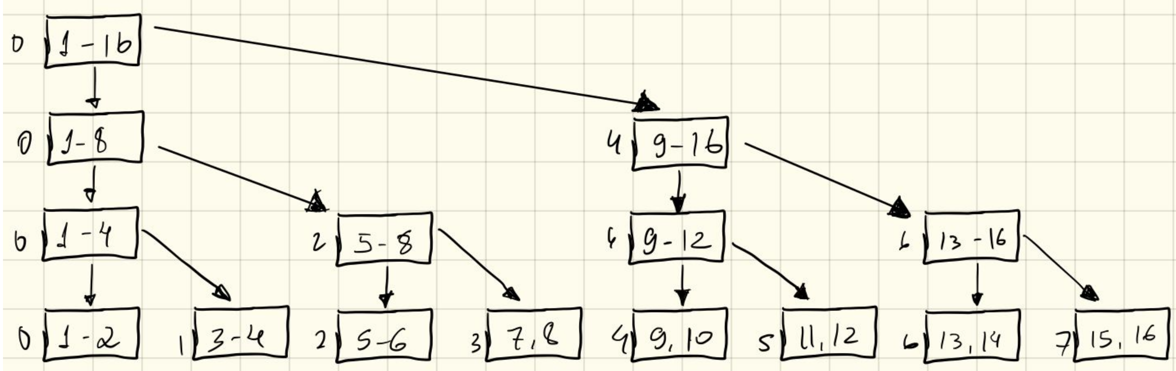
8.) comm_sz = 8 and n = 16

The following convention will be used:

rank $\boxed{1-n}$

Meaning process "$rank" has the values 1 through n.

a)



b)

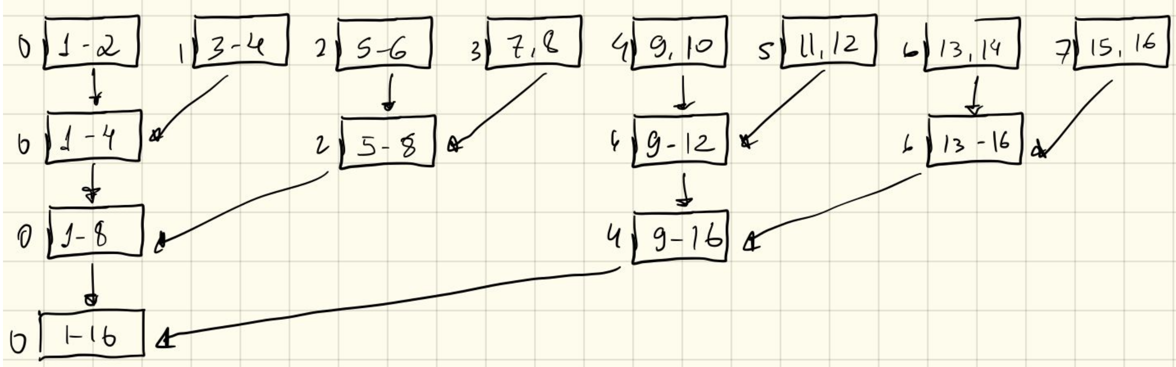


Figure 2: MPI_Scatter behavior

Question 3.9

Write an MPI program that implements multiplication of a vector by a scalar and dot product. The user should enter two vectors and a scalar, all of which are read in by process 0 and distributed among the processes. The results are calculated and collected onto process 0, which prints them. You can assume that n, the order of the vectors, is evenly divisible by $comm\_sz$.

Answer

To begin, let's first devise this problem in parts. The first part is when the user is required to enter two vectors and one scalar. For that, a function called Read_n was created to receive the order of the vector in the process 0 and broadcast it to all the other processes.

```
void Read_n(int* n_p, int* local_n_p, int my_rank, int comm_sz,
      MPI_Comm comm) {
   int local_ok = 1;

   if (my_rank == 0){
      printf("Please type the vector order. \n");
      scanf("%d", n_p);
   }
   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
   *local_n_p = *n_p / comm_sz;
}
```

Listing 6: Read_n function: read the vector order and broadcast it.

Note here that since we only have one value, the call for a MPI_Scatter is not needed. The MPI_Bcast function was called so that it would send a value *n_p*, which has the type MPI_INT, coming from process 0 that is contained in the communicator *comm*. After this call, all processes know that the vector has *n_p* positions.

The next part has to do with the initialization of the vectors and the scalar. In the main function, the program must, before anything, allocate the memory in which our pointers are going to point to.

```
{...}
   local_vec1 = malloc(local_n*sizeof(double));
   local_vec2 = malloc(local_n*sizeof(double));
   local_scalar_mult1 = malloc(local_n*sizeof(double)); //result of local_vec1 mult
   local_scalar_mult2 = malloc(local_n*sizeof(double)); //result of local_vec2 mult
{...}
```

Listing 7: Memory allocations for vectors

Note the following here: in each process, only the needed memory for the amount of memory is allocated. The reason is simple: we do not need to have a copy of all the vector on each core, we only need the ones that the particular core will really compute over it. And, of course, to take all advantage of this approach, the MPI_Scatter function will be used.

Now, the next step is to receive the vectors and the scalar in the process 0 and send the needed values to all the other processes. It is important to notice that for the broadcasting of the scalar, only the MPI_Bcast was used. However, for the vector, the MPI_Scatter was used. The purpose of that is to minimize the memory waste.

```
void Read_data(double local_vec1[], double local_vec2[], double* scalar_p,
      int local_n, int my_rank, int comm_sz, MPI_Comm comm) {
   double* aux = NULL; // auxiliar variable used to store the input values
   int i;
   if (my_rank == 0){
      printf("What is the scalar?\n");
      scanf("%lf", scalar_p);
   }

   MPI_Bcast(scalar_p, 1, MPI_DOUBLE, 0, comm);

   if (my_rank == 0){
      aux = malloc(local_n * comm_sz * sizeof(double));
      printf("Enter the first vector\n");
      for (i = 0; i < local_n * comm_sz; i++)
         scanf("%lf", &aux[i]);
      MPI_Scatter(aux, local_n, MPI_DOUBLE, local_vec1, local_n, MPI_DOUBLE, 0, comm);
      printf("Enter the second vector\n");
      for (i = 0; i < local_n * comm_sz; i++)
         scanf("%lf", &aux[i]);
      MPI_Scatter(aux, local_n, MPI_DOUBLE, local_vec2, /* Sender part. This part is
most important to the sender */
        local_n, MPI_DOUBLE, 0, comm);
      free(aux);
   } else {
      MPI_Scatter(aux, local_n, MPI_DOUBLE, /* Send part. Is important for the sender
but not to the receiver, which is the case here */
         local_vec1, local_n, MPI_DOUBLE, 0, comm); /* Receive part. This is what
really matters here. */
      MPI_Scatter(aux, local_n, MPI_DOUBLE, local_vec2, local_n,
            MPI_DOUBLE, 0, comm);
   }
}
```

Listing 8: Read_data function receives the values from the user

At this point, all the processes have local_vecl, local_vec2 and the scalar that was transmitted by the process 0. Let's now pay attention to what is happening at the MPI_Scatter.

When scattering the values, MPI_Scatter only sends it the values needed, as shown in Figure 2. For process 0, which is the one that is the source process, the value pointed by *aux* (the vector) is the *send_buf_p* (te value to be transmitted), which has *local_n* values and has the type MPI_Double. The chunks of *aux will be stored in *local_vec1* and *local_vec1*, after the respective reading from the user. Note also that since we are assuming that n is evenly divisible by $comm\_sz$, local_n (the order of *local_vec1* and *local_vec1*) has the same order for all processes. Thus, both output vectors have values of type MPI_Double coming from process 0 which is in the communicator *comm*.

For those processes that are not the process 0, should also have their calls to MPI_Scatter. The arguments here are the same as for the process 0 but their behavior is a little different: they will not compute the original aux value, only its chunks.

Now, each process must perform the dot product and assign the result value to a variable called dot_product. The function has the following definition.

```
double Perform_dot_product(double local_vec1[], double local_vec2[], int local_n,
MPI_Comm comm)
{
   int local_i;
   double dot_product, local_dot_product = 0;

   for (local_i = 0; local_i < local_n; local_i++)
      local_dot_product += local_vec1[local_i] * local_vec2[local_i];

   MPI_Reduce(&local_dot_product, &dot_product, 1, MPI_DOUBLE, MPI_SUM,
         0, comm);
   return dot_product;
}
```

Listing 9: Perfom_dot_product

Each process will take its local_vec1 and 2 and perform the values. And in order to gather the values, a call to MPI_Reduce is made. First, let's understand how the dot product is computed mathematically. The equation below show what is happening on the Listing 9.

$$dot\_product = local\_vec1 \cdot local\_vec2 \; = \sum_{i=0}^{n-1} local\_vec1_i * local\_vec2_i \quad \text{Eq.1}$$

Note here that the dot product is a sum, which can be perfoming by MPI_Reduce using the global sum. So, MPI_Reduce will use the local_dot_product of each process, store the result in dot_product which is only one value that has the type MPI_Double, though a MPI_SUM using process 0 as the source process that is contained as part of the comm communicator. Thus, the final result is stored into the variable dot_product.

And, to conclude the algorithm, the scalar product is performed. The function that implements this step is shown on the Listing 10 below.

```
void Vec_scalar_mult(double local_vec[], double scalar,
      double local_result[], int local_n) {
```

```
   int local_i;
   for (local_i = 0; local_i < local_n; local_i++)
      local_result[local_i] = local_vec[local_i] * scalar;
}
```

Listing 10: Scalar multiplication

The listing 10 shows the function that multiplies the scalar by each of the values of the vector. Note here that the processes are still with their local variables and the complete vector was not gathered to be displayed to the user. This step is performed when the final result is displayed to the user at the Print_vector function.

```
void Print_vector(double local_vec[], int local_n, int n, char title[],
      int my_rank, MPI_Comm comm) {
   double* aux = NULL;
   int i;

   if (my_rank == 0) {
      aux = malloc(n * sizeof(double));
      MPI_Gather(local_vec, local_n, MPI_DOUBLE, aux,
           local_n, MPI_DOUBLE, 0, comm); //note that local_n is the number of data
items received from each process, not the total number of data items selected.
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%.2f ", aux[i]);
      printf("\n");
      free(aux);
   } else {
      MPI_Gather(local_vec, local_n, MPI_DOUBLE,
        aux, local_n, MPI_DOUBLE, 0, comm); //the others will receive the values and
store in aux, with the amount of local_n values
   }
}
```

Listing 11: Printing final result and gathering chunks from processes.

This function collect all of the components of the distributed vector onto process 0, and then process 0 can print all of the components. This communication in the function is carried out by MPI_Gather. For process zero, the local_vector, which has local_n values and type MPI_Double will have its chunks gathered onto the auxiliar variable aux and print the vector. For those the are not process zero, they also must have a call to MPI_Gather so that they can send their chunks of the main vector to the process zero.

The complete code is shown on Listing 12.

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

void Read_n(int* n_p, int* local_n_p, int my_rank, int comm_sz,
      MPI_Comm comm);
void Read_data(double local_vec1[], double local_vec2[], double* scalar_p,
      int local_n, int my_rank, int comm_sz, MPI_Comm comm);
```

```
void Print_vector(double local_vec[], int local_n, int n, char title[],
      int my_rank, MPI_Comm comm);
double Perform_dot_product(double local_vec1[], double local_vec2[],
      int local_n, MPI_Comm comm);
void Par_vector_scalar_mult(double local_vec[], double scalar,
      double local_result[], int local_n);

int main(void) {
   int n, local_n;
   double *local_vec1, *local_vec2;
   double scalar;
   double *local_scalar_mult1, *local_scalar_mult2;
   double dot_product;
   int comm_sz, my_rank;
   MPI_Comm comm;

   MPI_Init(NULL, NULL);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &comm_sz);
   MPI_Comm_rank(comm, &my_rank);

   Read_n(&n, &local_n, my_rank, comm_sz, comm);
   /*
    para
    comm_sz = 4
    n = 16
    local_n = 16/4 = 4
   */

   local_vec1 = malloc(local_n*sizeof(double));
   local_vec2 = malloc(local_n*sizeof(double));
   local_scalar_mult1 = malloc(local_n*sizeof(double));
   local_scalar_mult2 = malloc(local_n*sizeof(double));

   Read_data(local_vec1, local_vec2, &scalar, local_n, my_rank, comm_sz, comm); // Here
everybdy has everything

   dot_product = Perform_dot_product(local_vec1, local_vec2, local_n, comm);
   if (my_rank == 0) {
      printf("Dot product is %f\n", dot_product);
   }

   Par_vector_scalar_mult(local_vec1, scalar, local_scalar_mult1, local_n);
   Par_vector_scalar_mult(local_vec2, scalar, local_scalar_mult2, local_n);

   Print_vector(local_scalar_mult1, local_n, n,
         "The product of the first vector with scalar is",
         my_rank, comm);
   Print_vector(local_scalar_mult2, local_n, n,
         "The product of the second vector with scalar is",
         my_rank, comm);

   free(local_scalar_mult2);
   free(local_scalar_mult1);
   free(local_vec2);
   free(local_vec1);

   MPI_Finalize();
   return 0;
}

void Read_n(int* n_p, int* local_n_p, int my_rank, int comm_sz,
      MPI_Comm comm) {
   int local_ok = 1;
   if (my_rank == 0){
      printf("Please type the vector order. \n");
      scanf("%d", n_p);
   }
   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);
   *local_n_p = *n_p / comm_sz;
}

void Read_data(double local_vec1[], double local_vec2[], double* scalar_p,
      int local_n, int my_rank, int comm_sz, MPI_Comm comm) {
   double* aux = NULL;
   int i;
   if (my_rank == 0){
      printf("What is the scalar?\n");
      scanf("%lf", scalar_p);
   }

   MPI_Bcast(scalar_p, 1, MPI_DOUBLE, 0, comm);

   if (my_rank == 0){
      aux = malloc(local_n * comm_sz * sizeof(double));
      printf("Enter the first vector\n");
      for (i = 0; i < local_n * comm_sz; i++)
         scanf("%lf", &aux[i]);
      MPI_Scatter(aux, local_n, MPI_DOUBLE, local_vec1, local_n,
            MPI_DOUBLE, 0, comm);
      printf("Enter the second vector\n");
      for (i = 0; i < local_n * comm_sz; i++)
         scanf("%lf", &aux[i]);
      MPI_Scatter(aux, local_n, MPI_DOUBLE, local_vec2, /* Sender part. This part is
most important to the sender */
        local_n, MPI_DOUBLE, 0, comm);
      free(aux);
   } else {
      MPI_Scatter(aux, local_n, MPI_DOUBLE, /* Send part. Is important for the sender
but not to the receiver, which is the case here */
          local_vec1, local_n, MPI_DOUBLE, 0, comm); /* Receive part. This is what
really matters here. */
      MPI_Scatter(aux, local_n, MPI_DOUBLE, local_vec2, local_n,
            MPI_DOUBLE, 0, comm);
   }
}

void Print_vector(double local_vec[], int local_n, int n, char title[],
      int my_rank, MPI_Comm comm) {
   double* a = NULL;
   int i;

   if (my_rank == 0) {
      a = malloc(n * sizeof(double));
      MPI_Gather(local_vec, local_n, MPI_DOUBLE, a, local_n,
            MPI_DOUBLE, 0, comm);
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%.2f ", a[i]);
      printf("\n");
      free(a);
   } else {
      MPI_Gather(local_vec, local_n, MPI_DOUBLE,
        a, local_n, MPI_DOUBLE, 0, comm); //the others will receive the values and store
in a, with the amount of local_n values
   }
}
```

```
double Perform_dot_product(double local_vec1[], double local_vec2[],
      int local_n, MPI_Comm comm) {
   int local_i;
   double dot_product, local_dot_product = 0;

   for (local_i = 0; local_i < local_n; local_i++)
      local_dot_product += local_vec1[local_i] * local_vec2[local_i];

   MPI_Reduce(&local_dot_product, &dot_product, 1, MPI_DOUBLE, MPI_SUM,
         0, comm);
   return dot_product;
}


void Vec_scalar_mult(double local_vec[], double scalar,
      double local_result[], int local_n) {
   int local_i;
   for (local_i = 0; local_i < local_n; local_i++)
      local_result[local_i] = local_vec[local_i] * scalar;
}
```

Listing 12: Complete algorithm

Question 3.10

In the Read_vector function shown in Program 3.9, we use local_n as the actual argument for of the formal arguments to MPI_Scatter: send_count and recv_count. Why is it OK to alias these arguments?

Answer:

Assuming two arguments are aliased if they refer to the same block of memory, there is no problem with the call to MPI_Scatter in Program 3.9. The reason for that is that both send_count and recv_count are input arguments to the function. The rule that prohibits aliasing in MPI only takes place when the arguments used are output arguments.

Furthermore, the use of aliasing on output arguments makes it, for example, a block of memory be used to both read and write, which could cause really unpredictable behavior.

Question 3.12

Question 3.13

MPI_Scatter and MPI_Gather have the limitation that each process must send or receive the same number of data items. When this is not the caser, we must use the MPI functions MPI_Gatherv and MPI_Scatterv. Look at the man pages for these functions, and modify your vector sum, dot product program so that it can correctly handle the case when *n* isn't evenly divisible by comm_sz.

Answer:

First, let's take a look at the signature for the MPI_Gatherv.

```
int MPI_Gatherv(const void *sendbuf, /* input */
             int sendcount,          /* input */
             MPI_Datatype sendtype, /* input */
             void *recvbuf,          /* output */
             const int recvcounts[], /* input */
             const int displs[],     /* input */
             MPI_Datatype recvtype, /* input */
             int root,              /* input */
             MPI_Comm comm)         /* input */
```

The main difference from the MPI_Gather here are the displs[] and recvcounts[]. The displs[] is a displacement integer array. The indexes of it specifies the displacement relative to recvbuf at which to place the incoming data from process i (signifcant only at the root, in this case the process 0). Also, the recvcounts[] is replacing the recv_count. The new argument is an integer array containing the number of elements that are received from each process (significant only at process 0).

Now, let's take a look at the signature of the MPI_Scatterv.

```
int MPI_Scatterv(const void *sendbuf,       /* input */
               const int sendcounts[],      /* input */
               const int displs[],          /* input */
               MPI_Datatype sendtype,       /* input */
               void *recvbuf,               /* output */
               int recvcount,               /* input */
               MPI_Datatype recvtype,       /* input */
               int root,                    /* input */
               MPI_Comm comm)               /* input */
```

Since now we now the main difference between the old and new ones, let's dig into the code. The first thing we must take in consideration is our old Read_n function. Now we must consider that ranks which are greater than a remainder value will have one less element to process. So, the new Read_n function, which is the one that reads the vectors order, is shown on Listing 13.

```
void Read_n(
         int*      n_p        /* out */,
         int*      local_n_p  /* out */) {
   int quotient, remainder;

   if (my_rank == 0) {
      printf("What's the order of the vectors?\n");
      scanf("%d", n_p);
   }
   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

   quotient = *n_p/comm_sz;
   remainder = *n_p % comm_sz;

   if (my_rank < remainder) {
      *local_n_p = quotient +1;
   } else {
      *local_n_p = quotient;
   }
}
```

Listing 13: New Read_n function.

Since now some processes will handle different amount of data, instead of do what was done to the previous implementation of this algorithm, the pointers for local vectors will have bit different amounts of memory. Each amount is dependent on the local_n for each process.

```
   local_x = malloc(local_n*sizeof(double));
   local_y = malloc(local_n*sizeof(double));
   local_z = malloc(local_n*sizeof(double));
```

Listing 14: Allocate vectors

Now starts the reading of the vectors. Here we have the first big change in comparison to the previous implementation: the usage of the displacement vector and the counts vector. So, first process 0 must allocate the amount of memory for both arrays. Since the displacement and the counts are, at maximum, the number of processes (*comm_sz*), we have the first part of the Read_vect function as below.

```
void Read_vect(
             double    local_a[]   /* out */,
             int       local_n     /* in  */,
             int       n           /* in  */,
             char      vec_name[]  /* in  */) {

   int* counts = NULL;
   int* displs = NULL;
   double* a = NULL;
   int i;
   if (my_rank == 0) {
      counts = malloc(comm_sz*sizeof(int));
      displs = malloc(comm_sz*sizeof(int));
{...}
```

Listing 15: Start of Read_vect

Then, it is time to handle it the counts and displacement vectors. Here, *q* represents a process with rank q. The following function will be called right after the {...} on listing 15.

```
void Init_counts_displs(int counts[], int displs[], int n) {
   int offset, q, quotient, remainder;
   // Remmember from man page:
   //  displs:    Integer array (of length group size). Entry i specifies the displacement relative to 
recvbuf
   //          at which to place the incoming data from process i (significant only at root).
   // counts:     Integer array (of length group size) containing the number of elements that are received 
from
   //          each process (significant only at root).

   quotient = n/comm_sz;
   remainder = n % comm_sz;
   offset = 0;
   for (q = 0; q < comm_sz; q++) {
      if (q < remainder)
         counts[q] = quotient+1;
      else
         counts[q] = quotient;
      displs[q] = offset;
      offset += counts[q];
   }
}
```

Listing 16: Initialization of count and displs

To be more specific about the initialization of the displacement vector and the counts, let's use some graphical representation. So, assuming that $n = 14$ and $comm\_sz = 4$, we would have counts[4] and displs[4]. So, at the end of the Init_counts_displs function, the results would be:

quotient = 14/4 = 3
remainder = n%comm_sz = 14/4 = 3

counts:

4	4	4	3

dspls:

0	4	8	12

Thus, with this information we can now pass on to the rest of the Read_n function. The Listing 17 shows the whole function.

```
void Read_vect(
             double    local_a[]   /* out */,
             int       local_n     /* in  */,
             int       n           /* in  */,
             char      vec_name[]  /* in  */) {

   int* counts = NULL;
   int* displs = NULL;

   double* a = NULL; //auxiliar vector to receive from the user.
   int i;

   if (my_rank == 0) {
      counts = malloc(comm_sz*sizeof(int));
      displs = malloc(comm_sz*sizeof(int));

      Init_counts_displs(counts, displs, n);

      a = malloc(n*sizeof(double)); //auxiliar vector

      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]); //receives vector from user and put into auxiliar vector
      MPI_Scatterv(a,     //sending auxiliar buffer
              counts,     //vector with # of elements in each process.
Process_rank=index
              displs,     //displacement vector. The process my_rank has displs[my_rank]
of dispacement
              MPI_DOUBLE, //The values are of type MPI_DOUBLE
              local_a,    //The receiving buffer
              local_n,    //Address of the receiving buffer. Here will be local_x or
local_y
              MPI_DOUBLE, //type of the receiving buffer
              0,          //source process
              comm);      // communicator
      free(a);
      free(displs);
      free(counts);
   } else {
      MPI_Scatterv(a, counts, displs, MPI_DOUBLE,
            local_a, local_n, MPI_DOUBLE, 0, comm);
   }
}
```

Listing 17: Read_vect function for vector of order n

Now, let's start the usage of MPI_Gatherv. Right after receiving the vectors from the user, let's print the vector onto the screen.

```
void Print_vect(
              double    local_b[]  /* in */,
              int       local_n    /* in */,
              int       n          /* in */,
              char      title[]    /* in */) {

   int* counts = NULL;
   int* displs = NULL;

   double* b = NULL;
   double* aux = NULL;
   int i;

   if (my_rank == 0) {
      counts = malloc(comm_sz*sizeof(int));
      displs = malloc(comm_sz*sizeof(int));
      Init_counts_displs(counts, displs, n);

      b = malloc(n*sizeof(double));

      MPI_Gatherv(local_b,  // Starting address of sednding buffer
            local_n,        // Number of elements in send buffer
            MPI_DOUBLE,     // Type of sending buffer
            b,              // receiving buffer. This will have all vector elements
            counts,         // receving counts array.
            displs,         // the displacement vector for each process
            MPI_DOUBLE,     // type fo receiving value
            0,              // source process
            comm);          // communicator
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(displs);
      free(counts);
      free(b);
   } else {
      MPI_Gatherv(local_b, local_n, MPI_DOUBLE, b, counts, displs,
            MPI_DOUBLE, 0, comm);
   }
}
```

Listing 18: Print vector

Ok, so now that we understand how to use the MPI_Gatherv and the the MPI_Scatterv using the displs and counts arrays, let's perform the dot product and the vector sum. First, the listing 19 shows the implementation of the vector sum.

```
void Par_vect_sum(
                  double  local_x[]  /* in  */,
                  double  local_y[]  /* in  */,
                  double  local_z[]  /* out */,
                  int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}
```

Listing 19: Vector sum function

Note here that we have not used the MPI_Gatherv yet. The reason is simple: we are just summing up the chunks of each process. We will gather right after the call for Par_vect_sum using the Print_vec function shown on Listing 18. The Listing

20 shows the implementation of the dot product. The approach here is the same as used in Eq.1 on Question 3.9.

```
double Par_dot_prod(double local_vec1[], double local_vec2[],
      int local_n) {

   int local_i;
   double loc_dot_prod = 0, dot_prod;

   for (local_i = 0; local_i < local_n; local_i++)
      loc_dot_prod += local_vec1[local_i] * local_vec2[local_i];

   MPI_Reduce(&loc_dot_prod, &dot_prod, 1, MPI_DOUBLE, MPI_SUM, 0, comm);

   return dot_prod;
}
```

Listing 20: dot_product perfmorming

The whole algorithm is shown on listing 21 below and on my GitHub.

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

int my_rank, comm_sz;
MPI_Comm comm;

void Read_n(int* n_p, int* local_n_p);
void Init_counts_displs(int counts[], int displs[], int n);
void Read_vect(double local_a[], int local_n, int n, char vec_name[]);
void Print_vect(double local_b[], int local_n, int n, char title[]);
void Par_vect_sum(double local_x[], double local_y[],
                  double local_z[], int local_n);
double Par_dot_prod(double local_vec1[], double local_vec2[], int local_n);

int main(void) {
   int n, local_n;
   double *local_x, *local_y, *local_z;
   double dot_product;

   MPI_Init(NULL, NULL);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &comm_sz);
   MPI_Comm_rank(comm, &my_rank);

   Read_n(&n, &local_n);

//   Allocate_vects(&local_x, &local_y, &local_z, local_n);
   local_x = malloc(local_n*sizeof(double));
   local_y = malloc(local_n*sizeof(double));
   local_z = malloc(local_n*sizeof(double));


   Read_vect(local_x, local_n, n, "x");
   Print_vect(local_x, local_n, n, "x is");
   Read_vect(local_y, local_n, n, "y");
   Print_vect(local_y, local_n, n, "y is");
   Par_vect_sum(local_x, local_y, local_z, local_n);
   Print_vect(local_z, local_n, n, "The sum is");

   dot_product = Par_dot_prod(local_x, local_y, local_n);
   if (my_rank == 0)
      printf("Dot product input vectors is %f\n", dot_product);

   free(local_x);
   free(local_y);
   free(local_z);

   MPI_Finalize();
   return 0;
}

void Read_n(
         int*      n_p        /* out */,
         int*      local_n_p  /* out */) {
   int quotient, remainder;

   if (my_rank == 0) {
      printf("What's the order of the vectors?\n");
      scanf("%d", n_p);
   }
   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

   quotient = *n_p/comm_sz;
   remainder = *n_p % comm_sz;

   if (my_rank < remainder) {
      *local_n_p = quotient +1;
   } else {
      *local_n_p = quotient;
   }
}

void Init_counts_displs(int counts[], int displs[], int n) {
   int offset, q, quotient, remainder;
   // Remmember from man page:
   //  displs:    Integer array (of length group size). Entry i specifies the
displacement relative to  recvbuf
   //           at which to place the incoming data from process i (significant only at
root).
   // counts:     Integer array (of length group size) containing the number of elements
that are received from
   //           each process (significant only at root).

   quotient = n/comm_sz;
   remainder = n % comm_sz;
   offset = 0;
   for (q = 0; q < comm_sz; q++) {
      if (q < remainder)
         counts[q] = quotient+1;
      else
         counts[q] = quotient;
      displs[q] = offset;
      offset += counts[q];
   }
}

void Read_vect(
             double    local_a[]   /* out */,
             int       local_n     /* in  */,
             int       n           /* in  */,
              char      vec_name[]  /* in  */) {

   int* counts = NULL;
   int* displs = NULL;

   double* a = NULL; //auxiliar vector to receive from the user.
   int i;

   if (my_rank == 0) {
      counts = malloc(comm_sz*sizeof(int));
      displs = malloc(comm_sz*sizeof(int));

      Init_counts_displs(counts, displs, n);

      a = malloc(n*sizeof(double)); //auxiliar vector

      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]); //receives vector from user and put into auxiliar vector
      MPI_Scatterv(a,     //sending auxiliar buffer
              counts,     //vector with # of elements in each process.
Process_rank=index
              displs,     //displacement vector. The process my_rank has displs[my_rank]
of dispacement
              MPI_DOUBLE, //The values are of type MPI_DOUBLE
              local_a,    //The receiving buffer
              local_n,    //Address of the receiving buffer. Here will be local_x or
local_y
              MPI_DOUBLE, //type of the receiving buffer
              0,          //source process
              comm);      // communicator
      free(a);
      free(displs);
      free(counts);
   } else {
      MPI_Scatterv(a, counts, displs, MPI_DOUBLE,
            local_a, local_n, MPI_DOUBLE, 0, comm);
   }
}

void Print_vect(
              double    local_b[]  /* in */,
              int       local_n    /* in */,
              int       n          /* in */,
              char      title[]    /* in */) {

   int* counts = NULL;
   int* displs = NULL;

   double* b = NULL;
   double* aux = NULL;
   int i;

   if (my_rank == 0) {
      counts = malloc(comm_sz*sizeof(int));
      displs = malloc(comm_sz*sizeof(int));
      Init_counts_displs(counts, displs, n);

      b = malloc(n*sizeof(double));

      MPI_Gatherv(local_b,  // Starting address of sednding buffer
            local_n,        // Number of elements in send buffer
            MPI_DOUBLE,     // Type of sending buffer
            b,              // receiving buffer. This will have all vector elements
            counts,         // receving counts array.
            displs,         // the displacement vector for each process
            MPI_DOUBLE,     // type fo receiving value
            0,              // source process
            comm);          // communicator
      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(displs);
      free(counts);
      free(b);
   } else {
      MPI_Gatherv(local_b, local_n, MPI_DOUBLE, b, counts, displs,
            MPI_DOUBLE, 0, comm);
   }
}

void Par_vect_sum(
                   double  local_x[]  /* in  */,
                   double  local_y[]  /* in  */,
                   double  local_z[]  /* out */,
                   int     local_n    /* in  */) {
   int local_i;

   for (local_i = 0; local_i < local_n; local_i++)
      local_z[local_i] = local_x[local_i] + local_y[local_i];
}

double Par_dot_prod(double local_vec1[], double local_vec2[],
      int local_n) {

   int local_i;
   double loc_dot_prod = 0, dot_prod;

   for (local_i = 0; local_i < local_n; local_i++)
      loc_dot_prod += local_vec1[local_i] * local_vec2[local_i];

   MPI_Reduce(&loc_dot_prod, &dot_prod, 1, MPI_DOUBLE, MPI_SUM, 0, comm);

   return dot_prod;
}
```

Question 3.14

a.   Write a serial C program that defines a two-dimensional array in the main function. Just use numeric constants for the dimensions:

```
Int two_d[3][4];
```

Initialize the array in the main function. After the array is initialized, call a function that attempts to print the array. The prototype for the function should look something like this.

```
void Print_two_d(int two_d[][], int rows, int cols);
```

After writing the function try to compile the program. Can you explain why won't compile?

b.   After consulting C reference, modify the program so that it will compile and run, but so that it still uses a two dimensional C array.

Answer:

a.

The reason for not compiling is the way C handles multidimensional arrays. The block of memory is a continuous block that has the size of rows*col*sizeof(type). So, because of that, the C compiler needs to know at least the number of columns so that it can parse the memory array and after *col* values find the next line.

b.

The modification needed to fix that was to change the function signature so that the matrix two_d has the amount of columns pre-defined. The code is shown on Listing 13.

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

void Print_two_d(int two[][4], int rows, int cols);

int main(void) {
   int i,j;
   int two_d[3][4];
   int temp = 0;
   for (i = 0; i < 3; i++)
      for(j =0; j<4; j++) {
         two_d[i][j] = temp;
         temp++;
      }
   Print_two_d(two_d, 3, 4);
   return 0;
}  /* main */
void Print_two_d(int two[][4], int rows, int cols) {
   int i,j;
   for (i = 0; i < rows ;i++){
      for(j =0; j < cols;j++) {
         printf("%d ", two[i][j]);
      }
      printf("\n");
   }
}
```

Listing 13

Question 3.15

What is the relationship between the "row-major" storage for two-dimensional arrays that we discussed in Section 2.2.3 and the one-dimensional storage we use in Section 3.4.9?

Answer:

There is no difference. Both store the elements in the memory using the same approach. The localization concept is the same for both. The first row comes first, the second comes right after and so on. For both cases, a A[2][2] matrix that is stored in the address 0x00 and each address represents an element would have the following representation in memory.

Address: 0x00 A[0][0]	Address: 0x01 A[0][1]	Address: 0x02 A[1][0]	Address: 0x03 A[1][1]

Question 3.16

Suppose $comm\_sz = 8$ and the vector $x = (0, 1, 2, ..., 15)$ has been distributed among the processes using a block distribution. Draw a diagram illustrating the steps in a butterfly implementation of Allgather of **x**.

Answer:

Allgather will perform a butterfly structured communication. This approach uses both Broadcasting and gathering. The Figure below demonstrates it the steps for it.

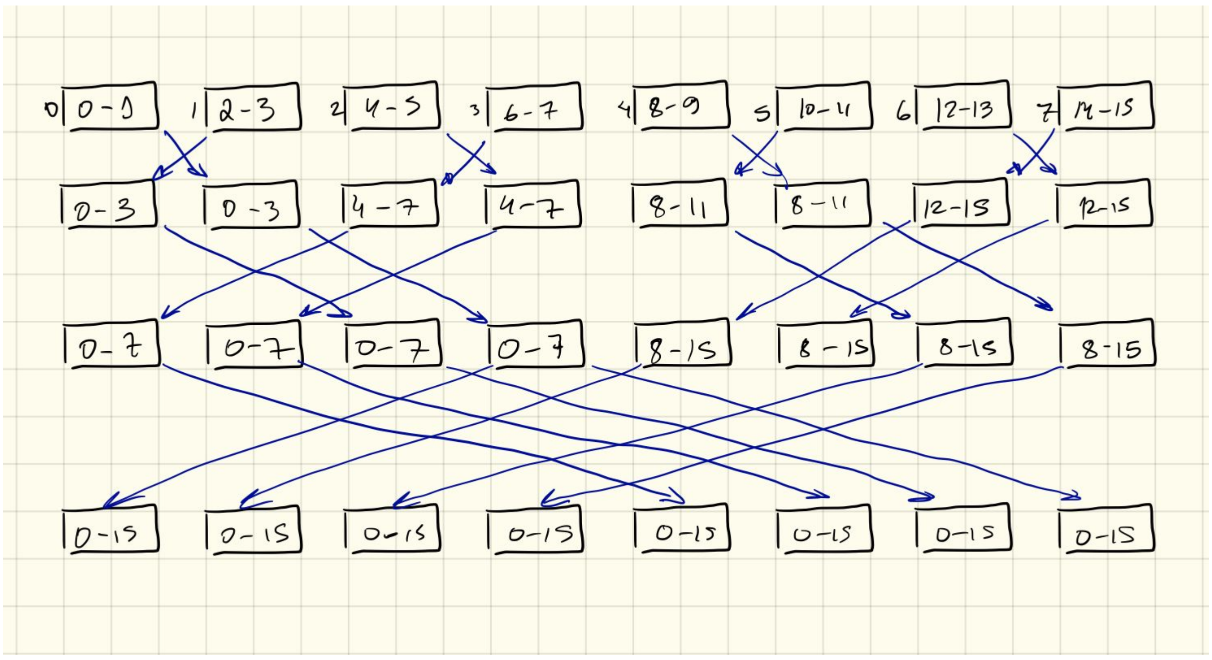


Figure: Butterfly steps for Allgather

Question 3.17

MPI_Type_contiguous can be used to build a derived datatype from a collection of contiguous elements in an array. Its syntax is:

```
int MPI_Type_contiguous(
        int      count,                    // int
        MPI_Datatype old_mpi_t,            // in
        MPI_Datatype* new_mpi_t_p);        // out
```

Modify the Read_vector and Print_vector functions so that they use an MPI datatype created by a call to MPI_Type_contiguous and a *count* argument of 1 in the calls to `MPI_Scatter` and `MPI_Gather`.

Answer:

The main difference here is that now we are not going to send multiple values of a vector anymore, we are going to send only one new structure, a new MPI type that we are calling here *cont_mpi_t*. This type will have *count* number of elements. For a comm_sz = 4 and n = 12, we have the following result:

$$local\_n = n/comm\_sz = 3$$

```
MPI_Type_contiguous( local_n, MPI_DOUBLE, cont_mpi_t);
```

{(double, 0) , (double, 8), (double, 16), (double, 24)}

This new type and format will be used instead of the regular MPI_DOUBLE. The listing below shows the Read_vector function.

```
void Read_vector(
             double        local_a[]   /* out */,
             int           local_n     /* in  */,
             int           n           /* in  */,
             char          vec_name[]  /* in  */,
             MPI_Datatype  cont_mpi_t  /* in  */) {

   double* a = NULL;
   int i;
   int local_ok = 1;
   char* fname = "Read_vector";

   if (my_rank == 0) {
      a = malloc(n*sizeof(double));
      printf("Enter the vector %s\n", vec_name);
      for (i = 0; i < n; i++)
         scanf("%lf", &a[i]);
      MPI_Scatter(a, 1, cont_mpi_t, local_a, 1, cont_mpi_t, 0, comm);
      free(a);
   } else {
      MPI_Scatter(a, 1, cont_mpi_t, local_a, 1, cont_mpi_t, 0, comm);
   }
}
```

Listing : Read_Vector

Just as the question requests, the second argument for the MPI_Scatter function, which is the *sendcount*, now says that we are sending only one value instead of local_n values. This happens because now we are sending one bigger package that we wrapped in the new structure called cont_mpi_t.

The listing below shows the implementation of the Print_vector, which follows the same approach as the Read_vector.

```
void Print_vector(
              double        local_b[]   /* in */,
              int           local_n     /* in */,
              int           n           /* in */,
              char          title[]     /* in */,
              MPI_Datatype  cont_mpi_t  /* in */) {

   double* b = NULL;
   int i;
   int local_ok = 1;
   char* fname = "Print_vector";

   if (my_rank == 0) {
      b = malloc(n*sizeof(double));

      MPI_Gather(local_b, 1, cont_mpi_t, b, 1, cont_mpi_t, 0, comm);

      printf("%s\n", title);
      for (i = 0; i < n; i++)
         printf("%f ", b[i]);
      printf("\n");
      free(b);
   } else {
      MPI_Gather(local_b, 1, cont_mpi_t, b, 1, cont_mpi_t, 0, comm);

   }
}
```

Listing: Print_vector implementation

Just as the listing for the Read_vector, we now do not need to have the local_n as a number of receiving and sending count. Now, since we have everything neatly packed into the structure, all MPI needs to send is the structure of type *cont_mpi_t*.

Question 3.18

MPI_Type_vector can be used to build a derived datatype from a collection of block of elements in an array as long as the blocks all have the same size and they're equally spaced. Its syntax is:

```
int MPI_Type_vector(
        int      count                /* in */,
        int      blocklenght          /* in */,
        int      stride               /* in */,
        MPI_Datatype  old_mpi_t       /* in */,
        MPI_Datatype* new_mpi_t_p     /* in */ );
```

For example, if we had an array x of 18 doubles and we wanted to build a type corresponding to the elements in positions 0, 1, 6,7, 12,13, we could call

```
int MPI_Type_vector( 3, 2 , 6, MPI_DOUBLE, &vect_mpi_t);
```

since the type consists of 3 blocks, each which has 2 elements, and the spacaing between the starts of the blocks is 6 doubles.

Write Read_vector and Print_vector functions that will allow process 0 to read and print, respectively, a vector with a block-cyclic distribution. But be aware! Do *not* use MPI_Scatter or MPI_Gather. There is a technincal issue involved in using these functions with types created with MPI_Type_vector and a loop of received on process 0 in Print_vector. The other processes should be able to complete their calls to Read_vector and Print_vector with a single call to MPI_Recv and MPI_Send. The communication on process 0 should use a derived datatype created by MPI_Type_vector. The calls on the other processes should just use the `count` argument to the communication function, since they're receiving/sending elements that they will store in contiguous array locations.

Answer

The new Read_vector function is shown below.

Question 3.19

Use MPI_Type_indexed to create a derived datatye that corresponds to the upper triangular part of a square matrix.

Process 0 should read in a $n \times n$ matrix as a one dimensional array, create the data type and send the triangular upper part with a single call to MPI_Send. Process 1 should receive the upper part triangular part with a single call to MPI_Recv and print the data it received.

Answer:

Up to this point, the part for receiving the order of the vectors from the user is the same. It reads a value and MPI_Bcast it to all processes. This is fine because we are using only one value. And right after, the allocation of memory now uses n*n*sizeof(double), since we are now dealing with a $n \times n$ matrix. At this moment we must create our new type using it MPI_Type_indexed.

Let's first understand how MPI_Type_indexed works. Based on the function signature given in the book, these are what the parameters do.

count	Number of blocks -- also number of entries in array_of_displacements and array_of_blocklengths (nonnegative integer)
array_of_blocklengths[]	Number of elements per block
array_of_displacements[]	Displacement for each block, in multiples of old type extend for MPI_Type_indexed
old_type	Old datatype
new type	New datatye

So, for example, imagine now that we have the following type as our old_type which has extension of 16bytes.

```
{ (double, 0) , (char,  8) }
```

And now we want to make a call to MPI_Type_indexed shuch as our count =2, array_of_blocklenghts[] = B = (3,1) and our array_of_displacements[] = D = (4,0). So, this means that our new type has 2 blocks (count =2), the first block has 3 copies of the original type and the second has 1 (B= 3,1), and also the first block will have a memory displacement of 4 times the extension of the original type and the second 0 times. Thus, our new type would result the following.

Block 1	Block 2
{(double, 64), (char, 72), (double, 80), (char, 88), (double, 96), (char, 104),	(double, 0), (char, 8)}

Now that we have a good understading of the functionality of the function, let's dig into the code. The following Listing shows how the new type is created.

```
void Build_indexed_type(
      int            n                /* in  */,
      MPI_Datatype*  indexed_mpi_t_p  /* out */) {
   int i;
   int* array_of_block_lens;
   int* array_of_disps;

   array_of_block_lens = malloc(n*sizeof(int));
   array_of_disps = malloc(n*sizeof(int));

   for (i = 0; i < n ; i++) {
      array_of_block_lens[i] = n-i; /* Note here that for a upper triangular, the blocks 
will be decrezing size */
      array_of_disps[i] = i*(n+1);  /* For the array of displacements, we must remind 
the way it is saved into memory,
                                       as a big continuous array that a[i][j] = 
i*(cols+1) */
   }

   MPI_Type_indexed(n,            /* square matrix order. Here, the # of blocks */
          array_of_block_lens,    /* # of elements per block */
          array_of_disps,         /* displacement for each block, in multiples of 
oldtype extend for MPI_Type_indexed */
                                  /* and bytes for MPI_type_create */
          MPI_DOUBLE,             /* Old mpi type */
          indexed_mpi_t_p);       /* New datatype */
   MPI_Type_commit(indexed_mpi_t_p); /* DO NOT FORGET TO COMMIT YOUR CHANGES! */

   free(array_of_block_lens);
   free(array_of_disps);
}
```

Listing: Build_indexed_type code

Now, let's request from the user the Matrix that we are going to use.

```
void Read_loc_mat(
             double    loc_mat[]  /* out */,
             int       n          /* in  */) {
   int i,j;

   printf("Enter the matrix\n");
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         scanf("%lf", &loc_mat[i*n + j]); // Againg, the continuous array is used here
to "simulate" a matrix.
}
```

Then, we must assign each process specific job. So, as requested from the question, the process have read and now its time to distribute the variables. Also, process 1 should receive it and print onto the screen. The following Listing shows the treatment in the main() function. Note that process 1 has its local matrix full of zeros because, from theory, an upper triangular matrix has the bottom triangular part full of zeros.

```
   if (my_rank == 0) {
      Read_loc_mat(loc_mat, n);
      MPI_Send(loc_mat, 1, indexed_mpi_t, 1, 0, comm);
   } else if (my_rank ==1){
      memset(loc_mat, 0, n*n*sizeof(double)); // for the local_matix, fill it with zeros
      MPI_Recv(loc_mat, 1, indexed_mpi_t, 0, 0, comm, MPI_STATUS_IGNORE); //receives the
resulting triangular matrix of type indexed_mpi_t
      Print_loc_mat("Received matrix", loc_mat, n); //prints it
   }
```

The printing function is shown on the listing below.

```
void Print_loc_mat(
             char      title[]    /* in */,
             double    loc_mat[]  /* in */,
             int       n          /* in */) {
   int i,j;

   printf("Proc %d > %s\n", my_rank, title);
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)
         printf("%.2f ", loc_mat[i*n + j]);
      printf("\n");
   }
   printf("\n");
}
```

The whole algorithm is shown below.

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

```
#include <mpi.h>

int my_rank, comm_sz;
MPI_Comm comm;


void Get_n(int* n_p);
void Read_loc_mat(double loc_mat[], int n);
void Build_indexed_type(int n, MPI_Datatype* indexed_mpi_t_p);
void Print_loc_mat(char title[], double loc_mat[], int n);


int main(void) {
   int n;
   double *loc_mat;
   MPI_Datatype indexed_mpi_t;

   MPI_Init(NULL, NULL);
   comm = MPI_COMM_WORLD;
   MPI_Comm_size(comm, &comm_sz);
   MPI_Comm_rank(comm, &my_rank);

   if (comm_sz < 2){
     printf("Please provide n greater than 2 \n", );
     return -1;
   }

   Get_n(&n);
   loc_mat = malloc(n*n*sizeof(double));

   Build_indexed_type(n, &indexed_mpi_t);
   if (my_rank == 0) {
      Read_loc_mat(loc_mat, n);
      MPI_Send(loc_mat, 1, indexed_mpi_t, 1, 0, comm);
   } else if (my_rank ==1){
      memset(loc_mat, 0, n*n*sizeof(double)); // for the local_matix, fill it with zeros
      MPI_Recv(loc_mat, 1, indexed_mpi_t, 0, 0, comm, MPI_STATUS_IGNORE); //receives the
resulting triangular matrix of type indexed_mpi_t
      Print_loc_mat("Received matrix", loc_mat, n); //prints it
   }

   free(loc_mat);
   MPI_Type_free(&indexed_mpi_t);
   MPI_Finalize();
   return 0;
}

void Get_n(int*    n_p          /* out */) {

   if (my_rank == 0) {
      printf("Enter the order of the matrix\n");
      scanf("%d", n_p);
   }

   MPI_Bcast(n_p, 1, MPI_INT, 0, comm);

}

void Build_indexed_type(
      int            n                /* in  */,
      MPI_Datatype*  indexed_mpi_t_p  /* out */) {
   int i;
   int* array_of_block_lens;
   int* array_of_disps;
   array_of_block_lens = malloc(n*sizeof(int));
   array_of_disps = malloc(n*sizeof(int));

   for (i = 0; i < n ; i++) {
      array_of_block_lens[i] = n-i; /* Note here that for a upper triangular, the blocks
will be decrezing size */
      array_of_disps[i] = i*(n+1);  /* For the array of displacements, we must remind
the way it is saved into memory,
                                      as a big continuous array that a[i][j] =
i*(cols+1) */
   }

   MPI_Type_indexed(n,             /* square matrix order. Here, the # of blocks */
          array_of_block_lens,     /* # of elements per block */
          array_of_disps,          /* displacement for each block, in multiples of
oldtype extend for MPI_Type_indexed */
                                   /* and bytes for MPI_type_create */
          MPI_DOUBLE,              /* Old mpi type */
          indexed_mpi_t_p);        /* New datatype */
   MPI_Type_commit(indexed_mpi_t_p); /* DO NOT FORGET TO COMMIT YOUR CHANGES! */

   free(array_of_block_lens);
   free(array_of_disps);
}

void Read_loc_mat(
             double    loc_mat[]  /* out */,
             int       n          /* in  */) {
   int i,j;

   printf("Enter the matrix\n");
   for (i = 0; i < n; i++)
      for (j = 0; j < n; j++)
         scanf("%lf", &loc_mat[i*n + j]); // Againg, the continuous array is used here
to "simulate" a matrix.
}


void Print_loc_mat(
             char      title[]    /* in */,
             double    loc_mat[]  /* in */,
             int       n          /* in */) {
   int i,j;

   printf("Proc %d > %s\n", my_rank, title);
   for (i = 0; i < n; i++) {
      for (j = 0; j < n; j++)
         printf("%.2f ", loc_mat[i*n + j]);
      printf("\n");
   }
   printf("\n");
}
```

Question 3.20

Write another Get_input function for the trapezoidal rule program. This one should use MPI_Pack on process 0 and MPI_Unpack on the other processes.

Answer

The major difference here is that now we are sending all values packed into a new format. This buffer is like a train: every group of people that comes in (pack_buf) is going to be allocated right after the previous. And when the train is ready, it is sent with everybody inside.

```
void Get_input(
      int      my_rank  /* in  */,
      double*  a_p      /* out */,
      double*  b_p      /* out */,
      int*     n_p      /* out */) {

   char pack_buf[PACK_BUF_SZ];
   int position = 0;

   if (my_rank == 0) {
      printf("Enter a, b, and n\n");
      scanf("%lf %lf %d", a_p, b_p, n_p);
      MPI_Pack(a_p, 1, MPI_DOUBLE, pack_buf, PACK_BUF_SZ, &position,
            MPI_COMM_WORLD);
      MPI_Pack(b_p, 1, MPI_DOUBLE, pack_buf, PACK_BUF_SZ, &position,
            MPI_COMM_WORLD);
      MPI_Pack(n_p, 1, MPI_INT, pack_buf, PACK_BUF_SZ, &position,
            MPI_COMM_WORLD);
   }

   MPI_Bcast(pack_buf, PACK_BUF_SZ, MPI_PACKED, 0, MPI_COMM_WORLD);

   if (my_rank > 0) {
      MPI_Unpack(pack_buf, PACK_BUF_SZ, &position, a_p, 1, MPI_DOUBLE,
            MPI_COMM_WORLD);
      MPI_Unpack(pack_buf, PACK_BUF_SZ, &position, b_p, 1, MPI_DOUBLE,
            MPI_COMM_WORLD);
      MPI_Unpack(pack_buf, PACK_BUF_SZ, &position, n_p, 1, MPI_INT,
            MPI_COMM_WORLD);
   }
}
```

Question 3.25

If comm_sz = p, we mentioned that the "ideal" speedup is p. Is it possible to do better?

a. Consider a parallel program that computers a vector sum. If we only time the vector sum - that is, we ignore input and output of the vectors- how might this program achieve speedup greater than p?
b. A program that achieves speedup greater than p is said to have superlinear speedup. Our vector sum example only achieved superlinear speedup by overcoming certain "resource limitations". What were these resource limitations? Is it possible for a program to obtain superlinear speedup without overcoming resource limitations?

Answer:

a. If the vectors don't fit into cache in single process run but they do fit into cache in the multiprocess run, it could happen that the multiprocess obtained better than linear speedup, since the average time to add two components might be less: the loads and stores in the multiprocess run might not have to directly access the main memory.

b. For the case of the vector sum, the reason lives in the cache size. When we ran the program on a single process, the vector could not fit into a single cache, which caused more requests to the memory.

When the vector sum was parallelized, the division into chunks made possible that the caches from the different processors fit way more of the vector. Thus, requiring way less requests to the memory.

Question 3.26

Serial odd-even transportation sort of an n-element list can sort the list in considerably slower fewer than n phases. As an extreme example, if the input list is already sorted, the algorithm requires 0 phases.

a. Write a serial Is_sorted function that determines whether a list is sorted.
b Modify the serial odd-even transposition sort program so that it checks whether the list is sorted after each phase.
c. If this program get improved performance by checking whether the list is sorted?

Answer:

a. The function returns an integer value meaning that the list a[] is sorted or not. If the array is sorted, it returns the value of 1. If it is not sorted, returns a value of 0.

```
int Is_sorted(int a[], int n) {
   int i;

   for (i = 1; i < n; i++)
      if (a[i-1] > a[i]) return 0;
   return 1;
}
```

b. The only thing that is done here is the checking, before each phase, if the array is already sorted using the function Is_sorted. If the array is already sorted, it returns from the function.

```
void Odd_even_sort(int a[], int n) {
   int phase, i, temp;
   for (phase = 0; phase < n; phase++) {
      if (Is_sorted(a, n) == 1) return; //checks if the list is sorted. If it is, 
return.
```

```
      if (phase % 2 == 0) { /* Even phase */
         for (i = 1; i < n; i += 2)
            if (a[i-1] > a[i]) {
               temp = a[i];
               a[i] = a[i-1];
               a[i-1] = temp;
            }
      } else { /* Odd phase */
         for (i = 1; i < n-1; i += 2)
            if (a[i] > a[i+1]) {
               temp = a[i];
               a[i] = a[i+1];
               a[i+1] = temp;
            }
      }
   }
}
```

c. If the code is executed using random numbers in an array, there should be no improvements. The reason is simple, the checking has an advantage over the non-checking should exist if the array has most part of it already sorted. So, when the first chunk is sorted, the algorithm should see that the array is already sorted and stop it since there is no need to keep doing it anymore.