1 Point-to-point communication

Question 1–1

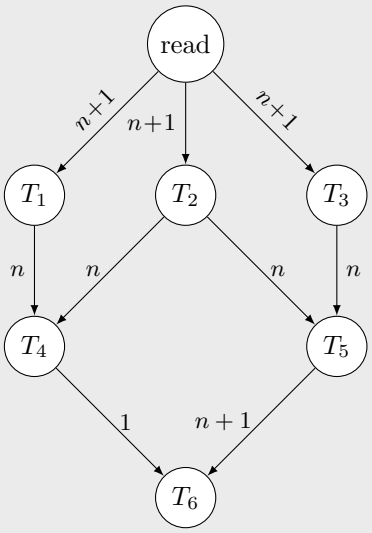
Given the following function:

```
double function()
{
   int i,j,n;
   double *v,*w,*z,sv,sw,x,res=0.0;

   /* Read vectors v, w, z, of size n */
   read(&n, &v, &w, &z);

   compute_v(n,v);              /* task 1 */
   compute_w(n,w);              /* task 2 */
   compute_z(n,z);              /* task 3 */

   /* task 4 */
   for (j=0; j<n; j++) {
      sv = 0;
      for (i=0; i<n; i++) sv = sv + v[i]*w[i];
      for (i=0; i<n; i++) v[i]=sv*v[i];
   }

   /* task 5 */
   for (j=0; j<n; j++) {
      sw = 0;
      for (i=0; i<n; i++) sw = sw + w[i]*z[i];
      for (i=0; i<n; i++) z[i]=sw*z[i];
   }

   /* task 6 */
   x = sv+sw;
   for (i=0; i<n; i++) res = res+x*z[i];

   return res;
}
```

Functions `compute_X` have as input the data received as arguments and from them the functions update vector `X`. For instance, `compute_v(n,v)` takes as input data the values of `n` and `v` and modifies vector `v`.

(a) Draw a dependency graph. for the different tasks, including in the graph the cost of each task and the volume of the communications. Assume that the functions `compute_X` have a cost of $2n^2$ flops.

Solution: The communication costs appear in the edges of the graph.



The cost (execution time) of task 4 is:

$$\sum_{j=0}^{n-1}\left(\sum_{i=0}^{n-1}2+\sum_{i=0}^{n-1}1\right)=\sum_{j=0}^{n-1}(2n+n)=3n^2$$

The cost of T_5 is the same as that of T_4, and the cost of T_6 is $2n$.

(b) Implement a parallel version using MPI, considering that all the MPI processes execute the different tasks without splitting them into sub-tasks. We can assume that the program will be executed with at least 3 processes.

Solution: There are different possibilities to do the assignment. It is necessary to take into account that only one of the processes must do the reading. Tasks 1, 2 and 3 are independent and therefore can be assigned to 3 distinct processes. The same occurs with tasks 4 and 5.

For example, process 0 can be in charge of reading, and doing tasks 1 and 4. Process 1 can do task 2, and process 2 can do tasks 3, 5 and 6.

Given that the processes other than P_0 do not know the value of `n`, it is necessary to send this value in a separate message, and once it has been received it is possible to allocate the necessary memory with `malloc`. This message has a smaller length, because we are sending an integer variable instead of one of `double` type. However, to simplify, this difference has not been considered with computing the costs.

```
double funcion()
{
   int i,n,j;
   double *v,*w,*z,sv,sw,x,res=0.0;
   int p,rank;
   MPI_Status status;

   MPI_Comm_size(MPI_COMM_WORLD,&p);
   MPI_Comm_rank(MPI_COMM_WORLD,&rank);

   if (rank==0) {
      /* T0: read vectors v, w, z, of dimension n */
      read(&n, &v, &w, &z);
```

```
            MPI_Send(&n, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
            MPI_Send(&n, 1, MPI_INT, 2, 0, MPI_COMM_WORLD);
            MPI_Send(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
            MPI_Send(z, n, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);

            calcula_v(n,v);              /* task 1 */
            MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);

            /* task 4 (same code as the sequential case) */
            ...

            MPI_Send(&sv, 1, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);

            MPI_Recv(&res, 1, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD, &status);
         }
         else if (rank==1) {
            MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            w = (double*) malloc(n*sizeof(double));
            MPI_Recv(w, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

            calcula_w(n,w);              /* task 2 */

            MPI_Send(w, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Send(w, n, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD);

            MPI_Recv(&res, 1, MPI_DOUBLE, 2, 0, MPI_COMM_WORLD, &status);
         }
         else if (rank==2) {
            MPI_Recv(&n, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
            z = (double*) malloc(n*sizeof(double));
            MPI_Recv(z, n, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

            calcula_z(n,z);              /* task 3 */
            MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);

            /* task 5 (same code as the sequential case) */
            ...

            MPI_Recv(&sv, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, &status);

            /* task 6 (same code as the sequential case) */
            ...

            /* Send the result of task 6 to the rest of processes */
            MPI_Send(&res, 1, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD);
            MPI_Send(&res, 1, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD);
         }
         return res;
      }
```

(c) Indicate the execution time of the sequential algorithm, that of the parallel algorithm, and the

associated Speedup. Assume that the cost of reading the vectors is negligible.

Solution: Taking into account that execution time of each of tasks 1, 2 and 3 is $2n^2$, while that of tasks 4 and 5 is $3n^2$, and that of task 6 is $2n$, the sequential execution time will be the sum of these times:

$$t(n) = 3 \cdot 2n^2 + 2 \cdot 3n^2 + 2n \approx 12n^2$$

Parallel execution time: arithmetic time. It will be equal to the arithmetic time of the process that performs the most number of operations, which in this case is process 2, performing tasks 3, 5 and 6. Therefore

$$t_a(n,p) = 2n^2 + 3n^2 + 2n \approx 5n^2$$

Parallel execution time: communications time. The sent messages are:

- 2 messages, from process 0 to the rest, with the value of `n`. Cost of $2(t_s + t_w)$.
- 2 messages, one from process 0 to 1 with vector `w`, and another from 0 to 2 with vector `z`. Cost of $2(t_s + nt_w)$.
- 2 messages, from process 1 to the rest, with vector `w`. Cost of $2(t_s + nt_w)$.
- 1 message, from process 0 to 2, with the value of `sv`. Cost of $(t_s + t_w)$
- 2 messages, from process 2 to the rest, with the value of `res`. Cost of $2(t_s + t_w)$

Therefore, the cost of communications will be:

$$t_c(n,p) = 5(t_s + t_w) + 4(t_s + nt_w) = 9t_s + (5 + 4n)t_w \approx 9t_s + 4nt_w$$

Total parallel execution time:

$$t(n,p) = t_a(n,p) + t_c(n,p) = 5n^2 + 9t_s + 4nt_w$$

Speedup:

$$S(n,p) = \frac{12n^2}{5n^2 + 9t_s + 4nt_w}$$

Question 1–2

Implement a function that, given a vector of dimension `n`, cyclically distributed by row blocks among `p` processes, perform the needed communication operations to end-up with a replicated copy of the whole vector in all the processes. N.B.: only use point-to-point communication.

The header for the function will be:

```
void communicate_vector(double vloc[], int n, int b, int p, double w[])
/* vloc: local part of the initial v vector
   n: global dimension of v vector
   b: block size used in the distribution of the vector v
   p: number of processes
   w: vector of size n, where a complete copy of the v vector is stored
 */
```

Solution: We assume that `n` is a multiple of the bloc size `b` (that is, all blocs have size `b`).

```
void comunicate_vector(double vloc[], int n, int b, int p, double w[])
/* vloc: local part of the initial v vector
   n: global dimension of v vector
   b: block size used in the distribution of the vector v
   p: number of processes
   w: vector of size n, where a complete copy of the v vector is stored
 */
{
   int i, rank, rank_pb, rank2;
   int ib, ib_loc;       /* Block index */
   int num_blq=n/b;      /* Number of blocks */
   MPI_Status status;

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   for (ib=0; ib<num_blq; ib++) {
      rank_pb = ib%p;           /* block owner */
      if (rank==rank_pb) {
         ib_loc = ib/p;         /* local block index */
         /* Send block ib to all processes except myself */
         for (rank2=0; rank2<p; rank2++) {
            if (rank!=rank2) {
               MPI_Send(&vloc[ib_loc*b], b, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
            }
         }
         /* Copy block ib in my own local vector */
         for (i=0; i<b; i++) {
            w[ib*b+i]=vloc[ib_loc*b+i];
         }
      } else {
         MPI_Recv(&w[ib*b], b, MPI_DOUBLE, rank_pb, 0, MPI_COMM_WORLD, &status);
      }
   }
}
```

Question 1–3

A set of T tasks will be executed on the elements of a real vector of size `n`. This tasks should be applied sequentially and ordered. The function implementing the task has this syntax:

```
void task(int task_type, int n, double *v);
```

where `task_type` identifies the number of the task from 1 to T. These tasks will be applied on m vectors. These vectors are stored in a matrix A in the master process, where each row is each one of those m vectors.

Implement an MPI parallel program following the *Pipeline* scheme, where each process ($P_1 \dots P_{p-1}$) will execute each one of the T tasks ($T = p - 1$).

Master process (P_0) will simply feed the pipeline and it will collect the vectors obtained at the end of the pipe, storing them back in the A matrix. An empty message with a special tag can be used to notify about the end of the process (we assume that the slaves do not know the number of vectors (m).

Solution: The following code implements the feeding of the pipe:

```
#define TASK_TAG  123
#define END_TAG    1
int continue,num;
MPI_Status stat;
if (!rank) {
  for (i=0;i<m;i++) {
    MPI_Send(&A[i*n], n, MPI_DOUBLE, 1, TASK_TAG, MPI_COMM_WORLD);
  }
  MPI_Send(0, 0, MPI_DOUBLE, 1, FIN_TAG, MPI_COMM_WORLD);
  for (i=0;i<m;i++) {
    MPI_Recv(&A[i*n], n, MPI_DOUBLE, p-1, TASK_TAG, MPI_COMM_WORLD, &stat);
  }
  MPI_Recv(0, 0, MPI_DOUBLE, p-1, FIN_TAG, MPI_COMM_WORLD, &stat);
} else {
  continue = 1;
  while (continue) {
    MPI_Recv(A, n, MPI_DOUBLE, rank-1, MPI_ANY_TAG, MPI_COMM_WORLD, &stat);
    MPI_Get_count(&stat, MPI_DOUBLE, &num);
    if (stat.MPI_TAG == TASK_TAG) {
      task(rank, n, A);
    } else {
      continue = 0;
    }
    MPI_Send(A, num, MPI_DOUBLE, (rank+1)%p, stat.MPI_TAG, MPI_COMM_WORLD);
  }
}
```

Question 1–4

A parallel program executed on p processes has a vector x with size n distributed by blocks, and a vector y replicated in all processes. Implement the following function, which should sum the local part of vector x with the corresponding part of vector y, leaving the result in the local vector z.

```
void sum(double xloc[], double y[], double z[], int n, int p, int pr)
/* pr is the index of the local process */
```

Solution:

```
void sum(double xloc[], double y[], double z[], int n, int p, int pr)
/* pr is the index of the local process */
{
  int i, iloc, mb;

  mb = ceil(((double) n)/p);
  for (i=pr*mb; i<MIN((pr+1)*mb,n); i++) {
    iloc=i%mb;
    z[iloc]=xloc[iloc]+y[i];
  }
}
```

Question 1–5

The Levenshtein distance computes a measure of similarity among two strings. The following sequential

code uses such distance to compute the position in which a sub-string is much alike that the input sequence, assuming that strings are read from a text file.

Example: if the string `ref` is `"aafsdluqhqwBANANAqewrqerBANAfqrqerqrABANArqwrBAANANqwe"` and `str` is `"BANAN"`, the program will show that `"BANAN"` has the maximum similarity with a substring that starts in the position 11.

```
int mindist, pos, dist, i, ls, lr;
FILE *f1, *f2;
char ref[500], str[100];

f1 = fopen("ref.txt","r");
fgets(ref,500,f1);
lr = strlen(ref);
printf("Ref: %s (%d)\n", ref, lr);
fclose(f1);

f2 = fopen("lines.txt","r");
while (fgets(str,100,f2)!=NULL) {
  ls = strlen(str);
  printf("Str: %s (%d)\n", str, ls);
  mindist = levenshtein(str, ref);
  pos = 0;
  for (i=1;i<lr-ls;i++) {
    dist = levenshtein(str, &ref[i]);
    if (dist < mindist) {
      mindist = dist;
      pos = i;
    }
  }
  printf("Distance %d for %s in %d\n", mindist, str, pos);
}
fclose(f2);
```

(a) Complete the following MPI parallel implementation of the algorithm according to the master-slave approach:

```
int mindist, pos, dist, i, ls, lr, count, rank, size, rc, org;
FILE *f1, *f2;
char ref[500], str[100], c;
MPI_Status status;

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

if (rank ==0) {    /* master */
  f1 = fopen("ref.txt","r");
  fgets(ref,500,f1);
  lr = strlen(ref);
  ref[lr-1]=0;
  lr--;
  MPI_Bcast(ref, lr+1, MPI_CHAR, 0, MPI_COMM_WORLD);
  printf("Ref: %s (%d)\n", ref, lr);
  fclose(f1);
```

```
  f2 = fopen("lines.txt","r");
  count = 1;
  while ( (fgets(str,100,f2)!=NULL) && (count<size) ) {
    ls = strlen(str);
    str[ls-1] = 0;
    ls--;
    MPI_Send(str, ls+1, MPI_CHAR, count, TAG_WORK, MPI_COMM_WORLD);
    count++;
  }

  do {
    printf("%d processes active\n", count);
    /*
      COMPLETE
      - receive three messages from the same process
      - read a new line from the file and send it
      - if end of file, send a termination message
    */
  } while (count>1);

  fclose(f2);
} else {    /* worker */
    MPI_Bcast(ref, 500, MPI_CHAR, 0, MPI_COMM_WORLD);
    lr = strlen(ref);
    printf("[%d], Ref: %s\n", rank, ref);
    rc = 0;
    do {
      MPI_Recv(str, 100, MPI_CHAR, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
      ls = strlen(str);
      if (status.MPI_TAG == TAG_WORK) {
        printf("[%d] Message received (%s)\n", rank, str);
        mindist = levenshtein(str, ref);
        pos = 0;
        for (i=1;i<lr-ls;i++) {
          dist = levenshtein(str, &ref[i]);
          if (dist < mindist) {
            mindist = dist;
            pos = i;
          }
        }

        printf("[%d] sends: %d, %d, and %s to 0\n", rank, mindist, pos, str);
        MPI_Send(&mindist, 1, MPI_INT, 0, TAG_RESULT, MPI_COMM_WORLD);
        MPI_Send(&pos, 1, MPI_INT, 0, TAG_POS, MPI_COMM_WORLD);
        MPI_Send(str, ls+1, MPI_CHAR, 0, TAG_STR, MPI_COMM_WORLD);
      } else {
        printf("[%d] receives message with tag %d\n", rank, status.MPI_TAG);
        rc = 1;
      }
   } while (!rc);
}
```

Solution:

```
do {
  printf("%d processes active\n", count);
  MPI_Recv(&mindist, 1, MPI_INT, MPI_ANY_SOURCE, TAG_RESULT,
           MPI_COMM_WORLD, &status);
  org = status.MPI_SOURCE;
  MPI_Recv(&pos, 1, MPI_INT, org, TAG_POS, MPI_COMM_WORLD, &status);
  MPI_Recv(str, 100, MPI_CHAR, org, TAG_STR, MPI_COMM_WORLD, &status);
  ls = strlen(str);
  printf("From [%d]: Distance %d for %s in %d\n", org, mindist, str, pos);
  count--;

  rc = (fgets(str,100,f2)!=NULL);
  if (rc) {
    ls = strlen(str);
    str[ls-1] = 0;
    ls--;
    MPI_Send(str, ls+1, MPI_CHAR, org, TAG_WORK, MPI_COMM_WORLD);
    count++;
  } else {
    printf("Sending terminate message to %d\n", status.MPI_SOURCE);
    MPI_Send(&c, 1, MPI_CHAR, org, TAG_END, MPI_COMM_WORLD);
  }
} while (count>1);
```

(b) Compute the communication cost of the parallel version developed considering that there are n lines of an average size of m, the reference is of length lr, and the number of processes is p.

Solution: In the version proposed, the communication cost is due to four main concepts:

- Broadcast of the reference ($lr + 1$ bytes): $(t_s + t_w \cdot (lr + 1)) \cdot (p - 1)$
- Individual message for each sequence ($ls_i + 1$ bytes): $(t_s + t_w \cdot (ls_i + 1)) \cdot n$
- Three messages for the response of each sequence (two integers plus $ls_i + 1$ bytes): $(t_s + t_w \cdot (ls_i + 1)) \cdot n + 2 \cdot n \cdot (t_s + 4 \cdot t_w)$
- Termination message (1 byte): $(t_s + t_w) \cdot (p - 1)$

Therefore, the total cost can be approximated by $2 \cdot n \cdot t_s + t_w \cdot (9 \cdot n + m)$.

Question 1–6

We want to parallelize the following code with MPI. Suppose that 3 processes are available.

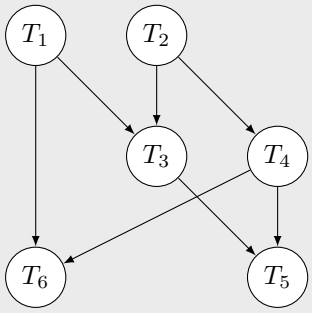
```
double a[N],b[N],c[N],v=0.0,w=0.0;
T1(a,&v);
T2(b,&w);
T3(b,&v);
T4(c,&w);
T5(c,&v);
T6(a,&w);
```

All functions read and modify both arguments, also the vectors. Suppose that vectors `a`, `b` and `c` are stored in P_0, P_1 and P_2, respectively, and they are too large to afford its efficient sending from one process

to another.

(a) Draw the dependency graph of the different tasks, indicating which task is assigned to each process.

Solution: The dependency graph is the following:



Due to the restriction of where the vectors are located, we will do the following assignment: T_1 and T_6 in P_0, T_2 and T_3 in P_1, T_4 and T_5 in P_2.

(b) Write the MPI code that solves the problem.

Solution:

```
double a[N],b[N],c[N],v=0.0,w=0.0;
int p,rank;
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0) {
  T1(a,&v);
  MPI_Send(&v, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD);
  MPI_Recv(&w, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  T6(a,&w);
} else if (rank==1) {
  T2(b,&w);
  MPI_Send(&w, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD);
  MPI_Recv(&v, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  T3(b,&v);
  MPI_Send(&v, 1, MPI_DOUBLE, 2, 111, MPI_COMM_WORLD);
} else { /* rank==2 */
  MPI_Recv(&w, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  T4(c,&w);
  MPI_Send(&w, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);
  MPI_Recv(&v, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  T5(c,&v);
}
```

Question 1–7

The following fragment of code is incorrect (from the semantic point of view, not for syntax errors). Describe the reason and propose two different solutions.

```
MPI_Status stat;
int sbuf[N], rbuf[N], rank, size, src, dst;
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &size);
src = (rank==0)? size-1: rank-1;
```

```
dst = (rank==size-1)? 0: rank+1;
MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
```

Solution: The code implements a ring-based communication, where each process sends `N` integer elements to the process on its right. The last process will send the message to process 0. It is incorrect since a deadlock situation appears. The `MPI_Ssend` API is synchronous and therefore, all processes will wait for the receiving operation to be completed. However, no process will reach the receiving call to `MPI_Recv`, so all processes will remain blocked. The use of standard sending primitives, such as `MPI_Send` does not solve the issue, since it does not guarantee that the communications are performed asynchronously (it depends on the size of the message and the MPI implementation).

One solution will be to implement an even-odd protocol changing the last two lines by:

```
if (rank%2==0) {
  MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
  MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
} else {
  MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
  MPI_Ssend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD);
}
```

Another solution will be the use of a combined primitive:

```
MPI_Sendrecv(sbuf, N, MPI_INT, dst, 111, rbuf, N, MPI_INT, src,
             111, MPI_COMM_WORLD, &stat);
```

Finally, another possibility could be the use of non-blocking primitives, such as:

```
MPI_Isend(sbuf, N, MPI_INT, dst, 111, MPI_COMM_WORLD, &req);
MPI_Recv(rbuf, N, MPI_INT, src, 111, MPI_COMM_WORLD, &stat);
MPI_Wait(&req, MPI_STATUS_IGNORE);
```

Question 1–8

We want to implement the computation of the ∞-norm of a square matrix, which is obtained as the maximum of the sums of the absolute values of the elements in each row, $\max_{i=0}^{n-1}\left\{\sum_{j=0}^{n-1}|a_{i,j}|\right\}$. For this, a master-slave scheme is proposed. The next function corresponds to the master (the process with identifier 0). The matrix is stored by rows in a uni-dimensional array, and we assume that it is sparse (it has many zeros), and therefore the master sends only the nonzero elements (function `compress`).

```
int compress(double *A,int n,int i,double *buf)
{
  int j,k = 0;
  for (j=0;j<n;j++)
    if (A[i*n+j]!=0.0) { buf[k] = A[i*n+j]; k++; }
  return k;
}
double master(double *A,int n)
{
  double buf[n];
  double norm=0.0,value;
  int row,complete=0,size,i,k;
```

```
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  for (row=0;row<size-1;row++) {
    if (row<n) {
      k = compress(A, n, row, buf);
      MPI_Send(buf, k, MPI_DOUBLE, row+1, TAG_ROW, MPI_COMM_WORLD);
    } else
      MPI_Send(buf, 0, MPI_DOUBLE, row+1, TAG_END, MPI_COMM_WORLD);
  }
  while (complete<n) {
    MPI_Recv(&value, 1, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_RESU,
             MPI_COMM_WORLD, &status);
    if (value>norm) norm=value;
    complete++;
    if (row<n) {
      k = compress(A, n, row, buf);
      row++;
      MPI_Send(buf, k, MPI_DOUBLE, status.MPI_SOURCE, TAG_ROW, MPI_COMM_WORLD);
    } else
      MPI_Send(buf, 0, MPI_DOUBLE, status.MPI_SOURCE, TAG_END, MPI_COMM_WORLD);
  }
  return norma;
}
```

Implement the part of the working processes, completing the following function:

```
void worker(int n)
{
  double buf[n];
```

Note: For the absolute value you can use

```
double fabs(double x)
```

Remember that `MPI_Status` contains, among other, the fields `MPI_SOURCE` and `MPI_TAG`.

Solution:

```
void worker(int n)
{
  double buf[n];
  double s;
  int i,k;
  MPI_Status status;
  MPI_Recv(buf, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  while (status.MPI_TAG==TAG_ROW) {
    MPI_Get_count(&status, MPI_DOUBLE, &k);
    s=0.0;
    for (i=0;i<k;i++) s+=fabs(buf[i]);
    MPI_Send(&s, 1, MPI_DOUBLE, 0, TAG_RESU, MPI_COMM_WORLD);
    MPI_Recv(buf, n, MPI_DOUBLE, 0, MPI_ANY_TAG, MPI_COMM_WORLD, &status);
  }
}
```

Question 1–9

We want to measure the latency of a ring of `p` MPI processes, where the latency must be understood as the time that a message of length 0 spends when circulating across all processes. A ring of `p` MPI processes works as follows: P_0 sends the message to P_1, when the reception is complete, it resends the message to P_2, and so on until it arrives to P_{p-1} who will send it again to P_0. Write an MPI program that implements this communication scheme and shows the latency. It is recommended that the message goes around the ring several times, and then take the average time in order to get a more reliable measurement.

Solution:

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

#define REPS 1000

int main(int argc, char *argv[]) {
  int rank, i, size, prevp, nextp;
  double t1, t2;
  unsigned char msg;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &size);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  nextp = (rank+1)%size;
  if (rank>0) prevp = rank-1;
  else prevp = size-1;
  printf("I am %d, my left process is %d and my right process is %d\n",
         rank, prevp, nextp);

  t1 = MPI_Wtime();
  for (i=0;i<REPS;i++) {
    if (rank==0) {
      MPI_Send(&msg, 0, MPI_BYTE, nextp, i, MPI_COMM_WORLD);
      MPI_Recv(&msg, 0, MPI_BYTE, prevp, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    } else {
      MPI_Recv(&msg, 0, MPI_BYTE, prevp, i, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      MPI_Send(&msg, 0, MPI_BYTE, nextp, i, MPI_COMM_WORLD);
    }
  }
  t2 = MPI_Wtime();

  if (rank==0) {
    printf("Latency in a ring of %d processes: %f\n", size, (t2-t1)/REPS);
  }
  MPI_Finalize();
  return 0;
}
```

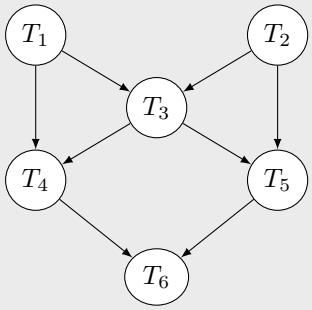
Question 1–10

Given the following function, where we suppose that the functions `T1`, `T3` and `T4` have a cost of n and the funtions `T2` and `T5` of $2n$, being n a constant value.

```
double example(int i,int j)
{
  double a,b,c,d,e;
  a = T1(i);
  b = T2(j);
  c = T3(a+b,i);
  d = T4(a/c);
  e = T5(b/c);
  return d+e;   /* T6 */
}
```

(a) Draw the dependency graph and compute the sequential cost.

Solution: The dependency graph is shown in the following figure:



The sequential cost is: $t(n) = n + 2n + n + n + 2n + 4 \approx 7n$

Note: the last 4 flops refer to the operations performed in the function `example` itself that do not correspond to any of the invoked functions.

(b) Parallelize it using MPI with two processes. Both processes invoke the function with the same value of the arguments `i`, `j` (it is not necessary to communicate them). The return value of the function must be correct in process 0 (it is not necessary that it is available in both processes).

Solution: In order to balance the load, we propose a solution in which process 1 performs `T2` and `T5`, and the rest of tasks are assigned to process 0.

```
double example(int i,int j)
{
  double a,b,c,d,e;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank==0) {
    a = T1(i);
    MPI_Recv(&b, 1, MPI_DOUBLE, 1, 111, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    c = T3(a+b,i);
    MPI_Send(&c, 1, MPI_DOUBLE, 1, 112, MPI_COMM_WORLD);
    d = T4(a/c);
    MPI_Recv(&e, 1, MPI_DOUBLE, 1, 113, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    b = T2(j);
    MPI_Send(&b, 1, MPI_DOUBLE, 0, 111, MPI_COMM_WORLD);
    MPI_Recv(&c, 1, MPI_DOUBLE, 0, 112, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    e = T5(b/c);
    MPI_Send(&e, 1, MPI_DOUBLE, 0, 113, MPI_COMM_WORLD);
```

```
    }
    return d+e;
  }
```

(c) Compute the parallel execution time (computation and communication) and the speedup with two processes.

Solution: The parallel time of the proposed implementation must be computed from the cost associated to the critical path of the dependency graph, corresponding to $T_2 - T_3 - T_5 - T_6$.

$$t(n,2) = t_{arit}(n,2) + t_{comm}(n,2)$$

$$t_{arit}(n,2) = 2n + n + 2n + 3 \approx 5n$$

$$t_{comm}(n,2) = 3 \cdot (t_s + t_w)$$

$$t(n,2) = 5n + 3t_s + 3t_w$$

The speedup:

$$S(n,2) = \frac{t(n)}{t(n,2)} = \frac{7n}{5n + 3t_s + 3t_w}$$

Question 1–11

Write a function using the provide header, which will make processes with ranks `proc1` and `proc2` exchange the elements of vector `x`. This vector is provided through the arguments. Vector `x` should not change in any other process.

```
void exchange(double x[N], int proc1, int proc2)
```

You should take into account the following:

- Potential dead-locks must be avoided.
- You cannot use functions `MPI_Sendrecv`, `MPI_Sendrecv_replace` or `MPI_Bsend`.
- You can declare any additional variable you need.
- The size of the vector (`N`) is a constant previously defined, and `proc1` and `proc2` are valid ranks for the processes (greater or equal than 0 and lower than the number of processes).

Solution:

```
int rank;
double x2[N];

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==proc1) {
  MPI_Send(x, N, MPI_DOUBLE, proc2, 100, MPI_COMM_WORLD);
  MPI_Recv(x, N, MPI_DOUBLE, proc2, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if (rank==proc2) {
  int i;
  MPI_Recv(x2, N, MPI_DOUBLE, proc1, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Send(x, N, MPI_DOUBLE, proc1, 100, MPI_COMM_WORLD);
  for (i=0; i<N; i++)
    x[i] = x2[i];
}
```

Question 1–12

The next function displays on the screen the maximum of a vector `v` with `n` elements and its location:

```
void func(double v[], int n) {
   double max = v[0];
   int i, posmax = 0;
   for (i=1; i<n; i++) {
      if (v[i]>max) {
         max = v[i];
         posmax=i;
      }
   }
   printf("Maxim: %f. Position: %d\n", max, posmax);
}
```

Write an MPI parallel version using the next header, where arguments `rank` and `np` have been obtained calling `MPI_Comm_rank` and `MPI_Comm_size`, respectively.

```
void func_par(double v[], int n, int rank, int np)
```

The function should assume that initially, process 0 will have the vector in the array `v`. In the rest of the processes, this array can be used to store each local part. You should exchange the necessary data to ensure that the computation of the maximum is balanced among all the processes. Finally, only process 0 should show the message on the screen. **You must use point-to-point communications calls and not collective operations**.

N.B. You can assume that `n` is an exact multiple of the number of processes.

Solution:

```
double max, max2;
int i, proc, posmax, posmax2, mb=n/np;

if (rank==0)
  for (proc=1; proc<np; proc++)
    MPI_Send(&v[proc*mb], mb, MPI_DOUBLE, proc, 100, MPI_COMM_WORLD);
else
  MPI_Recv(v, mb, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);

max = v[0];
posmax = 0;
for (i=1; i<mb; i++) {
  if (v[i]>max) {
    max = v[i];
    posmax=i;
  }
}
posmax += rank*mb;    /* Convert posmax in global index */

if (rank==0) {
  for (proc=1; proc<np; proc++) {
    MPI_Recv(&max2, 1, MPI_DOUBLE, proc, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(&posmax2, 1, MPI_INT, proc, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    if (max2>max) {
```

```
        max=max2;
        posmax=posmax2;
      }
    }
    printf("Maximum: %f. Position: %d\n", max, posmax);
  } else {
    MPI_Send(&max, 1, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD);
    MPI_Send(&posmax, 1, MPI_INT, 0, 100, MPI_COMM_WORLD);
  }
```

Question 1–13

We want to implement a function to distribute a square matrix across the processes of an MPI program, with the following header:

```
void communicate(double A[N][N], double Aloc[][N], int proc_row[N], int root)
```

The matrix `A` is stored initially in process `root`, and must be distributed by rows across the processes, in a such way that each row `i` must go to process `proc_row[i]`. The content of array `proc_row` is valid in all processes. Each process (including the `root`) must store the rows that have been assigned to it in the local matrix `Aloc`, occupying the first rows (that is, if a given process is assigned `k` rows, these must be stored in the first `k` rows of `Aloc`).

Example for 3 processes:

`A`

11	12	13	14	15
21	22	23	24	25
31	32	33	34	35
41	42	43	44	45
51	52	53	54	55

`proc_row`

0
2
0
1
1

`Aloc` in P_0

11	12	13	14	15
31	32	33	34	35

`Aloc` in P_1

41	42	43	44	45
51	52	53	54	55

`Aloc` in P_2

21	22	23	24	25

(a) Write the code of the function.

Solution:

```
void communicate(double A[][N], double Aloc[][N], int proc_row[], int root)
{
   int i, j, iloc, rank;

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   iloc=0;
   for (i=0; i<N; i++) {
      /* Treat row i */
      if (rank==root) {
         if (proc_row[i]==root) {       /* Local copy of the row */
            for (j=0; j<N; j++) Aloc[iloc][j] = A[i][j];
            iloc++;
         }
         else
```

```
                MPI_Send(&A[i][0], N, MPI_DOUBLE, proc_row[i], 0, MPI_COMM_WORLD);
            }
            else if (rank==proc_row[i]) {
               MPI_Recv(&Aloc[iloc][0], N, MPI_DOUBLE, root, 0, MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
               iloc++;
            }
         }
      }
```

(b) In a general case, would it be possible to use MPI's *vector* data type (`MPI_Type_vector`) to send all rows assigned to a given process by means of a single message? If it is possible, write the instructions to define it. Otherwise, justify why.

Solution: It is not possible because the rows that correspond to a process do not have, in general, a constant separation (stride) between them.

Question 1–14

Implement a *ping-pong* program.

The *ping-pong* parallel program will be executed by 2 processes, repeating 200 times sending one message with 100 integer values from process 0 to process 1 and then receiving the same message from process 1 in process 0. The program must print on the screen the average time for sending one integer value, obtained from the total time required for sending and receiving those messages.

The program can start like this:

```
int main(int argc,char *argv[])
{
  int v[100];
```

(a) Implement the parallel *ping-pong* program.

Solution:

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

#define V 200
#define N 100

int main(int argc,char *argv[])
{
  int i, me;
  double t;
  int v[N];

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&me);

  t = MPI_Wtime();

  for (i=0;i<V;i++)
    if (me==0) {
```

```
          MPI_Send(v, N, MPI_INT, 1, 20, MPI_COMM_WORLD);
          MPI_Recv(v, N, MPI_INT, 1, 17, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
        } else {
          MPI_Recv(v, N, MPI_INT, 0, 20, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
          MPI_Send(v, N, MPI_INT, 0, 17, MPI_COMM_WORLD);
        }

      t = MPI_Wtime() - t;
      if (me==0) printf("Average sending time for 1 integer: %g sec.\n",t/(2*V)/N);

      MPI_Finalize();
      return 0;
    }
```

(b) Compute the theoretical communication cost for the program.

Solution:

$$t_c = 2 * V * (t_s + N * t_w) = 400(t_s + 100t_w)$$

Question 1–15

A parallel program has already distributed a vector among the processes, using a block-oriented distribution, so each process stores its block in an array called `vloc`.

Implement a parallel function that will shift the elements of the vector one position to the right. The last element of the vector will be placed in the first position of the vector. For example, if we had 3 processes and the initial status is:

	P_0	P_1	P_2
`vloc`	`[2 5 3]`	`[7 1 0]`	`[6 4 9]`

The final status will be:

	P_0	P_1	P_2
`vloc`	`[9 2 5]`	`[3 7 1]`	`[0 6 4]`

The function will ensure that no deadlocks may happen. The header of the function will be:

```
void shift(double vloc[], int mb)
```

Where `mb` is the number of elements of local vector `vloc` (we will assume `mb` > 1).

Solution:

```
void shift(double vloc[], int mb)
{
   int prev, sig, p, rank, i;
   double elem;
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank==p-1) sig = 0;
   else sig = rank+1;
   if (rank==0) prev = p-1;
   else prev = rank-1;

   /* Local shift */
```

```
    elem = vloc[mb-1];
    for (i=mb-1; i>0; i--)
       vloc[i] = vloc[i-1];

    /* Communication among neighbouring processes */
    MPI_Sendrecv(&elem, 1, MPI_DOUBLE, sig, 0,
                 &vloc[0], 1, MPI_DOUBLE, prev, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

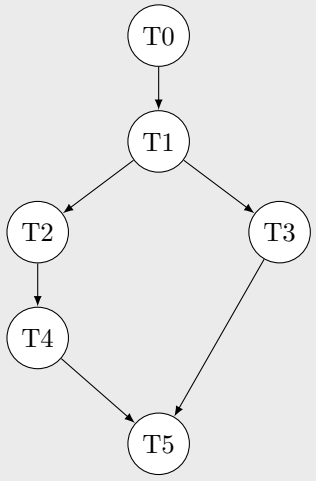
Question 1–16

In the next sequential program, where the computational cost of each function is indicated in the associated comment, all the invoked functions only modify the first argument. Take into account that `A`, `D` and `E` are vectors, while `B` and `C` are matrices.

```
#include <stdio.h>
int main (int argc, char *argv[]) {
    double A[N], B[N][N], C[N][N], D[N], E[N], res;
    read(A);              // T0, cost N
    generate(B,A);        // T1, cost 2N
    process2(C,B);        // T2, cost 2N^2
    process3(D,B);        // T3, cost 2N^2
    process4(E,C);        // T4, cost N^2
    res = process5(E,D); // T5, cost 2N
    printf("Result: %f\n", res);
    return 0;
}
```

(a) Draw the associated dependency graph.

Solution:



(b) Implement a parallel version using MPI, taking into account the following aspects:

- Use the most appropriate number of parallel processes to obtain the fastest run. Show an error message if the number of processes used when running the program is not the previous number. Only process P_0 should execute the calls to functions `read` and `printf`.
- Pay attention to the size of the messages and use merging and replication techniques if appropriate.

- Write the implementation as a whole program.

Solution: We will use two processes and we will replicate task `generate`, as the cost of sending a square matrix (`B`) will be higher than the cost of sending a vector (`A`)), and task `generate` cannot be executed in parallel with any other task.

```
#include <mpi.h>
#include <stdio.h>
int main (int argc, char *argv[]) {
    double A[N], B[N][N], C[N][N], D[N], E[N], res;
    int rank, p;
    MPI_Status status;

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

    if (p==2) {
       if (rank==0) {
           read(A);             // T0, cost N
           MPI_Send(A,N,MPI_DOUBLE,1,0,MPI_COMM_WORLD);
       } else {
           MPI_Recv(A,N,MPI_DOUBLE,0,0,MPI_COMM_WORLD,&status);
       }
       generate(B,A);           // T1, cost 2N
       if (rank==0) {
           process2(C,B);       // T2, cost 2N^2
           process4(E,C);       // T4, cost N^2
           MPI_Recv(D,N,MPI_DOUBLE,1,0,MPI_COMM_WORLD,&status);
           res = process5(E,D); // T5, cost 2N
           printf("Result: %f\n", res);
      } else {
           process3(D,B);       // T3, cost 2N^2
           MPI_Send(D,N,MPI_DOUBLE,0,0,MPI_COMM_WORLD);
       }
    } else
       if (rank==0)
          printf("Incorrect number of processes (%d)\n", p);

    MPI_Finalize();
    return 0;
}
```

(c) Compute the sequential and parallel cost, speed-up and efficiency.

Solution:

$$t(n) = N + 2N + 2N^2 + 2N^2 + N^2 + 2N = 5N + 5N^2 \approx 5N^2 \text{ flops}$$

$$t(n,p) = N+(t_s+Nt_w)+2N+2N^2+N^2+(t_s+Nt_w)+2N = 5N+3N^2+2t_s+2Nt_w \approx 3N^2+2t_s+2Nt_w$$

$$S(n,p) = \frac{5N^2}{3N^2 + 2t_s + 2Nt_w}$$

$$E(n,p) = \frac{5N^2}{2(3N^2 + 2t_s + 2Nt_w)}$$

2 Collective communication

Question 2–1

The following fragment of a code enables computing the product of a square matrix times a vector, both of the same size `N`:

```
int i, j;
int A[N][N], v[N], x[N];
leer(A,v);
for (i=0;i<N;i++) {
  x[i]=0;
  for (j=0;j<N;j++) x[i] += A[i][j]*v[j];
}
write(x);
```

Write an MPI program to implement the parallel product, assuming that the process P_0 will initially get matrix `A` and vector `v`, performing a block-row-wise distribution of `A` and replicating `v` in all the processes. Moreover, P_0 should have the result at the end.

N.B.: Assume that `N` is an exact multiple of the number of processes.

Solution: We define a auxiliary matrix `B` and an auxiliary vector `y`, that will contain the local portions of `A` and `x` in each process. Both `B` and `y` have `k=N/p` rows, but in order to simplify they have been dimensioned to `N` rows since the value of `k` is unknown at compile time (an efficient solution in terms of memory would allocate these variables with `malloc`).

```
int i, j, k, rank, p;
int A[N][N], B[N][N], v[N], x[N], y[N];

MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank == 0) read(A,v);
k = N/p;
MPI_Scatter(A, k*N, MPI_INT, B, k*N, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(v, N, MPI_INT, 0, MPI_COMM_WORLD);
for (i=0;i<k;i++) {
  y[i]=0;
  for (j=0;j<N;j++) y[i] += B[i][j]*v[j];
}
MPI_Gather(y, k, MPI_INT, x, k, MPI_INT, 0, MPI_COMM_WORLD);
if (rank == 0) write(x);
```

Question 2–2

The following fragment of a code computes the Frobenius norm of a square matrix obtained using the function `readmat`.

```
int i, j;
double s, norm, A[N][N];
readmat(A);
s = 0.0;
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) s += A[i][j]*A[i][j];
}
```

```
norm = sqrt(s);
printf("norm=%f\n",norm);
```

Implement a parallel program using MPI that computes the Frobenius norm from a matrix `A` read by process P_0. This process will cyclically distribute the matrix and it will finally collect the result `s`, which will be printed on the screen.

N.B.: Assume that `N` is an exact multiple of the number of processes.

Solution: We use an auxiliary matrix `B` to store the local part of `A` in each process (it will only use the first `k` rows). For distributing the matrix, `k` scatter operations are performed, one for each block with `p` rows.

```
int i, j, k, rank, p;
double s, norm, A[N][N], B[N][N];

MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
k = N/p;
if (rank == 0) read(A);
for (i=0;i<k;i++) {
  MPI_Scatter(&A[i*p][0],N, MPI_DOUBLE, &B[i][0], N, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
}
s=0;
for (i=0;i<k;i++) {
  for (j=0;j<N;j++) s += B[i][j]*B[i][j];
}
MPI_Reduce(&s, &norm, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
if (rank == 0) {
  norm = sqrt(norm);
  printf("norm=%f\n",norm);
}
```

Question 2–3

The following program has to be parallelised using MPI.

```
double *read_data(char *name, int *n) {
  ...  /* Read from the data file */
  /* It returns a pointer to the data and the number of elements in *n */
}

double processes(double x) {
  ...  /* Computational intensive function that performs a task depending on  x */
}

int main() {
  int i,n;
  double *a,res;

  a = read_data("data.txt",&n);
  res = 0.0;
  for (i=0; i<n; i++)
```

```
    res += process(a[i]);

  printf("Result: %.2f\n",res);
  free(a);
  return 0;
}
```

Notes:

- Only process 0 should call function `read_data` (only it will read the file).
- Only process 0 should show the results.
- The `n` computations should be split among the processes available using a block-wise distribution. Process 0 should send each process its part of `a` and it will collect its partial result `res`. Assume that `n` is exactly divided by the number of processes.

(a) Implement a version using point-to-point communication.

Solution:

```
int main(int argc,char *argv[])
{
  int i,n,p,np,nb;
  double *a,res,aux;
  MPI_Status stat;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&p);
  MPI_Comm_size(MPI_COMM_WORLD,&np);

  if (!p) a = read_data("data.txt",&n);

  /* Broadcast the size of the problem(1) */
  if (!p) {
    for (i=1; i<np; i++)
      MPI_Send(&n, 1, MPI_INT, i, 5, MPI_COMM_WORLD);
  } else {
    MPI_Recv(&n, 1, MPI_INT, 0, 5, MPI_COMM_WORLD, &stat);
  }
  nb = n/np;  /* Assuming n is an exact multiple of np */

  if (p) a = (double*) malloc(nb*sizeof(double));

  /* Split the 'a' vector among all the processes (2) */
  if (!p) {
    for (i=1; i<np; i++)
      MPI_Send(&a[i*nb], nb, MPI_DOUBLE, i, 25, MPI_COMM_WORLD);
  } else {
    MPI_Recv(a, nb, MPI_DOUBLE, 0, 25, MPI_COMM_WORLD, &stat);
  }
  res = 0.0;
  for (i=0; i<nb; i++)
    res += process(a[i]);
```

```
    /* Collection of results (3) */
    if (!p) {
      for (i=1; i<np; i++)
        MPI_Recv(&aux, 1, MPI_DOUBLE, i, 52, MPI_COMM_WORLD, &stat);
        res += aux;
      }
    } else {
      MPI_Send(&res, 1, MPI_DOUBLE, 0, 52, MPI_COMM_WORLD);
    }
    if (!p) printf("Result : %.2f\n",res);
    free(a);

    MPI_Finalize();
    return 0;
  }
```

(b) Implement a version using collective communication:

Solution:
Just change the blocks outlined with (1), (2) and (3) by:

```
    /* Broadcast the size of the problem (1) */
    MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

    /* Distribute the a among the different processes (2) */
    MPI_Scatter(a, nb, MPI_DOUBLE, b, nb, MPI_DOUBLE, 0, MPI_COMM_WORLD);

    /* Collect the results (3) */
    aux = res;
    MPI_Reduce(&aux, &res, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

In the scatter operation an auxiliary variable `b` has been used, since using the same buffer to send and receive is not allowed (except in the special case that `MPI_IN_PLACE` is used). So additionally `a` should be replaced by `b` in the calls to `malloc`, `free` and `process`.

Question 2–4

Implement an MPI program that will play the following game:

1. Each process randomly chooses a number and communicates it to the rest.
2. If all the processes have chosen the same number, the game finishes.
3. If not, the process is repeated (we return to the first step). After 1000 repetitions, the game ends with an error.
4. At the end, we should show on the screen (only once), how many times the process has been repeated until every process have chosen the same number.

The following function returns a random number:

```
int get_a_number(); /* returns a random number */
```

Use MPI collective communication operations when possible.

Solution:

```
int p,np;
int num,*vnum,cont,same,i;

MPI_Comm_rank(MPI_COMM_WORLD,&p);
MPI_Comm_size(MPI_COMM_WORLD,&np);
vnum = (int*) malloc(np*sizeof(int));
cont = 0;
do {
  cont++;
  num = get_a_number();
  MPI_Allgather(&num, 1, MPI_INT, vnum, 1, MPI_INT, MPI_COMM_WORLD);
  same = 0;
  for (i=0; i<np; i++) {
    if (vnum[i]==num) same++;
  }
} while (same!=np && cont<1000);

if (!p) {
  if (same==np)
    printf("Every process has chosen the same number %d.\n",cont);
  else
    printf("ERROR: After 1000 iterations, no coincidence appeared.\n");
}
free(vnum);
```

Question 2–5

The exercise aims at implementing a parallel generator of random numbers. Given p MPI processes, all processes will generate a sequence of numbers until P_0 indicates them to stop. In this moment, each process will send P_0 the last number generated and P_0 will combine all these numbers with the one generated by it. The pseudo-code would be like this:

```
n = initial(id)
if id=0
  for i=1 to 100
    n = next(n)
  end
  sends ending message to 1..np-1
  receive m[k] from process k for k=1..np-1
  n = combine(n,m[k]) for k=1..np-1
else
  n = initial
  while !receive message from 0
    n = next(n)
  end
  send n to 0
end
```

Implement using MPI an asynchronous communication scheme for this algorithm, using `MPI_Irecv` and `MPI_Test`. The reception of the results can be performed using a collective operation.

Solution: In the signalling message it is not necessary to send any data, so the buffer will be a null pointer and the length is 0.

```
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &np);
n = initial(id);
if (id==0) {
  for (i=1;i<=100;i++) n = next(n);
  for (k=1;k<np;k++) {
    MPI_Send(NULL, 0, MPI_INT, k, 1, MPI_COMM_WORLD);
  }
} else {
  MPI_Irecv(NULL, 0, MPI_INT, 0, 1, MPI_COMM_WORLD, &req);
  do {
    n = next(n);
    MPI_Test(&req, &flag, MPI_STATUS_IGNORE);
  } while (!flag);
}
MPI_Gather(&n, 1, MPI_DOUBLE, m, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
if (id==0) {
  for (k=1;k<np;k++) n = combine(n,m[k]);
}
```

Question 2–6

Given the following fragment of a program that computes an approximate value for π:

```
double rx, ry, computed_pi;
long int i, points, hits;
unsigned int seed = 1234;

hits = 0;
for (i=0; i<points; i++) {
  rx = (double)rand_r(&seed)/RAND_MAX;
  ry = (double)rand_r(&seed)/RAND_MAX;
  if ((rx-0.5)*(rx-0.5)+(ry-0.5)*(ry-0.5)<0.25) hits++;
}
computed_pi = 4.0*hits/points;
printf("Computed PI = %.10f\n", computed_pi);
```

Implement an MPI version that implements this computation in parallel.

Solution:

The parallelization is simple since the program is highly parallel. It mainly focuses on correctly using the `MPI_Reduce` function. Each process just has to compute the amount of random numbers it must generate and then generate them counting the ones that lie inside the circle. The seed is multiplied by the process id so that the sequence of random numbers is different in each process.

```
double rx, ry, computed_pi;
long int i, points_per_proc, points, hitproc, hits;
int myproc, nprocs;
```

```
MPI_Comm_rank(MPI_COMM_WORLD, &myproc);
MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

seed = myproc*1234;
points_per_proc = points/nprocs;
hitproc = 0;
for (i=0; i<points_per_proc; i++) {
  rx = (double)rand_r(&seed)/RAND_MAX;
  ry = (double)rand_r(&seed)/RAND_MAX;
  if ((rx-0.5)*(rx-0.5)+(ry-0.5)*(ry-0.5)<0.25) hitproc++;
}

MPI_Reduce(&hitproc, &hits, 1, MPI_LONG, MPI_SUM, 0, MPI_COMM_WORLD);

if (!myproc) {
  computed_pi = 4.0*hits/points;
  printf("Computed PI = %.10f\n", computed_pi);
}
```

Question 2–7

The ∞-norm of a matrix is defined as the maximum sum of the absolute values of the elements in each row: $\max_{i=1..n} \left\{ \sum_{j=0}^{m-1} |a_{i,j}| \right\}$. The following sequential code implements such operation for a square matrix.

```
#include <math.h>
#define N 800

double infNorm(double A[][N]) {
  int i,j;
  double s,nrm=0.0;

  for (i=0; i<N; i++) {
    s=0.0;
    for (j=0; j<N; j++)
      s+=fabs(A[i][j]);
    if (s>nrm)
      nrm=s;
  }
  return nrm;
}
```

(a) Implement an MPI parallel version using collective operations whenever possible. Assume that the size of the problem is an exact multiple of the number of processes. The matrix is stored initially in P_0 and the result must also end in P_0.

Note: suggest using the following header for the parallel function, where `Alocal` is a matrix that has already been allocated in memory and can be used by the function to store the local part of matrix `A`.

```
double infNormPar(double A[][N], double ALocal[][N])
```

Solution:

```
#include <mpi.h>
#include <math.h>
#define N 800

double infNormPar(double A[][N], double ALocal[][N]) {
  int i,j,p;
  double s,nrm,nrml=0.0;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Scatter(A, N*N/p, MPI_DOUBLE, ALocal, N*N/p, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
  for (i=0; i<N/p; i++) {
    s=0.0;
    for (j=0; j<N; j++)
      s+=fabs(Alocal[i][j]);
    if (s>nrml)
      nrml=s;
  }
  MPI_Reduce(&nrml, &nrm, 1, MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
  return nrm;
}
```

(b) Obtain the computational and communication cost for the parallel algorithm. Assume that the `fabs` operation has a negligible cost, as well as in the case of comparisons.

Solution: We denote by n the problem size (`N`). The cost includes three stages:

- Cost of scatter: $(p-1)\cdot\left(t_s + n\cdot\frac{n}{p}\cdot t_w\right)$
- Cost of processing $\frac{n}{p}$ rows: $\frac{n}{p}\cdot n = \frac{n^2}{p}$
- Cost of reduction (trivial implementation): $(p-1)\cdot(t_s+t_w)$

Therefore, the total cost is approximately: $2\cdot p\cdot t_s + n^2\cdot t_w + \frac{n^2}{p}$

(c) Obtain the speedup and efficiency when the problem size tends to infinity.

Solution: The computational cost of the sequential version is approximately n^2. Therefore, the speed-up is $S(n,p) = \frac{n^2}{2\cdot p\cdot t_s + n^2\cdot t_w + \frac{n^2}{p}}$ and the efficiency $E(n,p) = \frac{n^2}{2\cdot p^2\cdot t_s + p\cdot n^2\cdot t_w + n^2}$.

The asymptotic values of speed-up and efficiency when the problem size tends to infinity are the following:

$$\lim_{n\to\infty} S(n,p) = \lim_{n\to\infty} \frac{1}{2\cdot\frac{p}{n^2}\cdot t_s + t_w + \frac{1}{p}} = \frac{p}{p\cdot t_w + 1}$$

$$\lim_{n\to\infty} E(n,p) = \lim_{n\to\infty} \frac{1}{2\cdot\frac{p^2}{n^2}\cdot t_s + p\cdot t_w + 1} = \frac{1}{p\cdot t_w + 1}$$

Question 2–8

Given the following code:

```
for (i=0; i<m; i++) {
  for (j=0; j<n; j++) {
    w[j] = process(j, n, v);
```

```
  }
  for (j=0; j<n; j++) {
    v[j] = w[j];
  }
}
```

where function `process` has the following prototype:

```
double process(int j, int n, double *v);
```

being all input arguments.

(a) Indicate its theoretical cost (in flops) assuming that the cost of function `process` is $2n$ flops.

Solution: Sequential cost: $2n^2m$ flops.

(b) Parallelize such code in MPI and justify your answer. We assume that n is the dimension of vectors `v` and `w`, but also the number of MPI processes. The variable `p` contains the process identifier. The process `p=0` is the only one that has the initial value of vector `v`. It will be taken into consideration the most efficient way of doing the parallelization. This consists in using the appropriate MPI routines in such as way that its number is minimum.

Solution: In the fist place, process 0 broadcasts the vector to the rest of processes since function `process` needs this vector. The outer loop cannot be parallelized. We therefore parallelize inner loops. In an iteration (`i`), process `p` is in charge of executing function `process` and store the result in variable `a`. The update of vector `v` in the second loop corresponds to a gather operation in which each process sends the computed data stored in variable `a` to all the rest.

```
MPI_Bcast(v, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0; i<m; i++) {
  double a = process(p, n, v);
  MPI_Allgather(&a, 1, MPI_DOUBLE, v, 1, MPI_DOUBLE, MPI_COMM_WORLD);
}
```

(c) Indicate the communication cost assuming that the nodes are connected in a bus topology.

Solution: The cost of a broadcast of n elements in a bus is $\beta+n\tau$. The cost of the gather operation of one element in each process received in all processes is $n(\beta + \tau)$. The total cost, taking into account the number of iterations of the outer loop, is

$$(\beta + n\tau) + mn(\beta + \tau) \ .$$

(d) Indicate the attainable efficiency taking into account that both m and n are large.

Solution: The speedup S_n is computed as the ratio between the sequential and parallel time (assuming n processes):

$$S_n = \frac{2mn^2}{2mn + mn(\beta + \tau)} = \frac{2n}{2 + \beta + \tau} \ ,$$

where we have taken into account that the cost of the diffusion is negligible (for large values of m and n). Therefore, the efficiency E for n processes is

$$E_n = \frac{S_n}{n} = \frac{2}{2 + \beta + \tau} \ .$$

Question 2–9

Next program computes the number of occurrences for a specific value in a matrix.

```
#include <stdio.h>
#define DIM 1000

void read(double A[DIM][DIM], double *x)
{ ... }

int main(int argc, char *argv[])
{
  double A[DIM][DIM], x;
  int i,j,cont;

  read(A,&x);
  cont=0;
  for (i=0; i<DIM; i++)
    for (j=0; j<DIM; j++)
      if (A[i][j]==x) cont++;
  printf("%d occurrences\n", cont);
  return 0;
}
```

(a) Implement an MPI parallel version of the program above, using collective communication operations if possible and convenient. The function `read` should be called only by process 0. It can be assumed that `DIM` is exactly divisible by the number of processes. Note: Write the whole program including the declaration of variables and the necessary calls to initialize and finalize MPI.

Solution: Matrix `A` is distributed by blocks of consecutive rows among the processes.

```
...
int main(int argc, char *argv[])
{
  double A[DIM][DIM], Aloc[DIM][DIM], x;
  int i, j, cont, cont_loc;
  int p, rank;

  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  if (rank==0) read(A,&x);

  /* Data Distribution */
  MPI_Bcast(&x, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(A, DIM/p*DIM, MPI_DOUBLE, Aloc, DIM/p*DIM, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);

  /* Local computation */
  cont_loc=0;
  for (i=0; i<DIM/p; i++)
    for (j=0; j<DIM; j++)
      if (Aloc[i][j]==x) cont_loc++;

  /* Collection global result in P0 */
```

```
  MPI_Reduce(&cont_loc, &cont, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank==0) printf("%d occurrences\n", cont);

  MPI_Finalize();
  return 0;
}
```

(b) Obtain the parallel execution time, assuming that the cost of comparing two real numbers is 1 flop. Note: for the communication cost, consider a simple implementation of the collective operations used.

Solution: Considering n as the dimension of the matrix (`DIM`).
The parallel cost is the sum of the arithmetic cost plus the communication cost. The former is:

$$t_a(n,p) = \sum_{i=0}^{n/p-1} \sum_{j=0}^{n-1} 1 = n^2/p$$

Regarding the communications, we will assume that the broadcast implies sending $p-1$ messages from the root process to any other one. A similar behaviour is assumed for scatter and reduction. Therefore, the communication cost will be:

$$t_c(n,p) = 2(p-1)(t_s + t_w) + (p-1)(t_s + \frac{n^2}{p} t_w) \approx 3pt_s + (n^2 + 2p)t_w$$

Then, the parallel cost will be:

$$t(n,p) \approx n^2/p + 3pt_s + (n^2 + 2p)t_w$$

Question 2–10

(a) Implement a function that sums two square matrices `a` and `b` using MPI collective communication primitives. The function will store the result in `a`. Matrices `a` and `b` are initially stored in the memory of process P_0 and the final result should also be stored in P_0. We assume that the number of rows from the matrices (`N`, constant) is an exact multiple of the number of processes. The header for the function will be:

```
void sum_mat(double a[N][N],double b[N][N])
```

Solution:

```
void sum_mat(double a[N][N],double b[N][N])
{
   int i,j,np,tb;
   double al[N][N],bl[N][N];
   MPI_Comm_size(MPI_COMM_WORLD, &np);
   tb=N/np;
   MPI_Scatter(a, tb*N, MPI_DOUBLE, al, tb*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   MPI_Scatter(b, tb*N, MPI_DOUBLE, bl, tb*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
   for(i=0;i<tb;i++)
      for(j=0;j<N;j++)
         al[i][j]+=bl[i][j];
   MPI_Gather(al, tb*N, MPI_DOUBLE, a, tb*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

(b) Obtain the parallel time, the speed-up and the efficiency of the implementation proposed in the previous part. Describe how the cost of the collective operations is computed (number of messages, ant their length). You can assume a simple implementation of such collective operations.

Solution:

Sequential Time: N^2 flops.

Parallel Time assuming we have p processes:

The cost of distributing a square matrix of order N using `MPI_Scatter` (assuming that a trivial algorithm is used) can be seen as sending $p-1$ messages of equal size: $\frac{N}{p}N = \frac{N^2}{p}$ elements. Therefore, the cost of distributing `a` and `b` will be

$$2(p-1)\left(t_s + \frac{N^2}{p}t_w\right) \approx 2pt_s + 2N^2t_w$$

where in order to simplify the expression we have supposed a large value of p.

The parallel cost for concurrently computing the sum of the local chunks of the matrices `a` and `b` is

$$\sum_{i=0}^{\frac{N}{p}-1}\sum_{j=0}^{N-1} = \sum_{i=0}^{\frac{N}{p}-1} N = \frac{N^2}{p} \quad \text{flops.}$$

The cost of the collection of the resulting matrix `a` by P_0 (`MPI_Gather`) can be considered as sending one message from each process P_i $(i > 0)$ to process P_0 with a size of $\frac{N}{p}N = \frac{N^2}{p}$ elements. Therefore, the cost will be:

$$(p-1)\left(t_s + \frac{N^2}{p}t_w\right) \approx pt_s + N^2t_w$$

Summing the three previous times, we end up with a parallel cost of:

$$3pt_s + 3N^2t_w + \frac{N^2}{p}$$

Speed-up:

$$S(N,p) = \frac{N^2}{3pt_s + 3N^2t_w + \frac{N^2}{p}}$$

Efficiency:

$$E(N,p) = \frac{S(N,p)}{p} = \frac{N^2}{3p^2t_s + 3pN^2t_w + N^2}$$

Question 2–11

The following function computes the scalar product of two vectors:

```
double scalarprod(double X[], double Y[], int n) {
  double prod=0.0;
  int i;
  for (i=0;i<n;i++)
    prod += X[i]*Y[i];
  return prod;
}
```

(a) Implement a function to perform the scalar product in parallel by means of MPI, using collective operations whenever possible. The data are supposed to be available in process P_0 and the result

must also be left in P_0 (the function's return value need only be correct in P_0). It is allowed to assume that the problem size `n` is exactly divisible by the number of processes.

Note: we next show the header of the function to be implemented, including the declaration of the local vectors (assume that `MAX` is sufficiently large for any value of n and the number of processes).

```
double pscalarprod(double X[], double Y[], int n)
{
  double Xlcl[MAX], Ylcl[MAX];
```

Solution:

```
double pscalarprod(double X[], double Y[], int n)
{
  double Xlcl[MAX], Ylcl[MAX];
  double prod=0.0, prodf;
  int i, p, nb;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  nb = n/p;
  MPI_Scatter(X, nb, MPI_DOUBLE, Xlcl, nb, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Scatter(Y, nb, MPI_DOUBLE, Ylcl, nb, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  for (i=0;i<nb;i++)
    prod += Xlcl[i]*Ylcl[i];
  MPI_Reduce(&prod, &prodf, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  return prodf;
}
```

(b) Compute the speed-up. If for a sufficiently large message size, the sending time per element was equivalent to 0.1 flops, which would be the maximum speed-up that could be attained when the problem size tends to infinity and for a sufficiently large number of processes?

Solution: $t(n) = \sum_{i=0}^{n-1} 2 = 2n$ Flops

$$t(n,p) = 2(p-1)(t_s + \frac{n}{p}t_w) + \frac{2n}{p} + (p-1)(t_s + t_w) + p - 1 \approx 3pt_s + (2n+p)t_w + \frac{2n}{p}$$

$$S(n,p) = \frac{t(n)}{t(n,p)} = \frac{2n}{3p \cdot t_s + (2n+p)t_w + \frac{2n}{p}}$$

$$\lim_{n\to\infty} S(n,p) = \frac{2}{2 \cdot t_w + \frac{2}{p}} = \frac{2p}{2p \cdot t_w + 2}$$

If $t_w = 0.1$ Flops, then $S(n,p)$ would be limited by $\frac{2p}{0.2p} = 10$.

(c) Modify the previous code so that the return value is correct in all processes.

Solution: The call to `MPI_Reduce` must be changed by the following line:

```
MPI_Allreduce(&prod, &prodf, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
```

Question 2–12

Given the sequential code:

```
int i, j;
double A[N][N];
```

```
for (i=0;i<N;i++)
    for(j=0;j<N;j++)
        A[i][j]= A[i][j]*A[i][j];
```

(a) Implement an equivalent parallel version using MPI, taking into account the following aspects:

- Process P_0 initially obtains matrix `A`, performing a call `read(A)`, where `read` is a function already implemented.
- Matrix `A` must be distributed by blocks of rows among all processes.
- Finally P_0 must contain the resulting matrix `A`.
- Use collective communication whenever possible.

We assume that N is divisible by the number of processes and that the declaration of the used matrices is

```
double A[N][N], B[N][N];  /* B: distributed matrix */
```

Solution:

```
int i, j, rank, p, bs;
double A[N][N], B[N][N];

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
if (rank==0)
    read(A);
bs = N/p;
MPI_Scatter(A,bs*N,MPI_DOUBLE,B,bs*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
for (i=0;i<bs;i++)
    for(j=0;j<N;j++)
        B[i][j]= B[i][j]*B[i][j];
MPI_Gather(B,bs*N,MPI_DOUBLE,A,bs*N,MPI_DOUBLE,0,MPI_COMM_WORLD);
```

(b) Compute the speedup and efficiency.

Solution: The sequential computational cost is $t(N) = N^2$ flops.
Since the scatter (`MPI_Scatter`) or gather (`MPI_Gather`) of a matrix of order N among p processes implies sending/receiving $p-1$ messages of length $\frac{N^2}{p}$, the communication time is $t_c = 2(p-1)\left(t_s + \frac{N^2}{p}t_w\right)$. The parallel arithmetic cost is $\frac{N^2}{p}$. Therefore, the total parallel time is:

$$t(N,p) = 2(p-1)\left(t_s + \frac{N^2}{p}t_w\right) + \frac{N^2}{p}.$$

Then the speedup is equal to

$$S(N,p) = \frac{t(N)}{t(N,p)} = \frac{N^2}{2(p-1)\left(t_s + \frac{N^2}{p}t_w\right) + \frac{N^2}{p}}$$

and the efficiency

$$E(N,p) = \frac{S(N,p)}{p} = \frac{N^2}{2p(p-1)\left(t_s + \frac{N^2}{p}t_w\right) + N^2}$$

Question 2–13

Next program reads a square matrix A of dimension N and computes a vector v of N elements. Element

i in this vector contains the sum of all the elements of the $i-th$ row in A. Then, the program prints vector v.

```
int main(int argc, char *argv[])
{
  int i,j;
  double A[N][N],v[N];
  read_mat(A);
  for (i=0;i<N;i++) {
    v[i] = 0.0;
    for (j=0;j<N;j++)
      v[i] += A[i][j];
  }
  write_vec(v);
  return 0;
}
```

(a) Implement an MPI parallel program, using collective communication primitives whenever possible, that will perform the same computations as the sequential code. Take into account the next points:

- Process P_0 reads matrix A.
- P_0 distributes matrix A among all the processes.
- Each process computes its local part of v.
- P_0 composes vector v by collecting the local parts from each process.
- P_0 writes vector v.

N.B.: You can assume that N is an exact multiple of the number of processes.

Solution:

```
int main(int argc, char *argv[])
{
  int i,j,id,p,tb;
  double A[N][N],Al[N][N],v[N],vl[N];
  MPI_Init(&argc, &argv);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Comm_size(MPI_COMM_WORLD, &p);
  if (id==0) read_mat(A);
  tb = N/p;
  MPI_Scatter(A, tb*N, MPI_DOUBLE, Al, tb*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  for (i=0;i<tb;i++) {
    vl[i] = 0.0;
    for (j=0;j<N;j++)
      vl[i] += Al[i][j];
  }
  MPI_Gather(vl, tb, MPI_DOUBLE, v, tb, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  if (id==0) write_vec(v);
  MPI_Finalize();
  return 0;
}
```

(b) Obtain the sequential and parallel time, ignoring the cost of the read and write functions. Indicate separately the cost of each one of the collective operations.

Solution: Sequential time:

$$t(N) = \sum_{i=0}^{N-1} \sum_{j=0}^{N-1} 1 = N^2 \text{ flops}$$

Parallel arithmetic time with p processes:

$$t_{arit}(N,p) = \sum_{i=0}^{N/p-1} \sum_{j=0}^{N-1} 1 = \frac{N^2}{p} \text{ flops}$$

Parallel communication time with p processes:

- Scattering of matrix A;

$$(p-1)\left(t_s + \frac{N^2}{p} t_w\right)$$

- Gathering of vector v:

$$(p-1)\left(t_s + \frac{N}{p} t_w\right)$$

Therefore, the parallel time is:

$$t(N,p) = \frac{N^2}{p} \text{ flops} + (p-1)\left(2t_s + \frac{N}{p}(N+1)t_w\right) \approx \frac{N^2}{p} \text{ flops} + 2pt_s + N^2 t_w$$

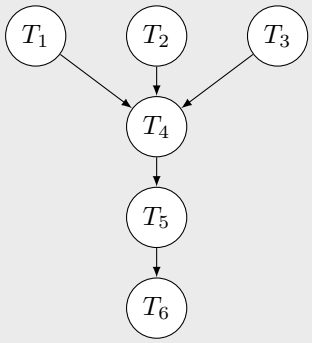
Question 2–14

Function `example` executes several tasks (`T1`–`T6`). The cost of the functions `T1`, `T2` and `T3` is $7n$, and the cost of the functions `T5` and `T6` is n, being n a constant value.

```
double example(int val[3])
{
  double a,b,c,d,e,f;
  a = T1(val[0]);
  b = T2(val[1]);
  c = T3(val[2]);
  d = a+b+c;       /* T4 */
  e = T5(val[2],d);
  f = T6(val[0],val[1],e);
  return f;
}
```

(a) Draw the dependency graph for the function and compute the sequential cost.

Solution: Next figure shows the dependency graph.



The sequential cost is: $t(n) = 7n + 7n + 7n + 2 + n + n \approx 23n$

(b) Implement a parallel version using MPI, assuming that there are three processes. All the processes will call the function with the same value for `val` (you do not need to communicate it). The return value only needs to be correct in process 0 (the value returned in the rest of the processes is irrelevant).
N.B.: Only use collective communication primitives.

Solution:
The first three tasks are assigned to different processes, for load balancing. The other three tasks must be executed sequentially, due to their dependencies. We allocate these three tasks to P_0, as this process is the one that needs the final correct return value for the function.

```
double example(int val[3])
{
  double a,b,c,d,e,f,operand;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  switch (rank) {
    case 0: a = T1(val[0]); operand = a; break;
    case 1: b = T2(val[1]); operand = b; break;
    case 2: c = T3(val[2]); operand = c; break;
  }
  MPI_Reduce(&operand, &d, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank==0) {
    e = T5(val[2],d);
    f = T6(val[0],val[1],e);
  }
  return f;
}
```

(c) Compute the parallel execution time (arithmetic and communications) and the Speed-Up with three processes. Compute also the asymptotic Speed-Up (that is, the limit when n tends to infinity).

Solution: The parallel time is computed from the cost associated to the critical path of the dependency graph, corresponding to $T_1 - T_4 - T_5 - T_6$, for example. For computing the communication cost, we assume that collective reduction internally will send just two messages, each one with 1 `double`. In the reduction operation, two additional flops will be needed, but they have not been included in the expressions.

$$t(n, 3) = t_{arit}(n, 3) + t_{comm}(n, 3)$$

$$t_{arit}(n,3) = 7n + n + n \approx 9n$$
$$t_{comm}(n,3) = 2 \cdot (t_s + t_w)$$
$$t(n,3) \approx 9n + 2t_s + 2t_w$$

The Speed-Up will be

$$S(n,3) = \frac{t(n)}{t(n,3)} \approx \frac{23n}{9n + 2t_s + 2t_w}$$

As the communication cost in this case is very low, the asymptotic (as n grows) Speed-Up tends to the value $S(n,3) \to \frac{23n}{9n} = 2,56$.

Question 2–15

Given the next sequential function:

```
int count(double v[], int n)
{
  int i, cont=0;
  double mean=0;

  for (i=0;i<n;i++)
    mean += v[i];
  mean = mean/n;

  for (i=0;i<n;i++)
    if (v[i]>mean/2.0 && v[i]<mean*2.0)
      cont++;

  return cont;
}
```

(a) Implement a parallel version using MPI, assuming that vector `v` is initially only in process 0, and that the result returned by the function needs to be correct only in process 0. You should distribute the data for achieving a balanced distribution of the processing. N.B.: You can assume that `n` is an exact multiple of the number of processes.

Solution:

```
int count(double v[], int n)
{
  int i, cont, cont_loc=0, p;
  double mean, sum_loc=0;
  double *vloc;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  vloc = (double*) malloc(n/p*sizeof(double));

  MPI_Scatter(v, n/p, MPI_DOUBLE, vloc, n/p, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  for (i=0;i<n/p;i++)
    sum_loc += vloc[i];

  MPI_Allreduce(&sum_loc, &mean, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  mean = mean/n;
```

```
    for (i=0;i<n/p;i++)
      if (vloc[i]>mean/2 && vloc[i]<mean*2)
        cont_loc++;

    MPI_Reduce(&cont_loc, &cont, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
    free(vloc);
    return cont;
}
```

(b) Compute the execution time of the parallel version in the previous part, as well as the asympthotic value of the Speed-up when `n` tends to infinite. If you have used collective operations, indicate the cost that you have considered for each one of them.

Solution:

- Scatter: Process 0 sends a message with n/p elements to each one of the rest of processes.

$$(p-1)(t_s + \frac{n}{p}t_w)$$

- Reduce: Process 0 receives a message with one element from each one of the other processes, and sums up the values.

$$(p-1)(t_s + t_w) + (p-1)$$

- Allreduce: it can be simply considered as a *reduce* operation over process 0, followed by a *broadcast* of the result. For the broadcast, we will assume that process 0 sends a message of 1 element to each one of the other processes.

$$2(p-1)(t_s + t_w) + (p-1)$$

- Computation loops:

$$\sum_{i=0}^{\frac{n}{p}-1} 1 + \sum_{i=0}^{\frac{n}{p}-1} 2 = \frac{3n}{p}$$

The parallel execution time is the sum of the above terms:

$$t(n,p) \approx 4pt_s + (n+3p)t_w + 2p + \frac{3n}{p}$$

On the other side, the sequential time is:

$$t(n) \approx \sum_{i=0}^{n-1} 1 + \sum_{i=0}^{n-1} 2 = 3n$$

And the Speed up is:

$$S(n,p) = \frac{3n}{4pt_s + (n+3p)t_w + 2p + \frac{3n}{p}}$$

$$\lim_{n\to\infty} S(n,p) = \frac{3}{t_w + 3/p}$$

Question 2–16

The following sequential program makes some calculations on a square matrix `A`.

```
#define N ...
int i, j;
double A[N][N], sum[N], fact, max;
...
for (i=0;i<N;i++) {
  sum[i] = 0.0;
  for (j=0;j<N;j++) sum[i] += A[i][j]*A[i][j];
}
fact = 1.0/sum[0];
for (i=0;i<N;i++) sum[i] *= fact;

max = 0.0;
for (i=0;i<N;i++) {
  if (sum[i]>max) max = sum[i];
}
for (i=0;i<N;i++) {
  for (j=0;j<N;j++) A[i][j] *= max;
}
```

(a) Parallelize the code by means of MPI supposing that each process has already stored `k=N/p` consecutive rows of the matrix, being `p` the number of processes (can assume that `N` can be divided by `p`). These rows occupy the first positions of the local matrix, that is, among the rows `0` and `*k-1` of the variable `A`. Note: Use collective communication primitives whenever possible.

Solution: Each process calculates a part of the vector of sums `sum`, storing the values in the first `k` positions. The value of `fact` is calculated in P_0 (who is the owner of row 0 of the matrix) and broadcasts to the rest of processes. In the last part of the algorithm, each process calculates the local maximum and makes a reduction whose result (`max`) is necessary in all the processes.

```
#define N ...
int i, j, k, rank, p;
double A[N][N], sum[N], fact, max, maxloc;
...
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &p);
k = N/p;
for (i=0;i<k;i++) {
  sum[i] = 0.0;
  for (j=0;j<N;j++) sum[i] += A[i][j]*A[i][j];
}
if (rank==0) fact = 1.0/sum[0];
MPI_Bcast(&fact, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
for (i=0;i<k;i++) sum[i] *= fact;

maxloc = 0.0;
for (i=0;i<k;i++) {
  if (sum[i]>maxloc) maxloc = sum[i];
}
MPI_Allreduce(&maxloc, &max, 1, MPI_DOUBLE, MPI_MAX, MPI_COMM_WORLD);
for (i=0;i<k;i++) {
  for (j=0;j<N;j++) A[i][j] *= max;
}
```

(b) Write the code to make the necessary communication after the previous calculation so that the

complete matrix remains stored in process 0 in the variable `Aglobal[N][N]`.

Solution:

```
MPI_Gather(A, k*N, MPI_DOUBLE, Aglobal, k*N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Question 2–17

Next piece of code implements the operation $C = aA + bB$, where A, B and C are matrices of size $M \times N$ and a and b are real numbers.

```
int main(int argc, char *argv[]) {
  int i, j;
  double a, b, A[M][N], B[M][N], C[M][N];
  ReadOperands(A, B, &a, &b);
  for (i=0; i<M; i++) {
    for (j=0; j<N; j++) {
      C[i][j] = a*A[i][j] + b*B[i][j];
    }
  }
  WriteMatrix(C);
  return 0;
}
```

Write a parallel version using MPI and using collective operations, assuming that:

- P_0 will get matrices A and B, as well as real numbers a and b, by calling function `ReadOperands`.
- Only P_0 should have the whole resulting matrix C, and it will be the process that calls function `WriteMatrix`.
- M is an exact multiple of the number of processes.
- Matrices A and B must be distributed cyclically by rows among the processes involved, which will execute in parallel the operation.

Solution:

```
int main(int argc, char *argv[]) {
  int i, j, k, p, myid;
  double a, b, A[M][N], B[M][N], C[M][N];
  double Alocal[M][N], Blocal[M][N], Clocal[M][N];
  MPI_Init(&argc, &argv);
  MPI_Comm_size(MPI_COMM_WORLD ,&p);
  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
  if (myid==0) ReadOperands(A, B, &a, &b);
  MPI_Bcast(&a, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(&b, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  k=M/p;
  for (i=0; i<k; i++) {
    MPI_Scatter(&A[i*p][0], N, MPI_DOUBLE, &Alocal[i][0], N, MPI_DOUBLE, 0,
                MPI_COMM_WORLD);
    MPI_Scatter(&B[i*p][0], N, MPI_DOUBLE, &Blocal[i][0], N, MPI_DOUBLE, 0,
                MPI_COMM_WORLD);
  }
  for (i=0; i<k; i++) {
```

```
      for (j=0; j<N; j++) {
        Clocal[i][j] = a*Alocal[i][j] + b*Blocal[i][j];
      }
    }
    for (i=0; i<k; i++) {
      MPI_Gather(&Clocal[i][0], N, MPI_DOUBLE, &C[i*p][0], N, MPI_DOUBLE, 0,
                 MPI_COMM_WORLD);
    }
    if (myid==0) WriteMatrix(C);
    MPI_Finalize();
    return 0;
  }
```

Question 2–18

Given a matrix `A` with `M` rows and `N` columns, the next function returns a vector `sup` with the number of elements in each row that are larger than the mean.

```
void func(double A[M][N], int sup[M]) {
   int i, j;
   double mean = 0;
   /* Computes the mean of matrix A */
   for (i=0; i<M; i++)
      for (j=0; j<N; j++)
         mean += A[i][j];
   mean = mean/(M*N);
   /* Counts the number of elements > mean in each row */
   for (i=0; i<M; i++) {
      sup[i] = 0;
      for (j=0; j<N; j++)
         if (A[i][j]>mean) sup[i]++;
   }
}
```

Write a parallel version of the previous function using collective communication MPI calls whenever possible. Take into account that matrix `A` is initially stored in process 0 and vector `sup` should also be in process 0 when the function ends. The computations inside the function should be evenly distributed among all the processes. You can assume that the number of rows of the matrix is an exact multiple of the number of processes.

Solution:

```
void funcpar(double A[M][N], int sup[M]) {
   double Aloc[M][N];
   int suploc[M];
   double sumloc, mean;
   int i, j, p;
   MPI_Comm_size(MPI_COMM_WORLD, &p);
   MPI_Scatter(A, M*N/p, MPI_DOUBLE, Aloc, M*N/p, MPI_DOUBLE, 0,
               MPI_COMM_WORLD);
   sumloc=0;
   for (i=0; i<M/p; i++)
      for (j=0; j<N; j++)
```

```
            sumloc += Aloc[i][j];
      MPI_Allreduce(&sumloc, &mean, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
      mean = mean/(M*N);
      for (i=0; i<M/p; i++) {
         suploc[i] = 0;
         for (j=0; j<N; j++)
            if (Aloc[i][j]>mean) suploc[i]++;
      }
      MPI_Gather(suploc, M/p, MPI_INT, sup, M/p, MPI_INT, 0, MPI_COMM_WORLD);
   }
```

Question 2–19

The next program reads a vector from a file, modifies it, and displays a summary on the screen, as well as writing the modified vector on a file.

```
double facto(int m,double x)
{
  int i;
  double p = 1.0;
  for (i=1; i<=m; i++) {
    p = p * x;
    x = x + 1.0;
  }
  return p;
}
```

```
int main(int argc,char *argv[])
{
  int i, n;
  double a = 1.0, v[MAXN];

  n = read_vector(v);
  for (i=0; i<n; i++) {
    v[i] = facto(n,v[i]);
    a = a * v[i];
  }
  printf("Factor alfalfa: %.2f\n",a);
  write_vector(n,v);
  return 0;
}
```

(a) Implement an MPI parallel version using collective communication primitives wherever possible. The input/output to the file and the display on the screen must be done only by process 0. You can assume that the size of the vector (`n`) is an exact multiple of the number of processes. Note that the size of the vector is not known a priori and it is returned by function `read_vector`.

Solution: Function `facto` does not require any change. In the main program, we will implement a block distribution of the vector.

```
int main(int argc,char *argv[])
{
  int i, n, id, np, k;
  double a = 1.0, v[MAXN], vloc[MAXN], total;

  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  if (id==0) n = read_vector(v);
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);
  k = n / np;
  MPI_Scatter(v,k,MPI_DOUBLE,vloc,k,MPI_DOUBLE,0,MPI_COMM_WORLD);
  for (i=0; i<k; i++) {
    vloc[i] = facto(n,vloc[i]);
    a = a * vloc[i];
```

```
    }
    MPI_Gather(vloc,k,MPI_DOUBLE,v,k,MPI_DOUBLE,0,MPI_COMM_WORLD);
    MPI_Reduce(&a,&total,1,MPI_DOUBLE,MPI_PROD,0,MPI_COMM_WORLD);
    if (id==0) {
      printf("Factor alfalfa: %.2f\n",total);
      write_vector(n,v);
    }
    MPI_Finalize();
    return 0;
  }
```

(b) Obtain the sequential execution time.

Solution:

$$t_{\text{facto}}(m) = \sum_{i=1}^{m} 2 = 2m \text{ flops}$$

$$t_1(n) = \sum_{i=0}^{n-1} (1 + t_{\text{facto}}(n)) = \sum_{i=0}^{n-1} (1 + 2n) = n + 2n^2 \approx 2n^2 \text{ flops}$$

(c) Obtain the parallel execution time, clearly indicating the communication cost of each operation. Do not simplify the expressions.

Solution:

$$t_p(n) = t_{\text{Bcast}} + t_{\text{Scatter}} + \frac{n + 2n^2}{p} \text{ flops} + t_{\text{Gather}} + t_{\text{Reduce}}$$

$$t_{\text{Bcast}} = (p-1)(t_s + t_w)$$

$$t_{\text{Scatter}} = t_{\text{Gather}} = (p-1)\left(t_s + \frac{n}{p} t_w\right)$$

$$t_{\text{Reduce}} = (p-1)(t_s + t_w + 1 \text{ flops})$$

Question 2–20

Given the following function, which computes the sum of a vector with `N` elements:

```
double sum(double v[N])
{
  int i;
  double s = 0.0;
  for (i=0; i<N; i++) s += v[i];
  return s;
}
```

(a) Implement an MPI parallel version using only point-to-point communication primitives. The vector `v` is initially in process 0 and the result must be correct in all the processes. You can assume that the size of the vector (`N`) is an exact multiple of the number of processes.

Solution:

```
double sum(double v[N])
{
  int i, id, np, nb, p;
```

```
  double s, sl = 0.0, vl[N];
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  nb = N / np;
  if (id==0) {
    for (p=1; p<np; p++)
      MPI_Send(&v[p*nb],nb,MPI_DOUBLE,p,22,MPI_COMM_WORLD);
    for (i=0; i<nb; i++) sl += v[i];
    s = sl;
    for (p=1; p<np; p++) {
      MPI_Recv(&sl,1,MPI_DOUBLE,p,23,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      s += sl;
    }
    for (p=1; p<np; p++)
      MPI_Send(&s,1,MPI_DOUBLE,p,24,MPI_COMM_WORLD);
  } else {
    MPI_Recv(vl,nb,MPI_DOUBLE,0,22,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    for (i=0; i<nb; i++) sl += vl[i];
    MPI_Send(&sl,1,MPI_DOUBLE,0,23,MPI_COMM_WORLD);
    MPI_Recv(&s,1,MPI_DOUBLE,0,24,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  }
  return s;
}
```

(b) Implement another MPI parallel version of the previous algorithm under the same conditions but using collective communications whenever is more convenient.

Solution:

```
double suma(double v[N])
{
  int i,np,nb;
  double s, sl = 0, vl[N];
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  nb = N / np;
  MPI_Scatter(v,nb,MPI_DOUBLE,vl,nb,MPI_DOUBLE,0,MPI_COMM_WORLD);
  for (i=0; i<nb; i++) sl += vl[i];
  MPI_Allreduce(&sl,&s,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  return s;
}
```

3 Data types

Question 3–1

Given a matrix of integers `A[M][N]`, write the fragment of code necessary to send from P_0 and receive in P_1 the data that are specified in each case, using a single message. If necessary, define an MPI derived data type.

(a) Send the third row of the matrix `A`.

Solution: In C, bi-dimensional arrays are stored by rows, so the separation of elements in the same row is 1. Therefore, in this case it is not necessary to create an MPI type, since the elements are contiguous in memory.

```
int A[M][N];
MPI_Status st;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
  MPI_Send(&A[2][0], N, MPI_INT, 1, 0, MPI_COMM_WORLD);
} else if (rank==1) {
  MPI_Recv(&A[2][0], N, MPI_INT, 0, 0, MPI_COMM_WORLD, &status);
}
```

(b) Send the third column of the matrix `A`.

Solution: The separation of elements from the same column is `N`.

```
int A[M][N];
MPI_Status status;
MPI_Datatype newtype;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Type_vector(M, 1, N, MPI_INT, &newtype);
MPI_Type_commit(&newtype);
if (rank==0) {
  MPI_Send(&A[0][2], 1, newtype, 1, 0, MPI_COMM_WORLD);
} else if (rank==1) {
  MPI_Recv(&A[0][2], 1, newtype, 0, 0, MPI_COMM_WORLD, &status);
}
MPI_Type_free(&newtype);
```

Question 3–2

Given the following fragment of an MPI code:

```
struct Tdata {
   int x;
   int y[N];
   double a[N];
};

void distribute_data(struct Tdata *data, int n, MPI_Comm comm) {
  int p, pr, pr2;
  MPI_Status status;

  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &pr);
  if (pr==0) {
    for (pr2=1; pr2<p; pr2++) {
      MPI_Send(&(data->x), 1, MPI_INT, pr2, 0, comm);
      MPI_Send(&(data->y[0]), n, MPI_INT, pr2, 0, comm);
      MPI_Send(&(data->a[0]), n, MPI_DOUBLE, pr2, 0, comm);
    }
  } else {
    MPI_Recv(&(data->x), 1, MPI_INT, 0, 0, comm, &status);
    MPI_Recv(&(data->y[0]), n, MPI_INT, 0, 0, comm, &status);
    MPI_Recv(&(data->a[0]), n, MPI_DOUBLE, 0, 0, comm, &status);
```

```
  }
}
```

Modify function `distribute_data` to optimize the communications.

(a) Implement a version using MPI derived types, performing one send (to each process) instead of three.

Solution: Since the data to send/receive are of different types, to be able to send them in a single message we must define a data type by means of `MPI_Type_create_struct`.

```
void distribute_data(struct Tdata *data, int n, MPI_Comm comm) {
  int p, pr, pr2;
  MPI_Status status;
  MPI_Datatype Tnew;
  int lengths[]={1,n,n};
  MPI_Datatype types[]={MPI_INT, MPI_INT, MPI_DOUBLE};
  MPI_Aint displs[3];
  MPI_Aint add1, addx, addy, adda;

  MPI_Comm_size(comm, &p);
  MPI_Comm_rank(comm, &pr);

  /* Compute displacements of each component */
  MPI_Get_address(data, &add1);
  MPI_Get_address(&(data->x), &addx);
  MPI_Get_address(&(data->y[0]), &addy);
  MPI_Get_address(&(data->a[0]), &adda);
  displs[0]=addx-add1;
  displs[1]=addy-add1;
  displs[2]=adda-add1;

  MPI_Type_create_struct(3, lengths, displs, types, &Tnew);
  MPI_Type_commit(&Tnew);
  if (pr==0) {
    for (pr2=1; pr2<p; pr2++) {
      MPI_Send(data, 1, Tnew, pr2, 0, comm);
    }
  }
  else {
    MPI_Recv(data, 1, Tnew, 0, 0, comm, &status);
  }
  MPI_Type_free(&Tnew);
}
```

(b) Implement a modification of the previous one using collective communication primitives.

Solution: It would be identical to the previous one, except for the last `if`, which should be changed to the following instruction:

```
MPI_Bcast(data, 1, Tnew, 0, comm);
```

Question 3–3

We want to implement a parallel program to solve the Sudoku problem. Every possible Sudoku configuration or "board" is represented by an array of 81 integers, containing values between 0 and 9 (0

represents an empty cell). Process 0 generates n solutions, that must be validated by the other processes. These solutions are stored in a matrix A of size $n \times 81$.

(a) Write a parallel code that distributed the whole matrix from process p_0 to the rest of the processes, in a way that each process will receive a different board (assuming $n = p$, where p is the number of processes).

Solution:

```
MPI_Scatter(A, 81, MPI_INT, board, 81, MPI_INT, 0, MPI_COMM_WORLD);
```

(b) Considering that the following struct is created for the MPI implementation:

```
struct task {
  int board[81];
  int initial[81];
  int is_solution;
};
typedef struct task Task;
```

Create an MPI datatype `ttask` representing the previous structure.

Solution:

```
Task t;
MPI_Datatype ttask;
int blocklen[3] = { 81, 81, 1 };
MPI_Aint ad1, ad2, ad3, ad4, disp[3];
MPI_Get_address(&t, &ad1);
MPI_Get_address(&t.board[0], &ad2);
MPI_Get_address(&t.initial[0], &ad3);
MPI_Get_address(&t.es_solution, &ad4);
disp[0] = ad2 - ad1;
disp[1] = ad3 - ad1;
disp[2] = ad4 - ad1;
MPI_Datatype types[3] = { MPI_INT, MPI_INT, MPI_INT };
MPI_Type_create_struct(3, blocklen, disp, types, &ttask);
MPI_Type_commit(&ttask);
```

Question 3–4

Let `A` be a bidimensional array of double precision real numbers, of dimension `N×N`. Define an MPI derived data type that allows to send a submatrix of size 3×3. For instance, the submatrix that starts in `A[0][0]` would be the elements marked with a $\star$:

$$\mathtt{A} = \begin{bmatrix} \star & \star & \star & \cdot & \cdot & \cdot \\ \star & \star & \star & \cdot & \cdot & \cdot \\ \star & \star & \star & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot & \cdot & \cdot \end{bmatrix}.$$

(a) Write the corresponding calls for sending the block in the figure from P_0 and receiving it in P_1.

Solution: Can be viewed as a vector of 3 blocks of elements, each of them with length 3 and stride `N`.

```
double A[N][N];
int rank;
MPI_Datatype newtype;
... /* fill matrix */
MPI_Type_vector(3, 3, N, MPI_DOUBLE, &newtype);
MPI_Type_commit(&newtype);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) {
  MPI_Send(&A[0][0], 1, newtype, 1, 0, MPI_COMM_WORLD);
} else if (rank==1) {
  MPI_Recv(&A[0][0], 1, newtype, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
MPI_Type_free(&newtype);
```

(b) Indicate what should be modified in the previous code so that the block sent by P_0 is the one that starts in the position (0,3), and is received in P_1 overwriting the block that starts in the position (3,0),

Solution: It is enough to change the buffer addresses in `MPI_Send` and `MPI_Recv`. In particular, `&A[0][3]` should be used instead of `&A[0][0]` in the call to `MPI_Send`, and `&A[3][0]` instead of `&A[0][0]` in the call to `MPI_Recv`.

Question 3–5

The following MPI program must compute the sum of two matrices A and B of dimensions $M \times N$ using a row cyclic distribution, assuming that the number of processes `p` is a divisor of M and having into account that P_0 has initially stored matrices A and B.

```
int p, rank, i, j, mb;
double A[M][N], B[M][N], Al[M][N], Bl[M][N];

MPI_Init(&argc, &argv);
MPI_Comm_size(MPI_COMM_WORLD, &p);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (rank==0) read(A,B);

/* (a) Cyclic distribution of rows of A and B */
/* (b) Local computation of Al+Bl */
/* (c) Gather the results in process 0 */

if (rank==0) write(A);
MPI_Finalize();
```

(a) Implement the cyclic distribution of rows of matrices A and B, where `Al` and `Bl` are the local matrices. In order to achieve this distribution you must either define a new MPI datatype or use collective communications.

Solution:
Solution defining a new datatype:

```
MPI_Datatype cyclic_row;
mb = M/p;
MPI_Type_vector(mb, N, p*N, MPI_DOUBLE, &cyclic_row);
MPI_Type_commit(&cyclic_row);
if (rank==0) {
```

```
    for (i=1;i<p;i++) {
      MPI_Send(&A[i][0], 1, cyclic_row, i, 0, MPI_COMM_WORLD);
      MPI_Send(&B[i][0], 1, cyclic_row, i, 1, MPI_COMM_WORLD);
    }
    MPI_Sendrecv(A, 1, cyclic_row, 0, 0, Al, mb*N, MPI_DOUBLE,
                 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(B, 1, cyclic_row, 0, 1, Bl, mb*N, MPI_DOUBLE,
                 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(Al, mb*N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(Bl, mb*N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
```

The `MPI_Sendrecv` operation has been used to copy the part stored locally at process 0; this could have also been accomplished with a loop or with `memcpy`.

Solution using collective communications:

```
mb = M/p;
for (i=0;i<mb;i++) {
  MPI_Scatter(&A[i*p][0], N, MPI_DOUBLE, &Al[i][0], N, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
  MPI_Scatter(&B[i*p][0], N, MPI_DOUBLE, &Bl[i][0], N, MPI_DOUBLE, 0,
              MPI_COMM_WORLD);
}
```

(b) Implement the local computation of the sum `Al`+`Bl`, storing the result in `Al`.

Solution:

```
for (i=0;i<mb;i++)
  for (j=0;j<N;j++)
    Al[i][j] += Bl[i][j];
```

(c) Write the necessary code so that P_0 stores in `A` the matrix $A + B$. For this, P_0 must receive from the rest of processes the local matrices `Al` obtained in the previous step.

Solution: Solution using the datatype defined above:

```
if (rank==0) {
  for (i=1;i<p;i++)
    MPI_Recv(&A[i][0], 1, cyclic_row, i, 3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  MPI_Sendrecv(Al, mb*N, MPI_DOUBLE, 0, 3, A, 1, cyclic_row, 0,
               3, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else MPI_Send(Al, mb*N, MPI_DOUBLE, 0, 3, MPI_COMM_WORLD);
MPI_Type_free(&cyclic_row);
```

Solution using collective communications:

```
for (i=0;i<mb;i++)
  MPI_Gather(&Al[i][0], N, MPI_DOUBLE, &A[i*p][0], N, MPI_DOUBLE, 0,
             MPI_COMM_WORLD);
```

Question 3–6

Implement a function where, given a matrix A of $N \times N$ real numbers and an index k (between 0 and $N-1$), the row k and column k of the matrix are communicated from process 0 to the rest of processes (without communicating any other element of the matrix). The header of the function will be:

```
void bcast_row_col(double A[N][N], int k)
```

You should create and use a datatype for representing a column from the matrix. It is not necessary that you send both the column and the row in the same message, you can send them separately.

Solution:

```
void bcast_row_col(double A[N][N], int k)
{
  MPI_Datatype colu;
  MPI_Type_vector(N, 1, N, MPI_DOUBLE, &colu);
  MPI_Type_commit(&colu);

  /* Sending the row */
  MPI_Bcast(&A[k][0], N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  /* Sending the column */
  MPI_Bcast(&A[0][k], 1, colu, 0, MPI_COMM_WORLD);

  MPI_Type_free(&colu);
}
```

Question 3–7

We want to distribute across 4 processes a square matrix of order $2N$ ($2N$ rows by $2N$ columns) defined by blocks as

$$A = \begin{pmatrix} A_{00} & A_{01} \\ A_{10} & A_{11} \end{pmatrix},$$

where each block A_{ij} corresponds to a square matrix of order N, in such a way that we want process P_0 to store locally matrix A_{00}, P_1 matrix A_{01}, P_2 matrix A_{10} and P_3 matrix A_{11}.

For example, the following matrix with $N = 2$ would be distributed as shown:

$$A = \left(\begin{array}{cc|cc} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{array}\right) \qquad \begin{array}{ll} \text{In } P_0\text{:} \begin{pmatrix} 1 & 2 \\ 5 & 6 \end{pmatrix} & \text{In } P_1\text{:} \begin{pmatrix} 3 & 4 \\ 7 & 8 \end{pmatrix} \\ \text{In } P_2\text{:} \begin{pmatrix} 9 & 10 \\ 13 & 14 \end{pmatrix} & \text{In } P_3\text{:} \begin{pmatrix} 11 & 12 \\ 15 & 16 \end{pmatrix} \end{array}$$

(a) Implement a function that performs the described distribution, by defining the necessary MPI data type. The header of the function would be:

```
void communicate(double A[2*N][2*N], double B[N][N])
```

where `A` is the initial matrix, stored in process 0, and `B` is the local matrix where each process must store the block of A assigned to it.

Note: it is allowed to assume that the number of processes in the communicator is 4.

Solution:

```
void communicate(double A[2*N][2*N], double B[N][N])
{
```

```
    int rank;
    MPI_Datatype mat;

    MPI_Comm_rank(MPI_COMM_WORLD, &rank);
    MPI_Type_vector(N,N,2*N,MPI_DOUBLE,&mat);
    MPI_Type_commit(&mat);
    if (rank==0) {
      MPI_Sendrecv(&A[0][0],1,mat,0,0,B,N*N,MPI_DOUBLE,0,0,MPI_COMM_WORLD,
                   MPI_STATUS_IGNORE);
      MPI_Send(&A[0][N],1,mat,1,0,MPI_COMM_WORLD);
      MPI_Send(&A[N][0],1,mat,2,0,MPI_COMM_WORLD);
      MPI_Send(&A[N][N],1,mat,3,0,MPI_COMM_WORLD);
    }
    else
      MPI_Recv(B,N*N,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Type_free(&mat);
  }
```

(b) Compute the communication time.

Solution: Since P_0 sends a total of three messages of N^2 data to the rest of processes, the communication time is $t_c = 3\left(t_s + N^2 t_w\right)$, being t_s the setup time and t_w the time required to send a single datum.

Question 3–8

Develop a function that can be used to send a submatrix from process 0 to process 1, where it will be stored as a vector. A new data type must be employed, so that a single message is sent. Remember that matrices in C are stored in memory by rows.

The header of the function will be:

```
void send(int m,int n,double A[M][N],double v[MAX],MPI_Comm comm)
```

Note: we can assume that `m*n<=MAX` and that the submatrix to be sent starts at element `A[0][0]`.

Example with $M = 4$, $N = 5$, $m = 3$, $n = 2$:

A (in P_0)						v (in P_1)
1	2	0	0	0		
3	4	0	0	0	$\rightarrow$	1 2 3 4 5 6
5	6	0	0	0		
0	0	0	0	0		

Solution:

```
void send(int m,int n,double A[M][N],double v[MAX],MPI_Comm comm)
{
  int myid;
  MPI_Datatype mat;

  MPI_Comm_rank(comm,&myid);
  MPI_Type_vector(m,n,N,MPI_DOUBLE,&mat);
  MPI_Type_commit(&mat);
  if (myid == 0)
```

```
    MPI_Send(&A[0][0],1,mat,1,2512,comm);
  else if (myid == 1)
    MPI_Recv(&v[0],m*n,MPI_DOUBLE,0,2512,comm,MPI_STATUS_IGNORE);
  MPI_Type_free(&mat);
}
```

Question 3–9

We want to distribute, using MPI, the square sub-matrix blocks in the diagonal of a square matrix of dimension $3 \cdot \mathrm{DIM}$ among 3 processes. For example, if the matrix is of dimension 6 (DIM=2), the distribution will be as follows:

$$\begin{pmatrix} a_{00} & a_{01} & \dots & \dots & \dots & \dots \\ a_{10} & a_{11} & \dots & \dots & \dots & \dots \\ \dots & \dots & a_{22} & a_{23} & \dots & \dots \\ \dots & \dots & a_{32} & a_{33} & \dots & \dots \\ \dots & \dots & \dots & \dots & a_{44} & a_{45} \\ \dots & \dots & \dots & \dots & a_{54} & a_{55} \end{pmatrix} \rightarrow \begin{array}{c} P_0 \\ \\ P_1 \\ \\ P_2 \\ \\ \end{array} \begin{bmatrix} a_{00} & a_{01} \\ a_{10} & a_{11} \end{bmatrix} \begin{bmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{bmatrix} \begin{bmatrix} a_{44} & a_{45} \\ a_{54} & a_{55} \end{bmatrix}$$

Implement a parallel function that sends the square blocks of the matrix with the minimum number of messages. We provide the header of the function to ease its implementation. Process 0 has the full matrix in `A` and after the call, every process must have its corresponding square block in `Alcl`. Use point-to-point communication primitives.

```
void SendBAD(double A[3*DIM][3*DIM], double Alcl[DIM][DIM]) {
```

Solution:

```
void SendBAD(double A[3*DIM][3*DIM], double Alcl[DIM][DIM]) {
  int i,rank,p,m,n;
  MPI_Datatype DiagBlock;

  MPI_Comm_size(MPI_COMM_WORLD, &p);
  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  n = 3*DIM;
  m = DIM;
  if (rank == 0) {
    MPI_Type_vector(m, m, n, MPI_DOUBLE, &DiagBlock);
    MPI_Type_commit(&DiagBlock);
    MPI_Sendrecv(A, 1, DiagBlock, 0, 0, Alcl, m*m, MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    for (i=1;i<p;i++)
      MPI_Send(&A[i*m][i*m], 1, DiagBlock, i, 0, MPI_COMM_WORLD);
    MPI_Type_free(&DiagBlock);
  } else {
    MPI_Recv(Alcl, m*m, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
}
```

Question 3–10

Given a matrix with `NF` rows and `NC` columns, initially stored in process 0, we want to distribute it by

blocks of columms between processes 0 and 1. Process 0 will keep the first half of the columns and process 1 will get the second half (we will assume that `NC` is even).

Implement a function, using the header provided, that will implement this distribution using MPI. You should define the data type required to ensure that the elements that belong to process 1 are sent with a single message. When the function finishes, both processes must have in `Aloc` its corresponding block of columns. The number of processes could be larger than 2, and then, only processes 0 and 1 will store its column block in `Aloc`.

```
void distribute(double A[NF][NC], double Aloc[NF][NC/2])
```

Solution:

```
void distribute(double A[NF][NC], double Aloc[NF][NC/2])
{
   int rank;
   MPI_Status stat;
   MPI_Datatype cols;
   MPI_Type_vector(NF, NC/2, NC, MPI_DOUBLE, &cols);
   MPI_Type_commit(&cols);
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if (rank==0) {
      MPI_Sendrecv(A, 1, cols, 0, 100, Aloc, NF*NC/2, MPI_DOUBLE, 0, 100,
                   MPI_COMM_WORLD, &stat);
      MPI_Send(&A[0][NC/2], 1, cols, 1, 100, MPI_COMM_WORLD);
   }
   else if (rank==1) {
      MPI_Recv(Aloc, NF*NC/2, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, &stat);
   }
   MPI_Type_free(&cols);
}
```

Question 3–11

We want to implement a communication operation among three MPI processes of a matrix `A` of size $N \times N$, stored at P_0. In this operation Process P_1 will receive the submatrix composed of the rows with even index and process P_2 will receive the submatrix composed of the rows with odd index. You must use MPI Derived types to minimize the number of messages. Each submatrix received at P_1 and P_2 must be stored in an $N/2 \times N$ local matrix `B`. N.B.: You can assume that N is an even number.

For example: If the matrix stored in P_0 is

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix}$$

The submatrix in P_1 must be:

$$B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 9 & 10 & 11 & 12 \end{pmatrix}$$

And the submatrix in P_2 must be:

$$B = \begin{pmatrix} 5 & 6 & 7 & 8 \\ 13 & 14 & 15 & 16 \end{pmatrix}.$$

(a) Implement a parallel MPI function for the above operation, using the following header:

```
void communicate(double A[N][N], double B[N/2][N])
```

Solution:

```
void communicate(double A[N][N], double B[N/2][N])
{
  int id;
  MPI_Datatype mitad;
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Type_vector(N/2, N, 2*N, MPI_DOUBLE, &mitad);
  MPI_Type_commit(&mitad);
  if (id==0) {
    MPI_Send(&A[0][0], 1, mitad, 1, 100, MPI_COMM_WORLD);
    MPI_Send(&A[1][0], 1, mitad, 2, 100, MPI_COMM_WORLD);
  }
  else if (id==1 || id==2)
    MPI_Recv(B,N/2*N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  MPI_Type_free(&mitad);
}
```

(b) Compute the communication time of the implemented function.

Solution: It comprises sending and receiving two messages with $N^2/2$ elements. Therefore, the communication time is:

$$t_c = 2\left(t_s + \frac{N^2}{2}t_w\right).$$

Question 3–12

Implement a function in C to send the three main diagonals of a square matrix to all the processes. You must consider neither the first nor the final rows of the matrix. For example, the elements that must be considered for a matrix of size 6 are the ones marked with `x`:

```
+ + + + + +
x x x + + +
+ x x x + +
+ + x x x +
+ + + x x x
+ + + + + +
```

The function must define a new MPI datatype that could be used to send the whole tridiagonal block in a single message. Bear in mind that matrices in C are stored in memory by rows. Use the following header for the function:

```
void send_tridiagonal(double A[N][N],int root,MPI_Comm comm)
```

where

- `N` is the number of rows and columns of the matrix.
- `A` is the matrix with the data to be sent (in the process that sends the data) and the matrix where the data must be received (in the rest of the processes).
- Parameter `root` indicates the process that initially has the data to be sent in matrix `A`.

- `comm` is the communicator for all the processes that will have the tridiagonal part of `A` in their memories.

For example, if the function is called as:

```
send_tridiagonal(A,5,comm);
```

Process 5 will be the one that has the valid data in `A` when the function is called, and at the return of the call, all processes in communicator `comm` will have the tridiagonals (except first and last rows) in `A`.

Solution:

```
void send_tridiagonal(double A[N][N],int root,MPI_Comm comm)
{
  MPI_Datatype diag3;
  MPI_Type_vector(N-2,3,N+1,MPI_DOUBLE,&diag3);
  MPI_Type_commit(&diag3);
  MPI_Bcast(&A[1][0],1,diag3,root,comm);
  MPI_Type_free(&diag3);
}
```

Question 3–13

The next MPI code fragment implements an algorithm in which each process computes a matrix of `M` rows and `N` columns. All those matrices are collected in process P_0 forming a global matrix with `M` rows and `N*p` columns (where `p` is the number of processes). In this global matrix, we have first the columns of P_0, then the columns of P_1, followed by the columns of P_2 and so on.

```
int rank, i, j, k, p;
double alocal[M][N];
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
/* initialization of alocal omitted here */
if (rank==0) {
  double aglobal[M][N*p];
  /* copy part belonging to P0 */
  for (i=0;i<M;i++)
    for (j=0;j<N;j++)
      aglobal[i][j] = alocal[i][j];
  /* receive data from other processes */
  for (k=1;k<p;k++)
    for (i=0;i<M;i++)
      MPI_Recv(&aglobal[i][k*N],N,MPI_DOUBLE,k,33,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  write(p,aglobal);
} else {
  for (i=0;i<M;i++)
    MPI_Send(&alocal[i][0],N,MPI_DOUBLE,0,33,MPI_COMM_WORLD);
}
```

(a) Change the previous code so that each process sends a single message, instead of one message per row. For this purpose, you should define an MPI derived type for the reception of the message.

Solution: The processes will send a single message with the `M*N` elements of the submatrix. However, process P_0 will not store those elements contiguously in memory, so we will define a "vector" MPI type. The new type will have `M` blocks (one per row) with `N` elements (as many

elements as columns in the sender process), with a *stride* between blocks of `N*p`, as the whole matrix in P_0 has `N*p` columns. In the `MPI_Recv` MPI operation, the buffer points to the first position of the submatrix for the `k`-th process, that is `aglobal[0][k*N]`.

```
int rank, i, j, k, p;
double alocal[M][N];
MPI_Datatype cols;
MPI_Comm_size(MPI_COMM_WORLD,&p);
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
/* initialization of alocal omitted here */
if (rank==0) {
  double aglobal[M][N*p];
  /* copy part belonging to P0 */
  for (i=0;i<M;i++)
    for (j=0;j<N;j++)
      aglobal[i][j] = alocal[i][j];
  /* receive data from other processes */
  MPI_Type_vector(M, N, N*p, MPI_DOUBLE, &cols);
  MPI_Type_commit(&cols);
  for (k=1;k<p;k++) {
    MPI_Recv(&aglobal[0][k*N], 1, cols, k, 33, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
  }
  MPI_Type_free(&cols);
  write(p,aglobal);
} else {
  MPI_Send(&alocal[0][0], M*N, MPI_DOUBLE, 0, 33, MPI_COMM_WORLD);
}
```

(b) Obtain the communication cost for both the original and the modified versions.

Solution: Original version: A total of $(p-1)M$ messages are sent, each one with N elements. Therefore,

$$t_c = (p-1)M(t_s + Nt_w).$$

Modified version: A total of $p-1$ messages are sent, each one with a length of MN. Therefore,

$$t_c = (p-1)(t_s + MNt_w).$$

Question 3–14

We want to distribute a matrix `A` with `F` rows and `C` columns among the processes in an MPI communicator, using a distribution based on blocks of columns. The number of processes is `C`/2, and `C` is an even number. The local matrix `Aloc` in each process will hold two columns.

Implement a function using the following header to perform the previous distribution and using point-to-point communication primitives. The matrix `A` is initially in process 0, and at the end of the function each process should have in `Aloc` the corresponding local part of the global matrix.

Use the proper MPI data types to ensure that only one message per process is sent.

```
void distrib(double A[F][C], double Aloc[F][2], MPI_Comm com)
```

Solution:

```
void distrib(double A[F][C], double Aloc[F][2], MPI_Comm com)
{
  int p, rank, i;
  MPI_Datatype blq;
  MPI_Comm_size(com, &p);
  MPI_Comm_rank(com, &rank);
  MPI_Type_vector(F, 2, C, MPI_DOUBLE, &blq);
  MPI_Type_commit(&blq);
  if (rank==0) {
    for (i=1; i<p; i++) {
      MPI_Send(&A[0][i*2], 1, blq, i, 33, com);
    }
    MPI_Sendrecv(&A[0][0], 1, blq, 0, 33, &Aloc[0][0], F*2, MPI_DOUBLE,
                 0, 33, com, MPI_STATUS_IGNORE);
  }
  else {
    MPI_Recv(&Aloc[0][0], F*2, MPI_DOUBLE, 0, 33, com, MPI_STATUS_IGNORE);
  }
  MPI_Type_free(&blq);
}
```