Question 1 (1.1 points)

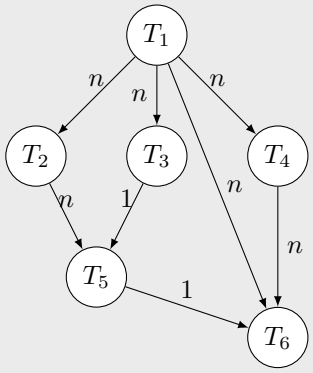
We want to parallelize the following code with MPI.

```
int calculate(int n, double x[], double y[], double z[]) {
  int i;
  double alpha, beta;

  /* Read vectors x, y, z, of dimension n */
  read(n, x, y, z);           /* task 1 */

  normalize(n,x);             /* task 2 */
  beta = obtain(n,y);         /* task 3 */
  normalize(n,z);             /* task 4 */

  /* task 5 */
  alpha = 0.0;
  for (i=0; i<n; i++)
    if (x[i] > 0.0) { alpha = alpha + beta*x[i]; }
    else { alpha = alpha + x[i]*x[i]; }

  /* task 6 */
  for (i=0; i<n; i++) z[i] = z[i] + alpha*y[i];
}
```

Suppose we are using 3 processes, from which only one has to call function `read`. We can assume that the value of `n` is available in all processes. The final result (`z`) may be stored in any of the 3 processes. Function `read` modifies the three vectors, function `normalize` modifies its second argument and function `obtain` does not modify any of its arguments.

0.25 p. (a) Draw the task dependency graph.

Solution: The task dependency graph is the following:



Although it is not requested, the graph shows the edges labeled with the data volume that is transferred between each pair of dependent tasks. This information has to be taken into account to select the optimal task assignment.

0.85 p.

(b) Write the MPI code that solves the problem using an assignment that maximizes the parallelism and minimizes the cost of communications.

Solution: A task assignment that satisfies the requirements is $P_0 : T_1, T_4, T_6$; $P_1 : T_3$; $P_2 : T_2, T_5$.

```
int calculate_mpi(int n, double x[], double y[], double z[]) {
  int i;
  double alpha, beta;

  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if( rank == 0 ) {
    /* Read vectors x, y, z, of dimension n */
    read( n, x, y, z );      /* task 1 */
    MPI_Send(x, n, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD);
    MPI_Send(y, n, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
    normalize(n,z);             /* task 4 */
    MPI_Recv(&alpha, 1, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /* task 6 */
    for (i=0; i<n; i++) z[i] = z[i] + alpha*y[i];
  }
  else if (rank == 1) {
    MPI_Recv(y, n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    beta = obtain(n,y);        /* task 3 */
    MPI_Send(&beta, 1, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD);
  }
  else if (rank == 2) {
    MPI_Recv(x, n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    normalize(n,x);             /* task 2 */
    MPI_Recv(&beta, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /* task 5 */
    alpha = 0.0;
    for (i=0; i<n; i++)
      if (x[i] > 0.0) { alpha = alpha + beta*x[i]; }
      else { alpha = alpha + x[i]*x[i]; }
    MPI_Send(&alpha, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD );
  }
}
```

Question 2 (1.3 points)

We want to implement with MPI the sending by process 0 (and reception by the rest of processes) of the main diagonal and antidiagonal of a matrix A, using derived data types (one type per each class of diagonal) and the smallest possible number of messages. Suppose that:

- N is a known constant.
- The elements of the main diagonal are: $A_{0,0}$, $A_{1,1}$, $A_{2,2}$, ..., $A_{N-1,N-1}$.
- The elements of the antidiagonal are: $A_{0,N-1}$, $A_{1,N-2}$, $A_{2,N-3}$, ..., $A_{N-1,0}$.
- Only process 0 owns matrix A and will send the full diagonals to the rest of processes.

An example for a matrix of size $N = 5$ would be: $A = \begin{pmatrix} * & & & & * \\ & * & & * & \\ & & * & & \\ & * & & * & \\ * & & & & * \end{pmatrix}$

0.6 p. (a) Complete the following function, where the processes from rank 1 onward will store on matrix `A` the received diagonals:

```
void sendrecv_diagonals(double A[N][N]) {
```

Solution:

```
void sendrecv_diagonals(double A[N][N]) {
  MPI_Datatype tmain,tanti;
  MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &tmain);
  MPI_Type_commit(&diagonal);
  MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &tanti);
  MPI_Type_commit(&tanti);
  MPI_Bcast(A, 1, tmain, 0, MPI_COMM_WORLD);
  MPI_Bcast(&A[0][N-1], 1, tanti, 0, MPI_COMM_WORLD);
  MPI_Type_free(&tmain);
  MPI_Type_free(&tanti);
}
```

0.7 p. (b) Complete this other function, a variant of the previous one, where all processes (including process 0) will store on vectors `maind` and `antid` the corresponding diagonals:

```
void sendrecv_diagonals(double A[N][N], double maind[N], double antid[N]) {
```

Solution:

```
void sendrecv_diagonals(double A[N][N], double maind[N], double antid[N]) {
  int nprocs, id, p;
  MPI_Datatype tmain,tanti;
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  if (id==0) {
    MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &tmain);
    MPI_Type_commit(&diagonal);
    MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &tanti);
    MPI_Type_commit(&tanti);
    MPI_Sendrecv(A, 1, tmain, 0, 0, maind, N ,MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&A[0][N-1], 1, tanti, 0, 1, antid, N,
                 MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    for (p=1; p<nprocs; p++) {
      MPI_Send(A, 1, tmain, p, 0, MPI_COMM_WORLD);
      MPI_Send(&A[0][N-1], 1, tanti, p, 1, MPI_COMM_WORLD);
    }
    MPI_Type_free(&tmain);
    MPI_Type_free(&tanti);
  }
  else {
    MPI_Recv(maind, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
```

```
    MPI_Recv(antid, N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
}
```

The previous solution can be simplified by using the `MPI_Bcast` primitive:

```
void sendrecv_diagonals(double A[N][N], double maind[N], double antid[N]) {
  int id;
  MPI_Datatype tmain,tanti;
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  if (id==0) {
    MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &tmain);
    MPI_Type_commit(&diagonal);
    MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &tanti);
    MPI_Type_commit(&tanti);
    MPI_Sendrecv(A, 1, tmain, 0, 0, maind, N ,MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&A[0][N-1], 1, tanti, 0, 1, antid, N,
                 MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Type_free(&tmain);
    MPI_Type_free(&tanti);
  }
  MPI_Bcast(maind, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
  MPI_Bcast(antid, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
}
```

Question 3 (1.1 points)

Observe the following function, that counts the number of occurences of a number in a matrix and also indicates the first row in which it appears:

```
void search(double A[M][N], double x) {
  int i,j,first,count;
  first = M ; count = 0;
  for (i=0; i<M; i++)
    for (j=0; j<N; j++)
      if (A[i][j] == x) {
        count++;
        if (i < first) first = i;
      }
  printf("%g is found %d times, the first one in row %d.\n",x,count,first);
}
```

0.9 p.

(a) Parallelize it by means of MPI distributing the `A` matrix among all available processes. Both the matrix and the value to be sought are initially only available at process `owner`. We assume that the number of rows and columns of the matrix is an exact multiple of the number of processes. The `printf` that shows the result on the screen must be done only by one process.

Use collective communication operations whenever possible.

For this, complete this function:

```
void par_search(double A[M][N], double x, int owner) {
  double Aloc[M][N];
```

Solution: We can use a distribution of the matrix by blocks of rows:

```
void par_search(double A[M][N], double x, int owner) {
  double Aloc[M][N];
  int i,j,first,count, fl,cl, id,np, rows,size;

  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  rows=M/np; size=rows*N;

  /* Distribute the matrix by blocks of rows */
  MPI_Scatter(A,size,MPI_DOUBLE,Aloc,size,MPI_DOUBLE,owner,MPI_COMM_WORLD);
  /* Broadcast the value to be sought */
  MPI_Bcast(&x,1,MPI_DOUBLE,owner,MPI_COMM_WORLD);

  /* Local computation */
  fl = M ; cl = 0;
  for (i=0; i<rows; i++)
    for (j=0; j<N; j++)
      if (Aloc[i][j] == x) {
        cl++;
        if (i < fl) fl = i;
      }

  /* Collect the results in a process and print */
  fl = rows*id + fl;  /* Pasar índice local a global */
  MPI_Reduce(&fl,&first,1,MPI_INT,MPI_MIN,0,MPI_COMM_WORLD);
  MPI_Reduce(&cl,&count,1,MPI_INT,MPI_SUM,0,MPI_COMM_WORLD);
  if (id == 0)
    printf("%g is found %d times, the first one in row %d.\n",x,count,first);
}
```

0.2 p.

(b) Indicate the communication cost of each communication operation that has been used in the previous code. Assume a basic implementation of the communications.

Solution:

$$t_{\text{scatter}} = (p-1) \cdot (t_s + \frac{MN}{p} t_w)$$
$$t_{\text{bcast}} = (p-1) \cdot (t_s + t_w)$$
$$t_{\text{reduce}} = (p-1) \cdot (t_s + t_w)$$