Question 1 (1.1 points)

We want to parallelize the following code:

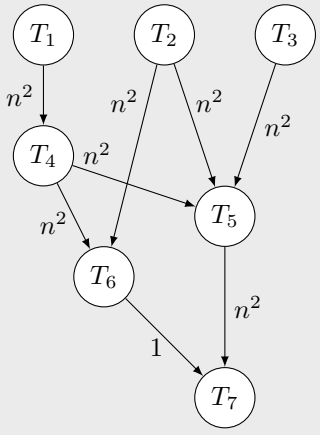
```
#define n 1000
double function( double A[n][n], double B[n][n], double C[n][n] ) {
  double a;
  f1(A);           /* T1 Cost = n^2   */
  f2(B);           /* T2 Cost = n^2   */
  f3(C);           /* T3 Cost = n^2   */
  f4(A);           /* T4 Cost = n^2   */
  f5(A,B,C);       /* T5 Cost = 2*n^2 */
  a = f6(A,B);     /* T6 Cost = n^2   */
  a *= f7(C);      /* T7 Cost = n     */
  return a;
}
```

taking into account the following:

- functions `f1`, `f2` and `f3` initialize their respective arguments (they are output arguments);
- function `f4` modifies its argument;
- function `f5` updates its third argument; and
- the rest of arguments are all input only.

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

Solution: The task dependency graph is the following:



Although not requested in the question, the graph shows the edges labeled with the volume of data that is transferred between each pair of dependent tasks, which has to be taken into account in order to select the best possible assignment of tasks to processes.

0.7 p. (b) Choose an assignment that maximizes parallelism and minimizes communication cost using 2 processes and taking into account that the return value must be valid in process P_0 at least. Indicate, for the chosen

assignment, which tasks are performed by each process. Then, write the MPI code that solves the problem using the chosen assignment.

Solution: A task assignment that satisfies the restrictions is $P_0 : T_2, T_3, T_5, T_7$; $P_1 : T_1, T_4, T_6$. The MPI code would be the following:

```
#define n 1000

double function( double A[n][n], double B[n][n], double C[n][n] ) {

  double a;
  int rank;
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if( rank == 0 ) {
    f2(B);           /* T2 Cost = n^2   */
    f3(C);           /* T3 Cost = n^2   */
    MPI_Recv( A, n*n, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    MPI_Send( B, n*n, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD );
    f5(A,B,C);       /* T5 Cost = 2*n^2 */
    MPI_Recv( &a, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    a *= f7(C);      /* T7 Cost = n     */
  } else if( rank == 1 ) {
    f1(A);           /* T1 Cost = n^2   */
    f4(A);           /* T4 Cost = n^2   */
    MPI_Send( A, n*n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD );
    MPI_Recv( B, n*n, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    a = f6(A,B);     /* T6 Cost = n^2   */
    MPI_Send( &a, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD );
  }
  return a;
}
```

The sending of `B` by P_0 could have been done before, right after executing `f2`. However, this could give rise to a deadlock, and for this reason P_0 performs the sending of `B` after receiving `A`.

0.2 p.

(c) Indicate the sequential cost, the parallel cost (arithmetic and communications) of the version implemented in the previous section. Show also the speedup and efficiency.

Solution:

- The sequential cost is: $T_s(n) = 7n^2 + n \approx 7n^2$.
- The parallel arithmetic cost is: $T_a(n,p) = 4n^2 + n \approx 4n^2$.
- The communications of the version corresponding to the chosen assignment include two messages of length n^2 and one message of 1 element, hence

$$T_c(n,2) = 2(t_s + n^2 t_w) + (t_s + t_w) = 3t_s + (2n^2+1)t_w \approx 3t_s + 2n^2 t_w.$$

- The speedup is: $S = \dfrac{7n^2}{4n^2 + 3t_s + 2n^2 t_w}$.
- The efficiency is: $E = \dfrac{S}{2} = \dfrac{7n^2}{8n^2 + 6t_s + 4n^2 t_w}$.

Question 2 (1.3 points)

The following program reads from disk a matrix A and two vectors x and y to update the values of the matrix by a rank-1 operation: $A \leftarrow A + xy^T$ and then saves it back to disk.

```
#include <stdio.h>
#define M 2000
#define N 1000
int main(int argc,char *argv[])
{ double A[M][N], x[M], y[N];
  int i,j;
  read(A,x,y);
  for ( i = 0 ; i < M ; i++ )
    for ( j = 0 ; j < N ; j++ )
      A[i][j] += x[i] * y[j];
  write(A);
  return 0;
}
```

1 p. (a) Parallelize this program with MPI, ensuring that the work is distributed among all available processes. Use collective communication operations wherever possible. Only process 0 has access to the disk, so the operations `read` and `write` must be done by this process. We assume that `M` and `N` are an exact multiple of the number of processes.

Solution:

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

#define M 2000
#define N 1000

int main(int argc,char *argv[])
{ double A[M][N], x[M], y[N], B[M][N], z[M];
  int i,j, id,np, nb;

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);

  if ( id == 0 ) read(A,x,y);

  nb = M / np;
  MPI_Scatter( A,nb*N,MPI_DOUBLE, B,nb*N,MPI_DOUBLE, 0, MPI_COMM_WORLD );
  MPI_Scatter( x,nb,MPI_DOUBLE, z,nb,MPI_DOUBLE, 0, MPI_COMM_WORLD );
  MPI_Bcast( y,N,MPI_DOUBLE, 0, MPI_COMM_WORLD );

  for ( i = 0 ; i < nb ; i++ )
    for ( j = 0 ; j < N ; j++ )
      B[i][j] += z[i] * y[j];

  MPI_Gather( B,nb*N,MPI_DOUBLE, A,nb*N,MPI_DOUBLE, 0, MPI_COMM_WORLD);
  if ( id == 0 ) write(A);
```

```
    MPI_Finalize();

    return 0;
  }
```

0.3 p. (b) Indicate the communications cost of each communication operation you have used, assuming a simple implementation of communications.

Solution: $t_{scatter1} = (p-1)(t_s + M/pNt_w)$
$t_{scatter2} = (p-1)(t_s + M/pt_w)$
$t_{gather} = (p-1)(t_s + M/pNt_w)$
$t_{bcast} = (p-1)(t_s + Nt_w)$

Question 3 (1.1 points)

The following function calculates the sum, in absolute value, of the difference between two matrices A and B premultiplied by the values a and b:

```
float sum_dif(float a,float A[M][N],float b,float B[M][N]) {
    int i,j;
    float sum;
    sum=0;
    for (j=0;j<N;j++) {
        for (i=0;i<M;i++) {
            sum+=fabs(a*A[i][j]-b*B[i][j]);
        }
    }
    return sum;
}
```

Parallelize such function in a way that the columns of the matrices are distributed cyclically. The parallel function will have the following header:

```
float sum_dif_par(float a,float A[M][N],float b,float B[M][N], int id)
```

where `id` indicates the rank of the process that stores the data initially. You must employ derived data types, so that each process receives a single message with all the elements of A assigned to it (and the same for B). We will understand that:

- The process identified by the argument `id` of the function will initially have the values a and b, which it will have to send to the rest of the processes, and the matrices A and B, which it will have to distribute among the processes. The processes will store the received data in the same variables a, b, A and B. The elements of A and B received by each process must be placed in the same positions they occupied in the original matrix. For example, if the matrix A of process 0 were as shown below (left), and assuming 3 processes, processes 1 and 2 would end up with the matrix A shown (the elements marked with $\times$ could have any value):

$$\underset{P_0}{\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 6 \\ 2 & 3 & 4 & 5 & 6 & 7 \end{pmatrix}} \rightarrow \underset{P_1}{\begin{pmatrix} \times & 1 & \times & \times & 4 & \times \\ \times & 2 & \times & \times & 5 & \times \\ \times & 3 & \times & \times & 6 & \times \end{pmatrix}} \quad \underset{P_2}{\begin{pmatrix} \times & \times & 2 & \times & \times & 5 \\ \times & \times & 3 & \times & \times & 6 \\ \times & \times & 4 & \times & \times & 7 \end{pmatrix}}$$

- The return value of the function must be valid in the process indicated by the argument `id`.
- We will assume that `N` is a known constant, multiple of the number of processes.

Solution:

```
float sum_dif_par(float a,float A[M][N],float b,float B[M][N],int id) {
    int i,j,myid,np;
    float sum,sum_local;
    MPI_Datatype columnas_ciclicas;
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    MPI_Comm_size(MPI_COMM_WORLD,&np);
    MPI_Type_vector(M*N/np,1,np,MPI_FLOAT,&columnas_ciclicas);
    MPI_Type_commit(&columnas_ciclicas);
    MPI_Bcast(&a,1,MPI_FLOAT,id,MPI_COMM_WORLD);
    MPI_Bcast(&b,1,MPI_FLOAT,id,MPI_COMM_WORLD);
    if (myid==id) {
        for (i=0;i<np;i++) {
            if (i!=myid) {
                MPI_Send(&A[0][i],1,columnas_ciclicas,i,0,MPI_COMM_WORLD);
                MPI_Send(&B[0][i],1,columnas_ciclicas,i,0,MPI_COMM_WORLD);
            }
        }
    }
    else {
        MPI_Recv(&A[0][myid],1,columnas_ciclicas,id,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
        MPI_Recv(&B[0][myid],1,columnas_ciclicas,id,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    }

    sum_local=0;
    for (j=myid;j<N;j+=np) {
        for (i=0;i<M;i++) {
            sum_local+=fabs(a*A[i][j]-b*B[i][j]);
        }
    }
    MPI_Reduce(&sum_local,&sum,1,MPI_FLOAT,MPI_SUM,id,MPI_COMM_WORLD);
    MPI_Type_free(&columnas_ciclicas);
    return sum;
}
```