Cuestión 1 (1.1 puntos)

Se quiere paralelizar el siguiente código:

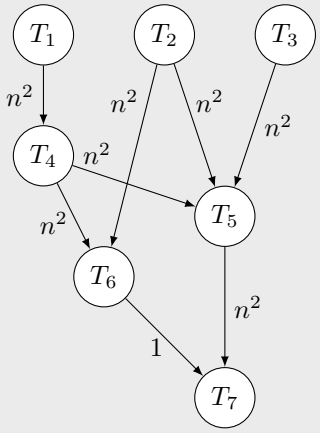
```
#define n 1000
double funcion( double A[n][n], double B[n][n], double C[n][n] ) {
  double a;
  f1(A);           /* T1 Coste = n^2   */
  f2(B);           /* T2 Coste = n^2   */
  f3(C);           /* T3 Coste = n^2   */
  f4(A);           /* T4 Coste = n^2   */
  f5(A,B,C);       /* T5 Coste = 2*n^2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

teniendo en cuenta lo siguiente:

- las funciones `f1`, `f2` y `f3` inicializan sus respectivos argumentos (son de salida);
- la función `f4` modifica su argumento;
- la función `f5` actualiza su tercer argumento; y
- el resto de argumentos son todos de lectura.

0.2 p. (a) Dibuja el grafo de dependencias de las diferentes tareas.

Solución: El grafo de dependencias es el siguiente:



Aunque no se pide en la pregunta, el grafo muestra los arcos etiquetados con el volumen de datos que se transfiere entre cada par de tareas dependientes, lo que ha de tenerse en cuenta para seleccionar la mejor asignación de tareas a procesos posible.

0.7 p. (b) Elige una asignación que maximice el paralelismo y minimice el coste de comunicaciones utilizando 2 procesos y teniendo en cuenta que el valor de retorno ha de ser válido en el proceso P_0 al menos. Indica,

para la asignación elegida, qué tareas realiza cada proceso. Después, escribe el código MPI que resuelve el problema utilizando la asignación elegida.

Solución: Una asignación de tareas que cumple los requisitos es $P_0 : T_2, T_3, T_5, T_7$; $P_1 : T_1, T_4, T_6$. El código MPI sería el siguiente:

```
#define n 1000

double funcion( 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 Coste = n^2   */
    f3(C);           /* T3 Coste = 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 Coste = 2*n^2 */
    MPI_Recv( &a, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE );
    a *= f7(C);      /* T7 Coste = n     */
  } else if( rank == 1 ) {
    f1(A);           /* T1 Coste = n^2   */
    f4(A);           /* T4 Coste = 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 Coste = n^2   */
    MPI_Send( &a, 1, MPI_DOUBLE, 0, 123, MPI_COMM_WORLD );
  }
  return a;
}
```

El envío de `B` por parte de P_0 podría haberse hecho antes, nada más ejecutar `f2`, sin embargo, esto podría dar lugar a un interbloqueo, por esa razón P_0 realiza el envío de `B` después de recibir `A`.

0.2 p.

(c) Indica el coste secuencial, el coste paralelo (aritmético y de comunicaciones) de la versión implementada en el apartado anterior. Muestra también el speedup y la eficiencia.

Solución:

- El coste secuencial es: $T_s(n) = 7n^2 + n \approx 7n^2$.
- El coste paralelo aritmético es: $T_a(n,p) = 4n^2 + n \approx 4n^2$.
- Las comunicaciones de la versión correspondiente a la asignación elegida están constituidas por dos mensajes de tamaño n^2 y uno de un elemento, luego

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

- El speedup es: $S = \dfrac{7n^2}{4n^2 + 3t_s + 2n^2 t_w}$.
- La eficiencia es: $E = \dfrac{S}{2} = \dfrac{7n^2}{8n^2 + 6t_s + 4n^2 t_w}$.

Cuestión 2 (1.3 puntos)

El siguiente programa lee de disco una matriz A y sendos vectores x e y para actualizar los valores de la matriz mediante una operación de rango 1: $A \leftarrow A + xy^T$ y la vuelve a guardar en disco.

```
#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;

  lee(A,x,y);
  for ( i = 0 ; i < M ; i++ )
    for ( j = 0 ; j < N ; j++ )
      A[i][j] += x[i] * y[j];
  escribe(A);

  return 0;
}
```

1 p. (a) Paraleliza este programa con MPI procurando que el trabajo quede repartido entre todos los procesos disponibles. Utiliza operaciones de comunicación colectiva allá donde sea posible. Sólo el proceso 0 tiene acceso al disco, con lo que las operaciones `lee` y `excribe` debe hacerlas este proceso. Asumimos que `M` y `N` son un múltiplo exacto del número de procesos.

Solución:

```
#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 ) lee(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 ) escribe(A);

    MPI_Finalize();

    return 0;
  }
```

0.3 p.

(b) Indica el coste de comunicaciones de cada operación de comunicación que hayas utilizado, suponiendo una implementación sencilla de las comunicaciones.

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

Cuestión 3 (1.1 puntos)

La siguiente función calcula la suma, en valor absoluto, de la diferencia entre dos matrices A y B premultiplicadas por los valores a y b:

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

Paraleliza dicha función de manera que se lleve a cabo un reparto cíclico de las columnas de las matrices. La función paralela tendrá la siguiente cabecera:

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

donde `id` indica el índice del proceso que tiene inicialmente los datos. Se deben emplear tipos de datos derivados, de manera que cada proceso reciba en un único mensaje todos los elementos de A que le correspondan (y lo mismo para B). Entenderemos que:

- El proceso identificado mediante el argumento `id` de la función dispondrá inicialmente de los valores a y b, que tendrá que enviar al resto de procesos, y de las matrices A y B, que tendrá que repartir entre los procesos. Los procesos guardarán los datos recibidos en las mismas variables a, b, A y B. Los elementos de A y B recibidos por cada proceso se deben colocar en las mismas posiciones que ocupaban en la matriz original. Por ejemplo, si la matriz A del proceso 0 fuera la indicada a continuación (izquierda), y suponiendo 3 procesos, los procesos 1 y 2 acabarían con la matriz A que se indica (los elementos marcados con $\times$ podrían tener cualquier valor):

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

- El valor de retorno de la función deberá ser válido en el proceso indicado por el argumento `id`.
- Supondremos que `N` es una constante conocida, múltiplo del número de procesos.

Solución:

```
float suma_dif_par(float a,float A[M][N],float b,float B[M][N],int id) {
    int i,j,myid,np;
    float suma,suma_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);
    }

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