Qüestió 1 (1.1 punts)

Es vol paral · lelitzar el següent codi:

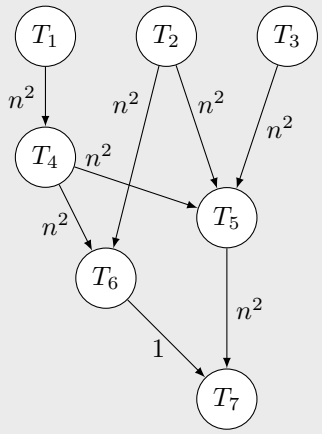
```
#define n 1000

double funcio( 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;

}
```

tenint en compte el següent:

- les funcions `f1`, `f2` i `f3` inicialitzen els seus respectius arguments (són d'eixida);
- la funció `f4` modifica el seu argument;
- la funció `f5` actualitza el seu tercer argument; i
- la resta d'arguments són tots de lectura.

0.2 p. (a) Dibuixa el graf de dependències de les diferents tasques.

Solució: El graf de dependències és el següent:



Encara que no es demana en la pregunta, el graf mostra els arcs etiquetats amb el volum de dades que es transfereix entre cada parella de tasques dependents, el que s'ha de tindre en compte per a seleccionar la millor assignació de tasques a processos possible.

0.7 p.

(b) Elegeix una assignació que maximitze el paral · lelisme i minimitze el cost de comunicacions, utilitzant 2 processos i tenint en compte que el valor de retorn ha de ser vàlid en el procés P_0 al menys. Indica, per a l'assignació elegida, quines tasques realitza cada procés. Després, escriu el codi MPI que resol el problema utilitzant l'assignació elegida.

Solució: Una assignació de tasques que compleix els requisits és $P_0 : T_2, T_3, T_5, T_7$; $P_1 : T_1, T_4, T_6$. El codi MPI seria el següent:

```
#define n 1000

double funcio( 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;
}
```

L'envíament de `B` per part de P_0 podria haver-se fet abans, només executar `f2`, però això podria donar lloc a un interbloqueig, per la qual cosa P_0 realitza l'enviament de `B` després de rebre `A`.

0.2 p.

(c) Indica el cost seqüencial i el cost paral · lel (aritmètic i de comunicacions) de la versió implementada en l'apartat anterior. Mostra també el speedup i l'eficiencia.

Solució:

- El cost seqüencial és: $T_s(n) = 7n^2 + n \approx 7n^2$.
- El cost paral · lel aritmètic es: $T_a(n,p) = 4n^2 + n \approx 4n^2$.
- Les comunicacions de la versió paral · lela són dos missatges de grandària n^2 i un missatge d'un element, per tant

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

- El speedup és: $S = \dfrac{7n^2}{4n^2 + 3t_s + 2n^2 t_w}$.
- L'eficiència és: $E = \dfrac{S}{2} = \dfrac{7n^2}{8n^2 + 6t_s + 4n^2 t_w}$.

Qüestió 2 (1.3 punts)

El següent programa llig de disc una matriu A i dos vectors x e y per a actualitzar els valors de la matriu mitjançant una operació de rang 1: $A \leftarrow A + xy^T$ i tornar a guardar la matriu en disc.

```
#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;

  llig(A,x,y);

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

  escriu(A);

  return 0;
}
```

1 p. (a) Paral · lelitza este programa amb MPI procurant que el treball quede repartit entre tots els processos disponibles. Utilitza operacions de comunicació col · lectiva allà on siga possible. Només el procés 0 té acces al disc, pel que les operacions `llig` i `escriu` ha de fer-les eixe procés. Assumim que `M` i `N` són un múltiple exacte del número de processos.

Solució:

```
#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 ) llig(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 ) escriu(A);

  MPI_Finalize();

  return 0;
}
```

0.3 p. (b) Indica el cost de comunicacions de cada operació de comunicació que hages utilitzat, suposant una implementació senzilla de las comunicacions.

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

Qüestió 3 (1.1 punts)

La següent funció calcula la suma, en valor absolut, de la diferència entre dos matrius A i B premultiplicades pels valors a i 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;
}
```

Paral · lelitza la funció de manera que s'utilitze un repartiment cíclic de les columnes de les matrius. La funció paral · lela tindrà la següent capçalera:

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

on `id` indica l'índex del procés que té inicialment les dades. S'hauran d'emprar tipus de dades derivats, de manera que cada procés reba en un sol missatge tots els elements de A que li corresponguen (i el mateix per a B). S'entendrà que:

- El procés identificat mitjançant l'argument `id` de la funció disposarà inicialment dels valors a i b, què haurà d'enviar a la resta de processos, i de les matrius A i B, que haurà de repartir entre els processos. Els processos guardaran les dades rebudes en les mateixes variables a, b, A i B. Els elements de A i B

rebuts per cada procés s'han de colocar en les mateixes posicions que ocupaven en la matriu original. Per exemple, si la matriu A del procés 0 fora la indicada a continuació (esquerra), i suposant 3 processos, els processos 1 i 2 acabarien amb la matriu A que s'indica (els elements marcats amb $\times$ podrien tindre qualsevol valor):

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

- El valor de retorn de la funció haurà de ser vàlid en el procés indicat per l'argument `id`.
- Suposarem que `N` és una constant coneguda, múltiple del número de processos.

Solució:

```
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;
}
```