1 Comunicació punt a punt

Qüestió 1–1

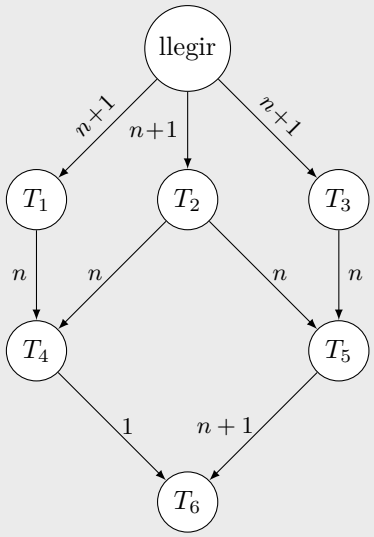
Donada la següent funció:

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

   /* Llegir els vectors v, w, z, de dimensio n */
   llegir(&n, &v, &w, &z);

   calcula_v(n,v);              /* tasca 1 */
   calcula_w(n,w);              /* tasca 2 */
   calcula_z(n,z);              /* tasca 3 */

   /* tasca 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];
   }

   /* tasca 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];
   }

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

   return res;
}
```

Les funcions `calcula_X` tenen com a entrada els vectors que reben com a arguments i amb ells modifiquen el vector `X` indicat. Per exemple, `calcula_v(n,v)` pren com a dades d'entrada els valors de `n` i `v` i modifica el vector `v`.

(a) Dibuixa el graf de dependències de les diferents tasques, incloent en el mateix el cost de cadascuna de les tasques i el volumn de les comunicacions. Suposar que les funcions `calcula_X` tenen un cost de $2n^2$.

Solució: Els costos de comunicacions apareixen a les arestes del graf.



El cost (temps d'execució) de la tasca 4 és:

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

El cost de T_5 és igual al de T_4, i el cost de T_6 és $2n$.

(b) Paral·lelitza'l usant MPI, de manera que els processos MPI disponibles executen les diferents tasques (sense dividir-les en subtasques). Es pot suposar que hi ha almenys 3 processos.

Solució: Hi ha diferents possibilitats de fer l'assignació. Cal tenir en compte que només un dels processos ha de realitzar la lectura. Les tasques 1, 2 i 3 són independents i per tant es poden assignar a 3 processos diferents. El mateix ocorre amb les 4 i 5.

Per exemple, el procés 0 es pot encarregar de llegir, i fer les tasques 1 i 4. El procés 1 pot fer la tasca 2, i el procés 2 les tasques 3, 5 i 6.

Atès que els processos diferents de P_0 no coneixen el valor de `n`, és necessari enviar aquest valor en un missatge separat, i després de la seua recepció ja es pot reservar la memòria necessària amb `malloc`. Aquest missatge és d'una longitud menor, ja que s'envia una variable entera en lloc de tipus `double`. No obstant açò, per simplificar, no s'ha considerat aquesta diferència en el càlcul de costos.

```
double funcio()
{
   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: Llegir els vectors v, w, z, de dimensio n */
      llegir(&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);              /* tasca 1 */
            MPI_Recv(w, n, MPI_DOUBLE, 1, 0, MPI_COMM_WORLD, &status);

            /* tasca 4 (mateix codi del cas sequencial) */
            ...

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

            /* tasca 5 (mateix codi del cas sequencial) */
            ...

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

            /* tasca 6 (mateix codi del cas sequencial) */
            ...

            /* Enviar el resultat de la tasca 6 als demes processos */
            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) Indica el temps d'execució de l'algorisme seqüencial, el de l'algorisme paral·lel, i el speedup que

s'obtindria. Ignorar el cost de la lectura dels vectors.

Solució: Tenint en compte que el temps d'execució de cadascuna de les tasques 1, 2 i 3 és de $2n^2$, mentre que el de les tasques 4 i 5 és de $3n^2$, i el de la tasca 6 és de $2n$, el temps d'execució seqüencial serà la suma d'eixos temps:

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

Temps d'execució paral·lel: temps aritmètic. Serà igual al temps aritmètic del procés que més operacions realitze, que en aquest cas és el procés 2, que fa les tasques 3, 5 i 6. Per tant

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

Temps d'execució paral·lel: temps de comunicacions. Els missatges que es trameten son:

- 2 missatges, del procés 0 als demés, amb el valor de `n`. Cost de $2(t_s + t_w)$.
- 2 missatges, un del procés 0 al 1 amb el vector `w` i un altre del 0 al 2 amb el vector `z`. Cost de $2(t_s + nt_w)$.
- 2 missatges, del procés 1 als demés, amb el vector `w`. Cost de $2(t_s + nt_w)$.
- 1 missatge, del procés 0 al 2, amb el valor de `sv`. Cost de $(t_s + t_w)$
- 2 missatges, del procés 2 als demés, amb el valor de `res`. Cost de $2(t_s + t_w)$

Per tant, el cost de comunicacions serà:

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

Temps d'execució paral·lel total:

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

Speedup:

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

Qüestió 1–2

Implementa una funció que, a partir d'un vector de dimensió `n`, distribuït entre `p` processos de forma cíclica per blocs, realitze les comunicacions necessàries perquè tots els processos acaben amb una còpia del vector complet. Nota: utilitza únicament comunicació punt a punt.

La capçalera de la funció serà:

```
void comunica_vector(double vloc[], int n, int b, int p, double w[])
/* vloc: part local del vector v inicial
   n: dimensio global del vector v
   b: grandària de bloc emprat en la distribucio del vector v
   p: nombre de processos
   w: vector de longitud n, on ha de guardar-se una còpia del vector v complet
 */
```

Solució: Assumim que `n` es múltiple del tamany de bloc `b` (o siga, tots els blocs tenen tamany `b`).

```
void comunica_vector(double vloc[], int n, int b, int p, double w[])
/* vloc: part local del vector v inicial
   n: dimensio global del vector v
   b: grandària de bloc emprat en la distribucio del vector v
   p: nombre de processos
   w: vector de longitud n, on ha de guardar-se una còpia del vector v complet
 */
{
   int i, rank, rank_pb, rank2;
   int ib, ib_loc;       /* Index de bloc */
   int num_blq=n/b;      /* Numero de blocs */
   MPI_Status status;

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   for (ib=0; ib<num_blq; ib++) {
      rank_pb = ib%p;           /* propietari del bloc */
      if (rank==rank_pb) {
         ib_loc = ib/p;         /* index local del bloc */
         /* Enviar bloc ib a tots els processos excepte a mi mateix */
         for (rank2=0; rank2<p; rank2++) {
            if (rank!=rank2) {
               MPI_Send(&vloc[ib_loc*b], b, MPI_DOUBLE, i, 0, MPI_COMM_WORLD);
            }
         }
         /* Copiar bloc ib en el meu propi vector local */
         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);
      }
   }
}
```

Qüestió 1–3

Es desitja aplicar un conjunt de T tasques sobre un vector de números reals de grandària `n`. Aquestes tasques han d'aplicar-se seqüencialment i en ordre. La funció que les representa té la següent capçalera:

```
void tasca(int tipus_tasca, int n, double *v);
```

on `tipus_tasca` identifica el nombre de tasca d'1 fins a T. No obstant açò, aquestes tasques seran aplicades a m vectors. Aquests vectors estan emmagatzemats en una matriu A en el procés mestre on cada fila representa un d'aqueixos m vectors.

Implementar un programa paral·lel en MPI en forma de *Pipeline* on cada procés ($P_1 \dots P_{p-1}$) executarà una de les T tasques ($T = p - 1$). El procés mestre (P_0) es limitarà a alimentar el pipeline i arreplegar cadascun dels vectors (i emmagatzemar-los de nou en la matriu A) una vegada hagen passat per tota la canonada. Utilitzeu un missatge buit identificat mitjançant una etiqueta per a acabar el programa (supose's que els esclaus desconeixen el nombre m de vectors).

Solució: La part de codi que pot servir per a implementar el pipeline proposat podria ser la següent:

```
#define TASCA_TAG  123
#define FI_TAG     1
int continuar,num;
MPI_Status stat;
if (!rank) {
  for (i=0;i<m;i++) {
    MPI_Send(&A[i*n], n, MPI_DOUBLE, 1, TASCA_TAG, MPI_COMM_WORLD);
  }
  MPI_Send(0, 0, MPI_DOUBLE, 1, FI_TAG, MPI_COMM_WORLD);
  for (i=0;i<m;i++) {
    MPI_Recv(&A[i*n], n, MPI_DOUBLE, p-1, TASCA_TAG, MPI_COMM_WORLD, &stat);
  }
  MPI_Recv(0, 0, MPI_DOUBLE, p-1, FI_TAG, MPI_COMM_WORLD, &stat);
} else {
  continuar = 1;
  while (continuar) {
    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 == TASCA_TAG) {
      tasca(rank, n, A);
    } else {
      continuar = 0;
    }
    MPI_Send(A, num, MPI_DOUBLE, (rank+1)%p, stat.MPI_TAG, MPI_COMM_WORLD);
  }
}
```

Qüestió 1–4

En un programa paral·lel executat en p processos, es té un vector x de dimensió n distribuït per blocs, i un vector y replicat en tots els processos. Implementar la següent funció, la qual ha de sumar la part local del vector x amb la part corresponent del vector y, deixant el resultat en un vector local z.

```
void suma(double xloc[], double y[], double z[], int n, int p, int pr)
/* pr és l'index del procés local */
```

Solució:

```
void suma(double xloc[], double y[], double z[], int n, int p, int pr)
/* pr és l'index del procés local */
{
  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];
  }
}
```

Qüestió 1–5

La distància de Levenshtein proporciona una mesura de similitud entre dues cadenes. El següent codi seqüencial utilitza aquesta distància per a calcular la posició en la qual una subcadena és més similar a una altra cadena, assumint que les cadenes es lligen des d'un fitxer de text.

Exemple: si la cadena `ref` conté `"aafsdluqhqwBANANAqewrqerBANAfqrqerqrABANArqwrBAANANqwe"` i la cadena `str` conté `"BANAN"`, el programa mostrarà que la cadena `"BANAN"` es troba en la menor diferència en la posició 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("Distància %d per a %s en %d\n", mindist, str, pos);
}
fclose(f2);
```

(a) Completa la següent implementació paral·lela MPI d'aquest algorisme segons el model mestre-treballadors.

```
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 processos actius\n", count);
    /*
      COMPLETAR
      - rebre tres missatges del mateix procés
      - llegir nova linia del fitxer i enviar-la
      - si fitxer acabat, enviar missatge de terminació
    */
  } 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] Missatge rebut (%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] envia: %d, %d, i %s a 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] rep missatge amb etiqueta %d\n", rank, status.MPI_TAG);
        rc = 1;
      }
```

```
    } while (!rc);
}
```

Solució:

```
do {
  printf("%d processos actius\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("De [%d]: Distància %d per a %s en %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("Enviant missatge de terminació a %d\n", status.MPI_SOURCE);
    MPI_Send(&c, 1, MPI_CHAR, org, TAG_END, MPI_COMM_WORLD);
  }
} while (count>1);
```

(b) Calcula el cost de comunicacions de la versió paral·lela desenvolupada depenent de la grandària del problema n i del nombre de processos p.

Solució: En la versió proposta, el cost de comunicació és degut a quatre conceptes principals:

- Difusió de la referència ($lr + 1$ bytes): $(t_s + t_w \cdot (lr + 1)) \cdot (p - 1)$
- Missatge individual per cada seqüència ($ls_i + 1$ bytes): $(t_s + t_w \cdot (ls_i + 1)) \cdot n$
- Tres missatges per a la resposta de cada seqüència (dos enters més $ls_i + 1$ bytes): $(t_s + t_w \cdot (ls_i + 1)) \cdot n + 2 \cdot n \cdot (t_s + 4 \cdot t_w)$
- Missatge de terminació (1 byte): $(t_s + t_w) \cdot (p - 1)$

Per tant, el cost total es pot aproximar per $2 \cdot n \cdot t_s + t_w \cdot (9 \cdot n + m)$.

Qüestió 1–6

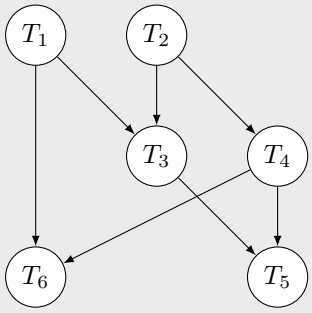
Es vol paral·lelitzar el següent codi mitjançant MPI. Suposem que es disposa de 3 procesos.

```
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);
```

Totes les funcions lligen i modifiquen ambdos arguments, també els vectors. Suposem que els vectors `a`, `b` i `c` estan emmagatzemats en P_0, P_1 i P_2, respectivament, i son massa grans per a poder ser enviats eficientement d'un procés a un altre.

(a) Dibuixa el graf de dependències de les diferents tasques, indicant quina tasca s'assigna a cada procés.

Solució: El graf de dependències és el següent:



Degut a la restricció d'on estan situats els vectors, farem la següent assignació: T_1 i T_6 en P_0, T_2 i T_3 en P_1, T_4 i T_5 en P_2.

(b) Escriu el codi MPI que resol el problema.

Solució:

```
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);
}
```

Qüestió 1–7

El següent fragment de codi és incorrecte (des del punt de vista semàntic, no perquè hi haja un error en els arguments). Indica per què i proposa dues solucions diferents.

```
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);
```

Solució: El codi realitza una comunicació en anell, on cada procés envia `N` dades de tipus enter al procés de la seua dreta (i l'últim procés envia al procés 0). És incorrecte perquè es produeix un interbloqueig, ja que la primitiva `MPI_Ssend` és síncrona i, per tant, tots els processos quedaran esperant a que es realitze la recepció en el procés destí. No obstant açò, cap procés aplegarà a la primitiva `MPI_Recv`, per la qual cosa l'execució no progressarà. L'ús de l'enviament estàndard `MPI_Send` no resol el problema, ja que no es garanteix que no es faça també de forma síncrona.

Una solució seria implementar un protocol parells-imparells, reemplaçant les dues últimes línies per:

```
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);
}
```

Una altra solució seria l'ús d'una primitiva combinada:

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

També es resoldria utilitzant primitives no bloquejants, per exemple:

```
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);
```

Qüestió 1–8

Es vol implementar el càlcul de la ∞-norma d'una matriu quadrada, que s'obté com el màxim de les sumes dels valors absoluts dels elements de cada fila, $\max_{i=0}^{n-1} \left\{ \sum_{j=0}^{n-1} |a_{i,j}| \right\}$. Per a açò, es proposa un esquema mestre-treballadors. A continuació, es mostra la funció corresponent al mestre (el procés amb identificador 0). La matriu s'emmagatzema per files en un array uni-dimensional, i suposem que és molt dispersa (té molts zeros), per la qual cosa el mestre envia únicament els elements no nuls (funció `comprimeix`).

```
int comprimeix(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 mestre(double *A,int n)
{
  double buf[n];
```

```
  double norma=0.0,valor;
  int fila,complets=0,size,i,k;
  MPI_Status status;
  MPI_Comm_size(MPI_COMM_WORLD,&size);
  for (fila=0;fila<size-1;fila++) {
    if (fila<n) {
      k = comprimeix(A, n, fila, buf);
      MPI_Send(buf, k, MPI_DOUBLE, fila+1, TAG_FILA, MPI_COMM_WORLD);
    } else
      MPI_Send(buf, 0, MPI_DOUBLE, fila+1, TAG_END, MPI_COMM_WORLD);
  }
  while (complets<n) {
    MPI_Recv(&valor, 1, MPI_DOUBLE, MPI_ANY_SOURCE, TAG_RESU,
             MPI_COMM_WORLD, &status);
    if (valor>norma) norma=valor;
    complets++;
    if (fila<n) {
      k = comprimeix(A, n, fila, buf);
      fila++;
      MPI_Send(buf, k, MPI_DOUBLE, status.MPI_SOURCE, TAG_FILA, MPI_COMM_WORLD);
    } else
      MPI_Send(buf, 0, MPI_DOUBLE, status.MPI_SOURCE, TAG_END, MPI_COMM_WORLD);
  }
  return norma;
}
```

Implementa la part dels processos treballadors, completant la següent funció:

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

Nota: Per al valor absolut es pot usar
`double fabs(double x)`
Recorda que `MPI_Status` conté, entre altres, els camps `MPI_SOURCE` i `MPI_TAG`.

Solució:

```
void treballador(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_FILA) {
    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);
  }
}
```

Qüestió 1–9

Volem mesurar la latència d'un anell de `p` processos en MPI, entenent per latència el temps que tarda un missatge de grandària 0 a circular entre tots els processos. Un anell de `p` processos MPI funciona de la següent manera: P_0 envia el missatge a P_1, quan aquest el rep, el reenvía a P_2, i així successivament fins que arriba a P_{p-1} que l'enviarà a P_0. Escriu un programa MPI que implemente aquest esquema de comunicació i mostre la latència. És recomanable fer que el missatge done més d'una volta a l'anell, i després traure el temps mitjà per volta, per a obtenir una mesura més fiable.

Solució:

```
#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("Soc %d, el meu proces esquerre es %d i el dret es %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("Latencia d'un anell de %d processos: %f\n", size, (t2-t1)/REPS);
  }
  MPI_Finalize();
  return 0;
}
```

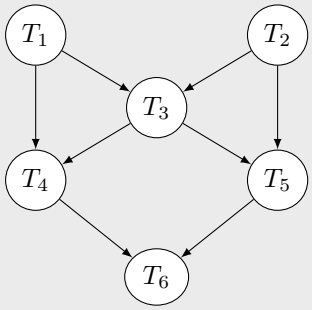
Qüestió 1–10

Donada la següent funció, on suposem que les funcions `T1`, `T3` i `T4` tenen un cost de n i les funcions `T2` i `T5` de $2n$, on n és un valor constant.

```
double exemple(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) Dibuixa el graf de dependències i calcula el cost seqüencial.

Solució: El graf de dependències es mostra en la següent figura:



El cost seqüencial és: $t(n) = n + 2n + n + n + 2n + 4 \approx 7n$

Nota: els últims 4 flops es refereixen a les operacions realitzades en la pròpia funció `exemple` que no correspon a cap de les funcions invocades.

(b) Paral·lelitza-la usant MPI amb dos processos. Tots dos processos invoquen la funció amb el mateix valor dels arguments `i`, `j` (no és necessari comunicar-los). El valor de retorn de la funció ha de ser correcte en el procés 0 (no és necessari que estiga en tots dos processos).

Solució: Per a equilibrar la càrrega, proposem una solució en la qual el procés 1 realitza `T2` i `T5`, i la resta de tasques s'assignen al procés 0.

```
double exemple(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) Calcula el temps d'execució en paral·lel (càlcul i comunicacions) i l'speedup amb dos processos.

Solució: El temps paral·lel de la implementació proposada s'ha de calcular a partir del cost associat al camí crític del graf de dependències, corresponent a $T_2 - T_3 - T_5 - T_6$.

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

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

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

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

L'speedup:

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

Qüestió 1–11

Escriu una funció amb la següent capçalera, la qual ha de fer que els processos amb índexs `proc1` i `proc2` intercanvien el vector `x` que es passa com a argument, mentre que en la resta de processos el vector `x` no patirà cap canvi.

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

Has de tenir en compte el següent:

- S'ha d'evitar la possibilitat d'interbloquejos.
- S'ha de fer sense utilitzar les funcions `MPI_Sendrecv`, `MPI_Sendrecv_replace` i `MPI_Bsend`.
- Declara les variables que consideres necessàries.
- Se suposa que `N` és una constant definida prèviament, i que `proc1` i `proc2` són índexs de processos vàlids (en el rang entre 0 i el nombre de processos menys un).

Solució:

```
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];
}
```

Qüestió 1–12

La següent funció mostra per pantalla el màxim d'un vector `v` de `n` elements i la seua posició:

```
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("Màxim: %f. Posició: %d\n", max, posmax);
}
```

Escriu una versió paral·lela MPI amb la següent capçalera, on els arguments `rank` i `np` han sigut obtinguts mitjançant `MPI_Comm_rank` i `MPI_Comm_size`, respectivament.

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

La funció ha d'assumir que l'array `v` del procés 0 contindrà inicialment el vector, mentre que en la resta de processos aquest array podrà usar-se per a emmagatzemar la part local que corresponga. Hauran de comunicar-se les dades necessàries de manera que el càlcul del màxim es repartisca de forma equitativa entre tots els processos. Finalment, només el procés 0 ha de mostrar el missatge per pantalla. S'han d'utilitzar operacions de comunicació punt a punt (no col·lectives).

Nota: es pot assumir que `n` és múltiple del nombre de processos.

Solució:

```
void func_par(double v[], int n, int rank, int np)
{
  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;    /* Convertir posmax en índex global */

  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("Màxim: %f. Posició: %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);
    }
  }
```

Qüestió 1–13

Volem implementar una funció per a distribuir una matriu quadrada entre els processos d'un programa MPI, amb la següent capçalera:

```
void comunica(double A[N][N], double Aloc[][N], int proc_fila[N], int root)
```

La matriu `A` es troba inicialment en el procés `root`, i ha de distribuir-se per files entre els processos, de manera que cada fila `i` ha d'anar al procés `proc_fila[i]`. El contingut del array `proc_fila` és vàlid en tots els processos. Cada procés (inclós el `root`) ha d'emmagatzemar les files que li corresponguen en la matriu local `Aloc`, ocupant les primeres files (o siga, si a un procés se li assignen `k` files, aquestes han de quedar emmagatzemades en les primeres `k` files de `Aloc`).

Exemple per a 3 processos:

`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_fila`

0
2
0
1
1

`Aloc` en P_0

11	12	13	14	15
31	32	33	34	35

`Aloc` en P_1

41	42	43	44	45
51	52	53	54	55

`Aloc` en P_2

21	22	23	24	25

(a) Escriu el codi de la funció.

Solució:

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

   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   iloc=0;
   for (i=0; i<N; i++) {
      /* Tractar la fila i */
      if (rank==root) {
         if (proc_fila[i]==root) {      /* Copia local de la fila */
            for (j=0; j<N; j++) Aloc[iloc][j] = A[i][j];
```

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

(b) En un cas general, es podria usar el tipus de dades *vector* de MPI (`MPI_Type_vector`) per a enviar a un procés totes les files que li toquen mitjançant un sol missatge? Si es pot, escriu les instruccions per a definir-lo. Si no es pot, justifica per què.

Solució: No es pot, perquè les files que li toquen a un procés no tenen, en general, una separació constant entre elles.

Qüestió 1–14

Desenvolupa un programa *ping-pong*.

Es desitja un programa paral·lel, per a ser executat en 2 processos, que repetisca 200 vegades l'enviament del procés 0 al procés 1 i la devolució del procés 1 al 0, d'un missatge de 100 enters. Al final s'haurà de mostrar per pantalla el temps mitjà d'enviament d'un enter, calculat a partir del temps d'enviar/rebre tots eixos missatges.

El programa pot començar així:

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

(a) Implementa el programa indicat.

Solució:

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

#define V 200
#define N 100

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

  MPI_Init(&argc,&argv);

  MPI_Comm_rank(MPI_COMM_WORLD,&jo);

  t = MPI_Wtime();
```

```
    for (i=0;i<V;i++)
      if (jo==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 (jo==0) printf("Temps mitjà d'enviament d'un enter: %g seg.\n",t/(2*V)/N);

    MPI_Finalize();
    return 0;
  }
```

(b) Calcula el temps de comunicacions (teòric) del programa.

Solució:

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

Qüestió 1–15

En un programa paral·lel es disposa d'un vector distribuït per blocs entre els processos, de manera que cada procés té el seu bloc en l'array `vloc`.

Implementa una funció que desplace els elements del vector una posició a la dreta, fent a més que l'últim element passe a ocupar la primera posició. Per exemple, si tenim 3 processos, donat l'estat inicial:

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

L'estat final seria:

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

La funció hauria d'evitar possibles interbloquejos. La capçalera de la funció serà:

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

on `mb` és el nombre d'elements de `vloc` (suposarem que `mb` > 1).

Solució:

```
void despl(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;
```

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

    /* Comunicació amb els veïns */
    MPI_Sendrecv(&elem, 1, MPI_DOUBLE, sig, 0,
                 &vloc[0], 1, MPI_DOUBLE, prev, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

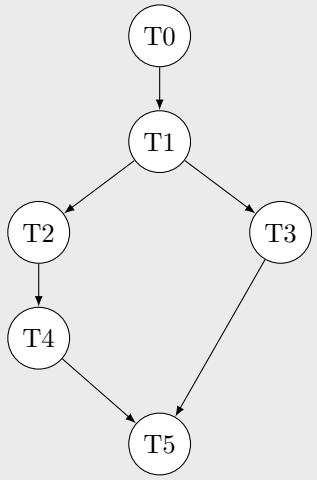
Qüestió 1–16

En el següent programa seqüencial, on indiquem amb comentaris el cost computacional de cada funció, totes les funcions invocades modifiquen únicament el primer argument. Observeu que `A`, `D` i `E` són vectors, mentres que `B` i `C` són matrius.

```
#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) Obteniu el graf de dependències.

Solució:



(b) Implementeu una versió paral·lela amb MPI, tenint en compte els següents aspectes:

- Utilitzeu el nombre més apropiat de processos paral·lels perquè l'execució siga el més ràpida possible, mostrant un missatge d'error en cas de que el nombre de processos en execució no coincidisca amb ell. Només el procés P_0 ha de realitzar les operacions `read` i `printf`.

- Presteu atenció a la grandària dels missatges i utilitzeu les tècniques d'agrupament i replicació, etc. si fora convenient.
- Realitzeu la implementació del programa complet.

Solució: Utilitzarem dos processos i aplicarem replicació en la tasca `generate` donat que el cost d'enviament de la matriu quadrada `B` és superior al de l'enviament del vector `A`, i la tasca `generate` no pot fer-se en paral·lel amb cap altra.

```
#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) Calculeu el cost seqüencial, cost paral·lel, speed-up i eficiència.

Solució:

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

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

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

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

Qüestió 1–17

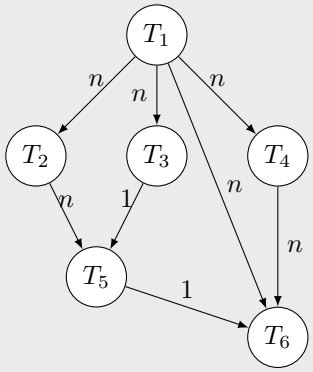
Es vol paral·lelitzar el següent codi mitjançant MPI.

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

  /* Llegir els vector x, y, z, de dimensio n */
  llegir(n, x, y, z);          /* tasca 1 */

  normalitza(n,x);             /* tasca 2 */
  beta = obtindre(n,y);        /* tasca 3 */
  normalitza(n,z);             /* tasca 4 */

  /* tasca 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]; }

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

Suposem que es disposa de 3 processos, dels quals només un ha de cridar a la funció `llegir`. Es pot assumir que el valor de `n` està disponible en tots els processos. El resultat final (`z`) pot quedar emmagatzemat en un qualsevol dels 3 processos. La funció `llegir` modifica els tres vectors, la funció `normalitza` modifica el seu segon argument i la funció `obtindre` no modifica cap dels seus arguments.

(a) Dibuixeu 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 volumn de dades que es transfereix entre cada par de tasques dependents, el que s'ha de tindre en compte per a seleccionar la millor assignació possible de tasques a processos.

(b) Escriviu el codi MPI que resol el problema utilitzant una assignació que maximitze el paral·lelisme i minimitze el cost de comunicacions.

Solució: Una assignació de tasques que compleix els requisits és $P_0 : T_1, T_4, T_6$; $P_1 : T_3$; $P_2 : T_2, T_5$.

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

  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  if (rank == 0) {
    /* Llegir els vectors x, y, z, de dimensio n */
    llegir(n, x, y, z);          /* tasca 1 */
    MPI_Send(x, n, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD);
    MPI_Send(y, n, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD);
    normalitza(n,z);             /* tasca 4 */
    MPI_Recv(&alpha, 1, MPI_DOUBLE, 2, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /* tasca 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 = obtindre(n,y);        /* tasca 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);
    normalitza(n,x);             /* tasca 2 */
    MPI_Recv(&beta, 1, MPI_DOUBLE, 1, 123, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    /* tasca 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 );
  }
}
```

Qüestió 1–18

Escriviu un programa paral·lel amb MPI en el qual el procés 0 llija una matriu de $M \times N$ nombres reals de disc (amb la funció `read_mat`) i eixa matriu vaja passant d'un procés a un altre fins a arribar a l'últim, que li la tornarà al procés 0. El programa haurà de medir el temps total d'execució, sense comptar la lectura de disc, i mostrar-lo per pantalla.

Utilitzeu esta capçalera per a la funció principal:

```
int main(int argc,char *argv[])
```

i teniu en compte que la funció de lectura de la matriu té esta capçalera:

```
void read_mat(double A[M][N]);
```

(a) Escriviu el programa demanat.

Solució:

```
#include <stdio.h>
#include <mpi.h>
#define M 1000
#define N 1000
int main(int argc,char *argv[]) {
  int id,np;
  double A[M][N],t1,t2;
  MPI_Init(&argc,&argv);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  MPI_Comm_size(MPI_COMM_WORLD,&np);
  if (id==0) read_mat(A);
  t1=MPI_Wtime();
  if (id==0) {
    MPI_Send(A,M*N,MPI_DOUBLE,1,1234,MPI_COMM_WORLD);
    MPI_Recv(A,M*N,MPI_DOUBLE,np-1,1234,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  } else {
    MPI_Recv(A,M*N,MPI_DOUBLE,id-1,1234,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    MPI_Send(A,M*N,MPI_DOUBLE,(id+1)%np,1234,MPI_COMM_WORLD);
  }
  t2=MPI_Wtime();
  if (id==0) printf("Temps: %.2f segons.\n",t2-t1);
  MPI_Finalize();
  return 0;
}
```

(b) Indiqueu el cost teòric total de les comunicacions.

Solució:

$$t = p \cdot (t_s + MNt_w)$$

2 Comunicació col·lectiva

Qüestió 2–1

El següent fragment de codi permet calcular el producte d'una matriu quadrada per un vector, tots dos de la mateixa dimensió `N`:

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

Escriu un programa MPI que realitze el producte en paral·lel, tenint en compte que el procés P_0 obté inicialment la matriu `A` i el vector `v`, realitza una distribució de `A` per blocs de files consecutives sobre tots els processos i envia `v` a tots. Així mateix, al final P_0 ha d'obtenir el resultat.

Nota: Per a simplificar, es pot assumir que `N` és divisible pel nombre de processos.

Solució: Definim una matriu auxiliar `B` i un vector auxiliar `y`, que contindran les porcions locals de `A` i `x` en cadascun dels processos. Tant `B` com `y` tenen `k=N/p` files, però per a simplificar s'han dimensionat a `N` files ja que el valor de `k` és desconegut en temps de compilació (una solució eficient en termes de memòria reservaria aquestes variables amb `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) llegir(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) escriure(x);
```

Qüestió 2–2

El següent fragment de codi calcula la norma de Frobenius d'una matriu quadrada obtinguda a partir de la funció `llegirmat`.

```
int i, j;
double s, norm, A[N][N];
llegirmat(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);
```

Implementa un programa paral·lel usant MPI que calcule la norma de Frobenius, de manera que el procés P_0 llija la matriu `A`, la repartisca segons una distribució cíclica de files, i finalment obtinga el resultat `s` i l'imprimisca en la pantalla.

Nota: Per a simplificar, es pot assumir que `N` és divisible pel nombre de processos.

Solució: Utilitzem una matriu auxiliar `B` perquè cada procés emmagatzeme la seua part local de `A` (només s'utilitzen les `k` primeres files). Per a la distribució de la matriu, es fan `k` operacions de repartiment, una per cada bloc de `p` files.

```
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) llegirmat(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);
}
```

Qüestió 2–3

Es vol paral·lelitzar el següent programa usant MPI.

```
double *llig_dades(char *nom, int *n) {
  ...  /* lectura des de fitxer de les dades */
  /* retorna un punter a les dades i el nombre de dades en n */
}

double processa(double x) {
  ...  /* funció costosa que fa un càlcul depenent de x */
}

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

  a = llig_dades("dades.txt",&n);
  res = 0.0;
  for (i=0; i<n; i++)
    res += processa(a[i]);

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

Coses a tenir en compte:

- Només el procés 0 ha de cridar a `llig_dades` (només ell llegirà del fitxer).
- Només el procés 0 ha de mostrar el resultat.
- Cal repartir els `n` càlculs entre els processos disponibles usant un repartiment per blocs. Caldrà enviar a cada procés la seua part de `a` i arreplegar la seua aportació al resultat `res`. Es pot suposar que `n` és divisible pel nombre de processos.

(a) Realitza una versió amb comunicació punt a punt.

Solució:

```
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 = llig_dades("dades.txt",&n);

    /* Difondre el tamany del problema (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;  /* Assumim que n és múltiple de np */

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

    /* Repartir la a entre tots els processos (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 += processa(a[i]);

    /* Recollida de resultats (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("Resultat: %.2f\n",res);
    free(a);

    MPI_Finalize();
    return 0;
  }
```

(b) Realitza una versió utilitzant primitives de comunicació col·lectiva.

Solució: Només cal canviar en el codi anterior les zones marcades amb (1), (2) i (3) per:

```
    /* Difondre la grandària del problema (1) */
```

```
MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);

/* Repartir la a entre tots els processos (2) */
MPI_Scatter(a, nb, MPI_DOUBLE, b, nb, MPI_DOUBLE, 0, MPI_COMM_WORLD);

/* Recollida de resultats (3) */
aux = res;
MPI_Reduce(&aux, &res, 1, MPI_DOUBLE, MPI_SUM, 0, MPI_COMM_WORLD);
```

En l'*scatter* s'ha utilitzat una variable auxiliar `b`, ja que no està permés usar el mateix buffer per a enviament i recepció (excepte en el cas especial en que s'utilitza `MPI_IN_PLACE`). Faltaria canviar `a` per `b` en les crides a `malloc`, `free` i `processa`.

Qüestió 2–4

Desenvolupa un programa usant MPI que jugue al següent joc:

1. Cada procés s'inventa un número i li'l comunica a la resta.
2. Si tots els processos han pensat el mateix número, s'acaba el joc.
3. Si no, es repeteix el procés (es torna a 1). Si ja hi ha hagut 1000 repeticions, es finalitza amb un error.
4. Al final cal indicar per pantalla (una sola vegada) quantes vegades s'ha hagut de repetir el procés perquè tots pensaren el mateix número.

Es disposa de la següent funció per a inventar els números:

```
int pensa_un_numero(); /* retorna un número aleatori */
```

Utilitza operacions de comunicació col·lectiva de MPI per a totes les comunicacions necessàries.

Solució:

```
int p,np;
int num,*vnum,cont,iguals,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 = pensa_un_numero();
  MPI_Allgather(&num, 1, MPI_INT, vnum, 1, MPI_INT, MPI_COMM_WORLD);
  iguals = 0;
  for (i=0; i<np; i++) {
    if (vnum[i]==num) iguals++;
  }
} while (iguals!=np && cont<1000);

if (!p) {
  if (iguals==np)
    printf("Han pensat el mateix número en la iteració %d.\n",cont);
  else
```

```
        printf("ERROR: Més de 1000 vegades i no coincideixen els números.\n");
    }
    free(vnum);
```

Qüestió 2–5

Es pretén implementar un generador de números aleatoris paral·lel. Donats p processos MPI, el programa funcionarà de la següent forma: tots els processos van generant una seqüència de números fins que P_0 els indica que paren. En eixe moment, cada procés enviarà P_0 al seu últim número generat i P_0 combinarà tots eixos números amb el número que ha generat ell. En pseudocodi seria una cosa així:

```
n = inicial(id)
si id=0
  per a i=1 fins a 100
    n = següent(n)
  fper
  envia missatge d'avís a processos 1..np-1
  rep m[k] de proces k per a k=1..np-1
  n = combina(n,m[k]) per a k=1..np-1
si no
  n = inicial
  mentre no arriba missatge de 0
    n = següent(n)
  fmentre
  envia n a 0
fsi
```

Implementar en MPI un esquema de comunicació asíncrona per a aquest algorisme, utilitzant `MPI_Irecv` i `MPI_Test`. La recollida de resultats pot realitzar-se amb una operació col·lectiva.

Solució: En el missatge d'avís no és necessari enviar cap dada, per la qual cosa el buffer serà un punter nul i la longitud 0.

```
MPI_Comm_rank(MPI_COMM_WORLD, &id);
MPI_Comm_size(MPI_COMM_WORLD, &np);
n = inicial(id);
if (id==0) {
  for (i=1;i<=100;i++) n = seguent(n);
  for (k=1;k<np;k++) {
    MPI_Send(0, 0, MPI_INT, k, 1, MPI_COMM_WORLD);
  }
} else {
  MPI_Irecv(0, 0, MPI_INT, 0, 1, MPI_COMM_WORLD, &req);
  do {
    n = seguent(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 = combina(n,m[k]);
}
```

Qüestió 2–6

Donat el següent fragment de programa que calcula un valor approximat per al número 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);
```

Implementar una versió en MPI que permeta el seu càlcul en paral·lel.

Solució: La paral·lelització és senzilla en aquest cas atès que el programa és molt paral·lelitzable. Consisteix en la utilització correcta de la rutina `MPI_Reduce`. Cada procés només ha de calcular la quantitat de nombres aleatoris que ha de generar i generar-los comptabilitzant els que cauen dins del cercle. Es multiplica el valor de la llavor per l'identificador del procés perquè la seqüència de nombres aleatòria siga diferent en cada procés.

```
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);
}
```

Qüestió 2–7

La ∞-norma d'una matriu es defineix com el màxim de les sumes dels valors absoluts dels elements de cada fila: $\max_{i=1..n} \left\{ \sum_{j=0}^{m-1} |a_{i,j}| \right\}$. El següent codi seqüencial implementa aquesta operació per al cas d'una matriu quadrada.

```
#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) Implementar una versió paral·lela mitjançant MPI utilitzant operacions col·lectives en la mesura que siga possible. Assumir que la grandària del problema és múltiple exacte del nombre de processos. La matriu està inicialment emmagatzemada en P_0 i el resultat deu quedar també en P_0.

Nota: es suggerix utilitzar la següent capçalera per a la funció paral·lela, on `Alocal` és una matriu que se suposa ja reservada en memòria, i que pot ser utilitzada per la funció per a emmagatzemar la part local de la matriu `A`.

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

Solució:

```
#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) Obtindre el cost computacional i de comunicacions de l'algorisme paral·lel. Assumir que l'operació `fabs` té un cost menyspreable, així com les comparacions.

Solució: Denotem per n la grandària del problema (`N`). El cost inclou tres etapes:

- Cost del repartiment: $(p-1) \cdot \left(t_s + n \cdot \frac{n}{p} \cdot t_w\right)$
- Cost del processament de $\frac{n}{p}$ files: $\frac{n}{p} \cdot n = \frac{n^2}{p}$
- Cost de la reducció (implementació trivial): $(p-1) \cdot (t_s + t_w)$

Per tant, el cost total és aproximadament: $2 \cdot p \cdot t_s + n^2 \cdot t_w + \frac{n^2}{p}$

(c) Calcular el speed-up i l'eficiència quan la grandària del problema tendeix a infinit.

Solució: El cost computacional de la versió seqüencial és aproximadament n^2. Per tant, el speed-up és $S(n,p) = \frac{n^2}{2 \cdot p \cdot t_s + n^2 \cdot t_w + \frac{n^2}{p}}$ y l'eficiència $E(n,p) = \frac{n^2}{2 \cdot p^2 \cdot t_s + p \cdot n^2 \cdot t_w + n^2}$.
Els valors asimptòtics del speed-up i l'eficiència quan la grandària del problema tendeix a infinit son els següents:

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

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

Qüestió 2–8

Siga el següent codi:

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

on la funció `processar` té la següent capçalera:

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

amb tots els arguments només d'entrada.

(a) Indica el seu cost teòric (en flops) suposant que el cost de la funció `processar` és $2n$ flops.

Solució: Cost seqüencial: $2n^2m$ flops.

(b) Paral·lelitza el codi amb MPI, justificant la resposta. Es suposa que n és la dimensió dels vectors `v` i `w`, però també el nombre de processos MPI. La variable `p` conté l'identificador de procés. El procés `p=0` és l'únic que té el valor inicial del vector `v`. Es valora la manera més eficient de realitzar aquesta paral·lelització. Açò és utilitzar les rutines MPI adequades de manera que el nombre d'aquestes siga mínim.

Solució: En primer lloc, el procés 0 difon el vector a la resta de procesos ja que la funció `processar` necessita aquest vector. El bucle extern no es pot paral·lelitzar. Paral·lelitzem, per tant, els bucles interns. En una iteració (`i`), el procés `p` s'encarregarà d'executar la funció `processar` i emmagatzemar el resultat en la variable `a`. L'actualització del vector `v` en el segon bucle correspon a un repartiment en el que cada procés envia a tots els demés la dada calculada en la variable `a`.

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

(c) Indica el cost de comunicacions suposant que els nodes estan conectats en una topologia de tipus bus.

Solució: El cost d'una difusió de n elements en un bus és de $\beta + n\tau$. El cost de l'operació de recollida d'un element de cada procés per part de tots els processos és de $n(\beta + \tau)$. El cost total, tenint en compte el nombre d'iteracions del bucle exterior, és de

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

(d) Indica l'eficiència assolible tenint en compte que tant m com n són grans.

Solució: El speedup S_n es calcula com la ràtio entre el temps seqüencial i el paral·lel (suposant n processos):

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

on s'ha tingut en compte que el cost de la difusió és menyspreable (enfront de valors de m i n grans). Per tant, l'eficiència E per a n processos seria

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

Qüestió 2–9

El següent programa compta el nombre d'ocurrències d'un valor en una matriu.

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

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

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

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

(a) Fes una versió paral·lela MPI del programa anterior, utilitzant operacions de comunicació col·lectiva quan siga possible. La funció `llegir` haurà de ser invocada solament pel procés 0. Es pot assumir

que DIM és divisible entre el nombre de processos. Nota: cal escriure el programa complet, incloent la declaració de les variables i les crides necessàries per a iniciar i tancar MPI.

Solució: La matriu A es distribueix per blocs de files consecutives entre els processos.

```
...
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) llegir(A,&x);

  /* Distribuir dades */
  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);

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

  /* Recollir resultat global en proc 0 */
  MPI_Reduce(&cont_loc, &cont, 1, MPI_INT, MPI_SUM, 0, MPI_COMM_WORLD);
  if (rank==0) printf("%d ocurrencies\n", cont);

  MPI_Finalize();
  return 0;
}
```

(b) Calcula el temps d'execució paral·lel, suposant que el cost de comparar dos nombres reals és d'1 flop. Nota: per al cost de les comunicacions, suposar una implementació senzilla de les operacions col·lectives

Solució: Per comoditat, anomenem n a la dimensió de la matriu (DIM).
El cost paral·lel serà la suma del cost aritmètic més el cost de comunicacions. El primer és:

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

Respecte a les comunicacions, suposarem que la difusió o *broadcast* realitza l'enviament de $p-1$ missatges (des del procés arrel a cadascun dels restants), i el mateix ocorre amb el *scatter* i amb la reducció. Per tant, el cost de comunicacions serà:

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

Amb la qual cosa, el cost paral·lel serà:

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

Qüestió 2–10

(a) Implementa mitjançant comunicacions col·lectives una funció en MPI que sume dues matrius quadrades `a` i `b` i deixe el resultat en `a`, tenint en compte que les matrius `a` i `b` es troben emmagatzemades en la memòria del process P_0 i el resultat final també haurà d'estar en P_0. Suposarem que el nombre de files de les matrius (`N`, constant) és divisible entre el nombre de processos. La capçalera de la funció és:

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

Solució:

```
void suma_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) Determina el temps paral·lel, l'speed-up i l'eficiència de la implementació detallada en l'apartat anterior, indicant breument per al seu càlcul com es realitzarien cadascuna de les operacions col·lectives (nombre de missatges i grandària de cadascun d'ells). Pots assumir una implementació senzilla (no òptima) d'aquestes operacions de comunicació.

Solució:

Temps seqüencial: N^2 flops.

Temps paral·lel, suposant que tenim p processos:

El cost de repartir una matriu quadrada d'ordre N mitjançant `MPI_Scatter` (suposant un algorisme trivial) es correspon al cost de l'enviament de $p-1$ missatges de grandària igual al nombre d'elements de cada bloc, és a dir $\frac{N}{p}N = \frac{N^2}{p}$ elements. Per tant, el cost de distribuir les matrius `a` i `b` serà

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

on per a fer la simplificació hem suposat un valor de p gran.

El cost paral·lel de calcular concurrentment la suma de les porcions locals de les matrius `a` i `b` és

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

El cost de que P_0 reculla en la matriu `a` el resultat de la suma (`MPI_Gather`) es correspon a l'enviament d'un missatge de cada procés P_i $(i>0)$ al procés P_0 de grandària igual a $\frac{N}{p}N = \frac{N^2}{p}$

elements. Per tant, el cost serà:

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

Sumant els tres temps anteriors, tenim que el cost paral·lel és:

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

Speedup:

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

Eficiència:

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

Qüestió 2–11

La següent funció calcula el producte escalar de dos 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) Implementa una funció per a realitzar el producte escalar en paral·lel mitjançant MPI, utilitzant en la mesura del possible operacions col·lectives. Se suposa que les dades estan disponibles en el procés P_0 i que el resultat ha de quedar també en P_0 (el valor de tornada de la funció solament és necessari que siga correcte en P_0). Es pot assumir que la grandària del problema `n` és exactament divisible entre el nombre de processos.

Nota: a continuació es mostra la capçalera de la funció a implementar, incloent la declaració dels vectors locals (suposem que `MAX` és suficientment gran per a qualsevol valor de n i nombre de processos).

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

Solució:

```
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) Calcula l'speed-up. Si per a una grandària suficientment gran de missatge, el temps d'enviament per element fóra equivalent a 0.1 flops, quin speed-up màxim es podria aconseguir quan la grandària del problema tendeix a infinit i per a un valor suficientment gran de processos?

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

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

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

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

Si $t_w = 0.1$ Flops, aleshores el $S(n,p)$ estaria limitat per $\frac{2p}{0.2p} = 10$.

(c) Modifica el codi anterior per a que el valor de retorn siga el correcte en tots els processos.

Solució: La crida a `MPI_Reduce` es canvia per la següent línia:

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

Qüestió 2–12

Siga el codi seqüencial:

```
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) Implementa una versió paral·lela equivalent utilitzant MPI, tenint en compte els següents aspectes:

- El procés P_0 obté inicialment la matriu `A`, realitzant la crida `llegir(A)`, sent `llegir` una funció ja implementada.
- La matriu `A` s'ha de distribuir per blocs de files entre tots els processos.
- Finalment P_0 ha de contenir el resultat en la matriu `A`.
- Utilitza comunicacions col·lectives sempre que siga possible.

Se suposa que N és divisible entre el nombre de processos i que la declaració de les matrius usades és

```
double A[N][N], B[N][N];  /* B: matriu distribuida */
```

Solució:

```
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)
    llegir(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) Calcula l'speedup i l'eficiència.

Solució: El cost computacional seqüencial és $t(N) = N^2$ flops.
Com el repartiment (`MPI_Scatter`) o recollida (`MPI_Gather`) d'una matriu d'ordre N entre p processos comporta l'enviament/recepció de $p-1$ missatges de grandària $\frac{N^2}{p}$, el temps de comunicacions és $t_c = 2(p-1)\left(t_s + \frac{N^2}{p}t_w\right)$. El cost aritmètic paral·lel és $\frac{N^2}{p}$. Per tant, el temps total paral·lel resulta ser:

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

L'speedup és igual a

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

i l'eficiència

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

Qüestió 2–13

El següent programa llig una matriu quadrada A d'ordre N i construeix, a partir d'ella, un vector v de dimensió N de manera que la seu component i-ésima, $0 \leq i < N$, és igual a la suma dels elements de la fila i-ésima de la matriu A. Finalment, el programa imprimeix el 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) Utilitza comunicacions col·lectives per a implementar un programa MPI que realitze el mateix càlcul, d'acord amb els següents passos:

- El procés P_0 llig la matriu A.
- P_0 reparteix la matriu A entre tots els processos.
- Cada procés calcula la part local de v.
- P_0 arreplega el vector v a partir de les parts locals de tots els processos.
- P_0 escriu el vector v.

Nota: Per a simplificar, es pot assumir que N és divisible entre el nombre de processos.

Solució:

```
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) Calcula els temps seqüencial i paral·lel, sense tenir en compte les funcions de lectura i escriptura. Indica el cost que has considerat per a cadascuna de les operacions col·lectives realitzades.

Solució: Temps seqüencial:

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

Temps paral·lel aritmètic amb p processos:

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

Temps paral·lel de comunicacions amb p processos:

- Repartiment de la matriu A:
$$(p-1)\left(t_s + \frac{N^2}{p} t_w\right)$$
- Recollida del vector v:
$$(p-1)\left(t_s + \frac{N}{p} t_w\right)$$

Per tant, el temps paral·lel és

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

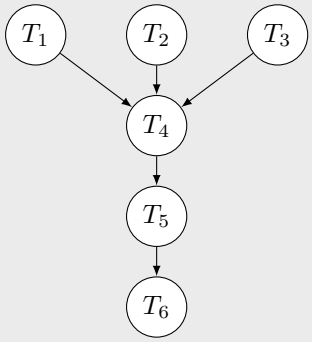
Qüestió 2–14

Donada la següent funció, on suposem que les funcions `T1`, `T2` i `T3` tenen un cost de $7n$ i les funcions `T5` i `T6` de n, sent n un valor constant.

```
double exemple(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) Dibuixa el graf de dependències i calcula el cost seqüencial.

Solució: El graf de dependències es mostra en la següent figura:



El cost seqüencial és: $t(n) = 7n + 7n + 7n + 2 + n + n \approx 23n$

(b) Paral·lelitza-la mitjançant MPI, suposant que hi ha tres processos. Tots els processos invoquen la funció amb el mateix valor de l'argument `val` (no és necessari comunicar-lo). El valor de retorn de la funció ha de ser correcte en el procés 0 (no és necessari que ho siga en els altres processos).

Nota: per a les comunicacions han d'utilitzar-se únicament operacions de comunicació col·lectiva.

Solució: Les tres primeres tasques s'han d'assignar a processos diferents, per a repartir la càrrega. Les altres tres tasques s'han d'executar en l'ordre seqüencial (a causa de les dependències), per la qual cosa les assignem a P_0, ja que aquest procés és el que ha d'emmagatzemar el valor correcte de retorn de la funció.

```
double exemple(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) Calcula el temps d'execució paral·lel (càlcul i comunicacions) i l'speedup amb tres processos. Calcula també l'speedup asimptòtic, és a dir, el límit quan n tendeix a infinit.

Solució: El temps paral·lel es calcula a partir del cost associat al camí crític del graf de dependències, corresponent a $T_1 - T_4 - T_5 - T_6$, per exemple. Per al cost de comunicació, considerem que l'operació de reducció internament enviarà únicament dos missatges, cadascun d'ells de longitud igual a 1 `double` (en la reducció també es realitzaran 2 flops, que no s'han inclòs en les expressions).

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

L'speedup serà:

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

Com en aquest cas el cost de comunicació és molt baix, asimptòticament (quan n creix) l'speedup tendeix al valor $S(n,3) \to \frac{23n}{9n} = 2,56$.

Qüestió 2–15

Donada la següent funció seqüencial:

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

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

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

  return cont;
}
```

(a) Fes una versió paral·lela utilitzant MPI, suposant que el vector `v` es troba inicialment només en el procés 0, i el resultat retornat per la funció només fa falta que siga correcte en el procés 0. Hauran de distribuir-se les dades necessàries perquè tots els càlculs es repartisquen de forma equitativa. Nota: Es pot assumir que `n` és divisible entre el nombre de processos.

Solució:

```
int comptar(double v[], int n)
{
  int i, cont, cont_loc=0, p;
  double mitja, suma_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++)
    suma_loc += vloc[i];

  MPI_Allreduce(&suma_loc, &mitja, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
  mitja = mitja/n;

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

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

(b) Calcula el temps d'execució de la versió paral·lela de l'apartat anterior, així com el límit del speedup quan `n` tendeix a infinit. Si has fet servir operacions col·lectives, indica quin és el cost que has considerat per a cadascuna d'elles.

Solució:

- Scatter: el procés 0 envia un missatge de n/p elements a cadascun dels altres processos. Per tant:
$$(p-1)(t_s + \frac{n}{p}t_w)$$
- Reduce: el procés 0 rep un missatge d'un element de cadascun dels altres processos i suma els elements, és a dir:
$$(p-1)(t_s + t_w) + (p-1)$$
- Allreduce: es pot realitzar mitjançant una operació *reduce* sobre el procés 0, seguida d'un *broadcast* del resultat. Per al *broadcast*, suposem que el procés 0 envia un missatge d'un element a cadascum dels altres processos.
$$2(p-1)(t_s + t_w) + (p-1)$$

- Bucles de càlcul:

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

El temps d'execució paral·lel és la suma de tot, és a dir:

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

D'altra banda, el temps seqüencial és:

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

D'on tinguem que:

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

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

Qüestió 2–16

El següent programa seqüencial realitza certs càlculs sobre una matriu quadrada `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) Paral·lelitza el codi mitjançant MPI suposant que cada procés té ja emmagatzemades `k=N/p` files consecutives de la matriu, sent `p` el nombre de processos (es pot asumir que `N` és divisible entre `p`). Aquestes files ocupen les primeres posicions de la matriu local, és a dir, entre les files `0` i `k-1` de la variable `A`. Nota: Han d'utilitzar-se primitives de comunicació col·lectiva sempre que siga possible.

Solució: Cada procés calcula una part del vector de sumes `sum`, emmagatzemant els valors en les primeres `k` posiciones. El valor de `fact` es calcula en P_0 (que és qui té la fila 0 de la matriu) i es difon a la resta de processos. En l'última part de l'algorisme, cada procés calcula el màxim local i es realitza una reducció el resultat de la qual (`max`) és necessari tenir en tots els processos.

```
#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) Escriu el codi per a realitzar la comunicació necessària després del càlcul anterior perquè la matriu completa quede emmagatzemada en el procés 0 en la variable `Aglobal[N][N]`.

Solució:

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

Qüestió 2–17

El següent codi proporciona el resultat de l'operació $C = aA + bB$, on A, B i C són matrius de $M \times N$ components, i a i b són nombres reals:

```
int main(int argc, char *argv[]) {
  int i, j;
  double a, b, A[M][N], B[M][N], C[M][N];
  LligOperands(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];
    }
  }
  EscriuMatriu(C);
  return 0;
}
```

Desenvolupa una versió paral·lela mitjançant MPI utilitzant operacions col·lectives, tenint en compte que:

- P_0 obtindrà inicialment les matrius A i B, així com els nombres reals a i b, mitjançant la invocació de la funció `LligOperands`.
- Únicament P_0 haurà de disposar de la matriu C completa com a resultat, i serà l'encarregat de cridar la funció `EscriuMatriu`.

- M és un múltiple exacte del nombre de processos.
- Les matrius A i B s'hauran de distribuir cíclicament per files entre els processos, per tal de fer en paral·lel l'operació esmentada.

Solució:

```
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) LligOperands(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) EscriuMatriu(C);
  MPI_Finalize();
  return 0;
}
```

Qüestió 2–18

Donada una matriu `A`, de `M` files i `N` columnes, la següent funció torna en el vector `sup` el nombre d'elements de cada fila que són superiors a la mitja.

```
void func(double A[M][N], int sup[M]) {
   int i, j;
   double mitja = 0;
   /* Calcula la mitja dels elements de A */
   for (i=0; i<M; i++)
      for (j=0; j<N; j++)
         mitja += A[i][j];
   mitja = mitja/(M*N);
   /* Conta nombre d'elements > mitja en cada fila */
   for (i=0; i<M; i++) {
```

```
      sup[i] = 0;
      for (j=0; j<N; j++)
         if (A[i][j]>mitja) sup[i]++;
   }
}
```

Escriviu una versió paral·lela de la funció anterior utilitzant MPI amb operacions de comunicació col·lectives, tenint en compte que la matriu `A` es troba inicialment en el procés 0, i que en finalitzar la funció el vector `sup` ha d'estar també en el procés 0. Els càlculs de la funció han de repartir-se de forma equitativa entre tots els processos. Es pot suposar que el nombre de files de la matriu és divisible entre el nombre de processos.

Solució:

```
void funcpar(double A[M][N], int sup[M]) {
   double Aloc[M][N];
   int suploc[M];
   double sumaloc, mitja;
   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);
   sumaloc=0;
   for (i=0; i<M/p; i++)
      for (j=0; j<N; j++)
         sumaloc += Aloc[i][j];
   MPI_Allreduce(&sumaloc, &mitja, 1, MPI_DOUBLE, MPI_SUM, MPI_COMM_WORLD);
   mitja = mitja/(M*N);
   for (i=0; i<M/p; i++) {
      suploc[i] = 0;
      for (j=0; j<N; j++)
         if (Aloc[i][j]>mitja) suploc[i]++;
   }
   MPI_Gather(suploc, M/p, MPI_INT, sup, M/p, MPI_INT, 0, MPI_COMM_WORLD);
}
```

Qüestió 2–19

Observa el següent programa, que llig un vector d'un fitxer, el modifica i mostra un resum per pantalla a més d'escriure el vector resultant en fitxer:

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

(a) Paral·lelitza'l amb MPI utilitzant operacions de comunicació col·lectiva allà on siga possible. L'entrada/eixida per pantalla i fitxer ha de fer-la només el procés 0. Se suposa que la grandària del vector (`n`) és un múltiple exacte del nombre de processos. Observa que la grandària del vector no és coneguda a priori, sinó que la retorna la funció `llig_vector`.

Solució: La funció `facto` no requereix modificació. En el programa principal, fem una distribució per blocs del 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 = llig_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);
    escriu_vector(n,v);
  }
  MPI_Finalize();
  return 0;
}
```

(b) Calculeu el temps d'execució seqüencial.

Solució:

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

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

(c) Calcula el temps d'execució paral·lel, indicant clarament el temps de cada operació de comunicació. No simplifiques les expressions, deixa-ho indicat.

Solució:

$$\begin{aligned}
t_p(n) &= t_{\text{Bcast}} + t_{\text{Scatter}} + \frac{n + 2n^2}{p} \text{ flops} + t_{\text{Gather}} + t_{\text{Reduce}} \\
t_{\text{Bcast}} &= (p-1)(t_s + t_w) \\
t_{\text{Scatter}} &= t_{\text{Gather}} = (p-1)\left(t_s + \frac{n}{p} t_w\right) \\
t_{\text{Reduce}} &= (p-1)(t_s + t_w + 1 \text{ flops})
\end{aligned}$$

Qüestió 2–20

Donada la següent funció, que calcula la suma d'un vector de `N` elements:

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

(a) Paral·lelitza-la amb MPI utilitzant únicament comunicacions punt a punt. El vector `v` està inicialment només en el procés 0 i el resultat es vol que siga correcte en *tots* els processos. Se suposa que la grandària del vector (`N`) és un múltiple exacte del nombre de processos.

Solució:

```
double suma(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) Paral·lelitza-la amb MPI amb les mateixes premisses de l'apartat anterior, però ara utilitzant comunicacions col·lectives on siga convenient.

Solució:

```
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;
}
```

Qüestió 2–21

Observeu la següent funció, que compta el nombre d'aparicions d'un valor en una matriu i indica també la primera fila en la que apareix:

```
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 està %d vegades, la primera vegada en la fila %d.\n",x,count,first);
}
```

(a) Paral·lelitzeu-la mitjançant MPI repartint la matriu A entre tots els processos disponibles. Tant la matriu com el valor a buscar estan inicialment disponibles únicament en el procés `owner`. Se suposa que el nombre de files i columnes de la matriu és un múltiple exacte del nombre de processos. El `printf` que mostra el resultat per pantalla l'ha de fer només un procés.

Utilitzeu operacions de comunicació col·lectiva allà on siga possible.

Per a fer-ho, completeu esta funció:

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

Solució: Podem utilitzar una distribució per blocs de files de la matriu:

```
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;

  /* Repartiment de la matriu per blocs de files */
  MPI_Scatter(A,size,MPI_DOUBLE,Aloc,size,MPI_DOUBLE,owner,MPI_COMM_WORLD);
  /* Difusió del valor a buscar */
  MPI_Bcast(&x,1,MPI_DOUBLE,owner,MPI_COMM_WORLD);

  /* Càlculs locals */
  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;
      }

  /* Arreplega de resultats en un procés i impressió */
  fl = rows*id + fl;  /* Passar índex 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 està %d vegades, la primera vegada en la fila %d.\n",x,count,
        first);
}
```

(b) Indiqueu el cost de comunicacions de cada operació de comunicació que heu utilitzat en l'apartat anterior. Suposeu una implementació bàsica de les comunicacions.

Solució:

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

Qüestió 2–22

(a) El següent fragment de codi utilitza primitives de comunicació punt a punt per a un patró de comunicació que pot efectuar-se mitjançant una única operació col·lectiva.

```
#define TAG 999
int sz, rank;
double val,res,aux;
MPI_Comm comm=MPI_COMM_WORLD;
MPI_Status stat;
val = ...
MPI_Comm_size(comm, &sz);
if (sz==1) res = val;
else {
```

```
  MPI_Comm_rank(comm, &rank);
  if (rank==0) {
    MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, TAG, comm, &stat);
    res = aux + val;
  } else if (rank==sz-1) {
    MPI_Send(&val, 1, MPI_DOUBLE, rank-1, TAG, comm);
  } else {
    MPI_Recv(&aux, 1, MPI_DOUBLE, rank+1, TAG, comm, &stat);
    aux = aux + val;
    MPI_Send(&aux, 1, MPI_DOUBLE, rank-1, TAG, comm);
  }
}
```

Escriviu la crida a la primitiva MPI de comunicació col·lectiva equivalent, amb els arguments corresponents.

Solució:

```
MPI_Reduce(&val, &res, 1, MPI_DOUBLE, MPI_SUM, 0, comm);
```

(b) Donada la següent crida a una primitiva de comunicació col·lectiva:

```
double val=...;
MPI_Bcast(&val, 1, MPI_DOUBLE, 0, comm);
```

Escriviu un fragment de codi equivalent (ha de realitzar la mateixa comunicació) però utilitzant únicament primitives de comunicació punt a punt.

Solució: Una possible implementació seria aquella en què el procés 0 realitza els enviaments mitjançant un bucle senzill.

```
#define TAG 999
double val=...;
int i, sz, rank;
MPI_Status stat;
...
MPI_Comm_size(comm, &sz);
MPI_Comm_rank(comm, &rank);
if (rank==0) {
  for (i=1; i<sz; i++) {
    MPI_Send(&val, 1, MPI_DOUBLE, i, TAG, comm);
  }
} else {
  MPI_Recv(&val, 1, MPI_DOUBLE, 0, TAG, comm, &stat);
}
```

3 Tipus de dades

Qüestió 3–1

Suposem definida una matriu d'enters `A[M][N]`. Escriu el fragment de codi necessari per a l'enviament des de P_0 i la recepció en P_1 de les dades que s'especifiquen en cada cas, usant per a açò un sol missatge. En cas necessari, defineix un tipus de dades derivat MPI.

(a) Enviament de la tercera fila de la matriu `A`.

Solució: En el llenguatge C, els arrays bidimensionals s'emmagatzemen per files, i per tant la separació entre elements de la mateixa fila és 1. Per tant, en aquest cas no és realment necessari crear un tipus MPI, per estar els elements contigus en memòria.

```
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) Enviament de la tercera columna de la matriu `A`.

Solució: La separació entre elements de la mateixa columna és `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);
```

Qüestió 3–2

Donat el següent fragment d'un programa MPI:

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

void distrib_dades(struct Tdades *dades, 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(&(dades->x), 1, MPI_INT, pr2, 0, comm);
      MPI_Send(&(dades->y[0]), n, MPI_INT, pr2, 0, comm);
      MPI_Send(&(dades->a[0]), n, MPI_DOUBLE, pr2, 0, comm);
    }
```

```
  } else {
    MPI_Recv(&(dades->x), 1, MPI_INT, 0, 0, comm, &status);
    MPI_Recv(&(dades->y[0]), n, MPI_INT, 0, 0, comm, &status);
    MPI_Recv(&(dades->a[0]), n, MPI_DOUBLE, 0, 0, comm, &status);
  }
}
```

Modificar la funció `distrib_dades` per a optimitzar les comunicacions.

(a) Realitza una versió que utilitze tipus de dades derivades de MPI, de manera que es realitze un enviament (a cada procés) en lloc de tres.

Solució: Atès que les dades a enviar/rebre són de diferents tipus, per a poder enviar-los en un mateix missatge s'ha de definir un tipus de dades mitjançant `MPI_Type_create_struct`.

```
void distrib_dades(struct Tdades *dades, int n, MPI_Comm comm) {
  int p, pr, pr2;
  MPI_Status status;
  MPI_Datatype Tnou;
  int longitut[]={1,n,n};
  MPI_Datatype tipus[]={MPI_INT, MPI_INT, MPI_DOUBLE};
  MPI_Aint despls[3];
  MPI_Aint dir1, dirx, diry, dira;

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

  /* Calcul dels desplaçaments de cada component */
  MPI_Get_address(dades, &dir1);
  MPI_Get_address(&(dades->x), &dirx);
  MPI_Get_address(&(dades->y[0]), &diry);
  MPI_Get_address(&(dades->a[0]), &dira);
  despls[0]=dirx-dir1;
  despls[1]=diry-dir1;
  despls[2]=dira-dir1;

  MPI_Type_create_struct(3, longitut, despls, tipus, &Tnou);
  MPI_Type_commit(&Tnou);
  if (pr==0) {
    for (pr2=1; pr2<p; pr2++) {
      MPI_Send(dades, 1, Tnou, pr2, 0, comm);
    }
  }
  else {
    MPI_Recv(dades, 1, Tnou, 0, 0, comm, &status);
  }
  MPI_Type_free(&Tnou);
}
```

(b) Realitza una modificació de l'anterior perquè utilitze primitives de comunicació col·lectiva.

Solució: Seria idèntica a l'anterior, excepte l'últim `if`, que es canviaria per la següent instrucció:

```
MPI_Bcast(dades, 1, Tnou, 0, comm);
```

Qüestió 3–3

Es vol implementar un programa paral·lel per a resoldre el problema del Sudoku. Cada possible configuració del Sudoku o "tauler" es representa per un array de 81 enters, contenint xifres entre 0 i 9 (0 representa una casella buida). El procés 0 genera n solucions, la validesa de les quals haurà de ser comprovada pels altres processos. Aquestes solucions s'emmagatzemen en una matriu A de grandària $n \times 81$.

(a) Escriure el codi que distribueix tota la matriu des del procés 0 fins a la resta de processos de manera que cada procés reba un tauler (suposant $n = p$, on p és el nombre de processos).

Solució:

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

(b) Suposem que per a implementar l'algorisme en MPI vam crear la següent estructura en C:

```
struct tasca {
  int tauler[81];
  int inicial[81];
  int es_solucio;
};
typedef struct tasca Tasca;
```

Crear un tipus de dades MPI `ttasca` que represente l'estructura anterior.

Solució:

```
Tasca t;
MPI_Datatype ttasca;
int blocklen[3] = { 81, 81, 1 };
MPI_Aint ad1, ad2, ad3, ad4, disp[3];
MPI_Get_address(&t, &ad1);
MPI_Get_address(&t.tauler[0], &ad2);
MPI_Get_address(&t.inicial[0], &ad3);
MPI_Get_address(&t.es_solucio, &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, &ttasca);
MPI_Type_commit(&ttasca);
```

Qüestió 3–4

Siga `A` un array bidimensional de números reals de doble precisió, de dimensió `N`×`N`. Defineix un tipus de dades derivades MPI que permeta enviar una submatriu de dimensió 3×3. Per exemple, la submatriu que comença en `A[0][0]` serien els elements marcats amb $\star$:

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

(a) Realitza les corresponents crides per a l'enviament des de P_0 i la recepció en P_1 del bloc de la figura.

Solució: Pot veure's com un vector de 3 blocs d'elements, cadascun d'ells amb longitud 3 i amb separació `N`.

```
double A[N][N];
int rank;
MPI_Datatype newtype;
... /* omplir matriu */
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) Indica què caldria modificar en el codi anterior perquè el bloc enviat per P_0 siga el que comença en la posició (0,3), i que es reba en P_1 sobre el bloc que comença en la posició (3,0).

Solució: Bastaria amb canviar l'adreça del buffer en `MPI_Send` i `MPI_Recv`. Concretament, hauria que posar `&A[0][3]` en lloc de `&A[0][0]` en la crida a `MPI_Send`, i `&A[3][0]` en lloc de `&A[0][0]` en la crida a `MPI_Recv`.

Qüestió 3–5

El següent programa paral·lel MPI deu calcular la suma de dues matrius A i B de dimensions $M \times N$ utilitzant una distribució cíclica de files, suposant que el nombre de processos `p` és divisor de M i tenint en compte que P_0 té emmagatzemades inicialment les matrius A i 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) llegir(A,B);

/* (a) Repartiment cíclic de files d'A i B */
/* (b) Càlcul local de Al+Bl */
/* (c) Recollida de resultats en el procés 0 */

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

(a) Implementa el repartiment cíclic de files de les matrius A i B, sent `Al` i `Bl` les matrius locals. Per a realitzar aquesta distribució deus o bé definir un nou tipus de dades de MPI o bé usar comunicacions col·lectives.

Solució:

Solució definint un nou tipus de dades:

```
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);
}
```

L'operació `MPI_Sendrecv` s'ha usat per a copiar la part localment emmagatzemada en el procés 0; també es podria fer amb un bucle o amb `memcpy`.

Solució usant comunicacions col·lectives:

```
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) Implementa el càlcul local de la suma `Al+Bl`, emmagatzemant el resultat en `Al`.

Solució:

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

(c) Escriu el codi necessari perquè P_0 emmagatzeme en `A` la matriu $A + B$. Per a açò, P_0 ha de rebre de la resta de processos les matrius locals `Al` obtingudes en l'apartat anterior.

Solució: Solució usant el tipus de dades definit en l'apartat (a):

```
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);
```

Solució usant comunicacions col·lectives:

```
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);
```

Qüestió 3–6

Implementa una funció on, donada una matriu A de $N \times N$ nombres reals i un índex k (entre 0 i $N-1$), la fila k i la columna k de la matriu es comuniquen des del procés 0 a la resta de processos (sense comunicar cap altre element de la matriu). La capçalera de la funció seria així:

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

Hauràs de crear i usar un tipus de dades que represente una columna de la matriu. No és necessari que s'envien juntes la fila i la columna. Es poden enviar per separat.

Solució:

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

  /* Envío de la fila */
  MPI_Bcast(&A[k][0], N, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  /* Envío de la columna */
  MPI_Bcast(&A[0][k], 1, colu, 0, MPI_COMM_WORLD);

  MPI_Type_free(&colu);
}
```

Qüestió 3–7

Es vol distribuir entre 4 processos una matriu quadrada d'ordre $2N$ ($2N$ files per $2N$ columnes) definida a blocs com

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

on cada bloc A_{ij} correspon a una matriu quadrada d'ordre N, de manera que es vol que el procés P_0 emmagatzeme localment la matriu A_{00}, P_1 la matriu A_{01}, P_2 la matriu A_{10} i P_3 la matriu A_{11}.

Per exemple, la següent matriu amb $N = 2$ quedaria distribuïda com es mostra:

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

(a) Implementa una funció que realitze la distribució esmentada, definint per a açò el tipus de dades MPI necessari. La capçalera de la funció seria:

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

on `A` és la matriu inicial, emmagatzemada en el procés 0, i `B` és la matriu local on cada procés ha de emmagatzemar el bloc que li corresponga de A.

Nota: es pot assumir que el nombre de processos del comunicador és 4.

Solució:

```
void comunica(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) Calcula el temps de comunicacions.

Solució: Com P_0 envia un total de tres missatges de N^2 dades a la resta de processos, el temps de comunicacions és $t_c = 3\left(t_s + N^2 t_w\right)$, sent t_s el temps d'establiment de la comunicació i t_w el temps necessari per a enviar una dada.

Qüestió 3–8

Desenvolupa una funció que servisca per a enviar una submatriu des del procés 0 al procés 1, on quedarà emmagatzemada en forma de vector. S'ha d'utilitzar un nou tipus de dades, de manera que s'utilitze un únic missatge. Recordeu que les matrius en C estan emmagatzemades en memòria per files.

La capçalera de la funció serà així:

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

Nota: s'assumeix que `m*n<==MAX` i que la submatriu a enviar comença en l'element `A[0][0]`.

Exemple amb $M = 4$, $N = 5$, $m = 3$, $n = 2$:

A (en P_0)						v (en 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		

Solució:

```
void envia(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);
}
```

Qüestió 3–9

Es pretén distribuir amb MPI els blocs quadrats de la diagonal d'una matriu quadrada de dimensió 3·DIM entre 3 processos. Si la matriu fora de dimensió 6 (DIM=2), la distribució seria com s'exemplifica:

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

Realitza una funció que permeta enviar els blocs amb el mínim nombre de missatges. Es proporciona la definició de la capçalera de la funció per a facilitar la implementació. El procés 0 té en `A` la matriu completa i després de la crida a la funció s'ha de tenir en `Alcl` de cada procés el bloc que li correspon. Utilitza primitives de comunicació punt a punt.

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

Solució:

```
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);
  }
}
```

Qüestió 3–10

Donada una matriu de `NF` files i `NC` columnes, inicialment emmagatzemada en el procés 0, es vol fer un

repartiment de la mateixa per blocs de columnes entre els processos 0 i 1, de manera que les primeres NC/2 columnes es queden en el procés 0 i la resta vagen al procés 1 (suposarem que `NC` és parell).

Implementa una funció amb la següent capçalera, que faça el repartiment indicat mitjançant MPI, definint el tipus de dades necessari perquè els elements que li toquen al procés 1 es transmeten mitjançant un sol missatge. Quan la funció acabe, tant el procés 0 com l'1 hauran de tindre en `Aloc` el bloc de columnes que els toca. És possible que el nombre de processos siga superior a 2. En eixe cas sols el 0 i l'1 hauran de tindre el seu bloc de matriu en `Aloc`.

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

Solució:

```
void distribueix(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);
}
```

Qüestió 3–11

Es vol implementar una operació de comunicació entre 3 processos MPI en la qual el procés P_0 té emmagatzemada una matriu `A` de dimensió $N \times N$, i ha d'enviar al procés P_1 la submatriu formada per les files d'índex parell, i al procés P_2 la submatriu formada per les files d'índex imparell. S'hauran d'utilitzar tipus de dades derivats de MPI per tal de realitzar el menor nombre possible d'enviaments. Cada matriu recollida en P_1 i P_2 haurà de quedar emmagatzemada en la matriu local `B` de dimensió $N/2 \times N$. Nota: Es pot assumir que N és un nombre parell.

Exemple: Si la matriu emmagatzemada en P_0 és

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

la matriu recollida per P_1 haurà de ser

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

i la matriu recollida per P_2 haurà de ser

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

(a) Implementa una funció amb la següent capçalera per a fer l'operació descrita:

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

Solució:

```
void comunica(double A[N][N], double B[N/2][N])
{
  int id;
  MPI_Datatype mitat;
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  MPI_Type_vector(N/2, N, 2*N, MPI_DOUBLE, &mitat);
  MPI_Type_commit(&mitat);
  if (id==0) {
    MPI_Send(&A[0][0], 1, mitat, 1, 100, MPI_COMM_WORLD);
    MPI_Send(&A[1][0], 1, mitat, 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(&mitat);
}
```

(b) Calcula el temps de comunicacions.

Solució: Es tracta de l'enviament/recepció de dos missatges de $N^2/2$ dades; per tant, el temps de comunicacions és

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

Qüestió 3–12

Implementa una funció en C per a enviar a tots els processos les tres diagonals principals d'una matriu, sense tindre en compte ni la primera ni l'última fila. Per exemple, per a a una matriu de grandària 6 s'haurien d'enviar els elements marcats amb `x`:

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

S'ha de fer definint un nou tipus de dades MPI que permeta enviar el bloc tridiagonal indicat utilitzant un sol missatge. Recorda que en C les matrius s'emmagatzemen en memòria per files. Utilitza esta capçalera per a la funció:

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

on

- `N` es el nombre de files i columnes de la matriu.
- `A` és la matriu amb les dades a enviar (en el procés que envia les dades) i la matriu on s'hauran de rebre (en la resta de procesos).
- El paràmetre `root` indica quin procés té inicialment en la seua matriu `A` les dades que s'han d'enviar a la resta de processos.

- `comm` és el comunicador que engloba tots els processos que hauran d'acabar tenint la part tridiagonal de `A`.

Per exemple, si es fera la crida a la funció:

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

el procés 5 seria el que té les dades vàlides en `A` a l'entrada a la funció, i a l'eixida tots els processos de `comm` haurien de tindre la part tridiagonal (menys la primera i última files).

Solució:

```
void envia_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);
}
```

Qüestió 3–13

El següent fragment de codi MPI implementa un algoritme en el que cada procés calcula una matriu de `M` files i `N` columnes, i totes eixes matrius s'arrepleguen en el procés P_0 formant una matriu global de `M` files i `N*p` columnes (on `p` és el nombre de processos), de forma que les columnes de P_1 apareixen a continuació de les de P_0, després les de P_2, i així successivament.

```
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) Modifiqueu el codi per tal que cada procés envie un únic missatge, en lloc d'un missatge per cada fila de la matriu. Per a fer-ho, s'haurà de definir un tipus de dades MPI per a la recepció.

Solució: Els processos envien un sol missatge de longitud `M*N` amb la submatriu completa. Però en el procés P_0 estes dades no s'emmagatzemen contiguament en memòria, per la qual cosa cal definir un tipus "vector" de MPI. El nou tipus ha de tindre `M` blocs (un per cada fila) de `N` elements (tants

com columnes en el procés que envia), amb una separació entre els blocs (*stride*) de `N*p` donat que la matriu completa (en P_0) té `N*p` columnes. En l'operació `MPI_Recv` el buffer ha d'apuntar a la primera posició de la submatriu corresponent al procés que envia (`k`), és a dir, `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) Calculeu el temps de comunicació tant de la versió original com de la modificada.

Solució: Versió original: s'envien un total de $(p-1)M$ missatges, cada un d'ells de longitud N, per tant

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

Versió modificada: s'envien un total de $p-1$ missatges, cada un d'ells de longitud MN, per tant

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

Qüestió 3–14

Es vol distribuir una matriu `A` de `F` files i `C` columnes entre els processos d'un comunicador MPI, utilitzant una distribució per blocs de columnes. El nombre de processos és `C`/2 on `C` és parell, de manera que la matriu local `Aloc` de cada procés tindrà 2 columnes.

Implementa una funció amb la següent capçalera que realitze la distribució esmentada, utilitzant comunicació punt a punt. La matriu `A` es troba inicialment en el procés 0, i en acabar la funció cada procés haurà de tindre en `Aloc` la part local que li corresponga de la matriu.

S'ha d'utilitzar el tipus de dades MPI adequat per a que només s'envie un missatge per procés.

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

Solució:

```
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);
}
```

Qüestió 3–15

Es vol implementar en MPI l'enviament pel procés 0 (i recepció en la resta de processos) de la diagonal principal i l'antidiagonal d'una matriu A, emprant per a tal fi tipus de dades derivades (un per a cada tipus de diagonal) i la menor quantitat possible de missatges. Suposarem que:

- N és una constant coneguda.
- Els elements de la diagonal principal són: $A_{0,0}$, $A_{1,1}$, $A_{2,2}$, …, $A_{N-1,N-1}$.
- Els elements de l'antidiagonal són: $A_{0,N-1}$, $A_{1,N-2}$, $A_{2,N-3}$, …, $A_{N-1,0}$.
- Només el procés 0 té la matriu A i enviarà les diagonals esmentades completes a la resta de processos.

Un exemple per a una matriu de grandària $N = 5$ seria: $A = \begin{pmatrix} * & & & & * \\ & * & & * & \\ & & * & & \\ & * & & * & \\ * & & & & * \end{pmatrix}$

(a) Completeu la següent funció, on els processos de l'1 en avant emmagatzemen sobre la matriu `A` les diagonals rebudes:

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

Solució:

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

```
  MPI_Bcast(&A[0][N-1], 1, antidiag, 0, MPI_COMM_WORLD);
  MPI_Type_free(&principal);
  MPI_Type_free(&antidiag);
}
```

(b) Completeu esta altra funció, variant de l'anterior, on tots els processos (incloent el procés 0) emmagatzemaran sobre els vectors `prin` i `anti` les corresponents diagonals:

```
void sendrecv_diagonals(double A[N][N], double prin[N], double anti[N]) {
```

Solució:

```
void sendrecv_diagonals(double A[N][N], double prin[N], double anti[N]) {
  int nprocs, id, p;
  MPI_Datatype principal,antidiag;
  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, &principal);
    MPI_Type_commit(&diagonal);
    MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);
    MPI_Type_commit(&antidiag);
    MPI_Sendrecv(A, 1, principal, 0, 0, prin, N ,MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&A[0][N-1], 1, antidiag, 0, 1, anti, N,
                 MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
    for (p=1; p<nprocs; p++) {
      MPI_Send(A, 1, principal, p, 0, MPI_COMM_WORLD);
      MPI_Send(&A[0][N-1], 1, antidiag, p, 1, MPI_COMM_WORLD);
    }
    MPI_Type_free(&principal);
    MPI_Type_free(&antidiag);
  }
  else {
    MPI_Recv(prin, N, MPI_DOUBLE, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Recv(anti, N, MPI_DOUBLE, 0, 1, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
  }
}
```

La solució anterior es pot simplificar mitjançant l'ús de la primitiva `MPI_Bcast`:

```
void sendrecv_diagonals(double A[N][N], double prin[N], double anti[N]) {
  int id;
  MPI_Datatype principal,antidiag;
  MPI_Comm_rank(MPI_COMM_WORLD, &id);
  if (id==0) {
    MPI_Type_vector(N, 1, N+1, MPI_DOUBLE, &principal);
    MPI_Type_commit(&diagonal);
    MPI_Type_vector(N, 1, N-1, MPI_DOUBLE, &antidiag);
    MPI_Type_commit(&antidiag);
    MPI_Sendrecv(A, 1, principal, 0, 0, prin, N ,MPI_DOUBLE, 0, 0,
                 MPI_COMM_WORLD, MPI_STATUS_IGNORE);
    MPI_Sendrecv(&A[0][N-1], 1, antidiag, 0, 1, anti, N,
                 MPI_DOUBLE, 0, 1, MPI_COMM_WORLD,MPI_STATUS_IGNORE);
```

```
        MPI_Type_free(&principal);
        MPI_Type_free(&antidiag);
      }
      MPI_Bcast(prin, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
      MPI_Bcast(anti, N, MPI_DOUBLE, 0, MPI_COMM_WORLD);
    }
```

Qüestió 3–16

Es vol repartir una matriu de M files i N columnes, que es troba en el procés 0, entre 4 processos mitjançant un repartiment per columnes cíclic. Com exemple es mostra el cas d'una matriu de 6 files i 8 columnes.

$$\begin{bmatrix} 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 \\ 11 & 12 & 13 & 14 & 15 & 16 & 17 & 18 \\ 21 & 22 & 23 & 24 & 25 & 26 & 27 & 28 \\ 31 & 32 & 33 & 34 & 35 & 36 & 37 & 38 \\ 41 & 42 & 43 & 44 & 45 & 46 & 47 & 48 \\ 51 & 52 & 53 & 54 & 55 & 56 & 57 & 58 \end{bmatrix}$$

Quedaria repartida de la següent forma:

$$P_0 = \begin{bmatrix} 1 & 5 \\ 11 & 15 \\ 21 & 25 \\ 31 & 35 \\ 41 & 45 \\ 51 & 55 \end{bmatrix}, \quad P_1 = \begin{bmatrix} 2 & 6 \\ 12 & 16 \\ 22 & 26 \\ 32 & 36 \\ 42 & 46 \\ 52 & 56 \end{bmatrix}, \quad P_2 = \begin{bmatrix} 3 & 7 \\ 13 & 17 \\ 23 & 27 \\ 33 & 37 \\ 43 & 47 \\ 53 & 57 \end{bmatrix}, \quad P_3 = \begin{bmatrix} 4 & 8 \\ 14 & 18 \\ 24 & 28 \\ 34 & 38 \\ 44 & 48 \\ 54 & 58 \end{bmatrix}$$

Implementeu una funció en MPI que realitze, mitjançant primitives punt a punt i de la forma més eficient possible, l'enviament i recepció de l'esmentada matriu. Nota: La recepció de la matriu haurà de fer-se en una matriu compacta (en `lmat`), com mostra l'exemple anterior. Nota: El nombre de columnes se suposa que es un múltiple exacte de 4 i es reparteix sempre entre 4 processos.

Per a la implementació es recomana utilitzar la següent capçalera:

```
int MPI_Reparte_col_cic(float mat[M][N], float lmat[M][N/4])
```

Solució:

```
int MPI_Reparte_col_cic(float mat[M][N], float lmat[M][N/4])
{
   int me, size, i;
   MPI_Datatype col;

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

   if (size!=4) {
      printf("Error, el nombre de processos ha de ser 4\n");
      return 1;
   }

   MPI_Type_vector(M*N/4,1,4,MPI_FLOAT,&col);
   MPI_Type_commit(&col);
```

```
        if (me==0) {
           for (i=1;i<size;i++)
              MPI_Send(&mat[0][i],1,col,i,0,MPI_COMM_WORLD);
           MPI_Sendrecv(mat,1,col,0,0,lmat,M*N/4,MPI_FLOAT,0,0,MPI_COMM_WORLD,
                        MPI_STATUS_IGNORE);
        } else
           MPI_Recv(lmat,M*N/4,MPI_FLOAT,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);

        MPI_Type_free(&col);
        return 0;
     }
```