Cuestión 1 (1.3 puntos)

Dada la siguiente función:

```
void computev(double v[N]) {
   double A[N][N], B[N][N], C[N][N];
   readm1(A);      // T1: lee una matriz A (coste N^2 flops)
   readm2(B);      // T2: lee una matriz B (coste N^2 flops)
   readv(v);       // T3: lee un vector v  (coste  N flops)
   p2Mat(A,C);     // T4
   pMatVec(B,v);   // T5
   pMatVec(C,v);   // T6
   pMatVec(B,v);   // T7
}
```

siendo

```
void p2Mat(double A[N][N],double B[N][N]){
   int i, j, k;
   for (i= 0; i< N; i++)
      for (j = 0; j < N; j++) {
          B[i][j]=0.0;
          for (k= 0; k< N; k++)
             B[i][j]+=A[i][k]*A[k][j];
      }
}

void pMatVec(double A[N][N],double v[N]){
   int i, j, k;
   double aux;
   for(k=0;k<N;k++)
      for (i= 0; i< N; i++){
         aux=v[i];
         for (j = 0; j < N; j++)
             aux+=A[i][j]*v[j]+2.0;
         v[i]=aux;
      }
}
```

0.1 p. (a) Calcula el coste computacional de las funciones `p2Mat` y `pMatVec`.

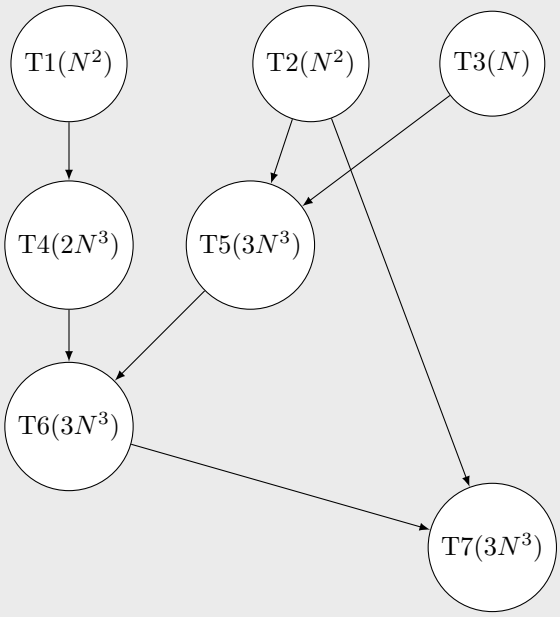
Solución:

$$\text{p2Mat}: \sum_{i=0}^{N-1}\sum_{j=0}^{N-1}\sum_{k=0}^{N-1} 2 = 2N^3 \text{ flops.}$$

$$\text{pMatVec}: \sum_{k=0}^{N-1}\sum_{i=0}^{N-1}\sum_{j=0}^{N-1} 3 = 3N^3 \text{ flops.}$$

0.3 p. (b) Obtén el grafo de dependencias.

Solución:



0.7 p. (c) Implementa una versión paralela con MPI utilizando solo dos procesos, teniendo en cuenta que el vector `v` obtenido debe quedar almacenado en la memoria local del procesador P0 y que la asignación de tareas a procesos maximice el paralelismo y minimice el coste de comunicaciones.

Solución: Una asignación que maximiza el paralelismo y minimiza el coste de comunicaciones consiste en que las tareas T2, T3, T5 y T7 sean asignadas al proceso P0 y las tareas T1, T4 y T6 sean asignadas al proceso P1.

```
void computevp(double v[N]) {
   double A[N][N], B[N][N], C[N][N];
   int rank, p;
   MPI_Comm_rank(MPI_COMM_WORLD, &rank);
   if(rank==0) {
      readm2(B);      // T2: lee una matriz B (coste N^2 flops)
      readv(v);       // T3: lee un vector v  (coste  N flops)
      pMatVec(B,v);   // T5 (coste 3N^3)
      MPI_Send(v, N, MPI_DOUBLE, 1, 100,MPI_COMM_WORLD);
      MPI_Recv(v, N, MPI_DOUBLE, 1, 200, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      pMatVec(B,v); // T7 (coste 3N^3)
   }
   else if(rank==1){
      readm1(A);      // T1: lee una matriz A (coste N^2 flops)
      p2Mat(A,C);     // T4 (coste 2N^3)
      MPI_Recv(v, N, MPI_DOUBLE, 0, 100, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
      pMatVec(C,v);   // T6  (coste 3N^3)
      MPI_Send(v, N, MPI_DOUBLE, 0, 200,MPI_COMM_WORLD);
   }
}
```

0.2 p. (d) Calcula el coste secuencial y el coste paralelo.

Solución:

Coste secuencial:

$$t_s(N) = N^2 + N^2 + N + 2N^3 + 3N^3 + 3N^3 + 3N^3 \approx 11N^3 \text{flops}.$$

Coste paralelo:

$$t_a(N,2) = N + N^2 + 3N^3 + 3N^3 + 3N^3 \approx 9N^3 \text{flops},$$
$$t_c(N,2) = 2(t_s + Nt_w),$$
$$t(N,2) = t_a(N,2) + t_c(N,2) \approx 9N^3 \text{flops} + 2(t_s + Nt_w).$$

Cuestión 2 (1.1 puntos)

Dada una matriz `A`, de `F` filas y `C` columnas, y un vector `v` de dos elementos, la siguiente función calcula el vector `x`, de `F` elementos.

```
void calcula(double A[F][C], double v[2], double x[F]) {
  int i, j;
  double sum, min;
  /* Calcula el vector x
   * x[i] = suma de elementos de la fila i de A que son >=v[0] y <=v[1] */
  for (i=0; i<F; i++) {
    sum=0;
    for (j=0; j<C; j++) {
      if (A[i][j]>=v[0] && A[i][j]<=v[1])
        sum += A[i][j];
    }
    x[i] = sum;
  }
  /* Calcular el mínimo elemento de x, y restarlo a todos los elementos */
  min=x[0];
  for (i=1; i<F; i++)
    if (x[i]<min) min = x[i];
  for (i=0; i<F; i++)
    x[i] = x[i]-min;
}
```

0.8 p. (a) Haz una versión paralela de la función anterior, donde los cálculos se repartan de forma equilibrada entre todos los procesos y las comunicaciones se hagan mediante operaciones colectivas. Inicialmente, tanto la matriz `A` como el vector `v` están disponibles únicamente en el proceso 0, y se requiere que, al finalizar la función, el vector `x` esté disponible en todos los procesos. Se puede suponer que `F` es divisible entre el número de procesos.

La cabecera de la función será la siguiente, donde `Aloc` y `xloc` pueden ser utilizadas para que cada proceso guarde su parte local de `A` y `x`, respectivamente.

```
void calculap(double A[F][C],double v[2],double x[F],
              double Aloc[][C],double xloc[])
```

Solución:

```
void calculap(double A[F][C],double v[2],double x[F],
              double Aloc[][C],double xloc[]) {
  int i, j, p;
  double sum, min, minloc;
  MPI_Comm_size(MPI_COMM_WORLD, &p);
```

```
      MPI_Scatter(A,F/p*C,MPI_DOUBLE,Aloc,F/p*C,MPI_DOUBLE,0,MPI_COMM_WORLD);
      MPI_Bcast(v,2,MPI_DOUBLE,0,MPI_COMM_WORLD);
      /* Calcula el vector x
       * x[i] = suma de elementos de la fila i de A que son >=v[0] y <=v[1] */
      for (i=0; i<F/p; i++) {
        sum=0;
        for (j=0; j<C; j++) {
          if (Aloc[i][j]>=v[0] && Aloc[i][j]<=v[1])
            sum += Aloc[i][j];
        }
        xloc[i] = sum;
      }
      /* Calcular el mínimo elemento de x, y restarlo a todos los elementos */
      minloc=x[0];
      for (i=1; i<F/p; i++)
        if (xloc[i]<minloc) minloc = xloc[i];
      MPI_Allreduce(&minloc,&min,1,MPI_DOUBLE,MPI_MIN,MPI_COMM_WORLD);
      for (i=0; i<F/p; i++)
        xloc[i] = xloc[i]-min;
      MPI_Allgather(xloc,F/p,MPI_DOUBLE,x,F/p,MPI_DOUBLE,MPI_COMM_WORLD);
    }
```

0.3 p. (b) Indica el coste de comunicaciones de dos operaciones de comunicación diferentes que hayas usado en el apartado anterior, suponiendo una implementación sencilla de las comunicaciones.

Solución: Aunque se piden solo dos, se indica el coste de todas:

- Scatter: $t_c = (p-1)\left(t_s + \frac{FC}{p}t_w\right) \approx pt_s + FCt_w$
- Bcast: $t_c = (p-1)(t_s + 2t_w) \approx pt_s + 2pt_w$
- Allreduce (equivale a Reduce+Bcast): $t_c = 2(p-1)(t_s + t_w) \approx 2pt_s + 2pt_w$
- Allgather (equivale a Gather+Bcast):
  $t_c = (p-1)\left(t_s + \frac{F}{p}t_w\right) + (p-1)\left(t_s + Ft_w\right) \approx 2pt_s + pFt_w$

Cuestión 3 (1.1 puntos)

La siguiente función calcula la raíz cuadrada de la suma de los elementos de una matriz triangular inferior elevados al cuadrado:

```
double raiz_sumacuad_triang_inf(double A[M][N]) {
  int i,j;
  double suma, raiz;
  suma=0;
  for (i=0;i<M;i++) {
     for (j=0;j<=i;j++) {
         suma+=A[i][j]*A[i][j];
     }
  }
  raiz=sqrt(suma);
  return raiz;
}
```

Se pide paralelizar dicho código de modo que se lleve a cabo un reparto cíclico de las filas de la matriz empleando tipos de datos derivados, a fin de equilibrar la carga y reducir el número de envíos. Se supone que:

- El proceso 0 dispondrá inicialmente de la matriz A completa y tendrá que repartirla entre el resto de procesos para llevar a cabo los cálculos oportunos en paralelo. Cada proceso almacenará las filas que le correspondan en una matriz A local, con espacio solo para el número de filas que le tocan a cada proceso, la cual se declarará como parámetro de entrada y salida, junto a la matriz A, en la función a implementar.
- Para que sea más sencillo, la matriz A se distribuirá entre los procesos sin tener en consideración el que sea triangular inferior (no hay que preocuparse por el hecho de enviar los elementos iguales a cero situados a la derecha de la diagonal, aunque ningún proceso deberá operar con dichos datos).
- El valor de retorno de la función deberá ser válido en todos los procesos.
- Supondremos que el número M de filas de la matriz siempre será múltiplo del número de procesos empleados y que M y N son constantes conocidas.

Solución:

```
double raiz_sumacuad_triang_inf(double A[M][N],double Alocal[][N]) {
  int i,j,id,np,k;
  double suma, suma_local,raiz;
  MPI_Datatype filas_ciclicas;

  MPI_Comm_size(MPI_COMM_WORLD,&np);
  MPI_Comm_rank(MPI_COMM_WORLD,&id);
  k=M/np;
  MPI_Type_vector(k,N,np*N,MPI_DOUBLE,&filas_ciclicas);
  MPI_Type_commit(&filas_ciclicas);
  if (id==0) {
      MPI_Sendrecv(A,1,filas_ciclicas,0,0,Alocal,k*N,MPI_DOUBLE,
                          0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
      for (i=1;i<np;i++)
             MPI_Send(&A[i][0],1,filas_ciclicas,i,0,MPI_COMM_WORLD);
  }
  else
     MPI_Recv(Alocal,k*N,MPI_DOUBLE,0,0,MPI_COMM_WORLD,MPI_STATUS_IGNORE);
  suma_local=0;
  for (i=0;i<k;i++) {
     for (j=0;j<=np*i+id;j++) {
        suma_local+=Alocal[i][j]*Alocal[i][j];
     }
  }
  MPI_Allreduce(&suma_local,&suma,1,MPI_DOUBLE,MPI_SUM,MPI_COMM_WORLD);
  raiz=sqrt(suma);
  MPI_Type_free(&filas_ciclicas);
  return raiz;
}
```