Práctica 3

Sesión 4

Objetivo

Resolver sistemas de ecuaciones lineales mediante la descomposición LU (ver memoria).

Estudio del problema y de una implementación paralela basada en el reparto por bloques de filas (sistbf.c).

Transformación a una implementación paralela basada en el reparto cíclico de filas (sistcf.c).

Descripción del problema

- Resolver sistemas de ecuaciones lineales *Ax=b,* donde $A \in \mathbb{R}^{nxn}, b \in \mathbb{R}^{nx1}$ mediante la descomposición *LU*:
  - Obtener la descomposición *A=LU (E. Gaussiana)*
  - Resolver el sistema triangular inferior unidad *Ly=b* (elementos diagonales de *L* iguales a *1*)
  - Resolver el sistema triangular superior *Ux=y*

Programa paralelo proporcionado

El programa (sistbf.c) genera un sistema lineal *Ax=b* y lo resuelve realizando una serie de pasos marcados con step (paso) *x*:

1. **Generar los datos**. El proceso *0* genera la matriz (*A*), y el vector (*b*) completos. Todos los procesos (incluido el *0*), reservan memoria para su matriz local (*Aloc*).
2. **Distribuir los datos.** La matriz se distribuye entre los procesos por bloques de *mb* filas consecutivas. El vector *b* se replica en todos los procesos.
3. **Descomposición LU.** En esta fase la matriz *A* se sobreescribe por *L* y por *U*. Los elementos de *L* quedan en la parte inferior de *A* y los de *U* en la superior.
4. **Resolver el sistema triangular inferior** *Ly=b*. El vector *y* se almacena sobre *b* sobreescribiéndolo.
5. **Resolver el sistema triangular superior** *Ux=*y. El vector *y* se encuentra almacenado en la variable *b*, y en esta fase se sobreescribe con el vector *x*.

Programa paralelo proporcionado (II)

Ejercicio 1:

Compila y ejecuta el programa. Veamos una prueba corta en el frontend. Para 5 ecuaciones usando tres procesos:

$$\begin{bmatrix} 25 & 4 & 3 & 2 & 1 \\ 4 & 25 & 4 & 3 & 2 \\ 3 & 4 & 25 & 5 & 3 \\ 2 & 3 & 4 & 25 & 4 \\ 1 & 2 & 3 & 4 & 25 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ x_4 \end{bmatrix} = \begin{bmatrix} 35 \\ 38 \\ 39 \\ 38 \\ 35 \end{bmatrix}$$

```
$ mpiexec –n 3 sistbf 5
```

Al ejecutar el programa, aparecerán el contenido de la matriz *A* y el vector *b*, en distintos puntos del programa:

Programa paralelo proporcionado (III)

Tras realizar el reparto inicial:

```
Matrix A:
---- proc. 0 ----
25.000 4.000 3.000 2.000 1.000
4.000 25.000 4.000 3.000 2.000
---- proc. 1 ----
3.000 4.000 25.000  4.000  3.000
2.000 3.000  4.000 25.000  4.000
---- proc. 2 ----
1.000 2.000  3.000 4.000  25.000

Vector b:
---- proc. 0 ----
35.000 38.000 39.000 38.000 35.000
---- proc. 1 ----
35.000 38.000 39.000 38.000 35.000
---- proc. 2 ----
35.000 38.000 39.000 38.000 35.000
```

Fase de distribucion de datos (I).

- Hay que cambiar el tipo de distribucion de datos Ejercicio 2:

1-**Bloques de filas consecutivas**  2-**Cíclica por filas**

Variables ***A*** **(Matriz global)**, y ***Aloc*** **(Matriz local)**

1-Se puede efectuar mediante un `MPI_Scatter`

```
/* STEP 2: Distribute data (A, b) */

  MPI_Bcast(b, n, MPI_DOUBLE, 0, MPI_COMM_WORLD);

  MPI_Scatter(A[0], mb * n, MPI_DOUBLE,
              Aloc[0], mb * n, MPI_DOUBLE, 0, MPI_COMM_WORLD);
```

Fase de distribucion de datos (I).

2- Se puede efectuar mediante varios `MPI_Scatter`:

Para la distribución cíclica de la matriz *A* en *p* procesos:

- Las primeras *p* filas de la matriz van cada una a un proceso. Esto sería una operación *scatter*.
- Lo mismo ocurre con las siguientes *p* filas. Y así sucesivamente. Se puede hacer mediante un bucle en el que en cada iteración corresponde a una operación scatter. Hay que prestar atención a:
  - La posición (sobre la matriz global *A*) donde empiezan los datos a enviarse en cada *scatter*.
  - La posición (sobre la matriz local *Aloc*) donde deben recibirse los datos en cada scatter

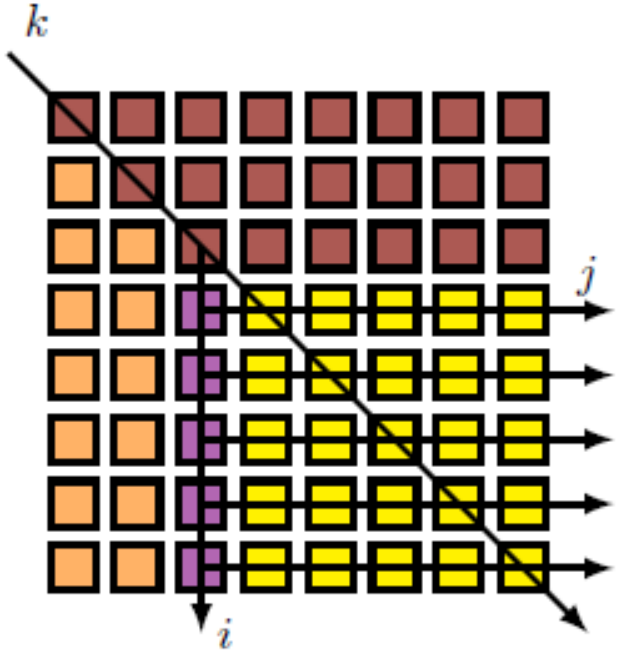
Fase de distribución de datos(II)

```
mpiexec –n 3 sistcf 5

Matriz A:
---- proc. 0 ----
 25.000   4.000   3.000   2.000   1.000
  2.000   3.000   4.000  25.000   4.000
---- proc. 1 ----
  4.000  25.000   4.000   3.000   2.000
  1.000   2.000   3.000   4.000  25.000
---- proc. 2 ----
  3.000   4.000  25.000   4.000   3.000

Vector b:
---- proc. 0 ----
 35.000  38.000  39.000  38.000  35.000
---- proc. 1 ----
 35.000  38.000  39.000  38.000  35.000
---- proc. 2 ----
 35.000  38.000  39.000  38.000  35.000
```

El resto de datos son incorrectos ya que aún no hemos modificado los algoritmos de la descomposición LU, y de resolución de sistemas triangulares.

Implementación paralela descomposición LU

```
Para k = 0, ..., n-2
  si A(k,k) = 0 entonces abandona
  Para i = k+1, ..., n-1
    /* Modificar fila i (elementos de la columna k a la n-1) */
    A(i,k) = A(i,k)/A(k,k)
    Para j = k+1, ..., n-1
      A(i,j) = A(i,j) - A(i,k)*A(k,j)
    Fin_para
  Fin_para
Fin_para
```



- Hay (*n-1*) etapas (bucle *k*)
- Cada etapa modifica el bloque *i=k+1…n-1* (filas) y *j=k+1…n-1* (columnas).
- Como las actualizaciones de filas (bucle *i*) son independientes, repartiremos *A* por filas.
- Antes de cada actualización, la fila pivote en cada iteración (fila k) deberá ser enviada a todos los procesos involucrados.

Implementación paralela (III)

- Paralelizar el bucle *i* del algoritmo secuencial.
  - Cada iteración actualiza una fila (la fila *i*).
  - La actualización de cada fila es independiente.
- Cada proceso actualiza las filas entre *k-1*, y n-1.
- Es necesario utilizar la fila pivote. Por tanto debe de ser enviada a todos los procesos (difusión), antes de empezar la actualización de las filas.

- Necesitamos usar dos funciones extra:
  - `propietario(i)`: proceso propietario de la fila *i*.
  - `iloc(i)`: indice local en *Aloc* de la fila *i* de *A*.

Implementación paralela descomposición LU (II)

```
Para k = 0, ..., n-2
  Si propietario(k) = yo
     si A(iloc(k),k) = 0 entonces abandona
  Fin_si
  difundir fila k
  Para i = k+1, ..., n-1
    /* Modificar fila i (elementos de la columna k a la n-1) */
    Si propietario(i) = yo
      A(iloc(i),k) = A(iloc(i),k)/A(k,k)
      Para j = k+1, ..., n-1
        A(iloc(i),j) = A(iloc(i),j) - A(iloc(i),k)*A(k,j)
      Fin_para
    Fin_si
  Fin_para
Fin_para
```

Implementación paralela de los sistemas triangulares(I)

Tras la factorización hay que resolver dos sistemas triangulares:

- Resolver el sistema triangular inferior unidad $Ly=b$ (elementos diagonales de L iguales a 1)
- Resolver el sistema triangular superior $Ux=y$.
- En ambos casos el vector b se sobreescribe con la solución del sistema.
- Ambos algoritmos son muy similares.

Implementación paralela de los sistemas triangulares(II)

Triangular Inferior

```
Para i = 0, 1, ..., n-1
  Para j = i+1, ..., n-1
    b(j) = b(j) - L(j,i)*b(i)
  Fin_para
Fin_para
```

Triangular Superior

```
Para i = n-1, ..., 0
  b(i) = b(i)/U(i,i)
  Para j = i-1, ..., 0
    b(j) = b(j) - U(j,i)*b(i)
  Fin_para
Fin_para
```

- En cada iteración del bucle *i* se actualizan mediante el bucle *j* los elementos del vector *b* que hay por debajo del elemento *i* (s. triangular inferior) o por encima (s. triangular superior).

- Esta actualización requiere usar el elemento de b(i).

Implementación paralela de los sistemas triangulares(III)

Triangular Inferior

```
Para i = 0, 1, ..., n-1
  difundir b(i)
  Para j = i+1, ..., n-1
    Si propietario(j) = yo
      b(j) = b(j) - L(iloc(j),i)*b(i)
    Fin_si
  Fin_para
Fin_para
```

Triangular Superior

```
Para i = n-1, ..., 0
  Si propietario(i) = yo
    b(i) = b(i)/U(iloc(i),i)
  Fin_si
  difundir b(i)
  Para j = i-1, ..., 0
    Si propietario(j) = yo
      b(j) = b(j) - U(iloc(j),i)*b(i)
    Fin_si
  Fin_para
Fin_para
```

La paralelización se basa en:

- Paralelizar el bucle *j*, de forma que cada proceso actualice los elementos de las filas que posee.
- Para ello el valor de *b(i)* deberá ser propagado previamente a todos los procesos.
- Al final, todos los procesos acaban con una copia del vector de incógnitas completo.

Modificaciones a realizar (I)

Los algoritmos paralelos descritos son válidos para cualquiera de las dos formas de distribución estudiadas (usando las funciones `propietario` e `iloc`):

Ejercicio 3: Modificar `propietario` e `iloc` (owner y `localindex` en el código) para trabajar con una distribución cíclica for filas. El comportamiento de estas funciones debe ser:

- Dado un índice de fila *i* de la matriz global *A*, la función `owner` debe devolver el índice del proceso que tiene esa fila en su matriz local *Aloc*.

- Dado un índice de fila *i* de la matriz global *A*, la función `localIndex` debe devolver el índice de dicha fila en la matriz local *Aloc* del proceso propietario de la fila.

- También es necesario modificar la función `numLocalRows`, que devuelve el número de filas locales de la matriz en un proceso (ese un número puede ser distinto de `mb`, puesto que el un número de filas puede no ser divisible entre el un número de procesos).

  En este caso se facilita el cambio a realizar, de manera que solo hay que descomentar la parte de código que corresponde a la distribución cíclica, y comentar o eliminar la otra parte.

Modificaciones a realizar (II)

Una vez hechos los cambios, hay que comprobar que todo funciona correctamente:

```
Matrix LU:
---- proc. 0 ----
25.000 4.000 3.000  2.000 1.000
 0.080 0.110 0.140 24.074 3.352
---- proc. 1 ----
0.160 24.360 3.520 2.680  1.840
0.040  0.076 0.108 0.139 24.071
---- proc. 2 ----
0.120 0.144 24.131 3.373 2.614

 Vector b after triInf:
---- proc. 0 ----
35.000 32.400 30.118 27.426 24.071
---- proc. 1 ----
35.000 32.400 30.118 27.426 24.071
---- proc. 2 ----
35.000 32.400 30.118 27.426 24.071
```

```
Vector b after triSup (system solution)
---- proc. 0 ----
1.000 1.000 1.000 1.000
---- proc. 1 ----
1.000 1.000 1.000 1.000
---- proc. 2 ----
1.000 1.000 1.000 1.000

Total accumulated error: 0.000000
```

Pruebas gran dimensión

- Efectuar pruebas con un tamaño suficiéntemente grande (entre *1000* y *2000*).

- Comentar la linea `: #define verbose`

  Evita la impresión de datos intermedios.

- Analizar cual de las dos versiones es mas eficiente y razonar a que puede deberse.