Introducción a Parallel toolbox de Matlab

Víctor M. García

Contenidos

- Descripción general de Parallel Toolbox: Preliminares
- Parfor
- SPMD
- pmode
- Computación en GPU

Parallel Computing en MATLAB:

-Orientado simultáneamente a modelo Multicore (1 sola máquina con varios cores) y a modelo Distribuido (varias máquinas);

-Modelo Distribuido: requiere de disponer de varias máquinas conectadas (el "cluster" de cálculo), y de la instalación en todas las máquinas de MATLAB y del Matlab Distributed Computing Server. Si se dispone de él, es posible ejecutar programas paralelos de Matlab en Cloud (Hadoop), entre otras posibilidades.

-Modelo Multicore: Solo se necesita Matlab con parallel toolbox

-Computación en GPU; funciona "aparte" de las extensiones paralelas, necesita una GPU de NVIDIA, con arquitectura FERMI (mínimo)

Parallel Computing en MATLAB: Configuración

Una máquina, modo “local”
-Matlab detecta automáticamente los “cores” disponibles en la máquina . Normalmente se descartan los cores virtuales que aparecen por hyperthreading, pero no siempre.

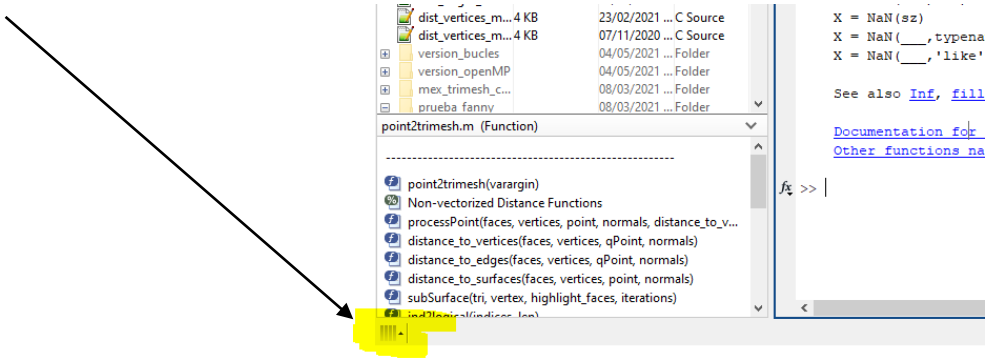
Versión “nueva” (Posterior R2013a):

El PCT arranca automáticamente al ejecutarse alguna instrucción paralela (parfor, spmd, etc).

-Es preferible arrancarlo de forma explícita con “parpool” o ”parpool 4” si queremos arrancar 4 “labs” o “workers”. En algún caso es útil arrancarlo como
>>p=parpool
En versiones viejas, el límite de workers era 12. En la versión actual (R2022) el límite son los cores físicos de la maquina

Parallel Computing en MATLAB: Configuración

-Es posible arrancar el parpool de forma explícita con el botón de abajo a la izquierda



Parallel Computing en MATLAB: Configuración

En versiones nuevas, hay un cambio bastante sustancial. Se puede arrancar el pool de forma que en vez de crear “procesos” se pueden crear “threads”

P=parpool % o
P=parpool(‘local’) % crea procesos, con espacios de memoria independiente

P=parpool(‘threads’) % crea threads, donde todos pueden acceder a la memoria de la máquina

Parallel Computing en MATLAB: Configuración

La instrucción gcp devuelve el pool de trabajadores actual.

Fuera de las instrucciones paralelas podemos averiguar el número de trabajadores activos así:

```
poolobj = gcp('nocreate'); % If no pool, do not create new one.
if isempty(poolobj) poolsize = 0;
   else poolsize = poolobj.NumWorkers
end
```

Parallel Computing en MATLAB: Administrador de tareas

Tras pulsar Ctrl+Alt+Supr, se abre el administrador de tareas de Windows.

Si ejecutamos “parpool”, podremos observar en la pestaña “procesos”del administrador de tareas que se arrancan varias copias de MATLAB, tantas como cores haya disponibles.

Muchas funciones de MATLAB utilizan todos los cores; se puede controlar que cores utiliza cada Matlab, seleccionando en el administrador de tareas cada copia de Matlab y, usando el botón derecho del ratón, escogemos la opción “Establecer afinidad”

Parallel Computing en MATLAB: Administrador de tareas

-Realiza, tomando tiempos, producto matriz por matriz, para matrices cuadradas de tamaño 5000. Procura visualizar al mismo tiempo el administrador de tareas, con la pestaña “rendimiento” seleccionada.

-Cambia la afinidad de Matlab usando el administrador de tareas, para que se pueda ejecutar en un solo core. Repite el producto de matrices.

Instrucción Parfor

Una vez abierto el "pool" de workers o labs, ya podemos usar instrucciones paralelas:

Parfor: pretende ser el equivalente paralelo del for. Distribuye las iteraciones del bucle parfor entre los "labs".

```
for i=1:1000
  a(i)=a(i)+sin(i)
end
```

```
parfor i=1:1000
  a(i)=a(i)+sin(i)
end
```

El cliente manda los datos (y el código) a los "workers" para que lo ejecuten. Las iteraciones se reparten entre los workers.

Matlab debe ser capaz de analizar el parfor y decidir que datos debe mandar a cada worker, antes de ejecutarlo: las iteraciones deben ser independientes y se deben poder ejecutar en cualquier orden.

Instrucción Parfor: Ejemplo (1)
Resolver 800 ecuaciones de segundo grado, en paralelo

```
N=800;
A=rand(N,1);
B=rand(N,1);
C=rand(N,1);
sol=zeros(N,2);
for i=1:N
   sol(i,:)=roots([A(i),B(i),C(i)]);
end
```

Experimenta con el tamaño del problema, comparando con la ejecución secuencial (con **for** en lugar de **parfor**); toma tiempos para ver cuando resulta ventajoso usar el **parfor**.
¿Y si en la toma de tiempos se incluye **parpool** ?
Probar con parpool('threads')

Instrucción Parfor: Restricciones

-NO debe haber dependencia de datos entre las distintas operaciones (salvo en el caso de reducciones)
Por ejemplo (código erróneo):

```
x(1)=1;
parfor i=2:100
    x(i)=x(i-1)+1;
end
```

-La variable de control de un parfor debe recorrer valores **enteros**, **crecientes** y **consecutivos.** Los siguientes tres ejemplos serían bucles normales cambiando el parfor por un for, pero tal como están no funcionarán correctamente:

-parfor i=0:0.1:1.0 (valores de la variable i no enteros)
-parfor i=5:-1:1 (valores de la variable i no crecientes)
-parfor i=2:2:10 (valores de la variable i no consecutivos)

Instrucción Parfor: Restricciones

En general no se puede anidar un parfor con otro parfor. Sí que es posible anidar un parfor dentro de un for, o un for dentro de un parfor.
Sin embargo, en estas últimas versiones del PCT ha surgido una restricción antes respecto a los límites de un for dentro de un parfor.

Supongamos que queremos procesar una matriz A, procesando las filas en paralelo, y, para cada fila, queremos procesar todas las columnas menos la última. Típicamente tendríamos algo así:

```
[m,n]=size(A)
parfor i=1:m          % procesado paralelo de las filas
    for j=1:n-1       % procesado de todas las columnas de la primera
                      %hasta la penúltima
    .....
    end
end
```

Instrucción Parfor: Restricciones

Este trozo de código funcionaba correctamente en versiones anteriores del PCT, pero ahora da un error porque en estas versiones los límites del bucle for no pueden contener expresiones, aunque sean tan simples como "n-1". Para que funcione, hay que crear una variable con valor n-1 y usarla como límite del bucle.

```
[m,n]=size(A);
nm1=n-1    %variable auxiliar
parfor i=1:m                    % procesado paralelo de las filas
    for j=1:nm1                  % procesado de todas las columnas de la
                                 % primera hasta la penúltima
    .....
    end
end
```

Instrucción Parfor

Matlab solo enviará a cada worker los datos necesarios; debe ser capaz automáticamente de “averiguar” cuales son esos datos.

Los vectores y matrices que se deben “partir” y enviar se dicen variables “Sliced” (En el ejemplo anterior, el vector a es “sliced”).

Otros tipos de variable involucrados en un parfor:
-índice del bucle
-Variables de tipo”Broadcast” : Se usan en el bucle pero no se escriben.
-variables temporales: definidas (o redefinidas en el bucle)
-Variables de “reducción”.

Instrucción Parfor: Reducción

Una reducción es una “acumulación” a lo largo de las iteraciones de un bucle: Reducciones correctas:

```
x=0;
parfor i=1:10
   x=x+i;
end
x
```

```
x2=0;
parfor i=1:10
   x2=[x2,i];
end
x2
```

Reducciones incorrectas (cálculo de sucesión de Fibonacci):

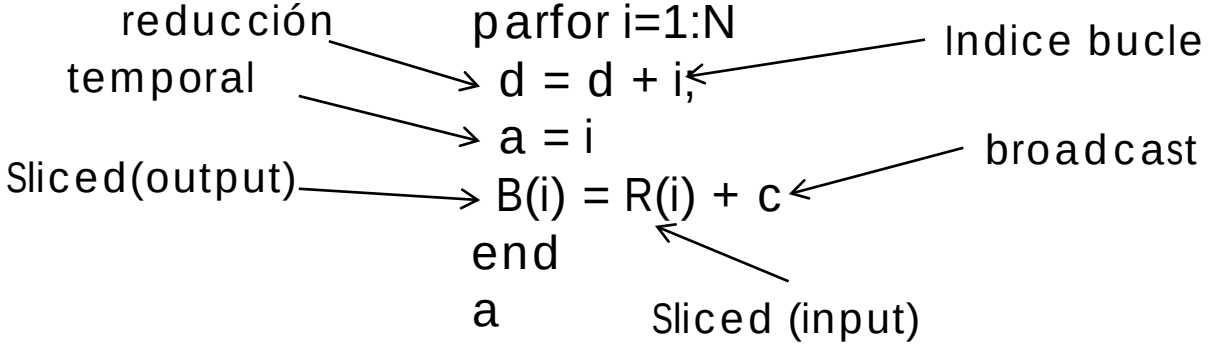
```
f=zeros(1,50);
f(1)=1;f(2)=2;
parfor n=3:50
   f(n)=f(n-1)+f(n-2);
end
```

Dependencia de datos entre iteraciones

Limitaciones del parfor

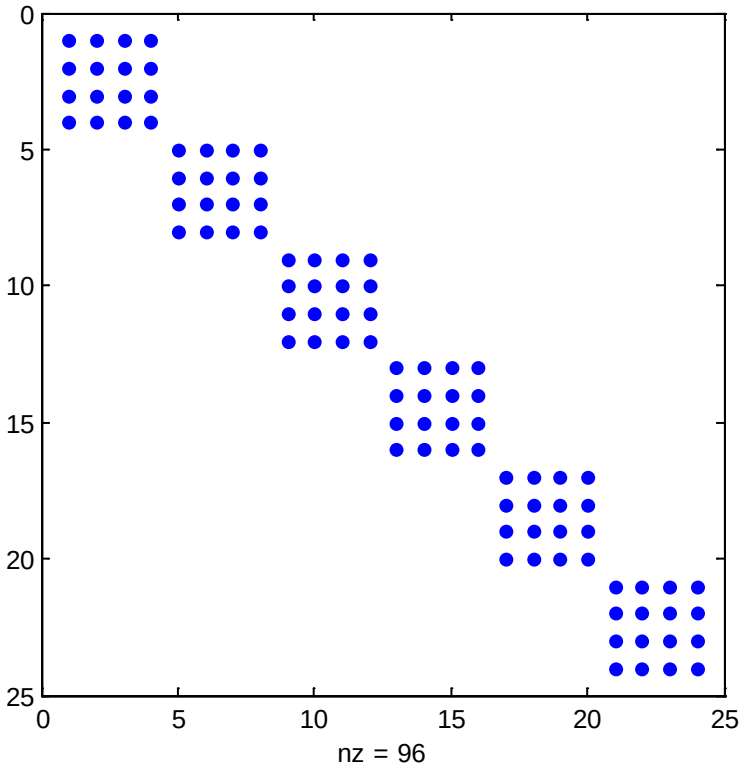
-No se pueden anidar parfor; sí se puede anidar for con parfor
-Un parfor no puede contener un spmd, ni a la inversa.
-Un parfor no puede contener instrucciones “break” o “return”
-Matlab debe ser capaz de “clasificar” las variables en el código, de forma que cada variable tenga un solo tipo:

```
N=10;
d=0;
c=0;
B = zeros(1,N);
R = rand(1,N);
parfor i=1:N
  d = d + i;
  a = i
  B(i) = R(i) + c
end
a
```



Instrucción Parfor: Ejemplo

Supongamos que tenemos una matriz como la de la figura, y deseamos hacer algo con cada uno de los bloques de la diagonal



Instrucción Parfor: Ejemplo

Supongamos que tenemos una matriz A como la de la figura, y deseamos hacer algo con cada uno de los bloques de la diagonal

```
A=migenmat(5,4)
tambloque=4;
parfor i=1: 5
   ini_bloque=(i-1)*tambloque+1
   fin_bloque=i*tambloque
   C (ini_bloque:fin_bloque, ini_bloque:fin_bloque) =lu(A(ini_bloque:fin_bloque,
ini_bloque:fin_bloque))
end
```

NO FUNCIONA

```
A=migenmat(5,4)
tambloque=4;
C=cell(5,1);
parfor i=1: 5
   ini_bloque=(i-1)*tambloque+1
   fin_bloque=i*tambloque
   C{i} =lu(A(ini_bloque:fin_bloque, ini_bloque:fin_bloque))
end
```

SÍ FUNCIONA (aunque no nos da el resultado directamente en la matriz C, sino en un cell array)

Instrucción Parfor: Discusión

Problemática del parfor:
-Parfor debe ser capaz de repartir los datos entre los labs, mediante inspección del código. Debe funcionar tanto para el caso (relativamente trivial) de una sóla máquina, como para el caso bastante complicado de varias máquinas.

-Al asignar valores a una variable “sliced” (lado izquierdo de una asignación), sólo uno de los índices puede depender de la variable del parfor (i).

-Costes parfor (comparando con un for):
Partir el conjunto de iteraciones
Clasificar variables
Enviar datos y códigos a los workers
Envío de los resultados de los workers al cliente.
Cliente combina resultados parciales de los trabajadores.

Al usar el parfor es recomendable (en la medida de lo posible) evitar el envío de datos

Instrucción Parfor: Discusión

Cuando usar parfor?

-Cuando el tiempo de cálculo sea bastante mayor que el tiempo de comunicación + overhead

-Suele ser cuando el número de iteraciones es bastante grande.

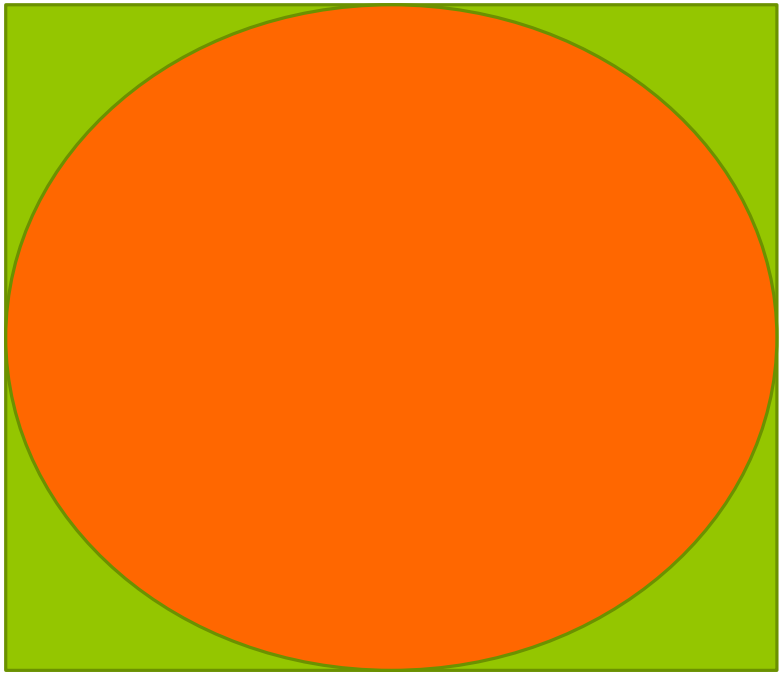
Al usar el parfor es recomendable (en la medida de lo posible) evitar el envío de datos

Las funciones ticBytes(gcp) y tocBytes(gcp) (se usan como tic y toc, pero para medir datos) permiten comprobar cuantos datos se envían a cada worker. Gcp devuelve el pool actual.

Instrucción Parfor: ejercicio (1)

Problema: Cálculo de pi usando el método Montecarlo

La razón de las areas de un círculo y el cuadrado que lo contiene es siempre pi/4.



Instrucción Parfor: ejercicio (1)

Problema: Cálculo de pi usando el método Montecarlo

Si coges N puntos aleatorios, aproximadamente N*pi/4 de esos puntos estarán en el círculo. El método Montecarlo genera un gran número de puntos aleatorios, y contamos cuantos de esos puntos están dentro. Pi se puede estimar como (Nº puntos_dentro)*4/total. La estimación mejorará cuantos mas puntos usemos.

-Disponemos de versiones secuenciales (compute_pi_matlab.m), con bucle (compute_pi_for.m), y con bucle, generando los puntos aleatorios dentro del bucle.(compute_pi_for_inside.m)

-genera una versión paralela a partir de compute_pi_for.m. Comparala con la secuencial para diferentes valores de N, grandes (N>=10000000)

-Comprueba con ticBytes y TocBytes cuantos datos se han enviado.

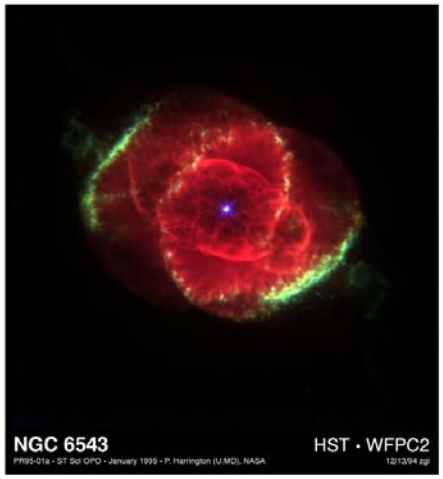
-Prueba con compute_pi_for_inside.m

-Prueba todo usando pool('threads')

Instrucción Parfor: ejercicio 2

Una imagen en formato ppm se guarda como un array tridimensional de enteros (uint8). Si la imagen se guarda en la variable im, en Matlab el array bidimensional im(:,:,1)contiene los valores “rojos” de la imagen, el array im(:,:,2) contiene la G y el array im(:,:,3) contiene la B.

>>matr=imread('ngc6543a.jpg'); imshow(matr);



Instrucción Parfor: ejercicio 2

```
function [ imagen_out ] = difumina( imorig )
[m,n,tam]=size(imorig)
im=double(imorig);
auxj=n-1;
auxi=m-1;
imagen_out=zeros(size(im),'uint8');
for ind=1:tam
   for j=2:auxj
      for i=2:auxi
        aux=double((1.0/9.0)*(im(i-1,j-1,ind)+im(i-1,j,ind)+im(i-1,j+1,ind)+...
            im(i,j-1,ind)+im(i,j,ind)+im(i,j+1,ind)+im(i+1,j-1,ind)+...
            im(i+1,j,ind)+im(i+1,j+1,ind)));
         aux=max(0,uint8(aux));
        imagen_out(i,j,ind)=min(255,aux);
      end
   end
end
end
```

```
>>mat_d=difumina(matr); imshow(mat_d);
```

Prueba diferentes paralelizaciones con parfor

Instrucción SPMD (single program, multiple data)

spmd
...
end

-Crea una región de código que es ejecutada por los workers disponibles, creados previamente con parpool

-Cada worker se identifica mediante la variable “labindex”. Cada worker se ejecuta en un core separado y tiene su propio espacio de trabajo

-La variable “numlabs” da el número de labs activos dentro de la región paralela. Cada core puede enviar datos a los otros cores.

-El cliente puede examinar datos de los cores.

Instrucción SPMD (single program, multiple data)

```
spmd
  a=ones(10);
end
```

Este código hace que en cada worker se genere una matriz 10 por 10. El cliente tiene acceso a las variables en los workers, por ejemplo, a{3}.

Instrucción SPMD (single program, multiple data)

Se puede controlar el número de labs que se crean:

```
spmd(3)
...
end
```

Crea tres labs (suponiendo que se ha ejecutado parpool y se han creado 3 o mas labs)

```
spmd(2,5)
...
end
```

(mínimo 2, máximo 5)

Instrucción SPMD (single program, multiple data)

Ejemplo 1: Resolver 800 ecuaciones de segundo grado, en paralelo

```
Clear          %archivo ejemplo3.m
N=800;A=rand(N,1);B=rand(N,1);C=rand(N,1);
spmd
  trozo=N/numlabs
  ini=(labindex-1)*trozo+1;
  fin=labindex*trozo
  for i=ini:fin
    sol(i,:)=roots([A(i),B(i),C(i)]);
end
end
```

El cliente ejecuta hasta el spmd, luego se para y espera a que los workers ejecuten el spmd

Instrucción SPMD (single program, multiple data): Variables **Composite**

Al ejecutar este código, y comprobar el "workspace" de MATLAB, observamos que las variables trozo, fin, ini, y especialmente sol, son de tipo **composite**.

Cuando en una región paralela se usan nuevas variables, estas se definen como variables diferentes en cada "lab", pero el mismo nombre para todos: estas son variables **composite**. Por ejemplo, la variable trozo en el lab 5 es: trozo{5} (Ojo, en Matlab existen los cell arrays que se referencian de la misma forma).

Podemos averiguar el número de elementos de un composite con la función "numel". Si la variable se ha creado en todos los workers, numel de esa variable me dará el número de workers.

Instrucción SPMD (single program, multiple data)

```
clear
N=800;A=rand(N,1);B=rand(N,1);C=rand(N,1); solf=zeros(N,2);

spmd
sol=zeros(N,2);
    trozo=N/numlabs;
    ini=(labindex-1)*trozo+1;
    fin=labindex*trozo
   for i=ini:fin
       sol(i,:)=roots([A(i,1),B(i,1),C(i,1)]);
   end
end

for i=1:numel(ini)
   init=ini{i}
  fint=fin{i}
  aux=sol{i};
  solf(init:fint,1:2)=aux(init:fint,1:2);
end
```

Este trozo recoge los resultados de cada lab y los guarda en la variable solf

Instrucción SPMD (single program, multiple data), V3

```
N=800
solf=zeros(N,2);
spmd
    trozo=N/numlabs;
    sol=zeros(trozo,2); %generamos dentro sol, A,B,C
     A=rand(trozo,1);B=rand(trozo,1);C=rand(trozo,1);
    ini=(labindex-1)*trozo+1;
    fin=labindex*trozo;
   for i=1:trozo
       sol(i,:)=roots([A(i,1),B(i,1),C(i,1)]);
   end
end

for i=1:numel(ini)
   init=ini{i};
  fint=fin{i};
  %aux=sol{i};
  solf(init:fint,1:2)=sol{i};
end
```

Instrucción SPMD (single program, multiple data):

```
a = 3;
b = 4;
spmd
c = labindex();
d = c + a;
end |
e = a + d{1};
c{2} = 5;
spmd
f = c * b;
end
```

```
Cliente    Worker 1    Worker 2
  a b e  | c d f | c d f
---------------------------
  3 - -  | - - - | - - -
  3 4 -  | - - - | - - -
         |       |
  3 4 -  | 1 - - | 2 - -
  3 4 -  | 1 4 - | 2 5 -

  3 4 7  | 1 4 - | 2 5 -
  3 4 7  | 1 4 - | 5 6 -
         |       |
  3 4 7  | 1 4 4 | 5 6 20
```

Instrucción SPMD (single program, multiple data)

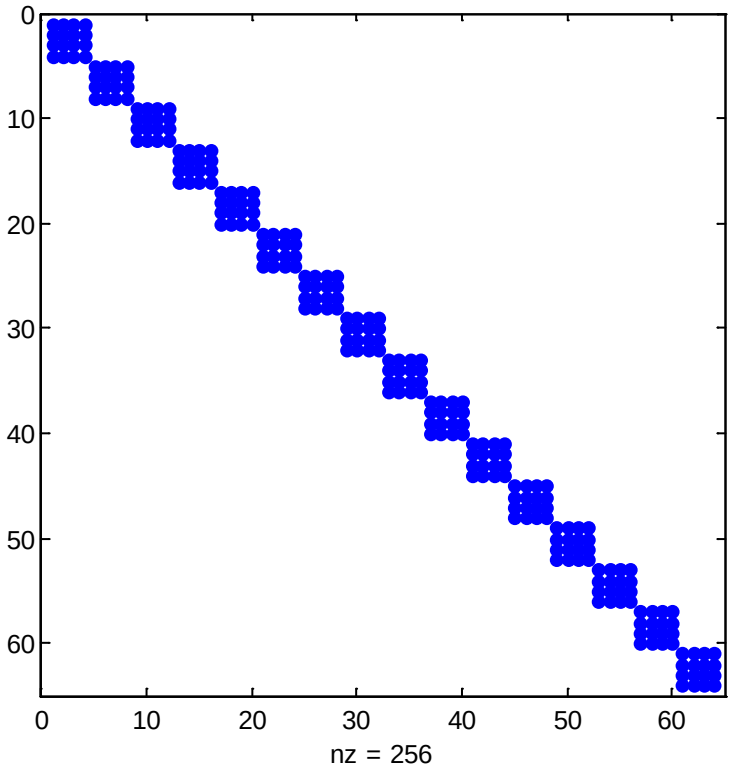
Las variables en un bloque spmd son **persistentes** entre diferentes bloques spmd.

```
spmd
 R=ones(10);
end

spmd
  B=R+1;
end
```

Instrucción SPMD (single program, multiple data) Ejemplo de matriz a bloques

```
>>A=genmat(16,4);
>>spy(A)
```



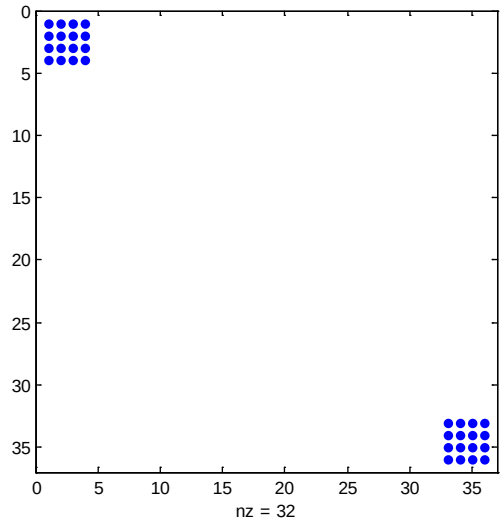
Instrucción SPMD (single program, multiple data) Ejemplo de matriz a bloques

```
A=genmat(16,4)
tambloque=4;
spmd
  for i=1:16
    if mod(i,numlabs)==labindex   % si esto es cierto, el bloque i es
                                  % procesado por labindex
      ini_bloque=(i-1)*tambloque+1
      fin_bloque=i*tambloque
  C (ini_bloque:fin_bloque, ini_bloque:fin_bloque)
=lu(A(ini_bloque:fin_bloque, ini_bloque:fin_bloque))
    end
  end
end
```

Instrucción SPMD (single program, multiple data) Ejemplo de matriz a bloques

Observemos como queda la matriz resultado C, para el lab 1:

>>spy(C{1})



Vemos que cada “lab” hace su parte correctamente , pero probablemente sea necesario “traer” el resultado de cada “lab”, si queremos el resultado en una sola matriz.

Instrucción SPMD (single program, multiple data)

Ejemplo útil: Cargar un archivo de datos diferente para cada lab, usando labindex, y procesarlo independientemente:

```
spmd (3)
  datos_lab=load(['datos_', num2str(labindex), '.dat'])
  result=funcion_calculo(datos_lab)
end
```

O, si hay N archivos (mas que el número de cores):

```
spmd
  for i=1:N
     if mod(i,numlabs)==labindex
        datos_lab=load(['datos_', num2str(labindex), '.dat'])
         result(i)=funcion_calculo(datos_lab)
     end
  end
end
```