Introducción a Parallel toolbox de Matlab

Víctor M. García

Contenidos

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

Ventajas Y Desventajas del Parfor

Ventajas: Claramente, simplicidad y facilidad de uso:

Desventajas:
-Solo se puede hacer paralelismo basado en bucles
-Rigidez
-No sabemos como divide Matlab las iteraciones
-No podemos saber que worker ejecuta que iteración
-No podemos examinar el trabajo de un trabajador concreto
-Usando parfor, los trabajadores son “anónimos”

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)

Fuera de una región spmd, el numero de workers activos del pool (poolsize) se puede averiguar así:

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

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".

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=floor(N/numlabs);
    if labindex<numlabs
      ini=(labindex-1)*trozo+1;
      fin=labindex*trozo;
    else
      ini=(labindex-1)*trozo+1;
      fin=N;
    end
    trozo2=fin-ini+1;
    sol=zeros(trozo2,2); %generamos dentro sol, A,B,C
     A=rand(trozo2,1);B=rand(trozo2,1);C=rand(trozo2,1);
   for i=1:trozo2
       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)

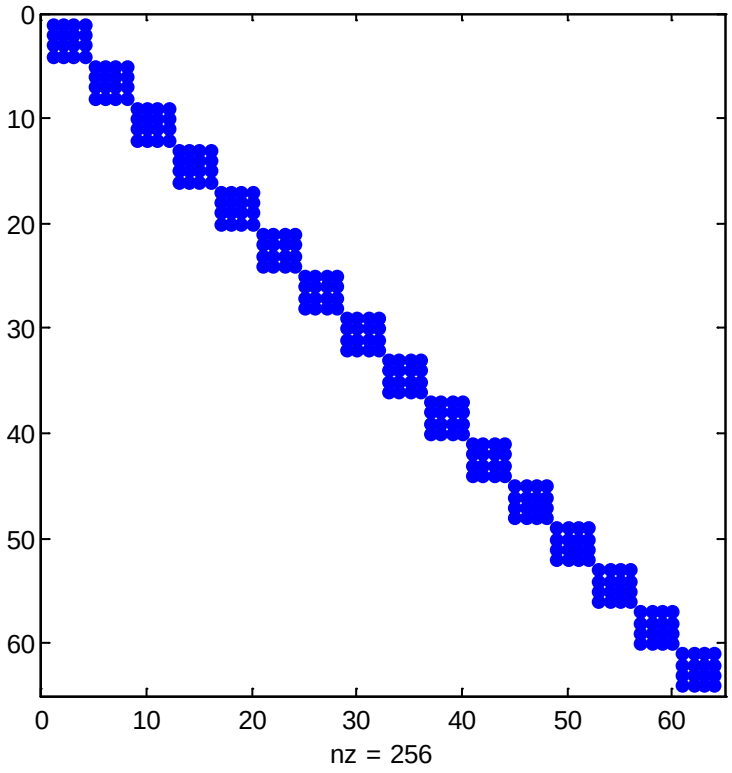
Las variables en un bloque spmd son **persistentes** entre diferentes bloques spmd.

```
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)
```



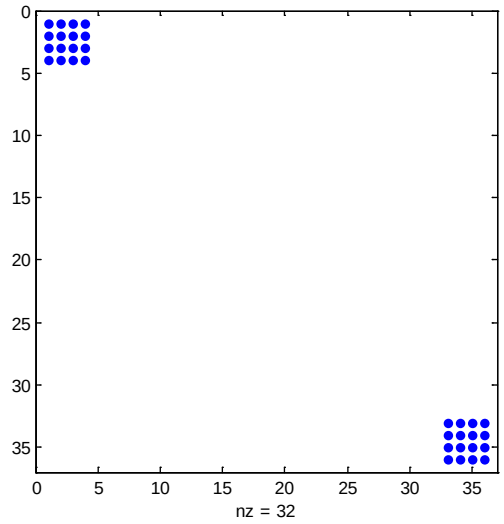
Instrucción SPMD (single program, multiple data) Ejemplo de matriz a bloques

```
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

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
```

Instrucción SPMD

Ejercicio :

1) Crea una nueva versión del programa para calcular pi por el método de Montecarlo, usando spmd.

El cliente debe combinar los resultados parciales (elementos contados en cada worker) iterando sobre la variable correspondiente en los workers.

Instrucción SPMD

Comunicación entre Workers de Matlab:

labSend(var,id) % Envia al worker “id” la variable “var”

Var=labReceive(id) % Recibe la variable Var enviada desde worker “id”

varFrom=labsendReceive(idTo,idFrom, varTo) %manda y recibe a la vez

Instrucción SPMD

Ejercicio (3):

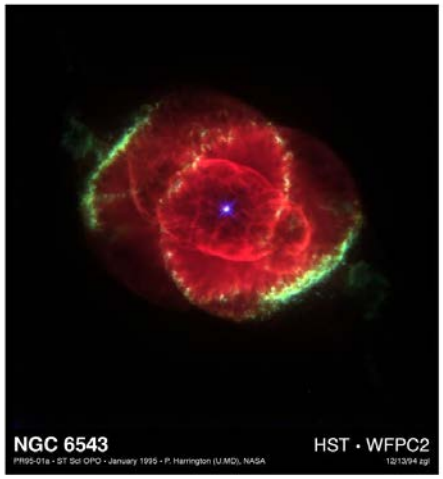
1) Crea una nueva versión del programa para calcular pi por el método de Montecarlo, usando spmd.

Hazlo de forma que cada worker envía su resultado parcial (elementos contados en cada worker) al siguiente worker. El último worker contendrá el resultado final, y por lo tanto el cliente tiene que leer sólo el resultado del último worker.

Instrucción SPMD

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 spmd: ejercicio 3

```
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);
```

Paraleliza esta función con spmd

Arrays “distribuidos”

Si hay un “pool” abierto, el cliente puede crear arrays “distribuidos de diferentes formas.

1) Con la función “distributed”, aplicada a un array existente en el cliente.

```
>>B=distributed(A)
```

2) Con las funciones disponibles para generar directamente arrays distribuidos:

```
>> A=distributed.eye(100);
>>B=distributed.ones(30,500);
>>C=distributed.rand(2,9);
```

etc.

Para ver la lista entera, “help distributed”.

Arrays “distribuidos”

-Se pueden distribuir arrays de cualquier tipo y estructura
-cada Worker contiene parte del array
-cada worker opera sólo su propia parte del Array
Podemos operar con el array completo como una sola entidad:

```
A=distributed.rand(1000); B=distributed.rand(1000);
tic
C=A*B;
toc;
```

Arrays “distribuidos”

La distribución se hace a lo largo de la última dimensión. En el caso de una matriz, por columnas.

Ejemplo, si a es un array de 300 por 400, distributed(a) sobre un pool de 4 workers, queda así:

```
          Worker 1   Worker 2  Worker 3  Worker 4
        Col: 1:100  |  101:200  |  201:300  |  301:400 ]
Fila
      1 [* * … * | * * … * | * * … * | * * … * ]
      2 [* * … * | * * … * | * * … * | * * … * ]
    ... [* * … * | * * … * | * * … * | * * … * ]
    300 [* * … * | * * … * | * * … * | * * … * ]
```

Ejemplo conos: vectorización

%El volumen de un cono se calcula, dados su Diametro D y su
% altura H, como V = 1/12*pi*(D^2)*H;
% Queremos calcular el volumen de 1.000.000 conos, dados
%1.000.000 Diametros y 1.000.000 alturas:

```
n=1000000;
D=rand(1,n);
H=rand(1,n);
% Versión con bucle
tic
for i=1:n
V(i) = 1/12*pi*D(i)^2*H(i);
end
tiempo_bucle=toc
% Versión vectorizada
tic
V=(1/12)*pi*D.^2.*H; % Observa el uso del punto antes
del operador
tiempo_vectorizado=toc
```

Arrays “distribuidos”; ejemplo conos

```
numero_conos=1000000;
% generamos aleatoriamente Diametros y alturas
D=distributed.rand(1,numero_conos)+10;
H=distributed.rand(1,numero_conos)+10;
V=distributed.zeros(1,numero_conos);
tic;    V = (1/12)*pi*D.^2.*H; toc
```

Arrays “distribuidos”

Funciones útiles para trabajar con arrays distribuidos:

-getLocalPart(A); llamada por un trabajador, en un spmd, devuelve la parte “local” del array distribuido.

-gather(A): Si la llama el cliente, devuelve la versión “no distribuida” de la matriz A; Si la llama un worker, crea una copia local de A en ese worker.

-isdistributed(A) devuelve (verdadero/falso) si el array (está/no está) distribuido.

Arrays “distribuidos +Instrucción SPMD

Es habitual combinar spmd con arrays distribuidos:
Una vez que el cliente ha distribuido la matriz con el comando

```
ad = distributed ( a );
```

Entonces cada worker puede hacer una copia de su parte local con la función “getLocalPart”

```
spmd
al = getLocalPart ( ad );
[ ml, nl ] = size ( al )
end
```

Ojo, los índices locales y globales serán diferentes

Al acabar se puede “recoger” la matriz distribuida: gather(ad)

Instrucción SPMD (single program, multiple data)

Ejercicio (3):

1) Crea una nueva versión del programa para calcular pi por el método de Montecarlo, sin spmd ni parfor, usando arrays distribuidos

Instrucción SPMD (single program, multiple data)

Es habitual combinar spmd con arrays distribuidos:
Una vez que el cliente ha distribuido la matriz con el comando

```
ad = distributed ( a );
```

Entonces cada worker puede hacer una copia de su parte local con la función “getLocalPart”

```
spmd
al = getLocalPart ( ad );
[ ml, nl ] = size ( al )
end
```

Ojo, los índices locales y globales serán diferentes

Al acabar se puede “recoger” la matriz distribuida: gather(ad)

Instrucción SPMD (single program, multiple data)

Probar esto:

```
A=ones(10);

Ad=distributed(A);
spmd
   Al=getLocalPart(Ad)
   [m,n]=size(Al);
   Ad=Ad*labindex();
end
An=gather(Ad)
```

drange

Si dentro de un spmd necesitamos un bucle sobre la parte distribuida (ejemplo de las raíces), en principio tenemos for sobre rango distribuido (drange)

```
A=ones(10);
Ad=distributed(A);
spmd
   Al=getLocalPart(Ad)
   [m,n]=size(Al);
   for i=drange(1:10)
      Ad(:,i)=Ad(:,i)*i;
   end
end
An=gather(Ad)
```

Sin embargo, drange es MUY LENTO (averiguado por experimentación)

drange

```
N=800;A=distributed.rand(1,N);
B=distributed.rand(1,N);
C=distributed.rand(1,N);
sol=distributed.zeros(2,N);
tic
spmd
   for i=drange(1:N)
     sol(:,i)=roots([A(i),B(i),C(i)]);
  end
end
toc
```

Es mejor extraer las partes locales con getlocalpart

Arrays “Distribuidos” y Arrays “codistribuidos”

Cuando el cliente posee un array y lo “distribuye” entre los workers, decimos que el array está distribuido.

-Si el array es demasiado grande para la memoria del cliente, no se puede usar este método. Es necesario generar el array directamente en la memoria de los workers (Tiene sentido si estamos accediendo a un cluster externo, con el Distributed Computing Server)

Cuando generamos el array directamente en los workers, el array es “codistribuido”.

Una vez generado, da igual si el array es distribuido o codistribuido.

Generar Arrays “codistribuidos”

```
spmd
    A=[11:20;21:30;31:40];
    D=codistributed(A);
   getLocalPart(D)
end
```

En este caso, la matriz A es la misma en todos los workers. El array D es igual al A, pero repartido entre los diferentes workers.

```
spmd
    Id=codistributed.eye(8)
  getLocalPart(Id)
End
```

En principio se distribuye por columnas; sin embargo, es posible modificar eso de muchas formas, con la función “codistributor”

Generar Arrays “codistribuidos”

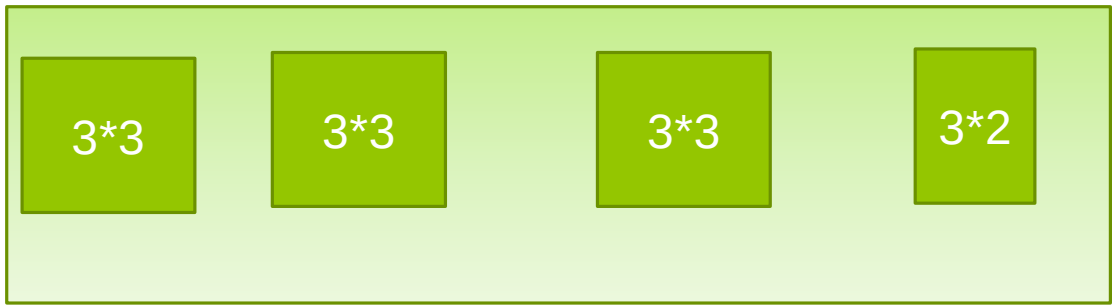
Veamos como ensamblar diferentes arrays locales en uno sólo, codistribuido.
Vamos a ver sólo distribución unidimensional, con la función “codistributor1d”, que crea un “codistribuidor”

Supongamos que tenemos 4 workers, los tres primeros con arrays A_parcial de 3 filas por 3 columnas y el cuarto worker con array A_parcial de 3 filas por 2 columnas: Queremos ensamblarlos en un array codistribuido A_total de 3 filas por 11 columnas:

Generar Arrays “codistribuidos”

Los argumentos de codistributor1d son:
-la dimensión a lo largo de la cual se distribuye (en este caso por columnas ,segunda dimensión)
-Vector con las columnas que se cogen en cada worker [3, 3, 3, 2]
-Dimensiones de la matriz final: [3,11]

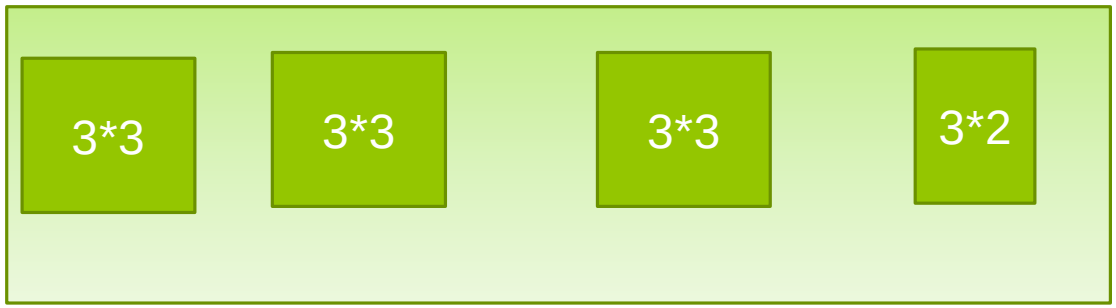
Hay que hacer la llamada
>>codist =codistributor1d(2,[3,3,3,2], [3,11])
Y a continuación usamos el codistributor
>>Atotal=codistributed.build(A_parcial, codist)



Generar Arrays “codistribuidos”

Podremos operar con A_total como una matriz normal;

Como en el caso de Arrays distribuidos podemos obtener su parte local (getLocalPart) o traerla a un worker o al cliente (gather)



3*11

Ejercicio:

Dado el ejemplo de calcular raíces:

```
N=800;A=distributed.rand(1,N);
B=distributed.rand(1,N);
C=distributed.rand(1,N);
sol=distributed.zeros(2,N);
tic
spmd
   for i=drange(1:N)
     sol(:,i)=roots([A(i),B(i),C(i)]);
  end
end
toc
```

Distribuir previamente los arrays de entrada, obtener la parte local dentro del spmd, codistribuir el array de salida y “recogerlo” en uno sólo

Pmode

Pmode es algo así como un spmd “interactivo”.

El pool debe estar cerrado

>>pmode start

También

>>pmode start 4

Se pueden introducir comandos y se ejecutan en todos los workers a la vez. (Prueba esto:
P>>a=ones(5)
P>>a=a*labindex;

No se puede ejecutar pmode si Matlab está corriendo en “modo texto”.

Pmode

Es posible usar arrays codistribuidos, como en los ejemplos anteriores.

Se puede enviar datos del cliente a los workers o de los workers al cliente:

```
>> pmode lab2client A 3
                %Envia la matriz A del worker 3 al cliente

>> pmode client2lab y 1:2
                %Envía la variable y del cliente a los workers 1 y 2;
                % en el  3 y 4, la variable y queda indefinida.
```

mpiprofile

Tenemos una versión del profiler de Matlab para programas paralelos; puede funcionar con spmd o con pmode (no con parfor).

```
spmd
   R1=rand(16, codistributor());
   R2=rand(16, codistributor());
   mpiprofile on
   P=R1*R2;
   info = mpiprofile('info');
   mpiprofile off;
   mpiprofile viewer;
end
```