Introducción a Parallel toolbox de Matlab

Víctor M. García

Contenidos

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

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 [* * … * | * * … * | * * … * | * * … * ]
```

Arrays “distribuidos”; ejemplo conos

```
%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 100000 conos, dados 100000
Diametros y
% 100000 alturas:
numero_conos=100000;
% 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
```

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)

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.

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
  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 mucho mas rápido 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”

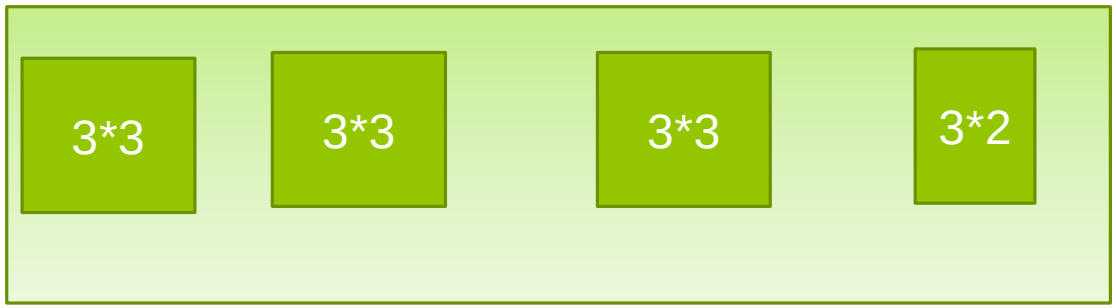
Veamos como ensamblar diferentes arrays locales en uno sólo, codistribuido.
Vamos a ver sólo distribución unidimensional, con la función “codistributor1d”, que crea un “codistribuidor”

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]

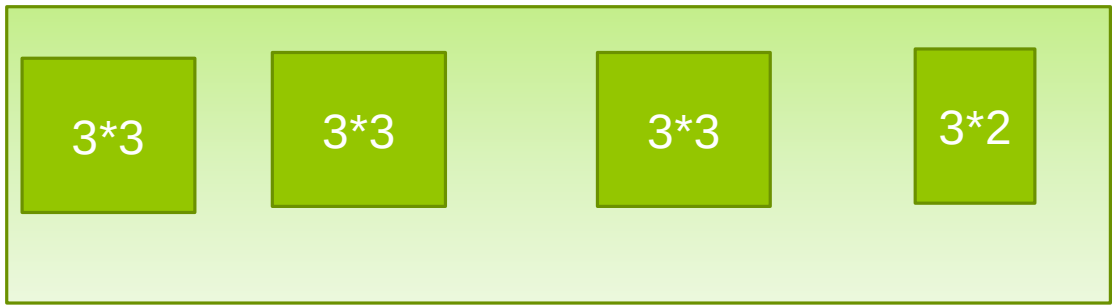
Hay que hacer la llamada
>>codist =codistributor1d(2,[3,3,3,2], [3,11])
Y a continuación usamos el codistributor
>>Atotal=codistributed.build(A_parcial, codist)



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
```

Computación en GPU desde MATLAB

Es posible utilizar Graphics Processing Units desde MATLAB, si se tiene disponible el Parallel toolbox.

-Tiene que ser GPUs NVIDIA, preferiblemente recientes.

-Hay que instalar un driver reciente (ultimo) de Nvidia para la tarjeta gráfica.

-Idealmente, para trabajar con Windows y dedicar la tarjeta gráfica sólo a cálculo, deberíamos tener dos tarjetas gráficas. Sin embargo, en la actualidad es complicado configurar un sistema así.

-Bastante más sencillo en Linux. Podemos usar los PCs del laboratorio o conectarnos a gpu.dsic.upv.es (o a knights, aunque da problemas)

Computación en GPU desde MATLAB

Para averiguar cuantas GPUs tenemos disponibles:

```
>> gpuDeviceCount
```

Para averiguar sus propiedades:

```
>> gpuDevice
```

O, si tenemos 2:

```
>>gpuDevice(1)
```

```
>>gpuDevice(2)
```

Computación en GPU desde MATLAB

Función gpuArray: Envía los datos a la GPU; lo que se haga con esos datos se hará en la GPU;

Restricción: Las matrices que se envíen deben ser DENSAS;
De momento, Matlab + GPU no trabajan con matrices dispersas.

Al acabar, se pueden recuperar los datos de la GPU con “gather”

Computación en GPU desde MATLAB: Posibilidad 1

Función gpuArray, combinado con funciones de MATLAB:

Ejemplo, calcular la factorización LU de una matriz densa

```
>> A=rand(5000);
>>tic, ,[l,u,p]=lu(A,'vector');toc
>> GA=gpuArray(A);
>> tic,[l,u,p]=lu(GA,'vector');toc
```

Ahora, repetimos con precisión simple:

```
>> A=rand(5000,'single');
>>tic, ,[l,u,p]=lu(A,'vector');toc
>> GA=gpuArray(A);
>> tic,[l,u,p]=lu(GA,'vector');toc
```

Ejercicio

Cálculo del número pi: Crea una nueva versión del programa para calcular pi por el método de Montecarlo, con gpuArray. A partir de compute_pi_Matlab (versión vectorizada)

Computación en GPU desde MATLAB: Posibilidad 2: Arrayfun

Arrayfun: es una forma “vectorizada” de llamar a funciones de Matlab, escritas por nosotros. Conceptualmente, es ejecutar muchas veces la misma función con diferentes datos.

Ejemplo: Función que devuelva una de las dos raíces de una ecuación de segundo grado:

```
function  x=miraiz (a,b,c)
  sol=roots([a,b,c]);
  x=sol(1);
end
```

Computación en GPU desde MATLAB: Arrayfun

A la función miraiz la podemos llamar (con números) de diferentes formas:

>>x=miraiz(1,2,3);

>>x=feval('miraiz',1,2,3)

>>x=feval(@miraiz,1,2,3)

En lugar de llamarla con números podemos pasarle vectores (todos de la misma dimensión) usando arrayfun:

Computación en GPU desde MATLAB: Arrayfun

Para ejecutar con Arrayfun

```
>>N=800;A=rand(N,1);B=rand(N,1);C=rand(N,1);
>>sol=arrayfun(@miraiz,A,B,C)
```

Arrayfun asume que las dimensiones de los arrays de entrada y de salida son las mismas; si la función "miraiz" devuelve un vector con las dos soluciones de la ecuación de segundo grado:

```
function  x=miraiz2 (a,b,c)
  x=roots([a,b,c]);
End
```

Entonces hay que llamarla con la opción UniformOutput a 0:

```
>>sol=arrayfun(@miraiz2,A,B,C,'UniformOutput',0)
```

(Ojo no va con gpus, por culpa de la función "roots")

Computación en GPU desde MATLAB: Arrayfun

Si los datos están en la CPU, arrayfun se ejecuta en la CPU;

Si los datos están en la GPU, arrayfun se ejecuta en la GPU

Esto permite extraer una gran parte del potencial de las GPUs

```
function [ x ] = miraizg( a,b,c )
x=(-b+sqrt(complex(b*b-4*a*c,0)))/(2*a)
end
```

```
>>Ag=gpuArray(A);
>>Bg=gpuArray(B);
>>Cg=gpuArray(C);
>>sol=arrayfun(@miraizg,Ag,Bg,Cg)
>>solgpu=gather(sol);
```

Computación en GPU desde MATLAB: Posibilidad 2

Ejercicio (3):

1) Crea una nueva versión del programa para calcular pi por el método de Montecarlo, usando arrayfun para calcular el vector sal, tal que sal(i)=1 si (x(i)^2+y(i)^2<1), y sal(i)=0 si (x(i)^2+y(i)^2>=1).

Partimos de compute_pi_for.m, pero quitando el bucle.

Computación en GPU desde MATLAB: Posibilidad 2

Arrayfun con matrices:
Arrayfun funciona también si tenemos un doble bucle que recorre una matriz

```
N=10; M=20;A=gpuArray.rand(N,M);B=gpuArray.rand(N,M);
C=gpuArray.rand(N,M);
for i=1:N
    for j=1:M
        sal(i,j)=miraizg(A(i,j),B(i,j),C(i,j));
     end
end
```

El doble bucle se puede sustituir por:

```
sal=arrayfun(@miraizg,A,B,C);
```

Computación en GPU desde MATLAB: Posibilidad 2

Caso especial: Arrayfun con matrices, accediendo dentro de la función a diferentes elementos de una matriz.

Arrayfun , en su forma básica, no es apropiado para acceder elementos de una matriz

```
function [ x ] = miraizg( a,b,c )
x=(-b+sqrt(complex(b*b-4*a*c,0)))/(2*a)
end
```

Las referencias a elementos de a,b,c,x deben ser escalares, sin subíndices. Así no podemos referenciar elementos de una matriz (por ejemplo, función difumina).

Computación en GPU desde MATLAB: Posibilidad 2

Es posible referenciar elementos de una matriz usando una técnica nueva, con dos detalles fundamentales:

1) Hay que usar “funciones anidadas” (Se pueden crear dentro de funciones, no dentro de scripts). Ejemplo: juego_vida_arrayfun.m

La variable **grid** (que es una matriz, de la cual queremos referenciar sus elementos) no se pasa como argumento a “Updateparent”, sino que es una variable de la función “juego_vida_arrayfun”, la cual contiene a “Updateparent”

Computación en GPU desde MATLAB: Posibilidad 2

2) Pasamos como argumentos vectores de índices.
La cabecera de la función updateparent:

function X = updateParent(row, col)

Dentro de updateparent, row funciona como índice de filas y col como índice de columnas.

X debe tener dimension M filas por N columnas. Entonces, en la llamada a updateParent,
-rows es un vector **columna** (dimension M por 1) con los índices de 1 a M (sería el vector columna con valores (1,2,3,...,M-1,M))
-cols es un vector fila (dimension 1 por N) con los índices de 1 a N (Igual que rows, pero como vector fila y de 1 a N)

Computación en GPU desde MATLAB: Posibilidad 2

Comparamos las opciones en arrayfun para ver las diferencias:

V1: Cuando llamamos a arrayfun pasándole vectores de la misma dimensión N , arrayfun realiza N llamadas a la función subyacente

V2: Cuando llamamos a arrayfun pasándole matrices de la misma dimensión M*N , arrayfun realiza M*N llamadas a la función subyacente (Como un doble bucle)
Pero no podemos acceder a los elementos de la matriz.

V3: Cuando llamamos a arrayfun pasándole un vector (rows en el ejemplo) **columna** de indices de fila (1…M) y un vector (columns en el ejemplo) **fila** de índices de columna(1..N), también hace M*N llamadas a la función subyacente. Se puede usar para acceder a los elementos de una matriz, pero la matriz debe ser “global”, no se puede pasar como argumento a la función subyacente.
También se pueden pasar variables “escalares” como argumentos de la función subyacente

Computación en GPU desde MATLAB: Posibilidad 2

Ejercicio: En Poliformat está la función difumina5, una versión simplificada de la función difumina que usamos en sesiones anteriores. Utiliza la técnica usada en juego_vida_arrayfun.m para ejecutar los cálculos en la GPU con arrayfun.

Vamos a sustituir los dos bucles mas internos por arrayfun, y dejamos el externo (ind=1:tam, tam es igual a 3) casi igual, debe quedar así:

```
for ind=1:tam
  imagen_out(:,:,i)=.... //llamada a arrayfun
End
```

Para cargar y visualizar la imagen, hacíamos esto:
matr=imread('ngc6543a.jpg'); imshow(matr);