MATLAB Parallel Computing Toolbox

Shaohao Chen
Research Computing Services
Information Services and Technology
Boston University
Why using Matlab Parallel Computing Toolbox (PCT)?

- To accelerate a MATLAB program, it is necessary to parallelize it.
- Take advantages of high-performance computing (HPC) resources, such as multi-core CPUs, GPUs, and computer clusters.
- Boston University (BU) Shared Computing Cluster (SCC) is an HPC cluster with over 11,000 CPU processors and over 250 GPUs.
- MATLAB site license is available to all BU users. (Unlimited on SCC).
- The PCT can be used not only on HPC clusters or but also on regular laptops/desktops.
- The skills you learn today should enable you to solve bigger problems faster using MATLAB.
Outline

- Start up MATLAB on BU SCC
- Parallelize Matlab codes
  - Implicit parallelism
  - Explicit parallelism
  - Using GPU
Access to BU SCC resources

◦ Log in:
$ ssh -X username@scc2.bu.edu

◦ Interactive session (for working interactively on compute nodes):
$ qrsh  # Start an interactive session
$ qrsh -pe omp 4  # Request 4 CPU cores
$ qrsh -1 gpus=1  # Request one GPU and one CPU core

◦ Use module to load Matlab:
$ moduleavail | grep matlab  # See all available versions
$ module load matlab  # Set up environment variables
- Open Matlab
  $ matlab &

- Use VNC to speed up graphical interface.

Refer to: https://www.bu.edu/tech/support/research/system-usage/getting-started/remote-desktop-vnc/
M-file

- An m-file is a simple text file where you can place MATLAB commands.
- Save your works
- Convenient for debugging
- Run directly. Pre-compiling is unnecessary.

```matlab
%% find a root of the polynomial x^3 - 2x - 5, 

a = 0; fa = -Inf;
b = 3; fb = Inf;
while b-a > eps*fb
    x = (a+b)/2;
    fx = x^3-2*x-5;
    if fx == 0
        break  %% already found the root, exit the loop
    elseif sign(fx) == sign(fa)
        a = x; fa = fx;
    else
        b = x; fb = fx;
    end
```
Text platform

$ matlab -nodisplay % Work in text interface. Does not display any graph.
$ matlab -nodesktop % Program in text interface. Pop out graphs when necessary.

- Many Linux commands (prefix an exclamation mark) are available within Matlab platform, such as:
  cd, ls, pwd, !cp, !rm, !mv, !cat, !vim, !diff, and !grep

- Edit M-file and run the program:
  >> !vim mfilename.m % edit in text window
  >> edit mfilename.m % create a new or open an existing m-file in graphical window
  >> open mfilename.m % open an existing m-file in graphical window
  >> run mfilename.m % run the program
  >> mfilename % run the program
Parallelize Matlab codes

- Parallel computing:
  run multiple tasks by different workers simultaneously.

- Matlab parallel computing toolbox (PCT)
  - Implicit parallelism: automatic multi-threaded vector operations
  - Explicit parallelism: `parpool`, `parfor`, `spmd`
  - Using GPU: `gpuArray`, `arrayfun`
Parallel Computing

- Parallel computing is a type of computation in which many calculations are carried out simultaneously.

- Speedup of a parallel program,

\[ S(p) = \frac{T(1)}{T(p)} = \frac{1}{\alpha + \frac{1}{p} (1 - \alpha)} \]

- \( p \): number of processors/cores,
- \( \alpha \): fraction of the program that is serial.

Figure from: https://en.wikipedia.org/wiki/Parallel_computing
Two types of Parallel Computers

- Shared memory
- Multiple CPU cores within one node

- Distributed memory
- Multiple nodes within one cluster

- Implicit (multithreaded) parallelism in Matlab is only for multiple cores on one node.
- Explicit parallelism in Matlab can be implemented on either single node or multiple nodes.

Only the single-node mode is supported on BU SCC currently.
Graphic Processing Unit (GPU)

* GPU is a device attached to a CPU-based system.
* Computer program can be parallelized and accelerated on GPU.
* CPU and GPU has separated memory. Data transfer between CPU and GPU is required.
* Many Matlab functions are enabled on GPU.
Implicit parallelism: multithreaded operations

Many built-in operators or functions are implicitly multi-threaded, such as,
Basic: +, -, .*, ./, ^, *, ^, MAX, MIN, SUM, SORT, ABS
Elementary math: ATAN2, COS, CSC, SEC, SIN, TAN, EXP, POW2, SQRT, ABS, LOG, LOG10
Linear algebra: INV, LINSOLVE, \, LU, QR, EIG, SVD
Data analysis: FFT, CONV2

Multithreading may be triggered for vector implementation but not for loop implementation.

By default on BU SCC, these operations automatically use the requested number of CPU cores in a batch job.

Use function maxNumCompThreads(n) to limit the number of cores to be used.
Multithreaded Matrix Multiplication

n=7000;
A=randn(n); B=randn(n); % initialize data
C=zeros(n); D=zeros(n);

tic  % start measuring time
C = A * B;  % multithreaded by default
toc  % end measuring time

maxNumCompThreads(1); % enforce using 1 thread
tic
D = A * B;  % single thread
toc
parpool

- parpool enables the full functionality of the parallel language features (e.g. parfor and spmd) by creating a special job on a pool of workers, and connecting the MATLAB client to the parallel pool.
  - Client/master: runs serial work. Interactive with users (e.g. for input, output, serial parts).
  - Workers/labs: run parallel work. Typically each worker uses one CPU core.

Syntax

parpool(n) % n is the number of workers (a user-defined variable).

% parallel codes

delete(gcp)

- On SCC, the default number of workers equals the number of CPU cores on the node.
- gcp (get current parpool) is a built-in variable.
parfor (1): Basics

◊ A simple implementation of parallel for-loop.
◊ Work load is distributed evenly and automatically based on loop index.
◊ Data starts on client (base workspace), automatically copy input data to workers’ workspaces, and copy output data back to client when necessary calculation is done.
◊ Details are intentionally opaque to user. There are many additional restrictions as to what can and cannot be done in a parfor loop – this is the price of simplicity.

◊ Syntax
parfor i=1:n
    % code block
end

◊ An example:
x=zeros(1,12);
parfor i=1:12
    t = getCurrentTask(); disp(t.ID);  % Disply worker ID
    x(i)=10*i;  % Computation is done by workers simultaneously
end
parfor (2): Rules for variables

- For the `parfor` loop to work, variables inside the loop must all fall into one of these categories:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>A loop index variable for arrays</td>
</tr>
<tr>
<td>Sliced</td>
<td>An array whose segments are manipulated on different loop iterations</td>
</tr>
<tr>
<td>Broadcast</td>
<td>A variable defined before the loop and is used inside the loop but never modified</td>
</tr>
<tr>
<td>Reduction</td>
<td>Accumulates a value across loop iterations, regardless of iteration order</td>
</tr>
<tr>
<td>Temporary</td>
<td>Variable created inside the loop, but not used outside the loop</td>
</tr>
</tbody>
</table>
parfor (3): Modify variables

n=12;  s = 0;
a = 100;  b=50;
X = rand(1,n);
parfor k = 1 : n
    a = 2*k;       % k - loop index;  a - temporary var: the value is not carried out of the loop
    Y(k) = X(k) + a*n;   % X, Y - sliced var;  n - broadcast var
    b = 5;       % b - broadcast var:
    s = s + a;   % s - reduction var: : the value is carried out of the loop
end

◊ Question: what are the values of variables a, b and s after the parfor loop is done?
parfor (4): Reduction

◊ Reduction variables appear on both sides of an assignment statement, such as:
  
  \[
  \text{X} = \text{X op expr}\\
  \text{X} = \text{expr op X} \text{ (except subtraction)}
  \]

✓ The operation \textit{op} could be \(+, -, \ast, \ast, \&\), \mid

◊ A failed case: not a reduction:

\[
\begin{align*}
\text{x} &= 1; \\
\text{parfor i} &= 1:10 \\
&\quad \text{x} = i - \text{x}; \\
\text{end}
\end{align*}
\]

◊ A successful case: a reduction:

\[
\begin{align*}
\text{x} &= 1; \\
\text{parfor i} &= 1:10 \\
&\quad \text{x} = \text{x} - \text{i}; \\
\text{end}
\end{align*}
\]
parfor (5): Data dependency

◊ Data dependency: loop iterations must be independent

• A failed case:

\[
\begin{align*}
 \text{n} &= 10; \\
 \text{a} &= 1: \text{n}; \\
 \text{parfor} \ i &= 2: \text{n} \\
 & \quad \text{a(i) = a(i-1)*2;} \\
 \text{end}
\end{align*}
\]
% This may return unexpected results.

• A successful case:

\[
\begin{align*}
 \text{n} &= 10; \\
 \text{a} &= 1: \text{n}; \\
 \text{parfor} \ i &= 1: \text{n} \\
 & \quad \text{a(i) = a(i)*2;} \\
 \text{end}
\end{align*}
\]
% Each a(i) is read and modified by a worker. Different indexes are independent.
parfor (6): Loop index

- Loop index must be consecutive integers.

parfor i= 1 : 100 % OK
parfor i= -20 : 20 % OK
parfor i= 1 : 2 : 25 % No
parfor i= -7.5 : 7.5 % No

A = [3 7 -2 6 4 -4 9 3 7]; parfor i= find(A > 0) % No
parfor (7): Nested loops and functions

- The body of a `parfor`-loop ……
- can contain `for`-loops, including further nested `for`-loops.
- can not contain another `parfor`-loop.
- can make reference to a regular function but not a nested function.
- can call a function that contains another `parfor`-loop, which runs in parallel only if the outer `parfor`-loop runs serially (e.g. specifying one worker).

Compute the value of Pi

◊ Compute the value of Pi using the integral formula

\[ \int_{0}^{1} \frac{4.0}{1 + x^2} \, dx = \pi \]

◊ The serial code

```plaintext
n=2000000000; dx=1/n; pi=0;
for i=1:n
  x = (i - 0.5) * dx;
  pi = pi + 4./(1+x*x);
end
format long
pi=pi*dx
```
Exercise 1

♦ Compute the value of Pi using parfor

  i) Parallelize the code using parfor. Check whether all variables in the parfor region fall into one of the valid categories.

  ii) Compare the performances of the serial and the parallel codes.
spmd (1): Basics

◊ spmd = Single Program Multiple Data
Explicitly and/or automatically…
✓ divide work and data between workers/labs
✓ communicate between workers/labs

◊ Syntax

% execute on client/master out of spmd region

spmd

% execute on all workers within spmd region

end

% execute on client/master out of spmd region
Get an array chunk on each worker using built-in variables `numlabs` and `labindex`.

```matlab
parpool(4)
spmd
    disp(numlabs);  % numlabs – total number of workers
    disp(labindex);  % labindex – index of workers
    N=24;
    A=1:2:N;
    I = find(A > N*(labindex-1)/numlabs & A <= N*labindex/numlabs)
end
delete(gcp)
```
Pmode: Interactive Parallel Command Window

- Workers receive commands entered in the Parallel Command Window, process them, and send the command output back to the Parallel Command Window.

- Launch pmode

```plaintext
>> pmode start 4  % Request for 4 workers
```

- Execute commands in pmode (at prompt P>>)

```plaintext
P>> x = 2 * labindex
P>> y = numlabs
P>> if labindex == 1
    z = x*10 + y
    end
```
spmd (3): Send and receive data

- labSendReceive(ID_send_to, ID_receive_from, send_data) - Send data to one worker and receive data from another worker.
- Example: circularly shift data between neighbor workers

```
spmd
    DataSent=labindex;
    right = mod(labindex, numlabs) + 1;  % the worker on the right
    left = mod(labindex - 2, numlabs) + 1;  % the worker on the left
    % Send data to the right and receive another data from left
    DataRcv = labSendReceive(right, left, DataSent)
end
```
spmd (4): Broadcast data

- labBroadcast - Broadcast data from one worker to all other workers.

```plaintext
spmd
    source=1;
    if labindex == source
        data=1:12;
    % send data from the source worker to other workers, and save it in shared_data on the source worker.
        shared_data = labBroadcast(source, data)
    else
        % receive data on other workers and save it in share_data
        shared_data = labBroadcast(source)
    end
end
```
spmd (5): Composite variable and distributed array

- Use Composite, distributed out of spmd region

```matlab
a=5; % Create a normal variable on client
b=Composite(); c=Composite(); % Create composite variables b and c on client
A=ones(4,4); A=distributed(A); % Create a matrix A on client and distribute it to workers

spmd
  x = a % Variable a is copied to workers and assigned to x. The local variable x is not accessible from client.
  y = labindex % Variable y is a local variable and is not accessible from client.
  b = labindex; % Composite variable b is modified by workers and is accessible from client
  c = magic(labindex+2); % Composite variable can be a matrix too.
  B = A * 2; % Computation is distributed to workers. The result matrix B is accessible from client.
end

b(:) % Output composite variable on client

c(:) % Output composite variable on client

B % Output distributed matrix on client
```
The **distributed** function can be used for parallel computing without using **spmd**.

```matlab
A=randn(n); B=randn(n);
a=zeros(n); b=zeros(n); c=zeros(n);

parpool(4)
a = distributed(A);  % Distributes A, B. a, b are distributed
b = distributed(B);
tic
c = a * b;  % Run the multiplication in parallel by workers. c is distributed.
toc
delete(gcp)
```
spmd (6): Codistributed matrices

- Use `codistributed` within `spmd` region

```
n=1000; A = rand(n); B = rand(n); % create matrices A and B on client
spmd
  u = codistributed(A, codistributor1d(1)); % distribute A by row
  v = codistributed(B, codistributor1d(2)); % distribute B by column, so that A and B are codistributed.
  w = u * v; % run in parallel by workers; the result w is distributed.
  p = rand(n, codistributor1d(1)); % create distributed matrix p on workers
  q = codistributed.rand(n); % create distributed matrix q on workers; p and q are codistributed
  s = p * q; % run in parallel by workers; the result s is distributed
end
x=3+w % use w directly on client
y=2*s % use s directly on client
```
Exercise 2

Ο Compute the value of Pi (using spmd)

   i) Write a parallel code for computing the value of Pi using spmd.

   ii) Compare the performances of the serial, the parfor parallel and the spmd parallel codes.

   (Hints: Distribute the grid to workers and compute local sum on all workers, then use the function gplus to compute the total sum.)
n=5000000000; dx=1/n; total_sum=Composite(); % total_sum will be modified in and used out of spmd region

tic % start measuring time

spmd % start spmd region

m=n/numlabs; % number of grid points on each worker

length=1/numlabs; % grid length on each worker

startx = (labindex -1)*length; % starting x of the current worker

endx = labindex*length; % ending x of the current worker

x = startx : dx : endx; % the portion of x held by the current worker

local_sum=0; % set 0 before accumulating

local_sum = sum( 4. / (1. + x .* x) ); % compute local sum on the current lab

total_sum = gplus(local_sum, 1); % add up all local sums and store it on lab 1

end % end spmd region

toc % end measuring time

format long

pi=total_sum{1}*dx % get the value of total_sum from worker 1 and output the result on client
Using Matlab on GPU (1)

- For many problems, GPUs achieve better performance than CPUs.
- MATLAB GPU utilities are growing.

- **Matrix operations on GPU:**

  ```matlab
  n = 6400;     % matrix size, better to be multiple of GPU warp-size (i.e. 32).
  a = rand(n);  % create n * n random matrix a on base workspace (host)
  A = gpuArray(a);     % A is created on GPU. The value of a is copied to A.
  B = gpuArray.rand(n); % Create random matrix directly on GPU
  C = A * B;       % Matrix multiplication is computed on GPU
  c = gather(C);    % bring result back to base workspace on CPU/host
  ```
Using Matlab on GPU (2)

◊ Run built-In functions on a GPU.

◊ Examples: Run fast-fourier-transform on GPU:

Ga = rand(1000,'single','gpuArray');
Gfft = fft(Ga);
Gb = (real(Gfft) + Ga) * 6;
G = gather(Gb);
whos
Using Matlab on GPU (3)

- **arrayfun**: Apply function to each element of array on GPU.

```matlab
n=10;
a = rand(n,1,'gpuArray');  % create random arrays on GPU
b = rand(n,1,'gpuArray');
c = rand(n,1,'gpuArray');
R = arrayfun(@(x,y,z)(x.*y+z), a, b, c);  % compute arrayfun on GPU
results = gather(R)  % bring result back to base workspace on CPU/host
```
Using Matlab on GPU (4)

- Use CUDA functions in Matlab.
- A CUDA C function for adding two vectors.

```c
__global__ void add2( double * v1, const double * v2 ) {
    int idx = threadIdx.x;
    v1[idx] += v2[idx];
}
```

- Compile the CUDA code to get an assembly-level ptx file

```
$ module load cuda/9.1
$ nvcc -ptx vecadd.cu
```
Using Matlab on GPU (5)

◊ A Matlab code to call the CUDA function.

```matlab
% Create GPU CUDA kernel object from PTX and CU code
k = parallel.gpu.CUDAKernel('addvec.ptx','addvec.cu','add2');
N = 128;
k.ThreadBlockSize = N; % Array size: better to be multiple of 32.
in1 = ones(N,1,'gpuArray'); % A GPU array with all elements equal to one.
in2 = rand(N,1,'gpuArray'); % A GPU array with random values between 0 and 1.
result = feval(k,in1,in2); % Run the kernel function on GPU and return the result.
disp(result) % Display the result.
```
Further Information

◊ MathWorks Web:

✓ MATLAB Parallel Computing Toolbox documentation:

◊ BU Research Computing Services (RCS) Web:

✓ MATLAB Parallel Computing Toolbox:
http://www.bu.edu/tech/support/research/software-and-programming/common-languages/matlab/pct/

◊ A book: Accelerating MATLAB Performance: 1001 tips to speed up MATLAB programs by Yair M. Altman

◊ RCS help: help@scc.bu.edu, shaohao@bu.edu