MATLAB on BioHPC

[web] portal.biohpc.swmed.edu
[email] biohpc-help@utsouthwestern.edu

Updated for 2015-06-17
What is MATLAB

• High level language and development environment for:
  - Algorithm and application development
  - Data analysis
  - Mathematical modeling

• Extensive math, engineering, and plotting functionality

• Add-on products for image and video processing, communications, signal processing and more.

powerful & easy to learn!
**Ways of MATLAB Parallel Computing**

- **Build-in Multithreading**
  *automatically handled by Matlab
  - use of vector operations in place of for-loops

Example: dot product

\[
C = A \times B 
\]

\[
\text{for } i = 1 : \text{size}(A, 1) \\
\text{for } j = 1 : \text{size}(A, 2) \\
C(i, j) = A(i, j) \times B(i, j) \\
\text{end} \\
\text{end}
\]

If both A and B are 100-by-100 Matrix, the Matlab build-in dot product is 50~100 times faster than for-loop.

* Testing environment : Matlab/R2015a + BioHPC computer node
Ways of MATLAB Parallel Computing

• **Parallel Computing Toolbox**
  *Acceleration MATLAB code with very little code changes
  job will running on your workstation, thin-client or a reserved compute node

  - Parallel for Loop (parfor)
  - Spmd (Single Program Multiple Data)
  - Pmode (Interactive environment for spmd development)
  - Parallel Computing with GPU

• **Matlab Distributed Computing Server (MDCS)**
  *Directly submit matlab job to BioHPC cluster
  - Matlab job scheduler integrated with SLURM
Parallel Computing Toolbox – Parallel for-Loops (parfor)

- Allow several MATLAB workers to execute individual loop iterations simultaneously.
- The only difference in parfor loop is the keyword parfor instead of for. When the loop begins, it opens a parallel pool of MATLAB sessions called workers for executing the iterations in parallel.

\[
\text{for } \ldots \\
\text{<statements>} \\
\text{end}
\]

\[
\text{parfor } \ldots \\
\text{<statements>} \\
\text{end}
\]

\[
\text{<open matlab pool>}
\]

\[
\text{<close matlab pool>}
\]
Example 1: Estimating an Integration

\[ F(x) = \int_{a}^{b} 0.2 \sin(2x) \, dx \]

The integration is estimated by calculating the total area under the \( F(x) \) curve.

\[ F(x) \approx \sum_{1}^{n} \text{height} \times \text{width} \]

We can calculate subareas at any order.
Example 1: Estimating an Integration

**quad_fun.m (serial version)**

```matlab
function q = quad_fun (n, a, b)
    q = 0.0;
    w=(b-a)/n;
    for i = 1 : n
        x = (n-i) * a + (i-1) * b) /(n-1);
        fx = 0.2*sin(2*x);
        q = q + w*fx;
    end
    return
end
```

**quad_fun_parfor.m (parfor version)**

```matlab
function q = quad_fun_parfor (n, a, b)
    q = 0.0;
    w=(b-a)/n;
    parfor i = 1 : n
        x = (n-i) * a + (i-1) * b) /(n-1);
        fx = 0.2*sin(2*x);
        q = q + w*fx;
    end
    return
end
```

```matlab
>> tic
q=quad_fun(120000000, 0.13, 1.53);
t1=toc

 t1 =
 5.7881

>> parpool('local');
Starting parallel pool (parpool) using the 'local' profile ... connected to 12 workers.
>> tic
q=quad_fun_parfor(120000000, 0.13, 1.53);
t2=toc

 t2 =
 0.9429

12 workers
Speedup = t1/t2 = 6.14x
Q: How many workers can I use for MATLAB parallel pool?

<table>
<thead>
<tr>
<th></th>
<th>Default</th>
<th>Maximum</th>
</tr>
</thead>
<tbody>
<tr>
<td>128GB, 384GB, GPU</td>
<td>12</td>
<td>16*</td>
</tr>
<tr>
<td>256GB</td>
<td>12</td>
<td>24*</td>
</tr>
<tr>
<td>Workstation</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>Thin client</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

* matlab_r2013a and matlab_r2013b limits the max number of workers to be 12
** Use `parpool('local', numOfWorkers)` to specify how many workers you want.
  (or use `matlabpool('local', numOfWorkers)` if you are using matlab_r2013a)
**Limitations of parfor**

**No Nested parfor loops**  
*Because a worker cannot open a parallel pool*

```matlab
parfor i = 1 : 10  
    parfor j = 1 : 10  
        <statement>  
    end  
end
```

**Loop variable must be increasing integers**

```matlab
parfor x = 0 : 0.1 : 1  
    <statement>  
end
```

**Loop iterations must be independent**

```matlab
parfor i = 2 : 10  
    A(i) = 0.5* ( A(i-1) + A(i+1) )  
end
```

```matlab
xAll = 0 : 0.1 :1;  
parfor ii = 1 : length(xAll)  
    x = xAll(ii);  
    <statement>  
end
```

And many more ...
spmd: Single Program Multiple Data

- **single program** -- the identical code runs on multiple workers.
- **multiple data** -- each worker can have different, unique data for that code.

- **numlabs** – number of workers.
- **labindex** -- each worker has a unique identifier between 1 and numlabs.
- perform point-to-point communications between workers with **labsend** and **labreceive**
- **Ideal for:**
  1) programs that takes a long time to execute.
  2) programs operating on large data set.

```matlab
spmd
    labdata = load([‘datafile_’ num2str(labindex) ‘.ascii’])
    result = MyFunction(labdata);
end
```
Example 1: Estimating an Integration

**quad_fun.m**

```matlab
function q = quad_fun (n, a, b)
    q = 0.0;
    w=(b-a)/n;
    for i = 1 : n
        fx = 0.2*sin(2*i);
        q = q + w*fx;
    end
    return
end
```

**quad_fun_spmd.m (spmd version)**

```matlab
a = 0.13;
b = 1.53;
n = 120000000;
spmd
    aa = (labindex - 1) / numlabs * (b-a) + a;
    bb = labindex / numlabs * (b-a) + a;
    nn = round(n / numlabs)
    q_part = quad_fun(nn, aa, bb);
end
q = sum([q_part{:}]);
```

12 workers

**Speedup = t1/t3 = 8.26x**

12 communications between client and workers to update q, whereas in parfor, it needs 120,000,000 communications between client and workers
Parallel Computing Toolbox – spmd and Distributed Array

**distributed array** - Data distributed from client and access readily on client. Data always distributed along the last dimension, and as evenly as possible along that dimension among the workers.

**codistributed array** – Data distributed within spmd. Array created directly (locally) on worker.

Matrix Multiply A*B

```matlab
parpool('local');
A = rand(3000);
B = rand(3000);
a = distributed(A);
b = distributed(B);
c = a*b;  % run on workers, c is distributed
delete(gcp);
```
```
parpool('local');
A = rand(3000);
B = rand(3000);
spmd
    u = codistributed(A, codistributor1d(1));    % by row
    v = codistributed(B, codistributor1d(2));    % by column
    w = u * v;
end
delete(gcp);
```
Example 2: $Ax = b$

linearSolver.m (serial version)

```matlab
n = 10000;
M = rand(n);
X = ones(n, 1);
A = M + M';
b = A * X;
u = A \ b;
```

```matlab
t1 = tic;
linearSolver;t1 = toc
```

$t1 = 8.4054$

linearSolverSpmd.m (spmd version)

```matlab
n = 10000;
M = rand(n);
X = ones(n, 1);
spmd
    m = codistributed(M, codistributor('1d', 2));
    x = codistributed(X, codistributor('1d', 1));
    A = m + m';
    b = A * x;
    utmp = A \ b;
end
u1 = gather(utmp);
```

```matlab
>> parpool('local');
Starting parallel pool (parpool) using the 'local' profile ... connected to 12 workers.
>> tic;linearSolverSpmd;t = toc
```

$t = 16.5482$
Factors reduce the speedup of parfor and spmd

- Computation inside the loop is simple.
- Memory limitations. (create more data compare to serial code)
- Transfer data is time consuming
- Unbalanced computational load
- Synchronization
Example 3: Contrast Enhancement

```
function y = contrast_enhance(x)
    x = double(x);
    n = size(x, 1);
    x_average = sum(sum(x(:,:)))/n/n;
    s = 3.0;  % the contrast s should be greater than 1
    y = (1.0 - s) * x_average + s * x((n+1)/2, (n+1)/2);
    return
end
```

```
x = imread('surfsup.tif');
yl = nlfilter(x, [3,3], @contrast_enhance);
y = uint8(y);
```

Running time is 14.46 s

(Example provided by John Burkardt @ FSU)
Example 3: Contrast Enhancement

**contrast_spmd.m (spmd version)**

```matlab
x = imread('surfsup.tif');
xd = distributed(x);
spmd
    xl = getlocalPart(xd);
    xl = nlfilter(xl, [3,3], @contrast_enhance);
end
y = [xl{:}];
```

General sliding-neighborhood operation

12 workers, running time is 2.01 s
speedup = 7.19x

When the image is divided by columns among the workers, artificial internal boundaries are created!

Solution: build up communication between workers.
Example 3: Contrast Enhancement

\[ \text{column} = \text{labSendReceive} \left( \text{next}, \text{previous}, \text{xl}(\cdot, \text{end}) \right); \]

For a 3*3 filtering window, you need 1 column of ghost boundary on each side.

\[ \text{column} = \text{labSendReceive} \left( \text{previous}, \text{next}, \text{xl}(\cdot, 1) \right); \]
Example 3: Contrast Enhancement

contrast_MPI.m (mpi version)

```matlab
x = imread('surfsup.tif');
xd = distributed(x);

spmd
    xl = getLocalPart(xd);
    if (labindex ~= 1)
        previous = labindex - 1;
    else
        previous = numlabs;
    end
    if (labindex ~= numlabs)
        next = labindex + 1;
    else
        next = 1;
    end
end

xl = nlfilter(xl,[3,3],@contrast_enhance);
```

find out previous & next by labindex

12 workers running time 2.00 s speedup = 7.23x

```matlab
column = labSendReceive( previous, next, xl(:,1) );
if (labindex < numlabs)
    xl = [xl, column];
eend

column = labSendReceive( next, previous, xl(:,end) );
if (1 < labindex)
    xl = [column, xl];
eend

xl = uint8(xl);
y = [xl{;}];
```

attache ghost boundaries

remove ghost boundaries
Parallel Computing Toolbox – pmode: learn parallel programming interactively

Command Window

New to MATLAB? See resources for Getting Started.

>> pmode start local 4
Starting pmode using the 'local' profile ... connected to 4 workers.

>> A = magic(5);
ans =
    1

>> A = codistributed(A, codistributor('1d', 1))
This worker stores A(:,1:2).
    LocalPart: [5x5 double]
    Codistributor: [1x1 codistributor1d]

>> bb = codistributed(A, codistributor('1d', 2))
This worker stores bb(:,1:2).
    LocalPart: [5x5 double]
    Codistributor: [1x1 codistributor1d]
Parallel Computing Toolbox – GPU

Parallel Computing Toolbox enables you to program MATLAB to use your computer’s graphics processing unit (GPU) for matrix operations. For some problems, execution in the GPU is faster than in the CPU.

How to enable GPU computing of Matlab on BioHPC cluster

**Step 1**: reserve a GPU node by remoteGPU

**Step 2**: inside terminal, type in

```bash
export CUDA_VISIBLE_DEVICES=“0”
```

to enable the hardware rendering.

* You can launch Matlab directly if you have GPU card on your workstation.
Example 2: $Ax = b$

linearSolverGPU.m (GPU version)

```matlab
n = 10000;
M = rand(n);
X = ones(n, 1);
Mgpu = gpuArray(M);
Xgpu = gpuArray(X);
Agpu = Mgpu + Mgpu';
Bgpu = Agpu * Xgpu;
ugpu = Agpu \ bgpu;
ug = gather(ugpu);
```

Copy data from CPU to GPU (~0.3 s)

Copy data from GPU to CPU (~0.0002 s)

```matlab
>> tic;linearSolverGPU;t3=toc
t3 =
3.3101
```

Speedup = $t_1/t_3 = 2.54x$
Parallel Computing Toolbox – GPU

Check GPU information (inside Matlab)

Q: If the GPU available.
   if (gpuDeviceCount > 0)

Q: How to check memory usage?
   currentGPU = gpuDevice;
   currentGPU.AvailableMemory;

>> gpuDevice
ans =

CUDADevice with properties:

   Name: 'Tesla K40m'
   Index: 1
   ComputeCapability: '3.5'
   SupportsDouble: 1
   DriverVersion: 7
   ToolkitVersion: 6.5000
   MaxThreadsPerBlock: 1024
   MaxShmemPerBlock: 49152
   MaxThreadBlockSize: [1024 1024 64]
   MaxGridSize: [2.1475e+09 65535 65535]
   SIMDWidth: 32
   TotalMemory: 1.2885e+10
   AvailableMemory: 1.1062e+10
   MultiprocessorCount: 15
   ClockRateKHz: 745000
   ComputeMode: 'Default'
   GPUOverlapsTransfers: 1
   KernelExecutionTimeout: 1
   CanMapHostMemory: 1
   DeviceSupported: 1
   DeviceSelected: 1

>> gpuDeviceCount
ans =
   1
Parallel Computing Toolbox – benchmark

Out of memory

The graph shows the time (seconds) taken for different methods to process matrices of various sizes. The x-axis represents the size of the matrix (n-by-n), while the y-axis shows the time in seconds. The methods compared include serial, spmd, and GPU. The graph highlights that as the size of the matrix increases, the time required for processing also increases, reaching a point where it exceeds the memory capacity, indicated by "Out of memory."
Matlab Distributed Computing Server
Matlab Distributed Computing Server: Setup and Test your MATLAB environment

Home -> ENVIRONMENT -> Parallel -> Manage Cluster Profiles

Add -> Import Cluster Profiles from file
And then select profile file from:

/project/apps/apps/MATLAB/profile/nucleus_r2015a.settings
/project/apps/apps/MATLAB/profile/nucleus_r2014b.settings
/project/apps/apps/MATLAB/profile/nucleus_r2014a.settings
/project/apps/apps/MATLAB/profile/nucleus_r2013b.settings
/project/apps/apps/MATLAB/profile/nucleus_r2013a.settings

*based on your matlab version
Matlab Distributed Computing Server: Setup and Test your MATLAB environment

You should have two Cluster Profiles ready for Matlab parallel computing:

“local” – for running job on your workstation, thin client or on any single compute node of BioHPC cluster (use Parallel Computing toolbox)

“nucleus_r<version No.>” – for running job on BioHPC cluster with multiple nodes (use MDCS)
Matlab Distributed Computing Server: Setup SLURM environment

From Matlab command window:

ClusterInfo.setQueueName(‘128GB’); % use 128 GB partition
ClusterInfo.setNNode(2); % request 2 nodes
ClusterInfo.setWallTime(‘1:00:00’); % setup time limit to 1 hour
ClusterInfo setEmailAddress(‘yi.du@utsouthwestern.edu’); % email notification

You could check your settings by:

ClusterInfo.getQueueName();
ClusterInfo.getNNode();
ClusterInfo.getWallTime();
ClusterInfo.getEmailAddress();
Matlab Distributed Computing Server: batch processing

Job submission

For scripts:
job = batch('ascript', 'Pool', 16, 'profile', 'nucleus_r2015a');

For functions:
job = batch(@sfun, 1, {50, 4000}, 'Pool', 16, 'profile', 'nucleus_r2015a');
Matlab Distributed Computing Server: batch processing

Wait for batch job to be finished

- Block session until job has finished: `wait(job, 'finished');`
- Occasionally checking on state of job: `get(job, 'state');`
- Load result after job finished: `results = fetchOutputs(job)` or `load(job);`
- Delete job: `delete(job);`

You can also open job monitor from Home->Parallel->Monitor Jobs
Example 1: Estimating an Integration

quad_submission.m

```matlab
ClusterInfo.setQueueName(‘super’);
ClusterInfo.setNNode(2);
ClusterInfo.setWallTime(‘00:30:00’);

job = batch(@quad_fun, 1, {120000000, 0.13, 1.53}, ‘Pool’, 63, ‘profile’, ‘nucleus_r2015a’);
wait(job, ‘finished’);
Results = fetchOutputs(job);
```

```plaintext
>> quad_submission

additionalSubmitArgs =

--ntasks=64 -p super -t 00:30:00 -N 2 --licenses=mdcs^64
```

```
<table>
<thead>
<tr>
<th>JOBID</th>
<th>PARTITION</th>
<th>NAME</th>
<th>USER</th>
<th>ST</th>
<th>TIME</th>
<th>NODES</th>
<th>NOODELIST (REASON)</th>
</tr>
</thead>
<tbody>
<tr>
<td>50474</td>
<td>GPU</td>
<td>remoteGP</td>
<td>ydu</td>
<td>R</td>
<td>3:12:45</td>
<td>1</td>
<td>Nucleus042</td>
</tr>
<tr>
<td>50473</td>
<td>super</td>
<td>remoteGU</td>
<td>ydu</td>
<td>R</td>
<td>3:12:49</td>
<td>1</td>
<td>Nucleus010</td>
</tr>
<tr>
<td>50823</td>
<td>super</td>
<td>Job2</td>
<td>ydu</td>
<td>R</td>
<td>0:11</td>
<td>2</td>
<td>Nucleus[024-025]</td>
</tr>
</tbody>
</table>
```
Matlab Distributed Computing Server

* Recall the serial job only needs 5s, MDCS is not designed for small jobs

If we submit spmd version code with MDCS

```
job = batch('quad_fun_spmd', 'Pool', 63, 'profile', 'nucleus_r2015a');
```

Running time is 15 s
Example 4: Vessel Extraction

1. Local normalization
   - remove pepper & salt
   - threshold
   - \( \sim 21.3 \text{ seconds} \)

2. Separate vessel groups based on connectivity
   - \( \sim 6.6 \text{ seconds} \)

3. Find the central line of vessel (fast matching method)
   - \( \sim 6.5 \text{ hours} \)

Microscopy image (size: \(2126 \times 5517\))

243 subsets of binary image
Example 4: Vessel Extraction (step 3)

```matlab
parfor i = 1:length(numSubImages)
    % load binary image
    I = im2double(Bwall(:,:,i));
    % find vessel by using fast matching method
    allS{i} = skeleton(I);
end
```

The time needed for processing each subimage varies. The job running on 4 nodes requires a minimum computational time of 784 seconds. The speedup is 28.3x.
For assistance with MATLAB, please contact the BioHPC Group:

- Email: biohpc-help@utsouthwestern.edu
- Submit help ticket at http://portal.biohpc.swmed.edu