Parallel Computing with MATLAB on Discovery Cluster

Northeastern University Research Computing: Nilay K Roy, MS Computer Science, Ph.D Computational Physics
Lets look at the Discovery Cluster

• Matlab environment
• Setting queues and no of cores
• Running matlab interactively
• Running matlab in batch
Long computations

- Multiple independent iterations

```matlab
parfor i = 1 : n
    % do something with i
end
```

- Series of tasks

Task 1  Task 2  Task 3  Task 4

Large data problems
Parallel Computing Toolbox API

- Task-parallel Applications
  - Using the `parfor` constructs
  - Using jobs and tasks

- Data-parallel Applications
  - Using `distributed arrays`
  - Using the `spmd` construct
Example: Parameter Sweep of ODEs

- Solve a 2nd order ODE

\[ m \ddot{x} + b \dot{x} + k x = 0 \]

- Simulate with different values for \( b \) and \( k \)

- Records and plots peak values
Converting for to parfor

- Requirements for `parfor` loops
  - Task independent
  - Order independent

- Constraints on the loop body
  - Cannot “introduce” variables (e.g. `eval`, `load`, `global`, etc.)
  - Cannot contain `break` or `return` statements
  - Cannot contain another `parfor` loop

- `parfor` automatically quits on error
- `parfor` doesn’t provide intermediate results
Use M-Lint to diagnose `parfor` issues

If your `for` loop cannot be converted to a `parfor`, consider wrapping a subset of the body to a function

Rather than submitting a single task containing a `parfor`, the jobscript can be used to create an array of tasks, each calling a unit of work

```bash
[nilay.roy@discovery2 matlab_seminar_test]$ which mlint
/shared/apps/matlab/matlab-2013b/INSTALL/bin/glnxa64/mlint
[nilay.roy@discovery2 matlab_seminar_test]$ mlint par_parallel.m
L 29 (C 2-5): The value assigned to variable 'mypi' might be unused.
L 30 (C 2-6): The value assigned to variable 'tttime' might be unused.
[nilay.roy@discovery2 matlab_seminar_test]$ mlint spmd_parallel.m
L 53 (C 3-6): The value assigned to variable 'mypi' might be unused.
L 54 (C 3-7): The value assigned to variable 'totpi' might be unused.
L 78 (C 30): Extra semicolon is unnecessary.
[nilay.roy@discovery2 matlab_seminar_test]$ module list
Currently Loaded Modulefiles:
  1) gnu-4.4-compilers  2) fftw-3.3.3  3) platform-mpi  4) oracle_java_1.7u40  5) matlab_dce_2013b  6) lammps_ind_dip_pol
[nilay.roy@discovery2 matlab_seminar_test]$
```
parfor or jobs and tasks

parfor

- Seamless integration to user’s code
- Several for loops throughout the code to convert
- Automatic load balancing

Jobs and tasks

- All tasks run
- Query results after each task is finished

Try parfor first. If it doesn’t apply to your application, create jobs and tasks.
Data-parallel Applications

- Using distributed arrays
- Using `spmd`
- Using mpi based functionality

Client-side Distributed Arrays and SPMD

- Client-side distributed arrays
  - Class `distributed`
  - Can be created and manipulated directly from the client.
  - Simpler access to memory on labs
  - Client-side visualization capabilities

- `spmd`
  - Block of code executed on workers
  - Worker specific commands
  - Explicit communication between workers
  - Mixture of parallel and serial code
spmd blocks (Data Parallel)

```matlab
spmd
    % single program across workers
end
```

- Mix data-parallel and serial code in the same function
- Run on a pool of MATLAB resources
- **Single Program** runs simultaneously across workers
  - Distributed arrays, message-passing
- **Multiple Data** spread across multiple workers
  - Data stays on workers
Composite Array in Memory

```matlab
>> matlabpool open 4
>> x = Composite(4)
>> x{1} = 2
>> x{2} = [2, 3, 5]
>> x{3} = @sin
>> x{4} = tobject()
```
- single program, multiple data
- Unlike variables used in multiple `parfor` loops, distributed arrays used in multiple `spmd` blocks retain state
- Use M-Lint to diagnose `spmd` issues
MPI-Based Functions in Parallel Computing Toolbox

Use when a high degree of control over parallel algorithm is required

- High-level abstractions of MPI functions
  - `labSendReceive`, `labBroadcast`, and others
  - Send, receive, and broadcast any data type in MATLAB

- Automatic bookkeeping
  - Setup: communication, ranks, etc.
  - Error detection: deadlocks and miscommunications

- Pluggable
  - Use any MPI implementation that is binary-compatible with MPICH2
Summary for Interactive Functionality

- **Client-side Distributed Arrays**
  - MATLAB array type across cluster
  - Accessible from client

- **SPMD … END**
  - Flow control from serial to parallel
  - Fine Grained
  - More control over distributed arrays

- **Composite Arrays**
  - Generic data container across cluster
  - Accessible from client
# Summary for Scheduled Functionality

<table>
<thead>
<tr>
<th></th>
<th>uses matlabpool</th>
<th>function</th>
<th>script</th>
<th>pure task parallel</th>
<th>pure data parallel</th>
<th>parallel and serial</th>
</tr>
</thead>
<tbody>
<tr>
<td>batch</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>matlabpool job</td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>jobs and tasks</td>
<td></td>
<td>✓</td>
<td>✓</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>parallel job</td>
<td></td>
<td>✓</td>
<td></td>
<td></td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>
- Profile your code to search for bottlenecks
- Make use of M-Lint when coding `parfor` and `spmd`
- Beware of writing to files
- Avoid the use of global variables
- Run locally before moving to cluster
PARFOR

tic
s = 100000000000;
parfor i = 1:s
s = s + 1;
end
toc

tic
s = 100000000000;
for i = 1:s
s = s + 1;
end
toc
Now for some computation and profiling runs

- PARPOOL (replaced by MATLABPOOL command in R2013b) example batch run
- SPMD example batch run
First a profiler run – lets do this on cluster

pmode start local 4

• Parallel Command Window (pmode) starts

P>> R1 = rand(16, codistributor())

P>> R2 = rand(16, codistributor())

P>> mpi_PROFILE on

P>> P = R1*R2

P>> mpi_PROFILE off

• P>> mpi_PROFILE viewer
GPU RUNS with Matlab

Will Execution on a GPU Accelerate My Application?

A GPU can accelerate an application if it fits both of the following criteria:

- Computationally intensive—The time spent on computation significantly exceeds the time spent on transferring data to and from GPU memory.
- Massively parallel—The computations can be broken down into hundreds or thousands of independent units of work.

Applications that do not satisfy these criteria might actually run slower on a GPU than on a CPU.
Over 100 operations (e.g. fft, ifft, eig) are now available as built-in MATLAB functions that can be executed directly on the GPU by providing an input argument of the type GPUArray.

GPU-enabled functions are overloaded.
Invoke built-in MATLAB functions on the GPU

1. Minimal effort, minimal level of control

- Define an array on the GPU
  ```
  A = rand(1000,1);
  B = rand(1000,1);
  A_gpu = gpuArray(A);
  B_gpu = gpuArray(B);
  ```

- Execute a built-in MATLAB function:
  ```
  Y_gpu = B_gpu \ A_gpu;
  ```

- Retrieve data from the GPU
  ```
  result = gather(Y_gpu);
  ```
Run MATLAB functions on the GPU
(2) Straightforward effort, regular level of control

- MATLAB function that perform element-wise arithmetic

  ```matlab
  function y = TaylorFun(x)
  y = 1 + x*(1 + x*(1 + x*(1 + ... 
     x*(1 + x*(1 + x*(1 + x*(1 + ... 
     x*(1 + ./9)./8)./7)./6)./5)./4)./3)./2);
  ```

- Load data on the GPU

  ```matlab
  A = rand(1000,1);
  A_gpu = gpuArray(A);
  ```

- Execute the function as GPU kernels

  ```matlab
  result = arrayfun(@TaylorFun, A_gpu);
  ```
Invoke CUDA Code from MATLAB

(3) Involved effort, extensive level of control

- Compile CUDA (or PTX) code on the GPU
  ```
  nvcc -ptx myconv.cu
  ```

- Construct the kernel
  ```
  k = parallel.gpu.CUDAKernel('myconv.ptx', 'myconv.cu');
  k.GridSize = [512 512];
  k.ThreadBlockSize = [32 32];
  ```

- Run the kernel using the MATLAB workspace
  ```
  o = feval(k, rand(100, 1), rand(100, 1));
  ```
  or gpu data
  ```
  ilgpu = gpuArray(rand(100, 1, 'single'));
  i2gpu = gpuArray(rand(100, 1, 'single'));
  ogpu = feval(k, ilgpu, i2gpu);
  ```
Command Window

```
100000
D = gpuArray.randn(N);
[L,U,p] = lu(D,'vector');
norm(L'*U-D(p,:),1)
```

See also `lu`, `gpuArray.randn`

```
>> maxIterations = 500;
gridSize = 1000;
xLin = [-0.74826671322161, -0.74826570771757];
yLin = [ 0.123640544894862, 0.1236405481045266];
>> % Setup
t = tic();
x = linspace(xLin(1), xLin(2), gridSize);
y = linspace(yLin(1), yLin(2), gridSize);
[xGrid,yGrid] = meshgrid(x, y);
z0 = xGrid + 1i*yGrid;
count = ones(size(z0));

% Calculate
z = z0;
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
    count = count + inside;
end

cpuTime = toc(t);
fig = gcf;
fig.Position = [200 200 600 600];
imagesc(x, y, count);
axis image
colormap([jet();flipud(jet());0 0 0]);
title(sprintf('%0.2fsecs (without GPU)', cpuTime));
```

Warning: Struct field assignment overwrites a variable with class "double". See MATLAB R14SP2 Release Notes, Assigning Nonstructure Variables As Structures Displays Warning. For details.
```matlab
% Initialize
z = 0;

% Calculate
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
    count = count + inside;
end

% Show
plotTime = tic();
fig = gcf;
fig.Position = [200 200 600 600];
imagesc(x, y, count);
axis image
colorbar([jet(1);flipud(jet(1));0 0 0]);
title(sprintf('%.3fsecs (without GPU)', plotTime));

% Warning: Scrool field assignment overwrites a value with class "double". See MATLAB documentation for more details.

% Setup
T = tic();
x = gpuArray.linspace(xlim(1), xlim(2), gridSize);
y = gpuArray.linspace(ylim(1), ylim(2), gridSize);
[xGrid, yGrid] = meshgrid(x, y);

z0 = complex(xGrid, yGrid);

% Calculate
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
    count = count + inside;
end

% Show
count = gather(count); % Fetch the data back from the GPU
naivePUTime = toc();
imagesc(x, y, count);
axis image
title(sprintf('%.3fsecs (naive GPU) - 1.6x faster', naivePUTime));

% Calculate
z = z0;

% Calculate
for n = 0:maxIterations
    z = z.*z + z0;
    inside = abs(z) <= 2;
    count = count + inside;
end

% Show
count = gather(count); % Fetch the data back from the GPU
naivePUTime = toc();
imagesc(x, y, count);
axis image
title(sprintf('%.3fsecs (naive GPU) = 1.6x Faster', naivePUTime));
```

**Figure 1 (on compute-2-140)**

![Image of the Mandelbrot set with GPU acceleration](image_url)
- Elementwise operation
- For GPU array inputs, the function used with arrayfun gets compiled into native GPU code
- Using arrayfun means that instead of many thousands of calls to separate GPU-optimized operations (at least 6 per iteration), we make one call to a parallelized GPU operation that performs the whole calculation
CUDA and Matlab

• CUDA/C++ implementation of the element processing algorithm needs to be handwritten
• manually compiled using nVidia's NVCC compiler to produce the assembly-level ptx file (.ptx stands for "Parallel Thread eXecution language")
• One GPU thread is required for location in the Mandelbrot Set, with the threads grouped into blocks
• The kernel indicates how big a thread-block is, and in the code below we use this to calculate the number of thread-blocks required
• This then becomes the GridSize
• We will look at this shortly
• Lets look at a simple handwritten cuda two vector addition

Command Window

```matlab
>> k = parallel.gpu.CUDAKernel('two_vectors.ptx','two_vectors.cu','add2');
>> N=128;
>> k.ThreadBlockSize = N;
>> in1 = ones(N,1,'gpuArray');
>> in2 = ones(N,1,'gpuArray');
>> result = feval(k,in1,in2);
>> result
result =
    2
    2
    2
```
global__ void add( double * v1, const double * v2 )
{
    int idx = threadIdx.x;
    v1[idx] += v2[idx];
}

ptxarch = 'sm_30';
kern = 'mandelbrotViewerProcessElement';

flags = sprintf( '-arch=%s', ptxarch );
if ismac
    % On Mac we must force the use of 64-bit pointers otherwise the host
    % uses 64-bit and the device 32!
    flags = [flags, ' -m 64'];
end

cmd = sprintf( 'nvcc -ptx %s.cu %s -o %s.ptx', ...
    kernel, flags, kernel, parallel.gpu.ptxext );
fprintf('Running command: %s\n', cmd);
result = system( cmd );
if (result ~= 0)
    fprintf( 'Failed with error %d.\n', result )
end

[nilay.roy@compute-2-140 matlab_seminar_test]$
maxIterations = 500;
gridSize = 1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [0.123640844894862, 0.123640851045266];

cpuTime = toc(t);
imagesc(x, y, count)
axis image
colormap([jet(); flipud(jet()); 0 0 0]);
title(sprintf('%1.2fsecs (without GPU)', cpuTime));

% When MATLAB encounters data on the GPU, calculations with that data are
% performed on the GPU. The class
% <matlab:doc('gpuArray') | gpuArray|> provides
% GPU versions of many functions that you can use to create data arrays,
% including the
% <matlab:doc('gpuArray.linspace') | linspace|>,
% <matlab:doc('gpuArray.logspace') | logspace|>, and
% <matlab:doc('gpuArray.meshgrid') | meshgrid|> functions
% needed here. Similarly, the |count| array is initialized directly on the
% GPU using the function
% <matlab:doc('gpuArray.ones') | gpuArray.ones|>.
% % With these changes to the data initialization the calculations will now
% % be performed on the GPU:
%
% Using `arrayfun` means that instead of many thousands of calls to
% separate GPU-optimized operations (at least 6 per iteration), we make one
% call to a parallelized GPU operation that performs the whole calculation.
% This significantly reduces overhead.

% Setup
\[
t = \text{tic}();
\]
\[
x = \text{gpuArray.linspace}( \text{xlim}(1), \text{xlim}(2), \text{gridSize} );
\]
\[
y = \text{gpuArray.linspace}( \text{ylim}(1), \text{ylim}(2), \text{gridSize} );
\]
\[
[xGrid, yGrid] = \text{meshgrid}(x, y);
\]

% Calculate
\[
count = \text{arrayfun}( \text{@pctdemo\_processMandelbrotElement}, \ldots
     xGrid, yGrid, \text{maxIterations} );
\]

% Show
\[
count = \text{gather}( \text{count} ); \% Fetch the data back from the GPU
\]
\[
\text{gpuArrayfunTime} = \text{toc}(t);
\]
\[
\text{imagesc}(x, y, count)
\]
\[
\text{axis image}
\]
\[
\text{title}(\text{sprintf}(\%1.3f\text{secs (GPU arrayfun)} = \%1.1fx faster', \ldots
     \text{gpuArrayfunTime}, \text{cpuTime/gpuArrayfunTime} ));
\]

%% Working with CUDA
% <matlab:doc('parallel.gpu.CUDAKernel') | parallel.gpu.CUDAKernel|>
% A CUDA/C++
% in <matlab:edit('pctdemo\_processMandelbrotElement.cu')>
% pctdemo\_processMandelbrotElement.cu |
% pctdemo\_processMandelbrotElement.ptx | (.ptx| stands for "Parallel
% Thread eXecution language")
% _device_
% __device__
% unsigned int doIterations( double const realPart0,
%   double const imagPart0,
%   unsigned int const maxIters ) {
%   // Initialize: z = z0
%   double realPart = realPart0;
%   double imagPart = imagPart0;
%   unsigned int count = 0;
%   // Loop until escape
%   while ( ( count <= maxIters )
%       && ((realPart*realPart + imagPart*imagPart) <= 4.0) ) {
%     ++count;
%     // Update: z = z*z + z0;
%     double const oldRealPart = realPart;
%     realPart = realPart*realPart - imagPart*imagPart + realPart0;
%     imagPart = 2.0*oldRealPart*imagPart + imagPart0;
%   }
%   return count;
% }
% One GPU thread is required for location in the Mandelbrot Set, with the
% threads grouped into blocks. The kernel indicates how big a thread-block
% is, and in the code below we use this to calculate the number of
% thread-blocks required. This then becomes the [ GridSize ].

% Load the kernel
\[
cudaFilename = 'pctdemo\_processMandelbrotElement.cu';
\]
\[
ptxFilename = ['pctdemo\_processMandelbrotElement.', parallel.gpu.ptxext];
\]
\[
kernel = parallel.gpu.CUDAKernel( ptxFilename, cudaFilename );
\]

% Setup
\[
t = \text{tic}();
\]
\[
x = \text{gpuArray.linspace}( \text{xlim}(1), \text{xlim}(2), \text{gridSize} );
\]
\[
y = \text{gpuArray.linspace}( \text{ylim}(1), \text{ylim}(2), \text{gridSize} );
\]
\[
[xGrid, yGrid] = \text{meshgrid}(x, y);
\]

% Make sure we have sufficient blocks to cover all of the locations
\[
\text{numElements} = \text{numel}(xGrid);
\]
\[
kernel.ThreadBlockSize = [kernel.MaxThreadsPerBlock, 1, 1];
\]
\[
kernel.GridSize = [\text{ceil(numElements/kernel.MaxThreadsPerBlock)}, 1];
\]

% Call the kernel
\[
count = \text{gpuArray.zeros}(\text{size(xGrid)});
\]
\[
count = \text{feval}(\text{kernel}, \text{count}, \text{xGrid}, \text{yGrid}, \text{maxIterations}, \text{numElements});
\]

% Show
\[
count = \text{gather}(\text{count}); \% Fetch the data back from the GPU
\]
\[
\text{gpuCUDAKernelTime} = \text{toc}(t);
\]
\[
\text{imagesc}(x, y, count)
\]
\[
\text{axis image}
\]
\[
\text{title}(\text{sprintf}(\%1.3f\text{secs (GPU CUDA Kernel)} = \%1.1fx faster', \ldots
     \text{gpuCUDAKernelTime}, \text{cpuTime/gpuCUDAKernelTime} ));
\]
• mapreduce reads a chunk of data from the input datastore \( \text{[data,info]} = \text{read(ds)} \) -> calls the map function to work on chunk
• map function receives the chunk of data, organizes it or performs a precursory calculation ->then uses the add and addmulti functions to add key-value pairs to an intermediate data storage object called a KeyValueStore - The number of calls to the map function by mapreduce is equal to the number of chunks in the input datastore
• after the map function works on all of the chunks of data in the datastore, mapreduce groups all of the values in the intermediate KeyValueStore object by unique key
• mapreduce calls the reduce function once for each unique key added by the map function -> each unique key can have many associated values
• mapreduce passes the values to the reduce function as a ValueIterator object (used to iterate over the values)
• ValueIterator object for each unique key contains all the associated values for that key
• reduce function uses the hasnext and getnext functions to iterate through the values in the ValueIterator object one at a time
• after aggregating the intermediate results from the map function, the reduce function adds final key-value pairs to the output using the add and addmulti functions
• order of the keys in the output is the same as the order in which the reduce function adds them to the final KeyValueStore object - mapreduce does not explicitly sort the output.
Note: reduce function writes the final key-value pairs to a final KeyValueStore object - from this object, mapreduce pulls the key-value pairs into the output datastore, which is a KeyValueDatastore object by default
- Matlab mapreduce can also be run on HADOOP hdfs file system
- M2M Translator – Matlab has their own but one can create ones own
- Coding in Java – Beyond scope of this talk.
Questions?

Thank you.