Workshop: Parallel Computing with MATLAB

Eric Johnson  Application Engineer
Konrad Malkowski  Application Support Engineer
Outline

- Introduction to Parallel Computing Tools
- Using Parallel Computing Toolbox
  - Task Parallel Applications
  - Data Parallel Applications
Parallel Computing with MATLAB
Parallel Computing with MATLAB

User’s Desktop

Parallel Computing Toolbox

MATLAB Workers

MATLAB Distributed Computing Server

Compute Cluster
Solving Big Technical Problems

Challenges

Long running

Computationally intensive

Large data set

You could...

Wait

Run similar tasks on independent processors in parallel

Reduce size of problem

Load data onto multiple machines that work together in parallel
Parallel Computing Toolbox API

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

- Data-parallel Applications
  - Using `distributed` arrays
  - Using the `spmd` construct
Task-parallel Applications

- Converting `for` to `parfor`
- Configurations
- Scheduling `parfor`
- Creating jobs and tasks
- When to Use `parfor` vs. jobs and tasks
- Resolving `parfor` Issues
- Resolving jobs and tasks Issues
Toolboxes with Built-in Support

- Optimization Toolbox
- Global Optimization Toolbox
- Statistics Toolbox
- Simulink Design Optimization
- Bioinformatics Toolbox
- Communications Toolbox
- ....

Contain functions that directly leverage functions from the Parallel Computing Toolbox
Opening and Closing a matlabpool...

Open and close a matlabpool with two labs
Determining the Size of the Pool...
One Pool at a Time

Even if you have not exceeded the number of labs, you can only open one matlabpool at a time
Add Shortcut for Starting the matlabpool

```matlab
if matlabpool('size') == 0
    matlabpool open local 2
end
```
Add Shortcut for Stopping the matlabpool

```
if matlabpool('size')>0
    matlabpool close
end
```
Example: Parameter Sweep of ODEs

- Solve a 2\textsuperscript{nd} order ODE

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

1,2,... 1,2,...

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

- Records and plots peak values

\texttt{\textbackslash task\_parallel\textbackslash paramSweepScript.m}
The Mechanics of `parfor` Loops

```matlab
a = zeros(10, 1);
parfor i = 1:10
    a(i) = i;
end
a
```

Pool of MATLAB Workers
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
Advice for Converting for to parfor

- 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

- Read the section in the documentation on classification of variables

Resolving `parfor` Issues

- Let’s look at a common `parfor` issues and how to go resolving them
Unclassified Variables

The variable A cannot be properly classified

`>> parfor_bug
??? Error: The variable a in a parfor cannot be classified.
    See Parallel for Loops in MATLAB, "Overview".

Error in ==> parfor_bug at 4
parfor idx = 1:N

fx >>`
parfor Variable Classification

- All variables referenced at the top level of the `parfor` must be resolved and classified

<table>
<thead>
<tr>
<th>Classification</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop</td>
<td>Serves as a loop index for arrays</td>
</tr>
<tr>
<td>Sliced</td>
<td>An array whose segments are operated on by different iterations of the loop</td>
</tr>
<tr>
<td>Broadcast</td>
<td>A variable defined before the loop whose value is used inside the loop, but never assigned inside the loop</td>
</tr>
<tr>
<td>Reduction</td>
<td>Accumulates a value across iterations of the loop, regardless of iteration order</td>
</tr>
<tr>
<td>Temporary</td>
<td>Variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop</td>
</tr>
</tbody>
</table>
Variable Classification Example

```
1  a = 0;
2  c = pi;
3  z = 0;
4  r = rand(1,10);
5  parfor idx = 1:10
6       a = idx;
7       z = z+idx;
8       b(idx) = r(idx);
9           if idx<=c
10              d = 2*a;
11          end
12  end
```

Temporary variable
Reduction variable
Sliced output variable
Loop variable
Sliced input variable
Broadcast variable
At the end of this loop, what is the value of each variable?
Results

a: ones(1:10) (broadcast)
b: undefined (temp)
c: undefined (temp)
d: 1:10 (sliced)
e: 55 (reduction)
f: 5 (temp)
g: 20 (reduction)
h: 10 (temp)
idx: undefined (loop)
parfor issue: Nested for loops

Within the list of indices for a sliced variable, one of these indices is of the form \( i, i+k, i-k, k+i, \) or \( k-i, \) where \( i \) is the loop variable and \( k \) is a constant or a simple (non-indexed) variable; and every other index is a constant, a simple variable, colon, or end.

```
A = zeros(10);
parfor i = 1:10
    for j = 1:10
        A(i,j) = i+j;
    end
end
```
parfor issue: Solution 1

Create a temporary variable, b to store the row vector. Use the looping index, i, to index the columns and the colon to assign the row vector to the temporary variable between the for loops.

task_parallel\valid_indexing_fix1.m
Use cell arrays. The restrictions on indexing only apply to the top-level indexing (i.e. indexing into the cell array). Indexing into contents of the cell arrays is allowed.
Using parfor with Simulink

- Can use `parfor` with `sim`.
- Must make sure that the Simulink workspace contains the variables you want to use.

- Within main `parfor` body: Use `‘base’` workspace
- Use `assignin` to place variables in base workspace.
- Note: the base workspace when using `parfor` is different than the base workspace when running serially.

`task_parallel\simParforEx1.m`
Parallel Computing Tools Address...

**Task-Parallel**

- Long computations
  - Multiple independent iterations
    ```
    parfor i = 1 : n
    % do something with i
    end
    ```
  - Series of tasks

**Data-Parallel**

- Large data problems
Data-parallel Applications

- Using distributed arrays
- Using `spmd`
- Using mpi based functionality
Client-side Distributed Arrays

Remotely Manipulate Array from Desktop

Distributed Array Lives on the Cluster

data_parallel\distributed_example.m
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

`data_parallel\spmd_example.m`
The Mechanics of `spmd` Blocks

```
x = 1
spmd
    y = x + 1
end
y
```

Pool of MATLAB Workers
Composite Arrays

- Created from client
- Stored on workers
- Syntax similar to cell arrays
Composite Array in Memory

>> matlabpool open 4

>> x = Composite(4)

>> x{1} = 2
>> x{2} = [2, 3, 5]
>> x{3} = @sin
>> x{4} = tsobject()
spmd

- 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
Noisy Image – too large for a desktop
Distribute Data
Distribute Data
Pass Overlap Data
Pass Overlap Data
Pass Overlap Data
Apply Median Filter
Combine as Distributed Data
Combine as Distributed Data
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

`data_parallel\mpi_example.m`
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
Migrating from Interactive to Scheduled

MATLAB®
SIMULINK®
TOOLBOXES
BLOCKSETS

Scheduler

Worker
Worker
Worker
Worker

Work

Result
Interactive to Scheduled

- Interactive
  - Great for prototyping
  - Immediate access to MATLAB workers

- Scheduled
  - Offloads work to other MATLAB workers (local or on a cluster)
  - Access to more computing resources for improved performance
  - Frees up local MATLAB session
Using Configurations

- **Managing configurations**
  - Typically created by Sys Admins
  - Label configurations based on the version of MATLAB
    - E.g. *linux_r2009a*

- **Import configurations generated by the Sys Admin**
  - Don’t modify them with two exceptions
    - Setting the `CaptureCommandWindowOutput` to true for debugging
    - Set the `ClusterSize` for the local scheduler to the number of cores
Creating and Submitting Jobs

```matlab
sched = findResource();
job = createJob(sched);

task = createTask(job, @rand, 1, {});

submit(job)

% waitForState(job, 'finished')
%
% if ~isempty(task.ErrorMessage)
% error(task.ErrorMessage)
% end
%
% y = getAllOutputArguments(job);
% s = y{1}.^2;
% display(s)
%
% destroy(job)
```

Rather than using a shell script to submit a job to a cluster, we’ll write our `jobscript` in MATLAB.

task_parallel/basic_jobs ascertain.m
Example: Scheduling the ODE Sweep

```matlab
*** Get handle to the job scheduler
sched = findResource();

*** Create a matlabpool job
% Split among pool of 2 labs, 1 lab acts as serial MATLAB does (total = 3)
nlabs = 3;
job = createMatlabPoolJob(sched,...
    'FileDependencies',    {'paramSweep.m'}, ...
    'MinimumNumberOfWorkers', nlabs, ...
    'MaximumNumberOfWorkers', nlabs);
set(job,'Tag','MyODEJob')  % Can use Tag to label jobs, not necessary

%% Create a single parfor task
task = createTask(job,@paramSweep,1,{});

%% Submit the job and wait
submit(job)
```

`task_parallel\jobs\script_ode.m`
Example: Retrieving Results

```matlab
%% Find your submitted job
job.state  % only when job is finished can you load results

%% How to find your job
clear job  % What happens if you lose your job variable
job = findJob(sched,'Tag','MyODEJob');  % Can search by other properties
job.State

%% Once job is finished, get outputs
if ~isempty(task.ErrorMessage)
    % If errors don't get output and display error
    disp(job.task.ErrorMessage)
    output = [];
else
    % No errors, get output
    output = getAllOutputArguments(job);
    celldisp(output)
end

%% When finished, destroy job
destroy(job)
```

task_parallel\ode_return.m
Considerations When Using `parfor`

- `parfor` automatically quits on error
- `parfor` doesn’t provide intermediate results
Creating Jobs and Tasks

- 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.
Example: Using Multiple Tasks

%% Submitting Jobs and Tasks

%% Get handle to the job scheduler
sched = findResource();

%% Create a simple distributed job
job = createJob(sched, 'FileDependencies', {'myfcn.m'});

%% Create the tasks
for tidx = 1:10
    tasks(tidx) = createTask(job,@myfcn, 1, {tidx}); %#ok<SAGROW>
end

submit(job)
Example: Retrieving Task Results

%%% Retrieving Output from Jobs and Tasks

%%% Check to make sure job is finished

job.State

%%% Once finished, get output (method 1, all at once)

output = getAllOutputArguments(job);
celldisp(output)

%%% Once finished, get output (method 2, output of specific task)

job.task(1).State % how to check state of particular task
output = job.task(1).OutputArguments; % output from task 1
celldisp(output)

%%% Destroy job when finished

destroy(job)
Resolving Jobs & Tasks Issues

- Code running on your client machine ought to be able to resolve functions on your path.

- When submitting jobs to a cluster, those files need to either be submitted as part of the job (FileDependencies) or the folder needs to be accessible (PathDependencies).

- There is overhead when adding too many files to the job; but setting path dependencies requires the Worker to be able to reach the path.
**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.
What is the probability that a randomly dropped needle will cross a grid line?

(Buffon-Laplace Method) Simulate random needles dropping, calculate $P$, and get an estimate for $\pi$.

$$P(l, a, b) = \frac{2l(a + b) - l^2}{\pi ab} = \frac{\text{crossing needles}}{\text{total needles}}$$

(data_parallel\jobs\script_Pi.m)
## 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>
Recommendations

- 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