Getting Started with Parallel Computing using MATLAB: Interactive and Scheduled Applications

Created by S. Zaranek, E. Johnson and A. Chakravarti

1. Objectives

This user guide provides an end user with instructions on how to get started running parallel MATLAB applications using a desktop computer or a cluster.

2. Assumptions

- User has access to **MATLAB** and **Parallel Computing Toolbox** on the desktop computer or head node of the cluster.

If running on a cluster:

- **MATLAB Distributed Computing Server** has been installed by an administrator on the cluster.
- The desktop MATLAB client has been configured to connect to the cluster. If this has not been done, you should contact the cluster administrator.

3. Getting the Example Files

Unzip the demoFiles.zip file that was provided along with this guide. You can add the files to the MATLAB path by running the **addpath** command in MATLAB.

```
>> addpath <location of files>
```

4. Examples Running Locally

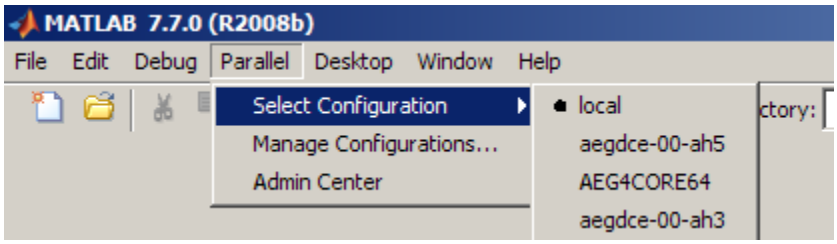
In this section, you will be running and submitting jobs using the local configuration.

If your workflow will ultimately involve submitting jobs to a cluster, you can follow this section by switching the default configuration from local to that of your cluster and running these jobs again. This is described in Section 5.

You can set the configuration to `local`, either at the command-line

```
>> defaultParallelConfig('local')
```

or by using the user interface found in the parallel menu. See screen snapshot below.



For more information on configurations and programming with user configurations, see:

http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/f5-16141.html#f5-16540

1. Using an Interactive MATLAB pool

To interactively run your parallel code, you first need to open a MATLAB pool. This reserves a collection of MATLAB worker sessions to run your code. The MATLAB pool can consist of MATLAB sessions running on your local machine or on a remote cluster. In this case, we are initially running on your local machine.

You can use `matlabpool open` to start an interactive worker pool. If the number of workers is not defined, the default number defined in your configuration will be used. A good rule of thumb is to not open more workers then cores available. If the `Configuration` argument is not provided, `matlabpool` will use the default configuration as setup in the beginning of this section. When you are finished running with your MATLAB pool, you can close it using `matlabpool close`.

Two of the main parallel constructs that can be run on a MATLAB pool are `parfor` loops (parallel for-loops) and `spmd` blocks (single program - multiple data blocks). Both constructs allow for a straight-forward mixture of serial and parallel code.

`parfor` loops are used for task-parallel (i.e. embarrassingly parallel) applications. `parfor` is used to speed up your code. Below is a simple `for` loop converted into a `parfor` to run in parallel, with different iterations of the loop running on

different workers. The code outside the `parfor` loop executes as traditional MATLAB code (serially, in your client MATLAB session).

Note: The example below is located in the m-file, 'parforExample1.m'.

```
matlabpool open 2 % can adjust according to your resources

N = 100;
M = 200;
a = zeros(N,1);

tic;   %  serial (regular) for-loop
for i = 1:N
    a(i) = a(i) + max(eig(rand(M)));
end
toc;

tic;   %  parallel for-loop
parfor i = 1:N
    a(i) = a(i) + max(eig(rand(M)));
end
toc;

matlabpool close
```

`spmd` blocks are a single program multiple data (SPMD) language construct. The "single program" aspect of `spmd` means that the identical code runs on multiple labs. The code within the `spmd` body executes simultaneously on the MATLAB workers. The "multiple data" aspect means that even though the `spmd` statement runs identical code on all workers, each worker can have different, unique data for that code.

`spmd` blocks are useful when dealing with large data that cannot fit on a single machine. Unlike `parfor`, `spmd` blocks support inter-worker communication. They allow:

- Arrays (and operations on them) to be distributed across multiple workers
- Messages to be explicitly passed amongst workers.

The example below creates a distributed array (different parts of the array are located on different workers) and computes the `svd` of this distributed array. The `spmd` block returns the data in the form of a composite object (behaves similarly to cells in serial MATLAB. For specifics, see the documentation link below).

Note: The example below is located in the m-file, 'spmdExample1.m'.

```
matlabpool open 2 % can adjust according to your resources

M = 200;

spmd
   N = rand(M,M,codistributor);  % 200x100 chunk per worker
   A = svd(N);
end

A = max(A{1}); % Indexing into the composite object
disp(A)

clear N

matlabpool close
```

For information on `matlabpool`, see:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/matlabpool.html

For information about getting started using `parfor` loops, see:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/brb2x2l-1.html

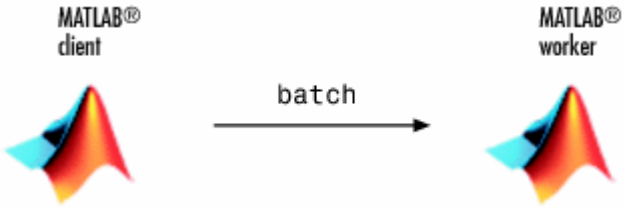
For information about getting started using `spmd` blocks, see:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/brukbno-2.html

For information regarding composite objects:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/brukctb-1.html

For information regarding distributed arrays:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/bqi9fln-1.html

2a. Using Batch to Submit Serial Code (Best for Scripts)

`batch` sends your serial script to run on one worker in your cluster. All of the variables in your client workspace (e.g. the MATLAB process you are



submitting from) are sent to the worker by default. You can alternatively pass a subset of these variables by defining the `Workspace` argument and passing the desired variables in a structure.

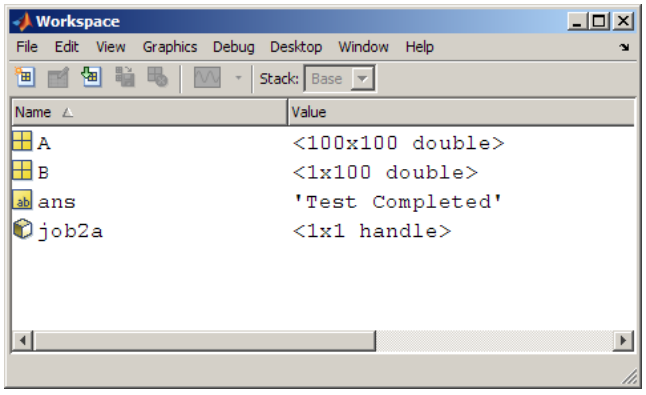
After your job has finished, you can use the `load` command to retrieve the results from the worker-workspace back into your client-workspace. In this and all examples following, we use a `wait` to ensure the job is done before we load back in worker-workspace. This is optional, but you can not load the data from a task or job until that task or job is finished. So, we use wait to block the MATLAB command line until that occurs.

If the `Configuration` argument is not provided, batch will use the default configuration that was set up above.

Note: For this example to work, you will need 'testBatch.m' on the machine that you are submitting from (i.e. the client machine). This example below is located in the m-file, 'submitJob2a.m'.

```
%% This script submits a serial script using batch
job2a = batch('testBatch');
wait(job2a);  % only can load when job is finished
sprintf('Finished Running Job')
load(job2a);  % loads all variables back
sprintf('Loaded Variables into Workspace')
% load(job2a, 'A');  % only loads variable A
destroy(job2a)  % permanently removes job data
sprintf('Test Completed')
```

If you have submitted successfully, you should see the following variables appear in your client workspace:



For more information on `batch`, see:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/batch.html

and here:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/brjw1e5-1.html#brjw1fx-3

2b. Using Batch to Submit Scripts that Run Using a MATLAB pool

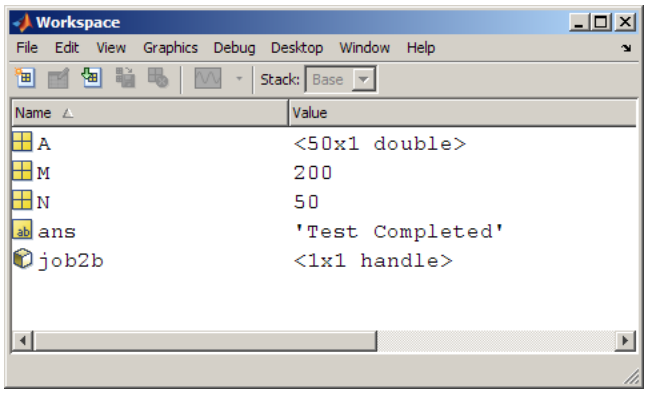
`batch` with the `'matlabpool'` option sends scripts containing `parfor` or `spmd` to run on workers via a MATLAB pool. In this process, one worker behaves like a MATLAB client process that facilitates the distribution of the job amongst the workers in the pool and runs the serial portion of the script. Therefore, specifying a `'matlabpool'` of size N actually will result in N+1 workers being used.

Just like in step 2a, all variables are automatically sent from your client workspace (i.e. the workspace of the MATLAB you are submitting from) to the worker's workspace on the cluster. `load` then brings the results from your worker's workspace back into your client's workspace. If a configuration is not specified, batch uses the default configuration as defined in the beginning of this section.

Note: For this example to work, you will need 'testParforBatch.m' on the machine that you are submitting from (i.e. the client machine). This example below is located in the m-file, submitJob2b.m.

```
%% This script submits a parfor script using batch
job2b = batch('testParforBatch','matlabpool',2);
wait(job2b);  % only can load when job is finished
sprintf('Finished Running Job')
load(job2b);  % loads all variables back
sprintf('Loaded Variables into Workspace')
% load(job2b, 'A');  % only loads variable A
destroy(job2b)  % permanently removes job data
sprintf('Test Completed')
```

If you have submitted successfully, you should see the following variables appear in your client workspace:

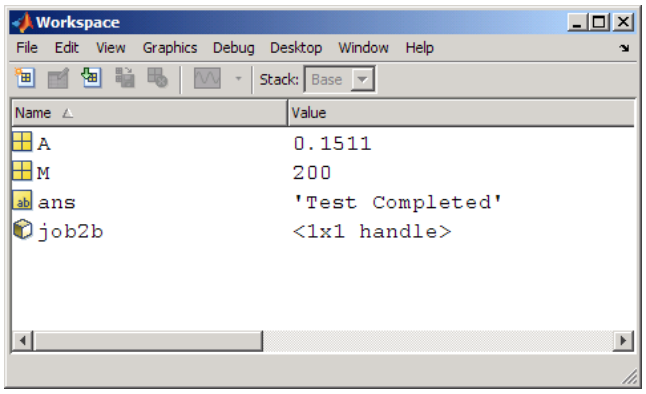


The above code submitted a script containing a parfor. You can submit a script containing a spmd block in the same fashion by changing the name of the submission script in the `batch` command.

Note: For this example to work, you will need 'testSpmdBatch.m' on the machine that you are submitting from (i.e. the client machine). This example below is located in the m-file, submitJob2b_spmd.m.

```
%% This script submits a spmd script using batch
job2b = batch('testSpmdBatch','matlabpool',2);
wait(job2b);  % only can load when job is finished
sprintf('Finished Running Job')
load(job2b);  % loads all variables back
sprintf('Loaded Variables into Workspace')
% load(job2b, 'A');  % only loads variable A
destroy(job2b)  % permanently removes job data
sprintf('Test Completed')
```

If you have submitted successfully, you should see the following variables appear in your client workspace:



3. Run Task-Parallel Example with Jobs and Tasks

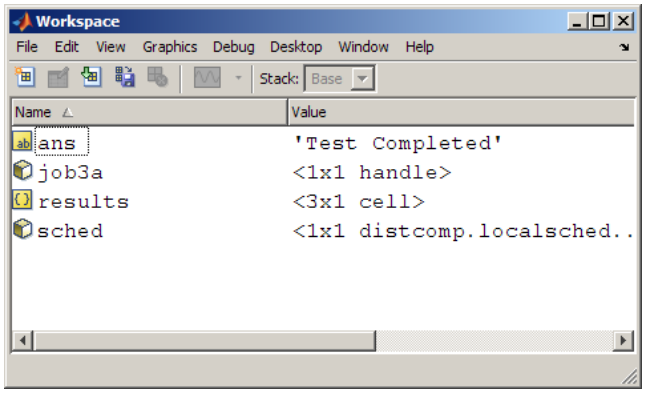
In this example, we are sending a task parallel job with multiple tasks. Each task evaluates the built-in MATLAB function. The `createTask` function in the below example is passed the job, the function to be run in the form of a function handle (`@sum`), the number of output arguments of the function (`1`), and the input argument to the `sum` function in the form of a cell array (`{[1 1]}`);

If not given a configuration, `findResource` uses the scheduler found in the default configuration defined in the beginning of this section.

Note: This example is located in the m-file, 'submitJob3a.m'.

```
%% This script submits a job with 3 tasks
sched = findResource();
job3a = createJob(sched);
createTask(job3a, @sum, 1, {[1 1]});
createTask(job3a, @sum, 1, {[2 2]});
createTask(job3a, @sum, 1, {[3 3]});
submit(job3a)
waitForState(job3a, 'finished') %optional
sprintf('Finished Running Job')
results = getAllOutputArguments(job3a);
sprintf('Got Output Arguments')
destroy(job3a) % permanently removes job data
sprintf('Test Completed')
```

If you have submitted successfully, you should see the following variables appear in your client workspace:



`results` should contain the following:

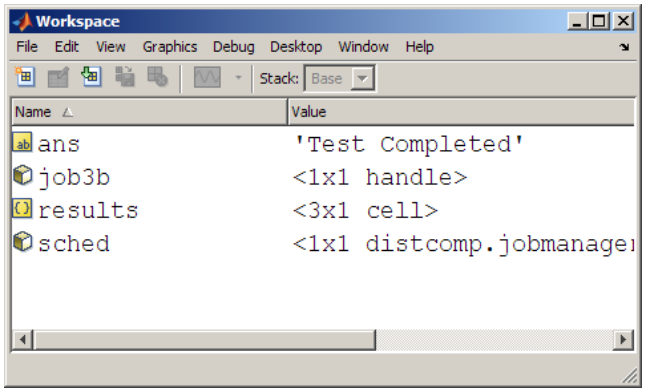
```
>> results

results =

    [2]
    [4]
    [6]
```

You can also call a user-created function in the same way as shown above. In that case, you will need to make sure that any scripts, files, or functions that the task function uses are accessible to the cluster. You can do this by sending those files to the cluster via the `FileDependencies` property or by directing the worker to a shared directory containing those files via the `PathDependencies` property. An example of using `FileDependencies` is shown below:

Note: you will need to have a 'testTask.m' file on the machine you are submitting from for this example to work. This example is located in the m-file, 'submitJob3b.m'.

```
% This script submits a job with 3 tasks
sched = findResource();
job3b = createJob(sched,'FileDependencies',{'testTask.m'});
createTask(job3b, @testTask, 1, {1,1});
createTask(job3b, @testTask, 1, {2,2});
createTask(job3b, @testTask, 1, {3,3});
submit(job3b)
waitForState(job3b, 'finished') % optional
sprintf('Finished Running Job')
results = getAllOutputArguments(job3b);
sprintf('Got Output Arguments')
destroy(job3b) % permanently removes job data
sprintf('Test Completed')
```

If you have submitted successfully, you should see the following variables appear in your client workspace:



`results` should contain the following:

```
>> results

results =

   [2]
   [4]
   [6]
```

For more information on File and Path Dependencies, see the below documentation.

File Dependencies:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/filedependencies.html

Path Dependencies:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/pathdependencies.html

More general overview about sharing code between client and workers:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/bqur7ev-2.html#bqur7ev-9

4. Run Task-Parallel Example with a MATLAB pool job (best for parfor or spmd in functions)

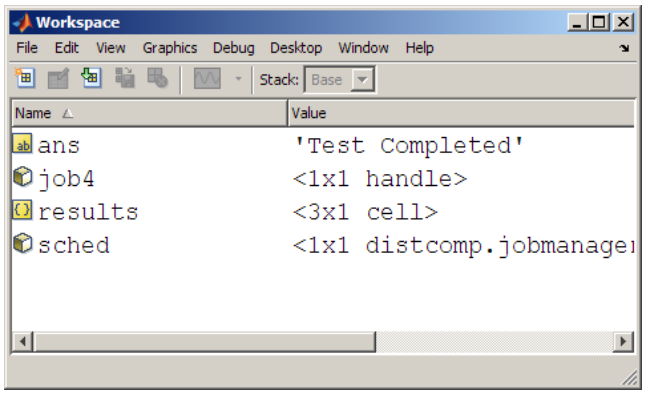
In this example, we are sending a MATLAB pool job with a single task. This is nearly equivalent to sending a batch job (see step 2b) with a `parfor` or a `spmd` block, except this method is best used when sending functions and not scripts. It behaves just like `jobs/tasks` explained in step 3. The function referenced in the task contains a `parfor`.

Note: For this example to work, you will need 'testParforJob.m' on the machine that you are submitting from (i.e. the client machine). This example is located in the m-file, 'submitJob4.m'.

```
% This script submits a function that contains parfor
sched = findResource();
job4 = createMatlabPoolJob(sched,'FileDependencies',...
    {'testParforJob.m'});
createTask(job4, @testParforJob, 1, {});
set(job4, 'MaximumNumberOfWorkers', 3);
set(job4, 'MinimumNumberOfWorkers', 3);
submit(job4)
waitForState(job4, 'finished') % optional
sprintf('Finished Running Job')
results = getAllOutputArguments(job4);
sprintf('Got Output Arguments')
```

```
destroy(job4) % permanently removes job data
sprintf('Test Completed')
```

If you have submitted successfully, you should see the following variables appear in your client workspace:



`results{1}` should contain a `[50x1 double]`.

For more information on creating and submitting MATLAB pool jobs, see http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/creatematlabpooljob.html

5. Run Data-Parallel Example

In this step, we are sending a data parallel job with a single task. The format is similar to that of jobs/tasks (see step 3). For parallel jobs, you only have one task. That task refers to a function that uses distributed arrays, `labindex`, or some mpi functionality. In this case, we are running a simple built in function (`labindex`) which takes no inputs and returns a single output.

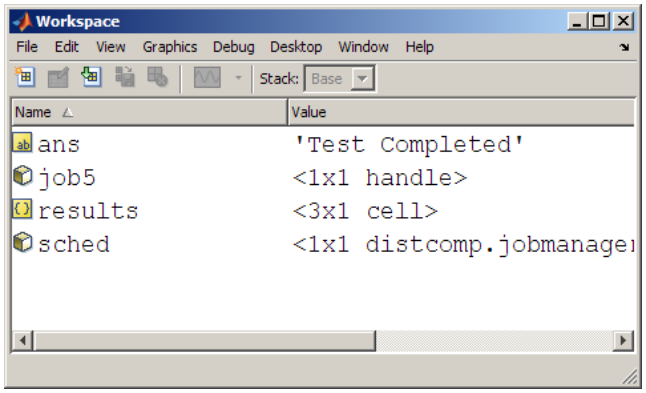
`labindex` returns the ID value for each of worker processes that ran the it . The value of `labindex` spans from 1 to n, where n is the number of labs running the current job

Note: This example is located in the m-file, 'submitJob5.m'.

```
%% Script submits a data parallel job, with one task
sched = findResource();
job5 = createParallelJob(sched);
createTask(job5, @labindex, 1, {});
set(job5, 'MaximumNumberOfWorkers', 3);
set(job5, 'MinimumNumberOfWorkers', 3);
submit(job5)
waitForState(job5, 'finished') % optional
sprintf('Finished Running Job')
results = getAllOutputArguments(job5);
sprintf('Got Output Arguments')
```

```
destroy(job5); % permanently removes job data
sprintf('Test Completed')
```

If you have submitted successfully, you should see the following variables appear in your client workspace:



`results` should contain the following:

```
>> results

results =

    [1]
    [2]
    [3]
```

For more information on creating and submitting data parallel jobs, see:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/createparalleljob.html

For more information on, `labindex`, see:
http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/labindex.html

5 Submitting Jobs to Your Cluster

Rerun the above section (Section 4) with your configuration changed from `local` to the configuration name corresponding to your cluster.

❖ 6. Summary Chart for Scheduling Options

	parfor	spmd	function	script	pure task parallel	pure data parallel	parallel and serial
batch	✓	✓		✓			✓
matlabpooljob	✓	✓	✓				✓
jobs and tasks			✓		✓		
paralleljob			✓			✓	

7. Next Steps

Refer to the documentation for the Parallel Computing Toolbox to learn about more functionality for solving your MATLAB problems in parallel. A good place to start is here:

http://www.mathworks.com/access/helpdesk/help/toolbox/distcomp/index.html?/access/helpdesk/help/toolbox/distcomp/f3-6010.html

You can also consider taking a training course through The MathWorks to learn more in focused, hands-on environment.

http://www.mathworks.com/services/training/index.html