# MATLAB Parallel Computing Toolbox

User’s guide, **R2013a**

Aim: This tutorial introduces a MATLAB user to The MathWorks parallel computing tools. Through code examples, the user will learn to run parallel MATLAB applications using a multicore desktop computer or a cluster of computers.

**Getting Started with Parallel Computing using MATLAB**

Parallel Computing Toolbox lets you solve computationally and data-intensive problems using multicore processors, GPUs, and computer clusters.

High-level constructs parallel for-loops, special array types, and parallelized numerical algorithms let you parallelize MATLABapplications without CUDA or MPI programming.

The toolbox provides twelve workers (MATLAB computational engines) to execute applications locally on a multicore desktop. Without changing the code, you can run the same application on a computer cluster or a grid computing service (using MATLAB Distributed Computing Server). You can run parallel applications interactively or in batch.



**Life Cycle of a Job**

The figure below illustrates the stages in the life cycle of a job. When you create and run a job, it progresses through a number of stages.

**Pending:** You create a job on the scheduler with the *createJob* function in your client session of Parallel Computing Toolbox software. The job’s first state is pending. This is when you define the job by adding tasks to it.

**Queued:** When you execute the submit function on a job, scheduler places the job in the queue, and the job’s state is queued. The scheduler executes jobs in the queue in the sequence in which they are submitted, all jobs moving up the queue as the jobs before them are finished.

**Running:** When a job reaches the top of the queue, the scheduler distributes the job’s tasks to worker sessions for evaluation. The job’s state is now running. If more workers are available than are required for a job’s tasks, the scheduler begins executing the next job. In this way, there can be more than one job running at a time.

**Finished:** When all of a job’s tasks have been evaluated, the job is moved to the finished state. At this time, you can retrieve the results from all the tasks in the job with the function *fetchOutputs.*

**Failed:** When using a third-party scheduler, a job might fail if the scheduler encounters an error when attempting to execute its commands or access necessary files.

**Deleted:** When a job’s data has been removed from its data location, the state of the job in the client is deleted. This state is available only as long as the job object remains in the client.



**Configuration**

**Cluster Profile Manager**

Cluster profiles let you define certain properties for your cluster, and then have these properties applied when you create cluster, job, and task objects in the MATLAB client. Some of the functions that support the use of cluster profiles are

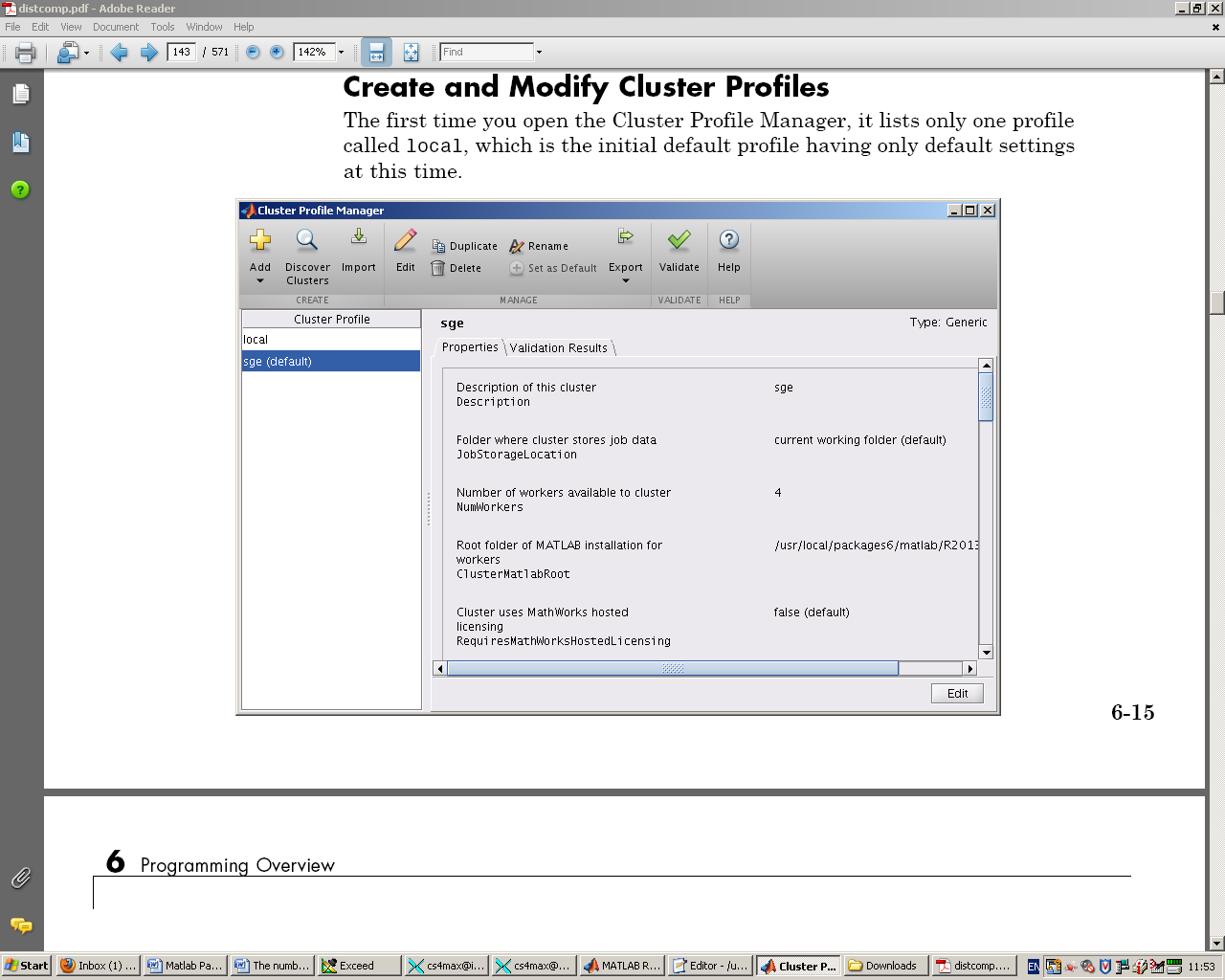
* batch
* matlabpool
* parcluster

To create, edit, and import cluster profiles, you can do this from the Cluster Profile Manager. To open the Cluster Profile Manager, on the **Home** tab in the **Environment** section, click **Parallel > Manage Cluster Profiles**.



**Modify Cluster Profiles**

When you open the Cluster Profile Manager, it lists profiles called local and sge (default). We initialised sge as the default profile.



**The number of workers which are determined for which version of Matlab:**

The number of local workers available with Parallel Computing Toolbox and no MDCS has changed according to the release of MATLAB you are using. When introduced, the limit was 4; this changed to 8 in R2009a; and to 12 in R2011b. ***The limit on the number of workers is still 12 in R2013a.*** (We are using R2013a)

By default, the maximum number of workers MATLABPOOL will start is equal to the number of physical cores you have on your machine. If you have, for example, 8 logical cores in addition to the 4 physical cores, you may utilise these as follows:  
  
1) Go to Parallel > Manage Cluster Profiles > local > Edit and change the value of "Number of workers to start on your local machine" to 12.  
  
2) Type the following command in the MATLAB command window  
  
>> matlabpool open 12  
  
If you want to use 16 workers, you will need a 16-node MDCS licence, and you will also need to set up a scheduler to manage those. After this you could use the syntax "matlabpool open 16".  
  
There should be no limit on the number of jobs as these will just be queued in the scheduler.

Note:

The only limitation on number of tasks is when you have a communicating job - where the workers have to all be running simultaneously. In such a job, which could be a matlabpool / spmd type job, the limit is equivalent to the number of workers. For non-communicating jobs, the scheduler will just keep the tasks in the queue until a worker becomes available.

**Monitoring Memory Usage with Matlab**

When submitting task to iceberg specify memory requirements e.g.

* qsh –l mem=2G etc…
* qsub –l mem=2G etc… (or make sure #$ -l mem=2G is at the head of the script file)

***Note: We specified mem=15G in case of running matlabpool in matlab2013a***

Maximum Number of workers available to Matlabpool in iceberg is 80.

**Parallel Computing with MATLAB on a Single Machine**

For setup on a single machine without a separate cluster, you can use the local cluster included with Parallel Computing Toolbox to run as many as 12 workers on a single MATLAB client machine. This local cluster does not require a separate job scheduler or MATLAB Distributed Computing Server, so these instructions are not necessary.

If you want to run more than 12 workers on a single machine, you can use the MATLAB Distributed Computing Server which is available in Iceberg cluster. We initialised sge the default profile.

**Parallel computing Unix scripts using MATLAB parallel computing toolbox**

**Submitting a matlab job to Sun Grid Engine on Iceberg:**

What you have to do:

* Run Matlab from Iceberg: When submitting task to iceberg specify memory requirements e.g.

qsh –l mem=2G etc…

qsub –l mem=2G etc… (or make sure #$ -l mem=2G is at the head of the script file)

* *qsub mymatlabjob.sh*
* *Where mymatlabjob.sh is:*

*#!/bin/sh*

*#$ -cwd*

*#$ -m be*

*#$ -M user@sheffield.ac.uk*

*#$ -l h\_rt=hh:mm:ss*

*#$ -l mem=6G*

*/usr/local/bin/matlab2013a -nosplash -nodisplay -r matlabscriptfile*

**Note: matlabscriptfile should be without “.m”**

**Running Parallel Applications Interactively and as Batch Jobs**

There are several ways to execute a parallel MATLAB program:

* interactive local (matlabpool), suitable for the desktop;
* indirect local, (batch or createJob/createTask);
* indirect remote, (batch or createJob/createTask), requires setup.

The University of Sheffield cluster ***Iceberg*** will accept parallel MATLAB jobs submitted from a user's desktop, and will return the results when the job is completed. Making this possible requires an Iceberg account.

You can execute parallel applications interactively and in batch using Parallel Computing Toolbox. Using the matlabpool command, you can connect your MATLAB session to a pool of MATLAB workers that can run either locally on your desktop (using the toolbox) or on a computer cluster (using MATLAB Distributed Computing Server) to setup a dedicated interactive parallel execution environment. You can execute parallel applications from the MATLAB prompt on these workers and retrieve results immediately as computations finish, just as you would in any MATLAB session.

Running applications interactively is suitable when execution time is relatively short. When your applications need to run for a long time, you can use the toolbox to set them up to run as batch jobs. This enables you to free your MATLAB session for other activities while you execute large MATLAB and Simulink applications.

While your application executes in batch, you can shut down your MATLAB session and retrieve results later.

**Parallel computing toolbox**

1. **Parallel for-Loops (parfor)**

The simplest path to parallelism is the parfor statement, which indicates that a given for loop can be executed in parallel. When the client MATLAB reaches such a loop, the iterations of the loop are automatically divided up among the workers, and the results gathered back onto the client. Using parfor requires that the iterations are completely independent; there are also some restrictions on data access. Using parfor is similar to OpenMP.

Parallel for-loops let you distribute a set of independent tasks over a set of workers. The parfor construct uses the familiar for-loop syntax and is ideal for parameter sweeps and similar tasks.

The parfor construct has mechanisms for detecting and exchanging the necessary data and code between the client MATLAB session and workers. It also detects the presence of workers automatically. As a result you do not have to construct and submit complex batch jobs to the cluster.

The parfor loop tries to divide the work among multiple processors by allocating iterations to the four different workers. The only requirement for distributing execution using parfor is that iterations must be independent of each other, and no communication can occur between workers during the execution of the loop.

Work distribution is dynamic. Instead of being allocated a fixed iteration range, the workers are allocated a new iteration only after they finish processing their current iteration, which results in an even work load distribution.

Note: Parallelism doesn't pay until your problem is big enough.

You interact with workers using the matlabpool command directly from MATLAB command window. The command sets up the interactive execution environment for parallel constructs such as parfor. The parfor loops can be issued from command line, as well as within functions and scripts. Using the batch command (for MATLAB scripts) and the createMatlabPoolJob command (for MATLAB functions), you can send code containing parfor for execution offline.

**MATLAB Functions**

1. ***matlabpool*** Open or close pool of MATLAB sessions for parallel computation

matlabpool enables the full functionality of the parallel language features (parfor) in MATLABby creating a special job on a pool of workers, and connecting the pool to the MATLAB client. matlabpool starts a worker pool using the default cluster profile, with the pool size specified by that profile. You can also specify the pool size using matlabpool poolsize, but most clusters have a maximum number of processes that they can start (12 for a local cluster). poolsize must be a literal numeric value.

***matlabpool******open*** ... indicates explicitly to open a pool. Without specifying open or close, the command default is open.

***matlabpool size***  returns the size of the worker pool if it is open, or 0 if the pool is closed.

***matlabpool close*** stops the worker pool, deletes the pool job, and makes all parallel language features revert to using the MATLAB client for computing their results. Without an open pool, spmd and parfor run as a single thread in the client.

**Examples**

1. Start a pool using the default profile to define the number of workers: (default number of workers Matlabpool in iceberg is : ---- )

***matlabpool***

1. Start a pool of 12 workers using a profile called myProf:

***matlabpool open myProf 12***

1. Start a pool of 2 workers using the local profile:

***matlabpool local 2***

1. Run matlabpool as a function to check whether the worker pool is currently open:

***isOpen = matlabpool('size') > 0***

1. Start a pool with the default profile, and pass two code files to the workers:

***matlabpool('open','AttachedFiles',{'mod1.m','mod2.m'})***

1. Create an object representing the cluster identified by the default profile, and use that cluster object to start a MATLAB pool. The pool size is determined by the default profile:

***c = parcluster***

***matlabpool(c)***

1. ***parfor*** Execute code loop in parallel

**Syntax**

parfor (i = 1 : n)

% do something with i

end

Allows you to write a loop for a statement or block of code that executes in parallel on a cluster of workers, which are identified and reserved with the matlabpool command.

**Example**

1. 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

Parfor will be significantly faster than the corresponding for statement.

1. Another example:

Tic;

for i=1:1024

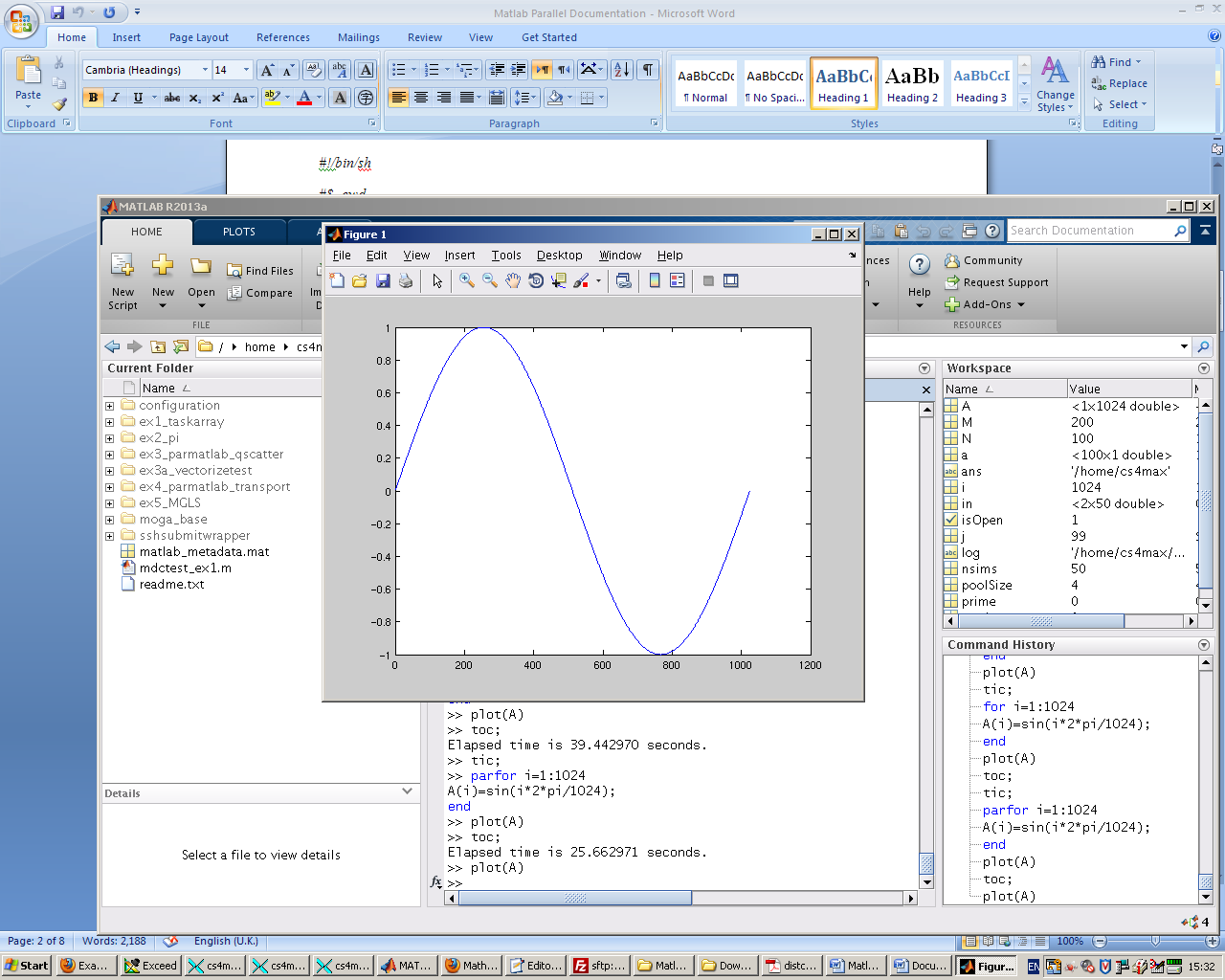
A(i) = sin(i\*2\*pi/1024);

end

plot(A)

toc;

Elapsed time is **39.442970** second.



matlabpool open

tic;

parfor i=1:1024

A(i) = sin(i\*2\*pi/1024);

end

plot(A)

toc;

Elapsed time is **25.662971** second.

**Deciding When to Use parfor**

A parfor-loop is useful in situations where you need many loop iterations of a simple calculation, such as a Monte Carlo simulation. Parfor divides the loop iterations into groups so that each worker executes some portion of the total number of iterations. Parfor-loops are also useful when you have loop iterations that take a long time to execute, because the workers can execute iterations simultaneously.

You cannot use a parfor-loop when iteration in your loop depends on the results of other iterations. Each iteration must be independent of all others. Since there is a communications cost involved in a parfor-loop, there might be no advantage to using one when you have only a small number of simple calculations.

**Example**

If you use a nested for-loop to index into a sliced array, you cannot use that array elsewhere in the parfor-loop. For example, in the following example, the code (Exp1) does not work because A is sliced and indexed inside the nested for-loop; the code (Exp2) works because v is assigned to A outside the nested loop:

**Exp1.**

A = zeros(4, 10);

parfor i = 1:4

for j = 1:10

A(i, j) = i + j;

end

disp(A(i, 1))

end

* Output : Error

**Exp2.**

A = zeros(4, 10);

parfor i = 1:4

v = zeros(1, 10);

for j = 1:10

v(j) = i + j;

end

disp(v(1))

A(i, :) = v;

End

* Output

3

5

4

2

1. **Batch Processing**
2. Run a batch script on a worker, without using a MATLAB pool:

j = batch('script1');

1. Run a batch script that requires two additional files for execution:

j = batch('myScript','AttachedFiles',{'mscr1.m','mscr2.m'});

wait(j); % Wait for the job to finish

load(j); % Load job workspace data into client workspace

1. Run a batch MATLAB pool job on a remote cluster, using eight workers for the MATLAB pool in addition to the worker running the batch script. Capture the diary, and load the results of the job into the workspace. This job requires a total of nine workers:

j = batch('script1', 'matlabpool', 8, 'CaptureDiary', true);

wait(j); % Wait for the job to finish

diary(j) % Display the diary

load(j) % Load job workspace data into client workspace

Note: diary(job) displays the Command Window output from the batch job in the MATLABCommand Window. The Command Window output will be captured only if the batch command included the 'CaptureDiary' argument with a value of true.

1. Run a batch MATLAB pool job on a local worker, which employs two other local workers for the pool. Note, this requires a total of three workers in addition to the client, all on the local machine:

j = batch('script1', 'Profile', 'local', ...

'matlabpool', 2);

1. Clean up a batch job's data after you are finished with it:

delete(j)

1. Run a batch function on a cluster that generates a 10-by-10 random matrix:

Note: in old examples instead of function parcluster, function findResource has been used. (In matlab2013a parcluster has been replaced with findResource.)

c = parcluster();

j = batch(c, @rand, 1, {10, 10});

wait(j) % Wait for the job to finish

diary(j) % Display the diary

r = fetchOutputs(j) % Get results into a cell array

r{1} % Display result

**Implementing Data-Parallel Applications using the Toolbox and MATLAB Distributed Computing Server**

Distributed arrays in Parallel Computing Toolbox are special arrays that hold several times the amount of data that your desktop computer’s memory (RAM) can hold. Distributed arrays apportion the data across several MATLAB worker processes running on a computer cluster (using MATLAB Distributed Computing Server). As a result, with distributed arrays you can overcome the memory limits of your desktop computer and solve problems that require manipulating very large matrices.

In this part, we introduce some examples using the Toolbox and MATLAB Distributed Computing Server: these examples are available in: ***/usr/local/courses/matlab\_examples/dct***

* In these examples a parallel.Job object provides access to a job, which you create, define, and submit for execution.
* creatTask: create new task in job
* createJob: Create independent job on cluster
* createCommunicatingJob: Create communicating job on cluster

job = createCommunicatingJob(cluster) creates a communicating job object for the identified cluster. (Note: In matlab2013a createCommunicatingJob has been replaced with creatParallelJob.)

**Examples:**

1. **Create and Run a Basic Job**

Construct an independent job object using the default profile:

c = parcluster % Create cluster object

j = createJob(c);

Add tasks to the job:

for i = 1:10

createTask(j,@rand,1,{10});

end

Run the job:

submit(j);

Wait for the job to finish running, and retrieve the job results:

wait(j);

out = fetchOutputs(j);

Display the random matrix returned from the third task:

disp(out{3});

Delete the job:

delete(j);

1. **Create a Job with One Task**

Create a job object.

c = parcluster(); % Use default profile

j = createJob(c);

Add a task object which generates a 10-by-10 random matrix.

t = createTask(j, @rand, 1, {10,10});

Run the job.

submit(j);

Wait for the job to finish running, and get the output from the task evaluation.

wait(j);

taskoutput = fetchOutputs(j);

Show the 10-by-10 random matrix.

disp(taskoutput{1});

1. **Create a Job with Three Tasks**

This example creates a job with three tasks, each of which generates a 10-by-10 random matrix.

c = parcluster(); % Use default profile

j = createJob(c);

t = createTask(j, @rand, 1, {{10,10} {10,10} {10,10}});

1. **GPU Computing**

Transfer data between MATLAB and a graphics processing unit (GPU); run code on a GPU

Parallel Computing Toolbox provides GPUArray, a special array type with several associated functions that lets you perform computations on CUDA-enabled NVIDIA GPUs directly from MATLAB.

* gpuDevice: Query or select GPU device
* gpuDeviceCount: Number of GPU devices present
* gpuArray: Create array on GPU
* gather: Transfer distributed array data or gpuArray to local workspace

**Note: running gpu from iceberg node:**

**Qsh –l arch=intel\* -l gpu=1 –l mem=12G –l rmem=12G –P gpu**

References:

1. [https://www.mathworks.co.uk/searchresults/?search\_submit=matlabcentral&query=matlab+parallel+computation+example&q=matlab+parallel+computation+example&c[]=matlabcentral](https://www.mathworks.co.uk/searchresults/?search_submit=matlabcentral&query=matlab+parallel+computation+example&q=matlab+parallel+computation+example&c%5b%5d=matlabcentral)
2. <http://www.mathworks.co.uk/help/distcomp/parallel-for-loops-parfor.html>
3. <http://www.mathworks.co.uk/help/distcomp/index.html>
4. <http://www.mathworks.co.uk/help/distcomp/examples/index.html>
5. <http://www.mathworks.co.uk/products/distriben/examples.html?s_cid=BB>