Parallel Computing & Code Optimization with MATLAB

Set-up the working directory

```
addpath(genpath('/Users/matteoconti/Documents/PhD/Lectures/Parallel Computing MATLAB/le
```

Basic MATLAB code optimizations

MATLAB Code Analyzer

The Code Analyzer in the MATLAB Editor checks your code while you are writing it. The Code Analyzer identifies potential problems and recommends modifications to maximize performance.

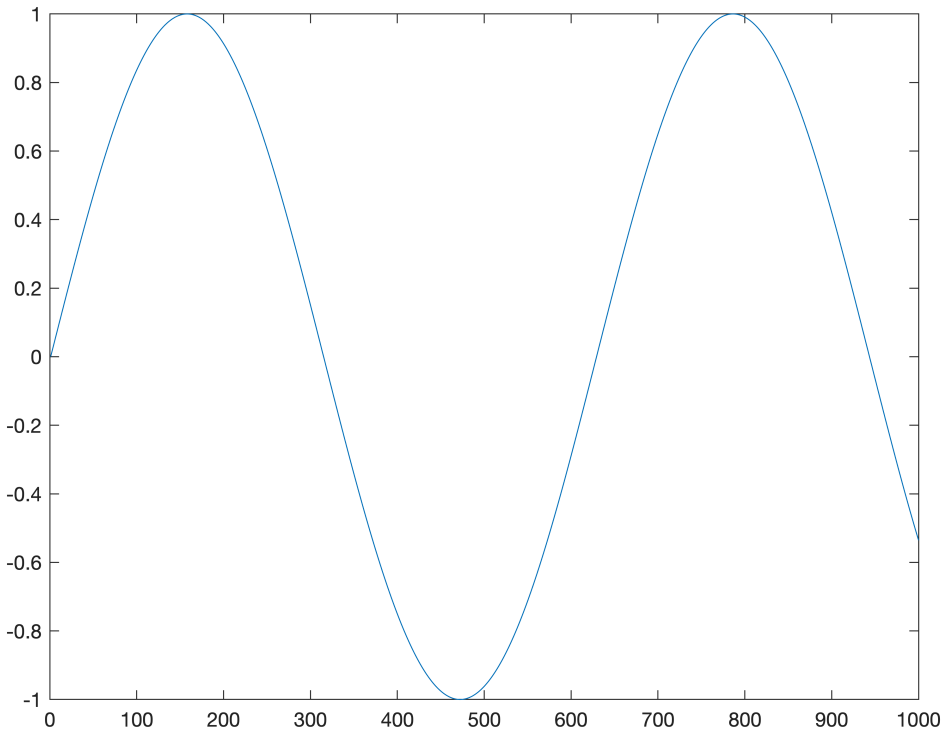
Memory preallocation

With preallocation, you initialize an array using the final size required for that array. Preallocation helps you **avoid dynamically resizing arrays**, particularly when code contains for and while loops.

```
% Without memory preallocation
tic
i = 0;
y1 = 0;
for t = 0:.01:1e4
    i = i + 1;
    y1(i) = sin(t);
end
plainTime = toc
```

```
plainTime = 0.4002
```

```
figure;plot(y1(1:1e3))
```




```
% With memory preallocation
tic
i = 0;
y2 = zeros(size(0:.01:1e4));
for t = 0:.01:1e4
    i = i + 1;
    y2(i) = sin(t);
end
prealTime = toc
```

```
prealTime = 0.0667
```

```
percentDifference = plainTime/prealTime;
sprintf('Speedup factor of %f',(percentDifference))
```

```
ans =
'Speedup factor of 5.997233'
```

```
% Show equality
max(abs(y1-y2))
```

```
ans = 0
```

Vectorization

Vectorization is the process of converting code from using loops to using **matrix** and **vector operations**. The larger the matrices and vectors, the more important this optimization becomes.

```
% Vectorization: Example 1
tic
t = 0:.01:1e4;
y3 = sin(t);
vecTime = toc
```

```
vecTime = 0.0282
```

```
percentDifference = plainTime/vecTime;
sprintf('Speedup factor of %f',(percentDifference))
```

```
ans =
'Speedup factor of 14.177503'
```

```
% Show equality
max(abs(y1-y3))
```

```
ans = 1.8190e-12
```

```
% Vectorization: Example 2
sz = [100 100 100];
A = rand(sz);
B = rand(sz);

% "Never do this in your life please" version
tic
for ii=1:sz(1)
    for jj=1:sz(2)
        for kk=1:sz(3)
            C1(ii,jj,kk) = A(ii,jj,kk)*B(ii,jj,kk);
        end
    end
end
ggTime = toc
```

```
ggTime = 0.2091
```

```
% Vectorized version
tic
C2 = A.*B;
vecTime2 = toc
```

```
vecTime2 = 0.0155
```

```
percentDifference = ggTime/vecTime2;
sprintf('Speedup factor of %f',(percentDifference))
```

```
ans =
```

```
'Speedup factor of 13.516506'
```

```
% Show equality
isequal(C1,C2)
```

```
ans = logical
   1
```

MEX Files

1. Preparing MATLAB Code for Code Generation

you should first examine your MATLAB code and make sure it is compliant with code generation standards. The first step in this process is adding the compilation directive

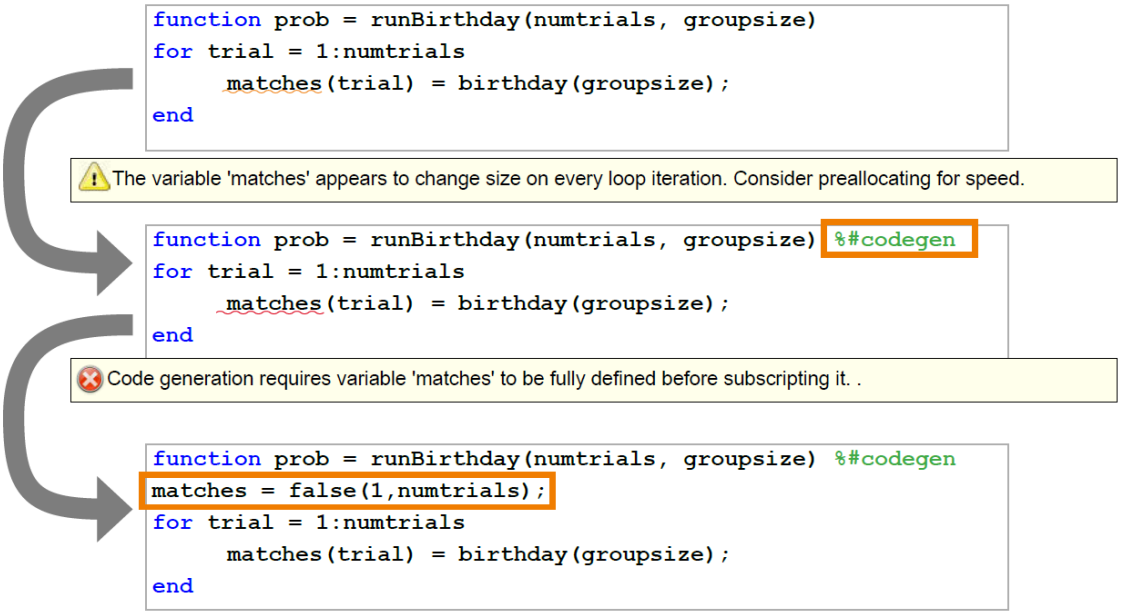
```
%#codegen
```

to the first line of your MATLAB code. This is a meta comment to indicate that you intend to generate code for the MATLAB algorithm. Adding this directive instructs the MATLAB Code Analyzer (which gives tooltips as indicated by the bar on the right of the MATLAB Editor) to help you diagnose and correct violations that would result in errors during code generation.

```
edit runBirthday
```

Add %#codegen to the first line and observe code generation errors indicated by MATLAB Code Analyzer. Add the preallocation code:

```
matches = false (1, numTrials);
```

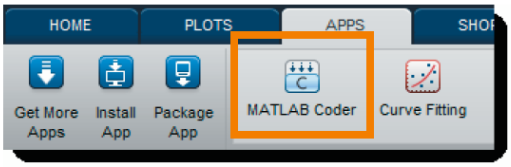


2. Creating a MEX file

You can use the command line to create a MEX file from a function

```
codegen runBirthday -args {double(0), double(0)}
```

Alternatively you can MATLAB Coder app interface



3. Verifying Generated Code

You can use the generated MEX-file to verify that the generated (and compiled) code yields the same result as the original MATLAB function and to compare the execution time.

To run the generated MEX-file using the same input you used for the original MATLAB function and verify the result

1. Run the original MATLAB function

```
tic, prob1 = runBirthday(1e5, 30), mlTime = toc
```

```
prob1 = 0.7074
mlTime = 0.7098
```

2. Run the generated MEX-function

```
tic, prob2 = runBirthday_mex(1e5, 30), mexTime = toc
```

```
prob2 = 0.7050
mexTime = 1.0557
```

3. Compute the speedup

```
percentDifference = mlTime/mexTime;
sprintf('Speedup factor of %f',(percentDifference))
```

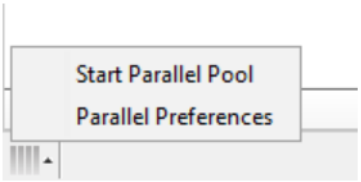
```
ans =
'Speedup factor of 0.672332'
```

Parallel Toolbox

How to start a parallel pool

1) By default, a parallel pool starts **automatically** (implicit way) when needed by certain parallel language features. Many functions can automatically start a parallel pool.

2) Control the Parallel Pool from MATLAB Desktop using the parallel status indicator in the lower left corner of the MATLAB desktop, or the parallel button in the toolstrip, to start a parallel pool manually.



3) Using the **Programming Interface** to start and stop a parallel pool programmatically by using default settings or specifying alternatives. To open a parallel pool based on your preference settings:

```
maxPool_size = 4;

if isempty(gcp('nocreate'))
    try
        parpool('local',[1 maxPool_size]);
    catch e
        if strcmp(e.identifier, 'parallel:convenience:ConnectionOpen')
            p = gcp('nocreate'); % do not create a pool if it does not exist already
            warning(e.identifier,['Parallel Computing Pool already running with %i work
                    'Details of the error:\nError ID: %s.\nError message: %s.\n'], p.
        else
            warning(e.identifier,['There was a problem while trying to open the Paralle
                    'Details of the error:\nError ID: %s.\nError message: %s.\n'], e.
        end
    end
end
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

```
% delete(gcp('nocreate')) % delete the current pool
```

Parallel-enabled Toolboxes

Example: Image Compression using Discrete Cosine Transform (DCT)

The discrete cosine transform (DCT) represents an image as a sum of sinusoids of varying magnitudes and frequencies. The `dct2` function computes the two-dimensional discrete cosine transform (DCT) of an image. The DCT has the property that, for a typical image, most of the visually significant information about the image is concentrated in just a few coefficients of the DCT. For this reason, the DCT is often used in image compression applications. For example, the DCT is at the heart of the international standard lossy image compression algorithm known as JPEG.

```
im = imread('https://effigis.com/wp-content/uploads/2015/02/DigitalGlobe_WorldView2_50c
imshow(im)
```

```
imfinfo('https://effigis.com/wp-content/uploads/2015/02/DigitalGlobe_WorldView2_50cm_8k
```

```
ans = struct with fields:
            Filename: 'https://effigis.com/wp-content/uploads/2015/02/DigitalGlobe_WorldView2_50cm_8bit_Pan
        FileModDate: '22-Oct-2020 02:24:19'
           FileSize: 17878139
             Format: 'jpg'
      FormatVersion: ''
              Width: 4098
             Height: 4177
           BitDepth: 24
          ColorType: 'truecolor'
    FormatSignature: ''
    NumberOfSamples: 3
```

```
      CodingMethod: 'Huffman'
     CodingProcess: 'Sequential'
           Comment: {}
       Orientation: 1
       XResolution: 72
       YResolution: 72
    ResolutionUnit: 'Inch'
          Software: 'Adobe Photoshop CS6 (Macintosh)'
          DateTime: '2015:02:05 16:28:40'
     DigitalCamera: [1×1 struct]
     ExifThumbnail: [1×1 struct]
```

```
I = double(rgb2gray(im));
I = imresize(I,[4000 4000]); %J = imresize(I,[numrows numcols])
imshow(I,[])
```

Running in serial

When images are either too large to load into memory, or else they can be loaded into memory but then be too large to process. To avoid these problems, you can process large images incrementally: reading, processing, and finally writing the results back to disk, one region at a time. The `blockproc` function helps you with this process. Using `blockproc`, specify an image, a block size, and a function handle. `blockproc` then divides the input image into blocks of the specified size, processes them using the function handle one block at a time, and then assembles the results into an output image. `blockproc` returns the output to memory or to a new file on disk.

```
mask = [1   1   1   1   0   0   0   0
        1   1   1   0   0   0   0   0
        1   1   0   0   0   0   0   0
        1   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0];
% Discard all but 10 of the 64 DCT coefficients in each block.
tic;
%-------------------------- Compression -------------------------
dctcoef = blockproc(I,[8 8],@(block_struct) dct2(block_struct.data)); % Compute the two
cuttcoef = blockproc(dctcoef,[8 8],@(block_struct) block_struct.data.*mask); % Discard
%-------------------------- Decompression ----------------------
decompI = blockproc(cuttcoef,[8 8], @(block_struct)idct2(block_struct.data)); % Reconst
time_serial = toc
```

```
time_serial = 56.0464
```

```
DIF = imsubtract(I,decompI);
mse = mean(mean(DIF.*DIF));
rmse = sqrt(mse);
psnr = 20*log(255/rmse);
disp('PSNR');disp(psnr)
```

```
PSNR
   64.5756
```

```
imshowpair(I,decompI,'montage')
```

Display the original image and the reconstructed image, side-by-side. Although there is some loss of quality in the reconstructed image, it is clearly recognizable, even though almost 85% of the DCT coefficients were discarded.

Running in parallel

The `blockproc` function can take advantage of multiple processor cores on your machine to perform parallel block processing.

```
mask = [1   1   1   1   0   0   0   0
        1   1   1   0   0   0   0   0
        1   1   0   0   0   0   0   0
        1   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0
        0   0   0   0   0   0   0   0];
tic;
%-------------------------- Compression -------------------------
dctcoef = blockproc(I,[8 8],@(block_struct) dct2(block_struct.data),'UseParallel',true)
cuttcoef = blockproc(dctcoef,[8 8],@(block_struct) block_struct.data.*mask,'UseParallel
%-------------------------- Decompression ----------------------
decompI = blockproc(cuttcoef,[8 8], @(block_struct)idct2(block_struct.data),'UseParalle
time_parallel = toc
```

```
time_parallel = 39.2576
```

```
disp(['Parallel computing ' num2str(time_serial/time_parallel) ' times faster'])
```

```
Parallel computing 1.4277 times faster
```

```
DIF = imsubtract(I,decompI);
mse = mean(mean(DIF.*DIF));
rmse = sqrt(mse);
psnr = 20*log(255/rmse);
disp('PSNR');disp(psnr)
```

```
PSNR
   64.5756
```

```
imshowpair(I,decompI,'montage')
```

Common parallel programming constructs: parfor

Example 1

```
% Running in serial
n = 100;
A = 500;
a = zeros(1,n);

tic
for i=1:n
    a(i) = max(abs(eig(rand(A))));
end
time_serial = toc
```

```
time_serial = 15.9780
```

```
% Running in parallel
n = 100;
```

```
A = 500;
a = zeros(1,n);

tic
parfor i=1:n
    a(i) = max(abs(eig(rand(A))));
end
time_parallel = toc
```

```
time_parallel = 11.0260
```

```
disp(['Parallel computing ' num2str(time_serial/time_parallel) ' times faster'])
```

```
Parallel computing 1.4491 times faster
```

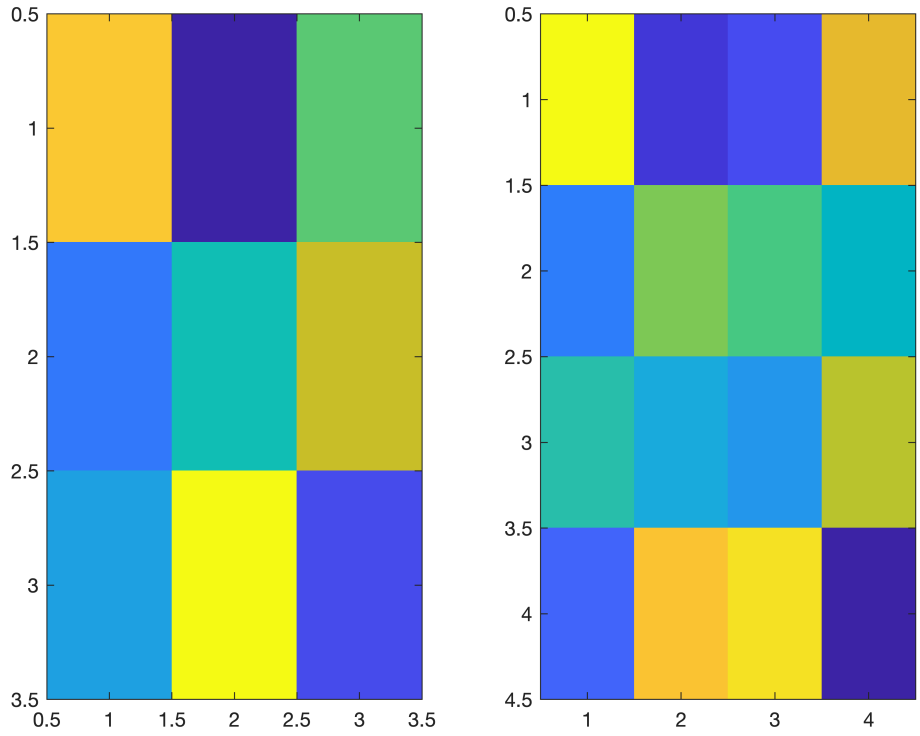
Advanced parallel programming constructs: spmd

```
maxPool_size = 4;
if isempty(gcp('nocreate')) % create a parallel pool if does not exist
    parpool('local',[1 maxPool_size]);
end
```

Example 1

```
spmd
  q = magic(labindex + 2); % q is the Composite object, indexed by labindes
  n = numlabs; % total number of workers
end

figure
for ii=1:length(q)
    subplot(1,length(q),ii), imagesc(q{ii});
end
```



Example 2

```
% Data Exchange among workers
spmd
    % Generate different data in each worker
    dataToSend = labindex*ones(labindex);

    % Sending data
    if labindex < numlabs
        labSend(dataToSend,labindex+1); % all workers but the last one
    else
        labSend(dataToSend,1); % last worker send to the 1st one
    end

    %Receiving data
    if labindex > 1
        dataReceived = labReceive(labindex-1); % all workers but the last one
    else
        dataReceived = labReceive(numlabs); % 1st worker receive from the last one
    end

    disp(dataReceived) % show data in each Worker
end
```

```
Lab 1:
     2     2
     2     2
```

```
Lab 2:
       1
```

Offloading executions: Batch processing

Submit batch job

You can Run a script or a function as a batch job by using the `batch` function. By default, `batch` uses your default cluster profile. In this example we use all the available workers but one (Npool-1), which is used to administer the others

```
delete(gcp('nocreate'));
clust = parcluster('local'); % create a cluster with the 'local' profile

Npool = 2; % 4 for a quad-core computer
M = 50; % number of rows and column
N = 10000; % number of trials
job = batch(clust,@ex_parallel, 1, {M,N}, 'Pool', Npool-1); % a = ex_parallel(M, N)
```

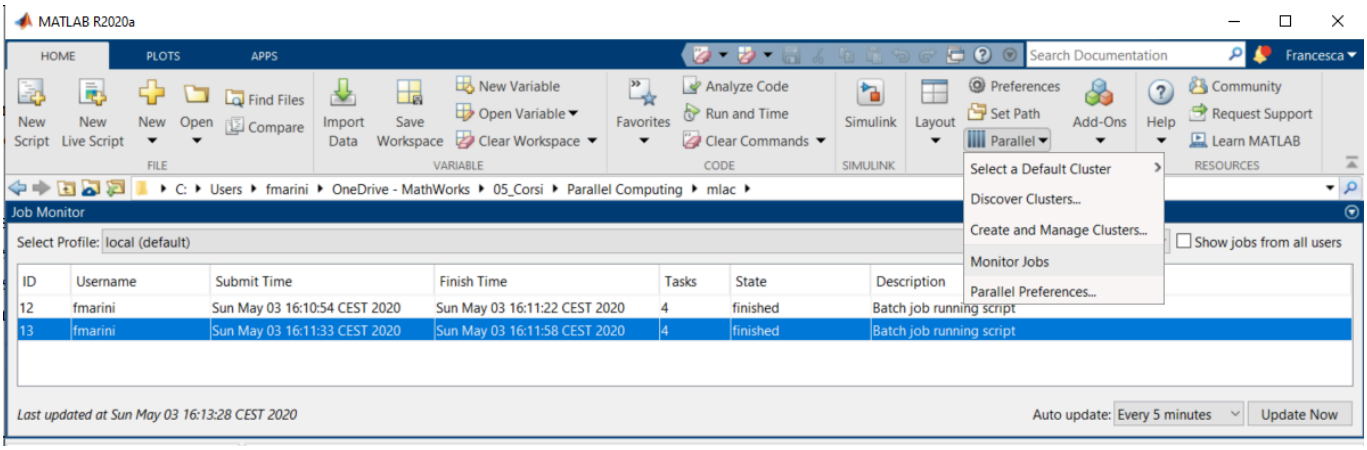
- `batch` runs the function `ex_parallel` on a worker in the cluster, returning a handle to the job object that runs the function
- The function is evaluated with the input arguments, `{M,N}`, and returns `1` output argument.
- With the Pool, Value pair it is possible to decide the dimension of the pool of worker executing the job
- The function file for `ex_parallel` is copied to the worker (do not include the `.m` file extension if you use a function or script)
- it is possible to assign different jobs to different workers calling `batch` multiple times, each one with a different function (`batch` will then assign the job to a free worker or put it in a queue if there aren't any)
- **contrary to `spmd` the job is not executed on multiple workers simultaneously but assigned to a specific one**

Monitor Jobs

```
get(job, 'State')
```

```
ans =
'running'
```

The Job Monitor displays the jobs in the queue for the scheduler determined by your selection of a cluster profile. Open the Job Monitor from the MATLAB® desktop on the **Home** tab in the **Environment** section, by selecting **Parallel** > **Monitor Jobs**

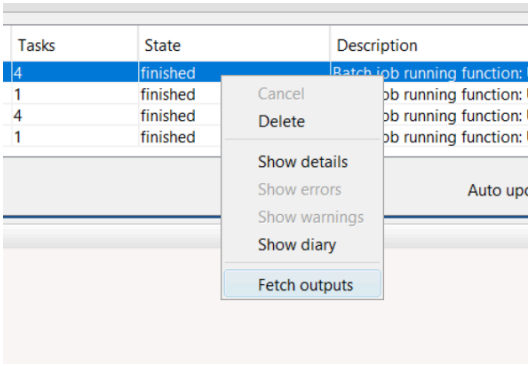


`batch` execution does not block MATLAB and you can continue working while computations take place. If you want to block MATLAB until the job finishes, use the `wait` function on the job object.

```
wait(job, 'finished') % Wait for job to be finished
```

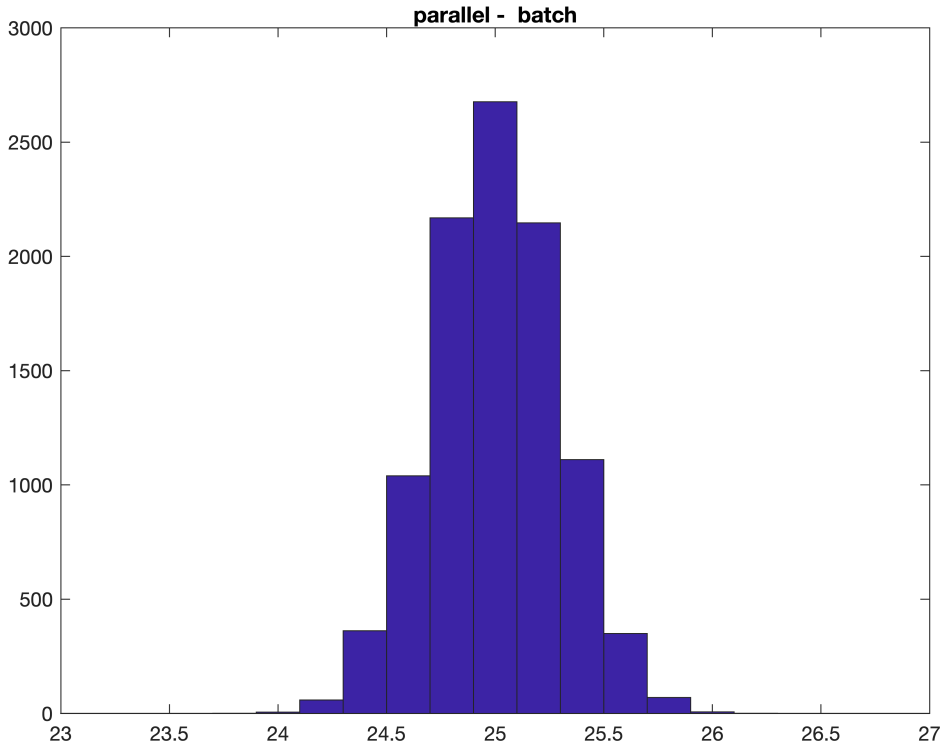
Retrieve and process results

You can use fetchOutputs when you use a function in your batch job, to retrieve the outputs into a a **cell array**, or use load when the job run a script, in order to retrieve the whole workspace




```
results = fetchOutputs(job);
a = results{1};

figure
hist(a, 23:0.2:27), xlim([23 27]), title('parallel -  batch')
```



Delete job when you don't need anymore

```
delete(job)
```

Example of Batch advanced control

By using `createJob` and `createTask` it is possible to split the code in chunks (task) and executing them in parallel (more tasks executed in each worker but still one job per worker). It is possible to check the intermediate results and retrieve the output of the whole computation at the end:

- `createJob(cluster)`: creates an independent job object for the identified cluster
- `t = createTask(j, F, N, {inputargs})`: creates a new task object in job `j`, and returns a reference, `t`, to the added task object. This task evaluates the function specified by a function handle or function name `F`, with the given input arguments `{inputargs}`, returning `N` output arguments

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

```
delete(gcp('nocreate'));
clust = parcluster('local'); % Use default profile

job = createJob(clust); % create a job for a worker
t = createTask(job, @rand, 1, {{10,10} {10,10} {10,10}}); % create multiple tasks to be
submit(job); % submit the job for the execution
wait(job, 'finished') % wait until the job is finished

results = fetchOutputs(job); % fetch and evaluate the output
```

```
for ii = 1:length(results)
    results{ii}
end
```

```
ans = 10×10
    0.1349    0.3414    0.0378    0.2873    0.6815    0.1700    0.6341    0.8666 ···
    0.6744    0.6596    0.1527    0.1777    0.8329    0.3007    0.9087    0.9242
    0.9301    0.9604    0.0199    0.4932    0.7620    0.8125    0.9334    0.4732
    0.5332    0.2081    0.7638    0.8810    0.3301    0.8027    0.9230    0.5052
    0.1150    0.0206    0.2389    0.3993    0.8738    0.4026    0.4597    0.4667
    0.6540    0.0097    0.7247    0.3138    0.4917    0.9944    0.2229    0.7484
    0.2621    0.4432    0.3819    0.3073    0.6435    0.7122    0.0043    0.2366
    0.9625    0.6220    0.1527    0.6538    0.5951    0.5486    0.6156    0.1400
    0.8972    0.9800    0.4316    0.3740    0.0846    0.9692    0.2890    0.7388
    0.3187    0.4841    0.8672    0.2539    0.1876    0.6113    0.0459    0.9253
ans = 10×10
    0.6383    0.6523    0.3167    0.4619    0.0667    0.7285    0.2120    0.7443 ···
    0.5195    0.9807    0.2194    0.3692    0.1896    0.1364    0.1990    0.4746
    0.1398    0.9821    0.9334    0.7850    0.7307    0.6309    0.7022    0.3114
    0.6509    0.3511    0.6939    0.6414    0.1504    0.4941    0.9530    0.6903
    0.3018    0.4784    0.7303    0.3279    0.1217    0.5902    0.5201    0.5605
    0.7101    0.8486    0.8608    0.3124    0.9460    0.5362    0.3814    0.8750
    0.4429    0.6760    0.2904    0.6072    0.3693    0.1661    0.3593    0.1518
    0.3972    0.8796    0.2788    0.2232    0.3236    0.9683    0.5298    0.1627
    0.7996    0.9114    0.3593    0.1642    0.9711    0.3347    0.1390    0.4014
    0.6656    0.8411    0.5984    0.8486    0.6938    0.0642    0.7508    0.6224
ans = 10×10
    0.9730    0.1454    0.7662    0.9601    0.0836    0.9843    0.2656    0.8780 ···
    0.7104    0.6426    0.9654    0.9145    0.6432    0.4296    0.4141    0.6507
    0.3614    0.3250    0.9843    0.8676    0.5964    0.1125    0.1687    0.2207
    0.2934    0.8229    0.9601    0.1388    0.8854    0.5494    0.1708    0.6235
    0.1558    0.8728    0.1856    0.2842    0.8472    0.5912    0.8205    0.3761
    0.3421    0.2005    0.9495    0.4687    0.8467    0.1968    0.7528    0.0899
    0.6071    0.9987    0.2639    0.0828    0.7554    0.7506    0.6331    0.0035
    0.5349    0.8446    0.1578    0.1702    0.6472    0.0087    0.8810    0.3619
    0.4118    0.9079    0.4784    0.5140    0.5163    0.5712    0.1639    0.7487
    0.1020    0.0982    0.2994    0.2809    0.4726    0.4986    0.3940    0.0096
```

Distributed Arrays

Example 1: Using matrix creation functions

```
delete(gcp('nocreate'));
parpool('local');
```

```
Starting parallel pool (parpool) using the 'local' profile ...
Connected to the parallel pool (number of workers: 2).
```

```
z = zeros(4,4,'distributed') % a parallel pool would be open if it is not open yet
```

```
z =

     0     0     0     0
     0     0     0     0
     0     0     0     0
     0     0     0     0
```

```
spmd
    disp(getLocalPart(z)) % returns the local portion of a codistributed array
```

```
end
```

```
Lab 1:
       0     0
       0     0
       0     0
       0     0

Lab 2:
       0     0
       0     0
       0     0
       0     0
```

Example 2: Distribute array (in the client) to workers

```
A = reshape(1:32,4,8); % create array on client
A = distributed(A) % distributed it on the workers (only some column to each worker)
```

```
A =

     1     5     9    13    17    21    25    29
     2     6    10    14    18    22    26    30
     3     7    11    15    19    23    27    31
     4     8    12    16    20    24    28    32
```

```
spmd
    disp(getLocalPart(A)) % returns the local portion of a codistributed array
end
```

```
Lab 1:
       1     5     9    13
       2     6    10    14
       3     7    11    15
       4     8    12    16

Lab 2:
      17    21    25    29
      18    22    26    30
      19    23    27    31
      20    24    28    32
```

Example 3: Codistributed arrays

```
% Create a distributed array from small pieces on different workers
spmd
    N = 1; % also try with larger N
    mat = repmat([1;2;3],1,N) + (labindex-1)*3; % create input array ([3xN] in each wor
```

Use default for dimensione and partition and specify a **distribution scheme** with 3 rows and numlabs*N columns

```
    codistr = codistributor1d( ...
        codistributor1d.unsetDimension, ...
        codistributor1d.unsetPartition, ...
```

```
        [3,numlabs*N]); % create distribution scheme
    distmat = codistributed.build(mat,codistr) % create the codistributed array [3x2N]
end
```

```
Lab 1:

  This worker stores distmat(:,1).

         LocalPart: [3x1 double]
     Codistributor: [1x1 codistributor1d]

Lab 2:

  This worker stores distmat(:,2).

         LocalPart: [3x1 double]
     Codistributor: [1x1 codistributor1d]
```

check all the vailable methods to work on distributed arrays by typimg method distributed on the console

```
% Compute sum on distributed array and collect data
distsum = sum(distmat);
numericsum = gather(distsum)
```

```
numericsum = 1×2
     6    15
```