LUND UNIVERSITY

Running MATLAB @NSC

ANDERS SJÖSTRÖM

Introduction

LUND UNIVERSITY

Why use MATLAB for HPC?

- Familiar environment
- Portability
- Rapid prototyping
- License conservation
- Computational speed
- Lots of available packages and code

LUND UNIVERSITY

MDCS introduction

- MATLAB Distributed Computing Server (MDCS) is built over the message parsing interface (MPI) to allow scaling of Parallel Computing Toolbox (PCT) enabled codes
- Workers need only the MDCS worker license (provided by SNIC)
- Workers inherit all toolbox licenses from the submitting process
- Regular MATLAB license + PCT license is only needed during job submission
- PCT constructs scale seamlessly. I.e. same code can be used locally and on the cluster
- Code can be prototyped on local machine and then used on a HPC cluster to scale in computational power and memory footprint

LUND UNIVERSITY

MATLAB@NSC

- Available using ThinLinc and through terminal
- Integrated with the queueing system
- Distributes work on workers in a parallel-pool
- Up to 700 workers
- Larger memory and more cores
- Arrays can be distributed over all workers
- Can run applications Interactively or as Batch jobs

Desktop or Console?

- For full MATLAB interface ThinLinc is recommended
- If only the ability to send in scripts is needed, the console can suffice
- The ThinLinc client is for most users preferable (available at www.cendio.com/thinlinc/download)

Setting up

Desktop

Command Window

New to MATLAB? See resources for Getting Started.

```
>> configCluster

                  >> % Set ProjectName and WallTime before submitting jobs to AURORA
                  >> c = parcluster;
                  >> c.AdditionalProperties.ProjectName = 'project-name';
                  >> c.AdditionalProperties.WallTime = '01:00:00';
                  >> c.saveProfile

fx >>
```

Desktop

```
Command Window
New to MATLAB? See resources for Getting Started.
>> configCluster

            >> % Set ProjectName and WallTime before submitting jobs to AURORA
            >> c = parcluster;
            >> c.AdditionalProperties.ProjectName = 'project-name';
            >> c.AdditionalProperties.WallTime = '01:00:00';
            >> c.saveProfile

>> c=parcluster;
>> c.AdditionalProperties.WallTime='00:10:00';
>> c.AdditionalProperties.ProjectName='aurora-test';
fx >>
```

Older Installations used:
configCluster
ClusterInfo.setProjectName('name-of-proj')
ClusterInfo.setWallTime('00:10:00')

Desktop

```
Command Window
New to MATLAB? See resources for Getting Started.
>> configCluster

                >> % Set ProjectName and WallTime before submitting jobs to AURORA
                >> c = parcluster;
                >> c.AdditionalProperties.ProjectName = 'project-name';
                >> c.AdditionalProperties.WallTime = '01:00:00';
                >> c.saveProfile

>> c=parcluster;
>> c.AdditionalProperties.WallTime='00:10:00';
>> c.AdditionalProperties.ProjectName='aurora-test';
>> !projinfo
(Counting the number of core hours used since 2017-10-06/00:00:00 until now.)

Project             Used[h]   Current allocation [h/month]
   User
-----------------------------------------------------
aurora-test           66.94              10000
   anfo               22.21
   hukebuck           15.13
   jhein               0.03
   marcosscc           0.37
   raymond            29.20

-----------------------------------------------------
snic2016-1-146         0.00               5000

-----------------------------------------------------
snic2017-1-126         0.09               5000
   hukebuck            0.09

-----------------------------------------------------
snic2017-4-9           0.00               5000

fx >> |
```

Desktop

```
Command Window
New to MATLAB? See resources for Getting Started.

                >> % Set ProjectName and WallTime before submitting jobs to AURORA
                >> c = parcluster;
                >> c.AdditionalProperties.ProjectName = 'project-name';
                >> c.AdditionalProperties.WallTime = '01:00:00';
                >> c.saveProfile

>> c=parcluster;
>> c.AdditionalProperties.WallTime='00:10:00';
>> c.AdditionalProperties.ProjectName='aurora-test';
>> !projinfo
(Counting the number of core hours used since 2017-10-06/00:00:00 until now.)

Project             Used[h]   Current allocation [h/month]
   User
-----------------------------------------------------
aurora-test           66.94            10000
   anfo               22.21
   hukebuck           15.13
   jhein               0.03
   marcosscc           0.37
   raymond            29.20

-----------------------------------------------------
snic2016-1-146         0.00             5000

-----------------------------------------------------
snic2017-1-126         0.09             5000
   hukebuck            0.09

-----------------------------------------------------
snic2017-4-9           0.00             5000

>> c.AdditionalProperties

ans =

  AdditionalProperties with properties:

     AdditionalSubmitArgs: ''
    DebugMessagesTurnedOn: 0
             EmailAddress: ''
                 MemUsage: ''
              ProcsPerNode: 0
              ProjectName: 'aurora-test'
                QueueName: ''
     RequestExclusiveNode: ''
              Reservation: ''
                   UseGpu: 0
          UseIdentityFile: 1
                 WallTime: '00:10:00'

fx >>
```

Older Installations used:

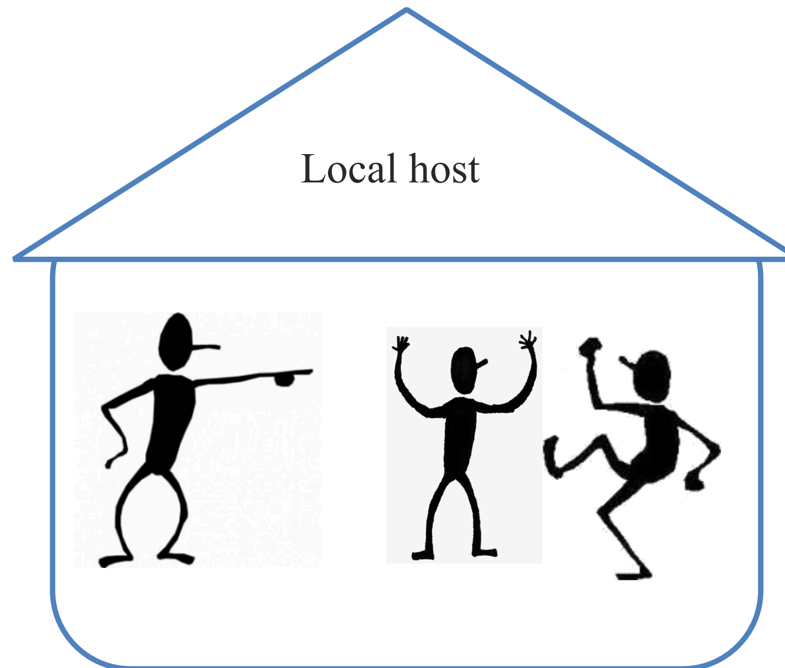
setArch
setClusterHost
setDataParallelism
setDebugMessagesTurnedOn
setDiskSpace
setEmailAddress
setGpusPerNode
setMemUsage
setNameSpace
setPrivateKeyFile
setPrivateKeyFileHasPassPhrase
setProcsPerNode
setProjectName
setQueueName
setRequireExclusiveNode
setReservation
setSshPort
setUseGpu
setUserDefinedOptions
setUserNameOnCluster
setWallTime

LUND UNIVERSITY

Basic concepts

Workflows

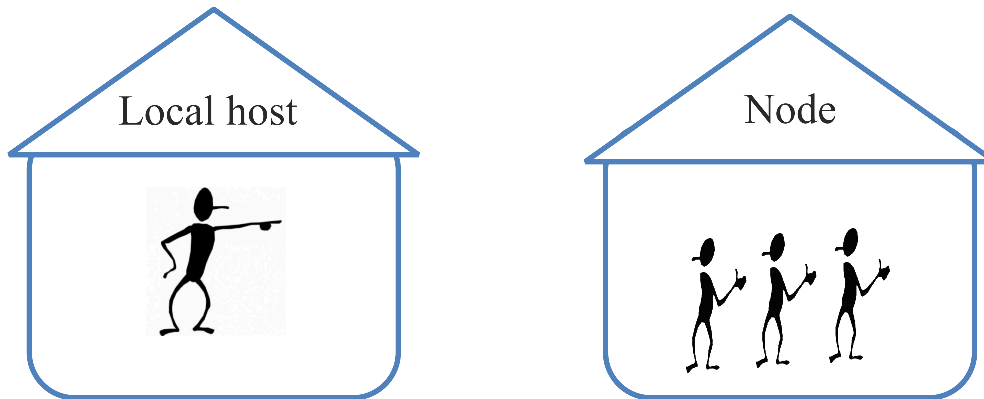
- Local



All workers are running on the local host

Workflows

- Interactive



All workers are running on a remote host, controlled by the user process more or less directly

LUND UNIVERSITY

Workflows

- Batch

Local host

Node

Node

Node

Node

Jobs are dispatched to workers on one or more nodes, each with its o
wn job id in SLURM. Note that each job has its own master worker

LUND UNIVERSITY

How do I start a parallel pool?

- parpool(n)
  - starts a parallel pool of n workers

- j = batch('script1','Pool',8)

  - starts a pool of 8 workers and runs the script script1 on those workers

LUND UNIVERSITY

How do I share code with workers?

- Workers Access Files Directly
- Pass Data to and from Worker Sessions
- Pass MATLAB Code for Startup and Finish

Workers Access Files Directly

- The workers all have access to the same drives on the network, they can access the necessary files that reside on these shared resources.
- Using the job's AdditionalPaths property
- Putting the path command in any of the appropriate startup files for the worker

'AdditionalPaths' — A string or cell array of strings that defines paths to be added to the MATLAB search path of the workers before the script or function executes.

```
matlabroot\toolbox\local\startup.m
matlabroot\toolbox\distcomp\user\jobStartup.m
matlabroot\toolbox\distcomp\user\taskStartup.m
```

LUND UNIVERSITY

Pass Data to and from Worker Sessions

- **InputArguments** — Contains the input data you specified when creating the task. This data gets passed into the function when the worker performs its evaluation.
- **OutputArguments** — Property of each task contains the results of the function's evaluation.

- **JobData** — Property of the job object. Contains data that gets sent to every worker that evaluates tasks for that job. The data is passed to a worker only once per job.
- **AttachedFiles** — Property of the job object. A cell array in which you manually specify all the folders and files that get sent to the workers. On the worker, the files are installed and the entries specified in the property are added to the search path of the worker session.
- **AutoAttachFiles** — Property of the job object. A logical value to specify that you want MATLAB to perform an analysis on the task functions in the job and on manually attached files to determine which code files are necessary for the workers, and to automatically send those files to the workers.

The supported code file formats for automatic attachment are MATLAB files (.m extension), P-code files (.p), and MEX-files (.mex). Note that AutoAttachFiles does not include data files for your job; use the AttachedFiles property to explicitly transfer these files to the workers.

LUND UNIVERSITY

Pass MATLAB Code for Startup and Finish

- As in a session of MATLAB, a worker session executes its startup.m file each time it starts. You can place the startup.m file in any folder on the worker's MATLAB search path, such as toolbox/distcomp/user.
- These additional files can initialize and clean up a worker session as it begins or completes evaluations of tasks for a job:

LUND UNIVERSITY

Pass MATLAB Code for Startup and Finish

- **jobStartup.m** automatically executes on a worker when the worker runs its first task of a job.
- **taskStartup.m** automatically executes on a worker each time the worker begins evaluation of a task.
- **poolStartup.m** automatically executes on a worker each time the worker is included in a newly started parallel pool.
- **taskFinish.m** automatically executes on a worker each time the worker completes evaluation of a task.

Empty versions of these files are provided in the folder:
matlabroot/toolbox/distcomp/user
You can edit these files to include whatever MATLAB code you want the worker to execute at the indicated times.
Alternatively, you can create your own versions of these files and pass them to the job as part of the AttachedFiles property, or include the path names to their locations in the AdditionalPaths property.
The worker gives precedence to the versions provided in the AttachedFiles property, then to those pointed to in the AdditionalPaths property. If any of these files is not included in these properties, the worker uses the version of the file in the toolbox/distcomp/user folder of the worker's MATLAB installation.

LUND UNIVERSITY

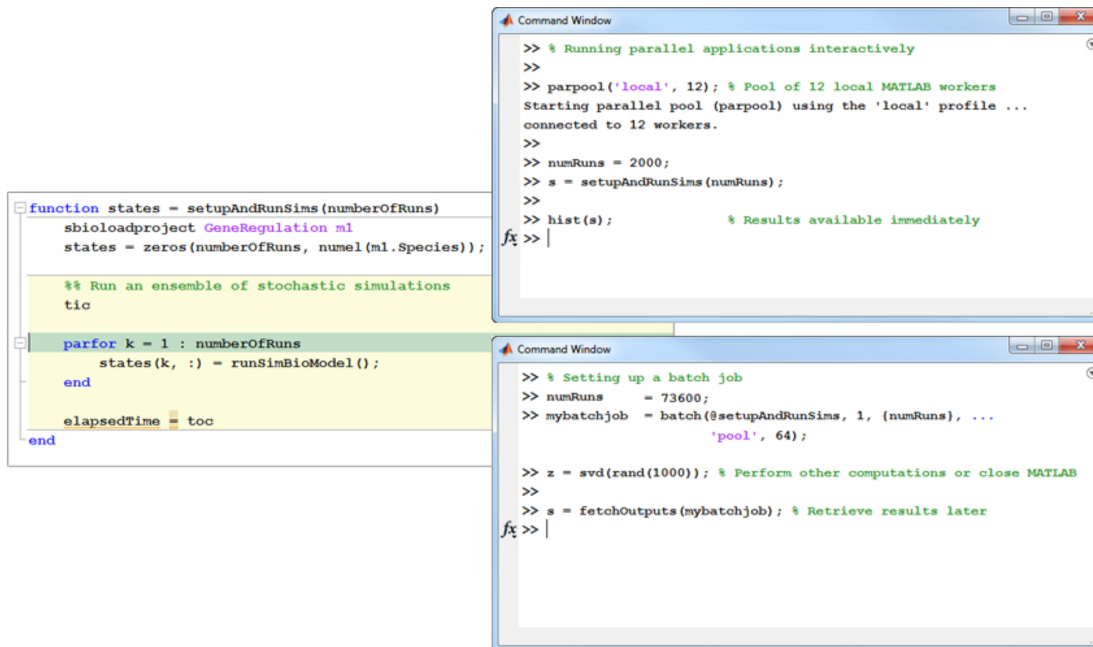
Slow code? Out of memory?

- Know your enemy:
  - Understand the constraints
  - Identify bottlenecks
- Exploit data and task parallelism
- Find the best tradeoff between programming effort and achieving your goals

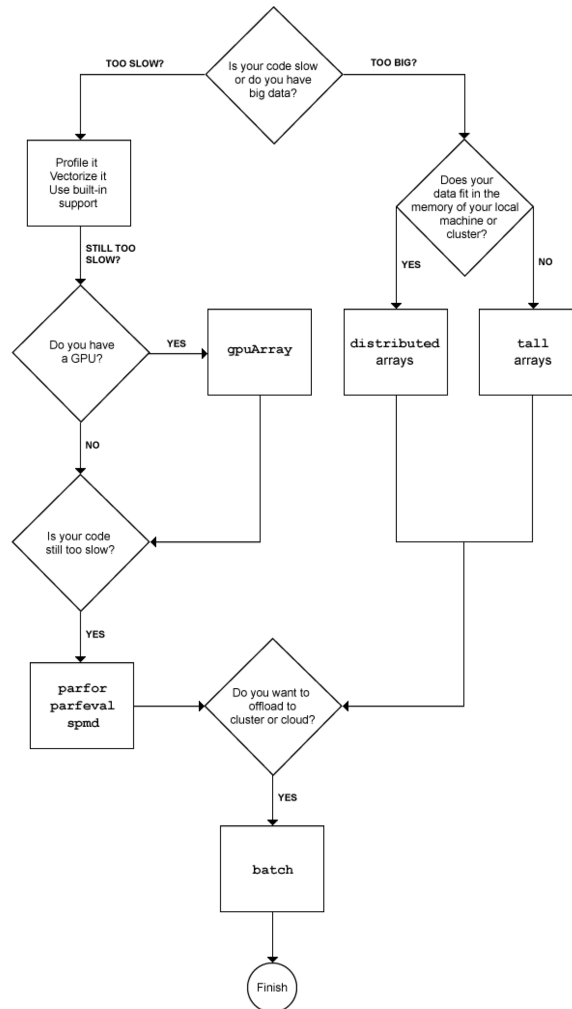
LUND UNIVERSITY

MATLAB@NSC

- Test your code locally on a small problem first
- Use MATLAB profiler to optimize code and spot parallelizable regions
- Avoid running interactively to the backend, use Batch instead
- Use MATLAB on the desktop as a launcher through the Batch-command



Speeding up MATLAB Flow-chart



Do you want to use a bigger/faster machine?

Yes

Use a cluster profile

Do you want to offload to the background?

Yes

Do you want one instance of your program to run faster?

No

Have you profiled it?

No

Improve it

Can the code be vectorized?

Yes

Do you have lots of intense independent work to run?

Yes

Do you want to be able to plot intermediate results?

Yes

Do you want to be able to break out of a loop early?

Yes

Use parfor

Use parfeval

Do you have a cluster and a data-parallel problem too large for the RAM of a single machine?

Yes

Use distributed arrays

Do you have a problem that can be parallelized using message passing?

Yes

Use spmd

Do you still need to speed up one instance of your application, and do you have a GPU and code that might run there?

Yes

Use gpuArray

Do you want to offload to the background?

Yes

Use batch

LUND UNIVERSITY

Parallelizing code

Interactively Run a Loop in Parallel

*

Create a sine waveform and plot it

Serial version

```
for i = 1:1024
  A(i) = sin(i*2*pi/1024);
end
plot(A)
```

Parallel version

```
parfor i = 1:1024
  A(i) = sin(i*2*pi/1024);
end
plot(A)
```

for - parfor

```
for loopvar = initval:endval
                <statements>
        END
```

```
parfor loopvar = initval:endval
                <statements>
        END
```

LUND UNIVERSITY

Parfor variables

- Loop Variable: Loop index
- Sliced Variables: Arrays whose segments are operated on by different iterations of the loop
- Broadcast Variables: Variables defined before the loop whose value is required inside the loop, but never assigned inside the loop
- Reduction Variables: Variables that accumulate a value across iterations of the loop, regardless of iteration order
- Temporary Variables: Variables created inside the loop, and not accessed outside the loop

```
a = 0;
c = pi;
z = 0;
r = rand(1,10);
parfor i = 1:10
    a = i;
    z = z+i;
    b(i) = r(i);
    if i <= c
        d = 2*a;
    end
end
```

What if my loop is nested?

- The body of a parfor-loop cannot contain another parfor-loop.
- A parfor-loop can call a function that contains another parfor-loop.
- A worker cannot open a parallel pool. Thus, a worker cannot run an inner nested parfor-loop in parallel.
- Only one level of nested parfor-loops can run in parallel.
- If the outer loop runs in parallel on a parallel pool, the inner loop runs serially on each worker.
- If the outer loop runs serially in the client, the function that contains the inner loop can run the inner loop in parallel on workers in a pool.
- The body of a parfor-loop can contain for-loops.
- You can use the inner loop variable for indexing the sliced array, but only in plain form, not part of an expression.

```
A = zeros(4,5);
parfor j = 1:4
    for k = 1:5
        A(j,k) = j + k;
    end
end
A
```

LUND UNIVERSITY

What if my loop is nested?

```
M1 = magic(10000);
M2=zeros(size(M1));
tic;
[j,k]=size(M1);
for x = 1:j
    for y = 1:k
        M2(x,y) = x*10 + y +   M1(x,y)/10000;
    end
end
t(1)=toc;
```

```
parfor x = 1:j
    for y = 1:k
        M2(x,y) = x*10 + y + M1(x,y)/10000;
    end
end
t(2)=toc;
```

0.3993 sec

```
for x = 1:j
    parfor y = 1:k
        M2(x,y) = x*10 + y + M1(x,y)/10000;
    end
end
t(3)=toc;
```

3.6440 sec

LUND UNIVERSITY

What if my loop is nested?

Invalid

```
A = zeros(100, 200);
parfor i = 1:size(A, 1)
   for j = 1:size(A, 2)
      A(i, j) = plus(i, j);
   end
end
```

Valid

```
A = zeros(100, 200);
n = size(A, 2);
parfor i = 1:size(A,1)
   for j = 1:n
      A(i, j) = plus(i, j);
   end
end
```

For proper variable classification, the range of a for-loop nested in a parfor must be defined by constant numbers or variables. In the example, the code on the left does not work because the for-loop upper limit is defined by a function call. The code on the right works around this by defining a broadcast or constant variable outside the parfor first.

LUND UNIVERSITY

What if my loop is nested?

Invalid

```
A = zeros(4, 11);
parfor i = 1:4
   for j = 1:10
      A(i, j + 1) = i + j;
   end
end
```

Valid

```
A = zeros(4, 11);
parfor i = 1:4
   for j = 2:11
      A(i, j) = i + j - 1;
   end
end
```

The index variable for the nested for-loop must never be explicitly assigned other than in its for-statement. When using the nested for-loop variable for indexing the sliced array, you must use the variable in plain form, not as part of an expression. In the example, the code on the left does not work, but the code on the right does:

LUND UNIVERSITY

What if my loop is nested?

Invalid

```
A = zeros(4, 10);
parfor i = 1:4
    for j = 1:10
        A(i, j) = i + j;
    end
    disp(A(i, 1))
end
```

Valid

```
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
```

If you use a nested for-loop to index into a sliced array, you cannot use that array elsewhere in the parfor-loop. In the example, the code on the left does not work because A is sliced and indexed inside the nested for-loop; the code on the right works because v is assigned to A outside the nested loop:

LUND UNIVERSITY

What if my loop is nested?

Invalid

```
A = zeros(4, 10);
parfor i = 1:4
    for j = 1:5
        A(i, j) = i + j;
    end
    for k = 6:10
        A(i, k) = pi;
    end
end
```

Valid

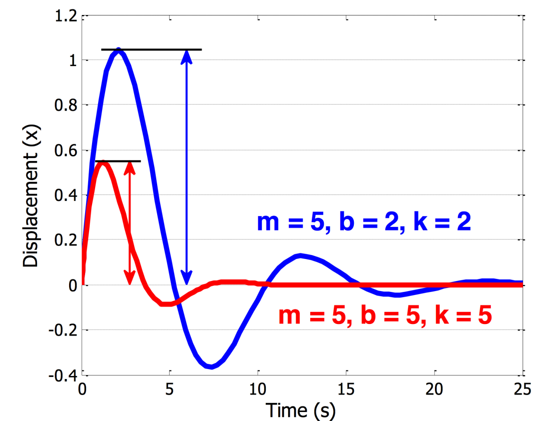
```
A = zeros(4, 10);
parfor i = 1:4
    for j = 1:10
        if j < 6
            A(i, j) = i + j;
        else
            A(i, j) = pi;
        end
    end
end
```

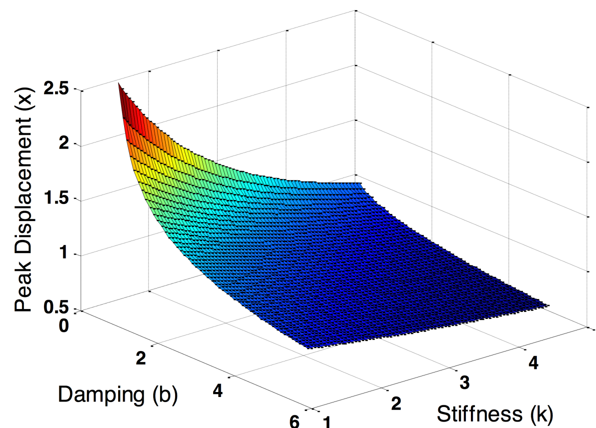
Inside a parfor, if you use multiple for-loops (not nested inside each other) to index into a single sliced array, they must loop over the same range of values. Furthermore, a sliced output variable can be used in only one nested for-loop. In the example, the code on the left does not work because j and k loop over different values; the code on the right works to index different portions of the sliced array A:

LUND UNIVERSITY

Example: parameter sweep

- Offload parameter sweep to local workers
- Get peak value results when processing is complete
- Plot results in local MATLAB





Parameter sweep speedup

time (s)

80
70
60
50
40
30
20
10

0 2 4 6 8 10 12 14 16

feval - parfeval

feval – evaluate function

```
fun = 'round';
x1 = pi;
y = feval(fun,x1)

>>y=3

x2 = 2;
y = feval(fun,x1,x2)

>>y=3.1400
```

parfeval - Execute function asynchronously on parallel pool worker

```
f = parfeval(p,@magic,1,10);
value = fetchOutputs(f);
```

Parfeval example

```
n = 10000000;
job = cell(1,6);
for idx = 1:6
        jobs(idx) = parfeval(pool, @test, 1, n, idx);
end

% wait for outputs as they finish
output = cell(1, 6);
for idx = 1:6
        [completedIdx, value] = fetchNext(jobs);
        output{completedIdx} = value;
end
delete(pool);
```

SPMD and Distributed Arrays

SPMD

LUND UNIVERSITY

Overview

- `spmd` (Single Program Multiple Data)
- `labindex` and `numlabs`
- Exchanging data between workers explicitly
- Data transfer to the client using composite arrays

parpool

- Similar to `parfor`, `spmd` requires a parpool in order for code to run on workers

- If a `parpool` doesn't exist, one will start if that is the default behavior

spmd

- Code inside `spmd` blocks run on all workers
- Unlike `parfor`, variables maintain state between calls to `spmd` as well as in `parfor`
- Can be used for loading data to be used in `parfor` loops

```
spmd
    % myfile.mat needs to be available on workers
    data = load('myfile.mat');
end

parfor I = 1:N
    % loop using data
end
```

labindex and numlabs

- Helps control what is executed on a worker
- Inside a `spmd` block
  - `labindex` returns the rank of the worker
  - `numlabs` returns the total number of workers in the pool

```
spmd
    switch labindex
        case 1
            % Code for worker 1
        case 2
            % Code for worker 2
        ...
    end
end
```

Create a Variant Array on Each of the Workers

```
>> magic_squares
```

```
spmd
    % Build magic squares in parallel
    m = magic(labindex + 2);
end

for ii=1:length(m)
    % Plot each magic square
    M = m{ii};
    figure, imagesc(M);
end
```

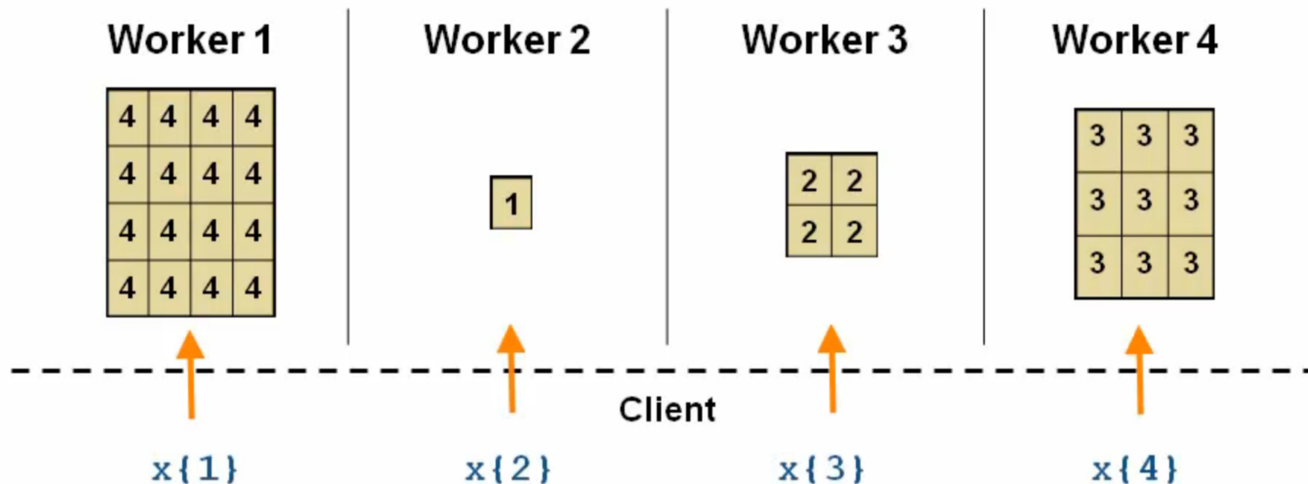
LUND UNIVERSITY

```
>> approx_pi
```

```matlab
quadpi = @(x) 4./(1 + x.^2);

spmd
    a = (labindex - 1)/numlabs;
    b = labindex/numlabs;
    fprintf('Subinterval: [%-4g, %-4g]\n', a, b);
end

spmd
    myIntegral = integral(quadpi, a, b);
    fprintf('Subinterval: [%-4g, %-4g]   Integral: %4g\n', ...
        a, b, myIntegral);
end

spmd
    piApprox = gplus(myIntegral);
end

approx1 = piApprox{1};   % 1st element holds value on worker 1.
fprintf('pi           : %.18f\n', pi);
fprintf('Approximation: %.18f\n', approx1);
fprintf('Error        : %g\n', abs(pi - approx1))
```

Composite Arrays

- Composite: client-side data-type for viewing data on the workers
- Outside of `spmd`, index with `()` or `{}` to get the data of one of the workers to the client



```
>> spmd, x = labindex/numlabs, end
Lab 1:
  x =
      0.5000
Lab 2:
  x =
       1
>> x
x =
   Lab 1: class = double, size = [1  1]
   Lab 2: class = double, size = [1  1]
>> y = x{1}
y =
    0.5000
>> whos x y
  Name       Size              Bytes  Class        Attributes

  x          1x2                 697  Composite
  y          1x1                   8  double
```

Types of Composite Arrays (non-distributed arrays)

- Replicated

```
spmd, x = numlabs, end
```

- Variant

```
spmd, x = labindex, end
```

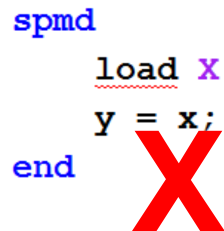
- Private

```
spmd, if labindex==1, x = rand, end, end
```

LUND UNIVERSITY

Limitations

- The body of an `spmd` statement must be transparent




```
spmd
    data = load(['X' num2str(labindex)]);
    y = data.x;
end
```

Distributed Arrays

LUND UNIVERSITY

Overview

- Distributed Arrays
- Constructing Distributed Arrays
- `distributed` and `codistributed`
- Working with Distributed Arrays

LUND UNIVERSITY

parpool

- Similar to `spmd`, distributed arrays require a `parpool` in order for code to run on workers

- If a `parpool` doesn't exist, one will start if that is the default behavior

Distributed Arrays

1	4	7	10
2	5	8	11
3	6	9	12

- One variable, split over multiple workers
- However, the MATLAB client sees the variable as one
- Mainly of interest with a cluster, combining the memory of multiple machines
- If the function has been overloaded for distributed arrays, there should be minimal changes to the code

LUND UNIVERSITY

Creating Distributed Arrays (1)

- Matrix creation functions have been overloaded for distributed arrays
  - `zeros(...,'distributed');`
  - `randn(...,'distributed');`

```
Z = zeros(4,4, 'distributed')
```

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

```
A = reshape(1:16, 4, 4)
A = distributed(A)
```

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

- If a variable has the same value on all of the workers, use `distributed` directly

LUND UNIVERSITY

Creating Distributed Arrays (2)

- Use case: creating a large matrix from multiple files or one large file would not fit into the memory of one computer

- Create data on each worker

- Combined into a distributed array using `codistributed.build` and `codistributed1d`

1	2	3	4
1	4	7	10
2	5	8	11
3	6	9	12

1	2	3	4
1	4	7	10
2	5	8	11
3	6	9	12

- Specify the size of the distributed array and optionally the partitioning

LUND UNIVERSITY

Working with Distributed Arrays

- A collection of MATLAB functions are overloaded for `distributed` arrays

- Overloaded functions can be called similar to other data types (e.g. numeric)

```
>> methods distributed
abs
acos
acosd
acosh
acot
acotd
acoth
acsc
acscd
acsch
all
and
angle
any
...
```

- Call `gather` to convert back to a numeric array

Using Distributed Arrays on Workers

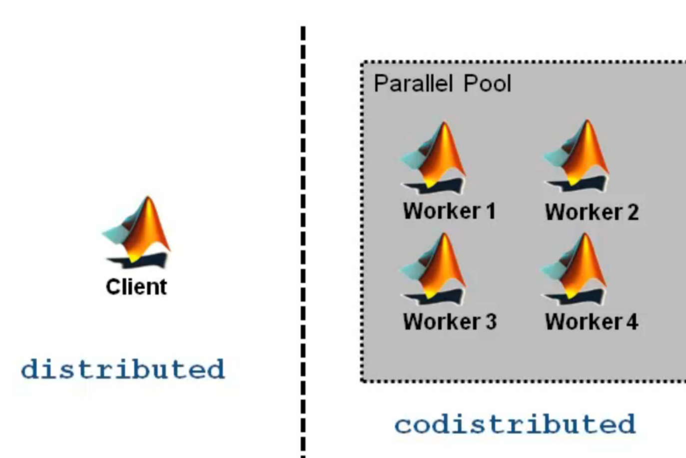
```
>> distrib_example
```

```
W = ones(8,'distributed');  % Distribute to the workers
spmd
    T = W*2; % Calculation performed on workers, in parallel.
             % T and W are both codistributed arrays here.
end
T            % View results in client.
whos         % T and W are both distributed arrays here.
```

LUND UNIVERSITY

distributed and codistributed

- The same distributed array will have a data type of:
  - `distributed`: on the client
  - `codistributed`: on the workers (within a `spmd` block)



Using Codistributed Arrays on Workers

```
>> codistrib_example
```

```
p = parpool(2);
spmd
    % Define 1-D distribution along the 3rd dimension
    % 4 parts on worker 1, and 12 parts on worker 2
    codist = codistributor1d(3,[4,12]);
    Z = zeros(3,3,16,codist);
    Z = Z + labindex;
end
Z % View results in client (distributed array here)
```

LUND UNIVERSITY

GPU-computing

What is needed?

- Matlab
- PCT
- GPU

Suitable problems

- Massively parallel tasks
- Computationally intensive tasks
- Tasks that have limited kernel size

LUND
UNIVERSITY

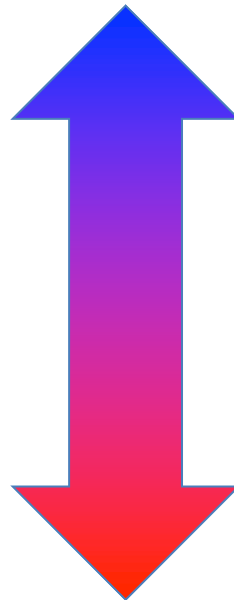
Options

- Built-in functions
- Functions on array data
- Directly invoke CUDA-code

LUND UNIVERSITY

Control vs Effort

Level of control		Required effort
Minimal		Built-in functions
Some		Functions on array data
Extensive		Directly invoke CUDA code



Built-in functions

- Accelerate standard (highly parallel) functions
  - More than 200 MATLAB functions are already supported

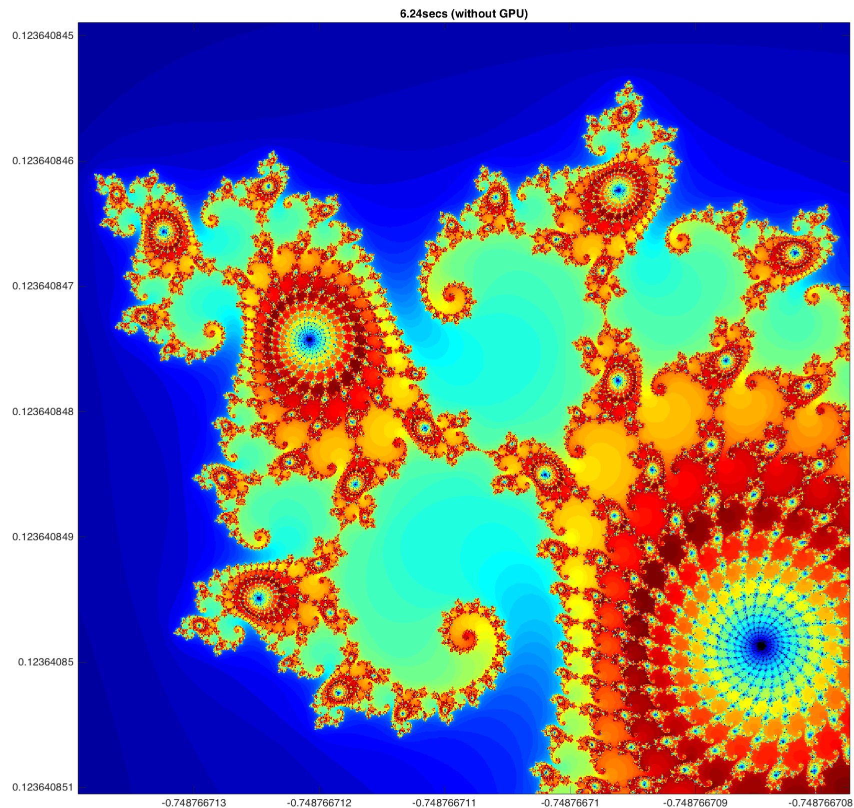
Random number generation	Min/max
FFT	SVD
Matrix multiplications	Cholesky and LU factorization
Solvers	
Convolutions	

- Out of the box:
  - No additional effort for programming the GPU
- No accuracy for speed trade-off
  - Double floating-point precision computations

LUND UNIVERSITY

Example

```
maxIterations = 500;
gridSize=1000;
xlim = [-0.748766713922161, -0.748766707771757];
ylim = [ 0.123640844894862,  0.123640851045266];

t = tic();
x = linspace( xlim(1), xlim(2), gridSize );
y = linspace( ylim(1), ylim(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

% show
count = log( count );
```



Example cont’d

Built-in functions

```
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 );
count = ones( size(z0), 'gpuArray' );

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

% Show
count = gather( count ); % Fetch the data back from the GPU
naiveGPUTime = toc( t );
```

LUND UNIVERSITY

Example cont'd

Functions on array data

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

% Calculate
count = arrayfun( @pctdemo_processMandelbrotElement, ...
                  xGrid, yGrid, maxIterations );

% Show
count = gather( count ); % Fetch the data back from the GPU
gpuArrayfunTime = toc( t );
```

```
function count = pctdemo_processMandelbrotElement(x0,y0,maxIterations)
z0 = complex(x0,y0);
z = z0;
count = 1;
while (count <= maxIterations) && (abs(z) <= 2)
    count = count + 1;
    z = z*z + z0;
end
count = log(count);
```

LUND UNIVERSITY

Example cont’d

```
% Load the kernel

cudaFilename = 'pctdemo_processMandelbrotElement.cu';

ptxFilename = ['pctdemo_processMandelbrotElement.',parallel.gpu.ptxext];

kernel = parallel.gpu.CUDAKernel( ptxFilename, cudaFilename );


% Setup

t = tic();

x = gpuArray.linspace( xlim(1), xlim(2), gridSize );

y = gpuArray.linspace( ylim(1), ylim(2), gridSize );

[xGrid,yGrid] = meshgrid( x, y );


% Make sure we have sufficient blocks to cover all of the locations

numElements = numel( xGrid );

kernel.ThreadBlockSize = [kernel.MaxThreadsPerBlock,1,1];

kernel.GridSize = [ceil(numElements/kernel.MaxThreadsPerBlock),1];


% Call the kernel

count = zeros( size(xGrid), 'gpuArray' );

count = feval( kernel, count, xGrid, yGrid, maxIterations, numElements );
```

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-19856038
// Cuda compilation tools, release 7.5, V7.5.17
// Based on LLVM 3.4svn
//

.version 4.3
.target sm_20
.address_size 64

        // .globl       _Z12doIterationsddj
```

```
__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;
}
```

Nvcc –ptx code.cu

```
        @%p1 bra        BB0_3;

        add.s32         %r6, %r6, 1;
        sub.f64         %fd10, %fd4, %fd3;
        add.f64         %fd5, %fd10, %fd7;
        add.f64         %fd11, %fd2, %fd2;
        fma.rn.f64      %fd6, %fd11, %fd1, %fd8;
        setp.le.u32     %p2, %r6, %r4;
        mov.f64         %fd12, %fd6;
        mov.f64         %fd13, %fd5;
        @%p2 bra        BB0_1;

BB0_3:
        st.param.b32    [func_retval0+0], %r6;
        ret;
}
```

LUND UNIVERSITY

Example cont'd

ans: 'finished'

count: [1000x1000 double]

cpuTime: 15.9700

gpuArrayfunTime: 0.7010

gridSize: 1000

inside: [1x1 gpuArray]

maxIterations: 500

n: 500

naiveGPUTime: 5.5109

t: 1443545040634310

x: [1x1 gpuArray]

xGrid: [1x1 gpuArray]

xlim: [-0.7488 -0.7488]

y: [1x1 gpuArray]

yGrid: [1x1 gpuArray]

ylim: [0.1236 0.1236]

z: [1x1 gpuArray]

z0: [1x1 gpuArray]

Laptop 6.24s

LUND UNIVERSITY

Simple example

Matlab

```
k = parallel.gpu.CUDAKerne('test.ptx','test.cu');
result = feval(k,2,3)
```

Kernel (test.cu)

```
__global__ void add1( double * pi, double c )
{
    *pi += c;
}
```

```
//
// Generated by NVIDIA NVVM Compiler
//
// Compiler Build ID: CL-19856038
// Cuda compilation tools, release 7.5, V7.5.17
// Based on LLVM 3.4svn
//

.version 4.3
.target sm_20
.address_size 64

                // .globl       _Z4add1Pdd

.visible .entry _Z4add1Pdd(
                .param .u64 _Z4add1Pdd_param_0,
                .param .f64 _Z4add1Pdd_param_1
)
{
                .reg .f64       %fd<4>;
                .reg .b64       %rd<3>;


                ld.param.u64     %rd1, [_Z4add1Pdd_param_0];
                ld.param.f64     %fd1, [_Z4add1Pdd_param_1];
                cvta.to.global.u64              %rd2, %rd1;
                ldu.global.f64   %fd2, [%rd2];
                add.f64          %fd3, %fd2, %fd1;
                st.global.f64    [%rd2], %fd3;
                ret;
}
```

LUND UNIVERSITY

Multiple GPUs

```
parpool(2)


spmd
gd=gpuDevice;
idx=gd.Index;
disp(['Using GPU ',num2str(idx)]);
end


parfor ix = 1:10
  gd=gpuDevice;
  d(ix)=gd.Index;
end
```

```
Lab 1:
  Using GPU 1
Lab 2:
  Using GPU 2

>> d

d =

     2     2     2     2
1     1     1     2     2
1
```

LUND
UNIVERSITY