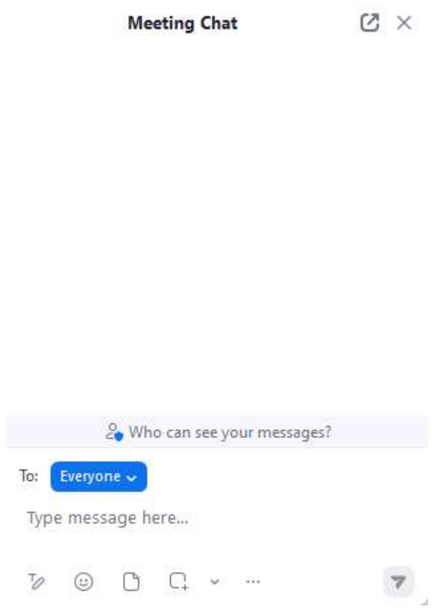
WORKSHOP: Parallel Computing With MATLAB (Part I)

Raymond Norris
Application Engineer
November 2023

Meeting Chat

- Please send chats to Everyone



Agenda

- Part I – Parallel Computing with MATLAB on the Desktop
  - Parallel Computing Toolbox
  - MATLAB Online
- Part II – Scaling MATLAB to Karolina
  - MATLAB Parallel Server
  - Karolina OnDemand

Agenda

- Part I – Parallel Computing with MATLAB on the Desktop
  - Parallel Computing Toolbox
  - MATLAB Online
- Part II – Scaling MATLAB to Karolina
  - MATLAB Parallel Server
  - Karolina OnDemand

Why use parallel computing?

Save time with parallel computing by carrying out computationally and data-intensive problems in parallel (simultaneously)

- distribute your tasks to be executed in parallel
- distribute your data to solve big data problems

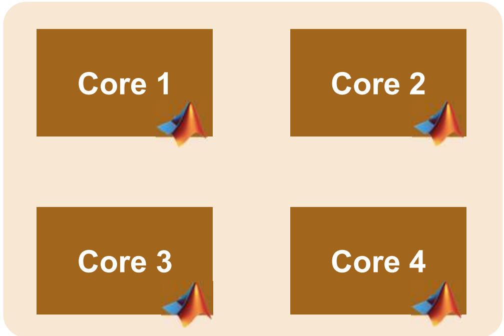
on your compute cores and GPUs or scaled up to clusters and cloud computing

Why use parallel computing with MATLAB?

Save time with parallel computing by carrying out computationally and data-intensive problems in parallel (simultaneously)

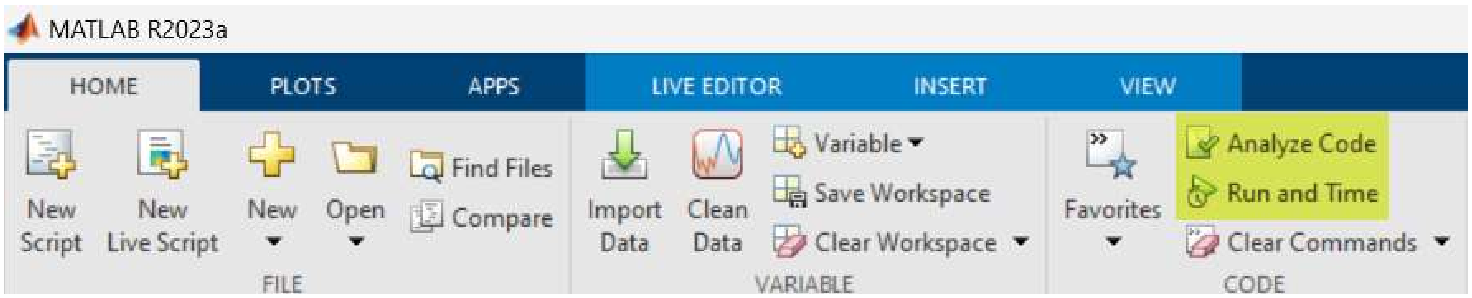
- distribute your tasks to be executed in parallel
- distribute your data to solve big data problems

on your compute cores and GPUs or scaled up to clusters and cloud computing with minimal code changes, so you can focus on your research



CPU with 4 cores

Before going parallel, optimize your code for the best performance



Before going parallel, optimize your code for the best performance

- Use the **Profiler** to find the code that runs slowest and determine possible performance improvements

```
rng(1)
x = rand(1,1e6);
for k = 1:numel(x)
    if x(k)<.5
        x(k) = 0;
    end
end
```

Time	Calls	Line	
0.035	1	1	rng(1)
0.012	1	2	x = rand(1,1e6);
< 0.001	1	3	for k = 1:numel(x)
0.061	1000000	4	if x(k)<.5
0.026	499837	5	x(k) = 0;
0.048	1000000	6	end
0.049	1000000	7	end

Use vectorization (matrix and vector operations) instead of for-loops

Time	Calls	Line	
0.007	1	1	rng(1)
0.011	1	2	x = rand(1,1e6);
0.007	1	3	x(x<.5) = 0;

Before going parallel, optimize your code for the best performance

- Use the **Code Analyzer** to automatically check your code for coding (and performance) problems

```
1 tic
2 x = 0;
3 for k = 2:1e6
4     x(k) = x(k-1) + 1;
5 end
6 toc
```

Line 4: Variable appears to change size on every loop iteration (within a script). Consider preallocating for speed.

Details ▼

Elapsed time is 0.075824 seconds.

```
1 tic
2 x = zeros(1,1e6);
3 for k = 2:1e6
4     x(k) = x(k-1) + 1;
5 end
6 toc
```

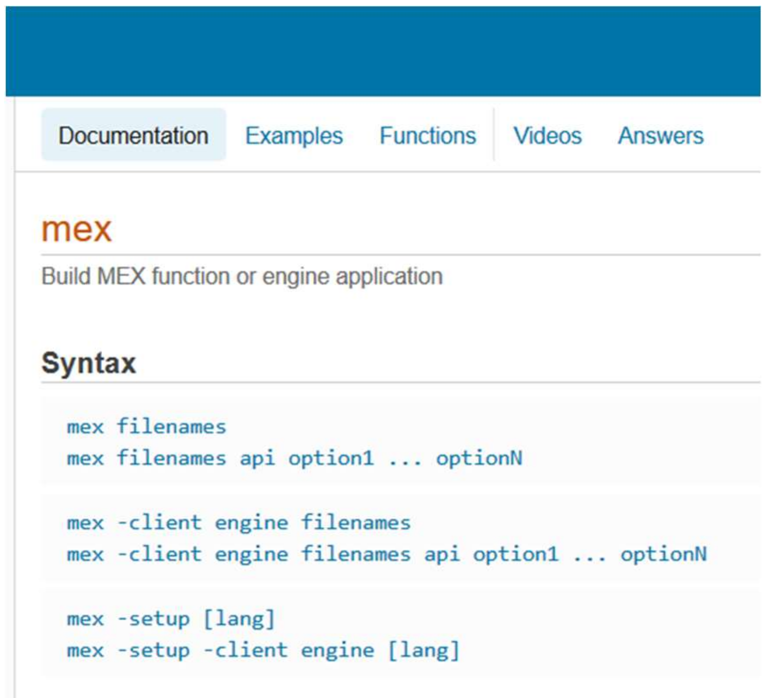
Preallocate the maximum amount of space required for the array instead of letting MATLAB repeatedly reallocate memory for the growing array

Elapsed time is 0.013109 seconds.

MathWorks

Before going parallel, optimize your code for the best performance

- Replace code with MEX functions



Documentation | Examples | Functions | Videos | Answers

mex

Build MEX function or engine application

Syntax

```
mex filenames
mex filenames api option1 ... optionN

mex -client engine filenames
mex -client engine filenames api option1 ... optionN

mex -setup [lang]
mex -setup -client engine [lang]
```

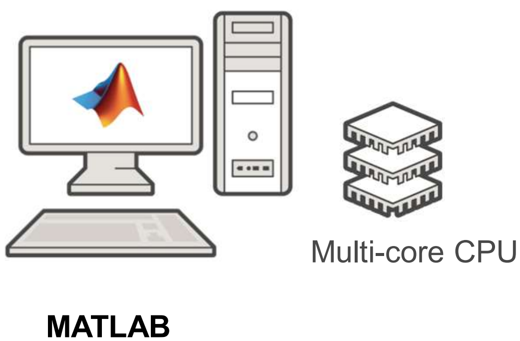
[Techniques for accelerating MATLAB algorithms and applications]

Before going parallel, optimize your code for the best performance with efficient programming practices

- **Pre-allocate** memory instead of letting arrays be resized dynamically
- **Vectorize** – Use matrix and vector operations instead of for-loops
- Try using functions instead of scripts. Functions are generally faster
- Create a new variable rather than assigning data of a different type to an existing variable
- Place independent operations outside loops to avoid redundant computations
- Avoid printing too much data on the screen, reuse existing graphics handles

Techniques to improve performance

MATLAB has built-in multithreading



MathWorks

MATLAB Multicore

Run MATLAB on multicore and multiprocessor machines

MATLAB® provides two main ways to take advantage of multicore and multiprocessor computers. By using the full computational power of your machine, you can run your MATLAB applications faster and more efficiently.

Built-in Multithreading

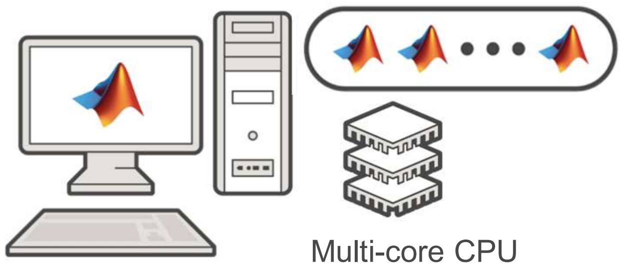
Linear algebra and numerical functions such as `fft`, `\` (`mldivide`), `eig`, `svd`, and `sort` are multithreaded in MATLAB. Multithreaded computations have been on by default in MATLAB since Release 2008a. These functions automatically execute on multiple computational threads in a single MATLAB session, allowing them to execute faster on multicore-enabled machines. Additionally, many functions in Image Processing Toolbox™ are multithreaded.

Parallelism Using MATLAB Workers

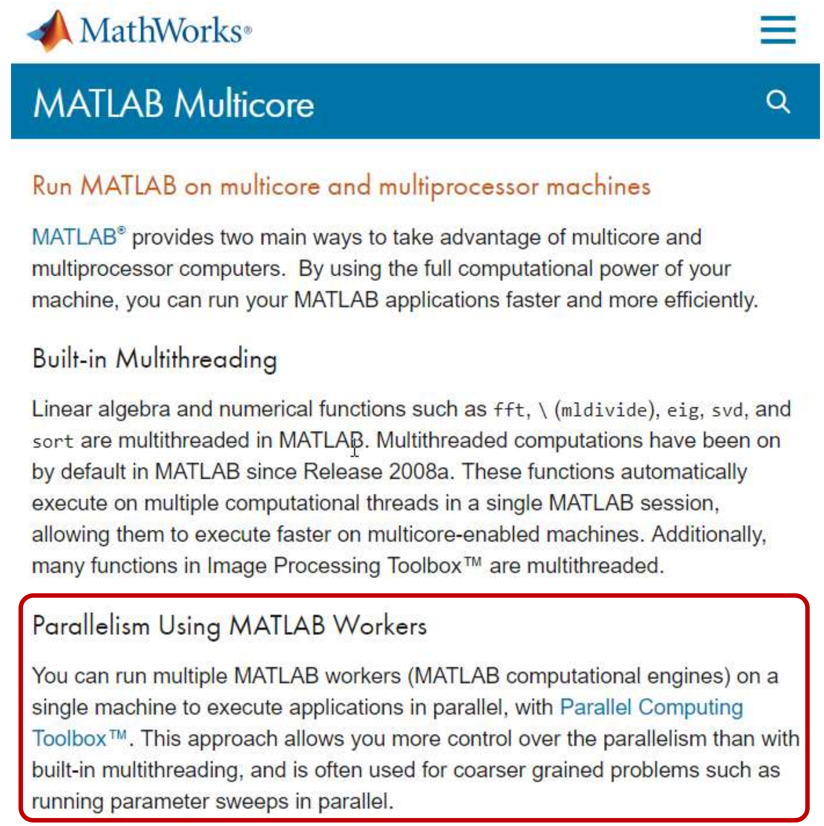
You can run multiple MATLAB workers (MATLAB computational engines) on a single machine to execute applications in parallel, with Parallel Computing Toolbox™. This approach allows you more control over the parallelism than with built-in multithreading, and is often used for coarser grained problems such as running parameter sweeps in parallel.

MATLAB multicore

Scale further with parallel computing



MATLAB
Parallel Computing Toolbox


MathWorks®

MATLAB Multicore

Run MATLAB on multicore and multiprocessor machines

MATLAB® provides two main ways to take advantage of multicore and multiprocessor computers. By using the full computational power of your machine, you can run your MATLAB applications faster and more efficiently.

Built-in Multithreading

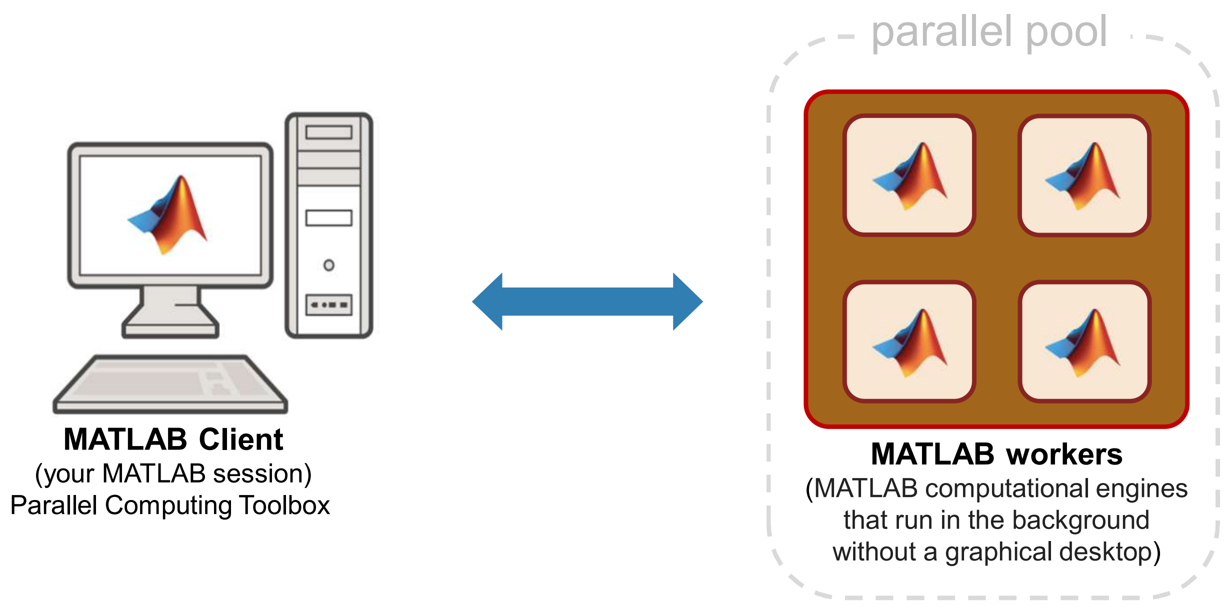
Linear algebra and numerical functions such as `fft`, `\` (`mldivide`), `eig`, `svd`, and `sort` are multithreaded in MATLAB. Multithreaded computations have been on by default in MATLAB since Release 2008a. These functions automatically execute on multiple computational threads in a single MATLAB session, allowing them to execute faster on multicore-enabled machines. Additionally, many functions in Image Processing Toolbox™ are multithreaded.

Parallelism Using MATLAB Workers

You can run multiple MATLAB workers (MATLAB computational engines) on a single machine to execute applications in parallel, with Parallel Computing Toolbox™. This approach allows you more control over the parallelism than with built-in multithreading, and is often used for coarser grained problems such as running parameter sweeps in parallel.

MATLAB multicore

Run parallel code by utilizing multiple CPU cores



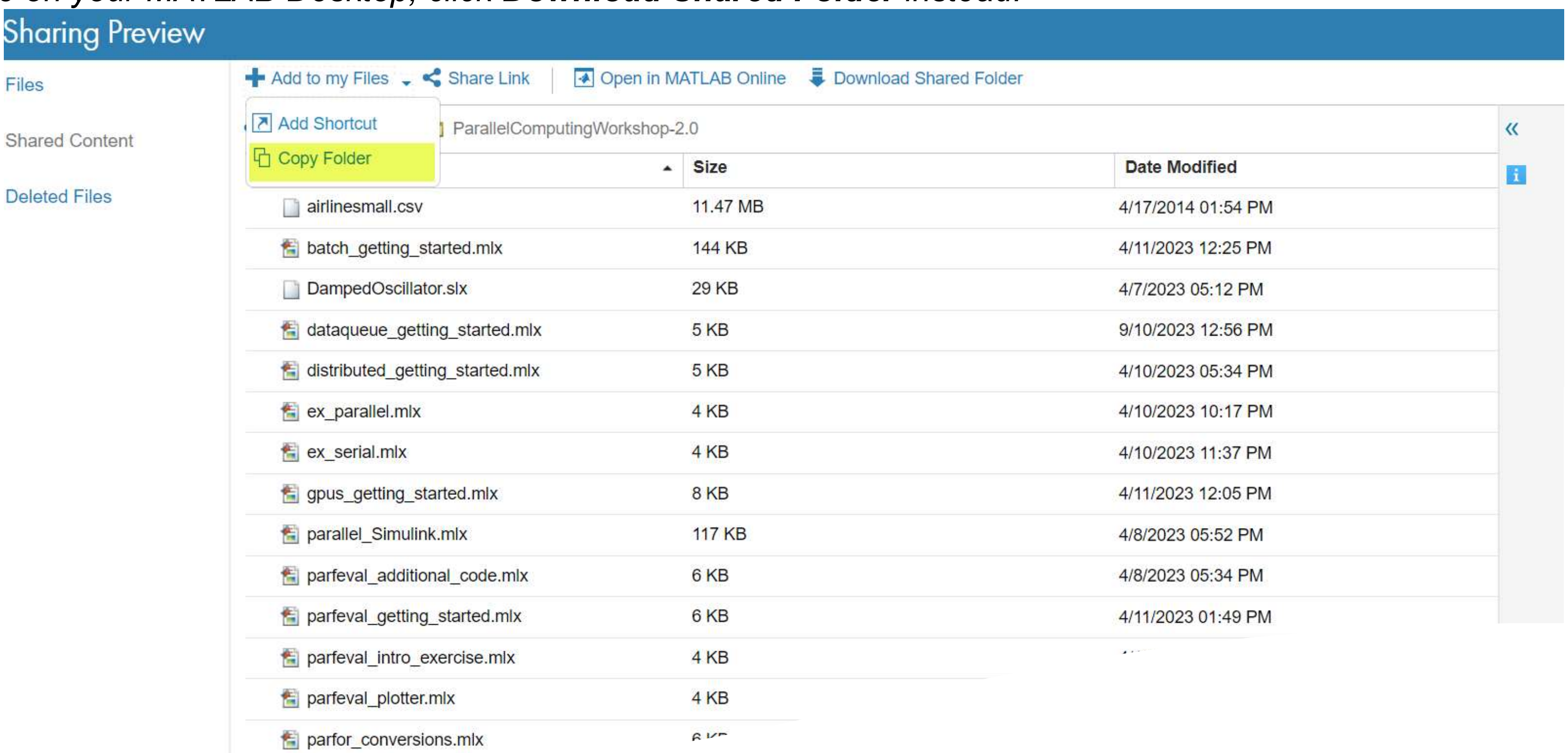
Download Instructions

- https://tinyurl.com/ParallelComputingWorkshop
  - Click on **Add to my Files**
  - Click **Copy Folder**

- https://www.mathworks.com/licensecenter/classroom/4234856
- Click **Access MATLAB Online** (maybe prompted to sign-in again)
  - Click **Open MATLAB Online**
  - In Current Folder, double click on **ParallelComputingWorkshop-2.0**

Setup: Step 1 – Copy materials via MATLAB Drive

Click **Add to my Files** and then click **Copy Folder**.
For use on your MATLAB Desktop, click ***Download Shared Folder*** *instead.*



Setup: Step 2 – Launch MATLAB Online

MathWorks®

MATLAB & Simulink

Access MATLAB for your Parallel Computing Workshop

MathWorks is pleased to provide a special license to you as a course participant to use for your Parallel Computing Workshop. This is a limited license for the duration of your course and is intended to be used only for course work and not for government, research, commercial, or other organization use.

Course Name:	Parallel Computing with MATLAB at IT4Innovations
Organization:	MathWorks Parallel Computing
Ending:	08 Nov 2023

Access MATLAB Online

https://www.mathworks.com/licensecenter/classroom/4152200

Hands-On Exercise: Starting a parallel pool

parpool Introduction

The purpose of this excercise is to learn what is a parallel pool and the different ways to start and shut it down.

To get started, run the following command:

```
doc parpool
```

Start a parallel pool programmatically

Start an interactive parallel pool of 2 workers through the command line.

```
% TODO: Start an interactive parallel pool of 2 workers programatically
```

What happens if you try to run the above command a second time?

```
% TODO: Rerun the section to start another pool
```

Only one interactive parallel pool can be open at a time.

parpool_intro

Scaling MATLAB applications and Simulink simulations

Ease of Use

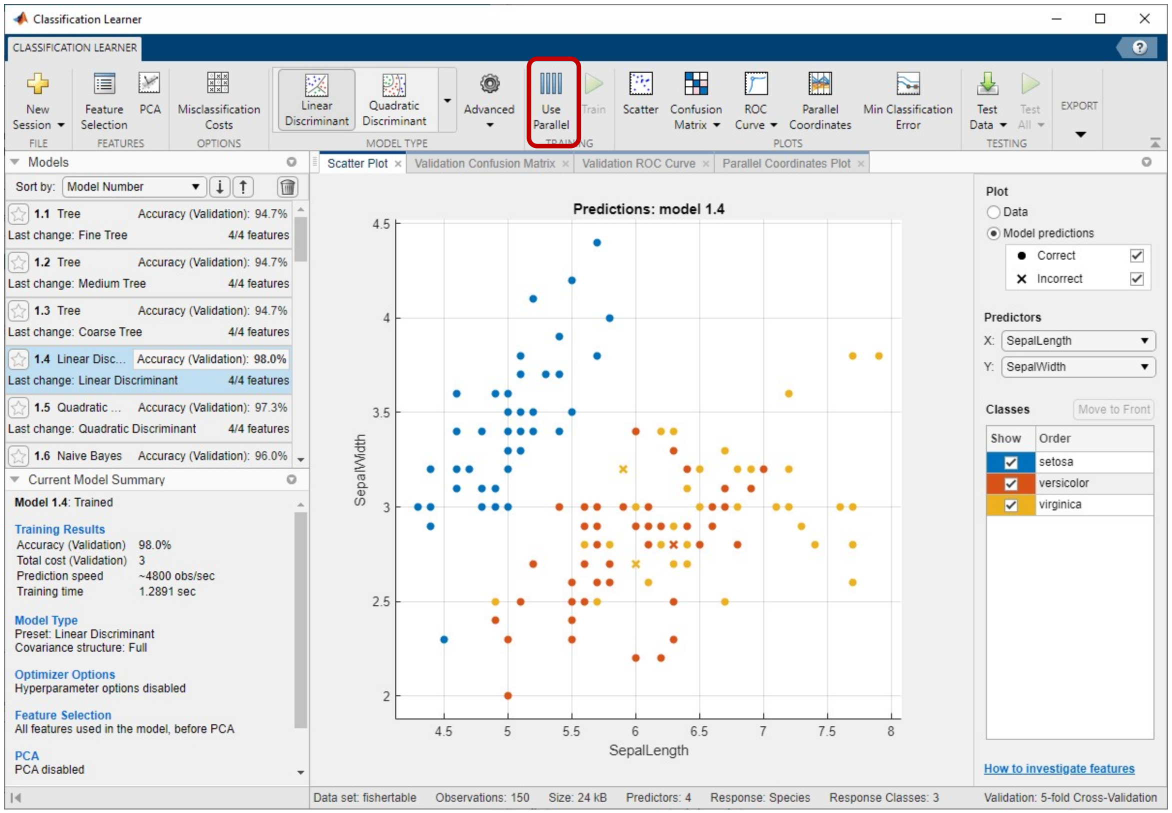
Automatic parallel support in toolboxes

Common programming constructs

Advanced programming constructs

Greater Control

Automatic parallel support in toolboxes



Scaling MATLAB applications and Simulink simulations

Ease of Use

Automatic parallel support in toolboxes

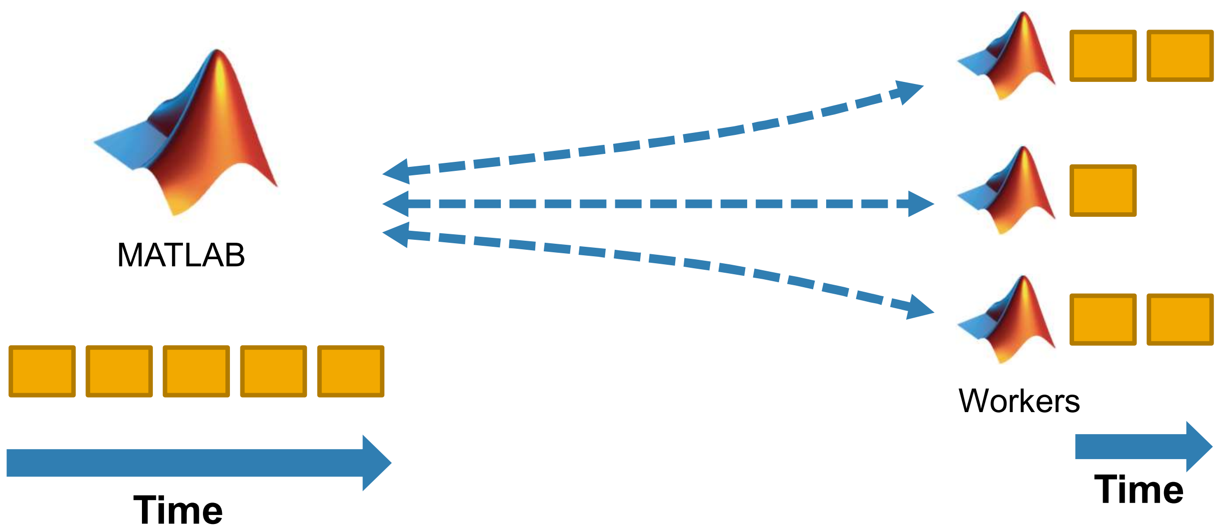
Common programming constructs
(`parfor`, `parfeval`, `parsim`, …)

Advanced programming constructs

Greater Control

Explicit parallelism using `parfor` (parallel `for`-loop)

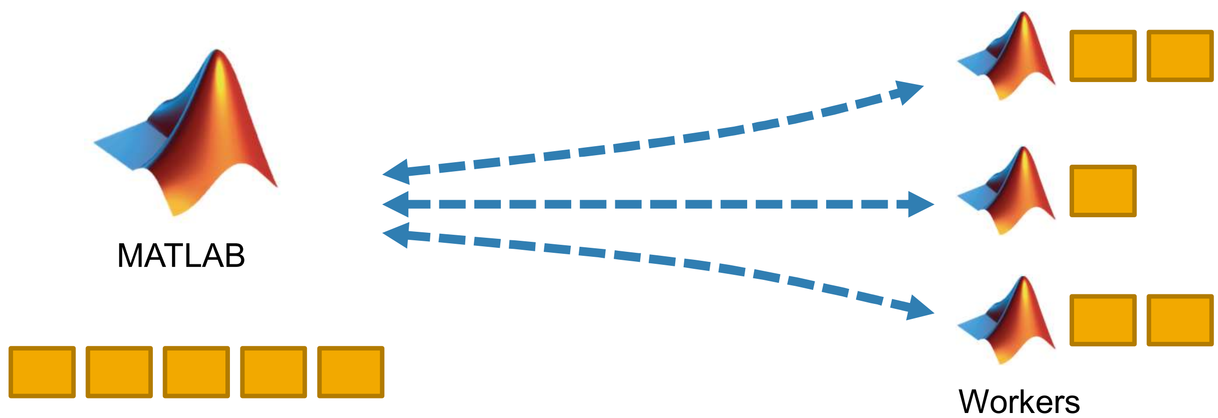
- Run iterations in parallel
- Examples: parameter sweeps, Monte Carlo simulations



Explicit parallelism using `parfor`

```
a = zeros(5, 1);
b = pi;
for i = 1:5
   a(i) = i + b;
end
disp(a)
```

```
a = zeros(5, 1);
b = pi;
parfor i = 1:5
   a(i) = i + b;
end
disp(a)
```



Hands-On Exercise: Writing our first `parfor`

Getting Started with `parfor`

In this exercise, we will rewrite a simple `for`-loop into a `parfor`-loop and understand the basic differences between the two types of loops.

```
doc parfor
```

How much time will the following `for`-loop take?

This is an example of a basic `for`-loop; in each iteration, `pause(1)` stops MATLAB execution for one second and then displays the index `idx` of the iteration. Since there are 10 iterations, the `for`-loop takes about 10 seconds (the added `tic` and `toc` measure the time elapsed) and the indices are displayed sequentially, from 1 to 10.

```
tic
for idx = 1:10
    pause(1)
    disp(idx)
end
toc
```

`parfor_getting_started`

DataQueue: Execute code as parfor iterations complete

- Send data or messages from parallel workers back to the MATLAB client
- Retrieve intermediate values and track computation progress

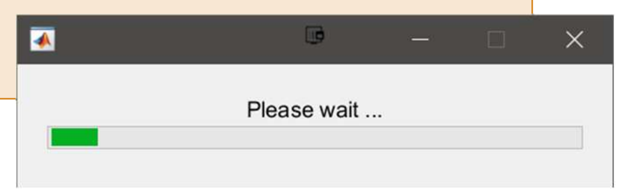
```
function a = parforWaitbar

D = parallel.pool.DataQueue;
h = waitbar(0, 'Please wait ...');
afterEach(D, @nUpdateWaitbar)

N = 200;
p = 1;

parfor i = 1:N
    a(i) = max(abs(eig(rand(400))));
    send(D, i)
end

    function nUpdateWaitbar(~)
        waitbar(p/N, h)
        p = p + 1;
    end
end
```



Hands-On Exercise: Sending data with a `dataqueue`

Getting Started with dataqueue

In this exercise, we will use a parallel data queue to send data from the workers to the MATLAB client.

```
doc dataqueue
```

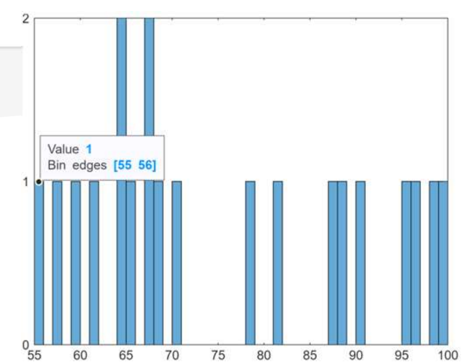
Give the quiz

In this problem, we are giving a quiz to 20 students. Each student will finish in a random period of time (max 5 minutes) and each student will get a random grade between 55 and 100.

We need to gather all of the scores to plot the results, but we also want to write the results as they happen in real-time to a "database" (the MATLAB Command Window in this case).

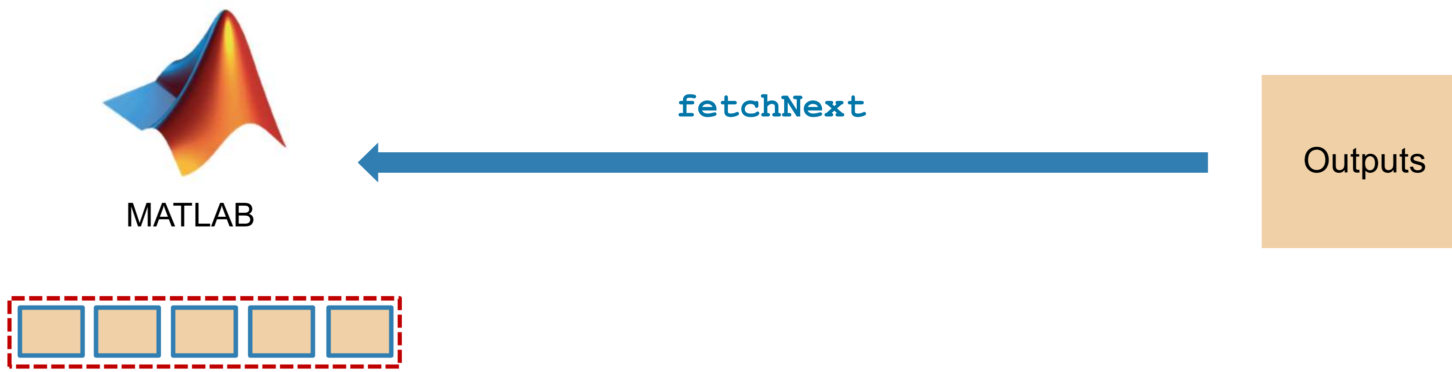
The for-loop will gather all the scores to eventually be plotted in a histogram, but MATLAB should also be listening to "events" (when the grade get posted) in the dataqueue.

```
% Create a data queue so workers can send the grade back to the client once
% their done
%
% TODO: Create a dataqueue object, q.
```



dataqueue_getting_started

Execute functions in parallel asynchronously using `parfeval`



Asynchronous execution on parallel workers
Useful for "needle in a haystack" problems

```
for idx = 1:10
   f(idx) = parfeval(@magic,1,idx);
end

for idx = 1:10
   [completedIdx,value] = fetchNext(f);
   magicResults{completedIdx} = value;
end
```

Run code in parallel

Synchronously with `parfor`

- You wait for your loop to complete to obtain your results
- Your MATLAB client is blocked from running any new computations
- You cannot break out of loop early

Asynchronously with `parfeval`*

- You can obtain intermediate results
- Your MATLAB client is free to run other computations
- You can break out of loop early

* Runs **function** on parallel workers

Hands-On Exercise: Use `parfeval` to run functions in the background

parfeval Plotter

Parallelize the code so that plots are done as quickly as possible. How do we get the plot to show the results? How can we view and plot intermediate results as the computation progresses?

Hint: Try parfor and parfeval

```
figure('Name','for')
hold on

tic
for idx = 1:100
    y = calc(idx);
    plot(idx,y,'o')
end
toc
```

parfor plotter solution 1

This solution demonstrates how to use a parfor-loop to speed

`parfeval_plotter`

Use Code Analyzer to fix problems when converting `for`-loops to `parfor`-loops

`parfor`-loop iterations have no guaranteed order, and one loop iteration cannot depend on a previous iteration; therefore, you may need to rewrite your code to use `parfor`

```
1    a = zeros(5, 1);
2    b = pi;
3    parfor i = 1:5
4        a(i) = i + b;
5    end
6    disp(a)
```

No warnings found.
(Using Default Settings)

```
1    a = zeros(5, 1);
2    b = pi;
3    parfor i = 2:6
4        a(i) = a(i-1) + b;
5    end
6    disp(a)
```

Line 4: In a PARFOR loop, variable 'a' is indexed in different ways, potentially causing dependencies between iterations.

Common problems when rewriting a `for`-loop as a `parfor`-loop

Noninteger loop variables

```
parfor x = 0:0.1:1
    parfor y = 2:10
        A(y) = A(y-1) + y;
    end
end
```

Nested parallel loops

Dependent loop body

Hands-On Exercise: Rewrite `for`-loops into `parfor`-loops

Rewriting `for`-Loops As `parfor`-Loops

This example focus on diagnosing and fixing issues you may run into rewriting a for-loop to a `parfor`-loop.

Hint: Review the code analyzer messages and check the documentation for additional help.

Ensure `parfor`-loop iterations are task independent

When you rewrite `for`-loops to `parfor`-loops, you need ensure that your `parfor`-loop iterations are independent. `parfor`-loop iterations have *no guaranteed order*, while the iteration order in `for`-loops is *sequential*.

Review the code below which was rewritten as a `parfor`-loop and the code analyzer messages - is this code suitable for parallelization?

```
len = 10;
A = [0 nan(1,len-1)];
parfor idx = 2:len
    A(idx) = A(idx-1) + rand;
end
```

the parfor loop variable is

Hands-On Exercise: Refactoring `for`-loops

Loading and Saving in `parfor`

One way to avoid issues that arise from rewriting for-loops to parfor-loops is to refactor the loop. The following example illustrates a best practice when rewriting loops.

Rewrite the for-loop as a parfor-loop.

```
for idx = 1:4
    load xy.mat
    z = x*y*rand;
    save(['z' num2str(idx) '.mat'], 'z')
end
```

First attempt

parfor-loops need to be able to classify each variable inside their block. View the parfor transparency documentation for help getting started. Once a variable has been classified, it can not change its classification. In the above
MATLAB is unaware that x and y are variables (they are assumed to be functions
introduces new variables when the parfor is running -- a violation

Consider parallel overhead* in deciding when to use `parfor`

`parfor` can be useful ☺

- `for`-loops with loop iterations that take long to execute
- `for`-loops with **many** loop iterations that take a short time, e.g., parameter sweep

`parfor` might not be useful ☹

- `for`-loops with loop iterations that take a short time to execute

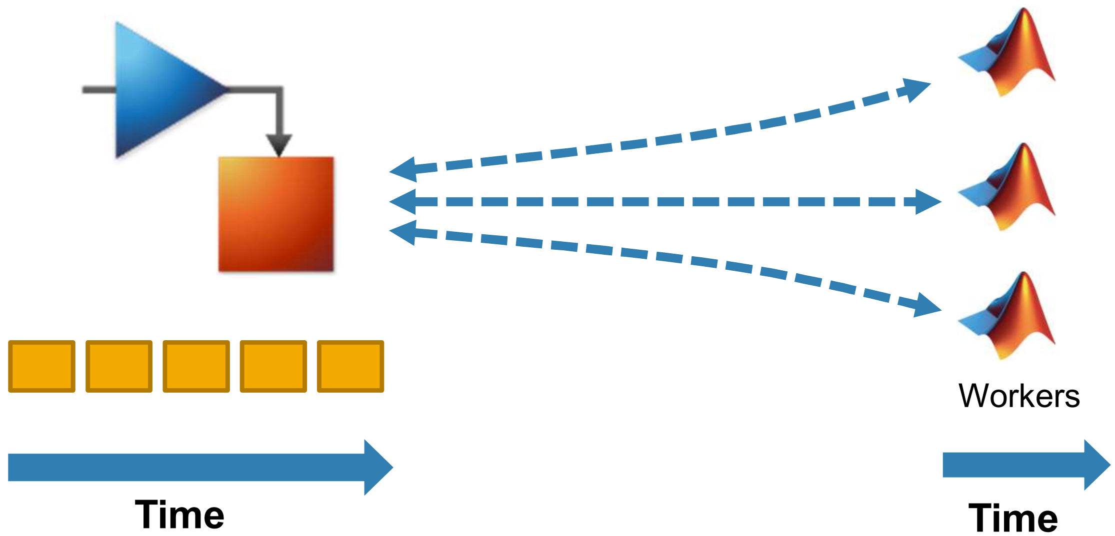
Ways to improve `parfor` performance:

- Experiment which is faster: creating arrays before the loop or have each worker create its own arrays inside the loop (saves transfer time, especially on a remote cluster)
- Use sliced variables or temporary variables

… Check mathworks.com/help/parallel-computing/improve-parfor-performance.html

* Parallel overhead: time required for communication, coordination, and data transfer from client to workers and back

Run multiple simulations in parallel with `parsim`



- Run independent Simulink simulations in parallel using the `parsim` function

```
for i = 10000:-1:1
    in(i) = Simulink.SimulationInput(my_model);
    in(i) = in(i).setVariable(my_var, i);
end
out = parsim(in);
```

Hands-On Exercise: Parameters sweeps with Simulink

Parallel Simulation with parsim

In this exercise, we'll run a parallel sweep through a Simulink model.

```
doc Simulink.SimulationInput
doc parsim
doc batchsim
```

Damping and stiffness

We are going to run a parameter sweep over different combinations of damping and stiffness. Define a grid of all combinations.

```
bNum = 5;
kNum = 5;
bVals = linspace(0.1, 5, bNum);  % damping
kVals = linspace(1.5, 5, kNum);  % stiffness
[kGrid, bGrid] = meshgrid(bVals, kVals);
```

Scaling MATLAB applications and Simulink simulations

Ease of Use

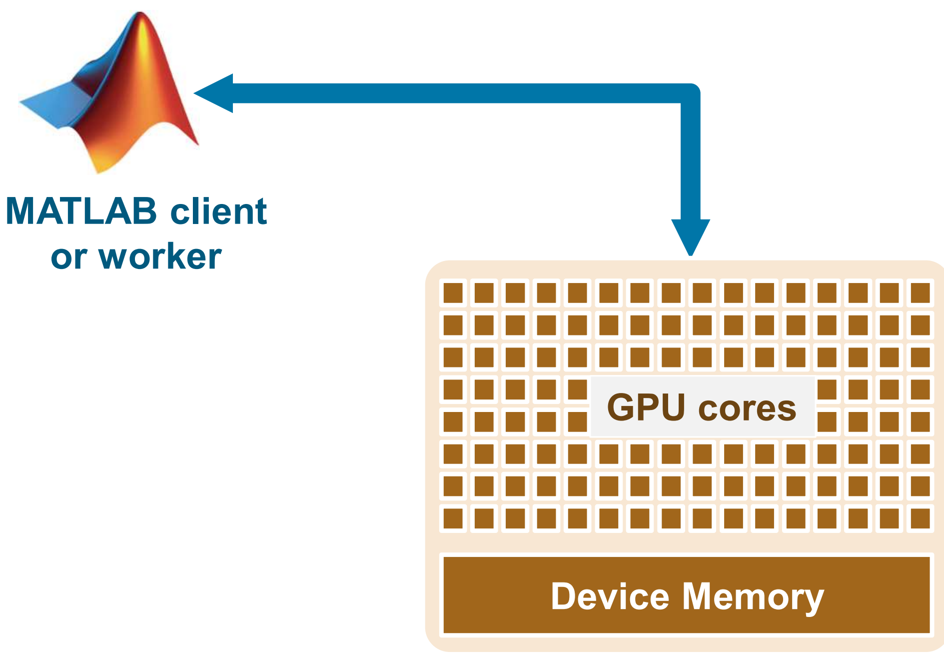
Automatic parallel support in toolboxes

Common programming constructs

Advanced programming constructs
`(spmd,etc.)`

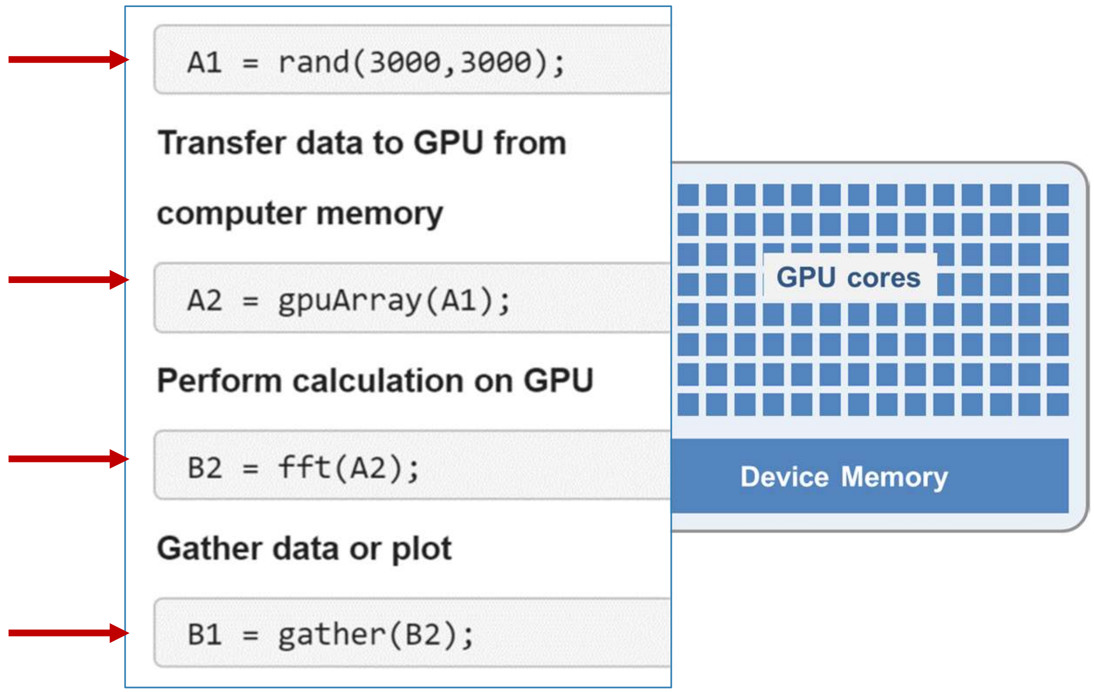
Greater Control

Leverage NVIDIA GPUs without learning CUDA



Leverage your GPU to accelerate your MATLAB code

- Ideal Problems
  - massively parallel and/or vectorized operations
  - computationally intensive
- 1000+ GPU-supported functions
- Use `gpuArray` and `gather` to transfer data between CPU and GPU



MATLAB GPU Computing

Hands-On Exercise: Offload computations to your GPU

Getting started with GPU Computing in MATLAB

This exercise will demonstrate how to speed up computations using your computer's GPU.

We'll start with an algorithm initially written to run on the CPU. If all the functions that you want to use are supported on the GPU, you can simply use `gpuArray` to transfer input data to the GPU and call `gather` to retrieve the output data from the GPU. With some minor changes to the code, we'll be able to offload the computation to the GPU.

Run a computation on the CPU

Compute the `fft` of a random matrix.

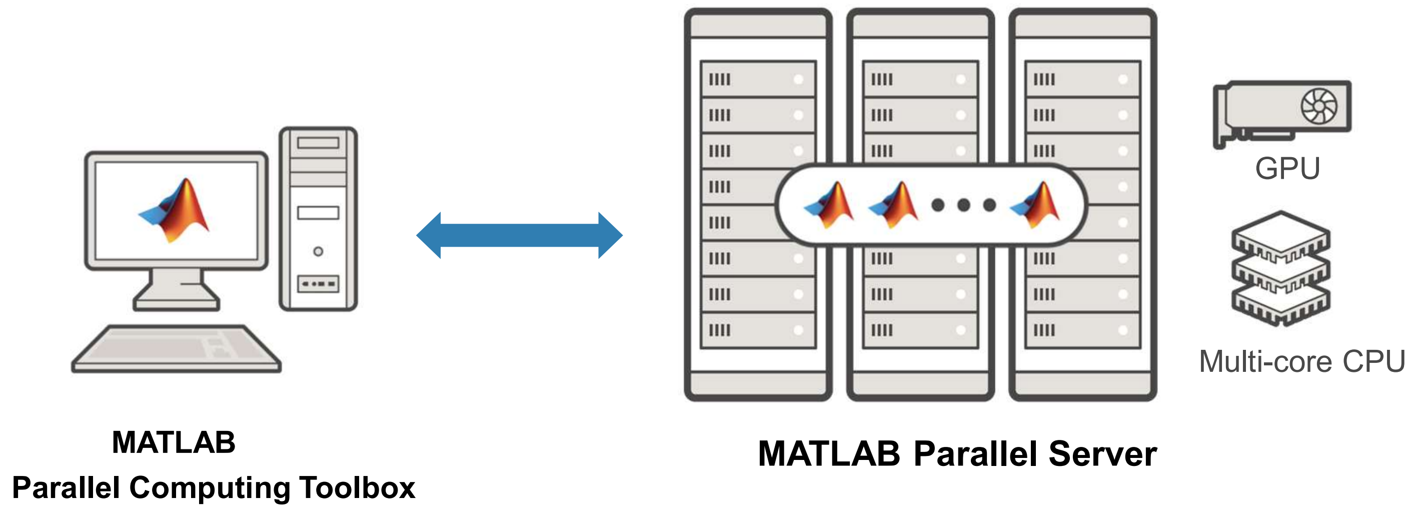
```
N = 8192;
matrix_cpu = rand(N,N);

tic
out_cpu = fft(matrix_cpu);
time_cpu = toc;

disp(['FFT time on CPU: ' num2str(time_cpu)])
```

gpus_getting_started

Parallel computing on your desktop, clusters, and clouds

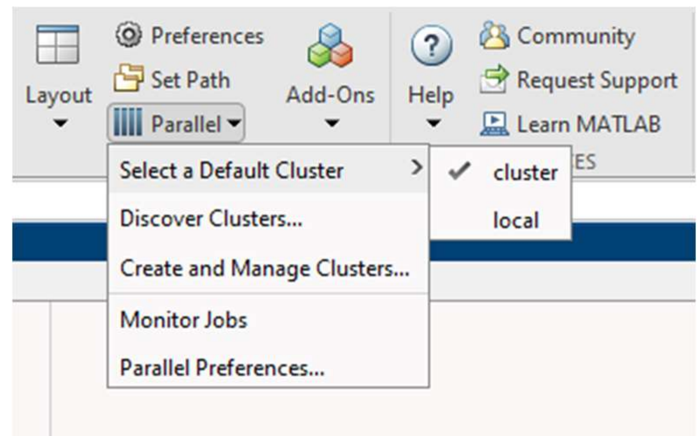


- Prototype on the desktop
- Integrate with infrastructure
- Access directly through MATLAB

Scale to clusters and clouds

With MATLAB Parallel Server, you can…

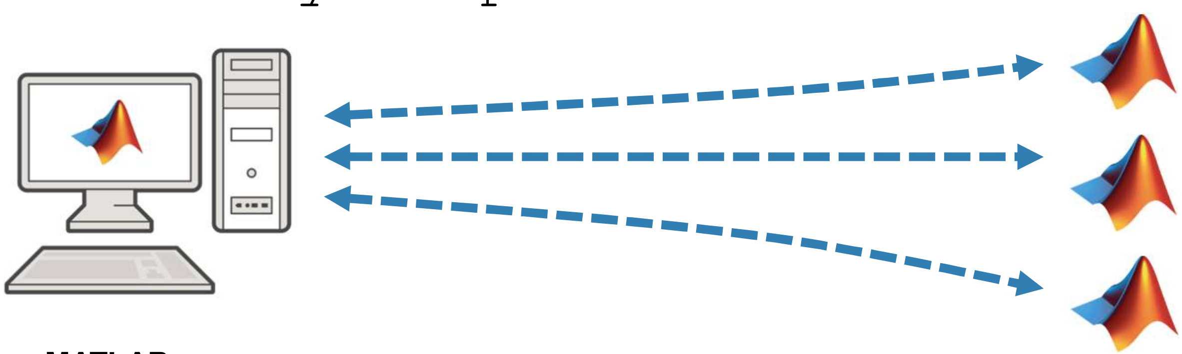
- Change hardware with minimal code change
- Submit to on-premise or cloud clusters
- Support cross-platform submission
  - Windows client to Linux cluster



Interactive parallel computing

Leverage cluster resources in MATLAB

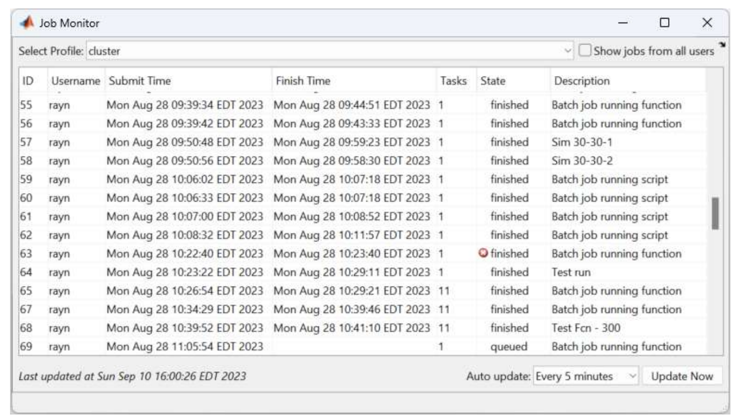
```
>> parpool('cluster', 3);
>> myscript
```



MATLAB
Parallel Computing Toolbox

myscript.m

```
a = zeros(5, 1);
b = pi;
parfor i = 1:5
   a(i) = i + b;
end
```



Job Monitor

Select Profile: cluster Show jobs from all users

ID	Username	Submit Time	Finish Time	Tasks	State	Description
55	rayn	Mon Aug 28 09:39:34 EDT 2023	Mon Aug 28 09:44:51 EDT 2023	1	finished	Batch job running function
56	rayn	Mon Aug 28 09:39:42 EDT 2023	Mon Aug 28 09:43:33 EDT 2023	1	finished	Batch job running function
57	rayn	Mon Aug 28 09:50:48 EDT 2023	Mon Aug 28 09:59:23 EDT 2023	1	finished	Sim 30-30-1
58	rayn	Mon Aug 28 09:50:56 EDT 2023	Mon Aug 28 09:58:30 EDT 2023	1	finished	Sim 30-30-2
59	rayn	Mon Aug 28 10:06:02 EDT 2023	Mon Aug 28 10:07:18 EDT 2023	1	finished	Batch job running script
60	rayn	Mon Aug 28 10:06:33 EDT 2023	Mon Aug 28 10:07:18 EDT 2023	1	finished	Batch job running script
61	rayn	Mon Aug 28 10:07:00 EDT 2023	Mon Aug 28 10:08:52 EDT 2023	1	finished	Batch job running script
62	rayn	Mon Aug 28 10:08:32 EDT 2023	Mon Aug 28 10:11:57 EDT 2023	1	finished	Batch job running script
63	rayn	Mon Aug 28 10:22:40 EDT 2023	Mon Aug 28 10:23:40 EDT 2023	1	finished	Batch job running function
64	rayn	Mon Aug 28 10:23:22 EDT 2023	Mon Aug 28 10:29:11 EDT 2023	1	finished	Test run
65	rayn	Mon Aug 28 10:26:54 EDT 2023	Mon Aug 28 10:29:21 EDT 2023	11	finished	Batch job running function
67	rayn	Mon Aug 28 10:34:29 EDT 2023	Mon Aug 28 10:39:46 EDT 2023	11	finished	Batch job running function
68	rayn	Mon Aug 28 10:39:52 EDT 2023	Mon Aug 28 10:41:10 EDT 2023	11	finished	Test Fcn - 300
69	rayn	Mon Aug 28 11:05:54 EDT 2023		1	queued	Batch job running function

Last updated at Sun Sep 10 16:00:26 EDT 2023 Auto update: Every 5 minutes Update Now

Run a parallel pool from specified profile

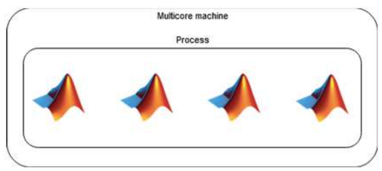
On local machine

- Start parallel pool of local workers

```
parpool('Processes');
```

- Start parallel pool of thread workers

```
parpool('Threads');
```



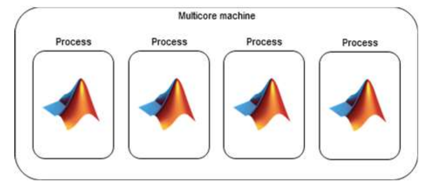
☺ Reduced memory usage, faster scheduling, lower data transfer costs

☹ Thread-based environments support only a subset of the functions available for process workers

On cluster

- Start parallel pool using cluster object

```
c = parcluster;
parpool(c);
```

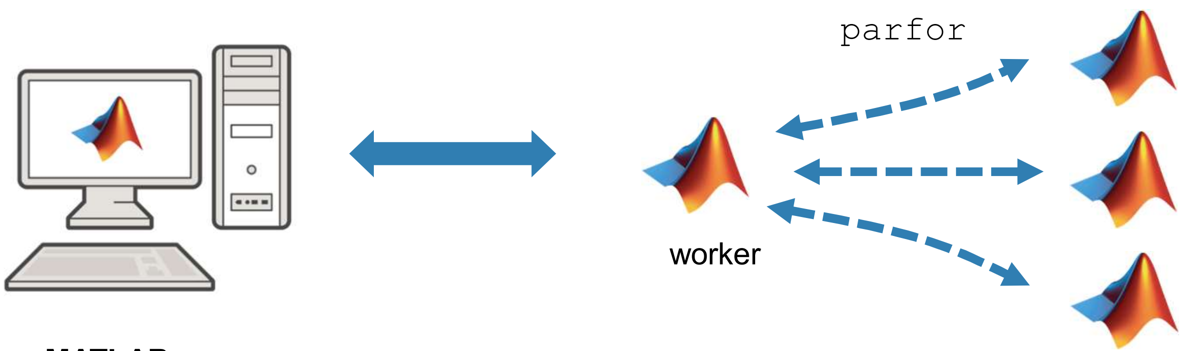


mathworks.com/help/parallel-computing/choose-between-thread-based-and-process-based-environments.html

`batch` simplifies offloading computations

Submit MATLAB jobs to the cluster

```
>> job = batch('myscript','Pool',3);
```



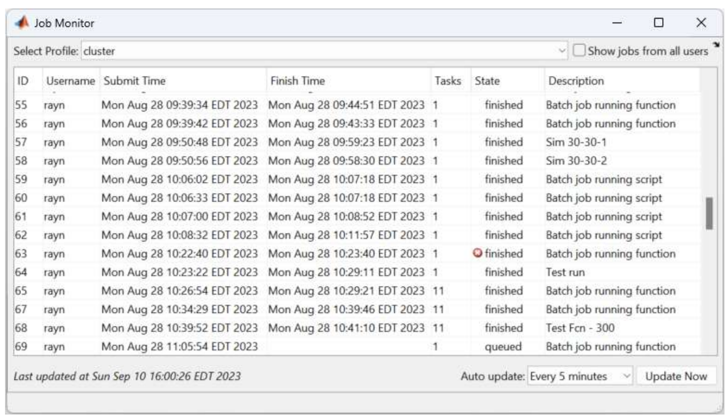
MATLAB

Parallel Computing Toolbox

```
>> j.State
ans =
    'running'
>> j.diary
Warning: The diary of this batch job might be incomplete
because the job is still running.
--- Start Diary ---

Analyzed 1 image.
Analyzed 2 images.
Analyzed 3 images.
Analyzed 4 images.

--- End Diary ---
```



Job Monitor

Select Profile: cluster   Show jobs from all users

ID	Username	Submit Time	Finish Time	Tasks	State	Description
55	rayn	Mon Aug 28 09:39:34 EDT 2023	Mon Aug 28 09:44:51 EDT 2023	1	finished	Batch job running function
56	rayn	Mon Aug 28 09:39:42 EDT 2023	Mon Aug 28 09:43:33 EDT 2023	1	finished	Batch job running function
57	rayn	Mon Aug 28 09:50:48 EDT 2023	Mon Aug 28 09:59:23 EDT 2023	1	finished	Sim 30-30-1
58	rayn	Mon Aug 28 09:50:56 EDT 2023	Mon Aug 28 09:58:30 EDT 2023	1	finished	Sim 30-30-2
59	rayn	Mon Aug 28 10:06:02 EDT 2023	Mon Aug 28 10:07:18 EDT 2023	1	finished	Batch job running script
60	rayn	Mon Aug 28 10:06:33 EDT 2023	Mon Aug 28 10:07:18 EDT 2023	1	finished	Batch job running script
61	rayn	Mon Aug 28 10:07:00 EDT 2023	Mon Aug 28 10:08:52 EDT 2023	1	finished	Batch job running script
62	rayn	Mon Aug 28 10:08:32 EDT 2023	Mon Aug 28 10:11:57 EDT 2023	1	finished	Batch job running script
63	rayn	Mon Aug 28 10:22:40 EDT 2023	Mon Aug 28 10:23:40 EDT 2023	1	finished	Batch job running function
64	rayn	Mon Aug 28 10:23:22 EDT 2023	Mon Aug 28 10:29:11 EDT 2023	1	finished	Test run
65	rayn	Mon Aug 28 10:26:54 EDT 2023	Mon Aug 28 10:29:21 EDT 2023	11	finished	Batch job running function
67	rayn	Mon Aug 28 10:34:29 EDT 2023	Mon Aug 28 10:39:46 EDT 2023	11	finished	Batch job running function
68	rayn	Mon Aug 28 10:39:52 EDT 2023	Mon Aug 28 10:41:10 EDT 2023	11	finished	Test Fcn - 300
69	rayn	Mon Aug 28 11:05:54 EDT 2023		1	queued	Batch job running function

Last updated at Sun Sep 10 16:00:26 EDT 2023   Auto update: Every 5 minutes   Update Now

Hands-On Exercise: Use `batch` to offload serial and parallel computations

Getting Started with `batch`

You can use batch jobs to offload the execution of long-running computations in the background and carry out other tasks while the batch job is processing. `batch` does not block MATLAB; allowing you to continue working while computations take place in the background. When you submit batch jobs to another computer or cluster, you can close MATLAB on the client and retrieve results later.

In this exercise, you will submit batch jobs from MATLAB to your local machine. The workers will run on the same machine as the client, but the same workflow can be used to submit jobs to a remote compute cluster or the cloud, freeing up your local resources.

Note: Close parallel pool if it's open.

Run a batch job to offload serial computation

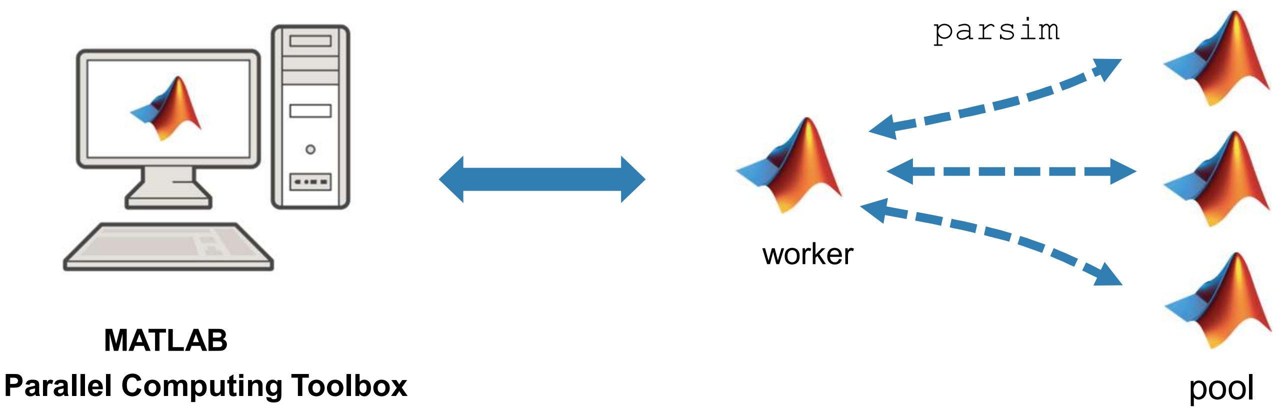
`ex_serial` performs, serially, `N` trials of computing the largest eigenvalue for an `M-by-M` random matrix. `ex_serial` is called as such

```
% matrix size: 50
% trials: 10000
result = ex_serial(50,10000);
```

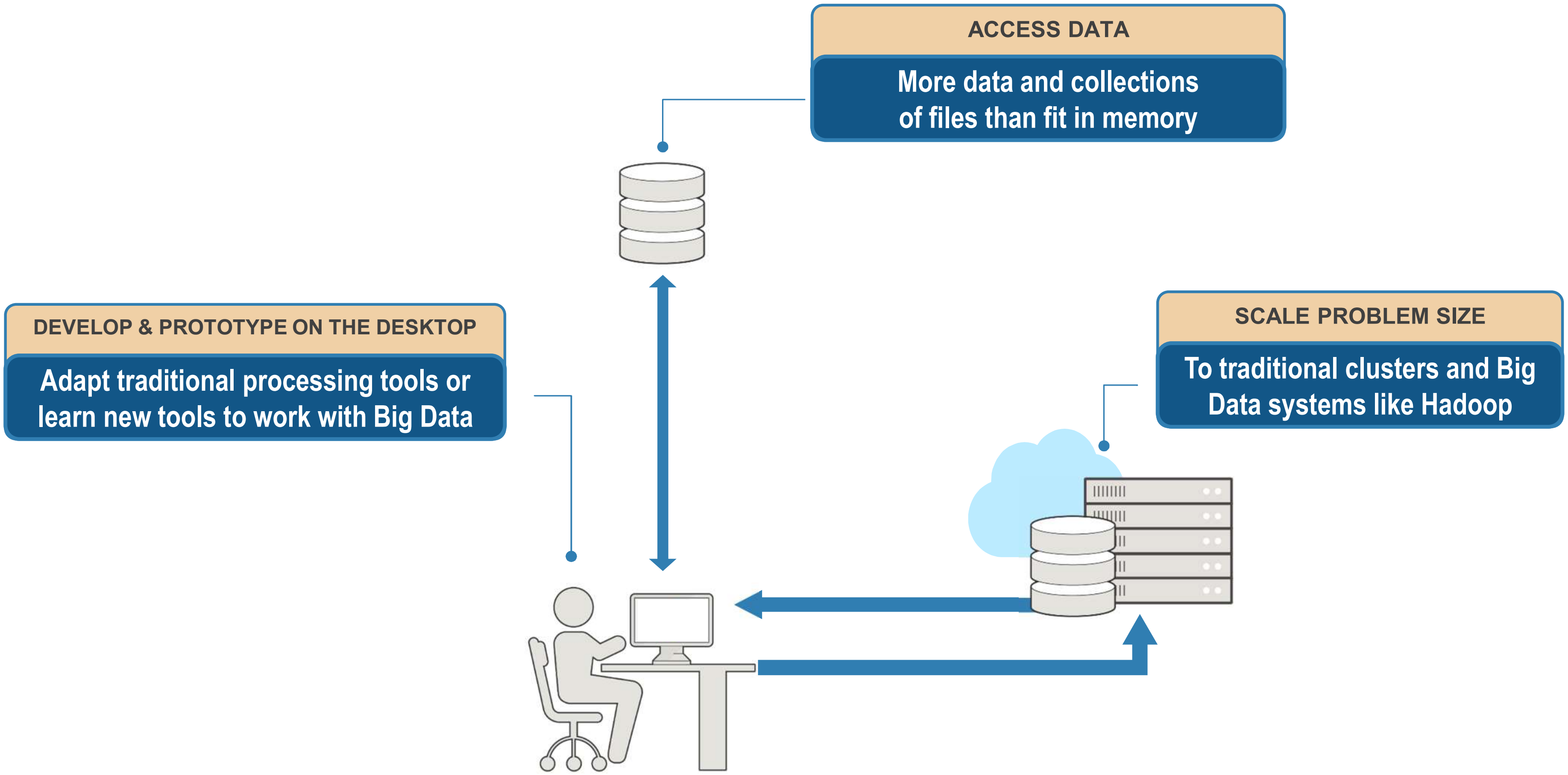
`batch` simplifies offloading simulations

Submit Simulink jobs to the cluster

```
job = batchsim(in,'Pool',3);
```

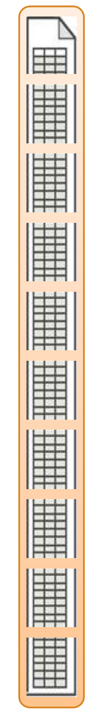


Big Data Workflows



`tall` arrays

- Data type designed for data that doesn’t fit into memory
- Lots of observations (hence “tall”)
- Looks like a normal MATLAB array
  - Supports numeric types, tables, datetimes, strings, etc.
  - Supports several hundred functions for basic math, stats, indexing, etc.
  - Statistics and Machine Learning Toolbox support (clustering, classification, etc.)



Working with tall arrays

Hands-On Exercise: Use `tall` arrays for Big Data

Getting Started with `tall` arrays for Big Data

In this exercise we'll use `tall` arrays to work with large data sets that have more rows than might fit into memory.

You can work with many operations and functions as you would with in-memory MATLAB arrays, but most results are evaluated only when you request them explicitly using `gather`, write a tall array to disk, or visualize the tall array. MATLAB automatically optimizes the queued calculations by minimizing the number of passes through the data.

Access Data: Create Datastore from Sample File(s)

The comma-separated text file `airlinesmall.csv` contains departure and arrival information about individual airline flights from the years 1987 through 2008, stored in a tabular manner.

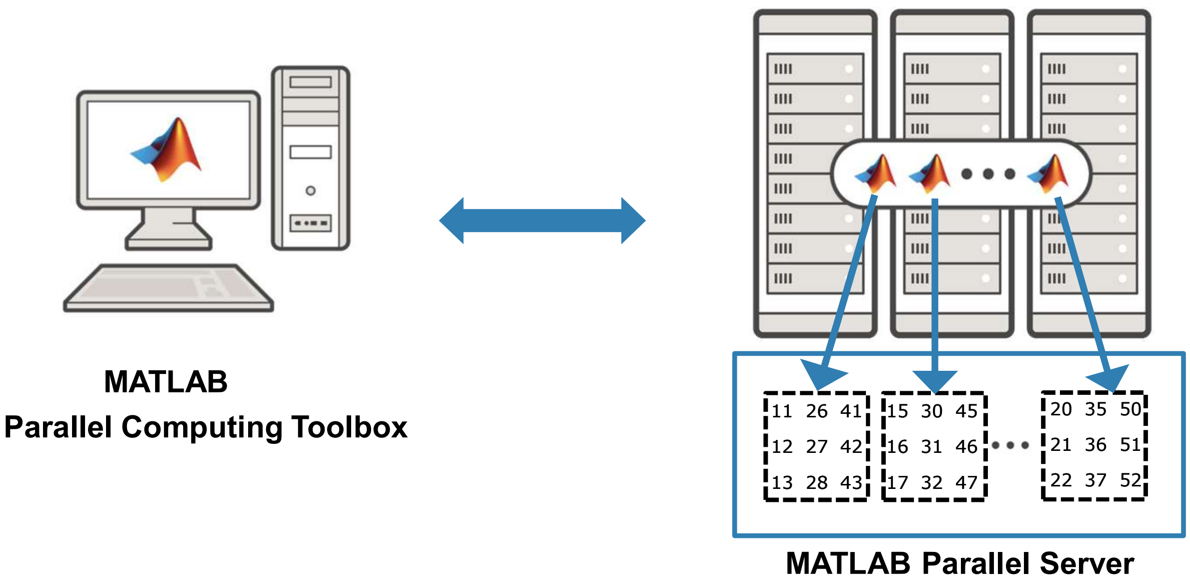
Tabular data that contains a mix of numeric and text data, as well as variable and row names, is best represented in MATLAB as a **table** (a container that stores column-oriented data in variables), because table variables can have different data types and sizes as long as all variables have the same number of rows.

If a file can be imported and processed in its entirety on our computer, we could also import the data in a table, either interactively, using the Import Tool, or programmatically, by reading the comma-separat
function with at least the file name as an input parameter, and then appl

`tall_getting_started`

`distributed` arrays

- Distribute large matrices across workers running on a cluster
- Support includes matrix manipulation, linear algebra, and signal processing
- Several hundred MATLAB functions overloaded for distributed arrays



Working with distributed arrays

Hands-On Exercise: Use `distributed` arrays for Big Data

Getting Started with `distributed` arrays for Big Data

In this example, we will work with distributed arrays, to partition large arrays across workers. We operate on the entire array as a single entity, without having to worry about its distributed nature, as workers operate only on their part of the array, and automatically transfer data between themselves when necessary. Unlike tall arrays, distributed arrays are in the memory of your computer/cluster, but that memory is shared across multiple workers.

Create `distributed` arrays

You can create a distributed array in different ways.

For instance, you can use the `distributed` function to distribute an existing array from the client workspace to the workers of a parallel pool.

Let's open a parallel pool and create a 1000-by-1000 array of uniformly random numbers in the client workspace:

```
p = gcp;
a = rand(1000,1000);
```

Then let's use the `distributed` function to distribute the array to the work

```
distributed(a);
```

distributed_getting_started

`tall` arrays vs. `distributed` arrays

- `tall` arrays are useful for out-of-memory datasets with a "tall" shape
  - Can be used on a desktop, cluster, or with Spark/Hadoop
  - Low-level alternatives are MapReduce and MATLAB API for Spark
- `distributed` arrays are useful for in-memory datasets on a cluster
  - Can be any shape ("tall", "wide", or both)
  - Low-level alternative is SPMD + gop (Global operation across all workers)

	Tall Array	Distributed Array
Support Focus	Data Analytics, Statistics and Machine Learning	Linear Algebra, Matrix Manipulations
Data Shape	"Tall" only	"Tall", "wide" or both
Prototype on Desktop	✓	✓
Helps on Desktop	✓	✗
Run on HPC	✓	✓
Run on Spark/Hadoop	✓	✗
Fault Tolerant	✓	✗

Further Resources

- MATLAB Documentation
  - [MATLAB → Software Development Tools → Performance and Memory]
  - [Parallel Computing Toolbox]
- Parallel and GPU Computing Tutorials
  - https://www.mathworks.com/videos/series/parallel-and-gpu-computing-tutorials-97719.html
- Parallel Computing with MATLAB and Simulink
  - https://www.mathworks.com/solutions/parallel-computing.html