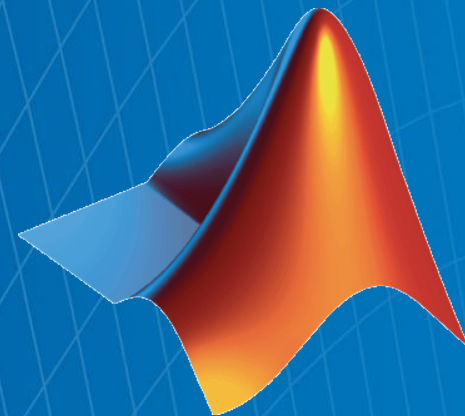


# Faster, Larger, Easier : Parallel Computing with MATLAB

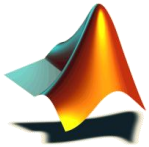
## Hands-On Workshop

**Application Engineer**

엄 준 상



# Agenda



## Best practices in MATLAB programming

- Introduction to parallel computing tools
- Constructs for multicore/multi-processor computers
- Scaling up to a cluster

# Steps for Improving Performance

1. Focus on getting your code working
2. Speed up the code within core MATLAB
  - Assess performance
  - Vectorize loops
  - Preallocate arrays
  - Use appropriate operators
3. Consider additional processing power

# MATLAB Underlying Technologies

- Commercial libraries
  - BLAS: Basic Linear Algebra Subroutines (multithreaded)
  - LAPACK: Linear Algebra Package
  - etc.
  
- JIT/Accelerator
  - Improves looping
  - Generates on-the-fly multithreaded code
  - Continually improving

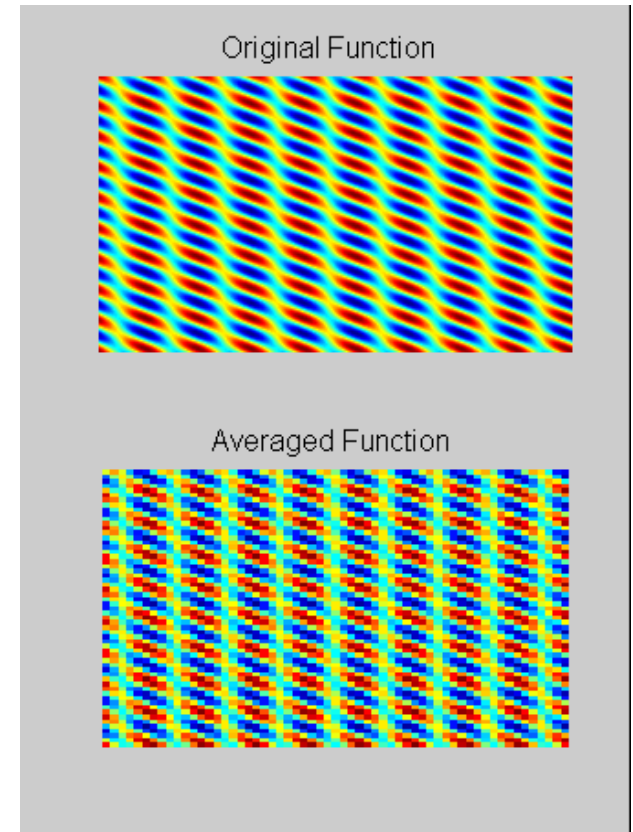
**BLAS and LAPACK  
require contiguous  
arrays**

# Glossary (1)

- Implicit Multithreaded Computation:
  - MATLAB supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads.
- M-Lint code analyzer:
  - Tool built into the Editor/Debugger. Continuously checks your code for problems and recommends modifications to maximize performance and maintainability.
- Profiler:
  - Graphical user interface that displays function execution times within your program

## Exercise : Block Processing Images

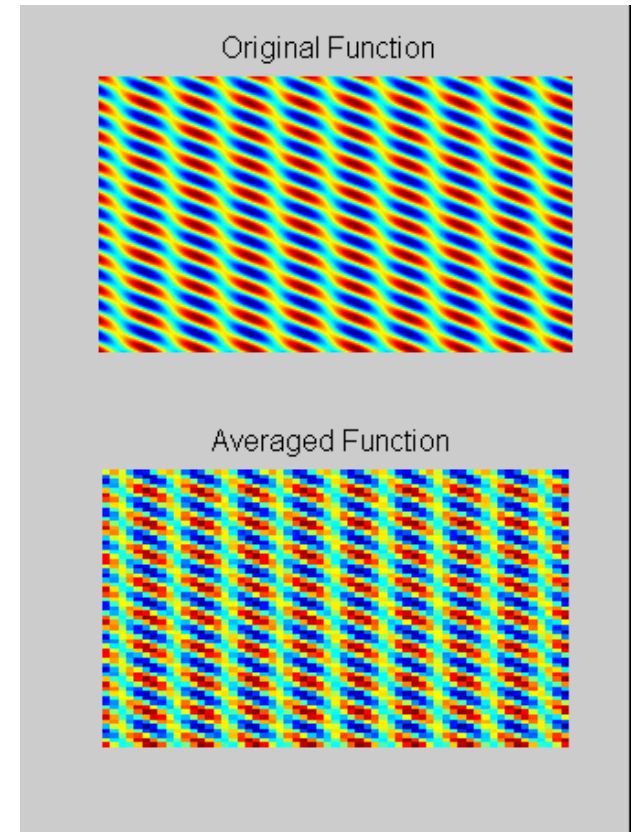
- Evaluate function at grid points
- Reevaluate function over larger blocks
- Compare the results
- Evaluate code performance



# Summary of Block Processing Exercise

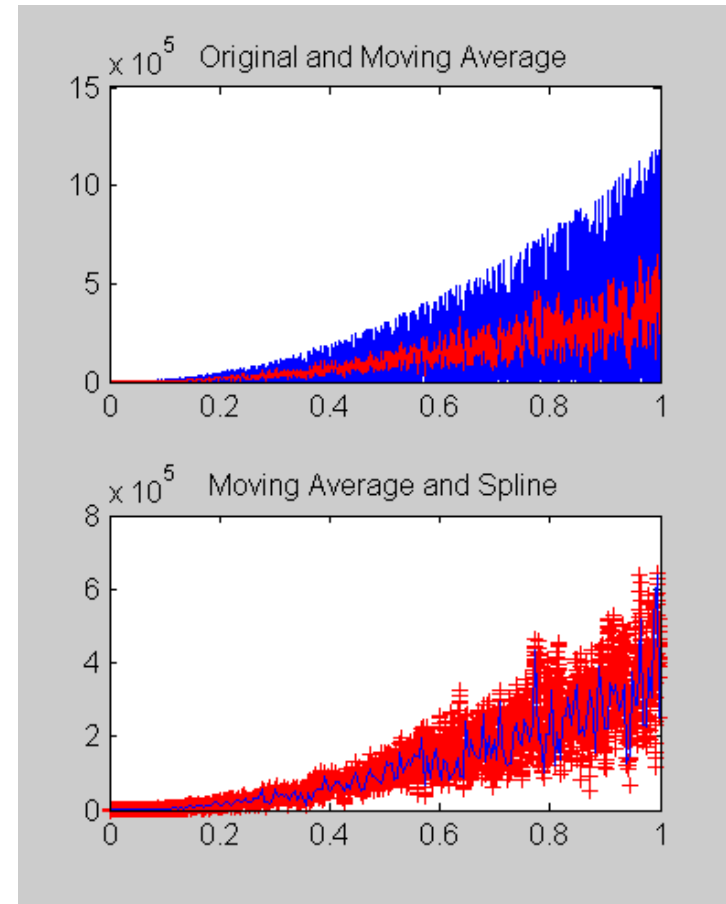
- Used built-in timing functions

```
>> tic
>> toc
```
- Used Code Analyser to find suboptimal code
- Preallocated arrays
- Vectorized code



## Exercise: Fitting Data

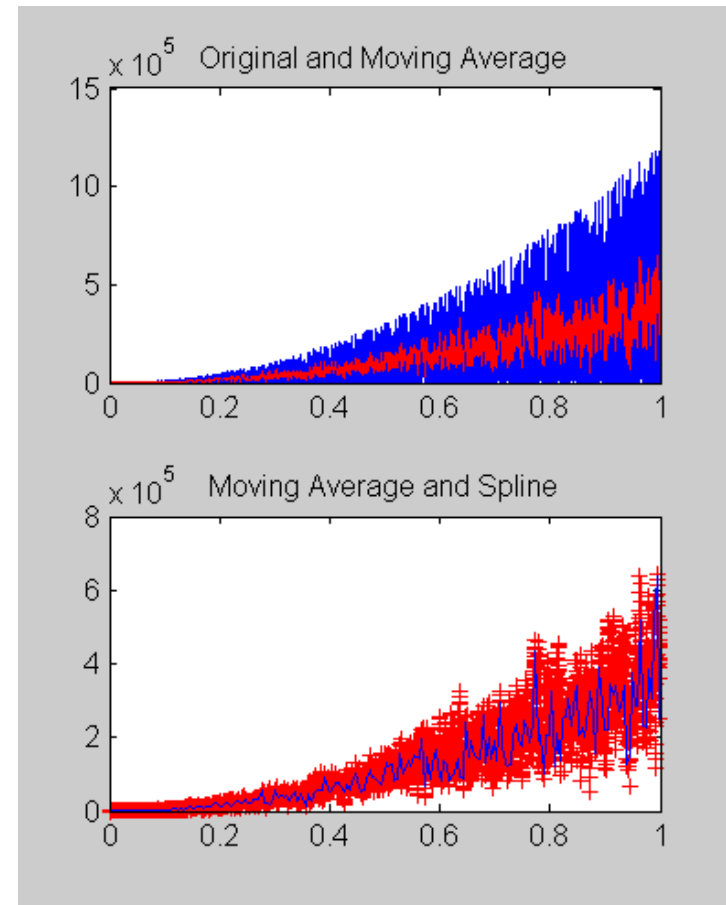
- Load data from multiple files
- Extract a specific test
- Fit a spline to the data
- Write results to Microsoft Excel





# Summary of Fitting Data Exercise

- Used profiler to analyze code
- Targeted significant bottlenecks
- Reduced file I/O
- Reused figure

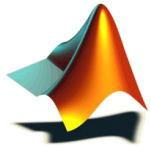


# Classes of Bottlenecks

- File I/O
  - Disk is slow compared to RAM
  - Specify required variables in **load** and **save** commands
- Displaying output
  - Creating new figures is expensive
  - Writing to command window is slow
- Computationally intensive
  - Use what you've learned today
  - Trade-off modularization, readability and performance
  - Integrate other languages or additional hardware
    - e.g. MEX, GPUs, FPGAs, clusters, etc.

# Agenda

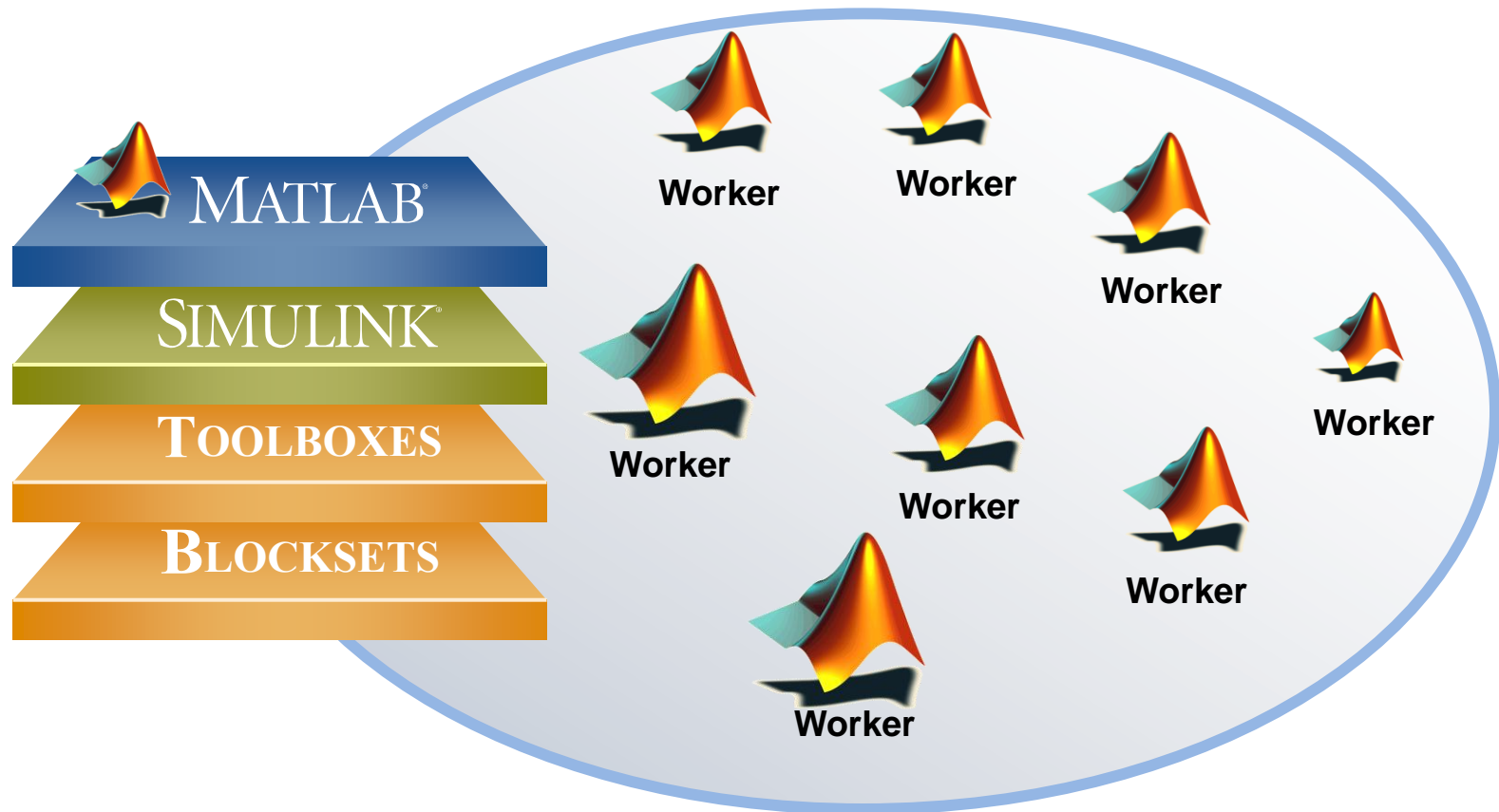
- Best practices in MATLAB programming



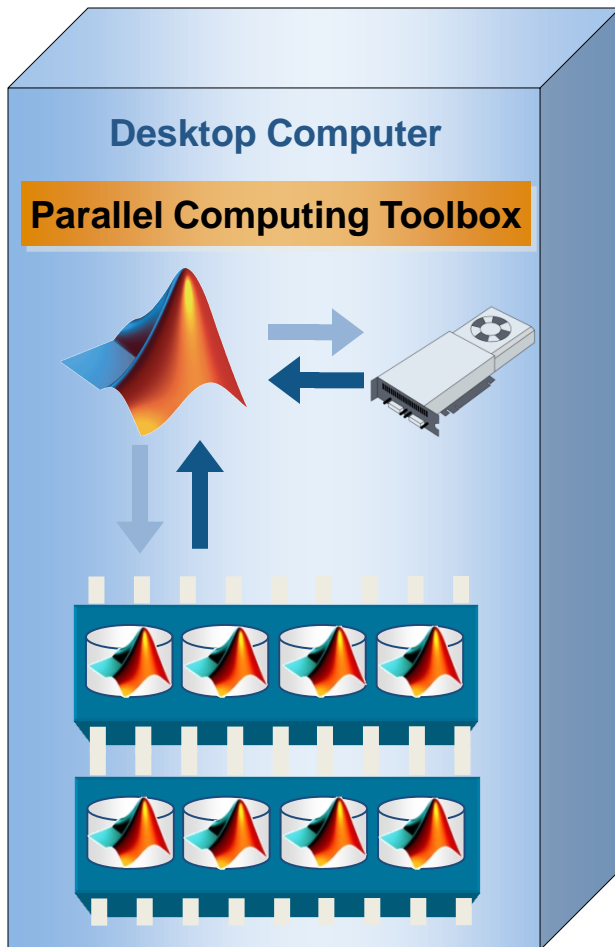
## Introduction to parallel computing tools

- Constructs for multicore/multi-processor computers
- Scaling up to a cluster

# Going Beyond Serial MATLAB Applications

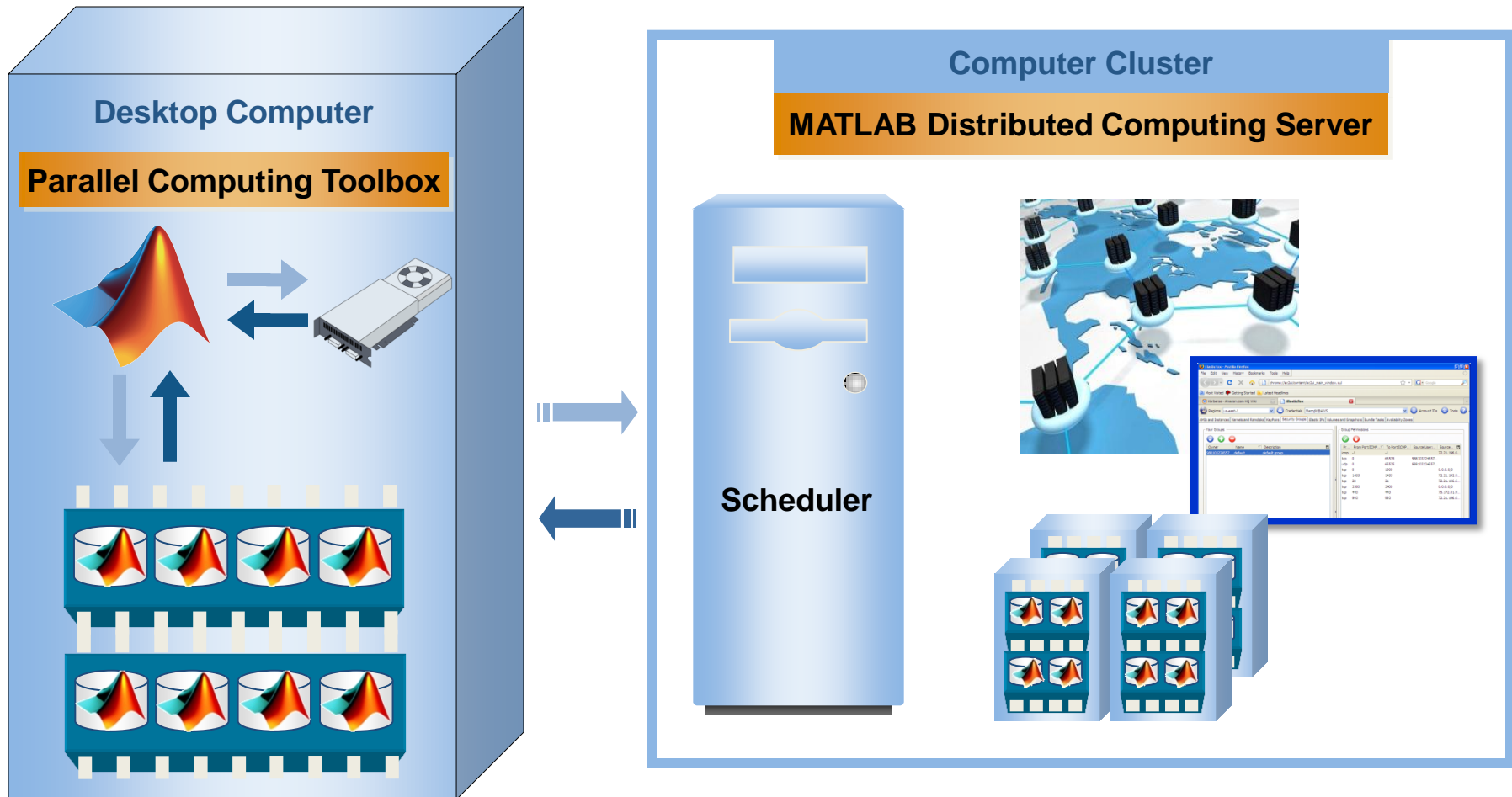


# Parallel Computing Toolbox for the Desktop

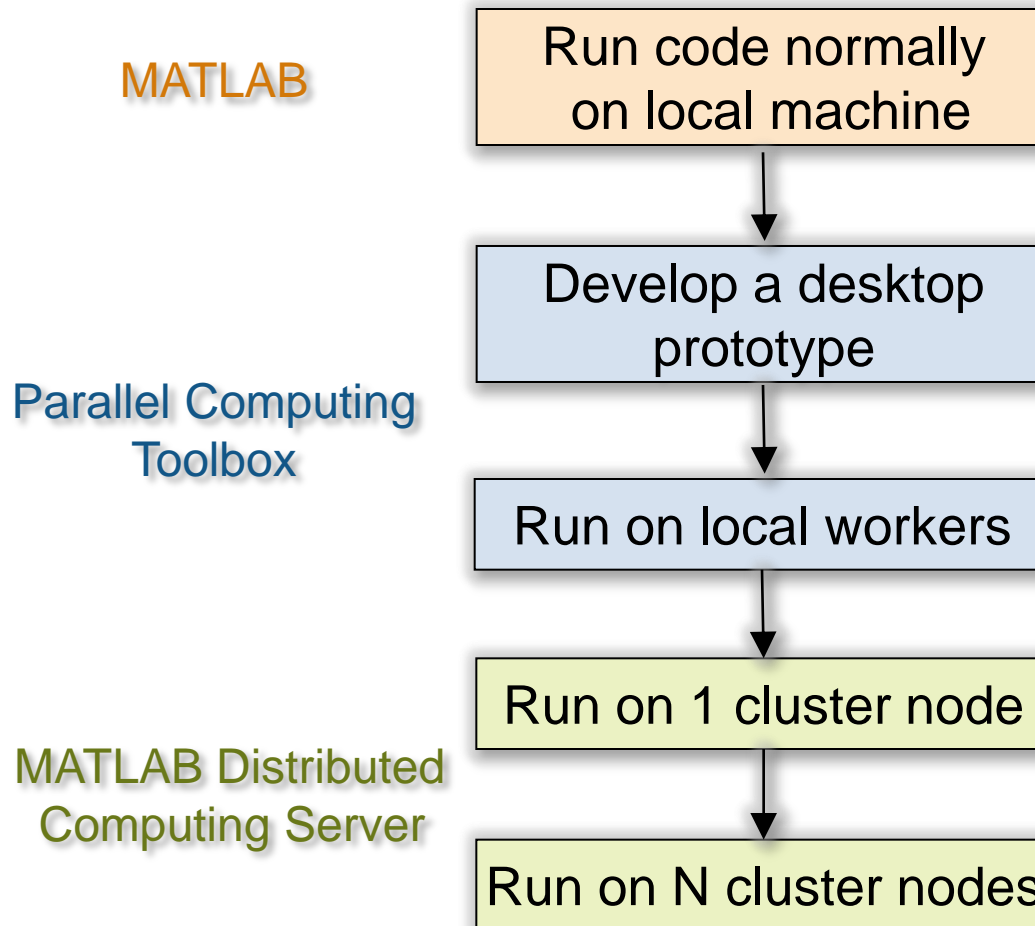


- Speed up parallel applications
- Take advantage of GPUs
- Prototype code for your cluster

# Scale Up to Clusters, Grids, and Clouds

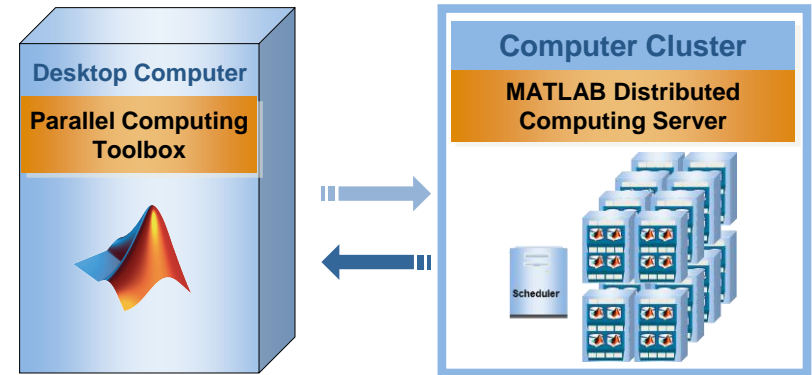


# Program Development Guidelines



# Use cases for Cluster Computing

- Offload computation:
  - Free up desktop
  - Access better computers
  
- Scale speed-up:
  - Use more cores
  - Go from hours to minutes
  
- Scale memory:
  - Utilize distributed arrays
  - Solve larger problems without re-coding

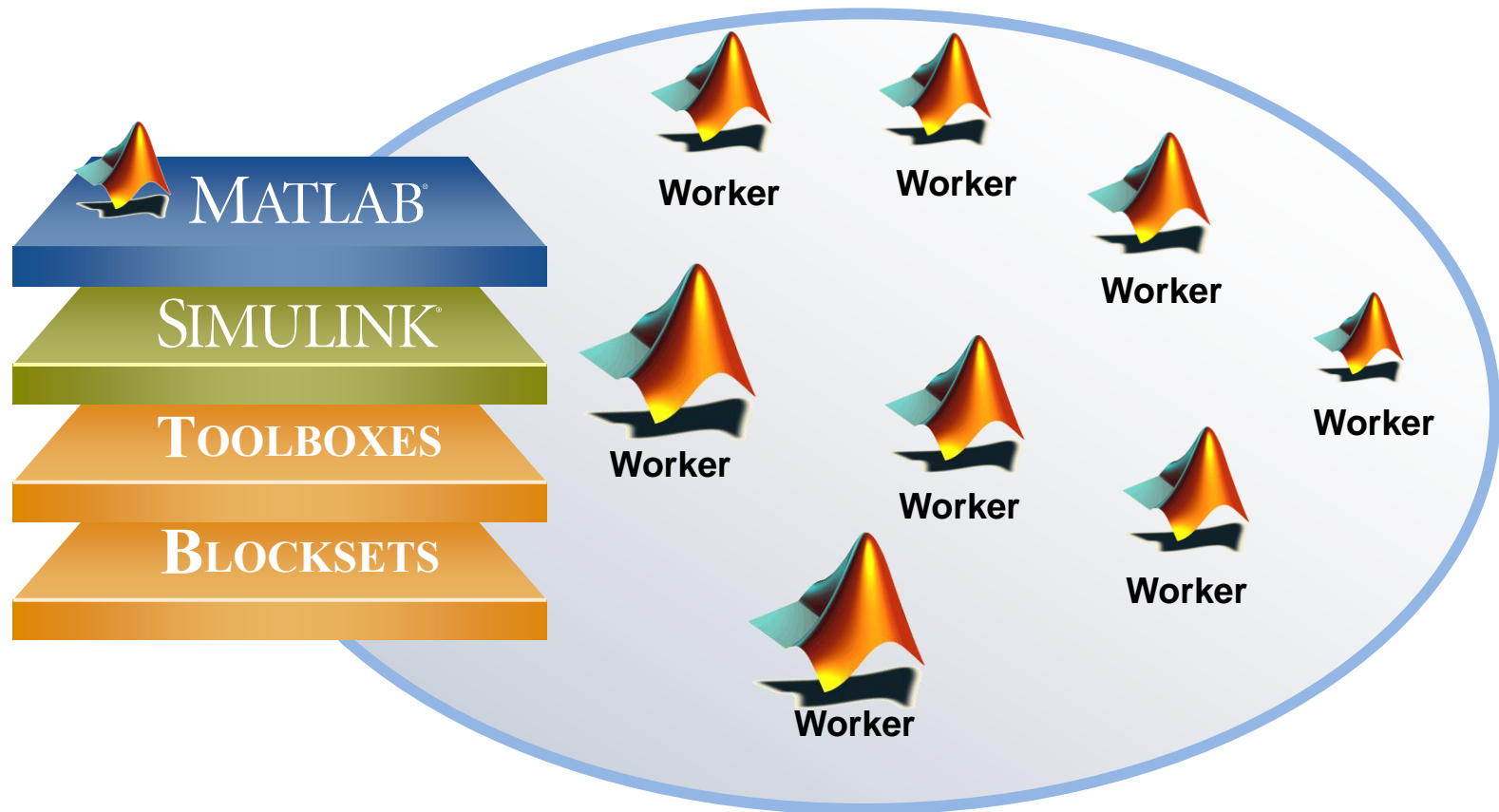




# Interactive to Scheduling

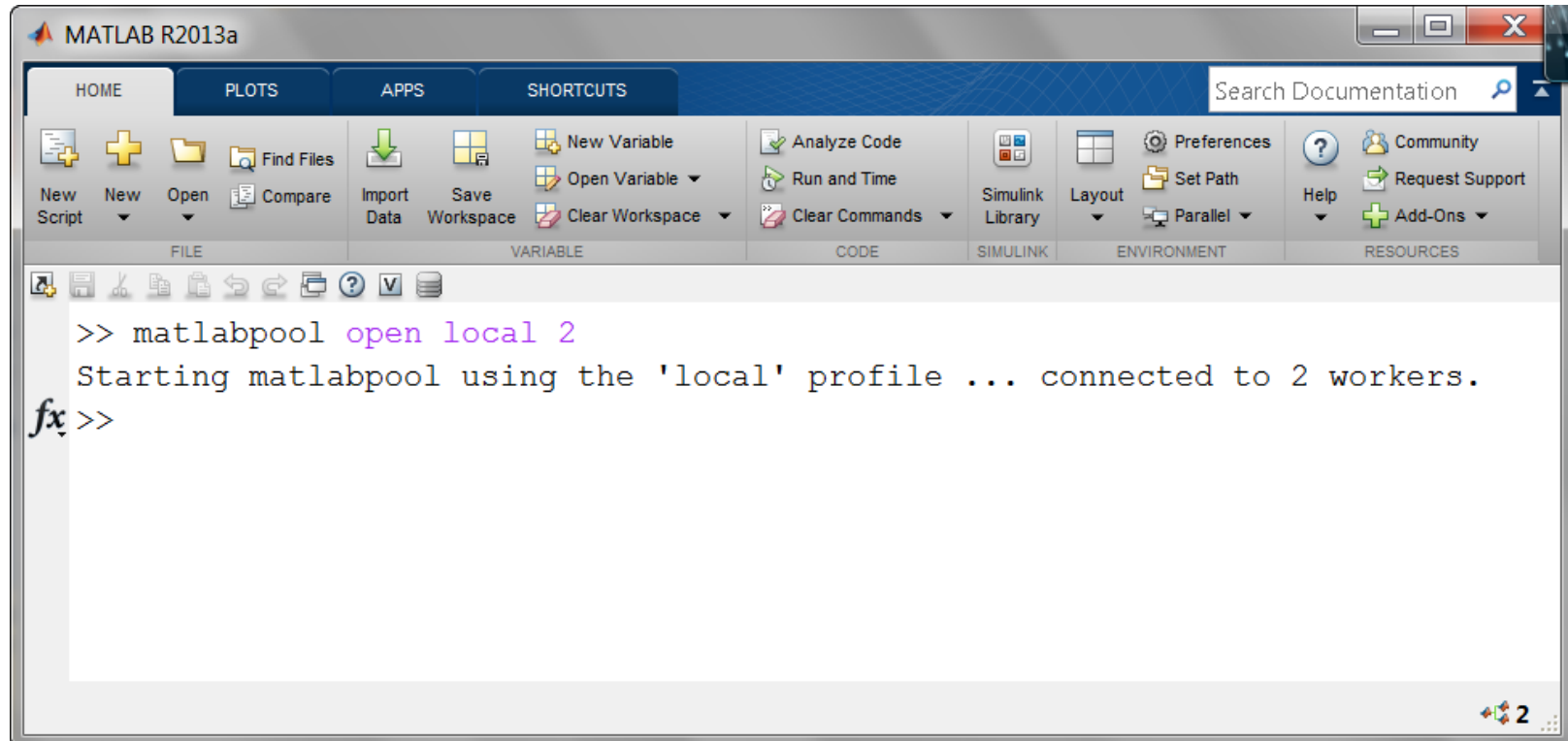
- Interactive
  - Great for prototyping
  - Immediate access to MATLAB workers
  
- Scheduling
  - Offloads work to other MATLAB workers (local or on a cluster)
  - Access to more computing resources for improved performance
  - Frees up local MATLAB session

# Exercise: Working with a matlabpool

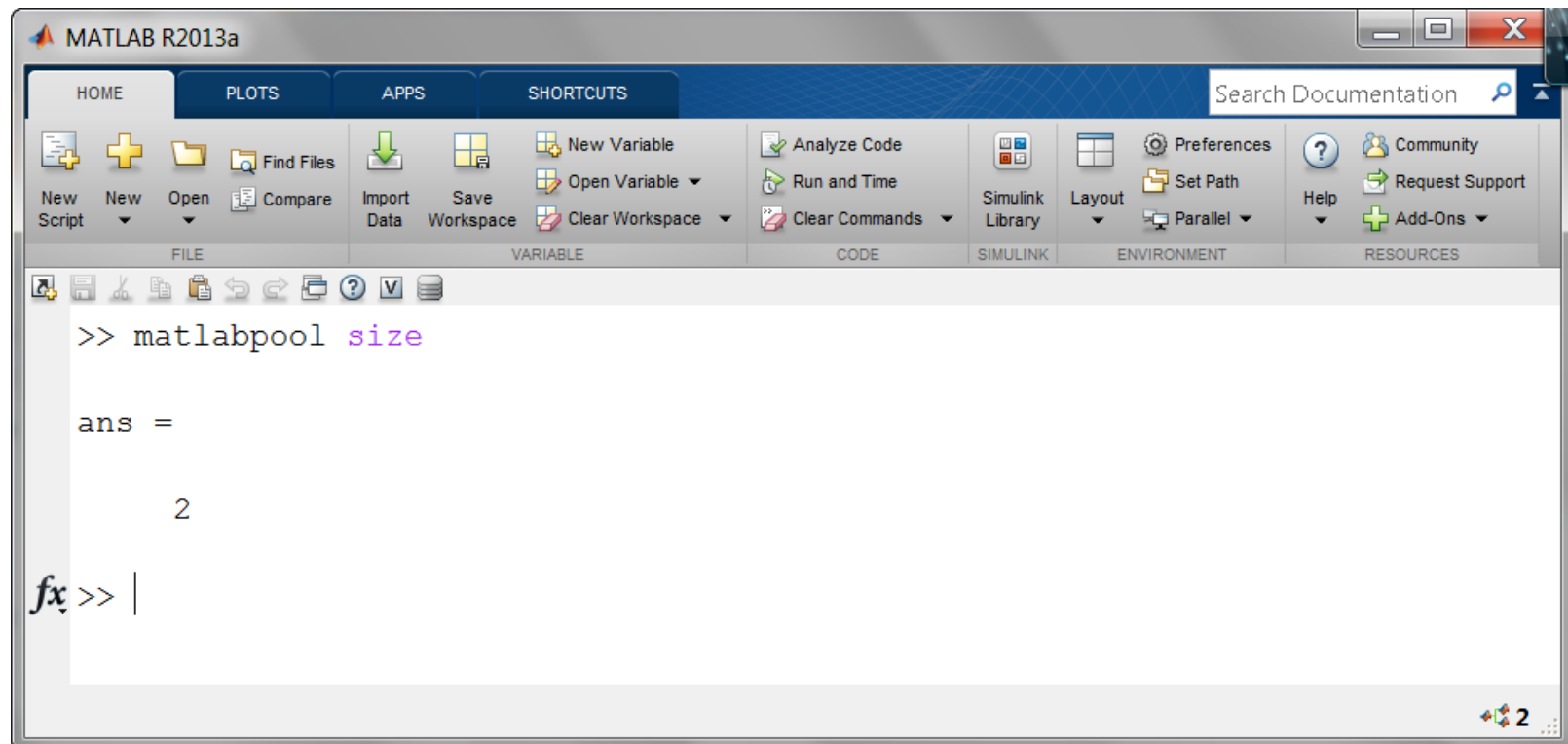


# Starting a matlabpool...

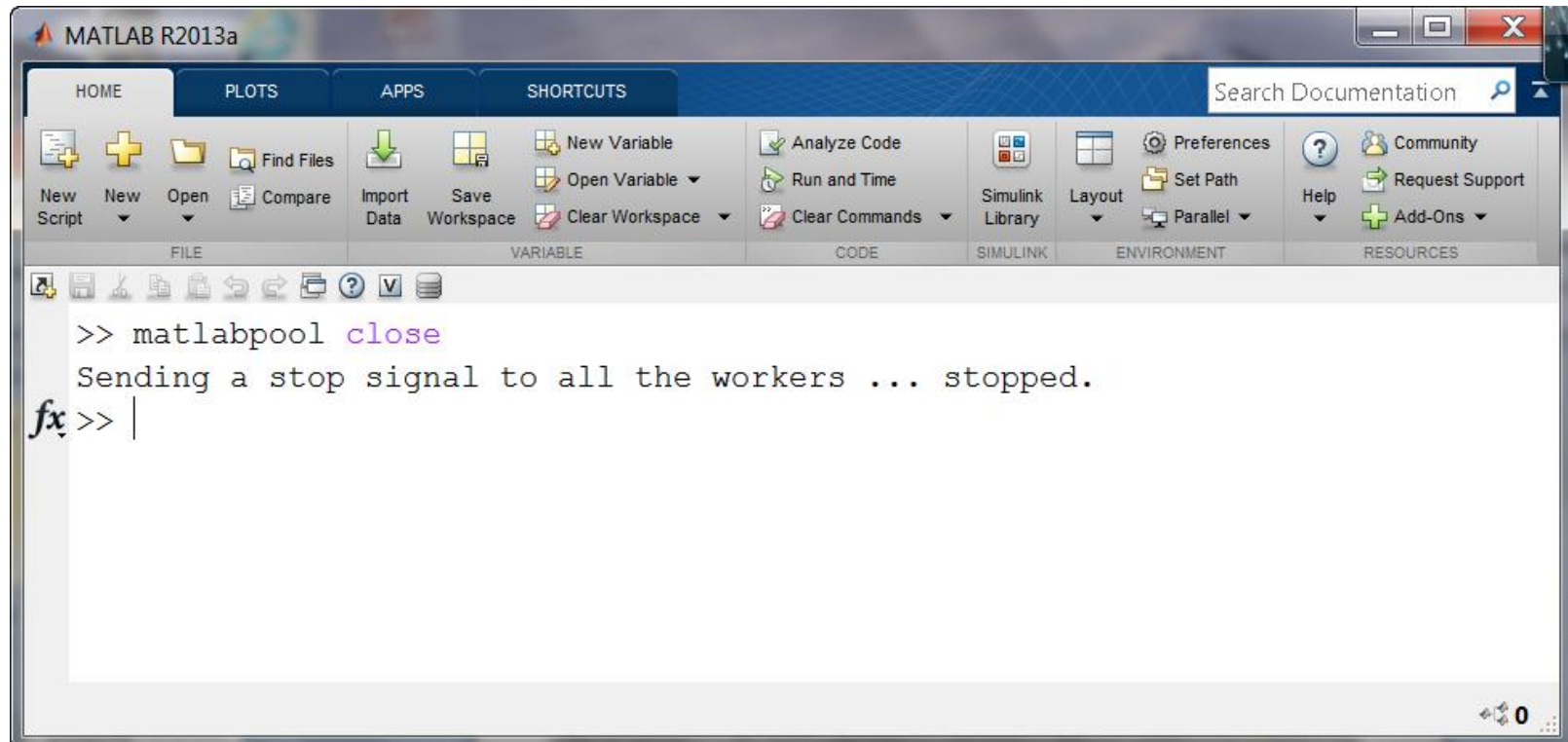
- Open a matlabpool with two Workers using the local configuration



# Determining the Size of the Pool...

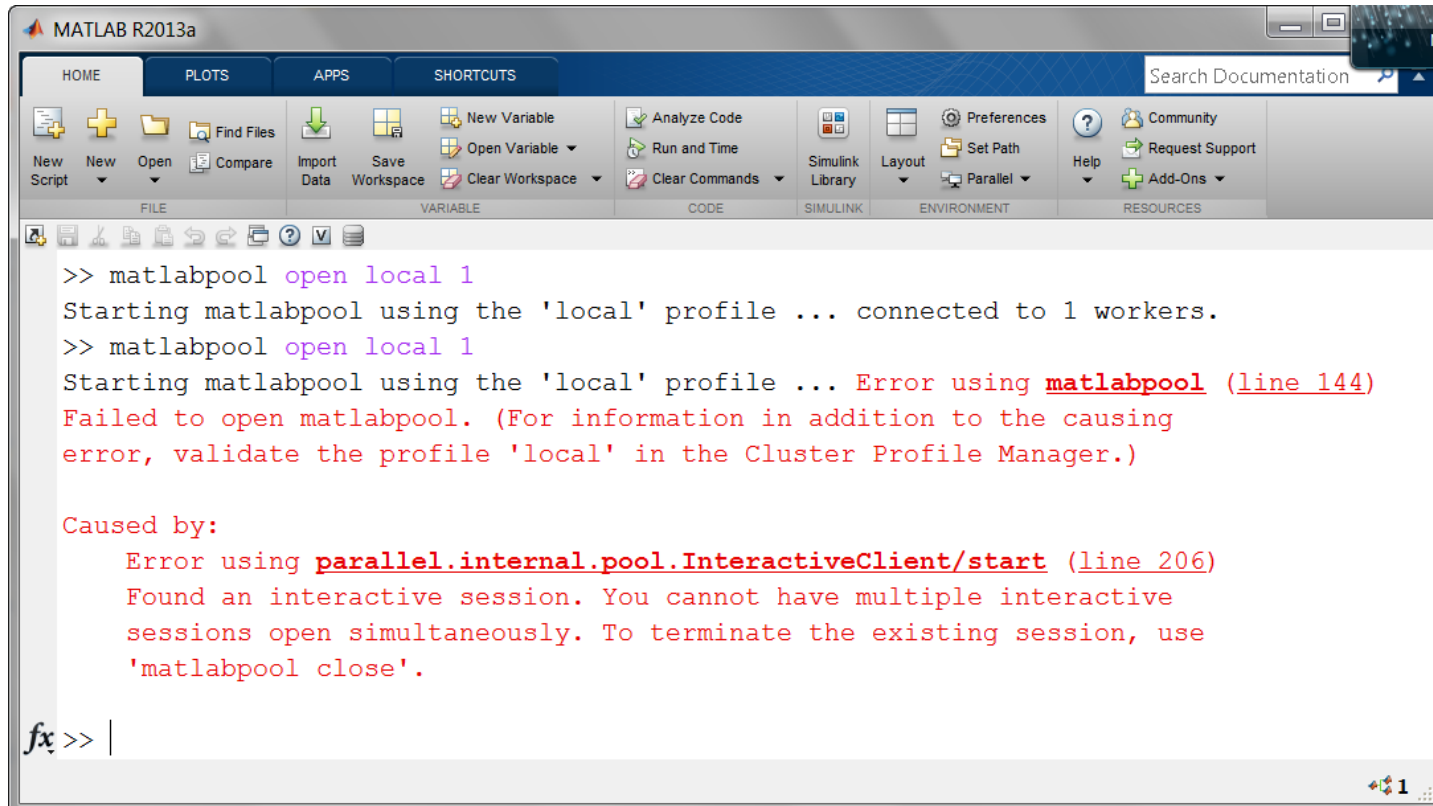


# Stopping a matlabpool



# How many pools of Workers at a time?

- Even if you have not exceeded the maximum number of Workers, you can only open one matlabpool at a time



```

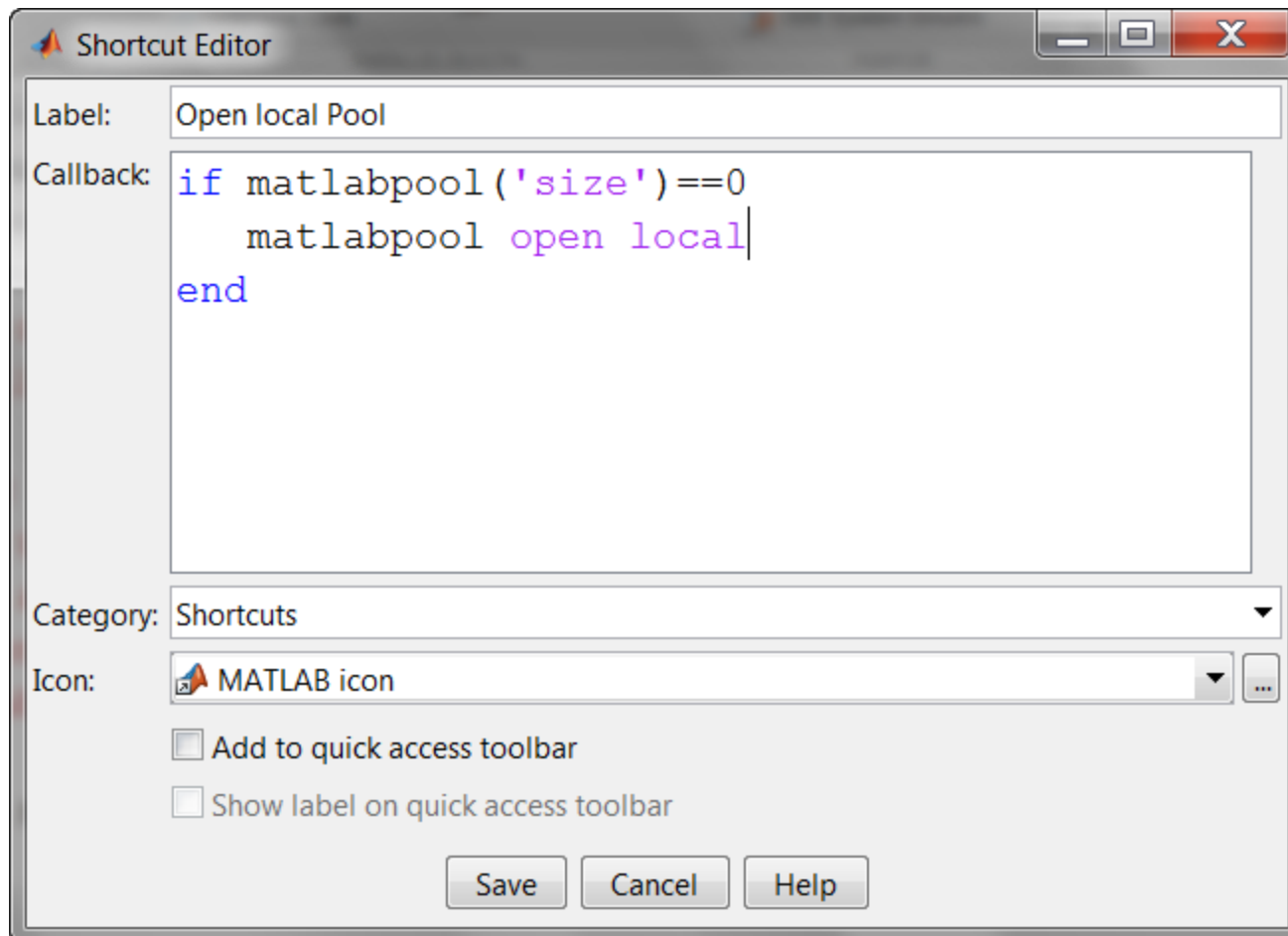
MATLAB R2013a
HOME PLOTS APPS SHORTCUTS Search Documentation
New Script New Open Find Files Import Data Save Workspace New Variable Open Variable Analyze Code Run and Time Simulink Library Layout Set Path Preferences Help Community Request Support Add-Ons
FILE VARIABLE CODE SIMULINK ENVIRONMENT RESOURCES

>> matlabpool open local 1
Starting matlabpool using the 'local' profile ... connected to 1 workers.
>> matlabpool open local 1
Starting matlabpool using the 'local' profile ... Error using matlabpool (line 144)
Failed to open matlabpool. (For information in addition to the causing
error, validate the profile 'local' in the Cluster Profile Manager.)

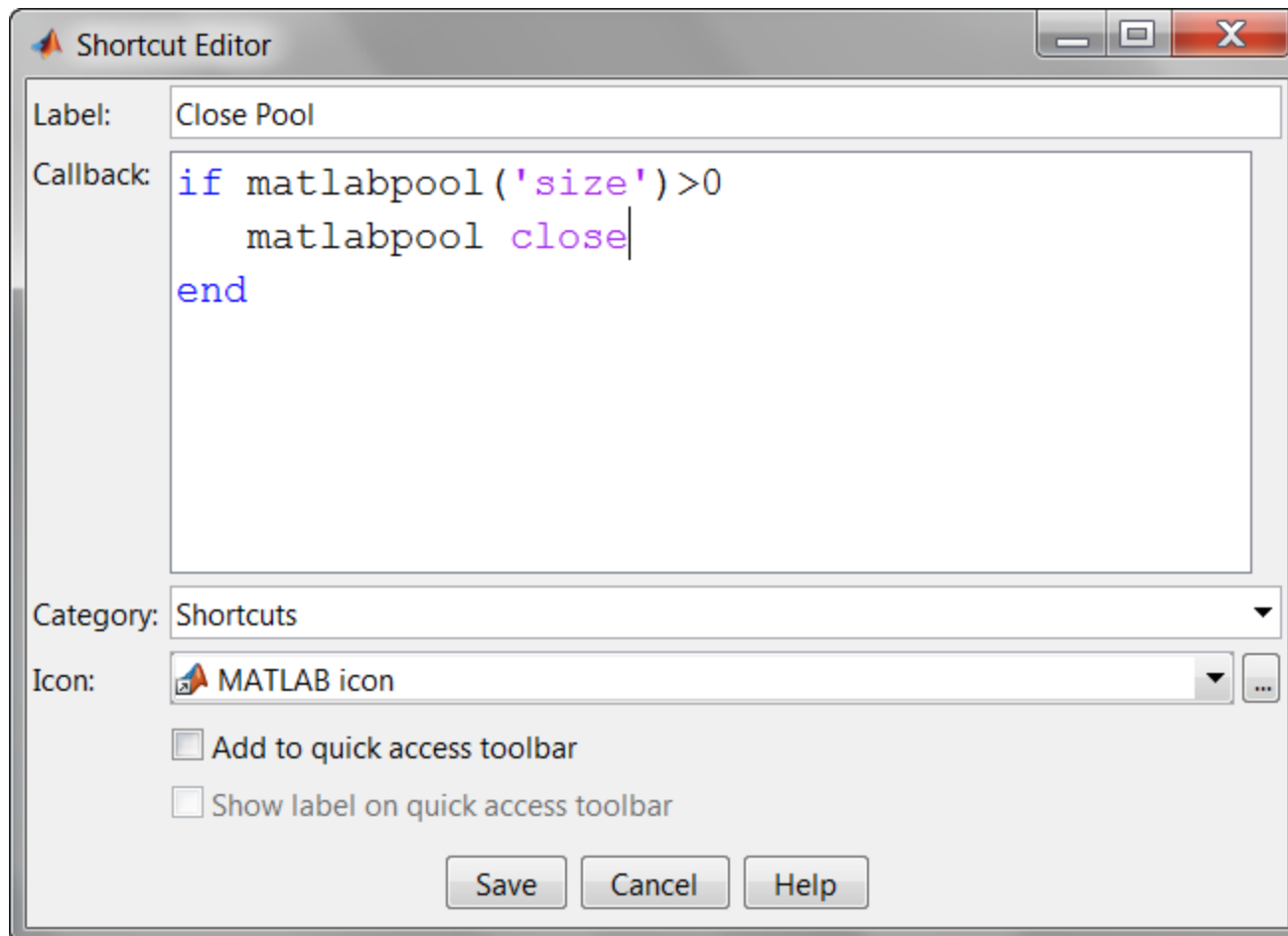
Caused by:
Error using parallel.internal.pool.InteractiveClient/start (line 206)
Found an interactive session. You cannot have multiple interactive
sessions open simultaneously. To terminate the existing session, use
'matlabpool close'.

fx >>
  
```

# Add Shortcut for starting a matlabpool



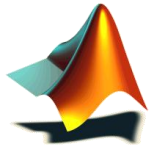
# Add Shortcut for stopping the matlabpool





# Agenda

- Best practices in MATLAB programming



## Introduction to parallel computing tools

- Constructs for multicore/multi-processor computers
- Scaling up to a cluster

# Programming Parallel Applications (CPU)



Ease of Use

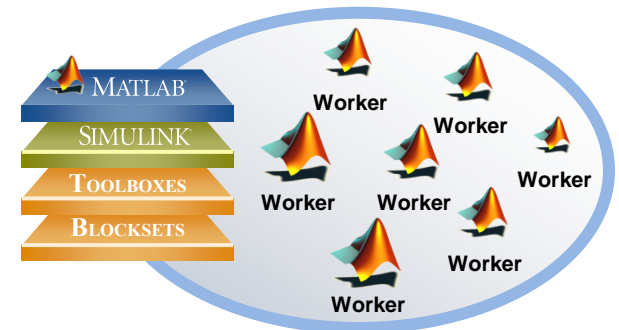
- Built-in support with Toolboxes



Greater Control

# Tools Providing Parallel Computing Support

- Optimization Toolbox, Global Optimization Toolbox
- Statistics Toolbox
- Signal Processing Toolbox
- Neural Network Toolbox
- Image Processing Toolbox
- ...



***Directly leverage functions in Parallel Computing Toolbox***

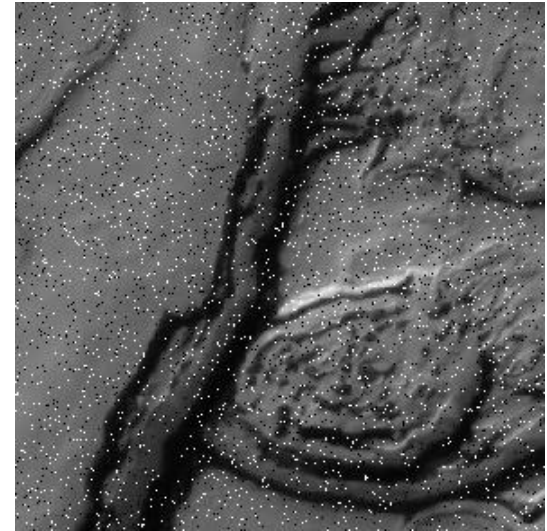
[www.mathworks.com/products/parallel-computing/builtin-parallel-support.html](http://www.mathworks.com/products/parallel-computing/builtin-parallel-support.html)

# Exercise: Filtering an Image

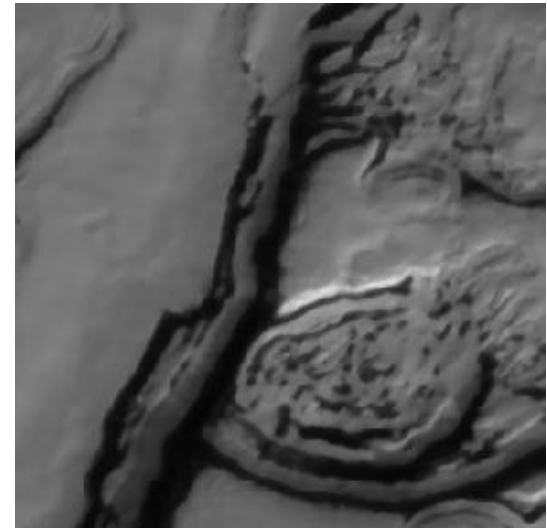
## Built-in parallel support

- With Parallel Computing Toolbox use built-in parallel algorithms in Image Processing Toolbox
- Run median filtering in parallel
- Use pool of MATLAB workers

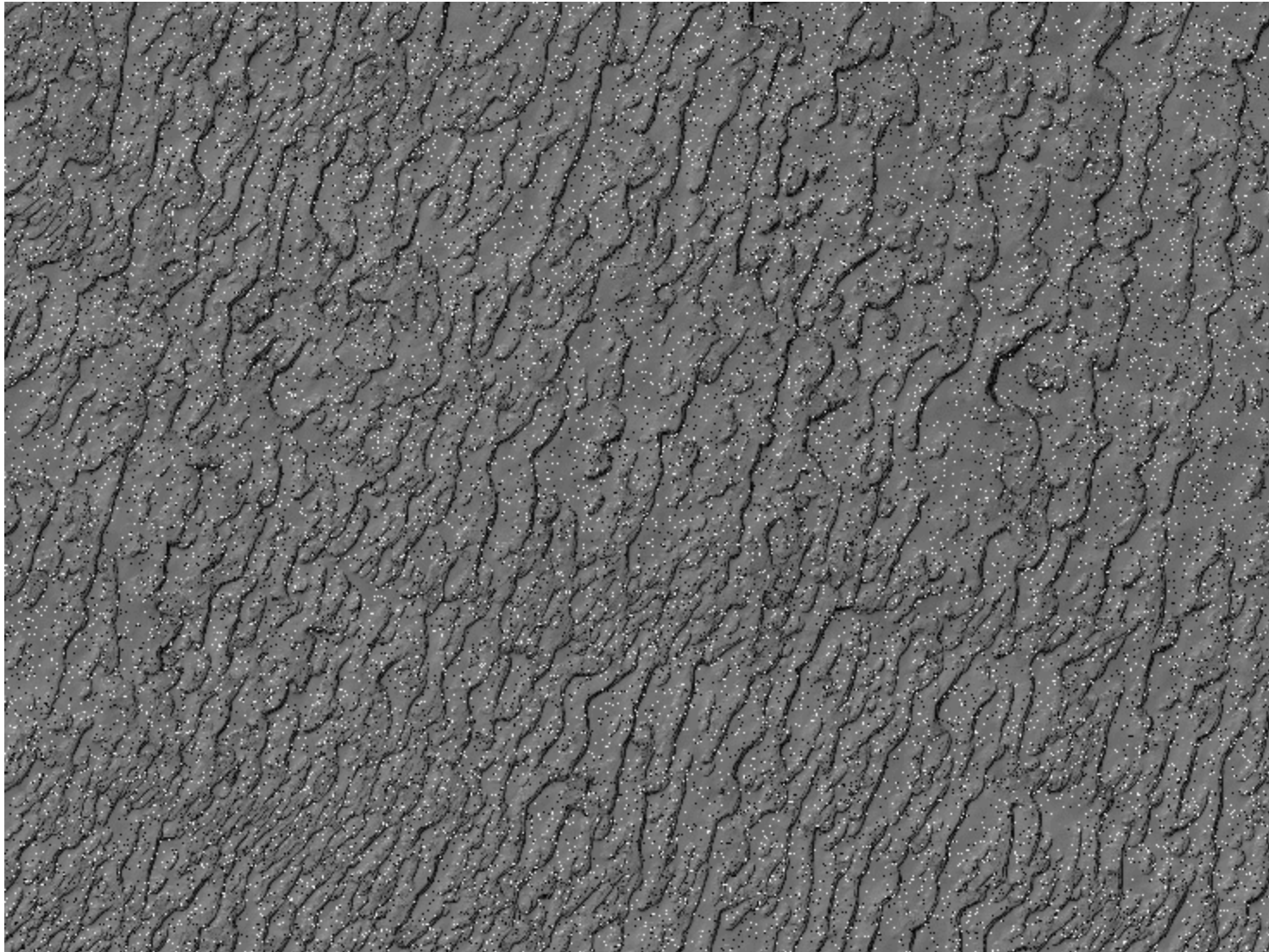
Noisy Image



Filtered Image



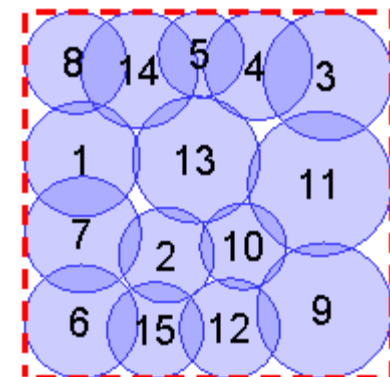
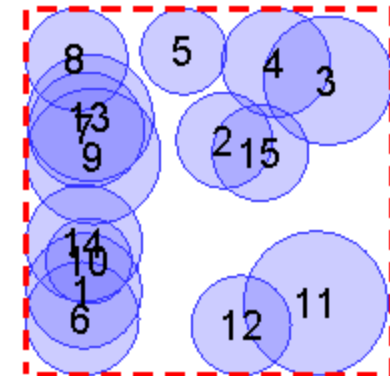
## 2D Median Filter



# Exercise: Optimizing Cell Tower Position

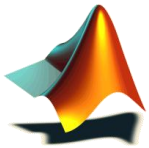
Built-in parallel support

- With Parallel Computing Toolbox use built-in parallel algorithms in Optimization Toolbox
- Run optimization in parallel
- Use pool of MATLAB workers



# Agenda

- Best practices in MATLAB programming
- Introduction to parallel computing tools



Constructs for multicore/multi-processor computers

- Scaling up to a cluster

# Programming Parallel Applications (CPU)



Ease of Use

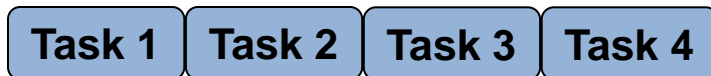
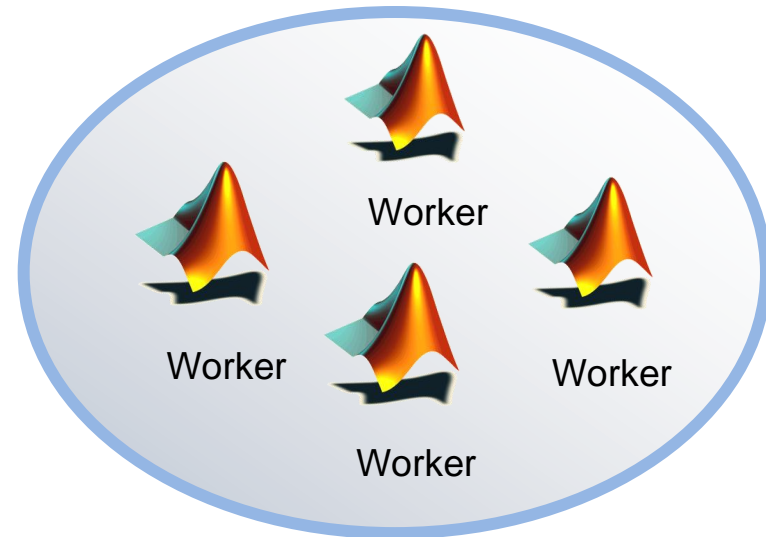
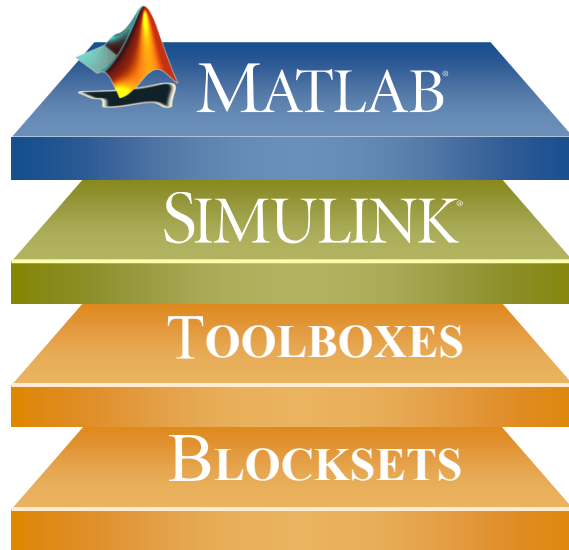
- Built-in support with Toolboxes
- Simple programming constructs:  
`parfor`, `batch`, `distributed`



Greater Control



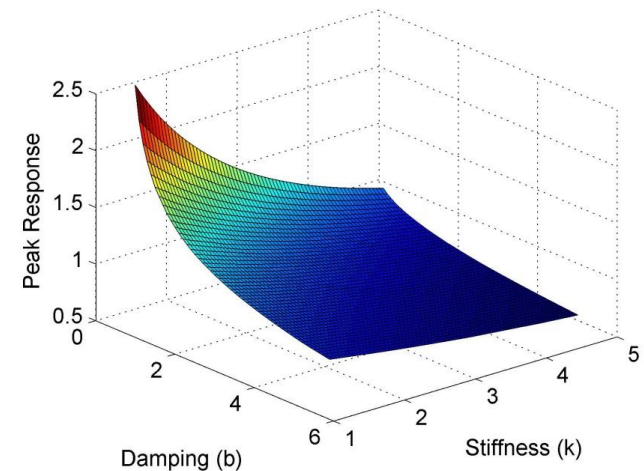
# Parallelizing independent tasks



## Exercise: Parallel for-loops

- Parameter sweep of ODE system
  - Damped spring oscillator
  - Sweep through different values of damping and stiffness
  - Record peak value for each simulation
- Convert **for** to **parfor**
- Use pool of MATLAB workers

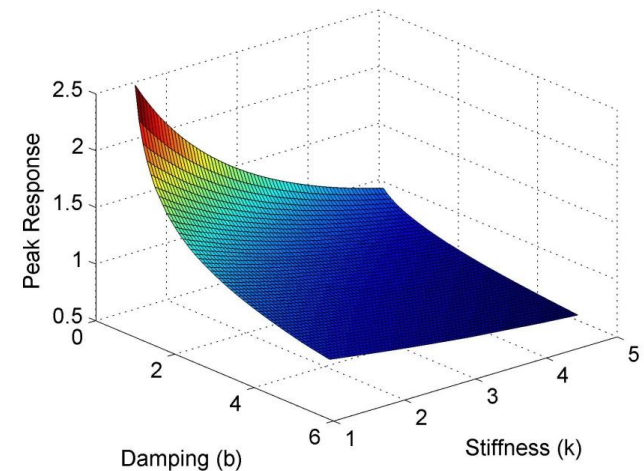
$$\overset{5}{m}\ddot{x} + \underset{1,2,\dots}{b}\dot{x} + \underset{1,2,\dots}{k}x = 0$$



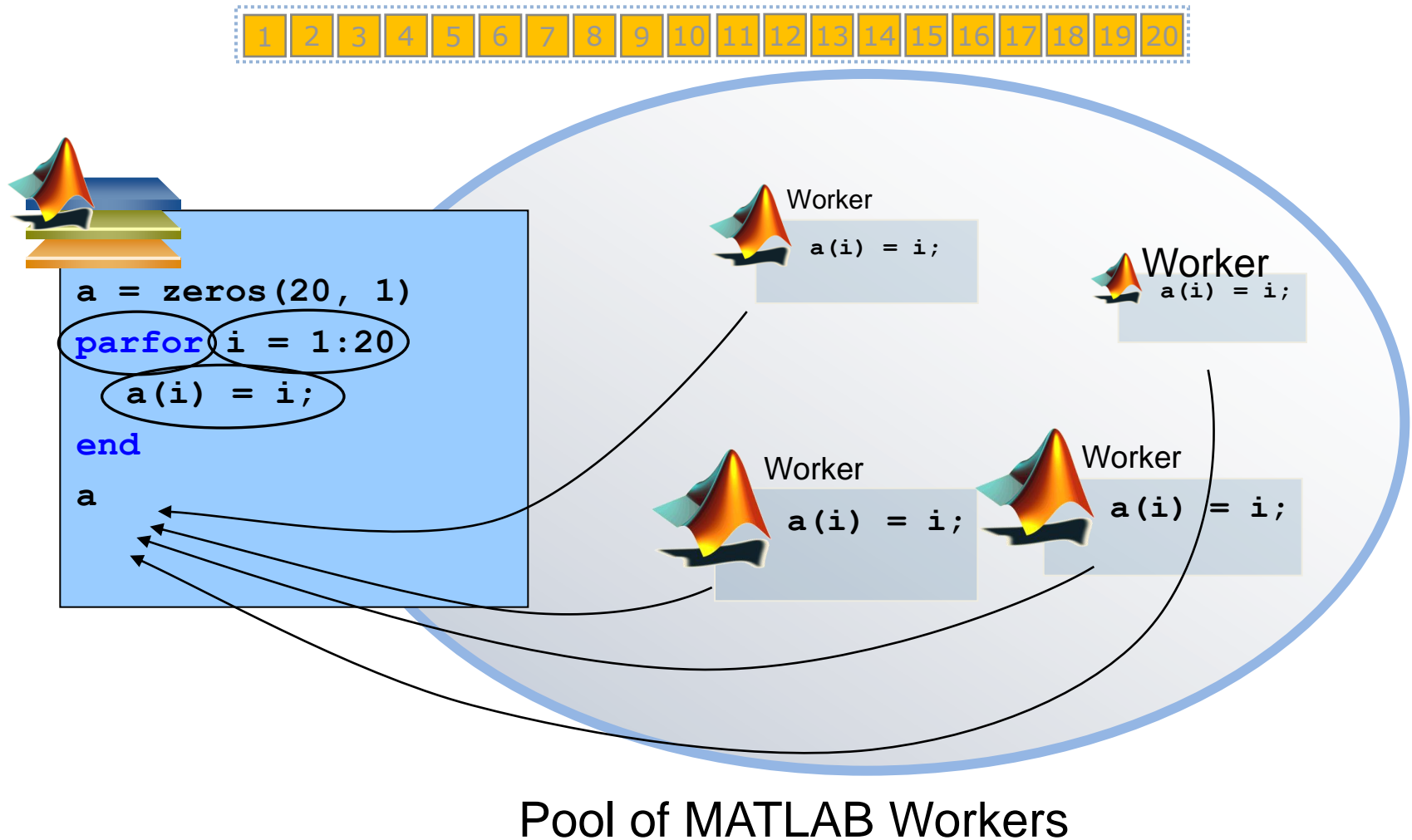
# Summary: Parallel for-loops Exercise

- Use parallel constructs and serial code in the same function
- Utilize pool of MATLAB Workers
- Used M-Lint analysis to help convert existing **for**-loop to **parfor**-loop

$$\overbrace{m}^5 \ddot{x} + \underbrace{b}_{1,2,\dots} \dot{x} + \underbrace{k}_{1,2,\dots} x = 0$$



# The Mechanics of parfor Loops



# Converting `for` to `parfor`

- Requirements for `parfor` loops
  - Task independent
  - Order independent
- Constraints on the loop body
  - Cannot “introduce” variables (e.g. `eval`, `load`, `global`, etc.)
  - Cannot contain `break` or `return` statements
  - Cannot contain another `parfor` loop

# Advice for Converting `for` to `parfor`

- Use the Code Analyzer to diagnose **`parfor`** issues
- If your `for` loop cannot be converted to a **`parfor`**, consider wrapping a subset of the body to a function
- Read the section in the documentation on classification of variables

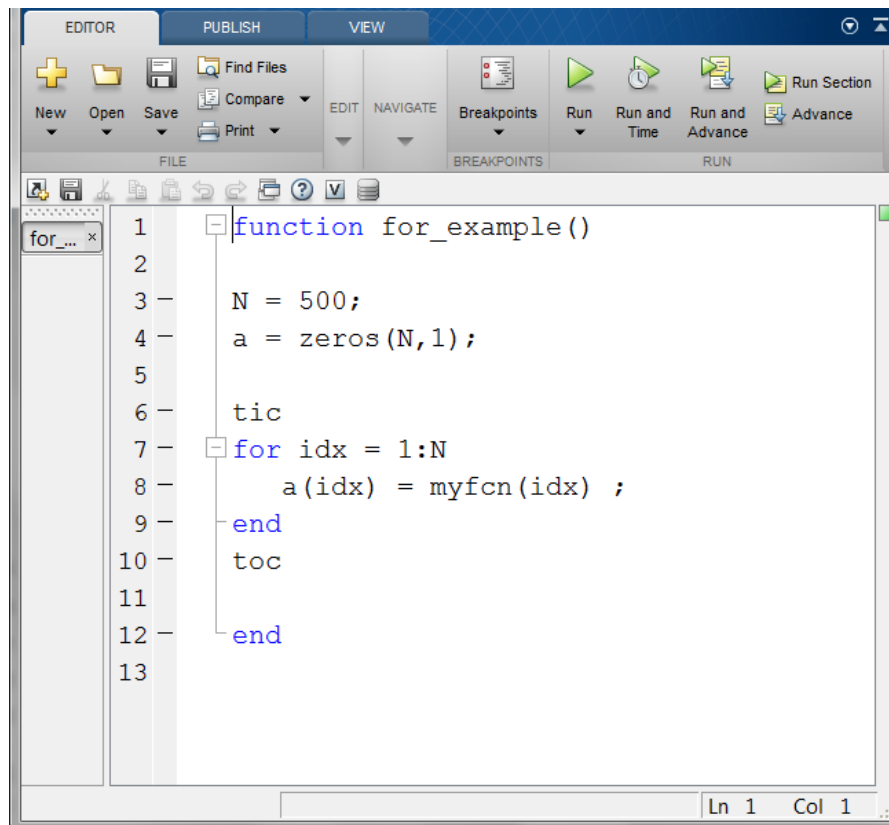
<http://blogs.mathworks.com/loren/2009/10/02/using-parfor-loops-getting-up-and-running/>

# Considerations when using `parfor`

- `parfor` automatically quits on error
- `parfor` doesn't provide intermediate results

# Exercise: parfor (1)

1. Code the example below. Save it as **for\_example.m**



The image shows a screenshot of the MATLAB Editor window. The window has a menu bar with 'EDITOR', 'PUBLISH', and 'VIEW'. Below the menu bar is a toolbar with icons for 'New', 'Open', 'Save', 'Find Files', 'Compare', 'Print', 'Breakpoints', 'Run', 'Run and Time', 'Run and Advance', and 'Run Section'. The main editing area shows the following code:

```
1 function for_example()  
2  
3     N = 500;  
4     a = zeros(N,1);  
5  
6     tic  
7     for idx = 1:N  
8         a(idx) = myfcn(idx);  
9     end  
10    toc  
11  
12    end  
13
```

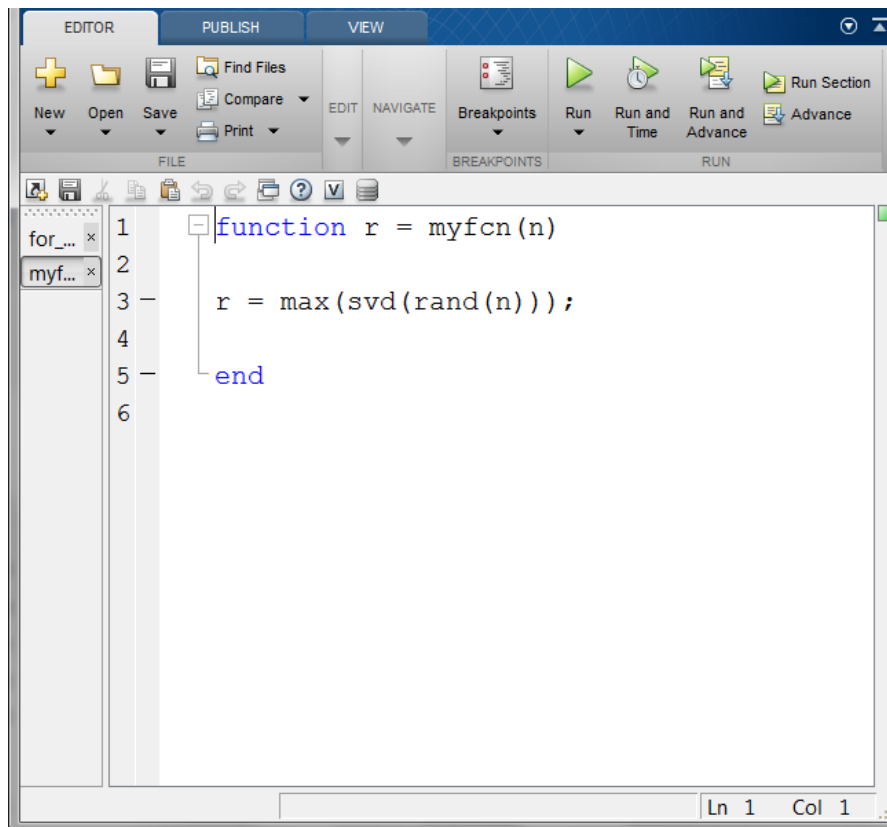
The status bar at the bottom right indicates 'Ln 1 Col 1'.

```
>> for_example
```



## Exercise: parfor (2)

2. Code the above helper function. Save it as **myfcn.m**



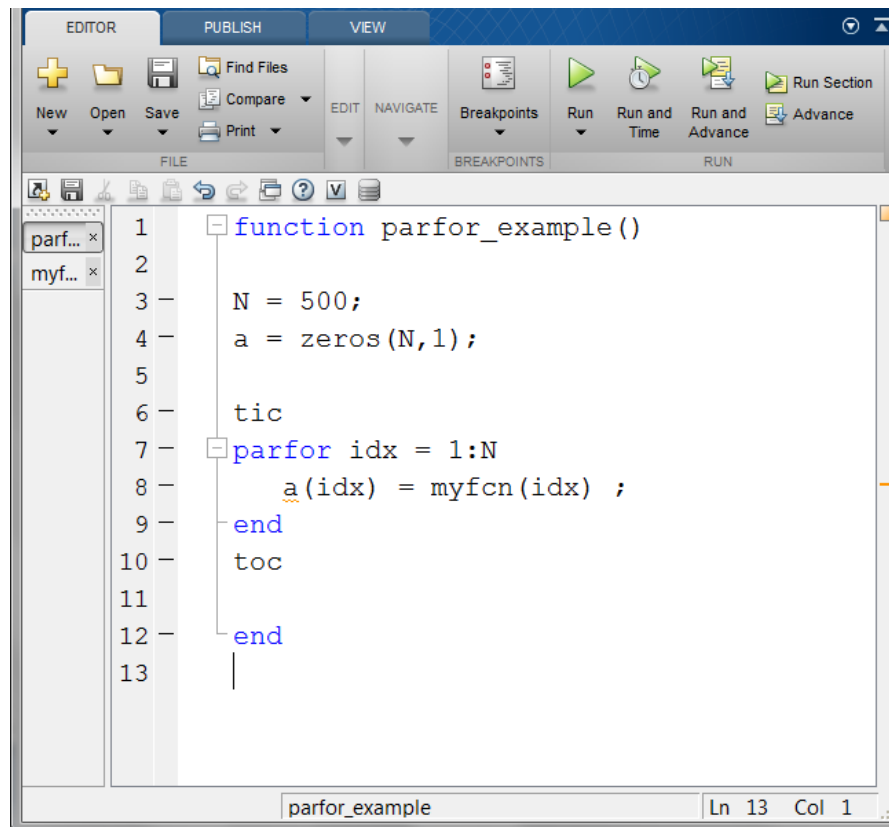
```
>> myfcn
```

## Exercise: `parfor` (3)

3. Convert the `for` to a `parfor`
  - save it as `parfor_example.m`
  - run `parfor_example`
4. Start a `matlabpool` with two workers and run `parfor_example`

```
>> parfor_example
```

## Exercise: parfor (3)



The image shows a screenshot of the MATLAB Editor window. The window has a menu bar with 'EDITOR', 'PUBLISH', and 'VIEW'. Below the menu bar is a toolbar with icons for 'New', 'Open', 'Save', 'Find Files', 'Compare', 'Print', 'Breakpoints', 'Run', 'Run and Time', 'Run and Advance', and 'Run Section'. The main editing area shows a function named 'parfor\_example()' with the following code:

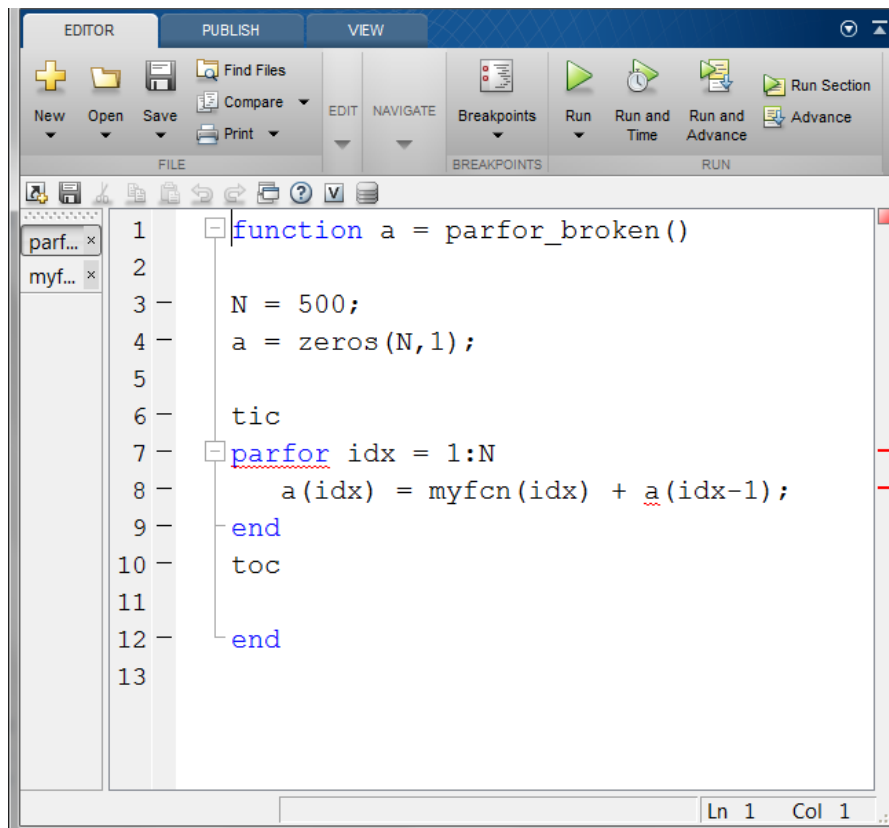
```
1 function parfor_example()
2
3     N = 500;
4     a = zeros(N,1);
5
6     tic
7     parfor idx = 1:N
8         a(idx) = myfcn(idx);
9     end
10    toc
11
12 end
13
```

The status bar at the bottom indicates the file name 'parfor\_example' and the current position 'Ln 13 Col 1'.

```
>> parfor_example
```

## Exercise: parfor (4)

5. Break the **parfor** loop, see Code Analyzer messages



The image shows the MATLAB Editor interface. The menu bar includes EDITOR, PUBLISH, and VIEW. The toolbar contains icons for New, Open, Save, Find Files, Compare, Print, Breakpoints, Run, Run and Time, Run and Advance, and Run Section. The main editor window displays the following code:

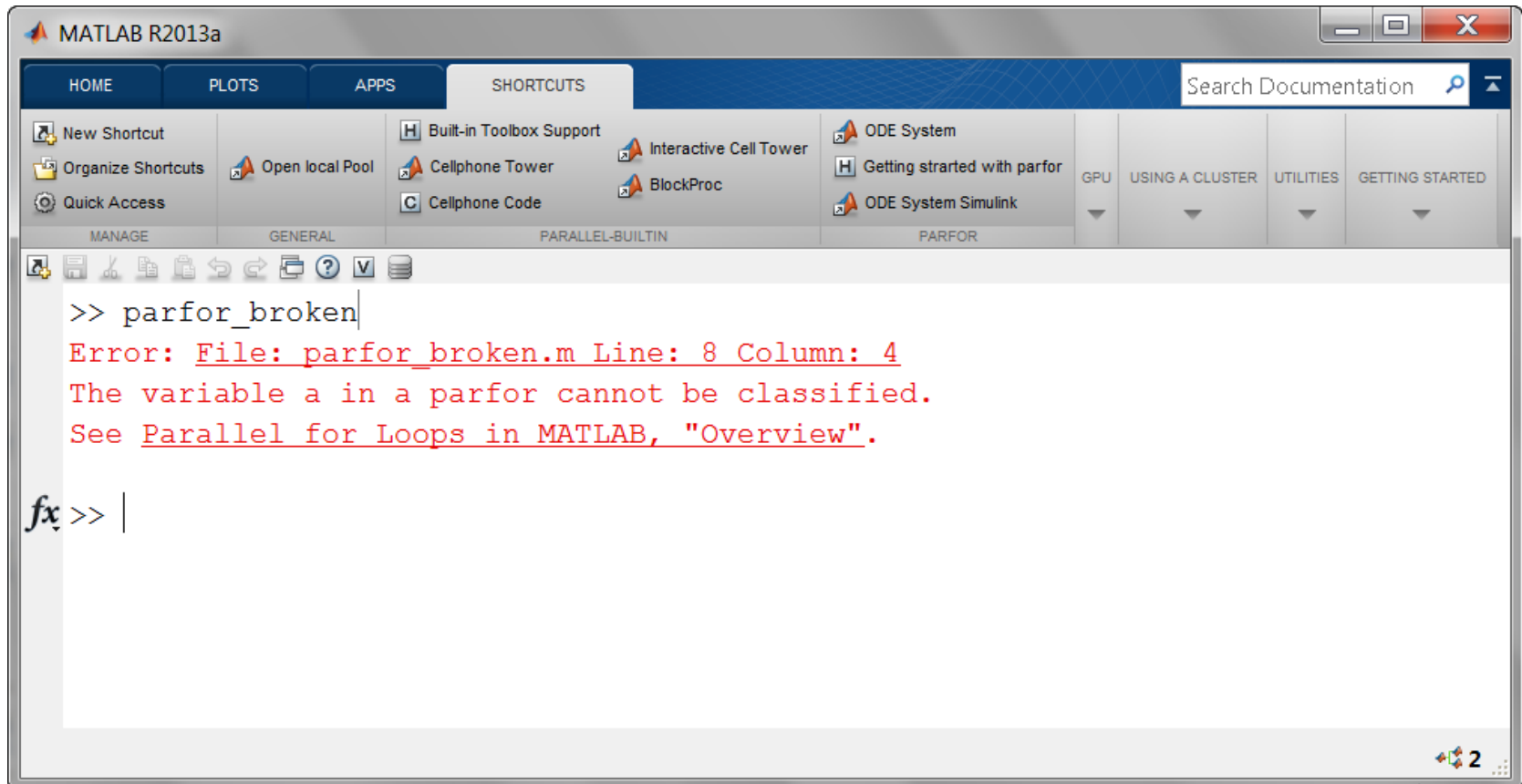
```
1 function a = parfor_broken()
2
3     N = 500;
4     a = zeros(N,1);
5
6     tic
7     parfor idx = 1:N
8         a(idx) = myfcn(idx) + a(idx-1);
9     end
10    toc
11
12 end
13
```

The status bar at the bottom indicates the cursor is at Line 1, Column 1.

```
>> parfor_broken
```

# Problem: Unclassified Variables

- The variable ***a*** cannot be properly classified



The screenshot shows the MATLAB R2013a interface. The command window displays the following error message:

```
>> parfor_broken|
Error: File: parfor_broken.m Line: 8 Column: 4
The variable a in a parfor cannot be classified.
See Parallel for Loops in MATLAB, "Overview".
```

Below the error message, the prompt *fx* is visible, followed by the command window prompt `>> |`.

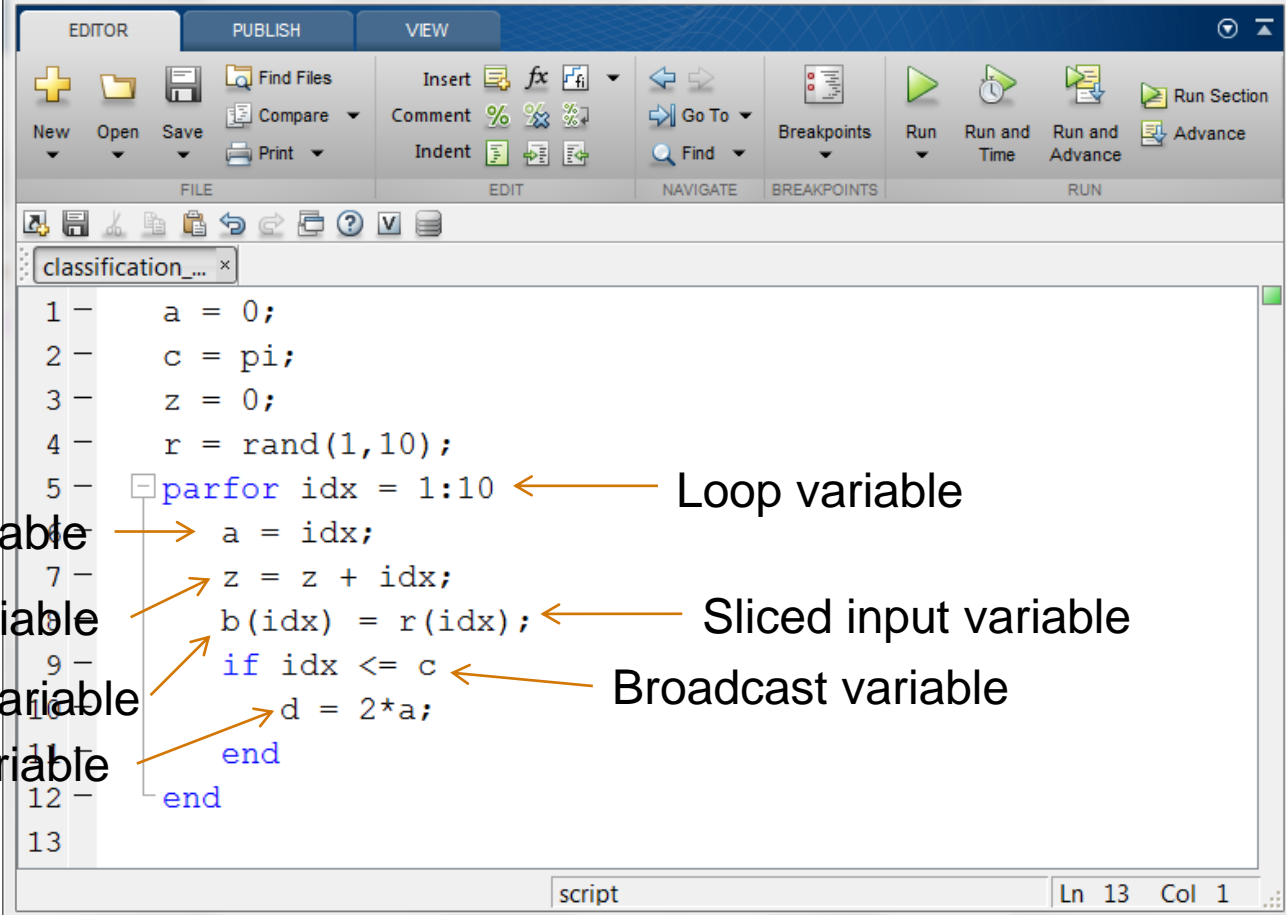
# parfor Variable Classification

- All variables referenced at the top level of the **parfor** must be resolved and classified

| Classification | Description  |
|----------------|--|
| Loop           | Serves as a loop index for arrays  |
| Sliced         | An array whose segments are operated on by different iterations of the loop                                |
| Broadcast      | A variable defined before the loop whose value is used inside the loop, but never assigned inside the loop |
| Reduction      | Accumulates a value across iterations of the loop, regardless of iteration order                           |
| Temporary      | Variable created inside the loop, but unlike sliced or reduction variables, not available outside the loop |

[http://www.mathworks.com/help/distcomp/advanced-topics.html#bq\\_of7\\_-1](http://www.mathworks.com/help/distcomp/advanced-topics.html#bq_of7_-1)

# Exercise: Variable Classification Example (1)



The image shows a MATLAB Editor window with a script named 'classification\_...'. The script contains the following code:

```

1 - a = 0;
2 - c = pi;
3 - z = 0;
4 - r = rand(1,10);
5 - parfor idx = 1:10
6 -     a = idx;
7 -     z = z + idx;
8 -     b(idx) = r(idx);
9 -     if idx <= c
10 -        d = 2*a;
11 -     end
12 - end
13

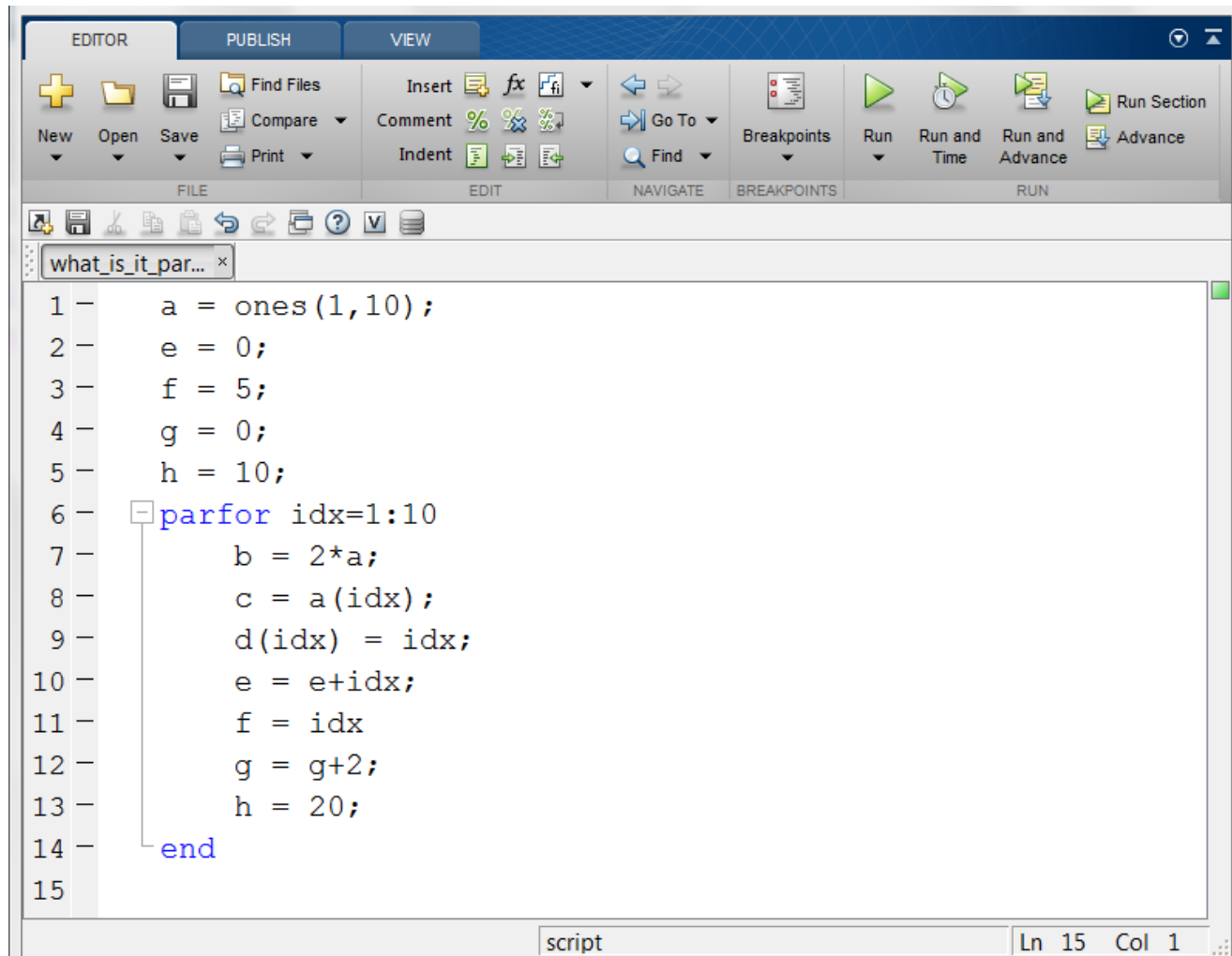
```

Annotations with arrows point to specific variables in the code:

- Loop variable:** points to `idx` in the `parfor` loop header.
- Temporary variable:** points to `a` in the assignment `a = idx;`.
- Reduction variable:** points to `z` in the accumulation `z = z + idx;`.
- Sliced input variable:** points to `r(idx)` in the assignment `b(idx) = r(idx);`.
- Sliced output variable:** points to `b(idx)` in the assignment `b(idx) = r(idx);`.
- Broadcast variable:** points to `c` in the conditional `if idx <= c`.
- Temporary variable:** points to `d` in the assignment `d = 2*a;`.

```
>> classification_example
```

# Exercise: Variable Classification Example (2)



The image shows a screenshot of the MATLAB Editor window. The window has a menu bar with 'EDITOR', 'PUBLISH', and 'VIEW' tabs. Below the menu bar is a toolbar with icons for file operations (New, Open, Save, Find Files, Compare, Print), editing (Insert, Comment, Indent), navigation (Go To, Find), breakpoints, and running (Run, Run and Time, Run and Advance). The main editor area shows a script with the following code:

```

1 - a = ones(1,10);
2 - e = 0;
3 - f = 5;
4 - g = 0;
5 - h = 10;
6 - parfor idx=1:10
7 -     b = 2*a;
8 -     c = a(idx);
9 -     d(idx) = idx;
10 -    e = e+idx;
11 -    f = idx
12 -    g = g+2;
13 -    h = 20;
14 - end
15

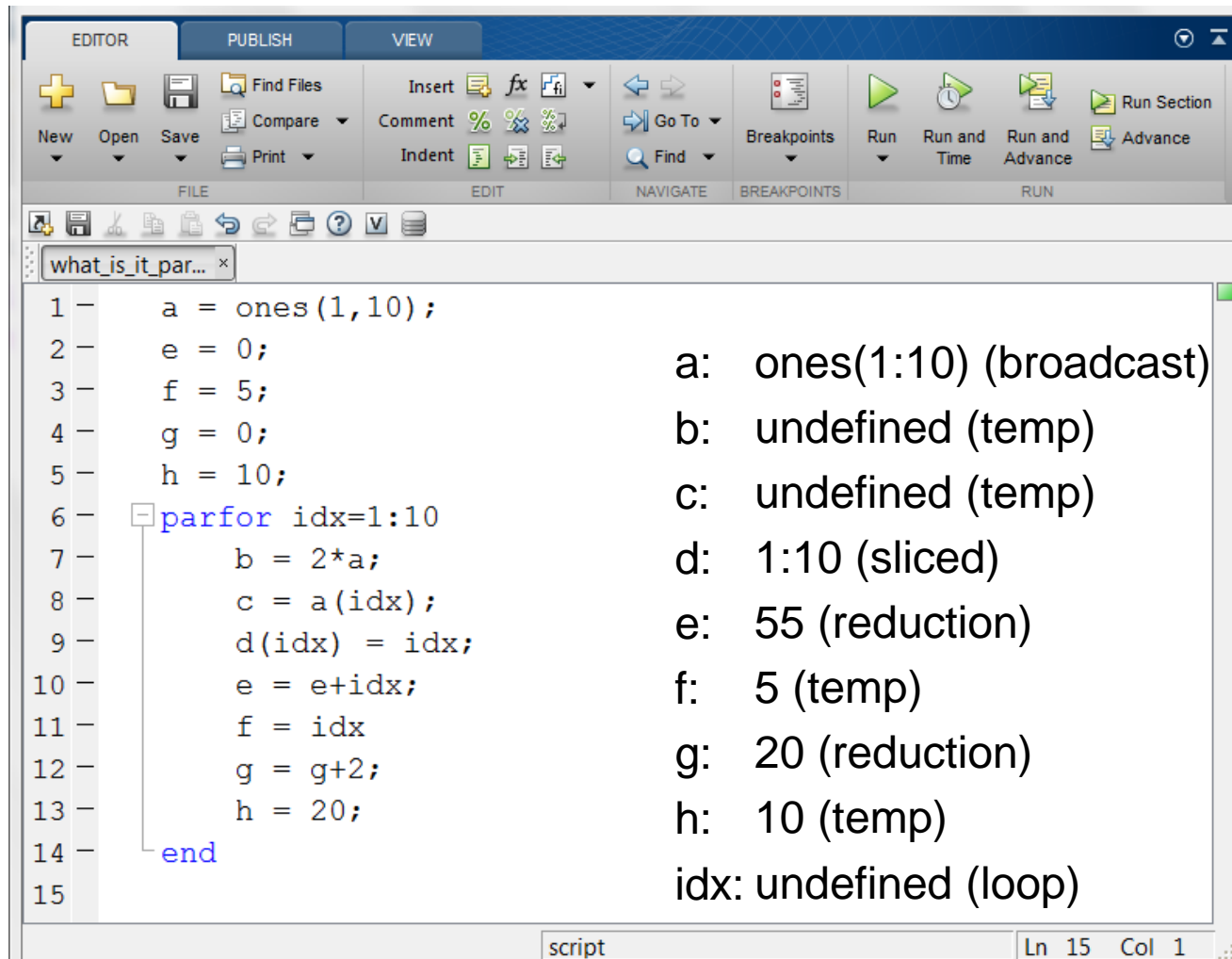
```

The status bar at the bottom indicates the file is named 'script' and the cursor is at line 15, column 1.

>> what\_is\_it\_parfor



# Exercise: Variable Classification Example (2)



The image shows a MATLAB Editor window with a script and its variable classification results. The script is as follows:

```

1 - a = ones(1,10);
2 - e = 0;
3 - f = 5;
4 - g = 0;
5 - h = 10;
6 - parfor idx=1:10
7 -     b = 2*a;
8 -     c = a(idx);
9 -     d(idx) = idx;
10 -    e = e+idx;
11 -    f = idx
12 -    g = g+2;
13 -    h = 20;
14 - end
15

```

To the right of the script, the variable classification results are listed:

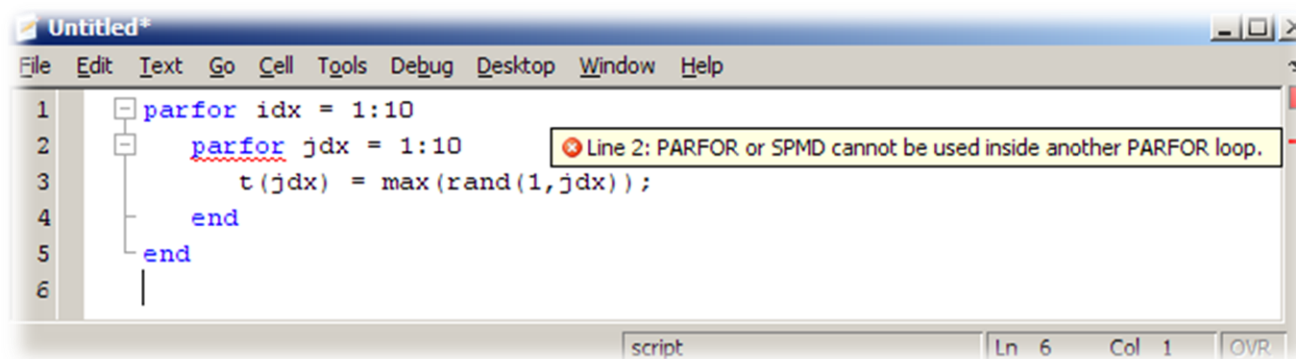
- a: ones(1:10) (broadcast)
- b: undefined (temp)
- c: undefined (temp)
- d: 1:10 (sliced)
- e: 55 (reduction)
- f: 5 (temp)
- g: 20 (reduction)
- h: 10 (temp)
- idx: undefined (loop)

The status bar at the bottom of the editor shows "script" and "Ln 15 Col 1".

>> what\_is\_it\_parfor

# Exercise: parfor within parfor

- MATLAB runs a static analyzer on the immediate **parfor** and will error out nested **parfor** loops. However, functions called from within the **parfor** that include **parfor** loops are run serially.

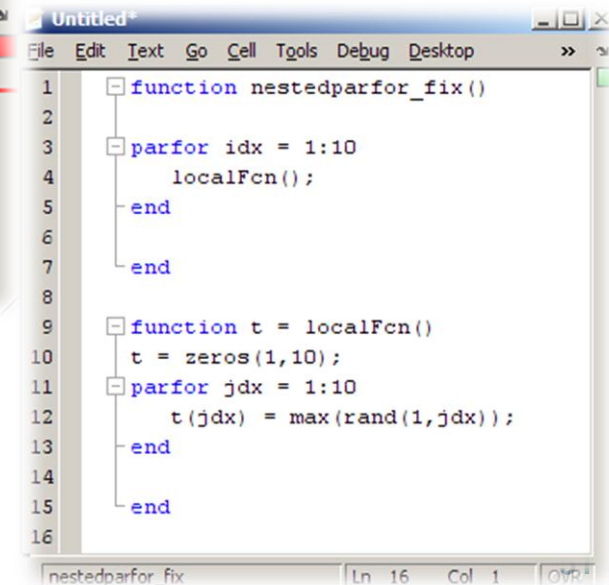


```

1  parfor idx = 1:10
2      parfor jdx = 1:10
3          t(jdx) = max(rand(1,jdx));
4      end
5  end
6

```

Line 2: PARFOR or SPMD cannot be used inside another PARFOR loop.



```

1  function nestedparfor_fix()
2
3      parfor idx = 1:10
4          localFcn();
5      end
6
7  end
8
9  function t = localFcn()
10     t = zeros(1,10);
11     parfor jdx = 1:10
12         t(jdx) = max(rand(1,jdx));
13     end
14 end
15
16

```

```

>> nestedparfor_bug
>> nestedparfor_fix

```

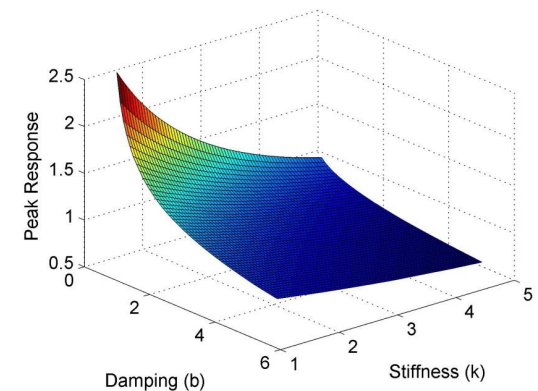
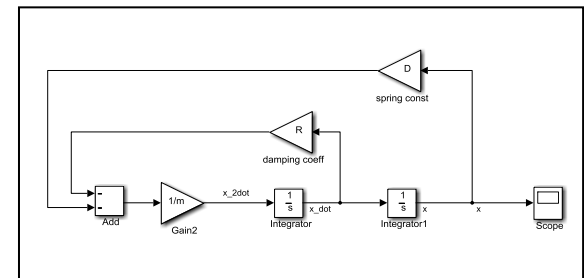
# Using parfor with Simulink

- Can use `parfor` with `sim`.
- Must make sure that the Simulink workspace contains the variables you want to use.
- Within main `parfor` body: Use `'base'` workspace
- Use `assignin` to place variables in base workspace.
- Note: the base workspace when using `parfor` is different than the base workspace when running serially.

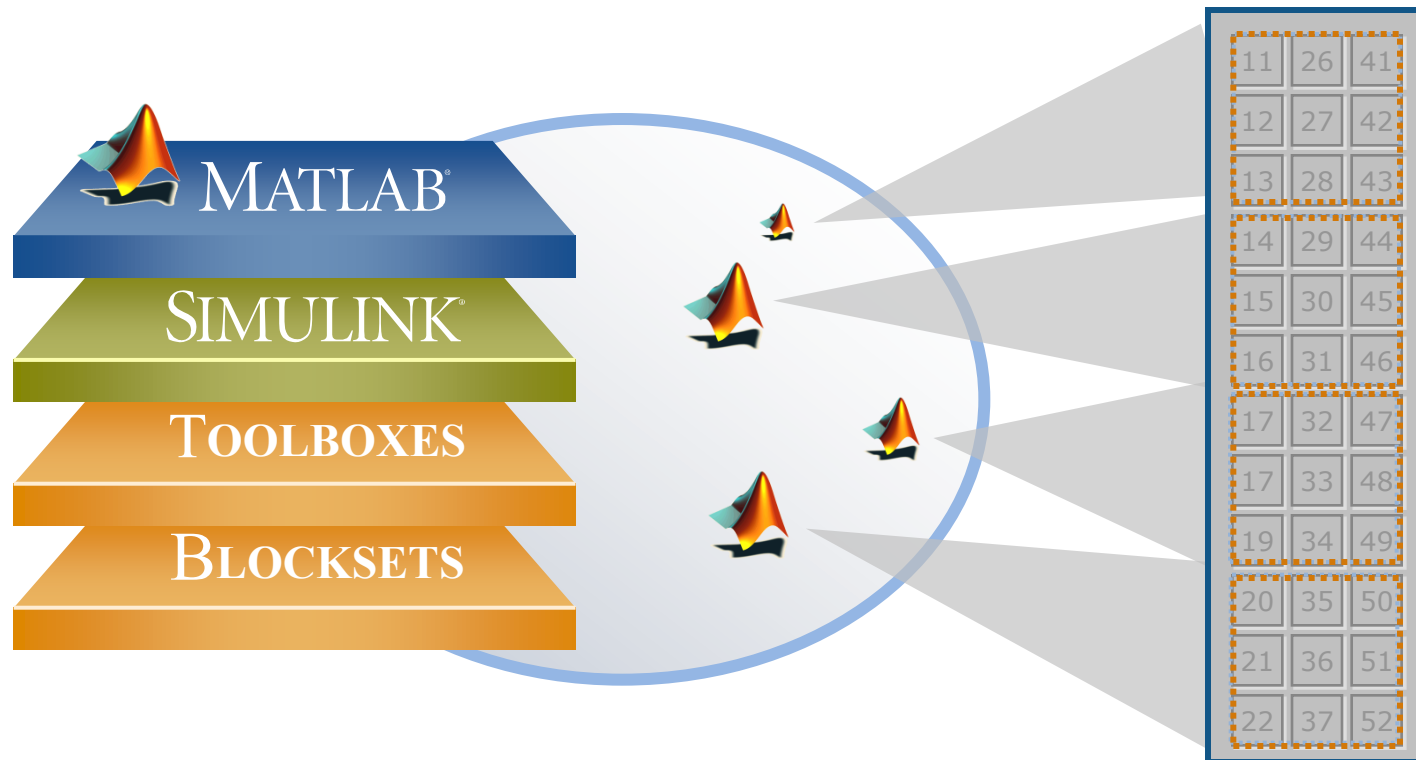
# Exercise: Parallel for-loops with Simulink

- Parameter sweep of ODE system
  - Damped spring oscillator in Simulink
  - Sweep through different values of damping and stiffness
  - Record peak value for each simulation
- Convert **for** to **parfor**
- Use pool of MATLAB workers

$$\underbrace{5}_m \ddot{x} + \underbrace{b}_{1,2,\dots} \dot{x} + \underbrace{k}_{1,2,\dots} x = 0$$



# Distributing Large Data



Remotely Manipulate Array  
from Client MATLAB

Distributed Array  
Lives on the Workers

## Regular MATLAB code

```
function benchmark_orig()  
    %% Create distributed arrays  
  
    A = rand(10000,10000);  
    b = rand(10000, 1);  
  
    %% Time solution of Ax = b  
    tic,  
  
    x = A\b;  
  
    t = toc;  
  
    %% Compute Gflops  
    gflops = (2/3*10000^3 + 3/2*10000^2) / t / 1e9 ;  
end
```

## Using Distributed Arrays

```
function benchmark()  
    %% Create distributed arrays  
  
    A = distributed.rand(10000,10000);  
    b = distributed.rand(10000, 1);  
  
    %% Time solution of Ax = b  
    tic,  
  
    x = A\b; % Parallel "\"  
  
    t = toc;  
  
    %% Compute Gflops  
    gflops = (2/3*10000^3 + 3/2*10000^2) / t / 1e9 ;  
end
```

# Using FORTRAN and MPI

```
+
+## F2C options
+
+F2CDEFS      := -DAdd__ -DF77_INTEGER=int -DStringSunStyle
+
+## HPL options
+
+HPL_DEFS := -D(HPLlib) -D(Lalib) -D(MPlib) -D(CSlib)
+
+CCNOOPT := -m64 -Wall $(HPL_DEFS)
+CCFLAGS := $(CCNOOPT) -O3 -fomit-frame-pointer -funroll-loops
+##CCFLAGS := $(CCNOOPT) -O0 -ggdb -g3
+LINKFLAGS := $(CCFLAGS)
+ARFLAGS := -r
+
+Index: Make.qs22
=====
RCS file: Make.qs22
diff -N Make.qs22
--- /dev/null 1 Jan 1970 00:00:00 -0000
+++ Make.qs22 20 Aug 2008 03:57:53 -0000 1.7
@@ -0,0 +1,74 @@
+#####
+## (C) Copyright IBM Corporation 2008
+##
+#####
+
+## Platform
+
+ARCH := qs22
+
+## Tools
+
+SHELL := /bin/sh
+CD := cd
+CP := cp
+LN_S := ln -s
+MKDIR := mkdir
+TOUCH := touch
+
+CC := mpicc
+LINKER := mpicc
+ARCHIVER := /usr/bin/ar
+RANLIB := echo
+
+## Directories
+
+INCdir := $(TOPdir)/include
+BINDir := $(TOPdir)/bin/$(ARCH)
+
+## HPL library
+
+HPLlib := $(TOPdir)/lib/$(ARCH)/libhpl.a
+ACCLib := $(TOPdir)/accel/lib/libhpl_accel_ppu.a
+
+## MPI package
```

# Using Distributed Arrays

```
function benchmark()

%% Create distributed arrays

A = distributed.rand(10000,10000);
b = distributed.rand(10000, 1);

%% Time solution of Ax = b
tic,

x = A\b; % Parallel "\"

t = toc;

%% Compute Gflops
gflops = (2/3*10000^3 + 3/2*10000^2) / t / 1e9 ;

end
```

# Creating Distributed Arrays

```
DI = distributed.eye(4)
```

|       |         |
|-------|---------|
| cell  | rand    |
| eye   | randn   |
| ones  | spalloc |
| zeros | sparse  |
| false | speye   |
| true  | sprand  |
| NaN   |         |
| Inf   |         |

See also:

- `codistributed`, `codistributor1d`, `codistributor2dbc`



# Enhanced MATLAB functions that operate on codistributed arrays

| Type of Function                       | Function Names  |
|--|---|
| Data functions                         | <a href="#">arrayfun</a> , <a href="#">bsxfun</a> , <a href="#">cumprod</a> , <a href="#">cumsum</a> , <a href="#">fft</a> , <a href="#">max</a> , <a href="#">min</a> , <a href="#">prod</a> , <a href="#">sum</a>   |
| Data type functions                    | <a href="#">cast</a> , <a href="#">cell2mat</a> , <a href="#">cell2struct</a> , <a href="#">celldisp</a> , <a href="#">cellfun</a> , <a href="#">char</a> , <a href="#">double</a> , <a href="#">fieldnames</a> , <a href="#">int16</a> , <a href="#">int32</a> , <a href="#">int64</a> , <a href="#">int8</a> , <a href="#">logical</a> , <a href="#">num2cell</a> , <a href="#">rmfield</a> , <a href="#">single</a> , <a href="#">struct2cell</a> , <a href="#">swapbytes</a> , <a href="#">typecast</a> , <a href="#">uint16</a> , <a href="#">uint32</a> , <a href="#">uint64</a> , <a href="#">uint8</a>  |
| Elementary and trigonometric functions | <a href="#">abs</a> , <a href="#">acos</a> , <a href="#">acosd</a> , <a href="#">acosh</a> , <a href="#">acot</a> , <a href="#">acotd</a> , <a href="#">acoth</a> , <a href="#">acsc</a> , <a href="#">acscd</a> , <a href="#">acsch</a> , <a href="#">angle</a> , <a href="#">asec</a> , <a href="#">asecd</a> , <a href="#">asech</a> , <a href="#">asin</a> , <a href="#">asind</a> , <a href="#">asinh</a> , <a href="#">atan</a> , <a href="#">atan2</a> , <a href="#">atan2d</a> , <a href="#">atand</a> , <a href="#">atanh</a> , <a href="#">ceil</a> , <a href="#">complex</a> , <a href="#">conj</a> , <a href="#">cos</a> , <a href="#">cosd</a> , <a href="#">cosh</a> , <a href="#">cot</a> , <a href="#">cotd</a> , <a href="#">coth</a> , <a href="#">csc</a> , <a href="#">cscd</a> , <a href="#">csch</a> , <a href="#">exp</a> , <a href="#">expm1</a> , <a href="#">fix</a> , <a href="#">floor</a> , <a href="#">hypot</a> , <a href="#">imag</a> , <a href="#">isreal</a> , <a href="#">log</a> , <a href="#">log10</a> , <a href="#">log1p</a> , <a href="#">log2</a> , <a href="#">mod</a> , <a href="#">nextpow2</a> , <a href="#">nthroot</a> , <a href="#">pow2</a> , <a href="#">real</a> , <a href="#">reallog</a> , <a href="#">realpow</a> , <a href="#">realsqrt</a> , <a href="#">rem</a> , <a href="#">round</a> , <a href="#">sec</a> , <a href="#">secd</a> , <a href="#">sech</a> , <a href="#">sign</a> , <a href="#">sin</a> , <a href="#">sind</a> , <a href="#">sinh</a> , <a href="#">sqrt</a> , <a href="#">tan</a> , <a href="#">tand</a> , <a href="#">tanh</a> |
| Elementary matrices                    | <a href="#">cat</a> , <a href="#">diag</a> , <a href="#">eps</a> , <a href="#">find</a> , <a href="#">isempty</a> , <a href="#">isequal</a> , <a href="#">isequaln</a> , <a href="#">isfinite</a> , <a href="#">isinf</a> , <a href="#">isnan</a> , <a href="#">length</a> , <a href="#">meshgrid</a> , <a href="#">ndgrid</a> , <a href="#">ndims</a> , <a href="#">numel</a> , <a href="#">repmat</a> , <a href="#">reshape</a> , <a href="#">size</a> , <a href="#">sort</a> , <a href="#">tril</a> , <a href="#">triu</a>   |
| Matrix functions                       | <a href="#">chol</a> , <a href="#">eig</a> , <a href="#">inv</a> , <a href="#">lu</a> , <a href="#">norm</a> , <a href="#">normest</a> , <a href="#">qr</a> , <a href="#">svd</a>   |
| Array operations                       | <a href="#">all</a> , <a href="#">and</a> (&), <a href="#">any</a> , <a href="#">bitand</a> , <a href="#">bitor</a> , <a href="#">bitxor</a> , <a href="#">ctranspose</a> ('), <a href="#">end</a> , <a href="#">eq</a> (==), <a href="#">ge</a> (>=), <a href="#">gt</a> (>), <a href="#">horzcat</a> ([ ]), <a href="#">ldivide</a> (.\), <a href="#">le</a> (<=), <a href="#">lt</a> (<), <a href="#">minus</a> (-), <a href="#">mldivide</a> (\), <a href="#">mrdivide</a> (/), <a href="#">mtimes</a> (*), <a href="#">ne</a> (~=), <a href="#">not</a> (~), <a href="#">or</a> ( ), <a href="#">plus</a> (+), <a href="#">power</a> (.^), <a href="#">rdivide</a> ./), <a href="#">subsasgn</a> , <a href="#">subsindex</a> , <a href="#">subsref</a> , <a href="#">times</a> (.*), <a href="#">transpose</a> (.'), <a href="#">uminus</a> (-), <a href="#">uplus</a> (+), <a href="#">vertcat</a> ([;]), <a href="#">xor</a>   |
| Sparse matrix functions                | <a href="#">full</a> , <a href="#">issparse</a> , <a href="#">nnz</a> , <a href="#">nonzeros</a> , <a href="#">nzmax</a> , <a href="#">sparse</a> , <a href="#">spfun</a> , <a href="#">spones</a>  |
| Special functions                      | <a href="#">dot</a>   |

# Programming Parallel Applications (CPU)



Ease of Use

- Built-in support with Toolboxes
- Simple programming constructs:  
`parfor`, `batch`, `distributed`
- Advanced programming constructs:  
`createJob`, `labSend`, `spmd`

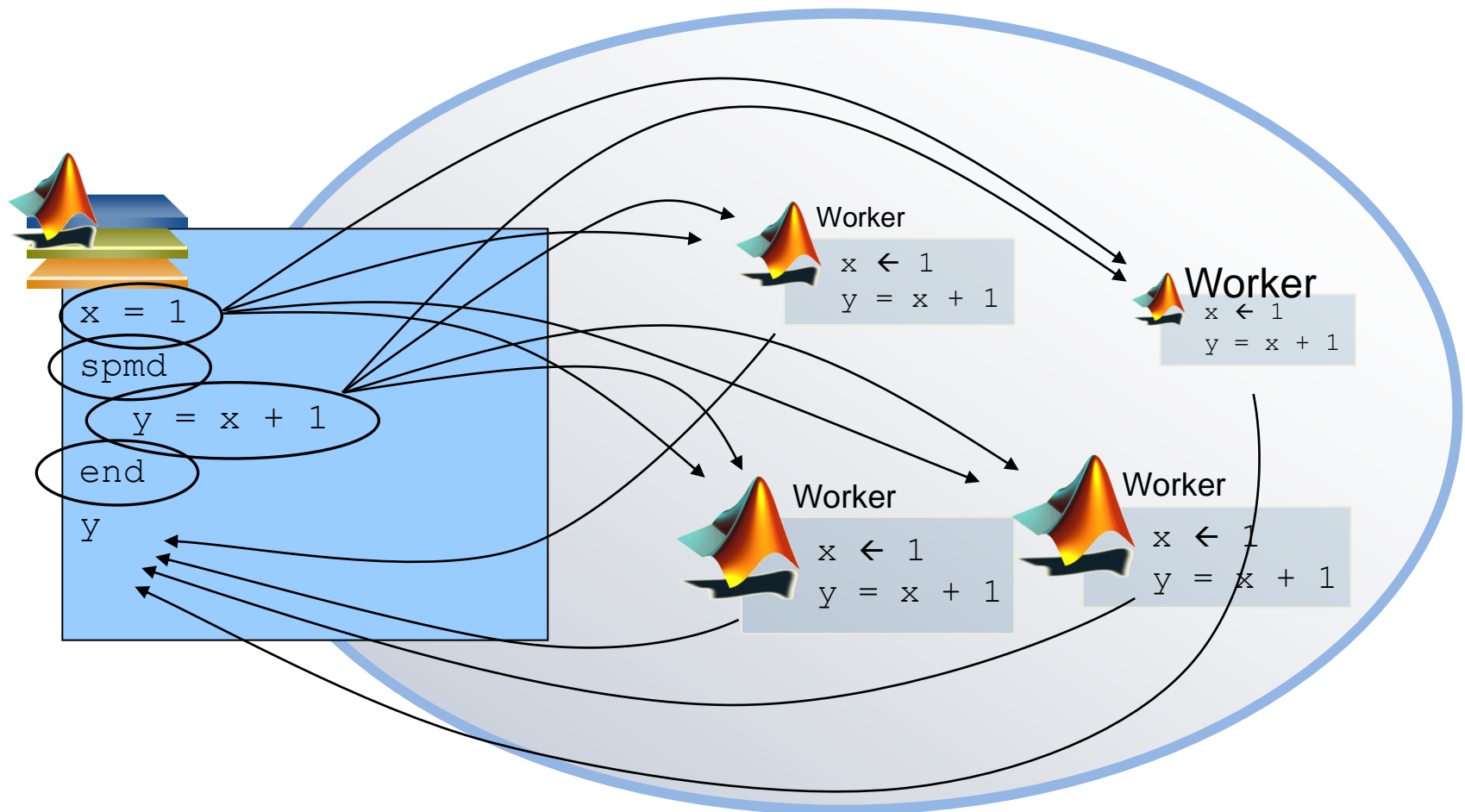


Greater Control

# Client-side Distributed Arrays and SPMD

- Client-side distributed arrays
  - Class `distributed`
  - Can be created and manipulated directly from the client.
  - Simpler access to memory on labs
  - Client-side visualization capabilities
  
- `spmd`
  - Block of code executed on workers
  - Worker specific commands
  - Explicit communication between workers
  - Mixture of parallel and serial code

# The Mechanics of `spmd` Blocks



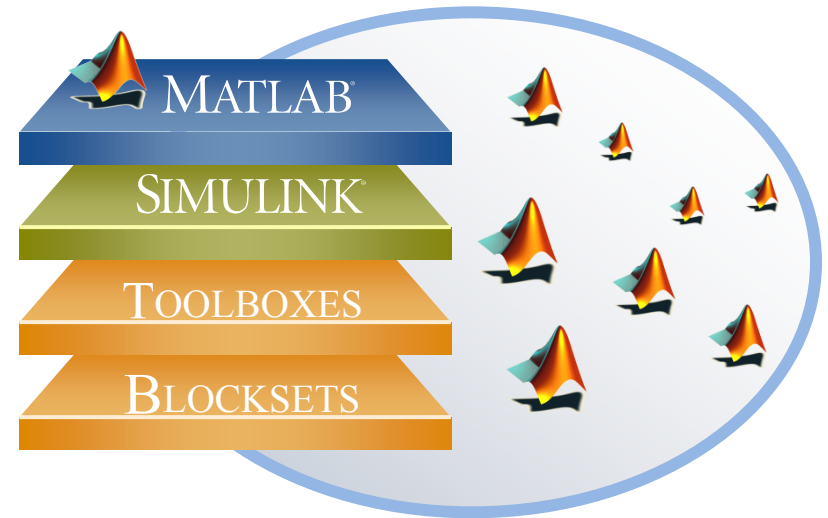
Pool of MATLAB Workers

## `spmd`

- single program, multiple data
- Unlike variables used in multiple `parfor` loops, distributed arrays used in multiple `spmd` blocks retain state
- Use Code Analyzer to diagnose `spmd` issues

# Composite Arrays

- Created from client
- Stored on workers
- Syntax *similar* to cell arrays

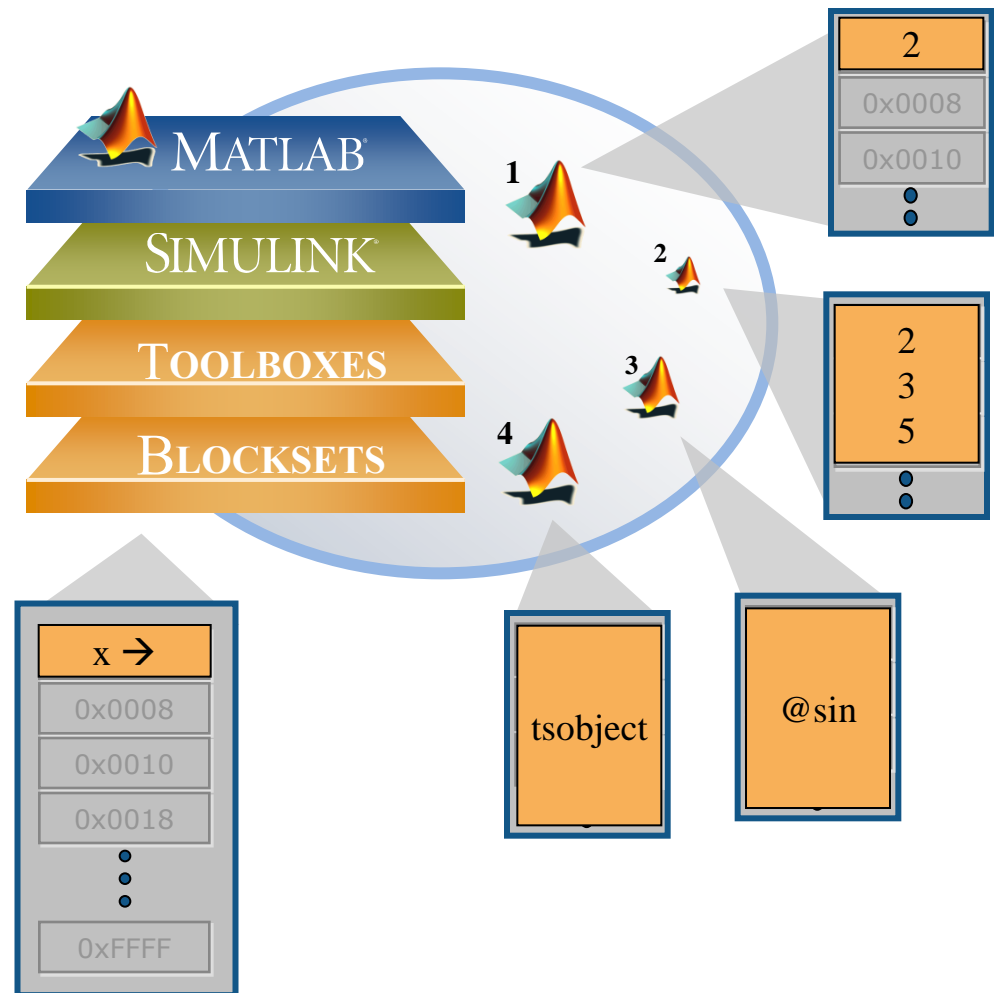


# Composite Array in Memory

```
>> matlabpool open 4

>> x = Composite(4)

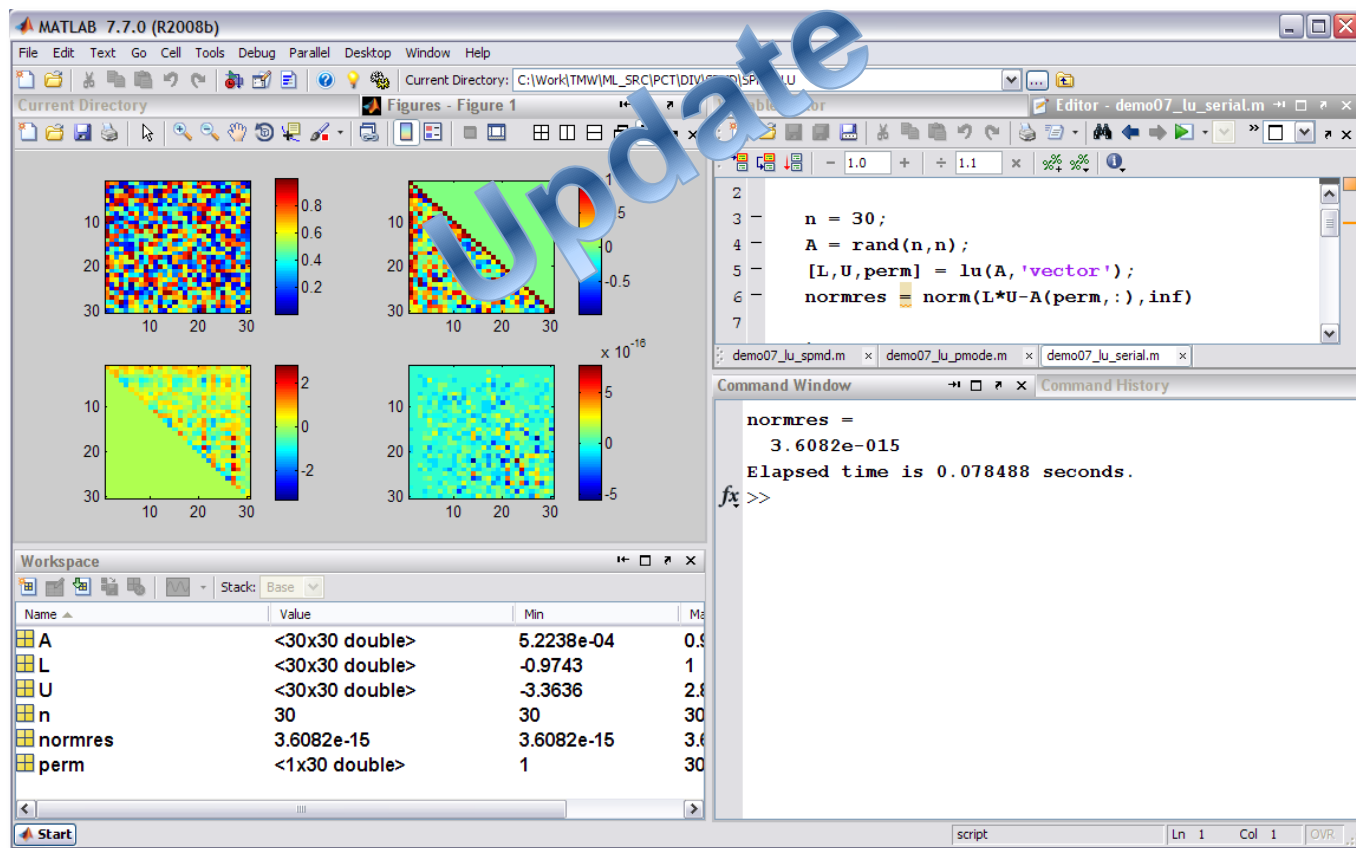
>> x{1} = 2
>> x{2} = [2, 3, 5]
>> x{3} = @sin
>> x{4} = tsobject()
```



# Exercise: Distributed Arrays

Convert the serial code to run data parallel

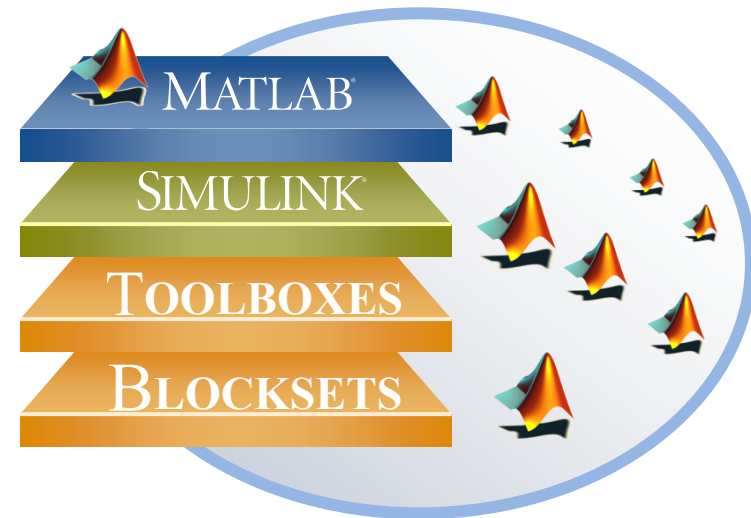
1. with an spmd block
2. with client side distributed arrays





# Summary for Interactive Functionality

- Client-side Distributed Arrays
  - MATLAB array type across cluster
  - Accessible from client
- SPMD
  - Flow control from serial to parallel
  - Fine-grained
  - More control over distributed arrays
- Composite Arrays
  - Generic data container across cluster
  - Accessible from client



# Parallel Profiler: `mpiprofile`

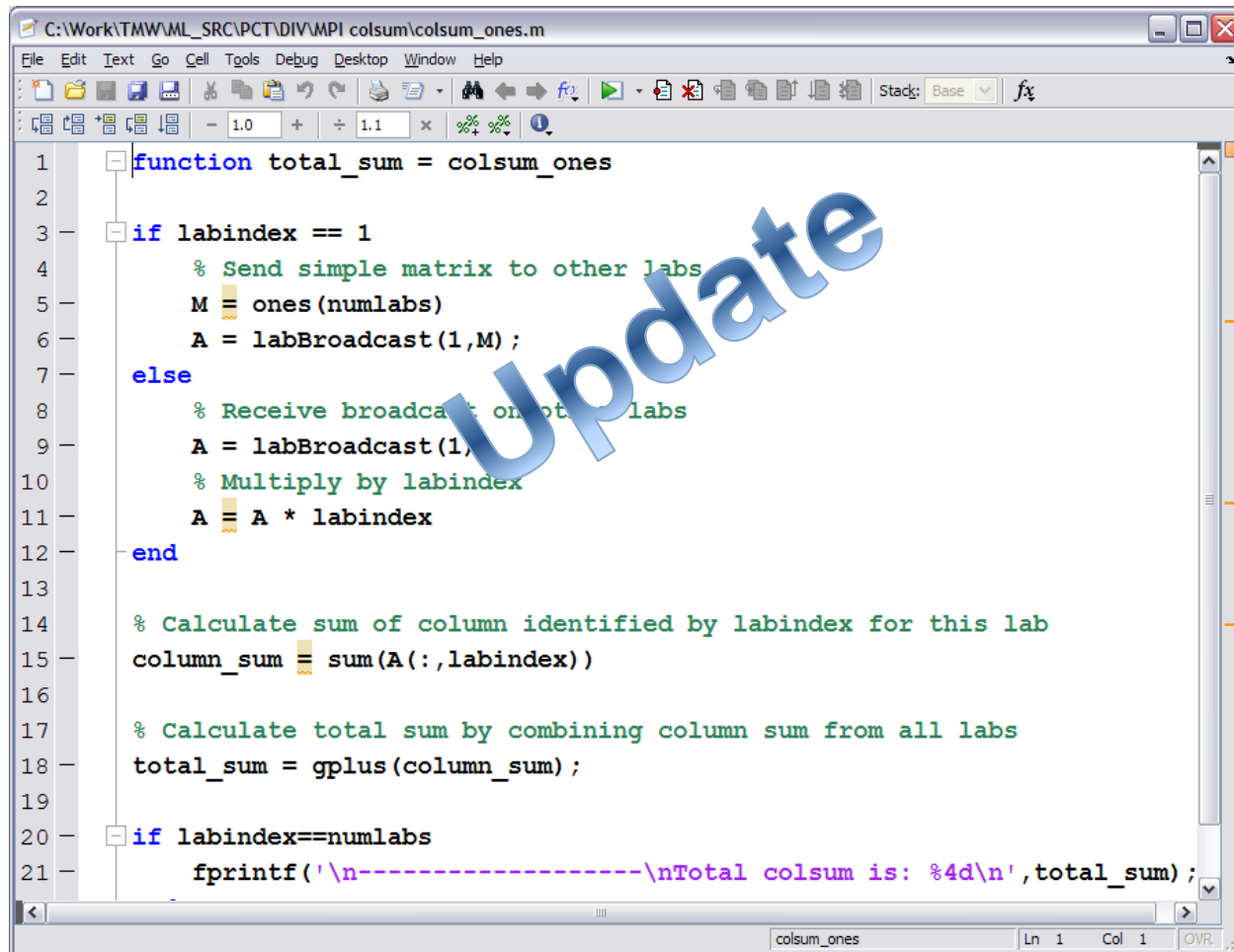
- Profiles the execution time for a function
  - Includes information about the communication between labs
    - Aggregates statistics on execution time and communication times
    - Collects information on communication with each of the other workers
  - should be executed in pmode or as part of a task in a parallel job
  
- Benefits
  - Identify the bottlenecks in your parallel algorithm
  - Understand which operations require communication

# MPI-Based Functions: Parallel Computing Toolbox

Use when a high degree of control over parallel algorithm is required

- High-level abstractions of MPI functions
  - `labSendReceive`, `labBroadcast`, and others
  - Send, receive, and broadcast any data type in MATLAB
- Automatic bookkeeping
  - Setup: communication, ranks, etc.
  - Error detection: deadlocks and miscommunications
- Pluggable
  - Use any MPI implementation that is *binary*-compatible with MPICH2

# Exercise: MPI-based Functions



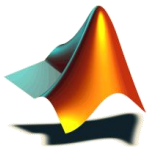
```

1 function total_sum = colsum_ones
2
3 if labindex == 1
4     % Send simple matrix to other labs
5     M = ones(numlabs)
6     A = labBroadcast(1,M);
7 else
8     % Receive broadcast on other labs
9     A = labBroadcast(1);
10    % Multiply by labindex
11    A = A * labindex
12 end
13
14 % Calculate sum of column identified by labindex for this lab
15 column_sum = sum(A(:,labindex))
16
17 % Calculate total sum by combining column sum from all labs
18 total_sum = gplus(column_sum);
19
20 if labindex==numlabs
21     fprintf('\n-----\nTotal colsum is: %4d\n',total_sum);

```

# Agenda

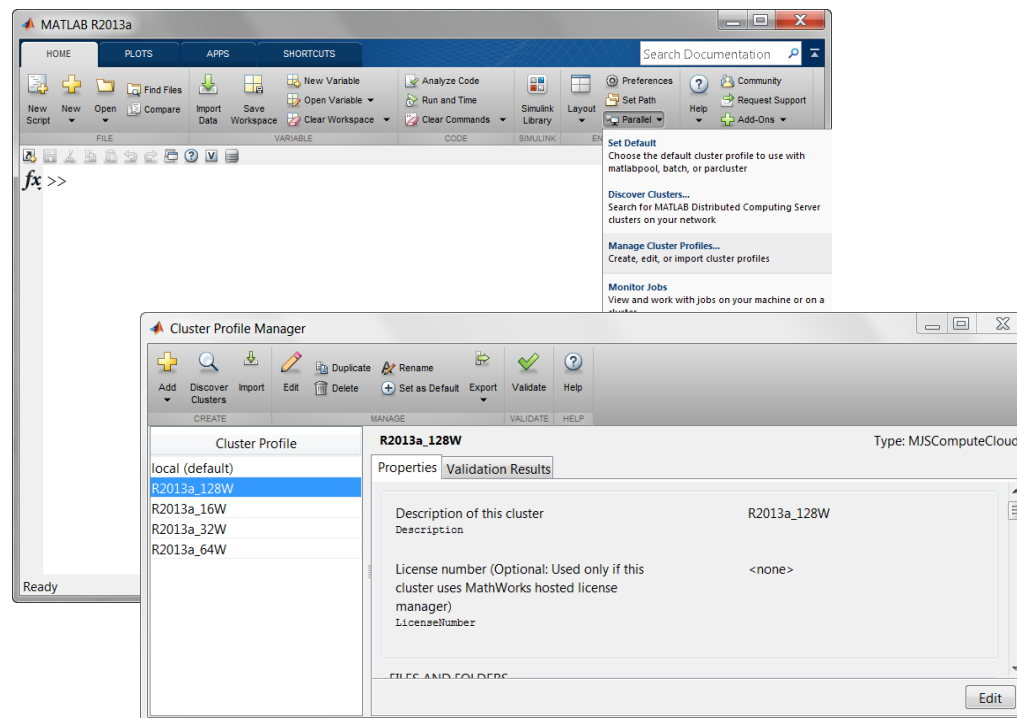
- Best practices in MATLAB programming
- Introduction to parallel computing tools
- Constructs for multicore/multi-processor computers



Scaling up to a cluster

# Cluster Profiles

- Create, edit, validate, import, and export profiles
- Switch profiles without changing MATLAB code



# Cluster Profile Best Practices

- Have Cluster Admin create cluster configuration files
- Label based on MATLAB version and cluster details
  - R2013a\_ServerRoom\_128core
  - R2012b\_BackOffice\_64core
- Validate configurations after importing
  - Ensures that system is configured properly
  - Helpful when debugging failed jobs

## Glossary (2)

- **Parallel Computing Toolbox (PCT)**
  - Desktop software that allows you to offload work from one MATLAB session (the client) to other MATLAB sessions, called Workers.
- **MATLAB Distributed Computing Server (MDCS)**
  - Software and license that provides extension of PCT constructs to clusters
- **Worker**
  - MATLAB computational engine process with interprocess communication capabilities and without full desktop



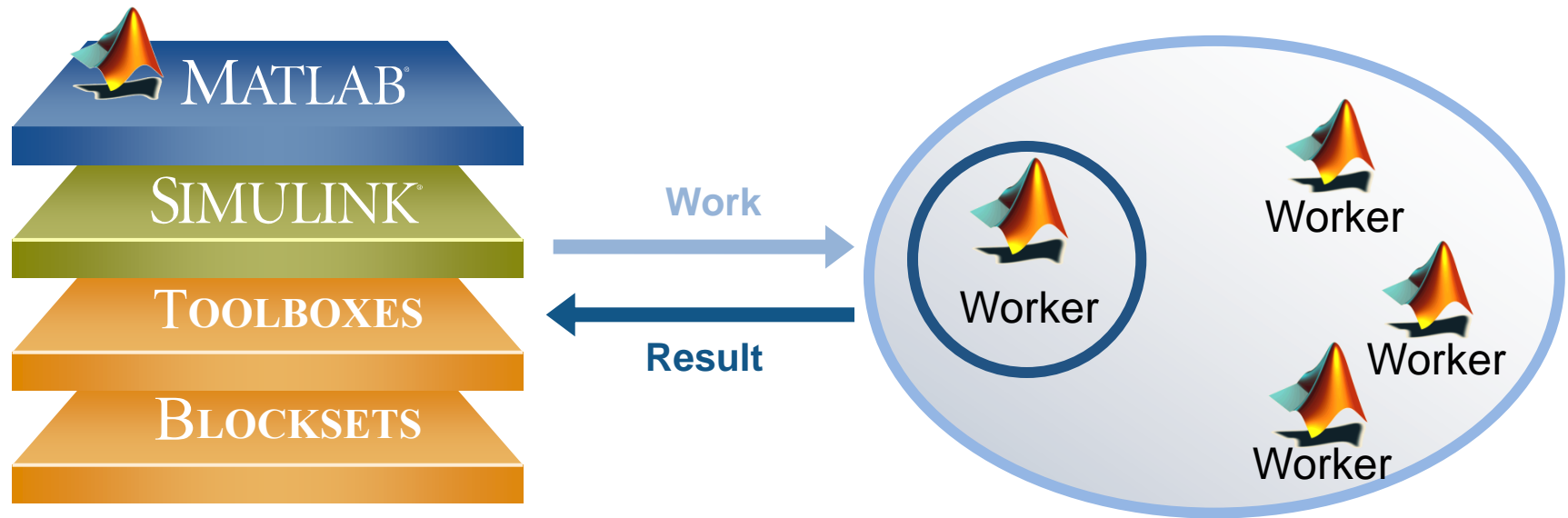
## Glossary (3)

- **Scheduler:**
  - A process, either third-party or by MathWorks (MathWorks Job Scheduler), that queues jobs and assigns tasks to workers.
  
- **Local Scheduler:**
  - MathWorks-provided scheduler that runs in the MATLAB client session. Queues jobs and assigns tasks to Workers on the local host.
  
- **MathWorks Job Scheduler:**
  - Built-in scheduler to provide a turn-key solution for MATLAB-only clusters

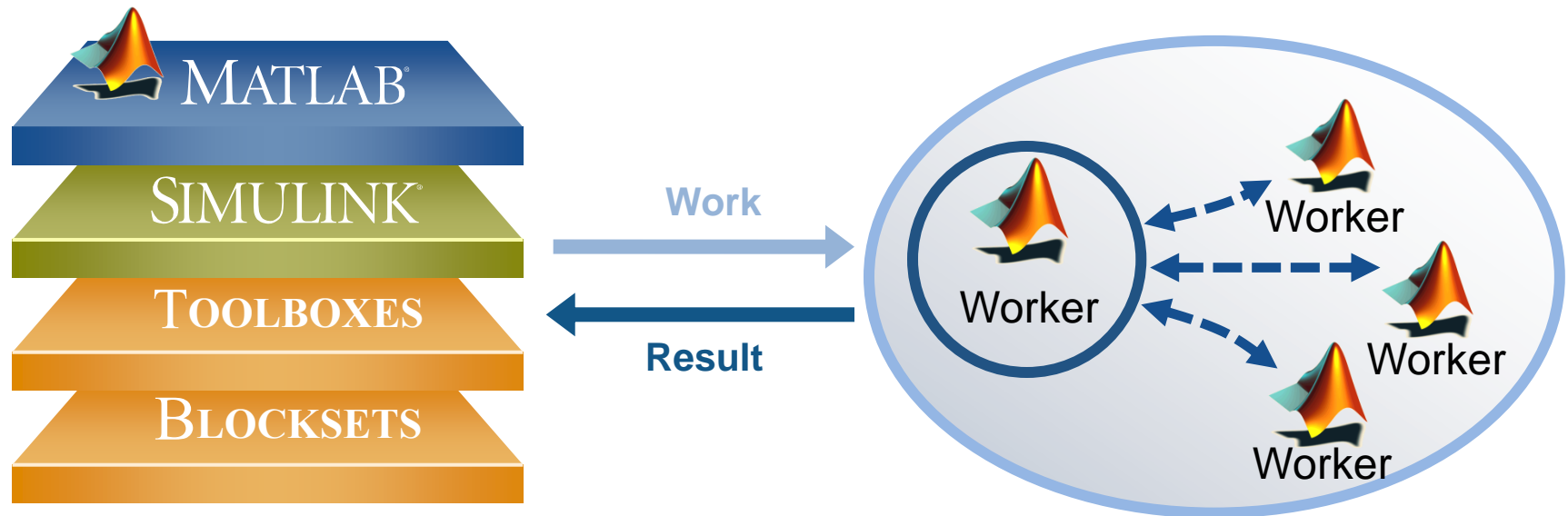
# Interactive to Scheduling

- Interactive
  - Great for prototyping
  - Immediate access to MATLAB workers
  
- Scheduling
  - Offloads work to other MATLAB workers (local or on a cluster)
  - Access to more computing resources for improved performance
  - Frees up local MATLAB session

# Scheduling Work



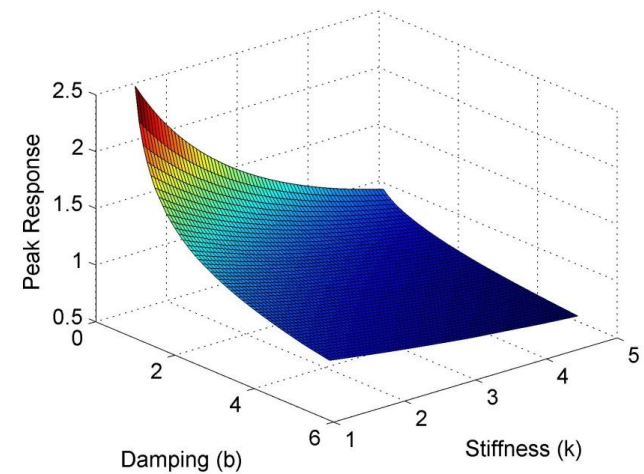
# Offload parallel computations with batch



## Exercise: Schedule Processing with batch

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

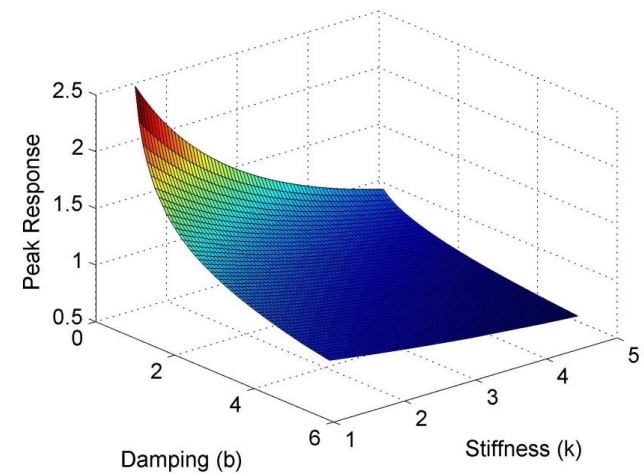
$$\overbrace{m}^5 \ddot{x} + \underbrace{b}_{1,2,\dots} \dot{x} + \underbrace{k}_{1,2,\dots} x = 0$$



# Summary: Schedule Processing with batch

- Offload processing to workers
  - `batch`
  - `matlabpool`
- Monitor progress of scheduled job
  - **Job Monitor**
- Retrieve results from job
  - `fetchOutputs`

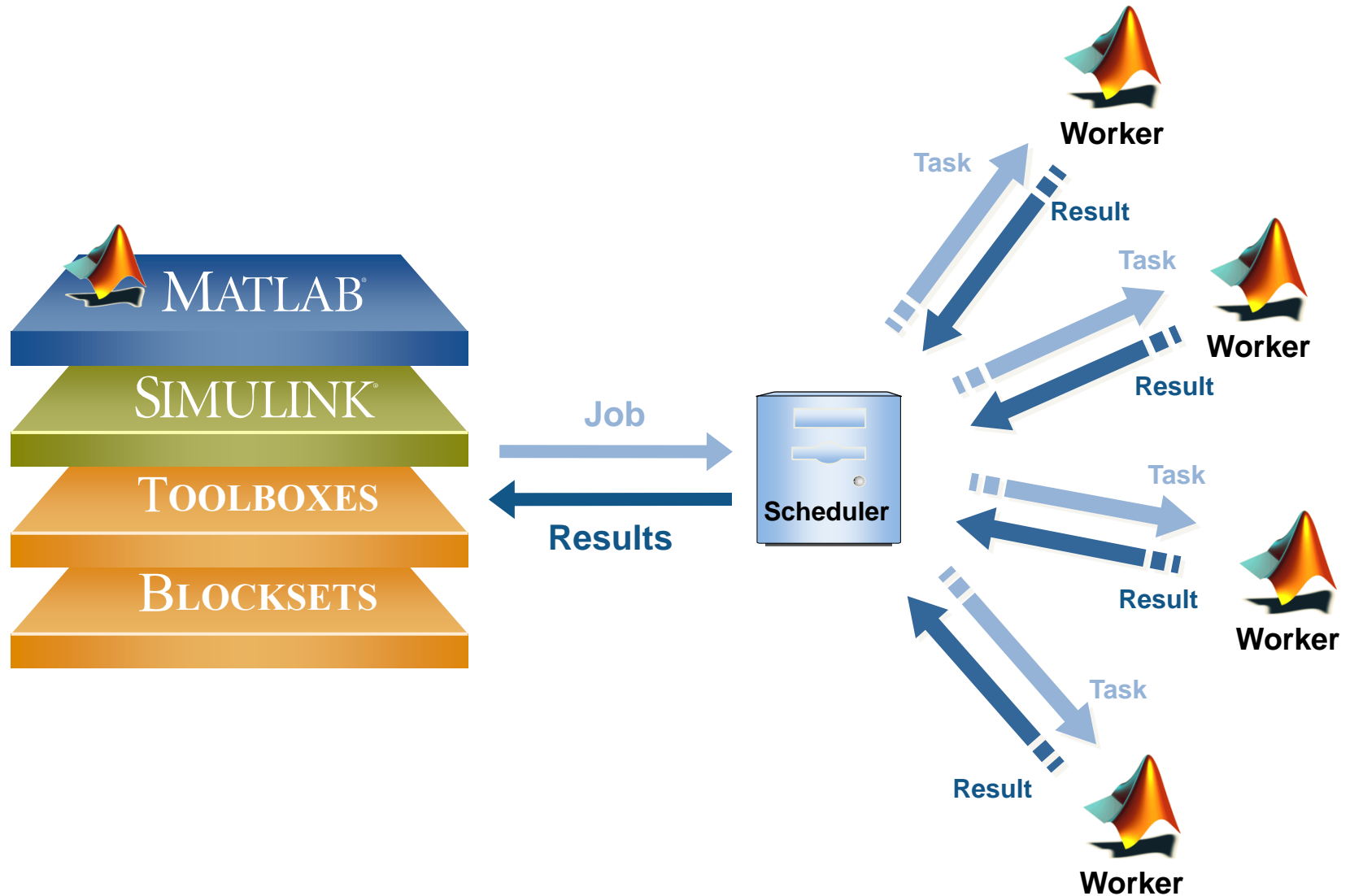
$$\overbrace{m}^5 \ddot{x} + \underbrace{b}_{1,2,\dots} \dot{x} + \underbrace{k}_{1,2,\dots} x = 0$$



# Creating jobs and tasks

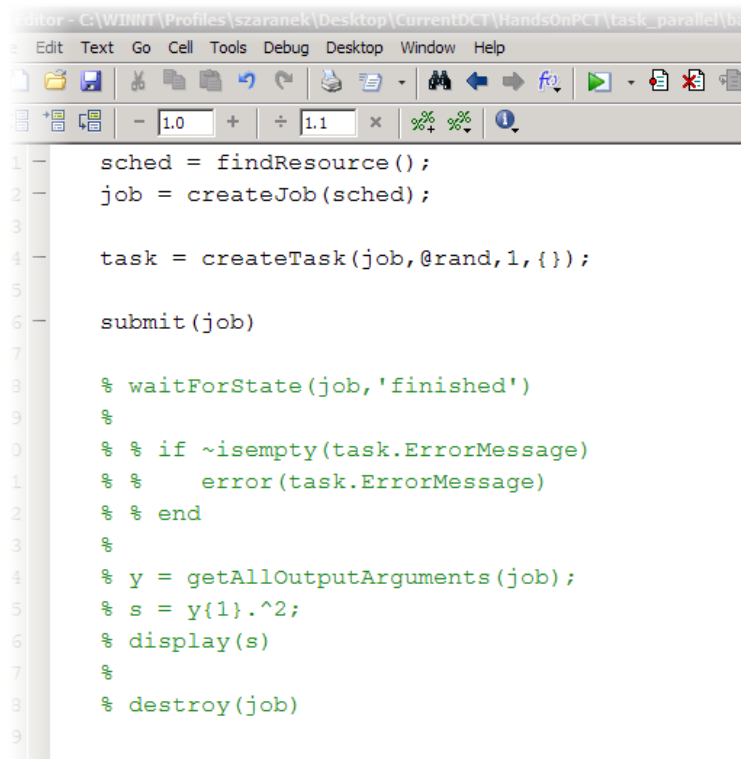
- Workflow management:
  - **batch**: offload execution of a function or script to run in a cluster or desktop background
  - Jobs and tasks: create a set of tasks that execute functions or scripts to run in a cluster or desktop background
- Using more processors:
  - **parfor**: write a loops for a statement or block of code that executes in parallel on a cluster of workers, which are identified and reserved with **matlabpool**
  - Jobs and tasks: organize workflow into independent units of work and executing multiple tasks concurrently

# Scheduling Jobs and Tasks





# Creating and Submitting Jobs



```

1  sched = findResource();
2  job = createJob(sched);
3
4  task = createTask(job,@rand,1,{});
5
6  submit(job)
7
8  % waitForState(job,'finished')
9  %
10 % % if ~isempty(task.ErrorMessage)
11 % %     error(task.ErrorMessage)
12 % % end
13 %
14 % y = getAllOutputArguments(job);
15 % s = y{1}.^2;
16 % display(s)
17 %
18 % destroy(job)
19

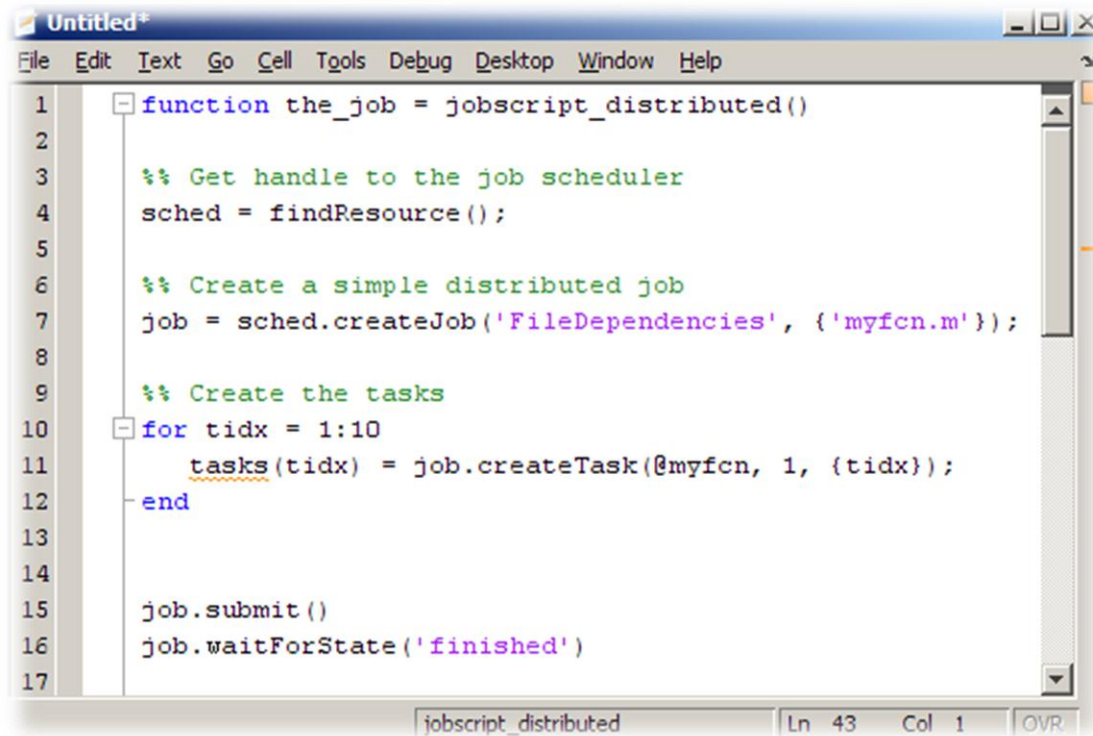
```



Rather than using a shell script to submit a job to a cluster, we'll write our *jobscrip*t in MATLAB.

task\_parallel\basic\_jobscript.m

# Exercise: A simple jobscript (1)



```

1  function the_job = jobscript_distributed()
2
3  %% Get handle to the job scheduler
4  sched = findResource();
5
6  %% Create a simple distributed job
7  job = sched.createJob('FileDependencies', {'myfcn.m'});
8
9  %% Create the tasks
10 for tid = 1:10
11     tasks(tid) = job.createTask(@myfcn, 1, {tid});
12 end
13
14
15 job.submit()
16 job.waitForState('finished')
17

```

jobscript\_distributed Ln 43 Col 1 OVR

>> jobscript\_distributed

## Exercise: A simple jobscript (2)

```

Untitled*
File Edit Text Go Cell Tools Debug Desktop Window Help
18 %% Check for Errors
19 em = get(tasks, 'ErrorMessage');
20 if all(cellfun(@isempty, em)) == false
21     disp('Tasks failed')
22     if nargout==0
23         job.destroy
24     else
25         the_job = job;
26     end
27
28     return
29 end
30
31 %% No errors, get output
32 y = job.getAllOutputArguments();
33 celldisp(y)
34
35 %% Cleanup
36 if nargout == 0
37     job.destroy()
38 else
39     the_job = job;
40 end
41
42 end
43
jobscript_distributed Ln 43 Col 1 OVR

```

# Resolving Jobs & Tasks Issues

- MATLAB can perform an analysis on the task functions in the job to determine which code files are necessary for the workers, and automatically send those files to the workers.
- Alternatively files need can either be submitted as part of the job (**AttachedFiles**) or the folder needs to be accessible (**AdditionalPaths**)
- There is overhead when adding too many files to the job; but setting path dependencies requires the Worker to be able to reach the path

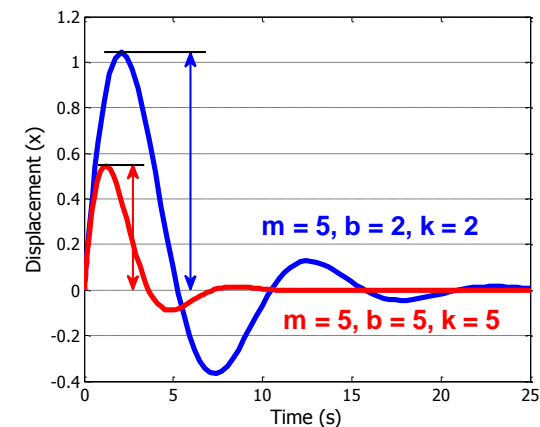
# Factors to Consider for Scheduling

- There is always an overhead to distribution
  - Combine small repetitive function calls
- Share code and data with workers efficiently
  - Set job properties (`AttachedFiles`, `AdditionalPaths`)
- Minimize I/O for scripts
  - Specify variables with `Workspace` option for `batch`
- Take advantage of diary as job runs (for `batch`)

# Exercise: Schedule Processing with tasks

- Offload three independent approaches to solving our previous ODE example
- Retrieve simulated displacement as a function of time for each simulation
- Plot comparison of results in local MATLAB

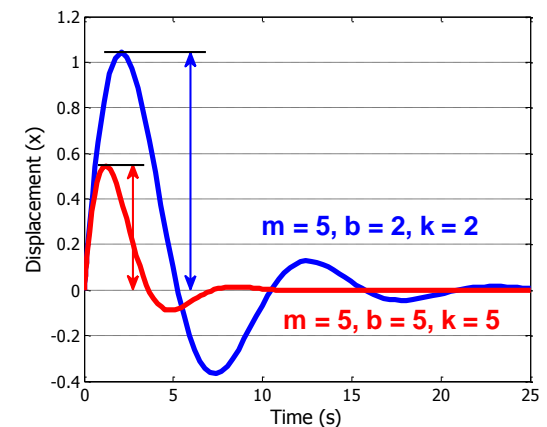
$$\overbrace{m}^5 \ddot{x} + \underbrace{b}_{1,2,\dots} \dot{x} + \underbrace{k}_{1,2,\dots} x = 0$$



# Summary: Schedule Processing with tasks

- Return cluster object for specific profile with `parcluster`
- Set up with `createJob` and `createTask`
- Begin processing with `submit`
- Check progress with **Job Monitor**
- Retrieve results with `fetchOutputs`

$$\overbrace{m}^5 \ddot{x} + \underbrace{b}_{1,2,\dots} \dot{x} + \underbrace{k}_{1,2,\dots} x = 0$$



# Summary and Recommendations



- Good parallel code starts with best practices for MATLAB
  - Profile your code to search for bottlenecks
  - Make use of M-Lint when coding `parfor` and `spmd`
  - Beware of writing to files and output to screen
- Take advantage of range of parallel functionality
  - `parfor`, `batch`, `distributed`, `spmd`, jobs and tasks
- Run locally before moving to cluster



# For more information

**Visit**

[www.mathworks.com/products/parallel-computing](http://www.mathworks.com/products/parallel-computing)