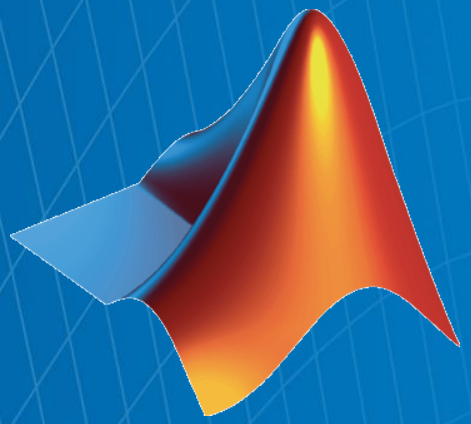
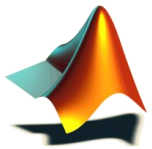
Faster, Larger, Easier : Parallel Computing with MATLAB

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.

BLAS and LAPACK require contiguous arrays

- JIT/Accelerator
  - Improves looping
  - Generates on-the-fly multithreaded code
  - Continually improving

Glossary (1)

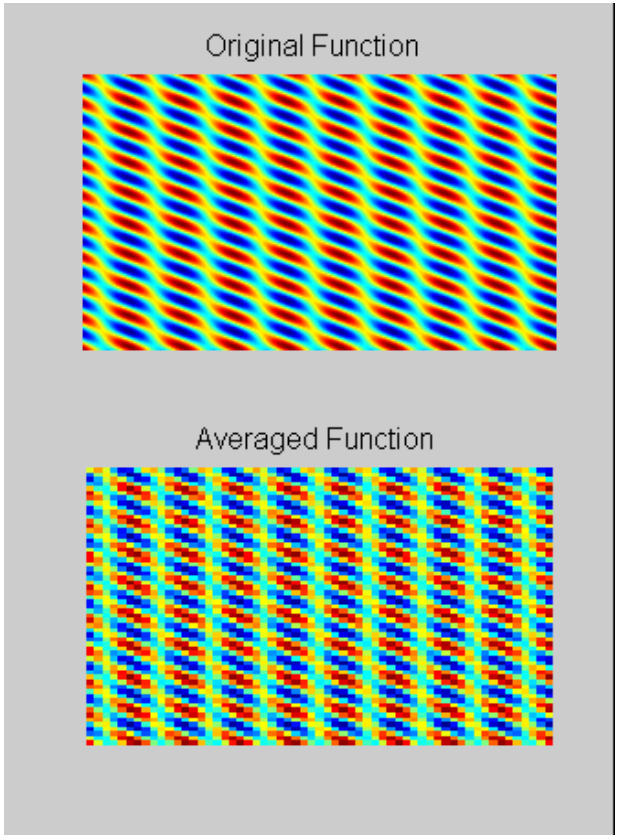
- Implicit Multithreaded Computation:
  - MATLAB supports multithreaded computation for a number of linear algebra and element-wise numerical functions. These functions automatically execute on multiple threads.

- 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

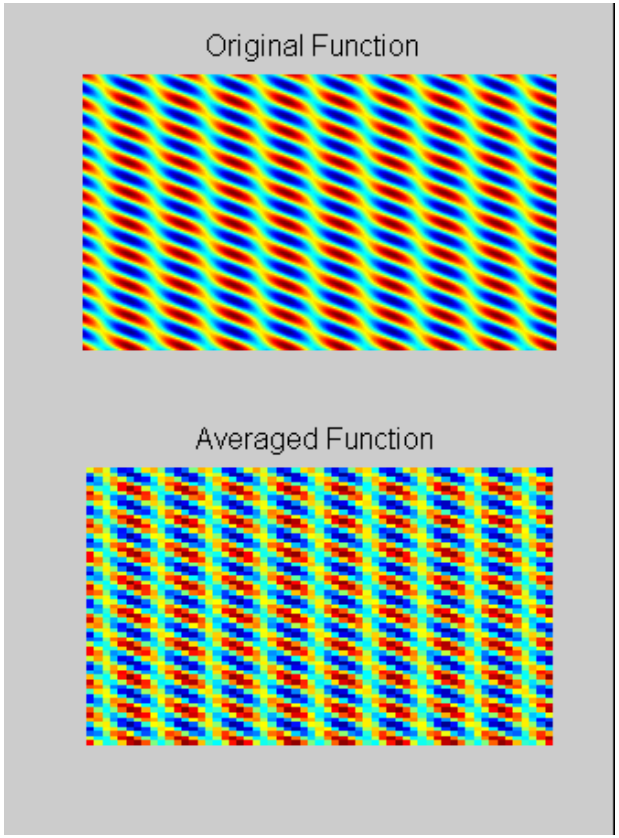


Summary of Block Processing Exercise

- Used built-in timing functions

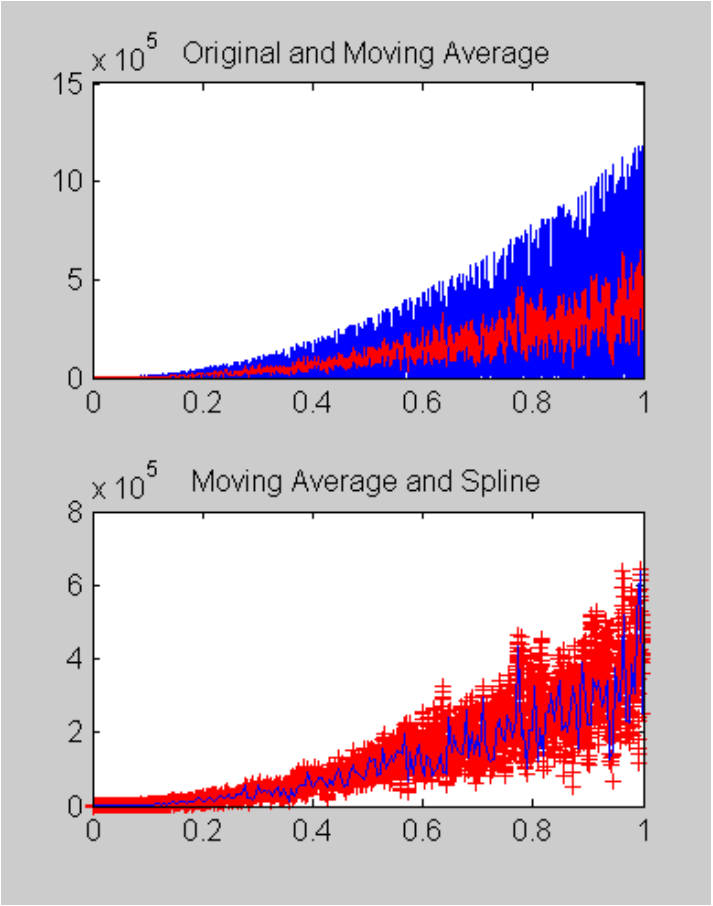
```
>> tic
>> toc
```

- Used Code Analyser to find suboptimal code
- Preallocated arrays
- Vectorized code



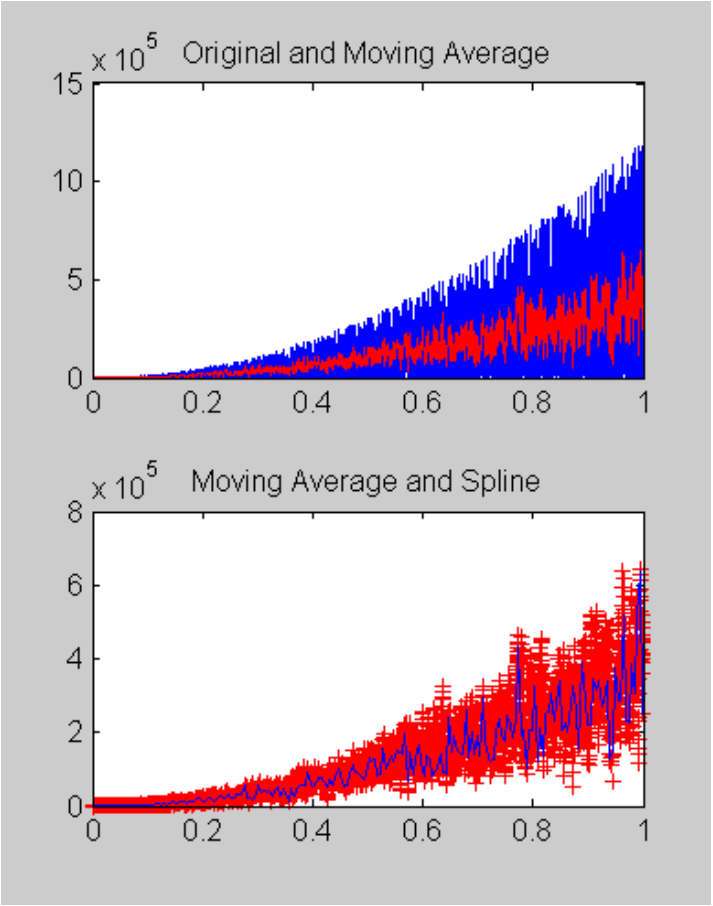
Exercise: Fitting Data

- 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

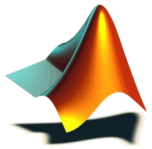


Classes of Bottlenecks

- 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

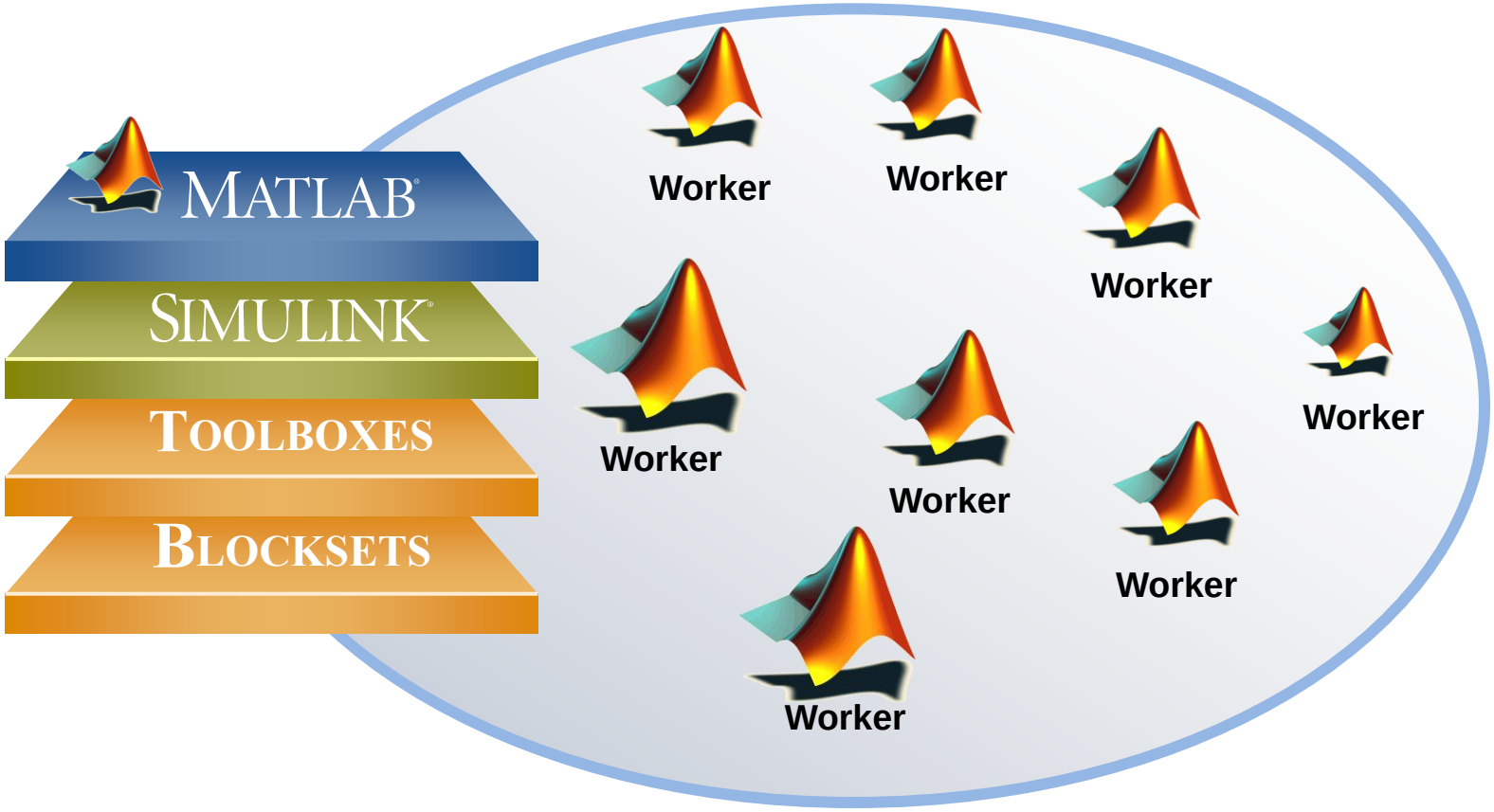
- Best practices in MATLAB programming



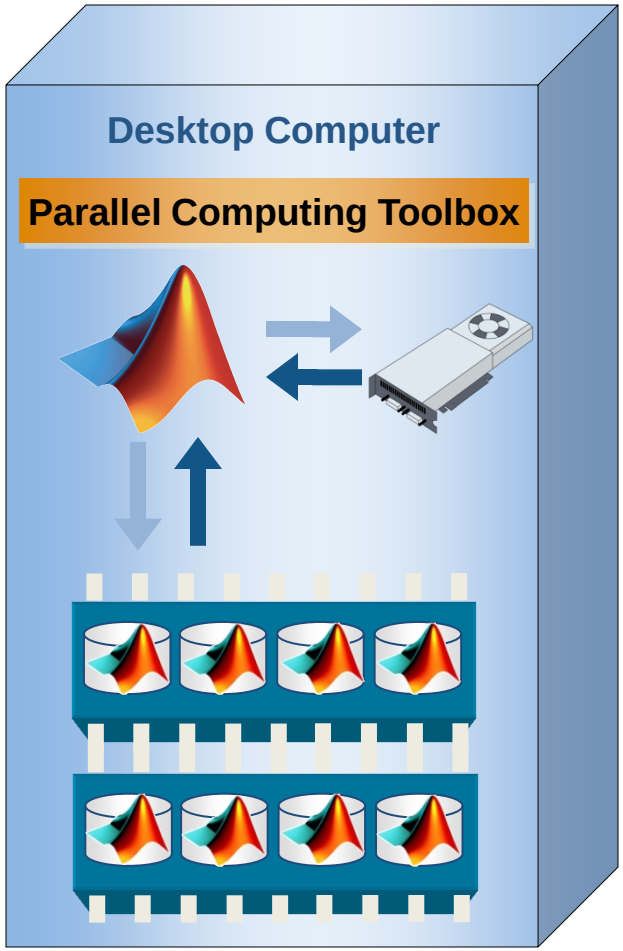
Introduction to parallel computing tools

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

Going Beyond Serial MATLAB Applications

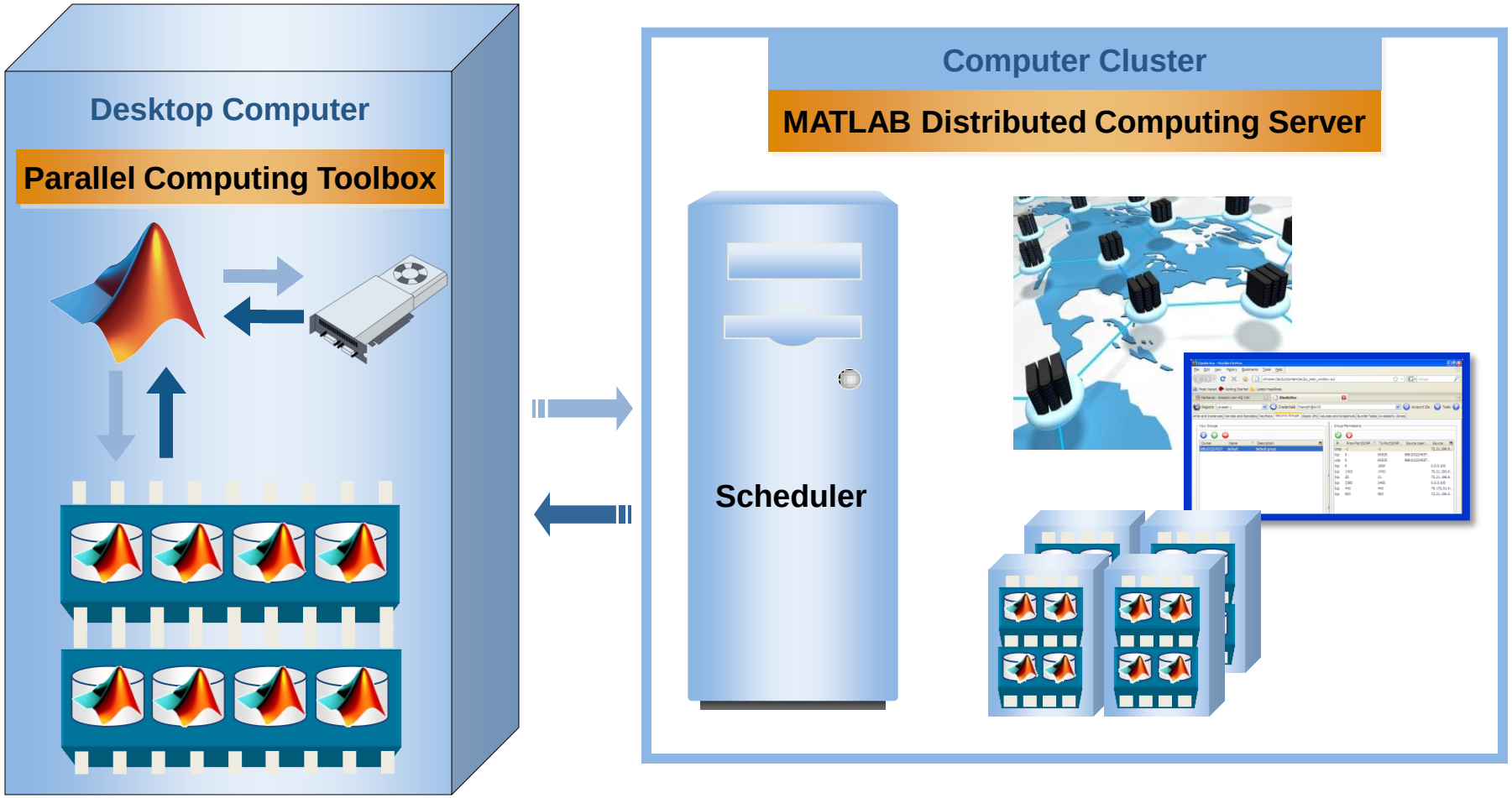


Parallel Computing Toolbox for the Desktop

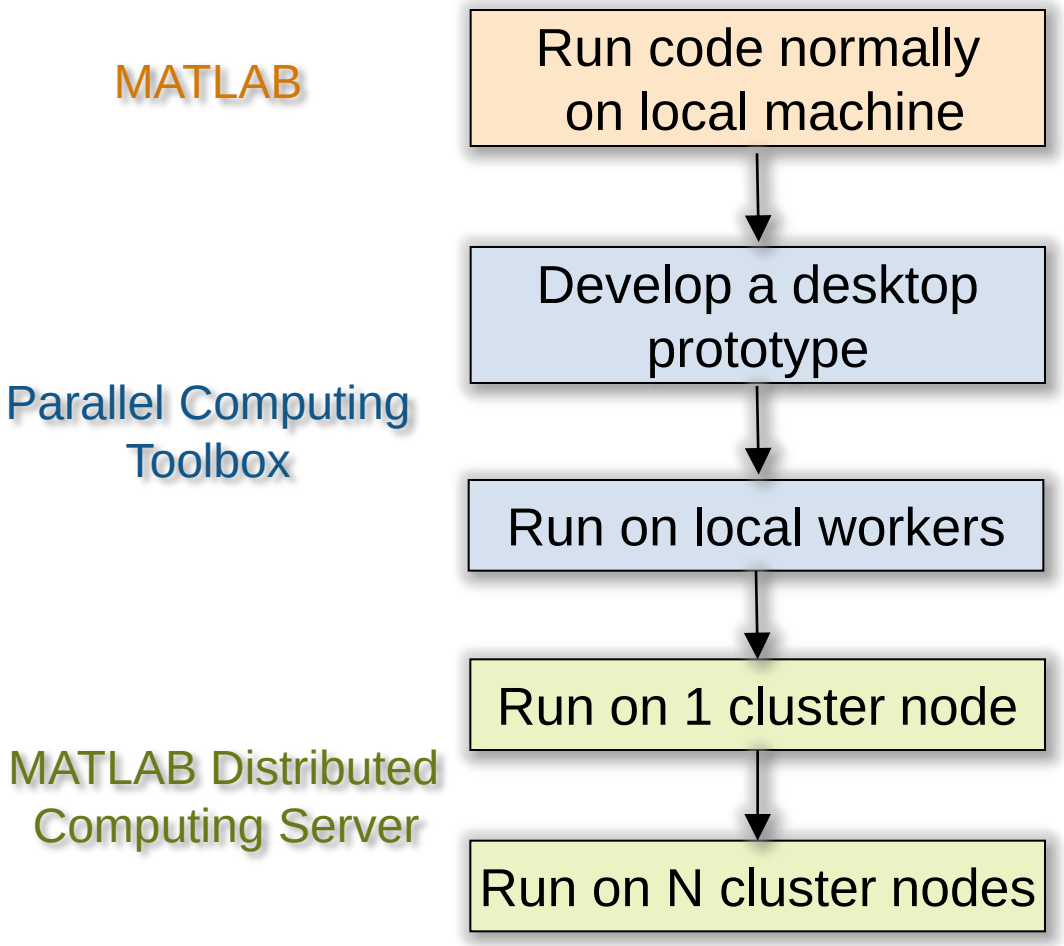


- Speed up parallel applications
- Take advantage of GPUs
- Prototype code for your cluster

Scale Up to Clusters, Grids, and Clouds

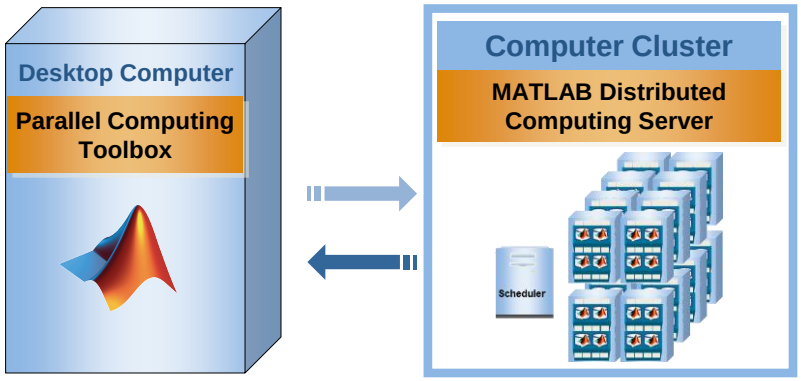


Program Development Guidelines



Use cases for Cluster Computing

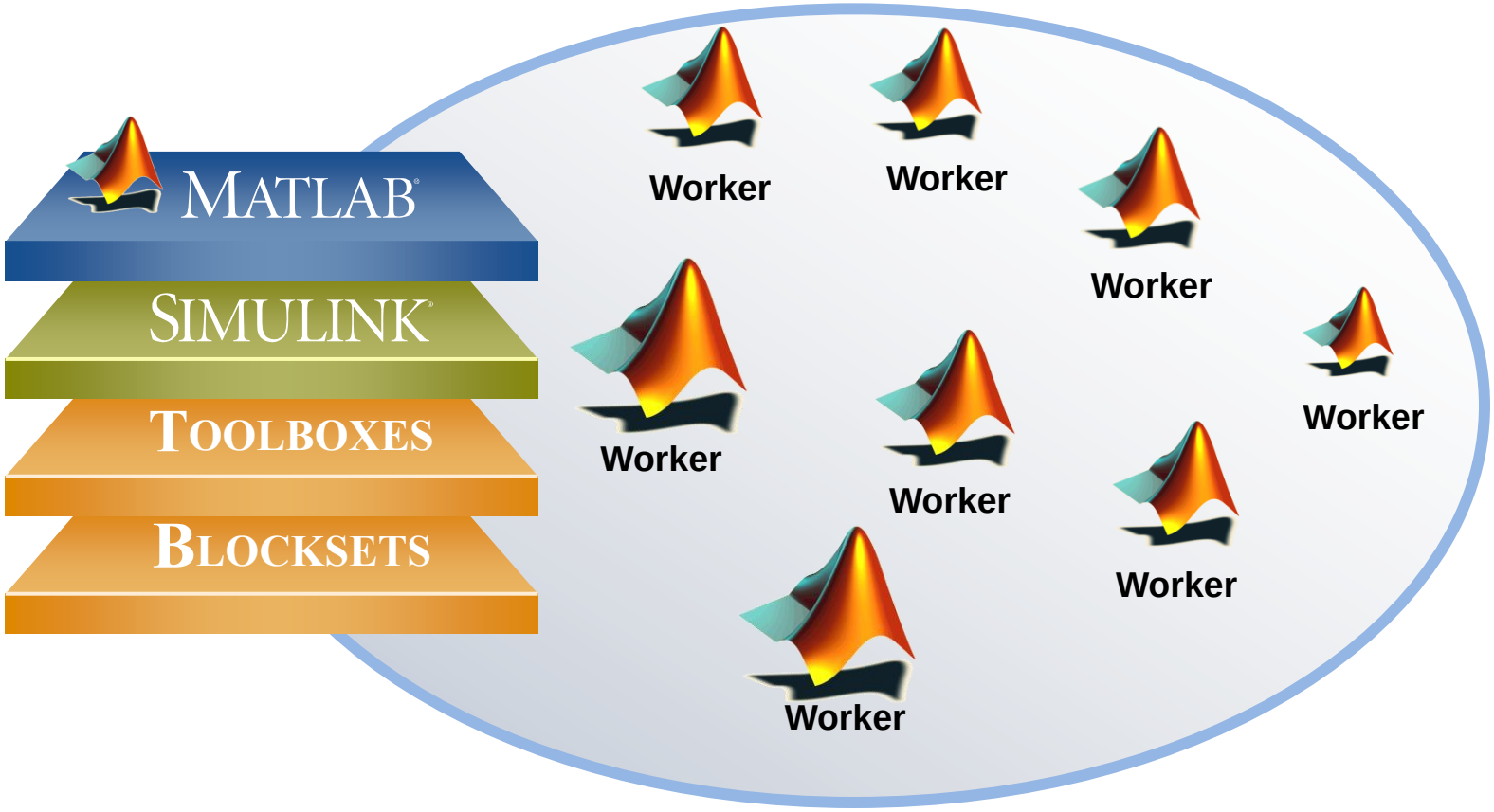
- Offload computation:
  - Free up desktop
  - Access better computers
- Scale speed-up:
  - Use more cores
  - Go from hours to minutes
- Scale memory:
  - Utilize distributed arrays
  - Solve larger problems without re-coding



Interactive to Scheduling

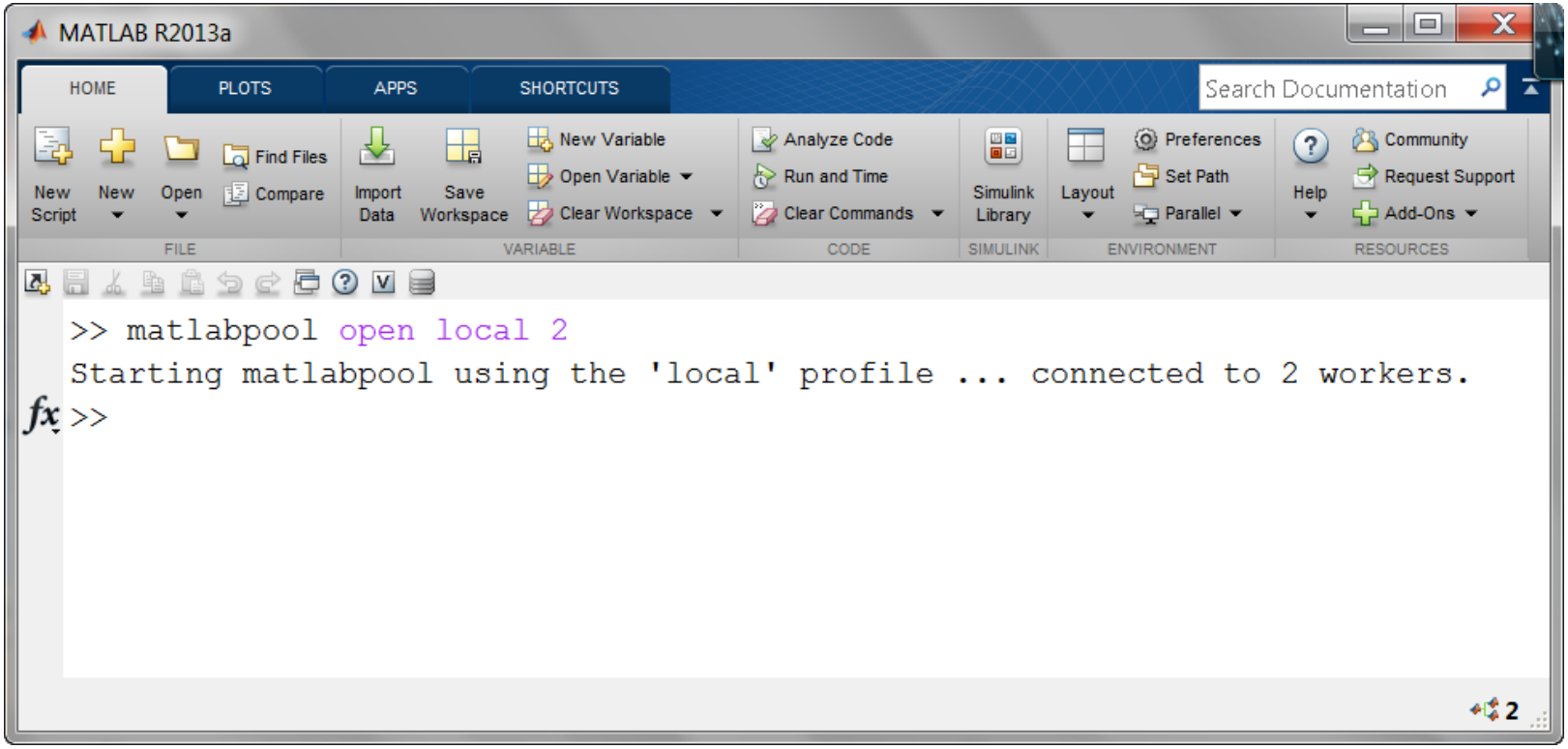
- Interactive
  - Great for prototyping
  - Immediate access to MATLAB workers
- Scheduling
  - Offloads work to other MATLAB workers (local or on a cluster)
  - Access to more computing resources for improved performance
  - Frees up local MATLAB session

Exercise: Working with a matlabpool

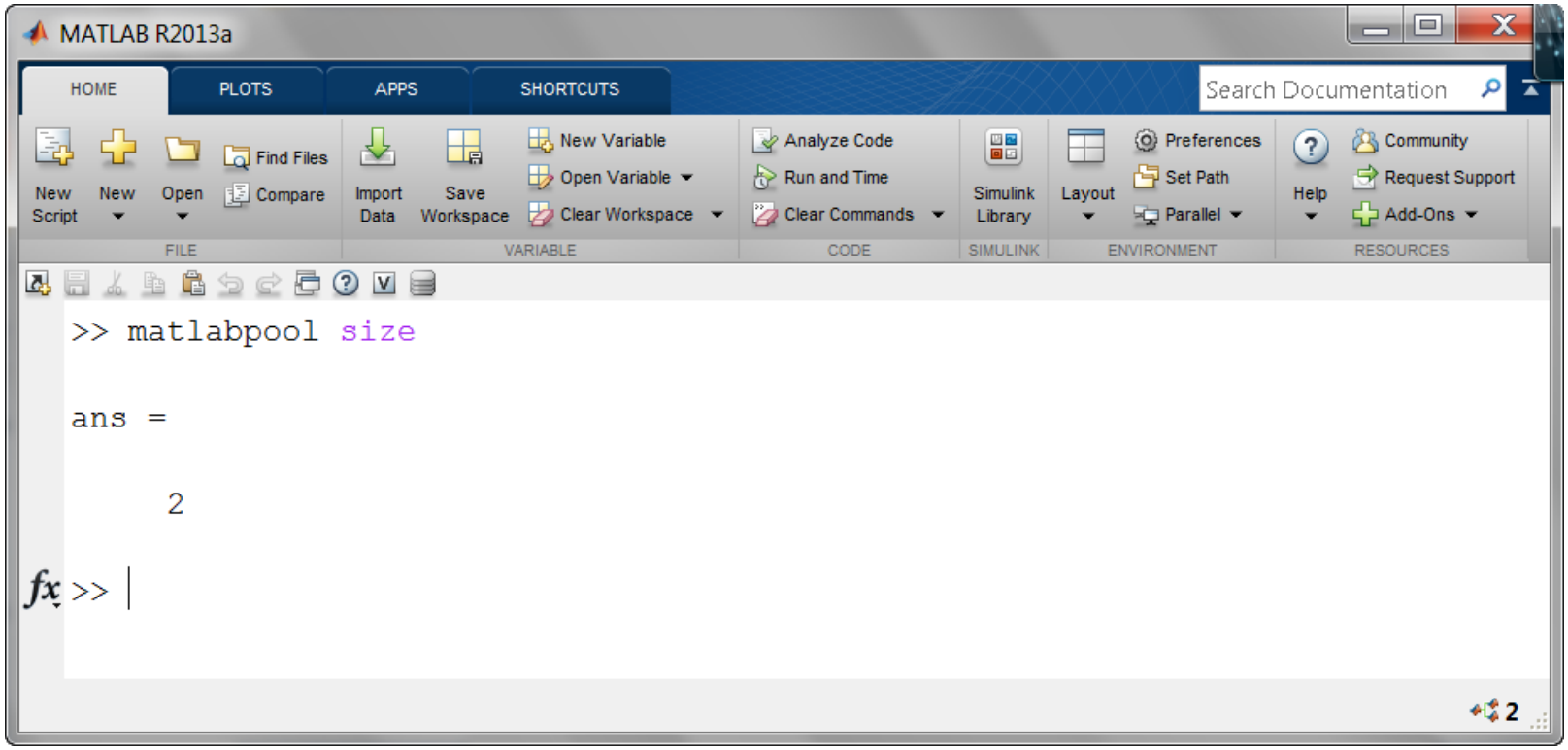


Starting a matlabpool…

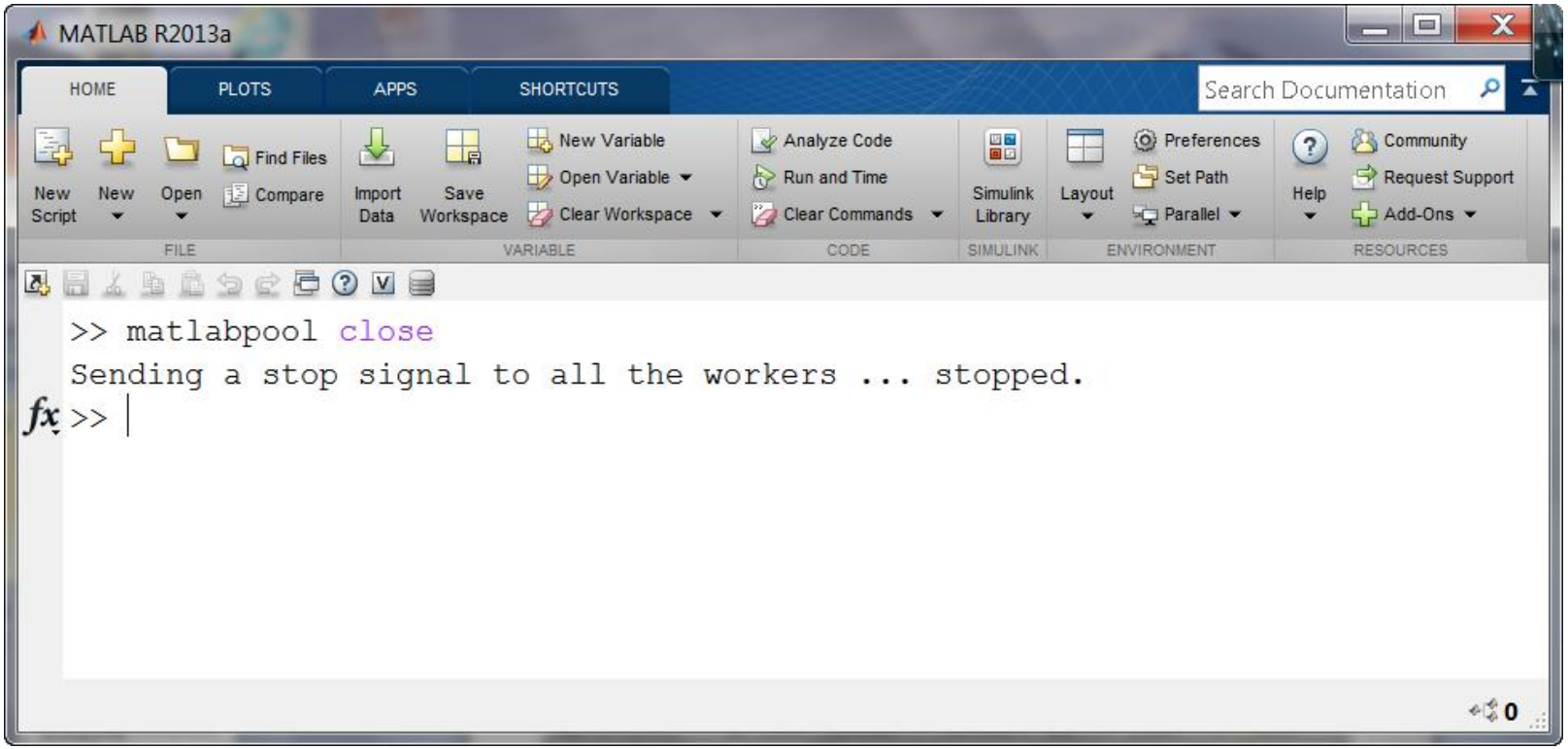
- Open a matlabpool with two Workers using the local configuration



Determining the Size of the Pool…

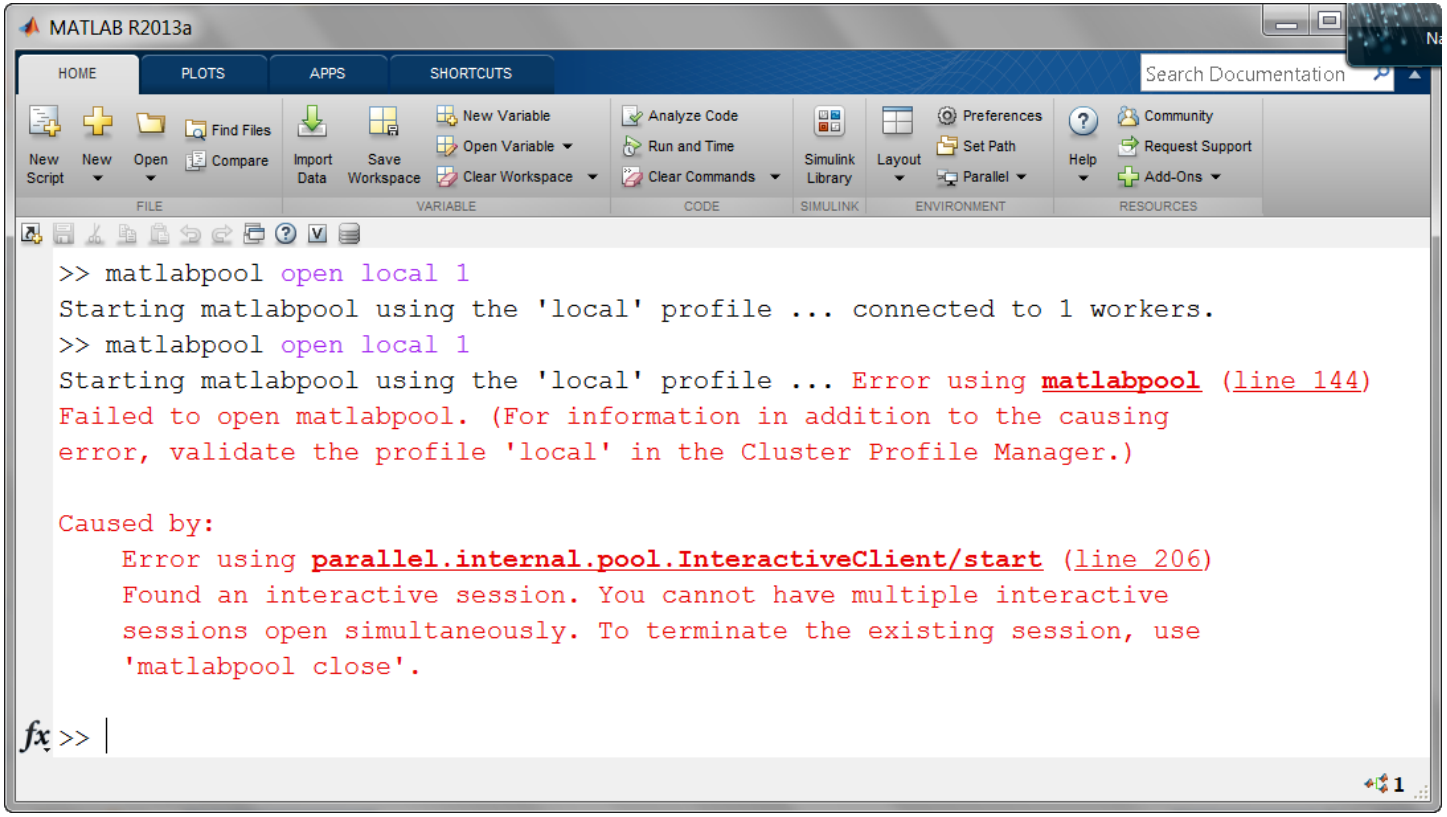


Stopping a matlabpool

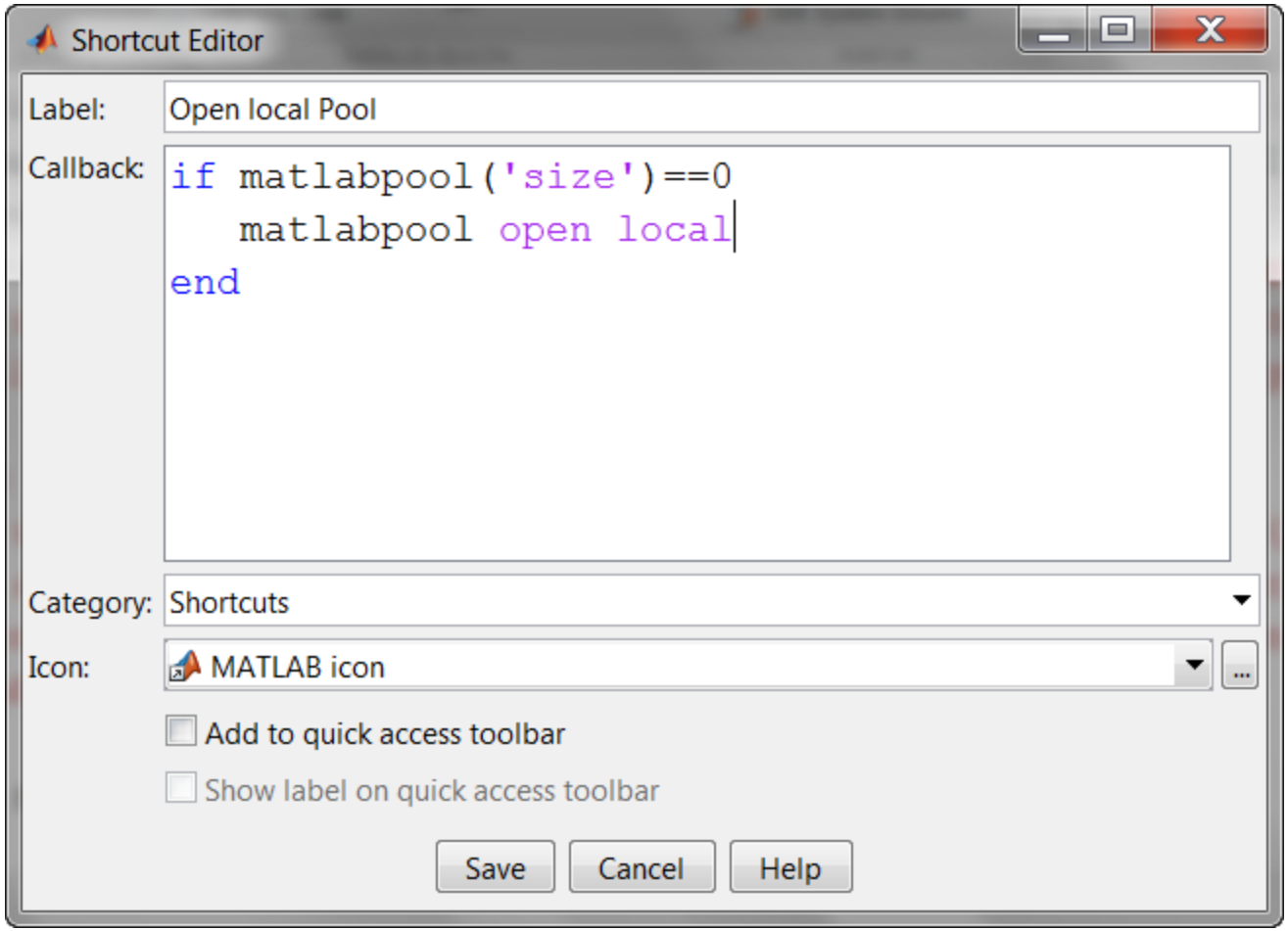


How many pools of Workers at a time?

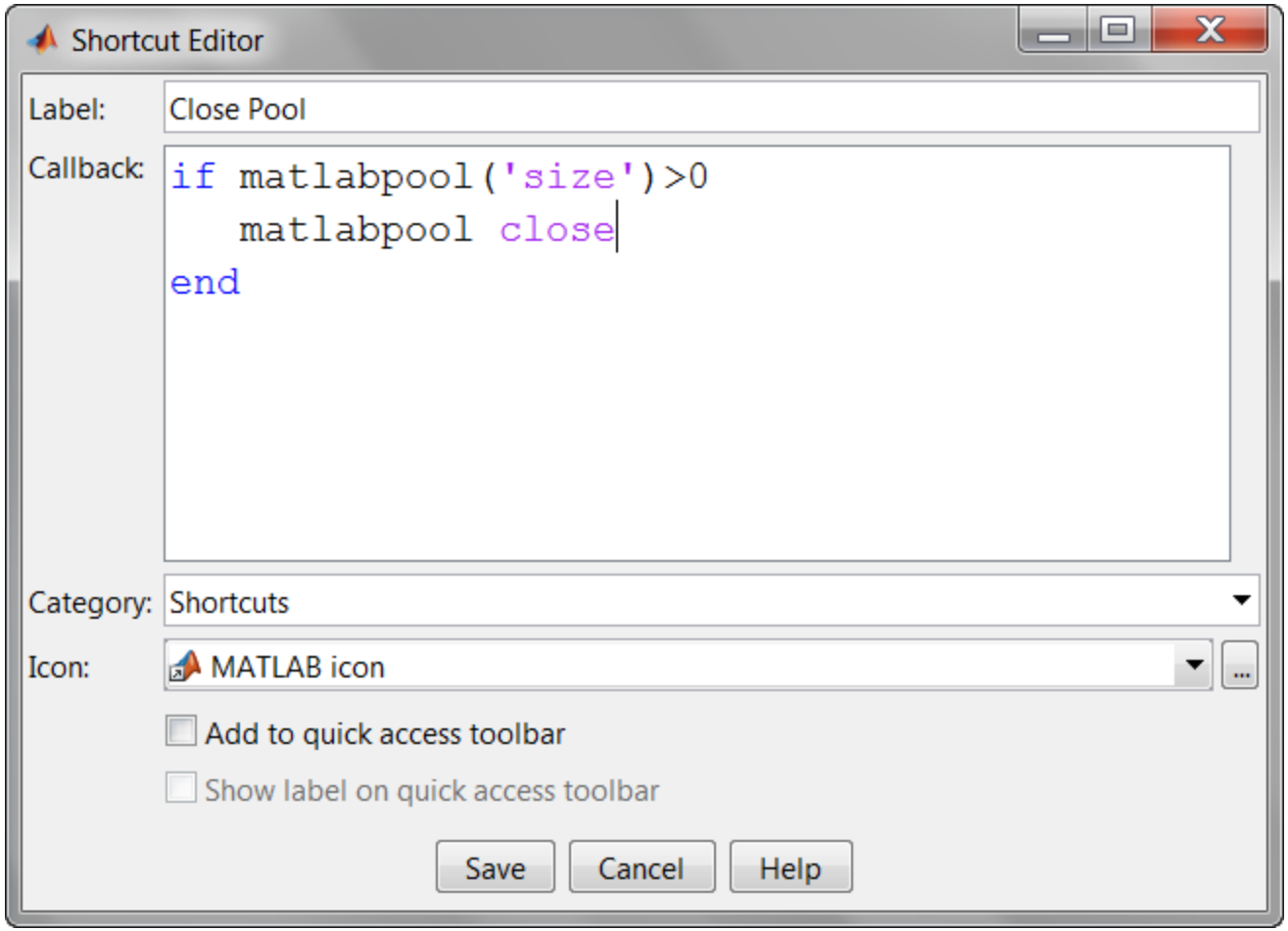
- Even if you have not exceeded the maximum number of Workers, you can only open one matlabpool at a time



Add Shortcut for starting a matlabpool

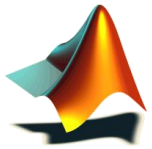


Add Shortcut for stopping the matlabpool



Agenda

- Best practices in MATLAB programming



Introduction to parallel computing tools

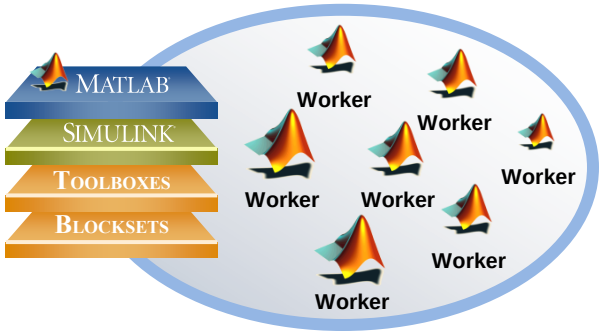
- Constructs for multicore/multi-processor computers
- Scaling up to a cluster

Programming Parallel Applications (CPU)

- Built-in support with Toolboxes

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

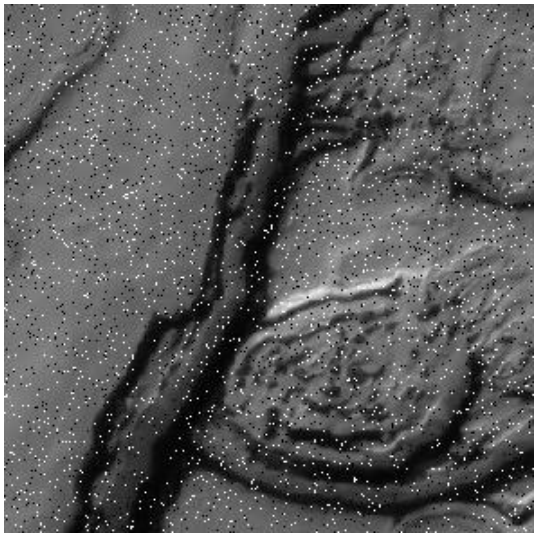
www.mathworks.com/products/parallel-computing/builtin-parallel-support.html

Exercise: Filtering an Image

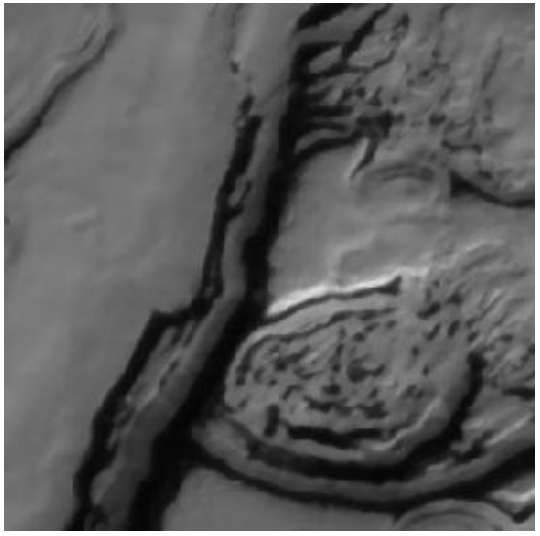
Built-in parallel support

- 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

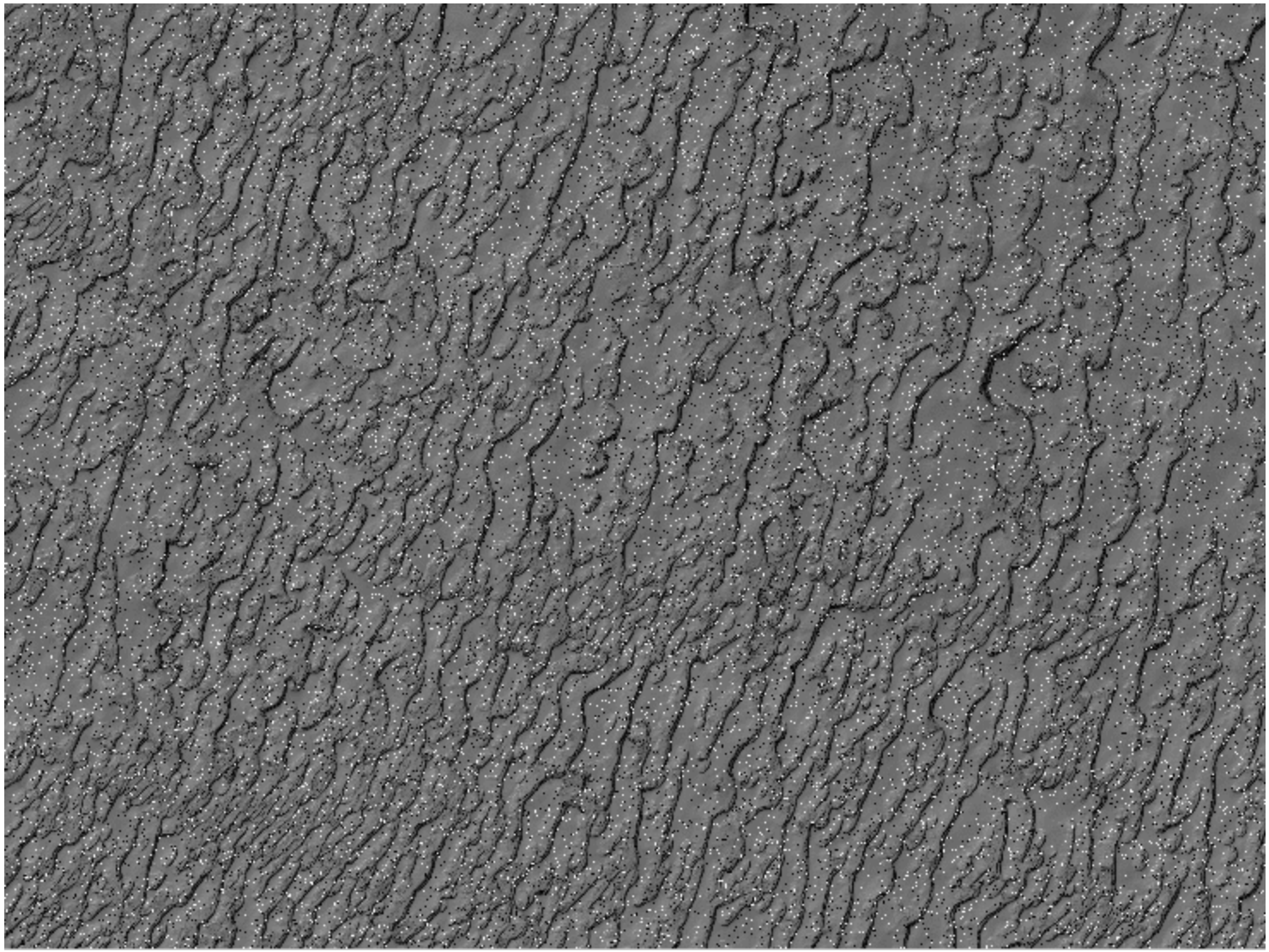


Filtered Image



http://hirise.lpl.arizona.edu/
From - NASA/JPL/University of Arizona

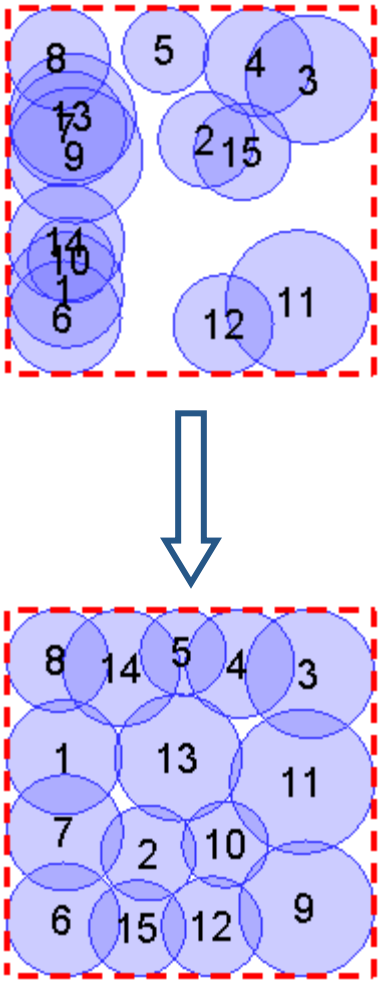
2D Median Filter



Exercise: Optimizing Cell Tower Position

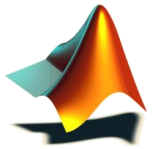
Built-in parallel support

- 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



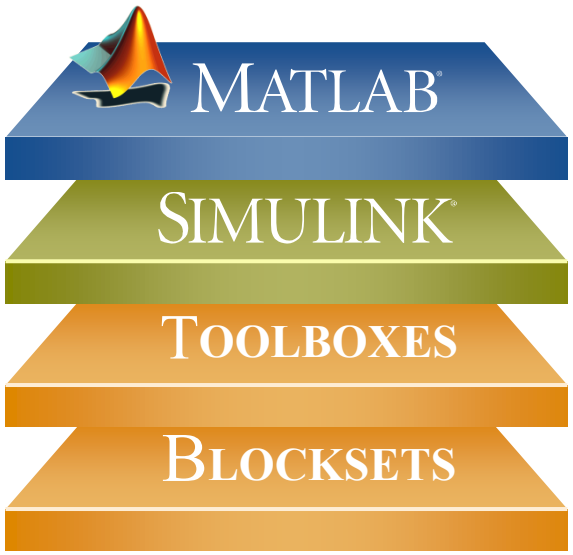
Constructs for multicore/multi-processor computers

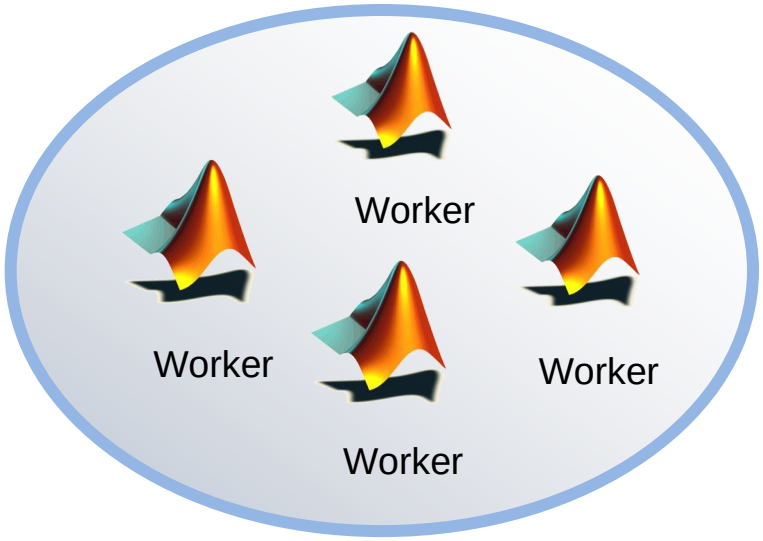
- Scaling up to a cluster

Programming Parallel Applications (CPU)

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

Parallelizing independent tasks



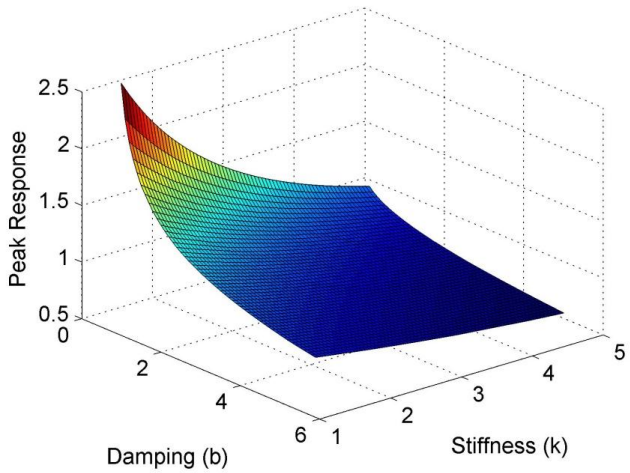


blogs.mathworks.com/loren/2009/10/02/using-parfor-loops-getting-up-and-running/

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

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



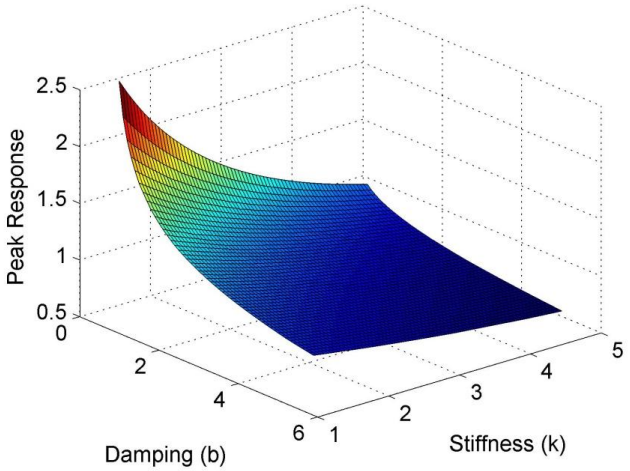
Summary: Parallel for-loops Exercise

- Use parallel constructs and serial code in the same function

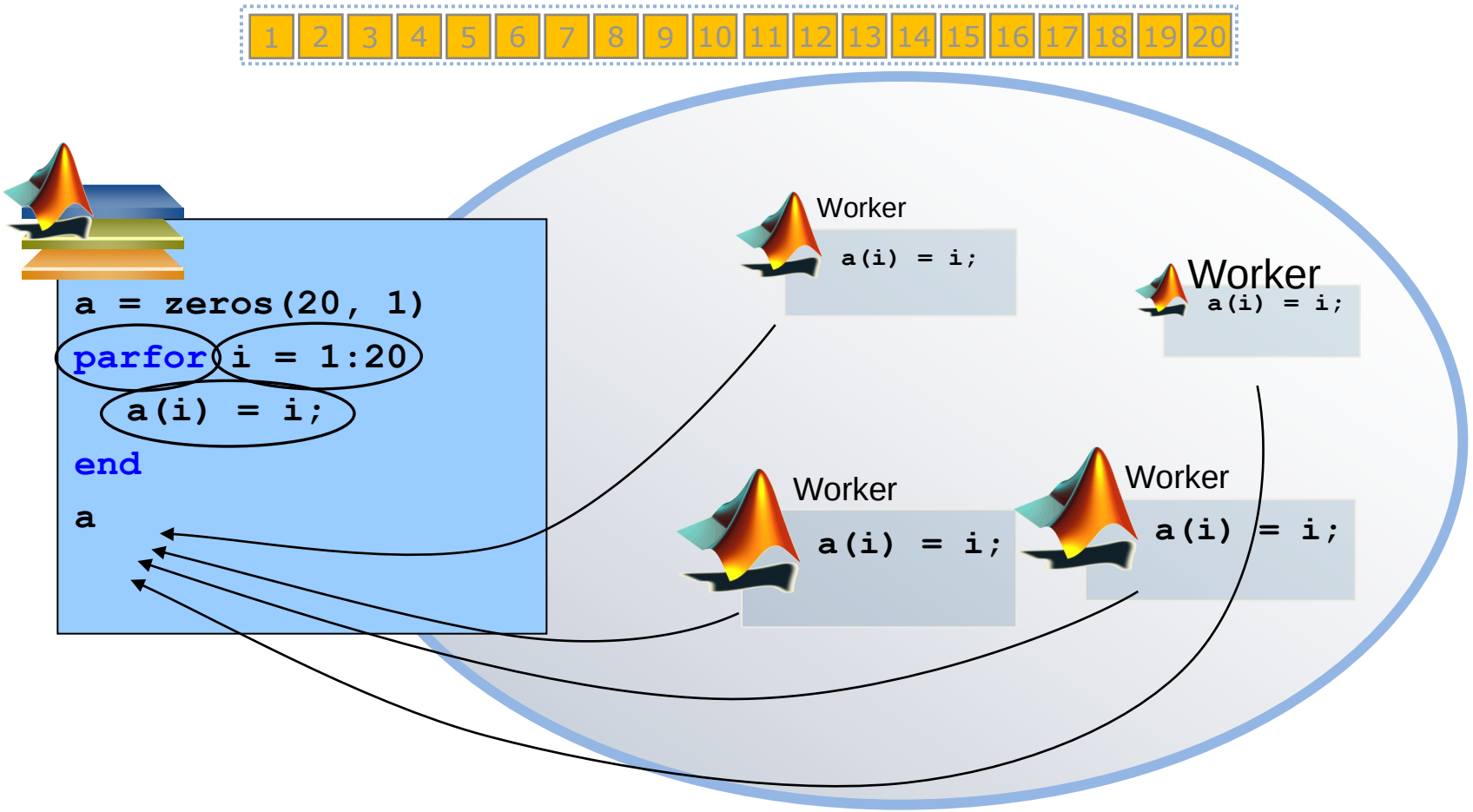
$$\overbrace{5}^{}\ m\ddot{x} + \underbrace{b}_{1,2,\ldots}\ \dot{x} + \underbrace{k}_{1,2,\ldots}\ x = 0$$

- Utilize pool of MATLAB Workers

- Used M-Lint analysis to help convert existing `for`-loop to `parfor`-loop



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

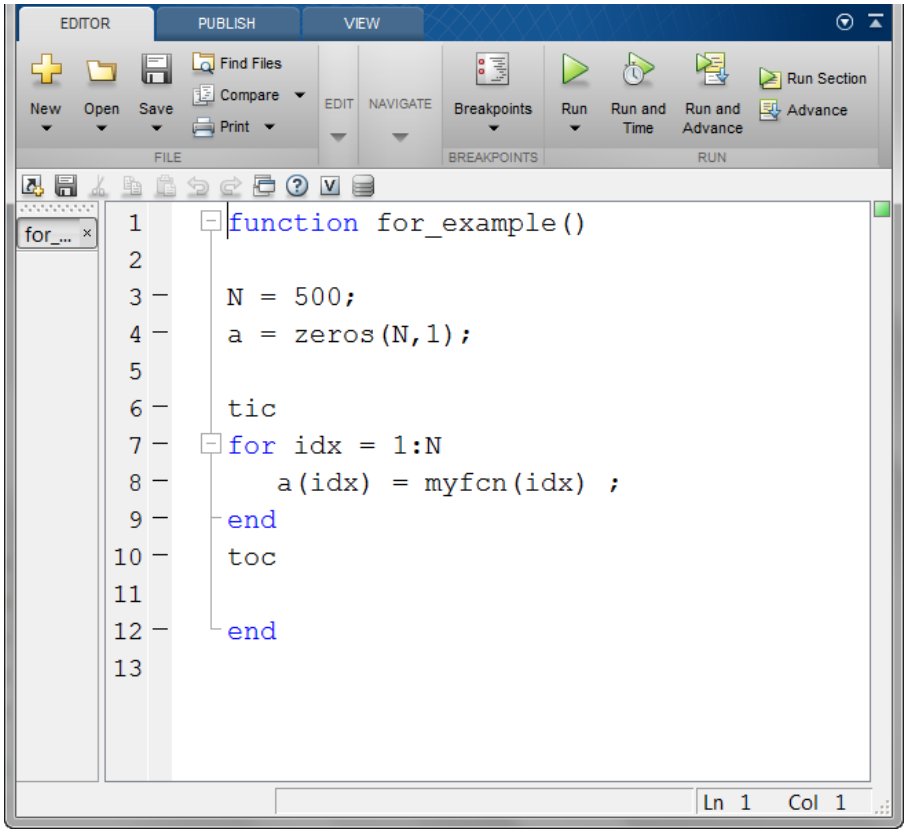
http://blogs.mathworks.com/loren/2009/10/02/using-parfor-loops-getting-up-and-running/

Considerations when using `parfor`

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

Exercise: `parfor` (1)

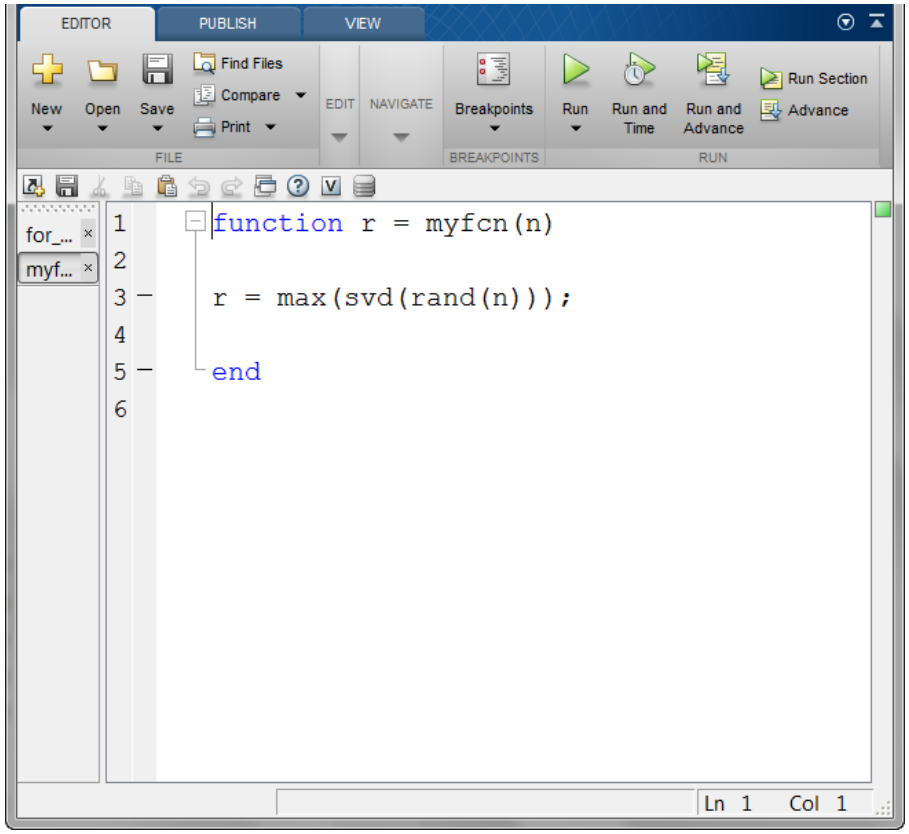
1. Code the example below. Save it as **for_example.m**




```
>> for_example
```

Exercise: `parfor` (2)

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




```
>> myfcn
```

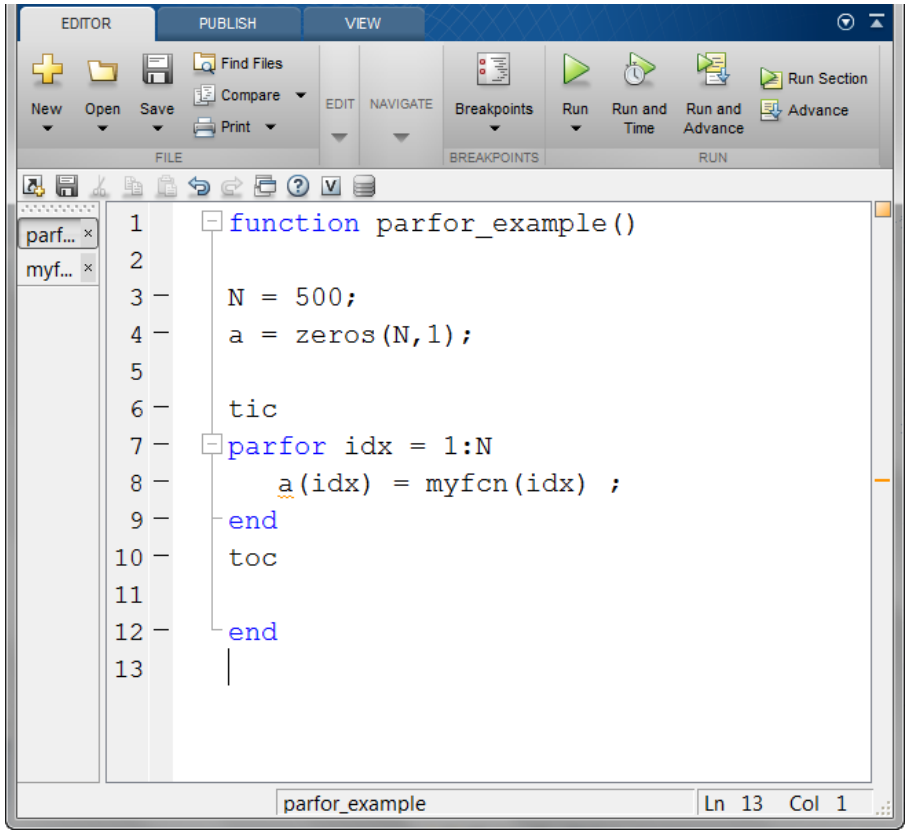
Exercise: `parfor` (3)

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
```

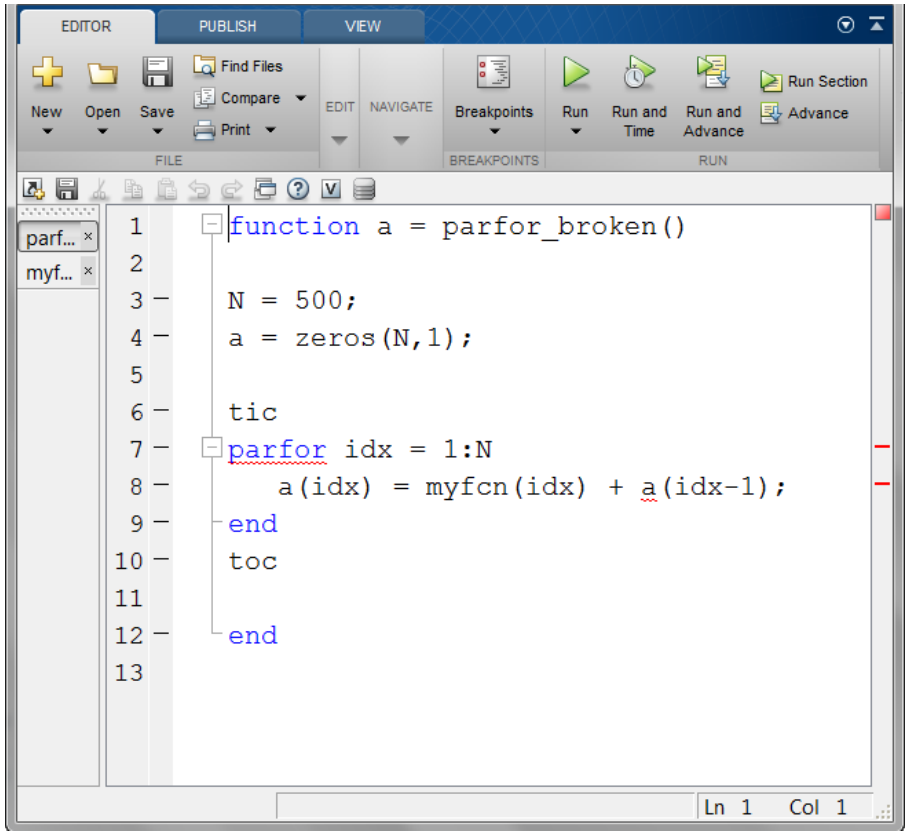
Exercise: `parfor` (3)




```
>> parfor_example
```

Exercise: `parfor` (4)

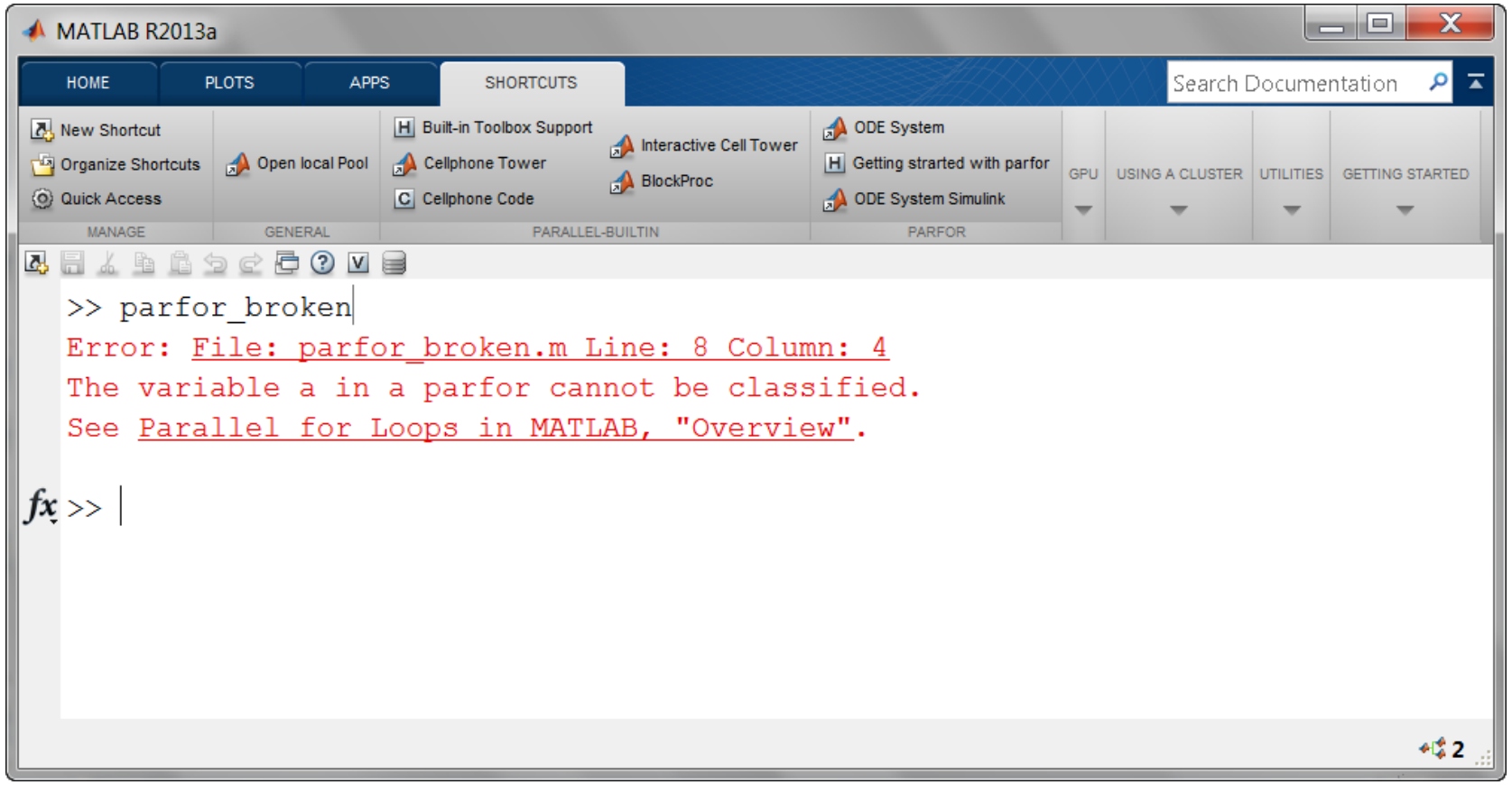
5. Break the **`parfor`** loop, see Code Analyzer messages




```
>> parfor_broken
```

Problem: Unclassified Variables

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



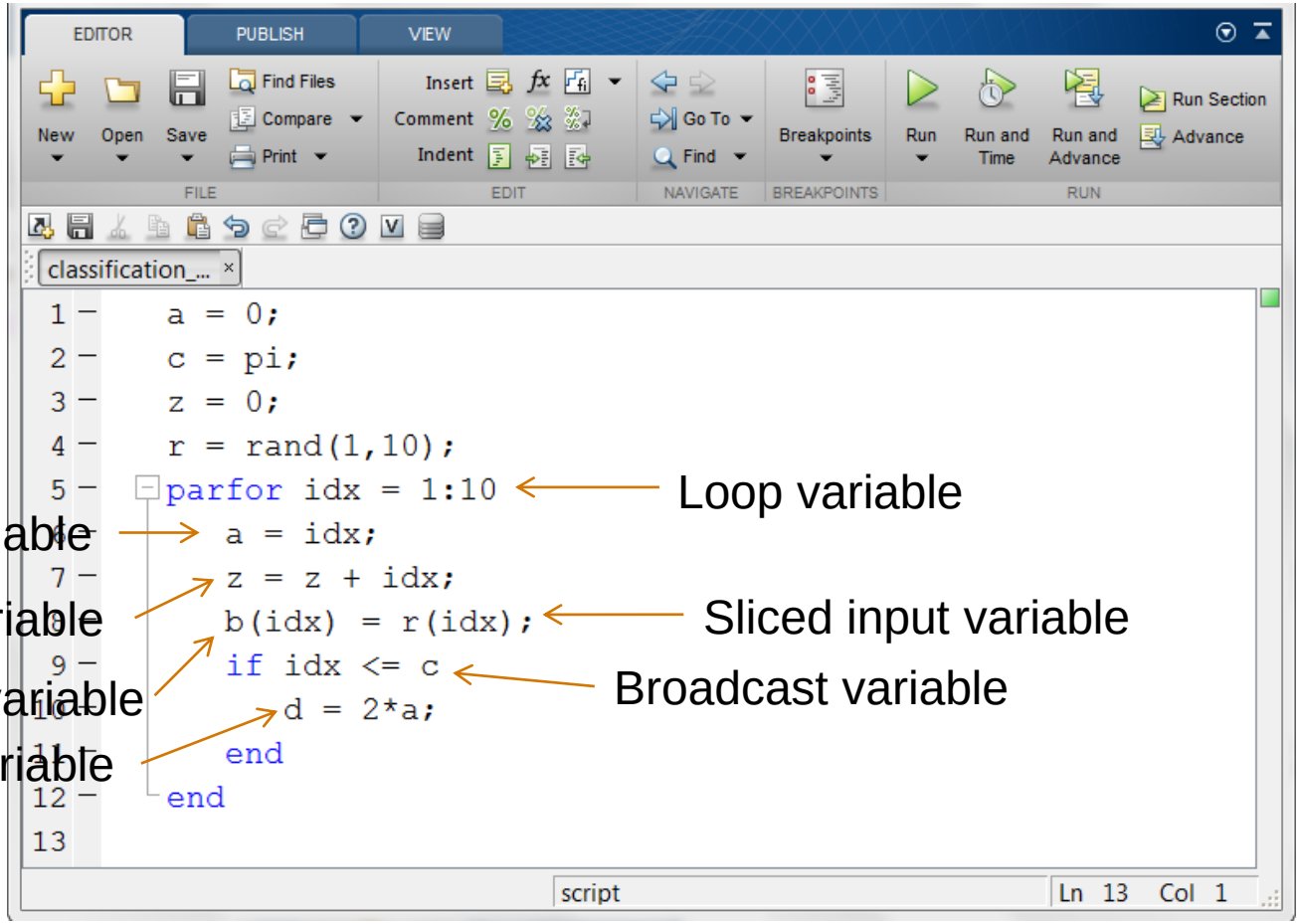
`parfor` Variable Classification

- 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

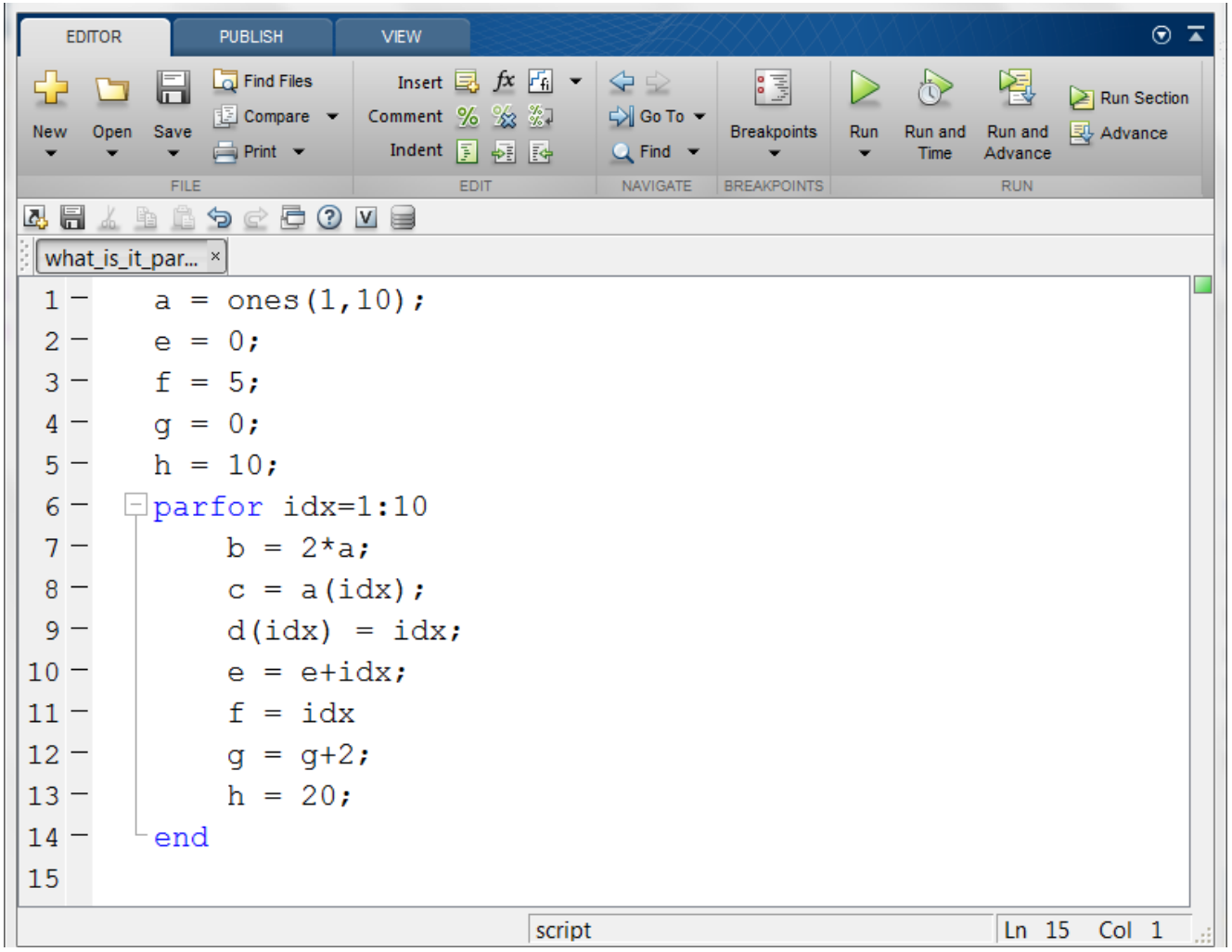
http://www.mathworks.com/help/distcomp/advanced-topics.html#bq_of7_-1

Exercise: Variable Classification Example (1)



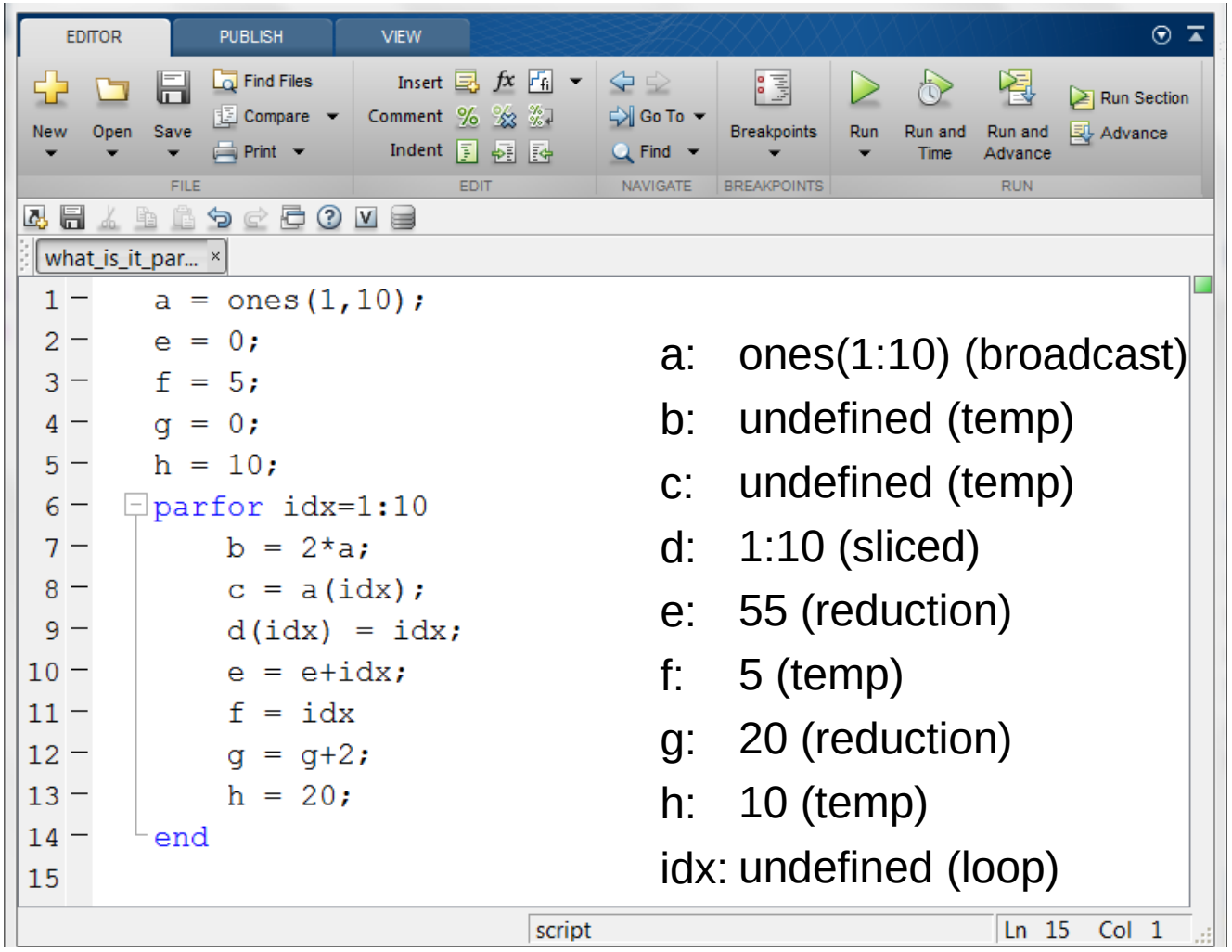

```
>> classification_example
```

Exercise: Variable Classification Example (2)




```
>> what_is_it_parfor
```

Exercise: Variable Classification Example (2)

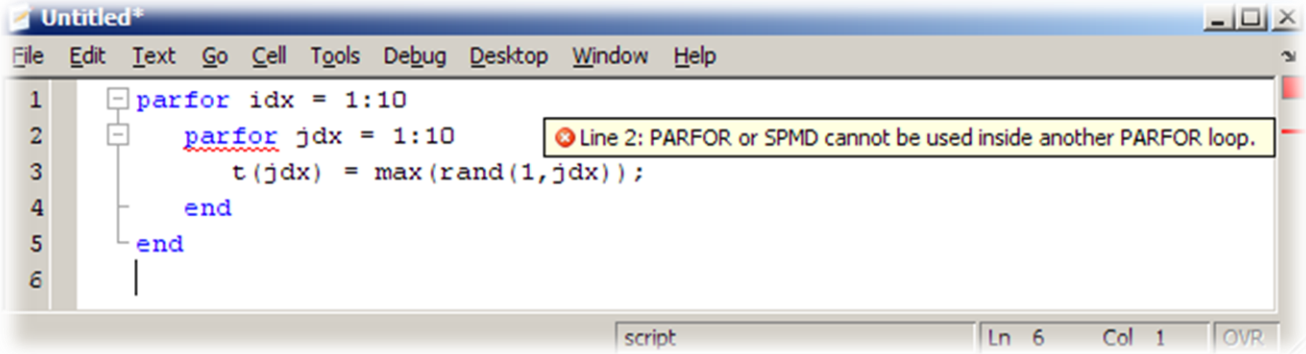


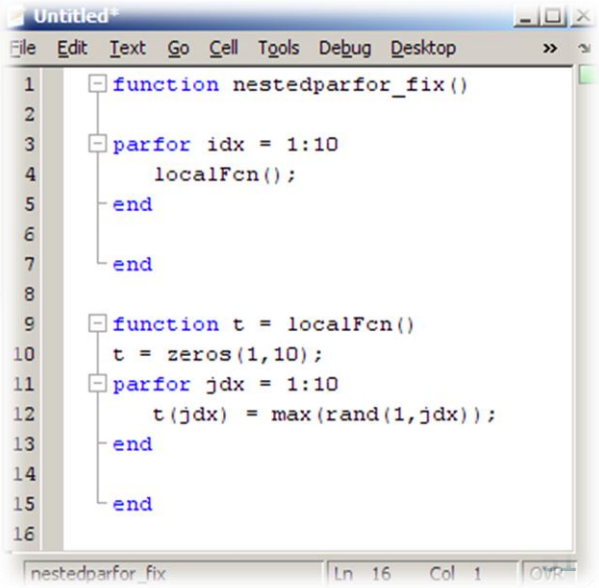
a: ones(1:10) (broadcast)
b: undefined (temp)
c: undefined (temp)
d: 1:10 (sliced)
e: 55 (reduction)
f: 5 (temp)
g: 20 (reduction)
h: 10 (temp)
idx: undefined (loop)

```
>> 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.






```
>> 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.

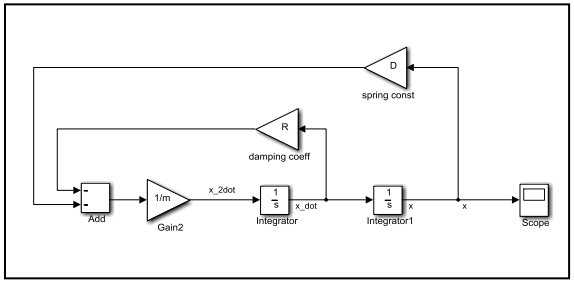
- Within main `parfor` body: Use `'base'` workspace
- Use `assignin` to place variables in base workspace.
- Note: the base workspace when using parfor is different than the base workspace when running serially.

`task_parallel\simParforEx1.m`

Exercise: Parallel for-loops with Simulink

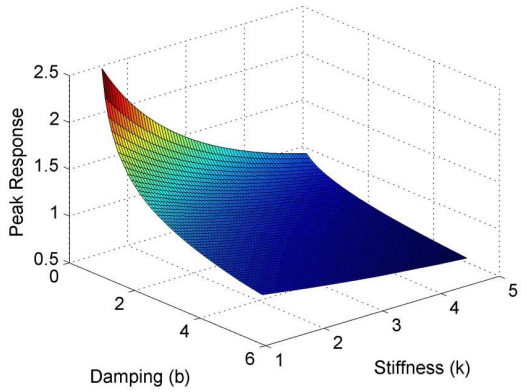
- Parameter sweep of ODE system
  - Damped spring oscillator in Simulink
  - Sweep through different values of damping and stiffness
  - Record peak value for each simulation

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

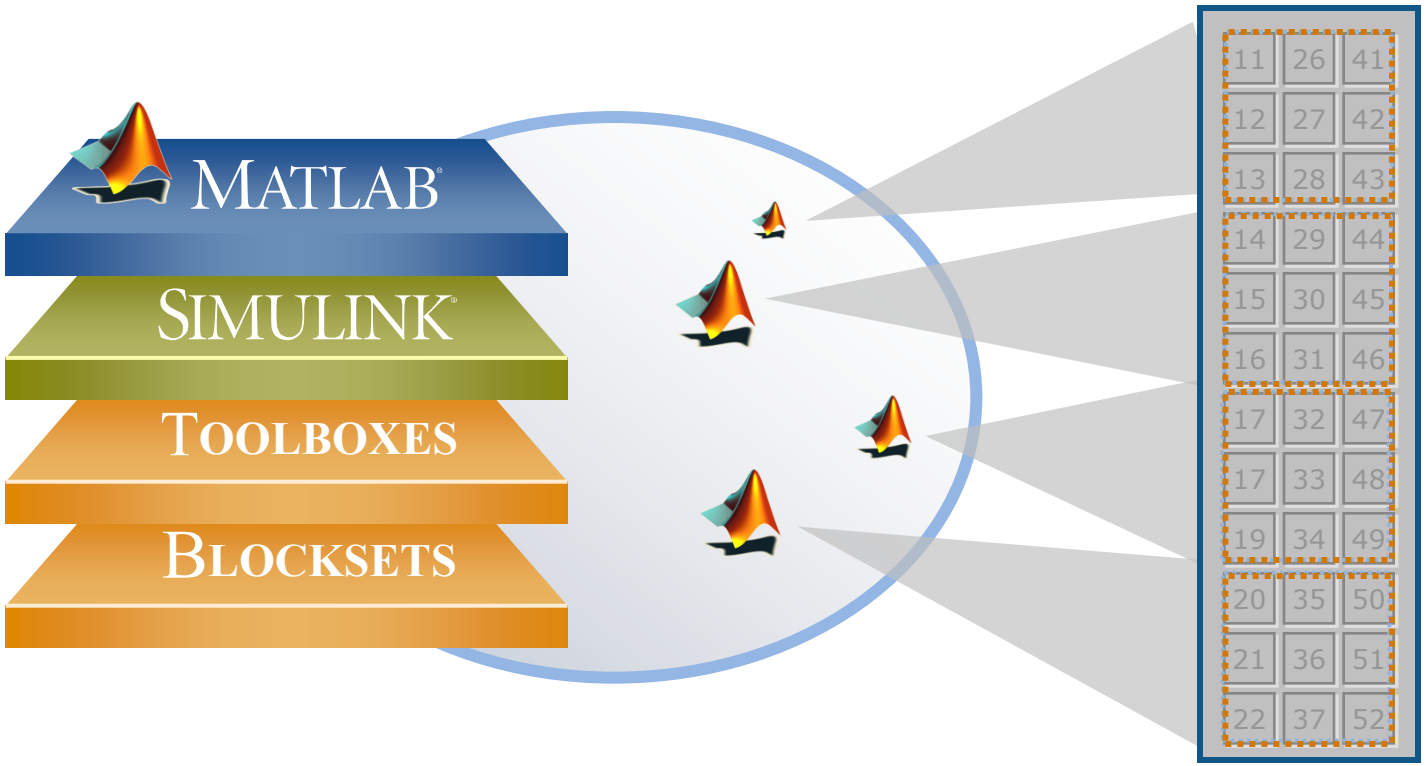


- Convert `for` to `parfor`

- Use pool of MATLAB workers



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
```

```
+LAlib   := -L$(LAdir)/lib -lf77blas -latlas -lgfortran
+
+# F2C options
+
+F2CDEFS        := -DAdd__ -DF77_INTEGER=int -DStringSunStyle
+
+# HPL options
+
```

Using FORTRAN and MPI

```
+HPL_DEFS += -DHPL_DETAILED_TIMING
+endif
+
+HPL_LIBS := $(HPLlib) $(LAlib) $(MPlib) $(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
+ACLlib  := $(TOPdir)/accel/lib/libhpl_accel_ppu.a
+
```

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

Programming Parallel Applications (CPU)

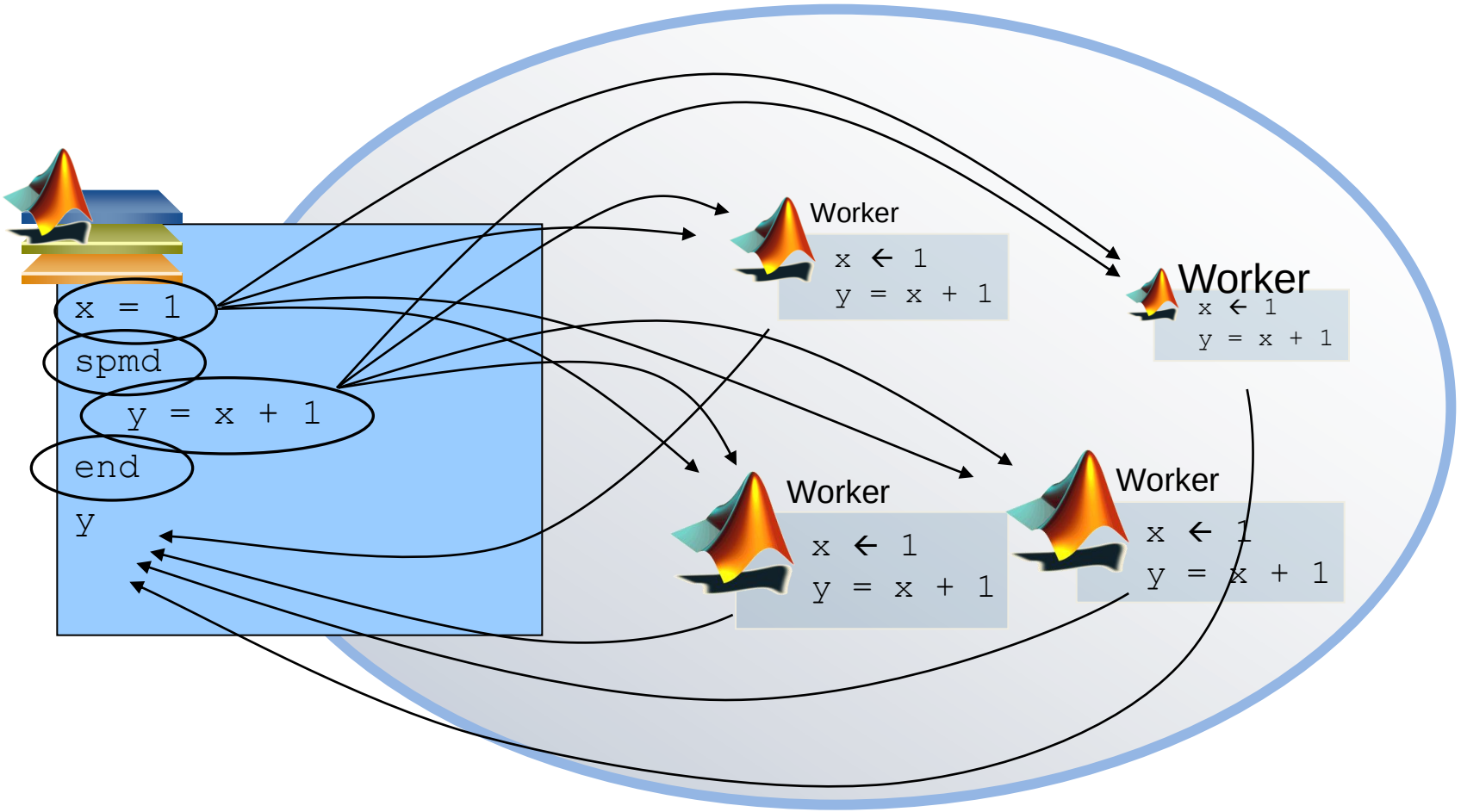
- Built-in support with Toolboxes
- Simple programming constructs:
  `parfor, batch, distributed`
- Advanced programming constructs:
  `createJob, labSend, spmd`

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

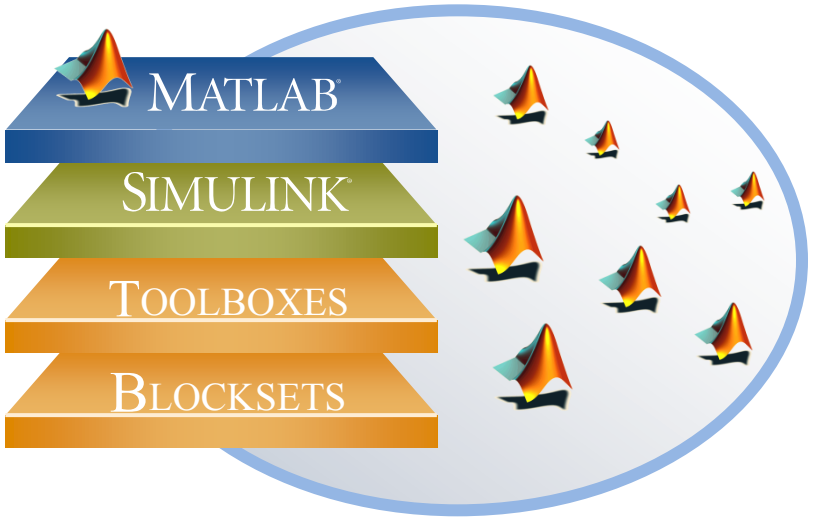


`spmd`

- 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

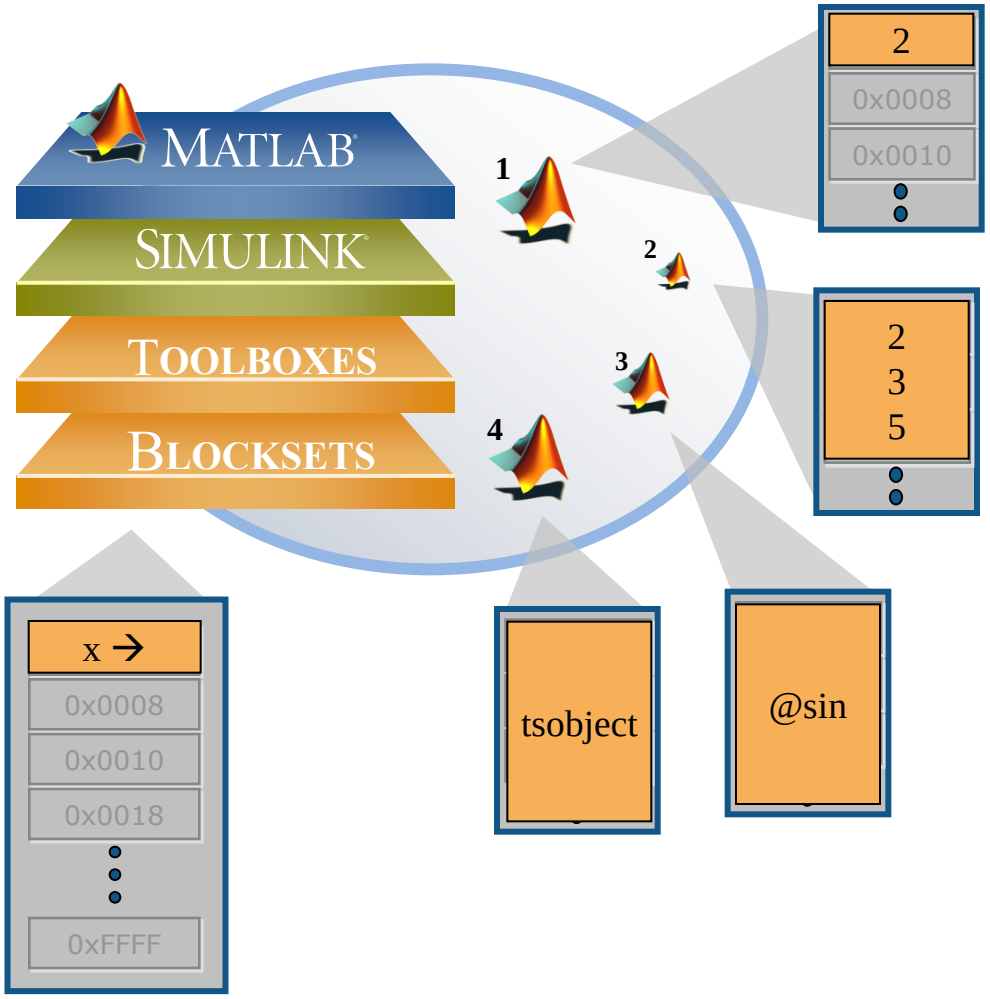
Composite Arrays

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



Composite Array in Memory

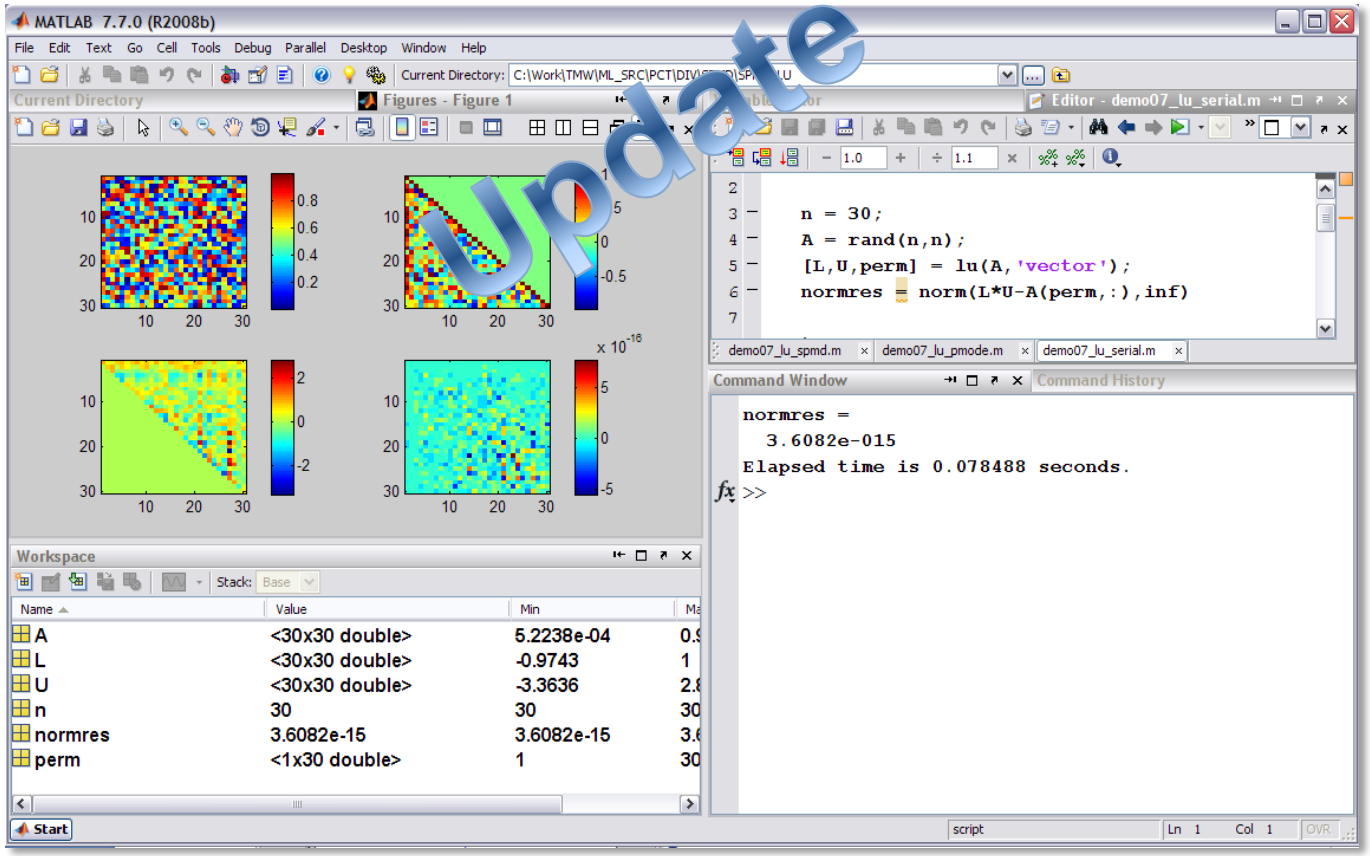
```
>> matlabpool open 4

>> x = Composite(4)

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



Exercise: Distributed Arrays

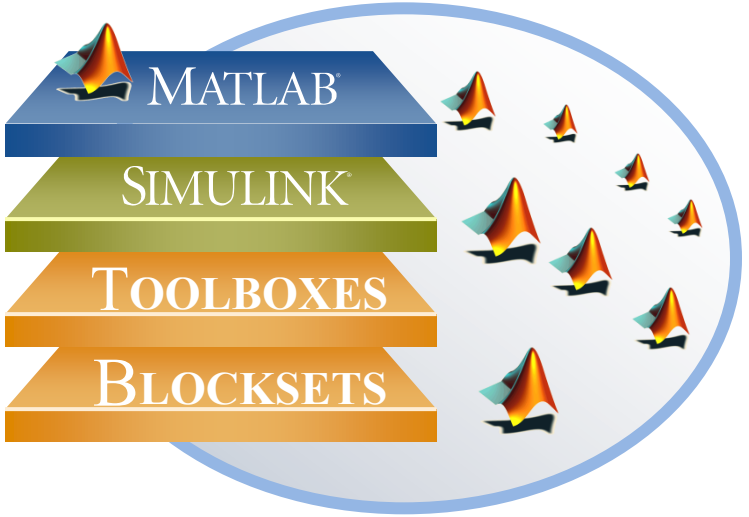
Convert the serial code to run data parallel

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

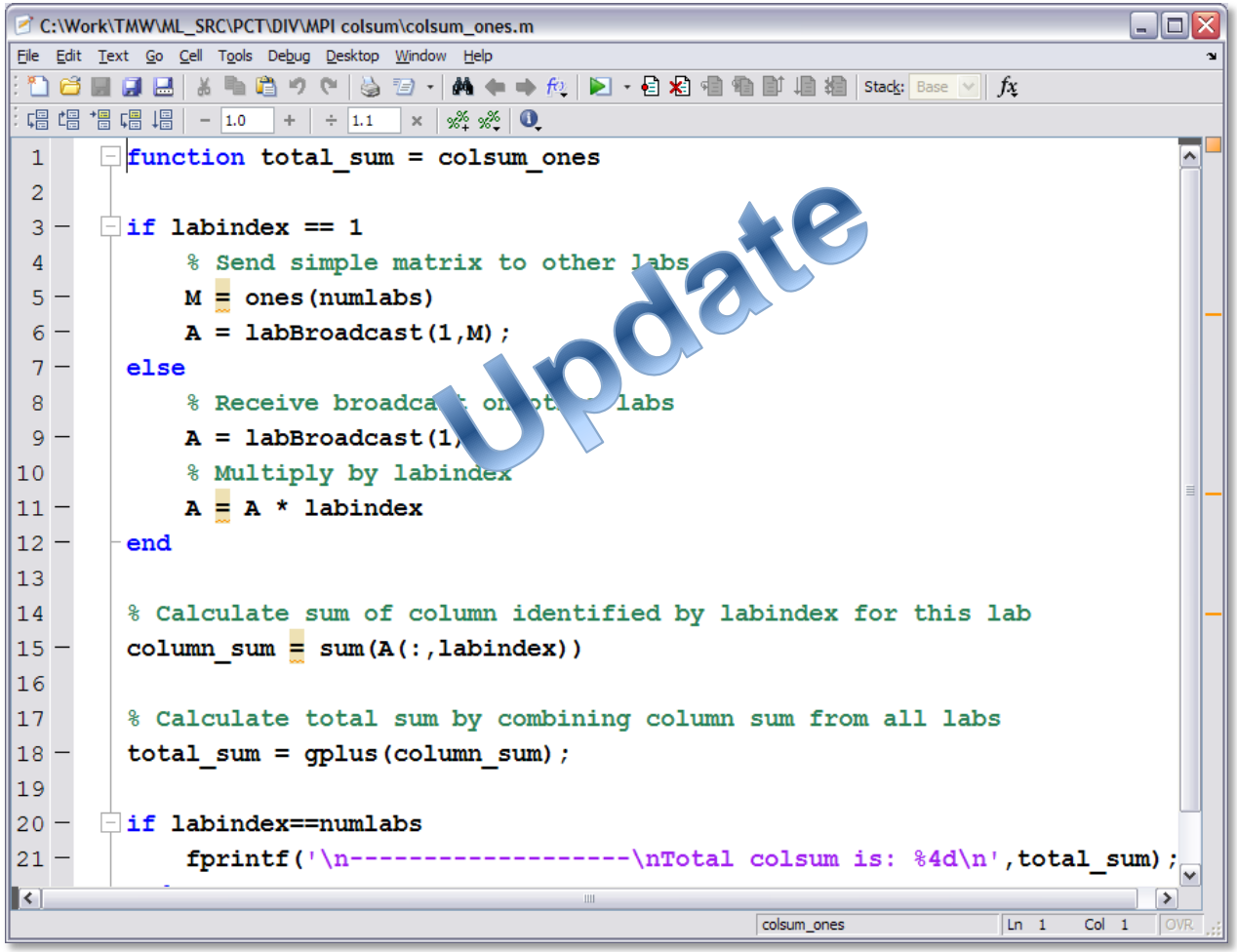
MPI-Based Functions: Parallel Computing Toolbox

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

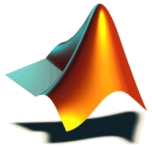
```
>> mpi_example
```

Exercise: MPI-based Functions



Agenda

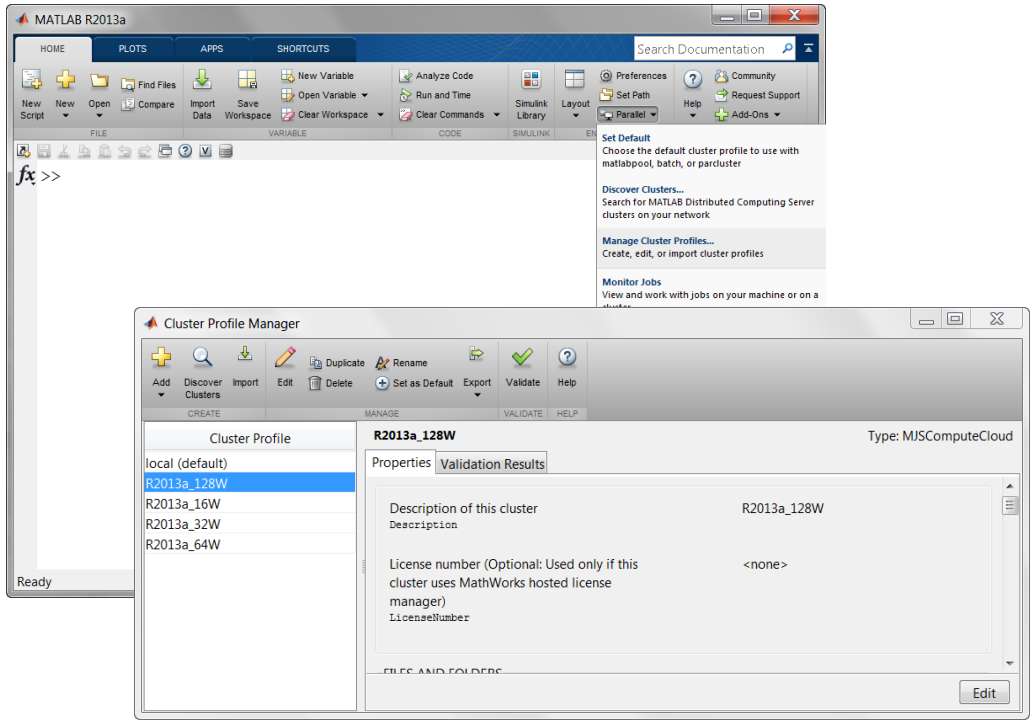
- Best practices in MATLAB programming
- Introduction to parallel computing tools
- Constructs for multicore/multi-processor computers



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

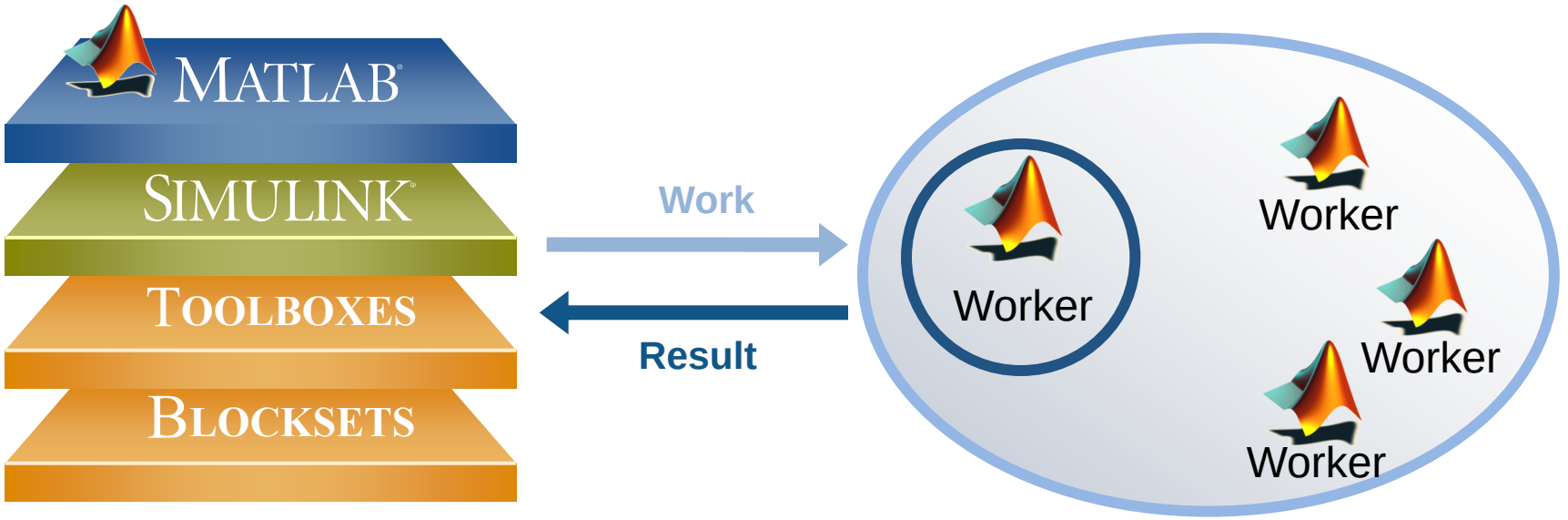
Glossary (3)

- 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

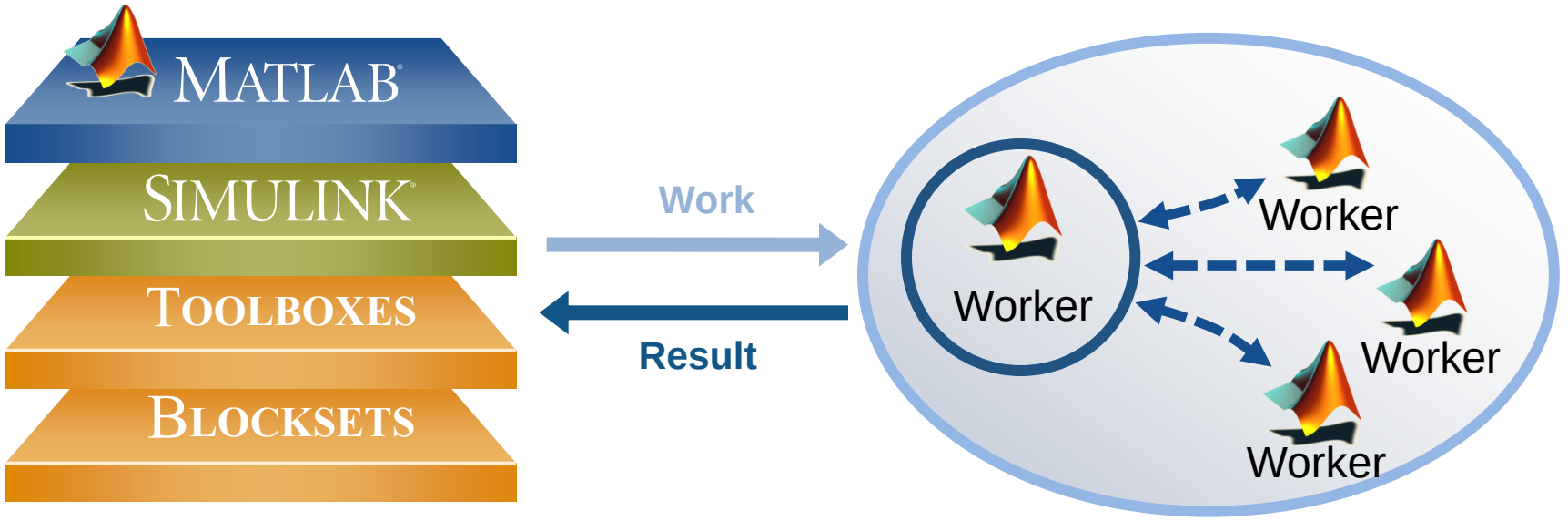
Interactive to Scheduling

- 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



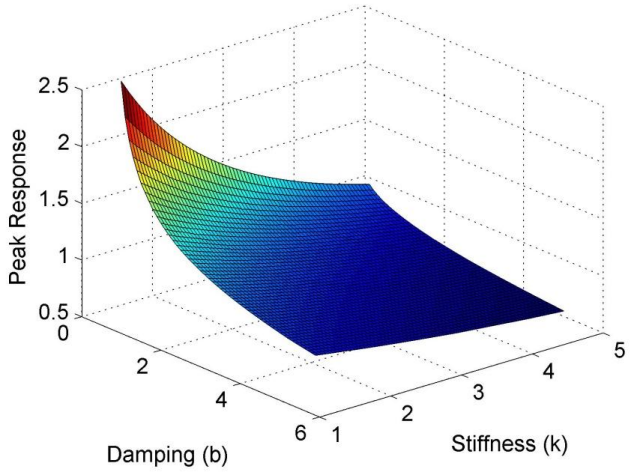
Offload parallel computations with `batch`



Exercise: Schedule Processing with `batch`

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

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



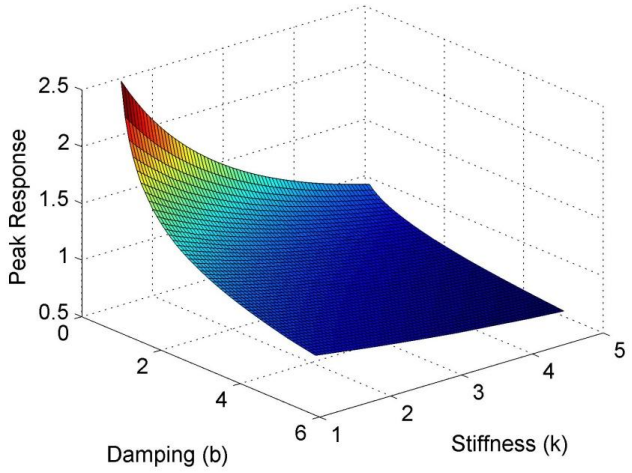
Summary: Schedule Processing with `batch`

- Offload processing to workers
  - `batch`
  - `matlabpool`

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

- Monitor progress of scheduled job
  - **Job Monitor**

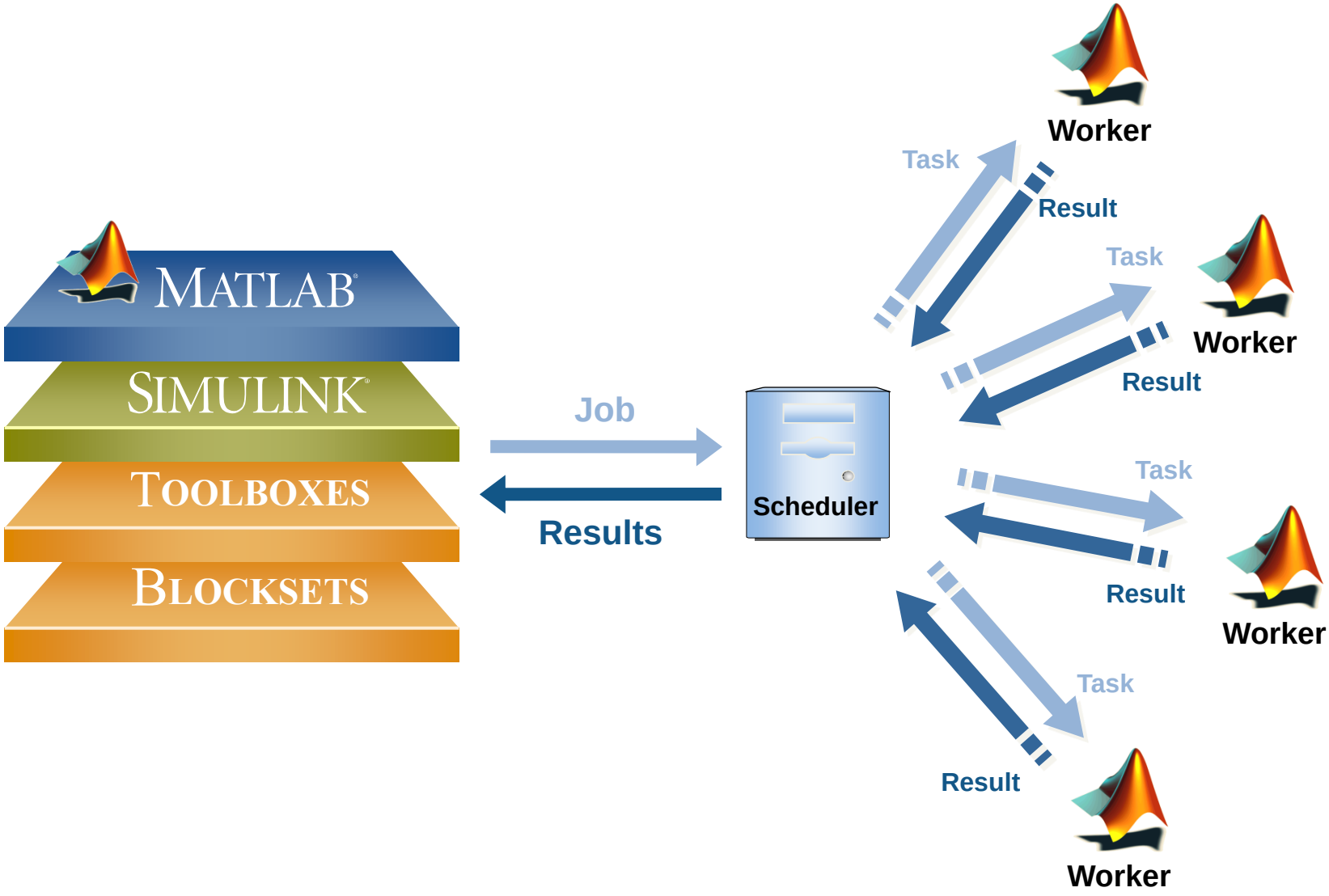
- Retrieve results from job
  - `fetchOutputs`



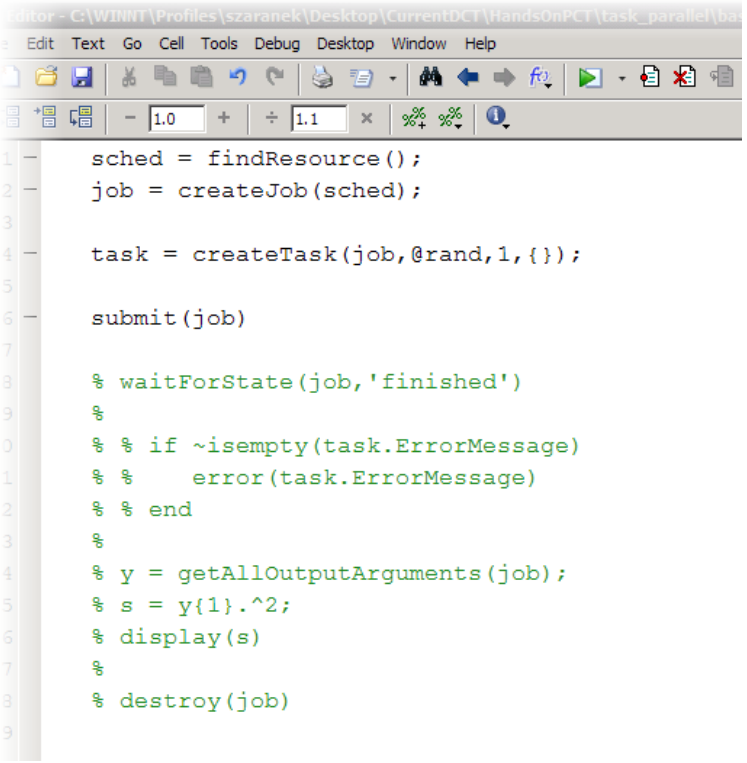
Creating jobs and tasks

- 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

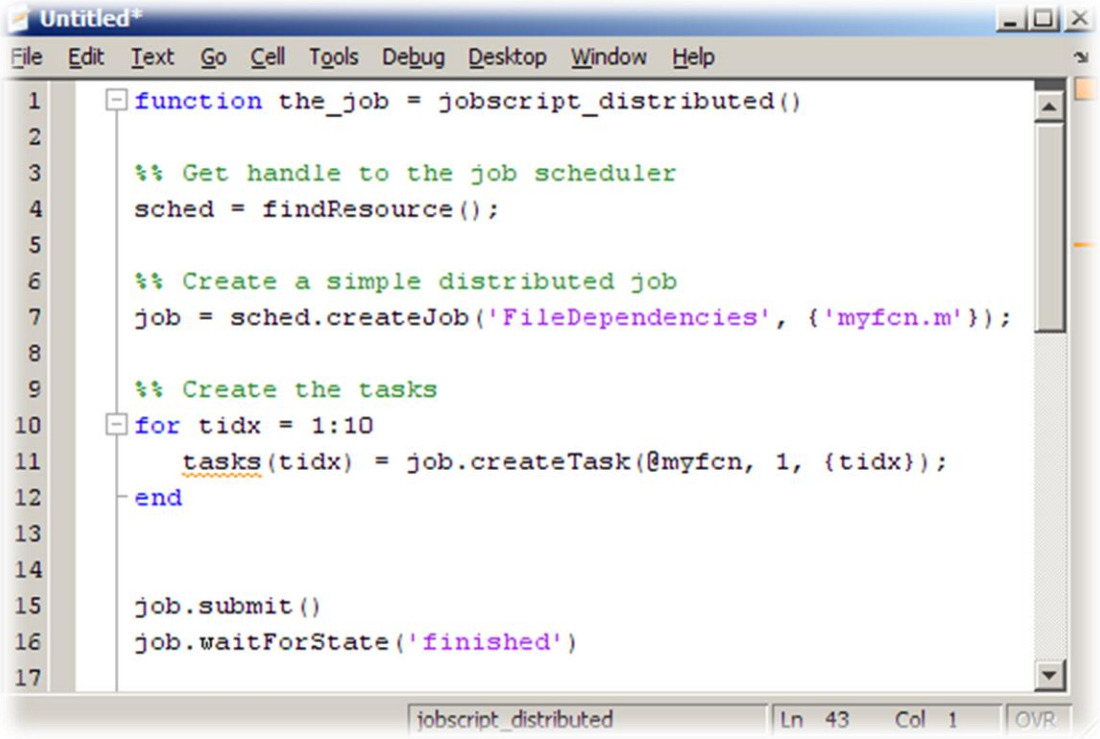



```
sched = findResource();
job = createJob(sched);

task = createTask(job,@rand,1,{});

submit(job)

% waitForState(job,'finished')
%
% % if ~isempty(task.ErrorMessage)
% %     error(task.ErrorMessage)
% % end
%
% y = getAllOutputArguments(job);
% s = y{1}.^2;
% display(s)
%
% destroy(job)
```

Rather than using a shell script to submit a job to a cluster, we'll write our *jobscript* in MATLAB.

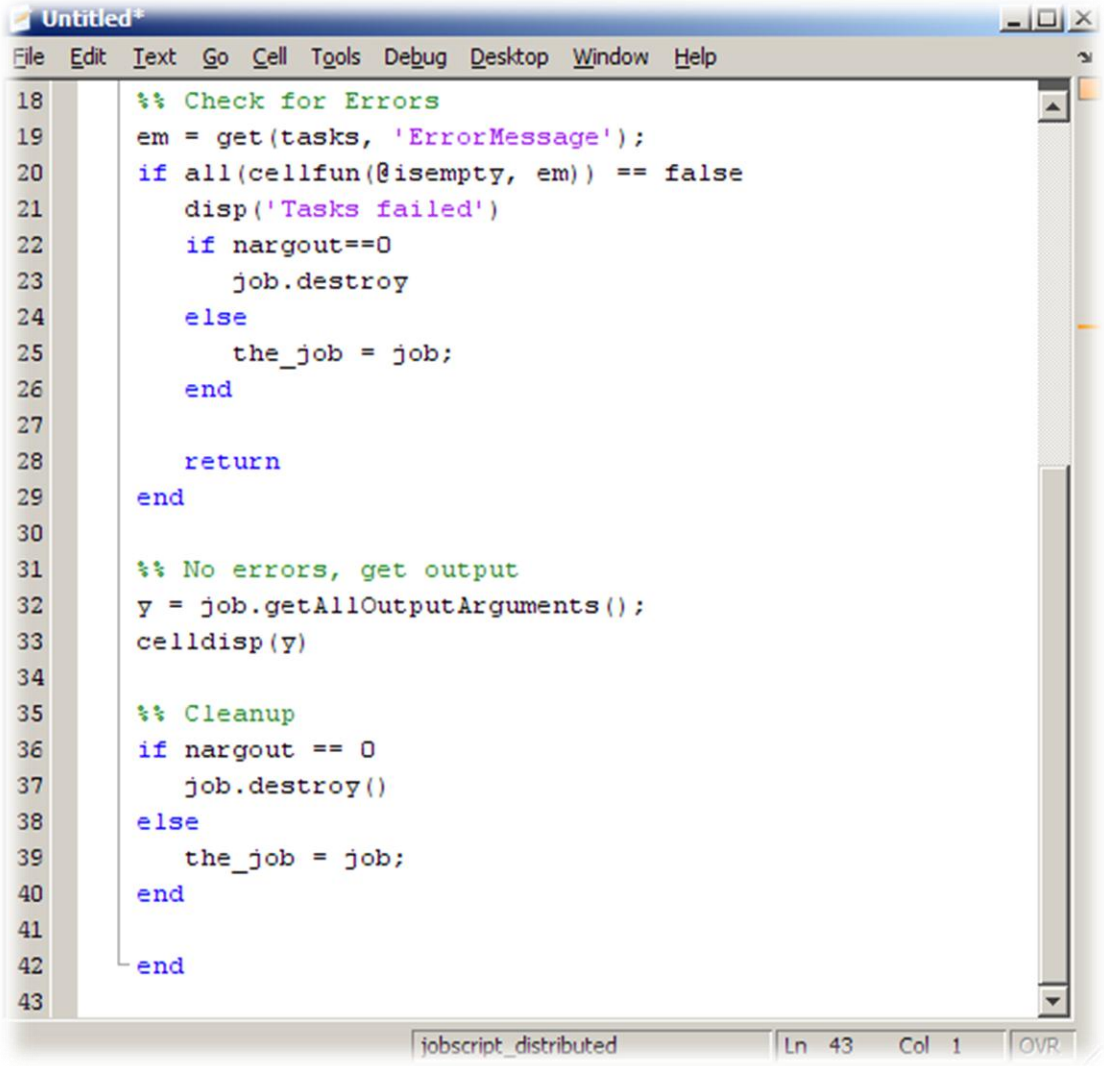
`task_parallel\basic_jobscript.m`

Exercise: A simple jobscript (1)




```
>> jobscript_distributed
```

Exercise: A simple jobscript (2)



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

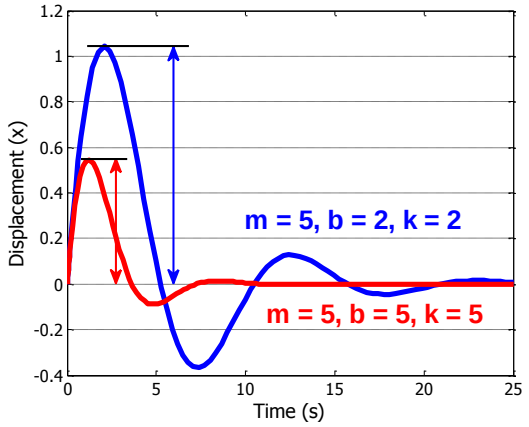
Factors to Consider for Scheduling

- 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

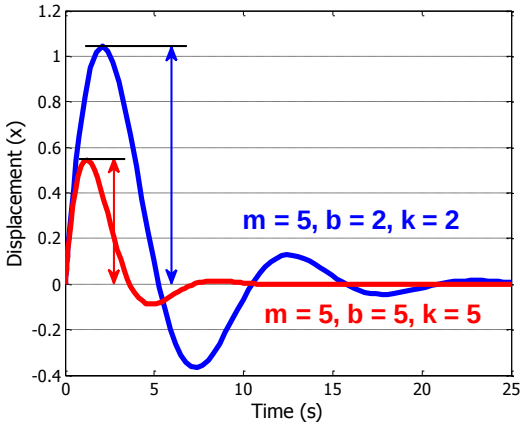
$$\underbrace{m}_{5}\ddot{x} + \underbrace{b}_{1,2,\ldots}\dot{x} + \underbrace{k}_{1,2,\ldots}x = 0$$



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`

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



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