Introduction to MDCS

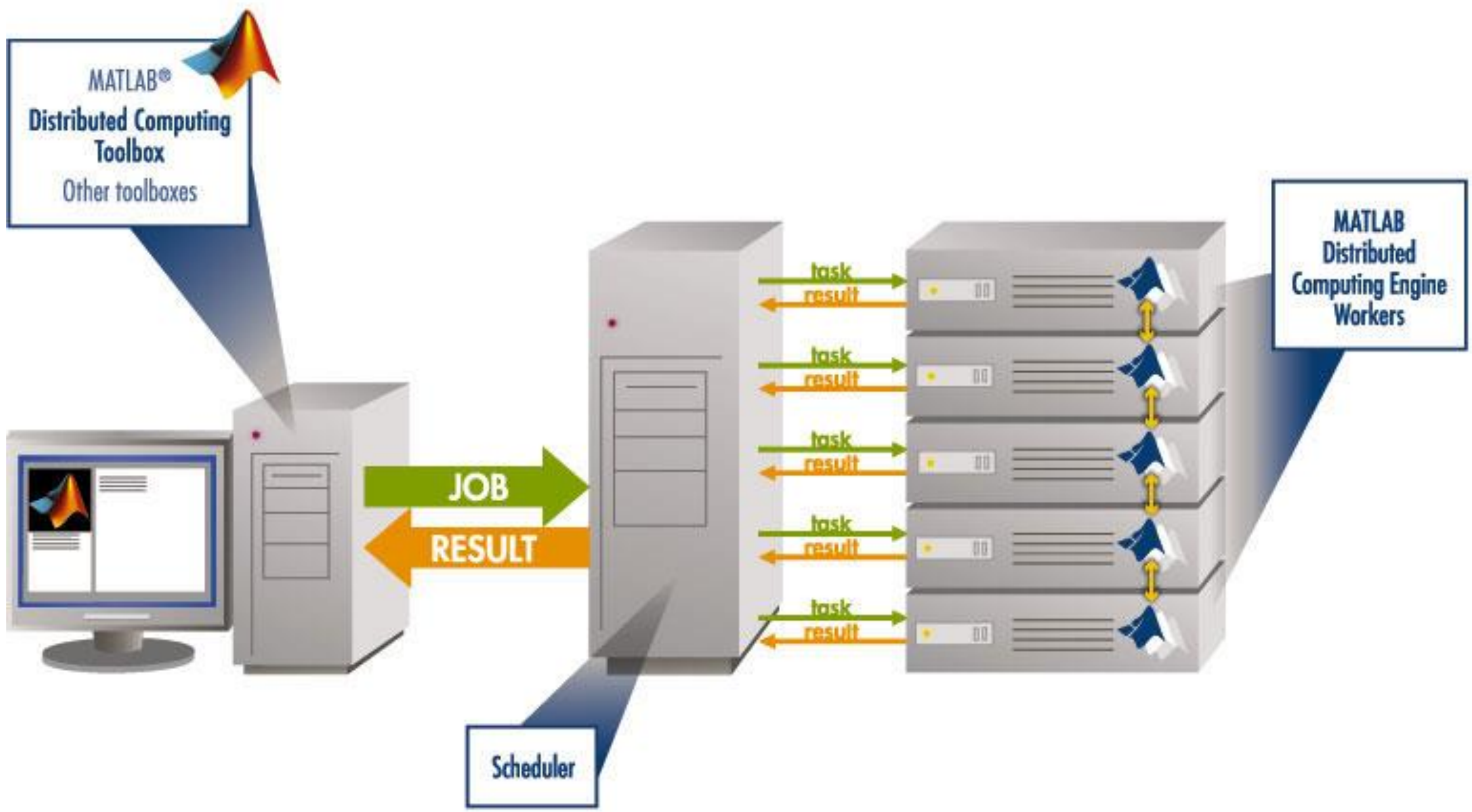
- Matlab Distributed Compute Server
- Preparing Matlab for MDCS
- Example

What is MDCS

Matlab on your desktop computer:

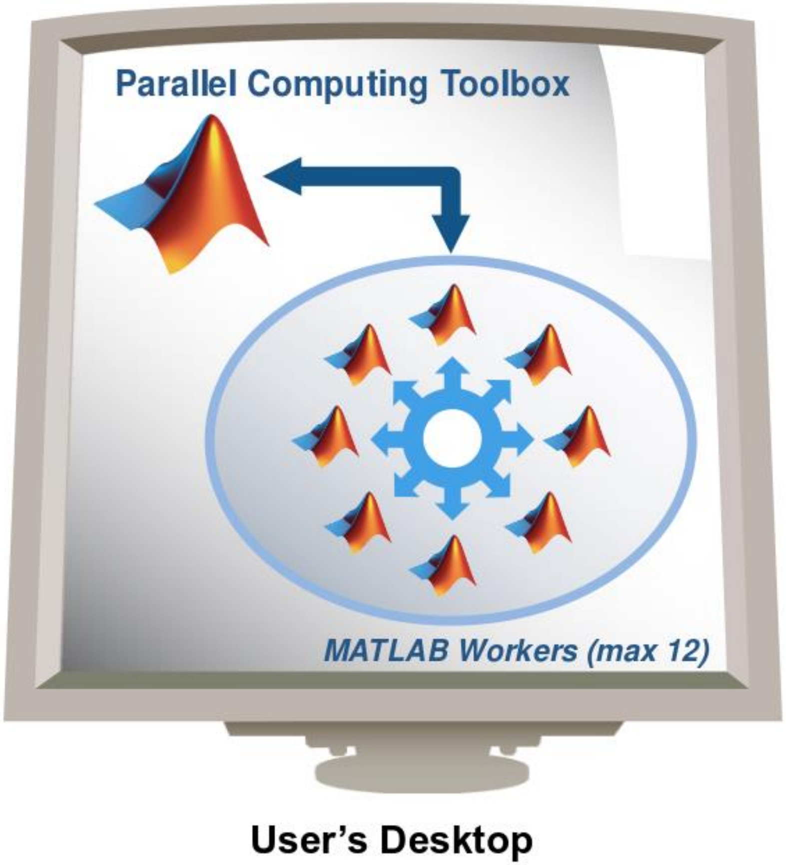
- you are limited by the compute power of your local machine
  - memory
  - CPU speed
- you can only run one job at a time
- your machine may become unusable while your Matlab job is running

What is MDCS


MATLAB®
Distributed Computing Toolbox
Other toolboxes
JOB
RESULT
task
result
task
result
task
result
task
result
task
result
MATLAB Distributed Computing Engine Workers
Scheduler

(taken from MathWorks marketing)

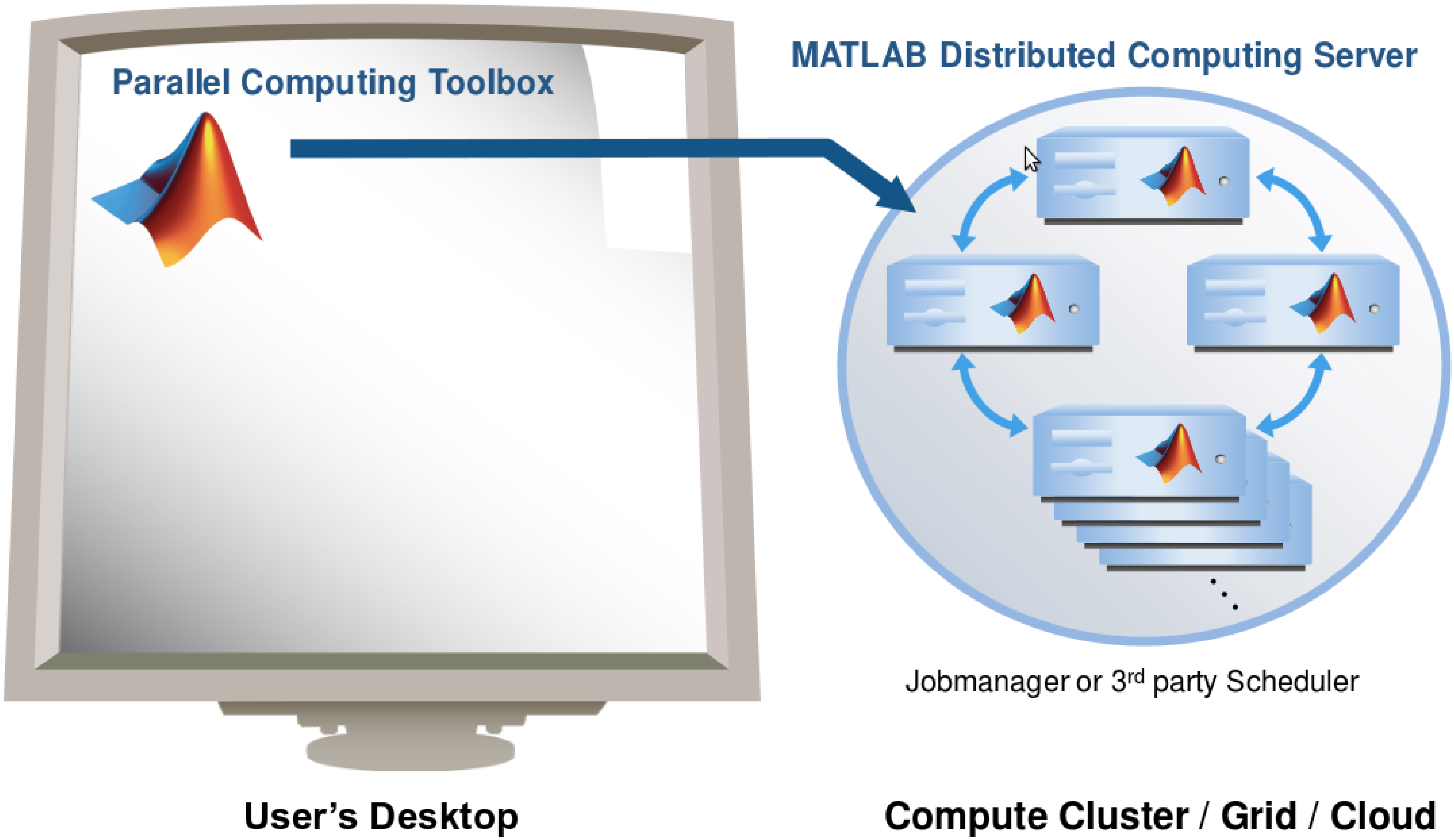
Parallel Computing with Matlab



- easily experiment with explicit parallelism on multicore machines
- rapidly develop parallel applications on local computer
- take full advantage of desktop power, incl. GPUs
- separate compute cluster not required

(taken from MathWorks marketing)

Parallel Computing with Matlab



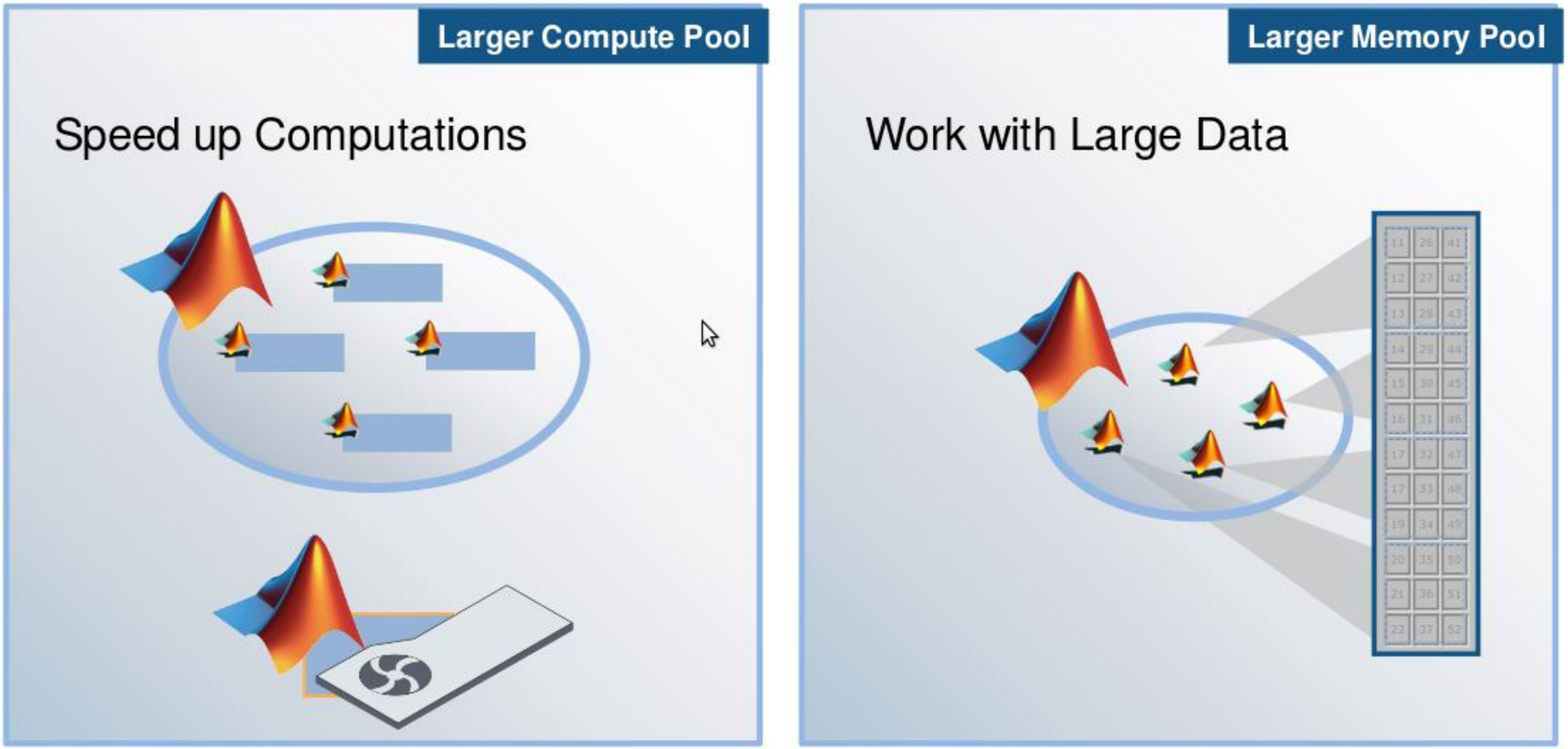
What is MDCS

- MDCS allows you to off-load Matlab programs to a compute server
- simplified workflow
  - you can develop and test your application locally before submitting jobs, also in parallel
  - results are automatically returned to your local machine for post-processing
- the Parallel Computing Toolbox provides utilities for parallelization
  - task-parallel
  - data-parallel

Why to use MDCS on the Cluster?

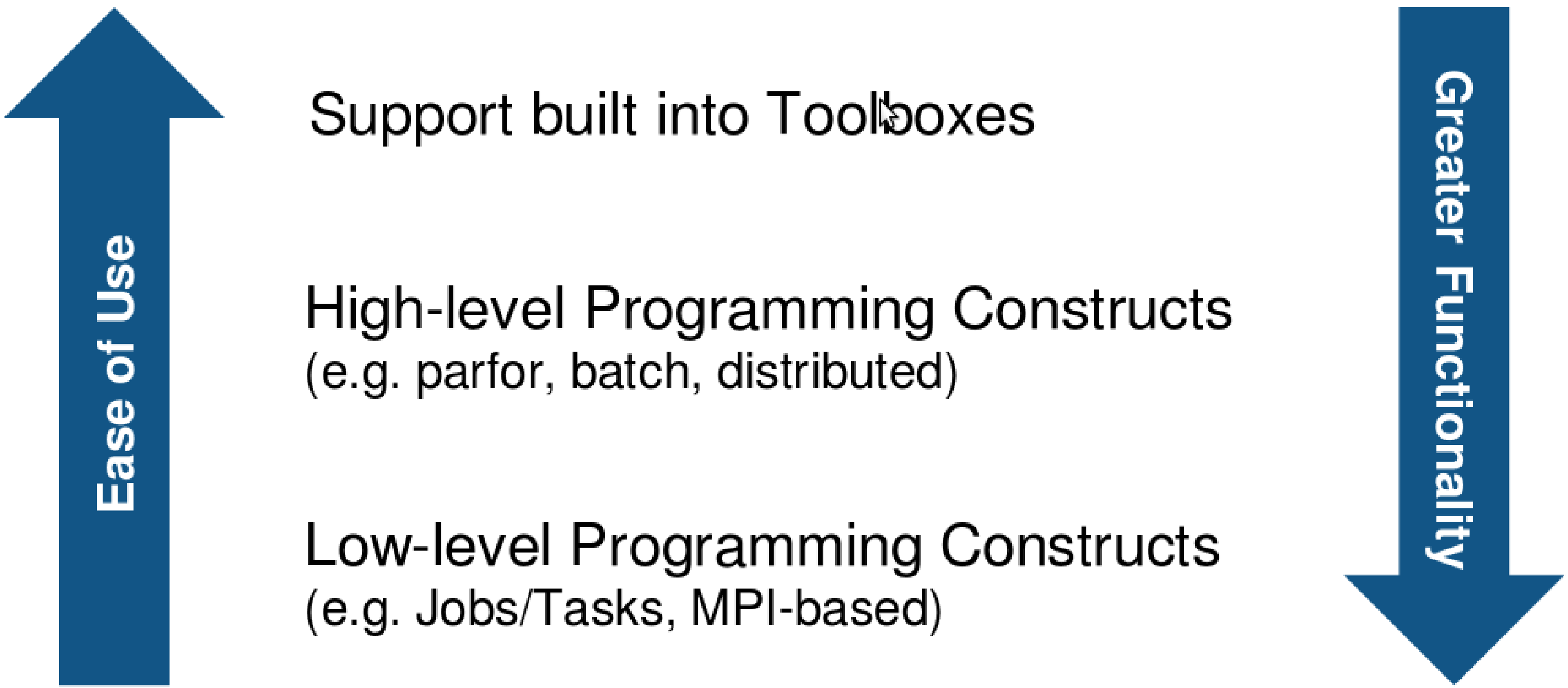
- with MDCS come 224 worker licenses
  - these are in addition to the normal Matlab licenses (200)
  - you can use also any of the toolboxes (50)
  - allows the control over used licenses and prevents failed jobs
  - for fair sharing not more than 36 MDCS licenses should be used

Parallel Computing with Matlab



Parallel Computing with Matlab

Three levels of Integration:



Parallel Computing Support in Toolboxes

- Optimization Toolbox
- Global Optimization Toolbox
- Statistics Toolbox
- Simulink Design Optimization
- Bioinformatics Toolbox
- Communications Toolbox
- Model-Based Calibration Toolbox
- ... and more

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

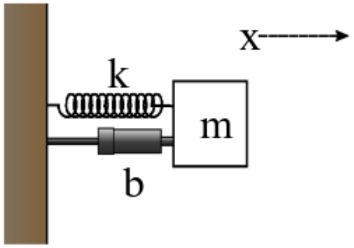
Using MDCS on FLOW/HERO

- before you can use MDCS a few preparations are needed (only needed to be done once)
  - Matlab needs to be installed (see local web page) on your local machine, only versions R2010b, R2011a, R2011b are licensed for MDCS
  - your local machine must be able to login to FLOW/HERO via ssh
    - Linux/Mac have ssh per default, for Windows you can use PuTTY
    - if you are not in the university network you also need to connect to a VPN (see HPC-Wiki for details)
  - a number of files (from a zipped archive from the HPC-Wiki) have to copied to your local Matlab directory (depending on the setup of your local machine, your system admin has to help you)
  - a parallel configuration has to be setup with Matlab

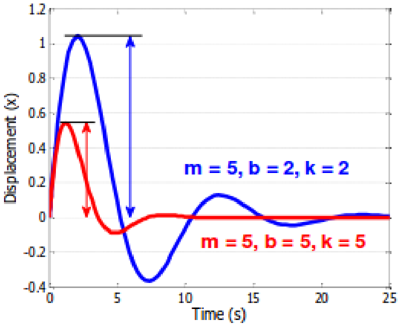
Using MDCS on FLOW/HERO

- once you have completed the setup you can submit jobs to the cluster
  - example parameter sweep for 2^{nd}-order ODE (taken from the HPC-Wiki)
  - dampened oscillator

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



  - simulate with different values for *b* and *k*
  - record peak value for each run



2nd-order ODE for example

odesystem.m

```
function dy = odesystem(t, y, m, b, k)
% 2nd-order ODE
%
%   m*X'' + b*X' + k*X = 0
%
% --> system of 1st-order ODEs
%
%   y  = X'
%   y' = -1/m * (k*y + b*y')
% Copyright 2009 The MathWorks, Inc.

dy(1) = y(2);
dy(2) = -1/m * (k * y(1) + b * y(2));

dy = dy(:); % convert to column vector
```

Parameter Sweep: serial Matlab code

paramSweep_batch.m

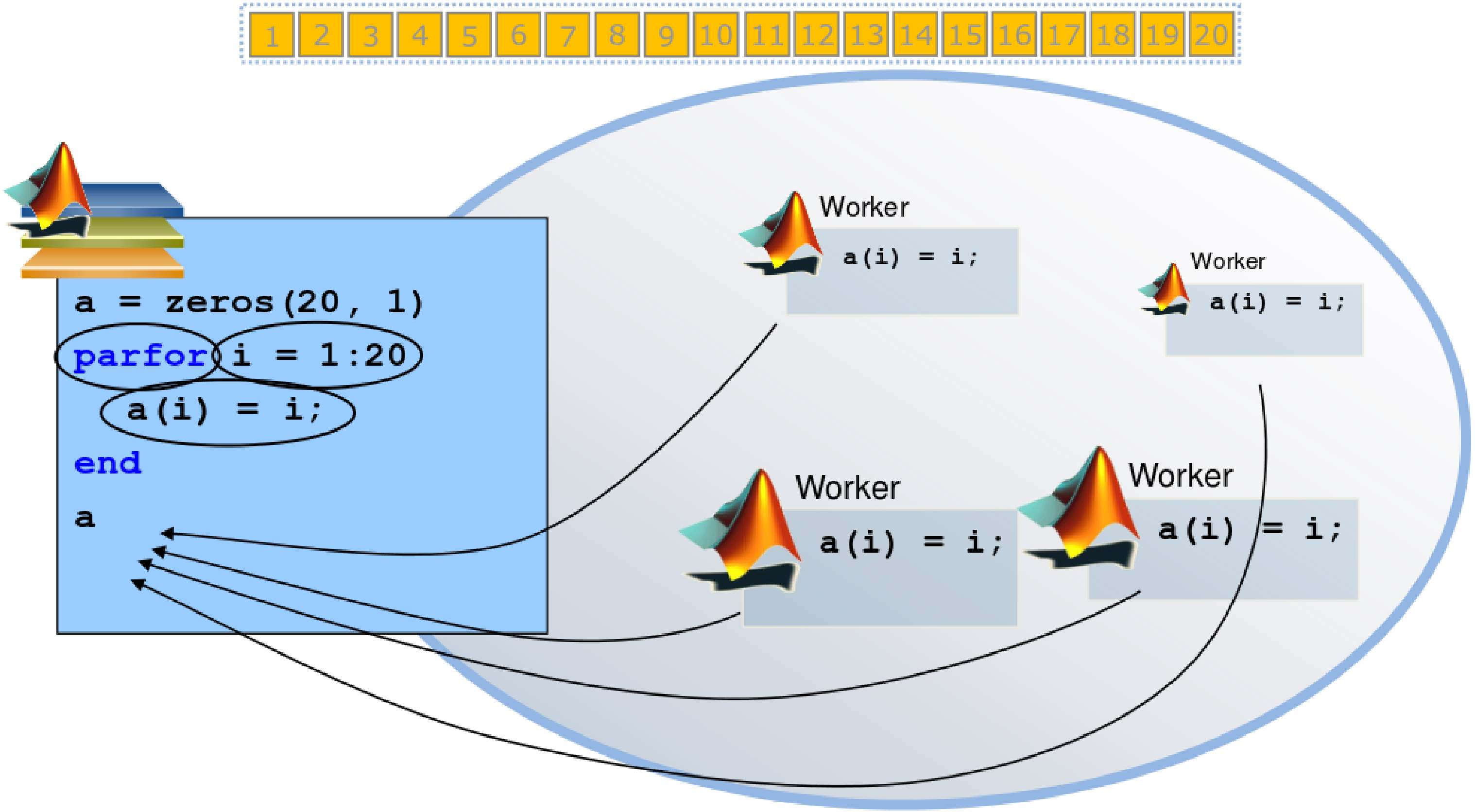
```
%% Initialize Problem
m      =         5;  % mass
bVals = 0.1:.1:15;  % damping values (step .1)
kVals = 1.5:.1:15;  % stiffness values (step .1) damping
[kGrid, bGrid] = meshgrid(bVals, kVals);
peakVals = nan(size(kGrid));

%% Parameter Sweep
tic;

for idx = 1:numel(kGrid)
  % Solve ODE
  [T,Y] = ode45(@(t,y) odesystem(t, y, m, bGrid(idx), kGrid(idx)), ...
    [0, 25], ...  % simulate for 25 seconds
    [0, 1]);      % initial conditions

  % Determine peak value
  peakVals(idx) = max(Y(:,1));
end

t1 = toc;
```

Parameter Sweep: parallel Matlab code

paramSweep_batch.m

```
%% Initialize Problem
m      =          5;  % mass
bVals = 0.1:.1:15;  % damping values (step .1)
kVals = 1.5:.1:15;  % stiffness values (step .1) damping
[kGrid, bGrid] = meshgrid(bVals, kVals);
peakVals = nan(size(kGrid));

%% Parameter Sweep
tic;

parfor idx = 1:numel(kGrid)
  % Solve ODE
  [T,Y] = ode45(@(t,y) odesystem(t, y, m, bGrid(idx), kGrid(idx)), ...
    [0, 25], ...  % simulate for 25 seconds
    [0, 1]);      % initial conditions

  % Determine peak value
  peakVals(idx) = max(Y(:,1));
end

t1 = toc;
```

Mechanics of `parfor` Loops



Pool of MATLAB Workers

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`**)
  - cannot contain **`break`** or **`return`** statements
  - cannot contain another **`parfor`** loop

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
broadcast	a variable defined before the loop whose value is used inside the loop, but never assigned in 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

parfor Examples

- this example cannot be parallized in parfor

```
j=zeros(100);     %pre-allocate vector
j(1)=5;
for i=2:100;
    j(i)=j(i-1)+5;
end;
```

  – order of iterations is important

parfor Examples

- functions with multiple output may confuse Matlab

```
for i=1:10
    [x{i}(:,1), x{i}(:,2)]=functionName(z,w);
end;
```

  - use this instead

```
for i=1:10
      [x1, x2]=functionName(z,w);
       x{i}=[x1 x2];
end;
```

parfor Considerations

- parfor often only involves minimal code changes
- if a for loop cannot be converted to parfor, consider wrapping a subset of loop body in a function
  - e.g. load works not in parfor, however it does work in function that is called inside a parfor loop
- more information http://blogs.mathworks.com/loren/2009/10/02/using-parfor-loops-getting-up-and-running/
- there is a Code-Analyzer to diagnose parfor issues

SPMD

- Matlab also knows a parallel environment SPMD
  - each worker has a separate workspace with variable having the same name (as in MPI)
  - client can modify data on any worker
  - workers can communicate by messages
  - useful for handling large data sets

- syntax

```
spmd
        statements;
end
```

SPMD

```
                      Client        Worker 1    Worker 2
                      a  b  e  |  c  d  f  |  c  d  f
                      --------------------------------
a = 3;                3  -  -  |  -  -  -  |  -  -  -
b = 4;                3  4  -  |  -  -  -  |  -  -  -
spmd                           |           |
   c = labindex();    3  4  -  |  1  -  -  |  2  -  -
   d = c + a;         3  4  -  |  1  4  -  |  2  5  -
end                            |           |
e = a + d{1};         3  4  7  |  1  4  -  |  2  5  -
c{2} = 5;             3  4  7  |  1  4  -  |  5  6  -
spmd                           |           |
   f = c * b;         3  4  7  |  1  4  4  |  5  6 20
end
```

SPMD

- when a SPMD block ends the workspace is saved, the worker is paused
- data is preserved from one block to the next
- does not apply to SPMD block in a function after the function is completed (as regular variables local to a function)

SPMD Example

```
x = imread ('balloons.tif');

y = imnoise ( x, 'salt & pepper', 0.30 );

yd = distributed ( y );

spmd
   yl = getLocalPart ( yd );
   yl = medfilt2 ( yl, [ 3, 3 ] );
end

z(1:480,1:640,1) = yl{1};
z(1:480,1:640,2) = yl{2};
z(1:480,1:640,3) = yl{3};
```

- read image
- add noise to image
- distribute data
- parallel working on image data (filter)
- on master process put together filtered image

SPMD Example

- increase contrast of an image

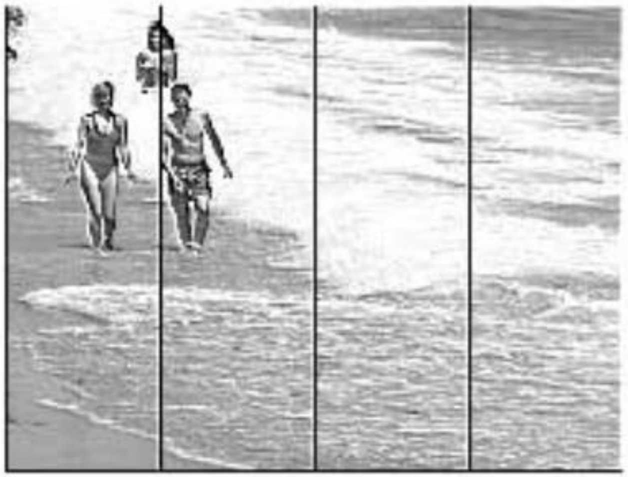
```
%
% Read an image
%
  x = imageread( 'surfsup.tif' );
%
% Since the image is black and white, it will be distributed by columns
%
  xd = distributed(x);
%
%  Each worker enhances the contrast on its portion of the picture
%
  spmd
      xl = getLocalPart(xd);
      xl = nlfilter( xl, [3, 3], @adjustContrast);
      xl = uint8(xl);
  end
%
% Concatenate the submatrices to assemble the whole image
%
  xf_spmd = [ xl{:} ];
```

SPMD Example

- algorithm produces artifacts when parallelized on multiple workers
- problem is that increasing contrast requires information from neighbouring pixel
- distributing the data adds additional boundaries

Filtered on Client

Filtered on 4 SPMD Workers



labSendReceives

- solution is communication between workers
  - each worker has to sent one boundary left and one right
  - each worker has to receive one boundary from left and one from right
  - extra columns are added before filter is applied, and need to be removed again afterwards

labSendReceive

```
column = labSendReceive ( previous, next, xl(:,1) );

if ( labindex() < numlabs() )
  xl = [ xl, column ];
end

column = labSendReceive ( next, previous, xl(:,end) );

if ( 1 < labindex() )
  xl = [ column, xl ];
end
```