CARL VON OSSIETZKY
universität OLDENBURG

Introduction to
High-Performance Computing

Session 06

Matlab Distributed Compute Server

(MDCS)
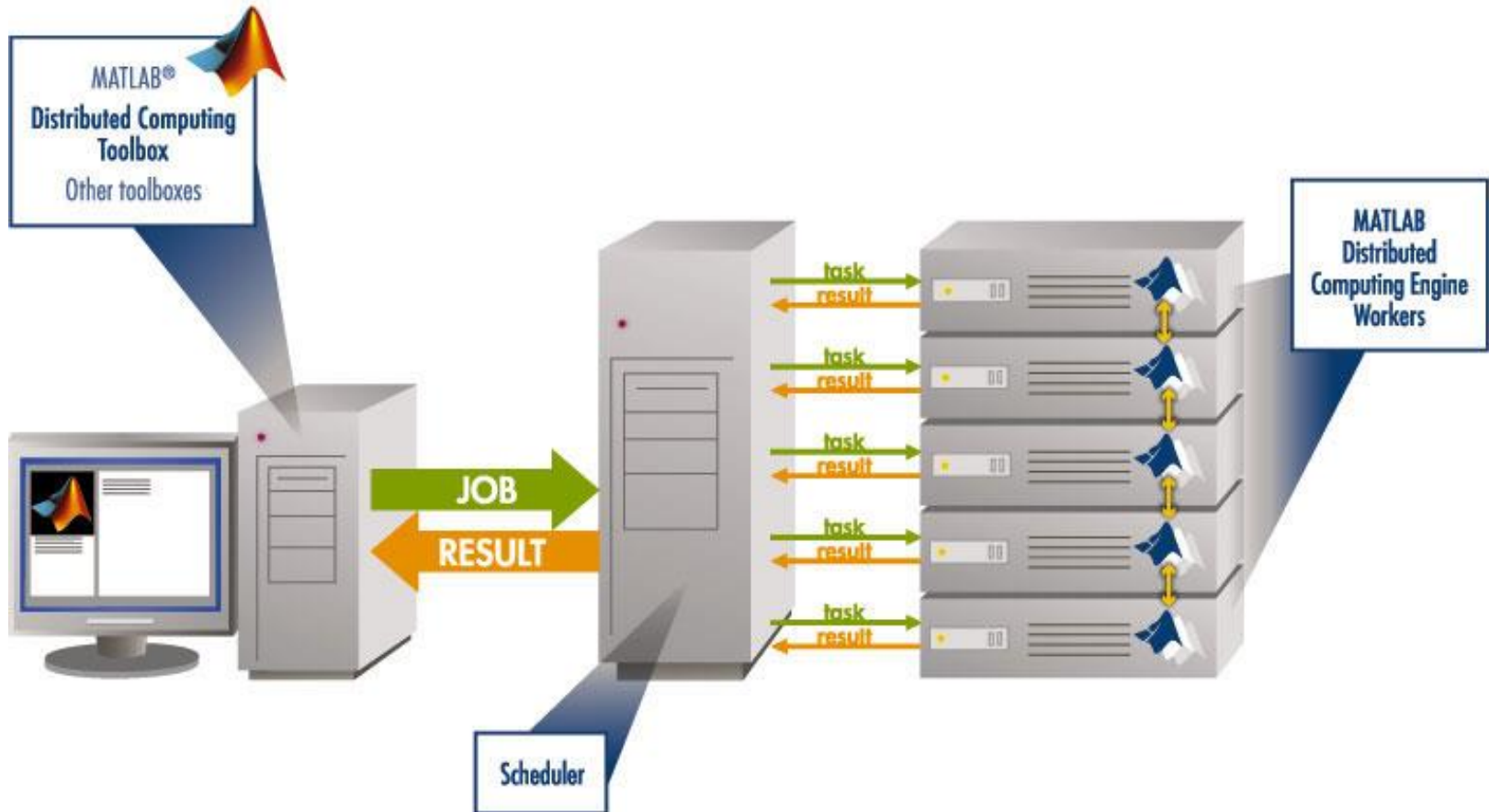
Introduction to MDCS

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
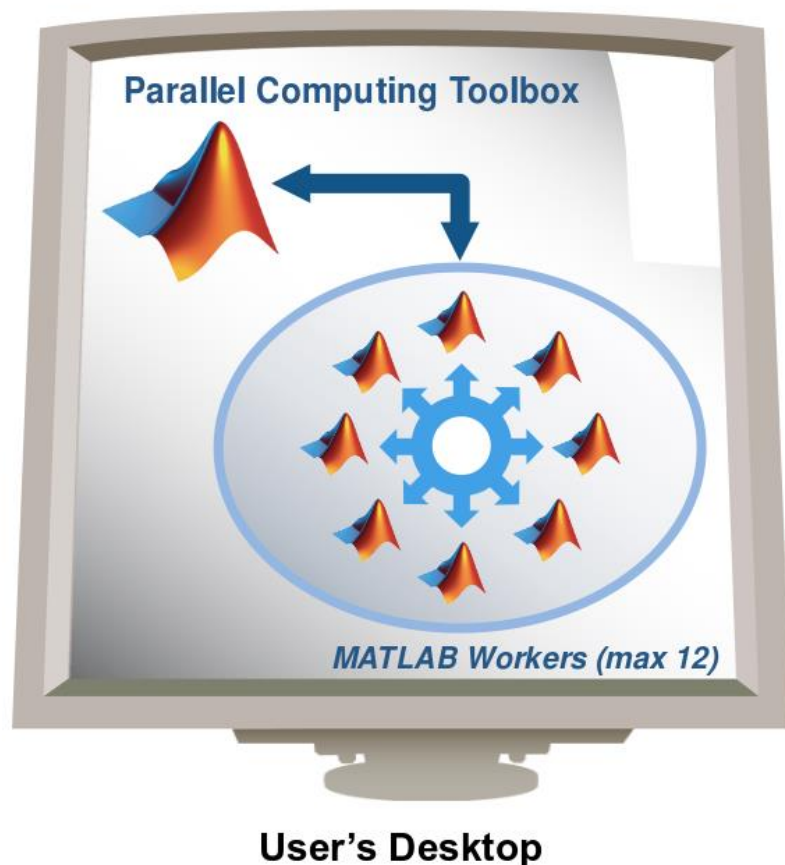- your machine may become unusable while your Matlab job is running

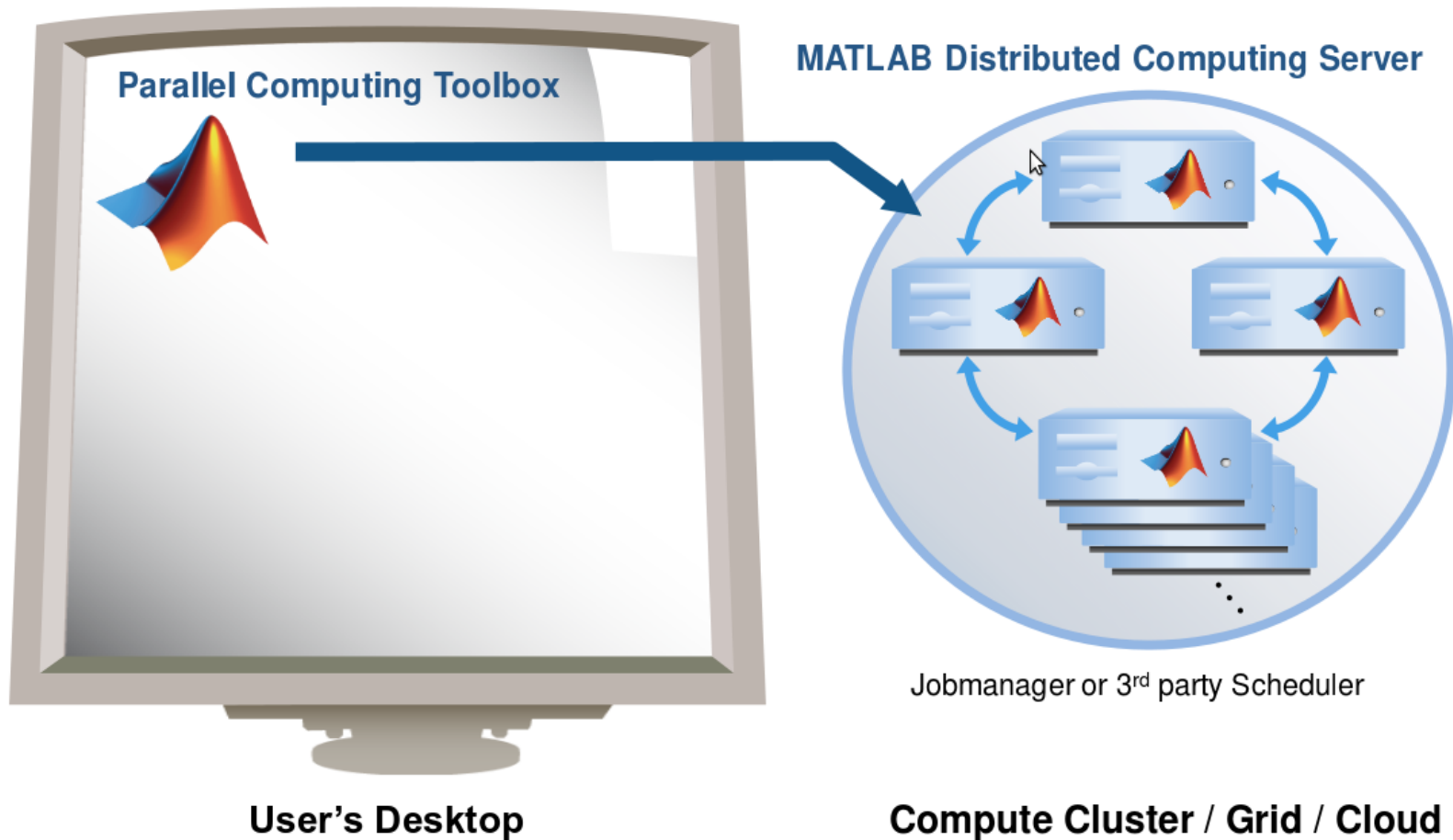# What is MDCS

Parallel Computing with Matlab
(taken from MathWorks marketing)

Parallel Computing Toolbox

MATLAB Workers (max 12)

User's Desktop

- 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

Parallel Computing with Matlab
(taken from MathWorks marketing)

Parallel Computing Toolbox

MATLAB Distributed Computing Server

Jobmanager or 3rd party Scheduler

User's Desktop

Compute Cluster / Grid / Cloud

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
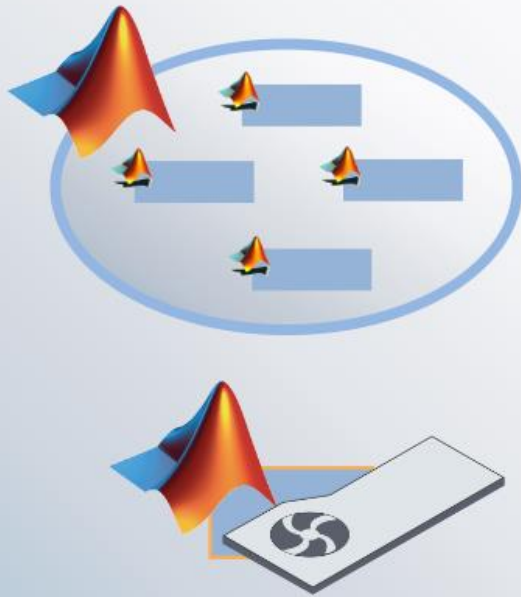 - task-parallel
  - data-parallel

Why to use MDCS on the Cluster?

- ## MDCS on the HPC cluster includes 272 worker licenses
    - these are in addition to the normal Matlab licenses (which used to be limited to 200 for the whole university)
    - you can use also any of the toolboxes (were limited to 50)
    - allows the control over used licenses and prevents failed jobs
    - for fair sharing not more than 36 MDCS licenses should be used per job and at most two jobs per user (hard limit)

- ## ease of use
    - no need to learn about job scripts
    - work within known Matlab environment
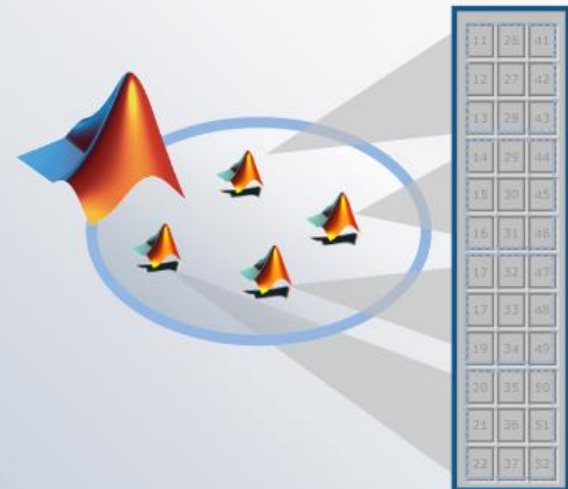
Parallel Computing with Matlab

Parallel Computing with Matlab
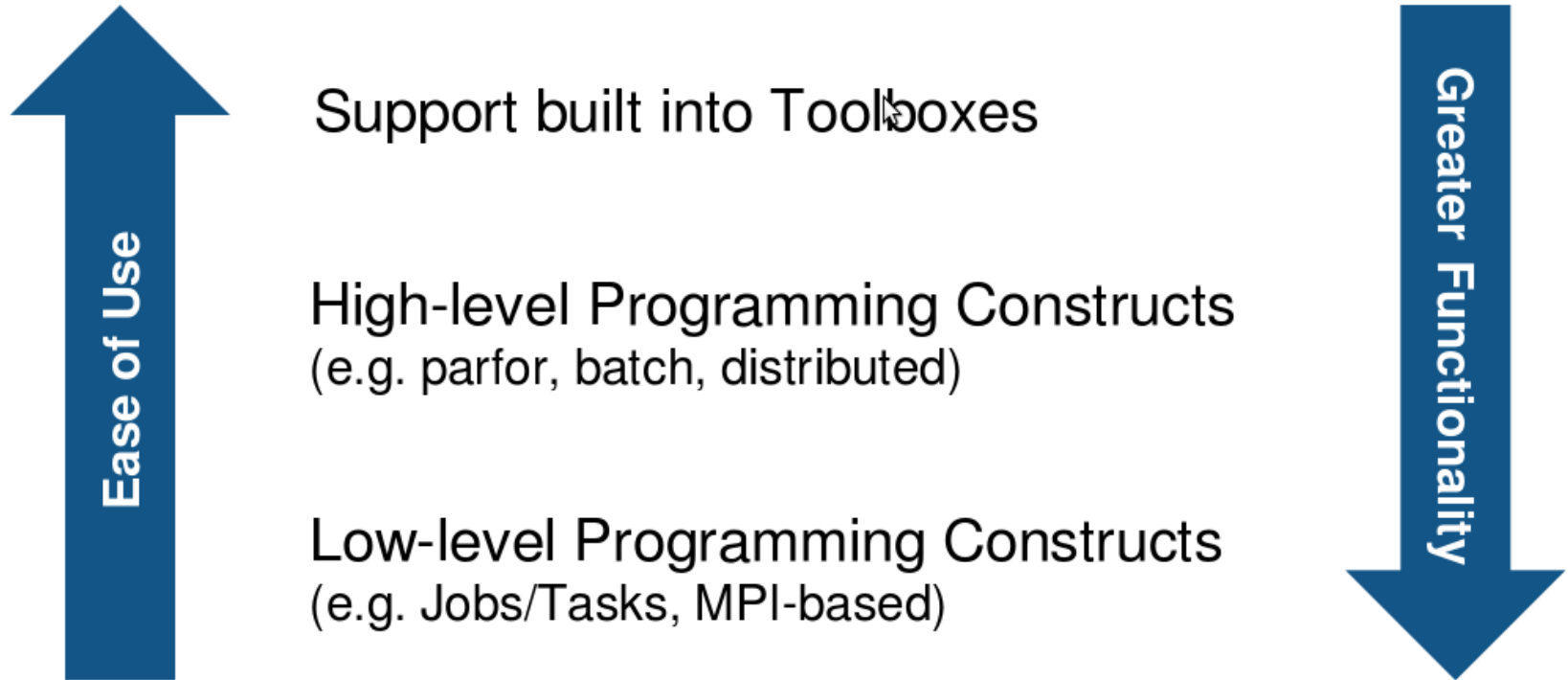
Three levels of Integration:

Support built into Toolboxes

High-level Programming Constructs
(e.g. parfor, batch, distributed)

Low-level Programming Constructs
(e.g. Jobs/Tasks, MPI-based)

Ease of Use ↑

Greater Functionality ↓

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

Configuration of MDCS

Using MDCS on CARL/EDDY

- 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, version must match to version on cluster (e.g. R2016b)
  - your local machine must be able to login to CARL/EDDY 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 be 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

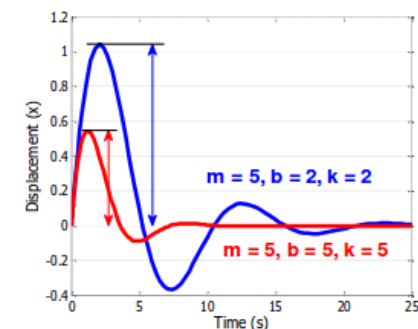see https://wiki.hpcuser.uni-oldenburg.de/index.php?title=Configuration_MDCS_2016

Using MDCS on CARL/EDDY

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

$$m\ddot{x} + b\ \dot{x} + k\ x = 0$$
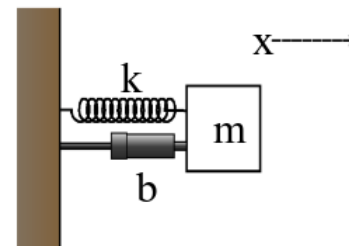
- simulate with different values for *b* and *k*
- record peak value for each run

2nd-order ODE for example

odesystem.m

```matlab
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

```matlab
%% 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;
```

Scientific Computing
V. School of Mathematics and Science

CARL
VON
OSSIETZKY
universität | OLDENBURG

Parameter Sweep: parallel Matlab code

paramSweep_batch.m

```matlab
%% 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;
```

Using MDCS on CARL/EDDY

- submitting jobs to the cluster

```
sched = parcluster('CARL');
job = batch(sched, 'paramSweep_batch', 'Pool', 7, ...
            'AttachedFiles', {'odesystem.m'});
```

- – first command creates a handle for the cluster using the available configuration
- – second command creates a job and sends it to the cluster
  - Matlab script is executed on the cluster
  - requests a pool of workers (number of processes is +1 for master)
  - uses default resources unless modified
  - files can be attached but Matlab also automatically attaches needed files (if it can find them and if not disabled)

Using MDCS on CARL/EDDY

- changing resource allocation

```
set(sched, 'CommunicatingSubmitFcn',
    cat(2, sched.CommunicatingSubmitFcn,
        {'runtime','72:0:0','memory','4G'}));
```

  – changes maximum runtime and memory per worker

- path-dependency as alternative to attaching files
  – use addpath within script (.m-files)
  – use AdditionalPath property of scheduler object
  – use absolute path names
  – copy files to the cluster before submitting job

Using MDCS on CARL/EDDY

- recovering jobs
  - it is possible to terminate the local Matlab session while jobs are running (or waiting on the cluster)
 - to reconnect

```
sched    = parcluster(,CARL');
sched.Jobs   % to list available jobs
job      = sched.Jobs(1)   % to get job information
jobData = load(job);
```

Monitoring Jobs and Error Tracking

- Matlab Job Monitor for basic information
- use squeue and sacct for additional information from SLURM
- job handle can be used to get information about errors
- Matlab diary for additional log output
- files in the job directory

MDCS with parfor

Mechanics of `parfor` Loops
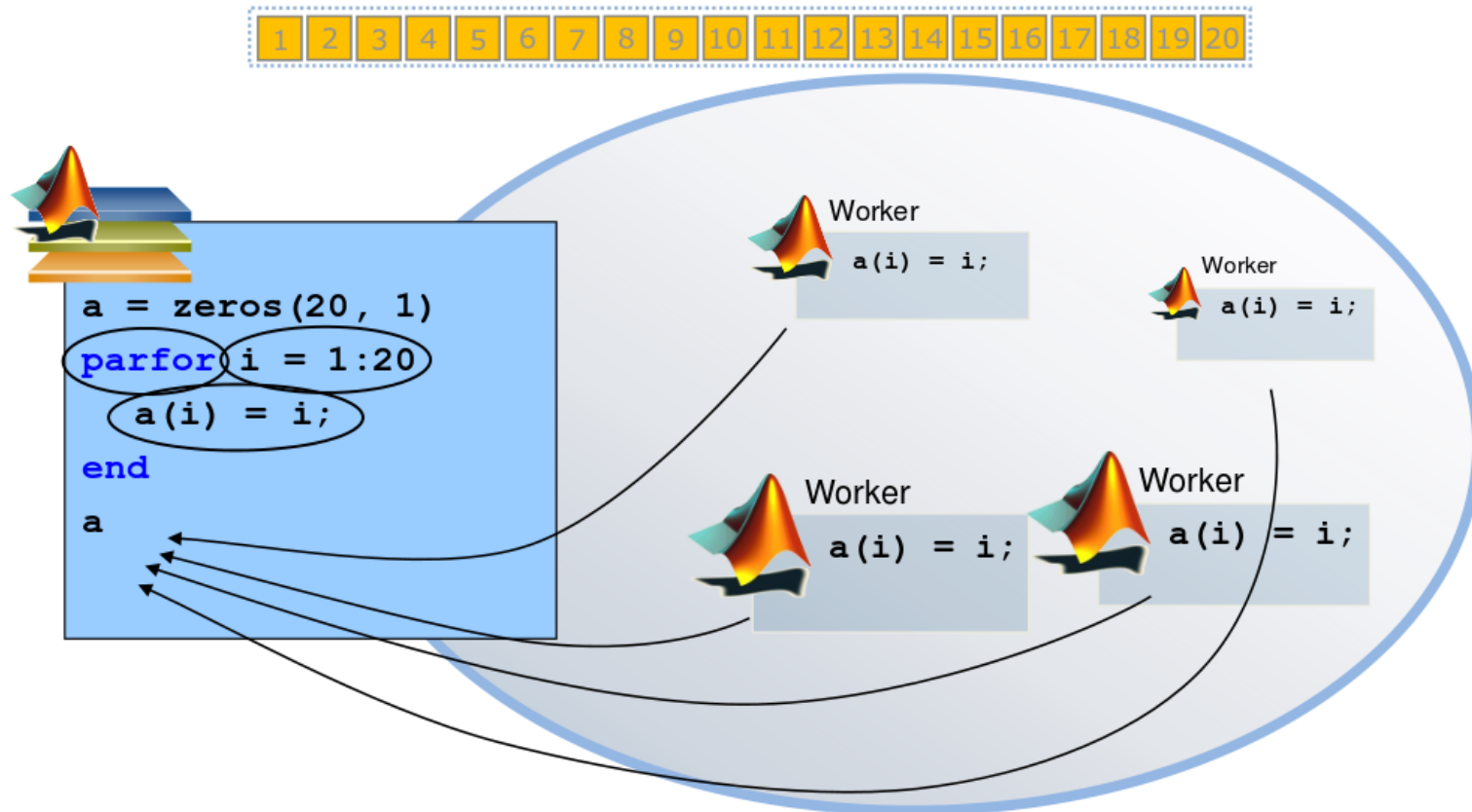
Pool of MATLAB Workers

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

Variable Classification Example

- matrix-vector multiplication

```
N=2048;                    % N is broadcast
b=rand(N,1);               % b is broadcast
A=rand(N,N);               % A is slices input

parfor i=1:N               % i is loop index
  c(i)=A(i,:)*b(:);        % c is sliced output
end
```

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

MDCS with spmd
(single program multiple data)

```
              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

Distributed Data

- Matlab provides different functions to manage distributed data
  - with distributed(X) you can distribute data among workers
  - with distributed.METHOD you can create data distributed among workers
  - workers can create codistributed data structures which become distributed data outside of the SPMD block
  - a datastore can be distributed to read manage large data files with multiple workers

  - see 'help distributed' for more information

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
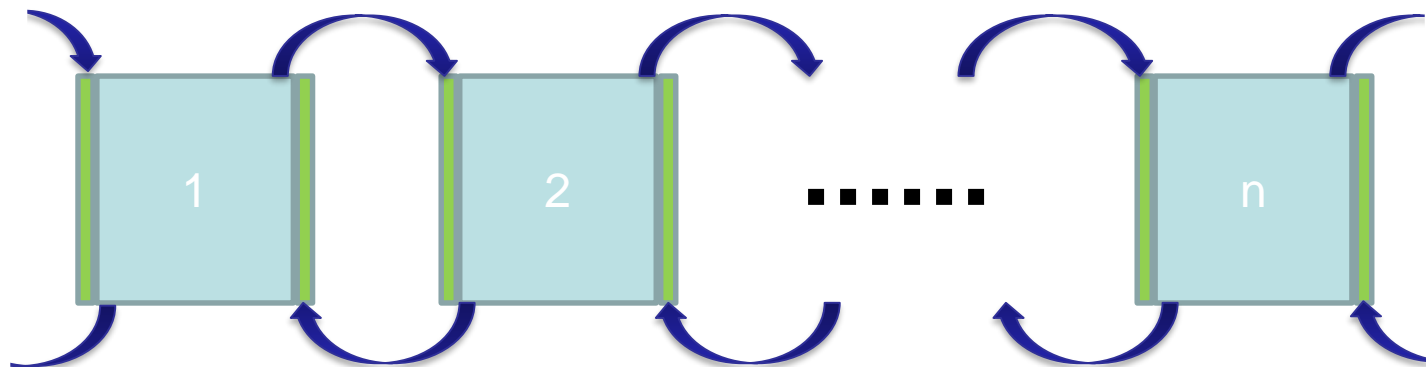
Filtered on Client | Filtered on 4 SPMD Workers

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
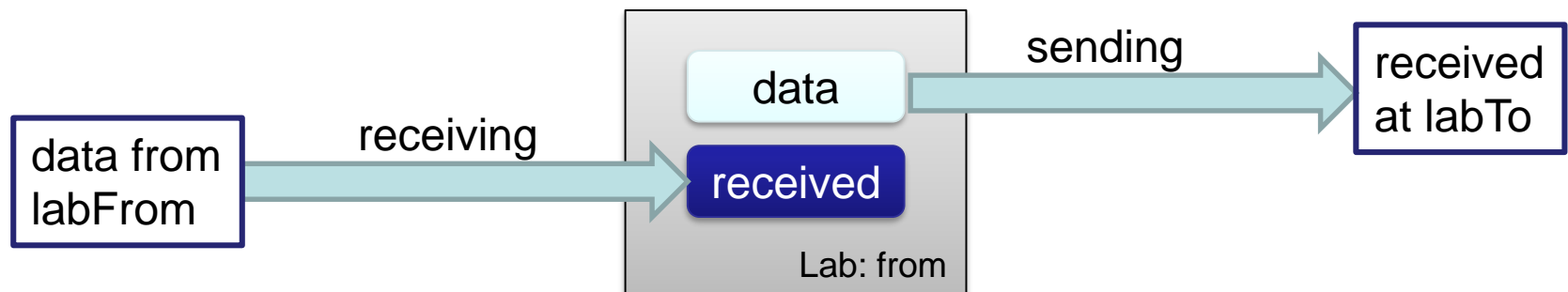 - extra columns are added before filter is applied, and need to be removed again afterwards

labSendReceives

- the function labSendReceive simultaneously sends and receives data

  received = labSendReceive(labTo, labFrom, data)

  – sends data to labTo

  – receives data from labFrom and stores it in received

labSendReceive

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

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


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

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

Heat Example in Matlab

```matlab
% 2d-heat example in Matlab
% initial setup
NXPROB = 20;        % number of grid rows
NYPROB = 20;        % number of grid columns
STEPS  = 100;       % number of iterations
TIME   = 0;         % initial and current time


uvals  = zeros(2, NXPROB, NYPROB);  % allocate grid
uvals  = inidat(uvals);             % initialize grid


plotdat(uvals, 1, TIME);            % make plot


it = 1;
for TIME=1:STEPS                    % time iteration
    uvals = updateu(uvals, it);    % update thermal energy
    it    = 3 - it;
end


plotdat(uvals, 1, TIME);            % make plot
```