Parallel Computing with MATLAB and Simulink

Jos Martin

jos.martin@mathworks.com

Senior Engineering Manager – Parallel Computing

Introduction

- Why Parallel Computing
  - Need faster insight to bring competitive products to market quickly
  - Computing infrastructure is broadly available (Multicore Desktops, GPUs, Clusters)

- With MathWorks Parallel Computing Tools
  - Leverage computational power of available hardware
  - Accelerate workflows with minimal to no code changes to your original code
  - Seamlessly scale from your desktop to clusters or on the cloud
  - Save engineering and research time and focus on insight

Agenda

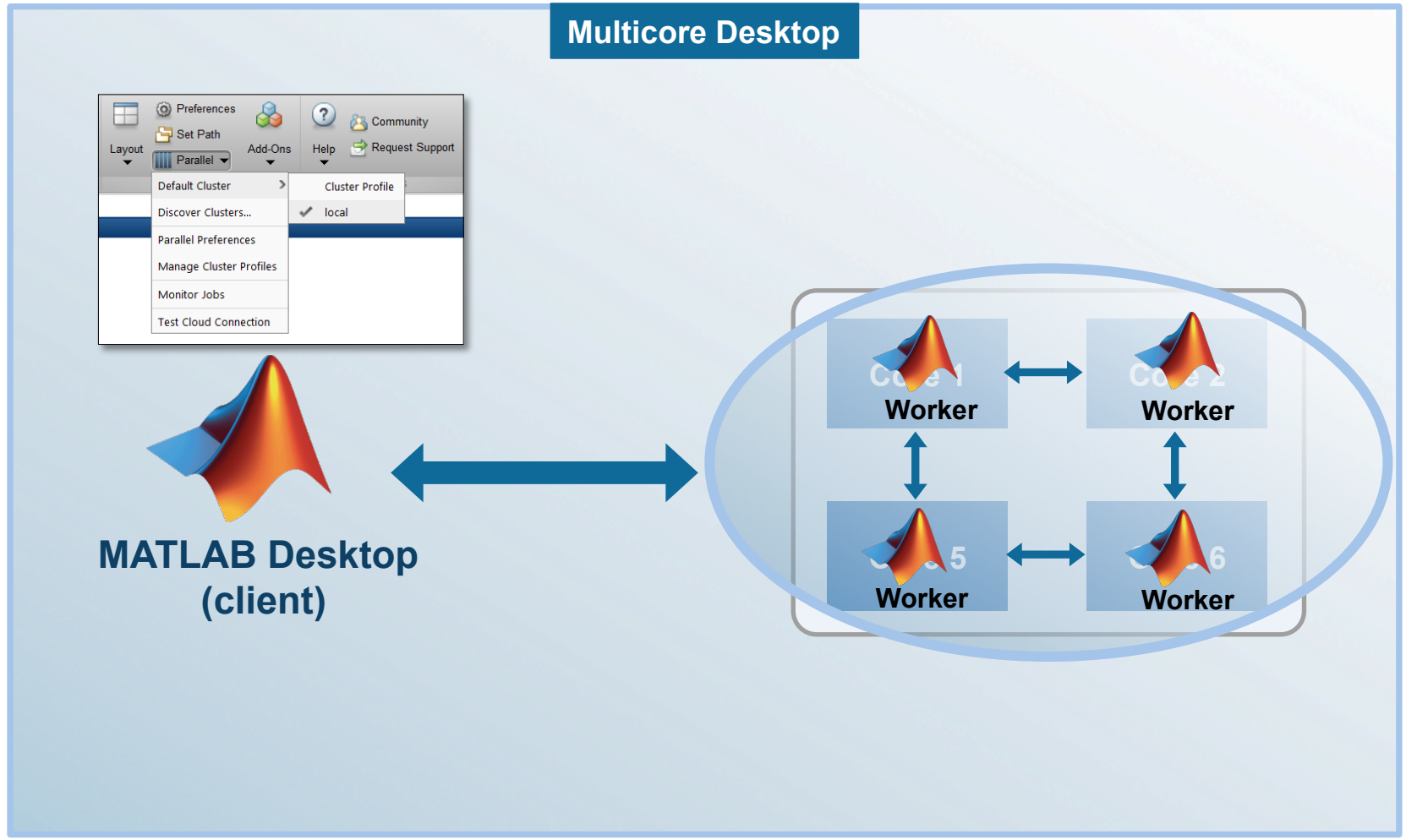
- Parallel Computing Paradigm
- Task Parallelism
- Data Parallelism
- Summary

Agenda

- Parallel Computing Paradigm
- Task Parallelism
- Data Parallelism
- Summary

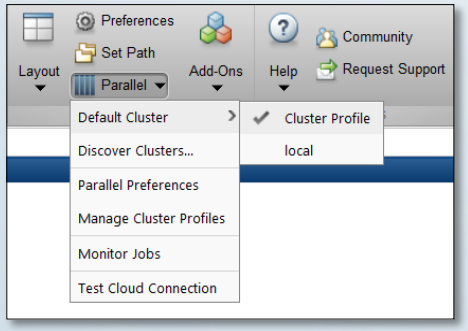
Parallel Computing Paradigm

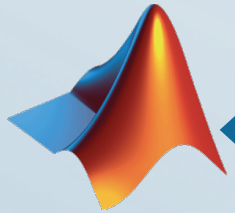
Multicore Desktops



Parallel Computing Paradigm

Cluster Hardware





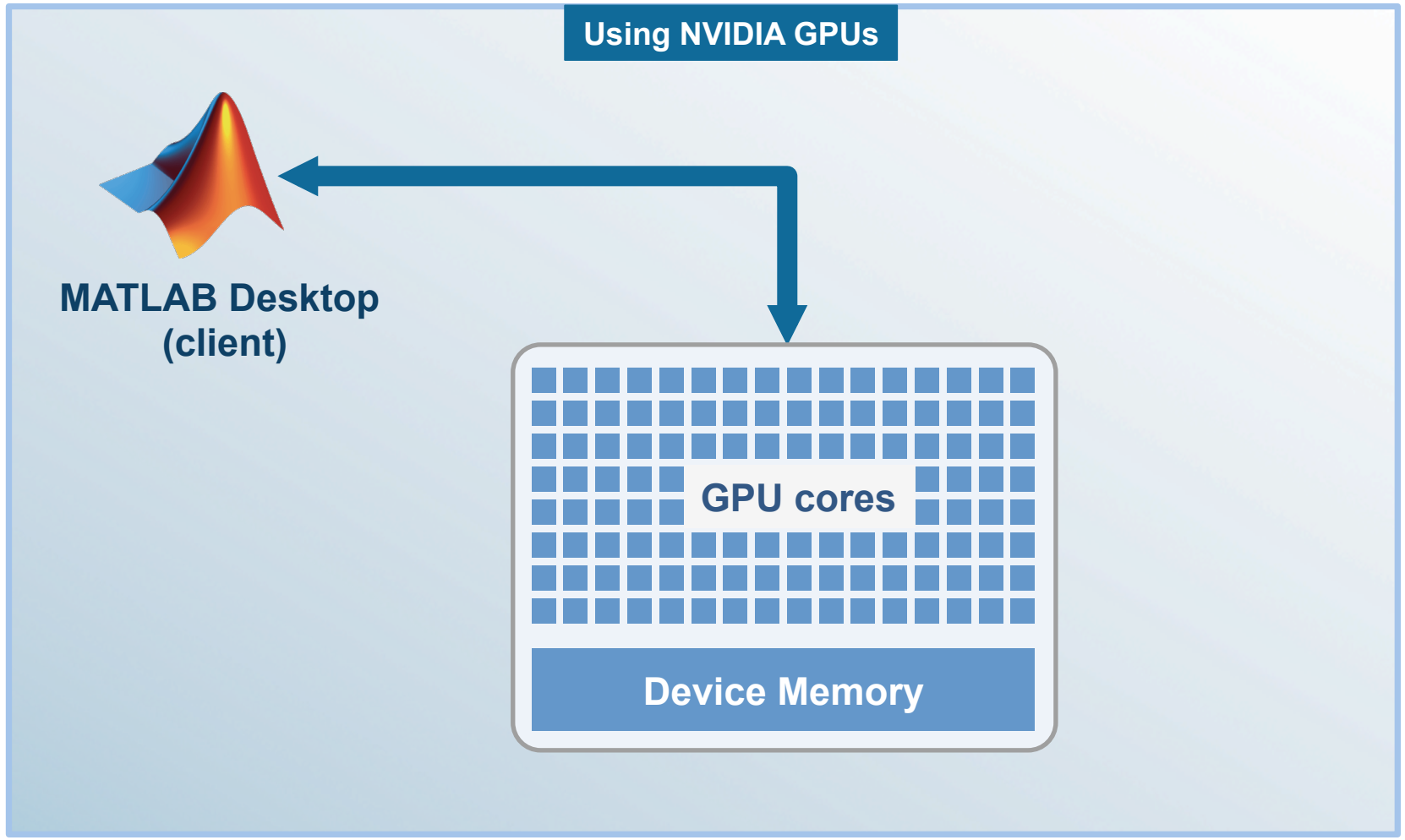
MATLAB Desktop (client)

Cluster of computers

Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker
Worker

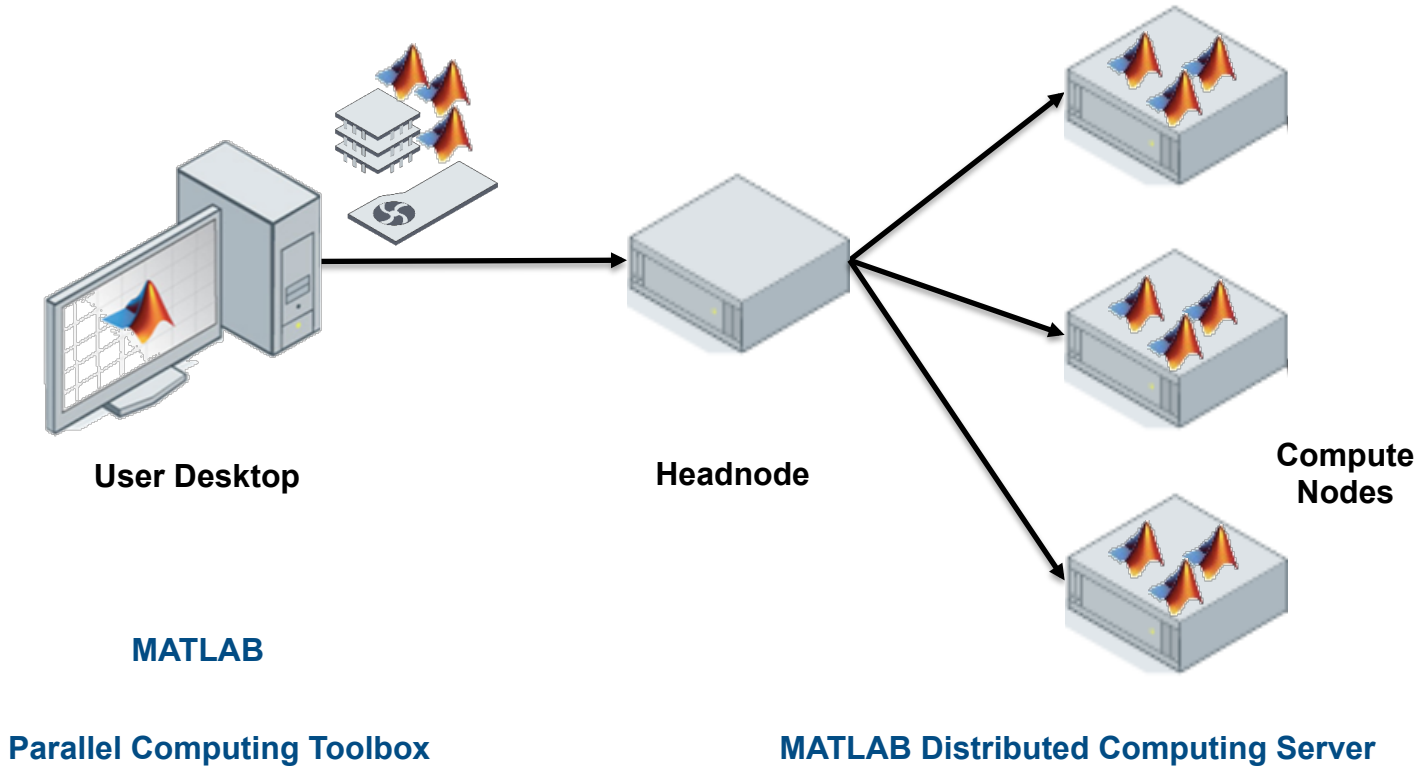
Parallel Computing Paradigm

NVIDIA GPUs



Cluster Computing Paradigm

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



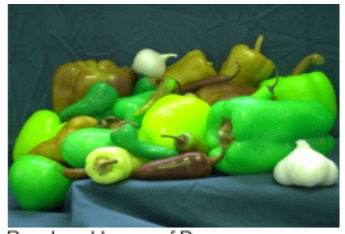
Parallel-enabled Toolboxes (MATLAB® Product Family)

Enable parallel computing support by setting a flag or preference

Image Processing

Batch Image Processor, Block Processing, GPU-enabled functions

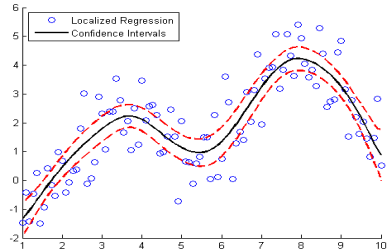
Original Image of Peppers



Recolored Image of Peppers

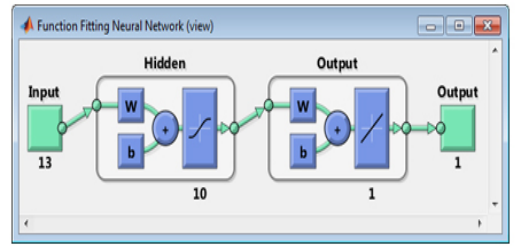
Statistics and Machine Learning

Resampling Methods, k-Means clustering, GPU-enabled functions



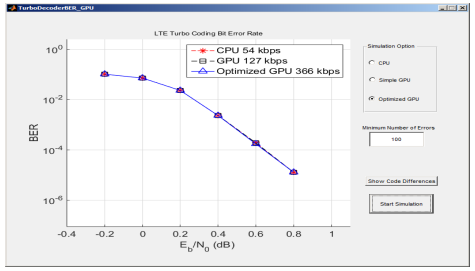
Neural Networks

Deep Learning, Neural Network training and simulation



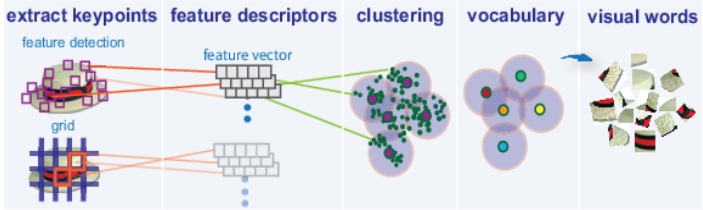
Signal Processing and Communications

GPU-enabled FFT filtering, cross correlation, BER



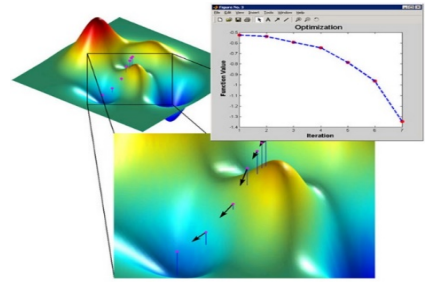
Computer Vision

Parallel-enabled functions in bag-of-words workflow



Optimization

Parallel estimation of gradients



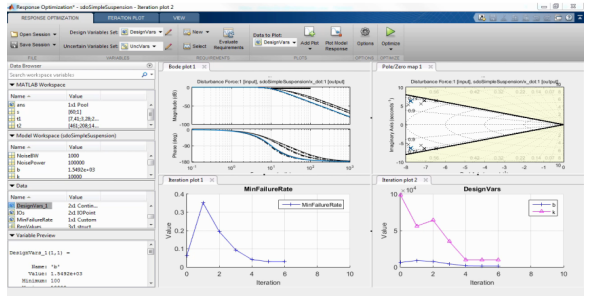
[Other Parallel-enabled Toolboxes]

Parallel-enabled Toolboxes (Simulink® Product Family)

Enable parallel computing support by setting a flag or preference

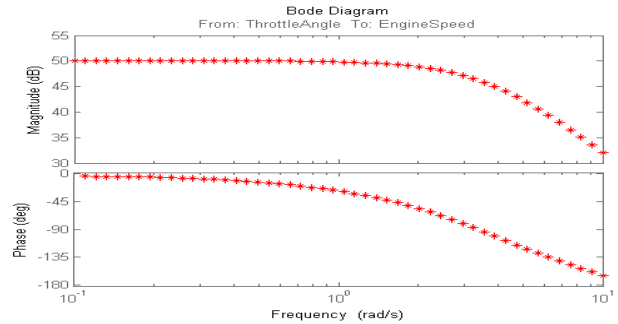
Simulink Design Optimization

Response optimization, sensitivity analysis, parameter estimation



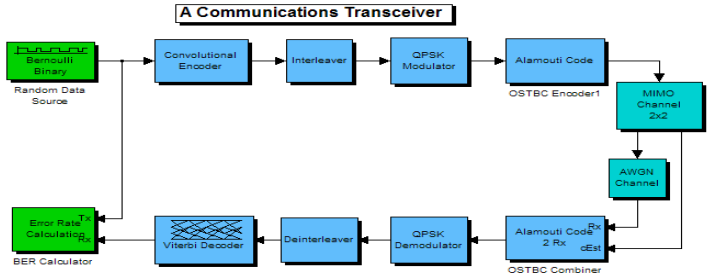
Simulink Control Design

Frequency response estimation



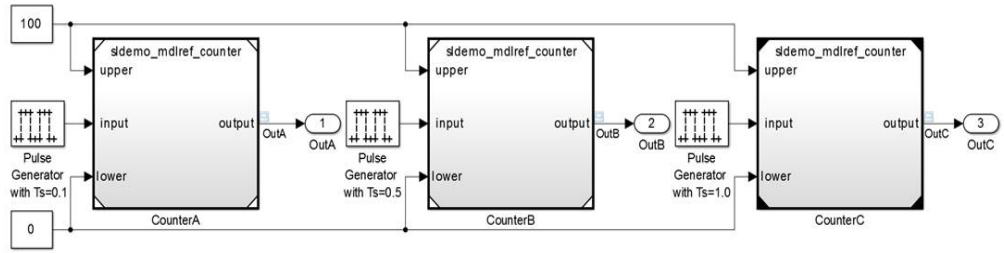
Communication Systems Toolbox

GPU-based System objects for Simulation Acceleration



Simulink/Embedded Coder

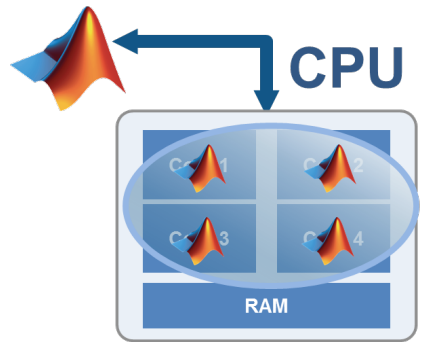
Generating and building code



Other Parallel-enabled Toolboxes

Agenda

- Parallel Computing Paradigm
- Task Parallelism
- Data Parallelism
- Summary



`parfor`

Definition

Code in a `parfor` *loop is guaranteed by the programmer to be execution order independent*

Why is that important?

We can execute the iterates of the loop in any order, potentially at the same time on many different workers.

`parfor` – how it works

- Static analysis to deduce variable classification
  - Works out what data will need to be passed to which iterates
- A loop from 1:N has N *iterates* which we partition into a number of *intervals*
  - Each *interval* will likely have a different number of *iterates*
- Start allocating the *intervals* to execute on the workers
- Stitch the results back together
  - Using functions from the static analysis

Variable Classification

```
reduce = 0; bcast = …; in = …;
parfor i = 1:N
    temp = foo1(bcast, i);
    out(i) = foo2(in(i), temp);
    reduce = reduce + foo3(temp);
end
```

Loop variable

```
reduce = 0; bcast = …; in = …;
parfor i = 1:N
    temp = foo1(bcast, i);
    out(i) = foo2(in(i), temp);
    reduce = reduce + foo3(temp);
end
```

Making extra parallelism

- No one loop appears to have enough iterations to go parallel effectively
  - Imagine if Na, Nb, and Nc are all reasonably small

```
for ii = 1:Na
    for jj = 1:Nb
        for kk = Nc
        end
    end
end

Na * Nb * Nc == quiteBigNumber
```

mergeLoopsDemo

Making extra parallelism

```
[N, fun] = mergeLoopRanges([Na Nb Nc]);
parfor xx = 1:N
    [ii,jj,kk] = fun(xx);
    doOriginalLoopCode
end
```

Sliced Variable

```
reduce = 0; bcast = …; in = …;
parfor i = 1:N
    temp = foo1(bcast, i);
    out(i) = foo2(in(i), temp);
    reduce = reduce + foo3(temp);
end
```

Broadcast variable

```
reduce = 0; bcast = …; in = …;
parfor i = 1:N
    temp = foo1(bcast, i);
    out(i) = foo2(in(i), temp);
    reduce = reduce + foo3(temp);
end
```

Reusing data

```
D = makeSomeBigData;
for ii = 1:N
    parfor jj = 1:M
        a(jj) = func(D, jj);
    end
end
```

Reusing data

```
D = parallel.pool.Constant( @makeSomeBigData );
for ii = 1:N
    parfor jj = 1:M
        a(jj) = func(D.value, jj);
    end
end

% Alternatively you can send data once and re-use it
oD = parallel.pool.Constant( someLargeData )
```

Common parallel program

```
set stuff going
while not all finished {
    for next available result do something;
}
```

parfeval

- Allows asynchronous programming

```
f = parfeval(@func, numOut, in1, in2, …)
```

- The return **`f`** is a future which allows you to
  - Wait for the completion of calling **`func(in1, in2, …)`**
  - Get the result of that call
  - … do other useful parallel programming tasks …

Fetch Next

- Fetch next available unread result from an array of futures.

```
[idx, out1, ...] = fetchNext(arrayOfFutures)
```

- **`idx`** is the index of the future from which the result is fetched
- Once a particular future has returned a result via **`fetchNext`** it will *never* do so again
  - That particular result is considered read, and will not be re-read

Common parallel program (MATLAB)

```
% Set stuff going
for ii = N:-1:1
    fs(ii) = parfeval(@stuff, 1);
end
% While not all finished
for ii = 1:N
    % For next available result
    [whichOne, result] = fetchNext(fs);
    doSomething(whichOne, result);
end
```

parfevalWaitbarDemo

`parfevalOnAll`

- Frequently you want setup and teardown operations
  - which execute once on each worker in the pool, before and after the actual work
- Execution order guarantee:

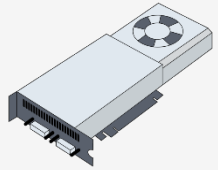
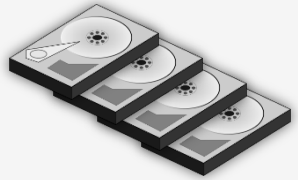
It is guaranteed that relative order of `parfeval` and `parfevalOnAll` as executed on the client will be preserved on all the workers.

Agenda

- Parallel Computing Paradigm
- Task Parallelism
- Data Parallelism
- Summary

Remote arrays in MATLAB

MATLAB provides array types for data that is not in “normal” memory

`distributed` array (since R2006b)		Data lives in the combined memory of a cluster of computers
`gpuArray` (since R2010b)		Data lives in the memory of the GPU card
`tall` array (since R2016b)		Data lives on disk, maybe spread across many disks (distributed file-system)

Remote arrays in MATLAB

Rule: take the calculation to where the data is

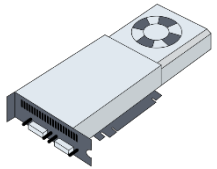
Normal array – calculation happens in main memory:

```
x = rand(...)

x_norm = (x - mean(x)) ./ std(x)
```

Remote arrays in MATLAB

Rule: take the calculation to where the data is

gpuArray – all calculation happens on the GPU:

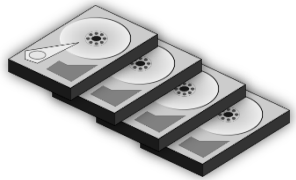


```
x = gpuArray(...)

x_norm = (x - mean(x)) ./ std(x)
```

distributed – calculation is spread across the cluster:

```
x = distributed(...)

x_norm = (x - mean(x)) ./ std(x)
```

tall – calculation is performed by stepping through files:



```
x = tall(...)

x_norm = (x - mean(x)) ./ std(x)
```

How big is big?

What does “Big Data” even mean?

“Any collection of data sets so large and complex that it becomes difficult to process using … traditional data processing applications.”

(Wikipedia)

“Any collection of data sets so large that it becomes difficult to process using traditional MATLAB functions, which assume all of the data is in memory.”

(MATLAB)

How big is big?

In 1085 William 1st commissioned a survey of England

- ~2 million words and figures collected over two years
- too big to handle in one piece
- collected and summarized in regional pieces
- used to generate revenue (tax), but most of the data then sat unused

The Large Hadron Collider reached peak performance on 29 June 2016

- 2076 bunches of 120 billion protons currently circulating in each direction
- ~1.6×10^{14} collisions per week, >30 petabytes of data per year
- too big to even store in one place
- used to explore interesting science, but taking researchers a long time to get through

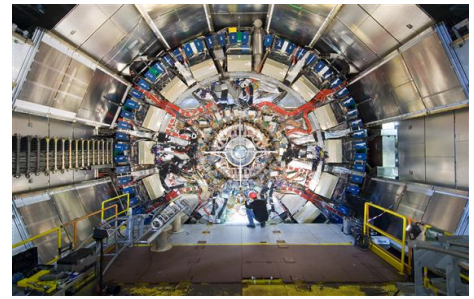
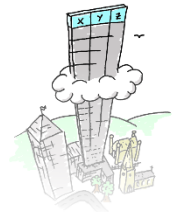
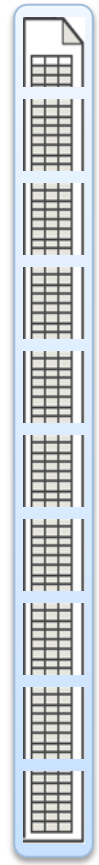


Image courtesy of CERN.

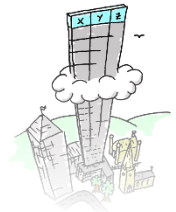
Tall arrays (new R2016b)



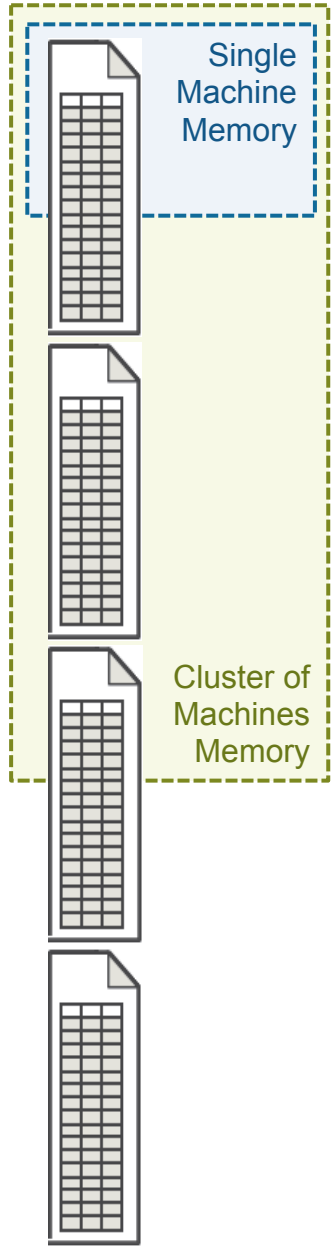
- MATLAB data-type for data that doesn't fit into memory
- Ideal for lots of observations, few variables (hence "tall")
- Looks like a normal MATLAB array
  - Supports numeric types, tables, datetimes, categoricals, strings, etc…
  - Basic maths, stats, indexing, etc.
  - **Statistics and Machine Learning Toolbox** support (clustering, classification, etc.)



Tall arrays (new R2016b)



- Data is in one or more files
- Typically tabular data
- Files stacked vertically
- Data doesn't fit into memory
  (even cluster memory)



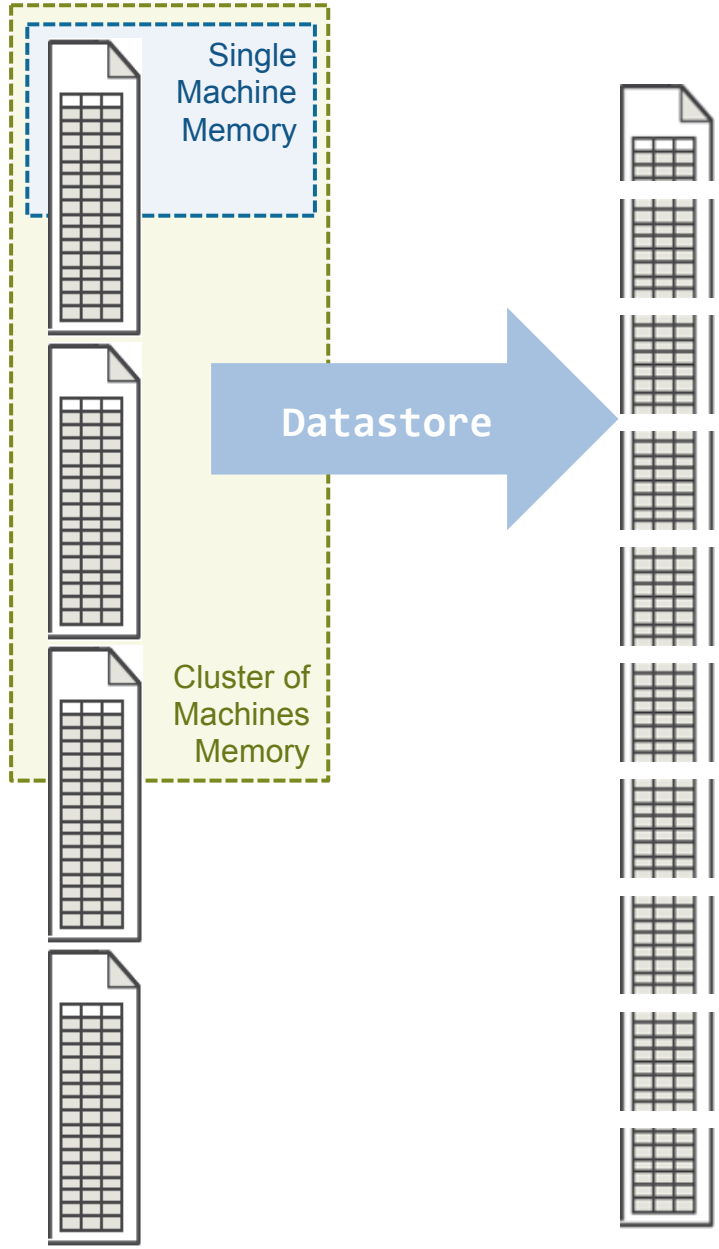
Tall arrays (new R2016b)

- Use datastore to define files

```
ds = datastore('*.csv')
```

- Allows access to small pieces of data that fit in memory.

```
while hasdata(ds)
    piece = read(ds);
    % Process piece
end
```



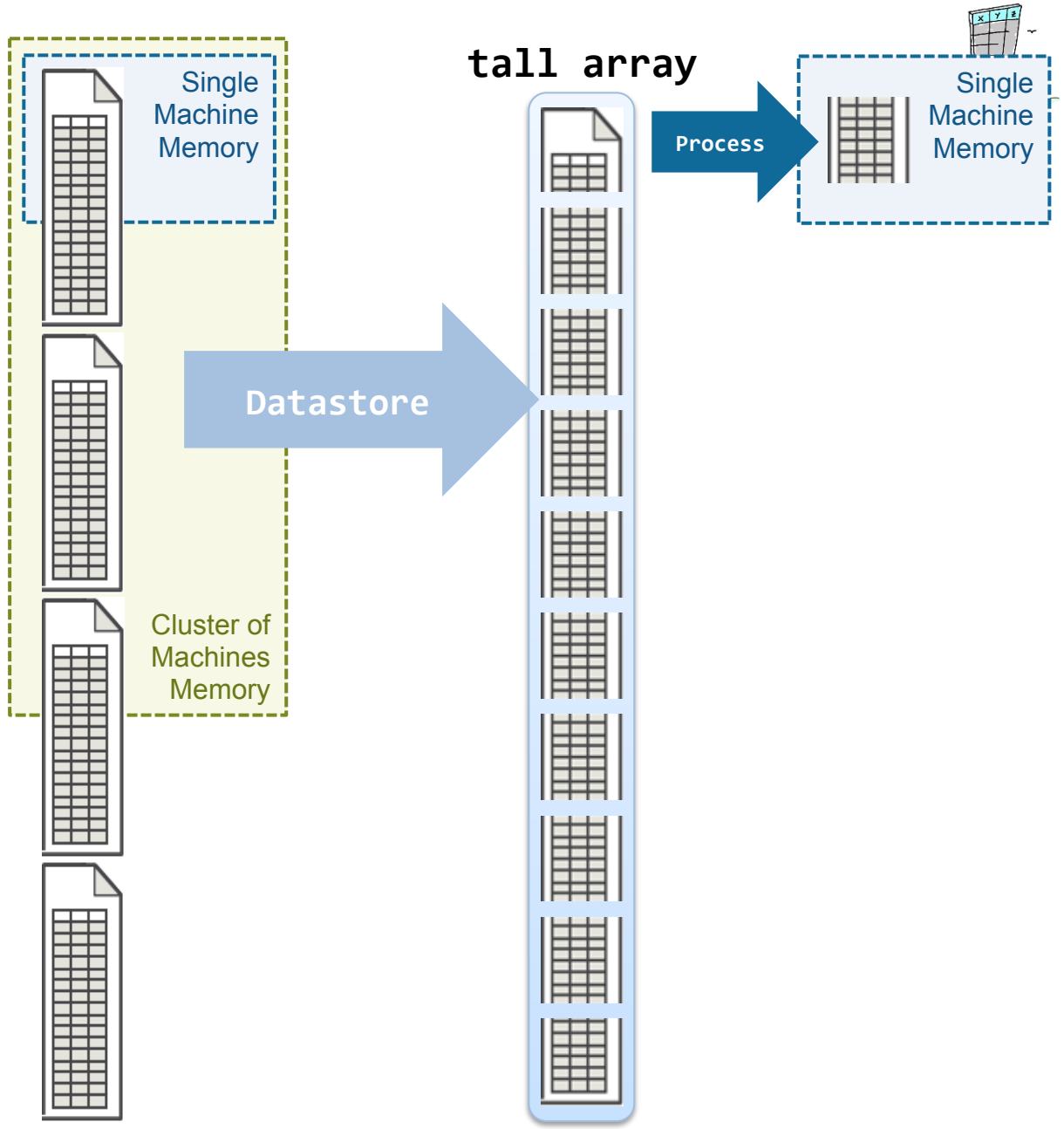
Tall arrays (new R2016b)

- Create tall table from datastore

```
ds = datastore('*.csv')
tt = tall(ds)
```

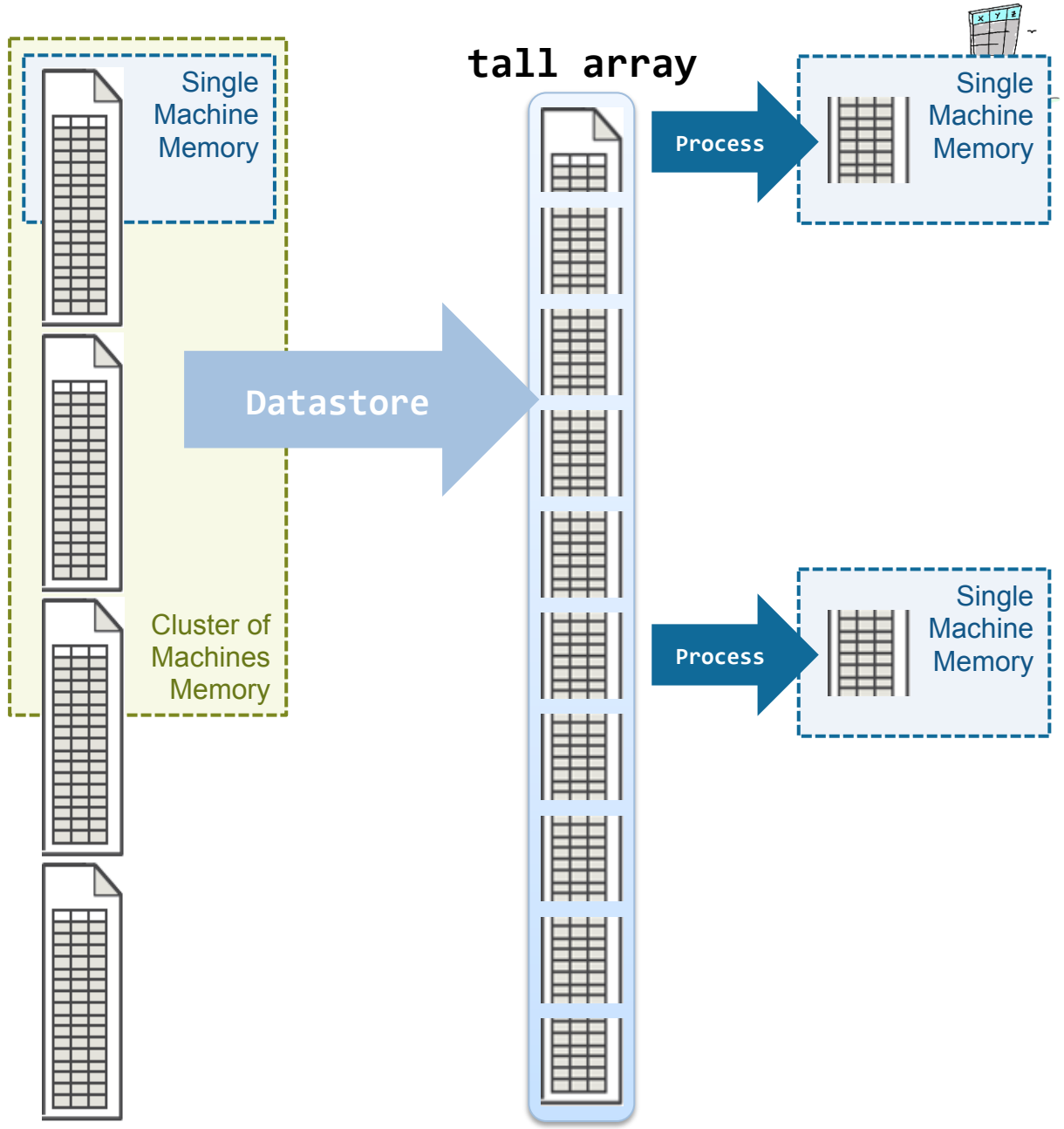
- Operate on whole tall table just like ordinary table

```
summary(tt)

max(tt.EndTime - tt.StartTime)
```



Tall arrays (new R2016b)

- With Parallel Computing Toolbox, process several pieces at once



Tall arrays (new R2016b)

Example

New York taxi fares (150,000,000 rows (~25GB) per year)

```
>> dataLocation = 'hdfs://hadoop01glnxa64:54310/datasets/nyctaxi/';
>> ds = datastore( fullfile(dataLocation, 'yellow_tripdata_2015-*.csv') );
>> tt = tall(ds)
tt =

  M×19 tall table

    VendorID    tpep_pickup_datetime     tpep_dropoff_datetime    passenger_count    trip_distance    pickup_longitude    pi
    ________    ____________________     _____________________    _______________    _____________    ________________    __

    2           '2015-01-15 19:05:39'    '2015-01-15 19:23:42'    1                  1.59             -73.994              4
    1           '2015-01-10 20:33:38'    '2015-01-10 20:53:28'    1                   3.3             -74.002             40
    1           '2015-01-10 20:33:38'    '2015-01-10 20:43:41'    1                   1.8             -73.963             40
    1           '2015-01-10 20:33:39'    '2015-01-10 20:35:31'    1                   0.5             -74.009             40
    1           '2015-01-10 20:33:39'    '2015-01-10 20:52:58'    1                     3             -73.971             40
    1           '2015-01-10 20:33:39'    '2015-01-10 20:53:52'    1                     9             -73.874             40
    1           '2015-01-10 20:33:39'    '2015-01-10 20:58:31'    1                   2.2             -73.983             40
    1           '2015-01-10 20:33:39'    '2015-01-10 20:42:20'    3                   0.8             -74.003             40.
    :           :                        :                        :                   :                :                   :
    :           :                        :                        :                   :                :                   :
```

Input data is tabular – result is a tall table

Tall arrays (new R2016b)

Example

New York taxi fares (150,000,000 rows (~25GB) per year)

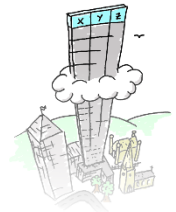
```
>> dataLocat                    oop01glnxa64:54310/datasets/nyctaxi/';
>> ds = data                    ataLocation, 'yellow_tripdata_2015-*.csv') );
>> tt = tall
tt =

  M×19 tall table

    VendorID    tpep_pickup_datetime     tpep_dropoff_datetime    passenger_count    trip_distance    pickup_longitude    pi
    ________    ____________________     _____________________    _______________    _____________    ________________    __

    2           '2015-01-15 19:05:39'    '2015-01-15 19:23:42'    1                  1.59             -73.994              4
    1           '2015-01-10 20:33:38'    '2015-01-10 20:53:28'    1                   3.3             -74.002             40
    1           '2015-01-10 20:33:38'    '2015-01-10 20:43:41'    1                   1.8             -73.963             40
    1           '2015-01-1               01-10 20:35:31'          1                   0.5             -74.009             40
    1           '2015-01-1               01-10 20:52:58'          1                     3             -73.971             40
    1           '2015-01-1               01-10 20:53:52'          1                     9             -73.874             40
    1           '2015-01-1               01-10 20:58:31'          1                   2.2             -73.983             40
    1           '2015-01-10 20:33:39     2015-01-10 20:42:20'     3                   0.8             -74.003             40.
    :           :                        :                        :                   :                :                   :
    :           :                        :                        :                   :                :                   :
```

Number of rows is unknown until all the data has been read

Only the first few rows are displayed

Tall arrays (new R2016b)

Example

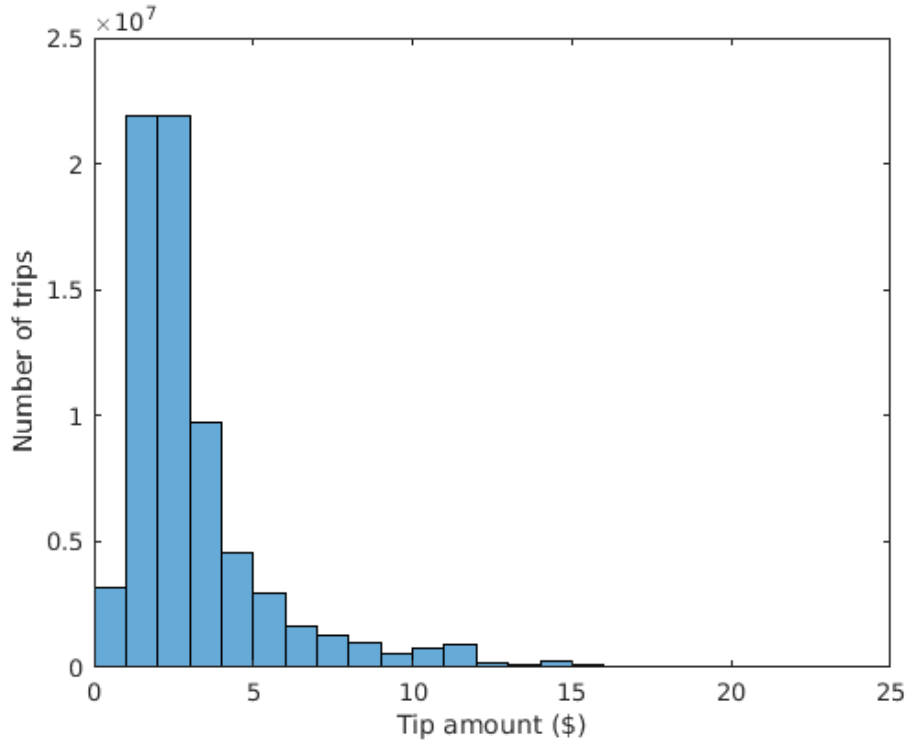


Once created, can process much like an ordinary table

```
% Remove some bad data
tt.trip_minutes = minutes(tt.tpep_dropoff_datetime - tt.tpep_pickup_datetime);
tt.speed_mph = tt.trip_distance ./ (tt.trip_minutes ./ 60);
ignore = tt.trip_minutes <= 1 | ...     % really short
    tt.trip_minutes >= 60 * 12 | ...    % unfeasibly long
    tt.trip_distance <= 1 | ...         % really short
    tt.trip_distance >= 12 * 55 | ...   % unfeasibly far
    tt.speed_mph > 55 | ...             % unfeasibly fast
    tt.total_amount < 0 | ...           % negative fares?!
    tt.total_amount > 10000;            % unfeasibly large fares
tt(ignore, :) = [];

% Credit card payments have the most accurate tip data
keep = tt.payment_type == {'Credit card'};
tt = tt(keep,:);

% Show tip distribu
histogram( tt.tip_a
```

Data only read once, despite 21 operations

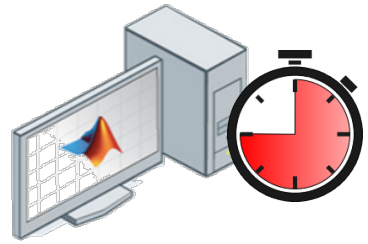
```
Evaluating tall expression using the Parallel Pool 'local':
- Pass 1 of 1: Completed in 4.9667 min
Evaluation completed in 5 min
```



Scaling up

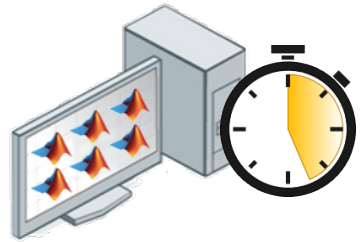
If you just have **MATLAB**:

- Run through each ‘chunk’ of data one by one



If you also have **Parallel Computing Toolbox**:

- Use all local cores to process several ‘chunks’ at once



If you also have a cluster with **MATLAB Distributed Computing Server (MDCS)**:

- Use the whole cluster to process many ‘chunks’ at once

Scaling up

Working with clusters from MATLAB desktop:

- General purpose MATLAB cluster
  - Can co-exist with other MATLAB workloads (parfor, parfeval, spmd, jobs and tasks, distributed arrays, …)
  - Uses local memory and file caches on workers for efficiency

- Spark-enabled Hadoop clusters
  - Data in HDFS
  - Calculation is scheduled to be near data
  - Uses Spark's built-in memory and disk caching

Distributed Arrays

- Unlike tall these need to fit in cluster memory
- Full range of numerical algorithms
  - Linear Algebra, eigenvalues, etc
- Sparse support

- Want to write your own function?
  - Full access to local part of the data
  - Ability to rebuild distributed array from local parts

Standard Benchmarks

HPL

```
spmd
    A = rand(10000, codistributor2dbc);
    b = rand(10000, 1, codistributor2dbc);
    x = A\b;
end
```

Create 2D Block Cyclic distribution appropriate for matrix solve

FFT

```
D = rand(1e8, 1, 'distributed');
F = fft(D);
```

STREAM Triad

```
B = rand(1e8, 1, 'distributed');
C = rand(1e8, 1, 'distributed');
q = rand;
A = B + q*C;
```

Single Program, Multiple Data (`spmd`)

- Everyone executes the same program
  - Just with different data
  - Inter-lab communication library enabled (MPI)
  - Call **`labindex`** and **`numlabs`** to distinguish labs
- Example

```
x = 1
spmd
    y = x + labindex;
end
```

Variable types can change across `spmd`

```
x = 1;
assert( isa(x, 'double') )
spmd
    assert( isa(x, 'double') )
    y = labindex;
end
assert( isa(y 'Composite') )
y{1} == 1; % TRUE

spmd
    assert( isa(y, 'double') )
end
```

Ordinary types are broadcast to all labs

Returned ordinary type is referenced by a Composite

Composite can be dereferenced on the client

Composite becomes the contained ordinary type on a lab

Want to write an MPI Program?

Send / receive data between labs

```
labSend
labReceive
labSendReceive
labProbe
```

Who am I? Where am I?

```
labindex
numlabs
```

Global Operations across labs

```
gplus
gcat
gop

labBarrier
labBroadcast
```

RandomAccess – an interesting MPI program

```
spmd
    % Initialize random number stream on each worker
    randRA( (labindex-1) * m * 4 / numlabs, 'StreamOffset' );
    t1 = tic;
    for k = 1:nloops
        % Make the local chunk of random data
        list = randRA( b );
        % Loop over the hyper-cube dimensions
        for d = 0:logNumlabs-1
            % Choose my partner for this dimension of the hypercube
            partner       = 1 + bitxor( (labindex-1), 2.^d );
            % Choose my mask for this dimension of the hypercube
            dim_mask      = uint64( 2.^( d + logLocalSize ) );
            % Choose which data to send and receive for this dimension
            dataToSend = logical( bitand( list, dim_mask ) );
            if partner <= labindex
                dataToSend = ~dataToSend;
            end
            % Setup a list of data that will be sent, and list I will keep
            send_list = list( dataToSend );
            keep_list = list( ~dataToSend );
            % Use send/receive to get some data that we should use next round
            recv_list = labSendReceive( partner, partner, send_list );
            % Our new list is the old list and what we've received
            list      = [keep_list, recv_list];
        end
        % Finally, after all rounds of communication, perform the table updates.
        idx = 1 + double( bitand( localMask, list ) );
        T(idx) = bitxor( T(idx), list );
    end
    % Calculate max time
    t = gop( @max, toc( t1 ) );
end
```

RandomAccess – an interesting MPI program

```
spmd
    t1 = tic;
    for k = 1:nloops
        % Make the local chunk of random data
        list = randRA( b );
        % Loop over the hyper-cube dimensions
        for d = myDims(0:logNumlabs-1, labindex)
            [send_list, keep_list] = partitionData(list, d);
            % Use send/receive to transfer data
            recv_list = labSendReceive(to(d), from(d), send_list);
            % Our new list is the old list and what we've received
            list = [keep_list, recv_list];
        end
        % Finally, perform the table updates.
        T = updateTable(list);
    end
    % Calculate max time
    t = gop( @max, toc( t1 ) );
end
```

Summary

- Simple array types for data analytics and PGAS programming
- Extensive parallel language suited to all types of problem

- All runs in both interactive and batch (off-line) mode