Adaptive Algorithms and Parallel Computing
Multicore and Multimachine Computing with MATLAB

Academic year 2016-2017

Simone Scardapane
Content of the Lecture

Main points:
- Parallelizing loops with the `par-for` construct.
- Understanding “single program, multiple data” (SPMD).
- Distributed arrays.
- Communication between workers.
- Interactive parallel programming.
- MapReduce and Hadoop.

Some caveats:
- This is not a comprehensive survey of parallelization in MATLAB.
- Code and functionalities are updated to MATLAB R2015b.
Overview

1. BASIC CONCEPTS
   Basic Ideas of Parallel Computing
   Introduction to the Parallel Computing Toolbox

2. PARALLEL-FOR
   Understanding par-for
   Sliced, reduction, and temporary variables
   Parfor limitations
   Parfor Examples

3. SPMD
   Single-program, multiple-data
   Example: parallel restarts in optimization
   Composites and distributed arrays

4. COMMUNICATION
   Communication functions
   Example: a simple diffusion LMS
   Debugging and Profiling

5. MAPREDUCE
   Commodity computing platforms
   MapReduce in MATLAB
   Example: counting words
   Example 2: Linear regression
Types of parallelism

Parallel computing refers to executing multiple portions of a program at the same time on different hardware components. Parallel behaviors can be categorized in several ways, such as the granularity of parallelism:

- **Bit-level**: each processor does not operate on single bits, but on *words* of multiple bits (e.g. 32 or 64 bit architectures). This is the most basic form of parallelization.

- **Instruction-level**: this is the capability of parallelizing *multiple* CPU instruction in a single cycle. This is declined in various forms, including pipelining, superscalar processors, speculative execution, etc.

- **Task-level**: executing the same or different code on multiple *threads*, which can operate on the same or different data.
Moore’s Law

• In 1965 Gordon Moore predicted an exponential increase in the number of elements per integrated circuit.

• 40 years later, Intel switched from increasing frequency scaling (due to the breakdown of the so-called ‘Dennard scaling’) in favor of multi-core architectures.

• Nowadays, we may be witnessing the end of the multi-core era towards different paradigms (three-dimensional circuits, quantum computing, etc.) [Mar14].

• At the same, there is a renewed interest in (a) commodity clusters (e.g. Hadoop/MapReduce); (b) computing with dedicated architectural components; and (c) increasing parallelism with general purpose GPU cards.
Amdahl’s Law

A first ‘problem’ of parallel computing (compared to classical CPU frequency scaling) is that the speedup of parallelization is not linear w.r.t. the number of threads. If we assume that only a portion \((1 - \beta) \in [0, 1]\) of a program can be fully parallelized, and the rest has to be run serially, the time \(T(n)\) required to run the program on \(n\) threads is given by:

\[
T(n) = \beta T(1) + \frac{1}{n} (1 - \beta) T(1)
\] (1)

This is known as Amdahl’s law. It is straightforward to compute the speedup provided by parallelizing the execution:

\[
S(n) = \frac{T(1)}{T(n)} = \frac{T(1)}{\beta T(1) + \frac{1}{n} (1 - \beta) T(1)} = \frac{1}{\beta + \frac{1}{n} (1 - \beta)}
\] (2)
Amdahl’s Law (2)

**Figure:** Graphical depiction of Amdahl’s Law when considering programs with different levels of parallelization [SOURCE].
Dependencies

Another essential concept in parallel computing is the idea of dependency. One portion of the program is dependent of a previous one if one of the following conditions hold:

1. One output of the first portion is required by the second (flow dependency).
2. One output of the second portion is required by the first (anti-dependency).
3. Both segments output the same variable (output dependency).

These are known as Bernstein’s conditions. One way of circumventing them is to consider some form of shared memory between the portions, which brings the necessity of synchronizing reading and writing operations.
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **SPMD**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
The MATLAB Ecosystem

Figure: Products in the MATLAB family [SOURCE].
Implicit and explicit parallelization

In MATLAB, several built-in functions performing numerical operations are multi-threaded by default (so-called *implicit parallelism*). Conversely, you can parallelize your own code using the capabilities of the Parallel Computing Toolbox (PCT). In this lecture, we focus on several of its features: par-for loops, spmd blocks, distributed arrays, etc.

An important aspect of the PCT is that the same code can scale transparently to any cluster configuration (more on this later on). Although we consider only the PCT, other open-source possibilities are available, such as the “Parallel Matlab Toolbox” (pMatlab):

http://www.ll.mit.edu/mission/isr/pmatlab/pmatlab.html
Under the hood of implicit multi-threading

How effective is the built-in multi-threading of MATLAB? Let us try a simple benchmark:

```matlab
tic;
randn(10000,10000)\randn(10000,1);
fprintf('Elapsed time is %.2f secs.\n', toc);
```

On an Intel i7, 16 GB RAM, with MATLAB R2013a, this takes (in average) 11 seconds. Disabling multi-threading, the time rises to 33 seconds.

You can disable multi-threading with the starting flag `-singleCompThread`. Currently, you can also control it in Matlab with the function `maxNumCompThreads`, but this feature will be removed in future releases.
Depreciated Functions

• A function such as `maxNumCompThreads` is called deprecated: it is kept for compatibility, but it will be removed in future releases.

• Starting in R2015a, the documentation of MATLAB provides an annotation detailing the release of introduction for each feature.

• Additionally, MATLAB offers several functions to check the version:

```
version % MATLAB version number and release
ver % Detailed information on the Mathworks products
```

• You also have the possibility of defining code conditionally on the version:

```
if verLessThan('matlab', '7.0.1')
    error('MATLAB version not supported. ');
end
```
Workers

A **worker** is the individual computational engine of the PCT. Typically, workers are started one per core, in a single-thread mode. They are independent one of another, having their own workspace, although they possess the capability of communicating between them. A set of workers is called a **pool**. To start it:

```matlab
matlabpool open  % Older versions
p = parpool       % New versions
```

Similarly, to stop it:

```matlab
matlabpool close  % Old versions
delete(p)         % New versions
```

The number of workers, along with their type, is specified in a **cluster configuration**. The opening command with no argument loads the “local” configuration.
Local and remote workers

Workers can be **local** or **remote**. Remote workers are handled by the Matlab Distributed Computing Server (DCS). Within a network environment, a **job scheduler** is used to distribute computation between the workers (image copyright of Mathworks):
Job Schedulers

- You can have one *head* node, several *worker* nodes, and one or more *client* nodes.
- You can run jobs using multiple job schedulers. This lecture assumes the standard MATLAB one, called MATLAB Job Scheduler.
- To install it, you can follow the instructions on the MATLAB documentation: Configure Cluster to Use a MATLAB Job Scheduler (MJS).
- On the client, you can select a specific cluster from Parallel >> Manage Cluster Profiles. You can have more than one profile, and switch them manually or programmatically.
Cluster Configuration (screenshot)
Admin Center (screenshot)
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
     - Sliced, reduction, and temporary variables
     - Parfor limitations
     - Parfor Examples

3. **SPMD**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - **MapReduce in MATLAB**
   - Example: counting words
   - Example 2: Linear regression
Introductory Example

Consider this very simple code:

```matlab
x = randn(50, 2000); y = zeros(50,1);
for ii = 1:50
    y(ii) = my_function(X(ii,:));
end
```

Each cycle of the for-loop is completely independent of the others. If we have a pool available, the previous computation can be distributed among workers using a par-for construct:

```matlab
x = randn(50, 2000); y = zeros(50,1);
parfor ii = 1:50
    y(ii) = my_function(X(ii,:));
end
```
Parfor execution

When designing a parallel for, two concepts are essential to understand:

• Iterations of the loop are executed in no particular order.
• MATLAB analyzes what data must be sent to the workers at the beginning of the par-for loop. This puts several constraints on your code (more on this later).

As a rule of thumb, you should consider using a par-for whenever you need to perform several times a relatively simple computation, without dependencies between each computation. As a second rule, try to decide on the par-for before writing the code, instead of adapting it later on.
Overview

1 **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2 **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3 **SPMD**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4 **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5 **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
Sliced variables

In a par-for, a variable is **sliced** if it can be decomposed into independent groups over the workers. A prerequisite is that a variable must be indexed *consistently* inside a par-for. In the following example, $X$ is sliced, $Y$ is not:

```plaintext
parfor ii = 1:3
    a = X(A(ii), A(ii));
    b = Y(A(ii), A(ii+1));
end
```

To be sliced, no elements can be inserted or deleted from a matrix inside a for-loop. Here, $X$ is not sliced:

```plaintext
parfor ii = 1:3
    X(ii,:) = [];
end
```
Sliced, reduction, and temporary variables

**Sliced variables /2**

Sliced variables are important because they reduce the communication overhead of transmitting data to the workers. Note that a variable can be either an **input** sliced variable, or an **output** sliced variable:

```matlab
parfor ii = 1:3
    a = X(ii); % X is an input sliced variable
    Y(ii) = a; % Y is an output sliced variable
end
```

A variable which is not sliced, nor assigned in the for-loop is called a **broadcast** variable.

You should strive to make as much variables sliced as possible.
Reduction variables

A **reduction** variable is the only exception to the rule that “loops must be independent”. A reduction assignment *accumulates* iteratively the result of a computation, with the prerequisite that the operation must not depend on the ordering of the iterations. The easiest example:

```matlab
a = 0;
parfor ii = 1:3
    a = a + X(ii);  % a is the reduction variable
end
```

MATLAB handles reduction assignment in a particular way, leading to some surprising results. Consider the following:

```matlab
a = [];
parfor ii = 1:3
    a = [a; X(ii)];  % concatenation is a valid reduction assignment
end
```

Despite the fact that iterations are not executed in order, the resulting concatenated vector is ordered.
Reduction variables /2

Other examples of reduction operators are $\ast$ (matrix product), .$\ast$ (point-wise product), $\&$ (logical and), $\text{min}$ (minimum), $\text{union}$ (set union), etc. You can use your own reduction function:

```matlab
f = @my_fcn;
parfor ii = 1:3
    a = f(a, X(ii));
end
```

You must ensure that the function $f$ is associative, i.e.:

$$f(a, f(b, c)) = f(f(a, b), c)$$

Commutativity is not required because you are not allowed to change the order of the operands inside the for-loop.
Temporary variables

The last special class of variables to consider are temporary variables. A temporary variable is a variable that we assign directly without a reduction assignment:

```plaintext
parfor ii = 1:3
    d = x(ii); % d is temporary
end
```

Temporary variables are effectively destroyed and recreated at every iteration. Moreover, they are not passed back to the client when the computation is over. The main consequence is the following:

```plaintext
d = 1;
parfor ii = 1:3
    d = 2;
end
display(d); % d has value 1
```
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **SPMD**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
Transparency

All the variables inside a par-for loop must be *transparent*, i.e., they must be visible to the interpreter. As a consequence, the following set of actions is not allowed inside a par-for:

- Calling a function such as `eval`, because the input to it is a string, not a variable. This code will not execute:

  ```
  my_variable = 3;
  parfor ii = 1:4
      eval('my_variable');
  end
  ```

- Loading/saving a .mat file, or clearing any workspace variables.
- Calling an external script (although functions are allowed).
- Requesting any form of user input.
Nested for-loops

A par-for cannot be nested inside another par-for, but a for loop can. However, this has a set of limitations. Let use see two of them. First of all, the range of a nested loop must be a constant. The following will not work:

```matlab
parfor ii = 1:3
    for jj = my_range(ii)
        [...]
    end
end
```

Additionally, the nested index variable cannot be used as part of an expression for indexing a sliced matrix. The following is invalid:

```matlab
A = zeros(3, 3);
parfor ii = 1:3
    for jj = 1:3
        A(ii, jj+1) = i + j;
    end
end
```
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **SPMD**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
Example 1 - Batch Image Processing

- In this experiment, we perform some basic image processing on a batch of images.
- We can easily parallelize the code by splitting the images over the different workers.
- Source images for this example are available on the web: SOURCE.
Example 1 - Serial version

```
d = 'data_GT';  % Directory of images
imgs = dir(d);  % Get a list of the files

% Remove '.' and '..' from the list
imgs = imgs(arrayfun(@(x) ~strcmp(x.name(1), '.' ) && ~strcmp(x.name(1), '..' ), imgs));

imgs = {imgs.name}';  % Simply store the image names
N_imgs = length(imgs);
imgs_o = cell(N_imgs, 1);

for ii = 1:N_imgs
    im = imread(fullfile(d, imgs{ii}));  % Read the image
    imgs_o{ii} = rgb2gray(im);  % Convert to bw
    imgs_o{ii} = imadjust(imgs_o{ii});  % Enhance contrast
end
```
Example 1 - Result

(a) Before processing
(b) After processing

Figure: First image in the batch before and after processing.
Example 1 - Parallel version

```matlab
parfor ii = 1:N_imgs
    fprintf('Processing %s...
', imgs{ii});
    im = imread(fullfile(d, imgs{ii}));
    imgs_o{ii} = rgb2gray(im);
    imgs_o{ii} = imadjust(imgs_o{ii});
end
```

- It’s the same!
- Note that `imgs` is a sliced input variable, `imgs_o` is a sliced output variable, `im` is temporary.
- Look at the order of printing instructions: the images are not elaborated in alphabetical order anymore.
Example 2 - Computing the Mandelbrot set

The following example is adapted from [Kep09]. The Mandelbrot set is the set of complex numbers $c$ such that the following succession remains bounded:

$$z_{i+1} = z_i^2 + c$$  \hspace{1cm} (3)

with initial value $z_0 = 0$. In the following, we sweep the region defined by $\Re(c) \in [-1.4, 0.6]$ and $\Im(c) \in [-1, 1]$.

Our method will take 3 parameters: $N$ is the number of points on each dimension, $\text{max\_iter}$ is the number of iterations to perform, and $\text{epsilon}$ is a tolerance.
Example 2 - The code (serial version)

```matlab
function W = mandelbrot_serial(N, max_iter, epsilon)
    Z = zeros(N, N); W = zeros(N, N); indices = true(N,N);
    steps = (1:N)';
    [ReC, ImC] = meshgrid(steps'./(N/2) - 1.6, steps./(N/2) - 1);
    C = ReC + 1i*ImC;

    for i = 1:max_iter
        Z(indices) = Z(indices).*Z(indices) + C(indices);
        W(indices) = exp(-abs(Z(indices)));
        indices = W > epsilon;
    end
end
```
Example 2 - Output

**Figure:** Weights of the Mandelbrot set, shown as an image with the "gray" colormap.
Example 2 - The code (parallel version)

We distribute the computation column-wise:

```matlab
function W = mandelbrot_parallel_parfor(N, max_iter, epsilon)

W = zeros(N, N);

parfor i = 1:N
    Z = zeros(1, N); W_local = zeros(1,N); indices = true(1,N);
    steps = (1:N)';
    C_local = (steps'./(N/2) − 1.6) + 1i*(i./(N/2) − 1);
    for j=1:max_iter
        Z(indices) = Z(indices).*Z(indices) + C_local(indices);
        W_local(indices) = exp(-abs(Z(indices)));
        indices = W_local > epsilon;
    end
    W(i,:) = W_local;
end
end
```
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **Spmd**
   - Single-program, multiple-data
     - Example: parallel restarts in optimization
     - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
The SPMD construct

The SPMD (Single-Program, Multiple-Data) construct is apparently similar to par-for, but it is largely different. In SPMD, there is a fixed number of workers (also called labs) that execute the same program. Differences are that each lab knows its id, and it can eventually communicate with all the others. Moreover, data on the workers is kept between SPMD commands. A simple operation on all the available workers:

```matlab
spmd
    my_fcn(2);
end
```

We can also instantiate a given number of labs, e.g. 2:

```matlab
spmd(2)
    my_fcn(2);
end
```
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **SPMD**
   - Single-program, multiple-data
   - **Example: parallel restarts in optimization**
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - **MapReduce in MATLAB**
   - Example: counting words
   - Example 2: Linear regression
Example: parallel restarts in optimization

Description of the example

As an example of SPMD programming, consider the following. We want to minimize the following function:

\[ f(x) = \frac{\cos(3\pi x)}{x} \]

We will use a *steepest-descent* algorithm (SDA) [BV04]. This procedure, however, may converge to a local minimum, hence we want to run it starting from several different points. In particular, we will run it twice in parallel using SPMD: one time starting in \([0, 1]\), the second time starting in \([1, 2]\).
Steepest-descent optimization (recall)

We remember some concepts from unconstrained optimization. We want to minimize a function $f(x)$, $x \in \mathbb{R}^d$ on the whole domain, under the assumption that $f$ is continuous and differentiable. Starting from a point $x_0$, SDA updates iteratively the estimate as:

$$x_{i+1} = x_i - \mu \nabla f(x_i)$$  \hspace{1cm} (4)

where $\nabla f(x_i)$ is the gradient of $f$ evaluated in $x_i$ and $\mu$ a parameter known as step-size. Supposing that $\mu$ satisfies the Wolfe conditions at every iteration [BV04], Eq. (4) will converge to a local minimum [BV04]. For simplicity, we will consider a small, fixed $\mu$ in our implementation.
Steepest-descent code

For computing the derivative, we use the Symbolic Toolbox of MATLAB. In the code, $f$ is a symbolic expression, $x_0$ is the starting point, step_size is the step size, and tol is a tolerance to use as stopping criterion:

```matlab
function x_best = grad_descent( f, x0, step_size, tol )

f_prime = diff(f); % Derivative of f

gap = Inf; % We exit whenever gap > tol
x_best = x0; % Current best estimate

while gap > tol
    descent_dir = -double(f_prime(x_best));
    x_new = x_best + step_size*descent_dir;
    gap = abs(x_new - x_best);
    x_best = x_new;
end

fprintf( 'Minimum reached: %.2f\n', double(f(x_best)));
end
```
Example: parallel restarts in optimization

SPMD code

```matlab
syms x;
f = symfun(cos(3*pi*x)./x, x);

matlabpool open

spmd
    % labindex returns the id of the current worker
    x_best = grad_descent(f, (labindex - 1) + rand(), 0.01, 10^-4);
end

matlabpool close
```

The output on the console:

Starting matlabpool using the 'local' profile ...
    connected to 2 workers.
Lab 1:
    Minimum reached: -3.17
Lab 2:
    Minimum reached: -1.01
Sending a stop signal to all the workers ... stopped.
Example: parallel restarts in optimization

The result

**Figure:** Result of running in parallel an SDA procedure from two different starting points.
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **Spmd**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
Composites

In the previous example, \( x_{\text{best}} \) is a Composite object. A composite is similar to an \( N \)-dimensional cell array, where each element is the result of the computation on the \( i \)-th lab.

You can create a composite outside a SPMD block:

\[
a = \text{Composite}();
\]

You can access and set each element using curly brackets:

\[
a = \text{Composite}(2);
\]
\[
\text{spmd}(2)
  \quad a = \text{my}_\text{fcn}();
\text{end}
\]

\[
b = a\{1\} \quad \text{% access value from lab 1}
\]
\[
a\{2\} = 1; \quad \text{% set value on lab 2 (for successive spmd calls)}
\]
Creating a distributed array

An important aspect of SPMD is the possibility of distributing a single array over the workers. This can be done in three different ways:

• Calling the distributed function with an existing array.
• Using an overloaded constructor with the “distributed” flag.
• Creating the distributed array directly on the workers (in this case, it is called codistributed).

A large set of MATLAB functions are overloaded to work on distributed arrays:

Distributing an existing array

Create a matrix, distribute it, and perform an inversion on the workers:

```matlab
A = randn(500, 500);
A = distributed(A);
Ainv = inv(A);
```

Note that `Ainv` is implicitly distributed, which can be seen by analyzing the workspace:

```bash
>> whos
Name      Size              Bytes  Class       Attributes
A         500x500           509    distributed
Ainv      500x500           509    distributed

We can collect the array on the client using the `gather` function:

```matlab
Ainv = gather(Ainv);
```
Overloaded constructors

In the previous examples, the computation is parallelized implicitly. Here is an equivalent version, using the distributed constructor for `randn`, and performing the computation inside a SPMD block:

```plaintext
A = distributed.randn(500, 500);
spmd
    Ainv = inv(A);
end
```

In cluster configurations, this allows you to construct arrays that are bigger than the memory limit on the single machine.

You can change the way in which the array is partitioned using a codistributor object:

http://www.mathworks.it/it/help/distcomp/codistributor.html
Codistributed arrays

A distributed array can also be created inside an SPMD block. In this case, it is called a **codistributed** array. Let us reformulate the previous example:

```matlab
spmd
    A = codistributed.randn(500, 500);
    Ainv = inv(A);
end
```

Inside a lab, you can obtain the local portion of the array using the `getLocalPart` method:

```matlab
spmd
    A = codistributed.randn(500, 500);
    A_local = getLocalPart(A);
end
```
Overview

1 Basic Concepts
   Basic Ideas of Parallel Computing
   Introduction to the Parallel Computing Toolbox

2 Parallel-For
   Understanding par-for
   Sliced, reduction, and temporary variables
   Parfor limitations
   Parfor Examples

3 Spmd
   Single-program, multiple-data
   Example: parallel restarts in optimization
   Composites and distributed arrays

4 Communication
   Communication functions
   Example: a simple diffusion LMS
   Debugging and Profiling

5 MapReduce
   Commodity computing platforms
   MapReduce in MATLAB
   Example: counting words
   Example 2: Linear regression
Worker communication

SPMD allows workers to communicate through the use of a simple set of functions:

- **labBarrier** blocks the execution until all the workers are at the given point.
- **labBroadcast** sends a value from a worker to all the others.
- **labSend** sends a value from a worker to another.
- **labReceive** receives a value from another worker.

And a few others:

http://www.mathworks.it/it/help/distcomp/task-control-and-worker-communication.html

We show the use of these functions by implementing a simple Diffusion Least Mean-square filter [LS08].
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **SPMD**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
Least mean-square filter (recall)

Given an input buffer $x_i, i = 1, \ldots, n$, and a desired output $d_i, i = 1, \ldots, n$, we want a linear model of the form $f(x) = w^T x$.

Starting from an initial estimate $w_0$, at every iteration we apply a steepest-descent approach, approximating the overall error with the instantaneous error $e_i = d_i - f(x_i)$. The resulting filter is called the Least Mean-Square (LMS) adaptive filter:

$$w_{i+1} = w_i - \mu e_i x_i$$  \hspace{1cm} (5)
Example: a simple diffusion LMS

Diffusion LMS

Consider a network of $N$ agents. Every agent has its own version of the input signal, and its own estimate of the optimal weights. In particular, at time instant $i$, the $k$-th agent receives an input buffer $x_{k,i}$, it can communicate with a set of neighbors $\mathcal{N}_k$, and it has an estimate of the weights given by $w_{k,i}$. A diffused version of the LMS is obtained as follows:

1. Each agent constructs a weighted sum of the estimates of its neighbors: $\phi_{k,i} = \sum_{l \in \mathcal{N}_k \cup k} s_{kl} w_{l,i}$. The weights $s_{jz}$ are called trusting factors, and they are set in the beginning (for simplicity).
2. Then, it applies an LMS rule to the new weight vector $\phi_{k,i}$, using its version of the input.

See a full discussion in [LS08].
Example: a simple diffusion LMS

The code

For simplicity, we suppose that:

• All the $N$ agents are connected to all the others.
• Step size at every node is extracted in $[0, 0.01]$.
• The trust weights are all equal to $s = 1/N$.
• The agents differentiate themselves on the initialization of the weight vector and on the specific input signal.
• The input signal is white noise.
• There is random noise on the output extracted from a Gaussian distribution with 0 mean and $\sigma^2 = 0.01$.

Removing the constraint on the connectivity is left as an exercise.
We start by initializing all the required values:

```matlab
% Define system to be identified
wo = [0.54; 0.95; -0.96; -0.02; -0.13];
n_agents = 2;
N = 10000;

matlabpool('open', n_agents);
m = length(wo);
equal_trust = 1/n_agents;

spmd
    step_size = rand*0.01;
    x = randn(N, m);
    y = x*wo + randn(N, 1)*0.01;
    wopt = rand(m, 1);
    err = zeros(N, 1);
end
```
Example: a simple diffusion LMS

The code /3

We apply the diffused rule at every instant:

```matlab
spmd
    for i = 1:N

        % Combine the weights
        phi = equal_trust*wopt;
        labSend(wopt, find(1:n_agents ~= labindex));

        for j = 1:n_agents
            if (j ~= labindex)
                while (~labProbe(j))
                    end
                phi = phi + equal_trust*labReceive(j);
            end
        end

        % Adapt the filter
        err(i) = y(i) - x(i, :) * phi;
        wopt = phi + step_size*err(i)*x(i, :)
    end
end
matlabpool('close');
```
Testing the diffused LMS (6 agents, no diffusion)

Figure: Evolution of the MSE in a network with no cooperation.
Testing the diffused LMS (6 agents, with diffusion)

**Figure:** Evolution of the MSE in a network with diffusion.
Overview

1 Basic Concepts
   Basic Ideas of Parallel Computing
   Introduction to the Parallel Computing Toolbox

2 Parallel-For
   Understanding par-for
   Sliced, reduction, and temporary variables
   Parfor limitations
   Parfor Examples

3 Spmd
   Single-program, multiple-data
   Example: parallel restarts in optimization
   Composites and distributed arrays

4 Communication
   Communication functions
   Example: a simple diffusion LMS
   Debugging and Profiling

5 MapReduce
   Commodity computing platforms
   MapReduce in MATLAB
   Example: counting words
   Example 2: Linear regression
Interactive mode

Debugging parallel applications can be complicated. MATLAB offers the possibility of running an interactive parallel mode, to see the output of a given set of commands on multiple workers. We can start it using:

```matlab
>> pmode start
```

We enter the desired command in the interactive window:

```matlab
P>> wopt = ones(d, 1)*labindex;
```
Interactive debugging

```python
>>> wopt = ones(d, 1)*labindex

wopt =

 1
 1
 1

>>> wocs = ones(d, 1)*labindex

wocs =

 2
 2
 2
```

Debugging and Profiling
Parallel profiler

A second tool for debugging is the *parallel profiler*, which provides information on the functions called inside each SPMD block at the worker level. In an SPMD block, the profiler is started as:

```
mpiprofile on -details builtin
```

Then, after the parallel code, but before exiting the SPMD block, you can save the information and close the profiler:

```
info = mpiprofile('info');
mpiprofile off
```

Finally, to view the information (e.g., for worker 1):

```
mpiprofile('viewer', info{1});
```
Parallel Profiler (Example 1/5)

We reformulate the main SPMD block as a function:

```matlab
function err = dlms_local(x, y, wopt, step_size, equal_trust, n_agents)

N = length(y); err = zeros(N, 1);

for i = 1:N
    % Combine the weights
    phi = equal_trust*wopt;
    labSend(wopt, find(1:n_agents ~= labindex));

    for j = 1:n_agents
        if(j ~= labindex)
            while(~labProbe(j))
                end
            phi = phi + equal_trust*labReceive(j);
        end
    end

    % Adapt the filter
    err(i) = y(i) - x(i, :) * phi;
    wopt = phi + step_size*err(i)*x(i, :)';
end
```
Parallel Profiler (Example 2/5)

Now we can call the profiler to monitor the function:

```matlab
spmd
    mpiprofile on -detail builtin
    err = dlms_local(x, y, wopt, step_size, equal_trust, n_agents);
    info = mpiprofile('info');
    mpiprofile('info');
end
mpiprofile('viewer', info{1}); % Get information on worker 1
```

Note that we are also tracking the builtin functions of MATLAB.
Parallel Profiler (Example 3/5)

Parallel Profile Summary

Showing all functions called in worker

<table>
<thead>
<tr>
<th>Automatic Comparison Selection</th>
<th>Manual Comparison Selection</th>
<th>Show Figures (all workers)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Compare (max vs. min TotalTime)</td>
<td>Go to worker: max TotalTime</td>
<td></td>
</tr>
<tr>
<td>Compare (max vs. min CommTime)</td>
<td>Compare with: None</td>
<td></td>
</tr>
</tbody>
</table>

**Communication statistics are not available for ScalAPACK functions, so data marked with ** might be inaccurate.

<table>
<thead>
<tr>
<th>Function Name</th>
<th>Calls</th>
<th>Total Time</th>
<th>Self Time*</th>
<th>Total Comm Time</th>
<th>Self Comm Waiting Time</th>
<th>Total Inter-worker Data</th>
<th>Computation Time Ratio</th>
<th>Total Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>dims_local</td>
<td>1</td>
<td>14.718 s</td>
<td>14.718 s</td>
<td>3.748 s</td>
<td>0.250 s</td>
<td>5.72 Mb</td>
<td>74.6%</td>
<td></td>
</tr>
</tbody>
</table>

Self time is the time spent in a function excluding the time spent in its child functions. Self time also includes overhead resulting from the process of profiling.

**Communication statistics are not available for ScalAPACK functions, so data marked with ** might be inaccurate.
Parallel Profiler (Example 4/5)

dlms_local(1 call(s), 14.718 sec, 2.86 Mb sent, 2.86 Mb rec, 0.250 s wait, 3.498 s act comm)
Total computation took 74.53% of total time.

Generated 02-Apr-2015 12:50:49 using real time.

Showing this function's statistics on worker 1

function in file C:/Users/SPAWM/Downloads/dlms_local.m
(Copy to new window for comparing multiple runs)

Please note that the code displayed is taken from the client, and might have changed since execution on the cluster. Only valid MATLAB functions will be shown

Refresh

☑ Show parent functions ☐ Show busy lines ☑ Show child functions
☑ Show Code Analyzer results Adj Show file coverage Adj Show function listing

Sort busy lines and graph according to

Parents (calling functions)
No parent

Lines where the most time was spent.

<table>
<thead>
<tr>
<th>Line Number</th>
<th>Code</th>
<th>Calls</th>
<th>Total Time</th>
<th>Data Sent</th>
<th>Data Rec</th>
<th>Comm/Wating Time</th>
<th>Active Comm Time</th>
<th>% Time</th>
<th>Time Plot</th>
</tr>
</thead>
<tbody>
<tr>
<td>15</td>
<td>end</td>
<td>484713</td>
<td>9.752 s</td>
<td>0 b</td>
<td>0 b</td>
<td>0 s</td>
<td>0 s</td>
<td>66.3%</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>phi = phi + equal Trust taskRec...</td>
<td>25000</td>
<td>2.531 s</td>
<td>0 b</td>
<td>2.86 Mb</td>
<td>0.250 s</td>
<td>1.845 s</td>
<td>20.3%</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>labSend(wopc, find(1:n_agents ...</td>
<td>5000</td>
<td>0.526 s</td>
<td>2.86 Mb</td>
<td>0 b</td>
<td>0 s</td>
<td>1.653 s</td>
<td>3.6%</td>
<td></td>
</tr>
<tr>
<td>14</td>
<td>while(~lab2cobe(j))</td>
<td>25000</td>
<td>0.476 s</td>
<td>0 b</td>
<td>0 b</td>
<td>0 s</td>
<td>0 s</td>
<td>3.2%</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>if(j ~= labindex)</td>
<td>30000</td>
<td>0.396 s</td>
<td>0 b</td>
<td>0 b</td>
<td>0 s</td>
<td>0 s</td>
<td>2.7%</td>
<td></td>
</tr>
<tr>
<td>All other lines</td>
<td></td>
<td>0.577 s</td>
<td>0 b</td>
<td>0 b</td>
<td>0 s</td>
<td>0 s</td>
<td>3.9%</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Totals</td>
<td></td>
<td>14.718 s</td>
<td>2.86 Mb</td>
<td>2.86 Mb</td>
<td>0.250 s</td>
<td>3.498 s</td>
<td>100%</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Debugging and Profiling

Parallel Profiler (Example 5/5)

```matlab
function err = dims_local(x, y, wopt, step_size, equal_trust, n_agents)
    N = length(y);
    err = zeros(N, 1);
    for i = 1:N
        % Combine the weights
        phi = equal_trust*wopt;
        labSend(wopt, find(1:n_agents ~= labindex));
        for j = 1:n_agents
            if(j ~= labindex)
                while(~labProbe(j))
                    end
                end
            end
        end
    end
    % Adapt the filter
    err(i) = y(i) - x(i, :) * phi;
    wopt = phi + step_size*err(i) * x(i, :)
end
```

There is probably a better way to implement this... You can also combine the parallel profiler with the interactive mode to get information on the full cluster.
Overview

1 Basic Concepts
   Basic Ideas of Parallel Computing
   Introduction to the Parallel Computing Toolbox

2 Parallel-For
   Understanding par-for
   Sliced, reduction, and temporary variables
   Parfor limitations
   Parfor Examples

3 Spmd
   Single-program, multiple-data
   Example: parallel restarts in optimization
   Composites and distributed arrays

4 Communication
   Communication functions
   Example: a simple diffusion LMS
   Debugging and Profiling

5 MapReduce
   Commodity computing platforms
   MapReduce in MATLAB
   Example: counting words
   Example 2: Linear regression
Commodity Computing

Commodity computing (CC) is the use of a large number of “commodity” (i.e. low performance) components for the purpose of parallelization.

A CC cluster is made of standard computing elements, using classical operating systems and protocols. It is driven by decreasing costs from the vendors.

It is opposed to other parallel paradigms such as supercomputers, high performance clusters and grid computing.
Challenges of CC clusters

**Faults** Components might fail while performing their tasks. Data duplication is needed to prevent loss of information.

**Changing topology** New computers can connect/disconnect at any time.

**Performance** Each component has its own performance (e.g. high-end against low-end computers).

**Distributed data** Data to the problem is distributed throughout the network.

...  ...
Google and MapReduce

MapReduce (MR) is a programming framework developed by Google to address the previous problems. An MR program requires (at least) two components:

1. A **mapper** is used to filter the input data.
2. A **reducer** performs a summary of the information provided by the mapper.

The MR framework takes charge of running in parallel multiple mappers/reducers, handles data redundancy, faults, etc.
A preliminary example

Figure: Example of running MapReduce for counting words. Taken from: [Image source]
Formal definition

Input data to the problem must be composed of key/value pairs \((k, v)\), belonging to two generic domains \(k \in M_{in}\) and \(v \in V_{in}\). The data is initially filtered according to a function:

\[
MAP(k, v) = \text{list}(k_2, v_2),
\]

where the output data can belong to different domains \(k_2 \in M_{map}\) and \(v_2 \in V_{map}\). The results from the map operations can be shuffled and collected, and finally reduced using a different function:

\[
\text{REDUCE}(k_2, \text{list}(v_2)) = (k_2, \text{list}(v_3)),
\]

with \(v_3 \in V_{out}\).
History of Hadoop

- A white paper on MR by Google was published in 2004.
- This inspired Doug Cutting (a Yahoo employee) and Mike Cafarella to develop an open source version in Java under Apache license.
- The first version of Hadoop was released in 2011.
- Today, the use of Hadoop is widespread (more than half the Fortune 50 companies use it).
- Hadoop clusters can be bought from most cloud vendors.
- Additionally, several modules can be installed on top of Hadoop (e.g. Pig, Spark, etc.).
Hadoop layers

Figure: Layered architecture of Hadoop. Taken from: [Image source]
Overview

1 **Basic Concepts**
   Basic Ideas of Parallel Computing
   Introduction to the Parallel Computing Toolbox

2 **Parallel-For**
   Understanding par-for
   Sliced, reduction, and temporary variables
   Parfor limitations
   Parfor Examples

3 **SPMD**
   Single-program, multiple-data
   Example: parallel restarts in optimization
   Composites and distributed arrays

4 **Communication**
   Communication functions
   Example: a simple diffusion LMS
   Debugging and Profiling

5 **MapReduce**
   Commodity computing platforms
   MapReduce in MATLAB
   Example: counting words
   Example 2: Linear regression
MapReduce and MATLAB

It is possible to easily build a CC cluster using the MATLAB Distributed Computing Server (see previous lecture). The only requirement is for each component to run the same version of MATLAB.

The widespread use of Hadoop pushed Mathworks to release a custom implementation of the MR framework in MATLAB R2014b, via the `mapreduce` function.

MR programs implemented in MATLAB can be run from the desktop on a pre-existing Hadoop cluster, by changing the default job scheduler (more on this next).
Overview of MR in MATLAB

Figure: Overview of MR components in MATLAB. Taken from: [Image source]
**MR workflow in MATLAB**

1. The input data is saved in a particular object called *datastore*, which handles data distribution and partitioning in *chunks*.

2. Each data chunk is processed by a different map function, and the result is stored in an intermediated object of class *KeyValueStore*.

3. The intermediate outputs are grouped by key (i.e. by $k_2$ in our formal definition).

4. Each group of *KeyValueStore* elements is processed by a reduce function.

5. Final results are saved in an output *datastore* object.
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **Spmd**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - MapReduce in MATLAB
   - Example: counting words
   - Example 2: Linear regression
Example: counting words

Input data definition

Our input data is given by different text files:

**Input1.txt:**
This is an example

**Input2.txt:**
Another example

**Input3.txt:**
Yet Another example

The objective is writing an MR program that counts words from all the files.
Loading data in the datastore

As a first step, we load the data in the datastore object:

```matlab
ds = datastore({'input1.txt', 'input2.txt', 'input3.txt'}, 'ReadVariableNames', false, 'VariableNames', {'Word'});
```

MATLAB provides two different datastores: a TabularTextDatastore for text files, and a KeyValueDatastore for key/value pairs. MATLAB automatically recognizes the correct type of datastore to use:

```matlab
>> whos
    Name      Size    Bytes  Class                       Attributes
    ds        1x1      112  matlab.io.datastore.TabularTextDatastore
```
Writing a map function

A map function must take three arguments:

1. **data**: this contains a chunk of data saved in the datastore, which can be accessed with the standard dot notation.
2. **info**: additional information on the data.
3. **intermKVStore**: *intermediate* KeyValueStore object that will store the output of the mapper.

The **KeyValueStore** has only two possible methods:

1. **add** to add a single key/value pair.
2. **addmulti** to add multiple key/value pairs.
Our map function

```matlab
function mapCountWords(data, info, intermKVStore)

% Get unique words
words = unique(data.Word);

% Get indices
[~, idxs] = ismember(data.Word, words);

% Count the occurrences
words_count = arrayfun(@(x) sum(idxs == x), 1:numel(words));

% Add to KeyValueStore
addmulti(intermKVStore, words, num2cell(words_count));
end
```
Writing a reduce function

The reduce function is similar to the map function and takes three arguments:

1. `intermKey` is the key associated to the values.
2. `intermValIter` is a `ValueIterator` object for accessing the value associated to the key.
3. `outKVStore` is the store for saving the output results.

The `ValueIterator` has only two functions:

1. `hasnext` returns a boolean indicating whether there is another value available.
2. `getnext` accesses the next value.
Our reduce function

```matlab
function reduceCountWords(intermKey, intermValIter, outKVStore)

% Initialize the sum
sum_occurences = 0;

% Compute the sum
while hasnext(intermValIter)
    sum_occurences = sum_occurences + getnext(intermValIter);
end

% Add to the output KeyValueStore
add(outKVStore, intermKey, sum_occurences);
end
```
Running the MR job

```plaintext
>> outds = mapreduce(ds, @mapCountWords, @reduceCountWords);

Starting parallel pool (parpool) using the 'local' profile ...
connected to 2 workers.

Parallel mapreduce execution on the parallel pool:
******************************************************************************
* MAPREDUCE PROGRESS *
******************************************************************************
Map 0%  Reduce 0%
Map 25%  Reduce 0%
Map 50%  Reduce 0%
Map 75%  Reduce 0%
Map 100% Reduce 50%
Map 100% Reduce 100%
```
Visualizing the results

```matlab
>> readall(outds)

ans =
    Key     Value
    _______   _____
    'Example'    [3]
    'Yet'       [1]
    'An'        [1]
    'Another'   [2]
    'Is'        [1]
    'This'      [1]
```
Overview

1. **Basic Concepts**
   - Basic Ideas of Parallel Computing
   - Introduction to the Parallel Computing Toolbox

2. **Parallel-For**
   - Understanding par-for
   - Sliced, reduction, and temporary variables
   - Parfor limitations
   - Parfor Examples

3. **SPMD**
   - Single-program, multiple-data
   - Example: parallel restarts in optimization
   - Composites and distributed arrays

4. **Communication**
   - Communication functions
   - Example: a simple diffusion LMS
   - Debugging and Profiling

5. **MapReduce**
   - Commodity computing platforms
   - **MapReduce in MATLAB**
   - Example: counting words
   - Example 2: Linear regression
Example 2: Linear regression

**Linear regression**

We want to learn an unknown function given by:

\[ y = w_0^T x, \quad (8) \]

with \( x \in \mathbb{R}^d \) and \( y \in \mathbb{R} \). We are provided with a set of \( N \) examples of it:

\[ (x_1, y_1), (x_2, y_2), \ldots, (x_N, y_N) \quad (9) \]

The aim is to find a \( w \) such that the mean-squared error is minimized:

\[ w_* = \min_{w \in \mathbb{R}^d} \sum_{i=1}^{N} (w^T x_i - y_i)^2. \quad (10) \]
Example 2: Linear regression

Least-square solution

We build the input matrix $X$ by stacking the inputs:

$$X = \begin{bmatrix} x_1^T \\ \vdots \\ x_N^T \end{bmatrix}$$

(11)

and the output vector $y = [y_1, \ldots, y_N]^T$. It is easy to show that the solution to the previous optimization problem can be expressed in closed form as:

$$w_* = (X^TX)^{-1}X^Ty.$$  

(12)
Example 2: Linear regression

Formulation of the problem

In an MR framework, each mapper has access to a subset of $X$ and $y$, such that:

$$X = \begin{bmatrix} X_1 \\ \vdots \\ X_L \end{bmatrix}$$  \hspace{1cm} (13)

where $L$ is the number of mappers, and similarly for $y$. The final solution is given by:

$$w_* = \left( \sum_{k=1}^{L} X_k^T X_k \right)^{-1} \left( \sum_{k=1}^{L} X_k^T y_k \right).$$ \hspace{1cm} (14)

In a straightforward implementation, each mapper will compute its portion of the sum, and the reducer will aggregate them.
Preparing the data

We will use the *abalone* dataset, which has 8 input variables and a single output variable. The dataset is stored as two variables in a .mat file, so we first need to transform it as a .csv and load it into the datastore:

```matlab
% Load the abalone dataset
load abalone_dataset.mat

% Write to csv and open with datastore
csvwrite('abalone.csv', [abaloneInputs' abaloneTargets']);
ds = datastore({'abalone.csv'}, 'ReadVariableNames', false);
```

[Learn more about the Abalone dataset]
Example 2: Linear regression

Map function

```matlab
function mapLinearRegression(data, info, intermKVStore)

% Initialize
vars = {'Var1', 'Var2', 'Var3', 'Var4', 'Var5', 'Var6', 'Var7', 'Var8'};
X = zeros(length(data.Var1), 8);
y = data.Var9;

% Get each column
for i = 1:8
    X(:, i) = data.(vars{i});
end

% Add to KeyValueStore
add(intermKVStore, 'key', {X'*X, X'*y});
end
```
Reduce function

```
function reduceLinearRegression(intermKey, intermValIter, outKVStore)

% Initialize the sum
xx = 0;
Xy = 0;

% Compute the sum
while(hasnext(intermValIter))
    curr = getnext(intermValIter);
    XX = XX + curr{1};
    Xy = Xy + curr{2};
end

% Add to the output KeyValueStore
add(outKVStore, intermKey, XX\Xy);
end
```
Example 2: Linear regression

Running the job

```matlab
% Call the mapreduce job
outds = mapreduce(ds, @mapLinearRegression, @reduceLinearRegression);
wmr = readall(outds);
wmr = wmr{1,2}{1};

% Run a centralized regression
wcent = (abaloneInputs*abaloneInputs')\abaloneInputs*abaloneTargets';

% Check equality
assert(all(abs(wmr - wcent) < 10^-6));
```
Example 2: Linear regression

The result

Figure: Result of linear regression estimation on 30 selected training samples.
Example 2: Linear regression

Learn more (Hadoop)

1 - Process big data in MATLAB with mapreduce
2 - MATLAB mapreduce APIs
3 - Using mapreduce to fit a logistic regression model
4 - Getting started with mapreduce
5 - Running mapreduce on a Hadoop cluster
Example 2: Linear regression

Bibliography (MATLAB) I


