Lecture x: MATLAB - advanced use cases

Parallel computing with Matlab's toolbox

Heikki Apiola and Juha Kuortti

February 22, 2018

Aalto University
juha.kuortti@aalto.fi, heikki.apiola@aalto.fi

Parallel Computing Toolbox

General concepts

The PCT (Parallel Computing Toolbox) allows exploitation of multicore processors, and it supports the use of computer clusters and graphics processing units (GPUs).

Goals of parallel computing

- **Efficiency**: Gain remarkable speedup distributing a computational task between several computing units, which work simultaneously with parts of the task. (Decompose your task into independent parts that may or may not communicate with each other.)
- **Memory**: Solve larger problems than can be done in one computational unit.

Difficulties with parallel computing

- Computer architectures evolve $\Rightarrow$ programming techniques have to evolve as well.
- How the data has to be stored and shared between computational resources
- In some cases it may be difficult to achive satisfactory speed improvements, especially when there's need to communicate between different computational units.
- Writing and debugging parallel code is often difficult

What Matlab's PCT is good for

- Relatively easy to write parallel programs using just minor extensions of the language.
- **Limitations:** Special parallel computing languages provide more flexibility (but require more programming effort).
- Prototype parallel code by working on your local (at least two kernel) computer before running on a remote cluster.

Note: The cluster on which you run jobs must be running the `MATLAB` Distributed Computing Server, such as our *Triton*.

Parallel pool, workers, client

Terminology:

- *Client*: Matlab-session running on your laptop or remote computer (like Triton).
- *Workers* are MATLAB computational engine processes running on laptop or remote cluster. Workers don't have a MATLAB desktop, they can communicate between themselves and the client.
- *Pool*: A set of workers forms a parallel pool. A pool can be started with the command `>>parpool` and closed with `>>delete(gcp)`

Example: Open pool, doc parallel.Pool

```
>> p=parpool
Starting parallel pool (parpool) using the 'local' ...
    profile ...
connected to 2 workers.
p =
 Pool with properties:
           Connected: true
           NumWorkers: 2
              Cluster: local
        AttachedFiles: {}
    AutoAddClientPath: true
          IdleTimeout: 30 minutes (30 minutes remaining)
          SpmdEnabled: true
>> p.NumWorkers
ans =
     2
```

Controlling pool, profile

- `parpool` - Open parallel pool with default profile.
- `parpool(n)` - Open parallel pool with n workers.
- `delete(gcp)` - Close pool (deault idle time: 30 min).

Most of the parallel computing constructs (`parfor`, `spmd`,etc.) to be discussed, will automatically open a pool with default profile.

Note: The parallel profile can also be viewed and edited by opening the "Parallel"-tab at the bottom right part of the "home"-view of the desktop. For more, see the Mathworks' pages which lead us to our next topic:

https://se.mathworks.com/help/distcomp/spmd.html and https://se.mathworks.com/help/distcomp/run-code-on-parallel-pools.html

Parallel computing constructs

spmd,parfor,...

We will be concerned with

- `spmd` : Single program multiple data
- `parfor` : parallel for-loop
- `parfeval` (possibly) : parallel evaluation

The `spmd`-construct gives the user full control and understanding of the parallelization process. It allows the same code to be run on multiple workers with each worker using different data, which might for example be different parts of the same array. `parfor` is technically simplest, just replace `for` by `parfor`. MATLAB decides how to distribute the computation among the workers. Some limitations compared to `for`: nested loops are not allowed and some loop index limitations exist, etc.

Toolbox specific parallelization tools

Some toolboxes, like the *optimization toolbox* provide support for parallel computation, provided that the PCT is installed (like with us).

There the user just needs to turn a "parallel switch on", and observe the difference in performance.

For example, in the Optimization Toolbox there is a setting `'UseParallel',true` in the `options` structure passed to the solver. (Compare to ODE-solvers for a similar structure for controlling tolerances etc.)

Single program, multiple data – spmd

The `spmd` statement defines a block of code to be run simultaneously on multiple workers that should be reserved using `parpool`.

The first couple of slides contain quotations of the text in:

https://se.mathworks.com/help/distcomp/spmd.html

The general form of an spmd (single program, multiple data) statement is:

```
% nlabs=2 % on laptop
% parpool(nlabs)   % Open pool if not open already
spmd
    statements
end
```

Open pool, `parpool,labindex,myPool.NumWorkers`

MATLAB executes the spmd body statements on `nlabs` workers simultaneously.

First open a pool of MATLAB workers using `parpool` or have your parallel prefences allow the automatic start of a pool.

Inside the body of the spmd statement, each MATLAB worker has a unique value of `labindex`.

Opening the pool by `>>myPool=parpool` uses the default number of workers stored in `myPool.NumWorkers`.
Alternatively one can start by
`>>nlabs=16; parpool(nlabs)`
in case there are `nlabs` workers available in the pool.

Communication functions such as `labSend` and `labReceive` can transfer data between the workers.

Examples of spmd

```
nlabs=2;        % laptop
% nlabs=24;     % Triton (default 24)
parpool(nlabs) % Opening the pool takes some time.
spmd
  % build magic squares in parallel
  q = magic(labindex + 2);
end
```

The variable `q` is a Composite object. The value `q{k}` is the value stored in the k^{th} worker, whereas `q(k)` is the cell array of values stored in the k^{th} worker.

Note: The indexing of a composite object is exactly similar to that of a *cell array*.

```
for ii=1:length(q);figure,imagesc(q{ii});end
```

Composite objects

Note: As the last line showed us, the composite object `q` remains available even outside the `spmd`-block as long as the pool is open.

```
>> q
q =
   Lab 1: class = double, size = [3  3]
   Lab 2: class = double, size = [4  4]

>> q{1:nlabs}
ans =
     8     1     6
     3     5     7
     4     9     2
ans =
    16     2     3    13
     5    11    10     8
     9     7     6    12
```

Composite objects to cell arrays, results from workers to client

As noted above, the variable `q` "behaves" like a cell array. After closing the pool it isn't available (the workers are gone). Here's how to store the contents of q "permanently" into the client:

```
Q=cell(1,nlabs);          % Create a cell array Q.
Q(1:nlabs)=q(1:nlabs);    % Copy q into cell array Q.
% Q=q;                    % Not allowed.
delete(gcp)    % q no more available, Q remains.
cellplot(Q), figure
surf(Q{nlabs})  % Plot (surf) the last (largest) magic.
```

Run the same script in Triton

```
slogin -X -lscip triton.aalto.fi
Mat...xLab...y2018
$ cd $WRKDIR
$ mkdir USER_OWN_DIR
module load matlab
```

```
>>parpool
  Pool with properties:
            Connected: true
            NumWorkers: 24
            Cluster: local
            ...
>>spmd
>> q = magic(labindex + 2);
>>end
```

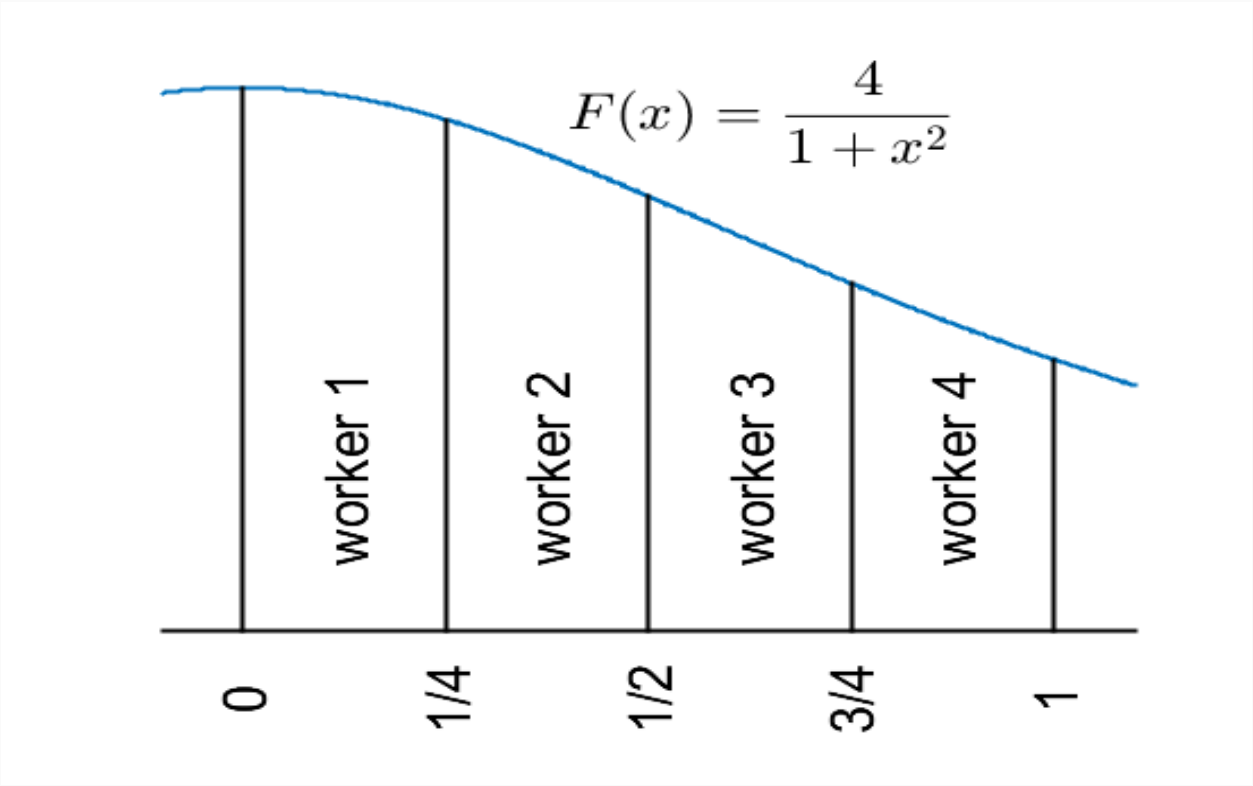
```
>> q(1:6:24)
ans =
  1×4 cell array
   {3×3 double}{9×9 double}{15×15 double}{21×21 double}
>> surf(q{24})
```

Note: The "protype code" (on laptop) and the remote computer code (on Triton) need no change, the latter, using Triton default uses 24 workers.

```
>> Q=cell(1,24);
>> Q(1:24)=q(1:24);
>> delete(gcp)   % close pool
                 % q is gone, Q remains.
```

Spmd-example: Numerical integration

Task: Integrate numerically: $\int_0^1 \frac{4}{1+x^2} dx$,
decomposing the interval of integration "Labwise".



Define the variables a and b on all the workers, but let their values depend on `labindex` so that the intervals $[a, b]$ correspond to the subintervals shown in the figure.

Note: The code in the body of the `spmd` statement is executed in parallel on all the workers in the parallel pool.

```
p=parpool  % If not open. (Take a while.)
numlabs=p.NumWorkers
spmd
    a = (labindex - 1)/numlabs;
    b = labindex/numlabs;
    fprintf('Subinterval: [%-4g, %-4g]\n', a, b);
end
```

Let's work on this both on Laptop and Triton:

`spmd_example1_numint.m` (Link doesn't work here, use index.html)

```
p = parpool;  % Start parallel pool.
numlabs=p.NumWorkers
%%
spmd
    a = (labindex - 1)/numlabs;
    b = labindex/numlabs;
    fprintf('Subinterval: [%-4g, %-4g]\n', a, b);
end
```

More examples on spmd

- Find all local min/max and/or zeros of a univariate function. Exercise on this.
- Multivariate optimization, "domain decomposition"
- "Method of lines" for certain PDE's, semidiscretization, uses a large number of calls to ODE-solvers. (Matlab-demo exists.)
- "Parameter sweep" on differential equations, optimization and many others. (One example on diff, equ. included as a link.)

Some of the above may be well (or better) suited to `parfor`

Parallel for – parfor

The easiest way to use the parallel toolbox is to replace `for` by `parfor`. A `parfor`-loop splits the computation inside the loop among available workers in an automatic way, in an unspecified order.

Restrictions:

1. The loop variable must increase in steps 1.
2. No data dependencies between different iterations. Example:

```
s=ones(10,1);
parfor i=1:10,s(i)=i^2                  % Allowed
parfor i=2:10, s(i)=s(i-1)^2+i^2     % Not allowed
```

3. A `parfor`-loop can't contain another for-loop, but it can contain a call to a function that contains a parfor loop