Bi7740: Scientific computing

Parallel computing in MATLAB

Vlad Popovici
popovici@iba.muni.cz

Institute of Biostatistics and Analyses
Masaryk University, Brno

Before starting

Bibliography

- J. Kepner: *Parallel* MATLAB. SIAM 2009
- Mathworks: *Parallel Computing Toolbox. User's guide* (≥R2013a)

Please download the files from IS: `sc11-ex*.m`, `countprimes*.m`.

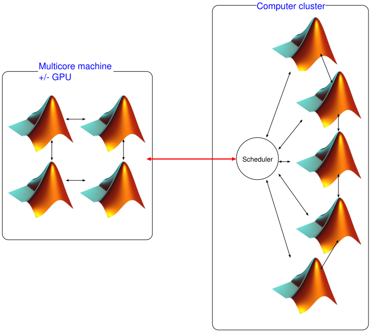
Outline

Introduction

- based around the Parallel Computing Toolbox and Distributed Computing Server
- can exploit different backends: multicore, cluster and GPUs
- requires commercial license - limits the number of workers
- low level matrix computing is multi-threaded since 2007 (e.g. LU-decomposition, etc)
- three ways to exploit parallelism:
  - `parfor`: for-loops executed in parallel
  - using `spmd` statement: single program multiple data
  - the `task` feature helps creating several independent programs
- still evolving, differences exist between 2013a, 2013b, 2014a versions

Overview

- locally: Parallel Computing Toolbox
- remotely: Distributed Computing Server
- lingo: a node the user uses as main entry point is called *client*; the other nodes are called *workers* or *labs*



- many functions and toolboxes were recoded to use the parallel computing infrastructure
- many/most of the functions from the standard toolboxes take a parameter `options` where an option for parallel computing can be set
  Example: for optimization functions, you can pass `'UseParallel'` option:

```
opts = optimset(...., 'UseParallel', 'Always');
[...] = fmincon(...., opts);
```

- toolboxes that use parallel computing: Statistics, Optimization, Computational Biology, Simulink, Image Processing, Signal Processing, etc
- the cluster should be used in *batch* mode, to avoid blocking the workers

Outline

Execution modes

Model	Command example	Where
interactive	`matlabpool`	local machine
indirect local	`batch`	local machine
indirect remote	`batch`	somewhere else

Outline

1. Introduction
2. Parallel execution modes
   - Parallel `for` loops = `parfor`
   - Distributed computing using `batch`
   - Single Program Multiple Data
3. Distributed data
4. Final remarks

parfor

- the simplest path to parallelism
- indicates a loop whose iterations are independent and which can be executed in parallel
- the iterations are automatically distributed to workers
- use `matlabpool` command to create/destroy a set (pool) of workers

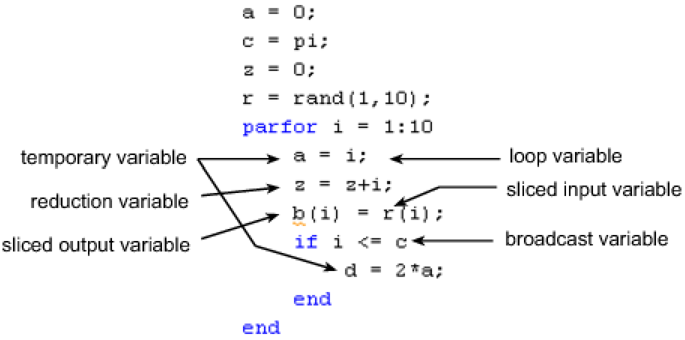
Consider computing the values of a vector (`for` is used for clarity, not efficiency!):

```
N = 1024;

a = zeros(1, N);
for i = 1:N
    a(i) = sin(i*2*pi/N);
end


plot(a);
```

```
N = 1024;
matlabpool open 12;
a = zeros(1, N);
parfor i=1:N
    a(i) = sin(i*2*pi/N);
end
% i is undefined after loop
matlabpool close;
plot(a);
```

Types of variables

Consider the block:



Source: Mathworks

- *loop variable* (`i`): assignments to the loop variable are forbidden

- *loop variable* (`i`): assignments to the loop variable are forbidden
- *sliced varible* (`b, r`): a variable that can be broken down in segments (slices) to be distributed to the workers. The variable is indexed by `[...]` or `.....` and does not change in shape during `parfor`. The index has of one of the forms `i, i+k, i-k, k+i, k-i` where `i` is the loop variable and `k` is a constant.

- *loop variable* (`i`): assignments to the loop variable are forbidden
- *sliced varible* (`b, r`): a variable that can be broken down in segments (slices) to be distributed to the workers. The variable is indexed by `[...]` or `.....` and does not change in shape during `parfor`. The index has of one of the forms `i, i+k, i-k, k+i, k-i` where `i` is the loop variable and `k` is a constant.
- *broadcast variable* (`c`): a variable (not loop or sliced) that is not changed within the loop and is distributed to the workers

- *loop variable* (`i`): assignments to the loop variable are forbidden
- *sliced varible* (`b, r`): a variable that can be broken down in segments (slices) to be distributed to the workers. The variable is indexed by `[...]` or `.....` and does not change in shape during `parfor`. The index has of one of the forms `i, i+k, i-k, k+i, k-i` where `i` is the loop variable and `k` is a constant.
- *broadcast variable* (`c`): a variable (not loop or sliced) that is not changed within the loop and is distributed to the workers
- *reduction variable* (`z`): the only exception to the independence of the iterations. Appears in constructions like `X = X operator something`

- *loop variable* (`i`): assignments to the loop variable are forbidden
- *sliced varible* (`b, r`): a variable that can be broken down in segments (slices) to be distributed to the workers. The variable is indexed by `[...]` or `.....` and does not change in shape during `parfor`. The index has of one of the forms `i, i+k, i-k, k+i, k-i` where `i` is the loop variable and `k` is a constant.
- *broadcast variable* (`c`): a variable (not loop or sliced) that is not changed within the loop and is distributed to the workers
- *reduction variable* (`z`): the only exception to the independence of the iterations. Appears in constructions like `X = X operator something`
- *temporary variable* (`a,d`): non-indexed assigned variable, but not a reduction; cleared before each iteration

Reduction variables

```
S = ...;
parfor i=1:N
  S = S + X(i);
end
```

```
S = ...;
S = S + X(1) + X(2) + ... +
 X(i) + X(i+1) + ... X(n);
```

- a simplified map-reduce parallelism
- the same reduction function/operator is applied at each iteration
- for non-commutative operators (e.g. $*$ or `[...]`) the reduction variable must always appear in the same position
- you can use an *associative* function: `S = f(S, expr)` or `S = f(expr, S)`
- note that floating point operators are not strictly associative (limited precision)

Example: the sieve of Eratosthenes

Find the number of prime numbers below *N*.

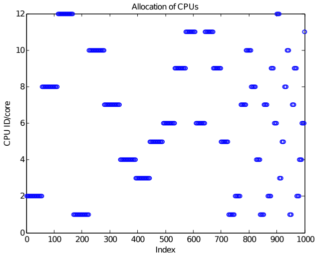
```
function np = countprimes(n)
np = 0;
for i = 2:n
  p = 1;
  for j = 2:(i-1)
    if mod(i, j) == 0
       p = 0;
    end
  end
  np = np + p;
end
return
```

```
function np = countprimes_p(n)
np = 0;
parfor i = 2:n
  p = 1;
  for j = 2:(i-1)
    if mod(i, j) == 0
       p = 0;
    end
  end
  np = np + p;
end
return
```

n	100	5000	10000	30000
T_1	0.00414	0.60726	2.34770	20.73521
T_{12}	* 0.09550	0.16312	0.34941	2.45712
Speed-up	0.0434	3.7228	6.7190	8.4388

- (*) this time was obtained after repeating the task - the first time (after creating the pool) running a parfor loop takes longer than usual: setting up all communications
- parallelized versions become efficient, once the overhead is negligible in comparison with the computation

Local resource allocation




```
ncores = 12;
matlabpool('open', 'local', ncores);
wi = zeros(1,1000);
parfor k=1:1000
    w = getCurrentWorker;
    wi(k) = get(w, 'ProcessId')
end
matlabpool('close');
```

```
pid = unique(wi);
core = zeros(1, 1000);
for k=1:ncores
    core(wi == pid(k)) = k;
end
plot(core, 'o'); title('Allocation of CPUs');
xlabel('Index'); ylabel('CPU ID/core')
```

Exercise

Implement the integral estimation by quadrature using, for example, the midpoint rule.

- write a function `quadint(f, n, a, b)` which estimates $\int_a^b f(x)dx$, by the approximation

$$\int_a^b f \approx \sum_{i=1}^{n} hf(x_i)$$

where $h = (b-a)/(n-1)$ and $x_i = ((n-1)a + (i-1)b)/(n-1)$

- implement both the serial and parallel version

Try

```
f = @(x) (sqrt(16 - x .^2)/8);

tic;
quadint(f, -4, 4, 100000)
toc
```

for serial and parallel versions.

Outline

batch system

- you can pass either a *function* or a *script* to be executed on a worker
- use several calls to `batch` to have more jobs run in parallel
- to *synchronize* use `wait()` function
- to retrieve data from the worker, use **`load`**`()`
- at the end, delete the job with **`delete`**`()`

batch command

```
j = batch('script')    % no '.m' in the script name!
j = batch(clstObj, 'script')
j = batch(fcn, N, {x1, ..., xn})
j = batch(clstObj, fcn, N, {x1, ..., xn})
j = batch(..., 'p1', 'v1', ...)
```

General structure for batches

```
% get an object describing the cluster
clst = parcluster('local'); % default profile
...
...
% dispatch the jobs and save their IDs:
jb = batch(....)

% wait for completion:
wait(jb);

% fetch the results:
load(jb);
% or
r = fetchOutputs(jb);
```

Integral approximation with `batch`

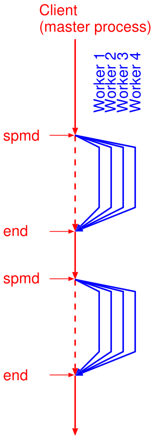
`sc11-ex05.m`

Outline

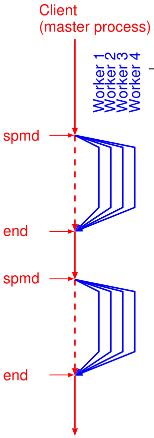
SPMD

```
spmd
    ...code block...
end
```

- similar to MPI (but simpler!): one master (called *client*) and several *workers* (sometimes called *labs*)
- the same code is applied to different data
- each worker inherits master's environment and adds his own (versions of) variables
- each worker knows its *identifier* (`labindex()`) and how many workers exist (`numlabs()`)
- synchronization points; communication via messages
- the client can inspect/alter variables on the workers



- the *code* is shared between client and workers
- the client executes till `spmd` and then it pauses
- the workers execute the code between `spmd` and **`end`**
- when workers reach **`end`** (for `spmd`) the client resumes execution
- the workers have read-only access to client's variables
- worker's environment is preserved between `spmd` blocks



	Client			Worker 1			Worker 2		
Code	a	b	c	d	e	f	d	e	f
`a = 1;`	1	.	.	.	.	.	.	.	.
`b = 2;`	1	2	.	.	.	.	.	.	.
`spmd`									
`d = numlabs();`	1	2	.	2	.	.	2	.	.
`e = a + labindex();`	1	2	.	2	2	.	2	3	.
`end`									
`c = b + e{2};`	1	2	5	2	2	.	2	3	.
`d{1} = 6;`	1	2	5	6	2	.	2	3	.
`spmd`									
`f = d * e;`	1	2	5	6	2	12	2	3	6
`end`									

Exercise

Write the code to compute the quadratic approximation of an integral using the `spmd` blocks.
Idea: use the property

$$\int_a^b f = \sum_{k=1}^{N} \int_{a_k}^{b_k} f$$

where $\{[a_k, b_k]\}$ is a partition of $[a, b]$. Divide the interval $[a, b]$ in subintervals $[a_k, b_k]$ on which compute the previous approximation given by `quadint(f, ak, bk, n)`.

Outline

Codistributed data

```
d = distributed(X);
d = distributed.cell(n,...);
d = distributed.eye(n,...);
d = distributed.zeros(n,...);
...
```

- create a distributed array and send slices to the workers where data will reside.
- for an array, the last dimension is used for distribution
- workers can get a local copy from another worker using `getLocalPart()`
- the client collects data using `d{i}` construct

Codistributed data, cont'd

- the technique allows creation of arrays that do not fit on a single machine
- the creation time is faster
- avoids need of communication
- the local parts are used to speed up access to data on other workers
- a distributed array can be copied by the client into a local array using `gather()`
- there are many functions and operators that automatically detect distributed data, so the code is uniform for all cases

Outline

Final remarks

- not everything is worth parallelizing - sometimes it may degrade the performance
- use `profile` to analyze your code
- you can use `pmode` for interactive parallel execution of commands. Example

```
pmode start 'local' 4
...
pmode exit
```

Good luck with your exams!