High Performance Embedded Systems

EEE4120F

Practical 2 – MATLAB Parallel Computing Toolkit

[40 Marks]

The focus of this task is on using the MATLAB Parallel Computing Toolkit (PCT). You need that activated in your MATLAB license of course, which is included in the UCT license. OCTAVE has a parallel computing package that has many of the same commands as found in the PCT, some are not exactly the same name.

This practical has two parts: Prac2.1, the first part, more links you to readings and a tutorial that is aimed to get you more familiar with the MATLAB PCT. Prac2.2, the second and more involved part of the prac involves you doing some coding, problem-solving and performance testing.

> *Notes:*
>
> *This Prac is intended to be done by a **team of two students**; however, if you prefer it can be done as an individual prac (but if you choose to do it on your own, be aware that there is no reduction to the work that needs to be done; hence you are encouraged to work as part of a team).*
>
> *Try to **use MATLAB for this practical**, but if you are having difficulties, e.g. access to MATLAB with pct installed, then you can attempt to use OCTAVE for this prac. But if using OCTAVE, then please indicate that you are using it as the start of your report. Info about OCTAVE parallel at: https://saturncloud.io/glossary/octave-parallel/*

What to hand in for Practical 2?

Hand in a short report (aim for about three pages: the page limit for this assignment is 4). Start your report with answers to the few Prac2.1 questions. Then in the next section of your report briefly describe the solution to the Prac 2.2 programming problem. Add speedup graphs for Prac2.2. The **callouts** show mark allocations for aspects to be submitted in your report.

Format your report as if it is an article (i.e. don't follow the same chronology as the prac-sheet – format it as "Introduction – Method – Results & Discussion – Conclusion"). In the "Method" section, theorise about what you expect and how you plan on testing said theory. In the "Results" section, confirm that you obtained what you expected (or explain why you obtained something unexpected).

> ***NB:*** Please hand in your report as per the due date set for Practical 2 in Amathuba.

Reminder: Calculating speed-up

Speed-up = T_{p1}/T_{p2}

Where T_{p1} = Run-time of original / non-optimal program

T_{p2} = Run-time of optimised program

Prac 2.1 / Tut: Learning about the Parallel Computing Toolkit

A super short start to the PCT. Read along these points:

The MATLAB Parallel Computing Toolbox (PCT) allows you to take advantage of multicore processors, GPUs, and computer clusters to speed-up your computations and handle large datasets efficiently. Here's some quick points to test and get started with the Parallel Computing Toolbox:

1. Installation:

Ensure you have MATLAB installed with the Parallel Computing Toolbox add-on (you may need to install this toolbox separately).

2. Basic Concepts surrounding the PCT:

As you should know, parallel computing is about breaking down a problem into smaller parts that can be solved simultaneously. Not always a trivial thing to do… if it is trivial, than we often call it embarrassingly parallel.

The term "Parallel Workers" is used in the MATLAB manuals to refer to the computational engines (or threads, although technically not always threads) that execute code in parallel.

3. Getting Started:

First you need to start a Parallel Pool:

```
parpool('local', N);
% This command starts a parallel pool with N workers on your local machine.
```

Check a parallel pool is available or was created successfully:

```
poolobj = gcp('nocreate');   % nocreate basically uses whatever pool is currently available
 % if there is no pool already available then no pool is created and poolobj is empty
% you could improve the code with this, basically if there is no existing parallel pool, then
% the next and simplest thing, is asked for a default pool to be created locally
if isempty(poolobj)
   parpool('local', N);
end
```

4. Some Parallelizing Coding

Example parfor Loop:

You can use the *parfor* command instead of a for loop to execute loop iterations in parallel… this is a rather smart command in that it analyses the look body and does a whole lot of data dependency checks and the like to try and implement a solution that will deliver the required result without causing problems such as breaking data dependencies, overwriting input data mid-way in processing, etc. Here is the general structure for a parallelized for loop:

```
parfor i = 1:N
    % Parallelizable computation to run
    %  (Note that it automatically makes it parallel and
    %   you cannot actually ask which thread you are in)
end
```

Example spmd block:

The term SPMD stands for 'single process multiple data', and more specifically this means the same code is run by different workers (or threads) but on different data. That is coincidentally the usual method of processing on done on a GPU, all the many cores are running in lockstep through the same instructions but applied to different data.

The syntax for the spmd block is simply:

```
spmd   % start of a block of code to be parallelized
  % put parallelized computation to be done here
end   % indicates end of the parallel block and back to the master thread
```

You can use the labIndex *within* the spmd block to ask which thread number you are in, if you get 1 returned then you are in the master thread (i.e. the one that launched the others and itself will do some of the work).

Here is a simple example of using spmd:

```
c = parcluster;
p = c.parpool(4);
% should say: Starting parallel pool (parpool) using the 'Processes' profile ...
% Start single program, multiple data i.e. all threads run same thing on different data
%  This example just prints hello world and thread number for each thread
spmd;
  fprintf('Worker %d says Hello World!\n', labindex);
end
  % note: spmdIndex is favoured in Matlab 2022 and later, but not available in earlier versions
% at end of program you might want to do:
p.delete    % delete and shutdown the created pool
```

5. Handling Data:

Data can be distributed across worker threads and regrouped to the master thread.

Distributed Arrays:

These arrays are automatically distributed across multiple MATLAB workers, using command:

```
D = distributed(A);
%  A is the array you want split up among threads, and D is the distributed version, as in
%  D is a piece of A that is accessible by the worker. You would use D within spmd or parfor
```

Slicing Data:

Use use *gplus* or spmdPlus (this is the new name for the same function) which adds

and *gather* functions to collect results.

```
p = parpool('Processes',4);
spmd
   C = rand(3, labindex-1);  % or spmdIndex instead of labindex in newer matlab
end
C
delete p
```

Now if you type in C you can see how the C array looks, except that it is sitting in memory that is private to the different processors, i.e. the output display shows that Lab 1 (i.e. the first thread) has the first column, Lab 2 the second and so on. Remembering that the matrix index coordinate are in the form M(*row*,*column*).

```
>>  C
C =
  Lab 1: class = double, size = [3  0]
  Lab 2: class = double, size = [3  1]
  Lab 3: class = double, size = [3  2]
  Lab 4: class = double, size = [3  3]
```

You can use *gather* to collect all the data in the split matrix into a single consolidated one on the master thread, e.g.:

finalResult = gather(C);

further info available at:

https://www.mathworks.com/help/parallel-computing/gpuarray.gather.html

6. Essence of GPU Computing

MATLAB can utilize GPU for certain computations, and indeed might do so automatically for various functions you call, particularly if the size and type of processing of the data merits use of a GPU.

To see if your GPU is available to MATLAB working, call the gpuArray. If it is working and accessible you should get the output as shown below:

```
>> gpuArray
ans =
    []
```

i.e. it should just return an empty array, [], if it is working. That empty array is theoretically on the GPU (actually a pointer to null applicated there). But if it is not working, you will see an error indicating no GPU available or drivers not found. In which case, if you do indeed have a compatible GPU (e.g. a not too ancient nVidia card in your machine) then you may need to shutdown matlab, update your graphics drivers, reboot, restart matlab and hopefully it works.

The usuall approach to GPU use is first creating or loading data to starting with in your master thread (you might use other threads to help) and then use gpuArray to transfer the data to GPU:

```
gpuArray(A);   % This function transfers data to the GPU.
```

For more info on GPU use and transferring data to/from the GPU, see:

https://www.mathworks.com/help/parallel-computing/gpuarray.html

Here's the highlights summarized, go through it on your MATLAB and see if it works:

```
% Create an array X.
X = [1,2,3];
% Transfer X to the GPU.
G = gpuArray(X);
% Check that the data is on the GPU.
isgpuarray(G)   % should give ans = logical   1
% Calculate the element-wise square of the array G.
GSq = G.^2;   % basically tells GPU what it needs to do, must be PSMD operation
% Transfer the result GSq back to the CPU:
XSq = gather(GSq)    % this gathers all the data blocks, which may have been
                     % distributed between blocks of cores in 'block memory'
%  gives answer:  XSq = 1×3
%                       1    4    9
```

```
% Check that the data is not on the GPU.
isgpuarray(XSq)   % should give  ans = logical   0
```

And that's the main points of using a GPU in MATLAB, easy…

unless you want to do something more complicated, which you can do in a moment 😉

7. Now over to you for some Parallelizing Coding

Now that you know some of the most essential aspects of parallel programming in MATLAB, proceed with reading the official MATLAB "Quick Start Parallel Computing in MATLAB" tutorial, which is available at: https://www.mathworks.com/help/parallel-computing/quick-start-parallel-computing-in-matlab.html

While going through the quickstart tutorial, do have MATLAB open and try out the commands so that you become more familiar with them.

Prac2.1 question to answer in your report

2.1(a) If you are wanting to use the GPU, do you have to put the code that you want to be parallelized on the GPU into a *spmd* block? Explain why you do or do not need to do so. Provide a short example.

Answer in your report. [2 marks]

2.1(b) Generate an 100 x 100 matrix of integers with values between 1 and 10. You can do this using X=randi([1 10],100).

Use a spmd block, or a parfor (and appropriate use of labIndex) to parallelize counting the number of 1s in the matrix. See how fast you can get that done. *Hint:* You can trial different ways for the summing to be distributed and then gathered to a final sum.

Answer in your report. [8 marks]

Prac 2.2: Parallel Processing in MATLAB

Programming Topic: "Parallel Sorting and compare, , data reduction and stream preparation"

2.2.(a) This short task involves implementing the easy Bubble sort routine, just using looping, don't attempt any parallelization. First implement Bubble sort, in the regular fashion, by using a nested for loop. A good explanation is given at: https://medium.com/codex/bubble-sort-how-it-works-psuedocode-and-c-python-implementation-c45306d44827. Apply sorting to a 100x100 matrix of random numbers, you just use rand for generating random floats in the range (0,1). Sort the columns from minimum value to maximum value, like the MATLAB sort command does. i.e. you will need another for loop for go through sorting each column of the matrix one column at a time. (NOTE: obviously not using the built in sort function). *[10 marks]*

Include code, with some comments, in your report.
[10 marks].

2.2.(b) How does the built-in sort function compare in run time to your sequential sort implementation? Test, at least, for square matrixes of sizes 100, 200, 500, 1000, 10000. *[5 marks]*

Indicate any code used to test this, short a speedup graph.
[10 marks].

2.3.(c) Now for a bit of explicit parallelism (the MATLAB sort function tends to use implicit parallelism, i.e. you aren't so aware of it being a parallel function). Use a parfor or spmd block to parallelize sorting of the columns of X. Test at least matrices of size 100, 5000 *[15 marks]*

Indicate your code, and speedup graphs.
[10 marks].

* 2.3.(d) [Optional question, doesn't count marks but help understanding] Try using gpuArray and related functions to get the sorting done on the GPU, as well as the sorted array back to host memory that regular MATLAB functions can access.

Hint: you want to tic and toc to get some timing stats for this.

END OF PRACTICAL ASSIGNMENT