Practical 2 – MATLAB Parallel Computing Toolkit

Cammaran Clark and Kian Frassek
EEE4021 Class of 20
University of Cape Town
South Africa
[†]CLRCAM007 [‡]FRSKIA001

***Abstract*—**

I. Introduction

This report details an investigation into MATLAB's Parallel Computing Toolkit

A. Part 1 Question A

No, the spmd block does not necessarily parallelize code in the GPU. The spmd block or Run Single Programs on Multiple Data Sets is used to parallelize code across multiple workers in a parallel pool but not specifically the GPU. To use a GPU to process code you would use the matlab functions gpuArray() function garrayfun() will force the code to use the GPU.

B. Part 1 Question B

Before analysing the speed of the different parallelism methods, the parallelism function would have to be created. For this question, they would simply count the number of 1s in a 100x100 matrix of random integers from 1 to 10 created using

```
X = randi([1 10], 100);
```

A function was created for a single thread, an spmd and a parfor method which are shown below.

```
% Single Thread
tic;
count = 0;
for i = 1:100
    for j = 1:100
        if X(i,j) == 1
            count = count + 1;
        end
    end
end
count
toc
```

```
% SPMD Block
tic;
spmd
    myStart = (spmdIndex - 1) * 25 + 1;
    myEnd = myStart + 24;
    myCount = 0;
    for i = myStart:myEnd
        for j = 1:100
            if X(i,j) == 1
                myCount = myCount + 1;
            end
        end
    end
    % sum all counts into a single array
    counts = spmdPlus(myCount, 1);
end
% display total count
sum(counts{1})
toc
```

```
% Parfor
tic;
parfor i = 1:100
    for j = 1:100
        if X(i,j) == 1
            count = count + 1;
        end
    end
end
count
toc
```

The final results were that single thread was the fastest, the parfor second and the SPMD block was the slowest. However, with such a small data set, the overhead to create the parallel workers is likely what makes the parallel functions slower.

II. Methodology of Part 2

A. Question A

Create a bubble sort function that can sort an array of randomised values. The function will be used to sort the columns of a 100x100 matrix.

B. Question B

Using the bubble sort function created in the previous question, test it against the inbuilt MATLABsort function. The testing will be conducted on square matrices of sizes 100, 200, 500, 1000, 10000 and calculate the speed up from the user created function to MATLAB's inbuilt function. It is expected that the inbuilt function should be faster for all sizes, and the speed up should increase as the size goes up.

C. Question C

Using MATLABparallelism to create a new bubble sort function which using explicit parallelism using a 'parfor' or 'spmd' block The testing will be conducted on square matrices of sizes 100 and 5000 and calculate the speed up from the user created function to MATLAB's inbuilt function It is expected that the inbuilt function should be faster for all sizes, but the explicit parallelism bubble sort should be faster than the user created bubble sort without parallelism.

III. Results of Part 2

A. Question A

The bubble sort function was created based off of sudo code for a generalised bubble sort. The code is shown below

```
function array = BubbleSort(array)
 for i = 1:length(array)
  for j = length(array):-1:i+1
   if array(j) > array(j-1)
    temp = array(j);
    array(j) = array(j-1);
    array(j-1) = temp;
   end
  end
 end
return;
end
```

This code took 0.0087 seconds to sort the 100x100 matrix

B. *Question B*

The code to test the user function is shown below. 'timetaken' is a matrix for storing the times of both the user and inbuilt functions.

```
% Initialize array
testsizes = [100; 200; 500; 1000; 10000];%grid sizes tested
timetaken = cell(1+length(testsizes),4);
timetaken(1,:) = {"size", "bubble sort time", "inbuilt sort time", "speed up"};
timetaken(2:end,1) = num2cell(testsizes);

% Loop over each row in timetaken
for i = 2:size(timetaken,1)
 % Call the timetest function
 %with the value in the first column (grid size) of the i-th row
 [timeBubble, timeInbuilt] = timetest(timetaken{i,1});
 timetaken{i,2} = timeBubble;
 timetaken{i,3} = timeInbuilt;
 % Calculate speed up
 timetaken{i,4} = timeBubble/timeInbuilt;
end
disp(timetaken)
```

The function 'timetest' is given below. It takes a matrix size, creates a square matrix, and then outputs the sort time for both the user and inbuilt functions.

```
function [t_user,t_inbulit ] = timetest(size)

 X=rand(size, size);
 Y=X;

 tic
 for i = 1:size
  X(:,i) = BubbleSort(X(:,i));
 end
 t_user = toc();

 tic
 for i = 1:size
  Y(:,i) = sort(Y(:,i));
 end
 t_inbulit = toc();

 display("Time tacken by the bubble sort function was " + t_user " s.")
 display(" s. Time tacken by the inbuilt sort function was: "+  t_inbulit +" s.")

 return;
end
```

The results from the tests are shown in the table below

TABLE I
USER VS INBUILT SORT FUNCTION

Matrix size	User	Inbuilt	Speed up
100	0.0087	0.0002	40.507
200	0.0205	0.0003	62.791
500	0.1780	0.0012	145.18
1000	2.1324	0.0048	445.16
1000	1813.9	1.5631	1160.4

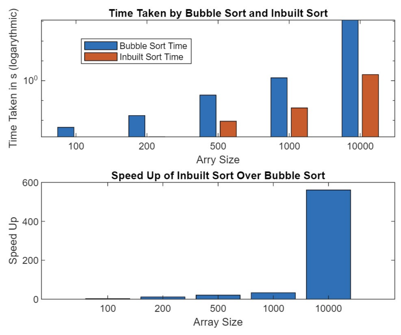

Fig. 1: User vs inbuilt sort function

These results are logical because the inbuilt sort function uses the Quick Sort algorithm. This means that the bubble sort function has a big O notation of $O(n^2)$, while the inbuilt sort function has a big O notation of $O(n\log(n))$ in most cases. This means that the inbuilt sort function will be faster than the bubble sort function for all sizes. However, the time difference will be more pronounced as the size of the matrix increases since the time taken by the BubbleSort algorithm increases at a faster rate than the inbuilt sort function.

C. *Question C*

The code to test the parallelism user function is shown below. 'timetaken' is a matrix for storing the times of both the user and inbuilt functions.

```
testsizes = [100, 5000];
timetaken_par = cell(1+length(testsizes),4);
timetaken_par(1,:) = {"size", "bubble sort time", "inbuilt sort time", "speed up"};
timetaken_par(2:end,1) = num2cell(testsizes);

% Loop over each row in timetaken_par
for i = 2:size(timetaken_par,1)
 % Call the timetest function with the value in the first column of the i-th row
 timeBubble = timebubble_parallelism(timetaken_par{i,1});
 timeInbuilt = timesort_parallelism(timetaken_par{i,1});
 timetaken_par{i,2} = timeBubble;
 timetaken_par{i,3} = timeInbuilt;
 % Calculate speed up
 timetaken_par{i,4} = timeBubble/timeInbuilt;
end

% Display the results
disp(timetaken_par)
```

The function 'timesort_parallelism' simply returns the time to sort a square matrix of a given size using the inbuilt sort function. The function 'timebubble_parallelism' is given below. It takes a matrix size, creates a square matrix, and then outputs the sort time for the user parallelism bubble-sort function.

```
function t_user = timebubble_parallelism(size)
    X=rand(size, size);
    tic
    spmd
        myStart = (spmdIndex - 1) * size/4 + 1;
        myEnd = myStart + size/4-1;
        for i = myStart:myEnd
            X(:,i) = BubbleSort(X(:,i));
        end
    end
    t_user = toc();

    display("Time tacken by the parallelism bubble sort function was " + t_user );
    return;
end
```

The results from the tests are shown in the table below

TABLE II

USER VS INBUILT SORT FUNCTION

Matrix size	parallelized BubbleSort	Inbuilt	Speed up
100	0.7642	0.001	764.96
5000	111.7174	1.0715	104.26

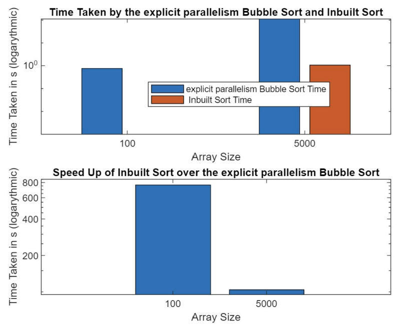

Fig. 2: User parallelism vs inbuilt sort function

The results show that the inbuilt sort function is faster than the parallelism bubble sort function. This is because when the parallelize BubbleSort run on a nxn matrix the big O notation is $O(n(n^2/p))$ where p is the number of workers in the parallel pool. The inbuilt sort function has a big O notation of $O(n(n* log(n)))$.(There is an extra n because every column need to be sorted).

It makes sense for the speed up to slow down as the size of the matrix increases form 100 to 5000 as building the SPMD function has an overhead. This overhead will have a more noticeable effect lower-sized arrays.

IV. CONCLUSION

In conclusion, while parallel function has significant benefits will very large data sets, the overhead required makes it inefficient to use on smaller data sets. It should also be noted that the inbuilt sort functions are faster than user created functions as they are highly optimised by the developers.