Investigating Performance of Parallel Counting and Sorting Algorithms on Dense Vector Spaces In MATLAB

Karan Abraham [†]and Bonga Njamela [‡]
EEE4120F 2024
University of Cape Town
South Africa
[§]ABRKAR004 [¶]NJMLUN002

***Abstract*—An experiment was conducted to investigate the performance of the high-level constructs provided in the MATLAB Parallel Computing Toolkit, such as `parfor` loop and the `gpuArray`.**

I. Introduction

Parallel computing involves dividing tasks into smaller processes that can be solved simultaneously. The MATLAB Parallel Computing Toolkit allows users to solve problems that require intensive computational and data resources by exploiting parallelism and multi-threading capabilities of processors and GPUs. This experiment leveraged functions and loops in the toolkit such as `parpool` which can initialise a pool of worker threads with one worker per physical CPU core [1]. The built-in multi-threading feature in MATLAB automatically prepares the models and blocks such that they can be executed on multiple threads. On the other hand, the Parallel Computing Toolbox has multiple explicit parallelism mechanisms that are facilitated by a parallel process-based pool or a thread-based parallel pool [2]. This enables the usage of additional hardware resources through threads and processes to improve the performance of a program.

An experiment investigated the improvements in performance when implementing thread-based pools that employed a parallel-for-loop (`parfor`), and a parallel execution of a single program with multiple data (`spmd`). The `parfor` loop was used performing counting and comparing operations on a matrix. Speedup graphs were plotted to evaluate the improvement in the execution time of the parallel program when compared to a sequential counterpart that used the standard `for` loop.

Parallel computing can also be performed on a GPU. MATLAB can accelerate linear algebra operations implicitly by running it on a GPU, provided that two criteria are met. The first criterion is that the time spent on the computation must exceed the time spent on transferring data to and from the GPU memory. Secondly, implicit GPU acceleration requires that computations to be able to be divided into smaller units of work [2]. The Parallel Computing Toolbox includes a special array type that can be initialised using the command `gpuArray` which relocates data from the MATLAB workspace to the GPU memory. A program is executed on the GPU whenever a function call contains at least one `gpuArray` as an input argument and the data is stored on the GPU [3]. While both a GPU and the `spmd` function can apply the same process on multiple data, `spmd` uses multiple workers from a parallel pool whereas programs that do not have a `gpuArray` do not execute on the GPU unless they fulfill the above criteria. However, if a machine has several GPUs, MATLAB assigns a different GPU to each worker by default or a user can initialise a parallel pool with as many workers as available GPUs.

II. Methodology

A. Hardware

The MATLAB programs were executed on a Linux machine equipped with an 11th Gen Intel(R) Core(TM) i7-1165G7 CPU with a clock speed of $2.80\,\mathrm{GHz}$, featuring 8 online processors and 4 core sockets per processor. The CPU's architecture, which includes 2 floating-point units (FPUs) per core, reduced the influence of hardware on MATLAB program performance by mitigating performance bottlenecks arising from shared FPUs across multiple cores. The machine provided 16 GB of RAM and 932 GB of SSD storage, ensuring sufficient memory capacity and disk space for executing the programs without thrashing or significant disk speed impacts. It is also noted that machine included an Intel TigerLake-LP GT2 [Iris Xe Graphics] GPU which cannot be used to run parallel programs by the MATLAB Parallel Computing Toolkit.

B. Implementation

The experiment investigated the benefits and trade-offs parallelizing processes using the MATLAB PCT and compared the performance of functions that implement multi-threading to their serial counterparts. The processes that were used to investigate gains in performance were embarrassingly parallel and involved reading and manipulating data stored in a matrix.

1) Parallel Summing and Comparing Elements of a Matrix: In the first part of the experiment, a MATLAB program for processing the total number of 1s in a 100x100 matrix, X, that is populated by random numbers between 1 and 10, was parallelised implicitly and explicity using the built-in `sum` function and `parfor` loop, respectively. The algorithm was chosen because it contains a comparison operation that makes the program time complexity $\mathcal{O}(n)$. This time complexity means that the amount of time required to perform the operation increases linear as the size of the matrix increases.

Implicitly, MATLAB uses its multi-threading feature to automatically parallelize computations using underlying libraries that breakdown and perform tasks on different cores and combine the result at the end. Implicit parallelization allows users to take advantage of multicore hardware capabilities without the complexity of writing parallel algorithms. For example, the parallelization of the summing algorithm was implicitly achieved using a single line as shown in the code below.

Listing 1: Performs the addition of all the ones in a matrix. MATLAB automatically parallelized linear algebra computations.

```
    sum(X(:) == 1);
```

The MATLAB PCT allows users to implement the parallel computing capabilities of a multicore machine explicitly by creating a parallel pool of workers. The experiment used the `parfor` loop to execute iterations in a parallel as shown below.

Listing 2: MATLAB code to perform a parallel sum of elements in a matrix that are equal to 1 using a `parfor` loop.

```
X = randi([1 10], 100); % create 100x100 matrix range [1-10]
num_1s = 0; % counter for ones
tic % start timer
parfor i = 1:numel(X) % parallel for loop through matrix
    if X(i) == 1
        num_1s = num_1s + 1; % sum and combine on different workers
    end
end
parTime = toc; % execution time
```

MATLAB uses all the available workers in a parallel pool. If there are zero available workers, the loop is run deterministically, but not in parallel [4]. The experiment used `parpool`, or parallel pool, of four workers, which is the default number of workers in a pool. Although `parfor` executes loop iterations in a nondeterministic order, the toolkit and underlying libraries ensure that there is no bad interleaving of threads and that there are no race conditions.

The execution time of each program was recorded and stored in the `parTime` variable to compare their speed. Due to the reduced overhead of parallelizing the code, the built-in `sum` function was expected to be parallelized more easily. The built-in sum function was also expected to have a shorter execution time than the program that uses a `parfor` loop because the built-in function was expected to implement multi-threading more efficiently.

2) Parallel Sorting: A serial program called `bubbleSort` was used as a golden measure to benchmark the built-in `sort` function for sorting the column entries of a matrix and a parallelized version of the *bubble sort* algorithm called `spmdBubbleSort`. The bubble sort algorithm iteratively compares each element in a the columns of an $m \times n$ matrix and swaps the items so that the columns of the returned matrix are arranged in ascending order by traversing columns from top to bottom. The serial version of the code is shown in Listing 3 below where nested `for` loops were used to sequentially sort elements of a matrix. Nesting the loops so that for each column, the program goes through all possible pairs of elements, means that the worst case time complexity of the algorithm is $\mathcal{O}(n^3)$, where n corresponds to the number of elements in each column.

Listing 3: Serial implementation of the *bubble sort* algorithm uses nested `for` loops to compare and swap the elements of a matrix. The program returns a matrix with columns that are sorted in ascending order from top to bottom.

```
function sorted_matrix = bubbleSort(matrix)
    % Loop through each column of the matrix
    num_cols = size(matrix,2);
    num_rows = size(matrix,1);
    for col = 1:num_cols
        % Apply bubble sort to the current column
        for i = 1:num_rows-1
            for j = 1:num_rows-i
                if matrix(j, col) > matrix(j+1, col)
                    % Swap elements
                    temp = matrix(j, col);
                    matrix(j, col) = matrix(j+1, col);
                    matrix(j+1, col) = temp;
                end
            end
        end
    end

    % Assign the sorted matrix to the output variable
    sorted_matrix = matrix;
end
```

The execution time of the serial bubble sort application was compared to execution time of the program that arranges the elements in a column using the built-in `sort` function. This function was called inside a `for` loop that iterates over N columns of the matrix and sorts each one, as shown in Listing 4. Execution times corresponding to the first and second run of the program were removed to account for cache warming effects.

Listing 4: The `builtinSort` program was used to sort N columns of a matrix using MATLAB's multi-threading feature to implicitly parallelize the code.

```
    function sorted_matrix = builtinSort(matrix)
    N = size(matrix, 2);%number of columns
    for col_j = 1:N
        %sort each column
        matrix(:,col_j) = sort(matrix(:,col_j));
    end
    sorted_matrix = matrix;
end
```

The `sort` function is similar to the `sum` function in that given a data set whose size exceeds a certain threshold, MATLAB implements multi-threading to accelerate computations. Since the applications used $N \geq 10^4$ data points, `builtinSort` was expected to be implicitly parallized and accelerated so that its average execution time was shorter than the execution time of the serial program.

The listing below shows how the `spmdBubbleSort` program used the MATLAB `spmd` statement, which executes the same operation on multiple data using several MATLAB workers simultaneously, to perform the bubble sort algorithm [5]. A parallel pool of four workers was created on the local machine using `parpool`. Since the parallel pool was created on the local machine, each worker was expected to correspond to one of 4 physical CPU cores out of the 8 total cores on the machine and was expected to use one computational thread [6]. The data stream was prepared by dividing an $n \times n$ (square) matrix equally among the available workers in the parallel pool. Therefore, the value of `numWorkers` in the code was expected to be 4 such that a 100x100 matrix would be divided into 4 100x25 sub-matrices, each stored and accessed privately in the matrix `m_sliced` by an individual worker. The application recombines the sorted sub-matrices from each worker using the `spmdCat` function.

Listing 5: The `spmdBubbleSort` program used the illustrated code to parallelize the bubble sort algorithm in the MATLAB `spmd` statement to perform sorting on different workers in the parallel pool.

```
function sorted_matrix = spmdBubbleSort(matrix)
    p = parpool('local',4);
    N = size(matrix, 2); %number of columns
    numWorkers = p.NumWorkers; %number of available workers

    spmd
          % select the index of the column to start sorting from
          startColumn = ceil(N/numW)*(labindex-1) + 1;
          % select the last index of the column we want to sort
          endColumn = startColumn + ceil(N/numW);

          % case where the matrix has an odd number of columns, last index
          % may be off slightly due to rounding
          if labindex == 4 && endColumn ~= N
              endColumn = N;
          end

          % slice out the appropriate number of columns to sort
          m_sliced = matrix(:,startColumn:endColumn); %divide matrix into 4 parts
          N_s = size(m_sliced,2);%number of columns in sliced portion
          M_s = size(m_sliced,1);%number of rows in sliced portion
          for col_j = 1:N_s %for each column
             for k = 1:M_s-1 %for each row in each column
                 for l = 1:M_s-k %for each item
                    if m_sliced(l, col_j) > m_sliced(l+1,col_j)
                     %Swap items
                     temp = m_sliced(l,col_j);
                     m_sliced(l,col_j) = m_sliced(l+1, col_j);
                     m_sliced(l+1, col_j) = temp;
                    end
                 end
             end
          end
          sorted_matrix = spmdCat({m_sliced},1,1);
%           disp(m_sliced);
    end
    p.delete; % shutdown parallel pool
end
```

The speedup of the `builtinSort` program was expected to be greater than the speedup of parallelization using `spmd` in the `spmdBubbleSort` program. This is because the overhead of preparing the data for the workers in the `spmd` parallel pool was expected to increase the execution time of the `spmdBubbleSort` application. Speedup graphs were compiled for different sizes of an input square matrix to determine if the speedup increases as the number of data points increased.

III. Results and Discussion

1) Parallel Summing and Comparing Matrix Elements:

2) Parallel Sorting: The following table shows the results comparing the average execution time of the applications that count the number of 1s in a 100x100 matrix. The table shows that the average execution time for the program that made a call to the built-in `sum` function was significantly shorter than the execution time of the counting program which uses the `parfor` loop to parallelize the process. Results also show that the execution time of the program that implements the `parfor` was significantly long for the first and second runs of the program. This was attributed to the overhead of parallelization, which was expected to lead to the longer execution times exhibited in the results. Values arising from the overhead of parallelization were removed from calculations of averages to account for cache warming effects.

Run	Built-In `sum` Time [ms]	`parfor` Sum Time [ms]
1	0.651	113.767
2	0.125	125.942
3	0.113	31.84
4	0.135	26.013
5	0.472	27.501

TABLE I: Table comparing the execution times for the different implementations of summing elements of a matrix in parallel. The built-in `sum` was significantly faster than the `parfor` loop method of parallelization. The first two values were removed when calculating average execution times in order to account for cache warming effects.

The average execution time of the built-in `sort` function, after running the program 30 times, was 77.18×10^{-3}ms. The average execution time of the method that used the `parfor` loop to count 1s through the matrix on different workers was 37.37 ms.

3) Parallel Sorting: The graph in Figure **??** shows the average speedup of the bubble sort algorithm implemented in parallel in the `builtinSort` program.

IV. Conclusion

References

[1] MathWorks. `parpool`: Creating parallel pool on cluster. [Online]. Available: https://www.mathworks.com/help/parallel-computing/parpool.html

[2] MathWorks. Run matlab on multicore and multiprocessor machines. [Online]. Available: https://www.mathworks.com/discovery/matlab-multicore.html

[3] MathWorks. Run matlab functions on a gpu. [Online]. Available: https://www.mathworks.com/help/parallel-computing/run-matlab-functions-on-a-gpu.html

[4] MathWorks. `parfor`: Execute `for`-loop in parallel on workers. [Online]. Available: https://www.mathworks.com/help/parallel-computing/parfor.html

[5] MathWorks. `spmd`: Execute code in parallel on workers of parallel pool. [Online]. Available: https://www.mathworks.com/help/parallel-computing/spmd.html

[6] MathWorks. What is parallel computing? [Online]. Available: https://www.mathworks.com/help/parallel-computing/what-is-parallel-computing.html