EPNM project 2.7

By Lars Vonk 336028

Introduction

In the second project of this semester we will dive deeper into the many ways to parallelise computation. In this project a large matrix is generated with the size of 25070 rows and 1000 columns and the following formulas will be used to challenge the computer and be able to see what effects multiple ways of parallelising this computation will have:

$$\sin(\ln(c7 + 2.7))^T \cdot \ln(\cos(c7) + 1.7)$$

Figure 2: Formula 1

$$\sin(\ln(c7 + 2.7)) \cdot \ln(\cos(c7) + 1.7)^T$$

Figure 1: Formula 2

The tasks

The tasks describe 9 different ways to complete the calculation over the whole matrix per formula, these are:

- A: Using basic matlab matrix operations
- B: Using basic matlab matrix operations but limited to a single thread
- C: Using a basic for loop to do all the operations on each resulting cell, still limited to a single thread
- D: Using a basic for loop to do all the operations on each resulting cell but all threads are used again
- E: Using the parallel for loop defined by the Parallel computing toolbox in matlab to do all the operations on each resulting cell
- F1: Manually splitting the rows into evenly sized chunks for each worker to calculate in parallel using the same parallel for loop as mentioned in the bulletpoint before this and using the basic matlab matrix operations in each worker
- F2: Manually splitting the rows into evenly sized chunks for each worker to calculate in parallel using the same parallel for loop as mentioned in the bulletpoint before this and using a for loop to do all the operations on each resulting cell
- G1: Automatically splitting using the distributed function from the parallel computing toolbox and then running the calculations over multiple threads using matrix operations
- G2: Automatically splitting using the distributed function from the parallel computing toolbox and then running the calculations over multiple threads using for loop

Using this wide range of possiblities give a wide range of results in time it takes to compute the total difference.

Methodology

I have followed the instructions closely. There are however some notes to make with this project.

1. I sadly did not succeed in making use of the distributed arrays using a for loop, my code seemed to get stuck eternally even after leaving it on for 2 hours there was no progress.
2. My matlab application crashes when performing the operations of step 4 of the assignment (moving the transposition from the first to the second matrix) because the amount of calculations that have to be done per iteration are too big to be stored by a single thread in memory. I have experimented with different sizes and the biggest size possible where this error does not occur seems to be a length of 10.000 (the non reduced size would be 25.070).

Aside from this I have used a variable called “reduction” to test my program and when everything was operating as intended I ran the program over the full sized matrix. When doing this however I stumbled onto the second problem described above (2. My matlab…).

Since I realised that the aim of the second task is to see how the system the speeds change when the rows are the smaller number (1000 initially) and the columns the larger number (25070). I also decided to not only run the program with reducing the rows of the second assignment (assignment 4) but also to reduce the whole set by 0.5 and see how the times would differ per run to also compare the first and second formulas. This program lives in the “mainscript_reduced.m” file.

Code

In this chapter I will describe how I solved all the assignments

2a

For 2a I just used the basic logic from matlab to run everything

```
function result = calculate(matA)
    result = sin(log(matA + 2.7))' * log(cos(matA) + 1.7);
end
```

2b

For 2b I did the same as 2a but I ran the following line before calling the function

```
% Single thread
maxNumCompThreads(1);
```

2c

For 2c I closely followed the knowledge I have about doing matrix multiplication and since the first matrix was getting transposed and multiplied by the second it was basically like doing n by n calculations where in every column all values were multplied and added. This code resulted in the following (still in single threaded mode)

```
function result = calculate_for(inputMat)
    % Preset a matrix with zeroes so memory is only allocated once
    result_size = size(inputMat, 2);
    tempResult = zeros(result_size, result_size);

    for i = 1:result_size
        for j = 1:result_size
            iterResult = 0;

            % Calculate for each cell in the matrix
            for k = 1:size(inputMat, 1)
                integerA = inputMat(k, i);
                integerB = inputMat(k, j);

                % Get result of the dot product of items
                iterResult = iterResult + (sin(log(integerA + 2.7)) * log(cos(integerB) + 1.7));
            end

            tempResult(i, j) = iterResult;
        end
    end

    result = tempResult;
end
```

2d

For 2d I called the function created for 2c again but I activated all the threads again by running the line

```
% Detect threads
maxNumCompThreads(MNCT);
```

2e

For 2e I did the same as 2c but I added a parallel for loop, because the parfor loop cannot use two indices to access data since it is blocking the user from having the risk that the parfor loop accesses the same data concurrently I added a temporary vector to store the results and add it after each row was calculated.

```
function result = calculate_parfor(inputMat)
    % Preset a matrix with zeroes so memory is only allocated once
    result_size = size(inputMat, 2);
    tempResult = zeros(result_size, result_size);

    parfor i = 1:result_size
        % Since in a parfor loop only the iterator can be used to access store values in vector
        iterVec = zeros(result_size, 1);

        for j = 1:result_size
            iterResult = 0

            % Calculate for each cell in the matrix
            for k = 1:size(inputMat, 1)
                integerA = inputMat(k, i);
                integerB = inputMat(k, j);

                % Get result of the dot product of items
                iterResult = iterResult + (sin(log(integerA + 2.7)) * log(cos(integerB) + 1.7));
            end

            iterVec(j) = iterResult;
        end

        % Save vector to row
        tempResult(i, :) = iterVec;
    end

    result = tempResult;
end
```

2f

For 2f since I used a lot of code splitting (moving everything into functions) I could very simply just split the data over the number of workers and then using a parfor loop calling this function get the result so for f1 and f2 I did the following

```
function result = calculate_split(inputMat, numWorkers)
    % Define sizes
    rowSize = size(inputMat, 1);
    resultSize = size(inputMat, 2);

    % Divide all the rows by the amount of workers available
    workerParts = rowSize / numWorkers;

    % Define all ends of ranges for workers
    workedRngEnd = ones(1, numWorkers + 1);
    for i = 1:numWorkers
        if i == numWorkers
            workedRngEnd(1, i+1) = rowSize;
            break;
        end

        workedRngEnd(1, i+1) = workerParts * i;
    end

    % Use cell since matrix will stack multiple matrices on top of each other which is not good
    workerResults = {};

    % Parallel iterate over all workers to do calculations
    parfor i = 1:numWorkers
        % Get rows that belong to worker
        part = inputMat(workedRngEnd(i):workedRngEnd(i+1), :);
        % Get result from rows
        workerResults{i} = calculate(part);
    end

    % Allocate result matrix
    result = zeros(resultSize, resultSize);

    % Iterate over all workers to combine results
    for i = 1:numWorkers
        % Add all results together
        result = result + workerResults{i};
    end
end
```

Figure 3: f1

```
function result = calculate_for_split(inputMat, numWorkers)
    % Define sizes
    rowSize = size(inputMat, 1);
    resultSize = size(inputMat, 2);

    % Divide all the rows by the amount of workers available
    workerParts = rowSize / numWorkers;

    % Define all ends of ranges for workers
    workedRngEnd = ones(1, numWorkers + 1);
    for i = 1:numWorkers
        if i == numWorkers
            workedRngEnd(1, i+1) = rowSize;
            break;
        end

        workedRngEnd(1, i+1) = workerParts * i;
    end

    % Use cell since matrix will stack multiple matrices on top of each other which is not good
    workerResults = {};

    % Parallel iterate over all workers to do calculations
    parfor i = 1:numWorkers
        % Get rows that belong to worker
        part = inputMat(workedRngEnd(i):workedRngEnd(i+1), :);
        % Get result from rows
        workerResults{i} = calculate_for(part);
    end

    % Allocate result matrix
    result = zeros(resultSize, resultSize);

    % Iterate over all workers to combine results
    for i = 1:numWorkers
        % Add all results together
        result = result + workerResults{i};
    end
end
```

Figure 4: f2

2g

For 2g I did the same as in 2f but I used the distributed function provided by the parallel computing toolbox

```
function result = calculate_split_distributed(inputMat)
    % Split the input matrix over workers using the distributed function
    inputMatPart = distributed(inputMat);

    % Parallel iterate over all workers to do calculations
    spmd
        % Get result from rows
        workerResults = calculate(inputMatPart);
    end

    result = gather(workerResults);
end
```

Sadly however I ran into errors when trying to use the for loop and I was not able to resolve this.

3

For assignment 3 I took the result from the B assignment (2b and 4b) and compared them using the norm

```
% Function definitions
function record_results(tasknum, time, result, printlabel, filename)
    global times;
    global norms;
    global diffs;
    global cmp_result;

    if cmp_result == 0
        cmp_result = result;
    end

    % Truncate the result to 1000x1000 or it gets stuck forever because memory problem
    if(size(cmp_result, 1) > 1000)
        cmp_result = cmp_result(1:1000, 1:1000);
    end

    % Truncate the result to 1000x1000 or it gets stuck forever because memory problem
    if(size(result, 1) > 1000)
        result = result(1:1000, 1:1000);
    end

    normVal = norm(result);
    diffVal = norm(cmp_result - result);

    fprintf('[%s] Time: %.12f\n', printlabel, time);
    fprintf('[%s] Norm: %.12f\n', printlabel, normVal);
    fprintf('[%s] Diff: %.12f\n', printlabel, diffVal);
    % filepath = sprintf('data/%s', filename);
    % csvwrite(filepath, result(100, 100));

    % Save results
    times(tasknum) = time;
    norms(tasknum) = normVal;
    diffs(tasknum) = diffVal;
end
```

4a

In the fourth assignment we were suppossed to reverse the calculation by not tranposing the first matrix but the second in the formula. This resulted in the same code but with everything reversed.

```
function result = reverse_calculate(matA)
    result = sin(log(matA + 2.7)) * log(cos(matA) + 1.7)';
end
```

4b

In 4b I was suppossed do the same as before but using a single thread I used the same logic as before

```
% Single thread
maxNumCompThreads(1);
```

4c

For 4c I did the same as 2c but reverse the input (still single threaded)

```
function result = reverse_calculate_for(inputMat)
    % Preset a matrix with zeroes so memory is only allocated once
    result_size = size(inputMat, 1);
    tempResult = zeros(result_size, result_size);

    for i = 1:result_size
        for j = 1:result_size
            iterResult = 0;

            % Calculate for each cell in the matrix
            for k = 1:size(inputMat, 2)
                integerA = inputMat(i, k);
                integerB = inputMat(j, k);

                % Get result of the dot product of items
                iterResult = iterResult + (sin(log(integerA + 2.7)) * log(cos(integerB) + 1.7));
            end

            tempResult(i, j) = iterResult;
        end
    end

    result = tempResult;
end
```

4d

For 4d I used the same operation as 2d

```
% Detect threads
maxNumCompThreads(MNCT);
```

4e

For 4e I used the same logic again but reversed the inputs

```
function result = reverse_calculate_parfor(inputMat)
    % Preset a matrix with zeroes so memory is only allocated once
    result_size = size(inputMat, 1);
    tempResult = zeros(result_size, result_size);

    parfor i = 1:result_size
        % Since in a parfor loop only the iterator can be used to access store values in vector
        iterVec = zeros(result_size, 1);

        for j = 1:result_size
            iterResult = 0

            % Calculate for each cell in the matrix
            for k = 1:size(inputMat, 2)
                integerA = inputMat(i, k);
                integerB = inputMat(j, k);

                % Get result of the dot product of items
                iterResult = iterResult + (sin(log(integerA + 2.7)) * log(cos(integerB) + 1.7));
            end

            iterVec(j) = iterResult;
        end

        % Save vector to row
        tempResult(i, :) = iterVec;
    end

    result = tempResult;
end
```

4f

For 4f the same applies as before I copied 2f and reveresed the inputs

```
function result = reverse_calculate_split(inputMat, numWorkers)
    % Define sizes
    colSize = size(inputMat, 2);
    resultSize = size(inputMat, 1);

    % Divide all the rows by the amount of workers available
    workerParts = floor(colSize / numWorkers);

    % Define all ends of ranges for workers
    workedRngEnd = ones(1, numWorkers + 1);
    for i = 1:numWorkers
        if i == numWorkers
            workedRngEnd(1, i+1) = colSize;
            break;
        end

        workedRngEnd(1, i+1) = workerParts * i;
    end

    % Use cell since matrix will stack multiple matrices on top of each other which is not good
    workerResults = {};

    % Parallel iterate over all workers to do calculations
    parfor i = 1:numWorkers
        % Get rows that belong to worker
        part = inputMat(:, workedRngEnd(i):workedRngEnd(i+1));
        % Get result from rows
        workerResults{i} = reverse_calculate(part);
    end

    % Allocate result matrix
    result = zeros(resultSize, resultSize);

    % Iterate over all workers to combine results
    for i = 1:numWorkers
        % Add all results together
        result = result + workerResults{i};
    end
end
```

```
function result = reverse_calculate_for_split(inputMat, numWorkers)
    % Define sizes
    colSize = size(inputMat, 2);
    resultSize = size(inputMat, 1);

    % Divide all the rows by the amount of workers available
    workerParts = colSize / numWorkers;

    % Define all ends of ranges for workers
    workedRngEnd = ones(1, numWorkers + 1);
    for i = 1:numWorkers
        workedRngEnd(1, i+1) = workerParts * i;
    end

    % Use cell since matrix will stack multiple matrices on top of each other which is not good
    workerResults = {};

    % Parallel iterate over all workers to do calculations
    parfor i = 1:numWorkers
        % Get rows that belong to worker
        part = inputMat(:, workedRngEnd(i):workedRngEnd(i+1));
        % Get result from rows
        workerResults{i} = reverse_calculate_for(part);
    end

    % Allocate result matrix
    result = zeros(resultSize, resultSize);

    % Iterate over all workers to combine results
    for i = 1:numWorkers
        % Add all results together
        result = result + workerResults{i};
    end
end
```

4g

For 4g the same applies and I reversed the inputs

```
function result = reverse_calculate_split_distributed(inputMat)
    % Split the input matrix over workers using the distributed function
    inputMatPart = distributed(inputMat);

    % Parallel iterate over all workers to do calculations
    spmd
        % Get result from rows
        workerResults = reverse_calculate(inputMatPart);
    end

    result = gather(workerResults);
end
```

Results

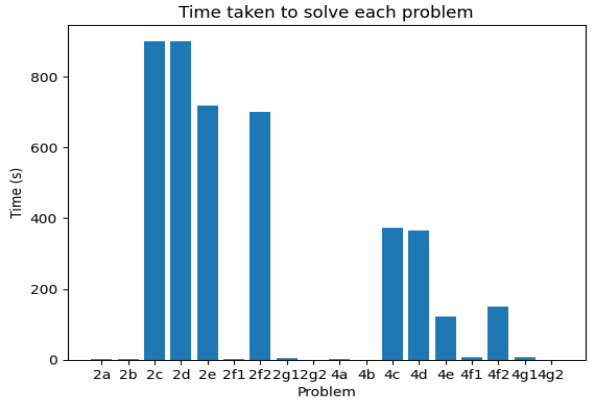
Here are the results from my experiments with only the set for the second formula reduced

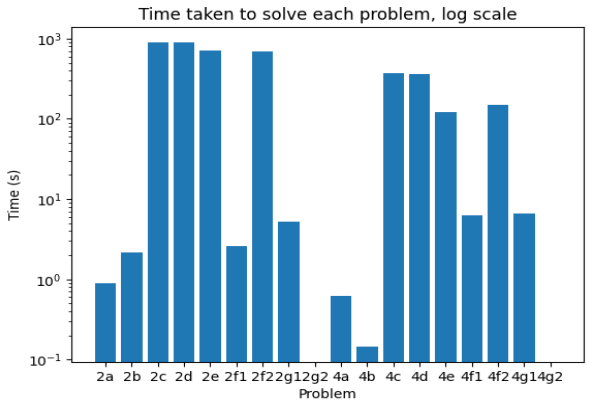
Table

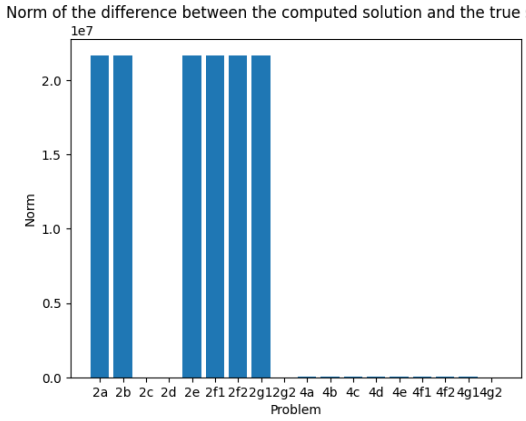
	Formula 1			Formula 2		
	Time	Norm	Diff	Time	Norm	Diff
Task A	0.890884000000	21678226.356345690787	0.000000005752	0.618432000000	42486.584789395558	0.000000000000
Task B	2.153624000000	21678226.356345690787	0	0.145287000000	42486.584789395558	0.000000000000
Task C	900	0	0	371.952780000000	42486.584789395558	0.000000000000
Task D	900	0	0	364.517514000000	42486.584789395558	0.000000000000
Task E	717.196058000000	21678226.356345690787	0.000002913663	121.422184000000	42486.584789395558	0.000000000000
Task F1	2.622502000000	21678226.356345690787	869.712054685470	6.272916000000	42486.584789395558	0.000000000000
Task F2	699.299431000000	21678226.356345690787	869.712054307089	150.084366000000	42486.584789395558	0.000000000000
Task G1	5.288033000000	21678226.356345690787	0.000000005752	6.591691000000	42486.584789395558	0.000000000000
Task G2	0	0	0	0	0	0

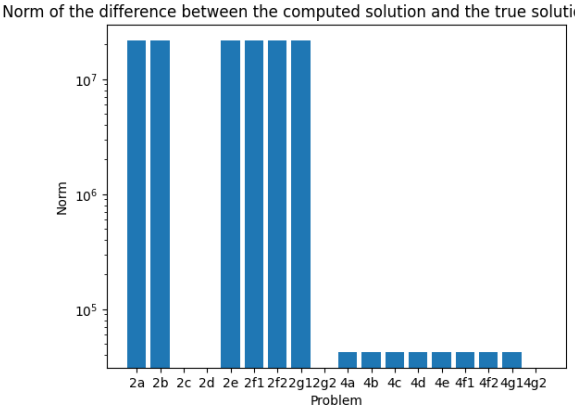
See next page for graphs

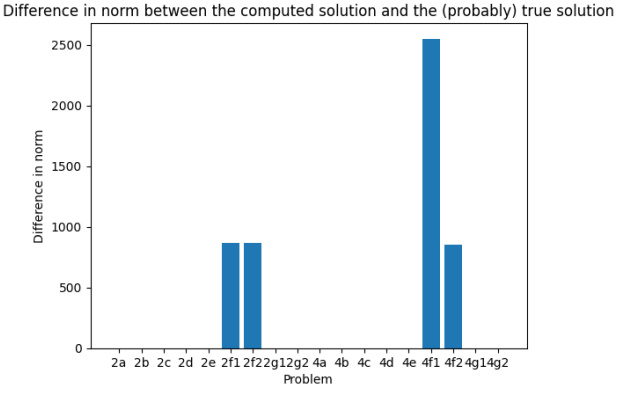
Graphs

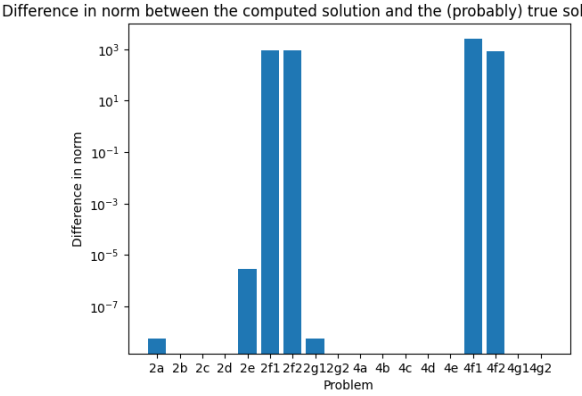

Time taken to solve each problem
Time (s)
0
200
400
600
800
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Time taken to solve each problem, log scale
Time (s)
10^{3}
10^{2}
10^{1}
10^{0}
10^{-1}
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Norm of the difference between the computed solution and the true solution
1e7
Norm
0.0
0.5
1.0
1.5
2.0
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Norm of the difference between the computed solution and the true solution, log scale
Norm
10^{7}
10^{6}
10^{5}
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Difference in norm between the computed solution and the (probably) true solution
Difference in norm
0
500
1000
1500
2000
2500
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Difference in norm between the computed solution and the (probably) true solution, log scale
Difference in norm
10^{3}
10^{1}
10^{-1}
10^{-3}
10^{-5}
10^{-7}
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem

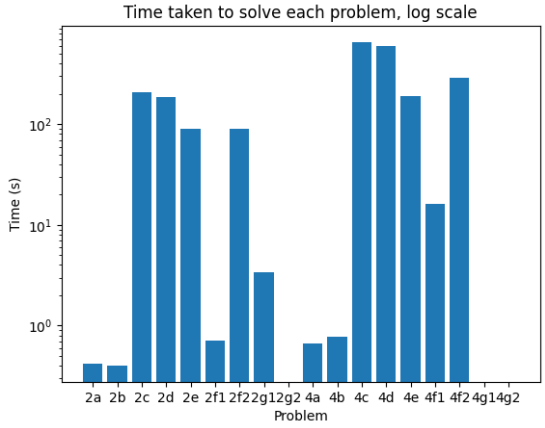
Here are the results from my experiments with both sets reduced so my system does not run into memory errors.

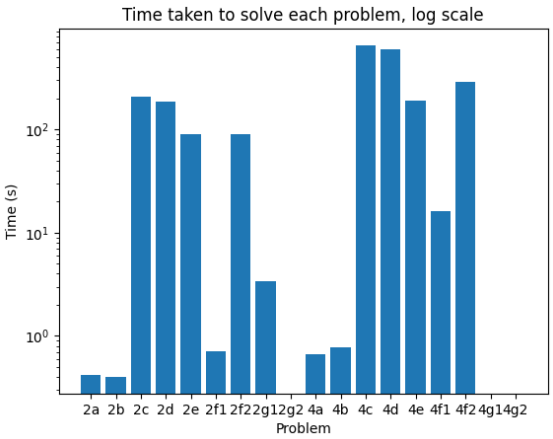
Table

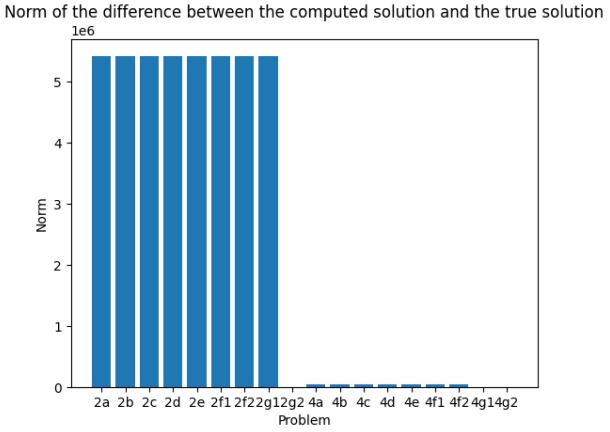
	Formula 1			Formula 2		
	Time	Norm	Diff	Time	Norm	Diff
Task A	0.418765000000	5419556.589086441323	0	0.666675000000	42486.584789395558	0.000000000000
Task B	0.400951000000	5419556.589086441323	0	0.776541000000	42486.584789395558	0.000000000000
Task C	208.807028000000	5419556.589086269960	0.000000174623	656.653535000000	42486.584789395558	0.000000000000
Task D	188.293802000000	5419556.589086269960	0.000000174623	364.517514000000	42486.584789395558	0.000000000000
Task E	89.3019540000000	5419556.589086269960	0.000000174623	121.422184000000	42486.584789395558	0.000000000000
Task F1	0.715744000000	21678226.356345690787	5419557.757233955897	6.272916000000	42486.584789395558	0.000000000000
Task F2	90.1205500000000	21678226.356345690787	5419557.757233940065	150.084366000000	42486.584789395558	0.000000000000
Task G1	3.396225000000	21678226.356345690787	5419556.589086442254	6.591691000000	42486.584789395558	0.000000000000
Task G2	0	0	0	0	0	0

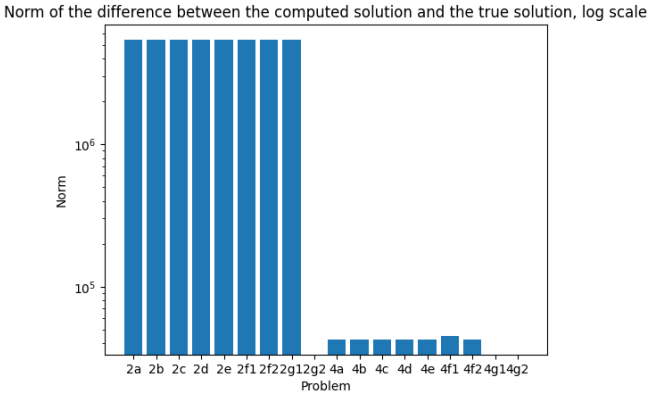
See next page for graphs

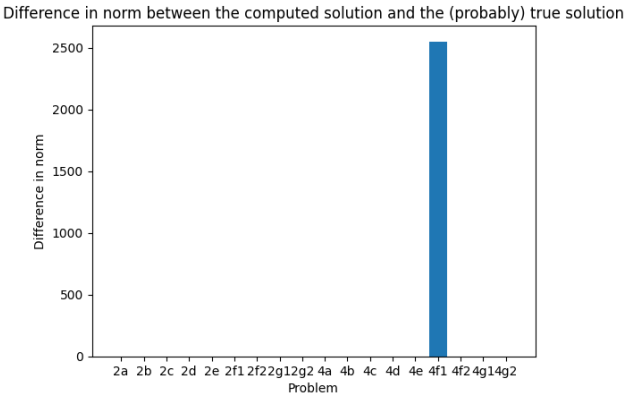
Graphs

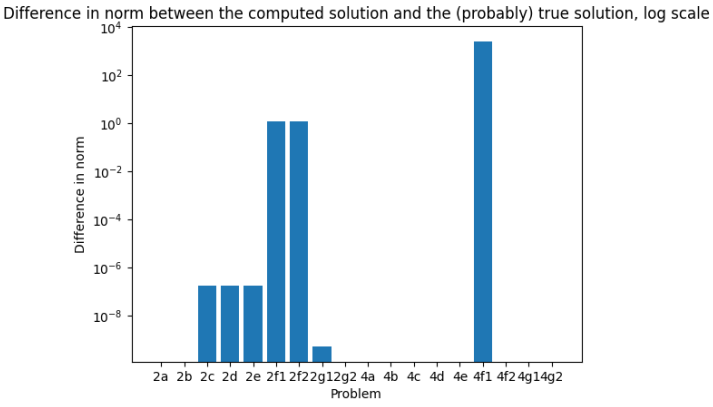

Time taken to solve each problem, log scale
Time (s)
10^2
10^1
10^0
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Time taken to solve each problem, log scale
Time (s)
10^2
10^1
10^0
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Norm of the difference between the computed solution and the true solution
1e6
Norm
5
4
3
2
1
0
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Norm of the difference between the computed solution and the true solution, log scale
Norm
10^6
10^5
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Difference in norm between the computed solution and the (probably) true solution
Difference in norm
2500
2000
1500
1000
500
0
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem


Difference in norm between the computed solution and the (probably) true solution, log scale
Difference in norm
10^4
10^2
10^0
10^{-2}
10^{-4}
10^{-6}
10^{-8}
2a 2b 2c 2d 2e 2f1 2f22g12g2 4a 4b 4c 4d 4e 4f1 4f2 4g14g2
Problem

Conclusions

From the results it can be seen that using default matrix operations is the most efficient. This is because in matlab it was introduced that basic matrix operations can be run multithreaded and using vector calculations are very efficient.

Once the logic is changed to a for loop it can be seend that not using this internal vector logic in the machine the speed reduces greatly. A normal for loop is run by a single thread so takes very long. In my results I have accessed that the running of the for loop single- or multithreaded results in a loading time of about 30 minutes (there was an e-mail that notified us that longer than 15 minutes can be notated as time limited expired (TLE)).

When switching to a parallel for loop the speed is already greatly increased by about 3 times (I run a system with 4 cores). The reason it is not 4 times faster is because the added overhead of running the for loops in parallel adds some delay.

When switching to parallel for loops using matrix operations it can be observed that the speed does increase a lot compared to the previous parallel for loop but is not equal to just using basic matrix operations. This is because the efficiency in this operation is higher than the efficiency of using a parallel for loop, which adds extra overhead.

When splitting the data manually and not letting the default parallel computing toolbox split the data per row, but splitting it equally by dividing the amount of rows by the amount of workers that are available the computation time decreases again. I conclude from this that the default behaviour is less efficient and searches for the next available row after each iteration which reduces speed, while using a pre split dataset does not require this behaviour.

Finally using distrubted arrays seems to not be more efficient than splitting the array by the amount of workers. My conclusion from this is that the spplitting of this data is less efficient than doing it manually and when reading the documentation it still leaves the current running thread free without any rows.

The assignment also stated to run the second formula (see introduction) which reverses the dimensions of the calculation. Where first the result was 1000 by 1000 the new result would be 25070 by 25070, since my system did not support the memory size needed to compute the norm I reduced by 50% to see what the relative difference would be between the first and second part. And it can be observed that using just the regular matrix operation is approximately the same duration but when it comes to for loop the time needed to compute greatly increases when compared to the 1000 by 1000 result (so now 500 by 500 after reduction). This shows that the biggest difference from using parallelisation is when the amount of calculations is high and not necessarily when a computation requires a lot of processing power.

To sum up the default matrix operations are very efficient at performing operations on big matrices and should be used. Multithreaded overhead of doing these calculations might be higher but it reduces the computation time greatly. Also it can be observed that the greatest improvement achieved by using parallelisation is for a number of rows.