KOÇ UNIVERSITY

COMP 429/529: Project 1

Faizan Safdar Ali
fali18@ku.edu.tr

1. Introduction:

In this assignment I parallelized two different applications; an image blurring algorithm and sudoku solver using OpenMP. While the first application exercises the knowledge in data parallelism, the second application uses the knowledge in task parallelism. The serial version for each application was provides and I developed my parallel implementation on top of those.

I have completed each subpart of both Image Blurring and Sudoku as well. Tests are also done for different specifications. The rest of the report is structured as follows: 2 is study on the image blurring where 2.1.1 is scalability test and 2.1.2 is thread distribution test. Lastly 3 is study on sudoku where the three parts of study are distributed as 3.1, 3.2 and 3.3.

2. Part I: Image Blurring:

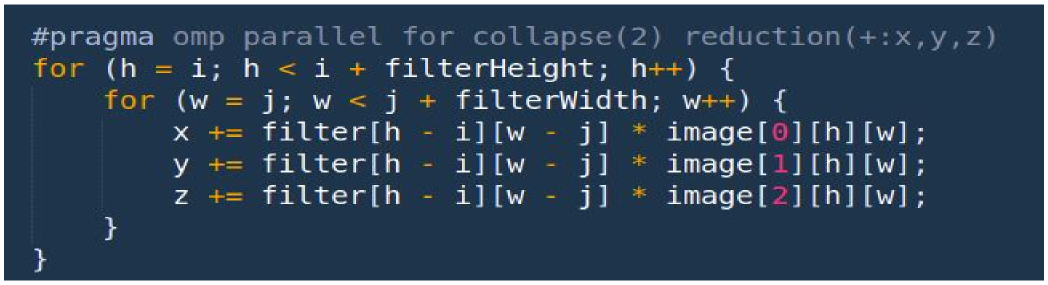
In the first part of this assignment, I implemented a parallel version of a simple image blurring algorithm with OpenMP. The image is represented as a 2-dimensional grid with three components. After reading the image to be filtered, the program generates an n by n filter. The filter is then applied to blur every pixel in the image. For pixels which are located along/near the edges of the image, zero padding technique is used to add additional zero-valued pixels beyond the edges of the image.

For this task, data parallelism was done. The objective was achieved by converting every (convertible) for loop into parallel for loop (See fig 1).

2.1. Concepts Used:

The following concepts of parallelism using OpenMP in C language were used:
1. Parallel for loop.
2. Collapse.
3. Reduction.
4. As there was no data dependency, so I did not need critical in this part.

Fig. 1



```
#pragma omp parallel for collapse(2) reduction(+:x,y,z)
for (h = i; h < i + filterHeight; h++) {
    for (w = j; w < j + filterWidth; w++) {
        x += filter[h - i][w - j] * image[0][h][w];
        y += filter[h - i][w - j] * image[1][h][w];
        z += filter[h - i][w - j] * image[2][h][w];
    }
}
```

2.2. Experimental Study:

All tests were run over the KUACC cluster provided for this assignment. Two types of tests were done on two images, “cilek.png” and “coffee.png”. The tests were:

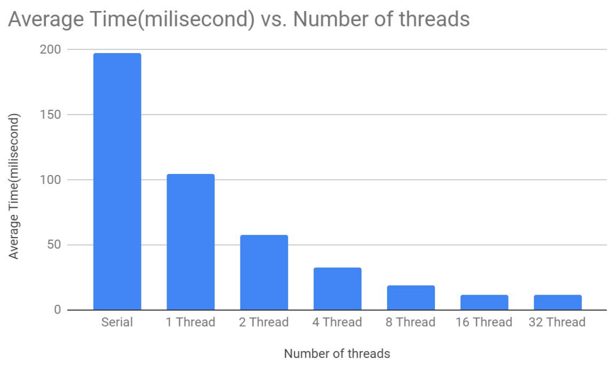
1. Scalability Test:
   - Run serially
   - Run with 1,2,4,8,16,32 Threads
2. Thread Binding Test:
   - Run compact
   - Run scatter

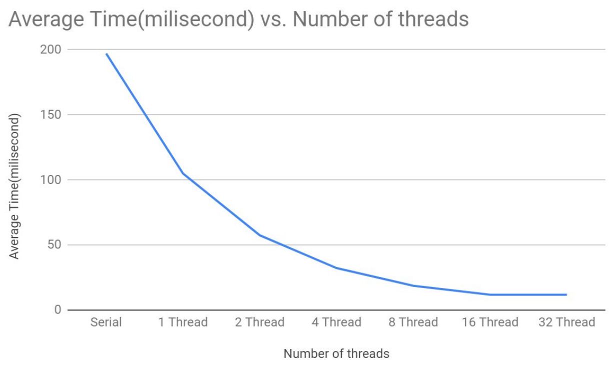
2.2.1. Scalability Test:
2.2.1.a. Test Results:

The tests were done using different number of threads, several times and on the two pictures. The results can be seen in the tables and the graphs provides as follows:

Sr Number	Number of threads	All values for time(milliseconds)	Average Time(millisecond)
1	Serial	197.55,199.03,196.85,197.96,198.51,194.50,195.55	197.1357143
2	1 Thread	104.95,104.83,104.53,105.13,105.62,104.81,104.67	104.9342857
3	2 Thread	57.31,57.46,57.16,57.89,58.11,57.08,57.22	57.46142857
4	4 Thread	32.46,32.29,32.27,31.47,33.01,32.25,32.20	32.27857143
5	8 Thread	18.59,18.54,18.66,18.25,19.20,18.60,18.56	18.62857143
6	16 Thread	11.74,11.61,11.65,11.44,12.53,11.77,11.69	11.77571429
7	32 Thread	11.67,11.74,11.77,11.45,12.17,11.85,11.75	11.77142857

Table 1. Variable number of threads. Picture “coffee.png”.

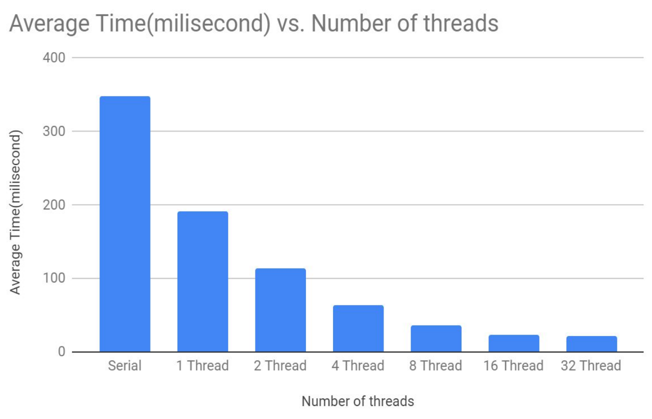


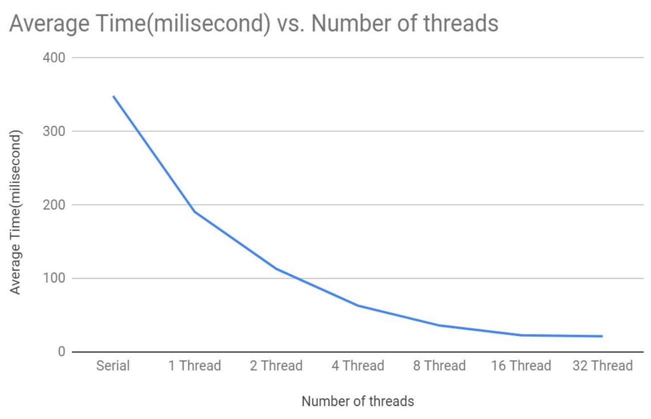


Graph 1,2 Variable number of threads. Picture “coffee.png”

Sr Number	Number of threads	All values for time(milliseconds)	Average Time(millisecond)
1	Serial	346.99,334.96,368.64,328.18,366.74,345.82,347.56	348.4128571
2	1 Thread	188.15,179.67,212.54,181.88,196.93,188.19,187.67	190.7185714
3	2 Thread	103.51,99.76,114.98,149.41,117.18,103.65,103.01	113.0714286
4	4 Thread	58.74,56.02,64.17,81.61,63.62,58.70,58.52	63.05428571
5	8 Thread	33.96,33.70,37.05,47.64 ,31.81,33.92,33.98	36.00857143
6	16 Thread	21.52,21.27,23.93,29.29,21.12,21.40,21.31	22.83428571
7	32 Thread	21.64,21.41,23.97,29.28,21.18,21.55,31.54	21.36714286

Table 2. Variable number of threads. Picture “cilek.png”.





Graph 3,4. Variable number of threads

2.2.1.b. Discussion:

The following results can be seen:

1. With the number of threads increasing, the time taken decreases.
2. The serial version takes more time than 1 thread.

We get this results because with more threads, the more work can be done simultaneously. In the task, there was no data dependency which was a good thing as it allowed more multi-threading. Also, false sharing is the important concept which can be effecting slow down, but that is out of scope of the assignment. It was also seen that the thread overhead was very little compared to the work done by the threads. Hence, more threads did show improvements.

2.2.2. Thread Binding Test:
2.2.2.a. Tests Results:

The tests were run several times first using compact and then scatter threads allocation. The number of threads were constant which was 16 and the tests were run on two pictures namely cilek.png and coffee.png. The results are described in the following tables.

Sr Number	Distribution	All values for time(milliseconds)	Average Time(millisecond)
1	Compact	22.12,22.08,21.33,21.11,21.35,22.04,21.50	21.64714286
2	Scatter	22.20,22.18,21.47, 22.16,21.34,22.14,21.57	21.86571429

Table 3. Variable distribution on picture “cilek.png”

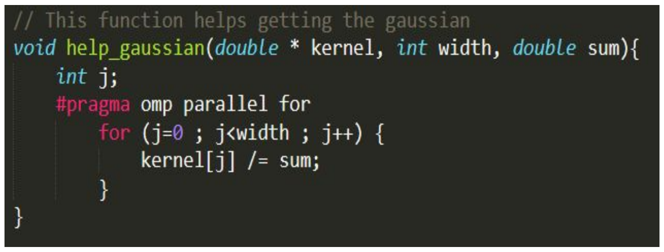
Sr Number	Distribution	All values for time(milliseconds)	Average Time(millisecond)
1	Compact	11.65, 11.71, 11.72,11.73,12.24,13.30,11.70	12.00714286
2	Scatter	11.67, 11.76, 11.78,11.79, 12.27,15.15,11,60	12.28857143

Table 4. Variable distribution on picture “coffee.png”

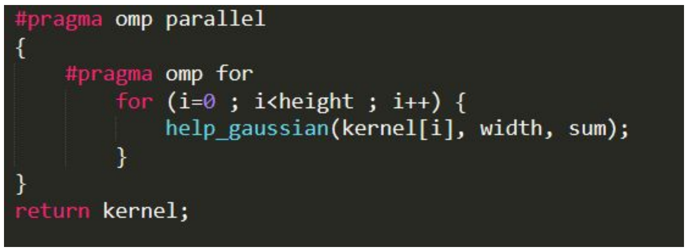
2.2.2.b. Discussion:

There was not much different could be seen during the different runs. But it could be seen that scatter was slower than compact with few milliseconds. This could be because there is false sharing happening and scatter performs poorer than compact in this situation.

2.3 Interesting Findings:



```
// This function helps getting the gaussian
void help_gaussian(double * kernel, int width, double sum){
    int j;
    #pragma omp parallel for
        for (j=0 ; j<width ; j++) {
            kernel[j] /= sum;
        }
}
```



```
#pragma omp parallel
{
    #pragma omp for
        for (i=0 ; i<height ; i++) {
            help_gaussian(kernel[i], width, sum);
        }
}
return kernel;
```

Fig. 2

While parallelising the serial code, I realized that:

1. OpenMP does not support nested for loops.
2. Even if it compiles, only one thread runs the nested loop.
3. Workaround is to create seperate functions and use parallel loops there (fig. 2).
4. Sometimes, adding parallel section increased the time. It is because of the overhead.
5. Collapse runs faster than multiple threads in above mentioned scenario.

3. Parallel Sudoku Solver:

In the second part of this assignment, I parallelized a serial sudoku solver with OpenMP. Similar to Part I, I was provided with a serial sudoku solver code, which takes a sudoku problem as an input and finds all possible solutions from it. The code is using a brute force search for searching for all possible solutions to the problem. The idea is to generate every possible move by trying each number within the game and then test to see whether it satisfies the sudoku. 16 by 16 matrices are the ones I used for performance study.

I did three types of the parallelism:

1. Task Parallelism.
2. Task Parallelism with Cutoff.
3. Task Parallelism with Early Termination.

It was achieved by using openMP tasks for calling the function again and again.

3.1. Concepts Used:

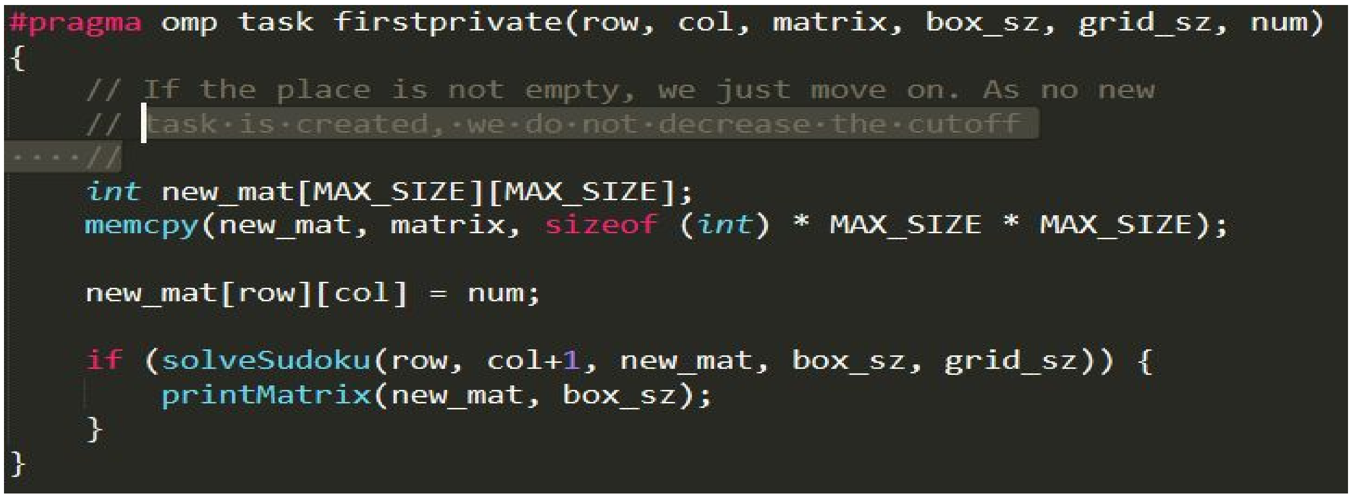
For this part, following concepts of OpenMP in C language were used:

1. Parallel tasks.
2. Critical Section (for only part-c).
3. Early termination (for only part-c).

3.2. Task Parallelism:

In the first part, I parallelized the sudoku solver with task parallelism. Similar to the serial version, the parallel version finds all possible solutions to a given sudoku problem.

For this I converted the recursive calls to the solver into parallel tasks using OpenMP pragma tasks.



```
#pragma omp task firstprivate(row, col, matrix, box_sz, grid_sz, num)
{
    // If the place is not empty, we just move on. As no new
    // task·is·created,·we·do·not·decrease·the·cutoff
····//
    int new_mat[MAX_SIZE][MAX_SIZE];
    memcpy(new_mat, matrix, sizeof (int) * MAX_SIZE * MAX_SIZE);

    new_mat[row][col] = num;

    if (solveSudoku(row, col+1, new_mat, box_sz, grid_sz)) {
        printMatrix(new_mat, box_sz);
    }
}
```

Fig. 3

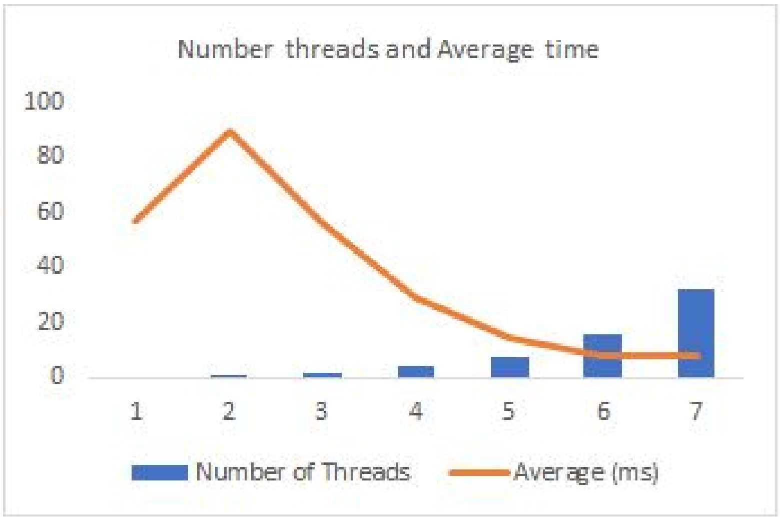
3.2.1. Experimental Study:

a) Scalability Test:

For this the 4x4_hard_3.csv was used. The number of threads were 1,2,4,8,16,32. The results can be seen in the graph and table below.

Sr Num	Number of Threads	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms)
1	0	57.88	57.83	57.87	49.8	62.49	57.174
2	1	89.51	89.61	89.02	75.26	102.68	89.216
3	2	56.5	55.79	55.77	49.15	62.36	55.914
4	4	29.3	29.49	29.48	25.91	31.28	29.092
5	8	14.98	15.12	14.95	13.74	15.72	14.902
6	16	7.74	7.66	7.59	7.13	8.09	7.642
7	32	8.5	8.55	8.39	7.96	8.56	8.392

Table 5. Variable number of thread.



Graph 5. Variable number of thread.

b) Thread Binding Test:

For this the 4x4_hard_3.csv was used. The number of threads was 16 and scatter and compact allocation was tested differently. The results can be seen in the graph and table below.

Serial	Thread Binding	Run 1(ms)	Run 2(ms)	Run 3(ms)	Run 4(ms)	Run 5(ms)

1	**Compact**	12.16	12.2	12.09	12.17	12.08
2	**Scatter**	12.31	12.51	12.52	12.51	12.35

Table 6. Thread binding.

c) Tests on Sudoku Problems of Different Grids:

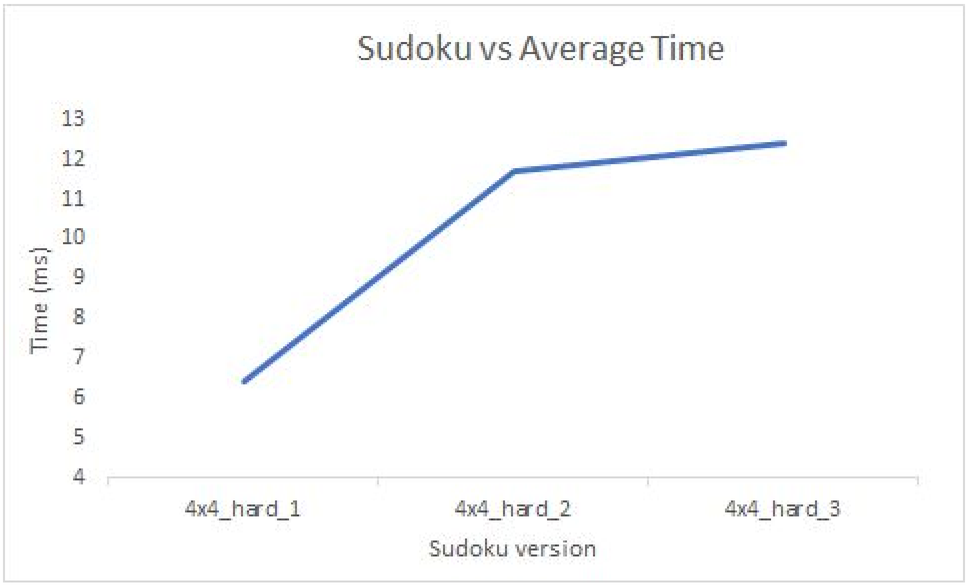
For this, number of threads were 16. Different grids of different sizes and different difficulties were used. I also ran serial threads with different grids. The results can be seen in the table below.

Serial	Sudoku	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms)
1	4x4_hard_1	29.96	28.37	27.63	29.94	25.14	28.208
2	4x4_hard_2	55.74	53.25	51.67	55.89	46.28	52.566
3	4x4_hard_3	59.95	57.71	55.99	60.02	48.99	56.532

Table 7. Different grids serial.

Serial	Sudoku	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms)
1	4x4_hard_1	6.39	6.45	6.36	6.42	6.67	6.458
2	4x4_hard_2	11.6	11.72	11.69	11.77	11.72	11.7
3	4x4_hard_3	12.39	12.55	12.38	12.5	12.23	12.41

Table 8. Different grids.



Graph 6. DIfferent grids

3.2.2 Result Analysis:

From table 5 and graph 5 we can see that:

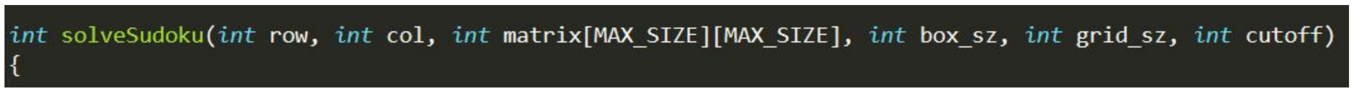
- The speed decreases when thread = 1, that is because of the switching overhead. This can not be shown in the image_filter because there are not many threads there.
- Then the speed increases with the number of threads.

From table 6, no conclusive analysis can be done, but compact is usually better.

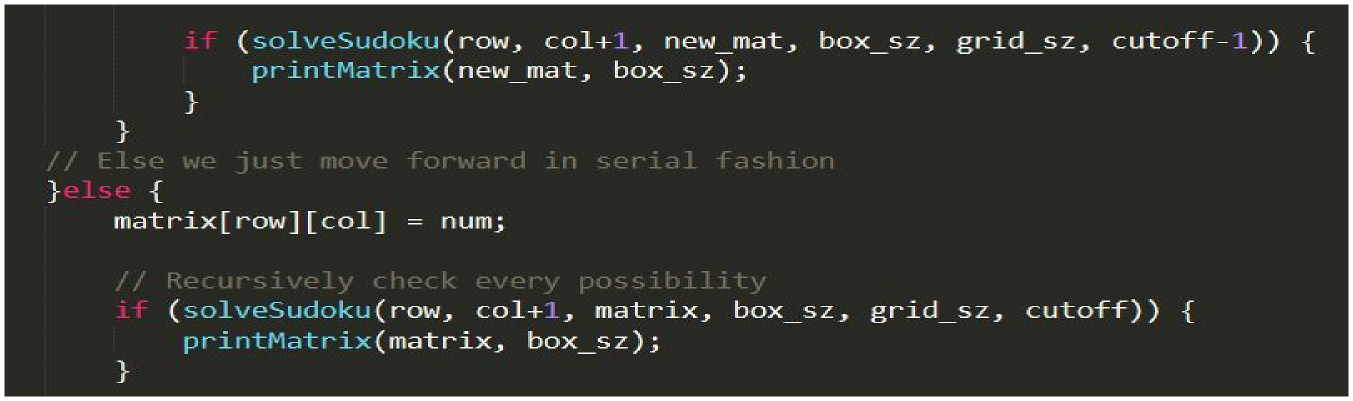
From table 7 and 8, we can see speed decreases with the increase in difficulty of the sudoku.

3.3 Task Parallelism with Cutoff:

The first implementation results in too many tasks in the system which can easily degrade the performance. As a result, I experienced very disappointing speedup. In this part, I implemented an optimization to improve the performance of the parallel code by using a cutoff parameter to limit the number of parallel tasks in your code. To limit task generations, beyond certain depth in the call-path tree of the recursive function, I switched to the serial execution and did not generate tasks afterwards. The code can be seen in figures below. The cutoff parameter used is 6.



```
int solveSudoku(int row, int col, int matrix[MAX_SIZE][MAX_SIZE], int box_sz, int grid_sz, int cutoff)
{
```



```
            if (solveSudoku(row, col+1, new_mat, box_sz, grid_sz, cutoff-1)) {
                printMatrix(new_mat, box_sz);
            }
        }
    // Else we just move forward in serial fashion
    }else {
        matrix[row][col] = num;

        // Recursively check every possibility
        if (solveSudoku(row, col+1, matrix, box_sz, grid_sz, cutoff)) {
            printMatrix(matrix, box_sz);
        }
```

fig 4.

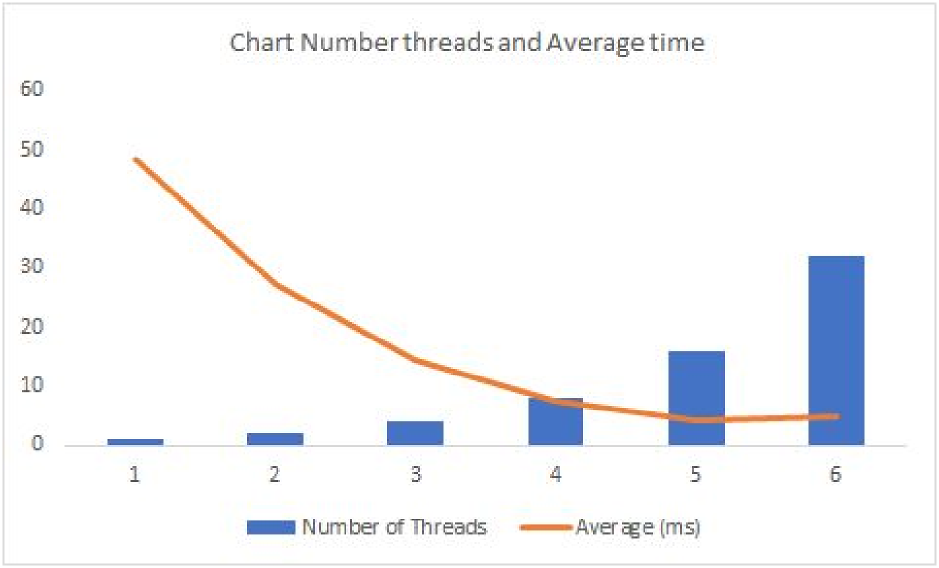
3.3.1. Experimental Study:

a) Scalability Test:

For this the 4x4_hard_3.csv was used. The number of threads were 1,2,4,8,16,32. The results can be seen in the graph and table below.

Sr Num	Number of Threads	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms)
1	1	48.35	47.94	44.2	55.54	46.77	48.56
2	2	25.39	25.03	23.53	28.13	35.3	27.476
3	4	13.62	13.49	12.85	14.48	18.25	14.538
4	8	7.16	6.95	6.9	7.42	9.96	7.678
5	16	4.12	3.88	3.77	4.04	5.76	4.314
6	32	4.14	4.23	4.43	4.54	7.55	4.978

Table 8. Variable Thread



Graph 7. Variable Thread

b) Thread Binding Test:

For this the 4x4_hard_3.csv was used. The number of threads was 16 and scatter and compact allocation was tested differently. The results can be seen in the tables below.

Serial	Thread Binding	Run 1(ms)	Run 2(ms)	Run 3(ms)	Run 4(ms)	Run 5(ms)
1	**Compact**	5.95	5.83	5.72	6.01	4.1
2	**Scatter**	5.76	5.66	5.85	5.9	3.66

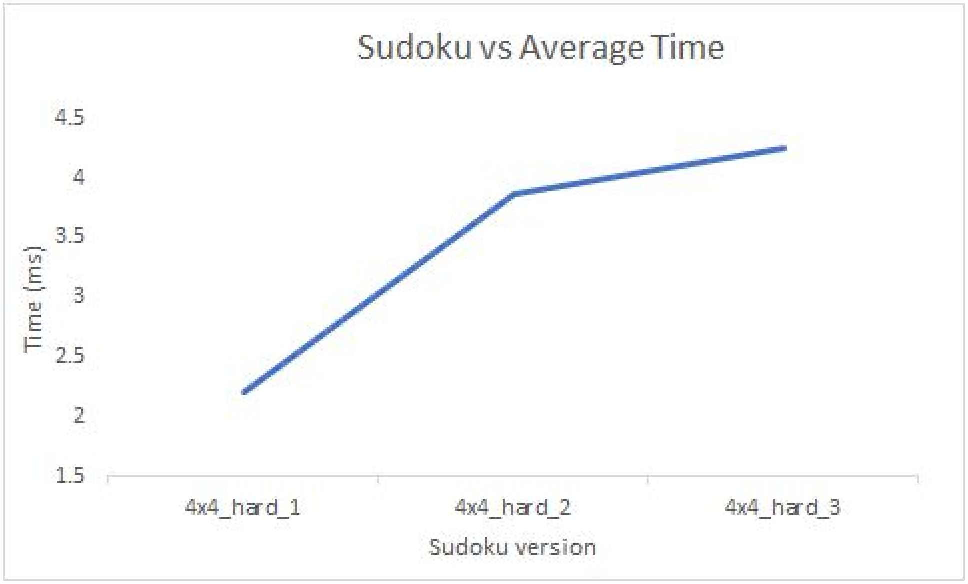
Table 9. Thread Binding Test

c) Tests on Sudoku Problems of Different Grids:

For this, number of threads were 16. Different grids of different sizes and different difficulties were used. Serial is done in the previous test so I am not including the results again here. The results can be seen in the graph and table below.

Serial	Sudoku	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms)
1	4x4_hard_1	2.09	1.97	1.92	2.16	2.91	2.21
2	4x4_hard_2	3.42	3.62	3.39	3.58	5.29	3.86
3	4x4_hard_3	3.86	4.12	3.72	3.9	5.64	4.248

Table 10. Tests on different grids



Graph 8. Different grids

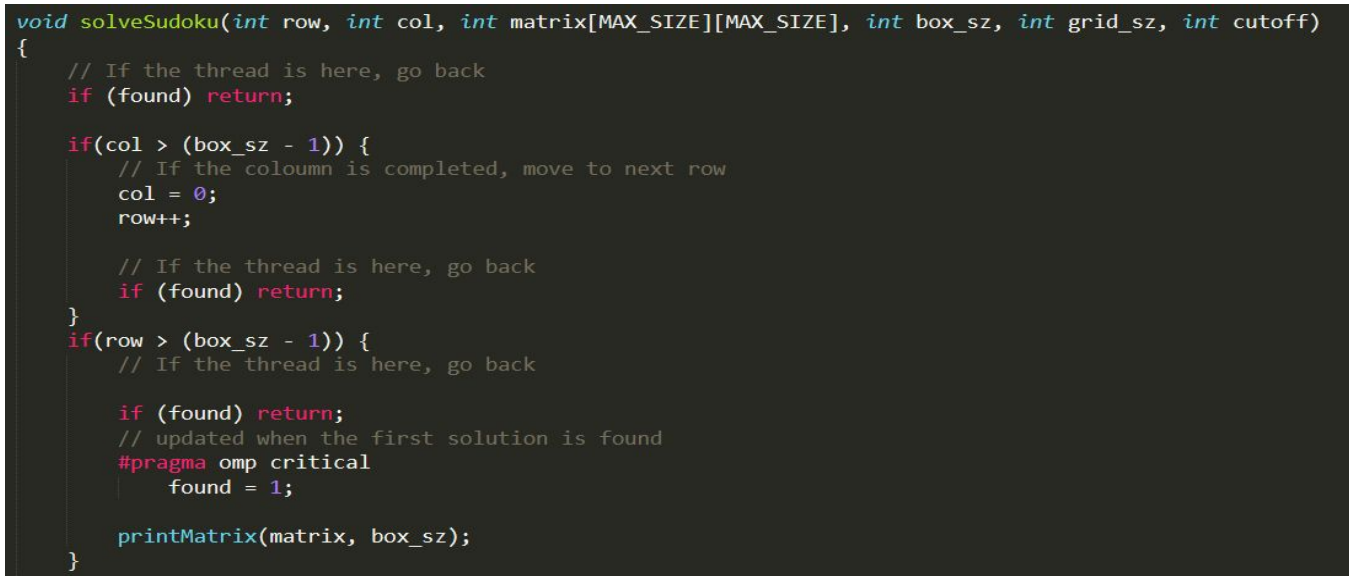
3.3.2 Result Analysis:

From the experiment we can see that the speed-up is better with any number of thread when compared to experiment 1. This is because the number of tasks is very big because of which we get thread overhead in experiment 1. When we go hybrid, we get more speed-up.

In addition:
- From table 8 we can see that the speed increases with the number of threads.
- From table 9, no conclusive analysis can be done, but compact is usually better.
- From table 10, we can see speed decreases with the increase in difficulty of the sudoku.

3.4. Task Parallelism with Early Termination:

In this part, I modified the code so that it stops after finding one solution. To make a fair performance comparison, I also modified the serial version so that it also stops after finding one solution to sudoku. In serial version, a line was added at the terminal 'if' to return after finding solution and also I changed function from returning 'int' to 'void'.



```
void solveSudoku(int row, int col, int matrix[MAX_SIZE][MAX_SIZE], int box_sz, int grid_sz, int cutoff)
{
    // If the thread is here, go back
    if (found) return;

    if(col > (box_sz - 1)) {
        // If the coloumn is completed, move to next row
        col = 0;
        row++;

        // If the thread is here, go back
        if (found) return;
    }
    if(row > (box_sz - 1)) {
        // If the thread is here, go back

        if (found) return;
        // updated when the first solution is found
        #pragma omp critical
            found = 1;

        printMatrix(matrix, box_sz);
    }
```

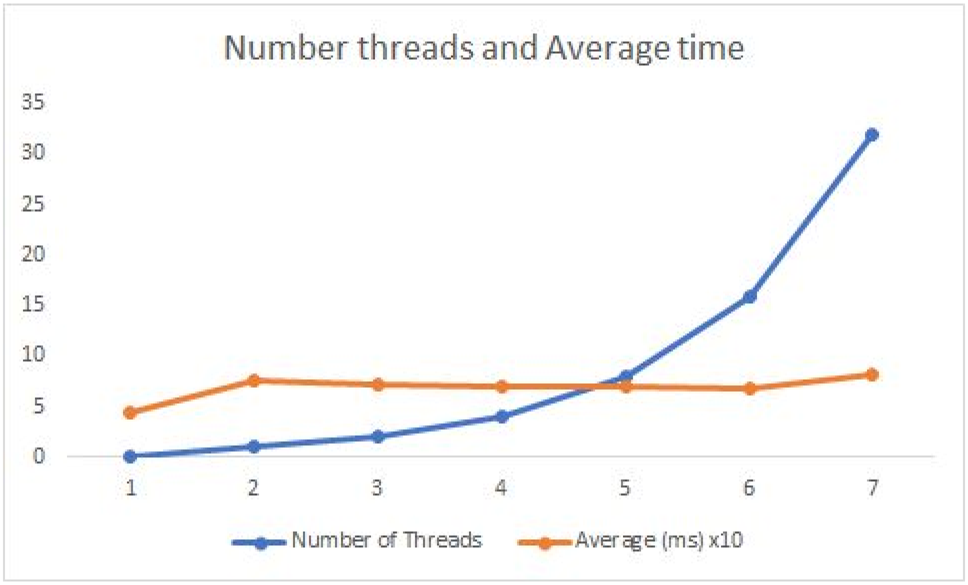
Fig. 5

3.4.1. Experimental Study:

a) Scalability Test:

For this the 4x4_hard_3.csv was used. The number of threads were 1,2,4,8,16,32. The results can be seen in the graphs and tables bellow.

Sr Num	Number of Threads	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms) x10
1	Serial	0.56	0.39	0.39	0.49	0.4	4.46
2	1	0.9	0.73	0.7	0.75	0.72	7.6
3	2	0.86	0.73	0.64	0.66	0.72	7.22
4	4	0.85	0.72	0.66	0.53	0.71	6.94
5	8	0.74	0.71	0.69	0.65	0.71	7
6	16	0.72	0.71	0.63	0.63	0.72	6.82
7	32	0.88	0.72	0.67	1.13	0.72	8.24



Graph 9. Variable threads.

b) Thread Binding Test:

For this the 4x4_hard_3.csv was used. The number of threads was 16 and scatter and compact allocation was tested differently. The results can be seen in the graphs and tables bellow.

Serial	Thread Binding	Run 1	Run 2	Run 3	Run 4	Run 5
1	**Compact**	0.9	0.85	0.72	0.67	0.91
2	**Scatter**	0.82	0.91	0.67	0.62	0.94

Table 12. Thread BInding

c) Tests on Sudoku Problems of Different Grids:

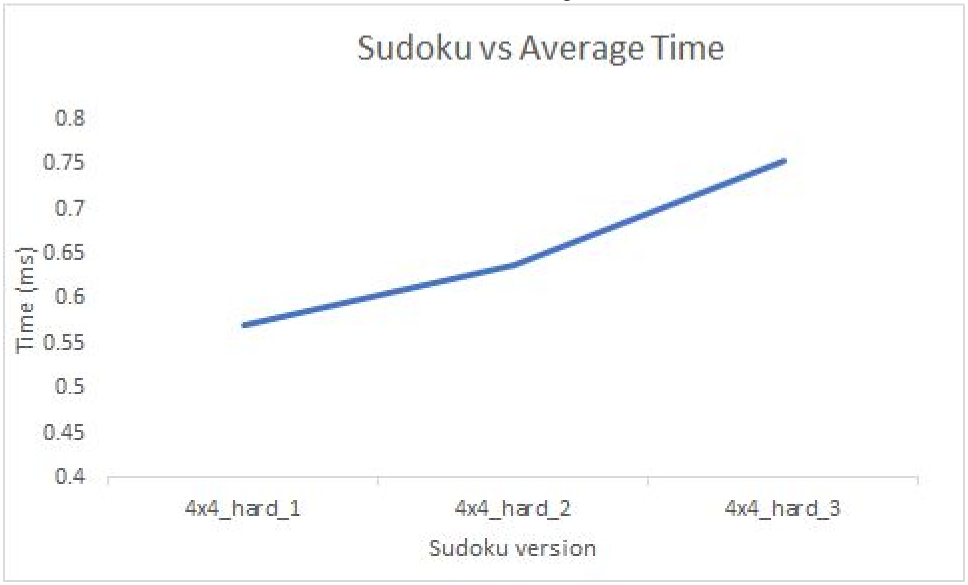
For this, number of threads were 16. Different grids of different sizes and different difficulties were used. The results can be seen in the graphs and tables bellow.

Serial	Sudoku	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms)
1	4x4_hard_1	0.27	0.28	0.32	0.34	0.25	0.292
2	4x4_hard_2	0.41	0.4	0.43	0.4	0.37	0.402
3	4x4_hard_3	0.45	0.43	0.45	0.44	0.4	0.434

Table 13. Different grid serial

Serial	Sudoku	Run 1 (ms)	Run 2 (ms)	Run 3 (ms)	Run 4 (ms)	Run 5 (ms)	Average (ms)
1	4x4_hard_1	0.62	0.35	0.38	0.9	0.6	0.57
2	4x4_hard_2	0.66	0.66	0.57	0.68	0.62	0.638
3	4x4_hard_3	0.91	0.72	0.61	0.89	0.64	0.754

Table 14. DIfferent grids

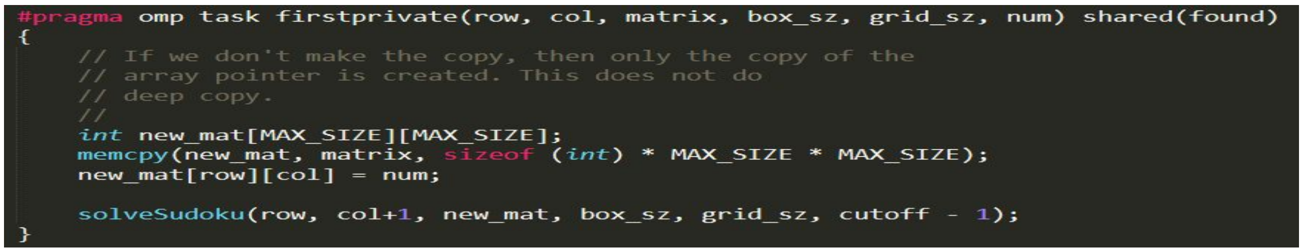


Graph 10. Different grids

3.4.2 Result Analysis:

From table 11, we can see that we do not see the speed-up. In Fact the speed decreases. This is because rather than stopping one recursion (in serial version), we have to stop the recursion of all the threads. This introduce overhead and when we couple it with the threading overhead, we see reduced speed performance.
From table 12, no conclusive analysis can be done, but compact is usually better.
From table 13, we can see speed decreases with the increase in difficulty of the sudoku both for serial and parallel version.

3.5. Interesting Finding:



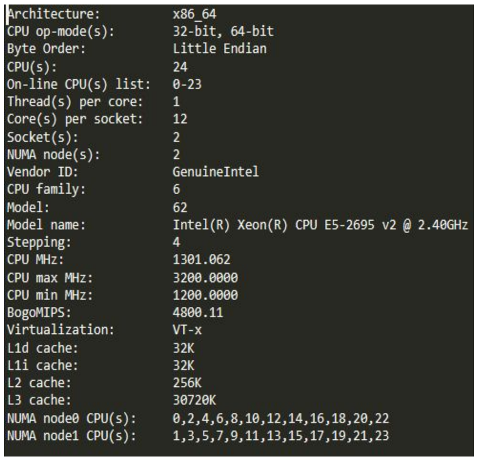
```
#pragma omp task firstprivate(row, col, matrix, box_sz, grid_sz, num) shared(found)
{
    // If we don't make the copy, then only the copy of the
    // array pointer is created. This does not do
    // deep copy.
    //
    int new_mat[MAX_SIZE][MAX_SIZE];
    memcpy(new_mat, matrix, sizeof (int) * MAX_SIZE * MAX_SIZE);
    new_mat[row][col] = num;

    solveSudoku(row, col+1, new_mat, box_sz, grid_sz, cutoff - 1);
}
```

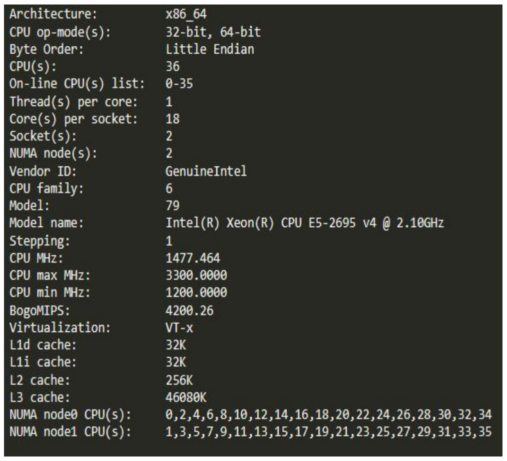
Fig 6. Copy of the array.

When we use firstprivate for the array, it does not do the deep copy, rather it just copy the pointer pointing to the same memory location. The problem arises because when the threads try to change the values, the value is changed at the same location and can create inconsistency. So I make a copy of the array and then send it to the new function call.

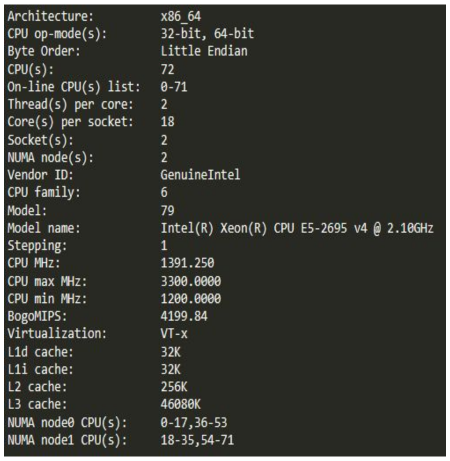
4. Architecture Analysis:



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                24
On-line CPU(s) list:   0-23
Thread(s) per core:    1
Core(s) per socket:    12
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 62
Model name:            Intel(R) Xeon(R) CPU E5-2695 v2 @ 2.40GHz
Stepping:              4
CPU MHz:               1301.062
CPU max MHz:           3200.0000
CPU min MHz:           1200.0000
BogoMIPS:              4800.11
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              30720K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23
```



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                36
On-line CPU(s) list:   0-35
Thread(s) per core:    1
Core(s) per socket:    18
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
Stepping:              1
CPU MHz:               1477.464
CPU max MHz:           3300.0000
CPU min MHz:           1200.0000
BogoMIPS:              4200.26
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              46080K
NUMA node0 CPU(s):     0,2,4,6,8,10,12,14,16,18,20,22,24,26,28,30,32,34
NUMA node1 CPU(s):     1,3,5,7,9,11,13,15,17,19,21,23,25,27,29,31,33,35
```



```
Architecture:          x86_64
CPU op-mode(s):        32-bit, 64-bit
Byte Order:            Little Endian
CPU(s):                72
On-line CPU(s) list:   0-71
Thread(s) per core:    2
Core(s) per socket:    18
Socket(s):             2
NUMA node(s):          2
Vendor ID:             GenuineIntel
CPU family:            6
Model:                 79
Model name:            Intel(R) Xeon(R) CPU E5-2695 v4 @ 2.10GHz
Stepping:              1
CPU MHz:               1391.250
CPU max MHz:           3300.0000
CPU min MHz:           1200.0000
BogoMIPS:              4199.84
Virtualization:        VT-x
L1d cache:             32K
L1i cache:             32K
L2 cache:              256K
L3 cache:              46080K
NUMA node0 CPU(s):     0-17,36-53
NUMA node1 CPU(s):     18-35,54-71
```

Fig 7.

Above are few different architectures on which I ran my jobs. I tried running each experiment several times to get different node to run my jobs on. It would be interesting to experiment on how different machine properties can affect our experiments. In addition to the time constraint for this assignment, I do not have knowledge about architecture and how it affect parallelization, so I could not do that analysis.

5. Conclusion:

In this project, I parallelized image_filter and sudoku solver using data parallelization and task parallelization respectively. I also conducted some tests to see the performance of parallelization. I concluded that sometimes it is better to do work serially than parallely.