Solved Parts:

I have solved all the parts

Question 1:

```
std::vector<std::pair<int, int>> zoom_in_my_test = {
   {223, 500},
   {555, 499},
   {252, 61},
   {620, 824},
   {843, 125},
   {459, 346},
   {921, 800},
   {588, 323},
   {668, 791},
   {503, 536},
};
```

These are my test cases that I created. I chose this to zoom in on the borders of each circle. I did this because I found that we have not tested the borders.

Code explanation:

```
#pragma omp parallel for collapse(2)
for (int y = 0; y < height; y++) {
   for (int x = 0; x < width; x++) {
     calculatePixel(x, y);
   }
 }
```

```
 vertexPixels[(y * width + x)].position = sf::Vector2f(x, y);
 vertexPixels[(y * width + x)].color =
     color == 1 ? sf::Color(red, green, blue % 256, 255)
                : sf::Color(0, 0, 0, 255);
```

These are the changed i did. I parallelized the two for loops and noticed that the pixels are being showed wrongly. After further investigation i noticed that the variable pixelcount is also benig paralleized and i didnt want that so i calculated it directly.

```
schedule(dynamic)
schedule(dynamic, 10)
schedule(dynamic, 30)
```

These are the values of the dynamic schedule.

Scheduling:

Zoom In Test Mixed:

Static: Decreases from 23685 ms (1 thread) to 2596 ms (32 threads).

Dynamic: Also decreases as threads increase, but the rate of decrease varies in different dynamic settings.

Zoom In Test All White:

Static: Fluctuates slightly, increasing at 32 threads.

Dynamic: Generally decreases, but increases with higher thread counts in some cases.

Zoom In Test All Black:

Static: Significant decrease as thread count increases.

Dynamic: Similar pattern, but the exact durations vary.

Zoom In my test:

Static: Steady decrease with more threads.

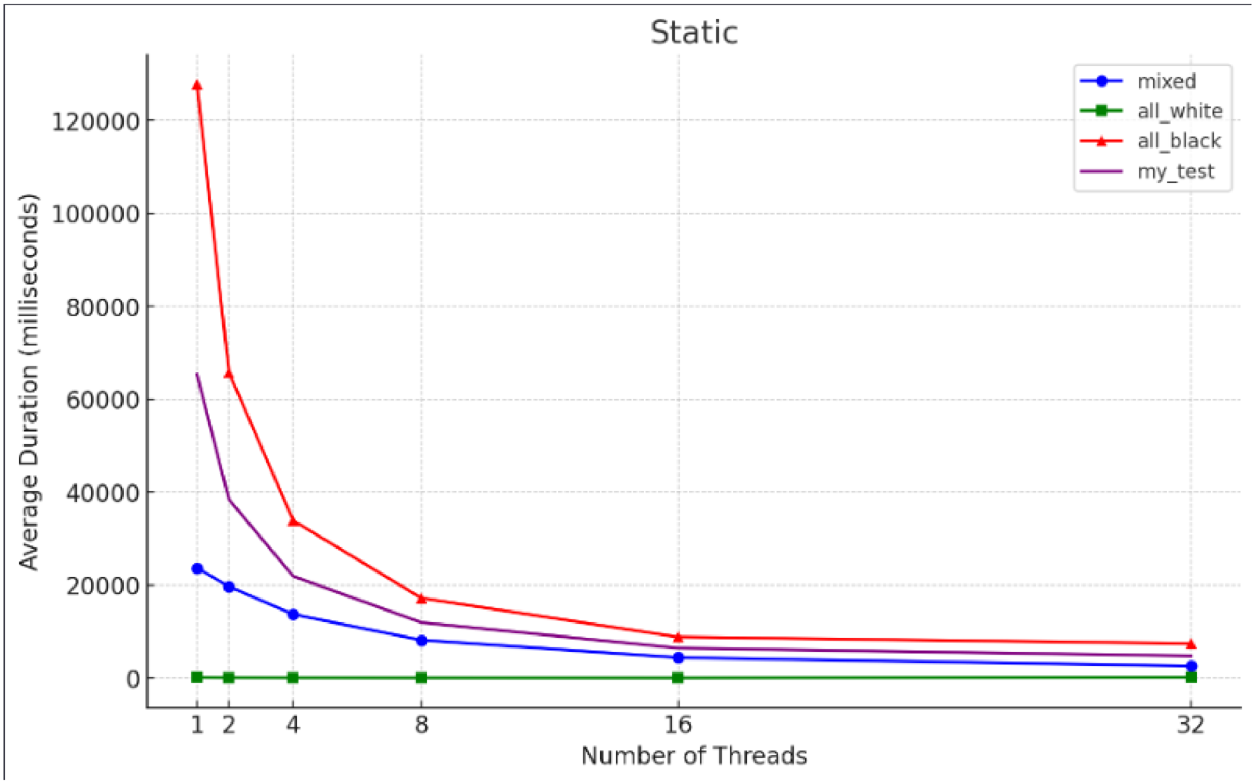
Dynamic: Decrease is evident, but the pattern varies with different dynamic settings.

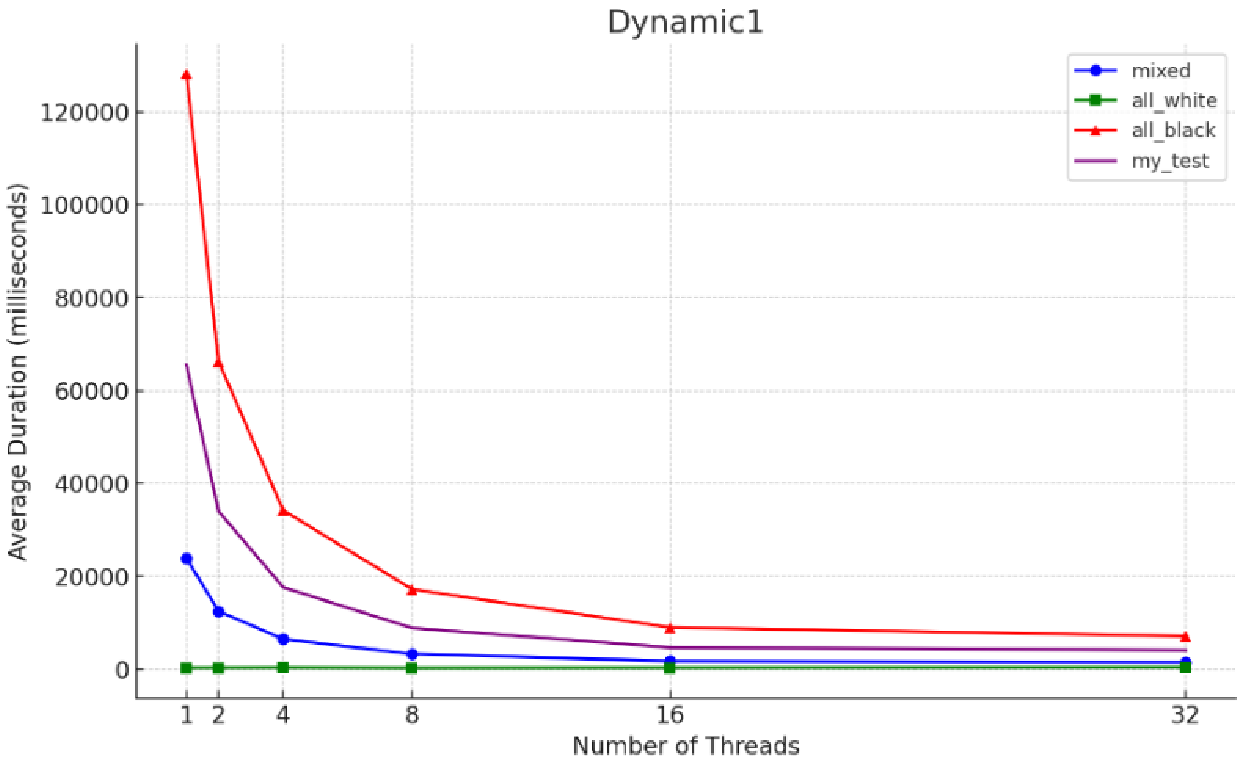
Static scheduler shows consistent speedup with more threads, likely due to efficient workload distribution among threads.

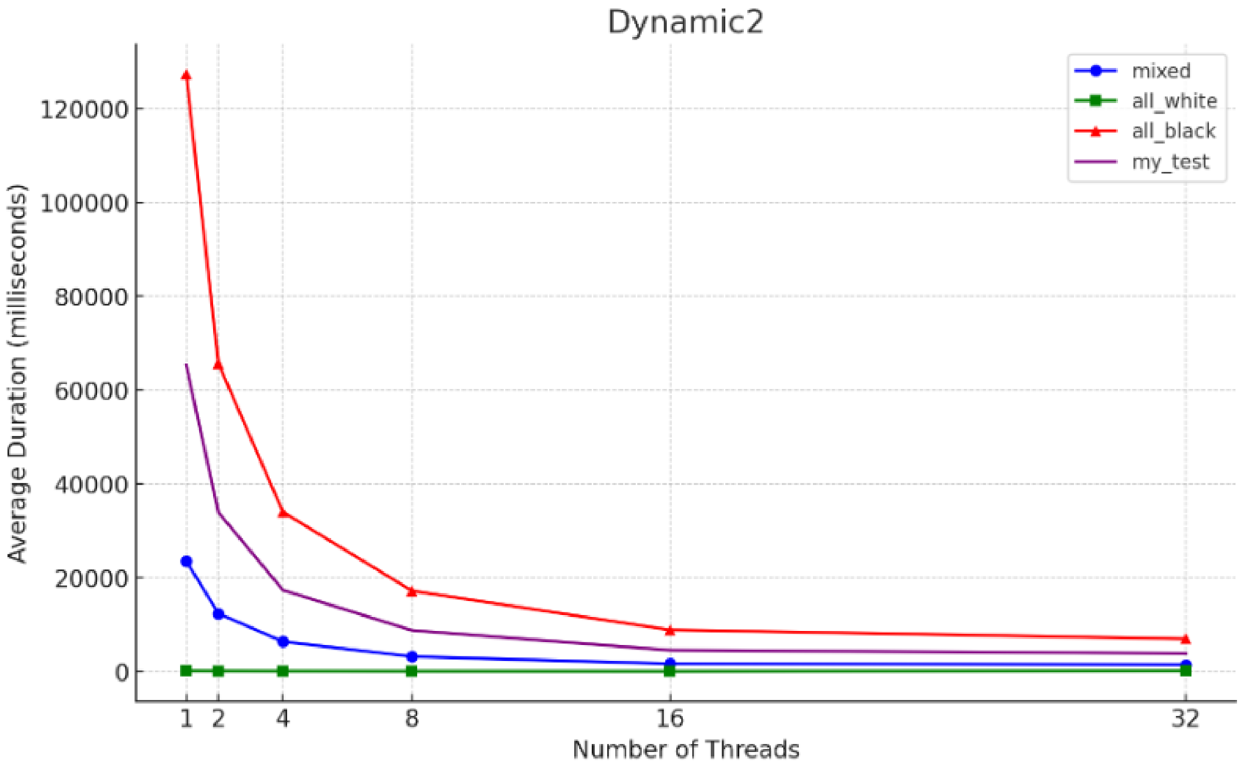
Dynamic schedulers might show variations in speedup due to the overhead of dynamic work allocation and the influence of chunk size on workload distribution.

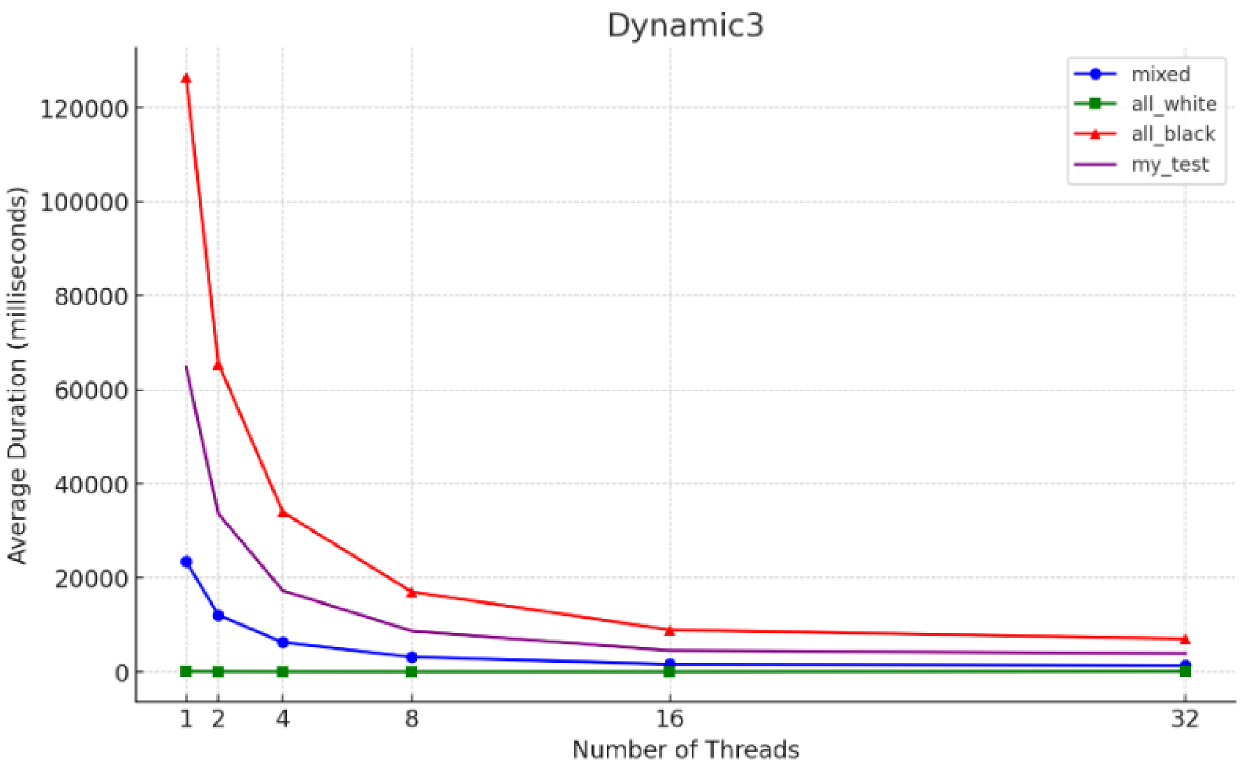
Scalability:

Test results generally demonstrated a trend toward faster performance (lower average duration) as the number of threads increased. Nevertheless, the increase in speed was not consistently linear, suggesting that there are parallelization overheads such as synchronization and thread management. The best results for each test were found at various thread counts.


Static
Average Duration (milliseconds)
0
20000
40000
60000
80000
100000
120000
Number of Threads
1
2
4
8
16
32
mixed
all_white
all_black
my_test


Dynamic1
Average Duration (milliseconds)
0
20000
40000
60000
80000
100000
120000
Number of Threads
1
2
4
8
16
32
mixed
all_white
all_black
my_test


Dynamic2
Average Duration (milliseconds)
0
20000
40000
60000
80000
100000
120000
Number of Threads
1
2
4
8
16
32
mixed
all_white
all_black
my_test



Question 2:

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

   if(col >= box_sz) {
       col = 0;
       row++;
   }
   if(row >= box_sz) {
       #pragma omp critical
       printMatrix(matrix, box_sz);
       return 1;
   }
   if(matrix[row][col] != EMPTY) { // you can do it serially
       return solveSudoku(row, col + 1, matrix, box_sz, grid_sz);
   } else {
       int num;
       for (num = 1; num <= box_sz; num++) {
           if (canBeFilled(matrix, row, col, num, box_sz, grid_sz)) {

               int local_matrix[MAX_SIZE][MAX_SIZE];
```

```
                memcpy(local_matrix, matrix, sizeof(int) * MAX_SIZE *
MAX_SIZE);
                local_matrix[row][col] = num;

                #pragma omp task firstprivate(matrix)
                {
                    solveSudoku(row, col + 1, local_matrix, box_sz,
grid_sz);
                }

                matrix[row][col] = EMPTY;
            }
        }
    }
    return 0;
}
```

```
    #pragma omp parallel
    {
        #pragma omp single
        {
            solveSudoku(0, 0, matrix, box_sz, grid_sz);
        }
    }
```

This is the general idea of my code part b and c build up from this but the general idea is the same. I the thing that i got stuck on the most was the creating a local matrix and copy the matric into it because i noticed the matrix is accessed by multiple threads at the same time. Part b checks if there is a we reached a certain depths and then continues serial. Part c builds up from that stops if we found a solution.

```
if(matrix[row][col] != EMPTY) { // you can do it serially
        return solveSudoku(row, col + 1, matrix, box_sz, grid_sz);
    }
```

This is the bottle neck i found

Scalability Test:

In the scalability test for the hard sudoku problem, the performance of parallel code in Parts A, B, and C was compared against their respective serial solvers, with a focus on speedup computation. For Parts A and B, the comparison was made against a standard serial sudoku solver, while for Part C, it was against a modified serial solver that also terminates after finding one valid solution.

Part A showed a marked improvement in performance with increasing thread counts. Starting from an average duration of 138.854 seconds with one thread, it reduced to 11.538 seconds with 32 threads, indicating significant speedup.

Part B displayed a similar trend, with durations decreasing from 124.58 seconds with one thread to 7.032 seconds with 32 threads, underscoring efficient parallelization.

Part C, being a different parallel code design that terminates after finding one solution, showed a remarkably low duration even with a single thread (0.724 seconds) and maintained a consistent performance across varying thread counts, with a slight increase in time at higher thread counts.

Tests on Sudoku Problems of Different Grids

Testing the parallel code from Part B on different sudoku grids of sizes and difficulties (4x4 hard1, 2, and 3) using 32 threads revealed:

4x4 Hard 1: Average duration was 4.552 seconds.
4x4 Hard 2: Average duration increased to 8.424 seconds.
4x4 Hard 3: Slightly higher duration at 8.008 seconds.

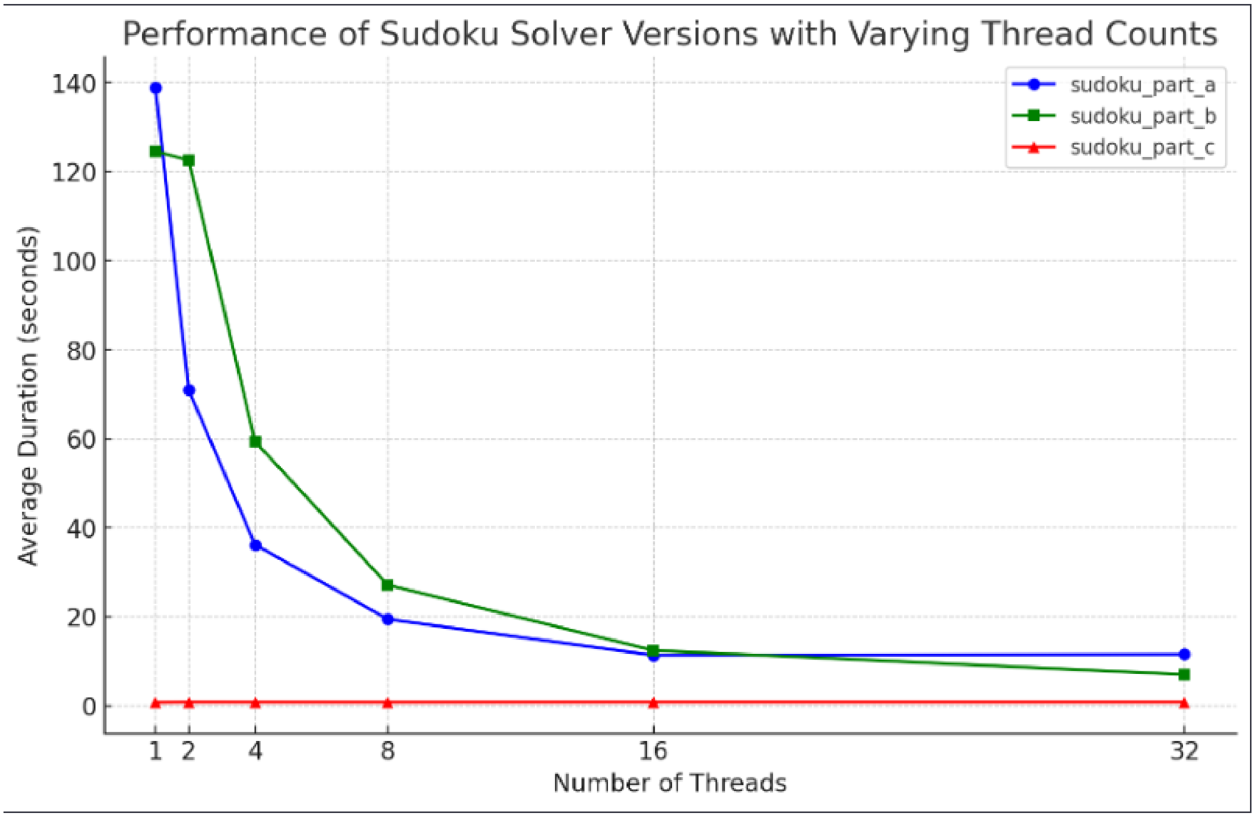
These results suggest that the complexity of the sudoku problem impacts the performance of the parallel solver. Despite using a high thread count (32 threads), the variation in solving times reflects how different puzzle complexities pose varying challenges to the parallel solver. The increase in solving time from Hard 1 to Hard 2 indicates that certain puzzles may inherently be more complex and require more computational resources, even in a parallelized environment. However, the decrease in time from Hard 2 to Hard 3, although slight, might indicate varying efficiencies in the solver's handling of specific puzzle configurations. This test emphasizes the importance of considering the nature of the problem while assessing the performance of parallel algorithms.

Tests on Sudoku Problems of Different Grids:

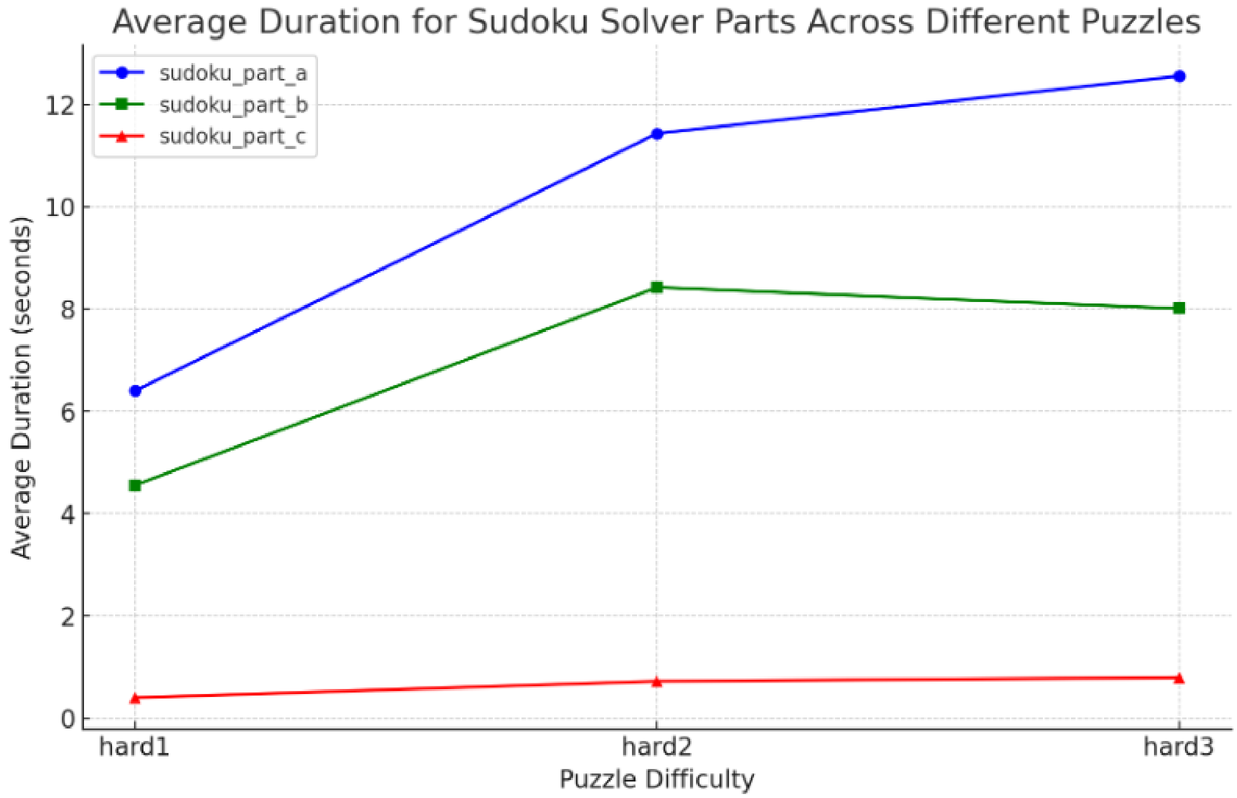
Testing the parallel code from Part B on different sudoku grids of sizes and difficulties (4x4 hard1, 2, and 3) using 32 threads revealed:

4x4 Hard 1: Average duration was 4.552 seconds.
4x4 Hard 2: Average duration increased to 8.424 seconds.
4x4 Hard 3: Slightly higher duration at 8.008 seconds.

The increase in solving time from Hard 1 to Hard 2 indicates that certain puzzles may inherently be more complex and require more computational resources, even in a parallelized environment. However, the decrease in time from Hard 2 to Hard 3, although slight, might indicate varying efficiencies in the solver's handling of specific puzzle configurations.



I have found I have achieved good results because There is a huge speedup in both parts after parallelizing them. Even after tested locally it has become faster.


Average Duration for Sudoku Solver Parts Across Different Puzzles
sudoku_part_a
sudoku_part_b
sudoku_part_c
Average Duration (seconds)
0
2
4
6
8
10
12
hard1
hard2
hard3
Puzzle Difficulty