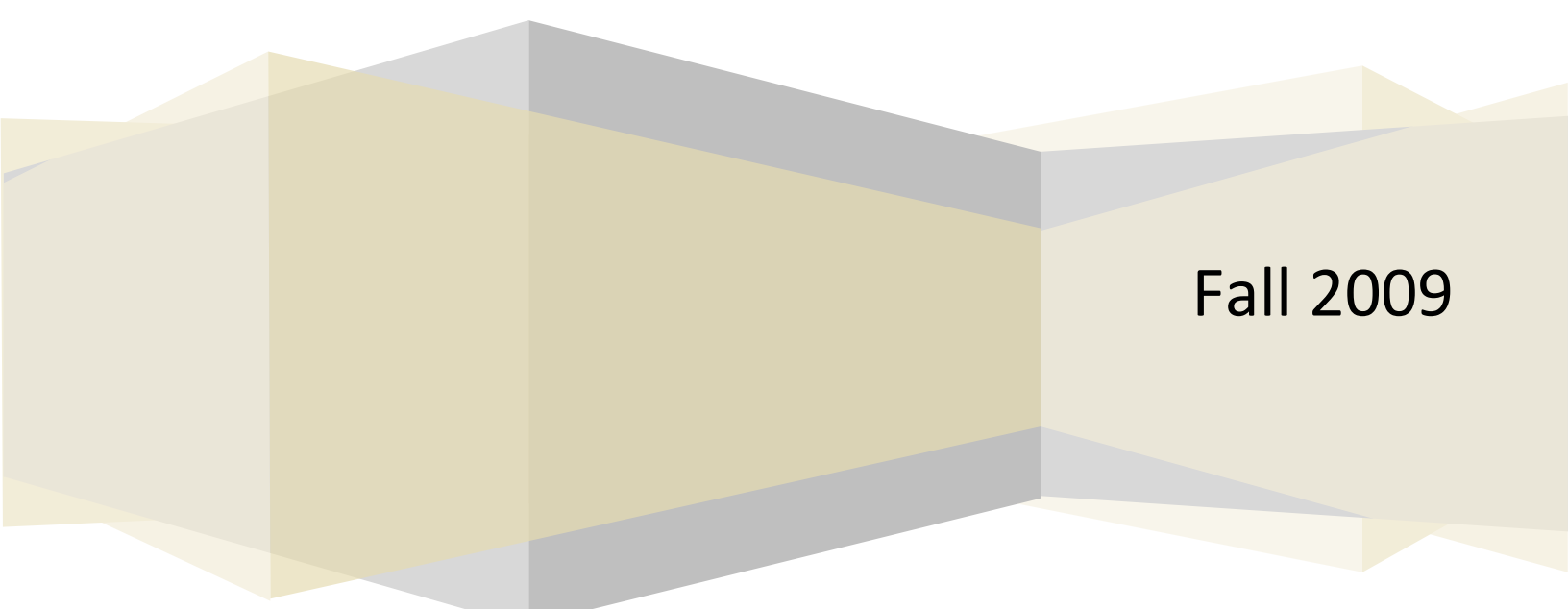


ECSE420 - Parallel Computing

# Multi Thread Performance Analysis of a Brute-Force Sudoku Solver

Simon Foucher G07

A decorative graphic at the bottom of the slide consisting of several overlapping, semi-transparent, 3D-style rectangular blocks in shades of gold, grey, and white, arranged in a horizontal line.

Fall 2009

## THE GAME

Sudoku is a popular puzzle game similar to crosswords, but using numbers. The Sudoku grid is a square, divided in subsquares containing as many elements as the length of the row of the main square. Which such a partition, every element in the Sudoku board has 3 constraints: row, column and sub square. The constraint stipulates that a number, ranging from 1 to the size of a subsquare, can only be placed once in a row, a column and a sub square.

There are many strategies one can use to solve Sudoku puzzles, and solutions might not be unique, or might not even exist! The difficulty rating of a board is relative to the number of indices provided. A standard Sudoku board contains 81 places, and is subdivided into 9 subsquares each containing 9 elements.

## APPROACH

When solved by a computer, one of the most complete approaches to solving a Sudoku is using brute force. This provides 2 advantages over algorithmic solvers: first, it can confirm if there are no solutions, and second, if there are many solutions, the computer can easily output a complete list of them. In this work we have developed a multi-threaded brute force Sudoku solver. To standardize the approach, the same board has been used throughout the development. The puzzle is rated as “Extremely difficult” and has been taken from:

[http://en.wikipedia.org/wiki/Algorithmics\\_of\\_sudoku](http://en.wikipedia.org/wiki/Algorithmics_of_sudoku)

```
-----  
| 1 * * | * * * | * * 2 |  
| * 9 * | 4 * * | * 5 * |  
| * * 6 | * * * | 7 * * |  
-----  
| * 5 * | 9 * 3 | * * * |  
| * * * | * 7 * | * * * |  
| * * * | 8 5 * | * 4 * |  
-----  
| 7 * * | * * * | 6 * * |  
| * 3 * | * * 9 | * 8 * |  
| * * 2 | * * * | * * * |  
-----
```

Figure 1: the Sudoku puzzle used for gathering the performance data.

The experiment will consist in repeatedly solving this puzzle using the same algorithm with a varying number of threads to observe the variation in performance obtained.

## ALGORITHM

The algorithm uses recursive calls on a test-and-backtrack approach. All the cells of the board will be visited.

1. Upon arriving, if the cell is not empty, it is skipped. Otherwise, a temporary value is assigned from 1 to 9. In figure 2, we can observe that the first cell (0,0) got skipped and that the value of '1' has been assigned to the next cell (0,1)

```
-----  
| 1 1 * | * * * | * * 2 |  
| * 9 * | 4 * * | * 5 * |  
| * * 6 | * * * | 7 * * |  
-----  
| * 5 * | 9 * 3 | * * * |  
| * * * | * 7 * | * * * |  
| * * * | 8 5 * | * 4 * |  
-----  
| 7 * * | * * * | 6 * * |  
| * 3 * | * * 9 | * 8 * |  
| * * 2 | * * * | * * * |  
-----
```

Figure2: Testing the legality of a '1' at the position (0,1)

2. Afterwards, a test is performed to see if the attempted value is allowed there. By scanning the 3 dimensions of neighbors, the new value is compared to all the elements contained in its row, its column and its sub square. If the same element is encountered, the next value is tested and the algorithm repeats the test. If all the values have been exhausted, the algorithm backtracks to the previous cell.
3. This process is repeated until the program finds an element that could potentially exist at that location

```
-----  
| 1 3 * | * * * | * * 2 |  
| * 9 * | 4 * * | * 5 * |  
| * * 6 | * * * | 7 * * |  
-----  
| * 5 * | 9 * 3 | * * * |  
| * * * | * 7 * | * * * |  
| * * * | 8 5 * | * 4 * |  
-----  
| 7 * * | * * * | 6 * * |  
| * 3 * | * * 9 | * 8 * |  
| * * 2 | * * * | * * * |  
-----
```

Figure 3: '3' is the first 'legal' value we can leave at position (0,1) before moving on to the next position

4. Once this condition is satisfied, the function is recursively called to the next location. To optimize the software, when the function is called a second time, the first element it looks at is

the previous element incremented by 1. For example here, there since we just guessed '3' to be at location (0,1), the algorithm will guess '4' the next location

5. Once the depth of the recursion reaches 81, we know that all the cells have been visited, so if no solution was found, we backtrack one cell and keep trying all the possibilities on that one by recalling the recursion.
6. This keeps happening until all the possibilities have been tried out. If no solution was bound in the mean time, an error message is printed on the board (for the purpose of this experiment, we know that there is a solution so it is always found sooner or later)

## OPTIMIZING BRUTE FORCE

By definition, a brute force algorithm is the least optimal solution to a problem, but guarantees results. The time required to solve the puzzle greatly depends on where the search is conducted, and which elements are visited first. It is even possible, although unlikely to take  $O(1)$  time if the first try is the right answer.

It might have been tempting to try and optimize the solver by testing various starting positions and various start number, but this optimization would have only been valid for this particular puzzle. We felt that the best approach was to randomize the start location, as well as the start guess. In order to implement this, we used circular loops to scan the rows, columns and tested values. To keep track of how 'deep' we got in the algorithm, we incremented an index at every function call that made the function exit when it reached 81 (the maximum depth of the grid).

This approach also gives us great versatility with regards to multithreading environment. By starting at a random location, the solving function can be called by many threads and generate good 'far apart' starting states for each thread. This added feature gives the user to freely add threads without having to modify anything in the algorithm for the new threads to pick up a work load.

## MULTI-THREADING AND SYNCHRONIZATION

Since the challenge is to test a very large possibility space, the use of parallel computing can greatly aid. For this particular problem, we used many threads to run a subset of the problem. Java was chosen as a platform because of its ease of implementation of multithreading environments. The main class takes care of creating threads, and every thread takes on a portion of the puzzle. As soon as one of them solves the problem, it prints out the solution and the time it took to solve it.

To handle synchronization, we used a global variable called 'finished'. When the main function starts, it sets 'finished' to false. As the threads try to solve the puzzle, they look at this variable and exit if it is true, otherwise continue to try and solve. As soon as a thread solves the puzzle, it sets the variable to true, which forces all the other threads out of their solving function. Meanwhile, in the main function, after all the threads have been initialized, the program falls into a `while(!finished)` loop. As soon as the barrier is crossed, we know that a thread has solved the puzzle, so all the threads are destroyed.

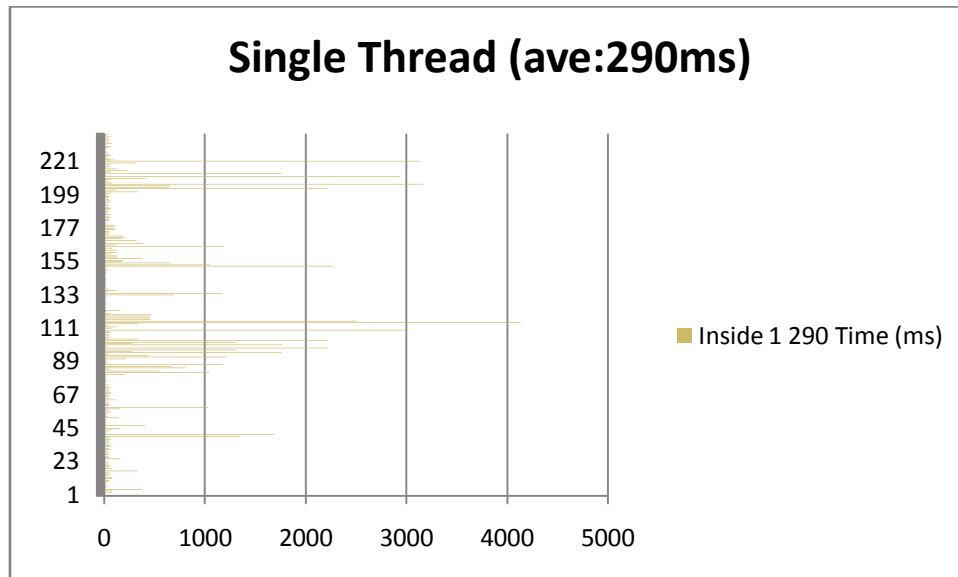
## EXPERIMENT

Since, in general, we do not have *a priori* information on the best starting location for a brute force scan; we felt the need to test various scenarios to make sure that the data collected was a good reflection of reality. Therefore, in order to gather a comparative idea of various times to solve, rather than the particular value for this puzzle, many runs of the same test were performed. We also tested for many different numbers of threads.

The tests were conducted on an Intel Dual core processor @1.4GHz, managed by the Windows 7 platform.

## RESULTS

First we ran some tests on a single thread to see how long it took on average to solve the puzzle. Here are the results we found.



**Figure 4:** Individual results of a single threaded application solving the puzzle. On the x axis, we can see how long it took, and on the right, we can see the trial number.

These results justified the assumption made that the time it takes to fully solve a Sudoku puzzle on a brute force approach is greatly relative to the starting location. We can observe a few instances where it took 3-4 full seconds to solve, these represents where statistically we picked the worst possible starting point and the solution was one of the last ones we tested. As the other extreme, we can observe some instances where the puzzle was solved in a matter of milliseconds. These instances occur when the solution is one of the first ones to be found.

Afterwards, we ran the application with a varying number of threads working on solving the puzzle, from 1 to 10, and got the following results, which are similar to the original results we got. For any given number of threads, 500 tests were performed to ensure that the data gathered was statistically significant. (The number of different starting positions is  $9 \times 9$  with 9 possible starting values, giving 729 possible initial states for the system.)

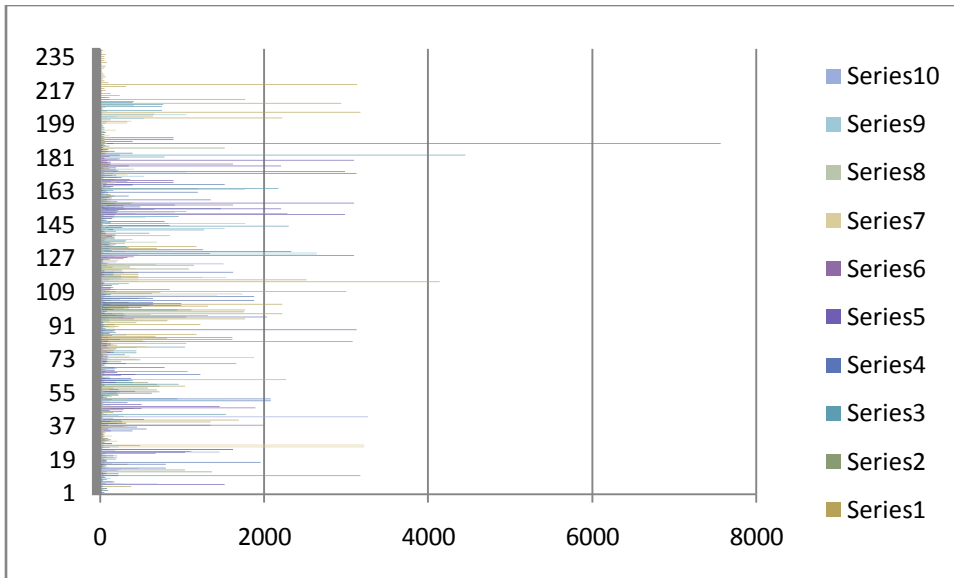


Figure 5: Combined results of all the tests performed with a varying number of threads. On the x axis, we can observe the time to solve, in ms.

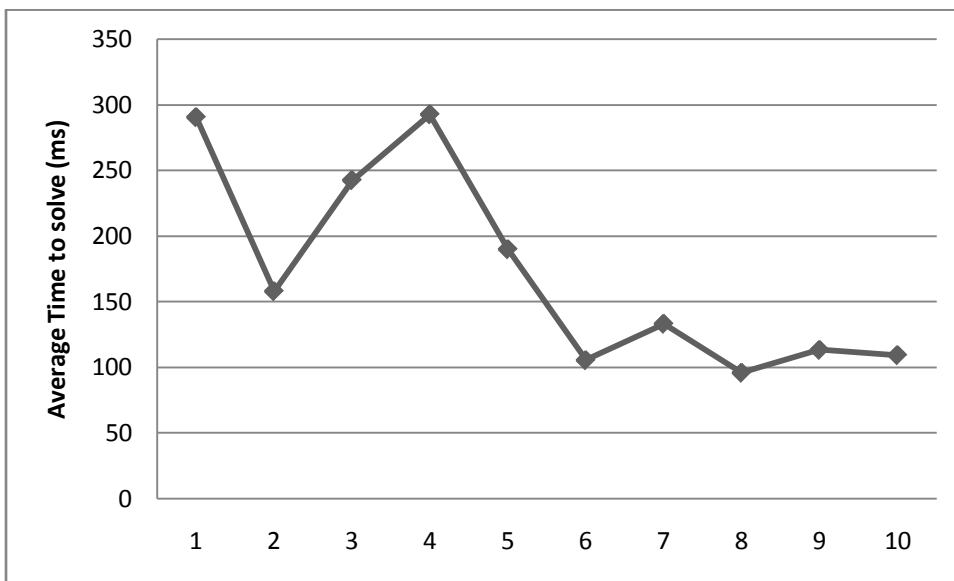


Figure 6: Average time to solve when using a variation of number of threads. On the x axis we can see the number of threads used, and on the y axis we can see the average time to solve in ms.

These are interesting results, which require a few explanations. First, as expected by running the code on a dual core processor, we can see that there is a sharp decrease in the average time required to solve the puzzle when going from a single threaded application to using 2 threads. This can intuitively be explained because of the fact that 2 processors can easily handle 2 threads without too much overhead.

When adding a 3<sup>rd</sup> and 4<sup>th</sup> thread, we can see that the average time to solve increased almost as far up as the original time it took for a single threaded application. This is caused by the added overhead brought by managing the extra threads without bringing too many benefits.

Afterwards, we observe a gradual decrease in the time to solve as the number of processors keeps increasing, until it bottoms out at 100ms. This continuous decrease is most probably due to the random nature of the starting position and starting number. As we can observe in figure 4, when comparing the performance we got from different starting locations, the data is mostly gathered in very low solving times (<200ms), with a few high peaks (>1s), which drive the average up. Since the vast majority of the starting positions can yield a fast answer on a brute force approach, by adding threads starting at random locations, we increase the chance that at least one of them starts in that location. If this is the case, as soon as this thread reaches the solution, all the other ones will be stopped anyways.

Finally, as the number of threads keeps increasing, we can observe a slight increase in the average time to solve. This average shows a weak asymptote at around 100ms, with a slight positive slope, indicating that the added threads do induce some extra overhead.



## APPENDIX I: SOURCE CODE USED

The code consists of a single java file and was developed using the IDE Eclipse, and has been inspired by an algorithm proposed by Bob Carpenter on the following web page: [http://www.colloquial.com/games/sudoku/java\\_sudoku.html](http://www.colloquial.com/games/sudoku/java_sudoku.html).

```
package sudokuSolver;
import java.util.Random;

// Class taking care of solving the Sudoku

public class solver extends Thread {

    static boolean finished;

    // This method is called when a thread starts running
    // It will call the recursive solver and output the data
    public void run() {
        int[][] board = setUpBoard(1);
        int i, j, value;
        long start, end;
        Random generator = new Random(System.currentTimeMillis());
        //printBoard(board);

        board = setUpBoard(1);
        i = generator.nextInt(8);
        j = generator.nextInt(8);
        value = generator.nextInt(8)+1;

        start = System.currentTimeMillis();
        if (solve(i, j, board, 0, value)) ; // solves in place
        //printBoard(board);
        else
            System.out.println("No answers for this puzzle!");

        end = System.currentTimeMillis();

        if(!finished){
            finished = true;
            System.out.print((end - start)+"\n");
        }

    }

    public static void main(String[] args) {

        // 1 threads thread
        System.out.println("\n\n1\nTime (ms)");
        for(int k = 0; k < 400; k++){

            finished = false;
            Thread thread1 = new solver();

            thread1.start();
            while(!finished);

            thread1.stop();

        }

        // 2 threads thread
        System.out.println("\n\n2\nTime (ms)");
        for(int k = 0; k < 400; k++){
            finished = false;
            Thread thread1 = new solver();
            Thread thread2 = new solver();

            thread1.start();
            thread2.start();

            while(!finished);
        }
    }
}

// Set up the board (option to select different puzzles)
// Start position (i,j) and start value will be
randomly selected
```

```

        thread1.stop();
        thread1= null;
        thread2.stop();
        thread2 = null;
    }

// 3 threads thread
System.out.println("\n\n3\nTime (ms)");
for(int k = 0; k < 400; k++){
    finished = false;
    Thread thread1 = new solver();
    Thread thread2 = new solver();
    Thread thread3 = new solver();

    thread1.start();
    thread2.start();
    thread3.start();

    while(!finished);

    thread1.stop();
    thread1= null;
    thread2.stop();
    thread2 = null;
    thread3.stop();
    thread3= null;
}

// 4 threads thread
System.out.println("\n\n4\nTime (ms)");
for(int k = 0; k < 400; k++){
    finished = false;
    Thread thread1 = new solver();
    Thread thread2 = new solver();
    Thread thread3 = new solver();
    Thread thread4 = new solver();

    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();

    while(!finished);

    thread1.stop();
    thread1= null;
    thread2.stop();
    thread2 = null;
    thread3.stop();
    thread3= null;
    thread4.stop();
    thread4 = null;
}

// 5 threads thread
System.out.println("\n\n5\nTime (ms)");
for(int k = 0; k < 400; k++){

    finished = false;
    Thread thread1 = new solver();
    Thread thread2 = new solver();
    Thread thread3 = new solver();
    Thread thread4 = new solver();
    Thread thread5 = new solver();

    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();
    thread5.start();

    while(!finished);

    thread1.stop();
    thread1= null;
    thread2.stop();
    thread2 = null;
    thread3.stop();
    thread3= null;
    thread4.stop();
    thread4 = null;
    thread5.stop();
    thread5 = null;
}

```

```

}

// 6 threads thread
System.out.println("\n\n6\nTime (ms)");
for(int k = 0; k < 400; k++){
    finished = false;
    Thread thread1 = new solver();
    Thread thread2 = new solver();
    Thread thread3 = new solver();
    Thread thread4 = new solver();
    Thread thread5 = new solver();
    Thread thread6 = new solver();

    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();
    thread5.start();
    thread6.start();

    while(!finished);

    thread1.stop();
    thread1 = null;
    thread2.stop();
    thread2 = null;
    thread3.stop();
    thread3 = null;
    thread4.stop();
    thread4 = null;
    thread5.stop();
    thread5 = null;
    thread6.stop();
    thread6 = null;
}

// 7 threads thread
System.out.println("\n\n7\nTime (ms)");
for(int k = 0; k < 400; k++){
    finished = false;
    Thread thread1 = new solver();
    Thread thread2 = new solver();
    Thread thread3 = new solver();
    Thread thread4 = new solver();
    Thread thread5 = new solver();
    Thread thread6 = new solver();
    Thread thread7 = new solver();

    thread1.start();
    thread2.start();
    thread3.start();
    thread4.start();
    thread5.start();
    thread6.start();
    thread7.start();

    while(!finished);

    thread1.stop();
    thread1 = null;
    thread2.stop();
    thread2 = null;
    thread3.stop();
    thread3 = null;
    thread4.stop();
    thread4 = null;
    thread5.stop();
    thread5 = null;
    thread6.stop();
    thread6 = null;
    thread7.stop();
    thread7 = null;
}

// 8 threads thread
System.out.println("\n\n8\nTime (ms)");
for(int k = 0; k < 400; k++){
    finished = false;
    Thread thread1 = new solver();
    Thread thread2 = new solver();
    Thread thread3 = new solver();
    Thread thread4 = new solver();
    Thread thread5 = new solver();
    Thread thread6 = new solver();
    Thread thread7 = new solver();
    Thread thread8 = new solver();

    thread1.start();

```

```

        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();
        thread6.start();
        thread7.start();
        thread8.start();

        while(!finished);

        thread1.stop();
        thread1 = null;
        thread2.stop();
        thread2 = null;
        thread3.stop();
        thread3 = null;
        thread4.stop();
        thread4 = null;
        thread5.stop();
        thread5 = null;
        thread6.stop();
        thread6 = null;
        thread7.stop();
        thread7 = null;
        thread8.stop();
        thread8 = null;
    }

    // 9threads thread
    System.out.println("\n\n9\nTime (ms)");
    for(int k = 0; k < 400; k++){
        finished = false;
        Thread thread1 = new solver();
        Thread thread2 = new solver();
        Thread thread3 = new solver();
        Thread thread4 = new solver();
        Thread thread5 = new solver();
        Thread thread6 = new solver();
        Thread thread7 = new solver();
        Thread thread8 = new solver();
        Thread thread9 = new solver();

        thread1.start();
        thread2.start();
        thread3.start();
        thread4.start();
        thread5.start();
        thread6.start();
        thread7.start();
        thread8.start();
        thread9.start();

        while(!finished);

        thread1.stop();
        thread1 = null;
        thread2.stop();
        thread2 = null;
        thread3.stop();
        thread3 = null;
        thread4.stop();
        thread4 = null;
        thread5.stop();
        thread5 = null;
        thread6.stop();
        thread6 = null;
        thread7.stop();
        thread7 = null;
        thread8.stop();
        thread8 = null;
        thread9.stop();
        thread9 = null;
    }

    // 10 threads thread
    System.out.println("\n\n10\nTime (ms)");
    for(int k = 0; k < 400; k++){
        finished = false;
        Thread thread1 = new solver();
        Thread thread2 = new solver();
        Thread thread3 = new solver();
        Thread thread4 = new solver();
        Thread thread5 = new solver();
        Thread thread6 = new solver();
        Thread thread7 = new solver();
        Thread thread8 = new solver();
    }

```

```

Thread thread9 = new solver();
Thread thread10 = new solver();

thread1.start();
thread2.start();
thread3.start();
thread4.start();
thread5.start();
thread6.start();
thread7.start();
thread8.start();
thread9.start();
thread10.start();

while(!finished);

thread1.stop();
thread1 = null;
thread2.stop();
thread2 = null;
thread3.stop();
thread3 = null;
thread4.stop();
thread4 = null;
thread5.stop();
thread5 = null;
thread6.stop();
thread6 = null;
thread7.stop();
thread7 = null;
thread8.stop();
thread8 = null;
thread9.stop();
thread9 = null;
thread10.stop();
thread10 = null;
}
}

static int[][] setUpBoard(int puzzleChoice) {
    // Set up a new board, all formatted to 0
    int[][] newBoard = new int[9][9]; // default 0 vals
    int i,j, n=0;

    // Puzzles are entered in standard format
    int[] puzzle = {1,0,0,0,0,0,0,0,2,0,9,0,4,0,0,0,5,0,0,0,6,0,0,0,7,0,0,0,5,0,9,0,3,0,0,0,0,0,0,7,0,0,0,0,0,0,8,5,0,0,4,0,7,0,0,0,0,0,6,0,0,0,3,0,0,0,9,0,8,0,0,0,2,0,0,0,0,0,0};

    for(i = 0; i < 9; i++)
        for(j = 0; j < 9; j++)
            newBoard[i][j] = puzzle[n++];

    // printBoard(newBoard);

    return newBoard;
}

static boolean solve(int row, int col, int[][] board, int xTimes, int startV) {
    // Return true if we have reached the depth of the recursion
    if(xTimes == 81) return true;
    if(finished) return true;

    // Do a loop of rows and columns
    if (++col == 9){
        col = 0;
        if(++row == 9)
            row = 0;
    }

    if (board[row][col] != 0){ // skip filled cells
        return solve(row ,col, board, xTimes+1, startV);
    }

    // The code can only get here if the cell was 0
    for (int val = 1; val <= 9; ++val) {
        if(++startV == 10) startV = 1;

```

```

        // first check of the value is allowed here
        if (allowedHere(row,col,startV,board)) {
            board[row][col] = startV; // If allowed, record it and run recursively
            if(solve(row ,col, board, xTimes+1, startV))
                return true;
        }
    }

    board[row][col] = 0; // reset on backtrack
    return false;
}

static boolean allowedHere(int row, int col, int value, int[][] board) {
    int i;

    // scan 9 neighboring possibilities
    for(i = 0; i < 9; i++){
        // look at columns in this row
        if(board[row][i] == value)
            return false;
        // look at rows in this column
        if(board[i][col] == value)
            return false;
        // look at sub square
        if (board[row/3*3+i%3][col/3*3+i/3] == value)
            return false;
    }
    return true; // no violations, so it's legal
}

static void printBoard(int[][] boardToPrint) {
    int i, j;

    for (i = 0; i < 9; i++) {
        if (i%3 == 0)
            System.out.println("-----");

        for (j = 0; j < 9; j++) {
            if (j%3 == 0) System.out.print(" | ");
            if (boardToPrint[i][j] == 0)
                System.out.print(" * ");
            else
                System.out.print(Integer.toString(boardToPrint[i][j]) + " ");
        }
        System.out.println(" | ");
    }
    System.out.println("-----");
}
}

```

## APPENDIX II: RAW DATA

Here is the raw data gathered by the experiment. This first row, in bold, indicated the number of threads used in the test run. The second column is the average time it took to solve the puzzle while the experiment. The following columns are the individual time data harvested by repeatedly performing the test.

Note: due to the random nature of the data collected, repeating the experiment should yield a similar average, but different individual values.

<b>1</b>	<b>2</b>	<b>3</b>	<b>4</b>	<b>5</b>	<b>6</b>	<b>7</b>	<b>8</b>	<b>9</b>	<b>10</b>
290	158	242	292	190	105	133	96	113	109
						Time (ms)			
1124	15	31	265	15	15	16	16	16	16
15	15	16	47	16	15	15	15	15	15
78	16	94	16	15	16	47	31	16	16
78	16	78	16	15	15	16	15	16	15
375	15	31	16	32	16	16	31	31	19
15	16	31	124	1515	172	16	31	281	702
31	15	62	15	15	15	31	16	140	172
16	15	156	31	16	16	16	16	31	16
16	78	32	16	15	16	15	15	125	15
46	16	63	16	16	15	16	16	15	47
47	16	3167	218	31	16	15	31	15	31
78	63	17	218	31	31	16	16	15	31
78	1357	16	11	16	32	31	15	31	203
31	1030	15	17	16	15	15	16	250	468
62	48	16	795	15	15	1	16	93	17
47	63	15	17	15	16	16	16	78	31
328	62	32	795	31	16	15	16	141	16
15	47	15	1950	16	15	16	15	16	16
78	62	78	16	16	62	15	32	78	15
47	47	187	16	15	16	31	15	16	203
63	156	78	15	16	16	16	16	15	203
31	62	156	17	16	16	16	32	31	15
47	62	670	16	1030	16	15	15	18	1451
15	1108	1077	16	15	16	16	15	15	177
156	1616	16	1616	79	15	16	15	32	16
47	110	32	15	16	32	3214	16	218	16
47	484	32	15	31	31	3214	15	16	31
47	15	15	141	16	140	15	16	16	31
46	63	32	16	16	16	203	16	16	31
32	93	16	47	125	15	31	15	16	31
62	93	15	16	31	16	31	31	15	31
47	47	16	16	16	16	140	15	32	31
47	16	15	31	31	15	47	16	15	31
62	31	16	31	15	16	47	15	15	31
47	16	16	390	125	15	47	15	15	21
47	16	15	561	93	16	47	31	16	132
46	15	16	444	32	281	140	328	15	16
63	15	1341	93	16	1982	32	47	16	15
47	93	265	203	16	312	312	15	15	16
1341	15	265	125	15	16	31	16	15	63
1685	16	249	531	31	16	31	16	16	156
15	16	31	16	15	16	31	140	15	3264
32	281	31	15	15	15	218	31	31	15
78	15	1529	15	31	15	16	31	16	15
156	16	16	16	16	266	31	31	16	32
16	16	16	109	15	281	15	31	31	32
406	15	16	15	500	1888	16	125	16	16
15	16	15	16	1451	32	31	125	15	15
16	16	16	16	499	16	31	31	32	31
15	15	16	327	16	31	16	47	15	15
16	16	110	2075	16	16	31	327	1529	31
140	16	936	2075	16	15	16	327	6	16
31	16	78	16	31	16	16	140	15	31
16	31	47	218	17	109	16	218	156	16
31	203	78	15	16	624	15	93	109	15
63	718	1.93E+02	234	31	421	16	47	15	16
47	687	15	15	15	219	15	109	15	16
156	577	16	16	16	15	16	31	7	16
1029	718	15	16	15	47	16	15	31	16
47	687	951	16	16	15	15	16	16	31
47	577	391	187	15	171	16	15	31	15
47	16	391	343	16	16	15	15	16	2262
16	31	125	377	63	16	16	16	31	15
109	16	32	16	15	15	16	16	31	32
47	15	250	1217	421	141	16	16	172	16
46	16	31	2	203	15	16	15	16	16
47	15	1061	64	171	16	62	16	109	15

63	32	15	15	171	16	31	15	203	15
62	15	15	172	780	16	16	15	93	16
47	16	16	15	16	47	16	16	16	16
47	1653	78	312	62	31	15	16	31	16
62	250	78	94	62	31	15	15	31	15
16	484	78	16	47	78	32	15	433	32
31	16	16	94	16	78	343	15	1872	17
15	31	78	93	31	31	359	16	2	15
33	16	297	78	31	16	78	16	16	16
16	16	436	163	109	124	78	31	15	15
15	16	436	78	156	31	172	16	16	16
15	15	187	32	109	47	203	15	16	436
15	16	1029	78	78	16	561	16	16	16
203	78	125	78	16	125	156	15	31	15
1045	15	125	78	15	16	31	16	16	31
546	16	3074	93	16	93	514	16	16	16
47	16	1607	31	31	234	141	15	15	15
811	15	1607	93	16	16	546	16	16	31
671	16	140	31	15	16	16	15	16	312
1170	15	32	109	32	15	16	16	16	94
16	31	188	31	15	16	172	16	1	140
32	16	171	62	3120	15	16	15	15	31
16	31	63	15	31	15	171	10	16	31
218	109	62	24	47	16	171	10	15	31
1217	171	31	33	16	16	16	15	15	16
438	33	31	33	16	16	16	16	16	31
31	811	109	109	15	62	827	452	15	16
1763	62	109	15	16	406	16	234	16	16
281	1045	94	2028	15	31	16	203	15	15
1311	281	312	16	187	78	15	608	32	15
2215	281	140	58	16	47	15	16	15	15
31	1747	16	47	16	78	31	15	16	16
1763	1108	936	32	156	31	16	31	16	15
281	343	94	32	499	125	16	15	15	15
1311	343	31	31	593	982	15	16	16	31
2215	125	109	640	16	982	16	16	31	16
343	343	16	640	63	15	32	16	46	31
47	125	16	1872	16	32	234	15	31	16
47	125	468	640	16	15	561	15	16	31
47	16	125	1872	32	31	1420	31	31	31
62	15	78	125	15	31	16	1731	16	16
47	16	624	73	16	16	718	733	31	15
2995	109	187	15	16	16	110	62	31	62
63	173	844	16	16	125	15	15	16	31
125	125	156	15	16	31	16	31	15	16
31	125	140	16	16	31	31	31	219	32
343	16	16	31	47	31	15	15	15	15
4134	16	16	31	15	16	16	453	218	15
2512	15	94	16	16	110	281	172	47	16
460	16	250	15	15	78	1529	1248	203	16
460	15	250	16	16	47	15	31	158	16
460	47	16	160	31	16	15	31	31	15
460	281	172	1616	15	47	15	17	31	16
46	265	47	15	16	16	15	171	249	16
16	1077	48	15	15	16	437	359	32	15
156	142	31	16	16	16	15	359	31	16
15	1139	78	16	16	15	296	670	15	1498
16	94	62	15	16	16	16	31	32	193
15	15	32	15	15	15	219	15	31	31
32	31	109	16	32	280	15	15	15	31
15	78	16	327	16	406	16	15	32	171
16	140	46	3089	31	15	234	16	2636	468
15	16	110	1332	15	31	16	15	2636	16
16	32	297	16	2325	16	47	16	889	16
16	141	109	31	1248	15	156	32	31	16
686	16	109	343	16	16	16	32	31	16
1170	16	312	16	16	109	47	16	156	15
31	78	109	15	15	187	16	31	16	296
125	47	109	32	16	32	31	687	78	296
46	16	312	15	31	31	16	32	390	31
16	78	172	15	64	31	15	172	16	164
16	47	172	16	94	141	47	78	16	842
15	15	172	62	140	187	173	31	15	15
16	63	592	62	15	31	31	16	16	33
15	15	187	16	15	47	15	16	11	15
16	47	1264	15	156	31	94	31	1515	15
16	62	94	16	265	47	62	15	16	16
15	32	2293	16	844	31	94	15	16	16
16	31	16	16	125	828	62	1763	15	16
15	31	125	15	780	16	94	15	16	47
17	47	78	15	31	31	62	31	15	15
31	46	140	47	140	31	546	172	31	16
16	47	951	16	156	31	32	171	15	47
15	31	171	16	2980	31	16	16	62	421
2278	47	140	15	187	203	15	31	15	905
1045	47	219	32	187	78	16	64	16	218
655	297	93	1467	2200	16	15	15	15	16
172	282	47	486	31	31	16	16	16	109



187	280	79	904	125	1616	78	156	15	16
375	16	78	170	3089	78	16	406	15	31
125	16	94	203	47	78	156	15	15	15
124	31	78	1341	16	31	32	16	16	16
78	16	140	31	16	111	16	16	15	16
125	171	31	345	16	16	78	15	16	47
125	31	93	16	15	16	47	16	15	31
16	47	93	1186	32	16	47	16	15	16
78	64	156	15	31	16	125	1763	16	15
1185	46	2168	15	16	15	109	15	16	15
124	47	48	16	156	16	31	31	47	16
390	62	62	1515	390	16	31	172	31	16
15	78	62	16	889	15	187	171	16	63
327	47	32	124	889	16	16	16	15	31
171	62	31	359	16	31	31	31	16	15
203	47	15	254	31	31	15	64	530	16
187	31	16	125	140	31	327	15	196	31
47	16	16	3123	156	31	374	16	1045	31
47	171	15	15	2980	203	16	156	16	218
47	48	16	15	187	78	78	406	16	62
111	47	16	31	187	16	16	15	16	15
93	16	16	343	2200	31	31	16	15	178
111	15	32	125	31	1616	32	16	16	79
93	16	32	16	125	78	16	15	15	31
16	15	32	15	3089	78	15	16	31	203
15	15	15	231	47	31	16	16	15	15