SUDOKU SOLVER

Project Write-Up

By Adrián Gómez Lamuedra

1. Introduction

Parallelization is a programming technique that can be used for many purposes. In this case, we will use it for an interesting and entertaining one, checking if a solution of a sudoku is valid. We will first implement a sequential version of the program, then we will parallelize more zones of the code and observe how it affects the timing.

2. Develop environment and libraries used

The four programs that will be shown in the write-up were developed in the language C++, with the environment Visual Studio. All the code was fully developed by the author, although the input sudokus were taken from the internet.

These were the libraries used throughout the program:

<iostream>: Input and output handling

<vector>: The creation of the matrix of the sudoku and the thread vectors

<chrono>: Getting the runtime of the program

<thread>: Creation and joining of threads

<mutex>: Locking the global variable wrong, that ends all the threads when a mistake is found

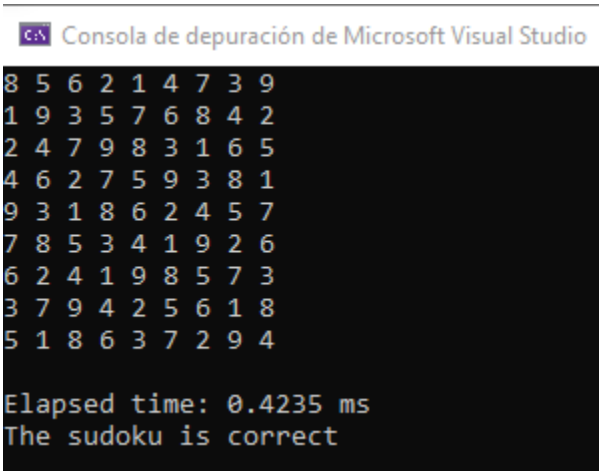
<unordered_set>: Keeping track of all the numbers that have already been used in a row, column or box

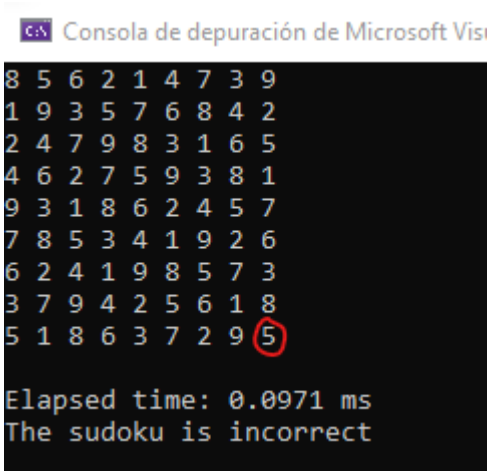
3. Sequential code

The first code is a very simple one that checks row by row, column by column and 3x3 box by 3x3 box if there is a number that is repeated or out of the range 1-9. For doing so, nested loops are used in an independent way.

The cost of the program in time should be O(3*N^2), but it is not very high as N= 9 in our case. That factor will affect our expected results in the runtime.

This are examples of the execution:





The code of the program can be found in the archive “Sudoku Sequential.cpp”.

4. First Optimization

The first optimization that appears in our mind that includes parallelization is to search for a mistake in rows, columns and boxes all simultaneously. For doing so, we must now include one thread for each of the functions that check them:

```
thread first(rowVal, ref(m)); // Pass the reference of m and not m itself (the value)
thread second(colVal, ref(m));
thread third(genBoxVal, ref(m));

first.join();
second.join();
third.join();
```

We will also create a global variable to exit the 3 threads whenever there is a mistake in either row, column or box. This a fragment of the row checking, but the others are very similar:

```
    while (i < x + 3 && !wrong) { // We check at every moment this thread or the others have found a mistake
        j = y;
        while (j < y + 3 && !wrong) {
            int k = m[i][j];
            if (k < 1 || s.count(k) > 0 || k > 9) {
                mu.lock(); // Acquire the lock to change the global variable
                wrong = true; // There is a number repeated
                mu.unlock(); // Release the lock just after the variable is modified
            }
```

The output of the program is this one:

```
2 3 7 8 4 1 5 6 9
1 8 6 7 9 5 2 4 3
5 9 4 3 2 6 7 1 8
3 1 5 6 7 4 8 9 2
4 6 9 5 8 2 1 3 7
7 2 8 1 3 9 4 5 6
6 4 2 9 1 8 3 7 5
8 5 3 4 6 7 9 2 1
9 7 1 2 5 3 6 8 4

Elapsed time: 8.122 ms
The sudoku is correct
```

```
2 3 7 8 4 1 5 6 9
1 8 6 7 9 5 2 4 3
5 9 4 3 2 6 7 1 8
3 1 5 6 7 4 8 9 2
4 6 9 5 8 2 1 3 7
7 2 8 1 3 9 4 5 6
6 4 2 9 1 8 3 7 5
8 5 3 4 6 7 9 2 1
9 7 1 2 7 3 6 8 4

Elapsed time: 7.262 ms
The sudoku is incorrect
```

Theoretically, the code is now in a timing of order O(N^2), less than the sequential one. But the creation of threads is more costful than the time we gain after the optimization. This should not be the case with a very big N, but it would be a very difficult case to test.

The rest of the code can be found in the archive “Sudoku Optimization 1.cpp”.

5. Second Optimization

For applying the second optimization we first must consider that the box checking is done in two functions apart, one that traverses the boxes and another one that explores each one. A very simple change can make every checking of the box be done in parallel by different threads:

```
void genBoxVal(tMatrix& const m) { // Checks that the 9 boxes have numbers from 1 to 9
    int i = 0, j = 0;
    vector<thread> threads;
    while (i < N && !wrong) {
        j = 0;
        while (j < N && !wrong) {
            threads.push_back(thread(boxVal, ref(m) , i, j)); // We are now creating a different thread for each box
            j += 3;
        }
        i += 3;
    }
    for (thread& t : threads) t.join();
}
```

This simple change makes every box be processed in parallel, yet, the results are again not what we expected:

```
1 3 2 5 6 7 9 4 8
5 4 6 3 8 9 2 1 7
9 7 8 2 4 1 6 3 5
2 6 4 9 1 8 7 5 3
7 1 5 6 3 2 8 9 4
3 8 9 4 7 5 1 2 6
8 5 7 1 2 3 4 6 9
6 9 1 7 5 4 3 8 2
4 2 3 8 9 6 5 7 1

Elapsed time: 12.2883 ms
The sudoku is correct
```

```
1 3 2 5 6 7 9 4 8
5 4 6 3 8 9 2 1 7
9 7 8 2 4 1 6 3 5
2 6 4 9 1 8 7 5 3
7 1 5 6 3 2 8 9 4
3 8 9 4 7 5 1 2 6
8 5 7 1 2 3 4 6 9
6 9 1 7 5 4 3 8 2
4 2 3 8 9 6 5 7 9

Elapsed time: 5.3999 ms
The sudoku is incorrect
```

However, we can observe that the timings are quite different, because the parallelization of the box part made possible a quicker identification of the wrong number. Again, this improvement would be more noticeable in a code with a much bigger N.

The rest of the code can be found in the archive “Sudoku Optimization 2.cpp”.

6. Third Optimization

The third optimization seems very natural after having implemented the second one. After having made a thread for every box checking, we may think that the same can be done to rows and columns. However, this change is not so straightforward as the one with the boxes. First, we should create another two functions that traverse the rows and columns respectively:

```
void genRowVal(tMatrix& const m) { // Checks that every row has the numbers 1 to 9
    int i = 0;
    vector<thread> threads; // We create a thread for every checking every row independently
    while (i < N && !wrong) {
        threads.push_back(thread(rowVal, ref(m), i));
        ++i;
    }
    for (thread& t : threads) t.join();
}
```

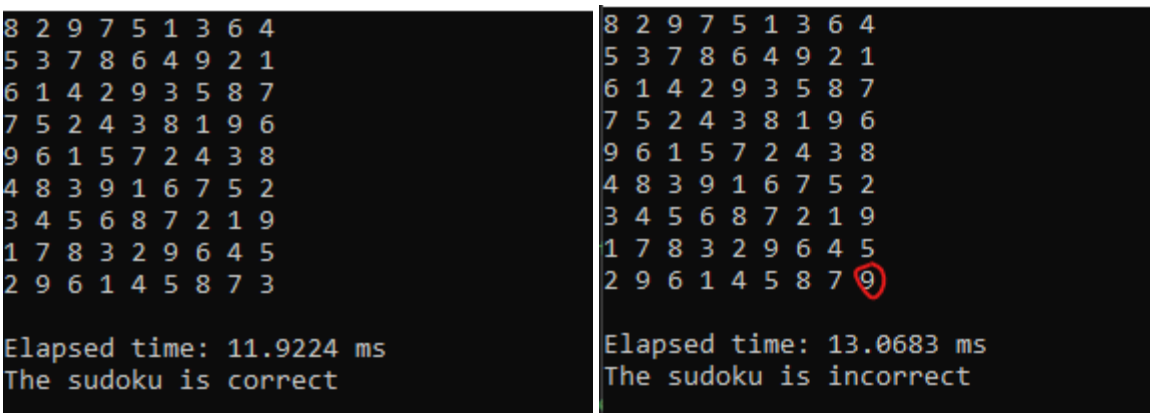
The functions that we already have, rowVal and colVal, will be limited to explore just one row or column (respectively):

```
void rowVal(tMatrix& const m, int i) {
    unordered_set<int> s;
    int j = 0;
    while (j < N && !wrong) { // We check at every moment this thread or the others have found a mistake
        int k = m[i][j];
        if (k < 1 || s.count(k) > 0 || k > 9) {
            mu.lock(); // Acquire the lock to change the global variable
            wrong = true; // There is a number repeated
            mu.unlock(); // Release the lock just after the variable is modified
        }
        else s.insert(k); // No rule is violated, we continue the search
        ++j;
    }
}
```

For the columns it is a very similar process.

After the optimizations 2 and 3, the runtime of the program should be close to the order of O(N), as the program is completely parallelized, so every row, column and box is being checked simultaneouly.

However, we find again inconsistencies in the execution, as the creation of so many threads makes the runtime increase drastically. It is our expectation that with a bigger N the execution results would match up properly with the reasoning:



The rest of the code can be found in the archive “Sudoku Optimization 3”.

7. Conclusion

This program is a clear example of where parallelization should be effective, as almost every part of the code except for the matrix creation and data entering is parallelizable.

However, the simplicity of the problem makes it difficult to match the execution results with the ones that we expected theoretically.

It would be a good practice to try for a sudoku with a million possible numbers to plug in every row, column or box, but for making a valid input for our program we should find a solution to that sudoku, which is impossible.