

# Assignment 2 REPORT

Tushita Sharva, CS21BTECH11022

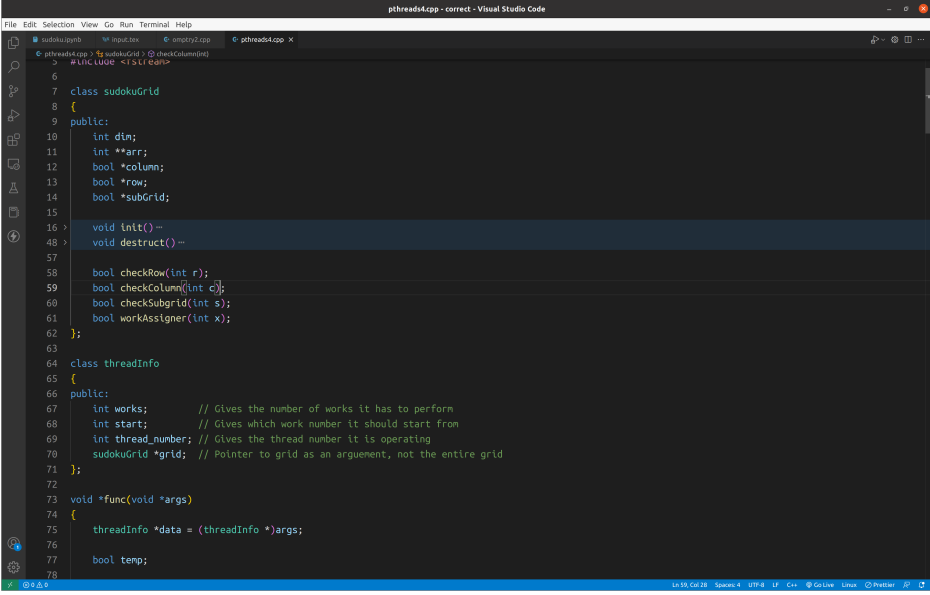
February 3, 2023

# Using pthreads

## Data structures used:

Here I used two classes of objects

- Class `sudokuGrid` has all the operations and data regarding the sudoku grid. It has
  - A multidimensional array of all the elements in the grid
  - An array of boolean values that stores if the columns are valid
  - An array of boolean values that stores if the subgrids are valid
  - An array of boolean values that stores if the rows are valid
  - An `init` operation that initialises the grid
  - An operation that frees all the memory allocated by the `init` function.
  - An operation that checks if a row is valid
  - An operation that checks if a subgrid is valid
  - An operation that checks if a column is valid
  - An operation that combines all the above three operations into a single operation called `workAssigner(int x)`.
- Class `threadInfo` is meant to store data regarding the threads, which are useful for preparing the `output.txt` file. It stores the values of
  - current thread number
  - which work the thread should start working
  - the number of operations it has to perform
  - It also has the pointer to the grid as it's member, so that it can be passed as a parameter to the thread function.



```
6
7 class sudokuGrid
8 {
9 public:
10     int dim;
11     int **arr;
12     bool *column;
13     bool *row;
14     bool *subGrid;
15
16     void init() {
17         // ...
18     }
19     void destruct() {
20         // ...
21     }
22
23     bool checkRow(int r);
24     bool checkColumn(int c);
25     bool checkSubgrid(int s);
26     bool workAssigner(int x);
27 };
28
29 class threadInfo
30 {
31 public:
32     int works; // Gives the number of works it has to perform
33     int start; // Gives which work number it should start from
34     int thread_number; // Gives the thread number it is operating
35     sudokuGrid *grid; // Pointer to grid as an argument, not the entire grid
36 };
37
38 void *func(void *args)
39 {
40     threadInfo *data = (threadInfo *)args;
41     bool temp;
```

Figure 1: Data Structures

## Implementation Details:

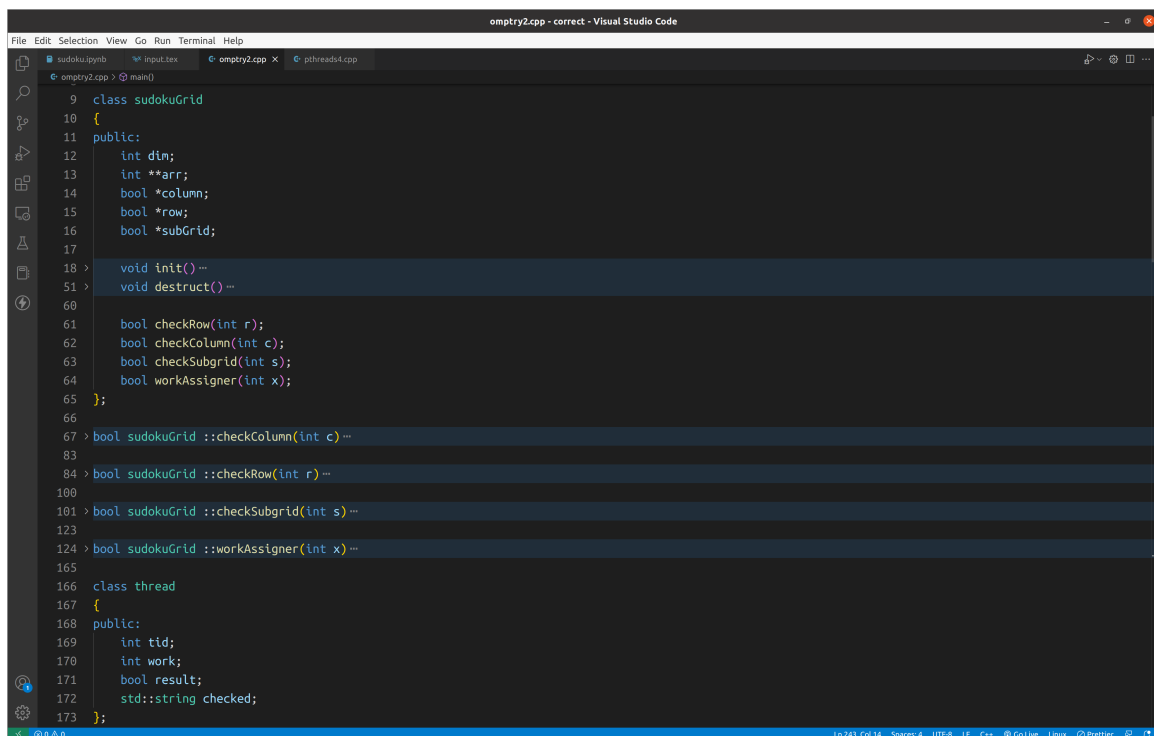
At first, after taking the values of `num_of_threads` and `dimension`, operations such as allocation of memory space, initialising of those values and checking if they are inside the given range are done. Then the threads are initialised. The `num_of_threads` number of threads and `threadInfo` objects are initialised.

Same as before, my implementation does the division of threads this way. If there are `w` number of works and `t` number of threads, then each thread will by default perform  $(w \div t)$  number of operations and the first  $(w \% t)$  threads perform  $(w \div t) + 1$  operations. Accordingly start positions and number of works to perform are given.

Now, with all the data, i.e., where to start and how many to operations to perform, we now create threads and send the `threadInfo` object as parameter to the thread. And all of them are joined and then the necessary comparisons are done to check if the data is proper or not.

## Using openMP

OpenMP simplified a lot of things. First of all, I didn't have to divide the work. It divided all the work by itself. All I needed to store now was the thread number a particular row/column/ subgrid was checked by, what it checked, i.e., a row/ column/ subgrid and what was the result (valid or invalid). So here also I used two data structures but now I had to store relatively less data as compared to pthreads.



```
9 class sudokuGrid
10 {
11 public:
12     int dim;
13     int **arr;
14     bool *column;
15     bool *row;
16     bool *subGrid;
17
18     void init() ~
19     void destruct() ~
20
21     bool checkRow(int r);
22     bool checkColumn(int c);
23     bool checkSubgrid(int s);
24     bool workAssigner(int x);
25 };
26
27 bool sudokuGrid::checkColumn(int c) ~
28
29 bool sudokuGrid::checkRow(int r) ~
30
31 bool sudokuGrid::checkSubgrid(int s) ~
32
33 bool sudokuGrid::workAssigner(int x) ~
34
35 class thread
36 {
37 public:
38     int tid;
39     int work;
40     bool result;
41     std::string checked;
42 };
43
```

Figure 2: Data Structures

All the functions and others were same as those used in pthreads.

## Performance analysis

In my way of implementation openMP were faster than pthreads. One main reason for this to happen is because while using pthreads I had to store a lot of data, from where to start and where to end etc. Whereas in openMP, it had internally implemented an optimised division of work. While using those pthreads, that much amount of data had to be stored and accessed constantly so it took more time and as the amount of data increased, the time taken also increased evidently. Whereas in openMP, there was a nice balanced compensation between increased number of data stored and increased number of concurrent operations. Here are the experimental results obtained.

TIME TAKEN vs. SUDOKU SIZE

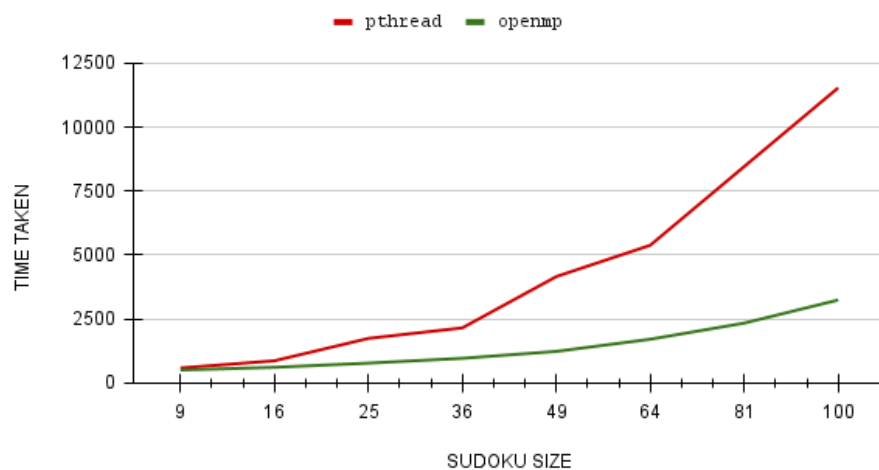


Figure 3: Data Structures

TIME TAKEN vs. NUMBER OF THREADS

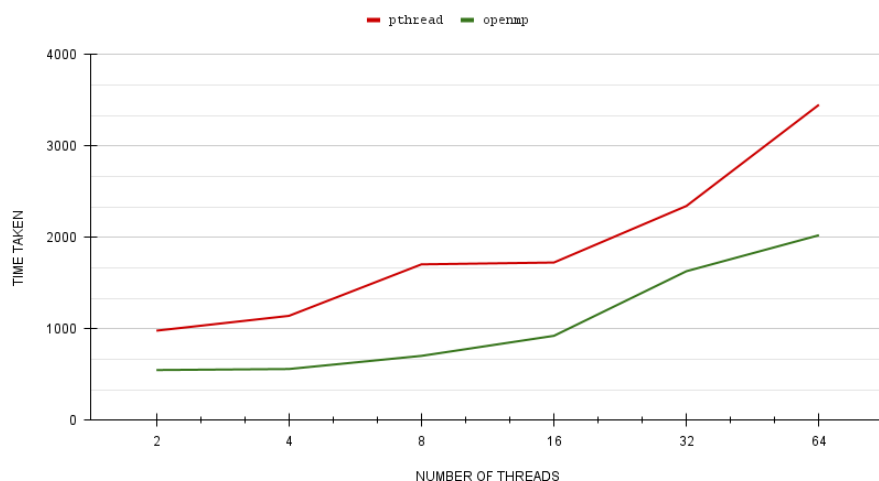


Figure 4: Data Structures