Design of the program

The input for the program is taken from input.txt file and the output is printed to output.txt file. The code is written in C++. The main thread assigns tasks to each child thread. The number of child threads to be created is input to the program.

Each child thread performs one task. A task can be either to verify a row, a column or a grid. The main thread passes a task object to the child thread. The task object contains information whether to verify row, column or grid as well as the number of the row, column or grid to be verified. The task object also contains the thread number on which the task is performed.

In order to verify a row, column or grid, each integer is first checked if it is between 1 and N and if so, it is put into a set. If the size of the set is N after putting in all the elements, then the thread verifies the assigned task. To verify any task, three arrays are created as global variables. First array is for verifying rows, second is for columns and third is for grids. Each array element is 0 or 1, 1 indicating that the task was verified and 0 indicating otherwise. Each child thread modifies these arrays based on the assigned task. The sudoku is valid only if all the elements are 1 in these 3 arrays. The main thread performs this operation of checking each element at the end.

Time for execution is calculated using the chrono package. Main thread reads input from input file and writes output and logs to output file. The sudoku, dimension of the sudoku, number of threads to be used and the three arrays used to store whether a task is verified or not are stored as global variables in the program so that each child thread can access them.

In Pthreads, a pthread array is created. The main thread can either assign a task or wait for a thread to complete. It cannot do both these tasks at the same time. Thus the main thread first assigns tasks to all the threads, then it waits for all the threads to complete execution and then it assigns new tasks to all the threads. This is introducing a barrier in the program. The tasks are assigned in a round-robin fashion until all tasks are completed. A better option would be to use a thread pool.

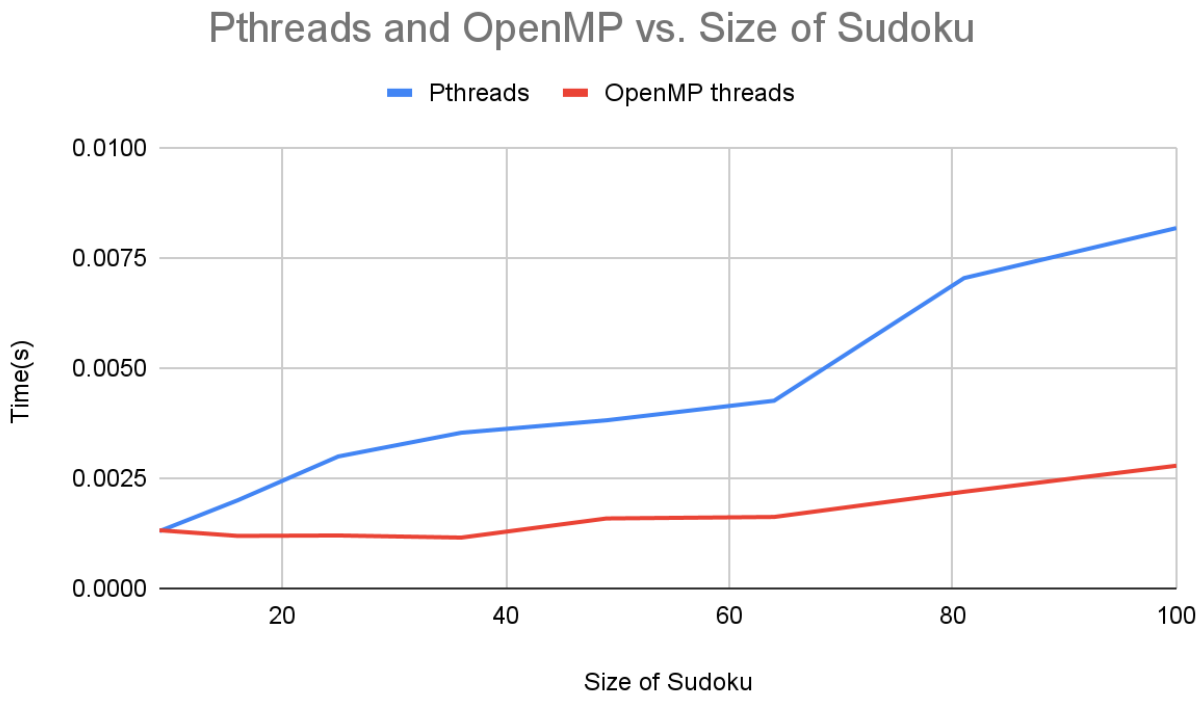
In OpenMP, the following code is used to perform multithreading. I used OpenMP to parallelize a for loop. The number of threads is set to the given number of threads to be used. Every iteration of the for loop is responsible for verifying 1 task. The tasks are stored in the “tasks” array. The “validator” function verifies the given task.

```
#pragma omp parallel for num_threads(n_threads)
        for(int i=0; i<3*dim; i++)
        {
            tasks[i].thread_num = omp_get_thread_num();
            validator( assigned_task: &tasks[i]);
        }
```

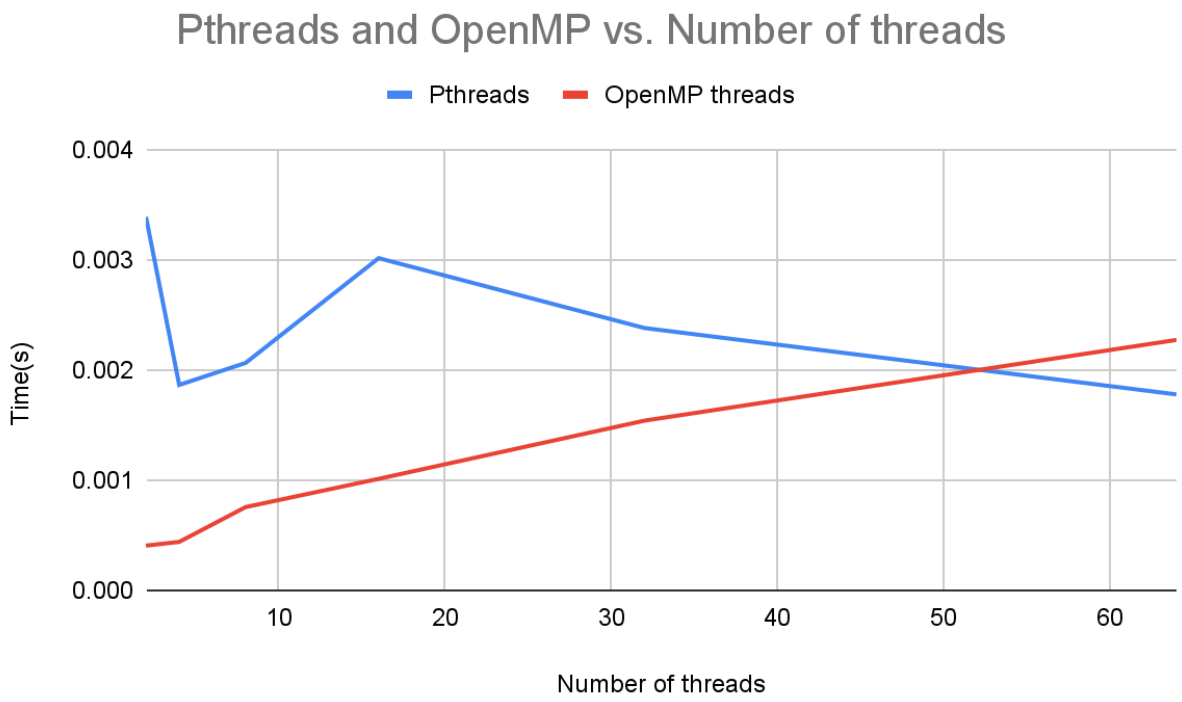
For more implementation details, please see the code comments.

Analysis of the results and plots

The following is the plot to compare performance of Pthreads and OpenMP, where the number of threads is kept fixed at 16 and the size of sudoku is varied from 9x9 to 100x100.



The following is the plot to compare performance of Pthreads and OpenMP, where the size of sudoku is kept fixed at 25x25 and the number of threads is varied from 2 to 64.



From the first plot, we can see that the time for both Pthreads and OpenMP increase as the size of sudoku increases. However, OpenMP takes less time as it uses thread pool internally but Pthread does not use thread pool. I am introducing a barrier in the multithreaded code. Therefore Pthreads is taking more time.

From the second plot, we can see that as the number of threads increases, the time for Pthread decreases but the time for OpenMP increases. This is because the overhead introduced by OpenMP to decide which thread performs which task is relatively high in this problem because each individual task is very small. The time for Pthreads is decreasing because of the barrier. As the number of threads increases, we are encountering the barrier a lesser number of times. Thus performance is improving.