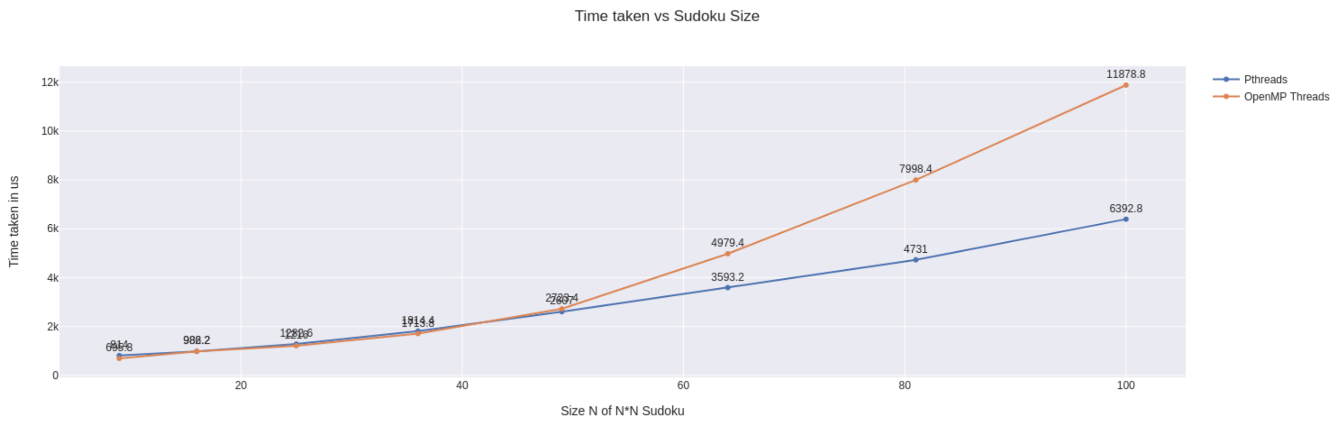
Comparative Analysis of Execution Times of PThread and OpenMP code

Task 1

In the first experiment, the x-axis will be the size of the sudoku varying from 9*9 to 100*100. The y-axis will show the time taken. Keep the total number of threads fixed at 16 for this experiment

Results

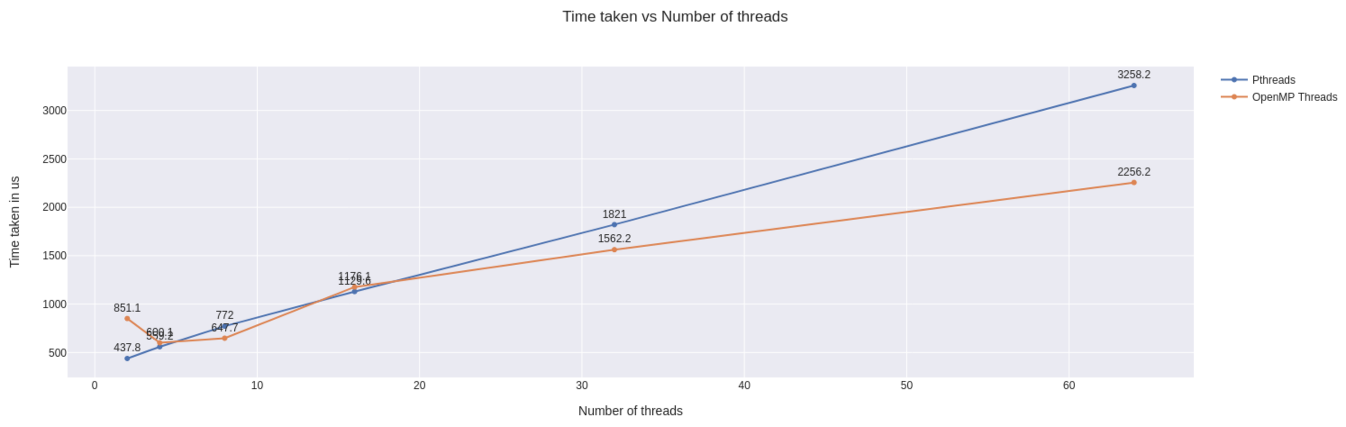


We can see that for smaller sizes, both approaches take nearly the same time. However later on Pthreads become more efficient because of pre-defined work load distribution with them as compared to OpenMP (which will take some additional time to allocate and run on the fixed number of threads). This is despite the less balanced distribution of work across threads.

Task 2

For the second experiment, the x-axis will consist of the number of threads varying from 2 to 64 in the powers of 2 (5 points). The y-axis will again show the time taken. Keep the suoku size to be fixed at 25 ∗ 25 for all the experiments.

Results



We can see that initially code written with Pthreads performs the same/ better than OpenMP. However later on OpenMP threads perform better because of a more balanced distribution than pthreads approach.

Explanation

The discrepancies in the code can be explained in terms of the distribution of the workload among the threads that are being created.

For Threads created with the PThreads library, workload has to be distributed manually and is thus more uneven as compared to the OpenMP implementation (possible cause of results of Task 2). This takes less time than manually allocating a job to a thread at time of creation (a possible reason for results of Task 1)

Distribution follows following logic :
For PThreads,

- Allocate K/3 threads each to Row checks, column checks and box/grid checks. We are still left with 0, 1 or 2 threads (the K%3 value) of threads that couldn't be distributed between the 3 tasks. If 1 is left, allocate that to checking rows, if 2 are left then allocate that one each to row checks and column checks.
- Each type of check (Row, Column or Box) has N tasks associated to it. Distribute these evenly to the number of threads obtained in the above process.

This can be very efficient when K comes near multiples of 3. However it can simultaneously be inefficient in cases like : 9×9 grid with 5 threads. This gives 2 threads to rows and columns, 1 to box checks. The row and column threads check 4 or 5 iterations but the box thread check all 9 boxes by itself. In larger thread counts, similar conditions can happen where the average thread checks <3 rows or columns, but a few threads reach O(N) checks. This becomes a bottleneck then.

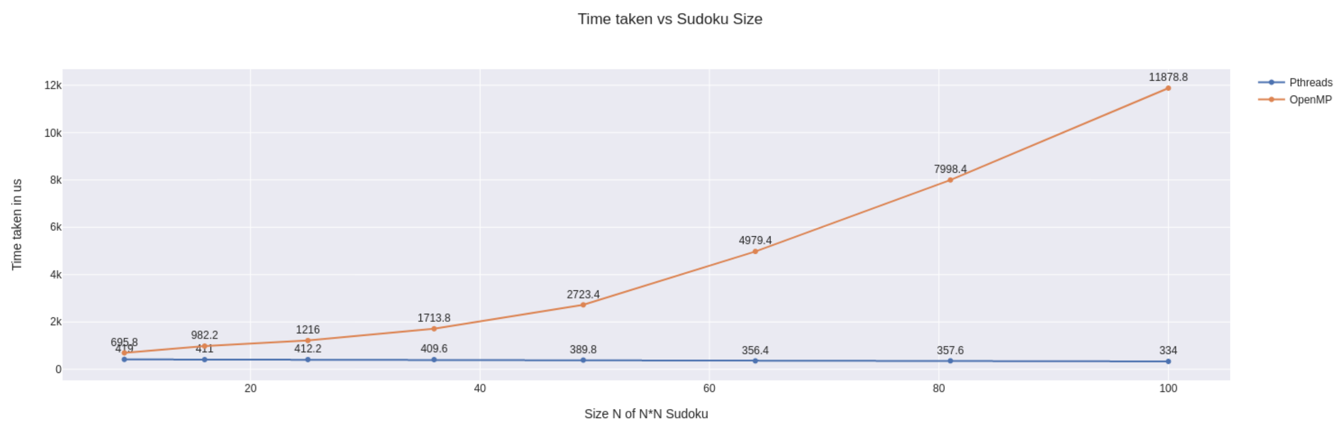
For OpenMP Threads,

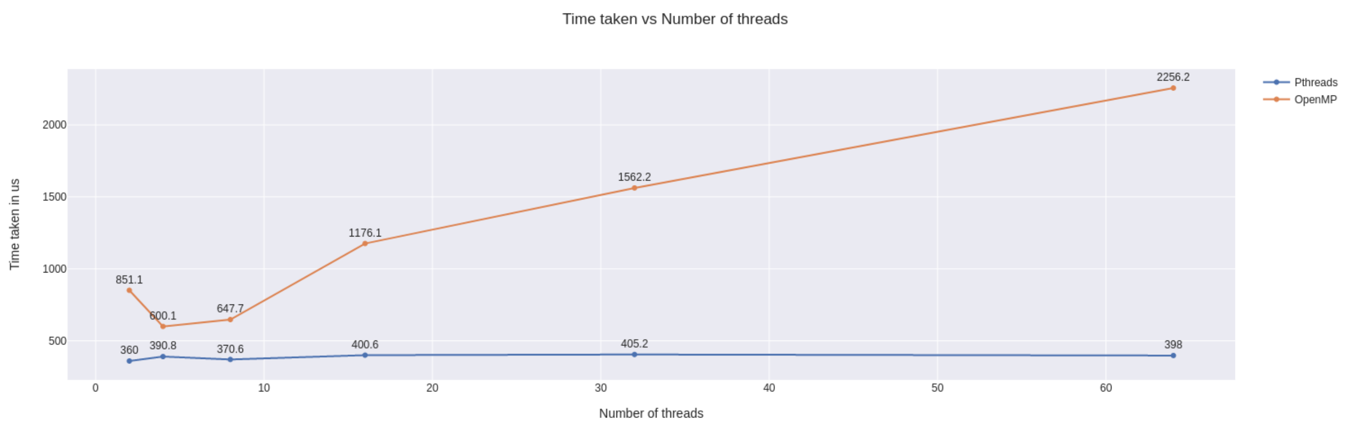
- Create a unit of work to be given to threads. Total N units. Each unit consists of 1 row check , 1 column check and 1 box check.
- This can lead to under-utilization of threads when the number of threads is much larger than N (in case when K is 64, N is 25, only 25 threads end up being used giving a larger compute time in comparison to Open Mp where the entire 64 threads are being used in some manner)

Extra Credit

These are the plots for adding a return statement from threads in pthreads upon reaching an invalid check for a row/ column/ box. Uncommeting at rows 228, 260, 298 to have the following makes the thread return early on seeing a duplicate in a row/column or box respectively.

```
return NUll;
```





EP19BTECH11002, Divyansh Kharbanda