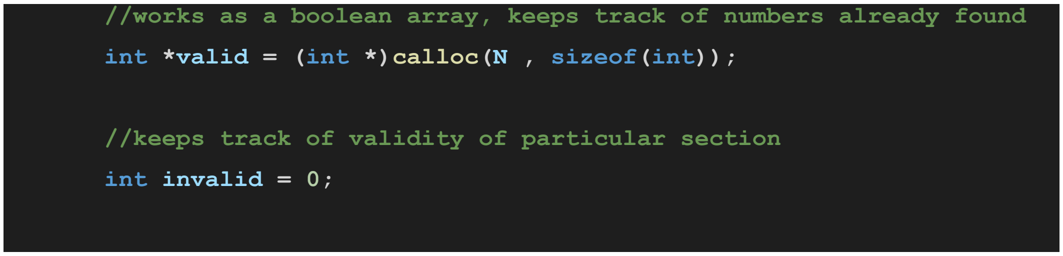
Sudoku Validator :

For validating a completed sudoku , We need to Check each row , column and grid individually for its validity

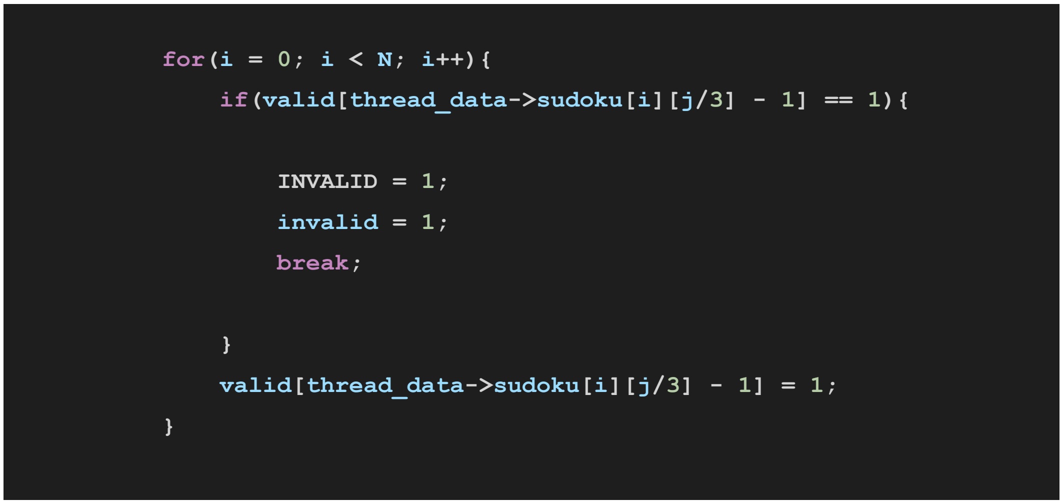
Validator Implementation :

- At first we create at **valid** array of length N which keeps tracks of each number found as we check each section , also a boolean **invalid** - initially 0 , but as we find a section is invalid , it changes to 1 and we break through the checker loop .



```
//works as a boolean array, keeps track of numbers already found
int *valid = (int *)calloc(N , sizeof(int));

//keeps track of validity of particular section
int invalid = 0;
```

1. **Row checker :**



```
    for(i = 0; i < N; i++){
        if(valid[thread_data->sudoku[i][j/3] - 1] == 1){

            INVALID = 1;
            invalid = 1;
            break;

        }
        valid[thread_data->sudoku[i][j/3] - 1] = 1;
    }
```

- This iterates through a row .
- For each element found , the value at index (element - 1) in valid array changes to 1 from 0.
- if for any element valid array is already 1 , it implies that the row is invalid - then the bool invalid is changed to one and we break.

- As we jump outside the loop , if a row is valid then the bool invalid remains 0 else it goes to 1.
- using this we print desired Output in log file

```
            // prints valid output for each row
            if(invalid == 0){
                fprintf(MainOut_fp, "Thread %d checks the row %d and is
valid.\n" , thread_data->thread_no , j/3 + 1);
            }
            else{
                fprintf(MainOut_fp, "Thread %d checks the row %d and is
invalid.\n" , thread_data->thread_no , j/3 + 1);

            }
```

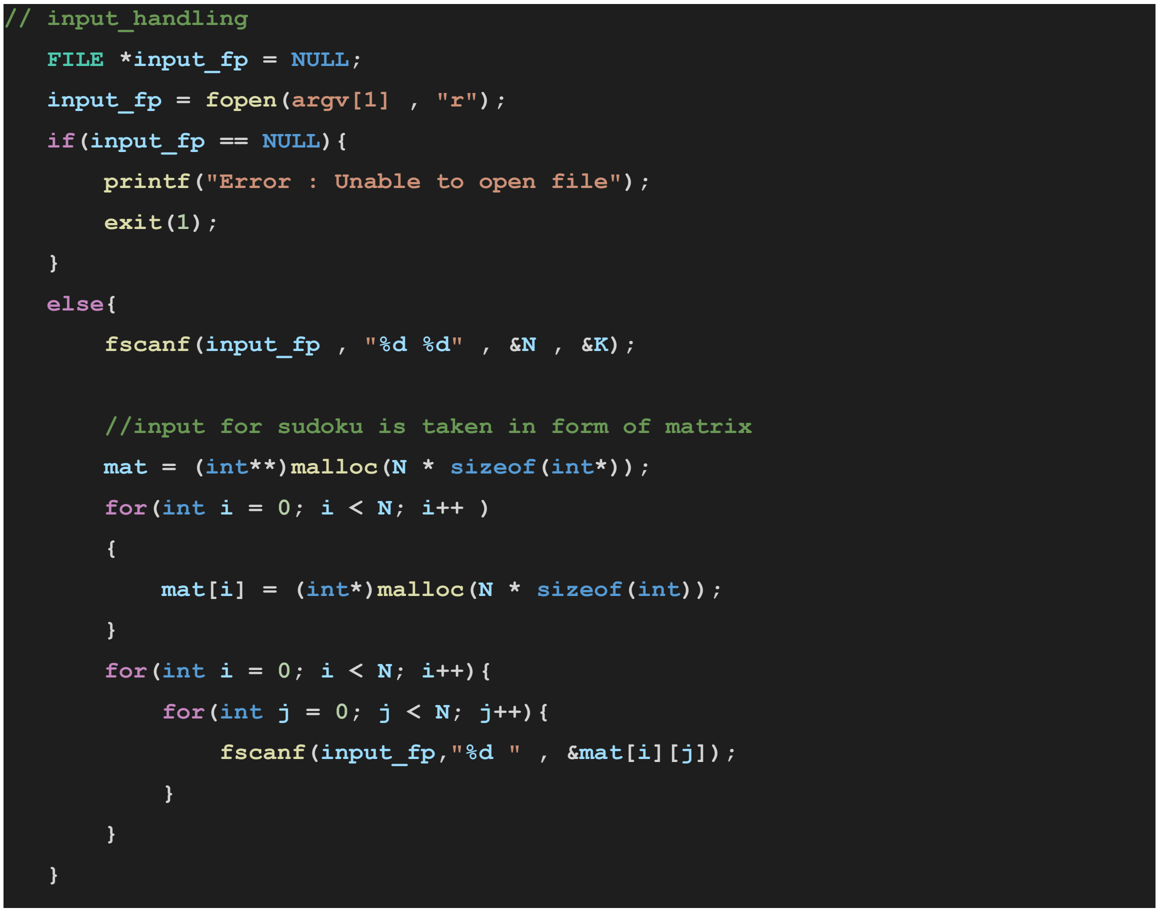
2. **Column Checker :**
   - The Implementation of column is same as that for row.
   - here we iterate over a column instead of row and check if it is invalid or not.

3. **Grid Checker**
   - here we have a int variable grid_no which store grid number which tells grid to check
   - grid_no also helps us start indices of each grid using which we iteration through the grid
   - for grid also we use valid array and invalid bool in similar manner as for row and column

```
            //start indices for each grid
            int grid_row = (grid_no/(int)sqrt_N)*sqrt_N;
            int grid_col = (grid_no%(int)sqrt_N)*sqrt_N;
```

Input_handling :

- Program takes input from a file name of which it take as a command line argument **argv[1]** . Using fopen & fscanf function it read first two space separated integer as N and K values .



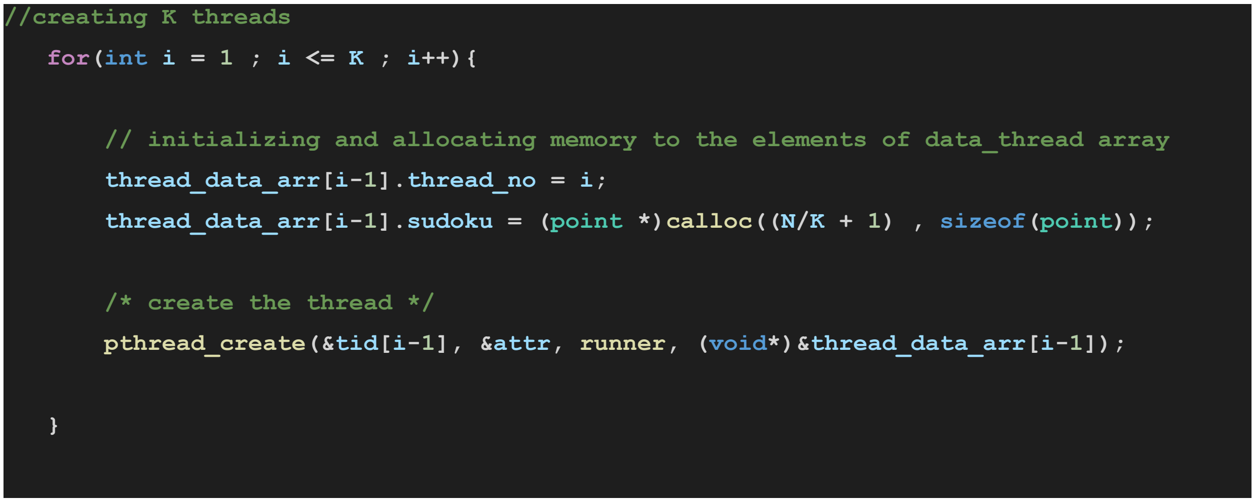
```
// input_handling
   FILE *input_fp = NULL;
   input_fp = fopen(argv[1] , "r");
   if(input_fp == NULL){
       printf("Error : Unable to open file");
       exit(1);
   }
   else{
       fscanf(input_fp , "%d %d" , &N , &K);

       //input for sudoku is taken in form of matrix
       mat = (int**)malloc(N * sizeof(int*));
       for(int i = 0; i < N; i++ )
       {
           mat[i] = (int*)malloc(N * sizeof(int));
       }
       for(int i = 0; i < N; i++){
           for(int j = 0; j < N; j++){
               fscanf(input_fp,"%d " , &mat[i][j]);
           }
       }
   }
```

- Also here we input entries of sudoku in form of two dimensional array or matrix

Pthread :

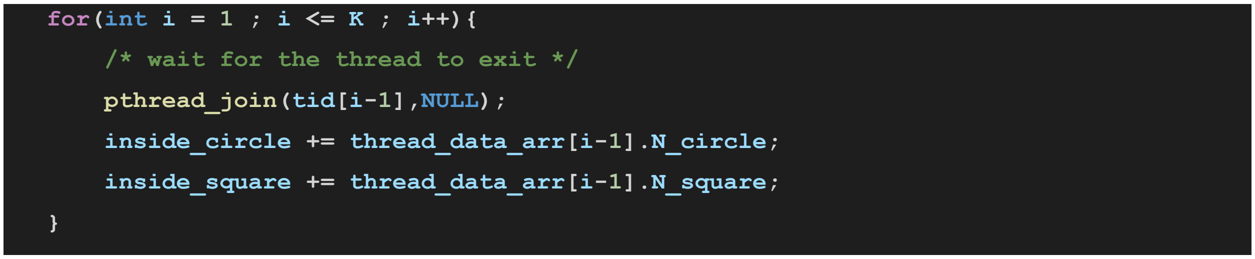
First we allocated an array of pthread tids and initialized their attributes

Creating and joining threads :



```
//creating K threads
   for(int i = 1 ; i <= K ; i++){

       // initializing and allocating memory to the elements of data_thread array
       thread_data_arr[i-1].thread_no = i;
       thread_data_arr[i-1].sudoku = (point *)calloc((N/K + 1)  , sizeof(point));

       /* create the thread */
       pthread_create(&tid[i-1], &attr, runner, (void*)&thread_data_arr[i-1]);


   }
```

Using a loop:
1) we initialize the parameters required for each of the thread
2) Using **pthread_create** we create each thread and invoke **runner** function
3) pthread_join stops the main thread and waits for all other thread to finish before further executing.



```
   for(int i = 1 ; i <= K ; i++){
       /* wait for the thread to exit */
       pthread_join(tid[i-1],NULL);
       inside_circle += thread_data_arr[i-1].N_circle;
       inside_square += thread_data_arr[i-1].N_square;
   }
```

Runner Function :

- For each thread , we take **j** as thread number and increase it by K for each iteration until j is greater than 3N
- For each we of **j** in the loop if j%3 = 1 then we check for a row
- If j%3 = 2 , we check for a column
- Else we check for a grid
- At last we exit the thread using `pthread_exit(0);`

OpenMP :

- In OpenMP , the works between threads is distributed by using in built OpenMp tool
- The Input handling , Output handling is same as that in pthreads
- For consistency in comparison of Pthread and OpenMP , the validator works same as that in Pthread file
- Using `#pragma omp parallel` , We start a parallel region in for validator to run
- Using `omp_set_num_threads(K)` , We can set the no. of threads

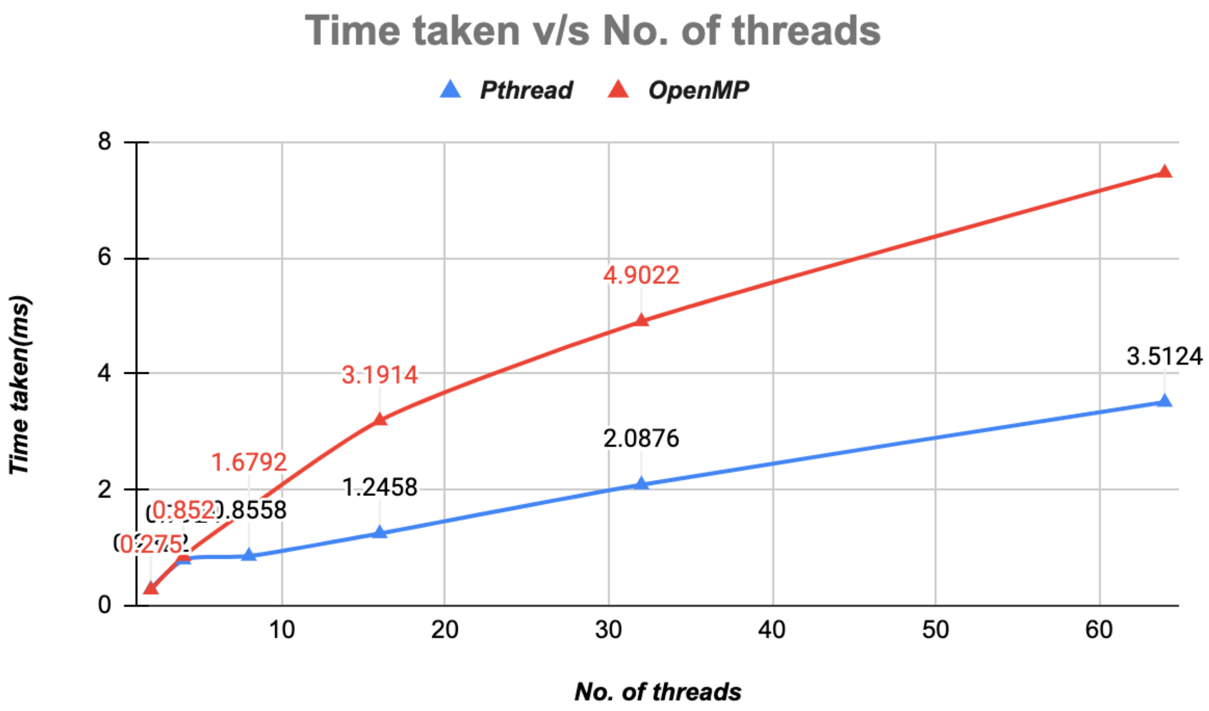
Note : For time analysis we use time.h library

Output Analysis :

1. Comparison between Pthread and OpenMP for time_taken if Number of threads are same(for 25x25 sudoku).

(Pthread) Threads	Time_taken in each iterations					Time average
	1st	2nd	3rd	4th	5th	
2	0.327	0.209	0.287	0.35	0.238	0.2822
4	0.821	0.904	0.625	0.837	0.775	0.7924
8	0.716	0.845	0.974	0.879	0.865	0.8558
16	1.295	1.199	1.279	1.186	1.27	1.2458
32	2.121	1.935	2.087	2.117	2.178	2.0876
64	3.471	3.345	3.584	3.474	3.688	3.5124

(OpenMP) Threads	Time_taken in each iterations					Time average
	1st	2nd	3rd	4th	5th	
2	0.292	0.198	0.294	0.35	0.241	0.275
4	0.817	0.878	0.847	0.877	0.841	0.852
8	1.516	1.62	1.73	1.69	1.84	1.6792
16	3.165	3.172	3.181	3.365	3.074	3.1914
32	4.903	4.902	4.943	4.834	4.929	4.9022
64	8.664	8.592	7.82	6.153	6.098	7.4654



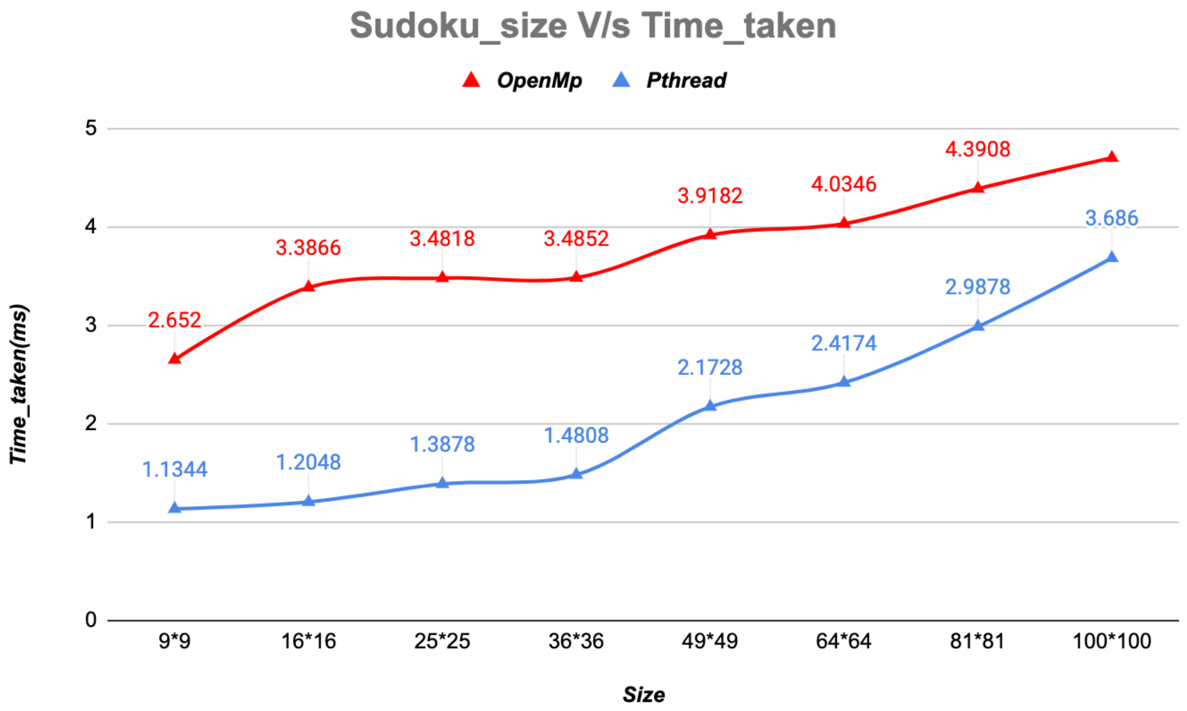
As we can clearly see here initially for low number of thread pthread and OpenMp behaves same but as we increase the no. of threads the time taken by Openmp increase by much more than Pthreads

Note : here the time increase for increase in thread because the 25x25 sudoku is pretty small for computation and the time for context switching and thread creation cant be neglected .

2. Comparison between the OpenMp and pthread for time taken if the size of sudoku is same(for number of threads = 16).

(Pthread) N*N Sudoku	Time_taken in each iterations					Time average
	1st	2nd	3rd	4th	5th	
9*9	1.203	1.228	1.071	1.126	1.044	1.1344
16*16	1.226	1.264	1.199	1.12	1.215	1.2048
25*25	1.27	1.312	1.376	1.529	1.452	1.3878
36*36	1.429	1.542	1.336	1.413	1.684	1.4808
49*49	2.119	1.94	2.104	2.37	2.331	2.1728
64*64	2.482	2.477	2.341	2.525	2.262	2.4174
81*81	3.005	2.881	2.922	3.048	3.083	2.9878
100*100	3.536	3.799	3.52	3.656	3.919	3.686

(OpenMP) N*N Sudoku	Time_taken in each iterations					Time average
	1st	2nd	3rd	4th	5th	
9*9	2.846	2.99	2.517	2.399	2.508	2.652
16*16	3.246	3.298	3.363	3.499	3.527	3.3866
25*25	3.767	3.326	3.396	3.582	3.338	3.4818
36*36	3.819	3.498	3.305	3.457	3.347	3.4852
49*49	4.099	3.874	3.735	3.921	3.962	3.9182
64*64	4.025	4.067	4.048	4.069	3.964	4.0346
81*81	4.447	4.219	4.458	4.49	4.34	4.3908
100*100	4.663	4.707	4.741	4.99	4.423	4.7048



Here also we can clearly see the time taken by OpenMp is greater than that by Pthreads for same size sudoku .

Also we can see as we increase the size of sudoku the time taken increases in both cases.