HPC LAB - 2
REPORT

- SE20UARI038
- SE20UARI089
- SE20UARI144
- SE20UARI147
- SE20UCSE028

Merge Sort Parallel implementation:

The C code implements a parallel version of the Merge Sort algorithm using OpenMP. Merge Sort is a divide-and-conquer algorithm that recursively splits an input array into two halves, sorts the halves, and merges them to produce a sorted output array. The parallel version of the algorithm exploits the inherent parallelism in the merge step.

The code begins with including the necessary headers for standard input and output, string manipulation, memory allocation, and OpenMP parallelization. The code defines a constant, MAX_SIZE, which is commented out and not used in the program. The program then defines three functions and the main.

The function generate_list() takes an integer array pointer and the size of the array as arguments. The function fills the array with integers from 0 to n-1 and shuffles the elements using the rand() function.

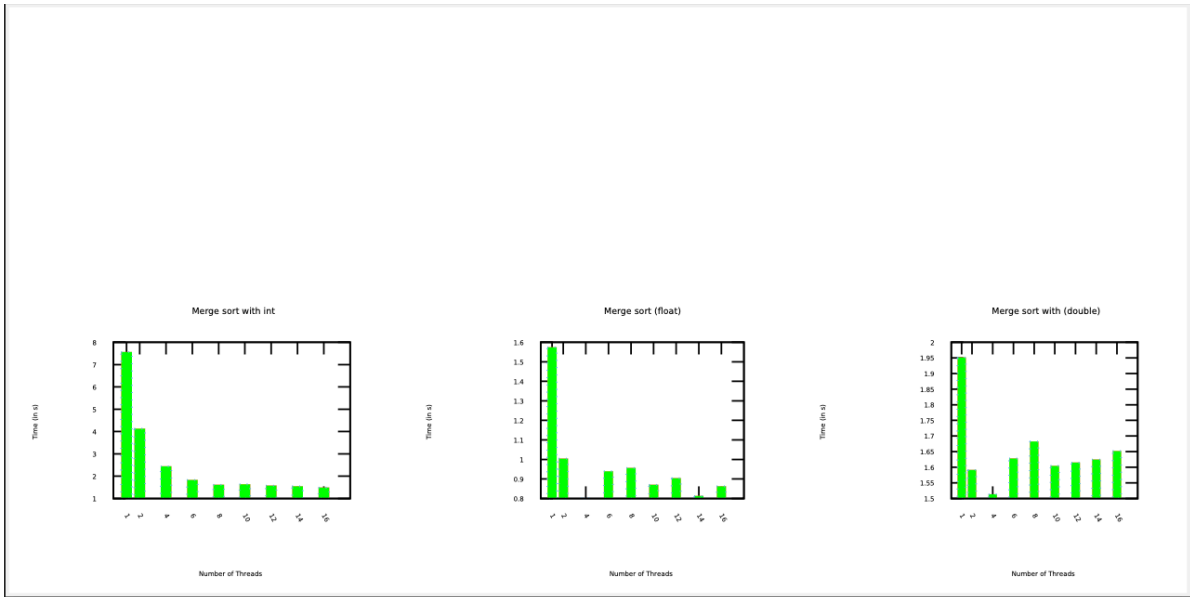
The function merge() takes an integer array pointer, the size of the array, and a temporary integer array pointer as arguments. The function merges two sorted subarrays into a single sorted array using a temporary array. The function first initializes three variables i, j, and ti. i represents the starting index of the first half of the array, j represents the starting index of the second half of the array, and ti represents the current index of the temporary array. The function then iterates through both halves of the array, comparing elements and copying them to the temporary array in sorted order.
The function then copies the remaining elements of the two halves to the temporary array. Finally, the function copies the sorted temporary array back to the original array.

The function Mergesort() takes an integer array pointer, the size of the array, and a temporary integer array pointer as arguments. The function recursively splits the array into two halves and sorts them in parallel using OpenMP tasks. The function creates two new tasks, one for the left half of the array and one for the right half, and waits for them to complete before merging the sorted halves using the merge() function.

The main function begins by defining an integer n and initializing it to 30. The main function then defines two integer arrays, data and tmp, each of size n. The main function calls the generate_list() function to fill the data array with shuffled integers. The main function then prints the unsorted data array using the print_list() function.

The main function then starts a timer using the omp_get_wtime() function and defines a parallel region using the #pragma omp parallel directive. The parallel region creates a single task to perform the Mergesort on the data array and wait for the task to complete using the #pragma omp single and #pragma omp taskwait directives.

After the parallel region ends, the main function stops the timer and prints the sorted data array using the print_list() function.



The main function also prints the time taken for the sorting process to complete.

Observations :-

As we can see, the time taken for double data type's sorting is greater than that of float and that of integer. This is mainly due to the precision while comparing the numbers. Also we can see that **the time of execution decreases from number of threads varying from 1 to 4, but then saturates from there.**

N-Queens Parallel implementation:

The code finds all possible solutions to the N-Queens problem.

The N-Queens problem is the problem of placing N chess queens on an N×N chessboard so that no two queens threaten each other. This code uses OpenMP to parallelize the search for all possible solutions.

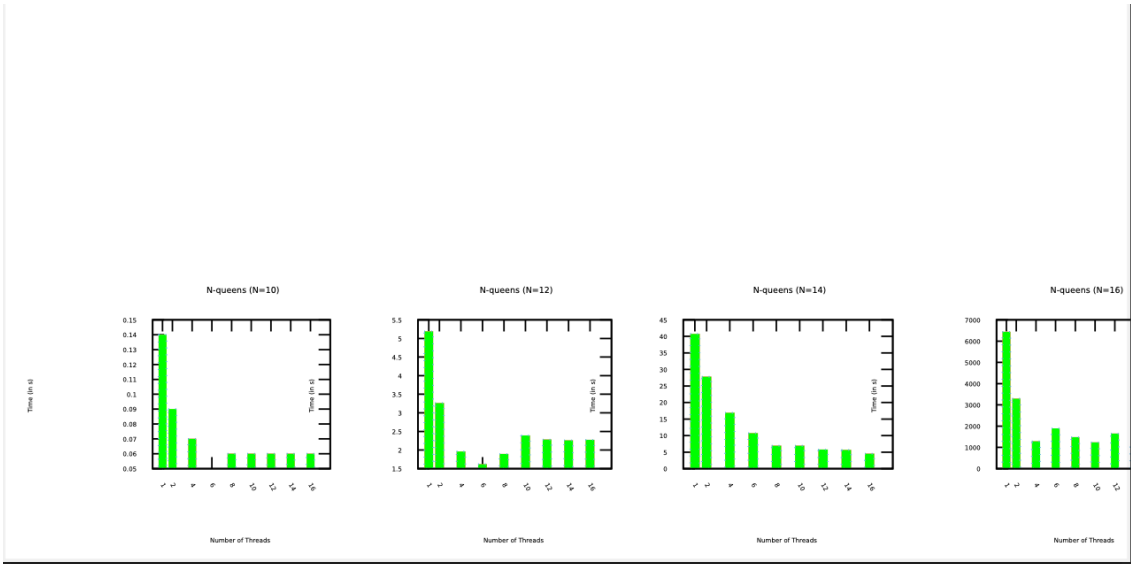
The function isSafe() is defined to check whether a queen can be placed at a particular position on the chessboard. The function takes in a 2D array representing the chessboard, the position where the queen is to be placed, and the size of the chessboard. It checks the row, column, and the diagonals of the given position and returns 1 if it is safe to place the queen at the given position.

The function print_board() is defined to print the chessboard after placing the queens. It takes in the 2D array representing the chessboard and the size of the chessboard. It prints the column position of the queens for each row.

The function n_queens() is defined to solve the N-Queens problem. It takes in the 2D array representing the chessboard, the current column, and the size of the chessboard. It recursively places the queens in all possible positions in the current column and then checks if it is safe to place the queen in that position using the isSafe() function.
If it is safe, then it places the queen at that position and moves to the next column. It continues the process until all the queens are placed on the chessboard. If a solution is found, it calls the print_board() function to print the solution. This function uses OpenMP to parallelize the search for all possible solutions.

The main() function is defined to read the input N and the number of threads from the command line arguments. It initializes a 2D array representing the chessboard and sets all values to 0. It then sets the number of threads to be used for parallel processing and starts the timer.

It calls the n_queens() function to find all the possible solutions to the N-Queens problem.
Once all the solutions are found, it stops the timer and prints the time taken to find all the solutions.



Observations :-

As we can see, the execution time increases as the size of chessboards increase, and it's obvious due tho the fact of searching all solutions in a larger search space tree. Also, there seem's to be an interesting trend while varying the number of threads, as the **execution time decreases very well for threads between 1 to 6 but then there seems to be some saturation and noise for more no.of threads.**