CITS5507 High Performance Computing: Project 1

Raj Joshi (22627911)

Introduction: In this project we are applying the knowledge gained from classroom, lab sessions and apply it using complex datastructure with the use of high precision floating point numbers. We have optimized the code using parallel computing concepts such as sections, which helps the processors to sub-divide the work and optimally reach a solution. Sections use the concept of forking and join to subdivide and merge the tasks.

The programs are tested across 2 randomly generated floating point numbers in the range of 0.0 to 10.0 having numbers with dataset size of 10,000 and 100,000 numbers. The numbers taken for this project are relatively small but noticeable difference can be observed in the test cases. Note that the test cases are run by commenting the print statement in all the programs to obtain the best output. The program is tested multiple times, however the results mentioned in this article are noted when the program is executed the first time. This is done to remove/clear the cache memory of previously loaded programs.

The programs used in this project are datastructure problems I.e. Quick sort, Merge Sort and Enumeration sort. Quick sort and Merge Sort are a type of divide and conquer algorithm with Best Case time complexity as O (n log n). Enumeration sort program is implemented by providing rank to all the element in the array and then sorting the array based on the ranks. The detailed description of the algorithm are mentioned below.

1. Merge Sort
    a. Serial Algorithm
        i. Merge Sort algorithm works on the 'Divide and Conquer' technique, where in the array is divided into 'n/2' splits where 'n' is the number of element s in th array. The grouped array is then sorted internally and again the first and last elements of each array is checked to regain the sorted order.
        ii. Pseudo Code:
            1. MergeSort(A, p, r):
            2. if p > r
            3. Stop
            4. q = (p+r)/2
            5. mergeSort(A, p, q)
            6. mergeSort(A, q+1, r)
            7. merge(A, p, q, r)
        iii. Time Analysis
            1. The time complexity of the Merge sort algorithm under best case is O(n log n) and under worst case is O(n^2).
            2. The average execution time can be summarized as O(n log n).
        iv. Execution analysis

1. We have used different datasets to test our code and denote the time taken to execute the code. This has been depicted in comparison with parallel computing in a table below.
2. The 3 datasets are floating point values that have are of sizes ranging from (10,000), (100,000) and the numbers are in the range of 0.0 to 10.0.

v. Code Compilation
1. For serial programming the code is compiled on MacOS with the following code 'gcc-11 merge_serial.c -o merge'
2. The 'gcc-11' denotes the c-lang version used. Using c-lang we compile the code and create an executable object file 'merge'.

b. Parallel Algorithm
i. The merge sort algorithm can be executed faster using parallel computing.
ii. In thus approach we will divide the array in to 'N/2' splits and then divide these arrays into different threads and sort it in such a way that we obtain the sorted array in the end.
iii. Here we have used the concept of sections from parallel computing. Sections speeds up the processing time
iv. Pseudo Code:
1. MergeSort(A, p, r):
2. if p > r
3. Stop
4. q = (p+r)/2
5. #pragma omp section
6. mergeSort(A, p, q)
7. #pragma omp section
8. mergeSort(A, q+1, r)
9. merge(A, p, q, r)

v. Time Analysis
1. As seen in the serial programming the best case Time complexity of Serial Merge sort is O(n log n) and by optimising the code using parallel computing, we reduce the execution time by using more threads. We have used default threads in our program as 8. Hence the time complexity for our parallel merge sort is O(n (log n)/8).

vi. Code Compilation
1. For parallel programming we must use 'gcc-11 –fopenmp merge_parallel.c -o merge_parallel'.
2. Here we must specially call the –fopenmp library to execute the code using parallel programming.

c. Comparison
i. Across three datasets, we have test both serial and parallel programming code and the parallel programming code is way faster when the dataset size is increased gradually
ii. The table below shows the exact execution time recorded for both programs.

Dataset Size	Serial execution	Parallel excution
10,000	0.03 sec	0.01 sec
100,000	1.418 sec	0.88 sec

2. Quick Sort
    a. Serial Algorithm
        i. Quick sort is another type of 'Divide and conquer' algorithm that takes in a pivot element and sorts the array until the pivot element reaches the correct order and then divides the array into 2 subparts. Again, the pivot element is selected, and the sorting of the pivot element take place and splits the array. This is done recursively until the correct order of the array is achieved.
        ii. We have implemented the quicksort algorithm using Lomuto partition scheme
        iii. Pseudo Code

```
algorithm quicksort(A, lo, hi)
 if lo >= 0 && hi >= 0 && lo < hi then
   // Partition array and get pivot index
p := partition(A, lo, hi)
// Sort the two partitions
quicksort(A, lo, p - 1) // Left side of pivot
quicksort(A, p + 1, hi) // Right side of pivot
// Divides array into two partitions
algorithm partition(A, lo, hi) is
pivot := A[hi] // The pivot must be the last element
// Pivot index
i := lo – 1
for j := lo to hi do
// If the current element is less than or equal to the pivot
if A[j] <= pivot then
// Move the pivot index forward
i := i + 1
// Swap the current element with the element at the pivot
swap A[i] with A[j]
 return i // the pivot index
```

        iv. Time Analysis
            1. The best case scenario for quick sort algorithm is O(n log n) and worst case scenario is still O(n log n). However, in rare conditions the time complexity for Quick Sort algorithm reaches O(n^2).
        v. Code Compilation
            1. We are executing the program using serial execution and hence the procedure will be same as merge sort.
    b. Parallel Algorithm
        i. To speed up the execution of quick sort we are using parallel computing. Quick sort using parallel programming speeds up the execution drastically. Here, we have used the concept of sections.
        ii. Some more information about sections
        iii. Pseudo code:

```
algorithm quicksort(A, lo, hi)
```

```
2. if lo >= 0 && hi >= 0 && lo < hi then
3.         // Partition array and get pivot index
4. p := partition(A, lo, hi)
5. // Sort the two partitions
6. quicksort(A, lo, p - 1) // Left side of pivot
7. quicksort(A, p + 1, hi) // Right side of pivot
8. // Divides array into two partitions
9. algorithm partition(A, lo, hi) is
10. pivot := A[hi] // The pivot must be the last element
11. // Pivot index
12. i := lo – 1
13. for j := lo to hi do
14. // If the current element is less than or equal to the pivot
15. if A[j] <= pivot then
16. // Move the pivot index forward
17. i := i + 1
18. // Swap the current element with the element at the pivot
19. swap A[i] with A[j]
20. return i // the pivot index
```

iv. Time Analysis
    1. The best and worst case scenario for quick sort is O(n log n) for our problem. The time complexity for our problem is O(n (log n)/8).
    2. The time complexity of quick sort is much lower than Merge sort and hence it takes lesser time to execute.

v. Code Compilation
    1. The code is executed using the parallel processing library of OpenMP.

c. Comparison
    i. The testing of serial and parallel execution of program does have a drastic effect on the execution time.
    ii. Both the programs are tested against 2 dataset of different size with numbers ranging from 0.0 to 10.0 .

Dataset size	Serial	Parallel
10,000	0.05 sec	0.025 sec
100,000	1.87 sec	0.75 sec

3. Enumeration Sort
    a. Serial Algorithm
        i. Enumeration sort algorithm works on the ranking element basis, hence ranks are calculated and allocated to all elements. The elements are then sorted with ranks being the indices for the elements.
        ii. Pseudo code
            1. Compute the length of the array
        iii. Time Analysis
            1. The enumeration sort has the best- and worst-case time complexity of O (n^2).
            2. It is the most time taking algorithm out of the three.
    b. Parallel Algorithm