Veermata Jijabai Technological Institute, Mumbai 400019

Experiment No.: 03

Aim : Implementation of parallel quick sort [Hyper quick sort] using CUDA

Name : Kiran K Patil

Enrolment No.: 211070904

Branch: Computer Engineering

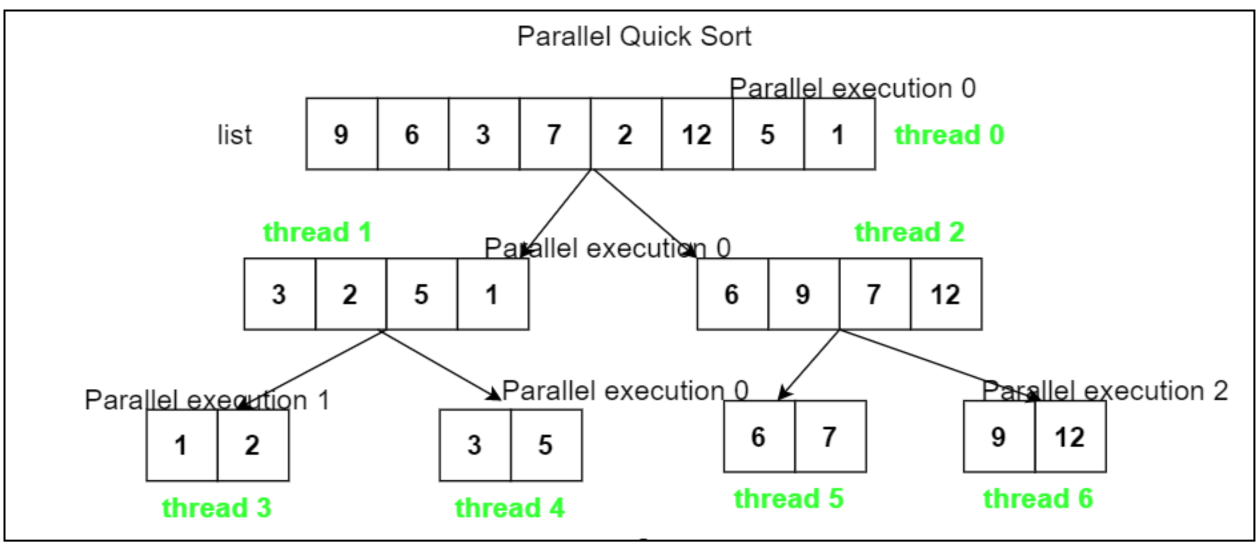
Batch: IV

Approach : Optimized Parallel Quick Sort

In this approach we change a small detail in the number of processes used at each step. Instead of doubling the number of processes at each step, this approach uses n number of processes throughout the whole algorithm to find pivot element and rearrange the list. All these processes run concurrently at each step sorting the lists.

Steps :

1. Start n processes which will partition the list and sort it using selected pivot element.
2. n processes will work on all partitions from the start of the algorithm till the list is sorted.
3. Each processes finds a pivot and partitions the list based on selected pivot.
4. Finally the list is merged forming a sorted list.



Program :

```
%%cu
#include<iostream>
#include<omp.h>

using std::cout;
using std::endl;

class ParallelQuickSort{
    //keep count of threads
    int k = 0;

    private:
        //partitioning procedure
        int partition(int arr[], int l, int r){
            int i = l + 1;
            int j = r;
            int key = arr[l];
            int temp;
            while(true){
                while(i < r && key >= arr[i])
                    i++;
                while(key < arr[j])
                    j--;
                if(i < j){
                    temp = arr[i];
                    arr[i] = arr[j];
                    arr[j] = temp;
                }else{
                    temp = arr[l];
                    arr[l] = arr[j];
                    arr[j] = temp;
                    return j;
```

```
                }
            }
        }

    public:
        void quickSort(int arr[], int l, int r){
            if(l < r){
                int p = partition(arr, l, r);
                cout <<  "pivot " << p << " found by th
read no. " << k << endl << endl;

                #pragma omp parallel sections
                {
                    #pragma omp section
                    {
                        k = k + 1;
                        quickSort(arr, l, p-1);
                    }
                    #pragma omp section
                    {
                        k = k + 1;
                        quickSort(arr, p+1, r);
                    }
                }
            }
        }
       //prints array
        void printArr(int arr[], int n){
            for(int i = 0; i < n; i++)
                cout << arr[i] << " ";
            cout << endl;
        }
        //run the whole procedure
        void run(){
```

```
            int arr[] = {9, 6, 3, 7, 2, 12, 5, 1};
            int n = sizeof(arr) / sizeof(arr[0]);
            printf("\n************ Implementation of Qu
ick Sort using CUDA ******************** \n\n");
            quickSort(arr, 0, n-1);
            printArr(arr, n);
        }
};

int main(){
    ParallelQuickSort pqs;
    pqs.run();
    return 0;
}
```

Output:

```
************ Implementation of Quick Sort using CUDA ********************

pivot 6 found by thread no. 0

pivot 3 found by thread no. 1

pivot 1 found by thread no. 2

pivot 5 found by thread no. 5

1 2 3 5 6 7 9 12
```

Parallel quick sort analysis

- At each step n processes process log(n) lists in constant time O(1). The parallel execution time is O(logn) and there are n processes.
- Total time complxity is O(n log n).
- This complexity did not change from the sequential one but we have a achieved an algorithm that can run on parallel processors, meaning it will execute much faster at a larger scale.
- Space complexity is O(logn).

Conclusion:

- Thus we have implemented the parallel search algorithm (BFS) using CUDA.
- This program uses a CUDA kernel to perform BFS in parallel on a GPU.
- **#pragma omp parallel sections** defines a parallel region containing the code that we will execute using multiple threads in parallel. This code will be divided among all thread