Comparing performance of Quick Sort under Sequential and Parallel Execution

Kashish Gulati, Laveesh Gupta, Madhur, Manvendra Bansal
(1MS18CS056,1MS18CS062,1MS18CS063 and 1MS18CS069)

Abstract: One of the most critical problem in computer science is the sorting process, for that reason many sorting algorithms have been developed, such as Quick sort, Merge sort, Bubble sort, Insertion sort and Selection sort...etc. In computer science sorting algorithm is an algorithm that arranges the components of a list in a specific order. Sorting algorithms are taught in some fields such as Computer Science and Mathematics. There are numerous sorting algorithms applied in the field of computer science. In our research we aim to parallelization the Quick sort algorithm using multi threading (OpenMP) platform. We divide it into two parts by partitions the given array around the first pivot where each part is processed by independent thread i.e. different threads will find out pivot element in each part recursively and sort the array independently.

1) INTRODUCTION

Sorting is a very important building block in most useful algorithms. We need to sort large amounts of data so we can process it efficiently.

The normal way to implement quick sort on serialized processors is whereby steps are executed sequentially until the program terminates when the task is completed. One process is started in the CPU which executes the code line by line.

Parallel programming is whereby a program is broken down into concurrent programs which are executed concurrently on multiple threads on a processor. Here coordination is required.

Why parallel processing?

- High Performance
- Less time taken to complete tasks.
- Lower work loads per processor.

i) Application- In this report, we will see how to parallelize Quicksort algorithm using OPENMP. In Quicksort, an input is any sequence of numbers and output is a Sorted array. QuickSort is a Divide and Conquer algorithm. It picks an element as pivot and partitions the give narray around the picked pivot. We start with one number, mostly the first number, and finds its position in the sorted array. Then we divide the array into two parts considering this pivot element's position in the sorted array. In each part, separately, we find the pivot elements. This process continues until all numbers are processed and we get the sorted array.

In Parallel Quicksort, we divide it into two parts by partitions the given array around the first pivot where each part is processed by independent thread i.e. different threads will find out pivot element in each part recursively and sort the array independently.

If we use #pragma omp parallel section inside quicksort function the total no of threads will be equal to the total no of calls to the quicksort function. The code is correct, but two threads are created and destroyed in each call which creates a lot of overhead which makes it slower than the serial code.

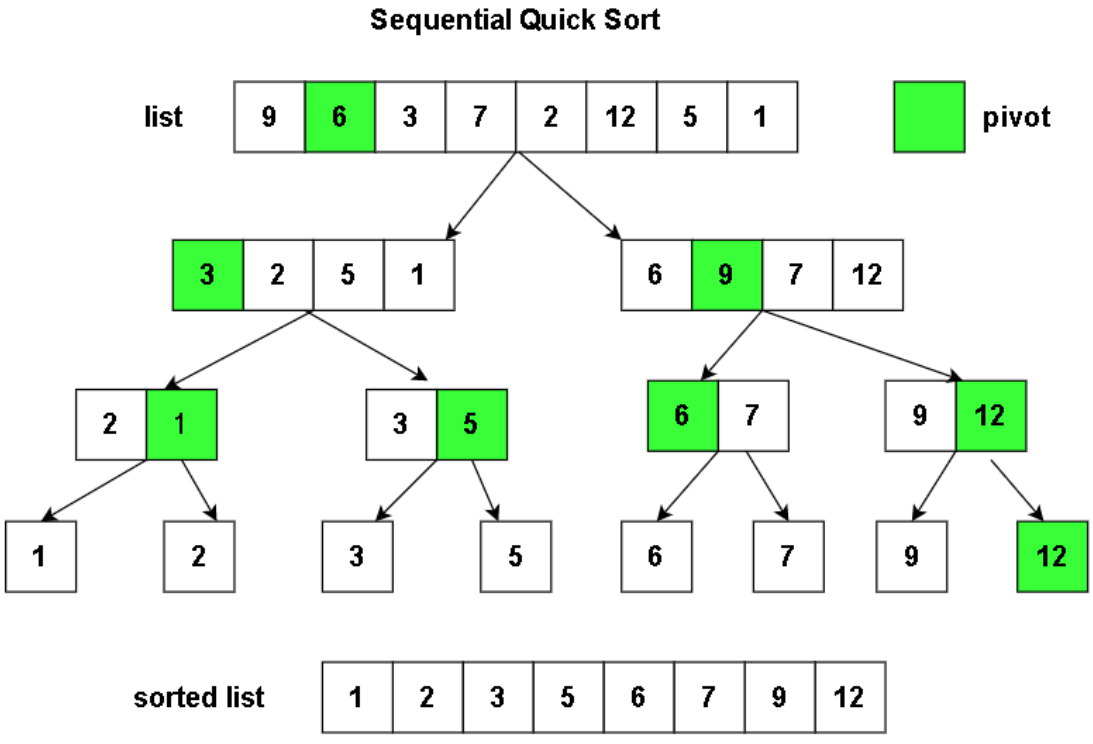
So we create only two threads instead of creating too many and got the performance gain from the serial version.

ii) Parallelism Technique- We use *openmp(open multiprocessing)*, an open source library for multi-threading which will enable use to implement the algorithm concurrently.
#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 threads.
Variable *k* stores the thread number which we print out just for understanding what is happening under the hood.

2) METHOD

(a) Sequential Quick Sort

i. Find a random pivot p.
ii. Partition the list in accordance with this pivot, elements less than pivot to the left of pivot, elements greater than pivot to the right of pivot and elements equal to pivot in the middle. <p =p >p.
   A. That is initialize i to first element in list and j to last element.
   B. Increment i until list[i] > pivot
   C. Decrement j until list[j] < pivot
   D. Repeat the above steps until i > j.
   E. Replace pivot element with list[j]
iii. Re-curse on each partition.
iv. When the list size is 1, it terminates. This acts as the base case. At this point the partitions are in sorted order so it merges them forming a complete sorted list.



Sequential quick sort analysis:

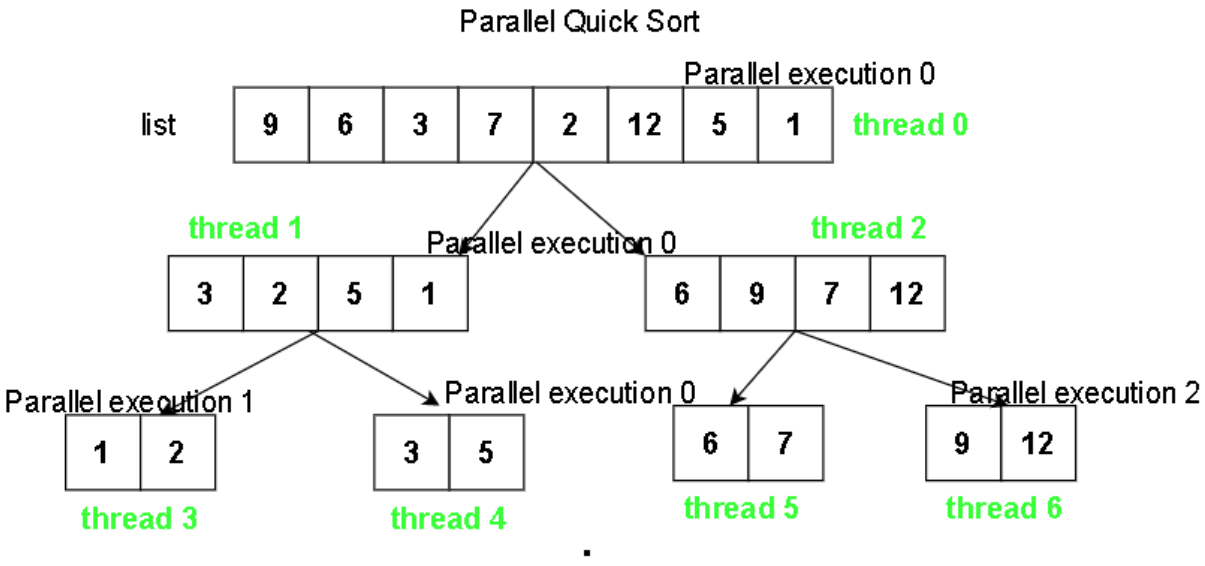
The time complexity is O(nlogn) in the average case.
The space complexity is O(logn).

(b)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.



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 θ(nlogn).
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).

CODE-

```
#include <assert.h>
#include <string.h>
#include <stdlib.h>
#include <stdio.h>
#include <omp.h>

#define TASK_SIZE 100


int partition(int * a, int p, int r)
{
   int lt[r-p];
   int gt[r-p];
   int i;
   int j;
   int key = a[r];
   int lt_n = 0;
   int gt_n = 0;

   for(i = p; i < r; i++){
      if(a[i] < a[r]){
         lt[lt_n++] = a[i];
      }else{
         gt[gt_n++] = a[i];
      }
   }

   for(i = 0; i < lt_n; i++){
      a[p + i] = lt[i];
   }

   a[p + lt_n] = key;
```

```
   for(j = 0; j < gt_n; j++){
      a[p + lt_n + j + 1] = gt[j];
   }

   return p + lt_n;
}

void seq_quicksort(int arr[], int low, int high)
{
        int j;

        if(low < high){
                j = partition(arr, low, high);

                seq_quicksort(arr, low, j - 1);
                seq_quicksort(arr, j + 1, high);

        }
}


void quicksort(int * a, int p, int r)
{
   int div;

   if(p < r){
      div = partition(a, p, r);
      #pragma omp task shared(a) if(r - p > TASK_SIZE)
      quicksort(a, p, div - 1);
      #pragma omp task shared(a) if(r - p > TASK_SIZE)
      quicksort(a, div + 1, r);
   }
}

int main(int argc, char *argv[])
{
        int n  = (argc > 1) ? atoi(argv[1]) : 200;
        int arr[n];
        int arr1[n];
      for(int i=0;i<n;i++){
                arr[i]=rand()%n;
                arr1[i]=arr[i];
```

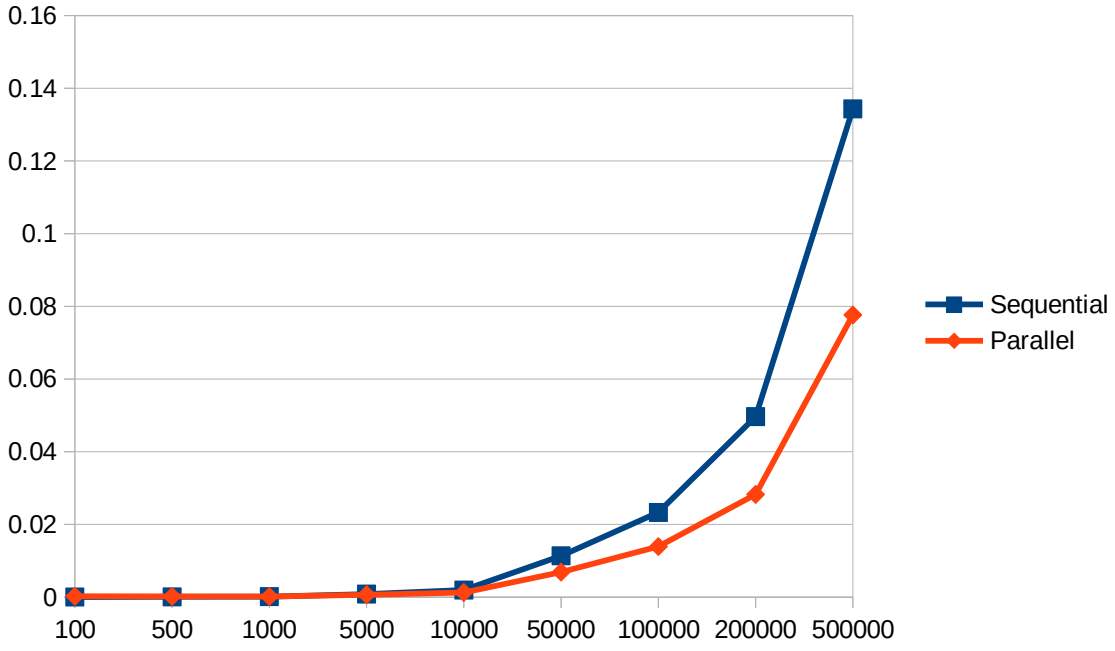
```
        }
    int numThreads = 2;

    omp_set_dynamic(0);
    omp_set_num_threads(numThreads);
        double begin,end;
    begin = omp_get_wtime();
    seq_quicksort(arr,0,n-1);
    end = omp_get_wtime();
    printf("Sequential Time: %f (s) \n",end-begin);
    begin = omp_get_wtime();
    #pragma omp parallel
    {
      #pragma omp single
       quicksort(arr1, 0, n-1);
    }
    end = omp_get_wtime();
    printf("Parallel Time: %f (s) \n",end-begin);
  return 0;
}
```

3) RESULTS

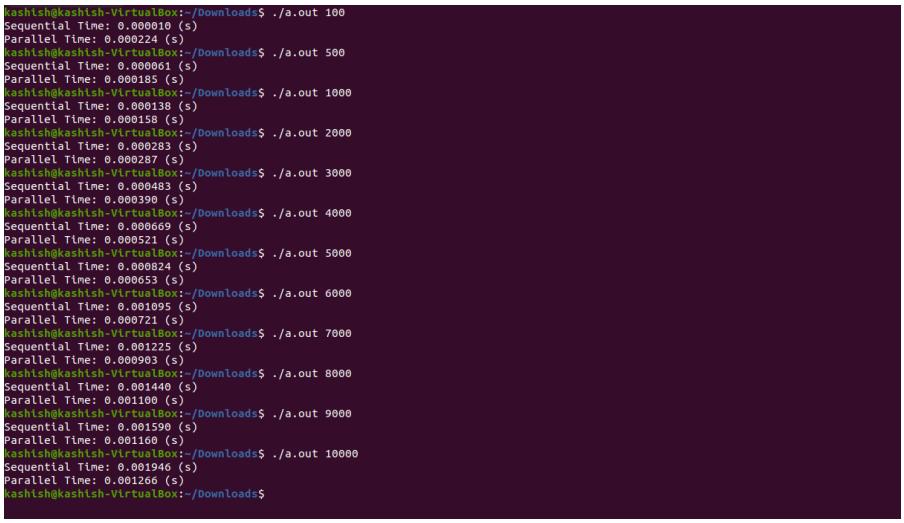
N	Sequential	Parallel
100	0.000010	0.000224
500	0.000061	0.000185
1000	0.000138	0.000158
5000	0.000824	0.000653
10000	0.001946	0.001266
50000	0.011358	0.006875
100000	0.023295	0.013906
200000	0.049674	0.028247
500000	0.134344	0.077629

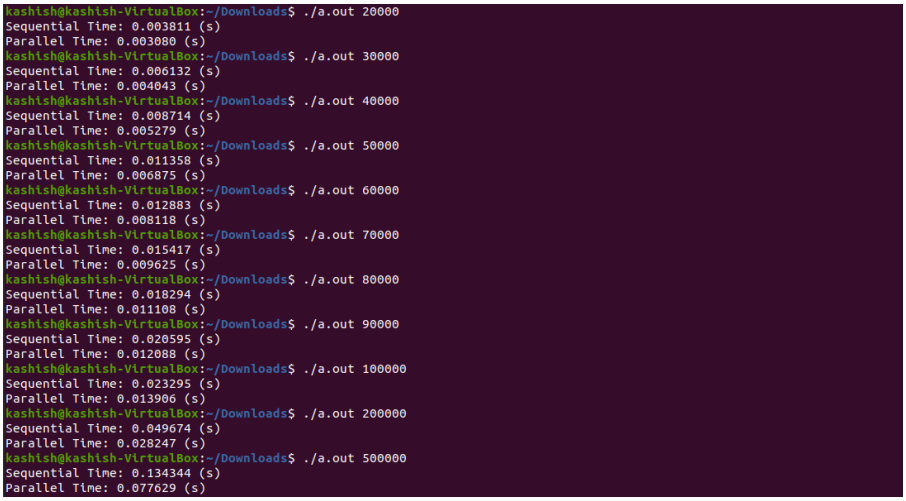
The table above shows the values of execution time in seconds for Sequential and Parallel Quick sort algorithm for different values of N where N represents the number of elements in the array. We have also plotted a graph for making comparisons



- The Graph above has number of elements (N) on the x-axis and the time in seconds on the y-axis.
- The Dark Blue line indicates the execution time of Sequential algorithm.
- The Orange line indicates the execution time of the Parallel algorithm.
- Initially when the value of is low i.e. form 100-5000, we can observe that Sequential algorithm is better, the parallel one is not much slower but is comparatively slower when compared to sequential in this initial stage.
- As the value of N increases and comes around 100000 we can observe that parallel execution time is almost half of that of the sequential.
- Taking the value of N as 500000 we can observe that parallel is way much more faster than the sequential.

Below are the snapshots of the execution of code for various values of N.





References

1. Sinan Sameer Mahmood Al-Dabbagh, and Nawaf Hazim "*Parallel Quicksort Algorithm using OpenMP*" figshare. Journal contribution. https://doi.org/10.6084/m9.figshare.3470033.v1

2.Philippas Tsigas, and Yi Zhang "*A Simple, Fast Parallel Implementation of Quicksort*" Eleventh Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2003.

3.Arne Maus "*A full parallel Quicksort algorithm for multicore processors*" Conference: NIK 2015, Norwegian Informatics Conference 2015 At: Alesund, Norwa