Comparative study of deterministic quick sort and randomized quick sort

Upasana Ghosh

Roll no: UE143110
U.I.E.T, Panjab University
Email: ghoshupasana05@gmail.com

AIM

This paper presents comparative study of deterministic quick sort and randomized quick sort. It is observed that the time required to execute and run deterministic quick sort is more than the time required to execute and run randomized quick sort for both sorted and unsorted array elements.

ABSTRACT

Quick sort is one of the fastest sorting algorithms. Quick sort outperform other sorting algorithm in practice like heap sort, merge sort, insertion sort, etc because it efficiently uses cache of CPU. Although the worst case time complexity of quick sort is $O(n^2)$.

APPROACH

Deterministic quick sort is based on divide and conquer algorithm. A pivot element is picked from the array element and the array elements are divided into two sub-arrays around the pivot element recursively. After partitioning, the array elements are ordered in such a way that all the elements smaller then pivot element comes before the pivot and the elements greater than the pivot comes after it. The elements whose values are same as that of pivot can go either way. After this partitioning, the pivot is in its final position. In deterministic quick sort, these steps are repeated recursively until the array contains at least 2 elements. In this paper, the first element of array is taken as pivot. This sorting is performed with different array sizes that are randomly filled with integer values generated by random function and the time taken to sort unsorted array is calculated. Deterministic quick sort is then applied on sorted array elements with different array sizes and the time taken to sort already sorted array is calculated.

Another approach to sort array elements that is presented in this paper is randomized quick sort. Randomized quick sort is similar to that of deterministic quick sort but in this, the pivot element is swapped randomly with other element in the array. After swapping, the first element is taken as pivot and partitioning is done similar to that of deterministic quick sort. Randomized quick sort is also performed for both sorted and unsorted array element for different array sizes and the time taken to sort these elements is calculated.

Time taken to sort ordered and unordered list is calculated using deterministic and randomized quick sort and graphs are plotted according to the results obtained.

EXPERIMENT

In this experiment, an array of size of 100 is taken and is populated randomly with integer values using rand function. Deterministic quick sort and randomized quick sort functions are invoked one by one to sort the elements in the array in ascending order. In deterministic quick sort, the partitioning is done in such a way around an index j such that all the elements smaller than j^{th} element will positioned in the left of that index and all the elements greater than j^{th} element will be positioned in the right of that index where j is the index of pivot after performing deterministic quick sort. The

resultant sub-arrays are further sorted recursively until array contains at least 1 element. In randomized quick sort, the pivot element is first swapped with other element in the array which is selected randomly and then partitioning is done in similar way as done in deterministic quick sort.

Both these sorting algorithms are performed on sorted array as well. Array is populated randomly with integer values using rand function and sorted using sort function. Then both these functions are invoked one by one and time taken is calculated. The array size is then increased by 300 and again times taken by both these functions are calculated. For every array size, the array elements are filled 50000 times, sorting operation is called 50000 times and time taken to sort elements is calculated 50000 times and average time is calculated from it. This analysis is done with the following bounds:
Initial array size: 100
Step size: 300
Final array size: 4900

Platform used for compilation and execution:
Operating system: Ubuntu 15.10
RAM: 4GB
IDE: Dev-C++ version 4.9.9.2
Compiler: GCC 3.4.2
Graph plotted: Octave-3.6.4.

ALGORITHM

Algorithm of deterministic quick sort:
Declaration:
a: array consisting of integer values.
beg: starting index of array
end: ending index of array

```
algo quicksort(int a[],int beg,int end)
{int i, j, v;
 i=beg; j=end; v=a[beg];
 while(i<j)
  {   while(v>=a[i] && i<=end)
                i++;
      while(v<a[j])
                j--;
     if(i<j)
        {
           swap(a[i],a[j]);
        }
  }
  swap(a[beg],a[j]);
 return j;
}

algo qsort(int a[],int beg,int end) //funtion to call quick sort in two parts
{if(beg<end)
  {
   int j=quicksort(a,beg,end);
   qsort(a,beg,j-1);
   qsort(a,j+1,end);

  }
}
```

Algorithm of randomized quick sort:
Declaration:
a: array consisting of integer values.
beg: starting index of array
end: ending index of array

```
algo r_quick(int a[],int beg,int end)      //for randomized quick sort
{
   Int newbeg=beg+rand()%(end-beg+1);
//taking random number from the remaining list
     swap(a[beg],a[newbeg]);
    return quicksort(a,beg,end);   //calling quick
//sort after changing pivot value
}

algo quicksort(int a[],int beg,int end)
{
 i=beg; j=end; v=a[beg];
 while(i<j)
  {   while(v>=a[i] && i<=end)
                i++;
     while(v<a[j])
                j--;
```

```
    if(i<j)
       {
          swap(a[i],a[j]);
       }
  }
  swap(a[beg],a[j]);
 return j;
}

algo r_qsort(int a[],int beg,int end) //fucntion to
//call randomized quick sort
{if(beg<end)
 {
   int j=r_quick(a,beg,end);
   r_qsort(a,beg,j-1);
   r_qsort(a,j+1,end);

 }
}
```

RESULT

Table 1 depict the array size and the corresponding time taken (average) in seconds for sorting unsorted array in ascending order by randomized quick sort and deterministic quick sort. The result is depicted graphically in Graph 1.

Table 2 depict the array size and the corresponding time taken (average) in seconds for sorting sorted array in ascending order by randomized quick sort and deterministic quick sort. The result is depicted graphically in Graph 2.

DISCUSSION

Theoretically, running time of both randomized and deterministic quick sort is O(nlogn) when the elements are unsorted and average case partitions are executed for all array sizes. Table 1 and Graph 1 depict the result but the question arises here is that with the increase in array size, why sorting time of deterministic quick sort increases more rapidly than that of randomized quick sort. This is because partitioning of array element around the pivot is failing significantly and is taking more time to run even when the randomized quick sort is using rand function to swap pivot element which also consume time when the array size is large. The best case, average case and worst case of both randomize and deterministic quick sort is O(n logn), O(n logn) and $O(n^2)$ respectively. The expected running time of randomized quick sort is O(n log n). It is unlikely that this algorithm will choose a terribly unbalanced partition each time, so the performance is very good almost all the time.

When the array elements are sorted, partitioning is failing around every pivot element in deterministic quick sort which is the worst case of quick sort and thus, the running time is $O(n^2)$. That's why the graph obtained is parabolic in nature. But in randomized quick sort, as the pivot element is swapped every time before partitioning, running time is extensively lesser than that of deterministic quick sort. So, instead of hoping that array elements are random, randomness can be added explicitly into the operation of deterministic quick sort in order to make it behave as if the input were random using randomized quick sort. By using randomness in the internal flow of the algorithm, an arbitrary input can be converted into a random input, thereby obtaining many of the nice properties of its average performance for nearly every input and defeating our adversary almost all of the time.

Hence, the time taken to sort elements in quick sort largely depends on the choice of pivot element. If the selected pivot element is medium of the array element then the array would be partitioned into two equal sub-arrays and minimum time would require to sort the array, but if the selected pivot element is the largest or smallest element in the array, the partitioning would failed and maximum time would require to sort it. In randomized quick sort, pivot is randomly picked up from the array, so it depends entirely on the random function which

element it is picking at run time. The question arises here is, there are chances of picking worst-case pivot so is there any way to improve the algorithm so that it always select the best-case pivot? One solution is to choose 3 numbers uniformly at random (in spite of one) and keep the median of the 3 numbers as pivot. This method will

1) improve the worst possible case
2) make it less probable to happen

But in the best case, it would spend too much time to make useless comparisons for determining the pivots.

A second solution is to exactly find out the median and then partitioning the array. This will always have the best possible case but has a high price as the algorithm will be slower in the best cases, because of the usage of other algorithm used for finding the median. So, there doesn't exist any fast method of choosing the pivot, which improves the algorithm in all possible cases!

To decrease the processing time of quick sort, parallel computing can be a solution. Four threads of CPU can be assigned at one quarter of the array and each would return the list of sorted element. After that the resultant array would be merged from the four threads to get the final result. If the data is in large amount, say terabytes or petabytes, data could be divided into multiple computers which would further be subdivided into threads. But if the data is not very large, then this approach would be inefficient as processing the data in local machine will be faster than transferring it across network and then merge those results by again transferring those results into local machine. Even on a single machine, process won't actually get 4 time speed from four threads as they fight for memory bandwidth. But process would get some speedup from this.

CONCLUSION

As time required for sorting array element is lesser for randomized quick sort then deterministic quick sort, it is concluded that randomized quick sort is a better approach for sorting array elements. If the data is in large amount then randomized quick sort should be performed in multiple computers.