Quicksort

Abstract

A short abstract summarising what your project is about and the main results you obtained.

Student 1
Pratyai Mazumder

Student 2
Lodovico Mazzei

Student 3
Michele Chersich

1 Introduction

1.1 The sorting problem

2 The algorithm

2.1 The partitioning problem

Quicksort belongs to the family of *partition sort* algorithms, where a partitioning routine is called recursively until a whole sequence is sorted.

Given an array $A[1, \ldots.., N]$, the partitioning algorithm should split a permutation of it, A', into two partitions $U = A'[1, \ldots.., p]$ and $V = A'[p+1, \ldots.., N]$ such that $U[i] \leq V[j]\ \forall\ i, j \in \{[1, p], [p+1, N]\}$. By recursively partitioning U and V themselves, base cases will eventually be reached, consisting of single-element or empty arrays, which are already sorted. Figure 1 illustrates the recursive routine.

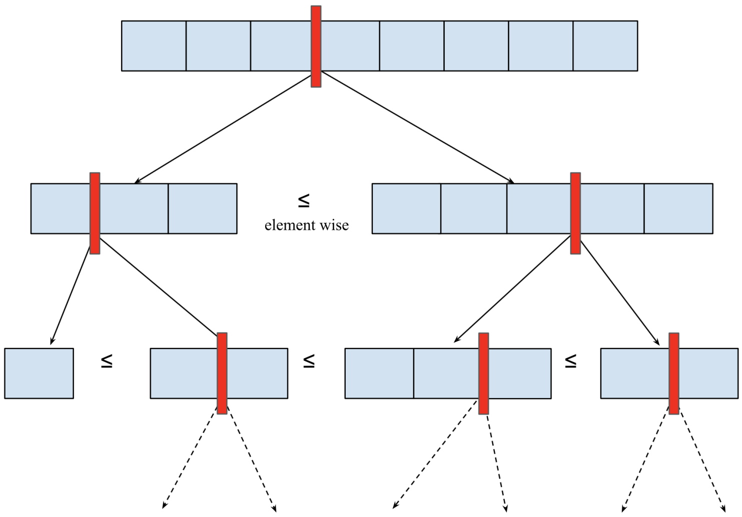

Figure 1. A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!A picture of the same tucan looking the other way!

The following pseudocode recursively sorts a given subarray of A. It assumes a partition routine which will be discussed in the following sections.

```
quicksort(A, low, high):
      if low >= 0 && high >= 0 && low < high then
            // Reorder A around the pivot in-place such that
            // the partition A[low],...,A[p] is pointwise
            // less than or equal to the partition A[p+1],...,A[high]
            p := partition(A, low, high)
            // Recursively solve the two partitions without affecting the partition invariant
            quicksort(A, low, p)
            quicksort(A, p+1, high)
```

2.2 Pivot selection

The partitioning in quicksort relies on a method called pivoting. This consists in picking a value γ to be our pivot. Given the array A, we then permute it in a way such that every element before γ is smaller than it, and every one after it is larger. The internal order in these subarrays is irrelevant, as it will be addressed in further iterations of the algorithm.

Clearly, the pivot choice will have an impact on the efficiency of the algorithm. We divide the selection methods into two categories: one-step and multi-step.
The following paragraphs provide a description using as an example the array $A[1,....,N]$ introduced above.

2.2.1 One-step selection

These methods only require chosing a value at each iteration and using it as the pivot, the difference being the position of A from which the value is selected.

- *First element*: set the pivot to be the value $A[1]$.
- *Last element*: set the pivot to be the value $A[N]$.
- *Central element*: set the pivot to be the value $A[\lfloor\frac{(N+1)}{2}\rfloor]$ or $A[\lceil\frac{(N+1)}{2}\rceil]$ if N is even, and $A[\frac{N+1}{2}]$ if N is odd.
- *Random element*: set the pivot to be value $A[r]$, where r is a random number such that $1 \leq r \leq N$.

2.2.2 Multi-step selection

Given that the most efficient case of the partitioning is achieved when picking the median, the following methods involve computing the median of some odd sample of values extracted from the array. The difference among the methods lies in the positions of the array from which the values are taken, and in the sample size. The latter is indicated by the suffix T, with most common implementations using $3 \leq T \leq 9$

- *Median-of-T with fixed selection:* T integers $1 \leq n_1,....,n_T \leq N$ are fixed beforehand. The median of $\{A[n_1],.....,A[n_T]\}$ is then computed and used as the pivot for the partitioning.
- *Median-of-T with random selection:* T random integers $1 \leq r_1,....,r_T \leq N$ are generated. The median of $\{A[r_1],.....,A[r_T]\}$ is then computed and used as the pivot for the partitioning.

2.3 Partitioning around the pivot

mention two main schemes Figure 2 illustrates the process of partitioning around a chosen value, 4 in this case. Recalling the previous paragraph, this could have been a one-step selection using a random index.

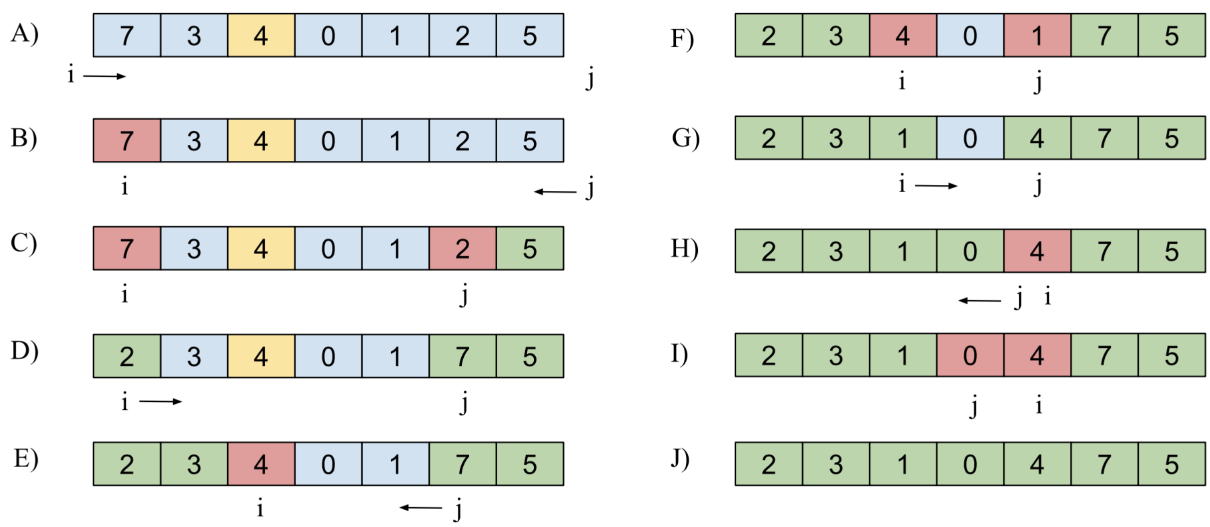

Figure 2. A picture of the same tucan looking the other way!

3 Complexity analysis

3.1 Worst-case analysis

3.2 Average-case analysis

3.3 Analysis of randomized Quicksort

4 Parallel processing

4.1 Scaling features

4.2 Parallelizing Quicksort

5 Computational geometry and the convex hull problem

Computational geometry is the study of algorithms for the solution of geometric problems in the Euclidean space [2]. The convex hull problem belongs to this class of problems, and has a wide range of applications across several disciplines (e.g. data classification, collision avoidance, image processing and recognition). It is defined as follows: given a set of points, find the smallest convex polygon containing all the points [1]. For the purpose of this project, we will consider only the planar convex hull (i.e. 2D Euclidean space).

5.1 The Quickhull algorithm

The Quickhull algorithm is a variation of Quicksort for the solution of the convex hull problem. The algorithms works as follows: an initial partitioning step is done by picking the two points with minimum and maximum x-coordinates: let these two points be p_1 and p_2, respectively. The line connecting p_1 and p_2 splits the set in two parts. Then, for each subset, we search for the farthest point from the line: let this point be p_{max}. If it exists, we partition each subset again around the lines $\overline{p_1 p_{max}}$ and $\overline{p_2 p_{max}}$, respectively, and we recursively search for the solution among the points that lie outside each line. Again, let p_1 and p_2 be the endpoints of the line in the recursive step. The stop condition occurs when p_{max} is not found, as there are no points outside the line $\overline{p_1 p_2}$. In this case we can add p_1 and p_2 to the solution set. The very first steps of the algorithm are illustrated in Figure 3.

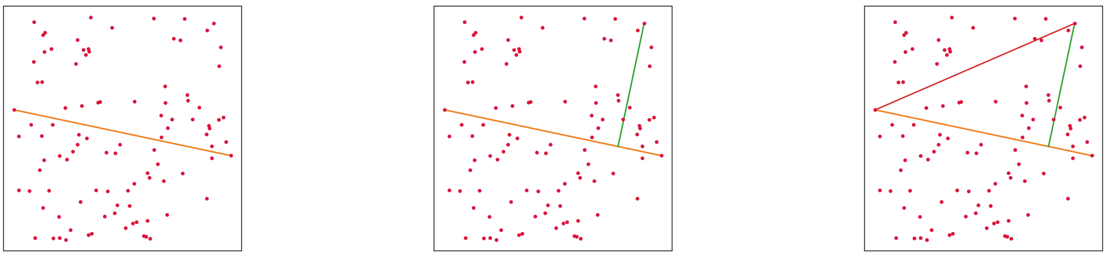

Figure 3. The set of points is partitioned in two subsets (left); the farthest point from the line is found (middle); the triangle's hypotenuse shrinks the subset of solution candidates, to be searched in the recursive step (right).

The pseudocode of the algorithm is given below:

```
pseudocode goes here
```

We can observe similarities between Quicksort and Quickhull, as both are divide-and-conquer and sorting algorithms. The former principle is applied by partitioning the domain and splitting the initial problem into smaller subproblems, whereas the latter is applied by comparing Euclidean distances between points. Again, the complexity is $O(n^2)$ in the worst case and $O(n \log n)$ in the average case.

5.2 Distributed Quickhull

The Quickhull algorithm is sequential by design, as each recursive call depends on the previous ones for the computation of the solution set [3]. Therefore, parallelizing the algorithm by multithreading is not feasible, as it is not possible to independently compute the solution to subproblems. However, the set of points can be distributed over multiple processes, and inter-process communication can be used to combine the partial results.
The pseudocode of the distributed version of Quickhull is given below:

```
parallel pseudocode goes here
```

As we can see, the procedure is almost identical to the sequential version. The key difference is the addition of collective communication (`allreduce`) among processes in the partitioning steps: the computation performed by each process to find the leftmost and rightmost points, as well as the farthest point at every iteration, is limited to a subset of points, so the results of all processes must be combined, in order to obtain the global result. Note that collective communication is used only for the aforementioned purpose, keeping the communication costs low.
Running the algorithm for large input sizes with different numbers of processes, we can observe significant gains in performance. A scaling chart is reported in Figure 4.

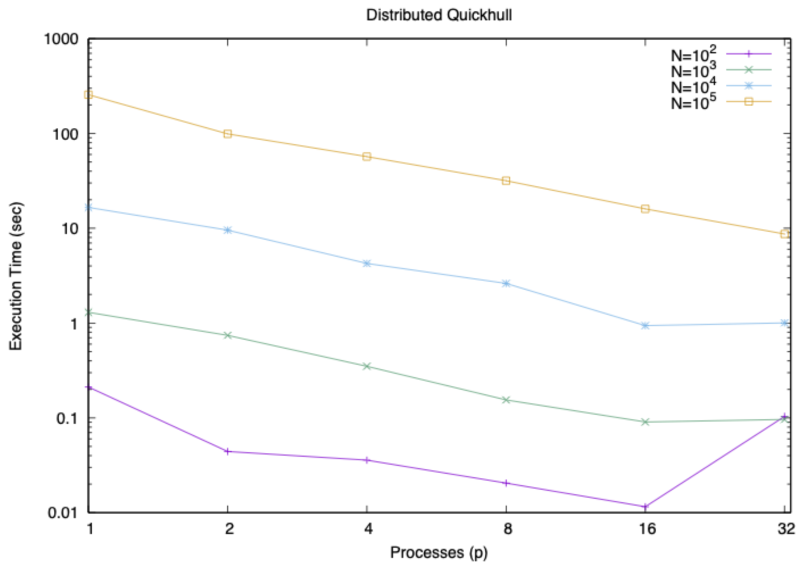

Figure 4. Performance scaling of distributed Quickhull.

We can observe the gain in performance obtained by the distributed version of the algorithm: by doubling the number of processes, the execution time roughly halves. However, we can also observe the effect of over-parallelizing with respect to the problem size: e.g. for $N = 10^2$, 32 processes perform worse than 16.

References

[1] https://en.wikipedia.org/wiki/Computational_geometry.

[2] A perspective on quicksort.

[3] S. Ramesh. Convex hull - parallel and distributed algorithms.