DC-Top-*k*: A Novel Top-*k* Selecting Algorithm and Its Parallelization

Zhengyuan Xue, Ruixuan Li[*], Heng Zhang, Xiwu Gu
School of Computer Science and Technology
Huazhong University of Science and Technology
Wuhan 430074, China
{xue, rxli, hengzhang, guxiwu}@hust.edu.cn

Zhiyong Xu
Department of Mathematics and Computer Science
Suffolk University
Boston, MA 02114, USA
zxu@mcs.suffolk.edu

***Abstract*—Sorting is a basic computational task in Computer Science. As a variant of the sorting problem, top-*k* selecting have been widely used. To our knowledge, on average, the state-of-the-art top-*k* selecting algorithm Partial Quicksort takes $C(n,k) = 2(n+1)H_n+2n-6k+6-2(n+3-k)H_{n+1-k}$ comparisons and about $C(n,k)/6$ exchanges to select the largest *k* terms from *n* terms, where H_n denotes the *n*-th harmonic number. In this paper, a novel top-*k* algorithm called DC-Top-*k* is proposed by employing a divide-and-conquer strategy. By a theoretical analysis, the algorithm is proved to be competitive with the state-of-the-art top-*k* algorithm on the compare time, with a significant improvement on the exchange time. On average, DC-Top-*k* takes at most $(2-1/k)n+O(k\log_2 k)$ comparisons and $O(k\log_2 k)$ exchanges to select the largest *k* terms from *n* terms. The effectiveness of the proposed algorithm is verified by a number of experiments which show that DC-Top-*k* is 1-3 times faster than Partial Quicksort and, moreover, is notably stabler than the latter. With an increase of *k*, it is also significantly more efficient than Min-heap based top-*k* algorithm (U.S. Patent, 2012). In the end, DC-Top-*k* is naturally implemented in a parallel computing environment, and a better scalability than Partial Quicksort is also demonstrated by experiments.**

Keywords—top-k selecting; partial sorting; quickselect; algorithm; parallelization

I. Introduction

In Computer Science, a common computational task is to sort a large number of objects, and then select the first *k* objects which suit our needs most. Let *A* be a set of size *n*, top-*k* obtains the largest *k* ($k<<n$) elements of *A* (the order of top-*k* elements is not requisite in this work). There are many important applications based on top-*k* selecting, such as recommendation [1], database and information retrieval [2,3], outlier detection and network analysis [4].

To select the top-*k* objects from *n* objects, theoretically, the necessary compare times $W_k(n) = n+(k-1)\lg n+\mathbf{O}(1)$, for any $k>=3$, in which term $\mathbf{O}(1)$ is correlated with *k* [5]. We can use Tree Selection Sort (i.e., Tournament Sort [6]) to reach this lower bound. However, Tournament Sort ignores the exchange times of elements [6], which is the reason why Tournament Sort is inefficient in real-world applications. And more importantly, it needs $\mathbf{O}(n)$ extra memory to select top-*k* elements, which is intolerable in practice. Aiming at the disadvantage of Tournament Sort, a new algorithm, Heap Sort, was proposed. Accordingly, we can find the top-*k* elements by constructing a Max-heap first, and then adjusting the heap *k* times. This algorithm has a worst-case $\mathbf{O}(n)+\mathbf{O}(k\log_2 n)$ runtime. Enlightened by Heap Sort, a new top-*k* algorithm was proposed based on Min-heap recently [3,7]. But this algorithm has a $\mathbf{O}(n\log_2 k)$ runtime, which decreases the efficiency dramatically, especially when *k* is large. Another main method for solving top-*k* problem is based on Quicksort and its variants (e.g., Partial Quicksort [8], Incremental Quicksort [9,10]), the main difference between these algorithms and traditional Quicksort is that the recursion carries on only in the groups which include top-*k* elements until all the *k* top elements have been found. The complexity is linear, i.e., $\mathbf{O}(n)$. In practice, Quicksort-based top-*k* algorithms have good average performance.

To the best of our knowledge, the most efficient algorithm for top-*k* problem at present is Partial Quicksort [4,8,9] which combines Quickselect and Quicksort [11]. The main weaknesses of this algorithm are as follows: Firstly, it is hard to implement Partial Quicksort in a parallel computing environment. The reason is as follows: the processes in Quicksort are not independent of each other, most of the related works about parallel Quicksort [12-15] adopt recursive method, in which the processes need to communicate with each other frequently. Note that $k<<n$ in most top-*k* selection problems, the communication cost in Partial Quicksort is relatively more serious than Quicksort when adopt recursive method. To our knowledge, there is no other work which formally proposes efficient parallel top-*k* selecting algorithm which can avoid recursion. Secondly, although the performance is of satisfactory on average, the execution time is not stable, and it can degenerated to $\mathbf{O}(n^2)$ in the worst case. The fluctuant efficiency lead to more time delay on synchronization in parallel computing environment. This can affect the user experience, especially in real time top-*k* retrieval systems (e.g., Search Engine System). Obviously, this is determined by the nature of Quicksort algorithm itself. Although we can use some methods [16-18] rather than randomly select the pivot to mitigate this problem somewhat, the efficiency maybe greatly reduced.

[*]Corresponding author.

To overcome these weaknesses, we propose a new algorithm called DC-Top-*k*. In summary, our contributions in this paper are as follows:

Firstly, we propose an efficient top-*k* selecting algorithm, called DC-Top-*k*, based on a divide-and-conquer strategy. The main idea of DC-Top-*k* is that we divide the top-*k* problem to *k* top-1 problems, and then subtly find out the relationship between the *k* top-1 terms and the final top-*k* terms. Thus DC-Top-*k* get high efficiency and moreover, it is suitable for parallelization.

Secondly, we naturally parallelize DC-Top-*k* algorithm in a parallel computing environment with MPI (Message Passing Interface). Since DC-Top-*k* seldom needs recursion and communication between each process, it is obvious for parallel DC-Top-*k* algorithm to get a significant speedup compared with the serial one.

Thirdly, we compare DC-Top-*k* with other top-*k* selecting algorithms which include the most efficient top-*k* selecting algorithm at present, both in theory and by a number of experiments, to verify the effectiveness of DC-Top-*k* algorithm. Our experiments show that DC-Top-*k* has more high and stable performance than Partial Quicksort, moreover the scalability is also better.

The rest of paper is organized as follows. Section II presents a review of the related work. Section III proposes the algorithm DC-Top-*k*, carries out a theoretical analyses, and then parallelizes the algorithm. Section IV implements some experiments. Finally, Section V concludes the paper.

II. Related Work

Compared with well-known full sorting algorithm [19-23], researches on top-*k* selecting algorithm are relatively few. We generalize the related work to three aspects.

A. *Quicksort and Quickselect*

Since Quicksort [20] and Quickselect (i.e., Find) [24] were proposed, many researchers have made deep study on them because of their outstanding performance. Martínez *et al.* [16,18] analyze the optimization of sampling strategy in Quicksort and Quickselect. Sedgewick [11] and Martínez *et al.* [25] study the average compare times, exchange times and move times in Quicksort. Devroye [26] makes analysis of the worst running time in Hoare's Selection (i.e., Find) from the aspect of probability. Fill *et al.* [27] study the bit compare times of Quickselect and find that the increase of its mathematical expectation gradually tend to linear increase as the key words increase. Blum *et al.* [17] also point out that the number of comparisons required to select the *i*-th smallest of *n* numbers is shown to be at most a linear function of *n* by analysis of a new selection algorithm. Furthermore, Cunto *et al.* [28] propose that any algorithm requires an average of at least $n+k-\mathbf{O}(1)$ comparisons, on average, to find the *k*-th smallest of *n* numbers. Recently, Fouz *et al.* [29] make smoothed analysis on Quicksort and Find. While in the worst case, the number of comparisons that Find needs is $\mathbf{O}(n^2)$, it is $\mathbf{O}(n)$ on average. They get the average compare times and its boundary.

B. *Partial sort and top-k*

Based on Quicksort, Martınez [8] proposes Partial Quicksort to solve top-*k* problem. On this basis, Kuba [30] gives a general solution. After referring to Find, Martínez *et al.* [31] put forward Quickpartitionsort, with asymptotic running time $c_1k\ln k+c_2k+n+\boldsymbol{o}(n)$. Navarro *et al.* [9,10] point out that top-*k* can be done in optimal $\mathbf{O}(n+k\log k)$ time. They present the Incremental Quicksort (IQS), in which calls Quickselect to find the minimums of arrays $r[0..n-1]$, $r[1..n-1]$, ... , $r[k-1..n-1]$, so that the first *k* elements are obtained in optimal expected time. IQS performs better in practice than existing online algorithms. Afshani *et al.* [32] and Navarro *et al.* [2] improve top-*k* by optimizing the data structure. Recently, Niu *et al.* [3] proposed a Min-heap based top-*k* (see also [7], a U.S. patent for invention, 2012). But this algorithm can not reach linear running time when *k* is not small enough, i.e., the runtime is related to *k*. Huang *et al.* [33] focus on the top-*k* selection problem in evolving data. Besides, it is worth noting that the well-know Threshold Algorithm (TA) proposed by Fagin *et al.* [34] is under the model that the input consists of *m sorted* lists with *n* data items, which is different with our top-*k* problem.

C. *Parallel top-k selecting*

About parallel top-*k* selection algorithms, there are also some works [35-37]. But [35,36] both considered the *sorted* models. If we follow the idea in [36], and consider the *unsorted* models, we should first select a element as a pivot, and partition all the processing elements (PEs) to two parts with this pivot in parallel. Then in each recursion, we must get the number of elements in one side of the pivot, then select another element as a pivot for the next recursion, until the number of the elements in the right side of the pivot is *k*. In fact, this is just a parallel version of Partial Quicksort. To deal with *unsorted* input, [37] proposes a Communication Efficient Selection algorithm (see Figure 1 in [37]). Its basic idea is similar with [36]. The main difference lies in that instead of using the binary search in *sorted* lists, [37] uses FR-Select [38], a modification of Quickselect using two pivots. Both of the methods based on the idea of Partial Quicksort, though the selection algorithm they use may be a bit different. They involve repeated communication, and severely limit the scalability, especially in the parallel computing environment with distributed memory.

III. Dc-top-k Algorithm

The key of divide-and-conquer is to divide a large scale problem into several independent small scale sub-problems. In fact, divide-and-conquer strategy exists in lots of sorting algorithms (e.g., Mergesort, Heapsort [19], and Quicksort [20], etc.). Based on divide-and-conquer strategy, in this section we design and implement DC-Top-*k* algorithm.

A. *Design of DC-Top-k algorithm*

The basic idea: Suppose the input array consist of *n* elements. Divide $r[0..n-1]$ into *k* groups and select the local maximum in each group, i.e., divide the top-*k* problem to *k* top-1 problems. Then we get *k* local maximums. From a probabilistic perspective, *k* local maximums partly overlap with *k* global maximums. We set the minimum in the local maximums as the *threshold*. Then we select all the numbers which are larger than the *threshold*, and put them in the valid elements set. Finally, we select top-*k* from the valid elements. See Algorithm 1 for the details of DC-Top-*k*.

Algorithm 1: DC-Top-*k* algorithm

```
Input: r[0..n-1]: the n source elements
       k: the total number of top elements we need
Output: top[0..k-1]: The top-k elements in r[0..n-1]
1  Divide r[0..n-1] into k groups, each of which has N=n/k elements;
2  for i ← 0 to k-1 do
3    Max[i] ← the maximum of group i;
4    Maxloc[i] ← the position of Max[i] in the group i;
5  endfor
6  threshold ← the minimum of Max[0..k-1];
7  $G_{\min}$ ← the group id of threshold;
8  Q ← 0;
9  for i ← 0 to k-1 do
10   if i == $G_{\min}$ then
11     continue;
12   endif
13   for j ← 0 to N-1 do
14     if r[i*N+j]>threshold and j!=Maxloc[i] then
15       r[Q] ← r[i*N+j];
16       Q ← Q+1;
17     endif
18   endfor
19 endfor
20 for i ← 0 to k-1 do
21   r[Q+i] ← Max[i];
22 endfor
23 top[0..k-1] ← the largest k elements of r[0..k+Q-1];
24 return top;
```

In Algorithm 1, line 1 divide $r[0..n-1]$ into k groups with the same size according to the sequence of n elements (Here we suppose that n is divisible by k. If not, then we can adjust the number of groups slightly so that each group has a balanced size). Lines 2-5 select the maximum in each group and all these k maximums make up array *Max*[]. Lines 6-7 select the minimum from *Max*[] as the *threshold* and set down its group id $G_{\min}$. Lines 8-19 select the elements which are bigger than *threshold* from all the groups except $G_{\min}$, and except the k local maximums. Denote the number of selected elements as Q, and put the elements into $r[0..Q-1]$ (note that, to save storage space, we put the valid elements into $r[]$ rather than into a new array). Lines 20-22 put the k local maximums into $r[Q..k+Q-1]$. Then elements in $r[0..k+Q-1]$ (totally $(k+Q)$) are valid elements. Line 23 use specified underlying algorithm (this will be discussed later) to select top-*k* from these valid elements.

It seems that it is difficult to judge whether this method is efficient or not, since we don't know Q is small or large. Fortunately, suppose that the input elements consist of n distinct real numbers ordered randomly, later we will prove that Q is small enough compared with n as Q is only related with k which is far smaller than n (see Theorem 1).

B. Theoretical analyses

First we analyze three classic sorting algorithms (i.e., Bubblesort, Heapsort [19], and Quicksort [20]) and two outstanding top-*k* algorithms (i.e., Partial Quicksort [4,8,9] and Min-heap based top-*k* algorithm [3,7]), then we analyze DC-Top-*k* algorithm, at last we theoretically compare DC-Top-*k* with Partial Quicksort algorithm.

1) Classic sorting algorithms

Bubblesort: When dealing with top-*k* problem, we can get top-*k* result after *k* round comparisons. Its running time is $\mathbf{O}(n)$ in the best case, $\mathbf{O}(kn)$ in the worst case, and $\mathbf{O}(kn)$ on average; its space complexity is $\mathbf{O}(1)$. **Heapsort:** When dealing with top-*k* problem, for building an initial heap needs $\mathbf{O}(n)$ complexity, adjusting the heap needs $\mathbf{O}(\log_2 n)$ complexity, and the total number of adjusting the heap is k, so its running time is $\mathbf{O}(n) + \mathbf{O}(k\log_2 n)$, both in the best and worst case; its space complexity is $\mathbf{O}(1)$. **Quicksort:** We sort all the items to get top-*k*. Its running time is $\mathbf{O}(n\log_2 n)$ in the best case, $\mathbf{O}(n^2)$ in the worst case, and $\mathbf{O}(n\log_2 n)$ on average. Its space complexity is $\mathbf{O}(\log_2 n)$ in the best case, $\mathbf{O}(n)$ in the worst case, and $\mathbf{O}(\log_2 n)$ on average.

2) Partial Quicksort algorithm

Quicksort is one of the most efficient sorting algorithms. However, it is full sorting and inappropriate to directly deal with top-*k* problem. Thus, researchers proposed Partial Quicksort [4,8,9]. The advantage of Partial Quicksort over traditional Quicksort is that the recursion carries on only in the groups which include top-*k* elements until all the top-*k* elements are found. According to [9], find the *k*-th largest element of array $r[0..n-1]$ using $\mathbf{O}(n)$ time Select algorithm at first, and then collect and sort the elements larger than the *k*-th element. The resulting complexity, $\mathbf{O}(n+k\log k)$, is optimal under the comparison model. In their algorithm they used Find [24] to solve this problem and it performs better in practice than the best existing online algorithm. In fact, it is almost the same with Partial Quicksort [8], and the QuickSortTopK algorithm (see Algorithm 1 in [4]). These algorithms are all based on Quickselect and are denoted by Partial Quicksort in the rest of this paper. According to [4,8,9], the running time of Partial Quicksort is $\mathbf{O}(n+k\log_2 k)$ in the best case, $\mathbf{O}(n^2)$ in the worst case, and $\mathbf{O}(n+k\log_2 k)$ on average. Its space complexity is $\mathbf{O}(\log_2 n)$ in the best case, $\mathbf{O}(n)$ in the worst case, and $\mathbf{O}(\log_2 n)$ on average.

3) Min-heap based top-k algorithm

Min-heap based top-*k* algorithm [3,7] is as follows: first, select the first *k* elements from the source data to build a min-heap; second, traverse other elements. If any element is bigger than the root node of the min-heap, it replaces the root node. Accordingly, the min-heap is adjusted. This operation goes on until all the elements in the source data have been traversed. Then, all the *k* elements in the min-heap are the top-*k* elements. According to [3,7], Min-heap based top-*k* algorithm mainly consists of building an initial heap and rebuilding a new heap repeatedly. The running time for building an initial heap is $\mathbf{O}(k)$. Adjusting the heap needs $\mathbf{O}(\log_2 k)$ complexity each time and the total number of adjusting the heap is $\mathbf{O}(n-k)$. Therefore, the total time complexity is $\mathbf{O}(n\log_2 k)$ on average. The best and worst time complexity are $\mathbf{O}(n)$ and $\mathbf{O}(n\log_2 k)$, respectively. Its space complexity is $\mathbf{O}(1)$, since it is only fixed auxiliary space needed to construct and adjust the min-heap.

4) Our DC-Top-k algorithm

In DC-Top-*k*, n elements are divided into k groups. The number of elements in each group is $N= n/k$. We consider the average total cost which takes into account the cost of both compare operation times *Cmp* and exchange operation times

Exc. If we do not care about the elements order of the top-*k* result, then according to the algorithm flow, we can get

$$Cmp = (n-k)+(k-1)+(N-1)*(k-1)+V_{Cmp}$$
$$= 2n - n/k - k + V_{Cmp} \quad \textbf{(1)}$$
$$Exc = (k+Q)+V_{Exc} \quad \textbf{(2)}$$

Where V_{Cmp} and V_{Exc} are the compare times and the exchange times required to select top-*k* from *r*[0..*k*+*Q*-1] respectively. See Algorithm 1 for the definition of *Q*. About **(2)**, strictly speaking, the component "*k*+*Q*" is the number of copying (where the *k* elements are copied for 2 times) rather than exchanging elements, but here we consider it as the same with exchanging (in fact, copying costs less than exchanging).

In order to get the average efficiency of DC-Top-*k* algorithm, we need to get the Expected value of *Q* at first. Note that the elements are typically assumed to be distinct [5,8,21,30], only minor adjustments are necessary to cope with duplicate elements. For analysis purposes, DC-Top-*k* is also based on this assumption, but it still works even if the elements are not distinct.

We assume, as it is usual in the analysis of comparison-based sorting algorithms, that any permutation of the given distinct *n* elements *r*[0..*n*-1] appears with equal possibility, i.e., all *n*! input orderings are with the probability of 1/*n*!. In this case, we say the given distinct *n* elements are "ordered randomly", and note this characteristic as *r*[0..*n*-1]~$\boldsymbol{Rand_n}$.

This assumption is standard in the probabilistic analysis of comparison-based sorting and selection algorithms [8,16,30]. In fact, the assumption can be removed if we introduce randomness in DC-Top-*k* algorithm (e.g., in the grouping procedure). When the source of randomness comes from the algorithm itself, DC-Top-*k* can works not by assuming any particular distribution on the inputs. Both approaches yield the same results, but we will talk in terms of the random permutation model for the rest of paper.

About the Expected value of *Q*, we have Theorem 1 which is based on above assumption.

Theorem 1. In Algorithm 1, suppose that the input elements *r*[0..*n*-1] are consist of *n* distinct real numbers, *r*[0..*n*-1]~$\boldsymbol{Rand_n}$. Then the Expected value of *Q* (denoted as E(*Q*)) satisfies E(*Q*)≈kH_k. H_k is the *k*-th harmonic number.

Proof. Suppose that *n* real numbers *r*[0..*n*-1]={e_1, e_2, …, e_n} are ordered randomly. Denote the *j*-th largest element as m_j. Then the probability of the *i*-th element e_i is just m_j

$$P(e_i = m_j) = (n-1)!/n! = 1/n \quad (1<=i<=n,\ 1<=j<=n)$$

Denote $\boldsymbol{R_s}$ as a set of *s* elements, $\boldsymbol{R_s} \subset r[0..n-1]$, and rm_s as the maximum in $\boldsymbol{R_s}$ (1<=*s*<*n*). For any element $e_i \notin \boldsymbol{R_s}$ (i.e., $e_i \in r[0..n-1]-\boldsymbol{R_s}$, totally *n*-*s*), we get the probability of $e_i > rm_s$

$$P(e_i > rm_s) = \sum_{j=1}^{n-s} P(e_i = m_j)P(e_i > rm_s \mid e_i = m_j)$$
$$= (1/n) * \sum_{j=1}^{n-s} A_{n-j}^{s} A_{n-1-s}^{n-1-s} / (n-1)!$$
$$= (n-1-s)!s!/n! \sum_{j=1}^{n-s} C_{n-j}^{s} = (n-1-s)!s!/n! C_n^{s+1}$$
$$= 1/(s+1) \quad \textbf{(3)}$$

Let *k* stand for the number of top elements and the number of groups (*k*<<*n*). *N*=*n*/*k* stands for the number of elements in each group. Denote $\boldsymbol{G_t}$ as a set of *t* groups totally *tN* elements, $\boldsymbol{G_t} \subset r[0..n-1]$, and gm_t as the maximum in $\boldsymbol{G_t}$ (1<=*t*<*k*). For any element $e_i \notin \boldsymbol{G_t}$ (i.e., $e_i \in r[0..n-1]-\boldsymbol{G_t}$, totally *n*-*tN*), we can get the probability of $e_i > gm_t$

$$P(e_i > gm_t) = \sum_{j=1}^{n-tN} P(e_i = m_j)P(e_i > gm_t \mid e_i = m_j) = 1/(tN+1)$$

according to **(3)**. Denote the number of all the elements which are bigger than gm_t as Q_t, we know that the random variable Q_t is binomial distributed, i.e., Q_t~B((*k*-*t*)*N*, 1/(*tN*+1)), according to the above analysis. So

$$E(Q_t) = (k-t)N/(tN+1) \approx k/t - 1$$

Denote the number of all the elements which are bigger than *threshold* as *Q*', we know that *Q*'=*k*+*Q*. According to the *Inclusion-exclusion Principle*, we get

$$E(Q') = C_k^1 E(Q_1) - C_k^2 E(Q_2) + C_k^3 E(Q_3) - \ldots + (-1)^k C_k^{k-1} E(Q_{k-1})$$
$$= \sum_{t=1}^{k-1} (-1)^{t+1} C_k^t E(Q_t) \approx \sum_{t=1}^{k-1} (-1)^{t+1} C_k^t (k/t - 1)$$
$$= k \sum_{t=1}^{k-1} (-1)^{t+1} C_k^t / t - \sum_{t=1}^{k-1} (-1)^{t+1} C_k^t$$
$$= k(H_{k-1} + 1) - 1 - (-1)^k \quad \textbf{(4)}$$

According to *Q*'=*k*+*Q*, we get the Expected value of *Q*

$$E(Q) = E(Q') - k = kH_{k-1} - 1 - (-1)^k \approx kH_k \quad \textbf{(5)}$$

where H_k denotes the *k*-th harmonic number

$$H_k = \sum_{i=1}^{k} \frac{1}{i} = \ln k + \gamma + o(1) \quad \textbf{(6)}$$

and *γ* the Euler constant. ■

From Theorem 1, we can see that E(*Q*) not about the data size *n*, but about the result size *k*, when elements are ordered randomly. Because *k*<<*n*, the value of E(*Q*) is not very large compared with *n*. For example, when *k*=10, then E(*Q*)≈28, according to **(5)**. That is, as for the problem of selecting top-10 from a large number of elements, after comparing with the *threshold*, the number of valid elements is about merely 28 on average. Moreover, from **(5)** we know that the ratio of E(*Q*) to *k* appears a logarithm increase with the linear increase of *k*. Based on the above analysis, we know that DC-Top-*k* has high efficiency on average, and its efficiency improves with the increase of *n* and decrease of *k*.

Besides, the selection of the underlying sorting algorithm in Algorithm 1 influences the values of V_{Cmp} in **(1)** and V_{Exc} in **(2)**. It has proved that the Expected value of *Q* is only correlated with *k*. So the valid (*k*+*Q*) elements accounts for a small proportion of the total *n* elements. Based on this characteristic, more appropriate and more efficient algorithm should be chose as the underlying algorithm. Based on the complexity analysis of various sorting algorithms, later we will explore a more practical and appropriate algorithm, and then theoretically compare DC-Top-*k* with Partial Quicksort.

5) Algorithm complexity and comparative analysis

Based on the above analyses, Table 1 concludes the average complexities of various algorithms solving top-*k* selection problem.

Considering that Partial Quicksort and DC-Top-*k* have the same time complexity (i.e., optimal complexity **O**(*n*)), next we will analyze their required basic operation times, both not including the sorting for the result top-*k* elements. Note that the underlying sorting algorithm, to some extent, affects the efficiency of DC-Top-*k*. We need decide which one is suitable to be used as the underlying algorithm. In the following analysis, we suppose Partial Quicksort is used for this purpose (See Experiment *B* in Section IV for details).

TABLE I. THE AVERAGE COMPLEXITIES OF VARIOUS ALGORITHMS

	Average time complexity	**Average space complexity**
Bubblesort	$\mathbf{O}(nk)$	$\mathbf{O}(1)$
Quicksort	$\mathbf{O}(n \log_2 n)$	$\mathbf{O}(\log_2 n)$
Heapsort	$\mathbf{O}(n)+\mathbf{O}(k \log_2 n)$	$\mathbf{O}(1)$
Min-heap based top-*k* [3,7]	$\mathbf{O}(n \log_2 k)$	$\mathbf{O}(1)$
Partial Quicksort [4,8,9]	$\mathbf{O}(n)$	$\mathbf{O}(\log_2 n)$
DC-Top-*k*	$\mathbf{O}(n)$	$\mathbf{O}(k)$

We still assume that the input array $r[0..n\text{-}1]$ consist of n distinct elements ordered randomly. We consider the total cost of both comparisons and exchanges on average.

a) Average operation times of Partial Quicksort

According to [8,30,31], the average number of key comparisons in Partial Quicksort is given by

$$C_1(n,k)=2(n+1)\mathrm{H}_n+2n-6k+6-2(n+3-k)\mathrm{H}_{n+1-k} \quad \textbf{(7)}$$

When $k<<n$ we can get

$$C_1(n,k)\approx 2n-4k+4+(2k-4)\mathrm{H}_n \quad \textbf{(8)}$$

Suppose the pivot elements are selected randomly each time, and we denote the i-th round compare times of Partial Quicksort as c_i. Then the number of elements pairs which need to be exchanged is $s_i \approx c_i/6$ [8,11], and two subgroups *MAX* and *MIN* formed by dividing each have $c_i/2$ elements in average case. This operation continues until the number of elements which need to be compared is reduced to k. So the average exchange times of Partial Quicksort is

$$E_1(n,k)\approx C_1(n,k)/6\approx(n-2k+2+(k-2)\mathrm{H}_n)/3 \quad \textbf{(9)}$$

b) Average operation times of DC-Top-k

As for the average basic operation times of DC-Top-*k*, we have Theorem 2.

Theorem 2. In Algorithm 1, suppose that the input elements $r[0..n\text{-}1]$ are consist of n distinct real numbers, $r[0..n\text{-}1]\sim\boldsymbol{Rand_n}$. We use Partial Quicksort as the underlying algorithm. Then the average compare times and exchange times are $Cmp = (2\text{-}1/k)n+\mathbf{O}(k\log_2 k)$, and $Exc = \mathbf{O}(k\log_2 k)$.

Proof. According to **(1)(2)** and **(8)(9)**, and based on the conclusion $\mathrm{E}(Q)\approx k\mathrm{H}_k$ in Theorem 1, in average case, we get

$$V_{Cmp}\approx C_1(k\mathrm{H}_k+k,k)$$
$$\approx(4k-4)\mathrm{H}_k-2k+4+(2k-4)\ln(\mathrm{H}_k+1)=\mathbf{O}(k\ln k)$$
$$V_{Exc}\approx C_1(k\mathrm{H}_k+k,k)/6=\mathbf{O}(k\ln k)$$

So we get the average basic operation times

$$Cmp{=}2n-n/k-k+V_{Cmp}{=}(2-1/k)n+\mathbf{O}(k\log_2 k) \quad \textbf{(10)}$$
$$Exc=k+E(Q)+V_{Exc}=\mathbf{O}(k\log_2 k) \quad \textbf{(11)}$$

■

Now we consider the average total cost of Partial Quicksort and DC-Top-*k*. According to [18], we define the total cost as a weighted sum of exchanges and comparisons. Let ξ_1 denote the unit cost of a comparison, ξ_2 denote the unit cost of an exchange. From a practical standpoint, if we take ξ_1=4 and ξ_2=11 as representative values for the cost of comparisons and exchanges (as suggested in Knuth, 1998), then based on **(8)(9)** and **(10)(11)**, respectively, we get the average total cost of Partial Quicksort and DC-Top-*k*

$$Cost_1=C_1(n,k)*4+E_1(n,k)*11\approx 11.67(n-2k+2+(k-2)\mathrm{H}_n) \quad \textbf{(12)}$$
$$Cost_2=Cmp*4+Exc*11=(8-4/k)n+\mathbf{O}(k\log_2 k) \quad \textbf{(13)}$$

By comparing **(12)** with **(13)**, we find that when $k<<n$, the average total cost of DC-Top-*k* is less than that of Partial Quicksort. And the theoretical speedup ratio is about 11.67/8 ≈ 1.46. One important factor is that DC-Top-*k* seldom requires the operation of exchanging while Partial Quicksort requires in most cases. Hence, though the difference between their compare times is little, DC-Top-*k* has better performance than Partial Quicksort in practice.

c) Analysis of the worst cases

For Partial Quicksort, we know that its worst complexity is $\mathbf{O}(n^2)$ [8,9,29]. For DC-Top-*k*, according to **(1)(2)**, the total compare times *Cmp* and the total exchange times *Exc* are $Cmp = 2n\text{-}n/k\text{-}k+V_{Cmp}$ and $Exc = k+Q+V_{Exc}$, where V_{Cmp} and V_{Exc} are the compare times and the exchange times required to select top-*k* from $(k+Q)$ elements respectively. In the most extreme case, when the n/k elements in the group which contains the *threshold* are just the smallest n/k elements among the total n elements; at the same time, the underlying algorithm (Partial Quicksort) also has the worst efficiency, i.e., $\mathbf{O}(n^2)$, then DC-Top-*k* has the worst efficiency. In this case, $Q_{\max} = n\text{-}n/k\text{-}k+1$, so the basic operation times

$$Time_{\text{worst}}=\mathbf{O}(Q_{\max}{}^2)=(1-1/k)^2\mathbf{O}(n^2) \quad \textbf{(14)}$$

From **(14)**, $Time_{\text{worst}}$ is smaller than $\mathbf{O}(n^2)$. It can be seen that comparing DC-Top-*k* with Partial Quicksort both in the worst cases, we can still get higher efficiency. Moreover, note that DC-Top-*k* is related to the underlying algorithm, so if the worst case of the underlying algorithm improved, the worst case of DC-Top-*k* will be improved, accordingly.

C. Parallel implementation

Mass data processing can not do without parallel computing. In this section, we parallelize DC-Top-*k* with MPI in distributed platform. Recall that we first divide $r[0..n\text{-}1]$ into k groups, select the local top-1 in each group. Then, set the minimum in the k elements as the *threshold*, and select the elements which are larger than *threshold*. Finally, use underlying algorithm to find top-*k* from them. Clearly, almost all the procedures (except the last step) are suitable for parallel computing. See Algorithm 2 for the parallel DC-Top-*k* algorithm. For discussion, we suppose k is divisible by p (the number of processes), else we only need minor adjustments.

In Algorithm 2, there are p PEs with n/p elements (i.e., k/p groups) in each PE. Sending a message of size l elements takes time $\alpha+\beta l$. The parameter α is start-up overhead and β the time to communicate one element. Then in Algorithm 2: (a) lines 7-10, each PE finds out the local maximums within the scope of their own groups, and then finds out the minimum *min* in the local maximums. This requires $(n/k\text{-}1)*k/p+(k/p\text{-}1)$

= (n/p-1) comparisons and (k/p+1) copying; (b) lines 11-12, 0# PE reduces all the *min* elements to the minimum *threshold*, and broadcasts it to other PEs. (c) lines 13-18, each PE finds out the elements which are bigger than the *threshold*. This requires n/p comparisons and ($k\ln k$)/p copying; (d) lines 19-32, 0# PE gathers ($k\ln k$)/p elements from each PE and then selects the final top-k from theses $k\ln k$ elements serially.

Algorithm 2：Parallel DC-Top-*k* algorithm

```
Input：r[0..n-1]: the n source elements
       k: the total number of top elements we need
Output：top[0..k-1]: The top-k elements in r[0..n-1]
1  MPI_Init();
2  MPI_Comm_rank(MPI_COMM_WORLD, &myid);
3  MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
4  eachProcess ← k/numprocs;
5  count ← n/numprocs;
6  random_produce(databuf,count,myid);
7  for j ← 0 to eachProcess-1 do
8     max[j] ← the maximum in databuf[n/k*j..n/k*j+n/k-1];
9  endfor
10 min ← the minimum in max[0..eachProcess-1];
11 MPI_Reduce(&min,&threshold, 1, MPI_DOUBLE, MPI_MIN, 0,
      MPI_COMM_WORLD);
12 MPI_Bcast(&threshold, 1, MPI_DOUBLE, 0, MPI_COMM_WORLD);
13 for j ← 0 to count-1 do
14    if databuf[j]>=threshold then
15       validnum[v] ← databuf[j];
16       v++;
17    endif
18 endfor
19 MPI_Reduce(&v, &allv, 1, MPI_INT, MPI_SUM, 0,
      MPI_COMM_WORLD);
20 MPI_Gather(&v, 1, MPI_INT, myv, 1, MPI_INT, 0,
      MPI_COMM_WORLD);
21 if myid==0 then
22    for i← 0 to numprocs-1 do
23       disp[i] ← 0;
24       for j ← 0 to i-1 do
25          disp[i] ← disp[i]+myv[j];
26       endfor
27    endfor
28 endif
29 MPI_Gatherv(validnum, v, MPI_DOUBLE, allvalidnum, myv, disp,
      MPI_DOUBLE, 0, MPI_COMM_WORLD);
30 if myid==0 then
31    top ← PartialQuickSort(allvalidnum,0,allv-1,k);
32 endif
33 MPI_Finalize();
34 return top;
```

We still consider the copying operations to be the same with exchanging operations. Then on average, the total number of comparisons and exchanges respectively

$$Cmp_2 \approx 2n/p + \mathbf{O}((k\log_2 k)/p) \quad \textbf{(15)}$$

$$Exc_2 \approx (k+E(Q))/p + V_{Exc} = \mathbf{O}(k\log_2 k) \quad \textbf{(16)}$$

About the communication operations in MPI, such as *Reduce*, *Broadcast*, and *Gather* [37], the total complexity is $\mathbf{O}((\beta k\ln k)+\alpha\log p)$.

According to **(15)(16)**, if $n>>k$ and n is large enough that the cost of communication can be ignored, then the speedup ratio of the parallel DC-Top-k against the serial one tends to be p, the optimal speedup. Moreover, it is worth noting that the underlying algorithm is still serial, so there is still room for improvement.

IV. Performance Evaluation

In this section we test the performance of DC-Top-k algorithm. Experiments A to D test Algorithm 1 (in serial execution) in a single machine, and they are all carried out on the same platform and resources. Experiment E test Algorithm 2 (in parallel execution) in a cluster with 20 nodes. The experimental environment is in Table 2.

All the elements with double precision are generated by the computer randomly, with a range of 1 to 10^8. The C standard library's *clock*() function is called to time the Experiments A to D, and MPI library's *MPI_Wtime*() function is called to time the Experiment E.

TABLE II. The experimental environment

	Experiments *A* to *D*	**Experiment *E***
CPU	Intel Core i3-3110M CPU @ 2.40GHz	Intel Xeon CPU E5-2670 0 @ 2.60GHz
Memory	4GB	64GB
OS	Windows7	Linux Red Hat 4.4.5-6
Language	C++	C++
Software	Microsoft VS2012 Version 4.5.5.709	OpenMPI Version 1.6
Data-sets	Synthetic data-set	Synthetic data-set

A. The relationship between Q and k in Algorithm 1

Recall that in Algorithm 1, n elements are divided into k groups, and the k local maximums are exactly the top-k elements in ideal conditions (i.e., Q=0). But in the extreme case, Q value may come to $Q_{\max} = n\text{-}n/k\text{-}k+1$, which is close to n. So the value of Q is of crucial importance to the overall performance of DC-Top-k. In Theorem 1 we get that E(Q) is about $k\ln k$. In this experiment we test the actual value. To get more precise results, we repeat the experiment 100 times and take the average value as the result. The ratio of Q to k in different data size is shown in Fig.1.

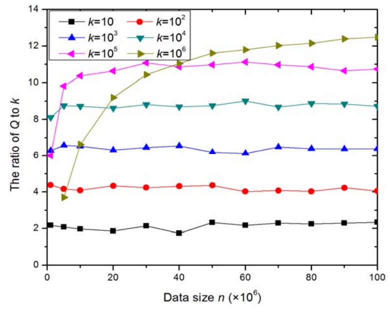

Fig. 1. The ratio of Q to k in different data size

By analyzing the experimental data in Fig.1, it can be found that when k is fixed and the data size n increases, the ratio of Q to k tends to be stable; as k increases, it increases very slow (it is similar with the logarithmic increase). This result is in accordance with Theorem 1 on Expected value of Q, which we have discussed in Section III, theoretical analyses.

At the same time we notice that, when k is very large (e.g., k=10^6) then the ratio of Q to k seems not to be a constant. This is because the current data size n is not large enough compared with k. We deduce that when n/k>100, the ratio of Q to k tends to be stable, according to the experimental result. In other words, the advantage of DC-Top-*k* can be reflected more clear when n/k>100. Of course, this result may be related to the range of the data size, more precise expression is expecting for further research.

B. The selection of the underlying sorting algorithm

The underlying algorithm, to some extent, affects the efficiency of DC-Top-*k*. Therefore, we need decide which one is suitable to be used as the underlying algorithm. From Table 1 we can see that, Partial Quicksort [4,8,9] is the best (i.e., linear complexity) in most cases. The classic Heapsort and Min-heap based top-*k* [3,7] take the second place (when $k<<n$, its complexity is approximately linear). The classic Bubblesort and Quicksort come last. Here we test these algorithms to verify their efficiency. Note that as for Bubblesort, because of its low efficiency, only top-10 to top-1,000 are tested (though we narrowed the testing scale, the performance comparison would not be influenced in essence). As for other algorithms, top-10 to top-10^6 are tested. We repeat the experiment 100 times and take the average value as the result. The result is shown in Fig.2.

From Fig.2 we find that on average, Partial Quicksort is indeed more efficient than other classic sorting algorithms. However, according to our observation, the experimental curve of Partial Quicksort would gradually fluctuate with the increase of data size *n*. Besides, Partial Quicksort is more random in a single experiment and its result is more fluctuant than other sorting algorithms, i.e., the performance is not stable. Moreover, when the needed top-*k* account for a very small proportion in total elements (e.g., below 10^{-4}, see Fig.2(a)), it is possible that Min-heap based top-*k* has advantage over Partial Quicksort. However, when the percentage of the needed top-*k* increases, Partial Quicksort will be much better than others (see Fig.2(c)).

Based on the above analysis, it is a good choice to adopt Partial Quicksort as the underlying algorithm. (1) Partial Quicksort is more efficient than other algorithms on average, which may benefit the efficiency. (2) Before the underlying algorithm is utilized, the valid elements have been greatly reduced, i.e., it only needs applying to a small data size. (3) For small data size, it is difficult for Min-heap based top-*k* [3,7] to reach high efficiency, but it is easy for Partial Quicksort. Furthermore, the drawback of randomness in Partial Quicksort will be effectively avoided.

C. The overall performance of DC-Top-k algorithm

This experiment uses Partial Quicksort as the underlying algorithm, aiming to test the overall performance of DC-Top-*k* algorithm. We repeat the experiment 100 times and take the average value as the result. The performance of DC-Top-*k* compared with various top-*k* algorithms under different data sizes is shown in Fig.3. Note that the results of Bubblesort and Quicksort are omitted because of their low efficiency (see Tab.1 and Fig.2), there are totally 3 algorithms, i.e., classic Heapsort top-*k*, Min-heap based top-*k*, and Partial Quicksort, which are used as the baselines.

From Fig.3 we can see that, as the data size *n* increases, the execution time curve of DC-Top-*k* is very similar to a linear curve. More precisely, the speed of Algorithm 1 is 3 to 4 times as fast as classic Heapsort top-*k* algorithm. This further proves that DC-Top-*k* is efficient and the efficiency is relatively stable. In other words, using Partial Quicksort as the underlying algorithm of DC-Top-*k* has no influence on the stability of efficiency. The main reason is as follows: after *n* elements are divided into groups, the value of *Q* is only correlated with the value of *k* and the number of valid elements is relatively small (see Fig.1), so using Partial Quicksort as the underlying algorithm to deal with top-*k* problem can hardly disturb the stability of the efficiency.

The other two baselines are Partial Quicksort (the state-of-the-art algorithm) [4,8,9] and Min-heap based top-*k* [3,7]. See Section III for the details of them. From Fig.3, the speedup ratio of DC-Top-*k* against Partial Quicksort is about 1 to 3. In most cases (Fig.3(a) and (b)), DC-Top-*k* is faster than Partial Quicksort, which is in accordance with the theoretical analysis in Section III (DC-Top-*k* has less average exchange times than Partial Quicksort, and this resulted in a theoretical speedup ratio of 1.46 for large *n*). In few cases (Fig.3(c)), it is not as fast as Partial Quicksort, and this is due to the factor that the value of n/k is small (e.g., n/k<100). But as the data size *n* increases, this case will not exist.

Moreover, when we repeat the experiment many times, the execution time of Partial Quicksort is greatly different each time. Fig.3 further proves that Partial Quicksort is not stable in efficiency, since its execution time curve is significantly not linear, which is deviate from the theoretical result. Obviously, this is determined by the nature of Quicksort algorithm itself. Although we can use some methods rather than randomly select the pivot to mitigate this problem somewhat, the efficiency maybe greatly reduced. By contrast, DC-Top-*k* can keep high and stable efficiency at the same time. Moreover, we can see that compared with the Min-heap based top-*k*, DC-Top-*k* is slightly better when *k* is very small (e.g., $k<=10^3$, see Fig.3(a) and (b)). As *k* gradually increases, the advantage of DC-Top-*k* has been significantly revealed. For example, when *k* reaches 10^6, the speedup ratio of DC-Top-*k* against Min-heap based top-*k* is almost 3.

D. The impact of the underlying sorting algorithm

It is worth noting that some researches concentrate on the improvement of the worst-case efficiency of Quicksort or Quickselect [16-18]. If we select the pivot by Median Of Medians algorithm [17] rather than by randomized methods, then the worst-case complexity of Quickselect will be $\mathbf{O}(n)$ (about $5.43n$). Accordingly, the worst-case complexity of Partial Quicksort can also be improved to $\mathbf{O}(n)$, however the average performance is not necessarily better.

Here we do not mainly discuss which one the actual state-of-the-art is, but concentrate on the effect of DC-Top-*k* itself. In fact, from Theorem 1 and Experiment *A* we know that, the advantage of DC-Top-*k* does not lie in the underlying algorithm, since only a very small proportion of elements need to be deal with by the underlying algorithm. Here we use various top-*k* algorithms (Bubblesort, Heapsort, Min-heapsort and Partial Quicksort) as the underlying algorithm, aiming to

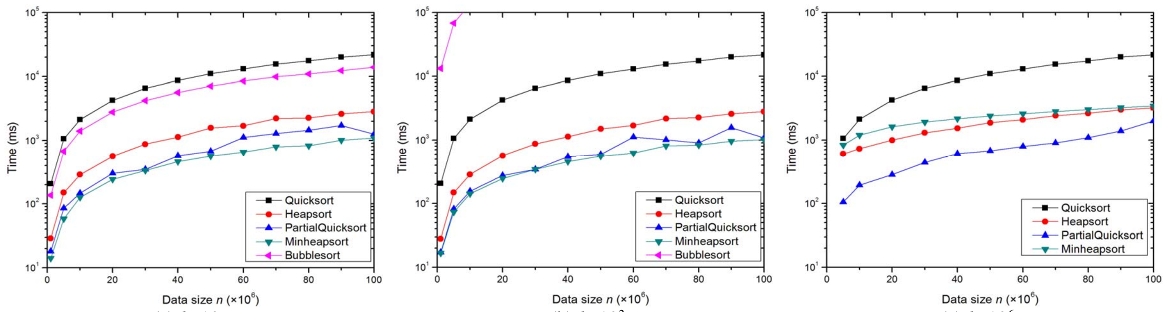
(a) k=10 (b) k=10^3 (c) k=10^6

Fig. 2. The execution time of various top-k algorithms under different data sizes n and k

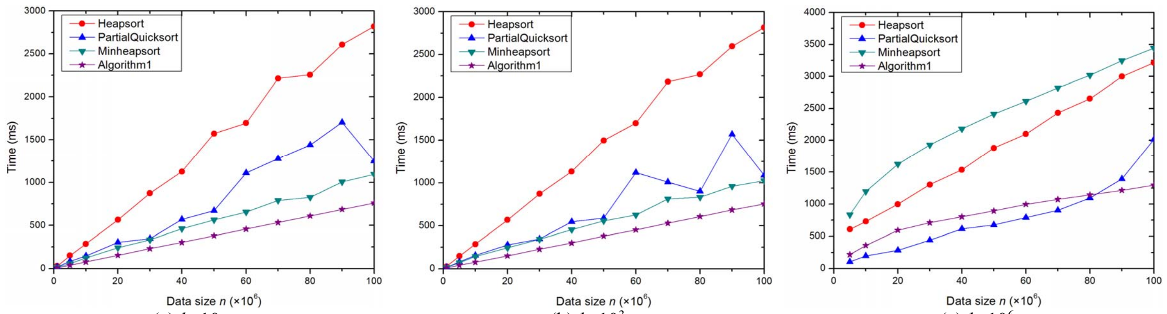
(a) k=10 (b) k=10^3 (c) k=10^6

Fig. 3. DC-Top-k algorithm against various top-k algorithms under different data sizes n and k

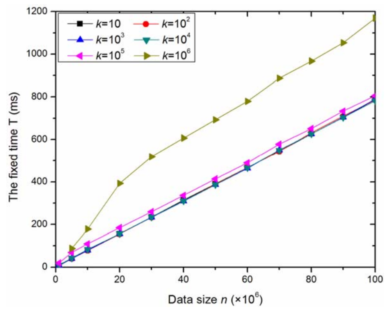
Fig. 4. The fixed time T of Algorithm 1

test the impact to the overall performance of DC-Top-k. DC-Top-k with various underlying algorithms are denoted as Algorithm 1-B, 1-H, 1-M and 1-P, respectively.

Denote T as the time for executing the core parts (lines 1-22) of Algorithm 1 (T is independent of the underlying algorithm), and t the time for the underlying algorithm to get top-k from (k+Q) elements, then the total execution time of DC-Top-k will be t+T. The time T and t+T is shown in Fig.4 and Fig.5, respectively. In Fig.5, only top-10 to top-1,000 are tested for Bubblesort and Algorithm 1-B, because of the low efficiency. Moreover, in Fig.5(b) we omit the result of Bubblesort since the time is too long to be shown.

From Fig.4 we can see that, as the data size n increases, the fixed time T is very similar to a linear curve. This is in accordance with the theoretical analysis in Section III.

From Fig.5 we know that, no matter which algorithm (except for Bubblesort whose complexity is not optimal $\mathbf{O}(n)$ but $\mathbf{O}(kn)$) serve as the underlying algorithm, the overall performance of DC-Top-k shows little difference, as to optimal, $\mathbf{O}(n)$. Obviously, the advantage of DC-Top-k mainly lies in the core parts of the algorithm rather than lies in the underlying algorithm. Moreover, when compare the various original algorithms (i.e., Bubblesort, Heapsort and Min-heap based top-k) with DC-Top-k which adopts them as the underlying algorithm respectively (i.e., Algorithm 1-B, 1-H and 1-M), the latter significantly outperforms the former (see Fig.5(a) and (b)). Note that in Fig.5(c), Algorithm 1-H is not as good as Heapsort algorithm when n<5×10^7. As the increases of n, the superiority of DC-Top-k also gradually appears. This is similar with the theoretical analysis in Section III, which focus on Partial Quicksort as the underlying algorithm.

In a word, no matter whether the underlying algorithm we adopt is the state-of-the-art or not, DC-Top-k can have the similar effect through our divide-and-conquer strategy.

E. *The scalability of DC-Top-k algorithm*

In Section III, we proposed parallel DC-Top-k algorithm (Algorithm 2), now we test its performance to confirm the effectiveness. By contrast, we also follow the idea in [36] to parallelize Partial Quicksort, and test the performance of this recursive Parallel Partial Quicksort (see Section II). The cluster has 20 nodes, for each node running one process by default. We repeat the experiment 20 times and take the average value as the result. The performance of parallel DC-Top-k and parallel Partial Quicksort under different data sizes is shown in Fig.6, where "DC" stands for DC-Top-k and "PQ" stands for Partial Quicksort.

Clearly, from Fig.6 we can see that the performance of Algorithm 2 significantly outperforms Partial Quicksort in most cases. For example, if the data size n=10^9, k=10, and p=20 (p is the number of nodes in the cluster), then DC-Top-k (about 230ms) is about 2.2 times as fast as Partial Quicksort

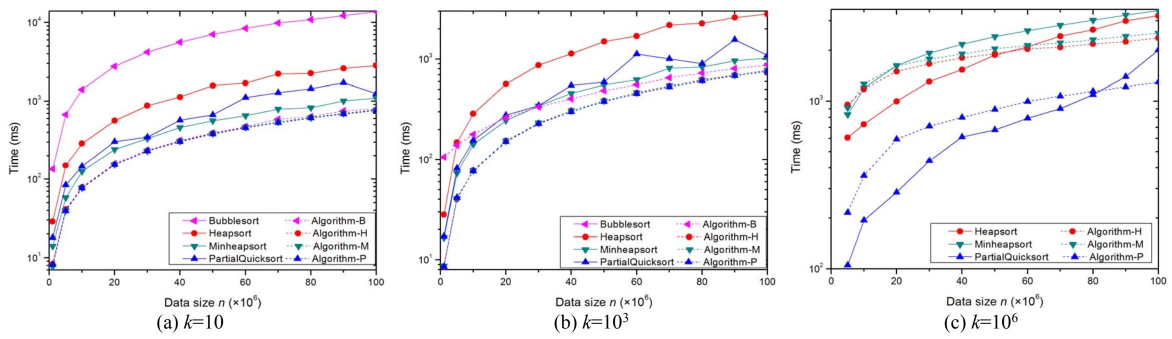

Fig. 5. Various algorithms as the underlying algorithm of Algorithm 1

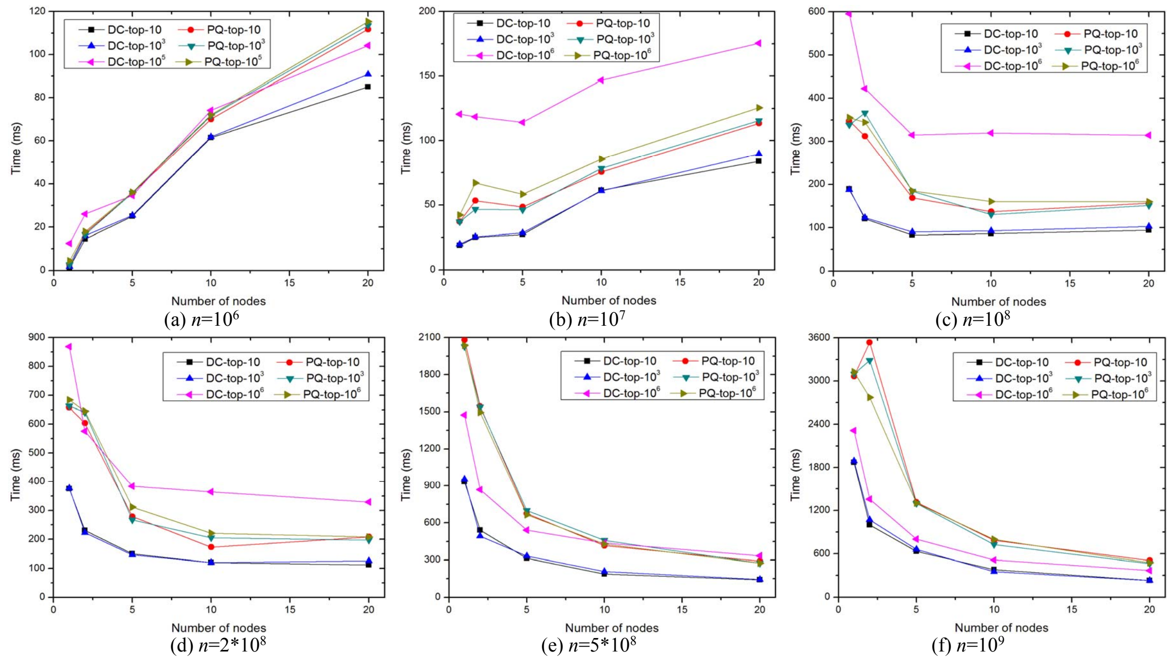

Fig. 6. Parallel DC-Top-k algorithm against parallel Partial Quicksort algorithm

(about 508ms). Moreover, from Fig.6 we can see that when the data size n is relatively small (e.g., n=10^6), because of the communication overhead, the advantage of DC-Top-k cannot be reflected with the increase of the cluster scale. About Partial Quicksort, the situation is similar to this, i.e., when n is small, the efficiency decreases with the increase of cluster scale. Only when the data size n is very small and meanwhile, k is very large, then Partial Quicksort seems to be better than DC-Top-k (e.g., n=10^7 while k=10^6). But in a practical application, k<<n, and it is unnecessary for small data size to compute in the distributed and parallel platform, so we mainly focus on the condition that n is large.

With the increase of data size, the scalability of DC-Top-k becomes obviously visible. For example, when n=10^9 and k=10, the speed of DC-Top-k with 10 nodes is about 5 times as high as that of a single node. However, in the same case, the speed of Partial Quicksort with 10 nodes is less than 4 times as high as that of a single node. Obviously, DC-Top-k has higher scalability than Partial Quicksort and is more suitable for distributed and parallel processing. The reason is that Partial Quicksort involves recursion and numerous communication between each process, while DC-Top-k does not.

It is worth noting that, because of the limit of single computer memory, we haven't tested larger scale data sets (e.g., n=10^{10}). However, from Fig.6, we believe that with the increase of data size n, the speedup ratio will be further improved if the computer memory permits. Moreover, note that though the underlying algorithm in parallel DC-Top-k (Algorithm 2) is serial, we can still get better performance than the parallel version of Partial Quicksort in [36]. If we find a better way to parallelize the underlying algorithm, then DC-Top-k will be improved accordingly.

V. Conclusions And Future Work

Aiming at the top-k problem for large data, we propose DC-Top-k, an efficient top-k algorithm based on divide-and-conquer strategy. The effectiveness of our algorithm is verified by both a theoretical analysis and a number of experiments. The experiments show that DC-Top-k has remarkable advantage compared with the Heapsort top-k, Min-heap based top-k [3,7] and the state-of-the-art algorithm Partial Quicksort [4,8,9]. DC-Top-k can keep high and stable efficiency at the same time. Moreover, it has good scalability and is more suitable for parallel computing.

To improve the efficiency of DC-Top-*k*, one future work is to explore more reasonable grouping strategies; another one is to set the recursion threshold properly to deal with different data sizes. We believe our approach will shed a new light on the further research and application of top-*k* algorithm.

ACKNOWLEDGMENT

This work is supported by National Natural Science Foundation of China under grants 61572221, 61433006, U1401258, 61300222, 61370230 and 61173170, Innovation Fund of Huazhong University of Science and Technology under grants 2015TS069 and 2015TS071, Science and Technology Support Program of Hubei Province under grant 2014BCH270 and 2015AAA013, and Science and Technology Program of Guangdong Province under grant 2014B010111007.

REFERENCES

[1] D. Song, D. A. Meyer, and D. Tao, “Top-k link recommendation in social networks,” in *Proc. of the 15th International Conference on Data Mining* (*ICDM*), IEEE, 2015, pp. 389-398.

[2] G. Navarro, and Y. Nekrich, “Top-k document retrieval in optimal time and linear space,” in *Proc. of the 23rd Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, SIAM, 2012, pp. 1066-1077.

[3] S. Niu, J. Guo, Y. Lan, and X. Cheng, “Top-k learning to rank: labeling, ranking and evaluation,” in *Proc. of the 35th International Conference on Research on Development in Information Retrieval (SIGIR)*, ACM, 2012, pp. 751-760.

[4] Z. Wang, and S. S. Tseng, “Knee point search using cascading top-k sorting with minimized time Complexity,” *Scientific World Journal*, 2013, vol. 2013, no. 4, pp. 405-421.

[5] A. C. Yao, “On selecting the k largest with median tests,” *Algorithmica*, 1989, vol. 4, no. 1-4, pp. 293-300.

[6] D. Knuth, “The Art of Computer Programming, Volume 3: Sorting and Searching,” Third Edition. Addison-Wesley, 1997. ISBN 0-201-89685-0. Section 5.3.3: Minimum-Comparison Selection, pp.207-219.

[7] K. Y. Whang, M. S. Kim, and J. H. Lee, “Linear-time top-k sort method,” U.S. Patent 8,296,306. 2012-10-23.

[8] C. Martınez, “Partial quicksort,” in *Proc. of 6th ACMSIAM Workshop on Algorithm Engineering and Experiments and 1st ACM-SIAM Workshop on Analytic Algorithmics and Combinatorics*, 2004, pp. 224-228.

[9] G. Navarro, and R. Paredes, “On sorting, heaps, and minimum spanning trees,” *Algorithmica*, 2010, vol. 57, no. 4, pp. 585-620.

[10] R. Paredes, and G. Navarro, “Optimal incremental sorting,” in *Proc. of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX)*, 2012, pp. 171-182.

[11] R. Sedgewick, “The analysis of quicksort programs,” *Acta Informatica*, 1977, vol. 7, no. 4, pp. 327-355.

[12] P. Tsigas, and Y. Zhang, “A simple, fast parallel implementation of quicksort and its performance evaluation on SUN enterprise 10000,” in *Proc. of the 11th Euromicro Conference on Parallel, Distributed and Network-Based Processing (PDP)*, IEEE, 2003, pp. 372-381.

[13] D. J. Haglin, R. D. Adolf, and G. E. Mackey, “Scalable, multithreaded, partially-in-place sorting,” in *Proc. of the 27th International conference on Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW)*, IEEE, 2013, pp. 1656-1664.

[14] D. Pasetto, and A. Akhriev, “A comparative study of parallel sort algorithms,” *in Proc. of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, ACM, 2011, pp. 203-204.

[15] D. Bozidar, and T. Dobravec, “Comparison of parallel sorting algorithms,” *arXiv preprint arXiv*:1511.03404, 2015.

[16] C. Martínez, and S. Roura, “Optimal sampling strategies in quicksort and quickselect,” *SIAM Journal on Computing*, 2001, vol. 31, no. 3, pp. 683-705.

[17] M. Blum, R. W. Floyd, V. Pratt, R. L. Rivest, and R. E. Tarjan, “Time bounds for selection,” *Journal of computer and system sciences*, 1973, vol. 7, no. 4, pp. 448-461.

[18] C. Martínez, D. Panario, and A. Viola, “Adaptive sampling strategies for quickselects,” *ACM Transactions on Algorithms*, 2010, vol. 6, no. 3, pp. 1-45.

[19] J. W. J. Williams, “Algorithm 232: Heapsort,” *Communications of the ACM*, 1964, vol. 7, no. 6, pp. 347-348.

[20] C. A. R. Hoare, “Quicksort,” *The Computer Journal*, 1962, vol. 5, no. 1, pp. 10-15

[21] M. T. Goodrich, “Spin-the-bottle sort and annealing sort: Oblivious sorting via round-robin random comparisons,” *Algorithmica*, 2014, vol. 68, no. 4, pp. 835-858.

[22] D. Early, and M. Schellekens, “Running time of the Treapsort algorithm,” *Theoretical Computer Science*, 2013, vol. 487, no. 2, pp. 65-73.

[23] Z. Huang, S. Kannan, and S. Khanna, “Algorithms for the generalized sorting problem,” in *Proc. of the 52nd Annual Symposium on Foundations of Computer Science (FOCS)*, IEEE, 2011, pp. 738-747.

[24] C. A. R. Hoare, “Algorithm 65: find,” *Communications of the ACM*, 1961, vol. 4, no. 7, pp. 321-322.

[25] C. Martínez, and H. Prodinger, “Moves and displacements of particular elements in Quicksort,” *Theoretical Computer Science*, 2009, vol. 410, no. 21, pp. 2279-2284.

[26] L. Devroye, “On the probablistic worst-case time of “Find”,” *Algorithmica*, 2001, vol. 31, no. 3, pp. 291-303.

[27] J. A. Fill, and T. Nakama, “Analysis of the expected number of bit comparisons required by Quickselect,” *Algorithmica*, 2010, vol. 58, no. 3, pp. 730-769.

[28] W. Cunto, and J. I. Munro. “Average case selection,” *Journal of the ACM (JACM)*, 1989, vol. 36, no. 2, pp. 270-279.

[29] M. Fouz, M. Kufleitner, B. Manthey, and N. Z. Jahromi, “On smoothed analysis of quicksort and Hoare’s find,” *Algorithmica*, 2012, vol. 62, no. 3-4, pp. 879-905.

[30] M. Kuba, “On Quickselect, partial sorting and multiple Quickselect,” *Information processing letters*, 2006, vol. 99, no. 5, pp. 181-186.

[31] C. Martínez, and U. Rösler, “Partial quicksort and Quickpartition sort,” in *Proc. of the 21st International Meeting on Probabilistic, Combinatorial, and Asymptotic Methods in the Analysis of Algorithms (AofA)*, 2010, pp. 505-512.

[32] P. Afshani, G. S. Brodal, and N. Zeh, “Ordered and unordered top-k range reporting in large data sets,” in *Proc. of the 22nd annual ACM-SIAM symposium on Discrete Algorithms (SODA)*, SIAM, 2011, pp. 390-400.

[33] Q. Huang, X. Liu, X. Sun, and J. Zhang, “How to select the top k elements from evolving data?,” in *Proc. of the 26th International Symposium on Algorithms and Computation (ISAAC)*, Springer, 2015, pp. 60-70.

[34] R. Fagin, A. Lotem, and M. Naor, “Optimal aggregation algorithms for middleware,” in *Proc. of the 20th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS)*, ACM, 2001, pp. 102- 113.

[35] P. Sanders, “Randomized priority queues for fast parallel access,” *Journal of Parallel and Distributed Computing*, 1998, vol. 49, no. 1, pp. 86-97.

[36] M. Axtmann, T. Bingmann, P. Sanders, and C. Schulz, “Practical massively parallel sorting,” *in Proc. of the 27th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA)*, ACM, 2015, pp. 13-23.

[37] L. Hübschle-Schneider, P. Sanders, and I. Müller, “Communication efficient algorithms for top-k selection problems,” *arXiv preprint arXiv*:1502.03942, 2015.

[38] R. W. Floyd, and R. L. Rivest, “Expected time bounds for selection,” *Communications of the ACM*, 1975, vol. 18, no. 3, pp. 165-172.