



# Distributed Selection

Fabian Kuhn

Thomas Locher

Roger Wattenhofer

By Michael Zabejansky

# [ Agenda ]

- Introduction
- Motivation
- The Random Algorithm
- The Deterministic Algorithm
  - Simple idea
  - Improved Idea

# [ Aggregation functions ]

- Distributive – max, min, sum, count
- Algebraic – average, variance
- Holistic – median, k-th smallest value

Combination of these functions support a wide range of queries.

# Distributed aggregation

The goal:

To compute aggregation function between distributed set of values.

- Data mining

- Different data queries in data network
- Reduce unnecessary data from different data sources (to save bandwidth and power)
- Statistic computations

# Distributed aggregation

- Sensor networks
  - Industrial process monitoring and control
    - Measuring maximal temperature of different parts of a machine
  - Healthcare applications
    - Body sensors
  - Environment monitoring
    - Volcanoes, Earthquakes
    - Agriculture

# Solutions

- Distributive, Algebraic - by convergecast  $O(D)$
- Holistic ,not possible by convergecast (when we bounded by the message size) because all values needed to be centralized in one node

# [ The problem ]

- Selection of K-th smallest elements among distributed set of values
  - Randomized algorithm –  $O(D \log_D n)$
  - Deterministic algorithm –  $O(D(\log_D n)^2)$

# [ Model and Definitions ]

- $G = (V, E)$
- $|V| = n$
- $v_1 \dots v_n$  Holds unique values  $x_1 \dots x_n$
- the goal – find  $k$  smallest elements among  $x_1 \dots x_n$  by exchanging messages
- Nodes can directly communicate if  $(v_i, v_j) \in E$

# [ Model and Definitions ]

- Asynchronous communication
- Reliable communication
- Message can contain constant number of ids and elements, and  $O(\log n)$  additional bits
  - Otherwise the problem can be solved by convergecast

# [ Model and Definitions ]

- All the nodes know the diameter of the graph
- BFS rooted in the node initiating the algorithm
- We refer to element of interest as *candidate nodes*. Number of candidate nodes in phase  $i$  is  $n^{(i)}$

# [ Random Algorithm ]

- Select  $t$  random elements in specific range  $((-\infty, \infty)$  in the beginning)
- Divide the range into intervals
- Count the number of candidate nodes in each interval
- Choose the interval for which sum of the candidate nodes in the intervals, in the first time larger than  $k$
- Repeat those steps for the chosen interval until we cover exactly  $k$  items

# Choosing one random element

- Root has  $l$  children  $v_1, \dots, v_l$  where child  $v_i$  is the root of a subtree with  $n_i$  candidate nodes including itself
- The root chooses its own element with probability  $1 / (1 + \sum_{j=1}^l n_j)$
- Otherwise, it sends a message to one of its children (if a leaf is reached, then it selected)
- The message is forwarded to node  $v_i$  with probability  $n_i / (1 + \sum_{j=1}^l n_j)$
- That scheme selects exactly one random node uniformly in maximum  $2D$  time (reach a node and report back bounded by  $D$ )

# Choosing several random elements

- We can instead select several random elements by including the number of needed elements in the request message.
- All random elements can be found in  $O(D)$  time independent of the number of elements
- The number of the elements is bounded by the message size

# [ Random Algorithm ]

- Parameters:
  - t- the number of random elements considered in each phase
  - k -the position of interest
- getRndElementsInRange collects t random elements in the range  $[x_{j-1}, x_j]$

# [ Random Algorithm ]

- When we have the random elements  $[x_1, \dots, x_t]$ ,  $(x_1 < \dots < x_t)$  we count the elements in each interval  $[x_{i-1}, x_i]$ 
  - $r_i = \text{countElementsInRange}([x_{i-1}, x_i])$
- We can do the counting in each part in parallel so it takes  $O(D+t)$  for all the operation
  - (the root is the bottleneck when receiving  $t$  items)

# [ Random Algorithm ]

- Set  $j$  to the minimal index such that  $\sum_{i=1..n} r_i > k$
- $k$  updated accordingly  
 $k = k - \sum_{i=1..j-1} r_i$
- The next iteration in the interval  $[x_{j-1}, x_j]$

# [ Time complexity ]

- In each phase we call `getRndElementsInRange` which selects  $t$  elements in specified interval
  - Counting the relevant nodes in the subtree
  - Once we have the number of candidate nodes in each subtree, the  $t$  random elements are selected and reported back to the root.
- Hence, this function call takes  $O(D+t)$  time.

# [ Time complexity ]

- The steps are repeated until  $k=0$  or the fraction is small enough such that all elements can be collected in  $O(D+t)$  time and the solution can be computed by convergecast.
- We can prove that the number of phases is  $O(\log_D n)$  with high probability, hence the complexity of the algorithm is  $O(D \log_D n)$
- This algorithm is faster than selecting only one random element in each phase
- Its provable that the algorithm is asymptotically optimal

# [ Deterministic algorithm ]

## The problem

- The upper bound of a random algorithm, we want deterministic solution
- Selection of elements that partition the entire space into approximately equal groups

# [ Deterministic algorithm ]

- Reduction of the range in each iteration – by choosing smaller range
- The root uses the elements returned to it to narrow down the range of potential candidates.
- Use those elements as in the randomized algorithm

# Select elements in range

## Simple idea

- Start sending elements from the leaves and then recursively forwarding a selection of  $t$  elements to the parents
- The problem is the reduction of all elements received from the children to  $t$  elements only.
- Suppose node  $v_i$  receives  $t$  elements from each of its  $c_i$  children.
- Needed to select  $t$  elements from  $c_i t$  nodes that partition all  $c_i t$  nodes into segments of approximately equal size

# Select elements in range

## Simple idea

- To find these  $t$  elements, the number of elements in each segment has to be counted
- The counting has to be repeated in each step along the path to the root
- Requires  $O(D(D + Ct))$  time, where
$$C := \max_{i \text{ in } \{1, \dots, n\}} C_i$$
- At least  $O(D^2)$  time just to find a partitioning, and the time complexity depends on the structure of the spanning tree.
- Not that good.

# [ Improved Idea ]

- We want to reduce the times we count
- In each phase  $i$  we partition the  $n^{(i)}$  items into  $O(\sqrt{D})$  groups of size  $O(n^{(i)}/\sqrt{D})$ , we do so recursively.
- Each group will report  $O(\sqrt{D})$  elements to its parent
- Each node receives  $O(D)$  elements from the  $O(\sqrt{D})$  groups it created
- It then reduces the elements to  $O(\sqrt{D})$  elements as in the simple idea

# Creating the groups

- Each node  $v$  partitions its subtrees into disjoint groups
- Each group will be of size  $\leq n_v^{(i)}/\sqrt{D}$ 
  - Each groups will be of size  $\geq [n_v^{(i)}/\sqrt{D}]/2$  otherwise at least two groups could be merged
- Number of groups will be bounded by  $2\sqrt{D}$

# [ Creating the groups ]

- Each group returns a leader which is one of the children of the root node
- Returns set of “leaders” from each of the groups.
- Each leader is responsible for further partitioning the groups

# [ Dividing to partitions ]

- `getPartitionInRange` at a particular (leader) node  $v$ , returns a subset of partitioning elements in the subgroup whose leader is  $v$ .
- `getPartitionInRange` is called recursively at each leader in order to further partition the groups.

# [ getPartitionInRange(g,range) ]

- If its group consists of less than the number of elements in a group -  $g = 2\sqrt{D}$ , all elements are returned to the node that issued the request for a further partitioning.
- In case the group is larger, the resulting sets of the recursive calls are accumulated and sorted.
- The number of elements in each interval induced by the sorted items are counted in parallel .
- Then the intervals are reduced

# [ getPartitionInRange ]

- The function *reduce* merges adjacent intervals as long as each interval contains at most  $n_v^{(i)} / \sqrt{D}$  elements
- The reduced set of elements inducing these new intervals is returned.

# [ Time complexity ]

- Lemma: At any node  $v$  in phase  $i$ , `getPartitionInRange` returns a set of at most  $2\sqrt{D} + 1$  elements which make intervals containing at most  $n^{(i)}/\sqrt{D}$  elements each.
  - Proof by induction
- From the lemma we get  $n^{(i+1)} \leq n^{(i)}/\sqrt{D}$  thus the number of phases is bounded by  $O(\log_D n)$
- We saw that the groups can be created in  $O(D)$  time

# [ Time complexity ]

- Now, we left with the collecting and counting the elements in each interval in each group
  - The number of groups is bounded by  $2\sqrt{D}$
  - Each group returns at most  $2\sqrt{D} + 1$  elements
  - At most  $4D + 2\sqrt{D}$  elements have to be collected
  - Collecting and counting takes  $O(D)$

# [ Time complexity ]

- Denote by  $T(n)$  the time complexity of *getPartitionInRange* if there are  $n$  candidate nodes
- We have that  $T(n) \leq T(n/\sqrt{D}) + cD$  for a suitable constant  $c$ , implying that  $T(n)$  is  $O(D \log_D n)$
- We have  $O(\log_D n)$  phases, so the time complexity of the deterministic algorithm is bounded by  $O(D(\log_D n)^2)$

# [ Summary ]

- Select k smallest elements between distributed set of values
- Randomized alg
  - $O(D \log_D n)$
  - Bad upper bound, not deterministic
- Deterministic alg
  - More intelligent partitioning
  - $O(D(\log_D n)^2)$

[ Questions? ]

---

# [ Improved Idea - reduction ]

- Each new interval will contain  $O(n_v^{(i)} / \sqrt{D})$  nodes
- At the root all  $O(\sqrt{D})$  intervals will contain at most  $O(n^{(i)} / \sqrt{D})$  nodes
- Thus  $n^{(i+1)} < n^{(i)} / O(\sqrt{D})$

[ **getPartitionInRange( $g, (x_{j-1}, x_j)$ )**  
returns  $g$  elements partitioning the range  $(x_{j-1}, x_j)$  ]

---

**Algorithm 2** `getPartitionInRange( $g, (x_{j-1}, x_j)$ )`

---

```
1:  $n' := \text{countElementsInRange}((x_{j-1}, x_j))$ 
2: if  $n' > g$  then
3:    $\{v_1, \dots, v_l\} := \text{createGroups}(g, (x_{j-1}, x_j))$ 
4:   for  $i = 1, \dots, l$  in parallel do
5:      $\{x_{i1}, \dots, x_{im}\} := \text{getPartitionInRange}(g, (x_{j-1}, x_j))$ 
       from  $v_i$ 
6:   od
7:    $\mathcal{X} := \bigcup_{i=1, \dots, l} \{x_{i1}, \dots, x_{im}\}$ 
8:    $\{x_1, \dots, x_s\} := \text{sort}(\mathcal{X})$ 
9:   for  $i = 1, \dots, s - 1$  in parallel do
10:     $r_i := \text{countElementsInRange}((x_i, x_{i+1}))$ 
11:   od
12:    $\{x'_1, \dots, x'_m\} := \text{reduce}(\{x_1, \dots, x_s\}, \{r_1, \dots, r_{s-1}\})$ 
13: else
14:    $\{x'_1, \dots, x'_m\} := \text{getElementsInRange}((x_{j-1}, x_j))$ 
15: fi
16: return  $\{x'_1, \dots, x'_m\}$ 
```

---