; revised ; date of current version 27 November, 2024.

Fast Distributed Selection with Graphics Processing Units

JEFFREY D. BLANCHARD*, RUIZHE FU†, AND TRISTAN KNOTH‡

[1]Mathematics Department, Grinnell College, Grinnell, IA 50112 USA
[2]Computer Science Department, Grinnell College, Grinnell, IA 50112 USA and
Fu Foundation School of Engineering and Applied Science, Columbia University, New York, NY 10027 USA
[3]JitX, Santa Cruz, CA 95060 USA

CORRESPONDING AUTHOR: Jeffrey D. Blanchard (email: jeff@math.grinnell.edu).

ABSTRACT We introduce two completely parallel algorithms for exactly solving the distributed multiple order statistic problem. The main algorithm, DIBMS, keeps communication costs proportional to the number of order statistics with minimal communication of actual data. Moreover, the work done at the remote nodes is fully parallelized on graphics processing units.

INDEX TERMS Selection, Distributed Selection, Multiselection, Order Statistics, Parallel Selection, Graphics Processing Units, GPGPU, Distributed Computing.

I. INTRODUCTION

The selection problem is a data aggregation task that is ubiquitous and well studied. Suppose you have a numerical data set of ten million values and wish to know the value which establishes the top 10%; you want to select the nine millionth largest value in your data set. This is *selection*, or k-*selection* with $k = 9,000,000$, and the kth value is also known as the kth *order statistic* [?], [?], [?], [?].

Common selection tasks include finding the minimum, median, and maximum, where finding the median feels quite different from the other two tasks which can be accomplished with a single pass through the data. The median has the special property that half of the data set will be larger and half smaller, so evaluating a candidate for the median requires us to count exactly how many elements of our data set are larger than the candidate and how many are smaller. The same is true for finding the kth largest value for some other k, like the tenth percentile rather than the fiftieth percentile. This requires us to know something about every value in the data set. Such an aggregation problem is more complicated than, say, taking a sum or computing an average. The selection problem is a non-decomposable aggregation problem [?].

Now suppose further that you want even more: rather than just one order statistic like the tenth percentile, you want to know all the percentiles. A simple linear interpolation of the percentiles provides a highly accurate approximation of the density function for your data allowing you to rapidly query this function rather than the data. This more complicated task of selecting multiple order statistics is known as *multiselection* [?], [?], [?].

When data sets are small enough, sorting is the best way to accomplish multiselection. Modern data sets are, however, enormous. In edition to being very large, modern data sets are often stored at multiple locations. The *distributed selection* problem has also been studied for nearly 50 years with an emphasis on minimizing communication [?], [?], [?], [?], [?], [?], [?]. The sizes of modern data sets at the distributed locations now demand distributed multiselection algorithms that both minimize communication and complete their tasks very quickly at the nodes.

In this paper we consider the multiselection problem for data distributed on a system of computers equipped with graphics processing units. The algorithm is designed to keep required communication costs proportional to the number of requested order statistics. Moreover, at the remote locations, the algorithm's tasks are completed in mere milliseconds. The combination of limited communication and rapid GPU execution at the nodes are the hallmarks our algorithm, DISTRIBUTEDITERATIVEBUCKETMULTISELECT or DIBMS. DIBMS can select the percentile order statistics from a data set of *268 million values distributed at four locations in roughly one tenth of a second* (see Tab. **??**).

A. Selection and Order Statistics

To understand the strengths and weaknesses of DIBMS, we build to the distributed multiselection problem. We begin in this subsection by clearly defining the problem of exactly finding a single order statistic, known as k *selection* or simply *selection*. In the next subsection we will establish the language for the remainder of the paper. The final two subsections extend the selection problem to the task of selecting multiple order statistics simultaneously and then doing so when the data is distributed to multiple locations.

The general scheme for k selection is to iteratively separate the data into subcollections [**?**], one made up of candidates for the kth order statistic and the other of elements eliminated from contention. Suppose our data S contains n numeric values, possibly with repeated values, and suppose we need to find the kth largest value in S. Given a pivot value p, we partition S into two collections: one containing those values less than or equal to p, and the other the values greater than p:

$$L = \{x \in S : x \leq p\},\ G = \{x \in S : x > p\}.$$

Now, we count the size of the collections $(|L|, |G|)$ and determine which subcollection contains the kth largest value. If L has more than k elements, then we perform this task again on L with a new pivot value. Otherwise, G contains our kth largest value and we iterate on G while updating k to the value $k - |L|$.

In each iteration, our goal is to eliminate from contention as many elements from the data as possible. This makes the selection of the pivots at each iteration very important and the focus of much study. Hoare's FIND [**?**], now referred to as QUICKSELECT, chooses the pivot randomly from S. The median-of-medians idea from Blum et al. [**?**] separates the S into smaller collections of data, computes medians on these smaller data sets, and then chooses the pivot, p, as the median among these subcollection medians. When resources are available, defining many more subcollections can accelerate selection [**?**], [**?**].

B. Language and Notation

The variety of fields and applications for which this problem is important has created a wide range of language for the problem. We shall adopt a specific, descriptive language for the rest of this paper outlining the key ideas in selection. The data set S will always contain n elements, and, like most real world data, is allowed to contain duplicates. We call the kth largest value the kth order statistic and any set of m desired values the order statistics. The general scheme described above will be referred to via four subroutines. The subcollections of S in which we partition the data will be called *buckets*, and our task will be to create buckets, assign every element in S to a bucket, determine which bucket contains the order statistic(s) we seek, and reduce the problem to only the important bucket(s).

Create Buckets
Determine the pivots which define the buckets.

Assign Buckets
For each element in our current data, determine which bucket it should be assigned based on our definitions.

Identify Active Buckets
By counting the number of elements assigned to each bucket (including counting duplicates), determine which bucket (buckets) contains the order statistic (order statistics) we seek.

Reduce
Eliminate all elements not assigned to an active bucket, update any order statistics based on this new, reduced data, and proceed with only those elements in the active buckets and the updated order statistics.

For example, in QUICKSELECT with an equality bucket, we randomly select p and Create Buckets L, E, G:

$$\begin{aligned} L &= \{x \in S : x < p\}, \\ E &= \{x \in S : x = p\}, \\ G &= \{x \in S : x > p\}. \end{aligned}$$

We then Assign Buckets by comparing each $x \in S$ to bucket definitions; if S contains duplicates, some buckets must contain duplicates. We Identify Active Buckets by computing the counts $|L|, |E|, |G|$. If $k < |L|$, L is the only active bucket. If $|L| < k \leq |L| + |E|$, E is the active bucket. Otherwise $k > |L| + |E|$ and G is active. We Reduce by keeping only the active bucket. If L is active, we do not need to alter the order statistic k since we are still looking for the kth order statistic on this smaller set L. If E is active, we are in a special situation and can terminate our search as the kth order statistics must be p. In the case that G is active, we are now looking for the $k - (|L| + |E|)$ largest value among only the elements in G.

After Reduce, we have a new, smaller problem, and we can solve that problem any way we would like. For example, we could switch algorithms to solve the reduced problem, or, as in the case of QUICKSELECT, we could simply iterate on the reduced problem.

C. Selecting Multiple Order Statistics

The main algorithmic consideration for multiselection is designing the Buckets so that the Reduce subroutine makes a considerable impact on the proportion of our data set S that remains in contention to be order statistics. When we seek m order statistics, and we Assign Buckets to B buckets, we should end up with at most m Active Buckets. This implies that we will eliminate the elements of the $B - m$ buckets that do not contain one of the targeted order statistics. We want each of these $B - m$ buckets of eliminated elements to contain a substantial portion of the elements from the data. The simplest idea is to attempt to design the buckets so that each bucket will contain approximately n/B elements. Seeking m order statistics from the data S, we will Identify at most m Active Buckets and can reduce the problem from the n elements of S to an expected total of $\frac{m}{B}n$ elements in our active buckets.

In our continuing example of wanting the $m = 101$ order statistics defining the percentiles in a data set of 10 million values, creating $B = 2^{13}$ buckets allows our first iteration to reduce the problem from ten million active elements to roughly 125,000 active elements. A second, equally successful iteration should further reduce to less than two thousand candidates. In fact, assuming even partitioning into buckets, the process will take $\left\lceil \log_{\frac{B}{m}}(n) \right\rceil$ iterations. In our example with $m = 101$, $B = 8192$, and $n = 10^7$, the expected number of iterations is 4.

D. Distributed Selection

The main problem under consideration is one with data stored at multiple locations. If communication costs and data security considerations were irrelevant, we could simply gather the data together at a single location and select order statistics. For most real applications, this is either unwise or expensive, and we want to complete the order statistic selection task without consolidating the data; in fact we want to move as little data as possible.

The distributed selection problem has a long history with early work studying the communication complexity of this problem, e.g. [?], [?], [?]. Frederickson went on to analyze this problem for various network topologies [?]. Santoro and Suen [?] described reduction techniques in the form of four phases to which our four subroutines roughly align. The overwhelming majority of algorithms for selection, and in particular for distributed selection, follow this general outline. Santoro, Sidney, and Sidney [?] produced an algorithm using this framework with a very low expected communication cost.

The evolution of data and hardware keep this problem relevant. Kuhn and collaborators revisited the distributed selection problem in [?] by getting more pivots to create more buckets. In our algorithm, we cap the number of buckets only by the minimum shared memory capacity of the local GPUs, allowing it to scale with hardware advances. Differing from the early research model of ignoring local computation, our algorithm DIBMS exploits the massive parallelization of GPUs to accelerate the local computation on large data sets.

E. Multiselection with GPUs

In the development of GPU algorithms, massively parallel sorting routines were quickly developed [?], [?], [?], [?], [?]. By sorting our collection S we learn every order statistic, since an element's rank corresponds to its position in the sorted list. We hereafter refer to this algorithm as SORT&CHOOSE. Even with the incredible performance of GPU sorting, SORT&CHOOSE can be highly wasteful when data sets are large enough.

In 2011, four initial k-selection algorithms were developed for GPUs: a probabilistic randomized pivot selection RANDOMIZEDSELECT [?], an optimization based partitioning CUTTINGPLANE [?], a radix selection algorithm RADIXSELECT [?], and a computational projection based partitioning BUCKETSELECT [?]. For data with more than roughly 100,000 elements, these were all shown to be clearly superior to GPU sorting and highly competitive with each other. More recently, a new uniform sampling strategy was used in SAMPLESELECT [?], which exploited advances in GPU architecture between 2011 and 2020.

S	distributed data set
n	size (total number of elements) of data set
m	number of order statistics
q	number of nodes (including host node 0)
n_j	size of data on node $j \in \{0, \dots, q-1\}$
B	total number of buckets used in algorithms
P	number of pivot intervals
b_i	number of buckets given to pivot interval $i \in \{0, \dots, P-1\}$

TABLE 1. Reference list of variables.

When we want m distinct order statistics that are not contiguous, it is best to employ specialized multiselection algorithms. While most selection algorithms have extensions to the multiselection problem, we focus[1] on a specific GPU multiselection extension of BUCKETSELECT which employs our familiar four subroutines of Create Buckets, Assign Buckets, Identify Active Buckets, and Reduce.

This algorithm, BUCKETMULTISELECT [?], has three substantial modifications. To facilitate dimension reduction in Assign Buckets, BUCKETMULTISELECT uses a sampling strategy to form a uniform kernel density estimator for defining the buckets; this density estimator helps balance the number of elements assigned to each bucket. Shared memory is exploited to accelerate performance, particularly to avoid atomic conflicts when counting and accessing data. Finally, after the Reduce phase, the newly reduced problem is solved by sorting. In the next section, we introduce a distributed version of BUCKETMULTISELECT. To avoid the communication requirements of sorting, we then extend the algorithm to an iterative version whose communication costs are proportional to the number of order statistics.

II. DISTRIBUTED MULTISELECTION

In the distributed setting, naívely executing SORT&CHOOSE by moving all the data to a centralized location and sorting requires a prohibitive level of communication. While it is surely possible to move all the data to a central location, this paper and the algorithms to follow clearly demonstrate that there is no advantage to consolidating the data. Our algorithm, DIBMS will allow us to find sets of order

[1]The probabilistically chosen pivots in RANDOMIZEDSELECT and the dynamic recursion in SAMPLESELECT pose complications for the distributed multiselection problem.

statistics while leaving nearly all of the data on the remote nodes.

In the following we introduce our algorithm in two parts. First, we describe a first phase of using the principles of BUCKETMULTISELECT to significantly reduce the size of data in contention to be the order statistics. We then must decide how to solve this reduced, yet still distributed problem. One solution, described in Sec. A, will be to consolidate to the host for sorting only the data still in contention; we call this algorithm DISTRIBUTEDBUCKETMULTISELECT since it is performs the same sequence of tasks as BUCKETMULTISELECT, albeit on distributed data. Section B describes our second proposed solution, DISTRIBUTEDITERATIVEBUCKETMULTISELECT, which iterates on only the data still in contention and avoids consolidation of data.

Performing some of the work in parallel on the distributed data prior to transmitting will be helpful. For illustrative and comparison purposes, we introduce a more sophisticated version of a distributed SORT&CHOOSE. With data distributed on several nodes, we first sort the local data on each node. Then, following a tree structure, merge the sorted lists between pairs of nodes climbing the tree until a fully sorted list is formed on the host. From this list, all order statistics are known, but the communication requirements for moving all of the data is severe (as shown in Section IV).

Throughout this section we consider systems of q machines, called *nodes*, equipped with both a CPU and GPU. One node will be identified as the *host* and coordinate all the tasks of the other $q-1$ machines. In our model, the host will also have its own share of data and also perform the tasks being performed at the remote nodes. We label the nodes with the indices $0, 1, \dots, q-1$ with the host always being node 0.

We assume the distributed system has a network topology that enables the host to communicate with each of the nodes. For simplicity, assume throughout that we are working with a mesh topology or star topology, though the algorithms can be adapted to other network topologies. Furthermore, the GPU kernels are launched with the number of blocks, `numBlocks`, equal to the number of multiprocessors on the GPU. This enables the algorithm to launch subsequent kernels to work an identical portion of the local data.

We study exactly finding a set of m order statistics from a distributed data set S with a total size of n elements. These n elements are distributed on q nodes each with n_j elements of the data set so that $\sum_{n=0}^{q-1} n_j = n$, where node 0 is the host machine controlling the algorithm. Our distributed algorithms based on BUCKETMULTISELECT use a computational process to assign elements of S to B buckets. The B buckets are determined from P pivot intervals. Each pivot interval is assigned a portion of the total buckets so that interval i has b_i buckets and $\sum_{i=0}^{P-1} b_i = B$.

A. Distributed Bucket Multiselect

A distributed extension of BUCKETMULTISELECT is to run the parallelized portions of the first phase of BUCKETMULTISELECT at each node, communicate the status of the local bucket assignments from the nodes to the host, identify the active buckets, and ask the nodes to send only the elements of the active buckets back at the host. With an expected reduction from n elements in S to $\frac{m}{B} \cdot n$ elements in the active buckets, we will have a substantial reduction in communication cost.

The aggregation of even the reduced set of data is still undesirable when the data sets are large. Even so, we first present DISTRIBUTEDBUCKETMULTISELECT for two reasons. First, in some situations, for example when the number of order statistics is small and the security of transmitting a portion of the data is ensured, DISTRIBUTEDBUCKETMULTISELECT could be useful and relatively fast. Second, the algorithm consists of a first phase where the distributed data is reduced followed by a second phase where the reduced data is consolidated and sorted. The first phase will be useful again later.

We describe this algorithm in the following subsections, each associated to the four subroutines of Create Buckets, Assign Buckets, Identify Active Buckets, and Reduce. For each subroutine, we will track the expected number of communications where a communication is defined to be sending a unit of information between a node and the host.

1) Phase 1: Create Buckets

In this first phase, we sample the data set to create reliable kernel density estimator (KDE) intervals. Since sampling theory provides reasonable confidence with 1024 samples, we ask each of our q nodes to send its share of $\left\lceil 1024 \cdot \frac{n_i}{n} \right\rceil$ samples to the host. Furthermore, the KDE in our setting needs the minimum and maximum values of S, so each node also transmits the local minimum and maximum to the host. Sampling the full data set to produce our initial KDE requires

$$2 \cdot (q-1) + 1024 \cdot \left(1 - \frac{n_0}{n}\right) \tag{1}$$

communications to send the locals samples and local extreme values to node 0 acting as host.

The remaining work to create the buckets is completed on the CPU on the host. The global minimum and maximum values are easy to find from the respective lists of local extreme values from the nodes. We then find $P-1$ order statistics from the consolidated random sample following the order statistic selection as in [**?**, Section 2.1]. Our initialization determines increasing pivots defining 16 pivot intervals, with $P = 16$ chosen from a combination of empirical investigation and sampling theory. The uniform sampling of 1024 elements gives roughly 95% confidence that each interval will contain roughly $n/16$ elements of S [**?**], [**?**, Section 2.5.3]. Therefore, we allot an equal number of buckets to each of the pivot intervals which define a kernel density estimator.

Let the vector of length 17 named `pivots` represent our ordered set of sampled approximate order statistics together with the min and max in the first and last position. We can define the following control parameters to completely create our B buckets.

$$\begin{aligned} &\text{for } i = 0, \dots, 15 \\ \texttt{leftPivots}[i] &= \texttt{pivots}[i] \\ \texttt{rightPivots}[i] &= \texttt{pivots}[i+1] \\ b_i &= B/16 \\ \texttt{slopes}[i] &= \frac{b_i}{(\texttt{rightPivots}[i] - \texttt{leftPivots}[i])} \end{aligned} \tag{2}$$

At this point, the host must send the three vectors defining the buckets, namely `leftPivots`, `rightPivots`, and `slopes` to each node for

$$3 \cdot P \cdot (q-1) = 48 \cdot (q-1) \tag{3}$$

communications. Since $\left(1 - \frac{n_i}{n}\right) < 1$ and $P = 16$ in this phase, (1) and (3) show that q nodes have the bucket definitions after at most

$$c_{1,1} = 50(q-1) + 1024 \tag{4}$$

communications.

2) Phase 1: Assign Buckets

The data set S is distributed among the q nodes, but each node now has the definitions of the B buckets. It is time to assign every element in S a bucket and count how many elements of S are assigned to each bucket. Fortunately, this can be done in a fully distributed, fully parallelized fashion.

Fix any $r \in \{0, \dots, q-1\}$ so that we can consider the work on the n_r elements of the data S which live on node r. Let's call this local portion of the data vec. Now, also fix $\ell \in \{0, \dots, n_r - 1\}$ and consider the local element $v = \text{vec}[\ell]$. Using a binary search of the sorted list `leftPivots`, we determine that v is at least as large as $\texttt{leftPivots}[j]$ and smaller than $\texttt{leftPivots}[j+1]$ so that v must fall in the pivot interval $[\texttt{leftPivots}[j], \texttt{rightPivots}[j]]$. We know for certain that v will not be assigned to any of the buckets allotted to the previous pivot intervals so we compute

$$\texttt{preCount}[j] = \sum_{i=0}^{j-1} b_i.$$

In this first phase using the approximate KDE, we allotted $B/16$ buckets to each pivot interval, so that

$$\texttt{preCount}[j] = \frac{B}{16}(j-1).$$

Now, we find which of the b_j buckets allotted to this pivot interval will hold our current element v. We compute this via a linear projection and add the previous buckets:

$$\begin{aligned} \texttt{buckets}[\ell] = \lfloor \texttt{slopes}[j] \cdot (\texttt{vec}[\ell] - \texttt{leftPivots}[j]) \rfloor \\ + \texttt{preCount}[j]. \end{aligned} \tag{5}$$

Note that nothing in this process required the use of any other element in the data; this is a completely independent operation for the individual data element v. Thus, once the bucket definition via `leftPivots`, `rightPivots`, and `slopes` have been received on the nodes and stored in shared memory, the assignment of the buckets is completely parallelized.

To track how many data elements have been assigned to each bucket, we simply increment a counter for each bucket. To maintain as much independence as possible, we provide a separate counter for each multiprocessor on each GPU on each node. In doing so, we minimize the likelihood that we need to simultaneously increment the same counter which would cause an atomic operation conflict and queuing. The local counts from each of the multiprocessors are then provided to the node's CPU to form a $B \times$ `numBlocks` array, `CounterArray`, since we launch the GPU with one block per multiprocessor. This array now defines the bucket counts for each portion of the local data that we assigned to buckets.

To prepare to send these counts to the host, we prepare the total bucket counts for the node by transforming `CounterArray` from individual counts to a $B \times$ `numBlocks` cumulative count for each bucket, analogous to [?]. This array also makes it possible for the future Reduce subroutine to known exactly how many elements of each block are assigned to a specific bucket while also knowing the number of elements assigned to that bucket from all lower indexed blocks. This will help avoid any writing conflicts between multiprocessors/blocks.

When all local elements have been assigned to buckets and the local bucket counts completed, the local bucket counts are then sent from each of the $q-1$ nodes back to the host. Since there are B buckets, this requires

$$c_{1,2} = B \cdot (q-1) \tag{6}$$

communications. At this point every element in the set S has been assigned to a bucket, and the host machine knows exactly how many elements each node assigned to each bucket.

3) Phase 1: Identify Active Buckets

While the host also participated in the assignment and counting of the data in its local memory, the host receives local bucket counts from each node and adds them to a universal bucket count. To know which bucket contains one of the desired order statistics, the host runs through this vector carrying a cumulative sum. When the cumulative sum is less than k after summing $i-1$ buckets but greater than or equal to k after adding the count from bucket i, then the kth order statistics lies in bucket i. In this simple pass through the universal bucket count vector, we flag as active all buckets which contain at least one of the desired order statistics.

In the implementation, we have a vector of desired order statistics, `DesiredOrderStats`, and an equal length vector indicating which bucket contains that order statistic. Since multiple order statistics might end up in the same bucket, we form an auxiliary vector, `UniqueActiveBuckets`, and we record the number of elements remaining in each unique active bucket with the vector `UniqueActiveCounts`. The elements assigned to the unique active buckets are now the only candidates still in consideration to be one of our desired order statistics. This subroutine requires only local action on the host and no communication between the host and the nodes ($c_{1,3} = 0$).

4) Phase 1: Reduce

To start this fourth subroutine, Reduce, the data still in contention are known to be exactly those data assigned to one of the buckets in `UniqueActiveBuckets`; this reduction necessitates updating the order statistics. If α is the kth order statistic from S, and we are able to eliminate ℓ elements less than α, then α becomes the $(k-\ell)$th order statistic of the reduced set. We apply this logic to the full set of order statistics by using the cumulative bucket counts and the counts of the active buckets, `UniqueActiveCounts`. The host stores the updated order statistics in `UpdatedOrderStats`.

To restrict our attention to the data still in contention, the data on the nodes assigned to a bucket in `UniqueActiveBuckets` will simply be written to a new local vector, `ReducedVector`. Thus, the host must send `UniqueActiveBuckets` and `UniqueActiveCounts` to each of the $q-1$ nodes. This requires at most

$$c_{1,4} = 2 \cdot m \cdot (q-1) \tag{7}$$

communications.

Let's again fix $r \in \{0, \ldots, q-1\}$, let `vec` represent the n_r elements of S on node r, and consider the local element $v = \text{vec}[\ell]$ for some fixed $\ell \in \{0, \ldots, n_r - 1\}$. To see if v is still in contention to be one of our order statistics, we check the bucket to which we assigned it and stored locally in `buckets`$[\ell]$. If `buckets`$[\ell]$ is not one of the active buckets in `UniqueActiveBuckets`, we do nothing. On the other hand, if `buckets`$[\ell]$ is a member of `UniqueActiveBuckets`, we want to write v to the new auxiliary vector.

The nodes, in particular the GPUs on these nodes, still have the same information they ended with after Assign Buckets. In addition to `leftPivots`, `rightPivots`, and `slopes`, each node has a local counter `CounterArray`, which was cumulatively summed across blocks. This makes the new task of writing the elements of the active buckets to a new reduced vector much faster. If v is to be written to the new auxiliary vector, we can use the information in `CounterArray` to write v from its specific instruction block to a specific segment of `ReducedVector`. To avoid conflicts, the address we write to is given as a pointer that is atomically incremented. By writing to these very small, specific segments of `ReducedVector`, potential atomic conflicts are overwhelmingly avoided.

After retaining only the elements in the active buckets in the vector `ReducedVector` on each node, and recomputing the order statistics in `UpdatedOrderStats`, we have a new distributed multiselection problem to address. If the sampling strategy provided good approximate order statistics for the pivot intervals, we expect them to form a KDE and thus the length of `ReducedVector` to be approximately $\frac{m}{B} \cdot n$.

5) DISTRIBUTEDBUCKETMULTISELECT: Phase 2

For DISTRIBUTEDBUCKETMULTISELECT, each node will send its portion of `ReducedVector` to the host where the host will apply SORT&CHOOSE to select `UpdatedOrderStats` from the consolidated `ReducedVector`. With data evenly distributed among the q nodes coming from a uniform distribution, this final step has the expected communication cost of

$$c^*_{2,1} = \frac{m}{B} \cdot \frac{n}{q} \cdot (q-1). \tag{8}$$

B. Distributed Iterative Bucket Multiselection

The communication requirement (8) to apply SORT&CHOOSE after the Reduce subroutine of DISTRIBUTEDBUCKETMULTISELECT may still be prohibitively expensive, e.g. when there are more than 2^{23} elements in the data (∼32MB) and we want 101 order statistics (see Fig. 1). To avoid consolidating the reduced data at the host, we employ the strategy of the original BUCKETSELECT algorithm [**?**] and iterate simultaneously on all of the active buckets at the nodes.

Our main algorithm, DISTRIBUTEDITERATIVEBUCKETMULTISELECT or DIBMS, performs an identical first phase to the first phase of DISTRIBUTEDBUCKETMULTISELECT. Instead of sending the local reduced vectors to the host, DIBMS leaves the data on the local machines and iterates on these reduced vectors. To do so, DIBMS defines new buckets based on the active buckets from Phase 1, and proceeds through the same four subroutines in a distributed fashion. The iteration continues until the order statistics are identified.

Phase 2 is ready to move forward with the problem defined by the Reduce subroutine from Phase 1, or from a previous iteration from Phase 2. The reduction in Phase 1 or the previous iteration produces a new problem that comes with important knowledge about the data. Phase 2 remembers the relevant information from the previous pass through the data in order the create the new pivot intervals and define the buckets for this iteration. In the following, we use "old" to indicate the information or variables have come from the prior phase or iteration.

1) Phase 2: Create Buckets

The previous reduction provides a new problem definition along with precise knowledge of exactly how many elements of S live in the each of the old active buckets. The endpoints of the old active buckets define our new pivot intervals. Fortunately, those endpoints were determined by the linear projections defining the bucket assignments ((5) or (13)). So, we can simply invert those linear projections to calculate the exact left and right endpoints of old active buckets. When we compute these left and right endpoints, we use them as the left and right pivots to form a pivot interval for each unique active bucket. So we now have P pivot intervals where P is the number of unique active buckets from the previous reduction (either phase 1 or the last iteration).

In other words, we now have

$$P = \texttt{numUniqueActive} \leq m$$

left and right pivots to define. For each $i = 0, \ldots, P-1$, a binary search finds the index j of the old pivots so that the active bucket is in the jth pivot interval. Letting `olPvt` and `orPvt` be abbreviations for `oldleftPivots` and `oldrightPivots`, the new pivots are defined as follows:

$$\begin{aligned} &\text{for } i = 0, \ldots, P-1, \\ &\quad \mathit{bucket} = \texttt{oldUniqueActiveBuckets}[i], \\ &\quad \text{Fix } j : \mathit{bucket} \subset [\texttt{olPvt}[j], \texttt{orPvt}[j]]\,. \\ &\text{Now let} \\ &\quad \mathit{left} = \texttt{oldleftPivots}[j], \\ &\quad \mathit{precount} = \texttt{oldpreCount}[j] = \sum_{k=0}^{j-1} b_k, \\ &\quad \mathit{slope} = \texttt{oldslopes}[j], \end{aligned} \tag{9}$$

so that

$$\begin{aligned} \texttt{leftPivots}[i] &= \mathit{left} + \frac{\mathit{bucket} - \mathit{precount}}{\mathit{slope}}, \\ \texttt{rightPivots}[i] &= \mathit{left} + \frac{\mathit{bucket} + 1 - \mathit{precount}}{\mathit{slope}}. \end{aligned} \tag{10}$$

In the spirit of the KDE intervals, we want to allot the buckets to the pivot intervals so that they will all capture roughly the same number of elements. Old active buckets containing lots of elements will need more buckets in this iteration than old active buckets with relatively few elements assigned. With `totalCount` being the number of elements left in our new vector, we define the number of buckets per pivot interval as

$$\begin{aligned} &\text{for } i = 0, \ldots, P-1 \\ &b_i = \text{round}\left(\frac{\texttt{oldUniqueActiveCounts}[i]}{\texttt{totalCount}} \cdot B\right). \end{aligned} \tag{11}$$

To ensure we always make progress, we insist that every pivot interval get at least two buckets, $b_i \geq 2$ for all i, while also ensuring we use at most B buckets, $B = \sum b_i$.

These new allotments require a new exclusive sum of the buckets allotted to previous pivot intervals. Starting with $\texttt{preCount}[0] = 0$, we then have

$$\texttt{preCount}[i] = \sum_{k=0}^{i-1} b_k$$

for $i = 1, \ldots, P-1$. Finally, with these precounts, we can define the new set of slopes for these pivot intervals. For the slopes, the definition is identical to that in (2):

$$\begin{aligned} &\text{for } i = 0, \ldots, P-1 \\ &\texttt{slopes}[i] = \frac{b_i}{(\texttt{rightPivots}[i] - \texttt{leftPivots}[i])}. \end{aligned} \tag{12}$$

We now have completely defined the buckets for this phase and no longer need to remember any of the old information. The host must send all of this information to each of the $q-1$ nodes. Sending `leftPivots`, `rightPivots`, $\{b_i\}$, and `slopes` to the remote nodes requires

$$c_{2,1} = 4 \cdot P \cdot (q-1)$$

communications.

2) Phase 2: Assign Buckets

The Assign Buckets subroutine is essentially identical to that in Phase 1. While the number of buckets allocated to each pivot interval may not be the same, this is accounted for in (12) so that the completely parallelized bucket assignment is defined by

$$\begin{aligned} \texttt{buckets}[i] = \lfloor \texttt{slopes}[i] \cdot (\texttt{vec}[i] - \texttt{leftPivots}[i]) \rfloor \\ + \texttt{preCount}[i]. \end{aligned} \tag{13}$$

In identical fashion to phase 1, when an element of the vector is assigned to bucket number i, a counter is incremented. Since we are using the same number of buckets, B, `CounterArray` remains a $B \times \texttt{numBocks}$ array. Prior to the bucket assignment it simply needs to be allocated in shared memory and set to zero. The use of an array allows each multiprocessor to increment its own atomic counter in order to minimize any atomic queuing. In order to determine which of these newly assigned buckets contain some of the desired order statistics, we inclusive sum the `CounterArray` across the blocks. The final column containing the local bucket counts is the sent back to the host; this again requires

$$c_{2,2} = B \cdot (q-1)$$

communications.

3) Phase 2: Identify Active Buckets

Identify Active Buckets is also unchanged: the host accumulates the buckets counts from the remote nodes and includes its own assignment tallies to form a complete count of bucket assignments for the active data. The host passes through this universal count, and, for each desired order statistic, determines the bucket that must contain it. It records both the bucket and the number of elements in this bucket for each order statistic. From these vectors, we can form UniqueActiveBuckets and UniqueActiveCounts as in Phase 1. Again, there are no distributed communications associated with Identify Active Buckets ($c_{2,3} = 0$).

4) Phase 2: Reduce

We begin the reduction by determining if we have found any of the desired order statistics. With both the KDE of Phase 1 and the proportionally distributed buckets in Phase 2, bucket counts should be roughly equal. Therefore, we have an expected reduction in problem size by a factor of $\left(\frac{m}{B}\right)^{1+k}$ where k is the number of iterations in Phase 2. So, if we are searching for the $m = 101$ percentiles with $B = 2^{13}$ buckets, then the first time we arrive at the Reduce subroutine in phase 2, we would expect our problem size to be roughly 1.52×10^{-4} times its original size. This reduction allows us to extract the identified order statistics for comparatively small communication and computation costs.

If a specific bucket is known to contain an order statistic and also has a count of 1, we have identified one of our desired order statistics. We need to retrieve the value of this single element from its currently unknown location, possibly on any of the nodes or the host. To minimize communication, DIBMS forms a list of all buckets that have a unique element that must be an order statistic. It sends this target list to all remote nodes who utilize their local counts to determine which desired order statistics reside in the data on that node. Each node then sends any order statistics it finds back to the host. This requires at most

$$m \cdot (q-1) + m$$

communications since the target list of at most m order statistics was sent to the $q-1$ nodes with at most m communications of the correct values back to the host.

It also then updates the list of yet-to-be-found-but-still-desired order statistics in UpdatedOrderStats, removes the corresponding buckets from UniqueActiveBuckets, and updates the UniqueActiveCounts. The two vectors of unique buckets and corresponding counts need to be sent out to all of the nodes; this requires $2 \cdot m \cdot (q-1)$ communications. Therefore, this subroutine in phase two uses at most a total of

$$c_{2,4} = 3 \cdot m \cdot (q-1) + m$$

communications.

To write the active elements to ReducedVector, the Reduce subroutine has all the same advantages it had in Phase 1. Locally, each node knows how many local elements were assigned to each bucket by each parallel instruction block. The cumulative summing at the end of the Assign Buckets subroutine helps the blocks know exactly which portions of a new[2] vector it will write. After the reduce subroutine, each node has a local version of ReducedVector.

5) Phase 2: Stopping Criteria

An often overlooked but incredibly important aspect of an algorithm is stopping it correctly. As noted in the Reduce subroutine of Phase 2 , each time an active bucket contains just a single element, we can extract the associated value from the data and remove this order statistic from the problem. In the ideal scenario, this happens with all desired order statistics. In reality, we must also be able to identify when the value of the desired order statistic is a repeated value in the data set.

Suppose we want to find the kth largest value, but that value appears in the data set multiple times so that it is also the jth and ℓth largest value for $j < k < \ell$. The bucket to which this value is assigned will never have fewer than $\ell - j$ values in it. Rather than scanning the entire data set to check the values assigned to a specific bucket, we use the proxy that the data size is not changing. For example, if we run an iteration of phase 2 and have w elements in our reduced vector, and then run another iteration of phase 2 resulting in a new reduced vector also containing w elements, we anticipate that the active buckets all contain only a single, repeated value.

Rather than check each of the active buckets for any variance, we simply run an extra iteration. When an iteration does not reduce the problem size, the probability the active buckets contain multiple values is very small. Since that probability is not zero, we repeat the reduction process one more time. If the data does not change for a second consecutive iteration, we declare that all remaining active buckets have a single repeated value; we return that value as the order statistic.

This extra iteration hedges against a very low probability event and costs the algorithm in performance. The payoff is this heuristic nearly guarantees we are stopping the algorithm correctly. In the incredibly low probability event[3] that we select an incorrect value $v + \epsilon$ for this order statistic value v, this extra iteration ensures that ϵ is reduced by an extra factor $\frac{1}{b_i}$ where b_i is the number of buckets allotted to the active bucket i containing v.

III. Analysis of Expected Communication Costs

In Section 5 we provided expected communication costs, or upper bounds on those costs, for each of the four subroutines in each phase. Those are consolidated in Table 2.

We can see from this collection that the expected communication cost of DISTRIBUTEDBUCKETMULTISELECT is

[2]In implementation, this "new" vector is the reuse of the vector that was reduced in the preceding iteration or phase.

[3]No incorrect order statistics were selected in empirical testing while using this stopping criteria.

TABLE 2. **Expected communication costs by phase and subroutine.**

Phase	Subroutine	Expected Cost: DISTRIBUTEDBMS	Expected Cost: DIBMS
1	Create	$50(q-1)+1024$	$50(q-1)+1024$
1	Assign	$B(q-1)$	$B(q-1)$
1	Reduce	$2m(q-1)$	$2m(q-1)$
2	Create	$\frac{m}{B}\cdot\frac{n}{q}\cdot(q-1)$	$4P(q-1)$
2	Assign	0	$B(q-1)$
2	Reduce	0	$3m(q-1)+m$

TABLE 3. **Expected Communication Savings. The ratios of expected communications for DIBMS to SORT&CHOOSE and to DISTRIBUTEDBMS.**

Data Size	# OS	# Nodes	Expected Communications Ratio	
n	m	q	$\frac{\text{DIBMS}}{\text{SORT\&CHOOSE}}$	$\frac{\text{DIBMS}}{\text{DISTRIBUTEDBMS}}$
26	11	4	0.0060	2.3476
26	101	4	0.0080	0.7358
26	501	4	0.0144	0.3036
26	1001	4	0.0241	0.2578
28	11	4	0.0015	1.0736
28	101	4	0.0024	0.2479
28	501	4	0.0041	0.0895
28	1001	4	0.0067	0.0730

given by

$$\left(\sum_{i=1}^{4} c_{1,i}\right)+c^*_{2,1} = \left(50+B+2m+\frac{m}{B}\frac{n}{q}\right)\cdot(q-1)+1024.$$

DISTRIBUTEDBUCKETMULTISELECT is not iterative so this cost does not depend on the required number of iterations.

On the other hand, DIBMS does depend on the number of iterations, Itr, in phase 2. Therefore, the expected communication for DIBMS is given by

$$\begin{aligned}\left(\sum_{i=1}^{4} c_{1,i}\right)+Itr\cdot\left(\sum_{j=1}^{4} c_{2,j}\right)\\ &= (50+B+2m)\cdot(q-1)+1024\\ &\quad + Itr\cdot((4P+B+3m)(q-1)+m)\end{aligned}$$

For the worst case scenario of uniformly spaced order statistics each needing their own active bucket, we expect that uniformly distributed data of size n will be reduced in each iteration by the factor $\frac{m}{B}$. The expected number of iterations is

$$\mathbb{E}[iter] = \log_{\frac{B}{m}}(n).$$

In accordance with the heuristic stopping condition described above, DIBMS will take one extra iteration.

In Table 3, we compute the expected ratios of communication costs for DIBMS compared to both SORT&CHOOSE and DISTRIBUTEDBMS. Regardless of the size of the data set, the number of order statistics, or the number of nodes, we expect a significant savings in communications using either of our algorithms versus SORT&CHOOSE. We can also see that when the number of order statistics is small ($m = 11$) the communication requirements of DISTRIBUTEDBMS are low enough that the iterative version of the algorithm requires more communication.

When we want a moderate number of order statistics ($m = 101$), the expected communication costs of the two algorithms are very similar until around $n = 2^{25}$. As the problem sizes grow, DIBMS has its advantage grow. We can see that the number of expected communications for SORT&CHOOSE is more than $600\times$ that of DIBMS; DISTRIBUTEDBMS requires roughly $4\times$ the communications of DIBMS. For a large number of order statistics, DIBMS is expected to be superior in communication cost starting around $n = 2^{23}$ for $m = 501$ and $n = 2^{22}$ for $m = 1001$.

In the next section, we will see that the communication savings is further enhanced by the computational saving. The ratios of performance times closely mirror Table 3, but are more favorable to DIBMS.

IV. Performance Comparisons

In this section, we measure and compare the performance of SORT&CHOOSE, DISTRIBUTEDBMS, and DIBMS through empirical testing. The tests are random and the order the algorithms are asked to solve the same problems is randomly assigned. We see in these results that while communication costs are a key factor, the computational advantages of DIBMS and DISTRIBUTEDBMS are also important.

A. Empirical Testing Environment

These tests were conducted on a local area network with machines of identical hardware and operating systems. Each machine running Debian GNU/Linux 12 (bookworm) is equipped with four Intel Xeon Gold 6326 CPUs @ 2.90GHz and one NVIDIA T1000 Graphics Processing Unit from the Turing architecture. Each T1000 GPU has 8GB of GDDR6 global memory and 896 compute cores across 14 streaming multiprocessors (SM) with 96KB of RAM of dedicated shared memory per SM. The code was compiled in sequence for distributed execution using CUDA Version: 12.4 and Open MPI 4.1.4.

For each test, a series of random data sets S were constructed using CUDA. Each distributed algorithm of SORT&CHOOSE, DISTRIBUTEDBMS, and DIBMS solved the multiselection problem and verified their results against each other[4]. No errors were detected in any of the tests by any of the algorithms.

The data in S was drawn from the uniform, normal, or half normal distribution. The reduction in performance for normal or half-normal distributions was as expected based on results from [?]. The data presented in this section is for data sets drawn from the uniform distribution; the additional results for data drawn from the normal and half-normal data sets can be found in the Supplementary Material [?].

[4]Many tests were also checked against the process of consolidating all of the data set at the host and using a local SORT&CHOOSE. No errors were found during any such test.

TABLE 4. **Mean timings (ms) and performance ratios for DIBMS selecting percentiles and $1/5$-percentiles.**

		Data Type	Float					
		Data Distribution	Uniform			Normal		
Data Size	# OS	# Nodes	Runtime (ms)	Observed Ratios		Runtime (ms)	Observed Ratios	
n	m	q	DIBMS	$\frac{\text{DIBMS}}{\text{SORT\&CHOOSE}}$	$\frac{\text{DIBMS}}{\text{DISTRIBUTEDBMS}}$	DIBMS	$\frac{\text{DIBMS}}{\text{SORT\&CHOOSE}}$	$\frac{\text{DIBMS}}{\text{DISTRIBUTEDBMS}}$
2^{24}	101	4	97.09	0.0224	1.1472	112.95	0.0261	1.6830
2^{24}	501	4	146.09	0.0337	0.4483	144.74	0.0334	0.3824
2^{26}	101	4	101.82	0.0059	0.4106	118.44	0.0068	0.6485
2^{26}	501	4	139.26	0.0080	0.1143	165.26	0.0095	0.1148
2^{28}	101	4	116.59	0.0017	0.1277	132.87	0.0019	0.2084
2^{28}	501	4	153.87	0.0022	0.0324	194.97	0.0028	0.0346
		Data Type	Double					
		Data Distribution	Uniform			Normal		
Data Size	# OS	# Nodes	Runtime (ms)	Observed Ratios		Runtime (ms)	Observed Ratios	
n	m	q	DIBMS	$\frac{\text{DIBMS}}{\text{SORT\&CHOOSE}}$	$\frac{\text{DIBMS}}{\text{DISTRIBUTEDBMS}}$	DIBMS	$\frac{\text{DIBMS}}{\text{SORT\&CHOOSE}}$	$\frac{\text{DIBMS}}{\text{DISTRIBUTEDBMS}}$
2^{24}	101	4	83.74	0.0097	0.5605	83.88	0.0097	0.5235
2^{24}	501	4	132.06	0.0153	0.2090	132.64	0.0153	0.1835
2^{26}	101	4	98.61	0.0028	0.1961	101.48	0.0029	0.1817
2^{26}	501	4	149.28	0.0043	0.0613	156.46	0.0045	0.0566
2^{28}	101	4	126.98	0.0009	0.0665	128.90	0.0009	0.0608
2^{28}	501	4	188.51	0.0014	0.0194	190.13	0.0014	0.0171

The order statistics were also drawn from a variety of distributions. However, uniformly spaced order statistics such as the percentiles is the most challenging for DISTRIBUTEDBMS and DIBMS. When the order statistics are uniformly spaced, the likelihood of having one active bucket for each order statistic is maximized. Therefore, we only present results for uniformly spaced order statistics. See [**?**] for a discussion on how the algorithms would benefit from other distributions of order statistics.

B. Results

The performance of SORT&CHOOSE is as expected in Figure 1 with complete dependence on the size of the data set, $n = |S|$. Similarly, we can observe from (b) and (c) that when the number of desired order statistics is high, DISTRIBUTEDBMS also has a run time that is proportional to $n = |S|$. In all three cases, we see that the communication avoiding DIBMS has limited growth as the vector length n grows.

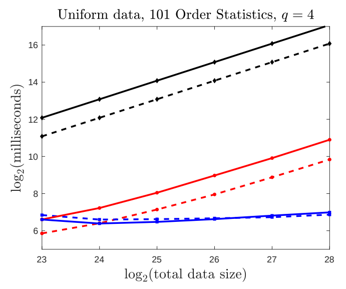
Instead, the run time for DIBMS is proportional to the number of order statistics. With uniformly spaced order statistics, we can assume that the number of pivots, P, in each iteration of phase 3 is equal to the number of order statistics m. Assuming $P = m$, an equivalent expression for the communication cost of phase 2 of DIBMS is

$$\sum_{j=1}^{4} c_{2,j} = Itr \cdot (7(q-1)+1) \cdot m + Itr \cdot B \cdot (q-1).$$

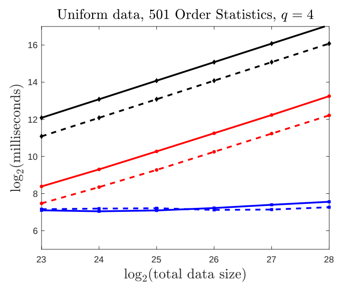
Since the number of buckets B and the number of nodes q are independent of the problem, we see that this communication cost is proportional to the number of order statistics. In Figure 2 observe that the run time performance of DIBMS matches that intuition.

Our interpretation of the overall performance costs for these algorithms is further supported with Figure 3. Here the size of the data set is fixed while the number of desired order statistics grows from 101 to 1001. We see that the change in number of order statistics has no bearing on the performance of SORT&CHOOSE since the entire data set is sorted. We can also see the performance of DISTRIBUTEDBMS is negatively impacted by the number of order statistics for data of this size. The time required to find the order statistics is growing super linearly. The most consistent performance is from DIBMS which we understand to be linearly dependent on the number of order statistics. The positive slope represents the increase in algorithm run time that is also impacted by a likely increase in the number of iterations. Also notice that because DIBMS only sends control parameters, DIBMS run time is not impacted by data type. The sorting step in the SORT&CHOOSE and DISTRIBUTEDBMS are responsible for decreased performance on double precision data.

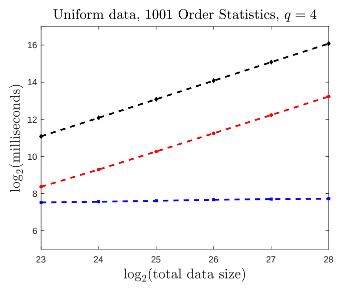
Finally, we investigate the run time performance impact of the number of nodes in our distributed system. In Figure 4 we present this data in two ways. First, in (a) we fix the total size of the data and spread it evenly across the nodes so the data points above the horizontal axis are for a fixed total size. In other words, the run time for 8 nodes is for solving a problem where each node has $\frac{n}{8}$ elements on it. In (b), we instead plot the number of elements at each node. Note that the plots look relatively similar in both cases because the run time of the algorithm grows slowly.



(a)



(b)



(c)

FIGURE 1. **Time ($\log_2$(ms)) required for SORT&CHOOSE (black) DISTRIBUTEDBMS (red), DIBMS (blue) to find the (a) 101 percentiles, (b) 501 $\frac{1}{5}$th-percentiles, and (c) 1001 $\frac{1}{10}$th-percentiles for varying sizes of data sets, S, with the entries drawn from the uniform distribution when the data is of type double (solid), float (dashed), or unsigned integer (dot-dashed).**

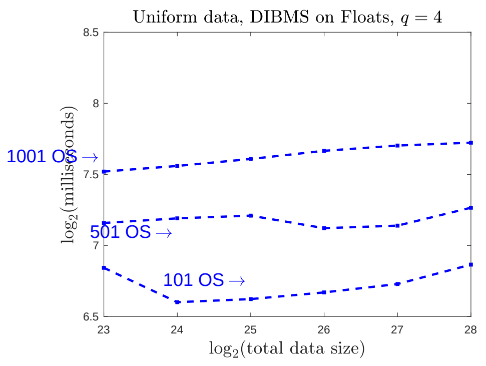


FIGURE 2. **Time (ms) required for DIBMS (blue) to find the 101, 501, and 1001 uniformly spaced order statistics for varying length of vectors with the entries drawn from the uniform vector distributions when the data is a double (solid) or float (dashed).**

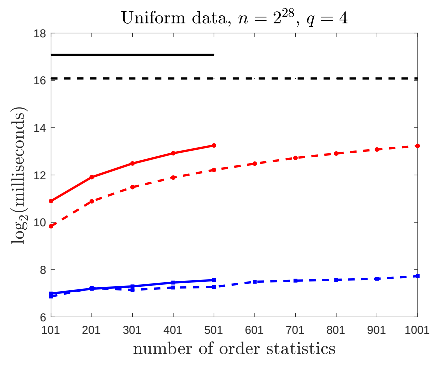
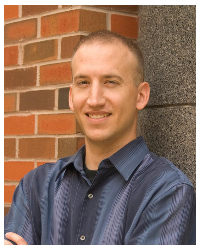


FIGURE 3. **Time ($\log_2$(ms) required for SORT&CHOOSE (black) DISTRIBUTEDBMS (red), DIBMS (blue) to find uniformly spaced order statistics for vectors of length of 2^{25} with the entries drawn from the uniform distribution when the data is a double (solid), or float (dashed).**

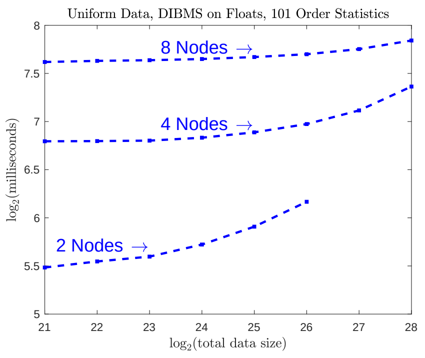
V. Conclusion

With data sets distributed across many locations, it can be advantageous to understand the full data set with a concise representation. Determining exact order statistics is superior to sampling methods and can provide highly accurate representations of the full set. Accurate approximate density functions are easily formulated with order statistics.

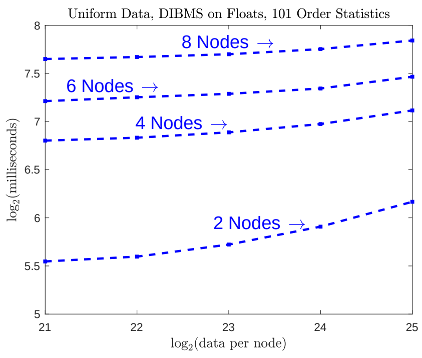
When the data size of the number of order statistics are small, and you are willing to communicate a nontrivial portion of your data, DISTRIBUTEDBMS can be highly effective and much faster than SORT&CHOOSE. Even when distributed data sets are enormous, DIBMS will obtain the percentile order statistics in fractions of a second. DIBMS can find 501 order statistics from a uniform data set with 2^{27} in roughly 170 milliseconds, about $190\times$ faster than SORT&CHOOSE while using between $50\times$ (8 nodes) and $300\times$ (2 nodes) fewer communications.



Jeffrey D. Blanchard received a B.A. (Hons.) in mathematics from Benedictine College (Kansas). After serving as an officer in the US Army, he earned both an A.M. and Ph.D. in mathematics from Washington University in St. Louis where he held a Department of Homeland Security Fellowship. He is currently a Professor of Mathematics at Grinnell College (Iowa) and Director of the Wilson Center for Innovation and Leadership. Previously, he was a National Science Foundation International Research Fellow at the School of Mathematics and the School of Electronics and Engineering at the University of Edinburgh (2010) and a VIGRE Research Assistant Professor in the Department of Mathematics at the University of Utah (2007-2009). He was an MAA Project NExT Fellow (2008-2009) and a Harris Faculty Fellow (2013-2014). His research interests include composite dilation wavelets, compressed sensing, matrix completion, scientific computing with graphics processing units, and directing undergraduate research.



(a)



(b)

FIGURE 4. Time ($\log_2$(ms)) required for DIBMS (blue) to find 101 uniformly spaced order statistics for data sets distributed across 2, 4, or 8 nodes. The data has elements drawn from the uniform distributions for types double (solid) and float (dashed). (a) The horizontal axis is the total data size. (b) The horizontal axis is the size of data on each node.

Ruizhe Fu received a B.A. in computer science from Grinnell College (Iowa) and a B.S.E. in computer engineering from Columbia University. He once cheated to win a single game of table tennis against Blanchard, but the other times Blanchard was victorious. He probably can't beat Blanchard at FIFA (now EA FC) either.

Tristan Knoth received a B.A. in computer science and mathematics from Grinnell College (Iowa) and a Ph.D. in computer science from the University of California, San Diego. He is pretty tall.