Parallel Q-Digest Design, Implementation and Benchmarking

Ettore Miglioranza
mat.257311
University of Trento
38123 Povo TN, Italy
ettore.miglioranza@studenti.unitn.it

Anthony Tricarico
mat.254957
University of Trento
38123 Povo TN, Italy
anthony.tricarico@studenti.unitn.it

***Abstract*—This paper introduces and benchmarks a parallel implementation of the Quantile Digest (Q-Digest) algorithm to produce descriptive statistical summaries of positive integer-valued data distributions. Parallelization is achieved using the Message Passing Interface (*MPI*), employing a data privatization model followed by a logarithmic tree-reduction phase. Experimental evaluations using synthetic datasets of up to 16 million elements show that the implementation achieves significant performance gains, including a 26-fold execution time reduction on 64 cores for the largest workloads. A notable finding is the emergence of *super-linear efficiency* (reaching approximately 124%) at the 16-core mark for large datasets, which is attributed to optimized CPU cache locality. While the results indicate that communication overhead can lead to diminishing returns for smaller datasets at high core counts, the proposed framework remains highly effective for large-scale, memory-constrained data analysis.**

***Index Terms*—MPI, parallelization, quantiles, data summaries**

I. Introduction

Understanding and effectively summarizing a data distribution using quantiles may appear simple and is often taken for granted whenever using statistical software. However, when handling large amounts of data, the process of computing quantiles of a distribution can be slow and resource-intensive [1]. For this reason, the Quantile Digest (Q-Digest) algorithm was introduced as a way of streamlining the quantile computations of positive integer-valued data distributions [2]. Although there are ways to extend quantile algorithms to potentially unbounded streams of data [3] the focus of this work is on bounded data, since originally the Q-Digest algorithm was designed to be easily portable to facilitate its implementation in sensors (and related networks) . This portability stems from various nice properties that the algorithm possesses that keep its memory usage and expected error (i.e., the divergence between the real quantile and the estimated quantile) bounded. Indeed, there exists a mathematically quantifiable trade-off between the amount of memory used and the expected error, which implies that with less strict constraints on memory usage the expected error would be reduced to a minimum. Given the importance of such algorithm, the aim of this paper is to provide parallelization strategies that can be safely used to make the execution of the program faster. First, the main ideas of the algorithm will be presented more in depth to understand how the properties discussed above are taken advantage of in its original implementation. Second, the proposed methodology and implementation are discussed to determine which characteristics of the problem at hand led to the current implementation. Third, the paper introduces the main experimental results contrasting the serial implementation with the main parallelization strategy that was implemented (i.e., MPI). Metrics such as *speedup* and *efficiency* are also detailed here to provide a solid benchmark analysis. Finally, results from the parallelization strategy are reported along with their respective interpretations.

II. Related Works

This section explores the original implementation of the Q-Digest algorithm as portrayed in the introductory paper by Shrivastava and collaborators [2]. The main aim of this section is to explore the most relevant details about the algorithm and how it is structured in its serial implementation.

A. Implementation

The Q-Digest is modeled as a complete binary tree (T) built over the discrete value range [1,σ], where σ is the largest possible value in the data universe of application. Each node v in the tree represents a "bucket" (i.e., a specific subrange) and maintains a counter, `count(v)`, of the "sensor" readings falling within that interval. To maintain efficiency, the algorithm uses a compression mechanism governed by two fundamental properties using the total number of readings n and a user-defined parameter k. The first property, referred to as *capacity property*, ensures that except for leaf nodes, no node v in the digest may have a count exceeding $\lfloor \frac{n}{k} \rfloor$. The second property, also called *density constraint*, states that for a node v to remain distinct, the combined counts of v, its parent v_p, and its sibling v_s must satisfy Inequality 1.

$$count(v) + count(v_p) + count(v_s) > \lfloor \frac{n}{k} \rfloor \tag{1}$$

Two main guarantees follow from the two properties illustrated above.

1) Memory Guarantees: The parameter k plays a vital role in this implementation since it acts as a compression parameter serving as the primary lever for balancing approximation accuracy against memory utilization. Indeed, setting

this parameter allows to have strict memory guarantees which keep the digest size bounded ensuring that it never exceeds $3k$ nodes regardless of the volume of input data.

2) Error Guarantees: k also controls the error guarantee of the estimated quantile of the data distribution. Specifically, for a given amount of allocated memory m (i.e., the number of active nodes stored in the digest), the Q-Digest can answer quantile queries with a relative error ϵ bounded by Inequality 2.

$$\epsilon \leq \frac{3 \log \sigma}{m} \tag{2}$$

B. Complexity

The computational complexity of the Q-Digest algorithm is primarily governed by the structural traversals required to maintain its space and error invariants, specifically through the compression, merging, and querying phases.

1) Compression and Construction: The construction and maintenance of the Q-Digest rely on the `COMPRESS` algorithm, which ensures the data structure adheres to the required capacity and density constraints. The compression logic performs a bottom-up traversal of the binary tree T defined over the data universe σ. The algorithm iterates through the levels of the tree, initializing at $l = \log \sigma - 1$ and decrementing to 0. At each level, it validates the digest properties for the active nodes, merging children into their parents if the combined weight of the node, its sibling, and its parent falls below the threshold $\lfloor n/k \rfloor$. Consequently, the complexity of the compression phase is proportional to the size of the digest scaled by the height of the tree, $O(m \cdot \log \sigma)$, where m is the number of nodes currently stored in the digest.

2) Merging Digests: In the context of a distributed sensor network, as presented in the original paper [2], the aggregation of data necessitates the combination of local summaries. The `MERGE` operation is responsible for combining two distinct digests, Q_1 and Q_2, into a single consistent summary. This is achieved via a two-step process. First, the algorithm computes the union $Q \leftarrow Q_1 \cup Q_2$, summing the counts of nodes that cover identical ranges. Following the union, the `COMPRESS` function is invoked on the combined structure using the aggregate total count $n_1 + n_2$. This re-compression ensures that the resulting digest respects the space bound of $3k$ nodes, maintaining the efficiency of the distributed aggregation tree.

3) Query Processing: The complexity of extracting statistical information from the Q-Digest is dominated by the requirement to order the internal nodes. To answer a quantile query (e.g., determining the value with rank qn, with $0 \leq q \leq 1$) the algorithm must first sort the m nodes of the digest into a list L based on increasing right endpoints (range maxima). Once sorted, the algorithm performs a linear scan of L, accumulating counts until the threshold qn is exceeded. Therefore, the time complexity for quantile queries is dominated by the sorting step, executing in $O(m \cdot \log m)$.

III. Methodology and implementation

Following the analysis of the sequential Q-Digest, this section details our parallel implementation developed using the Message Passing Interface (MPI) API. We provide a rigorous examination of the data dependencies stemming from the tree-based insertion logic, justifying the choice of a distributed data privatization model. Furthermore, we elucidate the structural relationship between the local digest construction and the global tree-reduction phase, which ensures consistency across processes. Finally, this section describes the PBS directives utilized to schedule jobs on the HPC cluster along with the specific datasets employed to evaluate the scalability and accuracy of our parallelized approach. The full repository is open and available on GitHub[1]. Moreover, for the sake of brevity, in this paper we only discuss the main core structures of our implementation while other details are better described in the online interactive project documentation[2].

A. Common Methods

Before detailing the distributed implementation, it is essential to identify the algorithmic components shared by both the sequential and parallel versions. Both approaches rely on the core Q-Digest operations to maintain the structure's statistical guarantees.

The most significant commonality is the `insert` function (see Algorithm 1), which performs the path traversal and node updates independently of the execution environment. By ensuring that each MPI process utilizes the same insertion logic locally, we maintain functional consistency with the serial counterpart. Furthermore, the `COMPRESS` and `MERGE` operations are utilized in both scenarios: the former to prune the tree and maintain the $3k$ node limit, and the latter to combine local structures, a capability that forms the basis of our global reduction phase.

The parallel implementation of our Q-Digest algorithm necessitates a coordinated multi-phase workflow to ensure statistical consistency across distributed memory spaces. Following the initial data distribution, each MPI process independently constructs a local Q-Digest, effectively summarizing its distinct portion of the global dataset (see Algorithm 1). Once local digests are finalized, the algorithm enters a hierarchical reduction phase. During this stage, partial Q-Digests residing on different processes are systematically combined through a binary merge tree, utilizing a custom MPI-based communication protocol to preserve the theoretical error bound (see Algorithm 2 and 3). This global consolidation ensures that the final root process contains a comprehensive summary equivalent to a serial execution on the unified dataset.

B. Data Dependencies

When delving into parallel implementation, a primary challenge lies in determining which operations within a given data structure can be safely offloaded to multiple processors. Before attempting a parallel version of an algorithm, it is vital to conduct a formal data dependency analysis. This allows us to identify potential race conditions or deadlocks that could compromise execution order or data integrity. By mapping

[1]https://github.com/ettoremiglioranza1012/Q-Digest

[2]https://ettoremiglioranza1012.github.io/Q-Digest/

Algorithm 1 Q-Digest Common Logic

function INSERT(q, *key*, *count*)
$curr \leftarrow q \rightarrow root$
while $lower\_bound \neq upper\_bound$ **do**
  $mid \leftarrow$ CALCULATEMID($lower, upper$)
  if $key \leq mid$ **then**
    $curr \leftarrow$ GOLEFT($curr$) {Create if NULL}
  else
    $curr \leftarrow$ GORIGHT($curr$) {Create if NULL}
  end if
end while
$curr \rightarrow count \leftarrow curr \rightarrow count + count$
$q \rightarrow N \leftarrow q \rightarrow N + count$

function MERGE($q1$, $q2$)
{Used in both Serial and MPI Reduction}
for all nodes n in $q2$ **do**
  INSERT($q1, n \rightarrow key, n \rightarrow count$)
end for

function COMPRESS(q, n)
{Post-order traversal to ensure bottom-up merging}
if n is NULL **then**
  return
end if
COMPRESS(q, $n.left$)
COMPRESS(q, $n.right$)
if n is not root **then**
  $isDeleted \leftarrow$ DELETEIFEMPTYLEAF(q, n)
  if not $isDeleted$ **then**
    $triadSum \leftarrow n.count + sibling(n).count + parent(n).count$
    if $triadSum \leq \lfloor N/k \rfloor$ **then**
      $parent(n).count \leftarrow triadSum$
      $n.count \leftarrow 0$
      $sibling(n).count \leftarrow 0$
      DELETEIFEMPTYLEAF(q, n)
      DELETEIFEMPTYLEAF(q, sibling(n))
    end if
  end if
end if

these dependencies, we can distinguish between sections that demand strict sequential execution and those that can be safely parallelized.

While a comprehensive data dependency analysis could be extended to every function within the Q-Digest implementation, the breadth of such an analysis would exceed the purpose of this research. Consequently, we have focused our study of dependencies on the most crucial and fundamental part of the algorithm, namely the section that deals with determining the recursive path and synchronous allocation of nodes within the insertion function, as showed in Listing 1. This logic represents the most important path of Q-Digest, where structural dependencies between parent and child nodes, combined with contention on shared frequency counters, create the main bottleneck for efficient parallel execution.

```
1  void insert(struct QDigest *q, size_t key, unsigned
       int count, bool try_compress) {
2      // ... Universe Expansion logic ...
3      struct QDigestNode *prev = q->root;
4      struct QDigestNode *curr = prev;
5
6      while (lower_bound != upper_bound) {
7          size_t mid = lower_bound + (upper_bound -
       lower_bound) / 2;
8          prev = curr;
9          if (key <= mid) {
10             if (!curr->left) {
11                 struct QDigestNode *new_node =
       create_node(lower_bound, mid);
12                 prev->left = new_node;
13                 prev->left->parent = prev;
14                 (q->num_nodes)++;
15             }
16             curr = prev->left;
17             upper_bound = mid;
18         } else {
19             if (!curr->right) {
20                 struct QDigestNode *new_node =
       create_node(mid + 1, upper_bound);
21                 prev->right = new_node;
22                 prev->right->parent = prev;
23                 (q->num_nodes)++;
24             }
25             curr = prev->right;
26             lower_bound = mid + 1;
27         }
28     }
29     curr->count += count;
30     q->N += count;
31 }
```

Listing 1. Q-Digest insertion logic highlighting structural and flow dependencies.

The data dependencies for the proposed piece of code are presented in Table I. Starting from the recursive path determination within the `insert`[3] function, the primary data dependency that emerges is with the traversal pointer `curr`, as detailed in Table I. Specifically, throughout the `while` loop, there is a flow dependency due to the pointer chasing operation, as this variable is updated at the end of each iteration (line 16 or 25) and immediately read at the start of the next (line 8) to verify the existence of subsequent child nodes. In an uncontrolled parallelism scenario applied to this loop, this behavior would cause critical errors. For example, during iteration $k+1$, instead of reading the correct child node address determined at iteration k, the thread might access a stale or uninitialized pointer, thus diverging from the correct path in the tree. Unlike the independent calculations found in other algorithms, the path determination here is strictly sequential; the decision to branch left or right in one iteration (k) dictates the starting state of the next ($k + 1$).

Focusing on the synchronous allocation and metadata update logic reveals a higher number of data dependencies, primarily because the analyzed code modifies global state shared across the entire Q-Digest structure. For memory locations such as `num_nodes`, `N`, and `count`, there is a critical output

[3]See full function specification here: https://ettoremiglioranza1012.github.io/Q-Digest/qcore_8c.html

dependence issue. When multiple threads are used to perform concurrent insertions, a race condition arises for all these locations since they are shared resources. Consequently, two or more threads (T_1, T_2) could attempt to increment the global node counter or the total weight simultaneously. Without atomic protection or proper synchronization, this leads to "lost updates" where the final count is lower than the actual number of insertions, compromising the mathematical correctness of the digest. Furthermore, the `count` variable presents a specific consistency risk: without proper precautions, different threads terminating at the same leaf node would overwrite each other's frequency updates, rendering the digest's quantile accuracy unreliable. This vulnerability stems from the fact that the += operator is not a single atomic step at the hardware level, but rather a "Read-Modify-Write" sequence. For instance, if two threads concurrently attempt to increment a node's count from 10 to 11, both may read the initial value of 10 into their local CPU registers before either has completed the write-back operation. Consequently, both threads will independently calculate and write 11 to the same memory location, effectively erasing one insertion and causing a "lost update" that compromises the statistical integrity of the entire digest.

TABLE I
DATA DEPENDENCY ANALYSIS OF THE Q-DIGEST INSERTION FUNCTION.

Memory Location	Earlier Statement			Later Statement			Loop Carr.?	Kind of Dataflow
	Line	Iter.	Acc.	Line	Iter.	Acc.		
`curr`	16	k	write	8	$k+1$	read	yes	flow
`num_nodes`	14	k	write	14	$k+n$	write	yes	output
`count`	29	T_1	write	29	T_2	write	no	output
`N`	30	T_1	write	30	T_2	write	no	output

As analyzed, the Q-Digest insertion mechanism relies on a path-traversal (pointer-chasing) logic within a tree structure. Consequently, implementing a standard reduction mechanism is unfeasible due to the inherent flow dependency on the `curr` pointer, which precludes loop-level parallelization. While a `#pragma omp critical` directive could technically resolve the output dependencies, such a coarse-grained locking strategy would likely render the parallel implementation slower than the sequential version, as traversal would be serialized. Therefore, we conclude that standard OpenMP loop-reduction patterns are ill-suited for this specific pointer-chasing scenario.

Given these architectural constraints, the most effective path toward parallelization is the adoption of a *Data Privatization* strategy. In this model, each MPI rank processes a discrete subset of the input data to construct a local, independent Q-Digest structure. The global state is subsequently synchronized through an efficient doubling-tree reduction logic, which merges the private structures into a single canonical digest. The details of this implementation are discussed in the following subsection.

C. Parallelization with MPI

Our parallel approach leverages the MPI standard to facilitate robust execution on distributed-memory systems. MPI allows parallel processes to operate within independent memory spaces, thereby eliminating complexities related to shared-memory race conditions and cache coherency. However, this distributed nature necessitates explicit inter-process communication, where the overhead of data movement and synchronization must be carefully managed to ensure scalability.

Adopting a data-parallel paradigm, our algorithm partitions the input dataset D uniformly across N available MPI processes. As illustrated in Figure 1, each process P_i independently executes the serial Q-Digest construction algorithm on its assigned data partition, generating a local, partial Q-Digest summary. These partial summaries form intermediate representations of the local data distribution. Subsequently, these local structures are progressively merged via global reduction operations, orchestrated primarily by the master process (rank 0), to construct a single, comprehensive global Q-Digest that accurately represents the quantiles of the entire dataset.

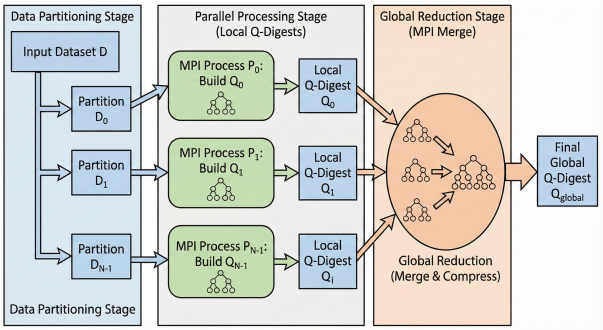

Fig. 1. Workflow of the parallel Q-Digest generation. The dataset is partitioned among parallel processes, which individually build local Q-Digest summaries. These partial summaries are subsequently reduced (merged) to form the final global Q-Digest.

Unlike the parallel file I/O approach utilized in comparable implementations, our implementation utilizes a centralized memory-to-memory distribution strategy via `MPI_Scatterv`. While simpler in initial setup, this design choice is grounded in a specific HPC consideration relevant to the Q-Digest algorithm. Our approach, which loads data on the Master node (Rank 0) and scatters it, serves as a functional proxy for a *Distributed Data Source* model.

In a real-world pipeline, data would most likely not originate from a single CSV file but would arrive via parallel streams across network interfaces or database shards. By scattering the data immediately upon initialization, we effectively mimic the system state *post-arrival*. This allows the performance analysis to focus on the true algorithmic bottleneck of the Q-Digest: the **Merge Phase**. The efficiency of reducing partial digests into a global summary remains the critical computational challenge, independent of the initial ingestion method.

To ensure uniform workload distribution across the cluster, our implementation addresses the case where the dataset size N is not perfectly divisible by the number of available processes P. We employ a dynamic displacement logic that calculates specific send counts and memory offsets for each

rank. The dataset is divided into a base quotient $Q = \lfloor N/P \rfloor$ and a remainder $R = N \bmod P$. To resolve the imbalance, the first R processes are assigned $Q + 1$ elements, while the remaining processes receive Q elements. This guarantees that the load difference between any two nodes never exceeds a single data point, optimizing processor utilization during the local digest construction phase. Algorithm 2 details the procedure used to calculate the *counts* and *displacements* vectors required for the vector variant of the MPI Scatter routine.

Algorithm 2 Data Distribution Logic

function DISTRIBUTEDATA(*global_data, N, P, r*)
{Calculate base load and remainder for balancing}
$base \leftarrow \lfloor N/P \rfloor$
$rest \leftarrow N \pmod P$
$counts \leftarrow$ array of size P
$displs \leftarrow$ array of size P
$offset \leftarrow 0$

for $i \leftarrow 0$ **to** $P - 1$ **do**
  $counts[i] \leftarrow base + (i < rest ? 1 : 0)$ {Handle non-perfect division}
  $displs[i] \leftarrow offset$
  $offset \leftarrow offset + counts[i]$ {Update displacement logic}
end for

$local\_n \leftarrow counts[r]$
$local\_buf \leftarrow$ MALLOC($local\_n \times$ sizeof(int))

if $r = 0$ **then**
  {Master node orchestrates the memory-to-memory scatter}
  MPI_Scatterv(*global_data, counts, displs, MPI_INT, local_buf, local_n, MPI_INT, 0, comm*)
else
  MPI_Scatterv(*NULL, counts, displs, MPI_INT, local_buf, local_n, MPI_INT, 0, comm*)
end if

return $local\_buf$

Once the local digest construction is complete, the parallel algorithm enters the synchronization phase to aggregate partial results into a unified global structure. Unlike centralized scheduling models that rely on a master node to coordinate every communication pair, our implementation employs a *Logarithmic Tree Reduction* strategy. In this model, the communication overhead is distributed across the cluster, allowing the system to scale with a complexity of $O(\log P)$, where P is the number of processes.

This phase is divided into two distinct logical steps: a preliminary trim of non-power-of-two processes and a subsequent balanced binary tree reduction. To maintain the mathematical symmetry of the binary tree, the algorithm first identifies "orphan" processes (i.e., those in excess of the largest power of two smaller than the total process count). Before the primary reduction loop begins, these orphans are paired with sibling processes to merge their data. This "trimming" step ensures that all remaining processes in the main reduction loop form a complete or full binary tree, preventing load imbalance or idle ranks in the later stages of the aggregation.

The core of the reduction utilizes a power-of-two bitwise logic to determine the communication topology. In each step k, active processes are partitioned into senders and receivers based on a step size of 2^k. A process remains active as a receiver if its rank is a multiple of the current step size; otherwise, it serializes its local state, transmits it to the corresponding receiver, and terminates its participation in the reduction. This decentralized approach ensures that as the reduction progresses the total number of active processes halves at each level.

Since the Q-Digest is a complex pointer-based hierarchical structure, it cannot be transmitted via standard MPI primitive types. We implement a *Serialization-Deserialization* pattern to facilitate inter-process communication:

- **Serialization:** The sender performs a pre-order traversal of its local digest, converting the nodes and their respective counts into a contiguous buffer of characters.
- **Integration:** The receiver, upon receiving the buffer, reconstructs the temporary digest and executes a formal merge operation. This operation ensures that the counts for identical ranges are summed and the resulting tree is re-compressed to maintain the specified error bounds (k).

Algorithm 3 illustrates the logical flow of this decentralized merging process.

D. PBS Directives and Cluster Execution

To execute the parallel Q-Digest implementation on the HPC cluster, we utilized the Portable Batch System (PBS) to manage resource allocation and task scheduling. The submission script utilized for these experiments is presented in Listing 2. Contrary to a standard distributed deployment, our resource request was deliberately configured to utilize a *single physical node* (`select=1`). This architectural decision was made in order to establish a baseline performance benchmark that focuses strictly on the algorithmic efficiency, by eliminating the latency and bandwidth bottlenecks associated with inter-node network communication.

```
#!/bin/bash
#PBS -N ${JOB_NAME}
#PBS -l select=1:ncpus=${c}:mem=1gb -l place=pack:excl
#PBS -l walltime=00:01:30
#PBS -q short_HPC4DS
```

Listing 2. MPI Q-Digest PBS Job Submission Script.

The configuration relies on specific PBS directives to ensure the rigorous reproducibility of the results:

- `place=pack:excl`: This directive is critical for our benchmarking accuracy. It forces the scheduler to *pack* the processes onto the selected node and grants *exclusive*

Algorithm 3 Q-Digest Tree Reduction

function TREEREDUCE(*q, comm_size, rank*)
{Phase 1: Balancing the Communicator}
$p2 \leftarrow$ FINDLARGESTPOWEROFTWO($comm\_size$)
$orphans \leftarrow comm\_size - p2$
if $rank < 2 \times orphans$ **then**
  if $rank$ is Odd **then**
    SERIALIZEANDSEND(*q, rank - 1*)
    return
  else
    $tmp \leftarrow$ RECEIVEANDDESERIALIZE(*rank + 1*)
    MERGE(*q, tmp*)
  end if
end if

{Phase 2: Logarithmic Aggregation}
$tree\_size \leftarrow p2$
for $k \leftarrow 0$ **to** $\lceil \log_2(tree\_size) \rceil - 1$ **do**
  $step \leftarrow 1 \ll k$
  if $rank\%(2 \times step) \neq 0$ **then**
    SERIALIZEANDSEND(*q, rank - step*)
    break {Process becomes inactive}
  else
    $sender \leftarrow rank + step$
    if $sender < tree\_size$ **then**
      $tmp \leftarrow$ RECEIVEANDDESERIALIZE(*sender*)
      MERGE(*q, tmp*)
      COMPRESSIFNEEDED(q)
    end if
  end if
end for

reservation of the allocated CPU cores. This prevents resource contention (noisy neighbors) and context switching overhead from other user jobs, ensuring that the measured execution times reflect the pure algorithmic performance.

- `-q short_HPC4DS`: The experiments were executed on a specialized queue created specifically for the "High Performance Computing for Data Science" course.
- `ncpus=${c}`: The number of MPI processes is dynamically injected at submission time, allowing us to go through all core counts (from 1 to 64) while maintaining the remaining environment variables constant.

E. Benchmark Datasets

To evaluate the performance, scalability, and numerical accuracy of both the serial and parallel Q-Digest implementations, we developed a specialized synthetic data generator. The Q-Digest evaluation requires workloads that stress the frequency-based compression of the underlying binary tree structure. To ensure the replicability of results across different high-performance computing (HPC) environments, the generator implements a deterministic seeding mechanism. The algorithms were evaluated using datasets with an increasing number of data samples. Specifically, five different artificial datasets were generated, comprising 1,000,000, 2,000,000, 4,000,000, 8,000,000, and 16,000,000 elements, respectively. For each generated set, the script computes the "ground truth" quantiles using the *NumPy* library, providing a reference baseline for calculating the ϵ-approximation error of our implementation.

```python
import argparse
import numpy as np
from pathlib import Path

# Supported distributions for stress-testing
valid_dists = {'gaussian', 'exponential', 'poisson', 'geometric'}

parser = argparse.ArgumentParser()
parser.add_argument("--seed", type=int, help="Random seed for reproducibility")
parser.add_argument("--dist", type=str, choices=valid_dists, help="Probability distribution")
parser.add_argument("--n", type=int, help="Size of the dataset")
parser.add_argument("--save", action='store_true', help="Write results to disk")

args = parser.parse_args()
generator = np.random.default_rng(seed=args.seed)

# Distribution-specific generation logic
if args.dist == 'gaussian':
    mean, sd = 10000, 1000
    result = generator.normal(mean, sd, size=args.n).astype(np.uint32)
    result = np.where(result < 0, 0, result)
elif args.dist == 'exponential':
    result = generator.exponential(size=args.n).astype(np.uint32)
# ... [additional distributions: poisson, geometric]

if args.save:
    np.savetxt(full_path, result.reshape(1,-1), delimiter=',', fmt='%d')
```

Listing 3. Python Script for Q-Digest Synthetic Dataset Generation.

```
$ python3 generate_dataset.py --seed 101 --n 1000000 --dist gaussian --save
```

Listing 4. Example of Benchmark Dataset Generation for Gaussian Distribution.

IV. EXPERIMENTAL EVALUATION

This section presents the empirical results of our parallel Q-Digest implementation. Our primary objective is to evaluate the performance and efficacy of the proposed parallel framework by examining key metrics that represent the standard in the field of HPC: *speedup* and *efficiency*. By comparing these results against a baseline serial implementation [2], we aim to provide an insightful analysis of the algorithm's operational strengths and weaknesses. This evaluation provides direct information into the overall effectiveness of the Q-Digest in approximating quantiles across large-scale, memory-constrained data streams.

A. Hardware

To evaluate the performance of our parallel Q-Digest implementation, we leveraged the High-Performance Computing

(HPC) cluster at the University of Trento, managed by the Altair PBS Professional software. The cluster provides a robust environment to test our proposed parallel algorithm, with the following primary specifications:

- **Operating System and Management:** Linux CentOS 7, managed via PBS Professional (version 19.1).
- **Compute Nodes:** The cluster consists of 142 CPU nodes (7,674 cores), 10 GPU nodes (48,128 CUDA cores), and 2 frontend nodes.
- **Performance:** The total theoretical peak performance is 478.1 TFLOPs (422.7 TFLOPs from CPUs and 55.4 TFLOPs from GPUs).
- **Memory:** 65 TB of total aggregate RAM.
- **Connectivity:** All nodes are interconnected via a 10 Gb/s network.

B. Experimental Setup and Parameters

The parallel implementation of the Q-Digest was subjected to rigorous testing using the synthetic datasets described in Section III-E, characterized by an increasing volume of data samples (1M to 16M elements). To evaluate the scalability and communication overhead of our solution, the algorithm was executed using six distinct core counts: 2, 4, 8, 16, 32, and 64. For all experimental runs, the compression parameter k was fixed at 200. This specific value was chosen to deliberately increase the depth and overall size of the resulting Q-Digest trees. In the context of a parallel implementation, a larger tree structure serves as a "stress test" for the MPI-based parallel reduction and merging algorithms. By increasing the volume of data that must be communicated and consolidated across nodes, we can more accurately evaluate the behavioral efficiency and latency of our reduction logic under heavy synchronization loads.

C. MPI Parallelization Results

Table II presents the comparative results between the serial and parallel implementations for each experimental run. The table correlates the number of processing cores allocated with the execution time required to generate the Q-Digest for datasets of varying magnitudes. To ensure the accuracy of these measurements, we adhered to standard high-performance computing timing protocols [4]. Specifically, we synchronized the MPI processes before starting the timer and, upon completion, identified the maximum elapsed time among all processes to represent the total parallel execution time. To account for system variability and OS-level jitter, the procedure detailed above was executed multiple times (specifically, 10 iterations for each case), and the minimum time from these runs was recorded [4].

```
//  ... LOCAL VARIABLES SET UP LOGIC ...
    // ============ Start timing ==============
    MPI_Barrier(MPI_COMM_WORLD);
    local_start = MPI_Wtime();

    distribute_data_array(
        buf,
        local_buf,
        counts,
        displs,
        local_n,
        rank,
        total_data_size,
        MPI_COMM_WORLD
    );

    if (rank == 0)
        free(buf);

    size_t local_upper_bound = _get_curr_upper_bound
    (local_buf, local_n);
    size_t global_upper_bound;
    MPI_Allreduce(&local_upper_bound, &
    global_upper_bound, 1, MPI_UNSIGNED_LONG,
    MPI_MAX, MPI_COMM_WORLD);

    struct QDigest *q = _build_q_from_vector(
    local_buf, local_n, global_upper_bound, K);

    tree_reduce(q, comm_sz, rank, MPI_COMM_WORLD);

    local_finish = MPI_Wtime();
    // ============ End timing ==============

    local_elapsed = local_finish - local_start;
    MPI_Reduce(&local_elapsed, &elapsed, 1,
    MPI_DOUBLE, MPI_MAX, 0, MPI_COMM_WORLD);
//  ... QUANTILE QUERY EXECUTION AND PRINT ...
```

Listing 5. MPI timing and data distribution logic.

TABLE II
EXECUTION TIME [S] FOR Q-DIGEST GENERATION.

Size	Number of Cores						
	1	**2**	**4**	**8**	**16**	**32**	**64**
1 M	3.018	1.552	0.831	0.512	0.316	0.217	0.251
2 M	5.935	3.028	1.576	0.926	0.531	0.348	0.393
4 M	11.790	5.978	3.067	1.749	0.731	0.596	0.612
8 M	23.524	11.859	6.034	3.428	1.390	1.095	0.984
16 M	46.929	23.587	11.993	6.763	2.429	2.156	1.774

The data reveals several pivotal characteristics of the parallel Q-Digest algorithm. Firstly, there is a consistent trend of performance improvement as hardware resources increase. This is most evident in the largest workload: for the 16 million element dataset, execution time drops dramatically from 46.93 seconds on a single core to just 1.77 seconds on 64 cores. This 26-fold reduction underscores the effectiveness of distributing the tree construction and merging phases across multiple nodes for extensive computational tasks. However, unlike an ideal linear speedup, the benefits of adding cores are subject to diminishing returns and, in specific cases, performance degradation. While the transition from 1 to 16 cores yields significant gains across all dataset sizes, further scaling to 32 and 64 cores reveals the impact of communication latency. Notably, for smaller datasets (1M and 2M), the execution time actually increases when moving from 32 to 64 cores (e.g., 0.217s to 0.251s for 1M). This discrepancy indicates that for lower data volumes, the overhead of MPI tree-reduction and inter-process synchronization begins to outweigh the benefits of parallel computation. This suggests an optimal range for core usage dependent on problem size; for datasets under 4 million elements, employing more than 32 cores becomes counter-

productive. Regarding dataset complexity, it is interesting to note that for a fixed number of cores, doubling the dataset size results in an almost exact doubling of execution time (e.g., 1M at 1 core takes ≈ 3.01s, while 2M takes ≈ 5.93s). In summary, the results highlight the significant scalability of the parallel Q-Digest algorithm for large-scale data analysis. While the algorithm demonstrates strong speedup characteristics, the observed overhead on smaller datasets emphasizes the importance of balancing resource allocation with the actual computational load.

D. Algorithm Evaluations

To rigorously assess the effectiveness of the parallel Q-Digest implementation, we extend our analysis beyond raw execution timestamps to derived metrics that quantify performance gains and resource utilization. Specifically, the following standard HPC metrics were employed to visualize the algorithmic trends:

- **Speedup (S):** Defined as the ratio between the execution time of the serial reference implementation (T_{serial}) and the parallel execution time on p cores ($T_{parallel}$):

$$S = \frac{T_{serial}}{T_{parallel}} \tag{3}$$

  Ideally, the speedup should grow linearly with the number of processing cores ($S \approx p$), indicating perfect parallel scaling.
- **Efficiency (E):** This metric normalizes the speedup against the hardware resources utilized, measuring how effectively the algorithm exploits the available cores (p):

$$E = \frac{S}{p} \tag{4}$$

  High efficiency values (close to 1.0) indicate optimal resource usage, whereas lower values suggest that communication overhead or load imbalance is dominating the runtime.
- **Scalability:** We evaluate the scalability of the solution through two distinct lenses:
  - *Strong Scalability:* Examines how efficiency behaves when the **problem size remains constant** while the number of cores increases. A system is strongly scalable if it maintains high efficiency as the computational load per core decreases.
  - *Weak Scalability:* Evaluates efficiency when the **workload per core remains constant**. This involves increasing the total problem size proportionally to the number of cores. If efficiency remains stable under these conditions, the algorithm is considered weakly scalable.

1) Speedup: Observing the graph in Figure 2, several key insights emerge regarding the scalability of the Q-Digest implementation. In general, there is a consistent trend of increasing speedup as the number of parallel processes grows, confirming that the partitioning strategy effectively distributes the computational load of the local quantile digestion. However, unlike the ideal linear progression, the behavior varies significantly depending on the dataset size. For the smallest datasets (1M and 2M elements), the speedup follows a parabolic trajectory. For the 1M dataset, performance peaks at 32 processes with a speedup factor of approximately $13.9\times$, but remarkably, it degrades to $12.0\times$ when scaled to 64 processes. A similar inversion is observed for the 2M dataset (dropping from $17.1\times$ to $15.1\times$). This phenomenon clearly illustrates the "communication wall": for small workloads, the granularity of the tasks becomes too fine, and the latency of the MPI tree-reduction phase overwhelms the diminishing returns of the local computation. Conversely, larger datasets exhibit a different and more robust behavior. Starting from the 4M dataset, we observe a distinctive *super-linear speedup* at 16 cores. For instance, the 16M dataset achieves a speedup of $19.3\times$ using only 16 cores. This efficiency boost is likely attributable to cache locality: partitioning a large dataset into smaller chunks allows the local working sets to fit entirely within the CPU caches, a benefit unavailable to the monolithic serial execution which likely suffers from cache thrashing.

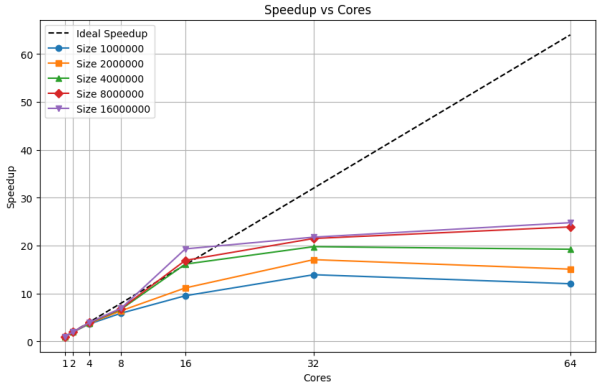

Fig. 2. Speedup Performance of the Parallel Execution.

Beyond the 16-core mark, the larger datasets (8M and 16M) continue to scale, albeit at a sub-linear rate, achieving maximum speedups of $23.9\times$ and $24.8\times$ on 32 and 64 cores, respectively. This indicates that while the communication overhead eventually begins to erode the computational gains, the algorithm remains effective for larger data streams. Consequently, these results highlight the importance of tailoring the parallelization degree to the workload size. For datasets under 4 million elements, employing more than 32 cores is counterproductive, whereas bigger datasets can effectively leverage the full extent of the HPC cluster's resources.

2) Efficiency: Turning our attention to the efficiency trends depicted in Figure 4, we observe a nuanced behavior that deviates from standard parallel scalability models. Initially, with a low degree of parallelism (2 and 4 cores), the efficiency remains exceptionally high (> 0.90) across all dataset sizes. This confirms that the MPI initialization and communication overheads are negligible when the cluster is lightly loaded. A divergence in behavior emerges as the core count increases

to 8 and 16. For smaller datasets (1M and 2M), we observe the expected monotonic decline. Specifically, for the 1 M dataset, efficiency drops to 0.60 at 16 cores and plummets to 0.19 at 64 cores. This sharp decay corroborates our previous "speedup analysis," indicating that for small workloads, the communication cost of the tree reduction dominates the runtime at high concurrency. However, for larger datasets ($N \geq 4$ M), we observe a remarkable *efficiency anomaly* at 16 cores. Instead of declining, the efficiency actually **increases**, surpassing the theoretical limit of 1.0 (e.g., $E_{16} \approx 1.21$ for the 16 M dataset). As previously hypothesized, this super-linear behavior suggests that at this specific granularity, the partitioned dataset fits optimally within the rapid L2/L3 caches of the CPUs, significantly accelerating the local digestion phase. Beyond this "sweet spot" of 16 cores, the efficiency degrades rapidly for all datasets. At 32 and 64 cores, even the largest datasets struggle to maintain efficient resource utilization (dropping to ≈ 0.39 for 16 M). This confirms that while the algorithm is scalable in terms of raw speedup, the marginal utility of adding resources diminishes sharply after 16 cores due to the logarithmic cost of the global reduction.

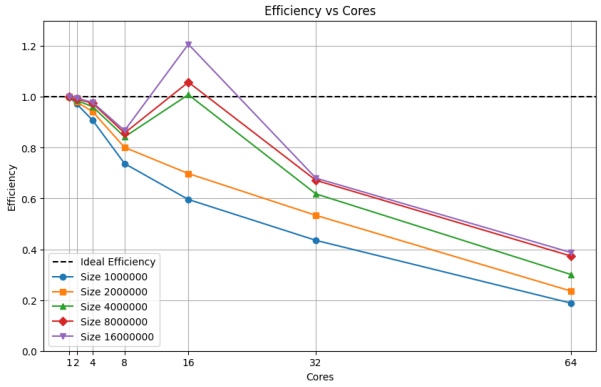

Fig. 3. Efficiency Performance of the Parallel Execution.

Therefore, the results suggest an **optimal parallelism range**. For small datasets ($<$ 4M), the optimal range is restricted to **2 to 4 cores**, beyond which resources are wasted. Conversely, for large datasets ($\geq$ 4M), the algorithm operates most efficiently at **16 cores**, leveraging cache effects.

3) Scalability: In addition to speedup and raw efficiency, we analyzed the scalability of the parallel Q-Digest algorithm through two distinct lenses: strong and weak scalability. The graph in Figure 4 inherently illustrates the strong scalability of our solution. By definition, strong scalability measures the system's ability to minimize time-to-solution for a **fixed problem size** as resources increase. Based on the experimental data, we structure our evaluation into three key observations:

1) **Baseline Stability ($P \leq 4$):** Across all dataset sizes, the algorithm maintains high efficiency ($> 90\%$) when using up to 4 cores. While positive, this is largely expected due to the low communication volume relative to computation at this scale and does not essentially confirm strong scalability for high-performance clusters.
2) **Global Degradation Trend:** A broader view reveals that as the number of processes increases beyond 8, efficiency progressively diminishes for nearly all configurations. This trend is most severe for smaller datasets (e.g., 1M elements), where efficiency collapses to $\approx 18\%$ at 64 cores. Since the algorithm cannot sustain high efficiency as resources increase indefinitely, we must conclude that the Q-Digest implementation does not exhibit global strong scalability.
3) **Hardware-Induced Exception:** A notable deviation occurs specifically at 16 cores for large datasets ($N \geq 4$ M), where we observe a *super-linear* efficiency spike (reaching $\approx 120\%$ for 16M). While this specific configuration technically satisfies the strong scalability criterion, once again we attribute this performance boost to hardware cache capacity rather than inherent algorithmic properties.

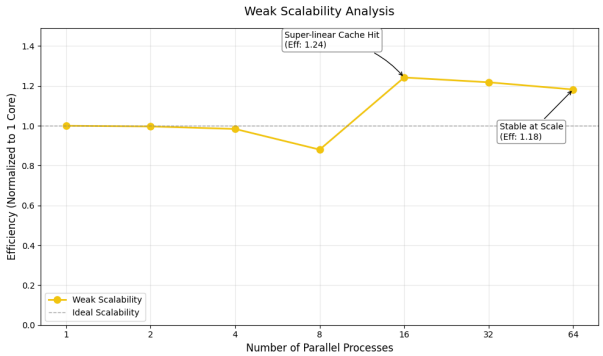

Fig. 4. Weak Scalability Analysis

To evaluate the weak scalability of the parallel Q-Digest implementation, we conducted a specialized series of experiments where the **workload per core remains constant**. Specifically, we analyzed the execution times (T) and the resulting Efficiency (E) for pairs (N, P) where the ratio is fixed at $N/P = 1,000,000$ elements per core. The efficiency is normalized against the baseline performance of a single core processing 1M elements ($T_{base} \approx 3.0179$ s). The results, detailed in Table III, present a sophisticated view of the system's performance across varying scales. The empirical data reveals a biphasic scaling trend. In the initial phase (1 to 8 cores), the system follows a traditional scaling curve where efficiency slightly degrades to 88.0% at 8 cores. This dip is characteristic of the increasing synchronization overhead in the binary reduction tree used for the merging phase. However, as the system scales to 16 cores and beyond, we observe a sustained **super-linear efficiency** ($> 100\%$). We again identify this hardware-driven phenomenon to be likely caused by the "Cache Locality Effect" and the increasing partitioning of the 10^6 workload per core.

While efficiency begins a gradual decline after the 16-core peak ($124.2\% \rightarrow 118.2\%$), it remains well above the ideal 100% mark. This suggests that the performance benefits gained from cache locality continue to outweigh the communication

TABLE III
EXTENDED WEAK SCALABILITY ANALYSIS ($N/P = 10^6$).

Cores (P)	**Dataset Size (N)**	**Time (s)**	**Efficiency (E)**
1	1 M	3.0179	100.0%
2	2 M	3.0280	99.7%
4	4 M	3.0667	98.4%
8	8 M	3.4279	88.0%
16	16 M	2.4292	**124.2%**
32	32 M	2.4773	**121.8%**
64	64 M	2.5530	**118.2%**

Note: Time measured in seconds. The super-linear efficiency from 16–64 cores indicates a significant cache-locality benefit.

overhead inherent in the MPI reduction tree, even at 64 cores. Notably, the efficiency keeps decreasing, indicating that the increasing communication overhead will eventually outgrow the architecture's benefits that are currently creating this high-performance boost.

V. CONCLUSION

This paper thoroughly explores, explains, and discusses a potential parallelization strategy using the MPI framework for the Q-Digest algorithm. Our goal was to leverage modern computational resources to achieve faster and more efficient approximate quantile estimation on massive numerical datasets. Our analysis demonstrates that the proposed parallelization approach significantly enhances the performance of the Q-Digest, ensuring excellent speedup relative to the serial implementation and promising weak scalability. Notably, we observed a *super-linear* efficiency improvement at specific configurations (16 cores), a behavior we attribute to hardware cache locality, where the partitioned dataset fits optimally within the CPU memory hierarchy. Although the algorithm does not provide global strong scalability and levels off when the communication overhead of the tree reduction outweighs the local computation, we believe our solution represents a valuable tool for analyzing distributed data streams where high precision and low memory footprint are required.

VI. FUTURE WORKS

In our opinion, the parallel version of the Q-Digest algorithm is already an effective tool for processing large-scale data. However, there are several areas for improvement and extension, particularly regarding the choice of parallel paradigm. One significant enhancement would be the benchmarking of a Hybrid $MPI + OpenMP$ approach. In Section III-B, we ruled out fine-grained multi-threading (threads inserting into a shared tree) due to the high complexity of managing race conditions during tree re-balancing. However, an OpenMP strategy, where each thread maintains a *thread-private* Q-Digest and merges them via a User Defined Reduction (UDR), has been implemented[4] and warrants rigorous testing. We then propose a two-phase roadmap to evaluate this:

1) **Intra-Node Efficiency (OpenMP vs. MPI):** The first step is to benchmark a pure OpenMP implementation against the current MPI baseline on a single node. We hypothesize that OpenMP threads, leveraging shared memory for the reduction phase, will outperform MPI processes by eliminating the overhead of serialization/deserialization and context switching. This could resolve the performance degradation observed at high core counts (32-64 cores) in our current results.
2) **Inter-Node Scalability (Hybrid Model):** Subsequently, the optimal configuration would likely be a hybrid architecture: using OpenMP for local parallelism within a node and MPI solely for inter-node communication. This drastically reduces the number of MPI ranks (from one per core to one per node), flattening the global reduction tree and minimizing network latency.

Furthermore, while our tests on datasets up to 16 million elements provided clear insights into CPU-bound performance, they do not fully stress the memory hierarchy of modern HPC nodes. Future benchmarks should target inputs in the order of billions of elements. Such scale is necessary to push the algorithm beyond the cache-resident regime, thereby testing its resilience against main-memory bandwidth saturation and providing a more realistic assessment of its throughput in data-intensive scenarios.

[4]https://github.com/ettoremiglioranza1012/Q-Digest/tree/main/omp-implementation

ACKNOWLEDGMENT

The current implementation uses portions of code that have been ported in C from a C++ implementation [5].

REFERENCES

[1] N. Ivkin, E. Liberty, K. Lang, Z. Karnin, and V. Braverman, "Streaming Quantiles Algorithms with Small Space and Update Time," *Sensors*, vol. 22, no. 24, p. 9612, Dec. 2022. [Online]. Available: https://www.mdpi.com/1424-8220/22/24/9612

[2] N. Shrivastava, C. Buragohain, D. Agrawal, and S. Suri, "Medians and beyond: new aggregation techniques for sensor networks," in *Proceedings of the 2nd international conference on Embedded networked sensor systems*. Baltimore MD USA: ACM, Nov. 2004, pp. 239–249. [Online]. Available: https://dl.acm.org/doi/10.1145/1031495.1031524

[3] L. Wang, G. Luo, K. Yi, and G. Cormode, "Quantiles over data streams: an experimental study," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. New York New York USA: ACM, Jun. 2013, pp. 737–748. [Online]. Available: https://dl.acm.org/doi/10.1145/2463676.2465312

[4] P. Pacheco, *An Introduction to Parallel Programming*. San Francisco, CA: Morgan Kaufmann, 2011.

[5] dhruvbird, "C++ Implementation of the Q-Digest," https://github.com/dhruvbird/q-digest, 2013.