Big Data Processing: Scalability with Extreme Single-Node Performance

Venkatraman Govindaraju, Sam Idicula, Sandeep Agrawal, Venkatanathan Vardarajan, Arun Raghavan, Jarod Wen, Cagri Balkesen, Georgios Giannikis, Nipun Agarwal, Eric Sedlar
Oracle Labs
{venkat.govindaraju, sam.idicula, sandeep.r.agrawal, venkatanathan.varadarajan}@oracle.com
{arun.r.raghavan, jarod.wen, cagri.balkesen, georgios.giannikis, nipun.agarwal, eric.sedlar}@oracle.com

Abstract—Contemporary frameworks for data analytics, such as Hadoop, Spark, and Flink seek to allow applications to scale performance flexibly by adding hardware nodes. However, we find that when the computation on each individual node is optimized, peripheral activities such as creating data partitions, messaging and synchronizing between nodes diminish the speedup obtainable from adding more hardware. We analyze workloads which distribute operations on correlated data—such as joins and aggregation found in SQL, text similarity searches, and image disparity computations. After optimizing computation on efficient, custom processors, we discover challenges in scaling the applications to hundreds of nodes on a high-bandwidth network. We then describe techniques to overcome these challenges towards prototyping a 512-node system which is able to execute SQL queries offloaded from a commercial database, and outperform SQL-on-hadoop and traditional parallel RDBMS executions by $173\times$ and $7\times$ respectively.

Keywords—bigdata, scalability, dynamic network scheduling, shuffle

I. INTRODUCTION

An increasing number of applications today benefit from scanning volumes which transcend several terabytes—sometimes approaching petabytes of data. To cater to such applications, recent systems tend to first partition data across a large cluster of nodes, then distribute computation amongst the individual nodes [1]. Distributed frameworks such as MapReduce [2], Spark [3] and Flink [4] are able to cluster hundreds, even thousands of nodes with impressively scalable performance. However, with careful optimization, much smaller engineered systems are often able to outperform such large, commodity installations [5], [6].

Furthermore, when scaling well optimized software to multiple nodes, latency in creating and orchestrating parallel processing—such as messaging, synchronization and partitioning—starts to impede performance. Although such operations are fundamental to all distributed systems, their overheads are largely hidden behind the slow computation cycles in popular distributed frameworks like MapReduce and others. As a result, adding more hardware to a suboptimal baseline manifests as scalable performance [7]. However, with a well-tuned single-node implementation, we find that naïvely adding hardware resources yields diminishing returns even with increasing workload inputs (i.e., weak scaling).

In this paper, we examine key causes for inefficient scaling on a variety of data analytics workloads including TPC-H queries, text and image similarity searches. We first tuned single node performance on an extremely low-power hardware platform where highly optimized software compensates for lack of CPU features, leaving few compute cycles to spare. We then experiment with these workloads at the scale of hundreds of nodes, identify key bottlenecks to scaling, and evaluate solutions which enable near-linear scalability over a high-performance single node baseline. (For lack of space, we focus on scalability and not single-node optimizations in this paper).

We find that the underlying factors which prevent good scaling arise due to data distribution, communication and aggregation inefficiencies in the big data processing systems. We then systematically explore solutions to mitigate each overhead: (i) Workload-aware partitioning helps reduce delays due to repartitioning, while heavy hitter awareness prevents skew from specific data trends. (ii) Topology aware dynamic scheduling of messages among the nodes reduces network congestion and delays. (iii) Dynamic, adaptive aggregation networks enable workloads that require high correlation among tasks running on multiple nodes to scale. The rest of this paper:

- describes and quantifies the scaling overheads that start to dominate when a single node implementation is heavily optimized. (Section II)
- presents scaling techniques to mitigate the effect of the overheads on overall system performance. (Section III)
- adapts these techniques to analytic operations from a variety of domains and show how these techniques can be used to scale the system well. (Section IV)

II. OVERHEADS IN BIG DATA PROCESSING

A typical big data workload is characterized by the following steps that aim to answer a given query. First, the input data required for answering the query is distributed across a cluster of nodes. This is followed by one or more stages of parallel computation and synchronization across the cluster. The process finally culminates with the desired result of the query. With increasing data sizes, scaling out to more nodes is essential to satisfy query SLAs, giving rise to many distributed big data platforms like Apache Spark. Although existing distributed systems built on an open source stack have been shown to scale well [8], they fail to compare well against highly optimized single node implementations [7]. In this section, we aim to quantify the various overheads that hinder scalability of a big data application.

A. Data Distribution

In a scale-out distributed data processing system, data is usually partitioned or sharded across the cluster, enabling
horizontal scaling across multiple nodes. For example, in a distributed database system, tables are split by rows across nodes and stored as partitioned relations. In addition to the initial partitioning step, most queries may require a repartition of any intermediate results, which when poorly handled may result in poor overall system performance [9]. Partitioning the data incurs significant overheads when scaling to a large number of nodes (> 64), depending on the type of query and operation [10]. Specifically, we identify three primary causes of partitioning overheads: (i) buffer management for each partition, (ii) repartitioning of data later in query execution, and (iii) data skew.

A naïve $N$-way partitioning across all $N$ nodes quickly becomes a bottleneck where the partitioning node needs to maintain a large number of buffers (at least one per partition). These buffers are typically used to batch partitions, amortizing network costs. A quick experiment on a table with 256M rows and 4 columns shows that this approach may incur a slowdown as high as $2.5\times$ relative to a single node run, which copies in the data once. Figure 1 shows the result of this naïve approach along with results from our architecture-aware algorithm presented in Section III-A. The partitioning process inherently has poor data-locality, especially when managing large number of buffers resulting in high cache and Translation Lookaside Buffer (TLB) miss rates.

The initial partitioning for data placement requires to be carefully orchestrated to handle any further repartitioning during query execution. For example, when partitioning a table, records in the table are distributed among nodes using a hash function on a key column. If a join is executed with the key column, the query can be distributed and independently executed on each node. However, if the join is executed using a different column, the data needs to be repartitioned in an expensive shuffling operation, causing every node to communicate with all other nodes in the system.

Aside from the direct overheads due to partitioning, data skew, a common problem in database applications, often hinders scalability as well. Partitioning the data with incorrect assumptions about the data distribution may introduce load imbalances in a distributed system, degrading performance and scalability [11]. In Section III-A, we present specific techniques to tackle each of these issues when distributing the data over a large cluster.

---

**B. Data Communication**

A common big data communication pattern is an all-to-all shuffle across nodes in the cluster [2], [3], [12], [13]. Certain big data operations such as join, sort and aggregation need to re-distribute intermediate results across all nodes in the system with an optional re-partitioning step. Map-reduce based systems also perform a shuffle operation before the reduce stage. These data shuffle operations are often considered an optimization aimed to fully utilize the network in a small cluster. At larger scales, a poorly orchestrated shuffle introduces new overheads that affect overall scalability especially when compared to a highly-optimized single-node implementation. These overheads are primarily caused due to: (i) congestion at the receiver, (ii) network topology specific congestion at choke-points and (iii) non-negligible time to first byte (or network latency).

To quantify the effect of these overheads, we ran the characteristic shuffle operation on an increasing number of nodes and measured the network bandwidth on a high bandwidth Infiniband network. Figure 2 shows the observed bandwidth on a cluster of x86 nodes with varying message sizes (64KB, 128KB and 512KB). Note that the network bandwidth corresponding to the two node run represents the peak achievable bandwidth of the system. As the number of communicating nodes increase, the observed bandwidth drops to more than half of this peak bandwidth at 64 nodes. Even sending a larger message (512KB) to amortize costs does not improve bandwidth utilization. This poor scalability is primarily caused due to congestion at the receiving nodes and the network switch choke points. Network contention causes back-pressure at the sending nodes, and bandwidth drops to a fraction of the peak. In Section III-B, we present techniques to overcome these overheads by making the nodes coordinate with each other while sending messages across the network.

**C. Data Aggregation**

Many big data workloads operate in stages and may require a final result aggregation phase that requires network synchronization across multiple nodes. For example, tasks such as aggregation and top-k need a merge operation on results scattered across all nodes, resulting in an all-to-one communication pattern where all nodes try to send results back to a master node. A typical scalable approach for this type
of communication is implemented using a distributed merge-tree network (as shown in Figure 3). Contrary to conventional wisdom, the merge-tree based data aggregation may not scale well over a large-scale cluster for all types of workloads. We run a machine vision task where two pixel maps (of size > 8MB) are aggregated using a min operation per pixel. We found that even with an 8-way wide merge-tree, the aggregation time scaled poorly with a large number of nodes.

Figure 4 shows the result of this merge-tree aggregation network. A major contributor to this overhead comes from the non-negligible time to the first byte. These latency overheads are further multiplied by a static merge-tree configuration, resulting in additional levels of merge with an increasing number of nodes (esp. > 16). Poor overlap of aggregation compute with network latency for large results (in order of MBs) also degrades scalability. In Section III-B3, we present an adaptive aggregation technique to mitigate the effects of an all-to-one communication pattern, by switching between a shuffle network and a k-ary merge-tree network schedule based on the aggregation characteristics.

III. Scaling Techniques

With a highly optimized single node implementation, the onus falls on the distributed runtime to invest in techniques to hide if not reduce the relatively high overheads mentioned in the previous section. We present a novel combination of techniques that we found essential to scale a variety of big data workloads in our setting. Although a few of the techniques are inspired or borrowed from prior work, we highlight our novel contributions in their respective sections.

A. Partition Oriented Design

Partitioning is a core aspect of any big data application [11], [14], and we design our system with partitioning as a first class operation. Partitioning enables parallelism among the nodes with limited synchronization, improving the overall performance of the system. For instance, a database should not only store the base tables in a partitioned fashion, but also allow repartitioning of the intermediate relations generated, allowing subsequent operations to be performed with less synchronization and coordination.

1) Multi-round Partitioning: Naïvely splitting data into a large number of partitions incurs significant overheads (Section II-A). To mitigate these overheads, we partition data in two phases starting with an \( M \)-way partitioning, followed by one or more rounds of parallel \( N \)-way partitioning. We adaptively pick the sizes \( M \) and \( N \) based on resources available per node (e.g., cache size) and the number of partitionings to be created. Each round of the partitioning effectively utilizes the cache-hierarchy and the memory subsystem, requiring only negligible bookkeeping. We achieve this by starting with a minimum memory allocation for each of the above structures, and then allocate the rest of the space incrementally using a greedy approach based on the observed performance gains of allocating extra space. Typically, performance benefits for input/output buffers saturate at 0.5 KB and hence a large part the cache space is allocated to the hash table. For our architecture, we fix \( M \) to 32 based on the number of physical cores (32) available per node. \( N \) is further computed based on the number of partitions required (\( N = 32 \) for 1K partitions). Figure 1 shows the relative speedup of our multi-round repartitioning in comparison with the traditional single round partitioning approach. Our algorithm performs and scales significantly better, especially with a large number of partitions. Note that although the idea of multi-round partitioning is not new [10], [15], our adaptation to the available number of cores is essential for scalability.

2) Workload-aware Data Placement: Any partitioning scheme should be efficient and scalable, but also account for overheads caused by subsequent repartitioning. For instance, queries can have multiple joins involving different key columns. Instead of picking any one key for initial partitioning and allowing subsequent unmanaged repartitions, we analyze the query plans, extracting appropriate key columns that primarily minimize network traffic during any repartition. We achieve this with a set of heuristics that prioritize large joins based on table sizes and the join key distribution information. In addition, we also place the partitions on nodes aware of the network topology in the cluster. We analyzed the TPC-H workload and simulated the benefits of such a workload-aware data placement and partitioning scheme (shown in Figure 5). Although many prior works optimize partitioning for SQL join queries based on table sizes, the idea of analyzing any bigdata workload’s query plans to optimize for network traffic across a large cluster is one of its kind.

3) Skew-aware Partitioning: We build an approximate hash table [11] to dynamically detect any data skew during the partitioning phase. Based on the skew, data is then repa-
titioned using range partitioning or specific skew mitigating techniques. To handle heavy hitter skew, we use a round-robin distribution across a set of nodes that cooperatively further partition the data [11]. We improve scalability by dynamically triggering these techniques for larger number of nodes in the cluster.

4) Tiled Data Layouts: In addition to the above data partitioning techniques for scalability, we also use specialized data layout when partitioning the data to enable efficient data processing. For instance, when partitioning a relation across rows, instead of naively storing them in row- (or column-) major order, we use a hybrid row-columnar memory layout that is optimized for any query specific memory access patterns. For example, when partitioning an image for machine vision workloads, we use a hybrid row-columnar layout for each submatrix (or tile) that is optimized for that workload. Such tile-oriented layouts efficiently utilize the single node’s cache and memory bandwidth improving system runtime and also helps with scalable result aggregation (Section III-B3). Although such tile-oriented layouts have been used for matrix-multiplication [16] (which we rely on), we uniquely exploit it for image processing workloads.

B. Dynamic Network Scheduling

Most of the techniques that are essential for scalability focus on reducing the network overheads. Specifically, we achieve this by dynamically scheduling network messages to hide network latencies behind useful computation, reduce contention on shared network structures and increase overall network bandwidth utilization. We believe all the following techniques are novel contributions essential for scaling across a large cluster of highly optimized nodes.

1) Task Graph Scheduling: Network latency can cause significant bottlenecks for workloads relying on many small messages. We improve performance by hiding any or all of this network latency behind some useful computation. All algorithms can be decomposed into multiple small and simple tasks organized into a task graph with data and control dependencies. We overlap communication and computation based on two key observations, the scheduler must start network communication as soon as possible, and tasks closer to the root in the task graph have fewer overlap candidates. This allows us to identify a novel set of heuristics to schedule a task.

First, we prefer tasks for which dependent communication has completed. Then, we prefer tasks that lead to network traffic sooner. Finally, we prefer tasks that are farther from the root of the task graph. With these heuristics we were able to hide a significant fraction of the network cost (≈ 56% in TPC-H Q3) behind compute, if not all.

2) Mitigating Network Congestion: Network utilization can be improved by coordinating communication between nodes to avoid contention at the receiver and at any potential choke points dynamically induced by network flows. We mitigate this congestion by ensuring the following two conditions: (i) no two nodes send data to a common receiver simultaneously, and (ii) no two nodes sending data to different receivers may share a single link in the switch fabric (called inter-switch link). For instance, in our Infiniband fabric with a two-level fat-tree topology, two distinct flows can contend for an inter-switch link connecting the leaf and spine switches. This contention for link bandwidth can be eliminated by using round-robin or token based scheduling of network messages, depending on the workload communication characteristics.

Round-robin Scheduling: Senders take turns to send messages to a particular receiver and synchronize among themselves by using a barrier or sending these messages in a pre-determined time slice. Although this approach incurs a non-negligible network synchronization cost, it can be easily amortized for large data transfers.

Topology Aware Dynamic Scheduling: Each sender picks a candidate receiver based on the network topology and a list of receivers that it wishes to communicate with. A sender uses a sequence of lock, grant and release messages to coordinate with a receiving node, and the receiving node grants or rejects the request based on-going data transfers. On a successful lock request, the sender transfers data up to a configured limit per lock. A larger data limit per lock amortizes the overhead of lock acquisition, whereas a smaller limit prevents starvation for other nodes. We use heuristics to prioritize candidate nodes based on the network topology and graceful handling of the rejection message. We also allow lock elision to reduce lock acquisition and release overheads when sending a large amount of data.

To compare these two scheduling mechanisms, we repeat the all-to-all shuffle experiment described in Section II-B on 36 and 54 nodes (result shown in Figure 6). Both these techniques...
sustain peak bandwidth (6 GB/s) at higher node counts. Round-robin scheduling allocates equal time for all nodes to communicate, and achieves peak bandwidth on uniformly distributed data. On the other hand, token based dynamic scheduling dynamically adapts to the workload, and is able to sustain peak bandwidth on both uniform and skewed data.

3) Adaptive Result Aggregation: Achieving scalable result aggregation over the network is often dependent on the workload and type of aggregation. The commonly used merge-tree aggregation network does not always scale well over a large number of nodes (Section II-C). We identify a novel aggregation network that scales better for aggregation with large output data. Our approach dynamically switches at runtime between different aggregation networks based on the aggregation characteristics. We efficiently handle large results and hide their high transfer costs behind aggregation compute, improving overall bandwidth.

Handling Large Results: Most aggregation networks focus on reduction, with smaller messages exchanged amongst nodes as we move up the tree. However, for certain workloads, final results during aggregation can be large (relative to network buffers, in MBs), requiring special techniques to hide the network communication overheads. First, large results destined to a single node were broken down into multiple smaller chunks which can be aggregated independently. This increases the chances of overlapping network latency over compute. Second, we tightly coupled the application with the network stack where the network buffers were passed to the application layer avoiding multiple memory copies for network buffer management. Third, we avoided de-serialization of messages by doing in-place aggregation of incoming chunks into pre-allocated per-node aggregation buffers. Finally, we use a work-stealing mechanism with one receiving worker and several aggregation workers that in parallel aggregate two corresponding chunks in a shared workqueue. Here each worker is pinned to a distinct core on that node, avoiding any overheads due to context-switching or interruptions from other workers.

Shuffle-Merge Network: We use a single-stage shuffle aggregation network (shown in Figure 7) as an alternative to the tree-merge network (Figure 3). Here, the aggregation result is repartitioned with one partition per node and each node receives and in parallel aggregates its own partition. The final result is aggregated on a master node. We dynamically detect the type of aggregation based on observed local aggregation characteristics and coordinate the result aggregation.

The benefits of the two networks is apparent after we formalize the cost of distributed result aggregation. The cost of result aggregation for tree and shuffle networks respectively:

\[ C_T = \log_{w_i} N \times (t_{fb} + w_i \times (C(s^i) + T(s^i))) \]

\[ C_S = (N - 1) \times (C(s^i) + T(s^i)) + t_{fb} + (N - 1) \times T(s^0) + t_{fb} \]

Here, N is the number of nodes in the cluster, t_{fb} is time to first byte and w_i is the width of the tree-network. The functions C(s) and T(s) characterize the aggregation compute and transfer time based on the size s of the intermediate and final output result. To demonstrate the benefits of the shuffle aggregation network, we compared it with the scalability of the merge-tree network for the same machine vision workload (workload details in Section IV-C) using the above analytical model. Scaling over several nodes shows that the shuffle network performs significantly better than tree-based network (result shown in Figure 4). This network scheduling choice requires dynamic adaptation based on the characteristics of the aggregation and we utilize this technique in achieving near-linear scalability in our big-data applications.

IV. APPLICATIONS

In this section, we show how the techniques presented above help to scale applications even when the single node baseline is heavily optimized. We evaluate these applications on a cluster of 512 low power diskless nodes connected by a high bandwidth InfiniBand network. Each node has a custom ultra low power processor running at 800MHz with 8GB of main memory and uses the network to perform input/output operations. Furthermore, we run all our experiments in bare metal to eliminate any overheads due to operating system and other software.

A. SQL Analytics

SQL analytics is a popular way to answer a variety of business problems since it can access a wide range features available in database management systems, thereby providing deeper insights from operational data. Typical bigdata systems analyze the queries written in SQL and break down each query into a sequence of operations such as Filter, Projections, Join, aggregations, etc. In order to scale SQL analytics for large number of nodes, most if not all such operations should be scalable to large number of nodes. In this section, we describe how the techniques presented above in Section III help to scale SQL analytics on a large cluster. We use TPC-H queries to benchmark scalability of our system.

1) Query Compiler: Our query compiler breaks the query into multiple tasks and represent them in an acyclic graph. The goal of the query compiler is to schedule (both during compilation and runtime) the tasks aimed at minimizing the cost of moving data, maximizing the overlap between computation and communication (Section III-B1). It also sets the arguments for query execution engine for a cluster-, workload- and skew-aware partitioning as described in Section III-A1.
2) **Scaling SQL Operations**: The filter and projection operations operate on one or more columns and generate one or more output columns. Since the data is partitioned (or sharded in SQL parlance) across the nodes, each node independently operates on its partition and generates the output column. We use in-memory meta-data and light weight synchronization to limit the parallelization overheads during these operations. However, for join and aggregation, just data placement is insufficient.

In order to scale join over a large cluster, we utilize the partitioning oriented design throughout our query processing engine. We partition the relations with cluster-aware partitioning as described in Section III-A. We implemented a variant of hash join that uses the cluster architecture parameters to correctly pick the size of the partitioning and the number of rounds to partition. For queries involving a join or any aggregation, we apply the topology aware dynamic scheduling of network message to improve the utilization of network bandwidth (Section III-B2). We found that a limit of 1 MB data per lock is sufficient for most SQL queries. Finally, the hash table for partitioning is sized to ensure it does not exceed the cache size. With these techniques, we achieved a near-linear scaling of the join operation over 512 nodes.

3) **Scaling Group-by**: Group-by consists of aggregating rows based on the values of certain columns or more generally, expressions and then calculating aggregates like sum and count of a given list of expressions within each group. It can be efficiently processed using a hash table as long as the number of distinct groups is small enough [17]. Since the access pattern is random and the hash table size grows linearly with the number of distinct groups, ensuring locality of access is very important for performance. However, there is no guarantee that the hash table will be cache-resident and a significant performance penalty is inevitable when it happens.

The techniques targetted towards a partition oriented design like fitting the hashtable in on-chip caches, which also guarantee low access latency to the hashtable (as described in Section III-A1) and help alleviate these overheads for groupby. The execution engine re-partitions the data if and when a partition is larger than the query compiler’s estimates. Finally, the task graph execution engine executes the operators to maximize overlap between network (Section III-B).

We evaluate groupby for two cases: low number of distinct values (Low-NDV) and high number of distinct values (High-NDV). In the Low-NDV case, aggregation can trivially scale since the aggregated data is small. However, in the high-NDV case, our techniques help the groupby to scale to large number of nodes. On top of partitioning across nodes, we also partition locally within a node to parallelize a SQL operation among the cores in a node. Each core can operate independently on a separate (local) partition. But when the number of distinct groups is low, partitioning is not beneficial; in this case, the input data is equally distributed among the cores and a merge operator is added to the query plan after the groupby operator. Since the merge operator only works on aggregated data, its overhead is very low.

4) **TPC-H Scalability**: Figure 8 shows the scalability of the TPC-H query from 2 nodes to 512 nodes compared to a 2 node system. The scalability metric is calculated using,

\[
SM = \left(\frac{t_0}{tn}\right)^{\frac{1}{\log t - \log t_0}}
\]

where \(t_0\) is the time taken to run the queries in a two node system and \(n_0 = 2\). The input size is also increased as the number of nodes increasing. The 512 node system operates on a 1TB input data set (TPC-H scaling factor of 1024). We are able to achieve near linear scalability (an average of 0.9 with 1 being linear) over a very large cluster of 512 nodes. Moreover, we outperform both SQL-on-Hadoop [18] and the top ranked parallel RDBMS system on a cluster [19] by 173\times and 7.1\times respectively. This is possible because of the several techniques described earlier, primarily: analyzing input tables and query plans during partitioning, data placement to reduce aggregate network bandwidth utilized during execution (Section III-A2), and careful orchestration of network messages to hide network overheads (Section III-B).

B. **Similarity Search**

The similarity search problem involves finding the \(k\) most similar documents in a given corpus of documents for a given query. These documents can be web pages, images or text files, and the corpus usually contains billions of such documents. Many similarity search queries have realtime constraints, and an exhaustive search across a billion large corpus requires an efficient and scalable solution. We search across the English Wikipedia (4M pages) with page titles as queries similar to [20], using the \(tf-idf\) score to map text documents to a feature space. The text similarity search problem is formulated as a Sparse Matrix Matrix Multiplication (\(C = A \times B\)), followed by a max-heap based top\(K\) implementation.

We distribute \(B\) across the cluster by partitioning across columns (Section III-A4). \(B\) is stored as an inverted index, therefore each node manages a shard of the corpus. Nodes compute top\(K\) results for their shards, and these results are aggregated by a master for each query. A naive implementation would repartition this shard across cores on a node accelerating per query performance. This creates a bottleneck at the top\(K\) phase during aggregation. We optimize this by exploiting query level parallelism, mapping queries to cores on a node.
Each core processes a query on an entire shard, and sends its topK results to a master node, where the corresponding core merges these shard results. We also batch queries on a core, filling a single network packet to the maximum extent, minimizing communication overheads.

Figure 9 shows the weak scalability achieved for 32 cores on a machine, with a batch size of 256 queries/core and k = 32, creating a network packet size of 64KB (256 queries × k × 8 bytes per value). We initially scaled out using a single master, using the approach described above. This gives us a weak scalability of 0.98 at 32 nodes. However, going beyond 32 nodes causes a severe degradation in scalability (0.5 at 64 nodes), due to a network bottleneck created at the master (32 cores × 64 nodes = 2048 messages received for each query). We create multiple masters to handle this, with core i on each node sending the shard results for a query batch to master i. Each master then splits a batch across its local cores, computing the final topK results. This allows us to scale to more than 256 nodes with a weak scalability of 0.94 at 256 nodes.

C. Image Disparity

Image disparity computation is an essential part of many machine vision applications like object and feature tracking, etc. The disparity computation consists of running four distinct image kernels repeatedly on a stereo image (left and right). The goal of these kernels is to identify how far pixels representing the same object moved in one image relative to the other. The final result of this computation is a disparity map. The disparity computation is represented as a flow-chart in Figure 10.

Parallelizing the disparity computation on a single node is well-known [21], where typically a large image is partitioned into tiles where each core independently works on its own tile and synchronizes results using shared memory. We call this approach as fine-grained parallelism. It immediately becomes challenging to do the same at rack-scale where a large number of nodes cooperatively engage in computing disparity of a large high-quality image. Specifically, it involves two main challenges: (i) work distribution, and (ii) scalable result aggregation.

**Hybrid Work Distribution.** Continuing with the fine-grained parallelism used within a single node across a whole rack is inefficient as any rack-wide synchronization and shared memory implementations incur high overheads. This calls for a hybrid approach with a coarse-grained work distribution across nodes and retaining the fine-grained parallelism (partitioning images into tiles) within a node.

In disparity, we achieve this by assigning each node to compute disparity for a separate shift out of all shifts from 0 to \( \text{max\_shift} \) (refer Figure 10) where at each node the four image kernels are parallelized across multiple-cores in a node using fine-grained parallelism. This results in each node parallelly computing a disparity map for a distinct shift with zero communication or synchronization overhead. The only step left is to aggregate all the disparity map into a single node and find the minimum disparity over all shifts. This brings us to the second challenge.

**Scalable Result Aggregation.** For the result aggregation we use the adaptive result aggregation techniques described in Section III-B3. As the image disparity is a characteristic example of a non-diminishing aggregation problem where pixelwise minimum is the aggregation function over two disparity maps, it enjoys the benefits of the adaptive result aggregation technique. Further, the runtime optimized for large messages (e.g. HD image size is 8MB) was also critical for its scalability.

The result of these techniques resulted in a near linear speedup of disparity with increasing number of nodes. The results for weak-scalability run shown in Figure 11. It compares the scalability of disparity runtime with the two aggregation networks and shows that our shuffle aggregation network is more scalable than the merge-tree aggregation network.

V. RELATED WORK

A majority of the related work in the big data domain focuses on techniques to improve algorithm performance. In contrast, we study the bottlenecks caused when scaling out to hundreds/thousands of nodes, once algorithm performance
has been optimized. There are extensive studies on overheads in the partitioning stage of joins on a single node [10], [22], [11], [15]. Manegold et al [10], [22] proposed a fixed partitioning algorithm based on TLB entries to eliminate TLB misses. Similar to our approach, Kim et al. [15] rely on a multi-round partitioning scheme to reduce cache misses in a multi-core aware hash-join algorithm. Recently, Balkesen et al. demonstrated that an architecture aware join algorithm performs better than hardware-oblivious approaches [23].

A number of studies have identified all-to-all communication or shuffle as a major bottleneck in big data processing systems [12], [13]. These studies try to mitigate the bottleneck by introducing new schemes for shuffling. There are several proposals to improve all-to-all communication in networks, both in hardware [1], [24] and software [25], [26], [27]. These algorithms either compute an optimal schedule [25] or a fixed scheduling policy [26] that minimizes communication time or uses heuristics to compute optimal schedules. These approaches require a central coordinator and apriori knowledge about the data distribution for the communication. By contrast, our topology aware dynamic scheduling does not require either and allows scaling to a larger number of nodes. Jalaparti et al. [27] use the characteristics of workloads to determine an offline schedule to improve the performance of data processing tasks. Our approach dynamically adapts to the current data traffic and improves performance with minimal implementation overheads, successfully dealing with both data and temporal skew.

VI. CONCLUSION

To speed up applications which process large volumes of data across hundreds of nodes, execution plans must account for delays in distributing data alongside performing computations on each node. When partitioning data amongst independent nodes, ensuring locality of references, by hierarchically localizing address ranges in each node (e.g., sizing hash tables to be cache and memory resident) is a consideration along with recognizing skewed data distributions. When redistributing data amongst nodes, scheduling transfers without over-subscribing links maximizes speed of movement. Overall, overlapping computation, memory and network communication reduces delays towards maximizing system utilization, and hence performance. On a range of example distributed applications spanning TPCH queries, text and image processing operations, such optimizations allow us to achieve order-of-magnitude performance improvements over comparable, commodity baselines.

REFERENCES