Distributed and Parallel Databases
https://doi.org/10.1007/s10619-019-07257-y

Scaling sparse matrix-matrix multiplication in the accumulo database

Gunduz Vehbi Demirci[1] · **Cevdet Aykanat**[1]

Abstract

We propose and implement a sparse matrix-matrix multiplication (SpGEMM) algorithm running on top of Accumulo's iterator framework which enables high performance distributed parallelism. The proposed algorithm provides write-locality while ingesting the output matrix back to database via utilizing row-by-row parallel SpGEMM. The proposed solution also alleviates scanning of input matrices multiple times by making use of Accumulo's batch scanning capability which is used for accessing multiple ranges of key-value pairs in parallel. Even though the use of batch-scanning introduces some latency overheads, these overheads are alleviated by the proposed solution and by using node-level parallelism structures. We also propose a matrix partitioning scheme which reduces the total communication volume and provides a balance of workload among servers. The results of extensive experiments performed on both real-world and synthetic sparse matrices show that the proposed algorithm scales significantly better than the outer-product parallel SpGEMM algorithm available in the Graphulo library. By applying the proposed matrix partitioning, the performance of the proposed algorithm is further improved considerably.

Keywords Databases · NoSQL · Accumulo · Graphulo · Parallel and distributed computing · Sparse matrices · Sparse matrix–matrix multiplication · SpGEMM · Matrix partitioning · Graph partitioning · Data locality

This work is partially supported by the Scientific and Technological Research Council of Turkey (TUBITAK) under project EEEAG-115E512.

✉ Cevdet Aykanat
aykanat@cs.bilkent.edu.tr

Gunduz Vehbi Demirci
gunduz.demirci@cs.bilkent.edu.tr

[1] Department of Computer Engineering, Bilkent University, Ankara, Turkey

Published online: 28 January 2019

1 Introduction

Relational databases have long been used as data persisting and processing layer for many applications. However, with the advent of big data, the need for storing and processing huge volumes of information made relational databases an unsuitable choice for many cases. Due to the limitations of relational databases, several NoSQL systems have emerged as an alternative solution. Today, big Internet companies use their own NoSQL database implementations especially designed for their own needs (e.g., Google Bigtable [1], Amazon Dynamo [2], Facebook Cassandra [3]). Especially, the design of Google's Bigtable has inspired the development of other NoSQL databases (e.g., Apache Accumulo [4], Apache HBase [5]). Among them, Accumulo has drawn much attention due to its high performance on ingest (i.e., writing data to database) and scan (i.e., reading data from database) operations, which make Accumulo a suitable choice for many big data applications [6].

Solutions for big data problems generally involve distributed computation and need to take the full advantage of data locality. Therefore, instead of using an external system, performing computations inside a database system is a preferable solution [7]. One approach to perform big data computations inside a database system is using NewSQL databases [8]. These type of databases seek solutions to provide scalability of NoSQL systems while retaining the SQL guarantees (ACID properties) of relational databases. However, even though using a NewSQL database can be a good alternative, some researchers take a different approach and seek solutions based on performing big data computations inside NoSQL databases [7]. To that extent, the Graphulo library [9] realizing the kernel operations of Graph Basic Linear Algebra Subprogram (GraphBLAS) [10] in Accumulo NoSQL database is recently developed. GraphBLAS is a community that specifies a set of computational kernels that can be used to recast a wide range of graph algorithms in terms of sparse linear algebraic operations. Therefore, realizing GraphBLAS kernels inside NoSQL databases enables performing big data computations inside these systems, since many big data problems involve graph computations [11].

One of the most important kernel operations in GraphBLAS specification is Sparse Generalized Matrix Multiplication (SpGEMM). SpGEMM forms a basis for many other GraphBLAS operations and used in a wide range of applications in big data domain such as subgraph detection and vertex nomination, graph traversal and exploration, community detection, vertex centrality and similarity computation [9,12,13]. An efficient implementation for SpGEMM in Accumulo NoSQL database is proposed in [14]. The authors actually discuss two multiplication algorithms which are referred to as inner-product and outer-product. Among the two algorithms, the outer-product is shown to be more efficient, and therefore is included in Graphulo Library [9]. The inner-product has the advantage of write-locality while ingesting the result matrix, but has the disadvantage of scanning one of the input matrices multiple times. On the other hand, the outer-product algorithm can not fully exploit write-locality, but requires scanning both of the input matrices only once.

In this work, we focus on improving the performance of SpGEMM in Accumulo for which we propose a new SpGEMM algorithm that overcomes the trade-offs presented earlier. The proposed solution alleviates scanning of input matrices multiple times by

making use of Accumulo's batch-scanning capability which is used for accessing multiple ranges of key-value pairs in parallel. Even though the use of the batch-scanning introduces some latency overheads, these overheads are alleviated by the proposed solution and by using node-level parallelism structures. Moreover, the proposed solution provides write-locality while ingesting the result matrix and does not require further computations when the result matrix need to be scanned, which was not the case for the previously proposed SpGEMM algorithm in [14].

We also propose a matrix partitioning scheme that improves the performance of the proposed SpGEMM algorithm via reducing the total communication volume and providing a balance on the workloads of the servers. Since matrices in Accumulo can only be partitioned according to sorted order of their rows and split points applied on rows, we propose a method that reorders input matrices in order to achieve the desired data distribution with respect to a precomputed partitioning. We cast the partitioning of matrices as a graph partitioning problem for which we make use of a previously proposed bipartite graph model in [15]. We propose a modification to this graph model in order to better comply with the proposed SpGEMM iterator algorithm and Accumulo's own architectural demands.

We conduct extensive experiments using 20 realistic matrices collected from various domains and synthetic matrices generated by graph500 random graph generator [16]. On all test instances, the proposed algorithm significantly performs better than the previously proposed solution without the use of any intelligent input matrix partitioning scheme. The performance of the proposed algorithm is improved even further with the use of the proposed matrix partitioning scheme.

2 Background

2.1 Accumulo

Accumulo is a highly scalable, distributed key-value store built on the design of Google's Bigtable. Accumulo runs on top of Hadoop Distributed File System (HDFS) [17] to store its data and uses Zookeeper [18] to keep coordination among its services. Data is stored in the form of key-value pairs where these key-value pairs are kept sorted at all times to allow fast look up and range-scan operations. Keys consist of five components namely as row, column family, column qualifier, visibility and timestamp. Values can be considered as byte arrays and there is no restriction for their format or size. These key-value pairs are kept sorted in ascending, lexicographical order with respect to the key fields.

Accumulo groups key-value pairs into tables and tables are partitioned into tablets. Tablets of a table are assigned to tablet servers by a master which stores all metadata information and keeps coordination among tablet servers. Tables are always split on row boundaries and all key-value pairs belonging to the same row are stored by the same tablet server. This allows modifications to be performed atomically on rows by the same server. All tables consist of one tablet when they are created for the first time, and as the number of key-value pairs in a tablet reaches to a certain threshold, the corresponding tablet is split into two tablets and one of these tablets is migrated to

another server. It is also possible to manually add split points to a table to create tablets a priori and assign them to servers. This eliminates the need of waiting the tablets to split on their own and allows writing or reading data in parallel, which increases the performance of ingest and scan operations.

Reading data on the client side can be performed using sequential-scanner or batch-scanner capabilities of Accumulo. Sequential-scanning allows access to a range of key-value pairs in sorted order, whereas batch-scanning allows concurrent access to multiple ranges of key-value pairs in unsorted order. Similarly, writing data is performed through using batch-writer which provides mechanisms to perform modifications to tablets in parallel.

It is also possible to perform distributed computations inside Accumulo by using its iterator framework. Iterators are configured on tables for specific scopes, and forms an iterator tree according to their priority. After configured on tables, iterators are applied in succession to the key-value pairs during scan or compaction times. Since a table may consist of multiple tablets and span to multiple tablet servers, each tablet server executes its own iterator stack concurrently, which provides a distributed execution. Users can implement customized iterators that can be plugged into available iterator tree on a table and obtain distributed parallelism by performing a batch-scan operation over a range of key-value pairs. An iterator applied during a batch-scan operation is executed on tablet servers that are spanned by the given range of key-value pairs.

2.2 Related work

Parallelization of SpGEMM operation on shared-memory architectures have been extensively studied in many research works [19–21]. More recently, matrix partitioning schemes that utilize spatial and temporal locality in row-by-row parallel SpGEMM on many-core architectures are proposed in [22].

Several publicly available libraries exist to perform SpGEMM on distributed memory architectures [23,24]. Buluc and Gilbert [25] studies sparse SUMMA algorithm, a message passing algorithm, which employs 2D block decomposition of matrices. Akbudak and Aykanat [26] propose hypergraph models for outer-product message passing SpGEMM algorithm to reduce communication volume and provide computational balance among processors. More recently, hypergraph and bipartite graph models are proposed in [15] for outer-product, inner-product and row-by-row-product formulations of message passing SpGEMM algorithms on distributed memory architectures.

Graphulo library[1] [9] also provides a distributed SpGEMM implementation developed by Hutchison et al. [14], running on top of Accumulo's iterator framework. The method proposed in [14] utilizes the outer-product formulation of SpGEMM in the form of $C = AB$. In this approach each column of A is multiplied by its corresponding row of B (i.e., ith column of A is multiplied by ith row of B), and the resulting matrices of partial products by such multiplications are summed to get the final matrix C. To benefit from high performance attained by rowwise table accesses in Accumulo,

[1] https://github.com/Accla/graphulo.

A^T and B are stored in separate tables (i.e., scanning columns of A corresponds to scanning rows of A^T).

The SpGEMM algorithm proposed in [14] is executed by an iterator applied on a batch-scan performed on the table storing matrix A^T. Therefore, each tablet server iterates through local rows of A^T and scans the corresponding rows of B to perform the outer-product operations between them. The required rows of B can be stored locally as well as stored by other servers, since the A^T and B matrices are stored as separate tables and Accumulo may assign respective tablets to different servers even the same split points are applied on these tables. If we assume that scanning rows of B does not incur any communication costs (i.e., the respective tablets of A^T and B are co-located), writing the resulting C matrix back to the database still incurs significant amount of communication costs in this approach. This is because, the partial result matrices cannot be aggregated before being written back to the database and in the worst case, a partial result matrix can have nonzero entries that should be broadcast to almost all tablet servers available in the system, which may necessitate a tablet server to communicate with all the other tablet servers during this phase. Because of these reasons, writing phase of this outer-product SpGEMM algorithm becomes the main bottleneck. Therefore, Hutchison et al. [14] discuss also an alternative approach and propose an inner-product algorithm which requires scanning the whole B matrix for each local row of A. Although this inner-product approach has the advantage of write-locality, it is considered infeasible due to the necessity of scanning the whole B matrix for each row of A. Therefore, the outer-product implementation of the SpGEMM is included in the Graphulo library.

Our solution to perform SpGEMM in Accumulo differs from [14] in the way that it provides the advantage of write-locality, similar to the one provided by the inner-product approach, and alleviates scanning all rows of matrix B for each row of A by a tablet server. To provide that, our solution makes use of Accumulo's batch-scanning capability which enables accessing to multiple rows of matrix B in parallel. However, our approach suffers from the latency overheads introduced by performing a batch-scan operation for each local row of A. As we discuss in the following sections, this latency overhead can be hugely resolved by batch-processing of multiple local rows of matrix A by tablet servers and using multi-threaded parallelism.

In our experimental framework, MPI-based distributed SpGEMM algorithms are omitted due to the significant differences between MPI and Accumolo iterators, since Accumulo's architectural properties violates fairness of performance comparisons between the proposed SpGEMM iterator algorithm and MPI-based algorithms. For instance, (1) Accumulo provides fault-tolerance mechanisms which incur additional computational overheads (e.g., disk accesses, data replication, cache updates etc.). (2) In MPI-based implementations, input matrices are present in memory of processors before performing communication and arithmetic operations, whereas in Accumulo, input matrices may be present in disk as well. (3) Communication operations are performed very differently in MPI and Accumulo: communication operations between servers are performed in a streaming manner in Accumulo, whereas in MPI, communication operations are performed in synchronized steps.

2.3 Graph partitioning

Let $G = (V, E)$ denote an undirected graph where each vertex $v_i \in V$ is associated with multiple weights of $w^c(v_i)$, for $c = 1 \ldots C$, and each undirected edge $(u_i, v_j) \in E$ between vertices u_i and v_j is associated with a $cost(u_i, v_j)$. A K-way partition $\Pi = \{V_1, V_2 \ldots V_K\}$ of G is composed of mutually exclusive, non-empty subsets of vertices $V_k \subset V$ (i.e., $V_k \cap V_\ell = \emptyset$ if $k \neq \ell$ and $V_k \neq \emptyset$ for each $V_k \in \Pi$) such that the union of these subsets is V (i.e., $\bigcup_{V_k \in \Pi} V_k = V$).

For a partition Π, the weight $W^c(V_k)$ of a part $V_k \in \Pi$ is defined to be the sum of the cth weights $w^c(v_i)$ of vertices in V_k (i.e., $W^c(V_k) = \sum_{v_i \in V_k} w^c(v_i)$). The balancing constraint over a partition Π is defined as

$$W^c(V_k) \leq W^c_{avg}(1 + \epsilon^c), \quad \forall\, V_k \in \Pi \text{ and } c = 1 \ldots C \tag{1}$$

where $W^c_{avg} = \sum_{v_i \in V} w^c(v_i)/K$ and ϵ^c is the maximum allowed imbalance ratio for the cth weight.

An edge $(u_i, v_j) \in E$ is said to be cut if its incident vertices u_i and v_j belong to different parts and uncut otherwise. The cutsize $\chi(\Pi)$ of a partition Π is defined as

$$\chi(\Pi) = \sum_{(u_i, v_j) \in \mathcal{E}^{\Pi}_{\text{cut}}} cost(u_i, v_j) \tag{2}$$

where $\mathcal{E}^{\Pi}_{\text{cut}}$ is the set of cut edges under the partition Π.

The K-way multi-constraint graph partitioning problem [27,28] is an NP-Hard problem which is defined as computing a K-way partition of G that satisfies the balancing constraint according to Eq. (1) and minimizes the cutsize according to Eq. (2). There exist efficient heuristic algorithms and tools producing quality results for the multi-constraint graph partitioning problem [29,30].

3 Row-by-row parallel SpGEMM iterator algorithm

Iterators are the most convenient way to achieve distributed parallelism in Accumulo, since they are concurrently executed by tablet servers on their locally stored data. The proposed iterator algorithm is based on row-by-row parallel matrix multiplication and data distribution.

3.1 Row-by-row parallel SpGEMM

We summarize the row-by-row SpGEMM which leads to row-by-row parallelization: Given matrices $A = (a_{i,j})$, $B = (b_{i,j})$ and $C = (c_{i,j})$, the product $C = AB$ is obtained by computing each row $c_{i,*}$ as follows:

$$c_{i,*} = \sum_{a_{i,j} \in a_{i,*}} a_{i,j} b_{j,*}. \tag{3}$$

Here, $a_{i,*}$, $b_{j,*}$ and $c_{i,*}$ denote the ith row, jth row and ith row of matrices A, B and C, respectively. That is, each nonzero $a_{i,j}$ of row i of A is multiplied by all nonzeros of

row j of B and each multiplication $a_{i,j}b_{j,k}$ for a nonzero $b_{j,k}$ of row j of B produces a partial result for the entry $c_{i,k}$ of row i of C.

Accumulo's data model presents a natural way of storing sparse matrices in such a way that each key-value pair stored in a table has row, column and value subfields which can together store all the necessary information to represent a nonzero entry. Therefore, tables can be seen as sparse matrices that are rowwise partitioned among tablet servers, since tables are always split on row boundaries among tablet servers and all key-value pairs belonging to the same row are always contained in the same tablet server. Due to this correspondence between key-value pairs and nonzero entries, and between tables and sparse matrices, we use these term pairs interchangeably.

Another important feature of Accumulo is that it builds row indexes on tables and allows efficient lookup and scan operations on rows. However, scanning key-value pairs in a column is impractical, because Accumulo does not keep a secondary index on columns and this operation necessitates scanning all rows of a table. If both columns and rows of a table need to be scanned, transpose of the table also need to be stored in a separate table in Accummulo. For instance, the outer-product parallel SpGEMM algorithm [14] needs to scan columns of matrix A for the multiplication $C = AB$; and therefore, keeps A^T instead of A.

3.2 Iterator algorithm

Let matrices A, B and C are stored by K tablet servers where we denote the kth tablet server by T_k for $k = 1 \ldots K$. For now, also assume that these matrices are stored in separate tables (e.g., matrix A is stored in table A).

Since these matrices are rowwise partitioned among tablet servers, also assume that each tablet server T_k stores kth row blocks A_k, B_k and C_k of matrices A, B and C, respectively. Note that a row block of a matrix consists of a number of consecutive rows of the respective matrix (e.g., A_k may consist of rows $a_{i,*}, a_{i+1,*}, \ldots, a_{j,*}$). Here, row blocks C_k and A_k are conformable, i.e., $c_{i,*} \in C_k$ iff $a_{i,*} \in A_k$. Before performing the computation $C = AB$, the matrix C can be thought of as an empty table. In this setting, the proposed iterator algorithm should be configured on a batch-scanner provided with a range covering all rows of matrix A (i.e., the entire range of table A). Performing this batch-scan operation ensures that each tablet server T_k concurrently executes the iterator algorithm on its local portion A_k of A.

After configured on the batch-scanner on matrix/table A, the proposed iterator algorithm proceeds as follows: Each tablet server T_k iterates through its locally stored rows of A_k and computes the corresponding rows $c_{i,*}$ according to Eq. 3. It is important to note that Accumulo provides a programming framework that allows only iteration through key-value pairs (i.e., nonzero entries); and therefore, we can assume that each tablet server T_k is provided with a sorted stream of its local nonzero entries in A_k. These nonzero entries are lexicographically sorted with respect to their first row and then their column indices. Thus, during the scan of this stream, it is possible to scan all nonzeros of A_k in row basis by keeping the nonzero entries belonging to the same row in memory before proceeding to the next row. In this way, iterations can be considered as proceeding rowwise in A_k.

Algorithm 1 Iterator Algorithm

Require: Matrices A, B and C are distributed among K tablet servers.
Require: Local matrices A_k, B_k and C_k stored by server T_k.
Output: Arithmetic results are written back to C_k

```
1: procedure ITERATOR
2:     Initialize a thread cache
3:     Initialize sets Φ, B(Φ)
4:     while source iterator has key-value pairs (i.e., ∃a_{i,j} ∈ A_k) do
5:         Iterate through all nonzero entries of a_{i,*}
6:         for each a_{i,j} ∈ a_{i,*} do
7:             if j ∉ B(Φ) then
8:                 B(Φ) = B(Φ) ∪ {j}
9:         Φ = Φ ∪ {a_{i,*}}
10:        if |B(Φ)| > threshold then
11:            Execute MULTIPLY(Φ, B(Φ)) by a worker thread in the thread cache
12:            Initialize new sets Φ, B(Φ)
13:    Join with all threads in the thread cache
14:    return
```

During the iteration of each row $a_{i,*} \in A_k$, the computation of row $c_{i,*}$ necessitates scanning row $b_{j,*}$ for each nonzero $a_{i,j} \in a_{i,*}$ to perform multiplication $a_{i,j}b_{j,*}$. Some of these required rows of B are stored locally, whereas the remaining rows are stored by other tablet servers. Therefore, to retrieve these B-matrix rows, T_k performs a batch-scan operation provided with multiple ranges covering these required rows over matrix B. As mentioned earlier, Accumulo allows simultaneous access to multiple ranges of key-value pairs via its batch-scanning capability. This operation provides an unsorted stream of nonzero entries belonging to the required rows of B, because these nonzero entries can be retrieved from remote servers in arbitrary order and batch-scan operation does not guarantee any order on them. As these nonzero entries are retrieved, for each retrieved nonzero $b_{j,k}$ of a required row $b_{j,*}$, the computation $c_{i,k} = c_{i,k} + a_{i,j}b_{j,k}$ is performed. The final row $c_{i,*}$ is obtained after the stream of nonzero entries belonging to the required rows of B are all processed. The pseudocode implementation of the proposed iterator algorithm is presented in Algorithm 1.

3.3 Communication and latency overheads

For each row of A_k, the tablet server T_k performs a batch-scan operation on multiple ranges covering the required rows of B (i.e., ranges need to cover each row $b_{j,*}$ of B for each nonzero $a_{i,j} \in a_{i,*}$ of A_k). This operation necessitates T_k to perform one lookup on the row index of B, which is stored by the master tablet server, in order to determine the locations of the required B-matrix rows. After this lookup operation and the servers storing these B-matrix rows are determined, data retrieval is performed via communicating with multiple tablet servers in parallel.

This operation incurs significant latency overheads due to the lookups performed on the master tablet server for each row $a_{i,*} \in A_k$ and establishing connections with tablet servers storing required rows of B. Additionally, this approach may necessitate redundant communication operations and increase the total communication volume among tablet servers, since the same row of B may be retrieved multiple times by the same tablet server for computing different rows of C. In other words, a tablet server

T_k needs to retrieve row j of B for each row of A_k that has a nonzero at column j. For instance, if A_k contains two nonzero entries $a_{i,k}$ and $a_{j,k}$, then two different batch-scan operations performed by T_k to compute rows $c_{i,*}$ and $c_{j,*}$ necessitates retrieval of the same row $b_{k,*}$ twice.

In the proposed algorithm, the aforementioned shortcomings are alleviated by processing multiple rows of A simultaneously. That is, a tablet server T_k iterates through multiple rows in A_k and creates batches of rows to be processed together before performing any computation or a batch-scan operation. Let $\Phi \subseteq A_k$ denote such a batch that is generated by iterating through multiple rows and contains rows $a_{i,*}, a_{i+1,*}, \ldots, a_{j,*}$. Further, let $B(\Phi)$ denote an index set indicating the required B-matrix rows to compute the corresponding rows $c_{i,*}, c_{i+1,*} \ldots c_{j,*}$ for the batch Φ. The row indices in the set $B(\Phi)$ are determined as the union of indices of columns on which rows in Φ have at least one nonzero. That is,

$$B(\Phi) = \left\{ j \mid \exists a_{i,j} \in a_{i,*} \wedge a_{i,*} \in \Phi \right\}. \tag{4}$$

By computing the set $B(\Phi)$, a single batch-scan operation can be performed for multiple rows in Φ to retrieve all of the required rows of B at once instead of performing separate batch-scans for each $a_{i,*} \in \Phi$. After performing a single batch-scan over rows in $B(\Phi)$, the tablet server T_k is again provided with an unsorted stream of nonzero entries belonging to the rows corresponding to row indices in $B(\Phi)$. Each retrieved nonzero entry $b_{j,k}$ of a required row $b_{j,*}$ is then multiplied with each nonzero entry in the jth column segment of Φ. Then, each partial result obtained by multiplication $a_{i,j}b_{j,k}$ is added to the entry $c_{i,k} \in c_{i,*}$. That is, each nonzero $b_{j,k}$ is multiplied by column j of A_k to contribute to the column j of C_k. So, the local matrix multiplication algorithm performed by T_k is a variant of column-by-column paralel SpGEMM although the proposed iterator algorithm utilizes the row-by-row parallelization to gather the B-matrix rows needed for processing nonzeros in A_k.

Processing rows in A_k in batches significantly reduces the latency overheads and communication volume among tablet servers, because both the number of lookup operations and the redundant retrieval of rows of B are reduced by this approach. Here, the size of a batch becomes an important parameter, since it directly affects the number of lookup operations and the communication volume among tablet servers. As the size of a batch increases, both the number of lookup operations and the total communication volume among tablet servers decreases, since as more rows of A_k are added to a single batch, it is more likely that nonzero entries belonging to different A-matrix rows share the same column. However, the size of a batch is bounded by the hardware specifications of tablet servers (i.e., total memory). In our experiments, we determined the best suitable batch size by testing various values in our experimentation environment.

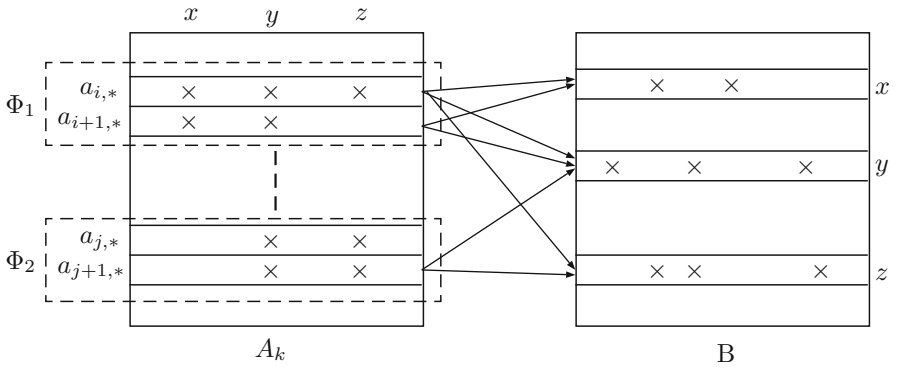


Fig. 1 Sample execution of the proposed iterator algorithm by a tablet server T_k. Batches Φ_1 and Φ_2 are processed by *two* separate threads. Arrows indicate the required rows of matrix B by each worker thread

3.4 Thread level parallelism

Even though the batch processing of multiple rows of A_k by the tablet server T_k significantly reduces the latency overheads, further enhancements for the proposed iterator algorithm can be achieved via using node-level parallelism structures, such as threads.

In a single-threaded execution, the main execution thread of T_k stays idle and performs no computation by the time between initiating the batch-scan and the stream of nonzero entries become ready to be processed during the processing of a batch Φ. In order to avoid this idle time, we utilize a multi-threaded approach in which the main thread assigns the task of processing the current batch Φ of A_k to a worker thread and continues iterating through the remaining rows to prepare the next batch. Whenever the main thread prepares the next batch, it assigns the task of processing this batch to a new worker different than the previous worker(s). This multi-threaded execution enables processing multiple batches of A_k concurrently by worker threads and thus achieving node-level parallelism in a streaming manner. After a batch Φ is fully processed by a worker thread, the resulting C-matrix rows are written back to database by the same worker thread. This write operation will not incur communication if row blocks A_k and C_k are stored by the same server.

In the proposed implementation, creating a new thread for each batch Φ also incurs additional computational cost and latency. In this regard, we make use of thread caches which create new threads only if there is no available thread to process the current batch (Java standard library also provides various thread caches/pools having different implementation schemes). These thread caches create new threads only if necessary and reuses them in order to alleviate the cost of creating new threads.

Figure 1 displays a sample execution of the proposed iterator algorithm by a tablet server T_k. The main execution thread creates batches Φ_1 and Φ_2 and assigns them to worker threads which then perform batch-scan operations to retrieve required rows of B and compute $\{a_{i,*}B, a_{i+1,*}\}B$ and $\{a_{j,*}B, a_{j+1,*}B\}$, respectively. For each batch, the required rows of B are denoted by arrows pointing to the respective rows. For

instance, the worker thread processing batch Φ_1 needs to receive rows x, y and z of matrix B, since rows $a_{i,*}$ and $a_{i+1,*}$ have nonzeros only in these three columns. Similarly, the worker thread processing batch Φ_2 concurrently performs a batch-scan to retrieve rows y and z of matrix B. However, rows y and z need to be retrieved by tablet server T_k twice, since each worker thread scans these rows separately. The redundant retrieval of rows y and z increases the total communication volume, but can be avoided by increasing the batch size. For instance, instead of processing Φ_1 and Φ_2 by separate threads, these batches can be merged into a single batch and processed by the same thread. By this way, a single batch-scan can be performed to retrieve rows $b_{y,*}$ and $b_{z,*}$, and the redundant retrieval of rows y and z can be avoided.

3.5 Write-locality

To obtain write-locality during ingestion of rows in row block C_k, the responsibility of storing C_k must be given to the same tablet server storing row block A_k, since rows of C_k are locally computed on that server. However, if matrices A and C are stored as two different tables, Accumulo can not guarantee that row blocks A_k and C_k are stored by the same tablet server, since Accumulo's load balancer may assign corresponding tablets to different servers according to the partition of key space. Therefore, creating different tables for each matrix may necessitate redundant communication operations during ingestion of the resulting matrix C.

To achieve write-locality discussed as above, we use a single table M instead of three different tables for matrices A, B and C. This approach ensures that rows $a_{i,*}$, $b_{i,*}$ and $c_{i,*}$ are stored by the same tablet server and these rows together belong to ith row of table M. Here, it is worth to note that storing row $b_{i,*}$ along with rows $a_{i,*}$ and $c_{i,*}$ is not necessary for the proposed iterator algorithm and matrix B can be stored in a separate table. On the other hand, we preferred to store all three matrices in a single table M due to the requirements of the input matrix partitioning scheme discussed later in Sect. 4. To distinguish the nonzero entries of these matrices in table M, we use the column family subfield of a key. However, scanning nonzero entries of a specific row of a matrix necessitates scanning nonzero entries of other matrices as well, since ith row of M contains nonzero entries of rows $a_{i,*}$, $b_{i,*}$ and $c_{i,*}$. This inefficiency can be resolved with the use of locality groups which directs Accumulo to separately store the key-value pairs belonging to different column families. That is, even rows $a_{i,*}$, $b_{i,*}$ and $c_{i,*}$ belong to the same row of M and stored by the same tablet server, these rows are separately stored on disk. This allows scanning a range of key-value pairs belonging to the same column family without accessing key-value pairs belonging to the other column families.

Keeping matrices A, B and C under different column families and defining locality groups for these matrices in table M creates an illusion of three different tables being stored in a single table. It is still possible to perform batch-scan over rows of A and B separately without accessing nonzero entries of each other. For instance, the proposed SpGEMM iterator algorithm can be configured on a batch-scanner, given the range covering column family of A, on table M. Each tablet server is still provided with a sorted stream of nonzero entries in A_k and nothing need to be modified in the proposed

iterator algorithm. The only difference is the range provided for the batch-scanner on table M.

In Fig. 2, the contents of table M are displayed for a sample SpGEMM instance, in which two tablet servers are used and a split point 3 is configured on table M. The nonzeros of the first three rows are stored in Tablet 1, whereas the others are stored in Tablet 2. Although the figure shows nonzero entries belonging to different matrices are placed one after each other and kept in lexicographically sorted order,these nonzero entries are stored separately on disk and it is possible to scan any row of a matrix in table M without redundantly accessing nonzero entries of other matrices. However, if the table M is scanned without specifying any column family, all nonzero entries are retrieved in this order.

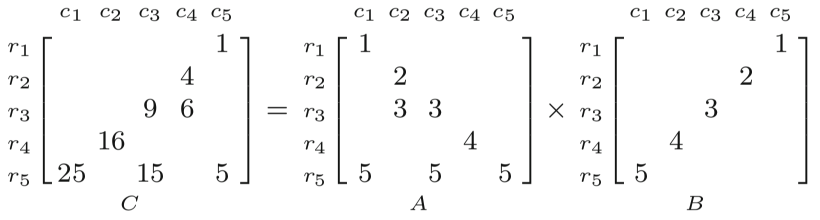
3.6 Implementation

In Algorithms 1 and 2, we present the pseudocode of our implementation for the proposed solution. Algorithm 1 is the main iterator algorithm executed by the main thread running on each tablet server T_k. The main thread iterates through rows in A_k and prepares batches of rows of matrix A and assigns these batches to worker threads to perform multiplication and communication operations. Algorithm 2 is executed by the worker threads to process a given batch Φ.

Accumulo iterators are Java classes that implement SortedKeyValueIterator (SKVI) interface which tablet servers load and execute during scanning or compaction phases on tablets of a table. In order to have custom logic inside Accumulo iterators, a Java class that implements SKVI interface should be written. These iterators are added to iterator trees of tables according to their priority, and the output of an iterator is used as the input of the next iterator whose priority is less than the previous. Therefore, the source of an iterator can be Accumulo's own data sources as well as another iterator having higher priority in the iterator tree. We implemented our algorithm in the *seek()* method of SKVI, which is the first method executed by the iterator after initialized by the tablet server (i.e., Algorithm 1 is executed in the *seek()* method).

In Algorithm 1, the main iterator thread starts with initializing a thread cache and two sets Φ and $B(\Phi)$. For the thread cache, we use Java's own cached thread pool implementation. The set Φ is represented with a two dimensional hash-based table which is available in Google Guava Library [31]. The table data structure supports efficient access to its cells since it is backed with a two dimensional hash-map (i.e., accessing to any cell in the table is performed in constant time). The set $B(\Phi)$ is a typical list data structure and used to keep distinct column indices of nonzero entries in the current batch Φ.

After the initialization step, the main thread starts iterating through the rows in local portion A_k of matrix A. As each row $a_{i,*} \in A_k$ is retrieved, it is included into the current batch Φ until the batch-size threshold is reached. The set $B(\Phi)$ is used to keep distinct column indices of nonzero entries encountered so far in the current batch. These column indices correspond to the rows of B that are required to perform all the multiplications for the batch Φ. These operations are carried out between lines 4 to 12 in Algorithm 1.



(a)

row	family	qualifier	value
1	A	1	1
1	B	5	1
1	C	5	1
2	A	2	2
2	B	4	2
2	C	4	4
3	A	2	3
3	A	3	3
3	B	3	3
3	C	3	9
3	C	4	6

Tablet 1

row	family	qualifier	value
4	A	4	4
4	B	2	4
4	C	2	16
5	A	1	5
5	A	3	5
5	A	5	5
5	B	1	5
5	C	1	25
5	C	3	15
5	C	5	5

Tablet 2

(b)

Fig. 2 A sample SpGEMM instance in which matrices A, B and C are stored in single a table M and partitioned among *two* tablet servers

Algorithm 2 Multiplication Algorithm

```
1: procedure MULTIPLY(Φ,B(Φ))
2:     Perform batch-scan on matrix B over rows b_{j,*} for each j ∈ B(Φ)
3:     Initialize an empty c_{i,*} for each a_{i,*} ∈ Φ
4:     for each retrieved nonzero b_{j,k} ∈ b_{j,*} for a j ∈ B(Φ) do
5:         for each entry a_{i,j} ∈ Φ do
6:             c_{i,k} = c_{i,k} + a_{i,j} * b_{j,k}
7:     for each computed c_{i,*} do
8:         Add c_{i,*} to batch-writer queue
9:     return
```

In the proposed algorithm, we define the batch size threshold in terms of the number of distinct column indices of the nonzero entries in Φ (i.e., the size of the batch is $|B(\Phi)|$). In this way, we try to increase the number of nonzeros that share the same column indices in Φ and therefore reduce the redundant retrieval of B-matrix rows as much as possible. For instance, if two distinct rows $a_{i,*}$ and $a_{j,*}$ have nonzero entries in common column indices and these rows are processed in different batches, then the same B-matrix rows corresponding to these column indices need to be redundantly retrieved for each of these batches by the same tablet server. In a different perspective, if a row of A_k introduced to Φ does not increase the batch size, then all of its nonzeros must share the same column indices with the rows added to Φ earlier. Hence, by increasing the batch size threshold, the likelihood of reducing redundant retrieval of B-matrix rows is possible. For example, in one extreme, if the whole A_k is processed in a single batch, then there will be no redundant communication, since each required B-matrix row need to be retrieved only once. Here, if the batch size is set to a large number, the level of concurrency may degrade in a tablet server, since fewer batches and threads will be generated and all CPU cores may not be efficiently utilized. On the other hand, if the batch size is set to a small number, the number of lookups and the total communication volume will increase. The size of the current batch is controlled in line 10 of Algorithm 1.

After the current batch Φ is prepared, Algorithm 2 provided with the parameters Φ and $B(\Phi)$, is executed on a worker thread chosen from the thread cache. Communication and multiplication operations for the batch Φ are handled by this worker thread. In Algorithm 2, the worker thread first initializes a batch-scanner on matrix B and provides this scanner with a range covering all rows $b_{j,*}$ for each $j \in B(\Phi)$.

After performing the batch-scan operation, the worker thread initializes data structures representing C matrix rows to be computed. To represent an empty $c_{i,*}$ row, we again make use of the two dimensional hash-based table data structure, which was previously used to represent the batch Φ. As mentioned earlier, this data structure enables efficient access to its entries (i.e., each nonzero $c_{i,j}$) and allows us to efficiently combine partial results contributing to the same entries of matrix C, as performed in line 6 of Algorithm 2.

In line 7, a batch-writer is initialized after computing row $c_{i,*}$ for each row $a_{i,*} \in \Phi$. In the for-loop between lines 8 and 9, each computed row $c_{i,*}$ is added to batch-writer queue buffer via a mutation object and written back to the database (i.e., to the ith row of table M under the respective column family for matrix C). As mentioned earlier, these operations do not necessitate any communication operations and can be locally

performed due to the usage of a single table M and the write-locality achieved through this approach.

In Algorithm 1, the execution of the main thread continues until each row $a_{i,*} \in A_k$ is processed. In line 13, the main thread waits for the worker threads in the thread cache to join. Upon joining with all threads, the iterator algorithm finishes and the resulting matrix C is available in table M. It is important to note that to scan the final matrix C, there is no need to apply a summing-combiner or another iterator, as was not the case in the algorithm previously proposed in [14].

The computational complexity of Algorithm 1 depends on the number of nonzero arithmetic operations $flops(A \cdot B)$ required to perform the multiplication $C = AB$ where $flops(A \cdot B) = \sum_i \sum_{a_{ij} \in a_{i,*}} nnz(b_{j,*})$. Insert and lookup operations performed on data structures Φ and $B(\Phi)$ can be performed in constant time, since these data structures are 2-dimensional hash-map-based table and hash-map-based set data structures, respectively. Elements in $B(\Phi)$ can be traversed in time linear time to the number of elements in this data structure. Assuming that the perfect workload load balance is achieved, each server approximately performs $\frac{flops(A \cdot B)}{K}$ nonzero arithmetic operations. Therefore, if the multi-threaded execution is not enabled, computation time of Algorithm 1 can be given as $\Theta(\frac{flops(A \cdot B)}{K})$, since the term $\frac{flops(A \cdot B)}{K}$ dominates other hidden factors associated with the number of local nonzero entries A_k, B_k and C_k on each server T_k. For the communication complexity, each server, in the worst case, may communicate with all other tablet servers and receive $\mathcal{O}(\frac{flops(A \cdot B)}{K})$ nonzero entries of matrix B, since the number of nonzero entries retrieved can not be higher than the number nonzero arithmetic operations performed by a tablet server (i.e., at most, one B-matrix entry can be retrieved for each nonzero arithmetic operation). The communication cost of Algorithm 1 is $\mathcal{O}(t_s(K-1) + t_w \frac{flops(A \cdot B)}{K})$ where t_s and t_w denotes per-message latency and per-word bandwidth costs, respectively. Therefore, the parallel execution time of Algorithm 1 can be given as $\mathcal{O}(t_s K + (1 + t_w) \frac{flops(A \cdot B)}{K})$.

4 Partitioning matrices

Here, we adapt the bipartite graph model recently proposed in [15] for row-by-row parallelization SpGEMM on distributed memory architectures. In this model, the SpGEMM instance $C = AB$ is represented by the undirected bipartite graph $G = (V_A \cup V_B, E)$. The vertex sets V_A and V_B represent the rows of A and B matrices, respectively. That is, V_A contains vertex u_i for each row i of A and V_B contains vertex v_j for each row j of B.

A K-way vertex partition of G

$$\Pi(V) = \left\{ V^1 = V_A^1 \cup V_B^1, V^2 = V_A^2 \cup V_B^2, \ldots, V^K = V_A^K \cup V_B^K \right\}$$

is decoded as a mapping of rows of input matrices to tablet servers as follows: $u_i \in V_A^k$ and $v_j \in V_B^\ell$ correspond to assigning row-i of A and row-j of B to tablet servers T_k and

T_ℓ, respectively. Here, a vertex u_i also represents row $c_{i,*}$, since the tablet server that owns row $a_{i,*}$ is given the responsibility of computing and storing row $c_{i,*}$. Therefore, a partition obtained over rows of A also determines the partition of rows of C.

Through our experimentation and analysis over the proposed iterator algorithm, we observed that the numbers of nonzero entries stored for each of the A, B and C matrices by a server are better measures than the number of floating-point operations performed for representing the associated computational load. That is, nonzero entries of each of the matrices A, B and C should be evenly distributed in order to achieve a workload balance among servers. Therefore, we assume that row i of A and row j of B respectively incur the computational loads of $nnz(a_{i,*})$ and $nnz(b_{j,*})$ to the servers they are assigned to. Here, $nnz(\cdot)$ denotes the number of nonzeros in a row. Note that storing row i of C also incurs the computational load of $nnz(c_{i,*})$, because writing nonzero entries of output matrix C may require more computation time as compared to the scanning entries of input matrices.

Instead of estimating the relative computational loads associated with individual nonzeros of input and output matrices, we propose a three constraint formulation in which we associate three weights with each vertex as follows:

$$\begin{array}{llll} w^1(u_i) = nnz(a_{i,*}), & w^2(u_i) = 0, & w^3(u_i) = nnz(c_{i,*}), & \forall u_i \in V_A \\ w^1(v_j) = 0, & w^2(v_j) = nnz(b_{j,*}), & w^3(v_j) = 0, & \forall v_j \in V_B \end{array}$$

This 3-constraint partitioning captures maintaining a balanced distribution of nonzero entries of all matrices A, B and C among tablet servers. We should note here that the bipartite graph model given here differs from the model given in [15] because of this multi-constraint formulation.

In order to compute w_i^3 of vertex v_i, we need to know the total number of nonzero entries in row $c_{i,*}$ before partitioning, which necessitates performing a symbolic multiplication. This symbolic multiplication can be efficiently performed for just one time, using the proposed SpGEMM algorithm without adopting any partitioning scheme.

In graph G, there exists an undirected edge $(u_i, v_j) \in E$ that connects vertices $u_i \in V_A$ and $v_j \in V_B$ for each nonzero $a_{i,j} \in A$. We associate each edge (u_i, v_j) with a cost equal to the number of nonzeros in the respective row j of B, i.e.,

$$cost(u_i, v_j) = nnz(b_{j,*})$$

This edge-cost definition refers to the amount of communication volume to incur if row i of A and row j of B are assigned to two different tablet servers.

In a given partition Π, uncut edges do not incur any communication. Cut edge $(u_i \in V_A^k, v_j \in V_B^\ell)$ refers to the fact that tablet server T_ℓ stores row j of B, whereas tablet server T_k stores row i of A and is responsible of computing row i of C. Hence, this cut edge will incur the transfer of B matrix row $b_{j,*}$ from tablet server T_ℓ. Thus, the partitioning objective of minimizing the cutsize according to Eq. (2) relates to minimizing the total communication volume that will be incurred due to the transfer of B-matrix rows. However, the cutsize overestimates the total communication volume in some cases: Consider two cut-edges $(u_i \in V_A^k, v_j \in V_B^\ell)$ and $(u_h \in V_A^k, v_j \in V_B^\ell)$ which are incident to the same vertex v_j. These two cut-edges show the need of tablet

server T_k to retrieve row j of B for computing rows h and i of C. The cutsize incurred by these cut-edges according to Eq. 2 will be equal to $cost(u_i, v_j) + cost(u_h, v_j) = 2nnz(b_{j,*})$. However, tablet server T_k may process rows h and i of matrix A in a single-batch, which causes T_k to retrieve row j of B only once, thus necessitating a communication volume of only $nnz(b_{j,*})$ rather than $2nnz(b_{j,*})$.

In general, consider a B-matrix row vertex that has d neighbors (A-matrix row vertices) in V_k. The cutsize definition encodes this situation as incurring a communication volume of $d \times nnz(b_{j,*})$; however, the actual communication volume will vary between $nnz(b_{j,*})$ and $d \times nnz(b_{j,*})$, depending on the number distinct batches of T_k that require row j of B. For example in Fig. 1, the degree of B-matrix row vertex v_y is 3 and its weight is $nnz(b_{y,*}) = 3$. The cutsize encodes the total communication volume as 9. However, row y of B will be retrieved from the respective server to T_k only once for each of the two batches Φ_1 and Φ_2, thus the total communication volume will be 6 instead of 9.

Accumulo partitions tables into tablets via split points defined over row keys. For instance, if the split points a, b, c are applied on table M, rows of matrices A, B and C will be distributed to intervals $[0, a]$, $(a, b]$, $(b, c]$, $(c, \infty]$. For instance, if a row index $i \in [0, a]$, than rows $a_{i,*}$, $b_{i,*}$ and $c_{i,*}$ will be stored by the tablet server responsible for storing interval $[0, a]$.

For a given partition Π of G, the desired data distribution of matrices A, B and C can only be achieved by reordering these matrices in table M and applying a set of proper split points. That is, the sorted order of the new row indices of matrices A, B and C, together with the set of split points, ensure Accumulo to automatically achieve the desired data distribution. So, a given partition Π is decoded as inducing a partial reordering on the rows of the matrices as follows: C-/A-matrix rows corresponding to the vertices in V_A^{k+1} are reordered after the rows corresponding to the vertices in V_A^k and B-matrix rows corresponding to the vertices in V_B^{k+1} are reordered after the rows corresponding to the vertices in V_B^k. The row ordering obtained by this method is referred to as a partial ordering; because rows corresponding to the vertices in a part are reordered arbitrarily. Then the split points are easily determined on the part boundaries of row blocks according to Π.

The size of the proposed bipartite graph model is linear in the number of rows, columns and nonzero entries of matrix $A \in \mathbb{R}^{m \times n}$, since there exist a vertex v_i for each row $a_{i,*}$, a vertex v_j for each row $b_{j,*}$ and an edge for each nonzero entry $a_{i,j}$ in matrix A. So the topology of the bipartite graph can be built in $\Theta(m+n+nnz(A))$ time. The first and second weights ($w^1(v_i)$ and $w^2(v_i)$) of vertices can be determined from input matrices in $\Theta(nnz(A) + nnz(B))$ time. Computing the third weight ($w^3(v_i)$) of vertices necessitates the symbolic multiplication of input matrices A and B (since $w^3(v_i) = nnz(c_{i,*})$). Hence, the complexity of building the proposed graph model is $\Theta(flops(A \cdot B) + m + n + nnz(A) + nnz(B))$. On the other hand, complexity of the partitioning phase depends on the partitioning tool. In [32], complexity of Metis is reported as $\Theta(V + E + K \log K)$ where V $(= m + n)$ is number of vertices, E $(= nnz(A))$ is number of edges in a graph and K is the number of parts. The running time of the partitioning phase becomes $\Theta(m + n + nnz(A) + K \log K)$ thus

leading to overall running time complexity of $\Theta(flops(A \cdot B) + m + n + nnz(A) + nnz(B) + K \log K) = \Theta(flops(A \cdot B) + K \log K)$.

5 Experimental evaluation

We compare the performance of the proposed SpGEMM iterator algorithm (RRp) against the baseline algorithm (BL) [14], which is currently available in the Graphulo Library, on a fully distributed Accumulo cluster. We also evaluate the performance of the graph partitioning-based RRp (gRRp), where the input and output matrices are reordered using the partitioning scheme proposed in Sect. 4. We use the state-of-the-art graph partitioning tool Metis to partition the graph model in gRRp. We set the maximum load imbalance $\epsilon = 0.005$ and performed edge cut minimization. We used both realistic and synthetically generated sparse matrices as SpGEMM instances in our experiments.

5.1 Datasets

Table 1 displays properties of matrices used in the experiments. These matrices are included in our dataset since they arise in various real-world applications and also used in recent research works [21,22,33,34].

We performed our experiments in two different categories: In the first category, a sparse matrix is multiplied with itself (i.e., $C = AA$), and in the second category, two different conformable sparse matrices are multiplied (i.e., $C = AB$). The first category $C = AA$ arises in graph applications such as finding all-pairs-shortest paths [35], self similarity joins and summarization of sparse dataset [36,37]. The second category $C = AB$ is a more general case and especially arise in applications such as collaborative filtering [38] and similarity joins of two different sparse datasets [37].

The $C = AA$ category contains 13 sparse matrices all selected from UFL sparse matrix collection [39]. As seen in Table 1 all of these matrices contain more than 100K rows.

The $C = AB$ category contains seven SpGEMM instances. The SpGEMM instances *amazon0302* and *amazon0312* are used for collaborative filtering in recommendation systems [38]. In these instances, A matrices represent similarities of items and B matrices represent preferences of users and synthetically generated following the approach in [22], where the item preferences of users follow Zipf distribution. The SpGEMM instances *boneS01*, *cfd2*, *offshore* and *shipsec5* are used during the setup phase of Algebraic multigrid methods (AMG) [40]. In these instances, A matrices are selected from UFL and their corresponding interpolation operators are generated as the B matrices by using a tool[2] in [40] (matrices with suffix ".P" in Table 1). The last SpGEMM instance contains two conformable matrices *thermomech_dK* and *thermomech_dM*.

The $C = AB$ category also contains four SpGEMM instances whose A and B matrices are synthetically generated by using the Graph500 power law graph generator [16].

[2] https://github.com/pyamg/pyamg.

Table 1 Dataset properties

Matrix	Number of			Number of nonzeros			
				in a row		in a column	
	Rows	Columns	Nonzeros	Avg	Max	Avg	Max
$C = AA$							
2cubes_sphere	101,492	101,492	1,647,264	16	31	16	31
filter3D	106,437	106,437	2,707,179	25	112	25	112
598a	110,971	110,971	1,483,868	13	26	13	26
torso2	115,967	115,967	1,033,473	9	10	9	10
cage12	130,228	130,228	2,032,536	16	33	16	33
144	144,649	144,649	2,148,786	15	26	15	26
wave	156,317	156,317	2,118,662	14	44	14	44
majorbasis	160,000	160,000	1,750,416	11	11	11	18
scircuit	170,998	170,998	958,936	6	353	6	353
mac_econ_fwd500	206,500	206,500	1,273,389	6	44	6	47
offshore	259,789	259,789	4,242,673	16	31	16	31
mario002	389,874	389,874	2,101,242	5	7	5	7
tmt_sym	726,713	726,713	5,080,961	7	9	7	9
$C = AB$							
cfd2 (A)	123,440	123,440	3,087,898	25	30	25	30
cfd2.P (B)	123,440	4825	528,769	4	10	110	181
boneS01 (A)	127,224	127,224	6,715,152	53	81	53	81
boneS01.P (B)	127,224	2,394	470,235	4	10	196	513
shipsec5 (A)	179,860	179,860	10,113,096	56	126	56	126
shipsec5.P (B)	179,860	2959	541,099	3	13	183	456
thermomech_dK (A)	204,316	204,316	2,846,228	14	20	14	20
thermomech_dM (B)	204,316	204,316	1,423,116	7	10	7	10
offshore (A)	259,789	259,789	4,242,673	16	31	16	31
offshore.P (B)	259,789	9893	1,159,999	4	13	117	221
amazon0302 (A)	262,111	262,111	1,234,877	5	5	5	420
amazon0302-user (B)	262,111	50,000	576,413	2	302	12	27
amazon0312 (A)	400,727	400,727	3,200,440	8	10	8	2747
amazon0312-user (B)	400,727	50,000	882,813	2	1, 675	18	38
$C = AB$ (graph500)							
scale = 15 (A)	32,768	32,768	441,173	13	5942	13	2067
(B)	32,768	32,768	441,755	13	5976	13	2041
scale = 16 (A)	65,536	65,536	909,301	13	9719	13	3273
(B)	65,536	65,536	909,854	13	9670	13	3407
scale = 17 (A)	131,072	131,072	1,864,398	14	15, 643	14	5227
(B)	131,072	131,072	1,864,338	14	15, 743	14	5301
scale = 18 (A)	262,144	262,144	3,806,212	14	25, 332	14	8303
(B)	262,144	262,144	3,804,831	14	25, 324	14	8277

These synthetic matrices are previously used in [14] to evaluate the performance of the BL algorithm. We also use this tool with the same set of parameters. The tool takes two parameters, referred to as *scale* and *edge-factor*, and produces square matrices with 2^{scale} rows and *edge-factor* $\times\ 2^{scale}$ nonzero entries. We fix *edge-factor* to 16 as in [14] and set *scale* $= 15, 16, 17, 18$ to generate *four* different sized SpGEMM instances (e.g., for *scale* $= 15$ we generate two different square matrices A and B with 2^{15} rows and 16×2^{15} nonzero entries).

5.2 Accumulo cluster

The cluster we used in our experiments consists of 12 nodes and these nodes are connected via *DGS-3120-24TC* ethernet switch. Each node has *two Intel-Xeon-E5-2690-v4* processors each of which consists of 14 cores and is able to run 28 threads concurrently. Additionally, each node has 256 GB main memory in addition to its 16 TB local storage. We designate *two* of these nodes as control nodes on which we run ZooKeeper, HDFS NameNode, Accumulo master, garbage collector and monitor processes. The remaining nodes are used as worker nodes and run only HDFS DataNode and Accumulo tablet server processes. In order to conduct strong scalability analysis, we run BL, RRp and gRRp algorithms for all SpGEMM instances on $K = 2, 4, 6, 8$ and 10 tablet servers.

5.3 Evaluation framework

To measure the running time of BL, we first ingest input matrices A and B as separate tables, then we define split points that distribute rows of matrices to tablet servers evenly. By this partitioning, the first $K - 1$ tablet servers are assigned $\lfloor n/K \rfloor$ rows (n being the number of rows of a matrix) and the last tablet server is assigned all the remaining rows. The Graphulo library does not support any other load balancing scheme other than defining split points on rows as performed in our implementation, and this approach is also followed in [14]. After ingesting the A and B matrices, we call the SpGEMM routine provided by Graphulo library from a client process running on one of the control nodes. We measure the running time of BL as the total time the corresponding routine takes within the client process.

To measure the running time of RRp, we ingest matrices A and B in a single table M and we define split points on table M to evenly distribute rows of M among tablet servers, as done in BL. After this operation, we call the proposed SpGEMM routine within a client process running on one of the control nodes and measure the total running time of the multiplication operation. In this routine, the proposed iterator is applied on a batch-scanner which is provided with a range covering all rows of A under the column family of table M. The running time of gRRp is measured similarly.

The running times of all algorithms are obtained by averaging 5 successive runs. These times cover the entire process of performing multiplications of matrices A and B and writing the resulting matrix C back to database. However, in BL, outer-product results that contribute to the same nonzero entries of C are not combined/summed before being written back to table C. Summations of these partial results are performed

by applying a scanning-time summing-combiner on table C. Therefore, to obtain the final matrix C in parallel by all tablet servers, a batch-scan operation covering all rows of matrix C need to be performed. On the other hand, the C matrices computed by RRp and gRRp do not require applying a summing-combiner or performing any further computations, since all partial results contributing to the same nonzero entries of C are already combined/summed before being written back to the database. This difference violates the fairness of the comparisons made among the algorithms, since BL performs less computation than both RRp and gRRp during SpGEMM operations. In this regard, we also compare the time required to scan the C matrices produced by all algorithms and include in our experimental results. In order to get scanning times, we performed a batch-scan operation covering all rows of C, after performing the SpGEMM operation on matrices A and B, and measure the time required to receive all the nonzero entries of C by the client process running on one of the control nodes.

5.4 Experimental results

Table 2 displays the measured running times of BL, RRp and gRRp to perform SpGEMM instances on $K = 2, 4, 6, 8$ and 10 tablet servers. The rows entitled as "norm avgs wrto BL" display the geometric means of the ratios of the running times of RRp and gRRp to those of BL in the respective categories.

As seen in Table 2, in both $C = AA$ and $C = AB$ categories, RRp consistently performs much better than BL on all test instances except for *amazon0312* on $K = 2$ servers. Moreover, the performance improvement of RRp over BL increases with increasing K. On average, in the $C = AA$ category, RRp runs 1.75×, 2.56×, 2.85×, 3.33× and 3.44× faster than BL on $K = 2, 4, 6, 8$ and 10 tablet servers, respectively. Similarly, these values become 2.50×, 3.70×, 3.80×, 4.34× and 4.34× in the $C = AB$ category. Additionally, we also observe that BL can not scale on some instances, especially on *scircuit* and *tmt_sym* instances, where the running time of BL increases as K increases from 8 to 10. However, RRp displays better scalability than BL, since in all test cases the running time of RRp decreases with the increasing value of K.

The performance of RRp is further improved by gRRp on almost all SpGEMM instances. On average, gRRp provides an improvement of 12% over RRp on $K = 2$ servers and this improvement significantly increases to 32% on $K = 10$ servers in the $C = AA$ category. In the $C = AB$ category, gRRp performs 12% better than RRp on $K = 2$ servers and this improvement slightly increases to 13% on $K = 10$ servers. These results indicate that the graph partitioning approach increases the scalability of RRp, since the decrease in the communication volume among tablet servers increases the efficiency of parallel algorithm.

Figure 3a displays the average speedup curves for the multiplication phase over each SpGEMM category. Speedup values on an SpGEMM instance are attained with respect to the running time of BL for the same instance on $K = 2$ tablet servers. In both categories, both RRp and gRRp scale linearly and display significantly better scalability than BL. For example, on $K = 10$ servers, RRp and gRRp achieve speedup values of 6.37 and 8.82 respectively, whereas BL achieves only 1.84 in the $C = AA$ category. Similarly, these speedup values become 11.50, 13.80 and 2.89 respectively

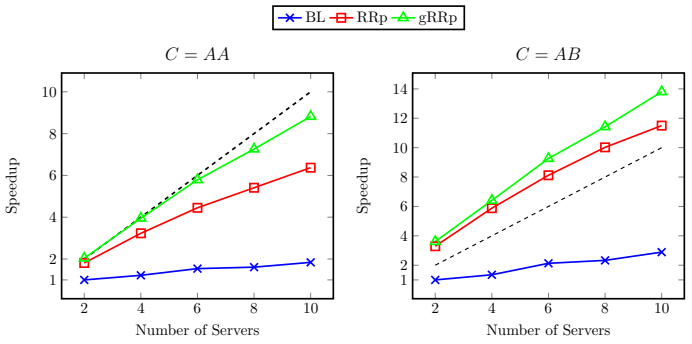
Table 2 Multiplication times (ms)

	$K = 2$			$K = 4$			$K = 6$			$K = 8$			$K = 10$		
	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp
$C = AA$															
2cubes_sphere	16,571	6735	**6057**	13,145	3741	**3135**	10,045	2803	**2182**	11,332	2264	**1731**	7373	1826	**1500**
filter3D	47,867	**19,199**	19,893	29,667	10,175	**9116**	22,837	6898	**5999**	19,025	5663	**4659**	12,916	5055	**3711**
598a	15,009	10,745	**7812**	14,126	6070	**3920**	13,145	5266	**2669**	13,991	4368	**2195**	9579	3491	**1654**
torso2	7434	3277	**3039**	7091	2095	**1775**	5949	1495	**1250**	5851	1162	**1033**	4810	938	**863**
cage12	25,809	16,024	**12,069**	23,050	9076	**6067**	15,319	6367	**4301**	11,264	4855	**3113**	13,282	4272	**2632**
144	21,852	15,422	**10,875**	14,429	9070	**5725**	13,038	6436	**3696**	12,838	5955	**3061**	11,974	5093	**2352**
wave	19,179	**8478**	8686	16,237	4553	**4193**	11,275	3389	**2757**	13,612	2915	**2330**	11,856	2440	**1873**
majorbasis	12,518	7030	**6109**	10,648	3623	**3170**	9990	2787	**2274**	9499	2123	**1776**	8074	1951	**1524**
scircuit	7203	4635	**3780**	7298	2949	**2089**	5770	2169	**1592**	5093	1809	**1225**	6363	1500	**1020**
mac_econ_fwd500	8454	5238	**4858**	6743	2899	**2681**	5210	2329	**2067**	5582	1795	**1580**	4630	1670	**1310**
offshore	40,348	23,047	**21,880**	28,652	12,457	**9985**	22,906	8163	**6276**	21,400	7274	**5157**	18,868	5602	**3892**
mario002	10,042	8053	**6866**	10,164	4702	**4146**	8039	3160	**2438**	8945	3058	**2023**	9074	2914	**1697**
tmt_sym	23,497	13,831	**14,059**	18,026	7341	**7271**	13,184	5111	**5107**	12,031	4084	**4059**	13,554	**3350**	3340
norm avgs wrto BL	1.00	0.57	0.50	1.00	0.39	0.31	1.00	0.35	0.27	1.00	0.30	0.22	1.00	0.29	0.20

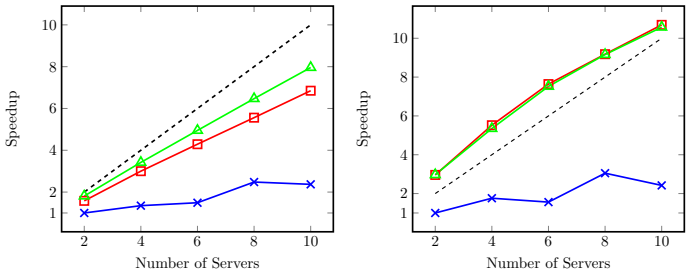
Table 2 continued

	$K = 2$			$K = 4$			$K = 6$			$K = 8$			$K = 10$		
	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp
$C = AB$															
cfd2	11,505	4958	**4345**	12,265	2471	**2254**	9891	1746	**1620**	8993	1336	**1322**	7765	**1125**	1220
boneS01	21,955	7693	**7169**	21,759	3994	**3914**	16,436	2717	**2665**	13,695	**2123**	2175	13,159	**1810**	1907
shipsec5	27,973	10,443	**10,336**	25,502	5659	**5463**	21,102	3829	**3780**	18,731	**2983**	3115	17,891	**2430**	2599
thermomech_dK	13,074	8471	**6836**	13,364	4575	**3738**	10,799	3396	**2616**	10,552	2914	**2205**	11,056	2626	**1758**
offshore	15,680	7854	**6977**	13,580	4437	**3826**	12,951	3270	**2761**	12,436	2754	**2231**	11,439	2330	**1844**
amazon0302	4573	3913	**3429**	5283	2353	**2062**	4821	1716	**1514**	4941	1592	**1551**	4410	1595	**1439**
amazon0312	9862	10,394	**8498**	9304	6333	**4616**	8735	4625	**3606**	8458	4223	**3123**	8408	3870	**2828**
$C = AB$ (Graph500)															
scale = 15	19,452	3776	**3186**	15,058	**2228**	2739	10,273	1611	**1476**	7794	**1273**	1346	6130	1285	**1049**
scale = 16	38,634	7037	**6750**	25,109	4064	**3351**	12,283	3231	**2641**	12,202	2617	**1888**	11,679	2171	**1627**
scale = 17	69,797	16,079	**15,161**	38,901	8959	**7652**	23,787	6280	**5472**	21,996	5139	**4313**	12,760	4518	**3422**
scale = 18	126,538	34,802	**32,704**	59,441	18,703	**16,498**	37,432	13,187	**11,333**	26,735	10,890	**9366**	21,816	8738	**7560**
norm avgs wrto BL	1.00	0.40	0.35	1.00	0.27	0.24	1.00	0.26	0.22	1.00	0.23	0.21	1.00	0.23	0.20

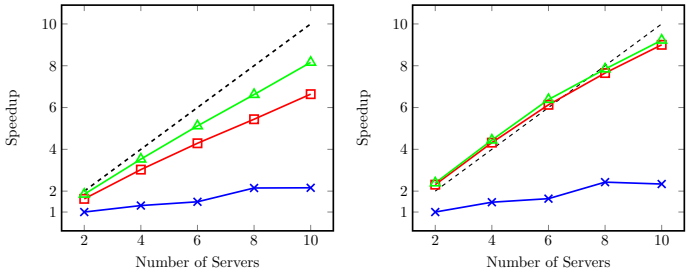
BL: SpGEMM implementation provided in the Graphulo library
RRp: The proposed row-row parallel SpGEMM algorithm
gRRp: RRp algorithm enhanced via the proposed graph partitioning scheme



(a) Speedup curves for multiplication phase.



(b) Speedup curves for scanning phase.



(c) Speedup curves for the overall execution time.

Fig. 3 Average speedup curves of BL, RRp and gRRp with respect to the running time of BL on $K = 2$ tablet servers. **a** Multiplication phase. **b** Scanning phase. **c** Overall execution time

in the $C = AB$ category. Additionally, as can be inferred from the speedup curves, the efficiency of RRp is further increased by gRRp and higher speedup values are obtained via intelligent partitioning of input matrices. As also seen in Fig. 3a, the performance improvement of gRRp over RRp on the scalability is much more pronounced in the $C = AA$ category than in the $C = AB$ category.

Table 3 displays the times required to scan C matrices produced by all algorithms after they perform respective SpGEMM instances. As seen in the table, scanning phase of BL requires significantly more time than those of RRp and gRRp due to the summing-combiner applied on table C by Graphulo library. In the $C = AA$ category, on average, scanning phase of RRp and gRRp algorithms runs 1.55× and 1.78×

Table 3 Scanning times (ms)

	$K = 2$			$K = 4$			$K = 6$			$K = 8$			$K = 10$		
	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp
$C = AA$															
2cubes_sphere	52,724	**28,486**	29,493	31,742	**14,585**	14,674	35,208	**10,045**	10,429	16,774	8007	**7780**	20,923	6377	**6309**
filter3D	143,178	63,810	**54,195**	71,325	33,302	**29,067**	91,690	23,924	**20,777**	43,151	18,290	**15,669**	60,406	15,005	12,848
598a	44,983	24,419	**22,803**	38,324	12,456	**11,541**	29,689	8943	**8173**	19,749	6939	**6187**	17,522	5901	**5040**
torso2	15,661	8171	**7052**	12,447	4482	**3907**	8923	3132	**2692**	5429	2438	**2096**	6408	1968	**1782**
cage12	81,726	60,256	**49,012**	74,981	34,524	**25,288**	53,328	25,208	**17,928**	32,859	18,130	**13,724**	31,844	15,248	**10,953**
144	61,819	34,998	**33,596**	39,872	18,325	**17,037**	41,106	13,134	**11,940**	22,026	9814	**9240**	26,003	7829	**7312**
wave	57,264	34,081	**29,535**	37,259	18,428	**15,747**	39,820	12,480	**10,996**	23,362	9700	**8391**	24,837	7897	**6898**
majorbasis	35,260	26,675	**23,223**	32,860	13,289	**11,997**	22,657	9313	**7940**	15,874	7178	**6223**	15,125	5774	**5083**
scircuit	19,146	19,087	**13,420**	19,463	11,357	**6972**	16,226	7783	**5068**	12,304	6151	**3899**	9678	4739	**3114**
mac_econ_fwd500	22,171	20,928	**19,090**	24,874	**9806**	10,086	15,467	6917	**6417**	12,342	5323	**4878**	9316	4570	**3941**
offshore	126,598	72,887	**62,443**	69,840	38,687	**33,695**	85,213	26,481	**23,554**	46,630	20,672	**18,455**	55,269	16,722	**14,625**
mario002	30,367	22,647	**17,219**	24,750	11,726	**10,092**	19,768	8002	**6398**	14,689	6351	**4855**	12,770	5171	**4001**
tmt_sym	63,949	43,907	**41,354**	50,607	23,569	**22,175**	47,486	16,106	**14,788**	24,992	12,495	**11,182**	28,101	9471	**9017**
norm avgs wrto BL	1.00	0.65	0.56	1.00	0.45	0.39	1.00	0.35	0.30	1.00	0.45	0.38	1.00	0.35	0.30

Table 3 continued

	$K = 2$			$K = 4$			$K = 6$			$K = 8$			$K = 10$		
	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp	BL	RRp	gRRp
$C = AB$															
cfd2	14,216	**3325**	3429	7159	**1871**	1935	8791	**1360**	1367	4971	**1092**	1140	5963	**988**	1017
boneS01	28,430	3043	**2989**	13,640	**1576**	1708	15,648	**1179**	1258	6992	**1016**	1045	10,483	**905**	918
shipsec5	26,568	3305	**3236**	14,656	**1800**	1872	19,807	**1286**	1346	7750	1132	**1127**	12,417	**953**	1000
thermomech_dK	34,299	22,103	**21,557**	34,025	**11,605**	11,799	24,330	8292	**7475**	15,009	6475	**6359**	14,238	**5095**	5266
offshore	24,088	**9488**	9670	15,157	5361	**5013**	15,392	3786	**3541**	9649	3061	**2889**	9346	2717	**2476**
amazon0302	8886	**7411**	7474	5022	3954	**3920**	6998	2826	**2793**	4930	2282	**2260**	4318	**1891**	1978
amazon0312	22,306	19,897	**19,057**	15,425	10,800	**10,051**	17,656	7871	**7431**	13,814	6147	**5599**	11,674	**5047**	5090
$C = AB$ (Graph500)															
scale = 15	14,498	**12,362**	12,814	7738	6541	**6537**	9342	4663	**4501**	4039	3702	**3587**	5381	3159	**2920**
scale = 16	29,010	**25,494**	26,976	14,348	**13,200**	13,704	16,136	9293	**9172**	7530	**7162**	7236	11,131	6079	**5856**
scale = 17	62,098	60,138	**59,154**	34,251	**30,955**	31,535	35,721	**21,170**	21,173	16,650	**16,006**	16,410	24,192	**13,063**	13,249
scale = 18	137,408	**122,971**	125,845	68,557	64,576	**64,350**	74,625	**43,305**	44,737	36,117	34,335	**34,205**	53,888	**27,252**	27,574
norm avgs wrto BL	1.00	0.49	0.49	1.00	0.45	0.46	1.00	0.29	0.28	1.00	0.43	0.43	1.00	0.30	0.30

BL: SpGEMM implementation provided in the Graphulo library
RRp: The proposed row-row parallel SpGEMM algorithm
gRRp: RRp algorithm enhanced via the proposed graph partitioning scheme

faster than that of BL on $K = 2$ servers and this performance improvement increases to $2.85\times$ and $3.33\times$ on $K = 10$ servers, respectively. Similarly in the $C = AB$ category, scanning phase of RRp and gRRp runs $2.04\times$ faster than that of BL on $K = 2$ servers and this improvement increases to $3.33\times$ on $K = 10$ servers, respectively.

Figure 3b displays the average speedup curves for the scanning phase over each SpGEMM category. As seen in Fig. 3b, both RRp and gRRp displays significantly better scalability than BL. Comparison of Fig. 3a and b show that the performance improvement of gRRp over RRp is considerably less on the scanning phase than the multiplication phase, as discussed earlier. This is because the scanning phase involves communication only between the control server and tablet servers (i.e., no communication among tablet servers) and the graph partitioning only encapsulates the amount of communication volume during the multiplication phase. The considerable performance improvement of gRRp over RRP in $C = AA$ category can be attributed to the fact that gRRp provides better nonzero distribution of C matrix among tablet servers, due to the third constraint placed on vertices (i.e., the w_i^3 weight), as compared to RRp, and therefore, achieves better load balance during the scan of C-matrix.

It is worth to mention that speedup values attained by BL in the scanning phase decrease as the number of tablet servers increases in some cases (e.g., when the number of servers increases from 8 to 10 in Fig. 3b). This performance decrease can be mainly attributed to the partitioning scheme used in BL where input matrices are partitioned rowwise among servers without considering the workload associated with these rows (i.e., each server is assigned $\lfloor n/K \rfloor$ rows). However, the computational load associated with each row may drastically deviate, since these rows, especially dense rows, may necessitate many summing-combining operations due to the partial results contributing to the same nonzero entries. Thus BL suffers from load imbalance due to uneven nonzero entry distribution among tablet servers, leading to lower speedup values.

Figure 3c displays the average speedup curves for the overall execution time (i.e. the total time spent on multiplication plus scanning phases) of the SpGEMM operations. As seen in Fig. 3c, both RRp and gRRp scales significantly better than BL in the total execution time.

Figure 4 displays the variation of the running times of the multiplication phase of all algorithms on $K = 10$ servers with increasing scale factor of SpGEMM instances that are produced by the graph500 tool. This figure is included here in accordance with the experimental results reported in [14]. As seen in the figure, both RRP and gRRp perform significantly better than BL, whereas gRRp performs slightly better than RRp.

5.5 Varying key-value size

In this section, we report the results of the experiments conducted to show the variation of the performance improvement of gRRp over RRp with increasing key-value pair sizes. For the previous experiments, we designate the sizes of row, column and value fields of key-value pairs as 8 bytes each (i.e., the size of each nonzero is 24 bytes). Hence, in order to obtain different sized key-value pairs for an SpGEMM instance,

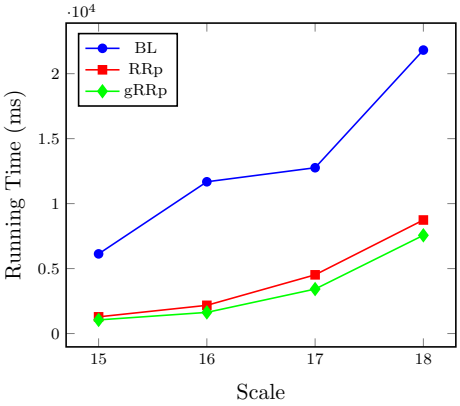


Fig. 4 Running times of BL, RRp and gRRp to perform SpGEMM instances, which are generated by graph500 tool, on $K = 10$ tablet servers

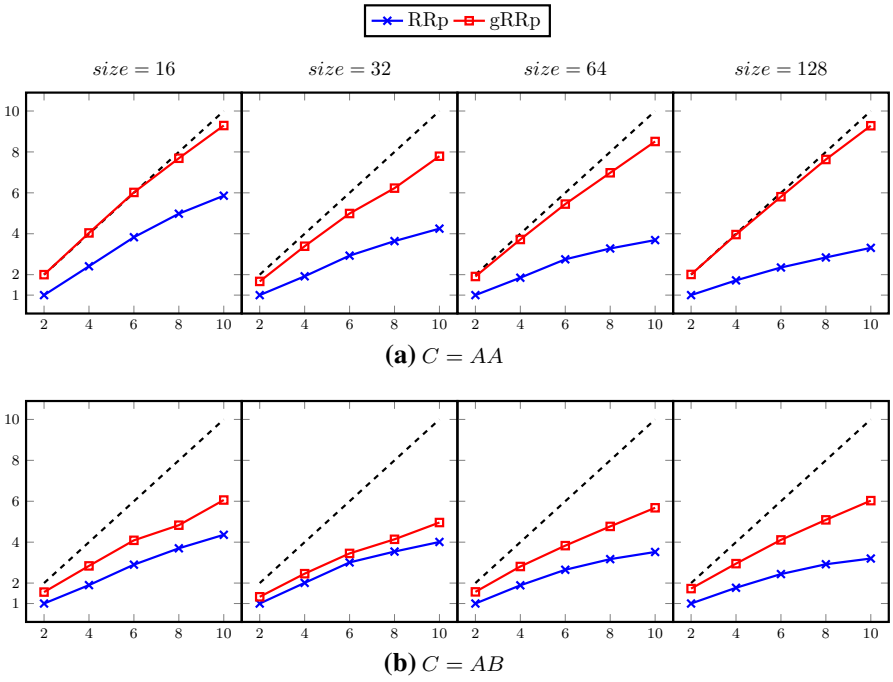


Fig. 5 Average speedup curves of RRp and gRRp with varying key-value sizes. Speedup values are computed with respect to the running times of RRp on $K = 2$ tablet servers

we keep the sizes of row and column fields fixed and set the size of the value field to 16, 32, 64 and 128-bytes. The scaler multiplication of two value fields is performed via Java's BigDecimal Class.

In Fig. 5, we plot the average speedup curves of RRp and gRRp, with different key-value sizes, over each SpGEMM category. For each SpGEMM instance and value-field

size, speedup values are computed with respect to the running time of RRp on $K = 2$ tablet servers. As seen in the figure, the performance improvement of gRRp over RRp increases significantly for both categories with increasing key-value pair size. For example, for the $C = AA$ category on $K = 10$ servers, gRRp runs 1.58×, 1.83, 2.30 and 2.80× faster than RRp for value-field sizes of 16, 32, 64 and 128 bytes, respectively. Similarly, for the $C = AB$ category on $K = 10$ servers, gRRp runs 1.39×, 1.24×, 1.61× and 1.88× faster than RRp for value-field sizes of 16, 32, 64 and 128 bytes, respectively. This increase in the performance improvement of gRRp over RRp is because of the fact that the communication volume increases with increasing value-field sizes and hence the improvements to be attained by graph partitioning, which has the objective of minimizing total communication volume, become more pronounced on the overall performance. So the preprocessing overhead is expected to amortize in such applications, since the partitioning overhead is independent of the sizes of key-value pairs.

6 Conclusion

We proposed an iterator algorithm to perform distributed SpGEMM in Accumulo database. The proposed algorithm utilizes row-by-row parallel SpGEMM which achieves write-locality during ingestion of the output matrix. However, this approach necessitates performing multiple batch-scan operations, which introduces significant latency overheads. These overheads are alleviated by performing local SpGEMM operations via multiple batches and utilizing multi-threaded parallelism. Extensive experiments performed on a wide range of realistic and synthetic SpGEMM instances showed that the proposed algorithm outperforms the outer-product implementation provided in the Graphulo library, by a large margin.

We also proposed a matrix partitioning scheme that reduces the total communication volume while maintaining workload balance among servers. The experiments also showed that the proposed matrix partitioning scheme provides significant improvements. The preprocessing overhead due to graph partitioning is expected to amortize in applications that require repeated SpGEMMs involving input matrices having the same sparsity patterns as well as applications that have large key-value sizes. The latter is because of the fact that the partitioning overhead is independent of the sizes of key-value pairs, whereas communication-volume overhead increases with increasing sizes of key-value pairs thus increasing the performance improvement attained by using graph partitioning.

The proposed SpGEMM algorithm adopts 1D matrix partitioning scheme in which matrices are rowwise partitioned among tablet servers. However, 1D algorithms face communication bottlenecks when the number of tablet servers increases, since the total communication volume and the number of messages need to be handled by a single server may drastically increase. In order to address this issue, as a future work, we will also investigate 2D matrix partitioning schemes in which matrices are partitioned both rowwise and column-wise among servers.

Iterative methods to solve problems such as sparse non-negative matrix factorization [41–44] heavily depend on repeated sparse matrix and low-rank matrix

multiplication. We observed significant computational and latency overheads during repeated invocation of iterator algorithms in Accumulo. Therefore, we are planning to investigate potential performance enhancements to the proposed SpGEMM iterator algorithm for efficient use in such cases.

Acknowledgements The numerical calculations reported in this paper were fully performed at TUBITAK ULAKBIM, High Performance and Grid Computing Center (TRUBA resources).

References

1. Chang, F., Dean, J., Ghemawat, S., Hsieh, W.C., Wallach, D.A., Burrows, M., Chandra, T., Fikes, A., Gruber, R.E.: Bigtable: a distributed storage system for structured data. ACM Trans. Comput. Syst. (TOCS) **26**(2), 4 (2008)
2. DeCandia, G., Hastorun, D., Jampani, M., Kakulapati, G., Lakshman, A., Pilchin, A., Sivasubramanian, S., Vosshall, P., Vogels, W.: Dynamo: amazon's highly available key-value store. In: ACM SIGOPS operating systems review, vol. 41, pp. 205–220. ACM (2007)
3. Lakshman, A., Malik, P.: Cassandra: a decentralized structured storage system. ACM SIGOPS Oper. Syst. Rev. **44**(2), 35–40 (2010)
4. Fuchs, A.: Accumulo-extensions to googles bigtable design, National Security Agency, Tech. Rep (2012)
5. Apache hbase. https://hbase.apache.org/ (2018). Accessed 15 April 2018
6. Sen, R., Farris, A., Guerra, P.: Benchmarking apache accumulo bigdata distributed table store using its continuous test suite. In: 2013 IEEE International Congress on Big Data (BigData Congress), pp. 334–341. IEEE (2013)
7. Hutchison, D., Kepner, J., Gadepally, V., Howe, B.: From nosql accumulo to newsql graphulo: Design and utility of graph algorithms inside a bigtable database. In: 2016 IEEE on High Performance Extreme Computing Conference (HPEC), pp. 1–9. IEEE (2016)
8. Grolinger, K., Higashino, W.A., Tiwari, A., Capretz, M.A.: Data management in cloud environments: Nosql and newsql data stores. J. Cloud Comput. **2**(1), 22 (2013)
9. Gadepally, V., Bolewski, J., Hook, D., Hutchison, D., Miller, B., Kepner, J.: Graphulo: Linear algebra graph kernels for nosql databases. In: 2015 IEEE International on Parallel and Distributed Processing Symposium Workshop (IPDPSW), pp. 822–830. IEEE (2015)
10. Kepner, J., Bader, D., Buluç, A., Gilbert, J., Mattson, T., Meyerhenke, H.: Graphs, matrices, and the graphblas: seven good reasons. Procedia Comput. Sci. **51**, 2453–2462 (2015)
11. Weale, T., Gadepally, V., Hutchison, D., Kepner, J.: Benchmarking the graphulo processing framework. In: 2016 IEEE on High Performance Extreme Computing Conference (HPEC), pp. 1–5. IEEE (2016)
12. Buluç, A., Gilbert, J.R.: Highly parallel sparse matrix-matrix multiplication, arXiv preprint arXiv:1006.2183 (2010)
13. Kepner, J., Gilbert, J.: Graph algorithms in the language of linear algebra. SIAM, Philadelphia (2011)
14. Hutchison, D., Kepner, J., Gadepally, V., Fuchs, A.: Graphulo implementation of server-side sparse matrix multiply in the accumulo database. In: 2015 IEEE on High Performance Extreme Computing Conference (HPEC), pp. 1–7. IEEE (2015)
15. Akbudak, K., Selvitopi, O., Aykanat, C.: Partitioning models for scaling parallel sparse matrix-matrix multiplication. ACM Trans. Parallel Comput. (TOPC) **4**(3), 13 (2018)
16. Bader, D., Madduri, K., Gilbert, J., Shah, V., Kepner, J., Meuse, T., Krishnamurthy, A.: Designing scalable synthetic compact applications for benchmarking high productivity computing systems. Cyberinfrastruct. Technol. Watch **2**, 1–10 (2006)
17. Shvachko, K., Kuang, H., Radia, S., Chansler, R.: The hadoop distributed file system. In: 2010 IEEE 26th symposium on Mass storage systems and technologies (MSST), pp. 1–10. IEEE (2010)
18. Hunt, P., Konar, M., Junqueira, F.P., Reed, B.: Zookeeper: Wait-free coordination for internet-scale systems. In: USENIX annual technical conference, vol. 8, p. 9 (2010)
19. Wang, E., Zhang, Q., Shen, B., Zhang, G., Lu, X., Wu, Q., Wang, Y.: Intel math kernel library. In: High-Performance Computing on the Intel® Xeon Phi, pp. 167–188. Springer, New York (2014)

20. Patwary, M.M.A., Satish, N.R., Sundaram, N., Park, J., Anderson, M.J., Vadlamudi, S.G., Das, D., Pudov, S.G., Pirogov, V.O., Dubey, P.: Parallel efficient sparse matrix-matrix multiplication on multicore platforms. In: International Conference on High Performance Computing, pp. 48–57. Springer, New York (2015)
21. Gremse, F., Hofter, A., Schwen, L.O., Kiessling, F., Naumann, U.: GPU-accelerated sparse matrix-matrix multiplication by iterative row merging. SIAM J. Sci. Comput. **37**(1), C54–C71 (2015)
22. Akbudak, K., Aykanat, C.: Exploiting locality in sparse matrix-matrix multiplication on many-core architectures. IEEE Trans. Parallel Distrib. Syst. **28**(8), 2258–2271 (2017)
23. Heroux, M.A., Bartlett, R.A., Howle, V.E., Hoekstra, R.J., Hu, J.J., Kolda, T.G., Lehoucq, R.B., Long, K.R., Pawlowski, R.P., Phipps, E.T.: An overview of the trilinos project. ACM Trans. Math. Softw. (TOMS) **31**(3), 397–423 (2005)
24. Buluç, A., Gilbert, J.R.: The combinatorial blas: design, implementation, and applications. Int. J. High Perform. Comput. Appl. **25**(4), 496–509 (2011)
25. Buluç, A., Gilbert, J.R.: Parallel sparse matrix-matrix multiplication and indexing: implementation and experiments. SIAM J. Sci. Comput. **34**(4), C170–C191 (2012)
26. Akbudak, K., Aykanat, C.: Simultaneous input and output matrix partitioning for outer-product-parallel sparse matrix-matrix multiplication. SIAM J. Sci. Comput. **36**(5), C568–C590 (2014)
27. Catalyurek, U., Aykanat, C.: A hypergraph-partitioning approach for coarse-grain decomposition. In: Proceedings of the 2001 ACM/IEEE Conference on Supercomputing, pp. 28–28. ACM (2001)
28. Karypis, G.: Multilevel algorithms for multi-constraint hypergraph partitioning, tech. rep., MINNESOTA UNIV MINNEAPOLIS DEPT OF COMPUTER SCIENCE (1999)
29. Karypis, G., Kumar, V.: Metis—unstructured graph partitioning and sparse matrix ordering system, version 2.0 (1995)
30. Chevalier, C., Pellegrini, F.: Pt-scotch: a tool for efficient parallel graph ordering. Parallel Comput. **34**(6–8), 318–331 (2008)
31. Bejeck, B.: Getting Started with Google Guava. Packt Publishing Ltd, Birmingham (2013)
32. Karypis, G., Kumar, V.: Multilevelk-way partitioning scheme for irregular graphs. J. Parallel Distrib. Comput. **48**(1), 96–129 (1998)
33. Liu, W., Vinter, B.: An efficient GPU general sparse matrix-matrix multiplication for irregular data. In: 2014 IEEE 28th International on Parallel and Distributed Processing Symposium, pp. 370–381. IEEE (2014)
34. McCourt, M., Smith, B., Zhang, H.: Sparse matrix-matrix products executed through coloring. SIAM J. Matrix Anal. Appl. **36**(1), 90–109 (2015)
35. D'Alberto, P., Nicolau, A.: R-kleene: a high-performance divide-and-conquer algorithm for the all-pair shortest path for densely connected networks. Algorithmica **47**(2), 203–213 (2007)
36. Ordonez, C.: Optimization of linear recursive queries in SQL. IEEE Trans. Knowl. Data Eng. **22**(2), 264–277 (2010)
37. Ordonez, C., Zhang, Y., Cabrera, W.: The gamma matrix to summarize dense and sparse data sets for big data analytics. IEEE Trans. Knowl. Data Eng. **28**(7), 1905–1918 (2016)
38. Linden, G., Smith, B., York, J.: Amazon. com recommendations: item-to-item collaborative filtering. IEEE Internet Comput. **7**(1), 76–80 (2003)
39. Davis, T.A., Hu, Y.: The university of florida sparse matrix collection. ACM Trans. Math. Softw. (TOMS) **38**(1), 1 (2011)
40. Bell, N., Dalton, S., Olson, L.N.: Exposing fine-grained parallelism in algebraic multigrid methods. SIAM J. Sci. Comput. **34**(4), C123–C152 (2012)
41. Li, H., Li, K., Peng, J., Hu, J., Li, K.: An efficient parallelization approach for large-scale sparse nonnegative matrix factorization using kullback-leibler divergence on multi-GPU. In: IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), 2017, pp. 511–518. IEEE (2017)
42. Li, H., Li, K., Peng, J., Li, K.: Cusnmf: A sparse non-negative matrix factorization approach for large-scale collaborative filtering recommender systems on multi-GPU. In: 2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC), pp. 1144–1151. IEEE (2017)
43. Kannan, R., Ballard, G., Park, H.: Mpi-faun: an MPI-based framework for alternating-updating nonnegative matrix factorization. IEEE Trans. Knowl. Data Eng. **30**(3), 544–558 (2018)

44. Lee, D.D., Seung, H.S.: Algorithms for non-negative matrix factorization. In: Advances in neural information processing systems, pp. 556–562 (2001)