Lawrence Berkeley National Laboratory
LBL Publications

Title
Locality-aware and load-balanced static task scheduling for MapReduce

Permalink
https://escholarship.org/uc/item/6r6821fx

Authors
Selvitopi, Oguz
Demirci, Gunduz Vehbi
Turk, Ata
et al.

Publication Date
2019

DOI
10.1016/j.future.2018.06.035

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Locality-aware load-balanced task scheduling for MapReduce☆

Oguz Selvitopi[a], Gunduz Vehbi Demirci[a], Ata Turk[b], Cevdet Aykanat[a,*]

[a]*Bilkent University, Computer Engineering Department, 06800, Ankara, TURKEY*
[b]*Boston University, ECE Department, Boston, MA 02215*

Abstract

Task scheduling for MapReduce jobs has been an active area of interest since the popularization of MapReduce framework. The objective of scheduling is to decrease the amount of data transfer during the shuffle phase via exploiting data locality. In the literature, generally only the scheduling of reduce tasks is considered with the assumption that scheduling of map tasks is already determined by the input data placement. However, in cloud or HPC deployments of MapReduce, the input data is located in a remote storage and scheduling map tasks gains importance. Here, we propose models for simultaneous scheduling of map and reduce tasks in order to improve data locality and balance the processors' loads in both map and reduce phases. Our approach is based on graph and hypergraph models which correctly encode the interactions between map and reduce tasks. Partitions on these graphs and hypergraphs are decoded to schedule map and reduce tasks. A two-constraint formulation utilized in these models enables balancing processors' loads in both map and reduce phases. The partitioning objective in the hypergraph models correctly encapsulates the minimization of data transfer when a local combine step is performed prior to shuffle, whereas the partitioning objective in the graph models achieve the same feat when a local combine is not performed. We show the validity of our scheduling on the MapReduce parallelizations of two important kernel operations–sparse matrix-vector multiplication (SpMV) and generalized sparse matrix-matrix multiplication (SpGEMM)–that are widely encountered in big data analytics and scientific computations. Compared to random scheduling, our models lead to tremendous savings in data transfer in the shuffle phase, leading up to 2.6x and 4.2x speedup for SpMV and SpGEMM, respectively.

Keywords: MapReduce, scheduling, locality, load balance, map task, reduce task

1. Introduction

MapReduce [1] simplifies the programming for large-scale data-parallel applications and greatly reduces the development effort by sparing the programmer from complex issues such as parallel execution, fault tolerance, data management, task scheduling, etc. Hadoop [2], an open source implementation of MapReduce, has been used in production environments of many big companies and is deployed on clusters that can scale up to tens of thousands of cores. Its generality, ease of use and scalability led to a wide acceptance and adoption in many fields.

A MapReduce job consists of map, shuffle and reduce phases which are carried out one after another by multiple parallel tasks that process data in parallel. The map tasks process the input data and emit $\langle key, value \rangle$ (KV) pairs. In the shuffle phase, the KV pairs are communicated and then they are sorted according to keys, thus grouping the values that belong to the same key. The reduce tasks then process the

☆This work was supported by The Scientific and Technological Research Council of Turkey (TUBITAK) under Grant EEEAG-115E212 and ICT COST Action IC1406 (cHiPSet).

*Corresponding author

Email addresses: reha@cs.bilkent.edu.tr (Oguz Selvitopi), gunduz.demirci@cs.bilkent.edu.tr (Gunduz Vehbi Demirci), atatürk@bu.edu (Ata Turk), aykanat@cs.bilkent.edu.tr (Cevdet Aykanat)

grouped values for keys and produce the final outputs belonging to keys. The tasks in a phase depend on the tasks in the preceding phase.

The performance of MapReduce jobs has been focus of interest in the literature. The studies that aim at improving the parallel performance of a MapReduce job generally either try to reduce data transfers during the shuffle phase [3–8] or try to balance the loads in the map and/or reduce phases [5, 9]. Task scheduling studies usually focus on only the assignment of reduce tasks with the belief that map scheduling is determined by the initial data distribution of the file system hosted on the MapReduce compute nodes. However, in cloud or high performance computing deployments of MapReduce this assumption is not valid. The input data often resides in a remote shared file system such as Lustre [10], or distributed object store such as Amazon S3 [11]. In such a setup, since all the data is loaded from a remote location, the scheduling of map tasks also becomes important.

In recent years, the MapReduce framework has attracted interest from the graph processing, machine learning, and scientific computing domains and there have been many studies towards parallelizing kernel operations in these fields using MapReduce. Examples include HAMA [12], Apache Mahout [13], MR-MPI [14] and [15]. In these domains, since the interactions among map and reduce tasks can be predetermined by a scan of the input datasets, and the applications often perform multiple iterations of MapReduce computations, intelligently scheduling map and reduce tasks can yield significant performance gains.

In this work we propose a task assignment mechanism that simultaneously schedules map and reduce tasks to improve performance of applications. We showcase the impact of our approach by improving the performance of two kernel operations: sparse matrix-vector multiplication (SpMV) and generalized sparse matrix-matrix multiplication (SpGEMM). SpMV is a common primitive that is encountered widely in numerical algebra [16] and iterative computations such as PageRank [17]. SpGEMM occurs in multigrid interpolation and restriction [18], linear programming [19], multi-source breadth first search [20], similarity join [21] and item-to-item collaborative filtering in recommendation systems [22]. The omnipresence of these kernel operations in machine learning, graph algorithms, and scientific computations make them attractive targets for performance optimization.

Our approach uses graph and hypergraph models tailored for these operations. These models' outputs are used as hash functions to distribute KV pairs to mappers and reducers, i.e., we make use of application-specific knowledge to schedule map and reduce tasks. This enables scheduling tasks in the map and reduce phases with the aim of balancing the loads of processors in these phases and decreasing the data transfer (volume) in the shuffle phase. The models' success of exploiting domain-specific knowledge in assigning tasks are validated with the experiments. Compared to random scheduling, the models lead to tremendous savings in data transfer in the shuffle phase, which leads up to 2.6x and 4.2x speedup for SpMV and SpGEMM, respectively.

The rest of the paper is organized as follows. The related work and background are given in Section 2. MapReduce parallelizations of SpMV and SpGEMM operations are respectively investigated in Section 4 and Section 5. Section 6 presents the experiments. Section 7 concludes.

2. Related work and Background

Scheduling jobs and tasks for MapReduce programs has been an active area of research since the popularization of MapReduce paradigm. Job scheduling [23–28] considers allocation and usage of the resources in case of multiple MapReduce jobs. Task scheduling, on the other hand, focuses on the assignment of map and reduce tasks regarding a single MapReduce job. Our work falls in the latter category, so we focus on the works in this category.

Task scheduling presents two challenges which are critical for parallel performance: balancing the load in map and reduce phases, and decreasing the communication in the shuffle phase. Both can be alleviated via various approaches depending on the environment and the application MapReduce is being realized. The approach proposed by [3] considers data locality for decreasing communication in the shuffle phase and schedules each reduce task to its center-of-gravity node. This node is determined by two main factors: network locations of this reduce task's feeders and the partitioning skew regarding this task. Similarly, the

authors in [4] argue the overhead of the large network traffic and exploit data locality on both map and reduce phases to decrease the network traffic. Data locality is achieved by considering factors related to virtual machine placement, properties of the MapReduce job being run and the system load. [5] proposes a locality-aware approach based on a cost model that schedules reduce tasks in order to decrease the amount of data transferred in the shuffle phase. This approach is similar to our work in the sense that it also makes use of hash functions in order to decrease the data transferred in the shuffle phase and balance the load in reduce phase. Our work uses the hash functions in a static manner where they are determined from the patterns inherent in the input data, while in [5] they are determined on-the-fly according to the key frequencies. Another locality-aware approach is studied by [6], in which a scheduler called LARTS makes use of the information about the network locations and partition sizes in the scheduling decisions. LARTS improves data locality by reduce task scheduling and hence is able to decrease the network traffic. In [7], the authors propose a method that monitors the execution of MapReduce jobs and schedules map and reduce tasks according to the pattern deduced. By doing so they are able to schedule tasks preserving locality hence able to decrease the amount of transferred data in the shuffle phase. Recently, the authors of [8] propose an algorithm to improve the data locality and further overlap local reduce and shuffle phases of MapReduce jobs. Another study [9] aims to balance the load in the reduce phase by collecting the key distribution of intermediate pairs and running an algorithm that utilizes this data to further make the scheduling decisions. The works in [29–31] all aim at decreasing communication overheads: [29] by overlapping map and shuffle phases, [30] by overlapping shuffle and reduce phases, and [31] with a barrier-less MapReduce framework. These studies do not consider data locality.

Most of these works perform dynamic scheduling and do not focus on improving the performance of a specific operation. Our approach is static, i.e., in a preprocessing stage we determine hash functions to exploit the target operation realized within the MapReduce paradigm. These mappings are then used to distribute key value pairs among mappers and reducers in the execution.

We realized the subject operations using MR-MPI library [14]. This library uses MPI for handling communication between processors and in that sense it is fast and flexible. However, these come at the expense of fault tolerance and redundancy, both of which may prove vital in a commodity cluster but are not of prime concern on high performance computing systems. The high performance computing systems, sometimes called tier-0 systems, are characterized with very high availability and they allow access to full resources, without any virtualization or whatsoever. As our focus in this work is such a system, we preferred MR-MPI for implementation.

MR-MPI library supports two basic data types on which the functions operate: $\langle Key, Value\rangle$ (KV) and $\langle Key, Multivalue\rangle$ (KMV). As the name suggests, a KMV pair stores all values related to a key while a KV pair stores a single one of them. The operations that are of interest to our work in this library are briefly described below:

- $map()$: Generates KV pairs.
- $reduce()$: Reduces KMV pairs to KV pairs.
- $collate()$: Communicates KV pairs and generates KMV pairs from them. Equivalent to MapReduce shuffle.
- $aggregate()$: Distributes KV pairs among processors. Necessitates communication.
- $convert()$: Creates KMV pairs from KV pairs in which the values belonging to the same key become a MultiValue.
- $add()$: Adds KV pairs of a MapReduce object to those of another.

These operations are used in our implementation. For more details on MR-MPI, see [14].

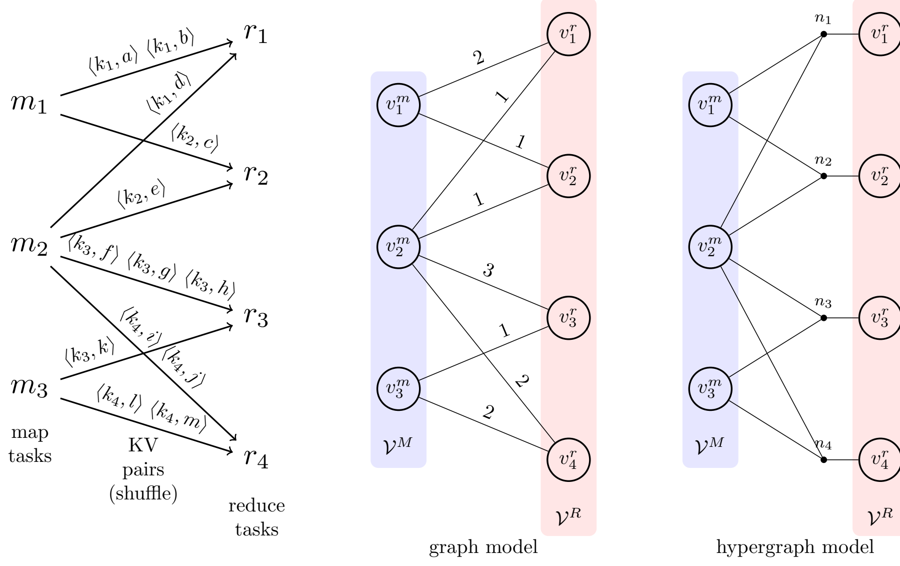

Figure 1: An example with three map and four reduce tasks, and the corresponding graph and hypergraph used to model them.

3. Modeling MapReduce applications

The map and reduce tasks in a MapReduce job can be scheduled with certain considerations in mind if the relations between map and reduce tasks are known apriori. These relations may be inferred from the target application's computational structure on the input data or the MapReduce job can be run beforehand to infer them. The latter case is particularly useful if the same MapReduce job will be executed multiple times. In this section, we show how the map and reduce tasks can be scheduled via graph and hypergraph partitioning models to address important issues such as load balancing and communication reduction.

Consider a set $\mathcal{M}$ of map tasks and a set $\mathcal{R}$ of reduce tasks, where the time to execute a map task $m_i \in \mathcal{M}$ and a reduce task $r_j \in \mathcal{R}$ is respectively denoted with $size(m_i)$ and $size(r_j)$. A KV pair is denoted with the tuple $\langle key, val\rangle$. The set of KV pairs generated by m_i is denoted by $kvp(m_i)$ and the set of KV pairs destined for r_j is denoted by $kvp(r_j)$. Note that it is assumed that the relation between map and reduce tasks is known, i.e., it is known that which map task produces value(s) for a certain key. The left of Figure 1 shows an example MapReduce job with three map and four reduce tasks. For example, $kvp(m_2) = \{\langle k_1,d\rangle, \langle k_2,e\rangle\langle k_3,f\rangle, \langle k_3,g\rangle, \langle k_3,h\rangle, \langle k_4,i\rangle, \langle k_4,j\rangle\}$ and $kvp(r_2) = \{\langle k_2,c\rangle, \langle k_2,e\rangle\}$.

3.1. Formation

In the bipartite graph $\mathcal{G} = (\mathcal{V}^M \cup \mathcal{V}^R, \mathcal{E})$ proposed to model a given MapReduce job, the map and reduce tasks are represented by different vertex sets. There exists a vertex $v_i^m \in \mathcal{V}^M$ for map task $m_i \in \mathcal{M}$ and a vertex $v_j^r \in \mathcal{V}^R$ for reduce task $r_j \in \mathcal{R}$. There exists an edge $(v_i^m, v_j^r) \in \mathcal{E}$ if the map task represented by v_i^m generates at least one KV pair for the reduce task represented by v_j^r, i.e., $kvp(m_i) \cap kvp(r_j) \neq \emptyset$. The edges represent the dependency of the reduce tasks to the map tasks. The graph in the middle of Figure 1 models the MapReduce job in the left of the same figure. For example, there exists an edge between v_2^m and v_3^r since m_2 generates the KV pairs $\langle k_3,f\rangle$, $\langle k_3,g\rangle$, $\langle k_3,h\rangle$, which are to be reduced by r_3.

The hypergraph $\mathcal{H} = (\mathcal{V}^M \cup \mathcal{V}^R, \mathcal{N})$ proposed to model a given MapReduce job is the same with $\mathcal{G}$ in terms of vertex sets and what they represent. The difference between $\mathcal{H}$ and $\mathcal{G}$ lies in representing the dependencies, which is achieved by nets in $\mathcal{H}$ as opposed to the edges in $\mathcal{G}$. There exists a net $n_j \in \mathcal{N}$ for each reduce task $r_j \in \mathcal{R}$ and this net connects the vertex that represents the reduce task r_j and the vertices corresponding to the map tasks that generate at least one KV pair for r_j. The vertices connected by n_j is denoted by

$$Pins(n_j) = \{v_i^m : kvp(m_i) \cap kvp(r_j) \neq \emptyset\} \cup \{v_j^r\}.$$

Compared to the edges, the nets are better means for capturing multi-way dependencies. The hypergraph in the right of Figure 1 models the MapReduce job in the left of the same figure. For example, n_3 connects v_2^m, v_3^m and v_3^r since the map tasks m_2 and m_3 respectively generates the KV pairs $\langle k_3, f\rangle$, $\langle k_3, g\rangle$, $\langle k_3, h\rangle$ and $\langle k_3, k\rangle$, which are to be reduced by r_3. Hence, $Pins(n_3) = \{v_2^m, v_3^m, v_3^r\}$.

In both $\mathcal{G}$ and $\mathcal{H}$, a two-constraint formulation is used for vertex weights to enable load balancing in two distinct computational phases of map and reduce. The weights of a vertex $v_i^m \in \mathcal{V}^M$ are assigned as

$$w_1(v_i^m) = size(m_i)$$
$$w_2(v_i^m) = 0$$

in order to balance the processors' loads in the map phase. The weights of a vertex $v_j^r \in \mathcal{V}^R$ are assigned as

$$w_1(v_j^r) = 0$$
$$w_2(v_j^r) = size(r_j)$$

in order to balance the processors' loads in the reduce phase. The cost of edge (v_i^m, v_j^r) in $\mathcal{G}$ is assigned the number of KV pairs generated by m_i for r_j to encapsulate the volume of communicated data, i.e., $c(v_i^m, v_j^r) = |kvp(m_i) \cap kvp(r_j)|$. The cost of net n_j in $\mathcal{H}$ is assigned as 1, i.e., $c(n_j) = 1$, reasons of which will be clear shortly.

3.2. Partitioning

$\mathcal{G}/\mathcal{H}$ is partitioned into K parts to obtain $\Pi(\mathcal{G}/\mathcal{H}) = \{\mathcal{V}_1 = \mathcal{V}_1^M \cup \mathcal{V}_1^R, \ldots, \mathcal{V}_K = \mathcal{V}_K^M \cup \mathcal{V}_K^R\}$. The obtained partition is used to schedule map and reduce tasks in a given MapReduce job. For convenience, the partitions on $\mathcal{V}^M$ and $\mathcal{V}^R$ are denoted by Π^M and Π^R, respectively. Without loss of generality we assume that the vertex part $\mathcal{V}_k$ is associated with processor P_k. The vertices in $\mathcal{V}_k^M$ are decoded as assigning the map tasks represented by these vertices to P_k. The vertices in $\mathcal{V}_k^R$ are decoded as assigning the reduce tasks represented by these vertices to P_k. In partitioning $\mathcal{G}$ and $\mathcal{H}$, the partitioning objective of minimizing the cutsize corresponds to decreasing communication volume in the *shuffle* phase, whereas the partitioning constraint of balancing part weights corresponds to balancing loads in *map* and *reduce* phases.

The correct encapsulation of communication volume in the shuffle phase depends on the specifics of the implementation. A processor may choose to introduce an additional *local reduction* phase for further reduction of communication volume at the expense of more computation. The idea is that if a processor generates multiple values for a specific key whose reduce task belongs to another processor, it can either send them all or it can reduce them first and then send a single KV pair. The former incurs more communication and the latter incurs less communication at the expense of additional computation. In the example in Figure 1, assume that the map tasks m_2 and m_3 are both assigned to processor P_k, whereas the reduce task r_3 is assigned to some other processor. Regarding the values generated for key k_3, P_k has two options:

(i) sending them all to the processor responsible for r_3,

(ii) first reducing the values for k_3 and then sending a single KV pair to the target processor.

The graph model correctly encapsulates the communication volume incurred in the shuffle phase if local reduction is not performed (case (i)). This is because the graph model represents KV pair(s) produced by a certain map task for a specific key with a different edge. On the other hand, the hypergraph model correctly encapsulates the communication volume if local reduction is performed (case (ii)). This is because the locally reduced values for a specific key are represented with the pins of a single net and in the partitioning the connectivity metric [32] is utilized. Unit net costs are required here since for any key, a processor may contribute at most a single KV pair due to local reduction, i.e., uniform data size.

An additional issue regarding the partitioning models and the optional local reduce is the computational load balance in the reduce phase. Recall that in both models, the vertex weights regarding the reduce phase were set to the number KV pairs generated for the respective reduce tasks. If local reduction is not performed, then these weights correctly represent the amount of computation in the reduce phase and

165 balancing part weights in the partitioning process balances processors' computations in the reduce phase. If local reduction is performed, however, both models overestimate the computations in the reduce phase as some of the KV pairs will be reduced locally. It is not possible to infer the exact amount of computation in the reduce phase if the optional local reduce is performed as this information depends on the distribution of data–the goal of the partitioning models. Hence, it is not possible to utilize the correct vertex weights in the 170 models for this case. Interestingly, however, the objective of minimizing cutsize in the graph model strongly correlates to the assigned vertex weights since the minimization of the cutsize translates to the maximization of the number of internal edges, which in turn implies the maximization of the number of KV pair reductions in the reduce phase, rather than in the local reduce. This correlation exists in the hypergraph model as well, but it is more loose.

175 4. Sparse matrix-vector multiplication

We first briefly review the parallel algorithm for sparse matrix-vector multiplication (SpMV) and discuss the graph and hypergraph models in the context of MapReduce framework. Then, we describe the MapReduce implementation of SpMV and show how to use the partitions obtained by the graph/hypergraph models to assign map and reduce tasks to processors.

180 *4.1. Parallel algorithm and MapReduce*

We focus on one-dimensional columnwise parallelization of $y = Ax$, where A is permuted into a block structure as:

$$\begin{bmatrix} A_{11} & \dots & A_{1K} \\ \vdots & \ddots & \vdots \\ A_{K1} & \dots & A_{KK} \end{bmatrix}.$$

Here, K is the number of processors, A is a square $n \times n$ matrix, and x and y are dense vectors of size n. The size of submatrix block $A_{k\ell}$ is $n_k \times n_\ell$. $a_{i,*}$ and $a_{*,j}$ respectively denote row i and column j of A and $a_{i,j}$ denotes the nonzero element at row i and column j of A. To denote the number of nonzeros in a row, column, or a (sub)matrix, we use the function $nnz(\cdot)$.

185 In the columnwise partitioning, processor P_k is held responsible for the computations related to kth column stripe $[A_{1k}^T \dots A_{Kk}^T]^T$ of A, whose size is $n \times n_k$. This columnwise partitioning of A induces a partition on the input vector x as well, where P_k stores the subvector x_k.

The parallel algorithm that results from the columnwise partitioning of A is called the *column-parallel* algorithm for SpMV and its basic steps for processor P_k are as follows:

190
1. For each submatrix block $A_{\ell k}$ owned by P_k, P_k computes $y_\ell^k = A_{\ell k} x_k$ for $1 \leq \ell \leq K$. Here, it is assumed that the submatrix blocks are ordered in such a way that the resulting elements from the multiplication containing a specific submatrix block $A_{\ell k}$ belong to P_ℓ. In other words, the sparse subvector y_ℓ^k contains the elements that are computed by P_k, but belong to P_ℓ ($\ell \neq k$). The elements in these subvectors are called the partial results. As P_k's portion of y, it computes $y_k^k = A_{kk} x_k$ and 195 sets $y_k = y_k^k$.
2. The partial results are communicated to aggregate y_k^ℓ at P_k with the aim of computing the final results of the elements in y_k. To do so, P_k receives the partial results computed by P_ℓ ($\ell \neq k$), i.e., y_k^ℓ. Note that P_k only needs interaction with processors that have partial results to send it.
3. In the final step, P_k sums the partial results by $y_k = y_k + y_k^\ell$ for each P_ℓ.

200 We assume there is no overlap of communication and computation in the above algorithm and the steps proceed in a similar manner to BSP model of computation. In addition, we retain the flexibility of having different partitions on input and output vectors in SpMV. In other words, it is not enforced for a processor to store the ith element of y if it stores the ith element of x. In the column-parallel algorithm, there is a single communication phase between two computational phases. Considering the two computational phases,

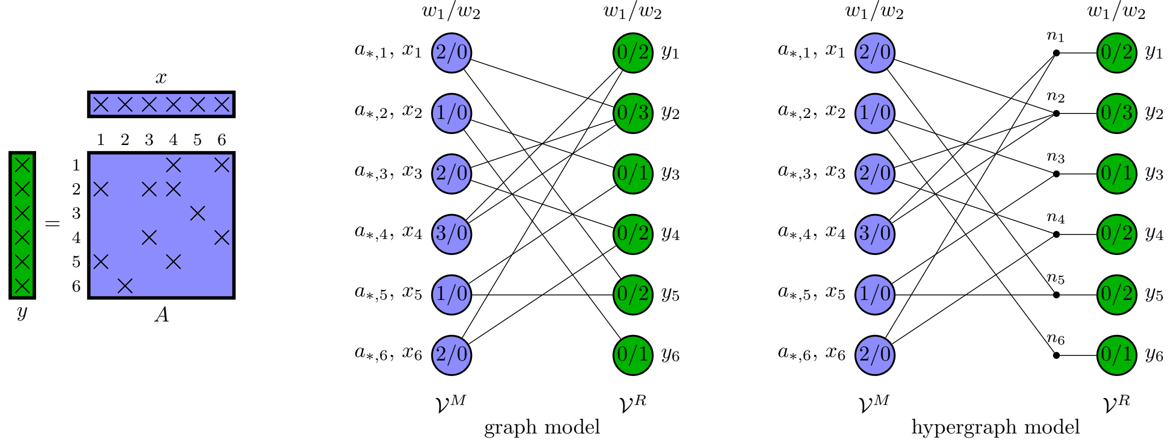

Figure 2: An example SpMV, and graph and hypergraph models to represent it. The numbers inside the vertices indicate the two weights associated with them. Vectors and matrices are color-matched with the vertices they are represented with.

the first computation phase is likely to be more expensive compared to the second one. However, there may be other linear vector operations that involve vectors x and y. For this reason, it is a good practice to balance the vector elements owned by the processors (i.e., number of x and/or y elements) besides the nonzeros of A owned by each processor. In this way, the processors' loads in each computational phase can be balanced.

In the parallel algorithm above, there are n map and n reduce tasks, i.e., $|\mathcal{M}| = |\mathcal{R}| = n$. A map task m_j is defined as the multiplication of $a_{*,j}$ with x_j (performed in the first step of the column-parallel algorithm). In the rest of the paper, we use x_i/y_i to denote a single element of x/y, rather than the portion owned by the processor. For each nonzero in $a_{*,j}$, the map task m_j generates a single KV pair, hence, $kvp(m_j) = \{\langle y_i, a_{i,j} * x_j\rangle : a_{i,j} \neq 0 \text{ for } 1 \leq i \leq n\}$. A reduce task r_i is defined as the summation of partial results generated for y_i (performed in the third step of the column-parallel algorithm). The KV pairs destined for r_i is given by $kvp(r_i) = \{\langle y_i, a_{i,j} * x_j\rangle : a_{i,j} \neq 0 \text{ for } 1 \leq j \leq n\}$. The size of m_j is proportional to the number of nonzeros in the respective column of A, hence, $size(m_j) = nnz(a_{*,j})$, whereas the size of r_i is proportional to the number of nonzeros in the respective row of A, hence, $size(r_i) = nnz(a_{i,*})$.

The formation and partitioning of the graph/hypergraph for efficient parallelization of column-parallel SpMV in MapReduce framework follow the methodology described in Sections 3.1 and 3.2, respectively. All edges in $\mathcal{G}$ have unit weights since m_j generates a single KV pair for r_i if $a_{i,j} \neq 0$, and it does not generate anything, otherwise. All nets in $\mathcal{H}$ have unit costs as well. The K-way partitions $\Pi^M(\mathcal{G}/\mathcal{H})$ and $\Pi^R(\mathcal{G}/\mathcal{H})$ are used to schedule map and reduce tasks, details of which will be described in the following section. Figure 2 shows an SpMV operation and its representation with the graph and hypergraph models.

4.2. Implementation

We describe the parallelization of the SpMV operation under MapReduce paradigm. The parallelization is realized using the MR-MPI library [14]. We first give the MapReduce-based parallelization, and then explain how to assign the tasks to the processors in order to decrease the communication overhead in the shuffle phase and balance the loads of the processors in both map and reduce phases.

Algorithm 1 presents the MapReduce-based parallelization of SpMV. The SpMV operation is assumed to be repeated in an application-specific context and it is highlighted in gray in the algorithm. We omit the application-specific details and focus solely on the SpMV operation itself. Note that a similar routine is also used in [14] without explicit usage of any hash function. In the algorithm, A and x are distributed among the processors via *aggregate*() operation prior to performing SpMV and they are keyed according to column index j (line 2 and 3). *aggregate*() operation can take a hash function as input (whose role is going to be clarified shortly). The keys regarding A and x are added to y and they are converted to the KMV pairs

Algorithm 1: Sparse matrix-vector multiplication

Input: A, h_M, h_R

1 Set initial x
2 A.aggregate($\boldsymbol{h_M}$) ▷ **Key** j, **Value** $(i, a_{i,j})$
3 x.aggregate($\boldsymbol{h_M}$) ▷ **Key** j, **Value** x_j
4 Let y be an empty MapReduce object
5 **repeat**
    ▷ other computations... (on vectors, etc.)
6   y.add(x)
7   y.add(A)
8   y.convert()
    ▷ **IN**: **Key** j, **MultiValue** $[(i, a_{i,j})], x_j$
9   y.reduce()
    ▷ **OUT**: **Key** i, **Value** $y_i^j = a_{i,j}x_j$
    ▷ optional *local reduce*
10 y.convert()
11 y.reduce()
    ▷ communication phase (shuffle)
12 y.collate($\boldsymbol{h_R}$) ▷ **OUT**: **Key** i, **MultiValue** $[y_i^j]$
    ▷ **IN**: **Key** i, **MultiValue** $[y_i^j]$
13 y.reduce()
    ▷ **OUT**: **Key** i, **Value** $\sum_j y_i^j$
    ▷ other computations... (on vectors, etc.)
until *application-specific condition is met*

(lines 6–8). Then, the multiplication operations are performed via *reduce*(), in which the multiple values belonging to key j are reduced via multiplying value of each with x_j. The results of this operation are the partial results for y that are keyed by index i. y_i^j denotes the partial value generated by column j for y_i.
240 The operations up to this point constitute the first computational phase of the column-parallel algorithm.

The first computational phase is followed by an optional local reduce in which the partial results are summed locally (note that these summations do not compute the final values of y yet). The partial results are then communicated and KMV pairs are created accordingly, producing possible multiple y_i^j values for y_i (line 12). Notice that *collate*() also accepts a hash function as input.

245 Finally, the partial results are reduced and the final values of y are computed by simply summing them (line 13). The computation of final values constitutes the second computational phase of the column-parallel algorithm. Note that the first computational phase is the "map" phase even though a reduce call has been performed, as it emits KV pairs and is followed by a shuffle phase, which is in turn followed by a reduce operation to compute the final results.

We make use of the partitions $\Pi^M = \{\mathcal{V}_1^M, \ldots, \mathcal{V}_K^M\}$ and $\Pi^R = \{\mathcal{V}_1^R, \ldots, \mathcal{V}_K^R\}$ described earlier in order to achieve an efficient distribution of data and computations in Algorithm 1. Π^M and Π^R can be utilized as hash functions in the algorithm, which are respectively denoted with h_M and h_R. h_M is simply obtained from Π^M as

$$h_M(j : v_j^m \in \mathcal{V}_k^M) = P_k, \quad 1 \leq j \leq n \text{ and } 1 \leq k \leq K,$$

which allows distributing matrix columns, elements of x and the respective map tasks via *aggregate*() with

h_M as its input. Similarly, h_R is obtained from Π^R as

$$h_R(i : v_i^r \in \mathcal{V}_k^R) = P_k, \quad 1 \leq i \leq n \text{ and } 1 \leq k \leq K,$$

250 which allows distributing elements of y and the respective reduce tasks on them via *collate*()[1] with h_R as its input.

5. Sparse matrix-sparse matrix multiplication

The literature on parallelization of sparse matrix-sparse matrix multiplication of form $C = AB$ (SpGEMM) is more recent compared to that on SpMV. One of the recent promising studies on this subject is based
255 on parallelization with one-dimensional partitioning of input matrices (A and B) and outer product tasks via hypergraph models [33]. We first briefly review the parallel algorithm for SpGEMM and discuss the graph and hypergraph models in the context of MapReduce framework. Then, we describe the MapReduce implementation of SpGEMM and show how to use the partitions obtained by the graph/hypergraph models to assign map and reduce task to processors.

260 ### *5.1. Parallel algorithm and MapReduce*

We focus on one-dimensional partitioning of input matrices A and B, and two-dimensional partitioning of output matrix C. The matrices A and B are permuted into block structures as

$$\begin{bmatrix} A_{11} & \dots & A_{1K} \\ \vdots & \ddots & \vdots \\ A_{K1} & \dots & A_{KK} \end{bmatrix} \quad \text{and} \quad \begin{bmatrix} B_{11} & \dots & B_{1K} \\ \vdots & \ddots & \vdots \\ B_{K1} & \dots & B_{KK} \end{bmatrix},$$

respectively, where A is an $m \times n$ and B is an $n \times p$ matrix. Processor P_k is held responsible from the outer products in kth column stripe $A_k^c = [A_{1k}^T \dots A_{Kk}^T]^T$ of A and the respective kth row stripe $B_k^r = [B_{k1} \dots B_{kK}]$ of B. An outer product performed between a column x of A and the respective row x of B is denoted with $a_{*,x} \otimes b_{x,*}$ It is assumed if P_k stores $a_{*,x}$, it also stores $b_{x,*}$ in order to avoid redundant communication (i.e.,
265 a conformal partition of A and B). The described partitions of A and B do not induce a natural partition of C since the outer products performed by a processor may contribute to any nonzero in C. In other words, there is no locality in access to elements of C.

The parallel algorithm that results from the columnwise partitioning of A and the rowwise partitioning of B is called the *outer-product–parallel* algorithm for SpGEMM and its basic steps for P_k are as follows:

270 1. For each column x in column stripe A_k^c (and hence each row in row stripe B_k^r), P_k computes the outer product $C^x = a_{*,x} \otimes b_{x,*}$. This outer product generates partial result(s) for the elements of C, denoted with C^x. There exists a complete partial result set for each such outer product. Observe that two such different partial result set C^x and C^y may contain partial results for the same element of C. P_k may sum them by $\sum_x C^x$ or it may not do so and leave them as they are. If $c_{i,j}$ belongs to P_k, it sets
275 the initial value of this nonzero by $c_{i,j} = c_{i,j}^k$.
2. The partial results are communicated to aggregate each $c_{i,j}^\ell$ at P_k with the aim of computing the final result of this nonzero whose accumulation responsibility is given to P_k. To do so, P_k receives each such partial result $c_{i,j}^\ell$ computed by P_ℓ ($\ell \neq k$).
3. In the final step, P_k sums the partial results by $c_{i,j} = c_{i,j} + c_{i,j}^\ell$ for each P_ℓ.

[1] *collate*() is actually an *aggregate*() followed by a *convert*().

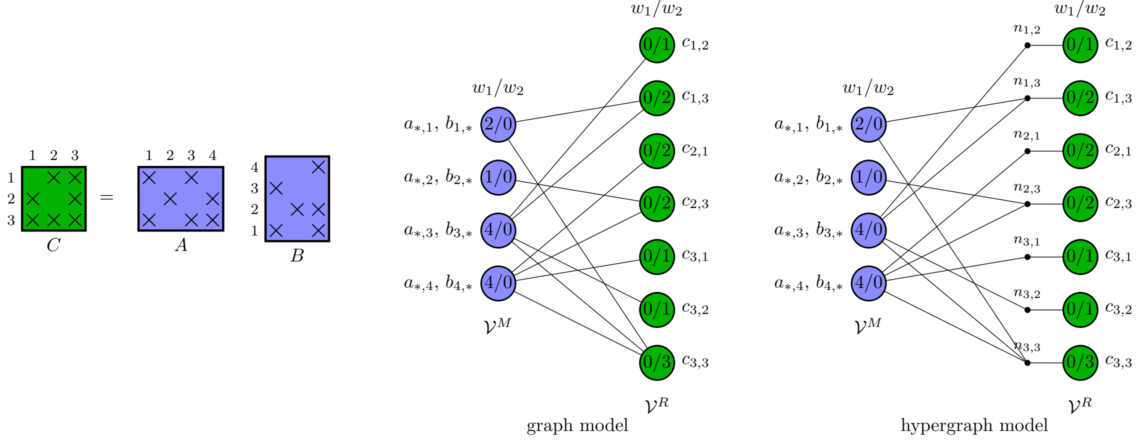

Figure 3: An example SpGEMM, and graph and hypergraph models to represent it. The numbers inside the vertices indicate the two weights associated with them. Matrices are color-matched with the vertices they are represented with.

As in the column-parallel SpMV, we assume no overlap of communication and computation and the steps proceed in a similar manner to the BSP model. Notice the resemblance of outer-product–parallel algorithm for SpGEMM to the column-parallel algorithm for SpMV. The outer-product–parallel SpGEMM has the same skeleton with the column-parallel SpMV, where there exists a single communication phase between two computational phases. Here too the first computational phase is likely to be more expensive compared to the second one.

In the parallel algorithm above, there are n map tasks and $nnz(C)$ reduce tasks, i.e., $|\mathcal{M}| = n$ and $|\mathcal{R}| = nnz(C)$. A map task m_x is defined as the outer product $a_{*,x} \otimes b_{x,*}$ (performed in the first step of the outer-product–parallel algorithm). For each $c_{i,j} \in C^x$, m_x generates a single KV pair, hence, $kvp(m_x) = \{\langle c_{i,j}, a_{i,x} * b_{x,j}\rangle : a_{i,x}, b_{x,j} \neq 0 \text{ for } 1 \leq i \leq m \text{ and } 1 \leq j \leq p\}$. A reduce task $r_{i,j}$ is defined as the summation of partial results generated for $c_{i,j}$ (performed in the third step of the outer-product–parallel algorithm). The KV pairs destined for $r_{i,j}$ is given by

$$kvp(r_{i,j}) = \{\langle c_{i,j}, a_{i,x} * b_{x,j}\rangle : a_{i,x}, b_{x,j} \neq 0 \text{ for } 1 \leq x \leq n\}.$$

The size of m_x is proportional to the number of operations performed in the respective outer product, hence, $size(m_x) = nnz(a_{*,x}) \times nnz(b_{x,*})$, whereas the size of $r_{i,j}$ is proportional to the number of outer products that generate a partial result for $c_{i,j}$, hence, $size(r_{i,j}) = |\{C^x : c_{i,j} \in C^x\}|$.

In the graph and hypergraph models used to parallelize the outer-product–parallel SpGEMM in the MapReduce framework, all edges and nets have unit costs, respectively. The K-way partitions $\Pi^M(\mathcal{G}/\mathcal{H})$ and $\Pi^R(\mathcal{G}/\mathcal{H})$ are used to schedule map and reduce tasks, details of which will be described in the following section. Figure 3 shows an SpGEMM operation and its representation with the graph and hypergraph models.

5.2. Implementation

We describe the parallelization of the SpGEMM operation under MapReduce paradigm. We first give the MapReduce-based parallelization, and then explain how to assign the tasks to the processors in order to decrease the communication overhead in the shuffle phase and balance the loads of the processors in both map and reduce phases.

Algorithm 2 presents the MapReduce-based parallelization of SpGEMM. The algorithm solely focuses on parallelizing SpGEMM and ignores the application-specific issues. In the algorithm, the matrices A and B are distributed among the processors via $aggregate()$ operation and matrix A is keyed according to column index x and matrix B is keyed according to row index x (lines 1 and 2). The values contained in these keys are the nonzero elements and additional information regarding row/column indices and identification

Algorithm 2: Sparse matrix-sparse matrix multiplication

Input: A, B, h_M, h_R

1 A.aggregate($\boldsymbol{h_M}$) ▷ **Key** x, **Value** $(i, a_{i,x}, \text{`}c\text{'})$
2 B.aggregate($\boldsymbol{h_M}$) ▷ **Key** x, **Value** $(j, b_{x,j}, \text{`}r\text{'})$
3 Let C be an empty MapReduce object
4 **repeat**
▷ other computations...
5 C.add(A)
6 C.add(B)
7 C.convert()
▷ **IN Key** j, **MultiValue** $[(i, a_{i,x}, \text{`}c\text{'}), \ldots, (j, b_{x,j}, \text{`}r\text{'}), \ldots]$
8 C.reduce()
▷ **OUT**: **Key** (i, j), **Value** $c^x_{i,j} = a_{i,x} b_{x,j}$
▷ optional *local reduce*
9 C.convert()
10 C.reduce()
▷ communication phase (shuffle)
11 C.collate($\boldsymbol{h_R}$) ▷ **OUT**: **Key** (i, j), **MultiValue** $[c^x_{i,j}]$
▷ **IN Key** (i, j), **MultiValue** $[c^x_{i,j}]$
12 C.reduce()
▷ **OUT Key** (i, j), **Value** $\sum_x c^x_{i,j}$
▷ other computations...
until *application-specific condition is met*

of matrices. C is initially empty and it is filled with the KV pairs of A and B (lines 5 and 6). These pairs
305 are converted to KMV pairs next (line 7). Then, the multiplication operations are performed via *reduce*(), in which each element of column x of A is multiplied with each element of row x of B, i.e., $a_{*,x} \otimes b_{x,*}$. The results of this outer product are the partial results for C that are keyed with the row and column pair indices, (i, j), in order to achieve a two-dimensional partitioning of C. This first computational phase is followed by an optional local reduce in which the partial results are summed. Next follows a *collate*()
310 in which the partial results are communicated and KMV pairs are created accordingly, producing possible multiple $c^x_{i,j}$ values for $c_{i,j}$ (line 11). The final step of SpGEMM corresponds to the second computational phase of the outer-product–parallel algorithm and it contains the reduction of $c_{i,j}$ via summation (line 12). Observe that similar to SpMV, the functions *aggregate*() and *collate*() take hash functions as their input, which we exploit to achieve task assignments in Algorithm 2.

We make use of the partitions $\Pi^M = \{\mathcal{V}^M_1, \ldots, \mathcal{V}^M_K\}$ and $\Pi^R = \{\mathcal{V}^R_1, \ldots, \mathcal{V}^R_K\}$ obtained by the graph/hypergraph models and use them as hash functions in order to achieve an efficient distribution of data and computations, as done for SpMV. h_M is obtained from Π^M as

$$h_M(x : v^m_x \in \mathcal{V}^M_k) = P_k, \quad 1 \leq x \leq n \text{ and } 1 \leq k \leq K,$$

and h_R is obtained from Π^R as

$$h_R((i, j) : v^r_{i,j} \in \mathcal{V}^R_k) = P_k, \quad 1 \leq i \leq m,\, 1 \leq j \leq p \text{ and } 1 \leq k \leq K.$$

315 h_M is used along with *aggregate*() to obtain a columnwise distribution of A, a rowwise distribution of B and

Table 1: Matrices used in the experiments.

operation	matrix	number of rows/columns	number of nonzeros	row/column degree average	row/column degree maximum
SpMV $y = Ax$	333SP	3,712,815	22,217,266	6.0	28
	adaptive	6,815,744	27,248,640	4.0	4
	circuit5M_dc	3,523,317	19,194,193	5.5	27
	CurlCurl_4	2,380,515	26,515,867	11.1	13
	delaunay_n23	8,388,608	50,331,568	6.0	28
	germany_osm	11,548,845	24,738,362	2.1	13
	hugetrace-00000	4,588,484	13,758,266	3.0	3
	rajat31	4,690,002	20,316,253	4.3	1252
	rgg_n_2_24_s0	16,777,216	265,114,400	15.8	40
	Transport	1,602,111	23,500,731	14.7	15
SpGEMM $C = AA^T$	crashbasis	160,000	1,750,416	10.9	18
	crystm03	24,696	583,770	23.6	27
	dawson5	51,537	1,010,777	19.6	33
	ia2010	216,007	1,021,170	4.7	49
	kim1	38,415	933,195	24.3	25
	lhr71	70,304	1,528,092	21.7	63
	olesnik0	88,263	744,216	8.4	11
	rgg_n_2_17_s0	131,072	1,457,506	11.1	28
	struct3	53,570	1,173,694	21.9	27
	xenon1	48,600	1,181,120	24.3	27

a distribution of map tasks. h_R, on the other hand, is used along with *collate*() to obtain a two-dimensional nonzero-based distribution of C and a distribution of reduce tasks.

6. Experiments

We test a total of six schemes in our experiments:

- `RN`: The tasks in the first and the second computation phases are distributed among the processors in a random manner and local reduce is not performed (i.e., lines 10 and 11 in Algorithm 1 and lines 9 and 10 in Algorithm 2 are not executed). This scheme is equivalent to using the default hash function in the MapReduce implementation in Algorithms 1 and 2 for aggregating data.
- `RNr`: Similar to `RN`, but with the optional local reduce.
- `GR`: The tasks in the first and the second computation phases are distributed among the processors with the graph models with the aim of decreasing communication overhead under the load balance constraint. Local reduce is not performed in this scheme.
- `GRr`: Similar to `GR`, but with the optional local reduce.
- `HY`: The tasks in the first and the second computation phases are distributed among the processors with the hypergraph models with the aim of decreasing communication overhead under the load balance constraint. Local reduce is not performed in this scheme.
- `HYr`: Similar to `HY`, but with the optional local reduce.

The experiments are performed on an IBM System x iDataPlex machine (dx360M4). A node on this machine consists of 16 cores (two 8-core Intel Xeon E5 processors) with 2.7 GHz clock frequency and 32 GB memory. The nodes are connected with an Infiniband non-blocking tree network topology. We tested for $32, 64, \ldots, 1024$ processors. Recall that these are also the number of parts in partitioning models.

Table 2: Volume, imbalance and runtime averages for SpMV (volume in megabytes and time in milliseconds).

		actual values						normalized within scheme			normalized wrt RN and RNr			
K	scheme	RNr	RN	GRr	GR	HYr	HY	RN/RNr	GR/GRr	HY/HYr	GRr/RNr	HYr/RNr	GR/RN	HY/RN
32	%imb-map	0.5	0.5	0.7	0.7	0.9	0.9	1.00	1.00	1.00	1.4	2.0	1.4	2.0
	%imb-reduce	0.5	0.5	1.0	1.0	1.6	1.6	1.00	1.00	1.00	1.9	3.2	1.9	3.2
	volume	406.3	448.9	0.6	1.0	0.5	1.6	1.10	1.60	2.91	0.002	0.001	0.002	0.004
	time	1.26	0.93	0.61	0.59	0.61	0.60	0.74	0.96	0.99	0.49	0.48	0.64	0.65
64	%imb-map	0.8	0.8	1.6	1.6	0.9	0.9	1.00	1.00	1.00	2.1	1.2	2.1	1.2
	%imb-reduce	0.7	0.7	2.1	2.1	2.0	2.0	1.00	1.00	1.00	2.9	2.7	2.9	2.7
	volume	433.7	456.2	0.9	1.5	0.8	2.4	1.05	1.59	2.88	0.002	0.002	0.003	0.005
	time	0.64	0.47	0.33	0.32	0.33	0.32	0.74	0.97	0.97	0.52	0.52	0.69	0.68
128	%imb-map	1.0	1.0	3.2	3.2	1.3	1.3	1.00	1.00	1.00	3.1	1.2	3.1	1.2
	%imb-reduce	1.2	1.2	3.6	3.6	2.7	2.7	1.00	1.00	1.00	2.9	2.2	2.9	2.2
	volume	448.5	459.9	1.4	2.2	1.2	3.5	1.03	1.59	2.84	0.003	0.003	0.005	0.008
	time	0.34	0.25	0.20	0.18	0.20	0.19	0.71	0.91	0.93	0.59	0.59	0.75	0.76
256	%imb-map	1.6	1.6	4.2	4.2	1.6	1.6	1.00	1.00	1.00	2.7	1.0	2.7	1.0
	%imb-reduce	2.0	2.0	5.3	5.3	3.6	3.6	1.00	1.00	1.00	2.6	1.8	2.6	1.8
	volume	455.9	461.7	2.0	3.1	1.8	5.0	1.01	1.58	2.76	0.004	0.004	0.007	0.011
	time	0.20	0.15	0.13	0.12	0.13	0.12	0.73	0.89	0.89	0.66	0.67	0.81	0.81
512	%imb-map	2.7	2.7	6.0	6.0	1.9	1.9	1.00	1.00	1.00	2.2	0.7	2.2	0.7
	%imb-reduce	3.3	3.3	7.4	7.4	5.1	5.1	1.00	1.00	1.00	2.2	1.5	2.2	1.5
	volume	459.7	462.6	2.9	4.5	2.6	7.1	1.01	1.57	2.71	0.006	0.006	0.010	0.015
	time	0.17	0.13	0.10	0.09	0.10	0.09	0.78	0.86	0.86	0.60	0.60	0.66	0.66
1024	%imb-map	3.9	3.9	7.1	7.1	2.2	2.2	1.00	1.00	1.00	1.8	0.6	1.8	0.6
	%imb-reduce	4.5	4.5	8.8	8.8	6.6	6.6	1.00	1.00	1.00	1.9	1.5	1.9	1.5
	volume	461.7	463.1	4.1	6.4	3.8	10.1	1.00	1.56	2.68	0.009	0.008	0.014	0.022
	time	0.23	0.19	0.09	0.07	0.09	0.07	0.85	0.83	0.82	0.39	0.40	0.38	0.39

All sparse matrix operations (SpMV, SpGEMM) are implemented using the MR-MPI library [14]. The partitions obtained by the graph/hypergraph models are fed to the *aggregate*() and *collate*() as hash functions. Each sparse matrix operation is repeated 10 times and the average is reported in the results in the upcoming sections. Metis [34] is used to partition the graphs and PaToH [32] is used to partition the hypergraphs, both in default settings. The maximum allowed imbalance in processors' loads in both computational phases is set to 10% for each of the two constraints. Recall that this imbalance determines the maximum allowed imbalance in both computational phases.

We evaluate the performance of all schemes for each operation with the matrices given in Table 1, which are from the UFL Sparse Matrix Collection [35]. For each type of operation, we include 10 matrices. The maximum degree values presented in the table are the maximum of maximum number of nonzeros in rows and columns. For SpGEMM, we test the operation $C = AA^T$, which is also listed as one of the key operations and included in the experiments of [33].

6.1. SpMV

The results obtained for the SpMV operation are presented in Table 2. We compare the schemes in terms of four metrics: computational imbalance in map and reduce phases in terms of KV pairs (indicated with imb-map and imb-reduce, respectively), communication volume (volume) and runtime (time). The volume is in terms of megabytes (Mb) and the time is in terms of milliseconds. The table is grouped under three basic column groups. The first column group presents the actual results obtained by the compared schemes. The second column group compares the schemes within themselves, i.e., with and without the optional local reduce. The last column group measures the performance of partitioning models against the baseline random assignment. Each value in the table is the geometric mean of the results obtained for the matrices used for SpMV on a specific number of processors. The last two column groups contain the normalized values in the format of A/B, which means scheme A is normalized with respect to scheme B.

When we compare the schemes that use partitioning models for task assignment (i.e., GR, GRr, HY, HYr) against the ones that do not (i.e., RN, RNr), the benefits of using a model are seen clearly. These models

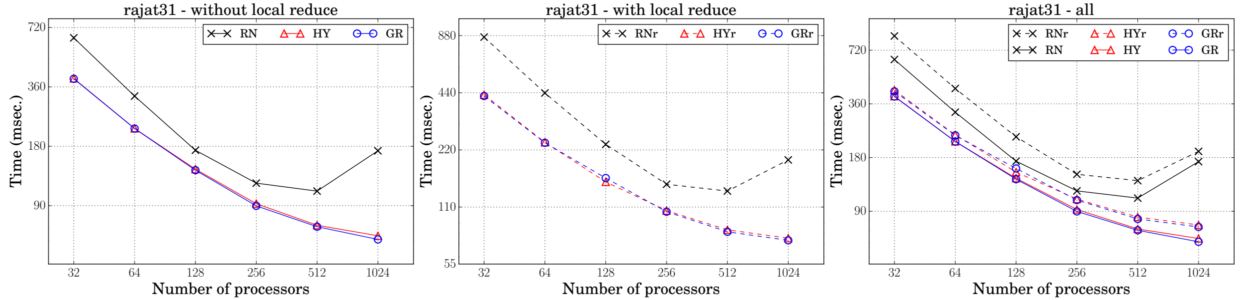

Figure 4: Speedups of compared schemes in parallel SpMV for matrix `rajat31`. Both axes are in logarithmic scale.

decrease the communication volume drastically by obtaining a volume of no more than 7 Mb in any K value, whereas the communication volume of `RN` or `RNr` is around 400 Mb. The reduction in communication volume is reflected as improvement in overall runtime of the SpMV. For example on 128 processors, `RN` obtains an SpMV time of 0.34 milliseconds, while `GR` obtains an SpMV time of 0.18 milliseconds. In terms of imbalance, the schemes that utilize random assignment usually exhibit better performance since the sole purpose of these schemes is maintaining such a balance, while for the schemes that utilize partitioning models balance is a constraint rather than objective.

The execution of the optional local reduce is expected to decrease the communication volume. This is validated from the values in the second column group and the volume row. For example on 128 processors, `RN` incurs 3% more volume than `RNr`, `GR` incurs 59% more volume than `GRr` and `HY` incurs 184% more volume than `HYr`. This difference is less in `RN` and `RNr` since random assignment already necessitates a large amount of communication. The results regarding the optional local reduce indicate that performing local reduce does not pay off as the parallel runtimes obtained by `RN`, `GR` and `HY` are lower than the ones obtained by `RNr`, `GRr` and `HYr`, respectively. However, this may not always be the case, especially when the savings from communication are drastic with the execution of local reduce, which happens not to be the case for SpMV. Note that the imbalances in KV pairs in the first and second phases of computations are the same with or without the local reduce as their counts are independent of it.

Recall that without the local reduce, the graph model correctly encapsulates the total volume during the partitioning process. From the volume results in Table 2, when we compare `GR` and `HY`, it is seen that `GR` obtains lower volume for any K: for example on 512 processors the volume of `GR` is 4.5 Mb while it is 7.1 Mb for `HY`. On the other hand, with the local reduce, the hypergraph model correctly encapsulates the total volume. When we compare `GRr` and `HYr`, it is seen that `HYr` obtains lower volume for any K: for example on 512 processors the volume of `HYr` is 2.6 Mb while it is 2.9 Mb for `GRr`.

Figure 4 presents the parallel SpMV runtimes obtained by the compared schemes for matrix `rajat31`. There are three plots: the one in the left compares the schemes that do not contain local reduce, i.e., `RN`, `GR`, `HY`, the one in the center compares the schemes that contain local reduce, i.e., `RNr`, `GRr`, `HYr`, and the one in the right compares all. We display the plots for a single matrix only as the plots for other matrices exhibit similar behaviors. Both with and without local reduce, the task assignments realized by the partitioning models scale much better. Observe that the schemes without local reduce obtain lower runtimes compared to their counterparts, as also observed in Table 2. Up to 256 processors, all schemes seem to scale, but after that point, the schemes relying on random assignment scale poorly while the schemes relying on partitioning models scale further by being able to decrease the runtime. The reason behind this is the increased importance of communication in overall runtime, which we investigate next.

Figure 5 illustrates the dissection of parallel SpMV times as bar charts for matrix `rajat31` on 64, 256 and 1024 processors. Blue and yellow bars in the figure respectively represent the computation and communication times. When we compare the performance of different schemes (`RNr`, `GRr`, `HYr`) for a specific number of processors with local reduce, it is seen that the computation times are roughly the same, whereas the communication times vary drastically. This is also the case without local reduce. When we compare the communication performance of a scheme for a specific number of processors, it is observed that local

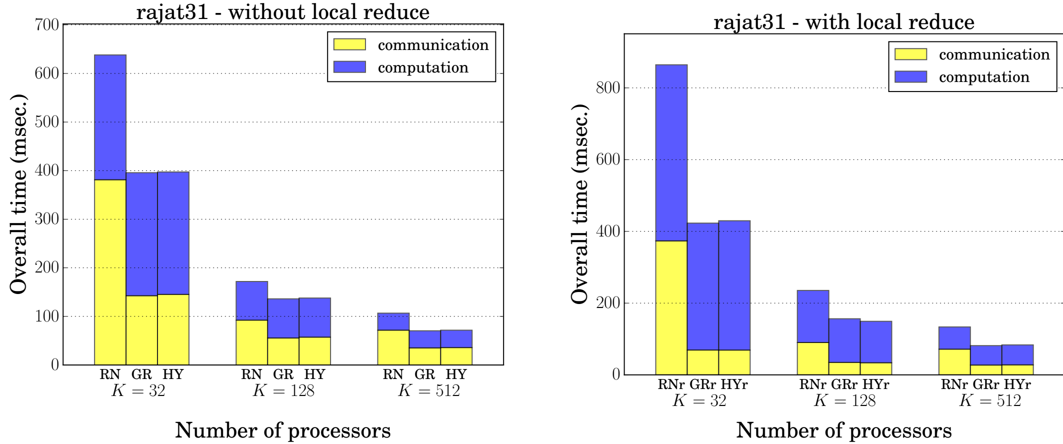

Figure 5: Dissection of computation and communication times in parallel SpMV for matrix `rajat31` on 32, 128 and 512 processors.

reduce decreases the communication time significantly as expected. Although it is expected that the total amount of computation of a scheme for a specific number of processors should stay the same with or without local reduce, this does not seem to be the case due to the overhead of the *convert*() and *reduce*() operations involved in local reduce. As seen from both bar charts, the key to scalability is to address the communication bottlenecks, which is achieved by the partitioning models in a very successful manner.

6.2. SpGEMM

The results obtained for the SpGEMM operation are presented in Table 3. The decoding of the table is the same with the one presented for SpMV (Table 2). We experiment up to 512 processors for this operation.

As seen from Table 3, the schemes that utilize a partitioning model decrease the communication volume drastically in the shuffle phase. For example on 128 processors, `GRr` and `HYr` incur a volume of 5-6 Mb, while `RNr` incurs a volume of 341.7 Mb. Similarly, on the same number of processors, `GR` and `HY` respectively incur a volume of 13.1 and 21.4 Mb, while `RN` incurs a volume of 356 Mb. The benefit of decreasing data transferred is seen as improvement in parallel SpGEMM runtime: the schemes relying on partitioning models obtain more than 2-4x speedup over the ones that do not so for any number of processors. The schemes exhibit close performance in computational balance in the map phase. However, `RN` and `RNr` obtain better balance in the reduce phase.

As also observed in the SpMV operation, performing the optional local reduce leads to reductions in data transfer in the shuffle phase. For random assignment schemes, the optional local reduce does not seem to work as `RNr` obtains higher parallel SpGEMM times than `RN`. This is because there is not much difference in the volumes incurred by these two schemes. On the other hand, for small number of processors, the optional local reduce pays off for the schemes that rely on partitioning models up to 256 processors. Comparing the volumes incurred by the graph and hypergraph models, when there is no local reduce `GR` always obtains lower volume than `HY` for any number of processors. In the existence of local reduce, `HYr` obtains lower volume than `GRr` for 128, 256 and 512 processors, while `GRr` obtains lower volume than `HYr` in 32 and 64 processors. Note that graph partitioners can perform close to hypergraph partitioners if the sparsity pattern of the underlying model accommodates uniformity.

Figure 6 presents the parallel SpGEMM runtimes obtained by the compared schemes for matrix `kim1`. The left plot compares the schemes that do not contain local reduce, i.e., `RN`, `GR`, `HY`, the center plot compares the schemes that contain local reduce, i.e., `RNr`, `GRr`, `HYr`, and the right plot compares all. With or without local reduce, the schemes relying on partitioning models exhibit better scalability. `RN` and `RNr` scale up to 128 processors, while `GR`, `GRr`, `HY` and `HYr` scale all the way up to 512 processors. As also observed in Table 3, `GRr` and `HYr` perform slightly better than `GR` and `HY` on small number of processors, while the opposite situation is observed on 256 and 512 processors.

Table 3: Volume, imbalance and runtime averages for SpGEMM (volume in megabytes and time in milliseconds).

		actual values						normalized within scheme			normalized wrt `RN` and `RNr`			
K	scheme	`RNr`	`RN`	`GRr`	`GR`	`HYr`	`HY`	`RN/RNr`	`GR/GRr`	`HY/HYr`	`GRr/RNr`	`HYr/RNr`	`GR/RN`	`HY/RN`
32	%imb-map	6.5	6.5	7.3	7.3	5.6	5.6	1.00	1.00	1.00	1.1	0.9	1.1	0.9
	%imb-reduce	0.6	0.6	4.9	4.9	2.9	2.9	1.00	1.00	1.00	7.8	4.7	7.8	4.7
	volume	299.7	347.6	2.2	4.4	2.4	10.1	1.16	2.00	4.28	0.007	0.008	0.013	0.029
	time	0.76	0.57	0.30	0.32	0.31	0.36	0.75	1.08	1.17	0.39	0.41	0.56	0.63
64	%imb-map	9.1	9.1	10.4	10.4	9.0	9.0	1.00	1.00	1.00	1.2	1.0	1.2	1.0
	%imb-reduce	0.9	0.9	7.2	7.2	4.5	4.5	1.00	1.00	1.00	7.7	4.8	7.7	4.8
	volume	326.4	353.1	3.6	7.0	3.8	15.2	1.08	1.97	4.00	0.011	0.012	0.020	0.043
	time	0.43	0.33	0.17	0.18	0.18	0.21	0.75	1.04	1.11	0.40	0.42	0.56	0.63
128	%imb-map	17.5	17.5	15.4	15.4	12.9	12.9	1.00	1.00	1.00	0.9	0.7	0.9	0.7
	%imb-reduce	1.3	1.3	10.1	10.1	5.8	5.8	1.00	1.00	1.00	8.0	4.6	8.0	4.6
	volume	341.7	356.0	6.1	13.1	5.7	21.4	1.04	2.15	3.73	0.018	0.017	0.037	0.060
	time	0.25	0.19	0.12	0.11	0.12	0.12	0.73	0.99	1.02	0.45	0.48	0.61	0.66
256	%imb-map	20.0	20.0	23.4	23.4	18.0	18.0	1.00	1.00	1.00	1.2	0.9	1.2	0.9
	%imb-reduce	2.0	2.0	15.9	15.9	8.2	8.2	1.00	1.00	1.00	8.0	4.2	8.0	4.2
	volume	350.1	357.4	9.6	21.7	8.8	32.2	1.02	2.26	3.65	0.027	0.025	0.061	0.090
	time	0.18	0.14	0.09	0.08	0.09	0.09	0.77	0.92	0.97	0.49	0.50	0.58	0.63
512	%imb-map	30.1	30.1	29.3	29.3	24.7	24.7	1.00	1.00	1.00	1.0	0.8	1.0	0.8
	%imb-reduce	3.9	3.9	18.3	18.3	12.9	12.9	1.00	1.00	1.00	4.7	3.3	4.7	3.3
	volume	354.4	357.8	14.2	32.4	13.4	46.8	1.01	2.28	3.49	0.040	0.038	0.091	0.131
	time	0.30	0.27	0.07	0.06	0.08	0.07	0.90	0.87	0.89	0.24	0.25	0.24	0.25

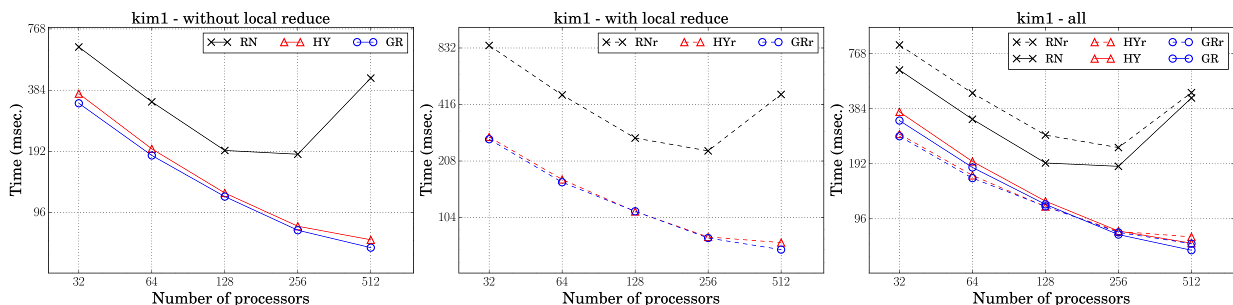

Figure 6: Speedups of compared schemes in parallel SpGEMM for matrix `kim1`. Both axes are in logarithmic scale.

Figure 7 illustrates the dissection of parallel SpGEMM times as bar charts for matrix `kim1` on 32, 128 and 512 processors. Observe that, as also was the case for SpMV, the schemes with local reduce have less communication overhead compared to the schemes without local reduce. The arguments made for SpMV are also valid for SpGEMM. Compared to SpMV, the improvements in the communication performance are more pronounced with the execution of the local reduce. This is due to the higher number of intermediate KV pairs produced in SpGEMM. Since all schemes achieve a good computational balance, the key to better parallel performance and scalability lies in the reduction of communication overheads.

7. Conclusions

Using MapReduce, we focused on efficient parallelization of two key operations, sparse matrix-vector multiplication and sparse matrix-sparse matrix multiplication- that are very common in scientific computations and graph algorithms. We fully exploited domain-specific knowledge with successful graph and hypergraph models by balancing processors' loads in map and reduce phases and decreasing volume in the shuffle phase. In order to utilize these models for the operations realized with MapReduce, the partitions produced by the models are used as hash functions for scheduling tasks in map and reduce phases. Utilization of these models lead to improvements in parallel runtime for both operations and improved their scalability.

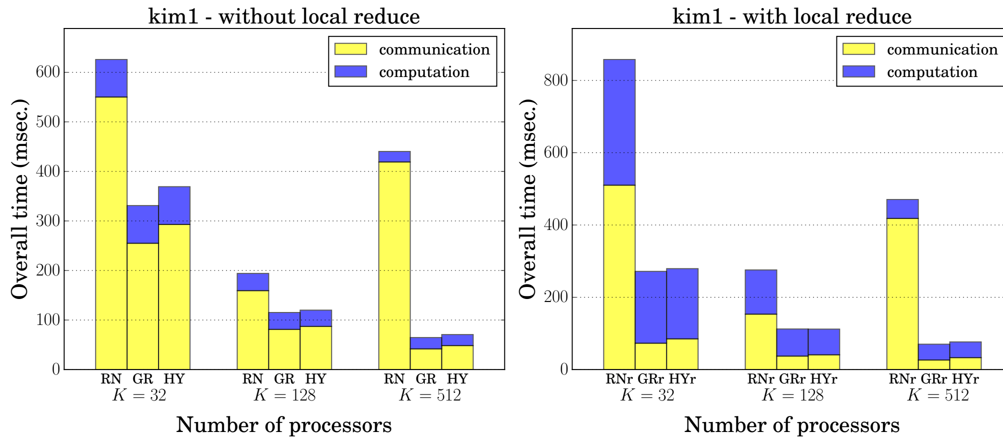

Figure 7: Dissection of computation and communication times in parallel SpGEMM for matrix `kim1` on 32, 128 and 512 processors.

Acknowledgements

450 We acknowledge PRACE (Partnership for Advanced Computing In Europe) for awarding us access to SuperMUC based in Germany at Leibniz Supercomputing Centre.

References

[1] J. Dean, S. Ghemawat, Mapreduce: Simplified data processing on large clusters, Commun. ACM 51 (1) (2008) 107–113. doi:10.1145/1327452.1327492.
455 URL `http://doi.acm.org/10.1145/1327452.1327492`

[2] Apache hadoop, `http://hadoop.apache.org/`, accessed: 2017-01-3.

[3] M. Hammoud, M. S. Rehman, M. F. Sakr, Center-of-gravity reduce task scheduling to lower mapreduce network traffic, in: 2012 IEEE Fifth International Conference on Cloud Computing, 2012, pp. 49–58. doi:10.1109/CLOUD.2012.92.

[4] B. Palanisamy, A. Singh, L. Liu, B. Jain, Purlieus: Locality-aware resource allocation for mapreduce in a cloud, in:
460 Proceedings of 2011 International Conference for High Performance Computing, Networking, Storage and Analysis, SC '11, ACM, New York, NY, USA, 2011, pp. 58:1–58:11. doi:10.1145/2063384.2063462.
URL `http://doi.acm.org/10.1145/2063384.2063462`

[5] S. Ibrahim, H. Jin, L. Lu, S. Wu, B. He, L. Qi, Leen: Locality/fairness-aware key partitioning for mapreduce in the cloud, in: 2010 IEEE Second International Conference on Cloud Computing Technology and Science, 2010, pp. 17–24.
465 doi:10.1109/CloudCom.2010.25.

[6] M. Hammoud, M. F. Sakr, Locality-aware reduce task scheduling for mapreduce, in: Proceedings of the 2011 IEEE Third International Conference on Cloud Computing Technology and Science, CLOUDCOM '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 570–576. doi:10.1109/CloudCom.2011.87.
URL `https://doi.org/10.1109/CloudCom.2011.87`

470 [7] M. Liroz-Gistau, R. Akbarinia, D. Agrawal, E. Pacitti, P. Valduriez, Data Partitioning for Minimizing Transferred Data in MapReduce, Springer Berlin Heidelberg, Berlin, Heidelberg, 2013, pp. 1–12.

[8] J. Li, J. Wu, X. Yang, S. Zhong, Optimizing mapreduce based on locality of k-v pairs and overlap between shuffle and local reduce, in: 2015 44th International Conference on Parallel Processing, 2015, pp. 939–948. doi:10.1109/ICPP.2015.103.

[9] L. Fan, B. Gao, X. Sun, F. Zhang, Z. Liu, Improving the load balance of mapreduce operations based on the key distribution
475 of pairs, CoRR abs/1401.0355.
URL `http://arxiv.org/abs/1401.0355`

[10] P. J. Braam, R. Zahir, Lustre: A scalable, high performance file system, Cluster File Systems, Inc.

[11] Amazon S3, Simple Storage Service, howpublished = `https://aws.amazon.com/s3/`, note = Accessed: 2017-12-21.

[12] S. Seo, E. J. Yoon, J. Kim, S. Jin, J.-S. Kim, S. Maeng, Hama: An efficient matrix computation with the mapreduce frame-
480 work, in: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 721–726. doi:10.1109/CloudCom.2010.17.
URL `http://dx.doi.org/10.1109/CloudCom.2010.17`

[13] Apache Mahout, howpublished = `http://mahout.apache.org/`, note = Accessed: 2017-12-21.

[14] S. J. Plimpton, K. D. Devine, Mapreduce in mpi for large-scale graph algorithms, Parallel Comput. 37 (9) (2011) 610–632.
485 doi:10.1016/j.parco.2011.02.004.
URL `http://dx.doi.org/10.1016/j.parco.2011.02.004`

[15] J. Ekanayake, G. Fox, High Performance Parallel Computing with Clouds and Cloud Technologies, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 20–38.

[16] Y. Saad, Iterative Methods for Sparse Linear Systems, 2nd Edition, Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 2003.

[17] L. Page, S. Brin, R. Motwani, T. Winograd, The pagerank citation ranking: Bringing order to the web., Technical Report 1999-66, Stanford InfoLab, previous number = SIDL-WP-1999-0120 (November 1999).
URL `http://ilpubs.stanford.edu:8090/422/`

[18] W. Briggs, V. Henson, S. McCormick, A Multigrid Tutorial, Second Edition, 2nd Edition, Society for Industrial and Applied Mathematics, 2000. arXiv:http://epubs.siam.org/doi/pdf/10.1137/1.9780898719505, doi:10.1137/1.9780898719505.
URL `http://epubs.siam.org/doi/abs/10.1137/1.9780898719505`

[19] R. H. Bisseling, T. M. Doup, L. D. J. C. Loyens, A parallel interior point algorithm for linear programming on a network of transputers, Annals of Operations Research 43 (2) (1993) 49–86. doi:10.1007/BF02024486.
URL `http://dx.doi.org/10.1007/BF02024486`

[20] A. Buluç, J. R. Gilbert, The Combinatorial BLAS: Design, implementation, and applications, Int. J. High Perform. Comput. Appl. 25 (4) (2011) 496–509. doi:10.1177/1094342011403516.
URL `http://dx.doi.org/10.1177/1094342011403516`

[21] C. Ordonez, Optimization of linear recursive queries in sql, IEEE Transactions on Knowledge and Data Engineering 22 (2) (2010) 264–277. doi:10.1109/TKDE.2009.83.

[22] G. Linden, B. Smith, J. York, Amazon.com recommendations: item-to-item collaborative filtering, IEEE Internet Computing 7 (1) (2003) 76–80. doi:10.1109/MIC.2003.1167344.

[23] K. Kc, K. Anyanwu, Scheduling hadoop jobs to meet deadlines, in: Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science, CLOUDCOM '10, IEEE Computer Society, Washington, DC, USA, 2010, pp. 388–392. doi:10.1109/CloudCom.2010.97.
URL `http://dx.doi.org/10.1109/CloudCom.2010.97`

[24] T. Sandholm, K. Lai, Dynamic proportional share scheduling in hadoop, in: Proceedings of the 15th International Conference on Job Scheduling Strategies for Parallel Processing, JSSPP'10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 110–131.
URL `http://dl.acm.org/citation.cfm?id=1927648.1927655`

[25] J. Polo, D. Carrera, Y. Becerra, M. Steinder, I. Whalley, Performance-driven task co-scheduling for mapreduce environments, in: 2010 IEEE Network Operations and Management Symposium - NOMS 2010, 2010, pp. 373–380. doi:10.1109/NOMS.2010.5488494.

[26] J. Wolf, D. Rajan, K. Hildrum, R. Khandekar, V. Kumar, S. Parekh, K.-L. Wu, A. balmin, Flex: A slot allocation scheduling optimizer for mapreduce workloads, in: Proceedings of the ACM/IFIP/USENIX 11th International Conference on Middleware, Middleware '10, Springer-Verlag, Berlin, Heidelberg, 2010, pp. 1–20.
URL `http://dl.acm.org/citation.cfm?id=2023718.2023720`

[27] C. Tian, H. Zhou, Y. He, L. Zha, A dynamic mapreduce scheduler for heterogeneous workloads, in: 2009 Eighth International Conference on Grid and Cooperative Computing, 2009, pp. 218–224. doi:10.1109/GCC.2009.19.

[28] Z. Tang, L. Jiang, J. Zhou, K. Li, K. Li, A self-adaptive scheduling algorithm for reduce start time, Future Generation Computer Systems 4344 (2015) 51 – 60. doi:http://dx.doi.org/10.1016/j.future.2014.08.011.
URL `http://www.sciencedirect.com/science/article/pii/S0167739X14001599`

[29] M. Lin, L. Zhang, A. Wierman, J. Tan, Joint optimization of overlapping phases in mapreduce, Performance Evaluation 70 (10) (2013) 720 – 735, proceedings of {IFIP} Performance 2013 Conference. doi:http://dx.doi.org/10.1016/j.peva.2013.08.013.
URL `http://www.sciencedirect.com/science/article/pii/S0166531613000916`

[30] F. Ahmad, S. Lee, M. Thottethodi, T. N. Vijaykumar, Mapreduce with communication overlap (marco), J. Parallel Distrib. Comput. 73 (5) (2013) 608–620. doi:10.1016/j.jpdc.2012.12.012.
URL `http://dx.doi.org/10.1016/j.jpdc.2012.12.012`

[31] A. Verma, N. Zea, B. Cho, I. Gupta, R. H. Campbell, Breaking the mapreduce stage barrier, in: 2010 IEEE International Conference on Cluster Computing, 2010, pp. 235–244. doi:10.1109/CLUSTER.2010.29.

[32] U. V. Çatalyürek, C. Aykanat, Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication, IEEE Trans. Parallel Distrib. Syst. 10 (1999) 673–693. doi:10.1109/71.780863.
URL `http://portal.acm.org/citation.cfm?id=311796.311798`

[33] K. Akbudak, C. Aykanat, Simultaneous input and output matrix partitioning for outer-product–parallel sparse matrix-matrix multiplication, SIAM Journal on Scientific Computing 36 (5) (2014) C568–C590. arXiv:http://dx.doi.org/10.1137/13092589X, doi:10.1137/13092589X.
URL `http://dx.doi.org/10.1137/13092589X`

[34] G. Karypis, V. Kumar, A fast and high quality multilevel scheme for partitioning irregular graphs, SIAM J. Sci. Comput. 20 (1) (1998) 359–392. doi:10.1137/S1064827595287997.
URL `http://dx.doi.org/10.1137/S1064827595287997`

[35] T. A. Davis, Y. Hu, The University of Florida Sparse Matrix Collection, ACM Trans. Math. Softw. 38 (1) (2011) 1:1–1:25. doi:10.1145/2049662.2049663.
URL `http://doi.acm.org/10.1145/2049662.2049663`