Algorithmic Aspects of Parallel Query Processing

Paris Koutris
University of Wisconsin-Madison
paris@cs.wisc.edu

Semih Salihoglu
University of Waterloo
semih.salihoglu@uwaterloo.ca

Dan Suciu
University of Washington
suciu@cs.washington.edu

ABSTRACT

In the last decade we have witnessed a growing interest in processing large data sets on large-scale distributed clusters. A big part of the complex data analysis pipelines performed by these systems consists of a sequence of relatively simple query operations, such as joining two or more tables, or sorting. This tutorial discusses several recent algorithmic developments for data processing in such large distributed clusters. It uses as a model of computation the Massively Parallel Computation (MPC) model, a simplification of the BSP model, where the only cost is given by the amount of communication and the number of communication rounds. Based on the MPC model, we study and analyze several algorithms for three core data processing tasks: multiway join queries, sorting and matrix multiplication. We discuss the common algorithmic techniques across all tasks, relate the algorithms to what is used in practical systems, and finally present open problems for future research.

KEYWORDS

Distributed Query Evaluation, Bulk Synchronous Parallel Model

ACM Reference Format:
Paris Koutris, Semih Salihoglu, and Dan Suciu. 2018. Algorithmic Aspects of Parallel Query Processing. In *SIGMOD'18: 2018 International Conference on Management of Data, June 10–15, 2018, Houston, TX, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3183713.3197388

1 INTRODUCTION

In the last decade we have witnessed a huge and growing interest in processing large data sets on large distributed clusters. This trend began with the MapReduce framework [12], and has been widely adopted by several other systems, including PigLatin [29], Hive [33], Scope [10], Dremmel [28], Spark [38], and Myria [36] to name a few. While the applications of such systems are diverse (e.g., machine learning and data analytics), most involve relatively standard data processing tasks, such as identifying relevant data, cleaning, filtering, sorting, joining, grouping, and extracting features [11, 15]. This has generated great interest in the study of algorithms for such data processing tasks on large distributed clusters.

This tutorial is based on an upcoming survey paper we wrote for the Foundations and Trends in Databases journal [25] and reviews recent results on efficient data processing on large distributed architectures, primarily focusing on join processing and the relevant classical results on parallel sorting and matrix multiplication. We highlight the common algorithmic techniques that are used to perform these computations, as well as techniques to provide lower bounds on the communication cost of algorithms. We identify the existing or potential applications of these algorithmic techniques by reviewing the actual algorithms used by systems in practice. For each data processing task we review, we also discuss the important open problems for future research.

Here is the outline of this tutorial proposal. Sections 2-5 give an overview of the topics that we will cover, each corresponding to one part of our tutorial. Each of these sections, except Section 2, finishes with a discussion of important open problems. We conclude in Section 6 with brief biographies of the authors.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
SIGMOD'18, June 10–15, 2018, Houston, TX, USA
© 2018 Copyright held by the owner/author(s). Publication rights licensed to the Association for Computing Machinery.
ACM ISBN 978-1-4503-4703-7/18/06…$15.00
https://doi.org/10.1145/3183713.3197388

2 MODELS OF PARALLEL COMPUTATION

The tutorial uses as model of computation the *Massively Parallel Computation Model* [6], Fig. 1, which is a simplified variant of Valiant's Bulk Synchronous Parallel (BSP) model [34]. In this model the data is distributed on several processors (servers), and computation proceeds in supersteps, where, in each superstep, the nodes perform local computation followed by a global communication (data reshuffle). We introduce MPC and discuss its parameters: the number of processors p, the load per processor L, the number of rounds r. We explain how the model captures speedup and scaleup, which are widely used metrics in parallel databases [14]. Next, we will discuss several classical models of parallel computation and their relationship to MPC.

2.1 Circuits

One of the earliest models of parallel computation represents an algorithm as a circuit. For any input of size N the computation is given by a DAG, where each leaf represents one of the N inputs, and each internal node represents some simple operation. The key parameters of a circuit are the total number of nodes called *work* W, the depth D, and the fan-in s of each node. We will briefly describe the complexity classes $\mathtt{NC}^k, \mathtt{AC}^k$, defined by family of Boolean circuits with restrictions on W, D, and s. We then discuss the connection between circuits and the MPC model. Circuits correspond naturally to *oblivious* MPC algorithms, however non-oblivious MPC algorithms are strictly more powerful than the corresponding circuit; this is due to an observation made in [31].

2.2 The PRAM Model

While the circuit model is adequate for defining complexity classes, and for proving lower bounds, the Parallel Random Access Machine (PRAM) model is intended primarily for algorithm design. We will introduce the PRAM model (a formal model corresponding to a

shared-memory architecture), and discuss Brent's theorem [8]. As before, we explain the gap between the MPC model and the PRAM model.

2.3 The BSP Model

To model shared-nothing architectures, Valiant introduced the *Bulk Synchronous Parallel (BSP)* model [34]. A BSP algorithm runs in *supersteps*, where each superstep consists of local computation and communication, followed by a synchronization barrier. The BSP model abstracts the communication in every superstep by introducing the notion of the *h-relation*, which is defined as a communication pattern where the maximum number of incoming or outgoing messages per machine is h. Several refinements of BSP were later considered in the literature, such as the EREW model, the CREW model, and the Weak-CREW model. The MPC model is a simplification, in that it ignores the internal computation time and the network parameters.

2.4 MapReduce-Based Models

Following the introduction of MapReduce [12], several theoretical models were introduced in the literature to capture computations in this setting. We will briefly describe the following: the MapReduce class ($\mathcal{MRC}$) of algorithms by Karloff et al. [22], a variant of the BSP model by Goodrich et al. [17], and the model developed by Afrati et al. [1].

2.5 Connection to the External Memory Model

We conclude this part of the tutorial by discussing an interesting connection between the MPC model and the External Memory Model [35], by presenting the result in [24]: any MPC algorithm can be converted naturally to an external memory algorithm.

3 CONJUNCTIVE QUERIES

In this part of the tutorial, we discuss algorithms for computing multiway joins in a distributed cluster. Join processing is one of the most critical and well-studied tasks in a data processing pipeline. We start by presenting algorithms for the simplest case of a two-way join, which serves as a warm-up scenario to introduce the main techniques. Next, we focus on algorithms for multiway joins that are restricted to a single communication round. Finally, we show how the algorithms can be extended to multiple rounds.

3.1 Two-way Joins

In the first part, we illustrate how to compute the natural join of two large relations,

$$R(A,B) \bowtie S(B,C)$$

There exist two main algorithmic techniques for join processing: hash-based and sort-based algorithms. We present the hash-based algorithm here, and defer the sort-based algorithm for Section 4.

The *parallel hash-join* is the most common parallel join algorithm in several parallel systems, and operates by distributing each tuple from R, S by hashing on the value of the join attribute. An analysis of the parallel hash-join tells us that, for input of size IN and p processors, there is a particular threshold for data skew (defined as the maximum frequency of a value in a relation), below which the load is $O(\mathsf{IN}/p)$, but above which there is no optimal guarantee. To deal with such skewed instances, we introduce an algorithm that achieves the optimal load in a single round, by treating each one of the skewed values separately.

3.2 Single-round Algorithms

We initiate the discussion of multiway join algorithms by considering algorithms that operate using a single communication round. The core algorithmic tool in this case is the Hypercube (or Shares) algorithm [2], which is a generalization of the parallel hash-join algorithm. We explain the Hypercube algorithm through the example of the *triangle query*, which will be used extensively throughout this tutorial. The Hypercube algorithm is optimal amongst single-round algorithms on skew-free databases. We examine the speedup of the Hypercube algorithm, showing how it generally differs from an ideal linear speedup: in particular, the speedup will be of the form $\sim p^{1/\tau^*}$, where τ^* is a parameter dependent on the join query called the *fractional edge packing number*.

When the data is skewed, the Hypercube algorithm does not perform optimally anymore. We show how we can extend the Hypercube algorithm from skew-free data to arbitrary data, and describe a worst-case optimal algorithm called SkewHC [7]. This algorithm is obtained by running different instances of the Hypercube algorithm on various skewed subsets of the input data. Skewed data causes the speedup of the algorithm to worsen compared to the speedup on skew-free data, and this can be proven to be unavoidable, if we want to compute the query in one round.

We end the presentation of single-round algorithms with a brief discussion of output-sensitive algorithms, which are algorithms designed to be sensitive to the size of the output data (and not only the input data).

3.3 Multi-round Algorithms

For a parallel algorithm in the MPC model, each additional round incurs a significant cost. Nevertheless, there are cases where running a query in multiple rounds can significantly reduce the communication cost per round, and in many cases it makes sense to trade off rounds for communication per round. We describe here several results known for query processing in multiple rounds.

We start with a lower bound [24] for any algorithm that computes the query in a constant number of rounds, which tells us that no multiway join query can be computed with load better than $\Omega(\mathsf{IN}/p^{1/\rho^*})$. Here, ρ^* is another query-dependent parameter called the *fractional edge covering number*.

The above lower bound can be matched for several classes of queries by using multiple rounds to remove the penalty incurred by skewed values [23]. The additional rounds are used to process the skewed values, and the number of rounds depends only on the structure of the query.

Next, we discuss a more traditional usage of multiple rounds. This technique computes simpler operators one at a time in order to reduce the load, or the total communication cost. For example, we may compute a query in a traditional way, one join at a time, using a number of rounds equal to the depth of the query plan. In the absence of skew each join has linear speedup, which makes this approach attractive in practice. The challenge here is that the size

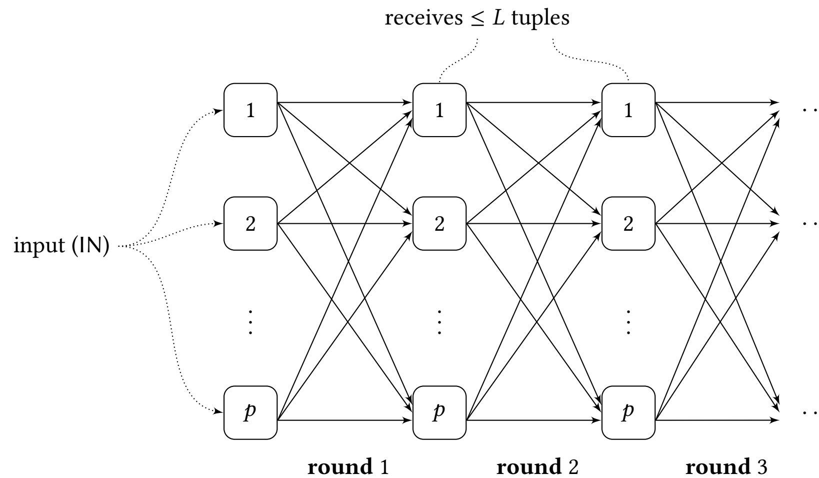

Figure 1: The MPC model of parallel computation.

of the intermediate results may increase significantly compared to the input size IN. We describe techniques based on Yannakakis' algorithm [37] and on tree decompositions, which lead to an output-sensitive algorithm.

3.4 Existing Systems and Open Problems

We conclude this part by discussing how multiway joins are computed in existing parallel systems, such as Hive, SparkSQL, Myria, and some graph processing systems that evaluate subgraph queries. We also present some of the open problems. One important problem we will emphasize is designing a constant-round distributed multiway join algorithm that is worst-case optimal for any query.

4 SORTING

We next present an overview of parallel sorting for shared-nothing architectures. Sorting is a fundamental operation in data processing, both as a stand-alone task and as a subroutine in other queries, such as joining two tables in a distributed cluster [18] or computing parallel similarity joins of tables [18]. We present lower bounds on the communication and round costs of sorting and algorithms from existing systems and literature. We finish by showing an application of sorting in a distributed sort-merge binary join algorithm that achieves stronger guarantees than the Hypercube algorithm.

4.1 Lower Bounds

We first review two simple lower bounds for parallel sorting in the MPC model, under the assumption that the load per round per processor is bounded by L. The first bound shows that the total amount of communication is $\Omega(N \log_L N)$ and is adapted from Aggarwal et. al. [5]'s I/O complexity of sorting in the external memory model. This bound assumes a comparison-based model. The second bound shows that the number of rounds is $\Omega(\log_L N)$ and derives from a fundamental result on circuits. It therefore applies to any computational model, and, thus, is stronger. Figure 2 summarizes these results. These lower bounds hold regardless of p. We must have $p \geq N/L$ in order for the processors to be able to store the

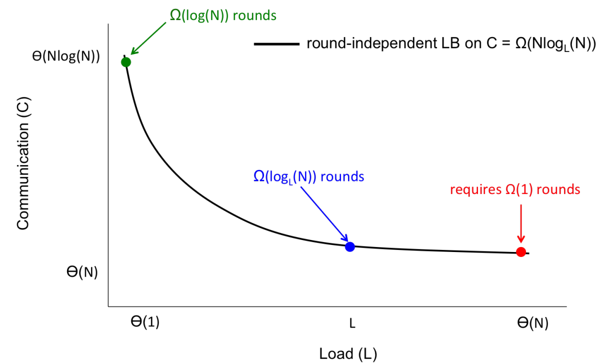

Figure 2: Lower Bounds on Sorting.

entire input data but, beyond that, p is not a relevant factor in determining the distributed complexity of sorting. This contrasts with both joins and matrix multiplication, where one can sometimes affect communication/load or rounds by increasing p.

4.2 Parallel Sorting Algorithms

We start by reviewing existing sorting algorithms used in practice. Specifically, we discuss the winners of the GraySort benchmark of the popular Sort Benchmark [32]. These algorithms use relatively small number of processors with large memory and therefore have coarse-grained parallelism. On a cluster with p processors, all of these algorithms use the same main idea of finding $p-1$ splitters, and partitioning the elements to be sorted in one round, then sorting locally. We discuss different ways to pick the splitters and demonstrate that these algorithms are essentially optimal for the load they incur: they asymptotically meet the communication lower bound curve and corresponding round lower bound on the right-end of Figure 2.

We next review algorithms from literature that handle the harder case of efficiently sorting under fine-grained parallelism. Our

primary goal here is to present the audience the ideas behind Goodrich's algorithm [16]. This algorithm has great theoretical value because it is the optimal deterministic distributed algorithm, i.e., matches the sorting lower bounds from Figure 2 asymptotically for every L. However the algorithm is very technical. Instead we briefly give overviews of two algorithms from theoretical literature and discuss how Goodrich's algorithm uses the ideas in these algorithms: (1) *Odd-Even Merge-Sort*, which is a $O(\log^2(n))$-step PRAM algorithm; and (2) *Cole's Algorithm*, which is a $O(\log(n))$-step PRAM algorithm. These algorithms develop the idea of distributed *ranked lists*, which is a core idea used by Goodrich's algorithm.

4.3 Sort-based Join

We next review a recent binary join algorithm from reference [18], which uses Goodrich's algorithm as a subroutine to achieve the stronger guarantees of the skew-resilient SkewHC algorithm we discussed in Section 3. Briefly to join two tables $R(A,B) \bowtie S(B,C)$, this algorithm first concatenates R,S into a single table (containing IN elements) and sorts these elements on B. For any given value $B = b$, there are two cases: (1) If all tuples with value $B = b$ are in the same machine, then the processor u holding these tuples can compute the join of these tuples locally. (2) If the tuples with $B = b$ span two or more processors, then the algorithm allocates a fraction of the p processors, in an extra round of computation, to compute their Cartesian product using the algorithm we discussed in Section 3. There can be at most $p - 1$ crossing values b, and by picking an appropriate fraction of processors for each Cartesian product, we show that the algorithm achieves a load L matching the load of SkewHC, implying that we can alternatively handle skew by sorting.

4.4 Open Problems

The complexity of sorting in the distributed setting is well understood and Goodrich's algorithm matches the communication and round lower bounds asymptotically. In contrast, research in how to use sorting to design efficient join algorithm has recently started and is less understood. One important limitation of the sort-merge algorithm from Section 4.3 is that it can process joins over two tables. We will discuss this limitation and the open problem of using sorting to join more than two tables.

5 DENSE MATRIX MULTIPLICATION

In the final part of the tutorial, we will give an overview of the results on multiplying two $n \times n$ dense matrices by *conventional algorithms*, i.e. those that perform all n^3 elementary multiplications. If the matrices A and B are stored in tables $A(i,k,v)$ and $B(k,j,v)$, where (i,k) and (k,j) represent the index of a non-zero entry, and v is the value of that entry, then matrix multiplication is equivalent to the following relational query that consists of a join and a group-by-aggregate operations:

```
select A.i, B.j, sum(A.v * B.v)
from A, B
where A.k = B.k
group by A.i, B.j
```

We refer to this query as the *matrix multiplication query*.

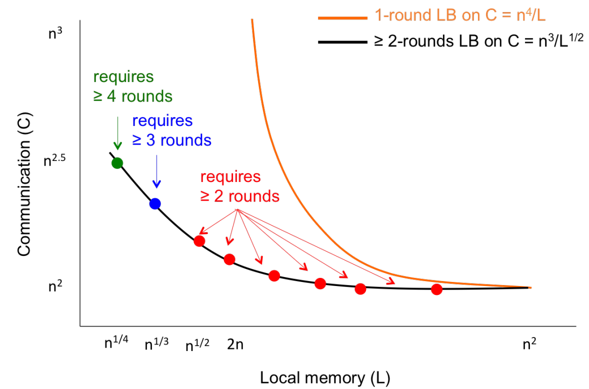

Figure 3: Lower Bounds on Dense Matrix Multiplication.

5.1 Lower Bounds

Similar to sorting, we start by stating three simple lower bounds, shown in Figure 3, under the assumption that the load per round per processor is bounded by L. First is the lower-bound on the communication cost of algorithm that run for a single round (orange curve in Figure 3) [1]. The second is the lower bound on the communication of algorithms that can run any number of rounds (black curve in Figure 3) [19, 20]. The third is the lower bound on the number of rounds needed by algorithms (shown by dots in Figure 3) [30]. Similar to sorting, the latter two lower bounds are adapted from results from I/O complexity of matrix multiplication in the external memory model. Interestingly, we will show that these lower bounds correspond separately to the communication and round lower bounds for the join and group-by-and-aggregate operations of the matrix multiplication query.

5.2 Single-round Algorithm

We next review the optimal one-round algorithm that has its roots in reference [27] and has been adapted to a model of MapReduce in reference [1]. The algorithm works values of L between $2n$ and $2n^2$, since if $L \geq 2n^2$, one can perform the entire multiplication in a single machine in one round, and if $L < 2n$, no one-round algorithm can get enough input elements to produce any of the outputs. We let $t = L/2n$. The algorithm partitions the $n \times n$ matrices A and B into n/t matrices of sizes $t \times n$. In other words, A is a block matrix that is horizontally partitioned into matrices $A_1, \ldots, A_{n/t}$, while B is vertically partitioned into matrices $B_1, \ldots, B_{n/t}$. Then the product C is a block matrix whose blocks are the products $A_u B_v$, for all $u, v = 1, n/t$. The algorithm uses $p = n^2/t^2 = 4n^4/L^2$ processors, identified by pairs (u,v) with $u,v = 1, n/t$. Thus, processor (u,v) receives the matrices A_u, B_v: it multiplies them locally, then outputs the result $A_u B_v$. We show that this single rectangular partitioning technique matches the communication lower bound for one-round algorithms exactly.

5.3 Multi-round Algorithms

The multi-round matrix multiplication algorithm allows a wider range of the load L, even below $2n$, and also a wider range of processors p, to capture the entire range of the curves in Figure 3.

The multi-round algorithm we describe is from reference [26]. Let $L^* \stackrel{\text{def}}{=} L/3$ instead of L and denote $H \stackrel{\text{def}}{=} \frac{n}{\sqrt{L^*}}$. The algorithm block partitions A and B into $H \times H$ square matrices A_{uh} and B_{hv} of size $\sqrt{L^*} \times \sqrt{L^*}$, where $u, h = 1, H$. The product $C = AB$ is a block matrix, where each block is $C_{uv} = \sum_{h=1,H} A_{uh}B_{hv}$, for $u, v = 1, H$. The algorithm computes the H^3 matrix products $A_{uh}B_{hv}$ and then aggregates these products to compute C_{uv}. Since the memory of each of the p processors is $L = 3L^*$, during each round a processor can compute only a single product $A_{uh}B_{hv}$. Depending on the number of processors available, the algorithm performs some fraction of these H^3 products. Once all of the products are performed, the algorithm uses an aggregation tree method to aggregate the products. We will show that this simple square partitioning technique asymptotically matches the communication and round lower bounds for multi-round algorithms.

We will also show that the well-studied *two-dimensional* algorithms [3, 9, 13] correspond to the special case of this multi-round algorithm when the number of processors p is minimum, i.e., enough to hold the input and output matrices. In contrast the *three-dimensional* algorithms [3, 4, 13, 21] correspond to the special case when there are enough processors to perform all of the H^3 partial products in a single round.

5.4 Open Problems

We will conclude by discussing some interesting open problems related to matrix multiplication. An important question we will discuss is the limitations for non conventional algorithms, such as Strassen-like algorithms. We will also discuss if such algorithm can find applications in join processing. Since Strassen-like algorithms use in an essential way subtraction, this would require a novel representation of database relations, where tuples may have either positive or negative multiplicities.

6 AUTHOR INFORMATION

Paris Koutris is an Assistant Professor in the Department of Computer Sciences at the University of Wisconsin-Madison. His work is at the intersection of theory and practice of data management, and spans various areas: data processing for massively parallel systems, data pricing, and managing data under uncertainty. He received the 2016 SIGMOD Jim Gray Doctoral Dissertation Award for his PhD thesis at the University of Washington.

Semih Salihoglu is an Assistant Professor at University of Waterloo's School of Computer Science. He works on large-scale and modern data processing systems particularly focusing on systems that manage and process graphs. He received his PhD from Stanford University in 2015.

Dan Suciu is a Professor in Computer Science at the University of Washington. He works in probabilistic databases, data pricing, parallel data processing, data security, and optimal query processing algorithms. He is a co-author of two books Data on the Web: from Relations to Semistructured Data and XML, 1999, and Probabilistic Databases, 2011. He is a Fellow of the ACM, received the ACM PODS Alberto Mendelzon Test of Time Award in 2010 and in 2012, the 10 Year Most Influential Paper Award in ICDE 2013, the VLDB Ten Year Best Paper Award in 2014, and is a recipient of the NSF Career Award and of an Alfred P. Sloan Fellowship.

REFERENCES

[1] F. N. Afrati, A. D. Sarma, S. Salihoglu, and J. D. Ullman. 2013. Upper and Lower Bounds on the Cost of a Map-Reduce Computation. *PVLDB* 6, 4 (2013).

[2] Foto N. Afrati and Jeffrey D. Ullman. 2011. Optimizing Multiway Joins in a Map-Reduce Environment. *TKDE* 23, 9 (2011).

[3] R. C. Agarwal, S. M. Balle, F. G. Gustavson, M. Joshi, and P. Palkar. 1995. A Three-dimensional Approach to Parallel Matrix Multiplication. *IBM Journal of Research and Development* 39, 5 (1995).

[4] Alok Aggarwal, Ashok K. Chandra, and Marc Snir. 1990. Communication Complexity of PRAMs. *Theoretical Computer Science* 71, 1 (1990).

[5] Alok Aggarwal and S. Vitter, Jeffrey. 1988. The Input/Output Complexity of Sorting and Related Problems. *CACM* 31, 9 (1988).

[6] Paul Beame, Paraschos Koutris, and Dan Suciu. 2013. Communication Steps for Parallel Query Processing. In *PODS*.

[7] Paul Beame, Paraschos Koutris, and Dan Suciu. 2014. Skew in Parallel Query Processing. In *PODS*.

[8] Guy E. Blelloch and Bruce M. Maggs. 2010. Parallel Algorithms. In *Algorithms and Theory of Computation Handbook*. Chapman & Hall/CRC, Chapter 25.

[9] Lynn Elliot Cannon. 1969. *A Cellular Computer to Implement the Kalman Filter Algorithm*. Ph.D. Dissertation. Montana State University, Bozeman, MT, USA.

[10] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. SCOPE: Easy and efficient parallel processing of massive data sets. *PVLDB* 1, 2 (2008).

[11] Surajit Chaudhuri. 2012. What Next?: A Half-dozen Data Management Research Goals for Big Data and the Cloud. In *PODS*.

[12] Jeffrey Dean and Sanjay Ghemawat. 2004. MapReduce: Simplified Data Processing on Large Clusters. In *OSDI*.

[13] Eliezer Dekel, David Nassimi, and Sartaj Sahni. 1984. Parallel Matrix and Graph Algorithms. *SIAM J. Comput.* 16, 3 (1984).

[14] David J. DeWitt and Jim Gray. 1992. Parallel Database Systems: The Future of High Performance Database Systems. *CACM* 35, 6 (1992).

[15] EMC Corporation. 2012. Data Science Revealed: A Data-Driven Glimpse into the Burgeoning New Field. http://www.emc.com/collateral/about/news/emc-data-science-study-wp.pdf. (2012).

[16] Michael T. Goodrich. 1999. Communication-Efficient Parallel Sorting. *SIAM J. Comput.* 29, 2 (1999).

[17] Michael T. Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, Searching, and Simulation in the Mapreduce Framework. In *ISAAC*.

[18] Xiao Hu, Yufei Tao, and Ke Yi. 2017. Output-optimal Parallel Algorithms for Similarity Joins. In *PODS*.

[19] Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication Lower Bounds for Distributed-memory Matrix Multiplication. *J. Parallel and Distrib. Comput.* 64, 9 (2004).

[20] Hong Jia-Wei and H. T. Kung. 1981. I/O Complexity: The Red-blue Pebble Game. In *STOC*.

[21] S. Lennart Johnsson. 1993. Minimizing the Communication Time for Matrix Multiplication on Multiprocessors. *Parallel Comput.* 19, 11 (1993).

[22] Howard Karloff, Siddharth Suri, and Sergei Vassilvitskii. 2010. A Model of Computation for MapReduce. In *SODA*.

[23] Bas Ketsman and Dan Suciu. 2017. A Worst-Case Optimal Multi-Round Algorithm for Parallel Computation of Conjunctive Queries. In *PODS*.

[24] Paraschos Koutris, Paul Beame, and Dan Suciu. 2016. Worst-Case Optimal Algorithms for Parallel Query Processing. In *ICDT*.

[25] Paraschos Koutris, Semih Salihoglu, and Dan Suciu. 2018. Algorithmic Aspects of Parallel Data Processing. *Foundations and TrendsÂő in Databases* 8, 4 (2018).

[26] W. F. McColl and A. Tiskin. 1999. Memory-Efficient Matrix Multiplication in the BSP Model. *Algorithmica* 24, 3 (1999).

[27] A. C. McKellar and E. G. Coffman, Jr. 1969. Organizing Matrices and Matrix Operations for Paged Memory Systems. *CACM* 12, 3 (1969).

[28] S. Melnik, A. Gubarev, J. J. Long, G. Romer, S. Shivakumar, M. Tolton, and T. Vassilakis. 2010. Dremel: Interactive Analysis of Web-Scale Datasets. *PVLDB* 3, 1 (2010).

[29] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. 2008. Pig Latin: A Not-So-Foreign Language for Data Processing. In *SIGMOD*.

[30] Andrea Pietracaprina, Geppino Pucci, Matteo Riondato, Francesco Silvestri, and Eli Upfal. 2012. Space-round Tradeoffs for MapReduce Computations. In *ICS*.

[31] Tim Roughgarden, Sergei Vassilvitskii, and Joshua R. Wang. 2016. Shuffles and Circuits: (On Lower Bounds for Modern Parallel Computation). In *SPAA*.

[32] SortBenchmark [n. d.]. Sort Benchmark Home Page. http://sortbenchmark.org/. ([n. d.]).

[33] A. Thusoo, J. S. Sarma, N. Jain, Z. Shao, P. Chakka, S. Anthony, H. Liu, P. Wyckoff, and R. Murthy. 2009. Hive - A Warehousing Solution Over a Map-Reduce Framework. *PVLDB* 2, 2 (2009).

[34] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *CACM* 33, 8 (1990).

[35] Jeffrey Scott Vitter. 2006. Algorithms and Data Structures for External Memory. *Foundations and Trends in Theoretical Computer Science* 2, 4 (2006).

[36] Jingjing Wang, Tobin Baker, Magdalena Balazinska, Daniel Halperin, Brandon Haynes, Bill Howe, Dylan Hutchison, Shrainik Jain, Ryan Maas, Parmita Mehta, Dominik Moritz, Brandon Myers, Jennifer Ortiz, Dan Suciu, Andrew Whitaker, and Shengliang Xu. 2017. The Myria Big Data Management and Analytics System and Cloud Services. In *Proceedings of the 8th Biennial Conference on Innovative Data Systems Research (CIDR)*.

[37] Mihalis Yannakakis. 1981. Algorithms for Acyclic Database Schemes. In *VLDB*.

[38] Zaharia, M. and Chowdhury, M. and Franklin, M. J. and Shenker, S. and Stoica, I. 2010. Spark: Cluster Computing with Working Sets. In *HotCloud*.