Foundations and Trends® in Databases
Vol. 8, No. 4 (2016) 239–370

DOI: 10.1561/1900000055

now
the essence of knowledge

Algorithmic Aspects of Parallel Data Processing

Paraschos 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

Contents

Abstract

In the last decade or so we have witnessed a growing interest in processing large data sets on large distributed clusters. The idea was pioneered by the MapReduce framework, and has been widely adopted by several other systems, including PigLatin, Hive, Scope, U-SQL, Dremmel, Spark and Myria. A large part of the complex data analysis performed by these systems consists of a sequence of relatively simple query operations, such as joining two or more tables. This survey discusses recent algorithmic developments for distributed data processing. It uses a theoretical model of parallel processing called the Massively Parallel Computation (MPC) model, which is a simplification of the BSP model where the only cost is given by the amount of communication and the number of communication rounds. The survey studies several algorithms for multi-join queries, for sorting, and for matrix multiplication, and discusses their relationships and common techniques applied across the different data processing tasks.

P. Koutris, S. Salihoglu and D. Suciu. *Algorithmic Aspects of Parallel Data Processing*. Foundations and Trends® in Databases, vol. 8, no. 4, pp. 239–370, 2016.
DOI: 10.1561/1900000055.

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 [31], and has been widely adopted by several other systems, including PigLatin [69], Hive [83], Scope [24], Dremmel [65], Spark [91] and Myria [88] to name a few. While the applications of such systems are diverse (e.g., machine learning, data analytics), most involve relatively standard data processing tasks, such as identifying relevant data, cleaning, filtering, joining, grouping, transforming, extracting features, and evaluating results [25, 35].

This has generated great interest in the study of algorithms for data processing on large distributed clusters. This survey reviews some of the recent theoretical results on efficient data processing on large distributed architectures, as well as some of the relevant classical results on parallel sorting and parallel matrix multiplication.

The survey begins in Chapter 2 with a review of parallel models used to analyze algorithms on large distributed clusters. Modern data analytics run on large, shared-nothing clusters, where the cost of communication during data reshuffling can dominate the running time. For example, individual jobs in Cosmos, Microsoft's distributed file sys-

tem, often execute on over 10k nodes [72]. We introduce a very simple model of parallel computation, called the Massively Parallel Computation model (MPC) where the cost of a distributed algorithm is measured in the amount of communication per processor and the number of communication rounds. This model is a simplification of Valiant's *Bulk Synchronous Parallel (BSP)* model [84], and allows us to separate the computation cost from the communication cost, and to focus solely on the latter. In this chapter we introduce the MPC model, then review several important classical models of parallel computation, and discuss their connection to the MPC model.

In Chapter 3 we present and analyze two different approaches for computing in parallel the join of two large relations. Join operations are the bread and butter of most database processing tasks, and the support of efficient join algorithms is a top priority for all major big data systems. We discuss Parallel Hash join, and Parallel Sort Join. The preferred algorithm in practice is the Parallel Hash join, because on most datasets this algorithm is very effective and scales up linearly with the number of processors. However, the Parallel Hash join performs poorly on skewed data, when a large number of records have the same value of the join attribute and, thus, are hashed to the same processor. We discuss in detail how to handle skewed data. In contrast, Parallel Sort join is simpler and less sensitive to skew, but requires extra communication rounds to do the actual sorting.

Next, we consider multi-join queries, and discuss a variety of hash-based algorithms in Chapter 4. In the standard architecture of a database system, a multi-join query is first converted into a query plan, which is then optimized, and finally the plan is executed. The plan consists of simple operators like join, selection, duplicate elimination, and each operator creates an intermediate result that, in distributed query processing, needs to be materialized and re-shuffled for the next operator. Afrati and Ullman [4] pioneered an alternative approach for computing a multi-join query on a distributed system, which computes the query using a single reshuffle operation. Their algorithm, initially described for the MapReduce system, organizes the processors (which correspond to *reducers* in a MapReduce job) in a multidimensional

cube, then partitions each input relation in a sub-cube. The theoretical aspects of the algorithm have been studied in [17], where the algorithm was called *HyperCube*, while extensions to skewed data and to multiple rounds of communication were further discussed in [18, 57]; these will be reviewed in this chapter. While these algorithms are appealing because of their strong theoretical guarantees, modern database systems compute multi-join queries in traditional ways, by converting the query into a join plan. We continue the chapter by discussing the theoretical aspects of join plans, which have a long history in database theory. We review Yannakakis' algorithm for computing acyclic queries [90], the concept of hypertree decomposition [42], and various notions of treewidth [43, 55], and describe how these have been put together in the GYM algorithm [3].

In Chapter 5 we discuss a few traditional aspects of parallel sorting algorithms. Similar to hashing, sorting is a core technique in database query processing, both in the sequential and in the parallel setting. Sort-based techniques suffer less than hash-based techniques from skew in the data. For example, recently Hu, Tao, and Yi [45] have shown how to use sorting to design a simple join algorithm that is provably optimal for any input data (reviewed in Chapter 3). In this chapter we review some fundamental lower bounds for sorting on a distributed system, and also review three classic parallel sorting algorithms: Batcher's odd-even sort [16], Cole's algorithm [27], and Goodrich's algorithm [40].

Finally, in Chapter 6 we discuss classic parallel algorithms for matrix multiplication. We focus on multiplication of dense square matrices and adopt the relational view of matrix multiplication as a join of two tables followed by a group-by-and aggregate computation. Using techniques similar to those used in proving lower bounds in sorting and multi-join queries, we review the communication and round lower bounds for matrix multiplication of square and dense matrices. Then, we review existing algorithms that match these lower bounds. The chapter ends with a very brief overview of other known results in linear algebra, such as multiplication of non-square and sparse matrices, or LU and Cholesky matrix factorization.

Table 1.1 summarizes the notations used in the survey.

Table 1.1: Notations Used Throughout the Survey.

Relation	R_j
Number of relations	ℓ
Variable	x_i
Number of variables	k
Query	q
Input size	IN or N
Output size	OUT
Number of processors	p
Number of communication rounds	r
Load (incoming communication per processor)	L
Memory per processor	M
Total communication	C
Fractional edge cover or edge packing	u_j
Fractional vertex cover or vertex packing	v_i
Fractional edge packing number	τ^*
Fractional edge covering number	ρ^*
Quasi-packing number	ψ^*

2

Models of Parallel Computation

In this survey, we use a simple parallel processing model designed to analyze the communication cost in parallel data processing algorithms, called the MPC (Massively Parallel Computation) model. Specifically, we are interested in the incoming communication cost, per machine, as explained momentarily. The model is a simplification of the well known Bulk Synchronous Processing (BSP) model, introduced by Valiant [84], by modeling only the communication cost and ignoring the computation cost, and it is intended to be an abstraction of the *shared-nothing* architecture [34], which is deployed in most modern massively parallel systems. In this chapter we also review some of the traditional models and concepts for parallel computation.

2.1 The Massively Parallel Computation Model

The model is illustrated in Figure 2.1. We assume a cluster of p machines, each with its own local processing unit and memory. Throughout this survey, we will use the term *processors* to refer to the machines or servers in the cluster. The p processors are connected through a complete network of private channels; this means that every processor can

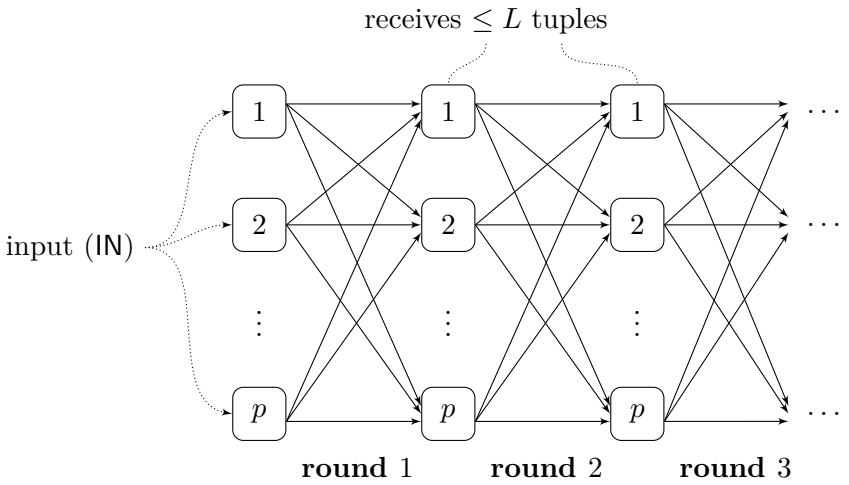


Figure 2.1: The MPC model of parallel computation.

send or receive data from any other processor in the cluster. Computation proceeds in *steps*, or *rounds*, where each round consists of two distinct phases:

Computation Phase: the processors can perform local computations by operating only on the data that is locally available.

Communication Phase: the processors globally exchange data, by communicating (sending and receiving) with all the other processors in the cluster.

The *input data* is initially uniformly partitioned among the p processors. We will use IN to denote the size of the input and, unless otherwise specified, IN is measured in number of data items, or tuples; in some cases we will measure the input in number of bits rather than number of tuples. There are no assumptions on how the data is initially partitioned: in other words, an algorithm should compute the answer correctly no matter how the input data is initially partitioned on the processors. At the end of the computation, we require that the *output data* must be present in the union of the memory of the p processors. This assumption is weaker than requiring that a single processor holds

the whole result, but it is necessary in parallel data processing, because often the output is even larger than the input. Hence, it is natural to allow it to be stored in a distributive way as well. We will typically use OUT to denote the size of the output data.

We will impose no restriction on the computational power of the processors, nor on the data exchange pattern during the communication phase. The parameters of the model consist only of the number of processors, the number of rounds, and the communication cost per processor.

2.1.1 The Parameters of the MPC Model

The parameters of the MPC model are the following:

Number of processors (p). We assume that the number of processors p is fixed throughout the computation, and the p processors can only communicate through the network. Some distributed systems in practice support *elasticity*, which means that they allow the number of processors to change dynamically, but in this survey we assume that p is fixed throughout the computation. However, p is not a constant, instead it is an input to the algorithm, and we will study its complexity as a function of p. As a simple example, in the abstract models of MapReduce [53], p corresponds to the maximum number of mappers during the map phase, or to the maximum number of reducers during the reduce phase.

Number of rounds (r). This parameter refers to the number of rounds r that the algorithm needs to terminate. It measures the number of synchronizations between processors required by an algorithm. Each additional synchronization step carries a cost, since (i) all processors need to wait for the slowest processor (sometimes called *the curse of the last reducer*), (ii) there is a physical cost in keeping track of the progress of each processor, and (iii) in each round special care must be given to load balance uniformly the data across processors. Thus, minimizing the number of rounds r is desirable when we design algorithms.

Load/Communication cost (L). The main parameter of the MPC model is the communication cost, which we will define formally as the maximum amount of data received by any processor during any round. A variant of this cost is the total communication, which is the total amount of data exchanged by all processors during all rounds. Formally, denote by $L_u^{(k)}$ the amount of data that processor $u = 1, \ldots, p$ receives during round $k = 0, 1, \ldots, r$. The *maximum load* is defined as

$$L \stackrel{\text{def}}{=} \max_{u=1}^{p} \max_{k=1}^{r} L_u^{(k)}$$

The *total communication* is defined as the total amount of data that is exchanged throughout the execution of the parallel algorithm:

$$C \stackrel{\text{def}}{=} \sum_{u=1}^{p} \sum_{k=1}^{r} L_u^{(k)}.$$

Notice that $\mathsf{IN} \leq C \leq r \cdot p \cdot L$.

By convention $L_u^{(0)} \stackrel{\text{def}}{=} \mathsf{IN}/p$ for every $u = 1, \ldots, p$, because the input data is initially distributed uniformly across the processors. For that reason, we do not include it in the definitions of L and C above, but will always assume $L \geq \mathsf{IN}/p$. The load of an algorithm in the MPC model depends on the input size, the number of processors, and the number of rounds. In this survey we will describe both upper bounds and lower bounds on the load L.

There are two variants of the MPC model. In a *stateless* MPC model each processor's internal memory is erased after each round. Each new round begins with the main memory containing only the messages, of size $\leq L$, received by the processor in that round. In a *stateful* MPC model the state of the processor's memory is preserved between rounds. Processors in the stateless MPC model can maintain state by sending messages to themselves. In particular, every stateful MPC algorithm with r rounds and load L can be simulated by a stateless MPC algorithm with the same number of rounds and load $L' = r \cdot L$, by having each server sent to itself all messages received during the previous rounds, then redo all the computations done during the previous rounds, before proceeding with the computation of the new round.

When $r = O(1)$ then $L' = O(L)$, and the two models are essentially equivalent. If the MPC is stateless then we allow it to produce parts of the output during each round. If the MPC is stateful, then we can assume w.l.o.g. that the entire output is produced by the processors after the last round, since any output that could be produced earlier can be stored until the end of the computation.

To give some intuition on the interplay between these three parameters, consider two naive algorithms on the MPC model.

Naive Algorithm 1 In the first algorithm, every processor sends its fragment of the input data to processor 1; once processor 1 collects the entire input, it computes the problem locally (since each processor is infinitely powerful). The algorithm terminates in 1 round, and has an optimal total communication cost[1] $C = \mathsf{IN}$. However, the algorithm does not use parallelism at all, and this is captured by the fact that $L = \mathsf{IN}$. Our main objective is to design algorithms with a small load, since that ensures that each processor only has to process a small amount of data. Ideally we would like $L = O(\mathsf{IN}/p)$, but as we shall see we sometimes need to settle for a slightly larger load.

Naive Algorithm 2 The second naive algorithm runs in $p-1$ rounds, and uses a stateful MPC model. During round $k = 1, 2, \ldots, p-1$, processor $k+1$ sends its input data to processor 1. After $p-1$ rounds processor 1 has received the entire input, and can compute the desired output. Here the load is $L = \mathsf{IN}/p$, however we run for a large number of rounds. For that reason we will either consider the stateless MPC model, or consider only $O(1)$ rounds. In fact, our second objective is to design algorithms that run in a small number of rounds, and most algorithms described in this survey run in a constant number of rounds, $r = O(1)$.

Discussion Ideally, we wish to design MPC algorithms that minimize the total running time. The total runtime has two components: (1)

[1]We ignore the initial load per processor, $L_u^{(0)}$, since this is always IN/p, and therefore $C = L_1^{(1)} = \mathsf{IN}$.

the time of the local computation, which depends on the load L, the algorithm used locally, and the physical performance of the processor, and (2) the number of rounds r. The MPC model does not have a formula for the total runtime (which would be system specific), but instead aims to optimize both parameters, load L and number of rounds r, thus it is a multi-objective optimization problem. In this survey, we will always optimize the load L, under two scenarios: when $r = 1$, and when r is unbounded, but $r = O(1)$, in other words r is any constant independent on the input size and the number of processors.

2.1.2 Speedup and Scaleup

The MPC model is designed as an abstraction for the shared-nothing architecture, which is the architecture used today by large data processing systems. Shared-nothing parallel database systems (and in general all parallel database systems), measure parallelism using two simple parameters, speedup and scaleup [34]. For a fixed input, the *speedup* is the speed of the algorithm as a function of the number of processors p. The ideal is a linear speedup, i.e., the speed should double when we double the number of processors, but in practice one almost always observes a sublinear speedup. *Scaleup* is the speed of the algorithm when both the input data and the number of processors increase at the same rate. The ideal scaleup is constant, i.e., when the data size doubles, then we should be able to keep the same performance by doubling the number of processors, but in practice scaleup is sub-constant.

In the MPC model, both speedup and scaleup depend on the load of an algorithm when using different number of processors, and the local processing time at each processor. Let $L(\mathsf{IN}, p)$ be the load of an algorithm when using p processors on an input of size IN (assuming the number of rounds r is fixed). Since the cost of the local computation is unspecified in the MPC model, for the purpose of discussing the speedup and scaleup we will simply assume that the local algorithm takes linear time in the amount of data received during one round, i.e., Time $= O(L)$. Thus, speedup is given by $f(p) = L(\mathsf{IN}, 1)/L(\mathsf{IN}, p) = c/L(\mathsf{IN}, p)$ for some constant c independent of p , while scaleup is given by $g(t) = L(\mathsf{IN}, p)/L(t \cdot \mathsf{IN}, t \cdot p) = c'/L(t \cdot \mathsf{IN}, t \cdot p)$. Most algorithms

discussed in this survey have loads of the form $L = O(\mathsf{IN}/p^{\delta})$, for some $0 < \delta \leq 1$. Their speedup is $\sim p^{\delta}$ and their scaleup is $\sim t^{\delta-1}$. For $\delta = 1$ we have linear speedup and constant scaleup, while for $\delta < 1$ the speedup is sub-linear and the scaleup is sub-constant.

2.2 Other Models of Parallel Computation

Parallel computation has been studied since the inception of computer science. We briefly discuss here some well studied models of parallel computation, and their relationships to the MPC model.

2.2.1 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. Each output is computed by one distinguished node of the DAG. For example, a Boolean function $F : \{0,1\}^N \to \{0,1\}$ is computed by some circuit whose leaves correspond to the input variables $x_1, \ldots, x_N$, and with a single designated output node, which can be assumed w.l.o.g. to be the root of the DAG. When all internal nodes are restricted to the Boolean operations AND, OR, and NOT then we call it a Boolean circuit, but in general, the nodes may represent more complex computations, with multiple inputs and multiple outputs. The size of the DAG (number of nodes) is called the *work* W, and the depth of the DAG is called the *depth* D. The intuition (to be made precise soon) is that the nodes can perform their computation in parallel, the only constraint being that a node needs to wait for its children to finish their computation before it can proceed. Thus, the number of parallel steps in this idealized computation model is the depth of the circuit, D. The *degree* or the *fan-in* of the circuit is defined as the largest in-degree of any node. The degree affects the depth significantly:

Theorem 2.1. The following hold:

1. Any Boolean function $F : \{0,1\}^N \to \{0,1\}$ can be computed by a Boolean circuit of depth 3 and unbounded degree.

2. If F is any function for which at least one of its outputs depends on all its inputs, then any circuit of degree s computing F has depth $\geq \log_s N$.

3. There exists a Boolean function F such that any circuit of degree s computing F has depth $\geq (N - s - \log(1 + (\log N)/2^s))/\log s$.

Proof. Item 1 follows by writing F in DNF: the Boolean circuit has one NOT node for each Boolean variable, one AND-node of degree N for each term in the DNF expression, and one OR-node of exponential in-degree computing the result of the DNF expression. Item 2 follows from the fact that any circuit of depth D can have at most s^D leafs (which can be shown by induction), and therefore if the output depends on all N inputs, we must have $s^D \geq N$. We prove item 3 by counting the number of distinct Boolean functions representable by a circuit of depth D, using a standard counting argument (see e.g. [52, pp.22]). Assume w.l.o.g. that each node has fan-out 1 (replace each node with fan-out s' with s' nodes with single output, without increasing the depth) and that the circuit is a tree. There are $(s^D - 1)/(s-1) \leq s^D$ internal nodes, each labeled with one of 2^{2^s} possible Boolean functions and there are s^D leaves, each labeled with one of N possible inputs, hence the total number of such functions is $\leq (2^{2^s})^{s^D} \cdot N^{s^D}$. On the other hand, the total number of Boolean functions is 2^{2^N}, and the claim follows from $(2^{2^s})^{s^D} \cdot N^{s^D} \geq 2^{2^N}$. □

We illustrate item 2 of the Theorem on two examples. The first is the Boolean function $\mathsf{OR} : \{0,1\}^N \to \{0,1\}$. The function depends on all its N inputs and, therefore, if we restrict the gates of a circuit to have fan-in 2, any circuit computing OR requires $D \geq \log N$. On the other hand, item 3 proves that there exists a Boolean function for which any Boolean circuit has depth $\Omega(N)$. The second is the sorting function $\mathsf{SORT} : \{0,1\}^N \to \{0,1\}^N$ which takes as input N bits and returns the same bits in sorted order. This, too, requires $D \geq \log N$, because its last output bit is the function OR.

Circuits and the MPC model An important distinction between MPC and circuits is that in an MPC algorithm the messages can be dynamic,

i.e., a server may decide where to send a message depending on the input data or on the messages it has received in the previous round. We call an MPC algorithm *oblivious* if the content of each message sent from a sender u to a receiver v during round k depends only on the input size IN, and not on the actual input. In other words, in an oblivious MPC model, the destinations and sizes of the messages sent during each round are pre-determined. The following holds:

Theorem 2.2. (1) Any circuit of size W, depth D, and fan-in s can be simulated by an oblivious MPC algorithm with $p = W$ processors, load $L = s$, and $r = D$ rounds. (2) If a problem can be solved by an oblivious MPC algorithm with p processors, load L and r rounds, then it can be represented by a circuit with $W = p \cdot r$ nodes, fan-in $s = L$, and depth $D = r$.

Proof. (Sketch) (1) Each processor simulating a node W waits until it receives the input messages from its children, then computes the function represented by that node, and sends its outputs to the parents. Since the parents are uniquely determined by the circuit, the algorithm is oblivious. (2) Associate to each processor u and each round k a unique node with L inputs (the messages received by u during round $k-1$), and multiple outputs (the messages sent by u during round k). Since the algorithm is oblivious, we can uniquely wire the outputs of this node to the inputs of the next round $k+1$. □

However, non-oblivious MPC algorithms can be strictly more powerful than circuits with the same parameters. Intuitively, the additional power comes from the ability of the MPC model to send messages anywhere depending on the input. In particular, processors can communicate by *not* sending a message. In contrast, in a circuit the wires between the gates must be fixed, hence the circuit is always oblivious.

In fact, we can prove formally the mismatch between the load L of the MPC model and the fan-in s of a circuit.

Theorem 2.3. Fix $L \geq \log N$. (1) Any Boolean function $F : \{0,1\}^N \to \{0,1\}$ can be computed by a non-oblivious MPC algorithm with $p = 2^N N/L$ processors in $r = \log_L N$ rounds, using load L. (2) There exists

a Boolean function for which any circuit of degree L has depth $\Omega((N - L - \log\log N)/\log L)$.

The theorem says that the non-oblivious algorithm requires a much smaller number of rounds than the depth of a circuit; this follows from $\log_L N \ll (N - L - \log\log N)/\log L$, when N is sufficiently large.

Proof. (of Theorem 2.3) (1) The proof is adapted from [74], and resembles the proof of item 1 of Theorem 2.1. The MPC algorithm has two parts. Part 1: in parallel for each assignment $\theta \in \{0,1\}^N$ on which $F[\theta] = 1$, check if the input is equal to θ. Part 2: compute the OR of those results. We explain Part 1. For each $\theta \in \{0,1\}^N$, the input can be compared with θ using $p = N/L$ processors in $\log_L N$ rounds: initially each of the N/L processors compares L input bits with the corresponding bits in θ, then the processors use $\log_L N$ rounds to compute the AND of those results. Notice that this part of the MPC algorithm is oblivious, and can also be realized by a circuit. Part 2 requires non-oblivious communication: the single processor that has uniquely identified the assignment θ equal to the input, sends a message of 1 bit to processor 1, and all other processors send nothing. In the last round, if processor 1 receives exactly one message then it outputs $F = 1$ and, if it does not receive any message then it outputs $F = 0$. Thus, part 2 requires a single communication round. In contrast, if we had to implement it using OR gates of degree L, the depth of the circuit could be a large as $N/\log L$, because the number of assignments θ s.t. $F[\theta] = 1$ can be as high as 2^N, which means that its depth is $\geq \log_L 2^N = N/\log L$.

(2) Follows immediately from Theorem 2.1 item 3, by setting $s = L$ and noting that $1+(\log N)/2^s \leq \log N$, whenever $s \geq 2$ and $N \geq 4$. □

The Complexity Classes NC^k and AC^k Fix a number $k \geq 0$. A decision problem is in NC^k if for every $N > 0$ there exist a *uniform boolean circuit*[2] with bounded fan-in, whose size is polynomial in N and whose depth is $\log^k N$, that solves all instances of size N of that problem. The acronym NC stands for Nick's Class. If a problem is in

[2]A uniform circuit is one that, given N, can be constructed in deterministic logarithmic space.

NC^k then it is also in polynomial time: indeed, given an input of size N, start by generating the DAG (here we use the uniformity), then evaluate the circuit bottom up. The class AC^k is defined similarly, by allowing the gates to have unbounded fan-in. By observing that we can expand a node with fan-in s to a tree with depth $\log s$ and bounded fan-in, we have that $\mathsf{NC}^k \subseteq \mathsf{AC}^k \subseteq \mathsf{NC}^{k+1}$: notice that $\log s = O(\log N)$, because the size of the circuit is polynomial in N. The class NC is defined as $\mathsf{NC} = \bigcup_k \mathsf{NC}^k = \bigcup_k \mathsf{AC}^k$.

NC is considered to capture the class of parallelizable problems. As we saw, $\mathsf{NC} \subseteq \mathsf{P}$, but it is believed that the converse does not hold. Proving that a problem is *not* in one of the classes $\mathsf{NC}^k, \mathsf{AC}^k$ is a difficult task. A celebrated result by Sipser [37] is that the parity (denoted XOR) function is not in AC^0, even when we allow non-uniform circuits.

Relational Queries are in AC^0 Most of this survey is concerned with computing relational queries in the MPC model. It is well known that every query expressible in First Order Logic is in AC^0.

Theorem 2.4. Let φ be a sentence in First Order Logic. Then the problem *given an input database D, is φ true in D?* is in AC^0.

We omit the proof, and refer the reader to [60].

While AC^0 consists only of decision problems, the main idea in Theorem 2.4 extends to non-Boolean queries: every relational query can be computed by a circuit of depth $O(1)$. We illustrate this here using a very simple query: the join of two relations, $R(A,B) \bowtie S(B,C)$. More precisely, we are given a set R of tuples (i,j), and a set S of tuples (j,k), and we have to output all triples (i,j,k) such that $(i,j) \in R$ and $(j,k) \in S$. Fix two finite input relations R, S, and let n denote the size of their active domain (i.e. the total number of constants occurring in both relations). We encode R using n^2 Boolean variables x_{ij}, where $x_{ij} = 1$ if $(i,j) \in R$, and $x_{ij} = 0$ otherwise; similarly, we encode S using n^2 Boolean variables y_{jk}. Thus, the size of the input is $N \stackrel{\text{def}}{=} 2n^2$, and the size of the output is n^3; in other words, the join is a function $F : \{0,1\}^{2n^2} \to \{0,1\}^{n^3}$. This can be computed with n^3 AND gates of arity two: $x_{ij} \wedge y_{jk}$ for $i,j,k \in [n]$. Finally, consider the following simple

example of a sentence φ in First Order logic: $\varphi = \exists x \exists y \exists z (R(x, y) \wedge S(y, z))$. Checking whether φ holds on an input database is a decision problem, $F : \{0,1\}^{2n^2} \to \{0,1\}$, and by Theorem 2.4, it must be in AC^0. Indeed, it can be computed using the circuit described above, extended with one additional OR gate with fan-in n^3: $F = \bigvee_{i,j,k}(x_{ij} \wedge y_{jk})$. Thus, the problem is indeed in AC^0.

2.2.2 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. In this model there are p processors, each with its own local state consisting of a small number of registers, and with access to a shared memory. At each time step, a processor can perform (any of) the following 3 operations: (*i*) read one shared memory cell, (*ii*) perform an operation on the local memory, and (*iii*) write to one shared memory cell. The parallel complexity is the time required for all processors to finish their computation.

Like the RAM model (with a single processor) there are two variants [66]: in the *bit model* the cost of a single operation performed by a processor is $\log n$, where n is the number of bits representing those values. In the *unit cost model* each operation is assumed to take constant time. The unit cost model is unrealistic for proving lower bounds, because it permits us to encode very large data structures as integers, and perform costly operations in small number of steps using arithmetic operations. Therefore, the lower bounds are usually expressed in the bit model, but the algorithms are more conveniently described in the unit model.

Three types of PRAM models are studied in the literature. In the Exclusive-Read-Exclusive-Write (EREW) model, only one processor is allowed to access any shared memory location at any time. In the Concurrent-Read-Concurrent-Write (CRCW) model, processors are allowed to read, or to write the same memory location concurrently. Finally, CREW allows concurrent reads, but exclusive writes. Models that allow concurrent writes are further distinguished by how they handle

conflicts between the different kind of writes. The difference between exclusive and concurrent writes is illustrated by the OR function: given an array of N values in $\{0, 1\}$, compute and store their OR at a particular address in the main memory. Suppose we have $p = N$ processors. In the ERCW model, the OR function can be computed in two parallel steps. In the first step one processor (say, processor 1) writes a 0 at the output location, and in the second step each processor reads one data item, and, if that item is 1, then writes 1 in the output location. Notice that multiple processors may write the same value 1 in the output, hence the need for concurrent writes. In the EREW model, and even in the CREW model, any algorithm requires $\log N$ parallel steps to compute the OR of N items [29].

Brent's Theorem A fundamental connection between the circuit model and the PRAM model is given by Brent's theorem:

Theorem 2.5 (Brent's Theorem [20])**.** If an algorithm can be represented by a circuit of work W and depth D, then it can be computed by a CREW PRAM model with p processors in time $T_p = W/p + D$.

The proof consists of a direct simulation of the DAG by the CREW PRAM. The p processors evaluate the DAG level by level, starting from the leaves and moving up. At each level i with W_i nodes, each processor performs the computation of $\lceil W_i/p \rceil$ nodes from that level. Thus, the total parallel time is $\sum_{i=1}^{D} \lceil W_i/p \rceil \leq \sum_{i=1}^{D} W_i/p + D = O(W/p + D)$.

Sometimes the size of the circuit (work) is denoted by $W = T_1$, since it is the time taken by a single processor to evaluate the circuit, while the depth is denoted by $D = T_\infty$, meaning the (theoretical) time assuming we have an unbounded number of processors. The implication of Brent's theorem is that we can never expect to run in time lower than T_∞; hence the depth D represents the intrinsic parallel cost of the problem.

Discussion of PRAM and MPC The most important distinction between the PRAM and MPC models is in the way they measure time. The parallel time in PRAM has a fine granularity, while the number

of rounds in MPC corresponds to super-steps (introduced by the BSP model, to be discussed next) and are usually at a higher granularity. In the PRAM model, communication happens through the shared memory, and the processors communicate at each step. In the MPC model, after one communication round the processors usually perform a significant amount of computation before the next communication round. Another distinction is that in the PRAM model the local state of a processor has size $O(1)$ (it usually consists of a small number of registers), while in the MPC model the local memory of each processor is $\geq L$, because it needs to store the incoming data, and also $\geq \mathsf{IN}/p$, because the p servers need hold the entire input data. In the PRAM model, more processors means *less* parallel steps (by Brent's theorem), while in the MPC model an increase of the number of processors is usually accompanied by decrease of the local memory, and of the load L, and results in *more* communication rounds. An extreme case is when $p = 1$. In this case the PRAM algorithm is at its slowest, since it degenerates to a sequential algorithm, while the MPC algorithm is at its best, since it requires no communication rounds (but the load, and local memory are at their worst).

MPC is better suited to explain the cost of parallel query processing on distributed architectures, because it can better explain the sublinear speedup (and subconstant scaleup) observed in practice. For example, consider the query that joins two relations, which, we have seen, can be expressed as a circuit of depth $D = O(1)$. By Brent's theorem, it takes $T_p = O(W/p + D)$ parallel time on a PRAM, hence the speedup is almost linear: $T_1/T_p = (W + D)/(W/p + D) \approx p$. In contrast, as we shall see in Section 3.1.5, the speedup of the parallel hash-join on the MPC model is linear only for small values of p, and degrades to sub-linear, $\sim p^{1/2}$, because of the presence of skew in the data.

2.2.3 The BSP Model

The PRAM model corresponds to a shared-memory architecture. In practice, shared-memory systems do not scale up to very large number of processors, because of the difficulty of accessing a single shared memory, and therefore p is limited to at most a few hundred processors.

Instead, large parallel systems use a shared-nothing architecture, which consists of a large number of independent processors connected by a fast communication network. To model such systems, Valiant introduced the *Bulk Synchronous Parallel (BSP)* model [84]. 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. The running time of a BSP algorithm is characterized by two parameters [40]: T_C, the number of communication rounds, and T_I, the total internal computation time. The total running time of an algorithm is then $O(T_I + (L + gh)T_C)$, where L represents the latency of the network, and g represents the gap between sending to data items on the network. Several flavors of the BSP model have been considered, see Goodrich [40]: in the EREW model (which is the only one discussed by Valiant [84]) each data item is sent to a single destination; the more powerful CREW model allows one data item to be broadcast to an arbitrary number of destinations; while the Weak-CREW model allows a data item to be broadcast to a set of processors of the form $\{i, i+1, \ldots, j\}$. A refinement of the BSP model is the LogP Model [30], which allows the communications to overlap with computation and uses a more detailed (and more complex) formula for computing the cost of an algorithm.

The MPC model in this survey is a simplification of the CREW BSP model that omits the internal computation time, and the network parameters (latency and gap).

2.2.4 MapReduce-Based Models

Following the introduction of MapReduce [31], several theoretical models were introduced in the literature to capture computations in this setting. MapReduce is a system based on the BSP model, where synchronization occurs at every step, and uses a simple programming model inspired by functional programming. In its vanilla version, the user defines two functions: `map` and `reduce`. The map function is applied

on a single data item x of the input and returns a set of key-value pairs (k, v). The reduce function is applied to a list of items with the same key, $(k, [v_1, \ldots, v_m])$ and returns a new list of values. During runtime, the system applies in parallel the map function, then performs a *shuffle step* that collects all key-value pairs with the same key to the same *reducer*; finally, the reduce function is applied in parallel to each key-group.

Karloff et al. [53] define the MapReduce class ($\mathcal{MRC}$) of algorithms. Denoting by IN the size of the input in bits, the model restricts both the memory per reducer and the number of reducers to $O(\mathsf{IN}^{1-\epsilon})$, for some constant $\epsilon > 0$. Reducers in this model correspond to processors in MPC. Observe that this means that the number of reducers is not explicitly defined. The reducers are restricted to perform only polynomial time computations, and further the number of rounds must be limited to $O(\log^c(\mathsf{IN}))$ for some constant c.

Goodrich et al. [41] use a similar variation on the BSP model. It has an explicit parameter M to bound the memory of each machine, and bounds not only the incoming data in each machine, but also the the outgoing data from each machine.

Afrati et al. [2] developed a model for MapReduce where the main parameter is an upper bound q on the number of input tuples a reducer can receive, called *reducer size*; this translates to the load L in our model. Given an input of size IN, a MapReduce algorithm is restricted to deterministically send each input tuple independently to some reducer, which will then produce all the outputs that can be extracted from the received tuples. If $q_i \leq q$ is the number of inputs assigned to the i-th reducer, where $i = 1, \ldots, p$, the *replication rate* of the algorithm is $r = \sum_{i=1}^{p} q_i/\mathsf{IN}$. Since the total communication of such an algorithm is $C = \sum_{i=1}^{p} q_i$, the replication rate captures the total communication cost normalized by the input size IN.

2.2.5 Communication Complexity

The MPC model can be viewed as a type of communication model, studied in communication complexity [58]. The simplest communication model is the two-party communication: Alice and Bob want to

compute a Boolean function $F(X, Y)$, where $X \in \{0,1\}^n$, $Y \in \{0,1\}^n$. The inputs X are given only to Alice, and the inputs Y are given only to Bob. The central question in *communication complexity* is to determine the minimum number of bits that Alice and Bob need to exchange in order to both compute the function F. A more refined question is to also determine the number of rounds needed for the exchange. Several generalizations exist, including multiparty communication, where the input is split into p different parties, and there exists a rich literature on lower bounds for various problems and models.

In the terminology of communication complexity, the MPC model assumes p parties, each holding a part of the input. The p parties want to cooperate to compute a function, with the goal of minimizing the number of rounds and the amount of incoming data per party. Hence, MPC can be viewed as one particular instance of the communication model, namely the multiparty, private channel (meaning that each processor may communicate directly with any other processor), number-in-hand model (meaning that each processor sees its local data). One difference between the MPC model and most settings in communication complexity is that, in the latter case, exchanging the entire input leads to a trivial protocol, and thus all lower bounds in communication complexity are strictly less than IN; in contrast, all lower bounds on the total communication in the MPC model are superlinear, and are typically of the form $\mathsf{IN} \cdot p^{\varepsilon}$, for some $\varepsilon > 0$.

2.3 Comparing Sequential and Parallel Algorithms

As a general principle, it is easy to turn a parallel algorithm into a sequential algorithm, but not vice versa. This means that it is much harder to design efficient parallel algorithms than efficient sequential algorithms, for most reasonable interpretations of "efficient". While there exist generic translations from parallel to sequential algorithms, one should expect no such generic translation in the opposite direction. For example, we have seen that every problem in NC is also in polynomial time, by using simple simulations of an NC circuit by a polynomial time algorithm. The converse is unlikely to hold, unless $\mathsf{NC} = \mathsf{PTIME}$.

We contrast here sequential and parallel algorithm in two settings. First, we show that every MPC algorithm can be converted to an efficient algorithm in the External Memory model. Second, we show a surprising generic translation in the reverse, from sequential to parallel algorithm: the catch is that the parallel computation is "efficient" only in its memory usage, but not in computation time.

2.3.1 From MPC to the External Memory Model

Figure 2.2 shows an overview of the External Memory model (EM). The model consists of a processor with a fixed amount of main memory M, and an external disk of unbounded size. Each I/O operation involves reading or writing one block of size B from or to disk. The complexity of an algorithm is measured solely in terms of number of block I/Os during the computation. The model has also been extended to multiple processors, which can compute in parallel, and to multiple disks, which can perform parallel I/O; see [86] for a survey. A lower bound for any problem where the output depends on the whole input is IN/B, since the entire input of size IN needs to be read into the main memory. If $\mathsf{IN} \leq M$, then the I/O complexity is trivially IN/B, since the entire data can be read in main memory; it is usually assumed that the processor is powerful enough to do arbitrary computations once the data is in memory. Thus, a usual assumption[3] is that $\mathsf{IN} > M$. The exact I/O complexity of several problems is known for the EM model. For example, the optimal I/O complexity for sorting in the EM model [86] is $O(\frac{\mathsf{IN}}{B} \log_{M/B} \frac{\mathsf{IN}}{B})$.

Consider an algorithm in the MPC model where each processor uses local memory of size M. We assume $M \geq L$, so that the message received by the processor can be stored in main memory. There exists a simple simulation of the algorithm by the EM model [57]. This can be used both to obtain efficent algorithms in the EM model, and to prove lower bounds for the MPC model.

Theorem 2.6. If a problem can be solved in the MPC model with load L, local memory M, $L \leq M$, and total communication C, then, for all

[3] A second assumption is that $M > 2B$, meaning that we have at least two buffers in main memory, one for reading and one for writing.

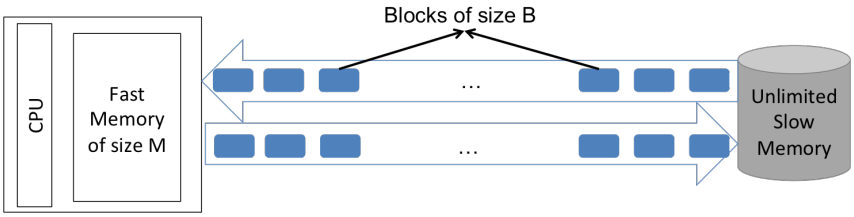


Figure 2.2: External Memory Model.

$B \leq L$, it can be solved in the EM model with block size B, memory size M, and I/O complexity

$$O\left(\frac{C}{B}\log_{M/B}\frac{C}{B}\right)$$

Proof. The EM algorithm simulates the MPC algorithm one round at a time. During the simulation of round k, all messages sent by all of the previous round are stored on disk. Denote C_k their total size, such that $\sum_k C_k = C$. We assume that each message includes its destination server. The simulation of round k starts by sorting the C_k message in increasing order of their destination; this ensures that all data items with the same destination processor u are packed into continuous blocks on disk. Then, the algorithm simulates each processor $u = 1, \ldots, p$; it reads from disk the blocks containing messages sent to u during round $k-1$, performs the computation for round k, then writes out to disk all message sent by u during the end of phase k. The I/O complexity each of each of the r rounds is the I/O complexity of sorting, which is $O(C_k/B \log_{M/B}(C_k/B))$, proving the theorem. □

2.3.2 From Streaming to Parallel Algorithms

A surprising translation from *order-independent* streaming algorithms to a parallel algorithm is described by Feldman et al. [36]. A streaming algorithm is similar to an algorithm in the EM model, except that it can only read the input data once, sequentially. There is a single processor, with a local memory of size M. The processor reads the input data sequentially, item by item. Each item is processed, in the local memory, then the next item is read. At the end of the input stream, the algorithm

produces some output. The assumption is that the input sequence is much larger than M, the local memory of the processor. The streaming algorithm is required to be order-independent, or symmetric, i.e., its final answer should be independent of the order of the input elements.

A MUD[4] algorithm uses p processors, each with local memory of size M, and arranged in a binary tree. Each leaf of the tree holds one input data item; they perform some local computation, then send some message to their parent processor. In general, each processor waits for inputs from its two children, performs some local computation, sends a message to the parent, and stops. When the root receives its two input messages, it performs the local computation and produces the final answer. Here, too, the assumption is that the input sequence is much larger than M. One can view a MUD algorithm as implementing a divide-and-conquer tree: the root partitions the data into two, processes them recursively, then combines the two results. The MUD algorithm is required to be both order-independent, and also independent on the tree, i.e., independent of the way the data is partitioned during the divide-and-conquer process.

It is easy to see that every MUD algorithm with memory M can be converted into a streaming algorithm with the same amount of memory. Indeed, the MUD algorithm must compute the output correctly no matter how the processors are organized in a binary tree, in particular it needs to compute correctly if its processors are organized in a left-deep linear tree. The streaming algorithm can simulate this tree using a single processor, by simulating the MUD processors bottom up. This is consistent with our general observation that parallel algorithms can be converted into sequential algorithms.

In fact, a result in database query languages has shown that, if the computations performed by the processors are restricted to Relational Algebra, then the streaming algorithms can express precisely the queries that are computable in PTIME, while the MUD algorithms can express precisely the queries computable in NC [79].

Given this, the result in [36] comes at a surprise: they showed that every order-independent streaming algorithm can be converted into

[4]MUD stands for *massively unordered distributed*.

a divide-and-conquer algorithm that uses the same amount of memory M. The catch is that the divide-and-conquer algorithm is a non-deterministic algorithm: by Sawitch's theorem, it can be converted into a deterministic algorithm using M^2 memory, but the latter may no longer be in PTIME. The construction in [36] is actually very simple. Each processor in the MUD algorithm will simulate the computation of the stream algorithm on the input $\mathbf{x} = x_i x_{i+1} \cdots x_j$ stored on the leaves of its subtree, as if this were a complete sequence (and not part of a larger sequence). If by processing the sequence $\mathbf{x}$ the streaming algorithm enters a state q, then the goal of the MUD processor is to produce the same state q. Let $\mathbf{x}'$ and $\mathbf{x}''$ denote the subsequences of the left and right child respectively. If these children could sent a message with the entire input sequence, then it is easy for the MUD processor to simulate the streaming algorithm on $\mathbf{x}$ and compute q, but the size of $\mathbf{x}$ is much larger than its local memory, M. Instead, the processor receives two messages q', q'' from its children, each of size M, representing the state of the streaming algorithm if it processes $\mathbf{x}'$ independently, and $\mathbf{x}''$ independently respectively. Given q', q'', the MUD processor guesses two input sequences $\mathbf{y}'$, $\mathbf{y}''$ on which the streaming computation returns q' and q'' respectively, concatenates them to obtain $\mathbf{y} = \mathbf{y}'\mathbf{y}''$, then simulates the streaming algorithm on $\mathbf{y}$ and returns the result q. Thus, the streaming algorithm does not know the sequence $\mathbf{x}'$ of its left child, but guesses some other sequence $\mathbf{y}'$, on which the streaming algorithm produces the same state q'; similarly for the right child. An elegant (and simple) proof in [36] shows that the output produced by the streaming algorithm on $\mathbf{y} = \mathbf{y}'\mathbf{y}''$ is the same q as that produced on $\mathbf{x} = \mathbf{x}'\mathbf{x}''$, proving that the MUD processor computes the correct output. The reader may have noticed that the MUD processor cannot not guess the sequences $\mathbf{y}'$ and $\mathbf{y}''$ and store them in its main memory. Instead, the processor guesses its items one by one. It guesses items in $\mathbf{y}'$ until the streaming computation reaches q'. (The length $\mathbf{y}'$ could be bounded by 2^M, but this is not necessary for a non-deterministic algorithm.) Then it starts guessing items in $\mathbf{y}''$, and performs two separate simulations of the streaming algorithm. In one simulation it starts from its initial state, processes $\mathbf{y}''$, and stops if and when it reaches q''. In the other

simulation it starts from the state q': as it processes items in $\mathbf{y}''$, it computes the states that correspond to the concatenation $\mathbf{y}'\mathbf{y}''$. When the first simulation stops, then the second has reached the desired state q, which is the final result of the computation.

3

Two-way Join

We begin our survey of distributed algorithms by illustrating how to compute the natural join of two large relations,

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

This is a very common operation in data management, for example it can be expressed in SQL as:

```
SELECT *
FROM   R, S
WHERE  R.B = S.B ;
```

We will describe *hash-based* and *sort-based* algorithms for the join. We denote the input size by $\mathsf{IN} = |R|+|S|$, and recall that both relations are initially arbitrarily but evenly partitioned among the p processors: each processor holds IN/p tuples from either one, or from both relations. Our goal is to design algorithms that compute the join in parallel, in a constant number of rounds, while minimizing the load L. We will assume that the input size is much larger than the number of processors, $\mathsf{IN} \gg p$, which is typically the case when processing big data. The algorithms we review in this chapter are used in current distributed

systems; we discuss how they are implemented in existing systems at the end of the chapter.

3.1 Hash-Based Algorithms

Most parallel data management systems compute joins using the *parallel hash-join*. We describe here this algorithm and analyze its load. As we shall see, the basic algorithm is simple and effective, but fails to handle skewed data, and, in fact, it can be prohibitively inefficient when the data is skewed. To handle skewed values, the algorithm needs to be modified and becomes more involved.

3.1.1 Parallel Hash-Join

The algorithm uses a hash function h that maps elements from the domain $\mathbf{dom}(B)$ of the join attribute B to $[p] = \{1, \ldots, p\}$. The hash function is chosen randomly at the beginning, and we assume it is known to all p processors. The algorithm proceeds as follows. In parallel, each of the p processors iterates over its local tuples. Each tuple r from R is then sent to the processor $h(r.B)$, and each tuple $s \in S$ is sent to processor $h(s.B)$. After the tuples have been sent to the specified destination, each processor computes the join on its local data.

The algorithm correctly computes the join, since any two tuples that join have been sent to the same processor. It also terminates in a single round, and, since each tuple is sent to exactly one processor, the total communication is $C = \mathsf{IN}$, which is the smallest possible value. We next turn our attention to the analysis of the maximum load L.

Fix one processor $u \in [p]$, and denote by L_u^R, L_u^S the number of R-tuples and S-tuples respectively received by u. Since we treat the hash function h as random, the quantities L_u^R, L_u^S are random variables. Their expectation are $\mathbb{E}[L_u^R] = |R|/p$, $\mathbb{E}[L_u^S] = |S|/p$, since each tuple is sent to this processor with the same probability $1/p$. Thus, the expected load at any fixed processor is

$$\mathbb{E}[L_u] = \mathbb{E}[L_u^R + L_u^S] = \mathbb{E}[L_u^R] + \mathbb{E}[L_u^S] = \mathsf{IN}/p.$$

This is the best possible value. However, the maximum load across all processors can be much larger. To see this, consider an extreme case,

where all tuples in the relations $R(A, B)$ and $S(B, C)$ have the same value of the attribute B. Then, for any choice of hash function h, the entire input is sent to only one processor u, and therefore $L_u = \mathsf{IN}$ and $L_v = 0$ for $v \neq u$. On this particular database instance, the algorithm's load is $L \stackrel{\text{def}}{=} \max_u L_u = \mathsf{IN}$.

A first question is to find a sufficient condition on the database that implies $\mathbb{E}[L] = O(\mathsf{IN}/p)$. The intuitive answer is that the database instance must have "no skew", meaning that no value of the B attribute occurs too often. A second question is how to modify the join algorithm to handle efficiently the case when the database is skewed; we will proceed in two steps by first describing an optimal algorithm for a Cartesian product (the extreme case of a skewed join) then give the general join algorithm.

3.1.2 Analysis of the Skew-free Case

We start with a very simple case, when each value of the attribute B appears only once in the database. This looks like a non-interesting case, since the result of the join is empty (because each value b occurs either in R or in S, but not in both), but it is a good warm-up for us. Fix a processor u. For a tuple t in the database, let X_t denote the indicator variable that is 1 if t is sent to u, and 0 otherwise. In other words, $X_t = 1$ iff $h(t.B) = u$. Because h is a random hash function, X_t is a random variable, and $\mathbb{E}[X_t] = 1/p$. The load is then simply $L_u = \sum_t X_t$, and its expectation is $\mathbb{E}[L_u] = \mathbb{E}[\sum_t X_t] = \mathsf{IN}/p$. Since the random variables $\{X_t\}_t$ are independent (because the values $t.B$ of the tuples are distinct) we can apply Chernoff's bound and prove that L_u is highly concentrated around its expectation: for any $\delta \in [0, 1]$,

$$\mathbb{P}[L_u \geq (1+\delta)\frac{\mathsf{IN}}{p}] = \mathbb{P}[\sum_t X_t \geq (1+\delta)\frac{\mathsf{IN}}{p}] \leq e^{-\frac{\delta^2 \mathsf{IN}}{3p}}$$

Applying a union bound over all the processors gives us

$$\mathbb{P}[L \geq (1+\delta)\frac{\mathsf{IN}}{p}] = \mathbb{P}[\max_u L_u \geq (1+\delta)\frac{\mathsf{IN}}{p}] \leq pe^{-\frac{\delta^2 \mathsf{IN}}{3p}} \tag{3.1}$$

Assume δ is fixed. Then, if IN/p is sufficiently large (in particular, larger than $3/\delta^2$), then $L \leq (1+\delta)\mathsf{IN}/p$ with probability approaching 1

exponentially fast in IN/p. Notice that this analysis is useful only when $\mathsf{IN} \gg p$; when $\mathsf{IN} \approx p$ then one can show that $L \approx (\mathsf{IN}/p)\log p$.

Next, we drop our simplifying assumption, and allow each value of B to occur multiple times. Given a value $b \in \mathbf{dom}(B)$, its *degree* is the number of tuples t in the database that have $t.B = b$; in other words, the degree of b is the number of times b occurs in column B. Let d denote the largest degree of any value. Obviously, a necessary condition for $L = O(\mathsf{IN}/p)$ is that $d = O(\mathsf{IN}/p)$, because all tuples with $t.B = b$ are sent to the same processor and therefore $L \geq d$. We prove next a weak form of the converse.

To simplify our discussion, assume that the database is perfectly uniform, meaning that every value b in the active domain of B occurs exactly d times in the database. This assumption simplifies the exposition, and is w.l.o.g. (we refer the reader to [19] for the general case). For a simple example, when $d = 1$ then each value occurs only once (which was our previous case), while in the extreme case $d = \mathsf{IN}$, then all IN tuples in the database have the same value $t.B = b$.

Let $b_1, \ldots, b_{\mathsf{IN}/d}$ be the active domain of the attribute B. Fix a processor u, and denote by X_i the random variable that is 1 iff $h(b_i) = u$. As before, $\mathbb{E}[X_i] = 1/p$, the load is $L_u = d \cdot \sum_{i=1}^{\mathsf{IN}/d} X_i$ (because each value b_i contributes d tuples to u) and its expectation $\mathbb{E}[L_u] = \mathsf{IN}/p$. With this observation we can repeat the formal analysis above and derive the following bound:

$$\mathbb{P}[\max_u L_u \geq (1+\delta)\frac{\mathsf{IN}}{p}] \leq pe^{-\frac{\delta^2 \mathsf{IN}}{3pd}} \tag{3.2}$$

because the condition $L_u \geq (1+\delta)\frac{\mathsf{IN}}{p}$ is equivalent to $\sum_i X_i \geq (1+\delta)\frac{\mathsf{IN}}{pd}$. The only difference between (3.1) and (3.2) is that the factor IN/p in the exponent has decreased to $\mathsf{IN}/(pd)$. We still have $L \leq (1+\delta)\mathsf{IN}/p$ with a high probability, approaching 1 exponentially fast in $\mathsf{IN}/(pd)$. Therefore, we must have the degree $d \ll \frac{\delta^2 \mathsf{IN}}{3p}$; assuming δ is a constant, then $d = O(\frac{\mathsf{IN}}{p})$. If we allow d to approach IN/p, then it can be shown (as before) that the load increases by a factor $\log p$, in other words $L = O(\frac{\mathsf{IN}\log p}{p})$.

To summarize, for the purpose of computing a join $R(A,B) \bowtie S(B,C)$, a database instance is "without skew" if the maximum possi-

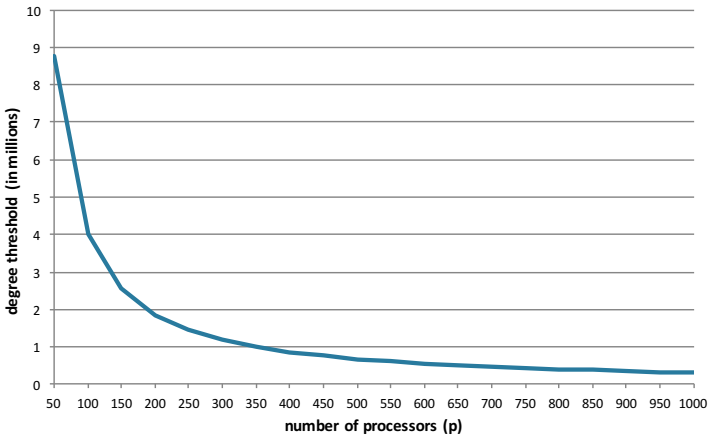


Figure 3.1: The graph shows the degree threshold for computing a hash-join for two tables totaling $\mathsf{IN} = 100$ billion tuples, to ensure that no processor has a load exceeding the expected load by more than 30%, with 95% probability, as a function of the number of processors p. For example, when $p = 100$, then the expected load is 1 billion tuples per processor: the graph shows that if all tuples have degree ≤ 4 million, then with probability > 0.95 no processor has a load larger than 1.3 billion tuples. On the other hand, if we increase the number of processors to $p = 1000$, then the data is skew-free only if all degrees are $\leq 300,000$.

ble degree of the join attribute is $O(\mathsf{IN}/p)$. The exact border between skewed and no-skewed databases is fuzzy, and depends on our choice of δ and of the probability: as an exercise, we invite the reader to derive an explicit formula for the largest degree d to guarantee the bound on the load with probability ≥ 0.95 (hint: $d = \Theta(\delta^2 \frac{\mathsf{IN}}{p \log p})$).

One important observation is that the degree threshold decreases as we increase the number of processors. If a database is skew-free for, say, $p = 100$ processors, it may become skewed when we increase to, say, $p = 1000$. Figure 3.1 shows the degree threshold for a database with 100 billion tuples that guarantees $\mathbb{P}[L < 1.3 \cdot \mathsf{IN}/p] \geq 0.95$. The figure shows that, for a fixed database instance, if we increase the number of processors that compute the join, then it becomes increasingly more likely that the database will be skewed, since the threshold decreases.

In summary, the parallel hash-join algorithms has a load $L = O(\mathsf{IN}/p)$ on skew-free data, and this implies that it has a linear speedup: as we increase p, the time to process the join should decreases at the

same rate. However, as p continues to increase, some data values in the database may become skewed, and then the performance of the join deteriorates dramatically. In that case, we need to handle the skewed elements separately, as we explain in the following sections.

In the rest of this survey, unless otherwise stated, a database is called "skew-free" if each value has a degree less than IN/p. This is an oversimplification for two reasons. First, the real threshold may be smaller, since it also depends on δ, the probability, and also carries an extra $\log p$ term. Second, for some forms of hash partitioning we will use strictly less than p bins: in that case the threshold may actually be larger.

3.1.3 Parallel Cartesian Product: the Most Skewed Join

We start our discussion of how to handle skewed data with an extreme case, when all tuples in R and all tuples in S have the same value b of the attribute B. In this case we cannot blindly apply the parallel hash-join algorithm, because it would send the entire data to only one processor, and the load becomes $L = \mathsf{IN}$. Instead, we observe that, in this case, the join degenerates to a Cartesian product, and design a specialized algorithm for it. Recall that the Cartesian product of two relations R', S' is $R' \times S'$ and consists of all pairs (a, c) with $a \in R', c \in S'$. It can be expressed in SQL as

```
SELECT *
FROM   R', S'
```

For example, the join of two relations $R(A, B) \bowtie S(B, C)$ where all tuples have the same value of the B attribute, is the cartesian product of $R' = \pi_A(R)$ and $S' = \pi_C(S)$.

We describe an algorithm called the Rectangle algorithm that computes the Cartesian product optimally. We will continue to assume that the input has size $\mathsf{IN} = |R'| + |S'|$. Notice that the size of the output is $\mathsf{OUT} = |R'| \cdot |S'|$.

Choose two numbers p_1, p_2, called *shares*, to be determined below, such that $p_1 \cdot p_2 = p$. We organize the p processors into a $p_1 \times p_2$

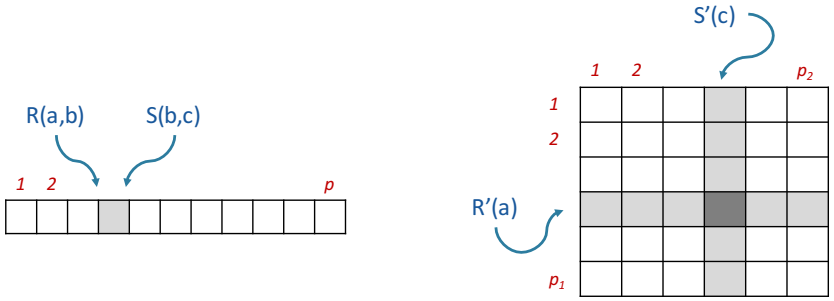


Figure 3.2: Different partitioning strategies: parallel hash-join (left) and Rectangle algorithm (right). When $p \cdot |R'| < |S'|$ then rectangle degenerates to a single row. However, the data is partitioned differently from the parallel hash-join: only S' is uniformly partitioned on the p processors, while the relation R' is broadcast to all processors.

rectangle; in other words, we identify each processor by two numbers $(u, v) \in [p_1] \times [p_2]$ and choose two random hash functions:

$$h_1 : \mathbf{dom}(A) \to [p_1]$$
$$h_2 : \mathbf{dom}(C) \to [p_2]$$

The Rectangle algorithm computes the Cartesian product in parallel as follows (see also Figure 3.2). Initially, both random functions h_1 and h_2 are known by all processors. During the communication step, for each tuple $R'(a)$ we compute $u = h_1(a)$, then send the tuple to all processors (u, v), for $v \in [p_2]$; similarly, for each tuple $S'(c)$ we compute $v = h_2(c)$ and send the tuple to all processors (u, v) for $u \in [p_1]$. After this communication round, the algorithm computes locally the Cartesian product of the tuples it has received. The correctness of the Rectangle algorithm follows from the fact that any pair (a, c) that will be in the output of the Cartesian product will "meet" at the processor with coordinates $(h_1(a), h_2(c))$.

The analysis of the algorithm is straightforward. For every fixed processor $(u, v) \in [p_1] \times [p_2]$, its load is $L_{(u,v)} = L^R_{(u,v)} + L^S_{(u,v)}$, where $L^R_{(u,v)}$, $L^S_{(u,v)}$ denote respectively the number of R-tuples and S-tuples (u, v) receives. Then $\mathbb{E}[L^R_{(u,v)}] = |R|/p_1$, since we hashed the values $R'(a)$ into p_1 buckets, and similarly $\mathbb{E}[L^S_{(u,v)}] = |S|/p_2$. Thus, the expected

value of the load is $\mathbb{E}[L_{(u,v)}] = |R|/p_1 + |S|/p_2$. The minimum value of this expression is $2\sqrt{|R| \cdot |S|/p} = 2\sqrt{\mathsf{OUT}/p}$, and is obtained when $|R|/p_1 = |S|/p_2$, or, equivalently, when the shares are $p_1 = \sqrt{p \cdot |R|/|S|}$ and $p_2 = \sqrt{p \cdot |S|/|R|}$. There is no skew in R' or in S' (they are sets), hence every value $a \in R'$ occurs only once, and $L_{(u,v)}$ is highly concentrated around its expectation. More precisely, we can apply a similar Chernoff bound as before and conclude:

$$\mathbb{P}\left[L \geq (1+\delta) \cdot 2\sqrt{\frac{\mathsf{OUT}}{p}}\right] \leq 2pe^{\frac{-2\delta^2\sqrt{\mathsf{OUT}}}{3\sqrt{p}}}$$

In short, the load L is close to $2\sqrt{\frac{\mathsf{OUT}}{p}}$ with high probability. Notice that the Rectangle algorithm adapts graciously to the relative sizes of R and S. When $|R| = |S| = \mathsf{IN}/2$, then $p_1 = p_2 = \sqrt{p}$ and the load is $2\sqrt{\mathsf{OUT}/p} = \mathsf{IN}/\sqrt{p}$ because $\mathsf{OUT} = (\mathsf{IN}/2)^2$. When $|R| < |S|$ then $p_1 < p_2$, allowing for more buckets to partition S. In an extreme case, $p \cdot |R| < |S|$, and we obtain $p_1 < 1$, $p_2 > p$. In this case we round up $p_1 = 1$, and round down $p_2 = p$. Thus, the Rectangle Algorithm broadcasts the small relation R to all processors, and hash partitions S uniformly across all processors.

Recall that the load of the parallel hash-join on skew-free data is IN/p, meaning that it has linear speedup. In contrast, the load of the Rectangle algorithm is $L = 2\sqrt{\mathsf{OUT}/p}$, which is $\Theta(\mathsf{IN}/\sqrt{p})$ when $|R| \approx |S|$. The speedup $\sim p^{1/2}$ is therefore sub-linear. A natural question is, can we compute the Cartesian product with a better load than the Rectangle algorithm, in a constant number of rounds? We will show next that the answer is no.

3.1.4 Lower Bounds for Cartesian Product.

Recall that in a stateful MPC model, the processors are allowed to keep their internal state between rounds. We will prove that any algorithm in the stateful MPC model that computes the Cartesian product $R \times S$ in r communication rounds with load L must satisfy $r \cdot L \geq 2\sqrt{\mathsf{OUT}/p}$; we will briefly discuss the stateless model below. In particular, when $r = 1$ then the load is $\geq 2\sqrt{\mathsf{OUT}/p}$, proving that the Rectangle algorithm is optimal. Hence, it is not possible to compute the Cartesian product with a linear speedup in a constant number of rounds.

Consider an arbitrary algorithm that uses p processors to compute the Cartesian product $R \times S$. Fix one processor $u \in [p]$, and define L_u^R, L_u^S the number of tuples that u receives from R and from S, during all rounds of communication. If the algorithm has r rounds of communication, then $L_u^R + L_u^S \leq r \cdot L$, because a stateful processor can accumulate the items received. Processor u can output at most $L_u^R \cdot L_u^S$ tuples from $R \times S$ and, since each output tuple must be present in one of the processors,

$$\sum_{u=1}^{p} L_u^R \cdot L_u^S \geq \mathsf{OUT}.$$

We now use the inequality $\alpha\beta \leq (\alpha+\beta)^2/4$ and derive:

$$\mathsf{OUT} \leq \sum_{u=1}^{p} L_u^R \cdot L_u^S \leq \sum_{u=1}^{p} \frac{(L_u^R + L_u^S)^2}{4} \leq \sum_{u=1}^{p} (r \cdot L)^2/4 = p \cdot (r \cdot L)^2/4$$

which implies $r \cdot L \geq 2\sqrt{\mathsf{OUT}/p}$. If the algorithm runs in a single round, then this load is exactly that of the Rectangle algorithm, proving that it is optimal. The central argument is based on the fact that the size of the output is large, and the processors must individually have enough local data to be able to produce this large output. We will see this argument appear many times throughout this article.

If the MPC model is stateless, then the processors cannot accumulate the input data items, since at each round their internal state is reset. Instead, the processors would produce some output tuples in each round. We invite the reader to prove in this case that $L \geq 2\sqrt{\mathsf{OUT}/(pr)}$.

3.1.5 Arbitrarily Skewed Case

To compute the natural join of two relations $R(A,B) \bowtie S(B,C)$ under arbitrarily skewed inputs we must combine the two algorithms described above: the Parallel Hash-join, and the Rectangle algorithm. Call a value b of the attribute B a *heavy hitter* if it occurs more than IN/p times in either R or S. Notice that there are $\leq p$ heavy hitter values because each contributes $> \mathsf{IN}/p$ tuples. Let $b_1, \ldots, b_k$ be all the heavy hitter values. A value that is not a heavy hitter will be called *light*. The

algorithm starts by computing all heavy hitter values and broadcasting them to all processors, which can be accomplished as follows. First each processor computes locally the set of values b whose local degree is $> \mathsf{IN}/p^2$, which is guaranteed to include all heavy hitters present locally. The processors broadcast these values (a total of $\leq p^2$ values), then each processor identifies the true heavy hitters by adding up their degrees at all processors, and retaining those with degree $> \mathsf{IN}/p$. We will assume that $p \leq \mathsf{IN}^{1/3}$, which implies that the processors can receive the p^2 values without exceeding the load, because $p^2 \leq \mathsf{IN}/p \leq L$.

Once the heavy hitter values are known by all processors, the algorithm proceeds as follows. First, it runs the parallel hash-join on the light hitters only: the load of this step is $L = O(\mathsf{IN}/p)$, since this fragment of the database is "without skew". Second, it computes the query on the heavy hitters, which is a union of Cartesian products $\bigcup_{i=1}^{k} \sigma_{R.B=b_i}(R) \times \sigma_{S.B=b_i}(S)$. Here, we will allocate a number of p_i exclusive processors to compute the Cartesian product for the value b_i, and compute all of the Cartesian products in parallel. The question is how many processors p_i to dedicate exclusively to the heavy hitter b_i; notice that we must have $\sum_i p_i = p$.

Let us denote the size of the output contributed by the heavy hitter b_i by $\mathsf{OUT}_i = |\sigma_{R.B=b_i}(R) \times \sigma_{S.B=b_i}(S)|$. Then we set $p_i \stackrel{\text{def}}{=} p\frac{\mathsf{OUT}_i}{\sum_j \mathsf{OUT}_j}$, and the load at each of the p_i processors that processes the heavy hitter b_i is $2\sqrt{\mathsf{OUT}_i/p_i} = 2\sqrt{\sum_j \mathsf{OUT}_j/p} \leq 2\sqrt{\mathsf{OUT}/p}$. The total load of the algorithm is the sum of the load due to the heavy hitters, and to the light hitters respectively, and thus is

$$L = O\left(\sqrt{\frac{\mathsf{OUT}}{p}} + \frac{\mathsf{IN}}{p}\right) \tag{3.3}$$

We invite the reader to prove that this load is optimal, by generalizing the argument used for the lower bound of the Cartesian product.

When $\mathsf{OUT} \leq \mathsf{IN}$ then the term IN/p dominates, hence $L = O(\mathsf{IN}/p)$ and the algorithm has linear speedup. But the output may be as large as IN^2; in that case $L = O(\mathsf{IN}/p^{1/2})$ and the algorithm has sub-linear speedup, similar to the Rectangle Algorithm for Cartesian product. In general, the speedup is $\sim 1/(\sqrt{\mathsf{OUT}/p}+\mathsf{IN}/p)$; Figure 3.3 illustrates the

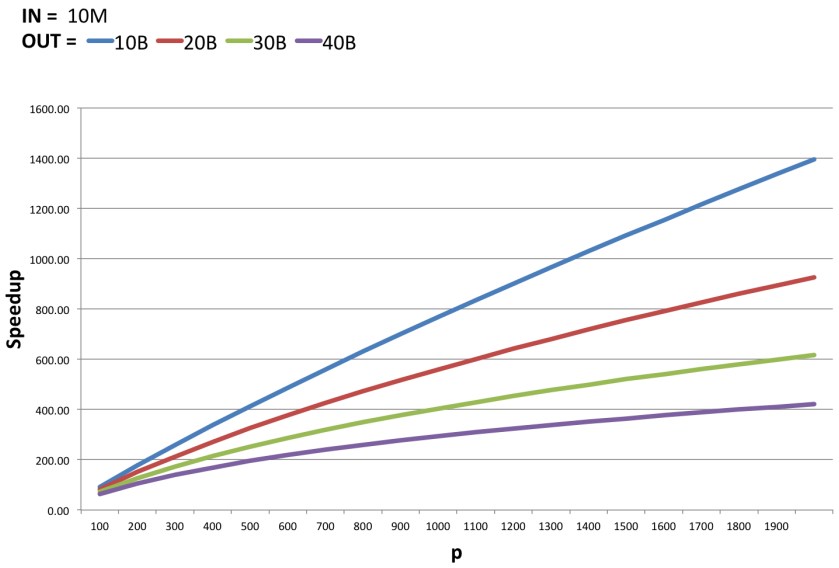


Figure 3.3: The speedup of the output-sensitive join algorithm.

speedup for an input of 10 Million tuples, and outputs varying from 10 to 40 Billion tuples. For small values of p the speedup is almost linear, but degrades as p increases, and degrades faster when the input is more skewed, causing the output to be larger.

We end with a brief discussion of the practical aspects of parallel join algorithms. If the main goal is to reduce number of communication rounds, then it is possible to run both steps above in parallel. Equation (3.3) essentially assumes this strategy, and computes the load as the sum of the loads corresponding to the heavy hitters, and to the light hitters. If the algorithm must ensure a strict bound on the load, then it may use more rounds. In particular, it may be necessary to use a heavy hitter threshold $\ll \mathsf{IN}/p$ in order to ensure that the load due to the light hitters is $O(\mathsf{IN}/p)$ (recall that when the degree is $\approx \mathsf{IN}/p$ the load is $O(\mathsf{IN}\log p/p)$), which may increase the number of heavy hitters values up to $O(p\log p)$. Since the second step of the algorithm can handle at most p heavy hitters at a time, we may need to increase the number of communication rounds to $r = O(\log p)$.

The algorithm discussed so far only handles a binary single join; we discuss optimal algorithms for multi-joins in Chapter 4.

3.2 Sort-Based Algorithms

Sort-based algorithms may require more rounds than hash-based algorithms, and for that reason they are not the first choice in practice, but their handling of heavy hitters is simpler, and leads to cleaner theoretical results. If the data can be arbitrary skewed, then sort-based algorithms are preferable to hash-based. The theoretical investigation of parallel sort-based algorithms for join queries in the MPC model has only recently been initiated [45].

3.2.1 Parallel Sort

Let $x_1, \ldots, x_N$ be a collection of N elements, which are initially uniformly partitioned on the p processors. Sorting this collection in parallel is relatively easy when $p \ll N$: simply find $p-1$ *splitters* $-\infty = y_0 < y_1 < \cdots < y_{p-1} < y_p = \infty$, then re-partition the data such that all data items in the same interval are sent to the same processor, then sort the items locally. The only challenge is to choose the splitters to ensure that the largest of the p intervals contains only $O(N/p)$ data items.

We describe a simple algorithm called *parallel sort by regular sample*, PSRS [75], that guarantees a load $L \leq 2N/p$. The algorithm assumes that there are no duplicate values in the set, i.e., $x_i \neq x_j$ for $i \neq j$. To ensure no duplicates, we can simply distinguish equal values by their processor number, and by their position within the processor. The PSRS algorithm has three steps:

- Each processor sorts its data locally, then computes a *regular sample* of its sorted data, which is defined as the set of $p-1$ splitters that partition uniformly the local (sorted) data into p equal sized intervals.
- Each processor broadcasts its regular sample to all other processors, then each processor sorts the entire regular sample, to obtain

a sorted sequence of length $p(p-1)$. Next, each processor computes the p splitters $y_1 < y_2 < \cdots < y_{p-1}$, defined as the elements on positions $p/2, p+p/2, 2p+p/2, 3p+p/2, \ldots, (p-2)p+p/2$ in the sorted sequence of length $p(p-1)$.

- Each processor partitions its local data into p intervals, using the splitters, and sends the data from each interval to the processor with the same number.
- After receiving the data in the interval defined by $y_{u-1} < y_u$, processor u sorts the data locally. The globally sorted sequence is the concatenation of the locally sorted sequences. As an optimization, if the last communication preserves order, then we can use the fact that the data was sorted locally before being sent: the processor can simply merge the p fragments it receives from all processors.

It can be shown that the maximum load of each processor is $L \leq 2N/p$, that is, at most twice the ideal load.

Notice that in the second step above each processor holds $p(p-1)$ data items. To ensure that this does not exceed the maximum load $2N/p$, we must have $p^2(p-1) \leq 2N$, or $p \lesssim N^{1/3}$. Modern parallel sorting implementations always assume that the data is much larger than the number of processors, and are based on some variant of PSRS; for example, they speed up the computation of the splitters by using sampling, and thus avoiding to sort the data locally; see Chapter 5. When p increases and approaches the size of the database, then theoretically optimal algorithms become more complex, as we shall discuss in Chapter 5.

3.2.2 Parallel Sort Join

To compute the natural join $R(A,B) \bowtie S(B,C)$ we proceed as follows (following [45]). First, we concatenate the two relations R, S to obtain a single set of IN elements. Next, we sort this set by the value of the B-attribute: for a given value $B = b$, we place (arbitrarily) the records in R before those in S. Consider now the sorted sequence. For any given value $B = b$, there are two cases:

- All tuples with value $B = b$ are at the same processor u. Then processor u can compute locally all tuples in the output that have this value of the attribute B, by joining the local R tuples with the local S tuples.

- The tuples with $B = b$ span two or more processors. We call such a value b a *crossing value*, and denote $b_1, \ldots b_\ell$ all crossing values. There are at most $p-1$ crossing values and, to compute their contribution to the join, we proceed exactly as in Section 3.1.5: allocate $p_i \stackrel{\text{def}}{=} p\frac{\mathsf{OUT}_i}{\sum_j \mathsf{OUT}_j}$ exclusive processors to the crossing value b, where $\mathsf{OUT}_i \stackrel{\text{def}}{=} |\sigma_{B=b_i}(R)| \cdot |\sigma_{B=b_i}(S)|$, and then use those processors to compute the Cartesian product $\sigma_{R.B=b_i}(R) \times \sigma_{S.B=b_i}(S)$.

It remains to show how to compute the crossing values b_i and the numbers $|\sigma_{B=b_i}(R)|$ and $|\sigma_{B=b_i}(S)|$ needed to determine the number of exclusive processors p_i. Every processor needs to know these. To compute them, each processors u examines its sorted sequence and determines the smallest value $b_u^{\min}$ and the largest value $b_u^{\max}$ (they may be the same value), then it broadcasts to all other processors the following information:

- $b_u^{\min}$ and the number of tuples t in R and in S with $t.B = b_u^{\min}$.

- $b_u^{\max}$ and the number of tuples t in R and in S with $t.B = b_u^{\max}$.

At this point each of the processors can determine all the required information. Specifically, a value b is a crossing iff there exists u such $b = b_{u-1}^{\max} = b_u^{\min}$, and for each crossing value it can determine $|\sigma_{B=b_i}(R)|$ and $|\sigma_{B=b_i}(S)|$ because it knows how many such such tuples are present at each processor.

We now compute the load of parallel sort join. The load for parallel sort is $\leq 2\mathsf{IN}/p$, assuming $p^3 \leq \mathsf{IN}$. The load needed to compute the crossing values is $O(p)$, which is much less than IN/p. Finally, the load needed to compute the join of the crossing values is bounded by $\sqrt{\mathsf{OUT}/p}$, using a similar argument to that in Section 3.1.5. Therefore, the combined load is:

$$L = O\left(\max\left(\sqrt{\frac{\mathsf{OUT}}{p}}, \frac{\mathsf{IN}}{p}\right)\right)$$

This expression is equal to the load of the hash-based join algorithm, Equation (3.3). We write it as a $\max(\cdots)$ rather than a sum, to emphasize that the two loads correspond to two different rounds; in contrast, in hash-join the light and heavy hitters could be computed in parallel, hence the total load is the sum.

3.3 Binary Join Algorithms in Existing Systems

Several of the algorithms we discussed in this chapter or their variants are actively deployed in existing systems to join two relations. We review each one next:

Hash-based Joins: The parallel hash-join algorithm from Section 3.1.1 and its variants are one of the most commonly used algorithms in practice. Recall that this algorithm hashes the tuples across the processors on the join attributes and then performs a local join algorithm at each processor. Most systems we are are aware of, such as Hive [83], Impala [56], Myria [44], PigLatin [69], or SparkSQL [77], use this algorithm, especially when the two relations are of similar sizes. There are typically two ways the tuples are hashed (or partitioned) to processors:

- *Random hashing:* Using a simple hash function that sends values randomly to the processors. This is the algorithm we discussed in Section 3.1.1.
- *Range hashing:* When the values in the join attributes can be ordered, e.g., they are numeric values, some systems divide the values into predefined buckets of ranges and assign each range to a processor. Each tuple is then sent to the processor that is responsible for its range.

Broadcast Join: When one of the input relations is significantly smaller than the other one, several systems, such as Hive, Impala, and

Spark SQL use a *Broadcast* join, where the larger relation is not shuffled and the smaller relation is broadcast to each processor. This method avoids shuffling the larger relation, which can save significant communication costs. Recall from Section 3.1.3 that the Rectangle algorithm does the same optimization when computing a Cartesian product.

Sort-Merge Join: Spark SQL supports an algorithm similar to the Parallel Sort Join algorithm we discussed in Section 3.2.2. When the join attribute is sortable, this algorithm first sorts the tuples of these two relations using a parallel sorting algorithm, ensuring that partitions of both relations with the same key range are in the same processor after sorting. Each processor then performs a local join by merging the two sorted list of tuples. We review parallel sorting algorithms used by existing systems in Chapter 5.

We note that when using both Hash-based joins and Broadcast joins, once the data is shuffled, different systems use different serial join algorithms, such as a hash join, nested-loop join, or Bloom filter join, to compute the outputs locally at each processor [44, 56, 69, 77, 83].

Unfortunately, there is no work we are aware of that systematically compares the performance of these different algorithms. An experimental demonstration of which algorithms work well in which settings would be very useful.

4

Multiway Joins

In this chapter, we discuss algorithms and lower bounds for computing multiway joins. More precisely, we focus on natural join queries, expressed as follows:

$$q(\mathbf{x}) = R_1(\mathbf{x}_1) \bowtie \cdots \bowtie R_\ell(\mathbf{x}_\ell) \tag{4.1}$$

where $\mathbf{x}_j$, $j = 1, \ldots, \ell$, are vectors of attributes/variables, also denoted by $\text{vars}(R_j)$, while $\mathbf{x}$ is the set of all variables, also denoted by $\text{vars}(q)$. We denote by $k \stackrel{\text{def}}{=} |\text{vars}(q)|$. Recall that IN denotes the total size of the input, and OUT the size of the output. We will also use N_j to denote the number of tuples in relation R_j. Examples of natural joins we will use throughout this chapter are the join query introduced in Chapter 3, and the triangle query, as shown below:

$$J(x, y, z) = R(x, y) \bowtie S(y, z)$$
$$\Delta(x, y, z) = R(x, y) \bowtie S(y, z) \bowtie T(z, x)$$

In this chapter we will follow the convention for conjunctive queries and use lower case letters $x, y, \ldots$ instead of attribute names $A, B, \ldots$; we refer the reader to [60] for a definition of conjunctive queries. In fact, our discussion of natural join queries in this chapter applies immediately to all *full* conjunctive queries, which are conjunctive queries

where the head contains all variables (i.e. there are no projections). The only difference between full conjunctive queries and natural join queries is that the latter do not allow selection predicates, like $x =$ "abc" or $y > 7$, repeated variables in an atom, like $R(x, x, y)$, and self-joins, like $R(x, y) \bowtie R(y, z)$. All algorithms described in this chapter for natural join queries extend easily to full conjunctive queries.

We classify the algorithms in this chapter based on two characteristics: (i) the number of rounds, and (ii) the optimality guarantees.

Number of rounds. We consider algorithms that operate using only a *single-round*, versus algorithms that need *multiple rounds*. In all cases considered, the number of rounds depends only on the query, and is independent of the input size.

Optimality guarantees. We always assume that the cardinality N_j of each relation R_j is known to the algorithm, and the input instance is guaranteed to satisfy the cardinality constraints $|R_j| \leq N_j$. In a *worst-case optimal* algorithm the load is expressed as a function of only the input size and the number of processors, $L(\mathsf{IN}, p)$ or, at a finer granularity, $L(N_1, \ldots, N_\ell, p)$. Thus, the algorithm guarantees this load for any input satisfying the cardinality constraints. An *output sensitive* algorithm is one where the load is expressed both in terms of the input and the output, $L(\mathsf{IN}, \mathsf{OUT}, p)$. For example, a 1-round output sensitive algorithm for single join has load $L = O(\mathsf{IN}/p + \sqrt{\mathsf{OUT}/p})$, while the worst-case guarantee increases to $L = O(\mathsf{IN}/\sqrt{p})$, because the output can be as large as $\mathsf{OUT} = \mathsf{IN}^2$. Both worst-case optimal and output sensitive algorithms attain their worst load when the data is skewed. A simpler algorithm can improve the load by further restricting to input data that is *skew-free*, in other words satisfy further constraints on the maximum frequency (or degree) that certain values appear in the database. The results discussed in this chapter are summarized in Table 4.1.

Table 4.1: Summary of the load for computing a natural join query. τ^* is the fractional edge packing number, ρ^* the fractional edge covering number, and ψ^* the quasi packing number; τ^*, ρ^* are incomparable, while $\psi^* \geq \max(\tau^*, \rho^*)$.

	One Round					Multiple Rounds	
	No Skew		Worst case		Output sensitive	Worst case	Output sensitive
	Uniform	Non-uniform	Uniform	Non-uniform			
Lower bound	$\frac{N}{p^{\frac{1}{\tau^*}}}$	$\left(\frac{\prod N_j^{u_j}}{p}\right)^{\frac{1}{\sum u_j}}$	$\frac{N}{p^{\frac{1}{\psi^*}}}$	max no-skew of residual queries	$\frac{\mathsf{IN}}{p} + \sqrt{\frac{\mathsf{OUT}}{p}}$ (join only)	$\frac{\mathsf{IN}}{r \cdot p^{\frac{1}{\rho^*}}}$	$\frac{1}{r} \cdot \left(\frac{\mathsf{OUT}}{p}\right)^{\frac{1}{\rho^*}}$
Upper bound	same	max of above for $\mathbf{u} \in \mathsf{pk}(q)$	same	same	same	same (binary tables only)	GYM

Section 4.1 reviews lower bounds and algorithms for one-round computations. Section 4.2 reviews lower bounds and algorithms that run for multiple rounds of computation. Finally, we briefly review multiway join algorithms that are used in some of the existing systems in Section 4.3. All algorithms discussed in Sections 4.1 and 4.2 are hash-based algorithms. Sorting-based algorithms for computing multiway joins are quite promising because they do not require a separate treatment of skewed and skew-free data, and their study has only recently begun [45].

4.1 Single Round

We start our discussion with algorithms that operate using a single communication round. The core algorithmic tool is the Hypercube algorithm, which is a generalization of both Parallel Hash-Join and the Rectangle algorithm in Chapter 3, and is optimal on skew-free databases. We examine the speedup of the Hypercube algorithm, showing how it differs from an ideal linear speedup. We then discuss a worst-case optimal algorithm, SkewHC, obtained by running different instances of the Hypercube algorithm on various skewed subsets of the data. We end with a brief discussion of output sensitive algorithms, which are less well understood for a single round.

4.1.1 The Hypercube Algorithm

The *Hypercube algorithm (HC)* is a hash-based algorithm originally introduced under the name *Shares algorithm* by Afrati and Ullman [4] in the context of MapReduce. It is similar to an algorithm by [81] to count triangles, and uses ideas that can be traced back to [38] for parallel processing of Datalog programs.

The HC algorithm is parametrized by a vector of *shares*: each variable x_i in the query, where $i = 1, \ldots, k$, is assigned a share p_i, and the shares are chosen such that their product is equal to the number of processors[1], $\prod_{i=1}^{k} p_i = p$. The name of the algorithm results from the fact that each server is mapped to a distinct point in the k-dimensional

[1] More generally, we can relax the restriction on the product to $\prod_i p_i \leq p$.

hypercube $[p_1] \times \cdots \times [p_k]$. The description of the HC algorithm is as follows:

Algorithm 1 HYPERCUBE ALGORITHM

Input: query $q(x_1, \ldots, x_k) = R_1(\mathbf{x}_1) \bowtie \cdots \bowtie R_\ell(\mathbf{x}_\ell)$,
shares $\mathbf{p} = (p_1, \ldots, p_k)$.

1: Assign each processor in $[p]$ to a distinct point in $[p_1] \times \cdots \times [p_k]$
2: Choose k independent hash functions $h_i : \mathbf{dom}(x_i) \to [p_i]$
3: **Communication**: send each tuple $R_j(t)$ to the subcube

$$\{\mathbf{u} \in [p_1] \times \cdots \times [p_k] \mid \bigwedge_{x_i \in \mathrm{vars}(R_j)} h_i(t.x_i) = u_i\}$$

4: **Computation**: each processor computes q on its local instance

During the communication phase, each tuple t in R_j is sent to several processors in the space $[p_1] \times \cdots \times [p_k]$. The algorithm applies a hash function to each attribute of t, and obtains some of the coordinates of the destination processors. For all missing coordinates (corresponding to variables not in R_j), it simply broadcasts that tuple along those dimensions. The correctness of the HC algorithm follows from the observation that, for every potential output tuple $(a_1, \ldots, a_k)$, the processor $(h_1(a_1), \ldots, h_k(a_k))$ contains all the necessary information to decide whether $(a_1, \ldots, a_k)$ belongs in the answer or not. Observe also that each choice of shares $p_1, \ldots, p_k$ gives a different parametrization of the HC algorithm. We will discuss how to optimally choose the right shares in the next section. For now, we give two examples of how we can use the HC algorithm to compute join queries in a single round.

Example 4.1 (HC for Triangles)**.** We illustrate how to compute the triangle query $\Delta(x, y, z) = R(x, y) \bowtie S(y, z) \bowtie T(z, x)$ using the HC algorithm. Choose three shares p_x, p_y, p_z such that their product equals p. For example, we could choose $p_x = p_y = p_z = p^{1/3}$, but the optimal choice may be different, and will be discussed in Section 4.1.2. Each of the p servers is uniquely identified by a triple (u_x, u_y, u_z), where $u_x \in [p_x]$, $u_y \in [p_y]$, $u_z \in [p_z]$. The algorithm makes use of three hash

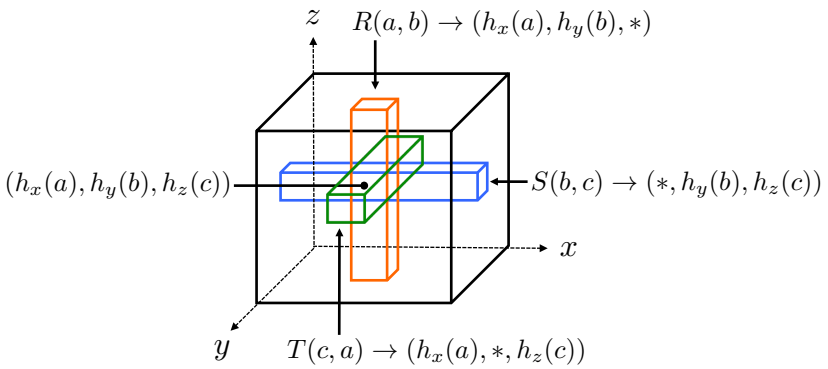


Figure 4.1: Depiction of the HC algorithm for the triangle query. The p processors are organized into a $p_x \times p_y \times p_z$ cube.

functions h_x, h_y, h_z, which map values from the domain to $[p_x]$, $[p_y]$, $[p_z]$ respectively.

During the first communication round, each tuple $R(a, b)$ is sent to all processors with index $(h_x(a), h_y(b), *)$, where the last entry takes all values in $[p_z]$: notice that each tuple is replicated p_z times. Similarly, a tuple $S(b, c)$ is sent to all processors $(*, h_y(b), h_z(c))$, and a tuple $T(c, a)$ is sent to all processors $(h_x(a), *, h_z(c))$. After round 1, any three tuples $R(a, b)$, $S(b, c)$, $T(c, a))$ that contribute to the output tuple $\Delta(a, b, c)$ will be seen by the processor $(h_x(a), h_y(b), h_z(c))$: any processor that detects three matching tuples outputs them. The communication pattern is given pictorially in Figure 4.1. Note that we use different hash functions for the three variables. This is clearly necessary when the shares p_x, p_y, p_z are different, but even when $p_x = p_y = p_z = p^{1/3}$ we still use independent hash functions in order to uniformly partition relations whose two attributes are correlated; for example if we used the same $h_x = h_y$ and the input relation R is such that $t.x = t.y$ for every tuple $t \in R$, then we use only a small subset of the hypercube to partition R.

Example 4.2 (HC for Joins)**.** We show that the HC algorithm subsumes both the Parallel Hash-Join and the Rectangle algorithm discussed in Chapter 3. Recall the two-way join query $J(x, y, z) = R(x, y) \bowtie S(y, z)$.

The Parallel Hash-Join algorithm corresponds to a choice of shares where $p_x = p_z = 1$, and $p_y = p$, in other words only dimension y is used to partition the two relations, while the Rectangle algorithm corresponds to a choice of shares where $p_x \cdot p_z = p$, and $p_y = 1$.

4.1.2 Choosing the Shares

We discuss here how to choose optimally the shares $p_1, \ldots, p_k$, to minimize the load of the algorithm on skew-free databases. We start by giving the precise definition of skew needed in this context.

Let R be a relation, $A \subseteq \text{vars}(R)$ be its set of attributes, and an A-tuple t be a tuple over the attributes A. We define t's *degree*, denoted $\mathsf{deg}_{R[A]}(t)$, or simply $\mathsf{deg}_A(t)$ when R is clear from the context, to be the number of tuples t' in R such that $t = t'.A$, where $t'.A$ denotes the tuple $(t'.x_i)_{x_i \in A}$. For example, if $R(x, y, z)$ is a ternary relation, $A = (x, y)$, and $t = (a, b)$, then $\mathsf{deg}_{x,y}(a, b)$ is the number of tuples $t' \in R$ such that $t'.x = a$ and $t'.y = b$, while, for $A = (x)$, $\mathsf{deg}_x(a)$ is the number of tuples such that $t'.x = a$; clearly $\mathsf{deg}_{x,y}(a, b) \leq \mathsf{deg}_x(a)$.

Definition 4.1 (Skew-Free)**.** Let $\mathbf{p} = (p_1, \ldots, p_k)$ be a vector of shares. We say that the input is *skew-free* w.r.t. $\mathbf{p}$ if for every relation R_j, for every $A \subseteq \text{vars}(R_j)$, and for every tuple $t \in \mathbf{dom}(A)$:

$$\mathsf{deg}_{R_j[A]}(t) \leq \frac{N_j}{\prod_{i:x_i \in A} p_i}$$

Example 4.3. We explain the rationale behind the definition. Referring to the triangle query discussed in Example 4.1, assume that we have chosen equal shares $p_x = p_y = p_z = p^{1/3}$. We focus on the relation R, whose cardinality is N_R, and examine the load per processor due to tuples from R. Fix one processor $(u, v, 1) \in [p_x] \times [p_y] \times [p_z]$. Since R is broadcast along the 3rd dimension, we simply set it to 1, since processors with all other values in this position will behave the same. A tuple $t \in R$ is sent to this processor iff $h_x(t.x) = u$ and $h_y(t.y) = v$, and this happens with probability $1/p^{2/3}$ over the random choices of the hash functions h_x, h_y, which we assume are chosen independently. Thus, the *expected* number of tuples sent to any fixed processor is $N_R/p^{2/3}$. Now we repeat the argument used in Section 3.1.2 and show that if the

degree of x or y is too large, then at least one processor will exceed this load. Suppose some value a occurs more than $N_R/p^{1/3}$ times in the column x, and let $u \stackrel{\text{def}}{=} h_x(a)$. Then more than $N_R/p^{1/3}$ tuples in R will be distributed to only $p^{1/3}$ processors, namely the processors $\{(u, v, 1) \mid v \in [p^{1/3}]\}$, and therefore at least one processor will have load $> N_R/p^{2/3}$. In other words, a necessary condition for *all* processors to have a load $O(N_R/p^{2/3})$ is that $\mathsf{deg}_x(a) = O(N_R/p^{1/3})$ for all values a, and similarly $\mathsf{deg}_y(b) = O(N_R/p^{1/3})$ for all values b. By Def. 4.1 the database is called skew-free, if both $\mathsf{deg}_x(a)$ and $\mathsf{deg}_y(b)$ are $\leq N_R/p^{1/3}$ forall a and b.

Generalizing the observation in the example, we note that, if the shares of the Hypercube algorithm are $p_1, \ldots, p_k$, then the expected number of tuples from R_j received by any fixed server is $N_j / \prod_{i:x_i \in \text{vars}(R_j)} p_i$. Indeed, the HC algorithm distributes R_j only to the subcube with dimensions corresponding to the variables in R_j, in other words it distributes R_j to $\prod_{i:x_i \in \text{vars}(R_j)} p_i$ servers. Moreover, if the database is skewed, then, for any choice of the hash functions, at least one server will exceed this load. The next theorem proves some form of a converse:

Theorem 4.1. Let $\mathbf{p} = (p_1, \ldots, p_k)$ be a shares vector used by the Hypercube algorithm. If the input is skew-free w.r.t $\mathbf{p}$, then, for each relation R_j, the maximum, over all servers, of the number of R_j-tuples sent to that server is with high probability

$$\tilde{O}\left(\frac{N_j}{\prod_{i:x_i \in R_j} p_i}\right)$$

Therefore, the load of the Hypercube algorithm is with high probability

$$L = \tilde{O}\left(\max_{j=1}^{\ell} \frac{N_j}{\prod_{i:x_i \in R_j} p_i}\right) \tag{4.2}$$

The $\tilde{O}$ notation hides a polylogarithmic dependence on the number of processors p. For example, when applied to the triangle query in Example 4.3, assuming shares $p_x = p_y = p_z = p^{1/3}$, the theorem implies that the maximum load of all processors is $L =$

$\tilde{O}(\max\{N_R, N_S, N_T\}/p^{2/3})$. Here, the reader will notice that we used max instead of a sum. The exact formula for load L in Equation (4.2) should use $\sum_{j=1}^{\ell}$ instead of $\max_{j=1}^{\ell}$, but we prefer to use max because this simplifies the task of choosing optimal shares, as we explain below; since ℓ is a constant (independent on the input data and the number of processors), the quantities $\sum_{j=1}^{\ell}$ and $\max_{j=1}^{\ell}$ differ by at most a constant factor.

Equation (4.2) gives us an explicit formula for the load of the HC algorithms, in terms of the shares $p_1, \ldots, p_k$. Using this formula, we can compute the optimal values of the shares to minimize the load. We first express the shares as $p_i = p^{e_i}$, where $e_i \in [0, 1]$ is called the *share exponent* for x_i, and denote by $\lambda = \log_p L$. Then, the optimal shares are obtained by optimizing the following linear program (LP):

$$\begin{aligned}
\textbf{minimize} \quad & \lambda \\
\textbf{subject to} \quad & \sum_{i\in[k]} -e_i \geq -1 \\
& \forall j \in [\ell] : \sum_{x_i \in R_j} e_i + \lambda \geq \log_p(N_j) \\
& \forall i \in [k] : e_i \geq 0, \quad \lambda \geq 0
\end{aligned} \tag{4.3}$$

The first inequality corresponds to $\prod_{i=1}^{k} p_i \leq p$. The second inequality states that $L \geq N_j / \prod_{i:x_i \in R_j} p_i$ for all j; since the objective is to minimize L, this is equivalent to setting $L = \max_j N_j / \prod_{i:x_i \in R_j} p_i$.

Example 4.4. Continuing Example 4.1, the LP for the triangle query is the following:

$$\begin{aligned}
\textbf{minimize} \quad & \lambda \\
\textbf{subject to} \quad & e_x + e_y + e_z \leq 1 \\
& e_x + e_y + \lambda \geq \log_p(N_R) \\
& e_y + e_z + \lambda \geq \log_p(N_S) \\
& e_x + e_z + \lambda \geq \log_p(N_T) \\
& e_x, e_y, e_z, \lambda \geq 0
\end{aligned}$$

4.1.3 The Load for Uniform Databases

The linear program Equation (4.3) does not give us an explicit formula either for the shares p_i or for the load L; instead, these are described as a solution to a linear optimization problem. To answer qualitative questions about the load of the Hypercube algorithm, we describe next an explicit formula for the load L, then we use it to analyze the speedup of the Hypercube algorithm. We start by discussing uniform databases, in other words $N_1 = \cdots = N_\ell$; in this case we can give an explicit formula for both the shares and the load; we discuss non-uniform databases in the next section.

Vertex cover Each multiway natural join query q given by Equation (4.1) is associated naturally to a hypergraph $\mathcal{H} = (V, E)$, whose nodes are the variables $x_1, \ldots, x_k$ of q and whose hyperedges are the relations $R_1, \ldots, R_\ell$ of q. When we define below the notions of fractional edge packing and fractional vertex cover, we will define them in the context of a query, although their standard definition is for a hypergraph.

A *fractional vertex cover* is a set of non-negative weights $\mathbf{v} = (v_1, \ldots, v_k)$ associated to the variables (nodes) $(x_1, \ldots, x_k)$, such that:

$$\text{for every relation } R_j: \qquad \sum_{i: x_i \in vars(R_j)} v_i \geq 1$$

The quantity $\min_{\mathbf{v}} \sum_{i=1,k} v_i$, where $\mathbf{v}$ ranges over fractional vertex covers, is denoted by $\tau^*(q)$ or simply τ^* when q is understood from the context; $\mathbf{v}^*$ denotes some optimal fractional vertex cover, in other words, $\sum_i v_i^* = \tau^*$. We note that $\mathbf{v}^*$ is not necessarily unique.

In the special case when $N_1 = N_2 = \cdots = N_\ell = N$ the optimal shares and optimal load of the HC algorithms can be expressed in terms of $\mathbf{v}^*$ and τ^* as follows:

$$\forall i \in [k]: \quad p_i = p^{\frac{v_i^*}{\tau^*}} \qquad\qquad L = \frac{N}{p^{\frac{1}{\tau^*}}} \tag{4.4}$$

Indeed, the reader may check that, if $\mathbf{v}^*$ is an optimal fractional vertex cover, then $e_i^* \stackrel{\text{def}}{=} v_i^*/\tau^*$, for all $i \in [k]$, $\lambda^* \stackrel{\text{def}}{=} \log N - 1/\tau^*$ is an optimal solution to the linear program of Equation (4.3).

Using the explicit formula (4.4) for L we derive the speedup of the Hypercube algorithm, as $\sim p^{1/\tau^*}$; this is sublinear whenever $\tau^* > 1$. The only queries that have $\tau^* = 1$, and, hence, have a linear speedup, are those that have some variable x_i present in all relations. In other words, the only queries that can be computed in one round with linear speedup (over skew-free databases) are multi-way star joins.

4.1.4 The Load for Non-Uniform Databases

When the cardinalities of the input relations are not equal, then the linear program Equation (4.3) no longer has a simple closed form solution. We can still find a closed form, but only for the load L, which is sufficient to study the speedup of the Hypercube algorithm. As we shall see later, the algorithm takes advantage of the unequal sizes of the input relations, e.g. by allocating more shares to the larger relations, and can achieve better speedup than $\sim p^{1/\tau^*}$. We start by defining fractional edge packings.

Edge packing A *fractional edge packing* is a set of non-negative weights $\mathbf{u} = (u_1, \ldots, u_\ell)$ associated to the relations (hyperedges) $(R_1, \ldots, R_\ell)$, such that:

$$\text{for every variable } x_i \in \text{vars}(q): \qquad \sum_{j:x_i \in R_j} u_j \leq 1$$

If all weights u_j are integers, then $\mathbf{u}$ is called an *integral edge packing*, or edge packing for short. Equivalently, an edge packing is defined as a subset of the relations $R_{j_1}, R_{j_2}, \ldots$ such that no two relations share any common variable.

By the strong duality theorem, the maximum value of a fractional edge packing is equal to the minimum value of a fractional vertex cover, which we denoted τ^*. Formally, $\max_{\mathbf{u}} \sum_{j=1,\ell} u_j = \min_{\mathbf{v}} \sum_{i=1,k} v_i = \tau^*(q)$, where $\mathbf{u}$ ranges over fractional edge packings, and $\mathbf{v}$ ranges over fractional vertex covers. For that reason, we call $\tau^*(q)$ the *fractional edge packing number* of the query q.

The connection to the optimal load of the Hypercube algorithm given by Equation (4.3) is the following.

Theorem 4.2 (Upper Bound [18]). Let $\lambda^*, e_1^*, \ldots, e_k^*$ be an optimal solution to the linear program (4.3), and denote by $L \stackrel{\text{def}}{=} p^{\lambda^*}$. Then,

$$L = \max_{\mathbf{u}} \left(\frac{\prod_{j=1}^{\ell} N_j^{u_j}}{p} \right)^{1/\sum_j u_j} \tag{4.5}$$

where $\mathbf{u}$ ranges over the fractional edge packings of q.

In other words, although we don't have a closed formula for the optimal share exponents $e_1^*, \ldots, e_k^*$, Equation (4.5) gives us an explicit formula for the optimal load; we will given an example and more intuition in Section 4.1.6.

If the database is uniform, meaning $N_1 = \cdots = N_\ell = N$, then this closed form is precisely the expression for the load L in Equation (4.4). Indeed, in this case $\left((\prod_{j=1}^{\ell} N_j^{u_j})/p\right)^{1/\sum_j u_j} = N/p^{1/\sum_j u_j}$ and this quantity is maximized precisely when $\mathbf{u}$ is an optimal fractional edge packing of the query; in that case $\sum_j u_j = \tau^*$ and therefore $L = \max_{\mathbf{u}} N/p^{1/\sum_j u_j} = N/p^{1/\tau^*}$, and we recover the expression (4.4).

While the linear program (4.3) is associated to an algorithm and thus is an upper bound for computing q, the formula (4.5) is not associated to an algorithm but instead is a lower bound for computing q. This explains why we need to maximize the expression: since each $\mathbf{u}$ is a lower bound, so is their maximum. The proof of Theorem 4.2 consists of showing that it is equivalent to the objective function of the dual of the linear program (4.3), which also justifies using max instead of min.

4.1.5 Lower Bound for 1-Round Algorithms on Skew-Free Data

We show here that the expression under (4.5) represents a lower bound on the load L of *any* one-round algorithm. The lower bound is quite strong, in the sense that it applies to any algorithm, even if we require the algorithm to compute q correctly only on some simple database instances called matching databases.

A *matching database* is a database instance where every attribute of every relation is a key. It follows that every degree is 1. By proving a lower bound on the load L on any algorithm that computes q correctly on all matching databases, we also show the same lower bound on

any algorithm that computes q correctly on all skew-free databases, no matter what formal definition of skew-free we adopt (since it will always include matching databases).

We start by giving a very simple intuition on the lower bound. Consider the s-way Cartesian product query, $R_1 \times R_2 \times \cdots \times R_s$, where the relations have sizes $N_1, \ldots, N_s$. By repeating the argument in Section 3.1.4 one can prove that any 1-round algorithm requires a load:

$$L \geq s \cdot \left(\frac{N_1 \cdots N_s}{p}\right)^{\frac{1}{s}}$$

Consider now an arbitrary query q (no longer a Cartesian product), and let $R_{j_1}, R_{j_2}, \ldots, R_{j_s}$ be an (integral) edge packing. More precisely, let these relations be the relations for which the integral edge packing had a value of 1; notice that no two relations in this set share any variables (attributes). Then we claim that any algorithm that computes the query in one round must also compute the s-way Cartesian product; in other words, for any set of tuples $t_1 \in R_{j_1}, \ldots, t_s \in R_{j_s}$, the algorithm must send all of these tuples to some common processor. The claim implies that the load of the algorithm must satisfy:

$$L \geq \max s \cdot \left(\frac{N_{j_1} \cdots N_{j_s}}{p}\right)^{\frac{1}{s}} \tag{4.6}$$

where the maximum is taken over all edge packings. To prove the claim, suppose otherwise, that no processor receives all s tuples $t_1, \ldots, t_s$. Then we modify the other relations R_j (not in the packing) to ensure that $(t_1, \ldots, t_s)$ is part of an output to the query q. This is possible because $R_{j_1}, \ldots, R_{j_s}$ is a matching, i.e. these tuples do not share any attribute, and we can ensure they are part of the output of the query by inserting appropriate tuples in the other relations. Changing the other relations R_j will not affect where the algorithm sends the tuples in the relations $R_{j_1}, \ldots, R_{j_s}$, since we may assume that the input is initially distributed on the p processors such that every processor holds tuples from a single relation, which implies that the action of the processors holding tuples from $R_{j_1}, \ldots, R_{j_s}$ will not be affected if we modify the other input relations. Therefore, if the tuples $t_1, \ldots, t_s$ don't meet at some processor, the algorithm fails to answer the query correctly on the modified input. This proves the lower bound (4.6).

To illustrate the argument we used in the proof, consider the query $R(x,y) \bowtie S(y,z) \bowtie T(z,u) \bowtie K(u,x)$, and the edge packing R, T. Any two tuples $t_1 = (a,b) \in R$, $t_2 = (c,d) \in T$ must be received by some processor. Indeed, suppose not. Then we modify S, K so that they contain tuples $(b,c) \in S$ and $(d,a) \in K$ respectively. This implies that (a,b,c,d) is an answer to q, but the algorithm will fail to discover this tuple because no processor sees both t_1 and t_2. Notice that our argument works even if the database instance is a matching database.

It turns out that the lower bound Equation (4.6) continues to hold if we replace the integral edge packing with a fractional edge packing.

Theorem 4.3 (Lower Bound [18]). Given a natural join query q, any (randomized) algorithm that computes q correctly on all matching databases, in a single round with p processors must have load

$$L = \Omega\left(\max_{\mathbf{u}}\left(\frac{\prod_{j=1}^{\ell} N_j^{u_j}}{p}\right)^{1/\sum_j u_j}\right) \tag{4.7}$$

where $\mathbf{u}$ ranges over the fractional edge packings of q.

This expression coincides with the load of the Hypercube algorithms, (4.5), so the algorithm is optimal on skew-free databases, up to the polylogarithmic factor in p hidden by the $\tilde{O}$ notation.

4.1.6 The Speedup of the Hypercube Algorithm

We discuss here the speedup of the Hypercube algorithm. We have seen that on uniform databases, i.e. $N_1 = N_2 = \cdots = N_\ell = N$, the speedup is $\sim p^{1/\tau^*}$. We show here that, on non-uniform databases, the HC algorithm adapts graciously and achieves increased speedup, essentially by allocating more shares on the larger relations.

We cannot use Equation (4.5) directly to analyze the load of the Hypercube algorithm, because $\mathbf{u}$ ranges over the *fractional edge packing polytope*, which is an infinite set. Instead, it turns out that we can restrict $\mathbf{u}$ to range only over the vertices of the fractional edge packing polytope. Intuitively, a vertex of a polytope P is a "corner" of the set P. Formally, $\mathbf{u} \in P$ is called a *vertex* if it is not the convex combination

of two other distinct points $\mathbf{u}_1, \mathbf{u}_2 \in P$, i.e. it is not of the form $(1-t)\cdot\mathbf{u}_1 + t\cdot\mathbf{u}_2$ for $t \in (0,1)$. We will denote by $\mathsf{pk}(q)$ the set of vertices of the fractional edge packings polytope for q. This set is finite, and its cardinality can be shown to be $\leq \binom{k+\ell}{\ell}$, where k, ℓ represent the number of variables and number of relations in q. It can be shown that the fractional edge packing $\mathbf{u}$ in Equation (4.5) can be restricted to range only over $\mathsf{pk}(q)$. The proof is not immediately obvious, because the function Equation (4.5) is not convex in $\mathbf{u}$, and can be found in [19].

Consider a database instance where the input relations have sizes $N_1, \ldots, N_\ell$. As we increase the number of processors p, the fractional edge packing $\mathbf{u}$ that maximizes Equation (4.5) moves between the vertices $\mathsf{pk}(q)$ of the edge packing polytope. Since there are only finitely many vertices, they partition the domain of p into a finite number of intervals, such that within each interval the load $L(p)$ is given by Equation (4.5) for some fixed vertex $\mathbf{u} \in \mathsf{pk}(q)$. In that interval, the speedup is $\sim p^{1/\sum_j u_j}$. It can be shown that as p increases and moves from one interval to the next, the quantity $\sum_j u_j$ can only increase, i.e. the vertex $\mathbf{u}$ moves to one with a larger value, eventually reaching τ^* when p is large enough. Thus, the speedup is initially high, perhaps even linear, but as we add more processors the speedup decreases, eventually reaching $\sim p^{1/\tau^*}$, which is the same as the speedup on uniform databases. We illustrate this on an example.

Example 4.5. Continuing the example of the triangle query, $\Delta(x,y,z) = R(x,y) \bowtie S(y,z) \bowtie T(z,x)$, the polytope $\mathsf{pk}(\Delta)$ has five vertices, each giving a different value for L in Equation (4.5):

vertex of the edge packing polytope (u_R, u_S, u_T)	$L = \left(\frac{N_R^{u_R} N_S^{u_S} N_T^{u_T}}{p}\right)^{\frac{1}{u_R+u_S+u_T}}$
$(1/2, 1/2, 1/2)$	$(N_R N_S N_T)^{1/3}/p^{2/3}$
$(1, 0, 0)$	N_R/p
$(0, 1, 0)$	N_S/p
$(0, 0, 1)$	N_T/p
$(0, 0, 0)$	0

The reader may convince herself that these five are the only vertices of the fractional edge packing polytope; for example, $(\frac{1}{3}, \frac{1}{3}, 0)$ is also a fractional edge packing, but it can be expressed as the convex combination $\frac{1}{3}(1,0,0) + \frac{1}{3}(0,1,0) + \frac{1}{3}(0,0,0)$. In the last row L is 0, because $(N_R^{u_R} N_S^{u_S} N_T^{u_T}/p)^{1/(u_R+u_S+u_T)} \leq \max(N_R, N_S, N_T)/(p^{1/(u_R+u_S+u_T)}) \to 0$ when $(u_R, u_S, u_T) \to (0,0,0)$. By Theorem 4.2, the optimal load of the HC algorithm is given by the largest of the five quantities in the table. To reduce the number of cases, it suffices to assume w.l.o.g. that $N_R \geq N_S \geq N_T$, in which case the maximum value is one of the first two rows, hence we have two cases.

The first case is when $(N_R N_S N_T)^{1/3}/p^{2/3} \leq N_R/p$, which we write equivalently as $\sqrt{\frac{N_S N_T}{p}} \leq \frac{N_R}{p}$, or also $p \leq N_R/\sqrt{N_S N_T}$. In this case the load of the Hypercube algorithm is $O(N_R/p)$. The relations S, T are small enough compared to R that the algorithm can ignore the variable z, and allocated $p_z = 1$ shares. More precisely, HC will compute the Cartesian product $S \times T$ using the Rectangle algorithm, by organizing the processors in a rectangle $[p_x] \times [p_y]$, while distributing $R(x,y)$ uniformly to all p processors. The load of the Cartesian product $S \times T$ is $\sqrt{\frac{N_S N_T}{p}}$, the load of partitioning R is N_R/p, and the latter dominates, hence the total load is $L = O(N_R/p)$. This implies that the algorithm has linear speedup $\sim p$ in the interval $p \in [1, N_R/\sqrt{N_S N_T}]$.

The second case is when $\sqrt{\frac{N_S N_T}{p}} > \frac{N_R}{p}$, or $p > N_R/\sqrt{N_S N_T}$. In this case the algorithm will allocate non-empty shares to all three variables, $p_x, p_y, p_z > 1$ (using the optimal solutions of the linear program (4.3), which are not necessarily $p_x = p_y = p_z = p^{1/3}$), and achieve the load $(N_R N_S N_T)^{1/3}/p^{2/3}$. Thus, the algorithm has sublinear speedup $\sim p^{2/3}$ in the interval $p \in (N_R/\sqrt{N_S N_T}, \infty)$.

4.1.7 Worst-case Optimal Algorithms

So far we have discussed and analyzed the Hypercube algorithm only on skew-free data. In this case, the HC with shares given by (4.3) is optimal among all 1-round algorithms. However, when the data is skewed, then these shares are sub-optimal; skewed data is best handled separately, using different choices of the shares. In other words, we will use different values of the shares for the heavy hitter tuples and for the

non-heavy hitter (or light) tuples.

In this section, we extend the Hypercube algorithm from skew-free data to arbitrary data, and describe a worst-case optimal algorithm called SkewHC. The principle is the same as that discussed for a single join in Chapter 3: heavy hitters are treated separately, by running the Hypercube algorithm with shares optimized differently. 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.

The SkewHC Algorithm Fix an input database instance I. We say that a value h over an attribute x_i in relation R_j is a *heavy hitter* in R_j if $\mathsf{deg}_{R_j[x_i]}(h) > N_j/p$. (This is similar to the definition of heavy hitter in Section 3.1.5.) Given an output tuple $t \in q(I)$, we say that t is *heavy at variable* x_i if the value $t.x_i$ is a heavy hitter in at least one of the relations R_j containing the variable x_i.

For any $\mathcal{X} \subseteq \text{vars}(q)$, let $I_{\mathcal{X}}$ denote the subset of the input database I that, for each relation R_j, includes only tuples $t \in R_j$ that are heavy exactly at $\mathcal{X} \cap \text{vars}(R_j)$. For example, if $\mathcal{X} = \emptyset$ then $I_{\mathcal{X}}$ is a skew-free subset of I, and when $\mathcal{X} = \text{vars}(q)$ then $I_{\mathcal{X}}$ consists only of tuples where all values are heavy hitters. If we consider all possible subsets of $\text{vars}(q)$, these induce a partition of the output tuples, hence we can recover the output over the whole instance I as follows:

$$q(I) = \bigcup_{\mathcal{X} \subseteq \text{vars}(q)} q(I_{\mathcal{X}})$$

For each subset of variables $\mathcal{X} \subseteq \text{vars}(q)$, define the *residual query* $q_{\mathcal{X}}$ as the query obtained from q by removing all variables in $\mathcal{X}$, decreasing the arity of each relation accordingly, then removing all relations whose arity became 0.

Algorithm 2, called SkewHC, proceeds as follows. First, it computes all heavy hitter values h of all attributes. There are at most $O(p)$ such values, because, for each relation R_j and attribute $x_i \in \text{vars}(R_j)$, there are at most N_j/p distinct heavy hitter values over x_i in R_j. These values can be computed and broadcast to all p processors, as we explained

in Section 3.1.5. Next, SkewHC evaluates q parallel[2] on all $2^{|\text{vars}(q)|}$ sub-instances $I_{\mathcal{X}}$ as follows. To compute $q(I_{\mathcal{X}})$, find the shares $\mathbf{p}^{\mathcal{X}}$ of the residual query $q_{\mathcal{X}}$ by solving Equation (4.3), and for every other variable, $x_i \in \mathcal{X}$, set $p_i = 1$; then run the Hypercube algorithm for q with shares $\mathbf{p}^{\mathcal{X}}$ over the subset $I_{\mathcal{X}} \subseteq I$. The final answer $q(I)$ is the union of all answers $q(I_{\mathcal{X}})$. Notice that we never strictly evaluate a residual query $q_{\mathcal{X}}$, but only use it to compute the optimal shares for q for the purpose of evaluating it on $I_{\mathcal{X}}$.

Algorithm 2 SkewHC Algorithm

Input: query $q = R_1(\mathbf{x}_1) \bowtie \cdots \bowtie R_\ell(\mathbf{x}_\ell)$.

1: **for** all $\mathcal{X} \subseteq \text{vars}(q)$ in parallel **do**
2: apply the HC algorithm parametrized by $\mathbf{p}^{\mathcal{X}}$ on $I_{\mathcal{X}}$
3: **end for**

Theorem 4.4. Let a be the largest arity of any relation R_j, and suppose that $p^{a+1} \leq \mathsf{IN}$. Then SkewHC computes any natural join query q in a single round using p servers with load

$$L = \tilde{O}\left(\max_{\mathcal{X} \subseteq \text{vars}(q)} \max_{\mathbf{u} \in \mathsf{pk}(q_{\mathcal{X}})} \left(\frac{\prod_j N_j^{u_j}}{p}\right)^{1/\sum_j u_j}\right) \tag{4.8}$$

Moreover, this load is optimal for all one-round algorithms.

Proof. Let us fix a set of variables $\mathcal{X} \subseteq \text{vars}(q)$; we show that the load to compute $q(I_{\mathcal{X}})$ is bounded by the expression (4.8). The upper bound then follows from the fact that we are running in parallel algorithms for all subinstances.

Consider any relation R_j. We distinguish two cases. If all variables of R_j are in $\mathcal{X}$, $\text{vars}(R_j) \subseteq \mathcal{X}$, then R_j is not included in the residual query $q_{\mathcal{X}}$. All its variables x_i have shares $p_i = 1$, which means that the whole relation R_j will be broadcast to all p processors during the computation of $q(I_{\mathcal{X}})$. However, observe that the part of R_j that contributes to $q(I_{\mathcal{X}})$

[2] In practice, the answers $q(I_{\mathcal{X}})$ can also be computed in separate communication rounds, to reduce the load per processor.

is of size at most p^{a_j}, where a_j is the arity of the relation. We claim that p^{a_j} is less than or equal to the expression (4.8), meaning that the load contributed by broadcasting R_j to all servers does not exceed (4.8). Indeed, by assumption $p^{a_j} \leq p^a \leq \mathsf{IN}/p$, and we also have

$$\frac{\mathsf{IN}}{p} = O(\max_j \frac{N_j}{p}) = O\left(\max_{\mathbf{u}\in\mathsf{pk}(q)} \left(\frac{\prod_j N_j^{u_j}}{p}\right)^{1/\sum_j u_j}\right)$$

because, on one hand $\mathsf{IN} = \sum_j N_j$ hence $\mathsf{IN} = O(\max_j N_j)$, and on the other hand $N_j/p = \left(\prod_j N_j^{u_j}/p\right)^{1/\sum_j u_j}$ when $\mathbf{u} \stackrel{\text{def}}{=} (0, 0, \ldots, 1, \ldots, 0)$ is the integral edge packing consisting of only the relation R_j.

It remains to compute the contribution to the load of the relations R_j that have at least one variable in $\mathcal{X}$, when we run the Hypercube algorithm with shares $\mathbf{p}^{\mathcal{X}}$. By Theorem 4.2 its load is $\tilde{O}\left(\max_{\mathbf{u}\in\mathsf{pk}(q_{\mathcal{X}})} \left(\prod_j N_j^{u_j}/p\right)^{1/\sum_j u_j}\right)$ provided that the instance $I_{\mathcal{X}}$ is skew-free w.r.t. the shares $\mathbf{p}^{\mathcal{X}}$ (as per Def. 4.1). It remains to prove that $I_{\mathcal{X}}$ is skew-free. Consider any tuple t over some attributes $A \subseteq \text{vars}(R_j)$, for some R_j. If $A \subseteq \mathcal{X}$, then, by construction, forall $x_i \in A$, its share is $p_i = 1$, hence Def. 4.1 allows the degree of t to be $\mathsf{deg}_{R_j[A]}(t) \leq N_j$, which trivially holds. If A contains at least one variable $x_i \notin \mathcal{X}$, then the value $t.x_i$ is a light hitter, hence by definition of the instance $I_{\mathcal{X}}$, $\mathsf{deg}_{R_j[x_i]}(t.x_i) \leq N_j/p$. This implies $\mathsf{deg}_{R_j[A]}(t) \leq \mathsf{deg}_{R_j[x_i]}(t.x_i) \leq N_j/p$, completing the proof that the instance $I_{\mathcal{X}}$ is skew-free. Hence, the execution of the HC algorithm with shares $\mathbf{p}^{\mathcal{X}}$ over $I_{\mathcal{X}}$ is skew-free, which means that we can apply Theorem 4.2 and its load is $O\left(\max_{\mathbf{u}\in\mathsf{pk}(q_{\mathcal{X}})} \left(\prod_j N_j^{u_j}/p\right)^{1/\sum_j u_j}\right)$. This proves the upper bound of the theorem.

For the lower bound, fix any $\mathcal{X} \subseteq \text{vars}(q)$, and notice that the algorithm computing q must compute q correctly on all input database consisting of a single heavy hitter for each variable in $\mathcal{X}$, and of a matching database for the residual query $q_{\mathcal{X}}$; by Theorem 4.3, any one-round algorithm requires a load $\Omega\left(\max_{\mathbf{u}\in\mathsf{pk}(q_{\mathcal{X}})} \left(\frac{\prod_j N_j^{u_j}}{p}\right)^{1/\sum_j u_j}\right)$ to compute q on such a database. Since the algorithm is assumed to

compute q correctly on any input database, the load is at least the maximum of the quantity above over all set of variables $\mathcal{X}$. □

Example 4.6. We illustrate SkewHC on the triangle query

$$\Delta(x, y, z) = R(x, y) \bowtie S(y, z) \bowtie T(z, x)$$

Assume for simplicity that the database is uniform, $N_R = N_S = N_T = N$; the general case is handled similarly. There are three different share allocations, for each choice of heavy variables (all other cases are symmetrical and omitted).

$\mathcal{X} = \emptyset$: we consider only tuples with values of degree $\leq N/p$. This corresponds to the skew-free case, and HC will assign shares $p_x = p_y = p_z = p^{1/3}$ to each variable. The maximum load will be $\tilde{O}(N/p^{2/3})$.

$\mathcal{X} = \{x\}$: the tuples have a heavy hitter value at variable x, either in relation R or T or in both. The residual query is

$$\Delta_{\{x\}}(y, z) = R_{\{x\}}(y) \bowtie S(y, z) \bowtie T_{\{x\}}(z)$$

where $R_{\{x\}}(y)$ represents $R(x, y)$ where we simply ignore the attribute x, and similarly for $T_{\{x\}}(z)$. The optimal shares for the residual query are $p_y = p_z = p^{1/2}$; we define $p_x = 1$, then run the Hypercube algorithm on the query q. Its load is $\tilde{O}(N/p^{1/2})$.

$\mathcal{X} = \{x, y\}$: both x and y are heavy. The residual query is

$$\Delta_{\{x,y\}} = S_{\{x,y\}}(z) \bowtie T_{\{x,y\}}(z)$$

In this case we assign shares $p_x = p_y = 1$, $p_z = p$. The load in this case is $\tilde{O}(N/p)$. Notice that we remove the relation $R_{\{x,y\}}()$ from the residual query for a simple technical reason: any query with a relation of arity zero has $\tau^* = \infty$. The relation $R(x, y)$ will be broadcast to all servers: since its size is only p^2, its contribution to the total load is $\leq N/p$ assuming $p^3 \leq N$.

$\mathcal{X} = \{x, y, z\}$: This case is handled by broadcasting all necessary information.

The load of the algorithm is the maximum of the above quantities, thus $\tilde{O}(N/p^{1/2})$. While the speedup on skew-free data is $\sim p^{2/3}$, over skewed data the speed-up deteriorates to $\sim p^{1/2}$. This is unavoidable: for example, if the database input has a single value for the variable x, then the query is equivalent to the residual query $\Delta_{\{x\}}$ and, since $\tau^*(\Delta_{\{x\}}) = 2$, by Theorem 4.3 any algorithm needs a load $\Omega(N/p^{1/2})$.

Edge Quasi-Packing As for the case of the HC algorithm, the formula for the load of the SkewHC algorithm is simpler when input database is uniform. Define the *edge quasi-packing number* of a query q to be the following quantity

$$\psi^*(q) \stackrel{\text{def}}{=} \max_{\mathcal{X} \subseteq \text{vars}(q)} \tau^*(q_{\mathcal{X}}) \tag{4.9}$$

If the database input is uniform, $N_1 = \cdots = N_\ell = N$, then the load of the SkewHC algorithm is $L = \tilde{O}(N/p^{1/\psi^*(q)})$. This follows immediately, because each residual query $q_{\mathcal{X}}$ has load $\tilde{O}(N/p^{1/\tau^*(q_{\mathcal{X}})})$.

For the triangle query, we have $\tau^*(\Delta) = 3/2$, while $\psi^*(\Delta) = 2$. As another example, consider the generalization of the join query, called *star join*:

$$T_k = R_1(z, x_1) \bowtie R_2(z, x_2) \bowtie \cdots \bowtie R_k(z, x_k)$$

The fractional edge packing number is $\tau^*(T_k) = 1$, since every relation includes the variable z. To obtain the maximum edge quasi-packing, we simply consider the residual query q_z that removes the common variable z. Then, we can pack each relation R_j with weight of one, thus achieving $\psi^*(T_k) = k$.

The edge quasi-packing number is always at least as large as the fractional edge packing number, that is, $\psi^*(q) \geq \tau^*(q)$, because ψ^* is the maximum over all $\tau^*(q_{\mathcal{X}})$ and the latter includes $\tau^*(q)$ when $\mathcal{X} = \emptyset$; this just means that the speedup on skewed data is no better than the speedup on skew-free data. As we saw, for some queries $\psi^*(q) > \tau^*(q)$, which means that we pay a penalty for the skew; we show in Section 4.2 that, for some queries, we can remove this penalty by using multiple rounds to handle the heavy hitters.

4.1.8 Output-Sensitive Algorithms

If the degrees of the "heavy" values of the relations are known, then it is possible to construct algorithms that perform better than worst-case algorithms, even in the case where the input has skew. Take for instance the join query $J = R(x, y) \bowtie S(y, z)$. In Chapter 3 we showed that we can compute the join in a single round with load

$$L = \tilde{O}\left(\frac{\mathsf{IN}}{p} + \sqrt{\frac{\mathsf{OUT}}{p}}\right)$$

Recall that the idea behind the join is to identify the heavy values of the join attribute y (which are the ones with degree $> \mathsf{IN}/p$), and for each such value compute the subinstance (which is a Cartesian product) by allocating an appropriate amount of processors. We can apply the same idea to the case of the triangle query.

Triangle Query For the triangle query, we define a value for x, y or z to be a *heavy hitter* if it has degree $> \mathsf{IN}/p^{1/3}$ for any of the two relations it belongs; otherwise, we say it is *light*. The algorithm will deal with the light values by running the vanilla HC algorithm, achieving load $\tilde{O}(\mathsf{IN}/p^{2/3})$. To handle the heavy hitters, we distinguish two cases.

Case 1. In this case, we handle the tuples that have values with degree $\geq \mathsf{IN}/p$ in at least two variables. Without loss of generality, suppose that both x, y are heavy in at least one of the two relations they belong to. The observation is that there are at most $2p$ such heavy values for each variable, and hence we can send all tuples in R with both x, y heavy (at most $4p^2$) to all processors. Then, it remains to compute the query $S'(y, z), T'(z, x)$, where x and y can take only p values. We can do this by running the parallel hash join algorithm; since the degree of z-values will be at most p for each relation, it is skew-free and the load will be $\tilde{O}(\mathsf{IN}/p)$.

Case 2. In this case, we handle the remaining output: this includes the tuples where one value has degree $\geq \mathsf{IN}/p^{1/3}$, and the other values

have degree $\leq \mathsf{IN}/p$. Without loss of generality, assume that we want to compute the query for the x-values that are heavy in either R or T. Observe that there are at most $2p^{1/3}$ such heavy hitters. If $\mathcal{H}_x$ denotes the set of heavy hitter values for variable x, the residual query for each $h \in \mathcal{H}_x$ is:

$$q_{\{x\}} = R(y) \ltimes S(y,z) \ltimes T(z)$$

where $R(y), T(z)$ are restricted to the tuples with x-value equal to h. Let d_h^R, d_h^T be the degrees of h in R, T respectively. We allocate an exclusive group of p_h servers for each heavy hitter, where:

$$p_h = p \cdot \max\left\{\frac{1}{p^{1/3}}, \frac{d_h^R d_h^S}{\sum_{j\in\mathcal{H}_x} d_j^R d_j^S}\right\}$$

Having Case 1 ensures that the input to the residual query is skew-free for every heavy hitter.

Summing up all the cases, we obtain that the load of the 1-round algorithm for computing triangles is:

$$\tilde{O}\left(\max\left\{\frac{\mathsf{IN}}{p^{2/3}}, \sqrt{\frac{\sum_h d_h^R d_h^S}{p}}, \sqrt{\frac{\sum_h d_h^R d_h^T}{p}}, \sqrt{\frac{\sum_h d_h^S d_h^T}{p}}\right\}\right)$$

The above algorithm is optimal for computing triangles in one round for the particular sequence of heavy hitter degrees. Notice that for skew-free instances the above formula gives a load of $\tilde{O}(\mathsf{IN}/p^{2/3})$, while in the case of extreme skew the load becomes equal to $\tilde{O}(\mathsf{IN}/p^{1/2})$, which is equal to the worst-case bound for the triangle query.

The ideas used in the above algorithms can be generalized to any join query by the BinHC algorithm proposed in [18]. However, it is not known whether BinHC is optimal, and hence it still remains an open problem to find a optimal single-round algorithm given the degree distribution for any natural join query.

4.2 Multiple Rounds

For a parallel algorithm in the MPC model, each additional round of communication incurs a significant cost. All servers must synchronize,

and thus wait for the slowest server, then the data must be reshuffled and, furthermore, skew in intermediate data may be much larger than in the input data. Nevertheless, there are cases where running a query in multiple rounds can significantly reduce the communication cost per round, and sometimes 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 for any algorithm that computes the query in a constant number of rounds. Unlike the skew-free/one-round case, where the load is expressed in terms of fractional edge packing number τ^* (e.g. the speedup is $\sim p^{1/\tau^*}$), this new skew/multi-round load is expressed in terms of ρ^*, the fractional edge covering number. The need for two different parameters is intriguing, as we discuss in Section 4.2.1.

Next, we discuss in Section 4.2.2 how to use multiple rounds to remove the penalty incurred by skewed values. We saw in Section 4.1.7 that, if we insist in computing the query in one round, then skew in the data leads to an increased cost from τ^* to ψ^*. As we shall see, $\psi^* \geq \max(\tau^*, \rho^*)$, and the inequality may be strict, hence in one round we exceed the absolute lower bound ρ^*. We will show that, in some cases, we can use additional rounds and reduce the load from ψ^* to ρ^*, which is optimal. The additional rounds are used to process the heavy hitters, and the number of rounds depends only on the structure of the query.

In Section 4.2.3 we discuss a more traditional usage of multiple rounds, namely in order to reduce the load, or the total communication cost, by computing simpler operators, one at a time. 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 intermediate results may increase significantly compared to the input size IN, so the joins in rounds $2, 3, 4, \ldots$ may be performed on much larger inputs. Thus, their load is significantly increased. A second challenge is that intermediate results tend to be more skewed than the input data. We describe

techniques based on Yannakakis' algorithm and on tree decomposition, which lead to an output sensitive algorithm, where the load is expressed in terms of both input and output.

Finally in Section 4.2.4 we examine the tradeoff between the number of rounds and the load per server, by discussing to some extent a special case, where this tradeoff can be analyzed precisely.

4.2.1 A Simple Multi-Round Lower Bound

We first present a lower bound for the best possible load of any multi-round algorithm. The construction of the lower bound is based on the idea of finding input instances that produce a large output result; in particular, it uses the *AGM bound* [9]. We start with a brief review of the AGM bound.

Edge cover Let q be a multiway natural join query, as given by Equation (4.1), with variables x_i, $i = 1, \ldots, k$ and relations R_j, $j = 1, \ldots, \ell$. An *edge cover* of q is a subset of its relations $R_{j_1}, R_{j_2}, \ldots$ that, together, contain all variables of the query. We denote by $\rho(q)$ the size of the smallest edge cover. A *fractional edge cover* is a set of non-negative weights $\{w_j \mid j = 1, \ldots, \ell\}$ such that for every variable x_i, $\sum_{j:x_i \in \text{vars}(R_j)} w_j \geq 1$. We denote by $\rho^*(q)$ the value $\min_{\mathbf{w}} \sum_j w_j$, where the minimum is taken over all fractional edge covers $\mathbf{w}$. Obviously, $\rho^*(q) \leq \rho(q)$.

In a celebrated result, Atserias, Grohe and Marx [9] proved the following:

Theorem 4.5. Let q be a natural join query (Equation (4.1)). Then:

- If every input relation R_j has size $\leq N$, then the output size of the query is $\leq N^{\rho^*(q)}$.
- The bound above is tight. In other words, there exists a "worst case" database instance where $|R_j| \leq N$ for every input relation R_j, and the output size is[3] $= N^{\rho^*(q)}$.

The quantity $N^{\rho^*(q)}$ is called the AGM bound of the query q.

[3]This holds only asymptotically, when $N \to \infty$, because of rounding errors.

We omit the proof of the first item, and instead give a weaker bound, which provides the intuition, by showing that $|q| \leq N^{\rho(q)}$. Indeed, for any edge cover, the answer to q is a subset of the Cartesian product of the relations in the cover, and the claim follows by choosing an optimal edge cover of size $\rho(q)$. The proof that $|q| \leq N^{\rho^*(q)}$ is less obvious, and is done by using Shearer's inequality in information theory.

We give now the proof of the second item. A set of non-negative numbers $\{v_i \mid i = 1, \ldots, |\text{vars}(q)|\}$ is a *fractional vertex packing* of q if for every relation R_j, $\sum_{i:x_i \in \text{vars}(R_j)} v_i \leq 1$. By duality, $\max_{\mathbf{v}} \sum_i v_i = \min_{\mathbf{w}} \sum_j w_j = \rho^*$, where $\mathbf{v}, \mathbf{w}$ range over all fractional vertex packings and edge covers respectively. The worst-case instance is defined as follows. Let $\mathbf{v}^*$ be an optimal fractional vertex packing. For each variable x_i define its domain $\mathbf{dom}(x_i) \stackrel{\text{def}}{=} [N^{v_i^*}]$, then define each relation instance to be the Cartesian product of the domains of its variables $R_j \stackrel{\text{def}}{=} \prod_{i:x_i \in \text{vars}(R_j)} \mathbf{dom}(x_i)$. Clearly $|R_j| = N^{\sum_{i:x_i \in \text{vars}(R_j)} v_i^*} \leq N$, and the answer to the query has size $|q| = N^{\sum_i v_i^*} = N^{\rho^*(q)}$.

Lower bound The following lower bound applies to any number of rounds:

Theorem 4.6 (Multiround Lower Bound [57])**.** Let q be a natural join query. Suppose $\mathcal{A}$ is an MPC algorithm that computes q in r rounds with load L. (1) If MPC is stateful, then $r \cdot L = \Omega(\mathsf{IN}/p^{1/\rho^*(q)})$. (2) If MPC is stateless, then $L = \Omega(\mathsf{IN}/(r \cdot p)^{1/\rho^*(q)})$.

Proof. We give here a sketch of the proof of (1). Fix a number N, and consider the worst case instance for the query q: the input has size $\mathsf{IN} = O(N)$, and the output has size $\mathsf{OUT} = N^{\rho^*(q)}$. Since each processor receives at most $r \cdot L$ tuples from each relation R_j, we can use the AGM bound to argue that the total number of tuples output by that processor is at most $(r \cdot L)^{\rho^*}$. Since the p processors must return all output tuples, we have $p(r \cdot L)^{\rho^*} \geq N^{\rho^*}$, which is the inequality claimed in the theorem. Of course, if the algorithm $\mathcal{A}$ needs to compute the output to a *single* input, then it can be designed to avoid any communication, so this proof idea will not work as given: instead, a rigorous proof is presented in [57], by considering a larger class of worst

case databases, whose input size is close to the N and whose output size is also close to $N^{\rho^*(q)}$. On that class, the load of any algorithm is $\Omega(N/p^{1/\rho^*(q)})$. The proof is quite general, and does not impose any restriction on what kind of information the processors encode in the message, instead uses an information-theoretic argument to show that the information received by the processors is insufficient to generate the required answer. For item (2), recall that for a stateless MPC we must allow the algorithm to produce outputs at each round. Using the same input database as above, at each round the algorithm can output at most $p \cdot L^{\rho^*}$ tuples. Since all tuples must be output during the r rounds, we have $r \cdot p \cdot L^{\rho^*} \geq N^{\rho^*}$, proving the claim. □

Contrast this theorem to Theorem 4.3: there, we showed that any one-round algorithm must have load $\Omega(\mathsf{IN}/p^{1/\tau^*})$, *even on skew-free databases*, and, moreover, there exists a matching algorithm. Now, we showed that, no matter how many rounds r we use, if $r = O(1)$ then the load is $\Omega(\mathsf{IN}/p^{1/\rho^*})$, and, moreover, this load is caused by the worst case instances, which are sometimes skewed. In the case when $\rho^* \geq \tau^*$ this means than, in order to process skewed data, the load must exceed that of the HyperCube algorithm on skew-free data, no matter how many rounds we use: in the next section we describe a restricted class of queries where we can process skewed data in multiple rounds, with a load matching $\tilde{O}(\mathsf{IN}/p^{1/\rho^*})$. In the case when $\rho^* \leq \tau^*$, we are left with open questions, since we don't know how to compute a query even on skew-free databases with load below $\tilde{O}(\mathsf{IN}/p^{1/\tau^*})$, even if we are allowed multiple rounds. Could we improve the upper bound $\mathsf{IN}/p^{1/\tau^*}$ given by the Hypercube algorithm? Or can we improve the lower bound $\mathsf{IN}/p^{1/\rho^*}$? We discuss multi-round algorithm that aim at improving the load of the Hypercube algorithm in Section 4.2.3. The exact complexity for general queries is open.

Three Examples We illustrate Theorem 4.6 with three simple examples. First, consider the join query $J(x, y, z) = R(x, y) \bowtie S(y, z)$. Here $\tau^* = 1$, $\rho^* = 2$. Parallel Hash-join computes this query with

load $\tilde{O}(N/p)$ on skew-free databases, but on the worst-case instance[4], $R \stackrel{\text{def}}{=} [N] \times [1]$, $S \stackrel{\text{def}}{=} [1] \times [N]$, the output has size N^2, hence we must have $r \cdot L = \Omega(N/p^{1/2})$. This generalizes our proof in Chapter 3 from one round $r = 1$, to arbitrary number of rounds.

Consider now the triangle query, $\Delta(x, y, z) = R(x, y) \bowtie S(y, z) \bowtie T(z, x)$. Here $\tau^* = \rho^* = 3/2$. If the input database is skew-free, then we can compute it in one round with load $O(N/p^{2/3})$. If the input database is skewed then we need a load $\Omega(N/p^{2/3})$ no matter how many rounds use, but it is not clear (yet) how to compute the query with this load if the database is skewed and, in fact, we have argued in Section 4.1.7 that, if we are restricted to one round, the load must be $\Omega(N/p^{1/2})$. We will show in the next section that, by using multiple rounds, we can reduce the load to $\tilde{O}(N/p^{2/3})$.

Finally, consider the query $q(x, y) = R(x), S(x, y), T(y)$. Here $\tau^* = 2$, $\rho^* = 1$. Computing it in one round requires a load of $N/p^{1/2}$ even on skew-free databases, but the more general lower bound for arbitrary number of rounds is only N/p. Thus, the Hypercube algorithm seems sub-optimal here even on skew-free database. We will show that this query can be computed in two rounds with load N/p. Hence, ρ^*, which is the smaller quantity in this case, captures the optimal load. However, in general we don't know what the optimal load is when $\rho^* < \tau^*$. An example is given by the query $R(x_1, x_2, x_3), S_1(x_1, y_1), S_2(x_2, y_2), S_3(x_3, y_3), T(y_1, y_2, y_3)$, where $\tau^* = 3$, $\rho^* = 2$. Our multi-round lower bound, expressed in terms of ρ^*, is $\Omega(N/p^{1/2})$. But, even assuming the input is skew-free database, the best known algorithm (Hypercube) has load $\tilde{O}(N/p^{1/3})$. This algorithm runs in one round, but we don't know how to improve its load by using multiple rounds.

Connection to the Quasi-Packing Number When restricted to one round of communication, the load for on skew-free databases is given in terms of τ^*, while on skewed databases is given in terms of ψ^*. We have already seen that $\psi^* \geq \tau^*$, and we should normally have $\psi^* \geq \rho^*$,

[4] An optimal vertex packing is $v_x = 1, v_y = 0, v_z = 1$, and the variable domains are $\mathbf{dom}(x) = [N^1]$, $\mathbf{dom}(y) = [N^0] = [1]$, $\mathbf{dom}(z) = [N^1]$.

since ρ^* is a lower bound on the load for *any* number of rounds. We prove here formally that $\psi^* \geq \rho^*$.

Fix a query q; we will show that $\psi^*(q) \geq \rho^*(q)$. Let $\mathbf{w}^*$ be an optimal fractional edge cover, in particular $\sum_j w_j^* = \rho^*$. Let $\mathcal{X}$ be the set of variables that have slack, more precisely $\mathcal{X} = \{x_i \mid \sum_{j:x_i \in \text{vars}(R_j)} w_j^* > 1\}$. Consider the residual query $q_{\mathcal{X}}$. We prove $\rho^*(q) \leq \tau^*(q_{\mathcal{X}})$, which immediately implies $\rho^*(q) \leq \psi^*(q)$. Recall that, in general, $q_{\mathcal{X}}$ contains only a subset of the relations R_j in q: those for which $\text{vars}(R_j) \subseteq \mathcal{X}$ are removed. The restriction of $\mathbf{w}^*$ to the relations in $q_{\mathcal{X}}$ forms a fractional edge packing for $q_{\mathcal{X}}$, because every remaining variable x_i is tight, meaning $\sum_{j:x_i \in \text{vars}(R_j)} w_j^* = 1$. Therefore, $\sum_j w_j^* \leq \tau^*(q_{\mathcal{X}})$, but here we need to be careful because j ranges only over the relations R_j in $q_{\mathcal{X}}$, hence it is not the case yet that $\rho^* = \sum_j w_j^*$. We will prove that $q_{\mathcal{X}}$ contains all relations R_j for which $w_j^* > 0$, which implies that the sum is over all values j, hence $\rho^*(q) = \sum_j w_j^* \leq \tau^*(q_{\mathcal{X}})$. Indeed, if some R_j with $w_j^* > 0$ were removed, then $\text{vars}(R_j) \subseteq \mathcal{X}$ which means that all variables in R_j have slack in $\mathbf{w}^*$: that contradicts the optimality of $\mathbf{w}^*$, since we can decrease w_j^* while still keeping it a fractional edge cover.

4.2.2 Removing the Skew Penalty with Multiple Rounds

If we are restricted to compute a query q in a single round, then we must pay a penalty to handle skewed data: the load increases from $\tilde{O}(\mathsf{IN}/p^{1/\tau^*(q)})$ on skew-free data to $\tilde{O}(\mathsf{IN}/p^{1/\psi^*(q)})$ on skewed data. For some queries, this penalty can be reduced by using multiple rounds: more precisely, the query can be computed with load $\tilde{O}(\mathsf{IN}/p^{1/\rho^*(q)})$, which is optimal by Theorem 4.6. For queries where $\tau^* = \rho^*$, this can be viewed as completely removing the penalty paid for skew. When $\rho^* > \tau^*$, we don't recover that penalty completely, but the algorithm is nevertheless optimal.

The building block consists of a simple algorithm, first described by [57], to compute the following two-way semi-join query:

$$q(x, y) = R(x) \bowtie S(x, y) \bowtie T(y)$$

Assume $|R| = |S| = |T| = N$. If we restrict to a single round, then

$L = \tilde{O}(N/p^{1/2})$, and this is optimal even on skew-free data, because $\tau^*(q) = \psi^*(q) = 2$. However, here $\rho^*(q) = 1$, and, it turns out that we can compute the query with load $\tilde{O}(N/p) = \tilde{O}(N/p^{1/\rho^*})$, in two rounds, as follows:

Round 1 Compute the intermediate table $A(x, y) = R(x) \bowtie S(x, y)$.

Round 2 Compute the result $q(x, y) = A(x, y) \bowtie T(y)$

In both rounds we use the output-aware join algorithm from Section 3.1.5. This algorithm needs to know the heavy hitters in S, and their degrees; an extra communication round may be needed to compute them, as we explained in Section 3.1.5. Recall that the load of the output-aware join algorithm is $\tilde{O}(\mathsf{IN}/p + \sqrt{\mathsf{OUT}/p})$, see Equation (3.3). For a semi-join, the size of the output is no larger than the input (more precisely, $|A| \leq |S|$ and $|q| \leq |S|$), and therefore, the load of the output-aware join in Round 1 is $\tilde{O}(N/p + \sqrt{N/p}) = \tilde{O}(N/p)$, because we assume that the input is much larger than p, hence $\sqrt{N/p} \leq N/p$. The same argument applies to round 2, therefore, the load for both rounds is $\tilde{O}(N/p)$.

Example 4.7. Using the two-way semijoin query as a building block, we show how to compute the triangle query $\Delta(x, y, z) = R(x, y) \bowtie S(y, z) \bowtie T(z, x)$ in 2 rounds with worst-case optimal load $\tilde{O}(\mathsf{IN}/p^{2/3})$, on arbitrary (possibly skewed) inputs. Recall Example 4.6, where we computed this query one round, with load $\tilde{O}(N/p^{1/2})$, using the SkewHC algorithm. We use here the same basic idea, with two changes: we modify the definition of heavy hitters, and we use two rounds to compute some of the residual queries. More precisely, a value h is a *heavy hitter* if for some relation the degree of h exceeds $\mathsf{IN}/p^{1/3}$; this is the same definition as in Section 4.1.8, but different from that used by SkewHC in Section4.1.7. The 2-round algorithm distinguishes two cases:

1. For the light values, it runs the vanilla HC algorithm in one round, with load $\tilde{O}(\mathsf{IN}/p^{2/3})$.

2. For the output tuples where at least one value is a heavy hitter, it does as follows. Without loss of generality, suppose variable x

has heavy values and observe that there are at most $2p^{1/3}$ such heavy values for x ($p^{1/3}$ for R and $p^{1/3}$ for T). For each heavy value h, we assign an *exclusive* set of $p' = p^{2/3}$ servers to compute the residual query $q' = R'(y), S(y,z), T'(z)$, where R', T' are restricted to tuples with value $x = h$. We can now compute q' in 2 rounds with load $\tilde{O}(\mathsf{IN}/p') = \tilde{O}(\mathsf{IN}/p^{2/3})$ by running one semi-join at each of the two rounds.

The two key takeaway ideas from this algorithm are: (i) a semijoin computation can be computed with optimal load, independent of the skew, and (ii) we can handle the heavy hitters by looking at the residual queries that occur once we fix a particular value.

More generally, for every multi-way natural join query whose relations have arity ≤ 2, it is possible to compute the query in $O(1)$ rounds with a load $\tilde{O}(\mathsf{IN}/p^{1/\rho^*})$ [54], which is optimal by Theorem 4.6. The high level idea of the algorithm is the same: (1) compute the query on the light hitters in one round, using the Hypercube algorithm. Here we use the fact that $\tau^*(q) \leq \rho^*(q)$ whenever all relation arities are ≤ 2, which means that the load of the Hypercube is $\tilde{O}(\mathsf{IN}/p^{1/\tau^*}) \leq \tilde{O}(\mathsf{IN}/p^{1/\rho^*})$. (2) compute the query on the heavy hitters, using additional rounds.

4.2.3 Ouput-Sensitive Algorithms

We discuss here a more traditional approach to use multiple rounds to decrease the load, namely by computing simpler operations at each round. We will express the query as a query plan, then compute the plan from the leaves to the root, where in each round we compute all operators on the same level in the plan.

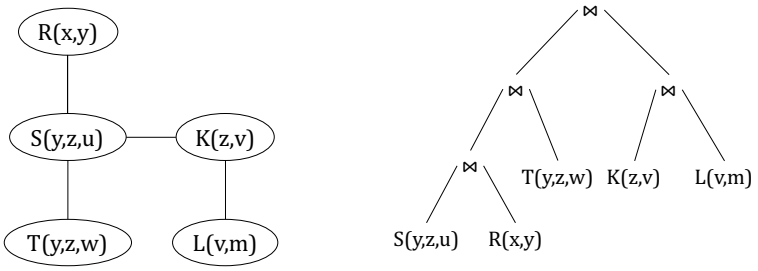
Yannakakis' Parallel Algorithm Let q be a conjunctive query, as given by Equation (4.1). The query q is called *acyclic* if there exists an undirected tree T whose nodes are in one-to-one correspondence with the relations in q such that the following property holds:

- For every variable x, the set of tree nodes that contain the variable x form a connected component.

Query:

$$Q(x, y, z, u, v, w, m) = R(x, y), S(y, z, u), T(y, z, w), K(z, v), L(v, m)$$

Tree decomposition and query plan (with S as root of the tree):



Semi-join reduction (with S as root of the tree):

Bottom-up: $K := K \ltimes L$; $S := S \ltimes R \ltimes T \ltimes K$;

Top-down: $R := R \ltimes S$; $T := T \ltimes S$; $K := K \ltimes S$; $L := L \ltimes K$;

Figure 4.2: Illustration of an acyclic query and its tree decomposition.

The tree T is called the tree decomposition of q. See Figure 4.2 for an illustration. Each acyclic query has a natural query plan, consisting of a sequence of join operators, as illustrated in the figure.

Yannakakis [90] described a simple algorithm for computing (sequentially) any acyclic query in time $\tilde{O}(\mathsf{IN} + \mathsf{OUT})$, by performing first a semi-join reduction of all relations, then computing the joins in any order. We review briefly the original, sequential algorithm, then describe the parallel variant. Recall that the semi-join of two tables, $R(x, y) \ltimes S(y, z)$, is defined as $\pi_{xy}(R(x, y) \bowtie S(y, z))$, in other words, it contains the subset of tuples in R that join with at least one tuple in S. The semi-join can be computed in time $\tilde{O}(\mathsf{IN})$. The semijoin is not commutative, however the following identity holds: $(R \ltimes S) \ltimes T = (R \ltimes T) \ltimes S$. Yannakakis' algorithm has two steps:

Semijoin Reduction: Pick an arbitrary node of the tree and call it the *root*, then perform semi-join reductions in two sweeps: (a) bottom-

up: from leaves to the root, semi-join each relation with each of its children, and (b) top-down: from the root to the leaves, semi-join each relation with its parent. This step takes time $\tilde{O}(\mathsf{IN})$.

Join: Compute all joins, in any order. This step takes time $\tilde{O}(\mathsf{OUT})$ because each tuple in each intermediate result is part of a tuple in the final output.

Figure 4.2 illustrates a simple acyclic query, its tree T, the semi-join reduction, and the final join computation.

Yannakakis' algorithm can be easily converted into a parallel algorithm. Let us denote by $\mathsf{IN}, \mathsf{OUT}$ the size of the input and output to the query q. Then, for the semi-join reduction step we use the output-sensitive join described in Section 3.1.5. It needs to know all heavy hitters and their degrees, and this may require another round of communication, and its load is $\tilde{O}(\mathsf{IN}/p + \sqrt{\mathsf{OUT}'/p}) = \tilde{O}(\mathsf{IN}/p)$, because the output of the semi-join, denoted here by OUT', is never larger than the input, and because $\sqrt{\mathsf{OUT}'/p} \leq \sqrt{\mathsf{IN}/p} \leq \mathsf{IN}/p$, assuming the input is much larger than the number of processors. For the second step of the algorithm, for each join we use the same output-sensitive join algorithm, which requires computing the heavy hitters for all intermediate relations (possibly requiring extra communication rounds), and has a load complexity $\tilde{O}(\mathsf{IN}'/p + \sqrt{\mathsf{OUT}'/p})$. Here $\mathsf{IN}', \mathsf{OUT}'$ denote the sizes of the input, and the output of this join operator. The semi-join reduction ensures that every intermediate output is no larger than the final output, hence $\mathsf{OUT}' \leq \mathsf{OUT}$. The input to this join consists of base tables and/or other intermediate relations, hence $\mathsf{IN}' \leq \mathsf{IN} + \mathsf{OUT}$. Therefore, the combined load of Yannakakis' parallel algorithm is $\tilde{O}((\mathsf{IN} + \mathsf{OUT})/p)$.

We illustrate with the path query of length ℓ:

$$P_\ell(x_0, \ldots, x_\ell) = R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \ldots \bowtie R_\ell(x_{\ell-1}, x_\ell)$$

Yannakakis' parallel algorithm first performs $2\ell - 1$ semi-joins to remove tuples from each relation that do not contribute to the final result: we first sweep from left to right, i.e. replace $R_j := R_j \ltimes R_{j-1}$ for $j = 2, 3, \ldots, \ell$, then we sweep from right to left, i.e., replace $R_{j-1} :=$

$R_{j-1} \ltimes R_j$, $j = \ell, \ell-1, \ldots, 2$. Then, we join the relations in some arbitrary order, e.g. $R_1 \bowtie R_2$, $(R_1 \bowtie R_2) \bowtie R_3$, etc. Therefore, the load of the algorithm is $\tilde{O}(\mathsf{IN}/p + \mathsf{OUT}/p)$, and the algorithm requires 3ℓ rounds. We compare this with the single round HyperCube algorithm, whose load is $\tilde{O}(N/p^{1/\tau^*})$, where $\tau^* = \lfloor \frac{\ell+1}{2} \rfloor$. To compare the two, notice that the worst case value of OUT is given by the AGM bound, $\mathsf{OUT} \leq N^{\rho^*}$, where $\rho^* = \lceil \frac{\ell+1}{2} \rceil$, thus, when the database is the "worst case", then the single round, HC algorithm is best. However, often the output is much smaller, and in that case the term N/p dominates the load of the multiround algorithm, which is much better than the load of the Hypercube algorithm.

The GYM Algorithm The GYM algorithm [3] is a natural extension of Yannakakis' Parallel algorithm to arbitrary natural join queries, by using a tree decomposition of the query. Before presenting GYM, we briefly review here the notion of a tree decomposition and its standard applications to sequential query processing; we refer the reader to [15] for a survey or to [55] for an excellent technical treatment.

Let q be an arbitrary natural join query, not necessarily acyclic.

Definition 4.2 (Tree Decomposition)**.** A *tree decomposition* of a natural join query q is a pair (T, χ), where:

- T is a tree,
- for every node $t \in \text{Nodes}(T)$, $\chi(t)$ is a subset of $\text{vars}(q)$, called the *bag* of t, such that:
  1. for each variable $x \in \text{vars}(q)$, the set of nodes $\{t \mid x \in \chi(t)\}$ is a connected component of T, and
  2. for each relation R_j of the query q, $\text{vars}(R_j)$ is contained in some bag: $\text{vars}(R_j) \subseteq \chi(t)$ for some t.

Figure 4.3 shows an example query and its tree decomposition. We note that our definition of a tree decomposition is equivalent to what is called in the literature a *generalized hypertree decomposition* [42]. A slightly more restrictive notion is called a *hypertree deomposition*,

Query:

$$Q = R(x,y), A(y,s), B(x,s), S(y,z,u), T(y,z,w), \\ C(z,w,t), D(w,t,y), E(t,y,z), K(z,v), F(z,v), L(v,m)$$

Standard tree decomposition (along with the bag queries q_t):

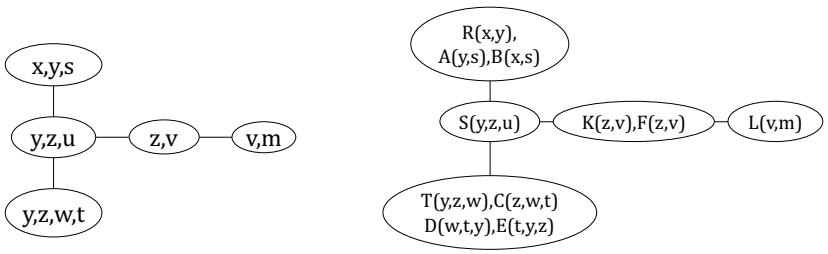


Figure 4.3: Illustration of a tree decomposition.

but we will follow common practice and use the the term hypertree decomposition, or just tree decomposition, to refer to the generalized tree decomposition, see the discussion in [43, Section 4].

Fix a tree decomposition T. For each tree node $t \in \text{Nodes}(T)$, we define q_t the following query. q_t has variables $\chi(t)$, and has one relation $R'_j \stackrel{\text{def}}{=} \pi_{\chi(t)}(R_j)$ for each relation R_j in q. Thus, the query q_t is the natural join of the projections on the variables in bag of t of all relations in q, in other words, $\text{vars}(R'_j) = \text{vars}(R_j) \cap \chi(t)$. We will call q_t the bag query at t. The *generalized hypertree width* of T, $ghtw(T)$, is the maximum edge covering number of all bag queries q_t. The *fractional hypertree width*[5] of T, $fhtw(T)$, is the maximum fractional edge covering number of all bag queries q_t

$$ghtw(T) \stackrel{\text{def}}{=} \max_t \rho(q_t)$$
$$fhtw(T) \stackrel{\text{def}}{=} \max_t \rho^*(q_t)$$

[5]Strictly speaking, this quantity should be called *fractional generalized hypertree width*, but it was coined fractional hypertree width in [43, Def.4.3].

The generalized (or fractional) hypertree width of the query q is defined as the minimum of these quantities over all tree decompositions:

$$ghtw(q) \stackrel{\text{def}}{=} \min_T ghtw(T)$$
$$fhtw(q) \stackrel{\text{def}}{=} \min_T fhtw(T)$$

Given a hypertree decomposition T of a query q, one can evaluate q in two steps, as follows:

Compute the bags For each tree node t, evaluate the bag query q_t, and store the result in an intermediate table. Notice that the intermediate tables form an acyclic query, with tree T.

Run Yannakakis' algorithm on the acyclic query consisting of the temporary tables, and return the result.

The complexity of this algorithm is dominated by the cost of computing the bag queries q_t. Two approaches have been described in the literature. The first approach (see, e.g., the survey [42]) chooses an optimal edge cover $\lambda(t) = \{R'_{j1}, R'_{j2}, \ldots\}$ for each bag query q_t, then computes their cross product, then semi-joins it with every relation of the query. The cost of computing all the intermediate tables is therefore $\tilde{O}(\sum_t N^{\rho(q_t)}) = \tilde{O}(N^{ghtw(T)})$, and the total cost of the algorithm is $\tilde{O}(N^{ghtw(T)} + \mathsf{OUT})$. As a historical comment, once we have chosen an edge cover $\lambda(t)$ for each node t, the triple (T, χ, λ) is called a *generalized hypertree decomposition* of q, and the generalized hypertree width of T can defined equivalently as $ghtw(T) = \max_t |\lambda(t)|$

The second approach (see, e.g., [55]) uses a worst-case optimal algorithm for computing q_t, such as Generic Join [68] or TrieJoin [85], whose runtime is $\tilde{O}(N^{\rho^*(q_t)})$. By using this approach, the entire cost of computing q becomes $\tilde{O}(N^{fhtw(T)} + \mathsf{OUT})$. Of course, in either case we want to choose the tree decomposition T that minimizes the costs, in that case the two optimal costs become $\tilde{O}(N^{ghtw(q)} + \mathsf{OUT})$ and $\tilde{O}(N^{fhtw(q)} + \mathsf{OUT})$ respectively.

The GYM algorithm [3] applies the same steps to compute the query in parallel, rather than sequentially: compute the bags first, then run

Yannakakis' parallel algorithm. We analyze the number of rounds and the load of the algorithm next.

While T is an undirected tree, we can convert it into a directed tree by choosing (arbitrarily) a root note. Then height(T) is the length of the longest path from the root to some leaf; if the root node is not specified, we assume it is chosen such as to minimize height(T). GYM runs in $O(\text{height}(T))$ rounds, by observing that Yannakakis' parallel algorithm can be run in parallel on different branches of the tree.

Next we analyze the load of GYM. For the first step we will use the SkewHC algorithm to compute each query q_t; then the load of the first step is $\tilde{O}(N/p^{1/(\max_t \psi^*(q_t))})$. Alternatively, if possible, we could use an optimal multi-round algorithm with load $\tilde{O}(N/p^{1/(\max_t \rho^*(q_t))})$, but, as we have seen, such algorithms are known only for some queries. For the second step we use Yannakakis' parallel algorithm for acyclic queries. The inputs are now the intermediate relations corresponding to the bags, and each bag t will have size $\leq N^{\rho^*(q_t)}$, by the AGM bound. Therefore, the load of the second step is $O((N^{\max_t \rho^*(q_t)} + \mathsf{OUT})/p)$

To summarize, given a query q, for any tree decomposition T of q we can compute q in parallel, in r rounds and with load L, where:

$$r = O(\text{height}(T)) \tag{4.10}$$

$$L = O\left(\frac{N}{p^{1/(\max_t \psi^*(q_t))}} + \frac{N^{\max_t \rho^*(q_t)} + \mathsf{OUT}}{p}\right) \tag{4.11}$$

4.2.4 Trading off Rounds for Communication

We discuss here the tradeoff between the number of rounds and the load, in Equations (4.10) and (4.11). One extreme of the tradeoff is to choose the tree decomposition T that minimizes the load L in Equation (4.11); the number of rounds is at most $O(\ell)$ (number of relations in q), since T can always be chosen to have at most ℓ nodes. The other extreme is to choose the tree decomposition consisting of a single bag, labeled with all variables in the query. Then height(T) = 1, and we are computing the query in a single round, using SkewHC, but now the load is much increased. In this section we discuss the space in between these two extremes. We start with a brief discussion of the techniques

used in GYM, then briefly present an exact formula for this tradeoff, in the case of skew-free database instances.

As part of the GYM algorithm [3], the authors show how we can reduce the number of rounds to logarithmic in the size of the query, $O(\log(\ell))$, by considering query decompositions with lower depth, but larger width. The number of nodes in a tree decomposition can always be decreased artificially. For example, a naive way is to choose two adjacent nodes in T and merge them (by creating a new node whose associated bag the union of the two bags), then repeat this process until the depth of the tree is reduced. However, by reducing the number of bags in the tree, we also increase the load L in Equation (4.11).

The authors in [3] showed that, any tree decomposition T can be converted into a tree decomposition T' such that $\text{height}(T') = O(\log(\ell))$, and $ghtw(T') \leq \max(ghtw(T), 3iw(T))$, where $iw(T)$ is the *intersection width*[6] of T. While the best load of GYM is given by the rather complicated formula Equation (4.11), the authors in [3] use a simpler formula, based on the generalized hypertree width, and, thus, can bound the load increases needed to reduce the number of rounds to $O(\log(\ell))$. We illustrate with an example.

Example 4.8. Consider the path query

$$P_k(x_0, \ldots, x_k) = R_1(x_0, x_1) \bowtie R_2(x_1, x_2) \bowtie \cdots \bowtie R_k(x_{k-1}, x_k)$$

The optimal (hyper-) tree decomposition consists of a linear tree T, with width 1, shown in Figure 4.4. This means that GYM takes $O(k)$ parallel rounds of communication. The second phase of Yannakakis' parallel algorithm can be easily done in $O(\log(k))$ rounds, by choosing a query plan that is a perfectly balanced binary tree. However, the first step, namely the semi-join reduction, requires $O(k)$ rounds.

However, there exists an alternative tree decomposition T' with width 3, as illustrated in Figure 4.5. Its depth is only $\log(k)$, and

[6]The intersection width of T is defined as $iw(T) = \max_{t,t'} \rho(q_{t,t'})$, where t, t' range over adjacent nodes in the tree, $q_{t,t'}$ is the query obtained by projecting all input relations on the common variables in t, t', i.e. projecting on $\chi(t) \cap \chi(t')$; this definition is analogue to that of q_t right after Def. 4.2; as usual, $\rho(q_{t,t'})$ denotes the edge covering number. Notice that $\rho(q_{t,t'}) \leq \rho(q_t)$ and $\rho(q_{t,t'}) \leq \rho(q_{t'})$, implying $iw(T) \leq ghtw(T)$.

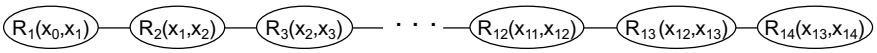


Figure 4.4: A tree decompositions T for the query P_{15}. Its width is 1 and its height is 15.

therefore we can perform the semi-join reduction in $O(\log(k))$ parallel rounds; the penalty is that its width has increased from 1 to 3. We explain how the tree decomposition of P_k can be obtained in general. Assume $k = 2^n - 1$. Start by placing the k relations $R_i(x_{i-1}, x_i)$ in a complete binary tree: the leaves consist of the odd-numbered relations $R_1, R_3, \ldots$, the internal nodes contain the even-numbered relations. This is not a correct tree decomposition yet, because it fails the first property of Def. 4.2, for example, in the figure, the root node $R_8(x_7, x_8)$ shares the variables x_7, x_8 with the leaves $R_7(x_6, x_7)$ and $R_9(x_8, x_9)$, and these variables do not (yet) occur anywhere else, hence they do not form a connected component in T'. To fix this, it suffices to add, to each internal node in the tree, the relations on its left-most, and its right-most leaf. In other words, the internal nodes will be of the form $R_a(x_{a-1}, x_a), R_i(x_{i-1}, x_i), R_b(x_{b-1}, x_b)$ where R_i is the even-numbered relation corresponding to the internal node, and R_a, R_b are odd-numbered relations corresponding to its left-most, and right-most leaf in T'. Each internal node t contains 3 relations; moreover, if its distance from the leaves is ≥ 2, then the 3 relations share no common variables, thus $\rho(q_t) = \rho^*(q_t) = 3$, and therefore $ghtw(T') = fhtw(T') = 3$.

Beame et al. [17] investigated the quantitative relationship between number of rounds and load per processor, in the restricted case when the input relations are matchings. Recall from Section 4.1.5 that a matching database is one where each attribute in each relation is a key. A matching database is skew-free, and greatly simplifies the analysis. Any intermediate result is no larger than the smallest input, and, thus, there is no need for a semi-join reduction. For that reason we will not compute a tree decomposition for the query, instead we will only describe query plans.

Fix a query q, and denote by $d(x_i, x_j)$ the distance between the

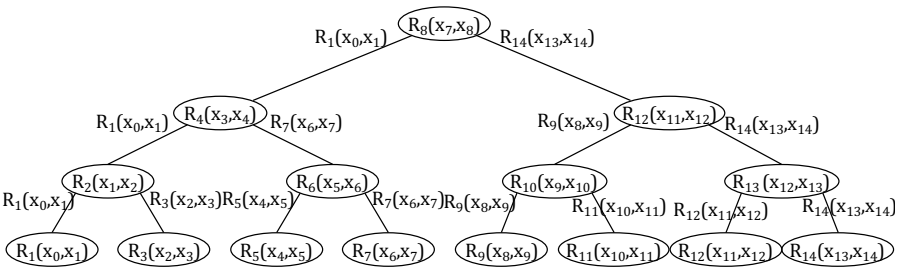


Figure 4.5: An alternative tree decomposition T for the query P_{15}. Its width is 3 and its height is 4.

nodes x_i, x_j in the hypergraph of q. The *radius* of q is defined as

$$\mathsf{rad}(q) = \min_u \max_v d(u, v)$$

For example, $\mathsf{rad}(P_k) = \lceil k/2 \rceil$, where P_k is the path query in Example 4.8.

At a high level, the connection between the number of rounds and the load needed to compute a query q on is the following: for any $\nu > 0$, one can compute the query in $r = \log_{2^\nu}(\mathsf{rad}(q)) + c$ rounds, where $c = 1$ or $c = 2$ (depending on the query), with load $L = \tilde{O}(\mathsf{IN}/p^{1/\nu})$, and this is, essentially, optimal. In particular, in order to compute the query with the ideal load $\tilde{O}(\mathsf{IN}/p)$, one needs to spend $\Omega(\log(\mathsf{rad}(q)))$ rounds.

We start with an example, which is also the main tool needed in the proof of the theorem below.

Example 4.9. Consider the path query P_k in Example 4.8. Since $\tau^*(P_k) = \lceil k/2 \rceil$, we can compute P_k in one round using the Hypercube algorithm, with a load $L = \tilde{O}(\mathsf{IN}/p^{1/\lceil k/2 \rceil})$. For example, if $k = 16$, then the load to compute P_{16} in one round is $L = \tilde{O}(\mathsf{IN}/p^{1/8})$.

Suppose we decide to compute P_{16} in two rounds, to reduce the load. In round one, compute, in parallel, the following intermediate relations:

$$\begin{aligned} T_1 &= R_1 \bowtie \cdots \bowtie R_4 \\ T_2 &= R_5 \bowtie \cdots \bowtie R_8 \\ T_3 &= R_9 \bowtie \cdots \bowtie R_{12} \end{aligned}$$

$$T_4 = R_{13} \bowtie \cdots \bowtie R_{16}$$

In the second round compute the query answer by joining these four relations:

$$P_{16} = T_1 \bowtie \cdots \bowtie T_4$$

Thus, we compute five sub-queries: four in parallel in the first round, and one in the second round. Each of the five queries is isomorphic to P_4, i.e. it is a chain of 4 relations, and therefore $\tau^* = 2$. It follows that the load of each of the five sub-queries, and therefore the load of the entire plan for P_{16}, is $\tilde{O}(\mathsf{IN}/p^{1/2})$.

Finally, we can decide to lower the load even further: we can compute P_{16} in 4 rounds, using a traditional bushy join tree of depth 4: in round one compute $R_1 \bowtie R_2$, $R_3 \bowtie R_4$, etc; in round two join pairs of results from round 1, i.e. compute $R_1 \bowtie \cdots R_4$; $R_5 \bowtie \cdots R_8$; etc; in round three join pairs of results from round 2, i.e. compute $R_1 \bowtie \cdots R_8$ and $R_9 \bowtie \cdots R_{16}$; and finally join them in round 4.

To state the general theorem connecting the number of rounds to the load, we need one more technical definition. We say that a query q is *tree-like* if (i) it is connected, and (ii) $\sum_{j=1}^{\ell}(|\text{vars}(S_j)|-1) = |\text{vars}(q)|-1$. For instance the path query P_k is tree-like, since $|\text{vars}(q)| = k+1$, and $\sum_j(|\text{vars}(S_j)|-1) = \sum_j(2-1) = k$. In fact, a query over a binary vocabulary is tree-like if and only if its hypergraph is a tree. Over non-binary vocabularies, if a query is tree-like then it is acyclic, but the converse does not hold in general.

Theorem 4.7. Let $\nu > 0$ be an integer, and q be any connected join query. Define

$$r(q) = \begin{cases} \lceil \log_{2\nu}(\mathsf{rad}(q)) \rceil + 1 & \text{if } q \text{ is tree-like,} \\ \lfloor \log_{2\nu}(\mathsf{rad}(q)) \rfloor + 2 & \text{otherwise.} \end{cases}$$

Then, q can be computed in $r(q)$ rounds on any matching database with maximum load $L = \tilde{O}(\mathsf{IN}/p^{1/\nu})$. Moreover, if the query is tree-like, then the load is optimal for the given number of rounds. (A similar result for non tree-like queries is not known.)

Proof. By definition of $\mathsf{rad}(q)$, there exists some node $v \in \text{vars}(q)$, such that the maximum distance of v to any other node in the hypergraph of q is at most $\mathsf{rad}(q)$. If q is tree-like then we can decompose q into a set of at most $|\ell|^{\mathsf{rad}(q)}$ (possibly overlapping) paths $\mathcal{P}$ of length $\leq \mathsf{rad}(q)$, each having v as one endpoint; here ℓ represents the number of relations in q. Since it is essentially isomorphic to L_ℓ, a path of length $\ell \leq \mathsf{rad}(q)$ can be computed in at most $\lceil \log_{2\nu}(\mathsf{rad}(q)) \rceil$ rounds with load $\tilde{O}(\mathsf{IN}/p^{1/\nu})$. Moreover, all the paths in $\mathcal{P}$ can be computed in parallel, because $|\mathcal{P}|$ is a constant depending only on q. Since every path will contain variable v, we can compute the join of all the paths in one final round with load $\tilde{O}(\mathsf{IN}/p)$.

If the query is not tree-like, then q may also contain atoms that join vertices at distance $\mathsf{rad}(q)$ from v that are not on any of the paths of length $\mathsf{rad}(q)$ from v: these can be covered using paths of length $\text{rad}(q) + 1$ from v. To get the final formula, we apply the equality $\lceil \log_a(b+1) \rceil = \lfloor \log_a(b) \rfloor + 1$, which holds for positive integers a, b. □

Notice that the theorem applies only to 2 or more rounds. In other words, the theorem makes no claim about the load for one round. It turns out that, one round is an outlier, as the following example shows.

Example 4.10. Consider the query

$$\begin{aligned} SP_k(z, x_1, \ldots, x_k, y_1, \ldots, y_k) = & R_1(z, x_1) \bowtie S_1(x_1, y_1) \bowtie \\ & R_2(z, x_2) \bowtie S_2(x_2, y_2) \bowtie \\ & \cdots \\ & R_k(z, x_k) \bowtie S_k(x_k, y_k) \end{aligned}$$

Since $\tau^*(SP_k) = k$, computing the query in round requires a load of $\tilde{O}(\mathsf{IN}/p^{1/k})$. However, using 2 rounds we can compute SP_k with load $\tilde{O}(\mathsf{IN}/p)$, as follows. In the first round we compute in parallel the following intermediate tables:

$$\begin{aligned} T_1(z, x_1, y_1) = & R_1(z, x_1) \bowtie S_1(x_1, y_1) \\ T_2(z, x_2, y_2) = & R_2(z, x_2) \bowtie S_2(x_2, y_2) \\ & \cdots \\ T_k(z, x_k, y_k) = & R_k(z, x_k) \bowtie S_k(x_k, y_k) \end{aligned}$$

Each sub-query is a 2-way join, has $\tau^* = 1$, and hence can be computed with load $\tilde{O}(\mathsf{IN}/p)$.

In the second round we join these intermediate tables to compute the final answer:

$$SP_k = T_1(z, x_1, y_1) \bowtie \cdots \bowtie T_k(z, x_k, y_k)$$

Here, too, $\tau^* = 1$, because the attribute z appears in every input relation. In other words, this query can be computed using a simple partitioned hash-join, by partitioning all relations T_i on the same attribute z; recall that all input relations are assumed to be matchings, hence so are the intermediate tables, and therefore there is no skew, and the hash partition will have a load of $\tilde{O}(\mathsf{IN}/p)$.

The query in this example has a very bad load when computed in one round, but its load becomes optimal if we compute it in two rounds. The smooth connection between the number of rounds and the load in Theorem 4.7 does not hold for one round.

4.3 Multiway Join Algorithms In Existing Systems

We end this chapter with a brief discussion on the algorithms that are used in practice to perform multiway joins. To our best knowledge, all existing relational or big data systems (except Myria [88]) use traditional binary join plans, e.g., a left-deep or bushy plan, when evaluating multiway joins. In other words, two base or intermediate relations of the query are joined iteratively using one of the two-way join algorithms supported by the system (see Section 3.3), until all of the relations are joined. The most popular multiway join queries in data warehouses are star joins, which are special cases of acyclic queries. Query plans can be very efficient for star joins. In particular, if the dimension tables are significantly smaller than a large fact table (which is quite common), then a broadcast join is particularly effective: simply broadcast the dimension tables without reshuffling the fact table. Some systems have specialized optimizations for star joins. For example, Hive [83] reduces the number of rounds by broadcast joins by broadcasting multiple dimension tables in a single round.

The only system we are aware of that uses one of the algorithms we have reviewed in this chapter is Myria, which uses an algorithm called *Tributary join* to evaluate some multiway join queries [26, 88]. Tributary join uses Hypercube to shuffle the input tuples in one communication round, then uses a variant of the serial worst-case optimal Leapfrog TrieJoin algorithm [85] to perform the joins locally in each processor. References [26] and [88], study different aspects of running Hypercube in practice. For example, recall that the LP 4.3 we described in Section 4.1.1 can output fractional share values, which cannot be used in practice. So one challenge is to ensure that one can pick good integral share values, which the authors address. These references also experimentally study when Tributary join outperforms traditional binary join plans. For example, they demonstrate that for two clique queries, a triangle and 4-clique query, Tributary join outperforms regular binary joins on a subset of Twitter's whom-follows-whom graph, whereas regular binary join plans work better on an acyclic query on a knowledge graph. We refer the readers to [26, 88] for more details.

In a *subgraph query* one is given a small graph and has to report all occurrences of the small graph in a much larger input graph. A subgraph query is a natural instance of a multiway join. If we represent the edges of an input graph as (src, dst) tuples of a binary `edges(`x_i`,` x_j`)` relation, subgraph queries can be shown to be equivalent to multiway self-joins on this table. Similarly SPARQL [78] queries in RDF triple stores, which are used to model and query semantic information on the web, can be equivalently thought of as multiway joins on (subject, predicate, object) triples. As a result distributed graph processing systems and RDF triple stores also support multiway join queries. Similar to database processing systems many of the subgraph algorithms resort to binary join plans for these queries [46, 59, 61, 67, 80, 92]. Several systems have optimizations, such as indexing commonly appearing subqueries, e.g., triangles, to speed up these plans. We omit a detailed review of these systems. Among graph processing systems we only mention *EmptyHeaded* [1], which is a shared-memory (not distributed) system that uses GYM-style query plans (recall Section 4.2.3) based on tree decompositions of queries. EmptyHeaded finds a gener-

alized hypertree decompsition D of an input query with the minimum ghtw. Then, it evaluates each bag of D uses a parallel version of a serial worst-case optimal join algorithm called *Generic Join* [68]. Finally, similar to GYM, EmptyHeaded runs a parallel version of Yannakkakis's algorithm to compute the final join. We believe integrating similar algorithms and plans that are different than traditional binary join plans in existing distributed systems is a very promising and interesting research direction.

5

Sorting

Sorting is a fundamental operation in data processing, both as a stand-alone task and as a subroutine in other queries. In sequential query processing, merge-join uses sorting to join two relations [71], while in parallel query processing sorting can also be used to join two tables, as we have seen in Section 3.2. Sorting can be used in many other data processing tasks, for example Hu et al. [45] describe how to use sorting for computing the parallel similarity join of two relations.

This chapter presents a brief overview of parallel sorting for shared-nothing architectures. We start with a review of lower bounds on the communication cost for parallel sorting, then we review three classical parallel sorting algorithms: odd-even merge sort, Cole's algorithm for PRAM, and Goodrich's sorting algorithm for the BSP model.

Throughout this chapter we will denote with N the number of input items to be sorted. The take away message of this chapter is that optimal sorting requires $\Theta(\log_L(N))$ rounds of communication, and $\Theta(N \log_L(N))$ total communication. The lower bounds are independent of p, for example we cannot sort in fewer rounds by using more processors. The upper bounds are given by algorithms that use at least $p = N/L$ processors, which is necessary since this is the

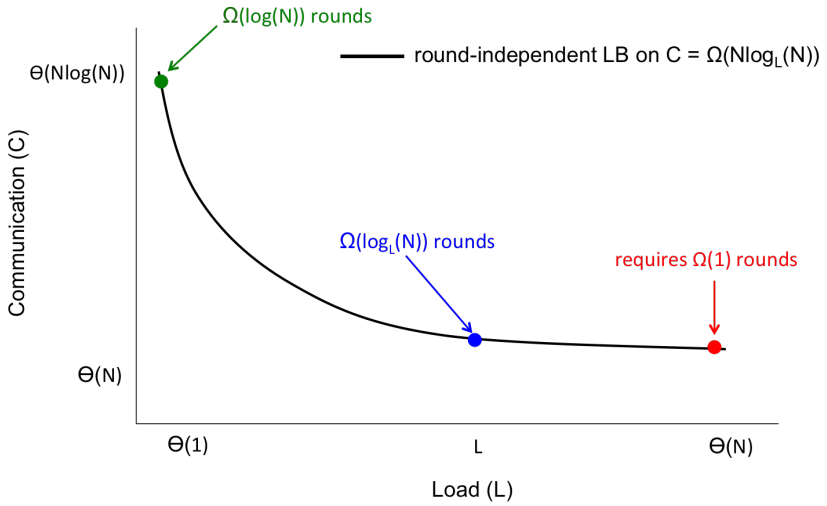


Figure 5.1: Lower Bounds on Sorting.

smallest number of processors required to store the entire data. Importantly, these bounds hold for any load L. For example, the parallel sort by regular sample (PSRS) algorithm described in Chapter 3 has a load $L = O(N/p)$ for $p \leq N^{1/3}$. For such values of L we have $\log_L(N) \in (1, \frac{3}{2})$ and the general bounds become $\Theta(1)$ rounds and $\Theta(N)$ total communication, matching those of the PSRS algorithm. Figure 5.1 summarizes these results.

5.1 Lower Bounds for Parallel Sorting

We describe here two simple lower bounds for parallel sorting on 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 $\Omega(N \log_L N)$; the second bound shows that the number of rounds is $\Omega(\log_L N)$. For the bound on the amount of communication, we assume a comparison-based model: the algorithm has an oracle that, given two inputs x, y, decides whether $x < y$, or $x > y$ (we assume that all input elements are distinct), and this oracle is the only way in which the algorithm can test the input items. The bound

on the number of rounds applies to any computational model, and, thus, is stronger.

We start with the lower bound on the total amount of communication, whose proof is an adaptation of Aggarwal et. al. [7]'s I/O complexity of sorting in the External Memory model (recall Section 2.3.1). Consider any distributed system that sorts N input items, such that at each moment, any processor can hold at most L items from the input. Processors may send messages arbitrarily, i.e., we allow the communications to be either bulk-synchronous or asynchronous, and the processors may perform arbitrary computations. Examples of such systems are a stateless MPC model with load L, and a BSP model with memory $M = L$. We let C denote the total number of input items sent and received between processors during the entire computation. Notice that this only counts the input items; any other information exchanged by the processors is not included in C.

Theorem 5.1. Consider any comparison based algorithm that sorts N input items on a distributed system, where each processor can hold at most L input items at any time. Assume that initially the input items are distributed arbitrarily on the processors without replication. Then, the minimum amount of communication required by the algorithm is $\Omega(N(\log_L(N)) - N)$.

Proof. Any correct comparison-based sorting algorithm must also be able to identify the input permutation. There are $N!$ possible permutations of the input items, and at the end the algorithm must be able to identify the correct input permutation. Each comparison performed by the algorithm reduces the size of the set of possible permutations, by keeping only the permutations consistent with the comparisons performed so far, until this number reaches 1. Let C denote the number of data items that need to be exchanged by the processors in order reduce the number of possible permutations to 1. Initially, each processor u holds $L_u \leq L$ items, where $\sum_u L_u = N$ (since we assume no replication of the input data). W.l.o.g. we assume that each processor performs all possible comparisons leading to $L_u!$ possible outcomes. So across all

N/L processors there are $\prod_u L_u! \leq (L!)^{N/L}$ possible outcomes[1], and leaves the algorithm in the best case with $N!/(L!)^{N/L}$ permutations that are consistent with all comparisons performed by all processors.

Let S_k denote the number of remaining permutations after the algorithm has exchanged k data items through communication, thus:

$$S_0 = \frac{N!}{(L!)^{N/L}} \qquad S_C = 1 \tag{5.1}$$

We now compute S_{k+1} inductively from S_k. Suppose one data item e is sent to some processor u. After receiving this data item, u can compare e with all its other $L-1$ data items (assumed to be already sorted), and establish its location. There are L possible locations of e relative to the $L-1$ items. Therefore if there were S_k possible permutations consistent with the comparisons, then, after communicating one more data item e, the number of remaining possible permutations decreases to $\frac{S_k}{L}$. In fact, one can construct inductively an adversarial input sequence such that:

$$S_{k+1} \geq \frac{S_k}{L} \tag{5.2}$$

Equations (5.1) and (5.2) imply:

$$1 = S_c \geq \frac{S_0}{L^C} = \frac{N!}{(L!)^{N/L} \cdot L^C}$$
$$C = \Omega(N \log_L(N) - N)$$

□

When $L = N$ then the bound in Theorem 5.1 is 0, which is as expected, since in that case the entire input can be stored on a single processor. In other words, the term $-N$ in the bound represents a saving in communication due to the initial distribution of the data.

The second bound is a lower bound on the number of rounds needed to sort N items. The bound applies to any sorting algorithm.

Theorem 5.2. [40] The minimum number of rounds needed by any MPC algorithm to sort N items is $\Omega(\log_L(N))$.

[1]The product $\prod_u L_u!$ is maximized when $L_1 = L_2 = \cdots = L_{N/L} = L$, and $L_u = 0$ for $u > N/L$.

Proof. We only provide the proof of the theorem when the MPC algorithm is oblivious (recall Section 2.2.1). By Theorem 2.1, any function F, in which one of the outputs depends on all inputs, requires a circuit of depth at least $\log_s(N)$ when the fan-in of the circuit is s, no matter how large the size of the circuit is. Recall also by Theorem 2.2 that any oblivious MPC algorithm with p processors, load L and r rounds can be represented by a circuit with $W = p \cdot r$ nodes, fan-in $s = L$, and depth $D = r$. Since sorting depends on all its inputs, there is no circuit of depth less than $\log_s(N)$ and therefore no oblivious MPC algorithm with load L that can run in $\log_L(N)$ rounds.

The proof for non-oblivious MPC algorithms requires a similar but more complex argument. The original proof is a corollary of a result from reference [40] that states that computing the OR of N bits on a BSP machine where each machine can read and write L bits is $\Omega(\log_L(n))$. The theorem follows from the observation that if we could sort n items in r rounds, we can also compute the OR of N bits in r rounds, since the OR is simply 1 if the maximum number is 1 and 0 otherwise. We refer the readers to reference [40] for the details. □

We make two observation. First, both Theorem 5.1 and 5.2 hold even if the processors have unlimited amount of main memory, as long as the number of data items that they can receive (and remember between rounds) is bounded by L. Second, neither bound depends on the number of processors p. Instead, the load alone, L, determines the total communication and the number of rounds. 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 entire input data but, beyond that, p is not a relevant factor in determining the distributed complexity of sorting. This contrasts with both conjunctive queries and matrix multiplication, where one can sometimes affect communication/load or rounds by increasing p. For example, if we restrict a sorting algorithms to run in r rounds, then $r = \Omega(\log_L N)$, which implies that the load is $L = \Omega(N^{1/r})$; we cannot reduce the load below this quantity by using more processors. In contrast recall that in conjunctive queries given that, say $r = 1$, L still depends on p, e.g., $L = \Omega(\mathsf{IN}/p^{1/\tau^*})$ on skew-free databases, which means that we lower L by increasing p.

Table 5.1: Winners of Gray Sort.

Year	Winner	Time	p and Memory/Processor
2016	Tencent Sort [49]	134s	512 (512GB)
2015	FuxiSort [87]	377s	3134 (96GB) + 243 (128GB)
2014	TritonSort [28]	1378s	186 (244GB)
2014	Apache Spark [89]	1406s	207 (244GB)
2013	Hadoop [82]	4328s	2100 (64GB)
2011	TritonSort [73]	8274s	52 (24GB)

5.2 Parallel Sorting Algorithms

Today's implementations of parallel sorting are designed for coarse-grained parallelism, where the number of processors p is much smaller that the number of data items N, and each processor has a relatively large memory (usually N/p). For example, the simple Parallel Sort by Regular Sample (PSRS) that we discussed in Section 3.2 is a coarse-grained parallel algorithm, since it assumes $p^3 \leq N$.

To illustrate this point, consider the Sort Benchmark [76], a very popular parallel sorting competition. In one of its benchmarks, GraySort, the task is to sort a large file of over 100TB, where each record has 100 bytes, and the key consists of 10 bytes, and therefore the input size is about 10^{12}, i.e., $N \sim 10^{12}$. The winners of this competition since 2011 are shown in Table 5.1. The table shows the time the systems took to sort 100TB of data and the number of processors used by the corresponding system. The numbers in parentheses in the last column are the memory used per processor by the algorithm. As shown in the table, some years the number of processors used by the winners have been as low as 52, and in all cases $p^3 < N$, hence all these implementations would be considered coarse-grained. All these algo-

rithms use the same main idea of PSRS: find $p-1$ splitters, partition the data in one round, then sort locally. However, they speed up the computation of the splitters, e.g. by using sampling, as opposed to the local sort in PSRS.

Most of the theoretical research on parallel sorting algorithms has focused instead on the much harder case of fine-grained parallelism, either in the PRAM model, or in some kind of circuit model, such as a sorting network. The ideal algorithm would use N processors and run in parallel time $O(\log N)$, but, as we shall see, sometimes we need to settle for less. In this section we describe three algorithms: Batcher's odd-even merge sort [16], Cole's optimal sorting algorithm for the PRAM model [27], and Goodrich's optimal sorting algorithm for the BSP model [40]. All three are based on merge sort: to sort a sequence of N items, first sort recursively in parallel the first half, and the second half of the sequence, then merge the two sorted subsequences. The number of recursive calls is $\log N$, hence the parallel time is given by the time needed to merge two sorted sequences.

5.2.1 Odd-Even Merge Sort

The merge operation, `merge`$(\mathbf{a}, \mathbf{b})$, takes as input two sorted sequences $\mathbf{a} = (a_1 < a_2 < \ldots)$ and $\mathbf{b} = (b_1 < b_2 < \ldots)$ and produces the sorted output consisting of their union. (For simplicity we assume all elements are distinct.) The standard sequential merge algorithm consists of repeatedly comparing the first elements of the two sequences and outputting the smallest; it is inherently sequential, and takes $O(N)$ steps. Batcher [16] describes a parallel algorithm for merging two sequences, called the `odd-even-merge`. It runs in $\log(N)$ parallel steps, and has total work $N \log(N)$.

Given a sequence $\mathbf{x} = x_1 x_2 x_3 \cdots$ we denote by $\text{odd}(\mathbf{x}) = x_1 x_3 x_5 \cdots$ and $\text{even}(\mathbf{x}) = x_2 x_4 x_6 \cdots$ Odd-even merge works as follows:

1. Recursively merge $\text{odd}(\mathbf{a})$ and $\text{odd}(\mathbf{b})$ to obtain $c_1 c_2 \cdots$
2. Recursively merge $\text{even}(\mathbf{a})$ and $\text{even}(\mathbf{b})$ to obtain $d_1 d_2 \cdots$
3. Interleave the two results, $c_1 d_1 c_2 d_2 c_3 d_3 \cdots$ then, in one parallel

step, for each $i = 1, 2, \ldots$ compare d_i and c_{i+1} and swap them if necessary.

The first two steps are done in parallel and, therefore, the total number of parallel steps is equal to the depth of the recursion tree in the `odd-even-merge` subroutine, which is $\log(N)$. The correctness of the algorithm follows from a very clever observation: in the interleaved sequence $c_1 d_1 c_2 d_2 c_3 d_3 \cdots$ the only elements that can be out of order are the adjacent elements d_i, c_{i+1}. To prove this, we show a simple property of `odd-even merge`, which will also be useful to understand the more complex algorithms below. Namely, we show that, for every i, the rank of c_i in the final sorted sequence is either[2] $2i-2$ or $2i-1$, and, similarly, the rank of d_i in the final sequence is either $2i$ or $2i+1$. The output rank of an element c_i is equal to the total number of elements in both **a**- and **b**-sequences that are $\leq c_i$. We prove the property. Suppose c_i is a_{2k-1}; the case when c_i is b_{2k-1} is similar and omitted. Clearly, the only **a**-elements that are $\leq c_i$ are $a_1, a_2, \ldots, a_{2k-1}$. Therefore, the sequence $c_1, \ldots, c_i$ contains precisely $\ell \stackrel{\text{def}}{=} i - k$ **b**-elements that are $< c_i$. Since these are odd **b**-elements, it means that $b_1, \ldots, b_{2\ell-1}$ are $< c_i$ while $b_{2\ell+1} > c_i$; the only **b**-element we are unsure about is $b_{2\ell}$. Hence, the rank of c_i is either $(2k-1)+(2\ell-1) = 2i-2$ or $(2k-1)+(2\ell) = 2i-1$.

`Odd-even merge` is not work-optimal: the total amount of work is $W = N\log(N)$, while a simple sequential algorithm can merge two sorted sequences in time (and work) N.

The `odd-even merge` algorithm immediately leads to a sorting algorithm that runs in $O(\log^2(N))$ parallel steps: sort recursively and in parallel the first half, and the second half, then merge the two sequences. The parallel time is thus $\log(N)+\log(N/2)+\log(N/4)+\cdots = \log(N)(\log(N)+1)/2 = O(\log^2 N)$. This raises the question whether we can reduce the number of parallel steps to $O(\log N)$.

In fact, it is easy to sort in $O(\log N)$ parallel steps in the PRAM model, by using $p = N^2$ processors. Assuming the input sequence is $x_1, x_2, \ldots, x_N$, use one processor for each pair of indices $i \neq j$ to compute the value $y_{ij} = 1$ if $x_j < x_i$ or $y_{ij} = 0$ if $x_j > x_i$ (recall that we assumed no ties); this can be done in $O(1)$ parallel steps using

[2] When $i = 1$, then the output rank of c_1 is 1.

$N(N-1)$ processors. Next, compute in parallel for each $i = 1, N$ the quantity $r_i = \sum_j y_{ij}$, which is the rank of x_i in the sorted sequence; this can be done in $O(\log N)$ parallel steps using N^2 processors. Finally, copy each item x_i to the position r_i in the output sequence. The total number of parallel steps is $O(\log N)$, but the total work (and number of processors) is $O(N^2)$. This leaves open the question whether we can sort in $O(\log N)$ parallel step *and* $O(N \log N)$ total work.

The first optimal sorting algorithm that achieves these bounds is due to Ajtai, Komlós, and Szemerédi [8], who described a sorting network of size $O(N \log(N))$ and depth $O(\log(N))$, called today the AKS sorting network. The algorithm, however, is not practical since the constants involved are too large. The first practical optimal algorithm was described by Cole.

5.2.2 Cole's Sorting Algorithm on the PRAM Model

Any merge-sort algorithm requires $\log N$ calls of the merge function. We have seen that one can merge two sequences in $O(\log N)$ parallel steps. This can be further improved to $O(\log \log N)$, which is tight: Borodin and Hopcroft [21] proved that merging two sorted sequences requires $\Omega(\log \log N)$ parallel steps. Thus, using the standard divide-and-conquer strategy, the best merge-sort algorithm runs in $(\log N \log \log N)$ parallel steps. Cole [27] described an algorithm that sorts in $O(\log N)$ parallel steps, by using pipelining. His algorithm overlaps the parallel steps of the merge functions, such that the amortized cost for merging is reduced to $O(1)$. We describe here the main ideas, following both [27] and [39], and omit many details.

At the core of Cole's algorithm is a procedure called `merge-with-help`$(\mathbf{a}, \mathbf{b}, \mathbf{c})$, which takes as input three sorted input sequences; $\mathbf{a}$ and $\mathbf{b}$ are the two input sequences to be merged, while $\mathbf{c}$ is the *sampler sequence* that is used as splitters to merge $\mathbf{a}$ and $\mathbf{b}$ and must have the following properties:

- Between any $t+1$ adjacent elements of $\mathbf{c}$ there are at most $2t+1$ elements in $\mathbf{a}$ and at most $2t+1$ elements in $\mathbf{b}$. For example, between c_i, c_{i+1} there are at most 3 $\mathbf{a}$-elements and at most 3 $\mathbf{b}$-elements, while between c_i, c_{i+2} there are at most 5 $\mathbf{a}$ and $\mathbf{b}$

elements. We say that $\mathbf{c}$ is a *good sampler* for $\mathbf{a}$ and for $\mathbf{b}$ respectively.

- For each element a_k we know its rank in $\mathbf{c}$, i.e. the number i s.t. $c_i \leq a_t < c_{i+1}$; we say that $\mathbf{a}$ is *ranked* in $\mathbf{c}$. Similarly, we require $\mathbf{b}$ to be ranked in $\mathbf{c}$, and vice versa, $\mathbf{c}$ is ranked in both $\mathbf{a}$ and $\mathbf{b}$.

`Merge-with-help`$(\mathbf{a}, \mathbf{b}, \mathbf{c})$ can be computed in $O(1)$ parallel steps on $O(N)$ processors, by using the same principle as in `odd-even merge`: there are only a limited number (namely 6) of possible output ranks for each element a_k, thus, with $O(1)$ comparisons, a processor can compute the correct output rank of a_k; similarly for b_ℓ. More precisely: in parallel, for each pair of adjacent elements c_i, c_{i+1}, identify the elements a_k, and the elements b_ℓ that fall in the interval c_i, c_{i+1} (at most 3 of each), merge them (in $O(1)$ time), then compute their final rank. We invite the reader to check the details and convince herself that `merge-with-help` can be computed in $O(1)$ parallel steps.

Cole's algorithm sorts an input sequence $x_1, \ldots, x_N$ in a *bottom-up* and *pipelined* fashion as follows. Consider a binary tree with N leaves corresponding to the input elements, and consider a generic node v in the tree. The node is responsible for computing the sorted sequence of the elements $x_i, x_{i+1}, \ldots, x_j$ below v, in parallel, using $j - i + 1$ processors. Denoting by $\mathbf{U}(v)$ this sorted sequence, the node v will compute $\mathbf{U}(v)$ by merging the two sorted sequences $\mathbf{U}(w_1)$ and $\mathbf{U}(w_2)$ at its children. However, it will not receive these sequences all at once, but instead it receives a stream of increasingly larger subsequences $\mathbf{L}(w_1)^{(1)}, \mathbf{L}(w_1)^{(2)}, \ldots$, each of size double the previous, and similarly a stream $\mathbf{L}(w_2)^{(1)}, \mathbf{L}(w_2)^{(2)}, \ldots$, and will compute a stream $\mathbf{U}(v)^{(k+1)}$ by merging $\mathbf{L}(w_1)^{(k)}$ with $\mathbf{L}(w_2)^{(k)}$, using $\mathbf{U}(v)^{(k)}$ as helper sequence. As its local sequence $\mathbf{U}(v)^{(k)}$ doubles its size after each step, node v also sends to the parent a stream of increasing sequences, namely the sequences $\mathbf{L}(w_j)^{(k)} = \texttt{even}(\texttt{even}(\mathbf{U}(w_j)^{(k)}))$. Essentially, at each iteration a node receives a 4-regular sample[3] of the current sequences of its children. We use the convention that a sequence with superscript k represents a sequence of length is 2^k. Formally, Cole's algorithm is

[3] A 4-regular sample consists of every 4'th element.

the following. At each node v perform the following:

- receive from its children new sequences $\mathbf{L}(w_1)^{(k)}$, $\mathbf{L}(w_2)^{(k)}$.
- compute $\mathbf{U}(v)^{(k+1)} = \texttt{merge-with-help}(\mathbf{L}(w_1)^{(k)}, \mathbf{L}(w_2)^{(k)}, \mathbf{U}(v)^{(k)})$.
- send to the parent $\mathbf{L}(v)^{(k+1)} = \texttt{even}(\texttt{even}(\mathbf{U}(v)^{(k+1)}))$.

After the node finishes (i.e. the length of $\mathbf{U}(v)^{(t)}$ is equal to the number of the elements below v), then it sends to its parent $\texttt{even}(\mathbf{U}(v)^{(t)})$ in the next step, then sends $\mathbf{U}(v)^{(t)}$ in one more step. Thus the parent finishes 3 steps behind v. After $3\log(N)$ parallel steps, the root r has computed its sequence $\mathbf{U}(r)$, which is the sorted sequence of the entire input. The algorithm is (a) a bottom-up algorithm because children pass samples of their currently sorted elements to their parents; and (b) a pipelined algorithm because instead of waiting to complete sorting their descendants children pass samples to their parents as soon as they get samples from their children.

Pictorially, the streaming sequences are as follows [39]:

lc	$\mathbf{L}(w_1)^0$	$\mathbf{L}(w_1)^1$	$\mathbf{L}(w_1)^2$	...	$\mathbf{L}(w_1)^{k-1}$	
rc	$\mathbf{L}(w_2)^0$	$\mathbf{L}(w_2)^1$	$\mathbf{L}(w_2)^2$	...	$\mathbf{L}(w_2)^{k-1}$	
v	$\mathbf{U}(v)^1$	$\mathbf{U}(v)^2$		...	$\mathbf{U}(v)^k$	
p		$\texttt{e}(\texttt{e}(\mathbf{U}(v)^2))$	...	...	$\texttt{e}(\texttt{e}(\mathbf{U}(v)^k))$	$\texttt{e}(\mathbf{U}(v)^k)$	$\mathbf{U}(v)^k$

where lc, rc, p stand for left-child, right-child, and parent, while $\texttt{e}(\cdots)$ stands for $\texttt{even}(\cdots)$. We omit many subtle details of the algorithm, which must ensure that all required invariants needed for `merge-with-help` hold, and refer the reader to [27, 39] for details. Summarizing:

Theorem 5.3. Cole's algorithm sorts N items on the PRAM model using N processors in $O(\log(N))$ parallel steps.

While described for the finest parallel model, with N processors, Cole's algorithm easily adapts to a coarse grained model. Given $p < N$

processors, one can sort in parallel time $O(N/p \log(N))$, by simply having each new processors simulate N/p of the processors in Cole's algorithm. Obviously, as p decreases, the parallel time increases. However, Cole's algorithm is restricted to the PRAM model. A naive implementation in the MPC model leads to $O(\log(N))$ communication rounds, possibly more for a fine grained model. Goodrich addressed this limitation by describing a non-trivial extension of Cole's algorithm to the BSP model.

5.2.3 Goodrich's Sorting Algorithm

Goodrich [40] describes an optimal algorithm for the BSP model, by doing a careful adaption of Cole's algorithm. Given the number of processors p, the algorithm uses $M = N/p$ local memory for each processor, and runs $O(\log_M(N))$ communication rounds. Notice that, when p decreases, the number of parallel rounds *decreases*: when $p = O(N^\varepsilon)$ for some $\varepsilon < 1$, then Goodrich's algorithm requires $O(1)$ communication rounds, and is similar to the coarse-grained algorithms used in practice today (it becomes similar to the PSRS).

The algorithm is quite complex and to our best knowledge is not used in practice. However, for completeness and for its great theoretical value as the optimal deterministic distributed algorithm, we give here a high-level overview.

High-level Overview

Instead of considering binary tree as in Cole's algorithm, let d be $\max\{\lceil\sqrt{N/p}\rceil, 2\}$ and T be a d-way balanced and complete tree as shown in Figure 5.2. For simplicity we assume $\sqrt{N/p}$ is an integer greater than 1, so $d = \sqrt{N/p}$. Each leaf has $N/p = d^2$ elements. Initially the elements assigned to each leaf is stored together in a single processor in the cluster. Instead of a 2-way bottom-up pipelined computation, Goodrich's algorithm does a d-way bottom-up pipelined computation. Specifically at iteration k, at each node v we perform the following:

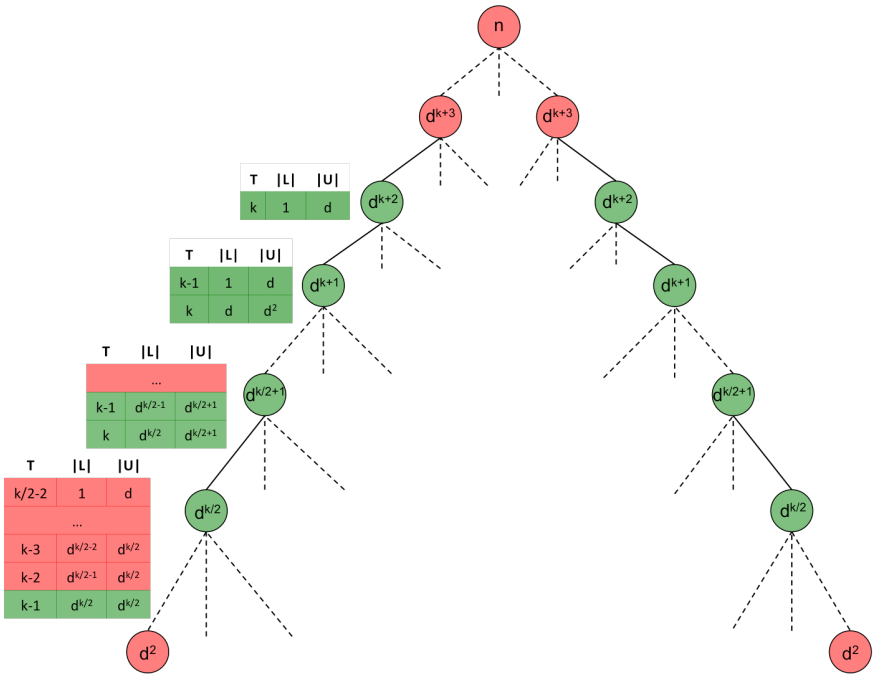


Figure 5.2: State of Goodrich's algorithm at stage k.

- receive from its children new sequences $\mathbf{L}(w_1)^{(k)}, \ldots \mathbf{L}(w_d)^{(k)}$.
- compute $\mathbf{U}(v)^{(k+1)} = \texttt{merge-with-help}(\mathbf{L}(w_1)^{(k)} \ldots \mathbf{L}(w_d)^{(k)}, \mathbf{U}(v)^{(k)})$.
- send to the parent $\mathbf{L}(v)^{(k+1)} = \texttt{d}(\texttt{d}(\mathbf{U}(v)^{(k+1)}))$.

In the above description $d(d(\mathbf{U}(v)^{(k+1)}))$ is a d^2-regular sample, one in every d^2th element, of $\mathbf{U}(v)^{(k+1)}$. $\texttt{merge-with-help}(\mathbf{L}(w_1)^{(t)} \ldots \mathbf{L}(w_d)^{(k)}, \mathbf{U}(v)^{(k)})$ is a generalization of Cole's `merge-with-help` subroutine that takes $O(1)$ rounds and uses $\mathbf{U}(v)^{(k)}$ as splitters to merge $\mathbf{L}(w_1)^{(k)} \ldots \mathbf{L}(w_d)^{(k)}$. The sizes of $\mathbf{L}(w_i)^{(k)}$ are now d^k instead of 2^k. Similar to Cole's algorithm, after the length of $\mathbf{U}(v)^{(t)}$ is equal to the number of descendants of v, v sends to its parent $\texttt{d}(\mathbf{U}(v)^{(t)})$ in the next step, then sends $\mathbf{U}(v)^{(t)}$ in one more step. So again, the parent finishes 3 steps behind v.

Figure 5.2 shows a snapshot of the algorithm at the end of stage k. The figure depicts the *active* nodes, those that have not yet finished sorting their descendants or passing samples to their parents, as green. The *inactive* nodes that have either not yet received any samplers from their children or are completely done sorting their descendants are depicted red. The tables at the levels where there are active nodes show the sizes of the L_k and U_k lists stored at the nodes in that level. In the table, the first column is the stage number k. We emphasize that nodes in T *do not* correspond to a processor in a distributed cluster. Except for the leaves, the algorithm uses multiple processors to sort the samples of samples of elements assigned to a node in each stage. Let L be the maximum number of elements that any processor can hold at any point. Then pipelining ensures that the algorithm uses asymptotically optimal $O(\log_L(N)$ rounds. In each stage k only the U_k, L_k, U_{k-1}, and L_{k-1} sets of active nodes are sorted and stored in processors, which ensures that the algorithm uses linear communication in each round. Therefore the communication cost of the algorithm is $O(N \log_L(N))$ and is optimal. In addition, the entire algorithm is deterministic justifying the following theorem:

Theorem 5.4. [40] Let L be maximum number of elements that any processor can hold at any point. For any value of L, Goodrich's deterministic algorithm uses $O(\log_L(N))$ rounds, $O(N \log_L(N))$ communication, which is asymptotically optimal.

Although we only gave the very high-level ingredients of the algorithm, we acknowledge that the full algorithm and its analysis is quite complex. We refer the readers to reference [40] for details and we believe it is an interesting question whether the algorithm and its analysis can be further simplified than its original description.

5.3 Discussion

We end this chapter by noting that for comparison-based algorithms the complexity of sorting is completely understood. Theorems 5.1 and 5.2 capture fully the relationship between the four variables of our model,

N, p, L and C, by providing lower bounds that are matched by the upper bound of Goodrich's sorting algorithm. For example, if we fix L, by Theorems 5.1 and 5.2 we directly get closed-form formulas for r and C. As another example, if we asked what's the minimum load L of any algorithm A that runs in r rounds with a total communication costs C, the answer is $\max\{N^{N/C}, N^{1/r}\}$, where $N^{N/C}$ comes from Theorem 5.1 and $N^{1/r}$ comes from Theorem 5.2. Some of these lower bounds can be trivial for example if we asked what's the minimum number of processors needed to sort N numbers if the total communication cost is C, where $C \geq N$, then the answer is 1, because L is not fixed. Nonetheless there is always a closed-form answer. This contrasts with the state-of-art in conjunctive queries, where the answers to some of these questions are open.

6

Matrix Multiplication

There have been several interesting results on the relation of local memory, number of machines, number of rounds, and communication for performing linear algebra computations in the BSP model. The computations studied include basic operations such as multiplications of dense and sparse matrices, LU and QR factorization, as well as computations involving sequences of these operations. This chapter gives a short 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, and briefly review the results for other computations. We present both lower bound results on these parameters and algorithms that match these bounds.

Matrix multiplication is closely related to a specific relational query. Consider the multiplication of two $n \times n$ matrices $A \cdot B = C$. Assume that the matrices A and B are stored in tables $A(i, j, v)$ and $B(j, k, v)$, where i, j and j, k represent the index of a non-zero entry, and v is the value of that entry. Then, one can express matrix multiplication as a query consisting of a join and a group-by-and-aggregate:

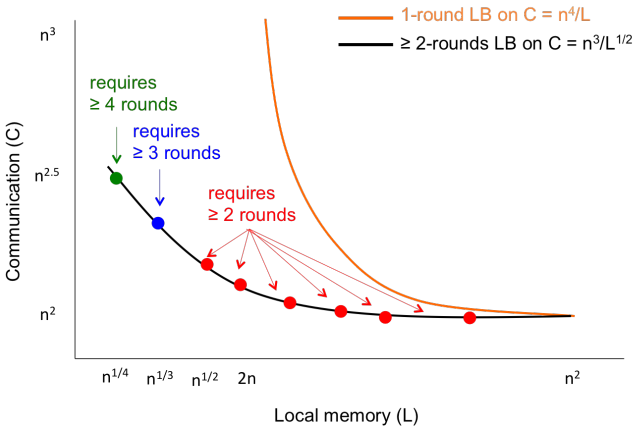


Figure 6.1: Communication Lower Bounds for Dense Matrix Multiplication.

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

We will refer to this query as the *matrix multiplication query*. In existing relational systems, this query will be evaluated by what is equivalent to a conventional matrix multiplication algorithm, as the system will perform all of the elementary multiplications inside the sum aggregation. As we will show, if we focus on conventional algorithms, one can study the complexities of the join part of this query to arrive at interesting communication lower bounds and on the join and on group-by-and-aggregate separately to derive round lower bounds.

This chapter starts with several lower bounds for conventional matrix multiplication algorithms in Section 6.1, then describes algorithms with matching upper bounds in Section 6.2. These results are summarized in Figure 6.1. We discuss the high level takaways in Section 6.3, and briefly list other related work in Section 6.4.

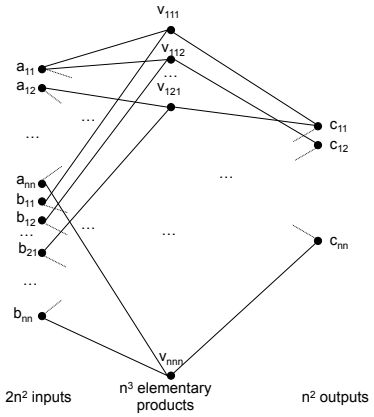


Figure 6.2: Inputs, intermediate products, and outputs of conventional algorithms. Intermediate products are equivalent to results of the join part of the matrix multiplication query. Outputs are equivalent to the final results of the query.

6.1 Lower-bounds for Conventional Matrix Multiplication

Consider the multiplication of two dense $n \times n$ square matrices A and B to compute the $n \times n$ output matrix C. Informally, we define conventional matrix multiplication algorithms as the ones that perform all of the n^3 elementary products $a_{ij}b_{jk} = v_{ijk}$ for $1 \leq i, j, k \leq n$, then sum these products to compute $c_{ik} = \Sigma_{j=1}^{n} v_{ijk}$. This definition rules out *Strassen-like* algorithms which do not perform all of the elementary products. We restrict our discussion to conventional algorithms because they correspond precisely to how the matrix multiplication query would be evaluated in existing relational systems. Figure 6.2 visually illustrates the circuit DAG (see Chapter 2) corresponding to matrix multiplication; there is one node for each input, one for each intermediate products, and one for each output, while edges represent their dependencies. Its depth is 2, its work is $n^3 + 3n^2$, and its fan-in is n (because the nodes c_{ik} have n children). Each intermediate product vertex v_{ijk}, and each output vertex c_{ik} represents an operation, such as a multiplication (or join) and summation (or group-by-and-aggregate) operation that the algorithm has to perform. This circuit is also called *computational directed acyclic graphs* in several previous work [14].

6.1.1 Round-independent Communication Lower Bound

We start with the lower bound on communication. Similar to the round-independent communication lower bound for sorting, the lower bound is an adaptation of an I/O complexity of matrix multiplication from references [47, 48] in the external memory model. Again similar to the lower bound argument for sorting we assume that at each moment, any processor can hold at most L numbers from the input but we allow the communications to be either bulk-synchronous or asynchronous, and the processors may perform arbitrary computations. We begin by a key lemma.

Lemma 6.1. [47, 48] Consider a conventional matrix multiplication algorithm for $AB = C$ of two $n \times n$ matrices A and B. Suppose that a processor accesses N_A elements of A, N_B elements of B, and outputs N_C elements of C. The the processor computes at most $(N_A N_B N_C)^{1/2}$ elementary products v_{ijk}.

Proof. We use the AGM bound presented inTheorem 4.5. Let A' denote the set of pairs of indices (i, j) such that the processor accesses the element A_{ij}; we think of A' as a binary relation, $A'(i, j)$, of size N_A. Similarly, let $B'(j, k)$ and $C'(i, k)$ be represent the set of pairs (j, k) and (i, k) such that the processor accesses B_{jk}, and computes C_{ik} respectively; their sizes are N_B, N_C respectively. Then, the set of indices (i, j, k) corresponding to the products v_{ijk} computed by the processor is given by the query $Q(i, j, k) = A'(i, j) \bowtie B'(j, k) \bowtie C'(i, k)$. By the AGM bound, its size is at most $(N_A N_B N_C)^{1/2}$, proving the claim[1]. □

Using Lemma 6.1, we can now prove a lower bound on the communication a single processor has to do in order perform W elementary products. This is a direct adaptation of an I/O lower bound. Recall that we assume that each processor holds at most L items.

Theorem 6.2. [47, 48] A processor that can store at most L input elements and computes W elementary products in the multiplication of two $n \times n$ matrices needs to perform at least $\frac{\mathsf{W}}{2\sqrt{2}\sqrt{L}} - L$ communication.

[1] The original proof of the lemma presented in reference [47] refers to the discrete Loomis-Whitney theorem [62] instead of the AGM bound. We note that the AGM bound is a generalization of Loomis-Whitney's theorem.

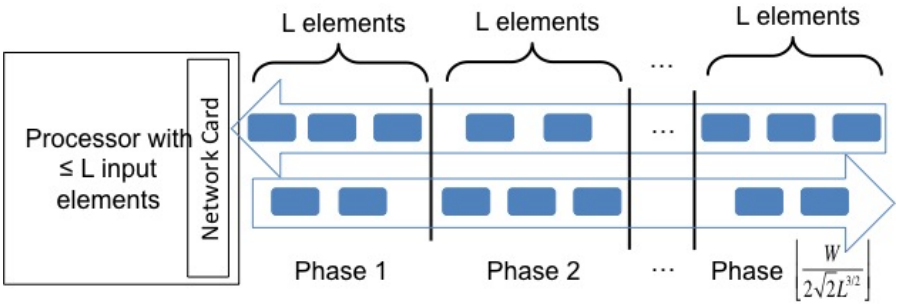


Figure 6.3: Phase break-down of the network I/O operations of a processor.

Proof. We break the computation into phases where phase t begins when the number of elements the processor reads from or writes to the network (i.e., sends as a message to another processor) is exactly tL. We assume each network I/O operation reads or writes one matrix element. Figure 6.3 shows the breakdown of computation into phases. Let N_A and N_B be the number of elements of matrix A that the processor accesses in a phase and let N_C be the number of elements of C that the processor contributes to. Note that N_A and N_B are at most $2L$ because each element of A and B must either be in one of the at most L elements in the processor's memory before a phase starts, or among the at most L elements read from the network. Similarly, N_C is less than $2L$ because each element that the processor contributes to must reside in the processor's memory at the end of the phase or be among the at most L elements written to the network. Therefore by Lemma 6.1 the maximum number of elementary products that the processor can perform in a single phase is at most $2\sqrt{2}L^{3/2}$. Therefore the number of phases is at least $\lfloor \frac{\mathsf{W}}{2\sqrt{2}\sqrt{L^{3/2}}} \rfloor$. Since the amount of network I/O in each phase is exactly L, the total communication for the processor is at least $(\lfloor \frac{\mathsf{W}}{2\sqrt{2}\sqrt{L^{3/2}}} \rfloor)L$ which is at least $\frac{\mathsf{W}}{2\sqrt{2}\sqrt{L}} - L$. □

Notice that Lemma 6.2 can equivalently be stated as saying that if a processor that can hold at most L input tuples is evaluating the triangle join and is outputting W triangles, then it has to perform $\frac{\mathsf{W}}{2\sqrt{2}\sqrt{L}} - L$ network I/O operations.

Theorem 6.3. Consider any distributed-memory system consisting of p processors each with limited memory of size L. The minimum amount of communication that any conventional algorithm has to perform for multiplying two dense $n \times n$ matrices is $\frac{n^3}{2\sqrt{2}\sqrt{L}} - pL$.

Proof. Let $\mathsf{W_m}$ be the number of elementary products performed by processor m. Then by Theorem 6.2, the minimum amount of communication that processor m has to perform is $\frac{\mathsf{W_p}}{2\sqrt{2}\sqrt{L}} - L$. Since $\sum_{m=1}^{p} W_m \geq n^3$, the minimum communication done by the system is $\sum_{m=1}^{p} (\frac{\mathsf{W_m}}{2\sqrt{2}\sqrt{L}} - L) \geq \frac{n^3}{2\sqrt{2}\sqrt{L}} - pL$. □

We note that although Theorem 6.3 is useful in the context of matrix multiplication, we can't directly obtain a similar corollary for triangle join in the general setting, as we do not know the number of output triangles on arbitrary database instances. In dense matrix multiplication we know that there are exactly n^3 products, which makes the theorem possible. We emphasize, for bulk synchronous systems, this result holds for algorithms running any number of rounds. We also note that for many practical values of p, n and L, this communication cost is $O(\frac{n^3}{\sqrt{L}})$. However as p grows very large, the bound goes to 0. This is merely a consequence of assuming that the processors have some initial data in the beginning. In other words, the $-pL$ is just the savings in communication due to the initial distribution of the data.

6.1.2 One-round Communication Lower-bound

Theorem 6.3 gives us a communication lower-bound that applies to any MPC algorithm, irrespective of the number of rounds that an algorithm executes. However, when computations are constrained to a single round, one can prove a higher lower bound by a counting argument involving not the number of elementary products but the output elements computed by processors. We next present a modification of a result from [2], which makes this argument.

Theorem 6.4. [2] The minimum amount of communication that any one-round conventional MPC algorithm multiplying two $n \times n$ matrices

using machines with limited memory L is $\frac{4n^4}{L} - pL$.

Proof. Consider processor m in the system. Processor m initially has at most L input elements and suppose it receives $L_m \leq L$ elements during the entire computation. Let us calculate the maximum number of output elements c_{ik} that the processor can compute. To produce c_{ik}, the processor needs the entire row i of matrix A and column k of matrix B. Moreover, note that if the processor outputs c_{ik} and $c_{i'k'}$, then it can also output $c_{i'k}$ and $c_{ik'}$. As a result, in order to maximize the number of output elements produced, the processor must receive a "rectangle" of rows and columns, i.e., a set of w rows, not necessarily consecutive, and h columns and output wh output elements. Since both rows and columns are of size n, processor m can maximize its output if $w = h = \frac{L+L_m}{2n}$. Therefore the maximum number of elements processor m can output is $\frac{(L+L_m)^2}{4n^2}$. Since there are n^2 outputs, $\sum_{m=1}^{p} \frac{(L+L_m)^2}{4n^2} \geq n^2$. Dividing both sides by L and using the fact that $L_m \leq L$, we can now derive that $C = \sum_{m=1}^{p} L_m \geq \frac{4n^4}{L} - pL$. □

Similar to Theorem 6.3, if we assume that the memories of the machines are empty in the beginning, we can get rid of the pL term in Theorem 6.4. We observe that Theorem 6.4 depends on both the join and the group-by-and-aggregate parts of the matrix multiplication query. Specifically, we used the fact that a processor needs an entire row and an entire column to perform the necessary products and aggregations to produce an output element. This contrasts with Theorem 6.3, which depended only on the join part of the query.

6.1.3 Round Lower-bounds

The round lower-bounds for conventional dense matrix multiplication algorithms were shown in reference [70] in the context of the MapReduce system. We recover their results using a simple extension of the round-independent communication lower-bound of Theorem 6.3. These lower bounds have two components expressed in the form of max{LB for products, LB for summations}. In the relational query terms, we can think of the round lower bounds similarly

as max{LB for join, LB for group-by-and-aggregate}. The first component lower bounds the number of rounds required to perform the n^3 elementary products in the computation. This subcomputation is represented in Figure 6.2 by the subgraph consisting of vertices a_{ij} and b_{jk} and the intermediate vertices v_{ijk}. The second component lower bounds the number of rounds needed for summing the v_{ijk} to produce the output elements c_{ik}. This subcomputation is represented in Figure 6.2 by the subgraph consisting of v_{ijk} vertices and the c_{ik} vertices. For almost all practical settings, we expect the the first component to be relevant because, as we will see, the summations can usually be done in a constant number of rounds. We begin by lower bounding the rounds that are needed to perform the summation operations.

Theorem 6.5. [40] The minimum number of rounds needed by any MPC algorithm summing n elements to compute any c_{ik} using processors with L memory is $O(\log_L(n))$. Therefore, any conventional dense matrix multiplication algorithm requires $O(\log_L(n))$ rounds to perform the summations in the computation.

Similar to Theorem 5.2, this result fundamentally follows from the fact that functions that depend on all of their inputs require $O(log_L(n))$ rounds. We note that in most modern practical settings the memory of machines will easily be more than n, which is asymptotically the square root of the input size; therefore the summations can in general be done in a very small number of rounds.

We next show the number of rounds needed to perform the elementary multiplications with a very simple argument extending our round-independent communication lower bound. The same result is shown in reference [70] using a different proof.

Theorem 6.6. Consider any conventional MPC dense matrix multiplication algorithm using machines with limited memory L. The minimum number of rounds needed by the algorithm to perform the n^3 elementary products is $O(\frac{n^3}{pL\sqrt{L}})$.

Proof. By Theorem 6.3 the minimum amount of communication needed by the algorithm is $\frac{n^3}{2\sqrt{2}\sqrt{L}} - pL$. Since in each round the maximum

amount of communication that the algorithm can perform is pL, the number of rounds needed is $\frac{n^3}{2\sqrt{2}pL\sqrt{L}} - 1$, which is $O(\frac{n^3}{pL\sqrt{L}})$. □

An interesting observation of Theorem 6.6 is that the number of rounds for dense matrix multiplication depends on both the number of machines, p and the per-machine memory sizes L. In contrast, the amount of communication depends only on the per-machine sizes L for most practical settings and certainly when we assume that machines have clean memories in the beginning of the computation. Notice that this is different from the lower bounds on sorting where both rounds and communication depended only on L (Chapter 5).

Combining Theorems 6.5 and 6.6, we can derive the round lower bound for the number of rounds needed by conventional bulk synchronous algorithms:

Corollary 6.7. [70] The minimum number of rounds needed by any conventional MPC dense matrix multiplication algorithm using p machines with limited memory L is $O(\max\{\frac{n^3}{pL\sqrt{L}}, \log_L(n)\})$.

6.2 Algorithms

In this section we discuss two types of parallel matrix multiplication algorithms: a single round algorithm in Section 6.2.1, and a multi-round algorithm in Section 6.2.2. Both match the lower bounds from Section 6.1 for a wide range of values of L and p. The one-round algorithm has its roots in reference [64] and has been adapted to a model of MapReduce in reference [2]. The multi-round algorithm is from reference [63], and is a generalization of the well-studied *two-dimensional* (2D) and *three-dimensional* (3D) matrix multiplication algorithms. Examples of 2D algorithms have been presented in references [5, 23, 32] and examples of 3D algorithms have been presented in references [5, 6, 32, 51]. We describe informally how 2D and 3D algorithms are special cases of the more general multi-round algorithm. The version of the multi-round algorithm we review has also been adapted to a model of MapReduce in reference [70].

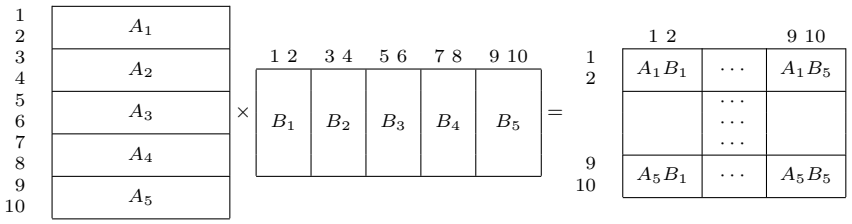


Figure 6.4: Computing the product $A \times B = C$ in one round. The matrices are 10×10 and $t = 2$.

6.2.1 Optimal One-Round Matrix Multiplication Algorithm

We start by describing a one-round parallel matrix multiplication algorithm. We first note that, if $L \geq 2n^2$, then one can perform the entire multiplication in a single machine in one round, because both input matrices fit on one processor. On the other hand, if $L < 2n$ then no one-round algorithm can get enough input elements to produce any of the outputs of matrix C, because, in order to produce c_{ik}, a processor needs to access $2n$ elements a_{ij}, b_{jk}, $k = 1, n$. The one-round algorithm described here works for values of L between $2n$ and $2n^2$. Assume both $L/2n$ and $2n^2/L$ are integers, and denote $t = L/2n$. The algorithm partitions the $n \times n$ matrix A horizontally into n/t matrices of sizes $t \times n$, and similarly partitions the matrix B vertically into n/t matrices of sizes $n \times t$. In other words, A is a block matrix of row matrices $A_1, \ldots, A_{n/t}$, while B is a block matrix of column 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 $u, v \in \{1, \ldots, n/t\}$. Figure 6.4 shows an example of multiplying two 10×10 matrices, using $t = 2$: since each of A and B is a block matrix with 5 blocks, their product C is a block matrix with 25 blocks.

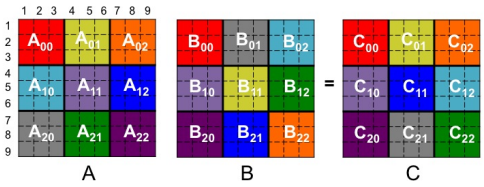
The algorithm proceeds as follows. It uses $p \stackrel{\text{def}}{=} n^2/t^2 = 4n^4/L^2$ processors. Each processor is identified by a pair (u, v), with $u, v \in \{1, \ldots, n/t\}$, and is responsible for computing the block matrix $A_u B_v$. Initially, each processor reads its inputs and sends each matrix element a_{ij} that is in the block A_u to all processors (u, v), with $v \in \{1, \ldots, n/t\}$;

similarly, it sends each element $_{jk} \in B_v$ to all processors (u, v), for $u \in \{1, \ldots, n/t\}$. After the communication round, each processor (u, v) multiplies locally the matrices A_u, B_v, multiplies and outputs the result $A_u B_v$. Since the size of A_u is tn, and similarly for B_v, the total load is $L = 2tn$. Each input of A and B is replicated n/t times, and therefore the total communication cost is equal to $2n^2(n/t) = 2n^3/t = 4n^4/L$, matching the lower bound from Theorem 6.4 exactly.

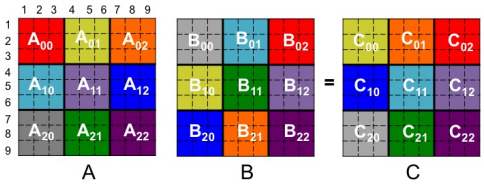
6.2.2 Optimal Multi-Round Matrix Multiplication Algorithm

Next, we describe a multi-round parallel matrix multiplication algorithm. Unlike the one-round algorithm, here we allow the load L to decrease below $2n$, as illustrated by the lower curve in Figure 6.1. While the one round algorithm partitioned A and B into row- and column-matrices respectively, the multi-round algorithm partitions them into smaller, squared matrices A_{uh} and B_{hv}, then computes the block $C_{uv} \stackrel{\text{def}}{=} \sum_h A_{uh} B_{hv}$ of the output matrix using multiple rounds. We describe here the algorithm in its most general form, then show how to derive some popular algorithms, known as 2D and 3D matrix multiplication algorithms, as special cases.

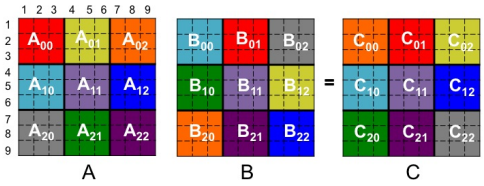
For simplicity of presentation, we will work with parameter $L^* \stackrel{\text{def}}{=} L/3$ instead of L, and denote $H \stackrel{\text{def}}{=} \frac{n}{\sqrt{L^*}}$, which we assume to be an integer. Throughout this section we will assume that the number of processors is $p \geq 3n^2/L$, thus $p \geq H^2$; this is w.l.o.g. because, collectively, the p processors must be able to store both the inputs A, B and the output C. We block partition A into $H \times H$ square matrices A_{uh} of size $\sqrt{L^*} \times \sqrt{L^*}$, where $u, h \in \{1, \ldots, H\}$; similarly, block partition B into matrices B_{hv} of size $\sqrt{L^*} \times \sqrt{L^*}$. The product $C = AB$ is a block matrix, where each block is $C_{uv} = \sum_{h=1,H} A_{uh} B_{hv}$, for $u, v \in \{1, \ldots, H\}$. The idea of the algorithm is to compute the H^3 matrix products $A_{uh} B_{hv}$ and then add them up. Since the memory of each of the p processors is $L = 3L^*$, during each round a processor can compute only a single product $A_{uh} B_{hv}$, and therefore we should expect the algorithm to use H^3/p rounds. We group the H^3 block multiplications into H groups $G_0, ..., G_{H-1}$, where each group G_ℓ consists of all products $A_{u,h} \times B_{h,v}$ where $h = (u + v + \ell) \mod H$. For an illustration, Figure 6.5 shows the



(a) Group 1 multiplications.



(b) Group 2 multiplications.



(c) Group 3 multiplications.

Figure 6.5: Grouping of the multiplications of the multiround algorithm.

groups for multiplying two 9×9 matrices: in each group, the blocks of A and B that have the same color are multiplied to give the block of C with the same color.

Since each group consists of H^2 matrix products, and $p \geq H^2$, we can compute, in one round, all the products in more than one group. To describe the algorithm, we start with the special case when $p = H^2$. In this case, at each round the p processors compute in parallel the products of a single group G_ℓ. The mod-based strategy of grouping guarantees that each block A_{uh}, B_{hv} and C_{uv} participates in exactly one multiplication in that group. Therefore, during each round, a processor computes one product $A_{uh}B_{hv}$, adds the result to the partial sum C_{uv}, then sends the matrices A_{uh}, B_{hv} and the partial result C_{uv} to the unique processors handling those blocks in the next round. The algorithm terminates in H rounds, after computing

all groups $G_0, G_1, \ldots, G_{H-1}$.

Now we describe the algorithm in the general case, when $p \geq H^2$. Then, during each round, we can compute in parallel the products in several groups, namely in $K = \min\{p/H^2, H\}$ groups $G_\ell, G_{\ell+1}, \ldots, G_{\ell+K-1}$. The only difference is that, now, the partial sums of each block C_{uv} are computed by K processors, and their results need to be summed up. We can postpone summing them up by keeping these partial sums separate until the end of the algorithm. In other words, for each u, v the processors compute K different matrices $C_{uv}^{(1)}, C_{uv}^{(2)}, \ldots, C_{uv}^{(K)}$, whose sum is C_{uv}; after each round these matrices are not added, but instead the processor handling $C_{uv}^{(h)}$ will send this matrix to the processor that handles it in the next round. After H/K rounds, the algorithm has $H \times H \times K$ block matrices $C_{uv}^{(h)}$, each stored on a different processor, and needs to add up the K copies of each block, $C_{uv} = \sum_{h=1,K} C_{uv}^{(h)}$. This can be done as follows. We show how the aggregation happens for a single block using only the K processors that store one u, v block; all other blocks are aggregated similarly and in parallel. If $L^* \geq K$, then we can partition each block into K *pieces* of size L^*/K, and in one round send each group of K pieces to a separate processor to perform the aggregation. If $L^* < K$, then we use an aggregation tree. We divide the K processors into K/L^* groups of L^* processors. The L^* processors in each group need to sum L^*-size blocks. Each processor is responsible for one cell of these L^* blocks in one round and receives L^* copies of this cell and aggregates in the first round. So at the end of the first round, each processor has one value (instead of L^*). In a separate round the L^* processors in each group send their values to one of the processors amongst them. So at the end of the second round one processor from each group has an L^*-size block and we remove the other $L^* - 1$ from the computation. So we are left with K/L^* processors each holding a L^*-size blocks and we repeat the process. After $\log_{L^*}(K)$ rounds, we aggregate all of the K copies of each block.

We compare the performance of the algorithm with the lower bounds given by Corollary 6.7 and Theorem 6.3. For the number of rounds, note that the algorithm takes H/K rounds to perform

the products. If $K = H$, this is 1, if $K = p/H^2$, this is equal to $H^3/p = O\left(\frac{n^3}{pL\sqrt{L}}\right)$. The aggregation takes either 1 or $\log_{L^*}(K)$ rounds. Note that K is at most $H = n/\sqrt{L}$, so the aggregation takes at most $O(\log_L(n))$ rounds. Therefore the number of rounds of the algorithm is:

$$r = O\left(\frac{n^3}{pL\sqrt{L}} + \log_L n\right) \tag{6.1}$$

Note that this matches Corollary 6.7.

We analyze the amount of communication the algorithm takes as follows. Performing the multiplications take $O\left(\frac{n^3}{pL\sqrt{L}}\right)$ rounds and in each round we incur $O(pL)$ communication. So the communication cost the algorithm incurs to perform the multiplications is $O\left(\frac{n^3}{\sqrt{L}}\right)$. If $L^* \geq K$, aggregating one of the L^*-size blocks takes 1 round, so takes KL^* communication cost. There are H^2 such blocks, so if $L^* \geq K$, the communication cost of aggregation is $O\left(K \times L \times H^2\right)$ which is at most $O\left(L \times H^3\right) = O\left(\frac{n^3}{\sqrt{L}}\right)$. If $L^* < K$, now aggregating one block takes $\log_{L^*}(K)$ rounds, but notice that the amount of communication in every two rounds decrease by a factor of L^*, so the total communication is dominated by the communication in the first round, which is $K \times L^*$. Similarly, there are H^2 blocks, so the communication is again $O\left(K \times L \times H^2\right) = O\left(\frac{n^3}{\sqrt{L}}\right)$. Therefore the communication cost of the algorithm is:

$$C = O(n^3/\sqrt{L}) \tag{6.2}$$

Note that this asymptotically matches Theorem 6.3 if we ignore the $-pL$ term in the theorem, which as we explained is due to assuming that the processors have some initial data in the beginning[2].

We finish this section with a short discussion of 2D and 3D matrix multiplication algorithms [5, 6, 32, 47, 51], explaining why they are two special instances of the multi-round algorithm we reviewed. The 2D algorithms correspond to the special case we discussed above when the number of processors p is minimum, equivalently $L = \Theta(\frac{n^2}{p})$. Note that in this case the total memory of all processors is $\Theta(n^2)$, thus the data

[2]In absence of this assumption the -pL term goes away from Theorem 6.3.

cannot be replicated. Recall that in this case the algorithm computes one group G_ℓ at each round. Using Equations (6.1) and (6.2) we obtain that the number of rounds of a 2D algorithm is $r = O\left(\sqrt{p} + \log_{n^2/p} p\right)$, and the total communication cost is $C = O\left(n^2\sqrt{p}\right)$. The 3D algorithms allow for more processors p or, equivalently, for more local memory L, in order to be able to compute all groups $G_0, G_1, \ldots$ in a constant number of rounds. For that, a 3D algorithm sets $L = \Theta(\frac{n^2}{p^{2/3}})$, thus, the total amount of memory used by all processors is $O(p^{1/3}n^2)$. Using again Equations (6.1) and (6.2) we obtain the number of rounds $r = O\left(1 + \log_{n^2/p^{2/3}}(n)\right)$ and the total communication cost $C = O(p^{1/3}n^2)$. We omit several details of the 2D and 3D algorithm, such as their communication pattern, and refer the reader to [5, 6, 32, 47, 51].

6.3 Discussion

The complexity of conventional matrix multiplication algorithms is fully captured by the results presented in this chapter. Theorem 6.3 shows that the total amount of communication is entirely determined by L, and is independent of the number of rounds and (to some extent) on the number of processors; Theorem 6.4 strengthens this lower bound in the case of one-round algorithms. Corollary 6.7 shows that, the benefit of having more processors p is that it reduces the number of rounds of computation, r, roughly as $r \sim 1/p$. All these lower bounds are asymptotically tight, as they correspond to upper bounds of the one- and multi-round algorithms we presented in Section 6.2.

Figure 6.1 depicts the lower bounds we reviewed partially. In the figure, x-axis is L. The red and black lines show the 1-round and round-independent communication lower bounds, respectively. The dots on the black line depict the lower bound on the number of rounds needed to multiply two matrices. We note that the figure captures the relationship between L and communication but does not capture the effects of p on the lower bounds. For example, the dot on the black line corresponding to $L = n^{1/3}$ says 3, indicating we need at least 3 rounds to multiply matrices with load $n^{1/3}$. This partially captures the the $\log_L(n)$ component of Corollary 6.7 that lower bounds the number of

rounds needed to perform the summation of the elementary products (Theorem 6.5). However it does not capture the $\frac{n^3}{pL\sqrt{L}}$ component of Corollary 6.7 that lower bounds the number of rounds needed to perform the elementary products (Theorem 6.6), as this depends on p.

6.4 Other Linear-Algebra Computations

Many other results exist on the communication and round complexities of other linear algebra operations. We briefly review several of these results.

Conventional Algorithms for Non-square Dense Matrix Multiplication: The I/O complexity lower bound results from reference [47] that we adapted to get the round-independent communication lower bounds were originally presented in the same reference in the more general form applying to non-square dense matrix multiplication. For general matrices A with dimensions $m \times n$ and B with dimensions $n \times r$, replacing the term n^3 with mnr in the communication lower bounds from Theorem 6.2 and Theorem 6.3 gives the results from reference [47]. Reference [47] has shown that existing 2D and 3D algorithms are optimal for certain L values and reference [33] has shown an algorithm that is optimal for general L values.

Conventional Algorithms for Sparse Matrix Multiplication: References [13] and [70] have independently shown that the lower bound arguments for dense matrix multiplication from reference [47] can be extended to achieve I/O and communication complexity lower bounds for sparse matrix multiplication. These results apply to multiplication of matrices in any dimensions. If N is the maximum of the number of non-zero elements in matrices A and B and T is number of elementary multiplications that need to happen in the multiplication, the I/O and communication lower bounds from Theorem 6.2 and Theorem 6.3 can be achieved by replacing n^3 with T. For example, the communication lower bound of any algorithm is $\frac{T}{\sqrt{L}}$. If T is not known, one can show that in the worst case there are $N\sqrt{n}$ multiplication that need to happen, so the communication lower bound is $\frac{N\sqrt{n}}{\sqrt{L}}$, although tighter results depending on the exact values of $\sqrt{n}$ and N and the

total number of output elements in C are also provided. Finally, reference [70] has shown round lower bounds for sparse matrix multiplications. Reference [70] also presents some new algorithms as well as an existing algorithm from reference [22] that are optimal or almost optimal in terms of communication and round complexities for sparse matrix multiplication.

Conventional Algorithms for Multiplying Sparse Random Matrices: Reference [11] has improved the worst-case lower bound for the communication cost of sparse matrix multiplication from references [13] and [70] for matrices representing Erdős Rényi random graphs. In these matrices, each entry of the input matrices A and B is set uniformly at random to 0 or 1 with fixed probabilities. The authors also present matching algorithms.

Conventional Algorithms for Other Computations: Reference [13] have proved I/O and communication lower bounds for the following basic linear algebra operations: LU, Cholesky, LDL^T, QR factorization, computing eigenvalues, and computing singular values. For dense versions of these problems they show that several existing algorithms are optimal in terms of their communications. Same reference also studies linear algebra computations that are composed of a sequence of basic operations, such as sparse or dense matrix multiplication, LU, or QR factorizations. Examples include, taking the power of a single matrix and multiplication of a matrix A with the power of a matrix B. The authors show examples when the sum of the lower bound of each basic operations in the sequence is a correct lower bound for the entire sequence and when it is not. Finally, the authors show applications of their linear algebra lower bounds and algorithms to the Floyd-Warshall algorithm for computing all pairs shortest paths problem. The Floyd-Warshall algorithm recursively performs a multiplication on the adjacency matrix of an input graph.

Results for Strassen-like Fast Matrix Multiplication: Reference [14] have shown lower-bounds on the I/O and communication costs of Strassen's and other fast algorithms for multiplying two square matrices. Unlike conventional algorithms these algorithms do not perform all of the elementary products in the computation. Informally,

the authors define the notion of *Strassen-like* algorithms as those that use a base algorithm that multiply two constant-size square matrices to recursively multiply two arbitrary size square matrices. Depending on the exact base algorithm, the total number of operations that the algorithms perform can be expressed as $\Theta(n^{w_0})$ where $w_0 < 3$. For example, they prove that the lower bound on the I/O for a class of such algorithms is $\Omega((\frac{n}{\sqrt{L}})^{w_0} L)$. Observe that for a recursive conventional algorithm that performs all of the elementary products, i.e., when $w_0 = 3$, this lower bound is the same as Theorem 6.2. They also show that algorithms from references [63] and [10] are optimal in terms of their I/O and communication bounds. In another paper, the authors extend their results to fast multiplication of non-square matrices [12].

7

Conclusion

In this survey, we presented new developments as well as older results on the algorithmic design and theoretical analysis of massively parallel algorithms for data processing. We introduced a formal theoretical model, called MPC, to analyze the performance of algorithms in massively parallel settings in terms of communication and synchronization (number of rounds). We focused on three fundamental data processing tasks: *join processing, sorting*, and *matrix multiplication*. For all these three core tasks, we presented both algorithms, as well as lower bounds, and discussed the tradeoffs between communication and number of rounds.

We conclude by briefly mentioning some open problems related to massively parallel data processing. As we have seen in the previous chapters, sorting and matrix multiplication are in general well-understood, but join processing less so.

Open Problems in Matrix Multiplication In Chapter 6, we discussed algorithms for matrix multiplication, but we focused on *conventional algorithms*, where all n^3 elementary multiplications must be performed. Some lower bounds for non-conventional parallel algorithms are known

(e.g. for Strassen-like algorithms), but the general question of figuring out the optimal communication complexity remains open. A intriguing question is whether Strassen-like algorithms can be applied somehow to 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.

Open Problems in Join Processing In join processing, many theoretical questions still remain open, especially for multi-round algorithms Here we mention a few of these open problems:

- **A worst-case optimal constant round join algorithm:** The Hypercube algorithm is optimal only for skew-free data. When the input data is arbitrary (in other words, it may be skewed) then we only know optimal algorithms in a special case (when all relations are binary). It is open how to design an optimal algorithm for arbitrary queries and data. All we know is that it must run in mulitiple rounds. We have seen that there exists queries where $\rho^* < \tau^*$, and in that case, such an optimal algorithm must resolve somehow our current gap between the one-round upper bound $O(\mathsf{IN}/p^{1/\tau^*})$ for skew-free data and the general multi-round lower bound $\Omega(\mathsf{IN}/p^{1/\rho^*})$ for arbitrary data: it will either compute the query on skew-free data with load better than $O(\mathsf{IN}/p^{1/\tau^*})$ (necessarily using multiple rounds), or one must provide a stronger lower bound than $\Omega(\mathsf{IN}/p^{1/\rho^*})$.

- **Output-sensitive algorithms** Design algorithms that are optimal for given input and output sizes, $\mathsf{IN}, \mathsf{OUT}$. We have seen such an example for a join and (to a lesser degree) for a triangle query. It would be interesting to see if the novel technique that uses sorting instead of hashing [45] can lead to optimal output-sensitive algorithm for general multi-join queries.

- **Multi-round lower bounds for skew-free data** If we restrict the algorithm to one round of communication, then we have a very powerful lower bound on the load, $L = \Omega(N/p^{1/\tau^*})$ (where

$N = \mathsf{IN}$); this lower bound holds even if the output is small, hence it truly captures the communication complexity of the query. By contrast, if we allow multiple rounds, the only way we know how to prove lower bounds is to use input data where the output very large. It remains open how to prove lower bounds when the output is small. The simplest example is the 5-chain query $R(x, y) \bowtie S(y, z) \bowtie T(z, u) \bowtie K(u, v) \bowtie L(v, w)$, where each of the five input relations is a permutation on $[N]$, and therefore the size of the output is exactly N: in this case prove that no two-round[1] algorithm can compute the query using a load $L = O(N/p)$.

- **Improved used of statistics** Reference [50] describes an algorithm that uses the degrees of the elements in the input relations to design better algorithms. Statistics such as degrees, number of distinct values, or histograms, are often available in big data processing, or can be computed on-the-fly using sampling, or even using an exact algorithm. Certain constraints, such as keys or functional dependencies, are special cases of such statistics; for example a key constraint corresponds to stating that the maximum degree of that attribute is 1. It is open how to design algorithms that are provably optimal for a given set of data statistics.

- **Beyond Joins** In this survey we only discussed *full conjunctive queries*, or natural joins. Most SQL queries in practice involve some kind of grouping and aggregates. In sequential query processing, such queries are handled using sophisticated tree decomposition methods [55], but the parallel complexity of such queries has not been explored yet.

Open Problems in Graph Processing Apart from the three core problems we discussed in this survey, there is a multitude of other funda-

[1]It is easy to compute it in three rounds: round one computes $A(x, y, z) = R(x, y) \bowtie S(y, z)$, round two computes (in parallel) $B(x, y, z, u) = A(x, y, z) \bowtie T(z, u)$ and $C(u, v, w) = K(u, v) \bowtie L(v, w)$, while round three computes the final output $B \bowtie C$. Since each intermediate relation has size N and no skew, each round requires a load $L = N/p$.

mental tasks, in particular core tasks related to graph processing. A graph problem that has recently seen a lot of interest is the problem of *connected components*, where given a graph G, we seek to group the vertices into connected components. Other problems include computing the transitive closure of the graph, computing the diameter or radius, and computing shortest paths, to name just a few. Although there are many parallel algorithms for graph processing tasks, they often lack a formal analysis in terms of the load and communication.

Out-of-core Procesing There seems to be deep connections between parallel processing and out-of-core processing, which is captured by the External Memory model we described in Chapter 2. This connection was mentioned both in [57] in the context of join processing, as well as in [47] in the context of matrix multiplication. We already know how to translate an MPC algorithm to an External Memory algorithm, but the open question is whether this translation will obtain an optimal algorithm for the External Memory model.

References

[1] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. EmptyHeaded: A Relational Engine for Graph Processing. In *SIGMOD*, 2016.

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

[3] Foto N. Afrati, Manas R. Joglekar, Christopher Ré, Semih Salihoglu, and Jeffrey D. Ullman. GYM: A Multiround Distributed Join Algorithm. In *ICDT*, 2017.

[4] Foto N. Afrati and Jeffrey D. Ullman. Optimizing multiway joins in a map-reduce environment. *IEEE Transactions on Knowledge and Data Engineering*, 23(9), 2011.

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

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

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

[8] Miklós Ajtai, János Komlós, and Endre Szemerédi. Sorting in $c \log n$ Parallel Sets. *Combinatorica*, 3(1), 1983.

[9] Albert Atserias, Martin Grohe, and Dániel Marx. Size Bounds and Query Plans for Relational Joins. *SIAM Journal on Computing*, 42(4), 2013.

[10] G. Ballard, J. Demmel, O. Holtz, B. Lipshitz, and O. Schwartz. Strong Scaling of Matrix Multiplication Algorithms and Memory-Independent Communication Lower Bounds. Technical report, EECS Department, University of California, Berkeley, March 2012.

[11] Grey Ballard, Aydin Buluç, James Demmel, Laura Grigori, Benjamin Lipshitz, Oded Schwartz, and Sivan Toledo. Communication Optimal Parallel Multiplication of Sparse Random Matrices. In *SPAA*, 2013.

[12] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Graph Expansion Analysis for Communication Costs of Fast Rectangular Matrix Multiplication. In *MedAlg*, 2012.

[13] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing Communication in Numerical Linear Algebra. *SIAM Journal of Matrix Analysis Applications*, 32(3), 2011.

[14] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Graph Expansion and Communication Costs of Fast Matrix Multiplication. *Journal of the ACM*, 59(6), 2013.

[15] Pablo Barceló, Georg Gottlob, and Andreas Pieris. Semantic Acyclicity Under Constraints. In *PODS*, 2016.

[16] Kenneth E. Batcher. Sorting Networks and Their Applications. In *AFIPS*, 1968.

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

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

[19] Paul Beame, Paraschos Koutris, and Dan Suciu. Communication Cost in Parallel Query Processing. *CoRR*, abs/1602.06236, 2016.

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

[21] A. Borodin and J. E. Hopcroft. Routing, Merging, and Sorting on Parallel Models of Computation. *Journal of Computer and System Sciences*, 30(1), 1985.

[22] Aydin Buluç and John R. Gilbert. Challenges and Advances in Parallel Sparse Matrix-Matrix Multiplication. In *ICPP*, 2008.

[23] Lynn Elliot Cannon. *A Cellular Computer to Implement the Kalman Filter Algorithm*. PhD thesis, Montana State University, 1969.

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

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

[26] Shumo Chu, Magdalena Balazinska, and Dan Suciu. From Theory to Practice: Efficient Join Query Evaluation in a Parallel Database System. In *SIGMOD*, 2015.

[27] Cole, Richard. Parallel Merge Sort. *SIAM Journal on Computing*, 17(4), 1988.

[28] Michael Conley, Amin Vahdat, and George Porter. TritonSort 2014. `http://sortbenchmark.org/TritonSort2014.pdf`.

[29] Stephen A. Cook, Cynthia Dwork, and Rüdiger Reischuk. Upper and Lower Time Bounds for Parallel Random Access Machines without Simultaneous Writes. *SIAM Journal on Computing*, 15(1), 1986.

[30] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice E. Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a Realistic Model of Parallel Computation. In *PPOPP*, 1993.

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

[32] Eliezer Dekel, David Nassimi, and Sartaj Sahni. Parallel Matrix and Graph Algorithms. *SIAM Journal on Computing*, 16(3), 1984.

[33] James Demmel, David Eliahu, Armando Fox, Shoaib Kamil, Benjamin Lipshitz, Oded Schwartz, and Omer Spillinger. Communication-Optimal Parallel Recursive Rectangular Matrix Multiplication. In *IPDPS*, 2013.

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

[35] EMC Corporation. 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`.

[36] Jon Feldman, S. Muthukrishnan, Anastasios Sidiropoulos, Clifford Stein, and Zoya Svitkina. On Distributing Symmetric Streaming Computations. *ACM Transactions on Algorithms*, 6(4), 2010.

[37] Merrick Furst, James B. Saxe, and Michael Sipser. Parity, circuits, and the polynomial-time hierarchy. *Mathematical Systems Theory*, 17(1), 1984.

[38] Sumit Ganguly, Abraham Silberschatz, and Shalom Tsur. Parallel Bottom-Up Processing of Datalog Queries. *Journal of Logic Programming*, 14(1&2), 1992.

[39] Alan Gibbons and Wojciech Rytter. *Efficient Parallel Algorithms*. Cambridge University Press, 1988.

[40] Michael T. Goodrich. Communication-Efficient Parallel Sorting. *SIAM Journal on Computing*, 29(2), 1999.

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

[42] Georg Gottlob, Gianluigi Greco, Nicola Leone, and Francesco Scarcello. Hypertree Decompositions: Questions and Answers. In *PODS*, 2016.

[43] Martin Grohe and Dániel Marx. Constraint Solving via Fractional Edge Covers. *ACM Transactions on Algorithms*, 11(1), 2014.

[44] D. Halperin, V. Teixeira de Almeida, L. Choo, S. Chu, P. Koutris, D. Moritz, J. Ortiz, V. Ruamviboonsuk, J. Wang, A. Whitaker, S. Xu, M. Balazinska, B. Howe, and D. Suciu. Demonstration of the Myria Big Data Management Service. In *SIGMOD*, 2014.

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

[46] M. Husain, J. McGlothlin, M. M. Masud, L. Khan, and B. M. Thuraisingham. Heuristics-Based Query Processing for Large RDF Graphs Using Cloud Computing. *IEEE Transactions on Knowledge and Data Engineering*, 23(9), 2011.

[47] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication Lower Bounds for Distributed-memory Matrix Multiplication. *Journal of Parallel and Distributed Computing*, 64(9), 2004.

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

[49] Jie Jiang, Lixiong Zheng, Junfeng Pu, Xiong Cheng, Chongqing Zhao, Mark R. Nutter, and Jeremy D. Schaub. Tencent Sort. `http://sortbenchmark.org/TencentSort2016.pdf`.

[50] Manas Joglekar and Christopher Ré. It's All a Matter of Degree: Using Degree Information to Optimize Multiway Joins. In *ICDT*, 2016.

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

[52] Stasys Jukna. *Boolean Function Complexity - Advances and Frontiers*, volume 27 of *Algorithms and Combinatorics*. Springer, 2012.

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

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

[55] Mahmoud Abo Khamis, Hung Q. Ngo, and Atri Rudra. FAQ: Questions Asked Frequently. In *PODS*, 2016.

[56] Marcel Kornacker, Alexander Behm, Victor Bittorf, Taras Bobrovytsky, Alan Choi, Justin Erickson, Martin Grund, Daniel Hecht, Matthew Jacobs, Ishaan Joshi, Lenni Kuff, Dileep Kumar, Alex Leblang, Nong Li, Henry Robinson, David Rorke, Silvius Rus, John Russell, Dimitris Tsirogiannis, Skye Wanderman-milne, and Michael Yoder. Impala: A Modern, Open-Source SQL Engine for Hadoop. In *CIDR*, 2015.

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

[58] Eyal Kushilevitz and Noam Nisan. *Communication Complexity*. Cambridge University Press, 1997.

[59] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. Scalable subgraph enumeration in mapreduce: A cost-oriented approach. *The VLDB Journal*, 26(3), 2017.

[60] Leonid Libkin. *Elements of Finite Model Theory*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[61] Longbin Lai and Lu Qin and Xuemin Lin and Ying Zhang and Lijun Chang. Scalable distributed subgraph enumeration. *PVLDB*, 10(3), 2016.

[62] L. H. Loomis and H. Whitney. An Inequality Related to the Isoperimetric Inequality. *Bulletin of the American Mathematical Society*, 55(10), 1949.

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

[64] A. C. McKellar and E. G. Coffman, Jr. Organizing Matrices and Matrix Operations for Paged Memory Systems. *Communications of the ACM*, 12(3), 1969.

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

[66] Rajeev Motwani and Prabhakar Raghavan. *Randomized Algorithms*. Cambridge University Press, 1995.

[67] Thomas Neumann and Gerhard Weikum. The RDF-3X Engine for Scalable Management of RDF Data. *VLDB Journal*, 19(1), 2010.

[68] H. Ngo, C. Ré, and A. Rudra. Skew Strikes Back: New Developments in the Theory of Join Algorithms. *SIGMOD Record*, 42(4), 2014.

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

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

[71] Raghu Ramakrishnan and Johannes Gehrke. *Database management systems (3rd edition)*. McGraw-Hill, 2003.

[72] Raghu Ramakrishnan, Baskar Sridharan, John R. Douceur, Pavan Kasturi, Balaji Krishnamachari-Sampath, Karthick Krishnamoorthy, Peng Li, Mitica Manu, Spiro Michaylov, Rogério Ramos, Neil Sharman, Zee Xu, Youssef Barakat, Chris Douglas, Richard Draves, Shrikant S. Naidu, Shankar Shastry, Atul Sikaria, Simon Sun, and Ramarathnam Venkatesan. Azure Data Lake Store: A Hyperscale Distributed File Service for Big Data Analytics. In *SIGMOD*, 2017.

[73] Alexander Rasmussen, George Porter, Michael Conley, Harsha V. Madhyastha, Radhika Niranjan Mysore, Alexander Pucher, and Amin Vahdat. TritonSort: A Balanced Large-scale Sorting System. In *NSDI*, 2011.

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

[75] Hanmao Shi and Jonathan Schaeffer. Parallel Sorting by Regular Sampling. *Journal of Parallel and Distributed Computing*, 14(4), 1992.

[76] Sort Benchmark Home Page. `http://sortbenchmark.org/`.

[77] Spark SQL. `https://spark.apache.org/sql/`.

[78] SPARQL Query Language for RDF. `https://www.w3.org/TR/rdf-sparql-query/`.

[79] Dan Suciu and Val Tannen. A Query Language for NC. *Journal of Computer and System Sciences*, 55(2), 1997.

[80] Zhao Sun, Hongzhi Wang, Haixun Wang, Bin Shao, and Jianzhong Li. Efficient Subgraph Matching on Billion Node Graphs. *PVLDB*, 5(9), 2012.

[81] Siddharth Suri and Sergei Vassilvitskii. Counting triangles and the curse of the last reducer. In *WWW*, 2011.

[82] Graves Thomas. GraySort and MinuteSort at Yahoo on Hadoop 0.23. `http://sortbenchmark.org/Yahoo2013Sort.pdf`.

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

[84] Leslie G. Valiant. A Bridging Model for Parallel Computation. *Communications of the ACM*, August 1990.

[85] Todd L. Veldhuizen. Leapfrog Triejoin: A Simple, Worst-Case Optimal Join Algorithm. In *ICDT*, 2014.

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

[87] Jiamang Wang, Yongjun Wu, Hua Cai, Zhipeng Tang, Zhiqiang Lv, Bin Lu, Yangyu Tao, Chao Li, Jingren Zhou, and Hong Tang. FuxiSort. `http://sortbenchmark.org/FuxiSort2015.pdf`.

[88] 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. The Myria Big Data Management and Analytics System and Cloud Services. In *CIDR*, 2017.

[89] Reynold Xin, Parviz Deyhim, Ali Ghodsi, Xiangrui Meng, and Matei Zaharia. GraySort on Apache Spark by Databricks. `http://sortbenchmark.org/ApacheSpark2014.pdf`.

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

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

[92] Zeng, Kai and Yang, Jiacheng and Wang, Haixun and Shao, Bin and Wang, Zhongyuan. A Distributed Graph Engine for Web Scale RDF Data. *VLDB*, 6(4), 2013.