Topology-aware Parallel Joins*

XIAO HU, University of Waterloo, Canada
PARASCHOS KOUTRIS, University of Wisconsin-Madison, USA

We study the design and analysis of parallel join algorithms in a topology-aware computational model. In this model, the network is modeled as a directed graph, where each edge is associated with a cost function that depends on the data transferred between the two endpoints and the link bandwidth. The computation proceeds in synchronous rounds and the cost of each round is measured as the maximum cost over all the edges in the network. Our main result is an asymptotically optimal join algorithm over symmetric tree topologies. The algorithm generalizes prior topology-aware protocols for set intersection and cartesian product to a binary join over an arbitrary input distribution with possible data skew.

CCS Concepts: • **Theory of computation → Database query processing and optimization (theory)**; **Massively parallel algorithms**.

Additional Key Words and Phrases: Join, Massively Parallel Processing, Topology-aware

ACM Reference Format:
Xiao Hu and Paraschos Koutris. 2024. Topology-aware Parallel Joins. *Proc. ACM Manag. Data* 2, 2 (PODS), Article 97 (May 2024), 25 pages. https://doi.org/10.1145/3651598

1 INTRODUCTION

Due to the rapid development of massively parallel data processing systems, such as MapReduce [14] and Spark [33], there has been a huge interest in the theoretical community to study algorithms [2–7, 15–19, 21, 22, 24, 24–26, 32] in massively parallel computation models, such as BSP [31] and MPC [27] models. Algorithms in these models are based on the assumption of homogeneity: nodes have the same computational power, and links have the same bandwidth. However, practical systems are much more complicated and display heterogeneous properties. For example, computational capability varies dramatically across nodes due to their different generations of CPUs and GPUs. Network parameters (e.g., bandwidth, latency) also differ among links depending on the network architecture. To capture these critical issues in practice, Blanas et al. [10] recently proposed a heterogeneous massively parallel data processing model. In their model, the underlying communication network is a directed graph, where nodes in the network are identified as *routers* and *compute nodes* depending on their computational capabilities. In addition, network links are associated with a cost function for data transmission, which is also general enough to capture the different computational capabilities of compute nodes.

In this model, Hu et al. [20] proved lower bounds and (almost) matching upper bounds for a few data processing tasks over symmetric tree topologies, including set intersection, Cartesian

*This work was partially done while the authors were visiting the Simons Institute for the Theory of Computing.

Authors' addresses: Xiao Hu, xiaohu@uwaterloo.ca, University of Waterloo, 200 University Ave W, Waterloo, Ontario, Canada, N2L 3G1; Paraschos Koutris, paris@cs.wisc.edu, University of Wisconsin-Madison, 500 Lincoln Dr, Madison, WI, USA, 53706.

ACM 2836-6573/2024/5-ART97
https://doi.org/10.1145/3651598

product, and sorting. It is worth mentioning that their algorithms assume that the cardinality of the initial data distributed across the nodes can be arbitrary and is known in advance as part of the parameter, while previous works assumed worst-case distribution or uniform distribution. Hence, this assumption leads to a more fine-grained notion of optimality, that is parameterized by the cardinality of the initial data distribution. This more strict notion of optimality has introduced significant challenges to algorithm design. As we discuss in Section 5, if only worst-case or uniform distribution is considered, many problems become trivial including all these three tasks as well as join to be investigated in this paper.

We take one further step for designing an (almost) optimal algorithm for the natural join between two relations $R(A, B) \bowtie S(B, C)$. Our algorithm captures the set intersection and Cartesian product on symmetric trees as special cases. However, constructing a general join algorithm requires new techniques, ideas, and primitives.

Indeed, one can view the set intersection as a particular case of a natural join, where tuples in one relation have distinct join keys. Applying a hashing protocol on the join keys works well, as each join key is associated with at most two tuples (one from each relation). Hence, there will be no load imbalance if the hashing works well. However, such a hash-based protocol does not work when each join key can be associated with multiple tuples. In the extreme case, the join becomes a Cartesian product (all tuples are associated with a single key), and all tuples are routed to the same node, incurring an unbounded cost. In [20], this problem was solved using a packing-based protocol that carefully load-balances each part of the Cartesian product to the compute nodes. However, it is unclear how to combine these two protocols to handle the general case with multiple keys associated with a different number of tuples, especially by targeting the fine-grained optimality with respect to the cardinality of initial data distribution.

An additional challenge is that the protocol for the Cartesian product in [20] solves only the case when two input relations have equal size. This turns out to be an essential restriction since the symmetry of the two relations (along with the symmetry in the network links) allows for a relatively simple optimal algorithm with a closed-form cost. When the symmetry breaks, no closed-form solution exists even in the simplest case of a symmetric star topology. This means we have to explore the behavior of an optimal protocol for Cartesian product, and further for join.

As the binary join is a critical primitive of more complex analytical queries, we finally discuss two interesting implications of our topology-aware binary join algorithms – acyclic join and sparse-matrix multiplication (a basic join-aggregate query). However, this is not a simple extension, as in the homogeneous massively parallel model [1, 21, 23], since the downstream operators bring new constraints to the initial data distribution, that would significantly affect the overall efficiency. We leave as future work how to *optimally* apply our topology-aware binary join algorithm to complicated analytical queries, to achieve fine-grained optimality with respect to the cardinality of initial data distribution.

1.1 Computational Model

Following prior work [10], we model the network topology using a directed graph $G = (V, E)$. Each directed edge e in E represents a network link, where the direction follows the flow of data on the link, and it is also associated with a bandwidth parameter w_e. Every node in V has full knowledge of the graph G. We distinguish a subset of nodes in the network $V_C \subseteq V$, to be *compute nodes*. Compute nodes are the only nodes that can store data and perform computation on their local data. The remaining nodes perform only routing decisions. More specifically, if a compute node $u \in V_C$ aims to send some amount of data to another compute node $v \in V_C$, u just wraps all associated data as a whole package and stamps it with the *destination address* v as well as a *routing path from* u *to* v. Once a non-compute node $u' \in V - V_C$ receives a data package, it first retrieves the destination

address v as well as the routing path, and forwards this package to the next node on this path from u' to v. We only need $O(\log |V|)$ bits to encode the destination information. Moreover, we will focus on *symmetric tree topologies*[1] such that G is a tree, where the links in E are symmetric, i.e., for two edges $e = (u, v)$ and $e' = (v, u)$, we have $w_e = w_{e'}$. In this case, G is a strongly connected tree where every pair of compute nodes is connected/communicated through a unique directed path. For simplicity of algorithm design and presentation, we assume that the compute nodes V_C are the leaves of G.[2] In a tree topology, there is a unique path between every pair of compute nodes, thus our algorithm does not need to specify the routing path explicitly. We only consider strongly connected networks where every pair of compute nodes is connected through a directed path; in other words, every compute node can send data to any other compute node.

A parallel algorithm $\mathcal{A}$ proceeds in sequential *rounds*. Initially, the input data is distributed across the compute nodes V_C. At every round, the compute nodes first perform computation on their local data and then communicate by sending data to other compute nodes in the network. After the algorithm finishes, every output result must be emitted by at least one compute node. We confine ourselves to the tuple-based assumption that when a join result is emitted, all participating tuples must reside on the same node.[3]

The cost of such an algorithm $\mathcal{A}$ comes from synchronization and data transmission. The synchronization cost is measured by the number of rounds, denoted as r. The data transmission cost denoted as $C(\mathcal{A})$, is defined as the sum of the costs over all rounds i, $C(\mathcal{A}) = \sum_{i=1}^{r} C_i(\mathcal{A})$. To model the cost of each round, we first assign to each link e in the network a cost function $f_e : \mathbb{N} \to \mathbb{R}$. The quantity $f_e(x) = x/w_e$ is interpreted as the cost of transmitting x bits through link e. Let the quantity $Y_i(e)$ denote the total data (measured in bits) that is routed through link e during round i. Then, we can express the cost as $C_i(\mathcal{A}) := \max_{e \in E} f_e(|Y_i(e)|)$, i.e., the cost of each round is captured by the cost of transferring data through the most bottlenecked link in the network.

In this coarse-grained computational model, we do not consider the delay in routing, since the congestion cost measured above will dominate the overall cost when data size is much larger than network size (i.e., the number of compute nodes), which is a quite reasonable assumption for practical systems dealing with big data.[4] In addition, we do not consider the network size as a constant; otherwise, this model will lose the power of parallelism. In previous works on massively parallel processing [6, 7, 21, 24–27, 32], the number of compute nodes is a commonly-used parameter in complexity measurement. We compare it with other parallel/distributed model in Section 6.

Initial Data Distribution. Given a symmetric tree topology $G = (V, E)$ with compute nodes V_C, and an initial data distribution $\mathcal{D}$ for two input relations $R(A, B), S(B, C)$ across V_C, we denote by $R_v^{\mathcal{D}}, S_v^{\mathcal{D}}$ the R-tuples and S-tuples respectively in node $v \in V_C$. Let $N_v^{\mathcal{D}} = |R_v^{\mathcal{D}}| + |S_v^{\mathcal{D}}|$. When the

[1]The restriction to tree topologies is natural, since many practical network topologies can be captured as trees, such as star and fat tree. There is also a technical reason that there is a unique routing path between any two nodes in a tree topology. In a non-tree network, the presence of possibly multiple routing paths would complicates things. To extend from trees to general topologies, one could consider embedding the general topology to a tree, and then use a tree-based algorithm.

[2]If some compute node in V_C is an internal node in G, we can transform G to a new topology G' by adding a new compute node v', introducing two links (v, v'), (v', v) with bandwidth $+\infty$, and making v a non-compute node. Any initial data stored at v is shifted to v', and any local computation performed by v is simulated by v'.

[3]While the tuple-based class of algorithms does not encompass all possible approaches, it does include all existing algorithms that we are aware of [6, 7, 19–26, 32] on massive parallel query processing. Therefore, analyzing the optimal communication complexity achievable by this class can provide valuable insights into the problem's characteristics.

[4]For example, if we have a network with link latency of 10 ms and bandwidth 10Gbps, then the cost of sending say 100G over the link will be 10.01 sec dominated by the bandwidth cost (which is 10 sec). In addition, latencies over long paths do not add up in this case, because of the effect of pipelining. For example, if we have a path of 100 links of the above type, then the cost will still remain around 10 sec, since nodes do not wait to receive all the data before they move it to the next node.

context is clear, we drop $\mathcal{D}$ from the notation. We note that an input instance I consists of the symmetric tree topology $G = (V, E)$ and the initial data distribution $\mathcal{D}$.

Optimality. We introduce the notion of *cardinality configuration* of data distribution. A cardinality configuration ϕ across the compute nodes V_C is denoted as $\{(\phi_v^1, \phi_v^2) : \phi_v^1, \phi_v^2 \in \mathbb{Z}^*, v \in V_C\}$. A data distribution $\mathcal{D}$ *conforms* to a cardinality configuration ϕ if $|R_v^{\mathcal{D}}| = \phi_v^1$ and $|S_v^{\mathcal{D}}| = \phi_v^2$ for every compute node $v \in V_C$. We adopt the notion of *parameterized analysis*, i.e., we partition the whole space of inputs into disjoint sub-spaces, where instances inside each sub-space share the same symmetric tree topology, cardinality configuration, and output size. Let $\mathbb{I}(G, \phi, \text{OUT})$ denote that class of instances characterized by the three parameters G, ϕ, OUT. Then the cost of an algorithm $\mathcal{A}$ is thus a function of G, ϕ, OUT, defined as $L_{\mathcal{A}}(G, \phi, \text{OUT}) = \max_{I \in \mathbb{I}(G,\phi,\text{OUT})} L_{\mathcal{A}}(I)$, where $L_{\mathcal{A}}(I)$ is the cost of computing I by $\mathcal{A}$. An algorithm $\mathcal{A}$ is *topology-aware-optimal* if for every algorithm $\mathcal{A}'$, $L_{\mathcal{A}}(G, \phi, \text{OUT}) = O\left(L_{\mathcal{A}'}(G, \phi, \text{OUT})\right)$.

1.2 Roadmap

We next describe some high-level ideas behind our algorithm, which also serve as a roadmap. For better understanding, we summarize commonly used notations in Table 1.

To construct a general join algorithm, we resort to sorting instead of hashing that has been used for set intersection in [20]. The sort-based protocol follows the general idea of [24] for topology-oblivious parallel joins. One first attempt would be to use the sorting protocol from [20] to sort the union of the two relations, $R \cup S$ using the join key. However, this idea does not give an optimal protocol. Indeed, consider the case where $R \ll S$. In this case, the algorithm would have to sort $R \cup S$, redistributing the tuples from the larger relation S. However, if we were to broadcast R to every compute node and not move S at all, we would obtain a much cheaper solution. In other words, instead of sorting globally, we need to sort the largest relation in a topologically "local" way. We achieve this by carefully grouping the compute nodes so that each group's input data is approximately balanced (Section 4.3 and 4.4).

The sorting step guarantees that we can locally compute the join results for join keys that are fully contained within a compute node. However, it cannot deal with join keys that span multiple compute nodes (we call these *boundary keys*, denoted as K). This case will happen when data skew exists since a key will not fit in a single node without blowing up the algorithm cost. The main technical contribution of this paper is how to handle the join computation for these boundary keys. To solve this problem, we view it as the simultaneous computation of multiple Cartesian products $\{R_b \times S_b\}_{b \in K}$. Our solution uses two primitives that may be of independent interest. The first primitive (Section 2) is an optimal protocol that redistributes the data of a relation from an input distribution to the desired output distribution with minimal cost. The second primitive (Section 3) designs an optimal protocol that allocates the computation of a set of Cartesian products from the root to the leaves of a tree. To achieve the topology-aware-optimality, we further explore the structural property of the underlying tree network with respect to the initial data distribution (Section 4.1), such that the tree network is decomposed into multiple edge-disjoint subtrees (denoted as *α-tree*) connected by a skeleton subtree (denoted as *β-tree*). The remaining challenging question is to pack Cartesian products over β-tree such that each α-tree will compute the allocated Cartesian products independently (Section 4.3 and 4.5).

2 DATA REDISTRIBUTION

In this section, we discuss a fundamental problem in topology-aware data processing, which will serve as a building block of our general join algorithm and also can be of independent interest. All missing proofs are given in Appendix A.

Notations	Definitions
$G = (V, E)$	tree network G with nodes V and edges E
V_C	the leaf nodes (also the compute nodes) of G
G_e^-, G_e^+	two subtrees after removing edge e from G
E_α, E_β	α-edges, β-edges
$G_\alpha^j (\vec{G}_\alpha^j)$	undirected (directed) α-trees induced on E_α
G_β	β-tree of G induced on E_β
V_β	nodes in G_β
$\mathcal{L}_\beta$	leaf nodes of G_β
$\mathcal{P}$	a balanced partition of $\mathcal{L}_\beta$
P	a block of $\mathcal{P}$ (also a subset of $\mathcal{L}_\beta$)
w_e	bandwidth of edge $e \in E$
w_v	bandwidth of edge between v and its parent
R, S	input relations
$\mathcal{D}$	initial data distribution
R_v, S_v	R-tuples, S-tuples distributed at v initially
N_v	total size of tuples distributed at v initially
K	boundary keys
R_b, S_b	R-tuples, S-tuples with key b

Table 1. Notations.

Data Redistribution Primitive. Consider a (not necessarily symmetric) tree topology $G = (V, E)$ with leaf nodes $V_C = \{v_1, \ldots, v_\ell\}$. Moreover, assume k input relations $R_1, \ldots, R_k$ distributed over leaf nodes according to a distribution $\mathcal{D}$. We say that $\mathcal{D}$ *conforms* to a matrix Λ of dimensions $k \times \ell$ with positive integers if for every input relation R_i and every leaf node v_j, there are Λ_{ij} tuples from R_i distributed at v_j. We call such a matrix a size-constraint matrix. Given two size-constraint matrices Λ, Λ', our goal is to design an algorithm that transforms any input data distribution $\mathcal{D}$ that conforms to Λ to a data distribution $\mathcal{D}'$ that conforms to Λ'. In other words, we want to redistribute the data such that a new set of size constraints is satisfied at every leaf, and we also want to achieve this with the minimum cost. Note that to properly define the task, we need to make sure that for every relation R_i, $\sum_j \Lambda_{ij} = \sum_j \Lambda'_{ij} = |R_i|$.

Lower Bound. We next show a lower bound by relating the data redistribution to tuple differentials. We introduce some terminologies. For an edge $e = (u, v) \in E$, removing e splits G into two connected subtrees, which are denoted as G_e^- (that contains u) and G_e^+ (that contains v).

Definition 2.1 (Tuple Differential). For a tree network $G = (V, E)$, and two size-constraint matrices Λ, Λ' of dimensions $k \times \ell$, the tuple differential of an edge $e \in E$ with respect to relation R_i is defined as $\Delta_e^i := \max\left\{0, \sum_{v_j \in G_e^+} \Lambda'_{ij} - \sum_{v_j \in G_e^+} \Lambda_{ij}\right\}$, and the tuple differential of $e \in E$ is defined as $\Delta_e = \sum_{i=1}^k \Delta_e^i$.

Consider an edge $e = (u, v) \in E$. Assume $\Delta_e^i > 0$ holds for some relation R_i. Then, G_e^+ needs to obtain at least Δ_e^i tuples from G_e^-. Thus, at least Δ_e^i tuples must cross the edge, and incur a transmission cost of Δ_e^i / w_e. Summing across all relations will yield the following lower bound:

Lemma 2.2. *Any algorithm that solves the data redistribution problem has cost at least* $\max_{e \in E}\{\Delta_e / w_e\}$.

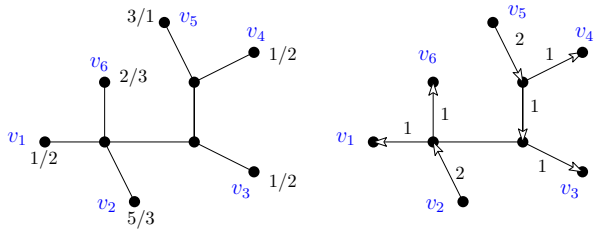

Fig. 1. An instance of data redistribution with $k = 1, \ell = 6$. The left shows two size-constraint matrices $\Lambda = \langle 1, 5, 1, 1, 3, 1\rangle$, $\Lambda' = \langle 2, 3, 2, 2, 1, 3\rangle$. The right shows a multi-commodity flow.

We next mention the following lemma about tuple differentials.

Lemma 2.3. *Let $e = (u, v)$, $e' = (v, u)$ be two edges in G. Then: $\Delta^i_e - \Delta^i_{e'} = \sum_{v_j \in G^+_e} \Lambda'_{ij} - \sum_{v_j \in G^+_e} \Lambda_{ij}$.*

Multi-commodity Flow. To construct an optimal algorithm for the data redistribution task, we will use the framework of multi-commodity flow. First, we construct a *flow network* $G_F = (V_F, E_F)$ as follows. The set of nodes V_F consists of the nodes V, with the addition of k source nodes $s_1, \ldots, s_k$ and k target nodes $t_1, \ldots, t_k$. The set of edges E_F includes all the edges in E with cost $1/w_e$ and capacity $+\infty$. In addition, for every $i \in [k]$, $j \in [\ell]$ we introduce two edges: (s_i, v_j) with cost 0 and capacity Λ_{ij}, and (v_j, t_i) with cost 0 and capacity Λ'_{ij}. Second, we introduce k commodities, one for each relation R_i, where commodity i has source s_i, target t_i, and demand Λ_i. A *flow* for G_F is an assignment $\{f_i(e)\}$, where $f_i(e)$ is a positive integer, such that the following hold:

- $f_i(s_i, v_j) = \Lambda_{ij}$ and $f_{i'}(s_i, v_j) = 0$ when $i' \neq i$.
- $f_i(v_j, t_i) = \Lambda'_{ij}$ and $f_{i'}(v_j, t_i) = 0$ when $i' \neq i$.
- For every node $v \in V$ and $i \in [k]$, $\sum_u f_i(v, u) = \sum_u f_i(u, v)$.

Any flow corresponds to an algorithm for the data redistribution problem. Indeed, we can interpret a unit of flow from commodity i as a tuple from relation R_i. We seek the flow with the minimum cost, which is $\max_{e \in E} \sum_i f_i(e)/w_e$. We can solve the multi-commodity flow problem via a linear program in polynomial time, but the flow solution may be fractional. Instead, we provide a more direct flow solution using tuple differentials, which simply assigns each flow $f_i(e)$ to be the tuple differential Δ^i_e, incurring an optimal cost of $\max_{e \in E}\{\Delta^i_e/w_e\}$. See an example in Figure 1.

Theorem 2.4. *In a tree network, data redistribution problem can be optimally computed in 1 round.*

3 CARTESIAN PRODUCT PACKING

In this section, we present a second fundamental primitive that we need for our join algorithm. Consider a tree topology $G = (V, E)$ that we have rooted at some node r. We are also given a finite set of relation pairs (R_b, S_b), where b belongs in a finite set B. Initially, all tuples are sitting at the root r. The task is to find an algorithm that routes all tuples from the root to the leaf nodes such that every pair of the Cartesian product $R_b \times S_b$ can be locally computed at some leaf node. We will refer to this task as *cartesian product packing*.

For a tree $G = (V, E)$ rooted at r, let $\mathcal{L}_v$ be the set of leaf nodes contained in the subtree rooted at node $v \in V$. If $v \in V_C$, $\mathcal{L}_v = \emptyset$. For any non-root node $v \in V \setminus \{r\}$, we use w_v to denote the bandwidth of the edge between node v and its (unique) parent node.

3.1 A Lower Bound

We first provide a lower bound for the task. Consider any algorithm $\mathcal{A}$ that correctly computes the task with cost L. Let $\square_{b,u}$ be the set of pairs from $R_b \times S_b$ that leaf node $u \in V_C$ emits locally. We

assume w.l.o.g. that the sets $\square_{b,u}$ are disjoint (indeed, if a pair is emitted by at least two leaf nodes, we can simply modify $\mathcal{A}$ to emit only in one location; that can only decrease its cost).[5] For an internal node $v \in V$, we define $\square_{b,v} = \bigcup_{u \in \mathcal{L}_v} \square_{b,u}$ (recall that $\mathcal{L}_v$ is the set of leaf nodes contained in the subtree rooted at v). Intuitively, $\square_{b,v}$ are the tuple pairs that node v is responsible for. Hence, for the root we have $\square_{b,r} = \bigcup_u \square_{b,u}$. Let $g_{b,v} = |\square_{b,v}|$. Since $\mathcal{A}$ is a correct algorithm, we must have $\square_{b,r} = R_b \times S_b$ and thus $g_{b,r} = |R_b| \cdot |S_b|$.

Now, for any $b \in B$ and node $v \neq r$, let $x_{b,v}$ ($y_{b,v}$ resp.) be the number of tuples from R_b (S_b resp.) routed through v by $\mathcal{A}$. We now have two observations. First, the cost of $\mathcal{A}$ is $L = \max_v \sum_b (x_{b,v} + y_{b,v})/w_v$. Second, since node v is responsible for emitting $g_{b,v}$ pairs from $R_b \times S_b$, we must have that $g_{b,v} \leq x_{b,v} \cdot y_{b,v}$; in other words, the tuples received must form enough area to "cover" the set $\square_{b,v}$. Putting all this together, the cost of any algorithm is at least the optimal cost L^* of the following non-linear integer program:

$$
\begin{aligned}
\min \quad & L \\
\text{s.t.} \quad & |R_b| \cdot |S_b| = g_{b,r}, \\
& \sum_{u \in \mathcal{L}_v} g_{b,u} = g_{b,v}, && b \in B, v \in V, \\
& \sum_{b \in B} (x_{b,v} + y_{b,v}) \leq L \cdot w_v, && v \in V \setminus \{r\}, \\
& x_{b,v} \leq |R_b|, y_{b,v} \leq |S_b|, x_{b,v} \cdot y_{b,v} \geq g_{b,v}, && b \in B, v \in V \setminus \{r\}, \\
& g_{b,v}, x_{b,v}, y_{b,v} \in \mathbb{Z}^*, && b \in B, v \in V \setminus \{r\}
\end{aligned}
\tag{1}
$$

Note that here we have added additional constraints that say that $x_{b,v}$ can never exceed $|R_b|$ and $y_{b,v}$ cannot exceed $|S_b|$. Later, we use $L^*(G, \{R_b \times S_b : b \in B\})$ to denote the optimal cost of (1) defined over network G and a set of Cartesian products $R_b \times S_b$ for $b \in B$.

Remark 1: Symmetry in input sizes of Cartesian products. Suppose $|R_b| = |S_b| = N_b$ for any $b \in B$. We can find a rather clean characterization on a constant-approximation solution of (1). Let's start with two critical observations on the optimal solution of (1). Firstly, $x_{b,v} = y_{b,v}$; otherwise, we replace $x_{b,v}, y_{b,v}$ with $\sqrt{x_{b,v} \cdot y_{b,v}}$, satisfying all constraints with $2 \cdot \sqrt{x_{b,v} \cdot y_{b,v}} \leq x_{b,v} + y_{b,v}$. Secondly, $x_{b,v} \cdot y_{b,v} = g_{b,v}$; otherwise, we replace $x_{b,v} = y_{b,v} = \sqrt{g_{b,v}}$, satisfying all constraints with $2\sqrt{g_{b,v}} \leq x_{b,v} + y_{b,v}$. This way, we can transform (1) into the following integer program:

$$
\begin{aligned}
\min \quad & L \\
\text{s.t.} \quad & N_b^2 \leq \sum_{v \in \mathcal{L}} z_{b,v}^2, && b \in K, \\
& \sum_{b \in K} \sqrt{\sum_{u \in \mathcal{L}_v} z_{b,u}^2} \leq \frac{L}{2} \cdot w_v, && v \in V \setminus \{r\}, \\
& \sum_{b \in K} z_{b,u} \leq \frac{L}{2} \cdot w_v, && v \in \mathcal{L}, \\
& z_{b,v} \leq N_b, z_{b,v} \in \mathbb{Z}^*, && v \in \mathcal{L}
\end{aligned}
\tag{2}
$$

[5] We can always strengthen the power of the algorithm for proving a lower bound. Suppose $V_C = \{v_1, v_2, \cdots, v_\ell\}$. If a pair is emitted by both v_i and v_j, we always let the node with the smallest index to emit the pair. Consider a leaf node v_i. If there exists some tuple $t_b \in R_b$, such that all pairs from $\square_{b,v_i}$ in a form of $(t_b, *)$ are emitted by other nodes, there is no need to send t_b to v_i, hence the cost can only be decreased.

Recall that $\mathcal{L}_v$ is the set of leaf nodes included by the subtree rooted at v, and w_v is the bandwidth parameter of the edge between v and its unique parent node. Here, $\mathcal{L}$ denotes the set of leaf nodes in $\vec{G}$. Note that $z_{b,v}$ is defined as a variable for every leaf node $v \in \mathcal{L}$ and key $b \in K$. Each solution $z : K \times \mathcal{L} \to \mathbb{Z}^*$ of (2) has a clean closed-form on its cost: $\mathsf{C}_z = \max_{v \in V-r} \frac{1}{w_v} \cdot \sum_{b \in K} \sqrt{\sum_{u \in \mathcal{L}_v} z_{b,u}^2}$.

Remark 2: One Cartesian product. Suppose there is only one Cartesian product, say $B = \{b\}$. Let $R = R_b$ and $S = S_b$. We can simplify (1) as the following integer program:

$$\begin{aligned} \min \quad & L \\ \text{s.t.} \quad & |R| \cdot |S| = g_r, \\ & \sum_{u \in \mathcal{L}_v} g_u = g_v, && v \in V, \\ & x_v + y_v \le L \cdot w_v, && v \in V \setminus \{r\}, \\ & x_v \le |R|, y_v \le |S|, x_v \cdot y_v \ge g_v, && v \in V \setminus \{r\}, \\ & g_v, x_v, y_v \in \mathbb{Z}^* && v \in V \setminus \{r\} \end{aligned} \tag{3}$$

Let's start with one critical observations on the optimal solution of (1). If $\max\{x_v, y_v\} \le |R|$, we have $x_v = y_v$. Suppose not, we replace x_v, y_v with $(x_v \cdot y_v)^{1/2}$, satisfying all constraints with $2 \cdot (x_v \cdot y_v)^{1/2} \le x_v + y_v$. We partition leaf nodes in $\mathcal{L}$ into two subsets $\mathcal{L}_1 = \{v \in \mathcal{L} : \max\{x_v, y_v\} \le |R|\}$ and $\mathcal{L}_2 = \mathcal{L} - \mathcal{L}_1$. The cost of L can be explicitly represented as

$$\max_{v \in V \setminus \{r\}} \frac{1}{w_v} \left(\max \left\{ \sum_{u \in \mathcal{L}_v \cap \mathcal{L}_1} \frac{x_u^2}{|R|}, \sqrt{\sum_{u \in \mathcal{L}_v \cap \mathcal{L}_1} x_u^2} \right\} + \sum_{u \in \mathcal{L}_v \cap \mathcal{L}_2} y_u \right).$$

3.2 An Algorithm

Given the optimization problem (1) that provides the lower bound, one can solve it to obtain the optimal values of $x_{b,v}$, $y_{b,v}$ and $g_{b,v}$. However, this does not directly imply an algorithm, since we also need to specify which tuples are sent where (instead of how many of them). Below, we use (1) to construct an almost-optimal algorithm.

Packing Squares. Consider the assignments $x^*_{b,v}, y^*_{b,v}, g^*_{b,v}$ for (1) that achieves the optimal solution with cost L^*. For any $b \in B$, we define: $i^*_b := \lceil \log_2 \min\{|R_b|, |S_b|\} \rceil$. Our algorithm recursively assigns to each node v a set of squares $S_{b,v}$ in the form of

$$\{(i, c_i) : c_i \in \{0, 1, 2, 3\}, 1 \le i < i^*_b\} \cup \{(i^*_b, c_{i^*_b}), c_{i^*_b} \ge 0\}.$$

We interpret this as that there are c_i squares of type i, with dimensions $2^i \times 2^i$. We are allowed to have up to 3 squares of type i when $i < i^*_b$, but for type i^*_b the number is unlimited. The recursion works in a bottom-up fashion starting from the leaf nodes. The square assignment will be done such that the following property holds: their total area is exactly $8 \cdot g^*_{b,v}$. In other words, $\sum_i c_i \cdot 4^i = 8 \cdot g^*_{b,v}$.

For the base case, we assign to compute node v a set $S_{b,v}$ such that $\sum_i c_i \cdot 4^i = 8 \cdot g^*_{b,v}$. This is always possible, and moreover there is a unique way to achieve this decomposition. Next, consider some internal node u. Each of its children v is assigned a set of squares $S_{b,v}$ from the recursion. We start the following procedure in an increasing order of $i \ge 0$: for each type $i < i^*_b$, if there are 4 squares of type i in $\bigcup_v S_{b,v}$, we pack them into one larger square of type $i+1$. In this way, we can transform the set $\bigcup_v S_{b,v}$ into a new set of squares $S_{b,u}$, where for every $i < i^*_b$, $c_i \le 3$. Note again that for type i^*_b there is no limit on the number of squares. See Figure 2.

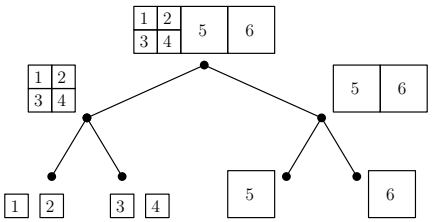

Fig. 2. An example of packing squares.

Theorem 3.1. *In a rooted tree topology G, the Cartesian product packing problem can be computed in a single round with cost $O(1)$ away from the optimal solution.*

Proof of Theorem 3.1. We can show that this assignment fully covers every $R_b \times S_b$ grid, so that every point in the grid (i.e., every pair in the Cartesian product) is assigned to some compute node. In addition, it has cost $O(L^*)$. We split the proof into two parts.

Covering: We will first show that the above process indeed fully covers every $R_b \times S_b$ grid, so that every point in the grid (i.e., every pair in the Cartesian product) is assigned to some compute node. W.l.o.g, assume that $R_b \le S_b$. Observe that the following property holds: all squares of type $< i$ can fit into a single square of type i (here, it is crucial that $c_i \le 3$). We can now write $|R_b| \cdot |S_b|$ as:

$$g^*_{b,r} = \frac{1}{8} \sum_{(i,c_i) \in S_{b,r}} c_i \cdot 4^i \le \frac{1}{8}(c_{i^*_b} + 1) \cdot 4^{i^*_b} \le \frac{1}{2}(c_{i^*_b} + 1) \cdot |R_b|^2.$$

Hence, we can conclude that $c_{i^*_b} \ge 2|S_b|/|R_b| - 1 \ge |S_b|/|R_b|$.

Note that we have $c_{i^*_b}$ fully packed squares of type i^*_b. Each such square has dimensions $2^{i^*_b} \times 2^{i^*_b}$, so at least $|R_b| \times |R_b|$. Moreover, if we place the $c_{i^*_b}$ squares next to each other along the horizontal axis, we have a fully packed rectangle where the first dimension has size at least $|R_b|$ and the second dimension has size at least $c_{i^*_b} \cdot |R_b| \ge |S_b|$. Thus, we can indeed covered the desired grid.

Cost: Consider any node $v \ne r$; we will calculate the amount of data that goes through its parent link with capacity w_v. Consider the largest i^* in $S_{b,v}$ such that $c_{i^*} > 0$. We can measure the amount of data that goes through the link and is associated with b as: $A_b = 2 \cdot \sum_i c_i \cdot 2^i \le 2(c_{i^*} + 3) \cdot 2^{i^*}$.

As before, assume w.l.o.g. that $|R_b| \le |S_b|$. We now distinguish two cases:

- $i^* < i^*_b$. Then, we have $c_{i^*} \le 3$. Also, note that $c_{i^*} 4^{i^*} \le 8 \cdot g^*_{b,v}$, so $2^{i^*} \le \sqrt{8 g^*_{b,v}}$. We can write:

$$A_b \le 12 \cdot 2^i \le 12\sqrt{2g^*_{b,v}} \le 12\sqrt{2x^*_{b,v}y^*_{b,v}} \le 6\sqrt{2} \cdot (x^*_{b,v} + y^*_{b,v})$$

- $i^* = i^*_b$. Then, we can write: $A_b \le 8 \cdot c_{i^*_b} \cdot 2^{i^*_b} \le 8 \cdot c_{i^*_b} \cdot |R_b|$. But we also have: $c_{i^*} 4^{i^*} \le 8 \cdot g^*_{b,v}$, so $c_{i^*_b} \cdot |R_b|^2 \le 8 \cdot g^*_{b,v}$. Thus: $A_b \le 64 \cdot \dfrac{g^*_{b,v}}{|R_b|} \le 64 \cdot \dfrac{x^*_{b,v}y^*_{b,v}}{|R_b|} \le 64 \cdot y^*_{b,v}$.

Hence, the total size of data going through the link with capacity w_v is: $\sum_b A_b \le 64 \sum_b (x^*_{b,v} + y^*_{b,v})$ $\le 64 \cdot L^* \cdot w_v$. We conclude that our algorithm has cost $O(1)$ away from the optimal solution L^*. □

4 THE JOIN ALGORITHM

In this section, we present the general algorithm for computing the join $R(A, B) \bowtie S(B, C)$ over a symmetric tree topology $G = (V, E)$. Throughout this section, we assume that $|R| \le |S|$. The

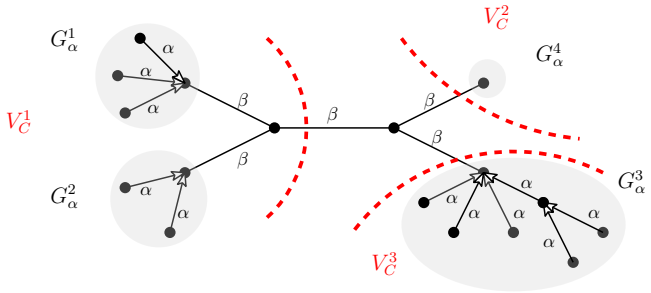

Fig. 3. A symmetric tree G, with four directed α-trees $\{G^1_\alpha, G^2_\alpha, G^3_\alpha, G^4_\alpha\}$ and a balanced partition $\{V^1_C, V^2_C, V^3_C\}$.

algorithm consists of two basic steps, where each step computes a partial join result. In the first step, we compute part of the result using a sort-based protocol. After sorting the tuples according to the join key (attribute B), the compute nodes can compute locally the result for the join keys that completely reside within a node. The join keys that cross different nodes (called *boundary keys*) have to be handled by the second step. The boundary keys can be roughly viewed as the "heavy" keys in the join computation.

Before we describe the steps of the algorithm, we need to introduce a few useful concepts over the symmetric tree topology.

4.1 Edges and Trees

Recall that each edge $e = (u, v)$ defines a partition of G as two connected subtrees: G^-_e and G^+_e. For simplicity, we also use G^-_e, G^+_e to denote the set of leaf nodes in the corresponding subtree.

We note a lower bound $C_{\text{int}} = \max_{e \in E} \frac{1}{w_e} \cdot \left\{ |R|, \sum_{v \in G^-_e} N_v, \sum_{v \in G^+_e} N_v \right\}$ for the cost of set intersection [20], hence it also forms a lower bound for a general join protocol. Using C_{int} as a guide, we classify the edges in E into two disjoint subsets:

$$E_\alpha = \left\{ e \in E : |R| > \min\{ \sum_{v \in G^-_e} N_v, \sum_{v \in G^+_e} N_v \} \right\}, \quad E_\beta = E - E_\alpha$$

An edge e is an α-edge if $e \in E_\alpha$ and a β-edge otherwise. Let G_β be the subgraph of G edge-induced by E_β; in [20], it has been shown that G_β always forms a connected tree, called the β-tree. We denote the nodes of G_β as V_β and the leaf nodes of G_β as $\mathcal{L}_\beta$. On the other hand, the graph edge-induced by E_α may not be connected; let $\left\{G^1_\alpha, G^2_\alpha, \cdots, G^\ell_\alpha\right\}$ be the connected subtrees (called α-trees). See Figure 3. If $E_\beta = \emptyset$, the network G itself forms a single α-tree.

For each α-tree G^j_α, we can define a *directed* version $\vec{G}^j_\alpha$ as follows: (i) $\vec{G}^j_\alpha$ has the same vertex set as G^j_α, and (ii) for an edge $e = (u, v)$ in G^j_α, if $\sum_{v' \in G^-_e} N_{v'} < \sum_{v' \in G^+_e} N_{v'}$[6], $\vec{G}^j_\alpha$ contains only an edge from u to v, otherwise only an edge from v to u. We say that a directed tree is *rooted* at the node r if every edge points towards r. It turns out that the set of directed trees $\left\{\vec{G}^1_\alpha, \cdots, \vec{G}^\ell_\alpha\right\}$ has a very specific structure, as the next lemma shows.

Lemma 4.1. *Every $\vec{G}^j_\alpha$ is a directed rooted tree. Moreover, if $E_\beta \neq \emptyset$, the root of each $\vec{G}^j_\alpha$ is the unique node that is adjacent to a β-edge.*

[6]Here we can assume w.l.o.g. that equality will never occur.

Proof of Lemma 4.1. The proof distinguishes two cases. First, consider the case where $E_\beta \neq \emptyset$. Consider an edge $e = (u, v)$ such that G_e^+ contains the root. Then, G_e^+ contains at least one β-edge, so it must be that $|R| \leq \sum_{w \in G_e^+} N_v$. But because e is an α-edge, we also have $|R| > \sum_{w \in G_e^-} N_v$. Hence, $\sum_{w \in G_e^+} N_v > \sum_{w \in G_e^-} N_v$ and the edge is indeed directed towards the root. Second, consider the case where $E_\beta = \emptyset$. Since $\vec{G}_\alpha$ is a directed tree, there must exist at least one node with no outgoing edges; otherwise, there would be a cycle in the graph, a contradiction. Hence, it suffices to show that there is at most one such a node. By contradiction, assume two nodes u, v with outdegree 0. Consider the unique path between u, v: then, there must be a node w in the path with out-degree at least two. In other words, we have edges $e_1 = (w, v_1)$ and $e_2 = (w, v_2)$. But then $N \geq \sum_{x \in G_{e_1}^+} N_x + \sum_{x \in G_{e_2}^+} N_x > N/2 + N/2 = N$, a contradiction. □

An illustration of this structure when $E_\beta \neq \emptyset$ is given in Figure 3. An important consequence of the lemma that will be of use later is that every compute node v can send its local data (of size N_v) to the root node of its corresponding α-tree (which will be a leaf node of the β-tree) with a cost that does not exceed the lower bound C_{int}.

4.2 Warm Up: A Single α-Tree

We start with an easy case when network G itself forms a single α-tree (i.e., $E_\beta = \emptyset$), rooted at some node r. This case could happen on some simple inputs, such as when there is symmetry on the input size (i.e., $|R| = |S|$), or when the initial data distribution is uniform (i.e., $N_u = N_v$ for every pair of compute nodes $u, v \in V_C$) together with the input size constraint $|S| < (|V_C| - 1) \cdot |R|$. Our algorithm is quite straightforward via three steps:

- **Step 1.** We first invoke the *weighted TeraSort* algorithm [20] to sort all tuples by their join values.[7] After the data is sorted, each node locally computes the join results with non-boundary keys, where a key is *boundary* if it is the smallest or the largest one among its local tuples (and hence it can possibly have tuples in other compute nodes as well). Let K be the set of boundary keys.
- **Step 2.** We next compute the input sizes $|R_b|, |S_b|$ for each $b \in K$ by invoking the multi-numbering primitive (see Appendix D), and then broadcast $|R_b|$'s, $|S_b|$'s to all compute nodes in V_C.
- **Step 3.** At last, we just send all tuples with $b \in K$ to the root r, and simply invoke the Cartesian product packing primitive in Section 3 to handle the boundary keys.[8] The whole topology G together with the data statistics of boundary keys are stored at the local memory of each compute node, hence the program (1) can be solved locally by each node.

This protocol incurs a cost of $O(\mathsf{C}_{\mathsf{int}} + \mathsf{C}_{\mathsf{pack}})$, where $\mathsf{C}_{\mathsf{pack}}$ is the optimal solution of (1) parameterized by $\{|R_b|, |S_b| : b \in K\}$.

In the remaining, we will deal with the general case where the network G has a β-tree as its core, with a set of α-trees incident to the leaf nodes of β-tree, as shown in Section 4.1.

4.3 Tree Partitioning

A crucial idea in general our algorithm is to further group together the α-trees into disjoint partitions, such that within each partition certain properties hold. Consider G_β, a weight function w that maps any subset of $\mathcal{L}_\beta$ to a positive number, and a threshold $\tau \geq 0$.

[7] Given an ordering of compute nodes V_C, the goal is to redistribute the input elements such that elements on node v_i are always no larger than those on node v_j if $i < j$. In our context, we can pick an arbitrary node as the root, and any left-to-right traversal of the tree is a valid ordering of V_C.

[8] Here, we "conceptually" split the routing into two steps (from leaf nodes to root and then from root to leaf nodes) for easy understanding. But this is actually done in a single round, since each leaf node can directly send data to other leaf nodes (maybe even without going through the root), where the root is only responsible for routing.

Definition 4.2. For G_β with leaf nodes $\mathcal{L}_\beta$, parameter $\tau \geq 0$ and function $\theta : 2^{\mathcal{L}_\beta} \to \mathbb{Z}^*$, a partition $\mathcal{P}$ of $\mathcal{L}_\beta$ is *balanced* with respect to θ and τ if the following hold:

- The minimum Steiner trees of the sets in $\mathcal{P}$ are edge-disjoint;[9]
- For every $P \in \mathcal{P}$, $\theta(P) \geq \tau$;
- For every β-edge e in the Steiner tree of $P \in \mathcal{P}$ in G_β, we have $\min\{\theta(P \cap G_e^+), \theta(P \cap G_e^-)\} \leq \tau$;
- For every $P \in \mathcal{P}$ with $|P| > 1$, $\theta(P) < 2\tau$.

We should note here that the above definition generalizes the definition of a balanced partition in [20]. It will be convenient to think about a balanced partition as a partition of the set of α-trees. Figure 3 shows an example of a balanced partition $\{\{G_\alpha^1, G_\alpha^2\}, \{G_\alpha^3\}, \{G_\alpha^4\}\}$.

It turns out that we can obtain a balanced partition through a simple routine (Algorithm 1) that runs in polynomial time in the size of the network. Our procedure initially creates a single group for each leaf node of G_β. Then, it starts merging the groups (starting from the leaves of the tree) as long as the total "weight" (i.e., the value under function θ) in the group is less than the threshold τ.

Lemma 4.3. *If $\theta(\mathcal{L}_\beta) \geq \tau$, Algorithm 1 returns a balanced partition in polynomial time.*

4.4 Step 1: Sort-based Join

The first step of the join algorithm performs a sort-based join. As we discussed in the introduction, we do not sort $R \cup S$ globally. Instead, we construct a balanced partition $\mathcal{P}$ of $\mathcal{L}_\beta$, and then separately sort R together with the local S-tuples in the nodes of each block of the partition $\mathcal{P}$. As described in Algorithm 2, it consists of three steps:

- **(Step 1.1)** We first construct a balanced partition $\mathcal{P}$ of $\mathcal{L}_\beta$ by invoking Algorithm 1, where $\tau = |R|$ and function $\theta(P)$ for a subset of leaf nodes $P \subseteq \mathcal{L}_\beta$ is defined as the sum of N_v for every compute node v in the α-tree rooted at some $u \in P$, i.e.,
$$\theta(P) = \sum_{u \in P} \sum_{v \in V_C \cap G_\alpha^i : \vec{G}_\alpha^i \text{is rooted at } u} N_v.$$
As $\theta(\mathcal{L}_\beta) = |R| + |S| \geq |R| = \tau$, we can obtain a balanced partition in polynomial time using Algorithm 1.
- **(Step 1.2)** Let $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ be the resulting partition. We now define a random hash function h^i that maps independently each tuple t to a leaf node v in one of the α-trees in P_i with probability $\Pr[h^i(t) = v] = \frac{N_v}{\theta(P_i)}$. As described in Algorithm 2, each R-tuple is hashed across all blocks of the partition $\mathcal{P}$.
- **(Step 1.3)** We then invoke the *weighted TeraSort* algorithm [20] to sort all R-tuples as well as local S-tuples by their join keys inside each block P_i of the partition $\mathcal{P}$.[10] Since each block defines an edge-disjoint Steiner tree, we can perform sorting simultaneously for all blocks. After each node locally computing the join results with non-boundary keys, it then remains to compute the join results with boundary keys across all leaf nodes.

We note an easy case when the network G itself is a β-Tree (i.e., $E_\alpha = \emptyset$). In **(Step 1.1)**, every compute node in V_C forms an individual block in $\mathcal{P}$. Our algorithm just degenerates to let all compute nodes broadcast their R-tuples (even simpler than line 2-3 in Algorithm 2) and compute local join results. This case could happen on some simple inputs, such as when the initial data distribution is uniform (i.e., $N_u = N_v$ for every pair of compute nodes $u, v \in V_C$) together with the

[9]For a tree $G = (V, E)$ and a subset of nodes $S \subseteq V$, the minimum Steiner tree of S is a subtree G' of G, that include all nodes in S and all leaf nodes of G' are from S.

[10]We can pick an arbitrary node in the minimum Steiner tree of P_i as the root, and any left-to-right traversal of the underlying tree is a valid ordering of P_i.

Algorithm 1: TreePartition(G_β, θ, τ)

```
foreach x ∈ L_β do Γ(x) ← {x};
foreach x ∈ V_β \ L_β do Γ(x) ← {};
P ← ∅;
while |V_β| > 0 do
    Pick the leaf x ∈ V_β with the smallest θ(Γ(x));
    if θ(Γ(x)) ≥ τ then Add Γ(x) to P;
    else
        y ← unique neighbor of x in G_β;
        Γ(y) ← Γ(y) ∪ Γ(x);
    V_β ← V_β \ {x};
return P;
```

Algorithm 2: SortTree$(G, \mathcal{D})$

```
P = {P_1, P_2, ..., P_k} ← TreePartition(G_β, θ, |R|);
foreach v ∈ V_C and i ∈ k do
    Send every tuple t ∈ R_v^D to h^i(t);
foreach i ∈ k do
    Sort tuples in R ∪ (∪_{v∈P_i} S_v^D) by their join key over the minimum Steiner tree of P_i;
foreach v ∈ V_C do
    Compute local join results for non-boundary keys;
return D;
```

input size constraint $|S| > (|V_C| - 1) \cdot |R|$. As the next lemma shows, the above algorithm for the general case has cost that matches the lower bound for set intersection.

Lemma 4.4. *Algorithm 2 has a cost of* $O(C_{\text{int}})$.

Let K be the set of boundary keys. It must be that $|K| = O(|V_C|)$. Define $R_b = \sigma_{B=b}(R)$ and $S_b = \sigma_{B=b}(S)$. We next will show how to compute the Cartesian products $R_b \times S_b$ for each $b \in K$.

4.5 Step 2: Handling the Boundary Keys

The second step of the join algorithm handles the join computation of the boundary keys with their input tuples as in the initial data distribution. To simplify the exposition, we will assume w.l.o.g. that for every $b \in K$, $|R_b| \le |S_b|$. The sizes $|R_b|, |S_b|$ for each boundary key $b \in K$ can be easily computed by invoking the topology-aware multi-numbering primitive in Appendix D, and then broadcast to all compute nodes in V_C. Recall that G has a β-tree as its core, with a set of α-trees $\{G_\alpha^1, G_\alpha^2, \cdots, G_\alpha^\ell\}$ incident to the leaf nodes of β-tree. As described in Algorithm 3, it consists of three steps:

- **(Step 2.1)** For each G_α^j, we let all compute nodes in G_α^j send their S-tuples to the root of $\vec{G}_\alpha^j$.
- **(Step 2.2)** We redistribute S-tuples across β-tree by invoking the redistribution primitive in Section 2. Let $S_{b,j}^f$ be the set of S-tuples for key b residing at the root of G_α^j after this step.

- **(Step 2.3)** Each G^j_α is responsible for computing $R^f_{b,j} \times S^f_{b,j}$ for key $b \in K$, where $R^f_{b,j}$ is a subset of R-tuples with key b. We invoke the Cartesian product packing primitive in Section 3 inside $\vec{G}^j_\alpha$.

It remains to identify $R^f_{b,j}$ and $S^f_{b,j}$ for each G^j_α and key $b \in K$. Our algorithm assigns to each G^j_α tree a *fractional value* $f_{b,j} \in [0,1]$ such that $\sum_{j\in[\ell]} f_{b,j} = 1$. We will discuss later how to compute these fractions. Intuitively, each G^j_α will be responsible to compute $f_{b,j} \cdot |R_b| \cdot |S_b|$ pairs for $R_b \times S_b$. More specifically, for G^j_α:

- if $f_{b,j} \geq |R_b|/|S_b|$, we assign the computation of a *rectangle* $\blacksquare_{b,j}$ with dimensions $|R_b| \times \lceil 2 \cdot f_{b,j} \cdot |S_b| \rceil$.
- otherwise, we assign the computation of a *square* $\blacksquare_{b,j}$ with size $(2 \cdot f_{b,j} \cdot |R_b| \cdot |S_b|)^{1/2}$ rounded up to the closest power of 2.

Now, suppose we are given such a fraction function f, as well as the rectangles/squares $\blacksquare_{b,j}$ for each G^j_α and $b \in K$. We will show for each boundary key $b \in K$, how to organize/pack the above squares/rectangles inside the Cartesian product $R_b \times S_b$ so that they fully cover every pair of $R_b \times S_b$. This packing procedure is conceptually the reverse of assigning tuples for $R_{b,j}$ and $S^f_{b,j}$.

Packing over the β-tree. To construct a packing for a boundary key $b \in K$, we will again deploy the tree partition algorithm (Algorithm 1), where $\tau = |R_b|/|S_b|$ and the function θ for a subset leaf nodes $P \subseteq \mathcal{L}_\beta$ is defined as sum of the fractions $f_{b,j}$ such that the associated directed α-tree is rooted at some node in P, i.e.,

$$\theta(P) = \sum_{u \in P : \vec{G}^j_\alpha \text{ is rooted at } u} f_{b,j}.$$

We slightly modify Algorithm 1 such that in the last possible step when we are left with two vertices u, v, we do not merge u with $\theta(\Gamma(u)) < \tau$ if its parent v has $\theta(\Gamma(v)) > \tau$. This modification means that we could have a block P_0 with $\theta(P_0) \leq \tau$, but there can be at most one of such blocks. We refer this modified version as TreePartition'. Let $\mathcal{P}$ be the resulting partition.

Now, we show how to pack these squares/rectangles over the β-tree based on $\mathcal{P}$. Note that a rectangle can appear in a block of $\mathcal{P}$ with a single α-tree (since it contributes more than τ to the total weight). Hence, any block that has more than one tree contains only squares. Since by construction the squares have dimensions that are powers of two, we can use the construction in [20] (Lemma 10) to pack the squares of any block $P \in \mathcal{P} \setminus \{P_0\}$ such that they fully cover a square of size at least: $\frac{1}{2}\sum_{j\in P}(2 \cdot f_{b,j} \cdot |R_b| \cdot |S_b|) \geq |R_b| \cdot |S_b| \cdot \tau \geq |R_b|^2$. Hence, any block $P \in \mathcal{P} \setminus \{P_0\}$ can fully pack a rectangle of dimensions $|R_b| \times \kappa_P$, where $\kappa_P \geq |R_b|$. We can now put these rectangles horizontally next to each other. Let κ be the horizontal length covered by these rectangles. We do not use the specific block P_0 (if it exists) to cover any part of the $R_b \times S_b$ grid.

We will next argue that the rectangles will fully cover the $R_b \times S_b$ grid, or equivalently, $\kappa \geq |S_b|$. We can now rewrite $|R_b| \cdot |S_b| = \sum_{j\in[\ell]} f_{b,j} \cdot |R_b| \cdot |S_b|$ as $\sum_{j\in[\ell]} f_{b,j} = 1$. By further distinguishing which block j falls into, we can expand it as

$$|R_b| \cdot |S_b| = \sum_{j\in P_0} f_{b,j} \cdot |R_b| \cdot |S_b| + \sum_{P\in\mathcal{P}\setminus\{P_0\}} \sum_{j\in P} f_{b,j} \cdot |R_b| \cdot |S_b|.$$

Recall that $\theta(P_0) \leq \tau$, so $\sum_{j\in P_0} f_{b,j} \cdot |R_b| \cdot |S_b| \leq |R_b|^2$. We obtain: $|R_b| \cdot |S_b| \leq |R_b|^2 + \frac{1}{2} \cdot |R_b| \cdot \kappa$ Hence, $\kappa \geq 2|S_b| - |R_b| \geq |S_b|$.

Redistributing S-tuples over β-tree. It remains to specify how to send the necessary tuples for each $\blacksquare_{b,j}$ to the root of the corresponding α-tree G^j_α. Observe that the root of each G^j_α can receive for each boundary key $b \in K$ all tuples from R_b with a cost that matches the lower bound C_{int}. Indeed, the total number of such tuples is $\sum_{b\in K} |R_b| \leq |R|$, so they can be routed over the β-edges without additional cost. Hence, it suffices to determine how to (re)-distribute the tuples from S_b.

Algorithm 3: BoundaryKey$(G, \mathcal{D}, K, f)$

Let $\{G_\alpha^1, G_\alpha^2, \cdots, G_\alpha^\ell\}$ be the set of α-trees of G;
foreach $j \in [\ell]$ **do**
  Let nodes in $V_C \cap G_\alpha^j$ send S-tuples to the root of $\vec{G}_\alpha^j$;
foreach $b \in K$ **do**
  $\mathcal{P} \leftarrow$ TreePartition'$(G_\beta, \theta, |R_b|/|S_b|)$;
  Run data redistribution primitive on Λ (4) and Λ^f (5);
  foreach $P \in \mathcal{P}$ **do**
    Assign a rectangle $\blacksquare_P$ of R and $\sum_{j \in P} \Lambda^f_{b,j}$ S-tuples;
    if $|P| > 1$ **then**
      foreach $u_j \in P$ **do**
        Assign a square $\blacksquare_{b,j}$ of $\Lambda^f_{b,j}$ R-tuples and S-tuples from $\blacksquare_P$ to $\vec{G}_\alpha^j$;
    else Assign $\blacksquare_P$ to u_j as $\blacksquare_{b,j}$ for $P = \{u_j\}$;
foreach $j \in [\ell]$ **do**
  Run Cartesian product packing primitive (Section 3) on $\vec{G}_\alpha^j$ for computing the assigned $\blacksquare_{b,j}$ for every $b \in K$;

For this, we can first route the S-tuples from the compute nodes in $V_C \cap G_\alpha^j$ to the root of the directed version $\vec{G}_\alpha^j$ with a cost at most C_{int}; this is because all edges are directed towards the root. Now, all S_b tuples are sitting at $\mathcal{L}_\beta$, i.e., the leaf nodes of the β-tree, which are exactly the roots of the directed α-trees.

We next use the routine (Section 2) to redistribute S_b-tuples across $\mathcal{L}_\beta$. For simplicity, let $\mathcal{L}_\beta = \{u_1, \ldots, u_\ell\}$ where u_i is the root of $\vec{G}_\alpha^i$. The input size-constraint matrix Λ is defined as:

$$\Lambda_{b,j} = \sum_{v \in V_C \cap G_\alpha^j} |S_{b,v}| \tag{4}$$

where $b \in K$ ranges over all boundary keys and $j \in [\ell]$ ranges over all root nodes of α-trees. Note that $\sum_{j \in [\ell]} \Lambda_{b,j} = |S_b|$. The new distribution Λ^f under the fractional allocation f is defined as follows. Suppose u_j (the root node of $\vec{G}_\alpha^j$) belongs to block $P \in \mathcal{P}$.

$$\Lambda^f_{b,j} = \begin{cases} \lceil 2 \cdot f_{b,j} \cdot |S_b| \rceil, & \text{if } |P| = 1 \\ \sqrt{\dfrac{2 \cdot f_{b,j}}{\sum_{i:u_i \in P} f_{b,i}}} \cdot |R_b|, & \text{otherwise} \end{cases} \tag{5}$$

Lemma 4.5. *The following properties hold for Λ^f:*

- $\frac{\sqrt{2}}{2} \cdot |\blacksquare_{b,j}| \le \Lambda^f_{b,j} \le |\blacksquare_{b,j}|$ *for every $b \in K$ and $j \in [\ell]$, where $|\blacksquare_{b,j}|$ is the number of S-tuples in the square/rectangle $\blacksquare_{b,j}$;*
- $|S_b| \le \sum_{j \in [\ell]} \Lambda^f_{b,j} \le c \cdot |S_b|$ *for some constant c.*

Proof of Lemma 4.5. For the first property, notice that the inequality holds with equality if $\blacksquare_{b,j}$ is a rectangle. Otherwise, we observe that: $R_b|/|S_b| \le \sum_{i:G_\alpha^i \in P} f_{b,i} \le 2 \cdot |R_b|/|S_b|$ from the property of

a balanced distribution in Definition 4.2. Thus, we conclude that

$$(f_{b,i} \cdot |R_b| \cdot |S_b|)^{1/2} \leq \Lambda^f_{b,j} \leq (2 \cdot f_{b,j} \cdot |R_b| \cdot |S_b|)^{1/2}.$$

The second property holds since we have scaled down the fractional allocation of each α-tree in a "square" block such that the sum of the S-tuples is within a constant factor of $|R_b|$. □

The second property implies that we can "pad" the initial distribution of data to match exactly $\sum_{j\in[\ell]} \Lambda^f_{b,j}$ without losing more than a constant factor. Now we can apply the redistribution protocol as is. Finally, we note that $\Lambda^f_{b,j} = |\blacksquare_{b,j}|$ holds for a rectangle $\blacksquare_{b,j}$, hence the α-tree will receive all the necessary tuples for the Cartesian product computation. However, if $\blacksquare_{b,j}$ is a square then it could be that $\Lambda^f_{b,j} < |\blacksquare_{b,j}|$. Suppose $u_j \in P$ for some block $P \in \mathcal{P}$. But since the total number of S-tuples distributed over the α-trees rooted at some node in P are $O(|R_b|)$ (this follows the property of a balanced partition in Definition 4.2), we can afford to send this much data through a β-edge, and thus we simply broadcast the necessary data in the minimum Steiner tree induced by P.

Optimal Fractions. Finally, we show how to find the optimal fraction f^* by iterating over all possible sizes of the grid allocation in $[1, |R_b| \cdot |S_b|]$ in powers of two. Hence, we only need to consider $\log(|R_b| \cdot |S_b|) = O(\log N)$ possible values for every fraction $f_{b,j}$, and only lose a constant factor in optimality. The f^* is computed locally in each compute node without incurring any data transmission cost. Putting everything together, we obtain:

Theorem 4.6. *Given a symmetric tree topology $G = (V, E)$, and any data distribution $\mathcal{D}$ for $R(A, B) \bowtie S(B, C)$, their join results can be computed in $O(1)$ rounds with cost $O\left(C_{\text{int}} + C_{pack}\right)$, for*

$$C_{pack} = \min_f \left\{ \max_{e\in E_\beta} \frac{\Delta_e}{w_e} + \max_{j\in[\ell]} L^* \left(\vec{G}^j_\alpha, \{\blacksquare_{b,j} : b \in K\} \right) \right\},$$

where Δ_e is the tuple differential defined over Λ in (4) and Λ^f in (5), and $L^*(\vec{G}^j_\alpha, \{\blacksquare_{b,j} : b \in K\})$ is the optimal solution of (1) defined on the directed tree $\vec{G}^j_\alpha$ and a set of Cartesian products $\blacksquare_{b,j}$ for $b \in K$.

4.6 Optimality

The topology-aware-optimality of Theorem 4.6 is established via the lower bound below, whose proof is given in Appendix E. The high-level idea is to construct two sub-instances $\mathcal{D}_1$ and $\mathcal{D}_2$ with disjoint domains such that $\mathcal{D}_1 \cup \mathcal{D}_2$ conforms to ϕ and OUT. We show the lower bound $\Omega(C_{\text{int}})$ for computing $\mathcal{D}_1$ via an reduction from the lopsided set disjointness problem. Conditioning on $\Omega(C_{\text{int}})$, we show the other lower bound $\Omega(\max_{\mathcal{D}'} C_{\text{pack}})$ for computing $\mathcal{D}_2$.

Theorem 4.7. *Given a symmetric tree topology $G = (V, E)$, a cardinality distribution ϕ and output size* OUT*, there is a data distribution $\mathcal{D}$ for $R(A, B) \bowtie S(B, C)$ conforming to ϕ and* OUT*, that any algorithm computing the join results incurs a cost of $\Omega\left(C_{int} + \max_{\mathcal{D}'} C_{pack}\right)$, where $\mathcal{D}'$ is over all data distributions conforming to ϕ and* OUT*.*

4.7 Simplification of the Algorithm on Easier Input

Below, we discuss some interesting cases when the underlying topology, the initial data distributions, or the join instance displays simpler properties.

Uniform Data Distribution. Assume each compute node holds the same amount of data, which is a common occurrence in practice. W.l.o.g., assume $|R| \leq |S|$. Recall one of our lower bounds C_{int} for the cost of set intersection. We simply distinguish two cases:

- If $|R| + |S| \le |R| \cdot |V_C|$, we have $E_\alpha = E$ and $E_\beta = \emptyset$. In this case, the network G itself forms a single α-tree, and our simplified algorithm in Section 4.2 directly applies. This case incurs a cost of $O(\mathsf{C}_{\mathsf{int}} + \mathsf{C}_{\mathsf{pack}})$, where $\mathsf{C}_{\mathsf{pack}}$ is the optimal solution of (1) parameterized by $\{|R_b|, |S_b| : b \in B\}$.
- If $|R| + |S| > |R| \cdot |V_C|$, we have $E_\alpha = \emptyset$ and $E_\beta = E$. In this case, the network G itself is the β-tree, and our algorithm just sends all R-tuples to every compute node. This case incurs a cost of $O\left(\max_{e\in E} \frac{|R|}{w_e}\right)$.

Symmetric star topology. Assume the underlying topology is a symmetric star $G = (V, E)$. Let $o \in V$ be the router. W.l.o.g., assume $|R| \le |S|$. We partition compute nodes in V_C into V_α and V_β:

$$V_\alpha = \{v \in V_C : |R| > \min\{N_v, N - N_v\}\}, V_\beta = V_C - V_\alpha$$

Equivalently, E_α (resp. E_β) is the set of edges incident to compute nodes in V_α (resp. V_β). In the tree partition $\mathcal{P}$ of G, every node in V_β forms an individual block, and all nodes in V_α form one block. Our algorithm can be simplified as follows:

(1) We let all compute nodes send R-tuples to V_β and let nodes in V_β compute local join results.
(2) We sort $R \cup \left(\cup_{v\in V_\alpha} S_v\right)$ over $V_\alpha \cup \{o\}$, and let each node in V_α compute the local join results with non-boundary keys.
(3) We let all compute nodes in V_α send their S-tuples of boundary keys to the router o.
(4) Consider an arbitrary boundary key $b \in K$ and its Cartesian product $R_b \times S_b$. Consider an arbitrary f. The router o simply forwards $\Lambda^f_{b,v}$ arbitrary S-tuples with key $b \in K$ to every $v \in V_\beta$; and a square $\blacksquare_{b,v}$ of $\Lambda^f_{b,v}$ R-tuples and S-tuples with key $b \in K$ to every $v \in V_\alpha$. After receiving the data, all nodes compute the local join results.

Cartesian product. Assume there is a single boundary key value in the join attribute. In this case, the join $R \bowtie S$ degenerates to the Cartesian product $R \times S$. Assume $|R| \le |S|$ without loss of generality. The algorithm can be simplified as follows:

(1) We first construct a balanced partition $\mathcal{P} = \{P_1, P_2, \ldots, P_k\}$ by Algorithm 1. Recall that G has a β-tree as its core, with a set of α-trees $\{G^1_\alpha, G^2_\alpha, \cdots, G^\ell_\alpha\}$ incident to the leaf nodes of β-tree.
(2) For each G^j_α, we let all compute nodes send their S-tuple to the root of $\vec{G}^j_\alpha$, and their R-tuples to every root node of subtrees.
(3) Consider an arbitrary f. We redistribute S-tuples across β-tree by invoking the redistribution primitive in Section 2. Assume S^f_j is the set of S-tuples residing at the root of G^j_α after this step. Let R^f_j be the set of R-tuples assigned to G^j_α.
(4) Each G^j_α is responsible for computing $R^f_j \times S^f_j$ inside the subtree using the Cartesian product packing primitive in Section 3.

5 DISCUSSIONS

Below, we discuss several interesting questions related to optimality.

Worst-case Data Distribution. If targeting optimality with respect to the worst-case data distribution, things becomes easy. We next show a simple algorithm for computing $R(A, B) \bowtie S(B, C)$ over a symmetric tree topology $G = (V, E)$. W.l.o.g., assume $|R| \le |S|$. If $|R| = |S|$, we simply pick one arbitrary node, say v and let all nodes send their data to v. If $|R| < |S|$, we let every compute node broadcast its R-tuples. The cost of this algorithm is bounded by $O(\max_{e\in E} \frac{|R|}{w_e})$. Meanwhile, consider an arbitrary cardinality distribution ϕ such that $\sum_{v\in G^-_{e^*}} N_v = \sum_{v\in G^+_{e^*}} N_v = \frac{1}{2}(|R| + |S|)$ for $e^* = \arg\max_{e\in E} w_e$. Implied by Lemma E.1, there exists an instance conforming to ϕ such that any algorithm must incur a cost of $\Omega(C_{\mathsf{int}}) = \Omega(\frac{|R|}{w_{e^*}})$. Thus, this algorithm is already optimal.

New challenges by Downstream Applications. For complicated analytical queries, such as acyclic joins and join-aggregate query, we can apply the standard pairwise framework by incorporating binary join as a primitive, but achieving the optimality with respect to the cardinality of initial data distribution becomes quite challenging.

Consider a join-aggregate query $\sum_B R(A,B) \bowtie S(B,C)$. A basic solution is to materialize the result of binary join and then aggregate the join results over B by invoking the topology-aware reduce-by-key primitive (in Appendix D). However, the distribution of binary join results will serve as the input distribution of the subsequent aggregation. If incorporating our join algorithm without considering subsequent aggregation, it may not be an overall optimal algorithm.

Consider an acyclic join. A basic solution is to remove all dangling tuples (i.e., those won't participate in any join results) by topology-aware semi-joins (in Appendix D) and then perform pairwise join. In the sequential RAM model as well as the homogeneous MPC model, the pairwise join ordering does not matter, as long as the size of intermediate join result can be bounded by that of final join result. However, this ordering makes a big difference in our topology-aware model, since the distribution of intermediate join results will serve as the initial distribution of an input relation in the subsequent join. Incorporating our binary join algorithm blindly to subsequent joins may not be optimal. Moreover, it would be very costly to pre-compute the best pairwise join ordering locally, since the distribution of intermediate join results could require full information of input data to compute.

6 RELATED WORK

Topology-aware models have been widely studied in parallel and distributed computations, such as the classical LOCAL/ CONGEST model [29, 30] and their variants [11, 28]. These models are different from ours in the following aspects: (i) all these models take both delay and congestion cost into account, such that limited size of data can only be exchanged between neighbor nodes, while our model only consider the congestion cost, such that any pair of nodes can exchange data in a single round as long as there exists a valid routing path; (ii) LOCAL/CONGEST model does not assume the knowledge of network topology, while our model as well as [11, 28] assume that the algorithm has complete knowledge of network topology; (iii) [28] assumes that one relation completely resides within one node while our model considers the case where relations are split or distributed across compute nodes; (iv) [11, 28] consider the worst-case distribution of input data while we consider the cardinality of initial data distribution as input parameters, thus being able to prove more fine-grained optimality. A hierarchical version of the BSP model has also been studied as D-BSP model [8, 9, 13]. In D-BSP model, nodes are the leaves of a complete binary tree, on which "clusters" are defined. In the i-th round, nodes can only communicate within the same i-level cluster. But, no such hierarchy exists in our topology-aware model since any pair of nodes with a routing path can communicate in a single round. In addition, nodes communicating within some specific cluster in D-BSP have the same transmission cost, while links in our topology-aware model can have arbitrary bandwidth cost, regardless of the topology structure. Later, Chattopadhyay et al. [11, 12] studied topology-dependent round complexity for k-party functions (e.g., set-disjointness and element distinctness) in a similar model as CONGEST. Langberg et al. [28] further improved the round complexity for functional aggregate queries. Topology-aware aggregation has been considered in other systematic works, but without theoretical guarantees. We refer interested readers to [10, 20] for more details.

REFERENCES

[1] Foto N Afrati, Manas R Joglekar, Christopher M Re, Semih Salihoglu, and Jeffrey D Ullman. 2017. GYM: A multiround distributed join algorithm. In *ICDT*.

[2] Pankaj K Agarwal, Kyle Fox, Kamesh Munagala, and Abhinandan Nath. 2016. Parallel algorithms for constructing range and nearest-neighbor searching data structures. In *PODS*. 429–440.

[3] Alexandr Andoni, Aleksandar Nikolov, Krzysztof Onak, and Grigory Yaroslavtsev. 2014. Parallel algorithms for geometric graph problems. In *STOC*. 574–583.

[4] Sepehr Assadi, Xiaorui Sun, and Omri Weinstein. 2019. Massively parallel algorithms for finding well-connected components in sparse graphs. In *PODC*. 461–470.

[5] Rafael da Ponte Barbosa, Alina Ene, Huy L Nguyen, and Justin Ward. 2016. A new framework for distributed submodular maximization. In *FOCS*. IEEE, 645–654.

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

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

[8] Gianfranco Bilardi, Carlo Fantozzi, Andrea Pietracaprina, and Geppino Pucci. 2001. On the effectiveness of D-BSP as a bridging model of parallel computation. In *ICCS*. Springer, 579–588.

[9] Gianfranco Bilardi, Andrea Pietracaprina, Geppino Pucci, and Francesco Silvestri. 2007. Network-oblivious algorithms. In *IPDPS*. IEEE, 1–10.

[10] Spyros Blanas, Paraschos Koutris, and Anastasios Sidiropoulos. 2020. Topology-aware parallel data processing: Models, algorithms and systems at scale. In *CIDR*.

[11] Arkadev Chattopadhyay, Michael Langberg, Shi Li, and Atri Rudra. 2017. Tight network topology dependent bounds on rounds of communication. In *SODA*. 2524–2539.

[12] Arkadev Chattopadhyay, Jaikumar Radhakrishnan, and Atri Rudra. 2014. Topology matters in communication. In *FOCS*. IEEE, 631–640.

[13] Pilar De la Torre and Clyde P Kruskal. 1996. Submachine locality in the bulk synchronous setting. In *Euro-Par'96 Parallel Processing: Second International Euro-Par Conference Lyon, Volume II 2*. Springer, 352–358.

[14] Jeffrey Dean and Sanjay Ghemawat. 2008. MapReduce: simplified data processing on large clusters. *Commun. ACM* 51, 1 (2008), 107–113.

[15] Mohsen Ghaffari, Themis Gouleakis, Christian Konrad, Slobodan Mitrović, and Ronitt Rubinfeld. 2018. Improved massively parallel computation algorithms for mis, matching, and vertex cover. In *PODC*. 129–138.

[16] Mohsen Ghaffari and Krzysztof Nowicki. 2020. Massively parallel algorithms for minimum cut. In *PODC*. 119–128.

[17] Michael T Goodrich, Nodari Sitchinava, and Qin Zhang. 2011. Sorting, searching, and simulation in the mapreduce framework. In *ISAAC*. Springer, 374–383.

[18] Guanhao Hou, Xingguang Chen, Sibo Wang, and Zhewei Wei. 2021. Massively parallel algorithms for Personalized PageRank. *Proceedings of the VLDB Endowment* 14, 9 (2021), 1668–1680.

[19] Xiao Hu. 2021. Cover or pack: New upper and lower bounds for massively parallel joins. In *PODS*. 181–198.

[20] Xiao Hu, Paraschos Koutris, and Spyros Blanas. 2021. Algorithms for a Topology-aware Massively Parallel Computation Model. In *PODS*. ACM, 199–214.

[21] Xiao Hu and Ke Yi. 2019. Instance and output optimal parallel algorithms for acyclic joins. In *PODS*. 450–463.

[22] Xiao Hu and Ke Yi. 2020. Massively Parallel Join Algorithms. *ACM SIGMOD Record* 49, 3 (2020), 6–17.

[23] Xiao Hu and Ke Yi. 2020. Parallel Algorithms for Sparse Matrix Multiplication and Join-Aggregate Queries. In *PODS*. 411–425.

[24] Xiao Hu, Ke Yi, and Yufei Tao. 2019. Output-Optimal Massively Parallel Algorithms for Similarity Joins. *TODS* 44, 2 (2019), 6.

[25] Bas Ketsman and Dan Suciu. 2017. A worst-case optimal multi-round algorithm for parallel computation of conjunctive queries. In *PODS*. ACM, 417–428.

[26] Paraschos Koutris, Paul Beame, and Dan Suciu. 2016. Worst-case optimal algorithms for parallel query processing. In *ICDT*.

[27] Paraschos Koutris and Dan Suciu. 2011. Parallel evaluation of conjunctive queries. In *PODS*. ACM, 223–234.

[28] Michael Langberg, Shi Li, Sai Vikneshwar Mani Jayaraman, and Atri Rudra. 2019. Topology dependent bounds for faqs. In *PODS*. 432–449.

[29] Nathan Linial. 1992. Locality in distributed graph algorithms. *SIAM Journal on computing* 21, 1 (1992), 193–201.

[30] David Peleg. 2000. *Distributed computing: a locality-sensitive approach*. SIAM.

[31] Leslie G Valiant. 1990. A bridging model for parallel computation. *Commun. ACM* 33, 8 (1990), 103–111.

[32] Tao Yufei. 2020. A Simple Parallel Algorithm for Natural Joins on Binary Relations. ICDT.

[33] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauly, Michael J Franklin, Scott Shenker, and Ion Stoica. 2012. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. In *NSDI*. 15–28.

A MISSING PROOF IN SECTION 2

Proof of Lemma 2.3. Indeed, we can write:

$$\begin{aligned}\Delta^i_e - \Delta^i_{e'} &= \max\{0, \sum_{v_j \in G^+_e} \Lambda'_{ij} - \sum_{v_j \in G^+_e} \Lambda_{ij}\} - \max\{0, \sum_{v_j \in G^+_{e'}} \Lambda'_{ij} - \sum_{v_j \in G^+_{e'}} \Lambda_{ij}\} \\ &= \max\{0, \sum_{v_j \in G^+_e} \Lambda'_{ij} - \sum_{v_j \in G^+_e} \Lambda_{ij}\} - \max\{0, (\Lambda_i - \sum_{v_j \in G^+_e} \Lambda'_{ij}) - (\Lambda_i - \sum_{v_j \in G^+_e} \Lambda_{ij})\} \\ &= \max\{0, \sum_{v_j \in G^+_e} \Lambda'_{ij} - \sum_{v_j \in G^+_e} \Lambda_{ij}\} - \max\{0, \sum_{v_j \in G^+_e} \Lambda_{ij} - \sum_{v_j \in G^+_e} \Lambda'_{ij}\} = \sum_{v_j \in G^+_e} \Lambda'_{ij} - \sum_{v_j \in G^+_e} \Lambda_{ij}\end{aligned}$$

This completes the proof. □

Lemma A.1. *The assignment $f_i(e) = \Delta^i_e$ is a flow with an optimal cost of* $\max_e \Delta^i_e / w_e$.

Proof of Lemma A.1. The cost calculation is straightforward, so we will provide a proof that $f_i(e) = \Delta^i_e$ is indeed a flow. Let v be a node in V. We will consider two cases. First, let v be a leaf node (say v_j) and let u be the unique node incident to v in G. Take any relation R_i. The total incoming flow to v_j is $\Lambda_{ij} + \Delta^i_{(u,v)} = \Lambda_{ij} + \max\{0, \Lambda'_{ij} - \Lambda_{ij}\}$. The total outgoing flow from v_j is $\Lambda'_{ij} + \Delta^i_{(v,u)} = \Lambda'_{ij} + \max\{0, \Lambda_{ij} - \Lambda'_{ij}\}$. One can check that these quantities are always equal. Now, let v be an internal node. Let U the set of nodes incident to v. Then, we can write:

$$\begin{aligned}\sum_{u \in U} f_i(v,u) - \sum_{u \in U} f_i(u,v) &= \sum_{u \in U} \left(\Delta^i_{(v,u)} - \Delta^i_{(u,v)}\right) = \sum_{u \in U} \left(\sum_{v_j \in G^+_{(v,u)}} \Lambda'_{ij} - \sum_{v_j \in G^+_{(v,u)}} \Lambda_{ij} \right) \\ &= \sum_{u \in U} \sum_{v_j \in G^+_{(v,u)}} \Lambda'_{ij} - \sum_{u \in U} \sum_{v_j \in G^+_{(v,u)}} \Lambda_{ij} = \Lambda_i - \Lambda_i = 0\end{aligned}$$

where the second-to-last equality is implied by the fact that $\{G^+_{(v,u)}\}_u$ are disjoint trees and cover all leaf nodes in G. □

B MISSING PROOFS IN SECTION 4

Proof of Lemma 4.3. Let $\mathcal{P}$ be the set returned by the algorithm. It is easy to see that the sets in $\mathcal{P}$ are disjoint. To prove that $\mathcal{P}$ is partition, we need to show that every leaf node in $\mathcal{L}_\beta$ will end up in some set of $\mathcal{P}$. The only issue may occur when we are left with a single vertex x: we claim that in this case we always have $\theta(\Gamma(x)) \geq \tau$. Indeed, if $\Gamma(x) = \mathcal{L}_\beta$, then this follows from our lemma assumption. Otherwise, suppose $\theta(\Gamma(x)) < \tau$, and consider the last vertex u for which $\Gamma(u)$ was added in $\mathcal{P}$. But then, the algorithm could not have picked u at this point, a contradiction.

To prove the first property, assume that there is an edge $e = (u, v)$ that appears in minimum Steiner trees of P_1, P_2. Then, there exist $x, y \in P_1$, $x', y' \in P_2$ such that $x, x' \in G^-_e$ and $y, y' \in G^+_e$. W.l.o.g., assume that u is visited before v. Then, since x, x' are placed in different blocks of the partition, $\Gamma(u)$ contains only one of x, x', say x. But then x' must be already put in some block with vertices of G^-_e. But then x', y' cannot be placed in the same block, a contradiction.

The second property is straightforward, since the algorithm adds a new set in $\mathcal{P}$ only if its total weight exceeds τ.

For the third property, we consider a set $P \in \mathcal{P}$ and a β-edge $e = (u, v)$ in its Steiner tree. W.l.o.g., assume that u is visited before v. At this point, it must be that $\theta(\Gamma(u)) < \tau$, since $\Gamma(u)$ was merged with $\Gamma(v)$. We also have that $\Gamma(u) = P \cap G^-_e$, since no other leaf nodes will be added to the "left" of u. Hence, $\min\{\theta(P \cap G^+_e), \theta(P \cap G^-_e)\} \leq \theta(P \cap G^-_e) < \tau$.

To prove the last property, consider a block $P \in \mathcal{P}$. When P is added to $\mathcal{P}$, it must be chosen by line 5. We consider the last iteration when y is chosen in line 9 and $\Gamma(y)$ is updated to P by merging with $\Gamma(x)$. In that case, $\theta(\Gamma(y)) < \tau$ must hold; otherwise, $\Gamma(y)$ must be added to $\mathcal{P}$ as a separate block. Similar argument applies to $\Gamma(x)$. So, $\theta(P) = \theta(\Gamma(x)) + \theta(\Gamma(y)) < 2\tau$. □

C SORTING-BASED SET INTERSECTION

We can borrow the analysis in [20] to prove Lemma 4.4. Equivalently, it suffices to show the cost of Algorithm 2 as:

$$C_{\text{int}} = \max\left\{\max_{e\in E_\alpha}\frac{1}{w_e}\cdot\min\left\{\sum_{v\in G_e^-}N_v, \sum_{v\in G_e^+}N_v\right\}, \max_{e\in E_\beta}\frac{|R|}{w_e}\right\}$$

Let $\mathcal{P} = \{P_1, \ldots, P_k\}$ be the balanced partition of the leaf nodes of G_β. We define V_C^i for $i \in [k]$ to be the set of compute nodes in the α-trees rooted at some node of P_i. In the hashing phase, the number of R-tuples that go through $e \in E_\beta$ is at most $|R|$, so it suffices to bound the number of R-tuples that go through $e \in E_\alpha$.

- if none of G_e^-, G_e^+ contain β-edges. Then, the partition consists of a single block, and the number of R-tuples is bounded as:

$$\frac{1}{\sum_{v\in V_C}N_v}\cdot\Big(\sum_{v\in G_e^-}N_v\Big)\cdot\Big(\sum_{v\in G_e^+}N_v\Big) \le \min\{\sum_{v\in G_e^-}N_v, \sum_{v\in G_e^+}N_v\}$$

- G_e^+ contains β-edges but G_e^- not. All vertices in G_β are in G_e^+. The R-data that goes through e is sent by nodes in G_e^-, so its size is bounded by $\sum_{v\in G_e^-}|R_v| \le \sum_{v\in G_e^-}N_v = \min\left\{\sum_{v\in G_e^-}N_v, \sum_{v\in G_e^+}N_v\right\}$. Here, the last equality follows from the fact that G_e^+ contains at least one β-edge, which implies $\sum_{v\in G_e^+}N_v \ge |R| > \sum_{v\in G_e^-}N_v$.
- G_e^- contains β-edges but G_e^+ not. Then, all nodes in G_e^+ belong in the same block V_C^i. We can abound the expected amount of R-tuples with:

$$\begin{aligned}\frac{1}{\sum_{v\in V_C^i}N_v}\cdot\Big(\sum_{v\in G_e^-}|R_v|\Big)\cdot\Big(\sum_{v\in V_C^i\cap G_e^+}N_v\Big) &\le \frac{\sum_{v\in G_e^-}|R_v| + \sum_{v\in V_C^i\cap G_e^+}N_v}{\sum_{v\in V_C^i}N_v}\min\{\sum_{v\in G_e^-}|R_v|, \sum_{v\in V_C^i\cap G_e^+}N_v\}\\ &\le \frac{|R| + \sum_{v\in V_C^i}N_v}{\sum_{v\in V_C^i}N_v}\min\{\sum_{v\in G_e^-}N_v, \sum_{v\in G_e^+}N_v\}\\ &\le 2\min\{\sum_{v\in G_e^-}N_v, \sum_{v\in G_e^+}N_v\}\end{aligned}$$

  where the last inequality is from the definition of a balanced partition.

We next focus on the cost of local sorting inside each block. For each compute node $v \in V_C$, let e_v be the unique edge incident to v. Note that if $e_v \in E_\beta$, then the number of R-tuples received from the hashing phase is $|R| \le N_v$ implied by the definition of β-edge. If $e_v \in E_\alpha$, the number of R-tuples received from the hashing phase is $\min\{N_v, N - N_v\} \le N_v$. Hence, each node v has received $O(N_v)$ tuples after hashing. In the local sorting over the Steiner tree of V_C^i, implied by the result in [20] (Theorem 17), the number of tuples goes through edge e is

$$M = \min\left\{\sum_{v\in G_e^-\cap V_C^i}N_v, \sum_{v\in G_e^+\cap V_C^i}N_v\right\} \tag{6}$$

Similarly, (6) is bounded by $|R|$ for $e \in E_\beta$, which is implied by the definition of a balanced partition. Also, the inequality $M \le \min\{\sum_{v \in G_e^-} N_v, \sum_{v \in G_e^+} N_v\}$ for $e \in E_\alpha$ follows directly.

D TOPOLOGY-AWARE PRIMITIVES

All Prefix-Sums. Given an array A of elements $A[1], A[2], \cdots, A[N]$ which are distributed according to the ordering π, the goal is to compute $S[i] = A[1] \oplus A[2] \oplus \cdots \oplus A[i]$ for all $i = 1, 2, \cdots, N$, where $\oplus$ is any associative operator. Assume there exists some compute node with its initial data size larger than $|V_C|$, say u.[11] Let $A[s_i : t_i]$ be the sub-array located at node v_i after sorting. Each node v_i first computes $B[i] = A[s_i] \oplus A[s_i + 1] \oplus \cdots \oplus A[t_i]$ and send it to u. After receiving all $B[i]$, u performs local computation and sends back to v_i the result of $B[1] \oplus B[2] \oplus \cdots \oplus B[i]$, for all $i \in [|V_C|]$. After receiving $B[i-1]$, node v_i computes $S[j] = B[i-1] + A[s_i] + A[s_i + 1] + \cdots + A[j]$ for each $j \in [s_i, t_i]$.

Reduce-By-Key. Given a set of (key, value) pairs and an associate operator $\oplus$, the goal is to for each key k compute the "sum" of corresponding values under $\oplus$. We adapt the algorithm [24] for solving reduce-by-key over MPC model based on prefix-sum. We first sort the pairs by their keys by invoking the algorithm in [20]. The i-th tuple in the sorted order will produce a pair (x, y), which will act as $A[i]$. For each tuple that is the first of its key in the sorted order, we produce the pair $(0, \text{value})$; otherwise, we produce $(1, \text{value})$. Note that we need another round of communication to determine whether each tuple is the first of its key, in case that its predecessor resides on another server. Then we define the operator $\oplus$ as $(x_1, y_1) \oplus (x_2, y_2) = (x_1 \cdot x_2, y)$, where $y = y_1 \oplus y_2$ if $x_2 = 1$ and otherwise $y = y_2$. After solving the all prefix-sums problem on the derived array A, the last tuple for each key is exactly the "sum" of the associated key.

Multi-number. Given a set of (key, value) pairs, the goal is to for each key k, assign consecutive numbers $1, 2, \cdots, n_k$ to elements with key k, respectively, where n_k is the total number of elements with key k. The multi-number can be solved by the reduce-by-key primitive. Each tuple will be associated with value 1, and the operator $\oplus$ is simply set as $+$. After solving the all prefix-sums problem on the derived array A, $S[i]$ is exactly the number that should be attached to i-th tuple.

Multi-Search. Given N_1 distinct keys and N_2 queries, where $N = N_1 + N_2$, for each query, find its predecessor, i.e., the largest key that is no larger than the query. The multi-search can be solved by the prefix-sum primitive. We first sort all the keys and queries together. Then for each key k, define its corresponding $A[i]$ as itself; for each query, define its $A[i] = -\infty$; define $\oplus = \max$. Then it should be obvious that $S[i]$ is the predecessor of the corresponding query.

Semi-join. Given two relations $R_1(A, B)$ and $R_2(B, C)$, the target is to find tuples in R_1 that can be joined with at least one tuple in R_2. The semi-join can be solved by the multi-search primitive. We treat tuples in R_1 as queries and tuples in R_2 as keys. We sort $R_1 \cup R_2$ by attribute B. After running the multi-search algorithm, we check for each tuple $t \in R_1$ with the attached $A[i]$. We keep t if and only if $\pi_B t = A[i]$.

E PROOF OF OPTIMALITY

Our optimality is parameterized by the symmetric tree topology G, the cardinality $\phi = \{(\phi_{rv}, \phi_{sv}) : v \in V_C\}$ of initial data distribution, and the join size $0 \le \text{OUT} \le \left(\sum_{v \in V_C} \phi_{rv}\right) \cdot \left(\sum_{v \in V_C} \phi_{sv}\right)$.

Lemma E.1. *Given a symmetric tree topology $G = (V, E)$ with compute nodes V_C, for any valid cardinality distribution $\phi' = \{(\frac{\phi_{rv}}{2}, \frac{\phi_{sv}}{2}) : v \in V_C\}$ and output size parameter $\text{OUT}' = \frac{3}{4}\text{OUT} > 0$,*

[11] We make this assumption based on three reasons: (i) the scenario when this primitive is used indeed satisfies this condition; (ii) this is necessary for completing this primitive in 2 rounds; (iii) any algorithm requires $\Omega(\log |V_C|)$ rounds in the worst case when every node contains a single element, which is too costly.

there exists a data distribution $\mathcal{D}$ for $R(A,B) \bowtie S(B,C)$ asymptotically conforming to ϕ' and OUT′*, such that any algorithm computing their join results incurs a cost of $\Omega(C_{int})$.*

Proof of Lemma E.1. Consider an arbitrary edge $e \in E$. For $\phi' = \{(\frac{\phi_{rv}}{2}, \frac{\phi_{sv}}{2}) : v \in V_C\}$, let $m_1 = \frac{1}{4} \cdot \sum_{v \in G_e^-} \phi_{rv}$, $n_1 = \frac{1}{4} \cdot \sum_{v \in G_e^-} \phi_{sv}$, $m_2 = \frac{1}{4} \cdot \sum_{v \in G_e^+} \phi_{rv}$ and $n_2 = \frac{1}{4} \cdot \sum_{v \in G_e^+} \phi_{sv}$. W.l.o.g., assume $\min\{m_1, n_2\} \geq \min\{m_2, n_1\}$. We will prove a lower bound of $\Omega(\min\{m_1 + m_2, m_1 + n_1, m_2 + n_2, n_1 + n_2\} = \Omega(\min\{m_1, n_2\})$ for computing any join instance that conforms to ϕ' and OUT′.

Consider a common domain $\mathbb{D}_1$ and an instance $I = (I_1, I_2)$ by choosing m_1 distinct elements from $\mathbb{D}_1$ and n_2 distinct elements from $\mathbb{D}_2$, such that $|I_1 \cap I_2| \leq \min\{\text{OUT}', m_1, n_2\}$. Below, we will construct an instance of $R(A,B) \bowtie S(B,C)$ based on I correspondingly.

Step (1): We construct a sub-relation $R_1(A,B)$ by including a pair (b,b) for every $b \in I_1$. Similarly, we construct a sub-relation $S_1(B,C)$ by including a pair (b',b') for every $b' \in I_2$. Note that R_1 will be distributed across compute nodes in G_e^- such that each node v receives $\frac{1}{4}\phi_{rv}$ pairs, and S_1 will be distributed across compute nodes in G_e^+ such that each node v receives $\frac{1}{4}\phi_{sv}$ pairs.

Step (2): We further construct four sub-relations - $R_2(A,B)$ of size m_1, $S_2(B,C)$ of size n_2, $R_3(A,B)$ of size $2m_2$ and $S_3(B,C)$ of size $2n_1$. Moreover, R_2 will be distributed across compute nodes in G_e^- such that each node v receives $\frac{1}{4}\phi_{rv}$ pairs, S_2 will be distributed across compute nodes in G_e^+ such that each node v receives $\frac{1}{4}\phi_{sv}$ pairs, R_3 will be distributed across compute nodes in G_e^+ such that each node v receives $\frac{1}{2}\phi_{rv}$ pairs, and S_1 will be distributed across compute nodes in G_e^- such that each node v receives $\frac{1}{2}\phi_{sv}$ pairs. It can be easily checked that each compute node v receives $\frac{1}{2}\phi_{rv}$ pairs from $R_1 \cup R_2 \cup R_3$ and $\frac{1}{2}\phi_{sv}$ pairs from $S_1 \cup S_2 \cup S_3$.

We next show how to set the B-values for these four sub-relations. We first set four more disjoint domains $\mathbb{D}_2, \mathbb{D}_3, \mathbb{D}_4, \mathbb{D}_5$ such that $\mathbb{D}_i \cap \mathbb{D}_j = \emptyset$ for any $i, j \in [5]$. Moreover, we set a special value $b^* \notin \mathbb{D}_1 \cup \mathbb{D}_2 \cup \mathbb{D}_3 \cup \mathbb{D}_4 \cup \mathbb{D}_5$. We distinguish two more cases:

- If $\text{OUT}' \leq \min\{m_1, n_2\}$, we set $\pi_B R_2 \subseteq \mathbb{D}_2$, $\pi_B S_2 \subseteq \mathbb{D}_3$, $\pi_B R_3 \subseteq \mathbb{D}_4$, and $\pi_B S_3 \subseteq \mathbb{D}_5$.
- If $\text{OUT}' > \min\{m_1, n_2\}$, then we choose two integers k_1, k_2 such that $1 \leq k_1 \leq m_1 + 2m_2$, $1 \leq k_2 \leq n_2 + 2n_1$, and $k_1 \cdot k_2 \leq \text{OUT}' - \min\{m_1, n_2\} \leq 4k_1k_2$. This is always feasible since $4(m_1 + m_2) \cdot (n_1 + n_2) \geq \text{OUT}'$.
  - If $k_1 < m_1$, we set k_1 tuples from R_2 with the same value b^*, and the remaining tuples with B-value chosen from $\mathbb{D}_2$. If $k_1 > m_1$, we set all tuples from R_2 with the same value b^*, $k_1 - m_1$ tuples from R_3 with the same value b^*, and remaining $2m_2 + m_1 - k_1$ tuples from R_3 with B-value chosen from $\mathbb{D}_3$.
  - If $k_2 < n_2$, we set k_2 tuples from S_2 with the same value b^*, and the remaining $n_2 - k_2$ tuples from S_2 with B-value chosen from $\mathbb{D}_2$. If $k_2 > n_2$, we set all tuples from S_2 with the same value b^*, $k_2 - n_2$ tuples from R_3 with the same value b^*, and remaining $2n_1 + n_2 - k_2$ tuples from S_3 with B-value chosen from $\mathbb{D}_5$.

The join instance will be $R = R_1 \cup R_2 \cup R_3$ and $S = S_1 \cup S_2 \cup S_3$. It can be easily checked that $|R \bowtie S| = \text{OUT}'$. Moreover, any algorithm for computing the join instance $R \bowtie S$ must also solve the set intersection problem I. Thus, the lower bound $\Omega(\min\{m_1, n_2\})$ of solving I will be applied to computing any instance $R \bowtie S$ that conforms to ϕ' and OUT′. □

Let the data distribution constructed in Lemma E.1 be $\mathcal{D}_1^*$. Let the data distribution in $\mathbb{D}(\phi, \text{OUT})$ that achieves the largest C_{pack} to be $\mathcal{D}_2^*$. Let K be the set of boundary keys in $\mathcal{D}_2^*$. We construct another data distribution $\mathcal{D}_3^*$ with initial cardinality distribution ϕ'. More specifically, we simply halve the number of tuples for each key distributed at every compute node. It can be easily checked that $\phi'_{rv} = \frac{1}{2}\phi_{rv}$ and $\phi'_{sv} = \frac{1}{2}\phi_{sv}$ for every $v \in V_C$. Moreover, the output size of this new data distribution is $\frac{1}{4}$OUT. For simplicity, we assume that $\mathcal{D}_1^*$ and $\mathcal{D}_2^*$ have disjoint domain of join keys.

Let $\mathcal{D}^*$ be the data distribution if putting $\mathcal{D}_1^*$ and $\mathcal{D}_3^*$ together. It can be easily checked that $\mathcal{D}^*$ conforms to ϕ and OUT.

Lemma E.2. *Any algorithm needs to incur a cost of* $\Omega\left(C_{int} + \max_{\mathcal{D}\in\mathbb{D}(\phi,\text{OUT})} C_{pack}\right)$ *for computing the join results of* $\mathcal{D}^*$.

Proof. First, we point out that the computation for $\mathcal{D}_1^*$ and $\mathcal{D}_3^*$ are independent, since they have disjoint domain of join keys. Then, any algorithm for computing $\mathcal{D}^*$ needs to incur a cost of $\Omega(C_{\text{int}})$, implied by the fact that any algorithm for computing $\mathcal{D}_1^*$ needs to incur such a cost. In the remaining, we only consider the task of computing the Cartesian products for boundary keys $\{R_b \times S_b\}_{b\in K}$, where K is defined by $\mathcal{D}_2^*$. It suffices to show that any algorithm for this task needs to incur a cost of $\Omega\left(C^*_{\text{pack}}\right)$, where $C^*_{\text{pack}} = \max_{\mathcal{D}\in\mathbb{D}(\phi,\text{OUT})} C_{\text{pack}}$.

Consider an optimal algorithm $\mathbb{A}$ for this task, and fix some $b \in K$. Since this algorithm correctly computes $R_b \times S_b$, each compute node in V_C must be responsible to produce some pair from the Cartesian product. W.l.o.g., we can assume that these are disjoint (otherwise we can tweak the algorithm to produce disjoint results without increasing the cost). Now, for a G_α^j tree, let $g_{b,j}(\mathbb{A})$ be the set of tuple pairs assigned to the compute nodes in G_α^j. Note that in this case, the tree is responsible to compute a fraction $\hat{f}_{b,j}(\mathbb{A}) = \dfrac{|g_{b,j}(\mathbb{A})|}{|R_b|\cdot|S_b|}$ of the Cartesian product. Since our protocol iterates over all possible fractional allocations, it will also consider the solution for which $f_{b,j} = \hat{f}_{b,j}(\mathbb{A})$.

Conditioning on the lower bound from set intersection, we first argue one important property:

- The input data already sitting at the compute nodes does not help the computation.
- For $\vec{G}_\alpha^j$, it is always more load-efficient for computing $g_{b,j}(\mathbb{A})$ as a contiguous rectangle such that the dimensions are as close as possible (without exceeding the smaller $|R_b|$).

Consider an arbitrary directed α-tree $\vec{G}_\alpha^j$ rooted at r. Let $g_{b,r} = g_{b,j}$, and $g_{b,v}$ be the number of pairs emitted by v for key b. Let $x_{b,v}, y_{b,v}$ be the number of R-tuples, S-tuples received by node v under $\mathbb{A}$. Let $R_{b,v}, S_{b,v}$ be the number of R-tuples, S-tuples with key b, distributed over the subtree rooted at v. We consider the following non-linear integer program:

$$
\begin{aligned}
\min \quad & L \\
\text{s.t.} \quad & |R_b|\cdot|S_b| = g_{b,r}, && b\in K, \\
& \sum_{u\in\mathcal{L}_v} g_{b,u} = g_{b,v}, && b\in K, v\in V, \\
& \sum_{b\in K}(x_{b,v}+y_{b,v}) \le L\cdot w_v, && v\in V\setminus\{r\}, \\
& \left(|R_{b,v}|+x_{b,v}\right)\left(|S_{b,v}|+y_{b,v}\right) \ge g_{b,v}, && b\in K, v\in V\setminus\{r\}, \\
& |R_{b,v}|+x_{b,v} \le |R_b|, && b\in K, v\in V\setminus\{r\}, \\
& |S_{b,v}|+y_{b,v} \le |S_b|, && b\in K, v\in V\setminus\{r\}, \\
& \sum_{b\in B}\left(|R_{b,v}|+|S_{b,v}|\right) \le L\cdot w_v, && v\in V\setminus\{r\}, \\
& g_{b,v}, x_{b,v}, y_{b,v} \in \mathbb{Z}^*, && v\in V\setminus\{r\}
\end{aligned}
\tag{7}
$$

The optimal solution of the above program provides a lower bound for the cost of computing the Cartesian pairs inside the α-tree. We relate it to (1) as follows: every solution of (1) with cost L

can be transformed into a solution of (7) with cost at most $2L$. From $x_{b,v}, y_{b,v}$, we can simply set $x'_{b,v} = |R_{b,v}| + x_{b,v}$ and $y'_{b,v} = |S_{b,v}| + y_{b,v}$ satisfying all constraints. Hence, it suffices to find the optimal solution of (1), which is a constant-approximation of the optimal solution of (7). This means that we can w.l.o.g. consider protocols that ignore the local input data, as our protocol class does.

Next, we will reason about the optimal shape of the $g_{b,j}$ area. It is easy to see that an optimal algorithm will always organize $g_{b,j}$ such that it forms a contiguous rectangle. Hence, we need to show that the optimal strategy will attempt to make the rectangle as close to a square as possible (will keeping the same area). Indeed, consider a slightly different non-linear integer program of (1), such that $|R_b| < |R'_b| \le |S'_b| < |S_b|$ with $|R'_b| \cdot |S'_b| = |R_b| \cdot |S_b|$.

$$
\begin{aligned}
\min \quad & L \\
\text{s.t.} \quad & |R'_b| \cdot |S'_b| = g_{b,r}, && b \in K, \\
& \sum_{u \in \mathcal{L}_v} g_{b,u} = g_{b,v}, && b \in K, v \in V, \\
& \sum_{b \in K} (x_{b,v} + y_{b,v}) \le L \cdot w_v, && v \in V \setminus \{r\}, \\
& x_{b,v} \cdot y_{b,v} \ge g_{b,v}, && b \in K, v \in V \setminus \{r\}, \\
& x_{b,v} \le |R'_b|, y_{b,v} \le |S'_b|, && b \in K, v \in V \setminus \{r\}, \\
& g_{b,v}, x_{b,v}, y_{b,v} \in \mathbb{Z}^*, && b \in K, v \in V \setminus \{r\}
\end{aligned}
\tag{8}
$$

Any solution of (1) with cost L can be transformed into another solution of (8) with cost at most L. Consider an arbitrary pair of (b, v). First, $x_{b,v} \le |R_b| \le |R'_b|$ always holds. If $y_{b,v} \le |S'_b|$, we just set $x'_{b,v} = x_{b,v}$ and $y'_{b,v} = y_{b,v}$. Otherwise, we set $x'_{b,v} = \frac{x_{b,v} \cdot y_{b,v}}{|S'_b|}$ and $y'_{b,v} = |S'_b|$. It is obvious that $x'_{b,v} < |R'_b|$ since $x_{b,v} \cdot y_{b,v} \le |R_b| \cdot |S_b| = |R'_b| \cdot |S'_b|$. Moreover, $x_{b,v} + y_{b,v} \ge x'_{b,v} + y'_{b,v}$, since $x_{b,v} \cdot y_{b,v} = x'_{b,v} \cdot y'_{b,v}$ and $x_{b,v} \le |R_b| < |R'_b| \le |S'_b| < y_{b,v} \le |S_b|$. Thus, the optimal solution of (8) always has a smaller cost than that of (1).

Conditioning on the lower bound from set intersection, it is safe to assume that for each directed α-tree $\vec{G}^j_\alpha$, the whole set of R set as well as the S-tuples initially distributed at the compute nodes residing at $\vec{G}^j_\alpha$. Now, we need to argue the last important property:

- it is also efficient for redistributing the S-tuples across the leaf nodes of β-tree, i.e., the root nodes of directed α-trees, when $g_{b,j}(\mathbb{A})$ is a contiguous rectangles with the two dimensions as close as possible.

Consider any edge $e \in G_\beta$. Recall that $\mathcal{L}_\beta$ is the set of leaf nodes of G_β. Assume some S-tuples need to be transferred from $G^+_e \cap \mathcal{L}_\beta$ to $G^-_e \cap \mathcal{L}_\beta$. It can be easily checked that the total number of S-tuples redistributed over $G^-_e \cap \mathcal{L}_\beta$ is minimized when $g_{b,j}$ is a contiguous rectangles with the two dimensions as close as possible, thus the tuple differential Δ_e is also minimized correspondingly. □

Received December 2023; revised February 2024; accepted March 2024