Communication-Avoiding Recursive Aggregation

***Abstract*—Recursive aggregation has been of considerable interest due to its unifying a wide range of deductive-analytic workloads, including social-media mining and graph analytics. For example, Single-Source Shortest Paths (SSSP), Connected Components (CC), and PageRank may all be expressed via recursive aggregates. Implementing recursive aggregation has posed a serious algorithmic challenge, with state-of-the-art work identifying sufficient conditions (*e.g.*, pre-mappability) under which implementations may push aggregation within recursion, avoiding the serious materialization overhead inherent to traditional reachability-based methods (*e.g.*, Datalog).**

State-of-the-art implementations of engines supporting recursive aggregates focus on large unified machines, due to the challenges posed by mixing semi-naïve evaluation with distribution. In this work, we present an approach to implementing recursive aggregates on high-performance clusters which avoids the communication overhead inhibiting current-generation distributed systems to scale recursive aggregates to extremely high process counts. Our approach leverages the observation that aggregators form functional dependencies, allowing us to implement recursive aggregates via a high-parallel local aggregation to ensure maximal throughput. Additionally, we present a dynamic join planning mechanism, which customizes join order per-iteration based on dynamic relation sizes. We implemented our approach in PARALAGG, a library which allows the declarative implementation of queries which utilize recursive aggregates and executes them using our MPI-based runtime. We evaluate PARALAGG on a large unified nodes and leadership-class supercomputers, demonstrating scalability up to 16,384 processes.

***Index Terms*—relational algebra, aggregation, communication-avoiding algorithms**

I. INTRODUCTION

Hybridizing deductive reasoning with monotonic aggregation is a key implementation strategy used in graph mining, social-media analytics, and program analysis. These applications involve extreme deductive throughput, computing results over graphs with billions of edges. Considerable attention has been given to the high-performance and scalable implementation of recursive aggregation, targeting both unified nodes [1] and clusters [2]–[4]. Unfortunately, it is not obvious how to scale current implementation strategies for recursive aggregates to high-performance clusters and supercomputers due to inherent communication overhead. State-of-the-art work in recursive aggregation is built upon unified multi-core machines, in part due to the inherent communication complexity in a distributed setting [1].

In this paper, we tackle this challenge of scaling recursive aggregation to high-performance clusters using the Message-Passing Interface (MPI), designing a communication-avoiding algorithm to ensure sufficient work is available at very high process counts. We leverage recent developments in parallel relational algebra to design PARALAGG, a C++ library which enables the declarative implementation of relational algebra kernels extended with recursive aggregates. Our experiments show that PARALAGG scales favorably, even at high process counts (up to 16,384 processes).

Key to our approach is to recognize that current approaches to distributed recursive aggregates force the communication of intermediate results–even when those intermediate results may not be usefully observed until the end of the fixed-point. This observation leads us to a restricted form of recursive aggregates based on functional dependencies, following semantic inspiration from systems such as *Datalog*^{FS} and DeALS [4]. Operationally, this restriction enables us to use the independent columns as keys to colocate tuples and perform a highly-parallel local aggregation. Additionally, our scheme allows local aggregation to be fused with tuple deduplication, enabling a highly-expressive class of operations with very low added communication overhead compared to state-of-the-art iterated relational algebra systems.

We make these contributions to the literature:

- An extension of double-hashed parallel relational algebra to include monotonic aggregation, and subsequently enable substantial algorithmic improvements that hinder the application of parallel RA to recursive aggregate queries.
- A communication-optimized join-layout algorithm which intelligently selects relation order for joins on-the-fly.
- An implementation of our algorithm as a C++ library, PARALAGG, and its application to declaratively implement recursive queries including SSSP, CC, and PageRank.
- An evaluation showing that PARALAGG outperforms known state-of-art tools for recursive aggregation on large unified nodes and that PARALAGG's results are robust across a variety of graphs (from SuiteSparse [5]) at medium scale (256 and 512 processes) and and that PARALAGG achieves healthy strong scaling (up to 16,384 processes) on the Theta supercomputer.

II. BACKGROUND AND OVERVIEW

We will introduce our setting by way of a set of narrative examples, which highlight the key concerns apropos the implementation of recursive aggregates on high-performance clusters. To start, we consider single-source shortest paths (SSSP), which calculates the length of the shortest path from a start node to all subsequently reachable nodes. In Figure 1, we see a weighted graph of four nodes forming a diamond; formally the input graph is a relation $\mathit{Graph} = \mathit{Node} \times \mathit{Node} \times \mathbb{N}$. We will *materialize* the relation $\mathit{Spath} = \mathit{Node} \times \mathit{Node} \times \mathbb{N}$, which accumulates—at every iteration—a measure of the shortest paths (thusfar) from nodes *from* to *to*, both reachable from the start node by construction. As the computation evolves, a given

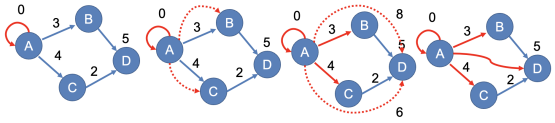

Fig. 1. Shortest path computed in 4 consecutive Iterations of SSSP

$$
\begin{array}{lcl}
\mathit{Spath}(n, n, 0) & \leftarrow & \mathit{Start}(n). \\
\mathit{Spath}(\mathit{from}, \mathit{to}, \$\text{MIN}(l + n)) & \leftarrow & \mathit{Spath}(\mathit{from}, \mathit{mid}, l), \\
 & & \mathit{Graph}(\mathit{mid}, \mathit{to}, n).
\end{array}
$$

Fig. 2. SSSP written using Horn clauses and recursive aggregates

pair of *from* and *to* will discover monotonically-decreasing shortest paths, eventually reaching a fixed point.

We begin by populating *Spath* with the tuple $(A, A, 0)$, indicating that (a) the node A has been reached and (b) the length of the shortest path from A to itself is 0; this can be seen as demanding (querying) the shortest paths reachable from A. The top rule in Figure 2 formalizes this notion as a Horn clause, and says that whenever a fact $\mathit{Start}(n)$ exists, $\mathit{Spath}(n, n, 0)$ is to be generated. Horn clauses generalize SQL to recursive queries (necessary in nontrivial uses of recursive aggregation) and anticipate compilation to high-performance relational algebra kernels [2], [6]. For example, our first rule is roughly equivalent to SELECT INTO in SQL. Because A has undiscovered neighbors we continue to iterate, discovering all neighbors (in this case B and C); we include these nodes in *Spath* (with their commensurate weights). The second rule makes this precise, when *Spath* contains a tuple $(\mathit{from}, \mathit{mid}, l)$, and Graph contains an edge $(\mathit{mid}, \mathit{to}, n)$, we may construct an *extended* path $(\mathit{from}, \mathit{mid}, l + n)$ connecting nodes *from* and *to* with weight $l + n$. This (second) rule illustrates the use of an aggregate: in general, there will be *multiple* (say i) paths between *from* and *to*, each through distinct nodes mid_i and with the ultimate weight $l_i + n$. The head clause here uses the $MIN aggregate, which forms a functional dependence and allows selecting for the minimum value of $l_i + n$ across all i. This functional dependence also ensures that for all tuples sharing the same *from* and *to* columns, only the smallest $l_i + n$ is materialized in the output database.

The second rule is inductive, and forces execution to a fixed-point. Efficient implementations use *semi-naïve* evaluation, wherein new paths are discovered incrementally. Intuitively, we track a frontier; operationally, we split relations into multiple versions: *delta* and *full*. New tuples are added to *delta* only when they are not present in *full*, and *delta* is merged into *full* at the end of each iteration. For example, during the third iteration in Figure 1, only new paths extensions (to D) need to be considered, ignoring previously-discovered paths; the second rule is implemented by joining on Spath_{Δ}.

There are several distinct sources of complexity, both semantic and practical, which complicate the high-performance implementation of recursive aggregates on clusters. Developing a well-founded semantics for recursive aggregates proved a serious challenge due to the complications of harmoniously integrating recursive aggregation with semi-naïve evaluation [4], [7]. In our setting, however, we face another distinct challenge: how to distribute relations among a set of nodes in a cluster.

Recent work in iterated parallel relational algebra (without aggregation) distributes relations via *double hashing*: tuples are distributed across a cluster via a bucket/sub-bucket decomposition and balanced dynamically [8], [9]. The semantic tension between semi-naïve evaluation and aggregation extends to practical challenges when distributing relations, and inhibits the direct application of these methods. To understand why, consider what happens when *Spath* is distributed across a cluster, with each tuple being assigned a rank via a hash of its (join) columns (following the common double-hashing technique). Now consider the third iteration in Figure **??**: two new paths from A to D are discovered, one through B and one through C. In transitive closure, these new paths would both (in general) be materialized on distinct nodes independently, with no need for the longer path to be aware of the shorter path. By contrast, this solution does *not* work for SSSP: distributing the relation would require additional communication to aggregate a single shortest path, essentially "deleting" the longer path in favor of the shorter path.

Our work targets this intersection of recursive aggregates, semi-naïve evaluation, and distribution. A number of scalable implementations (*e.g.*, RaSQL, BigDatalog) are based on a common restriction, pre-mappability (**PreM**), which characterizes when aggregation may be pushed down into joins, rather than generating products and aggregating at the end. **PreM**, obeyed by all aggregate operators provided by these systems, ensures that semi-naïve evaluation is monotonic even in the presence of partial aggregates.

Our observation is that these common recursive aggregates satisfying **PreM** also obey a restriction which enables us to implement them in a highly-parallel manner with minimal communication overhead: aggregated columns are never joined upon within a fixed point. This observation allows us to implement recursive aggregates as a combination of gather and local aggregation. Unfortunately however, no state-of-the-art distributed tools take advantage of this fact, instead treating aggregated columns in the same manner as normal relations with respect to indexing and query optimization. This presents communication overhead due to the "leaky" nature of partial results produced by recursive aggregates. To see why, consider an extension of SSSP to compute the *longest* shortest path (*Lsp*).

$$
\begin{array}{lcl}
\mathit{SpNorm}(f, t, v) & \leftarrow & \mathit{Spath}(f, t, v). \\
\mathit{Lsp}(l) & \leftarrow & \mathit{SpNorm}(\_, \_, v), l = \text{MAX}(v).
\end{array}
$$

Here, *SpNorm* is a copy of the *Spath* relation computed earlier. The second rule aggregates the globally-longest path. In the semi-naïve evaluation of *Lsp*, all tuples which appear in the *delta* version of *Spath* are copied to *SpNorm*. However, during the computation of *Spath*, many path lengths are transient

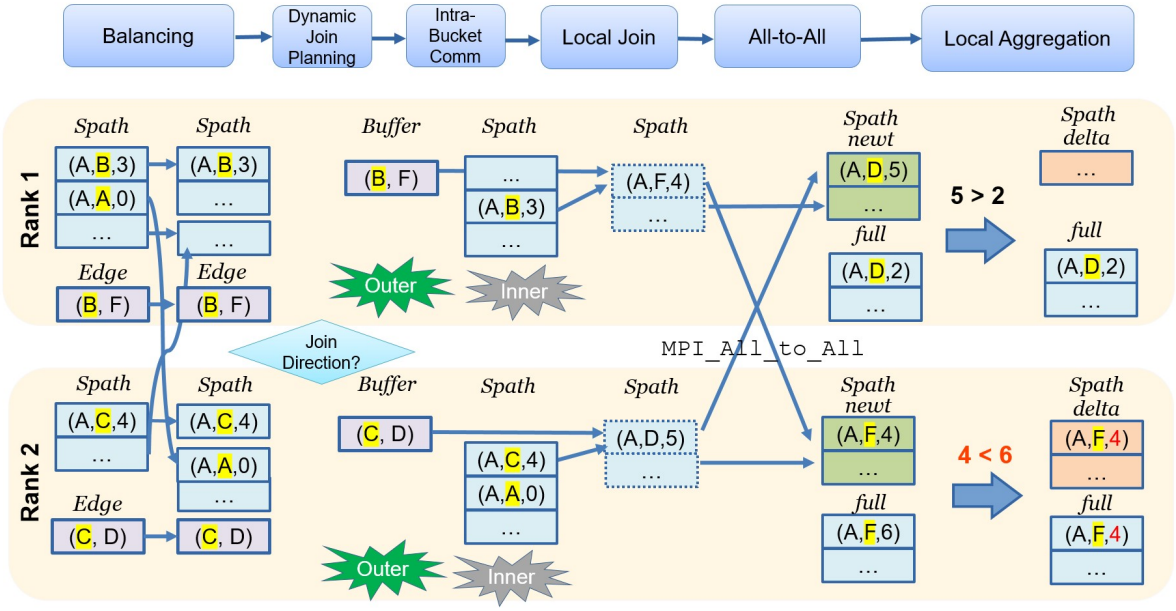

Fig. 3. Visualization of an example tuple flow during one iteration of the Single Source Shortest Path (SSSP) query in PARALAGG (letter "A" to "F" in Spath and Edge tuples are node identifier)

and will be purged from the database once a shorter path is found—thus, tracking their intermediate results imposes unnecessary communication in the setting where these partial results must be communicated. If the copy into *SpNorm* is computed within the same fixpoint as *Spath*, intermediate results will "leak" and cause *SpNorm* to contain all possible paths in the graph. However, only the shortest paths will actually be used in the computation of *Lsp*; this intermediate tuple leakage represents algorithmically-imposed overhead (see Definition 2 in [10]).

We now sketch an end-to-end example of how PARALAGG distributes SSSP. Figure 3 sketches the evolution of the relation *Spath* as computation evolves; we detail only two ranks for space. Reading left to right, we observe the changes that occur and illustrate how data propagates through the system. As in BPRA, iterations begin with balancing, which shuffles work to aid overall throughput–in this case the tuple $(A, A, 0)$ is moved to rank 2. Next, our system employs dynamic join planning, which uses relation sizes to switch relation direction on a per-iteration basis—avoiding communication overhead involved in serializing the larger relation. We further detail dynamic join planning in Section IV-D, but found it to be highly effective in practice, showing performance gains of $2\times$ (see Figure 4). Dynamic join planning decides an "outer" and "inner" relation (see Definition IV.1). Next, we perform intra-bucket communication; in our distribution scheme, tuples are distributed by a bucket/sub-bucket distribution, whose buckets are determined by hashes, and sub-buckets are dynamically tuned to grow with relation size. Thus, all tuples sharing the same join column are assigned the same bucket number, and the join result can be computed inside each bucket. However, after data balancing, two tuples in the same bucket may have different sub-bucket numbers and may reside on different physical ranks. In order to join tuples in different sub-buckets, intra-bucket communication serializes the smaller relation, replicating it to enable a local join. After intra-bucket communication, all joinable tuples are gathered on their correct rank, and local join happens in a highly-parallel manner. Once local join finishes, new tuples may generated and will be distributed via their bucket/sub-bucket decomposition.

The last step of the iteration is deduplication, which is efficiently fused with local aggregation in our framework. Deduplication distinguishes new tuples from elements of *full* (and subsequently drives semi-naïve evaluation); in our setting, independent columns are grouped and used to implement local aggregation alongside the typical deduplication phase used by non-aggregated relations. In our case, $(A, F, 4)$ is inserted into both *full* and *delta*, and used as the accumulator for the next iteration. In our system, transient results are discarded in a communication-avoiding way, effectively deleting $(A, F, 6)$ from the database.

III. RECURSIVE AGGREGATION AS MONOTONICITY

We formalize our approach as an extension of Datalog. As is standard, we present a model-theoretic and fixpoint semantics of rules extended to include recursive aggregation via functional dependencies. We begin by generalizing Datalog's relations to include lattice-value columns, and continue by

showing how a Datalog with these non-power-set-lattice relations can be used to solve the recursive aggregation problem.

Functional Dependency and Recursive Aggregation: Recursive aggregators are designed to solve the functional-dependence problem. Some relations may be functions in that a column's value is determined by the values of other columns. Such columns may be referred to as "dependent columns". For instance, in SSSP, the value of the third column in $Spath(from, to, \$\text{MIN}(l+n))$ is uniquely determined once the values of *from* and *to* are fixed. During each iteration of the computation, only the smallest l will be kept. Maintaining this unique mapping of dependent columns is difficult in traditional Datalog semantics ~~as normally Datalog evaluation ascends a strict product of power-set lattices, formed point-wise over the relations in the Database. However, in a Datalog engine that supports functional dependency, when a new value for recursive aggregate column is generated, simply adding the new tuple into database is not enough, the uniqueness property of a functional dependency will require evaluation to effectively delete all unqualified tuples from Database when a dependent column is updated.~~ **However, Datalog engines that support functional dependency can maintain the monotonicity of the database by effectively dropping unqualified tuples when a new dependent column value is generated.** For instance in our introduction example Figure 1, only the shortest edge length from node A to D in the database, which is 6, will finally be stored in the database, while the path length of 8, which is larger than 6, will be deleted from the database. To achieve this, we need to group all existing tuples in database that share the same non-aggregated columns as the newly generated tuple, and then use a function to aggregate them into one expected dependent value at each update.

~~The state-of-the-art Datalog engine Soufflé supports functional dependency through the "choice-domain" keyword . When a relation is annotated with "choice-domain", it only accepts the first-evaluated dependent-column value for each tuple of independent-column values, discarding any values discovered after the first. While this restriction of only retaining the first value deduced can be useful in practice, it will not support the recursive updating required in the Single-Source Shortest Path (SSSP) algorithm.~~

Before presenting the formal semantics of our recursive aggregator, we formally define functional dependency first. The functional dependency of a relation $R(r_1, ..., r_n, d_1, ..., d_n)$, where $r_1, ..., r_n$ are independent columns and $d_1, ..., d_n$ are dependent columns, can be represented as a pair $\langle F_R, g\rangle$. Here, F_R is a project function takes any tuple $(x_1, ..., x_n)$ value in $\Pi_{r_1,...,r_n}$ as input, producing a corresponding dependant-column value set. To compute F_R, we first perform a relation-algebra selection σ by grounding all independent columns with $(x_1, ..., x_n)$ and then project the dependent columns $\Pi_{d_1,...,d_n}$ on the selected results. Formally, for any tuple $(x_1, ..., x_n) \in \Pi_{r_1,...,r_n}(R)$, we have:

$$F_R(x_1, ..., x_n) = \Pi_{d_1,...,d_n}(\sigma_{r_1,...,r_n=x_1,...,x_n} R)$$

The function g is used to restrict the result of the function F_R. For example, in the shortest path example, g is min, which restricts the result of F_R to the minimum value. ~~Another example of g to emulate the Soufflé's "choice-domain" behavior ; in this case, g restricts the result to the first evaluated tuple.~~

Relations and Lattices: Some datalog engines have a limited number of internal recursive aggregators to solve various kinds of functional dependencies, such as DeALS, BigDatalog, and RaSQL [2]–[4]. In these systems, users cannot define recursive aggregators according to their needs, limiting the expressiveness of the system. To improve expressiveness, a more generalized semantic is needed to support arbitrary monotonic dependencies. Inspired by SociaLite [11] and some pioneering semantic research on recursive aggregators [7], we use semilattice-based relations to solve functional dependencies in PARALAGG.

A lattice is a partially-ordered set with two operators, join (least-upper bound, $x \sqcup y$) and meet (greatest-lower bound, $x \sqcap y$), defined on every pair of elements [12]. For example, for any set S, the power set $\mathcal{P}(S)$ is a lattice with $\sqcup = \cup$ and $\sqcap = \cap$. If only the join operator ($\sqcup$) is defined, the partially-ordered set is then called a join semilattice. In PARALAGG, users define a partial order $\preceq$ and meet operator $\sqcup$ over dependent columns to convert a normal relation R into a join semilattice L, which only allows comparing tuples sharing the same set of independent columns.

Following the definition of functional dependency, its not hard to find that grouping all comparable independent column values correspond to F_R, while restriction function g can be witnessed by continuously applying join operator on all comparable tuples until a least fixpoint. Based on this correspondence between lattice and functional dependency, recursive aggregators to solve functional dependency can implemented by extending datalog's powerset-based fixpoint computation to support semilattice-based relations.

In a normal set-based relation R, computing the least-fixpoint is done via semi-naïve evaluation, which can formalized as:

$$\begin{aligned} R_{i+1} &= R_i \cup R_{\Delta i} \\ R_{\Delta i+1} &= (R_i \cup R_{\Delta i}) - R_i \end{aligned}$$

After lifting a set-based relation into a semilattice, we can define following lifted fixpoint evaluation for our aggregated relations:

$$\begin{aligned} L_{i+1} &= \{ (r_1, ..., r_n, \bigsqcup_{dep}) \mid (r_1, ..., r_n) \in L_i \cup L_{\Delta i} \\ &\quad \wedge\ dep = F_L(r_1, ..., r_n)\} \\ L_{\Delta i+1} &= \{ (r_1, ..., r_n, d_1, ..., d_n) \\ &\quad \mid (r_1, ..., r_n, d_1, ..., d_n) \in L_{i+1} \\ &\quad \wedge\ (d_1, ..., d_n) \notin \Pi_{(d_1,...,d_n)}(L_i)\} \end{aligned}$$

The join operator in computation of L_i guarantees the ascending chain condition, and assures that (assuming the lattice is of finite height) the program eventually terminuates.

IV. IMPLEMENTATION

PARALAGG is implemented as a C++ library which provides a set of declarative relational algebra and recursive aggregator primitives to implement the relational algebra kernels detailed

in Section III. We now describe how our API and implementation of PARALAGG leverage parallelism and avoid unnecessary communication.

A. Data Distribution

Distributed engines like RaSQL, BigDatalog, BPRA, and SociaLite distribute relations based on indexing columns, necessitating the breaking down of k-arity joins into a series of binary joins. This approach ensures that all join candidates sharing the same indexed columns can be gathered on the same computation node. Similarly, in line with these engines, the focus of PARALAGG is specifically on enhancing the performance of distributed binary joins and not yet supports k-arity joins.

Existing Distributed engines ~~Distributed engines (including RaSQL, BigDatalog, and SociaLite)~~ do not differentiate their tuple distribution schemes based on whether a relation is a recursive aggregate, using hash-based partitioning to distribute both recursive aggregates and non-aggregated relations. We argue this strategy incurs communication overhead which we want to avoid (similar issues are seen in the engineering of lock-free datastructures on large unified machines [1]). These current-generation engines treat aggregated columns in an identical manner to non-aggregated relations. This forces aggregated columns, the results of functional dependencies to participate in indexing and query optimization, rather than ignoring these dependent columns.

Our implementation optimizes the distribution of recursive aggregates by leveraging two key insights. First, to minimize communication, it is essential to use independent columns for indexing while excluding aggregated columns from the indexing process. By doing so, all tuples sharing the same independent column will be automatically gathered together when generated, eliminating the need for additional communication overhead during recursive aggregation. This is not a straightforward task since pre-existing parallel relational algebra systems require indexing and partitioning to accommodate normal join operations first, and all join column values may affect data distribution. For example, our investigation into the implementations of *both* BigDatalog and RaSQL use a global hashmap with a special partition key to store intermediate results during recursive computations. This inter-node recursive aggregation operation and global auxiliary structure greatly increases the communication overhead and, we believe, represents a significant impediment to achieving the highest-possible scale.

Our restriction forbidding dependent columns to be joined upon within a fixpoint eliminates the need for global tables or extra communication by ensuring that all tuples sharing the same independent columns will be assigned the same bucket. Following BPRA, all tuples sharing the same join columns will be assigned to the same bucket, with their sub-bucket determined by their non-join columns. For example, in Figure 3, the second column of the *Spath* relation is joined with the first column of the *Edge* relation, so both columns form an index, and BPRA's distribution ensures that the tuples $(A, B, 3)$ and (B, F) are always located on the same node.

Notice that while our restriction prohibits a user from using dependent columns as index columns, it does not fully forbid us from using a recursive aggregate in a stratified manner. The longest shortest paths query *Lsp* is an example of this. Such stratified uses of recursive aggregates may be operationalized in PARALAGG via stratification.

The second key insight in distributing recursive aggregation is that, based on the above restriction, we may implement aggregation for no additional communication overhead by fusing local aggregation with deduplication. In normal relations, deduplication performs elementwise comparison, and (in our setting) occurs in parallel based on the tuple's bucket. In the setting of recursive aggregates, deduplication must also be extended to "collapse" information: instead of merely checking for the existence of a tuple, deduplication for recursively aggregated relations applies a reducer function across all gathered results to produce the result of the local aggregation. Like normal relations, this fused deduplication/aggregation pass is performed in a highly-parallel manner; no extra communication overhead is required.

```
class RecursiveAggregator {
    using dep_val_t = set<vector<column_t>>;
    // get the dependent column value from a tuple
    virtual vector<column_t>
    dependent_column(tuple_t t);
    // comparator to form a Partial order set
    virtual partail_order_t
    partial_cmp(dep_val_t t1, dep_val_t t2);
    // parital aggregate 2 values
    virtual dep_val_t
    partial_agg(dep_val_t t1, dep_val_t t2);
}
```

Listing 1: Recursive aggregate API in PARALAGG

```
class min_dep : RecursiveAggregator {
    // dependent columns has size 1
    vector<column_t>
    dependent_column(tuple_t t){
        return {t.back()};
    // ...
    // min_val return smallest value in set
    dep_val_t
    partial_agg(dep_val_t t1, dep_val_t t2) {
        return {min(t1, t2)};
}
```

Listing 2: Implementing the $MIN functional dependence using PARALAGG.

B. API

~~PARALAGG is implemented as a C++ API, shown in listing 1. The `dependent_column` function is used by PARALAGG to compute the dependent column of a stored tuple, while the `partial_cmp` function is overloaded to define the partial order for the independent columns. `partial_agg` will be applied on newly-generated tuples and the currently-accumulated result to generate the updated~~

accumulator. We used this API to implement a plethora of recursive aggregates from the literature including $MIN, $MAX, $MCOUNT, and several others. For example, to implement $MIN we implement `partial_agg` as a function which returns the smaller of $t1$ and $t2$, as shown in Listing 2.

C. Spatial Load Balancing

Balancing tuple load on each rank is important for the scalability of a distributed relational algebra engine. It has been shown that sub-bucketing based on non-indexed columns is a useful way to balance key-skewed data [8]. Sub-bucketing means that after normal data partitioning, if the data size on each process is still imbalanced, the imbalanced relation will be logically divided into some sub-buckets and then transmit imbalance part of data to other processes that have less data. **Such imbalance arises when numerous tuples stored in the engine share the same indexed columns. For example the SSSP query mentioned in Section II, where the $Graph$ relation is indexed based on the first column, a data imbalance may occur if certain vertices in the input graph have a large number of outgoing edges. The processes storing these vertices will have a larger amount of data compared to the processes that do not. By applying sub-bucketing, the data imbalance in the input graph can be effectively resolved, leading to a significant improvement in the query's scalability.** However, sub-bucketing will also causes a situation where even if two tuples are partitioned into the same computation node by their indexing column, they may still be computed on two different processes physically. This clearly violates the requirement for communication-avoiding distribution of monotonic aggregation discussed in the previous paragraph, which states that all tuples sharing the same independent column must reside in the same process, even if the indexing is based only on non-aggregated columns. Although we have an intra-bucket communication phase to keep the result correct, where all data on each sub-bucket is gathered by an MPI_ALL2ALLV-based intra-bucket communication operation, and this operation won't be as expensive as inter-bucket communication. To enable more fine-grain tuning of recursive aggregation computation communication overhead, we have added an option to allow the user to declare aggregated columns to be excluded during sub-bucket index computation. Note that at least one column should be kept in sub-bucket index computation to avoid spatial balancing being bypassed, unless the data is very balanced under normal join column-based partitioning.

D. Communication-Avoiding Join Layout

PARALAGG has several communication epochs interspersed between computation phases, in particular, there is an intra-bucket communication phase right before the local join (see Figure 3). This communication phase is instrumental in facilitating the local joins, as it performs the key task of co-locating all matching tuples to their appropriate processes. As PARALAGG is built atop MPI, we rely on point-to-point communication using `MPI_Isend` and `MPI_Ireceive` to perform the intra-bucket communication. MPI only works with serialized 1D data buffers, and therefore a relation that is internally stored using a nested BTree data structure must be serialized before it can be transmitted over the network.

To facilitate binary joins in parallel, one of the two relations must be transmitted over the network via the intra-bucket communication phase. In this section, we present our heuristic to decide which of the two relations is chosen to be transmitted over the network.

Definition IV.1 (Outer/Inner Relation)**.** In our parallel binary joins, we'll call the relation being serialized and transmitted over the network the *outer* relation and the one that does not move the *inner* relation. The *inner* relation continues to be stored in a nested Btree data structure.

A join between two relations entails iterating over the outer relation and performing a lookup for a matching entry in the inner relation. As 1D serialized buffers can be easily iterated over in linear time complexity, the outer relation is already suited to be iterated over. Similarly, the inner relation is represented using a Btree-based data structure and is better suited to support lookup of key values, with a logarithmic running-time complexity, for purposes of materializing relevant join output.

For efficiency, we want to select a smaller relation as the outer relation. This heuristic leads to less load on both the computation phase of the join and the pre-join communication phase (intra-bucket data exchange). The outer relation is scanned over in its entirety, and is transmitted among subbuckets within each bucket to prepare for a parallel join. The inner relation stays put and benefits from indexing and $O(\log n)$ access during the join.

At the start of iteration shown in Figure 3, the tuple size of relation *Edge* is much smaller than that of the relation *Spath* on both rank 1 and rank 2. Therefore, both rank vote for *Edge* as outer relation, and broadcasting their choice to all ranks. All MPI rank agreed to put *Edge* on outer position, so tuples *Edge* in edge are serialized and placed in inter-bucket buffer, and then sent to all sub bucket processes. Since relation *Edge* only has one subbucket in this example, no data transmission occur here. Figure 4 shows the difference it can make to select the smaller of the two relations as the outer one when using this design for parallel RA.

Algorithm 1 Join-order Selection Algorithm

```
localOuter ← relation1.size ≥ relation2.size ? 0 : 1
ranksWantOuter ← MPI_Allreduce(MPI_Sum, localOuter)
                                  ▷ All ranks synchronize here
InnerRelation ← relation1
OuterRelation ← relation2
if ranksWantOuter ≥ (TotalRankNum ÷ 2) then
    swap(OuterRelation,InnerRelation)
end if
intra_bucket_comm_buffer ← OuterRelation.serialize()
```

Optimizing communication may not be done via comparing

relation sizes alone. For example, the larger overall relation may have fewer tuples on a specific rank, complicating communication optimization. Our dynamic join planning utilizes a simple voting Algorithm 1 to coordinate each process and decide which relation should be used on the outer side. In a binary join $output \leftarrow relation_1(...), relation_2(...)$, before the start of each iteration, each process performs (in parallel) a local relation size comparison: if $relation_1$ is smaller, outer join local flag will be set to 1. Next, a collective *MPI_Allreduce* operation will quickly tally the results of the vote and decide join direction. If the summation is more than half of the total process number, $relation_1$ will be set the outer relation. ~~The inclusion of *MPI_Allreduce* incurs an extra synchronization, but our results indicate it is worth the payoff given the algorithmic improvement we observe. Given that static join planning is intractable in general, we believe that our communication-avoiding join direction heuristic is a useful form of dynamic join planning.~~ **The inclusion of *MPI_Allreduce* during outer relation selection does introduce additional communication overhead. However, it is necessary for ensuring that all nodes use the same outer relation. In Algorithm 1, it is evident that the communication data size in this voting step is limited to a single 8-bit integer having maximum value equals to number of processes. Despite this overhead, our experiments in Section V-B demonstrate that it significantly reduces the size of outer relation transmission and the computation required for local joins on real-world datasets. This optimization helps avoid the much greater overhead associated with poorly-operationalized joins.** Our technique takes inspiration from the dynamic plans of Soufflé and query planners for SQL, though our particular form of dynamic join planning is customized to avoid communication in the setting of iterated relational joins and recursive aggregates. **Similar to dynamic join planning, there is no formal guarantee that our communication-avoiding approach will universally accelerate join operations in arbitrary queries and input data. However, our large-scale experiments in Section V demonstrate promising results, indicating the practical usefulness of our approach in real-world scenarios.**

V. Evaluation

We evaluated PARALAGG, asking three research questions:

- RQ1 What is the impact of our optimization technique?
- RQ2 How do queries by PARALAGG perform compared to state-of-the-art systems on large servers?
- RQ3 How do queries written using PARALAGG scale on leadership-class supercomputers?

We evaluate **RQ1** by analysing data distribution and speed difference between optimized program and baseline version. We evaluate **RQ2** by comparing the runtime of PARALAGG (at several thread counts) versus two recursive aggregate engines (RaSQL [3] and SociaLite [11]) on a mid-size server. We evaluate **RQ3** by running CC and SSSP on Supercomputer at up to 16,384 processe.

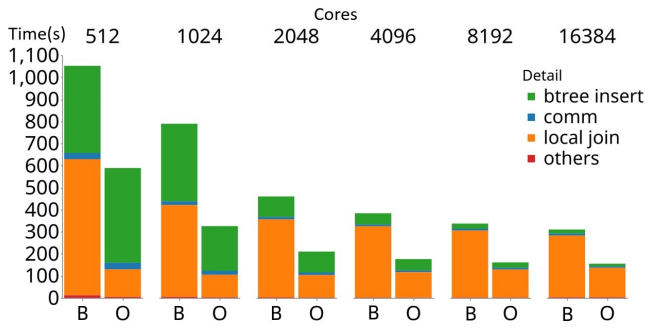

Fig. 4. SSSP running time with Communication avoiding join layout optimization off/on: Baseline ("B") vs. optimized ("O").(On twitter dataset)

In **RQ1** and **RQ3**, we used the Theta supercomputer [13]. Theta is a Cray XC40 machine deployed at Argonne Leadership Computing Facility. It has 4,392 computation nodes, each node contains a 64 core Intel Phi Knights Landing 7230@1.3Ghz and 192 GiB DDR4. Theta also supports MCDRAM and hyperthread on every computation node (in our experiments, no hyperthread is used). We set MCDRAM is set to flat mode, and NUMA mode to "quad." We turn on sub-bucketing and join direction optimization for all test running on Theta. The bucket size of the input relation is initially set to 8 sub buckets per rank. And in both testing we used a snapshot of Twitter from 2010 [14]. The Twitter graph contains 1,468,365,182 edges, roughly 35GB in size. In **RQ2** testing, Our experiments were run on a server running Ubuntu 20.04 LTS with an AMD EPYC 7713P 64-Core (128-thread) 1.996GHz processor and 480GiB RAM.

A. Queries

Single Source Shortest Path (SSSP): As we mentioned in section II, SSSP can be expressed elegantly using PARALAGG. It serves as a useful benchmark in measuring the throughput of recursive aggregates, as the recursive aggregate forms a tight loop. We designate ten arbitrarily selected start nodes from each of our graphs.

Connected Components (CC): Connected components are computed as follows:

$$\begin{array}{lcl} cc(n,n) & \leftarrow & edge(n,\_). \\ cc(y,\$\mathrm{MIN}(z)) & \leftarrow & cc(y,z), edge(x,y). \\ cc\_representive\_node(n) & \leftarrow & cc(\_,n). \end{array}$$

The $MIN aggregate canonicalizes a component representative, efficiently compressing connected components in space. By constrast, implementations of *CC* in Datalog engines run out of memory due to materialization overhead and the inability to avoid materializing a product of all nodes within the component.

B. RQ1: Measuring the Effects of our Optimizations

Communication-Avoiding Join Layout: ~~We compared the performance of an unoptimized SSSP program with~~

~~our communication-optimized join approach, using the same settings and sub-bucket number of 1 to eliminate load balancing effects.~~. **We compared the performance of the basic setting (baseline) SSSP program in PRALAGG with the communication-avoiding join optimization enabled. Both the optimized and basic setting programs were tested on the Twitter dataset with a sub-bucket number of 1 to eliminate the load balancing effect, while keeping all other settings the same.** The optimized program's running time was cut in half compared to the ~~unoptimized~~ **baseline** program, as shown in Figure 4. The improvement was mainly in local join computation, particularly at 512 cores where the optimized program's local join time was only 20% of the ~~unoptimized~~ **baseline** program's. This is because the *spath* relation is much smaller than the *edge* relation at beginning. Serialization of edges mistakenly placed on the left side in the join would cause significant memory consumption and almost reduce the join operation to 1 billion times linear comparison in some iterations. With optimization, *spath* is automatically placed on the outer side, making the join speed closer to a BTree search operation. **The "Comm" time, representing the communication for distributing join results, was independent of our optimized join layout and implemented using MPI_Alltotallv. Therefore, our optimization specifically targeting the join phase does not accelerate it**

Spatial Balancing: In Figure 4, we observed that program scaling nearly halts after 2028 cores. Breaking down the running time reveals that while optimization reduces the local join operation's overall running time, it does not impact the data distribution of tuples on each process. Turning off sub-bucket load balancing causes data imbalance across ranks. Twitter is a social network dataset, and some users have millions of followers, causing all edges with the same starting node to be stored on the same rank. ~~Figure 5 shows that without sub-bucketing, the largest rank has ten times more tuples than the smallest rank. Load balancing incurs overhead, as shown in Figure 6, and is only worthwhile when there are more than 1,000 nodes.~~ **We ran a CC query on the Twitter dataset to evaluate the effectiveness of PARALAGG's spatial load balancing. Figure 5 shows that without sub-bucketing, the largest rank had ten times more tuples than the smallest rank, but with 8 sub-buckets, the difference was reduced to around 2 times, indicating successful data imbalance mitigation.Furthermore, we compared the scalability and local join computation time of 8 sub-buckets and 1 sub-bucket, as shown in Figure 6. Due to imbalance, the query with 1 sub-bucket reached its scalability limit after 2048 processes. Beyond this point, adding more cores slowed down the query. However, the balanced program outperformed the imbalanced program significantly after 4096 processes and continued to scale up to 16384 processes. We also observed that the balanced program has longer running times compared to the imbalanced program when the number of processes is less than 1024. This can be attributed to the additional inter-bucket communication required for spatial load balancing, as explained in Section IV-C.** However, our work targets supercomputers, and 1024 ranks is a relatively small number of cores.

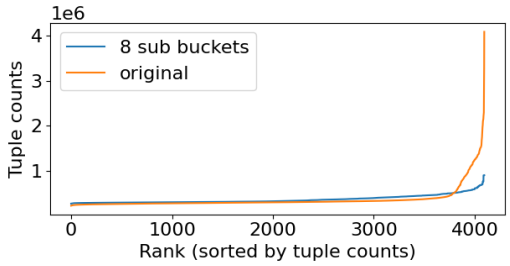

Fig. 5. Twitter dataset tuple distribution on 4096 ranks after sub bucketing optimization.

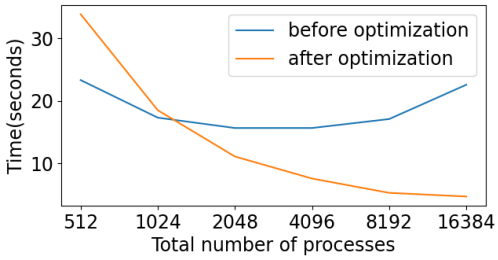

Fig. 6. CC query's local join computation time after enabling sub-bucketing (run on Theta with Twitter dataset).

C. RQ2: Comparisons against SOTA systems on large servers

In this subsection, We run experiments at three distinct scales: 32, 64, and 128 threads (128-thread experiments utilize SMT). `C++` code generated by PARALAGG is compiled with `gcc` version 11 and OpenMPI 4.1.2. The resulting binary is then invoked using `mpirun --use-hwthread-cpus`; wall-clock times are measured via GNU `time`; timings for SociaLite and RaSQL are captured using their built-in counters. SociaLite is tested in single-node mode (circumventing network-imposed overhead) with Java 1.7. SociaLite's parallelization also requires manual partitioning of each relation, which we achieve using `indexby` keywords. For RaSQL, we use Java 1.8 and Spark 2.0.3. We set up RaSQL according to its manual, setting data partitions equal to core count and turning on hash shuffle optimizations. We allocate 350GB for Java's heap. All reported results are best of five runs.

We ran experiments using three graphs of varying sizes. The first two, Livejournal and Orkurt, are medium-sized graphs from the SNAP Graph dataset [15], each consisting of roughly 100 million edges. The last graph, Topcats [16], is smaller, with only 25 million edges; we included Topcats to stress PARALAGG's performance with lower data loads, when the overhead of tuple distribution may not pay off. We selected five arbitrary nodes from each graph as entry points for SSSP.

Table I presents our experimental results, which are divided into SSSP and CC queries. The table shows runtimes for each tool on a per-graph basis, with results for PARALAGG, RaSQL,

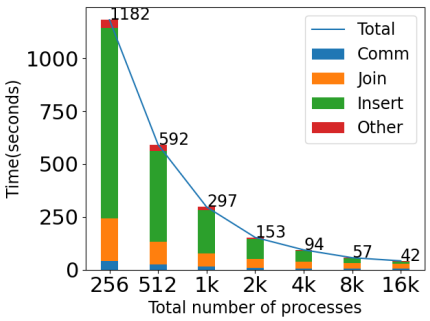

Fig. 7. Scaling SSSP query on Theta, using Twitter dataset.

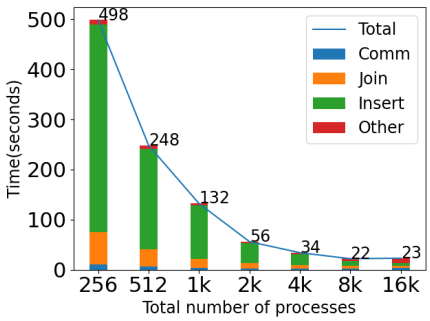

Fig. 8. Scaling CC query on Theta, using Twitter dataset.

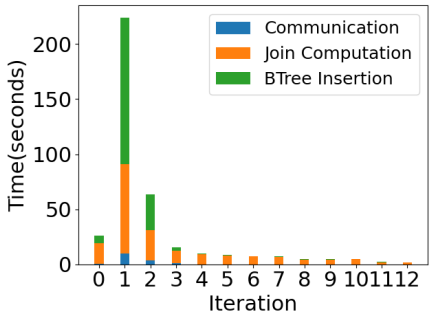

Fig. 9. Detail running times of each iteration for the SSSP on the Twitter Dataset (ran on Theta).

	Graph	Tool	Thread Count 32	64	128
Shortest Paths	LiveJournal	PARALAGG	0:31	0:19	**0:11**
		RaSQL	0:17	0:12	0:12
		SociaLite	1:06	0:37	0:41
	Orkurt	PARALAGG	0:29	0:19	**0:11**
		RaSQL	0:14	0:14	0:17
		SociaLite	0:42	0:43	0:45
	Topcats	PARALAGG	**0:04**	0:07	0:14
		RaSQL	0:13	0:17	0:23
		SociaLite	0:57	0:53	0:50
Connected Components	LiveJournal	PARALAGG	1:21	0:50	**0:32**
		RaSQL	2:06	1:51	1:54
		SociaLite	1:30	1:11	1:27
	Orkurt	PARALAGG	2:01	1:06	**0:36**
		RaSQL	0:58	0:59	0:56
		SociaLite	3:04	2:49	2:51
	Topcats	PARALAGG	0:18	**0:13**	0:23
		RaSQL	0:24	0:32	0:51
		SociaLite	0:47	0:39	0:41

TABLE I
SINGLE-NODE EXPERIMENTS COMPARING PARALAGG, RASQL, AND SOCIALITE ON A 64-CORE (128-THREAD) SERVER. TIMES ARE MINUTES:SECONDS.

and SociaLite displayed in separate rows, and columns indicating the number of threads used. The fastest runtime for each set of experiments is highlighted in bold. Our results indicate that PARALAGG provides the consistently fastest implementation at full thread count compared to RaSQL and SociaLite. However, in certain situations, particularly at lower thread counts, PARALAGG's extra communication overhead caused by sub-bucketing and dynamic join order optimization may not pay off, and tuple number on each rank can be easily balanced. For example, the CC query for Orkurt at 32 threads took PARALAGG two minutes and one second, versus only 58 seconds for RaSQL. However, when scaling up to 128 threads, the runtime reduced to only 36 seconds. While the PARALAGG-based implementations demonstrate much more satisfactory scalability than either RaSQL or SociaLite (which achieve only marginal scalability), scalability is limited when no more work is available, and parallel benefits will quickly be surpassed by communication overhead. For instance, the CC query for Topcats took 13 seconds at 64 threads, while it took 23 seconds at 128 threads. We also tested the systems on a Twitter graph benchmark, but both RaSQL and SociaLite failed due to integer overflow, while PARALAGG successfully completed the queries in slightly under an hour.

D. RQ3: Scaling on Theta

In previous testing, we found using larger nodes or more cores can't improve socialite and RaSQL's performance.Therefore, we consider it unnecessary to scale them on machines with more cores. However PARALAGG's running time continued to decrease, so We next sought to understand how PARALAGG would scale on a leadership-class supercomputer. In order to increase problem size, we run SSSP by running on 30 arbitrarily-picked start nodes simultaneously.

Before performing strong-scaling runs on Theta, we verified that PARALAGG scaled well and produced correct results by running in Theta's debug queue (512 physical cores in total). We selected 8 different graphs from SuiteSparse [5], varying in size, category, and graph properties such as sparsity and betweenness centrality. Our results are shown in Table II, indicating generally favorable scalability and near-ideal performance improvements when scaling from 256 to 512 processes, with scalability gains being more apparent on larger graphs.

We then conducted strong scaling tests on Theta and plotted the results for the SSSP query on the Twitter dataset, ranging from 256 to 16,384 cores (Fig 7). The graph shows that SSSP's running time on the Twitter dataset decreased by 96% from 256 to 16,384 cores, exhibiting near-perfect scalability until 2048 cores. While performance improvements slow down after 2048 cores, we still observed a 26% performance improvement when scaling from 8,096 to 16,384 cores. Our analysis revealed that BTree insertion dominated program performance at low core counts. However, as the core count increased, the BTree size on each rank scaled nearly-linearly, resulting in nearly-linear scaling for insertion and updating

		Shortest Paths				Connected Components		
Graph	Edges	Iters	Paths	256	512	Comp	256	512
flickr	9.8M	16	22M	14.4	9.3	0.6M	4.3	2.4
Freescale1	19.0M	126	9.8M	36.4	20.1	3.4M	32.4	17.4
wiki	37.2M	366	55.3M	49.8	27.5	28.9M	15.9	10.3
wb-edu	57.2M	242	15.5M	60.8	31.5	69.2M	47.1	26.3
ML_Geer	110.8M	500	43.6M	161.9	88.5	1.5M	1647.5	851.5
HV15R	283.1M	75	44.4M	320.6	164.3	2.0M	570.6	294.2
arabic	640.0M	52	402.5M	569.2	289.7	194.4M	353.6	181.9
stokes	349.3M	367	327.8M	644.3	326.4	11.4M	1755.5	892

TABLE II
MEDIUM-SCALE EXPERIMENTS ON A VARIETY OF GRAPHS FROM THE SUITESPARSE MATRIX COLLECTION [5]. PERFORMANCE OF SSSP AND CC ARE SHOWN AT 256 AND 512 PROCESSES ON THETA. ALL TIMES IN SECONDS.

operations all the way to 16,384 cores. Local join was also a significant factor in SSSP computation, and until 1,024 cores, it scaled mostly-linearly. However, the generation of only a few thousand new delta tuples per iteration caused some processes to starve at higher core counts, leading to non-linear scaling. Additionally, PARALAGG's join order optimization required an extra synchronization phase before the real local join operation, which was slowed down by more processes, causing scalability to gradually saturate after 1,024 cores.

To better understand the running time results of the SSSP query, we analyzed the running time of each iteration when running with 1024 cores, as shown in Figure 9. The results show that the computation of the SSSP query on this large graph has a long-tail dynamic, with most of the running time in the first few iterations, and the local join computation dominating the long tail. The btree operation scales well, which explains the overall good performance and scalability since most of the computation happens in the first few iterations. However, local join operation in long tail scales non-linearly, which explains why scalability drops fast in high core counts testing.

Results for the CC query (Figure 8) were similar to those of the SSSP query, with a 96% decrease in running time from 256 to 16384 cores. Near-perfect scalability was achieved until 2048 cores, with 60% running time improvement from 2048 to 8192 cores. However, at 16384 cores, the total running time stopped decreasing due to the "Other" category taking up half of the computation time. This bottleneck is caused by sub-bucket data rebalancing inducing intra-bucket communication(implemented using MPI_All2allv) overhead, which becomes non-negligible as the process number increases. The benefit of parallelization is gradually surpassed by communication overhead, resulting in decreased scalability.

VI. RELATED WORK

There are several relevant threads of related work.

Recursive and Monotonic Aggregation: Motivated by the algorithmic limitations of vanilla Datalog, its extension to arbitrary lattices has attracted significant interest in the programming languages, logic programming, and graph analytics communities [3], [4], [11], [17]–[22]. Much of this work attempts to reconcile the semantic challenges when adding general lattices to Datalog [3], [11], [23], [24]. PARALAGG's notion of recursive aggregation interleaved with projection ensures that all PARALAGG programs terminate, as the RA kernels provided by our library allow constructing queries which satisfy *Pre-Mappability* [3], [24]. By constrast, our semantics are implementation-focused, inspired by modern graph-analytics languages targeting recursive aggregation, namely RaSQL (RaSQL is built on an improved version of BigDatalog' query engine and use a new SQL-like query language called RaSQL), DeALS, and BigDatalog [2]–[4].

High-Performance Datalog Engines: Datalog has been an increasingly-popular target for optimized high-performance implementations given its applications to large-scale efforts in program analysis [25]–[27], graph analytics [4], [28], [29], and related fields. Initial efforts in scaling Datalog focused on novel representations (*e.g.*, binary decision diagrams [26]) to enable efficient joins or set operations (*e.g.*, leapfrog triejoin in LogicBlox [30]). Unfortunately, these representations have proven challenging to parallelize and distribute, and modern engines (namely Soufflé) rely upon explicit representations which use shared-memory datastructures and compile to efficient relational algebra kernels implemented via high-performance native code [6], [31]. Recently, there has been significant work in extending these engines to high-performance recursive aggregate queries on parallel unified-memory architectures. For example, recently Soufflé added union-find datastructures (a kind of lattice) [32], and DCDatalog works to scale recurisve aggregate queries while avoiding memory contention. Current efforts in distributed Datalog (*e.g.* DCDatalog and Cog [33]) turn away from Spark and MapReduce-style parallelism (increasingly understood to be a source of overhead [34], [35]); that work explores many related but orthogonal directions, primarily concerned with implementing lock-free (rather than communication-avoidant) algorithms. Our work, utilizing high-performance kernels implemented directly via MPI, is most directly related to the recent work of Kumar et al. [8], [9], which excludes lattices and monotonic aggregates.

Dynamic Query Plan Optimization: Relational query planning has a long history, largely within the context of RDBMS systems [36]–[40], often powered by cardinality estimation methods (*e.g.*, the IK/KBZ family of algorithms [41]). Modern work in join plan compilation focuses on efficiently utilizing modern architectural elements, [42], parallelizing plan selection [43], and dynamic switching [44]. The communication-optimized join switching in PARALAGG is largely orthogonal to this work, inspired more closely by the notion of communication-avoiding algorithms [45].

High-Performance Aggregate Queries: Our work focuses on the implementation of general-purpose relational algebra kernels to enable the rapid implementation of a broad class of problems. However, a wide body of work exploits problem (or hardware)-specific knowledge to scale the implementation of SSSP [46]–[50], connected components [51], union-find [52]. **These tools focus solely on graph processing and do not support recursive aggregation and other relational algebra operations. Hence, in this paper our evaluation specifically targeted the best publicly-available declarative engines (RadLog, SociaLite, and BigDatalog).** As future work, we hope to study how these application-specific insights may be applied to further scale programs written using PARALAGG.

VII. CONCLUSION

Recursive aggregation is a highly useful extension to standard reachability-based reasoning, forming the basis for many important problems which interpose monotonic aggregation with chain-forward reasoning. Unfortunately, the challenge of mixing semi-naïve evaluation with a distributed tuple representation poses serious challenges to scalability of recursive aggregates in a high performance setting, where the need for communication often hampers end-to-end throughput. For example, while several significant efforts to implement recursive aggregates have focused on distributed architectures [2]–[4], [53], the fastest current methods utilize unified machines with shared memory [1].

In this paper, we present the design of communication-avoiding algorithms to implement recursive aggregates at a scale never before seen. We leverage the observation that the semantic properties justifying well-founded recursive aggregation (*e.g.*, **PreM**) are of a restricted form which allow communication-avoiding implementations of recursive aggregates in a highly-parallel way (via local aggregation). We present PARALAGG, a C++ library which offers high-performance relational algebra kernels extended with recursive aggregates. Additionally, we present a new approach to on-the-fly join layout based on domain-specific communication issues. We demonstrate how PARALAGG approach allows expressing common recursive aggregate queries, and run a variety of experiments on mid-size servers and leadership-class supercomputers. Our results show that PARALAGG outperforms comparable state-of-the-art tools, demonstrating healthy scalability to tens of thousands of cores of the Theta supercomputer, and validating the usefulness of our heuristic-based dynamic join layout algorithm in applying recursive aggregation to large graphs.

REFERENCES

[1] J. Wu, J. Wang, and C. Zaniolo, "Optimizing parallel recursive datalog evaluation on multicore machines," in *Proceedings of the 2022 International Conference on Management of Data*, pp. 1433–1446, 2022.

[2] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, "Big data analytics with datalog queries on spark," in *Proceedings of the 2016 International Conference on Management of Data*, pp. 1135–1149, 2016.

[3] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo, "Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark," in *Proceedings of the 2019 International Conference on Management of Data*, pp. 467–484, 2019.

[4] M. Mazuran, E. Serra, and C. Zaniolo, "Extending the power of datalog recursion," *The VLDB Journal*, vol. 22, no. 4, pp. 471–493, 2013.

[5] T. A. Davis and Y. Hu, "The university of florida sparse matrix collection," *ACM Trans. Math. Softw.*, vol. 38, dec 2011.

[6] H. Jordan, B. Scholz, and P. Subotić, "Soufflé: On synthesis of program analyzers," in *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II 28*, pp. 422–430, Springer, 2016.

[7] K. A. Ross and Y. Sagiv, "Monotonic Aggregation in Deductive Databases," in *Proceedings of the eleventh ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*, pp. 114–126, 1992.

[8] S. Kumar and T. Gilray, "Load-balancing parallel relational algebra," in *High Performance Computing: 35th International Conference, ISC High Performance 2020, Frankfurt/Main, Germany, June 22–25, 2020, Proceedings*, (Berlin, Heidelberg), p. 288–308, Springer-Verlag, 2020.

[9] T. Gilray and S. Kumar, "Distributed relational algebra at scale," in *2019 IEEE 26th International Conference on High Performance Computing, Data, and Analytics (HiPC)*, pp. 12–22, 2019.

[10] J. Wang, J. Wu, M. Li, J. Gu, A. Das, and C. Zaniolo, "Formal semantics and high performance in declarative machine learning using datalog," *VLDB J.*, vol. 30, no. 5, pp. 859–881, 2021.

[11] J. Seo, J. Park, J. Shin, and M. S. Lam, "Distributed socialite: A datalog-based language for large-scale graph analysis," *Proceedings of the VLDB Endowment*, vol. 6, no. 14, pp. 1906–1917, 2013.

[12] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*. Cambridge University Press, 2 ed., 2002.

[13] A. L. C. Facility, "Theta' overview and user manual." https://www.alcf.anl.gov/support/user-guides/index.html.

[14] H. Kwak, C. Lee, H. Park, and S. Moon, "What is twitter, a social network or a news media?," in *Proceedings of the 19th international conference on World wide web*, pp. 591–600, 2010.

[15] J. Leskovec and A. Krevl, "SNAP Datasets: Stanford large network dataset collection." http://snap.stanford.edu/data, June 2014.

[16] H. Yin, A. R. Benson, J. Leskovec, and D. F. Gleich, "Local higher-order graph clustering," in *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*, pp. 555–564, 2017.

[17] M. Madsen, M.-H. Yee, and O. Lhoták, "From datalog to flix: A declarative language for fixed points on lattices," *ACM SIGPLAN Notices*, vol. 51, no. 6, pp. 194–208, 2016.

[18] K. A. Ross and Y. Sagiv, "Monotonic aggregation in deductive databases," *Journal of Computer and System Sciences*, vol. 54, no. 1, pp. 79–97, 1997.

[19] M. A. Khamis, H. Q. Ngo, R. Pichler, D. Suciu, and Y. Remy Wang, "Datalog in wonderland," *SIGMOD Rec.*, vol. 51, p. 6–17, jul 2022.

[20] M. Arntzenius and N. R. Krishnaswami, "Datafun: a functional datalog," in *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, pp. 214–227, 2016.

[21] S. Ganguly, S. Greco, and C. Zaniolo, "Minimum and maximum predicates in logic programming," in *Proceedings of the Tenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, PODS '91, (New York, NY, USA), p. 154–163, Association for Computing Machinery, 1991.

[22] J. Wang, G. Xiao, J. Gu, J. Wu, and C. Zaniolo, "Rasql: A powerful language and its system for big data applications," in *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, SIGMOD '20, (New York, NY, USA), p. 2673–2676, Association for Computing Machinery, 2020.

[23] R. Krishnamurthy and S. A. Naqvi, "Non-deterministic choice in datalog," in *International Conference on Data and Knowledge Bases*, 1988.

[24] T. CONDIE, A. DAS, M. INTERLANDI, A. SHKAPSKY, M. YANG, and C. ZANIOLO, "Scaling-up reasoning and advanced analytics on bigdata," *Theory and Practice of Logic Programming*, vol. 18, no. 5-6, p. 806–845, 2018.

[25] M. Bravenboer and Y. Smaragdakis, "Strictly declarative specification of sophisticated points-to analyses," in *Proceedings of the 24th ACM SIGPLAN conference on Object oriented programming systems languages and applications*, pp. 243–262, 2009.

[26] J. Whaley, D. Avots, M. Carbin, and M. S. Lam, "Using datalog with binary decision diagrams for program analysis," in *Programming Languages and Systems* (K. Yi, ed.), (Berlin, Heidelberg), pp. 97–118, Springer Berlin Heidelberg, 2005.

[27] T. Antoniadis, K. Triantafyllou, and Y. Smaragdakis, "Porting doop to soufflé: A tale of inter-engine portability for datalog-based analyses," in *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*, SOAP 2017, (New York, NY, USA), p. 25–30, Association for Computing Machinery, 2017.

[28] W. Moustafa, V. Papavasileiou, K. Yocum, and A. Deutsch, "Datalography: Scaling datalog graph analytics on graph processing systems," in *2016 IEEE International Conference on Big Data (Big Data)*, (Los Alamitos, CA, USA), pp. 56–65, IEEE Computer Society, dec 2016.

[29] Z. Fan, J. Zhu, Z. Zhang, A. Albarghouthi, P. Koutris, and J. M. Patel, "Scaling-up in-memory datalog processing: Observations and techniques," *Proceedings of the VLDB Endowment*, vol. 12, no. 6, 2019.

[30] M. Aref, B. ten Cate, T. J. Green, B. Kimelfeld, D. Olteanu, E. Pasalic, T. L. Veldhuizen, and G. Washburn, "Design and implementation of the logicblox system," in *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, (New York, NY, USA), p. 1371–1382, Association for Computing Machinery, 2015.

[31] H. Jordan, P. Subotić, D. Zhao, and B. Scholz, "Brie: A specialized trie for concurrent datalog," in *Proceedings of the 10th International Workshop on Programming Models and Applications for Multicores and Manycores*, PMAM'19, (New York, NY, USA), p. 31–40, Association for Computing Machinery, 2019.

[32] P. Nappa, D. Zhao, P. Subotić, and B. Scholz, "Fast parallel equivalence relations in a datalog compiler," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pp. 82–96, 2019.

[33] M. Imran, G. E. Gévay, and V. Markl, "Distributed graph analytics with datalog queries in flink," in *Software Foundations for Data Interoperability and Large Scale Graph Data Analytics* (L. Qin, W. Zhang, Y. Zhang, Y. Peng, H. Kato, W. Wang, and C. Xiao, eds.), (Cham), pp. 70–83, Springer International Publishing, 2020.

[34] M. Anderson, S. Smith, N. Sundaram, M. Capotă, Z. Zhao, S. Dulloor, N. Satish, and T. L. Willke, "Bridging the gap between hpc and big data frameworks," *Proc. VLDB Endow.*, vol. 10, p. 901–912, apr 2017.

[35] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on hadoop vs mpi/openmp on beowulf," *Procedia Computer Science*, vol. 53, pp. 121–130, 2015. INNS Conference on Big Data 2015 Program San Francisco, CA, USA 8-10 August 2015.

[36] S. Abiteboul, R. Hull, and V. Vianu, *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[37] R. Krishnamurthy, H. Boral, and C. Zaniolo, "Optimization of nonrecursive queries.," in *VLDB*, vol. 86, pp. 128–137, 1986.

[38] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access path selection in a relational database management system," in *Proceedings of the 1979 ACM SIGMOD international conference on Management of data*, pp. 23–34, 1979.

[39] S. Cluet and G. Moerkotte, "On the complexity of generating optimal left-deep processing trees with cross products," in *Database Theory—ICDT'95: 5th International Conference Prague, Czech Republic, January 11–13, 1995 Proceedings 5*, pp. 54–67, Springer, 1995.

[40] W. Cai, M. Balazinska, and D. Suciu, "Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities," in *Proceedings of the 2019 International Conference on Management of Data*, SIGMOD '19, (New York, NY, USA), p. 18–35, Association for Computing Machinery, 2019.

[41] T. Ibaraki and T. Kameda, "On the optimal nesting order for computing n-relational joins," *ACM Transactions on Database Systems (TODS)*, vol. 9, no. 3, pp. 482–502, 1984.

[42] T. Neumann and B. Radke, "Adaptive optimization of very large join queries," in *Proceedings of the 2018 International Conference on Management of Data*, pp. 677–692, 2018.

[43] R. Mancini, S. Karthik, B. Chandra, V. Mageirakos, and A. Ailamaki, "Efficient massively parallel join optimization for large queries," in *Proceedings of the 2022 International Conference on Management of Data*, SIGMOD '22, (New York, NY, USA), p. 122–135, Association for Computing Machinery, 2022.

[44] S. Arch, X. Hu, D. Zhao, P. Subotić, and B. Scholz, "Building a join optimizer for soufflé," in *Logic-Based Program Synthesis and Transformation: 32nd International Symposium, LOPSTR 2022, Tbilisi, Georgia, September 21–23, 2022, Proceedings*, pp. 83–102, Springer, 2022.

[45] J. Demmel, "Communication-avoiding algorithms for linear algebra and beyond," in *2013 IEEE 27th International Symposium on Parallel and Distributed Processing*, pp. 585–585, 2013.

[46] X. Gan, Y. Zhang, R. Wang, T. Li, T. Xiao, R. Zeng, J. Liu, and K. Lu, "Tianhegraph: Customizing graph search for graph500 on tianhe supercomputer," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 4, pp. 941–951, 2022.

[47] Y. Wang, H. Cao, Z. Ma, W. Yin, and W. Chen, "Scaling graph 500 sssp to 140 trillion edges with over 40 million cores," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '22, IEEE Press, 2022.

[48] Y. Zhang, X. Liao, H. Jin, L. He, B. He, H. Liu, and L. Gu, "Depgraph: A dependency-driven accelerator for efficient iterative graph processing," in *2021 IEEE International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 371–384, 2021.

[49] F. Checconi and F. Petrini, "Traversing trillions of edges in real time: Graph exploration on large-scale parallel machines," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 425–434, 2014.

[50] V. T. Chakaravarthy, F. Checconi, F. Petrini, and Y. Sabharwal, "Scalable single source shortest path algorithms for massively parallel systems," in *2014 IEEE 28th International Parallel and Distributed Processing Symposium*, pp. 889–901, 2014.

[51] Y. Zhang, A. Azad, and A. Buluç, "Parallel algorithms for finding connected components using linear algebra," *Journal of Parallel and Distributed Computing*, vol. 144, pp. 14–27, 2020.

[52] F. Manne and M. M. A. Patwary, "A scalable parallel union-find algorithm for distributed memory computers," in *Parallel Processing and Applied Mathematics* (R. Wyrzykowski, J. Dongarra, K. Karczewski, and J. Wasniewski, eds.), (Berlin, Heidelberg), pp. 186–195, Springer Berlin Heidelberg, 2010.

[53] M. Imran, G. E. Gévay, J.-A. Quiané-Ruiz, and V. Markl, "Fast datalog evaluation for batch and stream graph processing," *World Wide Web*, vol. 25, no. 2, pp. 971–1003, 2022.