Communication-Avoiding Recursive Aggregation

Yihao Sun
Dept. Electrical Engineering and Computer Science
Syracuse University
Syracuse, USA
ysun67@syr.edu

Sidharth Kumar
Dept. Computer Science
University of Illinois at Chicago
Chicago, USA
sidharth@uic.edu

Thomas Gilray
Dept. Computer Science
University of Alabama at Birmingham
Birmingham, USA
gilray@uab.edu

Kristopher Micinski
Dept. of Electrical Engineering and Computer Science
Syracuse University
Syracuse, USA
kkmicins@syr.edu

***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 node 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

A. Relational Algebra

Relational algebra (RA) comprises a set of mathematical primitives which operate over *tables* of *tuples* of some fixed arity; in the context of a relational database, each tuple corresponds to a row. Standard relational algebra operators include selection (σ), which filters tuples from a relation based on a specified condition, projection (Π), cartesian product ($\times$), renaming (ρ), and join ($\bowtie$), which combines two relations based on a common attributes. Used in the appropriate combination, these RA operators intuitively correspond to queries in relational query languages (namely SQL).

We are interested in *recursive* queries, which extend the expressive power of SQL to enable queries including graph reachability and program analysis [6]. Such queries comprise the Datalog language; Datalog programs consist of a set of rules, each of which is a Horn clause (clauses involving a single positive literal). For example, we may write transitive closure in Datalog as follows:

$$\begin{array}{lcl} \mathit{Path}(x,y) & \leftarrow & \mathit{Edge}(x,y). \\ \mathit{Path}(x,z) & \leftarrow & \mathit{Path}(x,y), \mathit{Edge}(y,z). \end{array}$$

Modern Datalog engines leverage a rich history of work in the databases community which understands such queries as efficient RA plans [7], [8]. These engines proceed in iterations, generating new tuples based on the results of an RA operation which is subsequently merged into the database. Specific to this example, the *Path* table is extended (via set union) with the results of both queries:

$$\mathit{Path} \quad \leftarrow \quad \mathit{Path} \cup \mathit{Edge} \cup \mathit{Path} \bowtie (\rho_{1/0}\ \mathit{Edge})$$

The first *Path* is due to the fact that all tuples in *Path* stay in *Path*, the inclusion of *Edge* corresponds to the first rule (which copies *Edge* into *Path*), and the join between *Path* and *Edge* operationalizes the last (inductive) rule.

B. Stratified Aggregation

Unlike SQL queries (which terminate by virtue of an obviously-finite state space), the tractability of Datalog queries is carefully constructed to ensure that Datalog programs enumerate a finite number of tuples. Vanilla Datalog achieves this by allowing only Horn clauses over a finite set of atoms (which may not be extended during computation), however practical implementations include Turing-equivalent extensions with operations such as builtins, negation, and aggregation. To ensure a computational interpretation, only *stratified* negation and aggregation are allowed: this enables straightforward aggregation and trivially deciding $\neg R$ via inclusion [9].

As an example of stratified aggregation and its shortcomings, we present this attempt at single-source shortest paths (SSSP) in Datalog (*Edge*'s third column is a length):

$$\begin{array}{lcl} \mathit{Path}(n,n,0) & \leftarrow & \mathit{Start}(n). \\ \mathit{Path}(\mathit{from},\mathit{to},l+n) & \leftarrow & \mathit{Path}(\mathit{from},\mathit{mid},l), \\ & & \mathit{Edge}(\mathit{mid},\mathit{to},n). \\ \mathit{Spath}(\mathit{from},\mathit{to},\$\textsc{min}(l)) & \leftarrow & Path(\mathit{from},\mathit{to},l). \end{array}$$

The program executes in two strata: the first computes *Path* recursively to a fixed-point, and the second stratum computes *Spath* by aggregating $\$\textsc{min}$ over the set of values for each pair $(\mathit{from},\mathit{to})$. This issue with this query is its poor asymptotic performance compared to Dijkstra's algorithm [10]. When we inspect the join plan, we can observe the algorithmic overhead:

$$\begin{array}{lcl} \mathit{Path} & = & \mathit{Path} \ \cup \ \Pi_{(\mathit{from},\mathit{to},l+n)}(Path \bowtie \mathit{Edge}) \\ \mathit{Spath} & = & \Pi_{(\mathit{from},\mathit{to},\$\textsc{min}(l))}(\mathit{Path}) \end{array}$$

C. Recursive Aggregation

Solving the materialization challenge inherent to the previous SSSP query is impossible in vanilla Datalog. Implementing SSSP with the desired asymptotic complexity requires the use of *recursive aggregation*: Horn clause rules in a loop with a monotonic usage of aggregation. Harmoniously integrating recursive aggregation with Datalog requires semantic care, especially to ensure termination. The next section details our approach, for now we sketch an improved version of SSSP using recursive aggregates provided by our tool ParaLagg:

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

The improved asymptotic complexity of the above program is crucially improved by the usage of the $\$\textsc{min}$ aggregate in the head of the recursive rule—it allows performing aggregation at each iteration, collapsing the information to store only the length of the shortest path. Recursive aggregation is strictly more expressive than stratified aggregation, capturing a broad array of graph algorithms beyond Datalog's reach including SSSP, connected components (CC), and PageRank.

Unlike stratified aggregation, extending Datalog with recursive aggregation requires serious effort, both semantically (establishing a well-founded semantics) and operationally (extended RA kernels supporting recursive aggregates) [11], [12]. The key intellectual complication is the challenge of seminaïve evaluation in the presence of recursive aggregates. Seminaïve evaluation prevents a Datalog engine from unnecessarily recomputing already-known results, providing crucial asymptotic benefits. With respect to plan compilation, this is achieved by splitting each table into several versions: *new* (yet-undiscovered tuples discovered this iteration), Δ (tuples discovered *last* iteration), and *full* (tuples discovered in any previous generation). Using recursive aggregation, we may finally compute SSSP with the desired asymptotic complexity:

$$\begin{array}{lcl} \mathit{Spath}_{\mathit{new}} & = & \Pi_{(\mathit{from},\mathit{to},l+n)}(\mathit{Spath}_{\Delta} \bowtie \mathit{Edge}) \\ \mathit{Spath}_{\Delta} & = & \Pi_{(\mathit{from},\mathit{to},\$\textsc{min}(l))}(\mathit{Spath}_{\mathit{new}} \ \cup \\ & & \quad (\sigma_{(\mathit{from},\mathit{to})}\mathit{Spath}_{\mathit{new}} \ \bowtie \ \mathit{Spath}_{\mathit{full}})) \end{array}$$

In the next section, we explicate a key restriction (obeyed by common recursive aggregates) which enables us to sidestep the necessary communication overhead otherwise necessitated by the combination of recursive aggregation with semi-naïve evaluation. This restriction is key to achieving the strong scaling results we present in section V.

D. Parallel Relational Algebra

Because recursive aggregation strictly extends Datalog, our work builds upon state-of-the-art efforts in the distribution and parallelization of RA kernels on supercomputing clusters. Specifically, we extend Balanced Parallel Relational Algebra (BPRA), a recent approach to iterated relational algebra without aggregation which provides an API of parallel RA primitives that scale to supercomputers [13], [14]. BPRA distributes relations via *double hashing* [15]: tuples are distributed across a cluster via a bucket/sub-bucket decomposition and balanced dynamically. This enables a highly-parallel implementation of binary join which isolates local join communication to communication within a sub-bucket; this avoids the costly all-to-all communication [16], [17] until new tuples are materialized.

The semantic tension between semi-naïve evaluation and aggregation extends to practical challenges when distributing relations, and inhibits the direct application of these methods. Our approach, realized in PARALAGG, targets this intersection of recursive aggregates, semi-naïve evaluation, and distribution. Many distributed 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 [18]. **PreM**, obeyed by all aggregate operators provided by these systems, ensures that semi-naïve evaluation is monotonic even in the presence of partial aggregates; this issue is closely related to communication, as we will soon see.

III. RECURSIVE AGGREGATION: WITH AND WITHOUT COMMUNICATION

We now explain the challenges involved in implementing recursive aggregation in the context of BPRA, using our SSSP query as an example. The high level approach is to implement aggregation as an extension of BPRA's binary joins, which is possible when aggregated columns are not joined upon themselves. Our approach is built on the intuition that a monotonic aggregation generalizes Datalog's fact deduplication, allowing information to be "collapsed" in a local manner. Throughout this section, we will consult Figure 1, which illustrates a single iteration of the inductive rule of SSSP:

$$\textit{Spath}(\textit{from}, \textit{to}, \$\textsc{min}(l+n)) \quad \leftarrow \quad \textit{Spath}(\textit{from}, \textit{mid}, l), \\ \textit{Edge}(\textit{mid}, \textit{to}, n).$$

Our framework construes this as one combined kernel which (a) joins *Spath* and *Edge* and then (b) aggregates the minimum value $l+n$ (c) for each pair of $(\textit{from}, \textit{to})$. For the purposes of illustration, we sketch two ranks only, in practice thousands of ranks may be used. As mentioned, BPRA distributes each relation throughout a cluster via a bucket-sub-bucket decomposition: we observe that both *Spath* and *Edge* are present on both ranks, as would be all other relations. For example, we see two tuples (among others) in *Spath* on rank 1: $(1, 2, 3)$ and $(1, 2, 0)$. These both reside on rank 1 because they have identical join columns (highlighted in yellow).

BPRA's Join Phases: We now review BPRA's iterated joins, in anticipation of our contribution which generalizes BPRA's joins to enable monotonic aggregation. BPRA's joins occur in a sequence of phases, shown at the top of Figure 1. BPRA achieves high throughput by separating joins into a *local* join within a bucket, followed by all-to-all communication to distribute the results. This approach avoids communication by ensuring joins are done in a highly parallel manner, with communication costs proportional to the amount of generated data. As we will soon see, our insight is that this naturally-communication-avoiding structure can be extended to support common usages of recursive aggregation.

Going from left to right, iterations begin with balancing, which shuffles work to aid overall throughput–in this case the tuple $(1, 1, 0)$ 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 discuss dynamic join planning in Section IV-D, but found it to be highly effective in practice, showing performance improvements of $2\times$ (see Figure 2).

Next, we perform intra-bucket communication to replicate tuples across a bucket; in our bucket/sub-bucket distribution, tuples are assigned a bucket based on their join columns and a sub-bucket via their non-join columns. Thus, all tuples sharing the same join column are assigned the same bucket (but not necessarily sub-bucket), and the join result can be computed inside each bucket. However, because two tuples in the same bucket may have different sub-buckets and may reside on different physical ranks, our local join depends upon intra-bucket communication. Intra-bucket communication serializes the smaller relation, replicating it across sub-buckets to enable a local join. After intra-bucket communication, all relevant tuples are gathered on their correct rank, and local join happens in a highly-parallel manner. Once local join finishes, newly-generated tuples will be distributed via their bucket/sub-bucket decomposition.

Deduplication: The last stage in effecting BPRA's binary join is deduplication, which gathers tuples on a single node and assigns each distinct tuple a unique ID. The preceding all-to-all phase sends each newly-generated tuple to its appropriate bucket based on the hash of its join columns. The corresponding rank checks if the tuple exists within its local copy of the relation and, if not, allocates a new tuple and assigns a unique ID via bump-pointer allocation, "materializing" the tuple.

A. Communication-Avoiding Local Aggregation

We now explain our approach to communication-avoiding recursive aggregation as an extension of BPRA's deduplciation. To see our approach in action, see the right side of Figure 1, which sketches two distinct scenarios. On the top, we see that rank 1 currently stores the tuple $(1, 4, 2)$, and has received a new tuple $(1, 4, 5)$. The $\$\textsc{min}$ aggregator operates on the last column; because $5 > 2$, no insertion is performed into *Spath*'s Δ (doing so would constitute excess work), which is sound as no new information has been discovered.

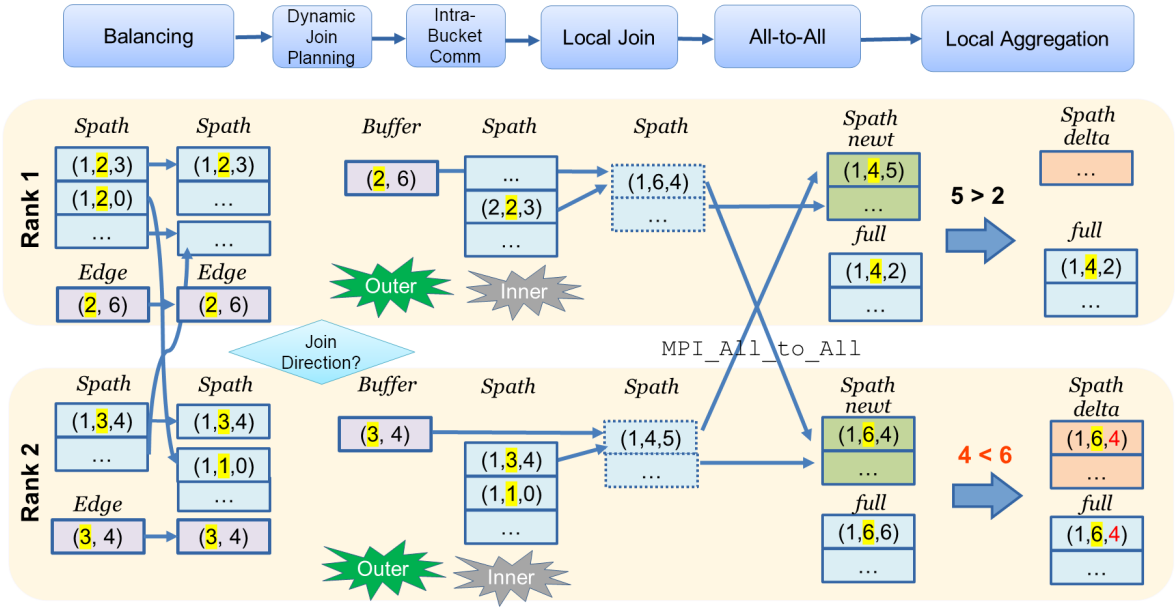

Fig. 1. Sequential flow diagram for a single iteration of the SSSP query in PARALAGG. The *Spath* and *Edge* relations are partitioned across two ranks. Dynamic join planning decides *Edge* should be the outer relation to minimize communication. Local aggregation happens in parallel within each rank.

We observe that 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, which we accomplish by extending BPRA's deduplication. Unfortunately however, state-of-the-art distributed tools do not 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*):

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

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 Δ version of *Spath* are copied to *SpNorm*. However, during the computation of *Spath*, many path lengths are transient 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 [19]).

Formalization: We now present a sketch of our formalization, which extends the set-theoretic and fixed-point semantics of Datalog to chains of deductions on semilattices. Inspired by SociaLite [20] and previous research on recursive aggregators [12], we utilize semilattice-based relations in PARALAGG to extend semi-naive evaluation efficiently. 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 [21]. 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.

After lifting a set-based relation into a semilattice, we can define the following lifted fixpoint evaluation for our aggregated relations $L(r_1, ..., r_n, d_1, ..., d_n)$, where $r_1, ..., r_n$ are independent columns and $d_1, ..., d_n$ are dependent columns:

$$\begin{aligned} L_{i+1} &= \{ \quad (r_1, ..., r_n, \bigsqcup_{dep}) \mid (r_1, ..., r_n) \in L_i \cup L_{\Delta i} \\ &\qquad \wedge\ dep = F_L(r_1, ..., r_n)\} \\ L_{\Delta i+1} &= \{ \quad (r_1, ..., r_n, d_1, ..., d_n) \\ &\qquad \mid \ (r_1, ..., r_n, d_1, ..., d_n) \in L_{i+1} \\ &\qquad \wedge\ (d_1, ..., d_n) \notin \Pi_{(d_1,...,d_n)}(L_i)\} \\ \textit{where} & \\ F_L &= \ \Pi_{d_1,...,d_n}(\sigma_{r_1,...,r_n = x_1,...,x_n} L) \end{aligned}$$

Using the lattice join operator in the computation of L_i guarantees the ascending chain condition and assures that,

assuming the lattice is of finite height, the program eventually terminates.

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. We now describe how our implementation of PARALAGG leverage parallelism and avoid unnecessary communication.

A. Data Distribution

The current 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 multicore machines [1]). These current-generation engines treat aggregated columns in an identical manner to non-aggregated relations. This forces aggregated columns 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 non-aggregated columns for indexing while excluding aggregated columns from the indexing process. By doing so, all tuples sharing the same non-aggregated columns will be automatically gathered together when generated, eliminating the need for additional communication overhead during recursive aggregation. 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.

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 and C++ implementation

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 across ranks 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 [13]. 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 the imbalanced part of data to other processes that have less data. Imbalance arises when multiple tuples in the engine share the same indexed columns. For instance, in the SSSP query (Section II), where the $Edge$ relation is indexed based on the first column, data imbalance may occur if certain vertices in the input graph have many outgoing edges. As a result, processes storing these vertices will have more data compared to others, leading to an uneven distribution of workload. Sub-bucketing effectively resolves this data imbalance, signifi-

cantly improving query scalability. However, sub-bucketing may cause two tuples with the same indexing column to be computed on different processes, violating the requirement for communication-avoiding data distribution. To maintain the correct result, we include a MPI_ALL2ALLV-based intra-bucket communication phase. Although spatial load balancing introduces extra overhead, our experiments in Section V-B demonstrate that it pays off when running with a large number of ranks..

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 1). 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 one of the relations in binary join that is internally stored using a nested BTree data structure must be serialized before it can be transmitted over the network. 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.

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 sub-buckets 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 1, the tuple size of relation *Edge* is much smaller than that of the relation *Spath* on both rank 1 and rank 2. Therefore, both ranks vote for *Edge* as outer relation, and broadcast 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 sub-bucket in this example, no data transmission occurs here. Figure 2 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.

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

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()
```

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 as the outer relation. 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. As we can see in Algorithm 1, the data size used in voting is limited to a single 8-bit integer. Despite this added coordination 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. 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. Ultimately we provide no formal guarantee that our communication-avoiding approach will universally accelerate join operations for arbitrary queries and input data. However, our large-scale experiments in Section V demonstrate promising results, indicating the practical usefulness of our approach for the queries we evaluate.

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 [20]) on a mid-size server. We evaluate **RQ3** by running CC and SSSP on Supercomputer at up to 16,384 processes.

In **RQ1** and **RQ3**, we used the Theta supercomputer [22]. 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

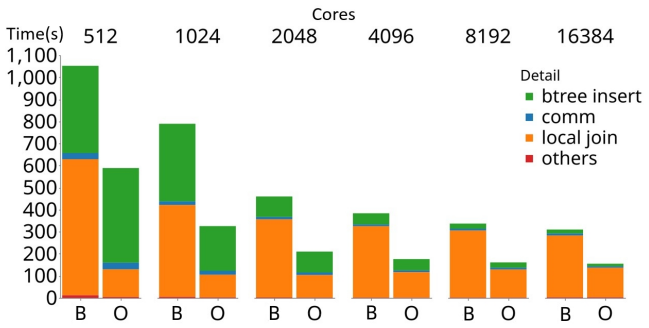

Fig. 2. Strong scaling comparisons for SSSP on Theta (Twitter dataset), broken down by phase. At each process count, we measure a Baseline ("B") and compare against our Optimized ("O") implementation.

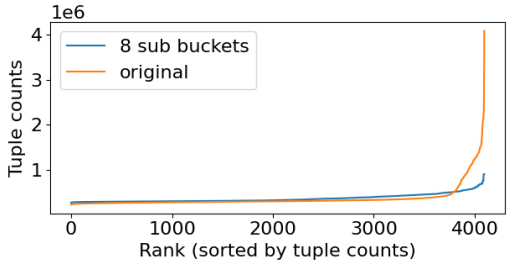

Fig. 3. Cumulative density of tuple distribution across 4,096 ranks, illustrating the imbalance when using (orange) one and (blue) eight sub-buckets.

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 to flat mode, and NUMA mode to "quad.". The default sub-bucket size of the input relation is 8 sub buckets per rank. In both testing we used a snapshot of Twitter from 2010 [23]. 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 aggregation 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:

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

The $MIN aggregate canonicalizes a component representative, efficiently compressing connected components in space. By contrast, 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

We compared the performance of two versions of the SSSP program in PARALAGG: the baseline version with no spatial load balancing or communication-avoiding join optimization, and the optimized version with balancing and communication-avoiding join optimization enabled. For both runs, we used the Twitter dataset on Theta. Our results are shown in Figure 2: generally, the optimized implementation's running time was cut in half compared to the baseline. The improvement was mainly in local join computation: at 512 cores, the optimized program's local join time was only 20% of the baseline program. This is because the *Spath* relation is much smaller than the *Edge* relation at the 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. It is worth noting that the "comm" in the figure, representing the communication time for distributing join results, remain the same in the optimized program. This phase was independent of our optimized join layout and implemented using MPI_Alltotallv. Therefore, our optimization, which specifically targets the join phase, does not accelerate the "comm" time.

In Figure 2, we observe scalability declines after roughly 2k processes. We investigated this issue by studying tuple distribution across ranks. The "original" curve in 3 shows that turning off sub-bucket load balancing in the baseline leads to data imbalance across ranks, with the largest rank had ten times more tuples than the smallest rank. This imbalance is mainly due to Twitter dataset's nature as a social network dataset, where some users have millions of followers, causing all edges with the same starting node to be stored on the same rank.

We next ran the CC query using the Twitter dataset to study whether PARALAGG's spatial load balancing can ameliorate the slowdowns we observed in Figure 2. The blue curve in Figure 3 shows that using 8 sub-buckets successfully mitigated the data imbalance, reducing the tuple size difference on ranks to around 2 times. In terms of scalability and local join computation time, we compared the performance of the 8 sub-bucket configuration with the baseline setting, as shown in Figure 4. Due to imbalance, the query with 1 sub-bucket reached its scalability limit after 2,048 processes. Beyond this point, adding more cores slowed down the query. However, the balanced program outperformed the imbalanced program significantly after 4,096 processes and continued to scale up to 16,384 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 1,024. This can be attributed to the additional intra-bucket communication re-

quired for spatial load balancing, as explained in Section IV-C. However, our work primarily targets supercomputers, where the impact of slower running times on relatively small numbers of cores is considered acceptable.

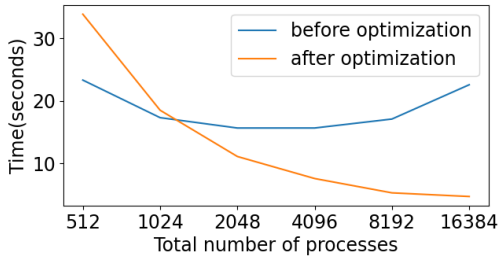

Fig. 4. Local join computation time for the CC query, with one (orange) and eight (blue) sub-buckets. Our results illustrate that sub-bucketing enables ever-increasing gains to local join; imbalance-imposed overhead halts scalability around 2k processes when one sub-bucket is used.

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 [24], each consisting of roughly 100 million edges. The last graph, Topcats [25], 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, 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

TABLE I
SINGLE-NODE EXPERIMENTS (M:SS) COMPARING PARALAGG, RASQL, AND SOCIALITE ON A 64-CORE (128-THREAD) SERVER.

	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
	Twitter	PARALAGG	N/A	8:13	**4:40**
		RaSQL	N/A	N/A	N/A
		SociaLite	N/A	N/A	N/A
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
	Twitter	PARALAGG	N/A	26:19	**15:24**
		RaSQL	N/A	N/A	N/A
		SociaLite	N/A	N/A	N/A

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. RaSQL and SociaLite failed on Twitter due to integer overflow issues, as does PARALAGG at 32 threads (balancing ensures PARALAGG works at 64 threads and beyond); we mark these with N/A in Table I.

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 supercom-

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.

		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

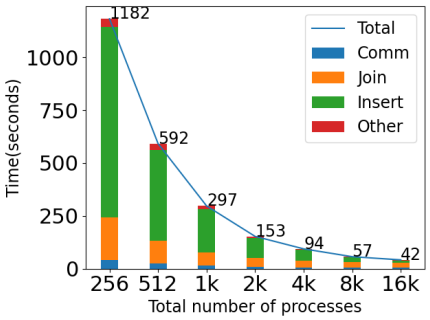

Fig. 5. Scaling SSSP query on Theta, using Twitter dataset.

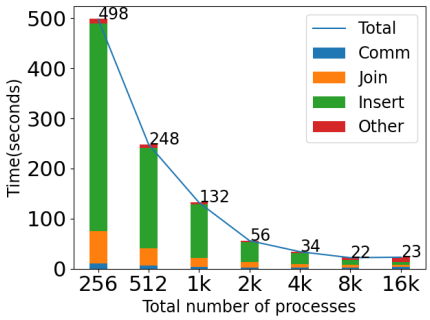

Fig. 6. Scaling CC query on Theta, using Twitter dataset.

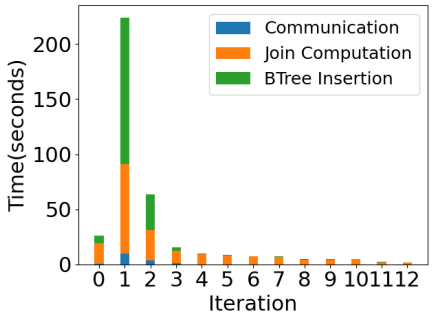

Fig. 7. Detail running times of each iteration for the SSSP on the Twitter Dataset (ran on Theta).

puter. In order to increase problem size, we run SSSP by running on 30 arbitrarily-picked start nodes simultaneously.

Before 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 5). 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 2,048 cores. While performance improvements slow down after 2,048 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 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 1,024 cores, as shown in Figure 7. 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.

Results for the CC query (Figure 6) were similar to those of the SSSP query, with a 96% decrease in running time from 256 to 16,384 cores. Near-perfect scalability was achieved until 2,048 cores, with 60% running time improvement from 2,048 to 8,192 cores. However, at 16,384 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], [20], [26]–[30]. Much of this work attempts to reconcile the semantic challenges when adding general lattices to Datalog [3], [20], [31], [32]. 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], [32]. By contrast, 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 gained popularity in optimized high-performance implementations due to its applications in large-scale program analysis [33]–[35], graph analytics [4], [36], [37], and related fields Initial efforts in scaling Datalog focused on novel representations (*e.g.*, binary decision diagrams [34]) to enable efficient joins or set operations (*e.g.*, leapfrog triejoin in LogicBlox [8]). 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 [7], [38]. 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) [39], and DCDatalog works to scale recursive aggregate queries while avoiding memory contention. Current efforts in distributed Datalog (*e.g.* DCDatalog and Cog [40]) turn away from Spark and MapReduce-style parallelism (increasingly understood to be a source of overhead [41], [42]); 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, most directly related to the recent work of Kumar et al. [13], [14], [43], [44], which excludes lattices and monotonic aggregates.

Dynamic Query Plan Optimization: Relational query planning has a long history, largely within the context of RDBMS systems [45]–[49], often powered by cardinality estimation methods (*e.g.*, the IK/KBZ family of algorithms [50]). Modern work in join plan compilation focuses on efficiently utilizing modern architectural elements, [51], parallelizing plan selection [52], and dynamic switching [53]. The communication-optimized join switching in PARALAGG is largely orthogonal to this work, inspired more closely by the notion of communication-avoiding algorithms [54].

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 [55]–[59], connected components [60], union-find [61]. 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], [62], 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.

VIII. Acknowledgement

This work was funded in part by NSF RII Track-4 award 2132013, NSF PPoSS planning award 2217036, NSF PPoSS large award 2316157 and, NSF collaborative research award

2221811. We are thankful to the ALCF's Director's Discretionary (DD) program for providing us with compute hours to run our experiments on the Theta supercomputer located at the Argonne National Laboratory. This material is based upon work supported by the Defense Advanced Research Projects Agency (DARPA) under Contract No. N66001-21-C-4023. Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of DARPA.

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.

[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] S. Ceri, G. Gottlob, and L. Tanca, "What you always wanted to know about datalog (and never dared to ask)," *IEEE Transactions on Knowledge and Data Engineering*, vol. 1, no. 1, pp. 146–166, 1989.

[7] 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.

[8] 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.

[9] I. S. Mumick and O. Shmueli, "How expressive is stratified aggregation?," *Annals of Mathematics and Artificial Intelligence*, vol. 15, pp. 407–435, 1995.

[10] E. W. Dijkstra, "A note on two problems in connexion with graphs," *Numerische Mathematik*, vol. 1, p. 269–271, 1959.

[11] 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.

[12] 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.

[13] 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.

[14] S. Kumar and T. Gilray, "Distributed relational algebra at scale," in *International Conference on High Performance Computing, Data, and Analytics (HiPC). IEEE*, vol. 1, 2019.

[15] J.-P. Cheiney and C. de Maindreville, "A parallel strategy for transitive closure usind double hash-based clustering.," in *VLDB*, pp. 347–358, 1990.

[16] K. Fan, T. Gilray, V. Pascucci, X. Huang, K. Micinski, and S. Kumar, "Optimizing the bruck algorithm for non-uniform all-to-all communication," in *Proceedings of the 31st International Symposium on High-Performance Parallel and Distributed Computing*, HPDC '22, (New York, NY, USA), p. 172–184, Association for Computing Machinery.

[17] N. Netterville, K. Fan, S. Kumar, and T. Gilray, "A visual guide to mpi all-to-all," in *2022 IEEE 29th International Conference on High Performance Computing, Data and Analytics Workshop (HiPCW)*, pp. 20–27, 2022.

[18] C. Zaniolo, M. Yang, A. Das, A. Shkapsky, T. Condie, and M. Interlandi, "Fixpoint semantics and optimization of recursive datalog programs with aggregates," *Theory and Practice of Logic Programming*, vol. 17, no. 5-6, pp. 1048–1065, 2017.

[19] 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.

[20] 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.

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

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

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] 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.

[37] 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.

[38] 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.

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

[40] 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.

[41] 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.

[42] 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.

[43] A. R. Shovon, T. Gilray, K. Micinski, and S. Kumar, "Towards iterative relational algebra on the {GPU}," in *2023 USENIX Annual Technical Conference (USENIX ATC 23)*, pp. 1009–1016, 2023.

[44] A. R. Shovon, L. R. Dyken, O. Green, T. Gilray, and S. Kumar, "Accelerating datalog applications with cudf," in *2022 IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, pp. 41–45, IEEE, 2022.

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

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

[47] 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.

[48] 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.

[49] 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.

[50] 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.

[51] 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.

[52] 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.

[53] 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, Proceedings*, pp. 83–102, Springer, 2022.

[54] 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.

[55] 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.

[56] 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.

[57] 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.

[58] 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.

[59] 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.

[60] 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.

[61] 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.

[62] 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.