Distributed Relational Algebra at Scale

Sidharth Kumar
Dept. of Computer Science
University of Alabama, Birmingham
sid14@uab.edu

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

***Abstract*—Relational algebra forms a basis of primitive operations suitable for applications in graphs and networks, program analysis, deductive databases, and constraint logic programming. Despite its expressive power, relational algebra has not received the same attention in high-performance-computing research as more common primitives like stencil computations, floating-point operations, numerical integration, and sparse linear algebra. Furthermore, specific challenges in addressing representation and communication among distributed portions of a relation, especially for inherently imbalanced relations, have previously thwarted successful scaling of relational algebra applications to supercomputers.**

In this paper, we present a set of efficient algorithms to effectively parallelize and scale key relational algebra primitives. We introduce a hybrid hash-tree approach to representing distributed imbalanced relations and permitting efficient communication. Finally, we demonstrate the scalability of our implementation with a fixed-point algorithm computing the transitive closure of a large graph (generating over 276 billion edges) on 32,768 processes.

***Index Terms*—Relation, Relational Algebra, Logic, Graph Algorithms, Transitive Closure, All to All Communication**

I. Introduction

Implementing application-specific code on supercomputers requires addressing the fundamental underlying primitives of an algorithm in a way that is flexible and scalable. Significant progress has been made on a wide variety of important problems due to a rigorous exploration of high-performance primitives such as stencil computations, floating-point arithmetic, numerical integration, and sparse linear algebra.

Relational algebra is a crucial primitive for a wide range of analytic problems in graphs, machine learning, logic programming, program analysis, deductive databases, and formal verification, that has been the subject of great interest in the literature [10], [11], [21], [22], [27], [29], but has had limited exploration on supercomputers, and at scale. Three central barriers to scaling operations on relations, such as union, selection, projection, and join, have been (a) how to represent distributed relations in a way that is amenable to efficient parallel operations, (b) how to handle communication to coordinate distinct portions of the distributed workload, and (c) how to handle inherent imbalance (key skew) in distributed relations, and dynamic changes in imbalance over time.

While some progress has been made in addressing these issues, (a) in particular [5], [8], [19], [25], no approach has yet provided a general framework that makes applications using a pipeline of *repeated* operations on relations—for fixed-point iteration, supporting applications such as Datalog and program analysis—possible at scale. For such applications to be implemented on distributed, many-core systems, existing algorithms (e.g., [5]) that distribute relations among available cores, perform a single operation, and return in map-reduce fashion, are not suitable as repeated operations require efficient granular communication at each step.

In this paper, we present a hybrid approach to representing relations on networked machines and performing efficient distributed operations on them, building on the current state of the art for single-node parallelism. Interestingly, in addressing the communication issue, we find that MPI's all-to-all communication paradigm suits relational algebra best. Today's supercomputers have specialized, high speed interconnects and data can be transmitted between processes with very low latency. When used with an appropriate configuration, all-to-all communication—known to be the most intensive mode of communication—can scale well.

A. Contributions

In particular, we make the following specific contributions to the literature:

1) We present a novel hybrid *hash-tree* based representation and its algorithms for distributed relational algebra.
2) We present a scalable implementation for a fixed-point algorithm employing distributed relational algebra: computing the *transitive closure* of a graph.
3) We present a *balanced hash-tree* based join algorithm to mitigate load-balance issues associated with inherently imbalanced (key-skewed) relations.
4) We demonstrate scalability of transitive closure up to 32,768 processes, producing a graph with more than 276 billion edges. To the best of our knowledge, this is the largest transitive closure operation discussed in the literature.

We understand our implementation to be the first truly scalable distributed relational algebra that addresses interprocess communication and load balancing, permitting fixed-point iteration, and laying the foundation for solving massive logical inference problems, graph problems, and more, on supercomputers.

II. Relational Algebra

This section reviews the standard relational operations union, product, intersection, natural join, selection, renaming,

2640-0316/19/$31.00 ©2019 IEEE
DOI 10.1109/HiPC.2019.00014

and projection, along with their use in implementing two closely related example applications: graph problems and bottom-up Datalog solvers.

A. Standard RA Operations

We make some standard assumptions about relational algebra that differ from those of traditional set operations. Specifically, we assume that all our relations are sets of flat (first-order) tuples of natural numbers with a fixed, homogeneous arity. This means that the relation $(\mathbb{N} \times \mathbb{N}) \times \mathbb{N}$ contains the tuple $(1, 2, 3)$, and not $((1, 2), 3)$. Further, in operations like union or intersection, both relations must be union-compatible by having the same arity and columns. Although our approach extends naturally to relations over arbitrary enumerable domains (such as integers, booleans, symbols/strings, lists of integers, etc), we also make the assumption that such values are interned and assigned an integer identity. Though many RA operations exist, several others are especially standard:

a) Cartesian product: The product of two relations R and S is defined:

$$R \times S \triangleq \{(r_0, \ldots, r_k, s_0, \ldots, s_j) \mid (r_0, \ldots, r_k) \in R \wedge (s_0, \ldots, s_j) \in S\}.$$

b) Union: The union of two relations R and R' may only be performed if both relations have the same arity but is otherwise set union:

$$R \cup R' \triangleq \{(r_0, \ldots, r_k) \mid (r_0, \ldots, r_k) \in R \vee (r_0, \ldots, r_k) \in R'\}.$$

c) Intersection: The intersection of two relations R and R' may only be performed if both have k arity but is otherwise set intersection:

$$R \cap R' \triangleq \{(r_0, \ldots, r_k) \mid (r_0, \ldots, r_k) \in R \wedge (r_0, \ldots, r_k) \in R'\}.$$

d) Projection: Projection is a unary operation that removes a column or columns from a relation—and thus any duplicate tuples that result from removing these columns. Projection of a relation R restricts R to a particular set of dimensions $\alpha_0, \ldots, \alpha_j$, where $\alpha_0 < \ldots < \alpha_j$, and is written $\Pi_{\alpha_0,\ldots,\alpha_j}(R)$. For each tuple, projection retains only stated columns: $\Pi_{\alpha_0,\ldots,\alpha_j}(R) \triangleq \{(r_{\alpha_0}, \ldots, r_{\alpha_j}) \mid (r_0, \ldots, r_k) \in R\}$.

e) Renaming: Renaming is a unary operation that renames (i.e., reorders) columns. Renaming columns can be defined in several different ways, including renaming all columns at once. We define our renaming operator, $\rho_{\alpha_i/\alpha_j}(R)$, to swap two columns, α_i and α_j where $\alpha_i < \alpha_j$—an operation that can be repeated to rename/reorder as many columns as desired: $\rho_{\alpha_i/\alpha_j}(R) \triangleq \{(\ldots, r_{\alpha_j}, \ldots, r_{\alpha_i}, \ldots) \mid (\ldots, r_{\alpha_i}, \ldots, r_{\alpha_j}, \ldots) \in R\}$.

f) Selection: Selection is a unary operation that restricts a relation to tuples where a particular column matches a particular value. As with renaming, a selection operator may alternatively be defined to allow multiple columns to be matched at once, or to allow inequality or other predicates to be used in matching tuples. In our formulation, selection on multiple columns can be accomplished by repeated selection on a single column at a time. Selecting just those tuples from relation R where column α_i matches a value v is defined: $\sigma_{\alpha_i=v}(R) \triangleq \{(r_{\alpha_0}, \ldots, r_{\alpha_k}) \in R \mid r_{\alpha_i} = v\}$.

Selecting just those tuples from relation R where the values in columns α_i and α_j must match is defined: $\sigma_{\alpha_i=\alpha_j}(R) \triangleq \{(r_{\alpha_0}, \ldots, r_{\alpha_k}) \in R \mid r_{\alpha_i} = r_{\alpha_j}\}$.

g) Natural Join: Two relations can also be *joined* into one on a subset of columns they have in common. Join combines two relations into one, where a subset of columns are required to have matching values, and generalizes both intersection and Cartesian product operations.

Consider an example of two tables in a database, one that encodes a system's users' `emails` (including their username, email address, and whether it's verified) and another that encodes successful `logins` (including a username, timestamp, and ip address):

`emails`

username	email	verified
samp	samwow@gmail.com	1
samp	samp9@uab.edu	0
lee	lee5@uab.edu	1

`logins`

username	time	ipaddr
samp	1554291414	162.103.150.12
lee	1554181337	171.31.15.120
lee	1554219962	155.28.11.102
lee	1554133720	171.31.15.120

A join operation on these two relations, written `emails` $\bowtie$ `logins`, yields a single relation with all five columns: username, email, verified, timestamp, address. For columns the two relations have in common, the natural join only considers pairs of tuples from the two input relations where the values for those columns match, as in an intersection operation; for other columns, the natural join computes all possible combinations of their values as in Cartesian product. If both input relations share all columns in common, a join is simply intersection and if both input relations share no columns in common, a join is simply Cartesian product. In this case, we have:

`emails` $\bowtie$ `logins`

username	email	verified	time	ipaddr
samp	samwow@...	1	...414	162...
samp	samp9@...	0	...414	162...
lee	lee5@...	1	...337	171...
lee	lee5@...	1	...962	155...
lee	lee5@...	1	...720	171...

For example, if we wanted to compute all email addresses and ip addresses that may be associated, we could compute the join of these two relations and then project the join down to these two attributes alone. Note that four (not five) rows result, as one becomes a duplicate after projection, in: $\Pi_{\texttt{email},\texttt{ipaddr}}(\texttt{emails} \bowtie \texttt{logins})$

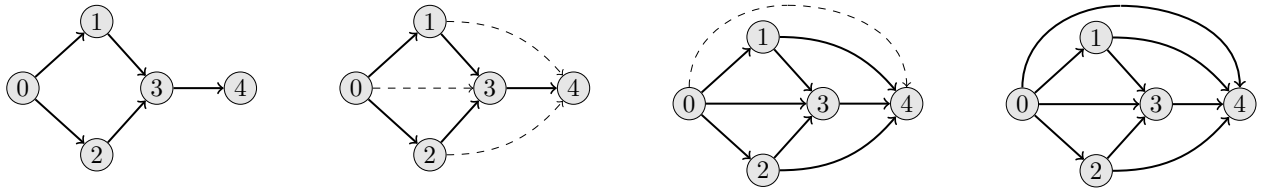


Fig. 1: Each iteration of computing transitive closure for a small example.

$\Pi_{\texttt{email,ipaddr}}(\texttt{emails} \bowtie \texttt{logins})$

email	ipaddr
samwow@gmail.com	162.103.150.12
samp9@uab.edu	162.103.150.12
lee5@uab.edu	171.31.15.120
lee5@uab.edu	155.28.11.102

In this example, we've shown relations with associated attribute (column) names (e.g., `email`, `ipaddr`). In our formalization of relations, we treat columns as ordered and identified by their index instead—naturally a programming model, RDBMS, or API for relations will associate these indices with their symbolic names. As formalized, the `emails` relation could be a set of tuples $R_{\texttt{emails}} = \{(0, 0, 1),\ (0, 1, 0),\ (1, 2, 1)\}$, where the attributes `username`, `email`, and `verified` are stored in columns 0, 1, and 2, respectively, the string "`samp`" is interned as username 0, the string "`lee`" is interned as username 1, and the three emails are interned as emails 0, 1, and 2. This is to say, although we treat relations exclusively as sets of tuples of integers in this paper, doing so is no meaningful restriction as databases and logic solvers add interning systems and schema to support other values and string-based names for columns.

To formalize natural join as an operation on such a relation, we parameterize it by the number of indices that must match, assumed to be the first j of each relation (if they are not, a renaming operation must come first). The join of relations R and S on the first j columns is written $R \bowtie_j S$ and defined:

$$R \bowtie_j S \triangleq \{\ (r_0, \ldots, r_k, s_j, \ldots, s_m) \mid (\ldots, r_k) \in R \wedge (\ldots, s_m) \in S \wedge \bigwedge_{i=0..j-1} r_i = s_i\ \}$$

B. Application: Transitive Closure

One of the simplest common algorithms that may be implemented efficiently as a loop over high-performance relational algebra primitives, is computing the *transitive closure* (TC) of a relation or graph. Consider a relation $G \subseteq \mathbb{N}^2$ encoding a graph where each point $(a, b) \in G$ encodes the existence of an edge from vertex a to vertex b.

For example, consider graph G (on the left in Figure 1) where $G = \{(0, 1), (1, 3), (0, 2), (2, 3), (3, 4)\}$. Renaming to swap the columns of G, results in a graph, $\rho_{0/1}(G)$, where all arrows are reversed in direction. If this graph is joined with G on only the first column (meaning G is joined on its second columns with G on its first column), via $\rho_{0/1}(G) \bowtie_1 G$, we get a set of triples (b, a, c)—specifically $\{(1, 0, 3), (2, 0, 3), (3, 1, 4), (3, 2, 4)\}$—representing paths of length two in the original graph where a leads to b which leads to c. Projecting out the first column with $\Pi_{1,2}(\ldots)$ yields pairs (a, c) encoding paths of length two from a to c in the original graph G. If we compute the union of this graph with the original G, we obtain a relation encoding paths of length one or two in G. This graph, $G \cup \Pi_{1,2}(\rho_{0/1}(G) \bowtie_1 G)$, is second in Figure 1 with new edges shown in dashes.

We can encapsulate this step in a function F_G which takes a relation T, encoding a graph, and returns the graph G unioned with T's edges extended with G's edges.

$$F_G(T) \triangleq G \cup \Pi_{1,2}(\rho_{0/1}(T) \bowtie_1 G)$$

The second graph shown can be produced by $F_G(G)$ and the graph G is returned if the input graph T is empty, as in $F_G(\bot)$. If F_G is repeatedly applied, each result encodes ever longer paths through G, as shown. In this case for example, the graph $F_G(F_G(G))$ or ${F_G}^3(\bot)$ encodes the transitive closure of G—all paths in G reified as edges. One final iteration is required to check that the process reached a fixed point.

In the general case, for any graph G, there exists some $n \in \mathbb{N}$ such that ${F_G}^n(\bot)$ encodes the transitive closure of G. The transitive closure may be computed by repeatedly applying F_G in a loop until reaching an n where ${F_G}^n(\bot) = {F_G}^{n-1}(\bot)$ in a process of *fixed-point iteration*. In the first iteration, paths of length one are computed; in the second, paths of length one or two are computed, and so forth. After the longest path in G is found, just one additional iteration is necessary as a fixed-point check to confirm that the final graph has stabilized.

C. Application: Datalog

Computing transitive closure is a simple example of logical inference. From paths of length zero (an empty graph) and the existence of edges in graph G, we may trivially deduce the existence of paths of length $0 \ldots 1$. From paths of length $0 \ldots n$ and the original edges in graph G, we may deduce the existence of paths of length $0 \ldots n{+}1$. The function F_G above performs a single round of inference, finding paths one edge longer than any found previously and exposing new deductions for the next iteration of F_G to make. When the computation reaches a fixed point, the solution has been found as no further paths may be deduced from the available facts.

In fact, the function F_G is a quite-immediate encoding in relational algebra of a variant of the transitivity property itself,

$$T(a,c) \Longleftarrow G(a,c) \vee T(a,b) \wedge G(b,c),$$

a logical constraint for which we desire a least solution. T satisfies this property *exactly* when T is a fixed-point for F_G.

Solving logical and constraint problems in this way is precisely the strategy of *bottom-up logic programming*. Bottom-up logic programming begins with a set of facts (such as $T(a,b)$—the existence of an edge in a graph T) and a set of inference rules and performs a least-fixed-point calculation, accumulating new facts that are immediately derivable, until reaching a minimal set of facts consistent with all rules.

Datalog is a bottom-up logic programming language supporting a restricted logic corresponding to first-order HornSAT—the satisfiability problem for conjunctions of Horn clauses [3]. A *Horn clause* is a disjunction of atoms where all but one is negated: $a_0 \vee \neg a_1 \vee \ldots \vee \neg a_j$. By DeMorgan's laws we may rewrite this as $a_0 \vee \neg(a_1 \wedge \ldots \wedge a_j)$ and note that this is an implication: $a_0 \leftarrow a_1 \wedge \ldots \wedge a_j$. In first-order logic, atoms are predicates with universally quantified variables.

A Datalog program is a set of rules $P(x_0,\ldots,x_k) \leftarrow Q(y_0,\ldots,y_j) \wedge \ldots \wedge S(z_0,\ldots,z_m)$ and its input is a database of facts called the *extensional database* (EDB). Running the datalog program reifies the *intensional database* (IDB) which extends facts from the EDB with all facts transitively derivable via the program's rules. In typical Datalog notation, computing transitive closure of a graph is accomplished with two rules:

```
T(x,y) <- G(x,y).
T(x,z) <- T(x,y), G(y,z).
```

The first rule says that any edge, in G, implies a path, in T (taking the role of the left operand of union in F_G or the left disjunct in our implication), and the second rule says that any path (x,y) and edge (y,z) imply a path (x,z) (adding edges for the right operand of union in F_G). Other kinds of graph mining problems, such as computing triangles, can also be naturally implemented as Datalog programs [26]. Our motivation for developing distributed RA is as a back-end for an expressive Datalog-like logic programming language.

D. Implementation approaches

In our previous discussion of both transitive closure and Datalog, we have elided important optimizations and implementation details in favor of focusing on the main ideas of both. In practice, it is inefficient to perform multiple granular RA operations separately to perform a selection, reorder columns, join relations, project out unneeded columns, reorder columns again, etc, when iteration overhead can be eliminated and cache coherence improved by performing loop fusion. In practice, high-performance Datalog solvers perform all necessary steps at once, supporting a generalization of the operations we have discussed that can join, select, reorder variables, project, and union, all at once. In a Datalog engine, this fusion may be accomplished automatically by applying *Futamura's projection* to a relational abstract machine semantics [21].

In addition, both transitive closure, and Datalog generally, as presented above, are using naïve fixed-point iteration, recomputing all previously discovered edges (resp. facts) at every iteration. Efficient implementations are *incrementalized*, only considering facts that can be extended to produce previously undiscovered facts. For example, when computing transitive closure, another relation T_Δ is used which only stores the longest paths—those discovered in the previous iteration. When computing paths of length n, in fixed-point iteration n, only new paths discovered in the previous iteration, paths of length $n-1$, need to be considered as shorter paths extended with edges from G yield paths which must have been discovered already. In Datalog and database theory, this optimization is known as *semi-naïve* evaluation [3].

Consider the second Datalog rule implementing transitive closure, discussed previously, defining paths in terms of paths and edges. Each iteration of our function F for this rule can be implemented as the following pseudocode:

```
new_T = {}
for [x,y] in delta_T.select_all():
  for [y,z] in total_G.select("y", y):
    if total_T.has_key([x,z]) == false:
      if delta_T.has_key([x,z]) == false:
        new_T.insert([x,z])
for [x,y] in delta_T.select_all():
  total_T.insert([x,y])
delta_T = new_T
```

III. Distributed Relational Algebra

This section discusses our implementation of distributed relational algebra. We first synthesize a small increment from the current state of the art: a hybrid approach we call *hash-tree* relational algebra. This consists of nesting B-trees within a hash-table that can be partitioned across multiple MPI processes. In a typical join, it will be necessary to iterate over all tuples where particular columns match a value or values taken from the first operand of join—in which case the variable being matched should come first and be a key in the outermost B-tree. The relation is thus explicitly indexed on this column.

In our implementation, a relation $R(a,b)$ indexed on join-variable a is encapsulated using a type `Rel<Rel<void>>` which provides an interface to a B-tree mapping `uint64_t` keys, storing each a, mapped to subrelations over just those values b that are paired with a particular a. In our distributed hash-tree approach, this nesting of B-trees is extended at the top-level by a distributed hash table so that each value a is also hashed to assign its tuple to one of $nproc$ (the number of MPI processes hosting the relation) *buckets* that are distributed across available MPI processes.

By contrast with the *double-hashing* approach of nesting hash tables within a distributed hash table, use of B-trees carries several advantages. They are faster at unstructured insertions overall and dynamically expand in non-amortized log-time while hash tables dynamically expand only in amortized

log-time and suffer slowdowns in practice at scale. In addition to being slower in isolation, the higher worst-case complexity of hash-tables leads to complications in a distributed setting where a single inner, per-bucket table (on one processes) resizes, delaying a synchronized communication phase for all processes. The state-of-the-art Datalog solver Soufflé also demonstrates the relative performance of tree-structures (using B-trees and prefix-trees) [21] and shows that tree structures permit indices to be shared optimally permitting a full order-of-magnitude speedup in a static program analysis task [23].

The standard way to parallelize the key-value store approach to relational algebra on multi-core systems is to partition the iteration space of the outermost loop. For example, the Soufflé uses OpenMP to parallelize its join operations, first partitioning the outermost key-value store into multiple disjoint iterators—one for each available thread. Soufflè's join algorithm is nearly identical to our previous pseudocode for join, except that it adds an outer parallel for loop.

A. Hash-tree relations

Our hybrid hash-tree representation for relations makes its parallelism explicit as physically separate partitions of the total relations are stored in distinct hash-table buckets—each owned by a single MPI process. Join operations then decompose into a separate join for each bucket, followed by hashing of output tuples, and a communication phase to insert these tuples into the output relation. In experiments designing our join operation, we found that, as hashing distributes keys evenly across buckets (although not necessarily tuples), MPI's all-to-all communication paradigm was actually most efficient for inserting output tuples in their receiving buckets (i.e., on processes hosting the output relation).

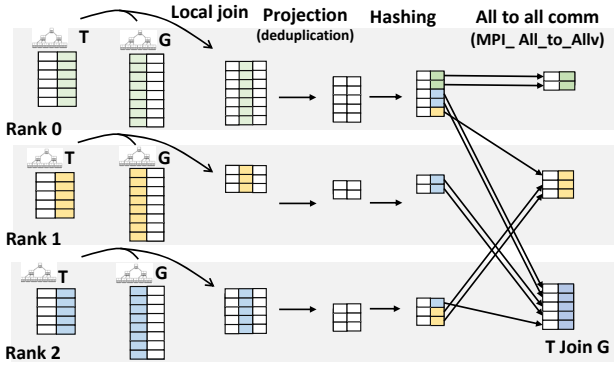


Fig. 2: Diagram showing different phases of hash-tree join.

In Figure 2 we show the process for computing a single iteration of a distributed TC computation. T (i.e., T_Δ as we implement incrementalized TC) is joined with G on a per-bucket basis (the diagram shows three buckets). Tuples in $T(x, y)$ are indexed on the second column (y) and tuples in $G(y, z)$ are indexed on the first column (y). This makes it possible to perform local (intra-bucket) joins as each tuple $(x, y) \in T$ is guaranteed to have all matching tuples $(y, z) \in G$ stored in the same bucket, managed by the same MPI process. Each resulting triple (x, y, z) has its middle column projected out on-the-fly as it is produced, and, as the resulting tuple (x, z) must be inserted into T, a relation indexed on its second column, each new edge is hashed and assigned to bucket $hash(x)$ modulo $nprocs$ in the output relation (T, T_Δ).

Our design requires distinct versions of a relation (e.g., T, T_Δ) are co-located on MPI processes via an identical bucket decomposition. In addition, this bucket for T and T_Δ will not generally be managed by the local MPI process, so a communication phase is required to actually perform insertion of each output tuple in its output relation. As each output tuple is produced, it is staged in one of $nprocs$ packets, ready to be sent across the network to the MPI process managing its bucket. Finally, an all-to-all communication phase is used to reorganize the output of the join operation—preparing T for subsequent fixed-point iterations. As tuples are received by their host process, they are inserted into the local B-tree structure, eliminating duplicates. This final insertion of tuples into an extant relation, without removing any existing tuples, performs the union required in our implementation of F_G from section II. As T and T_Δ are co-located across processes, no communication is needed to perform final insertion—if a tuple is in T it is not locally inserted into T_Δ, otherwise it is.

B. Balanced hash-tree relations

A crucial complication remains that, so far, has been only alluded to: although hashing on join columns distributes those keys uniformly across buckets, a single key can have many more tuples than another. This is to say, some relations are inherently highly imbalanced. Consider a graph with a million vertices, most with at-most one outgoing edge, but a single vertex has nearly a million outgoing edges of its own.

In our hash-tree approach as described, the vertex with many edges is mapped to a single *heavy* bucket that contains most of the relation. Each operation then runs as slow as the slowest bucket, which will be our heavy bucket—if it is even able to store so much of the overall relation in the first place. To distribute such a relation effectively, it is necessary to disproportionately allocate compute resources.

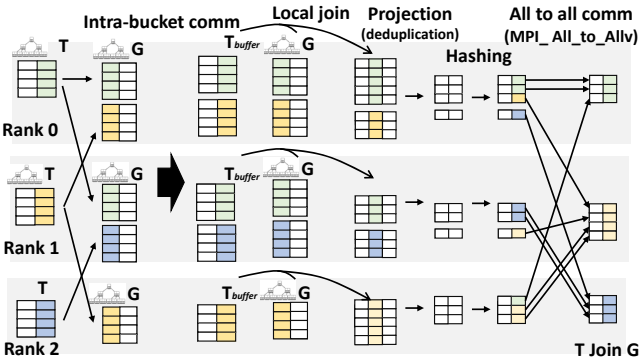


Fig. 3: Diagram showing phases of balanced-hash-tree join.

To address this issue, we develop a *balanced-hash-tree* approach: we partition each relation into buckets, and each

bucket into *subbuckets*, assigning a tuple to its bucket by hashing its key (set of join columns) and to its subbucket by hashing over the full tuple so there is uniform distribution of tuples in a bucket to subbuckets. Then, as the number of subbuckets per bucket may be heterogenous (this is the whole point of introducing subbuckets), we require a strategy for mapping subbuckets onto processes.

Figure 3 shows the phases of a balanced-hash-tree join. First, an intra-bucket communication phase is required. The example shows T balanced with three buckets, and one subbucket per bucket, and G balanced with 2, 2, and 1 subbuckets per bucket respectively. No longer is there an immediate 1-1 mapping between buckets and MPI ranks; each subbucket is mapped to some process (in this diagram, using a round-robin approach). The green-keyed bucket of T transmits its tuples to both subbuckets (hosted on ranks 1 and 2) as either could contain matching tuples in G. The yellow-keyed bucket of T transmits its tuples to both yellow subbuckets of G (hosted on ranks 3 and 1). The purple-keyed bucket of T transmits its tuples to the one purple subbucket of G (hosted on rank 2).

In an (imbalanced) hash-tree join, newly inserted tuples from the last join arrive in buckets for T_Δ, keyed on their second column, and are then immediately available on that same process this iteration to be joined with tuples in G (keyed on their first column). Now, however, the subbucket for T_Δ's new tuple, say $(14, 6)$, may *not* be the subbucket containing a tuple in G, say $(6, 11)$, which should join with it. This means all subbuckets for T_Δ must first transmit all their tuples to *all* subbuckets in the same bucket to be joined with tuples in G. As this is always a delta relation—it is not being strictly accumulated and these copies last only the current iteration.

Next, the join continues as before, except in terms of subbucket partitioning instead of bucket partitioning. Each subbucket computes a local join, hashes the resulting tuples to determine the bucket and subbucket it belongs to, identifies the process owning this subbucket, and transmits the tuple during the global communication phase.

1) Mapping subbuckets to processes: We implement two strategies for mapping subbuckets to processes, a *round-robin* approach and a *hashing* approach. The round robin approach uses communication to synchronize a map maintained on every process from bucket and subbucket to a rank—doing this we are able to evenly distribute subbuckets to processes. The hashing approach instead attempts to eliminate this overhead by also hashing the bucket and subbucket indexes to map them to a process—raising some possibility that with fewer subbuckets, the balancing will be less precise.

2) Dynamic Adaptive Refinement: It is also crucial to consider that in real-world applications, especially those based on fixed-point interation, the inherent balance of a relation may vary arbitrarily across time (becoming more or less balanced). For this reason, it is important that the balancing a relation's representation be permitted to efficiently vary across iterations.

We implement a bucket refinement procedure that can be triggered by periodic checks to see which buckets, if any, have become sufficiently imbalanced. Bucket refinement splits a bucket into $4\times$ as many smaller subbuckets, introducing a special nonblocking point-to-point communication phase for redistributing three-quarters of each bucket's tuples to new subbuckets.

Name	ID	Edges	Union	Join	TC Edges
cz40948	2567	412148	✓		1676697757
mc2depi	2377	2100225	✓		276491930625
delaunay_n21	2476	6291408	✓		308759592
Hardesty3	2833	40451631	✓		–
circuit5M	2276	59062957	✓		11687744437
mawi_201512020130	2803	136024430	✓		178113958
kmer_A2a	2805	180292586	✓		136525288391
union of all	–	424592810		✓	–

TABLE I: List of seven graphs used in our evaluation. Also listed is the number of edges in the transitive closure of each.

IV. Evaluation

As all our RA operations involve an all-to-all communication phase, we start by performing a benchmark of MPI's all-to-all (`MPI_Alltoallv`) communication capability in isolation. We then benchmark the scaling properties of our distributed *hash-tree* union, join, and transitive closure operations over a range of large graphs from the literature. Finally we perform a study of our *balanced hash-tree* join algorithm.

A. Dataset and HPC platforms

We performed our experiments using the SuiteSparse Matrix Collection [9]. Formerly known as the University of Florida Sparse Matrix Collection, this dataset is a large and actively maintained resource for sparse matrices that arise in real applications. The collection is widely used by the numerical linear algebra community for the development and performance evaluation of sparse-matrix algorithms. For our experiments, we use seven real-world graphs (listed in Table 1) representing a wide range of sizes, two random graphs, and six synthetic graphs to test extreme topologies (discussed below). The transitive closure of a graph with n edges can contain up to n^2 edges (a fully connected graph). The number of edges in the transitive closure of a graph depends on the connectedness and topology of the input graph.

The experiments presented in this work were performed on the Theta Supercomputer [2] at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray machine with a peak performance of 11.69 petaflops, $281{,}088$ compute cores, 843.264 TiB of DDR4 RAM, 70.272 TiB of MCDRAM and 10 PiB of online disk storage. The supercomputer has Dragonfly network topology and a Lustre filesystem.

B. MPI_Alltoallv

All-to-all communication is central to our distributed RA algorithms (section III), as each uses MPI's `MPI_Alltoallv` function to facilitate data communication. `MPI_Alltoallv` transmits data between all pairs of processes where each process can send a variable amount of data by providing offsets into a buffer. In this section we study both weak and strong scaling characteristics of `MPI_Alltoallv`. For

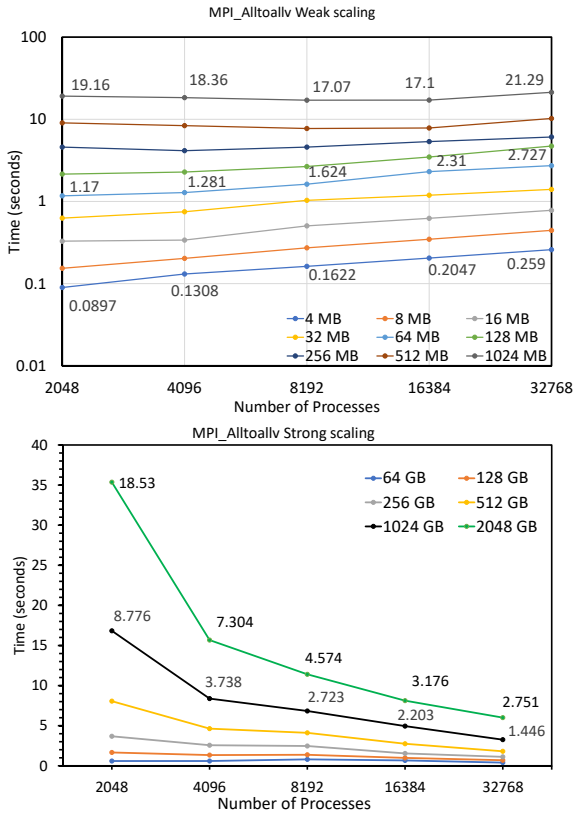


Fig. 4: Weak (top) and strong (bottom) scaling evaluation of `MPI_Alltoallv` function of MPI.

both sets of experiments, we varied the number of processes from 2,048 to 32,768. We performed 9 sets of weak scaling experiments where, in each, the amount of data transmitted by each process ($data_{proc}$) was varied from 4 megabytes to 1024 megabytes. For an n-process run, every process transmits $data_{proc}/n$ units of data to every other process. For strong scaling experiments, we performed 6 sets of experiments, varying the total amount of data generated across processes ($data_{total}$) from 64 gigabytes to 2,048 gigabytes. The amount of data generated by each process is the same; for example, for an n-process run, transmitting $data_{total}$ units of data, each process produces $data_{total}/n$ units of data and transmits $data_{total}/n^2$ units of data to every other process. The results of both scaling experiments can be seen in Figure 4.

For both strong and weak scaling runs, we observe a decline in performance with overall decreasing workload. For instance, with strong scaling, when total workload is 2,048 gigabytes, we observe near perfect scaling when the number of processes is doubled from 2,048 (18.5 seconds) to 4,096 (7.3 seconds) to 8,192 (4.5 seconds). After 8,192 processes, although total time continues to come down with increasing process count, we see the rate of improvement drop off. When total workload is only 64 gigabytes, we observe relatively poor scaling characteristics across the entire process range. Both these observations may be attributed to an overall reduction in per-process workload: with less data to transmit, total time is dominated by initialization costs. For weak scaling experiments we observe a similar trend: near perfect scaling for substantial data exchange that drops off as load decreases.

In the context of communication requirements for distributed RA operations, we find the scaling trends of `MPI_Alltoallv` to be encouraging. In general, for a given workload (i.e., overall tuple count for RA operations), there will be a range of processes that exhibits good all-to-all scaling characteristics. What remains is the challenge of identifying the ideal process count to balance the trade-off between computation and communication. As we observe in section IV-C, below, with larger per-process workload, computation cost dominates, as opposed to smaller per-process workload where total cost is dominated by communication.

C. Hash-tree Union and Join

We examine strong scaling to benchmark the performance of our distributed hash-tree union. We measure the time to union the 7 graphs listed in Table 1, not assuming tuples begin on their appropriate host process (hence the need for a single all-to-all communication phase). The number of processes are varied from 64 to 16,384. The total number of edges across all 7 graphs is 664,659,334 (9.9 gigabytes of data). The union of all 7 graphs has 424,592,810 edges, indicating significant overlap among the graphs. Results are plotted in the top of Figure 5, showing separate timings for in-memory deduplication, communication, and final insertion. We observe that the isolated union operation only scales well to 1024 cores, which can be attributed to an increase in communication time at higher core counts associated with movement of many small-sized data packets. This result corroborates the trend seen in section IV-B. At 2,048 and 4,096 processes, though insertion time is reduced, the per-process workload becomes small, impeding scalability of the communication phase.

We also examine strong scaling to benchmark the performance of our distributed join. We perform a join between the union of all graphs computed above and itself, to compute paths of length 2. This join operation yields a graph with 981,818,925 edges. The number of processes are varied from 64 to 16,384. We observe 1.8 billion output tuples generated per-second at $16k$ cores (note this includes three phase, local join, all to all comm and subsequent inserts in a nested b-tree). The pure insertion rate at the same core count is 14.83 billion keys per-second, peak aggregate throughput. Once both relations (i.e., the graph indexed on column 1 and on column 0) are initialized across all processes, we initiate the join operation. We plot the scaling results for join in the bottom of Figure 5. Unlike unions, distributed join demonstrates near perfect scaling to 8,192 processes. The trend can be attributed to the fact that there is a greater overall workload for join—enabling performance at higher process counts—and that the all-to-all communication phase increasingly dominates at these higher process counts and itself does not scale perfectly.

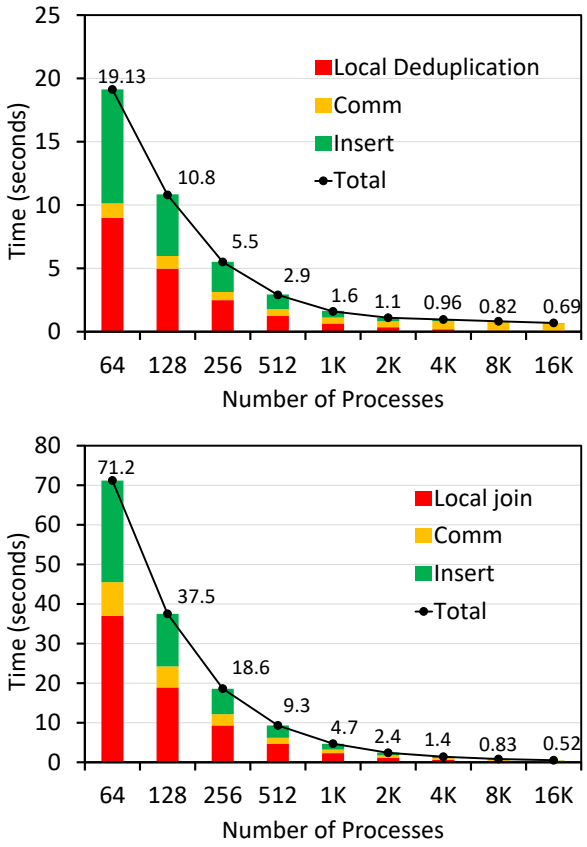


Fig. 5: Strong scaling for hash-tree union (top) and join.

Scaling thus diminishes after $8k$ processes, as the ratio of overall communication to per-process computation increases.

1) Transitive closure: The transitive closure (T) of an input graph (G) is iteratively extended by adding new paths discovered by a join operation until a fixed point is reached, and no new paths can be added to T. We performed strong scaling analysis for a graph with edge count 2,100,225 (mc2depi), varying the number of processes from $4,096$ to 32,768. The graph attained its fixed point after 2,956 iterations, generating a total of 276,491,930,625 edges (4 terabytes). As can be seen in Figure 6, our approach takes $1,114$ seconds at 32,768 cores to compute the transitive closure. To the best of our knowledge, this is the first implementation that has successfully computed the transitive closure of such a large graph. The time taken per iteration is roughly fixed at around 0.37 seconds per iteration (at 32,768 cores). This is mainly because of the ring-like topology of the graph; such graphs progress from $O(n)$ to $O(n^2)$ edges, adding a constant $O(n)$ number of edges at each iteration. The aggregate all-to-all communication time goes from 50% of the total time at 4,096 processes to 82% of total time at 32,768 processes. As reflected in our study on join in isolation, we do not observe ideal scaling beyond about 8,192 processes due to increasing all-to-all communication costs at higher process counts.

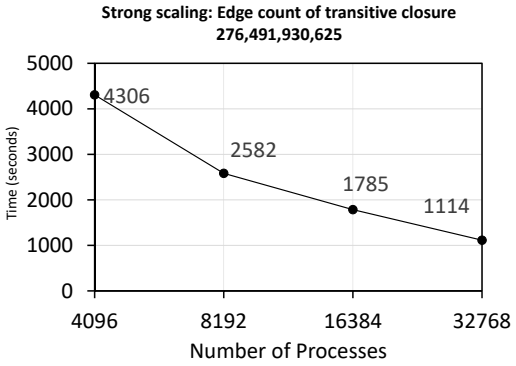


Fig. 6: Transitive closure of graph with edge count 2,100,225.

D. Balanced hash tree joins

In this section, we study two different aspects of the balanced-hash-tree join algorithm. First, we study the impact of two different subbucket-to-process mapping schemes: round-robin and hashing. Second, we study how the algorithm behaves with varying inherent imbalance (key skew).

1) Sub-bucket to process mapping: round robin vs hashing: We use the transitive-closure computation in a strong scaling setting to study the impact of our two mapping strategies. We use a graph with 412,148 (cz40948) edges, and vary the number of processes from 128 to 4,096. The graph attained its fixed point after 2,933 iterations, generating a total of 1,676,697,415 edges (25 gigabytes). We performed all our experiments using both the round-robin and the hashing scheme. The results are plotted in Figure 7. We observe that hashing is consistently slower than the round-robin scheme at all core counts. Further, looking at the subbucket-to-process distribution, we find, for all runs, around 35% of the processes remain idle without any sub-bucket to work on—hashing does a poor job in distributing the sub-buckets uniformly across the processes. The reason hashing is ineffective here, compared to hashing on keys, is because the number and range of subbuckets is small compared to the range values of the keys.

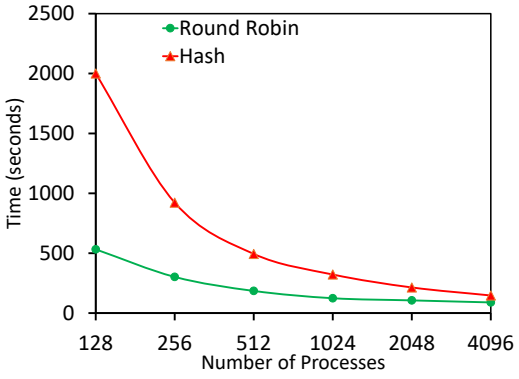


Fig. 7: Round robin vs. hashing

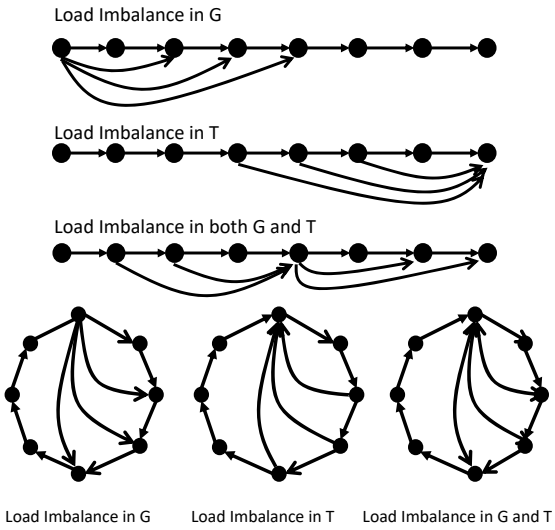


Fig. 8: The top three graphs show imbalances for the string topology; the three graphs in the last row show imbalance in the ring topology.

2) Sensitivity study: In this section, we evaluate the performance of our approach for inherently imbalanced relations. We first synthesize varying degrees of skew in two topologically distinct extreme cases. We use a *ring*-topology graph that leads to a fully connected (therefore balanced) transitive closure graph, and we use a *string*-topology graph whose transitive closure is smoothly imbalanced with each vertex connected to all downstream vertices.

We add varying degrees of initial skewness to these graphs by adding extra edges that either share one source node, one target node, or one of each. While computing the transitive closure, the first scenario leads to an imbalance in graph G, the second case corresponds to imbalance in graph T and the third case causes imbalance in both. See Figure 8 for examples. We vary the degree of skewness by adding a proportional number of edges. We conduct all our experiments at a core-count of 256 using a graph with $20,000$ nodes and varying the load imbalance factor by $20\%, 40\%, 60\%,$ and 80%. In our experiments, we perform just one load-balancing operation after the first iteration.

We observe that the balanced-hash-tree algorithm either outperforms, or is at-worst equal, compared to the hash-tree algorithm in each case. For graphs with imbalance in both G and T (first column), we observe similar trends for both string and ring topologies; balanced hash-tree join outperforms the hash-tree joins for all load imbalance factors. For ring-topology graphs, absolute timings are very-roughly double, due to the final TC having double the edges of the string-topology's TC. For the ring-topology graphs, the final TC is perfectly balanced, so when it is imbalanced in G, a single balancing operation suits the computation as G is never updated; when it is imbalanced in T, subbucket consolidation is required and higher degrees of skew continue to perform poorly. For the string-topology graphs, the final TC is smoothly imbalanced, so when it is imbalanced in G, initial balancing has little impact; when it is imbalanced in T, initial balancing at least partially supports the necessary final balancing of T. Overall, we find that bucket refinement helps reduce both the computation and communication overhead.

We also compared TC of two random graphs, generated via RMAT [17], at 256 processes, both with load balancing versus without. Both had $100{,}000$ vertices and $200{,}000$ edges. The first was generated with parameters $a = 0.6$, $b = 0.2$, $c = 0.2$, and had a maximum degree of 2636, average degree of 9.1, and a transitive closure with $471{,}606{,}725$ edges. This graph took 115 seconds without load balancing and 98 seconds with (a 14.8% improvement). The second was generated with parameters $a = 0.7$, $b = 0.15$, $c = 0.15$, and had a maximum degree of 2852, average degree of 13.5, and a transitive closure with $215{,}517{,}294$ edges. This graph took 92 seconds without load balancing and 78 seconds with (a 15.2% improvement).

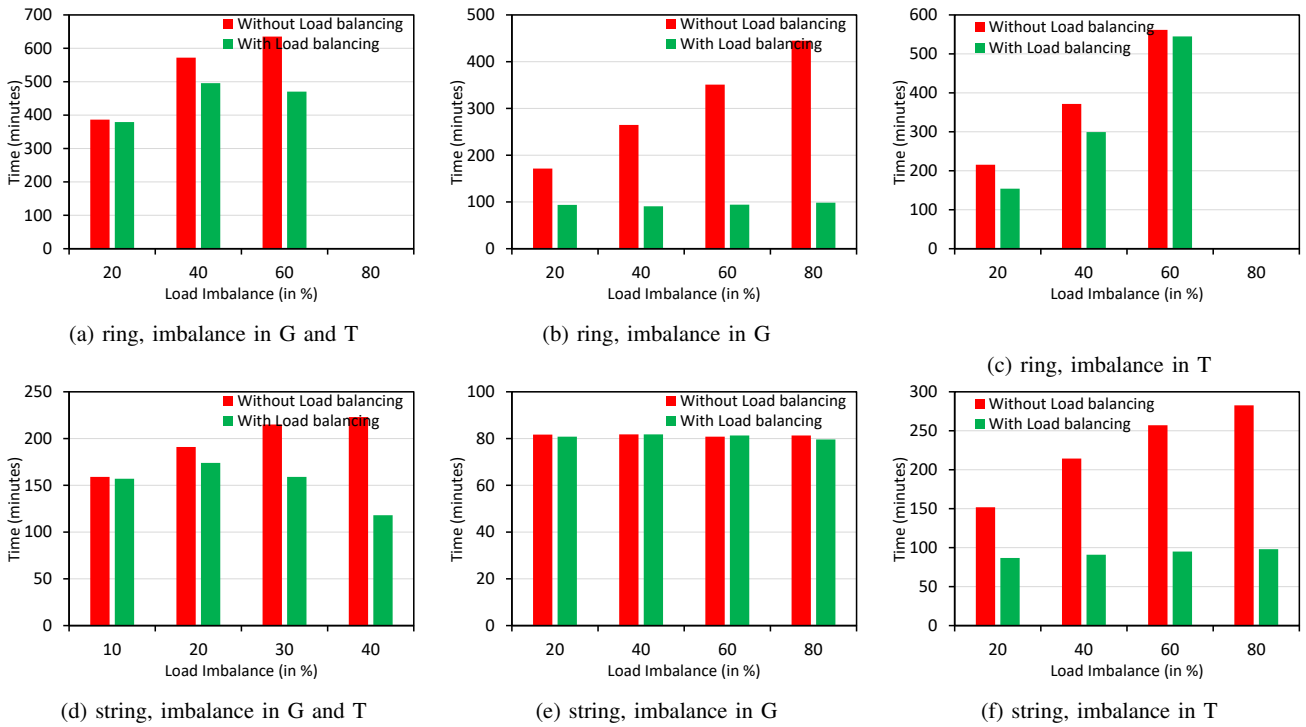
As part of the future work, we plan to develop an adaptive load balancing scheme that is robust both to arbitrary graph topologies and to arbitrary *changes in topology* across time. It is important such a system can adaptively both refine and merge subbuckets based on the relative sizes of buckets in a cost efficient manner.

V. Related Work

Much work has gone into accelerating single-node join performance [4], [12], [13], [15], however we tried to duplicate the TC computation that we used for our strong-scaling study using the state-of-the-art Datalog solver Soufflé [21], on a single 28-thread node, and it did not complete within 10 hours.

The double-hashing approach with local hash-based joins and hash-based distribution of relations is the most commonly used method to massively parallelize join operations. This algorithm involves partitioning the input data so that they can be efficiently distributed to the participating processes. This early foundational approach was first introduced in [25]. Much work such as [8] and [7] has since built on top of this approach. Our approach does as well, but differs from this method in several respects. First, we take lessons from state-of-the-art approaches to parallelism on single-node machines and use a nested tree-based structure to encode relations [12], [13]. In addition, we are the first to address efficient communication and the use of MPI's all-to-all communication paradigm to permit fixed-point iterations. Finally, ours is the first demonstration of RA load balancing and an end-to-end relational algebra application at HPC scale.

Recently, with [5], there has been a concerted effort to implement distributed join operations on clusters using MPI. The commonly used radix-hash join and merge-sort join have been re-designed for this purpose. Both these algorithms partition data so that they may be efficiently distributed to participating processes and are designed so that inter-process communication is minimized. In both of these implementations one-sided RMA operations are used to remotely coordinate



(a) ring, imbalance in G and T

(b) ring, imbalance in G

(c) ring, imbalance in T

(d) string, imbalance in G and T

(e) string, imbalance in G

(f) string, imbalance in T

Fig. 9: Sensitivity study for ring and string topology

distributed joins and to overlap communication and computation. This implementation involved scaling to 4,096 nodes, and reached extremely high peak tuples/second throughput, but did not specifically address the communication challenges required to implement fixed-point algorithms over RA, and only considered uniform (perfectly balanced) relations—citing replication-based balancing as future work. The experiments used to achieve peak throughput are also not representative of realistic workloads as each key is reported to have exactly one matching tuple in each relation being joined. Crucially, the work does not consider inter-node transmission of materialized joins and their remote staging for another join operation—which is central to applications such a transitive closure and distributed logical inference. This paper does integrate acceleration of local joins via AVX/SIMD instructions, as in [4], [15], pay careful attention to cache behavior, and to data compression. We plan to explore such orthogonal improvements to our per-process join as future work, but have focused presently on the issues of communication and load-balancing that are central to distributed RA applications.

There has been some work in the past to scale RA operations on GPUs. For example, Redfox is a single-GPU implementation of RA primitives [28]. While RedFox takes an interesting approach to RA on the GPU, it does not address crucial aspects of the task such as deduplication of tuples generated by joins and unions. Other work such as [18] and [30] have also explored the use of GPUs to scale dedicated RA tasks like the triangle listing problem. GPU-based hash tables have also been extended to multi-GPU algorithms that distribute relations over multiple accelerators [14].

Our implementation heavily relies on all-to-all communication. We use the `MPI_Alltoallv` function to transmit data from every process to every other process. Our use is related to distributed hash tables more generally [19], which make effective use of all-to-all communication, except that we co-locate multiple distributed hash tables for the purposes of performing efficient joins. `MPI_Alltoallv` is one of the most communication-intensive collective operation used across parallel applications such as CPMD [1], NAMD [20], LU factorization, fast fourier transform (FFT) and matrix transpose. Much research [16], [6], [24] has gone into developing scalable implementations of collective operations; most of the existing HPC platforms therefore have a scalable implementation of all-to-all operations.

VI. Conclusion & Future Work

We have presented the first general algorithms for scalable relational algebra on supercomputers that account for load balancing and fixed-point iteration. Our approach addresses both representation and communication among portions of a distributed relation, along with balancing inherently imbalanced relations, laying the groundwork for scaling algorithms that require a pipeline of repeated operations on relations, or fixed-point iteration, such as logical and constraint problems, deductive databases, and static program analyses.

Our existing system is not without limitations. We have not completely solved the problem of adaptive load-balancing.

In our existing solution we are successfully able to reduce imbalance by splitting, however, we do not have a scheme for merging light-weight sub-buckets—a necessary capability as illustrated by Figure 9c. Moving forward, we require a solution that is fully robust to changes across time. We also observe that some iterations can overflow memory by producing too many output tuples in a single iteration. To address this we are implementing a thresholding technique that allows partial joins to roll-over from one iteration to the next; we hypothesize this will allow us to compute the transitive closure of SuiteSparse graph `Hardesty3`. Finally, we plan to implement a distributed interns table so we may support primitive operations on base values, such as strings, in the context of our (Datalog-like) logic-programming language.

VII. Acknowledgment

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. This work used resources of the Argonne Leadership Computing Facility, which is a U.S. Department of Energy Office of Science User Facility, supported under Contract DE-AC02-06CH11357.

References

[1] CPMD home page. http://www.cpmd.org/.

[2] Theta alcf home page. https://www.alcf.anl.gov/theta.

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

[4] C. Balkesen, G. Alonso, J. Teubner, and M. T. Özsu. Multi-core, main-memory joins: Sort vs. hash revisited. *Proceedings of the VLDB Endowment*, 7(1):85–96, 2013.

[5] C. Barthels, I. Müller, T. Schneider, G. Alonso, and T. Hoefler. Distributed join algorithms on thousands of cores. *Proc. VLDB Endow.*, 10(5):517–528, Jan. 2017.

[6] J. Bruck, , S. Kipnis, E. Upfal, and D. Weathersby. Efficient algorithms for all-to-all communications in multiport message-passing systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(11):1143–1156, Nov 1997.

[7] F. Cacace, S. Ceri, and M. A. W. Houstma. An overview of parallel strategies for transitive closure on algebraic machines. In *Proceedings of the PRISMA Workshop on Parallel Database Systems*, pages 44–62, New York, NY, USA, 1991. Springer-Verlag New York, Inc.

[8] J.-P. Cheiney and C. de Maindreville. A parallel strategy for transitive closure using double hash-based clustering. In *Proceedings of the Sixteenth International Conference on Very Large Databases*, pages 347–358, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[9] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, Dec. 2011.

[10] T. Gilray and S. Kumar. Toward parallel cfa with datalog, mpi, and cuda. In *Scheme and Functional Programming Workshop*, 2017.

[11] H. Jordan, B. Scholz, and P. Subotić. Soufflé: On synthesis of program analyzers. In S. Chaudhuri and A. Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[12] 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, pages 31–40, New York, NY, USA, 2019. ACM.

[13] H. Jordan, P. Subotić, D. Zhao, and B. Scholz. A specialized b-tree for concurrent datalog evaluation. In *Proceedings of the 24th Symposium on Principles and Practice of Parallel Programming*, PPoPP '19, pages 327–339, New York, NY, USA, 2019. ACM.

[14] D. Jünger, C. Hundt, and B. Schmidt. Warpdrive: Massively parallel hashing on multi-gpu nodes. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 441–450. IEEE, 2018.

[15] C. Kim, T. Kaldewey, V. W. Lee, E. Sedlar, A. D. Nguyen, N. Satish, J. Chhugani, A. Di Blas, and P. Dubey. Sort vs. hash revisited: fast join implementation on modern multi-core cpus. *Proceedings of the VLDB Endowment*, 2(2):1378–1389, 2009.

[16] R. Kumar, A. Mamidala, and D. K. Panda. Scaling alltoall collective on multi-core systems. In *2008 IEEE International Symposium on Parallel and Distributed Processing*, pages 1–8, April 2008.

[17] J. Leskovec, D. Chakrabarti, J. Kleinberg, and C. Faloutsos. Realistic, mathematically tractable graph generation and evolution, using kronecker multiplication. In *European conference on principles of data mining and knowledge discovery*, pages 133–145. Springer, 2005.

[18] C. A. Martínez-Angeles, H. Wu, I. Dutra, V. S. Costa, and J. Buenabad-Chávez. Relational learning with gpus: Accelerating rule coverage. *Int. J. Parallel Program.*, 44(3):663–685, June 2016.

[19] T. C. Pan, S. Misra, and S. Aluru. Optimizing high performance distributed memory parallel hash tables for dna k-mer counting. In *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 135–147. IEEE, 2018.

[20] J. C. Phillips, , S. Kumar, and L. V. Kale. Namd: Biomolecular simulation on thousands of processors. In *SC '02: Proceedings of the 2002 ACM/IEEE Conference on Supercomputing*, pages 36–36, Nov 2002.

[21] B. Scholz, H. Jordan, P. Subotić, and T. Westmann. On fast large-scale program analysis in datalog. In *Proceedings of the 25th International Conference on Compiler Construction*, CC 2016, pages 196–206, New York, NY, USA, 2016. ACM.

[22] Y. Smaragdakis and M. Bravenboer. Using datalog for fast and easy program analysis. In *Proceedings of the First International Conference on Datalog Reloaded*, Datalog'10, pages 245–251, Berlin, Heidelberg, 2011. Springer-Verlag.

[23] P. Subotić, H. Jordan, L. Chang, A. Fekete, and B. Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, Oct. 2018.

[24] R. Thakur, R. Rabenseifner, and W. Gropp. Optimization of collective communication operations in mpich. *Int. J. High Perform. Comput. Appl.*, 19(1):49–66, Feb. 2005.

[25] P. Valduriez and S. Khoshafian. Parallel evaluation of the transitive closure of a database relation. *Int. J. Parallel Program.*, 17(1):19–42, Feb. 1988.

[26] K. Wang, Z. Zuo, J. Thorpe, T. Q. Nguyen, and G. H. Xu. Rstream: marrying relational algebra with streaming for efficient graph mining on a single machine. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 763–782, 2018.

[27] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. *SIGPLAN Not.*, 39(6):131–144, June 2004.

[28] H. Wu, G. Diamos, T. Sheard, M. Aref, S. Baxter, M. Garland, and S. Yalamanchili. Red fox: An execution environment for relational query processing on gpus. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, pages 44:44–44:54, New York, NY, USA, 2014. ACM.

[29] X. Zhang, R. Mangal, R. Grigore, M. Naik, and H. Yang. On abstraction refinement for program analyses in datalog. *ACM SIGPLAN Notices*, 49(6):239–248, 2014.

[30] D. Zinn, H. Wu, J. Wang, M. Aref, and S. Yalamanchili. General-purpose join algorithms for large graph triangle listing on heterogeneous systems. In *Proceedings of the 9th Annual Workshop on General Purpose Processing Using Graphics Processing Unit*, GPGPU '16, pages 12–21, New York, NY, USA, 2016. ACM.