

Relational Algebra at Scale

Anonymous Author(s)*

ABSTRACT

Relational algebra forms a basis of primitive operations useful for applications in graphs, networks, program analysis, deductive databases, and logic. Despite its expressive power, relational algebra has not received the same attention in high-performance computing research as linear algebra, X, Y, or Z.

In this paper we present a set of efficient algorithms that tackle the problem of distributed, parallel relational algebra and use experiments from applications in graphs, program analysis, and datalog to evaluate our approach.

CCS CONCEPTS

• **Computer systems organization** → **Embedded systems**; *Redundancy*; Robotics; • **Networks** → Network reliability.

KEYWORDS

datasets, neural networks, gaze detection, text tagging

ACM Reference Format:

Anonymous Author(s). 2018. Relational Algebra at Scale. In *Woodstock '18: ACM Symposium on Neural Gaze Detection, June 03–05, 2018, Woodstock, NY*. ACM, New York, NY, USA, 7 pages. <https://doi.org/10.1145/1122445.1122456>

1 INTRODUCTION

Lorem ipsum dolar sit amat

2 RELATIONAL ALGEBRA

Relational algebra (RA) provides a basis of operations on relations (i.e., predicates, or sets of tuples) sufficient to implement a broad range of algorithms for databases and queries, data analysis, machine learning, graph problems, and constraint logic problems []. Scaling these underlying primitives, and finding an effective strategy for parallel communication to distribute them across multiple nodes, is thus a avenue for scaling and distributing algorithms for high-performance program analyses, deductive databases, among other applications. This section reviews the standard relational operations union, product, intersection, natural join, selection, renaming, and projection, along with their use in implementing two closely related example applications: graph problems and bottom-up datalog solvers.

The Cartesian product of two finite enumerations D_0 and D_1 is defined $D_0 \times D_1 = \{(d_0, d_1) \mid \forall d_0 \in D_0, d_1 \in D_1\}$. A *relation* $R \subseteq D_0 \times D_1$ is some subset of this product that defines a set of

associated pairs of elements drawn from the two domains. For example, if R were the relation (\geq) over natural numbers, both domains D_0 and D_1 would be \mathbb{N} and the relation could be defined $(\geq) = \{(n_0, n_1) \mid n_0, n_1 \in \mathbb{N} \wedge n_0 \geq n_1\}$. Any relation R can also be viewed as a predicate P_R where $P_R(d_0, \dots, d_k) \iff (d_0, \dots, d_k) \in R$, or as a set of tuples, or as a database table.

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 $((1, 2), 3)$. It also means that although our approach extends naturally to relations over arbitrary enumerable domains (such as integers, booleans, symbols/strings, lists of integers, etc)—we make the assumption that natural numbers may be used in the place of other enumerable domains when they are needed. Finally, this means that for operations like union or intersection, both relations must by union-compatible by having the same arity and column names.

... talk about names as indices?

2.1 Standard RA operations

...

Cartesian product. The product of two relations R and S is defined: $R \times S = \{(r_0, \dots, r_k, s_0, \dots, s_j) \mid (r_0, \dots, r_k) \in R \wedge (s_0, \dots, s_j) \in S\}$.

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: $R \cup R' = \{(r_0, \dots, r_k) \mid (r_0, \dots, r_k) \in R \vee (r_0, \dots, r_k) \in R'\}$.

Intersection. The intersection of two relations R and R' may only be performed if both have k arity but is otherwise set intersection: $R \cap R' = \{(r_0, \dots, r_k) \mid (r_0, \dots, r_k) \in R \wedge (r_0, \dots, r_k) \in R'\}$.

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 $\alpha_0, \dots, \alpha_j$, where $\alpha_0 < \dots < \alpha_j$, and is written $\Pi_{\alpha_0, \dots, \alpha_j}(R)$. For each tuple, projection retains only stated columns: $\Pi_{\alpha_0, \dots, \alpha_j}(R) = \{(r_{\alpha_0}, \dots, r_{\alpha_j}) \mid (r_0, \dots, r_k) \in R\}$.

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:

$$\rho_{\alpha_i/\alpha_j}(R) = \{(\dots, r_{\alpha_j}, \dots, r_{\alpha_i}, \dots) \mid (\dots, r_{\alpha_i}, \dots, r_{\alpha_j}, \dots) \in R\}.$$

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,

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
Woodstock '18, June 03–05, 2018, Woodstock, NY

© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9999-9/18/06...\$15.00
<https://doi.org/10.1145/1122445.1122456>

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:

$$\sigma_{\alpha_i=v}(R) = \{(r_{\alpha_0}, \dots, r_{\alpha_k}) \in R \mid r_{\alpha_i} = v\}.$$

Selecting just those tuples from relation R where the values in columns α_i and α_j must match is defined:

$$\sigma_{\alpha_i=\alpha_j}(R) = \{(r_{\alpha_0}, \dots, r_{\alpha_k}) \in R \mid r_{\alpha_i} = r_{\alpha_j}\}.$$

Natural Join. Two relations can also be *joined* into one on a subset of columns they have in common. Join is a particularly important operation that 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
karenk	karenk5@uab.edu	1

logins		
username	timestamp	ipaddr
samp	1554291414	162.103.150.12
karenk	1554181337	171.31.15.120
karenk	1554219962	155.28.11.102
karenk	1554133720	171.31.15.120

A join operation on these two relations, written $\text{users} \bowtie \text{logins}$, yields a single relation with all five columns: username, email, passhash, 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. For the above tables, the natural join is shown:

emails \bowtie logins				
username	email	verified	timestamp	ipaddr
samp	samwow@...	1	...414	162...
samp	samp9@...	0	...414	162...
karenk	karenk5@...	1	...337	171...
karenk	karenk5@...	1	...962	155...
karenk	karenk5@...	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 one row is removed because it becomes a duplicate after projection:

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

email	ipaddr
samwow@gmail.com	162.103.150.12
samp9@uab.edu	162.103.150.12
karenk5@uab.edu	171.31.15.120
karenk5@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 likely associate these indices with their symbolic names. As formalized, the emails relation would be a set of tuples:

$$R_{\text{emails}} = \{ (0, 0, 1), \\ (0, 1, 0), \\ (1, 2, 1) \},$$

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 "karenk" is interned as username 1, and the three emails are interned as emails 0, 1, and 2.

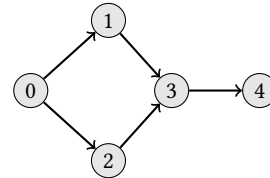
To formalize natural join as an operation on such a relation, we parameterize it by the number of indices that must match, assumed to be the first j of each relation (if they are not, a renaming operation must come first). The join of relations R and S on the first j columns is written $R \bowtie_j S$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

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

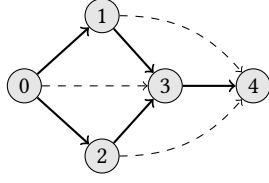
2.2 Application: transitive closure

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 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 node a to node b .

For example, consider graph G (shown below) where $G = \{(0, 1), (1, 3), (0, 2), (2, 3), (3, 4)\}$.



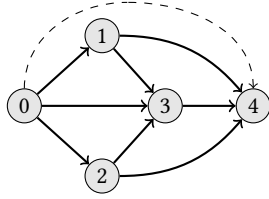
Renaming to swap the columns of G , results in a graph $\rho_{0/1}(G)$ where all arrows are reversed in direction. If this graph is joined with G on only the first column (meaning G is joined on its second columns with G on its first column), we get a set of triples (b, a, c) —specifically $\{(1, 0, 3), (2, 0, 3), (3, 1, 4), (3, 2, 4)\}$ —representing paths of length two in the original graph where a leads to b which leads to c . Projecting out the first column yields pairs (a, c) encoding paths of length two from a to c in the original graph G . If this is unioned with the original G , we obtain a relation encoding paths of length one or two in G . This graph, $G \cup \Pi_{\alpha_1, \alpha_2}(\rho_{0/1}(G) \bowtie_1 G)$, is shown below with new edges (paths of length two) shown in dashes.



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

$$F_G(T) \triangleq G \cup \Pi_{\alpha_1, \alpha_2}(\rho_{0/1}(G) \bowtie_1 T)$$

The graph shown above can be produced by $F_G(G)$ and the graph G is returned if the input graph T is empty: $F_G(\emptyset)$, or $F_G(\perp)$. If F_G is repeatedly applied, the results encodes ever longer paths through G . In this case for example, the graph $F_G(F_G(G))$ or $F_G^3(\perp)$ encodes the transitive closure of G —all paths in G reified as edges.



In the general case, for any graph G , there exists some $n \in \mathbb{N}$ such that $F_G^n(\perp)$ 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 $F_G^n(\perp) = F_G^{n-1}(\perp)$ in a process called *fixed-point iteration*. In the first iteration paths of length 1 are computed, in the second paths of length 1 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.

2.3 Application: Datalog

Computing transitive closure is a simple example of logical deduction. From paths of length 0 (an empty graph) and the existence of edges in graph G , we may deduce the existence of paths of length $0 \dots 1$. From paths of length $0 \dots n$ and the original edges in graph G , we may deduce the existence of paths $0 \dots n + 1$ edges long. The function F_G above performs a single round of this 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 its fixed point, a solution has been found because no further paths may be deduced from the available facts.

In fact, the function F_G is just an encoding in relational algebra of the transitivity property itself, $T(a, b) \wedge T(b, c) \implies T(a, c)$, a logical constraint for which we desire a minimal solution. A graph T satisfies this property exactly when T is a fixed-point for F_T .

Solving logical 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 (such as $T(a, b) \wedge T(b, c) \implies T(a, c)$) and performs a 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. A *Horn clause* is a disjunction of atoms where all but one is negated: $a_0 \vee \neg a_1 \vee \dots \vee \neg a_j$. By DeMorgan's laws we may rewrite this as $a_0 \vee \neg(a_1 \wedge \dots \wedge a_j)$ and note that this is an implication: $a_0 \leftarrow a_1 \wedge \dots \wedge a_j$. In first-order logic, atoms are predicates with universally quantified variables.

A Datalog program is a set of rules $P(x_0, \dots, x_k) \leftarrow Q(y_0, \dots, y_j) \wedge \dots \wedge S(z_0, \dots, z_m)$ and its input is a database of facts called the *intensional database* (IDB). Running the datalog program yields the *extensional database* (EDB) which extends all facts from the IDB with all facts transitively derivable via the program's rules.

In the typical notation of datalog, computing transitive closure of a graph is accomplished with just two rules:

```
path(x, y) :- edge(x, y).
path(x, z) :- path(x, y), edge(y, z).
```

The first says that any edge implies a path (taking the role of the left operand of union in F_G), and the second 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).

Each Datalog rule may be encoded as a function F (between databases) where a fixed point for the function is guaranteed to be a database that satisfies the particular rule. Atoms in the body (premise) of the implication, where two columns are required to match, are refined using a selection operation; e.g., atom $S(a, b, b)$ is computed by RA $\sigma_{\alpha_1=\alpha_2}(S)$. Conjunction of atoms in the body of the implication is computed with a join operation: e.g., in the second rule above, this is the second column of path joined with the first of edge, or $\rho_{0/1}(\text{path}) \bowtie_1 \text{edge}$. These steps are followed by projection to only the columns needed in the head of the rule and any necessary column reordering. Finally, the resulting relation is unioned with the existing relation in the head of the implication to produce F 's output, an updated database (e.g., with an updated path relation in the examples above).

Once a set of functions $F_0 \dots F_m$, one for each rule, are constructed, Datalog evaluation operates by iterating the IDB to a mutual fixed point for $F_0 \dots F_m$.

3 IMPLEMENTATION

This section discusses implementing relational algebra efficiently and in parallel. We take a hybrid approach of nesting key-value stores within a hash-table that can be partitioned across multiple cores or nodes.

3.1 Representations

First, we review the two main approaches to encoding relations in a way that is amenable to fast RA algorithms. Unsurprisingly, algorithms for computing join, union, selection, etc, depend greatly on the representation used for relations themselves.

Decision diagrams. Decision diagrams

Key-value stores. Another approach to encoding relations is to use a hash table, B-tree, prefix tree (trie), or other key-value store.

Hybrid hash-table and b-tree. Our approach is to use an efficient key-value store, but to

Input graph edge count	Union	Join	Transitive Closure	Graph name
412148	✓			
2100225	✓			
6291408	✓			
59062957	✓			
136024430	✓	✓		
180292586	✓	✓		
240023949	✓			

3.2 Hybrid Join

...

3.3 Distributed Join

...

4 EVALUATION

The goal of this section is to evaluate the performance of our implementation of parallel join, parallel union and parallel transitive closure at scale. We first individually study the computation and communication components of the RA operations. Computation is dominated by insertion of tuples and the major challenge faced is that of deduplication. We first study the efficacy of our btree-based relation container for inserts specifically in the context of deduplication. All our RA operations involve an all to all communication phase, hence, we perform a detailed benchmark of MPI's all to all communication capability. Finally we benchmark the efficacy of parallel union, parallel join and parallel transitive closure over a wider range of graphs.

4.1 Dataset and HPC platforms

We have performed our experiments using SuiteSparse Matrix Collection available at []. SuiteSparse Matrix Collection (formerly known as the University of Florida Sparse Matrix Collection), is a large and actively growing set of 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 chose six graphs representing a wide range in terms of the number of edges. The last three graphs are the largest available graphs. Transitive closure of a graph with n edges can generate upto n^2 edges (a fully connected graph). More generally, the number of edges in the transitive closure of a graph depends on the connectivity of the input graph. We found our third graph with X edges to be the most connected; the transitive closure of the graph generated 260 billion edges, which is 3 terabytes in size.

The experiments presented in this work were performed on Theta at the Argonne Leadership Computing Facility (ALCF). Theta is a Cray XC30 with a peak performance of X petaflops, 124, 608 compute cores, 332 TiB of RAM, and 7.5 PiB of online disk storage. We used Edison Lustre file system (168 GiB/s, 24 I/O servers and 4 Object Storage Targets).

4.2 Btree-Relation container

In this section we evaluate the efficacy of our relation container. We measure performance for two cases: insertions of unique tuples and insertions of tuples with duplicates. With the later set of experiments, every tuple had four duplicates. For both set of experiments, we compare two implementations of the relation class, one with a Btree back-end and the other with an hash map back-end. For the hash map, we used `unordered_map` from C++'s standard template library. The results can be seen in Figure 1. The X-axis corresponds to the total number of tuples being inserted and the Y-axis is the time taken for the task to finish. We observe that btree-based relation container outperforms the hash map based implementation for insertion of all tuple counts. Furthermore hash map based relation container fails to scale with insertion of very large number of tuples. For example hash based relation takes X seconds to insert Y tuples as opposed to only Z seconds taken by btree based relation. Similar results can be observed for insertion of tuples with duplicates. The btree based relation container successfully deduplicate tuples while maintaining high performance.

4.3 MPI_All_to_Allv

All to all communication forms the core of all our RA algorithms (refer to the algorithms). We use MPI's `MPI_Alltoallv` function to facilitate all to all data communication. `MPI_Alltoallv` sends data from all to all processes where each process can send a different amount of data by providing displacements for the input and output data. In this section we study both weak and strong scaling characteristics of `MPI_Alltoallv`. For both set of experiments, we varied the number of processes from 2048 to 32768. We performed 9 set of weak scaling experiments. For these 9 set of experiments, the amount of data transmitted by each process ($data_{process}$) was varied from 4 megabytes (1st set) to 1024 megabytes (9th set). For a n process run, every process transmits $data_{process}/n$ units of data to every other process. With strong scaling experiments, we performed 6 set of experiments, varying the total amount of data generated across processes ($data_{total}$) from 64 gigabytes (1st set) to 2048 gigabytes (6th set). The amount of data generated by every process is the same, for example for a n process run, and $data_{total}$ total amount of data, every process produces $data_{total}/n$ units of data. A process then transmits $data_{total}/n^2$ units of data to every other process. The results of both weak and strong scaling experiments can be seen in Figure 2.

For both strong and weak scaling runs, we observe a decline in performance with decreasing workload. For instance, with strong scaling, the 6th set of experiments where total workload is 2048 gigabytes, we observe near perfect scaling when the number of processes is doubled from 2048 (18.5 seconds) to 4096 (7.3 seconds) to 8192 (4.5 seconds). After 8192 processes, although total time continues to come down with increasing process count, we observe that the rate becomes much slower. Furthermore, looking at the 6th set of experiments where total workload is 64 gigabytes we observe relatively poor scaling characteristics across the entire process range. Both the observations can be attributed to reduced per-process workload. With less amount of data to transmit, total time is dominated by initialization costs as opposed to data transmission costs. Similarly for weak scaling experiments as well, when the

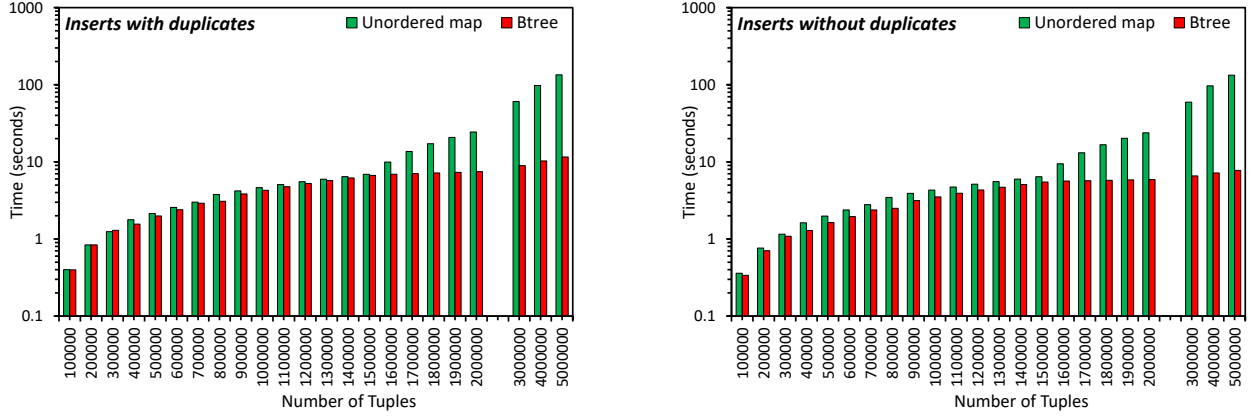


Figure 1: Performance evaluation of relation class implemented with btree and unordered map. (left) All tuples are distinct, (right) There are four copies of every tuple being inserted. Relation implemented with btree out-performs the unordered-map implementation.

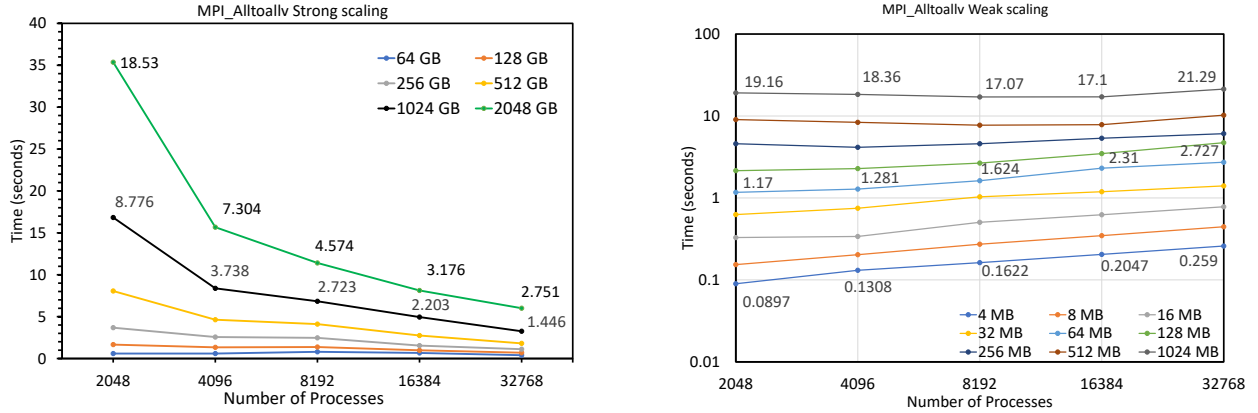


Figure 2: Strong (left) and Weak (right) scaling evaluation of MPI_alltoallv function of MPI.

amount of data exchanged is substantial, we see almost perfect scaling. For example, when the amount of data transmitted by every process is 1024 megabytes, we observe almost perfect scaling, whereas when the amount of data sent by every process is small (4MB) we observe poor scaling.

With the context of communication requirements of RA operations, we find the scaling trends of MPI_alltoallv to be encouraging. In general for a given workload (graph size for RA operations), there will always be a range of processes that exhibits good all to all scaling characteristics. We will have the challenging task to identify the right process count that balances the tradeoff between computation and communication. As we will see later in section 4.6 with

larger per-process load computation cost dominates as opposed to smaller per-process workload where total cost is dominated by communication.

4.4 Parallel Union

– union of 7 graphs from table – union comprises of an io phase followed by communication and then followed by inserts – We compare two union types, one is where we perform io, comm and compute separates, the other is where we first perform io for all and then we bundle all our comm and then we have one phase of compute – scales well upto X cores., this is strong scaling. – faces

work load deprecation at low core counts, needs more tuples for union to scale at high core counts – overall a good sign

4.5 Parallel Join

– setup, join of two of the largest graph, produces X number of tuples in output. – similar to union join scales well upto X cores, after that there is shortage of work and we do not see proper scaling.
– communication stops to scale after X core counts. This is because there is lack of work. –

4.6 Transitive closure

Computing the transitive closure of a graph involves repeated join operations until a fixed point is reached. We use the previously discussed radix-hash join algorithm to distribute the tuples across all processes. The algorithm can then be roughly divided into four phases: 1) Join 2) network communication 3) insertion 4) checking for a fixed point. In our join phase every process concurrently computes the join output of the local tuples. In the next phase every process sends the join output results to the relevant processes. This is a all-to-all communication phase, which we implemet using MPIâ€™s all to all routines. The next step involves inserting the join output result received from the network to the output graphâ€™s local partition. In the final step we check if the size of the output graph changed on any process, if it does then we have not yet

reached a fixed point and we continue to another iteration of these 4 steps. We performed a set of strong-scaling experiments to compute the transitive closure of graph with 412148 edgesâ€™the largest graph in the U. Florida Sparse Matrix set (Davis and Hu 2011). We used the Quartz supercomputer at the Lawrence Livermore National Laboratory (LLNL). For our runs, we varied the number of processes from 64 to 2048. A fixed point was attained after 2933 iterations, with the resulting graph containing 1676697415 edges. As can be seen in Figure 1, our approach takes 462 seconds at 64 cores and 235 seconds at 2048 cores, corresponds to an overall efficiency of 6.25of by the four major components (join, network communication, join, fixed-point check) of the algorithm. We observe (see Figure 2) that for all our runs the total time is dominated by computation rather than communication; insert and join together tended to take up close to 90as it shows that we are not bound primarily by the network bandwidth (at these scales and likely moderately higher ones) and it gives us the opportunity to optimize the computation phase

5 RELATED WORK

Lorem ipsum dolar sit amat

6 CONCLUSION

Lorem ipsum dolar sit amat

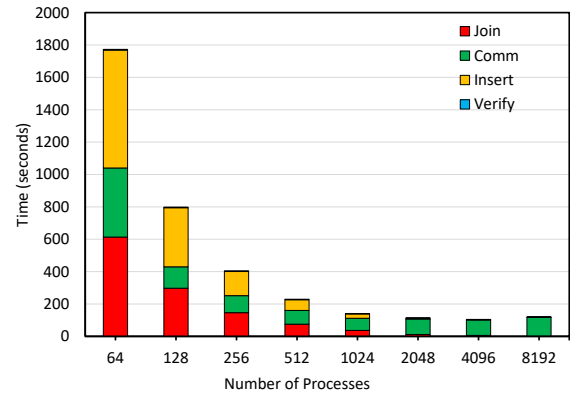
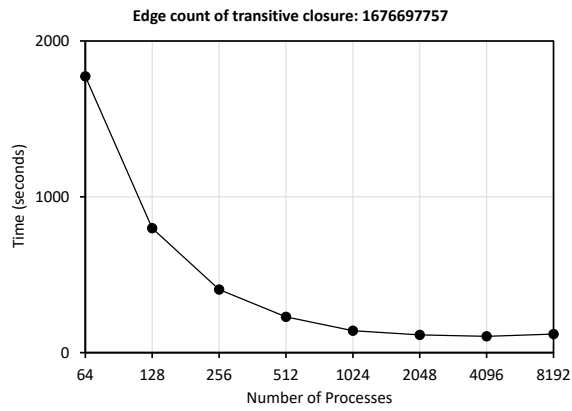


Figure 3: Transitive closure of a graph with X edges.

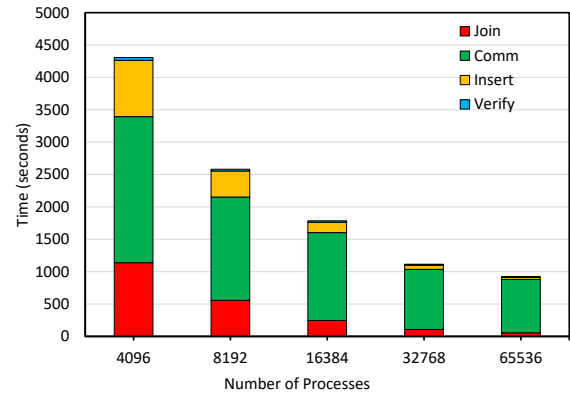
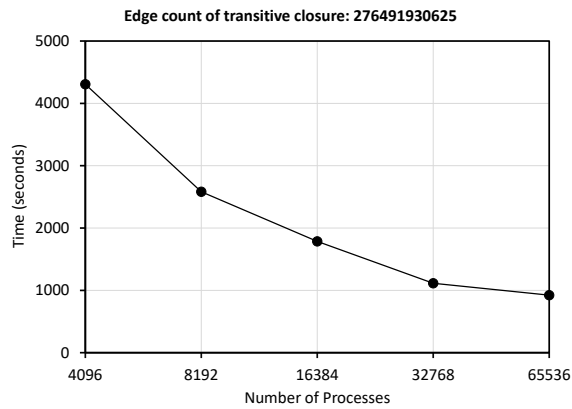


Figure 4: Transitive closure of a graph with X edges.