Exploring MPI Collective I/O and File-per-process I/O for Checkpointing a Logical Inference task

Ke Fan
University of Alabama at Birmingham
kefan@uab.edu

Kristopher Micinski
Syracuse University
kkmicins@syr.edu

Thomas Gilray
University of Alabama at Birmingham
gilray@uab.edu

Sidharth Kumar
University of Alabama at Birmingham
sid14@uab.edu

***Abstract*—We present a scalable parallel I/O system for a logical-inferencing application built atop a deductive database. Deductive databases can make logical deductions (i.e. conclude additional facts), based on a set of program rules, derived from facts already in the database. Datalog is a language or family of languages commonly used to specify rules and queries for a deductive database. Applications built using Datalog can range from graph mining (such as computing transitive closure, k-cliques, or FSM) to program analysis (~~control and data flow analysis~~). In our previous papers, we presented the first implementation of a data-parallel Datalog built using MPI. In this paper, we present a parallel I/O system used to checkpoint and restart ~~of~~ applications built on top of our Datalog system. State of the art Datalog implementations, such as Soufflé, only support serial I/O, mainly because the implementation itself does not support many-node parallel execution.**

Computing the transitive closure of a graph is one of the simplest logical-inferencing application built using Datalog; we use it as a micro-benchmark to demonstrate the efficacy of our parallel I/O system. Internally, we use a nested B-tree data-structure to facilitate fast and efficient in-memory access to relational data. Our I/O system therefore involves two steps, converting the application data-layout (a nested B-tree) to a stream of bytes followed by the actual parallel I/O. We explore two popular I/O techniques POSIX I/O and MPI collective I/O. For extracting performance out of MPI Collective I/O we use adaptive striping, and for POSIX I/O we use file-per-process I/O. We demonstrate the scalability of our system at up to $4{,}096$ processes on the Theta supercomputer at the Argonne National Laboratory.

I. Introduction

High-performance computing (HPC) applications are designed to enable massively-scalable solutions to challenging problems, but come at the expense of more complicated programming techniques to perform common tasks including file input and output (I/O). There has been a wealth of work on parallel I/O techniques [14], [18], [21], [25], [30], especially for traditional scientific applications [8], [26]. In our papers [12], [19], [20], we presented the first implementation of distributed relational algebra that forms the basis of a data-parallel Datalog. Datalog is a language used to specify facts, rules and queries in deductive databases. Deductive databases combines logic programming with relational databases to enable logical deductions (i.e. conclude additional facts) based on existing rules combined with input and inferred facts. Our work has facilitated a new class of HPC applications: declarative logical inferencing at scale. In this paper, we present a preliminary exploration of scalable I/O for HPC applications built on top of our data parallel Datalog, ones that leverage *parallel relational-algebra* [19]. These applications readily enable massively-parallel graph mining (e.g., transitive closure and k-clique) and program analysis (e.g., context-sensitive points-to analysis).

Relational algebra (RA) forms the underlying primitives for state-of-the-art deductive reasoning systems, including the Soufflé Datalog engine [27]. These systems commonly utilize some task-level parallelism (e.g., using PThreads or OpenMP) and run on a single node, and utilize standard serial I/O through the usual POSIX APIs. Systems built on parallel RA instead run on clusters, and thus require a different approach to enable parallel I/O across an entire cluster rather than a single node.

Datalog programs such as transitive-closure computation involve running relational algebra operations (e.g., join, projection, union) on relations until a fixed point is reached. This may be related to typical HPC applications that simulate scientific phenomenon, often involving data changing across time. Traditional HPC applications typically apply scientific models on data stored across grids/meshes (structured, unstructured or adaptive). Our application works on relations—tables of data. Relations are mainly unstructured data, however, our application works on discrete integral entities that requires tasks like deduplication, and iterating over a range of values, therefore necessitating us to use complex data structures that supports these tasks efficiently. Our parallel RA implementation is built on top of a nested B-tree data structure.

Traditional HPC applications deploy parallel I/O in the form of checkpointing as a popular fault-tolerance technique, where the entire state of the simulation is saved on the disk, in case there is a need to restart the simulation from a particular timestamp. With our parallel I/O system, we enable our Datalog applications to checkpoint data by writing our all relations at regular intervals. Since we deal with unstructured data which are stored in memory using a nested B-tree, our I/O system involves two primary steps: first, we convert the nested trees that form a relation's index (storing the table data, except organized for a particular access pattern) into a stream of bytes in memory. Next, is parallel I/O. The number of relations generated will depend on the specific Datalog program. For example, transitive closure generates only two

relations, the input graph and the output graph; on the other hand, even a simple *k-CFA* Datalog program generates around 20 relations. From an I/O perspective, there are several ways to pack these relations. In general, one can have files storing a tunable number of relations. From a technique perspective, one can use MPI-collective I/O that writes data to a shared file or write data in a file-per-process mode where every process writes data for every relation to a separate file. For initial exploration, we use transitive closure as a micro-benchmark and present and compare two popular I/O techniques: file-per-process I/O using POSIX I/O and shared file I/O using MPI collective I/O. We demonstrate the efficacy of our parallel I/O system by enabling checkpointing and restart capabilities for our RA based applications. In this paper, we present two novel contributions: (1) we describe the first parallel I/O system for a distributed deductive database, and (2) we present a preliminary exploration and comparison of two popular approaches to parallel I/O in this context.

II. Background

First, we will review the major components of our data-parallel deductive database, using path-finding in graphs as an illustrative example. In query languages for deductive databases, rules can be provided to define additional relations (tables) defined in terms of others.

```
B(x,y) :- G(x,y), G(y,x), x<y.
```

The above rule infers a relation B which gets a single tuple (x, y) for each bi-direction edge in an input table G, that encodes a graph. Because G appears twice in the body of the rule, we must effectively join the table with itself; then, a final constraint, $x < y$, filters this output so that edges are added to B only once, in a canonical order.

A. Deductive Databases

The databases rules can also be recursive, as in a Datalog program for computing the transitive closure, T, of a graph G:

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

The first rule represents a base case that says every x-to-y edge in G implies an immediate x-to-y path in T. The second rule is recursive and must be iterated repeatedly until stabilizing at a consistent value for T. The first rule can be implemented using a single relational union of G and T (unless we can assume T is empty), or using insertion of every element in G into T. The second rule can be implemented by iteration of a kernel function, composed of several relational operations, iterated to a least-fixed-point where T is minimally consistent with the second rule. One iteration of this function would join T on its second column with G on its first column, yielding all triples (x, y, z) where (x, y) can be drawn from T and (y, z) can be drawn from G. Projection to the set of unique (x, z) tuples, removing the middle column (as a graph, this is removing the intermediate vertex in the discovered path), and unioning this set of tuples with those in T completes one iteration of the second rule.

Consider a relation G, shown below as a table. Joining G on its second column with G on its first column yields a new relation, with three columns, encoding all paths of length 2 through the graph G, where each path is made of three nodes in order.

G

0	1
A	**B**
A	**C**
B	**D**
C	**D**
D	**E**

G joined with G

0	1	2
A	**B**	**D**
A	**C**	**D**
B	**D**	**E**
C	**D**	**E**

$\rho_{0/1}(\rho_{0/1}(G) \bowtie_1 G)$

Note that to compute a single join of G on its second column with G on its first column, we first reverse G's columns, computing $\rho_{0/1}(G)$ to reorder columns, so we may then compute a join on one column: $\rho_{0/1}(G) \bowtie_1 G$. To present the resulting paths of length two in order again, we may use renaming to swap the join column back to the middle position, as shown above. Our implementation provides more general operations that make this administrative renaming unnecessary. In the case of iterating the two rules above to compute a transitive closure of G, we use a persistent index for G, keyed on its first column, and a persistent index on T, keyed on its second column.

We can encapsulate each iteration of TC computation as a function Extend_G which takes a graph T, and returns T's edges extended with G's edges, unioned with G.

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

Consider a new graph G, at the top of Figure 1. The graph T, below, is returned by $\mathit{Extend}_G(\perp)$, the graph below it is returned by $\mathit{Extend}_G{}^2(\perp)$, the graph below that is returned by $\mathit{Extend}_G{}^3(\perp)$, etc. As Extend_G is repeatedly applied from an empty input, each result encodes ever longer paths through G, as shown. In this case for example, the graph $\mathit{Extend}_G{}^4(\perp)$ encodes the transitive closure of G—all paths in G reified as edges. One final iteration, computing $\mathit{Extend}_G{}^5(\perp)$, is required to check that the process successfully reached a fixed point for Extend_G.

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 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 deduce the existence of paths of length $0 \ldots n + 1$. This kind of logical inference forms the semantics of *Datalog*, a bottom-up logic-programming language supporting a restricted logic corresponding roughly to first-order HornSAT—the SAT problem for conjunctions of Horn clauses [4].

A Datalog program is a set of such 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 initial facts called the *extensional database* (EDB). Running the datalog program makes explicit

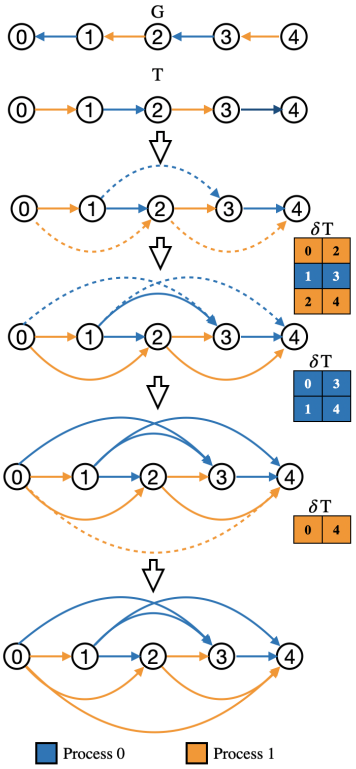

Fig. 1. Transitive closure of a string graph distributed over two processes.

the *intensional database* (IDB) which extends facts in the EDB with all facts transitively derivable via the program's rules. Each Datalog rule may be encoded as a monotonic function F (between databases) where a fixed point for the function is guaranteed to be a database that satisfies the particular rule. 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 $F_0 \dots F_m$. Datalog inference must be monotonic, so that evaluation is strictly increasing, however a Datalog program may also be decomposed into a stratified directed acyclic graph (DAG) of strongly connected components (SCCs) encoding sets of mutually recursive rules. In practical Datalog implementations, such as Soufflé [15], [27], the stratification of Datalog rules into SCCs also permits the weakening of monotonicity constraints and inclusion of negation and aggregation operations across SCCs. Crucially, once a Datalog program is compiled to SCCs, each SCC may be treated as an independent Datalog program with its own EDB and IDB—a property we can exploit when using IO for checkpointing.

B. Implementing Parallel Deductive Databases

Beyond the basic process described, typical Datalog implementations use highly efficient and compressed data-structures [16], [17], automated selection of efficient indices [29], and incrementalization or semi-naive evaluation [4]. Our implementation distributes relations across a set of MPI processes, using nested B-trees locally to store indices.

The double-hashing approach, with local hash-based joins and hash-based distribution of relations, is the most commonly used method to distribute join operations over many nodes in a networked cluster computer. This algorithm involves partitioning relations by their join-column values so that they can be efficiently distributed to participating processes [5], [7]. The main insight behind this approach is that for each tuple in the outer relation, all relevant tuples in the inner relation must be hashed to the same MPI process or node, permitting joins to be performed locally on each process.

Our recent approach proposes adapting the representation of imbalanced relations by using a two-layered distributed hash-table to partition tuples over a fixed set of *buckets*. Within each bucket, tuples are assigned to one element of a dynamic set of *subbuckets* which may vary across buckets [19] and across time. Each tuple is assigned to a bucket based on a hash of its join-column values, but within each bucket, tuples are hashed on non-join-column values, assigning them to a local subbucket, then mapped to an MPI process. This permits buckets that have more tuples to be split across multiple processes, but requires some additional communication among subbuckets for any particular bucket.

Our implementation heavily relies on all-to-all communication as tuples produced by a local join may belong to another rank and must be moved before a subsequent iteration. Consider Figure 1, which show initial graphs G and T and each step of TC computation in a database distributed over two processes. T is initially the same as graph G, but is indexed on its second column, so arrows are shown reversed. Each key value (vertex) is hashed to assign it to a bucket, and thus a process (shown as arrows colored blue or orange). Note how in the first iteration, an edge $T(0,1)$ keyed on value 1 and assigned to process 0 is composed with edge $G(1,0)$, keyed on value 1 and also on process 0, producing a new edge $T(0,2)$ that is keyed on value 2 and assigned to process 1. This requires our all-to-all communication phase between iterations which shuffles newly learned facts to their host process.

The Datalog program for TC shown above, compiles down to a program of two indices and two SCCs, one SCC for fixed initialization, and another SCC that performs an unbounded number of iterations to reach a fixed point for T.

The end-to-end pipeline showing each major phase of a parallel join can be seen in Figure 2. Intra-bucket communication first replicates tuples whose keys have matching hash values on a single node, only for the left-hand relation so that joins can be done efficiently in parallel. After the local join phase, an all-to-all phase propagates output tuples to their destination relations where they are inserted locally. At this point the iteration is over and can progress, with an optional I/O phase to save this point in evaluation.

III. Parallel I/O Implementations

Besides writing the relations once a fixed point is reached, one of the main goals of our parallel I/O system is to enable checkpointing and restarts. Checkpointing is a popular fault-tolerant technique where the current state is written to facilitate

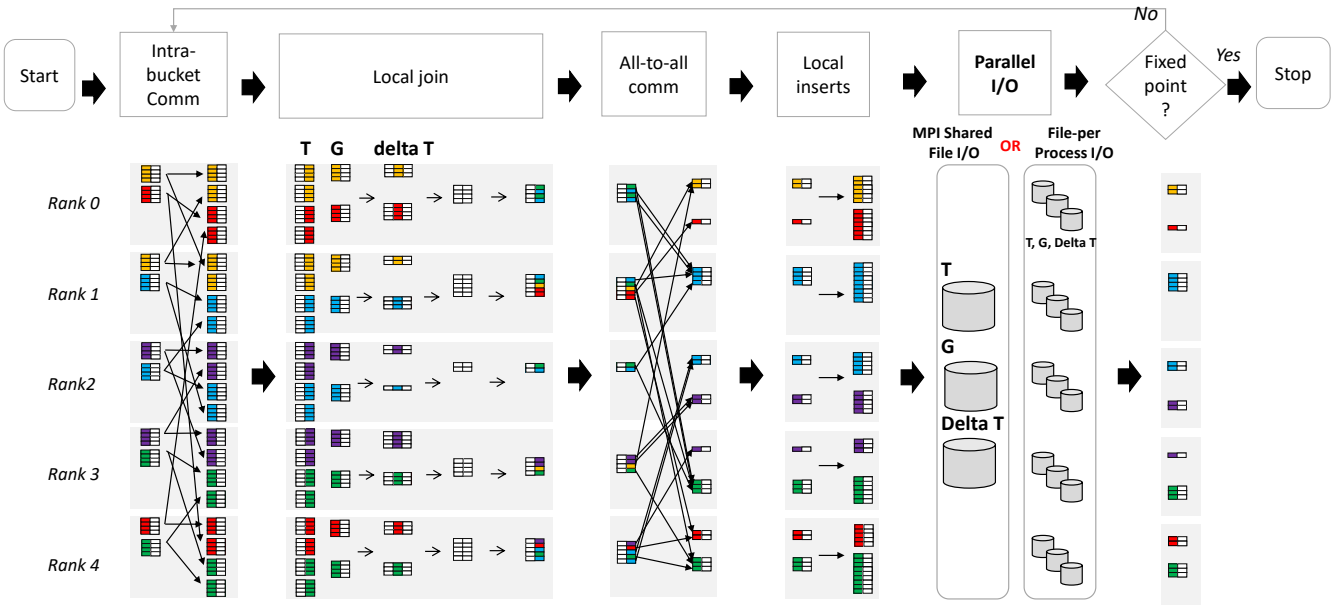

Fig. 2. Shows the major phases of a balanced join in the context of a TC computation [12].

a seamless restart. As shown in the previous section, our application is broken into a series of stratified tasks that operate on sets of relations. For example, TC consists of two stratified tasks and two relations (G and T). Checkpoints save the entire state of each relation, along with metadata about which tasks have completed. Each relation is marked either static or dynamic, depending on whether that relation is written to within a given task. Each relation also has several versions: static relations (not written to within a given task) store only a *total* version, while dynamic relations (which update as the task progresses) include both a total and *delta* version. *Delta* contains the new tuples produced at a given iteration, while *total* contains all existing tuples for a given relation. For example, the G relation in our TC program is static and thus only the *total* version is stored, whereas T is dynamic so both a *delta* and *total* version are stored.

With respect to parallel I/O, our RA-based system differs from traditional HPC applications in two ways: data representation and workload distribution. As opposed to typical grid-based HPC applications, our system stores tuple data for each relation in a nested B-tree data structure. To materialize a writable byte stream we must iterate over this nested B-tree. To handle workload distribution we use balanced hash-trees [20], which partition a relation across processes into similarly-sized chunks. The load balancing is approximate and does not guarantee uniform workload distribution across processes. Before writing data, processes perform an explicit metadata exchange phase to get a coherent view of the file.

There exists several high level parallel I/O systems such as PnetCDF [24], Parallel HDF5 [30], PIDX [23] and ADIOS [13] that are commonly used by scientific applications. Many of these I/O libraries such as PnetCDF and Parallel HDF5 are built on top of MPI I/O and default to one shared file. Despite increased convenience in dealing with one single file, existing work such as [11], [22] has demonstrated that it is often necessary to organize writes into a hierarchy of files to achieve optimal performance. We have therefore enabled our system with both MPI collective I/O that can write data to a shared file and POSIX I/O that can enable file-per-process I/O. We expose the parameter *cp_iteration* and *cp_mode*, that controls the frequency and mode (POSIX I/O or MPI I/O) of checkpoint dumps. Figure 2 shows how our I/O system fits in our parallel RA pipeline.

Portable Operating System Interface (POSIX) I/O is used to enable each process to write each relation separately. We perform file-per-process writes for both delta and full of every dynamic relation. Therefore, n processes and r relations may generate up to $2 \times n \times r$ (based on how many relations include delta versions) files. No metadata exchange is required for writing these files, as processes write data to their respective files in parallel. We also perform basic filtering to prevent empty files being created by processes that have no tuples for a given relation; this is especially useful given that delta is often sparsely distributed across processes.

MPI Collective I/O [24] is used for writing a shared file. It enables I/O optimizations that analyzes and merges I/O requests of processes. Merged I/O requests combine non-contiguous requests of individual processes into a single larger contiguous request. Before invoking collective I/O, we perform a metadata exchange phase wherein every process shares the size of each of its relations with each other process. Each process then uses this information to compute their appropriate starting offset into the file. We also perform adaptive striping, which selectively stripes files greater than 1 GiB across 48

OST with a striping unit of 8 MiB. This configuration is recommended by the ALCF guidelines for I/O performance on Theta [9]. Adaptive striping helps us extract better performance from the shared filesystem.

In our current implementation, with both modes, we do not write data from multiple relations to a common file. As can be seen in the evaluation section, this method is scalable for transitive closure. However, we believe that with more complex datalog programs, which will generate hundreds of relations, we will have to develop a system where we can write a tunable number of relation to a file.

A. Metadata

Metadata files store information which is necessary for restarting from a checkpoint. We have two metadata files: *scc-metadata* and *offset-metadata*. *scc-metadata* file stores a list of terminated tasks so that they may be skipped upon restart. The *offset-metadata* is needed only for shared files (one for every relation), and contains the local process size along with the global offset for every process. `MPI_Allgather()` is used to populate offset metadata, and is written by process with rank 0. This metadata is used during restarts by each process to load tuples specific to that process, avoiding a more expensive phase to shuffle tuples to the appropriate relation after reading.

IV. Evaluation

We use transitive closure (TC) computation as a microbenchmark to test the scalability of our parallel I/O system. We demonstrate the efficacy of parallel *writes* through checkpoint dumps and parallel *reads* through restarts. We corroborated the timings of our experimental results using Darshan [6] logs.

A. Dataset and HPC platforms

We performed our experiments using graphs from the SuiteSparse Matrix Collection [10]. For our experiments, we used the graph [1] containing 412,148 edges. There were 5,866 iterations necessary for our TC implementation to reach a fixed-point, generating a transitive closure of 1,676,697,757 edges. Space complexity for TC is quadratic—an input graph with n edges may generate a fully-connected transitive closure consisting of n^2 edges.

All experiments were performed on the Theta Supercomputer [2] at the ALCF. Theta is a Cray machine with peak performance of 11.69 petaflops, 281,088 compute cores, 843.264 TiB of DDR4 RAM and 10 PiB of online disk storage. Experiments used Theta's Lustre filesystem [3].

TABLE I
The size of 6 checkpoints

Checkpoint ID	0	1	2	3	4	final
Total size (GiB)	13	26	39	52	64	75

B. Checkpointing: parallel writes

We tested the efficacy of parallel writes via a set of strong scaling experiments whose results are shown in Figure 3. Checkpoints were dumped every 1,000 iterations. Given this checkpoint frequency and total of 5,866 iterations, the transitive closure (TC) computation generated 5 intermediate checkpoint dumps and one final output dump. The total sizes of each of these six checkpoints is shown in Table I. Our TC computation uses a balanced hash-tree join algorithm [20] to ensure workload is uniformly-distributed across processes. We varied the process count from 256 to 4,096, with the smallest run (256 processes) corresponding to 52 MiB (13 GiB/256) for the first checkpoint and 300 MiB (13 GiB/256) for the final checkpoint. We ran our experiments used both POSIX I/O and MPI collective I/O. We also executed IOR tests at each process count. IOR is a general-purpose parallel I/O benchmark [28] which we configured in this case to generate one file per process with the total workload similar to the final checkpoint dump (75 GiB). IOR was set to perform POSIX I/O with fsync enabled. The IOR performance gives us a measure of the maximum performance achievable for the filesystem.

We observed three different trends: (1) for all cases POSIX I/O demonstrates much better scalability than MPI collective I/O. For example, at 256 processes POSIX I/O writes data at 3.797 GiB/second (for final checkpoint), compared to 1.724 GiB/second with MPI collective I/O. Shared file I/O demonstrates sub-par performance as it involves a data aggregation phase that adds extra overhead. (2) For process counts, 256, 512 and 1,024, despite having different aggregate workloads, bandwidth across all checkpoint dumps remains roughly the same, whereas for process counts $2,048$ and $4,096$, we observe decreasing performance with smaller workload (initial checkpoints). For example, with POSIX I/O at process count 512, the first and last checkpoint have a bandwidth of 8.334 and 7.577 GiB/sec whereas at process count $4,096$ the two checkpoints have a bandwidth of 26.925 and 52.783 GiB/seconds. These observations may be attributed to an overall reduction in per-process workload with increasing process-counts, and with less data to write, total time is dominated by initialization costs. (3) We observe that we under-perform when compared to IOR file-per process I/O, this trend can be better understood by performing a component-wise breakdown of all I/O components (Figure 4).

In Figure 4 we graph time taken by the three components of our I/O scheme: metadata I/O, populating the file I/O buffer and actual file I/O operation. The results correspond to aggregated timing across all checkpoint dumps, therefore the blue chunk of the bar corresponds to the total time taken in populating the file I/O buffer across all 6 checkpoints. Populating file I/O buffer is very unique to our application, as for a typical grid-based scientific application data is stored in-memory in arrays that can be directly mapped to files. Unlike traditional HPC applications, our relational data exists in a nested B-tree data structure, which needs to be iterated over to obtain a stream of writable bytes. Looking at the graph, we

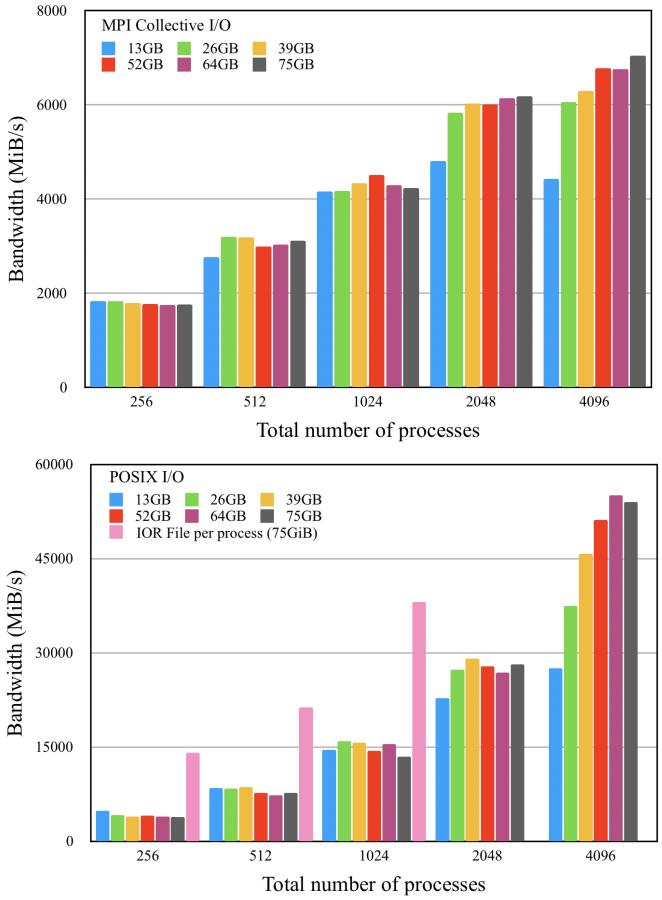

Fig. 3. Strong scaling evaluation of MPI Collective I/O (top) and Posix I/O (bottom) for writing all checkpoints of table I.

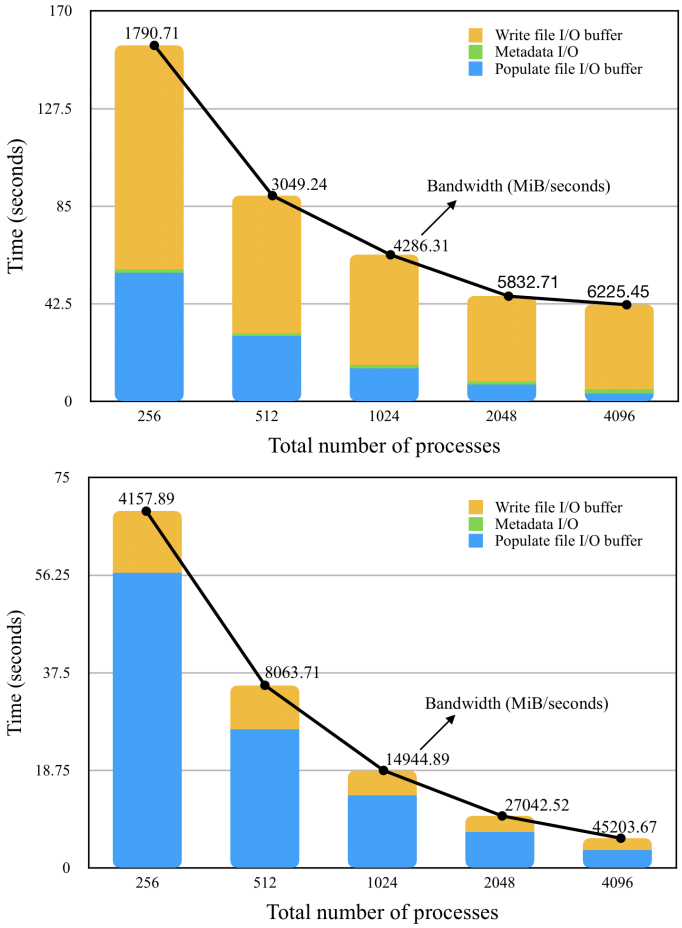

Fig. 4. Time break down of the three I/O components for MPI collective I/O (top) and Posix I/O (bottom).

can see that for all process counts, a major portion of time is spent during populating the file-I/O buffer. For example, at $4,096$ processes, with file per-process I/O, file I/O buffer takes 3.396 seconds which is 59.414 percent of the total I/O time. Our file I/O time is comparable to the actual IOR timings. Another related trend to note is that with increasing process count, populating file I/O buffer time reduces linearly, taking a smaller fraction of the total I/O time.

C. Restarting: parallel reads

We evaluate the efficacy of both our I/O schemes for parallel reads through restart experiments. Typically a restart is done using the same number of processes as was used to write the data. For shared files this can be efficiently facilitated by the *offset-metadata* file, using which every process can concurrently read the appropriate tuple directly from the data file. Figure 5 shows the performance of MPI collective I/O and POSIX I/O when restarting from the largest checkpoint with process counts 256, 1,024 and 4,096. Similar to parallel write trends, we observe POSIX I/O outperform MPI I/O; at 4,096 we report a bandwidth of 82.193 GiB/sec with POSIX I/O.

Restarts can also be performed using a different number of processes than was used to write the original data. Under this approach, each process reads the same amount of data from the file and then uses communication to transmit data to the appropriate process. These kinds of reads can be simulated by avoiding the use of the *offset-metadata* file. In Figure 6, we plot the timings to read data from a shared file with and without the *offset-metadata* file at 256 and 512 processes. In the figure, the blue bar is the time for reading data, and the green bar is the time for data transmission. We observe that the time taken for reading with *offset-metadata* is more than the time without the metadata file. This is because of loading and parsing overhead associated with the metadata. However, with the *offset-metadata* there is no communication time as tuples are correctly placed during file I/O itself. Therefore, the total time with *offset-metadata* is less compared to the other scheme that involves data transmission to send the tuples to the appropriate processes.

V. Conclusion and Future Work

In this paper, we presented a scalable checkpointing framework for a simple logical-inferencing task built using

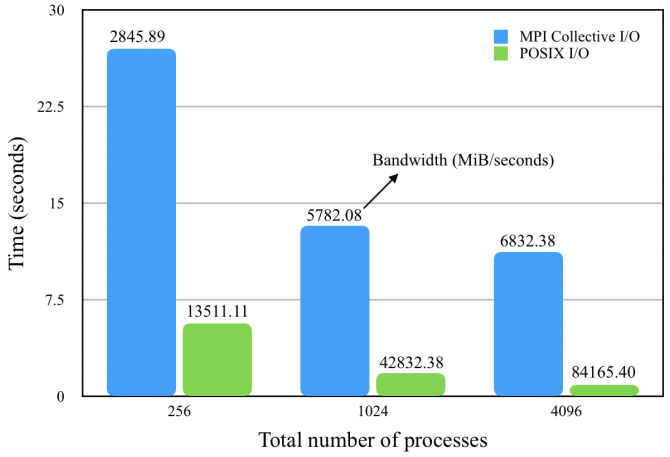

Fig. 5. Restarting from the largest checkpoint with MPI collective I/O versus POSIX I/O

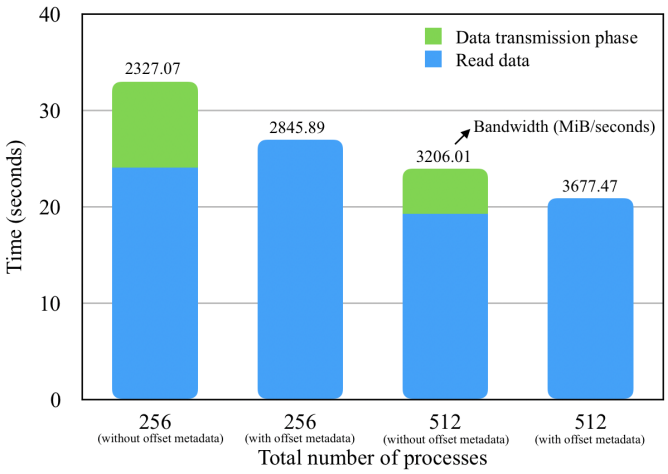

Fig. 6. Comparison of MPI collective I/O with and without *offset-metadata*.

our data-parallel Datalog [12]. We demonstrate that file-per-process consistently outperforms MPI-collective I/O. Although, the current framework is able to handle a simple logical-inferencing task, such as the transitive closure computation, we believe that a more complex I/O system will be required to handle larger Datalog programs from domains such as program analysis and business analysis. More complex Datalog programs will generate hundreds or thousands of relations. This work is the first step in developing a complete parallel I/O solution for our data-parallel Datalog system. Moving forward, we envision adding two degrees of freedom in our I/O framework, one controlling the total number of output files, and the other controlling the total number of relations written to a file. We want to create a file-level data layout that will facilitate effective queries on the datasets (the current implementation is devoid of any structure) and we want to explore compression techniques for relational data.

References

[1] https://www.cise.ufl.edu/research/sparse/matrices/CPM/cz40948.html.

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

[3] Theta-file-system. https://www.alcf.anl.gov/support-center/theta/theta-file-systems.

[4] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of databases: the logical level*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[5] Filippo Cacace, Stefano Ceri, and Maurice 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.

[6] Philip Carns, Kevin Harms, William Allcock, Charles Bacon, Samuel Lang, Robert Latham, and Robert Ross. Understanding and improving computational science storage access through continuous characterization. *ACM Trans. Storage*, 7(3), October 2011.

[7] Jean-Pierre Cheiney and Christophe 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.

[8] J H Chen, A Choudhary, B de Supinski, M DeVries, E R Hawkes, S Klasky, W K Liao, K L Ma, J Mellor Crummey, N Podhorszki, R Sankaran, S Shende, and C S Yoo. Terascale direct numerical simulations of turbulent combustion using s3d. In *Computational Science and Discovery Volume 2*, January 2009.

[9] Paul Coffman, Francois Tessier, Preeti Malakar, and George Brown. Parallel I/O on Theta with Best Practices. Talk at ALCF Simulation, Data, and Learning Workshop, 2018.

[10] Timothy A. Davis and Yifan Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1):1:1–1:25, December 2011.

[11] K. Gao, W. Liao, A. Nisar, A. Choudhary, R. Ross, and R. Latham. Using subfiling to improve programming flexibility and performance of parallel shared-file i/o. In *2009 International Conference on Parallel Processing*, pages 470–477, 2009.

[12] Thomas Gilray, Sidharth Kumar, and Kristopher Micinski. Compiling data-parallel datalog. In *Proceedings of the 30th ACM SIGPLAN International Conference on Compiler Construction*, CC 2021, page 23–35, New York, NY, USA, 2021. Association for Computing Machinery.

[13] William F. Godoy, Norbert Podhorszki, Ruonan Wang, Chuck Atkins, Greg Eisenhauer, Junmin Gu, Philip Davis, Jong Choi, Kai Germaschewski, Kevin Huck, Axel Huebl, Mark Kim, James Kress, Tahsin Kurc, Qing Liu, Jeremy Logan, Kshitij Mehta, George Ostrouchov, Manish Parashar, Franz Poeschel, David Pugmire, Eric Suchyta, Keichi Takahashi, Nick Thompson, Seiji Tsutsumi, Lipeng Wan, Matthew Wolf, Kesheng Wu, and Scott Klasky. Adios 2: The adaptable input output system. a framework for high-performance data management. *SoftwareX*, 12:100561, 2020.

[14] Jianwei Li, Wei-keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netCDF: A High-Performance Scientific I/O Interface. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, 2003.

[15] Herbert Jordan, Bernhard Scholz, and Pavle Subotić. Soufflé: On synthesis of program analyzers. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification*, pages 422–430, Cham, 2016. Springer International Publishing.

[16] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard 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. ACM.

[17] Herbert Jordan, Pavle Subotić, David Zhao, and Bernhard 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.

[18] S. Kumar, V. Vishwanath, P. Carns, J.A Levine, R. Latham, G. Scorzelli, H. Kolla, R. Grout, R. Ross, M.E. Papka, J. Chen, and V. Pascucci. Efficient data restructuring and aggregation for I/O acceleration in PIDX. In *High Performance Computing, Networking, Storage and Analysis (SC), 2012 International Conference for*, pages 1–11, Nov 2012.

[19] Sidharth Kumar and Thomas Gilray. Distributed relational algebra at scale. In *International Conference on High Performance Computing, Data, and Analytics (HiPC)*. IEEE, 2019.

[20] Sidharth Kumar and Thomas Gilray. Load-balancing parallel relational algebra. In Ponnuswamy Sadayappan, Bradford L. Chamberlain, Guido Juckeland, and Hatem Ltaief, editors, *High Performance Computing*, pages 288–308, Cham, 2020. Springer International Publishing.

[21] Sidharth Kumar, Steve Petruzza, Will Usher, and Valerio Pascucci. Spatially-aware parallel i/o for particle data. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, pages 84:1–84:10, New York, NY, USA, 2019. ACM.

[22] Sidharth Kumar, Steve Petruzza, Will Usher, and Valerio Pascucci. Spatially-aware parallel i/o for particle data. In *Proceedings of the 48th International Conference on Parallel Processing*, ICPP 2019, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Sidharth Kumar, Venkatram Vishwanath, Philip Carns, Brian Summa, Giorgio Scorzelli, Valerio Pascucci, Robert Ross, Jacqueline Chen, Hemanth Kolla, and Ray Grout. PIDX: Efficient parallel I/O for multi-resolution multi-dimensional scientific datasets. In *IEEE International Conference on Cluster Computing*, 2011.

[24] Jianwei Li, Jianwei Li, Jianwei Li, Wei-keng Liao, Alok Choudhary, Robert Ross, Rajeev Thakur, William Gropp, Rob Latham, Andrew Siegel, Brad Gallagher, and Michael Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, SC '03, pages 39–, New York, NY, USA, 2003. ACM.

[25] J. Lofstead, S. Klasky, K. Schwan, N. Podhorszki, and C. Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments, CLADE '08*, pages 15–24, New York, June 2008. ACM.

[26] Q. Meng, A. Humphrey, J. Schmidt, and M. Berzins. Investigating applications portability with the uintah dag-based runtime system on petascale supercomputers. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, Nov 2013.

[27] Bernhard Scholz, Herbert Jordan, Pavle Subotić, and Till 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.

[28] H. Shan, K. Antypas, and J. Shalf. Characterizing and predicting the I/O performance of HPC applications using a parameterized synthetic benchmark. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, Nov 2008.

[29] Pavle Subotić, Herbert Jordan, Lijun Chang, Alan Fekete, and Bernhard Scholz. Automatic index selection for large-scale datalog computation. *Proc. VLDB Endow.*, 12(2):141–153, October 2018.

[30] The HDF Group. Hierarchical Data Format, version 5, 1997-NNNN. http://www.hdfgroup.org/HDF5/.