Provenance Databases for Workflow Systems

Jennifer A. Steffens
Drake University
jennifer.steffens@drake.edu

Justin M. Wozniak [Advisor]
Argonne National Laboratory
wozniak@mcs.anl.gov

***Abstract*--In scientific computing, understanding the origins and derivation of data is crucial. Provenance models aim to provide a means of capturing this in an efficient and effective manner. For the Swift/T language, the current provenance handling system requires improvement. In this paper, we discuss the development of a new Swift/T provenance model based on the Swift/K provenance model, which would parallelize the real-time storage of provenance logs in a user-accessible database system. Utilizing multiple databases in high performance, parallel workflows can increase the practicality of lightweight, relational databases engines such as SQLite, and we show it to be more efficient than a single database.**

I. THE SWIFT SYSTEM

The Swift scripting language is designed for optimizing the execution of scientific computational experiments by performing independent tasks implicitly in parallel. The system operates on a given number of nodes, with one server node and many script-executing worker nodes. Swift/T, the current implementation of the language, utilizes MPI and ADLB libraries in its runtime, Turbine, letting it perform up to 1.5 billion tasks per second [1].

The provenance model for the previous generation of the Swift language, Swift/K, is based on the Open Provenance Model. After the execution of parallel scripts that specify many-task computations, this model extracts provenance information from the logs that Swift/K generates and stores it in a relational database, using SQLite, a lightweight and easy-to-use database engine, to process the information [2].

II. OUR APPROACH

Instead of collecting provenance information after a program finishes its execution, we collect it during runtime, providing the benefit of access before a large workflow finishes executing. This makes our model useful in tracking progress. To achieve this, we integrated a SQLite-utilizing C program into the Swift/T source code that inserts information into the relational databases as it is processed. In this system, the two largest and data-intensive tables are **ApplicationExecution**, which details external application calls (leaf tasks) in a Swift script, and **ScriptRun**, which details important general information [Figure 1]. These are both modified versions of the tables found in the Swift/K provenance system [3].

ApplicationExecution	
tries	int
startTime	datetime
try_duration	int
total_duration	int
command	char (128)
stdios	char (128)
arguments	char (128)
notes	text
tries	int

ScriptRun	
scriptRunId	int
scriptFileName	datetime
logFileName	int
swiftVersion	int
turbineVersion	char (128)
finalState	char (128)
startTime	char (128)
duration	char (128)
scriptHash	text
scriptRunId	int

Figure 1: Schema of ApplicationExecution and ScriptRun

Because Swift/T parallelizes its programs through the use of workers who execute asynchronous tasks in parallel, we assigned each worker its own database, which is schematically identical to the master database. By doing this, we can make sure each database is not being written to by multiple processes simultaneously. To query the data, we use attach statements to join all of the separate database files, eliminating the need to combine them into a single file. We perform this query upon the master database, which houses the general data for the script run (this is only inserted by the server node, and so does not need to be parallelized). By attaching the other database files to the master database, we can count this transaction as well as the view statements as one transaction, and therefore there will be no simultaneous writes to worker databases.

III. EVALUATION

To test the efficiency of our system, we ran a simple Swift script on a variable amount of nodes hosted by the Cooley computing system, a collection of 126 compute nodes housed at Argonne National Laboratory. Our script generated a hundred tasks per worker, and assigned them each their own database. The script was simply to echo integers from one to the number of workers executing the task multiplied by one hundred. We compared this to the same script ran with a single database, regardless of how many worker nodes participated in the execution. We also evaluated the script on multiple databases when they were pre-populated, i.e. they did not require the

Worker Nodes	M + 1 PPN + POP	M + 12 PPN + POP	M + 1 PPN + NO POP	S + POP
2	80.667	111.669	16.785	2.2172
4	111.592	228.506	106.8	2.594
8	128.673	432.654	109.27	4.056
16	138.925	173.461	122.822	4.506
32	133.662	163.913	105.351	1.085
64	123.193	126.203	97.207	0.5926
100	128.084	131.73	88.235	

Figure 2: A comparison of a single populated database, multiple populated databases, multiple unpopulated databases, and multiple populated databases with twelve processes per node.

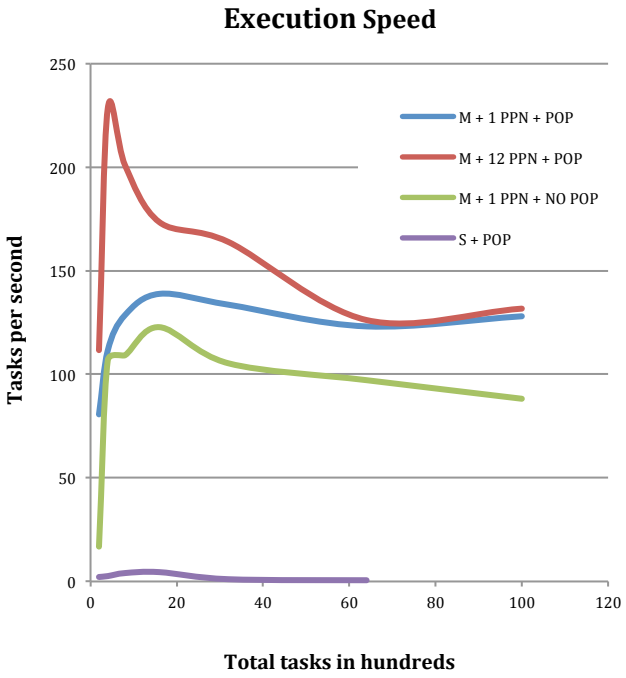

Figure 3: A visualization of the speed comparisons

execution of a schema creation prior and instead relied on a copy-paste to move them into each output file, and when they were prepopulated and had twelve processes per node, e.g. the eight-worker test was performed on a single node and the thirty-two-worker test was performed on three nodes [Figure 2].

We found a significant increase in performance for multiple databases, and immediate scaling. At a large scale, the increase in speed slows, however still out- performs the single database system. It was seen that prepopulated databases have an advantage as the create schema statements cause the non-populated database speed to slow as it approaches more than one hundred nodes. In contrast, the prepopulated databases have end behavior that implies a trend of increase. In addition, giving the program more than one process per node is beneficial until the number of processes exceeds six thousand; after, the speed is very close to the single-process-per-node test [Figure 3].

IV. CONCLUSION

We believe parallelizing databases in this fashion will make simple database engines practical for high performance computing. By making sure each file is only written to by one process, we can decrease the time needed for data storage and therefore increase the efficiency of the system. For the Swift/T language, this provenance storage model offers easy storage and access to valuable data collected, available as soon as it is processed.

ACKNOWLEDGMENTS

This work was supported in part by the National Science Foundation under awards NSF-1461260 (REU) and used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE- AC02-06CH11357.

REFERENCES

[1] Wozniak, Justin M., Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, and Ian T. Foster. "Swift/T: large-scale application composition via distributed-memory dataflow processing." In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pp. 95-102. IEEE, 2013.

[2] Gadelha, L. M., B. Clifford, M. Mattoso, M. Wilde, and I. Foster. *Provenance management in Swift with implementation details*. No. ANL/MCS-TM-311. Argonne National Laboratory (ANL), 2011.

[3] Gadelha Jr, Luiz MR, Michael Wilde, Marta Mattoso, and Ian Foster. "MTCProv: a practical provenance query framework for many-task scientific computing." *Distributed and Parallel Databases* 30, no. 5-6 (2012): 351-370.