Paper Review Writeup: Swift/T

CS 554: Data Intensive Computing

Reviewer Name: Nilesh Balu Jorwar (A20405042)

Paper Title: Swift/T: Large-Scale Application Composition via Distributed-Memory Dataflow Processing

Paper Authors: Justin M. Wozniak, Timothy G. Armstrong, Michael Wilde, Daniel S. Katz, Ewing Lusk, Ian T. Foster

Publication Venue: 2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing

Year of Publication: 2013

Summary

Today, high performance computing has become increasingly important and critical to organizations of all sizes and in a wide range of industries seeking to process massive amounts of data that requires compute power of supporting sophisticated engineering applications.

Authors of this paper presented a Swift/T, a new implementation of Swift language for HPC. Swift/T addressed the various problems in distributed processing systems like load balancing, concurrent programming, data distribution, scalability, data movement, parallelism.

Authors organized the paper by introducing Swift/T and how it is benefitted to large scale systems. Swift/T programming model described how to code in Swift to achieve scalability, implicit parallelism and other issues related to high computing processing.

The proposed Swift/T has distributed dataflow engine as a part of technical innovation for dataflow-driven task execution and distributed data store for global data access. The Swift/T compiler translates the user script into a fully scalable, decentralized MPI program through the use of enabling libraries. The programming model of Swift/T extends with file-based data passing to finer-grained applications using in-memory functions and in-memory data while addressing the intertwined programmability, scalability and concurrency.

Swift/T architecture has two parts Swift Turbine Compiler (STC) and Turbine scalable runtime where STC is used to compile the Swift script to Turbine code to be launched as MPI program with Turbine runtime system. Swift front end of STC parses Swift program, typed checked it, performed data analysis and generated Swift IC intermediate representation that is broken down to individual Turbine operations. Turbine programs are MPI programs that use ADLB and turbine libraries and has the compiler that breaks the code into fragments (tasks) to be executed whereas Distributed future store is used to pass the data and to track data dependencies between tasks. Scalable Swift runtime enables concurrency and determinism.

The performance of Swift/T is conducted on IBM Blue Gene/P and Blue Gene/Q systems while considering three application patterns, non-trivial real application.

Core Contributions

- Addressed characteristics of many task applications, described related work and Swift/T programming model, and presented performance results.
- Implementation of Swift with new compiler and runtime called Swift/T.
- Removes single-node evaluation bottleneck and enables Swift programs to execute with far greater scalability.
- Maximized concurrency through dataflow-driven task execution which came from dynamic evaluation of programs written in concurrent, expressive language.
- Avoided writing a massive concurrent program in MPI directly, rather user code "extensions" coded in a native programming language that has native types (such as integers, floats, pointers to byte data, etc.) and executed by the runtime.
- Development of distributed future store (drives data-independent execution and allows typed data operations) in decentralized manner by STC optimizing compiler.
- Analyzing graphs of collaboration between scientists to make control flow decisions and force frequent load balancing.
- Providing a system to run at extreme scale when given code by non-experts.
- Achieved implicit parallelism by executing each Swift statement concurrently with other statements.

Difference with related work

Earlier implementations of the Swift were Swift/K which runs on Karajan grid workflow engine to execute a workflow of program executions across wide area resources. Its libraries exceled at exploiting diverse schedulers and data transfer technologies. Additionally, Swift code was run using multithreaded programming on on single centralized node to coordinate external tasks running on additional nodes. Swift had maximum task dispatch rate of < 500 tasks per second, and its available memory was limited to that of a single node.

Dataflow programming models implemented by Tarragon and DaGue has efficient parallel execution of explicit workflow DAGs of tasks from within an MPI program. explicit dataflow DAGs of tasks from within an MPI program. TIDeFlow proposed a dynamic dataflow execution model while FOX uses dataflow graphs for fault tolerance.

However, Swift/T programming model focused on task-parallel applications with the use of asynchronous dynamic load balancer MPI library for distribution of tasks to achieve scalability. Swift/T did not use Socio (a library for distributed memory dynamic load balancing of tasks) as it did not provide features such as task priorities, work types.

Pros

- Swift/T provides enhanced performance by executing billion tasks per second
- It has ability to call native code functions (C, C++, Fortran)
- Execute scripts in embedded interpreters (Python, R)
- Removed single node evaluation bottleneck and exceeds the scalability of the system beyond that of a task management system based on a single node.
- Swift/T is well suited for “many task” applications.
- Dataflow scripting provides a powerful mechanism for high performance components enabling fault tolerance, dynamic load balancing and parallelism.
- Swift/T parallel scripting language allows developers to write C-like functions and expressions using high-level data structures (such as associative arrays).

Cons

- Swift/T relies on a global file system shared by each worker node to perform file I/O operations, though it reduces the complexity of developing compute-intensive applications. [presented in paper by Szalay and Blakeley [19], Dodson et al. [20], and Zhao et al.]
- Challenging to scale up in commonly used message-passing programming models.
- Swift/T is many task computing language making external code callable from Swift is crucial.
- Measures are not taken to handle caching and consistency hence hindering Swift/T performance.
- Swift/T does not make attempt generate efficient sequential code and aggregate data types do not have first class concept of partitioning.

Improvisation for Author

Implementation of garbage collection to support long running jobs resulting more global data to be garbage collected. Alternate solutions to load balancing methods need to implement to achieve many fold performance.

Future Work

Exploitation of dataflow driven task-parallel execution model need to be done for fault tolerance and power awareness at the runtime system level. Integration of Swift/T with file system to support efficient workflow-aware file access.