Distributed Garbage Collection at Extreme Scale in Swift/T

Swift/T Background

Swift is a programming language for large-scale parallel applications that execute on distributed computing resources including clouds, clusters, grids, and supercomputers. These programs are constructed through data flow composition of command-line programs, serial functions, and functions with internal message-passing or thread parallelism.

Swift/T is a highly scalable, fully decentralized implementation of the Swift language in which all aspects of program execution are distributed across all nodes of a distributed-memory system.

Swift/T Reference Counting Implementation

Reference counting in Swift/T requires us to track reference to data spanning many nodes in a distributed memory cluster. We track read and write references from running or waiting tasks to globally shared data and from globally shared data to other data.

Data model with reference counts, illustrating reference ownership by tasks and data and caching of extra reference counts within data

Compiler support is required to distinguish between read and write references, and to correctly manage reference counts by incrementing and decrementing them at points in the program where references are copied or lost. The Swift/T compiler uses an internal representation (IR) tree to represent the program being compiled. Reference counting is added to the IR tree as a post-processing step immediately before code is generated. An intelligent placement algorithm does multiple passes over the IR tree:

1. annotate tree with usage: which subtree of code need read or write references
2. collect read/write increment/decrement counts per variable per IR code block
3. place increments and decrements in IR code block with several optimization strategies

Merging and Cancelling

If a variable's reference count is modified multiple times in a block, we may be able to merge multiple operations into one or cancel out increments and decrements. Safely doing so requires analysis to determine that a reference count will not inadvertently drop to zero as a result of the change. In the case of statically determined data dependency DAGs, these optimizations allow all increments to be determined at compilation time, so reference count increments can be avoided by pre-inserting reference counts to the correct value.

Piggybacking

The main cost of global data operations is communication, particularly message round-trips. Appending additional data to these messages is very cheap, so Swift/T runtime data operations support "piggybacked" reference counts. The compiler can piggyback decrements if it identifies the last read or write to the reference. This is often possible, with exceptions such as certain conditional statements. For static data dependency DAGs, this eliminates all decrements.

Distributed Garbage Collection in Swift/T

The Swift programming language requires garbage collection for accurate automatic memory management. Swift also requires automatic detection of finalized variables, which have reached their final state and will not be further modified, a related problem.

Garbage collection in distributed-memory programs requires accurate tracking of references shared between many concurrent processes. This is challenging to do efficiently; large-scale applications create millions or even billions of data flow variables per second. Simple algorithms require many bookkeeping messages, often more than doubling the communication cost of an application.

Previous work has shown that a combination of compile-time and runtime techniques can enable efficient reference counting garbage collection in a smaller-scale shared memory context. For comparison, large-scale distributed-memory data flow languages present distinct challenges and opportunities: without shared memory, messages must be exchanged for every reference count modification. Even on high-performance networks, the latency of a message round-trip is multiple orders of magnitude longer than a memory access, and may require the cooperation of the other process.

Results

Five benchmark applications were used, representing a range of common parallel patterns. Various configurations were evaluated:

- ADLB: hand-coded C using the ADLB load balancer
- SWIFT/T with n optimizations (naive compilation)
- O1: basic optimizations, including reference counting optimizations
- O2/O3: advanced optimizations (e.g., task graph rearrangement)
- Disabled: O3 with memory management disabled
- Unopt: O3 with reference counting optimizations disabled

Conclusion

- For many common patterns, the communication cost of reference counting is provably reduced to zero or a small fraction of total communication.
- The overhead of automatic memory management is acceptable for all applications tested, proving the practicality of distributed reference counting.
- We can scale Swift/T’s automatic memory management to 100,000’s of cores: a critical stepping stone towards our goal of a scalable, high-performance programming model for implicitly parallel dataflow.