Compiler optimization for massively parallel data flow

Tim Armstrong, Justin M. Wozniak, Michael Wilde, Ian T. Foster
TA, MW, IF: University of Chicago  JW, MW, IF: Argonne National Laboratory

Abstract
Distributed, dynamic data flow is an execution model well-suited for many large-scale parallel applications, particularly scientific simulations and analysis pipelines running on large, distributed-memory clusters. Swift is a high-level declarative language that allows flexible data flow composition of functions written in other programming languages such as C or Fortran.

Swift/T is a high performance, scalable re-implementation of Swift for massively distributed memory clusters.

This poster focuses on one aspect of the implementation: compiler optimization to improve execution efficiency and scalability. We show that compiler optimization can reduce communication overhead by 70-93% on distributed memory systems at scales up to thousands of cores. With compiler optimization, the high-level Swift language becomes competitive with hand-coded coordination logic for certain common styles of computationally intensive applications.

Optimization Techniques

Standard compiler optimizations: constant folding, common subexpression elimination, dead code elimination, function inlining, loop invariant hoisting, etc.

Optimizations for asynchnronous data flow: detection of finalized variables, task spawn deferral, task merging, etc.

Optimizations for reference counting: cancel and combine reference count operations; piggy-back reference counts on other messages; batch reference counts for loops. Reference counting is used in both memory management and for detecting when data structures are finalized. Optimization is critical to avoid high overhead.

Experimental Setup

Five benchmark applications were used:

Sweep: a parameter sweep with two nested loops and completely independent tasks

Fibonacci: a synthetic divide-and-conquer application with the same dependency graph as a recursive Fibonacci calculation

Sudoku: a divide-and-conquer Sudoku solver that recursively prunes and divides the solution space, and terminates when solution found

Wavefront: 2D array with each cell a function of three neighbors

Simulated Annealing: an iterative optimization algorithm with a parallelized objective function

Four implementations were compared.

ADLB: hand-coded C using Swift’s distributed runtime library directly

O0: Absolutely no optimizations: naive compiled Swift/T

O1: Basic compiler optimizations: e.g. constant propagation

O2: More sophisticated optimizations: e.g. loop invariant hoisting

O3: Aggressive optimization: e.g. function inlining, loop unrolling

We measure a) number of task and data operations (a good proxy for total communication) and b) application time-to-solution.

Results

- Sweep: 10^5 tasks
- Fibonacci: n=24 0.2 ms tasks
- Sudoku: 100x100 board
- Wavefront: 100x100 0.2 ms tasks

Compiler optimization reduces the number of runtime data and task operations and increases throughput. The benefit varies: basic optimizations always help, advanced optimizations help greatly with complex dataflow.

Compiler optimization is essential for applications with short-running tasks and at large scale.

Further reading