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 implementation of the Swift language that distributes the execution of all aspects of a scripts across all nodes of a distributed-memory system up to scales of hundreds of thousands of cores. Swift/T has been in development at U.Chicago and Argonne since 2011.

Swift/T distributes script execution, tasks, data, and data dependency tracking across many worker and server processes:

Swift/T Compilation and Execution

Compiler Optimization

Compiling a high-level programming language to efficiently use this distributed runtime is challenging: translating Swift directly leads to excessive synchronization and communication between processes. We have implemented a full-fledged optimizing compiler for Swift that uses a custom intermediate representation (IR) with dataflow, distributed memory, and task-parallelism as first-class citizens:

We have implemented a large set of optimizations for our IR, including adaptations of classic compiler optimizations (value numbering, constant folding, dead code elimination, etc.), plus optimizations for task-parallelism, dataflow, and reference counting garbage collection. We will sketch a few select optimizations. Consider this Swift code:

```swift
let a = f(1);
let e = f4(f5(e));
let c = f3(a, b);
```

Naive compilation uses a task (blue circle) per function call and global data item (grey square) per variable:

```
ancestor task
```

Task graph rearrangement optimizations (specifically, wait pushdown) has tasks that write data spawn tasks that read that data, with data bypassing global data storage (reducing communication greatly):

```
```

Reference Counting Optimizations

Reference count operations in Swift/T can require a message to be sent to a remote process to update a reference count. We use static analysis techniques to avoid unnecessary reference count operations. For example, reference count increments are avoidable if the compiler determines the number of reads/writes to a data item:

Decrments can be “piggy-backed” on other messages if the compiler identifies the last read/write (shown in red) to a data:

Results

Five benchmark applications represent several common parallel patterns:

Sweep: parameter sweep (independent tasks)
UTS: Unbalanced Tree Search benchmark
ReduceTree: Tree-structured reduction
Wavefront: 2D wavefront pattern

Annealing: simulated annealing optimization algorithm (~500 Swift ~2000 C++ code lines)

Five configurations were evaluated:

- ADBL: hand-coded C using the ADBL load balancer
- 00: Swift/T with no optimizations
- 01: basic optimizations
- 02/03: advanced optimizations (e.g. task graph rearrangement)

Compiler optimizations reduce the # of runtime interprocess task and data operations required to run each benchmark:

This reduces communication and synchronization, leading to dramatically improved performance and scalability, often comparable to hand-coded applications:

Conclusions

- Swift/T is a simple and elegant parallel dataflow language that can scale to 100,000s of cores
- Compiler optimization has proven to be essential in achieving this scalability
- We have designed and implemented a range of novel compiler optimizations for distributed-memory task-parallelism with data flow dependencies

This research is supported by the U.S. DOE Office of Science under contract DE-AC02-06CH11357 and NSF award 1444443. Computing resources provided in part by NERSC through the Computation Institute and the University of Chicago BSD and Argonne National Laboratory. Under grant S15 RR029833-01A1 and by NSF award 1239693 and the state of Illinois through the Blue Waters sustained-petascale computing project.