Exploring Scientific Discovery with Large-Scale Parallel Scripting

Tim Armstrong\textsuperscript{1}  Justin M. Wozniak\textsuperscript{2}
Michael Wilde\textsuperscript{12}

\textsuperscript{1}University of Chicago
\textsuperscript{2}Argonne National Laboratory

May 15, 2013
Overview

Parallel scripting: massive scalability with (relative) ease

- Scaling up real science applications difficult:
  - Must adapt code to radically different programming model
  - Concurrency bugs
  - Load balancing, data management, etc
- SciColSim: compute-intensive science app
- Swift/T: super-scalable high-performance scripting system for parallel composition of existing code
The Scripting Paradigm

- Low-level language (e.g. C) + high-level language (e.g. Python)

High-level script orchestrates

Optimized performance-critical functions

Python

R

Swift

TM
Parallel Scripting

- Can retrofit parallelism onto sequential scripting languages:
  - Threads
  - Message passing (MPI, etc.)
  - Abstractions (MapReduce, etc.)
- But parallelism is a second-class concept in the language...
- Q: Why can’t I express parallelism with loops, conditionals, variables, etc?
Parallel Scripting in Swift/T

- Q: Why can’t I express parallelism with loops, conditionals, variables?
- A: you can in Swift!
- The Swift parallel scripting language[WFI+09]:
  - Implicit dataflow parallelism
  - Language statements execute concurrently in dataflow order
  - Single-assignment variables guarantee determinism
  - Determinism extends to additional, rich, data structures: arrays, hash tables, structs.

```swift
float results[];
file data = input_file("my.data");
foreach i in [1:N] {  // Independent parallel iterations
    if (predicate(i)) {
        results[i] = compute(i, data);
    }
}
mean, stdev = stat_summ(results);
```

Swift code with implied parallel dataflow
Swift/T Scalable Implementation

- Can harness tens or hundreds of thousands of cores
- All runtime components distributed and scalable: data store, task distributor & script executor
- Optimizing compiler (stc) reduces messaging

Swift/T runtime services breakdown (left) and task dispatch (right)
SciColSim Application: Simulating Scientific Discovery

- Ongoing research at University of Chicago
- Want to understand process of scientific discovery: [ER10]
  - How do scientists select hypotheses to work on?
  - What are the most effective strategies?
- Can explore with simulation:
  - Model knowledge as graph of concepts
  - Simulate different graph exploration strategies
  - Can measure how “efficient” strategy is
- Computational characteristics:
  - Each simulation implemented with sequential C++ code
  - Floating point intensive: many probability calculations
Evaluating model parameters

- “Ensemble” of randomized simulations
- Results of simulation are averaged to evaluate “goodness” of current parameters
- Task duration is 0.2-20s. Runtime depends on input parameters, plus significant random variation.

Evaluating objective function and updating parameters
Simulated Annealing

- Want to find “best” set of simulation parameters
- Optimize using a simulated annealing algorithm
- Basic idea:
  1. Perturb one parameter
  2. Evaluate objective function for current parameters
  3. Depending on result, maybe undo parameter change
  4. Repeat...

Visualization of parallel simulated annealing with 8-way parallelized objective function. Real runs have 1000-way parallelism.
Scale-up requirements

- Optimization + validation: 0.25–0.5M CPU-hours per model
- Fast feedback needed: scientists want to iterate models
- Need to get high speedup: 4000×+ to get timely results
- Relatively short-lived tasks: 0.2s-20s. Fan-out and fan-in every 1-2 minutes.
- Unpredictable task duration: need to dynamically assign tasks to processors, in scalable way
- High-performance dynamic task allocation mandatory

...
Adapting for Swift/T

- Kept compute-intensive simulation logic in C++
- Converted simulated annealing algorithm to Swift/T:
  - Nested parallel loops
  - Sequential iteration
  - Logic and formulas to update parameters
  - Logging and output

<table>
<thead>
<tr>
<th></th>
<th>Original</th>
<th>Swift/T Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Lines of Code</td>
<td>Python: 33 lines</td>
<td>Swift/T: 269 lines</td>
</tr>
<tr>
<td></td>
<td>C++: 1175 lines</td>
<td>C++: 861 lines</td>
</tr>
<tr>
<td>Scalability</td>
<td>One node, many cores</td>
<td>Many cores, 100’s or 1000’s of nodes</td>
</tr>
</tbody>
</table>
Scaling up!

- Strong scaling results for production workload at different compiler optimization levels: scales well!
- Mainly limited by amount of parallelism in workload ⇒ could scale further with different optimization algorithm
- STC compiler optimization: reduces messaging ⇒ better scaling

Strong scaling for down-scaled at different STC optimization levels (left) and full-scale problem (right)
Task Prioritization

- Key technique enabled by Swift/T: *task prioritization*
- Improves resource utilization and time-to-solution
- Exploits application knowledge:
  - “Catch-up” heuristic for slower optimization chains
  - Prioritize long-running tasks: target parameter correlated with runtime

@prio = 100*(nites - iter) + target
run_simulation(...);

![Graph showing comparison between 'without priorities' and 'with priorities' in terms of busy cores over time.](image)
References


**Acknowledgements** This research is supported in part by the U.S. DOE Office of Science under contract DE-AC02-06CH11357, FWP-57810. This research was supported in part by NIH through resources provided by the Computation Institute and the Biological Sciences Division of the University of Chicago and Argonne National Laboratory, under grant S10 RR029030-01.
Demo

- Compile application from scratch to illustrate toolchain
- Production-scale run of SciColSim on 8400 cores of Beagle Cray XE6 supercomputer @ UChicago
Conclusions

- Can scale up existing applications with parallel scripting
- Quick development cycle: easy to debug and modify code, compared with alternative cluster programming models
- Appropriate for applications that can be implemented as user-defined tasks with explicit data dependences
- Much better for moderately fine-grained workloads on large clusters than traditional centralized workflow systems
- Does not support wide-area grids/clouds (yet)
Task Dispatch Speed

- Cray XE6
- On 10 nodes, 24 cores per node
- Many independent 0s tasks
Scaling up to $10^5$

- Experiment on Blue Gene/P Intrepid at Argonne National Lab
- 100s task durations
- Experiment used old version of Swift/T. Many improvements since.
Optimizations

- **O0**: Only optimize write reference counts.
- **O1**: Basic optimizations: constant folding, dead code elimination, forward data flow, and loop fusion.
- **O2**: More aggressive optimizations: asynchronous op expansion, wait coalescing, hoisting, and small loop expansion.
- **O3**: All optimizations: function inlining, pipeline fusion, loop unrolling, intra-block instruction reordering, and simple algebra.
Comparison with Other Systems

- Hadoop
  - Fixed communication pattern
  - Must reorganize code to fit MapReduce model
  - Minimize per data-item overhead versus minimize per task-overhead

- Swift and other workflow systems
  - Single master node limits scalability
  - Optimizing compiler
  - Better foreign-function interface for directly calling C++ code
  - No support yet for wide-area systems
Comparison with PGAS

- Scripting paradigm versus one language for computation + coordination
- Focus on simplicity
- No explicit data placement: managed by runtime
- Strong safety guarantees (e.g., determinism)