A Scripting Interface for Swift/T Parallel Workloads Using Messaging Queues

[1]Michael Collins, [2,3]Justin M. Wozniak

[1] Department of Electrical and Computer Engineering, Rutgers University, New Brunswick, NJ, USA
[2] Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
[3] Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

michael.collins@rutgers.edu, wozniak@mcs.anl.gov

***Abstract*—Modern scientific computing applications require not only highly parallel high-performance computing workloads but also the expressiveness and simplicity of scripting languages. In this work we present an interface that allows external C and C++ programs to control a parallel workflow using Swift/T. This interface facilitates the use of existing C/C++ algorithm implementations to run distributed programming models in an inversion-of-control fashion without the requisite knowledge of scheduling and message passing. The implemented framework uses a messaging queue interface that requires minimal modification to existing libraries and applications. We evaluate this solution using a genetic algorithm in C++ to perform an adaptive parameter search on an agent-based model. We perform this evaluation on large-scale distributed computing machines and compare system utilization and computational runtime with existing frameworks.**

I. Introduction

Scientific computing applications are becoming increasingly focused on scripting languages. As seen in the growing popularity of libraries such as NumPy and SciPy, as well as computational scripting languages like R and Julia, the scientific community is favoring scripting languages for their expressiveness and relative ease of use. At the same time, advances in high-performance and distributed computing have allowed scientific computing applications to take advantage of parallel workloads through the use of distributed message passing protocols like MPI. The development of a programming model that fully appreciates these two trends, however, is not feasible for a scientist who lacks a background in parallel programming. Such a model would require a core library of C, C++, or Fortran code for the bulk of data processing, additional modules utilizing OpenMP or MPI for distributing the workload over multiple threads and/or processors, and wrapper scripts in a scripting language like Python to handle the application logic and abstract the lower-level code base.

To streamline this development process, the Swift programming language [1] provides an implicitly parallel framework for controlling native code tools and applications using wrapper scripts. The latest implementation of Swift, called Swift/T [2], can generate an MPI program from a given Swift script and supports direct calls to C, C++, and Fortran code, as well as calls to scripts written in Python, R, and Tcl [3].

In this work, we present an extension of Swift/T's external language interface to allow C and C++ programs to govern the workflow of a Swift programming model. The target use case is a programming model whereby a model-exploration algorithm controls simulations of an agent-based model through Swift/T. With the current framework, the user can leverage existing third-party model-exploration libraries such as Evolving Objects and GAlib.

The remainder of this paper is organized as follows. In §II, we provide background on Swift/T, agent-based modeling and simulation, and recent work in ABMS using Swift. In §III, we describe the design of our framework and its current implementation. In §IV, we evaluate the framework and report results. In §V, we conclude and present future work.

II. Background

A. Swift/T Parallel Scripting Language

Swift/T [2] is a programming language and runtime environment designed to facilitate the development of highly concurrent, task-parallel applications. The Swift/T compiler is able to convert a user's script into a fully scalable MPI program, and its runtime library Turbine handles the distribution of processes. The Turbine runtime, aided by the Asynchronous Dynamic Load Balancer (ADLB) library, has demonstrated scaling capabilities up to 128K processing cores while maintaining over 80% system utilization for independent tasks.

Recent modifications to ADLB [4] have allowed Swift/T to leverage MPI 3.0 group-collective communicators in

order to distribute multiple parallel tasks (i.e. tasks that each utilize multiple processes). Traditionally, MPI creates a sub-communicator for each task which functions as a group within the program's world communicator, resulting in a tree-like structure of groups that is collective on the parent communicator. MPI 3's group-collective communicator functionality [5] allows for a more asynchronous approach to group formation, whereby the creation of groups is collective only over the processes contained in the new communicator. As a result, groups can be allocated dynamically. This opens up new capabilities for Swift of running parallel instances of external parallel applications. We utilize this functionality in our evaluation of the queue interface.

B. Agent-Based Modeling and Simulation

Agent-based modeling and simulation (ABMS) [6] is a method of predicting the interactions and behavioral consequences of individuals in a system of many decision-makers. Each agent has the ability to make decisions that impact their own behavior as well as the behaviors of surrounding agents. This type of model is useful in simulating epidemic spread, population dynamics, social networking, business relations, and other complex systems.

AMBS can be used to analyze the sensitivity of a population of individuals to certain stimuli and can help lead to wiser decision-making strategies in the real world. A common application of ABMS is an adaptive parameter search, whereby a model-exploration algorithm (e.g. evolutionary algorithm, active learning algorithm, neural network) runs many simulations in order to determine optimal parameter sets for reaching a targeted outcome. In our evaluation, we use a genetic algorithm to perform an adaptive parameter search on an ABM, as this demonstrates the ability of the messaging queue framework to govern an inversion-of-control model between a C/C++ algorithm and a Swift/T program.

C. Related Work

Recent work in agent-based modeling and simulation (ABMS) has shown Swift's capability to perform a parallel workload utilizing external code libraries. As shown in [7], Swift has been used to run simulated annealing processes to optimize the model parameter space of an ABM created in Repast Simphony, an external Java toolkit. This model produced a 96.3% reduction in the total number of simulations required, compared to a total enumeration of the parameter space. Achieving these results, however, required the implementation of a custom simulated annealing algorithm design specifically for this task.

Another work from the MCS Division at Argonne National Laboratory (under review at the time of writing), uses a queuing interface with Swift/T to allow the Distributed Evolutionary Algorithms in Python (DEAP) library to perform an adaptive parameter search on a Repast Simphony ABM. This model was able to maintain an 88.95% system utilization on a 510-worker test. In this work we present a similar framework for C/C++ algorithms that leverages Swift/T's external task launching capabilities to run simulations that span multiple processes.

III. Design and Implementation

The interface uses messaging queues to facilitate communication between a standalone C/C++ program and the Swift/T framework, as shown in Figure 1. These queues govern the transfer of simulation parameters from the model-exploration algorithm to Swift for distribution among workers and results of completed simulations from Swift to the algorithm.

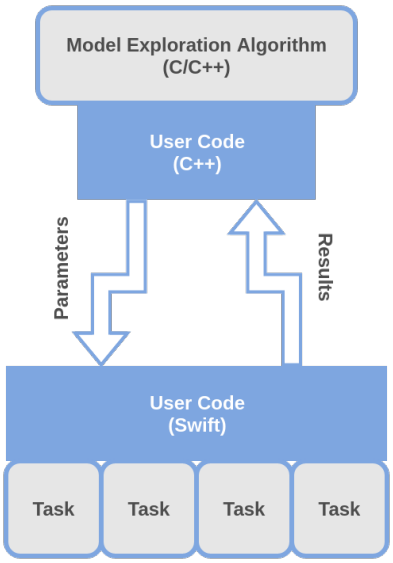

Figure 1: High-level design of queue interface

The current implementation utilizes Swift/T's Tcl extension function syntax and a SWIG wrapper script that allows the Tcl function to control both the messaging queue implementation (C++) and the user's algorithm, as shown in Figure 2. With this framework in place, the user need only produce three script files to run a distributed workflow: a C++ function that uses existing libraries to generate parameters and sends these to Swift; a Swift function that reads in parameters, launches simulations, and sends the results to the algorithm; and a Tcl package that provides the Swift extension function which launches the external simulation. None of these scripts requires knowledge of MPI from the user; only rudimentary C++ experience and the basics of Swift syntax are required.

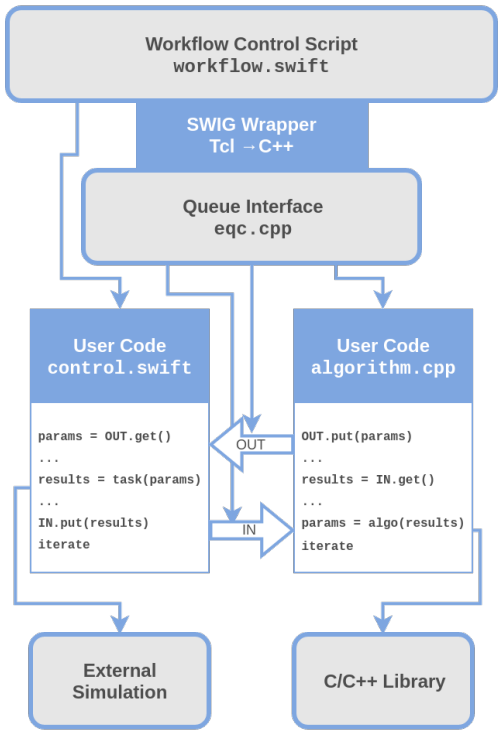

Figure 2: Implementation of queue interface

IV. Evaluation

A. *Preliminary Tests*

As a proof-of-concept, we use the following test to evaluate the feasibility of the interface.

This test uses a simple genetic algorithm written in C++ to maximize a function of two variables over a specified domain. The algorithm initializes a random selection of *(x,y)* coordinates, sends these to Swift for evaluation of the function, receives the fitnesses *f(x,y)* of these parameters, and creates a new generation of coordinates from the two with highest fitness. Each new generation is the result of a random selection from Gaussian distributions centered at the mean *x* and *y* of the two most fit parameters (i.e. the parents) with standard deviation equal to the Euclidean distance between the two parents.

The fitness of each parameter set is evaluated using a Tcl leaf function in Swift/T. Swift handles the distribution of these tasks to worker processes, collects the fitnesses of each generation, and returns the results to the genetic algorithm.

For this test, we use the following function for fitness evaluation.

$$f(x,y)=(9-x^2-y^2)\cdot(\sin(x)+\cos(y))$$

Each generation of parameters consists of 10 coordinates in the following domain.

$$D=\{(x,y) \mid -3\le x\le 3,-3\le y\le 3\}$$

The algorithm iterates until the sum of the Euclidean distances between the most fit coordinate and all others is below a given threshold. In this test, we used the threshold 0.00001.

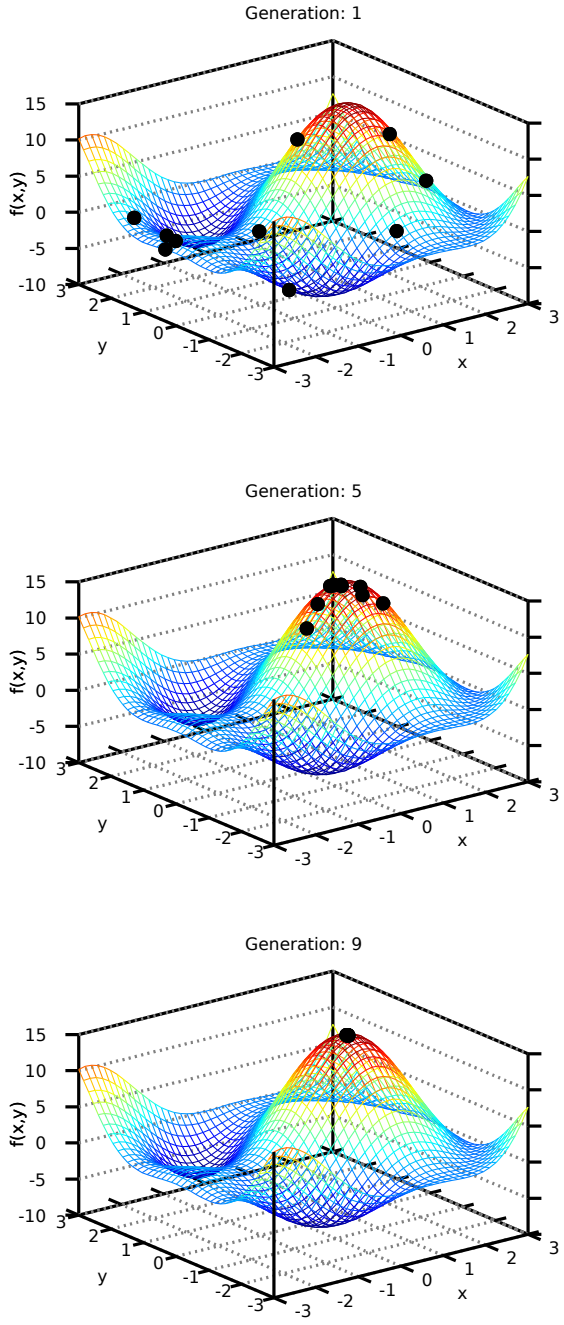

Figure 3: Maximization of a function of two variables over 9 generations.

The results of this test are shown in Figure 3. Over generations, the population of coordinates, shown as black dots, converge to a local maximum. After 28 generations, the population converged to the approximate

maximum 14.748 at the coordinate (1.05062 , 2.40316 · 10^{-7}).

This test demonstrates the ability of the queue interface to govern the transport of parameters and results between a C/C++ program and a Swift/T parallel workload. At the time of writing, this test runs successfully on the x86 cluster Cooley and the IBM Blue Gene/Q machine Cetus, both housed at Argonne National Lab. Cooley consists of 126 compute nodes, each equipped with two 2.4 GHz Intel Haswell E5-2620 v3 processors (12 cores total) and 384 GB of memory. Cetus contains 4096 compute nodes, each comprised of sixteen 1600 MHz PowerPC A2 cores and 16 GB of memory. This test has been scaled successfully up to 510 worker processes on each machine.

B. *Repast HPC Zombies Test*

The following test case is presented as an evaluation of the current implementation that utilizes Swift/T's parallel task capabilities.

A genetic algorithm developed using GAlib performs an optimization search on the parameter space of an ABM implemented with Repast HPC, a platform for distributed simulations of agent-based models.

The genetic algorithm used in this test is a modified version of the one used in the preliminary test, implemented using GAlib. It performs the same parameter search on a 3-variable parameter space.

The ABM used in this test is a modified version of the "zombies" example model included in the Repast HPC suite. The ABM consists of two agents, humans and zombies. Each can move at a speed specified by the simulation parameters *human_step* and *zombie_step* on a grid of specified size. A zombie infect humans in close proximity to it. After a given number of time units, the human will either die and become a zombie with probability specified by the parameter *lethality* or recover from infection and remain alive. The fitness of each simulation is the number of humans left alive at the end of a specified time units, and the genetic algorithm seeks to maximize this value.

This evaluation used the following parameters. The grid of the simulation is 200x200, the initial number of humans is 500, and the initial number of zombies is 50. The simulation runs for 100 time units, and an infected human will either recover or die after 14 time units.

The parameter space consists of three variables: *human_step*, *zombie_step*, and *lethality*. In this test, the two step variables are varied from 0 to 3, and *lethality* is varied from 0 to 1.

Due to time constraints and limited MPI 3 support, we are unable to run this test on the machines hosted at Argonne National Lab at this time.

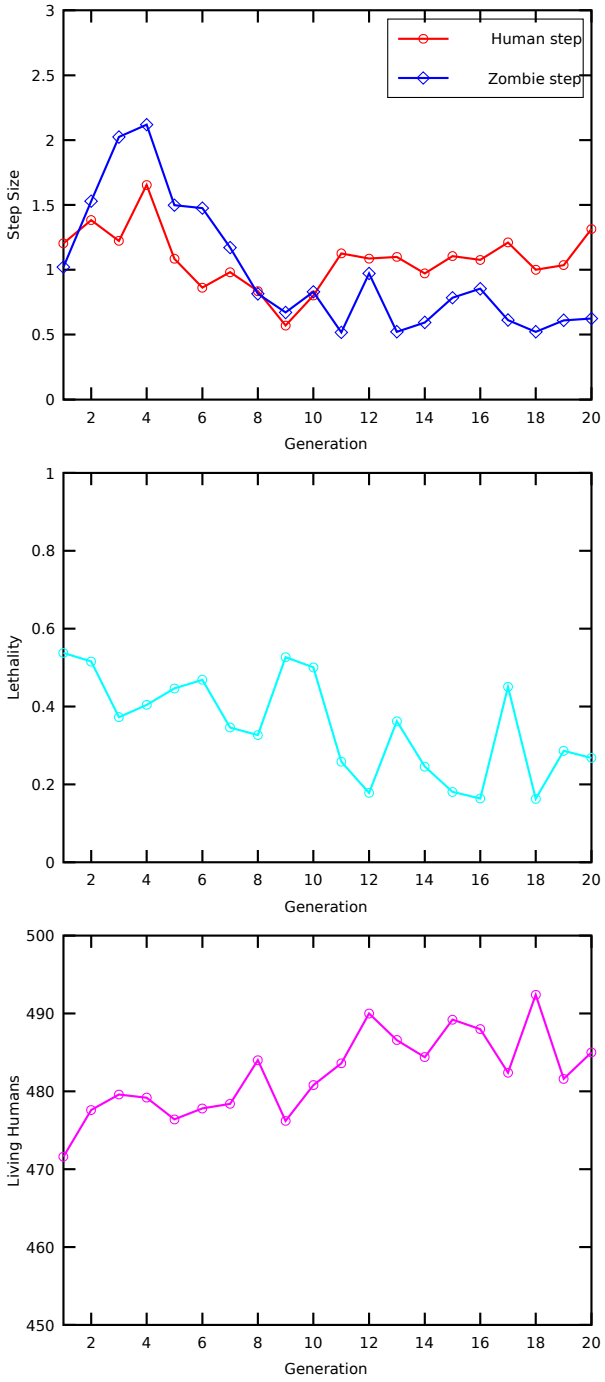

Figure 4: Adaptive parameter search of the Repast HPC Zombies agent-based model
(Average values for population size 5)

A run of this test on a local machine yields this following results for 20 generations of parameters of population size 5. While this is not the fullest extent of testing we plan to do, this test provides evidence that the framework is capable of an adaptive parameter search of an ABM.

In Figure 4, we see that the average zombie step size and lethality for a population of 5 parameter sets decreases over 20 generations. Both of these are natural results of

the algorithm's goal to maximize the number of humans alive at the end of 100 time units.

V. Conclusion

In this work, we have presented a framework for inversion-of-control programming models, allowing a C/C++ program to govern a Swift/T parallel workflow. We demonstrate the utility of this framework by performing an adaptive parameter search on an agent-based model using a genetic algorithm implemented in C++. Using this framework, users can utilize existing C and C++ libraries to control large-scale parallel ensembles of external third-party simulations.

Testing of this framework is ongoing at the time of writing. We aim to perform the Repast HPC Zombies test on larger numbers of processes, and compare this framework's system utilization and runtime with existing solutions.

VI. Acknowledgments

This work was supported in part by the National Science Foundation under awards NSF-1461260 (REU).

This research used resources of the Argonne Leadership Computing Facility at Argonne National Laboratory, which is supported by the Office of Science of the U.S. Department of Energy under contract DE-AC02-06CH11357.

VII. References

[1] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37:633–652, 2011.

[2] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/T: Scalable data flow programming for many-task applications. *Proc. CCGrid*, 2013.

[3] J. M. Wozniak, T. G. Armstrong, K. C. Maheshwari, D. S. Katz, M. Wilde, and I. T. Foster. Toward interlanguage parallel scripting for distributed-memory scientific computing. *Proc. - IEEE Int. Conf. Clust. Comput. ICCC*, vol. 2015-Octob, pp. 482–485, 2015.

[4] Wozniak, Justin M., Tom Peterka, Timothy G. Armstrong, James Dinan, Ewing Lusk, Michael Wilde, and Ian Foster. "Dataflow coordination of data-parallel tasks via MPI 3.0." *Proceedings of the 20th European MPI Users' Group Meeting*, pp. 1-6. ACM, 2013.

[5] J. Dinan, S. Krishnamoorthy, P. Balaji, J. R. Hammond, M. Krishnan, V. Tipparaju, and A. Vishnu. Noncollective communicator creation in MPI. In Recent Advances in the Message Passing Interface - 18th European MPI Users' Group Meeting, EuroMPI '11, 2011.

[6] M. J. North and C. M. Macal, Managing Business Complexity: Discovering Strategic Solutions with Agent- Based Modeling and Simulation. Oxford University Press, 1 ed., March 2007.

[7] J. Ozik, M. Wilde, N. Collier, and C. M. C. Macal. Adaptive Simulation with Repast Simphony and Swift. *Euro-Par 2014 Parallel Process.* …, pp. 418–429, 2014.