Swift/T REPL: An Interactive Shell for a Parallel Scripting Language

Basheer Subei
Dept. of Computer Science
University of Illinois at Chicago
Chicago, IL 60607, USA
bsubei2@uic.edu

Timothy G. Armstrong
Dept. of Computer Science
University of Chicago
Chicago, IL 60637, USA
tga@uchicago.edu

Justin M. Wozniak
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
wozniak@mcs.anl.gov

ABSTRACT

With the rapid technological advances in high-performance computing, access to parallel computing resources is becoming more prevalent in the scientific community. Scientific computation has been proven to be an invaluable tool for tackling challenges faced by science, but writing massively-parallel software for scientific applications remains difficult and requires years of training and practice, making the idea of writing one's own software a painful option for novices. Recently, software frameworks aimed at easing the development process, such as Swift/T, have been successfully used to allow scientists to write software in the form of high-level scripts that automatically parallelize and scale across huge high-performance computers. However, the learning curve can be further flattened by incorporating interactivity into Swift/T, specifically using a read-eval-print-loop (REPL) mode. This project investigates the validity and usefulness of adding an interactive REPL to Swift/T, as well as surveying other popular scientific computing environments and interfacing with them.

Categories and Subject Descriptors

D.3 [**Programming Languages**]: Language Classifications—*Data-flow languages*; D.2 [**Software Engineering**]: Programming Environments—*Interactive environments*; D.1 [**Programming Techniques**]: Concurrent Programming—*Parallel programming*; C.5 [**Computer System Implementation**]: Large and Medium ("Mainframe") Computers—*Super (very large) computers*

General Terms

Data-flow Language, Data-driven Task Parallelism, Implicitly Parallel Language, Interactive Environment, Interpreted Language, Literate Programming, in-situ computing

Keywords

Swift/T, Tcl, Java, ANTLR, Turbine, STC, MPI, Load Balancing, REPL, intermediate code, IPython

1. INTRODUCTION

As large scale and high-performance computing becomes more available to the industry and scientific community, it is becoming apparent that the traditional development model , specifically using message-passing frameworks such as MPI, is too difficult for non-experienced programmers. Moreover, even the most experienced programmers will find it difficult to build larger and more complex applications using traditional development methods. These problems are solved with implicitly parallel languages and frameworks such as Swift/T[4], where users write scripts in a high-level language, and where all the parallelism and required communication between processes happens implicitly and behind the scenes. Swift/T is able to scale the application in a very efficient manner to fit as many resources as the user has.

But the current development model for Swift/T can be further improved by allowing the user to interact with the runtime. There are many benefits for this in-situ approach in a data-driven scripting language such as Swift/T, as the user would be able to launch separate tasks and simultaneously monitor the data, giving them much deeper insight into the system. Moreover, extending this model to include interfaces to other scientific computing environments, such as IPython/Jupyter[3], gives the ability to visualize their work, and also more easily share their ideas via collaboration resembling "literate programming"[1] approaches by using IPython/Jupyter Notebooks[5, 6].

2. OVERVIEW OF COMPILATION AND EXECUTION MODEL

2.1 Compilation to intermediate code

The user's Swift script is processed by the Swift-to-Turbine compiler (STC)[7], which uses ANTLR[2] to generate the lexing and parsing rules in Java. These rules translate the user's scripts into language constructs, which are then used by STC to generate Turbine intermediate code (.tic), which is just Tcl code that calls Turbine runtime functions.

2.2 Turbine Runtime

Turbine is the runtime, which is a Tcl wrapper built on top of a load-balancing library that uses MPI. On startup, it spawns servers and workers each as its own MPI rank/process. The servers manage and distribute all the tasks that need to be done, while the workers perform the tasks , as shown in Figure 2 below.

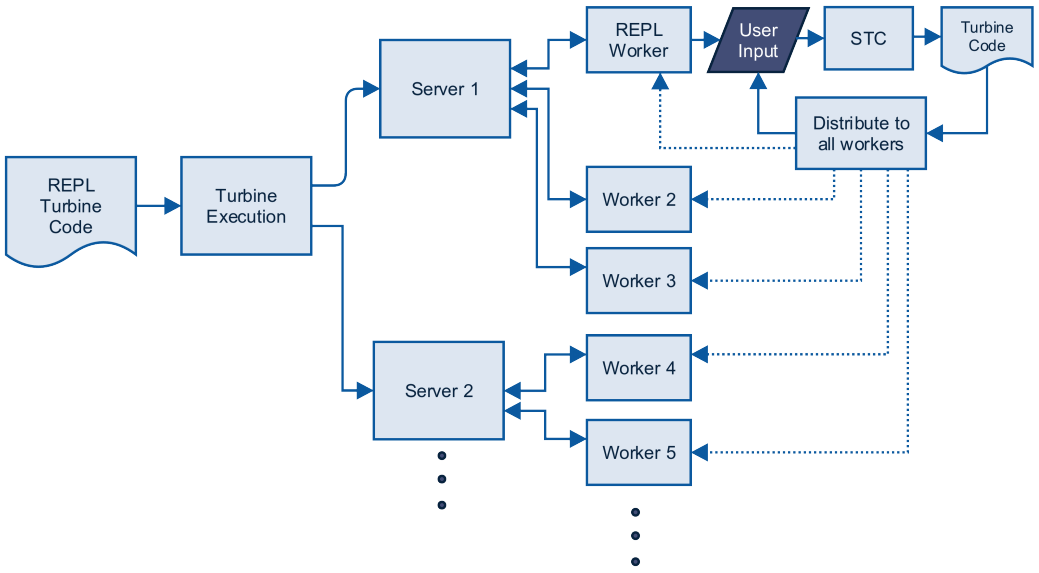

Figure 1: Overview of Initial Swift/T REPL running inside Turbine

3. SWIFT/T REPL

3.1 Features

3.1.1 Dynamic Compilation/Execution

An interactive implementation of Swift/T requires that Swift code can be compiled and executed dynamically, once the user starts a session. A straight-forward method for dynamic compilation is to simply call the compiler (STC) from shell and give it user input. Each line of code the user enters, if it's a complete statement, will be thus compiled into intermediate (.tic) code. Then, this intermediate code is simply evaluated/interpreted in the current Tcl Turbine instance. It is important to note that once the .tic code is evaluated, the appropriate tasks are spawned and any heavy-load work can then be executed in the background inside Turbine. User input returns instantaneously afterwards, allowing a user to operate in a parallel workflow dynamically.

3.1.2 Visual Output and Global Variable States

Any output from the user code (which executes in the background) is directed to standard out (STDOUT) by default, as in most applications, and allows the user to monitor the progress of their scripts. Furthermore, if the user wants to at any time probe the global variable states, they may use the "%ls" magic command to view global variable states, whether they are assigned yet or not. Since Turbine stores global variables references by unique ID, and not by names, a dictionary was created that maps global variable IDs to names, and this dictionary is filled dynamically as global variables are created.

3.1.3 Command History

A history of user-entered commands/scripts is a desired feature in any interactive environment, because it saves the user from re-typing often-used commands, and also enables the user to recall what they entered recently.

3.1.4 Code-Completion and Hints

As with any environment that is designed to help new users learn a programming language and framework, providing hints to the user for completing module/function/variable names can be very helpful. Not only does it help new users learn, but it can save a lot of time when entering code. This feature would require the REPL to parse the input, looking for language constructs and scan over already defined names. The implementation can be achieved more easily by reusing the ANTLR grammar already defined by the compiler to parse the language.

3.1.5 Syntax Highlighting

Similar to the code-completion feature, syntax highlighting can help improve the readability of the code and provide for a much friendlier user interface in the REPL. Parsing the input for this feature will be simpler than for code-completion, since all that's required is to identify the object types and keywords from what the user enters, but it will also use ANTLR for parsing.

3.1.6 Multiple Client Connection to a Single REPL

Once a REPL session is started, a user should be able to provide input from one or many clients (essentially just terminal shell sessions). Additionally, interacting with the REPL through alternate means, such as Notebooks as in IPython or Mathematica, can prove to be very helpful, and make the REPL quite versatile. The REPL was designed to keep this feature in mind, and the implementation is extensible to allow this to be achieved.

3.2 Initial Implementation of REPL

The initial implementation of the REPL mode is essentially just a program run in Swift/T, that takes user input and compiles it and executes it. Because the first worker (rank 0) is by default connected to standard input/output, the REPL resides on that first worker. On startup, there is nothing for the other ranks to do but sit and idle, waiting for the REPL rank (blocking) to receive user input.

Once the user inputs scripts/commands into the REPL, the REPL (written in Tcl) compiles the script using STC (with

special flags), and generates tic code. The global and function definitions are executed on the current rank, and then subsequently distributed to all worker ranks. Now that all the ranks are synchronized and have the same definitions, the actual tic code (main function) is executed on the REPL rank. From there, further task spawning can continue as usual, as shown in Figure 1 above. Thus, Swift code entered by the user is executed inside of an already-running Turbine instance.

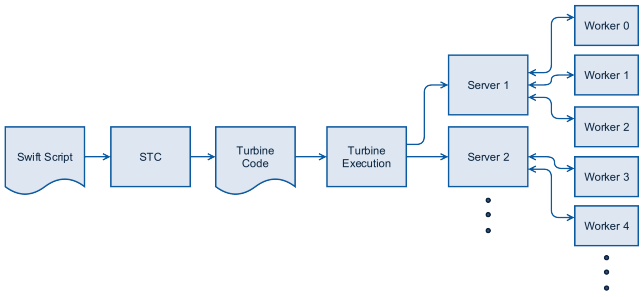

Figure 2: Overview of non-interactive Swift/T runtime

3.2.1 *Limitations*

One limitation in the Swift REPL is that redefining variables, although an important use case for interactive programming, is not currently supported. This limitation is due to the deterministic write-once nature of the language, and carelessly overwriting some variable values could lead to non-deterministic strange behavior of the program. A viable solution would be to allow users to restart certain portions of their code upon redefining dependent variables (or simply doing this automagically in the background) , as illustrated in use case #3 in the Appendix.

3.2.2 *Performance Overhead*

The only significant overhead comes from having to compile the user's script through STC, which takes a few seconds to run. However, the actual execution of the script inside the Turbine environment carries negligible overhead. We measured the average running times of Use Case #1 for both non-interactive (445ms) and REPL (581ms) versions given 40 trials each. As can be seen from the histogram in Figure 3, the runtime overhead is minimal. The overhead from the compiler (STC) has been determined to be caused by the long time it takes for the Java Virtual Machine (JVM) to start up and shut down, and the best solution is to avoid having to call STC every time the user enters input. This requires the REPL script to start up the JVM with STC idling, and then feeding it the user input as it comes through dynamically. This is a future feature that will be discussed in the last section.

4. FINALIZED DESIGN OF THE SWIFT/T REPL

4.1 IPython/Jupyter

After the initial implementation and the feature list, it was clear that a more suitable approach needed to be taken, and that writing Tcl scripts to do everything was sub-optimal. After reviewing ongoing interactive programming projects,

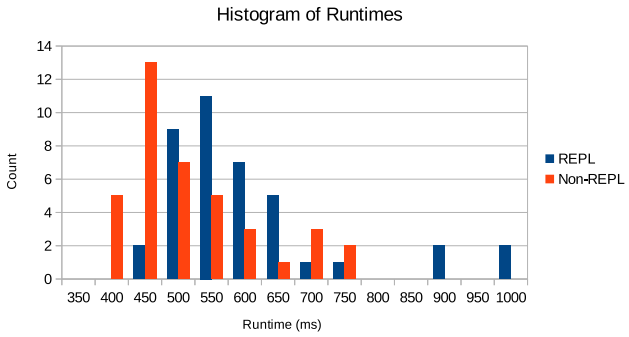

Figure 3: Histogram of runtimes for both REPL and non-REPL versions of Use Case #1.

especially Jupyter (formerly IPython), we decided it would be suitable to implement an interface to those environments. Jupyter provides clients in the form of terminal sessions and Notebooks, and it allows multiple clients to exist and connect to a single kernel (the code that processes and executes user input).

So, the new REPL design requires a Swift kernel to be written for Jupyter, allowing the Jupyter clients and kernel to communicate. For clarity, Jupyter already handles all of the work to receive user input, relay it to the kernel, and even provides hooks for advanced features, including all those we mentioned in the previous section on Features. Furthermore, this kernel would be written in Python, for which an ANTLR library exists, making the implementation of code-completion even easier.

4.2 Jupyter Kernel and REPL Communication

Once the kernel receives user input from the Jupyter client, it can process it (think of any of those fancy features), and then send the Swift script to a REPL instance for compiling/executing. However, since our solution should preferably be self-contained as a single process, we wanted to avoid needing both a Jupyter kernel process and a REPL/Turbine instance running separately and requiring to be synced.

Therefore, the first attempt to solving this problem was to use Unix pipes and subprocess to communicate between the Jupyter kernel and REPL instance. After numerous attempts, it seemed very difficult to implement in practice, because both processes were long-running and Python's pipe features were gimmicky. At any rate, an easier more elegant solution was to use sockets to communicate between the two processes, and this was not only easy in Python on the kernel side (unsurprisingly), but it was even easier to implement in Tcl on the REPL side. As a note, the socket connection had to be non-blocking, as the kernel and REPL had to alternate between sending and receiving from the socket, and the data could be multi-line or single line (which threw off Tcl at first).

4.3 Overview

To recap the overall design: a Jupyter client communicates with a kernel, which spawns a REPL instance (contain-

ing the existing code to run Turbine and dynamically compile/execute Swift script within it). The kernel and REPL communicate over a socket connection, and the REPL/Turbine instance takes care of spawning all the Turbine servers/workers and also takes care of spawning tasks to the appropriate queues. All the Turbine workers sit idle until the user enters something, which prompts the REPL worker to send off tasks, which the other Turbine workers start working on. Figure 4 shows an overview diagram of the entire process.

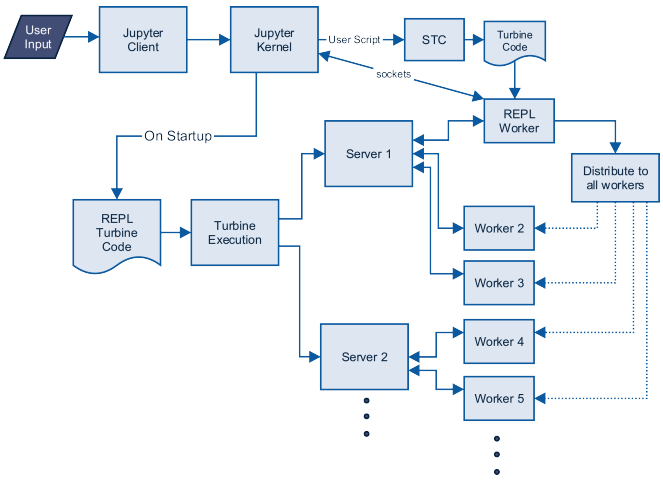

Figure 4: Overview of the Final REPL Design

5. PROJECT TIMELINE AND PROGRESS

The core features of the REPL, including the integration with Jupyter and the kernel, have already been implemented successfully. However, the advanced features, such as syntax highlighting and code-completion, have yet to be implemented as of this report. All the required interface between the Jupyter client and kernel for these features is already in place automatically by Jupyter. The only remaining needed code for code-completion is that which will parse the current input (passed down by client to kernel) and determine appropriate hint output (returned to the client from kernel), and similarly for syntax highlighting and others. A few issues remain to be solved as of yet, one of which is the fact that Jupyter and most interactive environments are not designed for tasks that run in the background, as is the main feature of Swift/T REPL. This means that STDOUT from Turbine workers running in the background needs to be redirected properly to the Jupyter client, and in such a way that the client can display that the output is yet to return from running in the background. This is crucial for implementing the Notebook feature.

Weeks 1-4 of the REU program were spent familiarizing with the overall Swift/T project architecture, including setting up binary Debian packages for Swift/T. During weeks 5-6, the initial REPL implementation was written, and the technical Turbine workarounds were figured out. At that point, the overall design for the REPL (Figure 1) was formed. The issues of linking multiple scripts and global "extern" variables were solved during weeks 7-8, which required writing new language constructs and numerous changes to the compiler (STC). The final weeks (9-10) were spent on writing the Jupyter kernel to interface between the Jupyter client (user) and the REPL.

6. CONCLUSION

Despite not being originally designed with interactivity in mind, Swift/T was adapted to work with an interactive shell, and its usability and accessibility to novice programmers has greatly improved. The main objectives of the REPL project were achieved, and as the use cases in the Appendix show, users can now interactively start tasks in Swift and monitor the status of the system with very little overhead . Moreover, the interface to Jupyter will add many useful features like code-completion and Notebook documents, which will make it even easier to use and collaborate on Swift/T scripts.

APPENDIX

A. EXAMPLE INTERACTIVE USE CASES / SCENARIOS

A.1 Monitor status of array filling

```
> int A[]; // create an array
> foreach i in [1:1000] { // loop 1000
    times and fill up A
...     A[i] = i;
...}
> // prompt returns as script is running in
    the background
> %ls     // user decides to view status of
    variables
variable u:A has id 125 and values 1 2 3 4
    5 6 7 8 9 10...
```

A.2 Referencing Variables from Multiple Scripts

```
> int x = 7;
...
> int y = x + 22;  // 'extern int x;'
    implicitly placed in script
```

A.3 Redefining Variables

```
> int x = f();
...
> int x = g();
```

B. REFERENCES

[1] D. E. Knuth. Literate programming. *The Computer Journal*, 27(2):97–111, 1984.

[2] T. J. Parr and R. W. Quong. Antlr: A predicated-ll (k) parser generator. *Software: Practice and Experience*, 25(7):789–810, 1995.

[3] F. Pérez and B. E. Granger. Ipython: a system for interactive scientific computing. *Computing in Science & Engineering*, 9(3):21–29, 2007.

[4] J. C. Phillips, J. E. Stone, K. L. Vandivort, T. G. Armstrong, J. M. Wozniak, M. Wilde, and K. Schulten. Petascale tcl with namd, vmd, and swift/t. In *Proceedings of the 1st First Workshop for High Performance Technical Computing in Dynamic Languages*, pages 6–17. IEEE Press, 2014.

[5] H. Shen et al. Interactive notebooks: Sharing the code. *Nature*, 515(7525):151–152, 2014.

[6] J.-L. R. Stevens, M. Elver, and J. A. Bednar. An automated and reproducible workflow for running and analyzing neural simulations using lancet and ipython notebook. *Frontiers in neuroinformatics*, 7, 2013.

[7] J. M. Wozniak, T. G. Armstrong, M. Wilde, D. S. Katz, E. Lusk, and I. T. Foster. Swift/t: large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 95–102. IEEE, 2013.