A model for tracing and debugging large-scale task-parallel programs with MPE

Justin M. Wozniak*‡, Anthony Chan*‡, Timothy G. Armstrong†, Michael Wilde*‡, Ewing Lusk*‡, Ian T. Foster*†‡
* Mathematics and Computer Science Division, Argonne National Laboratory, Argonne, IL, USA
† Dept. of Computer Science, University of Chicago, Chicago, IL, USA
‡ Computation Institute, University of Chicago and Argonne National Laboratory, Chicago, IL, USA

I. Introduction

Application frameworks and domain-specific languages (DSLs) (both here called *high-level tools*) aid developers when developing programs for next-generation, highly concurrent systems. In the best case, these tools allow developers to *mentally* remain in their area of expertise, such as a physical science, while producing high-level expressions for computation. An increase in the use of high-level tools, however, creates a problem: the prevention and detection of defects in the high-level program. Traditional debuggers, designed for operating on highly popular, line-oriented languages (C, C++, Java), will operate at too low a level to detect defects with the use of the high-level tool. With regard to scale, while line-oriented tools are expected to remain viable on foreseeable systems, the ability of the human user to effectively use these tools at larger scale is questionable.

Logging is a typical approach to monitoring and detecting problems in program execution. At present, logging faces similar challenges to traditional debuggers: utility and scale. In an application built in a high-level tool, logging must also be efficient and high-level. Low-level logging, say, at the messaging level, will be too complex for users of high-level tools. Thus, *the tool builders must integrate effective, high-level logging functionality*.

Here, we consider the use of the Message Passing Environment (MPE) [1] logging applied to Swift [2] programs. This case study presents a model for effectively detecting programming defects based on the MPE logs in a manner appropriate for Swift programs (not MPI programs in general).

II. Components: Swift and MPE

Swift is a naturally concurrent scripting language designed to ease the development of scientific applications built from large numbers of calls into existing programs and libraries [2]. Although its initial implementation was designed to run on a single submit host and manage external execution on the grid, its new implementation, Swift/Turbine (Swift/T), generates an MPI program from the user script via two libraries, Turbine [3] and the Asynchronous Dynamic Load Balancer (ADLB) [4]. Turbine manages the dataflow semantics of the Swift language in a scalable, distributed-memory framework. ADLB manages tasks resulting from the execution of the user script logic. A simple Swift program is diagrammed in Fig. 1:

```
1 data d[];
2 foreach i in [0:N-1]
3 {
4   data a = A(i);
5   data b = B(a);
6   data c = C(a);
7   d[i]   = D(b,c);
8 }
9 output(analyze(d));
```

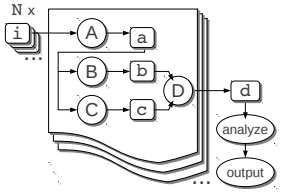

Fig. 1. Simple data flow application.

In this example, the implicitly parallel semantics of Swift are evident. Each iteration of the `foreach` loop happens concurrently. For each iteration, variables `a`, `b`, and `c` are allocated. Functions `A`, `B`, `C`, and `D` are linked to external, native user code. (Swift also supports functions.) Each task `A` is immediately ready to run; each task `B`, `C` waits for the intermediate output of `A`; `D` waits for intermediates `b`, `c`; `analyze` and `output` wait for the completion of all insertions into array `d`. Swift has been shown to be capable of handling extremely large use cases, including the use of 128K cores on the Blue Gene/P.

MPE provides a scalable system for logging, profiling, and visualizing MPI program executions. MPE is implemented as a pure MPI library; it may be used by any MPI implementation. MPE is often used to trace usage of the MPI library itself, for example, to track and visualize message patterns. However, it may be extended by the user to track arbitrary *events* and/or *states*. MPE limits impact on applications and achieves good performance by buffering events and I/O locally with respect to the MPI process.

MPE has been previously considered for use as a defect investigation tool; however, as it is an library built on MPI, it is not more reliable than the application program or the system implementation. Typical program defects in traditional languages will cause the termination of the whole application; MPE will not be able to complete writing the log. In this work, we seek to find defects in the use of the high-level tool, *errors that will not obviate the production of the MPE log*.

III. Model for Defect Investigation

Logs are produced from ADLB programs linked with MPE in multiple ways. First, ADLB operations from workers may be logged in a manner analogous to that for MPI operations; thus, calls to the task `put` and `get` functions are logged. Second, the time in user code may be measured; this feature allows for the user to observe the length of time in user work

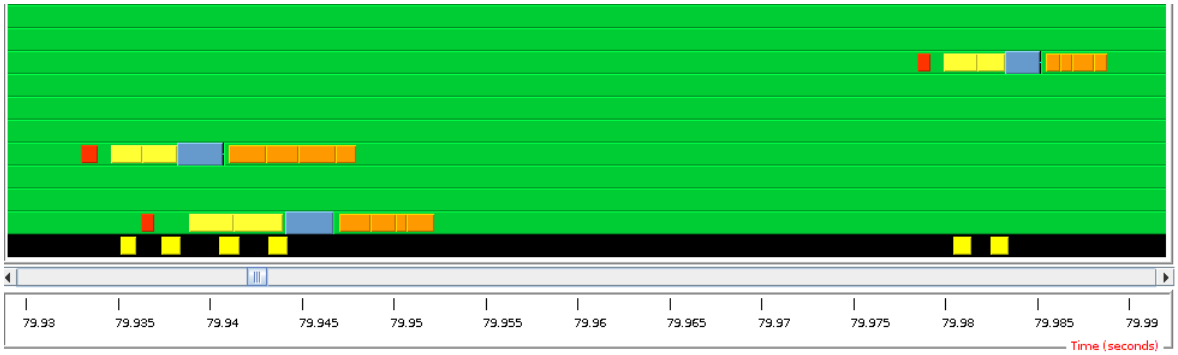

Fig. 2. Jumpshot visualization for PIPS use of ADLB in Swift/T.
This figure zooms in on representative task transitions.
Green: PIPS task computation; **Red:** Store variable; **Yellow:** Notification (via control task); **Blue:** Get next task;
Orange: Retrieve variable; **Black:** Server process (handling of control task is highlighted in yellow)

functions outside ADLB. Third, MPE *solo* events may be issued; these contain a timestamp, a type, and a user binary byte sequence (often a string). This third event type is the focus of our present work: we intend to encode program trace information in these events.

The Swift/T project uses ADLB features pervasively and added multiple features to support data-dependent execution. Thus, additional calls to `store`, `retrieve`, and `subscribe` to data are available. As a result, a trace of program execution may be represented in Jumpshot as shown in Figure 2 (MPE log from a run of a PIPS-related Swift/T application [5]). As shown in the figure, worker processes spend the bulk of time in user task computation. The transition between tasks is shown as the brief multi-colored regions; the length of the transition may be considered Swift/T overhead. At the end of a task, resultant variables are stored, notifications are issued, a new task is obtained, its input variables are read, and work resumes. This illustrates a load-store, Von Neumann-style computing model for distributed-memory task-parallel programs generated by a high-level functional language.

Diagnosing defects by querying the log begins in one of the following cases: **Bug 1)** An unintended user task is issued; **Bug 2)** An intended user task is not issued; **Bug 3)** Deadlock occurs. In the hierarchical, task-parallel Swift model, defects in native user code is treated as a separate problem.

Consider Bug 1. In this case, the user would have detected that a call to the native code was made in error through some external mechanism. To obtain information about the programming error in the Swift script, the user would benefit from a stack trace. This may be reconstructed from the MPE log as follows. As shown in Fig. 1, ADLB tasks are associated with Swift *leaf* functions; that is, certain user-specified Swift functions are implemented as calls into user code (written in C/C++/Fortran, etc.) to be executed as ADLB tasks. Swift *composite* functions make up a call stack in a conventional, concurrent manner, starting from the user Swift `main()` (not the C or system `main()`). At runtime, execution of Swift composite functions is realized as execution of ADLB tasks, which allows for high concurrency and load-balancing. ADLB task execution is already logged; thus, a record of these executions is readily available in the log.

Given an erroneous ADLB task, and associated events in the MPE log, the user is able to use Jumpshot to visualize the task. The task identification also allows the user to obtain the Swift composite function context from which the task was issued. Given the Swift composite function, variables in the stack frame may be identified, and their values may be inspected. The parent stack frame may also be identified, and the process of inspecting variables in the parent frame may be repeated up to Swift `main()`.

Bug types 2 and 3 may be investigated in a similar manner (condensed here for space consideration). Bug type 2 would likely be investigated starting from `main()` and working in the opposite direction in the call stack until the defect that led to erroneous execution was identified. Bug type 3 would likely present itself as Bug type 2, however, the Swift/T runtime issues warnings for incomplete execution of the program and exits with a valid log. The user could trace the call stack from `main()` to find the variables that form a cyclic dependency.

IV. Summary

In this work, we presented the use of MPE for detecting defects in parallel programs written using high-level tools relevant to the application framework ADLB and the high-level language Swift/T.

References

[1] A. Chan, W. Gropp, and E. Lusk, "An efficient format for nearly constant-time access to arbitrary time intervals in large trace files," *Scientific Programming*, vol. 16, no. 2-3, pp. 155–165, 2008.
[2] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster, "Swift: A language for distributed parallel scripting," *Parallel Computing*, vol. 37, pp. 633–652, 2011.
[3] J. M. Wozniak, T. G. Armstrong, K. Maheshwari, E. L. Lusk, D. S. Katz, M. Wilde, and I. T. Foster, "Turbine: A distributed memory data flow engine for many-task applications," in *Int'l Workshop Scalable Workflow Enactment Engines and Technologies (SWEET) 2012*, 2012.
[4] E. L. Lusk, S. C. Pieper, and R. M. Butler, "More scalability, less pain: A simple programming model and its implementation for extreme computing," *SciDAC Review*, vol. 17, pp. 30–37, Jan. 2010.
[5] M. Lubin, C. G. Petra, M. Anitescu, and V. Zavala, "Scalable stochastic optimization of complex energy systems," in *Proc. SC*, 2011.

(The following paragraph will be removed from the final version)