Productive composition of extreme-scale applications using implicitly parallel dataflow

Michael Wilde,*‡ Justin M. Wozniak,*‡ Timothy G. Armstrong,† Daniel S. Katz,‡ 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

In every decade since the 1970's, computer scientists have re-examined dataflow-based execution models, hoping the programming productivity benefits these models promise can be realized on practical hardware platforms to implement useful applications. Based on the recent Swift/T implementation of "implicitly parallel functional dataflow" (*IPFD*) for extreme-scale systems, we believe that the dataflow model can now provide significant software productivity benefits for developing the upper levels of petascale and exascale applications. We believe dataflow models can also yield significant secondary benefits in verifiability, reliability, and energy savings – areas that are closely related to overall systems development productivity at extreme scale. We discuss here the opportunities for software productivity at these scaled presented by the use of IPFD, and the research and development efforts needed to realize these benefits.

II. Drivers for large-scale many-task applications

Exaflop computers are expected to provide concurrency at the scale of O(10^9) threads on O(10^6) cores. Such extreme-scale systems will enable and demand new problem-solving methods that do not strictly follow today's dominant single-program, multiple data (SPMD) paradigm but instead involve many (often a time-varying number of) concurrent run-to-completion tasks. These tasks may themselves be the invocation of multi-core or multi-node programs, and may have possibly complex inter-task data dependencies. Such applications have been recently described as "many-task" in nature.

Writing correct, scalable programs at this level can be an herculean task, with significant investment of programmer time required to make a program run efficiently and verifiably on hundreds of thousands of cores. Applications at this scale could have a development cycle approaching a decade.

For some applications, intricate high-level coordination logic is necessary; but in other cases, the high-level coordination pattern is relatively straightforward and may be expressed as the composition of a number of computational tasks. In practice, the composition takes the form of scripted dataflow logic, in which tasks are linked together through their input and output data sets; the tasks themselves are developed separately as libraries or external programs. Important applications in methodologies such as rational materials design, uncertainty quantification, parameter estimation, and inverse modeling, and the analysis of complex datasets such as climate models and social network graphs all have this many-task property. Many will have aggregate computing use cases that require exascale computers.

III. Current approaches to many-task application development

Currently, many-task applications are programmed in one of two ways. In the first approach, the logic associated with the different tasks is integrated into a single, tightly coupled application using a load balancing library which allows tasks to add work dynamically to the system. This lacks a comprehensive programming model, data model, and other features required for productivity. In the second approach, a script or workflow is written that invokes the tasks, in sequence or in parallel, with each task reading and writing input and output files. Here, performance can be poor, because existing many-task languages are implemented with *centralized evaluators* that cannot sustain the high overall task rate necessary to efficiently utilize O(10^6) cores.

IV. Applying IPFD using Swift

As application scale increases these features are increasingly important, yet more difficult to implement. We and our users have significant experience working in this paradigm with the Swift parallel scripting language [1], which can compose existing programs into more sophisticated applications such as simulation or analysis pipelines, parameter sweeps, or workflow graphs. Our experience with real applications suggests that IPFD provides a comprehensive strategy to perform such high-level application coordination at extreme scales with greater programmability.

The approach we advocate here integrates these two models. First, we provide a very high-level, naturally concurrent programming model in the previously developed Swift language. Second, we developed translation strategies to render Swift semantics into a distributed-memory model, based on efficient primitives compatible with the highly scalable ADLB library – the primary focus of this paper. Swift's IPFD computing model draws on recent trends that emphasize the identification of coarse-grained high-level parallelism as a distinct and early step in application development [2].

Applications relevant to the DOE mission-science portfolio that have been developed in Swift include large-scale power grid simulation; analytic diagnostics of high-resolution climate models; simulation and live analysis of x-ray scattering results from the Advanced Photon Source; materials design using parallel ensembles of molecular dynamics simulations tools like NAMD and LAMMPS; and multi-scale subsurface flow modeling. Important DOE application suites like the NWChem suite, while not programmed in Swift, exhibit a high-level structure quite amenable to the IPFD model.

V. Scaling the Swift IPFD model

Until 2009, Swift had limited applicability to extreme-scale applications. The leaf tasks of a Swift script could only be complete application programs, and Swift's Java implementation could only sustain about 500 task invocations per second. In 2010, a new version

Swift/Turbine [3], now known as Swift/T, was created that addressed both of these limitations. Swift/T enabled ordinary in-memory functions to be called as leaf dataflow tasks, and the task rate was increased to what is now at the time of this writing over 14M tasks per second (as measured recently on Blue Waters at 2K nodes / 64K cores). We expect that the current engine is capable of much more speed on an even larger number of nodes. Swift/T programs are executed as MPI applications using enhanced versions of the ADLB load balancer, and can thus fully leverage the native interconnect and high speed RDMA transports of extreme-scale systems.

VI. Extreme-scale applications will be hybrids

Our view is that a significant fraction of extreme-scale applications will require a hierarchy of programming models. Diverse finer-grained parallel models will still be used to implement core application logic. However, an implicitly parallel, functional, dataflow-based programming model is attractive for top-level coordination logic, because load balancing, fault tolerance and resource management fit naturally as application-agnostic services within the model.

The IPFD model readily supports *parallel* leaf tasks (i.e leaf tasks written in models such as MPI and OpenMP). Both such models have been executed in Swift and Swift/T. For example, the SciCol graph-based simulation model/optimizer employs OpenMP leaf tasks, and parallel multi-node MPI applications like NAMD, CP2K, and LAMMPS have been used as leaf tasks in molecular dynamics applications.

The Swift many-task model has recently been extended to the realm of accelerators. The GPU-enabled Many-Task Computing framework from IIT has been integrated with Swift. Applications such as protein structure prediction[2] are being ported to GPUs for execution on Blue Waters and Stampede using this framework.

VII. Extreme applications will be multilingual

The above need for hybrid execution and programming models within large-scale applications is driven in large part by the natural evolution within the DOE mission sciences to build on top of the largest functional units of software reuse possible. We expect that applications will be most productive when they leverage large community codes packaged as libraries and compiled application programs.

The same forces will drive the requirement for extreme-scale applications to be written in (and thus to integrate) a diverse set of languages. Leaf codes for simulation will continue, we expect. to be coded in C/C++, Fortran, and perhaps a growing set of PGAS and global-view languages such as Chappell, X10, and UPC. However, many codes will employ languages like R for statistical and data analysis and to perform the logic of algorithms like uncertainty quantification, branch-and-bound, or of intelligent, dynamic ensembles. Languages such as Python will play a major role in fields like biology (BioPython) and as high-productivity interfaces to numerical libraries (such as NumPy).

These trends will require that an IPFD language for high-level composition integrate a diverse set of languages. Swift/T has begun this process by enabling leaf tasks to be written in C/C++, Fortran, R, and Python. These capabilities are already proving useful in coding genetic algorithms into intelligent simulation ensembles for x-ray scattering applications.

VIII. Research needed to realize this vision

The following research problems must be addressed to enable the IPFD model, using Swift or other future programming methods, to achieve productivity breakthroughs in extreme-scale application development and support.

1) Improve the ease of inter-language calls and flow generation
2) Develop the tools to visualize and tune scaling
3) Extend the programming model to handle speculative execution, efficient dynamic result reduction, and more powerful loop constructs such as "for-most-of", or "until-N-results", inspired by workflow models
4) Provide more language bindings e.g. for Python-based dataflow specification.
5) Integrate with structured dataset formats
6) Support rich logging and debugging for high-level tools.
7) Control and aggregate the level of detail of provenance data recording

Acknowledgments: This work was supported in part by the U.S. Dept. of Energy, Office of Science, Office of Advanced Scientific Computing Research, under Contract DE-AC02-06CH11357.

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*, vol. 39, no. 9, pp. 633–652, September 2011. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0167819111000524

[2] M. Wilde, I. Foster, K. Iskra, P. Beckman, Z. Zhang, A. Espinosa, M. Hategan, B. Clifford, and I. Raicu, "Parallel scripting for applications at the petascale and beyond," *Computer*, vol. 42, no. 11, pp. 50–60, 2009.

[3] 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," in *Proc. CCGrid*, 2013.