Implicitly parallel functional dataflow for DOE Science Workflows

Daniel S. Katz, Michael Wilde, Justin M. Wozniak

Over the past decade, “workflow” has come to describe an important class of techniques for the execution of related groups of tasks. Workflows may involve significant repetition of similar tasks, grouping of tasks, and sequential or parallel execution of tasks and groups.

What distinguishes “workflow” programming from other programming models—making it so important—is ***productivity***. Workflow programming typically focuses on the higher levels of some computational activity: running ten thousand instances of a protein model over many days, or running ten billion matrix computations in an hour based on their dataflow dependencies. We re-purpose the term “programming in the large” to refer to this class of computational activity: workflow programming almost always involves manipulating well-encapsulated units of code (tasks, e.g., functions or complete application programs) expressed in traditional programming or scripting languages.

Our view of workflow encompasses a vast range of scales:

- Task durations within workflows can range from days down to a few milliseconds;
- Workflows can have vastly different task counts: from two or three up to many trillions;
- Workflows can have different task arrangement patterns: from simple linear pipelines to complex dependency graphs with significant parallelism;
- Workflows can themselves be composed from other workflows of lower scale; and
- Workflow patterns can be statically specified, or can change dynamically during their execution, and from one execution to the next.

We thus refine the suggested taxonomy of “DA” and “IS” workflows using two axes:

Granularity:

- ***function-level (in-memory) workflow*** based on the composition of (library) functions and the passing of in-memory data as objects
- ***application-level (via-disk) workflow*** based on the composition of (compiled) application programs and the passing of data between invocations as files

Locality:

- **centralized workflows** that run on a single “platform” – one node, one cluster, one cloud (and often one administrative domain)
- **distributed workflows** that run across multiple platforms – across multiple clusters or cloud providers (and often multiple administrative domains)

We reserve the term “*in situ*” to describe the important case of efficiently passing data arguments to workflow functions in-place rather than moving data within the workflow.

Experiences with the Swift Parallel Scripting System

Our belief and experience indicates that basing workflow on data dependencies—i.e., on “dataflow”—yields significant productivity benefits at multiple programming granularity levels. The Swift (http://swift-lang.org/) language was initially designed for scientific computing and data analysis on distributed systems (c. 2007).[1] Its goal was to determine if distributed parallel application-level workflow could be productively, efficiently, and simply expressed using a functional dataflow-based programming language. The benefits of this approach have been demonstrated in over 30 significant scientific engagements

[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(9), 2011.

across a broad range of DOE, NSF, NIH, and NOAA application domains as documented at http://swift-lang.org/papers.

In 2010 we extended Swift to a centralized in-memory workflow model for extreme scales by compiling it into MPI.[2,3] This “Swift/T” can run 1.5B tasks/sec on 512K cores for 100-μs tasks, thus extending implicitly-parallel productivity to a wider range of applications and programming models.

Workflow Research Challenges and Opportunities

Many topics for further research are suggested by our experiences with Swift and other workflow systems/representations, at multiple granularities and across many applications and environments.

Workflow models based on periodic message exchange. Can models where the workflow is like a Petri net in which stateful tasks are scheduled and managed by the runtime, communicating over high-performance channels, but the overall workflow is programmed by a high-level script that controls progress via lightweight messages, be integrate into the functional dataflow model?

How is data passed between workflow tasks? (e.g., files, objects, messages, streams). Streams enable the receiver to begin processing data before the producer has finished producing it. Common examples are UNIX pipes or sockets. Can workflows control the use of high-performance channels? Can these channels be abstracted for external communication (sensors, visualization)?

Additional research on in situ argument passing holds great promise for reducing energy demands and latency in the analysis of extreme-scale capability-class simulations, where writing data to disk or even moving it from the node or out of cache is intractable.

How can multiple task languages be integrated within an extreme-scale workflow? Swift has demonstrated this; further research is needed to make inter-language calls easy, fast, and *in situ*.

Automating energy and fault management are critical for practical and productive use of extreme-scale systems. Both may be greatly facilitated by using knowledge of dataflow patterns.

How are workflows mapped to specific resources? What is known at deploy time vs. run time? Can the workflow system respond to dynamic resource changes? if so, how? Can a common dynamic resource description interface be defined? What emergent OS capabilities are relevant?

How can workflows be represented logically, and then mapped to multiple representations? Textually, workflows may be encoded in Makefiles, Python scripts, and visual and textual workflow languages like Swift. Workflow abstractions may be based on functional tasks, services, or actors. While it is not clear how to translate between all of them, what commonalities exist? How can a common representation ease integration with existing codes, languages and target environments?

Where should workflow data (state) be stored? (e.g., a distributed in-memory data store, a distributed file system, a shared file system, a database). This impacts fault recovery, restart, workflow mutability/evolution, etc.

What types of infrastructures are supported? (parallel vs. physically distributed, e.g. HPC vs. grid/cloud) How portable can a workflow system be across different types of computing systems?

[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 distributed-memory task-parallel applications. Proc. CCGrid 2013.

[3] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. Proc. SC 2014.