ExM: High-level dataflow programming for extreme-scale systems

Timothy G. Armstrong[a] Justin M. Wozniak[b] Michael Wilde[b] Ketan Maheshwari[b]
Daniel S. Katz[ab] Matei Ripeanu[c] Ewing L. Lusk[b] Ian T. Foster[ab]

[a]*University of Chicago,* [b]*Argonne National Laboratory,* [c]*University of British Columbia*

Abstract

We present here the ExM (extreme-scale many-task) programming and execution model as a practical solution to the challenges of programing the higher-level logic of complex parallel applications on current petascale and future exascale computing systems. ExM provides an expressive, high-level functional programming model that yields massive concurrency through implicit, automated parallelism. It comprises a judicious integration of dataflow constructs, highly parallel function evaluation, and extremely scalable task generation. It directly addresses the intertwined programmability and scalability requirements of systems with massive concurrency, while providing a programming model that may be attractive and feasible for systems of much lower scale. We describe here the benefits of the ExM programming and execution model, its potential applications, and the performance of its current implementation.

1 Introduction

Exaflop computers capable of 10^{18} floating-point operations/s are expected to provide concurrency at the scale of $O(10^9)$ threads on $O(10^6)$ cores [21]. Such extreme-scale systems will enable and demand new problem-solving methods that do not follow today's dominant single-program, multiple data (SPMD) paradigm but instead involve many (often a time-varying number of) concurrent and interacting tasks. Writing correct, scalable programs at this level can be an onerous task, with significant investment of programmer time required to make a program run efficiently on hundreds or thousands of cores. Applications at this scale can 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 design, uncertainty quantification, parameter estimation, and inverse modeling all have this many-task property. Many will have aggregate computing use cases that require exascale computers. The ExM computing model draws on recent trends that emphasize the identification of coarse-grained parallelism as a first and separate step in application development [13, 22, 23]

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 such as the MPI-based [14] Asynchronous Dynamic Load Balancing Library, ADLB [11], or the Global Arrays-based [17] Scioto [7]. They provide a master/worker system with a put/get API for task descriptions, thus allowing workers to add work dynamically to the system. However, they lack a comprehensive programming model, data model, and other features required for high-productivity programming. 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 or streams. However, performance can be poor, because existing many-task scripting languages are implemented with *centralized evaluators* that cannot sustain the high overall task rate necessary to efficiently communicate with and utilize $O(10^6)$ cores.

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. As application scale increases these features are increasingly important, yet more difficult to implement. We have previous experience working in this paradigm with the Swift parallel scripting language [24], which can compose existing programs into more sophisticated applications such as simulation or analysis pipelines, parameter sweeps, or workflow graphs. The contribution of this paper is a comprehensive strategy to perform such high-level application coordination at extreme scales with greater programmability.

Previous approaches to workflow execution on high-

performance resources have involved deploying a toolkit developed for distributed systems on the target infrastructure. Software systems relevant for this model include Dryad [9], Skywriting [15]/CIEL [16], and Swift [24]. This approach is convenient for the user, particularly when each task is a distinct executable program. The approach faces multiple performance challenges, however, including the ability to rapidly launch independent processes [20], manage large numbers of pilot jobs [12], communicate over an emulated TCP network [10], and coordinate data access [26].

Alternatively, the developer may hand-code a work distribution system using available high-performance tools, communicating through MPI messaging in distributed memory or function calls (as in the parallel version of the Common Component Architecture [2].) This approach uses familiar technologies but can be inefficient unless much effort is spent incorporating load-balancing algorithms into the application. Moreover, the approach can involve considerable programming effort if multiple component codes are to be integrated. Partitioned global address space (PGAS) [19] language features provide a partial solution to the data model but do not offer notifications and other features necessary for the construction of high-level scripts.

Our approach 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.

2 modFTDock: A sample application

Running many-task applications, efficiently, reliably, and easily on large-scale machines is challenging. We present *modFTDock* [18], a relatively simple application analyzing protein docking to highlight the challenges. As shown in Figure 1, *modFTDock* starts with M input files and N input parameters. Each of these $M \times N$ combinations is processed by the sequential `modftdock` task. The resulting docking data is stored and processed later by tasks `merge` and `score`, which produce the requisite results. All the application stages communicate only through their input and output data. Figure 2 illustrates the simple specification of this dataflow in Swift. Quantitative information for a contemporary *modFTDock* run is tabulated in Table 1; conceivable future experiments could be composed of trillions of tasks.

The challenge is to efficiently, reliably, and scalably coordinate the million tasks generated by the *modFTDock* application while at the same time using a compact, programmer-friendly specification that can support the integration of legacy code.

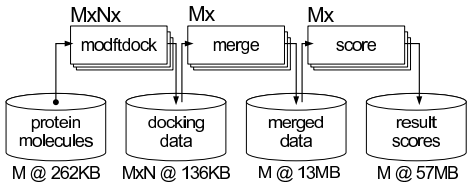

Figure 1: Dataflow schematic for modFTDock. The output sizes are for a single run of an application task.

```
1  dock_score scores[];
2  foreach (p1, i in proteins) {
3    dock_result docked[];
4    foreach (p2, j in proteins) {
5      if (i < j) {
6        docked[j] = modftdock(p1, p2);
7      }
8    }
9    scores[i] = score(merge(docked));
10 }
```

Figure 2: Swift implementation of modFTDock.

3 Swift: A dataflow language to support many-task applications

The canonical applications for which Swift was originally designed had most of the sequential computation code already written and encapsulated as command-line binaries that needed to be coordinated as a workflow. Traditionally UNIX shell scripts have been used to this end, but Swift was designed to better support running such applications in distributed and parallel contexts, where synchronization, data movement, explicit task scheduling, and fault tolerance are necessary.

The Swift execution model may be split into two processes: *task generation*, which generates concurrent tasks by interpreting the user dataflow script, and *task execution*, which distributes the resulting side-effect-free *leaf tasks* and orchestrates their execution. *Leaf tasks* may be implemented as procedures and correspond to library call invocations, or standalone executables, in which case they correspond to launching a new process. Leaf tasks themselves may use multiple cores or even multiple nodes.

To link the above to our sample application, we treat the components of modFTDock as leaf tasks coordinated by a Swift script. For modFTDock, the leaf tasks are single-process executables, with concurrency exposed in Swift. Data dependencies, task distribution, and data movement are managed by the system as follows.

Table 1: Statistics for a full modFTDock application run.

Task	Number of Tasks	Duration (s)
`modftdock`	1,200,000	1,000
`merge`	12,000	5
`score`	12,000	6,000

In Swift, unlike in shell scripts, all inputs and outputs of a (side-effect free) leaf task must be explicitly defined, so that the Swift runtime has enough information to manage its input and output. A Swift app function definition such as the one below converts a standalone executable (the `convert` utility) to a Swift function with arguments and return values (such as `image` files or `int` parameters).

```
app (image out) rotate(image in, int angle) {
    convert "-rotate" angle @in @out;
}
```

We will provide an extension mechanism to allow functions from other languages to be defined and compiled/linked as Swift tasks as well. In addition to these external functions, functions can also be defined within Swift, comprising multiple Swift statements.

The parallelism in a Swift script is exposed implicitly, with the order of execution of Swift statements determined entirely by data dependencies. Multiple statements and subexpressions can execute in parallel, given no data dependencies and sufficient parallel computing resources. Consider the following example:

```
(datafile result) process (datafile in) {
  datafile foo; datafile bar;
  foo = f(in);
  // g and h below can run concurrently with f
  bar = g(in);
  result = j(foo, h(bar));
}
```

Each iteration of a `foreach` loop in Swift runs independently, but data dependencies may serialize execution.

```
int out[];
foreach f, i in myfiles {
  // Each iteration is completely independent
  out[i] = readData(process(f));
}

int out[];
foreach f, i in myfiles {
  // But these are serialized by data dependencies
  if (i > 0) {
    out[i] = readData(process2(f, out[i-1]));
  } else {
    out[i] = readData(process(f));
  }
}
```

The Swift language design ensures that even high-concurrency programs are *deterministic by default* [4]. The core Swift language constructs are deterministic; non-determinism can originate only from non-Swift code. A valid Swift program always produces the same output (although the ordering of side-effects such as log messages can vary).

The main feature that enables this property is the use of write-once variables: each Swift variable can be written to only once. Writing twice causes a compile or runtime error.

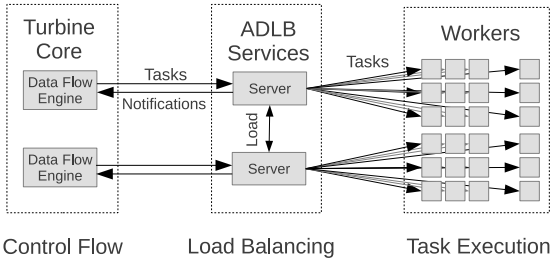

Figure 3: Task distribution in the Turbine runtime system.

Write-once variables give programming in Swift a nonimperative programming flavor and eliminate the possibility of many concurrency bugs. A Swift compiler can automatically detect or warn of many classes of errors such as deadlocks from circular dependencies or unassigned variables. Including arrays in the language allows more runtime errors because of the impossibility of deciding statically whether a particular array index is modified (cf. the halting problem), but these quasi-imperative arrays of write-once variables are more expressive than purely functional alternatives [3] and are less of a leap for programmers familiar with imperative languages.

In summary, Swift programs are well suited for expressing the upper-level concurrency of complex applications that integrate a variety of other functional components (often written in other diverse parallel programming models). The Swift runtime provides the scalability and performance necessary to manage millions of task definitions and input/output data objects. This allows the use of distributed memory to store script control variables and cache user datasets, while resolving the data dependencies that coordinate independent processes.

4 ExM architecture

We are developing a new implementation of Swift based on the ExM extreme-scale many-task execution model. This implementation performs fully distributed execution of a Swift program with no centralization of control flow.

The full ExM system comprises a distributed version of Swift and MosaStore, a distributed in-memory file system [6]. We discuss only the former in this paper, which is implemented as two subsystems: the runtime system (called Turbine) [25] and the Swift-to-Turbine compiler (called *stc*).

We think of the intermediate code, the crucial interface between the two, as the instructions for an abstract workflow machine. The set of runtime system primitives for task management, data management and synchronization is kept as minimal as possible, in order to make the Turbine runtime as robust and flexible as pos-

```
1  main {
2    int x;
3    int y;
4    int z = 1;
5    (x, y) = f(z);
6    if (x != 0) {
7      trace(y);
8    } else {
9      trace(z);
10   }
11 }
```

Figure 4: Swift script to be compiled by *stc*.

```
1  # Main program fragment- starts on a single process
2  proc _swiftmain {} {
3    # allocate data in global data store
4    allocate u:x integer
5    allocate u:y integer
6    allocate u:z integer
7    set_integer u:z 1
8
9    # post call to f, with input x and output x, y
10   call_composite f [ u:z ] [ u:x u:y ]
11
12   # post call waiting until conditional expression evaluated
13   rule [ u:x ] if-0 [ u:x u:y u:z ]
14 }
15
16 # This program fragment executes sometime after x is written
17 proc if-0 { u:x u:y u:z } {
18   set v:x [ get_integer u:x ]
19   if {v:x} {
20       call_builtin trace [] [ u:y ]
21     } else {
22       call_builtin trace [] [ u:z ]
23     }
24 }
```

Figure 5: Generated Turbine intermediate code. Each fragment is sequentially evaluated, with each `call` command creating an asynchronous task.

sible. Figures 4–7 illustrate how Swift code is translated into intermediate code for Turbine's consumption.

The Turbine runtime system currently is built on top of MPI, running on a cluster with all communication between components using messages. Using MPI made it easy to port to several cluster architectures, including IBM Blue Gene/P, Cray, and SiCortex systems. The MPI processes are divided among three roles: ADLB servers that manage the task queue and data store, Turbine rule engines that track data dependencies and execute intermediate code, and workers that exclusively execute leaf tasks. Typically the bulk of processes are workers, since the bulk of computation occurs in leaf tasks.

5 Implementation progress and challenges

Currently, we have a working compiler and runtime system for the core of the language, including functions, loops and recursion, conditionals, arrays, and structs. We are focusing now on scalability, running Swift scripts on tens of thousand of cores.

A number of challenges arise in making the Turbine interpreter scale. ADLB provides a strong base on which

```
1  # int A[];
2  allocate_container u:A integer #  container with int keys
3
4  # A[0] = 1;
5  allocate t:0 integer
6  set_integer t:0 1
7  container_insert_imm u:A 0 t:0
8
9  # A[f()] = g();
10 allocate t:1 integer
11 call_composite g [ t:1 ] []
12 allocate t:2 integer
13 call_composite f [ t:2 ] []
14 container_insert_future u:A t:2 t:1
15
16 # trace(A[0]);
17 allocate t:3 integer
18 # t:4 is a reference, stored as integer
19 allocate t:4 integer
20 container_lookup_ref_imm t:4 u:A 0
21 dereference_int t:3 t:4
22 call_builtin trace [] [ t:3 ]
23
24 # cleanup operation: decrement container writer count
25 container_decr_writers u:A
```

Figure 6: Generated Turbine intermediate code for array operations, with corresponding Swift lines in comments.

to implement task distribution, but a naive approach to task generation can put unnecessary strain on load balancing and data dependency management processes. A naive foreach loop, without throttling or loop splitting could create hundreds of thousands of tasks simultaneously, swamping ADLB. Long data dependency chains between tasks in the runtime also occur with a naive approach. Static analysis is necessary to defer task creation until data is ready and to coalesce tasks if possible.

Swift's data model also presents challenges, with data potentially shared by many tasks. With load balancing any data must remain accessible to tasks after relocation. Turbine provides a global data store for this purpose. Primitive data types such as numbers or strings are stored directly in the data store. Arrays and structures also reside in the data store as Turbine *containers*, a dictionary data type, with linked containers supporting more complex data structures. Containers are specialized to support language-level determinism. Inserts and lookups to containers must be commutative with each other to support distributed evaluation, meaning lookups must often wait for matching insertions to occur, and lookups must eventually fail if an array cell is never written. Hence, the interpreters need to reach a consensus on when a container is *closed* (i.e., no more writes will occur). To this end, we use static analysis in the compiler and special reference counting operations in Turbine. For scalability, Turbine supports *distributed* containers, with the container split between data servers by index range.

Logically, all Swift variables are values, rather than references; but for efficiency we want to avoid doing excessive copies-by-value, particularly of arrays. Copying references, however, introduces the problem of garbage

```
1  int A[]; int B[];
2  A = constructArray();
3  foreach x, i in A {
4     B[i] = f(x);
5  }
```

```
1  proc _swiftmain {} {
2    allocate u:A integer
3    allocate_container u:B integer
4    call_composite constructArray [ u:A ] []
5    #  wait until u:A closed
6    rule [ u:A ] foreach:0 [ u:B u:A ]
7  }
8
9  proc foreach:0 { u:B u:A } {
10   #  iterate over array sequentially
11   set dict:A [ enumerate u:A dict ]
12   dict for {v:i u:x} dict:A {
13     container_incr_writers u:B
14     rule [] foreach:0:body [ u:B ]
15   }
16 }
17
18 proc foreach:0:body { u:B u:x v:i } {
19   allocate t:0 integer
20   statement call_composite f [ t:0 ] [ x ]
21   container_insert_imm u:B v:i t:0
22   container_decr_writers u:B
23 }
```

Figure 7: Swift script with foreach loop to be compiled by *stc*, with the corresponding intermediate code. Simple sequential iteration over the loop is shown, but more performant implementations of iteration are possible for large containers or pipelined execution

collection. Various techniques exist for distributed garbage collection [1]; distributed reference counting is the most straightforward candidate for Swift but can be inefficient. Most Swift variables are not needed beyond the lifetime of a procedure stack frame, so we anticipate that escape analysis [5] should be sufficient to keep the reference counting overhead manageable.

In general, static analysis techniques in the compiler will be important for scalability and performance, with efficiency of individual runtime operations playing a secondary role. Often a naive approach to compilation results in severe inefficiencies, such as repeated redundant lookups of variable values or other inefficient usage patterns of the Turbine runtime.

The current Turbine design with processes divided into three roles has been easy to scale up, but it may eventually prove to limit the efficiency of task distribution. Past work on building extremely efficient task-parallel runtime systems (for example, Cilk [8]) has tended to use a symmetrical design, where each process cooperates equally in load balancing and task execution, with work stealing providing load balancing. Shifting to this model could reduce internode communication by keeping tasks and data local except when load balancing occurs.

6 Performance results

In this section, we demonstrate the ability of the ExM task distributor to run a synthetic user application that performs nontrivial script logic. This benchmark uses an algorithm similar to a recursive search and emulates user work at the leaf function calls.

We wrote a Swift script to evaluate the nth Fibonacci number `fib(n)` according to the recursive formulation $\text{fib}(0) = 0$; $\text{fib}(1) = 1$; $\text{fib}(n) = \text{fib}(n-1) + \text{fib}(n-2)$.

In the Swift model, these recursive calls generate a data-dependent workflow to be evaluated among the control flow components in the runtime system. As the workflow progresses, many recursive procedure invocations are triggered, exercising the control flow functions of Turbine. At the base cases $n = 1$ or $n = 0$, leaf tasks sleep for 10 seconds to emulate user computation time.

We ran this benchmark on the IBM Blue Gene/P *Intrepid* at Argonne National Laboratory. Intrepid has 40,960 nodes of 4 cores each. We used varying core counts, with one MPI process per core, with the workload and number of leaf tasks increased by increasing the input parameter n. We obtained a utilization result by dividing the user time (time spent in sleep) by the wall time of the run. Results are shown in Table 2.

Table 2: Detailed statistics for `fib` runs

Cores	n	Leaf Tasks	Time (s)	Util.
4,096	23	46,368	129.0	87.6%
8,192	26	196,418	168.7	87.8%
16,384	27	317,811	217.9	89.0%
32,768	29	832,040	284.1	89.3%
65,536	30	1,346,269	233.3	88.0%

7 Conclusion and future work

We have motivated and described the ExM distributed execution model for running Swift dataflow applications. Swift makes it easy to express massive coarse-grained parallelism, and ExM can execute Swift applications with extreme scalability. While much work remains to complete and validate the full Swift language on ExM and to achieve exascale performance targets, the system will soon be capable of supporting real scientific applications. We will use current petascale systems to extend testing to the 160K core range and to simulate ExM's performance at over 1M core concurrency. We believe ExM's many-task execution model and distributed hierarchical data model makes it well suited to address the resilience and energy-aware load balancing that will be required at the exascale. We will evalute these potential benefits as the implementation proceeds.

Acknowledgments

This work was supported in part by the U.S. Department of Energy under the ASCR X-Stack program (contract DE-SC0005380) and contract DE-AC02-06CH11357. Computing resources were provided by the Mathematics and Computer Science Division and Argonne Leadership Computing Facility at Argonne National Laboratory.

References

[1] ABDULLAHI, S. E., AND RINGWOOD, G. A. Garbage collecting the internet: a survey of distributed garbage collection. *ACM Comput. Surv. 30* (September 1998), 330–373.

[2] ALLAN, B. A., ARMSTRONG, R., ET AL. A component architecture for high-performance scientific computing. *Int. J. High Perform. Comput. Appl. 20* (May 2006), 163–202.

[3] ARVIND, NIKHIL, R. S., AND PINGALI, K. K. I-structures: data structures for parallel computing. *ACM Trans. Program. Lang. Syst. 11* (October 1989), 598–632.

[4] BOCCHINO, JR., R. L., ADVE, V. S., ADVE, S. V., AND SNIR, M. Parallel programming must be deterministic by default. In *Proc. First USENIX Conference on Hot Topics in Parallelism* (Berkeley, CA, USA, 2009), HotPar'09, USENIX Association, pp. 4–4.

[5] CHOI, J.-D., GUPTA, M., SERRANO, M., SREEDHAR, V. C., AND MIDKIFF, S. Escape analysis for Java. *SIGPLAN Not. 34* (October 1999), 1–19.

[6] COSTA, L. B., AL-KISWANY, S., LOPES, R. V., AND RIPEANU, M. Assessing data deduplication trade-offs from an energy perspective. In *Workshop on Energy Consumption and Reliability of Storage Systems ERSS* (2011), pp. 16–19.

[7] DINAN, J., KRISHNAMOORTHY, S., LARKINS, D. B., NIEPLOCHA, J., AND SADAYAPPAN, P. Scioto: A framework for global-view task parallelism. *International Conference on Parallel Processing 0* (2008), 586–593.

[8] FRIGO, M., LEISERSON, C. E., AND RANDALL, K. H. The implementation of the Cilk-5 multithreaded language. *SIGPLAN Not. 33* (May 1998), 212–223.

[9] ISARD, M., BUDIU, M., YU, Y., BIRRELL, A., AND FETTERLY, D. Dryad: distributed data-parallel programs from sequential building blocks. *SIGOPS Oper. Syst. Rev. 41* (March 2007), 59–72.

[10] ISKRA, K., ROMEIN, J. W., YOSHII, K., AND BECKMAN, P. ZOID: I/O-Forwarding infrastructure for petascale architectures. In *Proc. Principles and Practice of Parallel Programming* (2008).

[11] LUSK, E. L., PIEPER, S. C., AND BUTLER, R. M. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review 17* (January 2010), 30–37.

[12] MAHESHWARI, K., WOZNIAK, J. M., ESPINOSA, A., KATZ, D., AND WILDE, M. Flexible cloud computing through Swift Coasters. In *Proc. Cloud Computing and its Applications* (2011).

[13] MCCOOL, M. D. Structured parallel programming with deterministic patterns. In *Proc. HotPar* (2010).

[14] MESSAGE PASSING INTERFACE FORUM. MPI: A message-passing interface standard, 1994.

[15] MURRAY, D. G., AND HAND, S. Scripting the cloud with Skywriting. In *HotCloud '10: Second USENIX Workshop on Hot Topics in Cloud Computing* (Boston, MA, USA, June 2010), USENIX.

[16] MURRAY, D. G., SCHWARZKOPF, M., SMOWTON, C., SMITH, S., MADHAVAPEDDY, A., AND HAND, S. CIEL: a universal execution engine for distributed data-flow computing. In *Proc. NSDI* (2011).

[17] NIEPLOCHA, J., HARRISON, R. J., AND LITTLEFIELD, R. J. Global arrays: A nonuniform memory access programming model for high-performance computers. *Journal of Supercomuting 10*, 2 (1996), 169–189.

[18] PARISIEN, M., SOSNICK, T. R., AND PAN, T. Systematic prediction and validation of RNA-protein interactome. Poster, RNA Society, Kyoto, June 2011.

[19] Partitioned Global Address Space Languages. `http://pgas.org`.

[20] RAICU, I., ZHANG, Z., WILDE, M., FOSTER, I., BECKMAN, P., ISKRA, K., AND CLIFFORD, B. Toward loosely coupled programming on petascale systems. In *2008 ACM/IEEE Conference on Supercomputing* (Piscataway, NJ, 2008), SC '08, IEEE Press, pp. 22:1–22:12.

[21] SARKAR, V., ET AL. ExaScale software study: Software challenges in extreme scale systems. DARPA Report, 2009.

[22] WALKER, E., XU, W., AND CHANDAR, V. Composing and executing parallel data-flow graphs with shell pipes. In *Workshop on Workflows in Support of Large-Scale Science at SC'09* (2009).

[23] WILDE, M., FOSTER, I., ISKRA, K., BECKMAN, P., ZHANG, Z., ESPINOSA, A., HATEGAN, M., CLIFFORD, B., AND RAICU, I. Parallel scripting for applications at the petascale and beyond. *Computer 42*, 11 (2009), 50–60.

[24] WILDE, M., HATEGAN, M., WOZNIAK, J. M., CLIFFORD, B., KATZ, D. S., AND FOSTER, I. Swift: A language for distributed parallel scripting. *Parallel Computing 39*, 9 (September 2011), 633–652.

[25] WOZNIAK, J. M., ARMSTRONG, T. G., LUSK, E. L., KATZ, D. S., WILDE, M., AND FOSTER, I. T. Turbine: A distributed memory data flow engine for many-task applications. In *1st international workshop on Scalable Workflow Enactment Engines and Technologies: SWEET'12* (2012).

[26] WOZNIAK, J. M., AND WILDE, M. Case studies in storage access by loosely coupled petascale applications. In *Proc. Petascale Data Storage Workshop at SC'09* (2009).