Abstract

We present here the ExM (extreme-scale many-task) programming and execution model as a practical solution to the challenges of programming 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 $O(10^6)$ threads on $O(10^5)$ 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.](image1)

![Figure 2: Swift implementation of modFTDock.](image2)

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.

<table>
<thead>
<tr>
<th>Task</th>
<th>Number of Tasks</th>
<th>Duration (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>modftdock</td>
<td>1,200,000</td>
<td>1,000</td>
</tr>
<tr>
<td>merge</td>
<td>12,000</td>
<td>5</td>
</tr>
<tr>
<td>score</td>
<td>12,000</td>
<td>6,000</td>
</tr>
</tbody>
</table>
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).

```swift
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/link 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:

```swift
(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.

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

```swift
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.

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 possible.

![Figure 3: Task distribution in the Turbine runtime system.](image)
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 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 collection.

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

Figure 6: Generated Turbine intermediate code for array operations, with corresponding Swift lines in comments.
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 $n$th Fibonacci number $\text{fib}(n)$ 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

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

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 evaluate 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


