COVER FEATURE

Parallel Scripting for Applications at the Petascale and Beyond

Michael Wilde, Zhao Zhang, Ben Clifford, Mihael Hategan, Sarah Kenny, Kamil Iskra, Pete Beckman, and Ian Foster, *University of Chicago and Argonne National Laboratory*
Ioan Raicu and Allan Espinosa, *University of Chicago*

Scripting has accelerated and simplified programming by focusing on the composition of existing codes to form more powerful applications. Parallel scripting makes it possible to quickly develop highly parallel applications that run efficiently on a 16-core workstation, a 16,000-core cluster, or a 160,000-core petascale system.

John Ousterhout aptly characterized scripting as "higher-level programming for the 21st century."[1] Scripting has revolutionized application development on the desktop and server, accelerating and simplifying programming by focusing on the composition of programs to form more powerful applications. Understanding how to scale scripting to 21st-century computers should be a priority for researchers of next-generation parallel programming models. Might scripting not provide the same benefits for extreme-scale computers as it does on the workstation and server?

We believe that the answer to this question is yes. Scripting languages let users assemble sophisticated application logic quickly by composing existing codes. In *parallel scripting*, users apply parallel composition constructs to existing sequential or parallel programs. This approach makes it possible to quickly develop highly parallel applications that run efficiently on a 16-core workstation, a 16,000-core cluster, or a 160,000-core petascale system.

PARALLEL SCRIPTING

The application focus for extreme-scale machines is on applications with intensive tightly coupled floating-point parallel workloads—for example, finite-element models, computational fluid dynamic codes, adaptive mesh refinement, and partial differential equations. However, there is a need to run existing software applications at large scale with many, and perhaps an extreme number of, copies running in parallel. Each such application could itself be a parallel message passing or multithreaded code.

Parallel scripting is not a substitute for existing tightly coupled programming models such as the message passing interface (MPI). Rather, it is an alternative (and higher-level) path to massive parallelism, a path particularly suitable for increasingly feasible and important problem-solving methods such as the use of parameters sweeps and ensemble studies for exploring sensitivity to parametric, structural, and initial condition uncertainty. The availability of extreme-scale computers makes such methods feasible and attractive, even in the case of complex computations. Parallel scripting lets users apply these methods while leveraging the vast value embodied in modern application codes—both serial and parallel—that empower the scientific, engineering, and

commercial computing of today and the foreseeable future.

We have been exploring such many-task computing models[2] for several years, from the perspective of both technologies and applications. On the technology front, we have developed a dataflow-driven parallel programming model that treats application programs as functions and their datasets as structured objects mapped to a simple abstract data model. We have incorporated this model in Swift, a parallel scripting language, and implemented that language on large parallel computers, including a 160,000-core IBM Blue Gene/P (BG/P) and a 62,000-core Sun Constellation. Swift programs can define hundreds of thousands—and soon millions—of tasks that read and write an even greater number of files. We have developed task and data management methods that can scale to extremely high dispatch rates and data volumes, and used them to scale applications to up to 160,000 cores, with high efficiency and fault tolerance.

PETASCALE PARALLEL SCRIPTING ARCHITECTURE

Figure 1 shows a three-layer software architecture that supports high-performance parallel scripting. From the bottom up, it includes: ZeptoOS, a Posix-compliant operating system that provides basic script execution; Falkon, a resource provisioner and scheduler that allocates and holds compute resources across tasks for long and varying periods of time (typically hours rather than seconds) and that provides efficient and rapid scheduling of short (typically fractions of seconds to seconds or minutes) independent tasks; and Swift, which allows the expression of abstract graphs of highly parallel application invocations, their data accesses, and the data interchange between them.

Figure 1. Architecture for petascale scripting support. ZeptoOS provides basic script execution; Falkon allocates and holds compute resources across tasks for long and varying periods of time and provides efficient and rapid scheduling of short independent tasks; and Swift allows the expression of abstract graphs of highly parallel application invocations, their data accesses, and the data interchange between them.

ZeptoOS

Modern scripting concepts depend on Posix system services such as fork() and exec() to execute application programs from the script, and this requires an OS that supports these or similar capabilities—notably, the ability to launch a new application program and wait for it to complete. Some large-scale petascale systems, such as the BG/P using the native IBM compute node kernel, lack these features. On the BG/P, we provide these features through the ZeptoOS compute node kernel,[4] which implements them in a Posix-compliant manner. On other large systems such as the Constellation, we use the native compute node OS that provides complete Posix.

ZeptoOS could also be used on petascale systems such as the Cray XT5, which provides a full-featured Posix compute node kernel but lacks dynamic linking, a feature helpful for the easy installation of many applications.

Falkon

To leverage the benefits of existing application programs as functions, a petascale computing system must be able to handle cases with fairly short run times. For example, running 60-second tasks on 160,000 cores requires a dispatch rate of more than 2,700 tasks per second to fully utilize the machine.

Figure 2. The Falkon provisioning and scheduling system uses a distributed, hierarchical architecture with a suitably lightweight protocol designed to reduce latency between tasks. It allocates a compute resource's nodes in large quantities, using the system's native batch scheduler, and runs a persistent task execution agent on each compute core that rapidly executes arbitrary and independent Posix processes on the allocated nodes.

The Falcon (Fast and Lightweight tasK execution) scheduling and provisioning system, shown in Figure 2, meets this requirement by using a distributed, hierarchical architecture with a suitably lightweight protocol designed to reduce latency between tasks. At small scales of hundreds of processor cores, tasks can be subsecond long with good efficiency; at medium scales of thousands of cores, second-long tasks can be efficiently run; at the largest scales we have tested—160,000 processor cores—even with tasks as short as 60 seconds, Falkon can obtain high utilization of the

entire processing complex.

The compute node resources of petascale computing systems are typically managed by traditional batch schedulers, which are designed and configured with policies for running large parallel jobs that execute the same application program on all compute nodes allocated to the job, and which run for extended periods of time. Parallel scripting, however, requires that many application programs, each with an independent set of arguments and different sets of input and output files, and having likely short and often widely varying execution times, be executed on any compute node. This far more dynamic model demands a multilevel scheduling approach, which we have implemented in the Falkon system.[3]

Falkon allocates a compute resource's nodes in large quantities, using the system's native batch scheduler, and runs a persistent task execution agent on each compute core that rapidly executes arbitrary and independent Posix processes on the allocated nodes. It accomplishes this using several components: a compute node agent that executes one task at a time on a compute node core; a service that maintains a queue of jobs for a set of compute node resources, and which rapidly selects the next job to run in a first-in, first-out (FIFO) order; and a load-balancing client that evenly distributes work to the services.

Using Falkon, we have been able to meet performance requirements necessary for petascale scripting. BG/P measurements indicate it can execute over 3,000 tasks per second and process tasks at 160,000-core scale.[3]

Falkon users create simple scripts that contain a list of tasks to execute, with arguments. Tasks are executed with FIFO scheduling and maximum parallelism. Because Falcon sends tasks to available agents in the order they appear in the input script, users can sort tasks "longest first" when task duration can be estimated, thereby achieving optimal utilization of a block of compute nodes.

We recently integrated Falkon functionality, in the form of *coasters*, directly into Swift's "execution provider" plug-in framework. These include extensions for automatic deployment on distributed and remote environments and for dynamic block allocation, which enables the lightweight scheduler to batch the demand for CPUs into efficient units of varying sizes to present to the local resource manager's scheduler.

This capability is designed to handle scripts in which the demand for CPUs can fluctuate throughout the execution, due to either the shape of the script's data dependency graph—for example, when reduction jobs are processed—or due to widely varying task execution length. Tasks that exceed the allocation of larger, short-duration blocks can be reexecuted within smaller, long-duration blocks, enabling users to apply many scheduling optimizations and tradeoffs. This will enable excellent scalability and competition for resources on shared petascale machines such as Ranger at the University of Texas at Austin and the University of Tennessee's Kraken, which will soon dominate the TeraGrid's Extreme Data expansion.

Swift

The focus of our current parallel scripting efforts, Swift[4] comprises a

- high-level, functional scripting language, designed for expressing computations that invoke executable programs, with a dataflow model used to ensure that program invocations ("tasks") are executed only when their input data is available;
- data model that allows for the mapping of file system structures—individual files, directories, and so on—into Swift language variables; and
- runtime system to manage the scheduling of tasks for execution, the dispatch of executable tasks to parallel computers, and the movement of the data that those tasks consume and produce.

Swift makes it easy to specify and run application programs in parallel, and to specify the data passing and data dependencies between different application invocations as well as the structure of the data, typically in terms of the files and directories that these applications produce and consume. It provides runtime support that automates data management and enables the same Swift script to run on multicore workstations, local clusters, remote grids, cloud resources, and petascale supercomputers.

It is common to write Swift applications in three layers: a top layer that specifies the workflow, a middle layer of interface code to adapt specific applications, and a bottom layer that defines the interfaces of the applications themselves. Studies on early versions of Swift indicate that the amount of code needed to express applications in this form is substantially lower than by ad hoc scripting in shell scripts.[5]

Swift also provides a provenance recording mechanism that enables users to log how each data item was produced, query that knowledge base to locate data and methods, and retrieve the history of an object for validation, sharing, or reproduction of computational results.

The Swift data model provides the ability to describe nested on-disk directories as simple structures and arrays. The system transparently sends these datasets to remote and parallel Swift procedures on various platforms. A *mapping* operation translates between Swift's simple abstract data model and the potentially messy, complex model of real-world directory structures and file naming and structuring conventions.

Swift has a C-like syntax but also has many of the semantic aspects of a functional programming language. Procedures are expressed as functions, which can return multiple values; statements are executed in data-dependency order; variables including array elements and structure members are single assignment, which makes it significantly simpler for Swift to automatically execute independent operations in parallel. Application programs are abstracted as functions, for which the arguments and results are files and file-structured datasets. Layering on the ability to represent application programs as procedures, the user can define compound procedures to create libraries of higher-level processes that capture the essential protocols of an application domain's data preparation and analysis procedures.

Users interact with Swift at many levels. Scientific programmers create Swift libraries that encapsulate the execution of scientific applications, data preparation, and analysis methods. These libraries provide a stable base of functionality specific to a user community. Higher-level users write simple scripts using these libraries to perform large-scale computing tasks. The highest-level Swift users will not need to program at all; they will invoke their scripts and view the results through Web interfaces.

The following example, abstracted from protein structure prediction scripts, illustrates some basic Swift constructs:

```
1. app (PDT structure) predict(Fasta protein) {
2.         predict_structure @protein @structure;
3. }
4.
5. foreach pfile, i in @arg("proteins") {
6.       Fasta protein <pfile>;
7.       PDT structure[];
8.       structure[i] = predict(protein);
9. }
```

Lines 1-3 define an interface to an application program, predict_structure. This interface maps from typed Swift variables in the header of function predict() (line 1) to command-line program syntax (line 2). The function expects a protein sequence specified in FASTA format and returns a structure prediction in the form of a trajectory file in PDT format. In lines 5-9, the script invokes the procedure predict(), and thus the predict_structure application program, in parallel, for each file listed in the command line argument "proteins".

Swift's dataflow model ensures that the multiple invocations of predict() can run concurrently, as none is dependent on data produced by another. Swift's runtime system handles the dispatch of individual predict() calls to an available CPU or computer, and the movement of the associated data to and from those computers. Thus, just eight lines can describe a potentially large amount of computing.

The Swift runtime system can deal with computations that involve many concurrent activities. By using two-level scheduling methods, as implemented for example in Falkon, it has executed computations involving hundreds of thousands of tasks on supercomputers with tens of thousands of processors. (These methods first deploy task executors onto nodes and then stream tasks to those

executors.) The Swift runtime system can also dispatch tasks to multiple computers, using Globus and other distributed computing abstractions to overcome inevitable heterogeneities in authentication, job submission, and data movement methods. Swift implements various heuristics to decide where to send which task and when.

Collective data management

A straightforward implementation of parallel scripting uses parallel file systems and thus places a high burden on a globally addressable storage infrastructure such as IBM's General Parallel File System (GPFS). However, great speedups have been achieved by carefully tuning all file system accesses. Our solution to this problem, *collective data management* (CDM),[6] is inspired by collective I/O primitives from MPI; however, unlike collective I/O, which operates at the I/O level, CDM operates at the file level.

CDM enables efficient and easy distribution of input data files to computing nodes and aids in gathering output results from them. It eliminates the need for manual tuning and makes the programming of large-scale clusters using a loosely coupled model easier. CDM leverages fast local file systems to provide high-speed local file caches for parallel scripts, uses a broadcast approach to handle distribution of common input data, and employs efficient scatter/gather and caching techniques for input and output.

The CDM model applies the following concepts and heuristics:

- use collective networks and primitives to broadcast common input data to compute nodes;
- manage the use of compute node space as script execution proceeds, removing large files no longer needed;
- aggregate compute node space into larger file systems, which can leverage the high-performance interconnect to deliver data to applications;
- read datasets from large parallel file systems in efficient units into local disks, from where unmodified application programs can operate with suitable performance;
- collect datasets going back to permanent storage on parallel file systems into suitably large files; and
- write datasets back to parallel file systems in independent directories to avoid metadata contention.

Our experience is largely based on specific BG/P operating environment considerations—the BG/P interconnect architecture with a separate collective network, ZeptoOS compute node kernels with I/O forwarding, and GPFS with full multiprocessor data consistency guarantees. However, most if not all of these considerations apply to the majority of deployed petascale systems as they all run some form of parallel file system—for example, GPFS, Lustre, the Parallel Virtual File System (PVFS)—and all have some form of exotic and often hierarchical or heterogeneous network interconnects—for example, a mix of torus, tree, or Clos networks. CDM is currently in its experimental stages, and we are applying its concepts to both Swift and Falkon scripted applications through the explicit insertion of CDM primitives and heuristics.

PARALLEL SCRIPTING APPLICATIONS

Table 1 lists several scripting applications performed in cluster or grid environments that are candidates for execution on petascale systems.

Table 1. Example parallel scripting applications.

Field	Description	Characteristics	Status
Astronomy	Creation of montages from many digital images	Many one-core tasks, much communication, complex dependencies	Experimental
Astronomy	Stacking of cutouts from digital sky	Many one-core tasks, much	Experimental

	surveys	communication	
Biochemistry*	Analysis of mass-spectrometer data for posttranslational protein modifications	10,000-100 million jobs for proteomic searches using custom serial codes	In development
Biochemistry*	Protein folding using iterative fixing algorithm; exploring other biomoleculer interactions	Hundreds to thousands of 1-1,000-core simulations and data analysis	Operational
Biochemistry*	Identification of drug targets via computational screening	Up to 1 million × 1 core	Operational
Bioinformatics*	Metagenome modeling	Thousands of one-core integer programming problems	In development
Business economics	Mining of large text corpora to study media bias	Analysis and comparison of 70 million and more text files of news articles	In development
Climate science	Ensemble climate model runs and analysis of output data	Tens to hundreds of 100-1,000-core simulations	Experimental
Economics*	Generation of response surfaces for various economic models	1,000 to 1 million one-core runs (10,000 typical), then data analysis	Operational
Neuroscience*	Analysis of functional MRI datasets	Comparison of images; connectivity analysis with structural equation modeling, many tasks (100,000 and more)	Operational
Radiology	Training of computer-aided diagnosis algorithms	Comparison of images; many tasks, much communication	In development
Radiology	Image processing and brain mapping for neurosurgical planning research	Thousands of MPI application executions	Operational

*Applications being run on petascale machines—currently the Argonne National Laboratory's Blue Gene/P (Intrepid) or the TeraGrid Sun Constellation at the University of Texas at Austin (Ranger).

We have applied large-scale parallel scripting to several applications.[7-9] Each scripted application can consume a large fraction, or even all, of a petascale computer. All involve executing many tasks at once, with often quite substantial amounts of communication both within each task and among tasks.

Molecular docking

In the pharmaceutical domain, the DOCK 6 application is regularly run on the Argonne National Laboratory's BG/P, Intrepid, to simulate the docking of small ligand molecules to the active sites of large macromolecules called receptors. A compound that interacts strongly with a receptor, such as a protein molecule, associated with a disease may inhibit its function and thus act as a beneficial drug.

This application is challenging as there are many tasks, each with a wide range of execution times, and each computation involves significant I/O. Protein description files for docking range from tens to hundreds of megabytes and must be read for each computation.

Argonne biochemists use Falkon via Perl for molecular docking and surface screening, running at scales of up to 64,000 cores in a single scripted workload.

Uncertainty in economic models

The University of Chicago-Argonne Project on Social, Economic, and Environmental (SEE) modeling uses Swift on petascale systems to execute parameter sweeps of economic models of

energy usage to examine the effects of uncertainty. SEE researchers use the parallel scripting paradigm to refine several models for exploring uncertainty through large-scale parallelism.

Figure 3 shows the results of a parallel script exploring the implications of calibration dataset error; researchers analyzed 2,000 samples from a perturbed input dataset in parallel on Ranger and other systems. The model evaluates a global "business as usual" scenario with midsized aggregation for studying carbon leakage.

Figure 3. Example energy-economics model parameter sweeps. These graphs show sample results (for the US and China) of a parallel script exploring the implications of calibration dataset error; researchers analyzed 2,000 samples from a perturbed input dataset in parallel on Ranger and other systems. The model evaluates a global "business as usual" scenario with midsized aggregation for studying carbon leakage.

Structural equation modeling

The University of Chicago's Human Neuroscience Laboratory has developed a computational framework for a data-driven approach to structural equation modeling (SEM) and implemented several parallel scripts for modeling functional magnetic resonance imaging data within this framework. The Computational Neuroscience Applications Research Infrastructure (CNARI) employs Swift to execute hundreds of thousands of simultaneous processes running the R data analysis language, consisting of self-contained structural equation models, on Ranger. These self-contained R processing jobs are data objects generated by OpenMx, a plug-in for R that can generate a single model object containing the matrices and algebraic information necessary to estimate the model's parameters. With a framework like CNARI, structural modelers may begin to investigate exhaustive searches of the model space.

Posttranslational protein modification

The University of Chicago's Ben May Department for Cancer Research is applying petascale parallel scripting to the analysis of posttranslational protein modifications (PTMs), complex changes to proteins that play essential roles in protein function and cellular physiology. The new PTMap tool takes in raw data files from mass-spectrometry analysis of biological samples, the entire protein sequences of the organism's genome, and searches them for statistically significant evidence of unidentified PTMs. The tool reads in a mass-spectrometry file—typically 200 Mbytes of data in mzXML format—and one or more protein sequences from the genome in FASTA format.

The overall task of analyzing a mass-spectrometry run for a single genome has abundant opportunities for parallelization at the extreme scale. Researchers want to apply the latest version of PTMap to identify unknown PTMs across a wide range of organisms including *E. coli*, yeast, and the cow, mouse, and human genomes.

PARALLEL SCRIPTING CASE STUDY

University of Chicago researchers run the Open Protein Simulator (OOPS), an application that predicts 3D protein structure by running an iterative fixing algorithm (ItFix) that does potentially thousands of Monte Carlo annealing simulations of protein moves, with energy minimization. OOPS has been run in parallel on up to 64,000 Intrepid cores, as well as on Ranger and other TeraGrid systems.

A few excerpts from the ItFix program structure illustrate how Swift works in OOPS. The script's main functions are to call ItFix directly, predict a single protein's structure, or build up more complex programs. For example, the following program runs each of a set of protein sequences (from file plist) in up to maxrounds rounds:

```
main_loop()
{
    int nsim = @toint(@arg("nsim"), 3);
    int maxrounds = @toint(@arg("maxrounds", "3"));
    string protein[] = readData(@arg("plist"));
    foreach prot in protein {
      ItFix(prot, nsim, maxrounds,"","");
    }
}
```

This simple code fragment, given 10 proteins and nsim = 1,000, would execute 10 × 1,000 = 10,000 simulations in each of up to three prediction rounds. Swift runtime settings and processor resource availability determine the actual degree of parallelism.

With our library of OOPS Swift procedures, protein structure researchers can use flexible scripts to leverage many processors with relative ease, as in the following parameter sweep script:

```
int nsim = @toint(@arg("nsim"), 3);
int maxrounds = @toint(@arg("maxrounds", "3"));
string protein[] = readData(@arg("plist"));
string startT[] = readData(@arg("startT"));
string tUpdate[] = readData(@arg("tUpdate"));
foreach prot in protein {
  foreach sT in startT {
    foreach tUp in tUpdate {
      ItFix(prot, nsim, maxrounds, sT, tUp);
    }
  }
}
```

Given 10 proteins, nsim = 1,000, two starting temperatures, and five update intervals, this script would execute 10 × 1,000 × 2 × 5 = 100,000 simulations in each of up to three prediction rounds. On highly parallel systems such as Intrepid, this simple code fragment can fully utilize a substantial portion of the machine's 163,840 processor cores. Similar code with a slightly more general parameterization of ItFix can sweep across any combination of settable parameters that govern the OOPS structure prediction algorithm.

Figure 4. Results of running eight proteins on two racks (8,192 CPUs) of Intrepid. The scatter plot on the left indicates the correlation between statistical energy potential and protein structure accuracy for the 985 simulations that ran to completion. The image on the right shows the lowest root mean square deviation structure.

Figure 4 shows results of running OOPS under Swift for protein T1af7. The scatter plot on the left indicates the correlation between statistical energy potential and protein structure accuracy for the 985 simulations that ran to completion. The image on the right shows the lowest root mean square deviation structure. Our parallel scripts automatically generated the plot and image as well as the table, which is presented by a simple CGI script at our website.[10]

In the first two weeks of April 2009, just shortly after development of the ItFix Swift script, the system saw impressive use: 67,178 structure predictions, totaling 208,763 CPU hours, on Intrepid; and 17,488 jobs, totaling 1,425 CPU hours, on Ranger. The same scripts were used to perform 22,495 predictions totaling 2,397 CPU hours. The Intrepid runs alone produced more than 100 gigabytes of compressed protein folding trajectory data.

PARALLEL SCRIPTING MODEL PERFORMANCE

DOCK application performance measurements[3] indicate that on Intrepid, Falkon can

- execute more than 3,000 tasks per second;
- launch, execute, and terminate 160,000 tasks at 160,000-core scale in under one minute;
- execute workloads of 913,000 science tasks on 116,000 BG/P cores in two hours, totaling 21.4 CPU years at 99.7 percent efficiency and 99.6 percent utilization; and
- execute one billion trivial tasks in 18 hours in multicore stress tests.

As Figure 5a shows, these measurements meet the performance requirements necessary for petascale scripting.

Figure 5. Parallel scripting model performance: (a) DOCK application performance using Swift on Intrepid; (b) neuroscience SEM application performance using Swift on Ranger; (c) PTMap application performance on Intrepid.

Figure 5b shows performance results of running SEM scripts at large scale using Swift on Ranger to model neural pathway connectivity from experimental fMRI data. The left figure shows the active processes of a four-region SEM workflow over two experimental conditions. The red line represents the execution of jobs on Ranger; the blue and green lines represent the staging in and out of files, respectively, from a client application. The right figure shows the active processes of a four-region SEM workflow over multiple networks.

We performed some preliminary measurements of PTMap application performance under modest scales. Specifically, we ran the stage 1 processing of the *E.coli* K12 genome (4,127 sequences) on 2,048 Intrepid processor cores. Figure 5c shows the summary of this run; the left figure shows processor utilization—green indicates busy and red idle—as time progresses; the middle figure shows each of the 4,127 tasks as a horizontal line, where the line's length shows the task execution time; and the right figure shows a summary per processor core, where each task executed is denoted by a horizontal line indicating its execution length and is mapped to the processor core on which it executed. Overall, the average per task execution time was 64 seconds, with a standard deviation of 14 seconds. These 4,127 tasks consumed a total of 73 CPU hours, in a span of 161 seconds on 2,048 processor cores. This represents 80 percent utilization.

RELATED WORK

MapReduce—and the open source Hadoop and Sphere—are libraries that support parallel processing and reduction operations on large distributed datasets. The MapReduce paradigm is primarily oriented toward the distributed execution of a specific program that explicitly calls MapReduce library functions. In contrast, Swift and Falkon focus on the execution of arbitrary application program executables that can then be composed into larger applications. In addition, Swift supports mapping, dynamic resource pools, and task-parallel constructs, and can express MapReduce-style workflows for application programs in a simpler-to-use fashion on a broader set of larger-scale platforms.

MPI can be used to express some parallel workflows. However, it is less well-suited for the dynamic environments and applications at which Swift excels. We view Swift and MPI as complementary; we often use Swift to coordinate the execution of MPI applications. Similar observations apply to the commercial Star-P parallel Matlab, which can execute parallel Matlab programs as well as Python scripts on a single cluster.

MPI is quite rigid and will have issues—none insurmountable—in scaling to millions of processors and beyond, due to shorter and shorter mean time to failure as these machines grow in size. Swift and Falkon are more flexible, as failures are typically localized in a rerunnable compute node task and would not cause the entire application to fail.

A recent MPI innovation is the Asynchronous Dynamic Load Balancing library.[11] ADLB moves MPI programming closer to the loosely coupled Swift model, in that tasks are freed from the restrictions of two-sided communication and execute in a manner similar to the traditional master-worker model. It is still, however, a model for executing in-memory tasks, unlike the Swift model of executing independent programs linked by file exchange.

It is also useful to compare Swift to the evolving class of languages such as Chapel.[10] Swift and Chapel share the same goal of programming productivity; however, Chapel is oriented toward in-memory computing, while Swift focuses on loosely coupled application program coordination. Nonetheless, there are parallels between the two languages. Like Chapel, Swift is a "global view" rather than a "fragmented model" programming language, in which the compiler and runtime system determine a program's mapping to the available runtime parallel resources. Like Chapel's forall statement, Swift's foreach statement determines a parallel execution strategy for the programmer, without the explicit task assignment of MPI-style fragmented models. Swift is also strongly typed like Chapel, though it offers the programmer fewer ways to circumvent the typing model and lacks Chapel's semantics for type inference.

Compared with other approaches, the Swift parallel scripting programming model coupled with the Falkon resource provisioner and scheduler is unique in that it provides a simple science-friendly scripting language with abstract data types, mapping, and multiple levels of abstract; orchestrates the execution of virtually any scientific application program; and supports most of the supercomputer architectures that serve the scientific community: the IBM BG/P, the Sun

Constellation, and soon the Cray XT5.

Swift uses a functional model to rapidly and productively compose new, higher-level applications that can efficiently draw on the parallel resources of a wide range of systems from gigascale multicore computers to terascale clusters to petascale supercomputers.

Ongoing research continues to yield improvements in performance and programming productivity. We are experimenting on hybrid applications with parallel scripting at the top level, MPI at the midlevel, and OpenMP or pthreads for efficient multicore parallelism. A major effort is under way to integrate CDM semantics into the Swift compiler and runtime system in an automated manner to efficiently move data in and out of applications in a way that leverages the interconnect and file-system hardware of petascale systems.

A PTMap analysis of a single mass spec run on the human genome requires a level of computing that illustrates the benefits of extreme scripting. The human genome used for this experiment contains 80,000 protein sequences. The tandem mass-spectrometry run from human samples that we are evaluating contains 51 mzXML files, each about 200 Mbytes. The two-stage PTMap workflow will involve 8 million executions of the PTMap application and about 1.1 million node hours of BG/P time. With a sustained resource allocation of 16,000 cores, this will require three solid days of running time. Leveraging the entire Intrepid system, we could perform a PTMap search of the human genome for a large mass-spectrometry sample in less than seven hours.

We believe that, when complete, the parallel scripting model will play an indispensable role in the extreme-scale programming tool chest. ■

Acknowledgments

We thank Yingming Zhao and Yue Chen for contributing and collaborating with us on the PTMap application; Andrew Binkowski and Mike Kubal on the DOCK application; Joshua Elliott, Meredith Franklin, and Todd Munson on the SEE application; Steve Small, Michael Andric, and the OpenMx project team on the SEM application; and Glen Hocky, Joe DeBartolo, Karl Freed, and Tobin Sosnick on the OOPS application. This research is supported in part by NSF grant OCI-721939, NIH grants DC08638 and DA024304-02, U.S. Dept. of Energy under contract DE-AC02-06CH11357, NASA Ames Research Center GSRP grant NNA06CB89H, and the University of Chicago/Argonne National Laboratory Computation Institute.

References

1. J.K. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," *Computer,* Mar. 1998, pp. 23-30
2. I. Raicu, I.T. Foster, and Y. Zhao, "Many-Task Computing for Grids and Supercomputers," *Proc. 2008 IEEE Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS 08), IEEE Press, 2008; http://people.cs.uchicago.edu/~iraicu/publications/2008_MTAGS08_MTC.pdf.
3. I. Raicu et al., "Toward Loosely Coupled Programming on Petascale Systems," article no. 22, *Proc. 2008 IEEE/ACM Conf. Supercomputing* (SC 08), IEEE Press, 2008.
4. Y. Zhao et al., "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," *2007 IEEE Congress on Services,* IEEE Press, 2007, pp. 199-206.
5. Y. Zhao et al., "A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data," *ACM SIGMOD Record,* Sept. 2005, pp. 37-43.
6. Z. Zhang et al., "Design and Evaluation of a Collective I/O Model for Loosely-Coupled Petascale Programming," *Proc. 2008 IEEE Workshop on Many-Task Computing on Grids and Supercomputers* (MTAGS 08), IEEE Press, 2008; **[Author: Page numbers or link to PDF of article?]**.
7. S. Kenny et al., "Parallel Workflows for Data-Driven Structural Equation Modeling in Functional Neuroimaging," tech. report ANL/MCS-P1613-0409, Argonne National Laboratory, Argonne, Ill.

8. S.L. Small et al., "Database-Managed Grid-Enabled Analysis of Neuroimaging Data: The CNARI Framework," *Int'l J. Psychophysiology,* July 2009, pp. 62-72.
9. T. Stef-Praun et al., "Accelerating Medical Research Using the Swift Workflow System," *Studies in Health Technology and Informatics,* vol. 126, 2007, pp. 207-216.
10. B.L. Chamberlain, D. Callahan, and H.P. Zima, "Parallel Programmability and the Chapel Language," *Int'l J. High Performance Computing Applications,* Aug. 2007, pp. 291-312.
11. P. Balaji et al., "MPI on a Million Processors," *Proc. 2009 European PVM/MPI Users' Group Conf.* (EuroPVM/MPI 2009), CSC-IT Center for Science, 2009; www.mcs.anl.gov/~balaji/pubs/2009/europvm/europvm09.mpimill.pdf.

Michael Wilde *is a software architect in the Mathematics and Computer Science Division, Argonne National Laboratory, and a Fellow at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at wilde@mcs.anl.gov.*

Zhao Zhang *is* ***[Author: Title/position]*** *at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at* ***[Author: e-mail address]****.*

Ben Clifford *is* ***[Author: Title/position]*** *at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at* ***[Author: e-mail address]****.*

Mihael Hategan *is* ***[Author: Title/position]*** *at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at hategan@mcs.anl.gov.*

Sarah Kenny *is* ***[Author: Title/position]*** *at the University of Chicago/Argonne National Laboratory Computation Institute. Contact her at* ***[Author: e-mail address]****.*

Kamil Iskra *is an assistant computer scientist in the Mathematics and Computer Science Division, Argonne National Laboratory, and a Fellow at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at iskra@mcs.anl.gov.*

Pete Beckman *is a computer scientist in the Mathematics and Computer Science Division, Argonne National Laboratory, where he also serves as a division director at the Argonne Leadership Computing Facility, and a Senior Fellow at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at beckman@mcs.anl.gov.*

Ian Foster *is a Distinguished Fellow and associate division director of the Mathematics and Computer Science Division, Argonne National Laboratory, and director of the University of Chicago/Argonne National Laboratory Computation Institute. He is also a professor of computer science at the University of Chicago. Contact him at foster@mcs.anl.gov.*

Ioan Raicu *is a* ***[Author: Title]*** *in the Department of Computer Science at the University of Chicago. Contact him at iraicu@cs.uchicago.edu.*

Allan Espinosa *is a PhD student in the Department of Computer Science at the University of Chicago. Contact him at aespinosa@cs.uchicago.edu.*

Keywords: Parallel scripting, Scientific computing, Software engineering, Extreme-scale computing