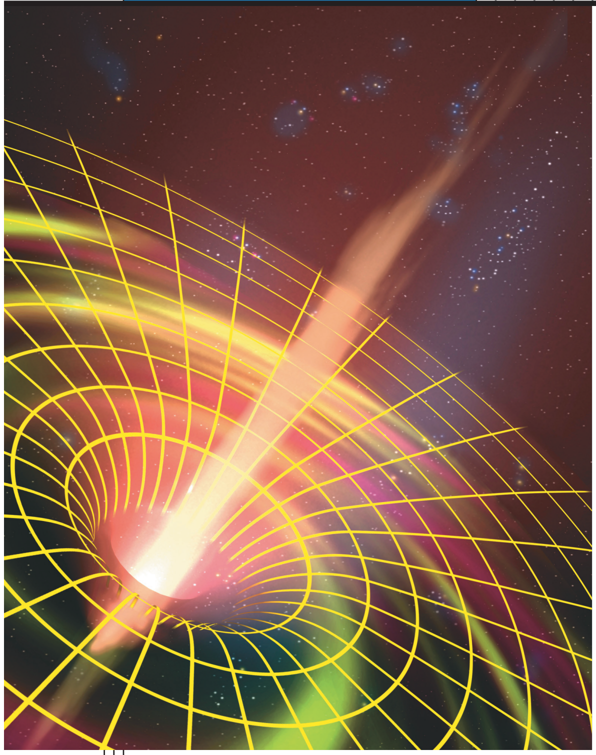

PARALLEL SCRIPTING FOR APPLICATIONS AT THE PETASCALE AND BEYOND

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

Scripting accelerates and simplifies the composition of existing codes to form more powerful applications. Parallel scripting extends this technique to allow for the rapid development of highly parallel applications that can run efficiently on platforms ranging from multicore workstations to petascale supercomputers.

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 allowing programmers to focus on the composition of programs to form more powerful applications.

Might scripting provide the same benefits for parallel computers—including extreme-scale computers—as it does for workstations and servers? We believe that the answer 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. With such methods, programmers can quickly specify highly parallel applications that may, depending on problem scale, require for their execution a 16-core workstation, a 16,000-core cluster, or a 160,000-core petascale system.

Understanding how to scale scripting to 21st-century computers should thus be a priority for researchers of next-generation parallel programming models. In addressing this priority, we have focused on parallel scripting for systems such as the IBM Blue Gene/P (BG/P) and Sun Constellation.

MOTIVATION FOR PARALLEL SCRIPTING

Most research and development on programming models for exascale machines is concerned with tightly coupled single-program, multiple-data (SPMD) applications—for example, computational fluid dynamic codes applied to weather modeling and structural mechanics codes applied to automobile design. Such applications certainly require large amounts of computing power and a high-performance messaging infrastructure.

However, it would be shortsighted to assume that such exascale applications are the only ones that require high-end supercomputers. Our experience suggests a substantial and unmet need to run existing programs at large scale, via the simple expedient of running many copies of programs at once. Each such application may itself be a parallel message-passing, multithreaded, or serial code. Developers of such applications, like developers of SPMD applications, require methods and tools to reduce complexity, enhance reuse, and optimize performance on different platforms. Parallel scripting can provide a basis for such methods and tools.

Example

A simple example illustrates parallel scripting in practice.

It is increasingly common for a weather modeler to run many instances of a model, each with different initial conditions, to quantify forecast uncertainty. In pseudocode, the modeler wants to do something like the following:

```
initial_conditions[ ] = initialize( )
forecast[ ] = null
foreach condition, index in initial_conditions:
   forecast[index] = weather_model(condition)
uncertainty = analyze(forecast)
```

This program first creates an array of files, each comprising a different set of initial conditions for the weather model. Then, it invokes the multiple instances of the weather model proper, using an operator (`foreach`) that performs parallel execution based on available resources. (The weather model runs on many processors; thus, on a small parallel computer, the multiple model invocations may be run one after the other. However, on a large parallel computer, many or all can be run in parallel.) The output from these multiple invocations is stored in a second array of files. The final step analyzes the computed forecasts.

A researcher may wish to explore the sensitivity of the same model to an input parameter, again for a range of initial conditions. This new strategy can be defined via a script that calls the same program in a different manner, this time sweeping over a range of parameters:

```
parameters[ ] = getParameterSets( )
initial_conditions[ ] = initialize()
foreach condition, cindex in initial_conditions:
   foreach parameterSet, pindex in parameters:
      forecast[cindex, pindex] = weather_model
         (parameterSet, condition)
```

Other variants of these simple scripts could select just those runs that generate excessive rainfall, pass their output to a flood model, and/or generate specialized images to highlight unusual conditions. Indeed, parallel scripts can become quite complex. Whether simple or complex, they have in common that they express large amounts of parallelism concisely, via the composition of existing programs that read and write files.

Advantages

As this example shows, parallel scripting is ideal for parameter sweeps and ensemble studies, methods that are increasingly used to explore sensitivity to parametric, structural, and initial condition uncertainty.

Parallel scripting enables developers to build on the codes of today to create the applications of tomorrow on the full spectrum of available parallel systems.

Another important problem class for parallel scripting is data analysis. A parallel script can be a natural tool for both specifying and accelerating the analysis of a large collection of discrete files or database records, particularly in the case of application programs designed to analyze a single file or database record. Biomedical researchers apply this form of parallel scripting, for example, to process images for training computer-aided medical diagnosis algorithms and for research in surgical planning. Starting with programs designed for analyzing single images, they use parallel constructs to create concise scripts capable of rapidly analyzing thousands of such images.

The compelling conclusion from such experiences is that parallel scripting enables developers to build on the codes of today to create the applications of tomorrow on the full spectrum of available parallel systems.

SWIFT: A LANGUAGE FOR PARALLEL SCRIPTING

The framework within which we investigate parallel scripting is the Swift language and system[2] (www.ci.uchicago.edu/swift). Linguistically, Swift blends a C-like syntax with functional programming characteristics. The language is designed to expose opportunities for parallel execution, avoid the unnecessary introduction of nondeterminism, simplify the development of programs that operate on file systems, and permit efficient implementation on distributed-memory parallel computers.

Swift integrates external persistent data—typically contained in files and directories—into the language model, improving the development process for programs that read and/or write large datasets. This integration is achieved via a mapping system that allows files and directories to be represented within programs as typed language variables. Thus, a nested directory structure may be represented in Swift as a nested data structure, permitting a program that operates over all files in those directories to be written as a nested set of `foreach` statements. Similar constructs allow for the definition of typed interfaces to external executables.

Swift reveals opportunities for parallel execution via a combination of explicitly parallel constructs (such as `foreach`) and a dataflow programming model. This model is based on single-assignment variables, a construct that also avoids unnecessary nondeterminism: If one program produces a file that a second program consumes, then Swift ensures that the shared variable representing that file is not assigned a value until the first program has completed execution. As a result of that assignment, the second program then becomes executable. Studies indicate that the amount of code needed to express applications in this form is substantially lower than by ad hoc scripting in shell scripts or less expressive notations such as directed acyclic graphs.[3] The Swift runtime system handles the dispatch of executable tasks to computers and the movement of the data that these programs consume and produce.

Swift integrates external persistent data—typically contained in files and directories—into the language model, improving the development process for programs that read and/or write large datasets.

PARALLEL SCRIPTING CASE STUDY

University of Chicago researchers have developed the Open Protein Simulator,[4] an application that predicts tertiary (3D) protein structure, an important computational problem in biochemistry due to the difficulty of experimental structure determination. Their approach to this problem involves running many instances of a structure prediction simulation, each with different random initial conditions. The simulation uses an "iterative fixing" algorithm[5] (ItFix) that performs multiple "rounds," each involving many parallel Monte Carlo simulated annealing models of molecular moves with energy minimization. After each round, ItFix analyzes the results and picks the best (usually lowest-energy) candidate structure as the basis for the next round, continuing until a convergence criterion is satisfied or a maximum number of rounds have been completed.

This application is a natural candidate for parallel scripting with Swift. Given an external executable `PSim` that computes a single model structure, we want to specify the higher-level structure of the ItFix application. A traditional implementation might involve multiple Bash or Perl scripts to allocate resources, structure on-disk data, and manage the thousands of concurrent tasks. In contrast, the following simplified Swift example of a single ItFix round emphasizes how concise a parallel script can be when using appropriate concepts and constructs:

```
app (ProtGeo pg) predict (Protein pseq)
{
    PSim @pseq.fasta @pg;
}

(ProtGeo pg[]) doRound (Protein p, int n) {
    foreach sim in [0:n-1] {
        pg[sim] = predict(p);
    }
}

Protein p <ext; exec="Pmap", id="1af7">;
ProtGeo structure[];
int nsim = 10000;
structure = doRound(p, nsim);
```

The `app` declaration defines an interface to the `PSim` (Open Protein Simulator) executable. This interface specifies how to map from the typed Swift variables `pg` (protein geometry file) and `pseq` (protein sequence structure) in the header of procedure `predict()` to the command-line program syntax expected by `PSim`. The expressions `@pseq.fasta` and `@pg` insert the filenames mapped to those arguments into the command line. The `predict` procedure expects a protein structure containing a FASTA-format file as its argument and returns a structure prediction in the form of a PDB (Protein Data Bank) file that describes the geometric locations of the protein's atoms in its predicted 3D structure. The `doRound()` procedure performs one "round" of parallel simulations by invoking the `predict()` procedure `n` times in parallel, with each `PSim` invocation executed by `predict()` performing a Monte-Carlo-based structure prediction and returning an array of predictions. The last four statements invoke `doRound` for one protein sequence, running the `PSim` application program 10,000 times in parallel.

Swift's dataflow model enables the multiple invocations of `predict()` to run concurrently, as none depend on data produced by another. Swift's runtime system handles the dispatch of each `predict()` call to an available node and the movement of the associated data to and from that node.

Having thus defined the form of a single round, we can then specify the iterative fixing algorithm proper. We do this as follows, with declarations and parameter lists elided:

```
ItFix( Protein p, int nsim, int maxr,
        float temp, float dt)
{
    ProtSim prediction[][];
    boolean converged[];
    PSimCf config;
    ...
    iterate r {
       prediction[r] =
          doRoundCf(p, nsim, config);
       converged[r] =
          analyze(prediction[r], r, maxr);
    } until ( converged[r] );
}
```

This code fragment uses the Swift `iterate` statement to perform prediction rounds until a convergence criterion has been satisfied or a maximum number of rounds have been performed. The procedure `doRoundCf()` enables science configuration parameters to be passed to the `PSim` application.

Given these Swift procedures, researchers can then use flexible scripts to leverage many processors with relative ease, as in the following parameter sweep script:

```
int nSim = 1000;
int maxRounds = 3;
Protein pSet[] <ext; exec="Protein.map">;
float startTemp[]=[100.0, 200.0];
float delT[]=[1.0, 1.5, 2.0, 5.0, 8.0];
foreach p, pn in pSet {
    foreach t in startTemp {
       foreach d in delT {
          ItFix(p, nSim, maxRounds, t, d);
       }
    }
}
```

Given 10 protein sequences from the external mapper script "Protein.map", nsim = 1,000, two starting temperatures, and five temperature increments (to control the simulated annealing algorithm), 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 the Argonne BG/P Intrepid, this script can use a substantial portion of the machine's 160,000 processor cores. (ItFix has run on up to 64,000 cores on Intrepid.) Similar code with a generalized parameterization of ItFix can sweep across any combination of settable parameters that govern the structure prediction algorithm.

Figure 1 shows results of running ItFix with Swift for eight protein structure predictions that were executed with a Swift script on 8,000 CPUs. The images show the predicted structure of three proteins from the run; the table shows their lowest root mean square deviation from the experimentally known structure, and their improvement over older runs done on clusters with ad hoc scripts ("DeBartolo"). The scatter plot indicates the correlation between statistical energy potential and protein structure accuracy for 985 simulations of protein 1af7. A parallel Swift script performs the predictions and then generates the plots, images, and a statistics summary table, which are made available to researchers via a Web interface.[4]

In the first two weeks of April 2009, 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 TeraGrid Constellation at the University of Texas at Austin. The same scripts were used in that period to perform 22,495 predictions totaling 2,397 CPU-hours on other TeraGrid sites with between 4,000 and 9,000 cores each. The Intrepid runs alone produced more than 100 gigabytes of compressed protein structure trajectory data.

Protein	Length	ST	TUI	Lowest RMSD (Å, BG/P)	Lowest RMSD (Å, DeBartalo)
T1af7	69	25	100	2.07	2.5
T1b72	50	25	100	1.41	1.6
T1r69	61	25	100	2.11	2.4

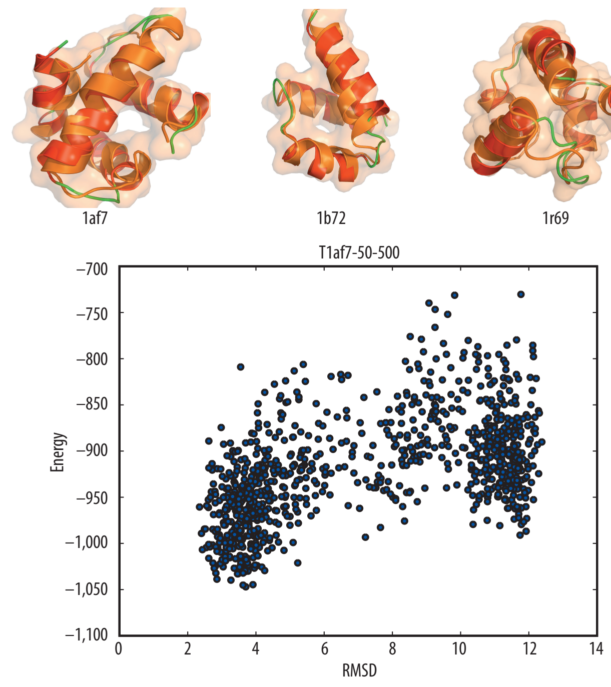

Figure 1. Results of script for predicting eight protein structures on 8,192 CPUs of the Intrepid BG/P, with details for three proteins and the Monte Carlo results for 1af7.

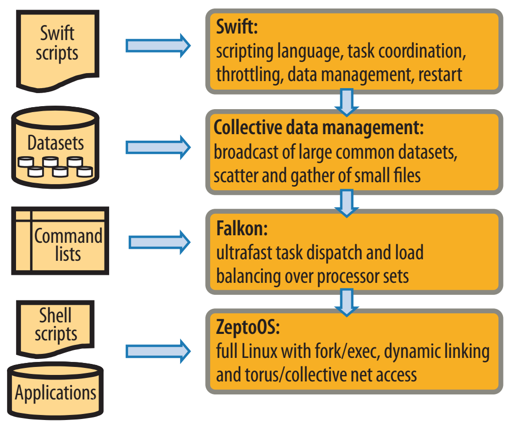
Swift scripts
Datasets
Command lists
Shell scripts
Applications
Swift:
scripting language, task coordination, throttling, data management, restart
Collective data management:
broadcast of large common datasets, scatter and gather of small files
Falkon:
ultrafast task dispatch and load balancing over processor sets
ZeptoOS:
full Linux with fork/exec, dynamic linking and torus/collective net access

Figure 2. Architecture for petascale scripting.

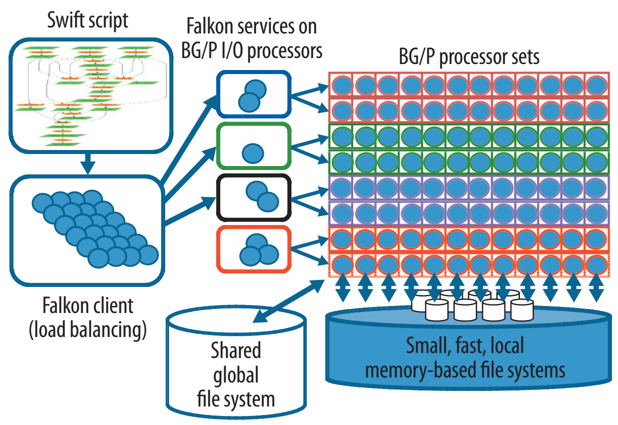
Swift script
Falkon services on BG/P I/O processors
BG/P processor sets
Falkon client (load balancing)
Shared global file system
Small, fast, local memory-based file systems

Figure 3. Swift scripts execute using the Falkon distributed resource manager on the BG/P architecture.

AN ARCHITECTURE FOR PETASCALE PARALLEL SCRIPTING

Petascale computing raises challenging problems for implementers of parallel scripting systems. Even a simple parallel script can define large numbers of concurrent tasks that may operate on even larger numbers of files. Task dispatch, data management and movement, mixed-mode parallelism, resource management, failure detection and recovery—these and other programming model functions can lead to difficulties when millions of tasks must execute efficiently and reliably on hundreds of thousands of cores.

Figure 2 shows the four-layer software architecture that we have developed in our investigations of parallel scripting systems. The four layers address, from the top down, the parallel scripting language and its engine and runtime system (Swift); a layer to support the data management demands that parallel scripting—and many-task computing in general—places on cluster file systems; runtime system support for high-performance resource provisioning and task dispatch (for example, the Falkon multilevel resource manager); and operating system support for basic script execution and access to high-performance communications networks (for example, the ZeptoOS Linux-based compute-node kernel). We have used these components to run parallel scripts on up to (so far) 160,000 cores.

Data management for petascale scripting

In a straightforward implementation of parallel scripting, large numbers of programs operate concurrently and independently on a shared parallel file system such as IBM's General Parallel File System (GPFS) on the BG/P. Such I/O patterns place a high burden on a persistent storage infrastructure and tend to be inefficient due to the consistency mechanisms enforced by traditional file system semantics. Our solution to this problem, which we refer to as *collective data management* (CDM),[6] is loosely inspired by collective parallel programming operations such as broadcast, gather, and two-phase I/O.

CDM, as currently conceived, comprises a set of communication strategies that leverage fast local file systems as a high-speed local file cache, use broadcast operations to handle distribution of common input data, employ efficient scatter/gather and caching techniques for input and output, and aggregate compute node storage into larger file systems that leverage a high-performance interconnect to deliver data to applications. In this way, CDM enables efficient and easy distribution of data files to and from computing nodes and can greatly reduce load on the underlying persistent storage system.

Our work to date with CDM has been performed largely on the BG/P, and leverages features such as the BG/P interconnect architecture with its separate collective network, ZeptoOS compute-node kernels with I/O forwarding, and GPFS with full multiprocessor data consistency guarantees. Most of these considerations apply to other deployed petascale systems, all of which run some form of parallel file system, such as GPFS, Lustre, or the Parallel Virtual File System (PVFS). Moreover, all have some form of high-performance, often hierarchical or heterogeneous, network interconnects—for example, a mix of torus, tree, or Clos networks.

We are currently experimenting with CDM concepts through the explicit insertion of CDM primitives and heuristics into applications. Our goal is that CDM operations will ultimately be invoked automatically and transparently by the Swift implementation, making them fully transparent to the programming model and user.

Falkon

To maximize the range of parallel scripts that we can run efficiently, we require rapid task dispatch and execution. For example, keeping 160,000 cores efficiently utilized running 60-second single-thread tasks requires that tasks be dispatched at more than 160,000/60 = 2,700

Table 1. Example parallel scripting applications.

Field	Description	Characteristics	Status
Astronomy	Creation of montages from many digital images	Many 1-core tasks, much communication, complex dependencies	Experimental
Astronomy	Stacking of cutouts from digital sky surveys	Many 1-core tasks, much communication	Experimental
Biochemistry*	Analysis of mass-spectrometer data for post-translational protein modifications	10,000-100 million jobs for proteomic searches using custom serial codes	In development
Biochemistry*	Protein structure prediction using iterative fixing algorithm; exploring other biomolecular interactions	Hundreds to thousands of 1- to 1,000-core simulations and data analysis	Operational
Biochemistry*	Identification of drug targets via computational docking/screening	Up to 1 million 1-core docking operations	Operational
Bioinformatics*	Metagenome modeling	Thousands of 1-core integer programming problems	In development
Business economics	Mining of large text corpora to study media bias	Analysis and comparison of over 70 million text files of news articles	In development
Climate science	Ensemble climate model runs and analysis of output data	Tens to hundreds of 100- to 1,000-core simulations	Experimental
Economics*	Generation of response surfaces for various economic models	1,000 to 1 million 1-core runs (10,000 typical), then data analysis	Operational
Neuroscience*	Analysis of functional MRI datasets	Comparison of images; connectivity analysis with structural equation modeling, 100,000+ tasks	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	Execution of MPI application in parallel	In development

Note: Asterisks indicate applications being run on Argonne National Laboratory's Blue Gene/P (Intrepid) and/or the TeraGrid Sun Constellation at the University of Texas at Austin (Ranger).

tasks per second. Given that the batch schedulers typically run on parallel computers can take 60 seconds to dispatch a single task, there is a clear need for alternative technologies.

In our work to date we have used the Falkon distributed resource manager[7] to address this need, as shown in Figure 3. Falkon uses a combination of multilevel scheduling and a hierarchical task dispatch architecture to enable rapid task dispatch. Its multilevel scheduling architecture—similar to that used in systems such as Condor and MyCluster—separates two activities that are normally combined on a supercomputer, namely allocating a node to a user and dispatching tasks to that node. In the first provisioning phase, Falkon requests nodes in large quantities, using a system's native batch scheduler, and starts a persistent task execution agent on each core capable of rapidly executing arbitrary and independent Posix processes. Once nodes are thus allocated, Falkon uses a hierarchical network of dispatchers to pass tasks to nodes that are, or soon will be, ready to execute them. These methods have allowed Falkon to dispatch more than 3,000 tasks per second on the BG/P and to run on up to 160,000 cores.[7]

ZeptoOS

The lowest layer in our parallel scripting architecture is a Posix-compliant operating system that provides system services such as the fork() and exec() used to launch a new application program and the I/O functions used to gain high-performance access to specialized communication networks. On the BG/P, we provide these features through the ZeptoOS Linux compute node kernel,[6] which implements Posix-compliant system services, full dynamic loading of executables, access to the BG/P collective ("tree") network through higher-level broadcast operations, IP connectivity over the torus network, and facilities to stripe the RAM-disk file systems of compute nodes and mount them as high-performance intermediate file systems. On other computers, such as the Constellation, we currently use the native compute node OS that provides a complete Posix interface, but we envision a role for ZeptoOS as a vehicle for kernel experimentation even on the Constellation and Cray XT5.

PARALLEL SCRIPTING APPLICATIONS

We have applied large-scale parallel scripting to numerous applications.[3-5,7-9] Each scripted application can consume a large fraction, or even all, of a petascale computer. All involve executing many tasks at once, often with substantial amounts of communication both within each task and among tasks. Table 1 lists some representative examples.

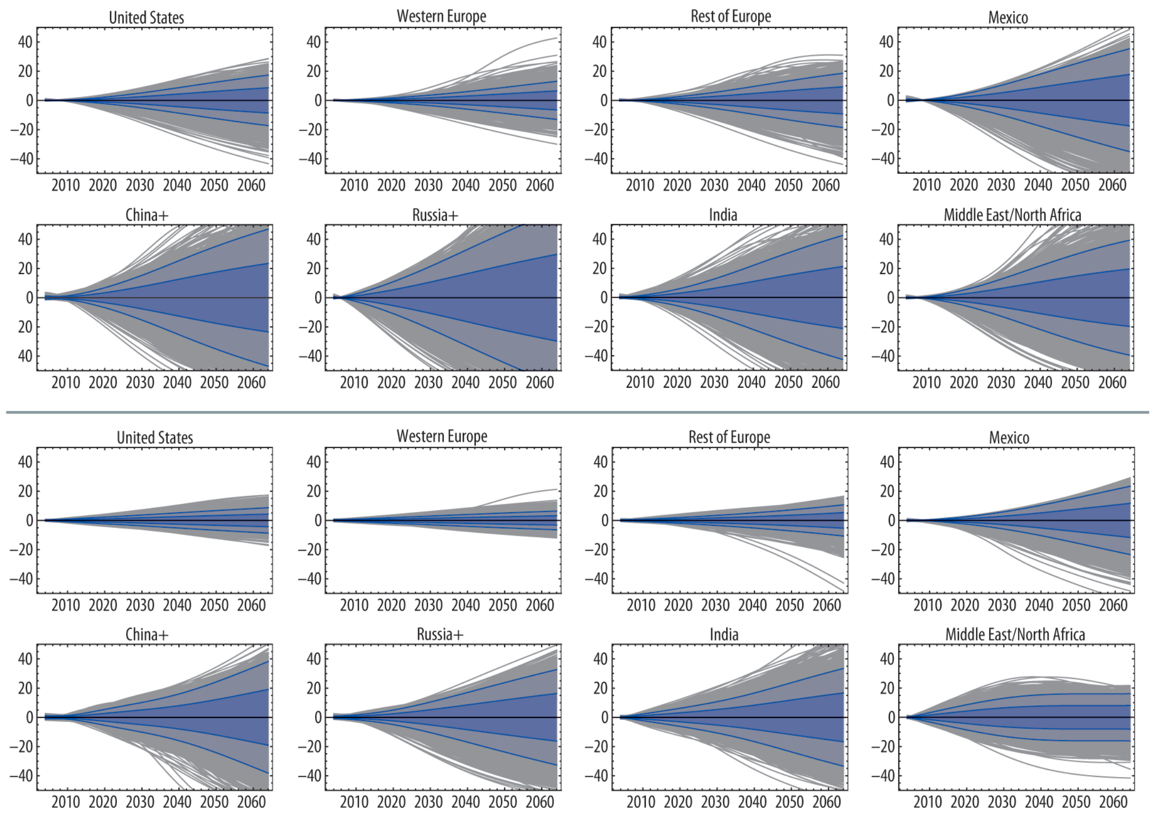

Figure 4. CIM-EARTH energy-economics parameter sweeps of 5,000 models exploring uncertainty in consumer (top) and industrial (bottom) electricity usage projections by region for the next five decades.

Molecular docking

The DOCK molecular dynamics application is run regularly on Intrepid to simulate the docking of small ligand molecules to large macromolecules (receptors). A compound that interacts strongly with a receptor associated with a disease may inhibit its function and thus prove useful in a beneficial drug.

This application is challenging because it involves 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 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 CIM-EARTH project for integrated social, economic, and environmental modeling (www.cim-earth.org) uses Swift on petascale systems to execute parameter sweeps of economic models that forecast energy use and other commodity demands to examine the effects of uncertainty. CIM-EARTH researchers use the parallel scripting paradigm to refine several models for exploring uncertainty through large-scale parallelism.

Figure 4 shows the results of a parallel script exploring the implications of uncertainty—in this case, parametric uncertainty in substitution elasticities. Researchers analyzed 5,000 samples from a perturbed input dataset in parallel on Ranger and other parallel systems of 5,000+ cores each. The model evaluates relative sensitivity to uncertainty (percent from the mean) for consumer and industrial demand for electricity in eight geographical regions. The dark-blue and light-blue envelopes are one and two standard deviations from the mean.

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[8] (SEM) and has implemented several parallel scripts for modeling functional MRI data within this framework. The Computational Neuroscience Applications Research Infrastructure[8] (CNARI, www.cnari.org) uses 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 (http://openmx.psyc.virginia.edu), a structural equation modeling package for R that can generate a single model object containing the matrices and algebraic information necessary to estimate the model's parameters. With the CNARI framework, neuroscientists run OpenMx from Swift scripts to conduct 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 PTMap application takes in raw data files from mass-spectrometry analysis of biological samples, along with the entire set of sequences of the organism's proteome, and searches them for statistically significant evidence of unidentified PTMs. The tool reads in a mass-spectrometry file—typically 200 megabytes of data in mzXML format—and protein sequences in FASTA format.

The analysis of a mass-spectrometry run for a single proteome 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, cows, mice, and humans.

PARALLEL SCRIPTING MODEL PERFORMANCE

Performance measurements indicate that on Intrepid, Falkon can execute more than 3,000 tasks per second, and launch, execute, and terminate 160,000 tasks on 160,000 cores in under one minute.[7]

Running DOCK under Falkon with a workload of 934,803 molecules (performing a DOCK execution for each one) on 116,000 CPU cores of the Intrepid BG/P took two hours,[7] as shown in Figure 5a, delivering 21.4 CPU-years. Per-task execution time varied considerably, from a minimum of 1 second to a maximum of 5,030 seconds, and a mean of 713$\pm$560 seconds. The two-hour run achieved a sustained utilization of 99.6 percent for the first 5,700 seconds and an overall utilization of 78 percent due to the workload tapering off at the end of the run. Despite the loosely coupled nature of this application, our results show that DOCK performs and scales well on a significant fraction of Intrepid, with 99.7 percent efficiency when compared to the same workload at 64,000 CPUs.

Figure 5b shows the progress and active processes of an SEM workflow with over 418,000 jobs, executing as a single Swift script invocation on Ranger to model neural pathway connectivity from experimental fMRI data.[8]

We performed preliminary measurements of the new PTMap application at modest scales, running the stage 1 processing of the *E. coli* K12 genome (4,127 sequences) on

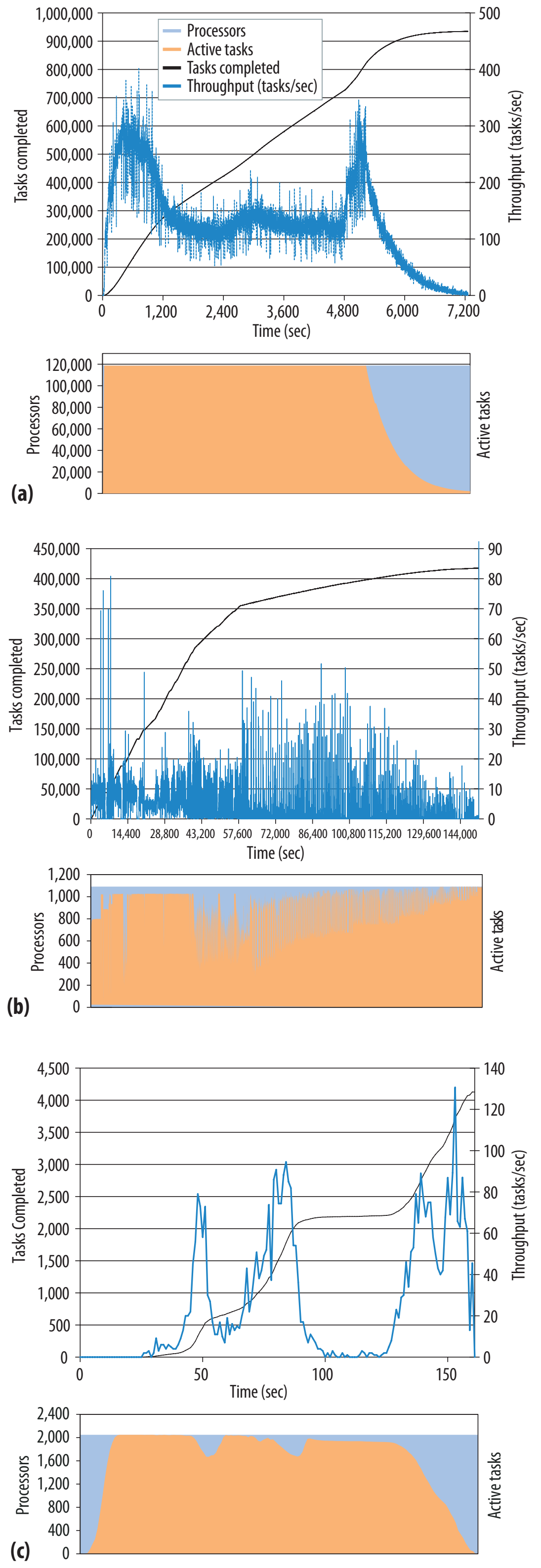

Figure 5. Performance of three parallel application scripts: (a) DOCK on BG/P—Falkon, 934,803 tasks, 2 hours; (b) SEM on Constellation—Swift, 418,000 tasks, 41 hours; (c) PTMap on BG/P—Swift, 4,127 tasks, 3 minutes.

2,048 Intrepid cores. Figure 5c summarizes this run. 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, achieving 80 percent utilization from a high-level Swift script.

We view these measurements—all on challenging short-task-length applications—as a promising milestone in meeting and in some cases exceeding the performance needed for petascale scripting and beyond.

Tools that make it easy to couple existing programs and apply programs to different data—in other words, scripting tools—align well with how people approach problem solving.

RELATED WORK

MapReduce,[10] Sphere,[11] and Dryad[12] implement library-based approaches to parallel processing of large datasets. For example, in the MapReduce paradigm, data is distributed over many nodes. SPMD applications can then call both local functions that execute on local data and reduction operations to combine distributed data. This model can require both substantial rewriting of programs and reorganization of data. In contrast, Swift programs require no modifications to application programs. Instead, Swift allows the programmer to focus on composing those programs into larger applications. We view the ability to leverage the vast value embedded in modern sequential and parallel application codes as an important property of parallel scripting. Swift's `foreach` construct performs a simple map operation, and the act of passing a multimember dataset to a procedure provides a simple and natural way to implement reduction operations.

The Nimrod system[13] is an example of a more specialized form of parallel programming system. Nimrod supports parallel computations involving many invocations of an external executable, driven by a high-level specification of a parameter study or, in more recent versions, a numerical optimization strategy.

SPMD message-passing systems such as MPI can be used to express some task-parallel computations. However, MPI is less well suited for the dynamic environments and applications at which Swift excels. In addition, any SPMD programming model, including MPI, faces issues of reliability when scaling to millions of processors and beyond, because of shorter mean time to failure as machines grow in size. The parallel scripting model is more flexible in this regard because failures are typically localized within a compute node task that can be re-executed and thus need not cause an entire application to fail. We view Swift and MPI as complementary in that Swift can be used to coordinate the execution of MPI applications.

Numerous dynamic load balancing libraries have been implemented over the years, varying in details but not general approach. Condor's manager-worker library is one example. Another, implemented within the MPI paradigm, is the Asynchronous Dynamic Load Balancing library.[14] 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.

The design of Falkon was inspired by the Condor Glide-in facility,[15] which established the utility of multilevel scheduling. Falkon is based on similar principles but implements a simpler facility that contains only the essential semantics needed for first-in, first-out task scheduling and thereby delivers orders of magnitude better scalability and throughput on petascale systems.[7]

High-performance languages for tightly coupled programming, such as Chapel,[16] also offer features similar to those found in Swift. 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. 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` 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, but offers the programmer fewer ways to circumvent the typing model and lacks Chapel's semantics for type inference.

Ousterhout's observation concerning the power of scripting reflects a profound truth about programming. As in other fields of human endeavor, complex artifacts are often created by coupling existing components. Thus, tools that make it easy to couple existing programs and apply programs to different data—in other words, scripting tools—align well with how people approach problem solving.

Historically, people used scripting to prototype programs on workstations, but for more serious programming tasks, such as for parallel computers, they used different methods and tools. It is time to reconsider that position on parallel computers, just as people are doing in other environments. Not only can a scripting approach facilitate the rapid construction of large computations via the composition of existing components, but a scripting language's composition operators often reveal opportunities for parallel execution. Swift shows how a language that supports simple dataflow concepts and file system mapping constructs can allow for the concise specification of highly parallel computations within a scripting framework.

There might be some skepticism about whether scripting methods can be implemented efficiently on large-scale parallel computers, given the need to schedule, dispatch, and manage many tasks on many processors, all the while supporting large numbers of fine-grained I/O operations within both shared and local file system namespaces. Yet our experience shows that these issues need not stand in the way of performance. Data dependency and task management activities can be scaled relatively easily with the use of hierarchical scheduling methods. File system operations can also be scaled, within the constraints that single-assignment semantics place on how parallel scripts access the file system: A file may have many readers, but only one writer. The resulting computations may sometimes stress a parallel computer's communication network, but they usually perform sufficiently well to accomplish a vast array of important scientific tasks with unprecedented speed.

We continue to explore new applications that benefit from parallel scripting and to extend the power and performance of the Swift scripting system. Based on what we have learned to date, we believe that parallel scripting has proven its value on petascale systems and will play an indispensable role in the exascale programming tool chest. C

Acknowledgments

We thank many people for collaborating with us: Sarah Kenny, Steve Small, Michael Andric, and the OpenMx project team on SEM and neuroscience applications; Glen Hocky, Joe DeBartolo, Karl Freed, and Tobin Sosnick on the Open Protein Simulator; Yingming Zhao and Yue Chen on PTMap; Andrew Binkowski and Mike Kubal on molecular docking; Joshua Elliott, Meredith Franklin, and Todd Munson on CIM-EARTH; Andriy Fedorov on image processing for surgical planning; Yong Zhao for early work on Swift; and Justin Wozniak and Gail Pieper for comments on this article. This research is supported in part by NSF grants OCI-721939 and OCI-0944332, NIH grants DC08638 and DA024304, the U.S. Department of Energy under contract DE-AC02-06CH11357, NASA Ames Research Center GSRP grant NNA06CB89H, and the University of Chicago/Argonne National Laboratory Computation Institute. Computing resources were provided by the Argonne Leadership Computing Facility, TeraGrid, the Open Science Grid, and Amazon Web Services.

References

1. J. Ousterhout, "Scripting: Higher-Level Programming for the 21st Century," *Computer*, Mar. 1998, pp. 23-30.
2. Y. Zhao et al., "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," *Proc. 2007 IEEE Congress on Services*, IEEE Press, 2007, pp. 199-206.
3. 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.
4. G. Hocky et al., *Toward Petascale ab initio Protein Folding through Parallel Scripting*, tech. report ANL/MCS-P1645-0609, Argonne National Laboratory, 2009.
5. J. DeBartolo et al., "Mimicking the Folding Pathway to Improve Homology-Free Protein Structure Prediction," *Proc. National Academy of Sciences*, 10 Mar. 2009, pp. 3734-3739.
6. Z. Zhang et al., "Design and Evaluation of a Collective I/O Model for Loosely-Coupled Petascale Programming," *Proc. 2008 IEEE Workshop Many-Task Computing on Grids and Supercomputers (*MTAGS 08), IEEE Press, 2008, pp. 1-10.
7. 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.
8. S. Kenny et al., "Parallel Workflows for Data-Driven Structural Equation Modeling in Functional Neuroimaging," *Frontiers in Neuroinformatics*, Nov. 2009.
9. A. Fedorov et al., *Non-Rigid Registration for Image-Guided Neurosurgery on the TeraGrid: A Case Study*, tech. report WM-CS-2009-05, Dept. of Computer Science, College of William and Mary, 2009.
10. J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," *Comm. ACM*, Jan. 2008, pp. 107-113.
11. Y. Gu and R.L. Grossman, "Sector and Sphere: The Design and Implementation of a High Performance Data Cloud," *Philosophical Trans. Royal Society A*, 28 June 2009, pp. 2429-2445.
12. M. Isard et al., "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," *ACM SIGOPS Operating System Rev.*, June 2007, pp. 59-72.
13. D. Abramson et al., "Parameter Space Exploration Using Scientific Workflows," *Computational Science—ICCS 2009*, LNCS 5544, Springer, 2009, pp. 104-113.
14. P. Balaji et al., "MPI on a Million Processors," *Proc. 2009 European PVM/MPI Users' Group Conf.* (EuroPVM/MPI 09), CSC-IT Center for Science, 2009.
15. D. Thain and M. Livny, "Building Reliable Clients and Services," *The Grid: Blueprint for a New Computing Infrastructure*, I. Foster and C. Kesselman, eds., Morgan Kaufmann, 2005, pp. 285-318.
16. B. Chamberlain, D. Callahan, and H. Zima, "Parallel Programmability and the Chapel Language," *Int'l J. High-Performance Computing Applications*, Aug. 2007, pp. 291-312.

***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.*

Ian Foster *is a Distinguished Fellow at Argonne National Laboratory, director of the University of Chicago/Argonne National Laboratory Computation Institute, Chan Soon-Shiong Scholar, and Arthur Holly Compton Distinguished Service Professor of Computer Science at the University of Chicago. Contact him at foster@anl.gov.*

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

Zhao Zhang *is a PhD student in the Department of Computer Science at the University of Chicago. Contact him at zhaozhang@wuchicago.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.*

Mihael Hategan *is a systems software developer at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at hategan@mcs.anl.gov.*

Ben Clifford *was formerly a systems software developer at the University of Chicago/Argonne National Laboratory Computation Institute. Contact him at benc@hawaga.org.uk.*

Ioan Raicu *is an NSF/CRA Computing Innovation Fellow at the Center for Ultra-scale Computing and Information Security, Department of Electrical Engineering and Computer Science, Northwestern University. Contact him at iraicu@eecs.northwestern.edu.*