Scalable Parallel Scripting for Scientific Computing

Researchers at the University of Chicago and Argonne National Laboratory have been extending the time-tested programming technique of scripting into new realms of performance. Through the Swift parallel scripting language, they aim to enhance scientific productivity by enabling scripts to execute many copies of ordinary application programs at very high degrees of parallelism. Parallel Swift scripts can run with little or no change across a range of platforms from multicore desktops to the largest petascale systems available. Scientists are using Swift in a broad range of disciplines to productively leverage highly-parallel resources such as the Blue Gene/P in the Argonne Leadership Computing Facility.

Scientists are using Swift on a range of projects including protein folding, molecular dynamics, climate, energy, economics, statistics, neurobiology, and others.

Many scientists, particularly those who use rather than write computational codes, find that the challenges involved in executing large-scale computing tasks consume tremendous amounts of time and intellectual focus – precious commodities that they would prefer to apply to their core science rather than to the mechanics of computing.

Some of these programs – certainly applications developed through SciDAC – are large parallel codes that can take up a whole machine for a single run. Others are serial codes, or modestly parallel codes that can efficiently utilize a few thousand CPU cores. But regardless of their scale, these codes often need to be run repeatedly to pursue a given inquiry – on diverse datasets, in various parameter sweeps, to process large datasets, to explore multiple hypotheses or parameter spaces, or to evaluate new codes and algorithms. To automate such repetitive execution, scientists typically turn to the technique of *scripting*.

John Ousterhout, the computer scientist who developed the Tcl scripting language, in an article in *IEEE Computer* in 1998 described scripting as "higher-level programming for the 21st century." But since that time, little work has been done on the *scaling* of scripting languages to utilize the increasingly parallel nature of available computing resources. Is it possible to take that next step – to develop simple and automatic parallel scripting methods so that more applications can be run efficiently, even on petascale computers?

Researchers at Argonne National Laboratory (ANL) and the University of Chicago – along with a growing user community – believe so, and to this end they have been exploring a dataflow-driven parallel programming model that treats applications as functions and datasets as structured objects. The new model has been implemented in a parallel scripting language called Swift, which has been run successfully on advanced computers, including the 160,000-core Blue Gene/P in the Argonne Leadership Computing Facility (ALCF), NSF supercomputers on TeraGrid, the Linux clusters of the Open Science Grid, and Amazon EC2 systems (figure 1).

Scientists are using Swift on a range of projects that include: protein folding; molecular dynamics; modeling the interactions of climate, energy, and economics; exploring the language functions of the human brain; predicting the structure of proteins and hunting for the posttranslational modifications that explain their behaviors; creating general statistical frameworks for powerful techniques such as structural equation modeling; and performing image processing for applications ranging from probing the neurobiology of

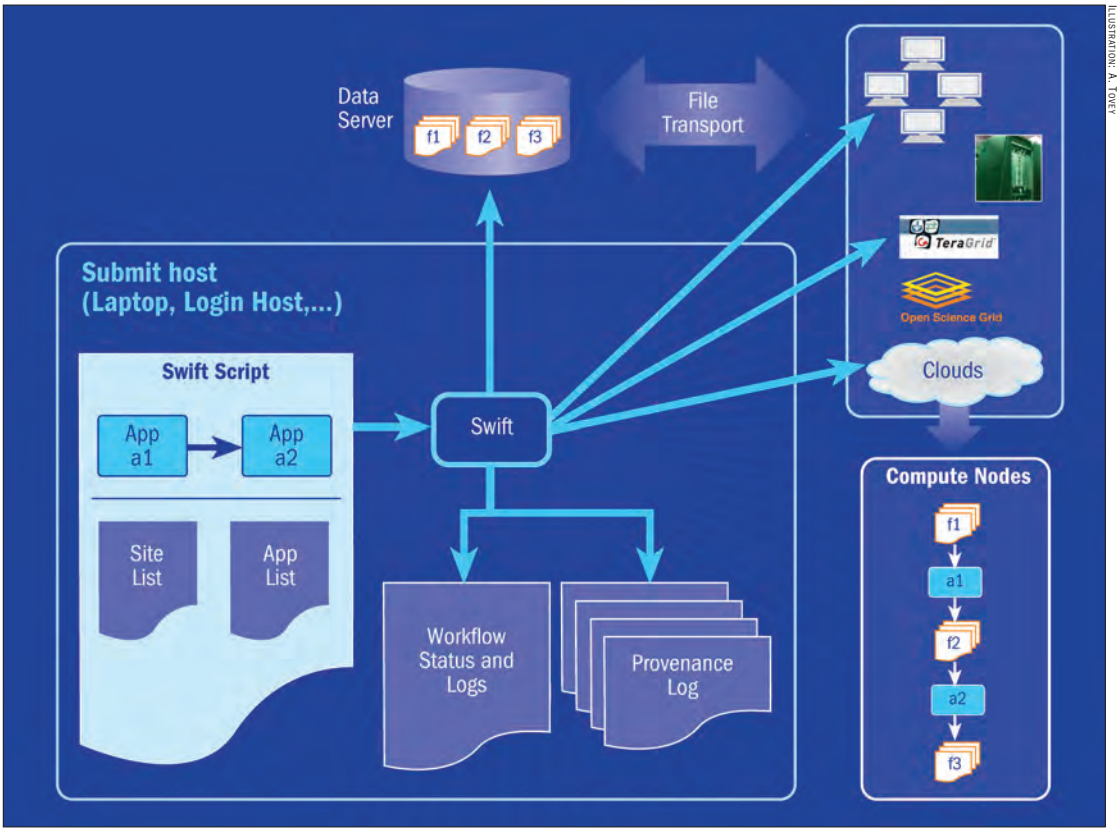

ILLUSTRATION: A. TOVEY

Figure 1. *Using Swift. The Swift command, a Java application, runs on any user-accessible computer, such as a workstation or login server. It compiles and executes Swift scripts, coordinates remote data transfers, and executes applications on local and distributed parallel resources.*

tiny model-organism worms to planning for human neurosurgery.

Challenges and Requirements for Parallel Scripting

With traditional serial scripting languages, writing a small script that performs the nested loops of a parameter sweep can be fairly simple. A more difficult challenge is mastering the complex scripts needed to distribute tasks and data to multiple remote systems, gather results, and record the provenance of these derived results. And the mechanisms to keep a petascale machine efficiently utilized using such scripts have not previously existed. Consider the following problem scenarios that illustrate the needs for parallel scripting, expressed in English-like pseudocode.

Many genomics communities need to perform "all-by-all" runs of BLAST, in which every sequence in a database is compared against every other sequence. These runs have the following simple form.

For each batch of sequences in the genome database
  results = blast(genome-database, query, parameters)

This simple workload can demand vast amounts of computing and can generate a huge demand for Grid, cloud, and petascale resources. Another scientist, examining drug targets, may need to do the following.

For each target protein in a set of 20 targets
  For each candidate drug compound in a set of 500,000 candidates
    (score, energy) = ***dock****(target, drug, docking parameters)*

Here "dock" is an application that simulates the interaction (docking) of small molecular compounds (ligands) with large target biomolecules, such as proteins. Once the initial results are available, the top-scoring 1% of the candidates for each target must be screened in a similar manner but with a more computationally demanding simulation involving higher-fidelity molecular dynamics.

And a climate scientist may want to execute a parameter sweep such as the following, for each of thousands of perturbations of the input parameters, to model the effects of uncertainty on predictions of emissions and energy demand.

For each of 15 geographic regions
  For each of 20 commodities
    For each of 4 energy sources
      Initial conditions = ***perturb****(input parameters)*
      Energy demand = ***model****(initial conditions)*

K. Freed, T. Sosnick, J. DeBartolo (U. Chicago), and G. Hocky (Columbia)

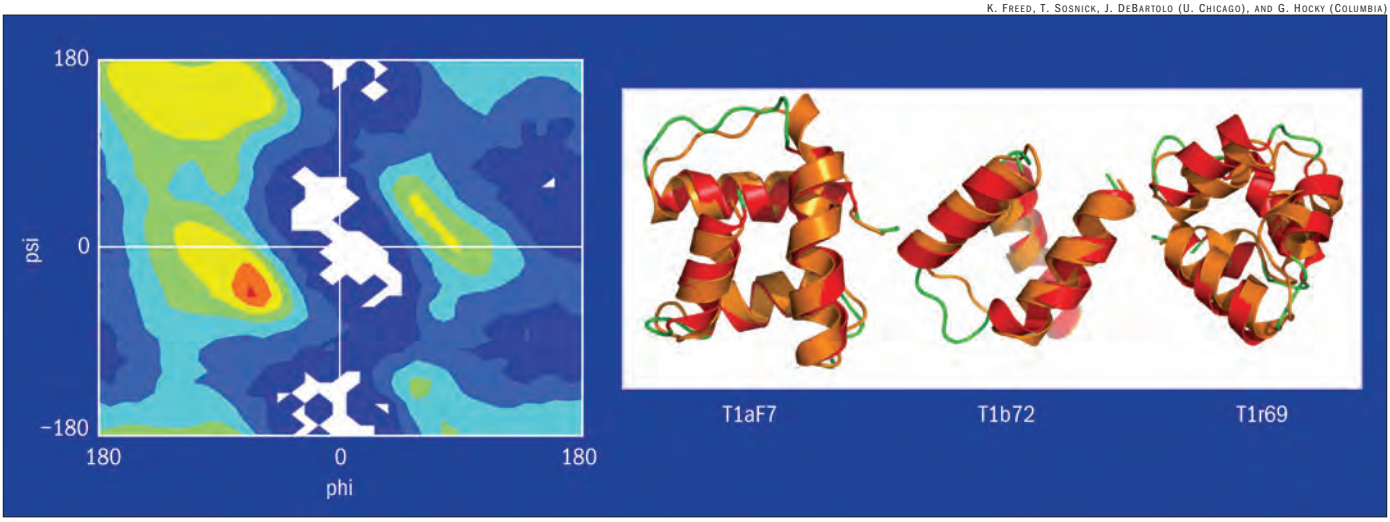

Figure 2. *Structure prediction with the Open Protein Simulator. Left, the "Ramachandra map" used by the ItFix iterative fixing algorithm, which helps guide the "moves" of a simulation. Such maps give phi–psi angle distributions of solved PDB structures for all three-residue trimers. They were used to predict the structure of proteins 1AF7, 1B72, and 1R69, shown with the prediction in red and green compared to the experimentally determined structure in orange.*

To perform a complete structure prediction for a single protein sequence, hundreds to thousands of copies of the predict program are run in each of multiple parallel rounds.

Each of these example scripts specifies a large number of (sequential or parallel) simulation or data analysis application program invocations that are independent of each other and can be run in parallel. Those program invocations can run for significant amounts of time – typically from minutes to hours – before they need to pass data from one program to another. Thus, these scripts all exhibit considerable opportunities for coarse-grained parallel execution. This simplicity has led to such applications to be termed "pleasingly," or, more often, "embarrassingly" parallel.

But while these problems have clear and explicit parallelism, writing scripts in a traditional scripting language to explicitly perform all the aspects of parallel scheduling, data passing, file management, error recovery, and logging is anything but simple. Scripting such coordination has proven to be a labor-intensive, error-prone, and time-consuming endeavor for many scientists. For others, the complexity of this endeavor has been a barrier to even trying parallel execution of their computing tasks. And when the execution pattern of a script involves multiple applications and many stages of execution, as is frequently the case, managing the parallelism and moving data efficiently from one program to the next are no longer a trivial problem.

Parallel Scripting with Swift

To solve problems like the ones just cited, the Argonne–University of Chicago research team has developed Swift, a new scripting language that enables ordinary application programs that read and write files to be executed, at a high degree of parallelism. Swift can execute application programs many thousands at a time, on a wide range of computing platforms, and can chain different programs together in the myriad patterns needed to perform the work of scientific computing.

A Taste of Parallel Scripting

The following example illustrates the power of Swift. Drs. Karl Freed and Tobin Sosnick at the University of Chicago use "Open Protein Simulator" software to perform protein structure prediction simulations (figure 2). The structure predictor's main application program, "predict," simulates the folding of a protein's sequence of amino acid molecules into its unique three-dimensional shape that determines much of the protein's functional behavior. The predict program uses simulated annealing to explore three-dimensional movements of the protein's amino acids, repeatedly changing angles within the protein's internal bonds in search of shapes having the minimal-energy configurations that protein molecules seek in nature.

In order to perform a complete structure prediction for a single protein sequence, hundreds to thousands of copies of the predict program are run in each of multiple parallel rounds. Each run of the predict program computes an independent, randomly seeded prediction. When a round of predictions is completed, an analysis program determines the best prediction and uses that configuration of protein shapes (the secondary structure of coils, strands, and helices) as the starting configuration of another round. This process of repeated parallel rounds, called iterative fixing, can be expressed clearly and concisely in Swift, which automates the parallel execution and the management of files within the algorithm, thus allowing the process to be conducted on a wide range of computing systems and at very high degrees of parallelism.

Swift Execution Semantics

Task synchronization in Swift (that is, determining when each given task can execute) is based on data availability. For example, in a Swift procedure, all the statements in each block of the procedure are conceptually started in parallel. The order in which the statements execute is determined by any data dependencies between the statements. For example, if two statements depend only on the variable *a*, then both statements are executed in parallel as soon as another statement sets the value of *a*. In addition, the swift `foreach` statement executes all instances of the statements in the body of the `foreach` in parallel. These two constructs provide the inherent parallelism of the Swift language.

Because every "atomic" Swift data element (that is, each simple scalar variable, array element, and structure field) has these same "write once, read when set" synchronization semantics, all computations in the execution of a Swift script are implicitly pipelined. For example, if one `foreach` loop is computing a set of values and storing the results in an output array, another `foreach` that reads the elements of the array will progress in parallel, with each member of the dependent `foreach` loop running as soon as the corresponding `foreach` loop has set the dependent array element. This approach enables Swift programs to execute with a much higher degree of overall parallelism than would be possible with a less fine-grained model of concurrency.

At the heart of the scripts, a round of parallel structure prediction simulation can be expressed in Swift as follows.

```
(ProtSim psim[]) doRound
 (Protein p, int n, PSimConfig cf)
{
   foreach sim in [1:n] {
      psim[sim] = predict(p, cf );
   }
}
```

This function invokes `n` copies of `predict`, each taking the same protein and configuration file as inputs, and each computing with a different random seed. The results are returned in an array, `psim`.

With this core parallel function, the iterative fixing algorithm can be expressed compactly in a short Swift script as a higher-level function, for example, `ItFix()`. Then, researchers can use `ItFix` as a computational laboratory to explore protein structures in a variety of applications. Thus, users with little programming experience can easily perform large-scale parallel simulation experiments with a simple high-level script. For example, they can explore the effects of varying two parameters of the simulation, on multiple proteins, by using Swift scripting code like the following.

```
foreach protein, i in proteins {
   foreach s in seeds {
      foreach c in coeff {
         results[i] =
         ItFix(protein, nSim, nRounds, s, c);
      }
   }
}
```

In this example, for each protein, initial seed, and temperature coefficient in the arrays `proteins`, `seeds`, and `coeff`, the Swift program runs an entire simulation.

Swift is a data flow language, meaning that programs are scheduled for execution when the data that they consume become available (often when produced by a prior program in a processing chain). Its functional programming model (sidebar "Swift Execution Semantics") encapsulates implementation details within application programs and enables users to create libraries that provide a functional abstraction of the program tools of an application domain, such as `predict` and `ItFix`.

Swift includes a powerful data model that permits individual files and directories to be mapped into Swift language variables. Thus, users can describe and access nested on-disk directories as simple structures and arrays within Swift programs, specifying, for example, that a particular program should be invoked on every file in a directory. Swift then handles the details of packaging the specified files for dispatch to a remote execution site.

Swift automatically performs the following actions for the user, moving the problem of orchestrating parallel program execution to a higher level:

- Calls mapping scripts (either built-in or provided by the user) to determine the names of input files and to assign names to output files
- Determines what resources to run on and how many jobs to run in parallel at once on each site (through configurable "throttle" settings)
- Moves input data from persistent locations on shared file systems to temporary (and often faster, local) working directories
- Moves result data back to the output files designated by the user's mapping request

Swift includes a powerful data model that permits individual files and directories to be mapped into Swift language variables.

- Generates a log of activities in the run, including details of what applications were called with what arguments

- Determines when subsequent tasks that depend on the output of a prior task can execute

- Automatically retries failing jobs a specified number of times, and can even replicate tasks to multiple sites to increase their chance of completing successfully, sooner

- Remembers the state of a script's execution, so that the script can be resumed from the point of failure if the Swift command terminates or fails for any reason

The Swift runtime system uses a two-level scheduling method to manage the scheduling of tasks for execution and the dispatch of executable tasks to parallel computers. First, task executors are deployed onto nodes; then tasks are streamed to those executors. To handle diverse authentication and job submission systems, Swift can use Globus or other distributed computing systems to access remote sites.

Roots in Scripting Languages

Scripting is not new. As far back as the dawn of commercial mainframes, scripts in the form of operating system command languages were used to automate repetitive tasks and group together sequences of program executions. The UNIX shell and its use of text files as a universal interchange format improved on prior command scripting approaches by making it easier to pipe, or redirect, data from one application to another and to store collections of data in hierarchical file systems.

Shell scripts were created in response to the needs of computer users to avoid the tedious re-entering of repeated sequences of commands. The design of many scripting languages soon followed, led by languages such as Perl, which facilitates manipulation of text files, and then tcl, Python, Ruby, Visual Basic, Java scripting classes, and many more.

Scripting languages have greatly simplified programming: by focusing on the composition of existing programs, they enable users to assemble sophisticated application logic quickly. Application developers today utilize scripting on their workstations and servers to explore parameters and test hypotheses, before running simulations on larger computer platforms. Researchers use scripting to assemble more powerful applications from existing sequential or parallel programs.

Based on this lineage, Swift provides a unique blend of prior capabilities that is kept deliberately simple and minimalist, while providing new capabilities not found in any other scripting language: (1) treating file data and in-memory data in a uniform way, with a data typing and mapping model; (2) associating interface definitions with scripts and their internal functions, based on the data model – a degree of modularity not present in other scripting languages; (3) supporting location-independent execution across diverse runtime environments (workstation, cluster, Grid, cloud, and petascale) through a generalized driver interface for execution and data management functions; and (4) enabling integrated, transparent provenance tracking. The first two features above – data typing and functional interfaces based on these data types – are the mechanisms that make the latter two features possible.

Swift provides a unique blend of prior capabilities that is kept deliberately simple and minimalist, while providing new capabilities not found in any other scripting language.

The nature of Swift's encapsulation of application programs, as well as the manner in which it imparts a location-independent data model and interface definition to Swift functions, is illustrated in figure 3.

Using Swift with Other Scripting Languages

Swift is, by intention, a minimal scripting language. The definition of a new programming language is a risky business, raising issues of community adoption. We define Swift as a language rather than as a library because its model of parallelism and of data abstraction can be expressed far more cleanly within a special-purpose language. Its functional programming data flow model, its model of datatypes and mapping, and its inherent parallelism yield a nice, compact expression of the backbone of scientific computations.

Other scripting languages integrate easily with Swift scripts. Since scripts in any language are executed by the operating system as if they were ordinary programs, such scripts are simply described as "app()" functions and called from Swift. In fact, a library of useful Swift utility functions is evolving to perform tasks such as string manipulation and format conversions, so that the Swift language can have the necessary expressive power while remaining small, compact, and concise.

Users are also using Swift to run complete applications in the form of Python, Octave, and compiled MATLAB® codes. As an example of such language interoperability, Swift is frequently used with the popular R data analysis framework. For example, Swift+R are used to achieve distributed parallel execution on Grid and petascale systems for the OpenMx R package for structural equation modeling (SEM) developed by a team led by Dr. Steven Boker (University of Virginia) and Dr. Michael Neale (Virginia Commonwealth University). R users use Swift to perform large OpenMx SEM runs in parallel to analyze neuroscience data.

ILLUSTRATION: A. TOVEY

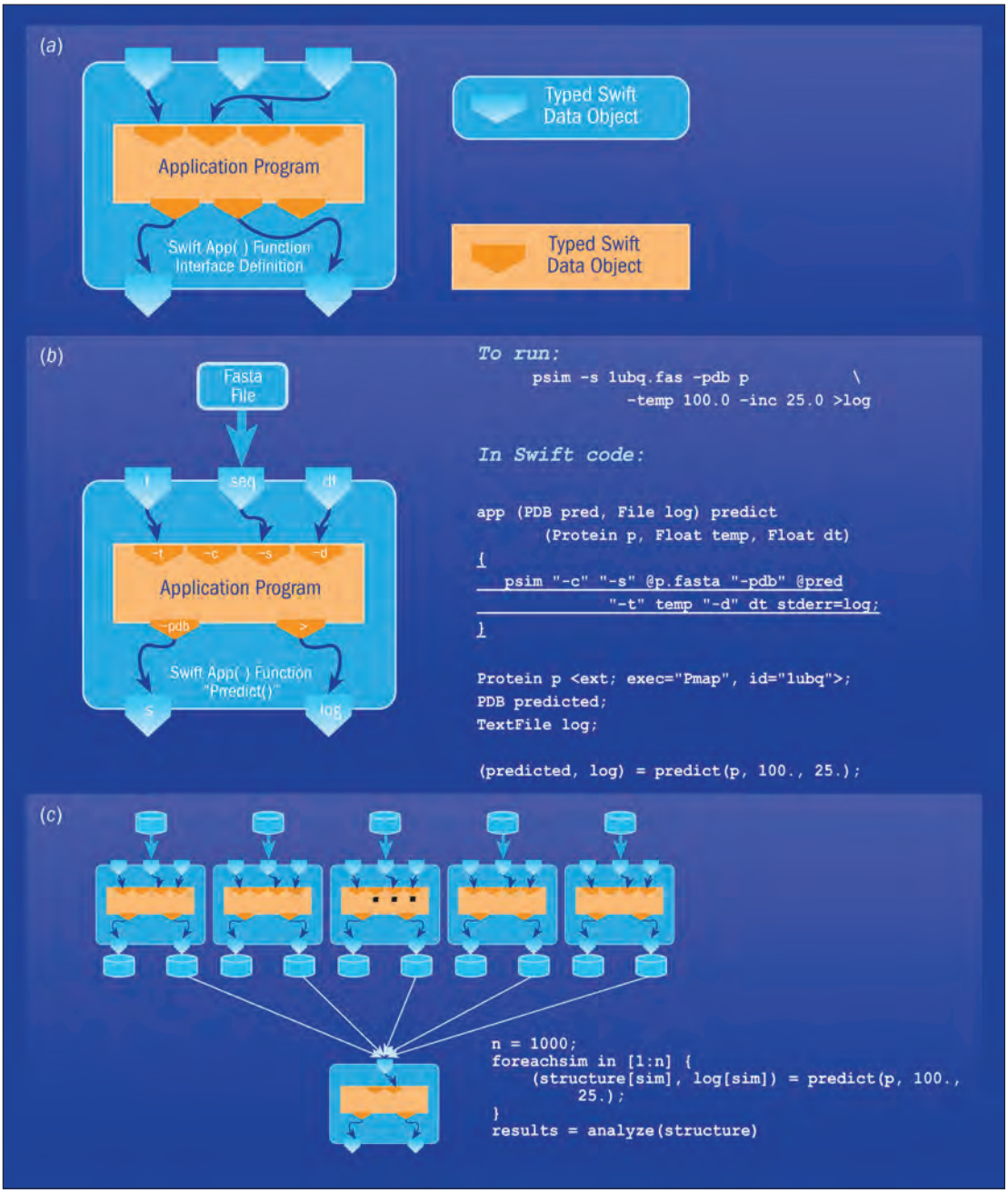

Figure 3. *(a) Encapsulation of application programs as Swift functions. Swift "app( )" functions provide a uniform interface for any application program, describing its inputs and outputs. This enables Swift to execute applications remotely and in parallel. Each argument is given a well-defined data type, which can be simple or structured sets of files and simple values. (b) An example of defining a program "psim" as a Swift "app" function. Now "predict()" can be run in parallel on distributed computing clusters. The script writer does not worry about data transport or scheduling. This reduces mental effort, and lets the scientist focus on the parameters and data objects to be passed to predict, and on the returned results. (c) The Swift "foreach()" statement is used to specify large-scale parallel execution of applications or of other Swift procedures.*

Over the past several years, the Swift research group has been successful in making Swift run efficiently on emerging petascale computing systems, validating the belief that scripting at this level is feasible, useful, and in fact necessary.

Extending the Parallel Scripting Model to Petascale Systems

Over the past several years, the Swift research group has been successful in making Swift run efficiently on emerging petascale computing systems such as the IBM Blue Gene/P "Intrepid" at ALCF and the Sun Constellation "Ranger" at the University of Texas–Austin, validating the belief that scripting at this level is feasible, useful, and in fact necessary. The motivation to elevate scripting to the level of petascale computing is both to solve new types of problems on leadership-class machines and to provide a solution for scientists who need to run their already highly-parallel applications in a hybrid manner, by creating scripts to perform many parallel runs of a large-scale parallel application.

ILLUSTRATION: A. TOVEY

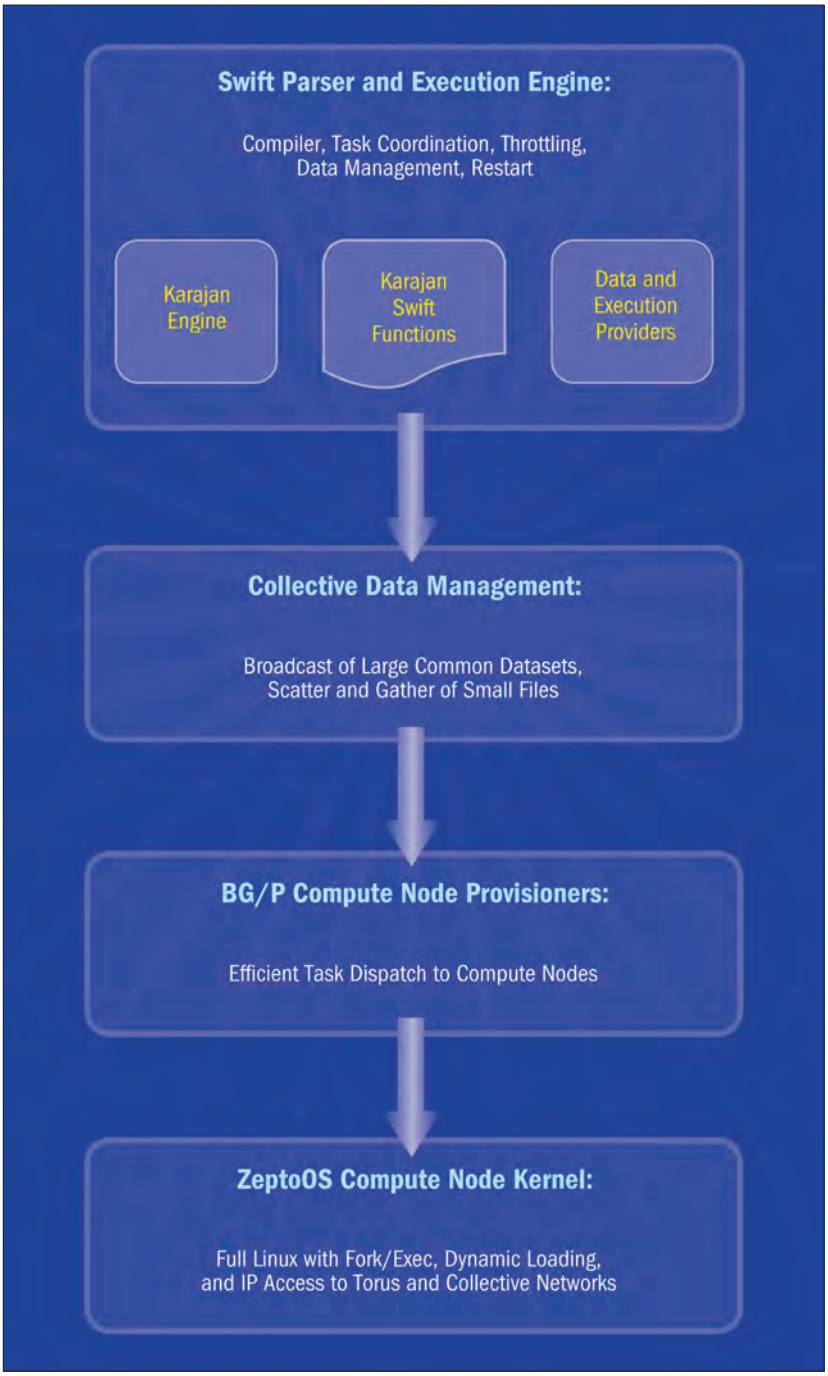

Figure 4. *Architectural layers for petascale scripting on the Blue Gene/P.*

To extend the parallel scripting architecture to petascale systems, the Argonne and Chicago researchers had to overcome three challenges: provisioning compute nodes and efficiently dispatching task executions to them, reducing the overhead of linking parallel computations by file exchange, and providing a compute node operating system that efficiently supports these two activities (figure 4).

Provisioning

When running parallel jobs on today's high-performance computers, the same application program code is typically executed on all compute nodes allocated to a job. But parallel scripting requires that many parallel instances of independent and possibly different application programs be executed on any compute node.

Thus, to run efficiently on the Blue Gene/P, the Argonne and University of Chicago researchers had to devise a new scheduling approach, which they dubbed Falkon, the Fast and Lightweight Task Execution Framework. Falkon uses multilevel scheduling to separate resource provisioning from the dispatching of user tasks to those resources (figure 5). Its streamlined task dispatcher can achieve order-of-magnitude higher task dispatch rates than can conventional schedulers. On the Blue Gene/P, Falkon has achieved rates of over 1,000 tasks per second.

Provisioning resources for parallel computing is the process of allocating compute nodes that will be used for the duration of a script's execution. As is typical for scripting languages like Perl and Python, each invocation of the Swift command runs one script. Each script can involve the execution of many – in some cases hundreds of thousands of – program invocations. On a typical cluster, each job is submitted to the cluster scheduler's batch queue. Running a single program through the batch queue has high overhead, in terms of both processing overhead and time spent waiting in the queue alongside other jobs that are typically waiting for many processors and a much longer allocation of processor time.

The Falkon provisioner requests resources in quantities of time and number of processors similar to those of typical parallel jobs: thousands of CPUs for many hours, rather than single CPUs for a few minutes. Once resources are allocated, Falkon launches a task execution agent on each compute node. This agent remains active for the duration of the resource allocation, and can rapidly and efficiently dispatch a program to run on each core as soon as that core is free. Thus, a script execution manager like Swift can efficiently utilize cores even for extremely short tasks. Provisioning is often called "multilevel scheduling" because the provisioner acts as a fast, simple, and low-latency scheduler running under the main job scheduler of the cluster.

The Falkon provisioner has been used in diverse environments, including local clusters, multisite Grids (examples include Open Science Grid and TeraGrid), specialized parallel clusters (such as the SiCortex 5832), and large supercomputers (like the Blue Gene/P, Ranger). Tests by the designer of Falkon, Dr. Ioan Raicu (then a University of Chicago graduate student, now a researcher at Northwestern University) have verified that Falkon can execute tasks efficiently at the full scale of the ALCF Blue Gene/P – 160,000 processor cores. The

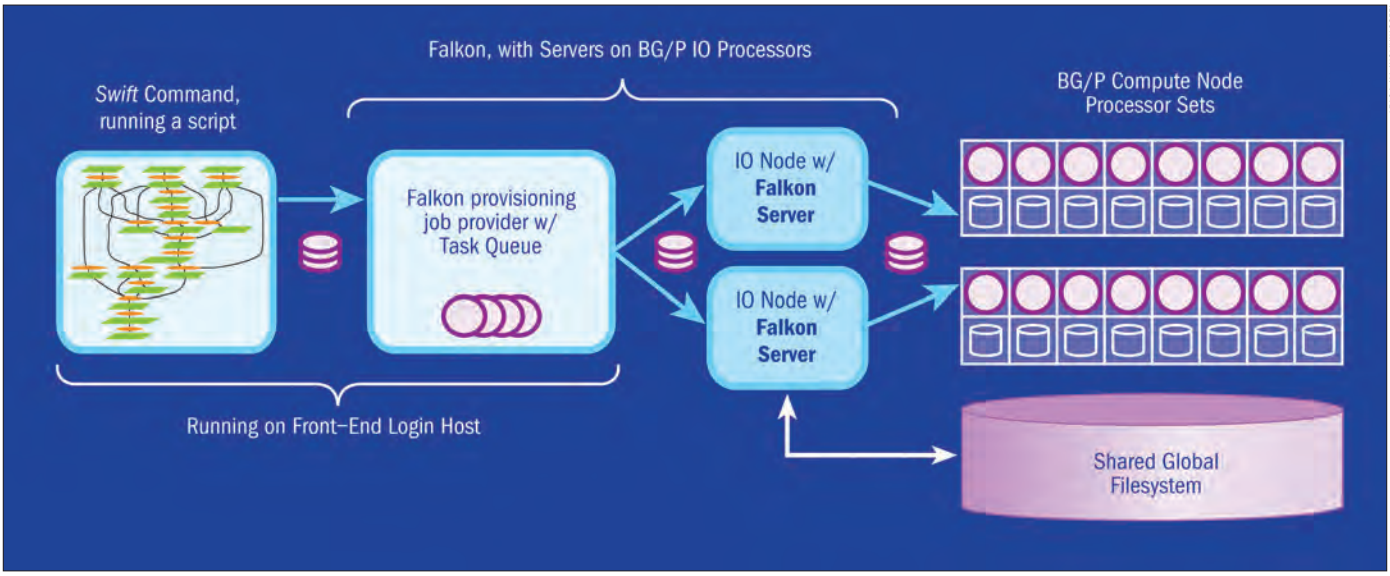

ILLUSTRATION: A. TOVEY

Figure 5. *Architecture of the Falkon provisioner on the Blue Gene/P. Falkon provides fast task execution services for Swift on the Blue Gene/P supercomputer. Its distributed servers run on up to the full set of 640 I/O nodes on Intrepid, providing highly-parallel and scalable task execution.*

largest science application benchmarks of Falkon have run over 900,000 molecular dynamics application tasks on 116,000 cores in two hours, totaling 21.4 CPU-years. This experiment was a key proof of concept for utilizing petascale machines with a scripted model of execution.

The Falkon research code has recently been reimplemented as a production Swift resource provisioner named Coasters. The name refers to the heritage of this form of provisioning, which originated with the "Glide-in" concept of the Condor high-throughput computing system. The Coasters provisioner enhances Falkon functionality with extensions for automatic deployment on petascale and distributed environments and for dynamic block allocation. Automatic deployment fully automates the mechanics of starting the provisioner's service processes and worker agents. Dynamic block allocation enables the scheduler to organize requests for required compute resources into units of varying size that can then be presented to the local resource management scheduler. This approach permits the provisioner to efficiently support the execution of scripts with fluctuating CPU demands. Coasters have made it possible to obtain increased scalability on shared petascale machines. The Coasters provisioner has been used on Ranger to run Swift scripts with nearly a million neuroscience analysis tasks in a single invocation – and this number is growing rapidly as the Coasters implementation is tuned and improved.

Collective Data Management

In the basic scripting model, programs invoked by scripts communicate by reading and writing files from a shared file system. For small scripts, it can be sufficient to allow such data interchange via whatever shared global file system is provided on the target computer: for example, on the Argonne Blue Gene/P, the General Parallel File System. For larger scripts, however, this approach rapidly becomes prohibitively slow. Instead, alternative implementation methods must be pursued. For example, one can leverage the large overall internal memory and high-performance internal interconnects of the Blue Gene/P by mapping files into RAM disk and using broadcasts to distribute files with many readers.

The development of file management methods to make scripting efficient at the petascale and beyond is a current computer science research thrust of the Swift project. Collective data management (CDM) is a prototype I/O model for file-based many-task computing. The model, inspired by collective I/O primitives from the Message Passing Interface (MPI), differs from MPI in that it operates at the file level rather than at the I/O level.

CDM uses a broadcast approach to enable efficient distribution of input data files to computing nodes and uses scatter/gather and caching methods to gather the output results from these nodes (sidebar "Collective Data Management for Petascale Scripting Performance" p46). It thus eliminates the need for tedious and error-prone manual tuning and makes the programming of large-scale clusters using a loosely-coupled model easier. The design has been prototyped for performance evaluation using simple scripts to coordinate off-the-shelf data management components. Early results indicate that such a file-based collective I/O model can handle on the order of 100,000 Blue Gene/P processors. The next major focus will be to integrate the model into the Swift parallel programming environment

The development of file management methods to make scripting efficient at the petascale and beyond is a current computer science research thrust of the Swift project.

Collective Data Management for Petascale Scripting Performance

Collective data management research for Swift – which will be of use in any similar style of "many task" computation – is focused on the design and implementation of a set of operations for exchanging files within scripts in a manner that can leverage the unique characteristics of diverse petascale systems. On the Blue Gene/P, for example, which has both a torus for point-to-point interconnects and a tree network for broadcasting data to all the nodes connected to a given input/output (I/O) processor, the Swift researchers are exploring the following approaches for collective data management.

Files and data that are read in common by all parallel instances of an application are ideally read once and are broadcast in parallel over the tree network. Such common datasets are read from their global file system location once and sent to all compute nodes, which will consume the data in parallel.

Files that are unique to each processor are "pulled" by the processor and its I/O node to the local RAM file system. This reduces I/O load by spreading the work among all participating I/O nodes (64 compute nodes per I/O node) and leverages the massive parallel I/O capability of the Blue Gene/P's shared file system. It also allows I/O to be read in efficient block sizes (for example, 256 kilobytes or more) rather than depending on the application to read efficiently, which many legacy applications cannot do and which swamps the I/O subsystem if not optimized.

Files too large to fit on RAM file systems are pulled to an aggregated "intermediate" file system, which is composed of striped RAM file systems and is accessed over the torus, again eliminating I/O from the application back to the global file system.

Output data are aggregated on the local RAM file systems, either on the compute nodes or on an "intermediate" striped RAM file system, and collected into sufficiently large batches to be written back to the global file system efficiently.

I/O between the global GPFS file system and the compute nodes can be done at high rates – over 60 gigabytes per second on the ALCF Blue Gene/P – since the system has 128 file servers serving GPFS. Accessing this system efficiently to achieve this rate, however, requires the optimizations above.

The CDM mechanism is still in the research phase, and not yet in the released version of Swift. But it is already possible to code Swift scripts to explicitly do their data management in the manner described above and to thereby significantly reduce the bottlenecks of shared I/O resources.

so that petascale users can benefit from this higher-level programming model without explicitly specifying the collective data management operations.

A Compute Node Kernel for Petascale Scripting

Implementing Swift scripts on the Blue Gene/P confronted the challenge of how to provide the POSIX interface that most off-the-shelf scientific applications require. (POSIX is the IEEE standard for "open" operating systems such as UNIX and Linux.) In particular, the Swift programming model of executing independent programs on each CPU core requires two critical operating system kernel services: *fork*, which creates a new independently executing process, and *exec*, which enables a process to execute a new application program. The standard Linux shell programs (for example, bash and tcsh) depend on variations of these two services for many of their capabilities, as does Swift for executing independent application programs on distributed and remote computing nodes as part of script execution.

While some systems, such as the Ranger supercomputer, provide a full POSIX operating system on compute nodes, the native Blue Gene/P compute node kernel does not. The solution to this problem was provided by the ZeptoOS compute node Linux kernel. ZeptoOS, the "small Linux for big computers," is a research project at Argonne seeking ways to improve the usability of Linux kernel in high-performance computing (see "ZeptoOS" under Further Reading, p53).

The ZeptoOS compute node kernel provides system services for efficiently executing POSIX-compliant application programs, as well as I/O services and access to petascale architecture capabilities such as broadcast networks, IP access to high-performance cluster interconnects, access to high-performance services for MPI communication, and enhanced memory management for large-memory tasks.

Parallel Scripting and Other Parallel Programming Models

While parallel programming models – both old and new – abound, a few models are particularly relevant to compare to the parallel scripting model: tightly-coupled parallel programming, including both message-passing and shared-memory multiprocessing; service-oriented computing; and the map-reduce programming model.

While parallel programming models – both old and new – abound, a few models are particularly relevant to compare to the parallel scripting model.

Loosely-Coupled versus Tightly-Coupled Programming Models

Swift often is called a "loosely-coupled" programming model, whereas other parallel programming approaches, such as message passing and shared-memory multiprocessing are "tightly-coupled." This concept of coupling specifically refers to two aspects of program execution: the granularity of the independent unit of parallel execution and the manner in which these independent units of execution exchange data.

In tightly-coupled parallel programs, independent streams of instructions (typically, independent statements in a program) execute in parallel and, on occasion, (at various rates) pause to exchange data with another stream of execution. This exchange takes place by sending messages in the case of the message-passing model or by synchronized access to shared data objects in the case of the shared-memory model. In contrast, in loosely-coupled parallelism, both the units of execution and of data passing are larger and more coarse-grained. The unit of parallel execution in the loosely-coupled model is a complete program, and the units of data exchange are complete and typically larger files, compared to shorter messages or objects. Furthermore, files are exchanged by reading and writing them from a file system, compared to operating solely in memory. Other persistent storage services, such as databases, can also be used for passing data between loosely-coupled parallel programs. As message-passing programs typically exchange data between independent address spaces using operating system kernel interfaces, often to access a network interconnect, and as file systems are becoming increasingly RAM-based, these differences can tend to blur. However, in general, loosely-coupled programming involves coarser-grained units of both parallelism and data exchange.

Another point of comparison between programming models can be seen in their differing task structures. While message-passing parallel programs typically map their parallel tasks statically to a finite number of processor cores, loosely-coupled programs are more likely to have a varying number of tasks at any given point in execution and are often based on a graph-structured model of execution, where the nodes of the graph represent tasks and the edges of the graph represent data objects passed between nodes. Each parallel task receives its inputs, processes to completion, and produces outputs, which can then be passed to a new task. Loosely-coupled programs are frequently composed in a structured manner, with a hierarchical "input–process–output" model, meaning that tasks can themselves be composed of graphs of subtasks, each with a similar processing model.

MPI is an example of a tightly-coupled programming model. Most of today's applications that run on high-end computers use MPI to achieve the needed inter-process communication. MPI has been an enormously popular and successful programming model. It has been the workhorse and the mainstay of parallel programming for the past two decades. The relationship between Swift and MPI is explored in the sidebar "Comparing and Connecting Swift and MPI."

Comparing and Connecting Swift and MPI

While MPI and Swift are complementary programming models (in that Swift can be used to run parallel MPI programs in the same manner as it runs serial programs), it is instructive to compare the two models in more detail.

MPI is an in-memory programming model. While MPI programs certainly read and write files, they compute on data structures in main memory. Swift, on the other hand, is a file-processing scripting language: rather than applying functions to in-memory data structures, it applies entire application programs to datasets composed of one or more files. These application programs can themselves be MPI codes, or codes written in any other parallel or serial language.

MPI programs typically (but not always) involve a static number of tasks. In Swift the number of tasks running concurrently varies dynamically, based on the parallelism of the script at any given point in its execution and then often "throttled back" by parameters and algorithms, and finally subject to the number of CPU cores that are dynamically available at any given time on the collected set of resources available to the running Swift program.

In Swift, unlike MPI, failure of a single node (or program) affects only the program(s) running on that node at the time of the failure. Swift maintains the state of each running script in a log file, which allows it to restart a parallel script from the point of failure. Only uncompleted tasks are re-executed upon a restart. And since each unit of execution is a complete program, these programs can be re-executed when they fail.

Swift is complementary to, and not a replacement for, shared memory or message-passing multiprocessing. Swift has been and will increasingly be used to coordinate the execution of MPI applications, creating a loosely-coupled ensemble of tightly-coupled programs.

Service-Oriented Model

Services – more precisely, web services – are network-accessible data processing functions that follow a coarse-grained functional programming model similar to that used in parallel scripting. Rather than consuming and producing files, however, web services consume and produce XML documents, which can be small and simple or large and complex. Individual web service functions are bundled into services that, once deployed, have a network address and, during a typically long lifetime, handle many service function invocations. Several scripting languages, usually called "workflow" or "orchestration" languages, execute programs that consist of multiple web service invocations. The services themselves are responsible for mapping invocation requests to the processors on which the service has been "provisioned." Web services are heavily used in commercial applications, for e-commerce, social networking, supply chain management, and myriad business applications. They are finding increasing use in scientific computing, primarily in applications where the arguments and results of a scientific function can be conveniently expressed as an XML document. There is much interest in blending the data models of parallel scripting and web services to provide powerful interoperability between these complementary programming models.

The number of applications to which parallel scripting has been applied is growing rapidly. Swift users have run applications in biochemistry, bioinformatics, economics, neuroscience, and radiology, with an increasing number of users executing on petascale machines.

Parallel Scripting Applications

Parallel scripting as described here is being applied in many different disciplines. The applications include the sciences and computational economics, and the characteristics range from hundreds to one million one-core simulations. Some of the applications listed here are already operational, others are in development, and a few were experimental efforts.

Operational

Biology
- Analysis of mass spectrometry data for post-translational protein modifications (has run on petascale machines)
- Protein structure prediction using iterative fixing; exploring other large-biomolecule interactions (has run on petascale machines)
- Identification of drug targets via computational docking/screening (has run on petascale machines)

Economics
- Generation of response surfaces for various economic models
- Analysis of uncertainly in large-scale economic models of climate and energy-related factors (has run on petascale machines)

Neuroscience
- Analysis of functional MRI datasets for studies in language and stroke recovery (has run on petascale machines)

In Development

Biology
- Protein structure prediction using Raptor threading algorithm with linear programming
- Metagenome modeling with integer programming
- Metagenome analysis with large-scale BLAST and phylogenetic applications (has run on petascale machines)
- Mining of large text corpora to study media bias

Cardiology
- Chesnokov analysis of ECG datasets for the cardiovascular research grid

Earth Systems
- Analysis of NASA MODIS satellite data using R

Neuroscience
- Analysis of large-scale image data from *C. elegans* experiments

Radiology
- Training of computer-aided diagnosis algorithms
- Image processing and brain mapping for neurosurgical planning research

Experimental

Astronomy
- Creation of montages from large sets of digital images
- Stacking of cutouts from digital sky surveys

Earth Systems
- Ensemble climate model runs and analysis of output data

Functional Models versus Map-Reduce

Another parallel and distributed programming model that has been gaining significant interest is map-reduce. This model has its origins in functional programming, dating back to early LISP, which implemented in a programming language the concepts of mapping a function to a set of arguments and of reducing a set of independent results back into a single answer. The force behind map-reduce has been Google, which utilizes it heavily to perform myriad aspects of the vast internal data processing behind its search and information services. In Google's map-reduce programming model, information is represented as pairs of textual keys and their associated arbitrary data values. Computation is performed by breaking a large dataset into key-value pairs, applying a processing function to each pair through the "map()" operation, and then gathering and reducing all the results with a distributed sort-and-merge mechanism that utilizes the fact that all data objects have keys.

The map stage of map-reduce is similar to the processing model of parallel scripting in that functions are applied to arguments in parallel. In most usage, map-reduce functions are more like in-memory application functions, since map-reduce views functions and data more as in-memory objects. The reduce stage takes unique advantage of this simple data model to perform reduction as a highly-parallel operation. In parallel scripting, applications that require such a parallel reduction implement it explicitly within the programming model, whereas in map-reduce the reduction is a built-in part of the framework.

Applications Leveraging Parallel Scripting at the Petascale

The number of applications to which parallel scripting has been applied is growing rapidly (sidebar "Parallel Scripting Applications"). Swift users have run applications in biochemistry, bioinformatics, economics, neuroscience, and radiology, with an increasing number of users executing on two petascale machines: the Intrepid Blue Gene/P at Argonne and the Ranger Constellation at the University of Texas–Austin.

Case Study: Protein Structure Prediction

University of Chicago researchers under Drs. Karl Freed and Tobin Sosnick use Swift to run the Open Protein Simulator (OOPS) to predict 3D protein structure for which little or no similar structure information is known. OOPS comprises a set of open-source applications for fast simulation of protein folding, docking, and refinement. It runs an iterative fixing algorithm called ItFix (figure 2, p40) that consists of multiple rounds of many parallel simulations. At each round ItFix carries out between 100 and 1,000 Monte Carlo simulated annealing computations. The statistical data from that round include the average origins of the secondary structures at each position in a genome sequence. Additionally, at each analysis step, ItFix creates plots from the output data, including average 3D contact maps. One of the limiting factors in widely applying ItFix, however, is that that the algorithm requires approximately 1,000 CPU-hours on a modern microprocessor for a medium-sized protein.

Swift was first applied to this problem by Glen Hocky, then a University of Chicago undergraduate chemistry student, now pursuing his Ph.D. at Columbia. Using Swift on thousands of processors. Hocky was able to rapidly improve on the accuracy of prior prediction results done on a smaller departmental cluster. By using the Swift parallel scripting system within OOPS, the Chicago protein researchers have been able to realize several benefits: concise, readable specification of high-level structure that exposes opportunities for parallel execution; robust, fault-tolerant management of large numbers of tasks; and convenient dispatch of computation to multiple parallel computers, both local and remote, without modification to their science scripts.

The OOPS framework fits into Swift in a natural and straightforward manner. Swift data typing and mapping were leveraged to abstract input and output, detect type errors, and map the simple logical structure to the specific data layout desired in the archival storage repository. Atomic Swift dataset types were declared to represent the files used by the OOPS application programs (such as FASTA, for the sequence being folded, and PDB, for files in the standard set by the Protein Data Bank for representing 3D protein structure). New Swift compound dataset types were declared to organize multiple related values for program inputs and output. Swift procedures were used to define interfaces to application codes and, in some cases, to define interfaces to small utility functions. Parallel application logic to specify how a single ItFix round is performed was defined as Swift functions. The main program then was coded, in which ItFix is called to predict the structure of a single protein, or to express much more complex science tasks such as parameter sweeps, structure comparisons, or explorations involving an entire genome.

The new Swift-based approach was used to test prediction capabilities for the structure of specific alpha-, alpha/beta-, and beta-proteins. Because the structure of these proteins was well known, they provided excellent test cases. For the alpha-protein investigation, the University of Chicago–Argonne team ran approximately 5,000 simulations using TeraGrid resources. The structures predicted were comparable to or better than the best published results and – most significant – used two orders of magnitude less computation time. For the alpha/beta- and beta-protein investigations, since a large amount of in-simulation sampling was required, the researchers ran their tests on the Blue Gene/P. Even though the per task runtimes were longer on the slower Blue Gene/P CPUs than on modern stock processors, the results confirmed that petascale systems such as Blue Gene will be useful in folding investigations using the ItFix algorithm with Swift.

Case Study: Protein–Ligand Docking

A good example of the value of parallel scripting on petascale machines is virtual drug screening. One of the first applications was to screen core metabolic targets against drug-candidate compounds from the ligand database of KEGG, the Kyoto Encyclopedia of Genes and Genomes. Argonne biochemists have applied parallel scripting to simulate the docking of small ligand molecules to the active sites of macromolecules. The application is of interest to the pharmaceutical industry in that compounds that interact strongly with a macromolecule associated with a particular disease may inhibit its function. This application is being run on up to 64,000 cores of Argonne's Blue Gene/P (figure 6, p50).

Development of antibiotic and anticancer drugs is a process fraught with dead ends. Each dead end costs potentially millions of dollars, as well as wasted years and lives. Computational screening of protein drug targets helps researchers prioritize targets and determine leads for drug candidates. While computational screening, which is relatively inexpensive, cannot replace wet lab assays, it can significantly reduce the number of dead ends by providing more qualified protein targets and leads.

In one typical computational screen, nine proteins that perform key enzymatic functions in the core metabolism of bacteria and humans were selected for screening using the DOCK6 molecular docking simulator from the University of California–San Francisco against a database of 15,351 natural compounds and existing drugs in KEGG's ligand database. The goal of this project was to validate the ability to approximate the binding mechanism of the protein's natural ligand, to determine key inter-

The results confirmed that petascale systems such as Blue Gene will be useful in folding investigations using the ItFix algorithm with Swift.

A. Binkowski and M. Kubal

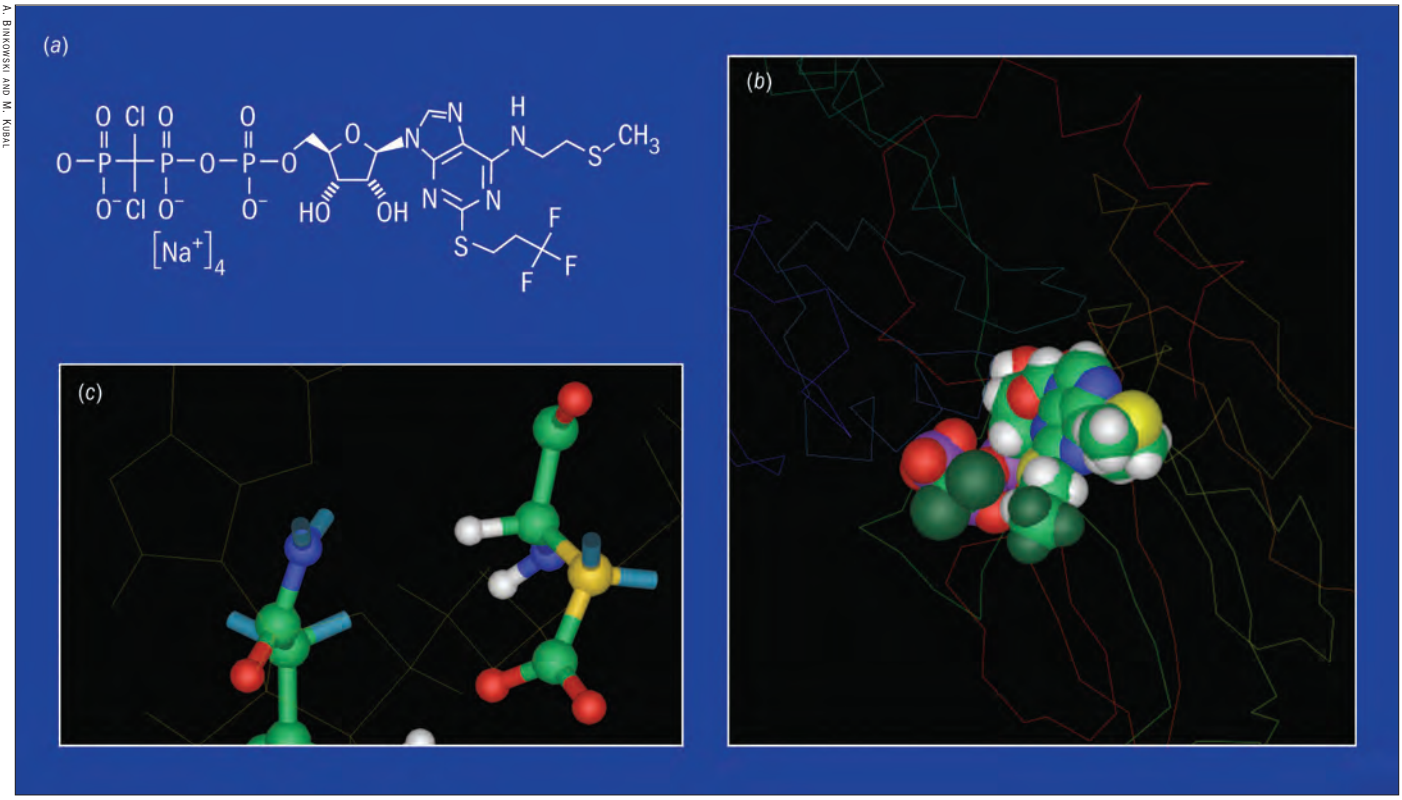

Figure 6. *Molecular docking simulations for drug discovery through parallel scripting. (a) 2D representation of best scoring compound, KEGG drug compound D03361, against protein NAD kinase. (b) Spherical representation of D03361, Cangrelor tetrasodium, docked in pocket of NAD kinase (only showing backbone of protein as wireframe). (c) Close-up of NAD kinase (backbone removed) with a side-chain carbon atom of residue ASP209 in yellow binding to an oxygen of D03361 (gold wire frame), and residue ASN115 interacting with core rings of D03361.*

action pairings of chemical functional groups from different compounds with the protein's amino acid residues, and to study the correlation between a natural ligand that is similar to other compounds and its binding affinity with the protein's binding pocket. As part of the process, the scientists sought to prioritize the proteins for further study.

The computation of the binding affinity between each compound in the database and each protein was performed with 138,159 runs of DOCK6 on the ALCF's Blue Gene/P. On a single 2 GHz machine this run would have taken approximately 48 days. Using two racks (8,192 cores) on Blue Gene/P these runs took three hours. This computation is, however, just the beginning of a much larger computational pipeline that will screen millions of compounds and tens of thousands of proteins. The downstream stages use even more computationally intensive and sophisticated programs that provide for more accurate binding affinities by allowing for the protein residues to be flexible and the water molecules to be explicitly modeled.

This computation was easy to perform and yielded fast and useful results. The natural compound for six of the targets scored reasonably well in terms of interaction energy and ranking (two in the top 2%, two in the top 10%, and two in the top 16%), especially considering these are natural compounds that rely on higher concentration levels for enzyme interaction compared to optimized inhibitors that rely on higher binding affinities. Reviewing the 3D structures of the compound-protein complexes generated by DOCK6 provided insight into the hydrogen bonding and chemical functional group placement within the pocket required for tight binding. For seven of the proteins targets, existing drug compounds were the top hit. These included an anti-platelet agent, an anti-hypertensive agent, a treatment for chronic dry eye, a vitamin precursor, and an opthalmic agent. Since these compounds have already undergone some degree of testing for human use, performing follow-up wet lab assays for inhibition could accelerate the discovery of a novel application for an existing drug.

Case Study: Proteomics Research

Drs. Yingming Zhao and Yue Chen, researchers at the University of Chicago Ben May Department for Cancer Research, are applying parallel scripting to the analysis of post-translational protein modifications (PTM). Such modifications play essential roles in living cells, dynamically regulating physiological processes by fine-tuning protein functions. Despite the importance of PTMs

in cellular processes, however, accurate identification of PTMs has been a challenging task.

Over the past decade, mass spectrometry has become an indispensable tool for accurate and sensitive identification of PTMs. Mass spectrometry is capable of experimentally measuring both the precursor mass and fragmentation patterns of all the peptides in a protein. Using such information, scientists can predict the theoretical mass and fragmentation pattern of the peptides in each protein.

Nevertheless, while commercial software tools have been widely applied in large-scale protein identification, they are incapable of the identification of unexpected or unknown protein modifications. To rectify this situation, Drs. Zhao and Chen developed PTMap, a tool for genome-wide characterization of protein post-translational modifications.

PTMap works as follows. The data from a mass spectrometer run on samples from a single organism is grouped into 50 or more "fractions," each on the order of 100–200 megabytes. Once a dataset is captured, analysis consists of executing a run of the PTMap workflow script – a set of parallel invocations of the PTMap application that search for PTMs by analyzing each mass-spec fraction against a number of FASTA protein sequences, from one to hundreds of the tens of thousands of proteins in most organisms of interest. Then the entire process is repeated, comparing against the FASTA sequences in reversed and permuted order to eliminate matches that are due to chance. The number of proteins compared in each invocation of PTMap can be used to control the overall degree of parallelism of the overall workflow run. Each such run may need to be repeated many times, for new datasets, new organisms, or newly improved versions of code or to explore the effects of varying parameters. The software also uses unmatched peaks to remove false positive identification. Compared with other software tools for PTM identification, PTMap shows higher accuracy and sensitivity and has been applied to successfully characterize novel lysine propionylation and butyrylation sites in yeast core histones.

Since each analysis requires the same input of mass spectrometry data and differs only in the candidate protein sequence, theoretically all the analysis can be performed in parallel to achieve high speed, and the final result can be integrated from all analysis results. Based on this principle, Drs. Zhao and Chen have developed PTMap2 to extend the capability of PTM analysis on a single protein to genome-wide analysis on a protein database. Tests show that PTMap2 can identify all types of PTMs in the *E. coli* protein database from a single mass spectrometry data file. Using petascale computing resources, the researchers currently are extending the application of PTMap2 to identify unknown PTMs in diverse cellular organisms.

Case Study: Economics, Environment, and Energy

Drs. Joshua Elliott and Meredith Franklin are collaborating on two University of Chicago–Argonne projects: "Social, Economic, and Environmental Modeling" and "Community Integrated Model of Economic and Resource Trajectories for Humankind." Using Swift on large Grid clusters, the researchers have conducted parameter sweeps of economic models of energy use. Recently they and their colleagues analyzed more than 10,000 models from a perturbed input dataset in parallel on TeraGrid and Open Science Grid resources (figure 7).

Dr. Tiberiu Stef-Praun of the University of Chicago/Argonne Computation Institute uses Swift to run parallel economic modeling tasks in MATLAB, Octave, and Python. Working with Dr. Robert Townsend of MIT, he integrates macroeconomic and microeconomic models of developing economies, focusing on entrepreneurship, access to the financial system, and individual choices. The models, run on a variety of computing resources including TeraGrid and the ALCF Blue Gene/P, have been used to explore topics such as evaluating choices of group organization for risk sharing purposes, linking growth to financial deepening and inequality, and borrowing choices.

Case Study: Neuroscience

Researchers in the University of Chicago Human Neuroscience Laboratory led by Dr. Steven Small use Swift to analyze data from functional magnetic resonance imaging (MRI) experiments. Dr. Small's laboratory makes extensive use of the R data analysis language and has leveraged the power of relational databases for the storage of time series signals of brain activity during cognitive experiments extracted from functional MRI images. The researchers benefit from the ability to run their data processing and analysis tools on a range of computing resources, from a local Condor pool that aggregates their lab's workstations into a fast but limited local cluster, to the resources of TeraGrid. Recently, they have started to use the ALCF Blue Gene/P for their structural equation modeling data analysis procedures.

Students in the laboratory of Dr. David Biron of the University of Chicago Department of Physics are applying Swift to analyze high-volume image data from biophysics experiments in which the behavior of large arrays of the model organism *Caenorhabditis elegans* are recorded with high-speed digital cameras. The researchers anticipate ultimately producing close to a terabyte a day of image data that will be analyzed using the MATLAB image-processing toolbox. Swift is able to call compiled MATLAB analysis scripts that can

Using Swift on large Grid clusters, researchers have conducted parameter sweeps of economic models of energy use. Recently they and their colleagues analyzed more than 10,000 models from a perturbed input dataset in parallel on TeraGrid and Open Science Grid resources.

J. ELLIOTT, M. FRANKLIN, AND A. ESPINOSA ILLUSTRATION: A. TOVEY

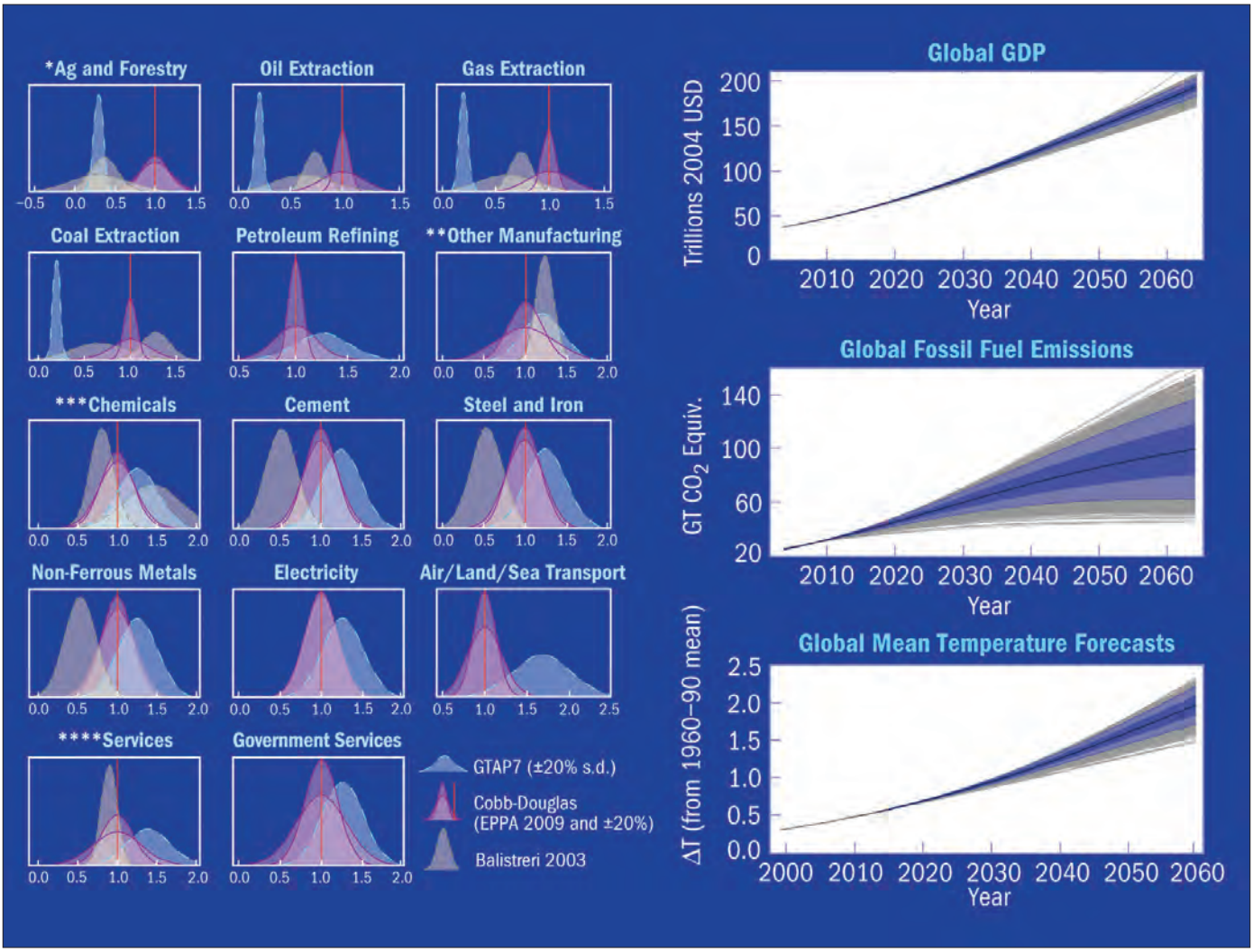

Figure 7. *CIM–EARTH simulations. Left, a comparison of uncertainties in the substitutability of labor and capital for a variety of industries. Top right, global GDP for 5,000 model runs with perturbed substitution elasticities. Right center, global emissions from fossil fuel consumption for 5,000 model runs with perturbed substitution elasticities. Bottom right, global mean temperature forecasts.*

execute on a variety of parallel clusters and Grid resources available to Dr. Biron's lab and to handle all the data transfer between the data capture system and the analysis computers in a seamless and transparent fashion.

What's Next?

The development and usage of Swift are proceeding in many new directions. Interesting challenges remain, in terms of not only engineering and productization but also open research questions in parallel programming methodology, systems architecture, and performance.

One effort focuses on platform support. Most Swift supercomputer studies have been performed on the IBM Blue Gene/P and Ranger. The Swift team members plan to explore the use of Swift on other petascale systems such as the Cray XT5.

Another effort involves ease of use. Argonne researchers Wenjun Wu, Tom Uram, and Dr. Mark Hereld have created a flexible portal framework that uses Web 2.0 interfaces to create a customized online interface for executing parallel scripts and organizing, visualizing, and sharing results. With this new framework, scripts can be added to the repertoire of computational tools without manually creating a new web page for each script. The portal mechanism will soon automate the translation of a Swift script's functional interface – created by members of a science team that are proficient in scripting – to a web task-submission form that can be readily used by non-programming science users. An example of a portal interface created for the Open Protein Simulator is shown in figure 8.

One broad area of Swift usability concerns expressivity. Swift researchers are exploring how best to express problems that lend themselves to map-reduce solutions and how to unify the notions of data typing and data passing so that web

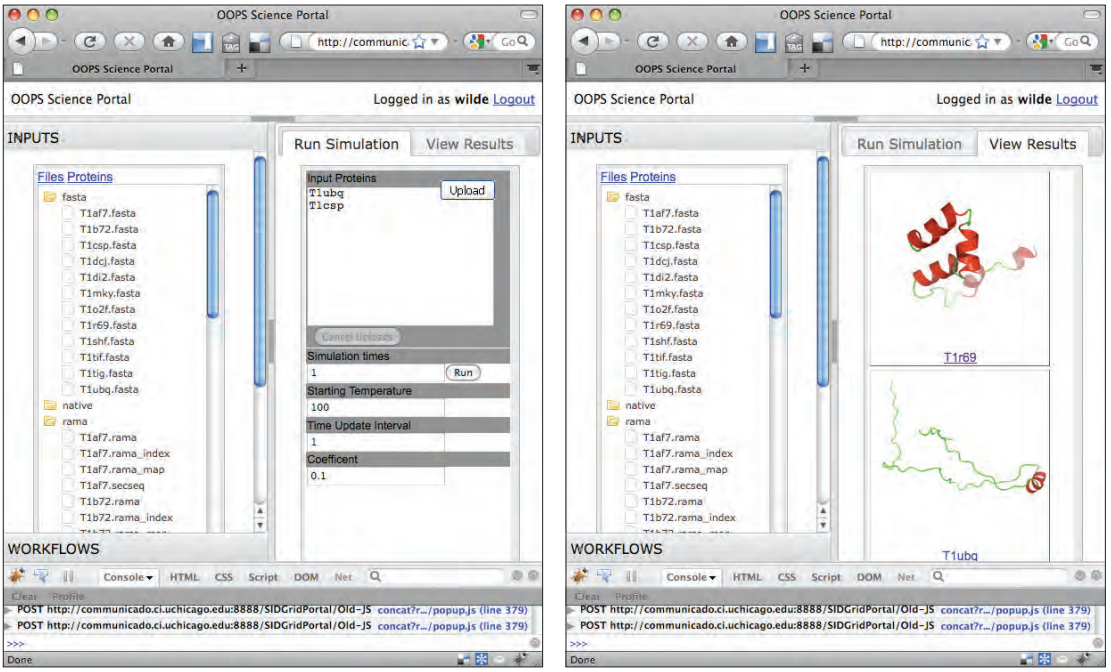

Figure 8. *Science portal for parallel script execution. Script execution request form (left) and results page (right). Portal hosted at UChicago Computation Institute. Workflow executed on TeraGrid system "Abe" at UIUC.*

As the high-performance community turns its attention to the daunting problems of exascale computing, Swift is expected to prove a valuable technology and paradigm in the programming model toolkit.

services and databases can be readily integrated into Swift scripts in a seamless manner with file-based application programs. Swift researchers are also exploring how best to integrate the Swift data typing and access model with structured storage methods such as HDF5 and NetCDF and how to integrate Swift file passing with the message-passing model. These structured storage mechanisms are playing an increasingly vital role in scientific data management. Research that can reduce the differences between such structured storage and hierarchical file systems can enable applications to achieve scalability at the petascale and beyond, without sacrificing the flexible exploration, management and exchange of data made possible by modern file systems.

Another area of interest is to make the Swift data management, program execution, and data-passing semantics available as native libraries for popular scripting systems – languages such as Perl, Python, and Ruby, as well as the various Java-based scripting languages such as Groovy. While these serial languages do not have the innate parallelism and simplicity of the Swift language, they do have vast user communities who could benefit from the parallel execution and data management support provided by the Swift runtime system, while continuing to express their scripts in a language with which they are already familiar.

As the high-performance community turns its attention to the daunting problems of exascale computing, Swift is expected to prove a valuable technology and paradigm in the programming model toolkit. Increasingly, Swift will be called on to serve as the "outer loop" for running large numbers of applications in parallel, each of which is using a large number of CPUs in a tightly-coupled manner. Swift's ability to re-execute the work from any number of failing CPUs is well suited for the exascale world where the processor failure counts within a large computing complex may rise over today's level. Swift's data dependency graph can be used to determine what needs to be re-executed when complex failures make previously computed data objects unavailable. And, as every desktop workstation and mobile device becomes an N-way multicore system, Swift will provide a uniquely scalable solution to a broad range of problems for an expanding community of users. ●

Contributors Dr. Pete Beckman, Dr. Ian Foster, and Michael Wilde, Argonne National Laboratory and University of Chicago; Dr. Ioan Raicu, Northwestern University

Further Reading:

Swift
http://www.ci.uchicago.edu/swift/
ZeptoOS
http://www.mcs.anl.gov/research/projects/zeptoos/
J. Ousterhout. 1998. Scripting: Higher-Level Programming for the 21st Century. *IEEE Computer* **31(3)**: 23–30.
M. Wilde, Z. Zhang, B. Clifford, M. Hategan, S. Kenny, K. Iskra, P. Beckman, I. Foster, I. Raicu, and A. Espinosa. 2009. Parallel Scripting for Applications at the Petascale and Beyond. *IEEE Computer* **42(11)**: 50–60.