Extreme-scale scripting: Opportunities for large task-parallel applications on petascale computers

Michael Wilde, Ioan Raicu, Allan Espinosa, Zhao Zhang, Ben Clifford, Mihael Hategan, Kamil Iskra, Pete Beckman, Ian Foster

Computation Institute,
University of Chicago and Argonne National Laboratory

Abstract. Parallel scripting is a loosely-coupled programming model in which applications are composed of highly parallel scripts of program invocations that process and exchange data via files. We characterize here the applications that can benefit from parallel scripting on petascale-class machines, describe the mechanisms that make this feasible on such systems, and present results achieved with parallel scripts on currently available petascale computers.

1. The parallel scripting paradigm

John Ousterhout describes scripting as *higher-level programming for the 21st Century*[3]. Scripting has revolutionized application development on the desktop and server, accelerating and simplifying programming by focusing on the composition of programs to form more powerful applications. Understanding how to scale scripting to 21st century computers should be among the priorities for researchers of next generation parallel programming models. Might scripting not provide the same benefits for extreme-scale computers?

We believe that the answer to this question is *yes*. We introduce the concept and implementation of *parallel scripting*, and describe what becomes possible when simple scripts turn "ordinary" scientific programs into petascale applications running on 100,000 cores and beyond. Scripting languages allow users to assemble sophisticated application logic quickly by composing existing codes. In parallel scripting, users apply parallel composition constructs to existing sequential or parallel programs. Using this approach, they can quickly develop highly parallel applications that can be run efficiently on a 16-core workstation, a 16,000-core cluster, or a 160K-core petascale system.

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

We have been exploring such "many task" computing models[6] for several years, from the perspective of both technologies and applications. On the technology front, we have explored,

in particular, a dataflow-driven parallel programming model that treats application programs as functions, and their datasets as structured objects mapped to a simple abstract data model. We have incorporated this model in a parallel scripting language, Swift, and implemented that language on large parallel computers, including a 160K-core Blue Gene/P and a 62K-core Sun Constellation. Swift programs may define hundreds of thousands (and soon millions) of tasks that read and write an even greater number of files. We have developed task and data management methods that can scale to extremely high dispatch rates and data volumes, and used them to scale applications to up to 160K cores, with high efficiency and fault tolerance.

2. Software architecture for petascale parallel scripting

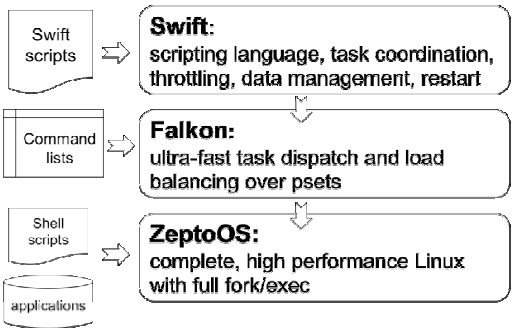

Figure 1: Architecture for petascale scripting

Our approach to high-performance parallel scripting requires three layers, which from the bottom up are: (1) a POSIX environment in which to execute the individual application tasks of a script; (2) a means to allocate compute node resources, hold them for long and varying periods of time (typically hours rather than seconds) while rapidly scheduling small independent tasks on the compute nodes – even tasks of very short duration (down to fractions of a second, but typically many seconds or minutes); and (3) a language in which to abstractly express graphs of highly parallel application invocations, their data accesses, and the data interchange between them.

Modern concepts of scripting depend on POSIX system services such as fork() and exec() in order to execute application programs from the script, and this require an operating system that supports these or similar capabilities – notably, the ability to launch a new application program and wait for it to complete. On the BG/P, the native IBM compute node kernel lacks these features, and we provide them instead through the ZeptoOS compute node kernel[4], which implements these features in a POSIX-compliant manner. On the Ranger Constellation system, we use the native compute node operating system, which provides complete POSIX support.

3. The Falkon resource provisioner and lightweight scheduler

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

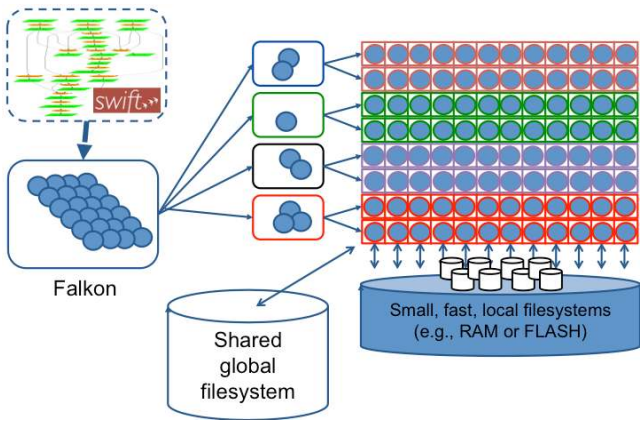

Figure 2: Falkon Provisioning and Scheduling System

implemented in a component called *Falkon*, a Fast and Light-weight tasK executiON framework[4].

Falkon allocates nodes of a compute resource in large quantities, using the native batch scheduler of the system, and runs a persistent task execution agent on each compute core that rapidly executes arbitrary and independent POSIX processes on the allocated nodes. Falkon consists of several components, shown in Figure 2: (1) a compute node agent that executes one task at a time on a compute node core; (2) a service that maintains a queue of jobs for a set of compute node resources, and which rapidly selects the next job to run on a FIFO basis; and (3) a load-balancing client that evenly distributes work to the services

Using Falkon, we have been able to meet performance requirements necessary for petascale scripting. Measurements of Falkon performance[4] indicate it can:

- execute over 3,000 tasks per second on the BG/P;
- launch, execute, and terminate 160K tasks on the BG/P at 160K-core scale in under one minute;
- execute workloads of 913K science tasks on 116K BG/P cores in 2 hours, totalling 21.4 CPU years at 99.7% efficiency and 99.6% utilization (Figure 3); and
- execute one billion trivial tasks in 18 hours in multicore stress tests.

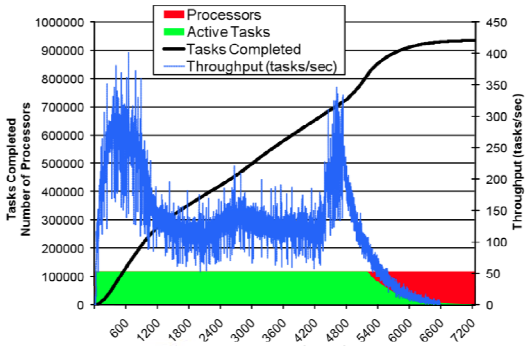

Figure 3: Falkon performance (DOCK application)

Users utilize Falkon by creating simple scripts that contain a list of tasks to execute, with arguments. Tasks are executed with FIFO scheduling and with maximum parallelism. Since tasks are sent to available agents in the order that they appear in the Falkon input script, users can sort tasks "longest first" when task duration can be estimated, thus achieving optimal utilization of a block of compute nodes. Falkon functionality has been packaged as a Swift "execution provider" to enable higher level programming (described in the next section). It has also been implemented as a newer Swift execution provider, which has the additional capability of allocating multiple "time-space" blocks of varying number of cores and wall-time duration, to best fit the current task demand.

4. The Swift parallel scripting language

Swift is a scripting language that makes it easy for a user to specify and run application programs in parallel, and to specify the data passing and dependencies between different application invocations, as well as the structure of the data (typically in terms of files and directories). It provides run time support that automates data management and enables the same Swift script to run on multi-core workstations, local clusters, remote grids, cloud resources, and petascale supercomputers. It is common to write Swift applications in three layers: a top layer which specifies the workflow, a middle layer of interface code to adapt specific applications, and the lowest layer, which defines the interfaces of the applications themselves. Studies on early versions of Swift have shown that the amount of code needed to express applications in this form is substantially lower than by *ad hoc* scripting in "shell scripts" [9]

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

To show the power of the Swift language, here is a fragment of Swift code to perform protein-folding simulations using an application called "ItFix"[1]. This simple code fragment, given ten

```
foreach prot in protein {
  foreach sT in startT {
    foreach tUp in tUpdate {
      ItFix(prot, nsim, maxrounds, sT, tUp);
    }
  }
}
```

protein sequences, nsim=1000, two starting temperatures and five update intervals, will, in each of three (maxrounds) rounds of prediction, execute 10 x 1000 x 2 x 5 = 100,000 simulations. ItFix scripts been executed on up to 32,000 cores on the Argonne BG/P. Similar code can sweep across any combination of ItFix parameters. This abstract script runs without change on multiple TeraGrid clusters including the 62K-core supercomputer "Ranger".

The Swift data model provides the ability to describe nested on-disk directories as simple structures and arrays. These datasets are transparently sent to remote and parallel Swift procedures on various platforms. An operation called "mapping" translates between the simple abstract data model of Swift and the potentially messy, complex model of real-world directory structures and file naming and structuring conventions.

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

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

5. Example applications

Applications to which we have applied large-scale parallel scripting [1][2][7][8] are listed in Table 1. Each is capable of consuming a large fraction, or even all, of a petascale computer. All involve doing many tasks at once, with often quite substantial amounts of communication both within each task and among tasks.

Table 1: Parallel Scripting Applications

Field	Description	Characteristics	Status
Astronomy	Creation of montages from many digital images	Many 1-core tasks, much communication, complex dependencies	E
Astronomy	Stacking of cutouts from digital sky surveys	Many 1-core tasks, much communication	E (Falkon)
Biochemistry	Analysis of mass-spec data for post-translational protein modifications	10,000 – 100,000 K jobs for proteomic searches using custom serial codes	D
Biochemistry	Protein folding using iterative fixing algorithm, also exploring other biomolecule interactions	100s to 1000s of 1-1000 core simulations & data analysis	O
Biochemistry	Identification of drug targets via computational screening	Up to 1M x 1 core	O (Falkon)
Bioinformatics	Metagenome modeling	1000's of 1-core integer programming problems	D
Business economics	Mining of large text corpora to study media bias	Analysis and comparison of 70M+ text files of news articles	D
Climate	Ensemble climate model runs and analysis of output data	10s to 100s of 100-1000 core simulations	F
Economics	Generation of response surfaces for various economic models	1K to 1M 1 core runs (10K typical), then data analysis	O
Neuroscience	Analysis of functional MRI datasets	Comparison of images; connectivity analysis with SEM, many tasks (100K+)	O
Radiology	Training of computer aided diagnosis algorithms	Comparison of images; many tasks, much communication	D
Radiology	Image processing and brain mapping for neurosurgical planning research	1000's of MPI application executions	O

Legend: E: evaluated, D: in development, O: in some degree of operational use

6. Conclusion

Our experience in applying Swift indicates that it enables the productive programming paradigm of knitting together existing application programs using a functional model to rapidly and productively compose new, higher level applications that can efficiently use the parallel resources of a petascale system. We are currently developing enhancements that apply

"collective" data management techniques[11] to efficiently move data in and out of applications in a way that leverages the interconnect and filesystem hardware of petascale systems. We believe that, when complete, the parallel scripting model will play an indispensible role in the extreme-scale programming tool chest.

Acknowledgements.
This research is supported in part by NSF grants OCI-721939 and PHY-636265, NIH grants DC08638 and DA024304-02, the Argonne/DOE LDRD program, NASA Ames Research Center GSRP grant NNA06CB89H, and the UChicago/Argonne Computation Institute.

References

[1] Hocky, G., Wilde, M., Debartolo, J., Hategan, M., Foster, I., Sosnick, T.R., and Freed, K.F.). Towards petascale ab initio protein folding through parallel scripting. Argonne Technical Report ANL/MCS-P1612-0409.

[2] Kenny, S, M. Andric, M. Wilde, Michael C. M. Neale, S. Boker, SL. Small, Parallel Workflows for Data-Driven Structural Equation Modeling in Functional Neuroimaging, Argonne Technical Report ANL/MCS-P1613-0409.

[3] Ousterhout, J. Scripting: Higher Level Programming for the 21st Century IEEE Computer March 1998

[4] Raicu, R,. Zhao Zhang, Mike Wilde, Ian Foster, Pete Beckman, Kamil Iskra, Ben Clifford. "Toward Loosely Coupled Programming on Petascale Systems", IEEE/ACM Supercomputing 2008.

[5] Raicu. I. "Many-Task Computing: Bridging the Gap between High Throughput Computing and High Performance Computing", Doctorate Dissertation, Computer Science Department, University of Chicago, March 2009

[6] Raicu, I, Ian Foster, Yong Zhao. "Many-Task Computing for Grids and Supercomputers", Invited Paper, IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08), 2008, co-located with IEEE/ACM Supercomputing 2008.

[7] Small, S. L., Wilde, M., Kenny, S., Andric, M., & Hasson, U. (2009). Database-managed Grid-enabled analysis of neuroimaging data: The CNARI framework. *International Journal of Psychophysiology, in press, published online 2/20/2009*

[8] Stef-Praun, I. Foster, U. Hasson, M. Hategan, S.L. Small and M. Wilde, Accelerating medical research using the Swift Workflow System, *Paper Presented at the HealthGrid 2007, Geneva* (2007).

[9] Zhao, Y., Dobson, J., Foster, I., Moreau, L., Wilde, M., A Notation and System for Expressing and Executing Cleanly Typed Workflows on Messy Scientific Data, SIGMOD Record, September 2005.

[10] Zhao, Y., Hategan, M., Clifford, B., Foster, I., von Laszewski, G., Nefedova, V., Raicu, I., Stef-Praun, T., Wilde, M. "Swift: Fast, Reliable, Loosely Coupled Parallel Computation," *IEEE Workshop on Scientific Workflows 2007*.

[11] Zhang, Z., Allan Espinosa, Kamil Iskra, Ioan Raicu, Ian Foster, Michael Wilde, "Design and Evaluation of a Collective I/O Model for Loosely-coupled Petascale Programming", IEEE Workshop on Many-Task Computing on Grids and Supercomputers (MTAGS08), co-located with IEEE/ACM Supercomputing, 2008.