Flexible Resource Allocation and Data Management for High-Performance Distributed Workflows

Jacob M Taylor
Wayne State University
42 W. Warren
Detroit, MI 48202
ez5504@wayne.edu

Yadu N Babuji
Computation Institute
5735 S Ellis Ave
Chicago, IL 60637
yadunand@uchicago.edu

Justin M Wozniak
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
wozniak@mcs.anl.gov

Michael Wilde*
Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439
wilde@anl.gov

ABSTRACT

Over the past decade, high-performance computing has become an important tool in a variety of areas, stretching from cosmological to subatomic simulations. With supercomputers, clusters, grids and clouds, the landscape of target computing environments is highly diverse. Often, it can be challenging to utilize all the available resources efficiently. Remote execution paired with a highly-scalable, implicitly parallel programming language can help to solve this problem of efficiency by not only making the resources more available, but by also helping to manage their use. We illustrate here how a workflow-level HPC programming model that was recently scaled up to extreme-scale systems can also target more dynamic multiple-job, multiple-system deployments by integrating it with more flexible resource allocation and data management strategies.

Categories and Subject Descriptors

D.3.2 [**Programming Languages**]: Language Classifications—*concurrent, distributed, and parallel languages*

Keywords

parallel computing, remote execution

1. INTRODUCTION

High-performance computing has become an invaluable tool for many areas of science over the past decade. With the ability to compute in minutes or hours what would take humans years to compute by hand, the various systems that compose the field are both incredibly powerful and diverse. The diversity often adds more complexity than the average user can handle. The majority of programming languages tend to complicate parallelism for the average user. Swift/T, however, is designed as an implicitly parallel language that is highly scalable for large systems and is primarily aimed at scientific users and their applications [3].

Swift/T improves on its predecessor, Swift/K, by providing an extremely scalable workflow engine. It is built around three main components: the Turbine runtime, the STC compiler, and ADLB (load balancer) libraries. The compiler converts native Swift/T code into Tcl, which is then executed by Turbine[3]. ADLB distribute tasks to Turbine workers [3, 7] and enables rapid task distribution, and transforms the single-node JVM engine of Swift/K into a scalable, multi-node VM that reduces latency and increases scalability.

2. MOTIVATION

While Swift/T is vastly more scalable than Swift/K, the older language is still powerful, especially with highly dynamic workflows. Swift/K distributed execution support allows tasks to be submitted and managed on multiple remote sites from a single client or head node. This is accomplished with a distributed Swift service [6]capable of sending tasks to multiple clusters with different scheduling systems and data architectures, and of dynamically allocating and freeing varying sized blocks of nodes based on workflow demand[6]. The Swift service can also execute workflows whose tasks require diverse data management policies[8]. It can manage shared and node-local file-systems, and can transfer data directly from the workflow client to any compute node. Files can be staged in by the client, sent through the diverse transfer services, and staged out upon completion [6].

In many scientific workflows, the demand for resources fluctuates drastically over time, with some time periods being highly compute intensive, while other periods may only re-

*Computation Institute, University of Chicago

quire a very small number of nodes. The files needed to execute certain sections of the workflow may be large and impractical to move , forcing computation to take place on that site. Swift/K provides both flexible data management and multiple-site execution, but when the sites are supercomputers, Swift/K is limited in its scalability as described above. Swift/T is capable of managing supercomputers, but is not capable of handling the multi-site execution or the data management. With computation time being expensive and very limited, Swift/T will waste valuable resources when workflow task parallelism fluctuates widely over time. Hence the motivation of the work described here is to augment Swift/T's scalability with Swift/K's flexible resource and data management scheme. This integration has been performed, and its results and benefits are reported here.

3. ARCHITECTURE

Our architectural goal is a single, unified Swift language that combines the scalability and efficiency of Swift/T with the execution and data management models of Swift/K. By optionally enabling these execution model in Swift/T, the Swift execution model becomes even more flexible and powerful. Workflows that do not have fluctuating demands can remain the same, but may also be performed on multiple sites if desired. Workflows requiring varying amounts of resources can effectively be managed both locally and on remote sites. Data management can be handled either through the Swift service or through providers such as GridFTP or Globus. The figures below illustrate the diversity of the new workflow execution model.

3.1 Dynamic Execution

Currently, the client through which Swift/T communicates with the Swift Service is limited to a single-site. This is one of the most important features for remote execution. This will aid in adding the dynamic execution capabilities of Swift/K to the functionality of Swift/T. This feature will also allow for multi-queue support within the same system. An example of this is show in Fig. 1. One portion of the workflow may require a great deal of memory or it may perform better on GPUs, while the other is indifferent and can run in a general queue. One queue may also have less of a load and may be preferable as it would allow for a job to start sooner and perhaps perform better due to lower strain on the system. Fig. 2 also demonstrates another situation where dynamic execution can be incredibly useful. An simulation may require data located on a read-only file system. Sometimes that data is read-only and cannot be moved or changed and so the simulation must be run on that specific site. That site, however, could be busy otherwise and the analysis phase of the simulation may be better run on a different system that may have a lower system load or it may have nodes better suited to the task, such as in the multi-queue example mentioned above.

3.2 Data Management

The Swift Service is currently capable of handling http://, cs:// (a prefix for files local to the Swift Service), and several others. Globus Online [2] support is one of the most important features to be added as this will allow for the transfer of large data sets between systems if necessary. It also allows for scientists to be able to track the progress of their

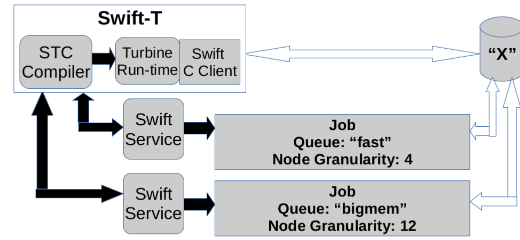

Figure 1: Swift/T multi-queue workflow with static resource demands

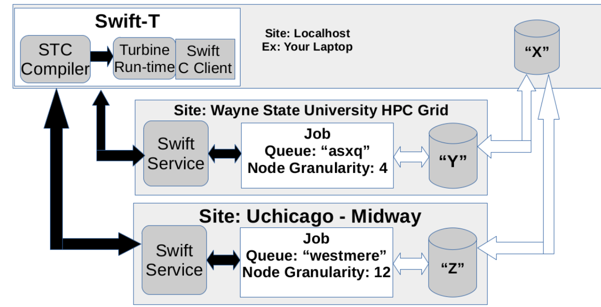

Figure 2: Swift/T multi-site workflow with static resource demands

experiments (via the Globus Online web interface) if Globus Online is used to transfer the resulting data into a remote file system for analysis purposes. This would also allow for real-time data transfer to a separate system for near-real-time analysis of their data. Near-real-time, or in-situ, analysis reduces resource waste not only for HPC systems but also for scientific resources. If preliminary analysis happening during experiment time can illuminate problems in the data, the necessary changes can be made to the experiment and valuable time can be saved. Another important aspect of data management is local staging which allows for the client to get and send files from a local system to the remote Swift service. While not ideal for staging in very large files or datasets, it can be used to stage-out any files resulting from computation, such as the results of the final analysis stages.

3.3 Performance

Perhaps the most important feature of the system is the performance allowed. Swift/T can dispatch tasks much more quickly to the Swift service than can Swift/K. Swift/T can very rapidly generate all of the required jobs for the Swift service. The Swift service then requests the desired blocks of nodes that, upon receipt, can be used to rapidly distribute the Swift tasks among the nodes received. Figs. 4 and 5 demonstrate some of this happening, although they both depict total time, which is composed of initial allocation, task completion, and queue waiting time which tends to dominate the total time for job completion.

Fig. 3 shows a Swift/T workflow that has multiple jobs each with a different amount of required resources. The distribution is handled via the Swift service. In Figs. 1

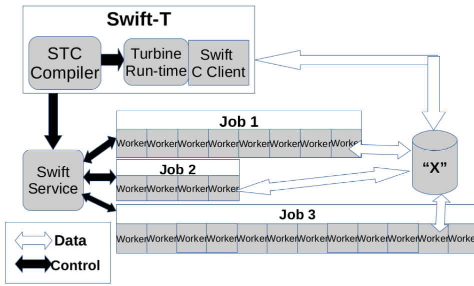

Figure 3: Swift/T single-site workflow with fluctuating resource demands

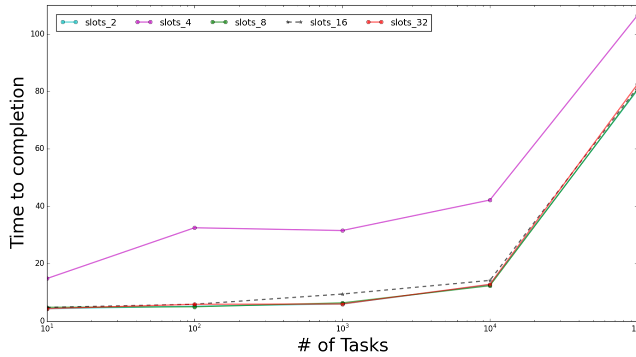

Figure 4: Task throughput via Swift/T and the Swift service with different slot sizes

and 3, "X" indicates a shared filesystem. In Fig. 2, "X" is local to the client, while "Y" and "Z" are shared filesystems within the site. Fig. 4 demonstrates the initial throughput testing of Swift/T with the Swift service running small, 0-second sleep tasks through a Slurm scheduling system on the University of Chicago's Midway system. Fig. 5 is a variation on Fig. 4, using a fixed slot size for larger numbers of tasks instead of multiple slot sizes.

4. RELATED WORK

Scientific workflow management systems such as Makeflow [1], DAGMan [4], and Pegasus [5] also aim to automate remote execution and data management for data-intensive workflows. None of these span the breadth of runtime environment achieved by the work reported here.

5. CONCLUSION

With such diverse target system architectures for high performance workflow, it is challenging to achieve an ideal parallel distribution of both data and computation without wasting resources, whether it be computation time, storage, or network bandwidth. By combining the execution and data management models of Swift/K with the scalability task model of Swift/T into a unified implementation, a more efficient workflow model has been achieved. This unified workflow scripting language with multiple run-time topologies reduces wasted resources while providing an execution model capable of extreme-scale performance.

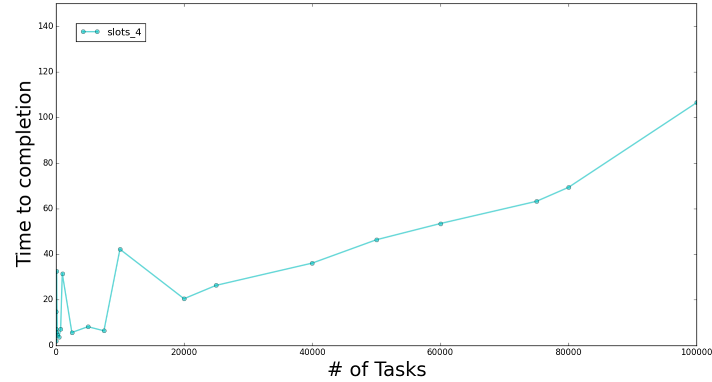

Figure 5: Task throughput via Swift/T and the Swift service with a fixed slot size

6. FUTURE WORK

A variation on Fig. 2 can be extrapolated when the Swift service is instead Swift itself that launches multiple MPI jobs. These MPI jobs work similarly to the Swift service by allowing for diverse resource allocation that can be configured during execution. This would also allow for the scheduling of jobs with varying lifetimes. Changes must be made to the core Swift/T engine and runtime in order to allow for this run-time topology to exist in future versions of the language. Other future tasks include the addition of the multi-site functionality mentioned in the Architecture section. Data management mechanisms also need to be finalized. The Globus Online feature is still not fully supported, although it is in development. Currently, the file stage-in is successful, but any operations that attempt to use the file fail for unknown reasons. Beyond the features directly affected by this work, many other aspects still need to be examined to unify the two languages. Further support for proper file management within Swift/T is one such thing. With this addition, any sort of file staging involved with data transfers would be greatly assisted once the language has more than a basic concept of a file. The majority of other features are outside the scope of this paper.

7. REFERENCES

[1] M. Albrecht, P. Donnelly, P. Bui, and D. Thain. Makeflow: A portable abstraction for data intensive computing on clusters, clouds, and grids. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 1:1–1:13, New York, NY, USA, 2012. ACM.

[2] B. Allen, J. Bresnahan, L. Childers, I. Foster, G. Kandaswamy, R. Kettimuthu, J. Kordas, M. Link, S. Martin, K. Pickett, and S. Tuecke. Globus online: Radical simplification of data movement via saas. 06/2011 2011.

[3] T. G. Armstrong, J. M. Wozniak, M. Wilde, and I. T. Foster. Compiler techniques for massively scalable implicit task parallelism. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 299–310. IEEE Press, 2014.

[4] P. Couvares, T. Kosar, A. Roy, J. Weber, and K. Wenger. Workflow management in condor. In *Workflows for e-Science*, pages 357–375. Springer, 2007.

[5] E. Deelman, K. Vahi, G. Juve, M. Rynge, S. Callaghan, P. J. Maechling, R. Mayani, W. Chen, R. Ferreira da Silva, M. Livny, and K. Wenger. Pegasus, a workflow management system for science automation. *Future Generation Computer Systems*, 46:17–35, 2015.

[6] M. Hategan, J. Wozniak, and K. Maheshwari. Coasters: uniform resource provisioning and access for clouds and grids. In *Utility and Cloud Computing (UCC), 2011 Fourth IEEE International Conference on*, pages 114–121. IEEE, 2011.

[7] E. L. Lusk, S. C. Pieper, R. M. Butler, et al. More scalability, less pain: A simple programming model and its implementation for extreme computing. *SciDAC Review*, 17(1):30–37, 2010.

[8] M. Wilde, M. Hategan, J. M. Wozniak, B. Clifford, D. S. Katz, and I. Foster. Swift: A language for distributed parallel scripting. *Parallel Computing*, 37(9):633–652, 2011.