Proceedings of the First International Workshop on Sustainable Ultrascale Computing Systems (NESUS 2014)
Porto, Portugal

Jesus Carretero, Javier Garcia Blas
Jorge Barbosa, Ricardo Morla
(Editors)

August 27-28, 2014

Exploiting data locality in Swift/T workflows using Hercules

Francisco Rodrigo Duro,† Javier Garcia Blas,∗ Florin Isaila,+ Jesus Carretero*

University Carlos III, Spain † ∗ + *
frodrigo@arcos.inf.uc3m.es†, fjblas@arcos.inf.uc3m.es∗, fisaila@inf.uc3m.es+, jesus.carretero@uc3m.es*

Justin M. Wozniak†, Rob Ross‡

Argonne National Laboratory, USA † ‡
wozniak@mcs.anl.gov†, rross@mcs.anl.gov‡

Abstract

The ever-increasing power of supercomputer systems is both driving and enabling the emergence of new problem-solving methods that require the efficient execution of many concurrent and interacting tasks. Swift/T, as a description language and runtime, offers the dynamic creation and execution of workflows, varying in granularity, on high-component-count platforms. Swift/T takes advantage of the Asynchronous Dynamic Load Balancing (ADLB) library to dynamically distribute the tasks among the nodes. These tasks may share data using a parallel file system, an approach that could degrade performance as a result of interference with other applications and poor exploitation of data locality. The objective of this work is to expose and exploit data locality in Swift/T through Hercules, a distributed in-memory store based on Memcached, and to explore tradeoffs between data locality and load balance in distributed workflow executions. In this paper we present our approach to enable locality-based optimizations in Swift/T by guiding ADLB to schedule computation jobs in the nodes containing the required data. We also analyze the interaction between locality and load balance: our initial measurements based on various raw file access patterns show promising results. Moreover, we present future work based on the promising results achieved so far.

Keywords Locality, In-memory storage, Swift/T, workflows

I. Introduction

Storage systems represent one of the main bottlenecks in modern high-performance systems and are expected to pose a significant challenge when building the next-generation *exascale* systems [2]. Large-scale storage systems are likely to be hierarchical [4], a configuration that will probably be achieved by exploiting data locality and asynchronously moving data among hierarchy levels [7].

In order to extract maximum performance from the new hardware, exascale systems will require new problem-solving approaches. One of the most promising candidate approaches is the many-task paradigm relying on a workflow model. In this paper we propose a solution using Swift/T, a programming model and runtime developed at Argonne National Laboratory that simplifies the development and deployment of many-task applications on large-scale systems. To expose and exploit data locality in Swift/T, we use Hercules, a distributed in-memory store based on Memcached. Hercules offers Swift/T workers a shared storage in which I/O nodes can be dynamically deployed for increasing data locality, while scaling better than traditional shared file systems. Additionally, the performance can be isolated from the shared file-system load peaks, a feature that will be especially important on exascale systems where several applications can be running concurrently.

II. Swift/T

Swift [11] is a programming model and runtime engine that permits users to easily express the logic of many-task applications by using a high-level language called Swift. The latest Swift implementation, called Swift/T [12], can quickly launch tasks in any of the available workers using the dataflow model Turbine.This model can be deployed and distributed and can generate tasks with the throughput required by next-generation exascale systems [13].

As seen in Figure 1, Swift/T comprises three main components: the Swift compiler, the Turbine engines, and the Asynchoronous Dynamic Loud Balancing (ADLB) module [8]. The first step consists in converting the Swift code into Turbine code. The Swift language is a scripting language that can easily used to describe parallel algorithms. A Swift program specifies different leaf tasks with their input and output clearly characterized. These tasks can be written in the Swift language or can be independent programs written in any other language, treated by Swift as black boxes.

The second step is the identification of dependencies in the Turbine code. Independent tasks can be run in parallel, while data-dependent tasks will be held by the Turbine engines until every dependency is fulfilled. When a task is ready to run, it is dispatched to the ADLB load-balancing module. The Turbine engines can run on any number of nodes for additional load balancing.

In the third step, the ADLB module schedules the tasks to be

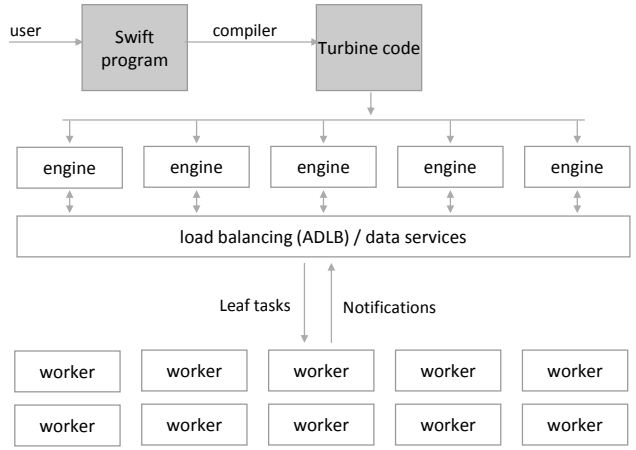

Figure 1: Architecture of the Swift/T runtime. Swift code is compiled into the Turbine code evaluated by the engines; workers execute leaf task applications.

launched on the available workers. When a task finishes, ADLB collects the results and notifies any Turbine engines subscribed to the outputs from that task. Upon notification, the Turbine engines update the data dependencies and release any remaining tasks that are ready to run.

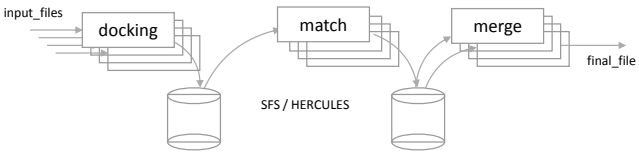

Figure 2: Scheme of a typical many-task workflow application: a protein-docking simulation. The output of one task is the input of the next task.

For a better understanding of the applications representative for Swift/T, Figure 2 shows an example of a workflow application. The application is a typical protein-docking workflow, in which a massive number of files describing protein characteristics are to be processed in order to evaluate how each combination of files docks. For this purpose, three different modules are applied consecutively, represented in the figure as boxes.

The *docking* module, written in C, evaluates two protein files and generates a temporary output file containing their combination. The *match* module, written in Java, takes the output of the previous file and determines whether the combination is correct, classifying it with a label and writing it to an output file. The *merge* application, written in Swift, uses all the files generated by the *match* application and merges them in a single file counting the number of labels of each kind. This file is the final output of the application. A snippet of the source code sample of this application is shown in Figure 3.

A typical protein-docking scenario involves thousands of different proteins producing millions of possible combinations. Manually launching these combinations is tedious work, and classical scripting languages do not have the capability to run independent

```
1  import string;
2  import files;
3
4  app (file f_output) dock (file f_in1, file f_in2)
5  {
6     "./docking" f_in1 f_in2 f_output;
7  }
8
9  app (file f_output) match (file f_input)
10 {
11    "java match" f_input f_output;
12 }
13
14 (file f_results) merge(file f_input[])
15 {
16    foreach f in f_input {
17       // Merge algorithm
18    }
19 }
20
21 main
22 {
23    file fin[];
24    file f_match[];
25    foreach fin1,i in fin{
26       foreach fin2,j in fin{
27          file f_tmp = dock(fin1,fin2);
28          index = i+j + j;
29          f_match[index] = match(f_tmp);
30       }
31    }
32    file f_out<"results.out"> = merge(f_match);
33 }
```

Figure 3: Example of Swift source code.

tasks in parallel. With the simple code from Figure 3, Swift/T evaluates the dependencies and runs millions of instances of existing programs without requiring any change in the source code—not even a recompilation if the code was previously compiled for the machine on which it is supposed to run. File reads and writes are made through a file system shared by every Swift/T worker node.

Currently, Swift/T uses distributed memory to store basic variables; but it requires a shared file system to store files, relying on the distributed memory mechanism only for solving file dependencies using file paths. As shown in Figure 3, this mechanism is used even when the files are going to be part of a workflow: the output of a task generates a temporary file used as the input of the next task. Hercules can alleviate this I/O bottleneck by storing the files in the main memory of the worker nodes. In addition to this problem, the original Swift/T scheduler was not able to exploit data locations, which prohibited the use of systems like Hercules. As shown in Section IV, the ADLB scheduler, driven by Swift, can be guided to exploit data locality and access Hercules files without network communication, colocating the computation in the worker that contains the data and obtaining performance improvements.

III. Hercules

The distributed memory space of Hercules [5] can be used by applications as a virtual storage device for I/O operations. We have adapted this space for use as in-memory shared storage for the Swift/T workers. Our approach relies on an improved version of the Memcached [6] servers, which provide a storage solution for the worker.

As can be seen in Figure 4, our solution consists of two levels: client library and servers. On top is the client user-level library with a layered design. Back-ends are based on the Memcached server, extending its functionality with persistence and tweaks. Main ad-

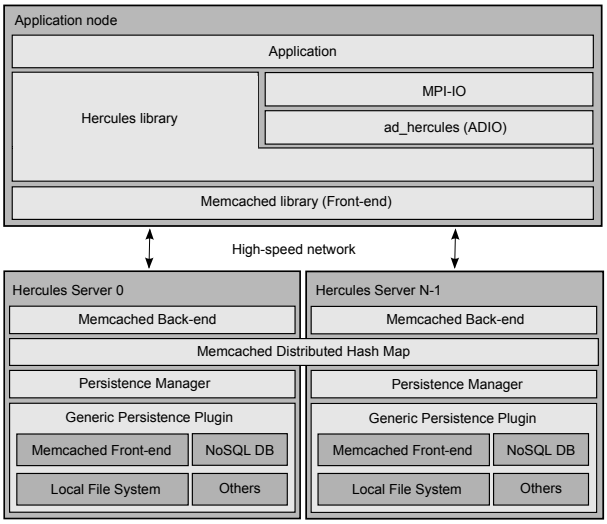

Figure 4: Hercules architecture. On the top is the client side, a user-level library. On the bottom is the server side with the Hercules I/O nodes divided in modules.

vantages offered by Hercules are scalability, ease of deployment, flexibility, and performance.

Scalability is achieved by fully distributing data and metadata information among all the nodes, avoiding the bottlenecks produced by centralized metadata servers, in cases of high metadata accesses load. Data and metadata placement is calculated on the client side by an algorithmic hashing. The servers, on the other hand, are completely stateless.

Ease of deployment and flexibility are tackled at the client side by using a POSIX-like user-level interface (open, read, write, close, etc.) in addition to the classic put/get approach in current NoSQL databases. Existing software requires minimal changes to run with Hercules. Servers can be deployed in any kind of Linux system at the user level, and persistence can be easily configured by using existing plugins or developing new ones.

Performance is achieved by exploiting the parallel I/O capabilities of Memcached servers. Hercules is easy to deploy on as many nodes as necessary, and every node can be accessed independently, multiplying the total throughput peak performance. Furthermore, each node can serve requests in a concurrent way thanks to a multithreading approach. The combination of these two factors results in full scalability, in both the number of nodes and the number of workers running on each node.

In addition to the default algorithmic hashing provided by Memcached, we have designed two new custom placement algorithms. The first algorithm is a locality-aware placement, capable of placing all the items related with one specific file in the same node. Thus, if a task needs to access an existing file and is running on the same node as the Hercules server containing the data, it can access the whole file locally. Combining data locality and concurrent request serving, our solution can achieve intranode scalability, serving requests in parallel from different workers running on the same node—a common approach in current and future multicore compute nodes—and avoiding the usual single network interface bottleneck. The second algorithm has been designed but is not yet fully developed; its objective is to take into account load factors (capacity, CPU load, burst peaks) when selecting the data placement.

IV. Integrating Swift/T and Hercules

The objective of this work is to combine the Swift/T many-task runtime with our Hercules storage system in order to perform I/O operations in-memory instead of using default systemwide shared file systems. The integration of Hercules and Swift/T takes advantage of two features offered by each solution: (1) Hercules offers an ad hoc distributed storage shared among all available workers, using their main memory for storing data; and (2) Swift/T has an experimental feature, called *@location* [14], that can be used to override the default scheduling, placing a specific task on any desired worker node.

To integrate both features, we have developed a mechanism that spawns one Hercules server on each of the worker nodes available for Swift/T. We have implemented a function to easily determine where a specific file is located or where it will be located if it has not yet been written to expose data locality. We have used the *@location* experimental feature to schedule read operations in nodes containing the required data and write operations in the nodes that are going to contain the data to exploit data locality. The combination of these three techniques enables users to perform any kind of read/write file operation querying the Hercules server running in the same node, without needing network communication or disk operations.

Our solution also can be used as an ad hoc distributed in-memory storage, resulting in easier deployment and better scalability than conventional shared file systems provide. Furthermore, our ad hoc storage can avoid the peak load performance penalties that occur from sharing storage resources between different applications running on the same system, thus reducing the shared file system noise in I/O operations.

V. Evaluation

To evaluate our integration of Swift/T and Hercules, we ran a series of tests on the Fusion cluster at Argonne National Laboratory. This cluster has 320 nodes, composed of dual-socket boards with quad-core 2.53 GHz processors and 36 GB of main memory. The intercommunication networks are InfiniBand QDR (4 GB/s) and Gigabit Ethernet.

For the evaluation, we developed a synthetic application with 64 tasks consisting of 32 tasks performing a file write of 2 GB each and 32 tasks performing reads of the previously created 2 GB files. We compared raw file reads/writes with GPFS, Hercules without locality, and Hercules using data locality in all the tasks (all the I/O operations were done inside the node, without using the network). Hercules used the InfiniBand network over TCP/IP for I/O; the GPFS file system has a peak performance of 3200 MB/s over the InfiniBand network. We focused on two cases: scalability in the number of nodes (launching one worker per node) and scalability in the number of workers per node (with a fixed node setup).

As can be seen in Figure 5 and Figure 6, our solution scales better than GPFS, especially when contention is high. In other words,

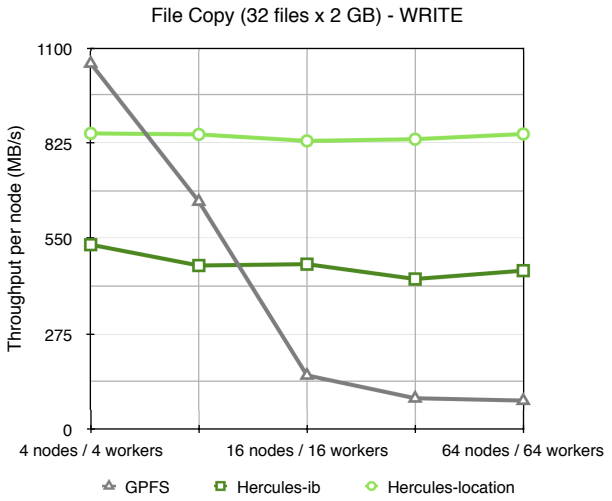

Figure 5: Raw file write throughput per node comparison between our proposed solution and GPFS, when scaling the number of nodes, running one worker per node.

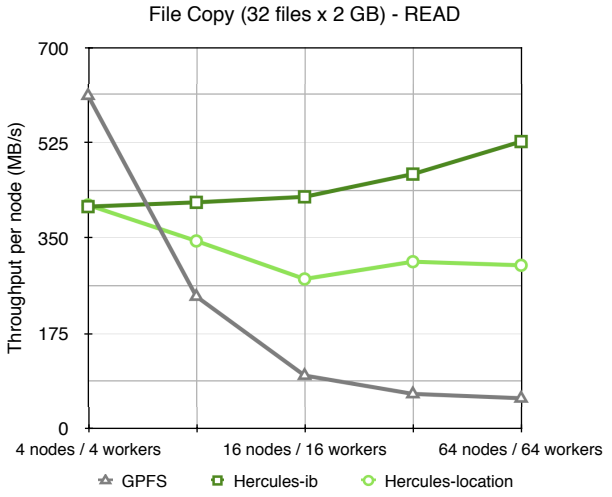

Figure 6: Raw file read throughput per node comparison between our proposed solution and GPFS, when scaling the number of nodes, running one worker per node.

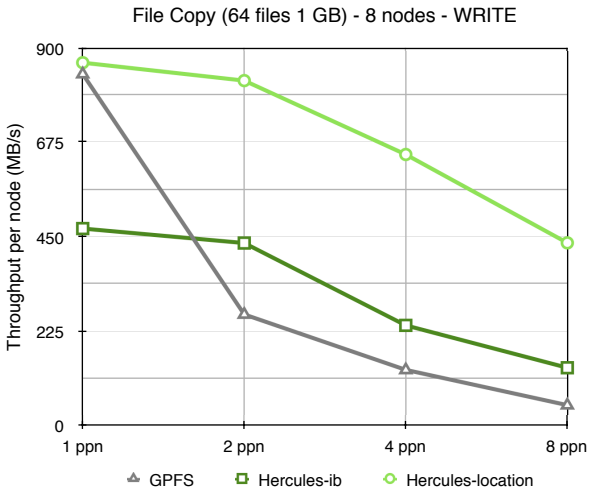

Figure 7: Raw file write throughput per node comparison between our proposed solution and GPFS scaling the number of workers per node running in a fixed 8-node setup.

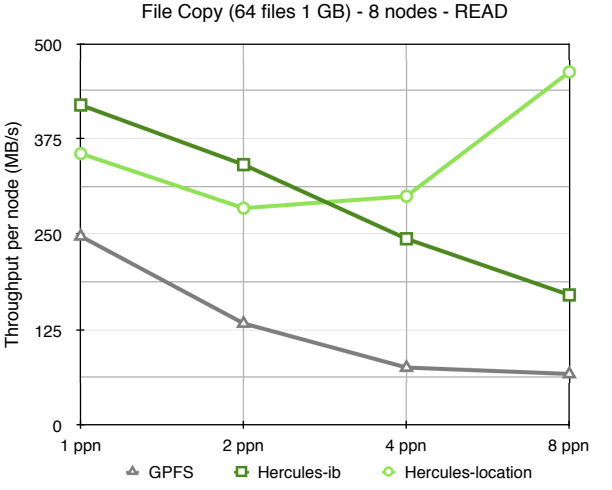

Figure 8: Raw file read throughput per node comparison between our proposed solution and GPFS scaling the number of workers per node running in a fixed 8-node setup.

when many workers are trying to access the file system concurrently, Hercules takes advantage of the increased number of I/O nodes launched and colocated with each worker to perform I/O accesses in a parallel way, whereas GPFS shares its maximum bandwidth between all the nodes. The throughput represented in both figures corresponds with the throughput per node, resulting in an aggregated throughput that scales with the number of worker nodes. Hercules performs similarly over the network and locally, probably because of the overhead of the TCP/IP stack even for local accesses.

Figure 7 and Figure 8 show how when we increased the number of workers per node, fixing the number of nodes to 8, the results are similar to the previous case: our solution scales better with the number of workers, whereas GPFS performance is affected by contention. Again, we note that the represented throughput is measured per worker, which explains the performance hit of GPFS, which has to share the available bandwidth among all the workers.

In this case, the locality-aware version performs better, because it can serve more workers in parallel without contention. That behavior is explained by the ability of the Hercules I/O nodes to serve various workers in parallel thanks to the multithreaded implementation. When the queries are done inside the same node, there is

no need for sharing the network interface, resulting in an improved performance over remote queries using the InfiniBand network interface over TCP. Requests can be served by the multithreaded Hercules I/O nodes using the loopback TCP stack, avoiding sharing the available bandwidth.

These tests demonstrate how our proposed solution scales better than current state-of-the-art parallel shared file systems for I/O-bound applications. We have also demonstrated how our solution suffers less from contention and offers a more stable performance when different applications share the same parallel file system. In contrast, however, real-life applications usually mix computation and I/O operations. This behavior results in less contention to the shared file system and should be evaluated in the future. Another issue that was exposed when evaluating our solution is related to the overridden scheduling. Currently the *@location* functionality allows only one specific node to be selected; but since the load is not balanced among workers running on the same node, a load imbalance can result, reducing the performance gains produced by the improved throughput.

VI. Related Work

The increasing popularity of many-task computing and workflow engines and the I/O bottleneck in such scenarios have led several researchers to investigate this problem.

Parrot [9] is a tool for attaching existing programs to remote I/O systems through the POSIX file-system interface, and Chirp [10] is a user-level file system for collaboration across distributed systems such as clusters, clouds, and grids. They are usually combined to easily deploy a distributed file system ready to use with current applications through a POSIX API. Many characteristics are shared with Hercules: user-level deployment without any special privileges, transparency through the use of a widely used interface, and easy deployment using a simple command to start a new server. Hercules, however, is designed to achieve high scalability and performance by taking advantage of as many compute nodes as possible for I/O operations. Moreover, Hercules uses main memory for storage improving performance in data-locality-aware accesses.

Costa et al. proposed using extended file attributes in MosaStore [1, 3] to provide communication between the workflow engine and the file system through the use of hints about the data. The workflow engine can provide these hints directly to the file system, or the file system can infer the patterns by analyzing the data accesses. The MosaStore approach is radically different from Hercules, using a centralized metadata server instead of the fully distributed, easy-to-use, and flexible deployment approach of our proposed solution.

The AMFS shell [15] offers programmers a simple scripting language for running parallel scripting applications in-memory on large-scale computers. The objective of this solution is similar to the combination of Swift/T and Hercules, but Swift/T can automatically solve data dependencies and launch tasks to workers in a more efficient way by using distributed Turbine engines. AMFS and Hercules also share the distributed metadata approach; the main difference is that AMFS shell programs can explicitly specify in-memory or persistent storage, whereas Hercules can be deployed with persistence enabled in a transparent way for the programmer.

HyCache+ [16] is a distributed storage middleware that allows I/O to effectively leverage the high bisection bandwidth of the high-speed interconnect of massively parallel high-end computing systems. HyCache+ acts as the primary place for holding hot data for the applications (e.g., metadata, intermediate results for large-scale data analysis) and only asynchronously swaps cold data on the remote parallel file system. Similarities between HyCache+ and Hercules include their fully distributed metadata approach, use of compute network instead of the shared storage network, and the high scalability capabilities. HyCache+ relies on POSIX, however, whereas Hercules offers the possibility of using a POSIX-like interface and get/set operations. Moreover, HyCache+ focuses on enhancing parallel file systems in a generic way, whereas Hercules has been designed to work specifically with a many-task engine, exposing and exploiting data locality in current applications. HyCache+ and Hercules thus share similar ideas, but Hercules is ready to improve many-task I/O performance by focusing on easy and flexible deployment options.

VII. Conclusions

In this paper we have presented the integration of Swift/T and Hercules in order to expose and exploit data locality in many-task workflows. We have evaluated the capabilities of our solution for raw file access. The approach achieves a substantial improvement of throughput performance over that of the GPFS file system. In addition, our solution can deploy as many I/O nodes as Swift/T workers running the application, achieving better scalability than possible with traditional static parallel file systems. Another advantage of our solution is isolation from shared file system noise. In the increasingly common case of various applications running at the same time on the same system, our solution ensures isolation of the I/O performance, independent of the file system load at any specific instant.

To tackle the load imbalance issue, we are working on two new approaches that can be combined. The first one will try to improve the load balance of workers inside the same node. An improved scheduler has been implemented in Swift/T, and we are evaluating it with Hercules. The second approach focuses on load balance among nodes. We are developing a new placement policy to map data in a load-aware way, placing data in the less-loaded nodes or in the nodes with more memory/capacity available. Moreover, to better demonstrate the capabilities of our solution, we will evaluate it with CCTW, a real MapReduce-like application.

Acknowledgment

This material is based upon work supported by the U.S. Department of Energy, Office of Science, under contract DE-AC02-06CH11357. Computing resources were provided by the Argonne Leadership Computing Facility. The work presented in this paper was supported by the COST Action IC1305, "Network for Sustainable Ultrascale Computing (NESUS)." The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement number 328582.

References

[1] Samer Al-Kiswany, Abdullah Gharaibeh, and Matei Ripeanu. The case for a versatile storage system. *Operating Systems Review*, 44(1):10–14, 2010.

[2] G. Bell, J. Gray, and A. Szalay. Petascale computational systems. *Computer*, 39(1):110 – 112, jan. 2006.

[3] L.B. Costa, H. Yang, E. Vairavanathan, A. Barros, K. Maheshwari, G. Fedak, D. Katz, M. Wilde, M. Ripeanu, and S. Al-Kiswany. The case for workflow-aware storage: An opportunity study. *Journal of Grid Computing*, pages 1–19, 2014.

[4] Jack Dongarra, Pete Beckman, Terry Moore, and Aerts. The International Exascale Software Project roadmap. *Int. J. High Perform. Comput. Appl.*, 25(1):3–60, February 2011.

[5] Francisco Rodrigo Duro, Javier Garcia Blas, and Jesus Carretero. A hierarchical parallel storage system based on distributed memory for large scale systems. In *Proceedings of the 20th European MPI Users' Group Meeting*, EuroMPI '13, pages 139–140, New York, NY, USA, 2013. ACM.

[6] Brad Fitzpatrick. Distributed caching with memcached. *Linux J.*, 2004(124):5–, August 2004.

[7] Florin Isaila, Javier Garcia Blas, Jesus Carretero, Robert Latham, and Robert Ross. Design and evaluation of multiple-level data staging for blue gene systems. *IEEE Transactions on Parallel and Distributed Systems*, 22(6):946–959, 2011.

[8] E. L. Lusk, S. C. Pieper, R. M. Butler, and Middle Tennessee State Univ. More scalability, less painâĂť: A simple programming model and its implementation for extreme computing. *SciDAC Rev*, 17, Jan 2010.

[9] Douglas Thain and Miron Livny. Parrot: Transparent user-level middleware for data-intensive computing. *Scalable Computing: Practice and Experience*, 6(3), 2005.

[10] Douglas Thain, Christopher Moretti, and Jeffrey Hemmes. Chirp: a practical global filesystem for cluster and grid computing. *Journal of Grid Computing*, 7(1):51–72, 2009.

[11] Michael Wilde, Mihael Hategan, Justin M. Wozniak, Ben Clifford, Daniel S. Katz, and Ian Foster. Swift: A language for distributed parallel scripting. *Parallel Comput.*, 37(9):633–652, September 2011.

[12] J.M. Wozniak, T.G. Armstrong, M. Wilde, D.S. Katz, E. Lusk, and IT. Foster. Swift/T: Large-scale application composition via distributed-memory dataflow processing. In *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*, pages 95–102, May 2013.

[13] Justin M. Wozniak, Timothy G. Armstrong, Ketan Maheshwari, Ewing L. Lusk, Daniel S. Katz, Michael Wilde, and Ian T. Foster. Turbine: A distributed-memory dataflow engine for extreme-scale many-task applications. In *Proceedings of the 1st ACM SIGMOD Workshop on Scalable Workflow Execution Engines and Technologies*, SWEET '12, pages 5:1–5:12, New York, NY, USA, 2012. ACM.

[14] Justin M. Wozniak, Michael Wilde, and Ian T. Foster. Language features for scalable distributed-memory dataflow computing. In *Proc. Data-Flow Execution Models for Extreme-Scale Computing at PACT*, 2014.

[15] Zhao Zhang, Daniel S. Katz, Timothy G. Armstrong, Justin M. Wozniak, and Ian Foster. Parallelizing the execution of sequential scripts. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '13, pages 31:1–31:12, New York, NY, USA, 2013. ACM.

[16] Dongfang Zhao, Kan Qiao, and Ioan Raicu. Hycache+: Towards scalable high-performance caching middleware for parallel file systems. In *IEEE/ACM CCGrid*, 2014.