Scalable In-Memory Computing

Alexandru Uta, Andreea Sandu, Stefania Costache, Thilo Kielmann
Dept. of Computer Science, VU University Amsterdam, The Netherlands
a.uta@vu.nl, a.sandu@vu.nl, s.v.costache@vu.nl, thilo.kielmann@vu.nl

***Abstract*—Data-intensive scientific workflows are composed of many tasks that exhibit data precedence constraints leading to communication schemes expressed by means of intermediate files. In such scenarios, the storage layer is often a bottleneck, limiting overall application scalability, due to large volumes of data being generated during runtime at high I/O rates. To alleviate the storage pressure, applications take advantage of in-memory runtime distributed file systems that act as a fast, distributed cache, which greatly enhances I/O performance.**

In this paper, we present scalability results for MemFS, a distributed in-memory runtime file system. MemFS takes an opposite approach to data locality, by scattering all data among the nodes, leading to well balanced storage and network traffic, and thus making the system both highly performant and scalable. Our results show that MemFS is platform independent, performing equally well on both private clusters and commercial clouds. On such platforms, running on up to 1024 cores, MemFS shows excellent *horizontal* scalability (using more nodes), while the *vertical* scalability (using more cores per node) is only limited by the network bandwith.

Furthermore, for this challenge we show how MemFS is able to scale *elastically*, at runtime, based on the application storage demands. In our experiments, we have successfully used up to 1TB memory when running a large instance of the Montage workflow.

I. Introduction

Nowadays, many scientific computations (e.g. bioinformatics, astronomy applications) are expressed as *scientific workflows*, or, to a broader extent, as *many-task computing* (MTC) applications [1]. Such computations often exhibit inter-task *file dependencies*, leading to communication by means of files as opposed to the traditional message passing mechanisms. Thus, in such scenarios, communication must be achieved through an underlying *distributed/shared file system*. Therefore, the application performance and *scalability* are determined by the storage layer. As data-intensive scientific workflows generate large amounts of data during their runtime, traditional distributed disk-based storage systems are unable to cope with their I/O rates, limiting the overall application performance and *scalability*.

To alleviate the storage pressure, state-of-the art proposes in-memory distributed file systems that store *runtime* generated data [2], [3]. Although memory is still seen as a scarce resource, the DRAM capacity of physical machines is increasing at a fast pace. For example, Amazon allows users to rent compute instances optimized for in-memory computing, with a RAM capacity of up to 244 GB. A common trend for implementing in-memory storage systems is to co-design them with schedulers such that applications can take advantage of data locality. In this way, tasks are scheduled where the data resides and issue only local reads/writes, thus utilizing the high I/O bandwidth of local memory.

However, *this locality-based approach is not entirely suited for scientific workflows, leading to performance and scalability bottlenecks*. First, because usually tasks read *multiple* input files, which may have been generated by different nodes, schedulers have a difficult job in placing tasks where data resides. Thus, only a subset of the input can be read locally, while the complementary subset has to be read remotely, leading to a performance penalty. Second, in the case of *data aggregation* tasks, where individual nodes gather large amounts of data, the locality approach generates large storage imbalances. Considering that memory is still a scarce resource compared to current disk sizes, the data-locality *imbalances* lead to nodes being overwhelmed by the generated data and render them incapable of running subsequent tasks, e.g., due to swapping and system thrashing. Moreover, workflows might also employ *data partitioning* stages, with many nodes reading data from a single source node. In this case, the source node becomes a bottleneck as the the many-to-one communication pattern saturates its network bandwidth.

Finally, current in-memory distributed file systems *are statically deployed onto a fixed number of nodes, lacking the flexibility of scaling elastically during the application runtime*. Although, elasticity is provided by in-memory caching solutions [4], [5], these solutions lack support for automatic scaling and load balancing. Such a capability, as introduced by MemFS, has two important advantages. **First**, the user does not need to estimate the size of data generated at runtime, which is a difficult task at times. In a static deployment, if the user is overestimating the application demands, resource utilization is poor. Conversely, in case of underestimation, the application would be unable to finish its execution or it will have high performance degradation. **Second**, when running scientific applications, users usually want to make trade-offs between execution speed and cost, and schedulers provide mechanisms that dynamically scale the application's resource demand to meet such objectives [6]. Integrating an elastic in-memory file-system with the scheduler leads to more accurate and flexible schedules. Even though elastic mechanisms exist for production file systems such as HDFS [7] or Ceph [8], the process is not automated and thus its applicability is limited.

To overcome these limitations, we introduce MemFS [9], [10], a highly *scalable,* in-memory runtime file system with symmetrical data distribution. The novelty of MemFS is given by its locality-agnostic design, which uniformly distributes data across all storage nodes (by means of file striping) and

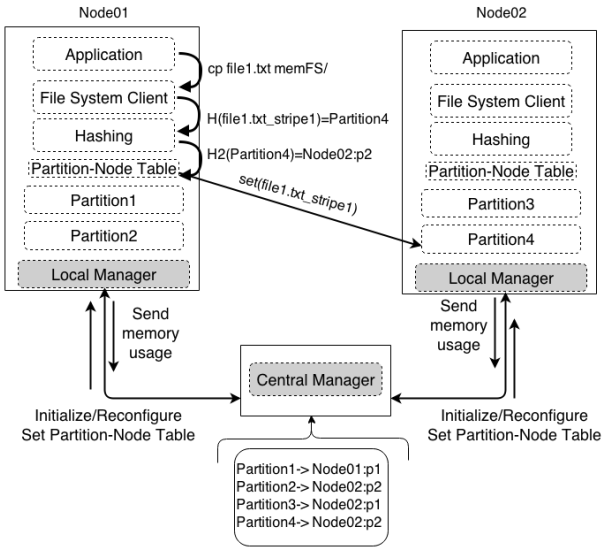


Fig. 1. Architecture of MemFS.

uses consistent hashing to access the files. Thus, the limitations of the data locality approach for scientific workflows are solved: (1) tasks are guaranteed *equal performance* for reading all input files, irrespective of data placement; (2) *data aggregation* stages do not generate storage imbalances as data is uniformly spread over all nodes; (3) *data partitioning* stages do not cause network traffic imbalance as all nodes engage in a many-to-many communication pattern. These features allow MemFS to *scale* to larger problem sizes than a locality-based approach, due to its balanced storage, and to exhibit better *horizontal* and *vertical* scalability, due to a more efficient use of the aggregate network bandwidth. Another important feature of MemFS is its *elastic* scalability, which enables MemFS to *scale out dynamically*, based on application demands while maintaining a uniform data distribution. Furthermore, MemFS is scheduler-agnostic and it can be integrated with a large variety of data processing platforms, resource managers or schedulers. Because it provides a POSIX-based API, MemFS can also be used as a data storage for a variety of applications without changing their code.

In this paper we present a scalability study of MemFS. We prove that MemFS is able to achieve better performance and scalability in running larger problem sizes than a state-of-the-art locality-based storage system. We also show that our approach is platform independent and performs equally well on both clusters and commercial clouds. Finally, as a demo for this challenge, we show that MemFS not only scales well *horizontally* and *vertically*, but it is also able to scale *elastically*, improving the cluster resource utilization.

II. MemFS

Figure 1 gives an overview of MemFS. MemFS consists of worker nodes and a *Central Manager* that gathers worker node statistics and orchestrates *elastic reconfigurations*. Furthermore, the Central Manager applies a *load balancing* scheme for rebalancing the system after reconfigurations, i.e., node additions or removals. The worker nodes run a FUSE *file system client*, which provides a POSIX-like I/O interface to applications, and a *Local Manager*, which gathers resource statistics, e.g., memory utilization, and forwards it to the Central Manager.

To achieve a balanced storage and network traffic, reducing the performance and scalability penalties, MemFS stripes files uniformly across system nodes based on a hash function. To optimize the data movement due to reconfigurations, e.g., nodes are added or removed, MemFS uses a consistent hashing [11] scheme. Consistent hashing guarantees that in a system that stores K objects on N nodes, when a node is added, at most $O(K/N)$ objects need to be rehashed.

MemFS implements consistent hashing through a **2-layer hashing** scheme that maps *file stripes to partitions*, and *partitions to nodes*. In this way, file stripes are not directly mapped to nodes, but rather to partitions. Each node holds multiple partitions, such that, when reconfiguring the file system, only partitions are migrated, thus avoiding rehashing the file stripes. The mapping of stripes to partitions is achieved using the xxhash [12] algorithm. We have chosen this non-cryptographic algorithm because it is optimized for 64-bit CPUs, leveraging up to 13GB/s throughput. This algorithm hashes an input string to a 64 bit number. Applying a modulo scheme, we then decide which partition stores the given file stripe. The mapping of partitions to nodes is kept in a table, called the partition-node table. This table is stored on each worker node and it is updated by the Central Manager at each reconfiguration. Thus, to read or write a file stripe, MemFS first determines the id of the partition responsible for the file stripe, then the node responsible for the partition, and then the query is sent directly to that node, achieving O(1) look-up.

Throughout the application runtime the number of partitions is constant. The total number of partitions sets the upper bound on the number of nodes to which the elastic distributed file system can scale out to. The size of each partition is limited by the node's memory capacity. When running on a small number of nodes with many partitions, the partition size will be small. When the number of nodes is increased, a subset of the partitions will be migrated to the newly added nodes, allowing all the partitions to grow in size.

We implemented the MemFS partitions using Redis [13]. Each partition is represented by one Redis process. Migrating the Redis processes follows a cold migration scheme: we first dump the database to a file; then, this file is copied to the remote node; the database file is reloaded into a new Redis process on the remote node.

A. Load Balancing

To compute the number of partitions each node stores, MemFS adapts the partition distribution algorithm proposed in Y0 [14]. We have chosen this algorithm because it achieves load balance even with heterogeneous nodes. The load imbalance factor of Y0 is at most 3.6, while DHTs usually generate a load imbalance in the order of $O(\log N)$. However,

in this paper we do not present experiments on heterogeneous systems. Therefore, the load imbalance achieved by Y0 on our target platforms is approximately 1 (perfect load balance).

B. Elastic Reconfiguration

For this paper we configured MemFS to use a simple policy to scale dynamically based on its current memory utilization. MemFS *scales out* when its current memory utilization is close to maximum, and *scales in* (reduces the number of nodes) when the memory utilization drops below 50%; in both cases the number of nodes added or removed is a constant defined by the user, e.g., 25% from the current capacity. This policy is used only for demonstrative purpose. MemFS' behaviour can be further extended to consider other metrics, e.g., to minimize resource usage or cost, or to guarantee a certain I/O throughput, and use more advanced scaling policies, e.g., adjusting the number of nodes based on the rate at which memory utilization increases/decreases.

The reconfiguration process is as follows. The *Central Manager* collects utilization metrics from the worker nodes' *Local Managers* and based on them it decides the new number of nodes. Then, the worker nodes' *Local Managers* are notified that a reconfiguration follows and, to avoid invalid requests, they suspend the I/O requests from running application processes. After the I/O is suspended, the Local Managers send back to the Central Manager an acknowledgement message to notify that it is safe to start the reconfiguration. The Central Manager allocates new nodes and re-runs the Y0 algorithm to determine how many partitions need to be migrated among the nodes. The partitions are then migrated and a new *partition-to-node mapping* is computed. Finally, the Central Manager broadcasts the new partition-to-node mapping to all nodes and notifies them that it is now safe to resume I/O requests.

III. Scalability and Elasticity Evaluation

We evaluate the vertical and horizontal scalability of MemFS on our local hardware infrastructure, DAS4 [15], and in a virtualized environment, on the Amazon *Elastic Compute Cloud* (EC2) [16]. By scaling *vertically*, we analyze the system behaviour on a fixed number of nodes, while gradually increasing the number of compute cores used for task processing. Conversely, by scaling *horizontally*, we analyze the system behaviour while gradually increasing the number of compute and storage nodes. We show that on DAS4 MemFS outperforms a state-of-the-art in-memory file system, that uses locality-based data management [17], and that in a virtualized environment its scalability is only limited by the available network bandwith. Then, we show how MemFS scales elastically based on current application data storage demand, using up to 1TB of memory.

A. Evaluated Environment

To evaluate MemFS on a hardware infrastructure, we use the local DAS4 cluster. The 64 compute nodes of DAS4 are equipped with dual-quad-core Intel E5620 2.4 GHz CPUs and 24GB memory. The nodes are connected using a commodity 1Gb/s Ethernet and a premium Quad Data Rate (QDR) InfiniBand providing a theoretical peak bandwidth of 32Gb/s. For our experiments we use the IP over InfiniBand (IPoIB) interface of the latter, which achieves approximately 1GB/s bandwidth.

When running on the Amazon EC2 cloud, we use 32 c3.8xlarge virtual machine (VM) instance types. These instances feature 32 virtual compute cores, 60GB of memory and are connected with 10G Ethernet links. Our *iperf* test shows that the achievable bandwidth between c3.8xlarge EC2 instances is approximately 1GB/s. We chose this specific instance type because its virtualized 10G Ethernet network link achieves similar bandwidth to our DAS4 cluster nodes.

We divided the memory of each node as follows. We reserve 4GB for running the applications and the operating system. The remaining memory is used by the file system for storing the data generated by the applications. In all of the following experiments, the compute nodes, which run the application tasks, also operate as storage nodes for the evaluated MTC file-system.

B. Applications

We use two the well-known scientific workflows, Montage [18] and BLAST [19]. Montage is an astronomy application that, given a set of input images of a galaxy, builds a mosaic. Montage is composed of a series of jobs containing many parallel tasks, i.e., *mProjectPP*, *mDiffFit* and *mBackground*, and serial jobs, which perform data aggregation and partitioning. BLAST is a bioinformatics application that searches for specific nucleotide sequences in a given database and is composed of two jobs containing parallel tasks, *formatdb* and *blastall*. According to [20], [21], the two applications feature different resource utilization. Montage exhibits low memory and CPU utilization, but high I/O. In contrast, BLAST shows high CPU utilization but only medium memory and I/O usage.

C. Vertical and Horizontal Scalability Results

First, we compare the performance and scalability achieved by MemFS to AMFS [17], a state-of-the-art in-memory MTC file system which uses a locality-based approach for data management and task scheduling. For better understanding the results, it is important to note the difference between AMFS' locality-based and our locality-agnostic approach.

AMFS improves the application performance by issuing only local writes and uses the AMFS Shell scheduler for executing compute tasks on those nodes that actually store needed files to improve read performance. AMFS Shell, however, can only guarantee that one file per job achieves data locality. Nevertheless, representative workflows, as the ones from our experiments, read multiple files per task and expensive remote reads become necessary. One could argue that, by using AMFS *collective operations*, all needed files could be made available in advance to all compute nodes. However, this is not always feasible since the output of one workflow stage could be in the order of hundreds of GB, and even higher, and could easily saturate each node's main memory.

The MemFS design guarantees equal performance for any file read operation, independent of actual task placement. Due to the file striping, better performance can be achieved by using the aggregate bandwidth of all nodes storing requested stripes, belonging to one or more files needed by a task. We show next that for the Montage workflow this assumption holds.

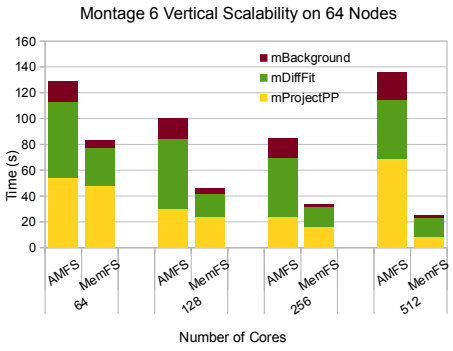
To compare MemFS and AMFS, we chose to run a 6×6 Montage mosaic (Montage 6) centered on the M17 galaxy. It features 2488 input files that sum up to 4.9GB of data. The volume of data generated during runtime is approximately 50GB. We also used a larger mosaic of 16×16 (Montage 16) which has an input size of 34GB and generates at runtime approximately 450GB. However, we show the results only for the first mosaic, as AMFS does not scale to the data amount generated by the second one. For the BLAST application, our evaluation scenario follows the pattern from [21]. However, we use the largest database available online, the *NCBI nt* database (57GB size) which is split offline into several fragments by using the *fastasplit* program. These fragments are copied at runtime into the file system (either AMFS or MemFS). The generated data volume is approximately 200GB.

Figure 2 shows the vertical scalability of MemFS and AMFS. The results were determined on 64 nodes, using gradually 1, 2, 4, and 8 compute cores each. We notice that MemFS shows good scalability *up to 512 compute cores* (8 compute cores per node), while AMFS *only up to 256 compute cores* (4 compute cores per node).

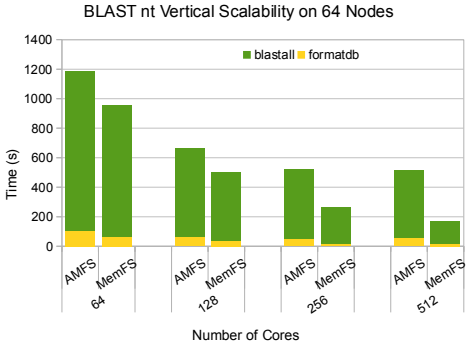
Figure 3 depicts the horizontal scalability of the two file systems. We scaled out the systems from 8 to 64 compute nodes. Because the vertical scalability results (Figure 2a) showed that AMFS could not scale up to 8 compute cores per node in the 64 node scenario, in the horizontal scalability graph we decided to show both the 8 compute core scenario, together with the 4 compute core scenario which achieves best performance on 64 nodes. For all configurations, when benchmarking MemFS we used 8 cores per node. The results show that while both file systems achieve good horizontal scalability, *the performance achieved by MemFS is superior to the AMFS performance*.

This behaviour is given by AMFS' inability to scale up to 8 compute cores per node (Figure 2b). While in the 1 and 2 compute core per node case, MemFS is only about 18% faster than AMFS, in the 4 and 8 compute core per node case, the difference becomes much higher. This performance difference also leads to faster completion times when scaling horizontally from 8 to 64 nodes. An interesting observation is that when increasing the number of nodes, AMFS' performance degrades when more tasks are run per node: in the 8 and 16 node setups, AMFS performed the best when running 8 tasks per node, while in the 32 and 64 node setups, AMFS performed the best with 4 running tasks per node. This can be explained by the fact that in setups with larger number of nodes, AMFS read operations are more expensive, as more data needs to be transfered among nodes.

Further, we assessed MemFS vertical scalability on larger

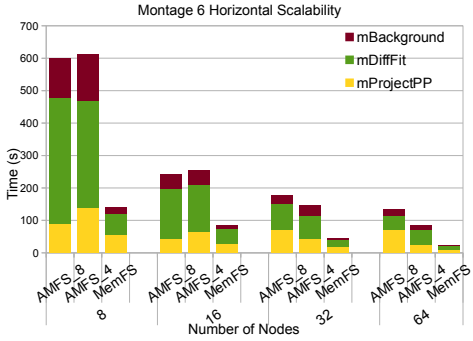


(a) 64 Nodes, scaling up to 512 Cores.

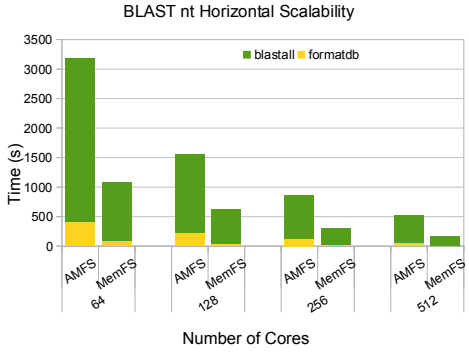


(b) 64 Nodes, scaling up to 512 Cores.

Fig. 2. Vertical scalability of MemFS vs. AMFS for two applications.

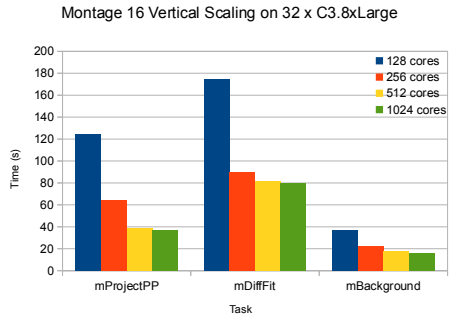


(a) Scaling out from 8 to 64 Nodes.

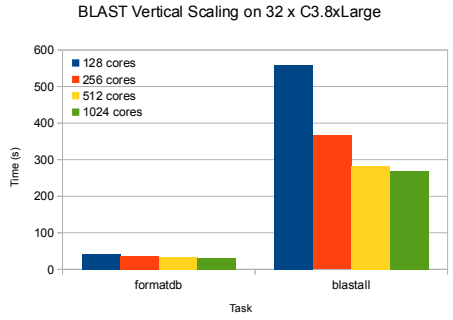


(b) Scaling out from 8 to 64 Nodes.

Fig. 3. Horizontal scalability of MemFS vs. AMFS for two applications.

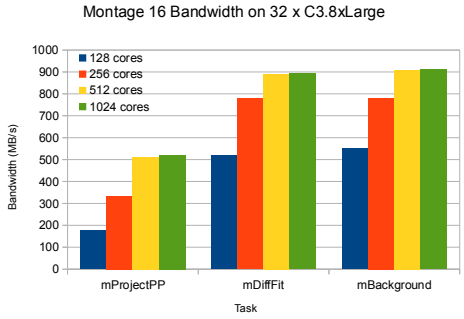


(a) Montage 16 Execution Time

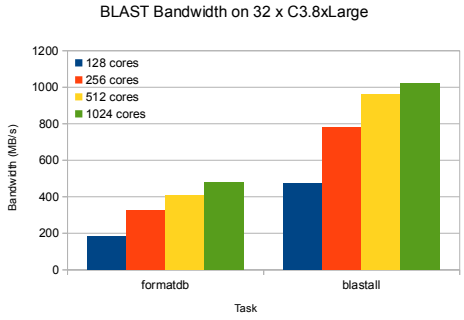


(b) BLAST Execution Time

Fig. 4. Application performance on Amazon EC2.



(a) Montage 16 Achieved Bandwidth per Node



(b) BLAST Achieved Bandwidth per Node

Fig. 5. Used bandwith on Amazon EC2.

problem sizes and when running in a virtualized environment. We ran Montage 16 and BLAST using gradually 4, 8, 16 and 32 virtual cores on each of our 32 c3.82xlarge VMs. Our largest setup uses 1024 virtual compute cores, twice as many cores as used on the DAS4 cluster. Figure 4 shows MemFS scalability, translated in application execution time, on 32 VMs using up to 1024 cores. For Montage, the *mProjectPP* stage is CPU-bound, while *mDiffFit* and *mBackground* are I/O-bound. For BLAST, *formatdb* is CPU-bound and *blastall* is I/O-bound. This is why for *mProjectPP* and *formatdb* MemFS shows better vertical scalability. To investigate the scaling behaviour of the I/O-bound stages of the workflows, we monitored the network activity of the EC2 virtual machines. Figure 5 shows the network bandwidth usage for our two applications per node. Our results show that the I/O-bound stages saturate the network bandwidth (of approximately 1GB/s) when running from 16 to 32 cores per VM. Thus, *the vertical scalability of MemFS is only bound by the network bandwidth.*

D. Elasticity Results

To demonstrate MemFS' ability to scale elastically, based on application demands, we run a 20×20 Montage mosaic that generates a maximum load of 1TB data on our DAS4 cluster. While for the vertical and horizontal scalability experiments we only report the performance achieved by the workflow parallel stages, for this experiment we report the entire application runtime. During the runtime, our scheduler removes data that is not needed, acting like a garbage collector, i.e., after the mProjectPP stage the input is deleted, and after the mBackground stage all the data previously generated. Removing data enables MemFS to *scale in*, i.e. decrease the number of nodes.

For this experiment, we designed two policies to show how MemFS could be used to optimize resource utilization in private clusters. *Policy 1* runs the application starting with 32 nodes. When the aggregate node memory utilization is approximately 90%, MemFS *scales out* by adding 16 nodes. Conversely, when the node memory utilization drops below 50%, MemFS *scales in* by removing 16 nodes. *Policy 2* is more conservative: MemFS starts using 16 nodes, *scaling out* by 16 nodes, while *scaling in* is done in increments of 8 nodes.

Figure 6 shows the difference between the memory allocation and utilization, i.e., the data generated by the application, of our two policies. During a static run, the memory of all 64 cluster nodes is allocated for the entire runtime of the application, to fit the maximum generated data amount. Our results show that elastically scaling MemFS based on the application demand can largely improve the cluster's resource utilization. This comes at the cost of having slightly longer execution times than for a static run, due to reconfiguration overheads and because parallel stages are executed with less worker nodes. Table I shows a more in-depth comparison betwen the static run and our two elastic scaling policies. Compared to the static run, for a performance overhead of 13 to 23%, the elastic scaling policies exhibit 55 to 63% better resource utilization efficiency, i.e., the difference between

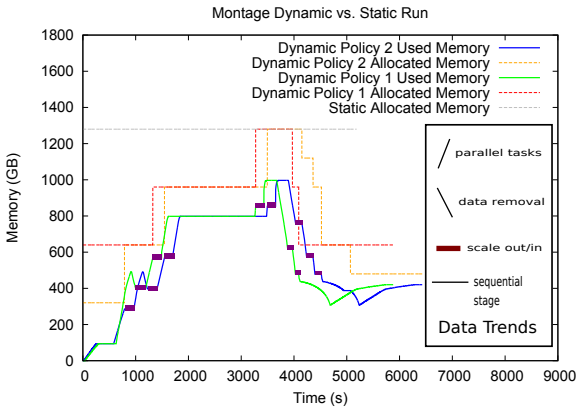


Fig. 6. Elastic Policies Comparison

TABLE I
COMPARISON WITH STATIC MONTAGE RUN

	Resource usage improvement	Performance overhead	Time half of the cluster is free
Policy 1	55.8%	23%	52.9%
Policy 2	63%	13%	53.23%

allocated memory and used memory during runtime. Thus, for more than 50% of the runtime, MemFS does not use 32 nodes (half of our cluster size). This enables other users to run their jobs concurrently with MemFS tasks, leading to shorter queue times and better general cluster resource utilization.

IV. CONCLUSION

Data-intensive scientific computations, like MTC applications, or more specifically, scientific workflows can take advantage of in-memory storage systems to store their runtime generated data, thus improving their performance and scalability. However, current in-memory storage solutions use a locality-based approach, in which the file-system is co-designed with the scheduler, to place the application tasks on nodes on which the data reside. This approach is not suitable for scientific workflows, which might exhibit data aggregation and partitioning stages, and moreover, tasks accesing multiple files simultaneously.

In this paper we introduced MemFS, a highly scalable in-memory distributed runtime file-system. The novelty of MemFS comes from its locality-agnostic and elastic design, which distributes application data among different nodes in a load-balanced way, even when reconfiguring the number of nodes to adapt to a dynamic data storage demand. Our results show that MemFS outperforms locality-based file-systems by providing better application performance and running larger problem sizes. We also show that MemFS performs equally well on different infrastructure types, both clusters and clouds. Finally, MemFS scales elastically, according to the dynamic data storage demands of the application. This opens up the path to future research on integration with application schedulers and in designing algorithms to provide users trade-offs between performance, resource utilization, energy and monetary costs.

ACKNOWLEDGMENTS

This work is partially funded by the Dutch public-private research community COMMIT/. The authors would like to thank Kees Verstoep for providing excellent support on the DAS-4 clusters.

REFERENCES

[1] I. Raicu, "Many-task computing: Bridging the gap between high throughput computing and high performance computing," 2009.
[2] J. Ousterhout, P. Agrawal, D. Erickson, C. Kozyrakis, J. Leverich, D. Mazières, S. Mitra, A. Narayanan, D. Ongaro, G. Parulkar *et al.*, "The case for ramcloud," *Communications of the ACM*, Vol. 54, no. 7, pp. 121–130, 2011.
[3] D. Zhao and I. Raicu, "Hycache: a user-level caching middleware for distributed file systems," *International Workshop on High Performance Data Intsensive Computing, IEEE IPDPS*, Vol. 13, 2013.
[4] "Hazelcast," *http://http://hazelcast.com/*, 2015.
[5] "Amazon ElastiCache," *http://aws.amazon.com/elasticache/*, 2015.
[6] A. Oprescu and T. Kielmann, "Bag-of-tasks scheduling under budget constraints," *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*. IEEE, 2010, pp. 351–359.
[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE, 2010, pp. 1–10.
[8] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," *Proceedings of the 7th symposium on Operating systems design and implementation*. USENIX Association, 2006, pp. 307–320.
[9] A. Uta, A. Sandu, and T. Kielmann, "MemFS: an in-memory runtime file system with symmetrical data distribution," *IEEE Cluster*, 2014, pp. 272–273, (poster paper).
[10] ——, "Overcoming data locality: An in-memory runtime file system with symmetrical data distribution," *Future Generation Computer Systems*, 2015.
[11] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, and D. Lewin, "Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web," *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*. ACM, 1997, pp. 654–663.
[12] "xxhash," https://code.google.com/p/xxhash/, 2014.
[13] S. Sanfilippo and P. Noordhuis, "Redis," http://redis.io, 2014.
[14] P. B. Godfrey and I. Stoica, "Heterogeneity and load balance in distributed hash tables," *INFOCOM 2005. 24th Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings IEEE*, Vol. 1. IEEE, 2005, pp. 596–606.
[15] "DAS-4, The Distributed ASCI Supercomputer," http://www.cs.vu.nl/das4/, 2014.
[16] "Amazon EC2," http://aws.amazon.com/ec2/, 2014.
[17] Z. Zhang, D. S. Katz, T. G. Armstrong, J. M. Wozniak, and I. Foster, "Parallelizing the execution of sequential scripts," *High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for*. IEEE, 2013.
[18] J. C. Jacob, D. S. Katz, G. B. Berriman, J. C. Good, A. Laity, E. Deelman, C. Kesselman, G. Singh, M.-H. Su, T. Prince *et al.*, "Montage: a grid portal and software toolkit for science-grade astronomical image mosaicking," *International Journal of Computational Science and Engineering*, Vol. 4, no. 2, pp. 73–87, 2009.
[19] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, and D. J. Lipman, "Basic local alignment search tool," *Journal of molecular biology*, Vol. 215, no. 3, pp. 403–410, 1990.
[20] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, and P. Maechling, "Scientific workflow applications on amazon ec2," *E-Science Workshops, 2009 5th IEEE International Conference on*. IEEE, 2009, pp. 59–66.
[21] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, and I. Foster, "Design and analysis of data management in scalable parallel scripting," *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press, 2012, p. 85.