Overcoming Data Locality: an In-Memory Runtime File System with Symmetrical Data Distribution

Alexandru Uta[a,*], Andreea Sandu[a], Thilo Kielmann[a]

[a]*Dept. of Computer Science, VU University Amsterdam, The Netherlands*

Abstract

In many-task computing (MTC), applications such as scientific workflows or parameter sweeps communicate via intermediate files; application performance strongly depends on the file system in use. The state of the art uses runtime systems providing in-memory file storage that is designed for data locality: files are placed on those nodes that write or read them. With data locality, however, task distribution conflicts with data distribution, leading to application slowdown, and worse, to prohibitive storage imbalance. To overcome these limitations, we present MemFS, a fully symmetrical, in-memory runtime file system that stripes files across all compute nodes, based on a distributed hash function. Our cluster experiments with Montage and BLAST workflows, using up to 512 cores, show that MemFS has both better performance and better scalability than the state-of-the-art, locality-based file system, AMFS. Furthermore, our evaluation on a public commercial cloud validates our cluster results. On this platform MemFS shows excellent scalability up to 1024 cores and is able to saturate the 10G Ethernet bandwidth when running BLAST and Montage.

Keywords: many-task computing, in-memory file system, distributed hashing, scalability

1. Introduction

Many scientific computations can be expressed as Many-Task Computing (MTC) applications. In such scenarios, application processes communicate by means of intermediate files. The performance of such applications depends, among other factors, on the speed of the underlying file systems.

In general, many-task computing applications use three types of data: input, temporary data generated during job execution (stored in a runtime file system), and output. In data-intensive scenarios, the temporary data is generally much larger than input and output. In a 6x6 degree Montage mosaic [1], for example, the input, output and intermediate data sizes are 3.2GB, 10.9GB and 45.5GB, respectively [2]. Thus, speeding up I/O access to temporary data is key to achieving good overall performance.

*Corresponding autor

Email addresses: a.uta@vu.nl (Alexandru Uta), a.sandu@vu.nl (Andreea Sandu), thilo.kielmann@vu.nl (Thilo Kielmann)

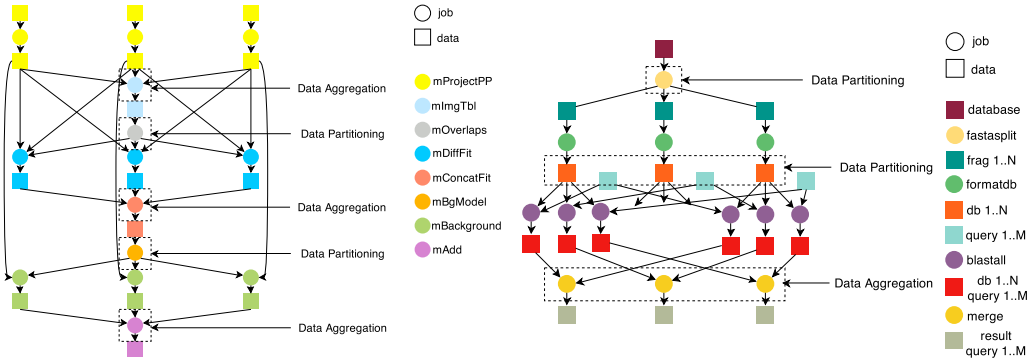
General-purpose, distributed or parallel file systems such as NFS, GPFS [3], or PVFS [4] provide less than desirable performance for temporary data for two reasons. First, they are typically backed by physical disks or SSDs, limiting the achievable bandwidth and latency of the file system. Second, they provide POSIX semantics which are both too costly and unnecessarily strict for temporary data of MTC applications that are written once and read several times. Tailoring a runtime file system to this pattern can lead to significant performance improvements.

The current state of the art suggests to use memory-based runtime file systems as they remove all overheads of mechanical disks or SSDs. For MTC applications, such file systems are co-designed with task schedulers, aiming at data locality [2]. Here, task scheduling places tasks onto nodes that contain the required input files, while write operations go to the node's own memory. Analyzing the communication patterns of the Montage [1] and BLAST [5] workflows, however, shows that temporary files often are created by a single task. In subsequent steps, tasks combine several files for their computation, and final results are based on global data aggregation. Implementing for data locality hence leads to two significant drawbacks: (1.) Local-only write operations can lead to significant storage imbalance across nodes, while local-only read operations cause file replication onto all nodes that need them, which in worst case might exceed the memory capacity of nodes performing global data reductions. (2.) Because tasks typically read more than a single input file, locality-aware task placement is difficult to achieve in the first place.

To overcome these drawbacks, we designed a distributed, in-memory runtime file system, called MemFS, that replaces data locality by uniformly spreading file stripes across all storage nodes. Due to its striping mechanism, MemFS leverages full bisection bandwidth of premium networks, maximizing I/O performance. Since remote operations have become less expensive on premium networks, MemFS is able to achieve better performance and (multi-core) scalability than the state-of-the-art, locality-based AMFS [2] file system, while avoiding the storage imbalance drawback. Furthermore, MemFS guarantees similar performance to any scheduler that uniformly distributes tasks to compute nodes. The main contributions of this work are the following:

- We present a new, locality-agnostic approach to speed up MTC applications' I/O access to runtime generated data.
- We show that our approach alleviates the locality-based bottlenecks (storage and network traffic imbalances, scheduling overheads), leading to faster application completion times and better scalability.
- We show that our approach is not limited to tightly-coupled compute clusters, it is also applicable to public commercial clouds with high speed interconnects.
- Our experiments, running real-world scientific applications, show that the locality-agnostic approach is only bound by network bandwidth.

This paper is organized as follows. Section 2 presents background and related work, while in Section 3 the overall system design is described. In Section 4, we evaluate MemFS and discuss achieved performance. Section 5 draws conclusions and indicates directions for future work.



(a) Montage Workflow

(b) BLAST Workflow

Figure 1: Many-Task Computing Applications

2. Background and Related Work

Traditional approaches for running scientific computations that analyze large volumes of data on clusters or supercomputers usually involve preserving data-locality to improve application performance. This is done because remote copying of the data is considered an expensive operation due to the overhead incurred by the network. However, preserving data-locality for MTC applications is a difficult task for both the underlying file system and the scheduler. Locality information needs to be exposed by the file system and the scheduler needs to place computation where the data resides. For application tasks that need multiple input files this is difficult to achieve and replication schemes or collective movement of data need to be implemented.

For example, the widely-known workflow applications Montage and BLAST, shown in Figures 1a and 1b, contain several stages of global data partitioning and aggregation. For global partitioning, a single task is creating input data for many other tasks. When writing locally, this can lead to severe storage imbalance among nodes. As soon as the dependent tasks start reading these files, several copies are created (for reading locally), leading to increased storage (memory) consumption. For global aggregation, a single task has to copy large amounts of data into its own storage, possibly exceeding its own capacity. Aiming at equally spreading data across nodes can help overcoming these limitations.

In addition, network technologies have evolved and current premium networks offer low latency overheads and high bandwidth. Building a network block device attaching remote memory over InfiniBand had already been proposed in [6]. As a current example, the Dutch National Supercomputer, Cartesius [7], is equipped with Mellanox ConnectX-3 InfiniBand adapters that are able to achieve a unidirectional bandwidth of 56Gb/s. This offers a theoretical bi-directional peak bandwidth of 14GB/s. The Stream benchmark [8] reports, on the same supercomputer, a 10GB/s memory bandwidth. Hence, theoretically, two nodes writing simultaneously to each other's memory would saturate both network and memory bandwidth. In such an environment, a file system that efficiently exploits available network bandwith and uses a uniform data distribution would remove the need of data-locality.

Traditional parallel or distributed file systems such as GPFS [3], PVFS [4], Lustre [9], GlusterFS [10], XtreemFS [11], and Ceph [12] are generally deployed statically on (a subset of) the nodes of a cluster/supercomputer. Such storage systems provide consistency, durability and

fault-tolerance. However, such general-purpose distributed file systems might not be suitable for data-intensive MTC workloads that generate large amounts of temporary data for the following reasons. First, they provide strict POSIX semantics that might add overhead while running MTC workloads. To optimize I/O accesses of MTC applications, MemFS relaxes POSIX compliancy offering a "write-once, read-many" semantics, while preserving POSIX interfaces to support legacy applications. In contrast, HDFS [13], while also optimising for the "multi-read, single-write" access pattern, can only support applications that use a HDFS-specific API. Second, disk-induced high latency and low bandwidth has been identified as a major bottleneck for storage systems [14]. Finally, metadata scalability has been pointed out in [15, 16] as another performance limiting factor.

Thus, for optimising MTC application performance, in-memory (or SSD-based) runtime file systems are being proposed [2, 17], as they increase application performance by improving I/O efficiency. As opposed to traditional, general-purpose distributed file systems, such runtime storage can have a limited life-time (the runtime of the application) and can accelerate I/O accesses to temporary files generated by applications. Subsequently, the output must be staged out to permanent storage.

While local, single-node, approaches to runtime storage are available, such as Aerie [18] or HyCache [19], our work focuses on distributed runtime file systems. Hence, MemFS is best compared to distributed systems that co-locate computation and storage to benefit from data-locality optimisations.

The AMFS Shell [2] is a state-of-the-art execution engine backed by an in-memory distributed file system that enables the parallelization of scripting applications. The underlying file system uses main memory for storing both data and metadata. To improve write performance, the file system issues only local writes. The execution engine then tries to improve read performance by moving computation to the data. In case a job reads more than one file, remote reads are performed and replication is used to store the remote files locally. Further, AMFS assumes that files fit in a node's memory.

HyCache+ [17] is a distributed file system caching middleware that accelerates application I/O accesses to persistent storage, such as GPFS. Similarly to AMFS, HyCache+ improves performance by issuing only local writes. To improve read performance, HyCache+ analyzes the job queue and prefetches the files needed by future tasks.

FusionFS [20] is another locality-based distributed file system designed with the goal of exascale scalability. This system can be used in conjunction with SPADE [21] to capture data provenance with negligible overheads [22]. Distributed metadata storage is implemented by means of ZHT [23] which leverages excellent scalability.

As opposed to those locality-based approaches, MemFS (briefly introduced in [24, 25, 26]) symmetrically stripes the files on the storage nodes based on a distributed hash function. This technique introduces three benefits. First, MemFS can leverage full bisection bandwidth of fast networks (such as InfiniBand) when reading or writing files. Second, MemFS achieves a balanced data distribution compared to the locality-based approaches. Finally, due to its locality-agnosticism, MemFS can be used in conjuction with any scheduler that evenly distributes tasks.

3. System Design

This section presents our approach for designing MemFS. First, we introduce the software components used for building MemFS and motivate our choices. Further, we present our *main*

contributions and *optimizations* that turn MemFS' building blocks into an in-memory runtime file system for MTC applications.

3.1. Components

The MemFS distributed file system consists of three key components. First, a *storage* layer that exposes each node's main memory for storing the data in a distributed fashion. Second, a *data distribution* component that enables each client to decide where to store data, or where to read data from. Third, a transparent *file system client* mounted on each compute node that serves as an interface between the storage layer and the MTC applications. In the following paragraphs we describe each component.

3.1.1. Storage

The actual DRAM storage layer we use is Memcached [27]. This is a distributed cache, extensively used in production, that achieves high-performance and good scalability. It uses simple key-value semantics and is locality-agnostic, does not offer replication capabilities and the storage servers do not communicate with each other. Hence, the client is responsible for data distribution and load balancing.

3.1.2. Data Distribution

MemFS equally distributes the files across the available Memcached servers, based on file striping. For mapping file stripes to servers, we use Libmemcached [28], a Memcached client library. For each stripe to be stored, we use the name of the file concatenated with the stripe number as key for the hash. Hence, based on the hash function value, the library decides which storage server to choose. In order to partition the data as evenly as possible between available storage servers, Libmemcached offers several hashing schemes. In our implementation we use the modulo hashing scheme. This is a simple hashing scheme that assigns each object to a storage server in a circular fashion, guaranteeing a balanced data distribution across all storage servers. For scenarios when nodes join and leave the system, a consistent hashing [29] scheme of Libmemcached can be used. However, in this paper we do not cover such a scenario which will be addressed in future work.

3.1.3. File System Client

We expose our storage system using a FUSE [30] layer, exposing a regular file system interface to the MTC applications. At startup, each FUSE client must be aware of each Memcached server and takes as input a list of IP addresses. Through the Libmemcached API, the FUSE file system communicates with the Memcached servers to store and retrieve files.

Figure 2 shows the overall system design of MemFS. The figure shows all nodes running a particular MTC application, reflecting our assumption that all application nodes also run MemFS. In general, however, it is not required that the application compute nodes also act as MemFS servers. It would also be possible to use a (partially) disjoint set of storage servers for running the Memcached processes, for example when the application itself has large memory requirements. It is necessary, however, that compute nodes run, next to the MTC application, the FUSE client and Libmemcached, in order to access the file storage. Figure 2 shows the example of a write operation, issuing Memcached *set* commands; for read operations, *get* commands would be used instead.

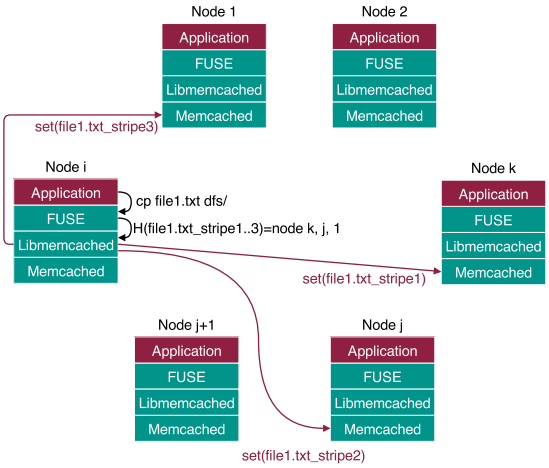


Figure 2: MemFS System Design

3.2. Implementation Building Blocks

For combining the above-mentioned components for storage, data distribution, and the file system client, the following building blocks are vital for an efficient implementation.

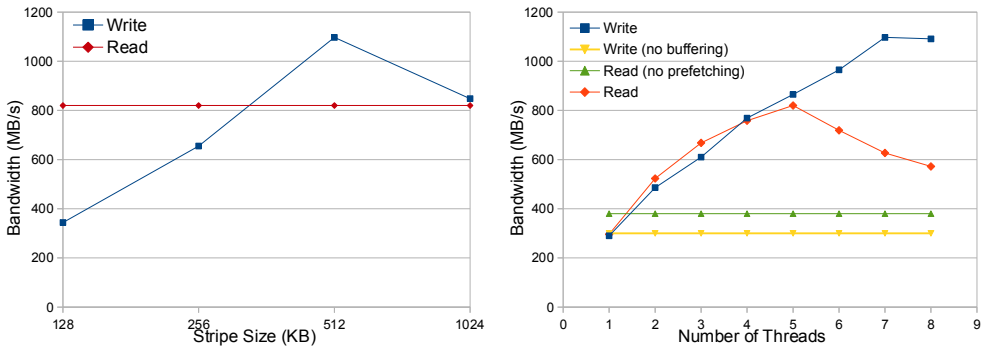
3.2.1. Striping Mechanism

The file system client is also responsible for the *striping mechanism* which is beneficial for several reasons: (1.) Using stripes as storage unit, file sizes are only limited by the total amount of memory on all Memcached servers. (Neither Memcached's object limit of 128MB nor the size of an individual node's memory limit the possible size of a file.) (2.) Distributing stripes across Memcached servers improves the read and write throughput by transferring data via parallel streams to/from multiple Memcached machines. (3.) The striping mechanism optimizes *small reads* for applications that do not read entire files. In this way, MemFS minimizes latency and data transfers for applications that read only small parts of (large) files.

3.2.2. Buffering and Prefetching

To further improve performance and network transfer efficiency, the file system client implements a *buffering* scheme for writing, and a *prefetching* scheme for reading. Buffering saturates write bandwith by using parallel streams to send the contents of the buffer to remote Memcached servers. Prefetching minimizes latency by overlapping communication and computation. Our prefetching scheme is simple and effective only for sequential reads: when an application requests data from a specific stripe, MemFS prefetches the consecutive stripes in a local cache.

Both buffering and prefetching work with thread pools to implement concurrent communication to the remote nodes. The buffering thread pool fetches file stripes from the *write* buffer and sends them, asynchronously, to the *memcached* servers. Whenever an application calls *close()*, or *flush()*, our file system waits until the *write* buffer has been emptied and then returns. The prefetching thread pool adds consecutive file stripes to the *read* buffer. For both buffering and prefetching, each thread is responsible for sending/retrieving a file stripe. Hence, when increasing the number of threads, MemFS is able to maximize the available bandwidth to achieve better



(a) Stripe Size Influence on MemFS I/O.

(b) Buffering and Prefetching Effect.

Figure 3: MemFS Design Decisions

performance. MemFS uses caches of 8MB per open file for the prefetching and buffering protocols.

Figures 3a and 3b motivate the design decisions we made when developing MemFS. We have chosen a stripe size of 512KB for the files stored in MemFS since this achieves the best bandwidth when writing files, as shown in Figure 3a. The stripe size does not influence reading speed because of the prefetching mechanism shown in Figure3b. This prefetching mechanism hides the communication latency and is thus not dependant on the stripe size, but rather on the number of threads that do prefetching.

3.2.3. Write-once semantics

With full POSIX-compliance, applications are allowed to read and write files both sequentially and non-sequentially, also concurrently by multiple processes. This flexibility, however, comes at the price of more complex file and concurrency management, causing runtime overhead. Since MTC applications exhibit only "single-write, multiple-read" I/O patterns, and files are usually written sequentially, MemFS restricts *write* operations to writing once, and only sequentially. Reading files in MemFS is POSIX-compliant: files can be read many times and in any order.

3.2.4. Metadata Organization

When a file is created, MemFS writes a special key containing the file name, along with an empty value via Libmemcached/Memcached. Once the file is closed after writing, the actual file size is replacing the empty key. When the file is subsequently opened for reading, the key containing the file name is looked up to retrieve the file size.

A similar protocol is used for *directory metadata*. When a directory is added, we create a Memcached key using the directory name, and an empty value. Whenever a file/directory is added under this directory, its corresponding Memcached value is appended with the file/directory name. For this operation we use the Memcached *append* function that is internally atomic and synchronized. When a file is deleted, we just add another entry to the directory value, marking the file as deleted.

Using this scheme, MemFS is able to achieve high performance and scalability for metadata operations. Scalability is achieved by distributing metadata over the available Memcached

servers, while high performance is achieved by using Libmemchached/Memcached lookups that can be done in constant time for each metadata operation.

3.2.5. Fault-Tolerance

At the moment, the only mechanism that MemFS could use to leverage fault-tolerance is replication. However, this incurs performance and storage limitation penalties. Assuming the replication factor is n, then the total storage capacity of MemFS would be decreased n times and n times more data will flow through the network when writing files. Hence, in this paper we do not include results using replication since it would limit the applicability and performance of MemFS. Moreover, current state-of-the-art MTC file systems also do not provide fault-tolerance, which is sacrificed for performance. Thus, designing a fault-tolerant MemFS will be addressed in future work.

4. Evaluation

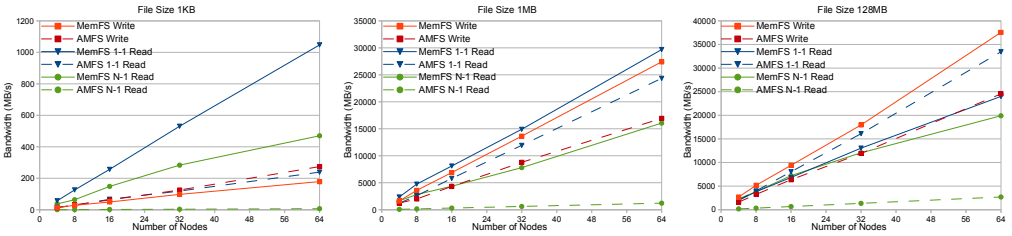
In this section we evaluate the MemFS file system and discuss the achieved results. The experiments were executed on our local DAS4 multi-cluster system [31] and on the Amazon *Elastic Compute Cloud* (EC2) [32].

The first part of the evaluation is performed on DAS4. The (in total 74) 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 chose to use the IP over InfiniBand (IPoIB) interface of the latter, which achieves approximately 1GB/s bandwidth.

On DAS4, we compare the performance and scalability achieved by MemFS to the AMFS [2] file system, a state-of-the-art in-memory file system for many-task computing. While both file systems provide similar write-once semantics, their design is inherently different: AMFS is locality-based, while MemFS is locality-agnostic. AMFS improves 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. In case multiple files are read per scheduled job, expensive remote reads become necessary. MemFS, in contrast, uniformly distributes file stripes across storage nodes by means of a distributed hash function to achieve balanced memory consumption, while utilizing the aggregate bandwidth among all nodes. For all experiments, the compute nodes also operate as storage nodes, for both AMFS and MemFS.

In the second part of our evaluation we study MemFS' applicability on the Amazon EC2 cloud. In this context, we analyze the scalability and performance of MemFS in a virtualized environment. When running on the Amazon EC2 cloud, we chose to use the c3.8xlarge instance types. These feature 32 virtual compute cores, 60GB of memory and are connected with 10G Ethernet links. Our *iperf* [33] 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.

When running our experiments, the memory of each compute node is divided as follows. Out of the total memory of a node, we reserve 4GB for running the applications or benchmarks and the operating system. The rest of the system memory is used by either MemFS or AMFS for storing the data generated by the applications or benchmarks.



(a) I/O Op. Bandwidth - File Size 1KB. (b) I/O Op. Bandwidth - File Size 1MB. (c) I/O Op. Bandwidth - File Size 128MB.

Figure 4: MTC Envelope Bandwidth Comparison

4.1. MTC Envelope Evaluation

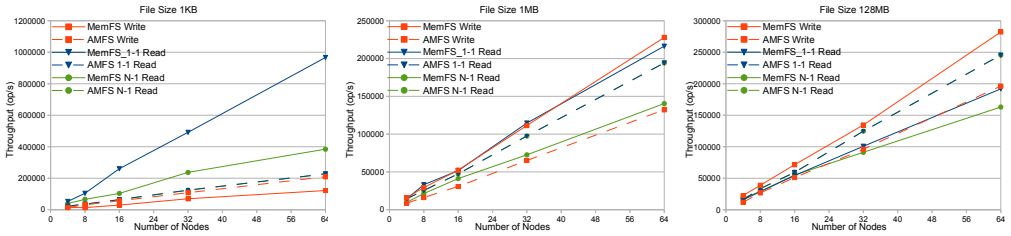
The MTC Envelope [34] introduces a set of eight metrics that fully characterize a system's ability to run MTC applications at a certain scale. The set of metrics is the following: *write* throughput and bandwidth, *1-1 read* throughput and bandwidth, *N-1 read* throughput and bandwidth and *metadata* operations (open, create) throughput. Here, the *1-1 read* depicts the case when each node reads a different file, while the *N-1 read* is the case when all nodes read the same file. Moreover, for the I/O operations, the bandwidth measures the volume of data an application could read or write per time unit; whereas operation throughput measures how many *read()* or *write()* calls the application is able to perform per time unit. While both types of metrics are related, read/write bandwidth reports on actual data movement while read/write throughput captures computational overhead of the operations.

We have chosen three file sizes - small (1KB), medium (1MB) and large (128MB) for which we measure the MTC Envelope metrics. These are representative file sizes for MTC applications as, for example, Montage mainly operates on file sizes in orders of megabytes, but it also generates some files in orders of kilobytes. In contrast, BLAST operates on files in the order of tens or hundreds of megabytes.

The results were generated using two well-known file system benchmarks: iozone [35] for I/O operations and mdtest [36] for metadata. For AMFS, we followed the benchmarking pattern presented in [2]. Thus, all *1-1 read* operations are local since it benefits from locality-aware scheduling enabled by AMFS Shell. In case of *N-1 read*, a file is first *multicast* from a source node and then the iozone benchmark is applied on local copies of the data on all compute nodes. For the *N-1 read* bandwidth reported, the time taken by the multicast is taken into account, for *N-1 read* throughput, it is ignored.

In Figures 4a and 5a we show a comparison of the MTC Envelope I/O operations using a file size of 1KB. MemFS *1-1 read* and *N-1 read* show excellent performance and scalability, outperforming corresponding AMFS operations and also both MemFS and AMFS *write*. With such small files, MemFS *read* operations are much faster than MemFS *write* because of two reasons. First, our buffering protocol cannot work with files smaller than the stripe size (512KB), while our prefetching mechanism is not dependant on the stripe size. Second, Memcached is reported to perform better for *get* rather than *set*.

For 1MB-sized files, as depicted in Figures 4b and 5b, we notice that the MemFS *write* buffering protocol starts to show its effectiveness, leveraging linear scalability and superior performance than *N-1* read. The latter achieves less bandwidth/throughput than *1-1 read* because, even with the file striping protocol, when all nodes read the same file, the maximum achievable



(a) I/O Op. Throughput - File Size 1KB. (b) I/O Op. Throughput - File Size 1MB. (c) I/O Op. Throughput - File Size 128MB.

Figure 5: MTC Envelope Throughput Comparison

bandwidth/throughput is that of a single Memcached server for each file stripe. We also notice that in this case, the MemFS approach of uniformly spreading data across compute nodes, based on a distributed hash function, when using a high-speed network achieves better I/O performance than locality-based AMFS for all MTC Envelope metrics.

When benchmarking with 128MB files (Figures 4c and 5c), we notice that MemFS is faster for *write* and *N-1 read* than AMFS. However, for the *1-1 read*, AMFS performs better because all reads are local, while MemFS read generates high volumes of network traffic when 128MB are being read. We also notice a decrease in bandwidth/throughput achieved for 128MB files compared to 1MB files. This is because our prefetching mechanism fetches more data from the network than in the 1MB case which puts more pressure on the Memcached servers and also on the network layers of the operating system.

When reading 1KB and 1MB files over the network, MemFS is faster than AMFS for the *1-1 read*, even though the AMFS reads are local. For small files, the total execution time for *1-1 read* is latency-bound rather than bandwidth-bound. Furthermore, the locality-aware scheduling algorithm of AMFS is slower than the locality-agnostic scheme used for MemFS, and generates an even larger latency for AMFS. In contrast, reading 128MB files is bandwidth-bound, such that when reading large amounts of data, the scheduling scheme of AMFS does not limit its performance, while MemFS is limited by the network bandwidth.

For all file sizes, MemFS outperforms AMFS for the *N-1 read* operation. This behaviour is explained by the performance achieved with the software *multicast* implemented in AMFS Shell. In [2], the authors show that multicast performance is determined by latency, bandwidth and file size for a certain scale. We manage to hide these overheads by prefetching and by symmetrically distributing a given file over a set of storage nodes.

While *N-1 read* bandwidth is low for AMFS, the *N-1 read* throughput is equal to the *1-1* read. This is explained by the fact that bandwidth is lowered by the multicast operation, while the reading throughput is determined by the local read issued after the multicast finishes.

Figure 6 shows metadata performance for the two file systems. We notice that MemFS achieves linear scalability for both *open* and *create*, while only AMFS *open* scales linearly. Non-linear scalability for AMFS *create* is explained as follows: AMFS distributes file metadata over all servers based on a hash function of the file name; according to [2], this distribution is not uniform. AMFS *open* performs much better than its MemFS counterpart because all queries are local. For MemFS, if N storage nodes are available, for opening a file there is a $1/N$ probablity that the operation will be local. MemFS *open* outperforms MemFS *create* due to Memcached behaviour: *get* is faster than the conjunction of *set* and *append*.

Table 1: MTC Envelope Scale 64, File Size 1MB, in MB/s

	AMFS IPoIB	**MemFS IPoIB**	**AMFS 1GbE**	**MemFS 1GbE**
Write Bw	16934	27403	16934	4928
1-1 Read Bw	24351	29686	24351	4850
1-1 Read Bw (remote)	6400		3385	
N-1 Read Bw	1216	16053	950	1232
Create	25073	22166	20424	21817
Open	221175	61097	168471	40198

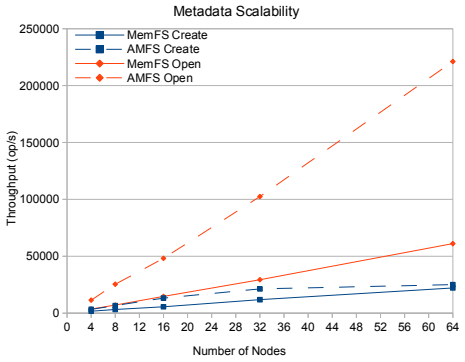


Figure 6: Metadata Operations Throughput

In Table 1 we present MTC Envelope metrics for both AMFS and MemFS at a scale of 64 nodes, using a file size of 1MB. For this test we also used the commodity 1GbE network in our cluster. We explicitly show here results for AMFS remote *1-1 read* performance. We chose to show this to cover the scenario when an application task reads more than one input file. In such a case, AMFS cannot guarantee locality for the subsequent files. As a consequence, the bandwidth achieved at a scale of 64 compute nodes is degraded by a factor of approximately 4 when the file system runs on top of IPoIB. For the 1GbE case, the performance decreases with a factor of approximately 7. In the worst-case scenario of losing data locality for a subset of the file reads, MemFS outperforms AMFS by a factor of 4.63 when running over IPoIB. Moreover, in such a case, even with a much slower network, MemFS is faster than AMFS by a factor of 1.4. The *N-1 read* performs better on 1GbE for MemFS compared to AMFS.

Table 1 also shows that metadata throughput is also determined by the network characteristics for both AMFS and MemFS. However, the decrease in performance is less visible than for bandwidth. This can be explained by the fact that metadata performance is mostly influenced by network latency rather than network bandwidth.

4.2. Application Benchmarks & Multicore Scalability

AMFS Shell has been designed such that it schedules one task per compute node. Since our evaluation was executed in a multicore cluster and on multicore EC2 virtual machines, running one task per compute node would under-utilise the available resources. To fully use the available compute resources, we have modified AMFS Shell such that it can schedule multiple jobs at a time on a given node. This multicore-aware scheduler preserves the data-locality scheme employed by AMFS when running tasks that store data in the AMFS file system. When using

Table 2: Application Description

Application	Input Size	Runtime Data	File Size
Montage 6×6	4.9 GB	50 GB	1-4.4MB
Montage 12×12	20 GB	250 GB	1-4.4MB
Montage 16×16	34 GB	450 GB	1-4.4MB
BLAST	57 GB	200 GB	10-120MB
BLAST (EC2)	57 GB	200 GB	5-60MB

MemFS as the storage backend, the multicore-aware scheduler simply submits multiple jobs at a time, without taking data-locality into account.

Using this multicore-aware version of the AMFS Shell scheduler, we measure application performance for MemFS compared to AMFS, while also assessing each systems' scaling behaviour. By scaling *up*, or *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 *out*, or *horizontally*, we analyze the system behaviour while gradually increasing the number of compute nodes.

The applications we use are the well-known Montage [1] and BLAST [5] workflows. Montage is an astronomy MTC application that, given a set of input images of a galaxy, builds a mosaic. BLAST is a bioinformatics application that searches for specific nucleotide sequences in a given database.

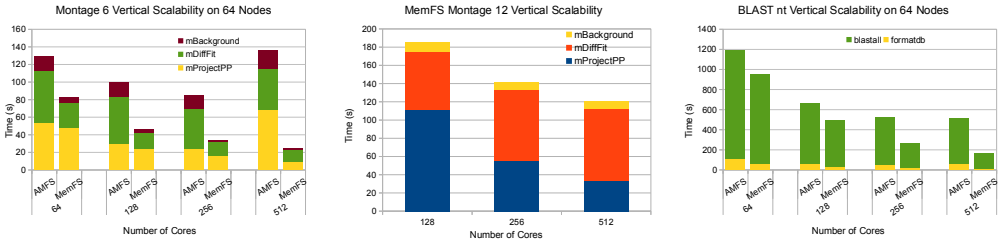
According to [37, 38], the two applications feature different resource utilization. Montage exhibits low memory and CPU utilization, but high I/O. In contrast, BLAST shows high CPU utilization anbut only medium memory and I/O usage.

Figures 1a and 1b depict workflow structure and dataflow patterns, for Montage and BLAST respectively. Analyzing the Montage workflow graph, one can notice that the *mDiffFit* task takes as input two files generated by distinct *mProjectPP* jobs. Unless the two *mProjectPP* jobs were executed on the same node, when scheduling *mDiffFit*, AMFS Shell cannot guarantee data locality for reading both files, which leads to a performance degradation for the second read. The same logic applies to the *blastall* BLAST task which reads two input files. One could argue that, by using AMFS *collective operations*, all needed files could be made available in advance to all compute nodes. However, at least in the case of Montage, this is not feasible since the output of *mProjectPP* is in the order of tens of GB and could easily saturate each node's main memory. In the case of BLAST, it is achievable to have the query files available on all nodes, since their total size is small. Then, AMFS Shell could schedule *blastall* jobs locally to each database fragment to achieve data locality, and thus both file reads will achieve similar performance.

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. In the experiments we describe next, we show that for the Montage workflow, this assumption holds, and *mDiffFit* performance is superior for MemFS compared to AMFS.

Table 2 shows the application use cases that we used for the evaluation. First, we chose to run a 6×6 Montage mosaic 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. The other two Montage use cases we used for benchmarking are 12×12 and 16×16 mosaics also centered on the M17 galaxy. Their input sizes are 20 and 34GB and they generate at runtime approximately 250 and 450GB, respectively.

For the BLAST application, our benchmarking scenario follows the same pattern as presented



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

Figure 7: Vertical Scalability Evaluation

in [38]. 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 MTC file system (either AMFS or MemFS) and *formatdb* is applied to each fragment. When running on DAS4 we generated 512 database fragments since we have only 512 compute cores available. Thus, there are 512 *formatdb* tasks. Then, a total number of 8192 *blastall* queries are run against the database fragments. The results are aggregated using 16 *merge* jobs. The data volume generated at runtime accounts for approximately 200GB. However, when running on EC2 on 32 c3.8xlarge virtual machines, there are 1024 virtual cores available. Hence, we split the *NCBI nt* database into 1024 fragments. These generate 1024 *formatdb* tasks and 16384 *blastall* tasks. Even if for this use case the number of tasks is doubled, running BLAST on the EC2 cloud still generates approximately 200GB of data, as on DAS4. This is because the database we query against is the same, however, only the fragments are half the size, but twice as many. Thus, the results between the two different runs are comparable as they are equal in data size.

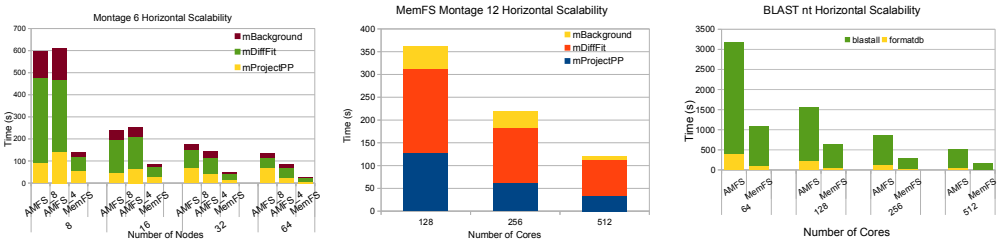
4.2.1. DAS4 Results

The experimental setup is as follows. We use the AMFS Shell scheduler on top of either AMFS or MemFS. In conjunction with MemFS, the AMFS Shell scheduler cannot perform locality-aware scheduling, thus all tasks are submitted in a uniform manner to all compute nodes.

Figure 7a shows the vertical scalability of the two file systems for the Montage 6 workflow. The results were determined on 64 nodes, using gradually 1, 2, 4, and 8 compute cores each. MemFS shows good scalability up to 512 cores, while AMFS only up to 256 compute cores.

Figure 8a depicts the horizontal scalability comparison of the two file systems with Montage 6. We scaled out the systems from 8 to 64 compute nodes. Because the vertical scalability results (Figure 7a) showed that AMFS could not scale up to 8 cores per node in the 64 node scenario, in the horizontal scalability graph we decided to show both the 8 core scenario, together with the 4 core scenario which achieves best efficiency 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, MemFS leverages better performance. Superior performance of MemFS follows from the MTC Envelope results: Figure 4b shows that for all metrics, MemFS achieves higher bandwidth than AMFS. The results of Figure 4b are relevant in this case because Montage operates with files with sizes in the order of megabytes.



(a) Scaling out from 8 to 64 Nodes. (b) Scaling out from 8 to 64 Nodes. (c) Scaling out from 8 to 64 Nodes.

Figure 8: Horizontal Scalability Evaluation

Table 3: AMFS Memory Distribution for Montage 6

Number of Nodes	Scheduler Node	Other Nodes
8	19 GB	9.5 GB
16	17 GB	5.5 GB
32	16 GB	3 GB
64	16 GB	1.8 GB

An interesting observation is that while on 32 and 64 nodes AMFS performs better running 4 tasks per node, the 8 and 16 node setups perform better when running 8 tasks per node. Table 3 offers a better understanding of the AMFS' data distribution when running Montage. In this table, by "scheduler node" we denote the node that runs the data aggregation stages of Montage (mImgTbl, mBgModel, mConcatFit - see Figure 1a). The rest of the nodes run only the parallel tasks (mProjectPP, mDiffFit, mBackground). Due to its replication scheme, AMFS aggregates large amounts of data in the "scheduler node", while the other nodes exhibit a balanced data distribution.

Using Table 3, we can infer that in the 32 and 64 node case, AMFS cannot scale to 8 cores per node because it cannot exploit data locality well. The nodes that run only parallel tasks contain only little data and they are only able to achieve data locality for one file per task. For the subsequent files of the same task, expensive remote reads (see Table 1) need to be performed from the "scheduler node" which holds most of the data, transforming this node into a centralized

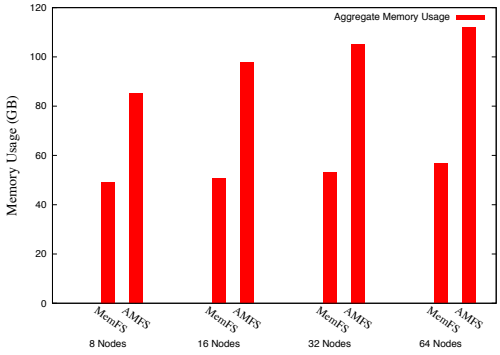


Figure 9: Montage 6 Memory Consumption

bottleneck. With fewer nodes, in the 8 and 16 scale scenario, AMFS achieves better performance with 8 tasks per node rather than 4, simply because the nodes hold more data, and benefit from higher chances of achieving data locality for more than one file of a given task.

Figure 9 depicts the aggregate memory consumption for the two file systems with Montage 6. The measurements were taken at the end of each experiment presented in Figure 8a on 8 to 64 nodes. The graph shows the superior memory management of MemFS for all scales. Increased data usage of AMFS can be explained by the replication-on-read performed by AMFS for improving data locality. For the same reason, when increasing scale, the memory consumption of AMFS increases at a higher rate than MemFS, since there is more replication involved. MemFS exhibits a much less steep curve for aggregate memory usage, which is generated by the overhead of running Memcached and the FUSE file system process on more servers. For MemFS, each FUSE process allocates around 200MB for storing various data structures that are used for optimizing I/O performance.

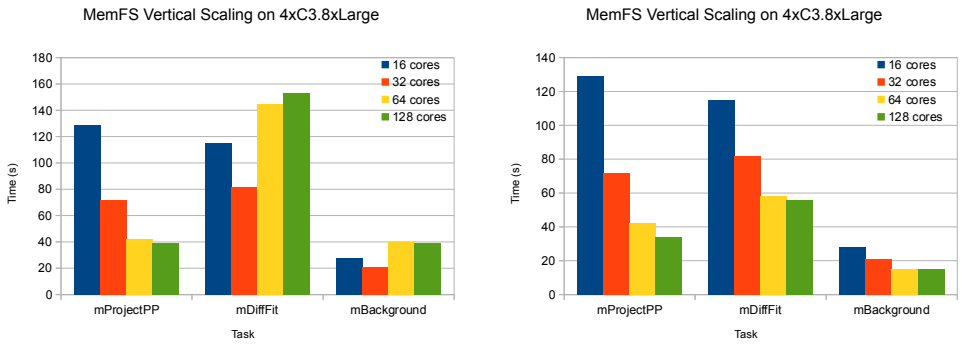
AMFS's increased memory utilization caused by its replicate-on-read policy prevented us from running larger Montage workflows, such as the 12×12 Montage mosaic. This workflow has a 20GB input and, during runtime, it generates approximately 250GB of data. AMFS is unable to run this workflow because the "scheduler node" crashes when trying to accumulate large amounts of data that do not fit in its main memory. Figures 7b and 8b show that our system, MemFS, does not only scale vertically or horizontally, but also while increasing the problem size, and is able to handle much larger amounts of data.

Figure 8b shows excellent horizontal scalability while scaling out from 16 to 64 nodes. All 8 cores of each compute node have been used to run tasks. In the case of vertical scalability (Figure 7b), 64 nodes have been used, while the Montage tasks were run on 2, 4 and 8 compute cores each. An interesting observation is that only the *mProjectPP* and *mBackground* parallel tasks are able to scale, while the *mDiffFit* tasks show similar performance on 2, 4 and 8 cores per node. This behaviour can be explained by the number and size of input files of the jobs. *mProjectPP* and *mBackground* read one input file of approximately 2MB and output one file of 4MB and 2MB, respectively. *mDiffFit* reads two input files of 4MB and outputs one file of 2MB, thus causing more network traffic and saturating the achievable bandwidth. Hence, the *mDiffFit* phase is not able to scale up to 4 and 8 cores because its scalability is limited by the network bandwidth.

For the BLAST evaluation, Figure 7c shows the vertical scalability of AMFS and MemFS on 64 nodes, while scaling up the number of compute cores used from 1 to 8 each. While AMFS scales only up to 4 cores per node, MemFS is able to scale up to 8 cores per node. The horizontal scalability experiments are depicted in Figure 8c. We scale both systems out from 8 to 64 nodes, while scheduling 8 tasks per node. The experimental results show that both systems scale well horizontally, although MemFS leads to much faster completion times. This behaviour follows from AMFS' inability to scale to 8 cores per node (Figure 7c): while in the 1 and 2 core per node case, MemFS is only about 18% faster than AMFS, in the 4 and 8 core per node case, the difference becomes much higher. Hence, when scaling horizontally from 8 to 64 nodes, while using 8 cores per node (best configuration for both file systems), MemFS shows better resource efficiency which leads to much faster completion times.

4.2.2. Cloud Results

Benchmarking MemFS and AMFS on our DAS4 cluster on up to 512 compute cores, we have shown that due to its design of efficiently using the network bandwidth and uniformly distributing data across nodes, MemFS not only is able to achieve better vertical scalability and faster



(a) 1 FUSE Mountpoint

(b) 1 FUSE Mountpoint per Process

Figure 10: MemFS Vertical Scalability While Running Montage 6 on 4 c3.8xlarge

completion times, but also run larger problem sizes. In this section, we study MemFS' applicability on the Amazon EC2 cloud and check if similar results could be achieved. Because MemFS largely depends on the network characteristics, we have chosen to run on *c3.8xlarge* instances. These instances are linked with 10Gb Ethernet adapters and feature 32 virtual compute cores, which are organized into two NUMA nodes, and 60GB of main memory. Our *iperf* tests have shown that the achievable bandwidth between these instances is approximately 1GB/s. Thus, the networking environment is comparable to our DAS4 cluster, while the nodes have more compute cores and more available DRAM. In this context, it is interesting to study MemFS' vertical scalability behaviour with more than 8 cores per node.

To check whether MemFS scales well vertically in this environment, we ran a Montage 6 workflow on 4 *c3.8xlarge* virtual machines while using 4, 8, 16 and 32 cores on each instance. Figure 10a shows the outcome of this experiment. The results show that MemFS is not able to scale beyond 8 cores per node. After analyzing the problem, we concluded that the bottleneck was not in the MemFS design or implementation, but rather in the FUSE design. The FUSE kernel module uses for each mountpoint a *spinlock* which is not able to scale when accessed from different NUMA nodes.

Hence, we modified the deployment of MemFS to use multiple mountpoints, one per each application task. In this way, we have been able to scale beyond 8 cores per node. Figure 10b shows MemFS vertical scalability with this new deployment strategy - *one-mountpoint-per-application-process*. Here, even though the speedup is not perfect, our file system does not slow down the application runtime as shown in Figure 10a. As we will further show, the speedup is not perfect because in this setup MemFS saturates the available network bandwidth.

After fixing this scalability issue, we have compared MemFS with AMFS, by running the same Montage 6 instance. Since the EC2 *c3.8xLarge* instance types have up to 32 compute cores, we assessed both systems' ability to scale vertically. Figure 11 depicts the achieved results. Similar to the DAS4 experiment, MemFS achieves much faster completion times on 4 and 8 compute cores, due to the locality imbalances of AMFS.

Unfortunately, AMFS could not run with more than 8 application processes per compute node. There are two reasons for this behavior. First, AMFS' storage imbalance prohibits scaling properly, even from 4 to 8 compute cores. Second, FUSE is unable to scale to more than 8

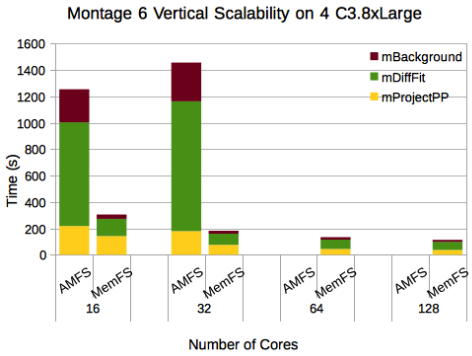
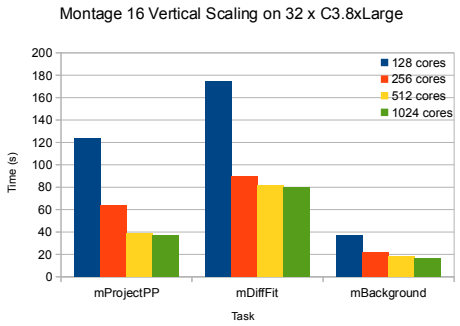
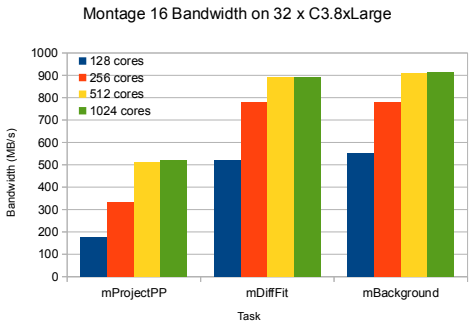


Figure 11: MemFS vs. AMFS vertical scalability on 4xC3.8xLarge (Amazon EC2)



(a) Montage 16 Execution Time



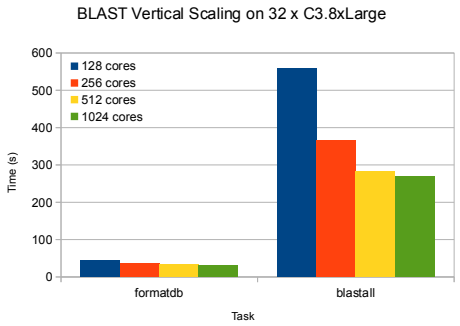
(b) Montage 16 Achieved Bandwidth per Node

Figure 12: Montage 16 Vertical Scalability on 32 c3.8xlarge

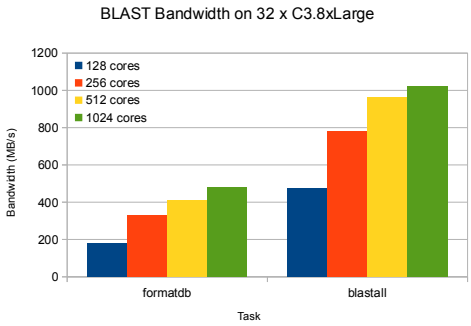
compute cores with only a single mountpoint. While we have fixed this issue in MemFS, for AMFS it is not straightforward to use multiple mountpoints per compute node and modifying the AMFS source code is beyond the scope of this paper. Since AMFS could only run on 8 compute cores from the total of 32 of the *c3.8xlarge* instance type, we conclude that AMFS is not well suited for these "fat" nodes and we do not make further comparisons between MemFS and AMFS.

Further, we assessed MemFS vertical scalability on larger problem sizes and on a higher number of virtual machines. We ran Montage-16 and BLAST on 32 c3.82xlarge nodes, using gradually 4, 8, 16 and 32 cores on each virtual machine. Our largest setup uses 1024 virtual compute cores, twice as many cores as used on the DAS4 cluster. Figures 12a and 13a show MemFS scalability on 32 machines 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 *mProjectPP* and *formatdb* show better vertical scalability. To investigate the scaling behaviour of the I/O-bound stages of the workflows, we have monitored the network activity of the EC2 virtual machines. Figures 12b and 13b show the network usage for Montage and BLAST, respectively.

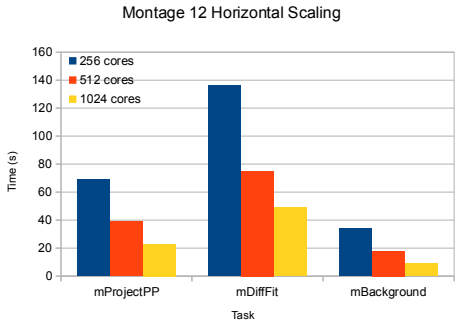


(a) BLAST Execution Time

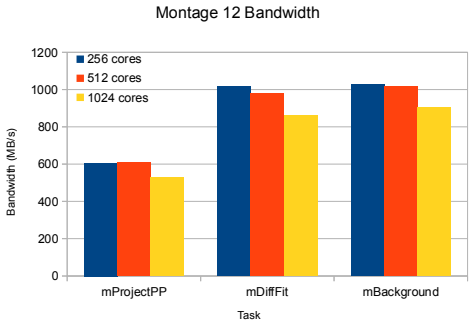


(b) BLAST Achieved Bandwidth per Node

Figure 13: BLAST Vertical Scalability on 32 c3.8xlarge



(a) Montage 12 Execution Time



(b) Montage 12 Achieved Bandwidth per Node

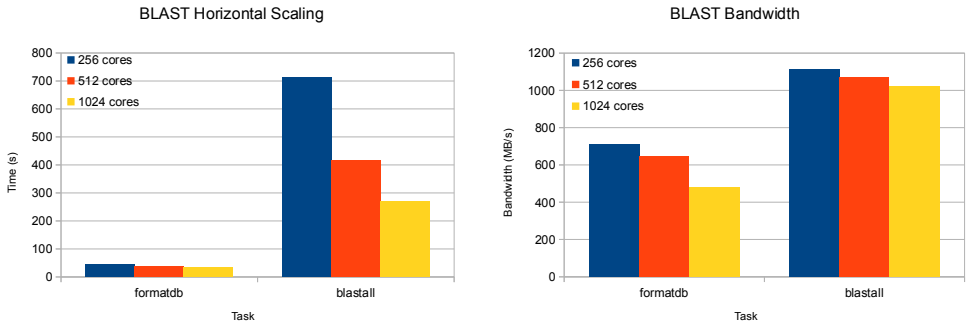
Figure 14: Montage 12 Horizontal Scalability on 8-32 c3.8xlarge, 32 Cores Each

The figures show the bandwidth utilisation per node. These experiments show that the I/O-bound stages from both Montage and BLAST saturate the network bandwidth (of approximately 1GB/s) when running on 16-32 cores. As a consequence, the vertical scalability of MemFS is bound by the network bandwidth.

Using 8, 16, and 32 nodes, where all 32 cores of each node were utilized, we evaluated the horizontal scalability of MemFS. Figures 14a and 15a show MemFS' scaling behaviour when running Montage 12 and BLAST, respectively. The results show good horizontal scalability for our distributed file system when running real-world applications on the EC2 cloud. Moreover, Figures 14b and 15b confirm the previous results: for the I/O-bound workflow stages, MemFS is bound by the network bandwidth.

It is important to emphasize that MemFS' vertical scalability is only limited by network bandwidth when running real-world MTC applications on a public compute cloud. This shows that MemFS is able to *fully saturate* premium networks (InfiniBand, 10Gb Ethernet) bisection bandwidth and its applicability is not limited to tightly-coupled compute clusters.

To further explain and analyze this behaviour we have designed a microbenchmark derived



(a) BLAST Execution Time

(b) BLAST Achieved Bandwidth per Node

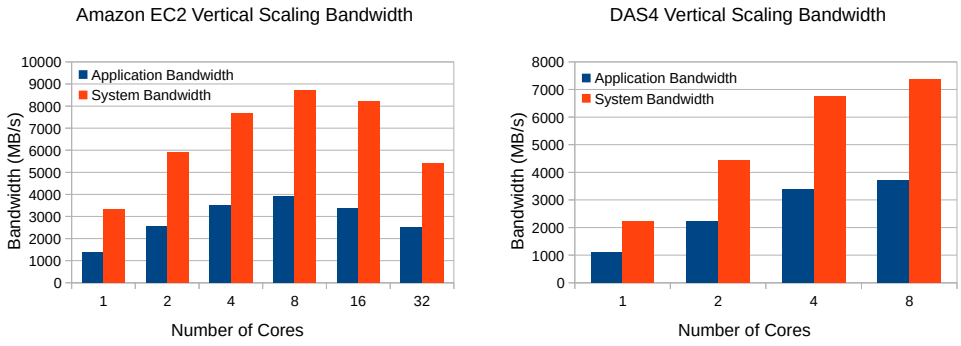
Figure 15: BLAST Horizontal Scalability on 8-32 c3.8xlarge, 32 Cores Each

from the MTC Envelope [34]. This microbenchmark measures MemFS' achieved bandwidth when using increasing numbers of compute cores per node to run application processes. For this experiment we have used *iozone* to read/write files using block sizes of 4KB, the same block sizes used by Montage and BLAST when doing I/O. This experiment is I/O bound, and does not utilize as much CPU or memory, as opposed to Montage or BLAST. Figure 16a shows the results achieved when running this benchmark on 8 *c3.8xLarge* instance types. We have measured both the *system bandwidth* and the *application bandwidth*. The former measures the MemFS total bandwidth, which is composed by the *application I/O* and the *memcached I/O*. The latter is just the *application I/O bandwidth*. It is straightforward to conclude why the *application bandwidth* is approximately half the *system bandwidth*: the application generated I/O is replicated in the memcached network traffic since all the data that is written/read by an application is also read/written by memcached.

Since this microbenchmark is only doing I/O, the network bandwidth is saturated already by 8 compute cores, as opposed to the real-world applications Montage or BLAST that needed 16 or 32 compute cores to saturate the network bandwidth. For this microbenchmark, when reaching 16 or 32 compute cores, we notice a slight decrease in the achieved bandwidth. This is a thrashing behaviour of the network driver. To show that this is not only applicable when running on Amazon EC2, and MemFS is able to saturate the network bandwidth independent of the platform, we have run an analogous experiment on the DAS4 cluster. These results are presented in Figure 16b. On 8 DAS4 nodes, the network bandwidth is saturated when running application processes on 8 cores per node.

4.3. Summary & Discussion

Co-designing in-memory runtime file systems with locality-aware schedulers for running MTC-applications has proven to achieve better performance than the traditional approach of using disk-based distributed file system, such as GPFS [2]. Storing data in memory removes the bottleneck of mechanical disks, and application tasks are directed to nodes that store the data to be processed to make use of the large local-memory bandwidth. In contrast, we have proposed a new design of an in-memory runtime file system that is locality-agnostic: MemFS evenly distributes data across system nodes and fully utilizes the bisection bandwidth of premium networks such as InfiniBand or 10Gb Ethernet. Our evaluation has shown that our design is



(a) Amazon EC2

(b) DAS4

Figure 16: MemFS Bandwidth Analysis Microbenchmark

able to outperform a state-of-the-art locality-centric approach when running both synthetic MTC workloads and real-world MTC applications. Moreover, MemFS was able to saturate the 10Gb Ethernet bandwidth of a public cloud while running on 1024 cores.

First, our results have shown that MemFS achieves similar or better performance when running micro-benchmarks. MemFS outperforms AMFS in the MTC Envelope I/O performance metrics with one exception: AMFS achieves better bandwidth for *1-1 read* for large file sizes (128MB). Also, even though AMFS metadata throughput is superior to MemFS, we achieve linear scalability for metadata operations. However, for bandwidth-bound applications that generate large amounts of data, such as Montage or BLAST, metadata is not a performance-limiting factor, and MemFS' superiority in MTC Envelope I/O metrics guarantees better application performance.

Second, when running the Montage and BLAST workflows, MemFS attains superior running times for the parallel stages. Moreover, we showed that AMFS' replication policy leads to increased memory consumption, prohibiting the use of larger data sets as the node memory gets exhausted. Furthermore, we showed that even though both file systems exhibit good horizontal scalability, MemFS achieves superior vertical scalability and hence, better resource utilisation, being able to scale to 512 compute cores on our DAS4 cluster.

Finally, we have deployed MemFS on a public comercial cloud on up to 1024 compute cores. Our distributed file system scales well both vertically and horizontally and is able to fully saturate the 10Gb Ethernet bandwidth while running the Montage and BLAST workflows. Hence, MemFS performance is only bound by network capacity. With current advances in premium network technologies, and their increase in bandwidth, the design we propose eliminates the need for data-locality techniques.

5. Conclusions and Future Work

We have presented MemFS, a fully-symmetrical, in-memory distributed runtime file system. Its design is based on uniformly distributing file stripes across the storage nodes belonging to an application by means of a distributed hash function, purposefully sacrificing data locality for balancing both network traffic and memory consumption. This way, reading and writing files can

benefit from full network intersection bandwidth, while data distribution is balanced across the storage servers.

Our evaluation shows that MemFS is able to scale well for all MTC Envelope metrics. Moreover, MemFS achieves superior performance than the state-of-the-art, locality-based, AMFS file system for most of the MTC Envelope metrics. MemFS also shows superior performance with the Montage and BLAST workflows. We have shown that AMFS' design for data locality prevents it from storing larger data sets, like from the 12×12 Montage workflow. In contrast, MemFS is able to run 12×12 Montage on any configuration of nodes for which the sum of the available memory from all nodes is big enough to store the total data generated at runtime.

Our current evaluation, limited to 64 nodes by the size of our cluster, indicates that MemFS scales well with increasing numbers of compute and storage nodes as memory pressure lowers when storage nodes are added. Furthermore, MemFS shows excellent vertical scalability, successfully scaling up to 512 compute cores, thus leveraging good resource utilisation. In contrast, the locality-based approach of AMFS is unable to scale up to 512 compute cores, with both the Montage and BLAST applications, due to locality-induced storage imbalance.

MemFS' applicability is not limited to tightly-coupled compute clusters. Our evaluation on a public commercial cloud shows that MemFS scales well on up to the 1024 virtual compute cores we have used. Its vertical scalability is only limited by the network bandwidth when running the I/O bound tasks of real-world applications Montage and BLAST.

With ongoing work, we are exploring ways to further improve MemFS' performance, for example by using RDMA instead of IP-based communication. In addition, we investigate schemes to dynamically scale out storage nodes for handling growing storage requirements at application runtime.

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, and the AMFS authors for making their software available to us.

References

[1] 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 4 (2) (2009) 73–87.

[2] Z. Zhang, D. S. Katz, T. G. Armstrong, J. M. Wozniak, I. Foster, Parallelizing the execution of sequential scripts, in: High Performance Computing, Networking, Storage and Analysis (SC), 2013 International Conference for, IEEE, 2013.

[3] F. B. Schmuck, R. L. Haskin, GPFS: A shared-disk file system for large computing clusters., in: FAST, Vol. 2, 2002, p. 19.

[4] R. B. Ross, R. Thakur, et al., PVFS: A parallel file system for linux clusters, in: in Proceedings of the 4th Annual Linux Showcase and Conference, 2000, pp. 391–430.

[5] S. F. Altschul, W. Gish, W. Miller, E. W. Myers, D. J. Lipman, Basic local alignment search tool, Journal of molecular biology 215 (3) (1990) 403–410.

[6] S. Liang, R. Noronha, D. K. Panda, Swapping to Remote Memory over InfiniBand: An Approach using a High Performance Network Block Device, in: Cluster Computing, IEEE, 2005, pp. 1–10.

[7] Dutch National Supercomputer, `https://www.surfsara.nl/systems/cartesius`.

[8] stream, `http://www.cs.virginia.edu/stream/` (2014).

[9] P. Schwan, Lustre: Building a file system for 1000-node clusters, in: Proceedings of the 2003 Linux Symposium, Vol. 2003, 2003.

[10] GlusterFS, http://www.gluster.org/.
[11] F. Hupfeld, T. Cortes, B. Kolbeck, J. Stender, E. Focht, M. Hess, J. Malo, J. Marti, E. Cesario, The XtreemFS Architecture - A Case for Object-based File Systems in Grids, Concurrency and Computation: Practice and Experience 20 (17) (2008) 2049–2060.
[12] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. Long, C. Maltzahn, Ceph: A scalable, high-performance distributed file system, in: Proceedings of the 7th symposium on Operating systems design and implementation, USENIX Association, 2006, pp. 307–320.
[13] K. Shvachko, H. Kuang, S. Radia, R. Chansler, The hadoop distributed file system, in: Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on, IEEE, 2010, pp. 1–10.
[14] 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 54 (7) (2011) 121–130.
[15] S. R. Alam, H. N. El-Harake, et al, Parallel I/O and the metadata wall, in: Proceedings of the sixth workshop on Parallel Data Storage, 2011.
[16] I. Raicu, I. T. Foster, P. Beckman, Making a case for distributed file systems at exascale, in: Proceedings of the third international workshop on Large-scale system and application performance, ACM, 2011, pp. 11–18.
[17] D. Zhao, K. Qiao, I. Raicu, Hycache+: Towards scalable high-performance caching middleware for parallel file systems, in: CCGrid, 2014.
[18] H. Volos, S. Nalli, S. Panneerselvam, V. Venkatanathan, P. Saxena, M. M. Swift, Aerie: Flexible file-system interfaces to storage-class memory, in: Eurosys, 2014.
[19] D. Zhao, I. Raicu, Hycache: a user-level caching middleware for distributed file systems, in: International Workshop on High Performance Data Intsensive Computing, IEEE IPDPS, Vol. 13, 2013.
[20] D. Zhao, et al., Distributed file systems for exascale computing, ACM/IEEE Supercomputing (Doctoral Showcase).
[21] A. Gehani, D. Tariq, Spade: Support for provenance auditing in distributed environments, in: Proceedings of the 13th International Middleware Conference, Springer-Verlag New York, Inc., 2012, pp. 101–120.
[22] D. Zhao, C. Shou, T. Malik, I. Raicu, Distributed data provenance for large-scale data-intensive computing, in: IEEE International Conference on Cluster Computing, IEEE CLUSTER, Vol. 13, 2013.
[23] T. Li, X. Zhou, K. Brandstatter, D. Zhao, K. Wang, A. Rajendran, Z. Zhang, I. Raicu, ZHT: A light-weight reliable persistent dynamic scalable zero-hop distributed hash table, in: Parallel & Distributed Processing Symposium (IPDPS), 2013.
[24] A. Uta, A. Sandu, T. Kielmann, MemFS: an in-memory runtime file system with symmetrical data distribution, in: IEEE Cluster, 2014, pp. 272–273, (poster paper).
[25] A. Uta, A. Sandu, I. Morozan, T. Kielmann, In-memory runtime file systems for many-task computing, in: Adaptive Resource Management and Scheduling for Cloud Computing, Springer, 2014, pp. 3–5.
[26] F. Pop, M. Potop-Butucaru, Adaptive resource management and scheduling for cloud computing, Springer, 2014.
[27] B. Fitzpatrick, Distributed caching with memcached, Linux journal 2004 (124) (2004) 5.
[28] B. Aker, Libmemcached, http://libmemcached.org/libMemcached.html (2014).
[29] D. Karger, E. Lehman, T. Leighton, R. Panigrahy, M. Levine, D. Lewin, Consistent hashing and random trees: Distributed caching protocols for relieving hot spots on the world wide web, in: Proceedings of the twenty-ninth annual ACM symposium on Theory of computing, ACM, 1997, pp. 654–663.
[30] M. Szeredi, et al., FUSE: Filesystem in userspace, http://fuse.sourceforge.net/.
[31] DAS-4, The Distributed ASCI Supercomputer, http://www.cs.vu.nl/das4/ (2014).
[32] Amazon Elastic Compute Cloud, http://aws.amazon.com/ec2 (2014).
[33] iperf, https://iperf.fr (2014).
[34] Z. Zhang, D. S. Katz, M. Wilde, J. M. Wozniak, I. Foster, MTC envelope: defining the capability of large scale computers in the context of parallel scripting applications, in: Proceedings of the 22nd international symposium on High-performance parallel and distributed computing, ACM, 2013, pp. 37–48.
[35] W. D. Norcott, D. Capps, Iozone filesystem benchmark, URL: www. iozone. org 55.
[36] mdtest, http://sourceforge.net/projects/mdtest/ (2014).
[37] G. Juve, E. Deelman, K. Vahi, G. Mehta, B. Berriman, B. P. Berman, P. Maechling, Scientific workflow applications on amazon ec2, in: E-Science Workshops, 2009 5th IEEE International Conference on, IEEE, 2009, pp. 59–66.
[38] Z. Zhang, D. S. Katz, J. M. Wozniak, A. Espinosa, I. Foster, Design and analysis of data management in scalable parallel scripting, in: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, IEEE Computer Society Press, 2012, p. 85.