IMSS: In-Memory Storage System for Data Intensive Applications

Javier Garcia-Blas(✉), David E. Singh, and Jesus Carretero

University Carlos III of Madrid, Leganes, Spain
{fjblas,dexposit,jcarrete}@inf.uc3m.es

Abstract. Computer applications are growing in terms of data management requirements. In both scientific and engineering domains, high-performance computing clusters tend to experience bottlenecks in the I/O layer, limiting the scalability of data-intensive based applications. Thus, minimizing the number of cycles required by I/O operations constitutes a widely addressed challenge. In order to cope with that constraint, distributed in-memory store solutions provide a network-attached storage system using the compute nodes main memory as storage device. This solution provides a temporary but faster storage approach than those based on non-volatile memory like SSDs. This work presents a novel ad-hoc in-memory storage system focused on data management and data distribution, namely IMSS. Our solution accelerates both data and metadata management, taking advantage of ZeroMQ, a fast and flexible communication mechanism. One of the main contributions of IMSS is that it incorporates multiple distribution policies for both optimizing network performance and increasing load-balance. The experimental evaluation demonstrates that our proposal outperforms Redis, a well-known in-memory data structure store, outperforming Redis in both write and read data accesses.

Keywords: HPC · Data intensive · In-memory storage

1 Introduction

Current scientific and engineering applications running on today's large-scale supercomputers are usually characterized by a data-intensive nature. A single

This work was partially supported by the EU project "ASPIDE: Exascale Programming Models for Extreme Data Processing" under grant 801091. This work has been partially funded by the European Union's Horizon 2020 under the ADMIRE project, grant Agreement number 956748-ADMIRE-H2020-JTI-EuroHPC-2019-1. This research was partially supported by Madrid regional Government (Spain) under the grant "Convergencia Big Data-HPC: de los sensores a las Aplicaciones. (CABAHLA-CM)". Finally, this work was partially supported by the Spanish Ministry of Science and Innovation Project "New Data Intensive Computing Methods for High-End and Edge Computing Platforms (DECIDE)" Ref. PID2019-107858GB-I00.

H. Anzt et al. (Eds.): ISC High Performance 2022 Workshops, LNCS 13387, pp. 190–205, 2022.
https://doi.org/10.1007/978-3-031-23220-6_13

application's workflow easily generates tens of terabytes of data, mostly produced by on-line operations. As M. Radulovic et al. [14] stated, from the performance point of view, that a set of tested applications behave as data intensive ones when all of them, but two, spent a significant portion of time with a memory bandwidth utilization above 60% or even 80%. Due to the appearance of these data-demanding high-performance applications, multiple software solutions have been introduced in an attempt to cope with challenges along the entire I/O software stack [6], such as high-level I/O libraries, parallel file systems, and I/O middleware, with a final objective consisting on reducing the amount of file system calls and offloading I/O functionalities from compute nodes, respectively. Those optimizations are even more important for data-intensive workflows, consisting of interdependent data processing tasks often connected in a DAG-style sequence, which communicate through intermediate storage abstractions, typically files. While workflow management systems deployed on HPC systems (e.g., parallel machines) typically exploit a monolithic parallel file system that ensures a high efficiency in data access [18], workflow systems implemented on distributed infrastructures (most often, a public Cloud) must borrow techniques from the Big Data computing field [7].

For several years, I/O-intensive HPC-based applications have been primarily based on distributed object-based file systems, which separate data from metadata management and allow each client to communicate in parallel directly with multiple storage servers. Exascale I/O raises the throughput and storage capacity requirements by several orders of magnitude. Therefore, to develop methods that can manage the network and storage resources accordingly is a must [12]. It is assumed that the systems already developed for data analytics are not directly applicable to HPC due to the fine-granularity I/O involved in scientific applications. Another weakness of existing systems is the semantic gap between the application requests and the way they are managed by the storage back-end at the block level.

Addressing the challenge, different solutions have been implemented throughout the years. Alluxio [9] conforms a storage solution located between computation frameworks and persistent data stores that aims to reduce the complexity of storage APIs while taking advantage of memory speed I/O. However, the former does not provide an application-dedicated ad-hoc storage facility. Approaching another viewpoint, Hermes [11] focuses on the implementation of a MRAM-based storage system improving file system performance through the effective use of MRAM devices. Nevertheless, it does not provide locality policies. Also, solutions, such as WekaIO[1], that provide a high-performance storage architecture, do not consider locality within the implementation neither ad-hoc storage characteristics.

This work presents the design, implementation, and evaluation of a distributed ad-hoc in-memory storage system (IMSS), a proposal to enhance I/O in both traditional HPC and High-Performance Data Analytics (HPDA) systems. The architectural design follows a client-server design model where the

[1] https://www.weka.io.

client itself will be responsible of the server entities deployment. We propose an application-attached deployment constrained to application's nodes and an application-detached considering offshore nodes. The client layer is in charge of dealing with data locality exploitation alongside the implementation of multiple I/O patterns providing diverse data distribution policies.

Our approach offers the following benefits. First, the storage facility provides a flexible API tackling the storage servers' elastic deployment. It is possible to specify the number of servers to be deployed as well as the compute nodes where those servers will execute. Each server will have a storage buffer, whose size is specified at server creation. Second, IMSS makes use of main memory as the storage device so as to reduce as much as possible response time within requests, avoiding querying data from disk. IMSS provides multiple data distribution policies, which consider data scattering among storage processes and adapts the distribution behavior to each application's use case. Finally, IMSS exposes a non-POSIX interface so as to cope with the semantic gap existing in current high-performance I/O systems. The interface provided relies on *get-set* functions that enable non-contiguous data-related operations, unlike the traditional POSIX interface.

The rest of the paper is structured as follows. Section 2 presents related work to our research. Section 3 introduces the architectural design of the IMSS system. Section 4 introduces the deployment options of IMSS. In Sect. 5, we discuss the experimental evaluation results. Finally, Sect. 6 closes the paper with the main conclusions from our work.

2 Related Work

General-purpose parallel file systems such as GPFS [16] and Lustre [2] have been providing for a long time well-known solutions for long term persistent storage. However, they are very rigid and cannot be modified or suited to an application one they are deployed. To avoid this problem, new distributed storage architectures, like CEPH, have been proposed. CEPH storage system provides a distributed architecture that can be deployed on virtual systems, allowing block- and file-level storage, replication, and custom storage backends in Distributed Storage Systems [1]. However, current HPC systems and applications are not well suited to that kind of systems.

Moreover, increasing the complexity of the I/O stack with traditional I/O devices, generates an increasing in I/O operations latency that hampers applications' performance. Thus, nowadays use cases have empowered the proliferation of low-latency storage systems using local or remote in-memory storage devices as a feasible approach to the problem [10,23]. Such has been the impact of these storage systems [24] that multiple solutions, such as in-memory relational databases, in-memory NoSQL databases, in-memory cache systems, and in-memory data processing systems, have been implemented in the last years.

Considering the widespread spectrum of solutions, Redis [15] is a well-known key-value in-memory store that offers storage support for multiple data struc-

tures. Redis' implementation employs a single thread in charge of both, I/O communications and data storage/retrieval operations. Redis provides a distributed version called Redis Cluster, which provides an absolute decentralization through a hash slot partition strategy to find out which server within the deployment will store a certain record. Nevertheless, our tool has significant enhancements over Redis. First of all, the IMSS storage system follows a multi-threaded design architecture. Secondly, our IMSS provides to the applications a set of distribution policies that can be chosen at dataset level. As a result, IMSS will increase awareness in terms of data distribution at the client side, providing benefits such as a better data locality exploitation.

Another alternative that has been explored in order to approach the data challenge is ad-hoc file systems [3]. Ad-hoc file systems provide a custom data resource at application level, taking advantage of internal storage devices while acting as a middleware between persistent storage entities and the application itself. Major features are: (i) negligible deployment overhead, to be deployed either on a HPC cluster for lifetimes as small as the runtime of a single job; (ii) global name space for all nodes linked to the same ad-hoc file system; and (iii) interaction with the back-end storage system through data staging.

Within the current state-of-the-art ad-hoc file systems, GekkoFS [19] conforms an exemplary implementation of an ad-hoc file system which offers a user-space file system that combines application's node-local persistent storage devices in order to provide a global name space within the context of a particular use case, such as an HPC job, distribution of data and metadata as evenly as possible among the nodes conforming the file system instance by using hash indexing to discover which server will be storing each data element. GekkoFS relaxes the POSIX semantics and relies on the application in order to ensure that data overlapping conflicts do not arise. Therefore, the main differences considering GekkoFS and IMSS involves data distribution strategies and storage resources. On the one hand, IMSS enables multiple data distribution policies at dataset level increasing the application's awareness about the location of the data itself. On the other hand, IMSS uses main memory so as to store records and also the possibility of persistent storage.

BurstFS [20] constitutes a burst-oriented storage system that shares basic design considerations with GekkoFS. The main difference between them involves write operations: BurstFS clients always write to the corresponding local storage in a log-type manner. BurstFS instances are dynamically deployed along with the allocation of a job over a set of compute nodes. Then, the storage system will be using whatever node-local burst buffers are available, which may consist of SSDs or any other fast storage device. Moreover, BurstFS uses the key-value data model in order to handle metadata. In this case, the distribution policies enabled by our IMSS arise as an advantage against the BurstFS system. The IMSS client will be able to write to local/internal storage devices and to distribute the same workload among the set of servers conforming the storage entity by means of different data distribution strategies achieving improved load-balance strategies respect to BurstFS. BurstFS system makes use of persistent storage devices

while IMSS store makes use primarily of main memory resources. As a result, the benefits of the data-locality exploitation will be achieved more easily using the IMSS tool.

In a previous work, we presented Hercules [4], a hierarchical parallel storage system based on distributed memory. IMMS differs in the following aspects. First, Hercules was based on Memcached [13] for both front and back-end layers. This approach suffers from the limitation of the Memchached protocol for data transferring modes, such as inter-process communication and inter-thread communication. IMSS employs its own communication protocol based on ZeroMQ, offering more flexible communication patterns. In contrast to Hercules, IMSS provides its own in-memory storage back-end. This alternative outperforms Memcached by eliminating the global cache lock system [22]. IMSS offers an ad-hoc oriented deployment, which facilitates the integration of IMSS in both applications and systems. Finally, IMSS offers a scalable metadata management layer that exploits data locality in large supercomputers.

3 IMSS Architecture Design

As, shown in Fig. 1, the architectural design of IMSS follows a client-server design model where the client itself will be responsible of the server entities deployment. We propose an application-attached deployment constrained to application's nodes and an application-detached considering offshore nodes.

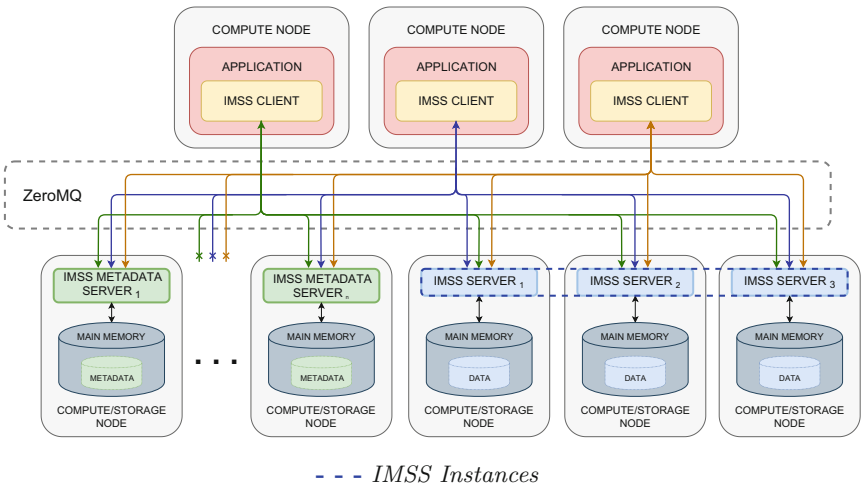

Fig. 1. Representation of an IMSS deployment.

The development of the present work was strictly conditioned by a set of well-defined objectives. Firstly, IMSS should provide flexibility in terms of deployment. To achieve this, the IMSS API provides a set of deployment methods where

the number of servers conforming the instance, as well as their locations, buffer sizes, and their coupled or decoupled nature, can be specified. Second, parallelism should be maximized. To achieve this, IMSS follows a multi-threaded design architecture. Each server conforming an instance counts with a dispatcher thread and a pool of worker threads. The dispatcher thread distributes the incoming workload between the worker threads with the aim of balancing the workload in a multi-threaded scenario. Main entities conforming the architectural design are IMSS clients (front-end), IMSS server (back-end), and IMSS metadata server. Addressing the interaction between these components, the IMSS client will exclusively communicate with the IMSS metadata server whenever a metadata-related operation is performed, such as: *create_dataset* and *open_imss*. Data-related operations (*get_data* & *set_data*) will be handled directly by the corresponding storage server. Finally, IMSS offers to the application a set of distribution policies at dataset level increasing the application's awareness about the location of the data. As a result, the storage system will increase awareness in terms of data distribution at the client side, providing benefits such as data locality exploitation and load balancing.

Two of the most suitable network interfaces are sockets and Remote Procedure Calls (RPC). To choose the best one, we made a comparison between several communication mechanisms (sockets, gRPC, and we chose ZeroMQ [5] in order to handle communications between the different entities conforming an IMSS instance[2]. ZeroMQ has been qualified as one of the most efficient libraries for creating distributed applications [8]. ZeroMQ provides multiple communication patterns across various transport layers, such as inter-threaded, inter-process, TCP, UDP, and multicast. ZeroMQ provides a performance-friendly API with an asynchronous I/O model that promotes scalability. In addition, ZeroMQ library offers zero-copy messages, avoiding further overheads due to data displacements.

Furthermore, to deal with the IMSS dynamic nature, a distributed metadata server, resembling CEPH model [21], was included in the design step. The metadata server is in charge of storing the structures representing each IMSS and dataset instances. Consequently, clients are able to join an already created IMSS as well as accessing an existing dataset among other operations.

3.1 Front-End Layer

The client application will handle IMSS and dataset instances through an IMSS client library. The API provides a set of operations to *create*, *join*, *get*, *set*, and *release* data, datasets, and IMSS instances.

Along any session, clients create and join multiple *IMSS instances*. An IMSS instance is defined as an ephemeral dedicated storage entity conformed by multiple servers distributed along a set of user-defined machines that use main memory in order to store datasets. An IMSS instance is identified by a unique *Uniform Resource Identifier* (URI) and it is represented by a data structure

[2] (https://gitlab.arcos.inf.uc3m.es/mandres/imss/blob/master/Middleware_Comparison.pdf).

storing parameters such as the number of servers conforming the instance and their respective location. Moreover, a dataset entity corresponds to a collection of data elements with a constant size that are distributed among the storage servers of a single IMSS instance following a certain data distribution policy. As IMSS instances, datasets are identified by a unique URI, which reflects the storing IMSS entity. A data structure representing the dataset abstraction is created per instance, gathering parameters such as the distribution policy assigned to the dataset, the number of data elements conforming the dataset, and the replication factor, among others.

3.2 Back-End Layer

Each IMSS instance is formed by multiple IMSS storage servers. Each one stores multiple data blocks of different datasets. Each IMSS server deploys a *dispatcher* thread that distributes and balances client connection requests among worker threads following a round-robin policy. In addition, worker threads belonging to the same server associate data blocks' identifiers to memory locations in a map-based memory container.

In order to handle *get* and *set* requests, each worker thread exclusively accesses the map container for the provided data block location. Afterwards, the requested data block is wrapped into a message and is sent back to the client in case of a *get* operation. If the requested data block is not found, an error code is returned. If the operation is a *set*, the worker thread overwrites the concerned block if it was already stored. Otherwise, the data block is written and a new key-value pair representing the previous block is added to the map.

Data persistency is provided through period dump operations that write all the buckets of an IMSS to SSD or hard disks. The period can be defined when the IMSS is created.

3.3 IMSS Metadata Server

Dataset and IMSS data structures appear whenever the client creates one of the previous instances. The metadata server was introduced in order to keep track of the aforementioned structures. In terms of internal design, IMSS metadata server aims to balance workload among a thread pool. The architecture consists of a single *dispatcher* thread and multiple *worker* threads. The dispatcher thread serves incoming connection requests distributing new clients between the worker threads following a round robin policy. A map container, which associates datasets' and IMSS instances' URIs to a memory location, is used to keep track of the stored structures.

The metadata server implements a persistence module. The server is able to write the structures associated to the dataset and IMSS entities handled along the session once it is over, as well as reading them during the deployment of a new session.

3.4 Data Distribution Policies

Dataset distribution policies included in IMSS define the distribution of each dataset in the instance deployed. The policy determines the back-end server in charge of storing data blocks. The IMSS front-end layer handles the policy assignment whenever a dataset is created. The IMSS metadata server maintains the dataset's data structure, annotating the distribution policies of each one. The following policies have been developed:

- **ROUND_ROBIN**: data blocks are distributed among the IMSS servers following a round-robin strategy.
- **BUCKETS**: each dataset is divided into the same number of chunks as number of servers. Each chunk is composed by a consecutive number of data blocks, equally distributed. Then, each chunk is assigned to a unique server.
- **HASHED**: a hash operation is applied over each data block key to discover the mapped server.
- **CRC16bits** & **CRC64bits**: similar to HASHED policy, but a sixteen/sixty four bits CRC operation is applied over the data block key.
- **LOCAL**: each data block is handled by the IMSS server running in the same node that the client. The data block key is not considered in this policy. If no IMSS server was deployed in the client node, every dataset's data-related operation will return an error.

With those policies, IMSS enables the possibility to tune the dataset distribution. These distribution policies aim to increase performance. As demonstrated in Sect. 5, the LOCAL policy experimentally obtains the greatest performance due to the exploitation of locality. In the current prototype, the distribution policy is established at creation time and it cannot be modified. In the future, we plan implement a dynamic distribution policy that enables to adapt the behavior in terms of system metrics (CPU, memory consumption, etc.). Within the previous possibilities, a *LOCAL* policy should be highlighted as it will have the objective of exploiting data locality as much as possible: data requests will be forwarded to the storage server running in the same machine where the request was made. Finally, a non-POSIX *get-set* interface will be provided in order to manage *datasets*, which conform a storage abstraction used by IMSS instances in order to manage data blocks (smallest data unit considered within the storage system).

4 Deployment Strategies

Two strategies were considered so as to adapt the storage system to the application's requirements. On the one hand, the *application-detached* strategy, consisting of deploying IMSS clients and servers as process entities on decoupled nodes. IMSS clients will be deployed in the same computing nodes as the application, using them to take advantage of all available computing resources within an HPC cluster, while IMSS servers will be in charge of storing the application

datasets and enabling the storage's execution in application's offshore nodes. In this strategy, IMSS clients do not store data locally, as this deployment was thought to provide an application-detached possibility. In this way, persistent IMSS storage servers could be created by the system and would be executed longer than a specific application, so as to avoid additional storage initialization overheads in execution time. Figure 2 (left) illustrates the topology of an IMSS application-detached deployment over a set of compute and/or storage nodes where the IMSS instance does not belong to the application context nor its nodes.

On the other hand, the *application-attached* deployment strategy seeks empowering locality exploitation constraining deployment possibilities to the set of nodes where the application is running, so that each application node will also include an IMSS client and an IMSS server, deployed as a thread within the application. Consequently, data could be forced to be sent and retrieved from the same node, thus maximizing locality possibilities for data. In this approach each process conforming the application will invoke a method initializing certain in-memory store resources preparing for future deployments. However, as the attached deployment executes in the applications machine, the amount of memory used by the storage system turns into a matter of concern. Considering that unexpectedly bigger memory buffers may harm the applications performance, we took the decision of letting the application determine the memory space that a set of servers (storage and metadata) executing in the same machine shall use through a parameter in the previous method. This decision was made because the final user is the only one conscious about the execution environment as well as the applications memory requirements. Flexibility aside, as main memory will be used as storage device, an in-memory store will be implemented so as to achieve faster data-related request management. Figure 2(right) displays the topology of an IMSS application-attached deployment where the IMSS instance is contained within the application.

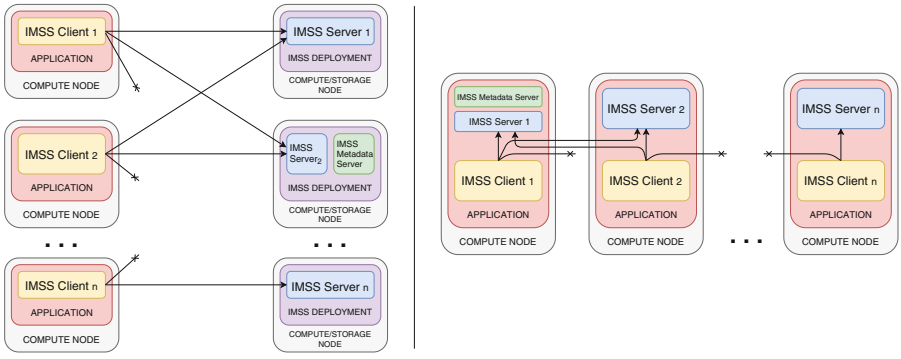

Fig. 2. IMSS application-detached deployment (left side) vs IMSS application-attached deployment (right side)

5 Experimental Evaluation

An IMSS prototype, as well as micro-benchmarks used for evaluation are available[3]. IMSS has been evaluated in different scenarios in order to ascertain the tool's appropriateness for the task addressed in the current work. First, we evaluated IMSS in a bare-metal cluster. Second, in order to evaluate the scalability of IMSS, we carried out experiments using the Google Cloud infrastructure. The former evaluations aimed to measure the system's scalability reaching up to 128 nodes. The workload was fixed to a single dataset of 8 GB that clients handle collectively: one client creates it and the remaining open it. Multiple distribution policies and block sizes were once again considered.

The Google Cloud Platform[4] was chosen as a feasible solution. The virtual instances are composed by nodes with 4 cores and 16 GB of RAM memory. We have employed up to 128 virtual nodes. The software layer is based on Ubuntu 18.10 LTS, GCC compiler 7.3.0, and MPICH 3.2.0. The results shown in the experiments correspond with the average value of five consecutive executions. In order to depict the performance of IMSS, we have compared our solution with four storage alternatives. First, IMSS was directly compared with the Redis object store. Second, IMSS was compared with maximum network bandwidth (in terms of MB/s), denoted in the plots with the label *network_limit*. The network throughput was obtained by using the *iperf* tool [17]. Within the results presented, Redis deployment time is not considered. However, it is important to note that the deployment step of IMSS is significantly smaller than Redis.

5.1 Block Size Variation

The first scenario presents the aggregated performance obtained from writing and further reading steps of an 8 GB dataset achieved by 128 clients. Figure 3 represents the aggregated throughput obtained from the previous experiments. In this case, the lack of variation of any kind is differentiated. In the first place, there is no performance increment with bigger block sizes. This takes place as the dataset's portion left to each client is so small (64 MB) that it does not leave possibility for improvement. There is no difference in writing such a small number of bytes with a block size of 4 KB (16384 blocks) or 16 MB (4 blocks) taking into account the asynchronous nature of the operation. Secondly, the previous condition plus the minimal number of write operations per compute node leaves no chance for any *LOCAL* policy improvement. Besides, another factor that locates the observed performance under the referenced corresponds to the reduction in the number of write operations per client and create dataset call.

Moreover, Fig. 4 shows the aggregated throughput obtained from the consequent reading step. In this case, a significant improvement paired to the block size is ascertained. The previous fact takes place as the read operations involve

[3] https://gitlab.arcos.inf.uc3m.es/mandres/IMSS.
[4] (https://cloud.google.com).

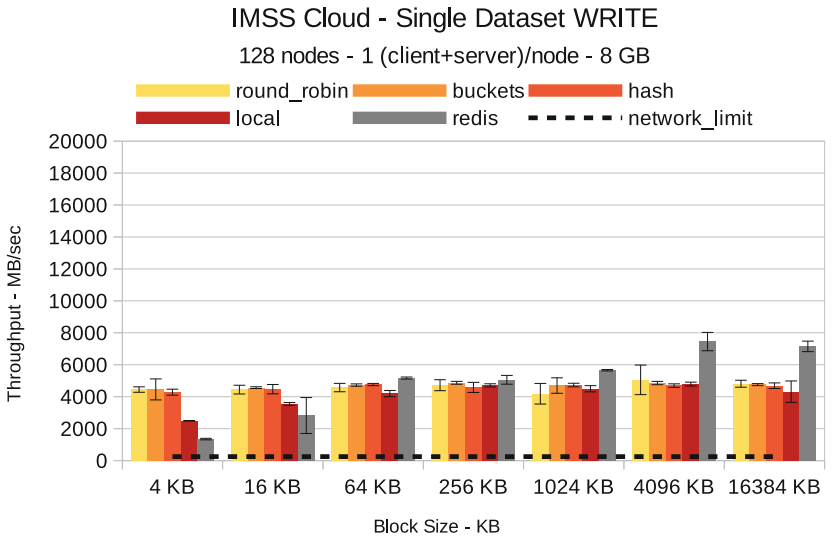

Fig. 3. Single dataset WRITE evaluation.

network overheads, which suffer from small blocks. In addition, a performance improvement could be noticed regarding the *LOCAL* distribution. The policy turns to be favored as each client's requests are handled by the server running in the same machine. As a whole, the *LOCAL* policy's effectiveness is once again justified through the results obtained in the reading step. In addition, the metadata influence is ascertained once more regarding the writing one.

5.2 Scalability

This scenario considers a scalability evaluation of the IMSS starting from 4 nodes up to 128 by writing and reading an 8 GB dataset collectivelly using a 16 MB block size. Figure 5 plots the performance obtained in the writing step. As it can be seen, the performance degradation detailed in the writing step explanation of Sect. 5.1 turns to be justified. The number of metadata operations also increases reaching a point where the number of clients is no longer an advantage, but a constraint. In case of IMSS, each execution involved both the initialization of the storage instances and the creation of datasets. We observe that as we increase the number of clients, IMSS suffers from a little metadata overhead due to the management of blocks and the distribution policies. In contrast, Redis does not suffer this constrains as it lacks those features. In addition, the asynchronous nature of the write operations is again considered as it justifies the lack of any *LOCAL* case improvement.

Reading results are shown in Fig. 6. In this case, the 4 and 8 clients cases stick out as the *BUCKETS* policy is able to reach the performance of the *LOCAL* policy. Consequently, considering the previous context and the *BUCKETS* distribution policy, the network limits the performance of the *LOCAL* policy's

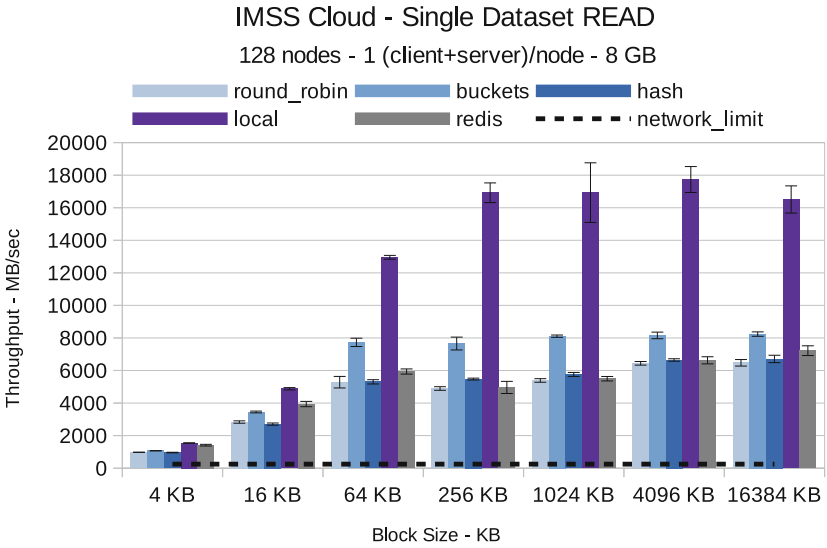

Fig. 4. Single dataset READ evaluation.

performance until the number of clients and the network topology constraint the case. Regarding the obtained results, the performance improvement of the *LOCAL* distribution policy through the exhaustive and aware exploitation of data locality are once again justified as well as the influence performed by the metadata operations.

5.3 Metadata Overhead

The last scenario evaluates the time required for invoking each API call of IMSS.

Figure 7 plots the mean time required in milliseconds to perform every metadata-related call. As shown in the figure, INIT operations are more computationally expensive as they have to create all IMSS environment. Thus, *stat_init* method produces an execution overhead as it initially creates the communication channel with the metadata server. Besides, this call involves the initialization of multiple internals required for an execution. We also observe that both *init_imss* and *open_imss* invocations constitute another couple of computationally expensive functions as they involve creating the corresponding communication channels with each server conforming the IMSS deployment. However, the *init_imss* execution time is above the *open_imss* one as the first function will initialize all server entities. It is important to highlight that the number of servers conforming the IMSS deployment significantly influences the execution time of the aforementioned functions as the number of servers to be awakened and the number of communications to be created increases with it. It is important to notice that those operations are executed only once at IMSS creation or opening.

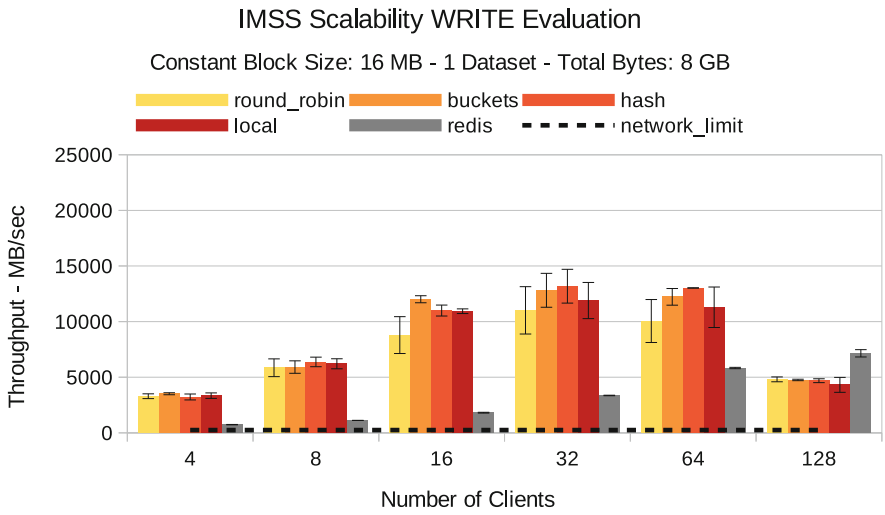

Fig. 5. Scalability evaluation. WRITE step of an 8 GB dataset.

However, status and operation calls creates a very low overhead. The *stat_release* and *release_imss* calls require a small execution time as they exclusively release the aforementioned communication resources and internals. Anyway, they are used only once, when the IMSS is destroyed. Moreover, *stat_imss* function, that requests an IMSS metadata structure to the metadata server, also creates a small execution overhead. Considering the set of metadata operations, *create_dataset*, *open_dataset* and *stat_dataset* methods, no significant execution overhead is created as they exclusively involve a request-reply dialogue with the metadata server, plus additional queries performed over the internal vectors storing datasets' metadata structures. Again, the *release_dataset* function will not suppose a significant overhead as it will just mark as free the position storing the involved metadata structure within the internal entity that keeps track of them.

As may be seen, the overhead of dataset operations is almost negligible. Those results are possible due to the usage of a metadata cluster, with a minimum of 3 nodes. The cluster could be enlarged, if needed, to ensure scalability and to keep performance.

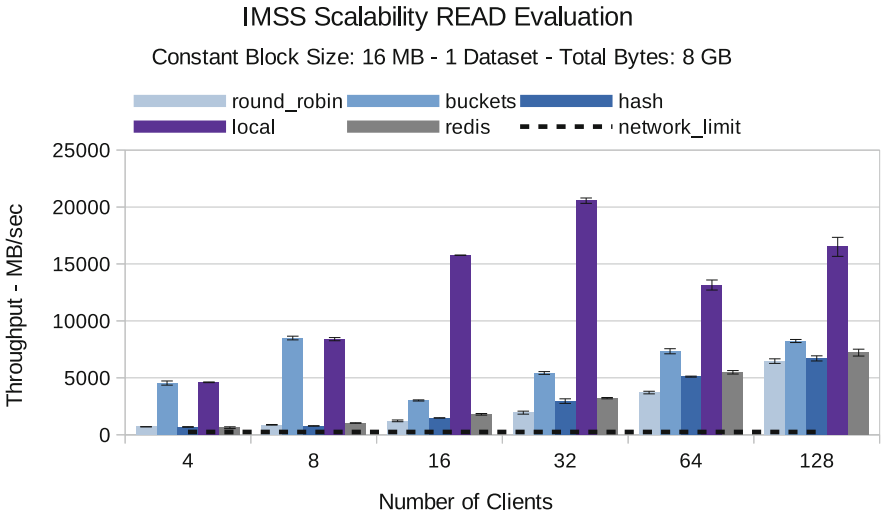

Fig. 6. Scalability evaluation. READ step of an 8 GB dataset.

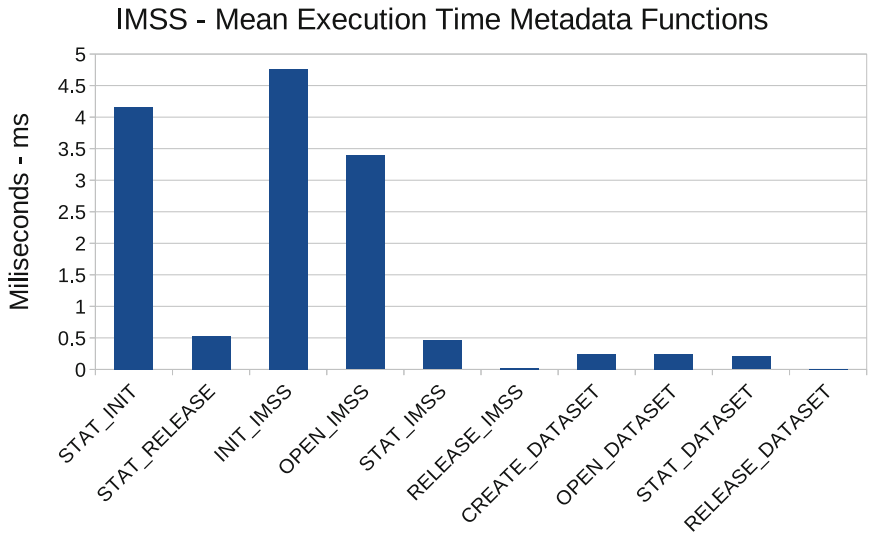

Fig. 7. Mean execution time of each IMSS API call.

6 Conclusions

In this work, we have introduced IMSS, an in-memory ad-hoc storage system for data intensive-based applications that provides a flexible API tackling the storage servers' elastic deployment, usage of main memory as the storage device so as to reduce as much as possible response time within requests, multiple data distribution policies at the dataset level to increased awareness at application

level, and a non-POSIX interface that relies on *get-set* functions. Evaluation results presented in Sect. 5, comparing IMSS, Redis and a POSIX-compliant ext4 file system with caching techniques under different scenarios, show that our IMSS performs better in any operation involving distributed datasets, outperforming Redis and POSIX file systems. Moreover, we showed a low overhead for the execution of IMSS's API operations.

Future work guidelines involve the development of a more sophisticated persistence storage module. This new module will allow to provide more efficient operations to dump data from IMSS to the persistent storage back-end. We are currently working on a extended evaluation that covers an experimental evaluation under larger scenarios in terms of number of clients involved and workload in order to provide a more detailed performance analysis.

References

1. Aghayev, A., Weil, S., Kuchnik, M., Nelson, M., Ganger, G.R., Amvrosiadis, G.: The case for custom storage backends in distributed storage systems. ACM Trans. Storage (TOS) **16**(2), 1–31 (2020)
2. Braam, P.J., Schwan, P.: Lustre: the intergalactic file system. In: Ottawa Linux Symposium, vol. 8, pp. 3429–3441 (2002)
3. Brinkmann, A., et al.: Ad hoc file systems for high-performance computing. J. Comput. Sci. Technol. **35**(1), 4–26 (2020). https://doi.org/10.1007/s11390-020-9801-1
4. Duro, F.R., Blas, J.G., Carretero, J.: A hierarchical parallel storage system based on distributed memory for large scale systems. In: Proceedings of the 20th European MPI Users' Group Meeting, pp. 139–140 (2013)
5. Hintjens, P.: Zeromq: an open-source universal messaging library (2007). https://zeromq.org
6. Isaila, F., Garcia, J., Carretero, J., Ross, R., Kimpe, D.: Making the case for reforming the I/O software stack of extreme-scale systems. Adv. Eng. Softw. **111**, 26–31 (2017). https://doi.org/10.1016/j.advengsoft.2016.07.003, http://www.sciencedirect.com/science/article/pii/S0965997816301740, advances in High Performance Computing: on the path to Exascale software
7. Kune, R., Konugurthi, P.K., Agarwal, A., Chillarige, R.R., Buyya, R.: The anatomy of big data computing. Softw. Pract. Exp. **46**(1), 79–105 (2016)
8. Lauener, J., Sliwinski, W.: How to design & implement a modern communication middleware based on ZeroMQ. In: 16th International Conference on Accelerator and Large Experimental Physics Control Systems, p. MOBPL05 (2018). https://doi.org/10.18429/JACoW-ICALEPCS2017-MOBPL05
9. Li, H.: Alluxio: A virtual distributed file system. Ph.D. thesis, UC Berkeley (2018)
10. Lu, Y., Shu, J., Chen, Y., Li, T.: Octopus: an rdma-enabled distributed persistent memory file system. In: 2017 {USENIX} Annual Technical Conference ({USENIX}{ATC} 2017), pp. 773–785 (2017)
11. Miller, E.L., Brandt, S.A., Long, D.D.: Hermes: high-performance reliable mram-enabled storage. In: Proceedings Eighth Workshop on Hot Topics in Operating Systems, pp. 95–99. IEEE (2001)
12. Narasimhamurthy, S., et al.: Sage: percipient storage for exascale data centric computing. Parallel Comput. **83**, 22–33 (2019)

13. Nishtala, R., et al.: Scaling memcache at facebook. In: Presented as part of the 10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 2013), pp. 385–398 (2013)
14. Radulovic, M., Asifuzzaman, K., Carpenter, P., Radojković, P., Ayguadé, E.: HPC benchmarking: scaling right and looking beyond the average. In: Aldinucci, M., Padovani, L., Torquati, M. (eds.) Euro-Par 2018. LNCS, vol. 11014, pp. 135–146. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-96983-1_10
15. Sanfilippo, S., Noordhuis., P.: Redis (2009). https://redis.io
16. Schmuck, F.B., Haskin, R.L.: GPFS: a shared-disk file system for large computing clusters. In: FAST, vol. 2 (2002)
17. Tirumala, A.: Iperf: the TCP/UDP bandwidth measurement tool (1999). http://dast.nlanr.net/Projects/Iperf/
18. Vahi, K., Rynge, M., Juve, G., Mayani, R., Deelman, E.: Rethinking data management for big data scientific workflows. In: 2013 IEEE International Conference on Big Data, pp. 27–35. IEEE (2013)
19. Vef, M., et al.: Gekkofs - a temporary distributed file system for hpc applications. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER), pp. 319–324 (2018)
20. Wang, T., Mohror, K., Moody, A., Sato, K., Yu, W.: An ephemeral burst-buffer file system for scientific applications. In: SC 2016: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 807–818 (2016)
21. Weil, S.A., Brandt, S.A., Miller, E.L., Maltzahn, C.: Crush: controlled, scalable, decentralized placement of replicated data. In: SC 2006: Proceedings of the 2006 ACM/IEEE Conference on Supercomputing, pp. 31–31. IEEE (2006)
22. Wiggins, A., Langston, J.: Enhancing the scalability of memcached. Intel document, unpublished (2012). http://software.intel.com/en-us/articles/enhancing-the-scalability-of-memcached
23. Yang, J., Izraelevitz, J., Swanson, S.: Orion: a distributed file system for non-volatile main memory and rdma-capable networks. In: 17th {USENIX} Conference on File and Storage Technologies ({FAST} 2019), pp. 221–234 (2019)
24. Zhang, H., Chen, G., Ooi, B.C., Tan, K.L., Zhang, M.: In-memory big data management and processing: a survey. IEEE Trans. Knowl. Data Eng. **27**(7), 1920–1948 (2015)