Using On-Demand File Systems in HPC Environments

Mehmet Soysal*, Marco Berghoff*, Thorsten Zirwes*, Marc-André Vef†, Sebastian Oeste‡,
André Brinkmann†, Wolfgang E. Nagel‡, and Achim Streit*
* Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany
† Johannes Gutenberg University Mainz, Mainz, Germany
‡ Technische Universität Dresden, Dresden, Germany

***Abstract*—In modern HPC systems, parallel (distributed) file systems are used to allow fast access from and to the storage infrastructure. However, I/O performance in large-scale HPC systems has failed to keep up with the increase in computational power. As a result, the I/O subsystem which also has to cope with a large number of demanding metadata operations is often the bottleneck of the entire HPC system. In some cases, even a single bad behaving application can be held responsible for slowing down the entire HPC system, disrupting other applications that use the same I/O subsystem. These kinds of situations are likely to become more frequent in the future with larger and more powerful HPC systems.**

In this work, we present a simple solution for applications with very high I/O demands. Our proposed solution is to create a private parallel file system on-demand for an HPC job and use the node-local storage devices, e.g. solid-state-disks (SSD). We show that this feature is easy to add to an existing HPC environment and requires only minimal configuration to the system. We conclude that the impact on running applications is manageable and the advantages to applications that generate a high load outweigh the disadvantages. We show that in some cases applications may run slower, but the reduction of load on the global file system is prevailing in these cases.

***Index Terms*—file system, on-demand, lustre, beegfs**

I. Introduction

Today's HPC systems utilize parallel file systems (PFSs), e.g., Lustre [1], IBM Spectrum Scale (GPFS) [2], or BeeGFS [3], that comply with POSIX semantics as the storage subsystem. In the past [4], research and development on PFSs focused on increasing the bandwidth of read and write operations in a parallel manner.

While the computing resources can often be allocated exclusively, the global PFS is shared by all users of a HPC system. This environment makes it difficult for the user to optimize the application concerning I/O as there is no clear understanding among developers about the impact of certain I/O patterns on the storage system. There are many possible factors a user would have to consider. Influences from the back-end storage device, network interface, storage servers, data distribution, request size, and other applications slowing down the PFS [5]. Besides the lack of optimization of the applications, there are essential factors on the system side. One of the reasons why PFSs struggle with certain I/O operations is that they have to cover a wide range of applications with the most frequent use cases. In addition, the PFS must be robust with high availability as the HPC system is dependent on the global storage system.

Moreover, PFSs are often shared by many users and their jobs. Consequently, badly behaving applications can result in poor performance of the storage subsystem and affecting all users. These applications create either a lot of metadata operations instead of reducing I/O through aggregation or there is no optimization at all regarding to I/O.

In this paper, we present an approach that targets use cases that do not work well with today's generic PFSs by creating a private on-demand file system. The difference of a PFS and a on-demand file system in our context is how these are used and created. By PFS we mean file systems which have dedicated nodes, as they are currently commonly used as the global file system for HPC systems and shared by all users and jobs. On-demand file systems use the storage directly connected to the nodes. The storage is only shared within a job. In our approach these file systems are built for a single job and purged after job completion. So we could also call these on-demand file system – "private" parallel file systems.

The remainder of the paper is structured as follows. First, we depict related work on the previously mentioned topics in Chapter 2. In Chapters 3 and 4, we describe the components of our idea and present benchmarks of how our concept could help with future large-scale systems. Chapter 5 discusses how our approach is expected behave on future systems and wand shows additional considerations compared with today's supercomputers.

II. Related Work

In the recent past there have been many developments and innovations to improve I/O throughput and performance. We cannot cover everything and try to give a brief overview of existing solutions. Although it is difficult to separate the solutions, four categories can be distinguished: file system features, hardware solutions, libraries and dynamic system re-configurations.

a) File system features: File Systems offer interesting new features to improve I/O bottlenecks. GPFS has introduced a Highly Available Write Cache (HAWC) [6]. HAWC uses node-local *solid-state drives* (SSDs) as buffers for the global file system. Lustre has received the Progressive File Layouts (PFL) [7] feature. PFL adjusts dynamically the chunk size and

stripe pattern depending on I/O traffic. BeeGFS offers storage pools [8] and allow to group storage targets in different classes, e.g., one pool with small but fast solid state drives, and another large pool for spinning disks. However, such solution are only available when using the vendor software solution and coupled with the global file system.

b) Hardware solutions: Today's wide spread use of SSDs in compute nodes of HPC systems has provided a new way of accelerating storage. SSDs provide higher throughput and better random I/O performance than spinning hard disks. SSDs have been considered for file system metadata [9] [10], as its meta-data performance is a major bottleneck in HPC environments.

A different kind of hardware solutions are burst buffers, which aim to reduce the load on the global file system [11]. These type of burst buffers can be categorized into two groups [12]: remote-shared and node-local. While remote-shared burst buffers [13], [14] use central, dedicated I/O nodes to forward I/O operations, node-local burst buffers are typically used within compute nodes and may be managed by the PFS (e.g., PLFS [15]). Examples for such node-local burst buffers are BurstFS [12] or BeeOND [16].

c) Libraries: There is a large number of libraries available for improving I/O behavior of an application. High-level libraries, such as HDF5 [17], NETCDF [18] or ADIOS [19], are trying help users to express I/O as data structures and not only as bytes and blocks. Middleware libaries, such as MPI-IO [20], help to improve usage of parallel storage systems, e.g. data sieving and collective I/O [21]. SionLib [22] is another library for task local file I/O, developed by Forschungszentrum Jülich. When files are accessed in parallel, only the open and close functions are collective while writing and reading files can be done asynchronously. These libraries are not in contrast to our approach. The advantages of using such libraries also apply to the on-demand file systems.

d) System reconfiguration: Similar to our approach, the configuration of the system can be modified. There are several basic methods. A Dynamic Remote Scratch [23] implementation was developed to create an on-demand block device and use it with local SSDs as a LVM [24] device. This approach relies on a dedicated storage system. In our concept such a dedicated storage is not needed for temporary storage. Another software based solution is the RAMDISK Storage Accelerator [25]. It introduces a additional cache layer into HPC systems. It uses the available memory of commodity servers to build a PFS using RAMDisks. The PFS constructed on the RAMDISK is offered to the computing nodes as a very high-speed and low-latency temporary storage. While this approach is similar to our idea, we use the local disks of the compute nodes and do not require other commodity servers to build a on-demand file system.

III. Methods

We deploy an on-demand file system on the allocated nodes per job. Therefore, we use BeeGFS as the on-demand parallel file system. BeeGFS is POSIX compliant. It offers a tool BeeOND to deploy the file system on-demand.

A. Integration into an HPC environment

Usually HPC systems use a *batch system*, such as SLURM [26], MOAB [27], or LSF [28]. The batch system manages the resources of the cluster and starts the user jobs on allocated nodes. At the start of the job, a prologue script may be started on one or all allocated nodes and, if necessary, an epilogue script at the end of a job (see Figure 1). These

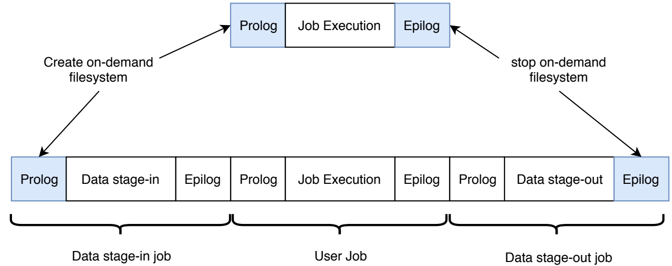


Fig. 1: Job flow for creating an on-demand file system ans staging data.

scripts are used to clean, prepare, or test the full functionality of the nodes. We modified these scripts to start the on-demand file system. During job submission a user can request an on-demand file system for the job.

MOAB offers a staging mechanism [29]. It divides a submitted job into three parts. The node allocation is inherited to the other jobs in the chain. The first part is responsible for the data stage-in and the third for the stage-out (see Fig. 1). We have adapted MOAB's stage-in process so that an on-demand file system is created and the data is staged in. After the user job execution, the data is staged back to the global file system, and the on-demand file system is purged. The HPC system only needs minor modifications in the scripts. The on-demand file system is provided as an additional feature upon request.

B. Benchmark environment

For our evaluation we used the ForHLR II [30] at Karlsruhe Institute for Technology. The system consists of 1152 nodes with two Intel Xeon processors E5-2660 v3 (20 cores). The nodes are partitioned into two island, a small island with 336 and a large island with 816 nodes. The topology is a two-layer CBB-network [31]. The nodes are connected to the fabric via an InfiniBand HCA. 24 nodes are connected to the leaf switches with 56 Gbit/s and from each leaf switch 12 uplinks with 100 Gbit/s are connected to the upper core switches (see Fig. 2). For our test we exclusively used the small island with 336 nodes. Thus, we ensure that no other jobs on this part of the fabric can interfere with our evaluation. Brown et al. [32] has shown that mixing large and small packet communication can increase fabric latency in fat tree topologies. The small island has 6 root and 14 leaf switches. The nodes have a local SATA SSD with approximately 600 MB/s read and 400 MB/s write performance. Lustre is used for the PFS and is connected

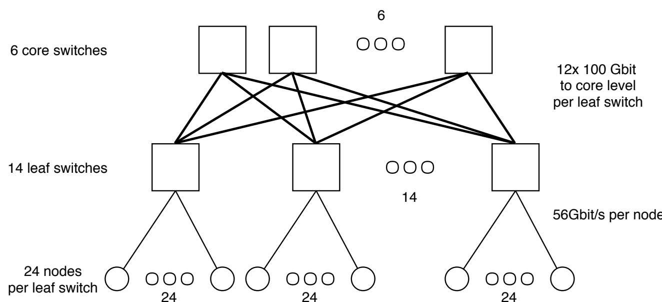


Fig. 2: The small island consists of 6 core switches and 14 leaf switches. Each leaf switch has 24 nodes with 56 Gbit/s and 2x100 Gbit/s links to each core switch.

via InfiniBand. It has a maximum throughput of 50 GB/s, one metadata server (MDS), 10 object storage servers (OSS), and 7 object storage targets (OST) per OSS. For storage, we create an image file on each node and offer it as loopback device to BeeGFS. Cleaning up the nodes can be accomplished by simply removing this image file. We decided to use a loopback device as Yamamoto et al. [33] showed that it can increase performance compared to storing data directly to disk due to Linux's internal caching mechanisms. In addition, a loopback device also helps to manage space consumption on each SSD.

In order to assess the suitability of an on-demand file system in an HPC environment, we have performed several investigations. Besides simple benchmarks to determine the throughput we have chosen two use cases of our users which generate an excessive load on our storage systems as discussed in the next section.

C. Applications and use cases

The first application we investigated is OpenFOAM [34]. OpenFOAM is a toolkit for computational fluid dynamics (CFD) written in C++. It is a widely used open-source fluid dynamics code [35] for engineering applications. Investigation of I/O performance has been performed with two OpenFOAM cases, both using OpenFOAM v1712 and a custom solver developed for detailed combustion simulations [36], [37]:

Use case 1: A production run simulation of an experimentally investigated burner of laboratory scale [38]. The focus of this setup lies on simulating the burner flame in great detail, including all intermediate chemical species that are formed during combustion. Due to the physical complexity of the flame, the computational domain consists of 150 million cells. The simulation is typically run on 5 000–28 000 CPU cores [39]. The challenge in terms of I/O stems from the most efficient I/O strategy OpenFOAM offers for check-pointing: every time the application generates data for check-pointing, each process writes one file per solution variable which may be temperature, pressure or the concentration of chemical species. Due to the large number of chemical species in this setup, generating checkpoint data creates a large number of files. In this work, the case has been run on 240 nodes or 4800 processes, leading to the creation of 95 files per process or half a million files in total, with about 0.2 MB per file or 25 MB per process or 120 GB in total. For previous runs with 28 800 CPU cores, the number of files which are written at the same time increases to 2.7 million.

Use case 2: In this simulation, the inert mixing of methane and air in a pipe leading to the burner from use case 1 is simulated. The computational mesh consists of 150 million cells. The case has been run on 240 nodes or 4 800 processes. Because there is no flame in this setup, the number of solution variables and therefore the I/O requirements for check-pointing are much lower compared to use case 1. The I/O challenge in this case stems from run time post-processing, because this is a precursor simulation to use case 1, solution variables at the opening of the pipe downstream of the burner are written after every time-step, which afterward serve as inlet conditions for the simulation of use case 1. The run time post-processing data, which consists of about 20 small files per time-step, is written at a high frequency during the whole simulation time.

The second application is the NAStJA framework [40]. It is a massively parallel stencil code solver. It is designed for simulations in several scientific domains. Modules for the calculation of the phase-field and the phase-field crystals as used in mesoscopic or atomistic computational material science are available, as well as biological cells can be calculated with the cellular Potts model, a cellular automaton. The framework provides several modules for writing out the results. For our purpose, we configure a single file per block per time-step.

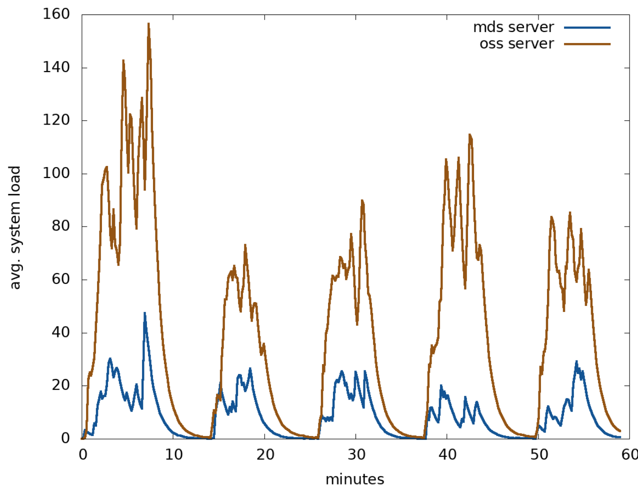
Use case 1: A run with 4800 parallel NAStJA processes is distributed to 240 working nodes with 20 processes each. We choose 4 800 blocks, i.e., one block per core, of the size of 1 MB each and write out every 20th time-step of a total of 100 000 time-steps.

Use case 2: Runs with 23 nodes running NAStJA processes. On the nodes, we reserved up to 4 cores out of the 20 available cores for the BeeGFS processes. Runs with 16, 19 and 20 NAStJA processes per node were performed. One block and thus one file per NAStJA process is used again. We increase the size to 4 MB per process and decrease the write frequency to every 100th time-step.

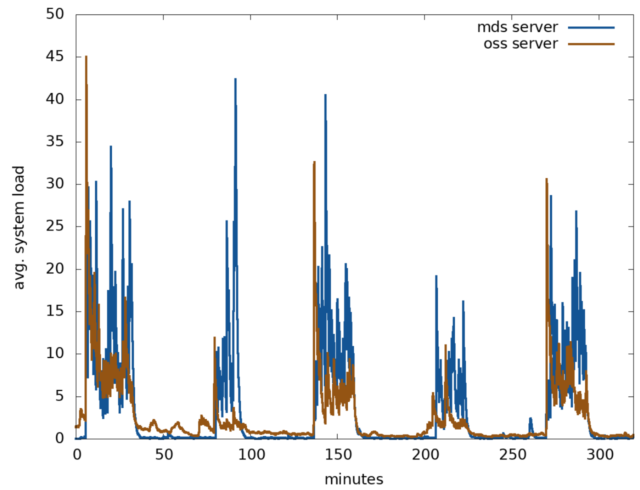
The reason why we have chosen these application and use cases is illustrated in Figs. 3a) and 3b). Here the unix load [41] of the Lustre file system server is depicted when running the use cases. Blue line show the load of the object storage servers and purple line is showing the load of the meta-data server. Both applications were started with five short consecutive runs. Writing alternately on the PFS and on-demand file system. The valleys indicates the periods when the application uses the on-demand file system. It clearly shows what high load these applications generate. Considering that these applications run for hours and create such a load, it's obvious that there's a need for action here.

IV. Results and Discussion

The aforementioned use cases have been performed to evaluate the on-demand file system in an HPC environment. We evaluated our two applications and explain how they



(a) NAStJA runs.



(b) OpenFOAM use case 1.

Fig. 3: Lustre server load during simulation runs. MDS server and average linux load of 10 OSS servers.

perform. Finally, we show how data staging influences the application.

A. Generic benchmark

We use IoZone [42] to measure the maximum throughput of read and write operations. Figure 4 depicts the throughput on the on-demand file system (solid line). The results show that performance increases linearly with the number of nodes. Our observation shows that the SATA-SSDs are the limiting factor. The small throughput variation occurs due to normal performance scattering of the SSDs [43]. The dotted line indicates the theoretical throughput with NVMe-SSD devices. At assumed speeds with $3500\,\mathrm{MB/s}$ read and $2000\,\mathrm{MB/s}$ write performance for common PCIe x4 NVMe-SSD devices [44]. The difference between the SATA-SSD and NVMe-SSD are the connection and communication interface. SATA-SSD are connected via SATA (Serial AT Attachment), as were the rotating hard disks at that time. The computer communicates via an AHCI (Advanced Host Controller Interface) protocol with the memory controller. Nevertheless, for modern SSDs a new way was devloped to reduce this overhead with a communication protocol. With the introduction of NVMe (nonvolatile memory express), the use of controllers was completely dispensed with. NVMe-enabled SSDs are connected directly to the PCIe bus. Compared to SATA, this reduces latencies and raises bandwidth restrictions [45].

We also measured the initialization time of the on-demand file system. Accordingly, we also measured how long it takes to stop it. Table I lists the duration for starting and stopping BeeGFS on-demand. The starting times increase up to 222 seconds on 256 nodes. The included tool BeeOND for starting the BeeGFS on-demand file system has a parallel mode using a parallel distributed shell [46]. At launch there is a sequential region and by parallelizing this piece of code, we were able to reduce startup times to under one minute at 512 nodes.

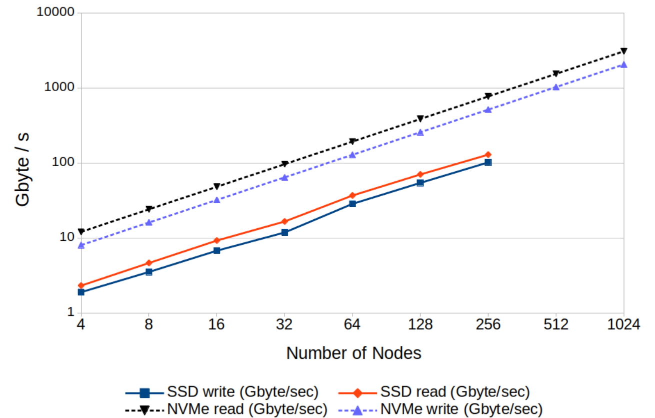


Fig. 4: Solid line: Read/write throughput. Dashed line: extrapolation with the theoretical peak of NVMe-SSDs.

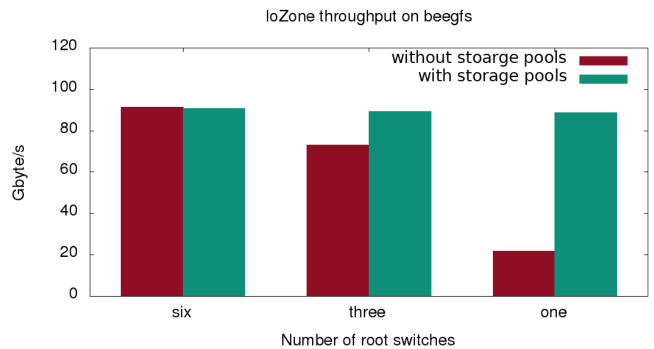


Fig. 5: IoZone write throughput with reduced number of core switches on 240 nodes.

Nodes	8	16	32	64	128	256
Startup (s)	10.2	16.7	29.3	56.5	152.1	222.4
Shutdown (s)	11.9	12.1	9.4	15.9	36.1	81.1

TABLE I: BeeGFS startup and shutdown

In a further test, we investigated the storage pooling feature of BeeGFS [8]. The question here is whether it is possible to deal with bottlenecks in topology with such a feature. We created a storage pool for each leaf switch(see Fig. 2). In other words, when writing to a storage pool, the data is distributed via the stripe count and chunk size, but remains within the storage pool and thus on a switch. Only the communication with the meta data server is forwarded across the core switches. Figure 5 shows the write throughput for three scenarios. Each scenario uses a different number of core switches with six being the full expansion level. In the first experiment, with all six core switches, there is only a minimal performance loss, which indicates a small overhead when using storage pools. In the second case we turned three switches off, and in the last case we turned off five switches. With reduced number of core switches the throughput drops due to the reduced network capacity. If the topology is taken into account, and storage pools are created accordingly, it is possible to achieve the same performance as with the full expansion.

B. Application benchmarks

We evaluated NAStJA and OpenFOAM on 240 nodes with 20 processor cores each. Intermediate results were written to the global and to the on-demand file system. Each test case was run five times for each data point to eliminate possible side effects.

Figure 6 shows the average execution time for the time-steps of NAStJA. Figure 6 a) shows the average time for the time-steps when using the on-demand file system. Figure 6 b) shows the same simulation on the global file system. On the PFS, the variation of the time-steps is significant. The black bars show the min-max value of the five consecutive runs. Using PFS the lowest times are around 0.02 seconds and maximum values range up to 0.1 seconds. With the on-demand file system, there is almost no deviation.

The two OpenFOAM use cases show a different behavior, depicted in Figure 7. When OpenFOAM is executing the time-steps in which data is written, both use cases show very high spikes (cmp. 7a 7c). Again a higher variance in the execution time is shown in Fig. 7b when using the PFS. Using the on-demand file system may cause less variance in the execution of the time-steps, but the execution time when writing data is longer. With the on-demand file system these steps need about 50 seconds, while using the PFS the same time-steps need in average under 25 seconds the OpenFOAM use case 2 shows only high spikes when using the on-demand file system (Figure 7d).

Table II shows the run times of the NAStJA simulations. The average simulation time is faster when using the on-demand file system. This is remarkable cause the compute nodes have to manage also the on-demand file system while running the application. Table III presents the run times for the two OpenFOAM use cases. Here, it is apparent that the total runtime is longer by about 60% on the on-demand file system. However, the total computation time is only about

TABLE II: NAStJA total run time on on-demand and global file system. Average of 5 runs.

file system used	Lustre	BeeGFS
fastest run (s)	304	319
slowest run (s)	435	326
average (s)	335	323

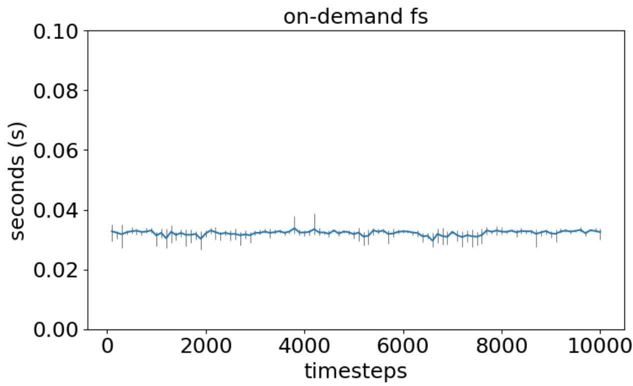
TABLE III: Initial time-step for OpenFOAM and total execution time.

	use case 1		use case 2	
file system used	Lustre	BeeGFS	Lustre	BeeGFS
initial time-step (s)	41	340	12	111
total computation time (s)	1331	1481	1111	1340
clocktime (s)	1416	2335	1195	1906

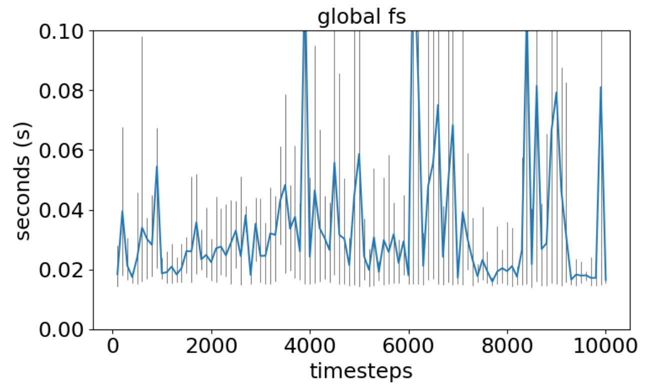
10% slower in use case 1 and about 20% in use case 2. It is quite interesting that the initial time-step is approximately ten times slower. Here the data for the simulation is read. Reading the initial time-step from the PFS would make more sense in this case.

C. Data staging

We also considered the case of copying data back to the PFS while the application is running. For this purpose, we used different NAStJA simulations on 23 nodes with 16, 19, and 20 cores per node for the application. This allows us to evaluate the impact of free resources for data staging. The remaining resources on the compute nodes are then available for the on-demand file system and data staging. Of course, when using all 20 cores for the application there are no physical cores left, but we wanted to evaluate how much the interference is. For reference, each simulation is performed without data staging. To stage the data, during the NAStJA execution, we used the parallel copy tool dcp [47]. As the staging workflow, we considered two cases: a single node with four dcp processes, and a case with one dcp process per compute node. In the case of a single node the MDS server node of the on-demand file system was used. Figure 8 show the average execution time per time-step of five runs in our different scenarios. With 16 cores for the application, from available 20 cores, the run times are similar whether the run was executed with or without data staging. When using 19 or 20 cores, the application is slowed down when the copy is executed with one process per node. At the beginning, the slowdown is significant (orange line) due to the high amount of metadata operations. In this case, a portion of the data is indexed on every node and this is causing interference with the application. When using only the MDS-server to copy the generated data (green line) the indexing is done only on the node with the MDS-server. Here, only a small influence can be observed, such that the interference to the application is negligible. Figure 9 shows the total time for the application run and data staging experiment. Using only the MDS node to copy the data, needs more time than the execution of our simulation. If there are enough free resources on the compute nodes, the data can be staged-out without

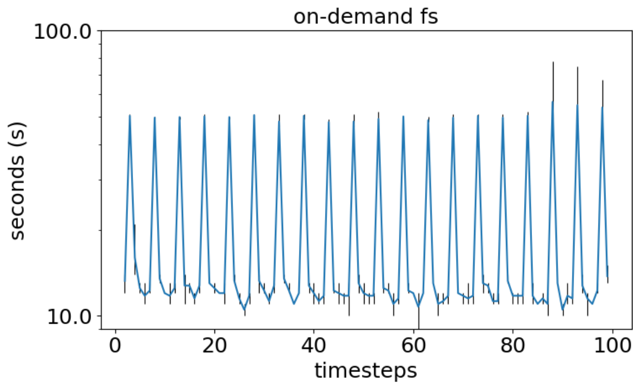


(a) NAStJA using on-demand fs

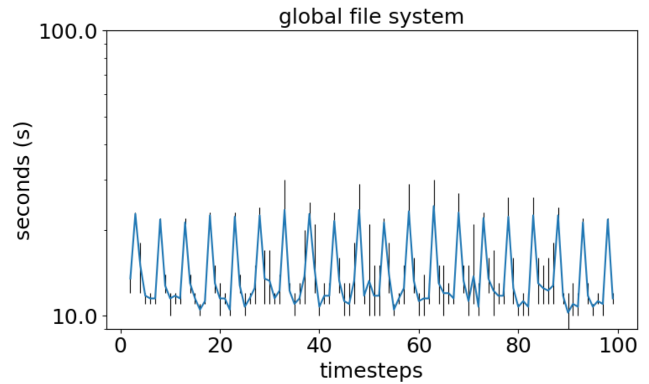


(b) NAStJA using global file system

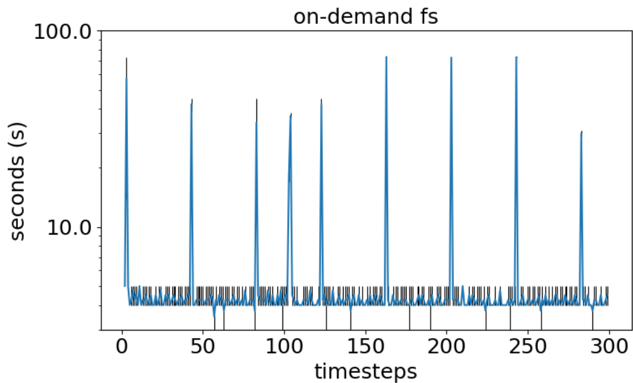
Fig. 6: Average execution of time-steps for five NAStJA runs. (Black bars Min/Max)



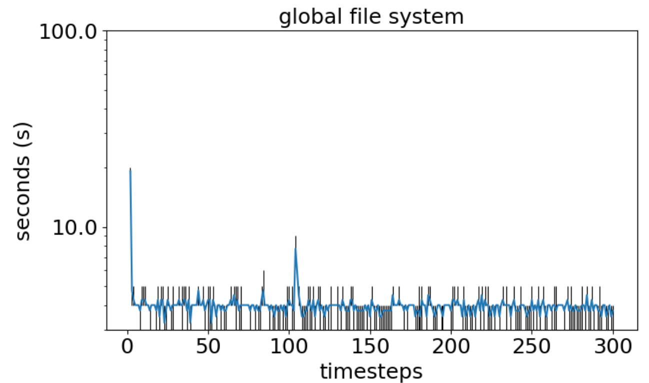
(a) OpenFOAM use case 1 on-demand fs



(b) OpenFOAM use case 1 global fs



(c) OpenFOAM use case 2 on-demand fs



(d) OpenFOAM use case 2 global fs

Fig. 7: Average execution of time-steps for OpenFOAM use cases. (Black bars Min/Max)

slowing down the application. Staging the data back afterwards with dcp needs approximately 30 seconds, and only 6 seconds for the pure data transfer. This raises the question of whether it makes sense to copy the data back during the simulation.

V. Conclusion & Future Work

We have shown a way to reduce the load on the global file system. This reduces the operational effort of the storage system. Despite the additional processes on the computing nodes, our chosen applications run more stable and with less variance in the run times. The applications are less affected by

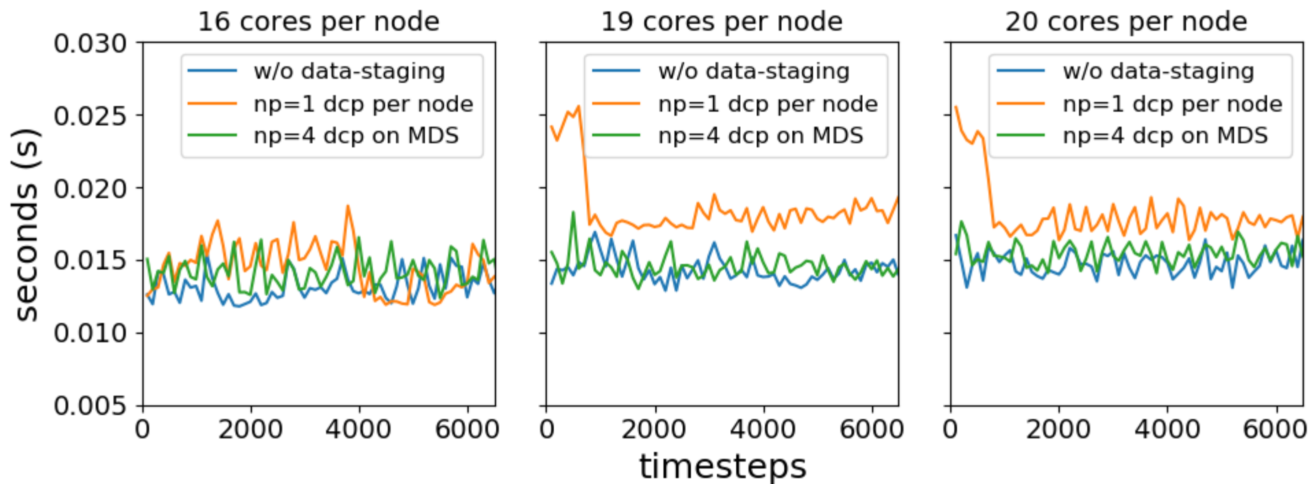


Fig. 8: Execution time(NAStJA) per time-step w/o data staging.

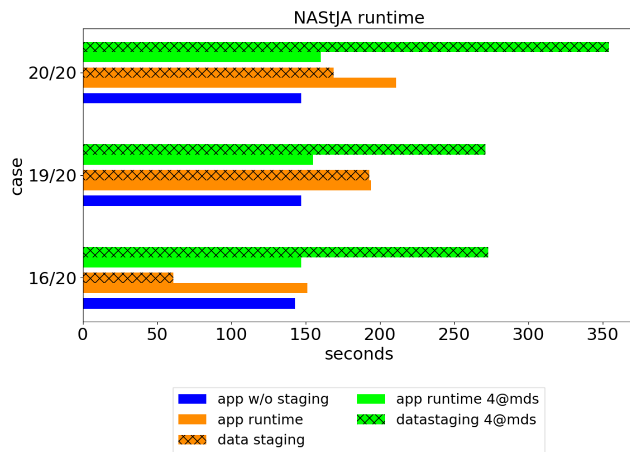


Fig. 9: Total execution time(NAStJA) for application w/o data staging.

side effects and generate less load on the PFS. Additionally, a job can run independently of the PFS. As a result, these storage systems are then only needed as staging storage.

The OpenFOAM use cases we evaluated are slower when using an on-demand file system. These cases are usually running for a whole day and slowing down the whole system. The user has to take precautions that only one job like this running at a time. Several such jobs at a time would completely slow down the storage subsystem, ending up with slowing down all job that are doing I/O. These jobs could be using an on-demand file system and would not have any impact to the storage system.

We have started the on-demand file system with standard configuration [48] [49]. With an analysis of the scientific applications, the on-demand file system can be optimized for specific use cases, e.g., more metadata performance instead of throughput. We have shown that staging back the data during execution can be very slow, if we want to keep the interference with the application low.

Using on-demand file systems is straightforward and the changes to the system are minimal. The advantage of this approach over others is that the application code does not need to be changed. Instead of changing the user code, a tailored file system for the application can be created. This gives additional flexibility for further optimization, e.g., file system permissions are not needed or file-locking could be disabled if the application do not need them.

However, further optimizations may be necessary for a production environment. For instance, the bootstrap time needs to be improved. There is no exact value for a acceptable time to create the on-demand file system. It makes little sense to wait an hour to start the on-demand file system if the job is not running for much longer. A few minutes for a one-day job on the other hand is acceptable. It must be carefully considered whether the waiting time for an on-demand file system is advantageous.

Furthermore, the question arises whether all the data written by the application has to be copied back to the PFS. With in-situ post-processing of the data on the fast, private on-demand file system, only the important data may be selected and copied back. This would reduce the demand and the load on the PFS.

Also the case of a node failure has to be considered. In such a situation we would lose the data on the node-local storage. This risk can be reduced, either by staging back the data from time to time or by using data duplication on different nodes of the on-demand file system.

The most important effect that can be achieved immediately is to reduce the load on the PFS. Nowadays the compute nodes are equipped with SSDs or faster storage devices and a distributed on-demand file system makes them easily accessible to

the user. Since no code adjustments are necessary, it is possible to provide an on-demand file system for applications with very high I/O demand. An on-demand file system does not change the I/O behavior of the application, but the influence on other HPC jobs and the whole system is reduced.

VI. Acknowledgement

This work as part of the project ADA-FS is funded by the DFG Priority Program “Software for Exascale Computing” (SPPEXA, SPP 1648), which is gratefully acknowledged. This research was conducted using the supercomputer ForHLR II and services offered by Karlsruher Insitute of Technology and the Steinbuch Centre for Computing. The authors gratefully acknowledge the time and granted access to ForHLR II.

References

[1] Peter J Braam and Philip Schwan. Lustre: The intergalactic file system. In *Ottawa Linux Symposium*, page 50, 2002.

[2] Frank Schmuck and Roger Haskin. GPFS: A shared-disk file system for large computing clusters. In *Proceedings of the 1st USENIX Conference on File and Storage Technologies*, FAST ’02, Berkeley, CA, USA, 2002. USENIX Association.

[3] Jan Heichler. An introduction to BeeGFS. http://www.beegfs.com/docs/Introduction_to_BeeGFS_by_ThinkParQ.pdf, 2014.

[4] Mehmet Soysal, Marco Berghoff, Thorsten Zirwes, Marc-André Vef, Sebastian Oeste, Andre Brinkman, Wolfgang E. Nagel, and Achim Streit. Using On-demand File Systems in HPC Environments. *Accepted @ The 2019 International Conference on High Performance Computing and Simulation (HPBench@HPCS)*.

[5] Orcun Yildiz, Matthieu Dorier, Shadi Ibrahim, Rob Ross, and Gabriel Antoniu. On the root causes of cross-application I/O interference in HPC storage systems. In *Parallel and Distributed Processing Symposium, 2016 IEEE International*, pages 750–759. IEEE, 2016.

[6] IBM. GPFS - highly available write cache (hawc), 2018.

[7] Richard Mohr, Michael J Brim, Sarp Oral, and Andreas Dilger. Evaluating progressive file layouts for lustre.

[8] BeeGFS. BeeGFS Storage Pool. https://www.beegfs.io/wiki/StoragePools, 2018. Accessed: August 18 2018.

[9] Jing Xing, Jin Xiong, Ninghui Sun, and Jie Ma. Adaptive and scalable metadata management to support a trillion files. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, SC ’09, pages 26:1–26:11, New York, NY, USA, 2009. ACM.

[10] S. Lang, P. Carns, R. Latham, R. Ross, K. Harms, and W. Allcock. I/O performance challenges at leadership scale. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, Nov 2009.

[11] Ning Liu, Jason Cope, Philip Carns, Christopher Carothers, Robert Ross, Gary Grider, Adam Crume, and Carlos Maltzahn. On the role of burst buffers in leadership-class storage systems. In *Mass Storage Systems and Technologies (MSST), 2012 IEEE 28th Symposium on*, pages 1–11. IEEE, 2012.

[12] Teng Wang, Kathryn Mohror, Adam Moody, Kento Sato, and Weikuan Yu. An ephemeral burst-buffer file system for scientific applications. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, page 69. IEEE Press, 2016.

[13] DataDirect Networks. IME - Flash native cache. https://www.ddn.com/products/ime-flash-native-data-cache/, 2018.

[14] CRAY. Cray® DataWarp™ Applications I/O Accelerator. https://www.cray.com/datawarp, 2018.

[15] John Bent, Garth Gibson, Gary Grider, Ben McClelland, Paul Nowoczynski, James Nunez, Milo Polte, and Meghan Wingate. Plfs: a checkpoint filesystem for parallel applications. In *Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, page 21. ACM, 2009.

[16] BeeGFS. BeeOND™: BeeGFS On Demand. http://www.beegfs.io/wiki/BeeOND, 2018. Accessed: August 18 2018.

[17] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the HDF5 technology suite and its applications. In *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases*, pages 36–47. ACM, 2011.

[18] Russ Rew and Glenn Davis. Netcdf: an interface for scientific data access. *IEEE computer graphics and applications*, 10(4):76–82, 1990.

[19] Jay F Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th international workshop on Challenges of large applications in distributed environments*, pages 15–24. ACM, 2008.

[20] Rajeev Thakur, William Gropp, and Ewing Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the sixth workshop on I/O in parallel and distributed systems*, pages 23–32. ACM, 1999.

[21] Rajeev Thakur, William Gropp, and Ewing Lusk. Data sieving and collective I/O in ROMIO. In *Frontiers of Massively Parallel Computation, 1999. Frontiers’ 99. The Seventh Symposium on the*, pages 182–189. IEEE, 1999.

[22] Wolfgang Frings. SIONlib. http://www.fz-juelich.de/ias/jsc/EN/Expertise/Support/Software/SIONlib/_node.html, 2009.

[23] Matthias Neuer, Jürgen Salk, Holger Berger, Erich Focht, Christian Mosch, Karsten Siegmund, Volodymyr Kushnarenko, Stefan Kombrink, and Stefan Wesner. Motivation and implementation of a dynamic remote storage system for I/O demanding HPC applications. In *International Conference on High Performance Computing*, pages 616–626. Springer, 2016.

[24] David Teigland and Heinz Mauelshagen. Volume managers in linux. In *USENIX Annual Technical Conference, FREENIX Track*, pages 185–197, 2001.

[25] Tim Wickberg and Christopher Carothers. The RAMDISK storage accelerator: A method of accelerating I/O performance on HPC systems using RAMDISKs. In *Proceedings of the 2nd International Workshop on Runtime and Operating Systems for Supercomputers*, ROSS ’12, pages 5:1–5:8, New York, NY, USA, 2012. ACM.

[26] Slurm - schedmd. http://www.schedmd.com.

[27] Adaptive Computing. http://www.adaptivecomputing.com.

[28] IBM - platform computing. http://www.ibm.com/systems/platformcomputing/products/lsf/.

[29] Adaptive Computing - Data staging. http://www.adaptivecomputing.com/blog-hpc/data-staging/.

[30] Steinbuch Center for Computing. Forschungshochleistungsrechner ForHLR 2. http://www.scc.kit.edu/dienste/forhlr2.php, 2016.

[31] Arden and Hikyu Lee. Analysis of chordal ring network. *IEEE Transactions on Computers*, C-30(4):291–295, April 1981.

[32] Kevin A. Brown, Nikhil Jain, Satoshi Matsuoka, Martin Schulz, and Abhinav Bhatele. Interference between I/O and MPI traffic on fat-tree networks. In *Proceedings of the 47th International Conference on Parallel Processing*, ICPP 2018, pages 7:1–7:10, New York, NY, USA, 2018. ACM.

[33] Keihji Yamamoto, Fumiyoshi Shoji, and Atsuya Uno. Analysis and elimination of client evictions on a large scale lustre based file system. Presentation at the Lustre User Group 15 (LUG’15), 2015.

[34] OpenCFD. *OpenFOAM: The Open Source CFD Toolbox. User Guide Version 1.4, OpenCFD Limited*. Reading UK, Apr. 2007.

[35] The OpenFOAM foundation. https://openfoam.org/, 2018.

[36] Thorsten Zirwes, Feichi Zhang, Jordan Denev, Peter Habisreuther, and Henning Bockhorn. Improved vectorization for efficient chemistry computations in openfoam for large scale combustion simulations. In W.E. Nagel, D.H. Kröner, and M.M. Resch, editors, *High Performance Computing in Science and Engineering ’17*. Springer, 2018.

[37] Thorsten Zirwes, Feichi Zhang, Thomas Häber, and Henning Bockhorn. Ignition of combustible mixtures by hot particles at varying relative speeds. *Combustion Science and Technology*, 0(0):1–18, 2018.

[38] R.S. Barlow, S. Meares, G. Magnotti, H. Cutcher, and A.R. Masri. Local extinction and near-field structure in piloted turbulent CH4/air jet flames with inhomogeneous inlets. *Combust. Flame*, 162(10):3516–3540, 2015.

[39] Thorsten Zirwes, Feichi Zhang, Jordan Denev, Peter Habisreuther, and Henning Bockhorn. Automated code generation for maximizing performance of detailed chemistry calculations in OpenFOAM. In W.E. Nagel, D.H. Kröner, and M.M. Resch, editors, *High Performance Computing in Science and Engineering ’17*, pages 189–204. Springer, 2017.

[40] Marco Berghoff, Ivan Kondov, and Johannes Hötzer. Massively parallel stencil code solver with autonomous adaptive block distribution. *IEEE Transactions on Parallel and Distributed Systems*, 2018.
[41] Neil Gunther. Unix load average, part 1: How it works. *Performance Dynamics Company, Feb*, 2003.
[42] Don Capps and William Norcott. Iozone filesystem benchmark, 2008.
[43] E Kim. SSD performance-a primer: An introduction to solid state drive perfromance, evaluation and test. Technical report, Tech. rep., Storage Networking Industry Association, 2013.
[44] Ji Jun Hung, Kai Bu, Zhao Lin Sun, Jie Tao Diao, and Jian Bin Liu. PCI express-based NVMe solid state disk. In *Applied Mechanics and Materials*, volume 464, pages 365–368. Trans Tech Publ, 2014.
[45] Qiumin Xu, Huzefa Siyamwala, Mrinmoy Ghosh, Tameesh Suri, Manu Awasthi, Zvika Guz, Anahita Shayesteh, and Vijay Balakrishnan. Performance analysis of nvme ssds and their implication on real world databases. In *Proceedings of the 8th ACM International Systems and Storage Conference*, page 6. ACM, 2015.
[46] Jim Garlick. Pdsh, 2018.
[47] D Sikich, G Di Natale, M LeGendre, and A Moody. mpifileutils: A parallel and distributed toolset for managing large datasets. Technical report, Lawrence Livermore National Lab.(LLNL), Livermore, CA (United States), 2017.
[48] "BeeGFS Wiki". BeeGFS - tips and recommendations for storage server tuning, 2018.
[49] BeeGFS. Beegfs wiki : Tuning advanced configuration. http://www.beegfs.com/wiki/TuningAdvancedConfiguration, 2016. Accessed: August 22 2016.