Data compression for climate data

DOI: 10.14529/jsfi160105

Data Compression for Climate Data

Michael Kuhn[1], *Julian Kunkel*[2], *Thomas Ludwig*[2]

The different rates of increase for computational power and storage capabilities of supercomputers turn data storage into a technical and economical problem. As storage capabilities are lagging behind, investments and operational costs for storage systems have increased to keep up with the supercomputers' I/O requirements. One promising approach is to reduce the amount of data that is stored. In this paper, we take a look at the impact of compression on performance and costs of high performance systems. To this end, we analyze the applicability of compression on all layers of the I/O stack, that is, main memory, network and storage. Based on the Mistral system of the German Climate Computing Center (Deutsches Klimarechenzentrum, DKRZ), we illustrate potential performance improvements and cost savings. Making use of compression on a large scale can decrease investments and operational costs by 50 % without negative impact on the performance. Additionally, we present ongoing work for supporting enhanced adaptive compression in the parallel distributed file system Lustre and application-specific compression.

Keywords: data compression, storage system, climate data, cost efficiency.

Introduction

Throughout the history of supercomputers as recorded by the TOP500 list, the computational power has been increasing exponentially, doubling roughly every 14.5 months [36]. While this increase in computational power has allowed more detailed numerical simulations to be performed, this has also caused the simulation results to grow in size exponentially. Computational speed and storage capacity have roughly increased by factors of 300 and 100 every 10 years, respectively. The storage speed, however, has only grown by a factor of 20 every 10 years, even when taking newer technologies such as SSDs into account. The different rates of improvement can be seen in Figure 1.[3] Thus, the importance of performing I/O efficiently and storing the resulting data cost-efficiently increases [23].

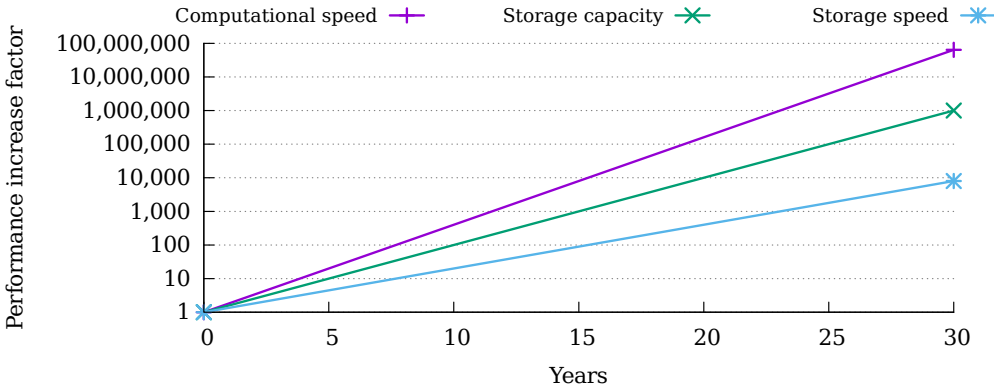


Figure 1. Development of computational speed, storage capacity and storage speed

Although it is theoretically possible to compensate for this fact in the short term by simply buying more storage hardware, new approaches are required to use the storage infrastructure as efficiently as possible due to the ever increasing gap between the faster-growing processing

[1]Universität Hamburg, Hamburg, Germany
[2]Deutsches Klimarechenzentrum GmbH, Hamburg, Germany
[3]It has to be noted that the computational speed is based on the TOP500 list while the storage capacity and speed are based on single devices.

power on the one hand and the lagging storage capacity and throughput on the other hand. Data reduction techniques present one such approach and play an important role in today's high performance computing (HPC) systems. Data reduction can be used to reduce the costs and size of storage systems, and to increase performance.

Overall, the storage subsystems can be responsible for a significant portion of a system's total cost of ownership. In the case of Mistral, it accounts for roughly 20 % of the overall costs. This amount of storage is necessary due to the huge amounts of data. In phase 5 of the Coupled Model Intercomparison Project (CMIP), a volume of 1.8 PB of data and 4.3 million files were created [1]. An estimate of the Earth System Grid Federation (ESGF)[4] for the upcoming phase 6 project is between 36 PB and 90 PB [9].

In an effort to keep storage costs from increasing, data reduction techniques are increasingly being deployed. Previous studies have shown that certain data reduction techniques can be beneficial for large-scale storage systems [25]. However, due to their inherent costs and complexities, techniques such as deduplication and re-computation are not suitable without restrictions. In particular, deduplication typically yields lower data reduction rates and may degrade performance, thus making it unsuitable for HPC [29, 30]. Compression can be deployed with relatively little overhead and the integration into existing systems is much easier.

Being such a versatile concept, compression can be introduced with different goals on various hardware and software levels. In this paper, we will investigate the possibilities of applying compression to various levels of the HPC hardware/software stack. In this regard, we will analyze typical storage behavior from a datacenter perspective and not focus on particular use cases or data formats. Specifically, the following goals will be addressed:

1. Memory capacity. By compressing data in memory, additional memory becomes usable, and out-of-core computing can be avoided. Usually, this can be done by providing two pools of memory: one compressed pool for slower access and a regular pool, in which the working set fits. Each time data from the compressed memory pool is required, it must be decompressed and either migrated into the regular pool or directly into the CPU caches.
2. Network throughput. By compressing data before transmitting it via the network, communication performance can be increased. This typically requires fast compression algorithms, but not necessarily compression speeds higher than the throughput of the network. In case of storage systems, data does not have to be decompressed on the receiving side, leading to further benefits.
3. I/O throughput. Compressing data before writing it to the storage hardware can increase overall I/O performance because less data has to be written. Conversely, less data has to be read from the storage hardware, also increasing performance for the read case.
4. Storage capacity. In addition to the already mentioned performance benefits, compressing data reduces its storage footprint, allowing more data to be stored with the same storage hardware.
5. Cost and energy efficiency. Compression can be used to improve the cost efficiency of data storage. Additionally, if data has to be archived for long periods of time, it can be worthwhile compressing it using slower algorithms yielding higher compression ratios.[5]

[4] `http://esgf.llnl.gov/`

[5] We explicitly define compression ratio as the fraction of uncompressed size over compressed size, that is, compression ratio $= \frac{\text{uncompressed size}}{\text{compressed size}}$. Due to its convenience, the inverse compression ratio, that is, 1 divided by the compression ratio, will also be used at some points in the paper; it indicates the fraction to which data can be compressed.

A proper integration of the mentioned approaches also allows avoiding decompression at different levels of the stack, further increasing its effectiveness. Compression offers the chance of improving the cost and energy efficiency of existing hardware, and also allows improving performance. With respect to the above list, the main contributions of this paper are:

1. A thorough analysis of existing compression technologies regarding their applicability in high performance computing and the I/O stack.
2. Models to estimate the impact of compression on performance and cost of supercomputers and their storage systems.

This paper is structured as follows: Section 1 contains state-of-the-art and related work regarding the compression and its applicability in HPC contexts. In Section 2, we model the impact of different compression techniques on performance and cost. We then describe ongoing work for applying different types of compression in a parallel distributed file system and applications in Section 3. Section 4 concludes the paper and discusses future work.

1. State-of-the-art and related work

A multitude of different compression algorithms and possibilities for applying compression exist. In this section, we will describe approaches for addressing the goals in more details and quantify their respective advantages and disadvantages.

1.1. Compression algorithms

There is a wide range of compression algorithms. General *lossless* compression algorithms, such as DEFLATE [10] and LZO [20], consider data to be an array of characters. Therewith, the algorithm has to ensure that any sequence of characters can be compressed and reconstructed. Usually, applications store more complex data types and compound structures. Knowing the data structures would allow to reduce the required information in the compressed representation as impossible configurations of data can be ignored. For instance, several lossless compression algorithms specifically tailored to floating-point data are available [27, 33].

To improve performance, several approaches have been investigated. LZRW [39] is a variant of the LZ77 [42] compression algorithm tuned for speed. The lz4fast algorithm is a new variant of the lz4 compression algorithm that allows users to specify an *acceleration factor* to improve compression speed by sacrificing compressibility [41]. Preliminary tests have shown that lz4fast can reach up to several GB/s per core both for compression and decompression, making it especially interesting for HPC. Because throughput requirements are increasing, algorithms have also been ported to accelerators [32] and have been implemented in hardware [2, 4, 8] to accelerate compression speed. With SLDC [12], the tape technology LTO offers hardware-accelerated compression performed by the tape drive. One drawback of this approach is that it is not possible to know in advance how much capacity a tape drive actually has.

If the application knows the tolerable errors in the representation, *lossy* compression schemes can reduce data further to the level that is needed to reconstruct the data with the required precision. However, such an application-specific schema requires insight from the domain. ISABELA [26] and ZFP [28] are compression schemes for floating-point data. ISABELA is a preconditioner that reorders data to improve the efficiency of the later compression stages. ZFP supports lossy compression and allows to define either the relative error tolerance or absolute error tolerance. Multidimensional variables usually exhibit some locality and this smoothness of

data even increases with the resolution of the simulation domain. Such regularities on grids can be exploited by preconditioning the input data, for example, as done by ZFP and our own approach MAFISC [18]. A compression scheme may even be tailored for a particular grid structure. Wang et al. exploit the topology of icosahedral grids that are now common in atmospheric models by mapping each rombus to a 2D matrix that can be compressed with wavelet schemes [37].

1.2. Memory capacity

Since the early days of personal computers, in-memory compression has been used to virtually enlarge memory capacity. For example, in the early days of Intel's 386 and MS-DOS, the Quarterdeck Expanded Memory Manager (QEMM) has been widely used. For modern systems, Linux's *zram* framework provides a simple way to create compressed block devices [31]. To increase the main memory capacity, these block devices can then be used as swap devices. For instance, given a machine with 8 GiB of physical main memory, an additional 8 GiB zram block device can be created. Assuming a compression ratio of 2.0, the zram block device would require 4 GiB of physical main memory, providing a total of 12 GiB of main memory. Obviously, the final amount of available main memory heavily depends on the data written to the compressed block device.

The impact on performance of this solution can be hard to predict because the data is not compressed and decompressed on every access, but only if Linux decides that a page should be swapped in or out. That is, live data will typically stay uncompressed and only be compressed when the page containing it is swapped out. Whether a page should be swapped out is determined by the kernel's `swappiness` parameter.

Listing 1 shows the steps of setting up a compressed zram block device and using it as a swap device. First, the `zram` kernel module has to be loaded to be able to use the zram framework. Afterwards, the `zramctl` command line tool can be used to set up compressed block devices. In this case, a block device with 8 GiB of space is set up using the first free identifier. It is also possible to change the used compression algorithm and the number of compression streams to increase performance. By default, the `lzo` algorithm and only one compression stream are used. Given sufficient parallelism from the application side, increasing the number of compression streams can be used to effectively scale the compression throughput.

```
$ modprobe zram
$ zramctl --find --size 8G
/dev/zram0
$ mkswap /dev/zram0
$ swapon /dev/zram0
```

Listing 1. Setting up a compressed block device with a size of 8 GiB and using it as swap space

1.3. Network throughput

There are several approaches for message compression in the Message Passing Interface (MPI), all of them virtually increasing network throughput. CoMPI allows compressing MPI messages by adding a compression layer to the ADI of MPICH [16]. While beneficial in many cases, the evaluation of HPC applications is conducted using Fast Ethernet and, thus, the applicability to recent HPC systems is limited. Adaptive-CoMPI improves this approach by selecting the compression algorithm at runtime and allowing developers to specify guiding information [17].

Since the compression ratio depends on data properties such as the inherent redundancy, the evaluation of this approach shows that it is able to improve throughput for some of the tested scientific applications and may only degrade performance slightly. The PRAcTICaL-MPI wrapper transfers the strategy to a library that utilizes the MPI standard profiling interface (PMPI) [14]. Due to the portability of PMPI, it can be deployed without changing code or MPI implementation.

In [32], it has been discussed whether it is worthwhile to compress CPU/GPU data transfers over the PCI-Express bus. However, due to the speed of PCIe 3.0 (x16) in the order of 16 GB/s, this idea has been discarded.

1.4. Storage capacity and throughput

Welton et al. compress network traffic between clients and the I/O forwarding layer to improve performance [38]. Their evaluation is conducted on an Ethernet and a QDR InfiniBand connected cluster. While on Ethernet, LZO increases performance significantly and even does not degrade performance for random workloads, the performance of IB networks is degraded when applying compression. The Autonomous and Parallel Compressed File System [22] is a FUSE file system stacked on top of a local file system that transparently compresses data passed through it. It uses LZ77 and adaptive Huffman encoding. The authors aim to select adaptively the algorithm based on the data type (for example, text and movies).

In [15], Filgueira et al. apply their adaptive compression scheme of CoMPI to I/O. They implement the I/O compression into the Papio parallel storage system that offers quality of service (QoS). Writes are performed in atoms of `stream_width`, that is, the number of strips times the `stripe_size`. Their approach provides heuristics to estimate compression rate and network performance, and, thus, the achieved speedup. Based on the estimate and selected QoS, the appropriate algorithm is selected. Compression of the data can then be performed by multiple threads. The evaluation is conducted on a 10 Gbit Ethernet system but QoS limits throughput to 300 MB/s. It is demonstrated that the heuristics approximate the speedup of the compression schemes well. The adaptive compression improves speedup in all cases but degrades with increasing throughput. For reads, the achieved speedup is about $^3/_4$ of the compression speed.

1.4.1. File systems

Currently, only a hand full of file systems support compression natively. All of them are local file systems, the most important ones being btrfs [21], NTFS and ZFS [5]. Due to NTFS only being available on Windows, however, it typically does not play a role in HPC.

btrfs supports compression on a per-extent level with a maximum size of 128 KiB. It supports the compression algorithms LZO and zlib, which can be selected at mount time by specifying the mount options `compress` or `compress-force`; they will either skip compression of incompressible files or force compression of all files, respectively. Compression can also be enabled and disabled for individual files and directories using the `c` attribute of `chattr`. ZFS supports compression on a per-record level with a typical maximum size of 128 KiB; newer versions of ZFS allow recording sizes of up to 1 MiB, which can increase the efficiency of compression algorithms. ZFS has support for zero-length encoding (zle), gzip (with levels 1–9), lzjb and lz4. Compression can be set individually for each file system using the `zfs set compression` command.

1.5. Cost and energy efficiency

Cost and energy efficiency are important topics in high performance computing, as demonstrated by initiatives, such as the Green500 list [35]. However, the impact of compression on these two metrics has so far mainly been analyzed in the context of embedded and mobile devices, where data transmission is especially expensive [3, 11, 34, 40]. Studies have shown that compression can increase energy consumption instead of decreasing it. This is due to the fact that algorithms might spend a disproportional amount of energy for compressing the data. That is, it is also important to look at how much energy is consumed to compress a given amount of data (MB/J) instead of only compression throughput (MB/s). While there have been studies on the energy and performance impact of compression for data-intensive workloads such as MapReduce [7] and for file systems in general [24], the cost aspect of the overall system has often been neglected. Our own studies have shown that efficient compression algorithms can be beneficial both in terms of performance and energy consumption even for HPC applications [6].

2. Modeling the impact of compression

An important factor when considering compression is its impact on performance. Depending on the used compression algorithm, its settings and where compression is applied, it can be either beneficial or detrimental for performance. The overall impact of introducing compression into scientific workflows depends on the strategy, that is, which data to compress and when to compress it, the characteristics of the data, the characteristics of the compression algorithm and the hardware characteristics. In detail, the strategy determines the number of times data is compressed/decompressed in the workflow. Time needed to compress/decompress is mainly determined by the compression algorithm and the CPU but also influenced by the structure of the data. The achieved compression ratio is mainly determined by the algorithm but is also influenced by the compressibility of the data. The benefit of any strategy comes from virtually increasing throughput and storage capacity in hardware components. Since characteristics of compression algorithms vary, the best fitting algorithm for a given scenario can be chosen or adaptively determined.

2.1. Performance considerations

For the following discussion, we assume an application consists of the three phases: compute, communication and I/O, as shown in Figure 2. The overall runtime of the application is $t = t_{cpu} + t_{net} + t_{io}$. Note that if multiple phases of the same type are observable, we just accumulate the time for each phase.

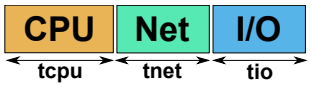


Figure 2. Phases of an HPC application

It could be an iterative algorithm – such as shown in Figure 3 – that uses asynchronous communication and I/O to hide the time needed for communication and I/O. The exact composition of the application does not matter, the total time spent in computation, communication and I/O are labeled as t_{cpu}, t_{net} and t_{io}, respectively, since they utilize these hardware components.

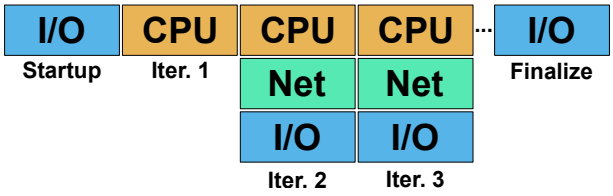


Figure 3. Phases of an HPC application when using asynchronous communication and I/O

When introducing a compression strategy to communication and I/O on the compute nodes, this requires additional time to compress (t_c) when sending/writing data and to decompress (t_d) when receiving/reading data. Similarly, when compressing data structures in memory, the computation pattern changes to include phases to compress/decompress from the compressed memory pool to the CPU cache (not shown in the figures).

Let us first discuss the benefit of hardware-accelerated compression performed by the network interface, this would allow to compress communication but also data on the I/O path without overhead for the computation while still allowing RDMA. Moreover, if the I/O server can disable decompression while receiving data blocks to be stored, that data can be stored in its compressed form without additional costs (similarly for the read path). The application runtime using this strategy becomes $\hat{t} = t_{cpu} + \hat{t}_{net} + \hat{t}_{io}$, where $\hat{t}$ includes the time for the compression and decompression. For larger amounts of data, the network interface can overlap compression with sending/receiving of data. Thus, it can hide the time needed for compression as long as the performance of the compression algorithm is, at least, as fast as the network throughput. $\hat{t}_{net} = \frac{t_{net}}{cr_{net}}$ and $\hat{t}_{io} = \frac{t_{io}}{cr_{io}}$, where cr_p is the compression ratio of phase p. In this case, this leads to a total time $\hat{t}$, as shown in Equation (1).

$$\hat{t} = t_{cpu} + \frac{t_{net}}{cr_{net}} + \frac{t_{io}}{cr_{io}} \tag{1}$$

As hardware-accelerated compression/decompression is typically not available using current hardware, it is important to also model the unaccelerated case. If the network interface does not support compression and data is uncompressed in memory, then we have to explicitly compress/decompress data, increasing the required CPU time t_{cpu} by the time required for compression (t_c) and decompression (t_d). This leads to an overall time $\hat{t}^+$, as shown in Equation (2), where s_p denotes the data size of phase p and p_p denotes the performance of phase p.

$$\begin{aligned} \hat{t}^+ &= t_{cpu} + \frac{t_{net}}{cr_{net}} + \frac{t_{io}}{cr_{io}} + t_c + t_d \\ &= t_{cpu} + \frac{s_{net}}{p_{net} \cdot cr_{net}} + \frac{s_{io}}{p_{io} \cdot cr_{io}} + \frac{s_c}{p_c} + \frac{s_d}{p_d} \end{aligned} \tag{2}$$

In a perfectly balanced system, $p_{io} = p_{net}$. Performing this process only pays off, if the time saved through compression ($t - t_{compressed}$) is larger than the time it takes to perform it ($t_c + t_d$). This leads us to the estimation in Equation (3), where cr^{-1} is the inverse compression ratio.

$$
\begin{aligned}
t_c + t_d &< t - t_{compressed} && \Leftrightarrow \\
\frac{s}{p_c} + \frac{s}{p_d} &< \frac{s}{p} - \frac{s \cdot cr^{-1}}{p} && \Leftrightarrow \\
\frac{p_c + p_d}{p_c \cdot p_d} &< \frac{1}{p} \cdot (1 - cr^{-1}) && \Leftrightarrow \\
p &< \frac{p_c \cdot p_d}{p_c + p_d} \cdot (1 - cr^{-1}) && (3)
\end{aligned}
$$

If we can pipeline compression and decompression on sender and receiver side, this equation becomes $p < \min(p_c, p_d) \cdot (1 - cr^{-1})$. Using this equation, we can generate thresholds for when compression pays off; the thresholds for three algorithms as measured on Mistral are shown in Table 1. Actually, the network interface bandwidth is shared among all cores, thus, the value determined can be multiplied with the number of cores.

Table 1. Network throughput thresholds for compression to pay off without hardware acceleration (network throughput has to be below given value)

Algorithm	CPU	Pipelined	CPU (24 cores)	Pipelined (24 cores)
pithy	791 MB/s	1,170 MB/s	19,000 MB/s	28,000 MB/s
blosc	389 MB/s	442 MB/s	9,300 MB/s	10,600 MB/s
lz4fast	914 MB/s	1,330 MB/s	21,900 MB/s	31,900 MB/s

The performance benefit or drawback, when compressing network communication, can be visualized in a 2D graph with the number of cores and the compression ratio as axes. Figure 4 illustrates the speedup for Mistral's FDR InfiniBand and using the lz4fast compression algorithm without hardware support. On our system, lz4fast achieves a throughput of roughly 2,900 MB/s for compression and 6,500 MB/s for decompression.[6] It can be observed that using 17 cores and an inverse compression ratio of 0.5, a speedup of 1.5 is possible. This almost matches the speedup when upgrading from FDR to EDR InfiniBand (which is faster by a factor of 1.77) and can thus be a viable approach to improve throughput without increasing costs for network equipment.

2.1.1. Parallel distributed file systems

Considering parallel distributed file systems, compression can either be applied on the clients or on the servers. Both approaches have benefits and limitations. The maximum throughput (p) in a parallel distributed file system is limited by several factors. The most important ones are client (p_c), network (p_n) and server (p_s) throughput, as shown in Equation (4).

$$p = \min(p_c, p_n, p_s) \tag{4}$$

p_n is static and independent on any potential data reduction taking place. p_c and p_s, however, depend on memory throughput and are therefore heavily dependent on which compression algo-

[6] A random selection of files on the storage system has been chosen for analysis (7,335 files with a volume of 300 GiB). The files contain a representative sample of scientific data formats and text files. They are compressed running independent compression/decompression jobs on 24 cores of a compute node concurrently. Throughout the paper, all specified values are the arithmetic mean across these files.

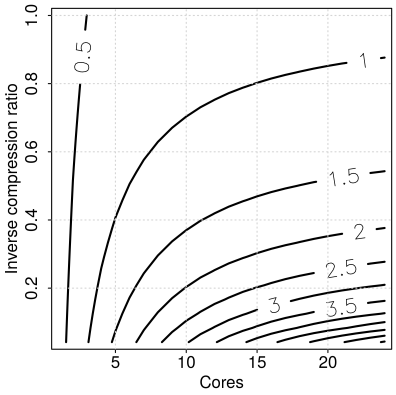
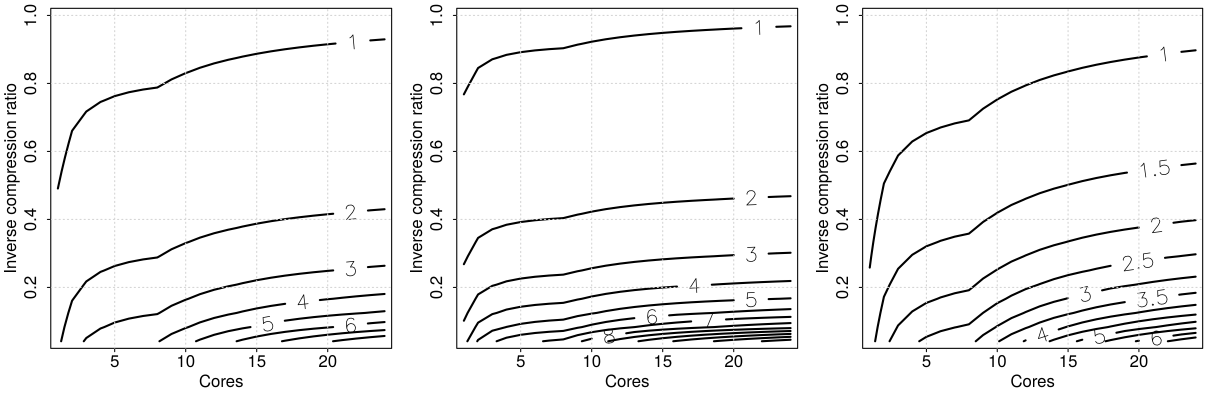


Figure 4. Performance multiplier when compressing communication on Mistral's FDR InfiniBand

rithm and settings are used. In real systems, slow compression algorithms can be counterbalanced to a certain extent by using write-behind caching, which allows applications to progress immediately and compression to take place in the background. For the sake of simplicity, we will assume that all data has to be written immediately and performance is thus limited by the compression algorithm's throughput. Slow compression algorithms can therefore slow down overall I/O speed significantly; their main benefits lie in further reducing the amount of data, saving costs. When applying compression in a file system, there are several possible approaches:

1. When writing data, it is compressed on the client, sent to the server and stored there in its compressed form. When reading the data again, the server returns the compressed data, which is then decompressed on the client. This approach might have disadvantages when performing small accesses because complete blocks of data have to be compressed and decompressed, causing additional overhead. Further analyses regarding the actual impact are necessary as HPC applications typically perform large accesses.
2. When writing data, it is compressed on the client and sent to the server. The server then decompresses the data and stores it in its decompressed form. When reading the data again, the server compresses the data, which is then decompressed on the client. This approach causes additional overhead on the server but does not suffer from the small access problem mentioned above because data can be accessed at finer granularity.



a) Write (compress) b) Read (decompress) c) Write with server decompress

Figure 5. Performance multiplier when (de)compressing data in the Lustre client

Figures 5a to 5c show the visualization of possible performance that increases for Mistral's Lustre file system. While Figures 5a and 5b contain the performance multipliers for writing and reading the data using the first approach, respectively, Figure 5c shows the performance when writing the data using the second approach. As can be seen, it is easier to improve performance when decompressing data on read than when compressing it on write due to the much higher decompression speeds of lz4fast. Additionally, compressing the data on the client and decompressing it on the server before actually writing it incurs additional overhead that requires more cores to be worth it. Overall, while compression requires a significant amount of cores to improve performance, it is also unlikely to degrade performance. Decompression, however, can achieve higher performance with even a small amount of cores, showing the usefulness of this approach.

2.2. Cost considerations

Besides its impact on performance, compression can also be an important factor in reducing costs. While this mainly applies to the reduction of the data's footprint, it can also have secondary benefits such as being able to spend less money on main memory or network infrastructure. Additionally, increase in throughput typically decrease costs indirectly because execution times are reduced. There are two approaches when utilizing compression:

1. Using compression to reach the desired metric, such as storage capacity, main memory capacity or network throughput. This allows spending less money for the respective component and using the remaining money for different purposes.
2. Using compression to improve the desired metric. This involves spending the same amount of money for the respective component while gaining more performance and/or capacity.

The following considerations will take both approaches into account. Depending on the examined hardware component, one or the other makes more sense. All cost considerations will be inspired by DKRZ's current Mistral supercomputer.

Table 2. Compression throughput and ratio for selected compression algorithms as measured using real data

Algorithm	Compression	Decompression	Ratio
lz4fast	2,945 MB/s	6,460 MB/s	1.825
lz4	1,796 MB/s	5,178 MB/s	1.923
lz4hc	258 MB/s	4,333 MB/s	2.0
lzo	380 MB/s	1,938 MB/s	1.887
xz	26 MB/s	97 MB/s	2.632
zlib	95 MB/s	610 MB/s	2.326
zstd	658 MB/s	2,019 MB/s	2.326

Table 2 shows the compression throughput and ratio for selected compression algorithms as measured on a set of real data on Mistral's storage system (for details, see Footnote 6). As can be seen, the algorithms have vastly different compression throughputs and ratios. Some algorithms such as zstd outperform other algorithms such as lzo for both throughput and ratio, but usually algorithms with lower throughputs also achieve higher compression ratios.

2.2.1. Main memory

Typically, the amount of main memory in a supercomputer depends on the applications that will be executed on it. Therefore, we will consider the goal of reaching a given amount of the main memory per node and use compression to reduce the amount of necessary hardware. Mistral has a total main memory capacity of roughly 320 TB. Assuming costs of € 200 per 32 GB of the main memory, the total costs for main memory only are € 2,000,000. Additionally, we will assume that each node is equipped with 128 GB of the main memory, resulting in 2,500 nodes.[7]

Since we are interested in reaching a per-node main memory capacity of 128 GB, we can calculate the amount of necessary physical main memory $mem = \frac{mem_c}{ratio}$, where mem is the amount of physical main memory, mem_c is the amount of the main memory after compression and $ratio$ is the average compression ratio. As mentioned previously, zram supports the lz4 and lzo compression algorithms. For lz4 this would be $\frac{128}{1.923} = 66.56$ GB and for lzo $\frac{128}{1.887} = 67.83$ GB. The main memory capacity of nodes can not be chosen freely and some main memory should be reserved for keeping uncompressed data. Therefore, it would make sense to either use a configuration with 96 GB of the main memory or to use a 64 GB configuration that relaxes the 128 GB requirement slightly. This configuration could reserve 4 GB for uncompressed data and provide $60\,\text{GB} \cdot 1.923 = 115.38\,\text{GB}$ or $60\,\text{GB} \cdot 1.887 = 113.2\,\text{GB}$ of compressed main memory for lz4 or lzo, respectively. One important factor that has to be kept in mind is the main memory throughput. While lz4 is typically fast enough to saturate the memory bus if enough parallel compression streams are used, lzo is much slower. Even with 24 parallel streams, lzo can only achieve $24 \cdot 380\,\text{MB/s} = 9.12\,\text{GB/s}$.

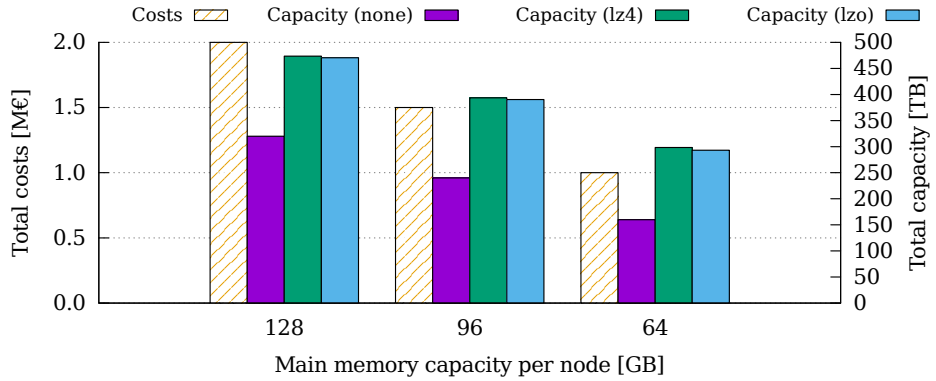


Figure 6. Total costs and capacity in relation to the amount of main memory per node

Figure 6 shows three node configurations with 128, 96 and 64 GB of the main memory. It contains the total costs required to procure the main memory and total capacities with no compression, lz4 and lzo. For the configurations with 128 and 96 GB, the main memory is split into an uncompressed and a compressed pool dynamically; only the necessary amount of the main memory to reach a per-node capacity of at least 128 GB is compressed, the remaining main memory is left uncompressed. For example, $\frac{128\,\text{GB}}{1.923} = 66.56\,\text{GB}$ are compressed for lz4, the remaining part is left uncompressed. For the 64 GB configuration, 60 GB are compressed while 4 GB are left uncompressed, resulting in less than 128 GB capacity per node. As it can be seen, the main memory's cost efficiency – that is, the amount of memory per € – increases as more main memory is compressed. Reducing the amount of the main memory is worthwhile because it is responsible for roughly 10 % of the overall power consumption according to [23, page

[7] Mistral has roughly 3,000 nodes and features different main memory configurations.

165]. However, as mentioned previously, zram's impact on performance can be hard to predict. Therefore, keeping a pool of uncompressed main memory for fast access makes sense. We will investigate this approach and its impact on performance in more detail in the future.

2.2.2. Network

Compression can also be used to virtually increase network throughput. In the following, we will investigate whether this can be translated into cost savings. As with the main memory, the network can not be scaled arbitrarily but is limited to certain stages given by the used network technology. For high performance computing, InfiniBand is one of the most common network technologies and is available in several different speeds. The most widely used version is FDR, which offers a latency of 0.7 μs and a throughput of 54.54 Gbit/s when using four links. The newest version, EDR, offers a latency of 0.5 μs and a throughput of 96.97 Gbit/s using four links. However, the older version QDR and Ethernet networks are also interesting due to their potentially lower costs.

Mistral is using FDR InfiniBand. When using lz4fast and 24 cores, we can reach a maximum compression throughput of $24 \cdot 2,945\,\text{MB/s} = 70,680\,\text{MB/s}$, which is enough to saturate even the fastest network. We can saturate the FDR InfiniBand network using only three cores and achieve a throughput of 99.54 Gbit/s due to the compression ratio of 1.825. This is even faster than what EDR InfiniBand can offer. When looking at saving costs by choosing a slower interconnect, however, even QDR InfiniBand could be used with its 32 Gbit/s throughput to achieve the same performance as FDR InfiniBand. The throughput increases to $32\,\text{Gbit/s} \cdot 1.825 = 58.4\,\text{Gbit/s}$ when applying compression with lz4fast in this case.

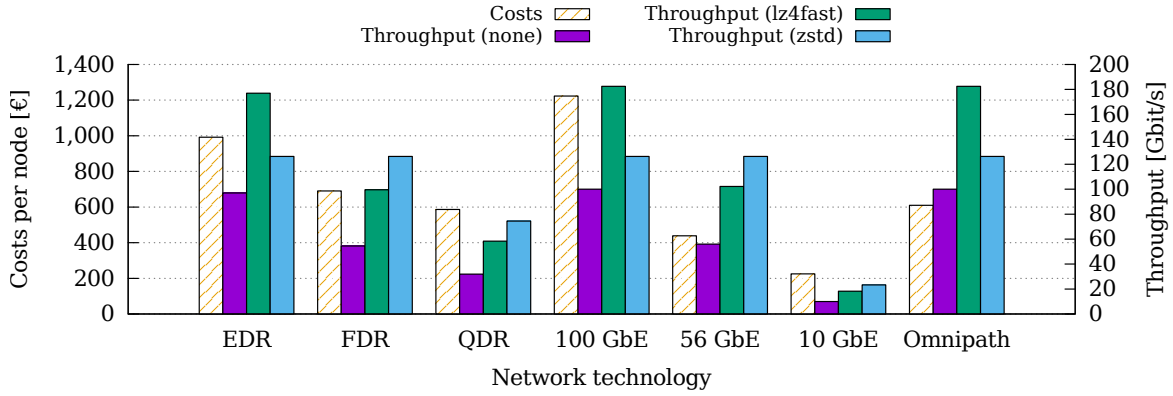


Figure 7. Costs and throughput for network technology when compressing communication

Figure 7 shows the costs and throughput per node when equipping nodes (and switches) with EDR, FDR, or QDR InfiniBand, 100 Gbit/s, 56 Gbit/s, or 10 Gbit/s Ethernet or Omnipath. The maximum throughput is shown for lz4fast and zstd. When using zstd with networks faster than 54 Gbit/s, the maximum throughput is limited due to zstd's compression throughput. Therefore, lz4fast is usually the better choice for high performance networks. Mistral's FDR InfiniBand network could be replaced with QDR InfiniBand when applying lz4fast compression for all network communication, decreasing costs by 15 %. Alternatively, the per-node throughput could be improved to roughly 100 Gbit/s or 125 Gbit/s when using lz4fast or zstd, respectively.

2.2.3. Storage

As mentioned previously, Mistral's storage system cost approximately €6,000,000. The storage is distributed across roughly 60 Scalable Storage Units (SSUs), which contain two complete storage servers, and 60 Expansion Storage Units (ESUs) that are just JBODs, each connected to one SSU. Therefore, we can assume a cost of €100,000 per SSU/ESU pair with a base cost of €10,000 per SSU/ESU pair and up to €90,000 for the HDDs.[8] Additionally, each SSU/ESU pair is capable of providing 833 TB of capacity and delivering 10.8 GB/s of throughput. For the following considerations, we will only take server-side compression into account because the performance aspect of client-side compression has been implicitly included in the network throughput analysis.

Since the storage capacity and throughput can be scaled linearly by simply adding or removing more SSU/ESU pairs, the resulting storage capacity only depends on the compression ratio. The storage throughput depends on the compression throughput and the number of SSU/ESU pairs. Therefore, we will consider different compression algorithms in this case. Additionally, we will assume that the SSU's CPUs are completely busy with the normal file system load, that is, we will have to buy additional CPUs for the compression overhead. Each SSU/ESU pair is already equipped with two 8-core CPUs and we will include additional costs of €1,500 per SSU/ESU pair for compression. To take different key aspects into account, we will consider two scenarios:

- S1: We determine the number of SSU/ESU pairs necessary to achieve a capacity of 50 PB and only purchase this amount. This scenario will typically result in lower costs and decreased throughput.
- S2: We determine the number of HDDs necessary to achieve a capacity of 50 PB and spread these disks evenly across 60 SSU/ESU pairs. This scenario will typically result in both slightly higher costs and higher throughput than S1.

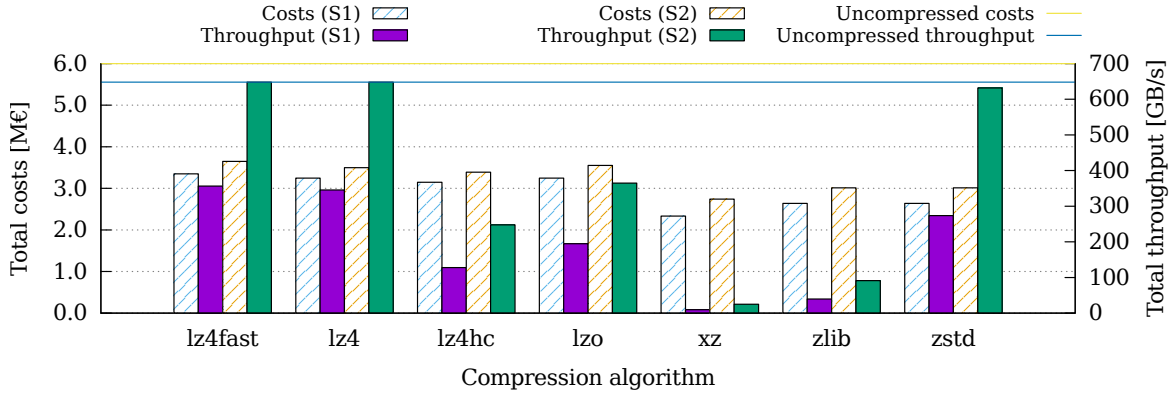


Figure 8. Storage costs and throughputs for different compression algorithms

Figure 8 shows the total costs and total throughput of Mistral's storage system when using the compression algorithms, shown in Table 2. First, we will take a look at scenario S1. As can be seen, employing compression allows decreasing the amount of necessary SSU/ESU pairs and thus costs. However, this also reduces the total throughput of the system because each SSU/ESU pair can only deliver at most 10.8 GB/s. The lz4fast and lz4 algorithms manage to achieve more than 10.8 GB/s per SSU/ESU pair and thus do not influence the throughput negatively on their own. The maximum throughput of zstd is only slightly less than that; therefore, the performance

[8] An SSU/ESU pair might be cheaper because auxiliary infrastructure is required in addition to the pair. However, for the purposes of modeling the possible savings, this estimation is good enough.

degradation can be mostly neglected. For these algorithms, the decrease in total throughput is caused exclusively by the reduction of the number of SSU/ESU pairs. However, lz4hc, xz and zlib have low throughputs and therefore do decrease total throughput.

When looking at scenario S2, costs are slightly increased overall due to the base cost of each SSU/ESU pair. However, the higher number of SSU/ESU pairs significantly increases throughput. For lz4 and lz4fast, performance is not degraded at all and costs are decreased to roughly €3,500,000. In zstd's case, throughput is decreased insignificantly by 20 GB/s while costs are reduced by 50 % to €3,000,000.

2.3. Summary

The results presented in the previous sections draw a clear picture: using compression on different levels of the I/O stack presents opportunities for both performance increases and cost reductions. In the best case, the different approaches should be combined for their synergetic effects. Table 3 shows an overview of the advantages and disadvantages on CPU utilization, memory capacity, network throughput, storage capacity and cost savings when deploying compression on different levels of the I/O stack.

Table 3. Design choices for compression on various levels

Strategy	CPU	Memory	Network	Storage	Cost savings
Memory	−	+	+	0	0
Communication	−	0	+	0	0
I/O (client)	−	0	+	+	+
I/O (server)	−	0	0	+	+
All levels	−	+	+	+	+

3. Ongoing work

Due to the very promising perspectives regarding performance improvements and cost savings, we have started to integrate compression into the HPC I/O stack on several levels. In the following, two approaches are described, one for lossless compression within the file system and one for lossy application-specific compression.

3.1. File system compression

Parallel distributed file systems typically do not have support for compression. However, several of them use underlying local file systems such as btrfs or ZFS and thus allow using those with compression support. While OrangeFS can use any POSIX file system, Lustre is limited to either ldiskfs or ZFS [19]. Consequently, using Lustre with its ZFS backend, it is possible to enable compression on a Lustre file system. However, Lustre currently has no support for compression, that is, it can not enable it by itself and has no knowledge of whether it is active or not.

When using this approach, data is only compressed within ZFS, as can be seen on the left side of Figure 9. This has several implications: since data is only compressed when arriving on the file system servers, there are no benefits regarding network throughput. If the network is a bottleneck for overall I/O performance, this can be a significant disadvantage. However,

since compression is performed on the servers, it also can not have an influence on application performance. This influence might be problematic, if I/O is performed asynchronously and in parallel to the computation. Another benefit is that this setup can be deployed without any modifications to Lustre as long as Lustre's ZFS backend is used. Depending on the chosen compression algorithm, CPU overhead might be introduced on the file system servers, which might also have an influence on energy consumption. If particularly slow algorithms are used, they will also negatively impact I/O performance. However, this level of support already enables the cost savings presented in Section 2.2.3.

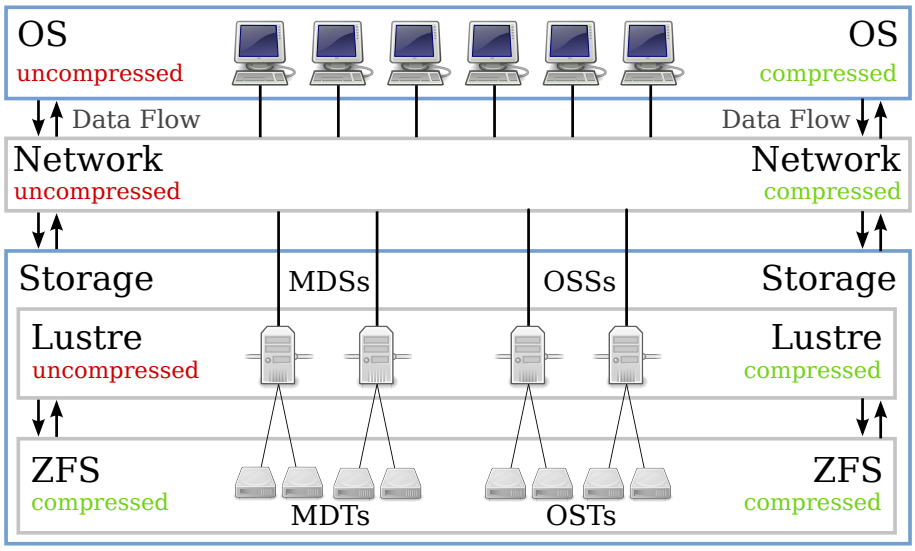


Figure 9. Comparison of different levels of support for compression in Lustre. Left: Compression is only performed on the servers. Right: Compression is performed on the clients

As part of our work within the Intel Parallel Computing Center "Enhanced Adaptive Compression in Lustre," we will add compression support to Lustre. This will include support for performing compression on both the clients and servers. Additionally, compression will be adaptive, that is, adjust itself according to the data to be compressed and several performance metrics. Applications will be able to influence the compression via Lustre's `ladvise` interface. This will allow proper integration with third-party applications and I/O libraries. On the one hand, applications already performing application-specific compression will be able to turn off compression as necessary through this interface. On the other hand, the compression support will be completely transparent to applications and libraries if they do not make use of this interface.

Once finished, this will bring compression support to the level depicted on the right side of Figure 9, that is, Lustre will have information about the compression status and act accordingly. For instance, data compressed on the client can be stored directly on the file system servers without decompression, reducing overhead. However, in case if a long-term archival is desired, the data might also be recompressed using different compression algorithms on the servers for higher space savings. While performing the compression on the clients might negatively influence applications, network throughput can be virtually increased by reducing the amount of data that has to be transferred via the network. However, as this compression will only be performed during the applications' I/O phases, interference should be minimal as long as sufficiently fast compression algorithms are used and/or I/O is not performed asynchronously.

Another major new feature will be adaptive compression. We have started working on support for adaptive compression within ZFS [13]. This support will be further modularized and its

underlying functionality will be used in the Lustre client. Even though ZFS stores the used compression algorithm on a per-record level, it can only be set on a per-file-system level. Because modifying this file system parameter frequently does not make sense and is too coarse-grained, additional functionality is required. While lz4 is an appropriate compression algorithm for high performance I/O, gzip is more suitable for archival purposes due to its higher compression ratios. Lustre uses a single ZFS file system for each OST and MDT, therefore making it impossible to work around the problem using multiple file systems. Consequently, Lustre could either be tuned for high performance or archival, but not both. Adaptive compression allows selecting an appropriate compression algorithm based on different selection criteria during runtime. Based on client-provided information about the desired compression mode (high performance or archival), a cost function can be used to select the best compression algorithm for the current data. Additionally, performance metrics will be included in the decision process to make sure that clients and servers do not get overloaded.

3.2. Application-specific compression

To complement our file system approach, we have started to develop the Scientific Compression Library (SCIL) that allows fine-grained control over expected data accuracy metrics and performance behavior within the DFG-funded project AIMES.[9] Based on user-supplied hints, data properties and system performance characteristics, the library will adaptively choose the compression pipeline (algorithms) on behalf of the user. Therewith, users do not have to predefine the algorithm but can define the parameters that matter from the application point of view. Deploying an improved algorithm can then automatically benefit many existing applications. Amongst other goals, this library will be embedded as a filter into HDF5. We also push domain-specific solutions for icosahedral grids. Integrating approaches on both the application level and within the file system will allow us to maximize the benefits achieved by compression.

4. Conclusion and future work

Based on the results obtained by modeling the impact of compression on performance and costs, applying compression throughout the whole I/O stack is a viable approach to improve performance and decrease costs. Due to new and upcoming high performance compression algorithms, such as lz4fast, it is now possible to compress data at all levels of the storage hierarchy without sacrificing performance or increasing costs. In fact, our analyses have shown that it is possible to reduce costs of large-scale storage systems by up to 50 % without any negative impact on throughput or capacity. If costs are not a concern, compression can be used to improve performance by up to 133 % for network communication.

To leverage these benefits, we will integrate support for adaptive compression in Lustre. This will allow users to increase their file system's capacity without significant performance impacts. However, as the file system is limited in its insight into the applications' data, we will continue to explore new possibilities of application-specific compression with our Scientific Compression Library. Both approaches will interact to avoid unnecessary overhead and increase efficiency.

[9] `https://wr.informatik.uni-hamburg.de/research/projects/aimes/`

We thank Intel Corporation for funding our Intel Parallel Computing Center to integrate enhanced adaptive compression into Lustre. AIMES is supported by the German Research Foundation (DFG) through the Priority Programme 1648 "Software for Exascale Computing."

References

1. CMIP5 – Overview. `http://cmip-pcmdi.llnl.gov/cmip5/`. Last accessed: 2016-04.

2. Mohamed S. Abdelfattah, Andrei Hagiescu, and Deshanand Singh. Gzip on a chip: High performance lossless data compression on fpgas using opencl. In *Proceedings of the International Workshop on OpenCL 2013 & 2014*, IWOCL '14, pages 4:1–4:9, New York, NY, USA, 2014. ACM.

3. Kenneth C. Barr and Krste Asanović. Energy-aware lossless data compression. *ACM Trans. Comput. Syst.*, 24(3):250–291, August 2006.

4. L. Benini, D. Bruni, A. Macii, and E. Macii. Hardware-assisted data compression for energy minimization in systems with embedded processors. In *Design, Automation and Test in Europe Conference and Exhibition, 2002. Proceedings*, pages 449–453, 2002.

5. Jeff Bonwick, Matt Ahrens, Val Henson, Mark Maybee, and Mark Shellenbaum. The Zettabyte File System. 2003.

6. Konstantinos Chasapis, Manuel Dolz, Michael Kuhn, and Thomas Ludwig. Evaluating Power-Performace Benefits of Data Compression in HPC Storage Servers. In Steffen Fries and Petre Dini, editors, *IARIA Conference*, pages 29–34. IARIA XPS Press, 04 2014.

7. Yanpei Chen, Archana Ganapathi, and Randy H. Katz. To compress or not to compress - compute vs. io tradeoffs for mapreduce energy efficiency. In *Proceedings of the First ACM SIGCOMM Workshop on Green Networking*, Green Networking '10, pages 23–28, New York, NY, USA, 2010. ACM.

8. D. J. Craft. A fast hardware data compression algorithm and some algorithmic extensions. *IBM J. Res. Dev.*, 42(6):733–745, November 1998.

9. Sébastien Denvil. The ESGF's organization with a detailed discussion of the CMIP6 project and upcoming challenges. talk, `https://rd-alliance.org/sites/default/files/attachment/RDA-ESGF-2015.pdf`, 2015.

10. Peter Deutsch. DEFLATE Compressed Data Format Specification version 1.3. RFC 1951, 1996.

11. A. Dzhagaryan, A. Milenkovic, and M. Burtscher. Energy efficiency of lossless data compression on a mobile device: An experimental evaluation. In *Performance Analysis of Systems and Software (ISPASS), 2013 IEEE International Symposium on*, pages 126–127, April 2013.

12. ECMA. Standard ECMA-321: Streaming Lossless Data Compression Algorithm – (SLDC). http://www.ecma-international.org/publications/standards/Ecma-321.htm, June 2011.

13. Florian Ehmke. Adaptive Compression for the Zettabyte File System. Master's thesis, Universität Hamburg, 02 2015.

14. Rosa Filgueira, Malcolm Atkinson, Alberto Nuñez, and Javier Fernández. *Euro-Par 2012 Parallel Processing: 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings*, chapter An Adaptive, Scalable, and Portable Technique for Speeding Up MPI-Based Applications, pages 729–740. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

15. Rosa Filgueira, Malcolm Atkinson, Yusuke Tanimura, and Isao Kojima. *Euro-Par 2014 Parallel Processing: 20th International Conference, Porto, Portugal, August 25-29, 2014. Proceedings*, chapter Applying Selectively Parallel I/O Compression to Parallel Storage Systems, pages 282–293. Springer International Publishing, Cham, 2014.

16. Rosa Filgueira, David E. Singh, Alejandro Calderón, and Jesús Carretero. CoMPI: Enhancing MPI Based Applications Performance and Scalability Using Run-Time Compression. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 207–218, Berlin, Heidelberg, 2009. Springer-Verlag.

17. Rosa Filgueira, David E. Singh, Jesús Carretero, Alejandro Calderón, and Félix García. Adaptive-Compi: Enhancing Mpi-Based Applications - Performance and Scalability by Using Adaptive Compression. *Int. J. High Perform. Comput. Appl.*, 25(1):93–114, February 2011.

18. Nathanel Hübbe and Julian Kunkel. Reducing the HPC-Datastorage Footprint with MAFISC – Multidimensional Adaptive Filtering Improved Scientific data Compression. *Computer Science - Research and Development*, pages 231–239, 05 2013.

19. Intel High Performance Data Division. Lustre – The High Performance File System, 2013.

20. J. Kane and Q. Yang. Compression speed enhancements to lzo for multi-core systems. In *Computer Architecture and High Performance Computing (SBAC-PAD), 2012 IEEE 24th International Symposium on*, pages 108–115, Oct 2012.

21. Meaza Taye Kebede. Performance Comparison of Btrfs and Ext4 Filesystems. Master's thesis, University of Oslo, 2012.

22. Kush K. Kella and Aasia Khanum. APCFS: Autonomous and Parallel Compressed File System. *International Journal of Parallel Programming*, 39(4):522–532, 2010.

23. Peter Kogge, Keren Bergman, Shekhar Borkar, Dan Campbell, William Carlson, William Dally, Monty Denneau, Paul Franzon, William Harrod, Kerry Hill, Jon Hiller, Sherman Karp, Stephen Keckler, Dean Klein, Robert Lucas, Mark Richards, Al Scarpelli, Steven Scott, Allan Snavely, Thomas Sterling, R. Stanley Williams, and Katherine Yelick. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, DARPA report. http://www.cse.nd.edu/Reports/2008/TR-2008-13.pdf, Sep 2008.

24. Rachita Kothiyal, Vasily Tarasov, Priya Sehgal, and Erez Zadok. Energy and Performance Evaluation of Lossless File Data Compression on Server Systems. In *Proceedings of SYSTOR 2009: The Israeli Experimental Systems Conference*, SYSTOR '09, pages 4:1–4:12, New York, NY, USA, 2009. ACM.

25. Julian Kunkel, Michael Kuhn, and Thomas Ludwig. Exascale Storage Systems – An Analytical Study of Expenses. *Supercomputing Frontiers and Innovations*, pages 116–134, 06 2014.

26. Sriram Lakshminarasimhan, Neil Shah, Stephane Ethier, Seung-Hoe Ku, Choong-Seock Chang, Scott Klasky, Rob Latham, Rob Ross, and Nagiza F Samatova. ISABELA for effective in situ compression of scientific data. *Concurrency and Computation: Practice and Experience*, 25(4):524–540, 2013.

27. P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, Sept 2006.

28. Peter Lindstrom. Fixed-Rate Compressed Floating-Point Arrays. *Visualization and Computer Graphics, IEEE Transactions on*, 20(12):2674–2683, 2014.

29. Dirk Meister, Jürgen Kaiser, Andre Brinkmann, Michael Kuhn, Julian Kunkel, and Toni Cortes. A Study on Data Deduplication in HPC Storage Systems. In *Proceedings of the ACM/IEEE Conference on High Performance Computing (SC)*, 11 2012.

30. Dutch T. Meyer and William J. Bolosky. A study of practical deduplication. In *Proceedings of the 9th USENIX Conference on File and Stroage Technologies*, FAST'11, pages 1–1, Berkeley, CA, USA, 2011. USENIX Association.

31. Nitin Gupta. zram: Compressed RAM based block devices. `https://www.kernel.org/doc/Documentation/blockdev/zram.txt`, 11 2015. Last accessed: 2016-04.

32. Ritesh A Patel, Yao Zhang, Jason Mak, Andrew Davidson, and John D Owens. *Parallel lossless data compression on the GPU*. IEEE, 2012.

33. P. Ratanaworabhan, Jian Ke, and M. Burtscher. Fast lossless compression of scientific floating-point data. In *Data Compression Conference (DCC'06)*, pages 133–142, March 2006.

34. Christopher M. Sadler and Margaret Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, SenSys '06, pages 265–278, New York, NY, USA, 2006. ACM.

35. The Green500 Editors. Green500. `http://www.green500.org/`, 2016. Last accessed: 2016-04.

36. The TOP500 Editors. TOP500. `http://www.top500.org/`, 06 2014. Last accessed: 2016-04.

37. Ning Wang, Jian-Wen Bao, Jin-Luen Lee, Fanthune Moeng, and Cliff Matsumoto. Wavelet Compression Technique for High-Resolution Global Model Data on an Icosahedral Grid. *Journal of Atmospheric and Oceanic Technology*, 32(9):1650–1667, 2015.

38. Benjamin Welton, Dries Kimpe, Jason Cope, Christina M. Patrick, Kamil Iskra, and Robert Ross. Improving I/O Forwarding Throughput with Data Compression. In *Proceedings of the 2011 IEEE International Conference on Cluster Computing*, CLUSTER '11, pages 438–445, Washington, DC, USA, 2011. IEEE Computer Society.

39. R.N. Williams. An extremely fast Ziv-Lempel data compression algorithm. In *Data Compression Conference, 1991. DCC '91.*, pages 362–371, Apr 1991.

40. Rong Xu, Zhiyuan Li, Cheng Wang, and Peifeng Ni. Impact of data compression on energy consumption of wireless-networked handheld devices. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 302–311, May 2003.

41. Yann Collet. lz4. `http://www.lz4.org/`, 04 2016. Last accessed: 2016-04.

42. J. Ziv and A. Lempel. A universal algorithm for sequential data compression. *Information Theory, IEEE Transactions on*, 23(3):337–343, May 1977.