



Lustre I/O performance investigations on Hazel Hen: experiments and heuristics

Marco Seiz¹ · Philipp Offenhäuser^{2,3} · Stefan Andersson^{4,5} · Johannes Hötzer^{1,6} · Henrik Hierl¹ · Britta Nestler^{1,6} · Michael Resch⁷

Accepted: 8 March 2021 / Published online: 9 April 2021
© The Author(s) 2021

Abstract

With ever-increasing computational power, larger computational domains are employed and thus the data output grows as well. Writing this data to disk can become a significant part of runtime if done serially. Even if the output is done in parallel, e.g., via MPI I/O, there are many user-space parameters for tuning the performance. This paper focuses on the available parameters for the Lustre file system and the Cray MPICH implementation of MPI I/O. Experiments on the Cray XC40 Hazel Hen using a Cray Sonexion 2000 Lustre file system were conducted. In the experiments, the core count, the block size and the striping configuration were varied. Based on these parameters, heuristics for striping configuration in terms of core count and block size were determined, yielding up to a 32-fold improvement in write rate compared to the default. This corresponds to 85 GB/s of the peak bandwidth of 202.5 GB/s. The heuristics are shown to be applicable to a small test program as well as a complex application.

Keywords High-performance computing · MPI I/O · Parallel I/O · Lustre · Striping

✉ Marco Seiz
marco.seiz@kit.edu

¹ Institute of Applied Materials (IAM), Karlsruhe Institute of Technology (KIT), Straße am Forum 7, 76131 Karlsruhe, Germany

² Numerical Methods and Libraries, High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Nobelstrasse 19, 70569 Stuttgart, Germany

³ Hewlett Packard Enterprise (HPE), Herrenberg Straße 140, 71034 Böblingen, Germany

⁴ Cray Computer Deutschland GmbH, Munich, Germany

⁵ Amazon Web Services EMEA SARL, Niederlassung Deutschland, Marcel-Breuer-Str. 12, 80807 München, Germany

⁶ Institute for Digital Materials (IDM), Hochschule Karlsruhe Technik und Wirtschaft (HSKA), Moltkestraße 30, 76131 Karlsruhe, Germany

⁷ High Performance Computing Center Stuttgart (HLRS), University of Stuttgart, Nobelstrasse 19, 70569 Stuttgart, Germany

1 Introduction

While current supercomputers provide hundreds of PFLOP/s [1], the speed of I/O has not grown as much. Furthermore, often applications do not fully exploit the parallelism of parallel file systems. Especially for voxel-based simulations on regular grids, the data required for checkpointing grows proportionally with the domain size. Hence, running large-scale simulations often implies large amounts of data, which need to be written to disk, necessitating high-performance I/O for current and future HPC applications. Such applications find use in many research areas, e.g., engineering, weather and climate research or material science, as they allow a deeper insight into the physical mechanisms and give predictions for future processes and design of process chains. The focus of past HPC optimizations is primarily found in computation and communication, whereas I/O tends to be neglected in optimization efforts. Considering all this, investigating and improving I/O for real applications are of great importance. The objective of this paper is to give the end-user as well as the application developer heuristics to tune the striping on Lustre file systems such that high-performance I/O is achieved. This paper first re-capitulates the state of the art in parallel I/O and then describes the software and hardware used for the performance measurements. Experiments are then conducted in which the time taken for a parallel write call to finish is measured, investigating the parameters of block size, processor count and striping configuration. Based on the analysis of the experimental results, heuristics employing the varied parameters are determined such that the write rate is substantially improved, up to a factor of 32. Finally, it is shown that the heuristics are transferable between similar I/O strategies as well as between similar Lustre setups.

2 State of the art in parallel I/O

The I/O stack of a parallel application is shown in Fig. 1. On the top of the parallel I/O stack, an application solves a numerical problem on a multi-dimensional discrete grid. At each grid point, a number of quantities such as velocity, concentrations, temperature and order parameters may be stored. The data structure fits to the numerical model and to the data structure of the code, not to the pattern in the file and the hardware where the data is actually stored.

High-level I/O libraries such as HDF5 or PnetCDF may be employed to facilitate data exchange with other scientists. Below this layer, MPI I/O provides the portable middleware for parallel file I/O. MPI I/O is part of the MPI-2 [2] standard and was introduced in 1997. The I/O forwarding layer bridges the gap between the application and the file system and may aggregate the I/O. Parallel file systems show a single, unified high-performance storage space while abstracting away many storage devices and servers. While there are many different parallel file systems, almost only Lustre and GPFS/IBM Spectrum Scale are

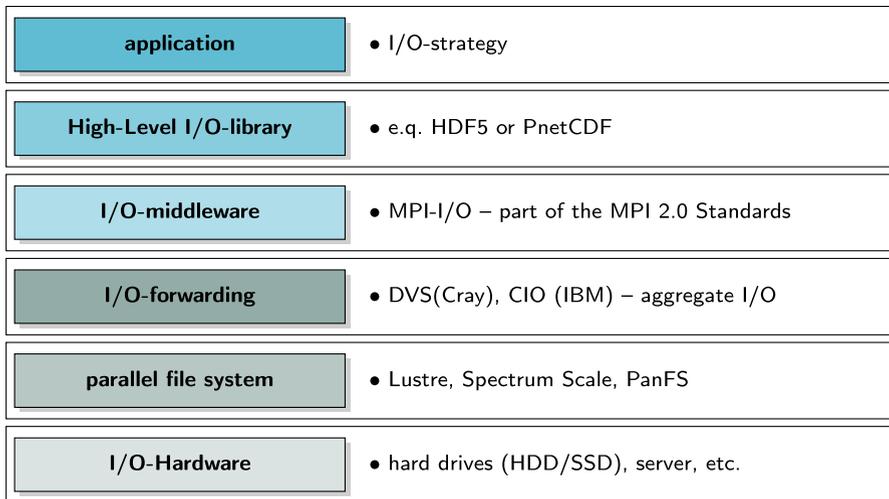


Fig. 1 I/O stack of a parallel application. The higher levels are designed to provide the developer with an interface to implement I/O, while the lower levels are designed to maintain the access to the hardware [4]

used in the top 100 supercomputers [1]. The focus of this paper is on Lustre as it allows the end-user to manipulate the striping of files, which GPFS does not, thus obviating the need to investigate. At the very bottom of the I/O stack is the actual hardware, classical spinning hard disk drives (HDD) or solid-state drives (SSD), where user data is actually stored [3, 4].

Almost all of the above-mentioned layers have adjustable parameters in order to enhance I/O performance. In this paper, we are primarily concerned with the parameters available to the user. Similar concerns were investigated by [5–7]. In [5] experiments were conducted on the Texas Advanced Computing Center’s Stampede employing Lustre. The authors investigated the influence of stripe count, number of aggregators and stripe size for a fixed number of processes and file size. They found that if the stripe count and the number of aggregators are not chosen appropriately together, performance drops abruptly. This problem is avoided by choosing the number of aggregators greater than or equal to the stripe count. No significant influence of stripe size was found in [5] for the chosen process count and file size. Based on these findings, the authors implemented a parallel I/O library to improve I/O performance called “TACC’s Terrific Tool for Parallel I/O.” In [6], the authors investigated the performance of the HDF5 and NetCDF-4 parallel I/O libraries on a Lustre system. Among other things, it was found that the highest I/O performance was found at transfer sizes as big or bigger than the stripe size for both libraries. The authors of [7] conducted experiments on various systems with several I/O benchmarks. The experiments form a base for a performance model employed in an autotuning framework to optimize I/O performance. Parameters include the file size, stripe size, stripe count and number of aggregators for a fixed process count.

2.1 I/O access patterns

The I/O access pattern determines the possible performance of a parallel I/O application. Thakur et al. [8] defined various I/O access patterns and provided a classification of the different ways to implement I/O with MPI I/O. They classify the access patterns into four levels, *level 0* to *level 3*, and explain why users should implement *level 3* MPI I/O access patterns for performance reasons. In the following, we represent the classification of I/O access patterns and discuss the advantage and disadvantage of three typical I/O access patterns for parallel applications.

2.1.1 Classification of I/O patterns

Based on the work of [8], the four levels of parallel I/O are recapitulated. In *level 0*, the application uses Unix-style I/O. Each process does independent I/O: To write a local array to disk each process will perform an independent write for each row in the local array. *Level 1* uses collective I/O functions: All processes write in a shared file by using a collective call without the knowledge of what the other processes do. To describe the non-contiguous file access pattern in the *level 2* file access, each process creates an MPI-derived data type, defines a file view and performs independent write calls. Similar to *level 2*, MPI-derived data types in *level 3* are used to describe the non-contiguous access pattern and a file view is defined, but a collective write call is used to perform the I/O.

2.1.2 Mapping data onto the file system

There are mainly three methods of mapping file I/O calls onto a file system in parallel programs. In the first method, each MPI process creates and writes to a separate file, the one-file-per-process I/O method or the N:N model as depicted in Fig. 2a. The implementation of this method is simple because no MPI communication is needed. The drawback of this method is that a large number of files are hard to manage [3]. Furthermore, the method does not scale to a large number of MPI processes as the number of metadata operations for the file creation is a bottleneck and the number of simultaneous disk accesses creates contention for file system resources. Recently, newer parallel filesystems, e.g., GekkoFS [9] or DAOS [10], are being developed in order to alleviate these problems, but they have not yet found widespread adoption. Adding onto this, reading the data back into the application is only simple for the same number of processes. Reading the data back into the application with a different number is complicated and error-prone [3]. The second method is the so-called spokesperson model or the 1:1-model. One MPI process receives all data from the other MPI processes and writes it to one file in the file system as shown in Fig. 2b. This approach leads to a poor performance for a large number of MPI processes because of the communication congestion of the all-to-one communication pattern. A second limiting factor is the memory space available for one process to handle all the data of the other MPI processes [3]. Even worse, one process cannot saturate the available bandwidth of a parallel file system. Because of the mentioned drawbacks, the spokesperson pattern will not scale, i.e., the time

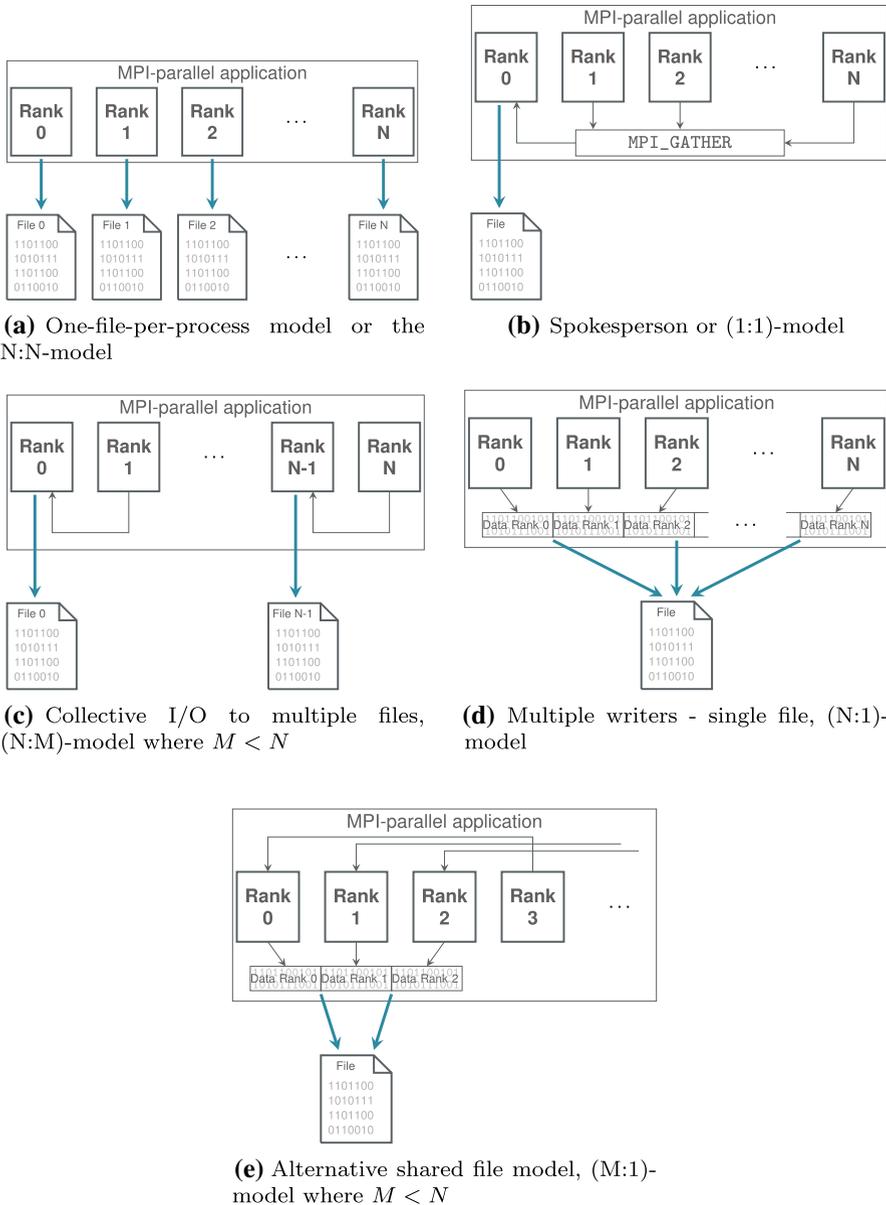


Fig. 2 Different I/O access patterns for I/O in MPI-parallel applications [3]

increases linearly with the amount of data and increases with the number of MPI processes. In order to alleviate some of the performance issues, it is possible to define groups of processes, which aggregate the data to one process (master) within a group. Each group master writes a separate file, which leads to a N:M model with $M < N$, as shown in Fig. 2c. This approach increases I/O performance relative to the

1:1 model, but incorporates problems of the N:N model such as increasing number of metadata operations and a more complex implementation. Parallel I/O to a shared file performed by all MPI processes as depicted in Fig. 2d can overcome the limitation of the one-file-per-process model and the spokesperson model. For the single shared file (SSF) model (N:1 model), one file is opened by all MPI processes and each MPI process performs I/O to a unique portion of the single file. The file could be physically distributed among disks, but appears to the program as a single logical file. With sufficient I/O hardware, a parallel file system and an efficient MPI I/O implementation, this model scales to a large number of MPI processes [3]. Figure 2e shows a modification of the SSF model, in which only a subgroup of MPI processes performs the I/O to the SSF.

Besides mapping the data per checkpoint onto files, there is another degree of freedom involved in how to store time series. This can be done in one file per checkpoint approach, e.g., VTK, or with many checkpoints per file as employed in the proprietary format used by PACE3D [11].

2.2 Two-phase MPI I/O

In collective I/O, all MPI processes, or a subgroup of processes, perform the I/O operations in one collective MPI call and provide a comprehensive view of the data movement over all processes. With the SSF model, shown in Fig. 2d, e, collective I/O is possible. A very successful process collaboration strategy is the two-phase I/O. It was first proposed by [12] and is used in the MPI I/O implementation ROMIO [8]. In the first phase, called the *request aggregation phase* or *communication phase*, a subset of MPI processes is selected as I/O aggregators. The file is divided into non-overlapping sections, called *file domains*, and each file domain is assigned to an aggregator. The non-aggregators send their data to write or request to read to the aggregators. In the second phase, called the *file access phase* or *I/O phase*, the aggregators perform the write or the read operations on the file system [3, 8].

Figure 3 exemplarily shows how the two-phase I/O works and how it improves the performance of the I/O. A 5×8 two-dimensional array is partitioned by a block–block pattern among four MPI processes. As each local array is non-contiguous in the file, if every MPI process were to write their portion of the file independently many small and non-contiguous file operations would be necessary. A collective I/O operation circumvents this by collecting data from all MPI processes to a smaller number of aggregators, which write to contiguous blocks in the file and thus improving I/O performance.

3 Experimental environment

In this section, we will describe the hardware and the software environment on which all experiments were performed.

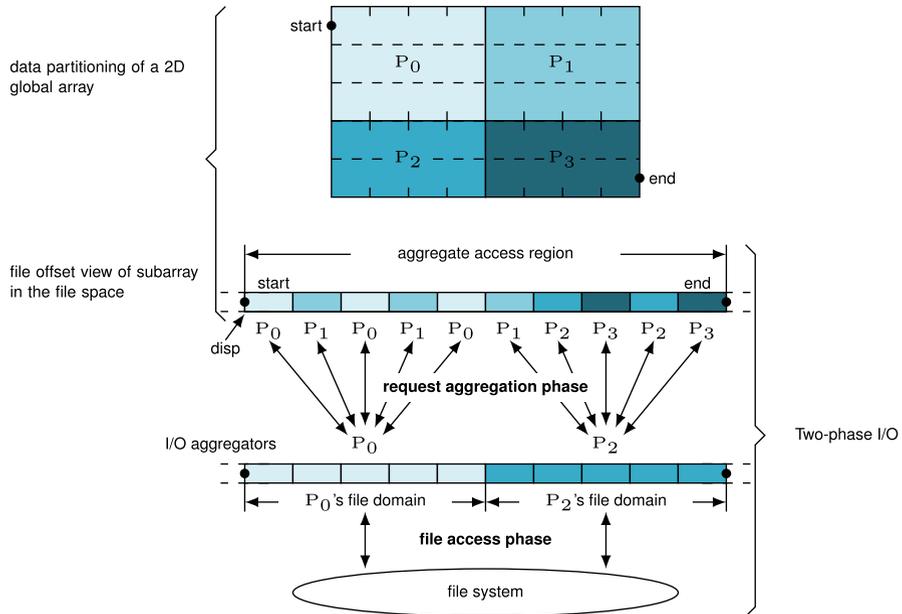


Fig. 3 A collective file operation with a two-phase I/O for a 5×8 two-dimensional array, distributed among four MPI processes in a block-block pattern and the data layout in the file. The MPI processes 0 and 2 are selected as I/O aggregators [3]

3.1 Cray XC40 Hazel Hen

All experiments are performed on the Cray XC40 supercomputer (Hazel Hen) at the High-Performance Computing Center Stuttgart (HLRS). The system consists of 7712 compute nodes, each with two sockets and 128 GiB of main memory. The nodes are connected via the high-performance Cray Aries interconnect. Each socket is equipped with a 12-core Intel® Xeon® E5-2680V3 (Haswell) processor with a base frequency of 2.5 GHz, so each node has 24 cores. This leads to a homogeneous massively parallel system with 185 088 compute cores and approximately 987 TiB of main memory. The Cray XC40 at HLRS has a peak performance of 7.4 PFLOP/s and 5.64 PFLOP/s in the LINPACK-benchmark. In the HPCG benchmark, the system at the HLRS reached 0.138 PFLOP/s. The HPC system is connected to three Lustre file systems with a capacity of about 13.5 PiB in total. The technical details of the file system on which the experiments were performed are presented in the next section. The employed MPI I/O implementation is the Cray MPI-I/O, which is based on the MPICH ROMIO implementation [8].

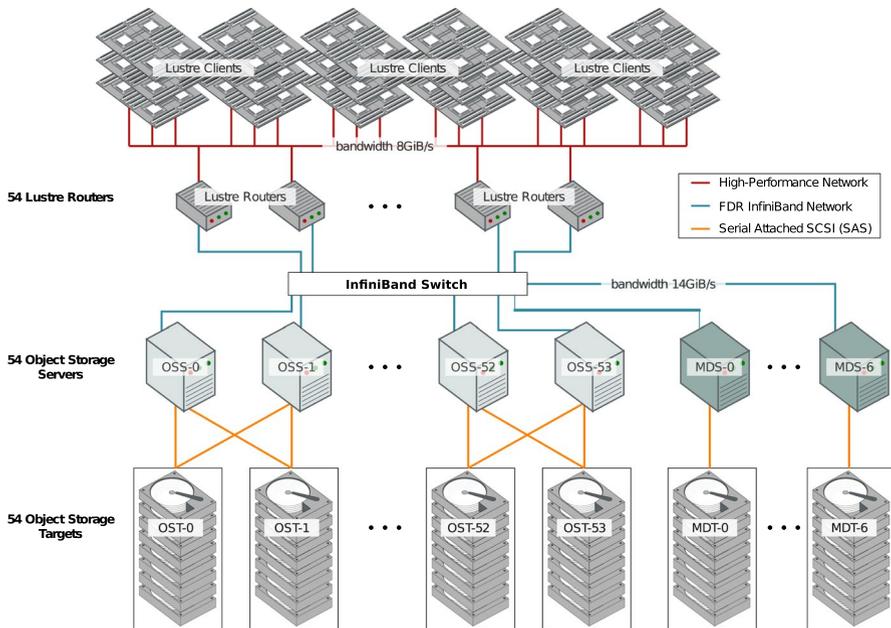


Fig. 4 Configuration of the Lustre file system ws_9 at the HRLS

3.2 The Lustre file system at HLRS

The Cray XC40 supercomputer at the HLRS is connected to three Lustre file systems. In this section, we only present the technical details of the file system ws_9 , which we used for the experiments, and explain the basic options to adapt the Lustre file system settings to the I/O pattern of the application. ws_9 is a Lustre-based file system, specifically a Cray Sonexion 2000 Data Storage system. The schema of the file system configuration and its connection to the Cray XC40 are shown in Fig. 4. Each compute node runs a Lustre client in order to access the Lustre file system.

The compute nodes are connected via the high-performance Cray Aries interconnect with a maximum bandwidth of 8 GiB/s for node-to-node communication. Hazel Hen has several service nodes, which connect to both the Aries high performance network (HPN) and to external networks. Fifty-four of these service nodes are used as Lustre routers (LNET routers). The LNET routers are connected to an InfiniBand switch as are the 54 Object Storage Servers (OSS). Each OSS has one Object Storage Targets (OST) with a capacity of about 169 TiB. The connection between the LNET routers, the InfiniBand switch and the InfiniBand switch with the Object Storage Servers is based on an FDR InfiniBand network with a bandwidth of 14 GiB/s. The number of OSTs sets a limit on the amount of parallel I/O throughput that can be achieved, which would ideally be the throughput which one OST can achieve times the number of OSTs.

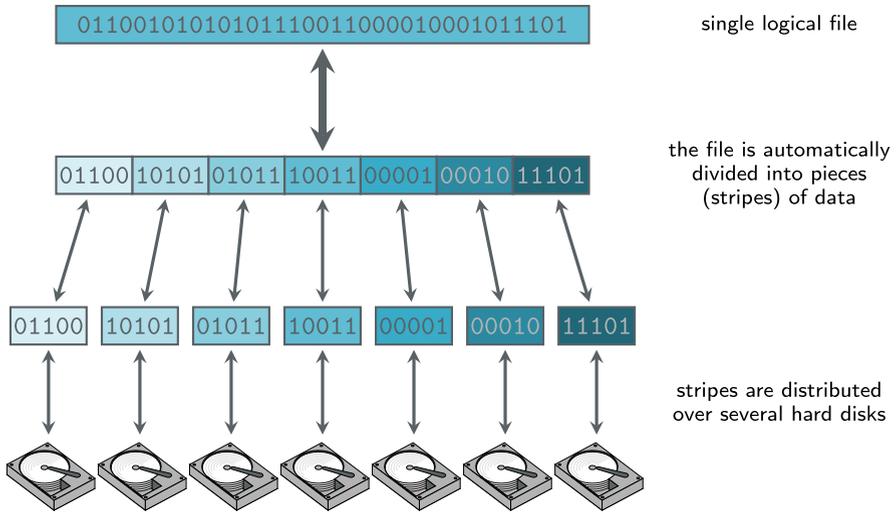


Fig. 5 A file is automatically divided into pieces and distributed among a number of OSTs (file striping). Users can adapt the size of the pieces (stripe size) and the number of OSTs on which the file is being distributed (stripe count) [13]

The data of the files are stored in the OSTs. Each OST consists of 41 HDDs and is connected to a GridRaid which is a technique for distributing data across multiple HDDs. The OSS provides file I/O services and network request handling for OSTs [13]. Two OSS and two OST together build one Scalable Storage Unit (SSU). For failover inside of a SSU, an OSS is connected via Serial Attached SCSI (SAS) to both OSTs. If one of the OSS in a SSU fails, the other OSS takes control of both OSTs. One SSU has a throughput of 7.5 GB/s read and write when using the IOR (Interleaved or Random) benchmark, respectively, one OST has a bandwidth of 3.75 GB/s [14]. As there are 54 OSTs available on *ws9*, the maximum available I/O bandwidth is 202.5 GB/s. A GridRaid is used to reduce the rebuild time in case of drive failure and to improve the performance when running in degraded state [14].

To get parallelism, and thus I/O performance, the file is distributed among a number of OSTs, which is called file striping. Figure 5 shows the principal idea of file striping. The user can define, for directories or for a single file, the size of the pieces into which the file is divided (stripe size) and the number of OSTs in which the file is to be distributed (stripe count). The stripe size and the stripe count are the basic options for the user to adapt file system setting to the I/O pattern of their application and improve the I/O performance. The stripe count can be varied from 1 to the maximum number of OSTs (54), whereas the stripe size can be varied between 64 KB and 4 GB in increments of 64 KB [13]. The number of aggregator nodes during two-phase I/O is automatically set to $\text{stripecount} \times \text{multiplier}$ by the Cray MPI-I/O library.

The metadata of the files are stored in seven Metadata Targets (MDT), and seven Metadata Servers (MDS) provide access for the Lustre clients to the metadata.

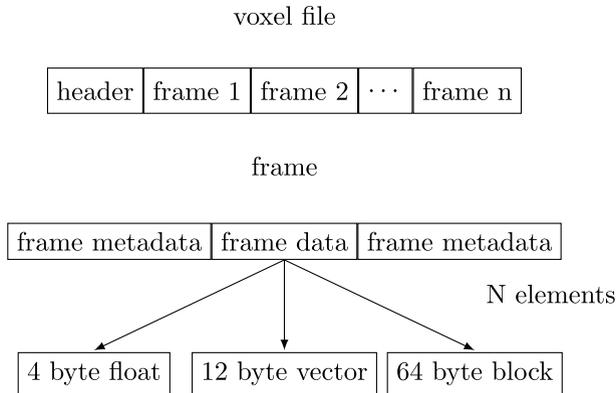


Fig. 6 The voxel format used by PACE3D. Each file consists of a header followed by frames representing the state of a field at a certain timestep. Each frame consists of metadata, followed by the actual field represented as N elements of either a 4-byte float, a 12-byte vector or a 64-byte block

4 Methods

In this section, the HPC application PACE3D and its I/O method are described. Following this, a small test application mimicking the I/O method of PACE3D is described. Finally, the experiments and measurements are detailed.

4.1 The PACE3D framework

The massive parallel PACE3D framework (“Parallel Algorithms for Crystal Evolution in 3D”) [11] is developed to study the microstructure evolution in different materials. The simulation framework is based on the phase-field method [15] and contains multi-physical coupling to fields such as temperature, concentration or stresses to include their effect on microstructure evolution. The solver contains modules for diffuse interface approaches (Allen–Cahn, Cahn–Hilliard) [15], grain growth [16–18], grain coarsening [19, 20], sintering [21, 22], solidification [23, 24], mass and heat transport, fluid flow (Lattice–Boltzmann, Navier–Stokes) [25], mechanical deformation (elasticity, plasticity) [26–28], magnetism [29], electrochemistry [30] and wetting [31, 32]. The equations are discretized in space with a finite difference scheme and in time with different time integration methods such as explicit Euler schemes and implicit Euler via conjugated gradient methods [22, 33]. To efficiently compute the complex evolution equations, the solver is parallelized using domain decomposition based on MPI. Selected models are also manually vectorized and achieve up to 32.5% single-core peak performance on the Hazel Hen as shown in [22]. Explicit kernels within the solver scale up to 98304 cores with 97% parallel efficiency [22]. In [22], early results of the present paper were used to massively improve I/O performance for large-scale simulations.

The framework utilizes several proprietary data formats in order to store the 3D voxel-based data. The principal structure is shown in Fig. 6: Each file has a header containing general data, and each checkpoint (henceforth called frame) consists of

frame-specific metadata, the data frame itself and an additional another block of frame-specific metadata. The data frame contains the data for each voxel, which may be one of the following: A single-precision floating point value (4 bytes), a single-precision floating point vector (12 bytes) or a block of phase-field values (64 bytes). For all kinds of data, the write process of each file works as follows: The first part of the metadata is written by a single process. Afterwards, each process converts its double-precision data to single-precision data and writes it into an output buffer. During this conversion, further field- and frame-specific data are also determined. Whenever the output buffer is filled entirely during this process, the buffer is flushed to the disk via `MPI_File_write_all`. The output buffer is scaled by the local amount of data, ensuring that `MPI_File_write_all` is called the same number of times from each MPI rank. Once the field has been traversed entirely, the output buffer is flushed in parallel and the final part of the metadata is written by a single process. For the PACE3D measurements, the output buffer size is equal to the biggest local output size, and hence, there is only one call to `MPI_File_write_all` per frame.

The write time is measured via `clock_gettime` calls placed around the entire I/O function. Thus, the results for PACE3D include the double-to-float conversion and its accompanying memory operations as well the actual I/O operations.

4.2 mpiotester

In order to decouple the I/O from any application effects, a small test program called `mpiotester` is developed. The goal of this program is to replicate the I/O pattern of the HPC application PACE3D without suffering from start-up and calculation noise. Its source code can be found at:

<https://git.scc.kit.edu/xt5201/mpiotester/-/tree/master>

It allows both reading and writing to files on a level-3 I/O pattern with different I/O styles as explained in Appendix. The resulting times can either be output per process or aggregated into a single average time. For the investigation of independent writes by single processes, a header may also be written and timed alongside the normal output.

The write time measurement is done via `MPI_Wtime` calls around the relevant sections, allowing per-process timing. For the following results, the relevant section is simply the call to `MPI_File_write_all`. Thus, the time spent in this call is measured, which is also the time an end-user would want to minimize, or conversely, maximize the write rate.

4.3 Measurement details

The previously described applications measure the time elapsed during their specific I/O actions. Since the data written to disk is known, the write rate (WR) can be calculated with $WR = \frac{\text{data written}}{\text{time elapsed}}$. Note that the measurement does not include the file opening and closing since this is done only once for many writes. If the write operation is only conducted once per file as in the VTK format, the opening and closing

Table 1 Investigated parameters and their respective limits

Parameter name	Minimum	Maximum
Process count	1	49,152
Block size in KB	64	64,000
Stripe count	1	54
Stripe size in KiB	64	1,048,576
Multiplier	1	128

have to be included in the measurements as well as shown in Appendix A. Besides the application-specific timers, Cray MPICH provides per-file write rates which are enabled via the environment variable `MPICH_MPIIO_TIMERS=1`. Further statistics per file are gathered with `MPICH_MPIIO_STATS=1`. Generally there is a significant difference between the write rates calculated via the timers and those reported by the Cray MPICH library. However, the differences are of a systematic kind and show a similar response behavior when a parameter is changed. Thus, the heuristics derived based on the results of one timer are similar to those of the other timers. In the rest of this paper, the application-specific timer is chosen since it can isolate the write time for repeated writes into the same file, which allows the gathering of more data per run.

The measurements are first collected by a series of scripts and then aggregated into a SQLite database. This database is accessed via a Jupyter notebook from which the analysis is done as well. Reduced¹ versions of the databases and Jupyter notebooks employing them are available at:

<https://git.scc.kit.edu/xt5201/io-on-hazelhen/-/tree/master>

The following results section will contain variation of several I/O parameters: The number of employed processes, the local domain size per process (block size), the stripe count, the stripe size and the aggregator node multiplier in the novel Lustre Lock-ahead (LLA) locking mechanism [34]. The ranges of the parameters can be found in Table 1, but note that the space spanned by these was subsampled based on prior results. The experiments without LLA were conducted at least 30 times per configuration at varying times of day and weekdays, with each run containing 20 write operations (frames). Between each frame a single time step of a simple, explicit kernel is calculated with this time not being measured since calculation time is not of interest in this paper. The rank-wise write rates for each frame are averaged by taking their median, which yields 20 values of write rates per run. As explained in Appendix, the first timing of these is dropped with the rest providing 19 data points per run; hence, each point is the median over at least 570 values. For the experiments with LLA and thus large core counts, only 1-3 runs were conducted per configuration due to time constraints, which is reflected in their much larger confidence intervals.

¹ The median over the process rank has already been calculated to reduce the size of the database.

The investigated local domain sizes range from 20^3 voxels to 200^3 voxels per process, which represent the span of domain sizes typically employed in PACE3D. With one double-precision float per voxel, the data written per process range between 64 KB and 64000 KB, which will henceforth be called block size. The lower end of block sizes might very well stay cached during the experiments. But this in turn means that the user does not need to care about I/O configuration as the data will be flushed in the background while computation resumes.

5 Results

In the following, the experiments conducted with both the `mpiootester` and PACE3D are detailed. First, the simpler `mpiootester` is used to investigate the general write performance. It is shown that the default striping is insufficient to fully exploit the parallel filesystem, especially for simulations employing more than one compute node. The striping configuration in terms of stripe count and stripe size is varied for different experimental configurations in terms of the block size and processor count. Based on these results, heuristics for both the stripe count and stripe size are derived. With both the stripe count and stripe size heuristics determined, the influence of the multiplier with active LLA is investigated. Finally, the derived heuristics are afterwards validated by employing PACE3D, and satisfactorily matching is observed.

5.1 `mpiootester` performance

This section will show the performance results on `ws9` on Hazel Hen with plots showing the median performance and its 95% confidence interval (CI), unless noted otherwise. Confidence intervals are employed for visual checks on whether the parameter variation caused significant differences or not. We note that the measurements were done at a time when `ws9` was not commonly accessible, i.e., the measurements were largely done in isolation in terms of file I/O but were still affected by network noise. A thorough analysis of the raw data is given in Appendix, determining vital information on the distribution to enable quantitative analysis.

5.1.1 Stripe count

The single `-node` write performance of the default striping on `ws9` is investigated by running the application described in section `mpiootester` with the single-file output style and without a header. The default striping consists of a stripe count of 8 OSTs as well as a stripe size of 1 MiB. In Fig. 7, the influence of the data written per MPI process, henceforth called block size, is shown. A general trend of increasing performance with increasing block size can be seen, with many configurations showing only small but still significant improvements at block sizes exceeding 4096 KB. Furthermore, at least 16 cores seem to be required to saturate the write rate for a given block size.

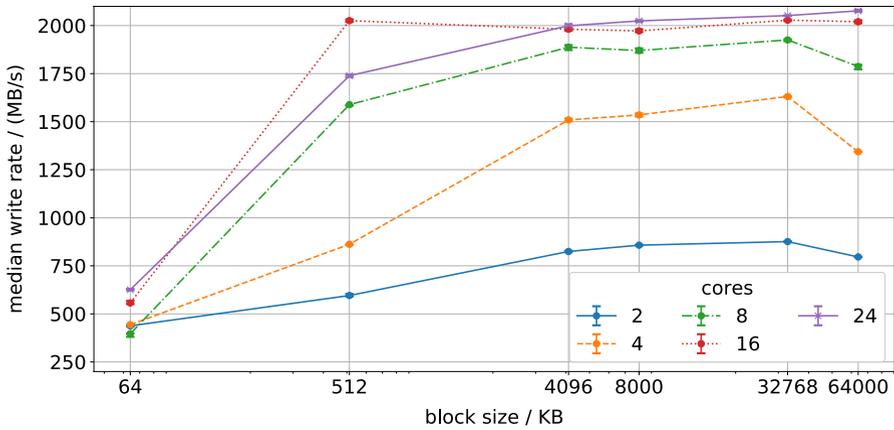


Fig. 7 Single-node performance of `mpiotester`, log-scaled x-axis. An increase in block size yields higher write performance up to a block size of 4096 KB, beyond which the effect is small but significant. At least 16 cores out of 24 seem to be necessary in order to saturate the write rate for a given block size

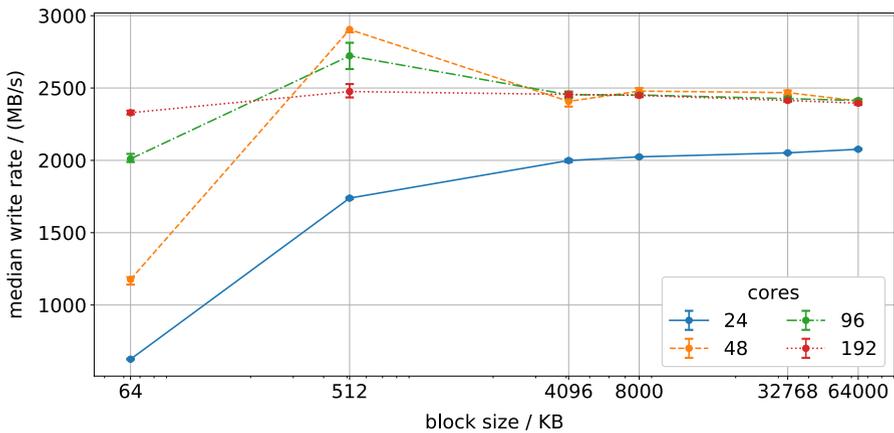


Fig. 8 Multi-node performance of `mpiotester`, log-scaled x-axis. The single-node performance is strictly below that of any multi-node performance, and thus, the bandwidth was not fully utilized in the single-node case. However, the write rate does not scale beyond 48 cores

Going on to look at multi-node performance, the write rate increases as shown in Fig. 8 with more than one node. Hence, the bandwidth was not fully utilized in the single-node case. However, the write rate does not scale beyond 48 cores. In order to achieve I/O scaling for multiple nodes, it is necessary to adjust the striping of the file. The striping consists of both the stripe count and the stripe size, of which the stripe count will be investigated first. This is done by varying the stripe count on a single node in order to determine a heuristic for the number of OSTs per node for good performance. Figure 9 shows the results of this study for 24 cores, indicating a performance peak at 8 OSTs.

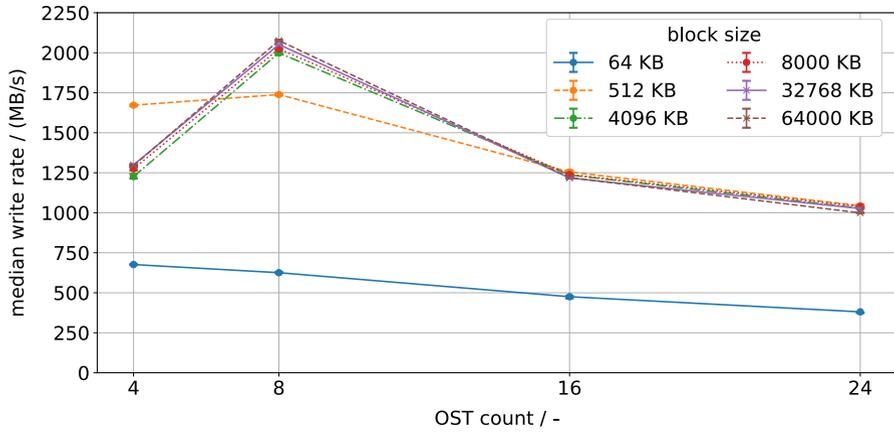


Fig. 9 Single-node performance of `mpiootester` with variable striping count employing a full node corresponding to 24 cores. A performance peak is evident for block sizes above 64 KB. All of the results of the block sizes above 512 KB are very close together. This suggests that the peak is independent of block size for block sizes above of 512 KB

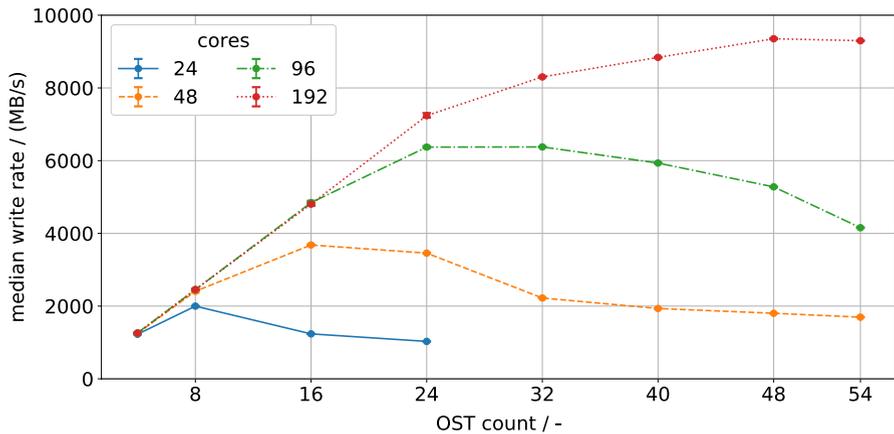
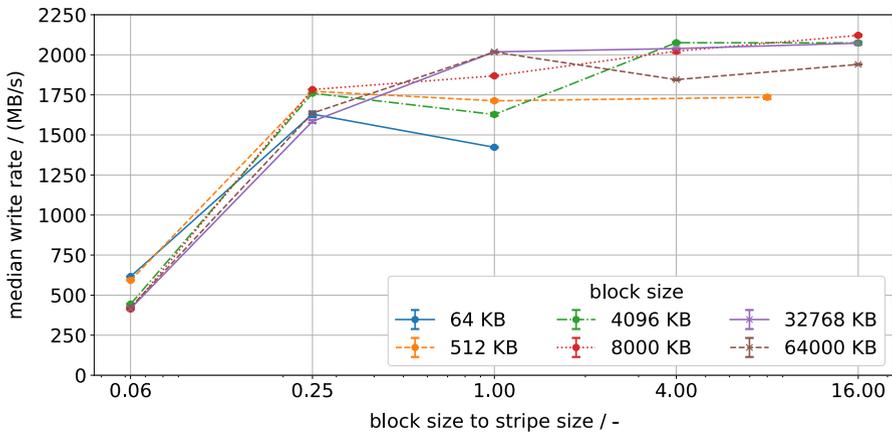
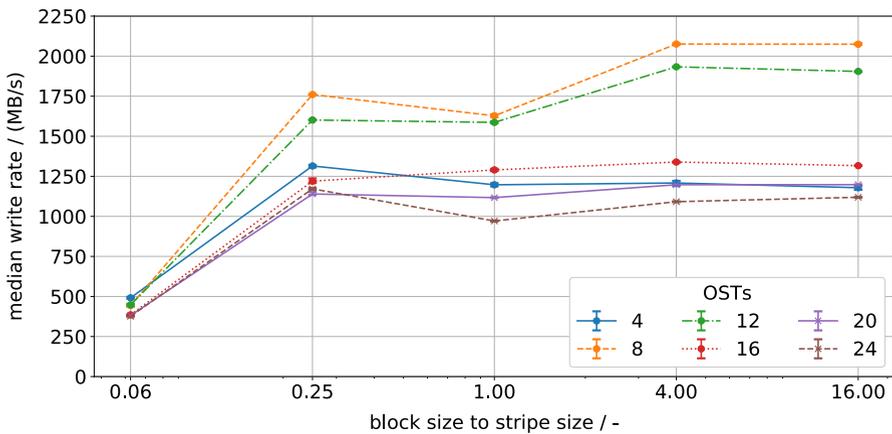


Fig. 10 Multi-node performance of `mpiootester` with variable striping count and a block size of 4096 KB. As expected, the write rate rises with increasing OST count up to a maximum but drops off afterwards. With increasing core counts, the peak moves to higher OST counts and widens

This may suggest that for any number of nodes, using 8 OSTs per node would show the highest observed performance, which is tested in a multi-node study utilizing multiples of 8 as the stripe count. The result of this study is shown for a block size of 4096 KB in Fig. 10. Two things are evident: The performance peak indeed moves to higher OST counts with more cores, but the peak itself also widens. Up to 4 nodes (96 cores), the 8 OST per node model describes the performance peak reasonably well, but for 8 nodes, the peak is reached before the predicted value at 64 OSTs. As `ws9` only has 54 OSTs, the predicted position cannot



(a) 8 OSTs



(b) 4096KB block size

Fig. 11 Effect of the stripe size on the write performance for a single node, log-scaled x-axis. Generally, having a block size as big or bigger than the stripe size yields good performance

even be reached; however, for small node counts, the estimate describes the peak adequately. Determining the OST count showing the highest performance for each line of processor count and fitting a linear function for, it yielded the function $\#OST = 5.6N + 4.9$, with N representing the node count. This is valid for node counts up to 8, after which the maximum of OSTs (54) should be used. A previous study [35] on the predecessor of *ws9* with 168 OSTs showed that the peak is best described with a linear function $\#OST = 5N + 3$ with N being the number of nodes. This function also fits the current data from *ws9* but tends to underpredict the number of OSTs. By adjusting the striping the multi-node, write rate was improved from 2.5 GB/s to up to 9.5 GB/s and thus almost a fourfold improvement over the default striping.

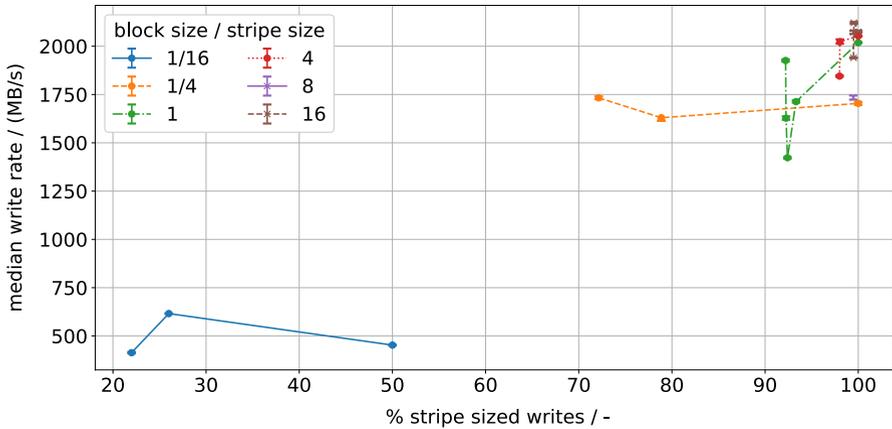


Fig. 12 Variation of the percentage of stripe-sized writes for 8 OSTs on a single node with different block- and stripe sizes. The best performance is generally reached at or close to 100% stripe-sized writes, which can be achieved by setting the stripe size to a small divisor of the block size

5.1.2 Stripe size

In order to analyze the effect of the stripe size, a study with varying block sizes, stripe sizes and striping counts is done for a fixed core count of 24 (single node). This is done in order to maximize the throughput at the OSTs, as the network link is assumed not to be a limiting factor. Figure 11a shows the results for a fixed striping count of 8 and Fig. 11b for a block size of 4096 KB. The plots introduce a new parameter $bsToss = \frac{blocksize}{stripesize}$, the ratio of block size to stripe size, as we are interested in stripe sizes showing good performance for a specific block size. For block sizes above 4096 KB, the write performance levels off at a ratio of 4, whereas for smaller block sizes the performance has a peak at a ratio of 1. These results are independent of the striping count being used in this case, as shown in Fig. 11b. This result can in fact be explained by considering the data gained via the environment variable `MPICH_MPIIO_STATS=1`. It yields, among other information, the number of system writes as well as how many of these writes were stripe-sized. Plotting the percentage of stripe-sized writes for the data from Fig. 11a yields Fig. 12. The highest performance is achieved at or close to 100% stripe-sized writes, which can be achieved by setting the stripe size to a small divisor of the block size. Thus, we suggest the following heuristic for the stripe size (SS) based on the block size (BS): $SS = kBS, k \in \{1/1, 1/2, 1/4\}$, rounded to the nearest multiple of 64 KiB. The following studies will be using a ratio of 1/4, i.e., the data of one process is distributed among 4 stripes. For a block size of 64 KB, a stripe size of 64 KiB is employed since that is the minimum available stripe size.

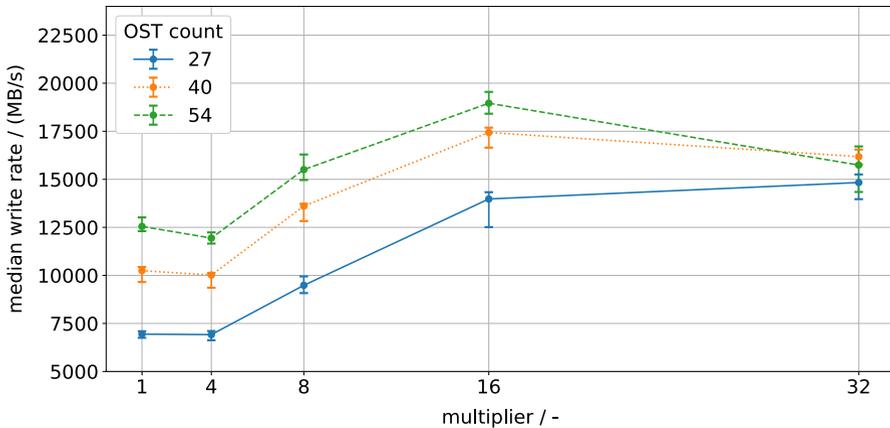


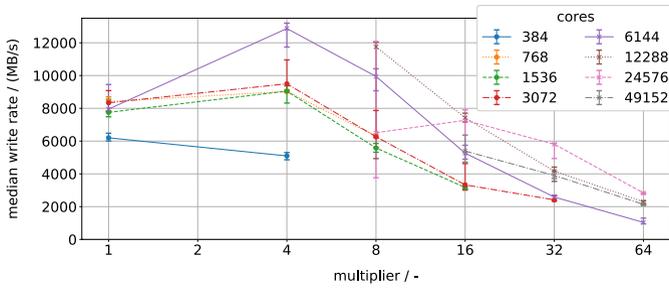
Fig. 13 Effect of multiplier on different stripe counts for a block size of 4096 KB at 3072 cores. The stripe counts below the maximum show no performance benefit compared to using the maximum number of OSTs

5.1.3 Multiplier

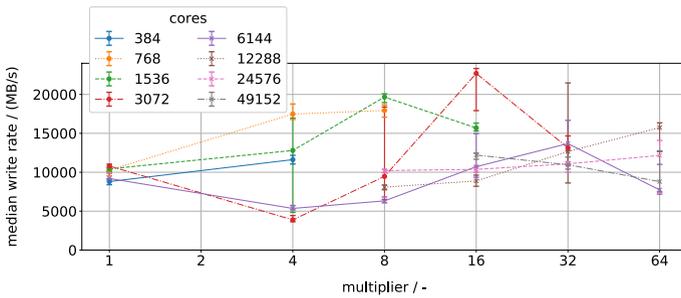
As we have already seen, the performance for 8 nodes levels off around the maximum number of OSTs available, most likely due to lock contention between aggregators. In order to get higher throughput from the OSTs, CRAY provides a novel locking mechanism called Lustre Lock-ahead (LLA) on *ws9*, which allows multiple aggregating I/O processes to write concurrently to the same OST [34].

This locking mechanism is activated via MPI I/O `hintscray_cb_write_lock_mode=2:cray_cb_nodes_multiplier=x` where x is the *multiplier*, i.e., the number of aggregators writing to the same OST. We shall first show that on *ws9*, increasing OSTs up to the maximum should be done before activating LLA by varying both OST count and the multiplier. The results are shown in Fig. 13 for a block size of 4096 KB and 3072 cores: For 27 OSTs to reach the same performance as 54 OSTs without any multiplier, a multiplier of 16 is required, whereas 54 OSTs with such a multiplier outperform the 27 OST case. Hence, the maximum performance is reached when using all available OSTs, and thus, in the following we only need to investigate the effect of different multipliers for the maximum number of OSTs.

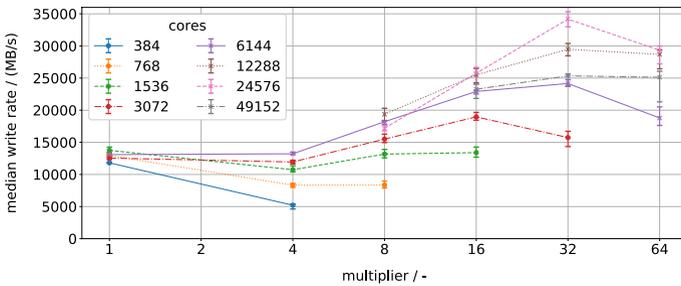
Figure 14 shows the results of varying the block size, the core count as well as the multiplier. Note that a multiplier of 1 refers to keeping LLA turned off, as running LLA with a multiplier of 1 shows strictly worse write rates than without LLA. The multiplier was increased until either more aggregators than cores would have been used or a multiplier of 64 was reached. The results for block sizes 64 KB and 512 KB should be taken with a grain of salt, as the writing time per frame for these was mostly below 0.1 s. For block sizes 64 KB and 512 KB and high core counts (> 6144), smaller multipliers were not investigated as these were expected to take excessive amounts of CPU time. In case of a 64 KB block size, using a multiplier of 4 when using more than 384 cores seems to



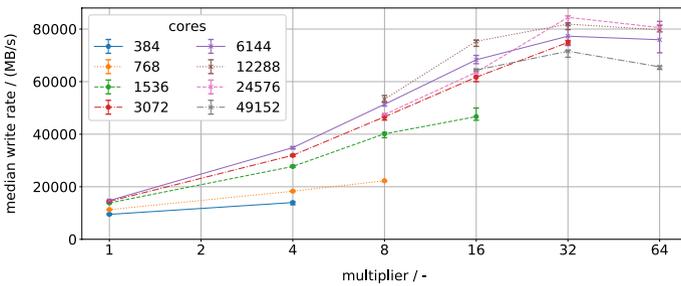
(a) 64KB block size



(b) 512KB block size



(c) 4096KB block size



(d) 32768KB block size

Fig. 14 Effect of the multiplier for 54 OSTs for different block sizes, log₂-scaled x-axes. While for block sizes > 4096 KB there seems to be a clear pattern for the highest observed performance, the lower block sizes show an erratic behavior

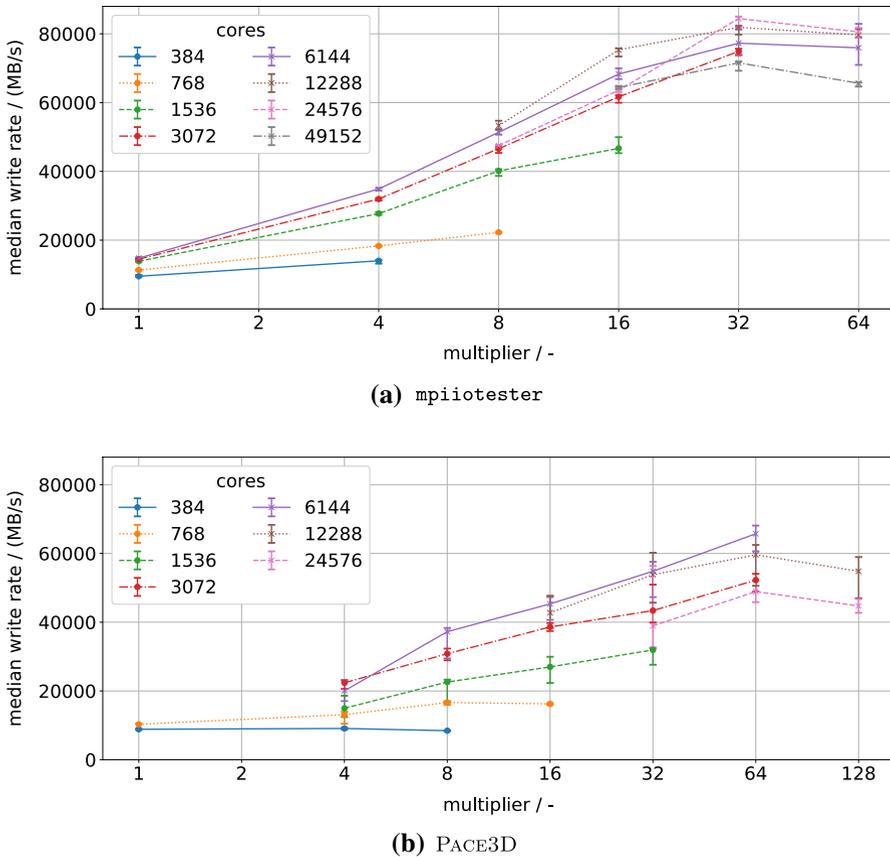


Fig. 15 Comparison of the writing rates achieved by PACE3D and mpiotester on a large number of nodes for a block size of 32768 KB. The difference is mainly attributable to the 3DDD as well as the I/O buffer filling being measured as well

give the peak write performance. Increasing the block size to 512 KB, the picture becomes much less clear, with the peak write performance being reached at a multiplier of 4, 8, 16, 32 and 64 for 768, 1536, 3072, 6144 and {12288, 24576} cores, respectively. The highest investigated core count 49152 reaches the highest performance at its lowest investigated multiplier of 16. A further increase in block size to 4096 KB yields a less muddled picture, with core counts above 6144 clearly exhibiting a significant peak at a multiplier of 32. Finally, a block size of 32768 KB shows this effect even more pronounced starting from 3072 cores. In total, large block sizes ≥ 4096 KB show pronounced performance increases (factor of 2-8) when employing a multiplier of 32 to 64. Below this block size, a performance increase of up to 2 is possible, but the optimal multiplier non-trivially depends on the employed core count.

5.2 PACE3D performance

We have performed similar weak scaling runs as in the previous section for the materials microstructure simulation framework PACE3D. There are two key differences to the `mpiootester` configuration: A 3D domain decomposition (3DDD) is employed and the time required for filling internal I/O buffers is measured with the internal timers. Thus, these results cannot be directly compared to `mpiootester` results; in advance, we can easily predict that the writing rate will be lower. In total, the results showed a similar response behavior to the investigated parameters, and the writing rates were indeed smaller than for `mpiootester`. Heuristics derived from the PACE3D I/O data for stripe count and stripe size yield very similar results as previously established for `mpiootester`. Consider Fig. 15 as an executive summary, which shows the performance of both PACE3D and `mpiootester` on a large number of nodes when the multiplier is varied. Both figures show a dependence of the multiplier effect on the core count with higher core counts being able to reach higher writing rates, given a sufficient multiplier. For the two highest process counts, a higher multiplier than in the previous study was also tested since no clear peak was visible. While this configuration showed a lower write rate, the confidence intervals overlap, suggesting that there is no significant difference between employing a multiplier of 32, 64 or 128 for these. Thus, a multiplier of 32 to 64 is likely to yield good performance for the more usual 3DDD, given a sufficiently big local domain, as well.

6 Conclusion and outlook

By performing a large range of write performance measurements with a specialized application, heuristics for good write performance for parallel I/O on the `ws9` filesystem of HLRS were determined. The striping configuration yielding the highest write rate while writing a spatially distributed array was found to be dependent on both the block size and the number of employed compute nodes. The stripe count or equivalently OST count should be set to $\#OST = 5.6N + 4.9$ with N being the number of employed compute nodes. The stripe size (SS) should be set to $SS = kBS$, $k \in \{1/1, 1/2, 1/4\}$ with BS representing the number of bytes to be written per processor, rounded to the nearest multiple of 64 KiB. These striping heuristics enabled up to fourfold improvement over the default striping configuration. Further performance was gained by employing Cray's LLA locking method, which introduces a new parameter called multiplier. Various choices of this multiplier were investigated. No general heuristic considering both block size and the number of employed compute nodes could be derived. However, for block sizes equal to or above 4096 KB and above 3072 cores (128 nodes), a multiplier of 32 was found to show the best performance, yielding another factor of 2–8 from the optimized striping configuration. Thus, a total factor of 10–32 of sustained throughput increase was gained for the single shared file model, reaching in these experiments up to 85 GB/s which

corresponds to 42% of the total I/O bandwidth of 202.5 GB/s. Finally, the striping heuristics were shown to be transferable from a similar Lustre setup [35] as well as to a general application code.

The transferability of heuristics between similar Lustre setups has been shown with this paper and previous work [35]. What needs to be done next is to test the transferability further for different Lustre setups and determine how to proceed from a Lustre setup to good I/O performance, obviating the need for experimental runs. Furthermore, the effect of burst buffers needs to be considered, as prior research [36, 37] has shown these to provide great performance benefits. With the coming of the next generation file system at HLRS, both points will be investigated.

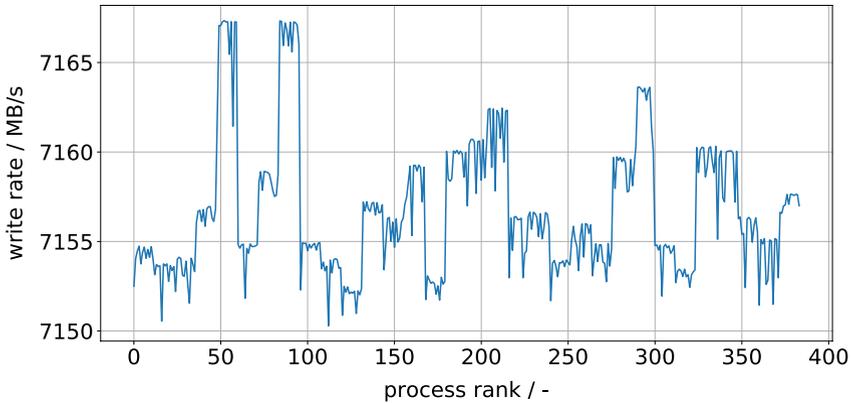
Appendix

We need to establish some basic information about the distribution of the measurements in order to quantitatively analyze them:

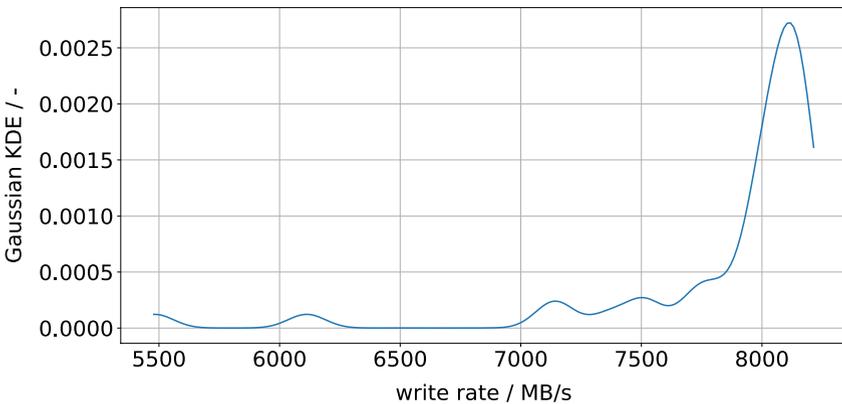
1. How are the writing times distributed?
2. Does the process rank influence the writing time?
3. Is there an effect of writing style on write performance?
4. Does the frame number influence the writing time?
5. Does the domain decomposition affect the writing rate?

As a first example, consider a number of runs N of constant block size, striping parameters, frames and number of MPI processes. For each MPI process and frame, there are N elapsed times during writing, which can be converted to write rates via $WR = \frac{\text{data written}}{\text{time elapsed}}$. The median of these writes rates for a specific frame is shown in Fig. 16a. Each point is based on $N = 42$ runs conducted at different times of days and weekdays. In order to show independence of process rank and writing rate, we will use random subsamples of this data and test for statistical significant differences; if the random subsamples are not significantly different, then the subgroups can be exchanged arbitrarily, and hence, there is no significant influence of the process rank.

For standard ANOVA to be applicable, the distribution of the writing rate has to be normal; Fig. 16b shows the Gaussian kernel density estimation (KDE) of the raw data for Fig. 16a which indicates that the data is not normally distributed. Furthermore, the Shapiro–Wilk test was applied for individual runs which showed that the distribution of write rates among ranks is not normal. Since the conditions for ANOVA are not fulfilled, we shall use the nonparametric Kruskal–Wallis test to test for significantly different medians for the data shown in Fig. 16a. The test shows no significant differences at the $p = 0.01$ level. With this we have answered **i** and **ii**; the data is not normally distributed and the process rank does not significantly influence the write rate. The latter also follows from the collective operation to determine the exit code at the end of a `MPI_File_write_all` call.



(a) Median write rate for several runs over process rank. While there are significant differences, their magnitude relative to the overall write rate is small.



(b) KDE of the raw data from fig. 16(a). The probability density is not normally distributed.

Fig. 16 Median write rates as well as the frequency distribution of write rates for an exemplary setup repeated 42 times

In the following, the rank-wise data is averaged by taking the median of the write time over the ranks, which is then used to compute the write rate via $WR = \frac{\text{data written}}{\text{med}(\text{time elapsed})}$. Figure 17 depicts a violin plot of this write rate as well as the corresponding write time for three exemplary configurations. Each point corresponds to a specific frame and run, with 800, 840 and 660 total points, respectively for 24, 384 and 768 cores.

This shows that while most of the data is closely clustered in and around the IQR, there is a significant number of outliers, with the distribution becoming broader with more processes. We suspect these effects are caused by contention over shared resources (network throughput & disk access) with the chance of contention rising with higher process counts.

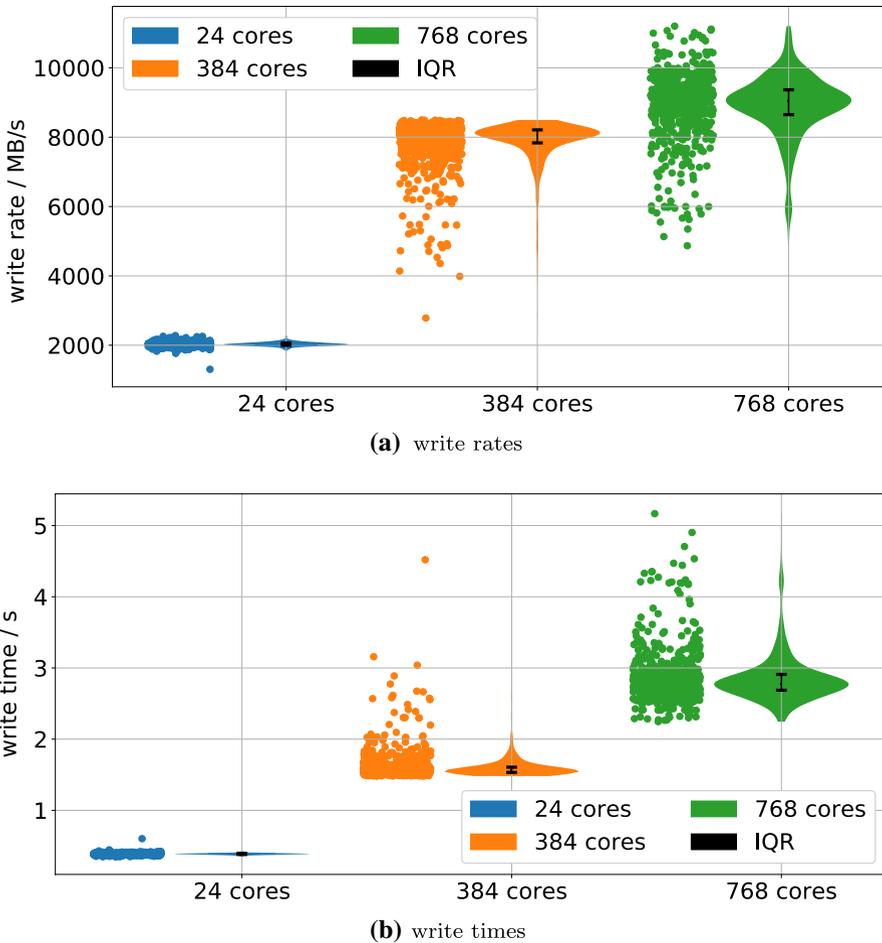
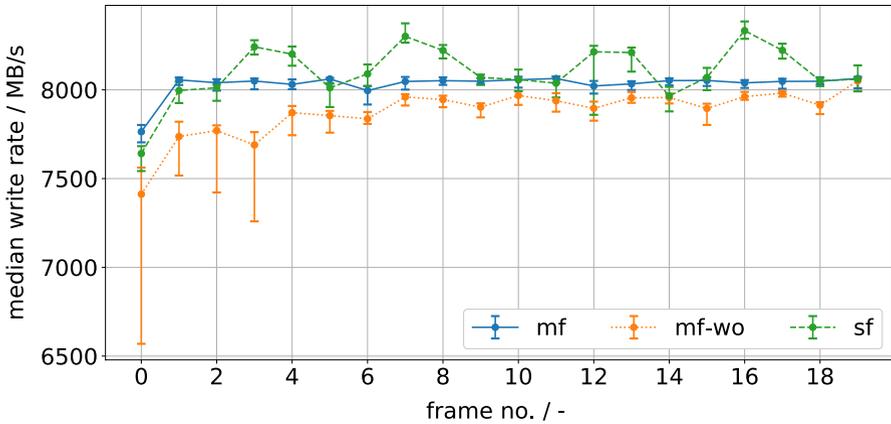


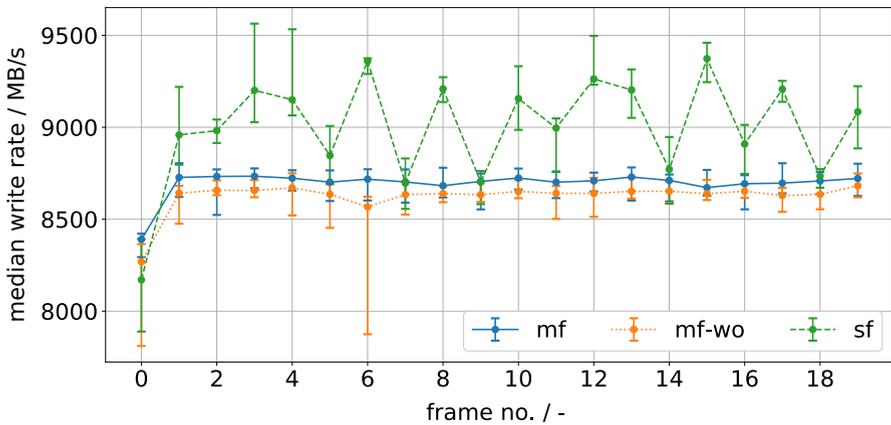
Fig. 17 Exemplary violin plots of three configurations: 24 cores with the default striping and 384 as well as 768 cores with the striping count set to 54, block size is 32,768 KB. Higher core counts lead to a more spread-out distribution of write rates

The shown data so far have all been for the “single file” (sf) writing style, i.e., writing all checkpoints into a single file and keeping it opened. Similar behavior is observed for the other two kinds of styles investigated, both inspired by the VTK file format: One file per written checkpoint (frame), excluding opening time (mf) as well as including the opening time (mf-wo). The write rates of the styles are now investigated in order to address **iii**. The plots in Fig. 18 indicate that there is a significant difference between the styles. Testing with Kruskal–Wallis shows that all differences are significant at the $p = 0.01$ level. Since the single-file style shows the best write performance, we chose to use this style for the performance tests.

Based on the confidence intervals of the figures in Fig. 18, it can be reasoned that past the first frame the performance is in a more or less steady state except for



(a) 384 cores, 54 OSTs

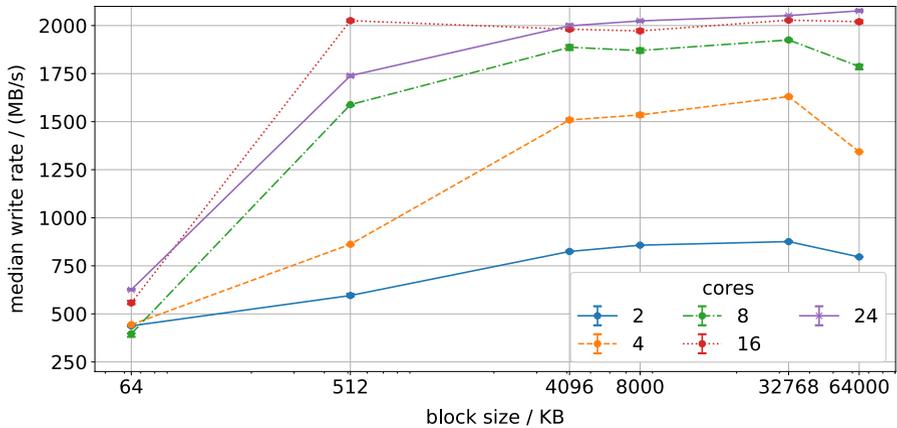


(b) 768 cores, 54 OSTs

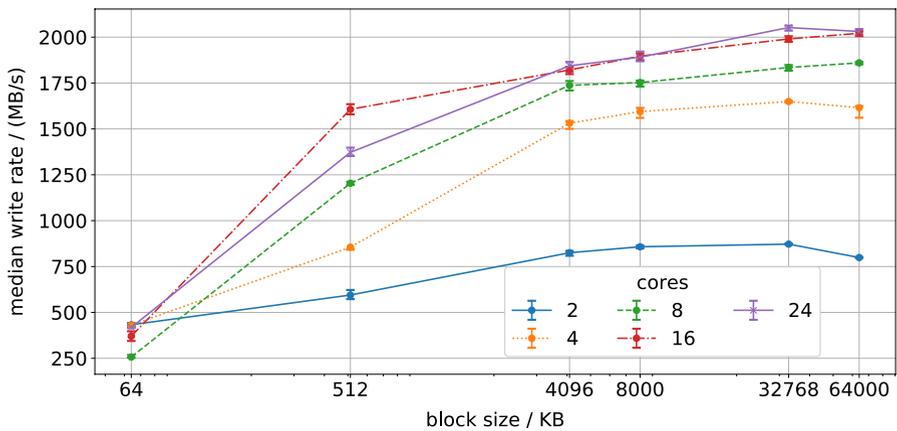
Fig. 18 Variation of write performance due to frame count and writing style. The single-file writing style generally shows the highest write rate but tends to scatter more. While there is a significant difference for the multiple file writing style with and without including the opening time, the difference is small relative to the total write rate

system jitter. Performing significance tests on pairs of frames shows that indeed the first frame is significantly different from other frames. Past the first frame, the number of significant differences per frame is distributed with no apparent pattern. We shall only exclude the first frame from further analysis as its deviation is by far the largest.

Finally, we address **iv** by comparing single-node and multi-node performance of a 1D domain decomposition (1DDD) with a 3D domain decomposition (3DDD). The 1DDD is split along the outermost coordinate and the 3DDD such that the number of cores per dimension increases or stays the same as the



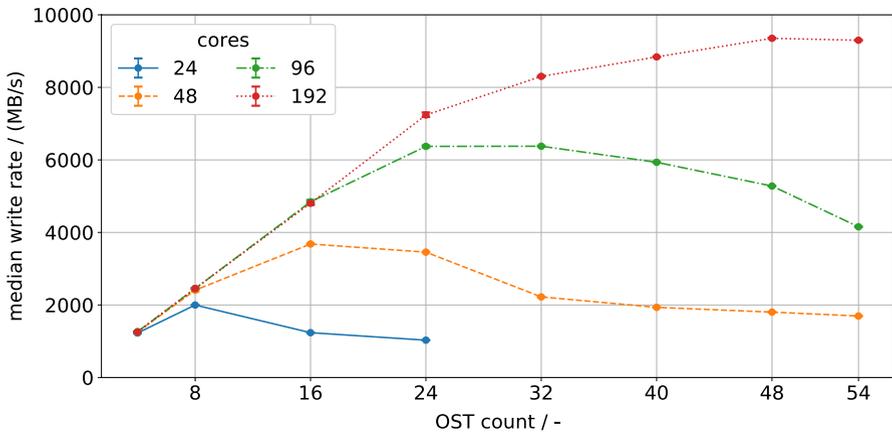
(a) 1DDD



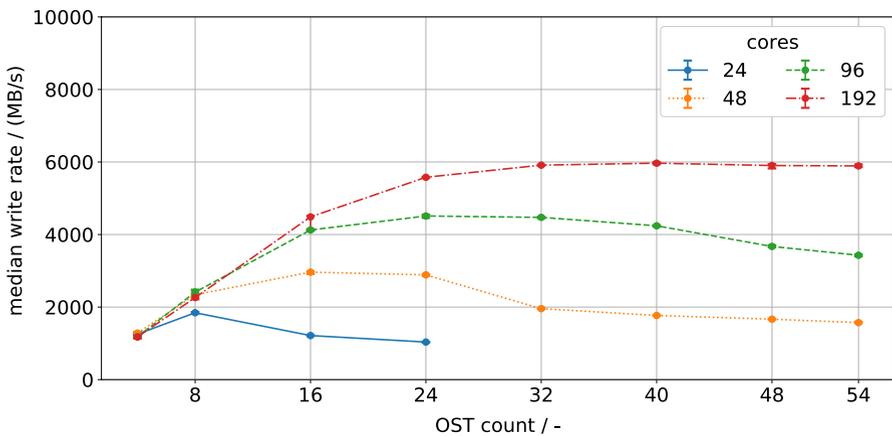
(b) 3DDD

Fig. 19 Influence of spatial decomposition on the writing rate for a single node. Highly similar results are achieved for both the 1DDD and 3DDD cases, with the 3DDD generally showing smaller write rates

number of cores is increased. Figure 19 shows the single-node case for both configurations, and there is a small but significant difference between the 1DDD and 3DDD cases. For multiple nodes as depicted in Fig. 20 the write rates and curve shapes differ substantially and the maximum write rate is smeared out for higher core counts, especially for 192 cores. However, the region of good performance in terms of striping does not differ significantly between the 1DDD and 3DDD cases. Hence, it is sufficient to investigate the 1DDD case and apply the derived heuristics to the more computationally efficient 3DDD.



(a) 1DDD



(b) 3DDD

Fig. 20 Influence of spatial decomposition on the writing rate for multiple nodes for a block size of 4096 KB. While there are now bigger differences than in the single-node case, the OST count at which the highest write rate is observed does not differ significantly

Acknowledgements Funding provided by the Deutsche Forschungsgemeinschaft under Grants NE 822/9-2 and NE 822/31-2 (Gottfried-Wilhelm-Leibniz-Preis) is gratefully acknowledged. Funding provided by the Bundesministerium für Bildung und Forschung under Grant 01IH16013 is gratefully acknowledged. This work was partially executed within the frame of the German SiVeGCS project. We are grateful to the computational resources on the Hazel Hen provided by the HLRS and for their support.

Funding Open Access funding enabled and organized by Projekt DEAL.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative

Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit <http://creativecommons.org/licenses/by/4.0/>.

References

1. Strohmaier E, Dongarra J, Simon H, Meuer M. Top500 list-november 2020. <https://www.top500.org/lists/top500/2020/11/>. Accessed 11 Feb, 2021
2. Message Passing Interface Forum (1997) Mpi-2: Extensions to the message-passing interface. <https://www.mpi-forum.org/docs/>. Accessed 13 Mar 2019
3. Liao W-k, Thakur R (2015) High performance parallel I/O, chapter MPI-IO. Chapman & Hall/ CRC computational science series : A Chapman & Hall book. CRC Press, pp 155–167
4. Latham R, Ross R (2013) Earth system modelling—volume 4: IO and postprocessing, chapter Parallel I/O Basics. Springer, Berlin, pp 3–12
5. McLay R, James D, Liu S, Cazes J, Barth W (2014) A user-friendly approach for tuning parallel file operations. In: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis. pp 229–236
6. Bartz C, Chasapis K, Kuhn M, Nerge P, Ludwig T (2015) A best practice analysis of hdf5 and netcdf-4 using lustre. In: Kunkel JM, Ludwig T (eds) High performance computing. Springer, Cham, pp 274–281
7. Behzad B, Byna SP, Snir M (2019) Optimizing i/o performance of hpc applications with autotuning. ACM Trans Parallel Comput. <https://doi.org/10.1145/3309205>
8. Thakur R, Gropp W, Lusk E (2002) Optimizing noncontiguous accesses in mpi-io. Parallel Comput 28(1):83–105
9. Vef M, Moti N, Süß T, Tocci T, Nou R, Miranda A, Cortes T, Brinkmann A (2018) Gekkofs—a temporary distributed file system for hpc applications. In: 2018 IEEE International Conference on Cluster Computing (CLUSTER). pp 319–324
10. Liang Z, Lombardi J, Charawi M, Hennecke M (2020) Daos: a scale-out high performance storage stack for storage class memory. In: Panda DK (ed) Supercomputing frontiers. Springer, Cham, pp 40–54
11. Hötzer J, Reiter A, Hierl H, Steinmetz P, Selzer M, Nestler B (2018) The parallel multi-physics phase-field framework pace3d 26:1–12
12. del Rosario JM, Bordawekar R, Choudhary A (1993) Improved parallel i/o via a two-phase run-time access strategy. SIGARCH Comput Archit News 21(5):31–38
13. Manual Lustre. Lustre software release 2.x - operations manual. <http://lustre.org/documentation/>. Accessed 11 Feb 2021
14. El-Harake HN, McMurtrie C (2015) Evaluation of the cray sonexion 2000 storage system. https://www.cscs.ch/fileadmin/user_upload/contents_publications/technical_reports/Evaluation_Cray_Sonexion2000_StorageSystem.pdf. Accessed 11 Feb 2021
15. Nestler B, Garcke H, Stinner B (2005) Multicomponent alloy solidification: phase-field modeling and simulations. Phys Rev E 71:041609
16. Ankit K, Nestler B, Selzer M, Reichardt M (2013) Phase-field study of grain boundary tracking behavior in crack-seal microstructures. Contrib Mineral Petrol 166(6):1709–1723
17. Ankit K, Urai JL, Nestler B (2015) Microstructural evolution in bitaxial crack-seal veins: a phase-field study. J Geophys Res Solid Earth 120(5):3096–3118
18. Ankit K, Selzer M, Hilgers C, Nestler B (2015) Phase-field modeling of fracture cementation processes in 3-d. J Petrol Sci Res 4(2):79–96
19. Vondrous A (2014) Grain growth behavior and efficient large scale simulations of recrystallization with the phase-field method, vol 44. KIT Scientific Publishing, New York
20. Selzer M (2014) Mechanische und strömungsmechanische topologieoptimierung mit der phasenfeldmethode

21. Hötzer J, Seiz M, Kellner M, Rheinheimer W, Nestler B (2019) Phase-field simulation of solid state sintering. *Acta Mater* 164:184–195
22. Hierl H, Hötzer J, Seiz M, Reiter A, Nestler B (2019) Extreme scale phase-field simulation of sintering processes. In: 2019 IEEE/ACM 10th workshop on latest advances in scalable algorithms for large-scale systems (ScalA). pp 25–32
23. Choudhury A, Geeta M, Nestler B (2013) Influence of solid-solid interface anisotropy on three-phase eutectic growth during directional solidification. *Europhys Lett* 101(2):26001
24. Choudhury A (2015) Pattern-formation during self-organization in three-phase eutectic solidification. *Trans Indian Inst Met* 68:1137–1143. <https://doi.org/10.1007/s12666-015-0659-9>
25. Ettrich J (2014) Fluid flow and heat transfer in cellular solids, vol 39. KIT Scientific Publishing, New York
26. Schneider D, Selzer M, Bette J, Rementeria I, Vondrous A, Hoffmann MJ, Nestler B (2014) Phase-field modeling of diffusion coupled crack propagation processes. *Adv Eng Mater* 16(2):142–146
27. Schneider D, Tschukin O, Choudhury A et al (2015) Phase-field elasticity model based on mechanical jump conditions. *Comput Mech* 55:887–901. <https://doi.org/10.1007/s00466-015-1141-6>
28. Schneider D, Schmid S, Selzer M, Böhlke T, Nestler B (2015) Small strain elasto-plastic multiphase-field model. *Comput Mech* 55(1):27–35
29. Mennerich C (2013) Phase-field modeling of multi-domain evolution in ferromagnetic shape memory alloys and of polycrystalline thin film growth, vol 19. KIT Scientific Publishing, New York
30. Mukherjee A, Ankit K, Mukherjee R, Nestler B (2016) Phase-field modeling of grain-boundary grooving under electromigration. *J Electron Mater* 45(12):6233–6246
31. Ben Said M, Selzer M, Nestler B, Braun D, Greiner C, Garcke H (2014) A phase-field approach for wetting phenomena of multiphase droplets on solid surfaces. *Langmuir* 30(14):4033–4039
32. Weyer F, Said MB, Hötzer J, Berghoff M, Dreesen L, Nestler B, Vandewalle N (2015) Compound droplets on fibers. *Langmuir* 31(28):7799–7805 (PMID: 26090699)
33. Hötzer J, Tschukin O, Ben SM, Berghoff M, Jainta M, Barthelemy G, Smorchkov N, Schneider D, Selzer M, Nestler B (2016) Calibration of a multi-phase field model with quantitative angle measurement. *J Mater Sci* 51(4):1788–1797
34. Moore M, Farrell P, Cernohous B (2018) Lustre lockahead: early experience and performance using optimized locking. *Concurr Comput Pract Exp* 30(1):e4332
35. Seiz M, Hötzer J, Hierl H, Reiter A, Schratz K, Nestler B (2021) High Performance Computing in Science and Engineering '19: Transactions of the High Performance Computing Center, Stuttgart (HLRS) 2019, chapter Accelerating phase-field simulations for HPC-systems. Springer, Berlin
36. Bard D (2017) Accelerate your i/o with the burst buffer. http://press3.mcs.anl.gov/atpesc/files/2017/08/ATPESC_2017_Track-3_04_8-4_1030am_Bard-Burst_Buffer.pdf. Accessed 11 Feb 2021
37. Schenck W, El Sayed S, Foszczynski M, Homberg W, Pleiter D (2016) Early evaluation of the infinite memory engine burst buffer solution. In: Taufer M, Mohr B, Kunkel JM (eds) High performance computing. Springer, Cham, pp 604–615

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.