Deduplication Potential of HPC Applications' Checkpoints

Jürgen Kaiser, Ramy Gad, Tim Süß, Federico Padua, Lars Nagel, André Brinkmann
Zentrum für Datenverarbeitung
Johannes Gutenberg University Mainz, Germany
Email:{j.kaiser, gad, t.suess, padua, nagell, brinkman}@uni-mainz.de

***Abstract*—HPC systems contain an increasing number of components, decreasing the mean time between failures. Checkpoint mechanisms help to overcome such failures for long-running applications. A viable solution to remove the resulting pressure from the I/O backends is to deduplicate the checkpoints. However, there is little knowledge about the potential to save I/Os for HPC applications by using deduplication within the checkpointing process. In this paper, we perform a broad study about the deduplication behavior of HPC application checkpointing and its impact on system design.**

I. Introduction

The era of exascale computing is approaching [1]. It is driven by the continuously growing demands of researchers and their applications for more computing power. This power is typically delivered by increasing the number of cores and components.

The desire for more computing power introduces new issues. The increasing number of nodes in clusters decreases the *mean time between failures* (MTBF). While the MTBF was in the order of days [2], it will decrease to the order of minutes for exascale systems [3]–[5]. Therefore, HPC programmers must consider that their (long running) jobs will not finish without any hardware or software failure. As a consequence, several programs integrate checkpoint mechanisms [6], where the application writes its state to the storage backend and resumes its computation from this state after a node failure. This checkpointing approach is called *application level checkpointing*. However, for legacy codes an integration of these mechanisms can be difficult and often exceeds the benefits. Even for non-legacy programs, the integration of checkpointing mechanisms is time consuming. Alternatively, *system level checkpointing* can be applied, where an external tool checkpoints the running application without any assistance from the researcher.

Checkpointing leads to a new issue: most approaches do not scale well with the number of nodes. Creating checkpoints for many processes puts high pressure on the storage backend and on the network. Furthermore, the energy costs of moving data will exceed the costs of local computation in exascale environments [7]. A possible solution to increase checkpointing scalability is to apply data deduplication. Data deduplication systems have been introduced to reduce the amount of stored data on disk-based backup systems and have made these economically applicable. Most of these systems divide the data into chunks, compute chunk fingerprints using a cryptographic hash function like SHA-1, and identify redundant chunks by comparing the chunks' fingerprints [8]–[11]. Besides backup, primary storage also provides deduplication potential. Meister et al. showed a *huge potential for data deduplication in HPC storage systems that is not facilitated by today's HPC file systems* [12]. Nicolae and Kulkarni et al. applied the deduplication approach to the checkpointing use case and discussed systems to reduce the I/O load during checkpointing [13], [14].

However, there is little knowledge about the general deduplication potential of HPC applications' checkpoints. The study of a broad spectrum of applications presented in this paper give new insights into the applicability of checkpoint deduplication in different application areas. The coupling of checkpointing and deduplication for applications, which have a high deduplication rate, can definitely help to improve the checkpointing scalability. Nevertheless, if an application does not have enough redundancy, the deduplication process can decrease the overall checkpointing performance. In addition, an improperly configured deduplication process can waste deduplication potential. Our experiments show, for example, that choosing the wrong chunking process alone can alter the volume of the data after deduplication by 10%.

In this study, we investigate the deduplication behavior of a wide range of HPC applications. For each application we show its deduplication potential for different deduplication configurations. Since not all applications provide built-in checkpointing, we use DMTCP for system level checkpointing. Using this type of checkpointing, we show that there is a high potential for saving data sent to disk and for increasing checkpointing performance. Furthermore, we show where redundancies occur and how they can be exploited best.

II. Related work

Reducing an application's checkpoint size is a potential source of improving the performance and the scalability of a checkpointing process. In addition to storage reduction, a smaller checkpoint means smaller dumping time, smaller bandwidth consumption, and less I/Os.

The simplest checkpoint approach is a memory core dump of the computation status, which is called system level checkpointing. It can be implemented in the kernel space in case of BLCR (Berkeley Lab Checkpoint/Restart) [15] or in the user

space in case of DMTCP (Distributed MultiThreaded CheckPointing) [16]. The advantage of system-level checkpointing is that it is transparent to the application and the checkpoint can be taken at arbitrary points. However, those tools produce relatively large checkpoints, as they include data that is not required for restarting the computation.

Application level checkpointing was introduced to provide smaller-sized checkpoints by adding more complexity. The programmer exploits the knowledge of the structure and the behavior of a given application to find the useful data structures that should be included in the checkpoint and encodes this knowledge in the application's source code.

Library and compiler support have been proposed to simplify the complexity of application level checkpointing. The `Libckpt` library provides transparent checkpoint/restart, but it requires user directives to mark the checkpoints' locations and data [17]. Bronevetsky et al. provide a source to source compiler tool that can automatically instruments the code to save and restore its own status. The tool coordinates checkpoints and restarts for parallel OpenMP [18], [19] and MPI programs [20]–[22].

Compression techniques [23] and incremental checkpointing [24] were introduced to decrease the checkpoint size. They are applied to the raw checkpoint data. Incremental checkpointing only save the differences between checkpoints instead of saving the complete checkpoints.

Differences between checkpoints also can be created by tracking writes to memory pages. Only these pages are included in the checkpoint [25] [26]. The disadvantage of this approach is that it requires kernel level support.

Fingerprinting-based deduplication is also used to find differences between checkpoints. The data is partitioned into non-overlapping data blocks (chunks). A cryptographic hash function is used to obtain a unique identifier (hash value) for each chunk. The chunks' hash values are compared against each other so that repeated chunks are only written once.

Data chunks can have a fixed size (static chunking, SC) or variable size (content-defined chunking, CDC). SC is simple and fast, but it fails to detect duplicates in near identical, but shifted data streams. Cores et al. combine SC hashing-based deduplication with incremental application level checkpointing [27]. CDC finds the boundaries of chunks based on their content and therefore overcomes the data shifting problem [28]. The chunks generated by CDC have variable chunk sizes within an upper and lower limit [28].

Kulkarni et al. [14] describe a deduplication-based file system to checkpoint HPC applications. The system is evaluated using checkpoint dumps of five proxy implementations of real HPC applications and three dumps of full applications. While the experiments show a great variation in the deduplication ratio, there is no analysis of how precisely the proxies reflect the deduplication potential of the respective full applications.

Moody et al. introduced multi-level checkpointing to improve scalability [29]. Traditional checkpoint systems use the parallel file system (PFS) to store the checkpoint data. The PFS is a potential source of bottlenecks at scale. Moody et al. use multi-level storage at different speeds to save the checkpoint data, a fast local storage on top and PFS in the bottom. Their work reduces the load on the PFS by a factor of two and allows a better scalability.

III. Deduplication of Checkpoints

A system designer faces several challenges and design decisions when he creates a deduplication system for HPC checkpoints. The overall goal is to perform deduplication fast while detecting as much redundancy as possible and keeping the resource requirements low. One major choice is the chunking method and the chunk size. Content-defined chunking can allow better redundancy detection than fixed-size chunking, but induces a higher computation overhead.

The chunk size is a vital parameter because it influences the quality of redundancy detection and the number of chunks and thus the processing time: the smaller the chunk size, the more fine-grained the detection and the longer the processing. We will investigate these correlations in Section V-A.

Another disadvantage of small chunks is that the size of critical data structures grows with the number of chunks. For example, each deduplication system holds an index mapping chunks to the storage location of their raw data. The size of an index entry typically ranges from 24 B to 32 B, including hash value, storage location, and counters and pointers for the index implementation; so, each stored terabyte of unique checkpoint data requires 4 GB of extra memory if we assume 20 B SHA1 hashes and 8 KB chunks, which allows it to hold the full index in memory. Consequently, no disk I/Os are required in the deduplication process except for writing new chunks to disk.

Since the index grows with every checkpoint, it is advisable to delete old checkpoints. Due to garbage collection, this implicates additional overhead which depends on the change rate of the process images. A low change rate implies high redundancy and less new data. We will consider the change rate of the applications in Section V-A.

Next, the system designer must consider scaling properties. The probably best scaling approach is to let each compute node perform its own deduplication and store raw chunk data on local storage. However, all checkpoints for that node would be lost in case of a hardware failure. On the other side, a single deduplication instance can easily become a performance bottleneck in the presence of thousands of processes and nodes performing checkpoints. Therefore, it is advisable to replicate chunk data to other nodes, which reduces the savings achieved by the deduplication process. Our results indicate that the redundancy detection improves with each additional process. Therefore, designers should consider a grouped approach where a group of nodes perform joint deduplication and replication. We will determine the deduplication potential for different group sizes in Section V-D.

Finally, the system can be further optimized by exploiting so-called *chunk biases*, i. e., the fact that some chunks occur more often than others. The most important one is the *zero chunk*, i. e., the chunk that only consists of zeroes. We will investigate different chunk biases in Section V-A and V-E.

IV. Methodology

a) Applications: We evaluated applications from different scientific areas like physics, chemistry, material science, meteorology, and biology to understand the deduplication behavior across a broad range of applications and user bases. In the following, we briefly describe each application.

mpiblast [30] is a parallel implementation of NCBI BLAST and computes alignments of DNA sequences. It scales to hundreds of processors and improves NCBI BLAST's performance by several orders of magnitude. mpiblast achieves this by using database fragmentation, query segmentation, intelligent scheduling, and parallel I/O.

pBWA [31] is an MPI implementation of BWA [32], a popular software package for mapping low-divergent sequences against a large reference genome. The data distribution among the available processors consists mainly of two parts: index distribution and sequences distribution. First an index file is generated as BWA. This file is read by a master process and then broadcast to all available processors. After that pBWA distributes the sequences and computes the alignments on each process.

bowtie computes alignments of DNA sequences similar to pBWA and mpiblast but is specialized in short read sequences. It *conducts a quality-aware, greedy, randomized, depth-first search through the space of possible alignments* [33]. bowtie itself is a serial program, which is often used in parallel fashion. An MPI-based tool called pMap [34] is used to parallelize the execution of bowtie. The parallelization performed by pMap is conceptually similar to the one of pBWA. The index file, which is associated with the reference genome, is replicated on every processor and the reads contained in the input file are distributed among the processors.

phylobayes [35] is a Bayesian Monte Carlo Markov Chain sampler for phylogenetic reconstruction using protein alignments.

ray [36] is a parallel *de novo genome assembler* for next-generation sequencing data. The sequences are distributed among the available MPI ranks and each rank gets an equal number of sequences assigned to it.

Espresso++ [37] is a software framework targeted at the simulation of soft matter systems. We ran a simulation that uses an adaptive resolution scheme (AdResS) in which regions of the simulation box have different levels of chemical details. Espresso++ uses domain decomposition to distribute the data among the available processors.

gromacs [38] is a widely used software to perform molecular dynamics (MD) simulations of molecules like proteins and lipids in which complex bonded interactions are involved. We ran a program that calculates the absolute solvation free energy of ethanol.

LAMMPS [39] is also a molecular dynamics program. By default it uses spatial decomposition with equal-size domains. It also implements a series of commands that can augment the distribution of the domains among processors based on different criteria. In addition to exploiting inter-node parallelism via the aforementioned decomposition, LAMMPS also exploits intra-node parallelism where possible by using a variety of packages (GPU, USER-CUDA, USER-OMP, USER-INTEL, KOKKOS). We ran the ReaxFF benchmark, which is a simulation of PETN crystal and is shipped with the software package.

NAMD [40] is a parallel and highly scalable code designed for the simulation of large biomolecular systems. It is written using the Charm++ [41] programming framework. It uses a combination of spatial and force decomposition to benefit from the available processors.

nwchem [42] is a software used for computational chemistry simulations. nwchem uses domain decomposition to distribute the data among the available processors.

CP2K [43] is an open source framework written in Fortran that enables users to perform molecular simulations based on the *density functional theory* (DFT). The simulation produces positions, velocities and forces at each time step for every atom.

Quantum ESPRESSO (QE) [44] is a set of codes written in Fortran used to calculate electronic structures and to model materials. We use a code called CP which performs a variable-cell Car-Parrinello molecular dynamics simulation. QE implements parallelization at different levels to use the available processors for the simulation.

eulag [45] (**Eu**lerian/semi-**Lag**rangian fluid solver) is a numerical solver for geophysical flows written in Fortran. In our case we ran a Large-Eddy simulation using the Smagorinsky subgrid-scale model and used PnetCDF [46] as a parallel I/O library. Domain grid decomposition is used to map the model grid to the available processors.

echam [47] is a climate model which simulates atmospheric general circulation. We used ECHAM5, which is written in Fortran, and simulated weather conditions starting from January 1998. The data is distributed via domain grid decomposition to the processors.

openfoam [48] is a toolbox of numerical solvers for computational fluid dynamics problems. Our test followed a standard user workflow including preprocessing steps before calling the final numerical solver. Preprocessing consists of domain decomposition (*decomposePar*), allocating the sub-domains to processors and checking the validity of the mesh produced. The solver was the *icoFoam* solver, a transient solver for incompressible, laminar flow of Newtonian fluids.

b) Checkpoint generation: In this study, we use the DMTCP tool [16] to generate the applications' checkpoints. We chose DMTCP because it does not require root privileges and is independent from the kernel version. DMTCP provides checkpointing at system level. These checkpoints can be compressed during creation. We disabled this feature for our analysis since a compression before the redundancy detection of the deduplication destroys the latter. Deduplication systems typically use compression after the chunk identification when they write the raw chunk data to disk. Unlike application level checkpointing, system level checkpointing is transparent to the applications and can be done at arbitrary points. Tools for system level checkpointing also allow the checkpointing of

TABLE I
CHECKPOINT STATISTICS FOR ALL APPLICATIONS, EACH RUNNING ON 64 PROCESSES.

App	avg	sum	min	25%	75%	max
pBWA	132GB	1.4TB	35GB	52GB	184GB	185GB
mpiblast	33GB	405GB	33GB	33GB	33GB	33GB
ray	75GB	902GB	37GB	70GB	89GB	93GB
bowtie	94GB	470GB	1.2GB	65GB	134GB	175GB
gromacs	34GB	418GB	34GB	34GB	34GB	34GB
NAMD	10GB	120GB	10GB	10GB	10GB	10GB
Espresso++	17GB	213GB	13GB	18GB	18GB	18GB
nwchem	42GB	511GB	29GB	43GB	43GB	43GB
LAMMPS	52GB	631GB	52GB	52GB	52GB	52GB
eulag	35GB	428GB	35GB	35GB	35GB	35GB
openfoam	17GB	213GB	3.2GB	19GB	19GB	19GB
phylobayes	39GB	473GB	39GB	39GB	39GB	39GB
CP2K	43GB	518GB	37GB	43GB	43GB	43GB
QE	99GB	1.2TB	74GB	88GB	109GB	109GB
echam	18GB	227GB	18GB	18GB	18GB	18GB

legacy codes, which cannot be modified. On the other hand, system level checkpoints generate relatively large checkpoints in comparison to application level checkpoints. As system level checkpointing includes more redundant data, it is expected that these process images also have a higher deduplication potential than application level checkpoints. However, system level checkpointing offers more freedom in choosing the checkpoints' locations transparently to the application.

DMTCP generates one checkpoint image for each MPI process. The image is composed of a global header section, a header for each contiguous memory area (contains address range, permissions, etc.), and the data section (memory pages) for the different contiguous memory areas. The header section consists of 4 KB or one memory page. The first memory address of a continuous memory block is always a multiple of 4,096. Therefore, all checkpoint images are page-aligned.

We generate checkpoints every ten minutes. Almost all applications perform their computations for two hours. Only bowtie (after 50 minutes) and pBWA (after 110 minutes) finished earlier. The computations are distributed among 64 processes for all applications. The checkpointing period, the total applications' execution time, and the number of processes are chosen based on the possibility of having comparable characteristics between a large set of applications. However, we vary the number of used processes in Section V-C. Table I shows the different sizes of the checkpoints.

c) Deduplication: We analyzed each checkpoint with the FS-C deduplication tool suite [49], which has already been applied in several deduplication studies [50], [51]. We chose fixed-sized chunking and content-defined chunking (CDC) as chunking methods. For CDC, the suite uses Rabin's fingerprinting [52] to determine chunk boundaries and computes the SHA1 chunk fingerprints. For each checkpoint, we generated traces with an (average) chunk size of 4, 8, 16, and 32 KB.

In other deduplication domains, the disadvantage of fixed-sized chunking is its low tolerance against data shifts. A single inserted byte shifts the content of each following chunk and, therefore, their fingerprints. As a result, the system would identify these chunks as new.

However, fixed-sized chunking can be used for memory deduplication as there are no global shifts. Instead, data shifts usually affect few memory pages. Memory deduplication typically uses fixed-size chunking with a chunk size equal to the physical memory page size [53], [54]. We generate the same page alignment for fixed sized chunking and 4 KB chunks since the embedded process images are page-aligned.

V. EVALUATION

The deduplication potential of data contains different aspects. We first show general deduplication properties of the applications, i.e., the general deduplication ratio, the ratio of new chunks per checkpoint, and the main sources of redundancy. Next, we investigate the deduplication behavior for different scaling (Section V-C) of the applications and distributions of the applications' processes among compute nodes (Section V-D). Finally, we discuss different chunk biases (Section V-E).

All experiments were run on nodes of the HPC cluster *Mogon* of the Johannes Gutenberg University Mainz. Each of the nodes is equipped with four AMD Opteron 6272 processors and at least 128 GB RAM. All nodes have access to a local scratch and a global GPFS file system.

A. General Deduplication

For each application, we ran a computation that was distributed among 64 cores. For each computation, we created checkpoints every 10 minutes for a total computation time of two hours. Next, we deduplicated all checkpoints and computed the overall deduplication ratio, which is defined as $1 - \frac{\text{stored capacity}}{\text{total capacity}} = \frac{\text{redundant capacity}}{\text{total capacity}}$. A deduplication ratio of 80% denotes that 80% of the data could be removed by a deduplication system and 20% of the original data would be stored. Figure 1 shows the deduplication ratios for all applications for fixed-size chunking (upper subfigure), content-defined chunking (lower subfigure), and different (average) chunk sizes. The values above the bars indicate the absolute total volume of the redundant data[1].

Except for ray, all applications show a deduplication ratio of more than 84%. Smaller chunks enable better redundancy detection. For fixed-size chunking, the maximum difference between 4 KB and 32 KB chunks for the same application is 9.8%. For CDC, the difference is 8.3%.

The white bars show the ratio of the zero chunk, i. e., the chunk that contains only zeroes. This ratio is defined as $\frac{\text{zero chunk capacity}}{\text{total capacity}}$. The zero chunk contributes significantly to the deduplication potential in enterprise backups and virtual machine images [55], [56]. In their HPC study, Meister et al. found that between 3.1% and 24.3% of their HPC data consist of zero chunks [12]. In our case, the zero chunk is the most used chunk and is the main source of redundant data for every application and chunk size, except CDC with an average chunk size of 32 KB. Note that the zero chunk has the property of

[1]Note that we ignored the last checkpoint in the figure so that pBWA could be included. Therefore, the values for the saved amount of data should not be compared to Table I which displays values for all available checkpoints.

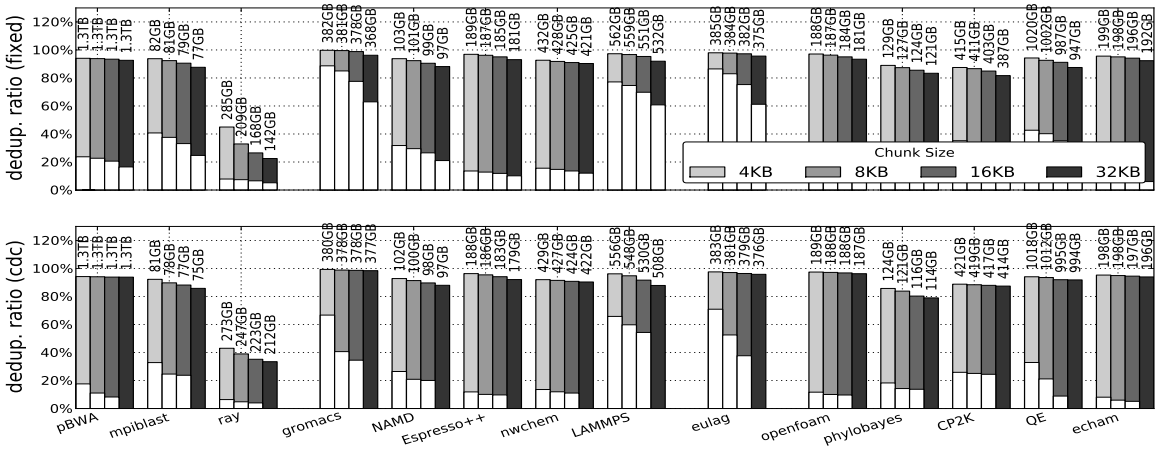

Fig. 1. Deduplication ratio of all applications. The upper figure shows the ratios for fixed-size chunking, the lower figure the ratios for content defined chunking. The values above bars show the absolute volume of the redundant chunks. The white bars indicate the relative volume of the zero chunk.

TABLE II
DEDUPLICATION RATIO AND ZERO CHUNK RATIO FOR ALL APPLICATIONS AND DIFFERENT DEDUPLICATION APPROACHES. THE VALUES FOR *single* DENOTE RATIOS FOR THE SINGLE CHECKPOINT. *window* DENOTES RATIOS OF A DEDUPLICATION OF BOTH THE GIVEN CHECKPOINT AND ITS PREDECESSOR. FINALLY, *accumulated* DENOTES A DEDUPLICATION OF ALL CHECKPOINTS UP TO THE GIVEN ONE (INCLUDING IT). ALL VALUES WERE COLLECTED FOR A 64 PROCESSES RUN AND FIXED-SIZED CHUNKING WITH A CHUNK SIZE OF 4 KB.

Application	single			window			accumulated		
	20 min	60 min	120 min	10+20 min	50+60 min	110+120 min	≤20 min	≤60 min	≤120 min
pBWA	91% (17%)	92% (17%)		92% (17%)	92% (17%)		92% (17%)	93% (17%)	
mpiblast	99% (92%)	99% (92%)	99% (91%)	99% (92%)	99% (92%)	99% (91%)	99% (92%)	99% (92%)	99% (92%)
ray	97% (77%)	39% (34%)	37% (32%)	98% (78%)	42% (33%)	50% (32%)	98% (78%)	63% (48%)	61% (39%)
bowtie	74% (23%)			88% (20%)			88% (20%)		
gromacs	99% (88%)	99% (88%)	99% (88%)	99% (88%)	99% (88%)	99% (88%)	99% (88%)	99% (88%)	99% (88%)
NAMD	81% (31%)	81% (31%)	81% (31%)	88% (31%)	88% (31%)	88% (31%)	88% (31%)	93% (31%)	94% (31%)
Espresso++	79% (13%)	79% (13%)	79% (12%)	87% (16%)	89% (12%)	89% (12%)	87% (16%)	95% (14%)	97% (13%)
nwchem	66% (12%)	89% (12%)	89% (12%)	76% (29%)	94% (12%)	94% (12%)	76% (29%)	86% (17%)	93% (15%)
LAMMPS	97% (77%)	97% (77%)	97% (77%)	97% (77%)	97% (77%)	97% (77%)	97% (77%)	97% (77%)	97% (77%)
eulag	97% (88%)	97% (85%)	97% (84%)	97% (89%)	97% (86%)	97% (84%)	97% (89%)	97% (87%)	97% (86%)
openfoam	89% (13%)	89% (13%)	89% (13%)	90% (14%)	93% (13%)	93% (13%)	90% (14%)	96% (13%)	97% (13%)
phylobayes	95% (79%)	95% (79%)	95% (78%)	96% (79%)	96% (79%)	96% (78%)	96% (79%)	97% (79%)	97% (79%)
CP2K	81% (32%)	81% (32%)	80% (32%)	89% (50%)	84% (32%)	84% (32%)	89% (50%)	87% (38%)	87% (34%)
QE	65% (55%)	57% (38%)	57% (38%)	81% (60%)	78% (38%)	78% (38%)	81% (60%)	89% (46%)	94% (42%)
echam	93% (10%)	92% (10%)	92% (10%)	94% (10%)	94% (10%)	94% (10%)	94% (10%)	95% (10%)	95% (10%)

always having the maximum chunk size if content-defined chunking is used. In our setting, this size is four times the (average) chunk size, i. e., a zero chunk for CDC, 16 KB ranges over 64 KB = 16 memory pages. Note that the zero chunk ratio for 16 KB and CDC is smaller than for 4 KB and fixed-size chunking because CDC does not preserve page alignment.

Next, we investigate how the zero chunk ratio and the deduplication potential vary over time. We determined the deduplication ratio and the zero chunk ratio for *single* checkpoints after 20min, 60min, and 120min. These values are shown in the left block (*single*) in Table II. The parenthesized values denote the zero chunk ratio. The zero chunk ratio is constant for 13 applications. Only for ray and QE we see a significant drop from 77% to 34% and 55% to 38%.

a) Change rate and garbage collection overhead: Next, we investigate the change rate of the internal data, which directly influences the garbage collection (GC) overhead of a deduplication system. During a garbage collection, a deduplication system deletes unreferenced chunks. A chunk can become unreferenced if a checkpoint is deleted and the checkpoint held the last reference. A high change rate causes a high GC overhead and is visible in form of a low deduplication ratio between consecutive checkpoints.

The middle block of Table II shows the ratios for *windowed* deduplication, i. e., the deduplication/zero ratio of both, the given checkpoint and its predecessor (10 min before) together. 13 of the 15 applications show a deduplication ratio of more than 87%. Therefore, they replace less than 13% of their volume with new chunks and induce the GC to remove 13% of the stored volume. Note that the actual fraction of the deleted chunks may be less since we compute the deduplication ratio based on the volume of changed data and not based on the number of changed chunks. In the most extreme case, a single chunk accounts for the 13% of changed volume, which would result in a negligible GC overhead of one chunk. Therefore, the 13% mark an upper bound on the GC overhead. In addition, there is no variation in the deduplication ratio over time for most applications. This causes a near constant garbage

TABLE III
COMPARISON OF APPLICATION-LEVEL AND SYSTEM-LEVEL CHECKPOINT SIZES FOR A SUBSET OF THE TESTED APPLICATIONS.

App	sys-lvl (+dedup)	app-lvl (+dedup)	$\frac{\text{sys-lvl+dedup}}{\text{app-lvl+dedup}}$
NAMD	10 GB (559 MB)	15 MB (15 MB)	37
gromacs	34 GB (83 MB)	65 KB (65 KB)	1328
LAMMPS	52 GB (1.4 GB)	1.5 MB (1.5 MB)	955
openfoam	17 GB (513 MB)	56 MB (55.9 MB)	12
CP2K	43 GB (5.4 GB)	21 MB (21 MB)	263
ray	75 GB (28 GB)	30 GB (29.6 GB)	0.93

collection overhead if the checkpoint library or tool regularly deletes old checkpoints.

b) Single vs. accumulated deduplication: The table also shows that most of the potential (exploited) lies in the deduplication of single checkpoints. For most applications, there is only a small increase of the deduplication ratio between the single checkpoints and the windowed ones. This also holds for *accumulated* deduplication, i. e., a deduplication of the current and all the previous checkpoints (right block). Only for QE and ray there is a significant deduplication ratio increase. Note that the single deduplication ratio decrease of nwchem (single/window 60min vs. acc. 60min) is caused by single checkpoints that yield a higher change rate.

Note that the values include synergy effects since we considered all 64 processes together. Therefore, the values represent a setup where all processes run on the same node. However, the impact of these effects decreases if the processes are distributed. We investigate this in Section V-D.

Overall, the zero chunk is the biggest single source of deduplication such that a zero chunk deduplication alone saves at least 10% of the checkpoint data.

c) Application-Level vs. System-Level Checkpointing: Many HPC applications provide their own checkpointing. These checkpoints can be significantly smaller because programmers know best which data is necessary to represent a computation's state. Table III compares the average checkpoint sizes of system-level checkpointing and and application-level checkpointing for a subset of the test applications. The last column in Table III displays the factors by which system-level and application-level checkpoints differ after deduplication.

For almost all applications, the average system-level checkpoint size is several orders of magnitude larger than the application-level checkpoint. However, the storage requirements for system-level checkpoints decrease significantly. In case of ray, the new storage capacity requirements is even lower than for the application-level checkpoint. Therefore, deduplicating system-level checkpoints can outperform application-level checkpointing.

Finding: There is a high deduplication potential in every application. The difference between fixed-size and content-defined chunking is small. The zero chunk is the dominant source of redundancy.

B. Stability of the Input

In this section, we take a closer look at the source of the redundancy, besides the zero chunk. In detail, we investigate whether the redundancy is defined by the input data or by data generated during the computations.

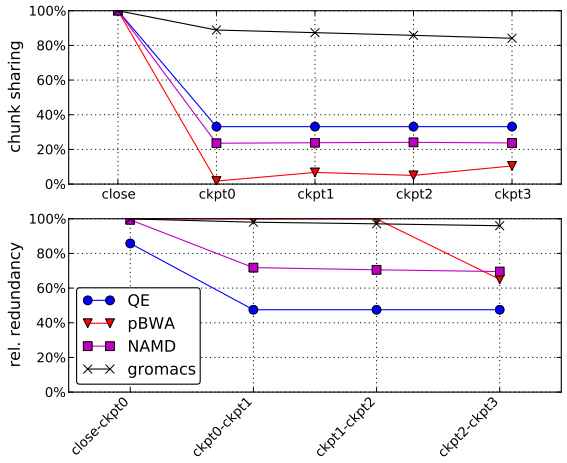

Fig. 2. Upper plot: relative volume of the input data in the following checkpoints. Lower plot: the input data's share of the redundancy in the (windowed) deduplication.

As input data, we define the content of the heap at the point the application closes the input file(s) the last time. This follows the intuition that the full state of the process up to that point contains data from initialization and the input but is untouched by the core computation.

For this, we ran single-process computations of QE, pBWA, NAMD, and gromacs and paused the computation after the last close call of the input file(s), as well as after every 10 minutes of computation afterwards. We choose these applications because of their different memory footprints. At each interrupt, we created a checkpoint by copying the full process image via the `/proc` file system. We are interested in the part of the image, which includes the input data and will contain most of the new generated data over time. Therefore, we extracted the heap part of the image and removed the data of shared libraries and the application's object code because this data is static or defined by external libraries. Next, we chunked and fingerprinted the heap with fixed-size chunking with a chunk size of 4 KB. Hence, every chunk corresponds to a memory page. We refer to the checkpoint that only includes the input data as *close-checkpoint*.

Next, we computed the input data's share of the later checkpoints by checking for each chunk of a later checkpoint whether it already existed in the close-checkpoint. The upper plot of Figure 2 shows the resulting relative volumes (chunk sharing). All shares are 100% for the close-checkpoint since a checkpoint shares every chunk with itself. As can be seen, the shares of NAMD and QE are near constant at 24% and 38%, respectively, while the share of gromacs decreases from 89% to 84%. pBWA input data's share increases from 2% to 10%. This is counterintuitive because applications generate data, which increase the checkpoint sizes and, therefore, should reduce the input data's share. A closer look showed that pBWA's checkpoints do not shrink; therefore, pBWA generates

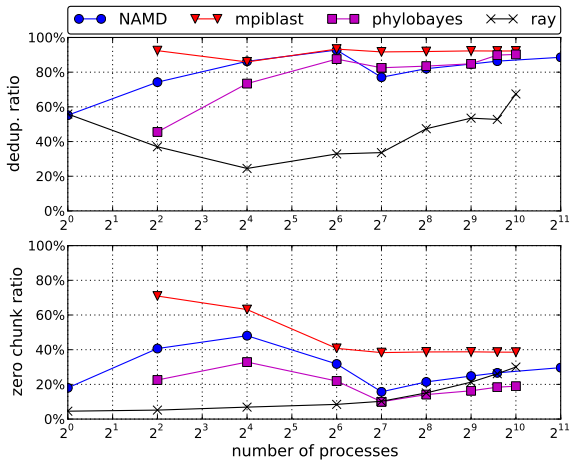

Fig. 3. Upper plot: deduplication ratio for a different number of processes. Lower plot: zero chunk ratio for the same experiments.

the share increase by copying parts of the input data internally.

However, the shared data has a big impact on the deduplication ratio. Next, we determined the input data's share of all redundant chunks. For this, we took each two consecutive checkpoints, determined the redundant chunks, and checked for each one whether it already existed in the input data. We do not use single checkpoints because this would ignore inter-checkpoint redundancy. A share value of 80% denotes that 80% of the redundant chunks also existed in the input and, therefore, that 80% of the redundancy bases on the input. The lower plot in Figure 2 shows the shares for the applications. In general, more than 48% of the redundancy bases on the input data. For all applications, the share decreases over time as they generate new data which is redundant among the checkpoints.

Finding: Most redundancy originates from input data and not from data generated during the computations.

C. Scaling Effects

In all experiments mentioned above, the number of processes was set to 64. Now we vary the number of processes and consequently also the distribution of input data and computation over the processes.

For this analysis, we selected applications from different domains, namely mpiblast, NAMD, and phylobayes, and also added ray because of its relatively low deduplication potential. We applied fixed-size chunking with a chunk size of 4 KB. For better scalability support, we changed the DMTCP version (2.4.0 instead of 2.3.1) and the MPI library (Open MPI 1.8.1 instead of MPICH 3.1) compared to the previous experiments. For this reason, the deduplication ratio in Figure 3 should not be compared to Table II. Figure 3 shows the accumulated deduplication ratio (solid lines) and the zero chunk ratio for a different number of processes.

The deduplication ratio of all tested applications but ray increases with the number of processes until 64 processes are reached. Beyond this number – which also marks the number of cores per node in our test system – we see three different behaviors: the ratios of mpiblast and phylobayes decrease, the ratio of NAMD increases after an initial drop, and the ratio of ray remains at the same level after an initial drop.

In the following, we will exclude the zero chunk from our analysis because its deduplication is free and usually receives special treatment in deduplication systems. Therefore, this chunk has a minor impact on the design of a deduplication system.

Finding: The deduplication potential is high, independent of the number of processes. The zero chunk is the dominant source of redundancy, even for a large number of processes.

D. Local vs. Global Deduplication

Most of the applications on exascale computers will use many cores or even many compute nodes. A deduplication system designer can choose to only deduplicate the checkpoint data *node-locally*, i. e., only deduplicate all processes running on the same compute node, *grouped*, i. e., deduplicate the processes of a subset of all compute nodes, or *globally*, i. e., deduplicate all processes from all nodes together. This design decision is critical, yet not simple: a node-local deduplication system has a comparably low complexity, whereas global deduplication yields a higher savings potential. Nicolae reports that global deduplication reduces the checkpoint volume to 5% compared to 30% for node-local deduplication [57].

In the following, we examine the deduplication potential for different degrees of clustered deduplication. We group all processes of a 64 processes run in incrementally growing group sizes. This represents a computation that is distributed across n nodes where $\frac{64}{n}$ processes run on the same compute node. Note that each run includes two additional MPI management processes that are spawned by the MPI environment itself and not part of the core computation. They increase the variance among the groups because their images do not contain computation data and the process groups do not have the same size. Next, we determine the deduplication ratio of all groups and compute the average. Figure 4 shows the average ratios of two consecutive checkpoints. The error bars indicate the quartiles of the deduplication ratios among the groups.

As the figure shows, bigger groups increase the deduplication ratio. Therefore, there is data sharing among the processes besides the zero chunk. The average deduplication ratio increases between 3% and 39%. For pBWA, we monitored a bigger fluctuation in the processes' behavior. Therefore, the deduplication ratio varies stronger.

The average deduplication ratio of the single-element groups is bigger than the ratio increase based on grouping. This shows that a deduplication system can exploit most of the deduplication potential by deduplicating the checkpoints of the same processes against each other.

Finding: Node-local deduplication yields the biggest savings. However, these savings can be significantly increased with global deduplication.

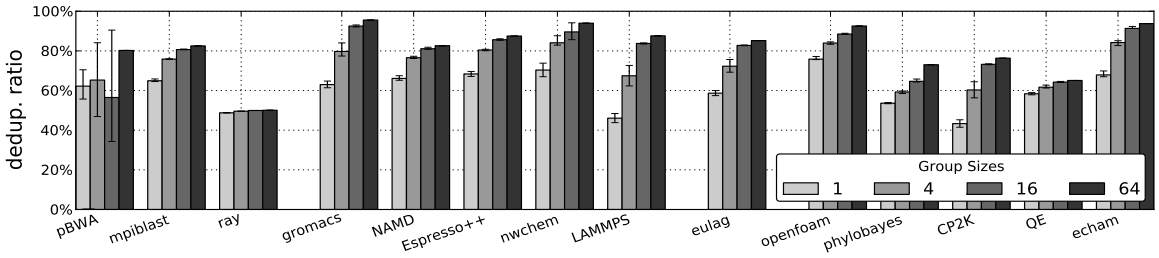

Fig. 4. Average deduplication ratio of all applications for different group sizes while ignoring the zero chunk. The chunking method was fixed-size chunking with 4 KB chunks. The error bars show the quartiles of the deduplication ratio among the groups. Since the zero chunks are removed from the data set, the values for 64 processes in this figure and the node-local values in Table II do not match.

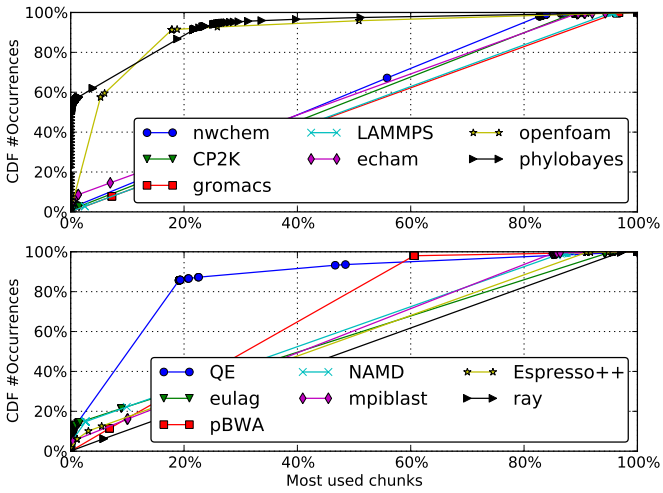

Fig. 5. Chunk bias of the most used chunks for the 10th checkpoint of a 64 processes computation. A point states that the first $x\%$ of the most used chunks account for $y\%$ of all chunk occurrences.

E. Chunk Biases

a) Chunk Bias: We have seen that the zero chunk typically contributes more to the redundancy than any other chunk. Apart from the zero chunk, there are usually chunks that occur more often than others so that the chunk usage distribution is more or less skewed. To investigate this, we look at the 10th checkpoint of a 64 processes run of all applications.

For 11 of the 14 applications, more than 86% of all chunks were referenced only once within a checkpoint, i. e., these chunks are unique and do not contribute to the deduplication. For the other applications, this value ranges from 68% to 81%.

Figure 5 shows the cumulative density function (CDF) for the most referenced chunks that contribute to the deduplication. A data point (x, y) expresses that the first $x\%$ of the most used chunks account for $y\%$ of all chunk occurrences.

The straight lines are caused by chunks that appear in every process. These chunks amount to about 80% of all redundant chunks and create about 95% of all occurrences.

Finding: In most applications, there is no significant chunk bias, disregarding the zero chunk.

b) Process Bias: Next, we investigate the process bias of the chunks, i. e., how the chunks are distributed among the processes of the applications. For this, we analyze the 10th checkpoint of a 64 processes run of each application. For each chunk in a checkpoint, we count the number of processes the chunk occurs in. We generate the CDF of this statistic, which is shown in the upper plots of Figure 6. As one can see, most chunks (80-98%) occur in only one process while the other chunks occur in almost every process. Thus, the processes of most applications hardly share any chunks. Note that all chunks that occur in more than one process contribute to the redundancy of the checkpoint as each chunk has at least one duplicate by definition. In addition, each of these chunks can occur multiple times within the processes.

The distribution of the shared volume, however, does not follow the same distribution. We investigated this performing the same steps of the upper plots, but counting the *volume of all occurrences* of the chunks instead. The resulting CDF is shown in the lower part of Figure 6. For most applications, 6-21% of the checkpoint volume is not shared among the processes. For most applications, between 82% and 94% of the checkpoint volume consists of chunks that occur in every process. This volume is also visible in Figure 5 in form of the long, straight lines.

Note that the lines do not stop at 64 processes because of the additional MPI processes mentioned in Section V-D.

Finding: Within one checkpoint, there is a small amount of different chunks (2-20%, upper plots in Figure 6) that 1) occur in most processes and 2) account for the majority of the checkpoint volume (82-94%, lower plots).

VI. Conclusion

In this work, we studied the deduplication potential of system-level checkpoints for a broad range of HPC applications. We can summarize that all applications show significant savings potential independent of their domain and their underlying computation model; the potential ranges from 37% to 99%. The results suggest that some applications sustain a high potential for a larger number of nodes. The evaluation further shows that even rather simple deduplication approaches can eliminate most of the redundant data. For example, removing the most frequent chunk, the zero chunk, reduces the checkpoint data by 10-92%.

In our experiments, deduplication could reduce the storage requirements of system-level checkpointing by several orders

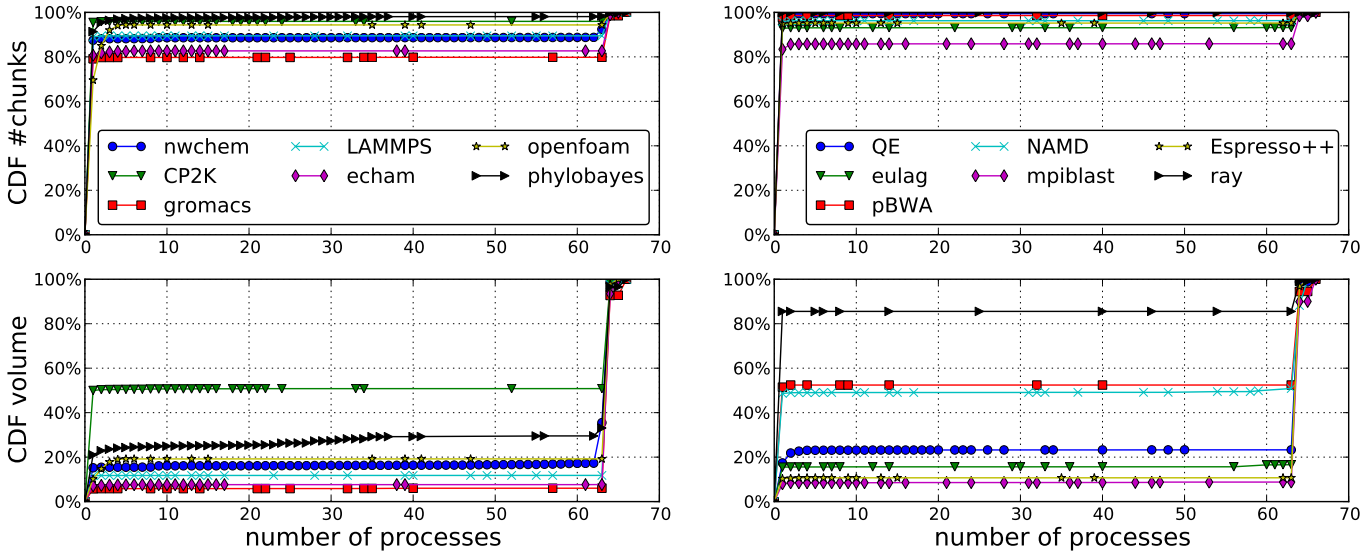

Fig. 6. Bias of the chunk distribution among the processes. The upper figures show the CDF for chunk sharing, where every chunk is counted only once. The lower figures show the CDF for the case in which every occurrence of a chunk is counted. The values were computed based on the 10th checkpoint.

of magnitude, but application-level checkpointing, with one exception, still required at least one order of magnitude less storage space.

Due to the given page structure, fixed-size chunking is a natural choice. In contrast to traditional deduplication systems, content-defined chunking does not detect redundancy better.

While we have focused on the deduplication potential, we have not discussed how to perform deduplication for checkpointing in a fast way. We hope that this study provides a solid foundation for the design of future deduplication systems.

Acknowledgment

This work was partially supported by the German Ministry for Education and Research (BMBF) under project grant 01IH13004 (Project FAST) and the Intel Parallel Computing Centers (IPCC) project.

References

[1] Top 500 supercomputer. [Online]. Available: http://top500.org/

[2] T. Z. Islam, K. Mohror, S. Bagchi, A. Moody, B. R. de Supinski, and R. Eigenmann, "McrEngine: A Scalable Checkpointing System Using Data-aware Aggregation and Compression," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE, 2012, pp. 17:1–17:11.

[3] J. Dongarra, P. Beckman, T. Moore, P. Aerts, G. Aloisio, J.-C. Andre, D. Barkai, J.-Y. Berthou, T. Boku, B. Braunschweig, F. Cappello, B. Chapman, X. Chi, A. Choudhary, S. Dosanjh, T. Dunning, S. Fiore, A. Geist, B. Gropp, R. Harrison, M. Hereld, M. Heroux, A. Hoisie, K. Hotta, Z. Jin, Y. Ishikawa, F. Johnson, S. Kale, R. Kenway, D. Keyes, B. Kramer, J. Labarta, A. Lichnewsky, T. Lippert, B. Lucas, B. Maccabe, S. Matsuoka, P. Messina, P. Michielse, B. Mohr, M. S. Mueller, W. E. Nagel, H. Nakashima, M. E. Papka, D. Reed, M. Sato, E. Seidel, J. Shalf, D. Skinner, M. Snir, T. Sterling, R. Stevens, F. Streitz, B. Sugar, S. Sumimoto, W. Tang, J. Taylor, R. Thakur, A. Trefethen, M. Valero, A. Van Der Steen, J. Vetter, P. Williams, R. Wisniewski, and K. Yelick, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[4] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Karp, S. Keckler, D. Klein, R. Lucas, M. Richards, A. Scarpelli, S. Scott, A. Snavely, T. Sterling, R. S. Williams, K. Yelick, K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, J. Hiller, S. Keckler, D. Klein, P. Kogge, R. S. Williams, and K. Yelick, "Exascale computing study: Technology challenges in achieving exascale systems peter kogge, editor & study lead," 2008.

[5] J. N. Glosli, D. F. Richards, K. J. Caspersen, R. E. Rudd, J. A. Gunnels, and F. H. Streitz, "Extending stability beyond cpu millennium: A micron-scale atomistic simulation of kelvin-helmholtz instability," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing*, ser. SC '07. New York, NY, USA: ACM, 2007, pp. 58:1–58:11.

[6] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys (CSUR)*, vol. 34, no. 3, pp. 375–408, 2002.

[7] D. A. Reed and J. Dongarra, "Exascale computing and big data: The next frontier," in *submitted Communications of the ACM*, April 2014.

[8] B. Zhu, K. Li, and R. H. Patterson, "Avoiding the Disk Bottleneck in the Data Domain Deduplication File System," in *Proceedings of the 6th USENIX Conference on File and Storage Technologies (FAST)*, 2008, pp. 269–282.

[9] M. Lillibridge, K. Eshghi, D. Bhagwat, V. Deolalikar, G. Trezis, and P. Camble, "Sparse Indexing: Large Scale, Inline Deduplication Using Sampling and Locality," in *Fast*, vol. 9, 2009, pp. 111–123.

[10] D. Bhagwat, K. Eshghi, D. D. E. Long, and M. Lillibridge, "Extreme Binning: Scalable, Parallel Deduplication for Chunk-based File Backup," in *Proceedings of the 17th IEEE International Symposium on Modeling, Analysis, and Simulation (MASCOTS)*, 2009, pp. 1–9.

[11] W. Xia, H. Jiang, D. Feng, and Y. Hua, "SiLo: A Similarity-Locality based Near-Exact Deduplication Scheme with Low RAM Overhead and High Throughput," in *Proceedings of the USENIX Annual Technical Conference (ATC)*, 2011.

[12] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A study on data deduplication in hpc storage systems," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '12. IEEE, 2012, pp. 1–11.

[13] B. Nicolae, "Towards scalable checkpoint restart: A collective inline memory contents deduplication proposal," in *Proceedings of the 27th International Symposium on Parallel and Distributed Processing*, ser. IPDPS '13. IEEE Computer Society, 2013, pp. 19–28.

[14] A. Kulkarni, A. Manzanares, L. Ionkov, M. Lang, and A. Lumsdaine, "The design and implementation of a multi-level content-addressable checkpoint file system," in *Proceedings of the 19th International Conference on High Performance Computing(HiPC)*. IEEE, 2012, pp. 1–10.

[15] J. Duell, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart," LBL, Tech. Rep., 2003.

[16] J. Ansel, K. Arya, and G. Cooperman, "DMTCP: Transparent checkpointing for cluster computations and the desktop," in *23rd IEEE International Parallel and Distributed Processing Symposium*, 2009.

[17] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "**Libckpt**: Transparent checkpointing under Unix," in *USENIX Winter Technical Conference*, January 1995, pp. 213–223.

[18] G. Bronevetsky, D. Marques, K. Pingali, P. Szwed, and M. Schulz, "Application-level checkpointing for shared memory programs," *SIGPLAN Not.*, vol. 39, no. 11, pp. 235–247, Oct. 2004.

[19] G. Bronevetsky, K. Pingali, and P. Stodghill, "Experimental evaluation of application-level checkpointing for openmp programs," in *Proceedings of the 20th Annual International Conference on Supercomputing*, ser. ICS '06. New York, NY, USA: ACM, 2006, pp. 2–13.

[20] M. Schulz, G. Bronevetsky, R. Fernandes, D. Marques, K. Pingali, and P. Stodghill, "Implementation and evaluation of a scalable application-level checkpoint-recovery scheme for mpi programs," in *Proceedings of the 2004 ACM/IEEE Conference on Supercomputing*, ser. SC '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 38–.

[21] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in an application-level fault tolerant mpi system," in *In International Conference on Supercomputing (ICS)*, 2003, pp. 234–243.

[22] ——, "Automated application-level checkpointing of mpi programs," in *Proceedings of the Ninth ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '03. New York, NY, USA: ACM, 2003, pp. 84–94.

[23] D. Ibtesham, D. Arnold, P. G. Bridges, K. B. Ferreira, and R. Brightwell, "On the viability of compression for reducing the overheads of checkpoint/restart-based fault tolerance," in *2012 41st International Conference on Parallel Processing*, Sept 2012, pp. 148–157.

[24] B. Nicolae and F. Cappello, "Ai-ckpt: Leveraging memory access patterns for adaptive asynchronous incremental checkpointing," in *Proceedings of the 22Nd International Symposium on High-performance Parallel and Distributed Computing*, ser. HPDC '13. New York, NY, USA: ACM, 2013, pp. 155–166. [Online]. Available: http://doi.acm.org/10.1145/2462902.2462918

[25] R. Gioiosa, J. Sancho, s. jiang, and F. Petrini, "Transparent, incremental checkpointing at kernel level: a foundation for fault tolerance for parallel computers," in *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, Nov 2005, pp. 9–9.

[26] M. Vasavada, F. Mueller, P. H. Hargrove, and E. Roman, "Comparing different approaches for incremental checkpointing: The showdown," Linux'11: The 13th Annual Linux Symposium, 2011.

[27] I. Cores, G. Rodriguez, M. Martin, and P. Gonz'lez, "Reducing application-level checkpoint file sizes: Towards scalable fault tolerance solutions," in *IEEE 10th International Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2012, pp. 371–378.

[28] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," in *Proceedings of the 18th ACM Symposium on Operating Systems Principles*, ser. SOSP '01, 2001, pp. 174–187.

[29] A. Moody, G. Bronevetsky, K. Mohror, and B. De Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *High Performance Computing, Networking, Storage and Analysis (SC), 2010 International Conference for*, Nov 2010, pp. 1–11.

[30] A. Darling, L. Carey, and W.-c. Feng, "The design, implementation, and evaluation of mpiBLAST," *ClusterWorld*, pp. 13–15, 2003.

[31] D. Peters, X. Luo, K. Qiu, and P. Liang, "Speeding up large-scale next generation sequencing data analysis with pbwa," *Journal of Applied Bioinformatics & Computational Biology*, vol. 1, no. 1, 2012.

[32] H. Li and R. Durbin, "Fast and accurate short read alignment with Burrows–Wheeler transform," *Bioinformatics*, vol. 25, no. 14, pp. 1754–1760, 2009.

[33] B. Langmead, C. Trapnell, M. Pop, and S. Salzberg, "Ultrafast and memory-efficient alignment of short DNA sequences to the human genome," *Genome Biology*, vol. 10, no. 3, 2009.

[34] "pMap: Parallel Sequence Mapping Tool," http://bmi.osu.edu/hpc/software/pmap/pmap.html.

[35] N. Lartillot, T. Lepage, and S. Blanquart, "PhyloBayes 3: a Bayesian software package for phylogenetic reconstruction and molecular dating," *Bioinformatics*, vol. 25, no. 17, pp. 2286–2288, 2009.

[36] S. Boisvert, F. Laviolette, and J. Corbeil, "Ray: Simultaneous Assembly of Reads from a Mix of High-Throughput Sequencing Technologies," *Journal of Computational Biology*, vol. 17, no. 11, pp. 1519–1533, 2010.

[37] J. D. Halverson, T. Brandes, O. Lenz, A. Arnold, S. Bevc, V. Starchenko, K. Kremer, T. Stuehn, and D. Reith, "ESPResSo++: A modern multiscale simulation package for soft matter systems," *Computer Physics Communications*, vol. 184, no. 4, pp. 1129–1149, 2013.

[38] B. Hess, C. Kutzner, D. Van Der Spoel, and E. Lindahl, "Gromacs 4: algorithms for highly efficient, load-balanced, and scalable molecular simulation," *Journal of chemical theory and computation*, vol. 4, no. 3, pp. 435–447, 2008.

[39] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *Journal of Computational Physics*, vol. 117, no. 1, pp. 1–19, 1995.

[40] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[41] L. V. Kale and S. Krishnan, "Charm++: A portable concurrent object oriented system based on c++," in *Proceedings of the Eighth Annual Conference on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '93. ACM, 1993, pp. 91–108.

[42] M. Valiev, E. Bylaska, N. Govind, K. Kowalski, T. Straatsma, H. V. Dam, D. Wang, J. Nieplocha, E. Apra, T. Windus, and W. de Jong, "Nwchem: A comprehensive and scalable open-source solution for large scale molecular simulations," *Computer Physics Communications*, vol. 181, no. 9, pp. 1477 – 1489, 2010.

[43] J. Hutter, M. Iannuzzi, F. Schiffmann, and J. VandeVondele, "cp2k: atomistic simulations of condensed matter systems," *Wiley Interdisciplinary Reviews: Computational Molecular Science*, vol. 4, no. 1, pp. 15–25, 2014.

[44] P. Giannozzi, S. Baroni, N. Bonini, M. Calandra, R. Car, C. Cavazzoni, D. Ceresoli, G. L. Chiarotti, M. Cococcioni, I. Dabo, A. D. Corso, S. de Gironcoli, S. Fabris, G. Fratesi, R. Gebauer, U. Gerstmann, C. Gougoussis, A. Kokalj, M. Lazzeri, L. Martin-Samos, N. Marzari, F. Mauri, R. Mazzarello, S. Paolini, A. Pasquarello, L. Paulatto, C. Sbraccia, S. Scandolo, G. Sclauzero, A. P. Seitsonen, A. Smogunov, P. Umari, and R. M. Wentzcovitch, "Quantum espresso: a modular and open-source software project for quantum simulations of materials," *Journal of Physics: Condensed Matter*, vol. 21, no. 39, 2009.

[45] J. M. Prusa, P. K. Smolarkiewicz, and A. A. Wyszogrodzki, "Eulag, a computational model for multiscale flows," *Computers & Fluids*, vol. 37, no. 9, pp. 1193–1207, 2008.

[46] J. Li, W. keng Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale, "Parallel netCDF: A High-Performance Scientific I/O Interface," *SC Conference*, vol. 0, p. 39, 2003.

[47] "ECHAM project website," http://www.mpimet.mpg.de/en/science/models/echam.html.

[48] H. Jasak, A. Jemcov, and Z. Tukovic, "OpenFOAM: A C++ library for complex physics simulations," in *Proceedings of the International Workshop on Coupled Methods in Numerical Dynamics*, 8 2007.

[49] D. Meister, "FS-C, File System Chunking Tool Suite, version 0.3.9," https://github.com/dmeister/fs-c, 2011.

[50] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A Study on Data Deduplication in HPC Storage Systems," in *Proceedings of International Conference on High Performance Computing, Networking, Storage and Analysis (SC)*. IEEE, 2012, pp. 7:1–7:11.

[51] J. Kaiser, D. Meister, A. Brinkmann, and T. Süß, "Deriving and Comparing Deduplication Techniques Using a Model-Based Classification," in *Proceedings of the European Conference on Computer Systems (EuroSys)*. ACM, 2015.

[52] M. O. Rabin, "Fingerprinting by Random Polynomials," TR-15-81, Center for Research in Computing Technology, Harvard University, Tech. Rep., 1981.

[53] S. Barker, T. Wood, P. Shenoy, and R. Sitaraman, "An Empirical Study of Memory Sharing in Virtual Machines," in *Proceedings of the 2012 USENIX Annual Technical Conference (ATC 12)*, 2012, pp. 273–284.

[54] K. Miller, F. Franz, M. Rittinghaus, M. Hillenbrand, and F. Bellosa, "Xlh: More effective memory deduplication scanners through cross-layer hints," in *Presented as part of the 2013 USENIX Annual Technical Conference (USENIX ATC 13)*. USENIX, 2013, pp. 279–290.

[55] K. Jin and E. L. Miller, "The effectiveness of deduplication on virtual machine disk images," in *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)*. ACM, 2009.

[56] D. Meister and A. Brinkmann, "Multi-level Comparison of Data Deduplication in a Backup Scenario," in *Proceedings of the 2nd Israeli Experimental Systems Conference (SYSTOR)*. ACM, 2009, pp. 1–12.

[57] B. Nicolae, "Leveraging naturally distributed data redundancy to collective I/O replication overhead," in *Proceedings of the 29th IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, ser. IPDPS '15. Washington, DC, USA: IEEE Computer Society, 2015.