Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead

Bogdan Nicolae

Leveraging naturally distributed data redundancy to reduce collective I/O replication overhead

Bogdan Nicolae
IBM Research, Ireland
bogdan.nicolae@ie.ibm.com

***Abstract*—Dumping large amounts of related data simultaneously to local storage devices instead of a parallel file system is a frequent I/O pattern of HPC applications running at large scale. Since local storage resources are prone to failures and have limited potential to serve multiple requests in parallel, techniques such as replication are often used to enable resilience and high availability. However, replication introduces overhead, both in terms of network traffic necessary to distribute replicas, as well as extra storage space requirements. To reduce this overhead, state-of-art techniques often apply redundancy elimination (e.g. compression or deduplication) before replication, ignoring the natural redundancy that is already present. By contrast, this paper proposes a novel scheme that treats redundancy elimination and replication as a single co-optimized phase: remotely duplicated data is detected and directly leveraged to maintain a desired replication factor by keeping only as many replicas as needed and adding more if necessary. In this context, we introduce a series of high performance algorithms specifically designed to operate under tight and controllable constrains at large scale. We present how this idea can be leveraged in practice and demonstrate its viability for two real-life HPC applications.**

***Index Terms*—data resilience; high availability; data replication; deduplication; collective I/O scalability; redundancy management**

I. Introduction

Scientific and data-intensive computing have matured over the last couple of years in all fields of science and industry. Their rapid increase in complexity and scale has prompted ongoing efforts dedicated to reach exascale infrastructure capability by the end of the decade. However, advances in this context are not homogeneous: I/O capabilities in terms of networking and storage are lagging behind computational power and are often considered a major limitation that that persists even at petascale [1].

A particularly difficult challenge in this context are collective I/O access patterns where all processes simultaneously dump large amounts of related data simultaneously to persistent storage (which we henceforth refer to as *collective dump*). This pattern is often exhibited by large-scale, bulk-synchronous applications in a variety of circumstances, e.g., when they use checkpoint-restart fault tolerance techniques to save intermediate computational states at regular time intervals [2] or when intermediate, globally synchronized results are needed during the lifetime of the computation (e.g. to understand how a simulation progresses during key phases). Under such circumstances, a decoupled storage system (e.g. a parallel file system such as *GPFS* [3]) does not provide sufficient I/O bandwidth to handle the explosion of data sizes: for example, Jones et al. [4] predict dump times in the order of several hours.

In order to overcome the I/O bandwidth limitation, one potential solution is to equip the compute nodes with local storage (i.e., HDDs, SSDs, NVMs, etc.). Using this approach, a large part of the data can be dumped locally, which completely avoids the need to consume and compete for the I/O bandwidth of a decoupled storage system. However, this is not without drawbacks: the local storage devices are prone to failures and as such the data they hold is volatile. Furthermore, the availability of the data may also suffer under concurrency due to the limited I/O bandwidth of the local storage devices and/or network links.

Partner replication is a technique often used to mitigate the limitations of using local storage devices: instead of storing only one local copy of the dataset, a predefined number of extra copies are send remotely to the local storage devices of other compute nodes. Using this approach, resilience and high availability of the data can be achieved in a scalable fashion by leveraging the network bandwidth allocated to the compute nodes for communication, which is often orders of magnitude higher than the I/O bandwidth of a decoupled storage system.

However, with increasing scale, partner replication quickly hits on an important limitation: due to an increasing failure rate and an increasing number of processes potentially interested in a dataset, it is necessary to increase the replication factor in order to guarantee the same level of resilience and/or availability. As a consequence, the processes need to send more data to each other: this increases the network bandwidth contention because of larger data transfers, as well as the space utilization and I/O pressure on the local storage devices because more data is received from other processes. Ultimately, both aspects introduce an overhead that not only negatively impacts performance, but also increases operational costs (e.g. need to buy larger local storage devices).

Thus, it is important to be able to achieve a high replication factor with minimal overhead. A common strategy in this context is to apply some form of redundancy elimination (i.e., compression or deduplication) before the replication, under the assumption that it leads to a significant reduction of replication overhead, which improves overall performance and reduces resource utilization. How-

ever, although straightforward, this two-phase approach is not optimal: first, an effort is made to eliminate data redundancy, only to reintroduce it later through replication.

In this paper, it is precisely this co-optimization aspect that we explore. Inspired by several studies that confirm high data redundancy for HPC workloads (such as Meister et al. [5] and our own previous work [6]), we propose to identify any data redundancy that already exists across distributed processes and group together duplicated data into natural replicas. Using this approach, redundancy elimination and partner replication are only selectively needed when a data piece is duplicated by more and, respectively, by less than the desired replication factor. We summarize our contributions as follows:

- We present a series of design principles that facilitate efficient deduplication of distributed chunks, eliminating those that are remotely duplicated beyond a fixed replication factor and evenly distributing the partner replication workload for the remaining chunks among the processes to achieve load balancing. Furthermore, all processes closely coordinate to help each other out and minimize the overhead of network traffic using single-sided communication. (Section III-B)
- We show how to materialize these design principles in practice through a series of algorithmic descriptions that are applied to implement an I/O library that exposes a dedicated collective I/O write primitive at application level. This library is then integrated with the *AC-FTE* [7] fault tolerance runtime, which leverages the collective I/O write capabilities of the library in the context of checkpoint-restart. (Sections III-C and IV)
- We evaluate our approach in a series of experiments conducted on the Shamrock testbed, using two representative real-life HPC applications that exhibit a high degree of redundancy in the context of checkpoint-restart. Our experiments demonstrate a large reduction of performance overhead and resource utilization compared to techniques that are not aware of naturally distributed duplicates. (Section V)

II. Related work

Replication is a widely used technique to improve resilience and high availability in parallel file systems and other special purpose storage services [3], [8], [9], [10].

However, due to limited scalability of remote I/O accesses, local storage devices saw an increasing adoption. At first, they were exploited as an intermediate write cache layer that is used to flush the application data asynchronously to remote storage systems in the background. This was for example introduced in the context of multi-level checkpointing [11], [12], node-level aggregation of I/O from multiple cores [13], or I/O forwarding [14]. To avoid or at least limit the need to make use of a remote storage system, several proposals aim to directly make local storage resilient either through point-to-point replication [15] or erasure codes [16].

Reducing the amount of replicated data is possible either using compression [17], [18] or deduplication. The latter broadly falls into two categories: *static* and *content-defined*. Static approaches split the input data into small, fixed-sized chunks that are then compared to each other, either directly or by using fingerprints. Since comparing only hash values increases speed at the expense of false positives (due to potential collisions), some approaches [19] even combine hash comparisons with direct comparisons in order to be able to leverage computationally cheap hash functions. Content defined approaches [20] on the other hand use a variable chunk size calculated using a sliding window over the data that hashes the window content at each step using Rabin's fingerprinting method [21]. This approach was used in several storage systems [22], [23].

Different studies of block replication and erasure codes [24] were performed before in the context of RAID systems [25], whose implementation is at hardware level, as well as for distributed data storage [26] implemented at the software level. Several works such as DiskReduce [27] and Zhe Zhang et al. [28] study the feasibility of replacing three-way replication with erasure codes in cloud storage and large data centers. Such techniques can complement our approach in the sense that data not duplicated to a sufficient degree can be made resilient through erasure codes as an alternative to replication.

Our own previous work [6] focuses on the benefits of eliminating duplicates at global level, before datasets are collectively written to persistent storage. Although closely related, the goal of our previous work is to minimize redundancy, which is precisely what we want to avoid in this work: it focuses on how to efficiently apply inter-process deduplication techniques in the context of partner replication in order to leverage naturally available distributed duplicated data pieces as natural replicas without compromising the desired resilience and high availability level, while at the same time decreasing performance overhead and resource utilization. To our best knowledge, *we are the first to explore the benefits of deduplication under such circumstances.*

III. System design

A. Assumptions

We target applications that are composed of a set of tightly coupled processes (also referred to as *ranks*) that need to simultaneously write a local dataset (potentially related to the other datsets) during runtime. A typical example of this I/O access pattern is exhibited by checkpoint-restart: at regular intervals, all processes save checkpointing information that can be later used to restart in case of failures. While we mainly use this scenario in Section V to demonstrate the benefits of our approach experimentally, our proposal is general enough to address other related scenarios as well, e.g. dumping of visualization output

during a numerical simulation. To this end, we define a collective I/O primitive that acts as a synchronization point and is used by all processes to specify the local dataset and initiate the parallel write:

$$\text{DUMP\_OUTPUT}(buffer, K)$$

For simplicity, we assume the local dataset $buffer$ resides in the memory of each process and needs to be written on the local storage device of the node that hosts the process. Note that $buffer$ does not necessarily need to be a contiguous region. Since the local storage device and the node as a whole is prone to failures, we also assume that the data needs to be replicated to at least $K-1$ other remote nodes and stored on their local storage devices as well. For the rest of this paper, we refer to K as the *replication factor* and to the $K-1$ remote nodes as the *replication partners* of the initial node. Although a common scenario, it is not required for all processes to write the same amount of data. Our goal is to maximize the performance of the DUMP_OUTPUT primitive while minimizing its resource utilization (i.e. storage space on the local devices and network traffic due to communication with replication partners).

B. Design overview

Our proposal relies on four key design principles, which are visually illustrated using an example in Figure 1.

Identify natural redundancy through collective inter-process deduplication: The central idea of of our approach is to identify the data pieces that are already duplicated between multiple processes hosted on different nodes, such that it is possible to replicate only the data pieces that are not already naturally duplicated at least as much as the desired replication factor.

To this end, we split the local dataset into small fixed sized chunks and compute a hash value (called *fingerprint*) for each chunk that "uniquely" represents it (the term unique is abused here, because hash collisions are theoretically possible but negligible in practice [6]). By using fingerprints, the complexity of the problem is greatly reduced, as comparisons and exchanges between partners involve only a small fraction of the original size of the local dataset. Based on this observation, we introduce a two-phase deduplication strategy: in the first phase, each process identifies the duplicate chunks of its own dataset and keeps only one copy, which results in a set of locally unique fingerprints. In the second phase, the processes identify the frequency of each fingerprint (i.e., number of processes where it shows up). Depending on the frequency of each fingerprint, duplicates of the corresponding chunk are either added or removed to match the desired replication factor.

How to identify the frequency of fingerprints in the second phase is a non-trivial issue: at large scale, the number of fingerprints can quickly explode, making an exact solution expensive to compute. Thus, we aim to enforce an upper bound on the computational complexity of the problem while accepting a non-optimal solution. To this end, we relax the problem in the sense that we select only a maximum of F fingerprints for which we count the frequency, while considering the rest of them unique even if they are not. Although this relaxation does not affect correctness, the quality of the deduplication is highly dependent on which F fingerprints are chosen. Obviously, selecting the most frequent F fingerprints would maximize the deduplication potential, however, it is not possible to rank the fingerprints apriori without computing an exact solution.

To deal with this dilemma, we propose an efficient (logarithmic in the number of processes) reduction-based algorithm that performs both the selection and the frequency counting in a hierarchic bottom-up fashion. More specifically, it is based on a merge step that given two sets of fingerprints and the frequency of their appearance, outputs the F most frequent fingerprints of the union (the frequency of a fingerprint in the union is defined as the sum of its frequencies in the two sets). This merge step is performed in parallel starting from the initial fingerprints of pairs of processes until a single set of F fingerprints remains. Besides counting the frequency, the merge step also associates at most K processes for each fingerprint (which we refer to as *designated ranks*). Thus, the end result is a set of F fingerprints, each of which is mapped to its frequency and set of designated ranks. Once this global view is obtained, it is broadcast to all processes.

At this point, each process can consult the global view in order to check whether there is any fingerprint it holds for which it is *not* among the K designated ranks. If that is the case, then it means there are other K processes designated to store the chunk corresponding to the fingerprint, so it can be safely discarded as the desired replication factor was reached. Otherwise, for each of the remaining fingerprints (regardless whether it is in the global view or not), it needs to store the corresponding chunk locally. If the fingerprint of the chunk is in the global view and the number of designated ranks D is less than K, the chunk needs to be replicated to $K-D$ remote partners. This can happen in parallel: each of the D designated ranks will be assigned to a subset of the $K-D$ partners (distributed in a round-robin fashion) and will send the chunk to all members of the subset. Otherwise, if the fingerprint of the chunk is not in the global view, each process needs to select $K-1$ partners and send them copies of the chunk. All chunks received by a process from its partners are saved locally together with the other chunks. Finally, once all processes have finished receiving and saving the chunks, the collective dump completes.

Load balancing by means of uniform rank assignment: How the designated ranks for each fingerprint are assigned in the reduction process is important. To better illustrate this point, consider the extreme case when all processes need to store the same local dataset that is not possible to deduplicate locally (i.e. all local chunks are

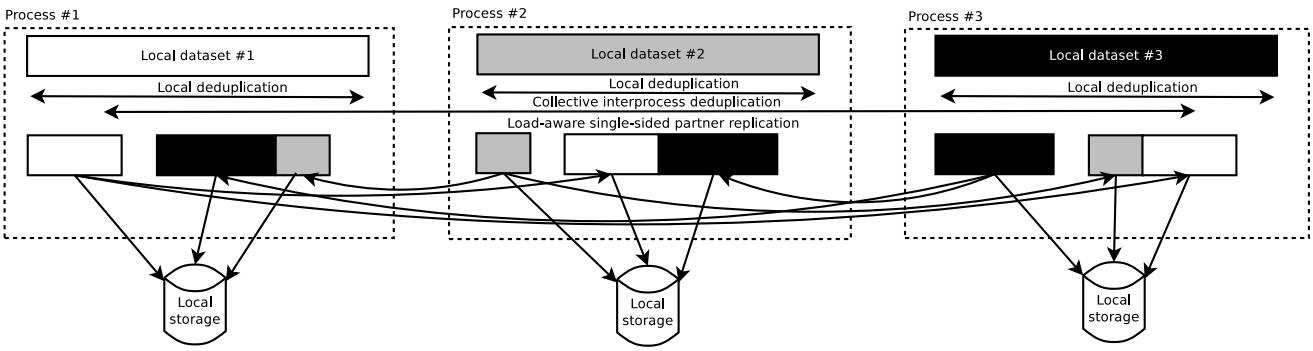


Fig. 1. Our approach in a nutshell: Example with three processes that call the DUMP_OUTPUT primitive with a replication factor $K = 3$.

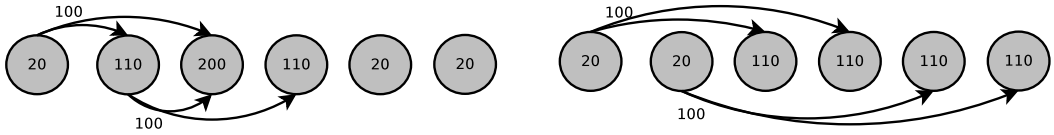


Fig. 2. Naïve partner selection (left) vs. load aware partner selection (right) for a replication factor of three: first two processes send 100 chunks (illustrated by arrows), the rest send 10 chunks (not illustrated). The value of each node is the total number of received chunks. Using a rank shuffling of (1,3,4,2,5,6) spreads the load more evenly: the maximum number of received chunks is lowered from 200 (left) to 110 (right).

unique). In this case, only K copies of the dataset needs to be stored overall. However, if we use a naive solution that designates the same ranks for each chunk, we end up with K processes that are fully loaded and a large number of remaining processes that are idle and simply need to wait until all K designated processes have finished.

Thus, a much better idea is to try to assign the ranks in such way that the overall load is evenly distributed, which speeds up the most loaded (and thus slowest) process, effectively leading to a better overall performance. To this end, we propose a load-balancing algorithm that is embedded into the merge phase. More specifically, for each process we count the number of fingerprints it was designated for. Whenever we need to merge two fingerprints, if the combined list of ranks is larger than K, we truncate it in such way that the most loaded ranks are eliminated first, effectively shifting the assignment in favor of the lesser loaded processes. Thus, as the reduction progresses, the rank assignment for a particular fingerprint is constantly changing in order to reflect a better global load balancing.

Reduce unavoidable imbalance using load-aware partner selection: Even in the ideal case when the load corresponding to the F fingerprints is evenly spread, the remaining chunks that were not identified as duplicates may not be themselves evenly spread, which can lead to an overall imbalance. However, the negative consequences of this "unavoidable" imbalance can be limited by choosing the partners in an optimal configuration. To illustrate this point, consider six processes that need to replicate their chunks to two partners ($K = 3$). Let's assume all processes have 20 chunks in common. In this case, after applying the collective deduplication strategy mentioned above using a uniform rank assignment, each process needs to store and replicate 10 chunks to its partners. However, assuming the first two processes also have 90 unique chunks each, using a naive partner selection strategy that simply chooses ranks $(i+1)..(i+K-1)\ (mod\ N)$ as partners for rank i results in a high imbalance, as shown in Figure 2.

To address this issue, we propose a load-aware partner selection strategy that is centered around the idea of shuffling the ranks in such way that we interleave the ranks that need to send a high number of chunks with the ranks that need to send a smaller number of chunks. Once the ranks are shuffled, applying the naive strategy will result in a much better load balancing, as shown in Figure 2. A key requirement in this context is that all ranks agree on the same shuffling. To this end, we gather from each rank information about the load: how many chunks need to be stored locally and how many chunks need to be sent to each partner. Once each rank is aware of the load of every other rank (e.g., by using an all-gather collective), we calculate an interleaving that is uniquely shared by all ranks and achieves our goal of load-balancing of receive size. We detail this process in Section III-C. Note that more elaborate schemes are possible (e.g. that take into account topology or rack-awareness), however this is outside the scope of this work.

Low-overhead exchanges using single sided communication planning: Once each node has identified its partners, the exchanges between the processes can begin. However, a straightforward solution where each process tells its partners how much data it wants to send and then starts streaming the chunks suffers from significant overhead on the receiving side: the chunks need to be collected from multiple parallel streams and buffered before being written to local storage.

To address this issue, we propose a different high-performance communication model that relies on single-sided operations and thus can take advantage of technologies such as RDMA. The key difficulty in this context is how to expose a designated memory region to each partner in a consistent fashion, such that they can independently send their chunks directly at the right location to avoid extra buffering overhead. This is a non-trivial issue, because standardized APIs for single-sided operations (e.g. MPI) use the concept of *window*, which implies that a single memory region is exposed by one process to all other processes. Thus, it is insufficient for one process to know how much data it needs to send to its partners, because it would not know at what offset in each of the destination window to put it.

However, in our context we can take advantage of the load information that was gathered during the partner selection phase: since there is a unique shuffling, rank i (in the shuffled order) knows how many chunks the other ranks need to send to its partners. Thus, it is possible to calculate an offset for each of the partners of rank i in such way that the other ranks that share the same partners can implicitly agree without extra communication. We detail in Section III-C how this can be efficiently calculated. Furthermore, since each rank knows how many chunks it needs to receive from all other ranks, it can open a window of the right size from the beginning, avoiding any waste. This is an important issue, because applications typically occupy a large part of the memory by the time they call DUMP_OUTPUT.

C. Algorithms

In this section, we show how to materialize the design principles presented in Section III-B through a series of algorithmic descriptions.

Algorithm 1 provides an overview of the process. In a first phase, we compute the hash values corresponding to locally unique chunks into the $LHashes$ set. Starting from $LHashes$, we apply the collective parallel reduction strategy described in Section III-B in order to obtain the most globally frequent fingerprints and their designated ranks. The result is stored in the $GHashes$ set. Note that the reduction can be efficiently parallelized using an optimized ALLREDUCE collective primitive (e.g., as implemented by MPI). The load balancing happens during the merge step (denoted HMERGE), as described in Section III-B.

By convention, $Load[0]$ denotes the number of chunks that need to be stored locally, while $Load[1..(K-1)]$ denotes the number of chunks that need to be sent to partner $1..(K-1)$. In the next step, we compute $Load$. This happens in two steps: first, for each fingerprint h_i that is part of $GHashes$ and for which the current rank M is among the list of designated ranks R_i, it is necessary to compute the number of partners P to which the corresponding chunk needs to be sent to. If $D = |R_i|$ is equal or larger than K, then $P = 0$ (because there are enough replicas already). Otherwise, the number can be calculated by applying the round-robin allocation of the $K - D$ replicas, as mentioned in Section III-B. Once P is calculated, $Load[0..P]$ is incremented. Since M is only one source for the remaining h_i that are not part of $GHashes$, in the second step, $Load[0..(K-1)]$ is incremented for each such h_i.

Algorithm 1 Overview of our approach

```
1: procedure DUMP_OUTPUT(buf, K)
2:     LHashes ← LOCAL_DEDUP(buf, K)
3:     GHashes ← ALLREDUCE(HMERGE, LHashes)
   ▷ M ← my rank
4:     for all h_i ∈ GHashes where M ∈ R_i do
5:         increment Load based on R_i
6:     end for
7:     for all h_i ∈ LHashes and h_i ∉ GHashes do
8:         increment all Load[0..K − 1] ▷ unique hashes
9:     end for
10:    SendLoad ← ALLGATHER(Load)
11:    Shuffle ← RANK_SHUFFE(SendLoad)
12:    Offsets ← CALC_OFF(Shuffle, SendLoad)
13:    for all i is my partner do
14:        put chunks into window of i at Offsets[i]
15:    end for
16:    write both designated and received chunks to local
   storage
17: end procedure
```

Once $Load$ is calculated, information about the load is gathered from all ranks and disseminated to everybody. Thus, each process has its own global view of the send load, which is held by $SendLoad$. Armed with this knowledge, the rank shuffling phase can begin, which is detailed in Algorithm 2: in a first step, we sort the ranks in descending order of the total send size. Then, we repeatedly pair a rank that has the most amount of chunks to send ($head$) with $K-1$ ranks that have the least amount of chunks to send ($tail$) until all ranks were processed. The result of the rank pairing is a new permutation $Shuffle$, which is then used to calculate the $Offsets$ corresponding to the partner windows. This process is detailed in Algorithm 3. Finally, in a last phase, each process opens a window of the appropriate size and puts the chunks that need to be replicated remotely into the partner windows at the appropriate offset. Once the chunk exchange is complete, both the chunks for which the process was designated and the chunks received from its partners are committed to the local storage.

Algorithm 3 zooms on the offset calculation that enables efficient data transfers through single-sided communication. The key idea here is to leverage the global knowledge about the load of each process in order to allocate a static well-defined region for each sender. More specifically, rank i uses offset 0 for its partner $i+1$, offset j for its partner

Algorithm 2 Load aware partner selection based on rank shuffling to balance receive size

```
1: function RANK_SHUFFLE(SendLoad)
2:     RankIndex ← [0..(N − 1)]    ▷ N = no. of ranks
3:     sort RankIndex according to SendLoad
4:     head ← tail ← i ← 0
5:     while i < N do
6:         Shuffle[i++] ← RankIndex[head++]
7:         j ← 1
8:         while j < K and head < tail do
9:             Shuffle[i++] ← RankIndex[tail−−]
10:        end while
11:    end while
12:    return Shuffle
13: end function
```

$i+2$ (where j is the send size from $i+1$ to $i+2$), offset $l+m$ for its partner $i+3$ (where l is the send size from $i+1$ to $i+3$ and m is the send size from $i+2$ to $i+3$), etc.

Algorithm 3 Compute the offsets for partner windows

```
1: function CALC_OFF(Shuffle, SendLoad)
2:     Off[0..(K − 1)] ← 0
3:     for all 1 <= i < K do
4:         P ← Shuffle[(i + 1) mod N]
5:         for all i + 1 <= j < K do
6:             Off[j] ← Off[j] + SendLoad[P][j − i]
7:         end for
8:     end for
9:     return Off
10: end function
```

IV. Implementation details

We implemented a prototype of our approach as a library that exposes the DUMP_OUTPUT primitive to the application. We make use of the Boost C++ collection of libraries for standard data structures and algorithms. For the implementation of the collectives (i.e. ALLREDUCE and ALLGATHER, used in Section III-C), we rely on the Boost API that builds on top of MPI to include support for advanced features, such as automatic serialization of data structures. With respect to the hash function used to calculate the hash values that represent the chunks, we decided to use *SHA1*, a crypto-grade hash function specifically designed to minimize the chance of collisions. However, our approach fully supports other hash functions if a better trade-off between performance and collision chance is desired (e.g. in some scenarios occasional collisions are acceptable).

For the purpose of this work, we integrated our library with *AC-FTE* [7], a run-time environment specifically designed to provide scalable checkpoint-restart functionality for tightly coupled applications, both transparently and at application level. In this context, we use the transparent mode to capture all memory pages that were allocated by the application during its runtime and then pass them to the DUMP_OUTPUT primitive when a checkpoint is desired. The mechanism by which the memory pages are captured is built on top of *jemalloc* [29], a scalable high performance malloc implementation designed to efficiently support concurrent allocations.

Note that since we rely on *AC-FTE* as a demonstrator, we match chunks with memory pages. Thus, the chunk size matches the system memory pages size (i.e. 4 KB by default). How to select an optimal chunk size to strike a good trade-off between deduplication overhead and amount of identified chunk redundancy is an interesting topic in itself but outside the scope of this work. Our library can be easily adapted to work with arbitrarily large chunk sizes or to integrate with other collective I/O standardization efforts, such as MPI-IO [30].

V. Evaluation

A. Experimental Setup

Our experiments were performed on the *Shamrock* testbed of the Exascale Systems group of IBM Research in Dublin. For the purpose of this work, we used a reservation of 34 nodes interconnected with Gigabit Ethernet, each of which features an Intel Xeon X5670 CPU (6 cores, 12 hardware threads), HDD local storage of 1 TB and 128 GB of RAM. With respect to the software configuration, each node runs RedHat 6.3 Linux distribution, while the MPI environment is MPICH2 1.4.1. The deduplication was performed using fixed-sized chunks of 4 KB (corresponding to the size of the memory page), which were hashed using the SHA1 function of OpenSSL 1.0.1. All internal data structures for the collective deduplication and the MPI wrappers are based on Boost 1.53.

B. Methodology

We compare three approaches throughout our evaluation.

a) Full replication: In this setting, each process dumps all chunks of the dataset to local storage and replicates them to its partners before returning control to the application. The partner selection is based on a simple strategy that sends each chunk stored by rank i to ranks $i+1, i+2, ... (i+j)$ *mod* N, where $0 < j < K$, depending on how many duplicates of the chunk were identified. Furthermore, it makes use of the single sided communication planning strategy introduced in Section III-C. For the rest of this paper, we refer to this setting as no−dedup.

b) Replication of locally deduplicated data: In this setting, each process eliminates the duplicate chunks of its dataset before storing them to local storage and replicating them to its partners. Other than this, it is identical to the previous setting: it uses a simple replication strategy and makes use of our single sided communication planning strategy. For the rest of this paper, we refer to this setting as local−dedup.

c) Replication using our collective approach: This setting implements our approach. It performs all stages introduced in Section III-C: in addition to the local deduplication, our collective scheme identifies the duplicate chunks across all processes and replicates only those that have not already reached the desired replication factor. The partner selection strategy performs a rank shuffling to improve the load balancing. and applies the single sided communication planning. For the rest of this paper, we refer to this setting as coll−dedup.

We use two real-life HPC applications to motivate the our approach. The collective I/O scenario we consider is checkpointing at regular intervals as part of a checkpoint-restart fault tolerance strategy. In both cases, the local HDD acts as the local storage and the dataset to be dumped to the local HDD and replicated to the partners represents a full checkpoint of the content of all dynamic memory allocated by the process at the time when the checkpoint was taken. This corresponds to a system-level approach as performed by a transparent checkpointing library. We detail the two applications below.

*1) **HPCCG**:* is a simple conjugate gradient benchmark code for a 3D chimney domain on an arbitrary number of processes that generates a 27-point finite difference matrix with a user-prescribed sub-block size on each process. It is part of the *Mantevo* [31] set of mini-apps, a set of small, self-contained programs that embody essential performance characteristics of key HPC applications. HPCCG was specifically designed to enable weak scalability benchmarking. This makes it an ideal candidate for the purpose of our work, because it is easy to control the amount of data per process. We fix the sub-block size per process to 150x150x150, which corresponds to approx. 1.5 GB per process. The total number of iterations is 127, with a checkpoint scheduled to be taken at iteration 100.

*2) **CM1**:* is a three-dimensional, non-hydrostatic, non-linear, time-dependent numerical model suitable for idealized studies of atmospheric phenomena. This application is used to study small-scale processes that occur in the atmosphere of the Earth, such as hurricanes. It is representative of a distributed stencil computation that studies how a phenomenon evolves in time by splitting a large spatial domain into sub-domains that are assigned to distributed tightly-coupled processes. For the purpose of this work, we have chosen as input data a 3D hurricane that is a version of the Bryan and Rotunno simulations [32]. We use a weak scalability setting for which the size of the subdomain solved by each process remains constant at 200x200. This corresponds to a memory usage of around 800 MB, out of which approx. 500 MB is constantly changed. The duration of the simulation is 70 time-steps, with a checkpoint taken each 30 time-steps.

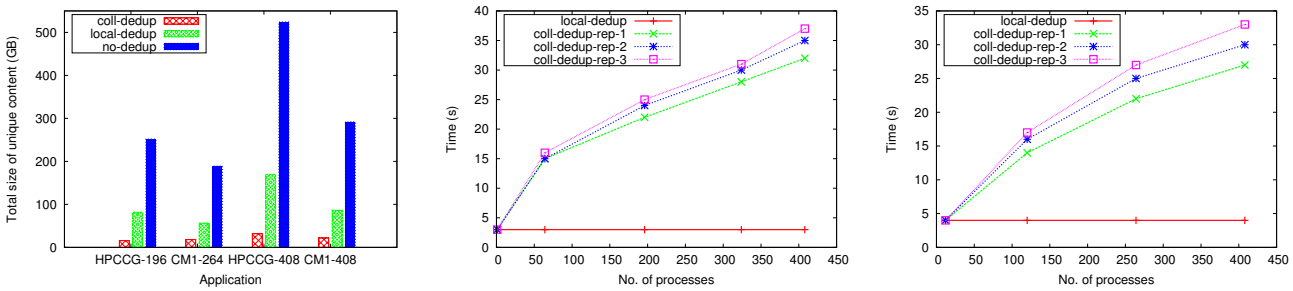
C. Results using a variable number of processes

In this section we evaluate the performance our our approach for a variable number of processes (up to 408, corresponding to our maximum reservation of 34 nodes). We solve a weak scalability scenario that keeps the size of problem for each process constant, as detailed above. We aim to study two aspects: how effective the deduplication is at reducing the total amount of data that needs to be stored, as well as the impact this reduction has on speeding up the DUMP_OUTPUT primitive.

Figure 3(a) shows the results for the total size of unique content identified by each of the three approaches in four configurations: HPCCG-196, CM1-256, HPCCG-408, CM1-408. Since, no−dedup does not attempt to identify any duplication, its total size of unique content coincides with the sum of the data sizes of all processes and will be used as a baseline. Comparatively, local−dedup identifies a large amount of data duplication for both applications: overall, the amount of unique content identified at the extreme of 408 is reduced to 33% for HPCCG and 30% for CM1. However, going even further, coll−dedup manages a reduction down to as little as 6% for HPCCG and 5% for CM1. These results have a significant impact on both the local HDD space needed to save the chunks as well as the network traffic generated by the replication, because both are proportional to the amount of unique content. Thus, under favorable circumstances, we conclude that even low-capacity local storage is feasible to use or perhaps even RAM itself. Furthermore, less network traffic has the benefit of lower energy consumption and frees up bandwidth for other purposes (e.g., application communication).

Since the process of identifying duplicated chunks across processes incurs and an increasing overhead at increasing scale, we illustrate in Figure 3(b) and Figure 3(c) this overhead for an increasing number of processes. The threshold F is fixed at 2^{17}. Since local deduplication is not affected by scale, it is used as a baseline for comparison. What is interesting to note is the relatively small overhead when increasing the number of replicas: even if the list of designated ranks grows for each fingerprint, the difference between the three coll−dedup curves is small. This shows that the parallel reduction can efficiently handle an increasing replication factor. Also interesting to note is the difference between HPCCG and CM1: the relative overheads are smaller for HPCCG compared to CM1. Further analysis revealed that this effect happens because the parallel reduction for HPCCG resulted in more fingerprints with less designated ranks when compared with CM1.

Overall, the overhead of hash calculation and parallel reduction pays off for the lesser amount of stored and replicated chunks, as can be observed from the performance improvements obtained for DUMP_OUTPUT in Table I. We have chosen to focus on the weak scalability of each of the applications for a replication factor of three. The baseline in this context represents the time it takes the application to run to completion without any call to DUMP_OUTPUT. For HPCCG, at the extreme of 408 processes, coll−dedup is 2.8x faster than local−dedup and 9.8x faster than no−dedup. Similarly, for CM1, coll−dedup is 2.5x faster



(a) Total size of unique content for various number of processes. Lower values are better.

(b) HPCCG: Overhead of the collective hash value reduction using a threshold of 2^{17}.

(c) CM1: Overhead of the collective hash value calculation using a threshold of 2^{17}.

Fig. 3. Effectiveness of deduplication: size of unique content and scalability of collective hash value reduction overhead.

than local−dedup and 7.4x faster than no−dedup. Thus, we conclude that our approach exhibits a good weak scalability and significantly improves the performance of DUMP_OUTPUT, in addition to the benefits related to storage space and network traffic reduction.

TABLE I
COMPLETION TIME USING A REPLICATION FACTOR OF 3. BASELINE MEANS NO CHECKPOINTING.

HPCCG				
# of processes	*no-dedup*	*local-dedup*	*coll-dedup*	baseline
1	148s	113s	113s	82s
64	921s	390s	227s	152s
196	1004s	447s	278s	186s
408	1188s	547s	375s	279s
CM1				
# of processes	*no-dedup*	*local-dedup*	*coll-dedup*	baseline
12	1401s	524s	242s	178s
120	1522s	734s	367s	259s
264	1647s	808s	505s	366s
408	1687s	828s	558s	382s

D. Results using a variable replication factor

In this section, we focus on the scalability of our approach for an increasing replication factor. For all our experiments, we fix the number of processes to the maximum of 408 (corresponding to 34 nodes, the reservation size).

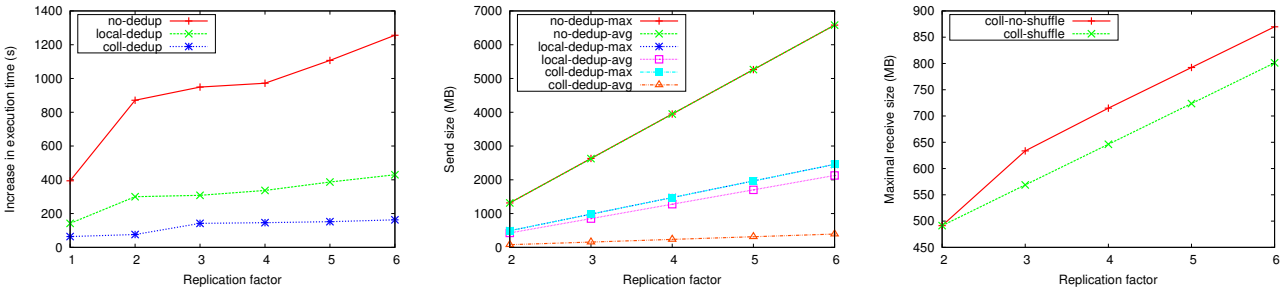
We depict the increase in execution time with respect to the baseline for all replication factors between one and six in Figure 4(a) and Figure 5(a). The baseline in this context means the completion time to run the application without checkpointing (i.e., without using DUMP_OUTPUT). Thus, the increase in execution time is the difference between the completion time of the approaches we compare and the completion time of the baseline.

As expected, the scalability of no−dedup is poor when the replication factor increases: both in the case of HPCCG and CM1, the increase in execution time is 3x and, respectively, 5x higher for a replication factor of six compared with a replication factor of one. On the other hand, coll−dedup exhibits excellent scalability: for both HPCCG and CM1, the performance cost of increasing the replication factor is minimal, leading to a situation where a replication factor of six with coll−dedup is faster than a minimalist replication scenario (i.e., replication factor of two), both compared with local−dedup and no−dedup. Performance-wise, in the case of HPCCG, coll−dedup is 2x faster than local−dedup and 6x faster than no−dedup for a replication factor of six. A similar trend is observable for CM1 as well: for a replication factor of six, coll−dedup is more than 8x faster than no−dedup, as well as 2.3x faster than local−dedup.

Since there is a strong correlation between the performance overhead and the communication overhead, we depict in Figure 4(b) and Figure 5(b) the average and maximal amount of data that each process has to send to its partners. In the case of HPCCG, no−dedup exhibits an overlap between the average and the maximum, which means every process needs to write and replicate the same amount of data. Comparatively, local−dedup exhibits a small gap between maximum and the average, which slowly grows with increasing replication factor. Since coll−dedup starts with a larger gap between average and maximum already at a replication factor of two, the growth of the gap is more pronounced. This insight reveals a potential limitation caused by load imbalance: for example, when using a replication factor of six, coll−dedup sends on the average 5x less data to its partners compared with local−dedup. However, the actual speedup is only 2x. In the case of CM1, we observe a growing gap between the maximum and the average send size for all three approaches. However, in this case, the maximum of coll−dedup is well below the average of local−dedup, enabling a higher speedup than in the case of HPCCG. Based on these observations, we conclude that the load imbalance resulting from collective deduplication can be significant in practice and needs to be addressed efficiently in order to fully exploit the reduction of communication overhead in order to improve the performance.

E. Impact of rank shuffling

In the previous section, we noted the importance of load balancing. While it is impossible to avoid the load imbalance caused by the different amounts of data that is

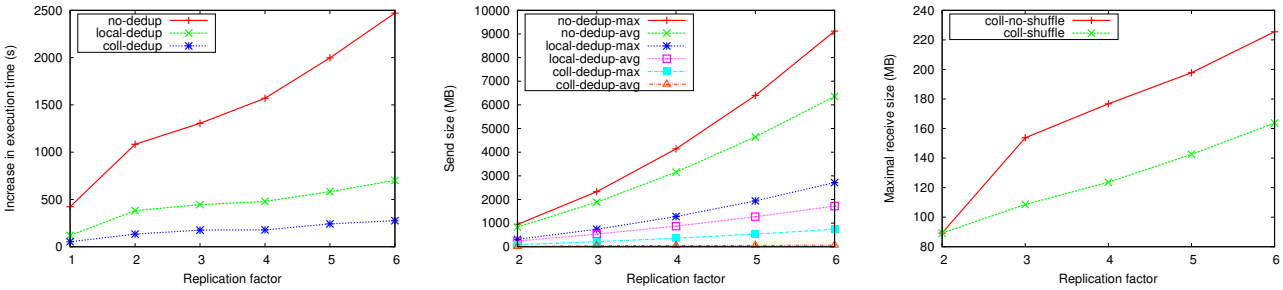


(a) Increase in execution time: lower is better. Baseline (no checkpointing) is 279s.

(b) Amount of replicated data per process: lower is better.

(c) Impact of rank shuffling: lower maximal receive size indicates better load balancing.

Fig. 4. HPCCG: scalability results for increasing number of replicas. Problem size is 408 processes (corresponding to 34 nodes).



(a) Increase in execution time: lower is better. Baseline (no checkpointing) is 382s.

(b) Amount of replicated data per process: lower is better.

(c) Impact of rank shuffling: lower maximal receive size indicates better load balancing.

Fig. 5. CM1: scalability results for increasing number of replicas. Problem size is 408 processes (corresponding to 34 nodes).

unique to each process (and consequently the imbalance in the amount of data that each process has to send to its partners), we argue that it is at least possible to mitigate its negative effects by means of rank shuffling, as detailed Section III-B.

To quantify the benefits of rank shuffling, in this section we analyze the maximal receive size for an increasing replication factor both with and without rank shuffling active. We focus on coll−dedup in both cases. The case when our approach does not use rank shuffling is denoted coll−no−shuffle, while the case it does is denoted coll−shuffle.

The results for both HPCCG and CM1 are depicted in Figure 4(c) and, respectively, Figure 5(c). As can be observed, for a replication factor of two, there is no difference between coll−shuffle and coll−noshuffle. However, with increasing replication factor, the gap between the two approaches becomes clearly visible. In the case of HPCCG, there is an 8% reduction in maximal receive size, while in the case of CM1, the reduction is much larger and almost reaches 30%. In both cases, the difference between the two approaches remains roughly constant. Note that it is not necessary to depict the average receive size, because it matches the average send size discussed in Section V-D.

Thanks to this reduction of maximal receive size through better selection of the replication partners, coll−shuffle manages a more even distribution of the overall communication overhead when compared to coll−no−shuffle. Furthermore, very important to note is also the fact that the receive size directly corresponds to the additional amount of data that each process has to store locally. Thus, the benefits of better load balancing of communication overhead directly translate into a better load balancing of write overhead to local storage devices.

VI. Conclusions

Partner replication to local storage is a crucial technique to enable resilience and high availability at large scale for massive datasets that result from collective I/O writes under concurrency. However, the explosion of data sizes, scale and the need for everincreasing replication factors poses difficult challenges in terms of I/O performance, scalability and resource utilization (bandwidth and storage space). This paper introduced a novel approach that identifies and leverages naturally distributed data redundancy to minimize the amount of data that needs to be replicated, while at the same time eliminates data pieces that are duplicated more than the desired level. To this end, we introduced a low-overhead collective deduplication technique that is enhanced with even rank distribution, load-aware partner selection and single-sided communication planning.

We illustrated the benefits of our approach for two

real-life applications in experiments that involve dozens of nodes and hundreds of cores. Performance-wise, our approach is 2.5x-2.8x faster compared with replication of locally deduplicated data and 7.4x-9.8x faster compared to full replication. Furthermore, as the replication factor increases, our approach can conserve more storage space and bandwidth (up to several orders of magnitude!) when compared to the other two approaches. Also important to note is the impact of the partner selection strategy: compared to a naive solution, our load-aware strategy managed to obtain a much better load balancing, with maximal receive size per process reduced by up to 30%.

Encouraged by these results, we plan to broaden the scope of our work in future efforts. One interesting direction is to combine our approach with other redundancy mechanisms, in particular erasure codes, which would act as a replacement for replication. Furthermore, we proposed a load-aware partner selection strategy that is based on the amount of send data only. What would be interesting to explore in this context are other partner selection criteria, such as rack-awareness or topology.

REFERENCES

[1] J. Dongarra *et al.*, "The international exascale software project roadmap," *Int. J. High Perform. Comput. Appl.*, vol. 25, no. 1, pp. 3–60, Feb. 2011.

[2] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Comput. Surv.*, vol. 34, pp. 375–408, September 2002.

[3] F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *FAST '02: Proceedings of the 1st USENIX Conference on File and Storage Technologies*, Monterey, USA, 2002.

[4] W. M. Jones, J. T. Daly, and N. DeBardeleben, "Application Monitoring and Checkpointing in HPC : Looking Towards Exascale Systems," in *ACM-SE '12: Proceedings of the 50th Annual Southeast Regional Conference*, Tuscaloosa, USA, 2012, pp. 262–267.

[5] D. Meister, J. Kaiser, A. Brinkmann, T. Cortes, M. Kuhn, and J. Kunkel, "A Study on Data Deduplication in HPC Storage Systems," in *SC '12: 25th International Conference for High Performance Computing, Networking, Storage and Analysis*, Salt Lake City, USA, 2012.

[6] B. Nicolae, "Towards Scalable Checkpoint Restart: A Collective Inline Memory Contents Deduplication Proposal," in *IPDPS '13: The 27th IEEE International Parallel and Distributed Processing Symposium*, Boston, USA, 2013, pp. 19–28.

[7] "Ac-fte," http://github.com/bnicolae/ac-fte.

[8] K. Shvachko, H. Huang, S. Radia, and R. Chansler, "The Hadoop distributed file system," in *26th IEEE (MSST2010) Symposium on Massive Storage Systems and Technologies*, May 2010.

[9] B. Nicolae, G. Antoniu, L. Bougé, D. Moise, and A. Carpen-Amarie, "BlobSeer: Next-generation data management for large scale infrastructures," *Journal of Parallel and Distributed Computing*, vol. 71, pp. 169–184, February 2011.

[10] B. Nicolae and F. Cappello, "BlobCR: Virtual disk based checkpoint-restart for HPC applications on IaaS clouds," *Journal of Parallel and Distributed Computing*, vol. 73, no. 5, pp. 698–711, May 2013.

[11] A. Moody, G. Bronevetsky, K. Mohror, and B. R. d. Supinski, "Design, modeling, and evaluation of a scalable multi-level checkpointing system," in *SC '10: Proceedings of the 23rd International Conference for High Performance Computing, Networking, Storage and Analysis*. New Orleans, USA: IEEE Computer Society, 2010, pp. 1–11.

[12] X. Dong, Y. Xie, N. Muralimanohar, and N. P. Jouppi, "Hybrid checkpointing using emerging nonvolatile memories for future exascale systems," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 2, pp. 6:1–6:29, Jun. 2011.

[13] M. Dorier, G. Antoniu, F. Cappello, M. Snir, and L. Orf, "Damaris: How to Efficiently Leverage Multicore Parallelism to Achieve Scalable, Jitter-free I/O," in *CLUSTER '12 - Proceedings of the 2012 IEEE International Conference on Cluster Computing*, Beijing, China, 2012, pp. 155–163.

[14] T. Ilsche, J. Schuchart, J. Cope, D. Kimpe, T. Jones, A. Knüpfer, K. Iskra, R. Ross, W. E. Nagel, and S. Poole, "Optimizing i/o forwarding techniques for extreme-scale event tracing," *Cluster Computing*, vol. 17, no. 1, pp. 1–18, Mar. 2014.

[15] H. Zhu, P. Gu, and J. Wang, "Shifted declustering: A placement-ideal layout scheme for multi-way replication storage architecture," in *ICS '08: Proceedings of the 22nd Annual International Conference on Supercomputing*. Island of Kos, Greece: ACM, 2008, pp. 134–144.

[16] L. Bautista-Gomez, S. Tsuboi, D. Komatitsch, F. Cappello, N. Maruyama, and S. Matsuoka, "FTI: High Performance Fault Tolerance Interface for Hybrid Systems," in *SC '11: Proceedings of 24th International Conference for High Performance Computing, Networking, Storage and Analysis*. Seattle, USA: ACM, 2011, pp. 32:1–32:32.

[17] D. Ibtesham, D. Arnold, K. B. Ferreira, and P. G. Bridges, "On the viability of checkpoint compression for extreme scale fault tolerance," in *Euro-Par Workshops (2)*, Bordeaux, France, 2011, pp. 302–311.

[18] S. Lakshminarasimhan, N. Shah, S. Ethier, S.-H. Ku, C.-S. Chang, S. Klasky, R. Latham, R. B. Ross, and N. F. Samatova, "Isabela for effective in situ compression of scientific data," *Concurrency and Computation: Practice and Experience*, vol. 25, no. 4, pp. 524–540, 2013.

[19] C. Alvarez, "Netapp deduplication for fas and v-series deployment and implementation guide," NetApp, Tech. Rep. TR-3505-0309, 2009.

[20] A. Muthitacharoen, B. Chen, and D. Mazières, "A low-bandwidth network file system," *SIGOPS Oper. Syst. Rev.*, vol. 35, no. 5, pp. 174–187, Oct. 2001.

[21] M. Rabin, "Fingerprinting by random polynomials," Center for Research in Computing Technology, Harvard University, Tech. Rep. TR-CSE-03-01, 1981.

[22] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST'08: Proceedings of the 6th USENIX Conference on File and Storage Technologies*. San Jose, USA: USENIX Association, 2008, pp. 18:1–18:14.

[23] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: a scalable secondary storage," in *FAST '09: Proccedings of the 7th conference on File and storage technologies*. San Francisco, USA: USENIX Association, 2009, pp. 197–210.

[24] I. S. Reed and G. Solomon, "Polynomial codes over certain finite fields," *Journal of the Society for Industrial and Applied Mathematics*, vol. 8, no. 2, pp. 300–304, 1960.

[25] A. Brown and D. A. Patterson, "Towards availability benchmarks: a case study of software raid systems," in *ATEC '00: Proceedings of the USENIX Annual Technical Conference*. San Diego, California: USENIX Association, 2000, pp. 22:1–22:15.

[26] H. Weatherspoon and J. Kubiatowicz, "Erasure coding vs. replication: A quantitative comparison," in *IPTPS '01: Revised Papers from the First International Workshop on Peer-to-Peer Systems*. London, UK: Springer-Verlag, 2002, pp. 328–338.

[27] B. Fan, W. Tantisiriroj, L. Xiao, and G. Gibson, "Diskreduce: Raid for data-intensive scalable computing," in *PDSW '09: Proceedings of the 4th Annual Workshop on Petascale Data Storage*. Portland, USA: ACM, 2009, pp. 6–10.

[28] Z. Zhang, A. Deshpande, X. Ma, E. Thereska, and D. Narayanan, "Does erasure coding have a role to play in my data center?" Microsoft Research, Tech. Rep. MSR-TR-2010-52, 2010.

[29] J. Evans, "A scalable concurrent malloc(3) implementation for freebsd," in *In Proceedings of BSDCan 2006*, Ottawa, Canada, 2006.

[30] M. Snir, S. Otto, S. Huss-Lederman, D. Walker, and J. Dongarra, *MPI-The Complete Reference, Volume 1: The MPI Core*, 2nd ed. Cambridge, MA, USA: MIT Press, 1998.

[31] "The mantevo project," http://mantevo.org/.

[32] G. H. Bryan and R. Rotunno, "The maximum intensity of tropical cyclones in axisymmetric numerical model simulations," *Journal of the American Meteorological Society*, vol. 137, pp. 1770–1789, 2009.