Reliable MPI-IO through Layout-Aware Replication

Seung Woo Son Samuel Lang Robert Latham Robert Ross Rajeev Thakur
Mathematics and Computer Science Division
Argonne National Laboratory
{sson,slang,robl,rross,thakur}@mcs.anl.gov

Abstract

The current deployment of petascale systems and the promise of future exascale systems have created unprecedented challenges in how to manage failures in such systems. While many parallel file systems provide some sort of redundancy mechanism to cope with failures, such systems rely heavily on a hardware-based solution such as RAID. In this paper, we propose a block replication approach to store data redundantly. The approach does not depend on file system fault-tolerance mechanisms. Rather, the approach replicates each file block transparently within MPI-IO, using replication-aware datatypes. File striping information is used to place blocks from each replica in a separate storage node. We have implemented this replication mechanism in the MPI-IO layer. Our experimental results using a microbenchmark and real MPI-IO applications with PVFS and Lustre demonstrate that block replication in MPI-IO can be achieved transparently.

1. Introduction

With current petascale machines in operation and the promise of exascale machines by the 2018 time frame, there is a pressing concern that component failures in such systems will be unavoidable and that such systems will need to operate in a faulty environment [5]. Furthermore, if the data projected in [24, 25] is correct, the reliability of storage systems will also be a big concern for building such extreme-scale machines. For instance, in storage systems in which no disk is shared, the loss of a storage node implies the loss of all the data stored on that node. Therefore, researchers have proposed storing redundant data across storage nodes, thus tolerating the failures in storage nodes [12, 24, 25].

In particular, efforts have focused on providing redundancy from the storage node or device angle by providing a certain type of RAID [22] underneath the parallel file system. These techniques typically work at a very fine granularity and suffer from high cost to purchase hardware disk arrays. Applying this RAID type of techniques in a parallel file systems also imposes excessively high overhead on a cluster because in such a system, the disks are distributed and network latencies are higher. For example, for reconstructing the missing data block during data recovery, the system may need to access two or more of the remaining data blocks, depending on the redundancy level, from the nodes on which it has been striped. Distributed file systems for data-intensive computing domains like MapReduce and Hadoop, on the other hand, use block replication to provide data reliability [8, 14, 18]. Each block in HDFS, for instance, is replicated to three nodes, one in a local node and two in remote nodes to tolerate two node failures. While such efforts have proved effective in the context of Hadoop/MapReduce workloads, one can conceivably achieve the same level of redundancy in the context of MPI-IO applications without relying on data redundancy mechanisms provided by the file and storage systems.

In this paper we propose and evaluate a reliable yet transparent mechanism for block replication in the MPI-IO layer. To the best of our knowledge, this is the first time such a mechanism has been investigated. We use file layout information from the underlying file system, easily extractable through the MPI hint mechanism, in order to decide how to lay out the original file blocks in a fault domain aware manner. Using the extracted file layout, we define a set of new file layouts using MPI derived datatypes, each of which can be used for writing the original data to a particular storage node. An additional hint is used for passing a user-defined replication factor. In other words, the number of replicas in our approach is a configurable option that the user can specify, say 3 as an example. Therefore, the user can choose a trade-off between cost (time and space) and reliability. We made all these block replications transparent to applications by using the profiling interface to MPI (PMPI).

Our experimental evaluations demonstrate that our scheme transparently achieves block replication within the context of MPI-IO applications, incurring reasonable overhead as compared with cases without replication. Our write-oriented benchmark results showed that our replication scheme incurs the same amount of replication cost, by a replication factor, as empirically observed in many prior studies. Since our replication scheme is built on top of MPI-IO, it can work with potentially any parallel file system that supports user-tunable file layout information. Our experiments with two commonly-used parallel file systems, PVFS and Lustre, clearly exemplified the usefulness of such feature. We also demonstrate a mechanism that allows users to determine the degree of replication by means of a user hint passed to our shim layer. Therefore, users can achieve different levels of fault tolerance depending on the estimated system health and storage capacity overhead.

The rest of the paper is organized as follows. Section 2 describes the background for handling failures through replication. Our replication technique in MPI-IO environments is described in Section 3. Section 4 presents an experimental evaluation of our proposed scheme. The limitations of our current implementation and possible future extensions to our approach are described in Section 5. Section 6 briefly discusses related work on fault-tolerant I/O. Section 7 concludes the paper with a summary of our key contributions.

2. Background

Failures are common in large-scale clusters; and those systems must therefore be able to detect, tolerate, and recover from such failures easily [24]. Data replication is one mechanism to ensure consistency between redundant resources by using multiple storage nodes or other devices to improve the reliability and availability of storage systems. Other techniques, such as redundant block striping or erasure coding, are used in parallel file systems such as GPFS, PVFS, Lustre, and PanFS [13, 21, 23, 31]. Although these approaches are more space efficient, they incur a performance penalty during data recovery. For example, with striping, the system may need to read two or more data blocks in order to reconstruct the missing block. Replication, on the other hand, always requires only one copy.

Distributed file systems for data-intensive application domains, such as HDFS [8] and GFS [16], provide high reliability and availability through replication. For example, HDFS uses replication to maintain at least three copies, one primary and two replicas, of every data chunk. Applications can specify a higher (or lower) replication factor during file creation time. All copies of a chunk are stored in different data nodes using a rack-aware replica placement policy to improve reliability as well as network bandwidth utilization. Placing the first copy in the local storage node and the second copy in a different node on the same rack is intended to improve network bandwidth utilization because local writes and intrarack communication are faster than interrack communication. In order to tolerate a rack failure, the third copy is stored on a data node in a different rack. In HDFS, a data node that receives a data chunk from the client will send the chunk to two other data nodes that store the second and third replicas.

Most parallel file systems, including PVFS and Lustre, rely on a hardware-based reliability solution such as RAID to get a larger number of drives and redundancy. In this paper, we provide an HDFS-style replication in MPI-IO environments by forcing the client application to write each replica to three different storage nodes through replication-aware file layouts.

3. Our Approach

Our goal is to replicate file blocks in the MPI-IO layer, thus providing data redundancy for fault tolerance without relying on those mechanisms provided in parallel file systems and enterprise storage systems. Conceptually, one can view our approach as an application-oriented replication mechanism, because the data replication traffic is all initiated by applications.

Our overall approach is depicted in Figure 1. We start by dividing the original file region into stripe boundaries. In this figure, the original file block has four blocks (0, 1, 2, and 3), each with the same stripe size. Obtaining striping information is important because it ensures that the unit of block is placed in a separate node, allowing redundancy across storage nodes. We then create three file layouts that have different block displacements to enforce placing each block in a different storage node. These layouts are used later for setting a fileview for each write. The displacement—that is, the number of bytes from the beginning of the structure at which the given data item appears in it—in each layout is determined by the replication factor. In our approach, we use a replication factor of 3, as used in HDFS [8]. This default replication factor is supposed to tolerate two node failures, since it stores three replicas. Using different displacement for each file block, one can potentially store each replica in either a single

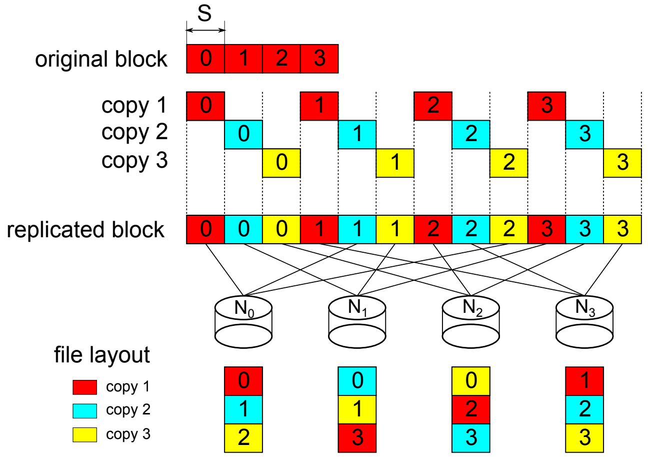

Figure 1. Overview of our block replication scheme in a single file when the replication factor is 3. The original file has 4 blocks (from 0 to 3), and each block is exactly same the stripe size (S).

file or a separate file. In our initial implementation, we use a single round-robin file for storing three copies. The implication of using three files in a round-robin layout is discussed in Section 5.

We now present the details of our replication scheme. To replicate the file across file stripe boundaries, we need detailed knowledge of the data layout for a given file. In parallel file systems, this layout information can be specified by using the following 3-tuple:

(`start_node`, `striping_unit`, `striping_factor`),

where `start_node` is the id of the first node where the file gets striped, `striping_unit` gives the stripe size, and `striping_factor` gives the number of nodes (disks) used for striping. As an example, in Figure 1, the original file is striped over all four nodes (N_o to N_3). Assuming that the stripe size is S and the total file size is $4S$ (for illustrative purposes), the data layout of this dataset can be expressed as (0, S, 4).

We note that many parallel file systems and I/O libraries for high-performance computing support calls that convey to them the disk layout information when the file is created through the MPI hint mechanism. For instance, one can get or set the striping parameter by setting corresponding hints in the `MPI_Info` structure. Then, the striping information is passed to the `MPI_File_open` calls parameter. We also note that, while these three hints are used in ROMIO, an implementation of MPI-IO, only `striping_unit` and `striping_factor` are defined as MPI-2 predefined hints. The `start_node` hint is platform-specific.

Given that the MPI program will be extracting the file layout information, we next define file layouts for writing the three replicas. In MPI-IO applications, a file is an ordered collection of typed data items. The blocks of a file are replicated for fault tolerance. The block size and replication factor are configurable per file. An application can specify the number of replicas of a file. The replication factor can be specified at file creation time and remains the same value throughout the execution time. With the file striping information defined above, the type map for the original block, D_{orig} can be described as follows:

$$D_{orig} = \{(s_0, d_0), (s_1, d_1), ..., (s_{n-1}, d_{n-1})\},$$

where s_i and d_i are the stripe size and displacement for the ith stripe, respectively, and n is the number of file blocks in terms of stripe size. For example, the type map for the original block depicted in Figure 1 is described as $\{(S, 0), (S, S), (S, 2S), (S, 3S)\}$.

Assuming the replication factor is R, our scheme next creates R number of datatypes, each representing an individual replica depicted in Figure 1:

$$\begin{aligned}
D_0 &= \{(s_0, d_0), (s_1, d_1 + S * R), ..., \\
&\quad (s_{n-1}, d_{n-1} + S * (n-1) * R)\}, \\
D_1 &= \{(s_0, d_0 + S), (s_1, d_1 + S + S * R), ..., \\
&\quad (s_{n-1}, d_{n-1} + S + S * (n-1) * R)\}, \\
&\quad ... \\
D_{R-1} &= \{(s_0, d_0 + S * (R-1)), ..., \\
&\quad (s_1, d_1 + S * (R-1) + S * R), \\
&\quad (s_{n-1}, d_{n-1} + S * (R-1) + S * (n-1) * R)\},
\end{aligned}$$

where S is the stripe size and n is the number of data blocks in terms of S. We use the `MPI_Type_create_hindexed` to specify the datatypes of each replica. We note that one can achieve the same datatypes for each replica using `MPI_Type_indexed`. The function `MPI_Type_indexed` is identical to `MPI_Type_create_hindexed` except that block displacements in array_of_displacements are specified in multiples of the oldtype extent, rather than in bytes [20].

It should be mentioned that our replication scheme currently supports independent MPI I/O calls only. Collective I/O calls typically comes with `MPI_File_set_view` calls. Therefore, there needs to be an additional step in order to make our replication scheme work with collective I/O. We discuss this issue in Section 5. We also want to mention that our replication scheme does not create a correctness issue when each process is writing with different offset values. This is because the location is a function of the stripe size as long as the correct fileview is used for each replica write.

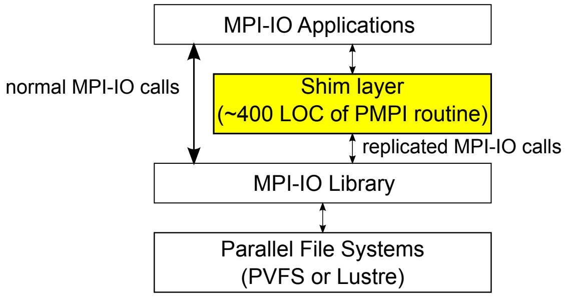

Figure 2. Overview showing how our block replication scheme implemented applications transparently. Note that no modification is required for any other software stacks including MPI-IO library and parallel file systems underneath.

```
char *fname = "test";
int *buf, N;
MPI_Info_create(&info);
MPI_Info_set(info, "replication_factor", "3");
MPI_File_open(..., fname, ..., info, &fh);

buf = (int *) malloc(SIZE);
N = SIZE/sizeof(int);

MPI_File_write(fh, buf, N, MPI_INT, &status);
```

(a) An example code

```
static int RF; /* replication factor */
static MPI_Datatype cur_dtype; /* current datatype */
static int initialized = 0;
static MPI_Datatype hindextype[RF];

init(MPI_Datatype dtype)
{
   int *b[3];      /* block length */
   MPI_Aint *d[3]; /* displacement */
   MPI_Info info;

   /* get replication factor passed from users */
   MPI_Info_get(info, "replication_factor", &RF, ...);

   /* define derived datatypes for specifying replicas */
   for(i=0; i < ncount; i++)
   {
      for(j=0; j < RF; j++)
      {
         b[j][i] = S;
         d[j][i] = S*j + S*i*RF;
      }
   }
   for(i=0; i < RF; i++)
   {
      MPI_Type_create_hindexed(ncount, b[i], d[i],
                       dtype, &hindextype[i]);
      MPI_type_commit(&hindextype[i]);
   }
   initialized = 1; /* set the init flag to 1 */
   cur_dtype = dtype;
}

MPI_File_write(MPI_File fh, void *buf, int count,
               MPI_Datatype dtype, ...)
{
   if(initialized == 0 || dtype != cur_dtype)
      init(dtype);

   for(i=0; i < RF; i++)
   {
     /* set fileview that corresponds to each replica */
     MPI_File_set_view(fh_shadow, 0, dtype, hindextype[i], ...);
     PMPI_File_write(fh_shadow, buf, count, dtype, &status);
   }
}
```

(b) PMPI routine

Figure 3. (a) Example code illustrating a typical MPI-IO example. (b) Code after applying our MPI-IO replication scheme when the replication factor (RF) is set to 3.

To support our replication scheme transparently, we need a shim layer that converts the original file write requests to those for replication. In this paper, we achieve this transparent replication using the profiling interface to MPI (PMPI). We wrote about 400 lines of PMPI module in C for our current implementation. Figure 2 shows how our replication scheme is deployed in the shim layer to interact with MPI-IO applications and underlying file system components. The shim layer intercepts the MPI-IO routines defined in the PMPI module and performs triplication. In other words, if the MPI client calls a normal file write call, the shim layer builds three file layouts and sends three write requests to the storage nodes using a corresponding fileview.

To illustrate how our replication scheme in MPI-IO works in practice, let us consider an example code in Figure 3(a) that uses the normal `MPI_File_write` call. This example code simply opens a file and writes N integers in it. The replication factor is defined as 3 by using the hint mechanism and passes the `MPI_File_open` call (file creation) as an argument. Assuming the S is 1,048,576 bytes and at least three file servers are available, Figure 3(b) gives the shim code for achieving our replication scheme. The data layouts for replication are created once in the init() function.

We note that we used `MPI_File_set_view`, a collective call, in order to set the fileview for each replica being written by the independent write call. The only way to safely call a collective function within an independent function is to make sure the collective function is collective over `MPI_COMM_SELF`. For this purpose, we maintain in our shim layer a shadow file handle (`fh_shadow`) opened with `MPI_COMM_SELF` and use it internally. This could potentially lead to I/O consistency issues, namely, the use of `MPI_COMM_SELF` may result in a weaker MPI-IO consistency model than the user expects having opened the file with a different communicator such as `MPI_COMM_WORLD`. To avoid this problem, our shim layer also needs to internally close and reopen the file when the user calls `MPI_File_sync`.

```
MPI_File_read(...)
{
  /* try the 1st copy */
  MPI_File_set_view(fh_shadow, 0, ..., hindextype[0], ...);
  PMPI_File_read(fh_shadow, buf, nints, MPI_INT, &status);

  if (status == ERROR)
  {
    /* try the 2nd copy */
    MPI_File_set_view(fh_shadow, 0, ..., hindextype[1], ...);
    PMPI_File_read(fh_shadow, buf, nints, MPI_INT, &status);
    if (status == ERROR) {
      /* try the 3rd (last) copy */
      MPI_File_set_view(fh_shadow, 0, ..., hindextype[2], ...);
      PMPI_File_read(fh_shadow, buf, nints, MPI_INT, &status);
    }
  }
}
```

Figure 4. Shim code illustrating how `MPI_File_read` is performed in our block replicated files. Reading appropriate file blocks is handled in a completely user-transparent manner.

Since our block replication scheme writes three replicated blocks in a single file, we also need to use the proper layout during read operations. Figure 4 shows our shim code that describes our read mechanism. In our implementation, the read request is always directed to the first replica, meaning that the `MPI_File_read` is performed using the first data layout (D_0). If the initial read attempt is not successful because of failure in any nodes that store the blocks in the first data layout, our next attempt goes to the blocks in the second layout. If more than three node failures occur, the read operation eventually fails. We note that failure in a read operation can be detected only if there is an error in each `PMPI_File_read` call, typically incurring tens of seconds of timeout value in current parallel file systems.

4. Experimental Evaluation

This section describes the experimental platform, schemes, and benchmarks we used in our evaluation. Also presented are the experimental results using a microbenchmark and real MPI-IO benchmarks with two commonly used parallel file systems, Lustre and PVFS.

4.1. Experimental Setup

To demonstrate the applicability of our replication scheme, we conducted a set of experiments on a cluster of 24 nodes, each of which is equipped with dual Intel Xeon Quad Core running at 2.66 GHz, 16 GB of main memory, and 50 GB of local disk storage space. All nodes ran the Linux 2.6.22 kernel. We used MPICH2-1.3.1 [4]. To demonstrate the applicability to different file systems, we evaluated our scheme with two parallel file systems, PVFS-2.8.1 [13] and Lustre 1.6.4.2 [21]. Both file systems were configured to use four storage nodes, and the default stripe unit was set to 1 MB (1,048,576 bytes). All other parameters were set to the default values without further modification. In other words, both file systems included no redundancy mechanisms by configuration.

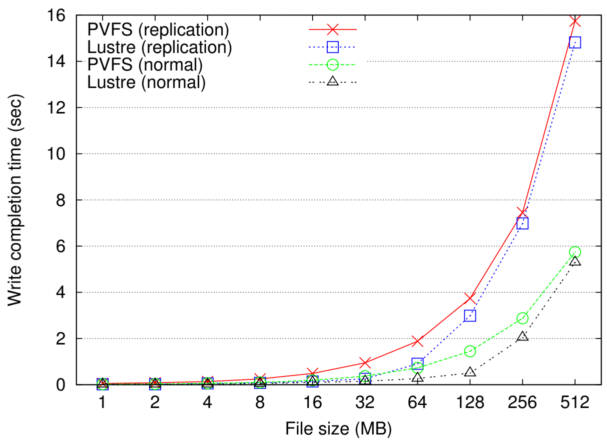

Figure 5. Replication overhead in terms of write completion time while varying the file sizes when executed in conjunction with PVFS and Lustre, respectively. We used 4 file servers and 1 MPI client.

We evaluated the proposed scheme using two schemes. The first scheme, denoted "normal," runs the normal MPI-IO applications that do *not* employ any replication underneath. We used either PVFS or Lustre as an underlying parallel file system for our evaluation. The second scheme, denoted "replication," is an implementation of our scheme, described in Section 3, that triplicates data blocks in a single file. Since the replication is performed in the shim layer we provide, neither the applications nor the parallel file systems required any modification. We note that, while we used the replication factor of 3 in our experiments, different replication factors can be applied by using the MPI hint mechanism as described in Section 3. All our experiments were run three times to get average results.

4.2. Microbenchmark Results

Figure 5 shows the overall replication overheads incurred by our scheme as compared with the normal case in terms of write completion. In this experiment,

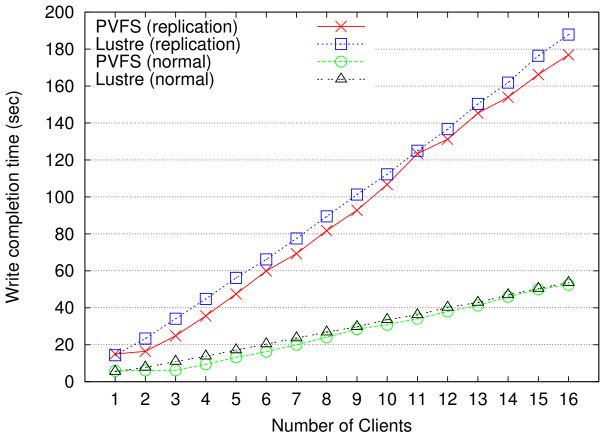

Figure 6. Write completion time as the number of clients is increased from 1 to 16. Each MPI client writes its own file of 512 MB. The number of servers is 4.

we ran a simple MPI program that opens a file and writes to it. The graphs in Figure 5 are generated by using one MPI client and four file servers while varying the file sizes (1 MB to 512 MB). As we can see in the figure, in both schemes the execution time increases as the file size increases. Also, as expected, the replication incurs about 3x slower completion time for both PVFS and Lustre. The reason is that the replicated scheme makes three write operations per write.

Figure 6 presents the write completion time with and without replication as the number of clients is changed from 1 to 16. In this experiment, each MPI client writes a unique file of 512 MB. Again, we present the results with both PVFS and Lustre. We observe similar trends in these graphs; both PVFS and Lustre show almost identical performance behaviors with and without replication. Since this experiment also involves pure write-only workloads, the replication scheme incurs about 3x performance slowdown as compared with the normal scheme.

We note that prior studies, those from data-intensive distributed file systems, also experimentally showed that the replication overhead is increased by a replication factor [3, 19, 28]. For example, Ko et al. [19] showed that increasing the Reduce replication factor to 2 doubled the job completion time. Therefore, we conclude that our scheme does *not* incur any additional overhead that would otherwise be needed to provide redundancy, while being able to work with several of parallel file systems as evidenced by our evaluation with both PVFS and Lustre. We also want to emphasize that this transparent replication within MPI-IO layer comes with user-tunable replication granularity (via changing stripe unit during file creation) and replication factor, which is not feasible for hardware-based RAID schemes.

Table 1. Characteristics of application benchmarks with the normal scheme.

Benchmark	No. of Procs.	Execution Time	Dataset Size
BTIO	16	63.9 sec	419.43 MB
MADbench2	16	21.9 sec	4,848 MB

4.3. Application Benchmarks Results

Our evaluation so far has made a case for our block replication scheme in MPI-IO and showed that providing redundancy through triplication is feasible without relying on mechanisms in the underlying parallel file system. One might wonder, however, what the impact would be on the performance of a real application. To quantify this, we measured the performance of two real I/O applications:

- **BTIO**: BTIO is an I/O version of the BT (block tridiagonal) benchmark found in the NAS Parallel Benchmarks (NPB) suite [32]. The BTIO benchmark performs MPI-IO writes and reads of a nested strided datatype. The BTIO benchmark was built from NPB version 3.3 with the “simple” subtype and the Class A problem size. Class A corresponds to a grid size of 64 x 64 x 64, which writes in aggregate 419.43 MB of data to a shared file every fifth timestep out of 200 total iterations.

- **MADbench2**: MADbench2 is a benchmark derived from the MADspec data analysis code [7]. As part of its calculations, the MADspec code (and likewise the MADbench2 benchmark) performs extremely large out-of-core matrix operations, requiring successive writes and reads of large, contiguous data from either shared or individual files. We built the MADbench2 to generate unique (individual) filetype using MPI-IO. In this I/O setup, each process writes its unique file, which corresponds to about 303 MB.

The characteristics of these two real I/O benchmarks are given in Table 1. Both applications are executed by using 16 MPI processes with PVFS as the underlying file system. Since the behavior of PVFS and Lustre is almost identical, as shown in our microbenchmark results, we expect similar results when Lustre is used.

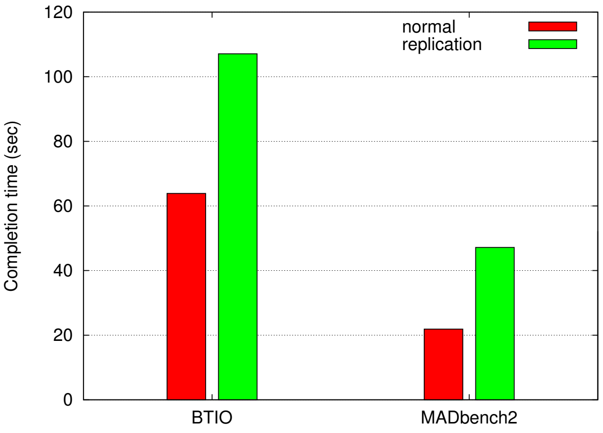

Figure 7. Performance of two application benchmarks for normal and replication schemes when PVFS is used.

Figure 7 gives the performance of two real benchmarks with and without our replication scheme. As we can see from the figure, the replication scheme is slower than the normal scheme, incurring 67.7% and 115.6% for BTIO and MADBench2, respectively. The reason is mainly that each I/O application exhibits different I/O access patterns in terms of read/write ratios. In other words, the more write intensive an application, the higher the overhead incurred by the replication.

5. Discussion

Our discussion so far has presented one way of providing reliable MPI-IO through replication, but there are some caveats as well. This section describes several of these that need to be handled.

Dealing with collective I/O: The current implementation of our replication scheme works only with independent calls such as `MPI_File_write` and `MPI_File_write_at`. Many MPI-IO applications, however, use collective I/O because many of them can benefit from data sieving and two-phase I/O. Our replication scheme uses replication-aware file layouts for writing each replica. Using derived datatypes to specify the different block locations as a fileview argument, however, conflicts with using the fileview for a particular process that is also described in a file datatype and displacement. Therefore, in order to make our replication scheme work with applications that use collective calls, our shim layer needs to generate a new derived datatype that combines the user-specified datatypes and our derived datatype for block replication. Special care also needs to be made when two-phase I/O, an optimization in collective I/O, is turned on. This is because, in two-phase I/O, each process first communicates with each other to determine what region to write or read from the aggregate access region, thereby making combining user-specified datatypes with our datatypes for replication more complex.

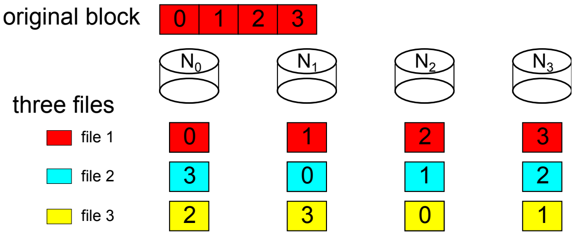

Figure 8. File layout using three files with different first data servers.

Storage overhead: Our scheme achieves reliability by replicating each file stripe across different file servers. Therefore, it does not require RAID storage on the file servers. However, the storage overhead incurred by replication increases with higher replication factor and is much higher than RAID. For instance, while conventional 8+2 RAID 6 encoding incurs 25% storage capacity overhead, our triplication scheme incurs 200% overhead. Again, our focus in this paper is not on comparing replication against erasure code. However, we believe that selective replication [29] or asynchronous and delayed encoding [27] can be used in conjunction with our approach if the storage capacity is under pressure.

Placing replicas in separate files: Our current implementation stores replicated blocks in a single file in which three replicas of the same stripe size each are placed successively using different displacement values. One can, however, achieve the same goal using multiple files with different `start_node`. This scheme, which uses the same round-robin data layout in three separate files, is depicted in Figure 8. Like other replication schemes, we need to have at least three storage nodes in order to use this scheme. While the resulting layout from this scheme looks similar to our single-file approach, it differs in at least two ways. First, since each replica is stored in a separate file for read/write calls, it does not require complex data layouts. Second, it has the same storage capacity as the single-file case, but it also creates 3x number of files. This could be an issue in metadata performance when applications use per process write policy.

Nonblocking writes for replicas: In our implementation of writing replicas, we used the blocking MPI-IO, that is, `MPI_File_write`, mainly because the underlying ROMIO implementation does not support nonblocking I/O for strided writes. Conceptually, one can enhance the replication performance by hiding I/O latency for writing replicas through nonblocking I/O. This can be achieved by issuing three `MPI_File_iwrite` calls for each replica and waiting for each MPI request (via `MPI_wait`) for each write to complete. This approach requires a modification to the ROMIO implementation, but we note that its performance gains are tightly coupled with the characteristics of the interconnection network used in storage nodes and the local disk used in each storage node.

6. Related Work

Many techniques and algorithms have been proposed for providing fault tolerance in file and storage systems based on replication or erasure codes. In this paper, however, we restrict our discussion to techniques in the context of MPI and MPI-IO.

Several MPI extensions for fault tolerance exist, such as FT-MPI [15] and VolpexMPI [2]; most are focused on providing capability of checkpointing/restart in MPI processes [9, 30]. Wang et al. [30] proposed two XOR-based double-erasure codes for in-memory checkpointing for MPI programs. Brightwell et al. [9] proposed and implemented two transparent redundancy approaches to MPI applications, thus tolerating loss of application processes and connectivity. Anand et al. [2] proposed an MPI library, called VolpexMPI, that deployed redundant MPI processes for dealing with unreliable execution environment. All these approaches have concentrated on providing reliable execution of MPI processes rather than on providing redundancy to data stored on the storage systems.

Recent work has focused on providing redundancy in MPI-IO and parallel file systems [17, 11, 1, 6, 2, 8, 26, 23]. Calderón [11] proposed a technique to provide fault tolerance in MPI applications on PVFS file systems. Brinkmann et al. [10] presented data structures that allow a continuous snapshot implementation in a clustered storage environment. Amer et al. [1] studied the reliability of storage from the perspective of using parity in redundant array layouts. Their proposed layout, called SSPiRAL, offers lower MTTDLs than complete 3 out of 6 erasure code, relying on simple pairwise parity computations. Birk and Kol [6] also studied providing efficient reconstruction of data using erasure correcting code in a peer-to-peer system. Targeting data-intensive computing workloads, HDFS is designed to store large files reliably in a large cluster [8]. HDFS uses triplication for fault tolerance; blocks of a file are replicated by using a rack-aware replica placement scheme. Like HDFS, our approach is also based on replication, but it provides replication within the context of MPI-IO.

The work most relevant to our study was conducted by Gropp et al. [17], who used erasure code to provide a lazy redundancy mechanism in MPI environments. Our approach is similar to the lazy redundancy technique in that we also use MPI datatypes to provide fault tolerance, but our technique is based on block replication. Unlike the lazy redundancy technique, which requires a modification to the ROMIO MPI-IO implementation, our approach does not require any modification to the MPI-IO library.

7. Conclusion and Future Work

In this paper, we have presented a block replication scheme in the context of MPI-IO by using replication-aware MPI datatypes. The proposed scheme uses MPI derived datatypes to represent the replicated file layout for providing reliability to MPI applications. Our experimental results demonstrate that our scheme provides transparent block replication to MPI-based applications; as far as we know, this is the first approach to provide this capability. We also show that our scheme can be used in conjunction with any existing parallel file system that allows one to extract and change the underlying file striping information.

We plan to extend the ideas presented in the paper in several ways. We want to explore a way to have the data layout provide not only reliability but also better read performance by using a different data layout for each replica. We also plan to quantify the impact of erasure code, such as the lazy redundancy scheme [17], and our replication scheme on various I/O access patterns.

8. Acknowledgment

We thank anonymous reviewers and Randal Burns for suggestions that helped improved the paper. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357.

References

[1] A. Amer, J.-F. Paris, T. Schwarz, V. Ciotola, and J. Larkby-Lahet. Outshining Mirrors: MTTDL of Fixed-Order Spiral Layouts. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, pages 11–16, 2007.

[2] R. Anand, E. Gabriel, and J. Subhlok. Communication Target Selection for Replicated MPI Processes. In *Proceedings of the EuroMPI*, pages 198–207, 2010.

[3] R. Ananthanarayanan, K. Gupta, P. Pandey, H. Pucha, P. Sarkar, M. Shah, and R. Tewari. Cloud analytics: Do We Really Need to Reinvent the Storage Stack? In *Proceedings of the Workshop on Hot Topics in Cloud Computing*, pages 15–15, 2009.

[4] Argonne National Laboratory. MPICH2 User's Guide, 2009.

[5] K. Bergman et al. ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems, 2008.

[6] Y. Birk and T. Kol. Coding and Scheduling Considerations for Peer-to-Peer Storage Backup Systems. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, pages 25–32, 2007.

[7] J. Borrill, J. Carter, L. Oliker, and D. Skinner. Integrated Performance Monitoring of a Cosmology Application on Leading HEC Platforms. In *Proceedings of the International Conference on Parallel Processing*, pages 119–128, 2005.

[8] D. Borthakur. HDFS Architecture. `http://hadoop.apache.org/common/docs/current/hdfs_design.html`.

[9] R. Brightwell, K. Ferreira, and R. Riesen. Transparent Redundant Computing with MPI. In *Proceedings of the EuroMPI*, pages 208–218, 2010.

[10] A. Brinkmann and S. Effert. Snapshots and Continuous Data Replication in Cluster Storage Environments. In *Proceedings of the International Workshop on Storage Network Architecture and Parallel I/Os*, pages 3–10, 2007.

[11] A. Calderón, F. G. Carballeira, F. Isaila, R. Keller, and A. Schulz. Fault Tolerant File Models for MPI-IO Parallel File Systems. In *Proceedings of the European PVM/MPI*, pages 153–160, 2007.

[12] F. Cappello. Fault Tolerance in Petascale/ Exascale Systems: Current Knowledge, Challenges and Research Opportunities. *Int. J. High Perform. Comput. Appl.*, 23:212–226, August 2009.

[13] P. H. Carns, W. B. Ligon III, R. B. Ross, and R. Thakur. PVFS: A Parallel File System for Linux Clusters. In *Proceedings of the Annual Linux Showcase and Conference*, pages 317–327, 2000.

[14] J. Dean and S. Ghemawat. MapReduce: Simplified Data Processing on Large Clusters. *Commun. ACM*, 51:107–113, January 2008.

[15] D. Dewolfs, J. Broeckhove, V. S. Sunderam, and G. E. Fagg. FT-MPI, Fault-Tolerant Metacomputing and Generic Name Services: A Case Study. In *Proceedings of the European PVM/MPI*, pages 133–140, 2006.

[16] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google File System. In *Proceedings of the ACM Symposium on Operating Systems Principles*, pages 29–43, 2003.

[17] W. Gropp, R. Ross, and N. Miller. Providing Efficient I/O Redundancy in MPI Environments. In *Proceedings of the European PVM/MPI*, pages 77–86, 2004.

[18] Hadoop. `http://hadoop.apache.org/`.

[19] S. Y. Ko, I. Hoque, B. Cho, and I. Gupta. Making Cloud Intermediate Data Fault-Tolerant. In *Proceedings of the 1st ACM symposium on Cloud computing*, pages 181–192, 2010.

[20] Message Passing Interface Forum. *MPI: A Message Passing Interface Standard, Version 2.2.* 2009.

[21] Oracle Corporation. Lustre File System. `http://www.oracle.com/us/products/servers-storage/storage/storage-software/031855.htm`.

[22] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 109–116, 1988.

[23] F. Schmuck and R. Haskin. GPFS: A Shared-Disk File System for Large Computing Clusters. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 231–244, 2002.

[24] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 1–16, 2007.

[25] B. Schroeder and G. A. Gibson. Understanding Failures in Petascale Computers. *Journal of Physics: Conference Series*, 78(1):012022, 2007.

[26] B. W. Settlemyer and W. B. Ligon III. A Technique for Lock-Less Mirroring in Parallel File Systems. In *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, pages 801–806, 2008.

[27] B. Tan, W. Tantisiriroj, L. Xiao, and G. Gibson. DiskReduce: RAID for Data-Intensive Scalable Computing. In *Proceedings of the Petascale Data Storage Workshop*, 2009.

[28] W. Tantisiriroj, S. Patil, and G. Gibson. Data-intensive file systems for Internet services: A rose by any other name ... Technical Report CMU-PDL-08-114, Parallel Data Laboratory, Carnegie Mellon University, October 2008.

[29] C. Wang, Z. Zhang, X. Ma, S. S. Vazhkudai, and F. Mueller. Improving the Availability of Supercomputer Job Input Data using Temporal Replication. *Computer Science - R&D*, 23(3-4):149–157, 2009.

[30] G. Wang, X. Liu, A. Li, and F. Zhang. In-Memory Checkpointing for MPI Programs by XOR-Based Double-Erasure Codes. In *Proceedings of the European PVM/MPI*, pages 84–93, 2009.

[31] B. Welch, M. Unangst, Z. Abbasi, G. Gibson, B. Mueller, J. Small, J. Zelenka, and B. Zhou. Scalable Performance of the Panasas Parallel File System. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 17–33, 2008.

[32] P. Wong and R. F. V. der Wijngaart. NAS Parallel Benchmarks I/O Version 2.4. Technical Report NAS-03-002, NASA Advanced Supercomputing Division, January 2003.