Providing Efficient I/O Redundancy in MPI Environments*

Willam Gropp, Robert Ross, and Neill Miller

Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, Illinois 60439
{gropp,rross,neillm}@mcs.anl.gov

Abstract. Highly parallel applications often make use of either highly parallel file systems or large numbers of independent disks. Either of these approaches can provide high data rates necessary for parallel applications. However, the failure of a single disk or server can render the data useless. Conventional techniques, such as those based on applying erasure correcting codes to each file write, are prohibitively expensive for massively parallel scientific applications because of the granularity of access at which codes are applied. In this paper we demonstrate a scalable method for recovering from single disk failures that is optimized for typical scientific data sets. This approach exploits coarser-grain (but precise) semantics to reduce the overhead of constructing recovery data and makes use of parallel computation (proportional to the data size and independent of number of processors) to construct data. Experiments showing the efficiency of this approach on a cluster with independent disks are presented, and a technique for hiding the creation of redundant data within the MPI-IO implementation is described.

1 Introduction

The scale of today's systems and applications requires very high performance I/O systems. Because single disk performance has not improved at an adequate rate, current I/O systems employ 100s of disks in order to obtain the needed aggregate performance. This approach effectively solves the performance problem, but it brings with it a new problem: increased chance of component failure. General approaches for maintaining redundant data have been applied to locally attached disk storage systems (e.g. RAID [7]). These approaches, however, work at a very fine granularity. Applying this type of approach in a parallel file system or across the local disks in a cluster, where disks are distributed and latencies are higher, imposes unacceptable overhead on the system.

Fortunately, there are characteristics of computational science applications and storage systems that provide opportunities for more efficient approaches.

* This work was supported by the Mathematical, Information, and Computational Sciences Division subprogram of the Office of Advanced Scientific Computing, U.S. Department of Energy, under Contract W-31-109-Eng-38.

First, the applications that generate large amounts of data typically have phases of I/O where much data is written at once, such as in the case of checkpointing. A partially written checkpoint isn't particularly useful, so generally one or more previous checkpoints are preserved in case a failure occurs during the checkpointing process. Thus creating redundant data for a checkpoint once the application is finished writing it adequately covers the failure cases in which the application is interested. Second, we can generally detect a failed disk. Failures where the location of lost data is known are categorized in coding theory as *erasures*. This category of data loss is more easily corrected than the general case of potentially corrupted data located anywhere in the system, meaning that more algorithms are applicable to the problem of recovering from failures in this environment.

The goal of this paper is to describe *Lazy Redundancy*, a technique for matching the generation of recovery data to the needs of high-performance parallel scientific applications and the systems on which these applications run. The lazy redundancy technique embodies two main principles: aggregating the creation of error correcting data at explicit points, and leveraging the resources of clients and storage devices in creating error correcting data or recovering from failures. Specifically we will describe a method for efficient creation of the data necessary to recover from a single erasure failure that is applied only when processes reach an I/O synchronization point (e.g. `MPI_File_close` or `MPI_File_sync`). Because all processes have reached this point, we may use the processes to perform the recovery data calculation. MPI collectives may be used to make communication as efficient as possible. This approach may be applied both in the case where applications write out individual files to local disks and in the case where a parallel file system is used as the backing store. Further, the approach can be generalized to provide redundancy in the presence of more than one failure through the application of well-known algorithms (such as [1] for two erasure (still using only XOR operations) or the more general approaches based on Reed-Solomon codes [9]).

There is a great deal of work on recovery from erasures in file systems. Much of this work is focused on relatively small numbers of separate disks and on preserving data after each `write` operation, as required by POSIX semantics. Work on large arrays of disks has often focused on the multiple-erasure case (e.g., [4, 1, 9]). These approaches can be adapted to the techniques in this paper. Other work has exploited the semantics of operations to provide improved performance. Ligon [6] demonstrated recovery of a lost data in a PVFS file system using serial recovery algorithms to create error recovery data after application completion and to reconstruct data after a single server failure. Pillai et. al. studied the implementation of redundancy schemes in PVFS maintaining the existing PVFS semantics and switching between mirroring and parity based on write size [8]. Ladin et. al. discusses *Lazy Replication* in the context of distributed services [5].

In Section 2 we describe the lazy redundancy technique and our implementation. In Section 3 we demonstrate the technique in a cluster with separate local disks. In Section 4 we summarize the contributions of this work and point to future research directions.

2 Implementing Lazy Redundancy

Let there be p "storage devices," where a storage device could be a disk local to a processor or a server in a parallel file system. For the purposes of this work an erasure failure will be assumed to affect exactly one storage device. Files are striped across all storage devices in a round-robin fashion. We will denote the piece of a stripe residing on one storage device as a *data block*, or *block*. Each data block is of some fixed size (e.g. a natural block size for the disk or parallel file system server, such as 64 Kbytes). A file consists of a set of blocks $A_{i,j}$, where storage device j has blocks $i = 0, 1, \ldots$. The top diagram in Figure 1 shows the layout of these blocks onto storage devices in a round-robin fashion. We wish to compute data P_j called the *parity blocks*, stored on device j, that allows the reconstruction of $A_{i,m}$ if any one storage device m is lost.

The particular algorithms for calculating these parity blocks and placing of these blocks on storage devices in this implementation of the lazy redundancy technique are based closely on the RAID5 [7] work.

For simplicity, assume that $0 \leq i < p$. If $i \geq p$, this approach may be applied iteratively, starting with a new parity block stored on the first storage device. Let $a \oplus b$ be bitwise exclusive or. Define

$$P_j = \left(\bigoplus_{0 \leq k < j} A_{j-1,k} \right) \bigoplus \left(\bigoplus_{j < k < p} A_{j,k} \right) \tag{1}$$

Note that the parity block P_j does not involve any data from device j and that, because of the property of exclusive OR, for any m, all of the data blocks $A_{i,m}$ can be recomputed using the data $A_{i,k}$ for $k \neq m$ and the parity blocks P_k for $k \neq m$.

These parity blocks are evenly distributed among the storage devices to best balance the I/O load, and just as in RAID5 the extra space required by the parity blocks is just $1/p$, where there are p disk storage devices. The middle diagram of Figure 1 illustrates the relationship between data blocks and parity blocks, including the placement of P_j so that it is not on a device storing one of the data blocks used in its calculation. The last diagram of Figure 1 points out that the parity blocks are not necessarily stored in the same file(s) as the data blocks; they could be stored in a separate file or files, or stored in blocks associated with the file.

There are three major differences between the lazy redundancy technique and RAID5: when error correcting data (parity blocks) are computed, how the parity blocks are computed, and how data is reconstructed in the event of an erasure. In the RAID5 approach parity blocks are recomputed on each individual write. This results in a great deal of I/O to parity blocks and at a very fine granularity ($1/p$ the size of the original data). In the lazy redundancy scheme, parity blocks are computed only at explicit synchronization points. This avoids the overhead of computing the parity blocks on small I/O operations and of writing the corresponding even smaller parity blocks, instead aggregating both the calculation and I/O into fewer, larger operations.

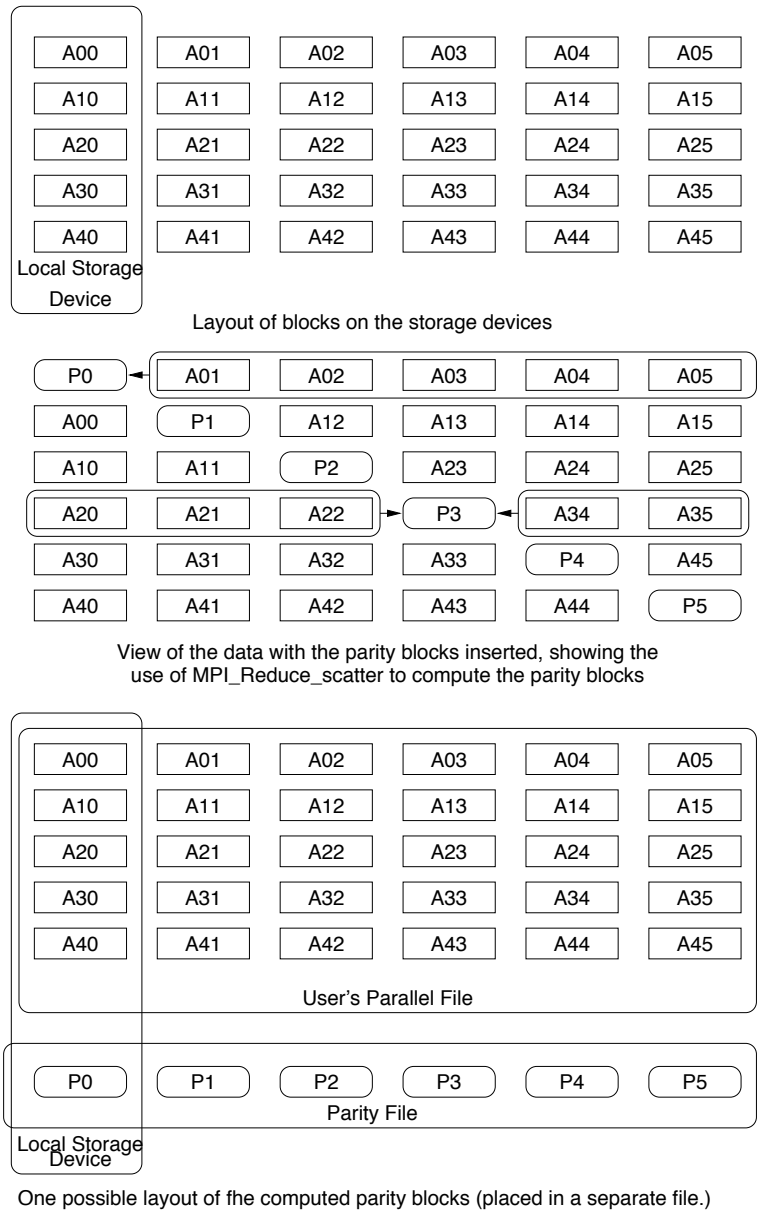

Fig. 1. Real and logical layout of the data blocks $A_{i,j}$ and parity blocks P_j.

The second difference is in the computation of the parity blocks themselves, in particular the communication cost associated with combining data from each storage device. We leverage the fact that all processes have reached a synchronization point to calculate parity blocks collectively (in the MPI sense). We can use algorithms that have been optimized for this sort of calculation, as opposed to performing many independent operations. Because there is no knowledge of a group of processes collectively operating on the file in the general RAID5 model, this type of optimization is impossible.

Equation 1 can be implemented in MPI using the `MPI_Reduce_scatter` operation, using the predefined combiner operation `MPI_BXOR`. The reduce-scatter operation in MPI is functionally equivalent to an `MPI_Reduce` followed by an `MPI_Scatter`. However, the combined operation can often be implemented more efficiently than using the two separate operations. The time to create the parity block can be estimated as

$$T_{parity} = T_{read}(b) + T_{write}(b/p) + T_c(b/p) \tag{2}$$

Here, $T_{read}(n)$ and $T_{write}(n)$ are the times to read and write n bytes respectively. These are single operations involving large blocks that, with some care, can be aligned with the blocks on the storage device itself, leading to optimal I/O transfers. The third term is the cost of the `MPI_Reduce_scatter` used to compute the parity blocks (this assumes an optimal implementation of the reduce-scatter operation, such as that described in [10], whose cost is proportional to the data size for sufficiently large data). Pseudo-code implementing the computation of the parity blocks is shown in Figure 2.

The use of `MPI_Reduce_scatter` allows this code to exploit any optimizations in the implementation of this operation that may be included within the MPI library [10].

Finally, a similar approach to the one used in computing parity blocks is also applied to reconstruction. To reconstruct the data blocks for a failed device, we exploit the properties of exclusive OR. If the rank of the failed device is f, then the data blocks of the failed storage device can be recovered using Equation 3:

$$A_{j,f} = P_j \oplus \left(\bigoplus_{\substack{0 \le k < j \\ k \neq f}} A_{j-1,k} \right) \bigoplus \left(\bigoplus_{\substack{j < k < p \\ k \neq f}} A_{j,k} \right) \tag{3}$$

Reconstruction of a block can be implemented with the routine `MPI_Reduce`; this combines data from all processes to a single process, as shown in Figure 3.

This combination of aggregation of calculations and use of optimized collectives results in very efficient parity calculations.

3 Experiments

In this section, we demonstrate the effectiveness of this approach on a cluster where each node has a separate disk. The parallel application writes data files,

```
/* Create the datatype */
b = 65536; /* Bytes per block */
MPI_Comm_rank( comm, &j );
MPI_Comm_size( comm, &p );
blklens[0] = j*b;
blklens[1] = (p-1-j)*b;
displs[0]  = 0;
displs[1]  = (j+1)*b;
MPI_Type_indexed( 2, blklens, displs, MPI_BYTE, &rtype );
MPI_Type_commit( &rtype );

/* Clear the part of the buffer corresponding to our process */
for (i=0; i<b; i++)
   buf[j*b + i] = 0;
MPI_File_open( MPI_COMM_SELF, filename, ..., &fh );
MPI_File_read( fh, buf, 1, rtype, MPI_STATUS_IGNORE );
MPI_File_close( &fh );

for (i=0; i<p; i++)
  recvcounts[i] = b;
MPI_Reduce_scatter( buf, rbuf, recvcounts, MPI_BYTE, MPI_BXOR, comm );
MPI_File_open( parityfile, ..., MPI_COMM_SELF, fh );
MPI_File_write( fh, rbuf, b, MPI_BYTE, MPI_STATUS_IGNORE );
MPI_File_close( &fh );
```

Fig. 2. Pseudo-code for computing the parity blocks and distributing them among the processes. This code assumes that each process in the communicator `comm` has an associated independent disk, rather than using a parallel file system.

```
/* Create the Datatype rtype as in the parity construction code */
/* Create a communicator ordered in the same way as the original job */
MPI_Comm_split( incomm, 0, oldrank, &comm );
MPI_Comm_rank( comm, &j );
if (j != failedRank) {
     MPI_File_open( MPI_COMM_SELF, filename, ..., &fh );
     MPI_File_read( fh, buf, 1, rtype, MPI_STATUS_IGNORE );
     MPI_File_close( &fh );
     MPI_File_open( MPI_COMM_SELF, parityfile, ..., &fh );
     MPI_File_read( fh, &buf[j*b], b, MPI_BYTE, MPI_STATUS_IGNORE );
     MPI_File_close( &fh );
}
else {
     for (i=0; i<p*b; i++) buf[i] = 0;
}
MPI_Reduce( buf, outbuf, p*b, MPI_BYTE, MPI_BXOR, failedRank, comm );
if (j == failedRank) {
     MPI_File_open( MPI_COMM_SELF, filename, ..., &fh );
     MPI_File_write( fh, outbuf, p*b, MPI_BYTE, MPI_STATUS_IGNORE );
     MPI_File_close( &fh );
}
```

Fig. 3. Pseudocode to reconstruct a block. MPI I/O was used to keep the MPI flavor and to use a single library for all operations in the examples. The read operation used is nothing more than a Unix-style `readv` with a two element `iovec`.

one per process, to the local disk. This is a common approach for parallel applications, particularly those running on systems that do not provide an effective parallel file system.

We do not compare this approach to a traditional RAID-5 approach for a couple of reasons. First, we did not have a comparable system available. Also, building such a system would be an endeavor of larger scope than the work presented here. Finally, the performance of a RAID-5 system would be very dependent on the pattern of access of the processes, while the lazy redundancy approach is not.

The specific scenario that we consider is this: each process in an MPI program writes data to a local disk (that is, a disk on the same processor node as the MPI process). This collection of files represents the output from the MPI program. If one node fails for any reason (such as a failure in the network card, fan, or power supply, not only the disk), the data from that disk is no longer accessible. With the lazy redundancy approach described in the previous section, it is possible to recover from this situation. Assuming that the parity blocks have been computed as described in Section 2, the user can restore the "lost" data by running an MPI program on the remaining $p - 1$ nodes and on one new node. In these experiments we assume that failure detection is handled by some other system software component.

Our experiment simulates this scenario by performing the following steps on a Linux cluster [3] where each node has a local disk. Two underlying communication methods implemented in MPICH2 were used: TCP and an implementation of the GASNet [2] protocol on top of the GM based Myrinet interconnect. We would expect the Myrinet results to be superior when communication is necessary, such as in the process of computing parity or restoring from an erasure. Each step is timed so that the relative cost of each step can be computed:

1. All processes generate and write out data to their local disks.
2. All processes read the data needed to compute the parity blocks.
3. The parity blocks are computed and distributed to each participating process.
4. The parity blocks are written to local disk in a separate file.
5. One process is selected as the one with the failed data (whose data and parity data is removed), and the reconstruction algorithm is executed on this storage device, recreating the missing data.

The results of running this experiment on 4, 16, and more storage devices (and the same number of MPI processes) are shown in Table 1. In each example, the total size of the data scales with p (there are $p - 1$ blocks of the same size on each storage device). Thus, the best performance is linear scaling of the time with respect to the number of processes. Hardware problems precluded running Myrinet experiments at 128 processes.

In this experiment, each process generates a separate data file with the name `datafile.`*rank*, where *rank* is the process rank according to MPI_Comm_rank. The parity data, once computed and gathered, is written to local disk alongside the data file as a "hidden" file with the name `.LZY.datafile.`*rank*. The size of

Table 1. Average time (in seconds) to create a file, create the parity blocks using lazy redundancy, and to restore a file on 4 to 128 storage devices. The step to create the parity block is broken down into the three substeps shown. Parity data size is fixed at 512 Kbytes per process.

	TCP/IP				Myrinet			
	4	16	64	128	4	16	32	64
Data per Client (KB)	1536	7680	32256	65024	1536	7680	16128	32256
Create File	0.040	0.193	0.810	1.632	0.041	0.155	0.319	0.645
Create Parity								
Read Blocks	0.030	0.160	0.656	1.741	0.023	0.123	0.270	0.506
Compute Parity	0.288	1.812	8.270	62.71	0.378	0.463	0.594	1.307
Write Parity	0.013	0.013	0.013	0.013	0.013	0.013	0.013	0.013
Restore Erasure	0.307	1.310	5.490	15.06	0.092	0.425	1.212	1.696

the parity data remains fixed for all tests, regardless of the increasing number of participating processes and data sizes. To achieve the larger data sizes used, the parity computation was applied iteratively a fixed number of times over the varying size of the data composed of data blocks.

The results in Table 1 show the scalability of the approach. For the tests on TCP, the performance scales with the data size to 64 processes. For Myrinet, the times scale better than linearly; we believe that this is due to the serial overheads; in fact, the time for the compute parity step is dominated by a constant overhead of about 0.36 seconds.

Note that the time to create the file is the time seen by the application; this does not necessarily include the time to write the data to disk (the data may be in a write cache).

One benefit of using this approach is that there is a very small amount of additional data necessary to provide fault tolerance. However, calculating this data does become more expensive as the number of clients increases, as we see in the TCP case at 128 processes. In this particular case we may be seeing the impact of memory exhaustion for large reduce-scatter operations (which may need to be broken up in the implementation). However, applying the redundancy scheme to smaller sets of storage devices would also alleviate the problem, at the cost of somewhat larger storage requirements. For example, we could consider the 128 processes as two groups of 64 for the purposes of redundant data calculation, doubling the amount of parity data but allowing for the reduce-scatters to proceed in parallel.

According to the data gathered, it is apparent that a major part of the cost in these examples is reading the file. If either the user-level I/O library (such as an MPI-I/O implementation or a higher-level library) or a parallel file system performed the "Compute Parity" step when sufficient data was available, rather than re-reading all the data as in our tests, then this cost would disappear.

4 Conclusions and Future Work

In this work we have presented the Lazy Redundancy technique for providing erasure correction in an efficient manner on highly parallel systems. The technique exploits collective computation, I/O, and message passing to best leverage the resources available in the system. While tested in an independent file environment, the same approach is equally usable for data stored on a parallel file system. Further, using user-defined reduction operations, this approach can be extended to more complex erasure-correcting codes, such as Reed-Solomon, that would handle larger numbers of failures than the algorithm shown in this work. An additional optimization would leverage accumulate operations during collective writes to avoid the read I/O step of recovery data calculation. This would of course impose additional complexity and overhead in the write phase, and that tradeoff warrants further study.

Although we have presented this work in the context of independent disks, the technique is equally suitable to the parallel file system environment. In order to implement lazy redundancy transparently in this environment, some augmentation is necessary to the parallel file system. We intend to implement this technique as a component of the PVFS2 file system [11]. Our approach will be to transparently perform the lazy redundancy operations within the PVFS2-specific portion of the ROMIO MPI-IO implementation, hiding the details of this operation from the user.

Three capabilities will be added to PVFS2 to facilitate this work: a mechanism for allowing clients to obtain the mapping of file data blocks to servers, an interface allowing clients to perform I/O operations when the file system is only partially available (and be notified of what servers have failed), and a distribution scheme that reserves space for parity data as part of the file object, while keeping parity data out of the file byte stream itself.

References

1. Mario Blaum, Jim Brady, Jehoshua Bruck, Jai Menon, and Alexander Vardy. The EVENODD code and its generalization: An efficient scheme for tolerating multiple disk failures in RAID architectures. In Hai Jin, Toni Cortes, and Rajkumar Buyya, editors, *High Performance Mass Storage and Parallel I/O: Technologies and Applications*, chapter 14, pages 187–208. IEEE Computer Society Press and Wiley, New York, NY, 2001.
2. Dan Bonachea. Gasnet specification, v1.1. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.
3. R. Evard, N. Desai, J. P. Navarro, and D. Nurmi. Clusters as large-scale development facilities. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER02)*, pages 54–66, 2002.
4. Lisa Hellerstein, Garth A. Gibson, Richard M. Karp, Randy H. Katz, and David A. Patterson. Coding techniques for handling failures in large disk arrays. *Algorithmica*, 12(2/3):182–208, 1994.
5. Rivka Ladin, Barbara Liskov, and Liuba Shrira. Lazy replication: Exploiting the semantics of distributed services. In *IEEE Computer Society Technical Committee*

on Operating Systems and Application Environments, volume 4, pages 4–7. IEEE Computer Society, 1990.

6. Walt Ligon. Private communication. 2002.
7. David A. Patterson, Garth Gibson, and Randy H. Katz. A case for redundant arrays of inexpensive disks (RAID). In ACM, editor, *Proceedings of Association for Computing Machinery Special Interest Group on Management of Data: 1988 Annual Conference, Chicago, Illinois, June 1–3*, pages 109–116, New York, NY 10036, USA, 1988. ACM Press.
8. Manoj Pillai and Mario Lauria. A high performance redundancy scheme for cluster file systems. In *Proceedings of the 2003 IEEE International Conference on Cluster Computing*, Kowloon, Hong Kong, December 2003.
9. James S. Plank. A tutorial on reed-solomon coding for fault-tolerance in RAID-like systems. *Software—Practice and Experience*, 27(9):995–1012, September 1997.
10. Rajeev Thakur and William Gropp. Improving the performance of collective operations in MPICH. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, number LNCS2840 in Lecture Notes in Computer Science, pages 257–267. Springer Verlag, 2003. 10th European PVM/MPI User's Group Meeting, Venice, Italy.
11. The PVFS2 parallel file system. `http://www.pvfs.org/pvfs2/`.