Get out of the way! Applying compression to internal data structures

Rob Latham and Matthieu Dorier and Rob Ross
Mathematics and Computer Science Division
Argonne National Laboratory
Lemont, IL

***Abstract*—As the amount of memory per core decreases in post-petascale machines, the memory footprint of any libraries and middleware used by HPC applications must be reduced. While scientific data can contain a great deal of entropy and require specialized compression techniques, the *descriptions* of scientific data layouts, as opposed to contents, turn out to be highly compressible. In this paper we present two approaches to compressing scientific data layout descriptions. We also describe two data structures for managing the compressed data. We incorporated our approach into the ROMIO MPI-IO implementation to reduce the memory consumption, observing an $89\times$ reduction in memory overhead with a 25% increase in CPU overhead.**

I. Introduction

> "Limited amounts of memory and low memory/flop ratios will make processing virtually free. In fact, the amount of memory is relatively decreasing, scaling far worse than computation." – Horst Simon [1]

Sites are procuring and deploying machines with an astounding amount of parallelism, but these large numbers of cores will have access to relatively little memory. Applications using these machines are going to consume as much memory as available and will appreciate any steps taken to reduce the memory footprint of middleware, such as the MPI library.

The ROMIO MPI-IO implementation [17], part of MPICH and many other MPI implementations, provides scalable, high-performance routines for MPI applications. ROMIO's 20-year-old design continues to provide powerful optimizations, but some choices need to be revisited as problem sizes scale up. In particular we have identified an area of the ROMIO MPI-IO implementation that can, for certain classes of MPI datatypes, consume large amounts of memory.

Our goal in this paper is to present data structures for managing random, partial access to compressed data with reasonable CPU costs. ROMIO provides an attractive and relevant context to evaluate our approaches though we have designed our approach to be a general solution and not tied specifically to ROMIO.

A. ROMIO and MPI Datatypes

The ROMIO datatype flattening code processes an MPI datatype to generate two arrays, one listing the "indices" (offsets of data regions relative to the address of the first element) and the other listing the "blocklens" of the type (size of each data region). Figure 1 depicts a small example. An `MPI_Type_vector` is created with five elements, each element being composed of 2 integers. The starting addresses of two consecutive elements are separated by 10 integers (40 bytes). This type presents a very high regularity: values in the array of indices are increasing by steps of 40, and all values are identical in the array of blocklens. The generated arrays for vector types always demonstrate this regularity. Generated arrays for indexed-based descriptions, however, may not do so, as we will discuss in Section IV-C.

C code describing type:

```
MPI_Type_vector(5, 2, 10,
        MPI_INT, &vec_type);
```

notional flattened representation:

```
(0,8),(40,8),(80,8),(120,8),(160,8)
```

C library representation:

```
indices[] = {0, 40, 80, 120, 160}
blocklens[] = {8, 8, 8, 8, 8}
```

Fig. 1. ROMIO's model for representing MPI types; example using a vector type.

Codes can call MPI-IO routines directly with handwritten datatype descriptions. Typically these handwritten versions are not elaborate. We also see library-generated datatype descriptions. The datatypes generated by HDF5 or Parallel-NetCDF, for example, tend to describe much more data and result in much larger flattened representations. One of these library-generated types [2] drew our attention to ROMIO's memory usage.

Other approaches to handling datatypes have been studied ([19], [14]), but the ROMIO flattening code still lives on, largely because it has seen 20 years of testing and bug fixes. Even in these other approaches, however, cases exist where the original user input must be maintained. For example, INDEXED and other types constructed from a list of user-provided arguments need to maintain those arguments: the MPI standard dictates that the type inspection routines (CONTENTS and ENVELOPE) must return the original parameters provided by the user.

We developed two approaches that dramatically reduce ROMIO's memory consumption with minimum impact on ROMIO's implementation. Since the data structures ROMIO uses to maintain datatype descriptions (simple arrays of 64-bit values) turn out to present low entropy, we use compression to lower their memory footprint. We developed an array-oriented interface allowing applications to store data in an abstract

Workload	Original	Compressed	Ratio
coll_perf	9.562 MiB	176.0 KiB	1.798%
mbconvert	3200 Bytes	657 Bytes	20.5%
bigio (subset)	15.00 MiB	101.7 KiB	.6623%

TABLE I
EFFECTIVENESS OF BLOSC-LZ COMPRESSION ON FLATTENED DATA

Workload	Original	Compressed	Ratio
coll_perf	9.562 MiB	336 Bytes	˜0.0%
mbconvert	3200 Bytes	4736 Byte	148%
bigio (subset)	15.00 MiB	48 Bytes	˜0.0%

TABLE II
EFFECTIVENESS OF GRAMMAR-BASED COMPRESSION ON FLATTENED DATA

arraylike form. Under the abstraction, data is compressed, possibly with a smaller decompressed working set. We exploit our knowledge of ROMIO's access patterns: by introducing a small cache of uncompressed data, ROMIO needs to consult the underlying compressed data structure less frequently. Our approaches reduce ROMIO's peak memory use and one of our approaches comes close to matching unmodified ROMIO's run time.

B. Preliminary Analysis

Before integrating any kind of compression strategy into ROMIO, we first confirmed our hypothesis that ROMIO flattening data was indeed compressible. We instrumented ROMIO to dump out in binary form the (offset, blocklength) pairs it generated after flattening a datatype. We used three codes generating datatypes: mbconvert, bigio, and coll_perf. For more information about mbconvert and bigio and their datatypes, see Section IV. Since the full bigio datatype dump was 64 GiB, we looked only at a representative subset (15 MiB). The coll_perf benchmark is one of ROMIO's internal tests and constructs a 3D DARRAY type with a (BLOCK, BLOCK, BLOCK) distribution. After capturing ROMIO's representation, we compressed these binary dumps with the Blosc [3], [4] framework. Table I shows our results. The data compresses to a tiny fraction of original size, except for mbconvert. This mesh-oriented utility generates many short, highly irregular indexed-based datatypes; but, even so, a general-purpose compressor can shrink the data by 80%.

While the internal ROMIO arrays may contain billions of elements, these arrays were generated from routines that in most cases take only a few parameters. A grammar-based approach captures the structure of the data and shrinks the representation even further in these cases. However, if there is no regularity, as in the mbconvert case, the grammar approach is unable to reduce the data, as shown in Table II.

II. RELATED WORK

Wilson et al. [21] and Douglis [9] looked at compressing virtual memory pages, allowing the system to keep more (compressed) memory pages resident and to avoid disk access. Our approach more closely focuses on a user-space workload where data is known to be highly-compressible and need not provide the complexity of a fully general approach required of the virtual memory subsystem.

Our approach here is distinctly different from compressing scientific datasets. Scientific datasets generally contain a great deal of entropy and so are not well suited for general-purpose compressors such as gzip or bzip2. Preconditioners such as ISOBAR [15] can separate high-entropy blocks from low-entropy blocks. Special-purpose compressors can operate better on floating point data [12].

We are not the first group to recognize and exploit regularity of I/O access patterns. PLFS speeds up log replay by identifying patterns, then replacing large metadata logs with a more concise record [10].

Whereas we are studying compression to work around memory limitations, other groups have used compression to work around storage bandwidth limitations. Zou et al. [22] used compression to mitigate data movement costs between simulation and analytics components. Likewise, Welton et al. [20] improved observed network bandwidth by compressing messages between compute nodes and I/O forwarding nodes. Similarly, Bui et al. [5] applied compression to scientific datasets to improve transfer rates. We are not operating on scientific data, but we did incorporate the Blosc compression framework into our design in part because of how well Blosc performed in Bui's work.

To be clear, we are making no claims to increased performance: outperforming an array access in C would be quite a feat. Rather, we are using compression to make possible what, because of system constraints, was not possible.

III. IMPLEMENTATION

When dealing with compressed data, one can decide to apply a general-purpose compression algorithm able to handle most workloads reasonably well or a more specialized algorithm able to handle a specific workload with superior efficiency. We explored both approaches in this work. For the general purpose compression algorithm we used the blosc compression framework [3] with the blosc-lz compressor.

The ROMIO flattened representation can often have a great deal of regularity. A grammar-based approach can capture that regularity and compress certain classes of arbitrarily long arrays into a few bytes worth of grammar rules. We cover both techniques in more detail in subsequent subsections.

A. Access Pattern

In addition to the kind of data stored in these ROMIO arrays, we also know how these data structures will be accessed. ROMIO has two behaviors: one for creating the flattened representation and one when using the representation to compute I/O and memory access patterns.

When creating the representation, the access will be append-only writes mixed in with reads of prior elements: the flattening code processes the types essentially linearly (a STRUCT-based type can exhibit more complexity, but in practice that is uncommon). In no cases are elements overwritten. ROMIO does read prior elements to compute the new elements (typical

```
/* initialize comparray library and dependencies */
void comparray_init();

/* clean up */
void comparray_finalize();

/* create a comparray container that stores data
   of size 'type_size' in chunks consisting of a
   'chunk_size' number of elements */
comparray comparray_create(size_t chunk_size,
                           size_t type_size);

/* given an array 'value' of 'count' items, store in
   compressed array 'array' at index 'index' */
int comparray_set_n(comparray array, int64_t index,
          int64_t count, void *value);

/* provide an array 'dest'  to comparray and have
   it fill it with 'count' items beginning from
   index 'index' */
int comparray_fill_n(comparray array,
       int64_t index, int64_t count, void * dest);

/* deallocate everything associated with 'id' */
void comparray_free(comparray id);
```

Fig. 2. Key routines of "compressed array" API

C-code examples would look like $index[i] = index[i-1] * stride$ or $block[i] = block[0]$).

During I/O operations, ROMIO switches to a read-only workload. ROMIO will walk through the flattened representation in a linear fashion. This walk, however, is not strictly sequential. An I/O access can be larger than the type describing the MPI file views. In that situation, the type is "tiled," and ROMIO will start over from the beginning, adding the number of prior filetypes to the offset of the (offset, length) tuple.

B. Compressed Arrays

When designing a data structure to manage compressed data in ROMIO, an arraylike interface provides a good fit for how ROMIO accesses the flattened representation. While we could simply generate the flattened representation, compress it, and then decompress it as needed, operating on the entire representation negates our goals of limiting peak memory consumption. At the same time, ROMIO's access patterns do not need random access to every element. We can keep a smaller working set decompressed while the majority of the data remains compressed.

We call this approach *compressed-arrays*. The compressed-arrays data structure partitions an array into chunks and applies compression to each of those chunks. To manage the collection of compressed chunks, we record the low and high index of each chunk and store the annotated chunks in an interval tree [6]. When accessing a single element of the array, an entire block is decompressed and kept in a cache. ROMIO tends to process the index and blocklength arrays from start to finish, with only a little back tracking, making such a chunk-at-a-time approach a good fit. Figure 2 provides the API. Note the option of operating on multiple values, helping to reduce overhead. The full code is available online [11].

```
 /* Creates a gramarray. The result will be stored in
  * g. gramtype should be either GRAMARRAY_ABSOLUTE or
  * GRAMARRAY_RELATIVE.  ABSOLUTE indicates that
  * values will be stored "as is".  RELATIVE
  * indicates that differences between consecutive
  * values are stored.  */

int gramarray_create(gramarray* g, int gramtype);

/*Frees the gramarray.  */
int gramarray_free(gramarray g);

/* Appends an item at the end of the stream.
 * This will invalidate all the iterators that have
 * been built before.  */
int gramarray_append(gramarray g, gramtype x);

/* Creates an iterator to navigate the grammar.
 * If direction = 1, the iterator points to the beginning
 * of the array and will move forward.
 * If direction = -1, the iterator points to the end of
 * the array and will move backward.  */
int gramarray_iterator_create(gramarray g,
         gramiter* it, int direction);

/* Frees an iterator.  */
int gramarray_iterator_free(gramiter it);

/* Get the current value pointed by the iterator.  */
int gramarray_iterator_value(gramiter it,
        gramtype* val);

/*Increments the iterator.  */
int gramarray_iterator_next(gramiter it);
```

Fig. 3. Key routines of "gramarray" API

C. Grammar-Based Approach

Since datatypes are naturally hierarchical structures because of the way they are created, one can expect to find such hierarchical structures in the list of indices and blocklens as well. Grammar-based approaches, such as Sequitur [13], have been proposed to find such hierarchical structures. In [8] we proposed the StarSequitur algorithm, which improves the Sequitur algorithm by more compactly representing repetitions of sequences. For the present work, we implemented StarSequitur as a C library called *gramarray* [7]. Its API is presented in Figure 3.

In the gramarray library's design, writing can be done only by appending new elements. The gramarray structure does not support the modification or deletion of already inserted elements. Contrary to the compressed arrays presented in the preceding section, in which blocks of elements have to be decompressed before being read, the gramarray structure offers constant-time sequential access (forward or backward) without requiring decompression. However, it does not provide random read access. In the ROMIO context, such limitations are fine: ROMIO writes proceed sequentially through the array, and ROMIO reads are either from the very beginning or looking back a few elements.

When integrated in ROMIO, gramarray compresses separately the array of indices and the array of blocklens. Indices are compressed in a relative manner; that is, after the first index is stored, differences of consecutive indices ($i_{n-1} - i_n$) are stored. Hence, the list of indices {0, 4, 12, 16, 24, 28} is

stored as $\{0, 4, 8, 4, 8, 4\}$, which compresses into the grammar

$$\begin{aligned} S &\rightarrow 0, A^2, 4 \\ A &\rightarrow 4, 8 \end{aligned} \tag{1}$$

(exponents here represent iterations over a value or a pattern). Arrays of blocklens are compressed in an absolute manner.

The StarSequitur algorithm has a linear complexity in the number of symbols being encoded. While experiments presented in the following section show that the grammar approach can be very efficient in terms of compression ratio, they also show that such a compression can be very costly in performance. This is mainly due to the fact that the Sequitur algorithm's implementation is based on pointer arithmetic to navigate in a graph (the grammar) of small elements that are frequently allocated and freed.

D. Cache Design

The compressed-arrays library contains a simple one-block cache under the hood, but with our knowledge of ROMIO's access patterns we can tailor a (simple) cache design to match ROMIO's needs. A ROMIO-level cache has the additional advantage of assisting both the compressed-array and gramarray approaches. The caching mechanism consists of two caches for each array.

- A write-back cache consisting of a single block to absorb updates and allow us to update multiple entries in one batch. An N-element cache will then write to the compressed data structure only once before every Nth append, resulting in a write cache miss rate of $1/N$.
- An N-element readahead cache providing a simple prefetch mechanism to exploit ROMIO's read access patterns. ROMIO has two behaviors with respect to accessing its datatype metadata. ROMIO will either repeatedly consult an item at the beginning of the array, as when constructing a VECTOR datatype, or will consult the previous element, as when constructing a DARRAY type. A least-recently used cache with a small number of entries might be sufficient, but for simplicity we maintained an N-element readahead cache. We expect low miss rates for this read cache as well.

The read and write cache also fits well with the grammar-based limitations, since every append to the grammar invalidates any iterators. Keeping a cache of recently read blocks prevents the need to recreate an invalidated iterator on every read, but instead only when a read access misses the cache.

IV. Evaluation

We measured the peak memory usage of ROMIO. Experiments were conducted on a Linux workstation (4-core Intel Xeon E5430). On Linux (and other systems with the necessary low-level system calls), the GNU "time" utility (not the shell built-in) can report a variety of resource metrics in addition to elapsed time. In these studies we report "maximum resident set size," as our goal is to reduce the peak memory utilization. The time utility does not understand MPI and will dump one report per process. We report the maximum value across all processes. When we report "unmodified MPICH," we are comparing with the master branch of MPICH (v3.2-418-gc5b844ad)

A. Degree of Noncontiguity

We wrote a simple benchmark to produce VECTOR types of increasing noncontiguity. The benchmark constructs a VECTOR MPI type with one block of 16 million integers, then 2 blocks of 8 million integers, and so on until we constructed 16 million blocks of one integer. For each type, we measured the maximum resident set size (as reported by the getrusage system call) and the time (as reported by the MPI_Wtime function) to construct the type and set an MPI file view. Figure 4 shows our results, comparing unmodified MPICH with our "compressed array" approach.

Our compressed representation keeps memory utilization under control. Figure 4b suggests that some additional tuning may be needed when processing the most highly noncontiguous types. We could, for example, use more memory and increase the cache and intermediate block size when types are highly noncontiguous.

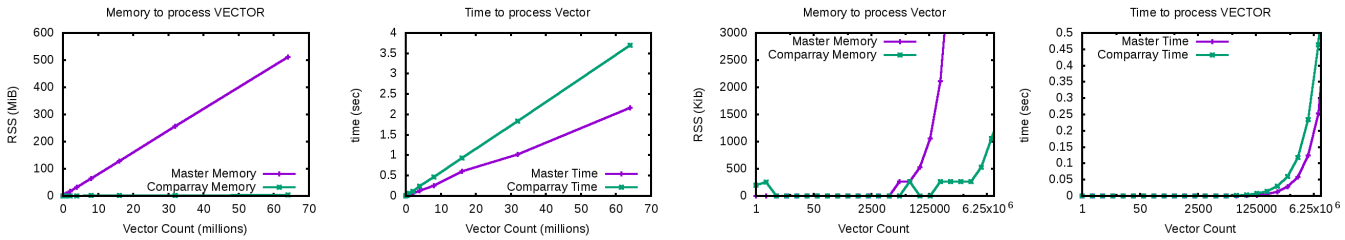
Figures 4c and 4d focuses on small-degree noncontiguity. We plot vector count on a log scale and zoom in on the smaller counts. Immediately one can see how unmodified MPICH memory use escalates quickly once an MPI datatype contains 10,000 or more pieces. The compressed representation is using much less memory while imposing a small CPU overhead. The chart suggests that a dynamic approach, which we have not implemented, should use a compressed representation only when a type description exceeds 10,000 items.

B. HDF5 bigIO

The HDF5 library [18] sits on top of MPI-IO, providing to developers an interface based on multidimensional arrays of typed data. It constructs types not usually seen by handwritten codes and hence provides an excellent test bed for MPI type processing implementations. Our HDF5 collaborators reported being able to exhaust memory when trying out a new feature [2]. Their MPI datatype was not too aggressive: a VECTOR with a single block consisting of a billion RESIZED items. ROMIO's flattened representation of this type, however, consumed 8 GiB of memory.

To highlight the behavior of compressing ROMIO's flattened representation, we focused solely on the creation of that representation. To compare our approach with unmodified MPICH, we had to cut the number of items in the vector block by a fourth in order to fit the problem in memory. We also reduced the benchmark to the creation of the MPI type and the call setting the MPI file view.[1] ROMIO creates its internal flattened representation in this file view call. Since we are omitting the writing and reading of this file, we are also omitting any evaluation of ROMIO's read-only consumption of this datatype. The omission allows us to avoid any variations in

[1]These modifications are reflected in the "bigio_viewonly" test case attached to the referenced ticket.



(a) Memory (b) Time (c) Zooming in Memory (d) Zooming in on Time

Fig. 4. ROMIO's internal representation compresses well, yielding a dramatic reduction in memory usage, though with some cost in processing time.

TABLE III
HDF5 "BIG-IO" CASE: COMPRESSED-ARRAY CAN GREATLY REDUCE THE MEMORY FOOTPRINT BY PAYING A LARGER TYPE PROCESSING COST. THE GRAMMAR APPROACH CAN FURTHER SHRINK MEMORY, THOUGH AT EVEN LARGER COST

	Peak Memory	Execution time
unmodified MPICH	61.25 GiB	66.88 sec
compressed array	0.6902 GiB	84.15 sec
grammar array	0.02594 GiB	2008 sec

TABLE IV
MOAB MBCONVERT UTILITY

	Peak Memory	Execution Time
unmodified MPICH	213136 KiB	0.28 sec
compressed array	214272 KiB	0.36 sec
grammar array	214624 KiB	0.56 sec

I/O performance between runs and make more reliable timing observations. Results are reported in Table III.

The ROMIO-level cache (maintained for both gramarray and compressed-array) reported a 0.78125 % miss rate for index access - an exact match for our theoretical miss rate for a linear walk through the array. It reported a 0.15625% miss rate for blocklengths, indicating a great deal of repeated accesses.

C. MOAB Mesh I/O

MOAB [16] is a component for operating on mesh-oriented data. It provides an interface for describing the various entities of a mesh (points, edges, faces, shapes), defines an object model for reasoning about these meshes, and includes tools for writing and reading meshes. In this work we focus on the I/O features. MOAB's I/O layer stores the mesh database in HDF5, which in turn uses ROMIO for parallel I/O. This deep software stack generates highly irregular MPI datatypes, making heavy use of the INDEXED family of datatypes to capture the location of the points of the mesh.

MOAB contains as part of its unit tests a partitioned mesh stored in a file called `64bricks_12ktet.h5m`. We applied the tool `mbconvert` and generated an output file in parallel and with the "resolve_shared_ents" option. We configured MOAB to use HDF5-1.8.17.

Unlike the HDF5 test case, which created a single large type, the MOAB case creates 6 types. Index and blocklength arrays show a ROMIO-level cache hit rate of less than 1% in all cases. The ROMIO and compressed-array caches are too large relative to the small types created by this configuration of MOAB. MOAB provides further evidence that we should selectively enable compression for only those types exceeding a threshold.

V. CONCLUSION

On-node memory is already a precious resource on today's leadership-class machines and will become more so in the future. Our technique to reduce the memory footprint consumes some CPU time but reduces the memory consumption of a key middleware library.

We presented two utility libraries for managing compressed data and applied them to ROMIO's datatype bookkeeping. The "compressed-arrays" approach, based on a general purpose compression framework, works well in all cases studied. The "gramarray" approach leads to higher compression in several cases but does not fare well when the data is irregular and incurs a high runtime overhead. Gramarray still needs significant work to improve its run time before it becomes viable in production. We can will identify approaches to select the best compression technique on the fly. VECTOR types, for example, will always be regular, while INDEXED types may or may not exhibit regularity. Small datatype descriptions are unlikely to benefit from compression; our dynamic compression selection should extend to a "no-compression" method in such cases.

We have targeted one specific memory-intensive part of ROMIO that was exhausting memory on a modest machine. Other latency-insensitive parts of MPI implementations may also benefit from compressing internal data structures. The compression utilities were written to be independent of ROMIO and could be applied to other codes.

ACKNOWLEDGMENT

This material was based upon work supported by the U.S. Department of Energy, Office of Science, Advanced Scientific Computer Research, under contract DE-AC02-06CH11357 and by the National Science Foundation under grant CCF-1409946

REFERENCES

[1] "No Exascale for You!" An Interview with Berkeley Lab's Horst Simon . http://www.top500.org/blog/no-exascale-for-you-an-interview-with-berkeley-labs-horst-simon/. Accessed: 2016-05-05.

[2] Out of memory with "big i/o". https://trac.mpich.org/projects/mpich/ticket/2082. Accessed: 2016-05-05.

[3] F. Alted. Blosc, an extremely fast, multi-threaded, meta-compressor library. http://blosc.org/. Accessed: 2016-05-05.

[4] F. Alted. Why modern cpus are starving and what can be done about it. *Computing in Science and Engineering*, 12(2):68–71, 2010.

[5] H. Bui, H. Finkel, V. Vishwanath, S. Habib, K. Heitmann, J. Leigh, M. Papka, and K. Harms. *Scalable parallel I/O on a blue Gene/Q supercomputer using compression, topology-aware data aggregation, and subfiling*, pages 107–112. IEEE Computer Society, 1 2014.

[6] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[7] M. Dorier. Gramarray. http://bitbucket.org/mdorier/gramarray.

[8] M. Dorier, S. Ibrahim, G. Antoniu, and R. Ross. Using Formal Grammars to Predict I/O Behaviors in HPC: the Omnisc'IO Approach. *IEEE Transactions on Parallel and Distributed Systems*, 2015.

[9] F. Douglis. The compression cache: Using on-line compression to extend physical memory. In *In Proceedings of 1993 Winter USENIX Conference*, pages 519–529, 1993.

[10] J. He, J. Bent, A. Torres, G. Grider, G. Gibson, C. Maltzahn, and X. H. Sun. Discovering structure in unstructured i/o. In *High Performance Computing, Networking, Storage and Analysis (SCC), 2012 SC Companion:*, pages 1–6, Nov 2012.

[11] R. Latham. A collection of data structures. https://xgitlab.cels.anl.gov/robl/data_structures. Accessed: 2016-05-12.

[12] P. Lindstrom and M. Isenburg. Fast and efficient compression of floating-point data. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1245–1250, Sept. 2006.

[13] C. G. Nevill-Manning and I. H. Witten. Identifying hierarchical structure in sequences: A linear-time algorithm. *J. Artif. Int. Res.*, 7(1):67–82, Sept. 1997.

[14] R. Ross, N. Miller, and W. Gropp. Implementing fast and reusable datatype processing. In *Proceedings of the 10th EuroPVM/MPI Conference*, Sept. 2003.

[15] E. R. Schendel, Y. Jin, N. Shah, J. Chen, C. Chang, S.-H. Ku, S. Ethier, S. Klasky, R. Latham, R. Ross, and N. F. Samatova. ISOBAR preconditioner for effective and high-throughput lossless data compression. In *Proceedings of the 28th IEEE International Conference on Data Engineering (ICDE)*, Washington, DC, Apr. 2012.

[16] T. J. Tautges, R. Meyers, K. Merkley, C. Stimpson, and C. Ernst. MOAB: a mesh-oriented database. SAND2004-1592, Sandia National Laboratories, Apr. 2004. Report.

[17] R. Thakur, W. Gropp, and E. Lusk. On implementing MPI-IO portably and with high performance. In *Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems*, pages 23–32. ACM Press, May 1999.

[18] The HDF Group. Hierarchical Data Format, version 5, 1997-2016. http://www.hdfgroup.org/HDF5/.

[19] J. L. Traff, R. Hempel, H. Ritzdorf, and F. Zimmermann. Flattening on the fly: Efficient handling of MPI derived datatypes. In *PVM/MPI 1999*, pages 109–116, 1999.

[20] B. Welton, D. Kimpe, J. Cope, C. M. Patrick, K. Iskra, and R. B. Ross. Improving I/O forwarding throughput with data compression. In *Interfaces and Abstractions for Scientific Data Storage Workshop, Austin, TX, USA, September 26-30, 2011*, pages 438–445, 2011.

[21] P. R. Wilson, S. F. Kaplan, and Y. Smaragdakis. The case for compressed caching in virtual memory systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '99, pages 8–8, Berkeley, CA, USA, 1999. USENIX Association.

[22] H. Zou, Y. Yu, W. Tang, and H. M. Chen. Improving i/o performance with adaptive data compression for big data applications. In *Proceedings of the 2014 IEEE International Parallel & Distributed Processing Symposium Workshops*, IPDPSW '14, pages 1228–1237, Washington, DC, USA, 2014. IEEE Computer Society.