Processing MPI Datatypes outside MPI

Robert Ross[1], Robert Latham[1], William Gropp[2],
Ewing Lusk[1], and Rajeev Thakur[1]

[1] Mathematics and Computer Science Division
Argonne National Laboratory
Argonne, IL 60439, USA
{rross|robl|lusk|thakur}@mcs.anl.gov
[2] Computer Science Department
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
wgropp@illinois.edu

Abstract. The MPI datatype functionality provides a powerful tool for describing structured memory and file regions in parallel applications, enabling noncontiguous data to be operated on by MPI communication and I/O routines. However, no facilities are provided by the MPI standard to allow users to efficiently manipulate MPI datatypes in their own codes.

We present *MPITypes*, an open source, portable library that enables the construction of efficient MPI datatype processing routines outside the MPI implementation. MPITypes enables programmers who are not MPI implementors to create efficient datatype processing routines. We show the use of MPITypes in three examples: copying data between user buffers and a "pack" buffer, encoding of data in a portable format, and transpacking. Our experimental evaluation shows that the implementation achieves rates comparable to existing MPI implementations.

1 Introduction

An overwhelming majority of high-performance computing (HPC) codes rely on the Message Passing Interface (MPI) standard. Because MPI is understood so well by application teams and is available on virtually all platforms, additional software has been developed that builds on MPI capabilities and concepts, such as ROMIO [1] and Parallel netCDF [2]. Rich datatype description capabilities are an integral part of the MPI standard and are used in all aspects of MPI, from point-to-point and collective communication to I/O and remote memory access. The MPI datatype facilities make it possible for users to conveniently describe complex, noncontiguous, structured data and operate on this data using a variety of MPI calls in an efficient manner.

Unfortunately, there is a distinct lack of functionality in the MPI standard to allow external libraries to efficiently process MPI datatypes. As a result it is difficult to build libraries that use the MPI descriptive capabilities beyond simply passing these descriptions directly to MPI calls. In this paper we describe

MPITypes, a portable, open source library for manipulating MPI datatypes in libraries and applications. Adapted from the datatype processing component of the MPICH2 implementation [3], the MPITypes library includes a set of commonly desired operations, such as copying structured data between a user buffer and a contiguous buffer or generating lists of offsets and lengths described by a datatype (flattening). Additionally, MPITypes provides a framework for developing more complex, case-specific datatype operations for use in libraries and applications. We describe the capabilities of MPITypes through a set of use cases: copying data described with an MPI datatype, data format conversion as performed in Parallel netCDF [2], and an implementation of transpacking [4].

2 Background

HPC libraries take advantage of MPI in various ways. The ROMIO MPI-IO library [1] relies heavily on MPI point-to-point and collective communication to maintain portability across MPI implementations. Parallel netCDF (PnetCDF) [2], a high-level I/O library that provides parallel access semantics to data stored in the netCDF data format [5], likewise relies on MPI file operations to maintain portability across a variety of serial and parallel file systems. In both cases, these libraries rely on MPI datatypes. In the case of PnetCDF, these are used to describe user data structures in memory, while in the case of ROMIO, these types describe regions in memory and in files.

When libraries such as these incorporate the use of MPI datatypes in their interfaces, it is often necessary to process these datatypes in various ways. For example, the ROMIO library must break datatypes apart in order to determine what regions of a file should be accessed and where data resides in memory. In the PnetCDF case, data in memory described by a datatype needs to be encoded in a portable format prior to data movement to file, or decoded prior to placement in the user's buffers. Unfortunately, the MPI standard provides virtually no support for processing of datatypes outside the MPI library. Even the `MPI_Pack` routine, which naïvely appears to provide the functionality needed to move data into a contiguous buffer, cannot be used by external libraries:

> The restriction on "atomic" packing and unpacking of packing units allows the implementation to add at the head of packing units additional information, such as a description of the sender architecture (to be used for type conversion, in a heterogeneous environment). [6]

In other words, the data placed in a buffer by `MPI_Pack` cannot reliably be interpreted by an external application.

Libraries building on MPI have addressed this deficiency in different ways. In ROMIO, facilities were implemented to "flatten" types into a list of offset-length pairs, a limiting factor for some access patterns [7]. PnetCDF processes noncontiguous datatypes by first calling `MPI_Pack` to put the data into a contiguous buffer in this undefined format and then using `MPI_Unpack` with a contiguous

datatype to put the data in a separate contiguous buffer. This data is then encoded in the appropriate format and written to file. Libraries such as these would benefit greatly from a library of routines that allow efficient processing of MPI datatypes. Several researchers have addressed the problem of efficiently processing MPI datatypes [8, 3]. To date, however, these capabilities have not been made available in a useful form to developers building applications or libraries outside the context of MPI implementations. MPITypes fills this gap.

3 The MPITypes Library

MPITypes is a portable library for efficiently processing MPI datatypes in HPC libraries and applications that rely on MPI. MPITypes is based on the MPICH2 datatype processing functionality [3]. It relies on the dataloop representation described in this previous work for efficient processing and operates in the same nonrecursive manner in order to maximize performance. MPITypes provides two capabilities: a set of datatype operators that provide functionality commonly needed by libraries processing MPI datatypes, and a set of functions that form a toolkit for building more specialized type processing.

3.1 Basic MPITypes Functionality

Figure 1 summarizes the MPITypes interface. The first five of these functions provide a basic interface for two of the most common operations on datatypes, packing/unpacking and flattening. The `init` function is responsible for generating the *dataloop* representation of the type used internally for processing, building this from the data available via the MPI datatype envelope and contents calls. These calls provide access to the parameters used to build the MPI datatype. By using these calls to gather this data, MPITypes is portable across all MPI-2 compliant MPI implementations. `init` also allocates a keyval with `MPI_Type_create_keyval` on initial execution and stores data as an attribute on the datatype to be processed. The delete callback associated with the keyval implicitly frees these internal data structures when the last reference to the datatype is freed, eliminating the need for an explicit free operation.

The next three functions perform useful operations on MPI datatypes. `memcpy` is the datatype equivalent of the UNIX function: it copies between a memory region described by an MPI {buffer, count, type} tuple and a contiguous memory region. `flatten` generates lists of displacements and block lengths that specify the regions of memory described by the datatype tuple. `blockct` provides a count of the distinct contiguous regions described by the datatype tuple. All of these functions take start and end offsets, in bytes, that provide the ability to perform partial processing, for example if limited memory is available.

These functions alone would greatly simplify the ROMIO implementation by eliminating the need for an internal flattening functionality and for keeping track of flattened representations. In the case of PnetCDF, the `memcpy` function provides a convenient way to move data between the user's buffer and a contiguous

```
/* Basic Functions (MPIT_Type) */
int MPIT_Type_init(MPI_Datatype type, int flag);

int MPIT_Type_memcpy(void *typebuf, int count, MPI_Datatype type,
   void *streambuf, int direction, MPI_Aint start, MPI_Aint *end);

int MPIT_Type_flatten(void *typebuf, int count, MPI_Datatype type,
   MPI_Aint start, MPI_Aint *end, MPI_Aint *disps, int *blocklens,
   int *count);

int MPIT_Type_blockct(int count, MPI_Datatype type, MPI_Aint start,
   MPI_Aint *end, MPI_Aint *blockct);

/* Toolkit Functions (MPIT_Segment) */
MPIT_Segment *MPIT_Segment_alloc();

int MPIT_Segment_init(void *buf, int count, MPI_Datatype type,
   MPIT_Segment *seg, int flag);

int MPIT_Segment_free(MPIT_Segment *seg);

int MPIT_Segment_manipulate(MPIT_Segment *seg, MPI_Aint start, MPI_Aint *end,
   int (*contigfn) (...), int (*vectorfn) (...), int (*blkidxfn) (...),
   int (*indexfn) (...), MPI_Aint (*sizefn) (MPI_Datatype el_type),
   void *pieceparams);
```

Fig. 1. The functions in MPITypes. The `MPI_Type` functions provide a basic set of operators on MPI datatypes, while the `MPI_Segment` functions serve as a toolkit for implementing more advanced functionality.

encode/decode buffer, eliminating the need for the current `MPI_Pack`/`MPI_Unpack` approach. In both cases, however, more task-specific datatype processing would provide additional benefits.

3.2 MPITypes as a Toolkit

The real power of MPITypes is in its use as a toolkit for building more complex, task-specific datatype processing operations. The `memcpy`, `flatten`, and `blockct` functions are all written in terms of the second set of functions in Figure 1. To explain the use of these functions, we first examine the implementation of `MPIT_Type_memcpy` (Figure 2). The `memcpy` implementation consists of three main components: the `MPIT_memcpy_params` structure, the `memcpy` function, and a set of leaf functions.

The `MPIT_memcpy_params` structure holds data specific to the processing we wish to perform (i.e. buffer locations and a direction for copying). For other types of processing different data might be needed, such as arrays to hold offsets and lengths in the case of flattening. The `memcpy` function initializes the task-specific data structure and calls *segment* functions to accomplish the datatype processing. A *segment* is a structure that holds state about the processing of an MPI datatype. This structure is also used to optimize partial processing of datatypes by maintaining the current position in the datatype. The segment functions are provided as part of MPITypes.

```
/* MPIT_Type_memcpy - Copies data between a region described by an MPI
   (buf, count, type) tuple and a contiguous data buffer.

   Start and end refer to starting and ending byte locations in the type
   map defined by the user datatype. Specifically, end refers to the
   byte offset just past the last to be processed (e.g. to process
   bytes [0..5], start = 0 and end = 6).
*/

typedef struct MPIT_memcpy_params_s {
   int direction;
   char *packbuf;
   char *userbuf;
} MPIT_memcpy_params;


int MPIT_Type_memcpy(void *typebuf, int count, MPI_Datatype type,
   void *streambuf, int direction, MPI_Aint start, MPI_Aint *end)
{
   int mpi_errno;
   MPIT_Segment *segp;
   MPIT_memcpy_params params;

   segp = MPIT_Segment_alloc();
   MPIT_Segment_init(NULL, count, type, segp, 0);

   params.userbuf   = typebuf;
   params.packbuf   = packbuf;
   params.direction = direction;

   MPIT_Segment_manipulate(segp, start, end, MPIT_Leaf_contig_memcpy,
      MPIT_Leaf_vector_memcpy, MPIT_Leaf_blkidx_memcpy,
      MPIT_Leaf_index_memcpy, NULL, &params);

   MPIT_Segment_free(segp);
   return MPI_SUCCESS;
}


int MPIT_Leaf_contig_memcpy(MPI_Aint *blocks_p, MPI_Type el_type,
   MPI_Aint dtype_pos, void *unused, void *v_paramp)
{
   MPI_Aint el_size;
   MPI_Aint size;
   MPIT_memcpy_params *paramp = v_paramp;

   MPI_Type_size(el_type, &el_size);
   size = *blocks_p * el_size;

   if (paramp->direction == MPIT_MEMCPY_TO_USERBUF)
      memcpy(paramp->userbuf + dtype_pos, paramp->packbuf, size);
   else
      memcpy(paramp->packbuf, paramp->userbuf + dtype_pos, size);

   paramp->packbuf += size;
   return 0;
}
```

Fig. 2. Excerpts from the implementation of `MPIT_Type_memcpy`. This implementation allows copying of subregions for pipelining or memory management purposes. Only contiguous leaf function is shown.

The third main component, the leaf functions, are used by `Segment_manipulate`. `Segment_manipulate` drives datatype processing by walking the dataloop representation of the datatype, tracking the current position in the datatype and storing this information in the segment. When it encounters a "leaf" node in the dataloop tree, it executes the leaf function corresponding to the type of leaf node encountered. One of these, `MPIT_Leaf_contig_memcpy`, is shown.

The four leaf functions correspond to the four possible dataloop leaf types: contiguous, vector, block indexed, and indexed. Along with the relative position tracked by the manipulate function, the data in the leaf node specifies the mapping of a specific set of user buffer locations to types in the MPI typemap. For example, in the case of a vector leaf node, the leaf would describe a set of strided data regions using a count, block size, and a stride. The task of the leaf function is to perform whatever operation is desired for these regions, in this case copying data between a contiguous user buffer and the region(s) described in the leaf node.

In `MPIT_Leaf_contig_memcpy`, the function copies data for a single contiguous region in both the user buffer and the contiguous buffer. The `MPIT_memcpy_params` tracks the initial user buffer pointer (`userbuf`) and the next contiguous buffer offset (`packbuf`), while the current datatype location, relative to the initial user buffer pointer, is provided by the manipulate function (`dtype_pos`).

`Segment_manipulate` understands how to use a contiguous leaf function to process any other type of leaf function, so a simple implementation of `memcpy` would only include our contiguous leaf function. However, in cases where the overhead of processing the type is expected to dominate, such as when simply copying data, implementing support for the other three types of leaf nodes provides a substantial boost in performance. The MPITypes implementation of `memcpy` includes all four leaf-processing functions.

4 Case Studies in Datatype Processing

MPITypes provides a great deal of convenience for users who wish to work with MPI datatypes, but if it does not perform efficiently, then it has little practical value. In this section we evaluate the MPITypes framework. First, we compare the performance of MPITypes when copying data to that of `MPI_Pack`/`MPI_Unpack` and manual copying of data. Next, we examine the cost of an MPITypes-based PnetCDF data coding implementation as compared to simply copying data and to the existing PnetCDF approach, for an example data type. We also examine the performance of a MPITypes-based transpacking implementation as compared to the naïve pack/unpack approach.

All tests were performed on the "breadboard" system at Argonne National Laboratory. The node used for testing is an 8-core, 2.66 GHz Intel Xeon system with 16 Gbytes of main memory, Linux 2.6.27, and gcc 4.2.4. Tests with MPICH2 use version 1.0.8p1, compiled with "`--enable-fast=O3`". Tests with Open MPI use version 1.3.1, compiled with "`CFLAGS=-O3 --disable-heterogeneous --enable-shared=no --enable-static --with-mpi-param-check=no`".

Table 1. Comparing `MPI_Pack`/`MPI_Unpack`, `MPIT_Type_memcpy`, and manual copy rates.

Test	Element Type	MPICH2 (MB/sec)	Open MPI (MB/sec)	MPITypes (MB/sec)	Manual (MB/sec)	Size (MB)	Extent (MB)
Contig	float	4578.49	4561.09	4579.84	2977.98	4.00	4.00
Contig	double	4152.07	4157.69	4149.13	2650.81	8.00	8.00
Vector	float	1788.85	1088.01	1789.37	1791.59	4.00	8.00
Vector	double	1776.81	1680.23	1777.04	1777.60	8.00	16.00
Indexed	float	803.49	632.32	829.75	1514.94	2.00	4.00
Indexed	double	1120.59	967.69	1123.97	1575.41	4.00	8.00
XY Face	float	18014.16	15700.85	17962.98	9630.47	0.25	0.25
XY Face	double	17564.43	18143.63	17520.11	16423.59	0.50	0.50
XZ Face	float	3205.28	3271.29	3190.69	3161.28	0.25	63.75
XZ Face	double	4004.26	4346.81	3975.23	3942.41	0.50	127.50
YZ Face	float	145.32	93.08	145.32	143.68	0.25	63.99
YZ Face	double	153.89	154.19	153.88	153.96	0.50	127.99

4.1 Moving Data between User Buffers and Contiguous Buffers

We have modified the synthetic tests used in [3] to compare the MPITypes implementation of `MPIT_Type_memcpy` with the `MPI_Pack` and `MPI_Unpack` routines and hand-coded routines that manually pack and unpack data using tight loops that exploit knowledge of data layout. These tests cover a wide variety of possible user types, from simple strided patterns to complex, nested strides and types with no apparent regularity.

Each test begins by allocating memory, initializing the data region, and creating a MPI type describing the region. Next, `MPIT_Type_init` is called to generate the dataloop representation used by MPITypes and store it as an attribute on the type. A set of iterations is performed using `MPI_Pack` and `MPI_Unpack` in order to get a rough estimate of the time of runs. Using this data, we then calculate a number of iterations to time and execute those iterations. The process is repeated for the MPITypes approach, replacing `MPI_Pack` and `MPI_Unpack` with two calls to `MPIT_Type_memcpy`. Finally, manual packing and unpacking routines (hand-coded loops) are timed. Table 1 summarizes the results of this testing.

The *Contig* test operates on contiguous data using `MPI_FLOAT` and `MPI_DOUBLE` datatypes. A contiguous datatype of 1,048,576 elements is created, and a count of 1 is used. The *Vector* tests operate on a vector of 1,048,576 basic types with a stride of 2 types (i.e. accessing every other type). A count of 1 is used when calling pack/unpack routines. MPITypes performance tracks MPICH2 in these tests. Open MPI performs slightly more slowly for the vector type, indicating room for improvement in this case.

The *Indexed* set of tests use an indexed type with a fixed, regular pattern with multiple strides. Every block in the indexed type consists of a single element (of type `MPI_FLOAT` or `MPI_DOUBLE`, depending on the particular test run). There are 1,048,576 such blocks in the type. In these tests we see higher performance for the manual approach, because our manual data movement implementation

takes advantage of knowledge of these two strides, whereas this information is not recognized by the MPI implementations or MPITypes. Performance is similar for the two MPI implementations and MPITypes in these tests.

The *3D Face* tests pull entire faces off a 3D cube of elements, described in Appendix E of [9]). Element types are varied between `MPI_FLOAT` and `MPI_DOUBLE` types. The 3D cube is 256 elements on a side. Performance is virtually identical for the MPI implementations, MPITypes, and the manual copying in these tests, with the exception of the XY Face results for hand-coded loop. This is due to an optimized UNIX `memcpy()` being used in the MPI and MPITypes cases.

4.2 Parallel netCDF Data Encoding and Decoding

As described earlier, PnetCDF uses an inefficient approach when encoding or decoding data for a noncontiguous user buffer, in order to avoid the need to process MPI datatypes. A more interesting use of MPITypes would remove the need for this intermediate buffer in the PnetCDF encode/decode process. There are two components to this process: data translation and byte reordering. Data translation occurs when the data in the user's buffer is of a different type from the variable in which it is to be stored. This can happen, for example, when writing data for visualization purposes in simulations codes: data in memory in double-precision floating points is stored in single-precision format to reduce I/O demands. If the two types are the same, format translation is not necessary. The netCDF file format calls for big-endian data. If the system is a little-endian machine (e.g., Intel), then byte translation must be performed. Since our test system is an Intel system, we will perform this byte swapping.

The MPITypes `memcpy` implementation provides a convenient starting point for an implementation of this PnetCDF functionality. Through experimentation, we found that for cases where data translation is not necessary, the most efficient approach was to use the MPITypes `memcpy` function unchanged to copy data into a contiguous buffer, then perform a single pass over the buffer to swap bytes.

In the case where data translation is necessary, we constructed a new function, `MPIT_Type_netcdf_translate`, to perform the data translation as part of the copy process (and to perform byte swapping when required). The type of data being stored in netCDF is passed along in the structure of parameters to a new set of leaf functions. In the case of encoding, the leaf functions first perform data translation into the new buffer, then byte swap. This process is reversed for decoding.

The availability of the "element type" in the leaf function provides critical information for the translation process. The combination of the type of data in the user's buffer, available via the element type parameter, and the desired format of the data in the file, stored in the parameters passed to the leaf functions, defines the translation to be applied. Notice that because the position in the contiguous buffer is tracked by the leaf function, data encoding that requires a change in the size of the data is possible; we simply increment the location by the size of the data in the encoded format, rather than by the size of the data in the native format.

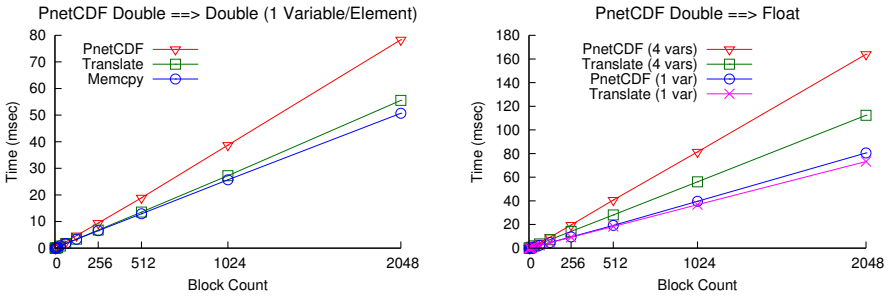

Fig. 3. Performance of PnetCDF data encode implementation for quadruply nested vector type, for the case where data only needs to be byte swapped (left) and for a double-to-float translation and byte swap (right). A block is an $8 \times 8 \times 8$ array of elements.

We use the basic data structure from the Flash astrophysics application as our test case. The Flash code is an adaptive mesh refinement application that solves fully compressible, reactive hydrodynamic equations, developed mainly for the study of nuclear flashes on neutron stars and white dwarfs [10]. The data consists of some number of 3D blocks of data. Each block consists of a $8 \times 8 \times 8$ array of elements surrounded by a guard cell region four elements deep on each side. Each element consists of 24 variables, each an `MPI_DOUBLE`. This type is representative of the most complex types we would expect to see from most applications.

In our tests we construct a quadruply nested vector type that references all local elements of one or more variables, skipping the guard cells. Figure 3 shows the results of our experiments. On the left, performance for encoding a single variable out of n blocks of the Flash type is examined. Our new approach, using `memcpy` followed by byte swapping, results in a 29% reduction in time as compared to the original PnetCDF approach. A standard MPITypes `memcpy` is shown as a reference. On the right, we examine performance for the case where data in the user buffer in double-precision floating point is converted to single precision for writing into the file. We show results for extraction of both one variable and four contiguous variables. For the case of a single variable, we see only approximately a 9% improvement over the existing approach despite a significant tuning effort. Performance for four adjacent variables is 31% faster using the new approach. This indicates that for types with very small contiguous regions, we should not expect substantial gains in directly manipulating the data as compared to moving it into a contiguous buffer. On the other hand, in a situation with memory constraints, the ability to operate in this manner with some performance improvement, while simultaneously reducing the total memory requirement, is a significant advantage.

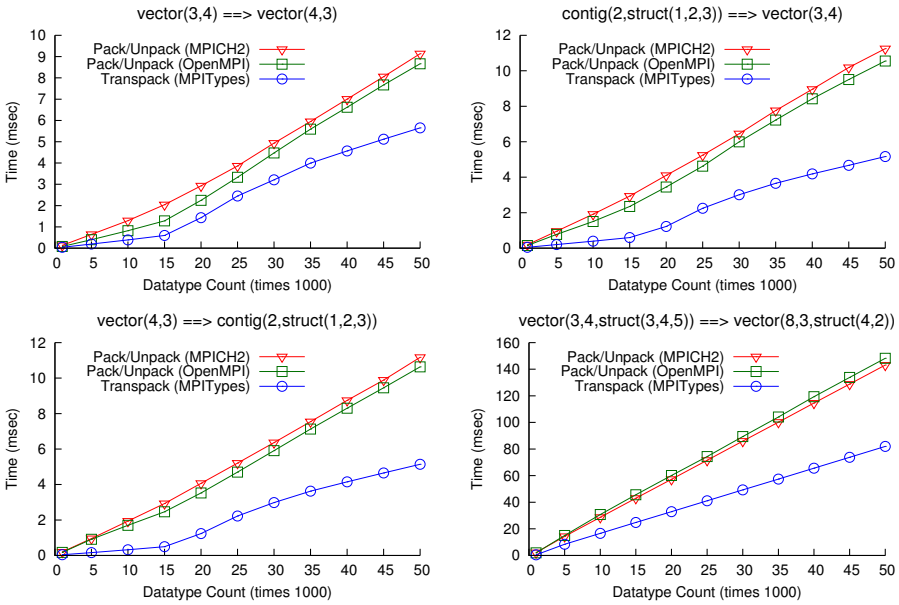

Fig. 4. Performance of template-based transpack implementation, as compared to MPICH2 and Open MPI `MPI_Pack`/`MPI_Unpack`. Time to create the MPITypes representation and to generate the template is included in the Transpack times.

4.3 Transpacking

Transforming data between datatype representations is a complex task that can be an important part of certain algorithms. For example, the data sieving implementation in ROMIO requires mapping data between the user's datatype and a portion of the file view defined on a file [11] in order to combine noncontiguous I/O operations into fewer, larger I/O accesses.

Data sieving can be thought of as an example of the *typed copy problem* [4].[3] The typed copy problem consists of moving data from one datatype representation to another, and a naïve solution to this problem is to simply pack and unpack using the two datatypes. The approach of moving the data directly from one representation to the other, without using an intermediate buffer, is known as *transpacking*. An elegant solution to transpacking has been described that relies on the generation of a new datatype representation that includes two offsets for a given element rather than one [4].

Our approach recognizes that based on the prior work in this area [4], transpacking is most often effective when applied on large counts of relatively small types

[3] Actually data sieving requires the ability to perform *partial* typed copy operations in order to limit buffer requirements.

and that in many cases the combination of types results in a pattern without any expressible regularity (i.e., the resulting combination type is an indexed type).

Working under these assumptions, and assuming identical type sizes as in the original work, we constructed an implementation that generates a template necessary to copy a single instance of each type (i.e. a flattened *input-output type*). This construction also uses nested calls to `Segment_manipulate`, but only for one instance of each type. For example, the template generated from a vector with a count of four would list the four regions described by the type. The template generated from this process is then used to copy between multiple instances of the types. The overhead of processing in this case is quite low.

We implemented a subset of the tests used in [4] to evaluate our prototype transpack implementation. We did not test cases where one or both buffers were contiguous, as these cases do not require the use of our new functionality. Figure 4 shows the results of these tests. Our template generation approach is able to exceed a factor of two performance gain for most cases. The cost to generate the template for copying is small enough that the approach is viable for even small numbers of types: the break-even point occurs before 100 type instances for all tests.

5 Conclusions and Future Work

In this paper we have presented MPITypes, a library for processing MPI datatypes and building case-specific processing functions in software using the MPI programming model. We have shown an example of the use of MPITypes for data copying, and we have described its application in PnetCDF as a tool for more efficiently performing data encoding and decoding in this library. We also discussed an advanced application, transpacking, and the nested use of MPITypes to implement this capability.

MPITypes is a portable package that relies only on standard interfaces available as part of the MPI-2 extensions, namely, datatype attributes and the envelope and contents calls. So far it has been tested and shown to work correctly on top of both MPICH2 and Open MPI, two popular implementations. We are releasing MPITypes under an open source license in the hope that it will encourage greater use of MPI datatypes as a language for describing noncontiguous data regions in parallel applications. [4] Perhaps the capabilities provided in MPITypes could be considered for incorporation into the MPI 3.0 standard, ensuring these facilities are available for each implementation.

We intend to further explore the use of MPITypes in I/O libraries, including the use of MPITypes functionality as the basis for new implementations of data sieving and two-phase optimizations in the ROMIO MPI-IO library. Augmenting MPITypes to include functionality for serializing and deserializing types, so that they may be efficiently passed between processes, seems a useful enhancement along these lines as well.

[4] See http://www.mcs.anl.gov/research/projects/mpitypes/

Acknowledgments

We would like to thank our reviewers for their helpful suggestions. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Dept. of Energy, under Contract DE-AC02-06CH11357, and in part by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy award DE-FG02-08ER25835.

References

1. Thakur, R., Gropp, W., Lusk, E.: On implementing MPI-IO portably and with high performance. In: Proceedings of the 6th Workshop on I/O in Parallel and Distributed Systems, ACM Press (May 1999) 23–32
2. Li, J., keng Liao, W., Choudhary, A., Ross, R., Thakur, R., Gropp, W., Latham, R., Siegel, A., Gallagher, B., Zingale, M.: Parallel netCDF: A high-performance scientific I/O interface. In: Proceedings of SC2003: High Performance Networking and Computing, Phoenix, AZ, IEEE Computer Society Press (November 2003)
3. Ross, R., Miller, N., Gropp, W.: Implementing fast and reusable datatype processing. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 2840 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2003) 404–413
4. Mir, F., Träff, J.: Constructing MPI input-output datatypes for efficient transpacking. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 5205 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (September 2008) 141–150
5. Rew, R., Davis, G.: The unidata netCDF: Software for scientific data access. In: Sixth International Conference on Interactive Information and Processing Systems for Meteorology, Oceanography and Hydrology. (February 1990) 33–40
6. Message Passing Interface Forum: MPI-2: Extensions to the message-passing interface (July 1997) `http://www.mpi-forum.org/docs/docs.html`.
7. Worringen, J., Träff, J.L., Ritzdorf, H.: Improving generic non-contiguous file access for MPI-IO. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Volume 2840 of Lecture Notes in Computer Science., Springer Berlin / Heidelberg (2003) 309–318
8. Träff, J., Hempel, R., Ritzdoff, H., Zimmermann, F.: Flattening on the fly: Efficient handling of MPI derived datatypes. In: Recent Advances in Parallel Virtual Machine and Message Passing Interface. Number 1697 in Lecture Notes in Computer Science, Berlin, Springer-Verlag (1999) 109–116
9. Gropp, W., Lusk, E., Skjellum, A.: Using MPI: Portable Parallel Programming with the Message-Passing Interface. MIT Press, Cambridge, MA (1994)
10. Fryxell, B., Olson, K., Ricker, P., Timmes, F.X., Zingale, M., Lamb, D.Q., MacNeice, P., Rosner, R., Tufo, H.: FLASH: An adaptive mesh hydrodynamics code for modelling astrophysical thermonuclear flashes. Astrophysical Journal Supplement **131** (2000) 273–334
11. Thakur, R., Gropp, W., Lusk, E.: Data sieving and collective I/O in ROMIO. In: Proceedings of the 7th Symposium on the Frontiers of Massively Parallel Computation, IEEE Computer Society Press (February 1999) 182–189