DIY User's Guide

Tom Peterka

February 26, 2013

1 Introduction

Computational science today requires some form of parallel analysis and visualization (henceforth simply called *analysis*) in order to extract meaning from data, and in the extreme case, such analysis needs to be carried out at the same scale as the original simulation. Even in postprocessing, parallel analysis algorithms need to scale efficiently to clusters with thousands of cores, since the core count in clusters is rapidly growing alongside that in supercomputers. Scalable, parallel analysis of data-intensive computational science relies on the decomposition of the analysis problem among a large number of distributed-memory compute nodes, the efficient data exchange among them, and data transport between compute nodes and a parallel storage system. Many analysis tasks rely on some form of global information, which in a distributed-memory setting must be communicated among nodes. Most, if not all, read and write to storage as well. These building blocks: configurable data partitioning, scalable data exchange, and efficient parallel I/O, are the main components of our library.

In today's supercomputing environment, MPI [6] is the prevalent model for distributed parallel programming and scales to the largest machines in the world, but it is a low-level interface that does not attempt to shield parallel constructs from developers. Even with a mastery of parallel programing, finding solutions that balance load, minimize data movement, and hide latency requires considerable effort. Hence, a core set of scalable data decomposition and data movement utilities is fundamental to most parallel analysis applications. These utilities are written by using MPI and provide complex operations that are difficult to scale. DIY ("**D**o-**I**t-**Y**ourself" Parallel Analysis) is a prototype library of scalable core components for the decomposition and movement of data, targeted for analysis workloads that are data-intensive and bound by I/O and communication. DIY includes

- Efficient I/O to and from parallel storage systems,
- Domain decomposition and assignment to MPI processes, and
- Local and global communication.

Potential users of DIY are computational scientists performing their own analysis, visualization researchers building new parallel analysis algorithms or parallelizing existing ones, and production tool builders and maintainers working on improving the scalability of their tools. DIY is a library that can be linked in a variety of environments, including in situ, in transit, and postprocessing analyses. DIY is written in C++ and uses the Standard Template Library, enabling the code to remain relatively small and maintainable. DIY's public API is C-style so that it can be called directly from both C and C++ applications. Fortran applications can call DIY through a shim layer of C wrapper functions that call DIY (not included).

2 Description

Figure 1 shows the overall structure of DIY and how it is used in the call stack of an application. The dependencies that DIY needs to build and run are listed below. We then discuss the data structures for

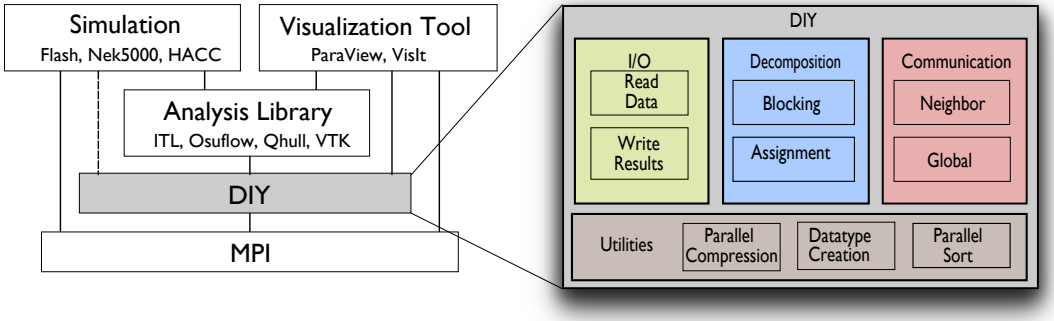

Figure 1: DIY is organized into three main modules for performing parallel I/O, block management (domain decomposition), and communication. It can be called directly from user applications or through a higher-level application-specific library. DIY makes calls to MPI, but user applications and libraries can also call MPI.

managing blocks and their neighbors and the mechanism for passing the application data structures to DIY. Each of DIY's three main modules–I/O, decomposition, and communication–are described in more detail.

2.1 Underlying Libraries

DIY depends on several lower-level tools in order to perform its work. At the same time, we do not attempt to entirely hide lower-level functions from applications; they are still free to call these tools directly. Our goal is adding useful functionality rather than encapsulating entire libraries.

MPI is mandatory. Distributed-memory message passing is a mainstream cluster and HPC parallel programming model in use today. Each processing element has a separate address space and explicitly passes messages when communicating data among address spaces. The de facto standard and implementation of this programming model is MPI, and we expect that it will remain so for the foreseeable future [1]. We recommend installing the current version of MPICH2[1], although most implementations of the MPI-2 standard will work. MPI can also be combined with finer-grained threading models in an address space [8], such as GPU or CPU threads. Currently, DIY assists in the high-level, internode MPI parallelism and leaves the finer-grained thread parallelism to the local computation.

We make optional use of the Zoltan parallel services library [4] in order to perform dynamic load balancing. Zoltan reports how the new processor assignment differs from the existing one but leaves DIY to actually perform the required data movement efficiently.

Unstructured meshes are supported through optional building with MOAB [15]. MOAB provides the ability to read, write, and query meshes for geometry, field data, and connectivity information.

Efficient, parallel I/O is often a bottleneck in analysis applications, especially those that read and write large amounts of data from and to parallel storage systems. MPI-IO [16] (part of MPI-2) serves as the foundation upon which higher-level parallel I/O libraries such as parallel netCDF [10] and parallel HDF5 [5] are built. We support reading raw binary data by using MPI-IO as well as netCDF and HDF5 formats with an efficient block-structured, two-phase I/O mechanism described in [9].

2.2 Data Structures

Today's petascale architectures limit memory at a few gigabytes per node, and all indications for exascale machines are that there will be even greater pressure to limit the memory size of a node. Therefore, not only must the performance of analysis algorithms scale with number of processors, so too must the memory

[1] http://www.mcs.anl.gov/research/projects/mpich2/

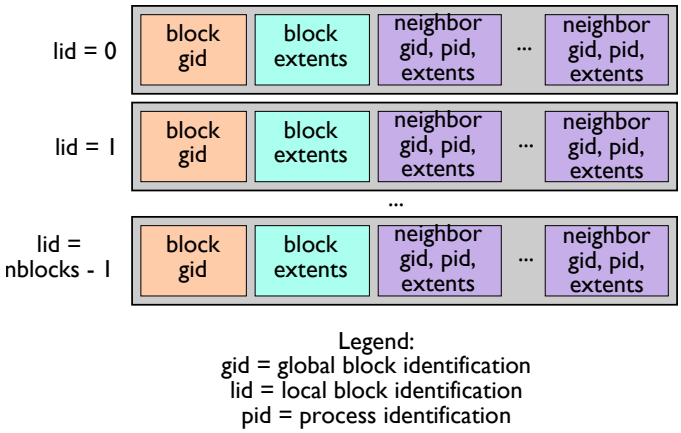

Figure 2: Data structure for one process is a list of local blocks. Each block contains bounds information and a list of neighboring blocks.

footprint. The way to achieve memory scalability is avoiding global data structures that are *O(total number of processes)* and *O(total amount of data)*.

DIY's data structures were designed with memory scalability in mind. Each process stores the data structure shown in Figure 2. There is one row for each local block that the process owns: it contains a global block identifier, block minima and maxima, and an adjacency list of neighboring blocks that share a face, edge, or vertex with the block in question. Hence, only local information about the blocks assigned to a process and their adjacent neighbors is stored at each process, minimizing memory overhead. When blocks are re-assigned to a new process for load balancing, the row in Figure 2 corresponding to the block that is being reassigned is transferred to the new owner, along with the data in the block.

The data model that DIY supports is the DIY data type. DIY data types are equivalent to MPI data types and can be used interchangeably, but DIY provides convenient functions to create custom DIY data types easier than MPI data types. Applications provide DIY with DIY data types or callback functions that create DIY data types for their custom application data structures. Mapping automatic types to an DIY data type is trivial; for example, a C `int` corresponds to `MPI_INT`. DIY provides a direct way to create more complex types such as subsets of multidimensional arrays and arbitrary data structures.

Our rationale in choosing DIY data types for our data model was generality; not only does our approach accommodate all data structures supported by the underlying programming language, but users are also free to alter their data types in order to improve performance. For example, a particular application that benefits from custom packing and unpacking during messaging can define such data types with `MPI_PACKED` or `MPI_BYTE`.

2.3 I/O

All I/O to and from storage is performed in parallel. Supported input file formats are raw, HDF5, and netCDF. We include in DIY the Block I/O Layer (BIL) [9] for block-structured input. BIL provides an abstraction for reading multifile and multivariate datasets by allowing processes to post requests for blocks and then collectively operating on the entire block set. Depending on the nature of the requests, BIL can utilize I/O bandwidth more efficiently than standard collective access. Furthermore, block-based requests which have ghost regions do not suffer from redundant I/O because data replication is performed in memory. An efficient parallel sorting utility based on the parallel sample sort algorithm [2] is also included.

Just as many analysis algorithms begin by reading data from storage, many conclude by writing their results to storage. We designed an output file model that allows complete generality in the type of data being stored. It is assumed that the analysis algorithm will assign higher-level semantic meaning to our

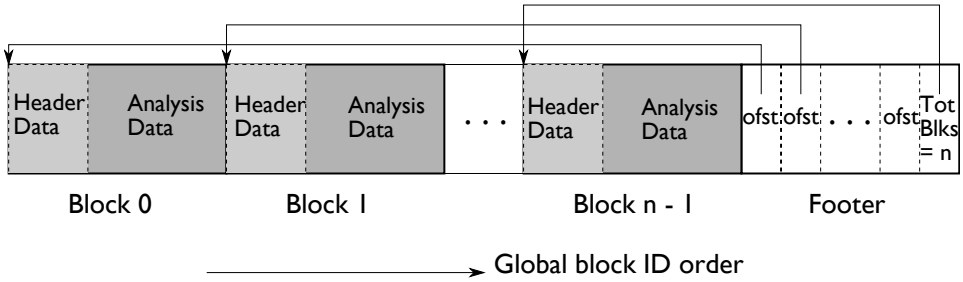

Figure 3: The output file is a list of analysis blocks containing arbitrary data structures. Among processes, output blocks need not be uniform in size or number. Each block contains an optional header that contains user-defined data such as quantities, sizes, and so forth, to assist in further processing of blocks. A footer contains indices to all of the blocks, as well as their total number. All output is contained in a single binary file.

generic output model. Figure 3 shows the output file structure, a binary format similar to the BP format of ADIOS [11]. We support any DIY data type and write a sequence of these data types, or "blocks," where an output block is the analysis output from a subdomain residing on a process. These blocks can be of arbitrary length and can be distributed among processes in any arrangement. Each block can contain an optional header with user-defined information about quantities contained in the block, and so forth. A footer with indices to the analysis blocks concludes the file. DIY also includes parallel reading functions for this format.

2.4 Domain Decomposition

Data parallel analysis problems are solved by dividing the domain into smaller pieces (blocks) and assigning collections of blocks to processes. DIY supports multiple domains, each identified by a unique domain identifier (*did*). Multiple domains are used to decompose one data set into multiple decompositions and to analyze multiple datasets simultaneously. DIY can decompose each domain and assign blocks to processes, or the user can describe an existing decomposition and block assignment to DIY for in situ analysis.

Despite the name, a block need not be "block-shaped;" rather, a block can be any subset of vertices in unstructured grids, particle data, graph nodes and edges, and so forth. Blocks are simply the units of decomposition for a given problem domain. Within a domain, blocks are identified by a local block ID (*lid*) so that the pair of identifiers (*did, lid*) are used to identify blocks local to a process. All blocks also have a global id (*gid*) that is unique across all domains.

The number of blocks is greater than or equal to the number of processes. Blocks and all later operations on them can be 2D, 3D, or 4D. For example, a 3D time-varying dataset can be divided into 4D (x, y, z, t) blocks. Data values can be of various types and lengths through the use of generic containers and templates in DIY.

Once blocks are formed, they are assigned to processes. Currently, DIY creates a default block assignment that is cyclic (round-robin), with one or more blocks per process. Each block is owned by a single process, although block replication for load balancing is an idea that we are considering. The assignment module also manages the dynamic reassignment of blocks to processes. This is used for dynamic load balancing in conjunction with the Zoltan parallel services library [4].

2.5 Communication

Because the assignment of blocks to processes is configurable, it helps to think of communication as occurring between *blocks* instead of between *processes*. Depending on the block-to-process assignment, DIY then translates block identifiers to process identifiers and packages items going to the same process into a single message.

Figure 4 shows the three communication algorithms implemented in DIY. All three diagrams demon-

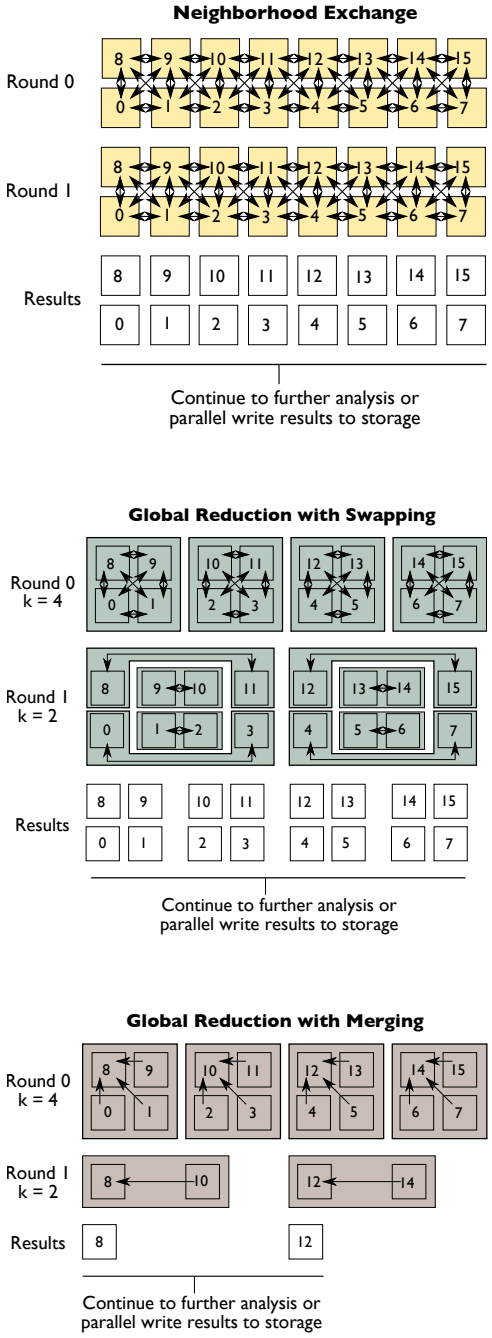

Figure 4: Three communication patterns: two rounds of (top) local nearest-neighbor communication, (center) global reduction with swapping, and (bottom) global reduction with merging among 16 blocks.

strate two rounds of information exchange among 16 blocks. The upper image exemplifies neighborhood communication; the center image is global reduction with swapping, and the bottom image shows an instance of global reduction with merging. The local neighborhood algorithm is a nonblocking message exchange among nearest-neighboring blocks that is based on the algorithm in [14]. Our focus in this paper is on the reduction algorithms, in particular on reduction with merging.

Reduction can be *complete*, where all blocks have communicated directly or indirectly with all other blocks, or *partial*, where this is not the case. The center and bottom panels of Figure 4 show examples of

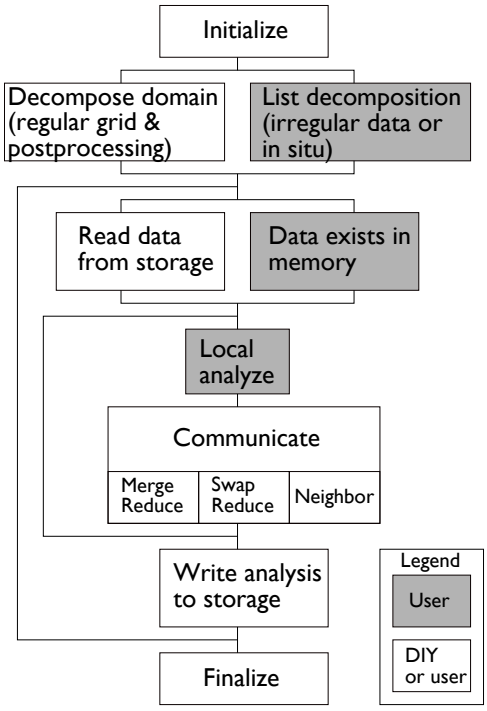

Figure 5: Typical workflow of a program using DIY. White blocks indicate functions provided by DIY, and gray blocks are provided by the user.

global reduction in two rounds with k-values of k = [4, 2]. The k-values define the group size in each round, as in [12]. Since the product of the k-values is less than the number of blocks, these are partial reductions. The shaded regions indicate communication groups in each round, and arrows indicate message transfer. In the swap algorithm this message transfer is bidirectional; while in the merge algorithm it is unidirectional.

A swapping-based reduction (center panel of Figure 4) is appropriate for analyses where the results can easily be partitioned among receiving blocks. This is the case in image compositing, where the image is split into pieces and exchanged; the Radix-k algorithm was introduced in [12] for this purpose. Other analyses result in unstructured or irregular output that cannot be arbitrarily segmented. The Morse-Smale complex, for instance, is a collection of nodes, arcs, and geometry that must remain intact in order for its structure to be preserved. A tree-based merge algorithm is needed for those cases.

In our merge algorithm (bottom panel of Figure 4), we applied the configurability of Radix-k to tree-based merging. By specifying the number of rounds and the k-values in each round, the size of communicating groups can be adjusted to maximize the bisection bandwidth of the network without causing network contention. Different k-values can be selected based on the underlying architecture and the size of the blocks being transmitted. The ability to select between complete merging and degrees of partial merging is a natural outcome of this configurable scheme. Another feature of our implementation is the use of nonblocking messaging in order to overlap the merge computation as much as possible with the merge communication.

3 Example Usage

Numerous examples are available in the DIY distribution, all following the general program flow illustrated in Figure 5. In this section, we demonstrate in more detail how two of these examples were used to generate the results published in [13] and shown in 6: particle tracing for vector flow visualization and the computation of the Morse-Smale complex for topological analysis. These examples are derived from the analysis of fuel jet combustion in the presence of an external cross-flow [7] generated by the S3D [3] simulation code.

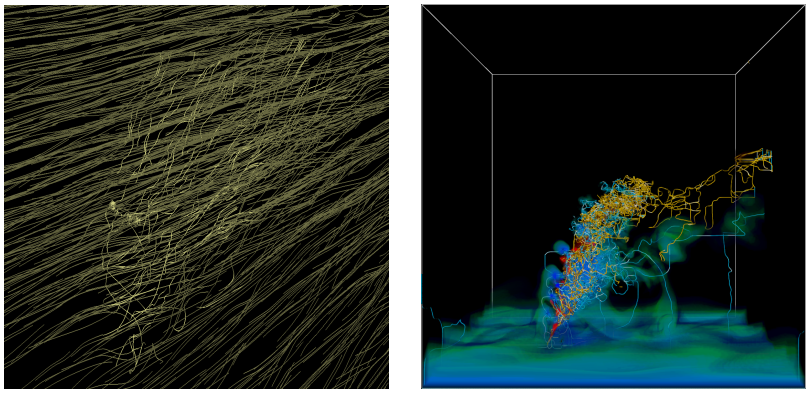

Figure 6: Two analysis techniques for locating the turbulent regions in flame stabilization. Streamlines in the top image are primarily in the x-direction, except for the turbulent region with velocity in the y and z directions. The same region is identified in the Morse-Smale complex in the right image.

Both particle tracing and computation of the Morse-Smale complex begin by subdividing the domain into blocks and assigning blocks to processes. This is accomplished by using DIY's initialization and decomposition functions:

```
DIY_Init(dim, data_size, num_threads, MPI_COMM_WORLD);
int did = DIY_Decompose(ROUND_ROBIN_ORDER, tot_blocks, &nblocks, share_face,
  ghost, given);
```

`DIY_Init` takes as input the dimensionality, data size, and number of threads allowed for DIY to use. `DIY_Decompose` takes information about the decomposition pattern, overlap between blocks and any constraints on the decomposition. It then decomposes the domain accordingly and outputs the number of local blocks assigned to the current process. Returns the domain ID. For existing decompositions (such as in situ analysis or decompositions taken from another data model), see `DIY_Decomposed` in the API reference.

The dataset is then read in parallel from storage. Each process posts read requests for its local blocks, and then the blocks are all read collectively from disk.

```
for (i = 0; i < nblocks; i++) {
  DIY_Block_starts_sizes(did, i, min, size);
  DIY_Read_add_data_raw(min, size, infile,
  datatype, data);
}
DIY_Read_data_all();
```

Each process then does its local analysis. This is a fourth-order Runge-Kutta integration in the case of particle tracing, and the computation of a discrete gradient field and subsequent complex construction for the Morse-Smale complex example. This custom analysis is provided by the user, in keeping with the DIY philosophy.

The communication for parallel particle tracing is a nearest-neighbor exchange enacted by the following calls.

```
for (i = 0; i < nblocks; i++) {
  DIY_Enqueue_item_all(did, i, item, hdr, item_size, transform_item_func);
}
```

```
DIY_Exchange_neighbors(did, items, wait_factor, item_datatype_func);
...
DIY_Flush_neighbors(did, items, item_datatype_func);
```

Items are enqueued individually and then exchanged as a group. The received items are returned after the exchange. This pattern may be repeated for multiple rounds if desired, and any remaining communication after all rounds are completed is flushed.

In Morse-Smale analysis, the communication is a global merge and is initiated as follows.

```
DIY_Merge_blocks(did, blocks, hdrs, num_rounds, k_values, &merge_func,
  &create_item_func, &destroy_item_func, &datatype_func, &num_out_blocks);
```

The input arguments include the local data, number of merge rounds, k-values in each round, and callback functions to perform the merge operation and to create the output item and datatype. The merged output blocks are returned.

Both applications then write their analysis results in parallel to storage:

```
DIY_Write_open_all(did, outfile);
DIY_Write_blocks_all(did, out_blocks, num_out_blocks, hdrs, datatype);
DIY_Write_close_all(did);
```

4 Creating Data Types

Analysis algorithms need to read, write, and communicate any data structure that could be defined in the user's programming language, and such data structures are described to DIY as DIY data types. DIY data types are aliases for MPI data types, which mirror all of the built-in and user-defined data objects in C/C++. The user can construct and use MPI data types anywhere that DIY data types are required, but DIY provides a simplified interface and utilities to assist in creating data types that are easier to use than the MPI equivalents. In the examples below and anywhere in DIY, data types can be constructed recursively from simpler ones, all the way down to the built-in types listed below. Thus, arbitrarily complex types can be constructed from a sequence of simpler constructions, and a base type in a data type construction is not limited to the built-in types.

DIY provides eight built-in types. These types can be used for either signed or unsigned values, as long as the size matches. The use of long and short types is not recommended if code is to be portable across different architectures, since the size of these types can vary in C and C++.

```
extern DIY_Datatype DIY_BYTE;        /* 1 byte */
extern DIY_Datatype DIY_SHORT;       /* 2 bytes */
extern DIY_Datatype DIY_INT;         /* 4 bytes */
extern DIY_Datatype DIY_LONG;        /* 4 bytes */
extern DIY_Datatype DIY_LONG_LONG;   /* 8 bytes */
extern DIY_Datatype DIY_FLOAT;       /* 4 bytes */
extern DIY_Datatype DIY_DOUBLE;      /* 8 bytes */
extern DIY_Datatype DIY_LONG_DOUBLE; /* 16 bytes */
```

In addition to the built-in types, DIY provides three constructors for more complex data types: vectors, subarrays, and structures. The following examples illustrate their use.

4.1 Example 1: Simple Array

Given the following C/C++ definition:

```
float[4] pt;
```

The data type would be constructed as:

```
DIY_Create_vector_datatype(4, 1, DIY_FLOAT, &type);
```

The first argument is the number of array elements. The second argument is the stride, or the distance between starting positions of adjacent array elements, measured in units of the third argument, the base data type of the array elements. A stride of 1, the most common case, is used for adjacent elements. The fourth argument is a pointer to the resulting data type.

4.2 Example 2: Subarray

Often a subset of a multidimensional array is the desired data type, where the subset is a contiguous region in volumetric space, but usually not contiguous in memory. DIY provides a subarray constructor for this purpose.

Assume the following data structure:

```
float [30][100] density;
```

In this example we will include the two left columns of the array, as in the exchange of a halo region that is two cells wide (the shaded region of Figure 7).

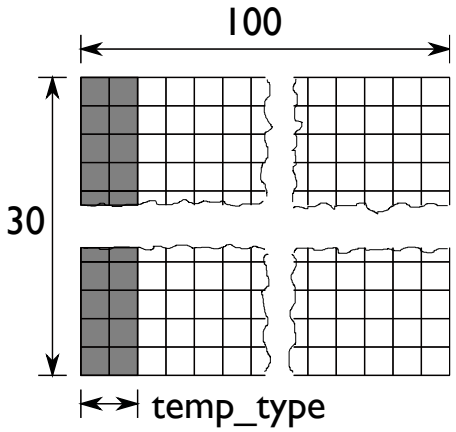

Figure 7: A halo region subset of an array

The data type is constructed as follows. Note below that the order of multidimensional arguments to DIY is always [x][y][z][t], the opposite order that the array is defined in C/C++ ([row][column]...).

```
int full_size[2] = {100, 30}; /* always [x][y]... order */
int sub_size[2] = {2, 30}; /* always [x][y]... order */
int start_pos[2] = {0, 0}; /* always [x][y]... order */
DIY_Create_subarray_datatype(2, full_size, sub_size,
  start_pos, DIY_FLOAT, &type);
```

The first argument is the number of dimensions of the array. The following two arguments are the sizes of the complete array and subarray, respectively. The fourth argument is the starting position of the subarray. The fifth argument is the base data type of the array elements, and the sixth argument is a pointer to the resulting data type.

4.3 Structures

The remaining examples all use the most general, structure data type. This datatype is created in two steps: (1) initialize a type map that describes the data type, and (2) have DIY create the data type based on the type map. These steps can be repeated to create more complex data types from simpler ones.The first step consists of creating a type map, which is an array of the following blocks:

```
/* typemap block for creating custom datatypes */
struct map_block_t {
  DIY_Datatype base_type; /* existing datatype used to create this one */
  int disp_type;          /* diplacement is relative OFST or
                             absolute ADDR */
  int count;              /* count of each element */
  DIY_Aint disp;          /* displacements of each element in bytes
                             OFSTs are from the start of the type and
                             ADDRs are from 0x */
};
```

The second step consists of calling the following function with the type map:

```
DIY_Create_struct_datatype(base_addr, num_blocks, map, &type);
```

Where the first argument is the base address that will be added to any relative displacement in the type map. The second argument is the number of blocks in the type map. The third argument is the type map, and the fourth argument is a pointer to the resulting data type.

The first argument in the above function declaration and the fourth field in the type map declaration is a DIY address (`DIY_Aint`), rather than customary native language address-of syntax. Addresses must be converted from native language references to DIY address integers when an `DIY_Aint` type is required. Do not type-cast this conversion; use the following DIY function instead.

```
/* addr: C/C++ pointer or address
   returns: equivalent DIY address integer */
DIY_Aint *DIY_Addr(void *addr) {
```

Data types that are no longer needed should be destroyed. A datatype that is created solely for the purpose of creating another higher-level data type may be destroyed once the higher-level datatype is created. Otherwise, data types used for DIY's data movement operations may be destroyed only upon completion of those operations. DIY provides the following function for data type cleanup:

```
DIY_Destroy_datatype(&type);
```

The following examples further illustrate how to create DIY structure data types.

4.4 Example 3: Simple Structure

Given the following data structure:

```
struct Particle {
 float[4] pt;
 int steps;
};
```

The data type would be constructed as:

```
DIY_Datatype type;
struct map_block_t map[] = {
 {DIY_FLOAT, OFST, 4, offsetof(struct Particle, pt)   },
 {DIY_INT,   OFST, 1, offsetof(struct Particle, steps)},
};
DIY_Create_struct_datatype(0, 2, map, &type);
```

Note that the second element of the type map blocks is OFST. This signifies that the displacement (fourth element of the type map block) is a relative offset from the start of the structure. Generally, a displacement type of OFST is accompanied by the the use of the C macro `offsetof()` in the fourth element to calculate the value of the displacement. We will see in Example 5 a case when the displacement is an absolute address instead of a relative offset.

4.5 Example 4: Nested Structure

Given the following data structures:

```
struct Particle {
 float[4] pt;
 int steps;
};

struct Cloud {
 struct Particle Particles[MAX_PARTICLES];
 int num_particles;
};
```

The DIY data type would be constructed as:

```
DIY_Datatype p_type;
struct map_block_t p_map[] = {
 {DIY_FLOAT, OFST, 4, offsetof(struct Particle, pt),  },
 {DIY_INT,   OFST, 1, offsetof(struct Particle, steps)},
};
DIY_Create_struct_datatype(0, 2, p_map, &p_type);

DIY_Datatype c_type;
struct map_block_t c_map[] = {
 {p_type,   OFST, MAX_PARTICLES,
   offsetof(struct Cloud, Particles),   },
 {DIY_INT,   OFST, 1,
   offsetof(struct Cloud, num_particles)},
};
DIY_Create_struct_datatype(0, 2, c__map, &c_type);

DIY_Destroy_datatype(&p_type);
```

4.6 Example 5: Pointers to Data

Assume the same `Particle` data structure as in Example 4, but now the `Cloud` data structure contains a pointer:

```
struct Cloud {
 struct Particle *Particles;
 int num_particles;
} my_cloud;
```

In this example, we want the data type to contain the referenced data, not the pointer itself. We use the ADDR (absolute address) displacement type instead of the OFST (relative offset) displacement type for this purpose. Also, the value of the pointer is used for the displacement, rather than the offset to the pointer element in the structure. Assuming that the `p_type` data type has already been created for the `Particle` structure as in Example 4, then the following code creates the new `c_type` data type for the `Cloud` structure:

```
DIY_Datatype c_type;
struct map_block_t c_map[] = {
 {p_type,  ADDR, my_cloud.num_particles, DIY_Addr(my_cloud.Particles)},
 {DIY_INT, OFST, 1,             offsetof(struct Cloud, num_particles)},
};
DIY_Create_struct_datatype(DIY_Addr(&my_cloud), 2, c__map, c_type);
```

If a data type containing pointer-referenced data is used repeatedly, the data type must be re-created whenever the value of the pointer changes between uses of the data type. This implies that lower-level objects (such as `Particle`) do not contain pointers if the objects are reused multiple times. A simple suggestion is to design the data structure so that lower level structures containing pointers are removed and instead expanded directly ("flattened") in the top level data structure.

4.7 Example 6: Skipping Data Fields

Given a data structure such as:

```
struct Particle {
 float[4] pt;
 char comment[256];
 int steps;
};
```

It may happen that not all fields in the original structure should be included in the desired data type. For example, assume the `comment` field should be skipped. It can be omitted from the type map, resulting in the same code as Example 3.

It may happen that a type map has a count of 0 for a block, for example, if a data element is sometimes not allocated or used. This is permitted by DIY, and `DIY_Create_struct_datatype` will automatically strip any blocks containing a count that is less than or equal to 0 from the creation of the data type. This has the same effect as if the block were skipped as in the example above, although this behavior can be controlled dynamically through the count.

Care should be taken, however, that a skipped (whether statically or dynamically) field does not appear as the first or last field in the original structure, because MPI will compute the extent of the data type incorrectly. It is usually safest to arrange the order of elements such that skipped fields are in the interior of the structure.

4.8 Specifying the Address When Creating and Using the Data Type

In the following discussion, it is important to distinguish between data type *creation* and data type *usage*. We have been discussing data type creation so far; usage refers to DIY subsequently moving the data defined by the data type.

Internally, data types encode the size and displacement of each element from a base address. After the data type is created and is ready to be used for data movement, all its displacements must refer to the same base address. Attempting to use a data type without providing a base address during either creation or use is erroneous. Vector and subarray base addresses are provided at time of usage. The base address to a struct datatype is provided upon creation. The safest way to avoid displacement errors when using struct data types is to provide a valid base address upon *creation* and then to *use* the struct with the base address `DIY_BOTTOM`. `DIY_BOTTOM` is the equivalent of 0x0, meaning no further displacement is added to data type upon usage.

5 API Reference

All functions have a C-style API and return an error code or a useful value that doubles as an error code: ¿= 0 being success and ¡ 0 indicating an error. To date, however, error codes are not implemented.

5.1 Start and End

Initializes DIY

```
int DIY_Init(int dim, int *data_size, int num_threads,
     MPI_Comm comm);

dim: number of dimensions (2, 3, or 4) (input)
data_size: data size in each dimension (only for structured data), pass NULL
  for other cases
num_threads: number of threads DIY is allowed to use (>= 1)
  ignored if DIY is built with openmp disabled (enabled by default)
comm: MPI communicator
returns: error code
```

Finalizes DIY

```
int DIY_Finalize();

returns: error code
```

5.2 Domain Decomposition

Decomposes the domain

```
int DIY_Decompose(int block_order, int glo_num_blocks, int *loc_num_blocks,
  int share_face, int *ghost, int *given);

block_order: ROUND_ROBIN_ORDER or CONTIGUOUS_ORDER numbering of
global block ids to processes (input)
glo_num__blocks: total number of blocks in the global domain (input)
  pass 0 or anything for CONTIGUOUS_ORDER (unused)
loc_num_blocks: local number of blocks on this process
```

```
share_face: whether neighboring blocks share a common face or are
  separated by a gap of one unit
  ghost: ghost layer for each dimension and side (min, max)
   each entry can be 0 (no ghost) or > 0 (this many ghost cells per side).
   Array element order:
   {x min side ghost, x max side ghost, y min side ghost, y max side ghost...}
given: constraints on the blocking entered as an array where
  0 implies no constraint in that direction and some value n > 0 is a given
  number of blocks in a given direction
  eg., {0, 0, 0, t} would result in t blocks in the 4th dimension
returns: id of domain (< 0 if error)
```

Describes the already decomposed domain

```
int DIY_Decomposed(int loc_num_blocks, int *gids, struct bb_t *bounds,
   struct ri_t **rem_ids,
   int *num_rem_ids, int **vids, int *num_vids,
   struct gb_t **neighbors, int *num_neighbors, int wrap);

loc_num_blocks: local number of blocks on this process
gids: global ids of my local blocks
bounds: block bounds (extents) of my local blocks
rem_ids: remote ids used for neighbor discovery (pass NULL if
  neighbor discovery not needed at all)
num_rem_ids: number of remote ids for each local block (pass 0 for any
  blocks that don't need neighbor discovery)
vids: local vertex ids for each local block that needs neighbor discovery (pass
  NULL if not needed at all)
num_vids: number of vids for each local block (pass 0 for any blocks that
  don't need neighbor discovery)
neighbors: neighbor lists for each of my local blocks, in lid order
  lists can be partial, containing only known information, as long as
  the rem_data lists contain enough information for DIY to discover the rest
  of the unknown neighbors. Unknown (remote) block bounds can be
  uninitialized. If wrapping is used, (see below) a neighbor direction
  must be provided for each neighbor
num_neighbors: number of neighbors for each of my local blocks, in lid order
wrap: whether wraparound neighbors are used (0 = no wraparound neighbors
  were provided, 1 = provided some wraparound neighbors)
returns: id of domain (< 0 if error)
```

Reads a decomposition from a file. Note: This is a temporary solution that only works for regular grids with share_face = 1. We eventually need a complete solution where everything needed to create a decomposition is included in the file (bounds, neighbors, etc.).

```
int DIY_Read_decomposed(char *filename, int swap_bytes,
                        int *glo_num_blocks, int *loc_num_blocks);

filename: input filename
```

```
swap_bytes: whether to swap bytes for endian conversion
glo_num__blocks: total number of blocks in the global domain (output)
loc_num_blocks: local number of blocks on this process (output)
returns: id of this domain (< 0 if error)
```

5.3 I/O

Reading scientific data, writing analysis results, and reading those results back into memory are supported.

5.3.1 Read Scientific Data

Posts a data read for a block

```
int DIY_Add_data_raw(int *block_starts, int *block_sizes, char *file_name,
     DIY_Datatype var_type, void** buffer);

block_starts: starting grid indices for minimum corner of block
block_sizes: size of block in each dimension (number of vertices)
file_name: input file name
var_type: input datatype
buffer: pointer to void * data buffer
returns: error code
```

Executes a parallel data read of all blocks

```
int DIY_Read_data_all();

note: performs an MPI_Barrier afterwards to eliminate any process
  skew before proceeding
returns: error code
```

5.3.2 Write Analysis Results

Initializes parallel writing of analysis blocks

```
int DIY_Write_open_all(int did, char *filename, int compress);

did: domain id
filename: output filename
compress: whether to compress output (0 = normal, 1 = compress)
  (1: zlib's default compression level 6 is applied blockwise)
returns: error code
```

Writes all analysis blocks in parallel with all other processes

```
int DIY_Write_blocks_all(int did, void **blocks, int num_blocks, int **hdrs,
 void (*type_func)(void*, int, int, DIY_Datatype*));

did: domain id
blocks: array of pointers to analysis blocks
num_blocks: number of blocks
hdrs: headers, one per analysis block (NULL if not used)
type_func: pointer to function that creates DIY datatype for item
returns: error code
```

Finalizes parallel writing of analysis blocks

```
int DIY_Write_close_all(int did);

did: domain id
returns: error code
```

5.3.3 Read Analysis Results

Initializes parallel reading of analysis blocks

```
int DIY_Read_open_all(int did, char *filename, int swap_bytes, int compress);

did: domain id
filename: output filename
swap_bytes: whether to swap bytes for endian conversion
  only applies to reading the headers and footer
  user must swap bytes manually for datatypes because they are custom
compress: whether to compress output (0 = normal, 1 = compress)
  (1: zlib's default compression level 6 is applied blockwise)
returns: number of local blocks to be read (< 0 if error)
```

Reads all analysis blocks in parallel with all other processes

```
int DIY_Read_blocks_all(int did, void ***blocks, int **hdrs,
void* (*create_type_func)(int, int int *,
                          DIY_Datatype*));

did: domain id
blocks: pointer to array of pointers for analysis blocks being read (output)
  DIY will allocate blocks for you
hdrs: headers, one per analysis block, allocated by caller
  (pass NULL if not used)
create_type_func: pointer to a function that allocates a block and
  creates an DIY datatype for it.
returns: error code
```

Finalizes parallel reading of analysis blocks

```
int DIY_Read_close_all(int did);

did: domain id
returns: error code
```

5.4 Communication

Point to point communication and two forms of global reduction–merge-based and swap-based–are included, as well as neighborhood communication.

5.4.1 Point to Point Asynchronous Send and Receive

Send

```
int DIY_Send(int did, int lid, void *item, int count, DIY_Datatype datatype,
     int dest_gid);

did: domain id
lid: local block id
item: item(s) to be sent
count: number of items
datatype: item datatype
dest_gid: destination global block id
returns: error code
```

Receive

```
int DIY_Recv(int did, int lid, void **items, int *count, int wait,
     DIY_Datatype datatype, int *src_gids, int *sizes);

did: domain id
lid: local block id
items: items to be received (output, array of pointers allocated by caller)
count: number of items received (output)
wait: whether to wait for one or more items to arrive (0 or 1)
datatype: item datatype
src_gids: source global block ids (output, array allocated by caller)
 only valid if MPI-3 is used, otherwise filled with -1 values
sizes: size of each item received in datatypes (not bytes)
 (output, array allocated by caller)
returns: error code
```

Flush sending and receiving (required by each process after each round or epoch of sending / receiving)

```
int DIY_Flush_send_recv(int barrier);

barrier: whether to issue a barrier (0 or 1)
 recommended if more sends / receives to follow
returns: error code
```

5.4.2 Merge-Based Global Reduction

Configurable in-place merge reduction

```
int DIY_Merge_blocks(int did, char **blocks, int **hdrs, int num_rounds,
                     int *k_values,
     void (*reduce_func)(char **, int *, int, int *),
     char *(*create_func)(int *),
     void (*destroy_func)(void *),
     void (*type_func)(void *, DIY_Datatype*, int *),
```

```
     int *num_blocks_out);

did: domain id
blocks: pointers to input/output blocks, result in in_blocks[0]
hdrs: pointers to input headers (optional, pass NULL if unnecessary)
num_rounds: number of rounds
k_values: radix (group size) for each round
reduce_func: pointer to merging or reduction function
create_func: pointer to function that creates item
destroy_func: pointer to function that destroys item
type_func: pointer to function that creates DIY datatype for item
num_blocks_out: number of output blocks (output)
side effects: allocates output items and array of pointers to them
    if not reducing in-place
returns: error code
```

5.4.3 Swap-Based Global Reduction

Configurable in-place swap reduction

```
int DIY_Swap_blocks(int did, char **blocks, int **hdrs, int num_elems,
    int num_rounds, int *k_values, int *starts, int *sizes,
    void (*reduce_func)(char **, int *, int, int),
    char *(*recv_create_func)(int *, int),
    void (*recv_destroy_func)(void *),
    void*(*send_type_func)(void*, DIY_Datatype*, int, int),
    void (*recv_type_func)(void*, DIY_Datatype*, int));

did: domain id
blocks: pointers to input/output blocks
hdrs: pointers to input headers (optional, pass NULL if unnecessary)
num_elems: number of elements in a block
num_rounds: number of rounds
k_values: radix (group size) for each round
starts: start of result in each block (output)
sizes: number of elements in result in each block (output)
  starts and sizes are allocated by the caller
reduce_func: pointer to reduction function
recv_create_func: pointer to function that creates received item
  with given number of elements (less than original item)
  part of a total number of parts in the item
recv_destroy_func: pointer to function that destroys received item
send_type_func: pointer to function that creates DIY datatype for sending
  a subset of the item starting at an element and having a given number
  of elements (less than the original item)
  returns the base address associated with the datatype
recv_type_func: pointer to function that creates DIY datatype for receiving
  a subset of the item with a given number of elements
  (less than the original item)
```

```
returns: error code
```

5.4.4 Neighborhood Exchange

Enqueues an item for sending to neighbors given their global block ids. Reflexive: sends to self block if dest_gids includes global id of self.

```
int DIY_Enqueue_item_gids(int did, int lid, void *item, int *hdr,
  int item_size, int *dest_gids, int num_gids,
  void (*TransformItem)(char *, unsigned char));

did: domain id
lid: local id of my block
item: item to be enqueued
hdr: pointer to header (or NULL)
size: size of item in bytes
dest_gids: array of gids of neighbors to send to
num_gids: the number of neighbors to send to
TransformItem: pointer to function that transforms the item before
  enqueueing to a wraparound neighbor, given the wrapping direction
  (pass NULL if wrapping is unused)
returns: error code
```

Enqueues an item for sending to a neighbor given a destination point in each neighbor. Reflexive: sends to self block if points are inside bounds of self.

```
int DIY_Enqueue_item_points(int did, int lid, void *item, int *hdr,
    int item_size, float *dest_pts, int num_dest_pts,
    void (*TransformItem)(char *, unsigned char));

did: domain id
lid: local id of my block
item: item to be enqueued
hdr: pointer to header (or NULL)
size: size of item in bytes
dest_pts: points in the destination blocks, by which the destinations can
  be identified. Points have dimension d and are listed as follows:
  eg, for dim = 4, x,y,z,t,x,y,z,t,....
num_dest_pts: number of destination points
TransformItem: pointer to function that transforms the item before
  enqueueing to a wraparound neighbor, given the wrapping direction
  (pass NULL if wrapping is unused)
returns: error code
```

Enqueues an item for sending to one or more neighbors given directions from the enumeration of possible neighbors. Each direction can be a bitwise OR of several directions. Not reflexive: no direction is defined for sending to self block.

```
int DIY_Enqueue_item_dirs(int did, int lid, void *item, int *hdr,
```

```
  int item_size, unsigned char *neigh_dirs,
  int num_neigh_dirs,
  void (*TransformItem)(char *, unsigned char));

did: domain id
lid: local id of my block
item: item to be enqueued
hdr: pointer to header (or NULL)
size: size of item in bytes
neigh_dirs: destination neighbor(s)
num_neigh_dirs: number of neighbors
TransformItem: pointer to function that transforms the item before
  enqueueing to a wraparound neighbor, given the wrapping direction
  (pass NULL if wrapping is unused)
returns: error code
```

Enqueues an item for sending to all neighbors. Not reflexive: skips sending to self block.

```
int DIY_Enqueue_item_all(int did, int lid, void *item, int *hdr, int item_size,
 void (*TransformItem)(char *, unsigned char));

did: domain id
lid: local id of my block
item: item to be enqueued
hdr: pointer to header (or NULL)
size: size of item in bytes
dest_pt: point in the destination block, by which the destination can
  be identified
TransformItem: pointer to function that transforms the item before
  enqueueing to a wraparound neighbor, given the wrapping direction
  (pass NULL if wrapping is unused)
returns: error code
```

Enqueues an item for sending to all neighbors near enough to receive it. Not reflexive: skips sending to self block.

```
int DIY_Enqueue_item_all_near(int did, int lid, void *item, int *hdr,
                              int item_size,
      float *near_pt, float near_dist,
      void (*TransformItem)(char *, unsigned char));

did: domain id
lid: local id of my block
item: item to be enqueued
hdr: pointer to header (or NULL)
size: size of item in bytes
near_pt: point near the destination block
near_dist: blocks less than or equal to near_dist of the near_pt will be
  destinations for the enqueued item . If an item is sent to more than one
```

```
  neighbor that share faces, it is also sent to the diagonal neighbor
  sharing a line or point
TransformItem: pointer to function that transforms the item before
  enqueueing to a wraparound neighbor, given the wrapping direction
  (pass NULL if wrapping is unused)
returns: error code
```

Exchanges items with all neighbors

```
int DIY_Exchange_neighbors(int did, void ***items, int *num_items, float wf,
   void (*ItemDtype)(DIY_Datatype *));

did: domain id
items: pointer to received items for each of my blocks [lid][item] (output)
num_items: number of items for each block (allocated by user)
wf: wait_factor for nonblocking communication [0.0-1.0]
  0.0 waits the minimum (1 message per round)
  1.0 waits the maximum (all messages per round)
  suggested value: 0.1
ItemDtype: pointer to user-supplied function that creates a DIY datatype
  for an item to be sent or received
side effects: allocates items and array of pointers to them
returns: error code
```

Flushes exchange with neighbors (required by each process periodically, usually after each round or epoch, or at least once at the end)

```
int DIY_Flush_neighbors(int did, void ***items, int *num_items,
void (*ItemDtype)(DIY_Datatype *));

did: domain id
items: pointer to received items for each of my blocks [lid][item] (output)
num_items: number of items for each block (allocated by user)
ItemDtype: pointer to user-supplied function that creates a DIY datatype
  for an item to be sent or received
side effects: allocates items and array of pointers to them
returns: error code
```

Finds neighbors that intersect bounds +/- extension t

```
int DIY_Bounds_intersect_neighbors(int did, int lid, struct bb_t cell_bounds,
                                   float t,
   int *num_intersect, int *gids_intersect);

did: domain id
lid: local block id
bounds: target bounds
t: additional extension on all sides of bounds
num_intersect (output) number of intersecting neighbors found
gids_intersect (output) the intersecting neighbor block gids
returns: error code
```

5.5 Utilities

Utilities for creating and destroying datatypes, compressing blocks, and miscellaneous functions are included.

5.5.1 Datatypes

Creates a vector datatype

```
int DIY_Create_vector_datatype(int num_elems, int stride,
       DIY_Datatype base_type, DIY_Datatype *type);

num_elems: number of elements in the vector
stride: number of elements between start of each element (usually 1)
base_type: data type of vector elements
type: new (output) data type
returns: error code
```

Creates a subarray datatype

```
int DIY_Create_subarray_datatype(int num_dims, int *full_size, int *sub_size,
 int *start_pos, DIY_Datatype base_type,
 DIY_Datatype *type);

num_dims: number of dimensions in the subarray
full_size: full sizes of the array ([x][y][z] order)
start_pos: starting indices of the array ([x][y][z] order)
sub_size: desired sizes of subarray ([x][y][z] order)
base_type: data type of array elements
type: new (output) data type
returns: error code
```

Creates a structure datatype

```
int DIY_Create_struct_datatype(DIY_Aint base_addr, int num_blocks,
       struct map_block_t *map, DIY_Datatype *type);

basse_addr: base address added to relative OFST displacements
num_map_blocks: number of map blocks
map: typemap with num_blocks blocks
type: new (output) datatype
returns: error code
```

Destroys a datatype

```
int DIY_Destroy_datatype(DIY_Datatype *type);

type: datatype
returns: error code
```

5.5.2 Compression

Compresses a block

```
int DIY_Compress_block(void* addr, DIY_Datatype dtype, MPI_Comm comm,
       unsigned char **comp_buf, int *comp_size);

addr: address of start of datatype
dtype: DIY datatype
comm: MPI communicator
comp_buf: pointer to compressed buffer, datatype DIY_BYTE (output)
comp_size: size of compressed buffer in bytes
side effects: allocates comp_buf, will be freed automatically the next
  time this function is called, user need not free comp_buf,
  but should use up the result before calling again, because it will disappear
returns: error code
```

Decompresses a block

```
int DIY_Decompress_block(unsigned char* in_buf, int in_size,
 unsigned char **decomp_buf, int *decomp_size);

in_buf: input block buffer (DIY_BYTE datatype)
in_size: input size in bytes
decomp_buf: decompressed buffer
decomp_size: decompressed size in bytes (output)
side effects: allocates comp_buf, will be freed automatically the next
  time this function is called, user need not free comp_buf,
  but should use up the result before calling again, because it will disappear
returns: error code
```

5.5.3 Miscellaneous

Finds block starts and sizes for blocks consisting of discrete, regular grid points

```
int DIY_Block_starts_sizes(int did, int lid, int *starts, int *sizes);

did: domain id
lid: local block id
starts: pointer to allocated array of starting block extents (output), index
  of starting grid point (not cell) in each direction
sizes: pointer to allocated array of block sizes (output), number of grid
  points (not cells) in each direction
returns: error code
```

Finds block bounds for blocks consisting of continuous spatiotemporal regions, including ghost

```
int DIY_Block_bounds(int did, int lid, struct bb_t *bounds);

did: domain id
```

```
lid: local block id
bounds; pointer to a block bounds structure (output), allocated or
  declared  by caller
returns: error code
```

Finds block bounds for blocks consisting of continuous spatiotemporal regions, excluding ghost

```
int DIY_No_ghost_block_bounds(int did, int lid, struct bb_t *bounds);

did: domain id
lid: local block id
bounds; pointer to a block bounds structure (output), allocated or
  declared  by caller
returns: error code
```

Finds the time block to which a local block belongs

```
int DIY_In_time_block(int did, int lid, int *time_block);

did: domain id
lid: local block id
time_block: time block containing lid (output)
return: error code
```

Checks whether all processes are done working

```
int DIY_Check_done_all(int done);

done: whether my local process is done (1 = done, 0 = still working)
returns: whether all processes are done (1 = all done, 0 = still working)
  (not an error code, unlike most functions)
```

Returns an DIY_Aint address given a pointer or address

```
DIY_Aint DIY_Addr(void *addr);

addr: pointer or address
returns: DIY address (not an error code, unlike most functions)
```

Returns the global block identification number (gid) given a local block number

```
int DIY_Gid(int did, int lid);

did: domain id
lid: local block id
returns: global block ID (< 0 if error)
```

Returns the total number of domains

```
int DIY_Num_dids();

returns: number of domains (< 0 if error)
```

Returns the total number of global blocks

```
int DIY_Tot_num_gids();

returns: number of blocks (< 0 if error)
```

Returns the number of global blocks in one domain

```
int DIY_Num_gids(int did);

did: domain id
returns: number of blocks (< 0 if error)
```

Returns the number of local blocks in one domain

```
int DIY_Num_lids(int did);

did: domain id
returns: number of blocks (< 0 if error)
```

Returns the starting gid in one domain, assuming gids are numbered consecutively across domains

```
int DIY_Start_gid(int did);

did: domain id
returns: starting gid (< 0 if error)
```

References

[1] P. Balaji, D. Buntinas, D. Goodell, W. Gropp, S. Kumar, E. Lusk, R. Thakur, and J. L. Träff. Mpi on a million processors. In *Proceedings of the 16th European PVM/MPI Users' Group Meeting on Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 20–30, Berlin, Heidelberg, 2009. Springer-Verlag.

[2] G. E. Blelloch, C. G. Plaxton, C. E. Leiserson, S. J. Smith, B. M. Maggs, and M. Zagha. An experimental analysis of parallel sorting algorithms, 1998.

[3] J. H. Chen, A. Choudhary, B. de Supinski, M. DeVries, E. R. Hawkes, S. Klasky, W. K. Liao, K. L. Ma, J. Mellor-Crummey, N. Podhorszki, R. Sankaran, S. Shende, and C. S. Yoo. Terascale direct numerical simulations of turbulent combustion using S3D. *Comput. Sci. Disc.*, 2:015001, 2009.

[4] K. Devine, E. Boman, R. Heapby, B. Hendrickson, and C. Vaughan. Zoltan data management service for parallel dynamic applications. *Computing in Science and Engg.*, 4(2):90–97, 2002.

[5] M. Folk, A. Cheng, and K. Yates. Hdf5: A file format and i/o library for high performance computing applications. In *Proceedings of Supercomputing 1999*, Portland, OR, 1999.

[6] A. Geist, W. Gropp, S. Huss-Lederman, A. Lumsdaine, E. Lusk, W. Saphir, and T. Skjellum. Mpi-2: Extending the message-passing interface. In *Proceedings of Euro-Par'96*, Lyon, France, 1996.

[7] R. W. Grout, A. Gruber, C. Yoo, and J. Chen. Direct numerical simulation of flame stabilization downstream of a transverse fuel jet in cross-flow. *Proceedings of the Combustion Institute*, 33:1629–1637, 2011.

[8] M. Howison, E. Bethel, and H. Childs. Mpi-hybrid parallelism for volume rendering on large, multi-core systems. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization EG PGV'10*, Norrkoping, Sweden, 2010.

[9] W. Kendall, J. Huang, T. Peterka, R. Latham, and R. Ross. Visualization Viewpoint: Towards a General I/O Layer for Parallel Visualization Applications. *To appear in IEEE Computer Graphics and Applications*, 31(6), 2011.

[10] J. Li, W.-k. Liao, A. Choudhary, R. Ross, R. Thakur, W. Gropp, R. Latham, A. Siegel, B. Gallagher, and M. Zingale. Parallel netcdf: A high-performance scientific i/o interface. In *Proceedings of Supercomputing 2003*, Phoenix, AZ, 2003.

[11] J. Lofstead, F. Zheng, S. Klasky, and K. Schwan. Adaptable, metadata rich io methods for portable high performance io. In *Proceedings of the 2009 IEEE International Symposium on Parallel&Distributed Processing*, pages 1–10, Washington, DC, USA, 2009. IEEE Computer Society.

[12] T. Peterka, D. Goodell, R. Ross, H.-W. Shen, and R. Thakur. A Configurable Algorithm for Parallel Image-Compositing Applications. In *Proceedings of SC 09*, Portland OR, 2009.

[13] T. Peterka, R. Ross, W. Kendall, A. Gyulassy, V. Pascucci, H.-W. Shen, T.-Y. Lee, and A. Chaudhuri. Scalable Parallel Building Blocks for Custom Data Analysis. In *Proceedings of the 2011 IEEE Large Data Analysis and Visualization Symposium LDAV'11*, Providence, RI, 2011.

[14] T. Peterka, R. Ross, B. Nouanesengsy, T.-Y. Lee, H.-W. Shen, W. Kendall, and J. Huang. A Study of Parallel Particle Tracing for Steady-State and Time-Varying Flow Fields. In *Proceedings of IPDPS 11*, Anchorage AK, 2011.

[15] T. Tautges, C. Ernst, C. Stimpson, R. Meyers, and K. Merkley. Moab: A mesh-oriented database. Technical Report SAND2004-1592, April 2004.

[16] R. Thakur, W. Gropp, and E. Lusk. Data sieving and collective i/o in romio. In *Proceedings of 7th Symposium on the Frontiers of Massively Parallel Computation*, pages 182–189, 1999.