Scalable Parallel Building Blocks for Custom Data Analysis

Tom Peterka,* Robert Ross Attila Gyulassy, Valerio Pascucci Wesley Kendall
Argonne National Laboratory University of Utah University of Tennessee at Knoxville
Han-Wei Shen, Teng-Yok Lee, Abon Chaudhuri
The Ohio State University

ABSTRACT

We present a set of building blocks that provide scalable data movement capability to computational scientists and visualization researchers for writing their own parallel analysis. The set includes scalable tools for domain decomposition, process assignment, parallel I/O, global reduction, and local neighborhood communication–tasks that are common across many analysis applications. The global reduction is performed with a new algorithm, described in this paper, that efficiently merges blocks of analysis results into a smaller number of larger blocks. The merging is configurable in the number of blocks that are reduced in each round, the number of rounds, and the total number of resulting blocks. We highlight the use of our library in two analysis applications: parallel streamline generation and parallel Morse-Smale topological analysis. The first case uses an existing local neighborhood communication algorithm, whereas the latter uses the new merge algorithm.

Index Terms: D.1.3 [Programming Techniques]: Concurrent Programming—Parallel Programming; H.3.4 [Information Storage and Retrieval]: Systems and Software—Distributed Systems;

1 INTRODUCTION

This paper introduces a library to assist in the development of parallel algorithms for the analysis of very large data. Computational science today requires some form of parallel analysis and visualization (henceforth simply called *analysis*) in order to extract meaning from these 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. The emphasis on data movement is the key point. As we will see, 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.

Designing and implementing scalable, parallel analysis algorithms from scratch, however, can be daunting. Parallel programming models have steep learning curves. In today's supercomputing environment, MPI [12] 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.

Our work is guided by the observation that 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. We do not provide the serial analysis per se; rather, we assume that the developer knows how to perform the sequential analysis task or that serial tools exist for this. What we provide is a library that assists developers in parallelizing such operations.

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,
- Parallel sorting,
- Domain decomposition and assignment to MPI processes, and
- Local and global communication.

Our target audience consists of 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. Our target execution modes are in a running simulation (in situ), alongside one (coprocessing), and in postprocessing after a simulation has completed. Our prototype is a library written in C++ that makes heavy use of the Standard Template Library. This enables the code to remain relatively small and maintainable, while wrappers hide high-level code constructs from languages that do not support them and allow DIY to be called from C and Fortran programs.

2 BACKGROUND

Our development of scalable data-related components is influenced by three related areas of research. We draw from our own and others' earlier work in parallel analysis algorithms in order to identify common data needs. We discuss the lower-level libraries on which DIY relies. We also compare DIY with other toolkits and parallel analysis approaches and explain how our solution differs from these and fills a set of needs not addressed elsewhere.

2.1 Survey of Analysis Algorithms

We examined the data movement requirements of a number of analysis problems. In all of these applications, data are distributed among MPI processes (we will use the term *process* for an MPI process from now on) such that each process contains a subset of the original problem. The purpose of this overview is to confirm that DIY's features cover the needs of numerous analysis algorithms, while Section 4 will spotlight two of these applications in greater detail.

In the following examples, we concentrate primarily on the communication strategy. Communication can take two forms in analysis algorithms: global reduction and local neighborhood communication. When the underlying operation is associative, global reduction is the best choice, because communication can be reordered

*e-mail: tpeterka@mcs.anl.gov

for greater efficiency. The alternative is some number of iterative neighborhood communications that relay information among processes. DIY allows arbitrary combinations of these communication steps, because analysis algorithms can exhibit a variety of communication patterns, as the following examples show.

In sort-last parallel volume rendering [5,27], each process locally renders an image of its data subdomain, and the resulting images are depth-blended together into one final image. While depth-blending is not commutative, it is associative; hence, the image composition stage is a global reduction, which can be performed efficiently with a configurable algorithm [17, 25].

Parallel particle tracing [26, 28] in flow field visualization and other applications imposes an ordering on operations that is not associative. Because a particle cannot be sent to a destination process until it has arrived at the source process, processes cannot be grouped arbitrarily. The communication pattern in particle tracing requires forming local neighborhoods and iterating over some number of rounds or until the algorithm converges.

Computing the Morse-Smale complex [14] in parallel can involve none, one, or both communication strategies. Local Morse-Smale computation introduces critical points at subdomain boundaries, which, if desired, can be removed through a global reduction. Additionally, further simplification can remove low persistence nodes by tracing arcs across block boundaries with local neighborhood communication.

Parallel feature identification can be performed with a watershed algorithm that combines connected component labeling with region growing. The region growing phase in [24] is performed with a global reduction followed by a local neighborhood exchange that iterates until it converges. An alternative approach is to perform the global reduction with a single destination process, called the visualization-accumulator in [3], thereby eliminating the need for a subsequent local neighborhood communication.

Parallel computation of the information content of a dataset can be accomplished using information-theoretic techniques [36] at the granularity of an individual vertex, a block, or the entire data domain. Global reduction is needed to compute a single entropy value for the domain, while local neighborhood communication is used to exchange ghost layers between blocks for entropy field calculations at each vertex.

As an example of iterative reduction, Parallel k-means clustering [19] requires several global reductions, one per clustering step until the algorithm converges. For every clustering step, each process computes a local k-means clustering followed by a reduction among all processes to merge local clusters into global ones.

Voronoi tessellations have been shown to be useful for identifying voids in cosmological simulations [7]. When computed in parallel, resolving the tessellation across block boundaries requires iterative local neighborhood communication until all artifacts in the tessellation have been resolved.

2.2 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 [15], 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 [9] in order to perform dynamic load balancing. It provides recursive bisection, graph, and hypergraph partitioning and allows us to use various combinations of weighting criteria in assigning blocks to processes. Zoltan reports how the new processor assignment differs from the existing one but leaves DIY to actually perform the required data movement efficiently.

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 [32] (part of MPI-2) serves as the foundation upon which higher-level parallel I/O libraries such as parallel netCDF [20] and parallel HDF5 [10] 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 [16].

2.3 Other Approaches to Parallel Analysis

VTK originated in 1993 and remains a popular platform for building visualization applications and tools [30]. Its strengths are portability and an object-oriented design for constructing and executing networks of execution pipelines. VTK also supports parallel dataflows using streaming and shared memory as described in [34]. Full-featured tools have been built using VTK, notably ParaView [35] and VisIt [6]. Both provide a variety of data models, utilities, and an extensible environment for adding new algorithms.

A different approach is taken by parallel analysis languages such as Parallel R [29] and Scout [22]. We decided that implementing DIY as a library built on a familiar Fortran/C/C++/MPI platform would enable us to reach more computational scientists, since that is the API that they most often use.

Other frameworks such as ADIOS [21] offer asynchronous "hands-off" data characterization in the background of writing out data by a simulation, without changing or rebuilding the simulation. This is in contrast to our "hands-on" philosophy of computational scientists taking direct control over analysis by embedding it directly into a simulation.

Google's MapReduce [8] model has been applied to scientific data analysis recently [31]. It features a simplified programming interface and a fixed-function analysis pipeline consisting of a local computation (map) followed by a global reduction (reduce). The DStep [18] model expands on MapReduce by adding a domain traversal stage prior to reduction. In contrast to fixed pipelines such as MapReduce and DStep, our approach requires more programming effort but allows arbitrary combinations of computation, global reduction, and local neighborhood communication.

GLEAN [33] is a framework for executing analysis in a variety of locations within a large-scale HPC ecosystem composed of compute, analysis, and I/O resources. It stages and moves data between these resources as needed. Our goal is to complement solutions such as VTK and GLEAN with DIY, rather than duplicating their capabilities. The integrated use of these tools is one of our active areas of research.

3 Method

We discuss the data structures for managing blocks and their neighbors and the mechanism for passing the application data structures to DIY. The components of the library organization are explained– I/O, block management, and communication.

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

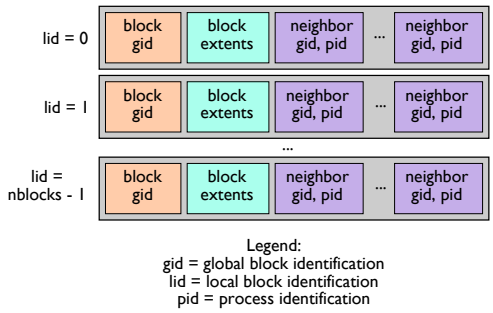

Figure 1: Data structure for one process is a list of local blocks. Each block contains bounds information and a list of neighboring blocks.

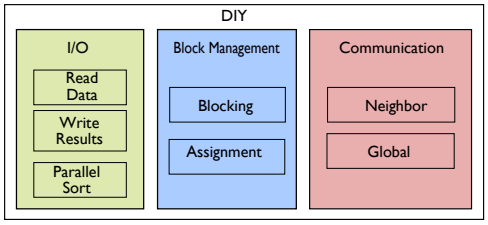

Figure 2: The library is organized into three main modules for performing parallel I/O, block management (domain decomposition), and communication. Examples in C++ and wrappers for C and Fortran applications to call these modules are also included.

3.1 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 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 1. 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 1 corresponding to the block that is being reassigned is transferred to the new owner, along with the data in the block.

Applications provide DIY with MPI data types or callback functions that create MPI data types for their custom application data structures. Mapping automatic types to an MPI data type is trivial; for example, a C `int` corresponds to `MPI_INT`. Arrays and simple data structures are also easy to map to an MPI data type; but, in general, users can have complex data structures that require effort in order to convert to corresponding MPI equivalents. Some tools for automating this task exist; AutoMap [23] is one example.

Our rationale in choosing MPI 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`.

Figure 2 shows the overall structure of DIY. Through wrappers for various languages, applications can call three different modules–I/O, block management, and communication–each described below.

3.2 I/O

All I/O to and from storage is performed in parallel. Supported input file formats are raw, HDF5, and netCDF. Currently we support a regular structured grid in our prototype, and we are actively developing adaptive and unstructured data models for future versions. We include in DIY the Block I/O Layer (BIL) [16] 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. Parallel sorting is a fundamental operation to many distributed algorithms. In many rendering applications, objects need to be sorted in depth order before rendering. Sorting is also a key operation in grouping together data items, for example, grouping spatial points before performing temporal analysis. For some clustering algorithms, the cluster assignments must be sorted and grouped before further analysis can occur.

Optimizing parallel sorting is difficult, primarily because of large communication requirements. In our benchmark tests of sorting 2.6 billion random integers (10 GB), we scaled from a sorting time of 26.46 seconds at 128 processes down to 0.36 seconds at 32,768 processes. We measured a strong scaling efficiency of 67% at 8192 processes. (Tests were conducted on Argonne's IBM Blue Gene/P *Intrepid* machine.)

Just as many analysis algorithms begin by reading data from storage, many conclude by writing their results to storage. Unlike input data models that may be grouped into a finite number of types (for example, structured grid, adaptive mesh, and so forth), the output defined by the analysis task is custom to it and impossible to predict or categorize by a general-purpose library such as DIY. Hence, 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 generic output model.

Figure 3 shows the output file structure, a binary format similar to the BP format of ADIOS [21]. We support any MPI 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.

3.3 Block Management

Block management consists of dividing the domain into smaller pieces (blocks) and assigning collections of blocks to processes. DIY can decompose the domain and assign blocks to processes, or the user can describe an existing decomposition and block assignment to DIY for in situ analysis. Although our blocks currently are regular hexahedral subdomains of a structured grid, a block is intended to be any subset of vertices in unstructured grids, particle data, and so forth, and we are working to implement such generic blocks.

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

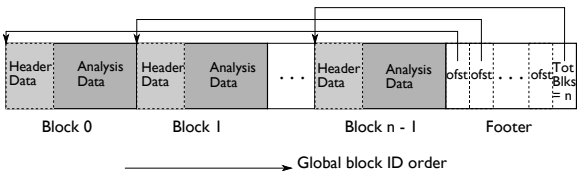

Figure 3: The output file is a list of analysis blocks containing arbitrary data structures. Among processes, output blocks need not be uniform in size or number. Each block contains an optional header that contains user-defined data such as quantities, sizes, and so forth, to assist in further processing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

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 [9].

3.4 Communication

Section 2.1 demonstrated the need for global reduction and local neighborhood communication in analysis algorithms, and we explain those communication patterns in detail here. 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 demonstrate 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 [26]. 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 global reduction in two rounds with k-values of k = [4, 2]. The k-values define the group size in each round, as in [25]. 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 [25] 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.

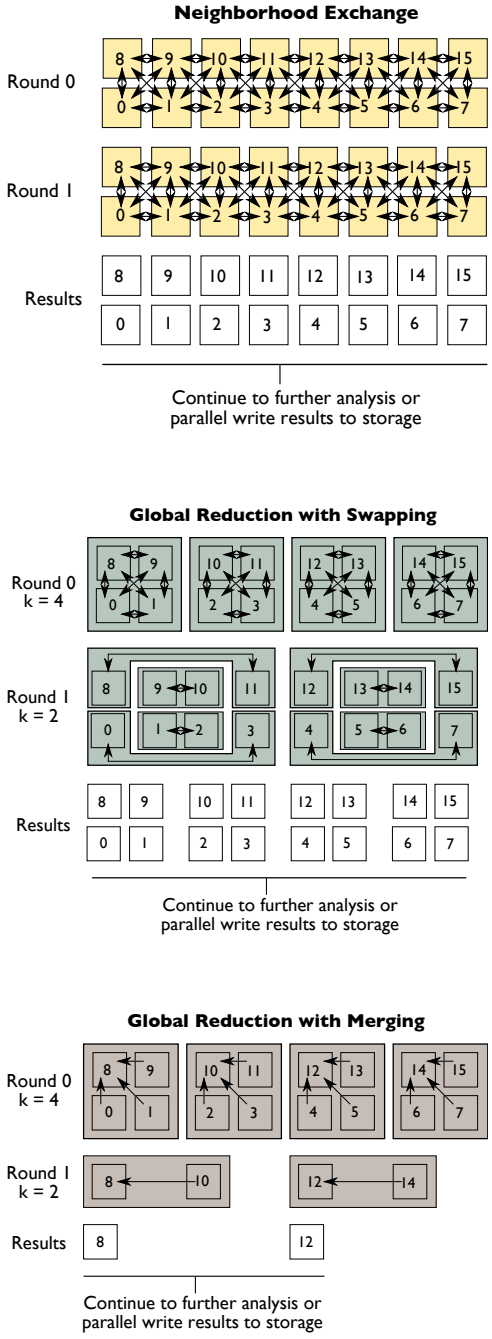

Figure 4: Three communication patterns: two rounds of (top) local nearest-neighbor communication, (center) global reduction with swapping, and (bottom) global reduction with merging among 16 blocks.

Algorithm 1 Merge algorithm

```
 1: for all rounds do
 2:     for all active blocks do
 3:         identify other blocks in same group as this block
 4:         select one block of the group to be the root block
 5:         if block is not root of this group then
 6:             post asynchronous send to the root block
 7:             mark block as inactive
 8:         else                                  ▷ root block of the group
 9:             for all other blocks in this group do
10:                 post asynchronous receive
11:             end for
12:         end if
13:     end for
14:     wait for sends/receives to complete
15:     for all root blocks do
16:         collect blocks in this group
17:         call user-defined merge operation
18:     end for
19: end for
20: return merged root blocks
```

In the merge algorithm, the pseudocode listed in Algorithm 1 is executed by each process, which owns some number of local blocks. Within each round (the outermost loop encompassing lines 1-19), two inner loops are performed. The first is over the blocks that have not already been merged into a larger block, or *active blocks* (lines 2-13). In this loop, blocks are clustered into communicating groups of size k_i, where k_i is the group size of the round i. One block is designated to be the root and takes ownership of the merged result of that group; other blocks in the group are sent to the root. The second loop (lines 15-18) iterates over the groups (identified by their root block) and computes the merged result for each group.

We analyze the communication complexity of Algorithm 1 in terms of the total number of sequential messages sent or received, since, in general, we cannot make any assumptions about the size of the actual messages being transmitted. If we begin with one block per process, there will be no more than one active block per process in any round. Because groups within the same round are executed in parallel, only the number of messages received by the root in one group per round are counted, or $\sum_{i=1}^{r} k_i - 1$, where r is the number of rounds and k_i is the k-value of the ith round.

This is an upper bound, because the use of nonblocking communication allows multiple messages in each round to overlap and to potentially complete in less than $k_i - 1$ steps. Selecting larger values of k_i, thus lowering the number of rounds needed, is advisable provided that k_i is not so large as to cause network contention. The computational complexity is also a function of the number of rounds. The actual steps within the merge operator are unknown, so we only count the number of calls to the merge operation in line 17 of Algorithm 1, which is r.

When the initial number of blocks is greater than the number of processes, several root blocks will reside on some processes and will incur additional sequential steps for both communication and computation. Depending on the initial number of blocks and processes, a distribution such as round-robin could result in an uneven distribution of root blocks to processes. In the worst case, all roots could be on the same process. We did not address this problem in this work and limited our use of the merge algorithm to one block per process.

4 Usage and Case Studies

We have successfully parallelized particle tracing for vector flow visualization and the computation of the Morse-Smale complex for topological analysis by using a common set of library components. In this section, we demonstrate how DIY supports both these tasks and present some of our findings.

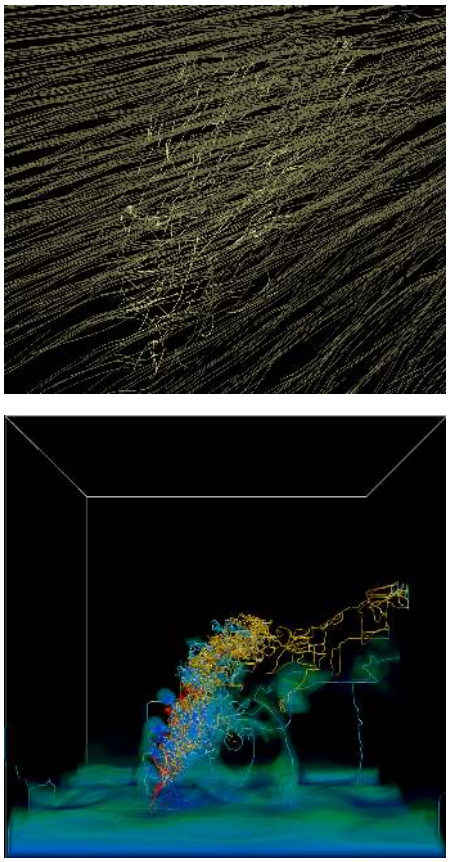

Figure 5: Two analysis techniques for locating the turbulent regions in flame stabilization. Streamlines in the top image are primarily in the x-direction, except for the turbulent region with velocity in the y and z directions. The same region is identified in the Morse-Smale complex in the lower image.

In both examples, we use the same combustion dataset: fuel jet combustion in the presence of an external cross-flow [13]. The flame is stabilized in the jet wake, and the formation of vortices at the jet edges enhances mixing. Our data were generated by the S3D [4] fully compressible Navier-Stokes flow solver and are courtesy of Ray Grout of the National Renewable Energy Laboratory and Jacqueline Chen of Sandia National Laboratories. We used one time-step at a spatial resolution of $1408 \times 1080 \times 1100$. The vector version for parallel flow visualization is 19 GB (32-bit floating-point values for the x, y, and z vector components) while the scalar version for the parallel Morse-Smale complex is only the x component of the velocity, or 6.3 GB.

Scientists such as Grout and Chen are looking at the area of intense turbulence due to the interaction of the jet and boundary layer. The region of interest can be identified by interacting with the flow visualization, navigating the view until the image in the top of Figure 5 emerges. Alternatively, this region is revealed in the topological analysis in the bottom of Figure 5. The Morse-Smale complex indicates regions of minimum x-velocity in this example, in other words, where the flow diverges from the laminar x-direction pattern.

4.1 Parallelizing Applications

We designed a C-style API that can be used in both C and C++ programs, with plans for Fortran bindings if they are needed. The following is a description of how the API is used in our two example applications. 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, ROUND_ROBIN_ORDER, tot_blocks,
  &nblocks, data_size, MPI_COMM_WORLD);
DIY_Decompose(share_face, ghost, ghost_dir,
  given);
```

`DIY_Init` takes as input the dimensionality, total number of blocks, and data size and outputs the local number of blocks assigned to this process. `DIY_Decompose` takes information about the overlap between blocks and any constraints on the decomposition and decomposes the domain accordingly.

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(i, min, size);
  DIY_Read_add_block_raw(min, size, infile,
  datatype, data);
}
DIY_Read_blocks_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.

```
DIY_Exchange_neighbors(items, wait_factor,
  recv_item_func, send_item_func);
...
DIY_Flush_neighbors(items, recv_item_func);
```

The callback functions for creating an MPI data type for sent and received items are provided as input arguments. The received items are returned as the output. Any remaining communication, after all rounds of exchanging neighboring items are completed, is flushed.

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

```
DIY_Merge_blocks(in_blocks, num_in_blocks,
  out_blocks, num_rounds, k_values,
  &merge_func, &item_func, &type_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 MPI data types for communication. The merged output blocks are returned.

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

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

4.2 Performance Results

Figure 6 shows the comparative parallel performance of particle tracing and the Morse-Smale computation. Both tests include reading the dataset from storage, computing the analysis, and writing results back to storage. Total time and strong scaling efficiency are shown for the overall end-to-end program execution. We also compare relative performance between the two analyses. Tests were run on *Intrepid*, a 557-teraflop IBM Blue Gene/P supercomputer operated by the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory.

For particle tracing, we advected 0.5 million particles in the flame stabilization dataset. The total output produced was approximately 8 GB of particle traces. A small subset of these is shown in the top of Figure 5, but the dense seeding in our test is useful for other downstream analysis, such as computing Lagrangian coherent structures [11]. We used four blocks per process and required each particle to advance at least 1,000 integration steps, or until it left the overall volume bounds. In total, approximately 500 million advection steps were executed. The data size is not large enough to scale efficiently beyond 2048 processes; but from 256 to 2,048 processes, we maintained 41% end-to-end strong scaling efficiency.

The Morse-Smale complex is computed in parallel for the x component of the flame velocity. The required steps are reading the dataset from storage, locally computing the discrete gradient field and the Morse-Smale complex, merging local complexes into a smaller number of larger ones, and writing the results out to storage. We used one block per process and reduced the number of blocks by a factor of 8 during the merging, before writing the complexes to disk. The output complexes were relatively sparse, as the bottom of Figure 5 shows, just over 100 MB. Approximately 80% end-to-end strong scaling efficiency was attained from 256 to 4,096 processes.

The Morse-Smale analysis scales better than particle tracing for several reasons. Global reduction often involves fewer communication phases than nearest-neighbor exchange because the reduction algorithm runs in $O(log_k)$ number of rounds. Morse-Smale also produces a sparse output in this example, targeting the regions of minimum x-component velocity. Hence, the output size of the Morse-Smale complex is 80 times smaller than the output of the particle traces. Particle tracing generates a dense output, but the majority of streamlines travel uniformly in the x-direction and mask the region of interest, whereas the Morse-Smale complex uncovers precisely this region while culling away areas of uniform flow.

Another reason for Morse-Smale's better scalability is evident in Figure 7. The distribution of time in this graph shows that this particular test of the Morse-Smale complex is compute-bound. Since the computational part is strictly local, it scales easily. Figure 7 also shows that other components of the overall time such as input and output I/O are beginning to take a larger percentage of the total time as the number of processes grows. We have also seen this trend continuing at larger scales with larger test data.

The merge phase remains at under 3% of the total time over this entire test, which also contributes to the overall efficiency. It should be noted, however, that this is a partial merge of one round, and more rounds or a complete merge would require a larger fraction of the total time. In this test, however, one round was sufficient to merge the output to a manageable size and to extract the features shown in Figure 5.

5 Summary

We presented a core set of building blocks for constructing parallel analysis applications, in particular, for scalable data movement. These building blocks are implemented in a prototype library called DIY that allows computational and computer scientists to focus on writing custom analysis while leaving domain decomposition, process assignment, I/O, and communication to DIY. After surveying

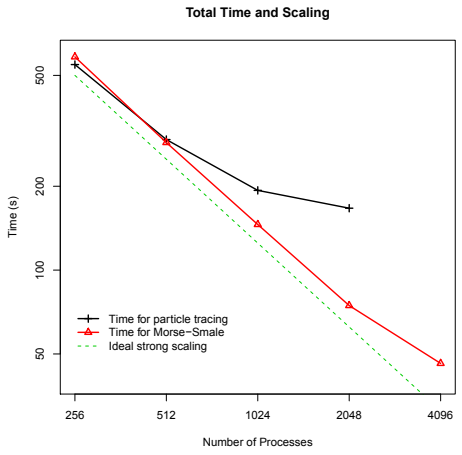

Figure 6: Performance of parallel particle tracing and parallel Morse-Smale analysis is plotted in log-log scale. The total time includes reading the dataset from storage, computing the analysis, and writing results to storage.

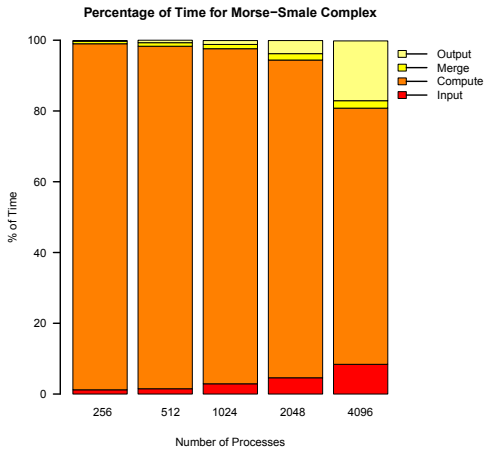

Figure 7: Percentage of time spent in each component of the Morse-Smale analysis.

the data movement needs of several analysis tasks, we demonstrated the use of our library in two of those applications: parallel particle tracing and parallel topology generation. Our solution enables the writing of different parallel analyses from the same building blocks. This presents scientists with different views of the same data and enables comparison between different techniques.

5.1 Discussion

Tools for data movement are necessary for highly-scalable analysis applications. Our solution is a departure from other approaches to analysis that include full-featured toolkits, analysis languages, asynchronous background characterization, and fixed-function pipelines. We focus on the central bottleneck in data analysis: presenting processes with the data they need in order to do their work. This is a fundamental issue that all parallel problems face as they scale up; by targeting parallel data analysis, we concentrate on the specific ways that analysis algorithms access data: through parallel I/O, local neighborhood communication, and global reduction.

A fundamental design decision is what data types or data models to support. One outcome of our examination of analysis algorithms was the need to read, write, and communicate any data structure that could be defined in C/C++/Fortran. Hence, the design of DIY defines data in terms of MPI data types, which mirror all of the built-in and user-defined language constructs. For example, our particle tracer defines a particle as the following C/C++ struct:

```
struct Particle {
  float[4] pt; // particle position
  int steps; // # of steps thus far
};
```

The consequence of this design decision is that the programmer must generate corresponding MPI data types. In the above example, this required 11 lines of C++ code:

```
MPI_Datatype types[2];
int lengths[2];
MPI_Aint displs[2];
types[0] = MPI_FLOAT;
displs[0] = offsetof(struct Item, pt);
lengths[0] = 4;
types[1] = MPI_INT;
displs[1] = offsetof(struct Item, steps);
lengths[1] = 1;
MPI_Datatype *dtype = new MPI_Datatype;
MPI_Type_create_struct(2, lengths,
  displs, types, dtype);
```

The Morse-Smale data type is considerably more complicated. It is a hierarchical organization of nodes, arcs, and geometry. The structures comprise 35 total fields, some containing pointers to dynamically allocated memory. Constructing the MPI data type for the Morse-Smale complex was by no means trivial and required writing approximately 130 lines of C++ code.

5.2 Future Work

We recognize the need for and welcome additional work in lightweight analysis infrastructures such as DIY. Our plans include applying data movement techniques to the other analyses in our survey. We will add domain decomposition for unstructured and adaptive grids to our library. Furthermore, we will investigate whether support for a hybrid parallelization model consisting of message passing between nodes and shared-memory threading inside nodes should be supported by DIY. Currently, we leave it to the application to manage thread parallelism as part of its local computation.

We continue to re-evaluate support for higher-level data models in DIY. Currently the application translates its data model into a low-level data type that DIY understands. This is in keeping with our design philosophy of allowing the application to use any in-memory or on-disk data model, without restriction. This is also efficient in run time and memory space because it minimizes data copying. To make this solution more usable, however, we encourage a renewed effort in the automated generation of MPI data types from source-code data structures or the development of intermediate data description tools to bridge the gap between application data models and DIY data types.

Acknowledgements

We gratefully acknowledge the use of the resources of the Argonne Leadership Computing Facility at Argonne National Laboratory. This work was supported by the Office of Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contract DE-AC02-06CH11357. Work is also supported by DOE with agreement No. DE-FC02-06ER25777.

We thank Ray Grout of the National Renewable Energy Laboratory and Jackie Chen of Sandia National Laboratories for providing the dataset and description of the case study used in this paper.

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. Chen, D. Silver, and L. Jiang. The feature tree: Visualizing feature tracking in distributed amr datasets. In *Proceedings of the 2003 IEEE Symposium on Parallel and Large-Data Visualization and Graphics*, PVG '03, pages 14–, Washington, DC, USA, 2003. IEEE Computer Society.

[4] 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.

[5] H. Childs, D. Pugmire, S. Ahern, B. Whitlock, M. Howison, Prabhat, G. H. Weber, and E. W. Bethel. Extreme scaling of production visualization software on diverse architectures. *IEEE Comput. Graph. Appl.*, 30(3):22–31, 2010.

[6] H. R. Childs, E. S. Brugger, K. S. Bonnell, J. S. Meredith, M. C. Miller, B. J. Whitlock, and N. L. Max. A contract based system for large data visualization. In *Proceedings of IEEE Visualization 2005*, pages 190–198, Minneapolis, MN, 2005.

[7] J. M. Colberg, F. Pearce, C. Foster, E. Platen, R. Brunino, M. Neyrinck, S. Basilakos, A. Fairall, H. Feldman, S. Gottloeber, O. Hahn, F. Hoyle, V. Mueller, L. Nelson, M. Plionis, C. Porciaini, S. Shandarin, M. S. Vogeley, and R. van de Weygaert. The aspen–amsterdam void finder comparison project. Technical Report arXiv:0803.0918, Mar 2008.

[8] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[9] 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.

[10] 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.

[11] C. Garth, F. Gerhardt, X. Tricoche, and H. Hagen. Efficient computation and visualization of coherent structures in fluid flow applications. *IEEE Transactions on Visualization and Computer Graphics*, 13(6):1464–1471, 2007.

[12] 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.

[13] 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.

[14] A. Gyulassy, P.-T. Bremer, B. Hamann, and V. Pascucci. A practical approach to morse-smale complex computation: Scalability and generality. *IEEE Transactions on Visualization and Computer Graphics*, 14(6):1619–1626, 2008.

[15] 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.

[16] 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.

[17] W. Kendall, T. Peterka, J. Huang, H.-W. Shen, and R. Ross. Accelerating and benchmarking radix-k image compositing at large scale. In *Proceedings of Eurographics Symposium on Parallel Graphics and Visualization EG PGV'10*, Norrkoping, Sweden, 2010.

[18] W. Kendall, J. Wang, M. Allen, T. Peterka, J. Huang, and D. Erickson. Simplified parallel domain traversal. In *Proceedings of SC11*, Seattle, WA, 2011.

[19] J. Kumar, R. T. Mills, F. M. Hoffman, and W. W. Hargrove. Parallel k-means cluster for quantitative ecoregion delinieation using large data sets. *Procedia Computer Science*, 4:1602–1611, 2011.

[20] 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.

[21] 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.

[22] P. McCormick, J. Inman, J. Ahrens, J. Mohd-Yusof, G. Roth, and S. Cummins. Scout: a data-parallel programming language for graphics processors. *Parallel Comput.*, 33(10-11):648–662, 2007.

[23] M. Michel and J. E. Devaney. A generalized approach for transferring data-types with arbitrary communication libraries. In *Proceedings of the Seventh International Conference on Parallel and Distributed Systems: Workshops*, ICPADS '00, pages 83–, Washington, DC, USA, 2000. IEEE Computer Society.

[24] A. N. Moga and M. Gabbouj. Parallel image component labeling with watershed transformation. *IEEE Trans. Pattern Anal. Mach. Intell.*, 19:441–450, May 1997.

[25] 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.

[26] T. Peterka, R. Ross, B. Nouanesengsey, 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.

[27] T. Peterka, H. Yu, R. Ross, K.-L. Ma, and R. Latham. End-to-end study of parallel volume rendering on the ibm blue gene/p. In *Proceedings of ICPP 09*, Vienna, Austria, 2009.

[28] D. Pugmire, H. Childs, C. Garth, S. Ahern, and G. H. Weber. Scalable computation of streamlines on very large datasets. In *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*, pages 1–12, New York, NY, 2009. ACM.

[29] N. Samatova, M. Branstetter, A. Ganguly, R. Hettich, S. Khan, G. Kora, J. Li, X. Ma, C. Pan, A. Shoshani, and S. Yoginath. High performance statistical computing with parallel r: Applications to biology and climate modeling. In *SciDAC 2006*, pages 505–509, Denver, CO, 2006.

[30] W. J. Schroeder, K. M. Martin, and W. E. Lorensen. The design and implementation of an object-oriented toolkit for 3d graphics and visualization. In *Proceedings of the 7th conference on Visualization '96*, VIS '96, pages 93–ff., Los Alamitos, CA, USA, 1996. IEEE Computer Society Press.

[31] J. A. Stuart, C.-K. Chen, K.-L. Ma, and J. D. Owens. Multi-gpu volume rendering using mapreduce. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 841–848, New York, NY, USA, 2010. ACM.

[32] 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.

[33] V. Vishwanath, M. Hereld, V. Morozov, and M. E. Papka. Topology-aware i/o acceleration for ibm blue gene/p supercomputing applications. In *Proceedings of SC11*, Seattle, WA, 2011.

[34] H. T. Vo, D. K. Osmari, B. Summa, J. L. D. Comba, V. Pascucci, and C. T. Silva. Streaming-enabled parallel dataflow architecture for multicore systems. *Computer Graphics Forum*, 29(3):1073–1082, 2010.

[35] B. Wylie, C. Pavlakos, V. Lewis, and K. Moreland. Scalable rendering on pc clusters. *IEEE Computer Graphics and Applications*, 21(4):62–69, 2001.

[36] L. Xu, T.-Y. Lee, and H.-W. Shen. An information-theoretic framework for flow visualization. *IEEE Transactions on Visualization and Computer Graphics*, 16:1216–1224, November 2010.