Block-Parallel Data Analysis with DIY2

Dmitriy Morozov* Tom Peterka†

May 11, 2016

Abstract

DIY2 is a programming model and runtime for block-parallel analytics on distributed-memory machines. Its main abstraction is block-structured data parallelism: data are decomposed into blocks; blocks are assigned to processing elements (processes or threads); computation is described as iterations over these blocks, and communication between blocks is defined by reusable patterns. By expressing computation in this general form, the DIY2 runtime is free to optimize the movement of blocks between slow and fast memories (disk and flash vs. DRAM) and to concurrently execute blocks residing in memory with multiple threads. This enables the same program to execute in-core, out-of-core, serial, parallel, single-threaded, multithreaded, or combinations thereof. This paper describes the implementation of the main features of the DIY2 programming model and optimizations to improve performance. DIY2 is evaluated on benchmark test cases to establish baseline performance for several common patterns and on larger complete analysis codes running on large-scale HPC machines.

*Lawrence Berkeley National Laboratory, dmitriy@mrzv.org
†Argonne National Laboratory, tpeterka@mcs.anl.gov

1 Introduction

The rapid growth of computing and sensing capabilities is generating enormous amounts of scientific data. Parallelism can reduce the time required to analyze these data, and distributed memory allows datasets larger than even the largest-memory nodes to be accommodated. The most familiar parallel computing model in distributed-memory environments is arguably data-parallel domain-decomposed message passing. In other words, divide the input data into subdomains, assign subdomains to processors, and communicate between processors with messages. Complicating data-intensive analysis, however, is the fact that it occurs in multiple environments ranging from supercomputers to smaller clusters and clouds to scientists' workstations and laptops. Hence, we need to develop portable analysis codes that are highly scalable on HPC architectures while remaining accessible on smaller machines with far fewer cores and memory capacity.

In HPC, the main challenge is that architectures evolve rapidly. Locality and the cost of data movement dominate energy efficiency. Supercomputers are already being designed with deeper memory/storage hierarchies and nonvolatile memory (NVM) that can be used as burst buffers or extended memory near to compute nodes with higher bandwidth and lower latency than traditional storage. For example, burst buffers already exist in small prototype instances, and next generation of supercomputers will include them in production. Additional levels of fast/slow memory [2] and near/far disks create opportunities while complicating algorithm design. A related issue is that the number of cores is rapidly increasing; hundreds per node is now the norm. Many-core architectures such as Intel Knight's Landing offer increased possibilities for parallelizing the part of the problem that is currently in core.

Traditional data-parallel codes are written in MPI [15]. Designed over twenty years ago for parallel computational science in Fortran and C, MPI is an ideal runtime for executing distributed communication, but in our experience the level of abstraction is too low for productive programming of data analytics. Higher level data-parallel models are needed that promote modularity and reuse of frequently recurring design patterns. Modern, convenient design combined with the proven performance of MPI can be attained by building libraries on top of MPI. We have found that to enable scalable data analytics in a diverse architectural landscape, a library that seamlessly integrates in-core parallel processing (both threading and message passing) with data migration between main memory and other levels of the memory/storage hierarchy is necessary.

DIY2 is our solution to this problem. DIY2 is a programming model and runtime that allows the same program to execute distributed-memory parallel and out-of-core data analysis. The integration of these two previously separate programming models is particularly useful for scientists who wish to do some of the analysis of large data sets from scientific computations in situ on a supercomputer and continue to do further processing on a smaller parallel cluster or out-of-core on a workstation. The need to easily switch between in-core, out-of-core, and mixed regimes consisting of processing some fraction of blocks in-core (in parallel) while others reside in NVM or storage is key to performance portability.

Even for applications that are strictly HPC, the ability to execute out-of-core becomes desirable when running an analysis code in situ with a simulation. While the memory size per core in next-generation supercomputers will remain constant or will slightly decrease, the collocation of simulation with analysis on the same node accentuates the limits of memory: the simulation would like to use as much of the memory as it can, and so would the analysis code. Analysis codes are often memory-bound; they may aggregate global data from numerous subdomains, or the task itself may require a large amount of memory to compute an intermediate result. For example, to convert particles to a continuous density field, one may want to compute a Voronoi tessellation first. The original particles require tens of bytes per particle while the tessellation uses hundreds of bytes per particle [24].

The key to our programming model is structuring the analysis program into data-parallel blocks. Blocks are the units of shared- and distributed-memory parallel computation, communication, and migration in the memory/storage hierarchy. Blocks and their message queues are mapped onto processes and placed in memory/storage by the DIY2 runtime. Building on the block abstraction, communication patterns and other algorithms can be developed once and reused. Decomposing a problem in terms of block-parallelism (instead of process-parallelism) enables migrating blocks during the program execution between different locations in the hardware. This is the main idea that lets us integrate in- and out-of-core programming in the same model and change modes without touching the source code. Until now, despite similarity in their design,

implementing the two types of algorithms required following very different programming models; converting between in- and out-of-core implementations was tantamount to rewriting the code. The same is true for multithreading. Block-parallelism enables performing the same operations on a block in one compute node, processor, core, or thread. Block-parallelism also makes debugging easier: the same program, with the same number of blocks, can be run on a single process for debugging.

The contributions of this paper are a set of high-level programming abstractions—block decomposition, block execution in processes and threads, communication patterns over blocks, parallel I/O—and high-level algorithms built on these abstractions. The programming model is BSP-style alternation of compute-communicate phases, formalized by DIY2's API. Numerous open-source scientific data analysis applications have already been released, and three of these are featured in our evaluation here. This paper explains the design of DIY2, describes how to write block-structured programs to take advantage of these concepts, and gives an experimental evaluation of DIY2's performance in a suite of mini applications and full data analysis codes. Section 2 places our solution in the context of other programming models. Section 3 explains the design of DIY2. Section 4 shows results of experiments using DIY2, on smaller benchmark applications and on complete data analysis codes.

2 Related work

Our review of relevant work on data-parallel computing includes block-based models, out-of-core algorithms, and other programming models such as those based on MapReduce.

2.1 Data parallelism and block-structured abstractions

Many simulation [6, 14, 17], visualization, and analysis frameworks [1, 7, 33, 38] are data-parallel, meaning each process executes the same program on a different part of the data. There is, however, a subtle but important difference between those parallel models and ours. The distinction concerns the difference between the data subdomain (which we call a block) and the processing element (be it a node, core, or thread) operating on the data (which we generically call a process). In those examples and many others, processes and blocks are equivalent; there is a one-to-one mapping between them, and data-parallelism is actually process-parallelism. In MPI, for example, messages are sent between process ranks, not blocks.

True data parallelism is decomposition of the global data domain into blocks first and a mapping of blocks onto processes second. The mapping need not be one-to-one. From the programmer's standpoint, operations are expressed on blocks, and all communication is expressed as information flow between blocks (not processes). It is the job of the runtime to map those instructions to code running on a process and messages exchanged between processes. The starting point of our work is DIY1 [32, 34], a C library that structures data into blocks. It expects the computation to be organized in a bulk-synchronous pattern, but does not enforce this structure through programming convention. The users are expected to iterate over the blocks one by one, using a for-loop, but nothing forces them to do so. Accordingly, DIY1 is only able to provide helper functions to simplify various communication patterns. It lacks threading and out-of-core support.

In contrast, DIY2 enforces the BSP design by requiring that blocks are processed through a callback function that it executes on each block, followed by a communication phase, both explained in the next section. It is this requirement that lets the library control how the blocks are managed. During the computation phase, the library moves the blocks in and out of core and processes them simultaneously using multiple threads. These are the key new features of DIY2 compared to DIY1.

Other programming models that share some similarity with our block-structured abstraction are Charm++, Legion, and Regent. Charm++ [19] has a dynamic mapping between data objects and processes (called chares) and between chares and physical processes. In Legion [3], blocks are called logical regions, and the runtime maps logical regions to physical resources. Regent [39] is a new language and compiler for the Legion runtime, which results in shorter, more readable code than the original Legion language.

2.2 Out-of-core and I/O-efficient algorithms

Many algorithms have been developed for specific applications running out-of-core. Vitter summarizes many of these in his survey of out-of-core algorithms and data structures for sorting, searching, fast Fourier transform, linear algebra, computational geometry, graphs, trees, and string processing [46].

Thakur and Choudhary developed runtime support for out-of-core multidimensional arrays [43] with an extended two-phase method [42] that used collective I/O to access subarrays in storage. The collective two-phase algorithm performed better than independent access. The same algorithm was deployed in the PASSION runtime [44] for collective I/O, a precursor to today's MPI-IO parallel I/O interface. The PASSION compiler was one part of an out-of-core high-performance Fortran (HPF) system [41]. Bordawekar and Choudhary [4] categorized communication strategies for out-of-core arrays in HPF based on the associated I/O required to execute the communication.

LaSalle and Karypis presented an out-of-core MPI runtime called BDMPI (Big Data MPI) [22], intended as a drop-in replacement for MPI. It launched more processes than available cores and used POSIX message queues to block individual processes (by waiting on a message not yet sent). When this happened, the OS virtual memory mechanism moved the data from memory to swap space. Unfortunately, this approach is not usable on IBM and Cray supercomputers that have no virtual memory and do not allow more processes than cores. Moreover, BDMPI implemented a small subset of the MPI standard and lacked nonblocking, one-sided, and many collective operations. Rather than replacing MPI, DIY2 is a C++ library built on top of MPI, and so the user is free to use any MPI facilities in the same code as DIY2.

Durand et al. [11] proposed I/O scheduling based on graph matching and coloring. Modeling the set of clients (compute nodes) as one side of a bipartite graph and the set of servers (disks) as the other side, the resulting schedule of I/O transfers attempted to compute a near-optimal schedule. It did so by maximizing the number of edges between clients and servers in a phase, subject to the constraint that no graph nodes have multiple edges, and by minimizing the number of such phases.

In contrast to previous methods that optimized the execution of storage accesses, Colvin et al. [9] proposed a language and compiler that minimized the number of storage accesses issued by an out-of-core program. Based on the C* language, the virtual C* (ViC*) compiler reorganized loops that accessed out-of-core variables primarily by fusing loops and recomputing data instead of reading it from storage.

In addition to reorganizing loops, Kandemir et al. [20] also reorganized file layouts to better match the loop structure in order to optimize I/O without sacrificing in-core parallelism or introducing additional communication. Brezany et al. [5] augmented the HPF language with directives for out-of-core handling of arrays. The augmented language and compiler, HPF+, executed data-parallel loops out-of-core while reducing redundant I/O accesses. It did so by reordering computations and hiding I/O latency by overlapping I/O with computation. All of the previous methods considered only loops over arrays. In contrast, DIY2 supports any general operations (not limited to loop structures) on any general data structures (not limited to arrays).

2.3 Data-intensive programming models

Programming models derived from MapReduce [10] built on the Hadoop runtime are inherently out-of-core because communication is based on files or TCP pipes between remote machines. Frameworks such as Dryad [18], Pig [28], and Hive [45] use SQL and/or MapReduce constructs to describe data processing tasks out-of-core. While convenient for programming, such frameworks are limited to data that can be expressed as key-value pairs and to processing of loosely-coupled problems that are expressed as a global reduction of embarrassingly parallel components. Thus far, neither MapReduce nor its descendants—iterative Twister [12], streaming CGL-MapReduce [13], and distributed shared memory Spark [47]—have been suited for HPC applications that demand highly scalable complex communication.

The bulk-synchronous parallel (BSP) style of algorithm development, used in DIY2, is also present in modern libraries for graph analytics. Pregel [26] and GraphLab [25] are two examples, although both operate only in core. Examples of out-of-core graph libraries are Giraph [8] and GraphChi [21]; they can accommodate much larger graphs. These tools, however, do not support distributed-memory parallelism. In contrast to these examples, our objective is a general-purpose in- and out-of-core data-parallel model that is not limited

to a particular application or analytics area.

3 Design

We begin with a short example. After a brief discussion of the main steps in the example, we explain the novel design points of DIY2 in greater detail.

3.1 Example

Listing 1 presents the typical structure of a DIY2 program. The top-level DIY2 object is called `master`. Its main responsibility is to keep track of the blocks of data. A single program can contain multiple master instances corresponding to different types of data. For example, a pipeline of several analyses, each requiring different resources can be constructed in one program with a different master object for each step. A master object is initialized with an MPI communicator *world* and parameters that specify how many threads to use, how many blocks to keep in memory at once, and where to store blocks (and their message queues) that must be evicted from memory.

All parameters other than the MPI communicator are optional: by default, DIY2 uses a single thread and allows all blocks to live in memory. The master is populated with blocks. Auxiliary facilities can help determine block boundaries when the decomposition is a regular lattice of blocks or a k-d tree; both are supported.

A typical execution phase is invoked by `foreach(&foo)`, which calls function `foo()` with every block stored in the master, possibly using multiple threads simultaneously, depending on the parameter passed to the master's constructor. A pointer to the current block *b*, a communication proxy *cp*, and custom auxiliary arguments are provided to `foo()`. A communication proxy is the object that manages communication between the current block and its neighbors. Inside `foo()`, data are dequeued from each neighbor using *cp*. Presumably some local work is done on the received data before enqueueing outgoing data to be sent to neighboring blocks. Those data are exchanged in the next communication phase using the `exchange()` function of the master.

Depending on the value of *mem_blocks*, the master keeps only a limited number of blocks in memory. In this case `foreach()` moves blocks and their queues between memory and `storage` as necessary; `exchange()` does the same but only with the queues. All blocks or just one block in memory are not the only two choices available to the user: any number of blocks may be selected to reside in memory. Depending on the value of *num_threads*, the master executes the blocks that are in memory concurrently using multiple threads. Both of these features, controlling the number of blocks in memory and multithreading their execution, are realized simply by changing these two parameters. If these parameters are command-line arguments, recompilation is not needed: all the combinations of these modes are available at run time.

3.2 Blocks

At the heart of DIY2's design is the idea of organizing data into blocks, which despite their name need not be "blocky": subsets of a triangulation or subgraphs of a full graph make perfectly good blocks. Blocks are indivisible units of data. They can reside in different levels of memory/storage transparently to the user, and DIY2 continues to manage communication between blocks as it does when blocks are in DRAM. Blocks are created by the user and are handed off to a master, the object responsible for managing both their placement in the memory hierarchy and their communication.

A single MPI process may own multiple blocks, so master provides a method `foreach()` to execute a callback function on every block. If not all blocks reside in memory, `foreach()` decides when (and whether) a block needs to be brought into memory and when a block can be moved out. By default, DIY2 cycles through all the blocks, starting with those that remain in memory from the previous loop. DIY2 may use multiple threads to process several blocks simultaneously.

```
// main program
Master             master(world, num_threads, mem_blocks, ...);
ContiguousAssigner assigner(world.size(), tot_blocks);

decompose(dim, world.rank(), domain, assigner, master);

master.foreach(&foo);
master.exchange();

// callback function for each block
void foo(void* b, const Proxy& cp, void* aux)
{
  for (size_t i = 0; i < in.size(); i++)
    cp.dequeue(cp.link()->target(i), incoming_data);

  // do work on incoming data

  for (size_t i = 0; i < out.size(); i++)
    cp.enqueue(cp.link()->target(i), outgoing_data[i]);
}
```

Listing 1: A typical DIY2 program. Omitted are details how to construct, destruct, and serialize a block.

The `foreach()` callback function receives a pointer to the block and also an auxiliary communication `proxy`. The latter allows the user to enqueue/dequeue information to/from the neighbor blocks, exchanged during the next/previous round of communication. Once `foreach()` is finished, the user may request the master to exchange its outgoing queues with the neighbors (and accordingly, receive incoming queues to be processed at the next round). Thus, the foreach/exchange mechanism accommodates the bulk-synchronous parallel (BSP) model of algorithm design and formalizes its use.

3.3 Data types

DIY2 supports arbitrary data types in blocks and messages. Both communication and block movement mechanisms rely on DIY2 serialization routines. The library uses C++ template specialization to facilitate serialization. The default implementation simply copies the binary contents of the object. This works as intended for *plain old data* (in C++ terminology), but if a class contains complicated members (e.g., pointers or STL containers), extra logic is necessary. DIY2 uses partial template specialization for many STL containers. Accordingly, data models consisting of complex types usually need to specify only which members to serialize; the actual logic for serialization of the base types (for example, `std::vector`) already exists in DIY2.

The serialization mechanism is deliberately simple: it does not chase pointers, does not track objects, and does not handle polymorphism. If the user wants such facilities, she can trivially wrap an external serialization mechanism such as the Boost serialization library[1] without adding overhead. We note, however, that despite its simplicity, DIY2's data type mechanism is powerful enough to accommodate all the data models in the applications described in Section 4.

3.4 Communication patterns

Communication happens strictly at the block level: blocks enqueue messages to each other, and DIY2 translates them into the messages between MPI processes (appending source and destination block IDs). There are two types of communication in DIY2: local and global.

[1] http://www.boost.org

```
// create partners (choose one)
RegularMergePartners       partners(dim, nblocks, k, contiguous);
RegularSwapPartners        partners(dim, nblocks, k, contiguous);
RegularAllReducePartners   partners(dim, nblocks, k, contiguous);

// execute the reduction
reduce(master, assigner, partners, &foo);

// callback function
void foo(void* b, const ReduceProxy& rp, const RegularMergePartners& partners)
{
  for (i = 0; i < rp.in_link.size(); i++)
    rp.dequeue(rp.in_link().target(i), incoming_data[i]);

  // do work on incoming data

  for (i = 0; i < rp.out_link.size(); i++)
    rp.enqueue(rp.out_link().target(i), outgoing_data[i]);
}
```

Listing 2: Examples of DIY2 global reductions. The user would include one of the partner objects shown followed by a call to the `reduce()` function. The user also provides the callback function `foo()`. The example shows an example of the callback for `RegularMergePartners`, callbacks for the other types of partners are similar.

Local neighbor exchange. For local communication, the master records, in a `link` class, the neighbors of every block it owns. The link is our abstraction for the local connectivity (sub)graph representing the edges to the blocks with which the current block can communicate. In its most basic form, the link stores the list of neighbor blocks and their corresponding MPI ranks. For certain regular decompositions, links record extra information for convenience: for example, the spatial direction of the neighbors or their block bounds. From the point of view of an algorithm designer, the link isolates her from the details of how a particular communication pattern maps onto hardware. The links can be set up manually by the user in the case of custom communication topologies; in the case of regular communication topologies, DIY2 sets up links automatically. The two regular topologies currently supported by automatic link creation are a regular lattice and a k-d tree of blocks.

We have already seen the local neighbor exchange pattern in Listing 1. The user and the DIY2 runtime communicate over the link using the communication proxy during alternating phases of `foreach()` callback functions and master exchanges.

Global reductions. Global reductions are invoked in a DIY2 program with a single function call and a custom callback for the operator to be executed in each round. Examples are shown in Listing 2. A user calls `reduce()` with a master, assigner, partners object, and a callback `foo()`. The assigner is a class aware of the global placement of all the blocks. The partners class specifies how the blocks communicate during the reduction, which proceeds in rounds. For each round, partners determines whether a given block is active, and what its `in_link` and `out_link` are. The two links record from which blocks a block receives messages and where it should send the results of its own computation. Both links are passed to the callback `foo()`, so that the user can enqueue and dequeue messages as before.

Internally, DIY2 uses the same foreach/exchange mechanism to implement global communication patterns. It dynamically creates links and adjusts how many messages each block expects to receive during an exchange. Because global reduction patterns are built on the same local exchange mechanism, they take advantage of the same automated block movement in and out of memory.

DIY2 implements several partners classes. Two of them are especially useful and serve as common building blocks: `RegularMergePartners` and `RegularSwapPartners`. The former expresses the communication

pattern for a k-ary reduction to a single block and can be used to implement familiar MPI operations such as `MPI_Reduce` and `MPI_Gather`.

Swap partners organize all b blocks into communication groups of size k, where, depending on a user-specified parameter, the distance between blocks in the groups either grows by a factor of k in each round (starting with groups of contiguous blocks) or shrinks by a factor of k (starting with i-th, $(b/k + i)$-th, $(2b/k + i)$-th, etc. blocks in the same group). The former (growing) arrangement is useful when one needs to unite contiguous data, for example during the computation of local–global merge trees [27]. The latter (shrinking) arrangement is useful for sorting values into specified ranges in $\log_k b$ iterations of $k \cdot b$ messages each. `MPI_Reduce_scatter` and `MPI_All_reduce` can be expressed as swap reductions.

The partner patterns are more flexible than their MPI counterparts, not least because they dissociate block IDs from MPI ranks. If the blocks form a regular decomposition of a d-dimensional lattice, both merge and swap partners can alternate between different dimensions between the rounds (hence the name "Regular"). Such alternation is useful when one wants to keep the shape of the merged data uniform in all dimensions. Furthermore, one may drop some of the dimensions from the reduction, which is useful for computing the projection of the data along those dimensions. In general, it is possible to initialize both types of partners with a list of pairs, each recording the dimension and the group size to use in the given round.

As mentioned above, motivated by the work in [30] and the knowledge that higher k are better on some architectures, DIY2 supports k-ary reductions. The interconnection fabrics of modern supercomputers are multiported and use RDMA, meaning that k can usually be higher than 2, often 8 or 16, before saturating the network (depending on the application and message size). Some collective algorithms in MPI use a binary tree reduction in order to be portable, and users of DIY2 can often do better by tuning k to their network and application characteristics.

The number of blocks b need not be a power of k. To support such a general arrangement, the number of blocks (in each dimension, in case of multidimensional decomposition) is factored as $b = k_1 \cdot k_2 \cdot \ldots$, where individual factors k_i are kept as close as possible to the target k. Specifically, prime factors of b that are less than or equal to k are grouped together, so that the product of individual groups does not exceed k. Prime factors of b that are greater than k become individual k_i multipliers out of necessity.

The global communication patterns discussed above are included in DIY2, but the user is not limited to them. Any pattern that proceeds in BSP rounds can be implemented in DIY2; such a pattern can be expressed by a user through a 'Partners' class (of which merge, swap, and allreduce partners are just examples). As a specific example, DIY2 includes an algorithm to distribute particles into blocks through a k-d tree decomposition. The pattern involved in that computation is more general than the pure merge or swap reductions. Section 4.2 shows how three analysis codes take advantage of the above patterns.

3.5 Out-of-core movement

A key feature of DIY2 is its ability to move data seamlessly across the memory/storage hierarchy. This feature is becoming increasingly important on HPC architectures with limited memory capacity per core but increasing number of levels of memory/storage; for example, the U.S. Department of Energy (DOE) leadership computing architectures [2].

In DIY2, the user can limit the number of blocks that a master stores in memory by specifying this limit to the constructor, together with details about where to store blocks that have been evicted from memory. Inside the library, external storage is abstracted as a dictionary where one can put arbitrary binary buffers to fetch back later. DIY2 saves each such buffer in its own file.

DIY2 does not hide the latency of fetching the data. Instead, it treats memory as the most limited resource. In general, the user should try to make blocks as large as possible, as long as the number of blocks equal to the number of threads still fits in memory. If they do not, the user has to increase the number of blocks (making individual blocks smaller), so that some of them may be kept in the external storage. Because of this goal to maximize the size of individual blocks, DIY2 does not use prefetching when loading the data, which would require extra space in memory. Similarly, when saving the data, DIY2 calls `fsync` to make sure that when the operation is over, not only are our user-space buffers free, but the data has left the kernel buffers as well.

Sometimes multiple smaller blocks per process are useful. They can enable multithreading, and assuming the blocks are not adjacent, having multiple blocks spread throughout the domain can be a useful load-balancing tool. In such cases, recall that DIY2 is not limited to all or one block in memory, but any number of blocks in between. If there is ample memory for more than one block, but not enough for all blocks, the user can configure DIY2 to keep as many blocks as will fit. This is more effective than keeping one block in memory while trying to prefetch others because there is no chance of guessing wrong.

The one instance where we foresee prefetching to be useful is if DIY2 were to support more than two levels of memory/storage. Currently, this is not the case: the user specifies the second level in the hierarchy as a filesystem path. Looking ahead, if DIY2 had three (or more) levels of block migration, then it would be beneficial to use the intermediate level(s) for prefetching and staging blocks en route to main memory.

By default, `foreach()` cycles through the blocks, starting with those that are already in memory. When it needs to fetch a new block from external storage, it unloads all the blocks from memory, together with their queues. (More accurately, the thread that needs to load a block unloads all the blocks that it owns.) The blocks are serialized as individual units (i.e., each block gets its own file); all the outgoing queues of a block are saved together as one unit. During an exchange phase, the master fetches outgoing queues from external storage as necessary while ensuring that the number of queues in flight (the queues posted by `MPI_Isend` whose `MPI_Request` s are still pending) does not exceed the allowed block limit. Upon receiving a queue, the master moves it to storage if its target block is stored out of core.

Skip mechanism. Although the above mechanism is sufficient in many cases, it is not always as efficient as it could be. For example, in Section 4.2, we describe an iterative code where blocks do not need to be updated if their incoming queues are empty. To avoid having to load a block, just to determine that we did not need to do so after all, `foreach()` accepts an optional parameter, *skip*. Skip is a C++ functor that DIY2 calls to decide whether a block needs to be processed. The functor has access to the master itself and, therefore, can query its different properties, including the sizes of the incoming queues for a block. If skip determines that a block does not need to be processed, the block is not loaded into memory.

Another example where the skip mechanism is essential to improve efficiency is the global merge reduction, described earlier. When `reduce()` is told by a partners class that a particular block is inactive during a processing round, it passes this information on to the `foreach()` function by supplying an appropriate instance of skip. This mechanism is essential to efficiently implement a global merge reduction, where, as data are reduced to a single block, the number of active blocks drops by a factor of k from round to round.

Swap elision. Another out-of-core optimization implemented in DIY2 is *swap elision.* If multiple `foreach()` operations are executed consecutively, then, by default, if the number of blocks in memory is limited, the master swaps each block in and out of core, once per operation. But if there are no exchange phases in between, there is no reason to unload the block after a `foreach()` only to reload it for the next callback, which receives no new information. So DIY2 lets the user switch the default *immediate* to a *delayed* mode, where `foreach()` only queues callbacks, to be later executed either explicitly by the user or implicitly by calling `exchange()`.

On the surface, swap elision looks like a trivial optimization: after all, one can always create an auxiliary function that calls back the different user functions—manually mimicking the queueing process. This optimization becomes important, when `foreach()` is used inside other functions that a user cannot easily modify. For example, a typical reduce operation that carries out global communication begins and ends with a call to `foreach()`, to enqueue data during the initial and to dequeue data during the final rounds of communication. Executing two such reduce operations in a row—for example, in a parallel sample sort where random samples are gathered, quantiles are determined, and then the data are sent to the correct blocks—the last `foreach()` of the first and the first `foreach()` of the second reduction do not exchange any information. Therefore, swapping blocks between them is inefficient.

The out-of-core mechanism employs other optimizations. For example, the master accepts an extra "queue policy" that specifies when not to evict queues; the default policy allows queues smaller than 4 KB to stay in memory. Overall, these optimizations are less important, and we do not describe them in detail.

3.6 Algorithms

The core features described above can be used to build more complex algorithms. Two such algorithms — parallel sort and k-d tree decomposition — are widely useful, hence, we include them in DIY2. We anticipate adding more algorithms in the future. We briefly describe the k-d tree decomposition in this section, with performance evaluation in Section 4.2.

Our implementation of radix-2 k-d tree decomposition uses histograms to select the split points for the data. We use merge reductions to compute histograms of subsets of the data projected on a particular coordinate and interleave them with swap rounds that exchange points between blocks.

We assume no ordering on the input data and start with roughly equal number of points in each block, points that can be located anywhere in the domain. During the first iteration of the algorithm, each block computes the histogram of the projections of its points onto the first coordinate; the number of bins in a histogram is specified in advance by the user. The histograms are merged following a binary reduction, and an approximate median value is chosen from the complete histogram.

Half of the blocks (those whose IDs fall in the bottom half of the range) receive the points whose projection onto the first coordinate is less than the median value; the other half receive the points with the projection above the median. Each block exchanges points with a single partner in the other half of the blocks (the partner is chosen by flipping the first bit of the block's ID).

After the exchange, the domain is effectively divided into two: with half of the blocks responsible for the points in the lower half of the domain, and half with the upper. From this point on, blocks in each half are restricted to communicate only among themselves. The entire process repeats for $\log b$ rounds (b being the total number of blocks), cycling through the dimensions chosen for projection.

Throughout the process, each block maintains its own link (the list of neighboring blocks used for local communication after the decomposition). Initially, the link is either empty or, in case of periodic boundaries, each block links to itself in every direction. After each iteration, once the points are exchanged between blocks, each block queries every neighbor in its link for the median value computed in the last round. Using this information it updates its local link, possibly adding or removing blocks from it. Each block also adds the new neighbor resulting from its own split; this neighbor is chosen to be the partner used for point exchange. The resulting neighborhood information is valuable for algorithms that use local communication (for example, Delaunay tessellation code described in Section 4.2) that can take advantage of the k-d tree decomposition as a preprocessing step.

4 Experiments

Performance tests were run on the IBM Blue Gene/Q *Mira* and Cray XC30 *Edison* machines at the Argonne Leadership Computing Facility (ALCF) at Argonne National Laboratory and at the National Energy Research Scientific Computing Center (NERSC) at Lawrence Berkeley National Laboratory, respectively. Mira is a 10-petaflop system consisting of 48K nodes, each node with 16 cores (PowerPC A2 1.6 GHz) and 16 GB RAM. Edison is a 2.57-petaflop machine with 5576 nodes, each node with 24 cores (Intel Ivy Bridge 2.4 GHz) and 64 GB RAM. GCC (version 4.4.7 on Mira, version 4.9.2 on Edison) with `-O3` optimization was used to compile the test code.

The out-of-core tests were performed on *Alva*, a small development cluster on Edison, used as a burst buffer testbed. The cluster has 14 compute nodes, which have access to six burst buffer nodes, exposed as six individual flash file systems. Each one has 3.2 TB of storage space, composed of two Sandisk Fusion 1.6 TB SSD cards.

4.1 Benchmark applications

We use a communication benchmark suite to test the communication patterns of DIY2 when all blocks fit in memory. We compare the performance with DIY1 [32, 34] (which only supports in-core blocks) and when possible, with equivalent collective functions in MPI itself (by assigning one block per MPI rank). We want to

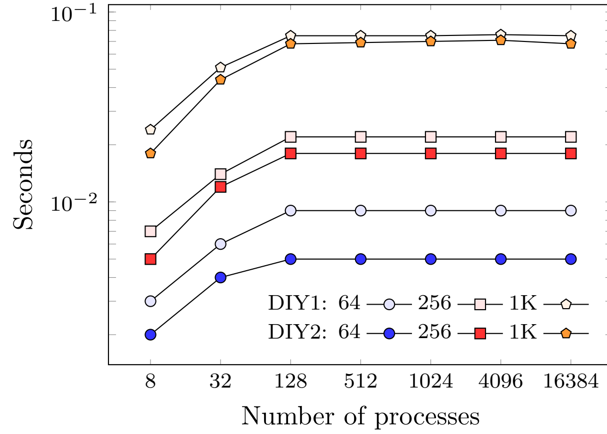
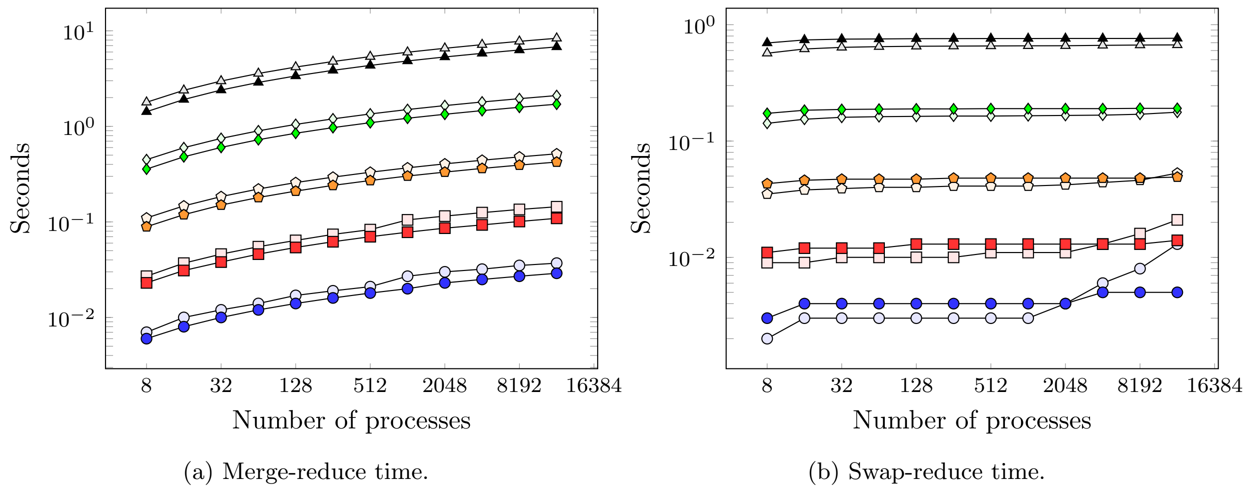


Figure 1: Cian mini-app neighbor exchange using DIY2 compared with DIY1; 20 bytes/item. (Number of items in the legend.)



(a) Merge-reduce time.

(b) Swap-reduce time.

Figure 2: Cian mini-app merge- and swap-reduce using DIY2 compared with MPI using reduce and reduce-scatter, respectively.

measure whether the out-of-core and threading features, the main improvements that DIY2 brings over DIY1, add any overhead when they are not used. Our benchmark suite is one of the mini applications from the DOE exascale co-design initiative. The Center for Exascale Simulation of Advanced Reactors (CESAR) [40], one of three co-design centers funded by the DOE, contains a suite of data analysis benchmarks called cian.[2] It exercises the main communication patterns used in many distributed-memory data analysis algorithms: neighbor exchange, merge reduction, swap reduction, and parallel sorting. The message sizes and numbers of blocks are configurable in the cian mini-app; hence, one may test a particular regime such as latency- or bandwidth-bound communication at a variety of system sizes.

Neighbor exchange. Figure 1 shows a comparison of DIY1 and DIY2 neighbor exchange with large numbers of small items exchanged. This test, run on the Mira BG/Q machine, is a stress test of neighbor exchange that measures the enqueue and dequeue rate. In cian, we can set the number of items and the item size. We used 20-byte items, emulating five 32-bit values $(x, y, z, mass, id)$ that identify a particle in a cosmology dataset. Figure 1 demonstrates that not only does DIY2 add no overhead compared with DIY1, its enqueue–exchange–dequeue rate is slightly better, especially at small numbers of items such as 64. Our experiments also indicate that the total message time is approximately linear in the total message size, meaning that DIY2 has no noticeable overheads as the number of small messages grows very large (we tested up to 1M 20-byte messages).

Merge and swap reductions. Figure 2 compares DIY2 reductions against their MPI counterparts; experiments were run on the Mira BG/Q machine. The parameters for our tests were chosen so that our results could be compared against MPI. This means that the number of rounds and group sizes k per round produced a merged result in a single block equivalent to `MPI_Reduce`, and the swapped result was distributed among all blocks equivalent to `MPI_Reduce_scatter`. We used one DIY2 block per MPI process and tested block counts that were powers of two. Tests were run in symmetric multiprocessor mode, one MPI process per node. Our reduction operator is the noncommutative *over* operator [35] used in image compositing, a linear combination of elements in a floating-point buffer that represents the red, green, blue, and opacity channels of pixels in an image. Our message sizes are based on images of various resolutions at 16 bytes per pixel. For merging, we found $k = 2$ to perform best; for swapping, $k = 8$ was used. For comparison with DIY1, please refer benchmark results in Figure 2 of reference [32]. The results here are directly comparable with those.

The merge communication pattern is used for associative reduction of heterogeneous data that cannot be readily distributed and instead must be merged in place at a smaller number of blocks during each round. Topological structures such as Morse–Smale complexes [16] and persistence diagrams [23] can be reduced this way. The merge-reduction algorithm in DIY1 was first published in 2011 [34]. The swap communication pattern is used for associative reduction of homogeneous contiguous data buffers that remain distributed instead of being merged into a smaller number of blocks. This case occurs in sort-last parallel rendering, when multiple image buffers are blended together. In fact, the algorithm is a generalization of the radix-k image compositing algorithm first published in 2009 [30].

Figure 2 shows merge- and swap-reduction performance compared with `MPI_Reduce` and `MPI_Reduce_scatter`, respectively. In the merge-reduce test, DIY2 was approximately 1.2 times faster than the BG/Q MPI implementation. As mentioned in Section 3, we can sometimes execute comparable reductions slightly faster than MPI collectives (DIY2 algorithms are built from MPI point-to-point messages) because DIY2 allows the user greater control over configuring the algorithm (in addition to the other advantages that communicating between blocks instead of ranks brings).

In the swap-reduce test, DIY2 was approximately 1.2 times slower over much of the runs. A notable exception occurs in swap-reduce of small message sizes (512 KB and 2 MB) with process counts higher than 2048. There, the MPI implementation exhibited poor performance. We have discussed the issue with MPI developers, and their best guess is that MPI makes an automatic decision to switch to a different communication algorithm at larger process counts; in this case, it appears to be a wrong decision. Reference [32] reported a similar trend. Details of the underlying MPI implementation are beyond the scope of this paper,

[2] https://github.com/tpeterka/cian2

Ranks	Regular				K-d tree			
	Min	Max	Max/Min	Time (s)	Min	Max	Max/Min	Time (s)
512	639,052	6,675,562	10.4	1,126	2,020,643	2,133,489	1.1	383
1,024	218,588	4,641,920	21.2	784	979,247	1,069,120	1.1	206
2,048	87,023	3,443,277	39.6	654	460,191	538,993	1.2	113
4,096	26,806	2,482,464	92.6	482	200,617	275,130	1.4	66
8,192	12,155	1,613,454	132.7	317	70,801	139,005	2.0	41

Table 1: Comparison of tessellation code with regular and k-d tree block decomposition: number of particles per block and total execution time.

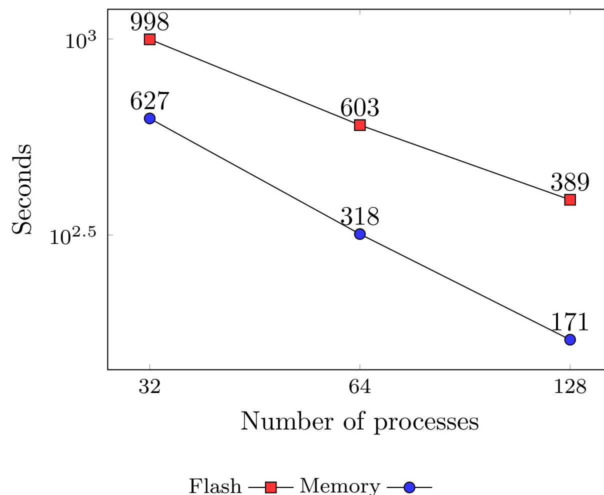


Figure 3: Time to compute Delaunay tessellation of 1024^3 points, split among 1024 blocks, using different numbers of processors. In the "Memory" setting, all blocks are kept in memory. In the "Flash" setting, only one block per process is kept in memory.

except to further illustrate the advantage of DIY2 leaving control over the choice of communication algorithms to the user.

4.2 Analysis codes

We evaluate three complete analysis codes built with DIY2. The first computes Delaunay and Voronoi tessellations of N-body particle datasets. The second uses the Voronoi tessellation to generate a density estimate of particles on a grid. The third computes distances on a grid to a set of obstacles; it is part of a larger geometric analysis package implemented on top of DIY2. Many more codes have been implemented with DIY2 than the ones featured here, including geometric, statistical, and topological analysis. For example, other applications include parallel computation of persistent homology described in [23] and distributed merge trees described in [27].

Voronoi and Delaunay tessellation. The first analysis code computes a Voronoi and Delaunay tessellation in parallel at large scale. We ported a parallel algorithm [31], originally implemented using DIY1, to DIY2. Our dataset contains 1024^3 dark matter tracer particles computed by the HACC cosmology code [17]. The data are taken from a late time step where the particles are clustered into halos and voids, regions where the number of particles can vary by a factor of 10^6 or more between the most dense and sparse areas. Hence, the

computational load for algorithms that decompose the domain into regular-size blocks varies widely. The load imbalance is exacerbated with a larger number of smaller blocks because regions of high and low density occupy a larger fraction of the block extent as the block gets smaller. This situation is a prime candidate for a k-d tree decomposition. Table 1 shows the results of the DIY2 implementation of the tess algorithm using both regular and k-d tree decompositions. With the k-d tree algorithm in DIY2, switching between the two different decompositions is enacted by changing a single runtime parameter. Table 1 shows that the regular decomposition can have a load imbalance as high as 100:1 while the k-d tree is 2:1. This is why the total time for the tessellation using the k-d tree is approximately 3 to 7 times faster than that using the regular decomposition.

Figure 3 illustrates an out-of-core strong scaling experiment using a regular block lattice. We tessellated 1024^3 input particles from a cosmological simulation, split into 1024 blocks, divided evenly between the processes. Figure 3 shows these results run on the Alva XC30 machine, comparing all blocks in memory with one block in memory and the rest in Alva's burst buffer. The flash-based out-of-core version is between 1.5 and 2 times slower than in-core, but otherwise scales with the number of processes almost as the in-core version. "Flash" refers to the fact that the storage medium consists of flash-memory burst buffer nodes.

The individual processes used the following amounts of memory, as reported by the high-water mark through Linux's `/proc` facility. When all blocks were stored in memory, the maximum high-water marks for any process were 29.5 GB for 32 processes (32 blocks per process), 15 GB for 64 processes (16 block per process), and 7.7 GB for 128 processes (8 blocks per process). When using 1024 processes, with 1 block per process, the high-water mark was 1.13 GB. When using external storage, and keeping only one block in memory, the maximum high-water mark was 1.19 GB. Therefore, the serialization and out-of-core movement mechanisms do not introduce a significant memory overhead.

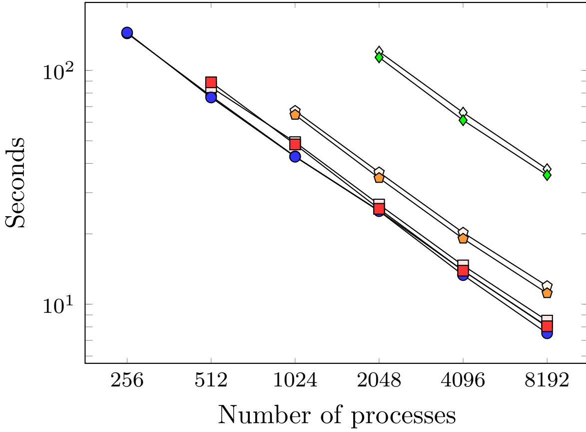
Tessellation-based density estimation. Our second application uses a Voronoi tessellation in order to estimate particle density on a regular grid. Schapp and van de Weygaert [36, 37] showed that using a tessellation as an intermediate step in estimating density can produce more accurate results than computing the density directly from the input particles. We implemented the tessellation-based density estimator [29] on top of both DIY1 and DIY2. We coupled the tessellation and density estimation into a single (tess-dense) pipeline, without an intervening disk write of the tessellation between the two stages.

Figure 4 shows a comparison of DIY1 and DIY2 implementations of the tess-dense pipeline when all blocks fit in memory. This experiment was run on the Mira BG/Q machine with an input of 512^3 randomly generated particle positions and an output density estimated on regular 3D grids ranging from 1024^3 to 8192^3. Similar to the benchmark results, Figure 4 shows that when all blocks fit in memory, the performance of DIY1 and DIY2 is virtually indistinguishable. In short, DIY2's added features such as out-of-core capability cost nothing in overhead when not needed. In fact, at the largest scale tested, DIY2 performed approximately 5% faster than DIY1 because of a better implementation of the neighbor exchange algorithm.

Another new feature that DIY2 adds over DIY1 is the ability to automatically multithread the `foreach()` block computations simply by assigning multiple threads to DIY2. The user enables this capability by changing a single run-time parameter. In comparison, manually multithreading the compute kernel of the block can be tedious and error-prone, especially when mutexes are required to protect shared data from race conditions. In the following experiment, we compared the performance of DIY2's block threading with no threading and with a manually threaded OpenMP kernel. The DIY2 automatically threaded version launches concurrent callback functions on as many blocks as available threads. In the manually-threaded OpenMP version, DIY2 is single-threaded, but the code inside the block callback function is threaded using OpenMP.

The following experiments were run on the BG/Q, which has 64 hardware threads per compute node. We divided those 64 threads into 8 MPI processes per compute node and 8 threads per MPI process. For DIY2 automatic threading, we used 8 blocks per MPI process; for OpenMP manual threading, we had 1 block per MPI process, but the block was 8 times larger than in the automatic case. In other words, the MPI processes were assigned the same amount of data in both cases. The density of 512^3 input particles was estimated onto a 1024^3 output grid using the Voronoi tessellation.

Figure 5 shows all three threading versions for the density estimation stage of the tess-dense pipeline.



DIY1: 1024^3 — 2048^3 — 4096^3 — 8192^3 —
DIY2: 1024^3 — 2048^3 — 4096^3 — 8192^3 —

Figure 4: Time to estimate density of 512^3 particles on grids of various dimensions. Grid dimensions appear in the legend.

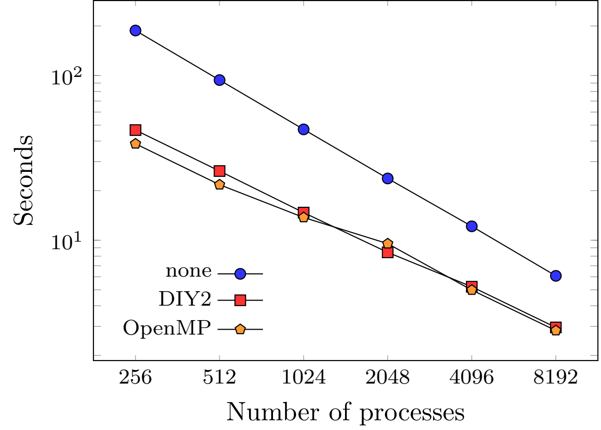


Figure 5: Density of 512^3 particles estimated onto a 1024^3 grid with different threading options.

The speedup from 1 thread to 8 DIY2 threads is approximately 4.0 times faster at 256 process and 2.3 times faster at 8192 processes. The interesting point is that the manually-written OpenMP threading is not much better. Its speedup ranges from 4.8 to 2.3 times faster. In other words, roughly the same performance was achieved with no programming effort by simply changing one parameter to DIY2 compared with manually threading a kernel using OpenMP.

Distance on a grid computation. As part of a suite of geometric analysis algorithms, we implemented a code to compute a signed distance at each grid point to a set of obstacles. The algorithm proceeds as follows. The grid is partitioned into regular blocks, each with a one-voxel wide ghost zone into neighboring blocks. We compute the distances to the obstacles within each block, recording the source, i.e., the nearest obstacle, responsible for the distance. Then, iteratively, the neighboring blocks exchange those voxels in their ghost zones where a source has changed. After the exchange, the distances (and sources) are updated, and the process repeats until no block has to update any of its distances. (The number of iterations, and the number of times a block needs to be updated depends heavily on the data.)

One notable property of this algorithm is that a block performs no updates if none of its neighbors changed the nearest obstacle sources in the block's ghost zones. When running in-core, this observation has no implications: the code quickly recognizes that incoming queues are empty and finishes processing the block without any updates. In the out-of-core setting, this observation is responsible for a key optimization. When running `foreach()` to process the blocks, it is passed a `skip` functor (described in Section 3) that checks if the block's incoming queues are empty. If they are, `skip` signals to the master that the block does not need to be loaded.

This optimization is responsible for the out-of-core running times measured on the Alva XC30 machine and labeled "Flash (skip)" in Figure 6; its advantage over the unoptimized "Flash (no skip)" version is evident. The input data set is a binary $2560^2 \times 2160$ image of a sandstone, acquired at Berkeley Lab's Advanced Light Source. The data are divided into 1024 blocks. Either all blocks are kept in *memory*, or only one block is kept in memory, while the rest are swapped out to *flash*. (In both cases, the blocks are divided evenly between the processors at the start of the program.) The imbalance in the amount of work required by the different

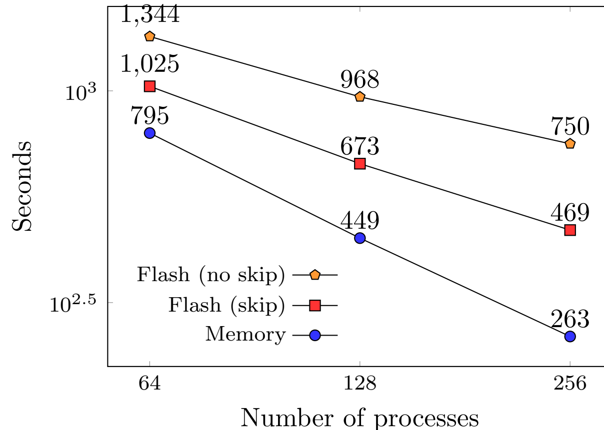


Figure 6: Time to compute distance function using different numbers of processors. In all cases, the data are divided into 1024 blocks. In the "Memory" setting, all blocks are kept in memory (evenly divided between the processors). In the "Flash" setting, only one block is kept in memory per process. "Skip" vs "no skip" refers to the optimization that skips blocks that have empty incoming queues.

blocks is difficult to exploit when running in-core, but it becomes a key other advantage in the out-of-core regime. Idle blocks stay on the external storage and do not interfere with the processing of the blocks that require updating. This is the principle reason why the block skipping optimization is so useful.

We also note the memory use of the individual processes, as reported by the high-water mark through Linux's `/proc` facility. For the in-memory regime, we get 4.12 GB for 64 processes (16 blocks per process), 2.29 GB for 128 processes (8 blocks per process), 1.36 GB for 256 processes (4 blocks per process). It is 721 MB for 1024 processes (1 block per process); the apparent overhead comes from the memory used by the serial algorithm that computes the distances inside a block. When only one block is kept in memory (in the cases of 64, 128, and 256 processes), the high-water mark stays between 693 and 767 MB, while the total external storage usage goes up to 217 GB. In other words, block serialization and out-of-core movement do not introduce significant overhead.

5 Conclusion

DIY2 establishes a foundation for developing data-parallel code that can run at high concurrency in and out of core. In this paper, we have presented the design of DIY2 as well as the experiments that illustrate its efficiency. In future work, we plan to pursue extensions to make such codes more efficient and robust.

- The partners mechanism used for global communication, described in Section 3, has a complete knowledge of which blocks will be used during which rounds. We should take this knowledge into account and choose an ordering of the blocks within rounds that would minimize their movement.
- For some algorithms, it is desirable to start with a larger number of blocks in memory and gradually decrease this limit as computation progresses. For example, the amount of memory used by the Delaunay tessellation code increases as more points are added after a round of communication. DIY2 would benefit from an ability to modify the limit on the number of blocks allowed in memory, as computation progresses. (One can even imagine a dynamic adjustment based on the total memory usage by the process.)
- Because unbalanced loads are common, we would like to add a mechanism for migrating blocks between processes. Such routines would need to be carefully integrated with the external storage mechanism, since

if the storage is shared by all the processors, migrating swapped-out blocks becomes very efficient, just a matter of sending file paths.

- DIY2 has most ingredients necessary for check-pointing. Moreover, when the code runs in an out-of-core regime, check-pointing can be especially efficient, if the swapped-out blocks from the previous round remain on the external storage during the following round. We plan to investigate the nuances of adding such a mechanism to DIY2.
- We are considering supporting more than two levels of block migration over the memory/storage hierarchy in DIY2. With three or more levels of block migration, the intermediate level(s) could be used for caching and prefetching blocks as they travel up and down the hierarchy.

None of these efforts would require serious restructuring of existing codes written on top of DIY2. As such, programmers can already take advantage of the block abstractions, communication, threading, and external storage facilities built into DIY2 to write data analysis codes whose performance is portable across different computing platforms.

Acknowledgements

We gratefully acknowledge the use of the resources of the Argonne Leadership Computing Facility (ALCF) and the National Energy Research Scientific Computing Center (NERSC). We are especially grateful to Zhengji Zhao, at NERSC, for helping us with the Alva burst buffer testbed. We are grateful to Michael Manga and Dula Parkinson for the sandstone dataset, and to Zarija Lukić for the cosmology dataset. This work was supported by Advanced Scientific Computing Research, Office of Science, U.S. Department of Energy, under Contracts DE-AC02-06CH11357 and DE-AC02-05CH11231. Work is also supported by DOE with agreement No. DE-FC02-06ER25777.

Disclaimer

This document was prepared as an account of work sponsored by the United States Government. While this document is believed to contain correct information, neither the United States Government nor any agency thereof, nor the Regents of the University of California, nor any of their employees, makes any warranty, express or implied, or assumes any legal responsibility for the accuracy, completeness, or usefulness of any information, apparatus, product, or process disclosed, or represents that its use would not infringe privately owned rights. Reference herein to any specific commercial product, process, or service by its trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government or any agency thereof, or the Regents of the University of California. The views and opinions of authors expressed herein do not necessarily state or reflect those of the United States Government or any agency thereof or the Regents of the University of California.

References

[1] J. Ahrens, B. Geveci, and C. Law. “ParaView: An End-User Tool for Large-Data Visualization”. In: *The Visualization Handbook* (2005), p. 717.

[2] J. Ang et al. “Abstract Machine Models and Proxy Architectures for Exascale Computing”. In: *Hardware-Software Co-Design for High Performance Computing (Co-HPC), 2014*. IEEE. 2014, pp. 25–32.

[3] M. Bauer, S. Treichler, E. Slaughter, and A. Aiken. “Legion: Expressing Locality and Independence with Logical Regions”. In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE Computer Society Press. 2012, p. 66.

[4] R. Bordawekar and A. Choudhary. “Communication Strategies for Out-of-Core Programs on Distributed Memory Machines”. In: *Proceedings of the 9th international conference on Supercomputing*. ACM. 1995, pp. 395–403.

[5] P. Brezany, A. Choudhary, and M. Dang. “Language and Compiler Support for Out-of-Core Irregular Applications on Distributed-Memory Multiprocessors”. In: *Languages, Compilers, and Run-Time Systems for Scalable Computers*. Springer, 1998, pp. 343–350.

[6] A. C. Calder et al. “High-Performance Reactive Fluid Flow Simulations Using Adaptive Mesh Refinement on Thousands of Processors”. In: *Supercomputing, ACM/IEEE 2000 Conference*. 2000.

[7] H. Childs et al. “Extreme Scaling of Production Visualization Software on Diverse Architectures”. In: *IEEE Computuer Graphics and Applications* 30.3 (2010), pp. 22–31.

[8] A. Ching and C. Kunz. “Giraph: Large-scale Graph Processing Infrastructure on Hadoop”. In: *Hadoop Summit* 6.29 (2011), p. 2011.

[9] A. Colvin and T. H. Cormen. “ViC*: A Compiler for Virtual-Memory C*”. In: *High-Level Parallel Programming Models and Supportive Environments, 1998. Proceedings. Third International Workshop on*. IEEE. 1998, pp. 23–33.

[10] J. Dean and S. Ghemawat. “MapReduce: Simplified Data Processing on Large Clusters”. In: *Commun. ACM* 51 (1 Jan. 2008), pp. 107–113.

[11] D. Durand, R. Jain, and D. Tseytlin. “Distributed Scheduling Algorithms to Improve the Performance of Parallel Data Transfers”. In: *ACM SIGARCH Computer Architecture News* 22.4 (1994), pp. 35–40.

[12] J. Ekanayake, H. Li, B. Zhang, T. Gunarathne, S.-H. Bae, J. Qiu, and G. Fox. “Twister: A Runtime for Iterative Mapreduce”. In: *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*. ACM. 2010, pp. 810–818.

[13] J. Ekanayake, S. Pallickara, and G. Fox. “Mapreduce for Data Intensive Scientific Analyses”. In: *eScience, 2008. eScience’08. IEEE Fourth International Conference on*. IEEE. 2008, pp. 277–284.

[14] P. Fischer, J. Lottes, D. Pointer, and A. Siegel. “Petascale Algorithms for Reactor Hydrodynamics”. In: *Journal of Physics Conference Series* 125.1 (July 2008), pp. 012076–+.

[15] 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.

[16] A. Gyulassy, T. Peterka, V. Pascucci, and R. Ross. “Characterizing the Parallel Computation of Morse-Smale Complexes”. In: *Proceedings of IPDPS ’12*. Shanghai, China, 2012.

[17] S. Habib, V. Morozov, N. Frontiere, H. Finkel, A. Pope, and K. Heitmann. “HACC: Extreme Scaling and Performance Across Diverse Architectures”. In: *Proceedings of SC13: International Conference for High Performance Computing, Networking, Storage and Analysis*. SC ’13. Denver, Colorado: ACM, 2013, 6:1–6:10.

[18] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. “Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks”. In: *ACM SIGOPS Operating Systems Review*. Vol. 41. 3. ACM. 2007, pp. 59–72.

[19] L. Kalé and S. Krishnan. “CHARM++: A Portable Concurrent Object Oriented System Based on C++”. In: *Proceedings of OOPSLA’93*. Ed. by A. Paepcke. ACM Press, Sept. 1993, pp. 91–108.

[20] M. Kandemir, A. Choudhary, J. Ramanujam, and M. A. Kandaswamy. “A Unified Framework for Optimizing Locality, Parallelism, and Communication in Out-of-Core Computations”. In: *Parallel and Distributed Systems, IEEE Transactions on* 11.7 (2000), pp. 648–668.

[21] A. Kyrola, G. E. Blelloch, and C. Guestrin. “GraphChi: Large-Scale Graph Computation on Just a PC.” In: *OSDI*. Vol. 12. 2012, pp. 31–46.

[22] D. LaSalle and G. Karypis. “BDMPI: Conquering BigData with Small Clusters using MPI”. In: *Proceedings of the 2013 International Workshop on Data-Intensive Scalable Computing Systems*. ACM. 2013, pp. 19–24.

[23] R. Lewis and D. Morozov. “Parallel Computation of Persistent Homology using the Blowup Complex”. In: *Proceedings of the Annual Symposium on Parallelism in Algorithms and Architectures*. 2015, pp. 323–331.

[24] Y. Liu and J. Snoeyink. “A Comparison of Five Implementations of 3D Delaunay Tessellation”. In: *Combinatorial and Computational Geometry* 52 (2005), pp. 439–458.

[25] Y. Low, D. Bickson, J. Gonzalez, C. Guestrin, A. Kyrola, and J. M. Hellerstein. “Distributed GraphLab: A Framework for Machine Learning and Data Mining in the Cloud”. In: *Proceedings of the VLDB Endowment* 5.8 (2012), pp. 716–727.

[26] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski. “Pregel: A System for Large-scale Graph Processing”. In: *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM. 2010, pp. 135–146.

[27] D. Morozov and G. Weber. “Distributed Merge Trees”. In: *Proceedings of the Annual Symposium on Principles and Practice of Parallel Programming*. 2013, pp. 93–102.

[28] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. “Pig Latin: A Not-So-Foreign Language for Data Processing”. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*. ACM. 2008, pp. 1099–1110.

[29] T. Peterka, H. Croubois, N. Li, S. Rangel, and F. Cappello. “Self-Adaptive Density Estimation of Particle Data”. In: *To appear in SIAM Journal on Scientific Computing SISC Special Edition on CSE’15: Software and Big Data* (2016).

[30] 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.

[31] T. Peterka, D. Morozov, and C. Phillips. “High-Performance Computation of Distributed-Memory Parallel 3D Voronoi and Delaunay Tessellation”. In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE Press. 2014, pp. 997–1007.

[32] T. Peterka and R. Ross. “Versatile Communication Algorithms for Data Analysis”. In: *EuroMPI Special Session on Improving MPI User and Developer Interaction IMUDI’12*. Vienna, AT, 2012.

[33] 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.

[34] T. Peterka et al. “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.

[35] T. Porter and T. Duff. “Compositing Digital Images”. In: *Proceedings of 11th Annual Conference on Computer Graphics and Interactive Techniques*. 1984, pp. 253–259.

[36] W. Schaap and R. van de Weygaert. “Continuous Fields and Discrete Samples: Reconstruction Through Delaunay Tessellations”. In: *Astronomy and Astrophysics* 363 (2000), pp. L29–L32.

[37] W. E. Schaap. *DTFE: The Delaunay Tesselation Field Estimator*. Ph.D. Dissertation. University of Groningen, The Netherlands, 2007.

[38] 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. San Francisco, California, United States: IEEE Computer Society Press, 1996, 93–ff.

[39] E. Slaughter, W. Lee, S. Treichler, M. Bauer, and A. Aiken. "Regent: A High-Productivity Programming Language for HPC with Logical Regions". In: *Supercomputing (SC)*. 2015.

[40] C. Team. *The CESAR Codesign Center: Early Results*. Tech. rep. Apr. 2012.

[41] R. Thakur, R. Bordawekar, and A. Choudhary. "Compiler and Runtime Support for Out-of-Core HPF Programs". In: *Proceedings of the 8th international conference on Supercomputing*. ACM. 1994, pp. 382–391.

[42] R. Thakur and A. Choudhary. "An Extended Two-Phase Method for Accessing Sections of Out-of-Core Arrays". In: *Scientific Programming* 5.4 (1996), pp. 301–317.

[43] R. Thakur and A. Choudhary. "Runtime Support for Out-of-Core Parallel Programs". English. In: *Input/Output in Parallel and Distributed Computer Systems*. Ed. by R. Jain, J. Werth, and J. Browne. Vol. 362. The Kluwer International Series in Engineering and Computer Science. Springer US, 1996, pp. 147–165.

[44] R. Thakur, A. Choudhary, R. Bordawekar, S. More, and S. Kuditipudi. "Passion: Optimized I/O for Parallel Applications". In: *Computer* 29.6 (1996), pp. 70–78.

[45] A. Thusoo et al. "Hive-A Petabyte Scale Data Warehouse Using Hadoop". In: *Data Engineering (ICDE), 2010 IEEE 26th International Conference on*. IEEE. 2010, pp. 996–1005.

[46] J. S. Vitter. "External Memory Algorithms". In: *Handbook of massive data sets*. Springer, 2002, pp. 359–416.

[47] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. "Spark: Cluster Computing with Working Sets". In: *Proceedings of the 2nd USENIX conference on Hot topics in cloud computing*. 2010, pp. 10–10.