Conveyors for Streaming Many-To-Many Communication

F. Miller Maley
IDA/CCR-Princeton

Jason G. DeVinney
IDA/CCS

June 10, 2019

Abstract

We report on a software package that offers high-bandwidth and memory-efficient ways for a parallel application to transmit numerous small data items among its processes. The package can be built atop any of several common programming models, including SHMEM, MPI, and UPC. It defines a simple interface to parallel objects called *conveyors*, and it provides a variety of conveyor implementations. Often the most efficient type of conveyor is an *asynchronous three-hop* conveyor, which makes heavy use of fast intranode communication. This type also uses the least memory internally. Conveyors of this type scale well to 2^{17} processes and beyond.

The conveyor interface is a low-level API, so conveyors do not solve the problem of making communication-intensive parallel code easy to read and write. They may, however, provide a useful foundation for future efforts in that direction.

1 Introduction

On most massively parallel computers, point-to-point communication of individual small data items makes poor use of the communication network. To achieve high throughput, one must aggregate the items in some way. In our experience, a surprising variety of applications can tolerate the resulting increase in latency. The problem we consider is how to present an efficient and convenient aggregation capability to the application programmer.

The project described here is part of the `bale` package [3] assembled by the second author and his colleagues, and was inspired by the older modules `exstack` and `exstack2` in `bale`. The common theme is that a library provides communication objects, in our case called *conveyors*, which support three

basic operations: `push`, which tries to enqueue an item for delivery to a specified process; `pull`, which tries to receive an item and learn which process sent it; and `advance`, which ensures forward progress. (The names used for these operations vary.) These communication objects are designed to support algorithms that involve irregular access to data distributed across a large parallel machine, with little or no exploitable locality.

Message aggregation is a natural and frequently implemented idea (see [2, 4, 5] for instance), so what is special about conveyors? The conveyor project demonstrates that aggregation can be performed in a way that is modular, portable, highly efficient, frugal with memory, and scalable to very large systems. In more detail, conveyors are:

- **Modular.** The conveyor API carefully specifies the contract between the application and the conveyor. If the application adheres to this contract, then the conveyor's communication mechanism can change drastically, even from bulk synchronous (see Section 2.1) to asynchronous, without affecting correctness.

- **Portable.** Our conveyors can be built atop SHMEM, MPI, or UPC (Unified Parallel C). In the MPI case, they only use features of MPI-1.

- **Efficient.** We describe a family of *asynchronous multi-hop* conveyors that, in many cases, achieve unsurpassed performance. These conveyors exploit the ability to communicate quickly between processes on the same node.

- **Frugal.** Compared with simple aggegration schemes, asynchronous three-hop conveyors reduce the number of buffers per process, in the case of N processes, from $O(N)$ to as little as $O(N^{1/3})$, yielding a massive reduction in memory usage. Remarkably, this savings comes at no cost in sustained bandwidth; at large scales, the three-hop conveyors outperform all others.

- **Scalable.** When we benchmark our best conveyors at scales of 2^{17} CPU cores and above, with one process per core, we observe only a 20–30% loss of injection bandwidth per node relative to small node counts.

Conveyors allow communication-heavy applications to perform well with many processes (SHMEM PEs, MPI ranks, UPC threads) per node, which

removes one major motivation for multithreading (e.g., OpenMP) and may allow the programmer to employ a simpler, one-level model of parallelism. And although our conveyors are most efficient when built on a PGAS programming model, they can help the programmer avoid the direct use of PGAS features. Conveyors can substitute for both remote memory access and point-to-point communication, allowing an application to use only conveyor calls and collective operations, which enhances its portability. All that is then required to change the communication substrate from, say, SHMEM to MPI is a thin translation layer for initialization, finalization, reductions and other collectives, and simple things like process identification (think `shmem_my_pe` versus `MPI_Comm_rank`).

Like their predecessors, however, conveyors are not particularly easy to use. Both the `push` and `pull` operations can fail: the former for lack of buffer space, and the latter for lack of an available item. Consequently, these operations must always be placed in a loop, and the loop must also call `advance` in order to ensure progress and detect termination. Especially when multiple conveyors are active simultaneously—for instance, one carrying queries and another carrying responses—the application code can become difficult to write, debug, and understand. Cleaner APIs can be devised, particularly in languages more expressive than plain C, but this is a topic for future work.

The rest of the paper is organized as follows. Section 2 reviews the motivation for the conveyor project and introduces `exstack` and `exstack2`, since these modules have not been detailed elsewhere. Section 3 presents the conveyor interface in some depth. Section 4 covers the architecture of the conveyor package and the different types of conveyors it offers as of version 0.5-b3. Section 5 records some representative benchmark results for conveyors on a few different platforms. Finally, Section 6 offers a higher-level view of conveyors and compares our software with `exstack` and `exstack2`.

2 Background and Motivation

The community of programmers we aim to assist writes high-performance parallel applications in various dialects of C, or sometimes C++. The model of parallelism is single program multiple data (SPMD): multiple processes run the program code independently in parallel. Communication is provided by the language itself in the case of UPC, or by an underlying communication library that implements a version of SHMEM or MPI. All these mechanisms

allow processes to exchange messages in a way that can be very efficient if the messages are of an appropriate size.

The communication pattern of greatest interest may be loosely described as all-to-all, but with different amounts of data flowing between each pair of processes. To forestall confusion, we call this pattern *many-to-many*. As a model for this pattern, think of each message as having a random destination. The expected number of items sent from one process to another, which we call the *load*, might be so large that the items must be handled in a streaming fashion, or so small (less than 1) that statistical fluctuations in item counts have a major influence on communication performance.

Many-to-many communication arises most naturally, but not exclusively, in two idioms commonly found in our parallel programs. In the first idiom, every process wants to make numerous updates to a distributed data structure. The updates can be done in any order, and no response to each update is needed. For instance, the data structure may be a large array that is spread across the processes, and each update may be an increment of a specified array element, which usually belongs to a different process. The idiom takes its name, *histogram*, from this very simple example. In UPC, or C with SHMEM, or a modern parallel language like Chapel, one can write very concise code for *histogram* using some form of remote atomic increment (Figure 1). But the resulting performance is likely to be somewhere between mediocre and dismal, depending on the compiler, communication library, and hardware platform. See Figure 2 for some relevant benchmark data.

The basic *histogram* operation is closely related to the HPC Challenge Random Access benchmark. Although the latter benchmark explicitly limits the allowed amount of buffering, both aggregation and multi-hop routing have been successfully applied to it [4].

```
for (i = 0; i < n; i++)
  atomic_add(&Tally[index[i]], 1);

for (i = 0; i < n; i++)
  gather[i] = Array[index[i]];
```

Figure 1: Pseudocode for the simplest cases of *histogram* (top) and *index-gather* (bottom). Capitalized symbols denote distributed arrays, while other symbols denote local variables. In particular, `n` can vary across processes.

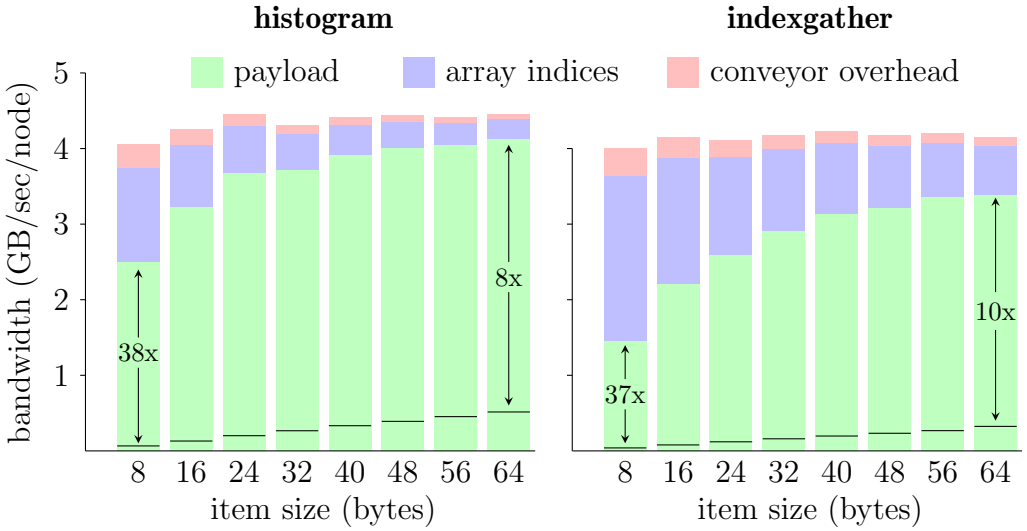

Figure 2: Performance of *histogram* and *indexgather* on a cluster with FDR InfiniBand, using well-tuned conveyors (green) versus individual nonblocking SHMEM puts and gets (horizontal lines). The job scale is 128 nodes with 32 PEs per node, and we measure injection bandwidth per node.

In the second idiom, every process wants to make numerous queries to a distributed data structure and operate somehow on the responses. Typically the queries have no side effects, so they can be handled in any order, and processing responses is a purely local affair. For example, the data structure may again be a distributed array, and each query may simply ask for the value of a specified element. The idiom takes the name *indexgather* from this special case: each query is an index into the array, and the responses are the gathered values. Again, one can often write concise code for *indexgather* (Figure 1), but it is unlikely to perform well because the individual messages are so small.

We emphasize that the *histogram* and *indexgather* examples in Figure 1 are unusually simple in that each desired action reduces to a single PGAS remote access (an atomic add or get). With a more complex distributed data structure, such as a hashmap or a sparse matrix, the relevant actions involve multiple memory accesses. If done via sequences of remote memory operations, the actions become yet more expensive [2]; in particular, when updates are in play, they require locking to avoid data races. Thus a further

reason to use something like conveyors, beyond the benefits of aggregation, is to delegate data structure access to the process that owns the data so that it can serialize access and perform efficient local memory operations.

2.1 Related Work

Years ago one could acquire parallel machines with custom interconnects designed to transmit small messages very efficiently. (The Cray T3E was an early example.) In this environment, PGAS languages such as UPC encouraged a natural coding style, as in Figure 1, that paid no attention to the cost of small transfers. When such code was ported to clusters with commodity interconnect, performance fell off a cliff. The second author found that tools that were expected to run in seconds or minutes were suddenly taking hours. To overcome this problem, he and Dan Pryor and Phil Merkey wrote the `exstack` module, which provides a simple aggregation capability.

An `exstack` object is a parallel data structure in which each process has both an outgoing and an incoming buffer for every other process, including itself. It can `push` items into its outgoing buffers until one fills up, and `pull` (actually "pop") items from its incoming buffers until they are empty. Between these two kinds of operations, the equivalent of an `advance` operation transfers all the data from the outgoing to the incoming buffers in bulk synchronous fashion, similar to `MPI_Alltoallv`. In effect, it performs an out-of-place parallel transpose of a matrix of buffers.

As previously suggested, code that relies on `exstack` objects can become quite ugly. In 2015, DeVinney and Merkey began work on an `exstack2` package with the goal of relaxing the requirements imposed on client code, particularly when driving multiple communication objects simultaneously. One idea of `exstack2` was to use asynchronous communication mechanisms (which really means point-to-point synchronization) to eliminate the global synchronization points that appear in every `exstack` loop. The authors of `exstack2` had reason to hope that asynchronous communication could ultimately be made more efficient than the bulk synchronous style.

Unfortunately, `exstack2` made only limited gains in performance and code clarity, for reasons we will mention in Sections 4.3.2 and 6.1. But it inspired the first author to launch the conveyor project, which looks a lot like a ground-up rewrite of `exstack2`. Part of the plan for conveyors was to unify the application programming interfaces (APIs) of `exstack` and `exstack2`. With a unified API, one can more easily experiment with different

conveyor implementations and can switch from one to another when porting an application to a new platform.

2.2 Scaling Challenges

It turns out that `exstack`, and to a lesser extent `exstack2`, suffers from two problems when scaling to a large number of processes. One problem arises when the average amount of data being transmitted between each pair of processes is small—too small for an efficient message. In this situation, the buffering provided by `exstack` or `exstack2` gains little or nothing. It can even lose performance if the underlying communication function, such as `shmemx_alltoallv` in Cray SHMEM, is poorly optimized for this low-data case. The other problem is simply the amount of buffer memory [1]. If there are 2^{16} processes, for example, and the desired message size is 2^{13} bytes (often it is larger), then a single `exstack` object consumes a gigabyte of memory per process. This quantity can be a substantial fraction of the available memory per core, and may not be affordable. Shrinking the buffers is an option, but it hurts performance.

One way out of this dilemma is to move the entire application to a two-level model of parallelism. Each process may be multithreaded, with the threads coordinating to perform communication in order to increase the memory directly visible to a thread and reduce the number of communicating entities. This change multiplies the average amount of data passing between a pair of processes by the square of the number n of threads per process, and it divides the total amount of buffer memory by at least n if not n^2. (In practice, threads may need private outgoing buffers to avoid contention.) For obvious reasons, however, programmers prefer the convenience of a one-level model of parallelism with one process per physical core or hardware thread.

Addressing the scaling problems became a major goal for the conveyor project. Fortunately, there is a way to gain the benefits of multithreading for many-to-many communication without changing the programming model. The idea is to exploit an optimization that almost every low-level communication library implements: messages sent from one process to another process on the same node are transmitted via shared memory without involving the communication fabric. This feature allows the processes on a node to cooperate efficiently, and thus to function in some ways like a single entity. Messages between processes end up making multiple hops, some of which stay within a node. This idea can benefit both synchronous and asynchronous conveyors.

Although a bit of programming is involved, the complexity of the multi-hop routing algorithm is encapsulated in the conveyor package and is almost invisible to the application programmer. Section 4 explains the details.

A good prior example of multi-hop routing is the Topological Routing and Aggregation Module (TRAM) in Charm++ [5]. It routes messages along the dimensions of an d-dimensional array so that each message takes d hops, and it aggregates messages for efficiency. If the array is adapted to the machine topology, then some of these hops can be fast intranode hops. Compared with this natural scheme, our three-hop routing algorithm (Section 4.3.1) makes greater use of intranode communication, which improves performance while minimizing buffer memory.

3 The Conveyor Interface

We now begin the technical portion of the paper by presenting the conveyor interface in the form of a *contract* between the conveyor and its client. We omit details of the interface such as the prototypes of conveyor functions and the full names of enumerated constants. Such details appear in the documentation and public header file (`convey.h`) of the conveyor package.

Conveyors aim to serve large-scale applications with many processes per node, in which each process is either single-threaded or has only one communication thread. Consequently, conveyor operations are not required to be thread-safe. Thread safety could be useful but is not a current goal.

3.1 Operations

A conveyor is a parallel object. By this we mean that it involves data spread across a set of processes, it has a local state on each process, and some of its operations are *collective*: they must be called concurrently by every process in the set. In particular, creation and destruction of conveyors are collective operations. Conveyors are opaque (encapsulated) objects so that clients cannot peek at their internal representations and thus subvert the contract. A client can only interact with a conveyor by invoking its *methods*, that is, by performing well-defined operations on it.

A conveyor, once created, has seven core methods: `begin`, `advance`, `push`, `pull`, `unpull`, `reset`, and `free`. We have mentioned `advance`, `push`, and `pull` before, but naturally the interface isn't quite that simple. The `begin` and `reset` operations are meant to bracket a loop that pushes items through

the conveyor until all items have been delivered. They offer opportunities for a long-lived conveyor to allocate and deallocate its internal buffers so that they only occupy space while they are in use. For conveyors that transmit only fixed-sized items, `begin` sets the item size. The `free` operation destroys the conveyor. That leaves `unpull`, which allows the client to "put back" the most recently pulled item. Conveyors (following `exstack2`) offer this service as a convenience to their clients, for reasons revealed in Section 3.4.

The `push`, `pull`, and `unpull` operations can succeed or fail, and their success or failure can be observed from their return values. Which operations a process has done, and their return values, determine the local state of the conveyor. The local state, in turn, determines which core operations are legal. An illegal operation always fails; it moves no data and has no effect on the state of the conveyor. It does not crash, but it may have side effects such as printing an error message.

3.2 States and Transitions

A conveyor not only transmits data; it also provides a mechanism for the client processes to determine that all transmissions are complete. As a result, a conveyor can locally be in any of several states. Each client process declares that it will push no further items by invoking the `advance` operation with boolean argument `true`. At this point, the conveyor is said to begin its *endgame*. Once all processes have entered the endgame, the conveyor only has to worry about delivering previously pushed items. When those items have all been received by their destination processes, the conveyor is nearly complete. Some items may remain to be pulled, however, in which case we say the conveyor is in its *cleanup* phase.

Formally, the local states of a conveyor are as follows.

DORMANT: The legal operations are `begin`, `reset` (which does nothing), and `free`. The `begin` operation causes a transition to the WORKING state.

WORKING: The legal operations are `push`, `pull`, `unpull`, and `advance`. If `advance` is given the argument `true`, then the state transitions to ENDGAME, CLEANUP, or COMPLETE according to whether the return value of advance is `ok`, `near`, or `done`. Otherwise `advance` returns `ok` and the state does not change.

ENDGAME: The legal operations are `pull`, `unpull`, and `advance` (with argument `true`). If `advance` returns `near` or `done`, then the state changes to CLEANUP or COMPLETE respectively; otherwise it does not change.

CLEANUP: The legal operations are `pull`, `unpull`, and `advance` (with argument `true`). If `advance` returns `done`, then the state changes to COMPLETE; otherwise it does not change.

COMPLETE: The only illegal operations are `push` and `begin`. The `pull` and `unpull` operations are legal but they fail, and `advance` returns `done`. The `reset` operation changes the state to DORMANT, and `free` destroys the conveyor.

A newly created conveyor begins in the DORMANT state on every process. The client can track the state thereafter by observing the return values from `advance`.

3.3 The Contract

Informally, the main points of the contract are as follows. The client of a conveyor promises to keep calling `pull` and `advance` on every process until `advance` returns `done`. The conveyor promises that repeated attempts to `push` an item will eventually succeed, that every item will be delivered to its intended recipient, that items sent from one particular process to another will arrive in order, and that `unpull` can be used to reverse one `pull`.

The full contract is somewhat complicated by the semantics of `unpull`. We present here a simplified version that assumes `unpull` is never called. The client of a conveyor promises that each process will obey the following rules:

- It will never call `begin` or `reset` or `free` when it is illegal, or in a non-collective way.
- If the conveyor state is neither DORMANT nor COMPLETE, the client will eventually attempt a `pull`.
- If the conveyor state is neither DORMANT nor COMPLETE, then the client will eventually call `advance` with argument `true`.

Provided that the client upholds its part of the bargain, the conveyor promises:

- If the conveyor is in state WORKING, then repeated attempts to `push` an item, interleaved with `advance` operations, will eventually lead to success.
- Every item that is successfully pushed is eventually delivered to exactly one `pull` operation on the desired destination process, prior to the next `reset` or `free`. Conversely, when a `pull` succeeds, the item it delivers is an item that was pushed to that process after the most recent `begin`.
- Items pushed on a particular process for delivery to a given process are delivered in the order that they are successfully pushed. In other words, the channel between each pair of processes is first-in, first-out (FIFO).
- If the conveyor is in state COMPLETE or CLEANUP on any process, then on no process is it in state WORKING. If the state is DORMANT on any process, then it is DORMANT on every process.
- One of the following two properties holds:
  - Every call to `advance` is nonblocking.
  - If the conveyor is in state COMPLETE or CLEANUP on any process, then it is in state COMPLETE or CLEANUP on every process. In this case `advance` is nonblocking. Otherwise the conveyor is in state WORKING or ENDGAME on every process, and `advance` synchronizes all the processes.

The one strange clause says that for each conveyor, `advance` is either collective and synchronizing (until interprocess communication is finished) or always nonblocking. The distinction is unimportant when only a single conveyor is active, but when a client uses two or more conveyors simultaneously, it must take care to advance each one in a collective way, lest they deadlock.

3.4 Examples of Use

Figures 3 and 4 illustrate the use of conveyors to implement the *histogram* and *indexgather* idioms, respectively. For concreteness, we assume that the shared arrays `Tally` and `Array` from Figure 1 are distributed in round-robin fashion across `PROCS` processes, as is typical in UPC, and we denote the local slice of such an array by a corresponding lowercase name (`tally` or `array`).

```
convey_begin(c, sizeof(long));
long spot, i = 0;
while (convey_advance(c, i == n)) {
  for (; i < n; i++) {
    spot = index[i] / PROCS;
    if (! convey_push(c, &spot, index[i] % PROCS))
      break;
  }
  while (convey_pull(c, &spot, NULL))
    tally[spot]++;
}
convey_reset(c);
```

Figure 3: C code to implement the *histogram* idiom from Figure 1 using a conveyor `c` that transmits items of type `long`. Construction and destruction of `c` are not shown.

We streamline the code by not checking the return values of conveyor methods for errors. Such errors could only indicate memory exhaustion or a fatal bug in either the client code or the conveyor library.

Figure 3 shows the recommended structure of a loop nest that drives a single conveyor. Each of the functions `convey_advance`, `convey_push`, and `convey_pull` returns a nonzero value on success and zero upon failure or termination. For correctness, the second argument of `convey_advance` must be `true` if and only if the `push` operations are complete, and the `advance`, `push`, and `pull` operations must be performed repeatedly until `convey_advance` returns zero. For efficiency, because `convey_advance` can be expensive, it is best to `push` and `pull` many items within the outer loop. The loop nest carefully maintains the state variable `i`, which represents the number of indices successfully pushed.

Figure 4 exhibits a loop nest that drives two conveyors, one for queries and one for responses. This example also illustrates two more features of the conveyor API. First, a successful call to `convey_pull` can provide the caller with the rank of the process that pushed the pulled item. In this example, the recipient of a query needs to know where to send the response. Second, if the response cannot be pushed, the query can be deferred by means of `convey_unpull`. Without this privilege, the application code would need to keep track of an unsatisfied query and would become yet more complicated.

```
struct packet { long slot; long value; } packet;
int64_t i = 0, from;
while (convey_advance(r, !convey_advance(q, i == n))) {
  for (; i < n; i++) {
    packet.slot = i;
    packet.value = index[i] / PROCS;
    if (! convey_push(q, &packet, index[i] % PROCS))
      break;
  }
  while (convey_pull(q, &packet, &from)) {
    packet.value = array[packet.value];
    if (! convey_push(r, &packet, from)) {
      convey_unpull(q);
      break;
    }
  }
  while (convey_pull(r, &packet, NULL))
    gather[packet.slot] = packet.value;
}
```

Figure 4: C code to implement the *indexgather* idiom from Figure 1 using conveyors `q` (for queries) and `r` (for responses) that both transmit items of type `struct packet`. Construction and destruction of `q` and `r` are not shown; neither are their `begin` and `reset` operations.

The trickiest part of the *indexgather* code in Figure 4 is the condition in the outer `while` loop. This incantation ensures that `convey_advance` is called on both conveyors, first on `q` and then on `r`, so that if these operations are collective, they will be called in the same sequence on every process. It also indicates that the response conveyor is not finished pushing until the query conveyor is COMPLETE, implying that this process will have no more queries to handle.

3.5 Error Handling

We do not insist that conveyor clients refrain from attempting illegal non-collective operations. Instead, a conveyor is required to track its state on each process so that it can detect and reject all such illegal operations. If the

client misuses collective operations, however, the program may well deadlock. A conveyor may detect such misuse and abort the program instead, but is not required to do so.

As noted in Section 3.4, the conveyor API uses a nonstandard convention for return values. When a conveyor method returns an `int`, as many do, a positive value indicates some kind of success, zero represents an ordinary failure, and a negative number represents a severe error, usually indicating that the API was misused in some way. If a client is not worried about catching and reporting its own bugs, then it can treat the return value as a boolean with `true` meaning success (or "go") and `false` meaning failure (or "stop"). This convention improves the readability of the client code.

Conveyors report misuse and other severe errors automatically by printing appropriate error messages, while trying to suppress duplicate messages. If a client prefers, it can compare every return value to zero and treat positive and negative results differently, and it can switch off the automatic error messages (see Section 4.6).

3.6 Optional Features

To support a greater variety of parallel algorithms, including some of the "apps" in `bale`, conveyors now offer two optional features. One feature is the ability to send items of varying sizes, including items much larger than a typical buffer. Another is the ability to guarantee delivery of all pushed items without having to enter the endgame. Partial support is also available for a third feature, automatic compression of internode messages, and additional features may be added in the future.

The presence or absence of a feature is determined when a conveyor is created and is immutable thereafter. The conveyor interface includes a function `convey_features` that a client can use to determine which features are present. The key point is that a feature can only strengthen the guarantees that a conveyor offers to its client; it cannot weaken them. Consequently, code that relies on a conveyor will continue to work, though possibly with reduced performance, if handed a conveyor with fancy new features.

Variable-length items. A conveyor that can transport variable-length items is called *elastic*. To support elastic conveyors, the conveyor interface includes "elastic" push and pull functions called `convey_epush` and `convey_epull`. In terms of the contract, these are just `push` and `pull` operations, but of course they have more elaborate prototypes than `convey_push`

and `convey_pull`. The elastic push takes an additional argument, namely the item size; the elastic pull, when successful, delivers this size to its caller. The item size can be zero, because receiving an empty message from a particular process can be meaningful.

Asking a non-elastic conveyor to perform an elastic push or pull is an error. But an elastic conveyor supports ordinary pushes and pulls using the default item size set by `begin`. When `convey_pull` is used with an elastic conveyor, it succeeds if the next item to pull has the default size, and otherwise it fails.

Steady progress. A typical conveyor stores pushed items in internal buffers and does not flush a buffer until either the buffer fills up or the conveyor enters the endgame. A *steady* conveyor, in contrast, never holds onto pushed items indefinitely. With a steady conveyor, as long as every process continues performing `advance` and `pull` operations every pushed item will eventually be delivered. With a steady conveyor, an algorithm can wait to see what items the conveyor delivers before deciding what to push into the conveyor. If such an algorithm were used with an ordinary conveyor, it would be prone to deadlock.

Compression. When constructing a conveyor, one can request that it support automatic buffer compression, either via user-supplied compression and decompression functions or via built-in functions. Whether the request is honored will depend on the type of conveyor and the item size. Currently no elastic conveyors support compression. The built-in "codec" is very simple, as it must be to keep up with a high-performance interconnect. Before transmitting a buffer over the network, it looks for a pattern of constant bits or bytes within its items, and it squeezes out these bits, sending them only once rather than as part of every item. Such constant bits arise, for instance, when sending unsigned integers (like array indices) many of whose high bits are zero. If this type of compression fails to shrink the buffer significantly, then the buffer is sent in uncompressed form.

4 The Conveyor Package

The conveyor package provides four different conveyor implementations, and it can be built atop any of six different substrates. Three of the substrates are SHMEM, the UPC language, and MPI, while the other three involve a library called `mpp_utilV4` that is not distributed with `bale`. The choice of

substrate is made when the conveyor package is configured as part of the build process, and it determines which types of conveyors are available. All types but one are supported by all six substrates.

Building the conveyor package results in the installation of the public header file `convey.h` and the library `libconvey.a`. To avoid complications in the build process for the conveyor package and client applications, no shared object is built.

The architecture of the conveyor package is object-oriented. A conveyor is an opaque object that, privately, points to a table of function pointers that provide implementations of its methods. A client invokes a conveyor method by calling a dispatch function such as `convey_advance`, which relays the arguments to the internal method while also performing some error checking and state tracking. This design allows several types of conveyors to coexist in one application, and it allows the application or the library to choose the most appropriate type for each algorithm at run time. The library provides a specialized constructor for each type of conveyor, but it also provides generic constructors `convey_new` and `convey_new_elastic` that try to choose the most efficient conveyor from those available.

4.1 Simple Conveyors

We begin by describing *simple* conveyors, which use the bulk synchronous scheme described in Section 2.1. On each process, a simple conveyor keeps an array of outgoing buffers and an array of incoming buffers, each of which holds a fixed number of fixed-sized items. A `push` appends its item to the appropriate outgoing buffer if there is room for it, and otherwise fails. The `pull` method delivers items from the incoming buffers, starting with the buffer from the lowest-numbered process and proceeding from start to end within each buffer (to ensure FIFO behavior). The `advance` method determines whether the processes have nothing left to pull, and if so, whether they have anything to send. The latter condition is true if any outgoing buffer is full, or if the conveyor is steady (see Section 3.6) and any outgoing buffer is nonempty. If all the incoming buffers have been drained and some items need to be sent, then the stored items are transferred from the outgoing buffers to the incoming buffers using a mechanism appropriate to the substrate.

The bulk communication mechanism can be either `MPI_Alltoallv` or `shmemx_alltoallv`, which is a Cray SHMEM extension. Otherwise the all-to-all transfer is synthesized from calls to `shmem_putmem` or `upc_memput`,

looping over destination processes in a pseudorandom way.

When the number of processes is very large, scattering items into outgoing buffers can account for a significant fraction of the cost of a conveyor loop, due to cache thrashing. Simple conveyors therefore offer an option called `SCATTER` to accelerate the `push` operation by means of prefetching. When this option is chosen, a `push` does three things: it copies the item into a small circular buffer, prefetches (with write intent) the relevant portion of the item's outgoing buffer, and, if necessary, copies an older item from the circular buffer to its place in an outgoing buffer. This trick does not eliminate cache thrashing but it overcomes the latency of cache misses. The first author also experimented with a more elaborate scheme that sorts pushed items into outgoing buffers through a tree of intermediate buffers. Although more cache-friendly, this scheme seemed to offer little additional benefit on recent CPUs, and it was difficult to know how to configure its internal parameters for a given platform. So this code has been switched off.

The main difference between simple conveyors and `exstack` is that the internal structure and, more importantly, the exact behavior of an `exstack` object are exposed to the client. This transparency makes it possible to use `exstack` in clever ways that the conveyor interface does not support. The resulting convenience sometimes outweighs the efficiency benefits that more-complex conveyors provide. We discuss this tradeoff in Section 6.1.

4.2 Two-Hop Synchronous Conveyors

The next type of conveyor, called `twohop`, alleviates the scaling problems of simple conveyors at the cost of transmitting somewhat more data. A `twohop` conveyor arranges the processes in a two-dimensional array, say with m rows and n columns. Each item first travels within its row to the column of its destination process, and then it travels within that column to its destination. The intent is that the processes within a row should lie on the same node, which is normally automatic if n divides the number of processes per node. One then hopes that the underlying library optimizes intranode communication. For communication, the conveyor uses either `MPI_Alltoallv` or the Cray SHMEM extensions `shmemx_alltoallv` and `shmemx_team_alltoallv`. UPC is not supported, and neither is non-Cray SHMEM.

To route each item, the conveyor prepends a 32-bit tag to the item before placing it into its buffer. Initially this tag holds the destination row and the source column. The conveyor keeps separate sets of buffers for transfers

within rows and transfers within columns. When the conveyor *pivots* (moves) an item from an incoming row buffer to an outgoing column buffer, it uses the tag to determine where to send the item. Then it modifies the tag to hold the identity of the source process, which it assembles from the source column (part of the old tag) and the source row (in which the pivot takes place). Thus `convey_pull` will be able to report the process that sent the item. A simple conveyor, in contrast, need not transmit this information because it associates each incoming buffer with a unique source process.

The two-hop routing algorithm has three benefits. First, it reduces the total number of buffers per process from $2mn$ to $2m+2n$. If n is 32, currently a common value for the number of cores per node, and if $m \geq n$, then the memory savings ranges from a factor of 16 to a factor of 32. Second, because there are fewer buffers, scattering into buffers (among other operations) is more efficient. Third, when the load and the item size are small, the items are aggregated into larger messages and fewer messages are sent overall, which greatly improves performance.

Early in the conveyor project, experiments indicated that `twohop` conveyors dominate both simple conveyors and the now-obsolete `sparse` conveyors (see Section 4.5) over a wide range of load values, provided that the item size was large. For items of 8 or 16 bytes, however, the cost of sending the routing tag and perhaps the work of pivoting gives the advantage to simple conveyors when the load is high. The reason that synchronous multi-hop conveyors have not been pursued further, and in particular have not been extended to UPC and portable SHMEM, is that a similar but asynchronous design was found to perform even better.

4.3 Multi-Hop Asynchronous Conveyors

Conversations with a Cray engineer [6], along with benchmark data he provided, suggested that the most efficient way to use the Cray Aries interconnect was to send messages using only point-to-point synchronization. It was intuitively plausible that global synchronization could cause unwanted delays, as processes that happened to fill a buffer early waited for those that did not. Furthermore, if each process could do communication work on its own schedule, then some processes on a node would be communicating while others were computing. The net effect would be to overlap communication with whatever local work was needed to construct items for pushing and to digest pulled items. This effect might not show up in benchmarks but should

be helpful in real applications. It could perhaps be enhanced further by using nonblocking communication primitives.

The most important contribution of the conveyor package is the construction of conveyors that combine asynchronous communication with multi-hop routing. Officially we call them `tensor` conveyors; in the source code, they are known as `vector`, `matrix`, or `tensor` conveyors according to whether they treat the processes as forming a one-dimensional, two-dimensional, or three-dimensional array. Individual items take one, two, or three hops respectively. In a three-hop tensor conveyor, for instance, an item first travels within its source node, then across the network to its destination node, and finally within that node to its destination process.

When the number of processes is large and there are many processes per node, the three-hop tensor conveyors generally provide the best performance of any application-level communication mechanism yet devised. They also use the least memory for buffers. Tensor conveyors have only two real drawbacks. They pay the bandwidth cost of transmitting routing tags, although their other efficiency advantages usually more than compensate. And in order to work optimally, they sometimes need to be told the number of processes to group together for local hops. Tensor conveyors attempt to compute an appropriate group size from the overall number of processes and the number of processes per node, but this computation does not always yield the optimum value.

4.3.1 Architecture

A tensor conveyor is built out of one, two, or three subsidiary objects called *porters*. The porter API is similar to the conveyor API but has a few important differences:

- On each process, the set of processes that a porter communicates with is specified explicitly, via an array of process numbers, when the porter is constructed.
- A porter's communication pattern forms a collection of complete graphs and complete bipartite graphs, rather than one large complete graph.
- Rather than pulling individual items, the client of a porter pulls an entire buffer of items at once. The client must understand the format of this buffer.

- Porter methods do very little error checking; they assume the caller has done what is needed.

The job of a tensor conveyor is to orchestrate the operations of its porters and to transfer items from each porter to the next. The `convey_push` operation, for instance, essentially just pushes the item into the first porter. The `convey_advance` operation advances each porter and, for each porter but the last, it tries to pull a few buffers and push their items into the next porter. It avoids dwelling too long on any one porter so that the flow of items through the system does not develop any bottlenecks.

Three-hop routing is somewhat interesting. The only precondition is that the number n of processes per local (intranode) group divides the total number of processes p. As with two-hop conveyors, we assume that processes are numbered consecutively within nodes. A tensor conveyor writes each process number r in mixed-radix notation (x, y, z) where $0 \le y, z < n$ so that $r = n^2x + ny + z$. The internode links from process (x, y, z) connect it with the processes (x', z, y) for all valid x'. A message from (x, y, z) to (x', y', z') travels through the intermediate processes (x, y, y') and (x', y', y). As it goes, its routing tag is updated to replace coordinates of the destination process with coordinates of the source process so that, upon arrival, the tag determines the identity of the source. Writing tags in brackets, we can describe the routing thus:

$$(x, y, z) \xrightarrow{[x', z', z]} (x, y, y') \xrightarrow{[x, z', z]} (x', y', y) \xrightarrow{[x, y, z]} (x', y', z').$$

The first and third hops connect processes whose numbers differ only in their last coordinate, so they should involve only intranode communication. Because n divides p, both (x, y, y') and (x', y', y) are guaranteed to exist if (x, y, z) and (x', y', z') exist.

The routing tags actually transmitted are simpler than the ones shown above, because one field of the tag is redundant with the identity of the sending process. Vector conveyors need no routing tags at all; matrix conveyors can usually shrink the routing tags for internode hops to 1 byte; and three-hop tensor conveyors can shrink them to 2 bytes, or to 1 byte if $n \le 16$. Thus the bandwidth cost for multi-hop routing becomes relatively small.

4.3.2 Low-Level Mechanisms

It remains to explain how porters work. We focus first on the SHMEM case, which affords the greatest range of possibilities.

Remote atomics. The mechanism used by `exstack2` works as follows. Each process has an array of outgoing buffers and an array of incoming buffers as usual. It also has a ring (circular buffer) for incoming *signals*. To send a buffer, the source process first checks a locally present but globally visible flag to see whether its buffer at the destination is available. If so, it writes into that buffer via a blocking put operation (actually `upc_memput` followed by `upc_fence`), then performs a remote atomic fetch-and-increment to obtain a location in the target's ring, and finally puts a short signal message into this ring. The signal encodes the source process, the amount of valid data in the buffer, and an indication of whether no further buffers will be sent from this source. The recipient can very easily look at its ring to see which buffers have arrived. After it pulls all items from an incoming buffer, it remotely sets the corresponding flag on the sending process to indicate that the buffer is free again.

We chose not to use this mechanism for porters because waiting for the atomic operation, on top of the blocking put, seemed inefficient. On some platforms, particularly those with commodity interconnect, the cost of atomic operations can be painfully high.

Signaling puts. Another attractive mechanism is based on *signaling puts*, which are currently available only in Cray SHMEM. A signaling put does two things with a single library call: it sends a buffer of data to a destination process, and when all the data has arrived, it writes a specified value into a specified location at the destination. The recipient must somehow poll these signal locations to see which puts have completed.

In a porter, each process keeps an array of words for incoming signals. A single signaling put fulfills the responsibilities of a buffer sender. The signal it sends is a count of the number of buffers that have now been sent from that source to that destination, together with a termination flag. The recipient tracks the number of buffers it has received from each sender, and by comparing these numbers with the incoming signals, it can see whether anything new has arrived. Upon disposing of an incoming buffer, it puts a small message back to the sender to update a count of consumed buffers. The sender uses such counts to determine which destination buffers are available.

Ideally, a signaling put would be handled almost entirely by the communication hardware. In particular, one would like for the signal to travel across the network along with the bulk of the data and be delivered automatically at the appropriate moment. So far this hope has not been realized. Experiments on Cray XC systems have shown that, in the context of porters, a blocking signaling put is no better than a put of the buffer, a wait for the put to complete, and then a put of the signal. For odd reasons, nonblocking signaling puts seem to perform even worse. Current porters therefore synthesize a signaling put as a `shmem_putmem` of the buffer, followed by a `shmem_fence`, followed by a `shmem_put64` of the signal (or the UPC equivalents of these operations).

This mechanism, based on portable and efficient library functions, works surprisingly well. Polling of signals turns out be very cheap, and forcing processes to wait for their large puts to complete is not harmful as long as there are plenty of processes per node.

Nonblocking puts. If waiting for puts to complete works well, then nonblocking puts should work even better. A nonblocking put function can simply tell the communication system which bytes to send where, and then return to the caller before copying anything out of the send buffer. The system can transfer the data asynchronously, while the caller must only beware not to overwrite the send buffer until it knows that the put is complete.

A porter that uses nonblocking puts must know when a put is complete so that it can send a corresponding signal. Unfortunately, the latest version (namely 1.4) of OpenSHMEM, and even Cray SHMEM, provide no way to test or wait for particular nonblocking puts to deliver their data. Instead, one must call `shmem_quiet`, which waits for all outstanding puts from this process to complete. So the porter must choose appropriate moments at which to call `shmem_quiet`. Having kept track of the nonblocking puts in flight, it can then send the corresponding signals and clear its list of outstanding puts.

Porters of this type have been implemented and tested in combination with the mechanism described next. The performance of the resulting tensor conveyors improves significantly when there are relatively few processes per node (such as 4 or 8 processes on a node of 32 cores), and seems to improve slightly even when there is a process for every core.

Local atomics. An awkward aspect of using `shmem_quiet` is that porters now interact with one another behind the scenes. In particular, if we rely on SHMEM optimizations to make local (intranode) porters run fast, then the

`shmem_quiet` operations in one porter are forced to wait unnecessarily for SHMEM operations that every other porter has in flight. This phenomenon may not hurt conveyors very much, but we can ameliorate it by using an entirely different mechanism for local porters, one that takes intranode optimization into our own hands. As a bonus, this mechanism may slightly streamline the code paths taken by local porters.

We exploit the `shmem_ptr` function, or the equivalent UPC function `upc_cast`. This function tries to turn a remote memory reference into an ordinary pointer that can be read and written directly. If the remote reference involves memory on the same node, then `shmem_ptr` should work; otherwise it returns `NULL`. Once we have the necessary pointers, a signaling put can be implemented as a `memcpy` of the buffer followed by a C11 atomic store to the signal location. We rely on the C11 memory model to ensure that the results of the `memcpy` are visible before the atomic store takes effect. (So far, this seems to work even though the parallelism involves multiple processes rather than multiple threads within a process.) Presumably, our original mechanism involving `shmem_put` and `shmem_fence` would do something quite similar in the intranode case, but the fence would also interact with nonblocking puts from other porters, which we do not want.

MPI nonblocking sends. Porters built on an MPI substrate use a completely different mechanism, because even in MPI-3, one-sided communication remains highly problematic. (In the first author's experience, the recipients of one-sided puts must make calls into the MPI library to ensure forward progress, which negates much of the benefit.) Fortunately, MPI supports a full suite of nonblocking send and receive operations that have been around since MPI-1, so they should be very portable.

To send a buffer, an MPI porter initiates a nonblocking send by calling `MPI_Isend`, passing it the buffer, its length, the intended destination, and a message tag unique to the porter. The caller gets back a request handle that it can use to query whether the send is complete, which implies that the buffer can be reused. On the receiving side, the porter maintains an outstanding `MPI_Irecv` for each source. Again, it gets back a request handle. When a pull operation wants a new buffer of items, it calls `MPI_Test` on this handle to see whether a new buffer has arrived. As usual, there are some details associated with the endgame: the header of each transmitted buffer includes a flag to indicate whether this transmission is the last one from its source, and the recipient uses these flags to determine when to stop calling `MPI_Irecv`.

No explicit signal messages are sent in either direction, because `MPI_Test` supplies the necessary information.

In our testing so far, MPI porters achieve a reasonable fraction of the performance of SHMEM porters, but their memory usage has not been evaluated. The MPI library must do a certain amount of buffering internally. We do not attempt to control this internal buffer space directly, but merely limit the number of outstanding sends between each pair of communicating processes, and hope this suffices. The current limit is one send in each direction.

4.3.3 Porter Summary

Currently, a tensor conveyor chooses its porters as follows. In MPI mode, it must use the MPI porter implementation for all porters. Otherwise, it typically uses the standard SHMEM or UPC porter based on puts and fences. But if a SHMEM nonblocking put is available, then nonlocal porters may use this mechanism instead, depending on how the package has been configured and tuned. Likewise, local porters use an optimized code path if C11 atomics are available and they are able to obtain all the necessary local pointers to shared memory via `shmem_ptr` or `upc_cast`.

The primitive operations used by porters are quite simple and widely available. Because no porter performs collective operations within the connected components of its communication graph, it need not gather processes into teams or communicators.

A wrinkle not previously mentioned is that most porters support multiple buffers per link. The multiplicity must be a power of two, and realistically should be 1, 2, or 4. The benefit of multi-buffering is that pushes can proceed into one buffer while other buffers are either in transit or waiting for the corresponding destination buffers to be consumed. In practice, multi-buffering almost always improves performance to some degree, so the number of buffers per link defaults to 2 (and is fixed at 2 in the MPI case).

4.4 Elastic Conveyors

Originally, conveyors dealt only with fixed-sized items, which kept their internals simple. Elastic conveyors, however, need to transmit a size along with each item, and they also need some way to deal with extra-large items. These requirements cannot easily be satisfied without writing a lot of new

code. Since tensor conveyors generally dominate the synchronous types, the only elastic conveyors developed so far are asynchronous. Their official name is `etensor` conveyors, but we refer to them here simply as elastic.

An elastic conveyor contains a tensor conveyor along with some additional buffers and state. It does not inherit any methods from the tensor conveyor, but it uses some of the tensor methods, notably `begin`, `reset`, and `advance`, to set up and tear down buffers and to make progress. It also modifies the tensor conveyor by replacing the pivot operations that move packets from one porter to the next. In a normal (rigid) tensor conveyor, each packet consists of a routing tag and a fixed-length item. In the elastic case, a packet contains a 32-bit routing tag, a 32-bit descriptor that mainly specifies the item size, and then an item of the appropriate size, padded to a multiple of four bytes. Naturally the pivot functions must understand this new format.

Porters also need to understand variable-length items, but fortunately no ramification into subtypes is needed. Instead every porter is required to support a new elastic push operation (`porter_epush`), which takes a descriptor as well as a tag, item, and destination. The porter API is otherwise unchanged. A porter may assume that its client will only call `porter_push` or `porter_epush`, not both. (Otherwise the client could hardly know how to interpret the resulting buffers.)

4.4.1 Monster Items

The functionality just described suffices to send variable-length items that fit in the buffers of the tensor conveyor. Larger items, known as *monster* items, require an additional mechanism. The elastic conveyor maintains on each process an outgoing and an incoming buffer of sufficient size to hold the largest item that it will ever transmit. This size must be specified when the conveyor is created. When the client attempts to push a monster item, the conveyor first checks whether the outgoing buffer is available. If so, it copies the item to the outgoing buffer and then sends a *ticket* through the tensor conveyor to the destination. The ticket informs the recipient that a monster item of a particular size is waiting at the sending process. Because the ticket carries the size as an 8-byte payload, monster items can be larger than 2^{32} bytes (except on MPI, where the limit is $2^{31} - 1$ bytes due to long-standing issues with the MPI interface that are annoying to work around). When a `pull` operation at the destination encounters the ticket, it uses `shmem_getmem` or `upc_memget` to copy the monster item into the local incoming buffer. Then

it signals receipt of the item by atomically incrementing a shared counter located at the sender. The MPI code for monster items is equally simple: the sender issues a `MPI_Isend` with a special "monster" tag, and the receiver issues a matching `MPI_Recv` upon seeing the ticket.

Conveying tickets is essential for maintaining first-in first-out behavior between the sender and the recipient, but it creates a problem. The sender of a monster item must ensure that its ticket eventually arrives at its destination even if the buffer containing the ticket never fills up. Otherwise, when the client attempts to push another monster item, the push will fail and no progress will be made, no matter how many `advance` and `pull` operations are done.

The chosen solution involves a further enhancement of porters. The descriptor passed to `porter_epush` encodes not only the item length but also a flag that indicates whether the item is a ticket. When a ticket is pushed, it terminates its outgoing buffer, as if that buffer had filled up. The buffer is then marked as *urgent*. Subsequent calls to `porter_advance` attempt to send all urgent buffers, along with any buffers that precede them. This behavior persists until the number of urgent buffers drops to zero.

Elastic conveyors were designed merely to handle monster items correctly, not efficiently. Nonetheless, preliminary benchmarks show that the loss of throughput caused by monster items can be minimal, probably because transferring a large buffer is often an efficient operation.

4.5 Obsolete Conveyors

Early on, the conveyor package included so-called `sparse` conveyors designed specifically for the case where number of processes is large but the load is much less than one item per pair of processes. The mechanism used by a sparse conveyor was similar to that of `exstack2`, but rather than using a remote atomic increment to help send a signal message, it used the atomic increment to obtain space in a circular buffer on the destination, and then performed a signaling put (either native or synthesized) of the data. It sent one item at a time without any buffering on the sending side. This mechanism tended to outperform all others as the load went to zero, but the benefit was so marginal that, after tensor conveyors were developed, sparse conveyors were deemed not worth maintaining and were withdrawn from the package.

4.6 Options, Alignment, and Allocation

Every constructor provided by the conveyor package supports a certain set of options. The options common to all the conveyors, and their meanings, are the following:

`ALERT` Print warning messages if the conveyor detects any unexpected condition that may reduce performance.

`COMPRESS` Turn on data compression, if possible, for non-local hops.

`DYNAMIC` Allocate large internal buffers on every `begin` and release them on every `reset`, rather than holding them throughout the conveyor's lifetime.

`NOALIGN` Do not attempt to align item pointers returned by "aligned pull" operations, perhaps saving some work.

`PROGRESS` Make the conveyor *steady*: see Section 3.6.

`QUIET` Suppress printing of errors and warnings; just return error codes.

`RECKLESS` Turn off some error checks—for instance, that the process number passed to `push` is in range—to gain speed.

The alignment option concerns an awkward but minor aspect of the conveyor API. The `convey_pull` operation copies the pulled item into a caller-provided location and returns a status code. Sometimes one would like to eliminate this copy operation, so the API also provides an "aligned pull" function `convey_apull` that just returns a pointer to the pulled item. Such a pointer needs to be properly aligned for the item, and unfortunately the conveyor does not know the type of an item as a C or C++ value. All it knows is the item size, or in the case of an elastic conveyor, a basic size that divides the size of every item. By default, therefore, `convey_apull` guarantees the strictest alignment that any item of the basic size may require. If the basic size is n bytes, then a pointer to an item may need to have its k lowest bits clear where 2^k is the largest power of two that divides n, up to some maximum. (Usually 16-byte alignment suffices for all standard types, while 32-byte or 64-byte alignment may be needed for non-portable SIMD or vector types.)

Simple conveyors can guarantee alignment naturally: if a buffer is strongly aligned and holds nothing but an array of fixed-sized items, then all items are properly aligned. But the multi-hop conveyors, which insert routing tags and perhaps descriptors, cannot naturally guarantee proper alignment. To ensure the correctness of client code, their `apull` methods must often copy each pulled item into an aligned internal buffer, which defeats the purpose of the operation.

The solution adopted for now is to provide constructor options that let the client define the required alignment for the item type. In addition to the `NOALIGN` option, there is an `ALIGN` option implemented as a macro that takes a power-of-two argument. These options permit the conveyor to omit the unwanted copy in some cases. For instance, the `NOALIGN` option can be appropriate on platforms like x86-64 where alignment of standard integer types is not required at the instruction level.

Another argument common to all the constructor functions relates to memory allocation. The argument points to a structure containing function pointers for allocating remotely accessible memory and for releasing it. The conveyor will call these functions collectively. If the argument is `NULL`, then the conveyor uses standard symmetric or shared allocation functions instead, such as `shmem_malloc` or `upc_all_alloc`. One reason to override these functions is to optimize frequent allocation and deallocation, as may occur when the `DYNAMIC` option is chosen. For instance, one can plug in an allocator library that can carve up a preallocated region of symmetric memory without requiring synchronization or other expensive operations. For maximum compatibility with such allocators, conveyors make sure to perform all their internal allocations in last-in first-out fashion.

5 Benchmark Results

In this section we provide some representative benchmark results for simple and tensor conveyors using both SHMEM and MPI. We do not present any results using UPC, but in the past, UPC performance has been comparable to SHMEM performance on systems that support both. We used three different parallel computers:

- a large Cray XC40 with Aries interconnect and Cray SHMEM 7.7.4;
- a large Cray cluster with a Mellanox FDR InfiniBand network and OSHMEM 4.0.0;

- a commodity cluster with a 100 Gbps Intel Omni-Path network and OpenMPI 3.1.

In each case, the nodes were dual-socket with 16 Intel x86-64 (Haswell or Skylake) cores per socket running at a nominal speed of 2.1 to 2.3 GHz, and we used one process per core (32 per node). The larger systems are almost never idle, so our benchmarks must share those systems with other work. The smaller system was idle when we ran our benchmarks.

Before gathering data on a machine, we used the autotuning feature of the conveyor package to choose a standard buffer size and, in the SHMEM case, to decide whether tensor conveyors should use blocking or nonblocking puts. The standard buffer size is optimized for 2-hop asynchronous conveyors on either 64 or 128 nodes. For the simple conveyors, we optimized the buffer size by hand. Then, for each combination of a conveyor type and node count, we launched a job that iterated over the item size and a few choices for conveyor parameters such as the number of hops (1, 2, or 3) and the size of a local group (either a node or a socket). For simple conveyors, the only parameter is the presence or absence of the `SCATTER` option. The reported results are obtained by maximizing over these parameters.

In an experiment where the conveyor and item size are given, each PE sends 256 MiB of data, distributed randomly to the other PEs. We perform one warm-up experiment and then average the outcomes of three subsequent experiments, where the outcome is the average injection bandwidth per node (2^{28} bytes divided by the elapsed time from `convey_begin` to `convey_reset`). On busy machines, we find that some jobs perform markedly worse than other jobs of similar size, perhaps due to unfortunate job placement or interaction with other network traffic. So on those systems we sometimes ran each job twice and ignored the worse job in each pair.

Figures 5, 6, and 7 display the results. Each graph has the same form: the horizontal coordinate is the base-2 logarithm of the number of nodes, and the vertical coordinate is the injection bandwidth per node. The results for simple conveyors are plotted in green, while the results for tensor conveyors are plotted in black. The data points (squares or circles) for a given item size are joined by lines that are dotted, dashed, long-dashed, or solid, according to whether the item size was 8, 16, 32, or 128 bytes.

The benchmark graphs illustrate the typical performance advantage of multi-hop asynchronous conveyors over single-hop bulk synchronous ones. They also reveal the cost of handling many individual small items. Some of

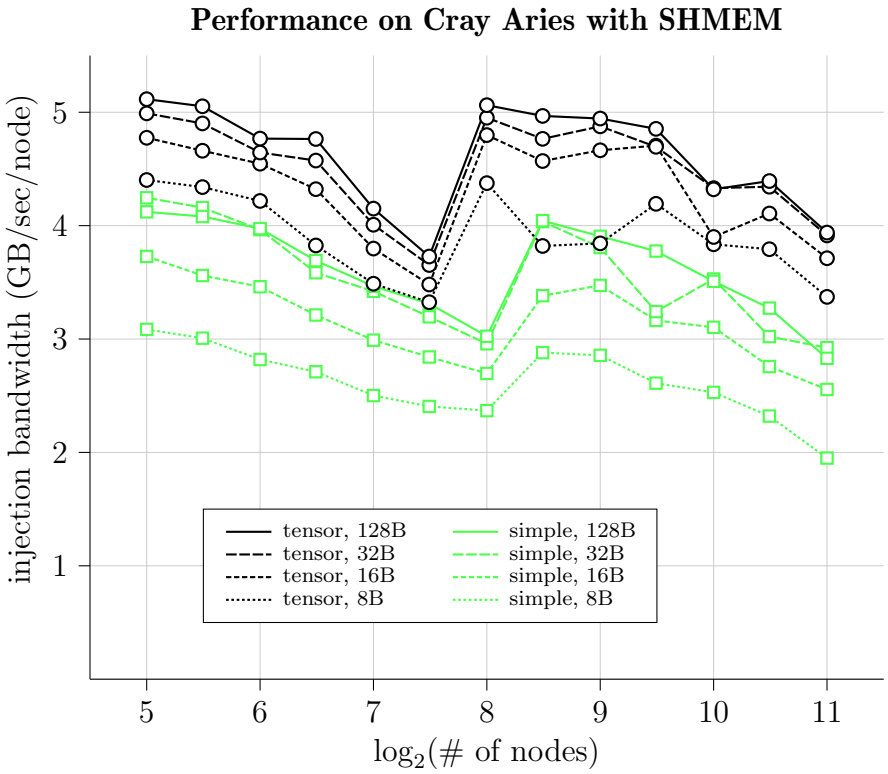

Figure 5: Comparing tensor conveyors (black) and simple conveyors (green) on a Cray XC40. Each curve corresponds to an item size. Simple conveyors use `shmemx_alltoallv` for bulk communication, while tensor conveyors use nonblocking SHMEM puts. The performance loss often seen for node counts in the range 2^6 to 2^8 is thought to arise when a job is placed within a single "group" of nodes, which is an electrically connected pair of cabinets.

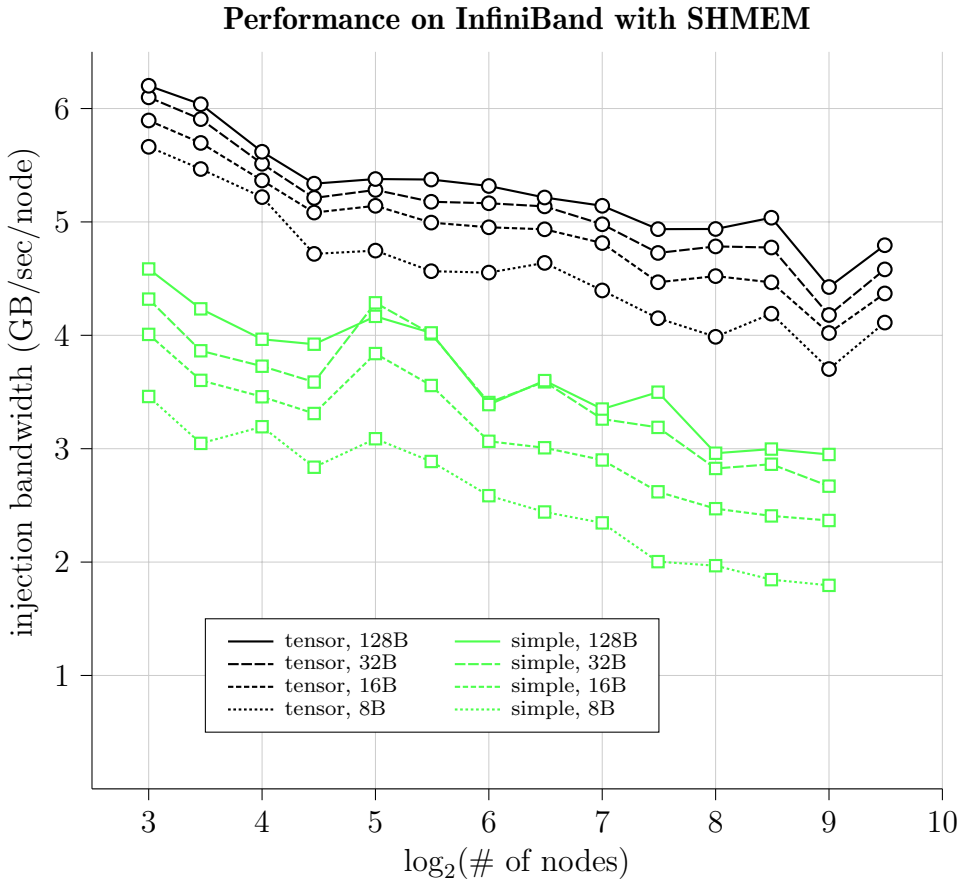

Figure 6: Comparing tensor conveyors (black) and simple conveyors (green) on a cluster with a Mellanox InfiniBand fabric. Each curve corresponds to an item size. For bulk communication on this system, both types of conveyors rely on blocking SHMEM puts.

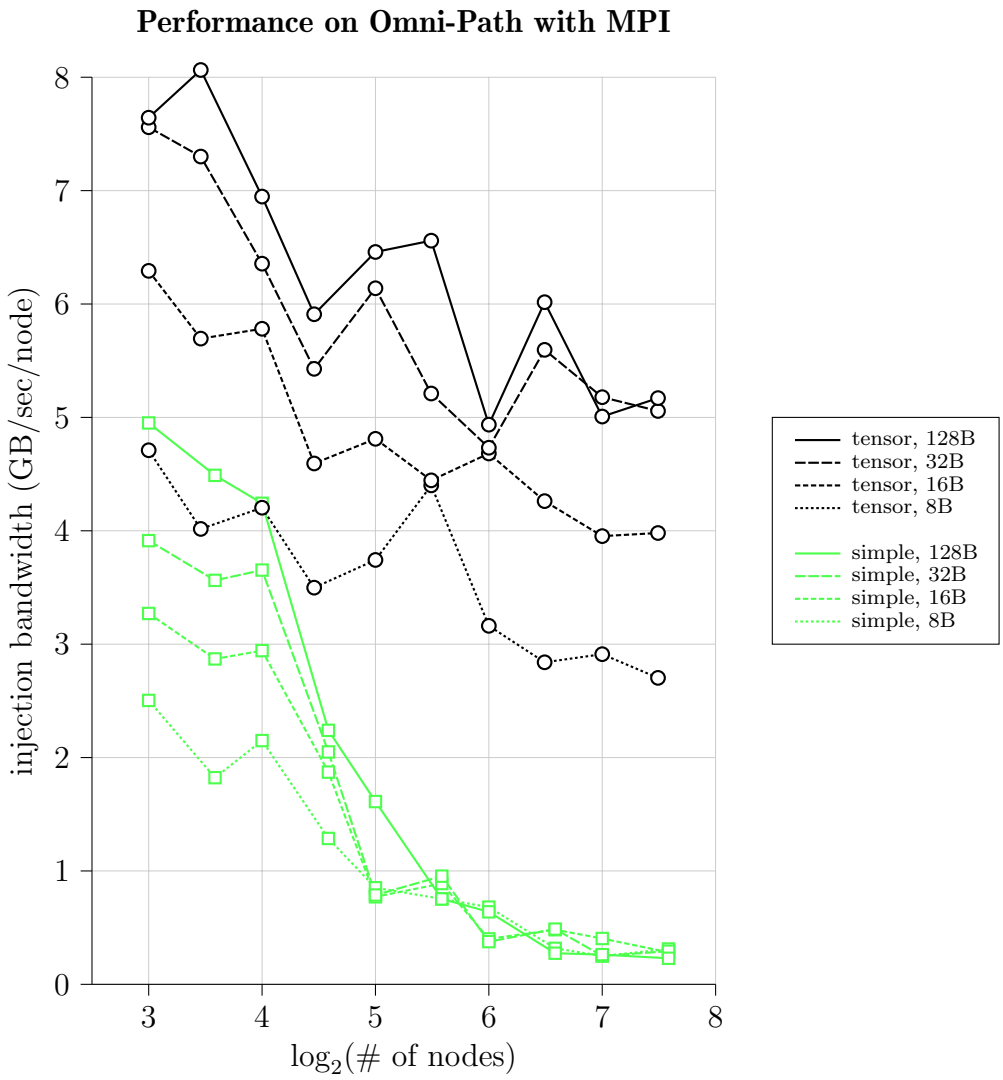

Figure 7: Comparing tensor conveyors (black) and simple conveyors (green) on a cluster with an Omni-Path fabric. Each curve corresponds to an item size. Simple conveyors use `MPI_Alltoallv` for bulk communication, while tensor conveyors use `MPI_Isend` and `MPI_Irecv`. Communication throughput on this sytem is notably erratic, and the reason for the poor scaling of `MPI_Alltoallv` is unknown.

the cost can be attributed to the small routing tags transmitted by 2-hop and 3-hop conveyors, but most of it is apparently due to CPU activity. The figure captions mention a couple of other striking features. All the results exhibit a nontrivial amount of noise.

Not shown here are the results of our rare benchmark runs at scales beyond 2048 nodes. Using an older version of tensor conveyors on a Cray XC30 system, we achieved a sustained injection bandwidth of 4.15 GB/sec/node on 4096 nodes and 3.50 GB/sec/node on 8192 nodes, compared with 5.01 GB/sec/node on 1024 nodes.

6 Conclusions

Abstractly, a conveyor is any parallel object that routes small messages among the processes in a parallel job in streaming fashion. Conveyors are particularly useful for mediating access to large distributed data structures. We have formalized a suitable notion of conveyor for C-like languages, and have demonstrated that conveyor implementations can provide high throughput and low memory use across a variety of machines and communication substrates. One can imagine building applications that employ numerous simultaneously active conveyors.

The C incarnation of conveyors, however, suffers from a relatively inconvenient interface. The conveyor client must interleave push and pull operations, and in particular must maintain any state associated with an interrupted sequence of pushes or pulls. The subloops of a conveyor loop that perform the pushes and pulls behave like separate threads of execution that temporarily block on failure. In effect, the client must implement a form of cooperative multithreading by hand. Moreover, if one wanted to use conveyors within a parallel data structure to provide efficient streaming access, the data structure API would inherit the awkwardness of the conveyor API.

The conveyor abstraction becomes more natural in languages with good native support for multithreading. The conveyor `push` and `pull` operations can be made to block rather than failing, as the client will perform them on separate threads; and the `advance` operation can be done automatically, by internal conveyor threads if necessary. How best to achieve this level of convenience in C++ or another C-like language is an open question.

6.1 Alternative APIs

For completeness, we finish by comparing conveyors to the other communication modules in the `bale` package. The synchronous `exstack` and asynchronous `exstack2` objects in `bale` can be regarded as primitive conveyors. In functionality and performance, `exstack2` should be dominated by the latest (version 0.5) asynchronous multi-hop conveyors in almost all circumstances. But `exstack` is different: its contract makes more demands on the client but also provides stronger guarantees, which can be helpful.

A single `exstack` object can substitute for a pair of conveyors in the *indexgather* idiom if responses are no larger than queries. After queries are pushed and the `advance` operation is performed, `exstack` guarantees that the outgoing buffers are empty, so the replies can be pushed into the same conveyor; they cannot overflow. A second `advance` operation transfers the replies back to their senders, where they will be arranged in their buffers in the same order as the original queries. Because `exstack` supports pulling the next item from a given source, the participating processes have the option to replay the most recent set of queries and pull the responses in order. Thus one can often implement the *indexgather* idiom without sending any identifying data with each query and response, saving both bandwidth and lines of source code at the cost of buffer memory.

Because the *indexgather* idiom is common and important, it may be worth encapsulating this bidirectional form of communication—involving queries and responses that arrive in the order that the queries were sent—in a new abstraction. A *biconconveyor* could be implemented either by a single simple conveyor or by a pair of multi-hop conveyors together with mechanisms for tracking sequence numbers of queries and reordering responses accordingly.

Acknowledgements

The authors would like to thank the other members of the biweekly "programming models" hangout, including Bill Carlson, John Fregeau, John Gilbert, Roger Golliver, Bob Lucas, Phil Merkey, Dan Pryor, Kevin Sheridan, and Kathy Yelick, for many fruitful discussions.

References

[1] Pavan Balaji et al., MPI on millions of cores, *Parallel Processing Letters* 21:1, pp. 45–60, 2011.

[2] Benjamin Brock, Aydın Buluç, and Katherine Yelick, BCL: A cross-platform distributed container library, arXiv:1810.13029, October 2018.

[3] Jason DeVinney et al., `https://github.com/jdevinney/bale`.

[4] Rahul Garg and Yogish Sabharwal, Software routing and aggregation of messages to optimize the performance of HPCC Randomaccess benchmark, *ACM/IEEE Conference on Supercomputing* (SC2006), November 2006.

[5] Lukasz Wesolowski et al., TRAM: Optimizing fine-grained communication with topological routing and aggregation of messages, *International Conference on Parallel Processing* (ICPP 2014), September 2014.

[6] Nathan Wichmann, private communication, March 2016.