Conveyors for Streaming Many-To-Many Communication

F. Miller Maley[1] and Jason G. DeVinney[2]

[1] IDA Center for Communications Research, 805 Bunn Drive, Princeton NJ 08540
[2] IDA Center for Computing Sciences, 17100 Science Drive, Bowie MD 20715

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 OpenSHMEM, MPI, or UPC, but it performs best when using recent OpenSHMEM features such as nonblocking puts. It defines a simple interface to parallel objects called *conveyors*, and it provides several 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. Conveyors of this type scale well to 2^{17} processes and beyond.

Keywords: all-to-all · nonblocking · SHMEM · routing · signaling put

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 buffer the items in some way. Fortunately, many applications can tolerate the resulting increase in latency (see [3], for instance). The problem we consider is how to present an efficient and convenient buffering capability to the application programmer.

The *conveyor* package described here was inspired by the modules `exstack` and `exstack2` written by the second author and his colleagues at IDA/CCS. All this software is available in a collection called `bale` [2]. In each case 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 of these operations vary.) An application creates and destroys its communication objects as needed. Typically, each object can only transmit items of a certain fixed size.

The conveyor package improves upon its predecessors in three ways. First, it carefully specifies the interface between the application and the conveyor. If the application uses this interface properly, then the conveyor's communication mechanism can change drastically, even from bulk synchronous to asynchronous, without affecting correctness. Second, it introduces a family of conveyors that, in many cases, achieve unsurpassed performance while also greatly reducing

memory usage. These conveyors use only point-to-point synchronization, and they exploit the ability to communicate quickly between processes on the same node. Third, the conveyor package extends the basic interface to permit the transmission of items of different sizes, including very large items. This capability simplifies certain kinds of application code.

Like `exstack` objects, 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. One can write cleaner, higher-level communication mechanisms on top of conveyors, but this is a topic for future research.

1.1 Motivation

The communication pattern we support 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, one can 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 conveyor performance.

The many-to-many communication pattern arises in two idioms commonly found in 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 example, 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 X10 or Chapel, one can write very concise code for *histogram* using some form of remote atomic increment (see Figure 1). But the resulting performance will be somewhere between mediocre and dismal, depending on the compiler, communication library, and hardware platform.

In the second idiom, every process wants to make numerous queries to a distributed data structure and operate 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.

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

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

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

Experience has shown that it is difficult, at least in a primitive language like C, to design a useful library to accelerate the *histogram* and *indexgather* idioms. Indeed, it is rare for two different instances of an idiom to be exactly the same, so the library functions would need to call back into application code to perform the desired updates, queries, and response processing. Interacting with such a library is awkward and potentially inefficient. And, of course, some communication patterns are more complex than these two idioms. Our efforts to date have therefore focused on providing a lower-level interface to a buffering system, on top of which the application programmer can write efficient, if prolix, code for any particular communication-heavy algorithm.

1.2 Scaling Issues

The simplest approach to buffering, which we presume has been implemented many times, is based on bulk synchronous communication. For example, 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 effect, it performs an out-of-place parallel transpose of a matrix of buffers. By placing these operations in a loop and adding a mechanism to detect global termination, one can perform many-to-many communication in streaming fashion with a fixed amount of buffer space.

It turns out that `exstack`, and to a lesser extent `exstack2` (a variant that uses asynchronous communication), 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` 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 optimal message size is 2^{13} bytes (sometimes 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. One can shrink the buffers, but that 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 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 t of threads per process, and it divides the total amount of buffer memory by at least t if not t^2. (In practice, threads may need private outgoing buffers to avoid contention.) But programmers prefer the convenience of a one-level model of parallelism, and rewriting large existing applications to permit multithreading is a daunting prospect.

Another 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. For example, the Topological Routing and Aggregation Module (TRAM [5]) in Charm++ 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.

The conveyor package also performs multi-hop routing, but it makes even greater use of intranode communication. If there are n processes per node, then it can save a factor proportional to n^2 in buffer space, while sacrificing little if any bandwidth. Furthermore, as the load becomes small, the total number of messages sent also drops by a factor proportional to n^2. Thus both scaling problems are greatly ameliorated. The complexity of the multi-hop routing algorithm is encapsulated in the conveyor package and is almost invisible to the application programmer.

2 The Conveyor Interface

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 a client cannot peek at their internal representations and thus subvert the contract between the conveyor and the client. 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 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. 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 a reason revealed in Section 2.3. The `push` and `pull` operations can succeed or fail, and their success or failure can be observed from their return values.

2.1 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, which we call DORMANT, WORKING, ENDGAME, CLEANUP, and COMPLETE. 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.

A newly created conveyor is DORMANT on every process, and the `begin` operation causes a transition to the WORKING state. The client can track the state thereafter by observing the return values from `advance`. 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; its local state becomes ENDGAME, CLEANUP, or COMPLETE. Once all processes have left the WORKING state, the conveyor only has to worry about delivering previously pushed items. When those items have all been received by their destination processes, each local state becomes CLEANUP if some items remain to be pulled, and otherwise COMPLETE. Once the state becomes COMPLETE, it is legal to `free` or `reset` the conveyor. The `reset` operation returns the state to DORMANT.

2.2 The Contract

For brevity, we omit a detailed specification of the contract between a conveyor and its client. 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, eventually changing the argument of `advance` from `false` to `true`, until `advance` returns 0. 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 any process to any another will arrive in order (first-in, first-out), and that `unpull` can be used to reverse one `pull` but maybe no more.

An unusual clause in the contract says that for each conveyor, `advance` is either collective and synchronizing (until the processes collectively transition to the CLEANUP and COMPLETE states) or always nonblocking. The distinction is unimportant when only a single conveyor is active, but when a client uses two conveyors simultaneously, it must take care to advance each one in a collective way, lest they deadlock. This clause allows conveyors to be implemented using either synchronous or asynchronous communication mechanisms.

```
convey_begin(c);
long i = 0, spot;
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);
```

Fig. 2. 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.

2.3 Examples of Use

Figures 2 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, in UPC style, and we denote the local slice of such an array by a corresponding lowercase name (`tally` or `array`). 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 2 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 benign 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 0. 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`. If this feature were not available, then the application code would need to keep track of an unsatisfied query and would become even more complicated.

The trickiest part of the *indexgather* code in Figure 4 is the condition in the outer `while` loop, which uses some relatively uncommon C operators. This formula 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.

2.4 Optional Features

Thinking about how to implement various parallel algorithms using conveyors soon suggests two features that would make programming with conveyors much easier. 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. So conveyors now support these two features as options. Additional features may be offered in the future. 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.

A conveyor that can transport variable-length items is called *elastic*. To support elastic conveyors, the conveyor interface includes additional functions called `convey_epush` and `convey_epull`. In terms of the contract, these are just `push` and `pull` operations, but they have more elaborate prototypes than `convey_push` and `convey_pull`. The elastic push takes an extra argument, namely the item size; the elastic pull, when successful, delivers this size to its caller.

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, without changing the argument of `advance`, every pushed item will eventually be delivered. In particular, pushes can be contingent on previous pulls without causing the conveyor to stall.

3 Asynchronous Multi-Hop Conveyors

The conveyor package provides four different types of conveyors, and it can be built atop any of five different communication substrates. We focus here on the two asynchronous conveyor types and the OpenSHMEM substrate.

Conversations with a Cray engineer [6], along with benchmark data he provided, suggested that the most efficient way to use a modern Cray interconnect was to send messages using only point-to-point synchronization. It was 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 simplified benchmarks but should be helpful in real applications.

The most important contribution of the conveyor package is the construction of conveyors that combine asynchronous communication with multi-hop routing. 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.

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 created.
- 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 (within-node) 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' (roughly p/n^2 links). A message from (x, y, z) to (x', y', z') travels through the intermediate processes (x, y, y') and (x', y', y). As the message proceeds, 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. With tags written in brackets, the routing looks like:

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

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.

3.2 Low-Level Mechanisms

It remains to explain how porters work. The SHMEM substrate affords a wide 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, 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 check 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.

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 function like `shmem_putmem_nbi` 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, OpenSHMEM provides 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 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, which 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.

Summary A tensor conveyor built atop OpenSHMEM typically uses the standard porter based on puts and fences. But if a nonblocking put is available, then nonlocal porters use this mechanism instead. Similarly, 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`.

The primitive operations used by porters are 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.

A wrinkle not previously mentioned is that 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.

3.3 Elastic Conveyors

Until recently, 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. We created a second type of asynchronous multi-hop conveyor to support this feature.

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, padded to a multiple of four bytes. In the elastic case, a packet contains a routing tag, a 32-bit *descriptor* that mainly specifies the item size, and then an item of the appropriate size, again padded to a multiple of four bytes. Naturally the pivot functions must understand this new format.

Porters also need to understand variable-length items. So 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 conveyor will only call `porter_push` or `porter_epush`, not both.

3.4 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. When a `pull` operation at the destination encounters the ticket, it uses `shmem_getmem` 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.

Conveying tickets is essential for maintaining FIFO 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 Benchmark Results

Figure 3 illustrates the performance of the OpenSHMEM tensor conveyors on two different parallel computers: a Cray XC40 with a Cray Aries network, and a cluster with an FDR InfiniBand network. A node in each of these computers has two Intel Xeon E5-2698v3 "Haswell-EP" CPUs, each with 16 cores running at a nominal frequency of 2.3 GHz, and 128 GiB of main memory. The machines were moderately or highly busy with other jobs when the experiments were run. The results, therefore, are probably worse than peak performance on an empty machine but better than what one would see if the machine were filled with communication-bound jobs.

The experiments used one PE per core, and hence 32 processes per node. Each experiment fixed the buffer size (at about 5KB for the Cray and 92KB for the cluster), and it varied the number of nodes, the item size, and two internal parameters of the conveyor, namely the number of hops (2 or 3) and the size of

each local group (16 or 32). The plotted result is the maximum, over those choices of internal parameters, of the average injection bandwidth per node measured in gigabytes per second. Bandwidth is the number of bytes transmitted divided by the elapsed time, and this time includes the overhead of pushes, pulls, and computing checksums of outgoing and incoming data.

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. The data points 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. Performance increases with the item size, so the item size can be inferred from the position of a curve relative to the other curves.

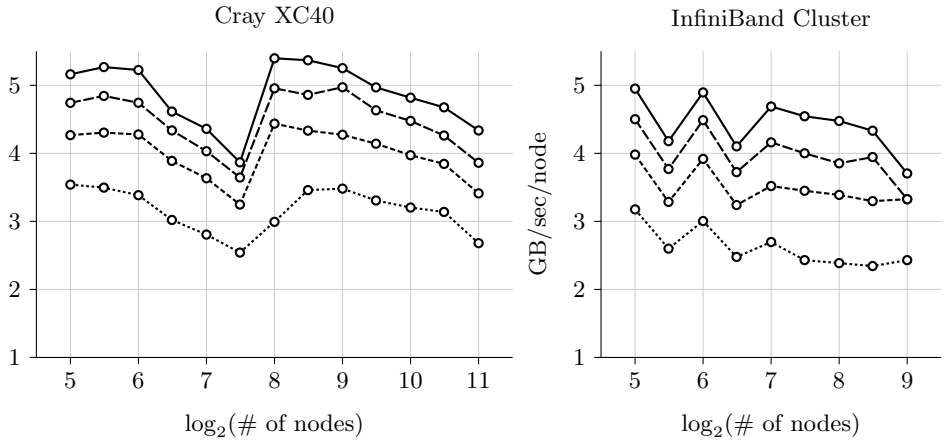

Fig. 3. Selected benchmark results for tensor conveyors.

Much could be said about the benchmark results, but for reasons of space we limit ourselves to three remarks.

- Prior to this work, we believed that 5 GB/sec/node was the practical limit for injection bandwidth on the Cray XC30 and XC40, beyond a very small number of nodes.
- We have occasionally been able to test various versions of the tensor conveyors at larger scales, up to 2^{17} cores and beyond. Performance continues to decline, but not severely.
- We also have performance data for "simple" bulk synchronous conveyors akin to `exstack`. The tensor conveyors very nearly dominate the simple conveyors at every item size despite transmitting more data (the routing tags). Often they are 20–40% faster.

5 Conclusions

The conveyor project has achieved all its original goals: to solve the scaling problems caused by using $2N^2$ buffers in aggregate for N processes; to improve many-to-many communication performance by exploring asynchronous mechanisms; and to provide a unified interface to different buffering schemes. Going beyond its initial goals, the project has also introduced elastic and steady conveyors, which expand the range of possible applications.

When the number of processes is large and there are many processes per node, the three-hop asynchronous "tensor" conveyors generally provide the best performance of any application-level communication mechanism we have seen. 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 to work optimally, they need to be told the number n of processes per local group. Sometimes n should be the number of processes per node, which the conveyor can determine for itself, but sometimes it should be the number of processes per NUMA domain, such as a socket.

Further performance improvements are within reach. First, we could fairly easily reduce the number of bytes occupied by routing tags. Second, we have begun to design mechanisms for on-the-fly compression of buffers. This idea has previously been implemented in HPX [4], for instance, using standard compression libraries like `zlib` and `snappy`. Even the fastest of these, however, adds significant overhead to communication on a fast interconnect. An alternative is to exploit the fact that the items in a buffer, especially when carrying integer data, often include many constant bits or bytes such as high bits that are always zero. The first author has written code which demonstrates that constant bits can be squeezed out cheaply, relative to communication costs, using the "parallel extract" and "parallel deposit" instructions available on recent Intel CPUs. A future enhancement of conveyors could exploit built-in and/or client-provided compression functions.

Finally, we mention an OpenSHMEM API issue raised by our method of polling for incoming signals. To guarantee correctness in the presence of weak memory models, we should really do this polling using some form of `shmem_test`. But making many calls to `shmem_test` in succession is inefficient, as each may involve a memory fence. So to date we have written our own architecture-dependent polling code. To resolve this issue, we would like to have a function such as `shmem_test_some` that would check many locations in one call. Discussions on this subject are underway.

Acknowledgements

The authors are indebted to 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. Balaji, P. et al.: MPI on millions of cores. Parallel Processing Letters **21**(1), 45–60 (2011)
2. DeVinney, J. et al.: https://github.com/jdevinney/bale.
3. Georganas, E. et al.: HipMer: An extreme-scale de novo genome assembler. In: 27th ACM/IEEE International Conference on High Performance Computing, Networking, Storage, and Analysis (SC 2015), Austin TX (2015).
4. Heller T., Diehl, P., Byerly, Z., Biddiscombe, J., and Kaiser, H.: HPX – An open-source C++ standard library for parallelism and concurrency. In: Proceedings of OpenSuCo 2017, Denver, Colorado (2017)
5. Wesolowski, L. et al.: TRAM: Optimizing fine-grained communication with topological routing and aggregation of messages, In: International Conference on Parallel Processing (ICPP 2014), Minneapolis, Minnesota (2014)
6. Wichmann, N.: Private communication (2016)

```
struct packet { long slot; long value; } packet;
long i = 0, from;
bool more;
while (more = convey_advance(q, i == n),
       more | convey_advance(r, !more)) {
  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;
}
```

Fig. 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.