Abstract—The Partitioned Global Address Space (PGAS) model is well suited for executing irregular applications on cluster-based systems, due to its efficient support for short, one-sided messages. However, there are currently two major limitations faced by PGAS applications. The first relates to scalability — despite the availability of APIs that support non-blocking operations in special cases, many PGAS operations on remote locations are synchronous by default, which can lead to long-latency stalls and poor scalability. The second relates to productivity — while it is simpler for the developer to express all communications at a fine-grained granularity that is natural to the application, experiments have shown that such a natural expression results in performance that is 20× slower than more efficient but less productive code that requires manual message aggregation and termination detection.

Separately, the actor model has been gaining popularity as a productive asynchronous message-passing approach for distributed objects in enterprise and cloud computing platforms, typically implemented in languages such as Erlang, Scala or Rust. To the best of our knowledge, there has been no past work on using the actor model to deliver both productivity and scalability to PGAS applications on clusters.

In this paper, we introduce a new programming system for PGAS applications, in which point-to-point remote operations can be expressed as fine-grained asynchronous actor messages. In this approach, the programmer does not need to worry about programming complexities related to message aggregation and termination detection. Our approach can also be viewed as extending the classical Bulk Synchronous Parallelism model with fine-grained asynchronous communications within a phase or superstep. We believe that our approach offers a desirable point in the productivity-performance space for PGAS applications, with more scalable performance and higher productivity relative to past approaches. Specifically, for seven irregular mini-applications from the Bale benchmark suite executed using 2048 cores in the NERSC Cori system, our approach shows geometric mean performance improvements of ≥ 20× relative to standard PGAS versions (UPC and OpenSHMEM) while maintaining comparable productivity to those versions.

Index Terms—Actors, Communication Aggregation, Conveyors, OpenSHMEM, PGAS, Selectors

I. INTRODUCTION

In today’s world, performance is improved mainly by increasing parallelism, thereby motivating the critical need for programming systems\(^1\) that can deliver both productivity and scalability for parallel applications. The Actor Model [1], [2] is the primary concurrency mechanism [3] in languages such as Erlang and Scala, and is also gaining popularity in modern system programming languages such as Rust. Large-scale cloud applications [4] from companies such as Facebook and Twitter that serve millions of users are based on the actor model. Actors express communication using “mailboxes” [5]; the term, “selector” [6], has been used to denote an actor with multiple mailboxes. The actor runtime maintains a separate logical mailbox for each actor. Any actor or non-actor, can send messages to an actor’s mailbox. An important property of communication in Actors/Selectors is their inherent asynchrony, i.e., there are no global constraints on the order in which messages are processed in mailboxes.

While many classical HPC applications focused on dense data structures, there has been a recent increase in the use of sparse data structures for HPC, including those used in graph algorithms, sparse linear algebra algorithms, and machine learning algorithms [7]. The Partitioned Global Address Space (PGAS) model [8] is well suited to such irregular applications due to its efficient support for short, non-blocking one-sided messages and the convenience of a non-uniform global address space abstraction which enables the programmer to implement scalable locality-aware algorithms. Notable PGAS programming systems include Co-array Fortran [9], OpenSHMEM [10], and Unified Parallel C (UPC) [11]. Distribution of irregular data structures across multiple PEs gives rise to large numbers of fine-grain communications, which can be expressed succinctly in the PGAS model.

However, a key challenge for PGAS applications is the need for careful aggregation and coordination of short messages to achieve low overhead, high network utilization, and correct termination logic. Communication aggregation libraries such as Conveyors [12] can help address this problem by locally buffering fine-grain communication calls and aggregating them into medium/coarse-grain messages. However, the use of such aggregation libraries places a significant burden on programmer productivity and assumes a high expertise level.

In this paper, we introduce a new programming system for PGAS applications, in which point-to-point remote operations can be expressed as fine-grained asynchronous actor messages. In this approach, the programmer does not need to worry about programming complexities related to message aggregation and
termination detection. Further, the actor model also supports the desirable goal of migrating computation to where the data is located, which is beneficial for many irregular applications \cite{13}.

Our approach can also be viewed as extending the classical Bulk Synchronous Parallelism (BSP) model with fine-grained asynchronous communications within a phase or superstep. Many current HPC execution models have been influenced by the simplicity and scalability of the BSP model, which consists of “supersteps” separated by barriers executing on homogeneous processors. However, the increasing degree of heterogeneity and performance variability in exascale machines has motivated the need for including asynchronous computations within a superstep so as to reduce the number of barriers performed and the total time spent waiting at barriers.

Specifically, this paper makes the following contributions:
1) An extension of the BSP model to a Fine-grained-Asynchronous Bulk-Synchronous Parallelism (FA-BSP) model.
2) A new PGAS programming system which extends the actor/selector model to enable asynchronous communication with automatic message aggregation for scalable performance.
3) Design of an automatic communication termination protocol that transfers the burden of termination detection and related communication bookkeeping from the programmer to the selector runtime.
4) An implementation of our approach as extensions to the Habanero-C/C++ task-parallel library (HClib) and the Conveyors communication aggregation library.
5) Development of a source-to-source translator that translates our lambda-based API for actors to a more efficient class-based API.
6) Our results show a geometric mean performance improvement of 25.59× relative to the UPC versions and 19.83× relative to the OpenSHMEM versions, while using 2048 cores in the NERSC Cori system on seven irregular mini-applications from the Bale suite \cite{14}, \cite{12}

II. BACKGROUND: COMMUNICATION IN PGAS APPLICATIONS

In this section, we summarize two fundamental messaging patterns in PGAS applications, namely read and update, as well as the Conveyors library that can be used to aggregate messages. Since the focus of our work is on scalable parallelism, we assume a Single Program Multiple Data (SPMD) model in which each processing element (PE) starts by executing the same code with a distinct rank, as illustrated in the following code examples.

A. Read Pattern

In this pattern, each PE sends a request for data from a dynamically identified remote location and then processes the data received in response to the request. An OpenSHMEM version of a program using this pattern is shown in \textbf{Listing 1}.

```
1 for(i = 0; i < n; i++) {
2   int col = index[i] / shmem_n_pes();
3   int pe = index[i] % shmem_n_pes();
4   gather[i] = shmem_g(data+col, pe);
}
```

This program reads values from a distributed array named \texttt{data} and stores the retrieved values in a local array named \texttt{index}. The corresponding operation can also be performed in MPI using \texttt{MPI_Get}.

\textbf{Listing 1: An OpenSHMEM program that reads data from a distributed array.}

```
1 for(i = 0; i < n; i++) {
2   int col = index[i] / shmem_n_pes();
3   int pe = index[i] % shmem_n_pes();
4   MPI_Get_Accumulate(gather+i, data+col, PE, 
5   col = index[i] / shmem_n_pes();
6   pe = index[i] % shmem_n_pes();
7   shmem_atomic_inc(histo+spot, PE);
8 }
```

B. Update Pattern

In this pattern, each PE updates a remote location at an address that is computed dynamically. An OpenSHMEM program that updates remote locations is shown in \textbf{Listing 2}.

```
1 for(i = 0; i < n; i++) {
2   int spot = index[i] / shmem_n_pes();
3   int PE = index[i] % shmem_n_pes();
4   shmem_atomic_inc(histo+spot, PE);
5 }
```

This program updates a distributed array named \texttt{histo} based on global indices stored in each PE’s local \texttt{index} array using atomic increment, thereby creating a histogram. The corresponding operation can also be performed in MPI using \texttt{MPI_Accumulate} or \texttt{MPI_Get_Accumulate} or \texttt{MPI_Put_and_op}.

\textbf{Listing 2: An OpenSHMEM program that creates a histogram by updating a distributed array.}

```
1 for(i = 0; i < n; i++) {
2   int col = index[i] / shmem_n_pes();
3   int pe = index[i] % shmem_n_pes();
4   convey_push(data+col, pe);
5 }
```

C. Conveyors

Conveyors \cite{12} is a C-based message aggregation library built on top of conventional communication libraries such as SHMEM, MPI, and UPC. It provides the following three basic operations:
1) convey_push: attempts to locally enqueque a message for delivery to a specified PE.
2) convey_pull: attempts to fetch a received message from the local buffer.
3) convey_advance: enables forward progress of communication by transferring buffers.

It is worth noting that both push and pull operations can fail (return false) due to resource constraints. push can fail due to a lack of available buffer space, and pull can fail due to a lack of an available item. Due to these failures, push and pull operations must always be placed in a loop that ensures that the operations are retried. Further, advance needs to be called to ensure progress and to also help with termination detection. These complexities place a significant burden on programmer productivity and assumes a high expertise level. \textbf{Table 1} demonstrates that user-directed message aggregation with Conveyors can achieve much higher performance compared to non-blocking operations in state of the art communication libraries/systems, some of which includes automatic message aggregation \cite{15}, \cite{16}. As a result, we decided to use Conveyors as a lower-level library for automatic message aggregation in our programming system.
TABLE I: Absolute performance in seconds using best performing variants for Read and Update benchmarks on 2048 PEs (64 nodes with 32 PEs per node) in the Cori supercomputer which performs $2^{23}$ (~8 million) reads and updates. Each version is annotated with a non-blocking (NB) specifier.

| Benchmark | OpenSHMEM (cray-shmem 7.7.10) | NB | Time |
|-----------|--------------------------------|----|------|
| Read      | OpenSHMEM NBI (cray-shmem 7.7.10) | Y  | 4.2  |
|           | UPC (Berkley-UPC 2020.4.0)       | N  | 22.6 |
|           | UPC NBI (Berkley-UPC 2020.4.0)   | Y  | 19.7 |
|           | MPI3-RMA (OpenMPI 4.0.2)         | Y  | 25.8 |
|           | MPI3-RMA (cray-mtich 7.7.10)     | Y  | 8.3  |
|           | Charm++ (6.10.1, gni-crayxc w/ TRAM) | Y  | 21.3 |
|           | Conveyors (2.1 on cray-shmem 7.7.10) | Y  | 2.3  |

| Benchmark | OpenSHMEM NBI (cray-shmem 7.7.10) | NB | Time |
|-----------|--------------------------------|----|------|
| Update    | OpenSHMEM NBI (cray-shmem 7.7.10) | Y  | 4.3  |
|           | UPC (Berkley-UPC 2020.4.0)       | N  | 23.9 |
|           | MPI3-RMA (OpenMPI 4.0.2)         | Y  | 88.9 |
|           | MPI3-RMA (cray-mtich 7.7.10)     | Y  | >300 |
|           | Charm++ (6.10.1, gni-crayxc w/ TRAM) | Y  | 9.7  |
|           | Conveyors (2.1 on cray-shmem 7.7.10) | Y  | 0.5  |

III. OUR APPROACH

A. Fine-grained-Asynchronous Bulk-Synchronous Parallelism (FA-BSP) model

The classical Bulk-Synchronous Parallelism (BSP) [17] model consists of "supersteps" separated by barriers executing on homogeneous processors. Each processor only performs local computations and asynchronous communications in a superstep, and the role of the barrier is to ensure that all communications in a superstep have been completed before moving to the next superstep. However, the increasing degree of heterogeneity and performance variability in modern cluster machines has motivated the need for including asynchronous computations within a superstep so as to reduce the number of barriers performed and the total time spent waiting at barriers. To that end, we propose extending BSP to a Fine-grained-Asynchronous Bulk-Synchronous Parallelism (FA-BSP) model, as follows.

Our proposal is to realize the FA-BSP model by building on three ideas from past work in an integrated approach. The first idea is the actor model, which enables distributed asynchronous computations via fine-grained active messages while ensuring that all messages are processed atomically within a single-mailbox actor. For FA-BSP, we extend classical actors with multiple symmetric mailboxes for scalability, and with automatic termination detection of messages initiated in a superstep. The second idea is message aggregation, which we believe should be performed automatically to ensure that the FA-BSP model can be supported with performance portability across different systems with different preferences for message sizes at the hardware level due to the overheads involved. The third idea is to build on an asynchronous tasking runtime within each node, and to extend it with message aggregation and message handling capabilities.

Our primary approach to delivering both productivity and scalability for PGAS applications is by building a programming system based on the actor model that also supports automatic message aggregation and termination detection. Relative to the Conveyors approach, we would like to remove the burden of the user having to worry about both 1) the lack of available buffer space (convey_push), 2) the lack of an available item (convey_pull), and 3) the progress and termination of communications (convey_advance). We believe that the use of the actor/selector model is well suited for this problem since its programming model productively enables the specification of fine-grained asynchronous messages. Some key elements of the high-level design are summarized below

1) Abstraction of buffers as mailboxes: We observe that buffer operations can be elevated to actor/selector mailbox operations with much higher productivity. For example, the convey_push operation on a buffer can be elevated to an actor/selector send operation, and a convey_pull operation can be made implicit in an actor/selector’s message processing routine, while leaving it to our programming system to handle buffer/item failures and progressing/terminating communications among actors/selectors. More details on how our runtime handles failure scenarios are given in Section IV-C.

An important design decision for scalability is to treat a collection of mailboxes as a distributed object so that the mailboxes can be partitioned across PEs, analogous to how memory is partitioned in the PGAS programming model. This partitioned global actor design allows users to access a target actor’s mailbox conveniently, instead of having to search for the corresponding actor object across multiple nodes as is done in many actor runtime systems.

2) Supporting Multiple Mailboxes: Among the two patterns discussed in Section II, the read pattern differs from the update pattern in that it involves communication in two directions, namely request and response. Since it is challenging for actors with a single mailbox to implement such synchronization and coordination patterns, this motivates us to instead use a ‘Selector’ [6], which is an actor with multiple mailboxes, as a high-level abstraction. For example, for the read operation, users are expected to create two mailboxes

Fig. 1: Bulk-Synchronous Parallelism (BSP) vs Fine-grained-Asynchronous Bulk-Synchronous Parallelism (FA-BSP). (Graphics adapted from [13].)
(one for Request, the other for Response) and implement the Selector's message processing functions for the two mailboxes. This partitioned global selector design enables a uniform programming interface across the different communication patterns.

3) **Progress and Termination:** In general, the Actors/Selectors model provides an `exit` operation to terminate actors/selectors. While it may seem somewhat natural to expose this operation to users, one problem with this termination semantics is that it requires users to ensure that all messages in the incoming mailbox are processed (or received in some cases) before invoking `exit`, which adds additional complexities even for the simplest mini-applications such as Histogram [Listing 2]. To mitigate this burden, we added a relaxed version of `exit`, which we call `done`, to enable the runtime do more of the heavy lifting. The semantics of `done` is that users tell the runtime that the PE on which a specific actor/selector object resides will not send any more messages in the future to a particular mailbox, so the runtime can still keep the corresponding actor/selector alive so it can continue to receive messages and process them. More details on progress and termination can be found [Section III-D](#).

C. **User-facing API**

Based on the discussions in [Section III-B](#) we provide a C/C++ based actor/selector programming framework as shown in **Listing 3**.

Listing 3: Actor/Selector Interface with partitioned global mailboxes.

```c
1 //L: lambda type
2
3 class Actor<L> {
4    void send(int PE, L msg);
5    void done();
6};
7
8 class Selector<N, L> { //N mailboxes
9    void send(int mailbox_id, int PE, L msg);
10   void done(int mailbox_id);
11};
```

**Update:** [Listing 4](#) shows our version of the histogram benchmark. We use C++ lambdas to succinctly describe both the message and its processing routine. The main program creates an Actor object as a collective operation in Line: 1, which is used for communication. Then to create the histogram, it finds the target PE in Line:4 and local index within the target in Line: 5 from the global index. Then it sends a message lambda to the target PE’s mailbox using the send API. Once the target PE’s mailbox gets the message, the actor invokes it, which updates the histo array. Note that the lambda automatically captures the value of `spot` inside it. Also, the code for the lambda does not need to be communicated since it is compiled ahead of time and available on nodes.

Listing 4: Actor version of the Update benchmark (Histogram) using lambda.

```c
1 Actor h_actor;
2 for(int i=0; i < n; i++) {
3    int spot = index[i] / shmem_n_pes();
4    int possibly_remote_PE = index[i] % shmem_n_pes();
5    h_actor.send(possibly_remote_PE, [=](){histo[spot]+=1;});
6 }
7 h_actor.done();
```

**Read:** [Listing 5](#) shows how the read pattern can be implemented using our selector-based approach with multiple mailboxes. (Recall that an actor is simply a selector with one mailbox.) Here the selector sends the message to the Request mailbox on the target PE on Line: 8. When the message gets processed in target PE, it fetches the required values in Line: 9 and replies with a message to the Response mailbox of the sender PE in Line: 10. In turn, when the reply message gets processed at the sender PE, it writes the values to the gather array at the appropriate index.

Subparts of the application that involve a large number of latency tolerant communication operations can use our API to achieve high throughput, while other parts of the application can continue using other PGAS interfaces as convenient.

Listing 5: Selector version of the Read benchmark using the lambda API.

```c
1 enum MB {Request, Response};
2 Selector<2> r_selector;
3
4 int sender_PE = shmem_my_pe();
5 for(int i=0; i < n; i++)
6    int col = index[i] / shmem_n_pes();
7    int possibly_remote_PE = index[i] % shmem_n_pes();
8    r_selector.send(Request, possibly_remote_PE, [=](){
9        int ret_val = data[col];
10       r_selector.send(Response, sender_PE,
11          [=](){ gather[i] = ret_val; });
12    });
13 }
14 r_selector.done(Request);
```

D. **Termination Graphs**

As mentioned earlier, we provide the `done` operation as an alternative to terminating actors/selectors by `exit` [19]. This design is based on our observation that 1) sending messages from a partition can be considered as the active part of communication where the user has to invoke `send` explicitly, but in contrast, 2) receiving messages is the passive part since the arrival of a message is not directly under the user’s control. Therefore, we designed the `done` termination interface to be more associated with sending of messages to a mailbox and leave it to the runtime to keep track and drain all messages sent to it in the future and also in flight.

One may have noticed that, in [Listing 5](#) the `done` operation is performed only for the Request mailbox and not for Response mailbox. This is possible since the Response mailbox depends on the Request mailbox - i.e., a message is only sent from the Request mailbox to the Response mailbox in Line: 8.

Here we introduce the concept of **Termination Graph** to discuss how this is possible. Let us first define that mailbox `Y` depends on mailbox `X` if a message is sent to mailbox `X` in the process function of mailbox `Y`. Based on the dependency relation, in general, we can create a directed graph between mailboxes within a selector. We assume the graph is acyclic, which is sufficient for supporting common irregular applications, and We assume an imaginary `Outside` mailbox, which is a virtual mailbox that does not depend on any mailboxes within the selector. A dependency on `Outside` mailbox implies a message is received from a non-actor/selector or a different actor/selector from the current distributed one.
In this section, we discuss the implementation of the selector runtime prototype created by extending HClib [20], a C/C++ Asynchronous Many-Task (AMT) Runtime library. We first discuss our execution model in Section IV-B and then describe our extensions to the HClib runtime to support our selector runtime in Section IV-C.

E. Class-based API

While lambdas help with productivity by automatically capturing variables from the environment and enabling the developer to write routines with in-line message-handling logic instead of separate functions, lambda-based operations can incur additional overhead relative to direct method calls. To avoid this overhead, we also created a class-based version of our APIs that gives the user more control regarding what data needs to be communicated and also allows for automatic translation from the lambda API to the class-based API.

Listing 6: Actor/Selector class-based interface with partitioned global mailboxes.

```java
class Actor<T> {
    void process(T msg, int PE);
    void send(T msg, int PE);
    void done();

    Actor() {
        mailbox[0].process = this->process;
        // ...!
    }
}

class Selector<N,T> { // N mailboxes
    void process_0(T msg, int PE);
    ...
    void process_N_1(T msg, int PE);
    void send(int mailbox_id, T msg, int PE);
    void done(int mailbox_id);

    Selector() {
        mailbox[0].process = this->process_0;
        // ...!
        mailbox[N-1].process = this->process_N_1;
    }
}
```

The class-based Actor/Selector interfaces are shown in Listing 6. As with the lambda version, the Actor.send API is implemented using `convey_push`, and the Actor.process API is implemented using `convey_pull` to process the received message. In the Selector version, we can see there are N message processing routines (process_0 to process_N_1), one per mailbox. One important difference that we can notice is that instead of a lambda, the user directly passes the message (msg parameter) that needs to be communicated, and the processing of the message is separately specified using the process routines. A concrete example of how to use the class-based APIs is demonstrated using the Read benchmark in Listing 7. To put in perspective, if we extend Table I with the class-based version, the time taken for Read benchmark is 2.5 sec, and that of Update benchmark is 0.5 sec. Thus we can see that the class-based version can give performance comparable to that of Conveyors and outperform non-blocking communication in the commonly used state-of-the-art communication libraries/systems.

One noticeable difference between the lambda version (Listing 5) and the class-based version (Listing 7) is code verbosity and complexity. This is due to class definition boilerplate code and the specification of process methods. To mitigate this issue of verbosity, which inversely affects productivity, we created a source-to-source translator that transforms from a lambda version to a class-based version. The translator also performs optimizations to match the performance of Conveyors. More details can be found in Section IV-D.

IV. IMPLEMENTATION

In this section, we discuss the implementation of the selector runtime prototype created by extending HClib [20], a C/C++ Asynchronous Many-Task (AMT) Runtime library. We first discuss our execution model in Section IV-B and then describe our extensions to the HClib runtime to support our selector runtime in Section IV-C.
Asynchronous Many-Task (AMT) runtime library. HClib exposes several programming constructs to the user, which in turn helps them to express parallelism easily and efficiently.

A brief summary of the relevant APIs is as follows:
1) async: Used to create asynchronous tasks dynamically.
2) finish: Used for bulk task synchronization. It waits on all tasks spawned (including nested tasks) within the scope of the finish.
3) promise and future: Used for point-to-point inter-task synchronization in C++11 [21]. A promise is a single-assignment thread-safe container, that is used to write some value and a future is a read-only handle for its value. Waiting on a future causes a task to suspend until the corresponding promise is satisfied by putting some value to the promise.

B. Execution Model

Figure 3 shows the high level structure of the execution model for our approach from the perspective of PE j, shown as process[j], with memory[j] representing that PE’s locally accessible memory. This local memory includes partitions of global distributed data, in accordance with the PGAS model. Users can create as many tasks as required by the application, which are shown as Computation Tasks. For the communication part, each mailbox corresponds to a Communication Task. All tasks get scheduled for execution on to underlying worker threads. For example, if an application uses a selector with two mailboxes and an actor/selector with one mailbox, it corresponds to three communication tasks — two for the selector and one for the actor. All computation and communication tasks are created using the HClib [20] Asynchronous Many-Task (AMT) runtime library.

To enable asynchronous communication, the computation tasks offload all remote accesses on to the communication tasks [22]. When the computation task sends a message, it is first pushed to the communication task associated with the mailbox using a local buffer. Eventually, the communication task uses the conveyors library to perform message aggregation and actual communication. Currently we use a single worker thread that multiplexes all the tasks. When a mailbox receives a message, the mailbox’s process routine is invoked.

It is worth noting that users are also allowed to directly invoke other communication calls outside the purview of our Selector runtime. For example, the user application can directly invoke the OpenSHMEM barrier or other collectives.

C. Selector Runtime

The implementation details presented are based on the class-based interface introduced in Section III-E since our results were obtained by coverted the lambda API to the class-based API using the translator described in Section IV-D. As mentioned earlier, we hide the low-level details of conveyors operations from the programmer and incorporate them into our Selector runtime instead. To reiterate, such details include maintaining the progress and the termination of communication as well as handling 1) the lack of available buffer space, and 2) the lack of an available item. This enables users to only stick with the send(), done(), and process() APIs. The implementation details of these APIs are as follows:

Selector.send(): We map each mailbox to a conveyor object. Each send in a mailbox gets eventually mapped to a conveyor.push. Note that the send does not directly invoke the conveyor.push because we want to relieve the computation task on which the application is running from dealing with the failure handling of conveyor.push. Instead, this API adds a packet with the message and receiver PE’s rank to a small local buffer that is based on the

Figure 3: The execution model showing internal structure of tasks and mailboxes within a single PE.
Boost Circular Buffer library [23]. The packet is later picked up by the communication task associated with the mailbox and is passed into a conveyor_push operation. Whenever the mailbox’s local circular buffer gets filled, the runtime automatically passes control to the communication task, which drains the buffer, thereby allowing us to keep its size fixed.

**Selector.done()**: Analogous to send, when done is invoked, we enqueue a special packet to the mailbox that denotes the end of sending messages from the current PE to that mailbox.

**Selector.process()**: When the communication task receives a data packet through conveyor_pull, the mailbox’s process routine is invoked.

**Worker Loop**: The selector runtime creates a conveyor object for each mailbox and processes them separately within its own worker loop, as shown in Algorithm 1. When a mailbox is started, it creates a corresponding conveyor object (conv_obj) and a communication task that executes the algorithm shown in Algorithm 1. Initially, the communication task waits for data packets in the mailbox’s local buffer, which gets added when the user performs a send from the mailbox partition. During this polling for packets from the buffer, the communication task yields control to other tasks, as shown in Line 2. Once the data is added to the buffer, it breaks out of the polling loop and starts to drain elements from the buffer in Line 6. It then pushes each element in the buffer to the target PE in Line 11 until push fails. Then it removes all the pushed items from the buffer and starts the pull cycle. It pulls the received data in Line 16 and creates a computation task, which in turn invokes the mailbox’s process method, as shown in Line 18. As mentioned before, in case there is only one worker that is shared by all the tasks, we invoke the process method directly without the creation of any computation task. Once we come out of the processing of the received data, the task yields so that other communication tasks can share the communication worker.

Once the user invokes done, a special packet is enqueued to the buffer. When this special packet is processed, the is_done API in Line 5 returns true, thereby informing the conveyor object to start its termination phase. Once the communication of all remaining items is finished, the convoy_advance API returns false, thereby exiting the work loop. Finally the communication task terminates and signals the completion of the mailbox using a variable of type promise named as end_promise, as shown in Line 23.

The signaling of the promise schedules a dependent cleanup task which informs all dependent mailboxes, as shown in Figure 2, about the termination of the current mailbox. This task also manages a counter to find out when all the mailboxes in the selector have performed cleanup, to signal the completion of the selector itself using a future variable associated with the selector. Since the selector runtime is integrated with the HClib runtime, the standard synchronization constructs in AMT runtimes such as finish scopes and future can be used by the user to coordinate with the completion of the selector. Other dependent tasks can use the future associated with the selector to wait for its completion. Users can also wait for completion by using a finish scope; for example each of Lines 1-7 in Listing 4 and 2-14 in Listing 5 can be enclosed in finish scopes.

### D. Source-to-source translation from Lambda-based to Class-based messaging

While the use of C++ lambda expressions further simplifies writing remote message handlers (Section III-C), the performance of the lambda-based API is lower than that of the class-based version (Section III-E). This motivates us to perform automatic source-to-source translation from the lambda version to the class version to improve productivity without this performance loss. This kind of translation could be beneficial to other lambda-based libraries as well.

Figure 4 illustrates the end-to-end flow for the translation. The translator is a standalone tool built on top of Clang LibTooling. First, it identifies the use of the send API with a lambda expression by utilizing Clang LibTooling’s AST traversing a packet structure that is used for the class-based version. Then, it synthesizes a class declaration with a message handler and a packet struct type for actor messages.

**Listing 7**: An auto-translated version of Index-Gather.

```cpp
enum MB{Request, Response};
struct packet0 { ...
  3  int slot0;
  4  int slot1;
  5 ;
  6 };
  7 class SynthesizedSelector : public Selector<2, packet0> { ...
  8 public:
  9   int64_t *data; int64_t *gather;
 10   void process0(packet0 pkt, int sender_rank) { ...
 11    pkt.slot0 = data[pkt.slot0];
 12    send(Response, pkt, sender_rank); // pkt.slot1 is unchanged
 13   }
 14   void process1(packet0 pkt, int sender_rank) { ...
 15    gather[pkt.slot1] = pkt.slot0;
 16   }
 17   SynthesizedSelector(int64_t *ltable, int64_t *tgt) {...}
 18   SynthesizedSelector r_selector(data, gather);
 19   int sender_PK = shmem_my_pe();
 20   for (int i = 0; i < n; i++) {
 21    int col = index[i] / shmem_n_pes();
 22    int possibly_remote_PK = index[i] % shmem_n_pes();
 23    packet0 pkt0;
 24    pkt0.slot0 = col;
 25    pkt0.slot1 = i;
 26    r_selector.send(Request, pkt, possibly_remote_PK); ...
 27 }
```

**Listing 7** shows an auto-translated version of **Listing 5**. Comparing the two programs, one can see that it is feasible to automate this translation, thereby enabling the results in Section V to be obtained using lambda-based APIs. All benchmarks were automatically translated from lambda-based versions to class-based versions.
One interesting challenge that we also overcame was how to synthesize a minimum packet structure for all lambda-based messages. To understand this challenge, let us first revisit the original lambda version in Listing 5. There is a send() API call with a nested lambda on Line 18. Notice that, since each invocation of these lambda expressions can happen on a random remote PE, variables captured by a lambda need to be transferred. In this example, the scalar variables $\text{col}$ and $\text{i}$ are captured by the outer lambda, which is executed on possibly_remote_PE. Those variables need to transferred from sender_PE to possibly_remote_PE. Similarly, the scalar variables $\text{ret_val}$ and $\text{i}$ are supposed to be transferred back from possibly_remote_PE to sender_PE. Since the lifetimes of $\text{col}$ and $\text{ret_val}$ do not overlap, our translator assigns $\text{col}$ and $\text{ret_val}$ to slot0 and assigns $\text{i}$ exclusively to slot1 in Listing 7.

V. EVALUATION

This section presents the results of an empirical evaluation of our selector runtime system on a multi-node platform to demonstrate its performance and scalability. The goal of our evaluation is twofold:

1) to demonstrate that our selector-based programming system based on the FA-BSP model can be used to express a range of irregular mini-applications, and
2) to compare the performance of our approach with that of UPC, OpenSHMEM and Conveyors versions of these mini-applications.

Platform: We ran the experiments on the Cori supercomputer located at NERSC. In Cori, each node has two sockets, with each socket containing a 16-core Intel Xeon E5-2698 v3 CPU @ 2.30GHz (Haswell). For inter-node connectivity, Cori uses the Cray Aries interconnect with Dragonfly topology that has a global peak bisection bandwidth of 45.0 TB/s. We used cray-shmem 7.7.10, Berkeley UPC 2020.4.0 and GCC 8.3.0 (all available in Cori) to build all the software. Cray-shmem and Berkeley UPC in Cori use Cross-partition memory (XPMMEM) technology for cross-process mapping of user-allocated memory within a node that enables load-and-store semantics and native atomics. We used the same to obtain results in Table 1. We use one worker thread per PE rank for the experiments; since the mini-applications have sufficient parallelism across PE ranks, there is no motivation to use multiple worker threads within a single PE rank. The Conveyors library was compiled using cray-shmem for our experiments since cray-shmem provided the best performance based on our evaluation in Table 1. Conveyors can also use UPC or MPI as backends, in which case our Selectors library can also be invoked from any UPC or MPI program.

Mini-applications: We used all seven mini-applications in Bale [14], [12] that have Conveyors versions for our study. Bale can be seen as a proxy for key components in an irregular application that involve a large number of irregular point-to-point communication operations. All scalability results were obtained with weak scaling.

3We believe NERSC setup the modules with the best parameters for Cori.

![Fig. 5: Comparison of execution time of the UPC, OpenSHMEM, conveyor and selector variants (Y-axis: lower is better).](image-url)
of the matrix according to two random permutations.

The fourth mini-application solves the **random permutation** problem [24], which is to generate a random permutation of \(1 \ldots m\), assuming that each permutation is equally likely. This mini-application generates a random permutation in parallel using the “dart-throwing algorithm” [23].

The fifth mini-application performs **topological sorting** on a distributed sparse matrix. It uses an upper-triangular matrix (with ones on the diagonal) as its input. We then randomly permute the rows and columns of the upper-triangular matrix to obtain a new matrix. Our goal is to find a row and a column permutation such that when these permutations are applied, we can reconstruct an upper triangular matrix.

The sixth mini-application finds the **transpose** of a distributed sparse matrix in parallel.

The seventh mini-application performs **triangle counting** in a graph. The graph is represented as an adjacency matrix, which, in turn is stored using a sparse matrix data structure.

**Experimental variants:** Each mini-application was evaluated by comparing the following four versions. Among these, the UPC and OpenSHMEM versions were obtained from the Bale release [14] by replacing calls to libgetput with direct UPC and OpenSHMEM constructs (which resulted in a similar performance to that of the libgetput calls). The Conveyor versions were used unchanged from the Bale release. All problem sizes are identical to those used in the Bale release with one exception - the average number of nonzeros per row was reduced from 35 to 10 for all versions of the topological-sorting mini-application to make its execution time more comparable to that of the other mini-applications.

1) **UPC:** This version is written using UPC.
2) **OpenSHMEM:** This version is written using OpenSHMEM.
3) **Conveyor:** This version directly invokes the Conveyors APIs, which includes explicit handling of failure cases and communication progress.
4) **Selector:** This version uses the class-based version of the Selector API introduced in this paper, obtained by automatic translation from the lambda version as described in Section IV-D.

In Figures 5(a) to 5(g), the Y-axis shows the execution time in seconds, so smaller is better. Since we used weak scaling across PEs, ideal weak scaling should show the same time used by a mini-application for all PE counts. From the figures, we can see that the Conveyor versions perform much better than their UPC and OpenSHMEM counterparts. For the 2048 PE/core case, the Conveyor versions show a geometric mean performance improvement of 27.77× relative to the UPC and 21.52× relative to the OpenSHMEM versions, across all seven mini-applications.

This justifies our decision to use the Conveyors library for message aggregation in our Selector-based approach. Overall, we see that the Selector version also performs much better than the UPC/OpenSHMEM versions and close to the Conveyor version. For the 2048 PE/core case, the Selector versions show a geometric mean performance improvement of 25.59× relative to the UPC and 19.83× relative to the OpenSHMEM versions, and a geometric mean slowdown of only 1.09× relative to the Conveyor versions. These results confirm the performance advantages of our approach, while the productivity advantages can be seen in the simpler programming interface for the Selector versions relative to the Conveyor versions.

For completeness, Table II shows the source lines of code (SLOC) for different versions of the kernel of each mini-application, as measured by the CLOC utility [25]. The table convincingly shows that moving to the Actor/Selector model results in lower SLOC values relative to the Conveyor model, which in turn demonstrates higher productivity for the Actor/Selector model.

We can also create non-blocking OpenSHMEM operations using Actors, thus enabling message aggregation in those operations. An example using Actors to implement message aggregation enabled OpenSHMEM `get_nbi` and `put_nbi` operations are given at [26], [27].

### Table II: Kernel size of each mini-application in terms of source lines of code (SLOC)

| Mini-application | UPC | OpenSHMEM | Conveyor | Actor/Selector |
|------------------|-----|-----------|----------|----------------|
| Histogram        | 18  | 19        | 30        | 21             |
| Index-gather     | 16  | 17        | 40        | 25             |
| Permute-matrix   | 37  | 51        | 108       | 78             |
| Random-permutation | 41 | 43        | 111       | 99             |
| Topological-sort | 72  | 92        | 148       | 130            |
| Transpose        | 43  | 50        | 83        | 69             |
| Triangle-counting | 43 | 49        | 61        | 55             |

**VI. RELATED WORK**

The Chare abstraction in Charm++ [28] has taken inspiration from the Actor model, and is also designed for scalability. As indicated earlier, the performance of Charm++ is below that of Conveyors (and hence that of our approach) for the workloads studied in this paper.

In the past, there has been much work on optimizing the communication of PGAS programs through communication aggregation. Avolo et al. [29] used techniques such as static coalescing and the inspector-executor model to optimize communication in UPC. [30] introduced new PGAS language-aware LLVM passes to reduce communication overheads. Wesolowski et al. [31] introduced the TRAM library that optimizes communication by routing and combining short messages. UPC [15], [16] performs automatic message aggregation to improve the performance of fine-grained communication but is unable to achieve performance compared to user-directed message aggregation. We use Conveyors [12], which is a modular, portable, efficient, and scalable library as our message aggregation runtime.

**VII. CONCLUSIONS AND FUTURE WORK**

This paper proposes a scalable programming system for PGAS runtimes to accelerate irregular distributed applications. Our approach is based on the actor/selector model, and introduces the concept of a **Partitioned Global Mailbox**. Subparts of the application that involve a large number of latency tolerant communication operations can use our system to achieve high throughput, while other parts of the application can keep using
the PGAS communication system/runtime directly. Actors are often used as the concurrency mechanism in newer languages such as Scala or Rust and in other domains such as the cloud. Thus Actor’s addition to the PGAS ecosystem creates a low overhead path for users of other domains to develop high-performance computing applications without ramping up on non-blocking primitives. Moreover, we have shown that our Actor system beats the non-blocking operations in the state of the art communication libraries/systems by a handsome margin, thereby demonstrating the need to add/improve message aggregation in such libraries. Our programming system also abstracts away low-level details of message aggregation (e.g., manipulating local buffers and managing progress and termination) so that the programmer can work with a high-level selector interface. Further, this approach can be integrated with the standard synchronization constructs in asynchronous task runtimes (e.g., async-finish, future constructs). Our Actor runtime is more than a message-aggregation system since it also supports user-defined active messages, which can support the migration of computation closer to data that is beneficial for irregular applications. For the 2048 PE case, our approach show a geometric mean performance improvement of 25.59× relative to the UPC versions, 19.83× relative to the OpenSHMEM versions, and a geometric mean slowdown of only 1.09× relative to the Conveyors versions. These results suggest that the FA-BSP model offers a desirable point in the productivity-performance spectrum, with higher performance relative to PGAS models such as UPC and OpenSHMEM and higher productivity relative to the use of low-level hand-coded approaches for communication management and message aggregation.

In future work, we plan to support variable-sized messages since our current Mailbox implementation only accepts messages of a fixed size that is specified when creating a selector. Further, the original Selectors model [6] allows for operations such as mailbox priorities and enable/disable operations on mailboxes; support for such operations could enable richer forms of coordination logic across messages in our implementations. Actors could also be used to implement non-blocking communication primitives thereby they can get the message aggregation capability without directly interfacing with low-level aggregation libraries. Finally, it would be interesting to explore compiler extensions to automatically translate from the natural version to our selector version, thereby directly improving the performance of natural PGAS programs.

Artifact Availability: 

https://github.com/srirajpaul/hclib/tree/bale

actor/modules/bale_actor

REFERENCES

[1] C. Hewitt et al., “A universal modular ACTOR formalism for artificial intelligence,” in Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Stanford, CA, USA, August 20-23, 1972.

[2] G. A. Agha, ACTORS - a model of concurrent computation in distributed systems, ser. MIT Press series in artificial intelligence. MIT Press, 1990.

[3] R. Vermeersch, “Concurrency in erlang and scala: The actor model.”

[4] G. Agha, “Actors programming for the mobile cloud,” in 2014 IEEE 13th International Symposium on Parallel and Distributed Computing.

[5] J. D. Koster et al., “43 years of actors: a taxonomy of actor models and their key properties,” in Proceedings of the 6th International Workshop on Programming Based on Actors, Agents, and Decentralized Control, AGERE 2016, Amsterdam, The Netherlands, October 30, 2016.

[6] S. M. Imam and V. Sarkar, “Selectors: Actors with multiple guarded mailboxes,” in Proceedings of the 4th International Workshop on Programming based on Actors & Decentralized Control, AGERE' 2014, Portland, OR, USA, October 20, 2014. ACM, 2014, pp. 1–14.

[7] G. Taylor, D. Ozog, M. Wasi-ur Rahman, and J. Dinan, “Scalable machine learning with openshmem.”

[8] K. A. Yelick et al., “Cost and performance using partitioned global address space languages,” in Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007, University of Western Ontario, London, Ontario, Canada. ACM, 2007, pp. 24–32.

[9] R. W. Nurmich and J. Reid, “Co-array fortran for parallel programming,” SIGPLAN Forum, vol. 17, no. 2, p. 1–31, Aug. 1988.

[10] B. M. Chapman et al., “Introducing openshmem: SHMEM for the PGAS community,” in Proceedings of the 4th Conference on Partitioned Global Address Space Programming Model, PGAS 2010, New York, NY, USA.

[11] W. W. Carlson et al., “Introduction to upc and language specification,” Technical Report CCS-TR-99-157, IDA Center for Computing Sciences, Tech. Rep., 1999.

[12] F. M. Maley and J. G. Devinney, “Conveyors for streaming many-to-many communication,” in 9th IEEE/ACM Workshop on Irregular Applications: Architectures and Algorithms, IA3 SC 2019, Denver, CO, USA, November 18, 2019. IEEE, 2019, pp. 1–8.

[13] P. M. Kogge and S. K. Kurtz, “A case for migrating execution for irregular applications,” in Proceedings of the Seventh Workshop on Irregular Applications: Architectures and Algorithms, IA3@SC 2017, Denver, CO, USA, November 12 - 17, 2017. ACM, 2017, pp. 6:1–6:8.

[14] F. M. Maley and J. G. Devinney, “A collection of buffered communications libraries and some mini-applications.” https://github.com/jdevinney/bale, 2020, [Online; accessed 20-Apr-2020].

[15] W.-Y. Chen, “Building a source-to-source upc-to-c translator,” EECS Department, University of California, Berkeley; Tech. Rep. UCB/CSD-04-1369, Dec 2004.

[16] W.-Y. Chen, D. Bonachea, C. Iancu, and K. Yelick, “Automatic non-blocking communication for partitioned global address space programs,” in Proceedings of the 21st Annual International Conference on Supercomputing, ser. ICS ’07. New York, NY, USA: Association for Computing Machinery, 2007, p. 159–167.

[17] L. G. Valiant, “A bridging model for parallel computation,” Commun. ACM, vol. 33, no. 8, p. 103–111, aug 1990.

[18] M. C. Scherger, “An overview of the bsp model of parallel computation,” 2007. [Online]. Available: http://www.cs.ken.tu.edu/~jakber/PDA-Sp07/slices/bsp%20model.pdf

[19] M. Joyner, “Introduction to the actor model,” 2020. [Online]. Available: https://wikis.rice.edu/confluence/download_attachments/4455861/comp322-ec01lec21-slices-wide.pdf

[20] M. Grossman et al., “A pluggable framework for composable HPC scheduling libraries,” in 2017 IEEE International Parallel and Distributed Processing Symposium Workshops, IPDPS Workshops 2017. cplusplus.com, “Future,” 2020. [Online]. Available: http://www.cplusplus.com/reference/future/

[21] M. Grossman et al., “Integrating asynchronous task parallelism with openshmem,” in 3rd Workshop, OpenSHMEM 2016, Baltimore, USA.

[22] J. Gaspar, “Boost.Circular.Buffer.” https://www.boost.org/doc/libs/1_72_0/doc/html/circular_buffer.html [Online; accessed 20-Apr-2020].

[23] F. P. Gibbons et al., “Efficient low-contention parallel algorithms,” J. Comput. Sys. Sci., vol. 53, no. 3, pp. 417–442, 1996.

[24] “cloc.” [Online]. Available: http://manpages.ubuntu.com/manpages/netbook/10

[25] S. R. Paul, “shmem get nbi selector.” 2021. [Online]. Available: https://github.com/srirajpaul/hclib/blob/bale_actor/modules/bale_actor/test/get_selector_shmem.hpp

[26] S. R. Paul, “shmem put nbi selector.” 2021. [Online]. Available: https://github.com/srirajpaul/hclib/blob/bale_actor/modules/bale_actor/test/put_selector_shmem.hpp

[27] L. V. Kalé et al., “CHARM++: A portable concurrent object oriented system based on C++,” in Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), Eighth Annual Conference, Washington, DC, USA, September 26 - October 1, 1993.

[28] M. Alvanos et al., “Improving communication in PGAS environments: static and dynamic coalescing in UPC,” in International Conference on Supercomputing, ICS’13, Eugene, OR, USA - June 10 - 14, 2013. ACM.

[29] A. Hayashi et al., “Livism-based communication optimizations for PGAS programs,” in Proceedings of the Second Workshop on the LLVM Compiler Infrastructure at SC 2017, Austin, Texas, USA.

[30] L. Wesołowski et al., “TRAM: optimizing fine-grained communication with topological routing and aggregation of messages,” in 43rd International Conference on Parallel Processing, ICPP 2014.