A Productive and Scalable Actor-based Programming System for PGAS Applications

Sri Raj Paul[1], Akihiro Hayashi[2], Kun Chen[2], and Vivek Sarkar[2]

[1] Intel Corporation, USA, sriraj.paul@intel.com
[2] Georgia Institute of Technology, USA, {ahayashi,kunfz,vsarkar}@gatech.edu

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. 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 $\geq 20\times$ relative to standard PGAS versions (UPC and OpenSHMEM) while maintaining comparable productivity to those versions.

Keywords: Actors · Communication Aggregation · Conveyors · OpenSHMEM · PGAS · Selectors.

1 Introduction

In today's world, performance is improved mainly by increasing parallelism, thereby motivating the critical need for programming systems[3] that can deliver both productivity and scalability for parallel applications. The *Actor Model* [4] is the primary concurrency mechanism in languages such as Erlang and Scala,

[3] Following standard practice, we use the term, "programming system", to refer to both compiler and runtime support for a programming model.

ICCS Camera Ready Version 2022
To cite this paper please use the final published version:
DOI: 10.1007/978-3-031-08751-6_17

and is also gaining popularity in modern system programming languages such as Rust. Large-scale cloud applications [3] from companies such as Facebook and Twitter that serve millions of users are based on the actor model. Actors express communication using “mailboxes” [16]; the term, “selector” [11], 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.

The Partitioned Global Address Space (PGAS) model [20] is well suited to such irregular applications due to its efficient support for short, non-blocking one-sided messages. 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 [17] 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 [15].

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. Development of a source-to-source translator that translates our lambda-based API for actors to a more efficient class-based API.
4. Our results show a geometric mean performance improvement of 25.59× relative to the UPC versions and 19.83× relative to the OpenSHMEM ver-

sions, while using 2048 cores in the NERSC Cori system on seven irregular mini-applications from the Bale suite [18,17]

2 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.

Listing 1.1: An OpenSHMEM program that reads data from a distributed array.

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

Listing 1.2: An OpenSHMEM program that creates a histogram by updating a distributed array.

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

2.1 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 Listing 1.1. This program reads values from a distributed array named `data` and stores the retrieved values in a local array named `gather` based on global indices stored in a local array named `index`. The corresponding operation can also be performed in MPI using `MPI_Get`.

2.2 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 Listing 1.2. This program updates a distributed array named `histo` based on global indices stored in each PE's local `index` array using atomic increment, thereby creating a histogram. The corresponding operation can also be performed in MPI using `MPI_Accumulate` or `MPI_Get_Accumulate` or `MPI_Fetch_and_op`.

2.3 Conveyors

Conveyors [17] 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`: locally enqueue 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 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. 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 [6]. Analysis using Rice University's HPCToolkit [2] showed that the conveyors version reduced stall cycles by an order of magnitude compared to the OpenSHMEM version. As a result, we decided to use Conveyors as a lower-level library for automatic message aggregation in our programming system.

Communication System	Non Blocking	Read (sec)	Update (sec)
OpenSHMEM (cray-shmem 7.7.10)	N	35.5	NA
OpenSHMEM NBI (cray-shmem 7.7.10)	Y	4.2	4.3
UPC (Berkley-UPC 2020.4.0)	N	22.6	23.9
UPC NBI (Berkley-UPC 2020.4.0)	Y	19.7	NA
MPI3-RMA (OpenMPI 4.0.2)	Y	25.8	88.9
MPI3-RMA (cray-mpich 7.7.10)	Y	8.3	>300
Charm++ (6.10.1, gni-crayxc w/ TRAM)	Y	21.3	9.7
Conveyors (2.1 on cray-shmem 7.7.10)	Y	2.3	0.5

Table 1: Absolute performance numbers 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.

3 Our Approach

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

The classical Bulk-Synchronous Parallelism (BSP) [19] 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.

3.2 High-Level Design of Programming System

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 about 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

Abstracting *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 4.3.

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. Thus we differ from classical actors through the usage of partitioned global mailbox.

Among the two patterns discussed in Section 2, the *read* patterns differs from the *update* pattern in that it involves communication in two directions, namely request and response. In this case we can use 'Selector' [11], which is an actor with multiple mailboxes.

Progress and Termination In general, the Actors/Selectors model provides an `exit` [13] 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 1.2. 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.

3.3 User-facing API

Based on the discussions in Section 3.2, we provide a C/C++ based actor/selector programming framework as shown in Listing 1.3.

Listing 1.3: Actor/Selector Interface with partitioned global mailboxes.

```
//L: lambda type

class Actor<L> {
  void send(int PE, L msg);
  void done();
};

class Selector<N, L> { // N mailboxes
  void send(int mailbox_id,
            int PE, L msg);
  void done(int mailbox_id);
};
```

Listing 1.4: Actor version of the Update benchmark (Histogram) using lambda.

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

Update:[4] Listing 1.4 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: 3 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.

[4] Showing only update pattern due to page limitation. Read pattern is in the artifact.

3.4 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 (Listing 1.5) 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. In the class based version, user need to express message handling using the `process` API and explicitly construct the message used in `send` API.

Listing 1.5: Actor/Selector class-based interface with partitioned global mailboxes.

```
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;
  }
};
```

4 Implementation

In this section, we discuss the implementation of the selector runtime prototype created by extending HClib [9], a C/C++ Asynchronous Many-Task (AMT) Runtime library. We first discuss our execution model in Section 4.2 and then describe our extensions to the HClib runtime to support our selector runtime in Section 4.3.

4.1 HClib Asynchronous Many-Task Runtime

Habanero C/C++ library (HClib) [9] is a lightweight asynchronous many-task (AMT) programming model-based runtime. It uses a lightweight work-stealing scheduler to schedule the tasks. HClib uses a persistent thread pool called workers, on which tasks are scheduled and load balanced using lock-free concurrent deques. 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 [7]. 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.

4.2 Execution Model

Figure 1 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 [9] Asynchronous Many-Task (AMT) runtime library.

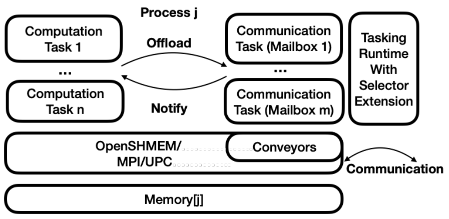
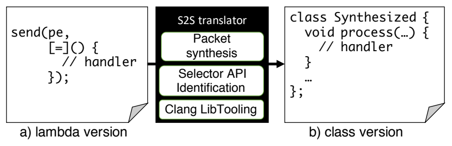


Fig. 1: The execution model showing internal structure of tasks and mailboxes within a single PE.



```
send(pe,
     [=]() {
       // handler
     });
```

```
class Synthesized {
  void process(…) {
    // handler
  }
  …
};
```

Fig. 2: Source-to-source translator from lambda version to class based version.

To enable asynchronous communication, the computation tasks offload all remote accesses on to the communication tasks. 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.

4.3 Selector Runtime

The implementation details presented are based on the class-based interface introduced in Section 3.4, since our results were obtained by converting the lambda API to the class-based API using the translator described in Section 4.4. As mentioned earlier, we hide the low-level details of Conveyors operations from the programmer and incorporate them into our Selector runtime instead. To

Algorithm 1 Worker loop associated with each mailbox

```
1: while buff.isempty() do
2:     yield()                                  ▷ yield until message is pushed to buffer
3: end while
4: pkt ← buffer[0]
5: while convey_advance(conv_obj, is_done(pkt)) do
6:     for i ← 0 to buffer.size-1 do
7:         pkt ← buffer[i]
8:         if is_done(pkt) then
9:             break
10:        end if
11:        if convey_push(conv_obj, pkt.data,pkt.rank) then
12:            break
13:        end if
14:    end for
15:    buffer.erase(0 to i)
16:    while convey_pull(conv_obj, &data, &from) do
17:        create computation_task(
18:            process(data, from)
19:        )
20:    end while
21:    yield()
22: end while
23: end_promise.put(1)                          ▷ To signal completion of mailbox
```

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[5] that is based on the Boost Circular Buffer library [8]. 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.

[5] This local buffer is different from the Conveyor's internal buffer.

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 `convey_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 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` scope 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.

4.4 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 3.3), during experiments the performance of the lambda-based version was found to be 2× slower than that of the class-based version (Section 3.4). 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 2 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 traversal APIs. For each lambda, it analyzes captured variables to synthesize 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.

5 Evaluation

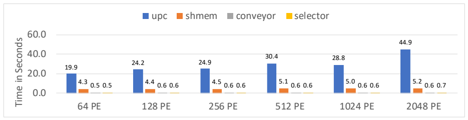
This section presents the results of an empirical evaluation of our selector runtime system on a multi-node platform to demonstrate its performance and scalability. The goal of our evaluation is twofold:

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.

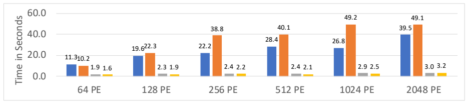
Platform: We ran the experiments on the Cori supercomputer located at NERSC. In Cori, each node has two sockets, with each socket containing a 16-core Intel Xeon E5-2698 v3 CPU @ 2.30GHz (Haswell). For inter-node connectivity, Cori uses the Cray Aries interconnect with Dragonfly topology that has a global peak bisection bandwidth of 45.0 TB/s. We use one worker thread per PE rank for the experiments; since the mini-applications have sufficient parallelism across PE ranks, there was 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.
Mini-applications: We used all seven mini-applications in Bale [18,17] 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.

Experimental variants:

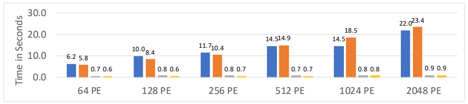
1. **UPC:** This version is written using UPC.
2. **OpenSHMEM:** This version is written using OpenSHMEM.
3. **Conveyor:** This version directly invokes the Conveyors APIs, which includes explicit handling of failure cases and communication progress.
4. **Selector:** This version uses the class-based version of the Selector API introduced in this paper, obtained by automatic translation from the lambda version as described in Section 4.4.



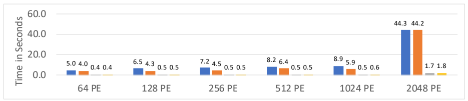
(a) Histogram mini-application with 10,000,000 updates per PE on a distributed table with 1,000 elements/PE.



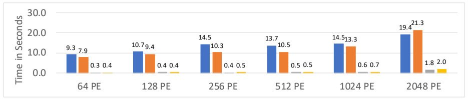
(b) Index-gather mini-application with 10,000,000 reads per PE on a distributed table with 100,000 elements/PE.



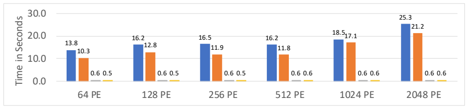
(c) Permute-matrix mini-application with 100,000 rows of the matrix/PE with an average of 10 nonzeros per row.



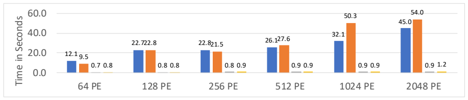
(d) Random-permutation mini-application with 1,000,000 elements per PE.



(e) Topological-sort mini-application with 100,000 rows of the matrix/PE with an average of 10 nonzeros per row.



(f) Transpose-matrix mini-application with 100,000 rows of the matrix per PE with an average of 10 nonzeros per row.



(g) Triangle-counting mini-application with 10,000 rows of the matrix per PE with an average of 35 nonzeros per row.

Fig. 3: Comparison of execution time of the UPC, OpenSHMEM, conveyor and selector variants (lower is better for the Y-axis).

In Figures 3(a) to 3(g), the Y-axis shows the weak scaling average execution time of five runs in seconds, so smaller is better. 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.

Table 2 shows the source lines of code (SLOC) for different versions of the kernel of each mini-application, as measured by the CLOC utility [1]. 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.

	Histogram	Index-gather	Permute-matrix	Random-permutation	Topological-sort	Transpose	Triangle-counting
UPC	18	16	37	41	72	43	43
OpenSHMEM	19	17	51	43	92	50	49
Conveyor	30	40	108	111	148	83	61
Actor/Selector	21	25	78	99	130	69	53

Table 2: Kernel size of each mini-application in terms of source lines of code.

6 Related Work

The actor is the primary concurrency mechanism in Scala, however it is not scalable for HPC workloads [5]. The Chare abstraction in Charm++[14] 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. Jenkins *et al.* [12] created the Chapel Aggregation Library (CAL) which aggregates user-defined data using an Aggregator object. UPC [6] performs automatic message aggregation to improve the performance of fine-grained communication but is unable to achieve performance compared to user-directed message aggregation.

7 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*. 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. 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, 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. We would also like to improve our performance result reporting based on the paper [10].

Artifact: `https://github.com/srirajpaul/hclib/tree/bale_actor/modules/bale_actor`

References

1. cloc, `http://manpages.ubuntu.com/manpages/man1/cloc.1.html`
2. Adhianto, L., et al.: HPCTOOLKIT: tools for performance analysis of optimized parallel programs. Concurr. Comput. Pract. Exp. **22**(6), 685–701 (2010). https://doi.org/10.1002/cpe.1553
3. Agha, G.: Actors programming for the mobile cloud. In: 2014 IEEE 13th International Symposium on Parallel and Distributed Computing. pp. 3–9 (2014). https://doi.org/10.1109/ISPDC.2014.31
4. Agha, G.A.: ACTORS - a model of concurrent computation in distributed systems. MIT Press series in artificial intelligence, MIT Press (1990)
5. Charousset, D., et al.: Revisiting actor programming in c++. Comput. Lang. Syst. Struct. **45**(C), 105–131 (Apr 2016). https://doi.org/10.1016/j.cl.2016.01.002
6. Chen, W.Y.: Building a source-to-source upc-to-c translator. Tech. rep., EECS Department, University of California, Berkeley (Dec 2004)
7. cplusplus.com: Future (2020), `http://www.cplusplus.com/reference/future/`
8. Gaspar, J.: Boost.Circular Buffer. `https://www.boost.org/doc/libs/1_72_0/doc/html/circular_buffer.html`, [Online; accessed 20-Apr-2020]
9. Grossman, M., et al.: A pluggable framework for composable HPC scheduling libraries. pp. 723–732. IEEE Computer Society (2017). https://doi.org/10.1109/IPDPSW.2017.13
10. Hoefler, T., Belli, R.: Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results. SC '15 (2015). https://doi.org/10.1145/2807591.2807644
11. Imam, S.M., Sarkar, V.: Selectors: Actors with multiple guarded mailboxes. pp. 1–14. ACM (2014). https://doi.org/10.1145/2687357.2687360
12. Jenkins, L., et al.: Chapel aggregation library (CAL). In: 2018 IEEE/ACM Parallel Applications Workshop, Alternatives To MPI, PAW-ATM SC 2018. pp. 34–43. IEEE (2018). https://doi.org/10.1109/PAW-ATM.2018.00009
13. Joyner, M.: Introduction to the actor model (2020), `https://wiki.rice.edu/confluence/download/attachments/4435861/comp322-s20-lec21-slides-wide.pdf`
14. Kalé, L.V., et al.: CHARM++: A portable concurrent object oriented system based on C++. In: Object-Oriented Programming Systems, Languages, and Applications (OOPSLA). ACM (1993). https://doi.org/10.1145/165854.165874
15. Kogge, P.M., Kuntz, S.K.: A case for migrating execution for irregular applications. In: Workshop on Irregular Applications: Architectures and Algorithms, IA3@SC 2017. pp. 6:1–6:8. ACM (2017). https://doi.org/10.1145/3149704.3149770
16. Koster, J.D., Cutsem, T.V., Meuter, W.D.: 43 years of actors: a taxonomy of actor models and their key properties. pp. 31–40. ACM (2016). https://doi.org/10.1145/3001886.3001890
17. Maley, F.M., DeVinney, J.G.: Conveyors for streaming many-to-many communication. In: Workshop on Irregular Applications: Architectures and Algorithms, IA3 SC 2019. pp. 1–8. IEEE (2019). https://doi.org/10.1109/IA349570.2019.00007
18. Maley, F.M., DeVinney, J.G.: A collection of buffered communication libraries and some mini-applications. `https://github.com/jdevinney/bale` (2020)
19. Valiant, L.G.: A bridging model for parallel computation. Commun. ACM **33**(8), 103–111 (aug 1990). https://doi.org/10.1145/79173.79181
20. Yelick, K.A., et al.: Productivity and performance using partitioned global address space languages. In: Parallel Symbolic Computation, PASCO 2007, International Workshop, 27-28 July 2007. https://doi.org/10.1145/1278177.1278183