Towards Safe HPC: Productivity and Performance via Rust Interfaces for a Distributed C++ Actors Library (Work in Progress)

John Parrish
jparrish34@gatech.edu
Georgia Institute of Technology
USA

Nicole Wren*
nicole@wren.systems
Block
USA

Tsz Hang Kiang
tszhang.kiang@gatech.edu
Georgia Institute of Technology
USA

Akihiro Hayashi
ahayashi@gatech.edu
Georgia Institute of Technology
USA

Jeffrey Young
jyoung9@gatech.edu
Georgia Institute of Technology
USA

Vivek Sarkar
vsarkar@gatech.edu
Georgia Institute of Technology
USA

Abstract

In this work-in-progress research paper, we make the case for using Rust to develop applications in the High Performance Computing (HPC) domain which is critically dependent on native C/C++ libraries. This work explores one example of Safe HPC via the design of a Rust interface to an existing distributed C++ Actors library. This existing library has been shown to deliver high performance to C++ developers of irregular Partitioned Global Address Space (PGAS) applications.

Our key contribution is a proof-of-concept framework to express parallel programs `safe`-ly in Rust (and potentially other languages/systems), along with a corresponding study of the problems solved by our runtime, the implementation challenges faced, and user productivity. We also conducted an early evaluation of our approach by converting C++ actor implementations of four applications taken from the Bale kernels to Rust Actors using our framework.

Our results show that the productivity benefits of our approach are significant since our Rust-based approach helped catch bugs statically during application development, without degrading performance relative to the original C++ actor versions.

CCS Concepts: • **Software and its engineering** → **Software safety**; • **Theory of computation** → *Distributed computing models.*

*This work was done while the author was at the Georgia Institute of Technology

MPLR '23, October 22, 2023, Cascais, Portugal

ACM ISBN 979-8-4007-0380-5/23/10.
https://doi.org/10.1145/3617651.3622992

Keywords: Rust, high performance computing, actors, unsafe annotations

ACM Reference Format:
John Parrish, Nicole Wren, Tsz Hang Kiang, Akihiro Hayashi, Jeffrey Young, and Vivek Sarkar. 2023. Towards Safe HPC: Productivity and Performance via Rust Interfaces for a Distributed C++ Actors Library (Work in Progress). In *Proceedings of the 20th ACM SIGPLAN International Conference on Managed Programming Languages and Runtimes (MPLR '23), October 22, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3617651.3622992

1 Introduction

Rust is a natural candidate to explore for enabling both productivity and performance in high-performance computing (HPC) applications due to simultaneously having a rich type system with memory-safety guarantees, as well as the ability to write specialized low-level code without the additional overhead that comes with managed runtimes.

One challenge in using Rust for HPC applications is in the writing of low-level APIs and in using APIs from another language (like C) that cannot be written easily or "safely" in Rust. Low-level primitives frequently cannot be verified as safe by the Rust compiler, due to the compiler's lack of knowledge about hardware or inability to recognize the implicit guarantees of an API. Rust has a special keyword, `unsafe`, which allows programmers to annotate specific blocks of code to be bypassed by the compiler's safety checks. The general design pattern for Rust libraries is to wrap necessary `unsafe` logic behind a safe interface that forces the user to meet Rust's safety guarantees.

In this paper, we summarize our experiences with implementing Rust interfaces for a recently developed high-performance C++ actor-based programming system for Partitioned Global Address Space (PGAS) applications [12]. This library enables the handlers for actor messages to be specified as C functions or as C++ lambdas. Following the terminology introduced in [9], we refer to this library as the Selectors library due to its support for actors with multiple

mailboxes. While Rust's Foreign Function Interface (FFI) enables low-overhead invocation of C, the callback nature of message handlers in the Selectors library raises challenges in avoiding the use of `unsafe` annotations. This work allows developers to write application logic in Rust while invoking this PGAS C/C++ library with minimal or no `unsafe` annotations. Throughout the paper we refer to code that must be annotated unsafe as `unsafe` where "unsafe" without any emphasis is reserved for code that has some fundamentally memory-unsafe quality about it. It is important to make this distinction because Rust's `unsafe` annotation should be used for safe code that cannot be automatically verified rather than fundamentally unsafe code.

This paper makes the following contributions:

1. A novel implementation of a Rust-based Selectors library that allows for developing PGAS-based HPC applications
2. An evaluation of this new implementation explores design tradeoffs needed to minimize unsafe access patterns when combining Rust and C++ codebases, especially for the Selectors API
3. Refinements of the Rust library are discussed as ways to further improve the safety of programs written with this interface

2 Background

2.1 HClib Actor/Selector Library

The Habanero C/C++ library (HClib) [6] is an asynchronous many-task (AMT) programming model-based runtime. The authors of HClib have recently introduced an actor-based programming system for HPC platforms [7, 12], where 1) they integrate the SPMD (Single Program Multiple Data) programming model with the conventional actor model [2, 8], 2) they accelerate asynchronous actor messaging by leveraging a high-performance message aggregation library (Conveyors [10]), and 3) they provide a mechanism that automatically automatically handles termination detection for actors with minmal input from the programmer.

Specific to the Rust-based Actor/Selector library, a brief summary of the relevant APIs that are adapted from HClib is as follows:

1. `launch`: Used to initialize the HClib runtime. This includes spawning the worker threads.
2. `send`: Used to send an asynchronous message to a remote actor or Processing Element (PE), which is typically a worker thread tied to a single physical core.
3. `done`: Used to inform the runtime that the current actor/PE will not send any more messages to a specific mailbox so as to aid the runtime with overall application termination.
4. `finish`: Used for bulk task synchronization. It waits on all tasks (including nested tasks) spawned within the scope of the finish. Specific to the actor runtime, it waits until all outgoing messages are sent and all incoming messages are processed on the current PE.
5. `async`: Used to create asynchronous tasks.
6. `async_await`: Used to create event-driven tasks that uses `future`'s to specify task dependencies.

As with [12], the terms "actor" and "selector", the latter of which is "actors with multiple mailboxes" [9], will be used interchangeably for the remainder of the paper.

2.2 Rust's Foreign Function Interface and `Unsafe` Annotation

Rust supports the use of externally-defined functions using the C ABI. This works in much the same way as it does in C++ when linking to C libraries. The relevant functions are declared inside an `extern` block, and the linking information must be specified (name/location of library to link). Because the C++ ABI is generally more complex and less consistent across different compilers, Rust does not support linking to C++ functions directly.

More importantly, any external functions are automatically considered `unsafe` by Rust because the Rust compiler cannot verify the safety of external functions. For instance, it has no way of knowing that such functions will not perform an improper access or save a supplied pointer that may be dereferenced later after the Rust application has deallocated the pointed-to object (lifetime verification).

2.3 Conveyors

A Conveyor is an abstraction that "conveys" messages between processes in parallel applications. For portability, the Conveyors library can be built atop SHMEM, MPI, or UPC.

Conveyors are interfaced with a few core operations, of which `push`, `pull` and `advance` are most important for our later discussion. The `push` operation "enqueue[s] an item for delivery to a specified process" while the `pull` operation "receive[s] an item and learn[s] which process sent it". The `advance` operation is used to "ensure forward progress" [10].

3 Rust Selectors API

3.1 Goals

Our approach to making the HClib Actor/Selector library APIs usable from Rust applications can be summarized by the following goals:

- Feature parity with the original C++ HClib library
- Similarity in interface to the C++ HClib library; Rust and C++ versions of applications should look similar, making it simple to translate applications from C++ to Rust
- Minimal friction with Rust's type checking and minimal use of unsafe application code

```rust
pub trait Selector<'s, const N: usize> {
    type Packet: Send + Sync;
    fn handlers()
        -> MailboxHandlers<'s, Self, N>
    fn create(&'s mut self)
        -> SelectorAddr<'s, Self, N>
    // ...
}

pub struct Mailbox<'s, S, const N: usize>
where
    S: Selector<'s, N>,
{
    state: *mut S,
    // ...
}

pub struct SelectorContext<'s, S, const N: usize>
where
    S: Selector<'s, N>,
    S::Packet: Send + Sync,
{
    mailboxes:
        Arc<MaybeUninit<[Mailbox<'s, S, N>; N]>>,
}


impl<'s, S, const N: usize> Mailbox<'s, S, N>
where
    S: Selector<'s, N>,
    S::Packet: Send + Sync,
{
    pub fn new(
        handle: SelectorContext<'s, S, N>,
        state: *mut S,
        process: MailboxHandler<'s, S, N>,
    ) -> Self
    pub fn start(&self)
    // ...
}

pub type MailboxHandler<'s, S, const N: usize> =
    Box<dyn Fn(&SelectorContext<'s, S, N>,
        &mut S, <S as Selector<'s, N>>::Packet, i32)>;
pub type MailboxHandlers<'s, S, const N: usize> =
    [MailboxHandler<'s, S, N>; N];


pub fn finish_with<'a, F, R>(f: F) -> R
where
    F: FnOnce(&FinishScope<'a>) -> R
```

Figure 1. Main Rust interface to the Selectors runtime

3.2 Example Applications and Comparisons

We present several figures that demonstrate the typing and usage of the API functions in our framework. Where applicable, we provide both Rust and C++ versions of the code snippets. Outlines of the primary Rust API functions are given in Figure 1. In Figures 2 and 3, we show usage of the `finish` construct in the Index Gather kernel, written with the Rust and C++ interfaces respectively. In Figures 4 and 5, we have code for the Index Gather kernel in both Rust and C++.

4 Bringing Safety to the API

It is less accurate to say that we directly wrapped the C++ Selectors API than it is to say that we wrapped its dependencies

```rust
hclib::finish_with(|scope| {
  ig_sel.start(scope);
  for (i, &idx) in pckindx.iter().enumerate() {
    let pkt = IgPkt {
      idx: i as i64,
      val: idx >> 16,
    };
    let dest_rank: i32 = (idx & 0xffff) as i32;
    ig_sel.send_to(
      MailBoxType::Request as usize, pkt, dest_rank);
  }
  ig_sel.done_on(MailBoxType::Request as usize);
});
```

Figure 2. Rust main program code for Index Gather kernel using the `finish` construct

```cpp
hclib::finish([=]() {
  igs_ptr->start();
  for(int i=0;i<l_num_req;i++) {
    IgPkt pkt;
    pkt.idx = i;
    pkt.val = pckindx[i] >> 16;
    int dest_rank = pckindx[i] & 0xffff;
    igs_ptr->send(REQUEST, pkt, dest_rank);
  }
  igs_ptr->done(REQUEST);
});
```

Figure 3. C++ main program code for Index Gather kernel using the `finish` construct

and then reimplemented the API. We wrapped conveyors and the core of HClib, and then we re-implemented the HClib Actor APIs on top of these. Conceptually, it was important to make sure that the API we choose for Conveyors and the lower-level HClib primitives to be safe because these safety constraints "bubble up" into the higher level APIs. In this section, we discuss our efforts to make some of these underlying APIs safe and the general lessons we think can be extracted from them.

4.1 Wrapping Conveyors

A natural question when creating a Rust API for Conveyors is "what types are safe to send between PEs?". Conveyors send data by shallow copying that data between buffers. In fact, we can ignore the specifics of the transmission and view it abstractly as a shallow-copy between distributed nodes. This high-level view also applies to data transmission using other distributed-parallelism APIs such as MPI. Often this shallow-copying takes the form of `memcpy` or a similar library call.

When assessing what types are safe to send across PE bounds, we need to determine what types can be safely shallow-copied. For example, Rust's primitive types can be safely transmitted because they can be safely shallow-copied.

```rust
struct IgSelector {
  base: Selector<IgPkt, 2>,

  ltable: *mut [i64],
  tgt: *mut [i64],
}

impl IgSelector {
  pub fn new(ltable: *mut [i64], tgt: *mut [i64])
    -> Box<Self> {
    let mut this: Box<MaybeUninit<Self>> =
      Box::new(MaybeUninit::uninit());

    let _this = this.as_mut_ptr();

    let mb0 = move |pkt: IgPkt, sender_rank: i32| unsafe {
      (*_this).req_process(pkt, sender_rank);
    };

    let mb1 = move |pkt: IgPkt, sender_rank: i32| unsafe {
      (*_this).resp_process(pkt, sender_rank);
    };

    *this = MaybeUninit::new(IgSelector {
      base: Selector::new([Box::new(mb0), Box::new(mb1)]),
      ltable,
      tgt,
    });

    unsafe { Box::from_raw(Box::into_raw(this) as *mut _) }
  }

  fn req_process(&mut self, mut pkt: IgPkt, sender_rank: i32) {
    unsafe {
      pkt.val = (*self.ltable)[pkt.val as usize];
    }
    self.base
      .send(MailBoxType::Response as usize, pkt, sender_rank);
  }

  fn resp_process(&mut self, pkt: IgPkt, _sender_rank: i32) {
    unsafe {
      (*self.tgt)[pkt.idx as usize] = pkt.val;
    }
  }
}
```

Figure 4. Rust Code for Index Gather kernel using the new APIs for the Selectors library with the baseline approach

```
class IgSelector: public hclib::Selector<2, IgPkt> {

  //shared table array src, target
  int64_t * ltable, *tgt;

  void req_process(IgPkt pkt, int sender_rank) {
    pkt.val = ltable[pkt.val];
    send(RESPONSE, pkt, sender_rank);
  }

  void resp_process(IgPkt pkt, int sender_rank) {
    tgt[pkt.idx] = pkt.val;
  }

public:
  IgSelector(int64_t *ltable, int64_t *tgt) :
    ltable(ltable), tgt(tgt) {
      mb[REQUEST].process =
        [this](IgPkt pkt, int sender_rank) {
          this->req_process(pkt, sender_rank);
        };
      mb[RESPONSE].process =
        [this](IgPkt pkt, int sender_rank) {
          this->resp_process(pkt, sender_rank);
        };
  }

};
```

Figure 5. Original code for Index Gather kernel using the C++ APIs for the Selectors library

Rust's `String`–on the other hand–cannot be safely shallow-copied because it contains a pointer to the backing heap allocation. Allowing new `String`s to be created by shallow-copying can lead to potentially unsafe code because if Rust deallocates the original `String`, the copied `String` will contain a pointer to the now-deallocated backing storage. To account for this, we consider the data moved into the conveyor when it gets pushed. However, there is another issue of safety on the receiver end. Assuming a `String`'s backing array is allocated in local storage, any use of the pointer to the backing storage by the receiver will be malformed. The backing-array pointer refers to storage on the sender and cannot be used by the receiver.

The appropriate type bound should capture the type of any object that can be safely duplicated by shallow-copy. The semantically closest trait Rust offers is `Copy`. This trait is a marker that is meant to indicate that a value can be copied with zero or little cost compared to moving. It is implemented–for instance–by primitives because it is an elementary operation to copy the value of a primitive to a different variable. `String`, on the other hand, is not marked `Copy` because properly duplicating a `String` would involve a potentially costly deep copy of the backing data. However, we cannot rely on the convention that a `Copy` type should be shallow-copiable, as it is only by convention. If a user so desired, they could safely make any type which implements `Clone`–which provides a (potentially costly) way to duplicate the type–also implement `Copy`. Another option is requiring that the type has the static lifetime. This captures that the type's validity is not tied to the validity of another object. A reference to a local variable, for instance, would not satisfy the static lifetime because it cannot outlive the value it references. Unfortunately, `String` and similar types also have a static lifetime because–although they reference externally allocated memory–they own the backing data. The only way to safely deallocate the backing data for a `String` is to drop that `String`.

Seeing as the standard trait bounds are not sufficient for ensuring a type can be safely transmitted using Conveyors, we propose a new trait that encapsulates when a type can safely be shallow copied. Rust's `Send` and `Sync` traits, which are used to mark the safety of sending/sharing types between threads, provide a model for a similar marker trait. We propose adding an `unsafe` marker trait that is meant to mark types which can be safely shallow copied and thus safely sent between PEs in a distributed parallelism message-passing context such as Conveyors or MPI. Users would have to use `unsafe` code to mark their custom types as able to be safely sent between PEs. The user is thus responsible for ensuring that this contract is upheld, but this is a safer API because, while users may erroneously mark types, they cannot use a type that has not been marked as the message type for a conveyor.

4.2 Role of Finish Construct in Establishing Lifetimes

Async tasks in the HClib C API are akin to Rust's standard threads; they describe asynchronous computation in closures

which are then passed to library functions to be scheduled and executed.

In HClib, any `async` task launched within a `finish` block needs to complete before the `finish` exits. Thus, it is sound lifetime-wise for code inside an `async` task spawned inside a `finish` block to reference any data that can be referenced from the `finish` block's scope.

The vanilla standard thread model is not powerful enough to encapsulate this invariant because the borrow checker is unable to understand that the lifetimes of data from outside the thread will outlive the thread itself.

Since we are unable to prove the soundness of this logic to the borrow checker, this naturally calls for `unsafe`. We do exactly this, by modelling HClib tasks after Rust's scoped threads [1] which uses `unsafe` under the hood to annotate lifetimes. Scoped threads are launched from within a scoped block, and joined upon exiting the block. Each scoped thread gets annotated with a `'scope` lifetime that spans the scope block. As a result, any variable with a lifetime that encompasses and outlives `'scope` can be borrowed within the scoped thread. We illustrate this interface in Figure 6.

```
extern crate hclib;

#[hclib::hclib_entrypoint("system")]
fn main() {
    let mut x = 0;
    let y = 1;
    hclib::finish_with(|scope| {
        hclib::r#async(|| {
            println!(
                "Without scoping, variables cannot be \
                borrowed without a static reference"
            );
            // won't compile!
            // println!("{}", &y);
        });

        scope.r#async(|| {
            x += 2;
            println!(
                "Variables can be borrowed mutably by one \
                async as usual: x = {}",
                x
            );
        });

        for _ in 1..=3 {
            scope.r#async(|| {
                println!(
                    "Variables can be immutably borrowed \
                    from many asyncs: y = {}",
                    &y
                );
            });
        }

        let z = 2;
        scope.r#async(|| {
            println!(
                "Variables can be borrowed as long as \
                the reference can outlive the \
                finish scope: y = {}",
                &y
            );
            // won't compile!
            // println!("{}", &z);
        });
    });
}
```

Figure 6. Working examples demonstrating usages of FinishScope to allow references in asynchronous tasks that the Rust borrow checker would otherwise deem `unsafe`

5 Inheritance and Circular References

5.1 General Approach for Removing Circular References

One pain point of porting C++ frameworks like Habanero Selectors to Rust is that often times these frameworks will contain complex webs of circular references. Cycles can be relatively benign in C++, but in Rust they can quickly cause problems. Reference cycles will likely lead to memory leaks and they can make the construction process impossible to do safely.

For example, in the C++ Selector implementation, the selector contains mailboxes, which then contain lambdas, which reference the containing actor. We solved this particular issue by separating the actor state–which the lambdas need to update–from the containing selector.

Another example of this circular structure appears between the various mailboxes in the selector. Because each `Mailbox` is responsible for sending to the matching mailboxes on the different actors, the `Mailboxes` must maintain contact with one another. In the C++ implementation this is done by maintaining a pointer back to the containing actor which then acts as an arbiter for getting messages from one mailbox's handler into another mailbox's conveyor as an outgoing message. This cycle is harder to break because of how tightly bound–at least conceptually–the mailbox is to its conveyor. One solution, however, is to separate the Conveyors from the Mailboxes, storing each in their own Array. The array of Conveyors would need to be stored in an Arc because each Conveyor needs access to the whole array. Then, each Mailbox could keep track of which Conveyor it needs to pull from in order to retrieve messages to process. When the `Mailbox` needs to send a message, instead of going to another `Mailbox`, it can directly access the Conveyor it needs to transmit over.

The general method we have used to untangle these dependencies is to try and decouple the various functionalities as much as possible. While it made sense to couple the selector "administration" with the selector state or the `Mailbox` management with the Conveyor management in C++, coupling these responsibilities together made it almost impossible to specify the structure in safe Rust. As a general rule, when porting complicated structures like these to Rust, one may have to give up encapsulation of responsibilities in order to untangle cyclic references.

5.2 Current Implementation

The latest version of our API takes inspiration from other actor libraries like the Actix library [11] for Rust. The application writer now defines a struct which implements the `Selector<'s, N>` trait. `N` is a usize constant referring to the number of mailboxes, and `'s` is the associated lifetime of the selector object and all of its internal handlers. The trait has one required method:

```
fn handlers()
    -> MailboxHandlers<'s, Self, N>
```

and one associated type: `Packet`. The associated type refers to the type of packet that will be sent and received by the selector. The `handlers` method returns an array of closures which are the handlers of the selector. Each handler now takes in a `SelectorContext` that gives the closure access to the other mailboxes through which they can send messages. The handler is also passed a mutable reference to the state object. This eliminates the need for the handlers to capture a pointer or other reference to the state object. This allows the handlers to be written using entirely safe code. The application writer now calls the function:

```
fn create(&'s mut self)
    -> SelectorAddr<'s, Self, N>
```

on the custom `Selector` implementation to get a `SelectorAddr` which then behaves as the `Selector` from the original C++ API.

In the backend of the API, `unsafe` code is still present. This is due to the circular reference structure between mailboxes and the sharing of the state data between the various mailboxes. The former requires the use of `MaybeUninit` and the latter requires using a raw mutable pointer to maintain access to the state object. Under the current model of HClib Selectors, only one hardware-thread is used to run the runtime, so it is correct to give message handlers safe, exclusive access to the selector state.

5.3 Actors Sending Messages to Themselves

Another part of the C++ applications that proved difficult to translate to Rust came from the application code manually manipulating the state simultaneously with the `Selector`'s manipulation. In the Rust implementation, the `Selector` owns the data while it is processing it, so these parallel manipulations had to be marked `unsafe`. The reason that these manipulations are in-fact safe is because HClib implements cooperative multithreading with well-defined synchronization points at which control can change hands. Therefore, truly simultaneous updates – which would result in race conditions – are not possible. The fact that these code sections must be marked `unsafe` is unsavory, but the required `unsafe` serves as a marker for the programmer to pay attention to the potentially delicate code which relies on the synchronization of HClib.

Ideally, however, we would like to remove all necessary instances of `unsafe` code from these applications by creating a safe way to accomplish the same thing using the API. We believe that these parallel mutation behaviors need not be parallel at all. In the applications we have reviewed with this property, the `unsafe` mutation can be incorporated into the flow of the application by sending messages from the local actor to itself. This may potentially incur the cost of an additional mailbox if the update is not one which is already included in the actor logic, but the overheads for local-only message passing should be minimal.

6 Evaluation

We wanted to ensure that the gains in safety and productivity with our Rust interface did not come at the cost of performance. Rust's zero-cost abstractions imply that it is theoretically possible to have no performance sacrifices.

To assess our runtime's performance, we selected a subset of the Bale kernels [4] that were presented in the paper for the original C++ runtime. Then we compared our new runtime alongside the original C++ Selectors runtime as well as the UPC, SHMEM, and Conveyors versions of these applications. We used the CPU nodes of the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC) for our performance evaluations.

Overall, our benchmark results show that our Rust versions of the original applications performed on par with the respective C++ versions. There is a slight exception in the performance of our Random-Permutation implementation being significantly slower for a larger number of PEs. We include these benchmark results in the appendix. Nevertheless, we are confident that this can be addressed by further review and refinement of our Rust library code.

In addition to performance, we also measured the amount of `unsafe` code required from application developers, as a proxy for productivity.

Table 1. Percentage of expressions that were `unsafe` for each of the test applications. Calculated by $\%\text{unsafe} = \frac{\text{Number of unsafe expressions}}{\text{Total number of expressions}}$ using the `count-unsafe` utility from https://crates.io/crates/count-unsafe.

	Original	Refined
Histogram	70%	20%
Index Gather	29%	13%
Permute Matrix	45%	36%
Random Permutation	34%	11%

In Table 1, we measured the percentage of `unsafe` code that comprised each of the applications. The "Original" column depicts the percentages that we had with our initial application implementations. These implementations are a good proxy for the safety of the original C++ versions. After refining the API for safety, we were able to cut down the `unsafe` percentages of the applications to those in the "Refined" column.

7 Related Work

There are some relevant projects that provide a comparison point for this work in the runtime space. One is PNNL's Lamellar runtime [5], which builds on related Rust work for the Rust Open Fabrics Interface (ROFI). Lamellar utilizes

active messages to perform data movement and computation on an HPC compute cluster. Currently, Lamellar works in local mode and with transports supported by ROFI, namely SHMEM and some InfiniBand Verbs implementations.

Another inspiration for our work is the Rust implementation of Conveyors by Bill Carlson [3]. Conveyors focuses on support for message aggregation and efficient data movement, and the recent Rust implementation supports the concepts of sessions and value-based collectives on top of OpenSHMEM 1.4. While this port and Lamellar are both experimental in nature, their approaches provides key insights into how we can approach the development of a safe and efficient Rust-based runtime.

8 Conclusions and Future Work

The biggest potential improvement to our library is implementing HClib completely in Rust. Since we have not formally verified the safety of our Rust interfaces to HClib, implementing in Rust would provide us the safety guarantees that come with Rust. Furthermore, a purely Rust code base would significantly improve portability.

We would also like to reduce the amount of necessary `unsafe` code written by users, as doing so will greatly reduce the programmer's burden of memory safety and thereby improve the productivity of users. One significant remaining instance where a user would need to invoke `unsafe`, is accessing the selector state while the selector is running. This seems to be a common pattern in the Bale mini-applications, and it creates problems for designing a Rust interface because it appears to be concurrent mutation of data by both the selector's handler and the top-level finish closure. We believe it may be possible to use a lifetime token—similar to `FinishScope`—to denote permission to access, per the method laid out in the GhostCell paper by Yanovski *et al.* [13].

Acknowledgements

We would like to thank Bill Carlson for his guidance and suggestions related to creating the refined Rust interface discussed in this paper. We would also like to acknowledge Sriraj Paul for his help in understanding the C++ Actor implementation and his feedback on the paper.

This research used resources of the National Energy Research Scientific Computing Center (NERSC), a U.S. Department of Energy Office of Science User Facility located at Lawrence Berkeley National Laboratory, operated under Contract No. DE-AC02-05CH11231 using NERSC award ASCR-ERCAP24680.

A Benchmark Results

For performance evaluations, we use the CPU nodes of the Perlmutter supercomputer at the National Energy Research Scientific Computing Center (NERSC), which is an HPE (Hewlett Packard Enterprise) Cray EX supercomputer. Each node has 2 AMD EPYC 7763 (Milan) CPUs with 64 physical cores per CPU, 512 GB of DDR4 memory, and 1 network

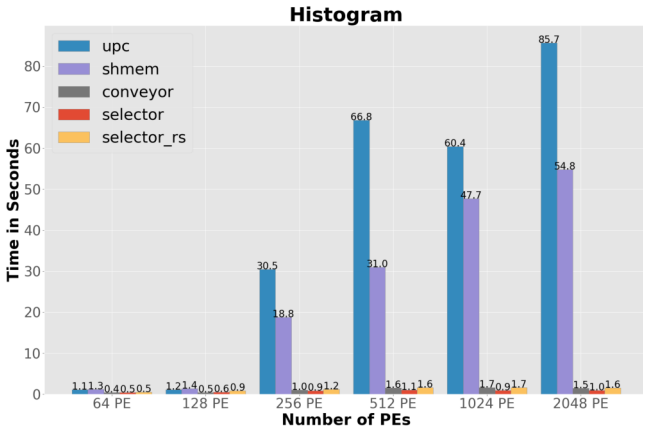

Figure 7. Histogram Kernel Results

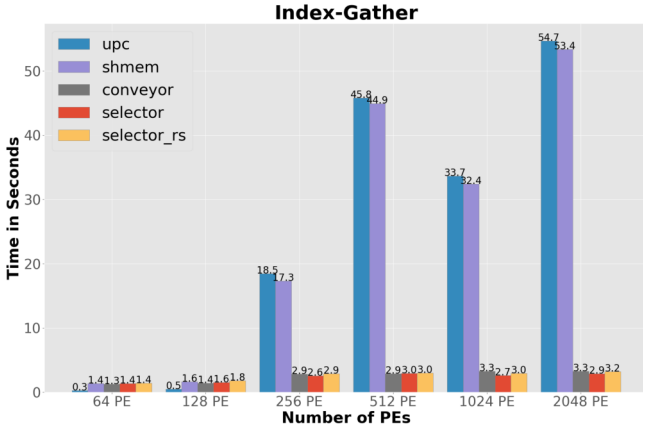

Figure 8. Index-Gather Kernel Results

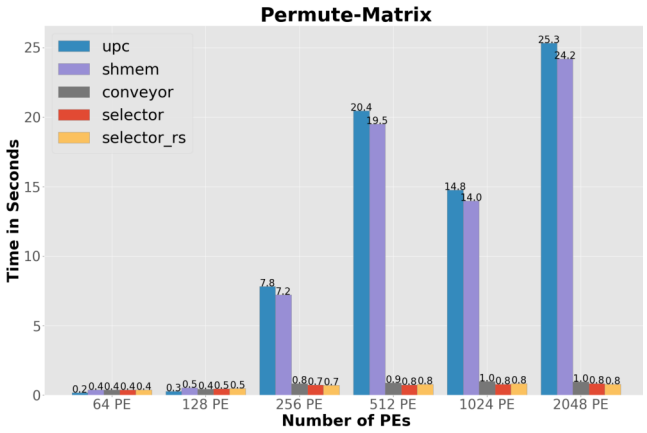

Figure 9. Permute-Matrix Kernel Results

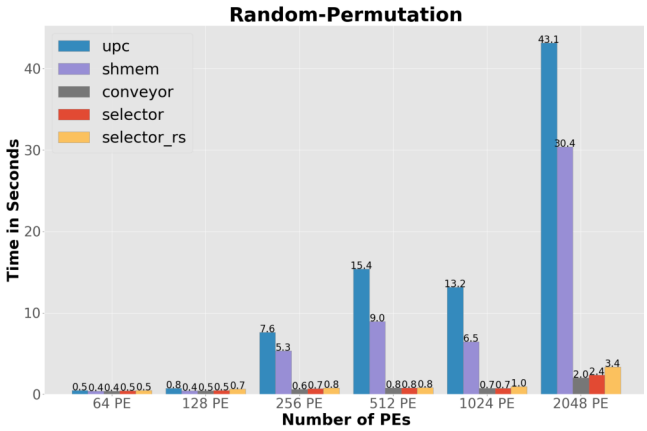

Figure 10. Random Permutation Kernel Results

card connected to the HPE Cray Slingshot 11 network. We fully utilize the 128 physical cores per node, each of which is mapped to a single PE (actor), and we use 1-16 nodes, resulting in 128-2048 physical cores. In addition to that, we evaluate 64 cores in a single node.

We investigate the performance of UPC, SHMEM, the original Conveyor baseline, and the original C++ actor version, all of which are available in HClib's public repository, in addition to the Rust actor version using four mini-applications from the Bale Kernels [4, 10] (histogram, index gather, permute matrix, random permutation):

1. **Histogram:** Using a generated list of random indices from each PE, the distributed table value at that index is incremented. It was run with 10,000,000 updates per PE on a distributed table with 1,000 elements/PE.
2. **Index Gather:** Executes an asynchronous read of random elements from a distributed array. It was run with 10,000,000 reads per PE on a distributed table with 100,000 elements/PE.
3. **Permute Matrix:** Creates a permuted distributed matrix by applying row and column permutations. It was run with 100,000 rows of the matrix/PE with an average of 10 nonzeros per row.
4. **Random Permutation:** Creates a distributed array holding a uniform permutation of $\{0, ..., N-1\}$. It was run with 1,000,000 elements per PE.

References

[1] 2019. Rust RFC: scoped_threads. https://rust-lang.github.io/rfcs/3151-scoped-threads.html. [Accessed 1-Sep-2023].

[2] Gul A. Agha. 1990. *ACTORS - a model of concurrent computation in distributed systems.* MIT Press.

[3] Bill Carlson. [n. d.]. GitHub - wwc559/convey: A conveyor implementation for Rust — github.com. https://github.com/wwc559/convey. [Accessed 30-Jun-2023].

[4] Jason Devinney. 2023. Bale Applications. https://github.com/jdevinney/bale.

[5] Ryan Friese. [n. d.]. GitHub - pnnl/lamellar-runtime: Lamellar is an asynchronous tasking runtime for HPC systems developed in RUST — github.com. https://github.com/pnnl/lamellar-runtime. [Accessed 30-Jun-2023].

[6] Max Grossman et al. 2017. A Pluggable Framework for Composable HPC Scheduling Libraries. IEEE Computer Society, 723–732. https://doi.org/10.1109/IPDPSW.2017.13

[7] Akihiro Hayashi, Sri Raj Paul, Youssef Elmougy, Jiawei Yang, and Vivek Sarkar. 2022. HClib Actor Documentation. https://hclib-actor.com/.

[8] Carl Hewitt et al. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence. Standford, CA, USA, August 20-23, 1973.* William Kaufmann, 235–245.

[9] Shams Mahmood Imam and Vivek Sarkar. 2014. Selectors: Actors with Multiple Guarded Mailboxes. ACM, 1–14. https://doi.org/10.1145/2687357.2687360

[10] F. Miller Maley and Jason G. DeVinney. 2019. Conveyors for Streaming Many-To-Many Communication. In *Workshop on Irregular Applications: Architectures and Algorithms, IA3 SC 2019.* IEEE, 1–8. https://doi.org/10.1109/IA349570.2019.00007

[11] Yuki Okushi. [n. d.]. GitHub - actix/actix: Actor framework for Rust. — github.com. https://github.com/actix/actix. [Accessed 30-Jun-2023].

[12] Sri Raj Paul, Akihiro Hayashi, Kun Chen, Youssef Elmougy, and Vivek Sarkar. 2023. A Fine-grained Asynchronous Bulk Synchronous parallelism model for PGAS applications. *Journal of Computational Science* 69 (2023), 102014. https://doi.org/10.1016/j.jocs.2023.102014

[13] Joshua Yanovski, Hoang-Hai Dang, Ralf Jung, and Derek Dreyer. 2021. Ghostcell: separating permissions from data in rust. *Proceedings of the ACM on Programming Languages* 5, ICFP (2021), 1–30.

Received 2023-06-29; accepted 2023-07-31