Lamellar: A Rust-based Asynchronous Tasking and PGAS Runtime for High Performance Computing

Ryan D. Friese
Pacific Northwest National Laboratory
Richland, Washington, USA
0000-0002-4121-2195

Roberto Gioiosa
Pacific Northwest National Laboratory
Richland, Washington, USA
0000-0001-9430-2656

Joseph Cottam
Pacific Northwest National Laboratory
Richland, Washington, USA
0000-0002-3097-5998

Erdal Mutlu
Pacific Northwest National Laboratory
Richland, Washington, USA
0000-0001-7472-3621

Gregory Roek
Pacific Northwest National Laboratory
Richland, Washington, USA
0000-0002-1889-8272

Polykarpos Thomadakis
Pacific Northwest National Laboratory
Richland, Washington, USA
0000-0002-4299-570X

Mark Raugas
Pacific Northwest National Laboratory
Richland, Washington, USA
0000-0002-5597-8781

***Abstract*—The discussion around "safe" programming languages has significantly increased in recent years, and is impacting how governments, industry, and academia plan to develop current and future software products. The White House Office of the National Cyber Director released a report [1] in February 2024 calling on the technical community to work towards proactively reducing attack surfaces in cyberspace, in part, specifically by adopting memory safe programming languages. While the main discourse thus far has been focused on cybersecurity, memory safety issues are also a concern in HPC, where memory related errors can result in wasted execution time, incorrect results, etc. Legacy programming languages in HPC such as C and C++ provide freedom and flexibility with memory management, but requires the developer to guarantee safety. While it is possible to develop "un-safe" code in all programming languages, "memory-safe" languages help guarantee safety by utilizing various compile time and runtime checks and validation systems.**

In this paper we introduce Lamellar, an asynchronous tasking and PGAS runtime system for HPC written in Rust, one such "memory-safe" language. We describe the entire Lamellar stack, from network interfaces to *safe* high-level abstractions such as distributed LamellarArrays and Active Messages. The goal of our runtime is to enable end-users to develop entirely safe Rust code in their applications, limiting the use of any "unsafe" code blocks to rigorously tested code blocks within the runtime itself. We conclude by showing comparable performance against several C, C++, and Chapel implementations of a subset of the BALE kernel suite while maintaining strong memory safety principles.

***Index Terms*—HPC, PGAS, Asynchronous, Runtime, Distributed Computing, Rust Programming Language**

I. Introduction

This work presents Lamellar, an HPC runtime system written in the Rust programming language. First and foremost, Lamellar is a tool for safe, productive development. It provides a performance-competitive alternative to C and C++ legacy runtimes, with a special focus on PGAS and asynchronous tasking approaches. Lamellar itself is open-sourced and available on both Github [2] and Crates.io [3].

The unique feature of Lamellar is a tiered programming abstraction that interfaces with Rust's native memory safeguards. Like the abstractions present in the Rust language itself, these features do not guarantee memory or thread safety in and of themselves. However, by transferring the responsibility of memory safety for large portions of the software development pipeline to the Rust compiler, they can reduce exposure to many of the unsafe behaviors endemic to HPC. They can also reduce total development time – sometimes dramatically. A main goal of Lamellar is to enable users to develop performant HPC applications using only *safe* APIs, thus limiting the opportunity for memory related errors to introduced by the users.

Memory related errors permeate every branch of computation. The United States National Security Agency (NSA) recently released a memo [4] on the "memory safety problem," highlighting the role of memory-related bugs in modern cyber vulnerabilities. Over the last decade, for example, Microsoft and Google state that up to 70% of the vulnerabilities in their products may have stemmed from memory issues.

The NSA memo recommends adoption of memory safe languages wherever possible, specifically including Rust. Rust [5] is a systems programming language designed to provide the speed and flexibility of a language like C++ without large classes of memory or thread safety issues that can lead to system instability, error, or remote exploitation by an attacker. Rust's goal is the generation of safe, efficient, and reliable code.

Lamellar occupies a new niche in the safe computing ecosystem generally and in Rust in particular. Specifically, it provides high-level support for HPC networks and pro-

979-8-3503-5554-3/24/$31.00 ©2024 IEEE
DOI 10.1109/SCW63240.2024.00165

gramming models (including PGAS) that are needed to fully leverage the memory and thread safety features available in Rust while making little compromises on the performance. While cybersecurity has not historically been a significant concern in the HPC community where tightly controlled systems are the norm, memory related errors are still a significant issue, potentially resulting in wasted execution time, incorrect results, and data corruption. We build on the OpenFabrics interface [6](OFI) to construct a lightweight C-library (Rofi) and a set of Rust bindings (Rofi-sys) to access high performance network fabrics within Rust applications. These libraries form the bottom layer of our Lamellar Software Stack (Fig. 1), and are available as stand alone libraries [7], [8].

We describe the Lamellar runtime design, architecture, and select APIs in Sec. III, provide experimental results for Lamellar communication abstractions and a comparison to Chapel and C and C++ approaches utilizing OpenSHMEM on a subset of the Bale kernel suite in Sec. IV, and discuss prospects for future work in Sec. V.

II. Related Work

MPI [9]/OpenMP [10] are the *de facto* standards for high-performance parallel programming models and have been dominant in HPC application development. MPI uses explicit inter-processor data exchange. OpenMP is a cross-language standard for shared memory programming within each node. While they don't implement the PGAS programming model, the combination embodies concepts used in PGAS systems.

SHMEM [11]is a CRAY-developed library for distributed computing. It includes one-sided, point-to-point and collective communications, shared memory views, and atomic operation on global variables. These capabilities use a distributed communication frameworks (such as MPI) and provide an interface for higher-level PGAS systems. There are SHMEM implementations for many different platforms. OpenSHMEM [12] includes a standard PGAS model.

Foundational work in PGAS models includes Aggregate Remote Memory Copy Interface (ARMCI) [13] and Communications Run-time for Extreme Scale (ComEx) [14]. This early work developed concepts such as remote memory access (RMA), inter-connect specific support, multi-threading, group-aware communications and generic active messages. Global Arrays (GA) [15] built on this early work and provides a clear implementation of the PGAS model.

GASNet [16] is a language-independent communication library to support PGAS systems through a high performance, network-independent communication interface. GASNet is widely used in open-source and vendor-supported libraries for PGAS programming models.

Chapel [17] is a language-based PGAS programming model that is developed by Cray aiming to fill the programmability gaps between mainstream and parallel programming languages. It makes use of GASNet for communication and data management in the runtime. It features general parallel programming concepts, global-view abstraction and various controls over the operations depending on the locality of the data.

UPC (Unified Parallel C) [18] is an extension to the C language where users define shared arrays and pointers that address the global memory space. In addition to providing language-level global access, it features a `forall` loop that can distribute iterations based on the affinity of the array elements. Later, a C++ extension named UPC++ [19] provided an object-oriented PGAS programming model with parallel programming concepts to support a wider range of asynchronous operations.

Charm++ [20] is an object-oriented portable parallel programming language based on C++. It provides explicit programming constructs to provide a separation between sequential and parallel objects with operations executed in a message driven manner.

HPX [21] provides a task-based PGAS programming model that focuses on defining the overall operation in a set of tasks. Using tasks, HPX tries to avoid the complexities of explicit data distribution and communication found in MPI. This task-based approach supports dynamic communication between nodes and dynamic data migration and autonomous load balancing.

Exstack, Exstack2 [22], and Conveyors [23] are all C-based aggregation libraries built on top of either OpenSHMEM or UPC. Exstack performs synchronous aggregation (resembling a bulk synchronous programming model). Exstack2 is an asynchronous version of Exstack. Conveyors implements a multi-hop aggregation approach to reduce memory footprint and increase bandwidth utilization. We compare these three libraries with Lamellar in Section IV.

The Habanero C/C++ library (HClib) [24] is an asynchronous many-task (AMT) programming model-based runtime used to implement the"Selectors" API for a fine-grained Asynchronous Bulk-Synchronous programming model [25]. Within this library, point-to-point remote operations are represented as fine-grained asynchronous actor messages, which abstracts the complexities of message aggregation and termination detection from the user. We compare the Lamellar with implementations using the Selector library in Section IV.

Within the Rust community we have found two other efforts targeting HPC. This first is a Rust implementation of the above mentioned Conveyors library. We were unable to successfully build Rust Conveyors on our system, and thus will not include it in our experimental analysis.

The second is Selectors-rs [26], which is a re-implementation of the HClib Selectors API. This is to say, the library does not simply create a Rust wrapper directly on top of the C++ Selectors API, rather it creates Rust wrappers for lower level dependencies so as HClib and re-implements the upper layer API. We were unable to find publicly available source of Selectors-rs and thus was unable to perform a comparison.

Outside of HPC two of the most popular async Rust runtimes are Tokio and Rayon. Tokio [27] caters to I/O applications including web servers and clients. It provides

an asynchronous version of the Rust standard library with a multi-threaded executor, and features a large ecosystem of users, developers, and associated libraries. Rayon [28] is the *de facto* standard for data parallel processing in Rust. It features multi-threaded, work-stealing execution engines and provides its own fork/join asynchronous execution model (independent of Rust futures). Parallelism in Rayon is typically exposed through a Parallel Iterator, and the the library itself provides parallel implementations of many sequential iterator methods. Tokio and Rayon are complimentary: Rayon offers no functionality for distributed execution or I/O bound applications, and Tokio offers little advantage for CPU-bound applications.

These packages belong to a growing Rust ecosystem which may be useful to HPC. Other tool sets include `async_std`, which offers well-vetted asynchronous counterparts to many components of the Rust standard library, `smol`, which re-exports several other packages in a small asynchronous runtime with concise, simplified trait definitions, and `fuchsia-async`, which provides an executor for use in the Fuchsia operating system.

III. Runtime Design

Throughout the following sub-sections we will discuss each layer in the runtime (Fig. 1), its responsibility, and any interfaces it provides to the user. Nomenclature used by the runtime includes:

- Node - A physical compute node.
- Processing Element (PE) - The smallest logical entity for executing Lamellar applications. PEs in Lamellar utilize asynchronous thread pools. Multiple PEs can be allocated to a compute node but one PE per Numa node is typical.
- World - All the PEs instantiated for an application. The number of PEs is controlled through the system's launcher (e.g. slurm).
- Team - A subset of PEs in the world; sub-teams are supported.
- Remote Direct Memory Access (RDMA) Memory Region - A segment of memory accessible from remote nodes.
- Active Message (AM) - A message which runs code when it arrives at a remote PE.
- One-sided - Operations that only requires the calling PE (no coordination with remote PEs occurs).

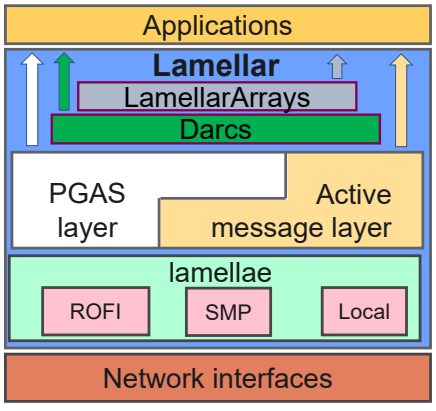

Fig. 1: The Lamellar stack, highlighting various layers and interfaces provided by the runtime.

- Collective - Operations that require all PEs in a team to coordinate before proceeding.

A. Lamellae Layer

At the base of the stack is the abstraction for communicating with network interfaces, called the `Lamellae` Trait. In Rust, a `Trait` is a collection of methods that are available to multiple different types. This is similar to an "interface" in other languages (e.g. Go and Java). Typically we say that a type `T` "implements" a Trait `U` if it can execute the methods in `U`.

The `Lamellae` Trait is the interface between the runtime and network interfaces via functions for: (de)initialization; getting PE ids and the number of PEs in the world; and (de)allocating Memory Regions. The Trait defines the functions for performing remote `put`/`get` transfers, and synchronization primitives (e.g., `barrier`). The rest of this section provides details on three implementors of the `Lamellae` Trait.

1) ROFI & ROFI-sys: ROFI is a light-weight transport layer built on OpenFabrics (OFI) [6] meant to facilitate data transfers in distributed environments. Its goal is to provide high-level abstractions compared to OFI, but not to implement sophisticated communication protocols, which are implemented at higher levels of the software stack. ROFI is designed to integrate with Rust code, minimizing issues that can occur using the Rust Foreign Function Interface (FFI) and utilizes popular packages that assist with Rust-C integration such as Bindgen [29]. It supports synchronous and asynchronous RDMA-based APIs (i.e., PUT and GET) treating messages as a sequence of bytes, without interpreting their content. It provides an API for collective synchronization (i.e. Barrier) and mechanisms to (de)allocate RDMA Memory regions. Future work will replace this C-library with a Rust binding of libfabric directly.

ROFI-sys is a low-level Rust crate exposing the public interface of ROFI (i.e., the methods in ROFI.h). These are implemented as *unsafe* Rust functions that can be called from any Rust program. Every function provided by ROFI-sys must be declared as *unsafe*, because the Rust compiler cannot guarantee the behavior and safety of libraries written in other languages. It is infeasible to expect mature libraries and operating systems to be rewritten fully in Rust, thus the standard compromise is to minimize user exposure to memory and thread related bugs by introducing higher-level *safe* abstractions over these "-sys" packages at the application layer. By marking these functions as *unsafe* we are explicitly warning users that extra care is needed when utilizing them.

Essentially there are two types of data transfer that occur within a Lamellar application. The first is a simple RDMA `put` or `get` to directly transfer a raw bitstream. The second uses runtime (de)serialization of messages between PEs. Building upon the ROFI C library and ROFI-sys crate described above we developed the `ROFI_Lamellae` to enable distributed execution of Lamellar applications. If we oversimplify ROFI(-sys) as a mechanism to transfer bytes of

data from one PE to another PE, then `ROFI_Lamellae` is responsible for all other aspects of transferring data from one PE to another.

For RDMA, the ROFI Lamellae simply provides a lightweight shim over the corresponding calls in ROFI(-sys). These types of transfers are (generally) *unsafe* as the Rust compiler is unable to reason about memory access on remote PEs. RDMA transfers used internally by the runtime are exposed as *unsafe* interfaces that can be called on Lamellar`MemoryRegions` (Sec. III-D) and `UnsafeArrays` (Sec. III-F). These transfers also are the building blocks for other *safe* abstractions.

For cases where messages need to be (de)serialized between PEs, the Lamellae implements a "flag" based transfer mechanism. Each PE is able to signal every other PE to let them know when data is to be read. Upon receiving this signal the Remote PE is then responsible for getting the data, once local buffers become available. The remote PE then signals the original PE to let it know it is now free to release any resources associated the the transferred data. Lamellar employs a double buffering message queue to both deal with asynchrony and allow for more efficient use of network resources by transferring larger messages.

The Lamellae is also responsible for managing RDMA Memory Regions used within an application. During initialization of a Lamellar application, a large RDMA Memory Region is allocated on each PE. A portion of this region is reserved for internal runtime use, including message buffers and the tables described above; this portion remains static during the lifetime of an application, and scales in size with the number of PEs. The remainder of the RDMA Memory Region is used as a one-sided dynamic heap for user-level data structures such as AMs and One-sided memory regions (discussed in Secs. III-C and III-D2 respectively).

The Lamellae also manage the allocation of RDMA Memory Regions for distributed data structures exposed to the user, such as the distributed arrays described in Sec. III-F. The construction of such objects are collective but only block the calling thread on each PE, other threads are free to continue working. Unlike the runtime RDMA memory regions, these memory regions will be deallocated once the associated data structure has been `dropped` (destructed) from the application.

2) Shmem: To allow development of Lamellar applications in a non-distributed environment and without the need for third party language dependencies (i.e. libfabric and ROFI), we have developed a POSIX shared-memory based implementation of the `Lamellae` Trait. While shared memory provides potential optimizations such as directly transferring data objects through shared-memory segments without (de)serialization, we chose to emulate the behavior of the ROFI Lamellae as much as possible. This eases development and debugging, and ensures a seamless transition when switching between the two Lamellae. In particular, it avoids tailoring design decisions to optimized Shared Memory Lamellae in ways that fail to scale to the distributed environment. Scaling beyond a single node always presents challenges, but the goal is to minimize risk.

The shared memory Lamellae implements all the same internal data structures as the ROFI Lamellae. The key difference is that instead of creating RDMA Memory Regions (via libfabrics) it simply allocates shared memory segments that all PEs on the local node can access. From a user perspective switching between the ROFI Lamellae and the Shared Memory Lamellae should be transparent.

3) SMP: The SMP Lamellae targets single-process multi-threaded applications where there is only one PE. No data transfer needs to occur, so there is no (de)serialization. This Lamellae has no RDMA Memory Regions (and related concepts), as all allocations requested by the application are handled locally (e.g., by the standard Rust allocator).

While we have tried to ensure that applications first written using only the SMP Lamellae will execute successfully on both the Shmem and ROFI Lamellaes, some care must be taken to ensure any distributed synchronization calls are correctly matched. Given that it is currently hard to reason about these calls at compile time, we perform some limited runtime analysis to warn users of theses issues. Additionally, our documentation explicitly highlights which calls are collective calls and which are one-sidied

B. Thread Pool Layer

Above the Lamellae layer in the Lamellar stack is the Thread Pool layer, which is responsible for managing all the worker threads in a PE (including enqueuing and executing asynchronous tasks). Lamellar fully supports Rust `Futures` and the **`async`**/**`await`** programming model; as such, Lamellar thread pools are considered Rust `Executors` [30]. Generally, the thread pool manages asynchronous tasks generated by various runtime APIs, including both the execution of AMs and the communication tasks produced by the `Lamellae` (which handle data transfers). Lamellar enables users to submit their own `Futures` for execution on the thread pool; this enables compatibility with popular third party Rust packages such as Async-std [31] and a subset of the Tokio Framework [27].

The Lamellar thread pool utilizes a work-stealing implementation with respect to individual PEs. This provides a general-purpose baseline approach. Future work will explore other implementations, such as the ability to utilize compute node topologies to minimize performance degradation when running across multiple NUMA domains, work-stealing across PEs and an API to enable user-defined custom thread pool implementations.

C. Active Message Layer

Active Messages (AMs) are a computing model where messages contain both data (that you want to compute something with) and metadata that indicates how to process this data when it arrives at its destination, e.g. a function pointer. Lamellar is built upon asynchronous AMs. The runtime allows for sending and executing user defined AMs on remote PEs in distributed environments.

AMs in Lamellar are exposed to users via a Rust Trait called LamellarAM and a powerful Rust language feature called a Procedural Macro. At a high level, Rust Procedural Macros allow you to execute code at *compile* time to provide custom syntax that is converted into valid Rust code. Using Procedural Macros, it is possible to create Domain Specific Languages (DSLs) and then have them be transformed into Rust code at compile time [32].

The "Hello World" application in Listing 1 covers the use and semantics of Lamellar AMs. Line 2 uses a "prelude module" to import the data structures and Traits required to construct an AM-based Lamellar application. Line 4 defines a **struct** for the AMs input data. This example shows a **struct** with a single member, but more complicated data structures are possible as long as they adhere to restrictions set by the AMs Trait Bounds. Trait bounds for AMs include (de)serialization [33], safe referencing from multiple threads (Sync [34]), and safety to send from one thread to another (Send [34]) (see the Lamellar documentation [3] for more details). Line 3 registers the **struct** with the runtime using the *#[AmData]* procedural macro, which parses the **struct** definition at compile time and attempts to automatically implement the required Traits (if this fails, a compile-time error is produced). Using a procedural macro reduces repetitious work that can often be automatically produced and is the expected practice in the Rust community [35].

Lines 7-12 define the computation that will be performed on the data in the AM. Line 7 is a procedural macro; it ingests the AM implementation (Lines 8-12) and generates the code needed to perform serialization, deserialization, execution, and the preparation of return data. The macro also assigns each AM to a unique identifier which is registered in a runtime lookup table, enabling AMs to properly deserialize and execute on remote PEs. Line 8 shows how to implement a Trait for a given type in Rust, in this case the LamellarAM Trait for the HelloWorldAM type. LamellarAM defines a single function, **async fn** exec(self), that must be implemented. The keyword **async** signals that AMs are asynchronous tasks. For further details on Asynchronous Rust see [36], [37].

Next we examine the body of the AM and see that we want to print something. println! is a Rust provided macro similar to printf in C. In this example we print the PE id where the AM is executing, and the value contained in the AMs name member. The value for the PE id is provided by Lamellar::current_pe. Other commonly used functions include: Lamellar::num_pes, which returns the total PEs in the world; Lamellar::world, a reference to the World; and Lamellar::team, a reference to the Team that launched the AM.

Though not shown in this example, both Lamellar::world and Lamellar::team can be used to launch new AMs from within a currently executing AM. This allows users to easily construct AM dependency chains and use recursive design patterns. It should be noted that this example does not return any data from the AM, but Lamellar supports returning both "normal" data (essentially anything that can be serialized and deserialized) and AMs. Details on these capabilities are available in the documentation [3].

Finally, Lines 13-24 show the main function for the Hello World example. Line 14 initializes the runtime environment, using a "builder" pattern to configure Lamellar at runtime. The build() initializes the underlying Lamellae and thread pools, and returns a LamellarWorld instance. Line 15 creates an instance of our HelloWorldAM. Line 16 launches the AM, directing the AM to be sent to, and executed on, every PE. This call produces a Rust Future, which allows us to await the result of this specific request, as shown in Line 17 (if our AM returned data, it would be returned here). Note that block_on only blocks the calling PE, while world.barrier() is a global synchronization point. Lines 19 - 23 show how to send AMs to one (not all) PEs, and illustrates the wait_all() function (Line 22) which blocks the calling PE until all of the AMs it launched have completed.

Calls to exec_am_*(...) will place the supplied AM into the thread pool, where it will either execute locally or be serialized, stored in a message buffer, and the transferred to a remote PE. Upon arrival at the remote PE, the communication task will create an asynchronous task to deserialize, execute and return results from each AM in the buffer.

In this example is there is no explicit finalize function call; rather we utilize Rust lifetimes and scoping rules. Specifically, the world variable is automatically dropped in Line 27 at the end of the main function, which in turn executes the Lamellar deinitialization process. Each PE remains active until all other PEs are ready to deinitialize. In this example, PE0 exits its main function before every other PE, but because it is still alive, its thread pool is still able to process AMs sent to it by other PEs.

```
1  // import necessary data stuctures and traits
2  use _______::active_messaging::prelude::*;
3  #[AmData] //this is an "attribute-like" procedural
   ↪ macro
4  struct HelloWorldAM{
5      name: String,
6  }
7  #[am] //this is also an "attribute-like" procedural
   ↪ macro
8  impl _______AM for HelloWorldAM
9    async fn exec(self) {
10     println!("PE{}:
       ↪ hello{}!",_______::current_pe,self.name);
11   }
12 }
13 fn main(){
14   let world = _______WorldBuilder::new().build();
15   let am = HelloWorldAM{ name: String::from("World") };
16   let request = world.exec_am_all(am); //all PEs to all
     ↪ PEss
17   world.block_on(request); //only blocks local PE
18   world.barrier(); //global sync
19   if world.my_pe() != 0 {
20     let am = ExampleAm{ name: String::from("World2") }
21     world.exec_am_pe(0,am); //send to PE0
22     world.wait_all();//only blocks local PE
23   }
24 }
```

Listing 1: Hello World in Lamellar

D. PGAS Layer

The Lamellar runtime uses RDMA operations to transfer AMs from PE to PE. Because these operations are inherently unsafe, Lamellar controls user access with two levels of PGAS abstraction: low (unsafe) and high (safe). High-level abstractions are designed specifically for end users, and provide a range of robust safety and productivity features. Low-level abstractions are designed for internal use by the runtime itself. They provide fewer safeguards, and their use by end users is discouraged. Where the need for fine-grained control makes low-level abstractions essential, users must confine their use to code regions fenced by the `unsafe` keyword. Minimizing the footprint of these fenced regions can substantially reduce risk of exposure (as well as code complexity), even when use of low-level abstractions are unavoidable[1]. This section describes low-level abstractions; high-level abstractions are described in Sec. III-F.

1) Shared Memory Regions: A `SharedMemoryRegion` is essentially a small wrapper around an RDMA Memory Region collectively allocated to the PEs of a team. The allocation occurs directly from the underlying network fabric (not the runtime-maintained heap). Although creating a new `SharedMemoryRegion` is a collective blocking call it only blocks the calling thread (typically main), allowing the thread pool to execute other tasks. `SharedMemoryRegions` will allocate the same-size RDMA Memory Region on every PE on which they are created.

A `SharedMemoryRegion` object provides methods to read and write to the corresponding memory regions on Remote PEs in the form of `fn put(dest_pe, index, src_buf)` and `fn get(src_pe, index, dst_buf)` (full API signatures and descriptions are documented in [3]). We provide both non-blocking versions of `put/get`, which rely on the user to determine when a transfer has completed, and blocking versions which use runtime provided transfer detection. A `SharedMemoryRegion` also provides direct local access to the underlying data as a Rust `slice` or `ptr`. As with the rest of the `SharedMemoryRegion` API, this is *unsafe* as there are no protections against remote PEs writing to local data while you hold a non-mutable reference.

`SharedMemoryRegions` are specialized types of distributed atomically reference counted objects (Darcs), which allows them to be passed along as a member of an AM. General purpose Darcs are introduced in Sec. III-E, but the key attribute is that they enable the runtime to track lifetimes of distributed objects in a way that guarantees an object is alive on all PEs it was created on as long as any PE still holds a reference to it (one can think it as a C++ `std::shared_ptr` in a distributed fashion).

2) OneSided Memory Regions: The `OneSidedMemoryRegion` is similar to the `SharedMemoryRegion`, except instead of requiring a blocking collective call across PEs, only the calling PE is involved in the allocation of the RDMA memory segment. This operation tends to be efficient, as the runtime can often allocate the memory directly from its internal RDMA memory heap. `OneSidedMemoryRegions` also expose blocking and non-blocking `put/get` APIs, with the key difference being that no remote PE is supplied as an argument. The `put/get` will always refer to to original constructing PE. `OneSidedMemoryRegions` are also specialized Darcs, so PEs can send them in AMs.

[1]The division of code into `safe` vs. `unsafe` blocks is a core paradigm in Rust programming [38]. It reflects a philosophical approach to computing that emphasizes explicit demarcation, and integrates natively with other design features, e.g. informative compiler errors.

E. Distributed Atomic Reference Counting (Darcs) Layer

As indicated in the PGAS Sec. III-D, Lamellar introduces a new smart pointer type called a `Darc`, or Distributed Atomically Reference Counted pointer. Darcs are a distributed extension to Rust language-provided `Arcs` (Atomically Reference Counted smart pointers). The `Arc` API is a cornerstone of safe concurrent (i.e. multi-threaded and asynchronous) programming in Rust, and allows for safe *shared ownership* of objects [39]. `Darcs` provide similar abstractions within a *distributed* environment.

The `Darc` API is a *safe* distributed computation abstraction exposed to the user, and used internally by the runtime. Specifically, `Darcs` have global lifetime tracking and management, meaning that the pointed-to objects remain valid and accessible as long as any PE maintains a reference to it. Inner mutability of the object pointed to by the Darc is disallowed by default. Similar to `Arcs`, if you need to mutate through a `Darc` you can use `Mutex`, `RwLocks`, or objects that implement the `Sync` Trait (such as `Atomic` types).

`Darcs` are passed via AMs and allow distributed access-to- and manipulation-of- Rust objects. Allocating a `Darc` is a collective call amongst all PEs on a team. Each PE must pass in an instance of the pointed-to type, as shown in the following (simplified) API: `fn Darc::new<T>(team, item: T)`. The data structure for the `Darc` itself is located in an RDMA Memory Region, which allows it to be sent in AMs. The inner object can exist on the Rust heap or in an RDMA memory region. When a `Darc` is created, each PE will maintain its own *independent* instance of the inner object. There does not exist a single shared instance across PEs; instead the `Darc` provides a mechanism for accessing a remote PE's instance.

Reference counting occurs as normal during `Clone` [40] , serialization and deserialization is used to track the transfer of `Darcs` to remote PEs in AMs. Destruction of a `Darc` is asynchronous and occurs once every PE agrees that no further references to the object exist. This is tracked in status bits in the RDMA memory regions and an AM does the actual deallocation.

F. LamellarArray Layer

Building upon all the layers below it, we end with the LamellarArrays layer, which provides a *safe* PGAS abstraction of distributed arrays. LamellarArrays utilize both `Darcs` and `SharedMemoryRegions`, so constructing an array is a blocking and collective operation with all PEs on a team.

While SharedMemoryRegions explicitly require users to calculate a PE-specific offset, LamellarArrays use 0-based indexing, with offsets calculated automatically by the runtime. LamellarArrays provide a range of features: multiple array types (depending on the safety guarantees required), *safe* RDMA put/get interfaces, element-wise operations, Block or Cyclic data layouts, iteration methods, reductions, and the ability to create sub arrays. We discuss some of these features below.

1) Lamellar Array Types: Lamellar supports four main array types varying by data-access based safety guarantees:

1) UnsafeArray - No Safety guarantees; PEs are free to read/write anywhere in the array with no access control. Similar to Memory Regions, UnsafeArrays are intended for internal use, but are exposed to users and marked *unsafe*.
2) ReadOnlyArray - No write access is permitted. PEs are free to read from anywhere in the array with no access control.
3) AtomicArray - Access to each element is an atomic (either intrinsically or enforced via the runtime), represented as the two sub types below:
   a) NativeAtomicArray - Elements are Rust atomic types, such as AtomicUsize, AtomicI8, etc.
   b) GenericAtomicArray - Elements are protected by a 1-byte Mutex.
4) LocalLockArray - The entire data region on each PE is protected by a single locally constructed RwLock.

LamellarArray instances can be converted between types, either via the Rust Into/From Traits, or by Lamellar APIs such as array.into_atomic(), array.into_read_only(), etc. Conversion is a collective call involving all PEs that originally constructed the array. It is also a blocking call that only succeeds when there is precisely one reference to the array on each PE, specifically, the reference which is executing the conversion call. These conditions guarantee that the underlying data is only ever pointed-to by one array type at any time, ensuring that the safety guarantees of each type are honored [2].

2) RDMA like operations: LamellarArrays provide RDMA like put/get operations similar to Shared/OneSidedMemoryRegions *but* each array obeys the safety guarantee corresponding to its type. Generally this means, the safe array types utilize AMs to emulate the behavior of direct RDMA operations, so all access to a remote PEs data is actually managed on that PE rather than by the PE initiating the access. UnsafeArrays provide *unsafe* APIs allowing for direct RDMA between PEs, as well as the AM based methods, but again we recommend users to avoid this array type, and instead use a safe variant. Due to the non-mutable access guarantee provided by ReadOnlyArrays we are also able to provide a direct RDMA get operation, as we know the underlying data cannot change (put does not exist for ReadOnlyArrays).

3) Element-wise Operations: LamellarArrays provide a number of operations for individual elements in the array, including arithmetic (+,-,*,/,%), bit-wise operators (&,|,^,/), shift left and shift right operators, compare_and_exchange operators, and load, load, swap operators. Operators apply either as single-element requests, or using a batch API that aggregates multiple operations in to a single request. Single element operators take the form array.op(index, val), for example array.add(5, 100) adds 100 to the element at index 5. Batch operators have several forms: *Many Indices - One value*, *One Index - Many values*, and *Many Indices - Many values (one-to-one)*. For example, array.batch_store([20, 2], 10) sets the elements at indices 20 and 2 to 10, whereas array.batch_mul(20, [2, 10]) first multiplies the element at index 20 by 2, then multiplies the result by 10, and in the call array.batch_bit_or([0, 105, 67], [127, 0, 64]) the element at index 0 performs a bit-wise Or with 127, the element at index 105 performs one with 0, and the element at index 64 performs one with 64. Lamellar also exposes fetch variants that return the current value before applying the operation.

The runtime calculates the correct PEs and offsets for each array index, batching operations by destination PE within a single message. Batched messages can only contain operations of the same type. Generally, operations are executed in the same order they are launched, but the runtime makes no ordering guarantees. Each operation returns a Future that can either be awaited for completion, or, for fetch operations, used to retrieve a result.

4) Iteration: LamellarArrays offer three powerful iterators: DistributedIterator, LocalIterator, and OneSidedIterator.

DistributedIterator enables distributed and parallel iteration over elements. It is blocking and collective over all PEs that contain the array's data. Generally, the runtime tries to have PEs only iterate over their own data, but will automatically manage data transfer from remote PEs as needed. DistributedIterators support the methods collect, enumerate, filter, filter_map, for_each, map, skip, step_by, take.

LocalIterator is a one-sided iterator which enables parallel iteration over the calling PEs local data. From the perspective of the iterator it is completely unaware that it exists within a distributed context, therefore it will never transfer data from a remote PE. LocalIterator supports collect, chunks, enumerate, filter, filter_map, for_each, map, skip, step_by, take, zip.

Both DistributedIterator and LocalIterator are asynchronous and return a Future after calling for_each or collect. Thus users must **await** this future to ensure the iteration has completed.

Lastly, OneSidedIterator enables serial iteration over all the elements of the array on the calling PE. This iterator operates over the entire array, thus data transfer from remote PEs must occur, but it is managed by the runtime. OneSidedIterator implements chunks, skip, step_by,

[2]Conversion can induce deadlock due to Rust's scoping rules (the compiler does *not* do deadlock detection)

`zip` to reduce data movement, but otherwise can be used with any iterator methods supported by the Rust standard library (by calling `into_iter()` to convert the `OneSidedIterator` into a normal Rust `Iterator`).

We provide a fully executable example showing how we use an AtomicArray to implement the Histogram benchmark discussed in Sec. IV-B1. Of particular note to the Lamellar arrays abstractions, are creating a new array in line 8. We specify the array type, tell the compiler the element type (`usize`), associated the array with a team (in this case the world), provide a global length, and specify the distributed data layout. Line 15 performs a `batch_add` operation, accepting a list of random indices (lines 9-12) and indicating we want to add 1 to the element at each index. Although not discussed above, we show in line 18 applying a `sum` reduction on our array. We use this sum to ensure no updates were missed (line 19).

```
1  use _______::array::prelude::*;
2  use rand::Rng;
3  use std::time::Instant;
4  const T_LEN: usize = 1_000_000; //global len
5  const L_UPDATES: usize = 10_000_000; //updates per pe
6  fn main() {
7    let world = _______WorldBuilder::new().build();
8    let table = AtomicArray::<usize>::new(&world, T_LEN,
     ↪  Distribution::Block);
9    let mut rng = rand::thread_rng();
10   let rnd_i = (0..L_UPDATES) //generate random indices
11     .map(|_| rng.gen_range(0, T_LEN))
12     .collect::<Vec<_>>();
13   world.barrier();
14   let timer = Instant::now();
15   world.block_on(table.batch_add(rnd_i, 1));
     ↪  //histogram kernel
16   world.barrier();
17   println!("Elapsed time: {:?}", timer.elapsed());
18   let sum = world.block_on(table.sum());
19   assert_eq!(sum, L_UPDATES * world.num_pes());
20 }
```

Listing 2: Example AtomicArray Histogram implementation

IV. Benchmarks

This section examines Lamellar performance. Sec. IV-A discusses bandwidth curves for *put*- like operations and Sec. IV-B compares Lamellar to existing C and C++ approaches built on top of OpenSHMEM and Chapel based implementations of a subset of the BALE Kernel suite [41].

All results were gathered using a 48 node local cluster. During testing only 32 nodes were available. Each node contains dual socket AMD EPYC 7543 32 Core CPUs with 8 Numa nodes per CPU (64 cores and 16 Numa nodes total per compute node). There is 256 GB of DDR4-3200 memory per compute node. The network consists of Mellanox HDR-100 ConnectX-6 InfiniBand HBA cards, with a peak bandwidth of 100Gb/s (12.5GB/s), organized as a full fat tree. The cluster is comprised of 4 racks, 12 nodes each. Each rack has a Mellanox Spectrum®-3® SN4600 64-port 100GbE leaf switch. 24 ports are used to connect the 12 nodes within a rack to the switch. Each leaf switch is connected to three spine switches with 8 connections to create a fat-tree. The spines are Mellanox Spectrum® SN2700 32-port 100GbE switches.

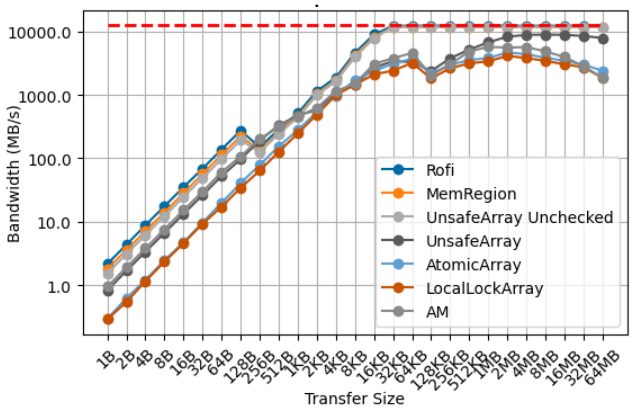

Fig. 2: *Put*-like bandwidth curves (higher is better).

A. Bandwidth

As detailed in Secs. III-D and III-C there are numerous communication primitives Lamellar can use to transfer data. Our first set of results compares the potential performance of each method on *put* like operations. Performance (MB/s) was measured while conducting N transfers of various sizes. For transfer sizes in the range 1B - 4KB we perform 262143 individual transfers. The remaining transfer sizes send $\frac{1GB}{transfer\ size}$ individual transfers. Results are presented as bandwidth curves. The tests are performed on two PEs, located on different compute nodes, each PE occupied a single CPU.

Fig. 2 shows these results. The theoretical max bandwidth of our network is shown by the red dotted-line. *Put* operations take data stored on PE0 and transfer it to PE1. The Rofi(libfabric) performance presented is essentially an upper-bound on Lamellar's performance. `MemRegion` is the lowest-level Lamellar call (*unsafe*); i.e. light-wrappers around the Rofi(libfabric) calls. For the distributed arrays, data is stored in a `OneSidedMemoryRegion` on PE0 and transferred into the array on PE1. The AM-based implementation contains a `Vec<u8>`, and the `exec` function returns immediately (on PE1).

The Rofi(libfabric), `MemRegion`, and "unchecked" `UnsafeArray` all perform manual termination detection, e.g. checking for a known pattern and entering a `barrier` (which PE0 will be blocked in). The remaining methods rely on the runtime to provide termination detection, i.e. calling `wait_all()`.

These results show that safety comes with a performance cost. Reducing this gap is a focus of future work. Rofi(libfabric), `MemRegion`, and "unchecked" `UnsafeArray` have similar performance, with Rofi(libfabric) and `MemRegion` performing slightly better than "unchecked" `UnsafeArray`. The difference between these cases can be attributed to Lamellar runtime overhead. All three approaches perform near the maximum of the network for transfer sizes 32KB and larger. The drop in performance between 128B and 256B corresponds to a threshold in the libfabrics verbs

provider to switch between `fi_inject_write` (which is optimized for small messages) and `fi_write`.

The LamellarArray and AM based approaches are similiar to each other up to 100KB, though they exhibit a runtime cost vs. Rofi(libfabric) and MemRegion approaches. The array approaches copy data into a `Vec`, which is then transferred to PE1. When it arrives, the array type store the data according to its safety guarantees. `UnsafeArray` does a memcopy. `LocalLockArray` first grabs the local RwLock, and then performs a memcopy. Finally, `AtomicArray` iterates through the the elements of the `Vec` and performs an atomic store to the corresponding array element. The overhead of these safety guarantees results in worse performance when comparing against the `UnsafeArray`, which itself performs slightly worse than the AM approach due to the additional memcopy.

For messages larger than 100k, there is an initial dip in performance because the runtime performs aggregation for messages sizes smaller than 100K (this threshold is configurable; 100KB is the default, with this test indicating 512KB - 1MB are more appropriate for our system). Fig. 2 shows `UnsafeArray` starts to perform significantly better than the other array types and the AM approach around the 100KB threshold as well. This is for two reasons, first the cost of copying data into the AM `Vec` becomes non-trival at larger message sizes. Second, `UnsafeArray` uses the same aggregation threshold to switch transfer methods from the `Vec` AMs to using an RDMA `get` from the target (PE1). (It does not use RDMA `put` from PE0 into PE1 because termination detection can be difficult without explicit coordination from PE1, we hope to resolve through future work on Rofi). [3]

B. Bale Kernel Suite

Bale is a suite of kernels designed as "... a vehicle for discussion for parallel programming productivity" [41]. Bale kernels are an appropriate benchmark because Bale's design goals include demonstrating patterns for (1) irregular distributed parallel applications (2) with internode communication (3) to make it easier to write/maintain/achieve top performance. These goals align well with Lamellar's.

We compare to five other Bale aggregation implementations. Three C implementations are evaluated: Exstack, which is synchronous and resembles Bulk Synchronous Programming, Exstack2 which is an asynchronous version of Exstack, Conveyors [23], [41] which implements a multi-hop aggregation approach to reduce memory footprint and increases bandwidth utilization. Conveyors-rs is a Rust implementation of conveyors, which we were unable to successfully build on our system, and thus excluded. One C++ implementation is evaluated: Selectors [25] which is built upon HClib an asynchronous many-task (AMT) programming model-based runtime. We were unable to find the source for the Rust implementation of Selectors in [26], but based on the results there, its performance was comparable to the C++ version [25]). The last is

[3]Lamellar `get` transfers follow the same trends as `put` and are omitted for brevity.

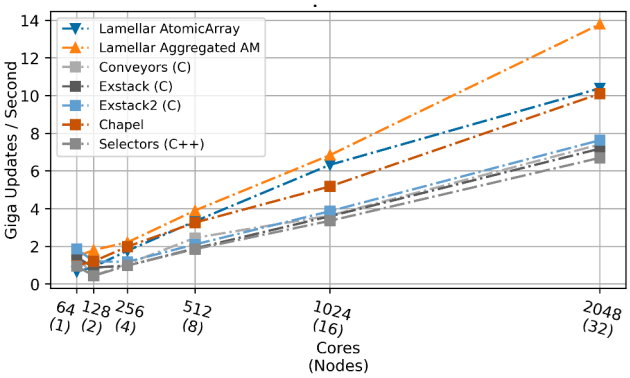

Fig. 3: Histogram kernel performance (higher is better).

implemented using the automatic aggregation capabilities of Chapel. We compare Lamellar to these five implementations using the Histogram, IndexGather, and Randperm kernels.

For all experiments, Exstack, Exstack2, Conveyors, and Selectors were built using OpenSHMEM v5.0.0rc2, with an OpenSHMEM PE allocated per core, for 64 PEs per node. Lamellar and Chapel were tested in several configurations of PEs per node. In all cases presented, Lamellar implementations performed best with 16 PEs per node (1 PE per NUMA node) utilizing 4 threads per PE (1 thread per core). Chapel implementations performed best with a varying number of Locales per node at each scale (within the range of 1 - 8 Locales per node) The average results of 10 runs is presented for each approach.

1) Histogram: Histogram is a simple kernel similar to the GUPS benchmark. Each PE generates N indices uniformly at random from the range of a distributed array. It then increments the table's value at that index. Although the kernel is simple it represents a common communication pattern (small message all-to-all) in many parallel applications.

Fig. 3 presents benchmark results. The first Lamellar version [4] uses AMs to manually aggregate indices (into a `Vec`) by destination PE. This method iterates over the random indices in parallel, each thread maintains its own update buffers (in an effort to match the PE per core behavior of the OpenSHMEM based approaches). When executing on a remote PE, the AM iterates through the `Vec` of indices and atomically updates the corresponding entries in the distributed table.

The second Lamellar approach utilizes the `batch_add` API on an `AtomicArray`. A reduced (but executable) example of our implementation is shown in Listing 2. The actual Histogram kernel is shown in line 15. Experiments used a distributed table with 1,000 elements per core, performed 10,000,000 updates per core, and limited aggregations to 10,000 operations per buffer. For `AtomicArray`, the runtime automatically splits `batch_add` into sub-batches of up to 10,000 elements.

[4]Lamellar full Histogram code is available at [42], [43].

Fig. 3 shows that all seven approaches have similar scaling properties, with the Lamellar AM approach performing best as scale increases, followed by the Lamellar AtomicArray approach. The AM approach includes many hand-optimized components utilizing only *safe* abstractions and represent opportunities we may be able to apply to the runtime. In comparison, the `AtomicArray` also only uses *safe* API, but places all responsibility of the computation on the runtime, including creating sub-batches (by batch size and destination PE), multi-threaded batch dispatch, and creation and execution of the internal AMs. The runtime tries do this as efficiently as possible, but introduces overhead related to resource allocation, batching, and execution of the array operations (that the manually aggregated AM approach can avoid). These inefficiencies appear to scale with the number of PEs, and highlights an area of focus for future work.

Overall, the Lamellar based approaches efficiently aggregate the small messages and achieve good scaling performance, using safe abstractions. Although manual aggregation performs better, the `AtomicArray` approach is simpler to implement and still achieves respectable performance.

2) IndexGather: IndexGather is similar to Histogram, however it reads random elements from a table (instead of writing to random elements). IndexGather is more difficult to execute efficiently since the runtime needs to both (1) manage the initial remote read requests and (2) return the results of those reads. Psuedocode for IndexGather is:

```
for (i, rand_i) in random_indices.enumerate(){
    target[i] = table[rand_i];
}
```

`target` is a local array that stores the remote read results.

The Lamellar IndexGather implementations [5] are similar to the Histogram kernel, adding a `ReadOnlyArray` implementation in place of an `AtomicArray`. The core line for `ReadOnlyArray` is shown below:

```
target =
↪  world.block_on(table.batch_load(rnd_idxs));
```

Calls to `batch_load` will return a `Vec` containing the values at the indices specified in `rnd_idxs`.

Experiments used a distributed table with 1,000 elements per core, performed 10,000,000 requests per core, and limited aggregations to 10,000 operations per buffer. Index gather results are shown in Fig. 4, Generally, it is not surprising that IndexGather performance is less than Histogram as the operation includes a return value (a 2nd message) in addition to the index value that is transferred in Histogram.) Chapel achieves highest performance as internally this implementation uses a specialized `CopyAggregator`, which is optimized for simple assignment operations and allocates additional buffers for each PE to communicate with one another using RDMA. The performance curves of the Lamellar based implementations have reversed, with the `ReadOnlyArray` performing better at larger scales that the Active Message approach. In part this is because the runtime based aggregation is better able to balance both sending and receiving data simultaneously.

[5]Lamellar IndexGather is available at [44].

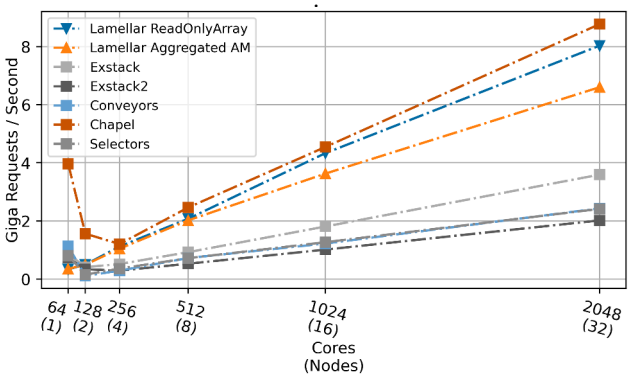

Fig. 4: IG kernel performance (higher is better).

3) Randperm: Randperm creates, in parallel, a distributed array of size N that holds a random permutation of $0..N-1$. Although there are communication-free ways to implement this kernel [45], Bale employs the "dart throwing algorithm" [46]. Each PE manages a contiguous slice of the array, whose indices are called "darts." There is a distributed target array at least as large as N (fewer messages are needed as target table size goes up). Each PE "throws" its darts to random locations in target array. A dart that hits an empty location sticks, and its value is recorded at that location in the target. If the location is already occupied, the dart must be thrown again until it sticks. Once all darts have stuck, the target array iterates to collect darts in the order they appear, forming a size-N random permutation.

We present four implementations, highlighting the flexibility of AMs, `Darcs`, and Lamellar`Arrays` in distributed kernels. The first ("Array Darts") uses `AtomicArrays` for storage; it throws darts with `batch_compare_exchange`, and moves results to the final permutation with the `Collect` iterator. The second ("AM Dart") uses AMs to manually aggregate (1) darts by destination PE, and (2) throw results. The two remaining implementations may not maintain the "uniform at random" property in all cases, but they do highlight the ability to quickly iterate designs for different constraints. The third ("AM Dart Opt") removes some communication from the second: when a dart encounters a occupied slot, it will randomly select a new location on the current PE (unless all locations on this PE are filled). The final implementation ("AM Push") first randomizes the darts slice on each PE (locally), then randomly selects another PE for each dart. When the dart arrives, it is pushed to the end of the Target vector on that PE, a dart throw never fails, so communication is minimized. All AM approaches and use `Darcs` for remote access to the target arrays. [6] For all implementations, the desired array will have 1,000,000 elements per core, with the target array being twice that size. Fig. 5 presents the results. Ideally, the execution time would remain constant regardless of the number of cores used (as the work per core remains the same). While

[6]Lamellar Randperm code is available at [47].

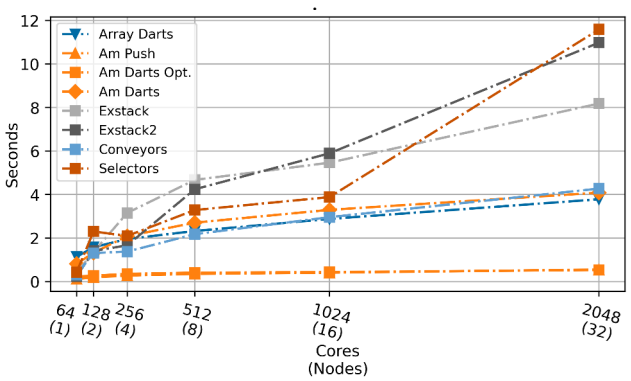

Fig. 5: Randperm kernel running time (lower is better).

this is not true across all cores for any implementation, the Lamellar approaches and Conveyors exhibit more consistent behavior. The OpenSHMEM implementations exhibit good single node performance, but exhibit a noticeable performance penalty at 2048 cores. This could stem from our network topology (two racks for 1024 cores, versus four racks for 2048 cores). Unsurprisingly, the two Lamellar approaches that minimize communication (Am Darts Opt and Am Push) perform best, while the other two Lamellar approaches exhibit similar communication patterns and performance.

V. Conclusions and Future Work

The increasing discussion around memory-safe programming languages underscores a significant shift in the industry towards prioritizing security, reliability, and developer productivity, making them essential tools for current and future software development. This work introduces Lamellar, an asynchronous task and PGAS runtime targeting HPC which has been written in Rust, one such "memory safe" language. Lamellar supports single process, single node shared memory SMP, and distributed shared memory environments. Furthermore, Lamellar does this using Rust-provided memory and async/await task management tools, enabling both performance and safety through techniques that are common in the Rust community.

The Lamellar runtime exposes diverse abstractions for remote communication and computation. The recommended *safe* PGAS interface is provided by Lamellar`Array`, with different implementations providing different access characteristics and safety guarantees. All arrays provide access and iterator patterns familiar to Rust programmers and useful for distributed programming, as well as optimized routines for common distributed-memory operations. The runtime also provides low-level (but *unsafe*) PGAS APIs via `Shared` and `OneSidedMemoryRegions`. Finally, Lamellar supports asynchronous tasking via its Active Message interface, utilizing Rust Proc-Macros to allow users to provide their own implementations. We showed that raw performance is accessible via Lamellar, and acceptable performance is often convenient. Across all examined kernels, we find Lamellar is expressive enough to quickly enable numerous *safe* approaches when developing distributed applications.

Future work on Lamellar includes (1) improving automatic aggregation in the runtime to improve performance, (2) expanding array iterator types to improve parity with standard Rust arrays and (3) reducing the overhead of array types to make truely high-performance more accessible. We plan to work with domain scientists to further test the viability of Lamellar in specific domains. Industry has already started to embrace memory safe languages such as Rust, and we believe there is a use case for them in HPC as well.

VI. Acknowledgments

We thank the reviewers for their helpful feedback. This work was partially funded under the High Performance Data Analytics (HPDA) program at the Department of Energy's Pacific Northwest National Laboratory. The Pacific Northwest National Laboratory is operated by Battelle for the U.S. Department of Energy under contract DE-AC05-76RL01830.

References

[1] The white house office of the national cyber director, national cybersecurity strategy report. [Online]. Available: https://www.whitehouse.gov/wp-content/uploads/2023/03/National-Cybersecurity-Strategy-2023.pdf

[2] R. D. Friese, R. Gioiosa, J. Cottam, E. Mutlu, G. Henselman-Petrusek, and P. Thomadakis. (2024) Lamellar: rust hpc runtime. [Online]. Available: https://github.com/pnnl/lamellar-runtime

[3] ——. (2024) Lamellar. [Online]. Available: https://docs.rs/lamellar/latest/lamellar

[4] *NSA — Software Memory Safety*, Nov 2022. [Online]. Available: https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

[5] N. D. Matsakis and F. S. Klock, "The rust language," *Ada Lett.*, vol. 34, no. 3, p. 103–104, oct 2014. [Online]. Available: https://doi.org/10.1145/2692956.2663188

[6] P. Grun, S. Hefty, S. Sur, D. Goodell, R. D. Russell, H. Pritchard, and J. M. Squyres, "A brief introduction to the openfabrics interfaces - a new network api for maximizing high performance application efficiency," in *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*, 2015, pp. 34–39.

[7] R. Gioiosa and R. D. Friese. (2024) Rust openfabrics interface transport layer (rofi). [Online]. Available: https://github.com/pnnl/rofi

[8] R. D. Friese and R. Gioiosa. (2024) rofi-sys. [Online]. Available: https://github.com/pnnl/rofi-sys

[9] M. Snir, S. Otto, and S. Huss-Lederman, *MPI–the Complete Reference: the MPI core*. MIT press, 1998, vol. 1.

[10] L. Dagum and R. Menon, "Openmp: an industry standard api for shared-memory programming," *IEEE Computational Science and Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[11] K. Feind, "Shared memory access (shmem) routines," *Cray Research*, vol. 53, 1995.

[12] B. Chapman, T. Curtis, S. Pophale, S. Poole, J. Kuehn, C. Koelbel, and L. Smith, "Introducing openshmem: Shmem for the pgas community," in *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, 2010, pp. 1–3.

[13] J. Nieplocha and B. Carpenter, "Armci: A portable remote memory copy library for distributed array libraries and compiler run-time systems," in *Parallel and Distributed Processing: 11th IPPS/SPDP'99 Workshops Held in Conjunction with the 13th International Parallel Processing Symposium and 10th Symposium on Parallel and Distributed Processing San Juan, Puerto Rico, USA, April 12–16, 1999 Proceedings 13*. Springer, 1999, pp. 533–546.

[14] J. Daily, A. Vishnu, H. van Dam, B. Palmer, and D. J. Kerbyson, "On the suitability of mpi as a pgas runtime," in *International Conference on High Performance Computing (HiPC)*, 2014.

[15] J. Nieplocha, B. Palmer, V. Tipparaju, M. Krishnan, H. Trease, and E. Aprà, "Advances, applications and performance of the global arrays shared memory programming toolkit," *The International Journal of High Performance Computing Applications*, vol. 20, no. 2, pp. 203–231, 2006.

[16] D. Bonachea and J. Jeong, "Gasnet: A portable high-performance communication layer for global address-space languages," *CS258 Parallel Computer Architecture Project, Spring*, vol. 31, p. 17, 2002.

[17] B. L. Chamberlain, D. Callahan, and H. P. Zima, "Parallel programmability and the chapel language," *The International Journal of High Performance Computing Applications*, vol. 21, no. 3, pp. 291–312, 2007.

[18] U. Consortium *et al.*, "Upc language specifications v1. 2," 2023.

[19] Y. Zheng, A. Kamil, M. B. Driscoll, H. Shan, and K. Yelick, "Upc++: a pgas extension for c++," in *2014 IEEE 28th international parallel and distributed processing symposium*. IEEE, 2014, pp. 1105–1114.

[20] L. V. Kale and S. Krishnan, "Charm++ a portable concurrent object oriented system based on c++," in *Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications*, 1993, pp. 91–108.

[21] H. Kaiser, T. Heller, B. Adelstein-Lelbach, A. Serio, and D. Fey, "Hpx: A task based programming model in a global address space," in *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, ser. PGAS '14. New York, NY, USA: Association for Computing Machinery, 2014. [Online]. Available: https://doi.org/10.1145/2676870.2676883

[22] J. G. DeVinney. Bale: Kernels for irregular parallel computation. [Online]. Available: https://github.com/jdevinney/bale/blob/master/docs/Bale-StGirons-Final.pdf

[23] F. M. Maley and J. G. DeVinney, "Conveyors for streaming many-to-many communication," in *2019 IEEE/ACM 9th Workshop on Irregular Applications: Architectures and Algorithms (IA3)*, 2019, pp. 1–8.

[24] M. Grossman, V. Kumar, N. Vrvilo, Z. Budimlic, and V. Sarkar, "A pluggable framework for composable hpc scheduling libraries," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2017, pp. 723–732.

[25] S. R. Paul, A. Hayashi, K. Chen, Y. Elmougy, and V. Sarkar, "A fine-grained asynchronous bulk synchronous parallelism model for pgas applications," *Journal of Computational Science*, vol. 69, p. 102014, 2023. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877750323000741

[26] J. Parrish, N. Wren, T. H. Kiang, A. Hayashi, J. Young, and V. Sarkar, "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*, ser. MPLR 2023. New York, NY, USA: Association for Computing Machinery, 2023, p. 165–172. [Online]. Available: https://doi.org/10.1145/3617651.3622992

[27] (2024) Tokio. [Online]. Available: https://github.com/tokio-rs/tokio

[28] (2024) Rayon. [Online]. Available: https://github.com/rayon-rs/rayon

[29] (2024) Bindgen. [Online]. Available: https://github.com/rust-lang/rust-bindgen

[30] (2023) Applied: Build an executor. [Online]. Available: https://rust-lang.github.io/async-book/02_execution/04_executor.html

[31] (2024) Async-std. [Online]. Available: https://github.com/async-rs/async-std

[32] The Rust Project Developers, *The Rust programming language*. No Starch Press, 2023.

[33] (2023) Trait serde::deserialize. [Online]. Available: https://docs.rs/serde/latest/serde/trait.Deserialize.html

[34] (2023) Send and sync. [Online]. Available: https://doc.rust-lang.org/nomicon/send-and-sync.html

[35] (2023) Derive. [Online]. Available: https://doc.rust-lang.org/rust-by-example/trait/derive.html

[36] (2023) Asynchronous programming in rust. [Online]. Available: https://rust-lang.github.io/async-book/01_getting_started/01_chapter.html

[37] (2023) Async in depth. [Online]. Available: https://tokio.rs/tokio/tutorial/async

[38] (2023) The rustomicon. [Online]. Available: https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html

[39] (2024) Atomically reference counted. [Online]. Available: https://doc.rust-lang.org/std/sync/struct.Arc.html

[40] (2023) Clone. [Online]. Available: https://doc.rust-lang.org/std/clone/trait.Clone.html

[41] (2023) Bale kernel suite. [Online]. Available: https://github.com/jdevinney/bale

[42] R. D. Friese, R. Gioiosa, J. Cottam, E. Mutlu, and G. Henselman-Petrusek. (2024) Lamellar: atomic array histogram benchmark. [Online]. Available: https://github.com/pnnl/lamellar-benchmarks/blob/april_2023/histo/src/histo_buffered_safe_am.rs

[43] ——. (2024) Lamellar: atomic array histogram benchmark. [Online]. Available: https://github.com/pnnl/lamellar-benchmarks/blob/april_2023/histo/src/histo_lamellar_atomicarray.rs

[44] ——. (2024) Lamellar: indexgather benchmark. [Online]. Available: https://github.com/pnnl/lamellar-benchmarks/tree/april_2023/index_gather

[45] P. Sanders, S. Lamm, L. Hübschle-Schneider, E. Schrade, and C. Dachsbacher, "Efficient parallel random sampling—vectorized, cache-efficient, and online," *ACM Trans. Math. Softw.*, vol. 44, no. 3, jan 2018. [Online]. Available: https://doi.org/10.1145/3157734

[46] P. B. Gibbons, Y. Matias, and V. Ramachandran, "Efficient low-contention parallel algorithms," *Journal of Computer and System Sciences*, vol. 53, no. 3, pp. 417–442, 1996. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0022000096900793

[47] R. D. Friese, R. Gioiosa, J. Cottam, E. Mutlu, and G. Henselman-Petrusek. (2024) Lamellar: randperm benchmark. [Online]. Available: https://github.com/pnnl/lamellar-benchmarks/tree/april_2023/randperm

Appendix: Artifact Description/Artifact Evaluation

Artifact Description (AD)

I. Overview of Contributions and Artifacts

A. Paper's Main Contributions

- C_1 Introduction Lamellar, a new asynchronous active message and PGAS runtime for HPC written in the Rust programming language.
- C_2 In-depth overview of the various layers of the runtime.
- C_3 Performance comparison against various other runtimes.

B. Computational Artifacts

- A_1 https://github.com/pnnl/lamellar-runtime
- A_2 https://github.com/pnnl/rofi
- A_3 https://github.com/pnnl/rofi-sys
- A_4 https://github.com/pnnl/lamellar-benchmarks
- A_5 https://github.com/jdevinney/bale/tree/master/src/bale_classic
- A_6 https://github.com/chapel-lang/chapel/tree/main/test/studies/bale
- A_7 https://github.com/singhalshubh/hclib/tree/bale_actor

Artifact ID	Contributions Supported	Related Paper Elements
A_1, A_2, A_3	C_1, C_2, C_3	Listings 1-2 Figures 1-5
A_4, A_5 A_6, A_7	C_3	Figures 2-5

II. Artifact Identification

A. Computational Artifact A_1

Relation To Contributions

This is the main Lamellar Runtime Repository, providing the various abstractions detailed in the paper. It also contains the application implementations for performing the bandwidth tests.

Expected Results

It is expected that safe abstractions exposed by the runtime perform at least as well as legacy "unsafe" runtimes.

Expected Reproduction Time (in Minutes)

The expected time to build the artifact is 15-25 minutes depending on hardware (e.g. storage performance).

The expected execution time of the bandwidth tests is 2-3 minutes per test, with 7 tests executed.

Artifact Setup (incl. Inputs)

Hardware: All results were gathered using a 48 node cluster at PNNL. During testing only 32 nodes were available. Each node contains a dual socket with AMD EPYC 7543 32 Core CPUs with 8 Numa nodes per CPU (64 cores and 16 Numa nodes total per compute node). There is 256 GB of DDR4-3200 memory per compute node. The network is constructed from Mellanox HDR-100 ConnectX-6 InfiniBand HBA cards, with a peak bandwidth of 100Gb/s (12.5GB/s), organized as a full fat tree. The cluster is comprised of 4 racks, 12 nodes each. Each rack has a Mellanox Spectrum®-3® SN4600 64-port 100GbE leaf switch. 24 ports are used to connect the 12 nodes within a rack to the switch. Each leaf switch is connected to three spine switches with 8 connections to create a fat-tree. The spines are Mellanox Spectrum® SN2700 32-port 100GbE switches.

Software: Rust Programming Lanugage v 1.78.0 – https://www.rust-lang.org/ All other dependencies managed and contained in the lamellar "Cargo.toml" file – https://github.com/pnnl/lamellar-runtime/blob/master/Cargo.toml

Datasets / Inputs: N/A

Installation and Deployment: Rust Programming Lanugage v 1.78.0 – https://www.rust-lang.org/ It should be possible to simply clone the repository and then run the following command:

- cargo build –release –features enable-rofi -j4 –examples

Artifact Execution

All presented Lamellar application results rely on this artifact. This artifact in turn relies of A2 and A3 to execute within a distributed environment.

The implementation of all the bandwidth tests require no parameters to be provided and can be executed as provided. For transfer sizes in the range 1B - 4KB we perform 262143 individual transfers. The remaining transfer sizes send $1GB/transfer\_size$ individual messages. Presented bandwidth numbers are the average of all messages for a given transfer size.

Artifact Analysis (incl. Outputs)

Output from the bandwidth applications are average performance for each transfer size, these are injested by a python script and plotted.

B. Computational Artifact A_2

Relation To Contributions

This is the ROFI C-library repository, it is lightweight transport layer between libfabrics and the higher levels of the lamellar runtime.

Expected Results

Generally, ROFI provides the same performance as achieved by the libfabric providers. Lamellar performance is bounded by ROFI performance

Expected Reproduction Time (in Minutes)

The expected time to build the artifact is less than 5 minutes.

Artifact Setup (incl. Inputs)

Hardware: Same as A1

Software: Tested with various versions of both CLANG and GCC.

Libfabric v1.21 – https://ofiwg.github.io/libfabric/

Datasets / Inputs: N/A

Installation and Deployment: This should not be installed manually, instead letting the Lamellar crate handle installation as described in artifact A1.

Artifact Execution

All presented Lamellar application results rely on this artifact. This Artifact is a direct dependency of A3

Artifact Analysis (incl. Outputs)

see A1.

C. Computational Artifact A_3

Relation To Contributions

This is the rofi-sys repository, it is a crate providing Rust bindings to the Rofi C-Library.

Expected Results

Generally, rofi-sys should not introduce any performances regressions and should perform the same as ROFI.

Expected Reproduction Time (in Minutes)

The expected time to build the artifact is less than 5 minutes.

Artifact Setup (incl. Inputs)

Hardware: Same as A1

Software: Rust Programming Lanugage v 1.78.0 – https://www.rust-lang.org/

All other dependencies contained in the crate Cargo.toml file.

Datasets / Inputs: N/A

Installation and Deployment: This should not be installed manually, instead letting the Lamellar crate handle installation as described in artifact A1.

Artifact Execution

All presented Lamellar application results rely on this artifact. This Artifact is a direct dependency of A1.

Artifact Analysis (incl. Outputs)

See A1.

D. Computational Artifact A_4

Relation To Contributions

This repository contains our implementations of the presented benchmarks.

Expected Results

Generally, we expect our implementations to perform better than those in A5 and A7, and be competitive with those in A6.

Expected Reproduction Time (in Minutes)

The expected time to build the artifact is 10 minutes.

Expected time to run a complete set of experiments for each algorithm implementation is less than 1 hour. There are 2 implementations for the Histo and IndexGather kernels, and 4 implementations of the Randperm kernel. Many of these tests can be run in parallel (when utilizing less than 32 nodes).

Artifact Setup (incl. Inputs)

Hardware: Same as A1.

Software: Rust Programming Lanugage v 1.78.0 – https://www.rust-lang.org/

All other dependencies automatically managed and listed in each applications Cargo.toml file.

Datasets / Inputs: All inputs generated as part of the tests.

Installation and Deployment: Install by cloning the repository and executing:

- cargo build –release

Artifact Execution

All presented results depend on A1,A2,A3.

Each benchmark accepts input parameters for specifying the problem size and the number of times to run the test. For Histogram and Indexgather:

- 1000 elements in distributed table per core.
- 10,000,000 operations performed per core.
- aggregation limited to 10000 operations.
- Each test executed 10 times

For Randperm

- 1,000,000 elements per core to be permuted.
- 2,000,000 elements per core in ”target” array.
- Each test executed 10 times

Additional parameters we experiment with are the number of Processes per node, controlled through the clusters job management system. The number of total threads per node remained constant regardless of the number of processes per node, and was equal to the number of cores. For example a single process per node would launch 64 threads, while 8 processes per node would only launch 8 nodes.

Artifact Analysis (incl. Outputs)

Each run of an implementation prints the average value over however many runs were executed. The output of each execution is stored into a file, with the path describing the number of nodes used, the number of processes per node, and the implementation used. A python script parses an ingests these files to plot the final results.

E. Computational Artifact A_5

Relation To Contributions

This repository contains the original implementations of the BALE suite implemented in C utilizing OpenSHMEM.

Expected Results

Generally, we expect these to perform below the Lamellar implementations

Expected Reproduction Time (in Minutes)

The expected time to build the artifact is 10 minutes.

Expected time to run a complete set of experiments for each algorithm implementation is less than 1 hour. There are three implementations for each application.

Artifact Setup (incl. Inputs)

Hardware: Same as A1.

Software: Tested with various versions of GCC. Tested with OpenSHMEM v5.0.0rc2

Datasets / Inputs: All inputs generated as part of the tests.

Installation and Deployment: Followed the directions provided in the README.

Artifact Execution

Each benchmark accepts input parameters for specifying the problem size and the number of times to run the test. For Histogram and Indexgather:

- 1000 elements in distributed table per core.
- 10,000,000 operations performed per core.
- aggregation limited to 10000 operations.
- Each test executed 10 times

For Randperm

- 1,000,000 elements per core to be permuted.
- 2,000,000 elements per core in "target" array.
- Each test executed 10 times

OpenSHMEM operates with a single Process per core.

Artifact Analysis (incl. Outputs)

Each run of an implementation prints the average value over however many runs were executed. The output of each execution is stored into a file, with the path describing the number of nodes used, the number of processes per node, and the implementation used. A python script parses an ingests these files to plot the final results.

F. Computational Artifact A_6

Relation To Contributions

This repository is from the Chapel programming language, and provides implementations of the tested bale kernels.

Expected Results

Generally, we expect these implementations to be as good as Lamellar.

Expected Reproduction Time (in Minutes)

The expected time to build the artifact is 10 minutes.

Expected time to run a complete set of experiments for each algorithm implementation is less than 1 hour. There is one implementation for each benchmark.

Artifact Setup (incl. Inputs)

Hardware: Same as A1.

Software: Chapel V2.0

Datasets / Inputs: All inputs generated as part of the tests.

Installation and Deployment: Followed the directions provided in the README.

Artifact Execution

Each benchmark accepts input parameters for specifying the problem size and the number of times to run the test. For Histogram and Indexgather:

- 1000 elements in distributed table per core.
- 10,000,000 operations performed per core.
- aggregation limited to 10000 operations.
- Each test executed 10 times

Additional parameters we experiment with are the number of Processes per node, controlled through the clusters job management system. The number of total threads per node remained constant regardless of the number of processes per node, and was equal to the number of cores. For example a single process per node would launch 64 threads, while 8 processes per node would only launch 8 nodes.

Artifact Analysis (incl. Outputs)

Each run of an implementation prints the average value over however many runs were executed. The output of each execution is stored into a file, with the path describing the number of nodes used, the number of processes per node, and the implementation used. A python script parses an ingests these files to plot the final results.

G. Computational Artifact A_7

Relation To Contributions

This repository is a fork of HCLib and contains C++ "Selectors" implementations of the examined kernels.

Expected Results

Generally, we expect these implementations to be on par with the OpenSHMEM implementations.

Expected Reproduction Time (in Minutes)

The expected time to build the artifact is 10 minutes.

Expected time to run a complete set of experiments for each algorithm implementation is less than 1 hour. There is one implementation for each benchmark.

Artifact Setup (incl. Inputs)

Hardware: Same as A1.

Software: automake modern versions of gcc or clang

Datasets / Inputs: All inputs generated as part of the tests.

Installation and Deployment: Followed the directions provided in the README.

Artifact Execution

Each benchmark accepts input parameters for specifying the problem size and the number of times to run the test. For Histogram and Indexgather:

- 1000 elements in distributed table per core.
- 10,000,000 operations performed per core.
- aggregation limited to 10000 operations.
- Each test executed 10 times

For Randperm

- 1,000,000 elements per core to be permuted.
- 2,000,000 elements per core in ”target” array.
- Each test executed 10 times

Selectors execute with a single process per core.

Artifact Analysis (incl. Outputs)

Each run of an implementation prints the average value over however many runs were executed. The output of each execution is stored into a file, with the path describing the number of nodes used, the number of processes per node, and the implementation used. A python script parses an ingests these files to plot the final results.