**Lamellar：基于Rust的异步任务与PGAS运行时系统，面向高性能计算**

**摘要**

近年来，关于“安全”编程语言的讨论显著增加，并正在影响政府、工业界和学术界如何规划当前和未来软件的开发。美国白宫国家网络总监办公室在2024年2月发布的报告中呼吁技术社区通过采用内存安全编程语言来主动减少网络空间的攻击面。尽管目前主要讨论集中在网络安全，但内存安全问题在高性能计算（HPC）中同样值得关注，内存相关错误可能导致执行时间浪费、结果错误等。HPC中传统使用的C/C++等语言在内存管理上赋予开发者自由和灵活性，但需要开发者去保证安全性。虽然所有编程语言都可能编写“不安全”的代码，但“内存安全”语言可以通过编译时和运行时的检查与验证系统帮助保障安全性。

本文介绍了Lamellar——一种基于Rust，一种“内存安全”的语言，开发的异步任务与PGAS运行时系统，适用于HPC。我们描述了Lamellar的完整技术栈，从网络接口到**安全**的高层抽象，如分布式LamellarArray和Active Messages。该运行时的目标是让最终用户能够在应用程序中使用完全安全的Rust代码，令“不安全”代码块被运行时内部严格测试的模块阻挡。最后，我们展示了Lamellar在BALE内核套件子集上与多种C、C++和Chapel实现的性能对比，同时保持严格的内存安全原则。

**关键词**：HPC、PGAS、异步、运行时、分布式计算、Rust编程语言

**I. 引言**

本文提出了Lamellar——一种用Rust编写的HPC运行时系统。其首要目标是实现安全、高效的开发，为传统C/C++运行时提供性能竞争的替代方案，重点关注PGAS和异步任务模型。Lamellar本身已在GitHub和Crates.io开源。

Lamellar的核心特性是**分层编程抽象**，与Rust原生内存安全机制相结合。如同Rust语言自身的抽象，这些特性本身不保证内存或线程安全，但通过将大部分软件开发流程中的内存安全责任转移给Rust编译器，能够减少HPC中常见的不安全行为，并显著缩短开发时间。Lamellar的主要目标是让用户仅通过**安全API**开发高性能HPC应用，从而限制用户引入内存相关错误的机会。

内存相关错误渗透计算的各个领域。美国国家安全局（NSA）近期发布的备忘录强调了内存相关漏洞在现代网络攻击中的角色。例如，微软和谷歌表示其产品中高达70%的漏洞可能源于内存问题。

NSA备忘录建议尽可能采用内存安全语言，特别是Rust。Rust是一种系统编程语言，旨在提供类似C++的速度和灵活性，同时避免导致系统不稳定、错误或被攻击者远程利用的大类内存或线程安全问题。Rust的目标是生成安全、高效且可靠的代码。

Lamellar在安全计算生态（尤其是Rust生态）中占据独特地位。具体而言，它为HPC网络和编程模型（包括PGAS）提供高层支持，以充分利用Rust的内存和线程安全特性，同时几乎不牺牲性能。尽管网络安全在受控系统为主的HPC社区中历来并非关注重点，但内存相关错误仍可能导致执行时间浪费、结果错误和数据损坏。我们基于OpenFabrics接口（OFI）构建了轻量级C库（ROFI）和Rust绑定（ROFI-sys），以在Rust应用中访问高性能网络架构。这些库构成了Lamellar软件栈的底层（图1），并可作为独立库使用。

本文结构如下：第三节描述Lamellar运行时的设计、架构和关键API；第四节通过实验对比Lamellar通信抽象与基于OpenSHMEM的Chapel、C/C++方案在BALE内核子集上的性能；第五节讨论未来工作方向。

**II. 相关工作**

MPI/OpenMP是高性能并行编程模型的**事实标准**，长期主导HPC应用开发。MPI采用显式处理器间数据交换，OpenMP是面向节点内共享内存编程的跨语言标准。尽管未实现PGAS编程模型，二者的组合体现了PGAS系统的核心概念。

SHMEM是CRAY开发的分布式计算库，包含单向通信、点对点通信、集体通信、共享内存视图和对全局变量的原子操作。这些能力基于分布式通信框架（如MPI），为高层PGAS系统提供接口。OpenSHMEM是标准化的PGAS模型。

PGAS模型的奠基性工作包括Aggregate Remote Memory Copy Interface（ARMCI）和Communications Run-time for Extreme Scale（ComEx）。这些早期研究提出了远程内存访问（RMA）、互连特定支持、多线程、组感知通信和通用Active Message等概念。Global Arrays（GA）在此基础上实现了清晰的PGAS模型。

GASNet是支持PGAS系统的语言无关通信库，通过高性能、网络无关的通信接口实现。GASNet广泛用于开源和厂商支持的PGAS编程模型库。

Chapel是Cray开发的基于语言的PGAS编程模型，利用GASNet进行通信和运行时数据管理。其特点包括通用并行编程概念、全局视图抽象和基于数据局部性的操作控制。

UPC（Unified Parallel C）是C语言的扩展，用户可定义共享数组和指向全局内存空间的指针，并提供基于数组元素亲和性分发迭代的**forall**循环。其C++扩展UPC++提供了面向对象的PGAS编程模型，支持更广泛的异步操作。

Charm++是基于C++的面向对象可移植并行编程语言，通过显式编程结构分离串行与并行对象，以消息驱动方式执行操作。

HPX提供基于任务的PGAS编程模型，通过任务定义整体操作，避免MPI中显式数据分发和通信的复杂性，支持节点间动态通信和自主负载均衡。

Exstack、Exstack2和Conveyors是基于OpenSHMEM或UPC的C语言聚合库。Exstack采用同步聚合（类似BSP模型），Exstack2是其异步版本，Conveyors通过多跳聚合减少内存占用并提升带宽利用率。第四节将Lamellar与这三者对比。

Habanero C/C++库（HClib）是基于异步多任务（AMT）编程模型的运行时，用于实现细粒度异步BSP模型的"Selectors" API。其点对点远程操作表示为细粒度Actor消息，抽象了消息聚合和终止检测的复杂性。第四节将Lamellar与基于Selectors的实现对比。

在Rust社区中，我们发现另两项HPC相关尝试：一是Conveyors库的Rust实现，因构建失败未纳入实验分析，二是Selectors-rs——HClib Selectors API的Rust重实现，不是简单的进行了封装。但因未找到公开代码，无法进行比较。

在HPC领域外，Tokio和Rayon是Rust生态中最流行的异步运行时。Tokio面向I/O密集型应用，如Web服务器/客户端，提供异步标准库和多线程执行器。Rayon是Rust数据并行的**事实标准**，支持多线程工作窃取和独立于Rust Futures的fork/join模型。二者互补：Rayon不支持分布式执行，Tokio对CPU密集型应用无优势。

其他工具包括提供异步标准库组件的async\_std、轻量运行时smol和Fuchsia操作系统执行器fuchsia-async。这些工具为HPC提供了潜在支持。

**III. 运行时设计**

以下小节将介绍Lamellar运行时的各层（如图1所示），包括其职责和用户接口。运行时使用的术语包括：

**•节点（Node）**：物理计算节点。

**•处理元素（PE, Processing Element）**：执行Lamellar应用的最小逻辑单元。PE使用异步线程池，一个计算节点可分配多个PE，但通常每个节点运行一个PE。

**•世界（World）**：应用中所有PE的集合，数量由系统启动器（如Slurm）控制。

**•团队（Team）**：World的子集，支持子团队。

**•远程直接内存访问（RDMA）内存区域**：可从远程节点访问的内存段。

**•Active Message（AM）**：一种消息，到达远程PE时执行代码。

**•单向操作（One-sided）**：仅需调用PE参与，无需与远程PE协调。

**•集体操作（Collective）**：需团队内所有PE协调后才能继续。

**A. Lamellae层**

运行时最底层是与网络接口通信的抽象层，称为**Lamellae Trait**。在Rust中，Trait是一组可供多种类型调用的方法集合，类似于Go/Java中的“接口”。若类型T实现了Trait U，即可执行U中的方法。

Lamellae Trait是运行时与网络接口之间的桥梁，功能包括：

**•**初始化/反初始化

**•**获取PE ID和World中的PE数量

**•**分配/释放内存区域

**•**执行远程**put**/**get**传输

**•**提供同步原语（如屏障）

本节介绍Lamellae Trait的三种实现。

**1. ROFI & ROFI-sys**

**ROFI**是基于OpenFabrics（OFI）的轻量传输层，旨在简化分布式环境中的数据传递。其目标是提供比OFI更高层的抽象，但不实现复杂通信协议（由上层处理）。ROFI专为与Rust代码集成设计，通过Bindgen等工具减少Rust FFI（外部函数接口）的问题，支持同步/异步RDMA API（如**PUT**/**GET**），将消息视为字节序列，不解析内容。未来计划直接用Rust绑定**libfabric**替代此C库。

**ROFI-sys**是暴露ROFI公共接口的底层Rust crate。由于Rust编译器无法保证外部库的行为安全，所有ROFI-sys函数均标记为**unsafe**。Lamellar的策略是通过高层**安全抽象**封装这些**unsafe**调用，减少用户直接接触内存/线程错误的风险。

Lamellar应用中的数据传递分为两类：

**第一种是简单的RDMA put/get**：直接传输原始比特流。**第二种是运行时序列化/反序列化**：通过Active Message在PE间传递结构化数据。

ROFI\_Lamellae在ROFI(-sys)基础上扩展，支持Lamellar应用的分布式执行。若将ROFI(-sys)视为PE间字节传输工具，则ROFI\_Lamellae负责所有其他数据传输逻辑。

对于RDMA，ROFI\_Lamellae仅对ROFI(-sys)调用做轻量封装。这类传输通常为**unsafe**，因为Rust无法推断远程PE的内存访问。运行时内部使用的RDMA传输通过**unsafe**接口暴露，例如**LamellarMemoryRegion**和**UnsafeArray**（见III-D和III-F节）。

对于需序列化的消息，Lamellae采用“标志位”传输机制：发送PE通知接收PE数据可读，接收PE在本地缓冲区就绪后拉取数据，再通知发送PE释放资源。Lamellar使用双缓冲消息队列处理异步性，并通过聚合大消息优化网络利用率。

Lamellae还管理应用中的RDMA内存区域。初始化时，每个PE分配一大块RDMA内存区域，部分供运行时内部使用，包括如消息缓冲和状态表，剩余部分作为用户数据结构的动态堆，如AM和单向内存区域。

分布式数据结构,如III-F节的分布式数组,的内存区域由Lamellae管理，其构造为集体操作，但仅阻塞调用线程，其他线程可继续工作。与运行时内存区域不同，这些区域会在数据结构销毁时释放。

**2. Shmem实现**

为支持非分布式环境开发，我们实现了基于POSIX共享内存的Lamellae Trait。尽管共享内存可直接传递数据对象而无需序列化，但为保持与ROFI Lamellae行为一致，我们选择模拟其逻辑。这简化了调试，并确保从共享内存切换到分布式环境时的无缝迁移。

共享内存Lamellae的内部数据结构与ROFI Lamellae相同，关键区别在于用共享内存段替代RDMA内存区域。对用户而言，切换二者应是透明的。

**3. SMP实现**

SMP Lamellae面向单进程多线程应用（仅一个PE），无需数据传输或序列化。由于所有内存分配由Rust标准分配器处理，此实现不涉及RDMA内存区域等概念。

尽管我们确保基于SMP Lamellae开发的应用可在Shmem和ROFI Lamellae上运行，但用户需注意分布式同步调用的正确匹配。由于编译时难以推断这些调用，运行时会进行有限检查并警告潜在问题。文档明确标明了集体调用和单向调用。

**B. 线程池层**

Lamellae层之上是**线程池层**，负责管理PE中的所有工作线程（包括异步任务的入队和执行）。Lamellar全面支持Rust Futures和**async/await**模型，因此其线程池被视为Rust执行器。通常，线程池处理运行时API生成的异步任务，包括AM执行和Lamellae的通信任务（处理数据传输）。用户可提交自定义Futures到线程池，从而兼容第三方库（如Async-std和Tokio的部分功能）。

Lamellar线程池采用基于PE的工作窃取（work-stealing）策略。未来计划探索更多优化，如：

**•**利用NUMA拓扑减少跨域性能损耗

**•**支持跨PE工作窃取

**•**允许用户自定义线程池实现

**C. Active Message层  
Lamellar 基于异步主动消息（AMs）构建，其核心思想是：消息不仅包含待处理的数据，还包含元数据（如函数指针），用于指示数据到达目标 PE（处理单元）后的处理逻辑。Lamellar 的运行时支持在分布式环境中向远程 PE 发送并执行用户自定义的 AMs。**

**AMs 的 Rust 实现**

**Lamellar 通过以下方式向用户暴露 AMs 功能：**

**1.LamellarAM Trait：定义 AMs 的核心行为。**

**2.过程宏（Procedural Macros）：利用 Rust 的过程宏特性，在编译时将自定义语法转换为合法 Rust 代码，实现领域特定语言（DSL）。**

**D. PGAS层**

Lamellar通过RDMA传输AM，由于这些操作本质不安全，运行时提供两层PGAS抽象：**底层（不安全）**供运行时内部使用，用户需用**unsafe**代码块隔离。**高层（安全）**为最终用户设计，提供安全性和生产力特性。

**1. 共享内存区域（SharedMemoryRegion）**

**SharedMemoryRegion**是团队内PE集体分配的RDMA内存区域的包装器。分配直接来自网络硬件（而非运行时堆）。创建**SharedMemoryRegion**是集体阻塞调用，但仅阻塞调用线程（如**main**），线程池可继续执行其他任务。

**SharedMemoryRegion**提供以下方法：

**•put(dest\_pe, index, src\_buf)**：写入远程PE的内存区域。

**•get(src\_pe, index, dst\_buf)**：从远程PE读取数据。

提供非阻塞（需用户检查完成状态）和阻塞（依赖运行时检测）版本。用户可直接通过指针或切片访问本地数据，但此为**unsafe**操作（可能被远程PE写入）。

**SharedMemoryRegion**是**分布式原子引用计数对象（Darc）**的特化类型，可通过AM传递（见III-E节）。

**2. 单向内存区域（OneSidedMemoryRegion）**

**OneSidedMemoryRegion**类似**SharedMemoryRegion**，但仅需调用PE参与分配。其**put**/**get**操作始终针对创建PE，同样通过Darc机制支持AM传递。

**E. 分布式原子引用计数（Darcs）层**

Darc（分布式原子引用计数指针）是Lamellar引入的新型智能指针，扩展了Rust标准库中的**Arc**（原子引用计数指针）功能。**Arc**是Rust安全并发编程的核心，支持多线程环境下的**共享所有权**，而Darc将这一概念扩展到**分布式环境**。

Darc API是暴露给用户的**安全**分布式计算抽象，其核心特性包括：

**•全局生命周期管理**：只要任意PE持有引用，被指向的对象在所有创建PE上保持有效。

**•默认禁用内部可变性**：类似**Arc**，若需通过Darc修改数据，需结合**Mutex**、**RwLock**或实现**Sync** Trait的类型（如原子类型）。

Darc通过AM传递，支持对Rust对象的分布式访问和操作。创建Darc是团队内所有PE的集体调用，每个PE需提供被指向类型的实例：

rust

复制

fn Darc::new<T>(team, item: T) -> Darc<T>

Darc数据结构存储在RDMA内存区域中，支持通过AM传递。引用计数通过**Clone**、序列化和反序列化自动维护。销毁Darc是异步过程，需所有PE确认无剩余引用后触发实际释放。

**F. LamellarArray层**

LamellarArray是基于下层组件构建的**安全PGAS抽象**，提供分布式数组功能，包括：

**•**多种数组类型（根据安全需求）

**•**类RDMA的**put**/**get**接口

**•**逐元素操作

**•**块（Block）或循环（Cyclic）数据分布

**•**迭代器方法

**•**归约操作

**•**子数组创建

**1. 数组类型**

Lamellar支持四类数组，按安全保证划分：

1. **UnsafeArray**：无安全保证，PE可任意读写，标记为**unsafe**，建议仅内部使用。
2. **ReadOnlyArray**：禁止写入，PE可自由读取任意位置。
3. **AtomicArray**：元素访问为原子操作，分两种子类型：
4. **NativeAtomicArray**：元素为Rust原子类型（如**AtomicUsize**）。
5. **GenericAtomicArray**：元素受1字节**Mutex**保护。

**4）LocalLockArray**：每个PE上的数据区域由单个**RwLock**保护。

数组类型可通过**into\_atomic()**、**into\_read\_only()**等API转换，转换是集体调用且需满足唯一引用条件。

**2. RDMA类操作**

LamellarArray提供类似**SharedMemoryRegion**的**put**/**get**操作，但遵循数组类型的安全约束。例如：

**•ReadOnlyArray**支持安全的RDMA **get**（因数据不可变）。

**•UnsafeArray**提供**unsafe**的直接RDMA接口（不推荐用户使用）。

**3. 逐元素操作**

LamellarArrays 为数组中的单个元素提供了多种操作。需注意以下几点：

**1.转换可能导致死锁**：由于 Rust 的作用域规则（编译器不进行死锁检测），部分操作可能引发死锁。

**2.支持的操作类型**：包括算术运算（**+**,**-**,**\***,**/**,**%**）、位运算符（**&**,**|**,**^**,**~**）、左移/右移运算符、**compare\_and\_exchange**运算符，以及**load**、**store**、**swap**等原子操作。

操作可表现为两种形式：**单元素请求**以**array.op(index,val)**形式调用。例如，**array.add(5,100)**表示对索引5处的元素加100。**批处理API**将多个操作聚合到单个请求中，支持以下模式：**多索引-单值**：如**array.batch\_store([20,2],10)**将索引20和2处的元素设为10。**单索引-多值**：如**array.batch\_mul(20,[2,10])**先将索引20处的元素乘以2，再将结果乘以10。**多索引-多值**如**array.batch\_bit\_or([0,105,67],[127,0,64])**，索引0处的元素与127按位或，索引105处的元素与0按位或，索引67处的元素与 64 按位或。Lamellar 还提供fetch变体操作，可在应用操作前返回当前值。

**运行时**会计算每个数组索引对应的目标 PE（处理单元）和偏移量，并将操作按目标 PE 批量打包为单条消息。批处理消息仅支持包含相同类型的操作。操作通常按发起顺序执行，但运行时不保证严格顺序。每个操作返回一个 **Future**，可通过**await**等待其完成；对于**fetch**操作，还可通过该**Future**获取结果。

**4. 迭代器**

提供三类迭代器：**DistributedIterator**可以分布式并行迭代，集体操作，自动处理远程数据拉取。**LocalIterator**是单侧迭代器，仅遍历调用PE的本地数据。**OneSidedIterator**串行迭代整个数组，运行时自动管理远程数据。

**可执行示例：使用 AtomicArray 实现直方图基准测试**

（对应第 IV-B1 节讨论的直方图实现）  
以下代码重点展示了 Lamellar 数组的抽象特性：

**•第 8 行**：创建新数组，指定类型（**usize**）、关联的团队（此处为 **world**）、全局长度及分布式数据布局。

**•第 15 行**：执行 **batch\_add** 操作，接受随机生成的索引列表（第 9–12 行），对每个索引处的元素加 1。

**•第 18 行**：对数组应用 **sum** 归约操作，并通过总和验证未遗漏任何更新（第 19 行）。

1 use \_\_\_\_\_\_\_::array::prelude::\*;

2 use rand::Rng;

3 use std::time::Instant;

4 const T\_LEN: usize = 1\_000\_000; // 全局长度

5 const L\_UPDATES: usize = 10\_000\_000; // 每 PE 的更新次数

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) // 生成随机索引

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)); // 直方图核心逻辑

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 }

**代码清单 2**：**AtomicArray** 直方图实现示例

**IV. 基准测试**

本节评估Lamellar性能：IV-A节分析类**put**操作的带宽曲线，IV-B节对比Lamellar与基于OpenSHMEM的C/C++和Chapel在BALE内核子集上的表现。

测试环境：48节点集群（实际使用32节点），每节点配备：

**•**双路AMD EPYC 7543（32核/CPU，共64核）

**•**256GB DDR4-3200内存

**•**Mellanox HDR-100 InfiniBand（100Gb/s峰值带宽）

**A. 带宽测试**

图2对比不同通信原语的**put**操作带宽：

**•Rofi(libfabric)**：性能上限（直接调用libfabric）。

**•MemRegion**：Lamellar最低层**unsafe**调用（接近Rofi性能）。

**•UnsafeArray**：手动终止检测，性能略低于MemRegion。

**•安全数组（AtomicArray等）**：因运行时开销，性能较低，但随消息增大（>100KB）趋近UnsafeArray。

安全抽象的性能损耗是未来优化重点。

**B. BALE内核测试**

对比以下实现：

1. **C语言**：Exstack（同步）、Exstack2（异步）、Conveyors（多跳聚合）。
2. **C++**：Selectors（基于HClib AMT模型）。
3. **Chapel**：自动聚合优化。
4. **Lamellar**：两种实现（手动AM聚合与AtomicArray API）。

**1. 直方图（Histogram）**

* **模式**：每个PE随机递增分布式数组元素。
* **结果**（图3）：Lamellar AM聚合方案扩展性最佳，AtomicArray因运行时批量调度开销略低但仍优于其他实现。

**2. 索引聚集（IndexGather）**

* **模式**：从分布式数组随机读取元素。
* **结果**（图4）：Chapel的专用**CopyAggregator**性能最优，Lamellar的**ReadOnlyArray**优于手动AM实现（因更好平衡收发负载）。

**3. 随机排列（Randperm）**

* **模式**：生成分布式随机排列数组。
* **Lamellar四种实现**：
  + **Array Darts**：基于**AtomicArray**的批量比较交换。
  + **AM Dart**：手动AM聚合。
  + **AM Dart Opt**：优化通信（失败时本地重试）。
  + **AM Push**：最小化通信（直接推送数据到目标PE）。
* **结果**（图5）：**AM Push**性能最佳（但可能牺牲严格随机性）。

**V. 结论与未来工作**

Lamellar证明了Rust可作为HPC运行时的可行选择，通过**安全抽象**实现与C/C++/Chapel竞争的性能。未来方向包括：

1. 优化运行时自动聚合性能。
2. 扩展数组迭代器功能以匹配Rust标准库。
3. 减少数组类型开销，提升易用性。

我们计划与领域科学家合作，进一步验证Lamellar在特定场景的适用性。随着工业界对Rust等内存安全语言的采纳，HPC领域亦有望受益。