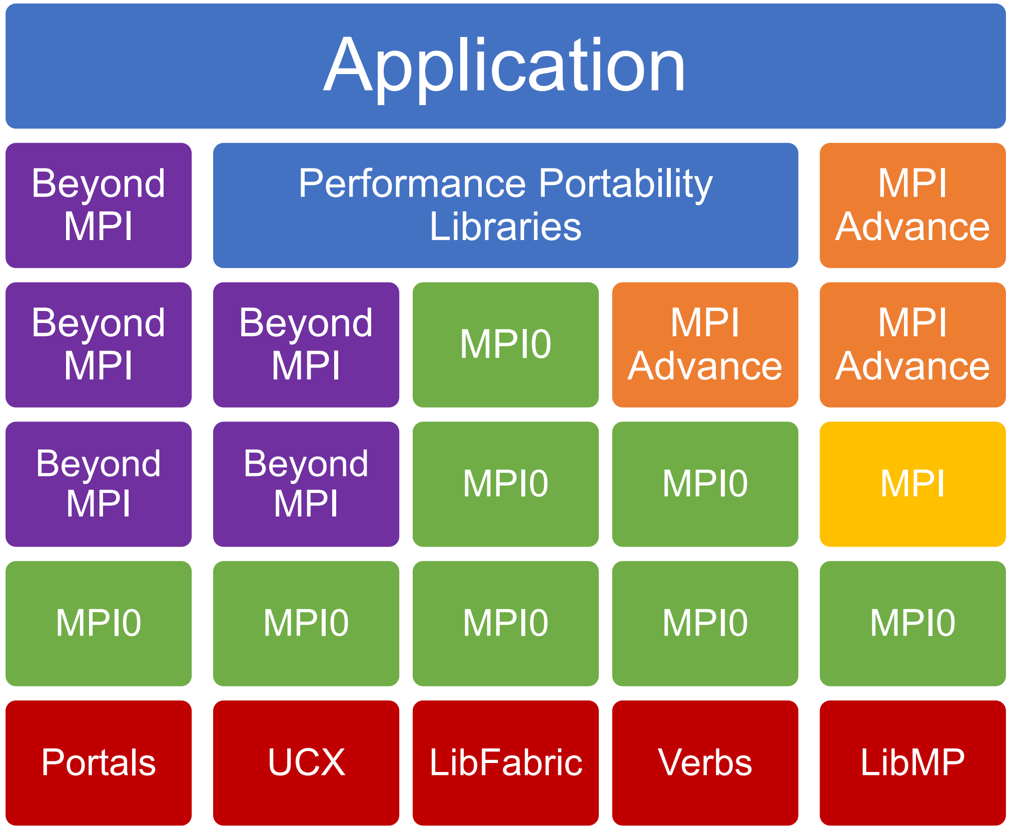
Rethinking Communication Abstractions

Thomas Hines, Anthony Skjellum, Purushotham Bangalore

askjellum@tntech.edu

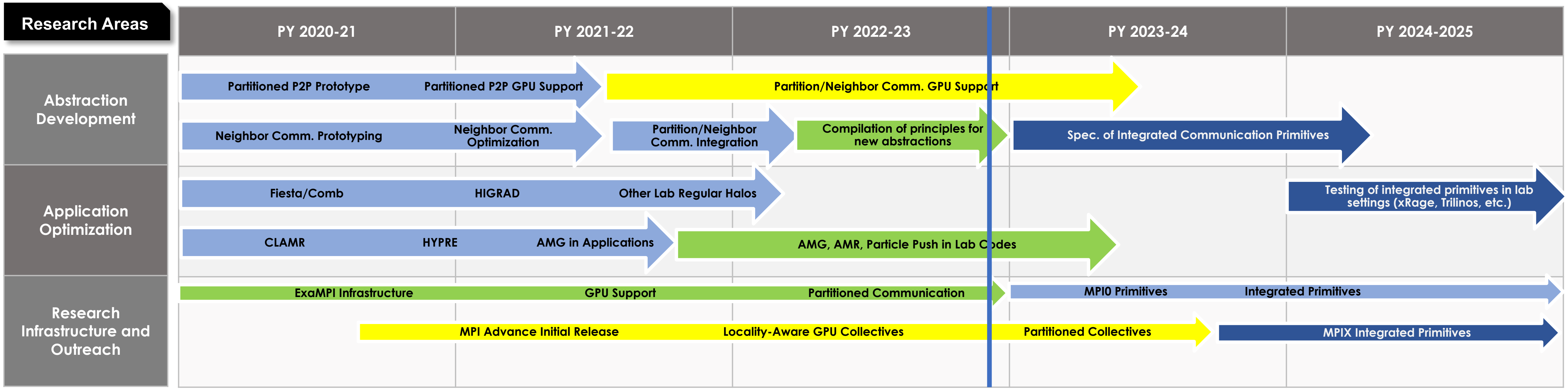
Preview

- New performant primitives are needed well beyond what MPI can do… driven by our experience and community best practices we've discovered
  - Low-level – implementer-facing
  - High-level – library and application facing
- C isn't a sufficient interface anymore for optimizing – C++
- We will explain aspects of "Beyond MPI" as our strategy



Updated 5-year Project Roadmap

- Beyond MPI goals in FYs 4 and 5
- MPI0, MPI Advance, Higher-level Abstractions



Research Areas
PY 2020-21
PY 2021-22
PY 2022-23
PY 2023-24
PY 2024-2025
Abstraction Development
Partitioned P2P Prototype
Partitioned P2P GPU Support
Partition/Neighbor Comm. GPU Support
Neighbor Comm. Prototyping
Neighbor Comm. Optimization
Partition/Neighbor Comm. Integration
Compilation of principles for new abstractions
Spec. of Integrated Communication Primitives
Application Optimization
Fiesta/Comb
HIGRAD
Other Lab Regular Halos
Testing of integrated primitives in lab settings (xRage, Trilinos, etc.)
CLAMR
HYPRE
AMG in Applications
AMG, AMR, Particle Push in Lab Codes
Research Infrastructure and Outreach
ExaMPI Infrastructure
GPU Support
Partitioned Communication
MPI0 Primitives
Integrated Primitives
MPI Advance Initial Release
Locality-Aware GPU Collectives
Partitioned Collectives
MPIX Integrated Primitives

Beyond MPI—Leaving MPI Behind (or Aside)

- MPI has provided a lot of great success
- New abstractions are needed, forum is too slow and conservative
- MPI will remain in the framework role of runtime and non-performance critical APIs
- Two kinds of APIs are needed based on our findings
    - Application-facing + performance portable APIs for message passing and data reductions
    - Implementation-facing, low-level APIs in groupings to support semantics that lightly abstracts the various, complex network/accelerator/CPU

This talk is about

- MPI0---low-level APIs for really high performance (down-up)
- Rethinking datatypes (more performance and offload potential)
- High-level abstractions, including some based on community best practices (e.g., Token library)
- Achieving high performance and recovering portability
- Working alongside MPI

We love MPI, but we need more (and less)…

- +Persistent collective communication is a good addition in MPI-4
- +Compromises in new APIs we've helped put in MPI-4 --- partitioned point-to-point communication is a great start but has issues
- -Derived Datatypes are a slow mechanism
- -Communicators pose high overheads when working with neighborhood collectives
- -General send/receive semantics bad for accelerators
- -Asynchronous notification and triggering absent in MPI still
- --Full MPI semantics and too much or mismatch in various ways

Lessons Learned

- Two-sided “look and feel” important/good for programmers
- Efficient semantics for 1-sided implementation essential – channels, persistence, partitions
- MPI doesn’t really fit the accelerator programming model
- C++ native interfaces needed – for performance, integration, productivity

MPI will still orchestrate the application, but…

- New abstractions will run in performance-critical regions
- High-level abstractions will be designed for performance portability and a degree of application domain specificity
- Low-level primitives (aka MPI0) will be performant, but different groups will work better on different hardware
- High-level abstractions used by implementers choose which MPI0 approaches meet their performance and semantic needs

Low-level Primitives (MPI0)

MPI0 at a glance

- Work within a standard MPI execution environment
- C/C++ facing APIs (small sets)
- Flavors – semantic complexity varied for different use cases (e.g., static flavor)
- Buffer/memory coupled with transfers including memory kinds
- Point-to-point channels (partitions emphasized)
- Make it easy for users to compose operations and trigger sequences/graphs [also considered in MPI-5 standard]
- Communicators reconsidered – maybe replaced, achieve separation through point-to-point and collective channels …
- MPI message passing APIs may also be implemented over MPI0 abstractions in future

MPI0 – Static

- For known buffers and a static communication pattern
- Move as much as possible into the setup phase and out of the main loop
- Receive not needed – the receive buffer is known at setup
  - No such thing as an unexpected message

MPI0 – Static Interface

- A transfer is the basic unit, set up once
  - Setup_send(buffer, size, dest, tag, &transfer)
  - Setup_recv(buffer, size, src, tag, &transfer)
  - Setup_wait(transfer)
- A transfer can be triggered over and over again (send side)
  - Trigger(transfer, msg_size) – msg_size must be <= the setup size
  - Wait(transfer, &size)
  - Test(transfer, &size)
- Wait or Test is called on both the sender and the receiver side

MPI0 – Static Implementation

- Implemented using IB Verbs
- Send boils down to RDMA put
  - Matching, pinning, RDMA address transfer all done during setup

- Tested in Pulse on Lassen
  - Whole app 3%-5% faster than Spectrum MPI (1 rank per node, 27 nodes)
  - Test case is a simple, structured problem

MPI0 – Dynamic

- The Dynamic library controls the buffers
- Send side
  - Acquire buffer, pack into buffer, send
  - Destructive send – application cannot reuse buffer
- Receive side
  - Application does not give buffer; library gives the application a buffer
  - No matching
  - Get the next message in (any source, any tag)

High-level Abstractions

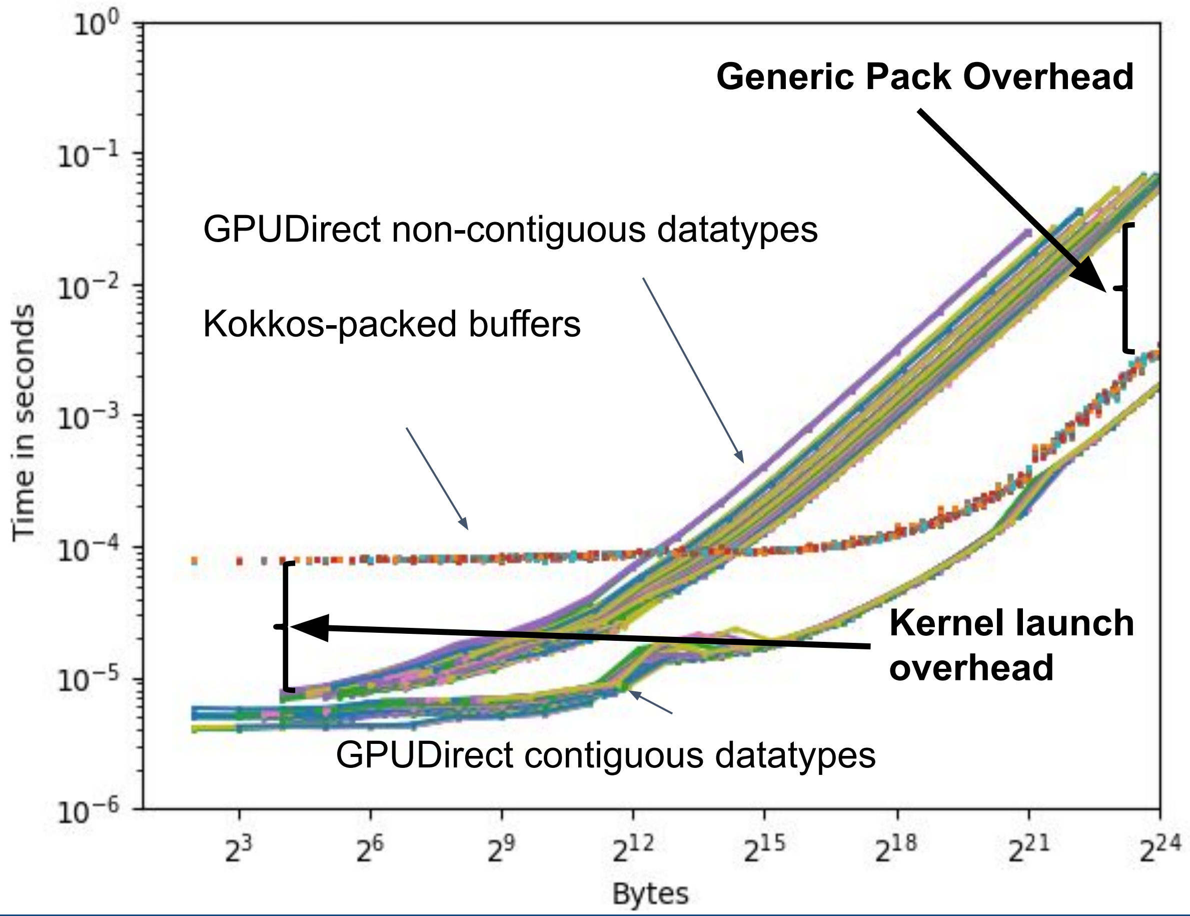
Datatypes

Tokens

MPI+Kokkos

Irregular Data Communication Abstractions

- Need to efficiently communicate and compute (e.g. reduce or gather) irregular data
  - Generic datatypes can be faster for small irregular data
  - Generic datatypes prohibitively slow for large irregular data
- Need fast application-customized datatype abstractions!
  - Support effective collectives
  - Manage communication memory costs



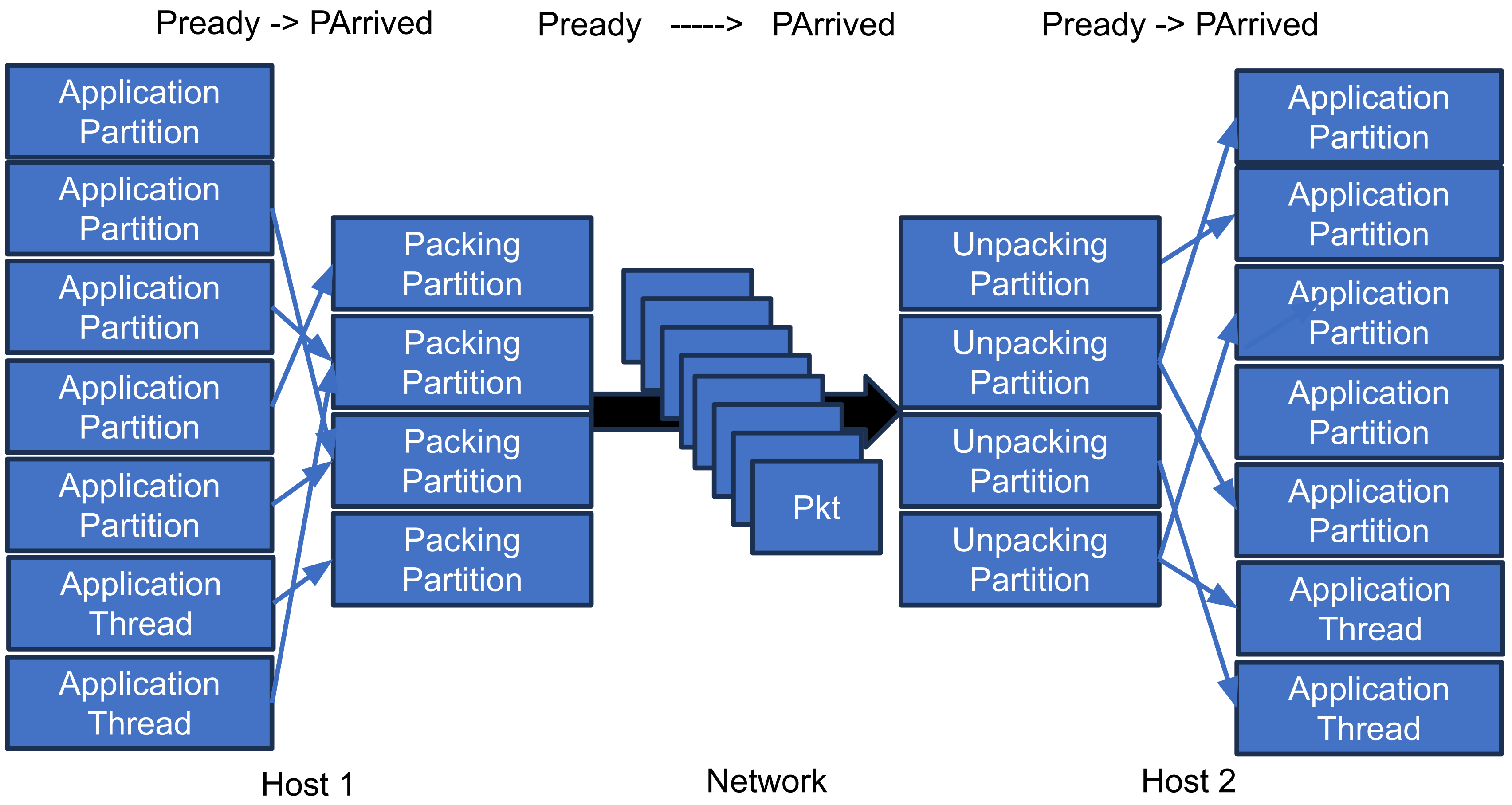
New Abstraction: Active Datatypes

- Idea – application provides custom packing code that can be tightly integrated with communication
- Signpost: Applications *already* write fused packing loops
  - Simple and faster than highly-optimized datatype generic implementations
  - Lack benefits of tight integration with the communication infrastructure
- Challenges: integrate application packing with communication system

1. Preserve/manage parallelism granularity through packing and communication
2. Persist application accelerator packing code to avoid kernel launch overheads

Preserving Fine-Grain Parallelism

- Maintaining parallelism paramount to maximize performance
  - Fine-grain application parallelism creates/ consumes data
  - Communication system aggregates to manage parallelism on the wire
- Enabling management of parallelism and aggregation essential
- Partitioned concept enables this in packing!
- Manage parallelism at all levels (end-to-end)



Token Library Model--- Not in MPI

- Best practices of application communication community
- Usable and performant, can be faster if not layered on MPI
- Key aspects
  - Service to locate where data goes (unknown senders or unknown receivers)
  - Efficient collective data transfer
  - Neighborhood-type reductions on classes of messages received
- Common abstraction for over a decade in NNSA-code helper libraries (L7, Token)
- Not close to anything under consideration by the MPI Forum
- We will create a performance-portable abstraction with efficient implementation using MPI0 (prevent reinvention of wheel in many apps)

Kokkos+MPI Goals

- To improve the general programming experience when using MPI with Kokkos.
- To minimize the possibility of bugs from MPI+Kokkos programs
- To enable significant optimizations for MPI+Kokkos at the language binding level or below
- Kokkos offers the opportunity to move beyond legacy MPI

Kokkos + ExaMPI Implementation

- The template parameters decide the datatype information for View operations
- MPI_Send's counterpart, MPI_Recv receives the Payload send in MPI_Send, then wraps that in a View object and sends that to the pointer passed as a parameter.
- Moving soon to

  MPI_Send<View_t>(View_t * buf, int dest, int tag, MPI Comm comm)

  MPI_Recv<View_t>(View_t * buf, int source, int tag, MPI Comm comm)

  with type and structural inference from view

```
Kokkos::View<int*>check( ”check”, n );
MPI_Kokkos_Send<Kokkos::View<int*>, int>(&check, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
int* check_arr = check.data();
MPI_Send(check_arr, n, MPI_INT, 0, 0, MPI_COMM_WORLD);
```

Integration

Integrated Primitives—Work for YRs 4&5

- Pulling our abstractions and primitives into best practices we've discovered
- Top-down
  - Collectives
  - Token APIs (Topology discovery, management, partitions, new datatype abstractions)

- Mid-level
  - C++ API alternatives with Kokkos integration
  - Removing mid-level code
  - Ability to push kernels down into GPU, DPU
- Down-up
  - MPI0 – fast transfers
  - Support for tokens
  - Support for topology

Summary

- New performant primitives are needed well beyond what MPI can do… driven by our experience and community best practices we've discovered
- We've learned that new abstractions/primitives are needed
  - Low-level – simpler semantics, early evidence of greater performance; **implementer facing**
  - High-level – collective operations that reflect applications; **library and application facing** (Token library abstraction as efficient, app facing API; Datatypes – need to replace with effective alternatives)
- Connect with Kokkos+ExaMPI work to improve abstractions (see Evan Suggs' poster) – C isn't a sufficient interface anymore for optimizing
- Next steps include
  - Resolving lack of vendor-independent GPU-triggering comm
  - Integrating low-level primitives and high-level abstractions into more use cases/apps/libraries
  - Demonstrating greater performance advantages

Thank you!