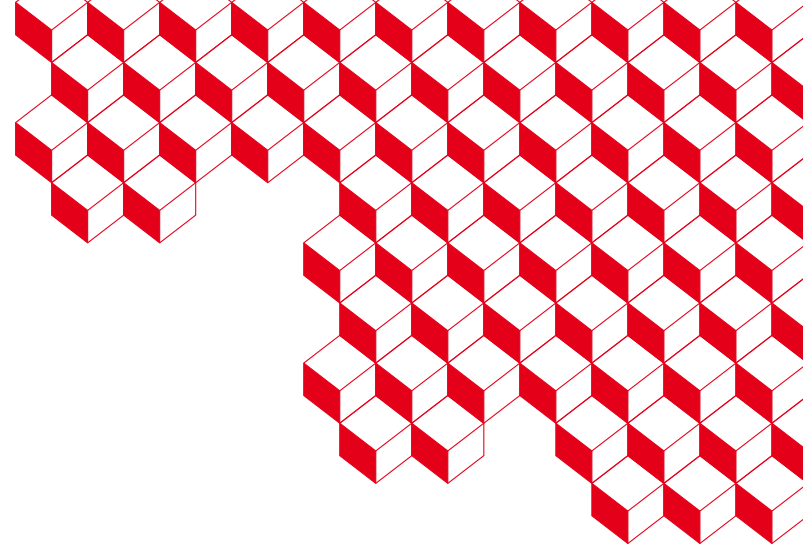




Write Once, Communicate Anywhere

API Design for Performance Portable Communication

Gabriel Dos Santos^{1,2}, Hugo Taboada^{1,2}, Cédric Chevalier^{1,2} & Marc Pérache^{1,2}

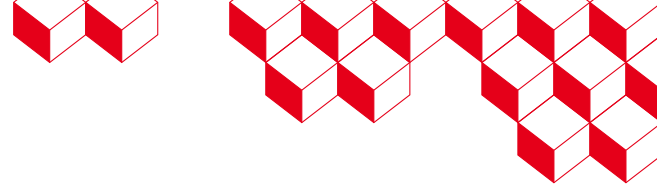
¹ CEA, DAM, DIF, F-91297 Arpajon, France

² LiHPC, Université Paris-Saclay, CEA, Paris, France

Acknowledgements

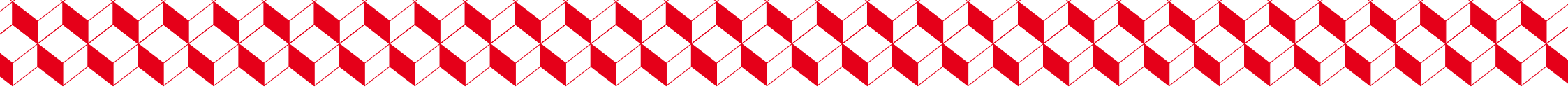
Thanks to all the Kokkos Comm contributors!

- CEA, France
- Sandia National Labs, NM, USA
- Tennessee Technological University, TN, USA



Contents

- 1. Motivation**
- 2. Context**
- 3. API Design**
- 4. Evaluation**
- 5. Conclusion**



1. Motivation

Distributed communication ecosystem



HPC community: MPI

- General-purpose message passing
- Used everywhere, works everywhere

AI community: *CCLs

- Vendor-optimized communications on the GPU
- NVIDIA NCCL, AMD RCCL, Intel oneCCL, Meta HCCL
- Portability wrappers: HiCCL

Distributed communication ecosystem



HPC community: MPI

- General-purpose message passing
- Used everywhere, works everywhere

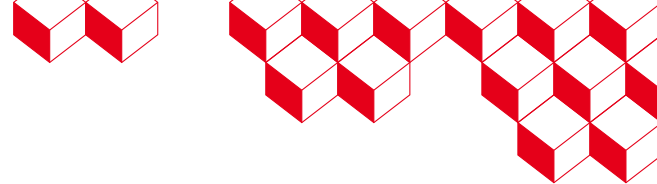
HPC interest in *CCLs for performance reasons

- Integrated with and optimized for GPU-driven communication
- Real-world example: NVIDIA's HPL runs NCCL instead of MPI

AI community: *CCLs

- Vendor-optimized communications on the GPU
- NVIDIA NCCL, AMD RCCL, Intel oneCCL, Meta HCCL
- Portability wrappers: HiCCL

Distributed communication ecosystem



HPC community: MPI

- General-purpose message passing
- Used everywhere, works everywhere

AI community: *CCLs

- Vendor-optimized communications on the GPU
- NVIDIA NCCL, AMD RCCL, Intel oneCCL, Meta HCCL
- Portability wrappers: HiCCL

HPC interest in *CCLs for performance reasons

- Integrated with and optimized for GPU-driven communication
- Real-world example: NVIDIA's HPL runs NCCL instead of MPI

APIs and capabilities differ

- Exposed communication primitives
- Accessible memory kind
- Library-specific objects (communicator, datatypes, reduction operators, etc.)
- Execution model: ordering, matching and completion

Distributed communication ecosystem

HPC community: MPI

- General-purpose message passing
- Used everywhere, works everywhere

HPC interest in *CCLs for performance reasons

- Integrated with and optimized for GPU-driven communication
- Real-world example: NVIDIA's HPL runs NCCL instead of MPI

APIs and capabilities differ

- Exposed communication primitives
- Accessible memory kind
- Library-specific objects (communicator, datatypes, reduction operators, etc.)
- Execution model: ordering, matching and completion

AI community: *CCLs

- Vendor-optimized communications on the GPU
- NVIDIA NCCL, AMD RCCL, Intel oneCCL, Meta HCCL
- Portability wrappers: HiCCL

Challenge

How to design an unified interface that enables HPC applications to leverage *CCLs alongside MPI?

Application interaction: data model

Modern HPC applications structure their data:

- Often leverages modern C++ containers
- May reside in different kinds of memory (host, device, etc.)
- May be multidimensional and/or non-contiguous



Application interaction: data model

Modern HPC applications structure their data:

- Often leverages modern C++ containers
- May reside in different kinds of memory (host, device, etc.)
- May be multidimensional and/or non-contiguous

Communication libraries do the opposite:

- Commonly destructure the data
- Describe the buffer to communicate as a `(address, count, datatype)` tuple.

All of these can be statically inferred from the structured data.



Application interaction: data model

Modern HPC applications structure their data:

- Often leverages modern C++ containers
- May reside in different kinds of memory (host, device, etc.)
- May be multidimensional and/or non-contiguous

Communication libraries do the opposite:

- Commonly destructure the data
- Describe the buffer to communicate as a `(address, count, datatype)` tuple.

All of these can be statically inferred from the structured data.

But, data may need to be adapted so it is “communicable”; i.e., it must be accessible and contiguous:

- **Memory staging:** copy the data in an accessible memory kind
- **Layout mapping:** represent the data as a contiguous range

Application interaction: data model

Modern HPC applications structure their data:

- Often leverages modern C++ containers
- May reside in different kinds of memory (host, device, etc.)
- May be multidimensional and/or non-contiguous

Communication libraries do the opposite:

- Commonly destructure the data
- Describe the buffer to communicate as a `(address, count, datatype)` tuple.

All of these can be statically inferred from the structured data.

But, data may need to be adapted so it is “communicable”; i.e., it must be accessible and contiguous:

- **Memory staging:** copy the data in an accessible memory kind
- **Layout mapping:** represent the data as a contiguous range

Challenge

How to design a unified interface that consumes and adapts structured data while preserving performance?

Application interaction: execution model

Data adaptation impacts the execution too: **we need to model dependencies between operations**

Required data adaptation operation must:

1. Start *after* a kernel producing that data finishes
2. Finish *before* the associated communication operation starts

Application interaction: execution model

Data adaptation impacts the execution too: **we need to model dependencies between operations**

Required data adaptation operation must:

1. Start *after* a kernel producing that data finishes
2. Finish *before* the associated communication operation starts

Execution models for asynchronous communication differ

MPI:

- Does not know about dependent operations
- Weak progress guarantess: needs polling to make progress
- Fine-grain synchronizations with `MPI_Request`

Application interaction: execution model

Data adaptation impacts the execution too: **we need to model dependencies between operations**

Required data adaptation operation must:

1. Start *after* a kernel producing that data finishes
2. Finish *before* the associated communication operation starts

Execution models for asynchronous communication differ

MPI:

- Does not know about dependent operations
- Weak progress guarantess: needs polling to make progress
- Fine-grain synchronizations with `MPI_Request`

*CCLs:

- Operations are enqueued on a device stream
- Progress “automatically” once started
- Coarser-grain synchronizations through the device stream

Application interaction: execution model

Data adaptation impacts the execution too: **we need to model dependencies between operations**

Required data adaptation operation must:

1. Start *after* a kernel producing that data finishes
2. Finish *before* the associated communication operation starts

Execution models for asynchronous communication differ

MPI:

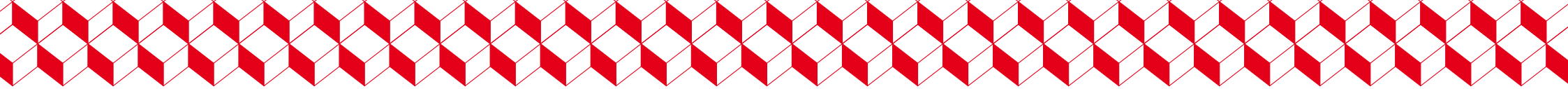
- Does not know about dependent operations
- Weak progress guarantess: needs polling to make progress
- Fine-grain synchronizations with `MPI_Request`

*CCLs:

- Operations are enqueued on a device stream
- Progress “automatically” once started
- Coarser-grain synchronizations through the device stream

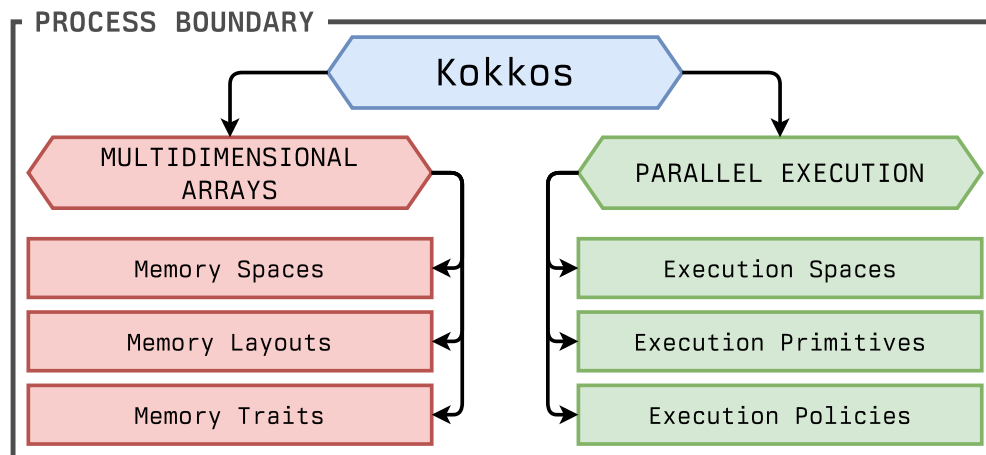
Challenge

How to design a unified interface with common execution semantics while preserving performance?



2. Context

The Kokkos performance portability ecosystem



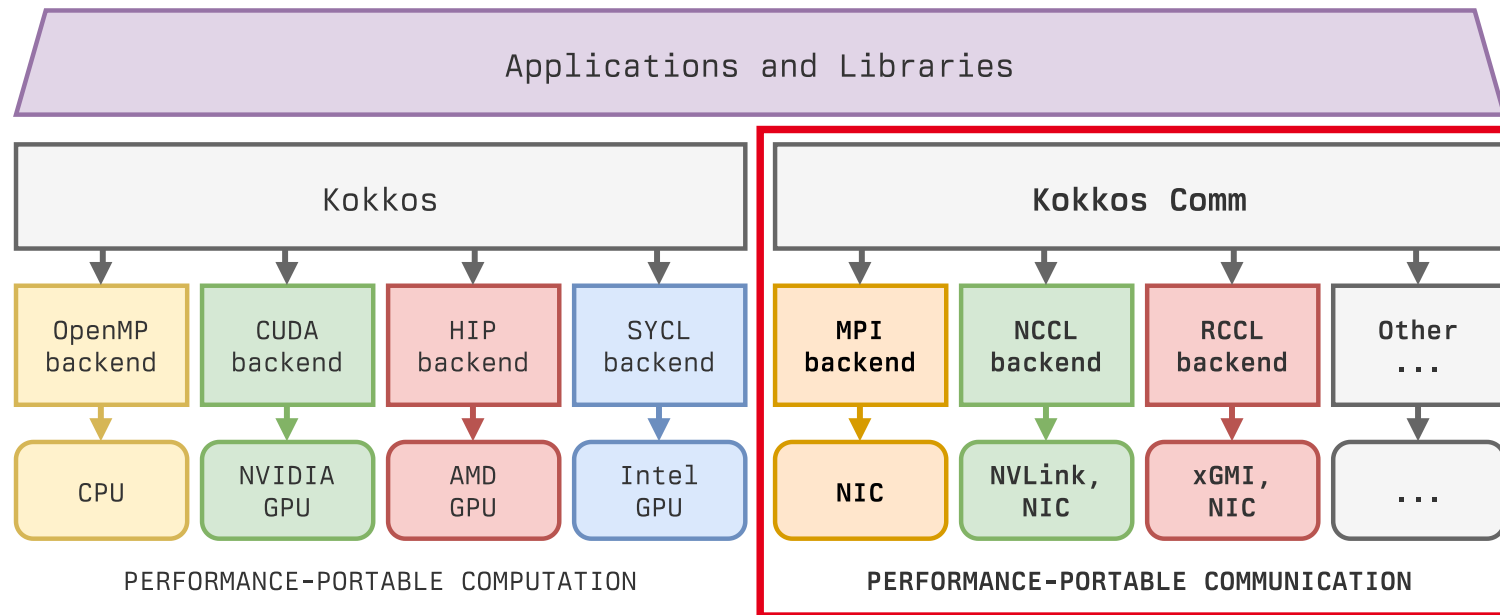
A shared-memory C++ programming model

- Views describe multidimensional data and where it lives
- Execution Spaces dispatch work to CPUs and/or GPUs
- Parallel patterns and policies describe how to run work
- Distributed communication lies outside this model

Design direction

We choose Kokkos as the target performance portability programming model for this work.
Our design bridges its `View`s, `ExecutionSpace`s, and `MemorySpace`s to multiple communication libraries.

Kokkos Comm in the Kokkos ecosystem

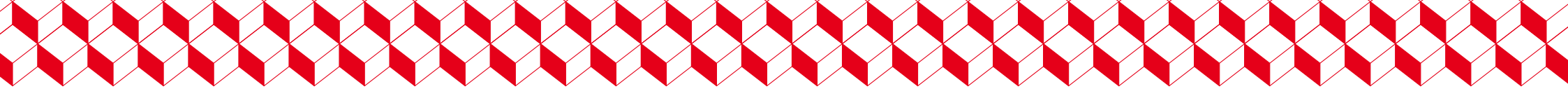


1

A unified model for communication primitives, data description, and asynchronous execution.

¹Figure extracted from [Kokkos Comm's PASC '26 paper](#).

C. Nicole Avans et al. *Kokkos Comm: Performance Portable Communication for Distributed Kokkos Applications*.



3. API Design

Design principles

Guiding API design principles for achieving performance-portable communication

- **Single source:** shared, compact API targeting all communication backends
- **Asynchronous-only:** operations return completion handles, enabling fine-grained synchronization and computation-communication overlap
- **(Near) zero overhead:** performance comparable to an equivalent handwritten implementation
- **Structural extensibility:** backends implements a set of core primitives, and may provide extensions
- **Easy to use, hard to misuse:** provide an ergonomic API that simplifies the users' life, is efficient, and rejects invalid calls as early as possible.
- **Kokkos-native:** `View` s, `ExecutionSpace` s, and `MemorySpace` s are first-class citizens.

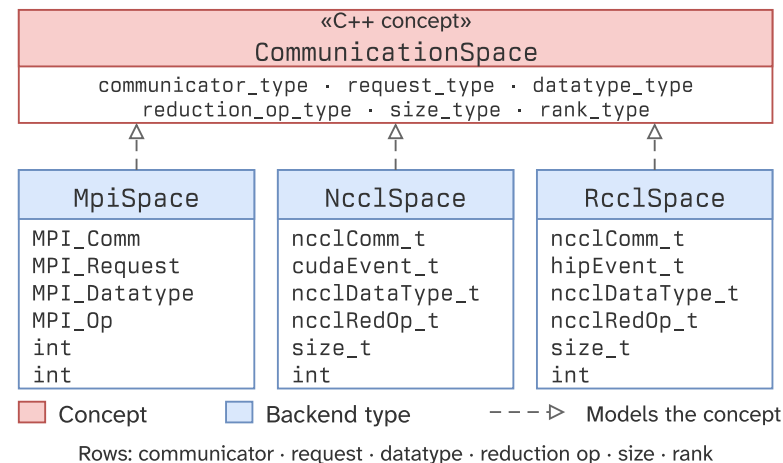
Communication Space concept

C++20 concept for compile-time selection

CommunicationSpace mirrors Kokkos' ExecutionSpace and MemorySpace

Currently supported:

- MpiSpace
- NcclSpace
- RccclSpace (experimental)



Communication Space concept

C++20 concept for compile-time selection

`CommunicationSpace` mirrors Kokkos' `ExecutionSpace` and `MemorySpace`

Currently supported:

- `MpiSpace`
- `NcclSpace`
- `RcclSpace` (experimental)

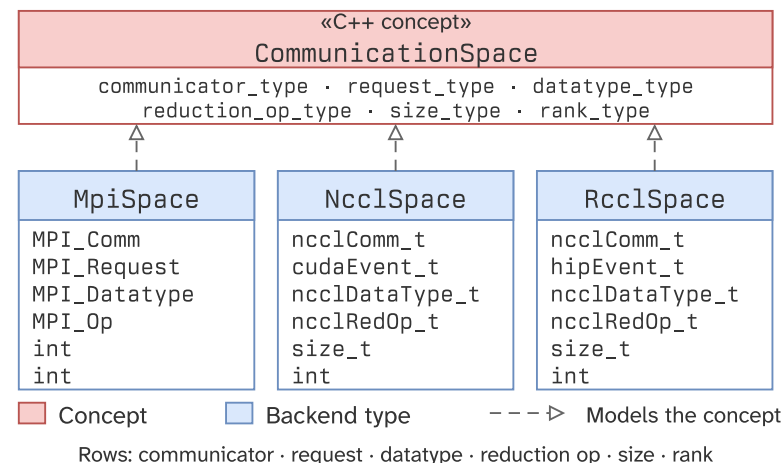
Objects are specialized per `CommunicationSpace`

```
kc::Communicator<CommSpace, ExecSpace>
```

- Couples a backend communicator with an execution space instance
- Identifies which ranks are involved in a communication, and its associated execution resource

```
kc::Request<CommSpace>
```

- Unified mechanism for communication synchronization
- Wraps a backend-specific completion handle



Unified communication interface

Core communication primitives

- Communication primitives are free functions
- All primitives are asynchronous and return a `Request`
- Supported operations resolve through compile-time tag dispatch

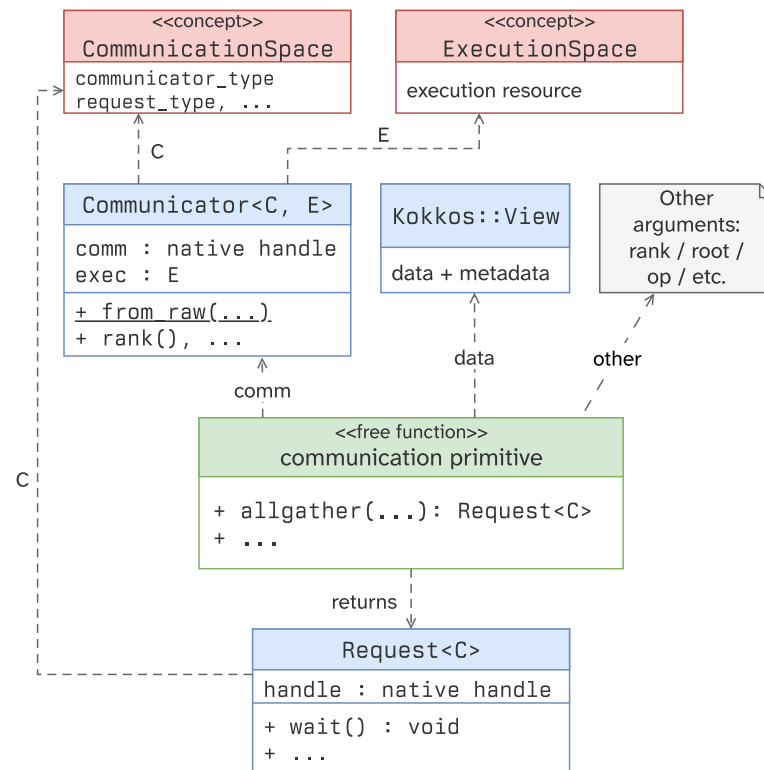
P2P

```
1 template <CommSpace C, ExecSpace E, View SendV>  
2 auto send(Communicator<C, E> comm, SendV sv, int rank)  
3   -> Request<C>
```

■ `recv`

Collectives

- `broadcast`
- `reduce`
- `allreduce`
- `allgather`
- `alltoall`

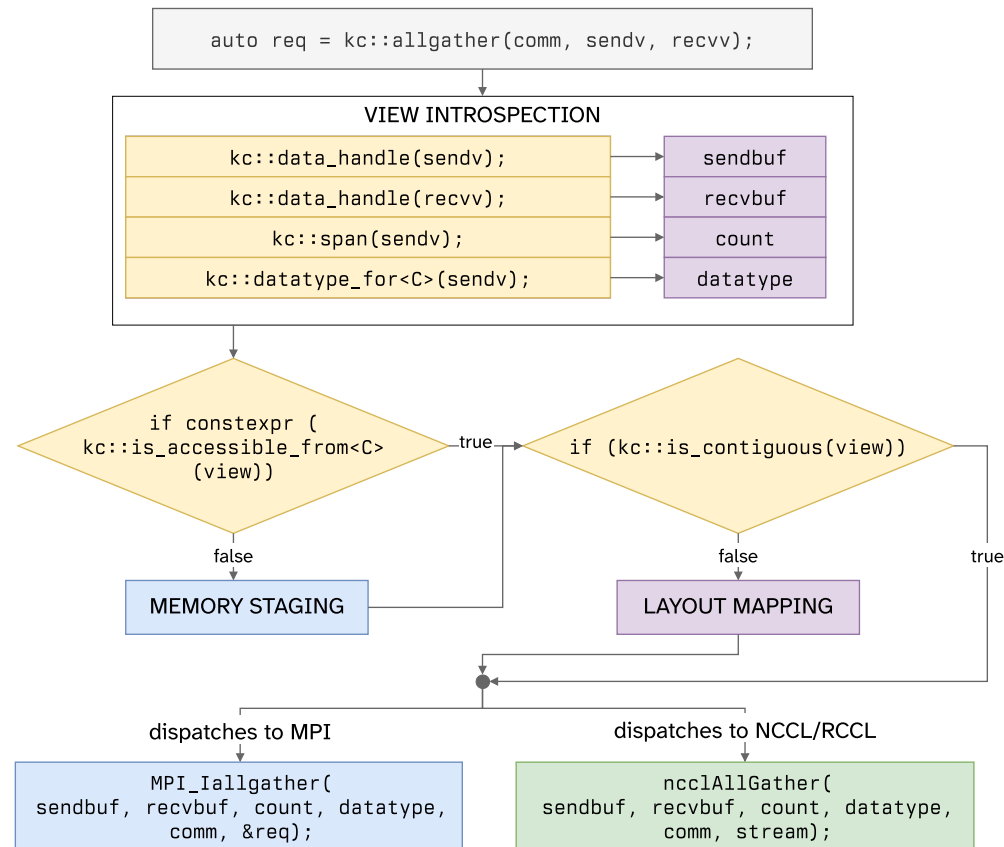


■ Concept ■ Type ■ Free functions
---> Dependency: uses / returns / requires
C: communication space · E: execution space

Kokkos-native data model

Communication primitives consume **Views**

- Infer buffer address, scalar type, and extents
- Memory staging:
 - If required, stage data in a backend-accessible memory space
 - non-GPU-aware MPI
 - host-side data for GPU-native communications
- Layout mapping:
 - If required, map non-contiguous data as a contiguous range
 - physical data packing (universal mechanism)
 - derived datatype creation (MPI-specific)
- Reject invalid calls at compile-time (e.g., type mismatch)



Kokkos-native execution model

NCCL/RCCL: order through the stream

- Kokkos CUDA/HIP `ExecutionSpace` instances wrap a device stream
- Queue-ordered semantics ensures kernels and data adaptation finish before communication executes
- No explicit synchronization: submit the work and return
- Completing a communication is synchronizing a `{cuda,hip}Event_t` associated to it

Kokkos-native execution model

NCCL/RCCL: order through the stream

- Kokkos CUDA/HIP `ExecutionSpace` instances wrap a device stream
- Queue-ordered semantics ensures kernels and data adaptation finish before communication executes
- No explicit synchronization: submit the work and return
- Completing a communication is synchronizing a `{cuda,hip}Event_t` associated to it

MPI: synchronize before initiation

- Standard MPI has no stream-awareness mechanism
- Synchronize the associated execution resource (`fence` in Kokkos terms) so kernels and data adaptation finish before calling `MPI_I*`
- Explicit synchronization: the caller blocks until all previously enqueued work is done and it can safely initiate the communication
- Completing a communication is waiting for the associated `MPI_Request`

Kokkos-native execution model

NCCL/RCCL: order through the stream

- Kokkos CUDA/HIP `ExecutionSpace` instances wrap a device stream
- Queue-ordered semantics ensures kernels and data adaptation finish before communication executes
- No explicit synchronization: submit the work and return
- Completing a communication is synchronizing a `{cuda,hip}Event_t` associated to it

MPI: synchronize before initiation

- Standard MPI has no stream-awareness mechanism
- Synchronize the associated execution resource (`fence` in Kokkos terms) so kernels and data adaptation finish before calling `MPI_I*`
- Explicit synchronization: the caller blocks until all previously enqueued work is done and it can safely initiate the communication
- Completing a communication is waiting for the associated `MPI_Request`

Same interface and ordering contract.

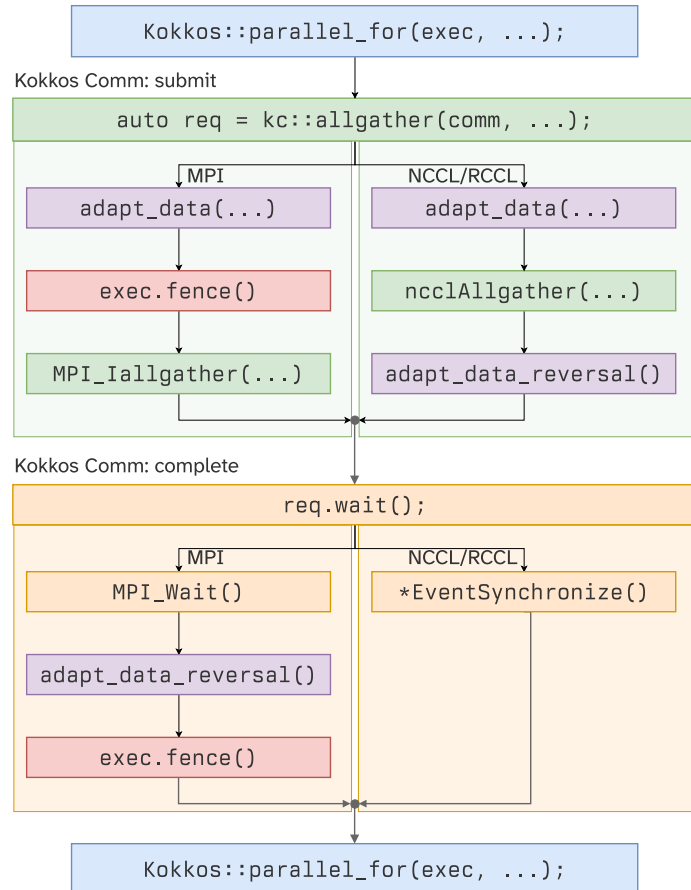
Operations follow the same semantic ordering.

Upon return, the data cannot be reused until communication completion.

Callers control where to wait for the result to be available.

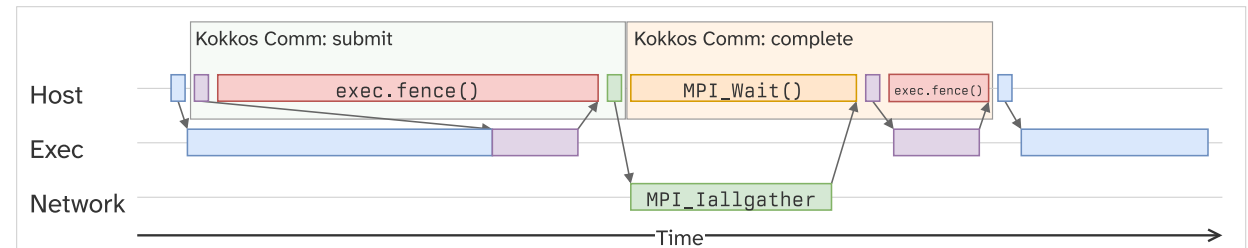
Ordering communication and computation

Host control flow

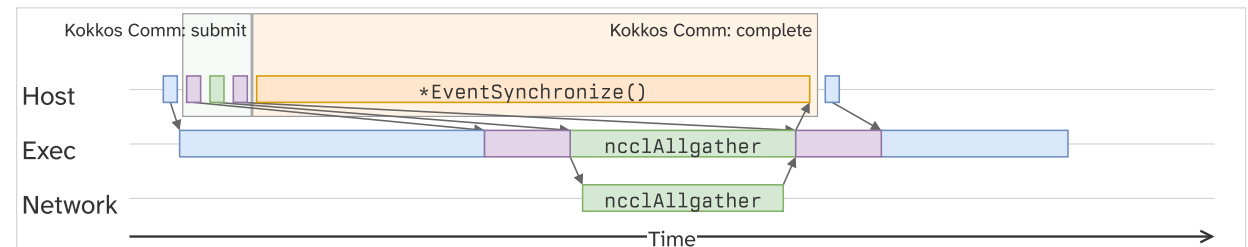


Execution traces

MPI



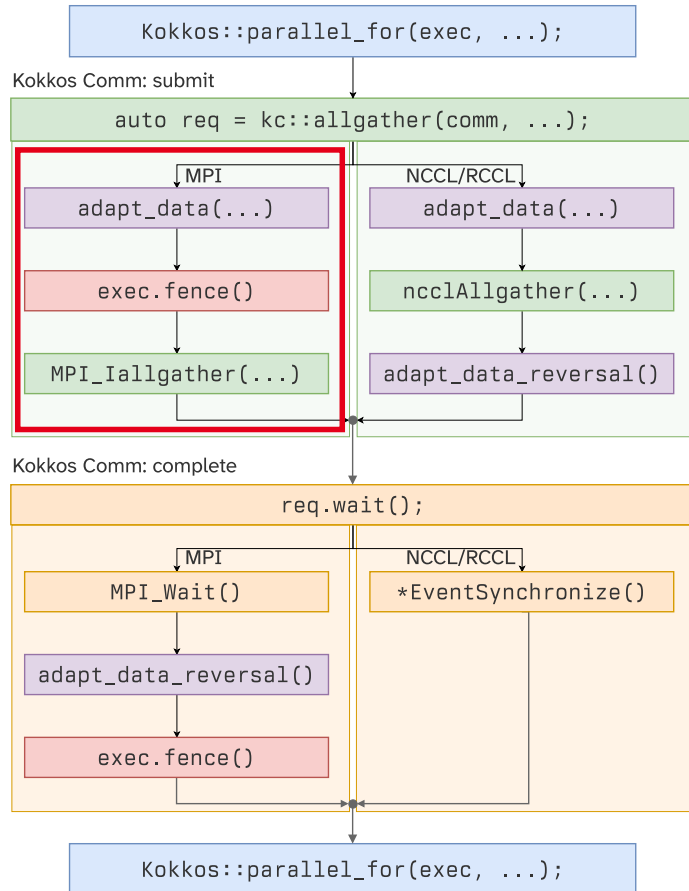
NCCL/RCCL



■ Kernel ■ Communication ■ Data adaptation ■ Exec Synchronization ■ Comm. wait

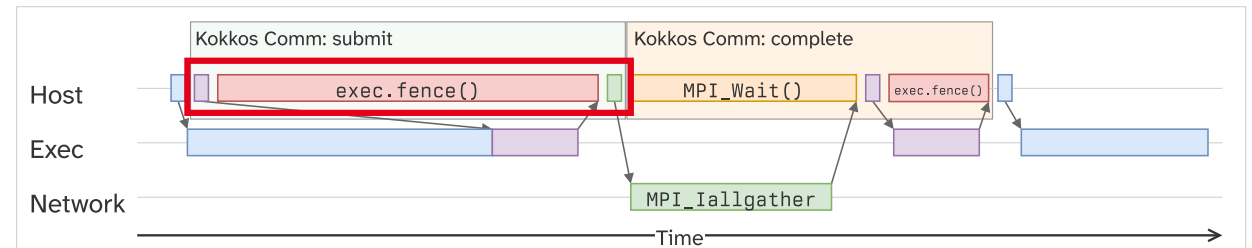
Ordering communication and computation

Host control flow

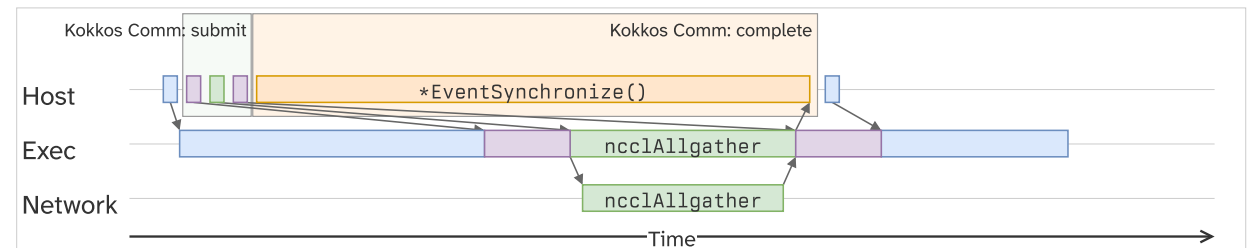


Execution traces

MPI



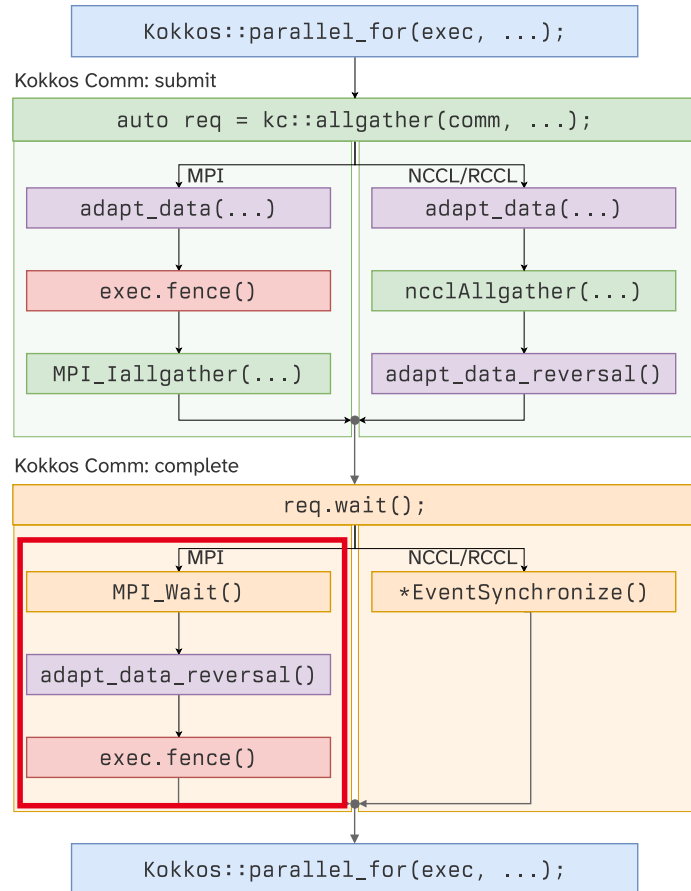
NCCL/RCCl



■ Kernel
 ■ Communication
 ■ Data adaptation
 ■ Exec Synchronization
 ■ Comm. wait

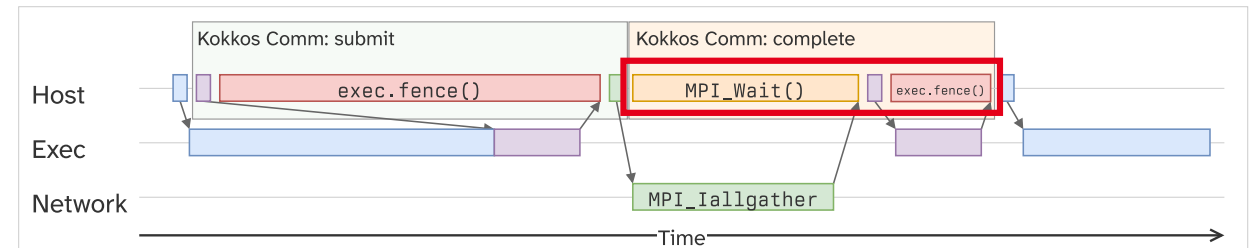
Ordering communication and computation

Host control flow

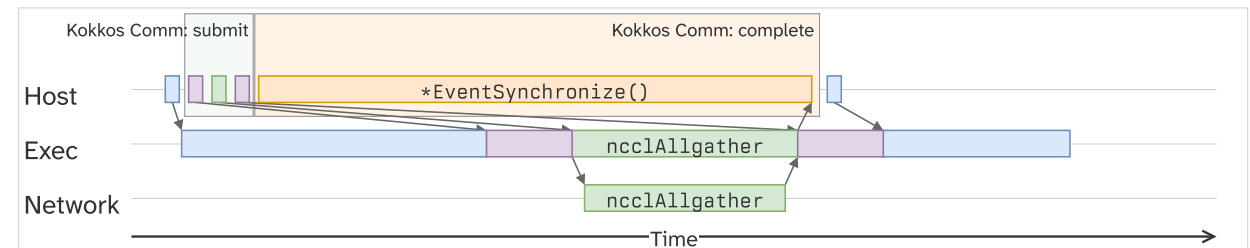


Execution traces

MPI



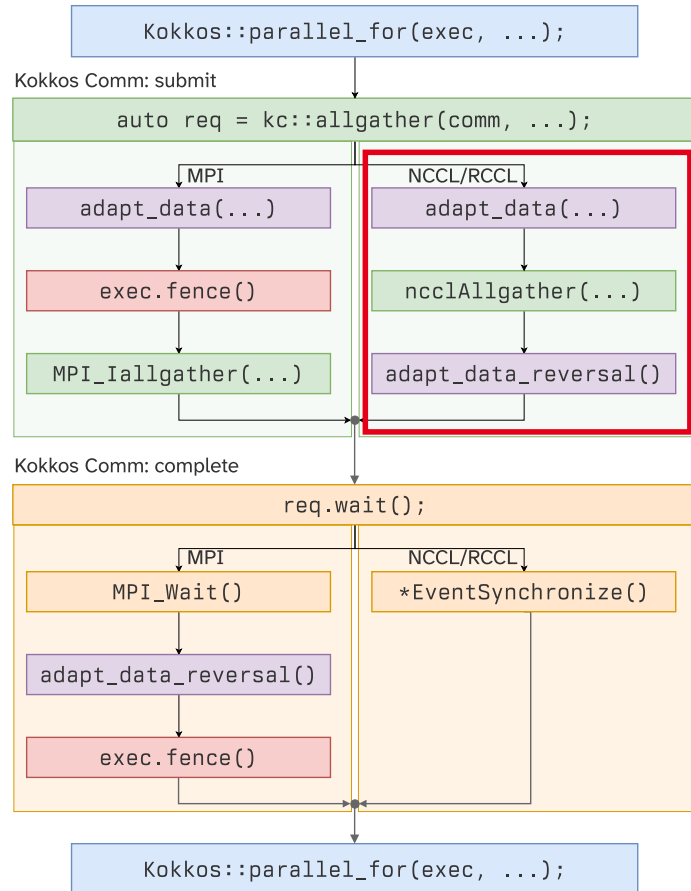
NCCL/RCCl



■ Kernel
 ■ Communication
 ■ Data adaptation
 ■ Exec Synchronization
 ■ Comm. wait

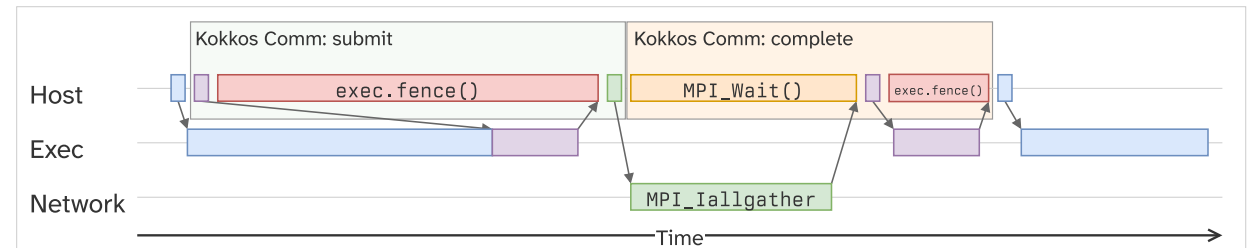
Ordering communication and computation

Host control flow

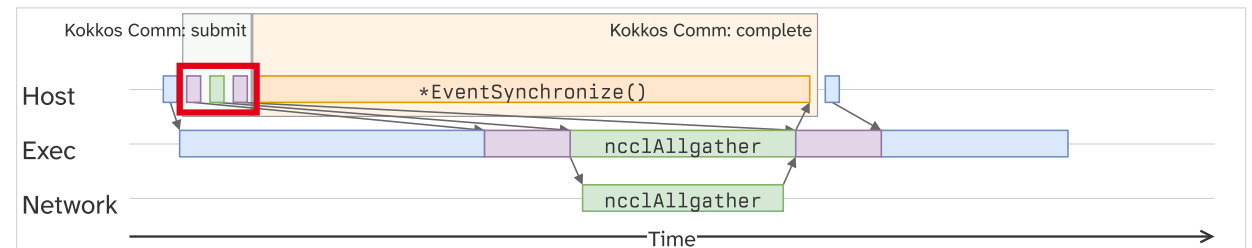


Execution traces

MPI



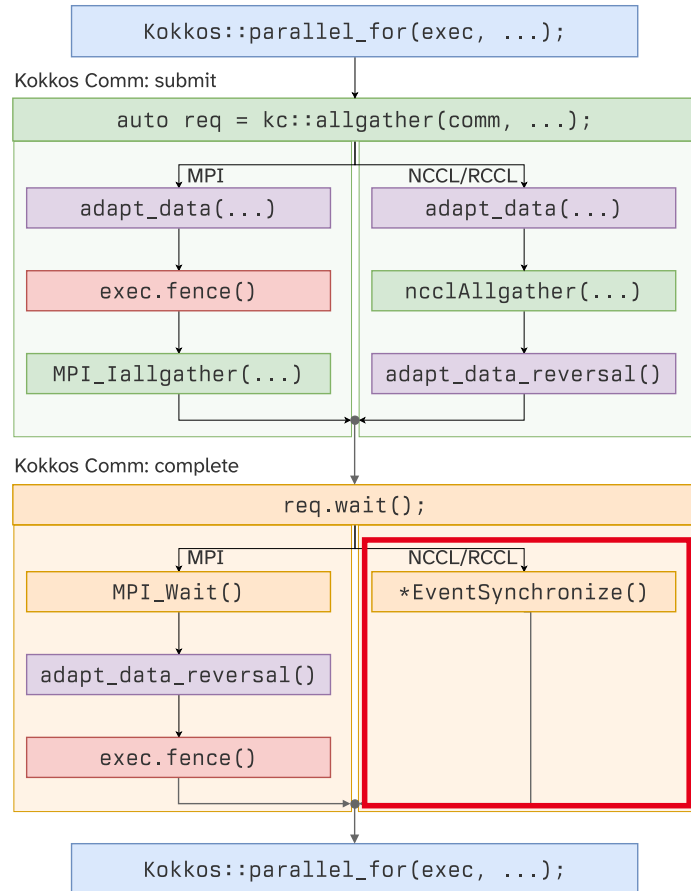
NCCL/RCCL



■ Kernel ■ Communication ■ Data adaptation ■ Exec Synchronization ■ Comm. wait

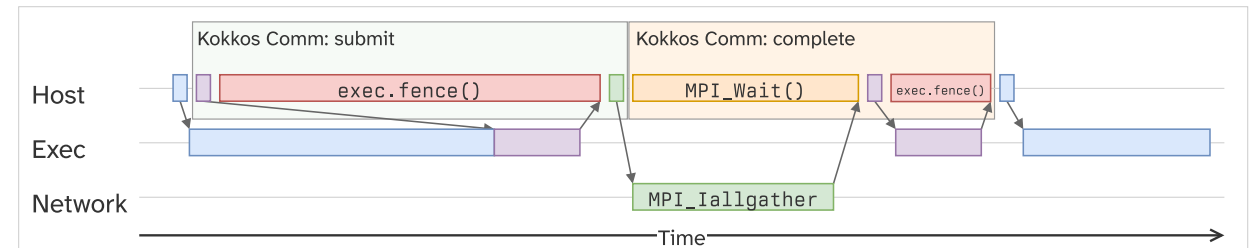
Ordering communication and computation

Host control flow

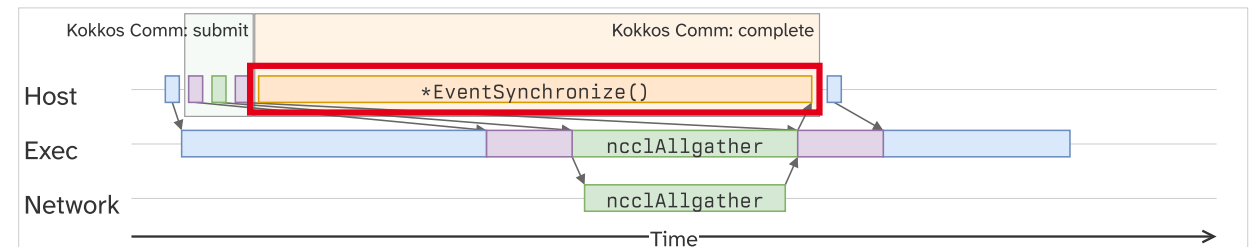


Execution traces

MPI



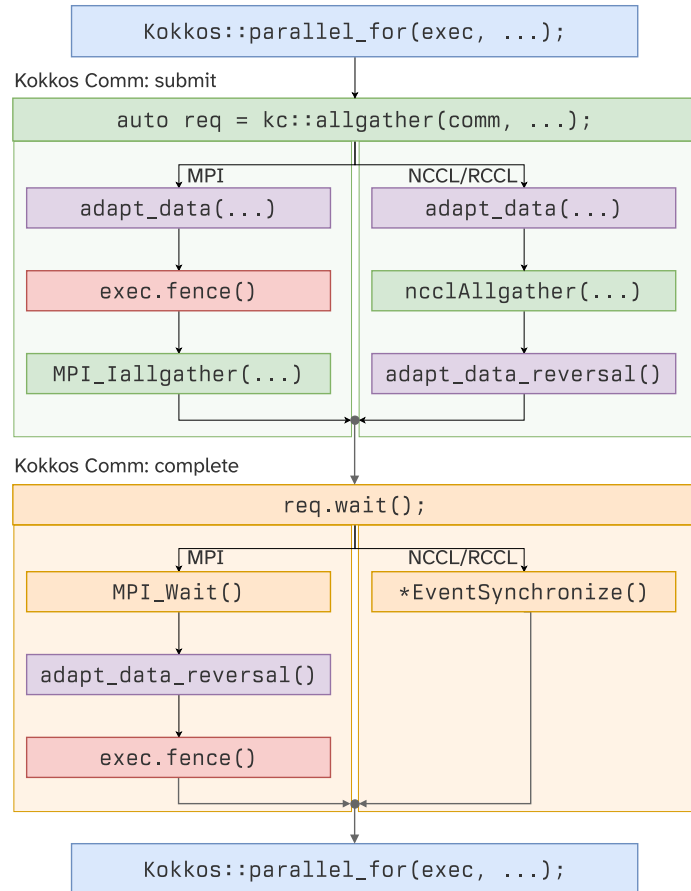
NCCL/RCCl



■ Kernel ■ Communication ■ Data adaptation ■ Exec Synchronization ■ Comm. wait

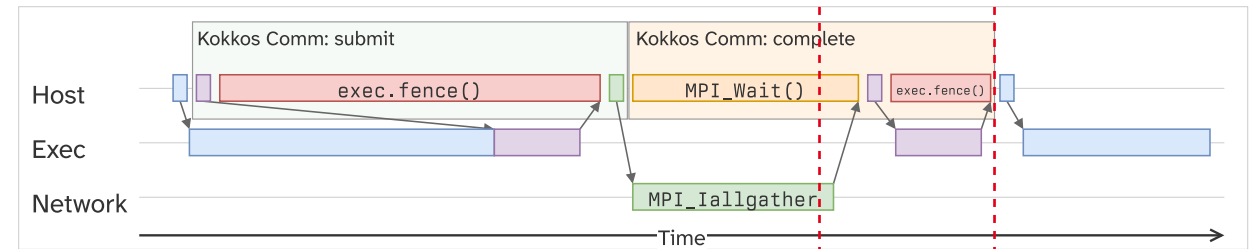
Ordering communication and computation

Host control flow

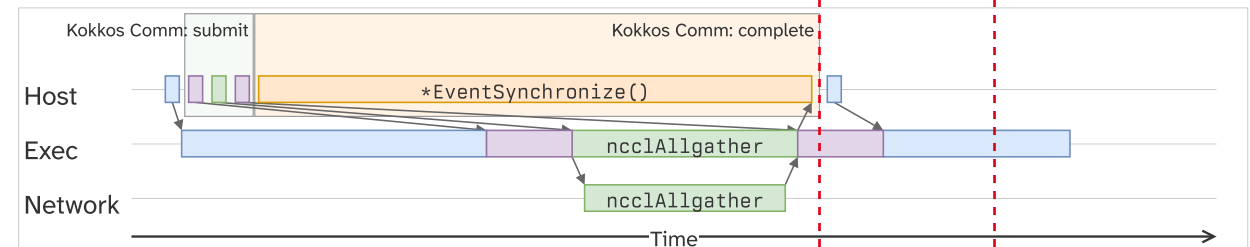


Execution traces

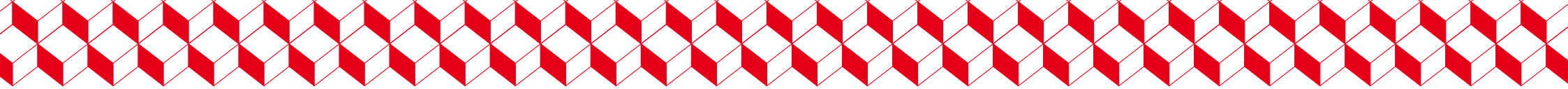
MPI



NCCL/RCCL



■ Kernel ■ Communication ■ Data adaptation ■ Exec Synchronization ■ Comm. wait



4. Evaluation

Performance evaluation

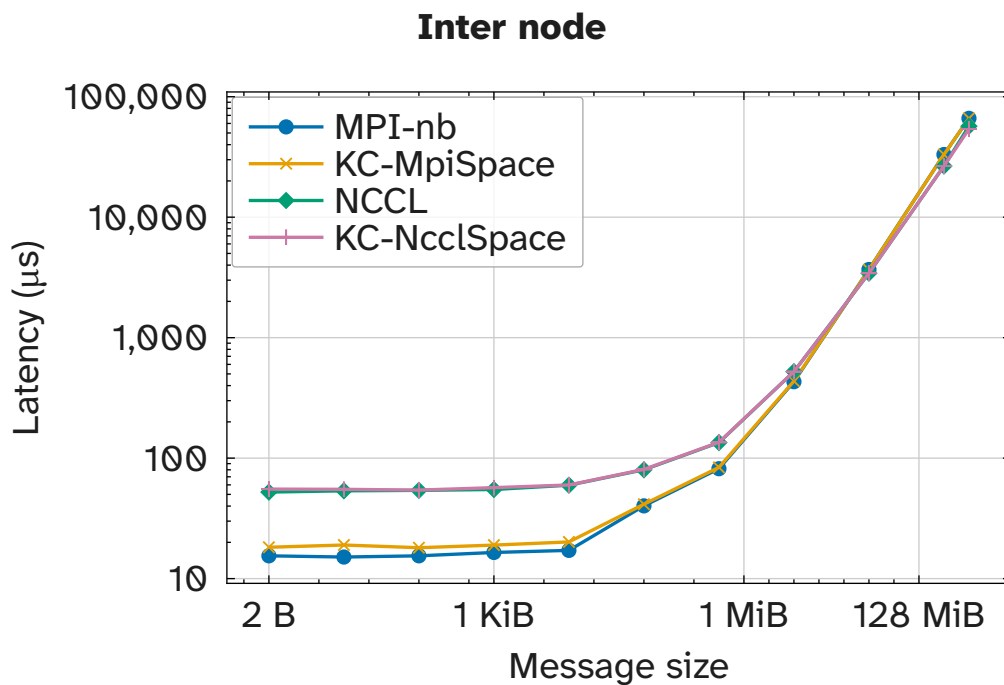
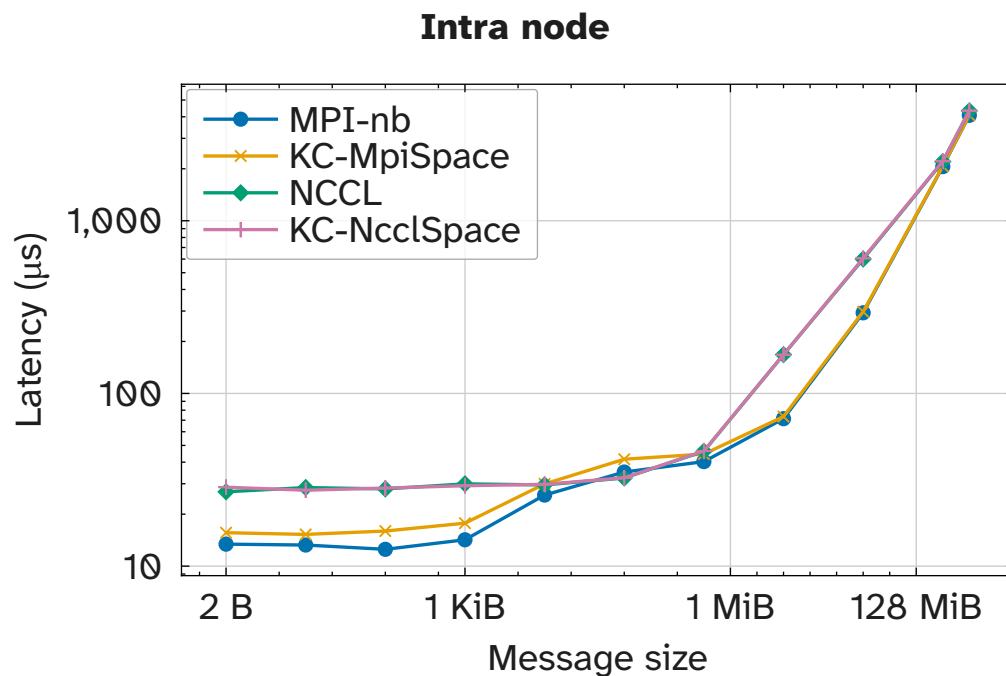
Experimental setup

- Hardware: 4 × NVIDIA GH200 nodes
- Software: Kokkos 5.2.2, CUDA 13.0, Open MPI 5.0.8, NCCL 2.28.7
 - Views are stored on the device

Microbenchmarks: Kokkos Comm versus direct MPI/NCCL calls on the same operation

- P2P latency
- Collectives bandwidth

Point-to-point latency

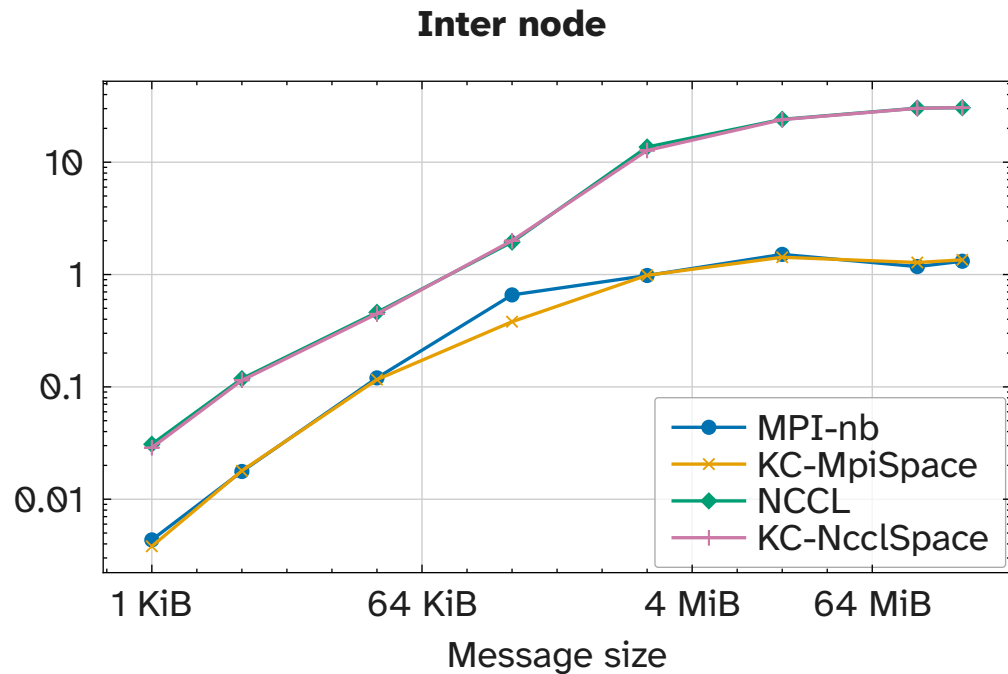
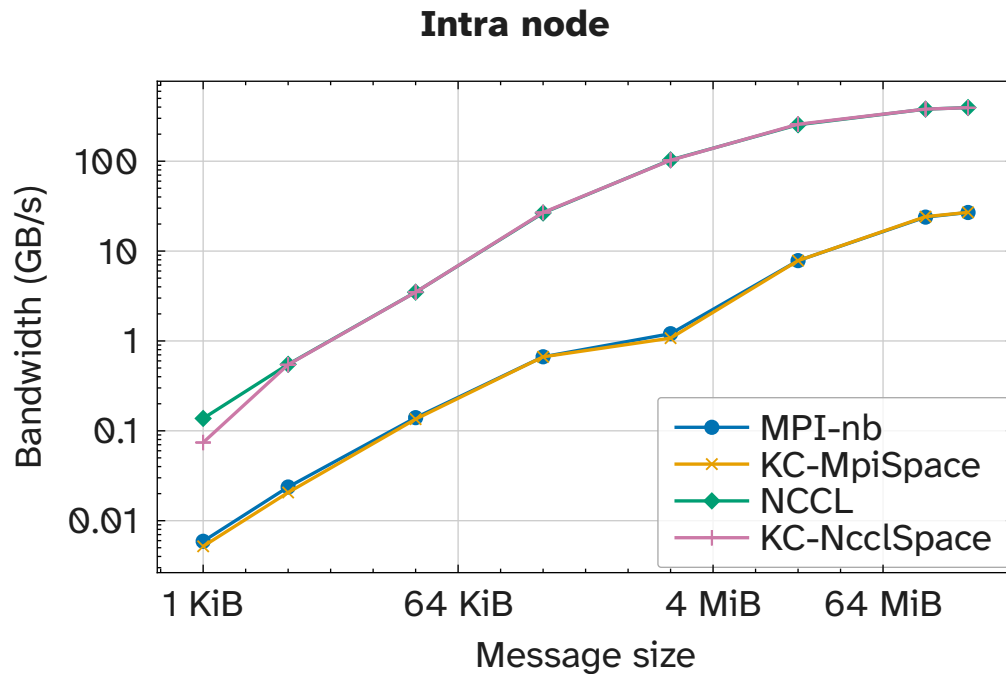


Log scales: x = base 2; y = base 10 · **Lower is better**

Takeaway:

- On small messages `KC-MpiSpace` pays the execution space fence overhead (roughly 3–4 μs);
- `KC-NcclSpace` closely tracks direct NCCL.

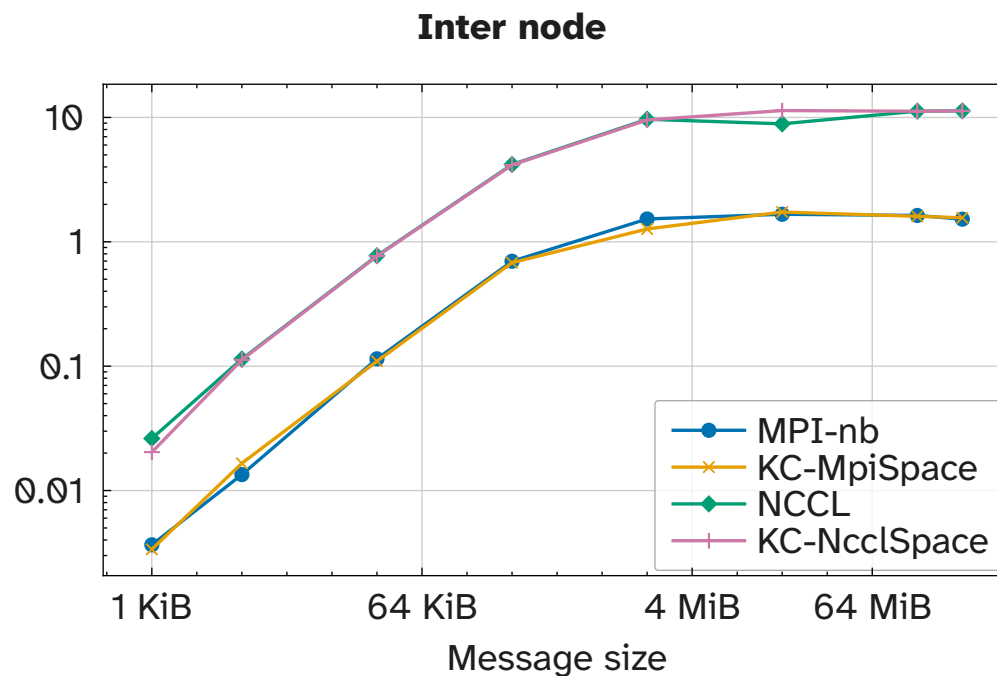
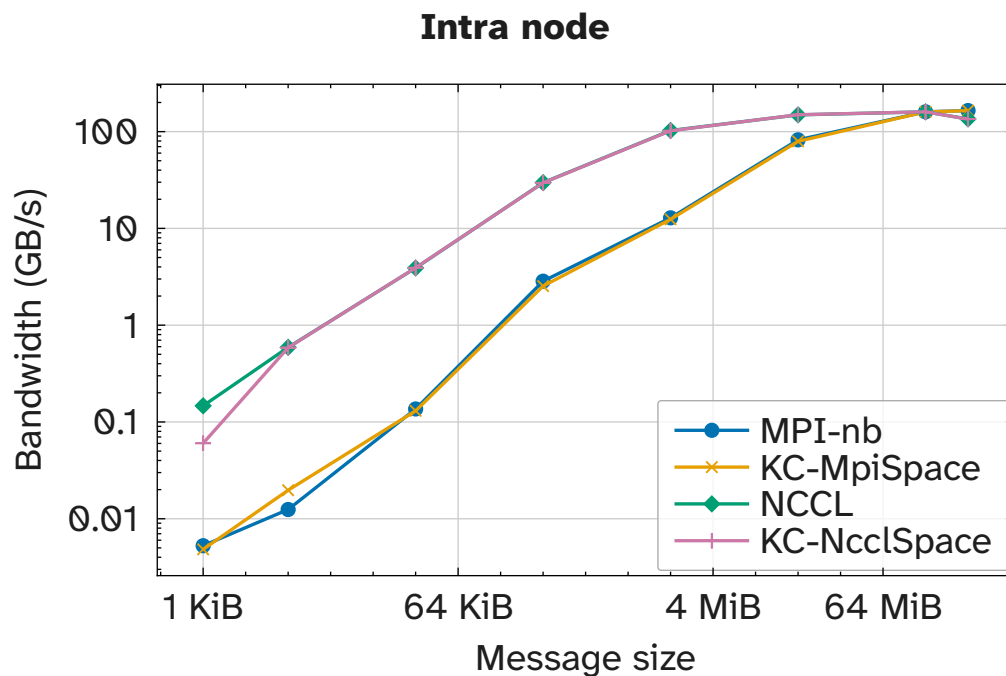
All-Gather bandwidth



Log scales: x = base 2; y = base 10 · **Higher is better**

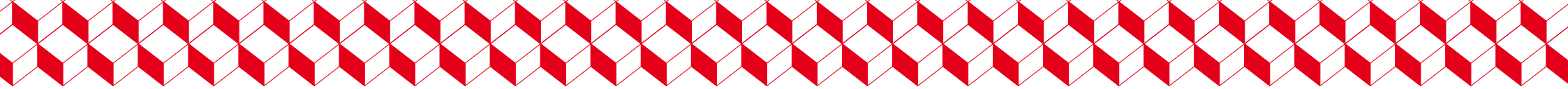
Takeaway: within each backend, Kokkos Comm follows the direct implementation over the message range.

All-to-All bandwidth



Log scales: x = base 2; y = base 10 · **Higher is better**

Takeaway: within each backend, Kokkos Comm follows the direct implementation over the message range.



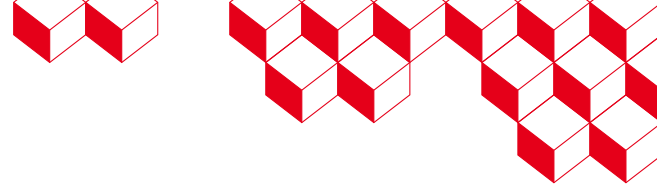
5. Conclusion

Write once, communicate anywhere

How to design an unified interface that enables HPC applications to leverage CCLs alongside MPI?

- `CommunicationSpace` concept enables single-source communication code for MPI and NCCL/RCCL
- Native integration with the Kokkos ecosystem provides unified data and execution models

Write once, communicate anywhere



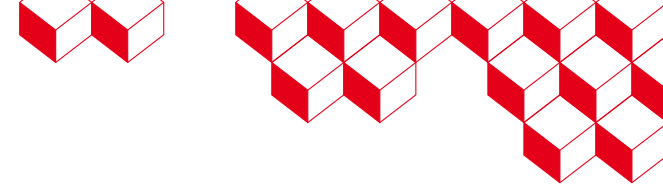
How to design an unified interface that enables HPC applications to leverage CCLs alongside MPI?

- `CommunicationSpace` concept enables single-source communication code for MPI and NCCL/RCCL
- Native integration with the Kokkos ecosystem provides unified data and execution models

How to design a unified interface that consumes and adapts structured data while preserving performance?

- A compact, ergonomic API: `View` introspection infers buffer's metadata; compile-time checks catch mistakes early.
- Automatic memory staging and layout mapping adapt data to backend capabilities and non-contiguous layouts.

Write once, communicate anywhere



How to design an unified interface that enables HPC applications to leverage CCLs alongside MPI?

- `CommunicationSpace` concept enables single-source communication code for MPI and NCCL/RCCL
- Native integration with the Kokkos ecosystem provides unified data and execution models

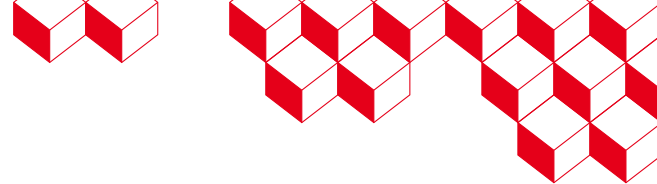
How to design a unified interface that consumes and adapts structured data while preserving performance?

- A compact, ergonomic API: `View` introspection infers buffer's metadata; compile-time checks catch mistakes early.
- Automatic memory staging and layout mapping adapt data to backend capabilities and non-contiguous layouts.

How to design a unified interface with common execution semantics while preserving performance?

- Communicators bind communication and execution resources together
- Asynchronous-only calls enable computation-communication overlap for performance
- Same interface, ordering guarantees, and a unified completion model

Perspectives



Current and future goals of Kokkos Comm

- Integration of stream-aware and GPU-initiated networking
 - Stream-triggered MPI backend in the works at Tennessee Tech
 - Explore NCCL's GIN API as a potential future backend
- API support for registering user-managed buffers and/or memory pools
 - Avoid costly and hidden allocations when data adaptation is needed
 - Reuse already allocated memory to perform these operations
- Composable, asynchronous execution through a `std::execution`-based interface
 - Explore integration with [uliegecsm/kokkos-execution](https://github.com/uliegecsm/kokkos-execution)
- Portable initialization and finalization across MPI and *CCLs backends
 - Remove the need for any backend-specific code in Kokkos Comm applications

Get involved with Kokkos Comm

Discuss the API, share feedback and contribute through GitHub, Slack, or the developer meetings!

Slack

<https://kokkosteam.slack.com/>

#mpi-interop channel

GitHub

Repository

<https://github.com/kokkos/kokkos-comm>

Discussions

<https://github.com/kokkos/kokkos-comm/discussions>

Issues

<https://github.com/kokkos/kokkos-comm/issues>

Developer Meeting

<https://zoom-lfx.platform.linuxfoundation.org/meeting/95426536306?password=fd73274e-8949-4c61-baa5-fe6e1b71e18f>

Bi-weekly on Thursdays

07:30–08:00 PST (UTC–9)

10:30–11:00 EST (UTC–6)

16:30–17:00 CET (UTC+1)

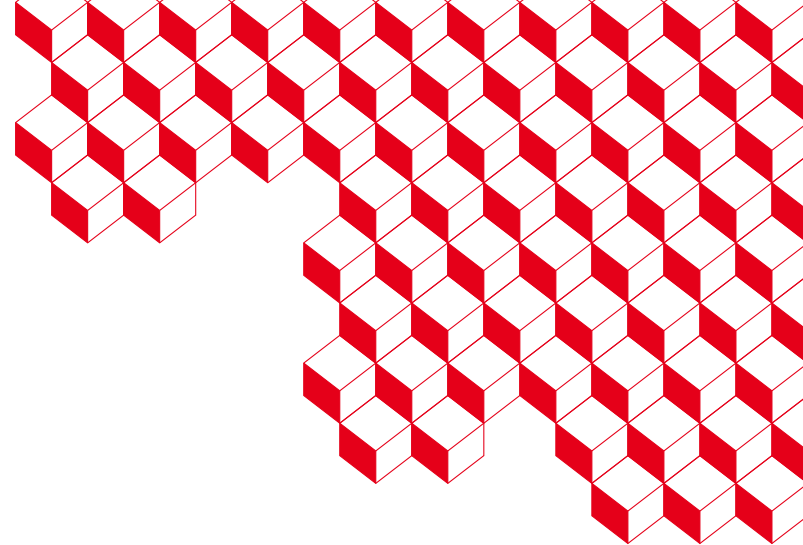
23:30–00:00 JST (UTC+8)

kokkos-comm Public

Experimental Explicit Communications API for Kokkos

C++ ☆ 28 🔗 13 🔄 20 👤 9 Updated 5 hours ago





Thank you for your attention

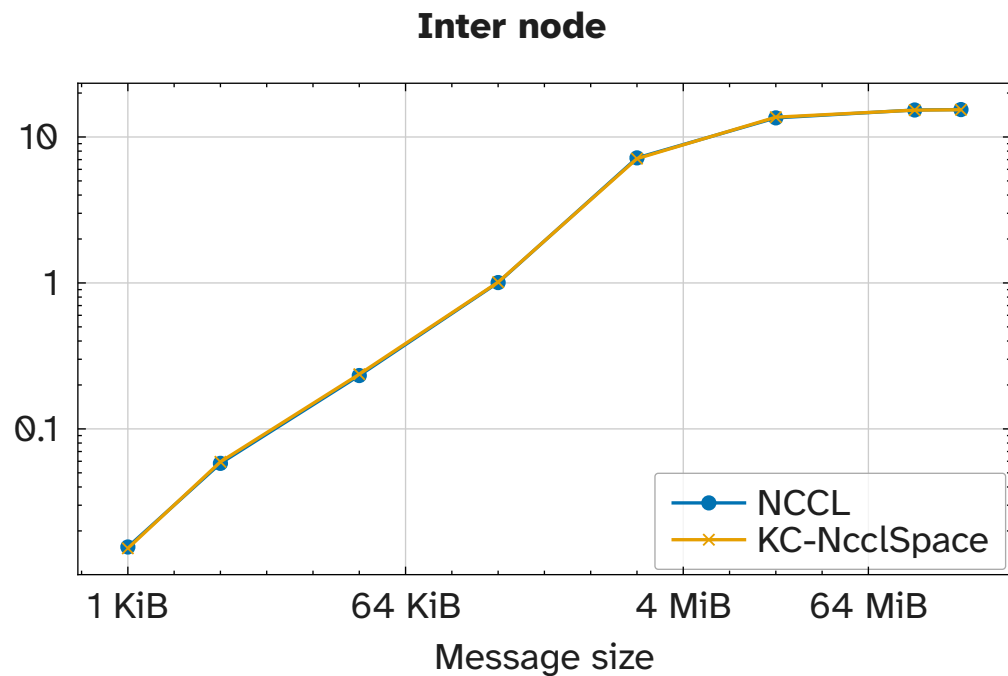
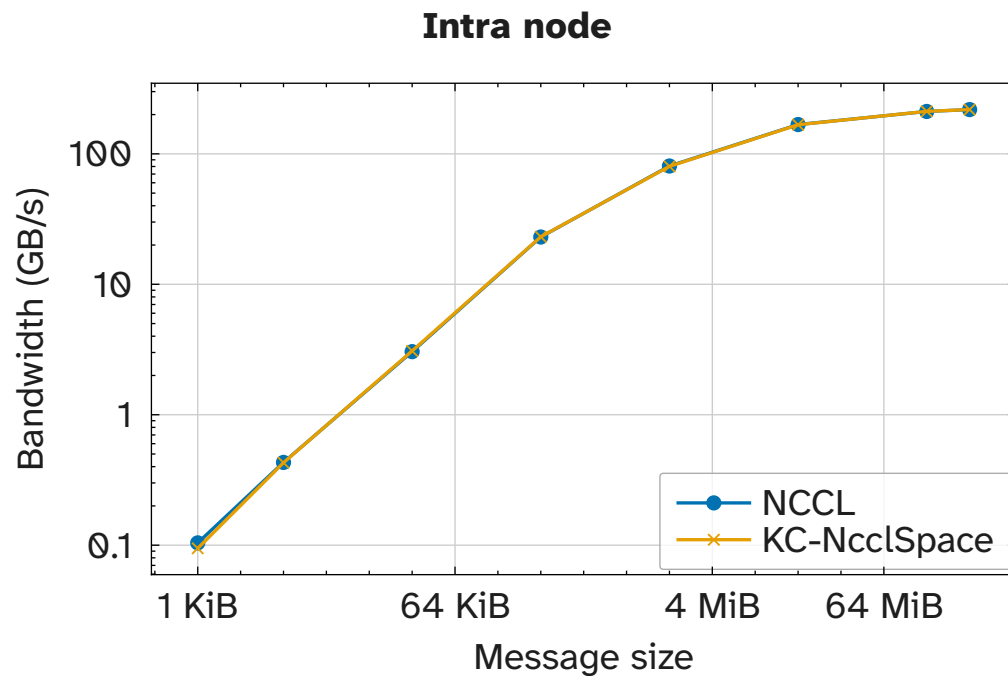
Questions?

CEA, DAM, DIF, F-91297 Arpajon, France

Gabriel Dos Santos

gabriel.dossantos@cea.fr

All-Reduce bandwidth



Log scales: $x = \text{base } 2$; $y = \text{base } 10$ · **Higher is better**

Note: System's Open MPI does not support nonblocking reductions with device buffers.

Dense power iteration method

Given a diagonalizable matrix A of order N , the algorithm will produce a number λ , which is the greatest eigenvalue of A , and a nonzero vector v , which is a corresponding eigenvector of λ , that is, $Av = \lambda v$

The method is described by the recurrence relation:

$$b_{k+1} = \frac{Ab_k}{\|Ab_k\|}$$

Power iteration

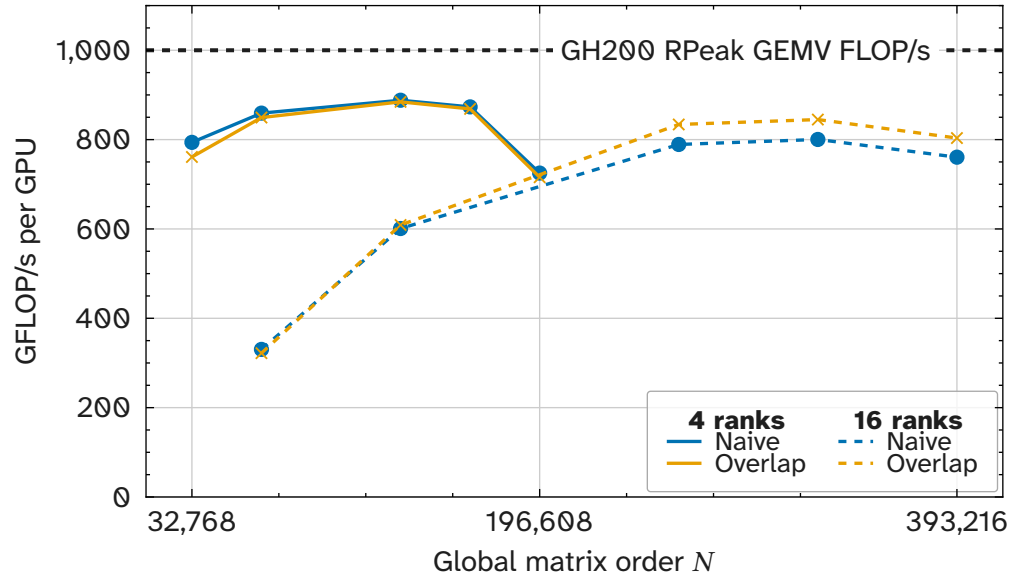
```
1  $b_0 \leftarrow$  random nonzero vector of length  $N$ 
2  $b_0 \leftarrow \frac{b_0}{\|b_0\|}$ 
3 for  $k = 0, \dots, K - 1$  do
4    $b_{k+1} \leftarrow Ab_k$ 
5    $\rho \leftarrow \|b_{k+1}\|$ 
6    $b_{k+1} \leftarrow \frac{b_{k+1}}{\rho}$ 
7 end for
8 return  $b_K$  (approximate dominant
   eigenvector)
```

We compare naive 1D partitioning with the overlap variant.

Power iteration: performance and scaling

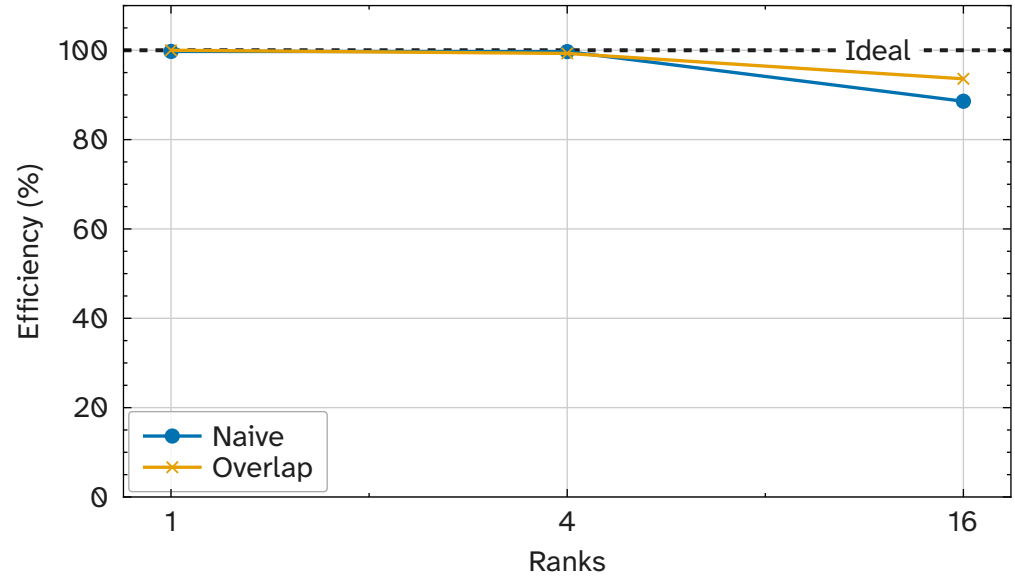


Performance — 4 and 16 ranks



Overlap gains 5.7% at 16 ranks and $N = 262,144$.

Weak scaling — $N_0 = 65,536$



Normalized to the best single-GPU result among naive and overlap.

Higher is better