



Making MPI Collectives Pluggable

Carsten Clauss, Sonja Happ, and Simon Pickartz
EuroMPI 2026, Vienna, October 08, 2026

Overview

Talk seeks to answer three questions:

1. Could it make sense for users to plug their own collectives into an MPI library?
2. What could the interface for such a registration of user collectives look like?
3. Could this interface also be used to integrate other libraries for collective communication?



Motivation

- MPI provides applications with a standardized interface for collectives.
- However, the implementation of the collective communication patterns is:
 - Not standardized and implementation-specific.
 - Closely coupled either to the respective MPI library.
 - Or up to an external backend, e.g., like HCOLL or UCC.

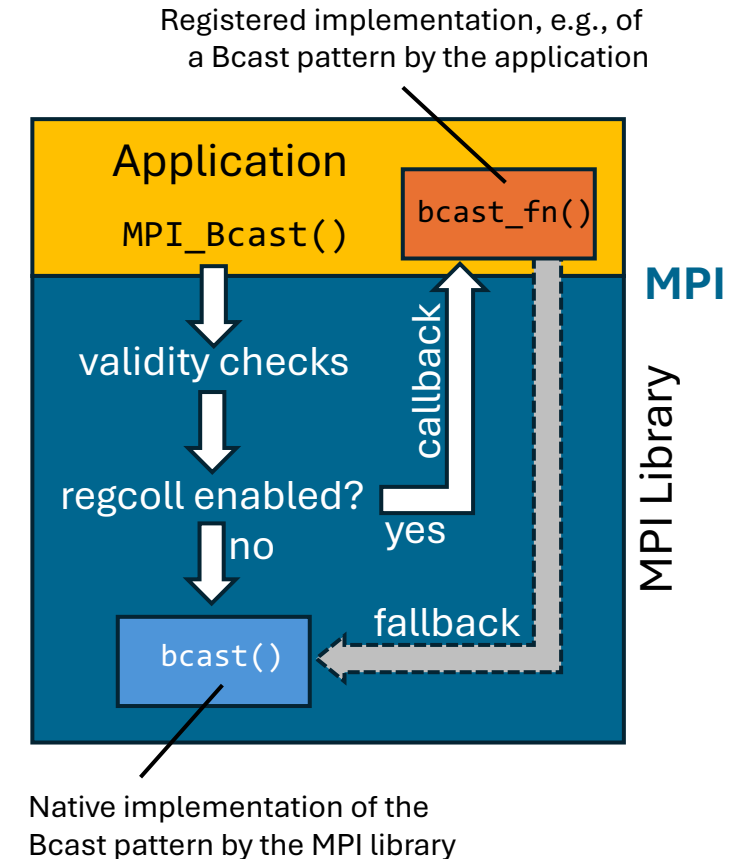
→ Applications cannot transparently integrate their own collective algorithms.

- Moreover, also external backends require a dedicated integration.
 - Typically, for each backend API, a wrapper must be written.
 - And each of these wrappers is tailored to one specific MPI library.

→ A public and portable extension interface for MPI collectives is missing.

Idea and Architecture

- Application registers its implementation of a collective.
(→ "regcoll")
- The application continues to use standard MPI calls.
- Upon a call of a collective, MPI first does the usual checks:
 - General parameters like buffers and lengths valid?
 - MPI communicator, datatypes and operation valid?
- Then MPI checks if a regcoll is enabled for the communicator:
 - If no regcoll is enabled, execute the native MPI algorithm.
 - If a regcoll is enabled, call the registered algorithm.
 - If support is lacking, the regcoll can hand back to the MPI library.



MPI Interface Extension

MPIX_Register_collaps(**name**, **collops**, **collapse_dtypes**, **algorithm_fn**,
comm_init_fn, **comm_free_fn**, **deregister_fn**,
info, **extra_state**)

- **name** regcoll identifier, used to reference and activate it via Info objects or environment variables (string)
- **collops** bit vector (integer) of constants as MPIX_COLLAPS_BARRIER, _BCAST etc. that can be combined via a logical OR (integer)
- **collapse_dtypes** flag indicating if the MPI library should try to collapse the given MPI datatype into a basic contiguous one (integer)
- **algorithm_fn** function implementing the collective patterns supported by this regcoll (callback)
- **comm_init_fn** function invoked if a new communicator is created by the MPI library (callback)
- **comm_free_fn** function invoked if a communicator is freed by the MPI library (callback)
- **deregister_fn** function invoked when all resources are released at the end of the MPI runtime (callback)
- **info** optional MPI info parameter (handle)
- **extra_state** extra state for regcoll

→ Similar to existing MPI concepts like `MPI_Register_datarep()` or `MPI_Op_create()`.

Central Callback

```
int (MPIX_Collops_algorithm_function)(collop, // enum specifying the collective  
                                     sbuf, scounts[], sdispls[], sdatatype,  
                                     rbuf, rcounts[], rdispls[], rdatatype,  
                                     op, root, comm, extra_comm_state)
```

Activation and execution

- Callback is not executed by default:
 - After registration, it must first be enabled/activated for the given communicator.
 - Activation via MPI Info hints or env. variables.
- Additionally, regcoll can decide if execution:
 - ...is reasonable, e.g., based on communicator size.
 - ...is possible, depending on reduce op and dtype.

Possible return scenarios:

- **MPI_SUCCESS**: The regcoll algorithm has completed the operation successfully.
- **MPIX_ERR_FALLBACK**: The MPI library shall take over and execute the native algorithm instead.
- Other errors: Invoke the standard MPI error handling.

→ Fallback decision must be consistent!

Use-case Example

(see Section 6.9, Global Reduction Operations, in MPI 5.0)

- *Advice to users:* Floating-point operations and some user-ops are non-associative!
 - Standard reductions yield different results depending on the reduction tree.
 - Applications that cannot ignore this non-associative nature, should enforce the order.
- Possible approach:
 1. Gather all operands at root (`MPI_Gather()`)
 2. Locally reduce all at root in the desired order (`MPI_Reduce_local()` or via a custom loop)

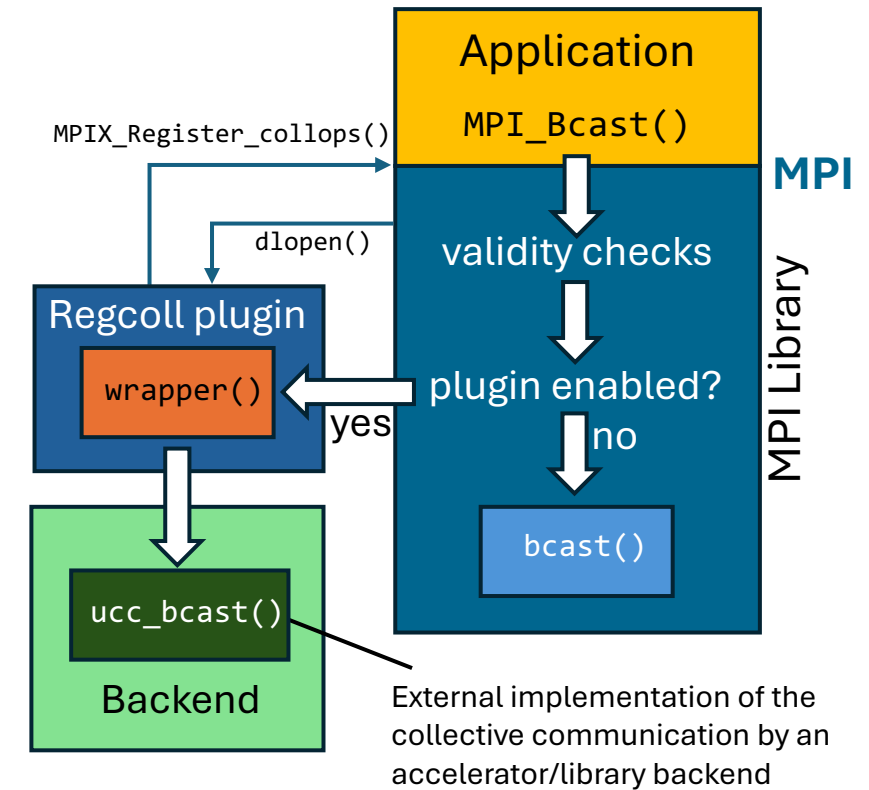
```
MPIX_Register_collaps("my_reduce",  
                      MPIX_COLLOPS_REDUCE, ...  
                      my_reduce_strict_order, ...);  
MPI_Info_set(info_coll, "collops", "my_reduce");  
MPI_Comm_set_info(MPI_COMM_WORLD, info_coll);  
...  
MPI_Reduce(..., MPI_COMM_WORLD);  
...  
↓ Invoke the algorithm_fn() callback internally  
int my_reduce_strict_order(int collop, ...) {  
    assert(collop == MPI_COLLORS_REDUCE);  
    ...  
    MPI_Gather(..., root, extra_comm_state->comm);  
    if (extra_comm_state->my_rank == root) {  
        MPI_Reduce_local(...);  
    }  
    return MPI_SUCCESS;  
}
```

Support for Plugins and Backends

**Application-level collectives may sound great...
...but can the interface do even more?**

- Idea: The regcolls do not necessarily have to be registered by the application.
- Instead, they can also be registered by external libraries or even by the MPI library itself.
- Advantage: It would still have a standardized API.
 - Implicit registration and communicator activation could be completely transparent to applications.
 - Implemented regcolls would still be portable and interchangeable between different MPI libraries.

→ That way, integrating external backends is also possible!



Datatype Collapsing (1)

The Problem

- External backends often support only contiguous buffers of basic data types.
- In a collective operation, different processes can use different type maps, e.g.:
 - Root sends a contiguous buffer of a single basic datatype (say `MPI_INT`).
 - Receivers use a derived data type with gaps between the same basic datatype (i.e., `MPI_INT`).
 - Using these different datatypes is permitted since the type signatures match.

Datatype at Sender

(e.g., root of a Bcast)

`n x MPI_INT`



Basic datatype?

Yes! → No fallback!

`ucc_bcast()`

Datatype at Receivers

`MPI_Type_vector(count=n, len=1, stride=2, MPI_INT)`



Basic datatype?

No! → Fallback!

`mpi_bcast()`



→ A naïve local data type check (basic vs. derived) leads to inconsistent fallback decisions!

Datatype Collapsing (2)

The Solution

- The MPI library checks, if a given (derived) datatype is based on a single basic one.
- If this is the case:
 - The MPI library collapses the datatype.
 - Provides temporary (contiguous) buffers for the callback and thus the backend.
 - Packs/unpacks the data transparently.
- In all other cases, a fallback is conducted (and is guaranteed to be consistent).

→ Enabled via `collapse_dtype` option of `MPID_Register_collaps()`.

Datatype at Sender

(e.g., root of a Bcast)

$n \times \text{MPI_INT}$



Basic datatype?

Yes! → No fallback!

`ucc_bcast()`

Datatype at Receivers

`MPI_Type_vector(count=n, len=1, stride=2, MPI_INT)`



Based on a single basic datatype?

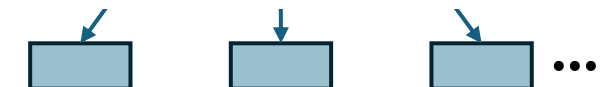
Yes! (`MPI_INT`)

→ Receive into temp buffer:

`ucc_bcast()`



Unpack the data:



Prototype Implementation

<https://github.com/carsten-clauss/psmpi/tree/cc/mr-mpix-register-collops>
<https://github.com/carsten-clauss/mpc/tree/cc/mpix-register-collops>

- Registration interface implemented as a proof-of-concept in:
 - ParaStation MPI (MPICH derivative by ParTec)
 - Multi-Processor Computing (MPC) MPI (by CEA)
- Transparent regcoll use via plugin mechanism:
 - Selection and activation via environment variables.
 - `MPIR_CVAR_COLLOPS_PRESET=regcoll_name`
 - Automatically adds "**collops**" Info key with `regcoll_name` as value to all communicators.
 - Can also be more selective base on the communicator name.
 - `MPIR_CVAR_COLLOPS_PLUGINS=regcoll_plugin_names`
 - A comma-separated list of names/paths of regcoll plugins to be loaded.
 - Each plugin registers its algorithms via a call of `MPPIX_Register_collops()`.

ParaStation
MPI



→ The implemented plugins/collectives are API-compatible between MPICH and MPC MPI.

Plugins Implemented



Binned Sum-based MPI_(All)reduce

- offers numerical reproducibility of floating-point data summation
- purely MPI-based (uses derived datatypes + custom reduce op)

`MPIR_CVAR_COLLOPS_PRESET=pscoll_binsum`



MSA (Modular Supercomputing Architecture)

- offers hierarchical collectives for MSA environments
- uses shadow-communicators and hierarchical split types

`MPIR_CVAR_COLLOPS_PRESET=pscoll_msa`



UCC (Unified Collective Communication)

- uses UCC API and its transport layers (NCCL, RCCL, SHARP etc.)
- based on the UCC wrappers (developed by ParTec) in MPICH

`MPIR_CVAR_COLLOPS_PRESET=pscoll_ucc`



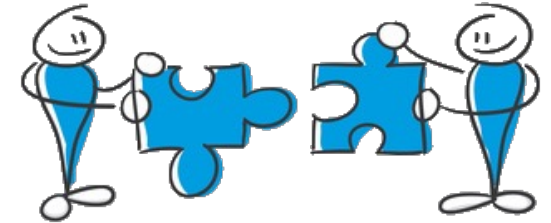
Gloo (part of the PyTorch framework)

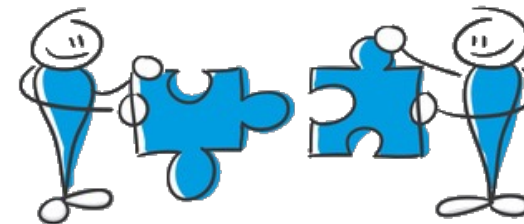
- uses Gloo library with collectives for Machine Learning workloads
- C++ template API, blocking communication semantics

`MPIR_CVAR_COLLOPS_PRESET=pscoll_gloo`

Summary

- Proposal for an MPI extension for registering user-defined collectives.
- This new interface opens the collective layer for:
 - application-specific algorithms under the hood of the familiar MPI API,
 - external collective libraries like UCC, and thus also hardware-accelerated backends.
- Integration into the standard MPI call path:
 - Selection of plugins at the communicator level via MPI Info objects.
 - Fallback to native MPI algorithms if pattern/datatype is not supported.
- Prototype implementations:
 - Integrated in two different MPI libraries: ParaStation MPI and MPC.
 - Proof-of-concept using different plugins: MPI-based Binned-Sum, MSA, UCC, and Gloo.
- Optional: transparent loading and activation of plugins through the MPI library.





Conclusion and Outlook

Open Issues and Questions

- So far, only **blocking** collectives are supported!
- Non-blocking and persistent collectives require:
 - an algorithm-specific request (extra) state,
 - progress poking and completion detection.
- MPI progress is a key challenge when integrating external libraries/backends
 - ...especially with blocking backends such as Gloo.
 - Possible/current solution: additional threads.
- Performance implications in the critical path?

Next Steps and Future Directions

- Looking for feedback from the MPI community.
- Questions for MPI users:
 - Can you imagine using such an interface to register your own and/or external collectives?
 - And if so, what use cases can you think of?
- Question for MPI library developers:
 - What challenges do you see when it would come to integrating such an interface into your library?
 - Could the interface enable cross-library sharing of collective algorithms and backend wrappers?

Or is it even time to consider standardizing the **modularization** of the MPI layer and, if required, introducing also harmonized but **"southbound" interfaces**?



Thank you! Any questions?

Carsten Clauss, Sonja Happ, and Simon Pickartz
EuroMPI 2026, Vienna, October 08, 2026