



Extending MPI for Portable Payload Compression

Carsten Clauss¹, Diego Jiménez^{2,3},
Felix Jung³, Sonja Happ¹,
Tilman Dannert², Simon Pickartz¹,
Erwin Laure², and Martin Schulz³

EuroMPI 2026, Vienna, October 8, 2026

¹ ParTec AG,

² Max Planck Computing and Data Facility,

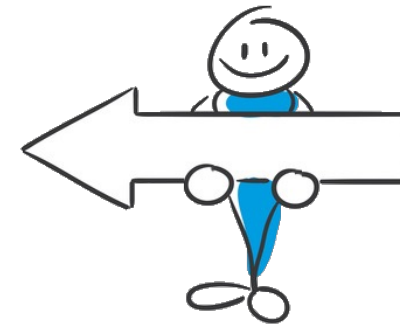
³ Technische Universität München

This work was supported by the German Federal Ministry of Research, Technology, and Space within the **DaREXA-F project** (16ME0646, 16ME0642K, 16ME0644) as well as by the European Commission through the EuroHPC project Plasma-PEPSC (grant number 101093261).



Overview

- Motivation
- Challenge: Integration of lossy compression into an MPI library
- Proposal of an MPI extension:
 - Registration of MPI payload compressors
 - Selection and control via MPI Info hints
- Prototype: Lossy compression with Partitioned Communication
- Conclusions and some open points



Motivation

- Current trends:
 - Computing power increases through parallelism and accelerators.
 - Communication is increasingly dominating runtime and energy consumption.
- At the same time, costs are rising for:
 - Memory accesses
 - I/O data movement
 - Communication
- Particularly relevant for inter-process message-passing in data-intensive simulations.

→ Less data transmitted may boost the speed-up more than extra computing power!

Possible Solution: Data Compression

Application Data → Compression → Network → Decompression → Application Data

- Compress MPI payload data before and decompress after transmission.
 - Smaller payload potentially reduces communication time.
 - Trade-off: data reduction vs. (de)compression time.
 - Advantage if: **time saved on transmission > (compression + decompression time)**.
- Two types of data compression:
 - **Lossless** → Usually only little reduction, but can be applied **transparently**.
 - **Lossy** → Significant potential for reduction, but must be used **deliberately**.

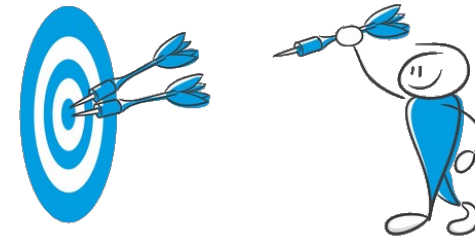


→ Particularly lossy compression promises a notably improved performance!

The Challenges of Lossy Compression

Lossy compression alters the data! → Worst case: data gets corrupted!

- The application must decide:
 - What level of deviation is acceptable?
 - Which compressor is appropriate?
 - Which parameters should be used?
- Conflicting goals between:
 - Accuracy and acceptable errors.
 - Compression ratio and compression overhead.
 - Achievable reduction in communication time.



→ Lossy compression requires close coordination between the application and the compressor!

Limitations of Existing Approaches

Lossy compression at the application level

- Control over data, accuracy, and error tolerances.
- Application-specific code to control the compressor.
- Limited reusability and portability.

Wrappers and specialized frameworks

- Typically, wrap directly around MPI Send/Recv functions.
- Use-case and domain-specific.
- Lack of integration with the MPI-internal communication path.

→ Idea: A non-transparent yet portable integration of lossy compression with MPI?



Research Questions of the DaREXA-F Project

- Can lossy compression of MPI payload be integrated directly into an MPI library?
- How can different compressors be used without integrating them firmly into MPI?
- How can the required parameters for steering the compressor be passed?
- What extensions to the MPI programming model and API are necessary for this?
- Can the feasibility be demonstrated with a working integration into an MPI library?



Research Result: Proposal for an MPI Extension

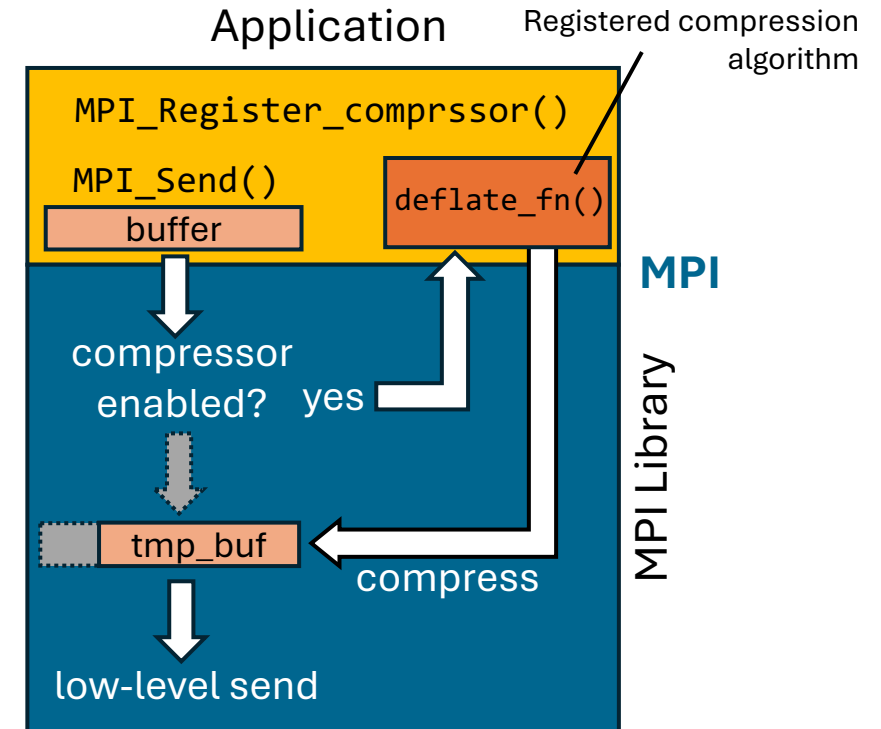
Registration of MPI Payload Compressors

- Extend the MPI interface for registering compression algorithms as **callbacks**.

`MPIX_Register_compressor()`

(...inspired by the existing `MPI_Register_datarep()`)

- Selection and control of registered compression callbacks by the application via **MPI Info objects**.
- Info objects can be passed directly (e.g., in case of part. comm.) or via the communicator used.



A closer look at Compressor Registration

MPIX_Register_compressor(**name**, **req_init_fn**, **req_free_fn**, **deflate_fn**,
inflate_fn, **deregister_fn**, **info**, **extra_state**)

- **name** compressor identifier (string)
- **req_init_fn** function invoked to initialize a compressor for a communication request (callback)
- **req_free_fn** function invoked to release compressor resources allocated for a request (callback)
- **deflate_fn** function invoked to deflate (i.e., compress) a payload buffer before sending (callback)
- **inflate_fn** function invoked to inflate (i.e., decompress) a payload buffer after receiving again (callback)
- **deregister_fn** function invoked when all resources are to be released at the end of the MPI runtime (callback)
- **info** optional MPI info parameter (handle)
- **extra_state** extra state for compressor

→ Callbacks to be implemented by the application (or provided by libraries or frameworks).

A closer look at Compressor Usage

Example: Sending side of a partitioned communication

```

MPI_Info_set(info_compr, "compressor", "darexa:zfp");
MPI_Info_set(info_compr, "zfp:backend", "cpu");
MPI_Info_set(info_compr, "zfp:datatype", "double");
MPI_Info_set(info_compr, "zfp:fixed_rate", "8");
...
MPI_Psend_init(buffer, num_parts, part_size, MPI_DOUBLE, dest, tag, comm, info_compr, &request);
...
MPI_Start(&request); // Start partitioned sending scheme
...
#pragma omp parallel for shared(request)
for (int i=0; i < num_parts; i++) MPI_Pready(i, request);
MPI_Wait(&request, &status);
...

```

Name (string) as used during
MPIX_Register_compressor()

Callback for data compression
(deflate_fn) is called **transparently**
by the MPI library before sending

→ Callbacks are invoked by the MPI library as part of the communication scheme.

Integration with Partitioned Communication

- Large messages are broken down into individual partitions.
 - Each partition can be transmitted (and compressed/decompressed) independently
 - Compression can take place immediately before transmission (i.e., triggered by `MPI_Pready()`)
 - Decompression can occur directly after a partition arrives (i.e., before `MPI_Parrived()`)

- Possible overlap between:

- Computation
- Compression
- Communication

Calculate Part0 → Compress Part0 → Send Part0

Calculate Part1 → Compress Part1 → Send Part1

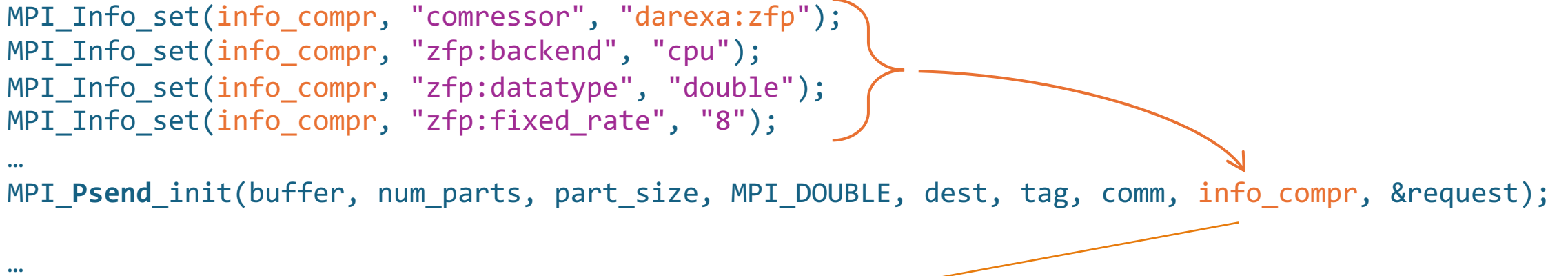
Calculate Part2 → ...

→ Interleaving partitioned communication with compression promises beneficial pipelining.

Integration with Partitioned Communication

```

MPI_Info_set(info_compr, "compressor", "darexa:zfp");
MPI_Info_set(info_compr, "zfp:backend", "cpu");
MPI_Info_set(info_compr, "zfp:datatype", "double");
MPI_Info_set(info_compr, "zfp:fixed_rate", "8");
...
MPI_Psend_init(buffer, num_parts, part_size, MPI_DOUBLE, dest, tag, comm, info_compr, &request);
...
    
```

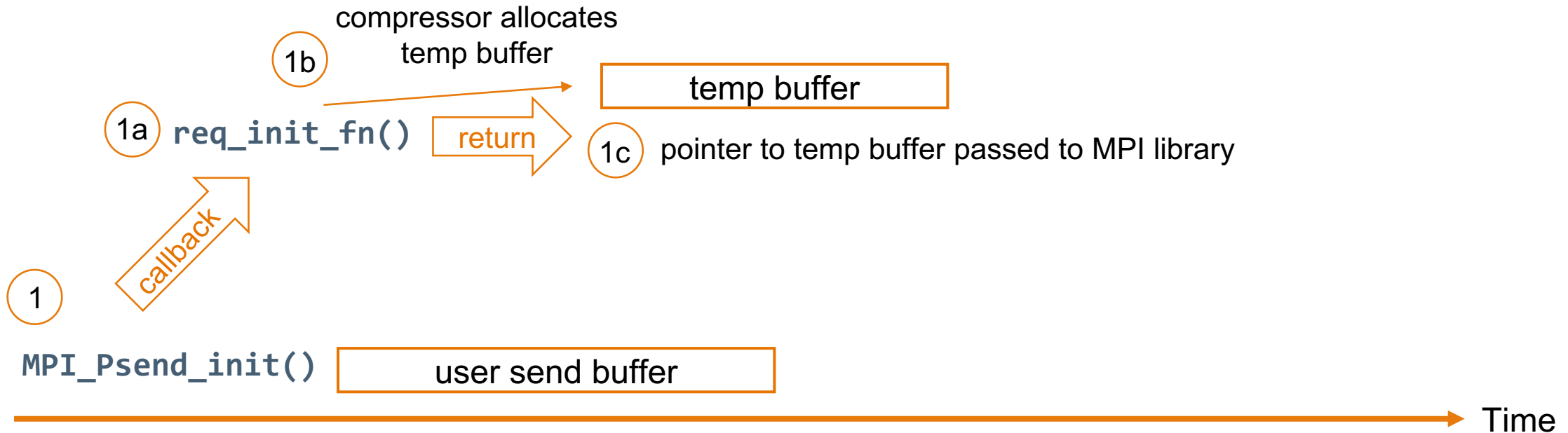


1

`MPI_Psend_init()` user send buffer

Time →

Integration with Partitioned Communication



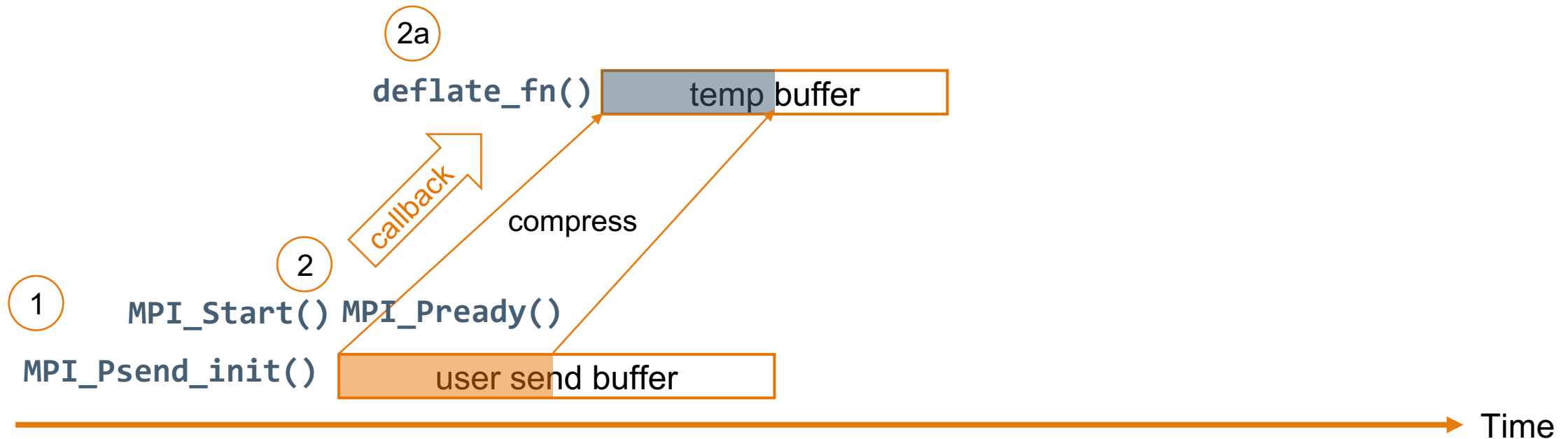
Integration with Partitioned Communication

temp buffer

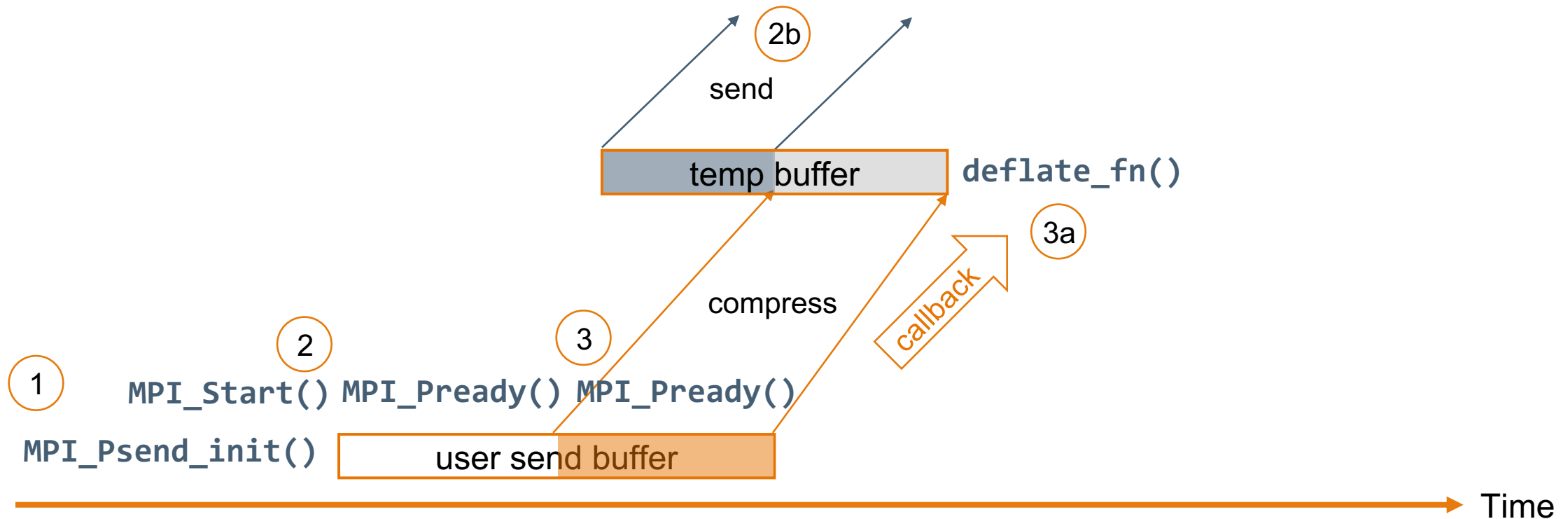


Time

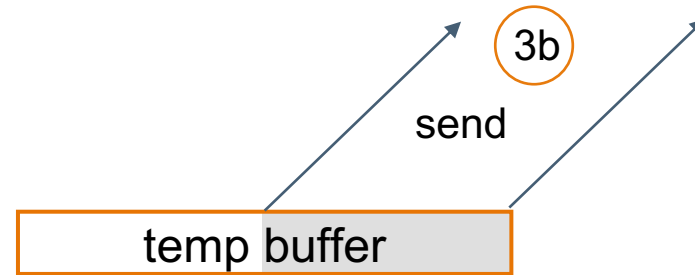
Integration with Partitioned Communication



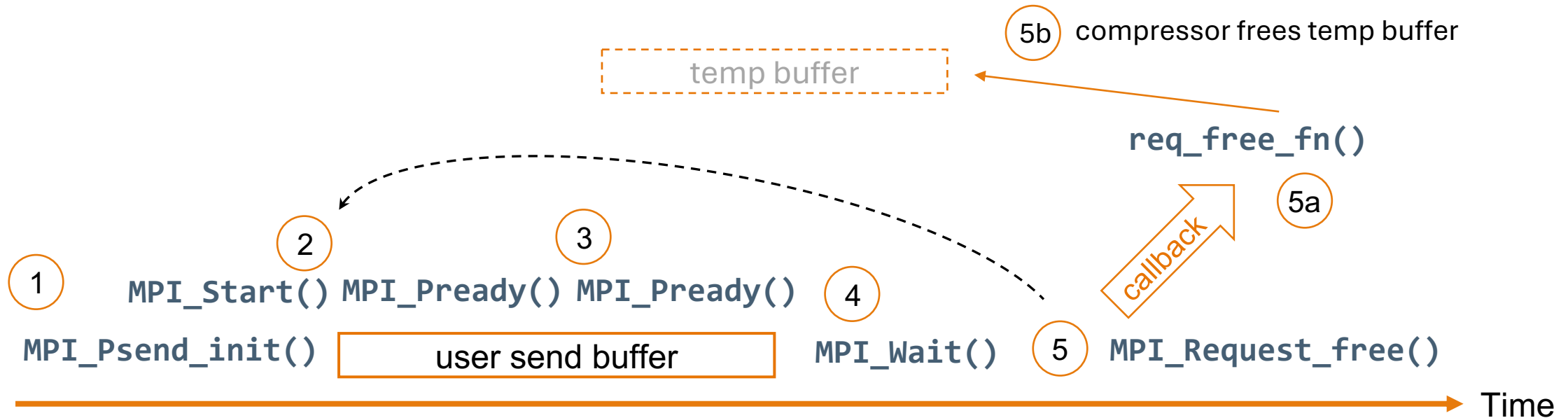
Integration with Partitioned Communication



Integration with Partitioned Communication



Integration with Partitioned Communication



Prototype Implementation

MPI extensions are part of ParaStation MPI

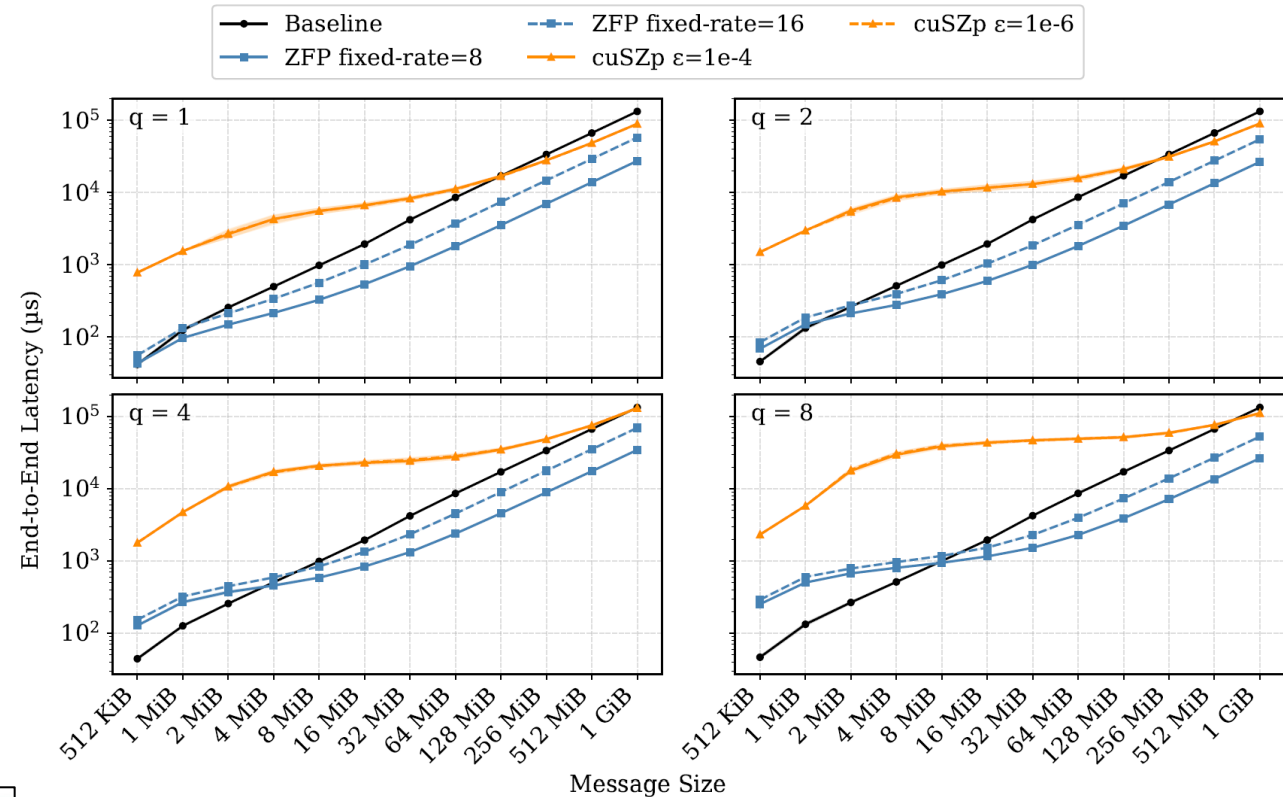
- Production-grade, high-performance MPI library.
- Based on MPICH and developed by ParTec AG.
- Available as open-source on GitHub:
<https://github.com/ParaStation/psmpi>
- Usage examples and tests are also available:
see [psmpi/mpich2/test/mpi/parastation/compr](#)

ParaStation
MPI

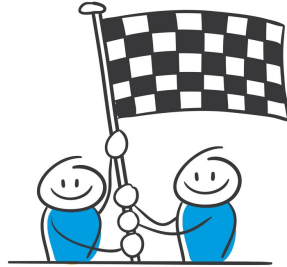
Prototype Implementation

Proof-of-concept MPI payload compressors

- Implemented by MPCDF and supported by TUM as part of the DaREXA-F project:
 - ZFP-based (fixed-rate) compressor,
 - cuSZp-base (error-bound) compressor.
- Use the MPIX_Register_compressor extension.
- Here evaluated with an (only slightly modified) OSU partitioned communication benchmark.



Conclusion



Summary

- API extension based on existing MPI concepts.
- Compressors are modular and interchangeable.
- Application retains control over compression.
- Close integration with the MPI-internal communication paths via callbacks.
- Effective pipelining possible in combination with partitioned communication schemes.
- Successful demonstration with ParaStation MPI and two MPI payload compressors as proof-of-concept.

Open Points

- More thorough benchmarking (eps. focusing on the benefits of pipelining).
- Investigation into how the compressor could provide feedback on data quality.
- Alternative implementations and other MPI libs?
- Perspectives for standardization?





Thank you!

Any questions?

Carsten Clauss¹, Diego Jiménez^{2,3},
Felix Jung³, Sonja Happ¹,
Tilman Dannert², Simon Pickartz¹,
Erwin Laure², and Martin Schulz³

EuroMPI 2026, Vienna, October 8, 2026

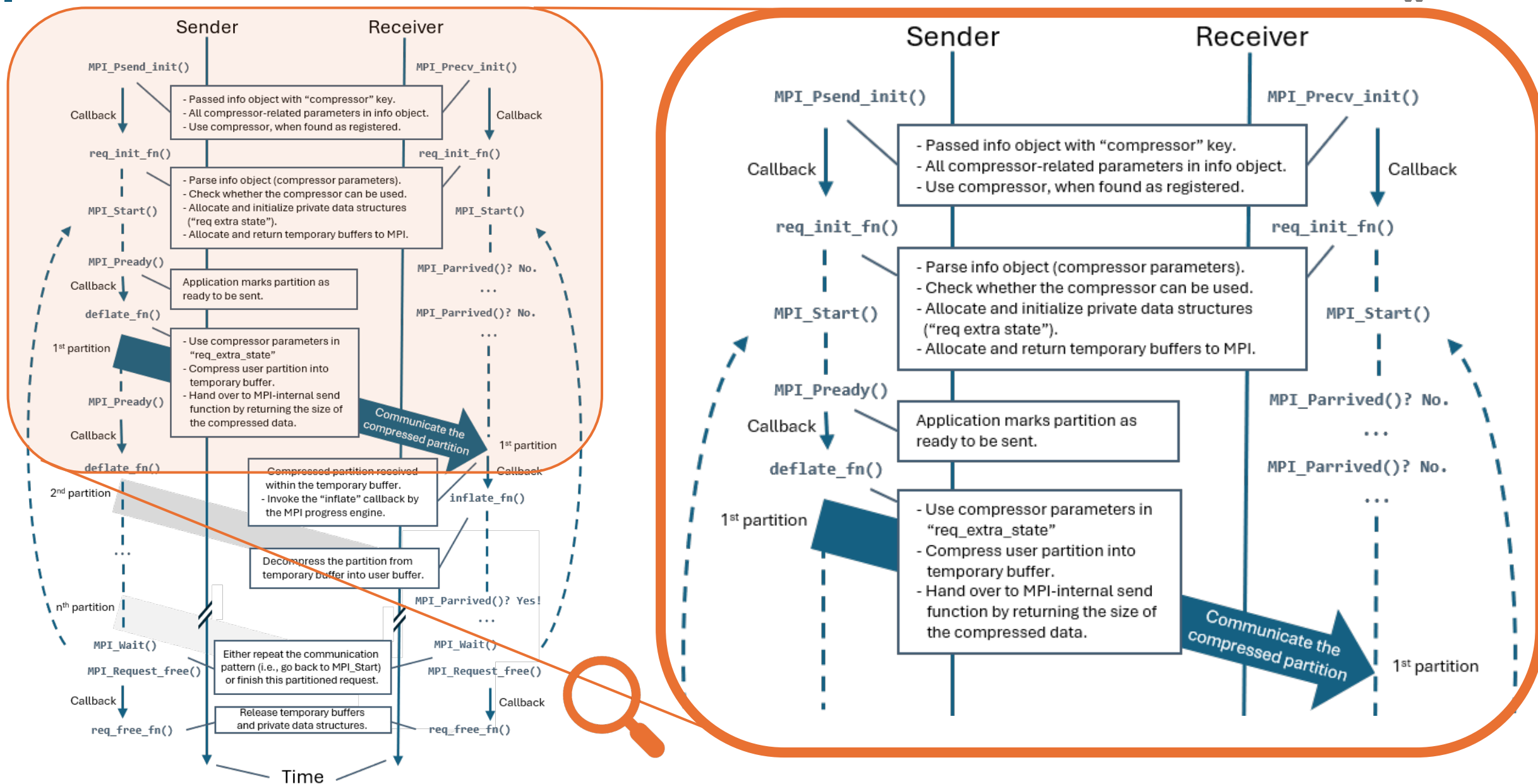
¹ ParTec AG,

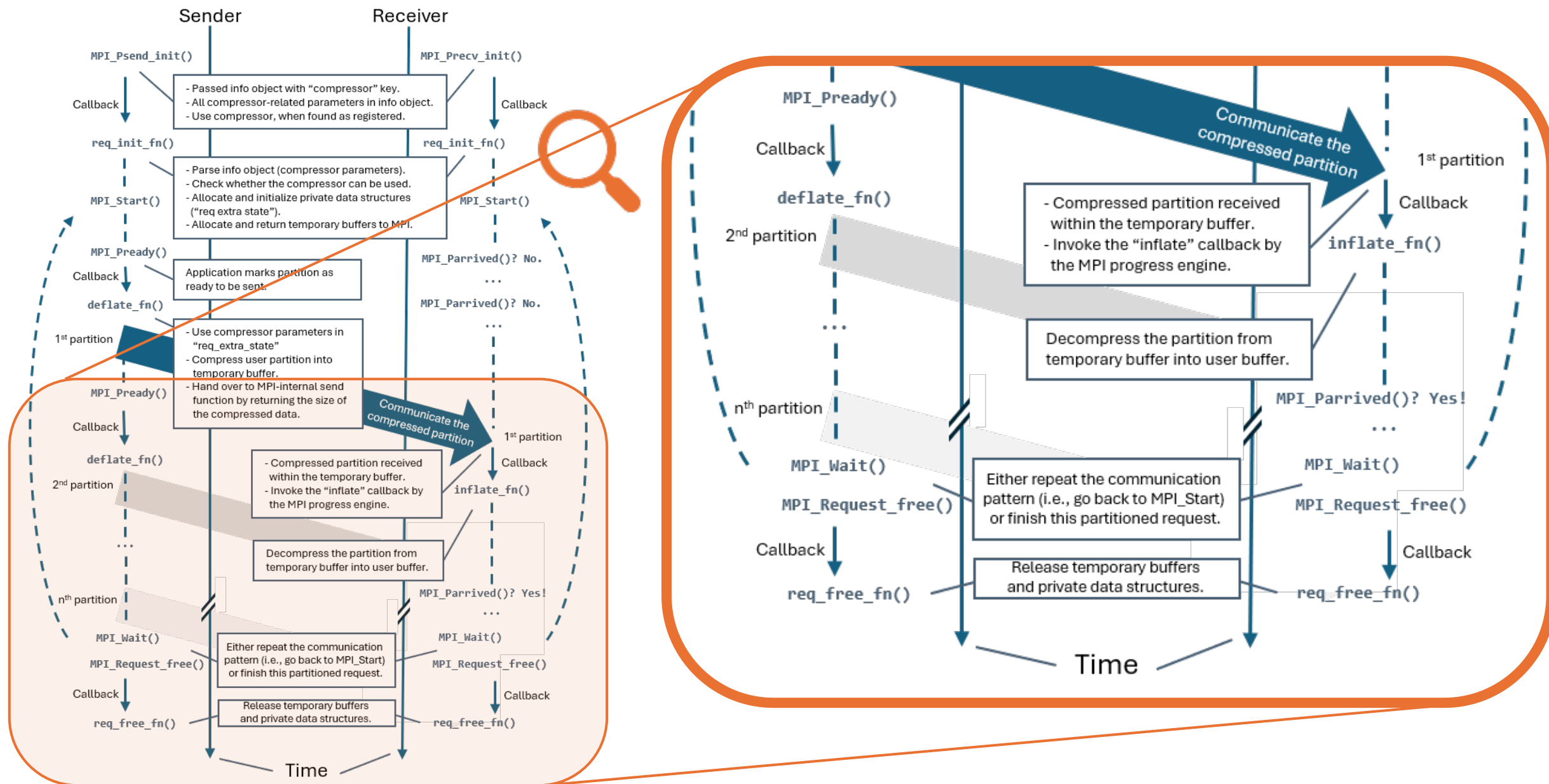
² Max Planck Computing and Data Facility,

³ Technische Universität München

This work was supported by the German Federal Ministry of Research, Technology, and Space within the **DaREXA-F project** (16ME0646, 16ME0642K, 16ME0644) as well as by the European Commission through the EuroHPC project Plasma-PEPSC (grant number 101093261).



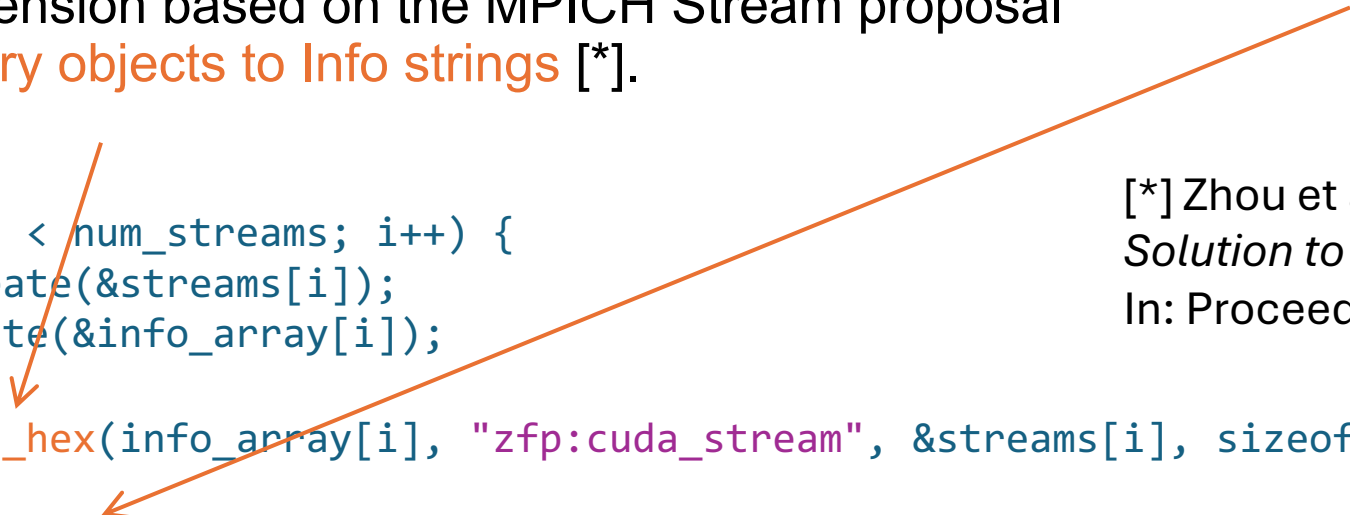




Integration with CUDA Streams

- Offloading the compression to a GPU can further speed things up.
- Especially if the data is already located in GPU memory and sent directly from there.
- For partitioned communication, parallel CUDA streams could enable effective pipelining.
- The parallel CUDA streams could then be passed again as **array-type info objects**...
- ...using an extension based on the MPICH Stream proposal to **convert binary objects to Info strings** [*].

```
for (int i = 0; i < num_streams; i++) {  
    cudaStreamCreate(&streams[i]);  
    MPI_Info_create(&info_array[i]);  
  
    MPIX_Info_set_hex(info_array[i], "zfp:cuda_stream", &streams[i], sizeof(*streams));  
}  
MPIX_Info_merge_from_array(num_streams, info_array, &info_merge);
```



[*] Zhou et al.: *MPICH Stream: An Explicit Solution to Hybrid MPI+X Programming*.
In: Proceedings of the EuroMPI/USA'22.

Outlook on a further extension

Status-attached MPI Info objects

- Add a further callback: **req_done**
- Within, an Info object can be attached to the status struct to be returned after communication.

```
MPI_Precv_init(buffer, num_parts, part_size, ...);  
...  
MPI_Start(&request); // Start receiving  
...  
MPI_Wait(&request, &status);  
MPIX_Status_get_info(&status, &info_status);  
MPI_Info_get_string(info_status,  
                    "compressor:accuracy", ...);  
...  
MPI_Request_free(&request);  
...
```

Open Questions

- How valid is the evaluation as it is based solely on a prototype and microbenchmarks?
- How can consistent configuration on the sender and receiver be made easier?
- **How can a compressor report back accuracy deviations to the application?**
- How portable is this approach as the metadata ultimately depends on the compressors used?
- How useful would be pursuing standardization?

