LLNL-PRES-460253

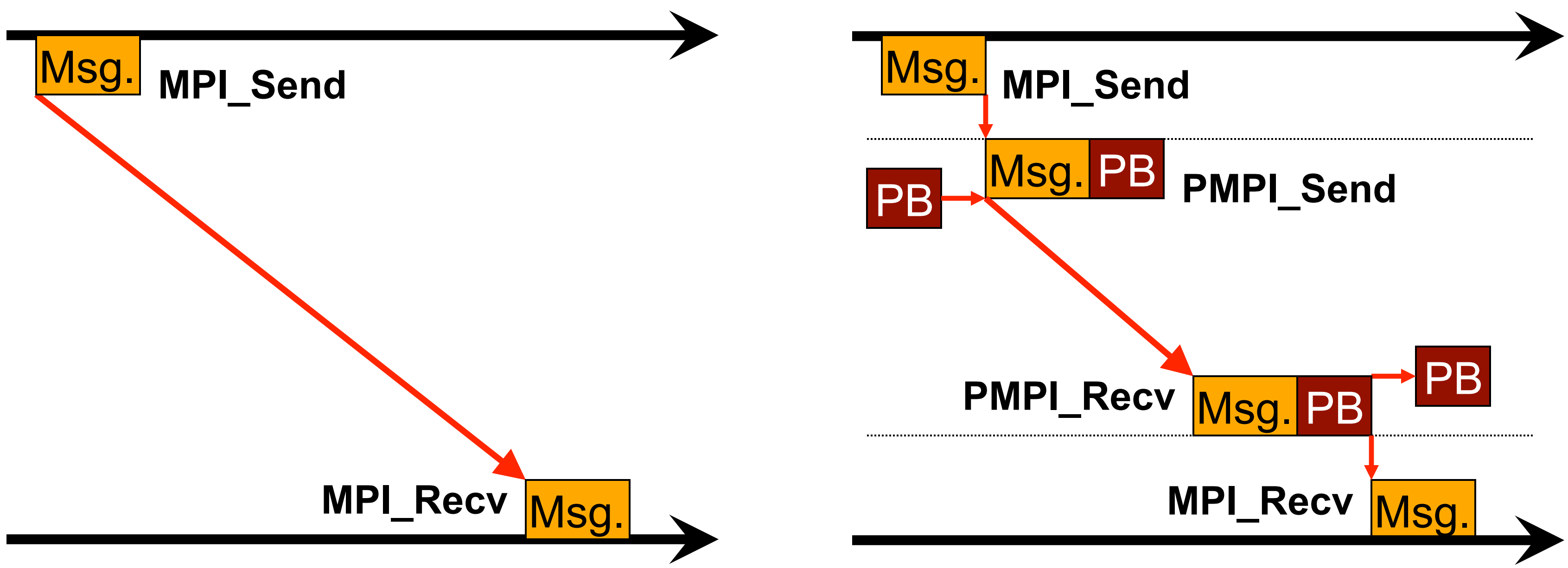
Piggybacking in MPI-3

MPI Forum Meeting, October 2010

Martin Schulz with input from
George Bosilica, Greg Bronevetsky, Darius Buntinas,
Bronis de Supinski, Kathryn Mohror, Anh Vo

What do we mean by piggybacking?

- Transparently attach data to messages
  - Uniquely associated with a specific message
  - Piggyback data added/removed without influencing application
  - Typically used for small amounts of data



Use Cases

- Fault Tolerance Protocols
  - Transparent propagation of checkpoint interval IDs
  - Message logging
- Performance Analysis
  - Overhead propagation/detection
  - Critical path analysis
  - Late sender instrumentation
- Debugging
  - MPI correctness tools
- General
  - Transparent state propagation in libraries
  - Message matching

Requirements

- Must work for the complete MPI standard
  - All point to point operations, including wild card receives
  - Collective operations, including reductions and barriers
  - Need to look into all corner cases (e.g., Issend matching Irecv(*))
- Thread safety
- Low overhead, minimal impact on performance
  - Raw bandwidth and latency
  - Application performance
- "Selective piggyback", ideally per message
  - Ability to do piggybacking on a subset of messages
  - Receiver needs to know/query whether piggyback is present
  - Selective piggybacking infeasible for PMPI wrappers

Solution on top of MPI are insufficient (1)

- Three main options
  - A) Two messages
  - B) MPI datatypes (as they are now)
  - C) Manual packing into a separate buffer
- A) Two messages
  - Fails for wildcard receives (can lead to incorrect matchings)
  - Transparent data return difficult (need to attach to requests/ requires memory allocations)
  - Thread safety unclear (how to handle two sends from two threads in the same process)

Solution on top of MPI are insufficient (2)

- B) MPI datatypes (as they are now)
  - Requires creation of new datatype for each message (expensive and hard to optimize)
  - Use of MPI_BOTTOM may reduce optimization potential
  - Large overhead on applications (see next slides)
  - Partial receives cause problems (how to correctly count number of elements and bytes)
  - Does not work for reduce/barrier (no datatypes used), difficult for other collectives
- C) Manual packing into a separate buffer
  - High overhead (memory copy for each messages)
  - Partial receives hard (how to correctly count number of elements and bytes)
  - Does not work for reduce (different ops)/barrier (no data), difficult for other collectives

Solution on top of MPI are insufficient (3)

- In all cases: no selective piggybacking
  - Receiver does not know whether sender included piggyback
  - Sender does not know whether receiver expects piggyback
  - Only solution: always send piggyback data
- Even if they would work, performance is problematic
  - Significant overheads in latency and bandwidth
  - Performance not portable across implementations

Measurement Setup

- MPI implementations and platforms
  - MVAPICH on LLNL's TLCC clusters
  - Open MPI on LLNL's TLCC clusters
  - Blue Gene/P (dawndev) at LLNL
  - [IBM AIX (up) and IA-64/Quadrix (thunder) at LLNL]
- Experiments
  - Raw bandwidth and latency for piggybacking
    - Three naïve implementations mentioned before
    - Ignoring correctness problems
  - Sending with complex datatypes
  - Data type creation (incl. complex datatypes)
  - Application performance (Sweep3D and SMG2000)

Latency

- Hera 4 byte (MVAPICH 1)
  - Base version: 2.163us
  - Datatype: 3.306us (**52.8%**)
  - Two messages: 3.154us (**45.8%**)
  - Packing: 2.926us (**35.3%**)
- Hera 4 byte (Open MPI 1.4.2)
  - Base version: 1.804us
  - Datatype: 3.444us (**90.9%**)
  - Two messages: 3.623us (**100.8%**)
  - Packing: 3.411us (**89.1%**)
- BG/P 4 byte (MPICH-2)
  - Base version: 3.573us
  - Datatype: 53.959us (**1410.2%**)
  - Two messages: 8.099us (**126.7%**)
  - Packing: 14.509us (**306.1%**)

Bandwidth

- Hera 4 byte (MVAPICH 1)
  - Base version: 1238 MB/s
  - Datatypes: 579 MB/s (**-53.2%**)
  - Two messages: 1231 MB/s (**-0.6%**)
  - Packing: 572 MB/s (**-53.8%**)
- Hera 4 byte (Open MPI 1.4.2)
  - Base version: 1429 MB/s
  - Datatype: 1106 MB/s (**-22.6%**)
  - Two messages: 1104 MB/s (**-22.7%**)
  - Packing: 1105 MB/s (**-22.7%**)
- BG/P 4 byte (MPICH-2)
  - Base version: 355 MB/s
  - Datatypes: 239 MB/s (**-32.7%**)
  - Two messages: 351 MB/s (**-1.1%**)
  - Packing: 175 MB/s (**-50.7%**)

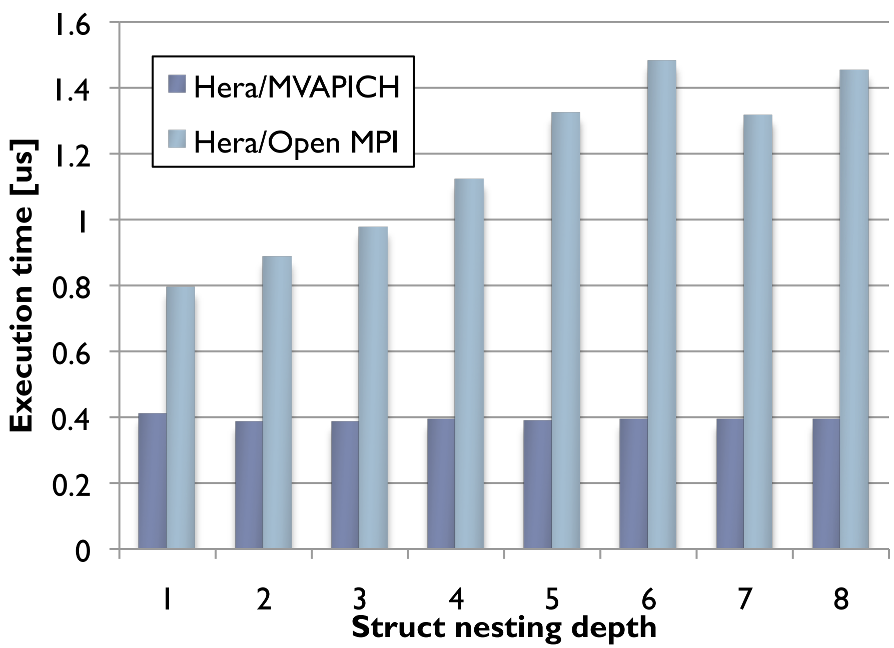
Sending with Wrapped Datatypes

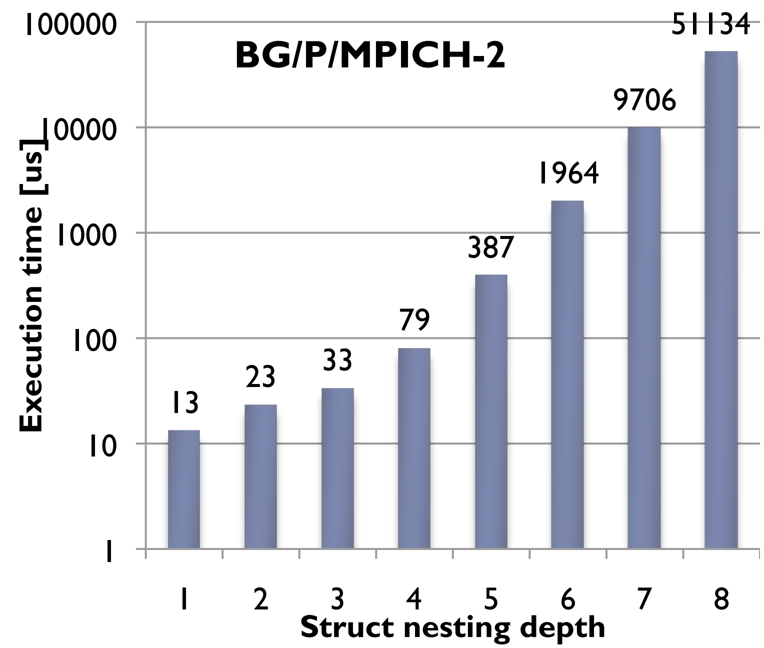
- Latency for sending with struct datatypes
  - Pre-create one struct datatype and send repeatedly with it
  - Hera/MVAPICH: **17.4%** overhead (2.540us vs. 2.163us)
  - Hera/Open MPI: **8.8%** overhead (1.964us vs. 1.804us)
  - BG/P: **77.9%** overhead (6.357us vs. 3.573us)

- Latency for sending with continuous datatype creation
  - Create one struct datatype for each send and receive operation
  - Hera/MVAPICH: **52.3%** overhead (3.295us vs. 2.163us)
  - Hera/Open MPI: **73.8%** overhead (3.136us vs. 1.804us)
  - BG/P: **1220.2%** overhead (47.171us vs. 3.573us)

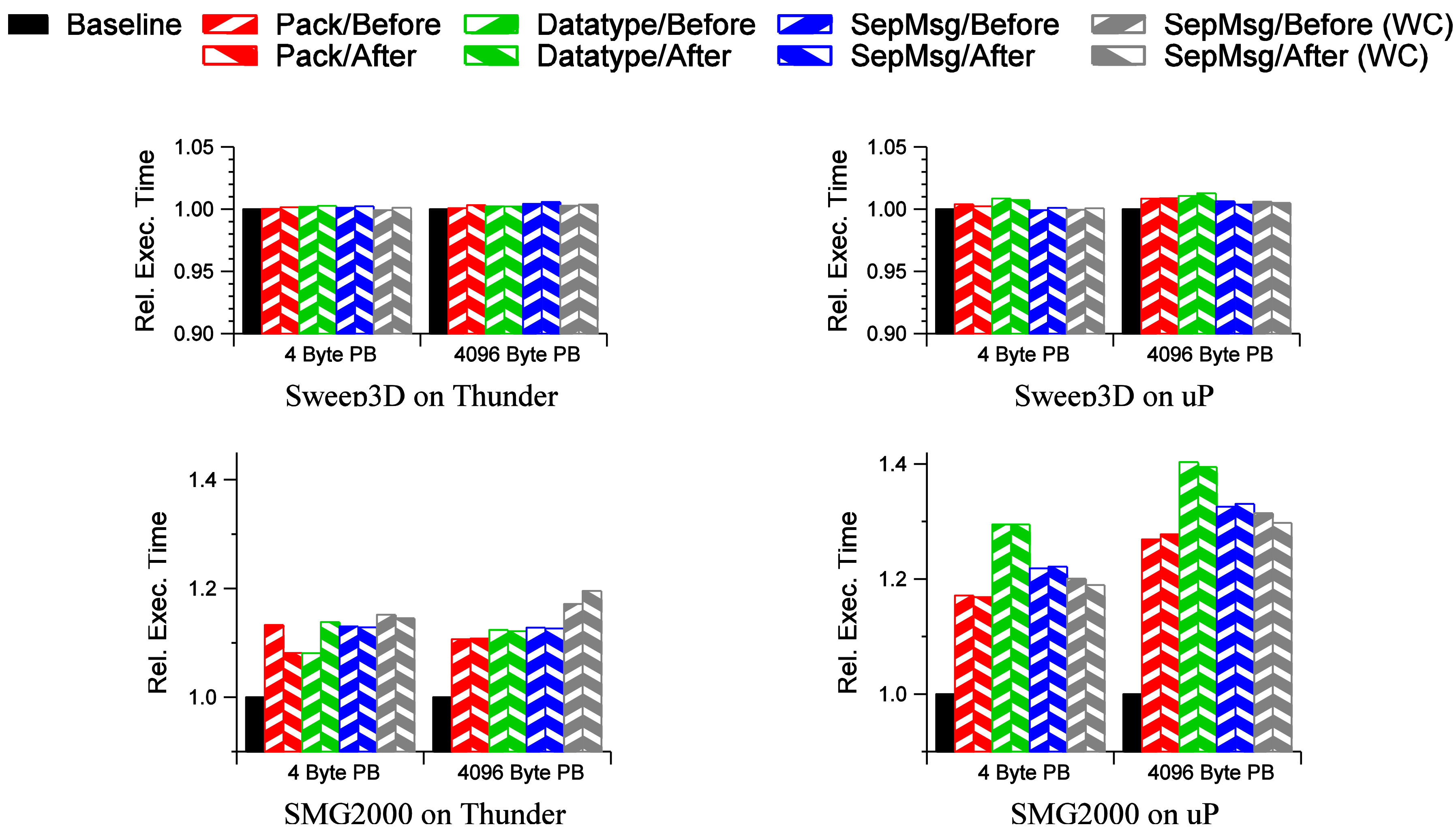
Datatype Creation

- Continuous creation and freeing of datatypes
  - Single node execution
  - Increasing depth of structured datatypes
    (from "struct" to "struct of struct" to "struct of struct of struct")





Application Performance



Pack	= Use memory copy to add PB to message
Datatype	= Use new struct datatype for each message
SepMsg	= Use two messages
SepMsg(WC)	= SepMsg with using MPI_ANY_SOURCE in the code

We Need a New API for Piggybacking

- Requirement for many tools, but
  - Existing solutions don't work or are too expensive
  - Solutions using PMPI are infeasible
- However, this is not a difficult requirement
  - Can easily be implemented within MPI library
  - Portable access to this functionality requires standardized API
- Design guidelines
  - Clean semantics
  - Specific for the intended use, not too general
  - Allow optimizations in MPI for better piggybacking

Ideal Solution: Separate Set of API Calls

- Explicit PB buffers as arguments to send calls
  - **MPI_SendPB**(buf,count,datatype,dest,tag,comm
    pb_buf, pb_count,pb_datatype)
  - pb_count and pb_datatype could be restricted
- Receiving PB data is done through the status object
  - Accessor routine maintains compatibility
  - **MPI_CheckPB(status, flag, buf,count,datatype)**
    returns flag=1 if PB has been received
    buf,count, datatype act like a receive operation
- Separate arguments in collective calls
  - **MPI_BcastPB**(buf,count,datatype,root,comm,
    pb_buf,pb_count,pb_datatype)
- Full functionality by allowing many cross combinations (e.g., gather/reduce) (reduction operation for PB depends on PB user and is independent of the host operation in the MPI code)
  - **MPI_BcastPBReduce**(buf,count,datatype,root,comm,
    pb_buf,pb_count,pb_datatype,reduction_operator)

Pros and Cons

- Advantages
  - Clean semantics of adding an orthogonal aspect
  - Directly/Only implements the required functionality
- Disadvantages
  - More API calls
  - Cross product of all collectives and all PB collectives necessary
  - Wrapped PB messages a bit more difficult
- Open questions
  - Which collective operations should be supported for PB?
  - Updates to message matching rules (?)

Alternatives

- Prototype datatypes
  - Precreate “datatype prototypes”
  - Populate prototypes before sending/receiving

- Overloading parameters
  - Reuse existing messaging APIs
  - “Attach” piggyback information to a parameter
    - Datatypes
    - Communicators

Datatype Prototypes for Point to Point

- Precreate datatype
  - Create a two element struct
    - Specify pb_count, pb_datatype, but not pb_buf
    - Don't specify user buffer
  - Before message send/receive
    - Add pb_buf information
    - Instantiate prototype with user buffer information
- Idea: expensive step only once
  - MPI has chance to preallocate datatype
  - Per message operation simple and cheap
  - Maintain datatype optimization principle

Example Part 1: Setup (once)

- **MPI_Aint offsets[2];**
  int counts[2];
  MPI_Datatype types[2];
  MPI_Datatype mine;
- offsets[0] = **MPI_UNKNOWN;**
  offsets[1] = **MPI_UNKNOWN;**
- types[0] = pb_dtype;
  types[1] = **MPI_TYPE_UNKNOWN;**
- counts[0] = pb_count;
  counts[1] = **MPI_CNT_UNKNOWN;**
- MPI_Type_create_pstruct(2, counts, offsets, types, &mine);
- MPI_Type_commit(&mine);

Example Part 2: Communication

- **MPI_Datatype mine_complete;**
- offsets[0] = pb_buf;
  offsets[1] = user_buf;
- types[1] = user_dtype;
  counts[1] = user_count;
- **PMPI_Type_complete_pstruct**(2, counts, offsets, types, mine, &mine_complete);
- **PMPI_Send(**MPI_BOTTOM, 1, mine_complete, dest, tag, comm**);**
- PMPI_Type_free(&mine_complete);

Discussion

- Pros
  - Small API extension
  - Builds on top of familiar concepts
- Cons
  - Still a complex datatype creation
  - Piggyback too generic
  - Requires memory allocation/free for each send/receive
  - Difficult for collectives
  - Does not apply to barriers
- Open questions
  - Performance
  - Support for collectives
    - API to support grouping of collectives?
  - Support for selective piggybacking

Attach to Existing Parameters

- We can't change existing APIs
  - Assumption: we don't want to duplicate API
  - Need to use existing parameters to convey new information
  - Overload arguments
- Can be done with any parameter; attractive choices are
  - Datatypes
  - Communicators
- Main idea
  - Pre-create template to notify MPI that this may happen
  - Dynamically attach content to parameter before call
  - Automatic detach after call / use once

Attach PB to Datatypes

- Create a template ahead of time
  - MPI_Datatype_PB_Template(MPI_Datatype pb_dt, int pb_count, MPI_Datatype_template *pb_template)
- Optionally provide collective operation (default none)
  - MPI_Datatype_PB_ReduceOp(MPI_Op pb_op, MPI_Datatype_template *pb_template)
- Create new sending datatype
  - MPI_Datatype_PB_Instatiate(void *buf, MPI_Datatype_template pb_template, MPI_Datatype dt, MPI_Datatype *dt_new)
- Communication with new datatype
  - MPI_Send(<use new datatype>)
  - MPI_Recv(<use new datatype>)
- Check whether we received a piggyback operation
  - MPI_Datatype_PB_Filled(MPI_Datatype dt_new, int flag)
- Note: no free operation required

Attach PB to Communicator

- Create a template ahead of time
  - MPI_Comm_PB_Template(MPI_Comm comm, int count, MPI_Comm_template *pb_template)
- Optionally provide collective operation (default none)
  - MPI_Comm_PB_ReduceOp(MPI_Ops pb_ops, MPI_Comm_template *pb_template)
- Create new sending datatype
  - MPI_Comm_PB_Instatiate(void *buf, MPI_Comm_template pb_template, MPI_Comm comm, MPI_Comm *comm_new)
- Communication with new communicator
  - MPI_Send(<use new communicator>)
  - MPI_Recv(<use new communicator>)
- Check whether we received a piggyback operation
  - MPI_Comm_PB_Filled(MPI_Comm comm_new, int flag)
- Note: no free operation required

Example for Usage

```
/* One time setup */
MPI_Comm_template tcw;
MPI_Comm_PB_Template(MPI_COMM_WORLD,1,&tcw);
…
/* Send wrapper */
MPI_Send(buf,c,dt,target,tag,comm)
    int pbbuf;
    MPI_Comm newcomm;
    MPI_Comm_PB_instantiate(&pbbuf,tcw,comm,&newcomm);
    PMPI_Send(buf,c,dt,target,tag,newcomm);
…
/* Receive wrapper */
MPI_Recv(buf,c,dt,target,tag,comm,status)
    int pbbuf,flag;
    MPI_Comm newcomm;
    MPI_Comm_PB_instantiate(&pbbuf,tcw,comm,&newcomm);
    PMPI_Recv(buf,c,dt,target,tag,newcomm,status);
    MPI_Comm_PB_filled(newcomm,&flag);
    if (flag) { <use data in pbbuf> }
```

Conclusions

- We need a piggybacking API in MPI
  - Many use cases for a wide range of tools
  - Existing functionality insufficient
- Semantically clean solution
  - Duplicate messaging API
  - Add new parameters to specify piggybacking
- Alternative with small API footprint
  - Datatype prototypes
  - Overloading datatypes
  - Overloading communicators