Effective MPI Programming:

Concepts, Advanced Features, Do's and Don'ts

Jesper Larsson Träff
traff@par.tuwien.ac.at

Vienna University of Technology
Faculty of Informatics, Institute for Information Systems
Research Group Parallel Computing

MPI is a large and complex, but well-structured interface for message-passing based parallel programming for high-performance systems. This tutorial aims to provide an understanding of basic concepts of the interface, showing how concepts and (advanced) features of the interface can be put to work effectively in applications and lead to more read- and maintainable, better performing and more robust code. More advanced features that will be covered include non-blocking collectives, neighborhood collectives, one-sided communication, derived datatypes and process topologies. The tutorial will provide room for discussion and interaction.

Contents of this tutorial

- What MPI is (and is not)
- Performance expectations and guidelines
- Procedural (communication) features:
  - Point-to-point (reminder)
  - Collective
  - One-sided
  - Blocking and non-blocking semantics
- Declarative features:
  - Derived (user-defined) datatypes
  - Communicators and process groups
  - Virtual Topologies
  - Attributes
- Miscellaneous:
  - (Threads and hybrid programming)

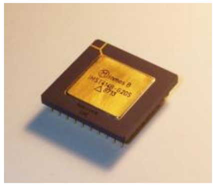
But not in this order…

Not in this tutorial

- MPI I/O
- Dynamic process management, intercommunicators
- Tools interface (PMPI and MPI_T)

The design of MPI

- Long (and ongoing) process that started around 1992
- A fortunate convergence of ideas: many distributed systems around, many programming interfaces and languages, "best practice"…



Interfaces/languages mostly based on message passing model:

- Finite set of "processes" that operate on local data, running their own program
- All exchange of data explicit (CSP: synchronous, point-to-point communication)

[C. A. R. Hoare: Communicating Sequential Processes. Commun. ACM 21(8): 666-677 (1978)]

Principle 1:

Message passing does not require (explicit) synchronization, synchronization is implied with message exchange

In MPI, no need

```
MPI_Barri (MP Comm omm)
```

…except, of cou n you need it (e.g., approximate temporal synchronization for benchmarking)

If there seems to be a semantic need, probably something is wrong. And barriers may adversely affect performance by forcing processes to wait

MPI: A liberal incarnation of the message passing model

- "Processes" (often Unix/Linux processes, but can be something else, e.g. threads) operate on local data
- Processes execute own program (different processes may have different programs, MIMD)
- Message delivery is reliable, and ordered
- Abstracts (ignores) properties of communication system

- Communication models:
  - Point-to-point (two processes explicitly involved), different semantic variations
  - One-sided (two processes involved, only one explicitly)
  - Collective (many processes involved), many operations, different semantics
  - MPI/IO (communication with file system)

Principle 2:

Message passing leads to deterministic parallel execution

…unless you program non-deterministically (randomization, MPI wildcard communication)

MPI has very clear rules for when communication takes place (matching communication calls)

Principle 3:

Communicate as little as possible, as much as necessary

Performance rule of thumb:
Communication operations are expensive (compared to local computation), avoid excessive communication

Communication is necessary for processes to collaborate and to achieve speed-up when solving non-trivial problems

Disclaimer and apologies: all examples will be in C

MPI, *the* Message-Passing Interface

- MPI is an interface (library) specification, (C and FORTRAN, not C++), it is not a particular implementation

Plenty: mpich, OpenMPI, mvapich, vendors…

- An MPI implementation is a library, not a programming language

Many things a compiler cannot optimize, use sometimes tedious

- MPI is defined for high performance

Internal buffering rarely needed, no argument checking, no meta-information with messages

- MPI does not have a performance model, does not mandate specific algorithms, nor guarantee certain performance

A very wise decision: MPI implementations possible on all kinds of different systems, has enabled scalability

MPI, *the* Message-Passing Interface

MPI is designed to be portable (can be implemented on very different types of systems)

MPI applications are portable to a very high degree

But needs some care, correct use of the standard

MPI, *the* Message-Passing Interface

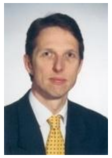
MPI itself is designed for library building: Strong concepts for

- Encapsulation, isolation (communicators)
- Information hiding (attributes)
- Querying
- Call interception (PMPI)

Many application programmers will (should?) not see any MPI calls

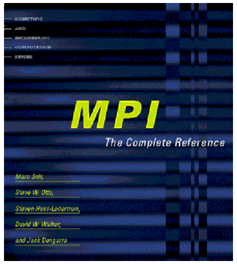
[Torsten Hoefler, Marc Snir: Writing Parallel Libraries with MPI - Common Practice, Issues, and Extensions. EuroMPI 2011: 345-355]

Evolution of the MPI Standard



⋮

- MPI 1.0, 1.1, 1.2: 1994-1995
  - Point-to-point and collective communication, datatypes, …
- MPI 2.0: 1997
  - One-sided communication, parallel I/O, dynamic process management
- MPI 2.1: 2008
  - consolidation
- MPI 2.2:2009
  - Scalable topologies, new collectives



Implementations:

ANL: `mpich`, 1996

NEC: MPI/SX, 2000

`mpich2`, 2004
OpenMPI, 2006

Evolution of the MPI Standard

MPI 3.0: 2012

- Non-blocking collectives, topological (neighbor) collectives, extended one-sided communication, performance tool support
- C++ bindings removed
- Still no fault-tolerance support

MPI: A Message-Passing Interface Standard
Version 3.0
Message Passing Interface Forum
September 21, 2012

MPI 3.1, 2015

- Fixes
- Non-blocking I/O

Implementations:

`mpich` but (still) not mature; quality, performance? Announced at SC 2012

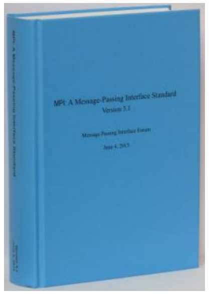
MPI: A Message-Passing Interface Standard
Version 3.1
Message Passing Interface Forum
June 4, 2015

MPI 3.1
June 2015

MPI 3.0: Vienna, Sept. 21, 2012

The MPI standard: now (from 2015) 3.1

- *The standard* is the rock-bottom for questions on MPI
- Not a formal specification, but (quite) precise (a lot of effort has been invested…)
- …except where it is deliberately imprecise (progress rules: when exactly message transfers happen)
- Quite readable (recommended, better than many other books)
- Maintained by the MPI Forum: an open gathering for interested parties, regular meetings 4-6 times a year (MPI is not an ISO, ANSI or IEEE standard, it's free)


MPI: A Message-Passing Interface Standard
Version 3.1
Message Passing Interface Forum
June 4, 2015

MPI 3.1: June 4th, 2015, see
www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf

Most MPI implementations implement (most of) MPI 3.1 (yearly status at SC and other events)

„MPI is designed not to make easy things easy, but to make difficult things possible" , Gropp, EuroPVM/MPI 2004

MPI is a large standard

- MPI 3.1 around 416 C functions
- There is a (good) reason for most
- …but it is possible to do with with only 6 functions (init, finalize, size, rank, send, recv) plus a few more
- Concepts and functionality are expressive and powerful and work well together

Conjecture (tested at EuroPVM/MPI 2002):
For any MPI feature there will be at least one (significant) user depending essentially on exactly this feature

Contagious issues

- Is MPI too large after all?
- Is MPI scalable (specification and implementations)?

[Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, Jesper Larsson Träff:MPI on millions of Cores. Parallel Processing Letters 21(1): 45-60 (2011)]

MPI has scaled to >1,000,000 cores!

- Support for fault-tolerance (some argue that the MPI specification is fault-tolerant enough, and that fault-tolerance is an implementation issue; some think differently, and FT has been debated for 10 years in the MPI Forum)

[William Gropp, Ewing L. Lusk: Fault Tolerance in Message Passing Interface Programs. IJHPCA 18(3): 363-372 (2004)]

- One-sided communication

MPI for communication

MPI is a communication interface that makes it possible to exploit modern hardware (oops: GPU? FPGA? Accelerators?) very efficiently (given a good implementation)

MPI is not a high-level interface for general purpose (distributed) parallel programming. In particular, application programmer (or library on top of MPI) must handle

- Data structure distribution
- Domain decomposition
- Load balancing
- …

but MPI can help

Example: first (noop) MPI program

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // …

  MPI_Finalize();
  return 0;
}
```

The MPI header file

First MPI call: init the library

MPI runs: Use it

Last MPI call: close it

Compile with mpicc, run with mpirun/mpiexec…

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // …

  MPI_Finalize();
  return 0;
}
```

Not here: Consult your systems manuals

mpirun starts processes, after MPI_Init() they are MPI processes

- MPI header file `mpi.h`: defines MPI constants, all function prototypes
- No MPI calls possible before MPI_Init(); no MPI calls possible after MPI_Finalize(); MPI cannot be restarted
- MPI_Init(): initializes internal data structures
- MPI_Finalize(): cleans up internal data structures, but does not free application created MPI objects
- MPI_Init() and MPI_Finalize() are collective, all started processes must call

Two exceptions:

```
MPI_Initialized(int *flag)
MPI_Finalized(int *flag)
```

For layered libraries

A word on error handling

Almost all MPI functions return an error code, ≠MPI_SUCCESS means something wrong or unexpected happened

Good practice always to check the error code:

```
…
err = MPI_Comm_size(MPI_COMM_WORLD,&size);
if (err!=MPI_SUCCESS) {
  // do something
  …
}
err = MPI_Comm_rank(MPI_COMM_WORLD,&rank);
```

Most often, this is not done… (errors mostly leads to abort)

Depending on the MPI library implementation, it is sometimes possible to do something sensible if errors occur, see standard on error handlers

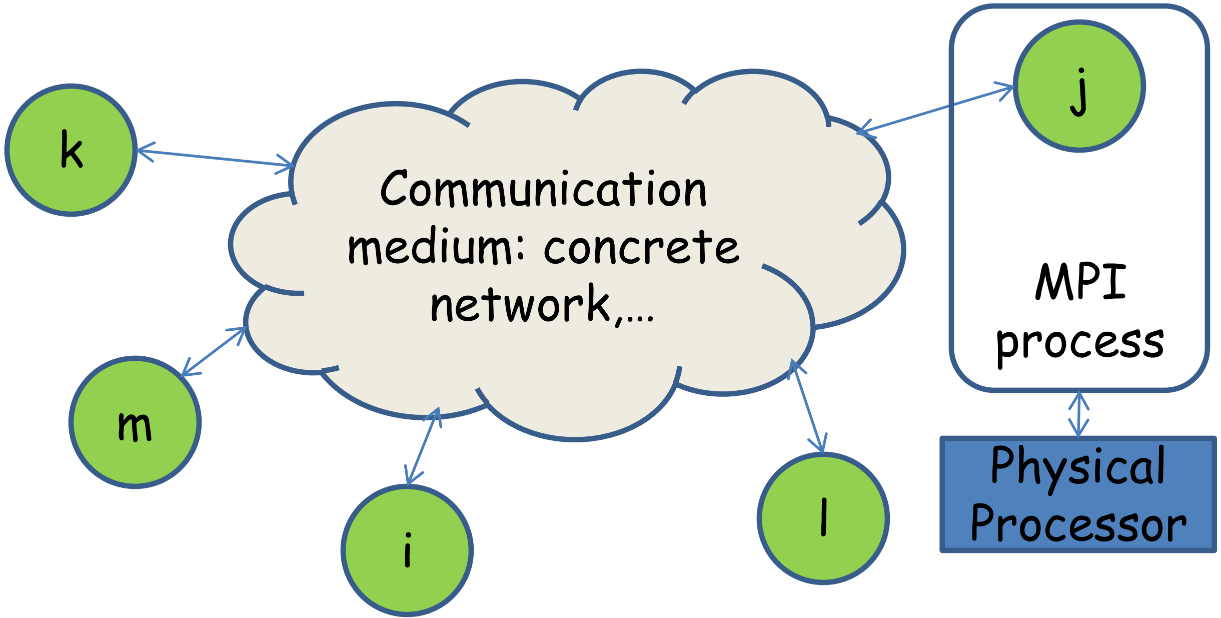
Note:
„text that states that errors *will* be handled, should be read as *may* be handled", MPI 3.1, Section 8.3, p. 340

MPI is designed for high performance. It is user responsibility to write correct programs

First MPI communicator, first MPI operations

```
#include <mpi.h>

int main(int argc, char *argv[])
{
  int rank, size;

  MPI_Init(&argc,&argv);

  MPI_Comm_size(MPI_COMM_WORLD,&size);
  MPI_Comm_rank(MPI_COMM_WORLD,&rank);

  // …

  MPI_Finalize();
  return 0;
}
```

After MPI_Init():
MPI processes have a rank in MPI_COMM_WORLD



MPI processes (mostly) statically bound to physical processor/core. Do not migrate (across node boundaries)

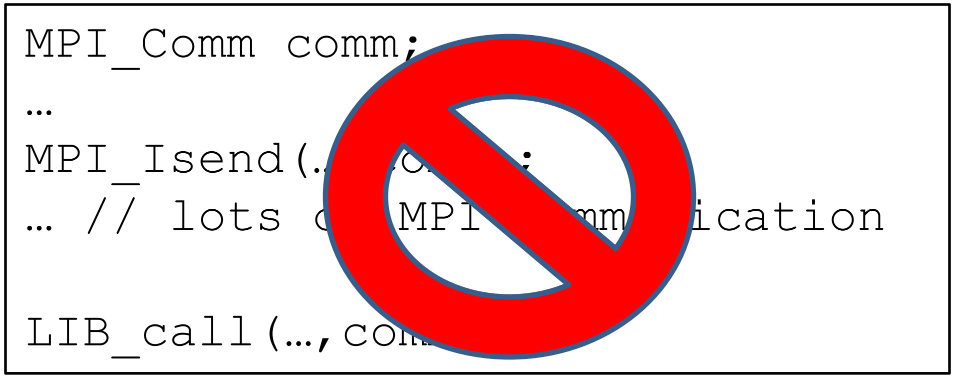
MPI_COMM_WORLD

Initial communicator created by MPI_Init(), contains all started processes

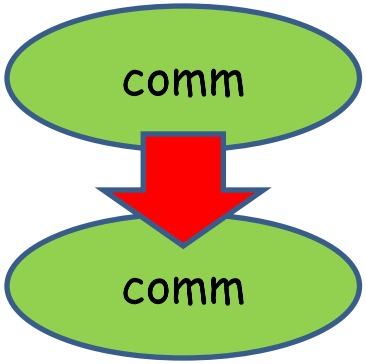
Communicator:

- Fundamental MPI abstraction (object) representing an ordered set (group) of processes that can communicate
- All communication operations are wrt. a communicator: only processes in the same communicator can communicate
- A communicator has a size: the number of processes
- Each process in communicator uniquely identified by a rank, $0 \leq \text{rank} < \text{size}$ in communicator
- Processes can belong to many communicators

Communicators are the fundamental mechanism for encapsulation and library building



```
MPI_Comm comm;
…
MPI_Isend(.  co    ;
… // lots o  MPI   mm  ication

LIB_call(…,co
```

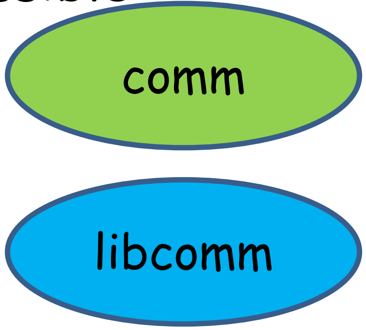


LIB:
MPI_Irecv(…, comm);

Communicators are the fundamental mechanism for encapsulation and library building

```
MPI_Comm comm, libcomm;

// initialize LIB
MPI_Comm_dup(comm,&libcomm);
…
MPI_Isend(…,comm);
… // lots of MPI communication

LIB_call(…,libcomm);
```

Communication between the two different communicators not possible



```
MPI_Comm_dup(MPI Comm oldcomm, MPI_Comm *newcomm)
MPI_Comm_dup(MPI Comm comm, MPI_Comm *newcomm)
```

is a blocking, collective operation: must be called by all processes in comm, upon return, new communicator has been created

Communicators are global, distributed objects, can only be manipulated by collective operations

- MPI_Comm_dup()
- MPI 3.1: MPI_Comm_dup_with_info(), MPI_Comm_idup()
- MPI_Comm_split()
- MPI 3.0: MPI_Comm_split_type()
- MPI_Comm_create()
- MPI 3.1: MPI_Comm_create_group()

Many other operations implicitly create new communicators

```
MPI_Comm_free(MPI_Comm *comm)
```

Collective operation for freeing a communicator, call returns MPI_COMM_NULL if successful

Good practice always to free created MPI objects after use

Observations:

- New communicators can be created out of old ones. Processes can only be removed, not added (dynamic process management is not treated here)
- Communicators never change
- Operations on communicators are all collective

Principle 4:

MPI objects (communicators, groups, windows, datatypes, requests, operators, …) are opaque, static and immutable

- It is (mostly) not possible to look into the MPI objects (partial exception: MPI_Status), objects are manipulated through operations (local and collective)
- New objects can be created out of old ones
- There are NULL objects (invalid object, returned when objects is freed) for most MPI objects

Collective semantic rules

- A collective operation (MPI_Comm_dup(…,comm), MPI_Bcast(…,comm), …) must be called by all processes in communicator (collective over comm, or group)

- Correctness: if some process in communicator calls collective operation A, then all other processes (in this communicator) must also call A, and no other collective operation (on communicator) inbetween

- Operation is blocking if operation is (locally) completed upon return: communicator has been created, messages sent, buffers can be reused…
- Operation has non-local completion semantics, if completion depends on action by other processes

Process i:

```
MPI_Comm_dup(comm,&newcomm);
MPI_Bcast(…,comm);
```

Process j:

```
MPI_Bcast(…,comm);
MPI_Comm_dup(comm,&newcomm);
```

is erroneous: Processes do not call collectives on comm in the same order

What happens?

- May deadlock (likely)
- May crash (also possible)
- May "work" (sometimes) – with disastrous consequences later

MPI is designed for high performance, it is user responsibility to write correct programs

Process i:

```
MPI_Comm_dup(comm,&newcomm);
MPI_Bcast(…,comm);
```

🚫

Process j:

```
MPI_Bcast(…,comm);
MPI_Comm_dup(comm,&newcomm);
```

is erroneous: Processes do not call collectives on comm in the same order

Correctness tools can help, check your local MPI installation

[Jesper Larsson Träff, Joachim Worringen: Verifying Collective MPI Calls. PVM/MPI 2004: 18-27]
[Christopher Falzone, Anthony Chan, Ewing L. Lusk, William Gropp: A Portable Method for Finding User Errors in the Usage of MPI Collective Operations. IJHPCA 21(2): 155-165 (2007)]

Process i:

```
MPI_Bcast(…,comm);
MPI_Comm_dup(comm,&newcomm);
```

Process j:

```
MPI_Bcast(…,comm);
MPI_Comm_dup(comm,&newcomm);
```

is correct: Processes now call collectives on comm in the same order

Local representations of MPI processes: groups

```
MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

Get the process group (rank ordered set of processes) out of the communicator. Local-completion operations for creating new groups out of old ones include

•MPI_Group_size(), MPI_Group_rank()
MPI_Group_translate_ranks()
•MPI_Group_compare()

•MPI_Group_union(), MPI_Group_intersection(),
MPI_Group_difference()
•MPI_Group_incl(), MPI_Group_excl()
•MPI_Group_range_incl(), MPI_Group_range_excl()

```
MPI_Comm_group(MPI_Comm comm, MPI_Group *group)
```

```
MPI_Group_free(MPI_Group *group)
```

Groups needed for one-sided communication (and communicator creation)

As with communicators, only possible to restrict/reorder the set of processes in group, not to add new processes

Example: 2d stencil computation

Solving Poisson Equation on the unit square [0,1]x[0,1]

$$\delta^2u(x,y)/\delta x^2 + \delta^2u(x,y)/\delta y^2 = f(x,y)$$

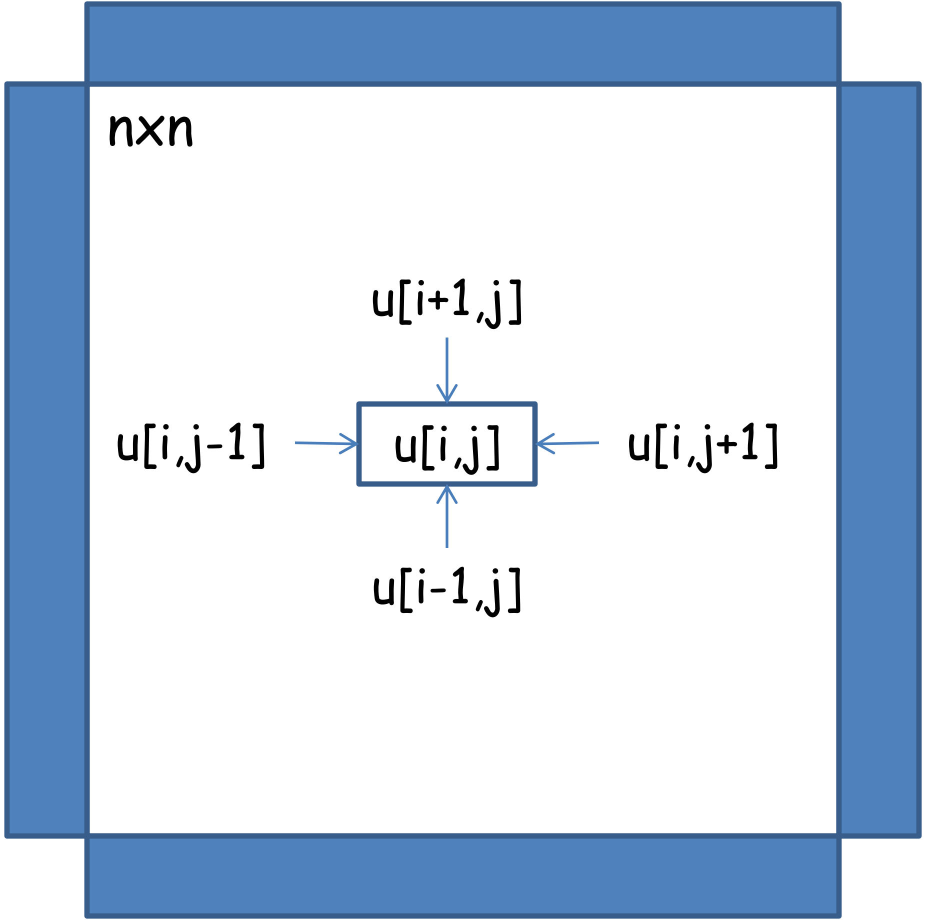
$$u(x,y) = g(x,y) \text{ on the boundary}$$

boils down to iterating

$$u^{k+1}[i,j] = \tfrac{1}{4}(u^k[i+1,j]+u^k[i-1,j]+u^k[i,j+1]+u^k[i,j-1] - h^2f(i/n,j/n)$$

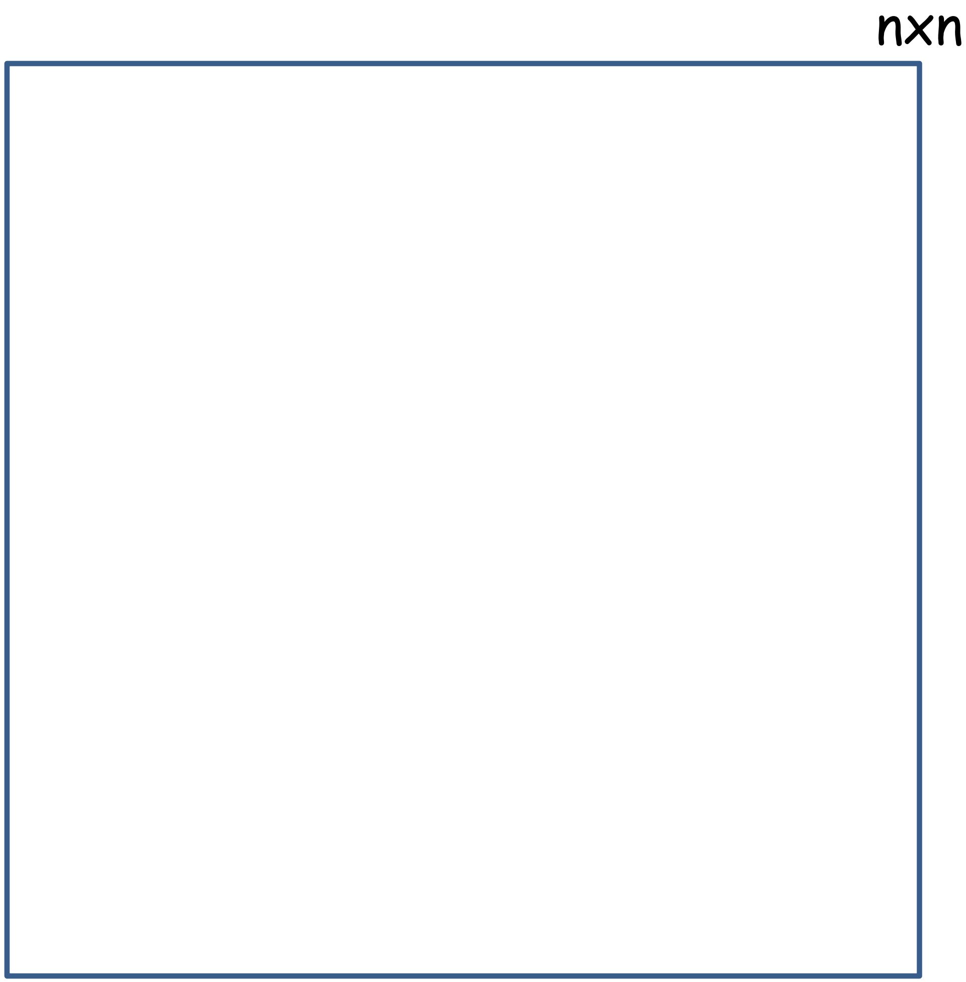
on an nxn matrix (n large).

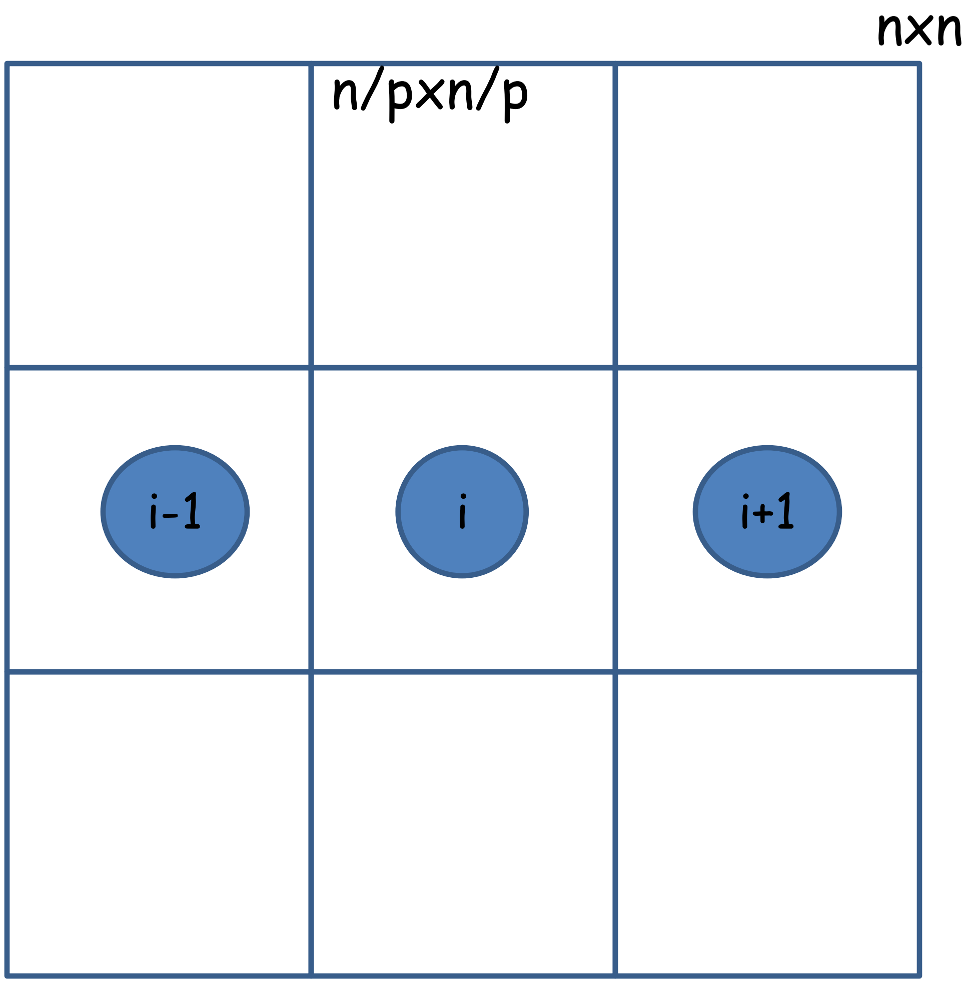
Updates follow a regular 5-point stencil pattern



Boundary conditions

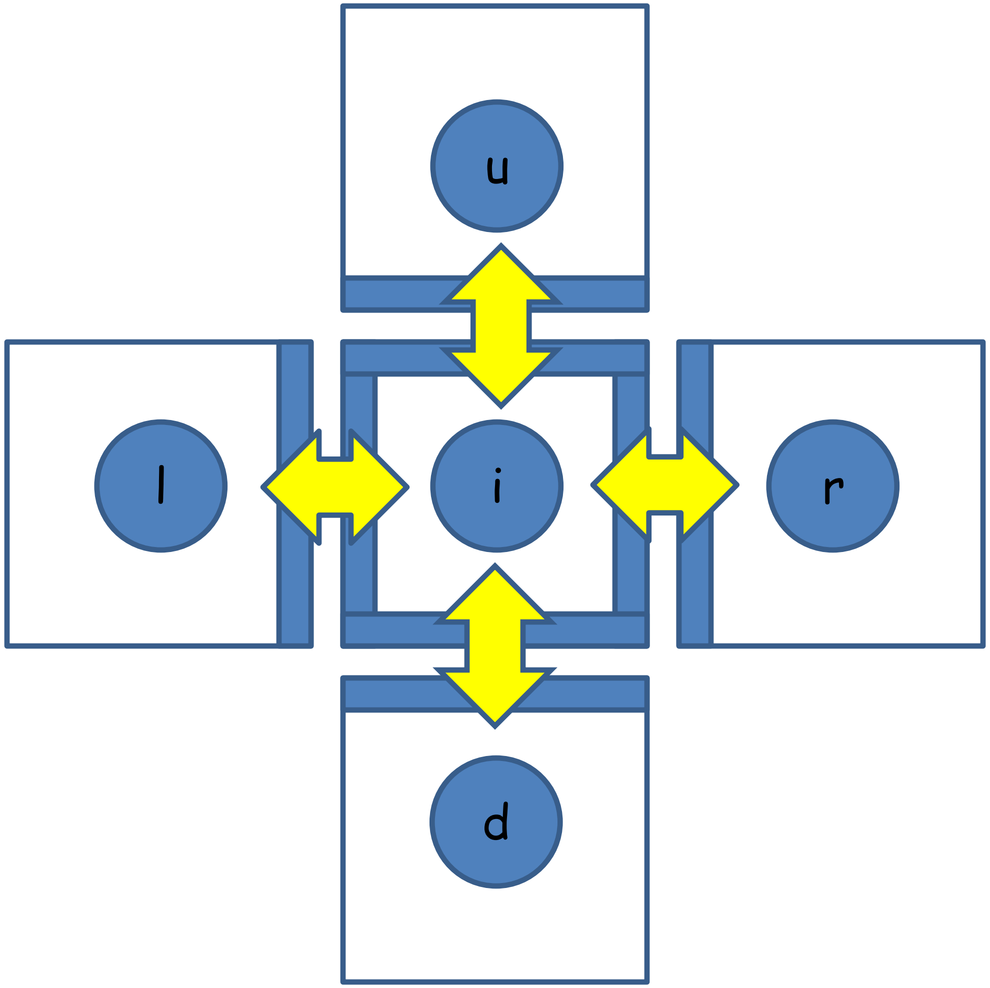
Typical parallelization with p (some square) processes:
2d regular decomposition


nxn



MPI process assigned to each n/p x n/p block, updates locally on block

Assume p divides n, use regular decomposition to minimize surface (communication) to volume (work) ratio



Before next iteration, each MPI process needs to exchange boundary row/column with 4 neighboring processes

Problem 1: Identifying neighboring processes

Processes in communicator are ordered linearly, 0, 1, … size-1; need to find rank of neighboring processes in virtual mesh

Solution 1: by hand, chose row or column major order of processes, do the calculations…

Solution 2: Use MPI

Process 0 coordinate (0,0)	Process 1 coordinate (0,1)	Process 2 coordinate (0,2)
Process 3 coordinate (1,0)	Process 4 coordinate (1,1)	Process 5 coordinate (1,2)
Process 6 coordinate (2,0)	Process 7 coordinate (2,1)	Process 8 coordinate (2,2)

Virtual Cartesian topology:
MPI processes organized by row order numbering into d-dimensional mesh/torus.

Support functions for translating between ranks and coordinates

```
MPI_Cart_create(MPI Comm comm, int ndims,
                const int dims[], const int periods[],
                int reorder, MPI_Comm *cartcomm)
```

Creates new communicator, where processes have been organized into mesh (periods[i]=0) or torus (periods[i]=1) of ndims dimesions; dimension i has size dims[i]

$\prod_{i=0}^{ndims-1} dims[i] \leq size(comm)$

If there are too many processes in comm, some processes will not be in Cartesian communicator, cartcomm=MPI_COMM_NULL

Application decides on best dimension sizes (often close to squareroot of p, but depends…), before calling MPI_Cart_create()

MPI provides some helper functionality to suggest a good factorization of p (nodes) i dims) factors

```
MPI_Dims_create(in nod ir ndims, int dims[])
```

Beware: current implementations are not good at all (and not really suited for modern, hierarchical systems)

[Jesper Larsson Träff, Felix Donatus Lübbe: Specification Guideline Violations by MPI_Dims_create. EuroMPI 2015: 19:1-19:2]

```
MPI_Cart_shift(MPI_Comm cartcomm,
               int direction, int displacement,
               int *source, int *destination)
```

Find ranks of neighbors in dimension direction

```
MPI_Cart_rank(MPI_Comm cartcomm, int coords[],
              int *rank)
```

Translates d-dimensional coordinate vector into rank

```
MPI_Cart_coords(MPI_Comm cartcomm, int rank, int
maxdims, int coords[])
```

Translates rank into d-dimensional coordinate

```
MPI_Comm comm, cartcomm;

MPI_Comm_size(comm,&p);
r = sqrt(p); c = p/r; // or try MPI_Dims_create
dim[0] = c; dim[1] = r;
period[0] = 0; period[1] = 0;

reorder = 0;
MPI_Cart_create(comm,2,dim,period,reorder,&newcomm);

Int r, l, u, d; // the ranks of the 4 neighbors
MPI_Cart_shift(newcomm,0,1,&r,&l);
MPI_Cart_shift(newcomm,1,1,&u,&d);
```

Topology is a (non-periodic) mesh, neighbors on border are MPI_PROC_NULL (special rank that can be used in point-to-point communication to sometimes avoid handling special cases)

```
MPI_Comm comm, cartcomm;

MPI_Comm_size(comm,&p);
r = sqrt(p); c = p/r; // or try MPI_Dims_create
dim[0] = c; dim[1] = r;
period[0] = 0; period[1] = 0;

reorder = 0;
MPI_Cart_create(comm,2,dim,period,reorder,&newcomm);

Int r, l, u, d; // the ranks of the 4 neighbors
MPI_Cart_shift(newcomm,0,1,&r,&l);
MPI_Cart_shift(newcomm,1,1,&u,&d);
```

What is reorder? What if reorder = 1; ?

Virtual topologies

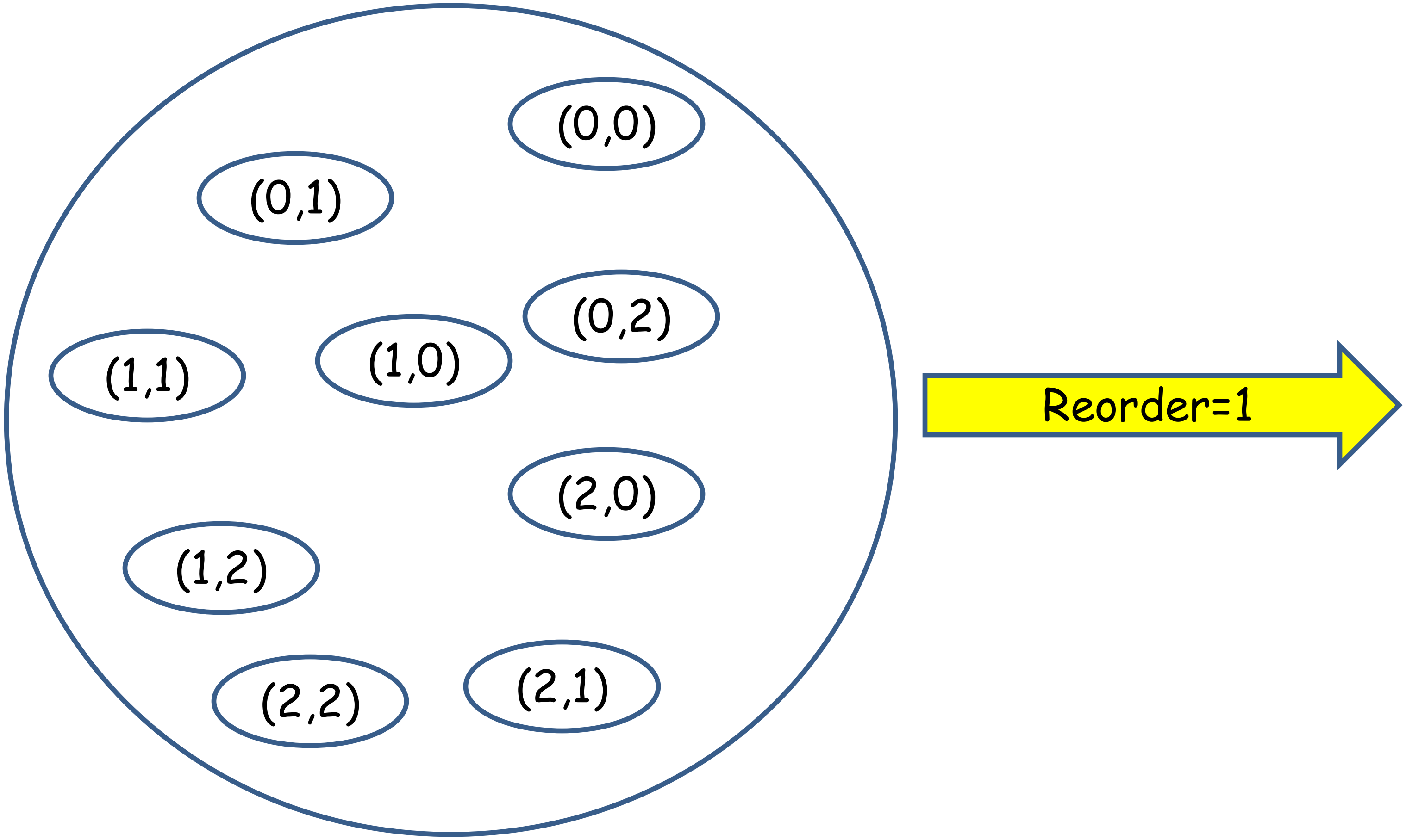
A virtual process topology describes a pattern of most likely or most significant communication between MPI processes:

- Cartesian: MPI processes likely to communicate along the dimensions of d-dimensional mesh/torus
- (Distributed) Graph: pattern described as a directed graph

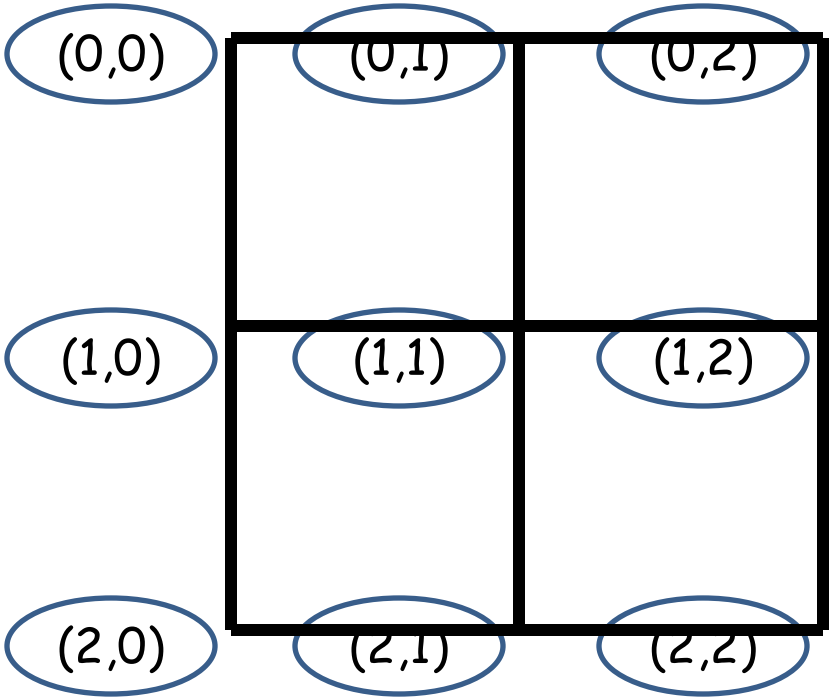
`reorder=1`:

MPI may attempt to reorder processes in new communicator, such that neighbors in virtual topology are close in physical communication network

Communicator with some mapping of ranks to physical processors/cores



…may be mapped to match physical topology



Physical 2d, 3d, 5d torus (was: BlueGene, Cray, etc.; is: Fujitsu K-Computer)

Does it work? Do MPI libraries reorder virtual topologies?

Rarely, but try…

1. Graph embedding is an NP-hard optimization problem
2. Not obvious what the objectives really are

Data redistribution may be necessary:

- For each rank i in old comm, find out where i is in cartcomm
- Send data (from old i to new i in either of communicator)
- MPI does not do data redistribution, user responsibility

But MPI can help:

```
assert(cartcomm!=MPI_COMM_NULL);

int torank, fromrank; // to be computed
int rank, cartrank;

MPI_Group group, cartgroup;
MPI_Comm_group(comm,&group);
MPI_Comm_group(cartcomm,&cartgroup);

// where has rank been mapped to?
MPI_Comm_rank(comm,&rank); // rank in old comm
MPI_Group_translate_rank(cartcomm,1,&rank,comm,
                         &torank);
// torank may be MPI_UNDEFINED
// if cartcomm smaller than comm

MPI_Comm_rank(cartcomm,&cartrank);
MPI_Group_translate_ranks(cartcomm,1,&cartrank,comm,
                          &fromrank);
```

```
MPI_Group_translate_ranks(MPI_Group group1,
                          int n, const int ranks1[],
                          MPI_Group group 2,
                          int ranks2[])
```

Translate list of ranks of processes in group1 into ranks in group2

Virtual topologies

A communicator with a virtual topology is still a fully general MPI communicator

- All processes can communicate, point-to-point, collective, one-sided
- Communicator can be used to create further, new communicators

Virtual topology does not forbid communication (but certain communication may be preferred, more efficient)

Query functionality (for library building)

```
MPI_Topo_test(MPI_Comm comm, int *topotype)
```

The returned `topotype` can be

- MPI_CART: The communicator is Cartesian
- MPI_GRAPH: The communicator is a (non-scalable) graph
- MPI_DIST_GRAPH: The communicator is a distributed graph
- MPI_UNDEFINED: The communicator has no virtual topology

The topotype is (implementation wise) stored as an attribute of the communicator:

Additional query functionality (similar for distributed graphs)

```
MPI_Cartdim_get(MPI_Comm cartcomm, int *ndims)
```

```
MPI_Cart_get(MPI_Comm cartcomm,
             int maxdims,
             int dims[], int periods[], int coords[])
```

maxdims is the size of the arrays in the call, only this much information is returned

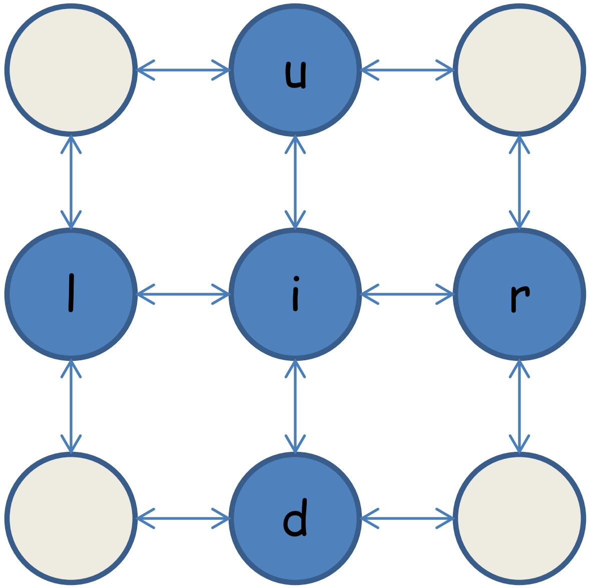
User defined attributes

For library building

Possible to associate own, user-defined attributes with MPI objects like communicators, datatypes, and windows

MPI mechanism is somewhat awkward (generate key, write copy and deletion functions), check the standard

Solution 1: Point-to-point communication



Iterate:

```
MPI_Send(…,u,…,comm);
MPI_Send(…,d,…,comm);
MPI_Send(…          );
MPI_Send(      ,comm

MPI_Recv( u,…,c      stat);
MPI_Recv(…,       ,&stat);
MPI_Recv(…,l,…,comm,&stat);
MPI_Recv(…,r,…,comm,&stat);
// update
```

May or may not deadlock! Depends on implementation and size of data sent

```
MPI_Send(void *buf, int count, MPI_Datatype type,
         int destination, int tag, MPI_Comm comm )
```

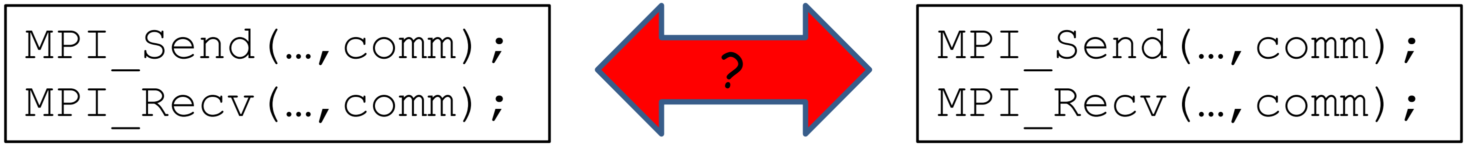
- Blocking send: when call returns, data have left buffer, `buf` can be reused
- Non-local completion semantics: completion may depend on action by receiving process
- Return from call does not imply anything about action by receiving process

MPI library implementation practice (for high performance):
- Short messages are internally buffered
- Medium messages handled by special protocol
- Long messages pipelined, rendezvous protocol

MPI library implementation practice (for high performance):

- Short messages are internally buffered
- Medium messages handled by special protocol
- Long messages pipelined, rendezvous protocol

This practice is NOT prescribed by MPI standard, how it is done is implementation dependent, and differ among libraries and systems



```
MPI_Send(…,comm);
MPI_Recv(…,comm);
```

```
MPI_Send(…,comm);
MPI_Recv(…,comm);
```

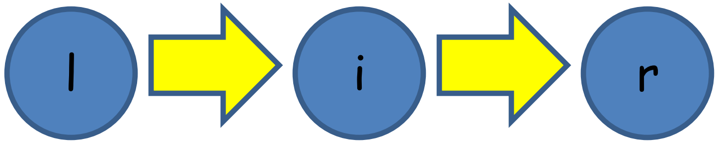
Exchange may work, but is implementation dependent. Unsafe!

Definition: MPI program is unsafe if termination depends on whether messages are internally buffered.

- An unsafe program may or may not deadlock
- Behavior is dependent on MPI library implementation (how large internal buffers are allowed) and perhaps on concrete context (how may processes, which communication)
- Unsafe programs are not portable

Remedy: MPI_Sendrecv()

```
MPI_Sendrecv(…,u,…,d,…,comm);
MPI_Sendrecv(…,d,…,u,…,comm);
MPI_Sendrecv(…,l,…,r,…,comm);
MPI_Sendrecv(…,r,…,l,…,comm);
// update
```



Remedy: Non-blocking communication

```
MPI_Request request[8];
MPI_Status statuses[8];

MPI_Isend(…,u,…,comm,&request[0]);
MPI_Isend(…,d,…,comm,&request[1]);
MPI_Isend(…,l,…,comm,&request[2]);
MPI_Isend(…,r,…,comm,&request[3]);

MPI_Irecv(…,u,…,comm,&request[4]);
MPI_Irecv(…,d,…,comm,&request[5]);
MPI_Irecv(…,l,…,comm,&request[6]);
MPI_Irecv(…,r,…,comm,&request[7]);

MPI_Waitall(8,request,statuses);
// update
```

Non-blocking communication

Semantic advantages: makes it easier to avoid deadlocks, and make programs safe. Non-blocking collectives with MPI 3.0

Performance advantage(?): makes it possible to overlap communication with computation

```
MPI_Isend(void *buf, int count, MPI_Datatype type,
          int destination, int tag, MPI_Comm comm,
          MPI_Request *request)
```

Non-blocking send: local completion semantics, call returns Immediately, but buffer is still in use

```
MPI_Isend(void *buf, int count, MPI_Datatype type,
          int destination, int tag, MPI_Comm comm,
          MPI_Request *request)
```

Non-blocking send: local completion semantics, call returns Immediately, but buffer is still in use

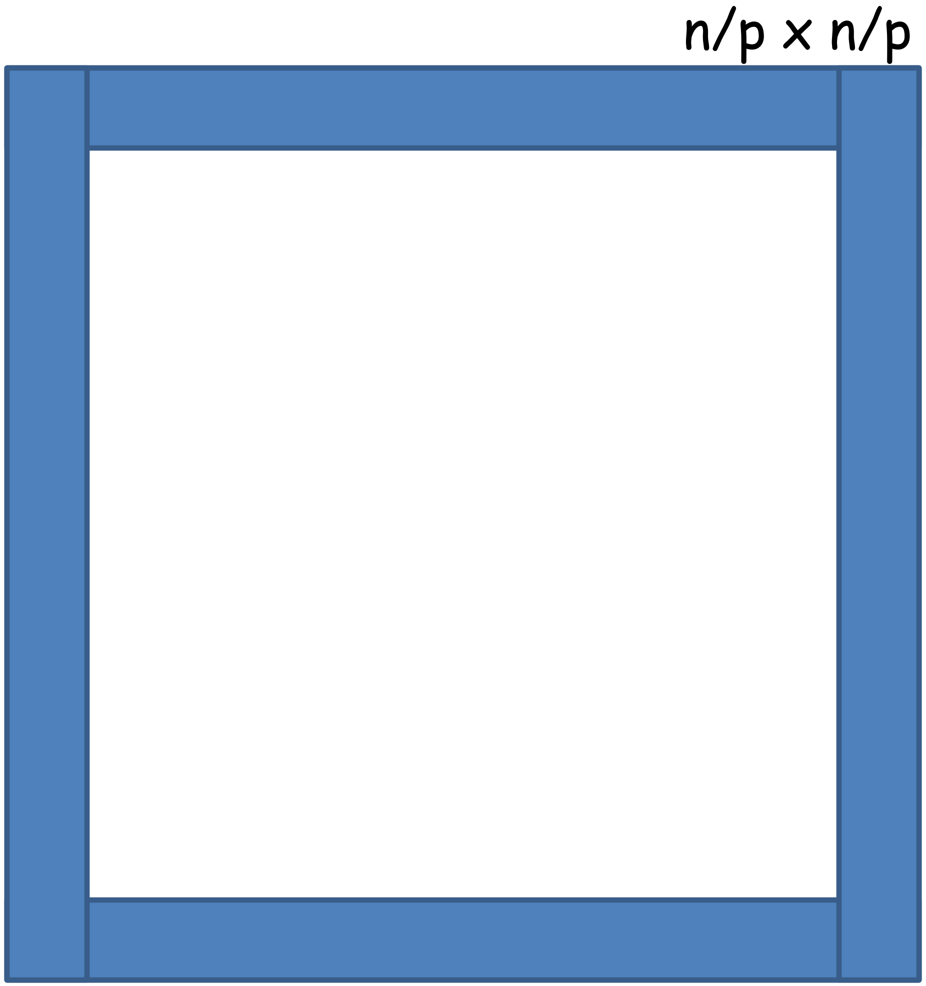
Effect as MPI_Send() achieved by MPI_Wait() call, buffer free, but no guarantees about what has happened at receiving side

- MPI_(I)Ssend(): Synchronous send, returns when receiver has started reception
- MPI_(I)Bsend(): Buffered send, local completion semantics, data buffered in attached buffer in user space

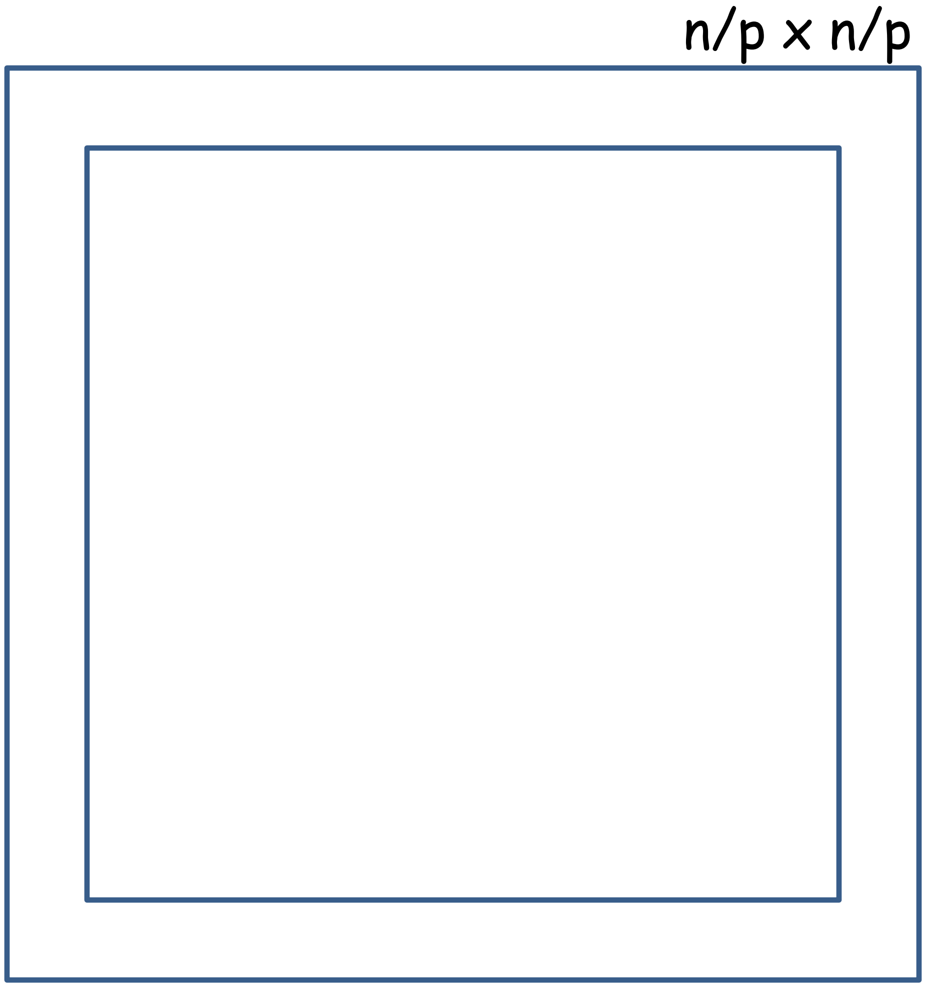
```
MPI_Request request[8];
MPI_Status statuses[8];

MPI_Isend(…,u,…,comm,&request[0]);
MPI_Isend(…,d,…,comm,&request[1]);
MPI_Isend(…,l,…,comm,&request[2]);
MPI_Isend(…,r,…,comm,&request[3]);

MPI_Irecv(…,u,…,comm,&request[4]);
MPI_Irecv(…,d,…,comm,&request[5]);
MPI_Irecv(…,l,…,comm,&request[6]);
MPI_Irecv(…,r,…,comm,&request[7]);
// update non-border region
MPI_Waitall(8,request,statuses);
// update border
```

Communication has been initiated

Do computation here

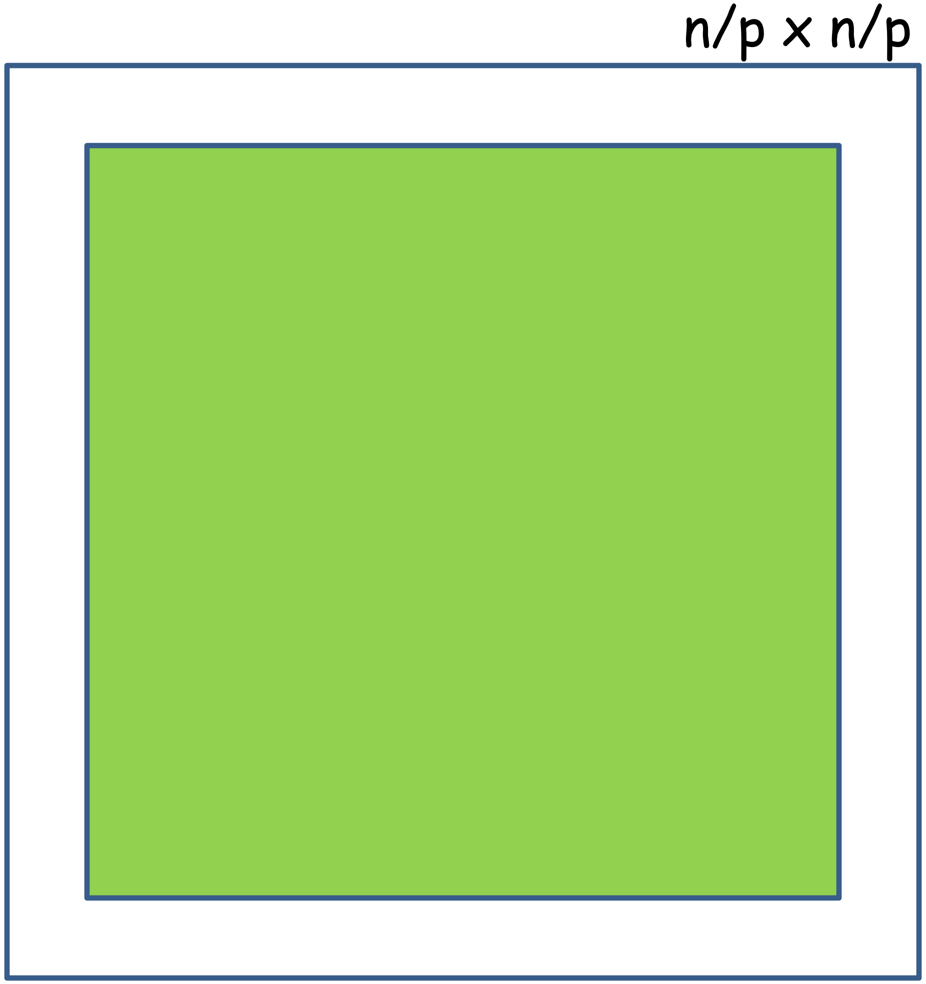


1. Initiate communication on boundary



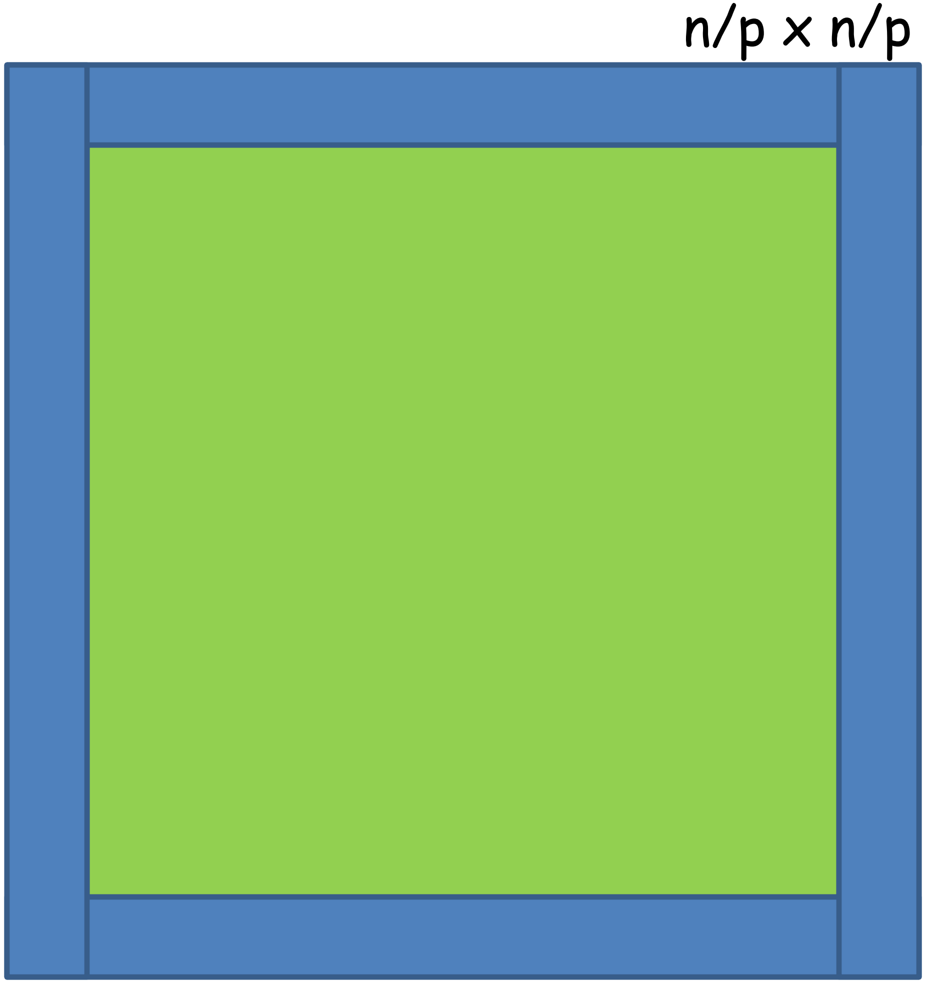
1. Initiate communication on boundary

2. Compute on internal elements, as long as there are no dependencies on boundary elements



1. Initiate communication on boundary

2. Compute on internal elements, as long as there are no dependencies on boundary elements

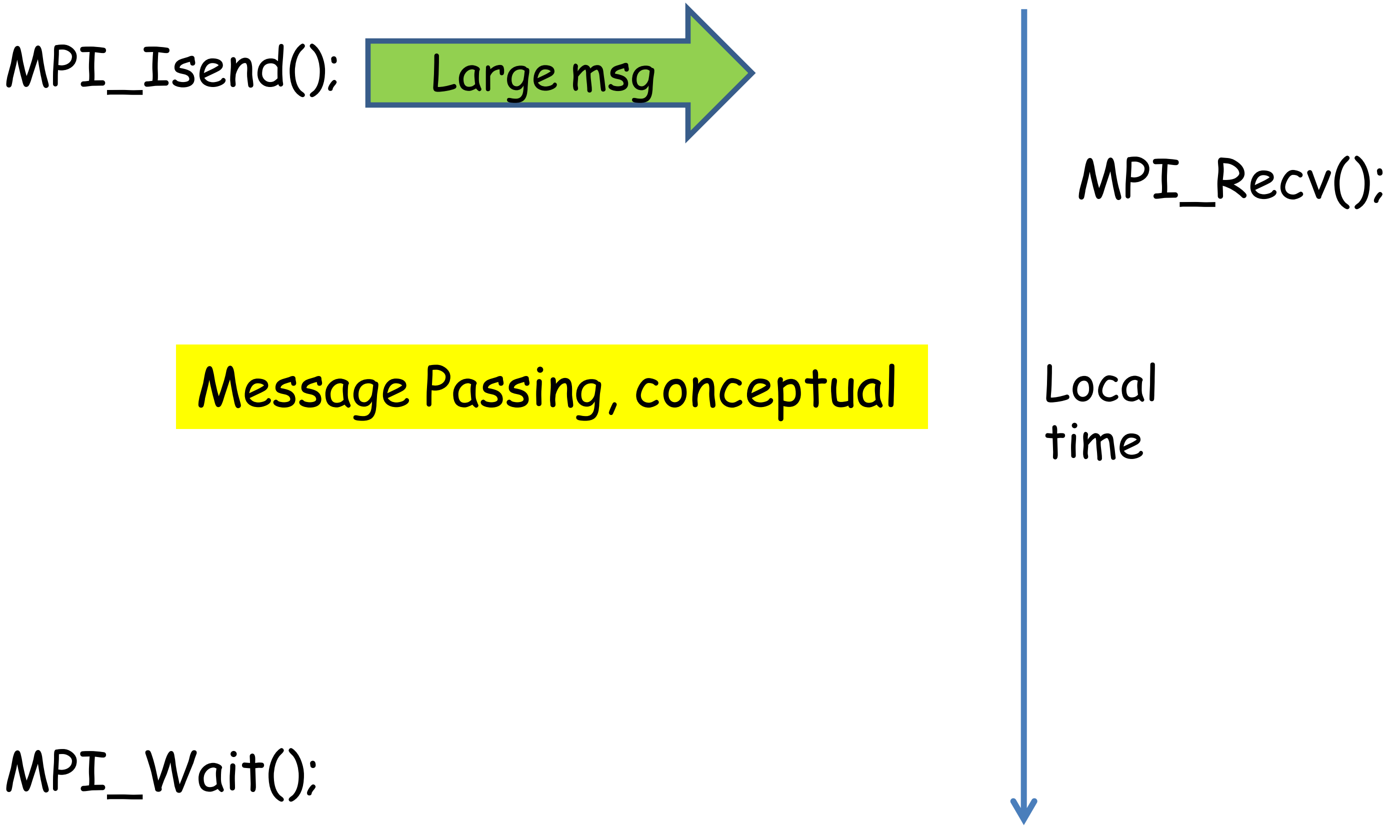


1. Initiate communication on boundary

2. Compute on internal elements, as long as there are no dependencies on boundary elements

3. Complete communication, complete computation on boundary

On progress of communication



MPI_Isend();

Large msg

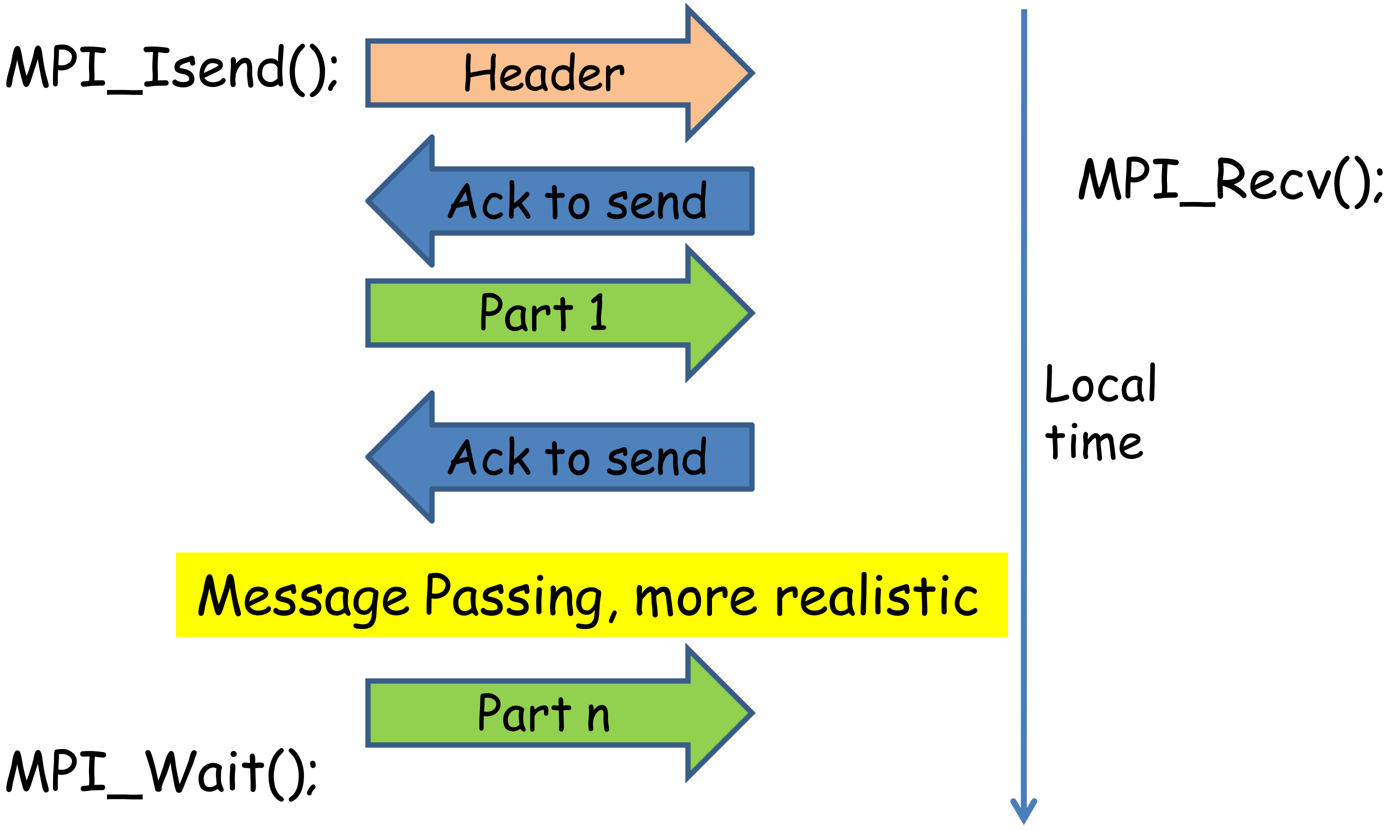
MPI_Recv();

Message Passing, conceptual

Local time

MPI_Wait();

Message Passing, more realistic



MPI_Isend();

Header

Ack to send

Part 1

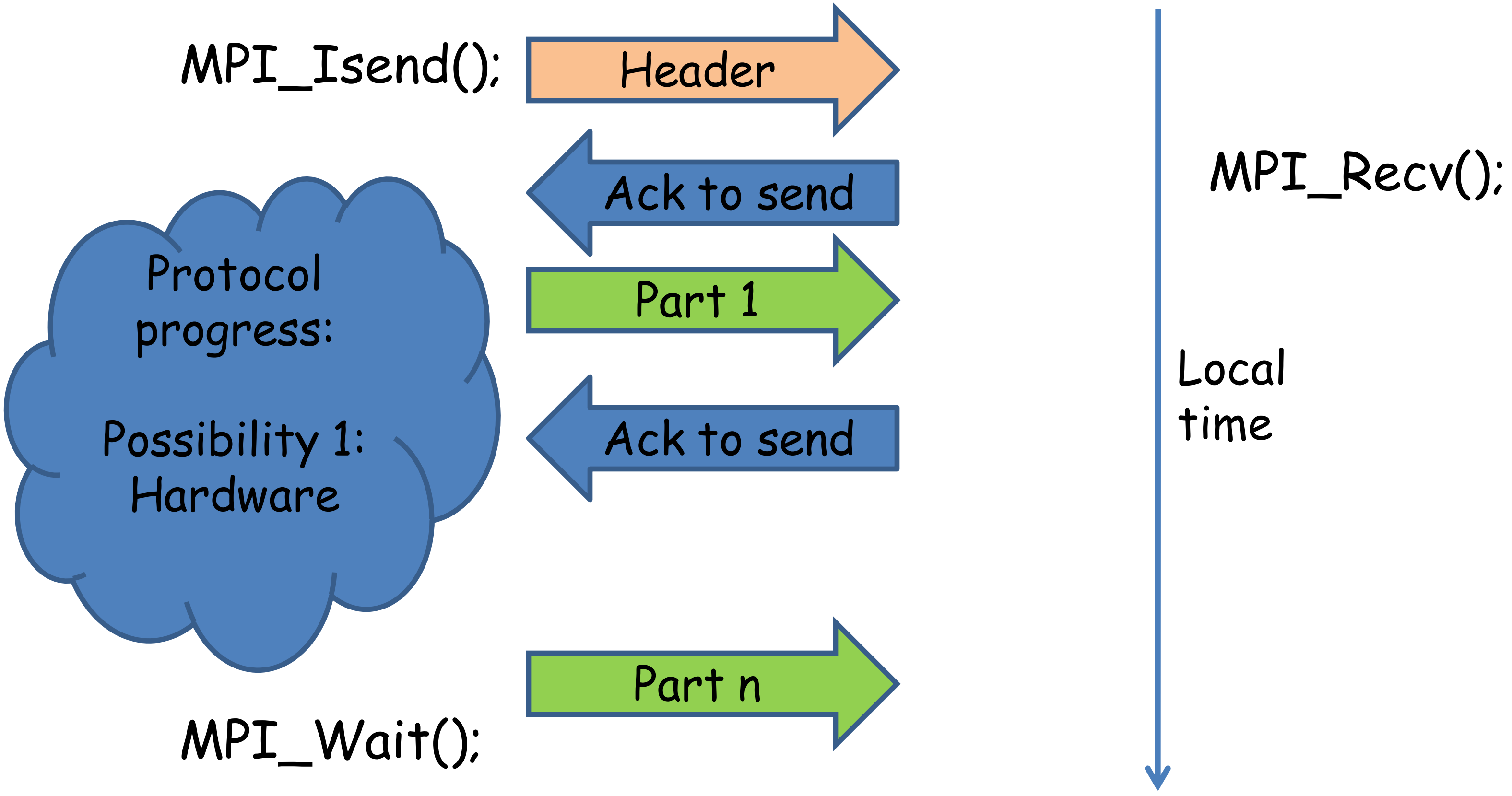
Ack to send

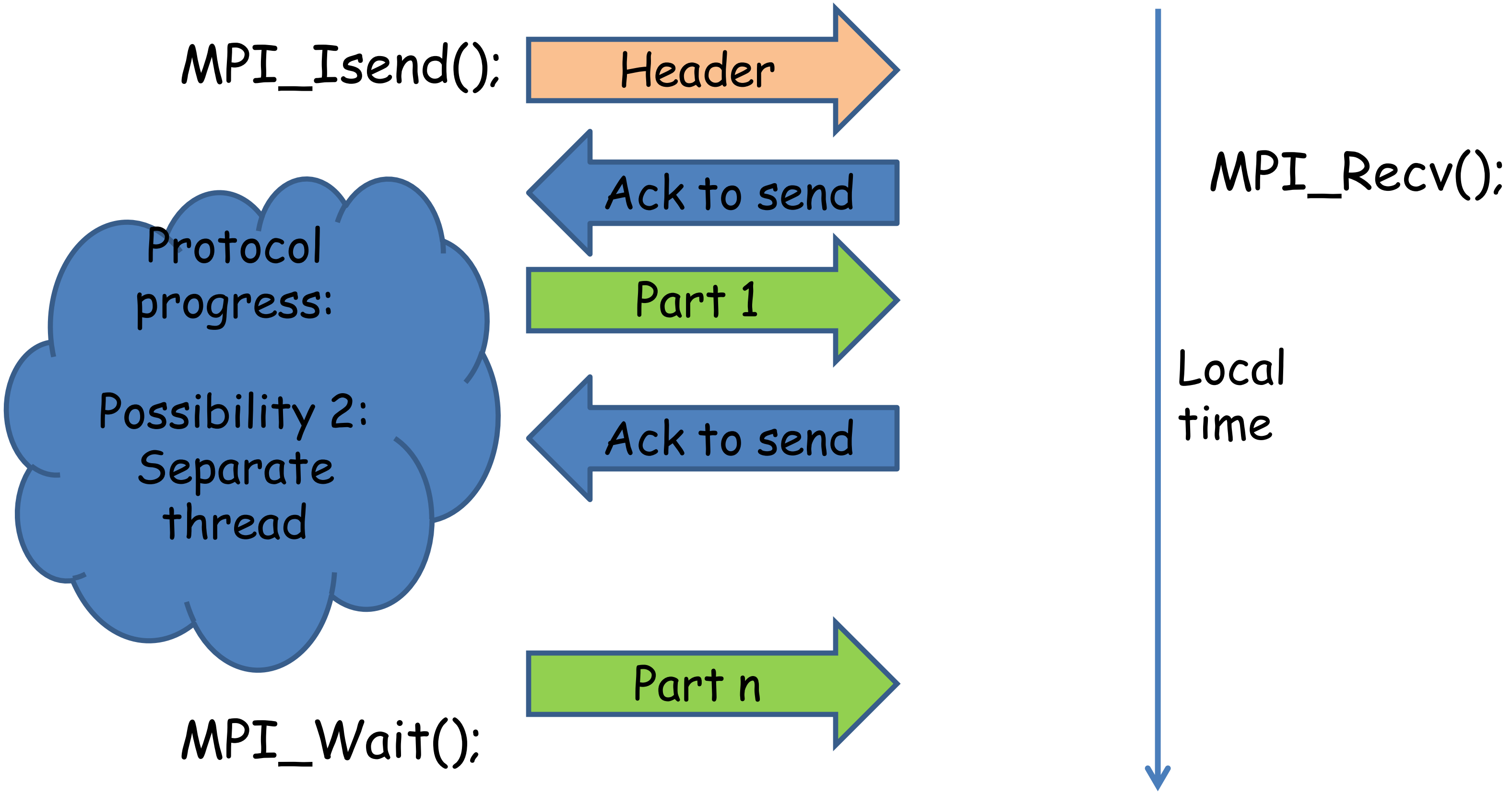
Part n

MPI_Wait();

MPI_Recv();

Local
time


MPI_Isend();
Header
Ack to send
Protocol progress:
Possibility 1: Hardware
Part 1
Ack to send
Part n
MPI_Wait();
MPI_Recv();
Local time



MPI_Isend();

Header

Ack to send

Part 1

Ack to send

Part n

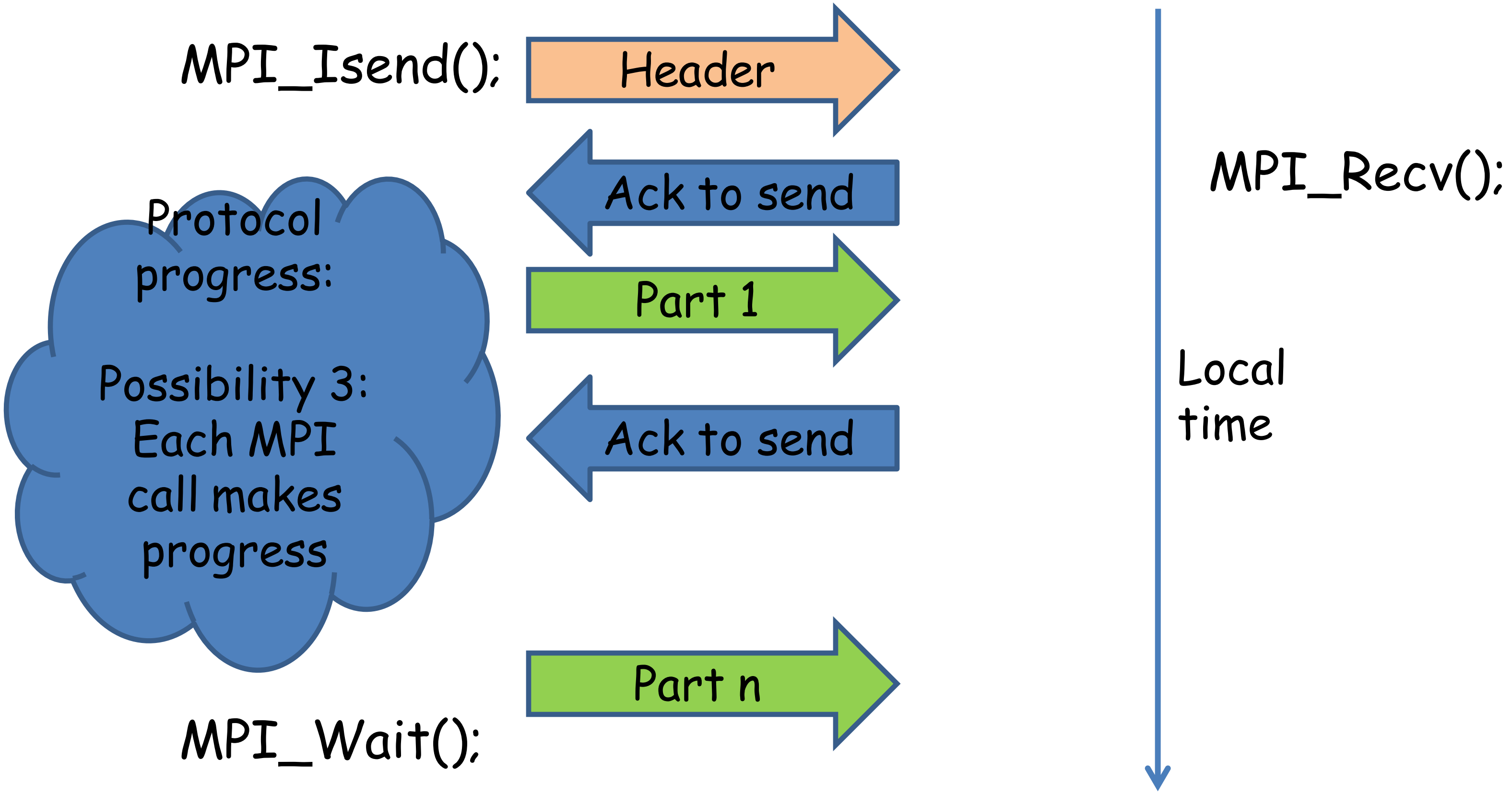
MPI_Wait();

Protocol progress:

Possibility 2: Separate thread

MPI_Recv();

Local time



MPI_Isend();

Header

Ack to send

Part 1

Ack to send

Part n

MPI_Wait();

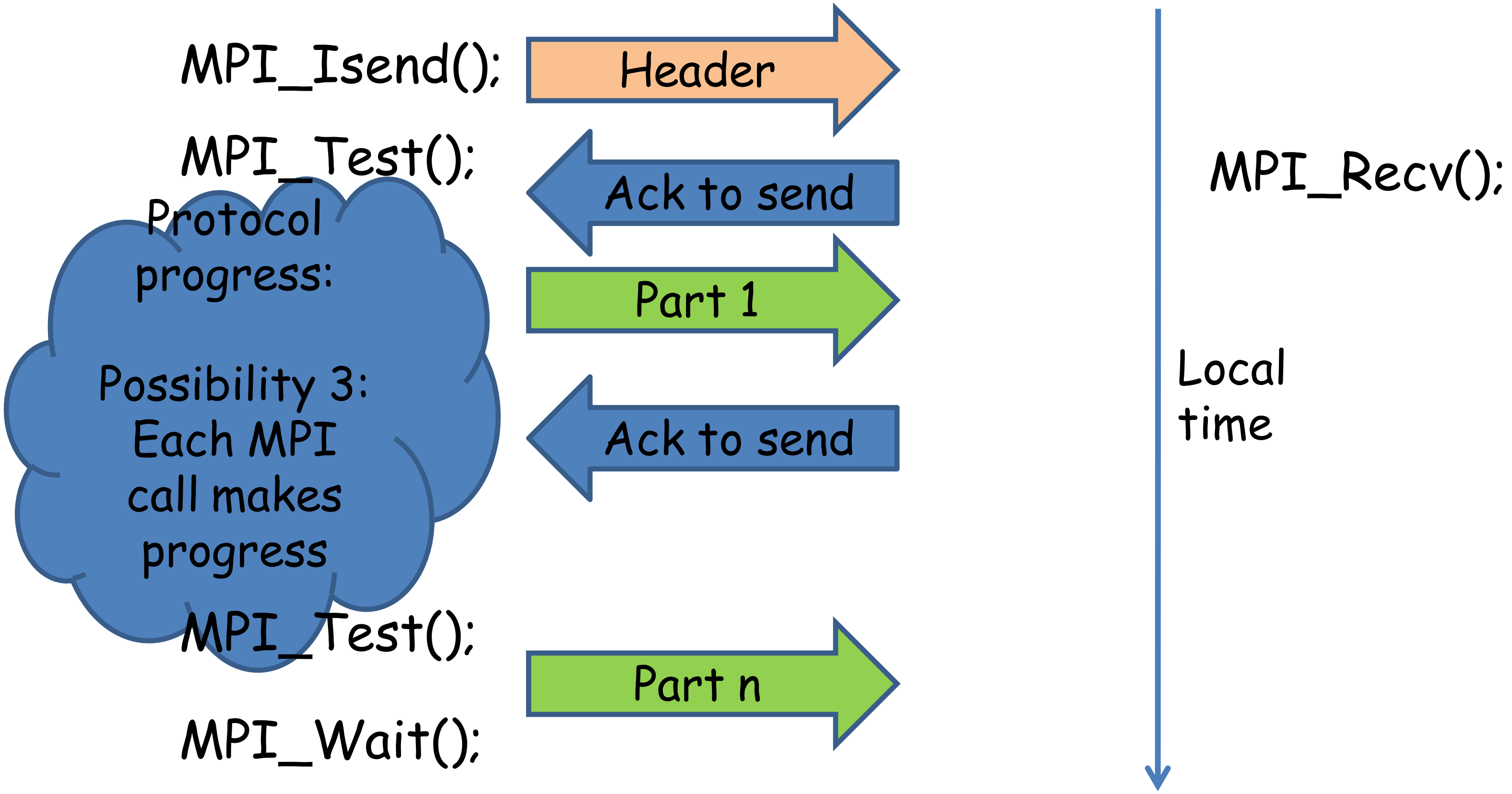
Protocol progress:

Possibility 3: Each MPI call makes progress

MPI_Recv();

Local time

Application enforced progress: difficult to tune, may be non-portable, but I sometimes (often?) necessary



Progress

MPI libraries often use mixed strategies:

1. Hardware, whenever possible („offload to NIC")
2. MPI calls to make progress
3. Sometimes thread support („progress thread")

Note:
Thread support sometimes considered too expensive for HPC, sometimes not possible (because of simple OS)

Good practice: frequent MPI calls when using non-blocking operations; but difficult to tune, possibly not portable

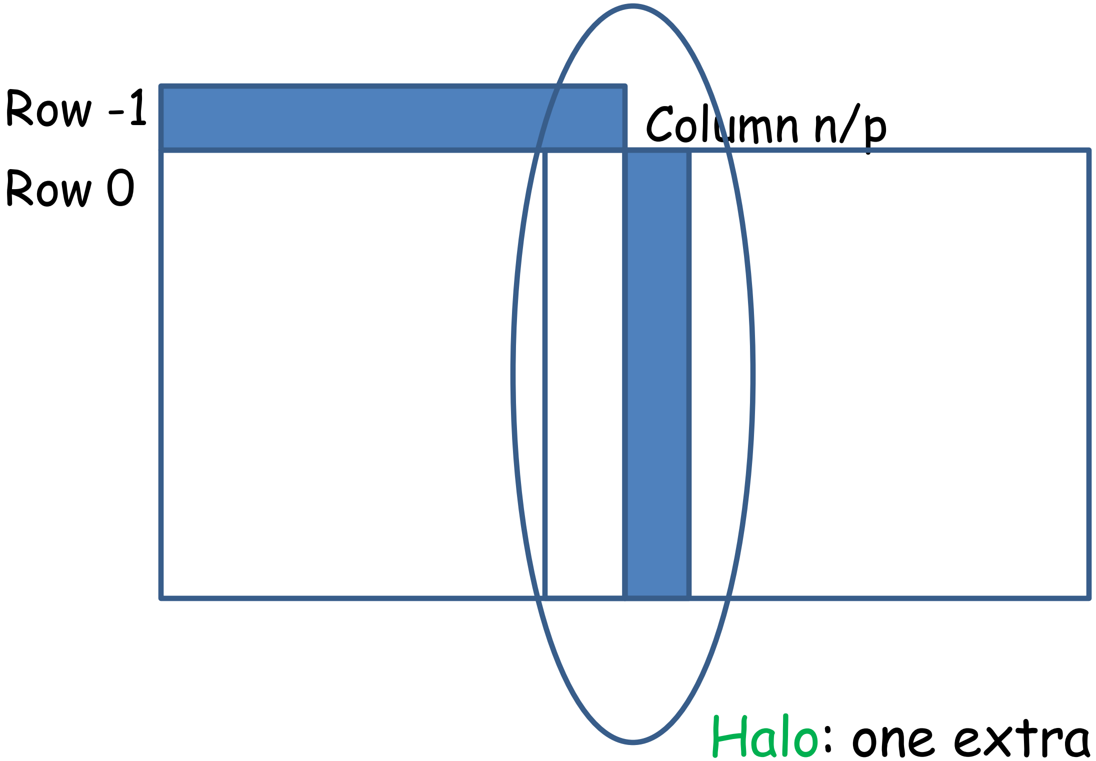
MPI standard is intentionally loose on progress to allow different implementations

Problem 2: Where are the data?

Each process

1. needs data to row -1 and row n/p from up and down processes
2. contributes data in row 0 and row n/p-1 to up and down processes
3. needs data to column -1 and column n/p from left and right processes
4. contributes data in column 0 and column n/p-1 to left and right processes

Region of overlap is called halo, can be deeper than 1 row/column (and save communication at the cost of space)



Halo: one extra column per process

Communication calls take buffer arguments and communicate serially from/to these buffers

Communication buffers always have a datatype and an element count, for example

```
MPI_Bcast(void* buffer, int count, MPI_Datatype type,
          int root, MPI_Comm comm)
```

Datatypes are MPI objects that (can) correspond to the basic datatypes in C (and FORTRAN)

Basic MPI_Datatype	C type
MPI_CHAR	char
MPI_SHORT	(signed) short (int)
MPI_INT	int
MPI_LONG	(signed) long (int)
MPI_LONG_LONG	signed long long int
MPI_SIGNED_CHAR	signed char
MPI_UNSIGNED_CHAR	unsigned char
MPI_UNSIGNED_SHORT	unsigned short int
MPI_UNSIGNED	unsigned int
MPI_UNSIGNED_LONG	unsigned long int
MPI_UNSIGNED_LONG_LONG	unsigned long long int
MPI_C_BOOL	_Bool
MPI_WCHAR	wchar_t

Basic MPI_Datatype	C type
MPI_INT8_T	`int8__t`
MPI_INT16_T	`int16_t`
MPI_INT32_T	`int32_t`
MPI_INT64_T	`int64_t`
MPI_INT8_T	`uint8__t`
MPI_INT16_T	`uint16_t`
MPI_INT32_T	`uint32_t`
MPI_INT64_T	`uint64_t`

Basic MPI_Datatype	C type
MPI_FLOAT	float
MPI_DOUBLE	double
MPI_LONG_DOUBLE	long double
MPI_C_COMPLEX	float _Complex
MPI_C_DOUBLE_COMPLEX	double _Complex
MPI_LONG_DOUBLE_COMPLEX	long double _Complex

Special datatypes

Basic MPI_Datatype	
MPI_BYTE	Uninterpreted bytes
MPI_PACKED	Special, packed data (*)

(*) generated by MPI_Pack/MPI_Unpack only

Basic MPI_Datatype	C type	Fortran type
MPI_AINT	`MPI_Aint`	`INTEGER (KIND=MPI_ADDRESS_KIND)`
MPI_OFFSET	`MPI_Offset`	`INTEGER (KIND=MPI_OFFSET_KIND)`

MPI_Aint: address sized int

Communication operations transfer a serialized stream of elements of basic datatype from one process to another

Stream of basic elements is called a type signature

Example: count=7 of MPI_INT describes the signature <int,int,int,int,int,int,int>

Correctness rule for any type of communication: the signature of the sent elements must be identical(*) to the signature of the elements to be received

(*) for point-to-point and one-sided: sent signature must be a prefix of expected to be received signature

Communication operations transfer a serialized stream of elements of basic datatype from one process to another

Stream of basic elements is called a type signature

Example: count=7 of MPI_INT describes the signature <int,int,int,int,int,int,int>

Correctness rule for any type of communication: the signature of the sent elements must be identical(*) to the signature of the elements to be received

MPI is designed for high performance: no meta-information on type signature is sent, it is user responsibility to write correct programs

Rank i:

```
int N = 1000;
int a[N];
MPI_Send(a,N,MPI_INT,j,777,MPI_COMM_WORLD);
```

Rank j:

```
MPI_Status status;
int N = 1000; // or larger
int b[N];
MPI_Recv(b,N,MPI_INT,i,MPI_COMM_WORLD,&status);
```

Rank i:

```
int N = 1000;
int a[N];
MPI_Send(a,N,MPI_INT,j,777,MPI_COMM_WORLD);
```

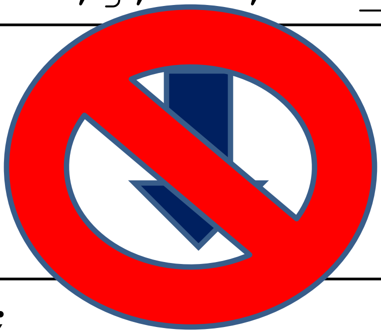
Rank j:

```
MPI_Status status;
int N = 1000; // or larger
double b[N];
MPI_Recv(b,N,MPI_DOUBLE,i,MPI_COMM_WORLD,&status);
```

Erroneous. But MPI library will (most likely) not complain. MPI is designed for high performance

Rank i:

```
int N = 1000;
int a[N];
MPI_Send(a,N,MPI_INT,j,777,MPI_COMM_WORLD);
```



Rank j:

```
MPI_Status status;
int N = 1000; // or larger
int b[N];
MPI_Recv(b,N*sizeof(int),MPI_BYTE,i,MPI_COMM_WORLD,
         &status);
```

Also erroneous, and very bad practice

Principle 5:

All MPI communication is typed, and consists of sequences of elements of basic types. Types must be respected

Basic datatypes have underlying semantics, and keeping elements typed (not streams of uninterpreted bytes) makes it possible to communicate across heterogeneous systems, e.g. different word size, different endianness, …

Stencil example: row and column data

C stores 2-dimensional arrays in row major order, so rows form consecutive sequences of elements in memory. Communication can be done as sequences of basic elements, e.g., MPI_DOUBLE

Column elements are non-consecutive, but regularly strided in process memory. How can this be handled?

Solution 0: Element by element

Solution 1: by hand, pack column elements into consecutive, intermediate buffer, send as sequence of MPI_DOUBLE

Solution 2: Use MPI to do the work

Element by element:

```
for (i=0; i<n/p; i+       {
   MPI_Send(&matrix    ][      1,MI    _DOUBLE,…,comm);
}
```

Packing by hand:

- Takes time
- Takes space
- For complex, irregular data layouts, writing efficient packing routines is non-trivial (and may not be portable: cache-system)

Principle 6:

Message-passing is better for bulk transfers, exchange as large messages as possible

Communication operations have latency/overhead independent of message size. Overhead can be significant:

- Hardware
- Software
- Algorithmic (for collective operations: lower bounds on number of required communication rounds)

Overhead/latency captured in communication cost models (linear, LogGP, …)

Overheads can be amortized by large messages that can exploit full communication bandwidth

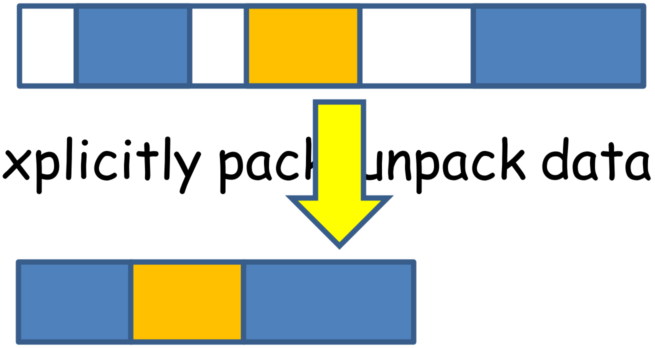
Derived (user-defined) datatypes

General mechanism for

- Describing arbitrarily complex, non-consecutive and heterogeneous (different basic datatypes) layouts of data in memory
- Serializing access to structured data in communication operations: from type map (sequence of basic datatypes with and addresses) to type signature
- Fixing units of communication

- Can be used with all communication models: point-to-point, one-sided, collective; essential for MPI I/O specification

Applications with non-contiguous data layouts (sub-matrices, stencils, irregular structures, …)



```
MPI_Bcast(buf,newtype,…);
```

- either explicitly pack/unpack data at communication operations, or

```
MPI_Bcast(buf,MPI_BYTE,…);
```

- describe data layout as derived datatype, and delegate any necessary packing/unpacking to MPI library

```
MPI_Type_create…(…,&newtype);
MPI_Type_commit(&newtype);
```

Derived datatype advantages

Descriptive:

- "Higher level" description of structure of data, no need to bother with tedious&specialized pack/unpack routines
- All handling of structured data delegated to MPI library

Performance:

- Efficient, pipelined, once-and-for-all pack/unpack functionality, saves space for intermediate buffers in user space
- Communication operations with internal buffering genuinely benefit, datatype engine copies directly into internal buffers
- Communication system with non-contiguous (strided) operations can be exploited

Potential for MPI aware compilers…

Derived datatype mechanism

Set of increasingly general/complex/confusing constructors that describe where the basic elements are (displacements), what they are (basic datatype), and in what order they shall be accessed

```
MPI_Type_commit(MPI_Datatype *type)
```

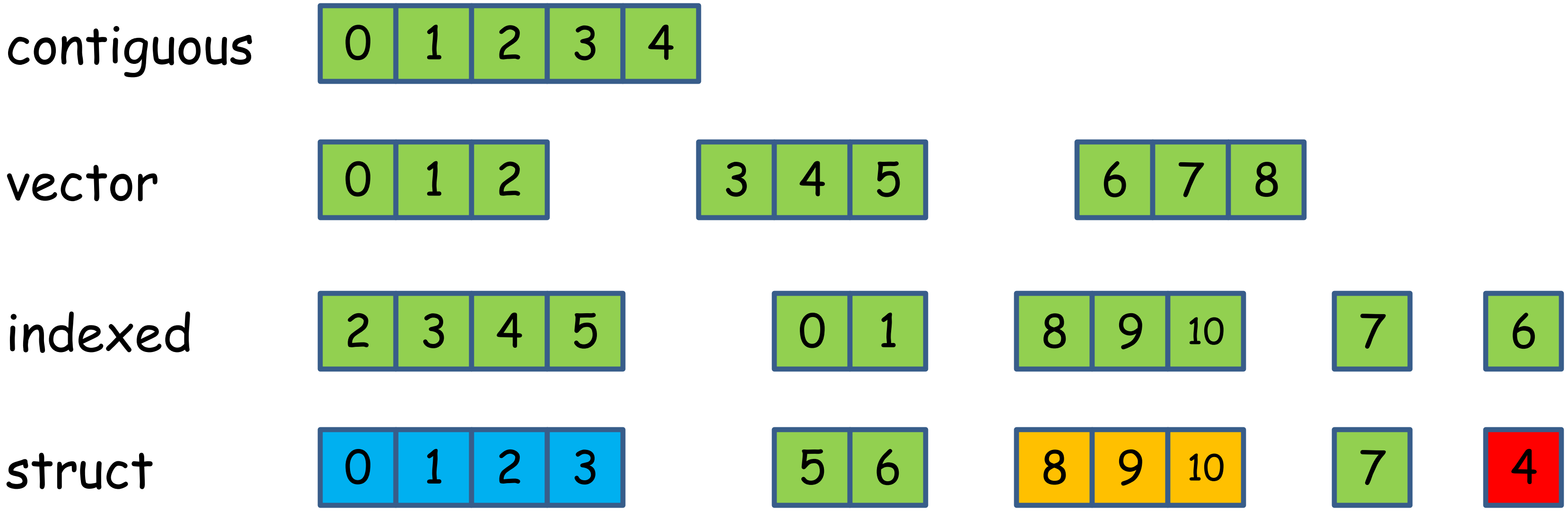
Commit needed to make new datatype usable in communication; handle for MPI library to perform optimizations

```
MPI_Type_free(MPI_Datatype *type)
```

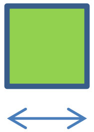
Good practice to free datatypes when no longer used; datatypes can take up some space

Constructor overview

🟩 Basetype: basic or user-defined



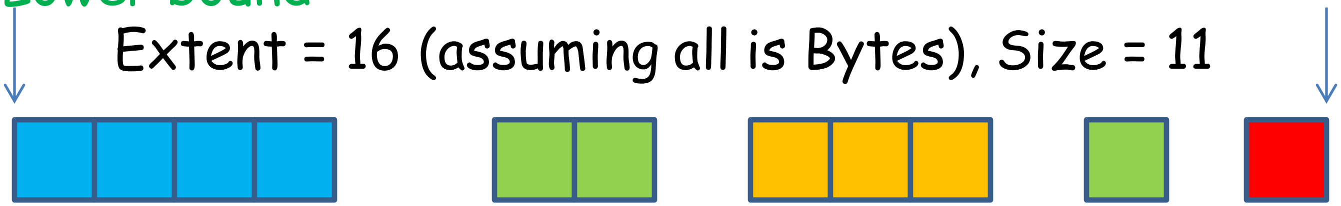
Sequence numbers: order in which basetype are serialized



Basetype: basic or derived

True extent: difference between first and last element (byte) in datatype, “footprint”

Size: number of actual elements (Bytes) occupied by datatype



Lower bound

Extent = 16 (assuming all is Bytes), Size = 11

The extent (not true extent) is used put typed elements after each other

```
MPI_Type_size (MPI_Datatype type, int *size)
```

```
MPI_Type_get_extent(MPI_Datatype type,
                    MPI_Aint *lb, MPI_Aint *extent)
```

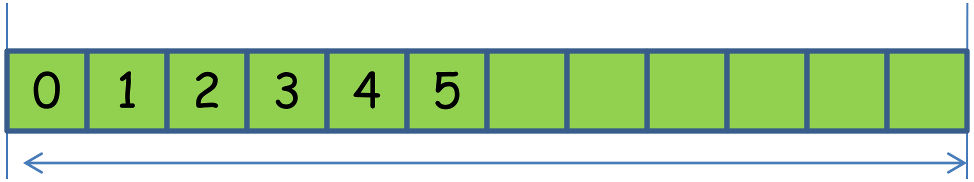
Careful: extent and lb are MPI_Aint, bytes is int

Extent is a unit for putting elements after each other, and need not be the actual footprint of the datatype. The footprint is the true extent

```
MPI_Type_get_true_extent(MPI_Datatype type,
                         MPI_Aint *lb,
                         MPI_Aint *extent)
```

```
MPI_Type_contiguous(int count, MPI_Datatype oldtype,
                    MPI_Datatype *newtype)
```

🟩 Basetype: basic or derived

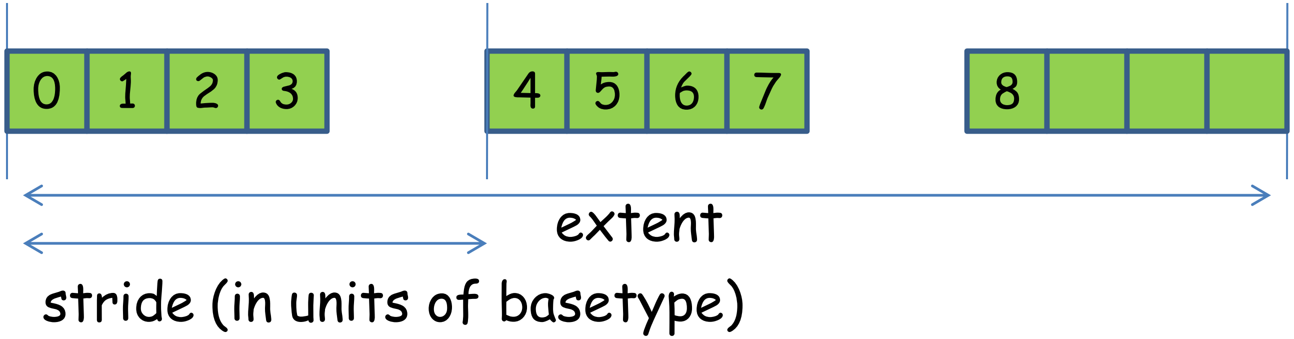


extent (for repetition)

Space for description: 1 word

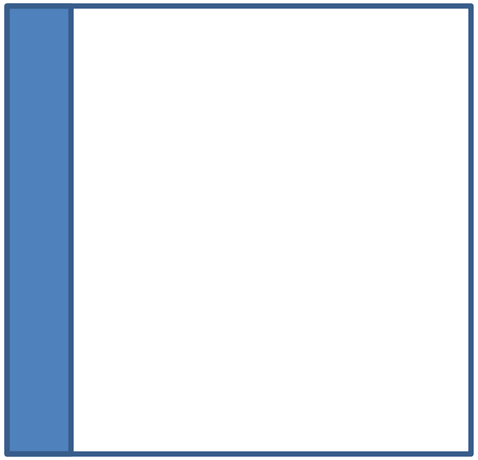
```
MPI_Type_vector(int count,
                int blocklength, int stride,
                MPI_Datatype oldtype,
                MPI_Datatype *newtype)
```

🟩 Basetype: basic or derived

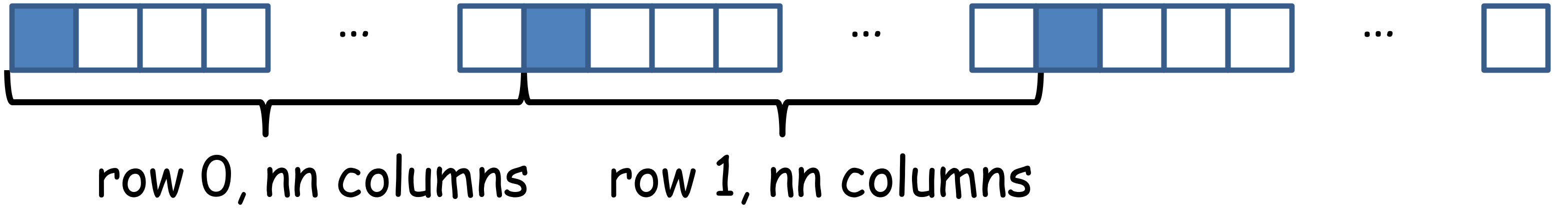


Space for description: 3 words

Column of n/p x n/p matrix



```
MPI_Datatype column;
int nn = n/p;
MPI_Type_vector(nn,1,nn,MPI_DOUBLE,
                &column);
MPI_Type_commit(&column);

MPI_Isend(&m[0][0],1,column,l,…,comm);
MPI_Isend(&m[0][nn-1],1,column,r,…,
          comm);
```



```
MPI_Type_free(&column);
```

Principle 7:

Use MPI datatypes for transferring static, complex data layouts whenever possible and convenient

- Communication using a datatype should be no slower (faster!) than first copying (manually, or with MPI_Pack/MPI_Unpack) into intermediate buffer and then sending contiguous (MPI_PACKED) buffer

- Apply principle with some care:
- Performance of MPI derived datatypes was bad in early MPI libraries
- Performance has improved significantly, but there is still work to do

Principle 7:

Use MPI datatypes for transferring static, complex data layouts whenever possible and convenient

Golden MPI rule

- Use derived datatypes for conciseness and performance whereever possible

- Complain to MPI library implementer (and MPI community) if performance anomalies are discovered

Data type performance expectation:

```
MPI_Send(sendbuf,1,type,…,comm);
```

should be no slower (and hopefully faster) than

```
int position = 0;
MPI_Pack(sendbuf,1,type,packbuf,packsize,position,
         comm);
MPI_Send(packbuf,1,MPI_PACKED,…,comm);
```

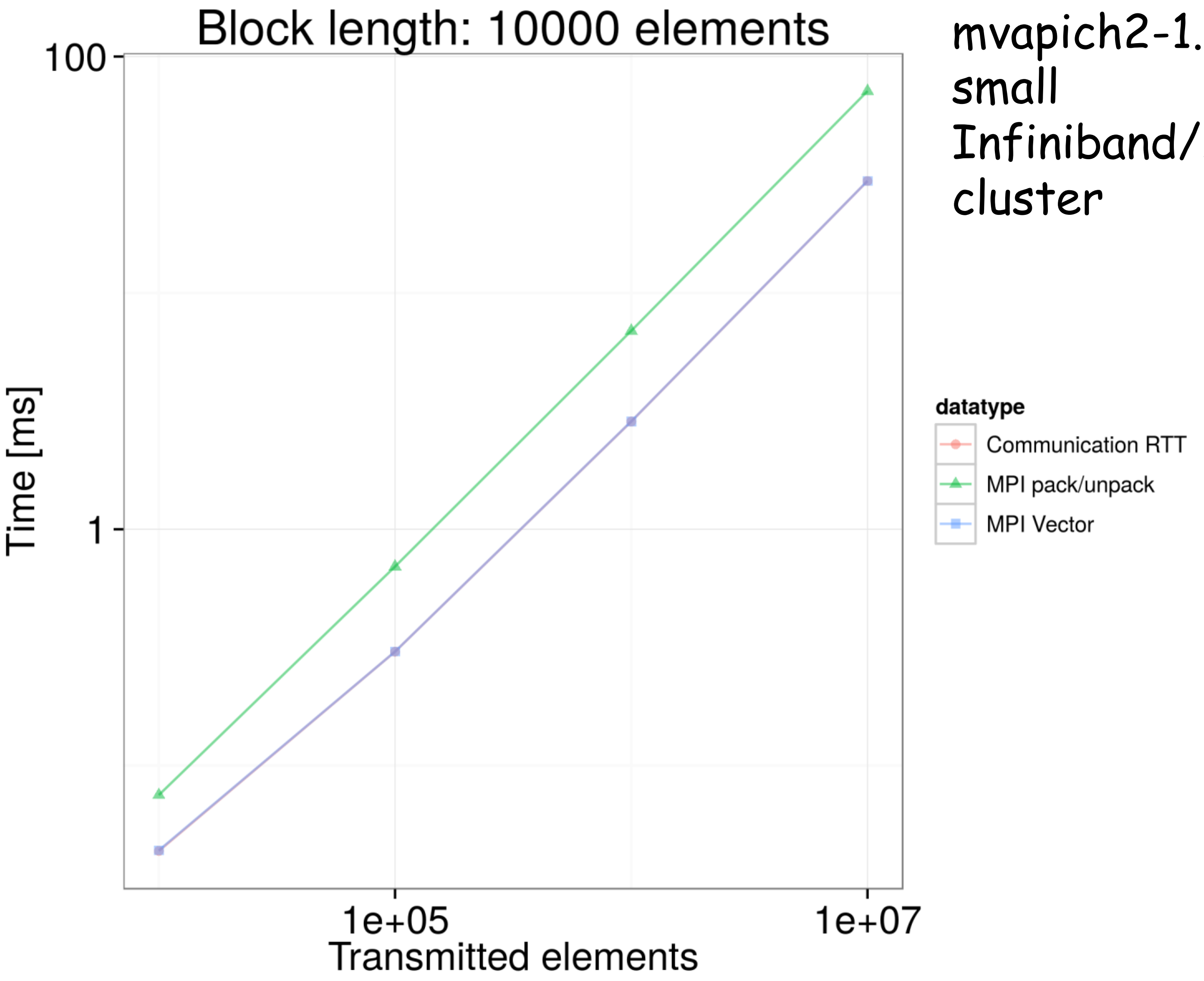
Otherwise, datatype mechanism would not seem to make any sense (performance wise), user could do better with MPI_Pack()/MPI_Unpack()

Experiment:

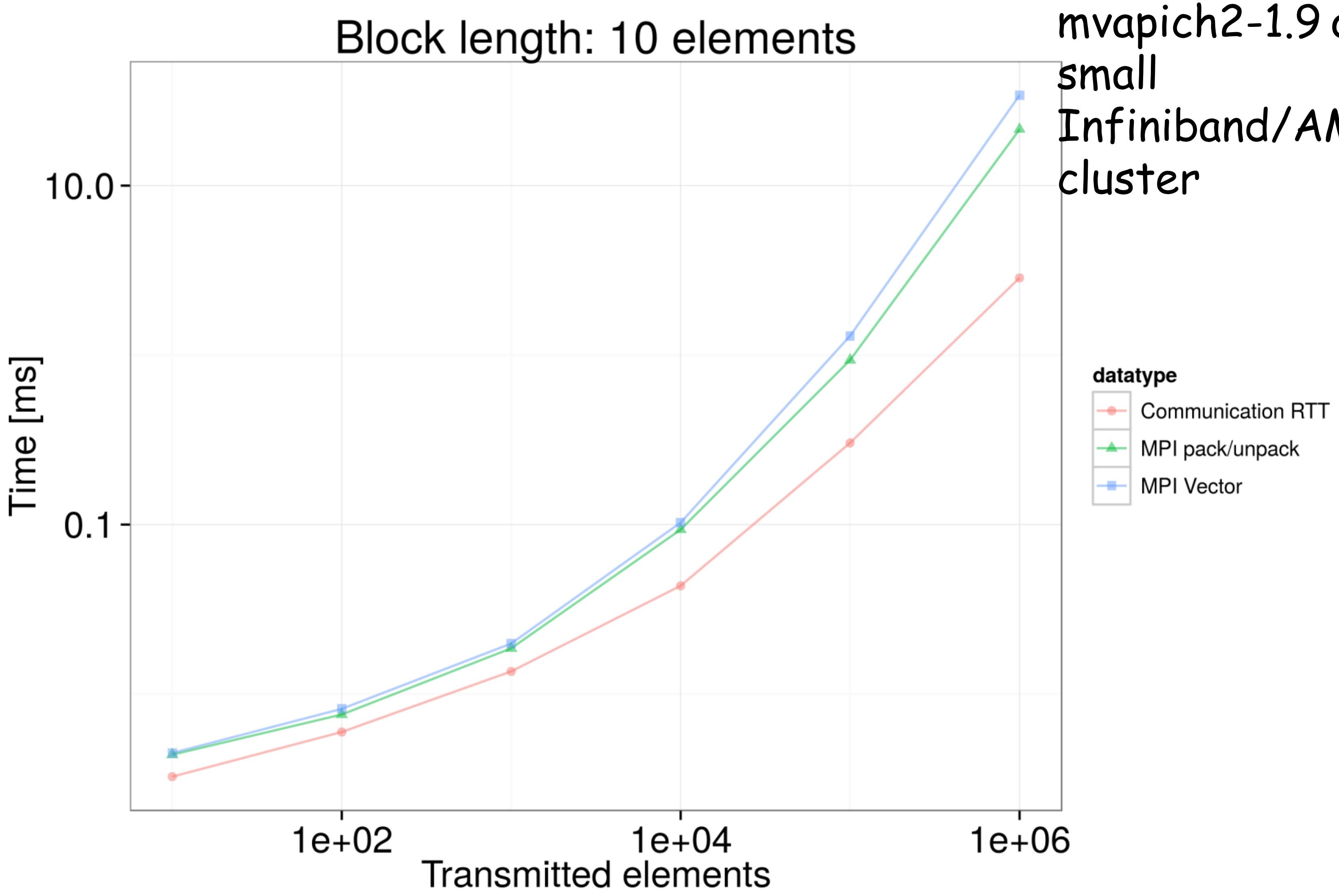
Send and receive (ping-pong benchmark) some amount of data as vector with fixed stride and varying blocklength; compare to MPI_Pack()/MPI_Unpack() and intermediate, packed buffer; compare to raw performance with consecutive buffer

Which is better?

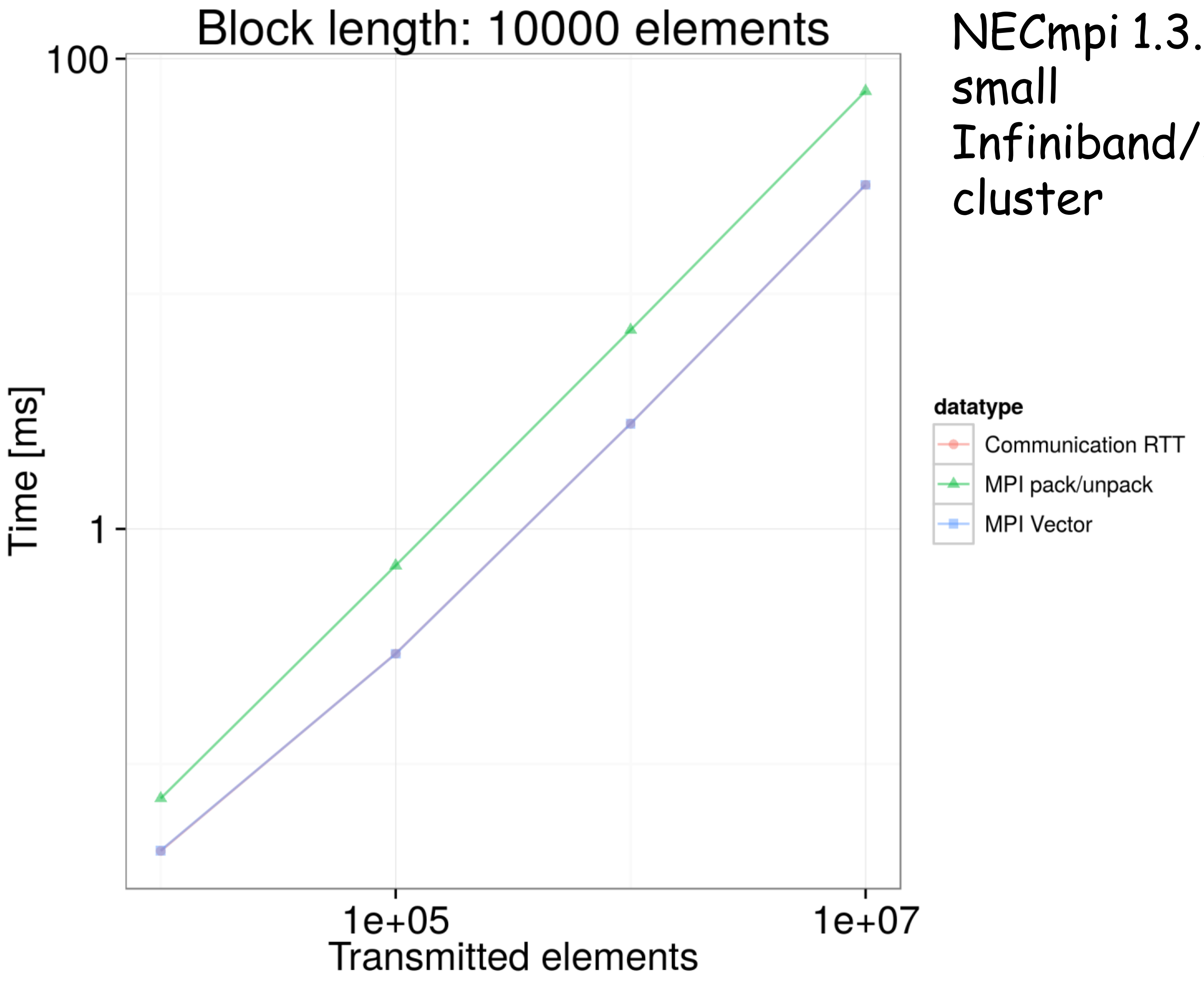
Stride = 100000 elements (MPI_DOUBLE)



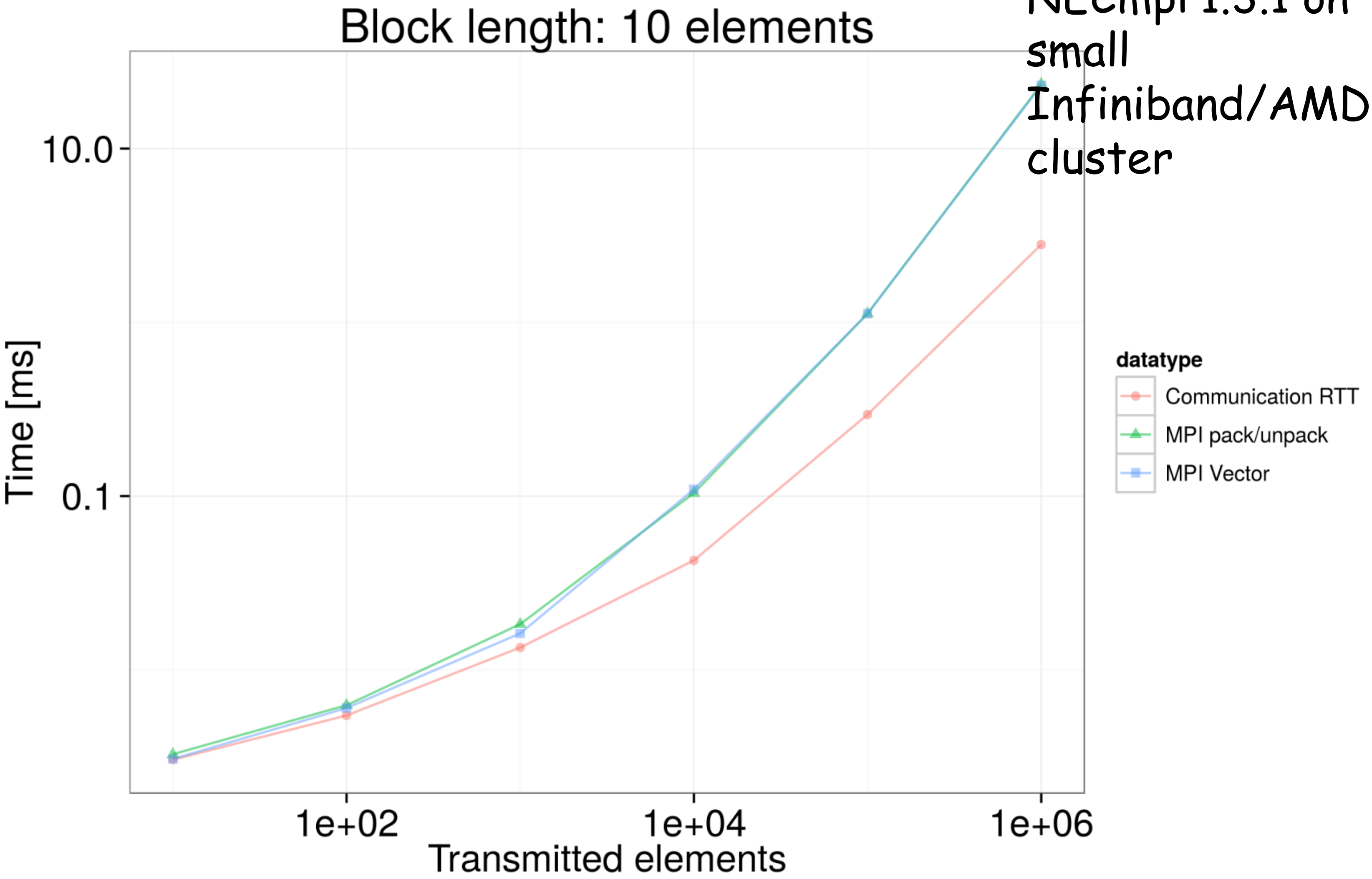
mvapich2-1.9 on small Infiniband/AMD cluster



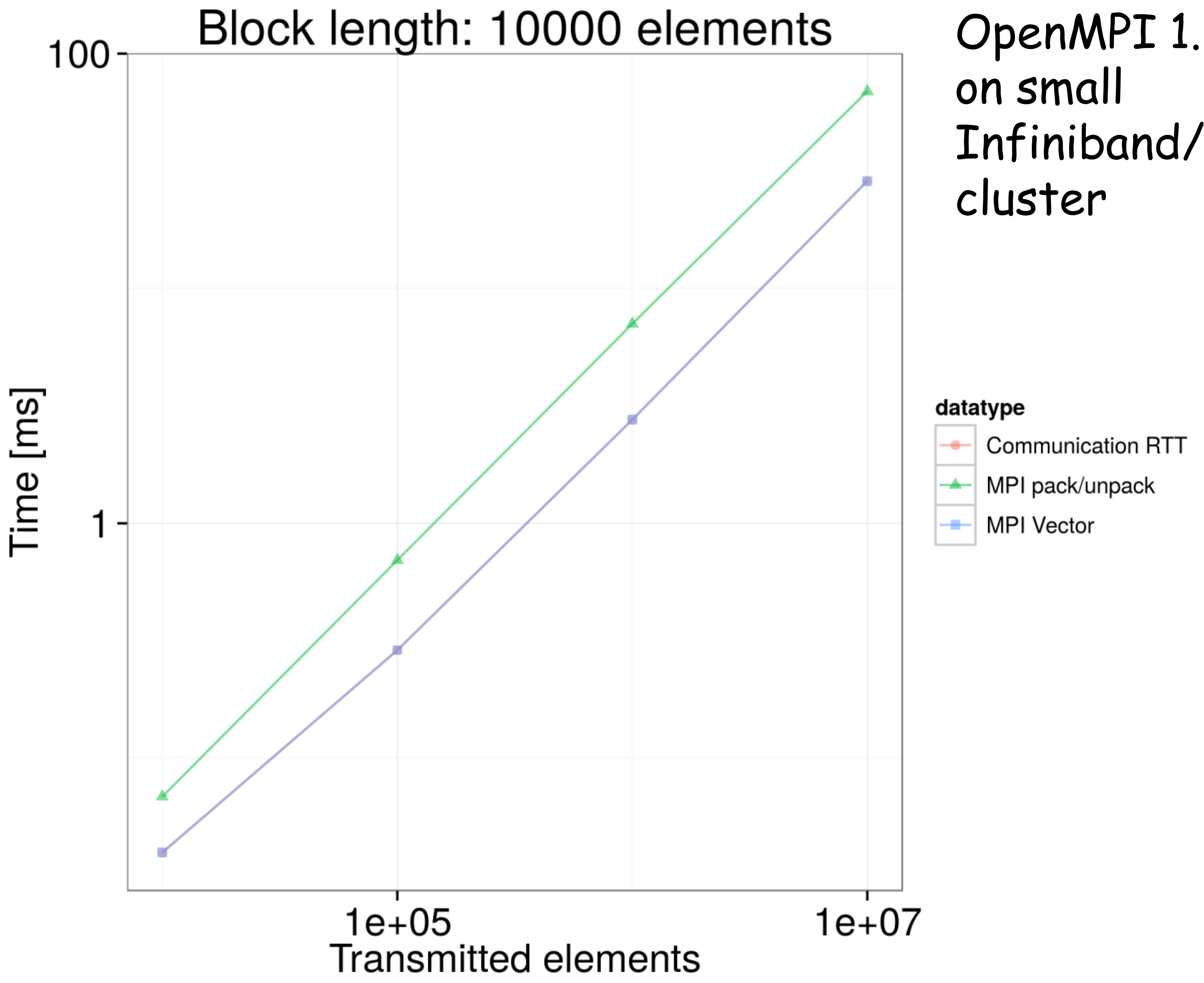
mvapich2-1.9 on small Infiniband/AMD cluster



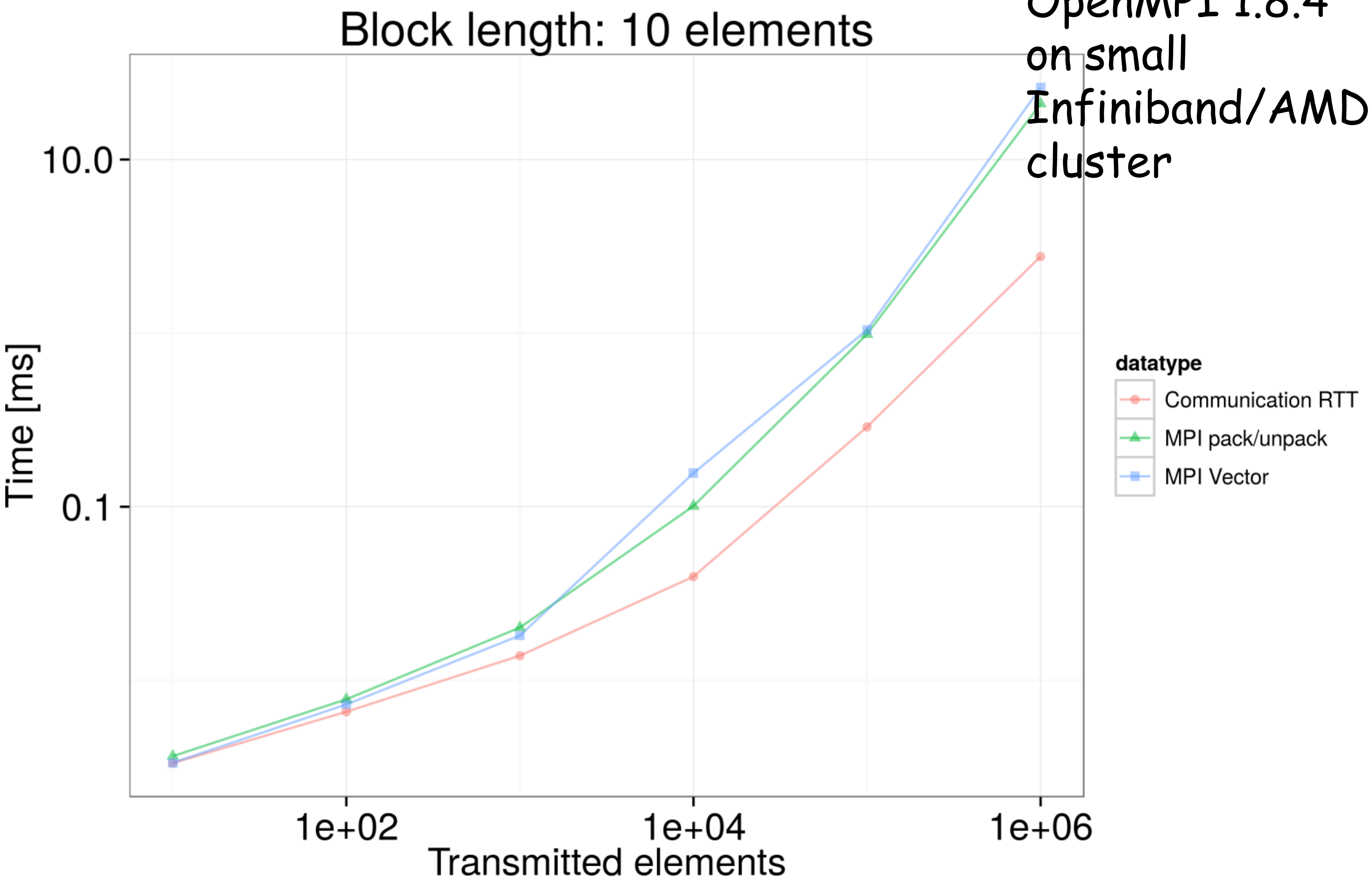
NECmpi 1.3.1 on small Infiniband/AMD cluster



NECmpi 1.3.1 on small Infiniband/AMD cluster

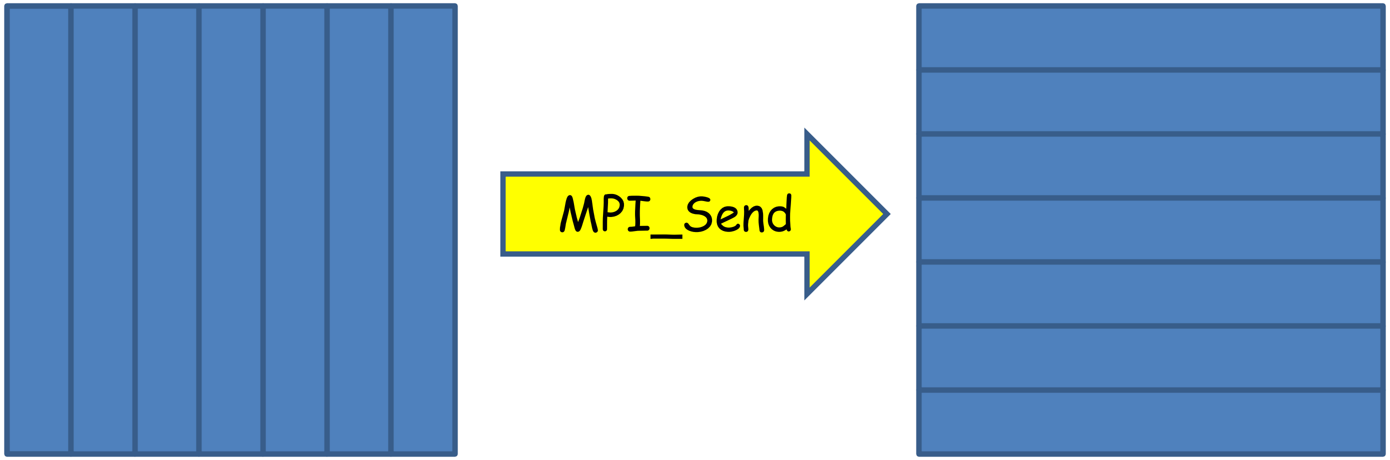


OpenMPI 1.8.4 on small Infiniband/AMD cluster



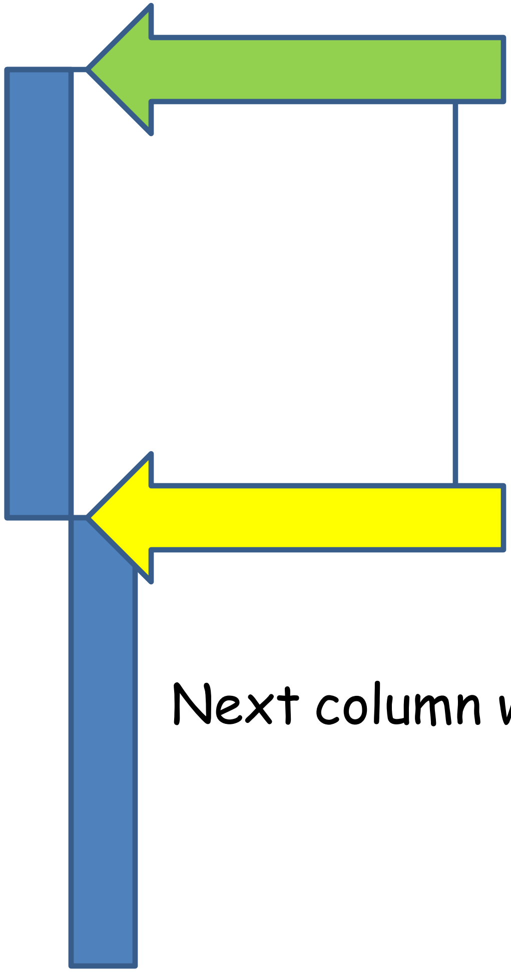
OpenMPI 1.8.4 on small Infiniband/AMD cluster

Example: Matrix transpose by communication



```
MPI_Datatype column, columnmatrix;
MPI_Type_vector(n,1,n,MPI_DOUBLE,&column);
MPI_Type_contiguous(n,column,&columnmatrix);
MPI_Type_commit(&columnmatrix);
```

…is wrong. Why?



To put the columns together inside matrix, a different extent of the column vector is needed

Extent of vector ends here

Next column would go out of allocated space for matrix

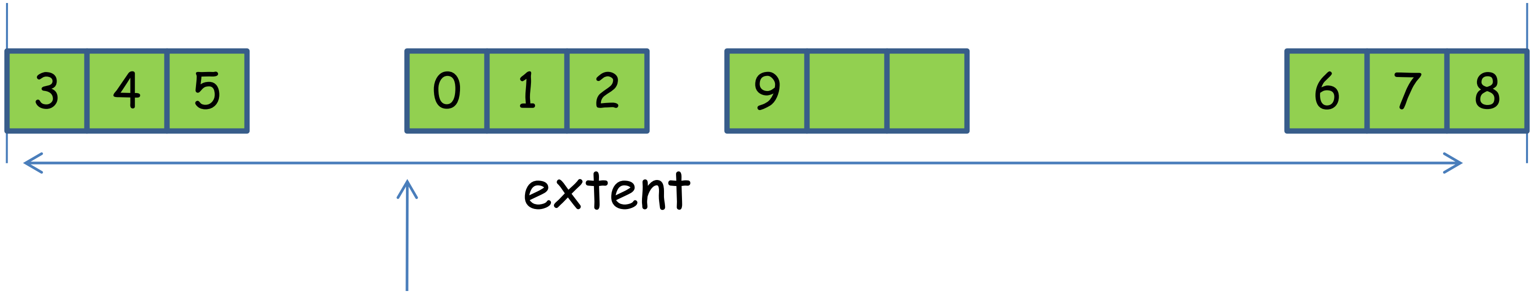
```
MPI_Type_create_resized (MPI_Datatype oldtype,
     MPI_Aint newlb, MPI_Aint newextent,
     MPI_Datatype *newtype)
```

Copy derived datatype, but explicitly force new extent and lb

```
MPI_Datatype column, columnmatrix, col;
MPI_Type_vector(n,1,n,MPI_DOUBLE,&column);
MPI_Type_create_resized(column,0,sizeof(double),&col);
MPI_Type_contiguous(n,col,&columnmatrix);
MPI_Type_commit(&columnmatrix);

MPI_Type_free(&column); // free intermediate types
MPI_Type_free(&col);
```

```
MPI_Type_create_indexed_block(int count,
    const int blocklength, const int displacements[],
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

🟩 Basetype: basic or derived



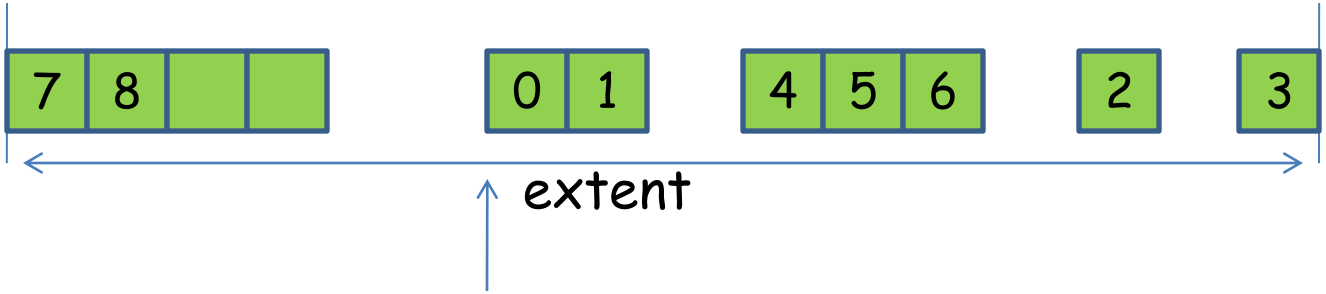
extent

displacement in units of basetype

Space for description: 1 word, 1 array (displs)

```
MPI_Type_indexed(int count, const int blocklengths[],
                 const int displacements[],
                 MPI_Datatype oldtype,
                 MPI_Datatype *newtype)
```

🟩 Basetype: basic or derived



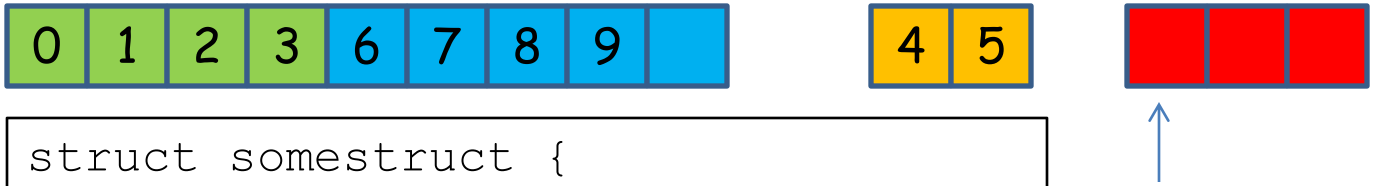
extent

displacement in units of basetype

Space for description: 1 word, 2 arrays (blocks, displs)

```
MPI_Type_create_struct(int count,
    const int blengths[], const MPI_Aint disps[],
    const MPI_Datatype oldtypes[],
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

🟩 🟨 🟦 🟥 Basetypes: basic or derived



```
struct somestruct {
        int a[4];
        double b[4];
        int somepaddingthatweignore;
        double c;
        char d[2];
        long e[3];
}
```

Displacements in bytes

Cost of description:
1 word, 3 arrays

```
struct somestruct {
       int a[4];
       double b[4];
       int somepaddingthatweignore;
       double c;
       char d[2];
       long e[3];
} data[100];
```

Presenting structure to MPI can be tedious

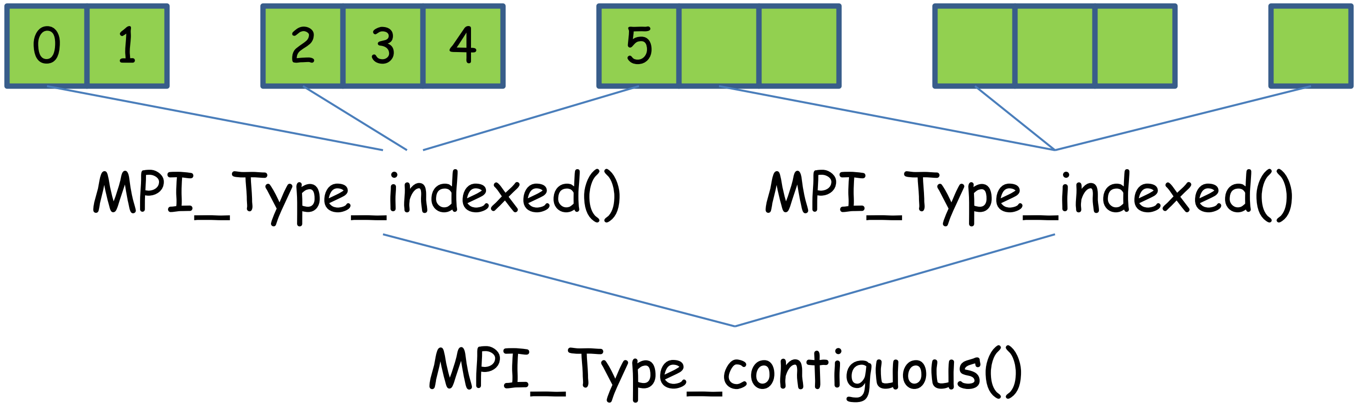
```
int blocks[5] = {4,4,1,2,3};
MPI_Aint displs[5];
displs[0] = 0;
displs[1] = data[0].b-data; // rel.offset of b in struct
displs[2] = &data[0].c-data;
displs[3] = data[0].d-data;
displs[3] = data[0].e-data;
MPI_Datatype etypes[5] =
  {MPI_INT,MPI_DOUBLE,MPI_DOUBLE,MPI_CHAR,MPI_LONG};

MPI_Type_struct(5,blocks,displs,etypes,&newdatatype);
MPI_Type_commit(&newdatatype);
```

```
struct somestruct {
       int a[4];
       double b[4];
       int somepaddingthatweignore;
       double c;
       char d[2];
       long e[3];
} data[100];
```

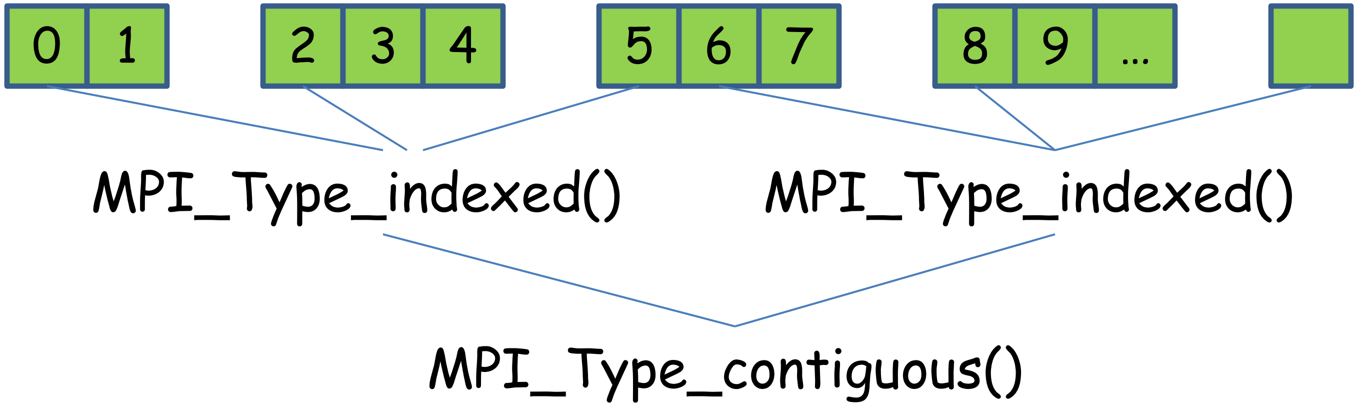
Not good practice. Use MPI functions to manipulate addresses

```
int blocks[5] = {4,4,1,2,3};
MPI_Aint displs[5];
displs[0] = 0;
displs[1] = data[0].b-data; // rel.offset of b in struct
displs[2] = &data[0].c-data;
displs[3] = data[0].d-data;
displs[3] = data[0].e-data;
MPI_Datatype etypes[5] =
  {MPI_INT,MPI_DOUBLE,MPI_DOUBLE,MPI_CHAR,MPI_LONG};

MPI_Type_struct(5,blocks,displs,etypes,&newdatatype);
MPI_Type_commit(&newdatatype);
```

Derived datatypes can be constructed recursively



Derived datatypes can be constructed recursively

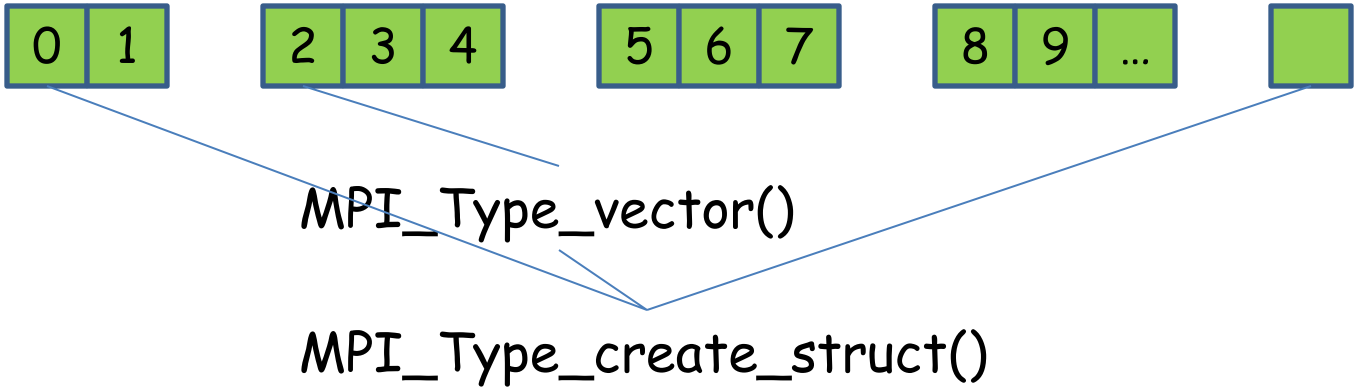


MPI_Type_indexed()

MPI_Type_indexed()

MPI_Type_contiguous()

Type map also described by



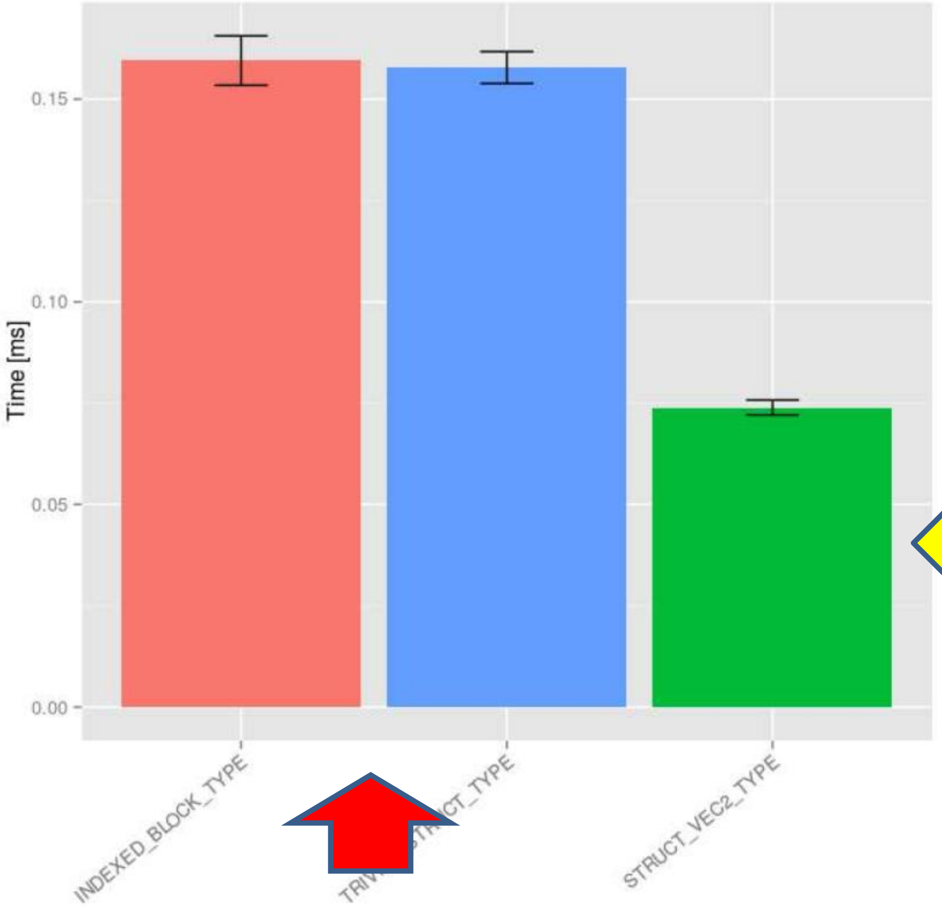
MPI_Type_vector()

MPI_Type_create_struct()

Which description is better?

- Latter uses less space, and is more regular
- Task of MPI_Type_commit(); to "normalize" user-defined description into best description (system dependent)
- Finding optimal decription (in some meaningful sense) a difficult problem
- Representation can make a significant difference

[Robert Ganian, Martin Kalany, Stefan Szeider, Jesper Larsson Träff: Polynomial-time construction of optimal MPI derived datatype trees. To appead, IPDPS 2016]



Layout:
Colulmn+Diagonal of nxn matrix

Layout described in three different ways

Optimal representation

Ping-pong benchmark on small AMD/InfiniBand cluster

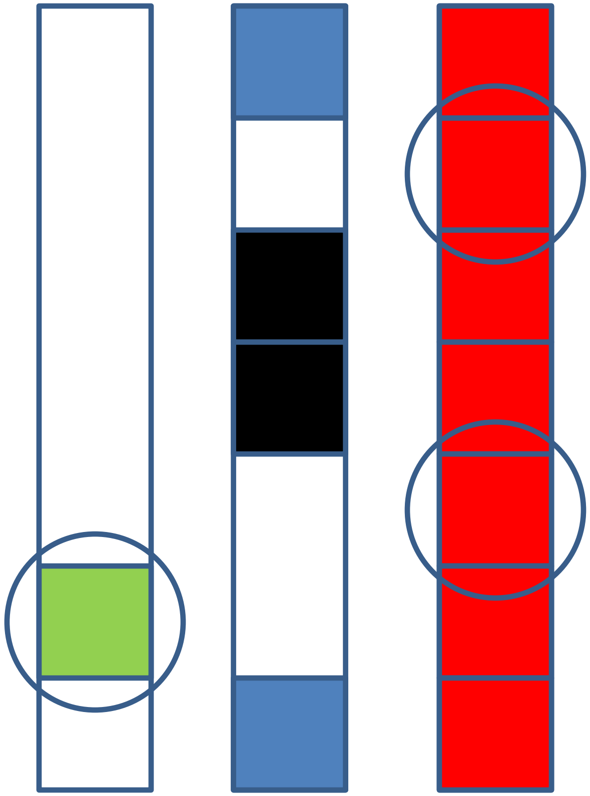
Displacements and strides are in units of old datatype, sometimes not sufficiently expressive. There are corresponding constructors with byte units

```
MPI_Type_create_hvector(int count,
    int blocklength, MPI_Aint stride,
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

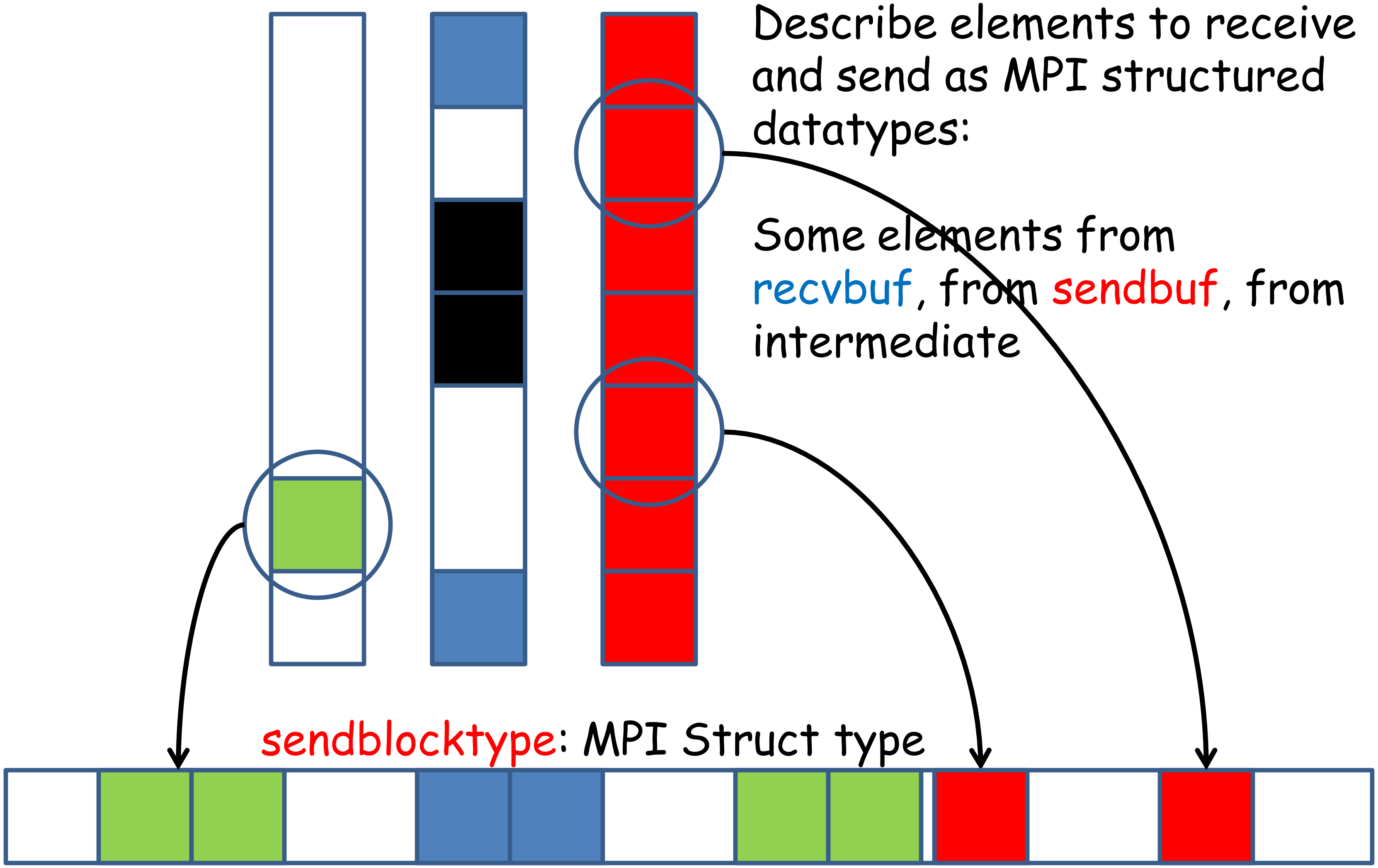
```
MPI_Type_create_hindexed_block(int count,
    const int blocklength,
    const MPI_Aint displacements[],
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

```
MPI_Type_create_hindexed(int count,
    const int blocklengths[],
    const MPI_Aint displacements[],
    MPI_Datatype oldtype, MPI_Datatype *newtype)
```

Example: non-consecutive blocks from multiple buffers



From an all-to-all algorithm:
In each of log p rounds, different blocks from three buffers (sendbuffer, receive buffer, intermediate buffer) to be sent and received



Describe elements to receive and send as MPI structured datatypes:

Some elements from recvbuf, from sendbuf, from intermediate

sendblocktype: MPI Struct type

Leads to zero-copy implementation:

•No explicit, process-local copying of data, all data movements by derived datatypes

•Derived datatypes specifies where the data are, MPI library fetches the data and copies where necessary if at all)

[Torsten Hoefler, Steven Gottlieb: Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient Using MPI Datatypes. EuroMPI 2010: 132-141]
[Jesper Larsson Träff, Antoine Rougier, Sascha Hunold: Implementing a classic: zero-copy all-to-all communication with mpi datatypes. ICS 2014: 135-144]
[Enes Bajrovic, Jesper Larsson Träff: Using MPI Derived Datatypes in Numerical Libraries. EuroMPI 2011: 29-38]

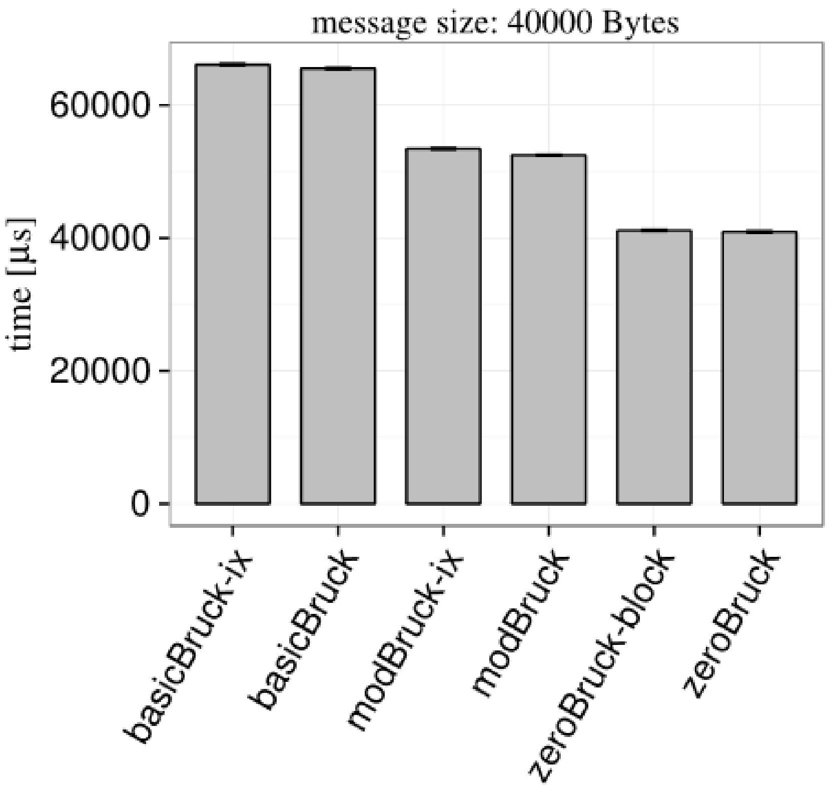
Zero-copy implementation

```
for (k=1; k<p; k<<1) { // communication round
  for (j=k; j<p; j++) {
    // analyze bits of j
    ..
    b++; // number of blocks to send/receive
  }
  MPI_Type_create_struct(b,…,&sendblocktype);
  MPI_Type_create_struct(b,…,&recvblocktype);
  MPI_Type_commit(&sendblocktype);
  MPI_Type_commit(&recvblocktype);

  MPI_Sendrecv(MPI_BOTTOM,1,sendblocktype,
               MPI_BOTTOM,1,recvblocktype,…);
  MPI_Type_free(&sendblocktype);
  MPI_Type_free(&recvblocktype);
}
```

specify

execute

Experiment on shared-memory system with 80 MPI processes (large problems)



- ModBruck saves 2xp element copies over BasicBruck

- ZeroBruck save p element copies and (log p)p/2 element unpack over ModBruck

Useful address MPI_BOTTOM

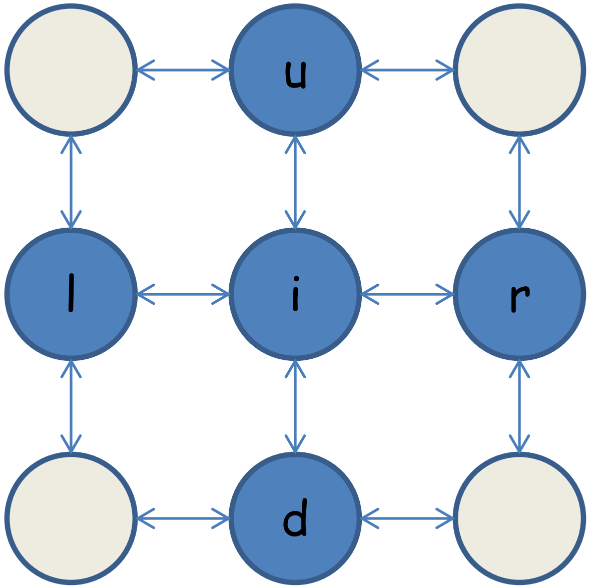
Standardized, portable NULL address

Can be used as buffer argument in all communication operations

Absolute addresses as MPI_Aint types inside derived datatype argument

Use with MPI_Get_address() that returns address relative to MPI_BOTTOM

Solution 2: (Non-blocking) collective communication



All processes communicate at the same time (at the same point in the execution), always with the same set of neighboring processes

Possibility for using collective operations?

17 "MPI collectives" for data exchange and reduction. Possibly some type of all-to-all communication useful for stencil?

Class	regular	Irregular, vector
Symmetric, no data	MPI_Barrier	
Rooted	MPI_Bcast	
Rooted	MPI_Scatter	MPI_Scatterv
Rooted	MPI_Gather	MPI_Gatherv
Symmetric, non-rooted	MPI_Allgather	MPI_Allgatherv
Symmetric, non-rooted	MPI_Alltoall	MPI_Alltoallv, MPI_Alltoallw
Rooted	MPI_Reduce	
Non-rooted	MPI_Reduce_scatter_block	MPI_Reduce_scatter
Symmetric, non-rooted	MPI_Allreduce	
Non-rooted	MPI_Scan	
Non-rooted	MPI_Exscan	

Principle 8:

- Identify collective communication patterns, and use MPI collectives, instead of (re)implementing with point-to-point or one-sided communication
- Use most specific collective operation applicable

- There is a wealth of literature on algorithms and implementations for collective operations (see IPDPS, SC, EuroMPI, ICPP, … conferences)
- MPI libraries often do quite well

[Ernie Chan, Marcel Heimlich, Avi Purkayastha, Robert A. van de Geijn: Collective communication: theory, practice, and experience. Concurrency and Computation: Practice and Experience 19(13): 1749-1783 (2007)]

Complexity of collective operations (fully connected network)

MPI Collective	Complexity
MPI_Barrier	Θ(log p)
MPI_Bcast	Θ(n+log p)
MPI_Gather/Scatter	Θ(n+log p)
MPI_Allgather	Θ(n+log p)
MPI_Alltoall	O(n+p) (*)
MPI_Reduce	Θ(n+log p)
MPI_Allreduce	Θ(n+log p)

Important to find algorithms with small O-constants

(*) interesting tradeoffs posssible

MPI_Bcast should NOT be O(n log p)

- p MPI processes (on p processors), n is total amount of data per process
- Strong (fully connected) network, linear communication cost

Complexity of collective operations (general)

MPI Collective	Complexity
MPI_Barrier	$\Theta(d)$
MPI_Bcast	$\Theta(n+d)$
MPI_Gather/Scatter	$\Theta(n+d)$
MPI_Allgather	$\Theta(n+d)$
MPI_Alltoall	$O(n+p)$ (*)
MPI_Reduce	$\Theta(n+d)$
MPI_Allreduce	$\Theta(n+d)$

Important to find algorithms with small O-constants

(*) interesting tradeoffs posssible

- p MPI processes (on p processors), n is total amount of data per process
- d = max(log p, diameter of network)

Principle 7:

- Identify collective communication patterns, and use MPI collectives, instead of implementing with point-to-point or one-sided communication
- Use most specific collective operation applicable

Golden MPI rule

- Use collectives for conciseness and performance whereever possible

- Complain to MPI library implementer if performance anomalies are discovered

```
MPI_Barrier(MPI_Comm comm)
```

- Semantic synchronization: All processes in comm must call (and not other collective call inbetween)
- Process returns when all other processes have performed call
- NO TEMPORAL SYNCHRONIZATION IMPLIED
- But MPI libraries usually try to make processes return from barrier "at the same time" (for benchmarking)

Use-cases:

- Enforcing consistent global system state, all processes at the same point in execution
- Separating passive-target (lock) epochs in one-sided communication
- Using MPI_Rsend()
- Checkpointing

…

```
MPI_Ibarrier(MPI_Comm comm, MPI_Request &request)
```

Non-blocking collective operation:

- Returns immediately, no guarantee
- Check progress with MPI_Test(&request)
- Enforce completion (semantic barrier synchronization) with MPI_Wait(&request)
- Completion possible when all processes have called MPI_Ibarrier(); (not MPI_Wait();)

Non-blocking barrier???

…is actually useful, processes (locally) notify, and later enforce synchronization

Non-blocking collectives semantic rules

- If some process calls non-blocking collective A on communicator, all other processes must eventually also call non-blocking collective A on (same) communicator
- Non-blocking collectives called this way match
- Order of completion of outstanding collectives may not be same as call order
- Completion of some out-standing non-blocking collective does not imply anything about completion of other out-standing, non-blocking calls (neither collective, nor collective), not even for calls performed "before"

Non-blocking and blocking collectives (for the same operation) do not match

Process i:

```
MPI_Request req1, req2;

MPI_Ibcast(…,comm,&req1);
MPI_Igather(…,comm,&req2);
MPI_Bcast(…,comm);

MPI_Wait(&req2);

MPI_Wait(&req1);
```

Nothing can be inferred about MPI_Ibcast();

Process j:

```
MPI_Request req1, req2;

MPI_Ibcast(…,comm,&req1);
MPI_Igather(…,comm,&req2);
MPI_Wait(&req1);
MPI_Wait(&req2);

MPI_Bcast(comm,&newcomm);
```

Process i:

```
MPI_Request req1, req2;

MPI_Ibcast(…,comm,&req1);
MPI_Igather(…,comm,&req2);
MPI_Bcast(…,comm);

MPI_Wait(&req2);

MPI_Wait(&req1);
```

Process j:

```
MPI_Request req1, req2;

MPI_Igather(…,comm,&req2);
MPI_Ibcast(…,comm,&req1);

MPI_Wait(&req1);
MPI_Wait(&req2);

MPI_Bcast(comm,&newcomm);
```

Wrong: Not same sequence of collective calls

Process i:

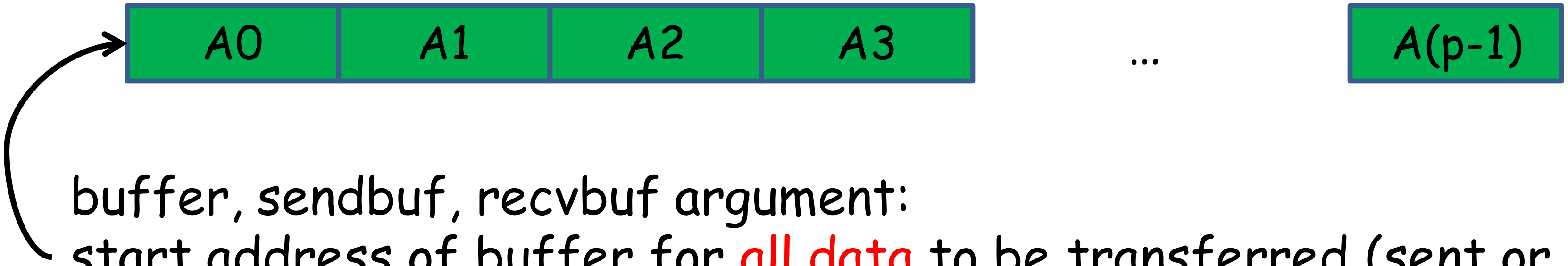
```
MPI_Request req1, req2;

MPI_Igather(…,comm,&req2);
MPI_Bcast(…,comm);
MPI_Ibcast(…,comm,&req1);

MPI_Wait(&req2);

MPI_Wait(&req1);
```

Process j:

```
MPI_Request req1, req2;

MPI_Igather(…,comm,&req2);
MPI_Ibcast(…,comm,&req1);

MPI_Wait(&req1);
MPI_Wait(&req2);

MPI_Bcast(comm,&newcomm);
```

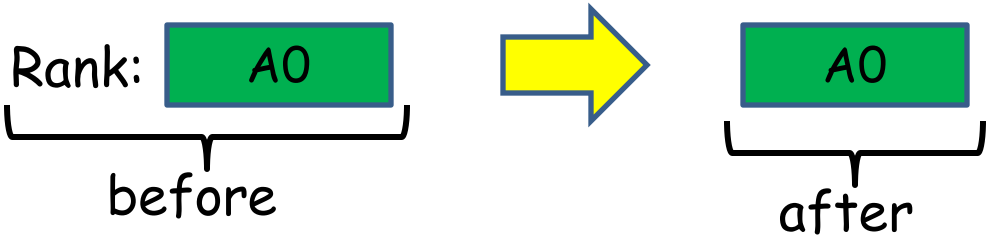
Wrong: Non-blocking and blocking calls do not match

Regular collectives

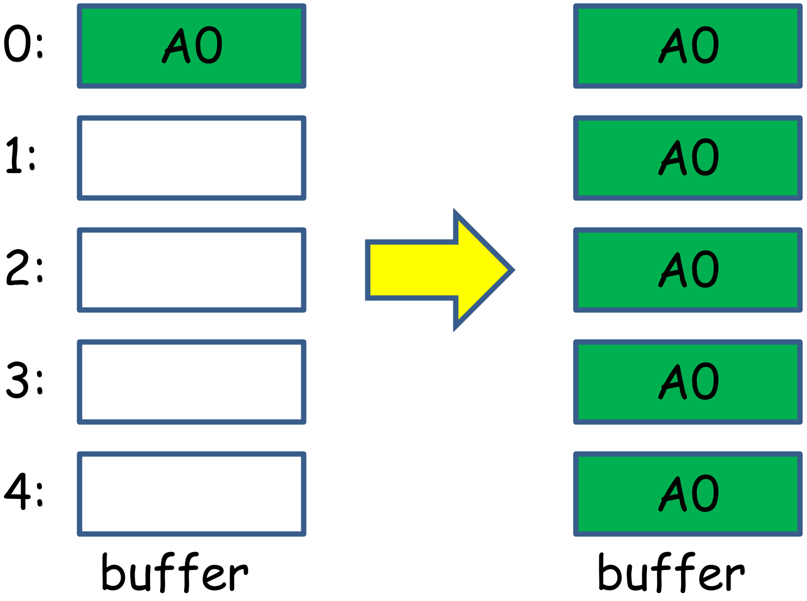


buffer, sendbuf, recvbuf argument:
start address of buffer for all data to be transferred (sent or received)

Segments to/from other processes all have the same size (count) and datatype



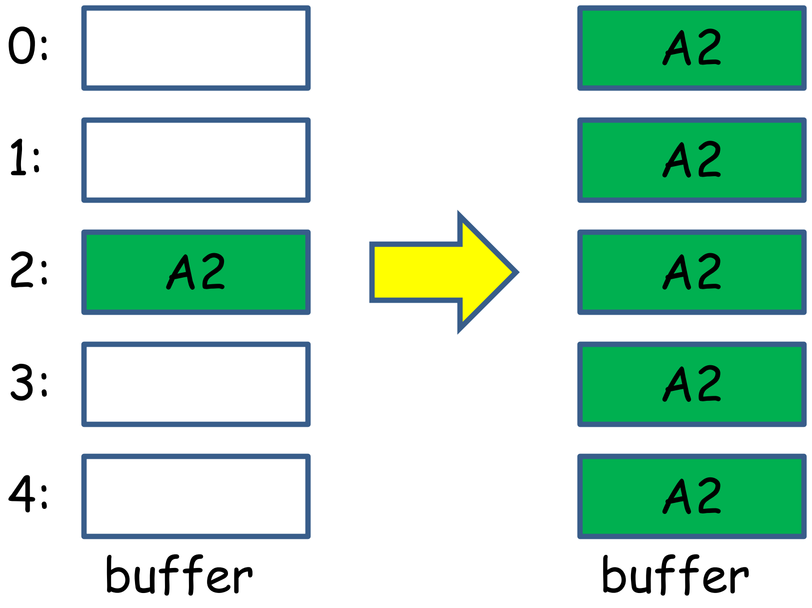
```
MPI_Bcast(void* buffer, int count, MPI_Datatype type,
          int root, MPI_Comm comm)
```



Example: root==0

All processes Bcast with same root, buffer with same type signature (same count for basic datatypes like MPI_FLOAT)

```
MPI_Bcast(void* buffer, int count, MPI_Datatype type,
          int root, MPI_Comm comm)
```



Example: root==2

All processes Bcast with same root, buffer with same type signature (same count for basic datatypes like MPI_FLOAT)

MPI collectives requirement

Collective functions must be called with consistent arguments:

- same root
- matching type signatures (in particular: pairwise same size)
- Note: number of elements sent and received must match exactly (unlike Send-Recv: sent≤recv and Get/Put)
- Same op (MPI_Reduce etc.)

```
int matrixdims[3]; // 3 dimensional matrix
if (rank==0) {
  MPI_Bcast(matrixdims,2,MPI    T,0,c     m);
} else {
  // do something on non-ro     fi
  MPI_Bcast(matrixdims,3,MPI_    T,0       m);
  // uhuh, Bcast probably works,      e later…
}
```

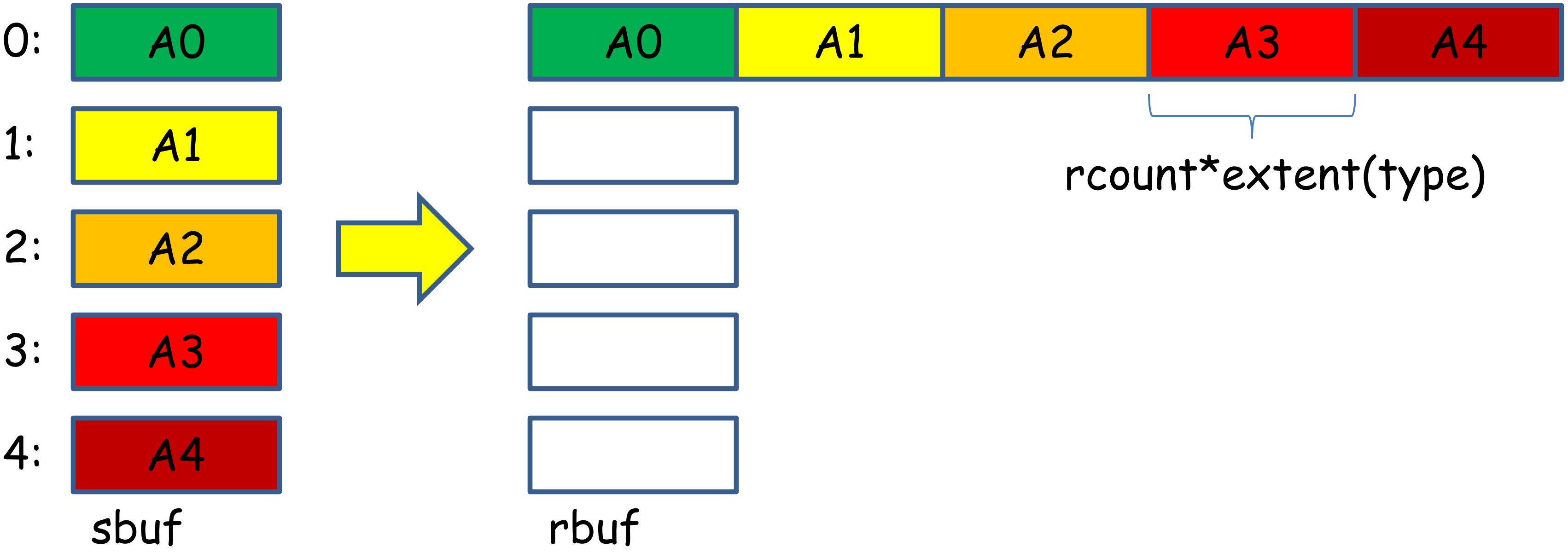
MPI collectives requirement

Collective functions must be called with consistent arguments:

- same root
- matching type signatures (in particular: pairwise same size)
- Note: number of elements sent and received must match exactly (unlike Send-Recv: sent≤recv and Get/Put)
- Same op (MPI_Reduce etc.)

MPI is designed for high performance. No (communication requiring) argument checking, it is user responsibility to write correct programs

```
MPI_Gather(void* sbuf, int scount, MPI_Datatype stype,
           void *rbuf, int rcount, MPI_Datatype rtype,
           int root, MPI_Comm comm)
```



Block from process i is stored at rbuf+i*rcount*extent(rtype)
rcount is count of one block, not of whole rbuf

```
MPI_Gather(void* sbuf, int scount, MPI_Datatype stype,
           void *rbuf, int rcount, MPI_Datatype rtype,
           int root, MPI_Comm comm)
```

- Root process also gathers from itself

- Receive arguments significant at root only

- MPI_IN_PLACE argument as sbuf at root indicates that block from root is already "in place"

- One receive datatype determines structure of all received blocks. Sometimes a problem

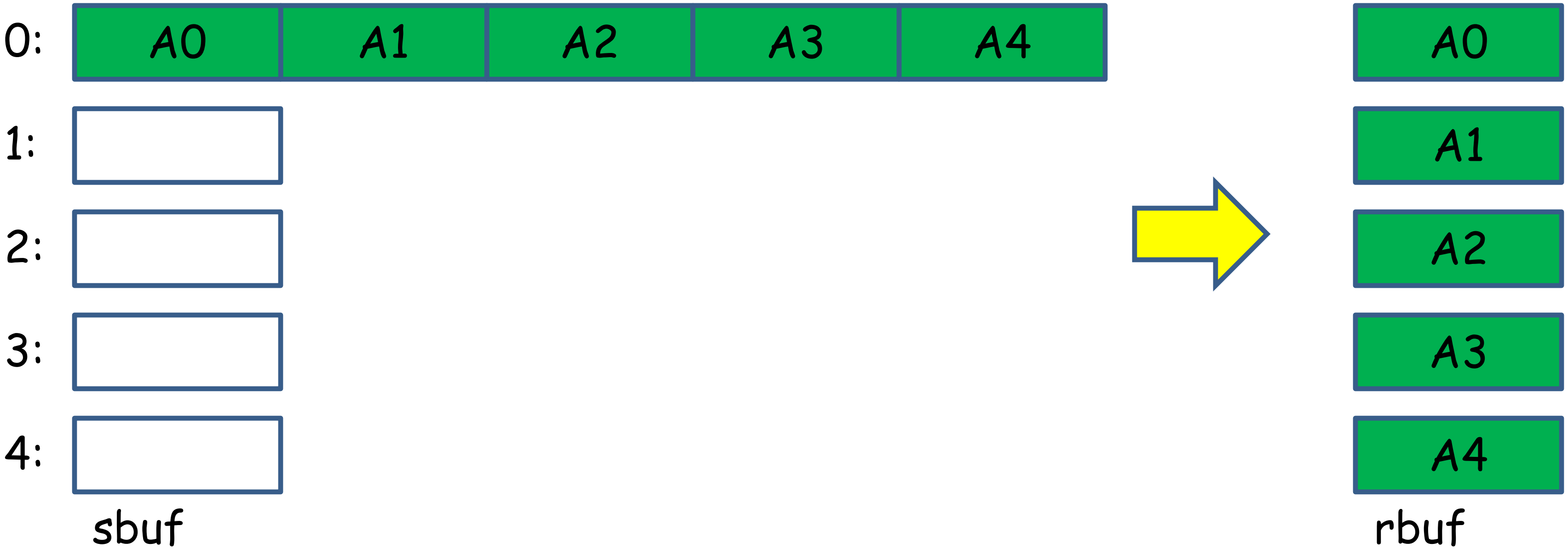
```
MPI_Gather(void* sbuf, int scount, MPI_Datatype stype,
           void *rbuf, int rcount, MPI_Datatype rtype,
           int root, MPI_Comm comm)
```

Possible implementation

```
if (rank==root) {
  for (…i!=root…) {
    MPI_Recv(rbuf+i*rcount*extent(rtype),rcount,rtype,
                      i,GATTAG,comm,MPI_STATUS_IGNORE);
  }
  MPI_Sendrecv(sbuf,…,root,…,
                        rbuf+root*rcount*extent(rtype),…,root,…);
} else MPI_Send(sbuf,scount,stype,root,GATTAG,comm);
```

Semantic equivalence, MPI_Gather() typically not implemented this way (except for large problems)

```
MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype,
            void *rbuf, int rcount, MPI_Datatype rtype,
            int root, MPI_Comm comm)
```



Block from root is stored at sbuf+i*scount*extent(stype)

```
MPI_Scatter(void* sbuf, int scount, MPI_Datatype stype,
            void *rbuf, int rcount, MPI_Datatype rtype,
            int root, MPI_Comm comm)
```

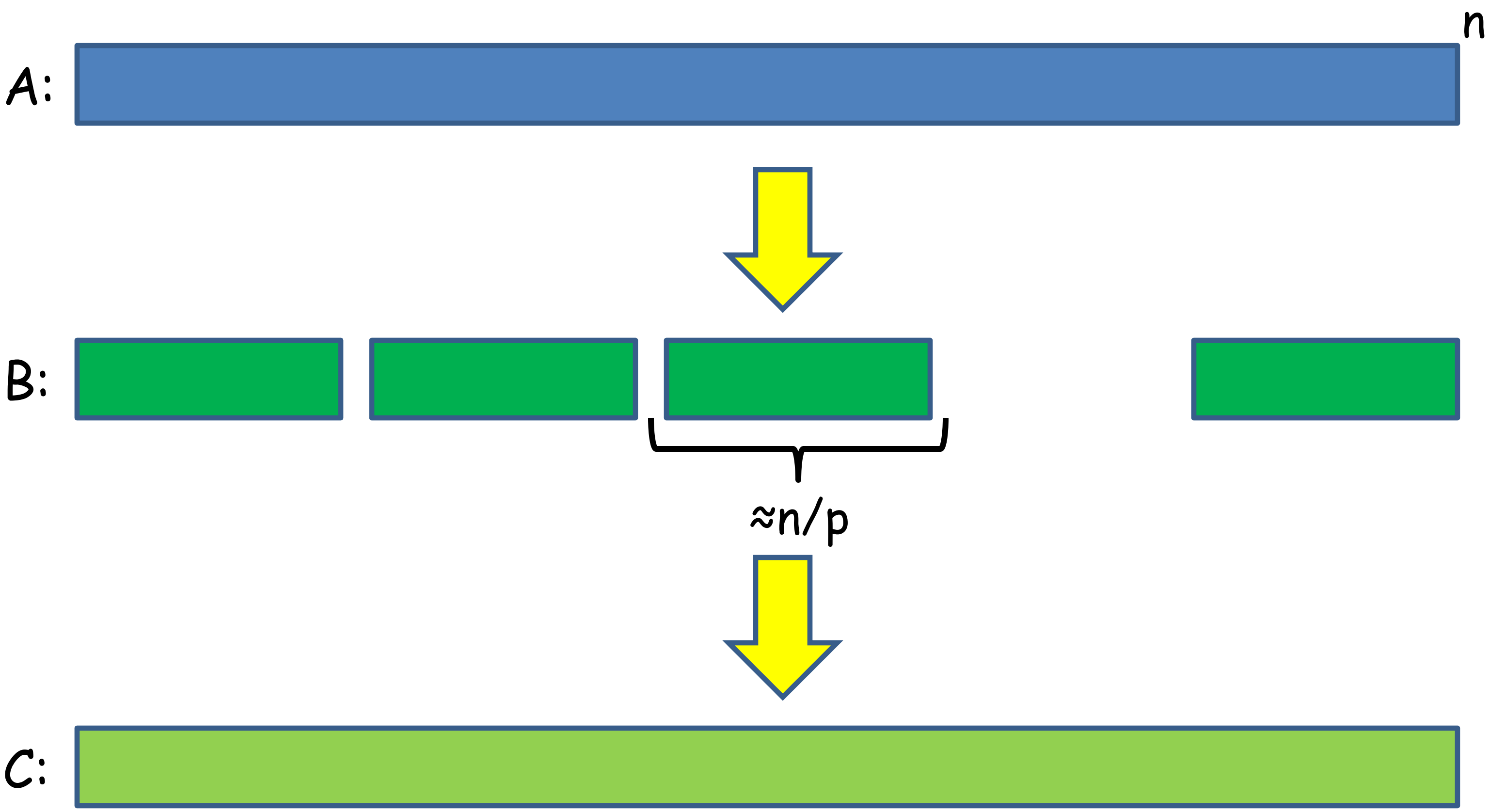
•Root process also scatters to itself

•Send arguments significant at root only

•MPI_IN_PLACE argument as rbuf at root indicates that block from root is already "in place"

•One receive datatype determines structure of all sent blocks. Sometimes a problem

Example: Distribute an initial array, collecting result

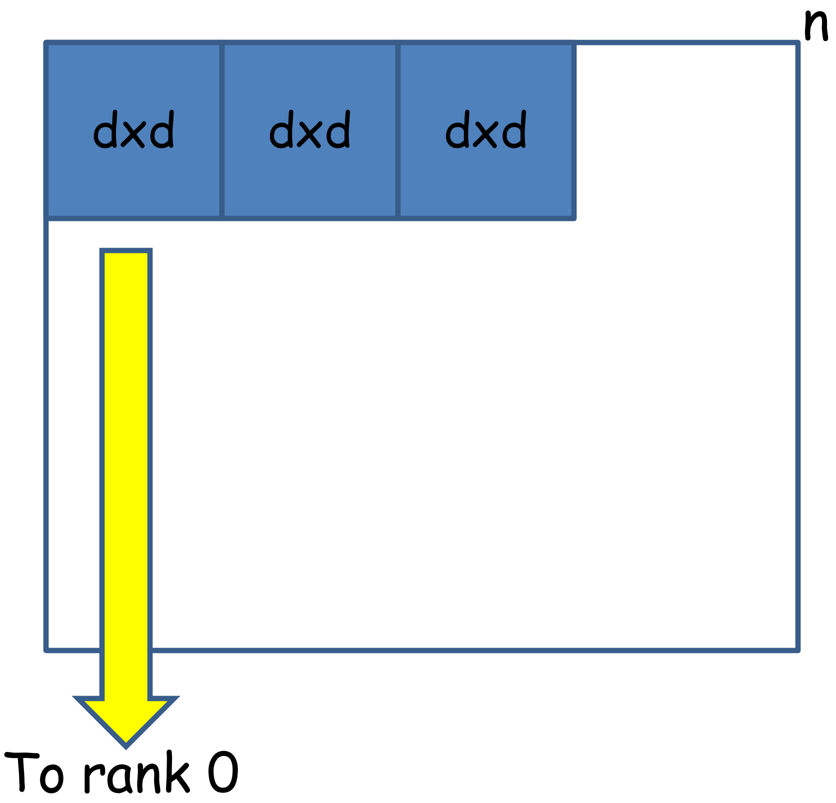


Tacitly assumed p divides n. If not, irregular MPI_Scatterv()

```
int root = 0;
MPI_Comm_size(comm,&p);
MPI_Comm_rank(comm,&r);
assert(n%p==0);
if (rank==root) {
  int *a = (int*)malloc(n*sizeof(int));
  int *c = (int*)malloc(n*sizeof(int));
  for (i=0; i<n; i++) a[i] = <init>;
}
int *b = (int*)malloc((n/p)*sizeof(int));
MPI_Scatter(a,n/p,MPI_INT,
            b,n/p,MPI_INT,root,comm);

… // compute on b: all processes

MPI_Gather(b,n/p,MPI_INT,
           c,n/p,MPI_INT,root,comm);
```

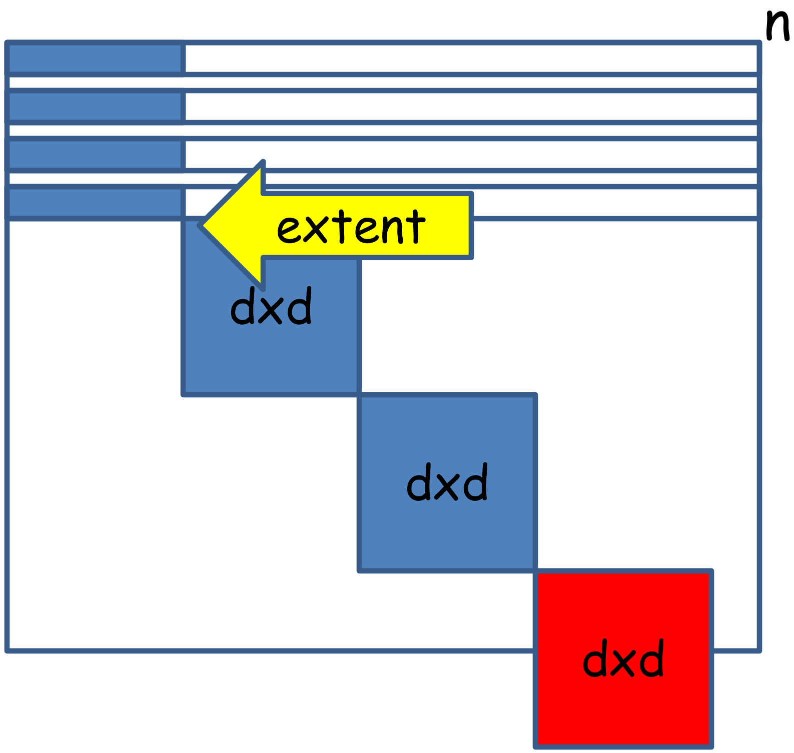
Argument is size per rank-block

Challenge: Scattering submatrices



1. Describe submatrix as vector, block of d elements, stride n
2. MPI_Scatter() will not work…

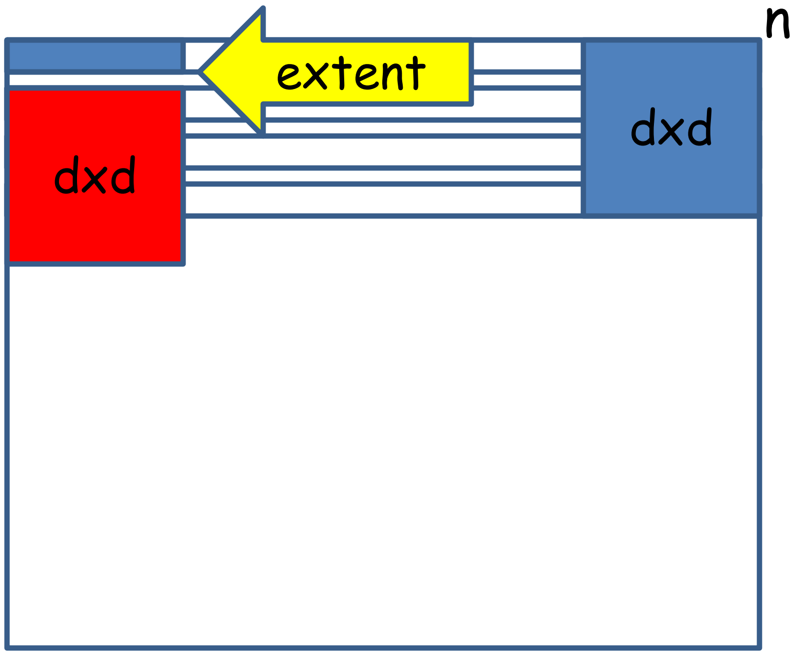
Challenge: Scattering submatrices



1. Describe submatrix as vector, block of d elements, stride n

2. MPI_Scatter() will not work…

Challenge: Scattering submatrices



1. Describe submatrix as vector, block of d elements, stride n

2. MPI_Scatter() will not work…

3. Changing extent with MPI_Type_create_resize() does not solve the problem

Solution (not ideal): Use irregular MPI_Scatterv(), careful with displacement, units is extent of sendtype

Differences to point-to-point communication:

- Collective operations do not have tag argument
- Amount of data from process i to process j must equal amount of data expected by process j from process i
- Buffers of size 0 do not have to be sent

Process i:

```
MPI_Bcast(buffer,0,MPI_CHAR,…,root,comm);
```

Process j:

```
MPI_Bcast(buffer,0,MPI_CHAR,…,root,comm);
```

Correct. May be implemented as no-op, no synchronization effect whatsoever

- Collective operations do not have tag argument
- Amount of data from process i to process j must equal amount of data expected by process j from process i
- Buffers of size 0 do not have to be sent

Process i:

```
MPI_Send(buffer,0,MPI_CHAR,j,TAG,comm);
```

Process j:

```
MPI_Recv(buffer,0,MPI_CHAR,j,TAG,
         comm,&status);
```

Correct. But an empty message must be sent

- Collective operations do not have tag argument
- Amount of data from process i to process j must equal amount of data expected by process j from process i
- Buffers of size 0 do not have to be sent

Process i:

```
MPI_Send(buffer,0,MPI_CHAR,j,TAG,comm);
```

Process j:

```
MPI_Recv(buffer,10,MPI_CHAR,j,TAG,
         comm,&status);
```

Correct. But an empty message must be sent, since receive count could be greater 0 and sender cannot know this

A barrier implementation?

```
MPI_Gather(sendbuf,0,MPI_BYTE,recvbuf,0,MPI_INT,0,
           comm);
/* blocking, so now rank 0 knows that all processes have
   arrived at barrier */
MPI_Scatter(sendbuf,0,MPI_BYTE,recvbuf,0,MPI_INT,0,
            comm);
// and now all other processes know
```

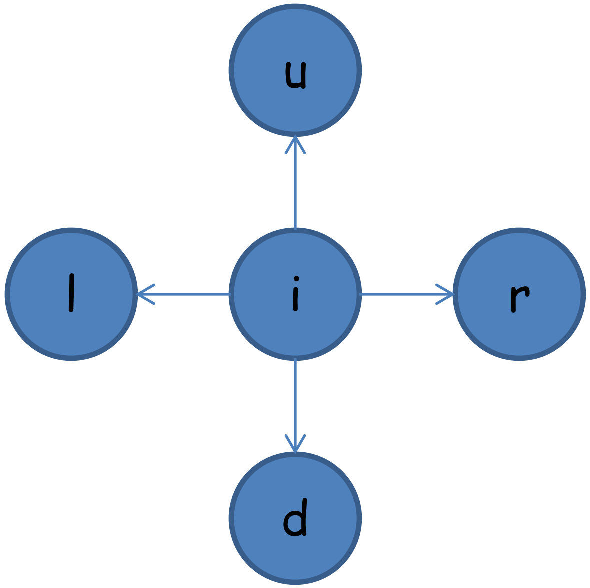
May work, but is not correct implementation of MPI_Barrier(): it tacitly assumes that communication will take place and that MPI_Gather() and MPI_Scatter() synchronizes

Performance: at least two (log p) rounds of communication, better algorithms exist

Definition: Collective operations are not synchronizing. MPI programs relying on synchronizing assumptions for collective operations are unsafe.

- Except for MPI_Barrier(), MPI collective operations are non-synchronizing
- Blocking does not mean “synchronizing“, no assumptions can be made on what other processes have done

2d stencil: Gather or scatter on small communicators



1. Create small communicators, each process belongs to 5 communicators: a) root in 1, b) left, right, up, down partner in 4 other
2. MPI_Scatter(…,rootcomm);
3. MPI_Scatter(…,ucomm);
4. MPI_Scatter(…,dcomm);
5. MPI_Scatter(…,lcomm);
6. MPI_Scatter(…,rcomm);

•DEADLOCKS

•Problem with scatter, different neighbors have different datatypes (row or column), but MPI_Scatter() has only one

🚫

```
MPI_Comm stencilcomm[5]
  {rootcomm, ucomm, d      lco  , rcomm};
for (i=0; i<5; i++)
  MPI_Scatter(sbuf[i] ,sty    stencilcomm[i]);
}
```

Deadlock avoided with non-blocking collectives

```
MPI_Comm stencilcomm[5] =
  {rootcomm, ucomm, dcomm, lcomm, rcomm};
MPI_Request stencilreq[5];
for (i=0; i<5; i++) {
  MPI_Iscatter(sbuf[i],1,stype,…,stencilcomm[i],
               &stencilreq[i]);
}
MPI_Waitall(5,stencilreq,MPI_STATUSES_IGNORE);
```

- The datatype issue (scatter with differently structured blocks) cannot be resolved with current MPI_Scatter() variants, there is no MPI_Scatterw()

- For some problems, zero-copy implementations are not possible

[Jesper Larsson Träff, Antoine Rougier:
Zero-copy, Hierarchical Gather is not possible with MPI Datatypes and Collectives. EuroMPI/ASIA 2014: 39]

```
MPI_Comm_split(MPI_Comm comm, int color, int key,
               MPI_Comm *newcomm)
```

Create new communicators by “splitting” old one: all processes calling with the same color will belong to the same, new communicator. Key argument determines relative order of process, process calling smaller key than some other process will have smaller rank in new communicator

```
MPI_Comm_create(MPI_Comm comm, MPI_Group group,
                MPI_Comm *newcomm)
```

Create new communicators, processes belong to the same group will be in same communicator

Both operations are collective over the old communicator

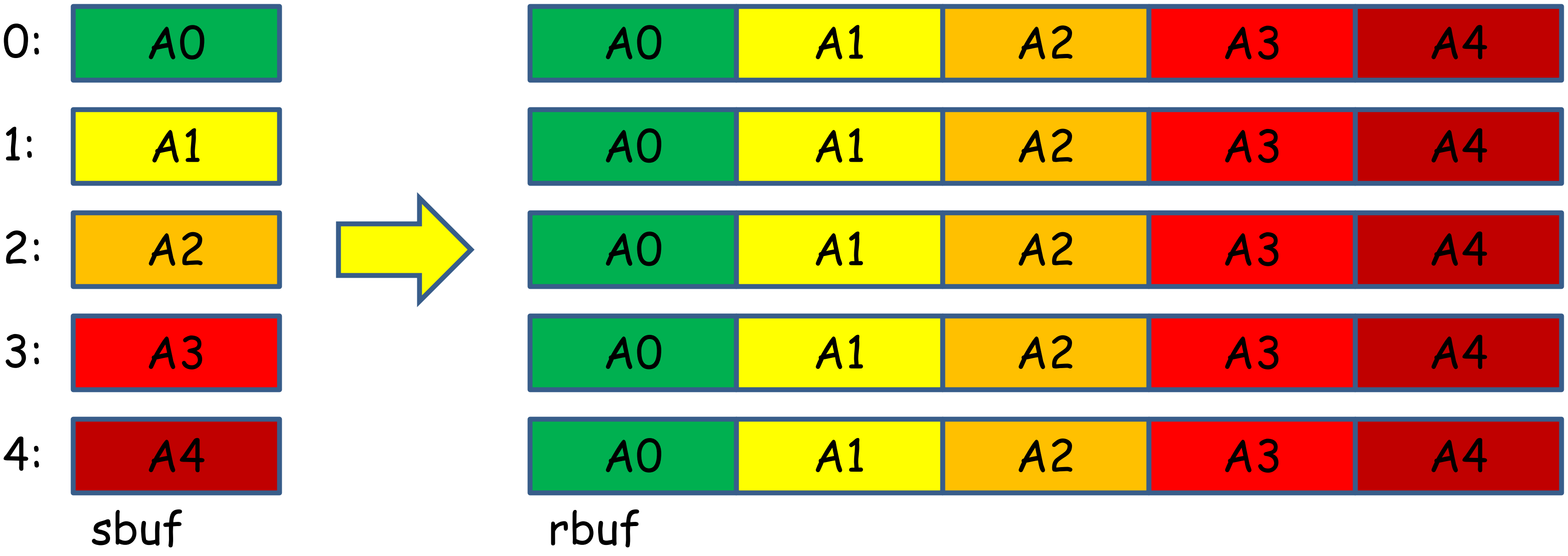
```
MPI_Comm_create_group(MPI_Comm comm, MPI_Group group,
                      MPI_Comm *newcomm)
```

Collective only over process in calling group

```
MPI_Comm_idup(MPI_Comm comm, MPI_Comm *newcomm)
```

A non-blocking version of the communicator duplication function

```
MPI_Allgather(void* sbuf, int scount,
              MPI_Datatype stype,
              void *rbuf,
              int rcount, MPI_Datatype rtype,
              MPI_Comm comm)
```



	sbuf		rbuf				
0:	A0	→	A0	A1	A2	A3	A4
1:	A1		A0	A1	A2	A3	A4
2:	A2		A0	A1	A2	A3	A4
3:	A3		A0	A1	A2	A3	A4
4:	A4		A0	A1	A2	A3	A4

Block from process i is stored at rbuf+i*rcount*extent(rtype)

```
MPI_Allgather(void* sbuf, int scount,
              MPI_Datatype stype,
              void *rbuf,
              int rcount, MPI_Datatype rtype,
              MPI_Comm comm)
```

•Aka broadcast-to-all, all processes get result of gather operation

•MPI_IN_PLACE argument as sbuf indicates that block from root is already "in place" in the corresponding position of the rbuf

•One receive datatype determines structure of all sent blocks. Sometimes a problem

```
MPI_Allgather(sbuf,…rbuf,rcount,rtype,…comm);
```

equivalent to

```
MPI_Gather(sbuf,…,rbuf,…,0,comm);
MPI_Bcast(rbuf,size*rcount,rtype,…,0,comm);
```

and

```
for (i=0; i<size; i++) { // all-to-all broadcast
  if (i==rank) MPI_Bcast(sbuf,…,i,comm); else
  MPI_Bcast(rbuf+i*rcount*extent(rtype),…,i,comm);
}
memcpy(rbuf+rank*rcount*extent(rtype),sbuf,…);
```

Performance of MPI library function should be better

Fact:
Much better algorithms for MPI_Allgather() than MPI_Gather()+MPI_Bcast() exist

A good MPI implementation will ensure that "best possible" algorithm is implemented. Expectation: MPI_Allgather() always (all other things being equal) performs better than MPI_Gather()+MPI_Bcast()

Golden MPI rule

- Use collectives for conciseness and performance whereever possible

- Complain to MPI library implementer if performance anomalies are discovered

Performance expectations/guidelines

MPI does not have a performance model

It makes sense to formalize defensible expectations on how certain operations should perform relative to other operations

Expectations can be checked for each MPI library and system by systematic benchmarking

Portability problem: performance expectation anomalies may lead application programmers to spend time on non-portable workarounds

[Jesper Larsson Träff, William D. Gropp, Rajeev Thakur: Self-Consistent MPI Performance Guidelines. IEEE Trans. Parallel Distrib. Syst. 21(5): 698-709 (2010)]

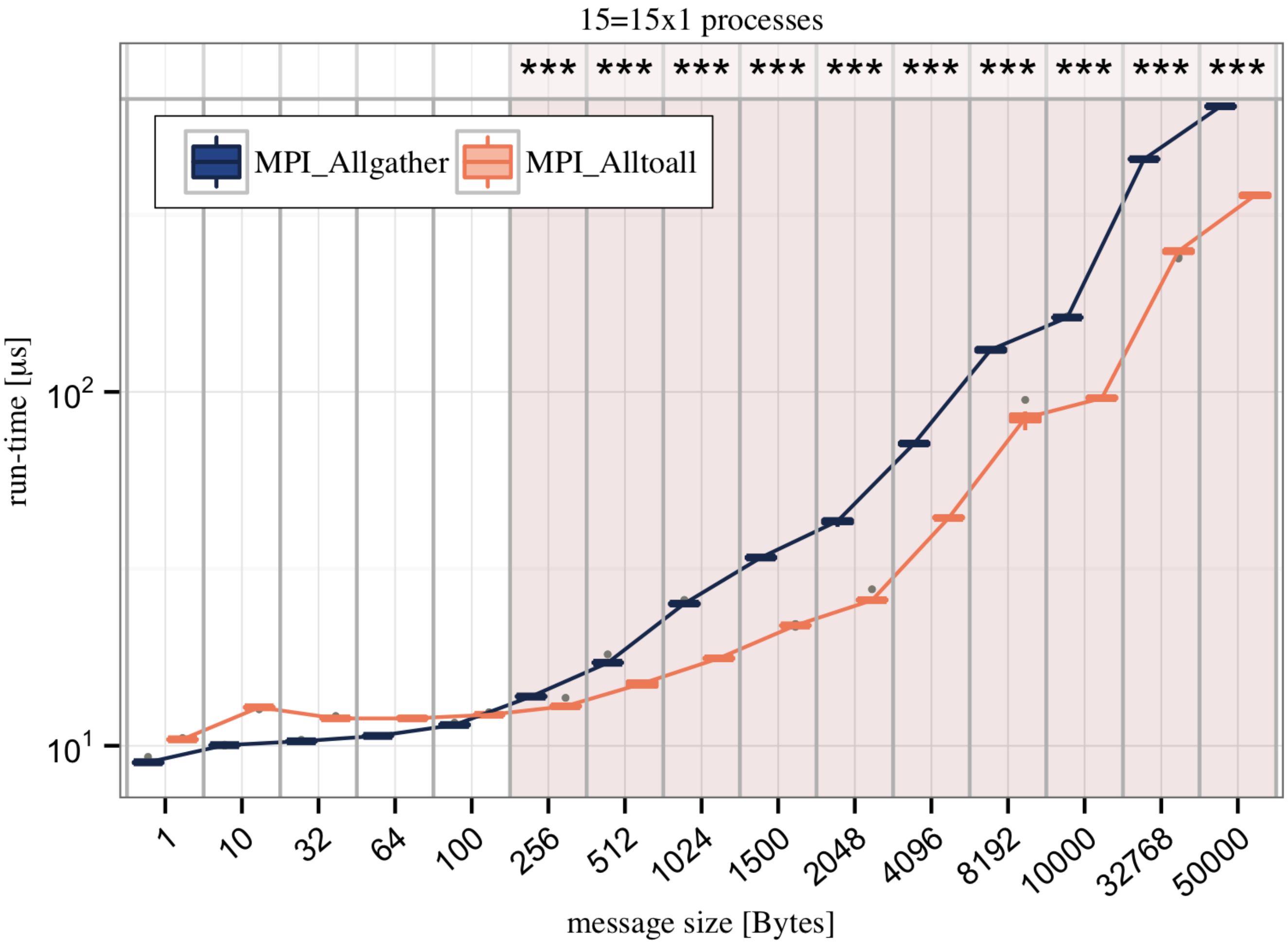
A few example expectations tested at VSC-3:

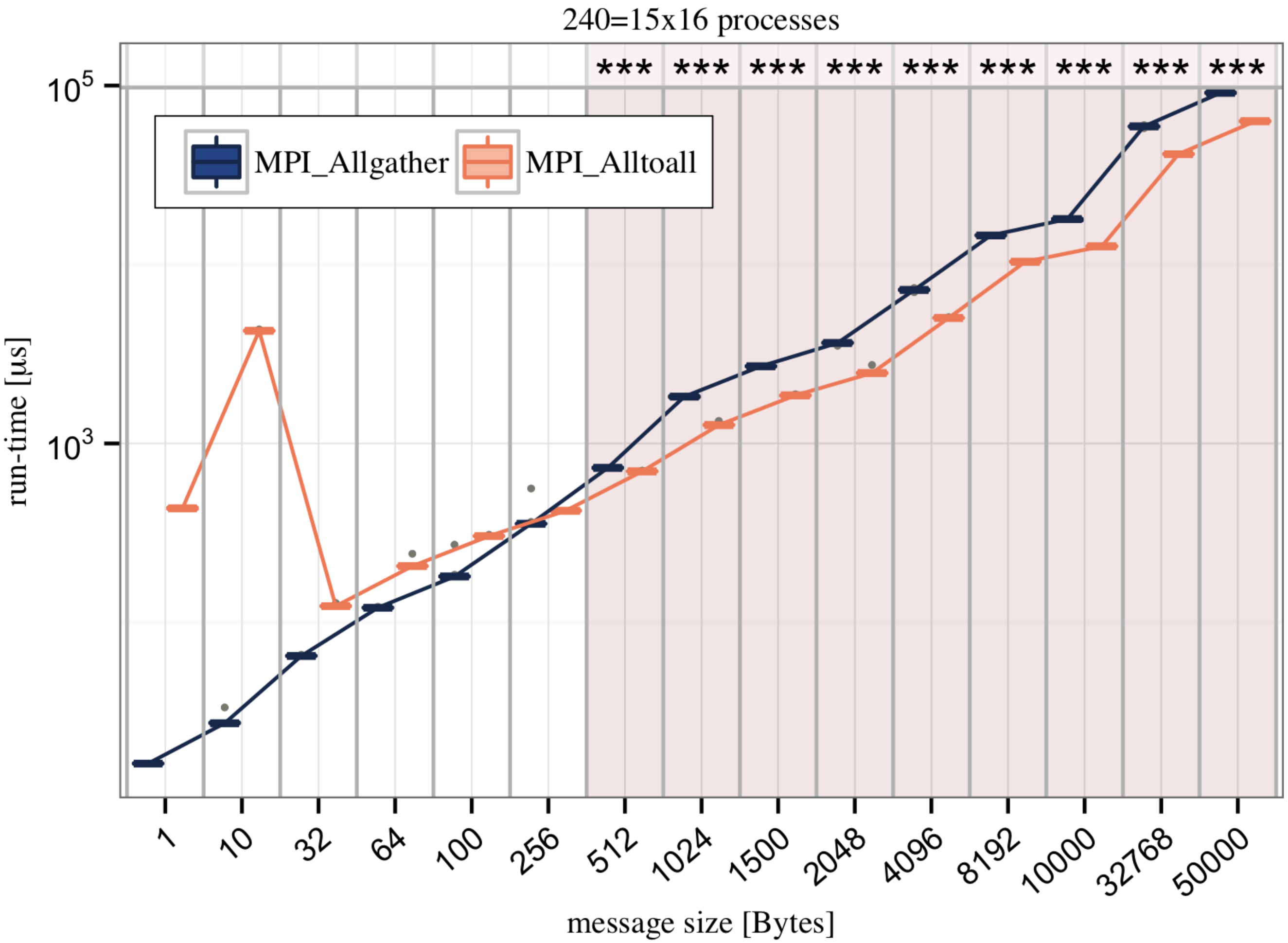
1. MPI_Allgather(n) ≤ MPI_Alltoall(n)
2. MPI_Allreduce (n) ≤ MPI_Reduce(n) + MPI_Bcast(n)
3. MPI_Gather(n) ≤ MPI_Allgather(n)
4. MPI_Reduce(n) ≤ MPI_Allreduce(n)
5. MPI_Scatter(n) ≤ MPI_Bcast(n)

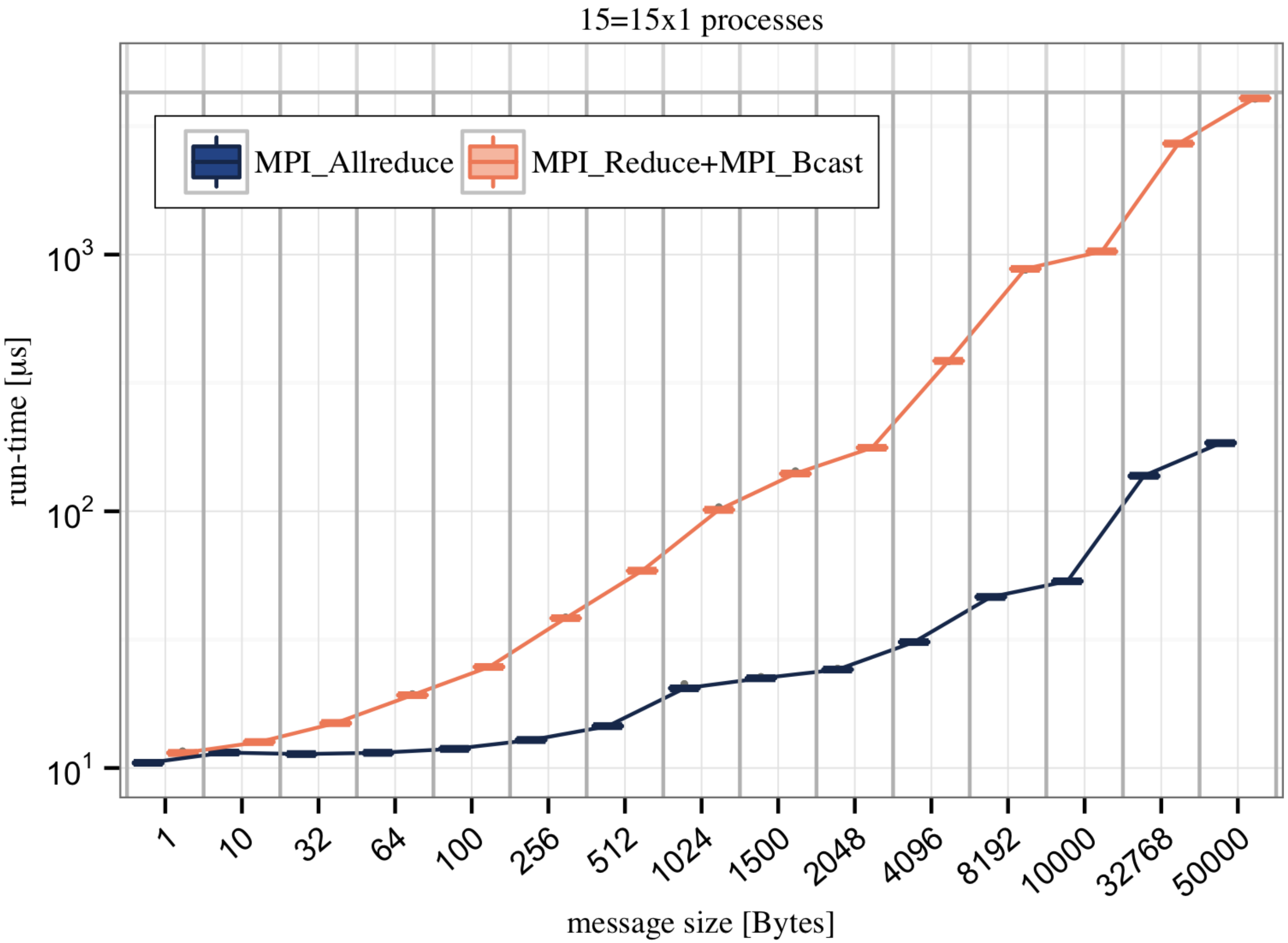
≤: "not slower than"

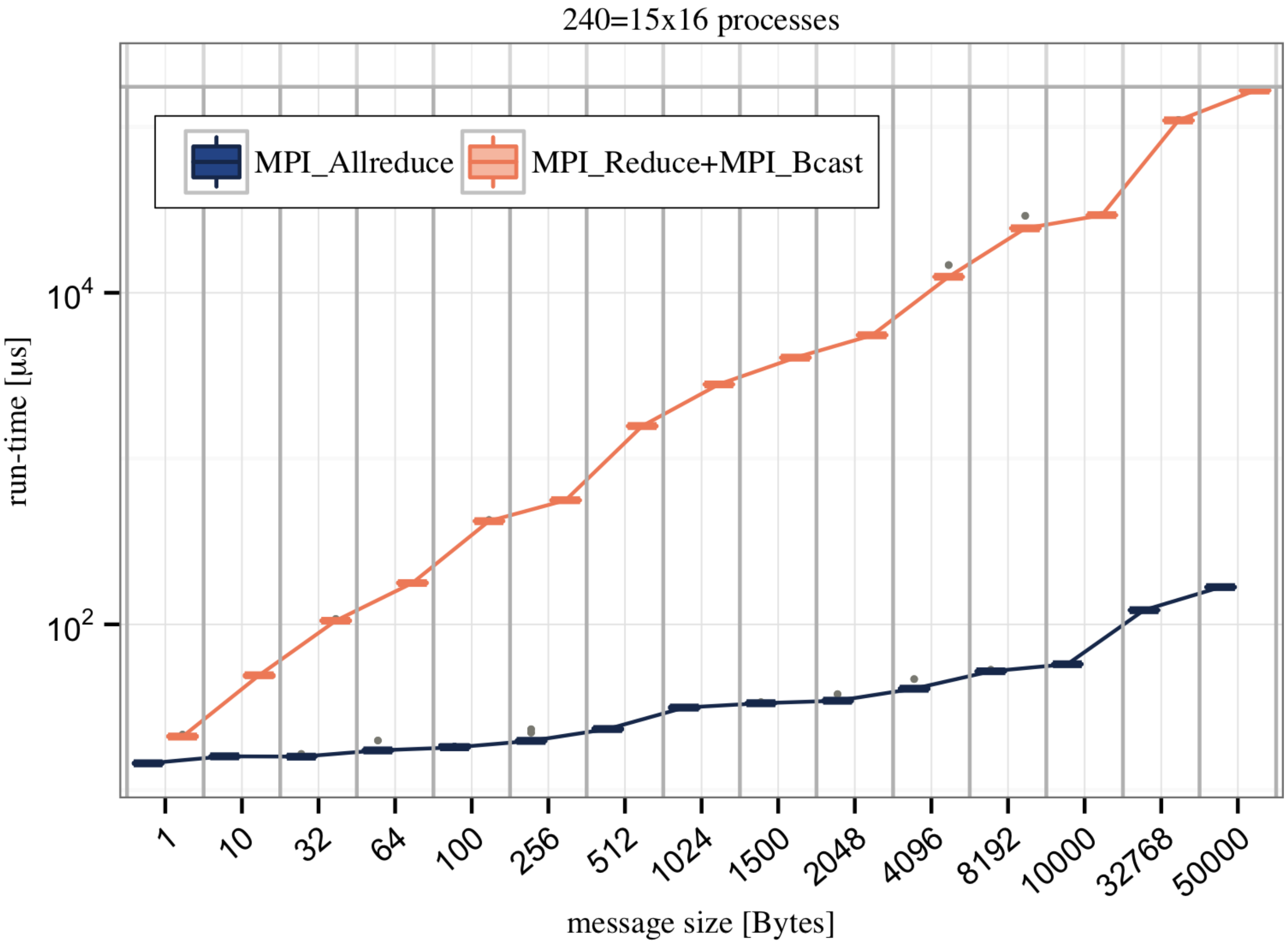
Defenses:

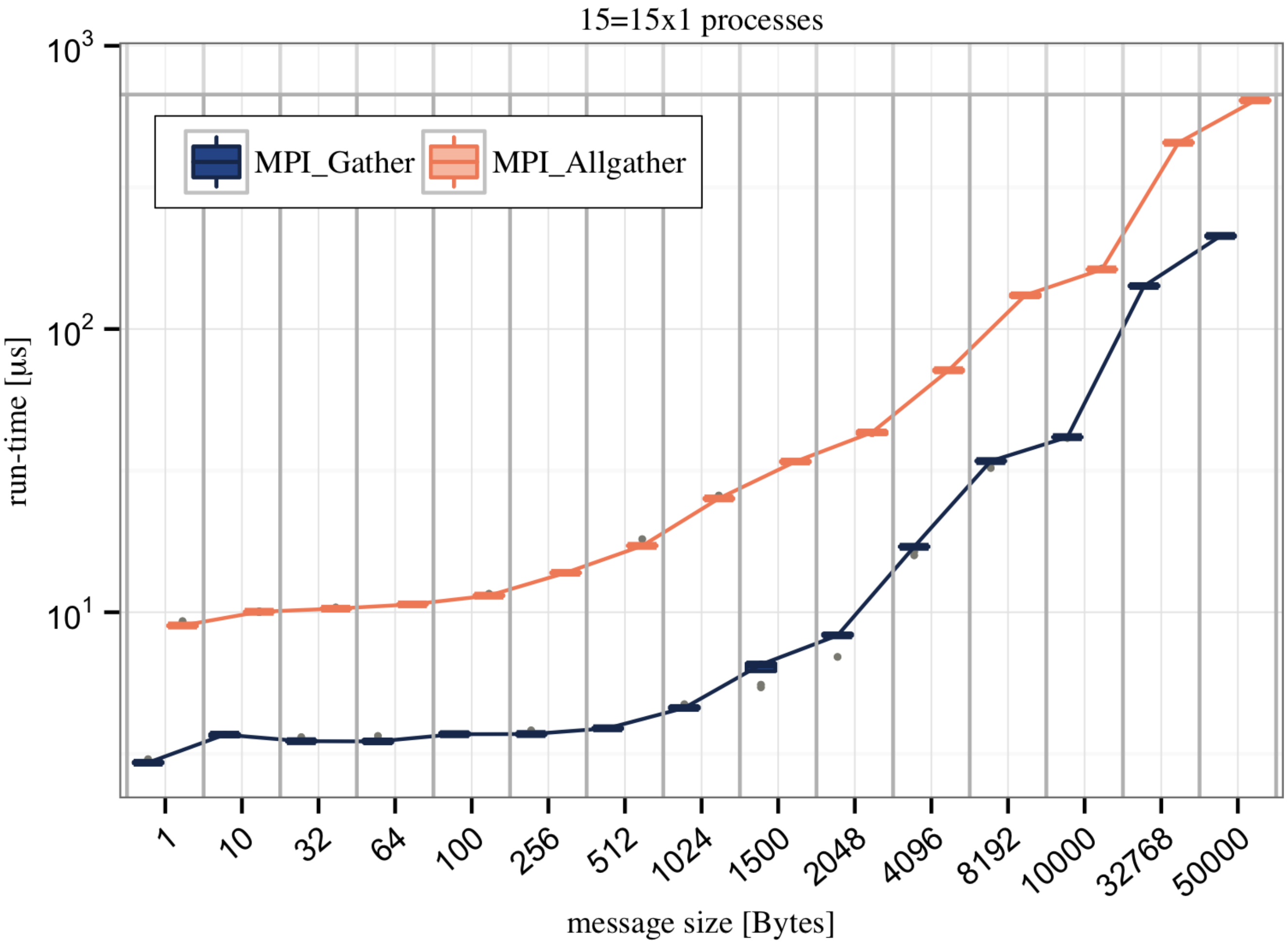
1. Allgather is more specific, same block of data sent to all processes
2. If not, the poor allreduce can be replaced by the composite implementation
3. Gather is more specific, if worse, could be replaced
4. Same
5. Scatter is like bcast where only part of the broadcasted data are significant for a process

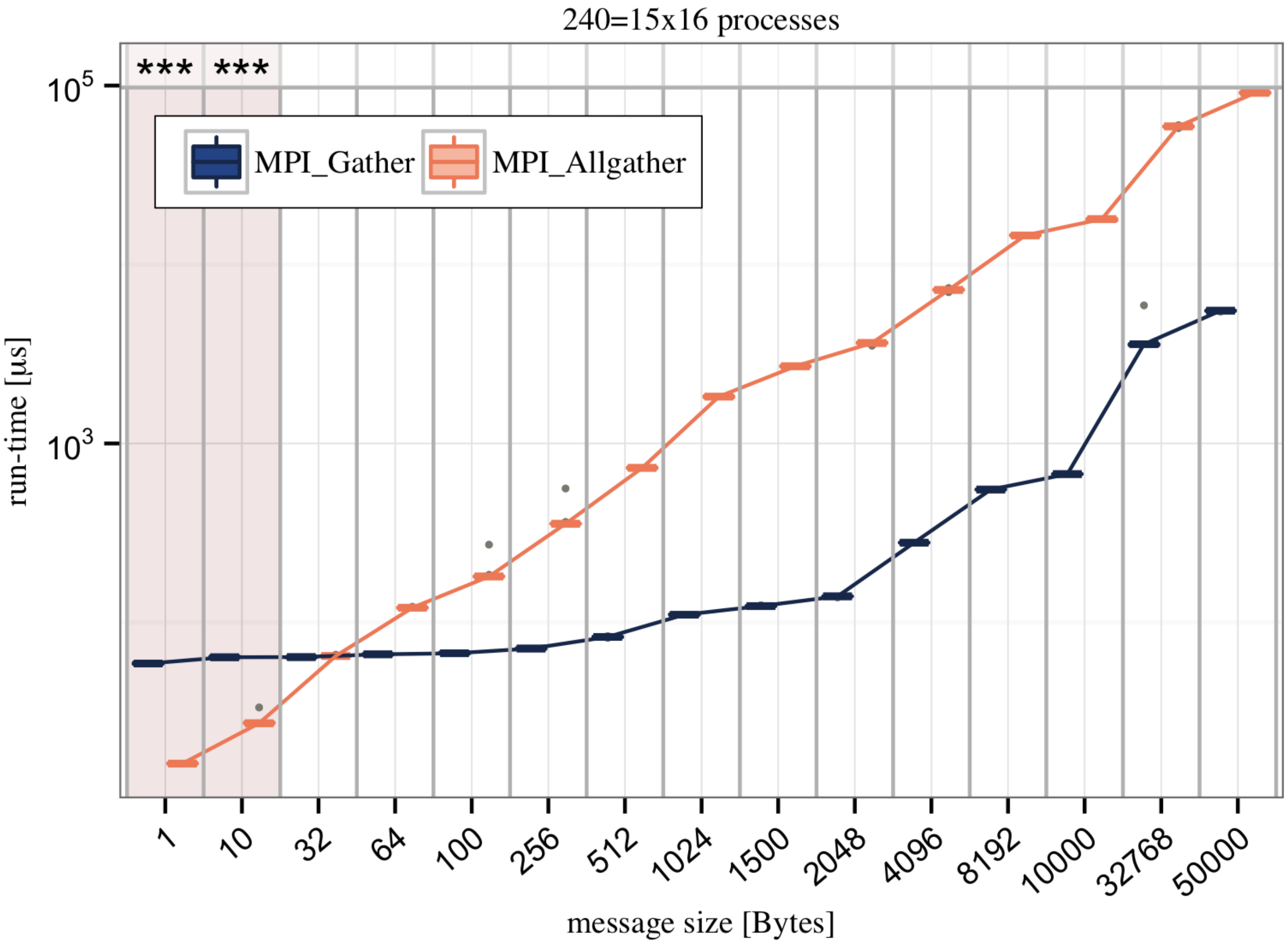

15=15x1 processes
*** *** *** *** *** *** *** *** *** ***
MPI_Allgather
MPI_Alltoall
run-time [µs]
10^2
10^1
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]

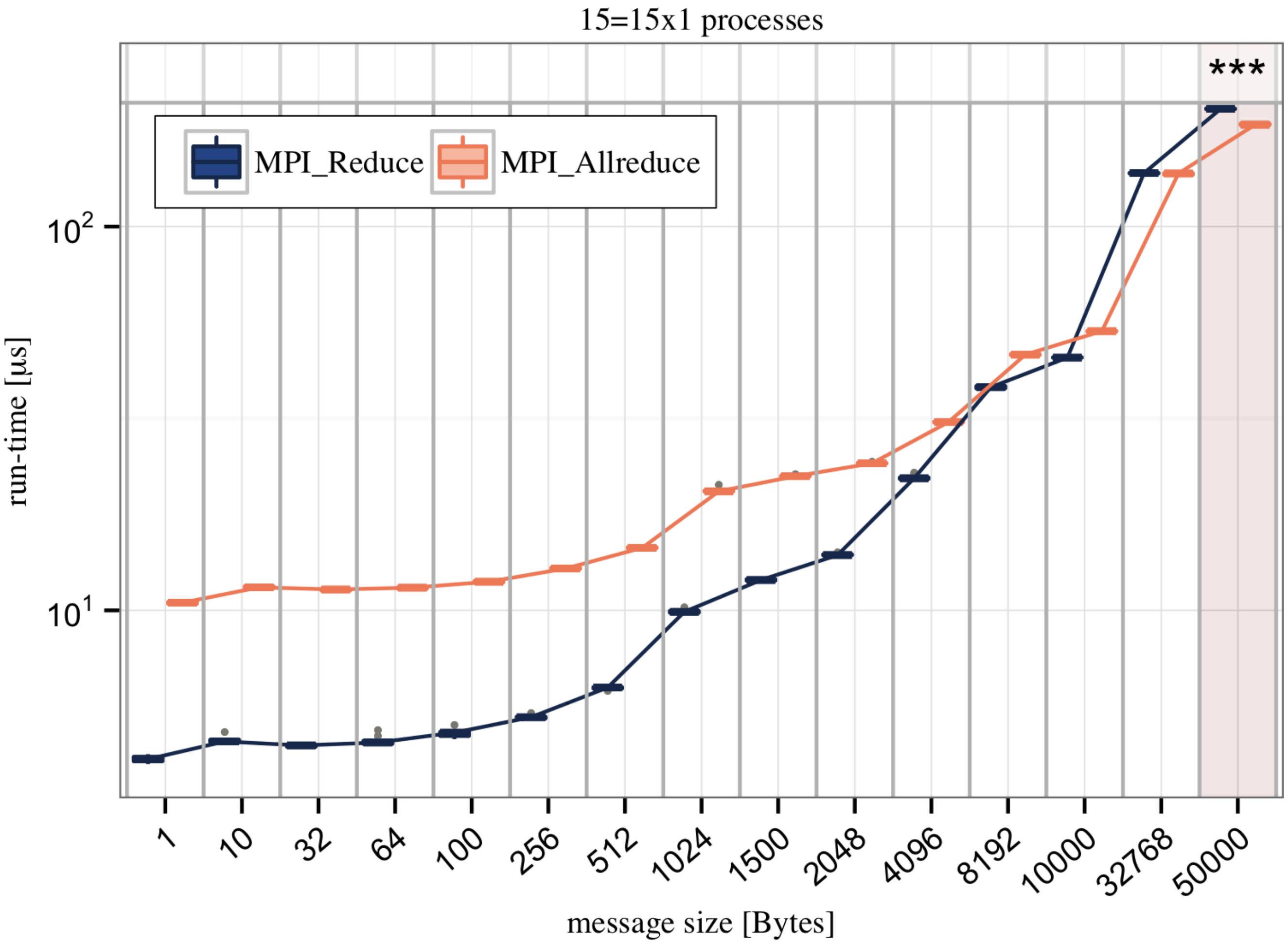

240=15x16 processes
MPI_Allgather
MPI_Alltoall
***
***
***
***
***
***
***
***
***
10^5
10^3
run-time [µs]
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]

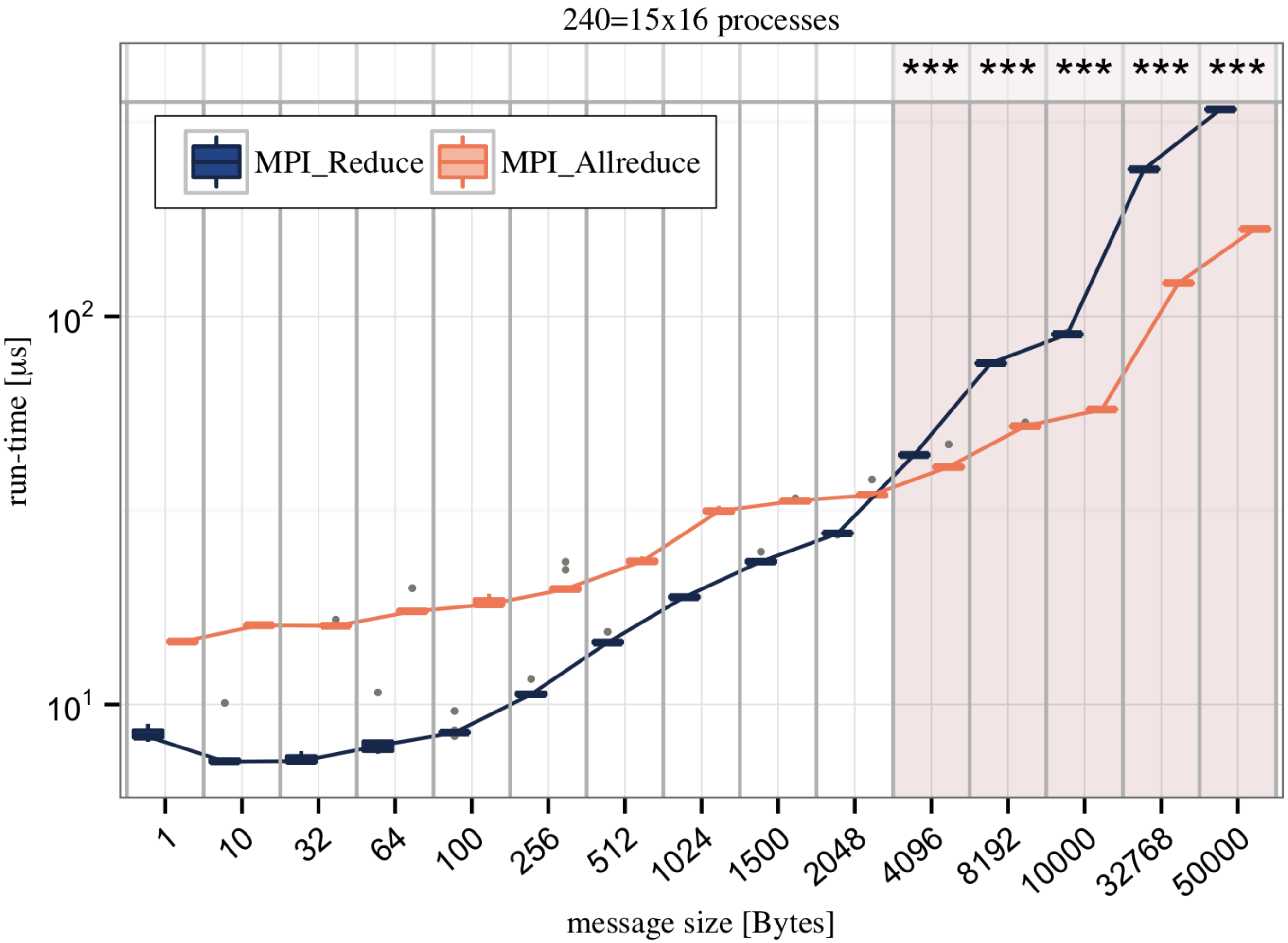

15=15x1 processes
MPI_Allreduce
MPI_Reduce+MPI_Bcast
run-time [µs]
10^3
10^2
10^1
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]


240=15x16 processes
MPI_Allreduce
MPI_Reduce+MPI_Bcast
run-time [μs]
10^4
10^2
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]

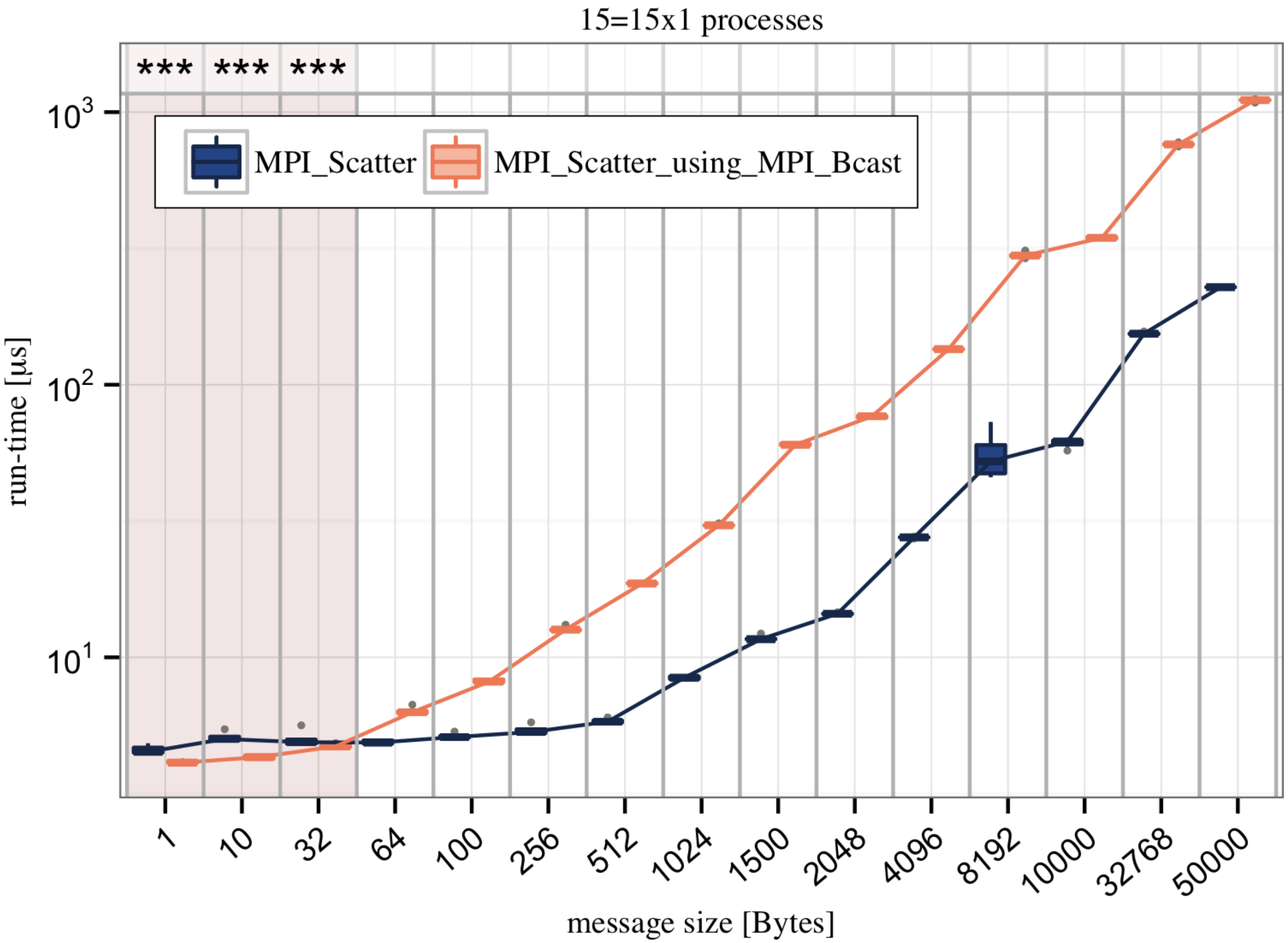

15=15x1 processes
MPI_Gather
MPI_Allgather
run-time [µs]
10^3
10^2
10^1
message size [Bytes]
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000

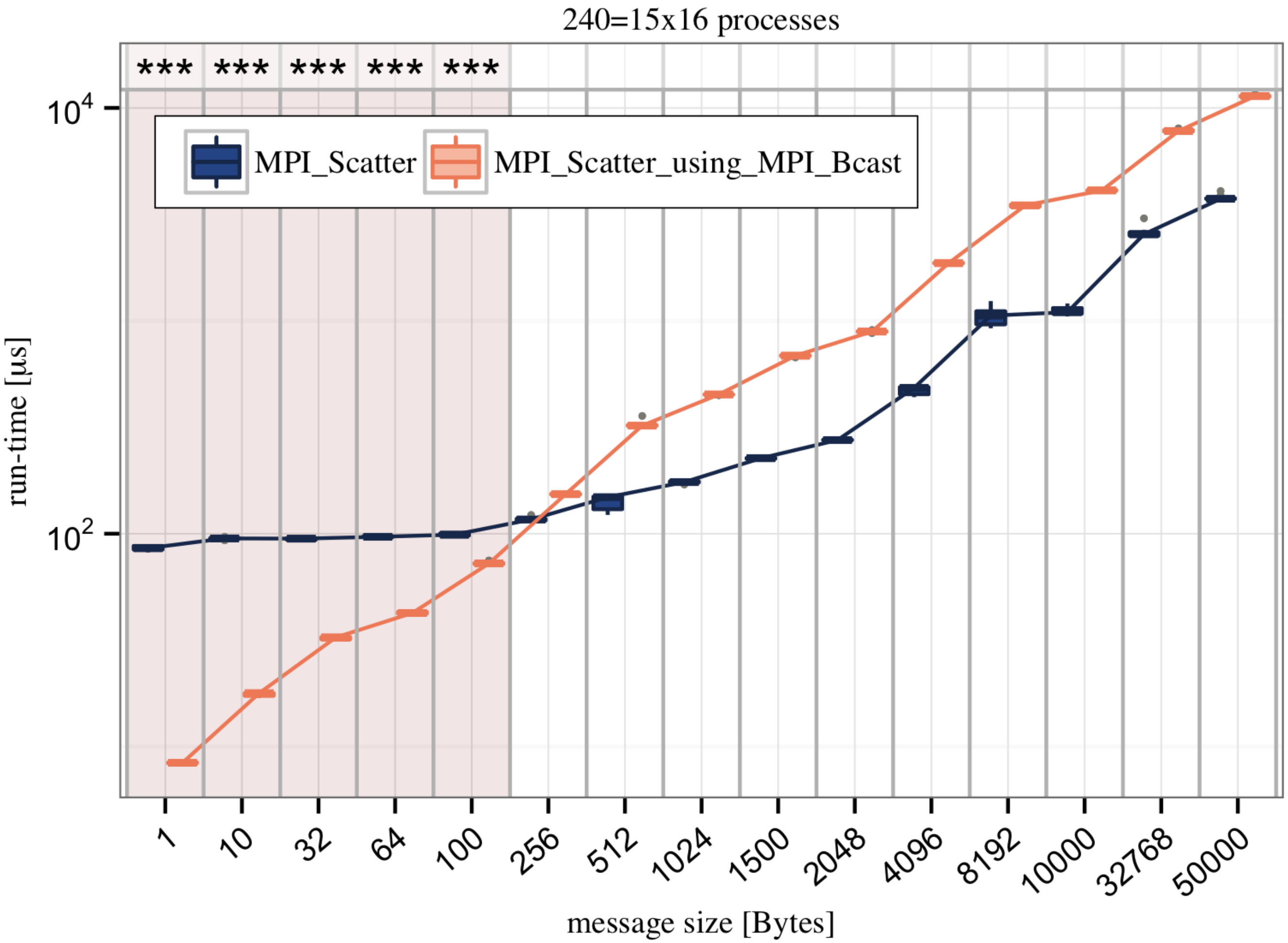

240=15x16 processes
***
***
MPI_Gather
MPI_Allgather
run-time [µs]
10^5
10^3
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]


15=15x1 processes
***
MPI_Reduce
MPI_Allreduce
run-time [µs]
10^2
10^1
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]



240=15x16 processes
MPI_Reduce
MPI_Allreduce
run-time [µs]
10^2
10^1
*** *** *** *** ***
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]


15=15x1 processes
***
***
***
MPI_Scatter
MPI_Scatter_using_MPI_Bcast
run-time [µs]
10^3
10^2
10^1
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]


240=15x16 processes
*** *** *** *** ***
MPI_Scatter
MPI_Scatter_using_MPI_Bcast
run-time [µs]
10^4
10^2
1
10
32
64
100
256
512
1024
1500
2048
4096
8192
10000
32768
50000
message size [Bytes]

```
MPI_Alltoall(void* sbuf, int scount, MPI_Datatype stype,
             void *rbuf, int rcount, MPI_Datatype rtype,
             MPI_Comm comm)
```

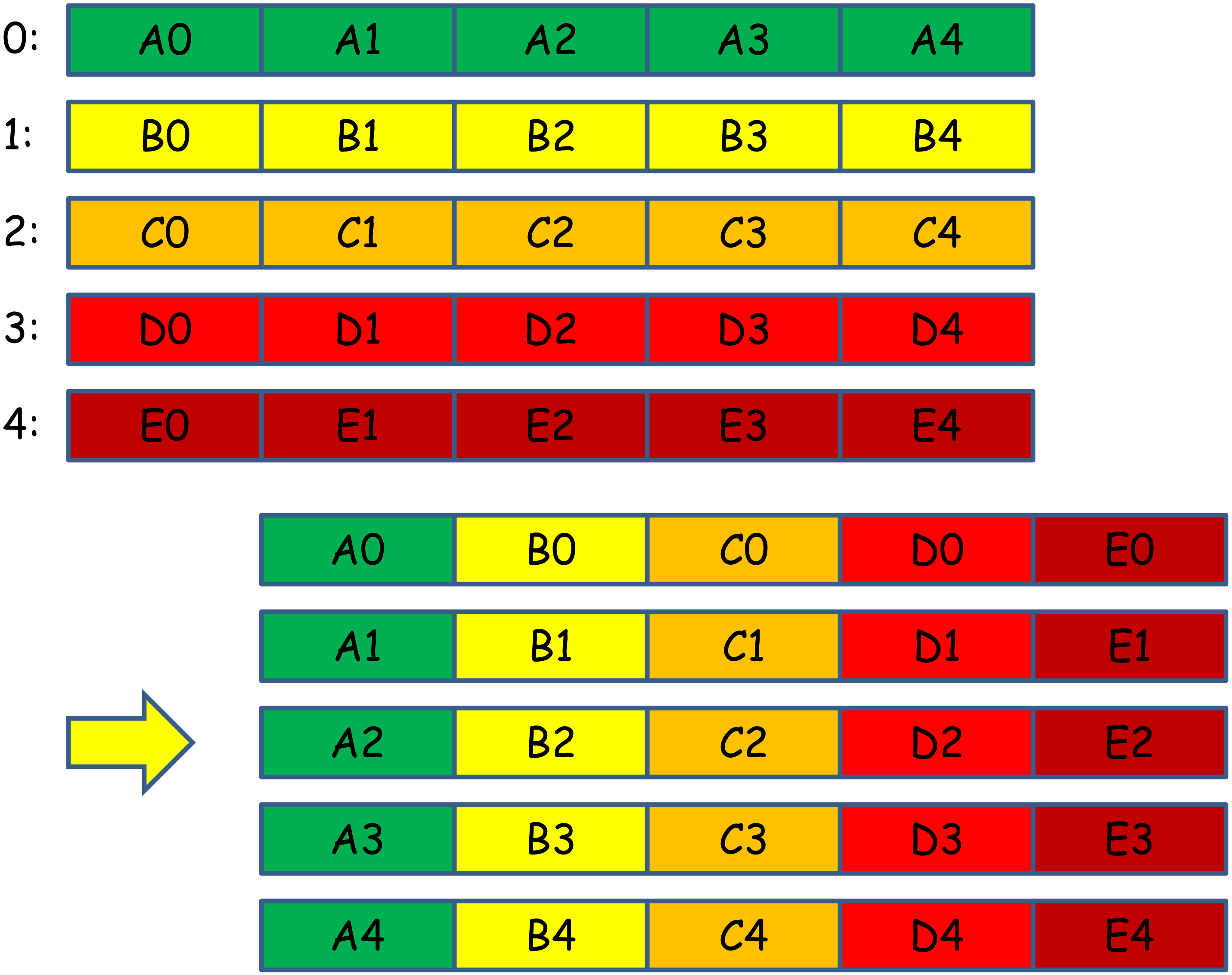
•All processes have an individual (unlike allgather) message to all other processes

•All messages have the same size, but different processes can use different datatypes (signatures must match)

•MPI_IN_PLACE can be used as sbuf, must be given by all processes, communication is in and out of rbuf

•Aka Transpose

•Aka all-to-all personalized communication



0:	A0	A1	A2	A3	A4
1:	B0	B1	B2	B3	B4
2:	C0	C1	C2	C3	C4
3:	D0	D1	D2	D3	D4
4:	E0	E1	E2	E3	E4

A0	B0	C0	D0	E0
A1	B1	C1	D1	E1
A2	B2	C2	D2	E2
A3	B3	C3	D3	E3
A4	B4	C4	D4	E4

```
MPI_Alltoall(void* sbuf, int scount, MPI_Datatype stype,
             void *rbuf, int rcount, MPI_Datatype rtype,
             MPI_Comm comm)
```

- Block to process i is stored at sbuf+i*scount*extent(stype)

- Block from process i is stored at rbuf+i*rcount*extent(rtype)

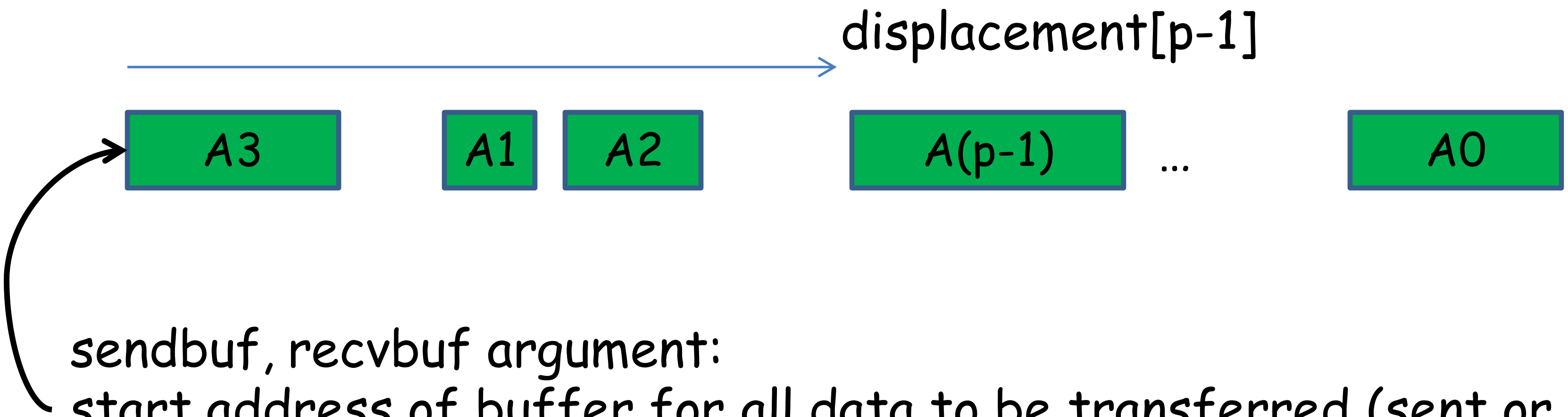
Irregular (vector, v-) collectives

Regular collectives strictly require that all processes use the same message sizes. Sometimes more flexibility is needed

- MPI_Gatherv(), MPI_Scatterv()
- MPI_Allgatherv()
- MPI_Alltoallv(), MPI_Alltoallw()

Data sizes and signatures must match pairwise, amount destined to a process must match what is required by that process

Processes can use different datatypes, data need not have the same structure, but signature must match

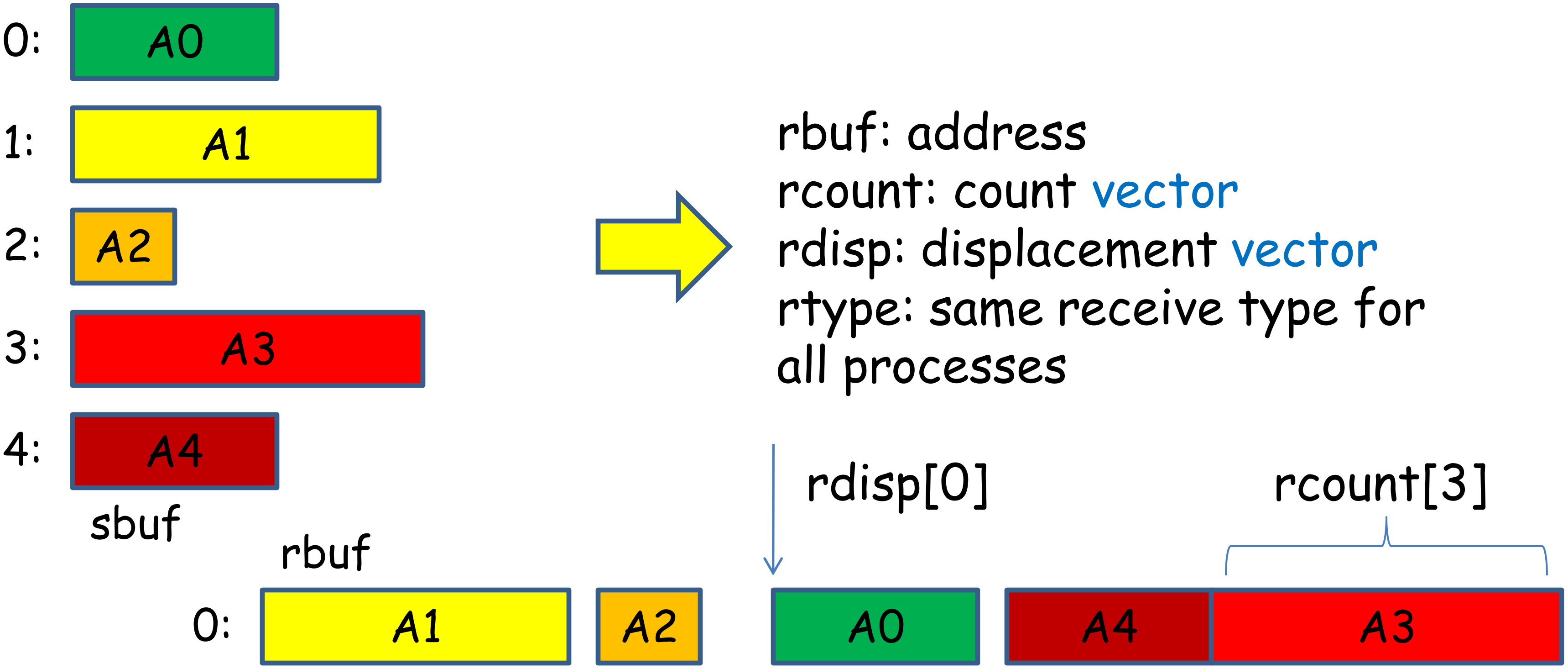


sendbuf, recvbuf argument:
start address of buffer for all data to be transferred (sent or received)

Segments to be transferred to/from different ranks may have different size (count[i]), and different displacement (displ[i]) relative to start address.

Displacement is in datatype units

```
MPI_Gatherv(void* sbuf, int scount, MPI_Datatype stype,
            void *rbuf, int rcount[], int rdisp[],
            MPI_Datatype rtype,
            int root, MPI_Comm comm)
```



rbuf: address
rcount: count vector
rdisp: displacement vector
rtype: same receive type for all processes

```
MPI_Gatherv(void* sbuf, int scount, MPI_Datatype stype,
            void *rbuf, int rcount[], int rdisp[],
            MPI_Datatype rtype,
            int root, MPI_Comm comm)
```

- Received data must not overlap
- Displacement and count vectors significant only at root
- Size/signature match pairwise

Example: root gathers unknown amount of data from all processes

```
if (rank==root) {
  int rcounts[size], rdisp[size];
  MPI_Gatherv(sbuf,…rbuf,rcounts,rdisp,…,comm);
} else {
  MPI_Gatherv(sbuf,scount,…,comm);
}
```

Array of receive counts for all processes

Send count for process i, must match rcounts[i] at root

MPI_Gatherv() requires that rcount[i] equals scount of process i (if stype and rtype are same). Root must know all scounts

Example: root gathers unknown amount of data from all processes

```
if (rank==root) {
  int rcounts[size], rdisp[size];
  MPI_Gather(scount,1,MPI_INT,rcounts,1,MPI_INT,comm);
  // compute displacements
  rdispls[0] = 0;
  for (i=1; i<size; i++)
    rdisp[i] = rdisp[i-1]+rcounts[i-1];
  MPI_Gatherv(sbuf,…rbuf,rcounts,rdisp,…,comm);
} else {
  MPI_Gather(scount,1,MPI_INT,rcounts,1,MPI_INT,comm);
  MPI_Gatherv(sbuf,scount,…,comm);
}
```

Regular MPI_Gather used to gather rcount vector, each process transmits its scount to root

```
MPI_Gatherv(void* sbuf, int scount, MPI_Datatype stype,
            void *rbuf, int rcount[], int rdisp[],
            MPI_Datatype rtype,
            int root, MPI_Comm comm)
```

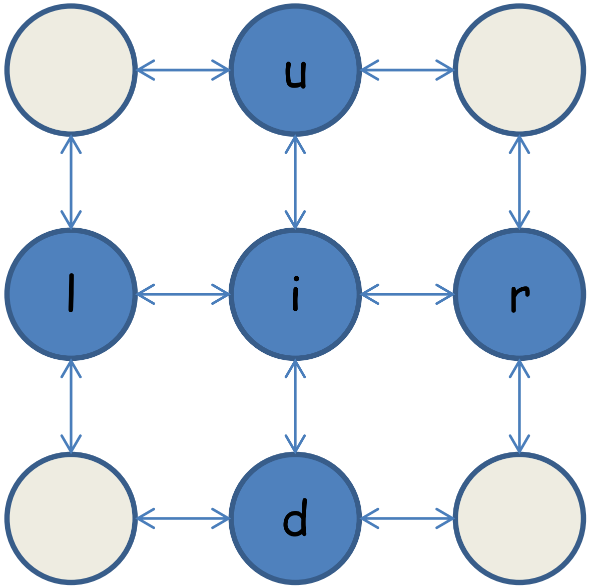
```
MPI_Scatterv(void* sbuf, int scount[], int sdisp[],
             MPI_Datatype stype[],
             void *rbuf, int rcount, MPI_Datatype rtype,
             int root, MPI_Comm comm)
```

```
MPI_Allgatherv(void* sbuf,
               int scount, MPI_Datatype stype,
               void *rbuf, int rcount[], int rdisp[],
               MPI_Datatype rtype,
               MPI_Comm comm)
```

```
MPI_Alltoallv(void* sbuf, int scount[], int sdisp[],
              MPI_Datatype stype,
              void *rbuf, int rcount[], int rdisp[],
              MPI_Datatype rtype,
              MPI_Comm comm)
```

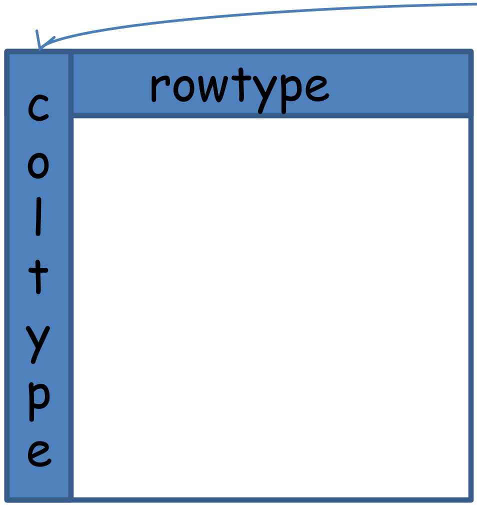
```
MPI_Alltoallw(void* sbuf, int scount[], int sdisp[],
              MPI_Datatype stype[],
              void *rbuf, int rcount[], int rdisp[],
              MPI_Datatype rtype[],
              MPI_Comm comm)
```

In 2d stencil all processes perform exchange with four other processes. An irregular all-to-all communication pattern



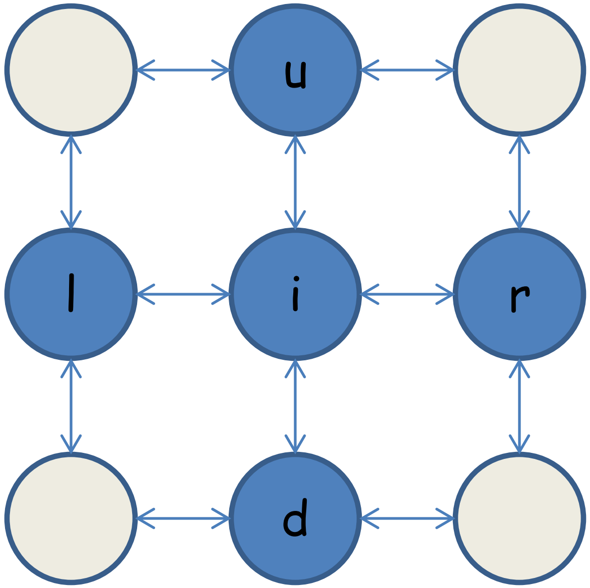
```
for (i=0; i<size; i++) {
  scount[i] = 0; rcount[i] = 0;
  rdispl[i] = 0; sdispl[i] = 0;
  stype[i] =
}
scount[l] = 1; sdispl[l] = …;
stype[l] = coltype;
… // other 3 neighbors
MPI_Alltoallw(sbuf,scount,sdispl,
              stype,
              rbuf,rcount,rdispl,
              rtype,
              cartcomm);
// update
```

In 2d stencil all processes perform exchange with four other processes. Irregular all-to-all communication pattern



```
for (i=0; i<size; i++) {
  scount[i] = 0; rcount[i] = 0;
  rdispl[i] = 0; sdispl[i] = 0;
  stype[i] =
}
scount[l] = 1; sdispl[l] = …;
stype[l] = coltype;
… // other 3 neighbors
MPI_Alltoallw(sbuf,scount,sdispl,
              stype,
              rbuf,rcount,rdispl,
              rtype,
              cartcomm);
// update
```

Non-blocking communication to allow overlap with computation



```
for (i=0; i<size; i++) {
  scount[i] = 0; rcount[i] = 0;
  rdispl[i] = 0; sdispl[i] = 0;
  stype[i] =
}
scount[l] = 1; sdispl[l] = …;
stype[l] = coltype;
… // other 3 neighbors
MPI_Ialltoallw(sbuf,scount,sdispl,
               stype,
               rbuf,rcount,rdispl,
               rtype,
               cartcomm,
               &stencilreq);
// update non-border region
MPI_Wait(&stencilrequest);
// update border
```

```
MPI_Alltoallw(void* sbuf, int scount[], int sdisp[],
              MPI_Datatype stype[],
              void *rbuf, int rcount[], int rdisp[],
              MPI_Datatype rtype[],
              MPI_Comm comm)
```

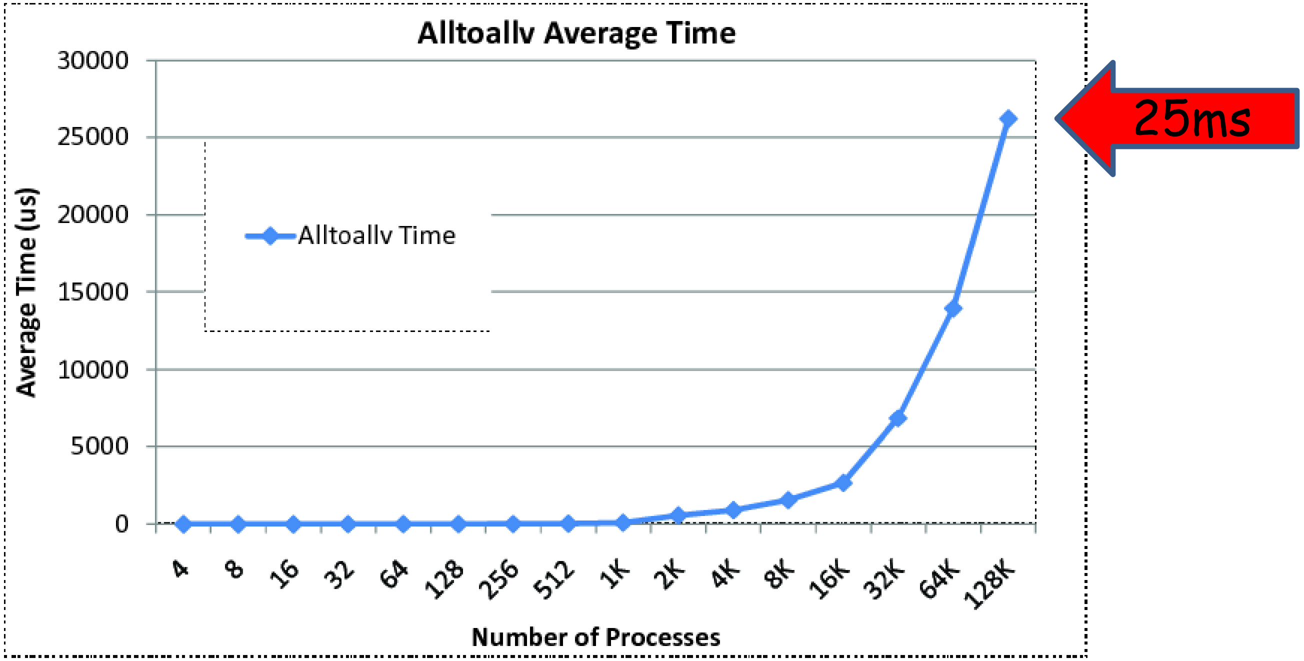
Most general (and most costly) collective communication operation:

- Separate count, displacement, and type for each other process

- Type signatures and sizes must match pairwise (not checked, user responsibility)

- Non-scalable misuse: many zero-size messages

Experiment on BlueGene/M: time for MPI_Alltoallv() call with all counts[i]= 0 for all i



[Pavan Balaji, Darius Buntinas, David Goodell, William Gropp, Torsten Hoefler, Sameer Kumar, Ewing L. Lusk, Rajeev Thakur, Jesper Larsson Träff:MPI on millions of Cores. Parallel Processing Letters 21(1): 45-60 (2011)]

Non-blocking communication collectives

```
MPI_Ibarrier(MPI_Comm comm, MPI_Request &request)
MPI_Ibcast(…,MPI_Comm comm, MPI_Request &request)
MPI_Igather(…,MPI_Comm comm, MPI_Request &request)
MPI_Igatherv(…,MPI_Comm comm, MPI_Request &request)
MPI_Iscatter(…,MPI_Comm comm, MPI_Request &request)
MPI_Iscatterv(…,MPI_Comm comm, MPI_Request &request)
MPI_Iallgather(…,MPI_Comm comm, MPI_Request &request)
MPI_Iallgatherv(…,MPI_Comm comm,
                MPI_Request &request)
MPI_Ialltoall(…,MPI_Comm comm, MPI_Request &request)
MPI_Ialltoallv(…,MPI_Comm comm, MPI_Request &request)
MPI_Ialltoallw(…,MPI_Comm comm, MPI_Request &request)
```

Reduction collectives

•Each process has vector of data (same size, same signature)

•Binary, associative operation (MPI builtin, MPI_SUM,…, or user defined)

•Collectively, processes perform element-wise reduction, result is stored at:

1. Root: MPI_Reduce()
2. All processes: MPI_Allreduce()
3. Scattered in blocks: MPI_Reduce_Scatter()

By associativity

Y = X0+X1+X2+…+X7 = (((X0+X1)+(x2+X3))+((X4+X5)+(X6+X7)))

result vector Y can be computed in parallel, bracketing/order does not matter

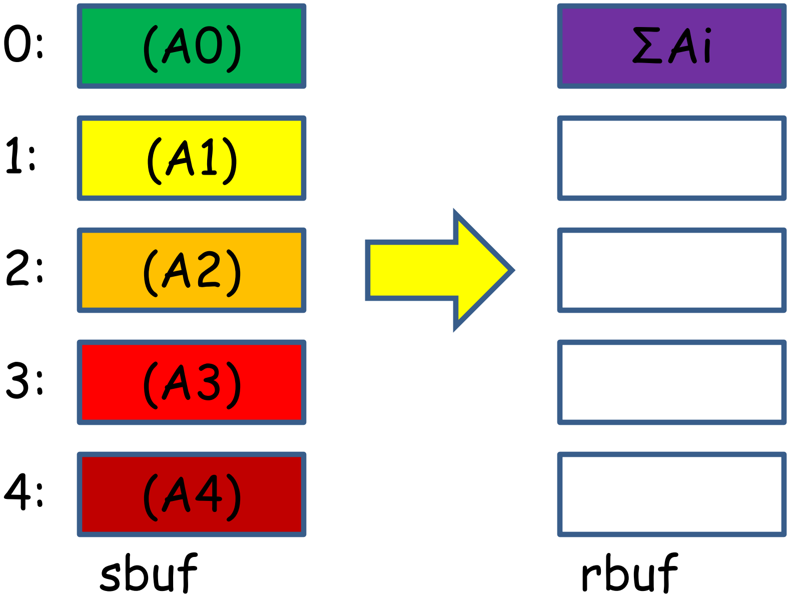
Associative in theory. In finite-precision reality, MPI_Op functions are not mathematical functions, MPI_SUM is not associative on MPI_FLOAT

If specific reduction order is required, MPI reduction collectives cannot used (unless more is known about the implementation)

Commutativity can and is sometimes be exploited

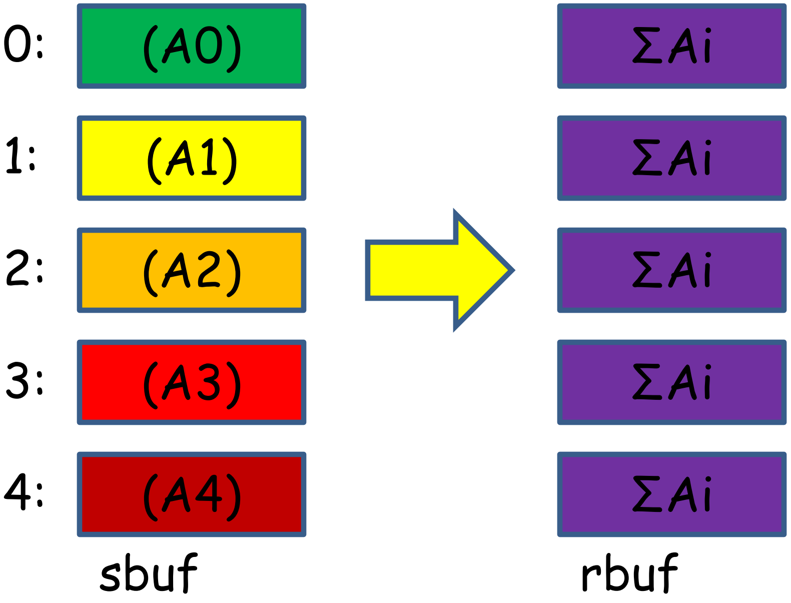
MPI_Op	function	Operand type
MPI_MAX	max	Integer, Floating
MPI_MIN	min	Integer, Floating
MPI_SUM	sum	Integer, Floating
MPI_PROD	product	Integer, Floating
MPI_LAND	logical and	Integer, Logical
MPI_BAND	bitwise and	Integer, Byte
MPI_LOR	logical or	Integer, Logical
MPI_BOR	bitwise or	Integer, Byte
MPI_LXOR	logical exclusive or	Integer, Logical
MPI_BXOR	bitwise exclusive or	Integer, Byte
MPI_MAXLOC	max value and location of max	Special pair type
MPI_MINLOC	min value and location of min	Special pair type

```
MPI_Reduce(void *sbuf,
           void *rbuf, int count, MPI_Datatype type,
           MPI_Op op, int root, MPI_Comm comm)
```

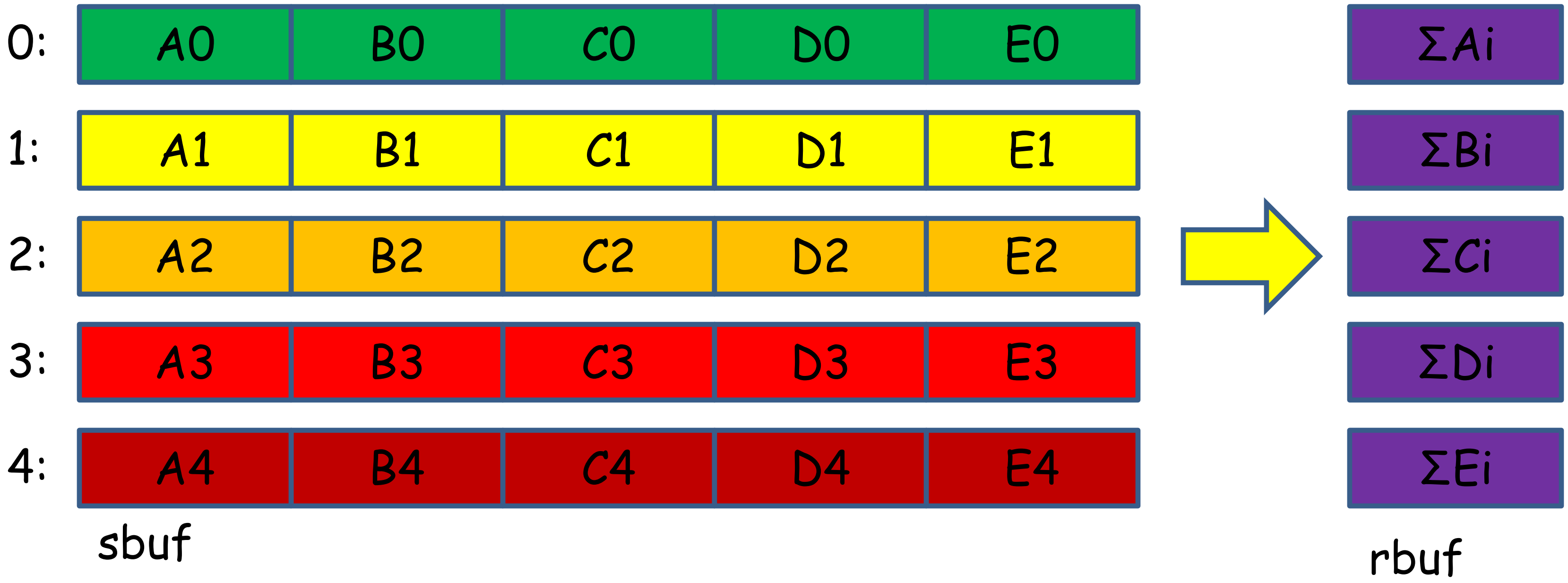


MPI_IN_PLACE at sbuf of root takes operand from rbuf

```
MPI_Allreduce(void *sbuf,
              void *rbuf, int count, MPI_Datatype type,
              MPI_Op op, MPI_Comm comm)
```



```
MPI_Reduce_scatter_block(void *sbuf,
    void *rbuf, int count, MPI_Datatype type,
    MPI_Op op, MPI_Comm comm)
```



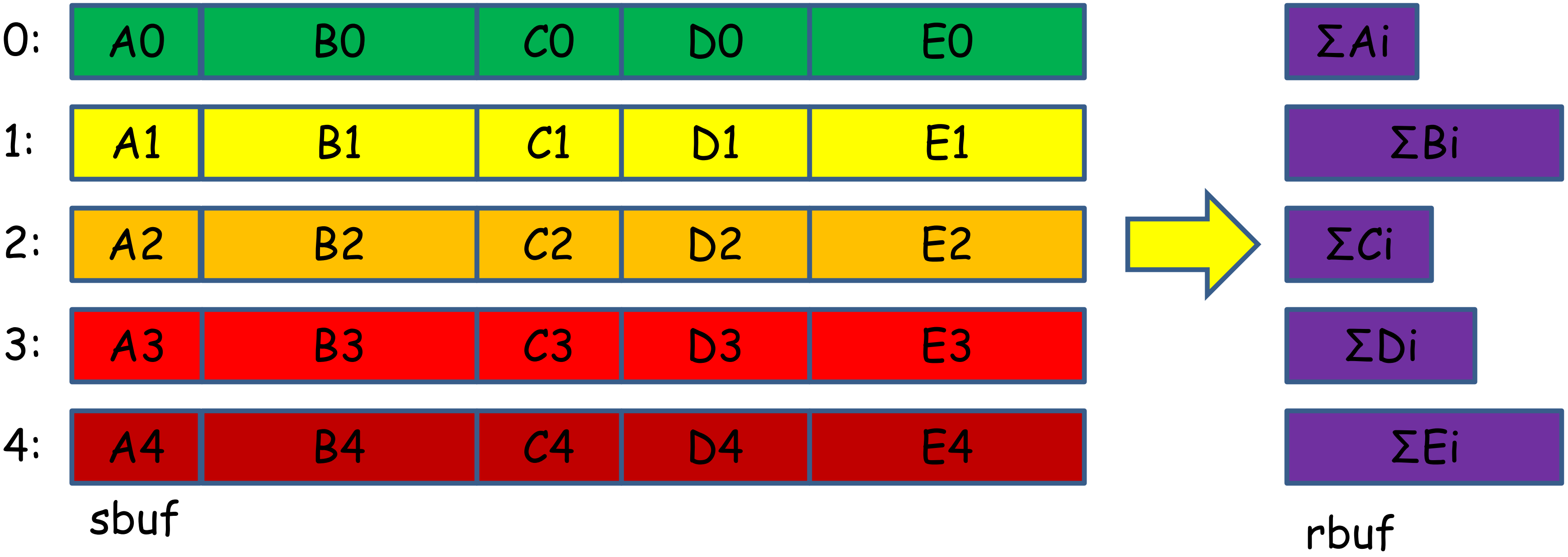
0:	A0	B0	C0	D0	E0	ΣAi
1:	A1	B1	C1	D1	E1	ΣBi
2:	A2	B2	C2	D2	E2	ΣCi
3:	A3	B3	C3	D3	E3	ΣDi
4:	A4	B4	C4	D4	E4	ΣEi

sbuf

rbuf

All result blocks have the same size

```
MPI_Reduce_scatter(void *sbuf,
    void *rbuf, int count[], MPI_Datatype type,
    MPI_Op op, MPI_Comm comm)
```



Result blocks have different size, count[] is a vector

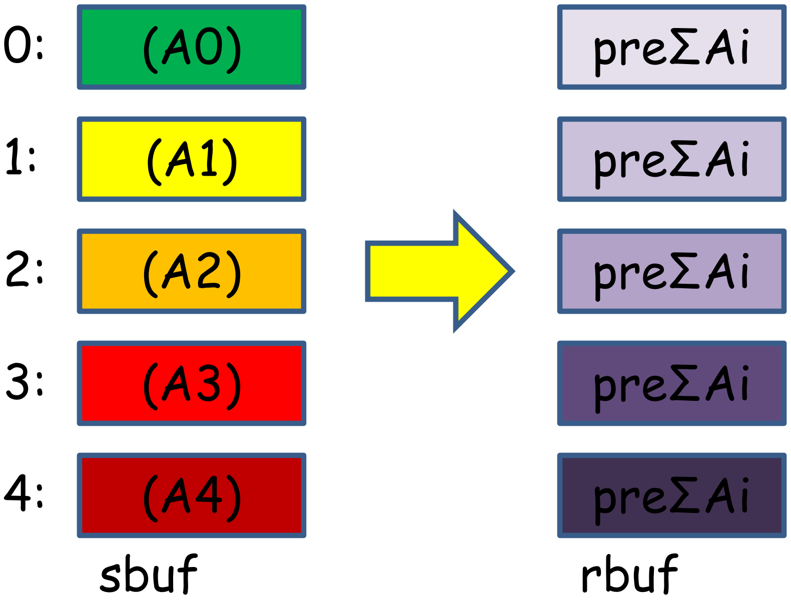
Scan collectives

Each process vector Xi (same size, same signature)
Binary, associative MPI operation

All vector prefix sums Yi = X0+X1+…+Xi are computed and stored:

1. Yi at rank i (inclusive prefix sums): MPI_Scan()
2. Y(i-1) at rank i (exclusve prefix sums): MPI_Exscan()

```
MPI_Scan(void *sbuf,
         void *rbuf, int count, MPI_Datatype type,
         MPI_Op op, MPI_Comm comm)
```



0: (A0) preΣAi

1: (A1) preΣAi

2: (A2) preΣAi

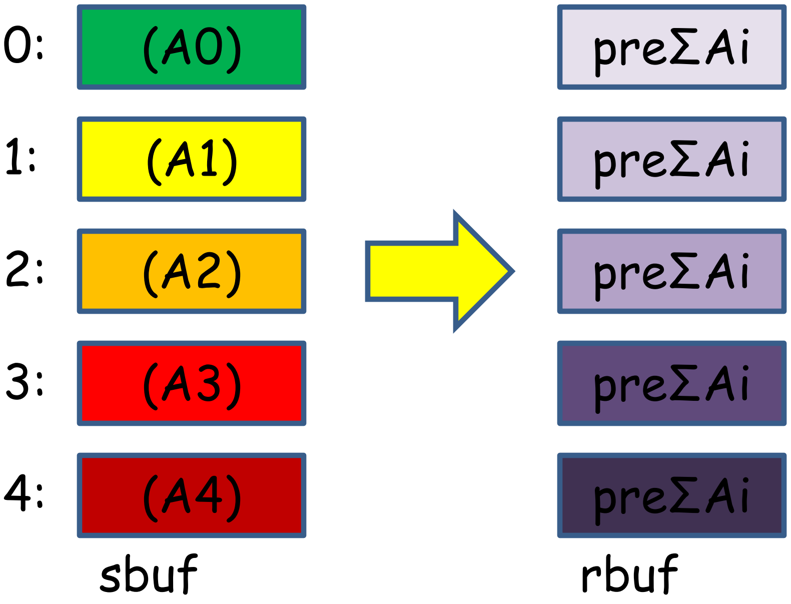
3: (A3) preΣAi

4: (A4) preΣAi

sbuf rbuf

Inclusive vector prefix-sums

```
MPI_Exscan(void *sbuf,
           void *rbuf, int count, MPI_Datatype type,
           MPI_Op op, MPI_Comm comm)
```



Exclusive vector prefix-sums

```
MPI_Reduce_local(void *sbuf, void *rbuf, int count,
                 MPI_Datatype type, MPI_Op op)
```

Important for libraries that use MPI_op reductions

```
MPI_Op_create(MPI_User_function *user_fn,
              int commute, MPI_Op *op)
```

```
MPI_Op_free(MPI_Op *op)
```

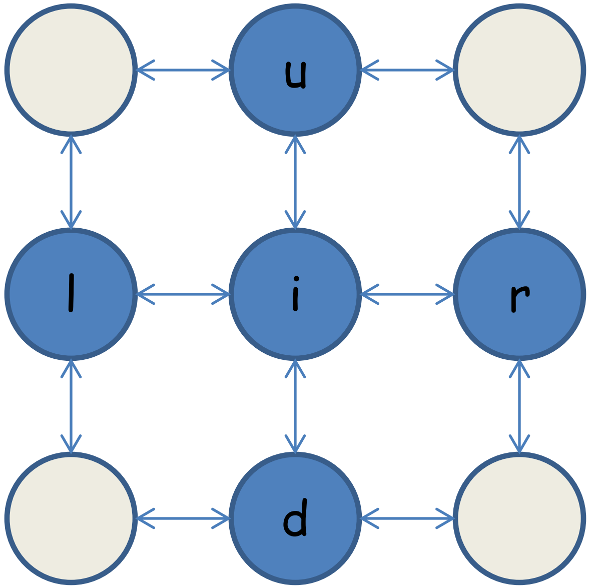
Possible to define/register own, “user-defined“, binary, associative operators that can even work on derived datatypes

Allow powerful uses of general reductions, map-reduce…

Non-blocking reduction collectives

```
MPI_Ireduce(…,MPI_Comm comm, MPI_Request &request)
MPI_Iallreduce(…,MPI_Comm comm, MPI_Request &request)
MPI_Ireduce_scatter_block(…,MPI_Comm comm,
                          MPI_Request &request)
MPI_Ireduce_scatter(…,MPI_Comm comm,
                    MPI_Request &request)
MPI_Iscan(…,MPI_Comm comm, MPI_Request &request)
MPI_Iexscan(…,MPI_Comm comm, MPI_Request &request)
```

Solution 3: One-sided communication



Iterate:

```
MPI_Put(…,up,…,window);
MPI_Put(…,down,…,window);
MPI_Put(…,left,…,window);
MPI_Put(…,right,…,window);
```

or:

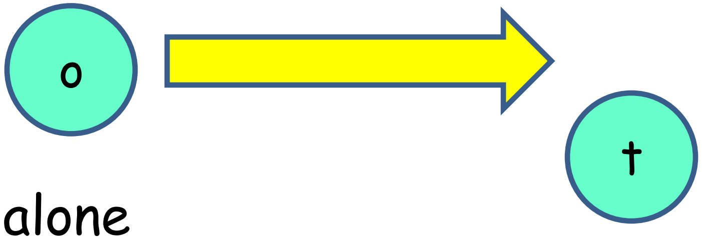
```
MPI_Get(…,up,…,window);
MPI_Get(…,down,…,window);
MPI_Get(…,left,…,window);
MPI_Get(…,right,…,window);
```

Process alone gets or puts its data from the four neighboring processes

MPI one-sided communication

Communication between two processes, only one is explicitly involved (aka Remote Memory Access, RMA)

- Which communication operations?
- Where are the data?
- How are operations completed?



Origin process alone responsible for initiating communication, provides all arguments

Target process (semantically) not involved in communication

MPI one-sided communication calls

- Put: Origin initiates transfer to target
- Get: Origin initiates transfer from target
- Accumulate: Origin initiates transfer to target, at target data are combined with data already at target
- Get&accumulate: Accumulate, but old data at target are transferred to origin

- Some atomic operations (fetch-and-ops, compare-and-swap)

Origin supplies all necessary parameters, describing data at origin (buffer, count, datatype), at target (count, datatype), and the accumulate operation to be performed (MPI_Op)

```
MPI_Put(void *obuf, int ocount, MPI_Datatype otype,
        int target,
        MPI_Aint tdisp, int tcount, MPI_Datatype ttype,
        MPI_Win window)
```

```
MPI_Get(void *obuf, int ocount, MPI_Datatype otype,
        int target,
        MPI_Aint tdisp, int tcount, MPI_Datatype ttype,
        MPI_Win window)
```

```
MPI_Accumulate(void *obuf,
               int ocount, MPI_Datatype otype,
               int target, MPI_Aint tdisp,
               int tcount, MPI_Datatype ttype,
               MPI_Op op, MPI_Win window)
```

```
MPI_Get_accumulate(void *obuf,
                   int ocount, MPI_Datatype otype,
                   void *resultbuf,
                   int rcound, MPI_Datatype rtype,
                   int target, MPI_Aint tdisp,
                   int tcount, MPI_Datatype ttype,
                   MPI_Op op, MPI_Win window)
```

- Communication calls are non-blocking, calls return Immediately

- Communication calls have local completion semantics: no guarantee that data have arrived before synchronization operation, or that any action had happened at target

```
•MPI_Put(obuf,ocount,otype,…,win)
•MPI_Get(obuf,ocount,otype,…,win)
•MPI_Accumulate(obuf,ocount,otype,…,op,win);

•MPI_Get_accumulate(obuf,ocount,otype,
                    rbuf,rcount,rtype,…,
                    op,win);
```

First sequence of arguments describe local buffer(s) at origin:

- buffer address
- element count
- datatype structure of elements

MPI processes do not have access to memory of other processes, buffer addresses on one processes have no meaning at other processes

Instead, MPI processes expose allocated memory to each other in an MPI window object MPI_Win

Addressing at target done by a relative displacement, actual address computed at target by offsetting relative to the base of the targets window memory

A displacement unit at each target is used for interpreting displacements

Process allocate memory locally, supply an address and size when window is created

```
MPI_Win_create(void *base,
               MPI_Aint size, int disp_unit,
               MPI_Info info, MPI_Comm comm,
               MPI_Win &window)
```

Window creation is always collective

An MPI_Info object (key,value pairs) gives hints and assertions that can be used at window creation to improve future use.

Some predefined MPI_Info for window creation
•no_locks: locks will not be used at this window
•same_size: all supplied memory has the same size
•same_disp_unit: …

MPI_INFO_NULL always legal info, no hints provided

```
MPI_Win_create(void *base,
               MPI_Aint size, int disp_unit,
               MPI_Info info, MPI_Comm comm,
               MPI_Win &window)
```

Memory can be allocated using memory allocators malloc(), calloc(), …, but no reallocation

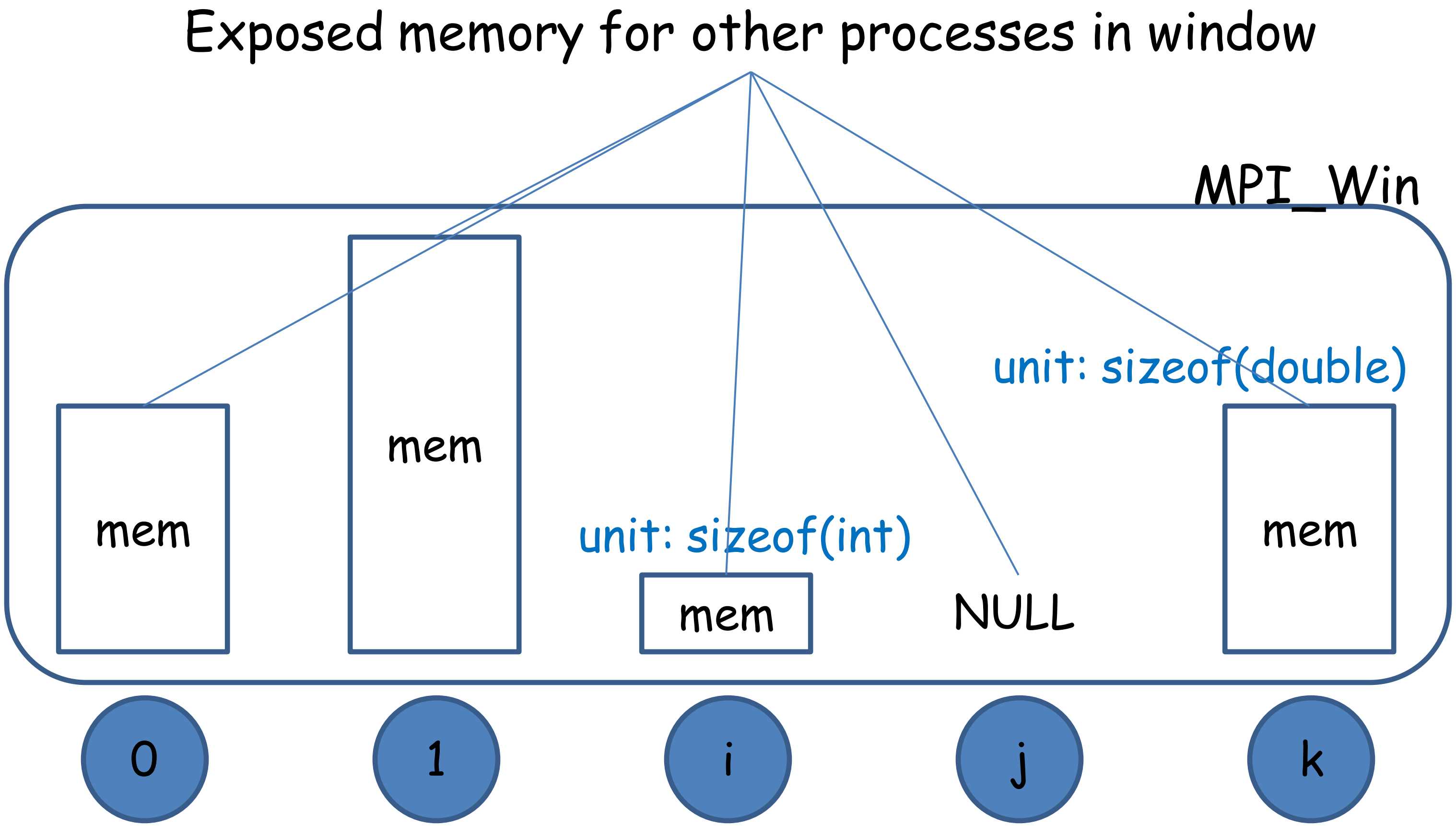
MPI provides a memory allocator, that can be used with advantage (good memory)

- MPI_Alloc_mem()
- MPI_Free_mem()

```
MPI_Win_free(MPI_Win &window)
```

Good practice to free window object after use; freeing is collective over the processes in window

Memory in window is not freed


Exposed memory for other processes in window
MPI_Win
unit: sizeof(double)
mem
mem
unit: sizeof(int)
mem
NULL
mem
0
1
i
j
k

```
MPI_Win_allocate(MPI_Aint size, int disp_unit,
                 MPI_Info info, MPI_Comm comm,
                 void &baseptr, MPI_Win &window)
```

Allocates (good) memory and creates window in one, collective operation

MPI_Win_free() on this type of window also frees memory

Alleviates need for using special MPI memory allocator

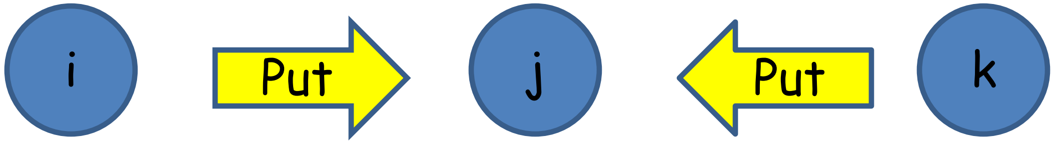
Last sequence of arguments describe remote buffer at target

- displacement
- count
- datatype

```
•MPI_Put(…,target,tdisp,tcount,ttype,win);
•MPI_Get(…,target,tdisp,tcount,ttype,win);
•MPI_Accumulate(…,target,tdisp,tcount,ttype,op,win);

•MPI_Get_accumulate(…,target,tdisp,tcount,ttype,
                    op,win);
```

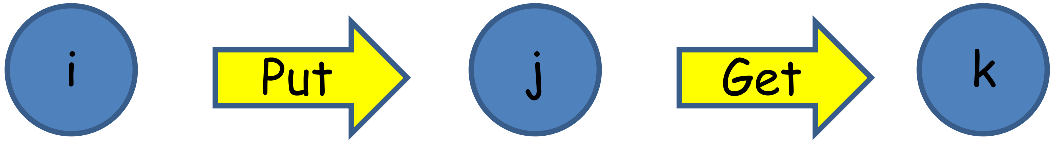
Address at target is always base+targetdispunit*targetdisp

Origin data must fit into target buffer, type signatures must match, i.e. length of origin data at most length of target data, same sequence of basic types



- Concurrent gets/puts must access disjoint target addresses
- Data races are disallowed
- Very strict rules, violation means undefined result (not illegal, so no need to check)

MPI_Accumulate: atomic (at level of basic datatype) update at target, concurrent accumulates to same target address allowed



- Concurrent gets/puts must access disjoint target addresses

- Data races are disallowed

- Very strict rules, violation means undefined result (not illegal, so no need to check)

MPI_Accumulate: atomic (at level of basic datatype) update at target, concurrent accumulates to same target address allowed

Communication epoch model

One-sided communication happens in epochs

- Origin opens an access epoch in which it needs access to one or more targets
- Target opens an exposure epoch in which one or more or origins are allow to access its window memory

Communication completes at end of epochs

At origin at end of access epoch
At target at of exposure epoch

Data become visible at end of epoch

One-sided synchronization

One-sided model decouples communication and completion/synchronization

Makes it possible to amortize synchronization overheads over many communication operations

- Active synchronization: Both origin and target involved
- Passive synchronization: Only origin involved

```
MPI_Win_fence(int assert, MPI_Win window)
```

Active synchronization, collective over all processes in window. All processes accessible and exposed

Closes preceding access and exposure epoch, opens new access and exposure epoch

Possible to assert

- MPI_NO_PRECEDE: no preceding one-sided communication operations to complete
- MPI_NO_SUCCEED: there will be no ensuing one-sided communication operations
- …

Assertions are conditions that must hold, allow optimization in MPI_Win_fence()

MPI is designed for high performance, asserted properties are not checked. It is user responsibility to write correct programs

Iterate:

```
MPI_Win_fence(0,window);

MPI_Put(…,l,…,window);
MPI_Put(…,r,…,window);
MPI_Put(…,u,…,window);
MPI_Put(…,d,…,window);

MPI_Win_fence(0,window);
```

Possibly better, Iterate:

```
MPI_Win_fence(MPI_NO_PRECEDE,window);

MPI_Put(…,l,…,window);
MPI_Put(…,r,…,window);
MPI_Put(…,u,…,window);
MPI_Put(…,d,…,window);

MPI_Win_fence(MPI_NO_SUCCEED,window);
```

```
MPI_Win_start(MPI_Group group, int assert,
              MPI_Win window)
```

```
MPI_Win_complete(MPI_Win window)
```

Active synchronization, opens access to (sparse) group of targets

```
MPI_Win_post(MPI_Group group, int assert,
             MPI_Win window)
```

```
MPI_Win_wait(MPI_Win window)
```

Active synchronization, gives exposure to (sparse) group or origins

Access and exposure group (same) setup:

```
MPI_Group group, stencilgroup;

MPI_Win_get_group(win,&group);
int stencil[4] = {u,d,l,r};
MPI_Group_incl(group,4,stencil,&stencilgroup);
MPI_Group_free(&group);
```

Iterate:

```
MPI_Win_start(stencilgroup,0,window);
MPI_Win_post(stencilgroup,0,window);

MPI_Put(…,l,…,window);
MPI_Put(…,r,…,window);
MPI_Put(…,u,…,window);
MPI_Put(…,d,…,window);

MPI_Win_complete(window);
MPI_Win_wait(window);
```

```
MPI_Win_lock(int lock_type, int target, int assert,
             MPI_Win window)
```

- Passive synchronization, opens access epoch at origin and exposure epoch at target

- Locks can be MPI_LOCK_SHARED or MPI_LOCK_EXCLUSIVE

- More than one target can be locked

Bad misnomer, one-sided locks do not provide mutual exclusion in the usual sense

```
MPI_Win_unlock(int target, MPI_Win window)
```

- Closes exposure and access epoch
- Care needed, complex memory model, data put to target may not be immediately visible at target after unlock
- A process may need to lock itself in order to properly access data put to it by another process with passive synchronization

Iterate:

```
MPI_Win_lock(MPI_SHARED,l,0,window);
MPI_Win_lock(MPI_SHARED,r,0,window);
MPI_Win_lock(MPI_SHARED,u,0,window);
MPI_Win_lock(MPI_SHARED,d,0,window);

MPI_Get(…,l,…,window);
MPI_Get(…,r,…,window);
MPI_Get(…,d,…,window);
MPI_Get(…,u,…,window);

MPI_Win_unlock(l,window);
MPI_Win_unlock(r,window);
MPI_Win_unlock(u,window);
MPI_Win_unlock(d,window);
```

Example: black-board exchange

Writing process

```
int blackboard = … // some process
MPI_Win_lock(MPI_EXCLUSIVE,
                 blackboard,0,window);
MPI_Put(…,blackboard,…,window);
MPI_Win_unlock(blackboard,window);
```

Need to be separated by other form of synchronization: barrier, point-to-point

Reading process

```
int blackboard = … // some process
MPI_Win_lock(MPI_EXCLUSIVE,
                 blackboard,0,window);
MPI_Get(…,blackboard,…,window);
MPI_Win_unlock(blackboard,window);
```

```
MPI_Win_lock_all(int assert, MPI_Win window)
```

```
MPI_Win_unlock_all(MPI_Win window)
```

- Passive synchronization, opens access epoch at origin and exposure epoch at all targets

- All-locks are shared

MPI one-sided memory model

Complex, in order to guarantee efficiency and implementability on different systems, also without cache-coherence

Memory model based on concept of private and public window copies

[Torsten Hoefler, James Dinan, Rajeev Thakur, Brian Barrett, Pavan Balaji, William Gropp, Keith D. Underwood: Remote Memory Access Programming in MPI-3. TOPC 2(2): 9 (2015)]
[Torsten Hoefler, James Dinan, Darius Buntinas, Pavan Balaji, Brian Barrett, Ron Brightwell, William Gropp, Vivek Kale, Rajeev Thakur: MPI + MPI: a new hybrid approach to parallel programming with MPI plus shared memory. Computing 95(12): 1121-1136 (2013)]

It is possible to control completion inside epochs opened with passive synchronization

```
MPI_Win_flush(int target, MPI_Win window)
```

Completes all outstanding one-sided communication operations at both origin and target rank

```
MPI_Win_flush_local(int target, MPI_Win window)
```

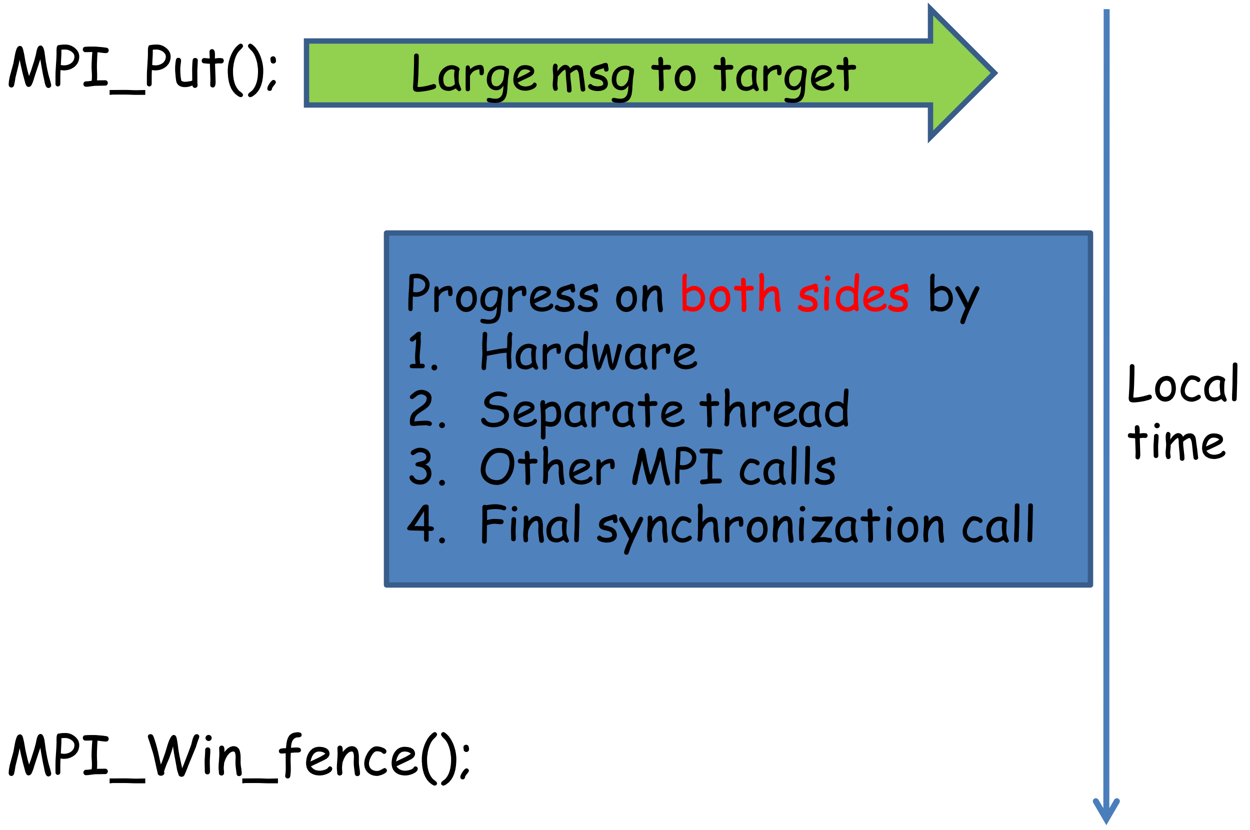
Completes all outstanding one-sided communication operations locally at origin

```
MPI_Win_flush_all(MPI_Win window)
```

```
MPI_Win_flush_local_all(MPI_Win window)
```

Completion for all targets

A note on progress



MPI_Put();

Large msg to target

Progress on both sides by

1. Hardware
2. Separate thread
3. Other MPI calls
4. Final synchronization call

Local time

MPI_Win_fence();

One-sided vs. point-to-point communication?

One-sided and point-to-point are two different communication models, good for different types of applications

- Point-to-point: structured, oblivious, both processes naturally know who their partner is
- One-sided: irregular, dynamic, data-dependent, partners change, and both parties may not know their partner

Either may have a performance advantage, but in raw terms, performance should be similar, after all same hardware and software

Modern hardware (InfiniBand) support RDMA, some argue that this gives an advantage for one-sided application level model

Principle 9:

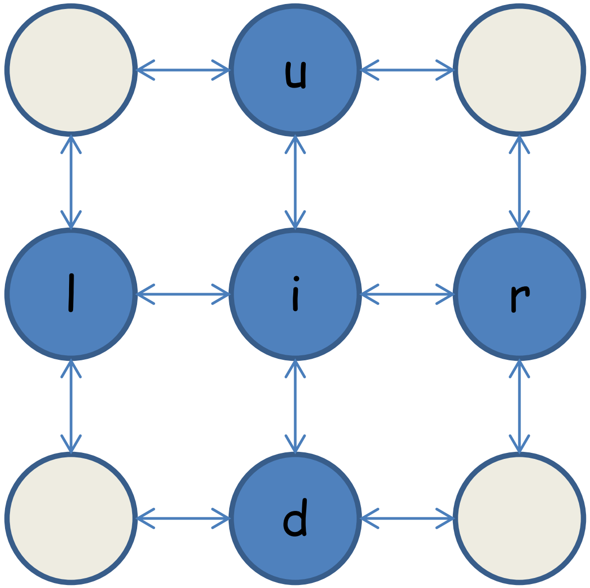
Apply assertions and info arguments carefully, only assert what is actually true, give hint only where applicable

MPI will not (and often: cannot) check that asserted conditions hold. Subtle errors may happen if conditions that do not hold are asserted

May improve performance, some assertions (on fence and lock) can improve quite a lot

Start with assertion 0, and MPI_INFO_NULL, check MPI standard and system documentation for additional, non-standardized info values

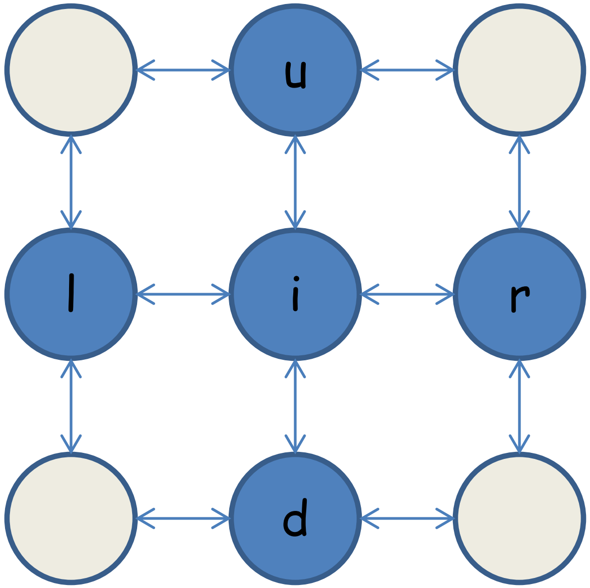
Solution 4: Neighbor (sparse) collective communication



Iterate:

```
MPI_Neighbor_alltoallw(sendbuf,
                       …,
                       recvbuf,
                       comm);
// update
```

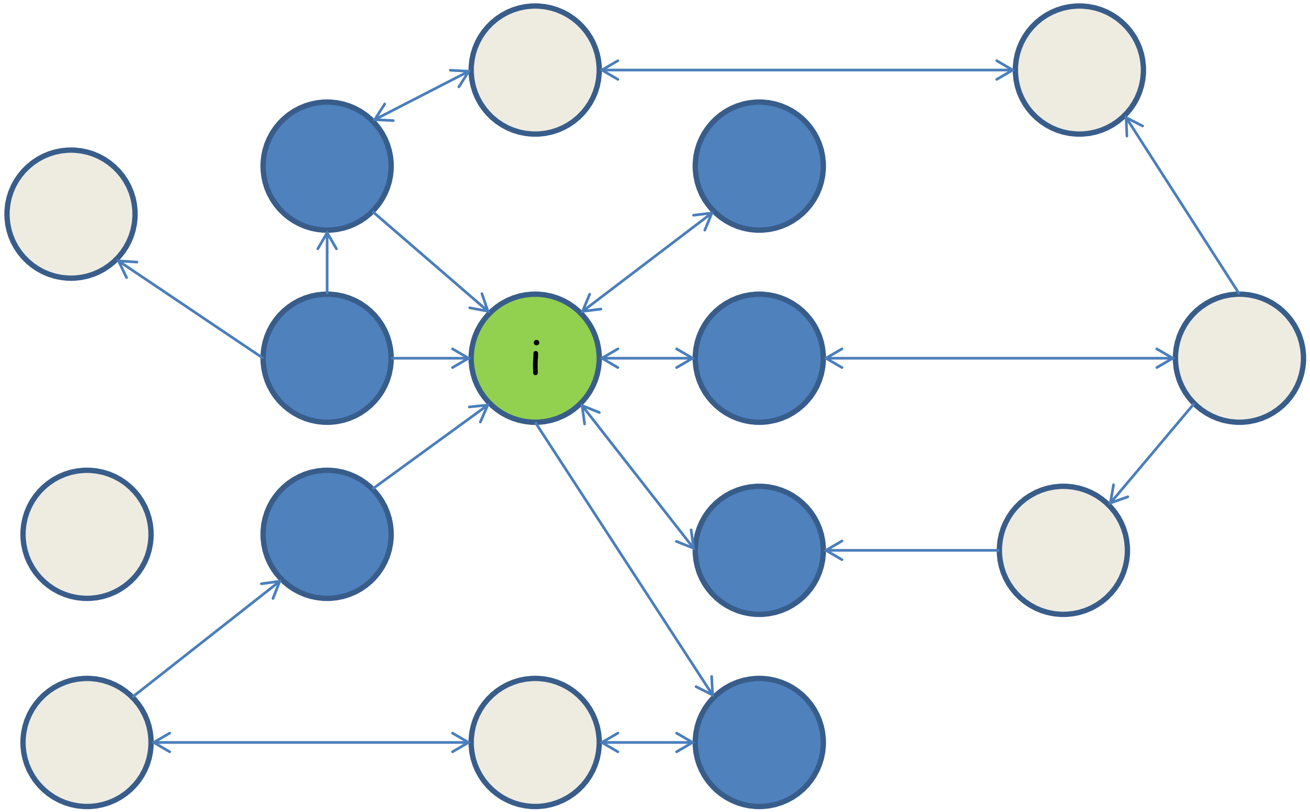
Scalable, collective communication in sparse neighborhoods



Iterate:

```
MPI_Request request;
MPI_Ineighbor_alltoallw(sendbuf,
                      …,
                      recvbuf,
                      comm,
                      &request);
// update inner region
MPI_Wait(&request);
// update border
```

An MPI neighborhood of a process is the immediate neighbors of the process in a virtual topology



An MPI neighborhood of a process is the immediate neighbors of the process in a virtual topology

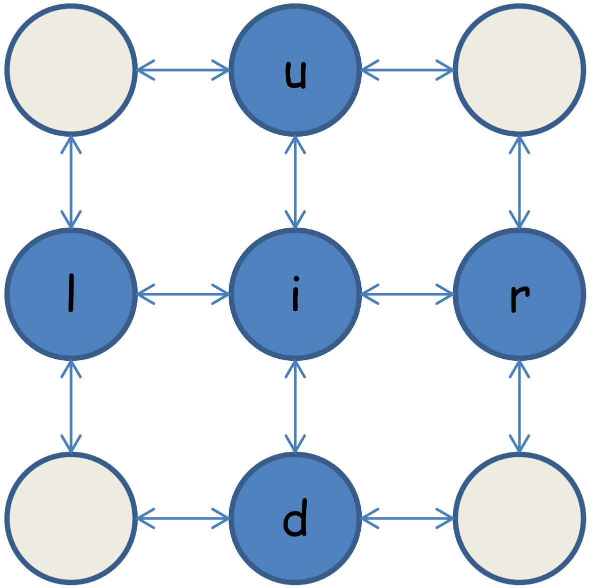
Virtual topology:

- Directed graph, edges between two processes denote (likely) communication
- Communication between any pair of processes is possible
- Edges may be weighted, weights qualify communication in some way: volume, frequency, … (not defined in MPI)
- Edges out of and into a process are ordered
- Associated with a communicator

Can be defined implicitly or explicitly

- Cartesian communicators
- Graph communicators

```
MPI_Cart_create(MPI Comm comm, int ndims,
                const int dims[], const int periods[],
                int reorder, MPI_Comm *cartcomm)
```



Implicitly creates d-dimensional mesh/torus topology

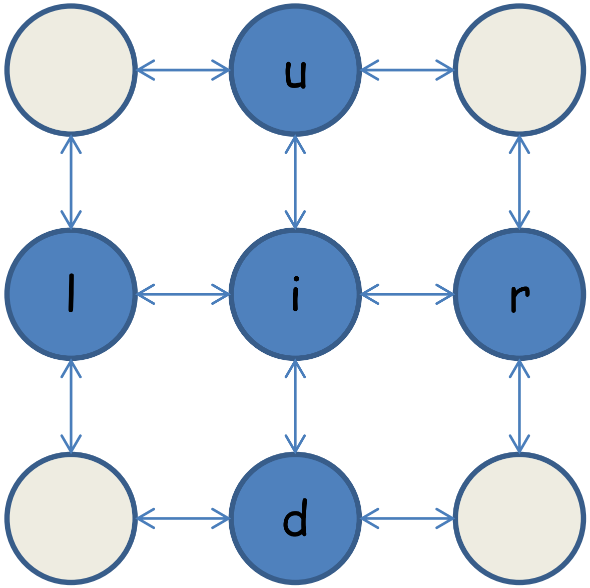
Process neighbors are torus-neighbors along dimensions

Bidirectional, unweighted

Order is dimension-wise, negative, positive neighbor

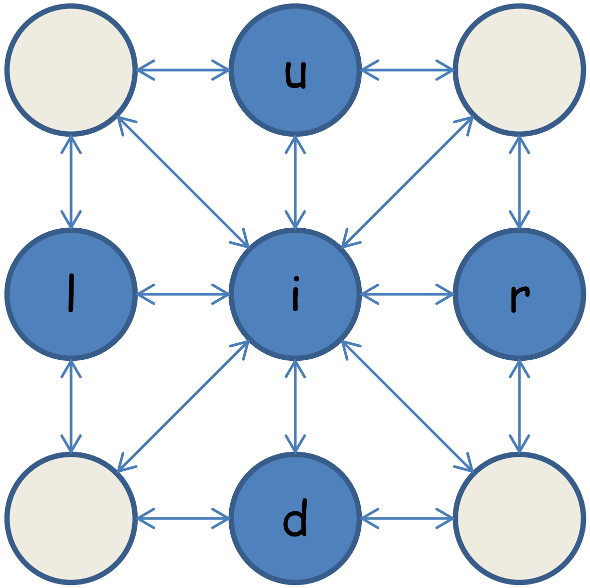
MPI_PROC_NULL processes on mesh borders are also neighbors

```
MPI_Cart_create(MPI Comm comm, int ndims,
                const int dims[], const int periods[],
                int reorder, MPI_Comm *cartcomm)
```



Sufficient for 5-point stencil

```
MPI_Cart_create(MPI Comm comm, int ndims,
                const int dims[], const int periods[],
                int reorder, MPI_Comm *cartcomm)
```



Sufficient for 5-point stencil, only

Cannot support 9-point stencil

```
MPI_Dist_graph_create(MPI_Comm comm,
                      int n, const int sources[],
                      const int degrees[],
                      const int destinations[],
                      const int weights[],
                      MPI_Info info, int reorder,
                      MPI_Comm *graphcomm)
```

Most general, distributed graph constructor

Collective over all processes in communicator, each process supplies only the edges it know

Edges can have weights, or MPI_UNWEIGHTED must be given as weights argument; if some process supplies MPI_UNWEIGHTED, all must do

No info defined (yet)

```
MPI_Dist_graph_create(MPI_Comm comm,
                      int n, const int sources[],
                      const int degrees[],
                      const int destinations[],
                      const int weights[],
                      MPI_Info info, int reorder,
                      MPI_Comm *graphcomm)
```

Edges are specified by

1. List of source vertices (process ranks in communicator)
2. List of degrees (as known by the process) for the source vertices
3. The corresponding destinations for the sources (the size of the destinations array is the sum of the given degrees)
4. List of associated weights or MPI_UNWEIGHTED

```
MPI_Dist_graph_create(MPI_Comm comm,
                      int n, const int sources[],
                      const int degrees[],
                      const int destinations[],
                      const int weights[],
                      MPI_Info info, int reorder,
                      MPI_Comm *graphcomm)
```

MPI library needs to compute for each process rank all its outgoing and all its incoming edges

The MPI library shall not (scalability) construct the full communication graph at any one process (unless of course it consists of only edges from one process), the virtual topology is kept distributed and with as little replication as possible/necessary

```
MPI_Dist_graph_neighbors_count(MPI_Comm graphcomm,
    int *indegree, int *outdegree, int *weighted)
```

```
MPI_Dist_graph_neighbors(MPI_Comm graphcomm,
    int maxindegree, int sources[], int sourceweights[],
    int maxoutdegree,
    int destinations[], int destweights[])
```

A process with a distributed graph topology (use MPI_Topo_test() call) can query for its immediate neighbors, both incoming and outgoing processes

Needed for collective operations: defines the order of the neighbors

Not possible to query for neighbors of other processes

```
MPI_Dist_graph_create_adjacent(MPI_Comm comm,
    int indegree,
    const int sources[], const int sourceweights[],
    int outdegree,
    const int destinations[], int destweights[],
    MPI_Info info, int reorder,
    MPI_Comm *graphcomm)
```

Alternative constructor if neighbors, both incoming (sources) and outgoing (destinations) are already known

```
MPI_Graph_create(MPI_Comm comm,
                 int nnodes,
                 const int index[],
                 cons[illegible]ges[],
                 in[illegible]rder,[illegible]PI_Comm *graphcomm)
```

- Each process has to supply vi[illegible]ual communication graph, represented as list of end vertices
- Each process has to know full communication graph. How can it?
- Non-scalable in space O(p(n+m))

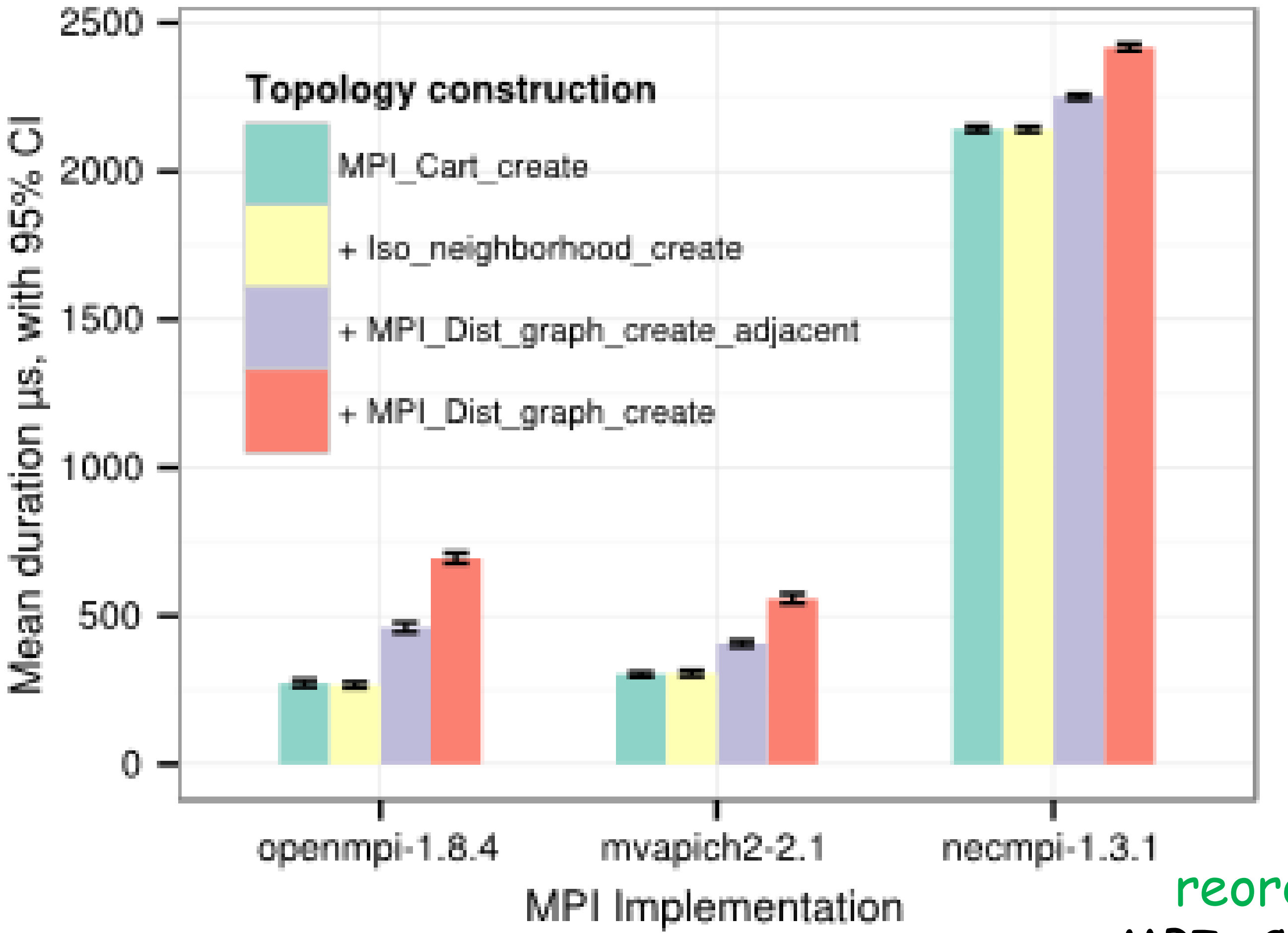
- Don't use (old MPI 1 functionality, will eventually be deprecated)

Which topology creation function is better?

Small, 36-node InfiniBand cluster, each node with two 8-core AMD 6134 Opteron processors at 2.3GHz, Mellanox IB MT4036 QDR

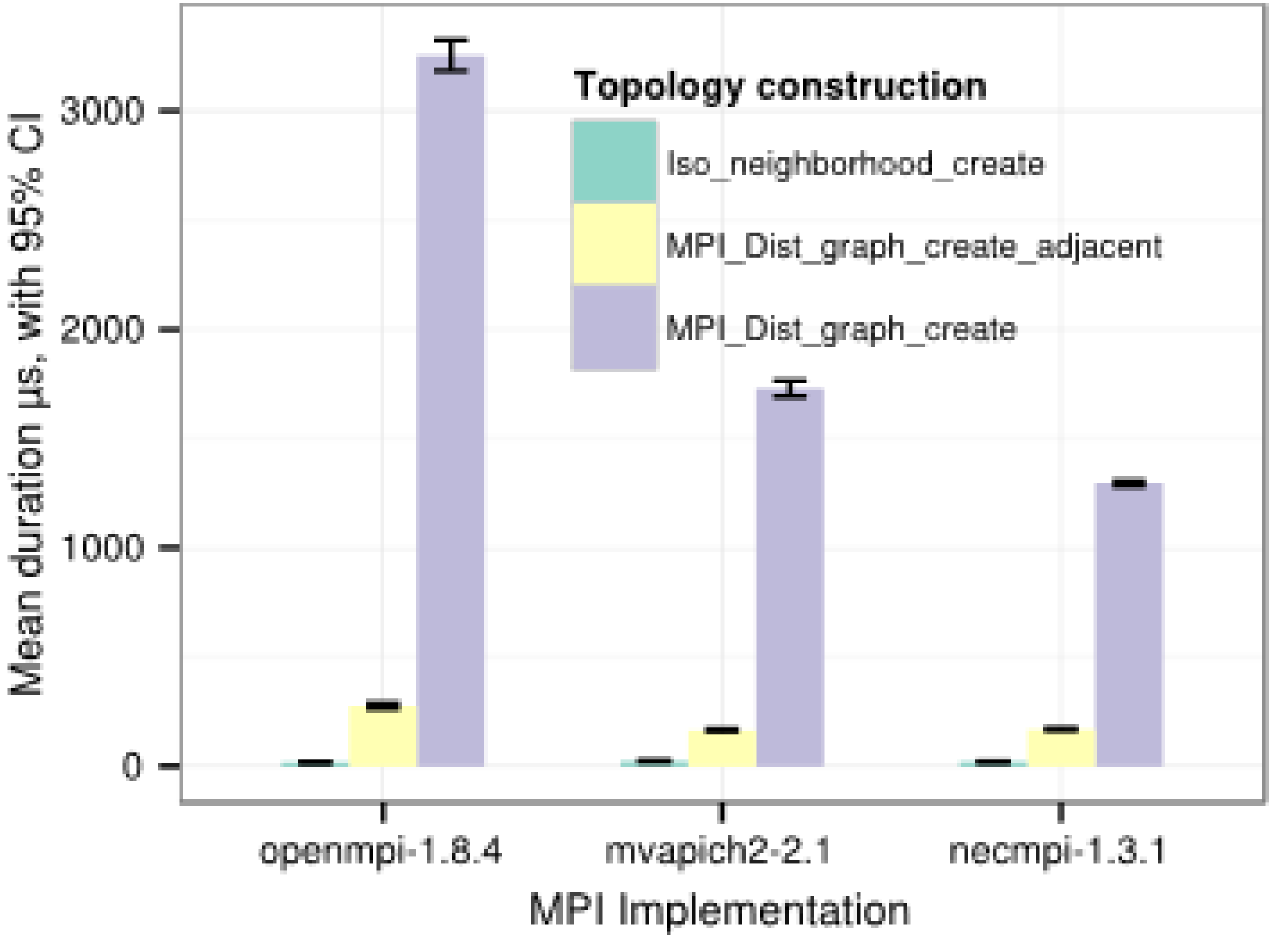
Three different MPI libraries:
necMPI (1.3.1), mvapich (2.2.1), OpenMPI (1.8.4)

d=2, 24x20 virtual torus, r=1 von Neumann neighborhood



Note: reorder=true in MPI_Cart_create

d=2, 24x20 virtual torus, r=3 Moore neighborhood



Principle 10:

Use most specific virtual topology creation function

[Jesper Larsson Träff, Felix Donatus Lübbe, Antoine Rougier, Sascha Hunold: Isomorphic, Sparse MPI-like Collective Communication Operations for Parallel Stencil Computations. EuroMPI 2015: 10:1-10:10]

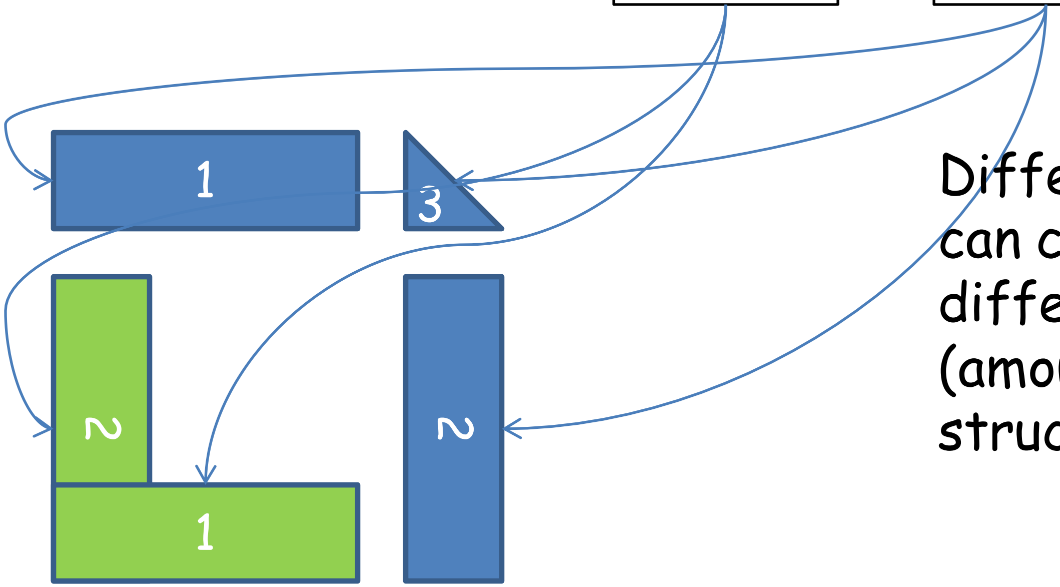
```
MPI_Neighbor_alltoallw(const void *sbuf,
                       int scount[], int sdispl[],
                       int MPI_Datatype stype[],
                       void *rbuf,
                       int rcount[], int rdispl[],
                       MPI_Datatype rtype[],
                       MPI_Comm graphcomm)
```

Most useful collective operation (over all processes in communicator) for neighborhood communication

Separate datatype argument for each neighbor, as needed for zero-copy stencil exchange

Correctness requirement as for ordinary, global collectives: signature of data sent by some source neighbor must be as specified by the count and datatype argument for that neighbor

```
MPI_Neighbor_alltoallw(sendbuf,…,recvbuf,…,comm);
```



Different neighbors can communicate different data (amount and structure)

- Local neighborhood: implicit list of source (incoming) and destination (outgoing) neighbors (MPI processes)
- Order of neighbors determine order of communication buffers
- Correctness: Destination and source message signatures must match

```
MPI_Neighbor_alltoallv(const void *sbuf,
                       int scount[], int sdispl[],
                       int MPI_Datatype stype,
                       void *rbuf,
                       int rcount[], int rdispl[],
                       MPI_Datatype rtype,
                       MPI_Comm graphcomm)
```

```
MPI_Neighbor_alltoall(const void *sbuf,
                      int scount, MPI_Datatype stype,
                      void *rbuf,
                      int rcount, MPI_Datatype rtype,
                      MPI_Comm graphcomm)
```

```
MPI_Neighbor_allgatherv(const void *sbuf,
                        int scount, MPI_Datatype stype,
                        void *rbuf,
                        int rcount[], int rdispl[],
                        MPI_Datatype rtype,
                        MPI_Comm graphcomm)
```

```
MPI_Neighbor_allgather(const void *sbuf,
                       int scount, MPI_Datatype stype,
                       void *rbuf,
                       int rcount, MPI_Datatype rtype,
                       MPI_Comm graphcomm)
```

```
MPI_Ineighbor_alltoall(…,MPI_Comm graphcomm,
                       MPI_Request *request)
MPI_Ineighbor_alltoallv(…,MPI_Comm graphcomm,
                        MPI_Request *request)
MPI_Ineighbor_alltoallw(…,MPI_Comm graphcomm,
                        MPI_Request *request)

MPI_Ineighbor_allgather(…,MPI_Comm graphcomm,
                        MPI_Request *request)
MPI_Ineighbor_allgatherv(…,MPI_Comm graphcomm,
                         MPI_Request *request)
```

With corresponding non-blocking sparse collective operations

Principle 11:

MPI objects are static, immutable, and require resources (time) to manipulate. Amortize creation time of new MPI objects over many subsequent uses

Datatypes, virtual topologies, communicators, windows are such examples. Frequent, new such objects will generate substantial overhead

Examples:

- Irregular, frequently changing neighborhoods frequently requiring new datatypes and/or virtual topologies
- Very fine-grained divide-and-conquer algorithms frequently requiring new communicators

Hybrid programming with MPI

Common practice to use MPI together with other programming framework for shared memory programming:

- OpenMP
- pthreads
- …

Sometimes better performance (sometimes not)

Are MPI routines (required to be) thread safe? Which threads can call MPI routines?

MPI defines 4 levels of thread support that an implementation may offer

1. MPI_THREAD_SINGLE: No multi-threads, only a single thread is permitted
2. MPI_THREAD_FUNNELED: Multiple threads are allowed, but only some dedicated master thread may perform MPI calls
3. MPI_THREAD_SERIALIZED: Multiple threads are allowed, any thread may perform MPI calls, but at most one at a time, no concurrent MPI operations
4. MPI_THREAD_MULTIPLE: Multiple threads allowed, and threads may perform MPI calls concurrently

Multi-threaded MPI applications must respect the provided level of thread support

Explicitly synchronize or use mutex'es to ensure that only a single thread is calling MPI functions (in case of MPI_THREAD_SERIALIZED or MPI_THREAD_FUNNELED)

Care needed for non-blocking operations and applications using MPI_Probe() – the message probed for may have been consumed by another thread…

A solution is to use MPI_Mprobe() functionality

```
MPI_Init_thread(int *argc, char **argv,
                int required, int *provided)
```

Initialize MPI library with a desired, required level of thread support; call returns what is provided (application must check), not that provided may be more than required

```
MPI_Query_thread(int *provided)
```

Query for the level of provided thread support

```
MPI_Is_thread_main(int *flag)
```

Is calling thread the “master” thread?

Recap

- MPI is a large standard, but there is good reason for most of it and most features work well together

- MPI makes many difficult things possible, often in an elegant and concise way, but it takes practice

- MPI can give very high performance on many types of systems

- Active research on interface, algorithms and implementations

- Wealth of literature

Where to learn more (advanced tutorials)

Scientific and Engineering Computation Series

Using MPI

Portable Parallel Programming with the Message-Passing Interface

third edition

William Gropp

Ewing Lusk

Anthony Skjellum

Gropp, Lusk, Skjellum: MPI 1

Scientific and Engineering Computation Series

Using Advanced MPI

Modern Features of the Message-Passing Interface

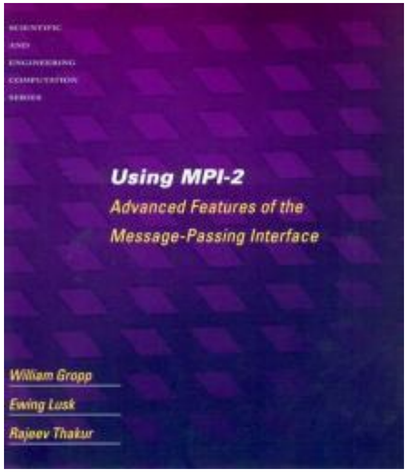
William Gropp

Torsten Hoefler

Rajeev Thakur

Ewing Lusk

Gropp, Hoefler, Thakur, Lusk: MPI 3


Scientific and Engineering Computation Series

Using MPI-2

Advanced Features of the Message-Passing Interface

William Gropp

Ewing Lusk

Rajeev Thakur

Gropp, Lusk, Thakur: MPI 2