Introduction to Parallel Computing

Distributed memory systems and programming

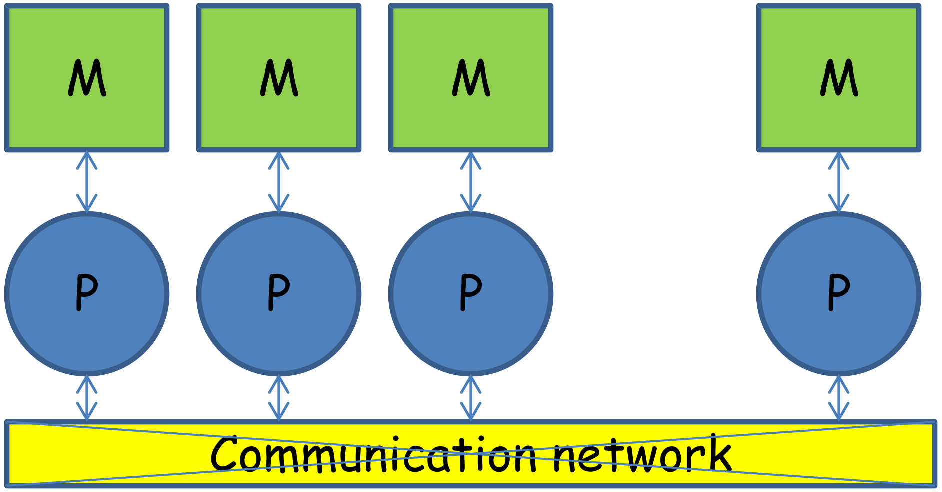
Jesper Larsson Träff

Technical University of Vienna

Parallel Computing

WS11/12

Distributed memory architectures & machines



Distributed memory architecture model

Naive distributed memory parallel programming model: independent, non-synchronized processors execute locally stored program on local data, interaction with other processors exclusively through (explicit) communication facilitated by communication network

Programming model:

- How is communication done, which communication operations?
- Synchronization and coordination
- Local vs. non-local data?
- How is locality expressed ? Explicit/implicit/hierarchical?

Cost model:

- Communication, local vs. non-local memory access

„Pure“ distributed memory system architecture:

Single processors with local memory communicate through communication network. Properties of network determines performance.

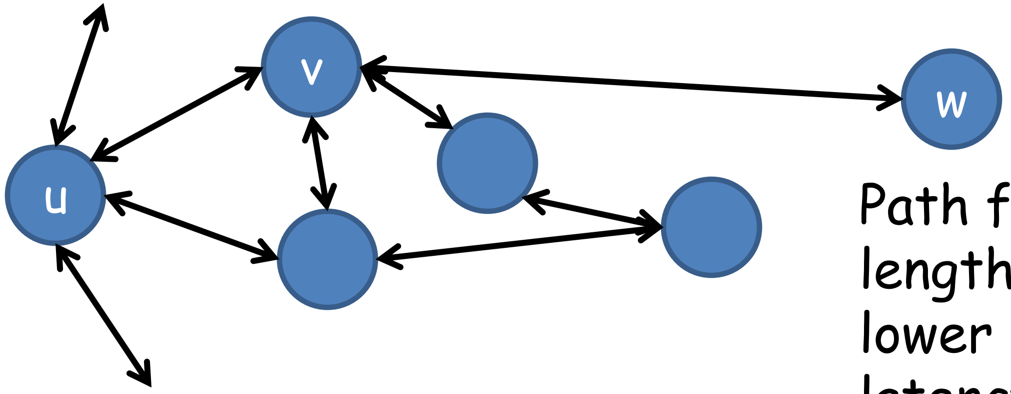
Network properties:
- Structure : topology
- Capabilities: one or several operations per network component
- Routing technique
- Switching strategy

This lecture: a little bit about topology

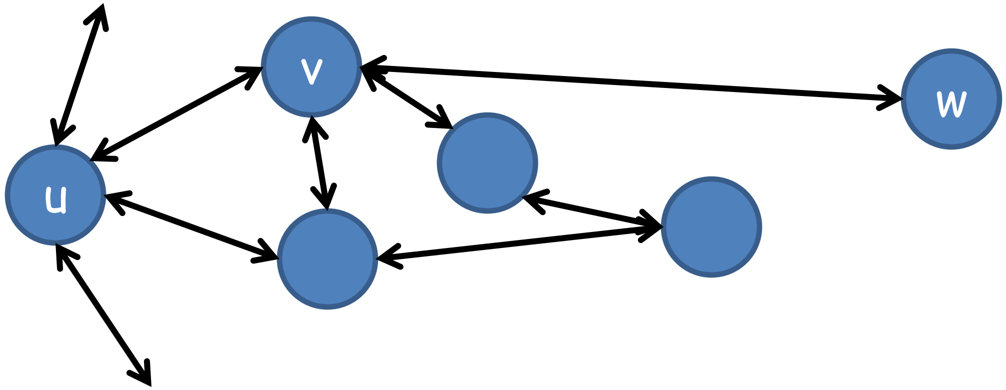
Network topology modeled as (un)directed graph G=(V,E)

Nodes V: processors and network switches – network elements
Edges E: links between network elements

(u,v) in E:
there is a direct link from element u to element v



Path from u to w:
length of shortest path
lower bounds communication
latency between u and w

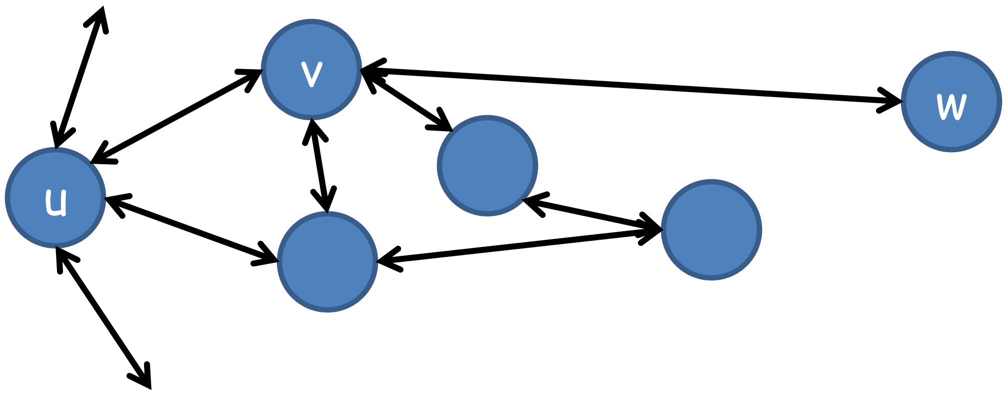


diameter(G): max(|shortest path(u,v)| over all u,v in V)

Lower bounds number of communication rounds for collective communication operations

degree(G): max degree (edges of) a node in G

„cost factor“. High-degree gives potential for more simultaneous communication (multi-port)



Note:
finding bisection width of arbitrary topology is NP-hard. Graph Partitioning

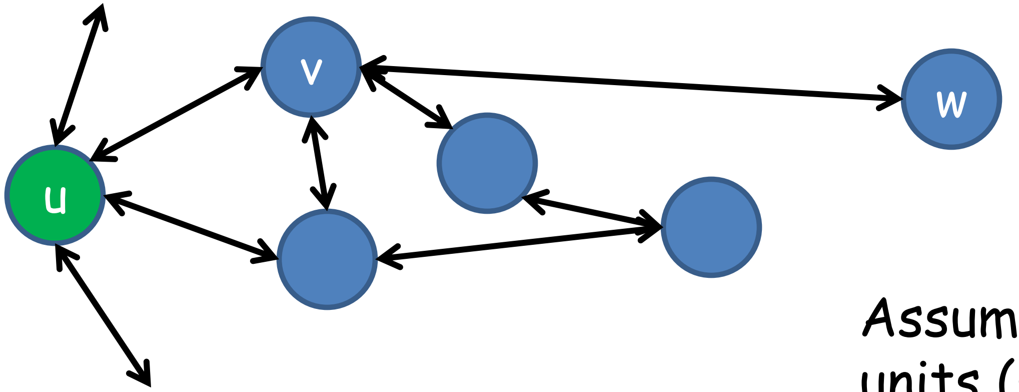
bisection width(G): minimum number of edges to remove to partition V into two equal-sized, disconnected parts

bisection width(G): min(|{(u,v) in E, u in V1, v in V2}|) over all partitions V1, V2 of V with |V1|≈|V2|)

Lower bounds transpose operations: all processors have to exchange information with all other processors

Broadcast in communication networks

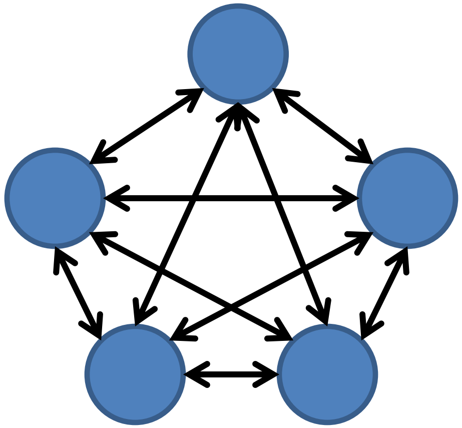
Problem: one processor has data to be communicated to all other processors. Processor with data initially called root



Assume data are indivisible units (still no cost model)

The ideal case: fully connected network

G =(V,E) is the complete graph, each processor is directly connected to each other processor

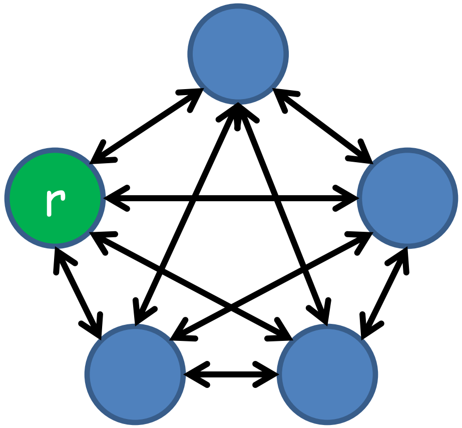


diameter = 1
bisection width = (p/2)^2

Expensive: p^2-p links (cables, switch-ports, …), degree = p-1

Broadcast in fully connected network

Problem: one processor has data to be communicated to all other processors. Processor with data initially called root



Algorithm:
1. If |V|=1 done
2. Divide processors into two roughly equal-sized sets V1 and V2
3. Assume root r in V1, choose local root rr in V2
4. Send data from r to rr
5. Recursively broadcast in V1 and V2

Algorithm:

1. If |V|=1 done
2. Divide processors into two roughly equal-sized sets V1 and V2
3. Assume root r in V1, choose local root rr in V2
4. Send data from r to rr
5. Recursively broadcast in V1 and V2

Analysis: assume communication takes place in synchronized communication rounds. After step 4, two problems of half the original size are solved independently. Algorithm takes ceil(log_2 p) rounds for all processors to have received data

Note: ceil(log_2 p)>diameter(G). Can we do better?

Algorithm:

1. If |V|=1 done
2. Divide processors into two roughly equal-sized sets V1 and V2
3. Assume root r in V1, choose local root rr in V2
4. Send data from r to rr
5. Recursively broadcast in V1 and V2

Fundamental lower bound:
At least ceil(log_2 p) communication rounds are needed for the broadcast problem.

Proof: in each round the number of processors that have the data can at most double (namely when each processor sends to a processor that did not have data)

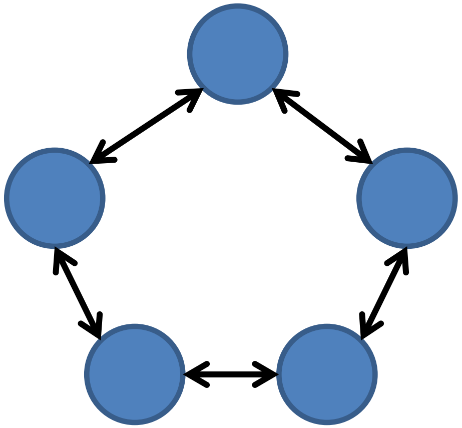
Algorithm:

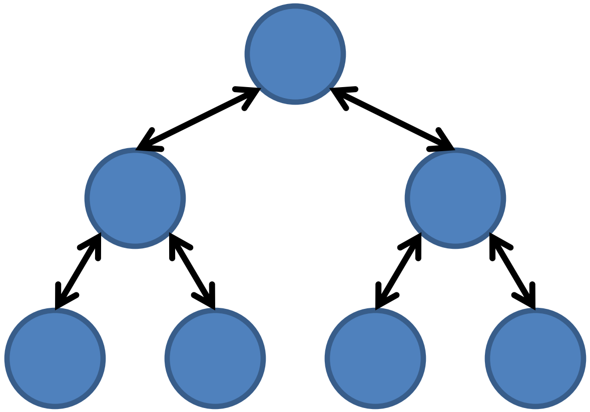
1. If |V|=1 done
2. Divide processors into two roughly equal-sized sets V1 and V2
3. Assume root r in V1, choose local root rr in V2
4. Send data from r to rr
5. Recursively broadcast in V1 and V2

Theorem:
recursive (binomial tree – why?) algorithm matches lower bound on number of communication rounds

Hidden assumption: only one communication operation per processor in each round (1-ported communication)

The worst case: linear array, ring, tree



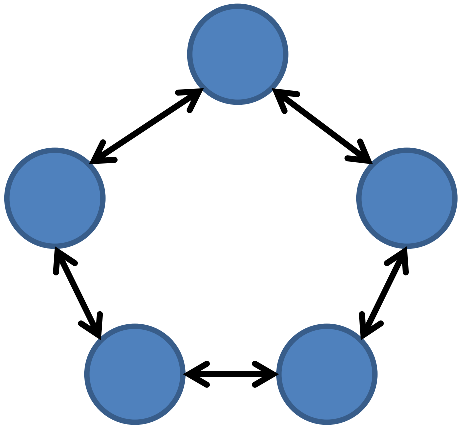


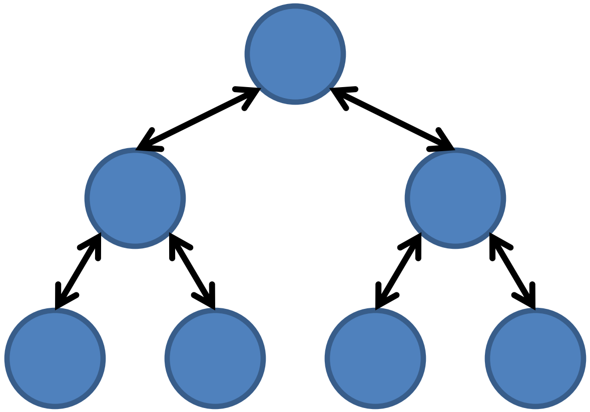
Both: removing one (two for ring) link disconnects network.
Bisection width is therefore 1 (2 for ring)

diameter = p-1 (p/2 for ring)   diameter = 2 log_2 ((p+1)/2)

Both: diameter determines broadcast complexity

The worst case: linear array, ring, tree



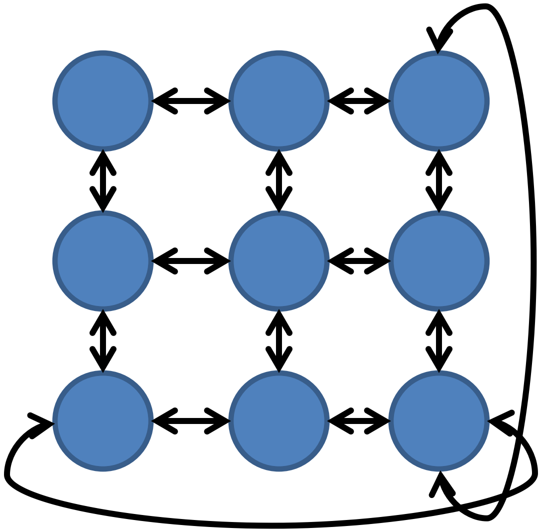


Both: removing one (two for ring) link disconnects network.
Bisection width is therefore 1 (2 for ring)

degree =2

degree = 3

Mesh, torus

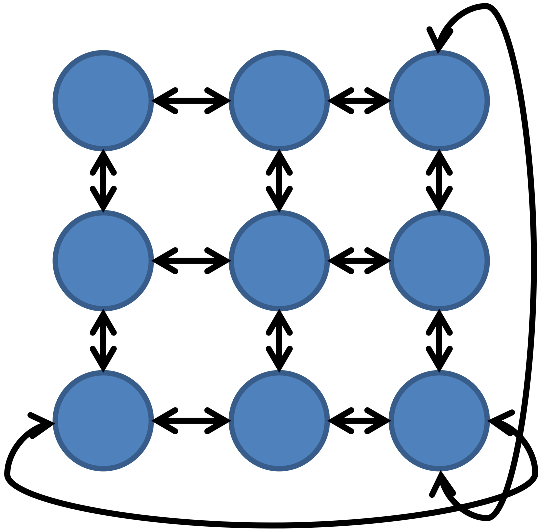


„wrap-around" for tori

$$\text{diameter(mesh)} = d\,(\sqrt[d]{p}-1)$$
$$\text{diameter(torus)} = d\,\text{floor}(\sqrt[d]{p}/2)$$

d'th root

Both: diameter determines broadcast complexity

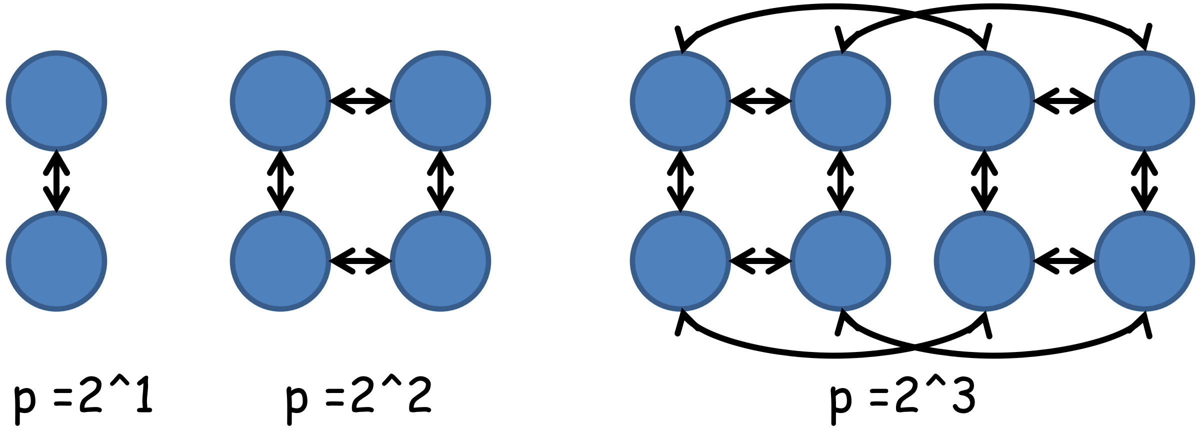


„wrap-around“ for tori

bisection width(mesh) = p^((d-1)/d)
bisection width(torus) = 2p^((d-1)/d)

Both: bisection bandwidth determines tranpose/alltoall communication complexity

Hypercube



k dimensional hypercube composed from 2 (k-1) dimensional hypecubes

diameter = k (= log_2 p)
bisection width = p/2
degree = k (= log_2 p)

Diameter determines broadcast complexity

Examples:

Fully connected:
rare, expensive; full crossbar between shared-memory nodes in NEC Earth Simulator (2002-2004). In switches of multi-stage networks

Ring: low-end, ethernet???

Tree: rare; fat tree variant common (perhaps later)

Mesh/Torus: Blue Gene (+ tree shaped collective network), Cray, Fujitsu K-1, (dead) Blue Waters

Other topologies (perhaps later lecture)

Multi-stage networks:

- Clos
- Butterfly
- Fat tree
- …

Routing terminology

Transmission cost model

Simple, first assumption

Cost of transmitting (indivisible) data of size m along edge (u,v) in communication network linear in m

$$T = \alpha+\beta m$$

α: „start-up" latency
β : time per unit (Byte)

In this model:
Recursive/binomial tree broadcast: log_2 p($\alpha+\beta m$)

Lower bound on broadcast in linear cost, fully connected network model is

min(α log_2 p,a+βm)

a log_2 p: log_2 p communication rounds, each communication incurring one „start-up"
βm: the m data units have to leave the root

Why not log_2 p(a+ βm) ?

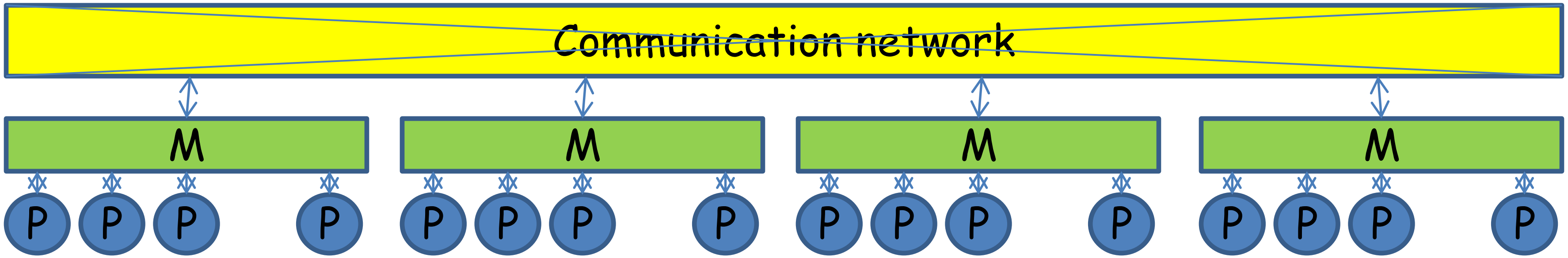
Answer: m need not be sent as one unit, „pipelining"

Question: possible to achieve both lower bounds?

Answer: yes; perhaps other lecture

Hybrid/hierarchical architectures:

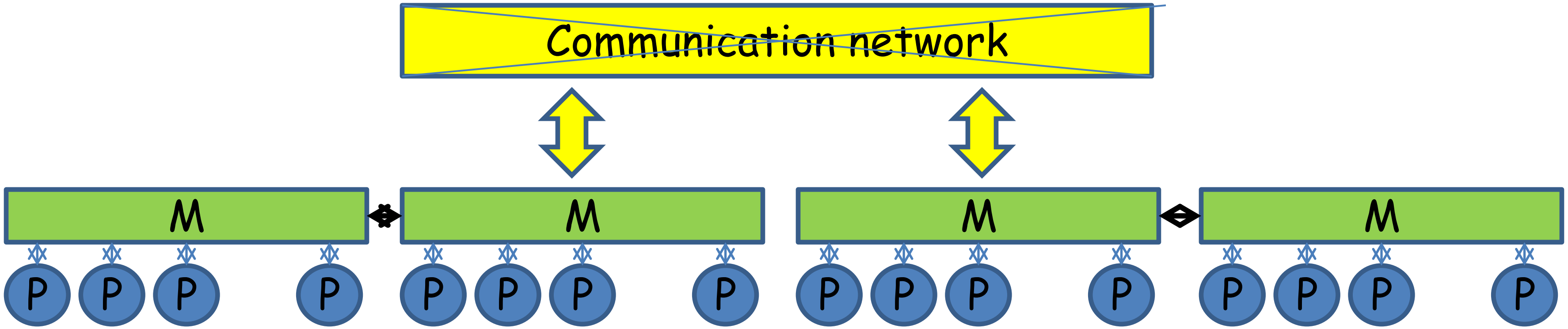
Shared-memory „nodes“ connected through communication network



E.g. traditional SMP cluster

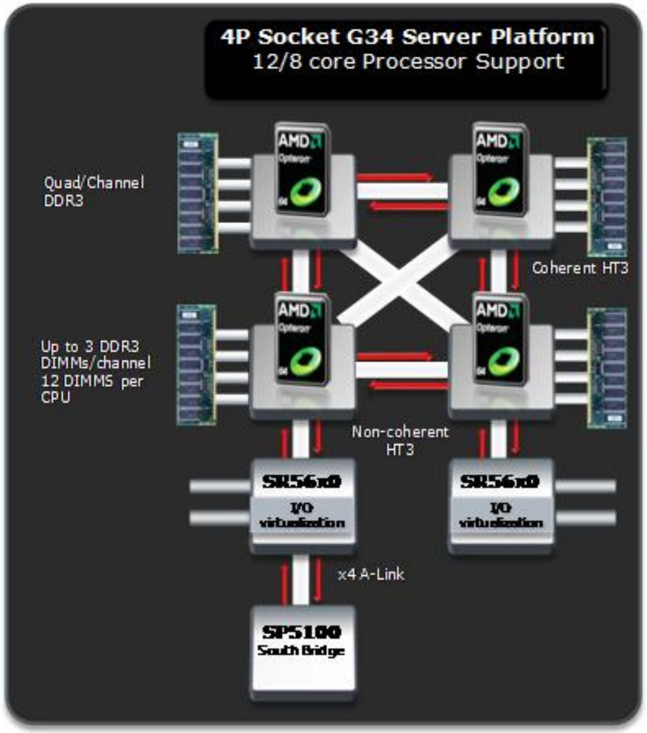
Hybrid/hierarchical architectures:

Shared-memory „nodes“ connected through communication network



Multi-core based SMP cluster

Shared vs. distributed: A matter of degree…


4P Socket G34 Server Platform
12/8 core Processor Support
Quad/Channel DDR3
Up to 3 DDR3 DIMMs/channel 12 DIMMS per CPU
AMD Opteron
Coherent HT3
Non-coherent HT3
SR56x0 I/O virtualization
SR56x0 I/O virtualization
x4 A-Link
SP5100 South Bridge

Shared memory architecture, because hardware transparently provides access to remote memory

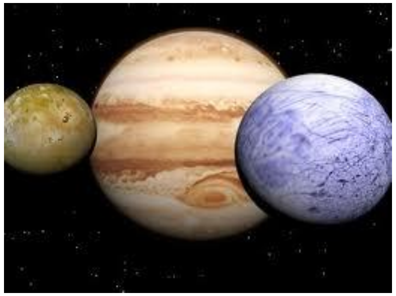
Programming-model wise: could make sense to treat as distributed memory system – to emphasize locality

TU Wien parallel computing hybrid distributed memory machine

- 36 shared-memory nodes
- InfiniBand QDR switch,
- Node with 2x8-core AMD „magny cours" processor, 2,3GHz
- 32 GByte shared-memory/node
- 1TB local disk/node

Name: jupiter.par.tuwien.ac.at

- Total 576 processor-cores
- Total 1052GByte (~1TB) system memory



Exercise: peak performance?

Mellanox InfiniBand switch MT4036

- 36 40Gb/s ports
- up to 2.88 Tb/s of available bandwidth
- latency of 100 nanoseconds

System configuration by NEC Empowered by Innovation

Basic software:
- NEC MPI
- Mpich2 MPI
- OpenMPI

MPI: the Message-Passing Interface

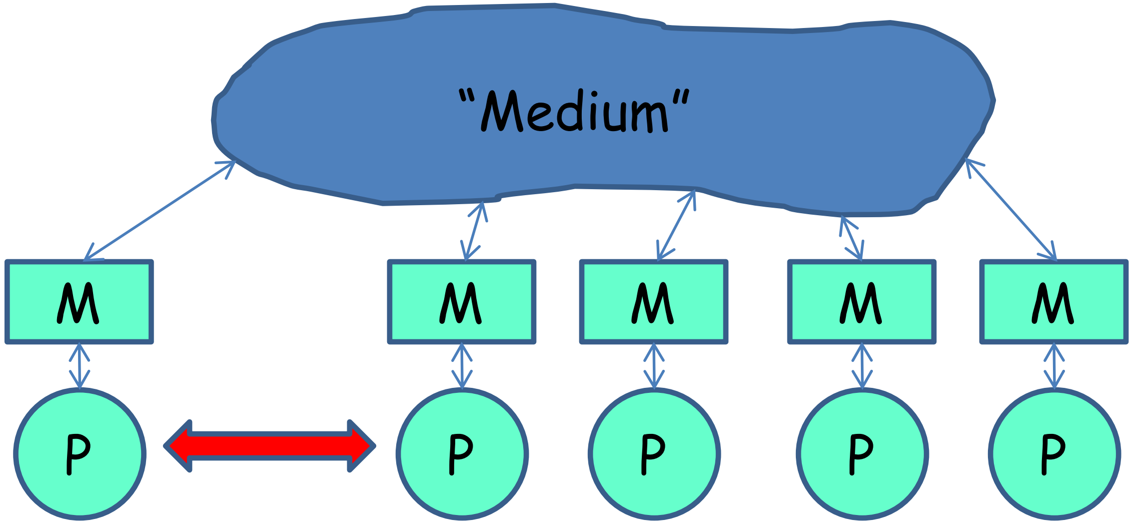
MPI – *the* Message-Passing Interface

De facto standard for parallel programming in the message passing paradigm; most well-known implementation of message passing, shared nothing programming model:

> Single applications on dedicated clusters and HPC systems with non-trivial communication requirements

- HPC applications (almost) exclusively with MPI
- Many, many parallel application for clusters, medium sized systems
- Paradigmatic realization of the message passing abstraction
- Well-engineered standard, lots to learn for other interfaces

Message passing abstraction/programming model



•Finite set of sequential processes communicate through a communication medium; communication between all processes possible

•Processes communicate by (explicitly) sending and receiving messages

•No implicit synchronization between processes, only communication

•Roots in e.g. CSP (Communicating Sequential Processes) [Hoare78]

•Semantic/logical abstraction

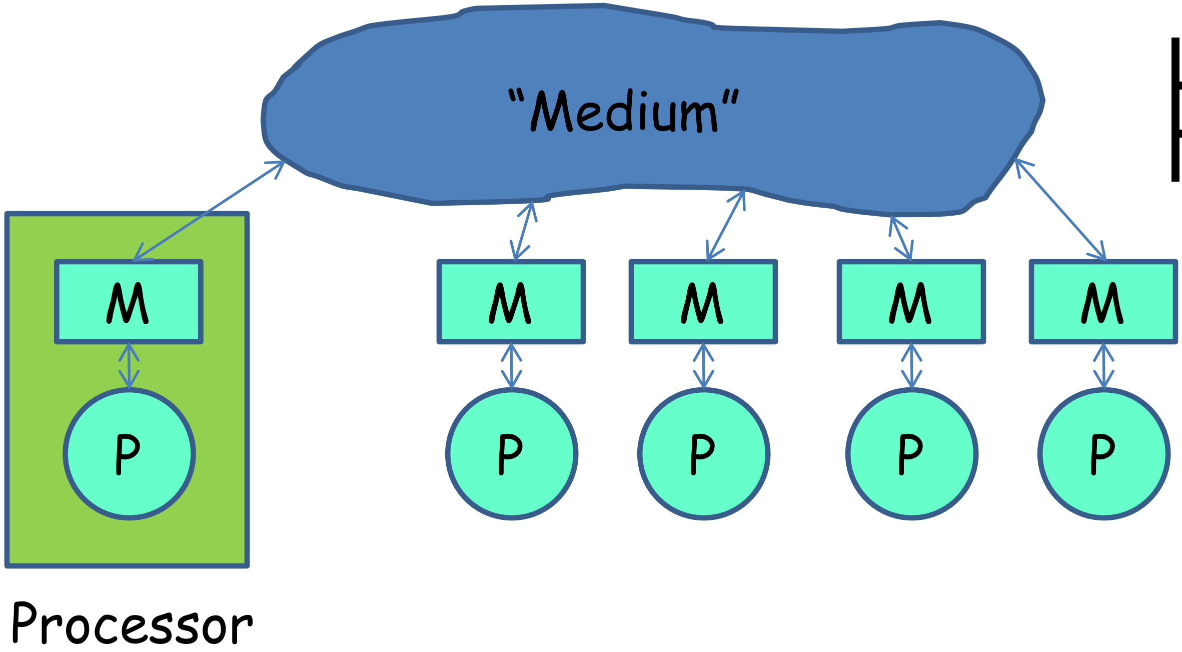
•No performance model

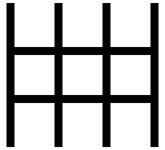
Inherent strengths of message passing model

No global data, no race conditions, no global clock, synchronization implicit with communication
•Enforces to think in terms of locality; where are the data?

Message passing abstraction

Communication medium realized by some physical communication network





Topology:
- Fully connected
- Mesh/Torus
- Fat tree
- …

Realization:
- InfiniBand
- Myrinet
- …

MPI realizes the message passing abstraction

- MPI processes bound to processors/cores
- Private address spaces, ordinary C or Fortran programs
- Explicit communication: point-to-point, collective, one-sided
- No performance model

… with many extra features

- Parallel I/O
- Dynamic process management
- Data descriptions
- Process topologies

MPI design principles/imperatives

- High-performance: communication functions close to typical "hardware" functionality, low protocol stack overhead
- Portability!!!! Scalability!!!
- Support library building, application specific libraries
- Memory efficient: little dynamic memory (O(1)?) needed by MPI functions, memory (communication buffers) in user-space
- Coexist with other parallel interfaces (OpenMP, threads, …)
- Support (not hinder) construction of tools
- Support heterogeneous systems (data representation)
- Support SPMD or MIMD paradigm

… and has been (quite) successful towards these goals

SPMD: Same Program, Multiple Data

Loosely synchronous, all processors run the same program, processes distinguish themselves by their rank (proceess ID)

MIMD: Multiple Programs, Multiple Data

Loosely synchronous, processors may run different programs, processes distinguish themselves by their rank (proceess ID)

MPI supports MIMD, application can consist of (many) different object files, most applications are SPMD, same object file

MPI realization

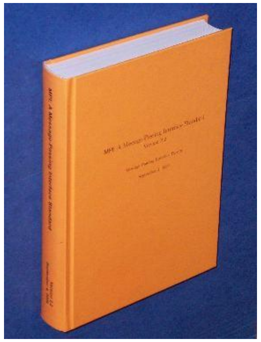
- Library, not a programming language!

- Pros: can be developed independently of compiler support, bindings for C and Fortran (not really C++), maximum freedom for library developer
- Cons: things that compiler knows cannot really be exploited, user sometimes have to convey information from language (data layouts) to library (tedious)

MPI is large

306 C functions in current MPI 2.2

but centered around few basic concepts



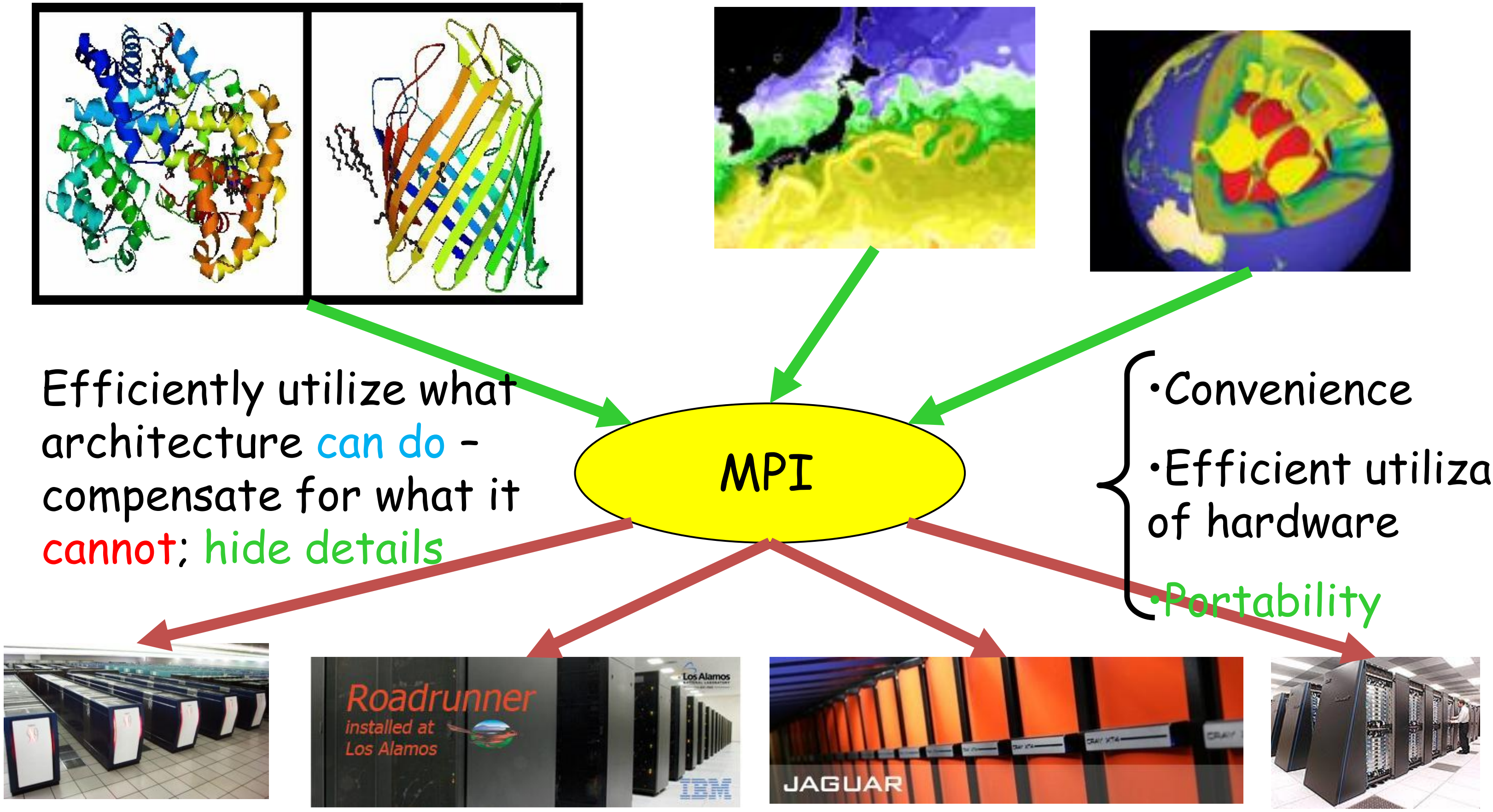
- Natural functionalities, use standard for concrete details

Often criticized as too low-level (“assembly language”)

> MPI designed “not to make easy things easy, but difficult things possible”
>
> W. Gropp, EuroPVM/MPI 2004

Challenge: be better than MPI! PGAS?

Role of MPI



Efficiently utilize what architecture can do – compensate for what it cannot; hide details

- Convenience
- Efficient utilization of hardware
- Portability

Coupled (multi-physics) applications are often MIMD/dynamic

Code/application portability:
Application developed on system A will run unchanged on system B; perhaps with recompilation/relinking. No code change/work-around needed

Requires: well-defined language, parallel interface; implementations that meet specifications

C/Fortran + MPI gives a high degree of application portability.

Shared-memory models (memory consistency, atomic operations, … architecture dependency), GPU models may not

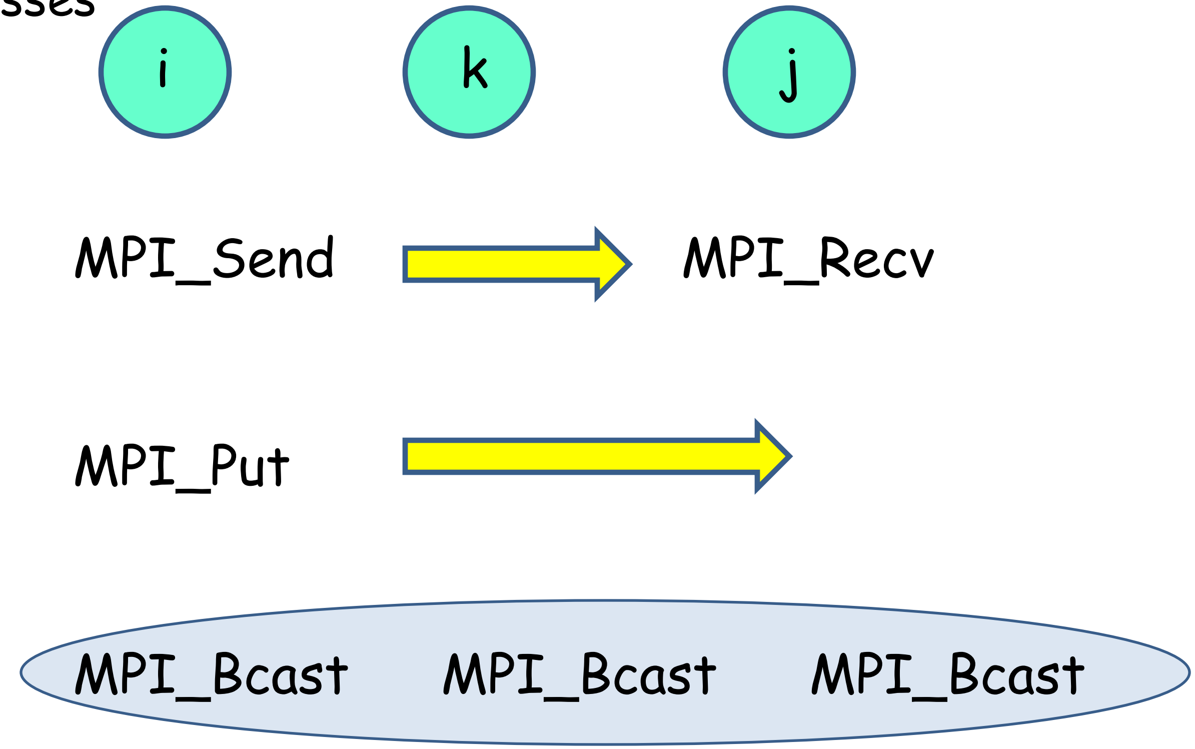
„Performance portability“:
Could mean: no change in application needed to efficiently exploit system B with code developed on system A

Distributed memory programming model could provide: all communication explicit, delegated to library (MPI)

Requires: efficient implementation of library for each new system, certain consistency conditions to be fulfilled

Major (performance) portability HPC disruption: transition from „vector“ to „scalar“ systems late 90ties – consult Top500

MPI communication models



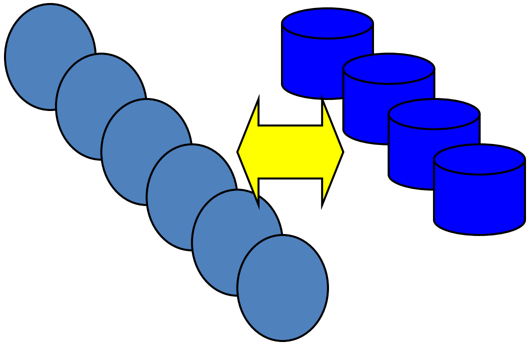
MPI processes

i k j

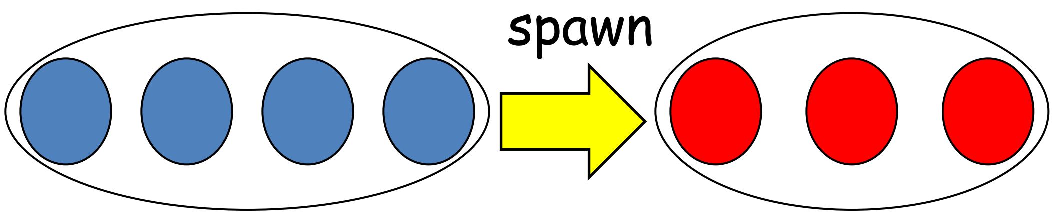
- Point-to-point: MPI_Send → MPI_Recv
- One-sided: MPI_Put →
- Collective: MPI_Bcast MPI_Bcast MPI_Bcast

Extended "communication"

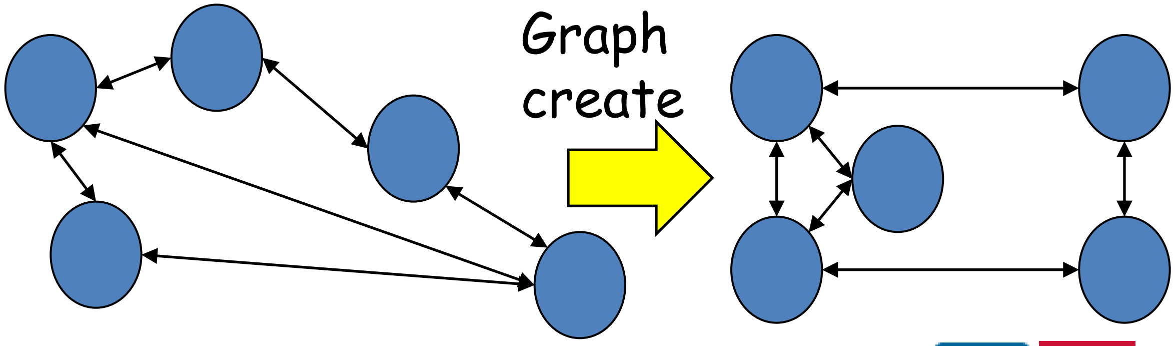
- Parallel I/O:



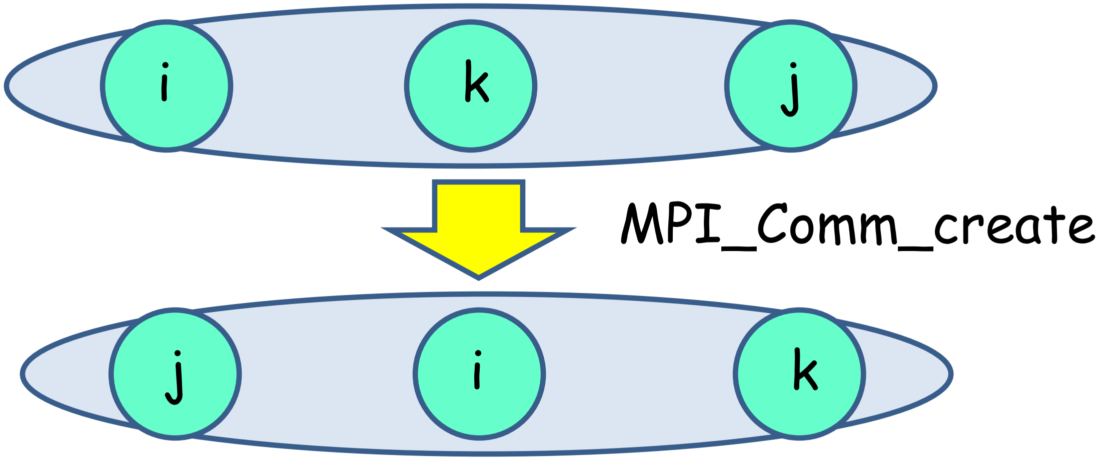
- Process management:



- Virtual topologies:



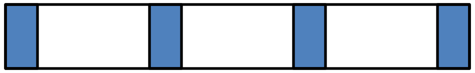
Library building



- Communicator management

- Attributes – additional information attached to MPI objects

- Datatypes:



MPI_Type_vector

Basic concepts

1. Communicators/process groups/windows – sets of processes that can communicate
2. Data types – for description of data layouts in memory
3. Local and non-local (collective) completion semantics
4. Blocking and non-blocking communication

MPI standard

Not a formal specification, trying to be precise, sometimes (intentionally) vague… :

- Progress rule (*)
- Modalities (when things will happen: immediately, eventually, …)
- No performance model (**)

(*) to avoid prescribing a specific kind of implementation (communication thread, e.g.)

(**) specific requirements might not be feasible for all communication systems; could limit portability of MPI

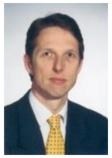
Before MPI (early 90ties)

Distributed memory machines (Intel hypercube, IBM SP systems, Meiko computing surface, …) with own message-passing interfaces or language extensions

- Intel NX
- Meiko
- IBM CCL
- Zipcode
- PARMACS
- OCCAM
- …

⇨ Lots of commonalities, need for a standard (ca. 1994)

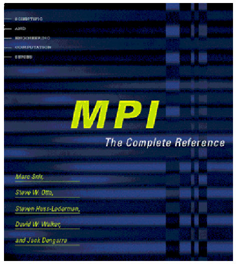
Evolution of the MPI Standard



⋮

- 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

Growing experience with MPI 2.0 extensions from 2000ff…

Some positive (RMA on Earth Simulator), some (very) negative…

Pressure from various sides, new MPI implementations (OpenMPI), new players (Microsoft)

No replacement for MPI on the horizon (despite many interesting efforts, HPCS, PGAS, …)

EuroPVM/MPI 2006 (Bonn), 2007 (Paris): "Open Forum"

Late 2007: MPI Forum starts convening regularly again

MPI Forum (December 2007ff):

MPI 2.1: consolidation, minor error corrections (issues accumulated over past 5 years)

MPI 2.2: mild extensions, not allowed to break existing code

MPI 3.0: genuine additions to standard, may break existing code (recompilation necessary, possibly smaller rewrites)

+

Dublin, 4th September 2008

June 2009

MPI Forum – towards MPI 3.0

- Open body maintaining the MPI standard
- Not a formal (IEEE, ANSI) standardizations body
- Everybody can participate

- Discussions: wiki/TRAC at www.mpi-forum.org + mailing lists
- Regular meetings every 6-8 weeks, mostly US, Europe with EuroMPI conference
- Regular participation required to vote

- 30-50 organizations involved, about 30 participants at meetings
- All major MPI developers (mpich, openMPI, mvapich,…), all major vendors, major labs with applications

- More application input, please!

MPI programming model

1. Set of processes (in communication domain) that can communicate
2. Processes identified by rank in communication domain
3. Ranks successive 0, …, p-1; p number of processes in domain (size)
4. More than one communication domain possible; created relative to default domain of all started processes

5. Processes operate on local data, all communication explicit

6. Three basic communication models:

    6. Point-to-point communication – different modes, non-local and local completion semantics
    7. One-sided communication – different synchronization mechanisms, local completion mechanisms
    8. Collective operations, non-local completion semantics (*)

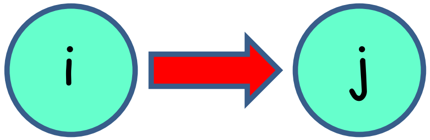
7. Structure of communicated data orthogonal to model/mode

8. Communication domains may reflect physical topology

9. No communication cost model

(*) MPI 3.0 will feature non-blocking collective operations

Point-to-point communication



```
MPI_Send(buffer,count,datatype,tag,rank,comm);
```

```
MPI_Recv(buffer,count,datatype,tag,rank,comm,&status);
```

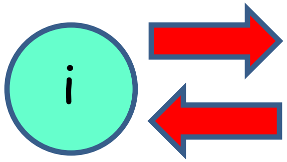
User-space buffers of any size, arbitrary structure can be communicated, no limitations

Native (e.g. InfiniBand) communication system may have all sorts of restrictions (e.g. consecutive data, max size)

Processes identified by a rank in a communication domain (communicator)

Different communication modes and semantics

One-sided communication



Only one process (conceptually) involved. Abstracts remote memory access, supported natively by some networks, not all

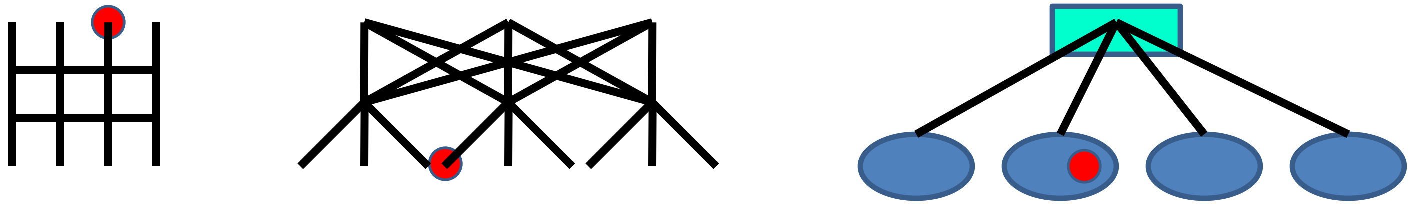
```
MPI_Put(origin_buffer,origin_count,origin_type,
        target,
        target_displacement,target_count,target_type,
        win);
```

```
MPI_Get(…);
```

Memory exposed as communication window. Origin specifies communication with target. Any size and structure.

Collective communication

MPI_Bcast – one root process has data, everybody else needs



Strive for best possible performance on given network/topology

Leave details to MPI implementer!

„Performance portability“

```
MPI_Bcast(buffer,count,datatype,root,comm);
```

Any size and structure

MPI_Bcast – data from root to all

MPI_Scatter – individual (personalized) data from root to all
MPI_Gather – individual data from all to root

MPI_Alltoall – individual (personalized) data from all to all, "transpose)

MPI_Allgather – data from all to all

MPI_Reduce – apply associative function (e.g. "+") to data from each process, result at root

MPI_Allreduce – result to all
MPI_Reduce_scatter – result scattered (parts) to all

MPI_Barrier – (semantic) synchronization

Safe parallel libraries

Communication inside library independent of communication outside library, no interference

```
MPI_Comm_dup(comm,newcomm);
```

Attributes to record state, properties of library (communicators and other objects)

MPI attribute mechanism not in this lecture

Parallel Computing

Additional literature:

- MPI standard, MPI 2.2 www.mpi-forum.org/docs/mpi-2.2/mpi22-report.pdf

- Gropp, Lusk, Skjellum: Using MPI. Portable Parallel Programming… MIT Press 1995
- Gropp, Lusk, Thakur: Using MPI-2: Advances features… MIT Press 1999

- Karniadakis, Kirby: Parallel Scientific Computing in C++ and MPI. Cambridge University Press, 2003
- Peter S. Pacheco: Parallel Programming with MPI, Morgan-Kaufmann, 1997
- Michael J. Quinn: Parallel Programming in C with MPI and OpenMP, McGraw-Hill 2003

First 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);

  fprintf(stdout,”Here is %d out of %d\n”,rank,size);

  MPI_Finalize();
  return 0;
}
```

First 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);

  fprintf(stdout,”Here is %d out of %d\n”,rank,size);

  MPI_Finalize();
  return 0;
}
```

Standard MPI header
FORTRAN:
INCLUDE „mpif.h“

First 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);

  fprintf(stdout,”Here is %d out of %d\n”,rank,size);

  MPI_Finalize();
  return 0;
}
```

First MPI call, performed by all. Exception MPI_Initialized(flag)

Last MPI call, must be performed by all. Exception MPI_Finalized(flag)

First 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);

  fprintf(stdout,"Here is %d out of %d\n",rank,size);

  MPI_Finalize();
  return 0;
}
```

Initial communication context, set of processes

Who am I?

Compiling and running MPI programs

- mpicc, mpif77, mpif90 – like cc, f77, f90
- mpirun –np <procs> …

- Batch system?
- See later

MPI Conventions

“Namespace”, C

```
err = MPI_<some MPI function>(…);
```

MPI function may return an error code (normally MPI_SUCCESS), but often just abort on error

“Namespace”, Fortran

```
CALL MPI_<some MPI function>(…,IERROR)
```

MPI constants (MPI_SUCCESS, MPI_INT, …) allCAPS

MPI_ - prefix reserved, don’t use in own programs!!

Good practice to always check error status – MPI programmers often don't…

Error behavior can be controlled to some extent by error handlers

```
MPI_Comm_set_errhandler(comm,errhandle)
```

errhandle: handle to function that will be called on error…

BUT(!!): „text that states that errors *will* be handled, should be read as *may* be handled", MPI 2.2, p. 276

```
MPI_Abort(comm,errorcode)
```

In practice, most often no error handling in MPI. Abort

MPI error codes
MPI_SUCCESS
MPI_ERR_BUFFER
MPI_ERR_COUNT
MPI_ERR_TYPE
MPI_ERR_TAG
MPI_ERR_COUNT
MPI_ERR_RANK
…
MPI_ERR_UNKNOWN
MPI_ERR_TRUNCATE
…
MPI_ERR_WIN
MPI_ERR_LASTCODE

New error codes/classes can be defined (use: own, higher-level libraries)

Sometimes returned in point-to-point

MPI standard bindings

"language independent":

```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm)
```

IN sendbuf
OUT recvbuf
IN count
IN datatype (handle)
IN op (handle)
IN root
IN comm (handle)

C prototype

```
int MPI_Reduce(void *sendbuf,
               void *recvbuf, int count,
               MPI_Datatype datatype,
               MPI_Op op, int root, MPI_Comm comm);
```

OUT arguments: pointers
IN arguments: pointers or value
Handles: special MPI typedef's

FORTRAN binding

```
MPI_REDUCE(SENDBUF, RECVBUF, COUNT, DATATYPE, OP,
           ROOT, COMM, IERROR)
<type> SENDBUF(*),  RECVBUF(*)
INTEGER COUNT, DATATYPE, OP, ROOT, COMM, IERROR
```

Handles are INTEGERs (problems with F90 typing)

The 6 basic functions

```
MPI_Init(&argc,&argv);
MPI_Finalize();
```

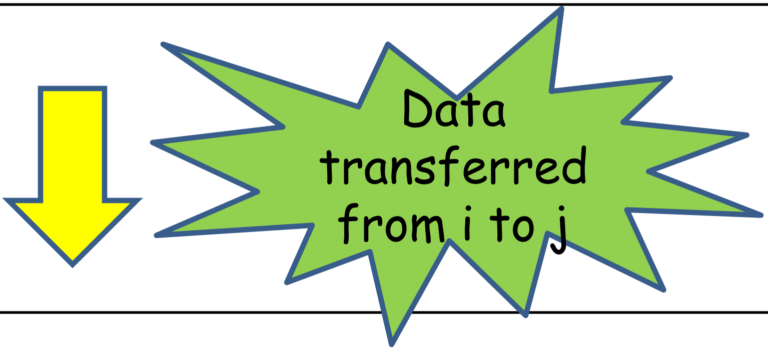
First and last call in MPI part of application; can only be called once

„Who/where am I?“ in communication context/set of processes. numbered from 0 to size-1

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

Process rank i:

```
int a[N];
float area;
MPI_Send(a,N,MPI_INT,j,TAG1,MPI_COMM_WORLD);
MPI_Send(&area,1,MPI_FLOAT,j,TAG2,MPI_COMM_WORLD);
```



Data transferred from i to j

Process rank j:

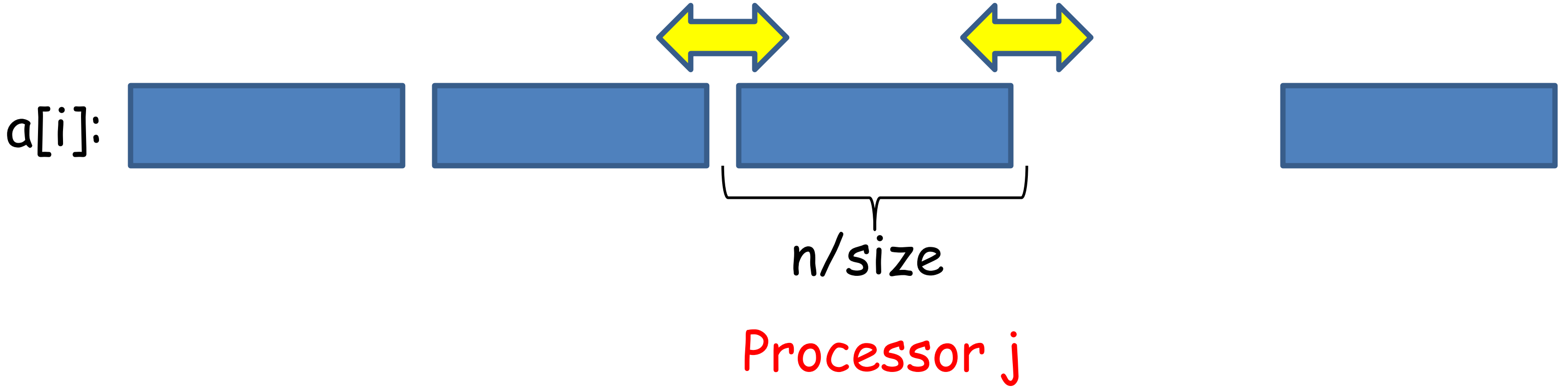
```
int b[N];
float area;
MPI_Recv(b,N,MPI_INT,i,TAG1,MPI_COMM_WORLD,&status);
MPI_Recv(&area,1,MPI_FLOAT,i,TAG2,MPI_COMM_WORLD,
         &status);
```

Example: loop with some dependencies

Processor j, 0≤j<p

```
for (i=n[j]; i<n[j+1]; i++) {
  b[i] = a[i-1]+a[i]+a[i+1];
}
```

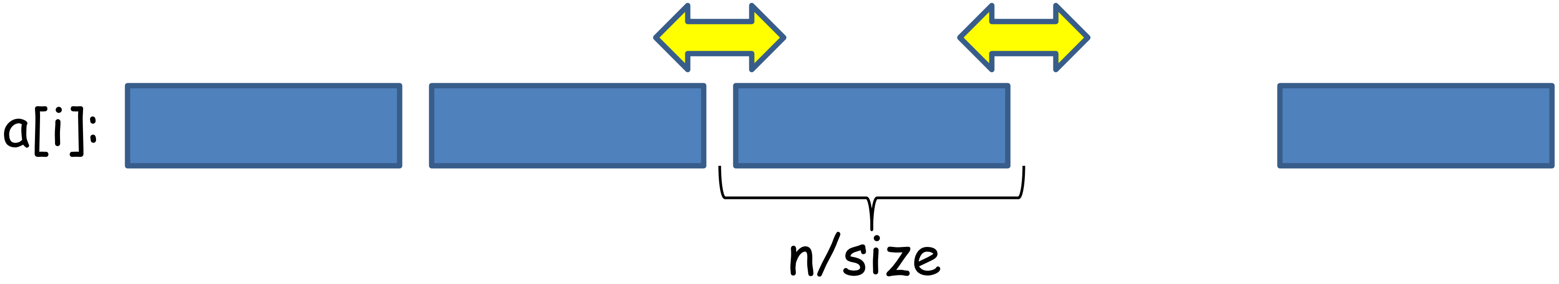
Arrays a and b distributed in blocks over processes



Parallelization of data parallel loop example

```
float *a = malloc((n/p+2)*sizeof(float));
a += 1; // offset, such that -1 and n/p can b addressed
if (rank>0) {
  MPI_Send(&a[0],1,MPI_FLOAT,rank-1,999,comm);
  MPI_Recv(&a[-1],1,MPI_FLOAT,rank-1,999,comm,&status);
}
if (rank<size-1) {
  MPI_Send(&a[n/p-1],1,MPI_FLOAT,rank+1,999,comm;
  MPI_Recv(&a[n/p],1,MPI_FLOAT,rank+1,999,comm,&status);
}
for (i=0; i<n/p; i++) {
  b[i] = a[i-1]+a[i]+a[i+1];
}
```

Why is this wrong ???



a[i]:

n/size

Process j

```
MPI_Send(…,rank-1, …);
MPI_Recv(…,rank-1,…);
```

```
MPI_Send(…,rank-1,…);
MPI_Recv(…,rank-1,…);
```

```
MPI_Send(…,rank+1,…);
MPI_Recv(…,rank+1,…);
```

DEADLOCK! All processes waiting to send ?

In MPI: behavior depending on data size - unsafe

DEADLOCK:

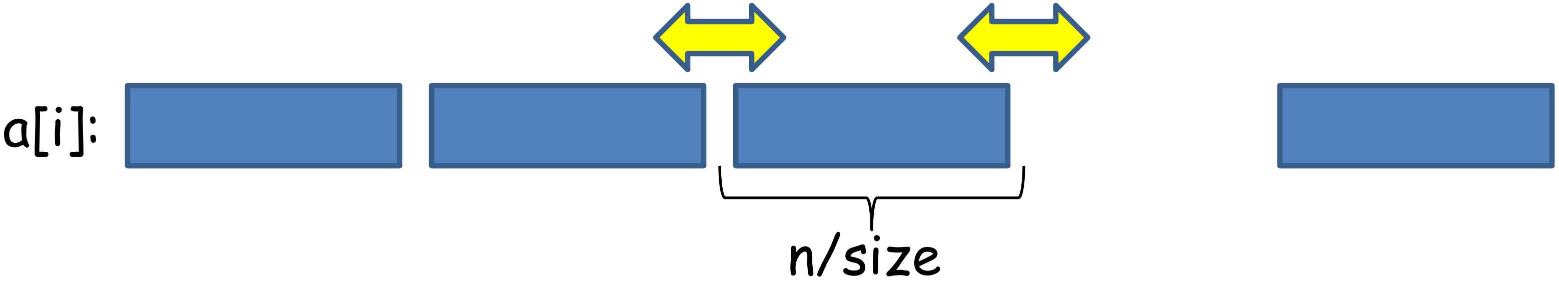
a. All processes waiting for event that does not/cannot happen
b. Process i waiting for action by process j, process j waiting for action by process i
c. Process i0 waiting for action by process i1, process i1 waiting for action by process i2, … process i(p-1) waiting for action by process i0

All forms are possible with MPI programs

Particularly problematic: some are context and MPI library implementation dependent: unsafe programming (see later)

Correct(er)

```
float *a = malloc((n/p+2)*sizeof(float));
a += 1;
if (rank>0) {
  MPI_Send(&a[0],1,MPI_FLOAT,rank-1,999,comm);
  MPI_Recv(&a[-1],1,MPI_FLOAT,rank-1,999,comm,&status);
}
if (rank<size-1) {
  MPI_Recv(&a[n/p],1,MPI_FLOAT,rank+1,999,comm);
  MPI_Send(&a[n/p-1],1,MPI_FLOAT,rank+1,999,comm,
           &status);
}
for (i=0; i<n/p; i++) {
   b[i] = a[i-1]+a[i]+a[i+1];
}
```



```
MPI_Send(…,rank-1, …);
MPI_Recv(…,rank-1,…);
```

Process j

```
MPI_Send(…,rank-1,…);
MPI_Recv(…,rank-1,…);
```

```
MPI_Recv(…,rank+1,…);
MPI_Send(…,rank+1,…);
```

```
MPI_Recv(…,rank+1,…);
MPI_Send(…,rank+1,…);
```

Serialization: Last process size-1 receives after 2p steps!

The 6 basic functions (plus two)…

Get time (in micro-seconds with suitably high resolution) since some time in the past:

```
float point_in_time = MPI_Wtime();
```

Synchronize the processes (really: only semantically); often used for benchmarking applications

```
MPI_Barrier(MPI_COMM_WORLD);
```

MPI: pt2pt and one-sided comm

- Communicators
- Point-to-point communication
- One-sided communication

Communication , processes, communicators

```
mpirun –np <procs> <program>
```

starts <procs> MPI processes executing <program> on available resources (processors, cores, threads, …)

Same <program> will run on all resources: SPMD

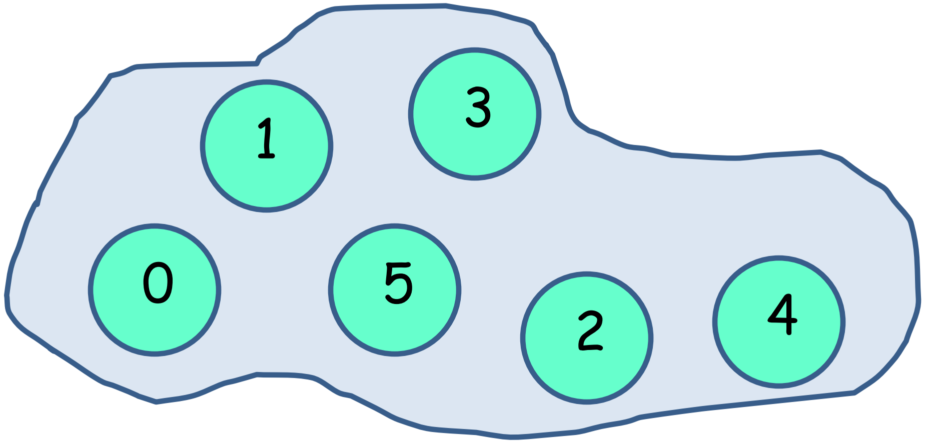
Other options to mpirun can influence where/which programs are started, rank order of MPI processes, etc.

Note: not standardized, see local installation/manpages

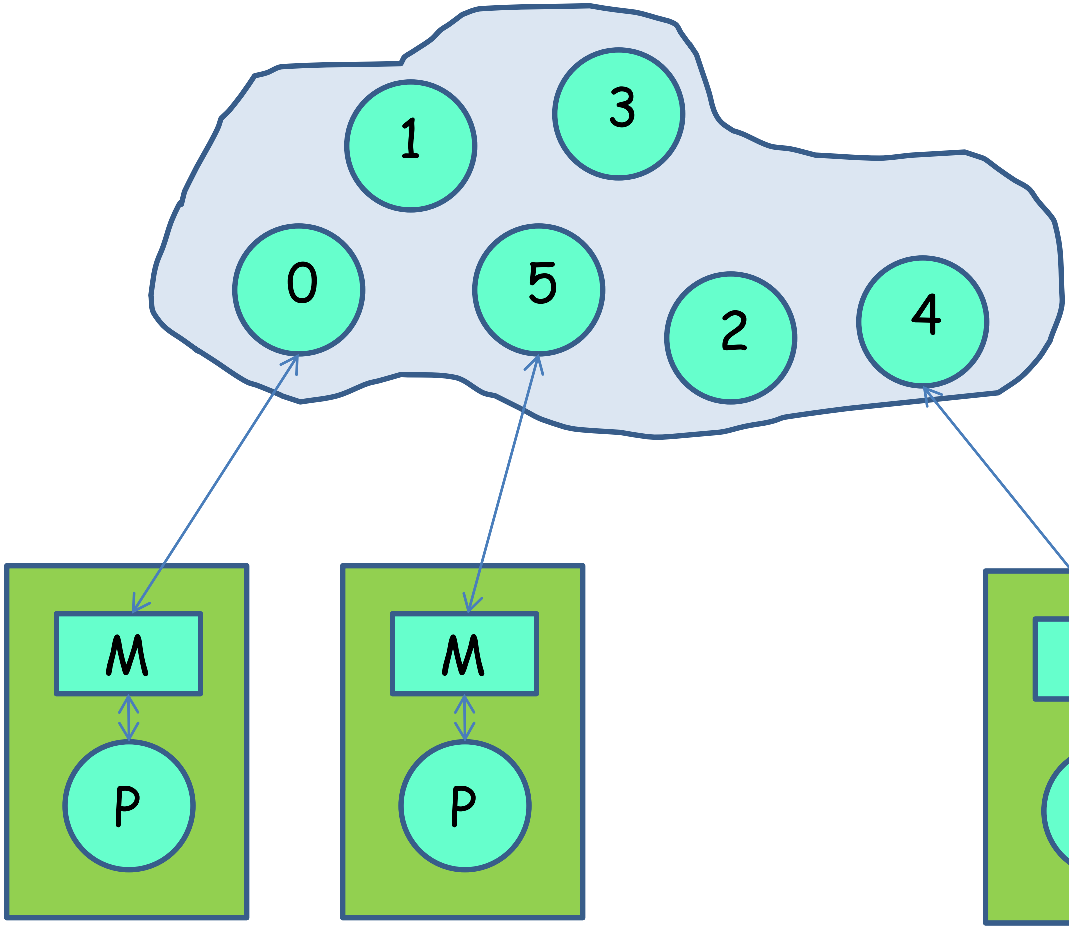
<program> executes

```
MPI_Init(&argc,&argv);
// sets up internal data structures, incl:
…
MPI_Comm_size(MPI_COMM_WORLD,&size);
```

MPI_COMM_WORLD: initial communicator containing all started processes; static - <u>never changes</u>!



Communicator: distributed, global object, communication context, finite set of processes that can communicate



Communicator: distributed, global object, communication context, finite set of processes that can communicate

Binding of MPI processes (statically) to processors outside of MPI, not standardized

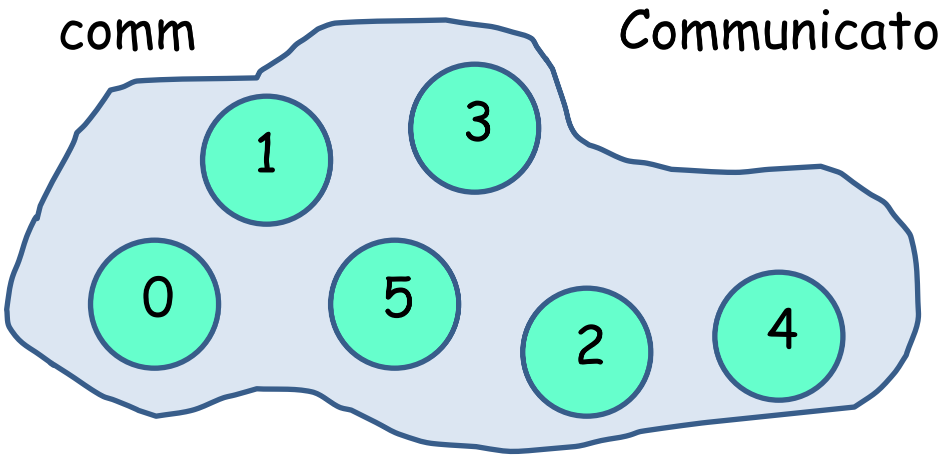
Physical processor may run more than one MPI process

Good SPMD practice:
Write programs to work correctly for any number of processes

```
MPI_Comm_rank(MPI_COMM_WORLD,&rank);
if (rank==0) {
  // code for rank 0; may be special
} else if (rank%2==0) {
  // remainder even ranks
} else if (rank==7) {
  // another special one
} else {
  // all other (odd) processes - perhaps do nothing?
}
```

Bad taste/dangerous practice:
don't rely on C conventions: if (rank) {…}

Communicators, universal object, ALWAYS:



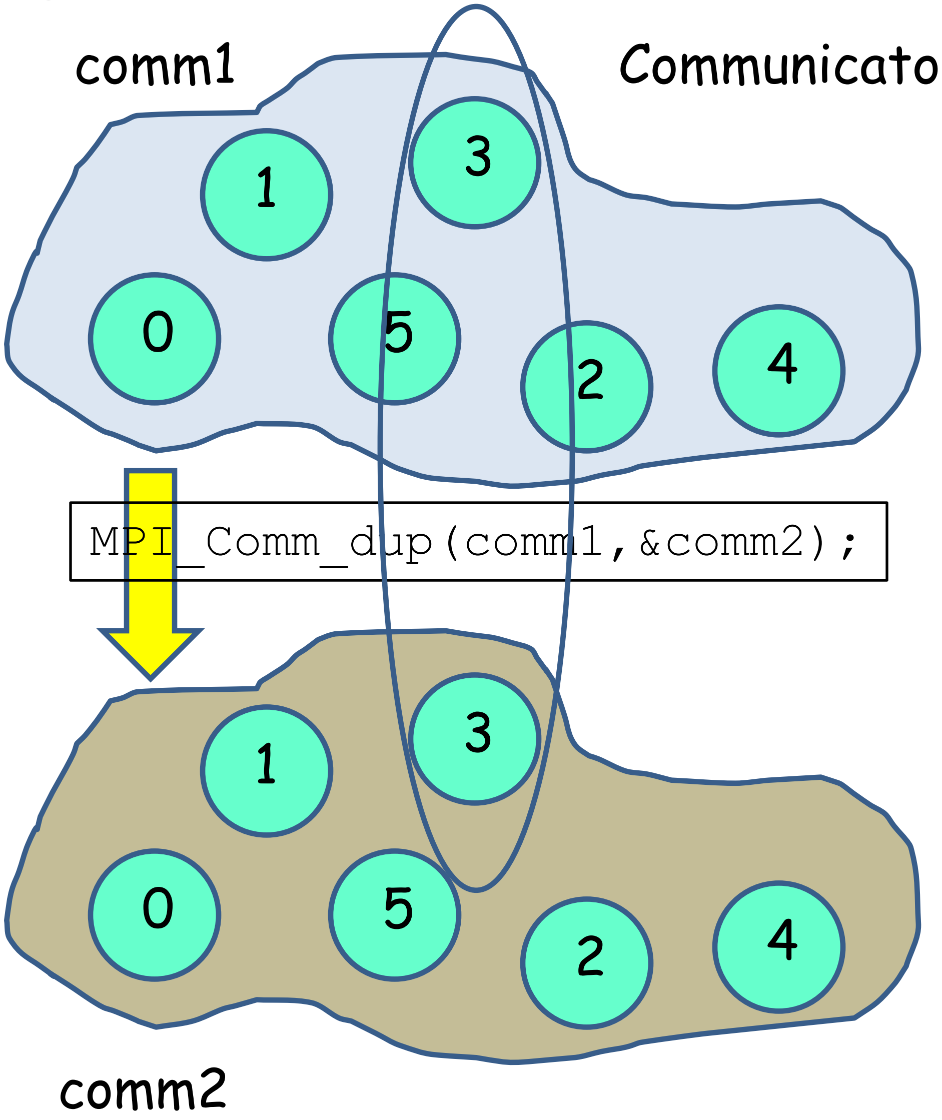
```
MPI_Comm_size(comm,&size);
MPI_Comm_rank(comm,&rank);
```

- All processes in a communicator can communicate
- All models (point-to-point, one-sided, collective; all other functionality)
- Has a size: number of processes
- Each process has a rank (0≤rank<size)
- A process can belong to several communicators (at the same time)

MPI process: (normally) statically bound to some processor resource; can have different ranks in different communicators;

canonically identified by rank in MPI_COMM_WORLD

Communicators, universal object, ALWAYS:



```
MPI_Comm_dup(comm1,&comm2);
```

- All processes in a communicator can communicate
- All models (point-to-point, one-sided, collective; all other functionality)
- Has a size: number of processes
- Each process has a rank (0≤rank<size)
- A process can belong to several communicators (at the same time)

Good practice, when building own libraries

```
int my_special_library_init(comm,&libcomm)
{
  MPI_Comm_dup(comm,&libcomm);

  // library communication wrt. libcomm; store somewhere
  // initialize other library data structures
  // could be cached with libcomm (attributes)
}
```

MPI_Comm_dup:
Collective function, MUST be called by all processes in comm

MPI handles

MPI_COMM_WORLD, comm1, comm2:
An MPI (predefined) handle, a way to access MPI objects (communicators, windows, datatypes, attributes)

- Handles are (almost always) opaque, i.e. internal MPI data structures cannot be accessed; but only manipulated through the operations defined on them
- MPI does not define how handles are represented (index into table, pointer, …)
- Handles in C and Fortran may be different

MPI_Comm_f2c(comm) [for example]:
returns C handle of Fortran communicator (no error code here)

Other MPI handles

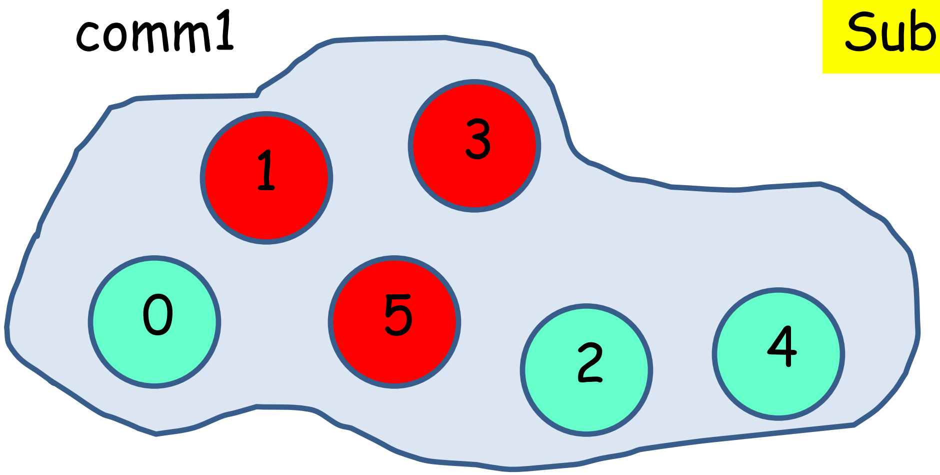
- MPI_Comm: communicators
- MPI_Group: process groups
- MPI_Win: windows for one-sided communication

- MPI_Datatype: datatypes (basic/primitive – or user-defined/derived)
- MPI_Op: binary operators (built-in or user defined)

- MPI_Request: request handle for point-to-point
- MPI_Status: communication status

- MPI_Errhandler:
- …

Subcommunicators:

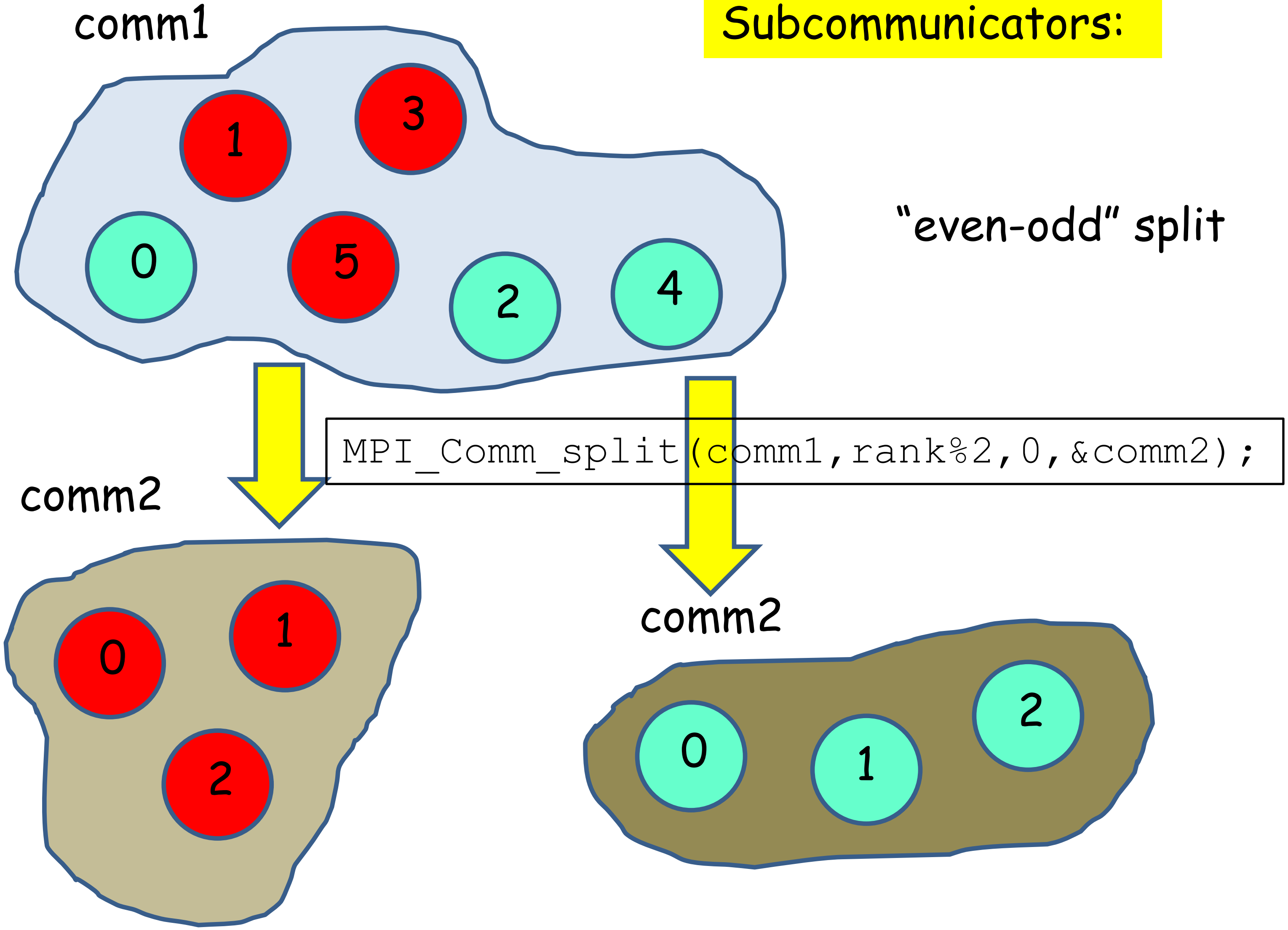


"even-odd" split

```
MPI_Comm_split(comm1,rank%2,0,&comm2);
```

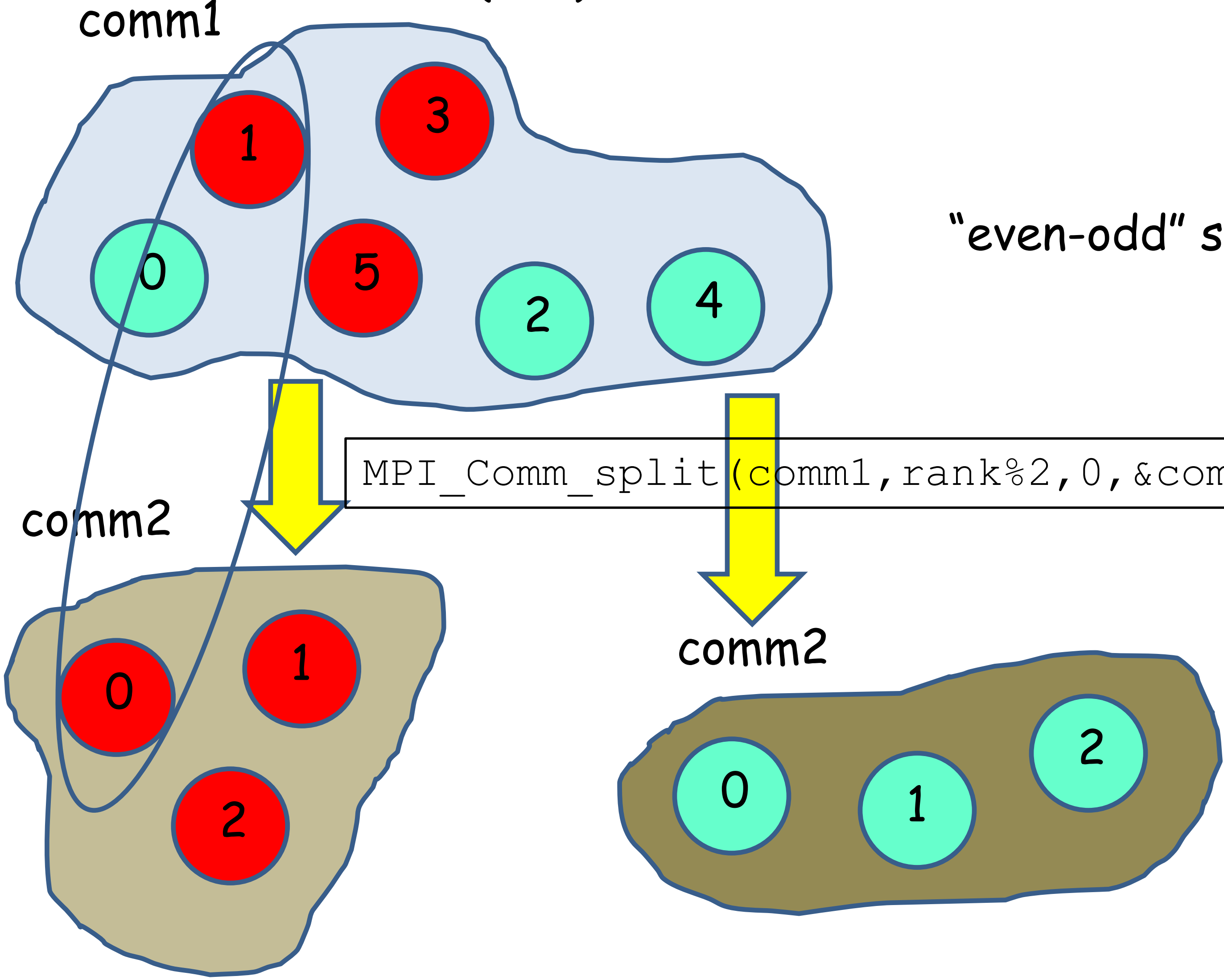
Subcommunicators:

"even-odd" split



```
MPI_Comm_split(comm1,rank%2,0,&comm2);
```

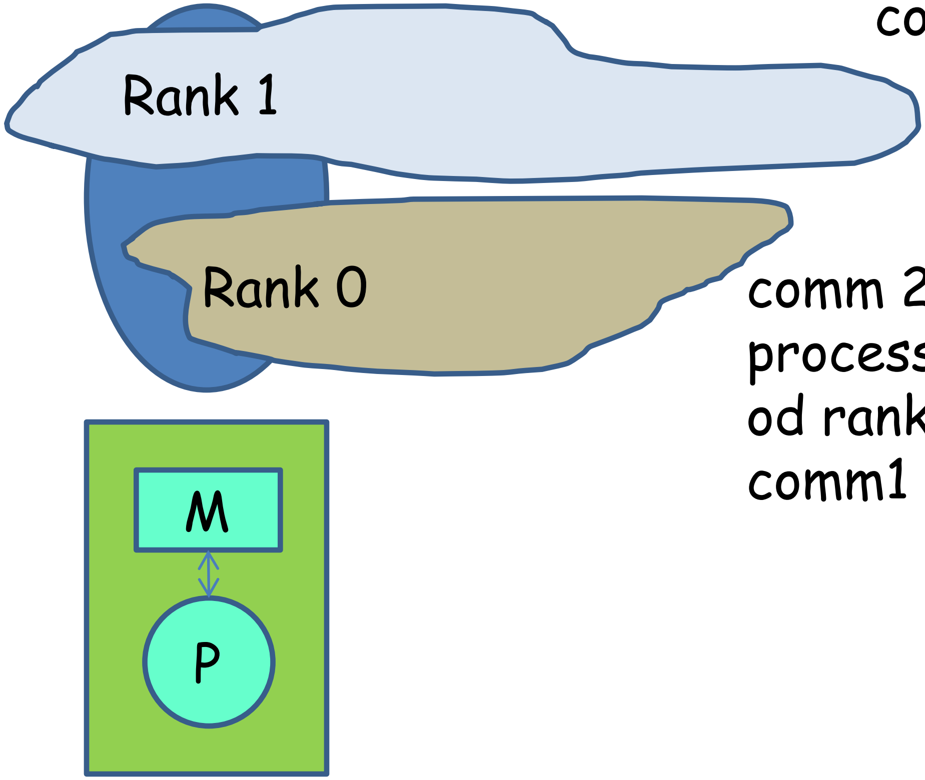
Rank 1 (odd) in comm1 has rank 0 in comm2



"even-odd" split

```
MPI_Comm_split(comm1,rank%2,0,&comm2);
```

MPI process



comm 1

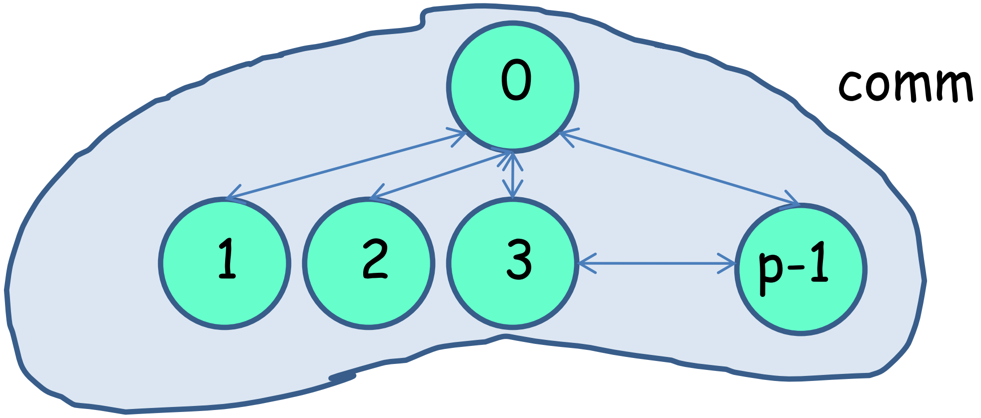
comm 2:
processes with
od rank in
comm1

```
MPI Comm comm1, comm2;

MPI_Comm_rank(comm1,&rank); // get rank in comm1

MPI_Comm_split(comm1,rank%2,0,&comm2);
// Collective operation: all processes in comm1 must call

/* comm2:
   two different communication domains for even and odd
   processes
*/
```

MPI_Comm_split (collective operation):
All processes with same color are grouped, order determined by key

Use:
parallel "divide-and-conquer" applications, computations in subcommunicators fully independent (collectives, everything)

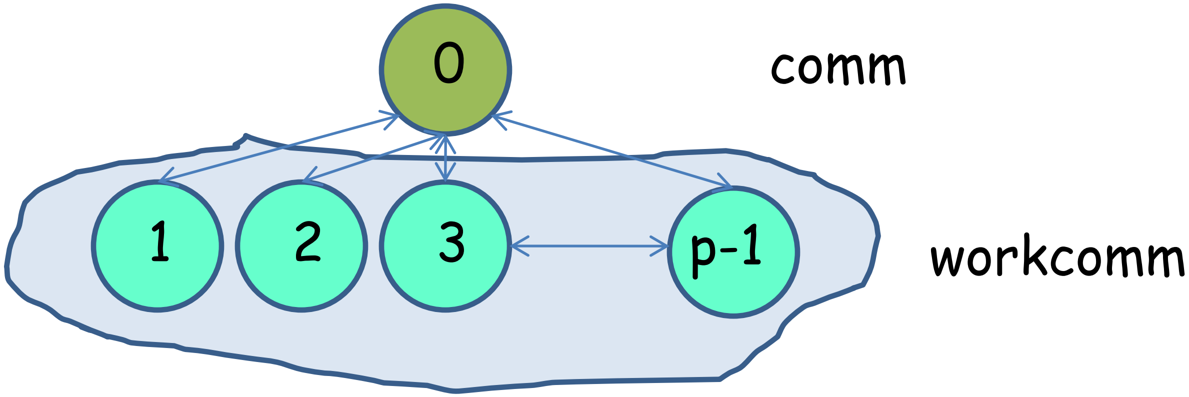
Example: Master-worker (careful: centralized, non-scalable!)



- Master distributes work to individual workers, workers send results/new work to master
- Workers want to synchronize etc. independently of master

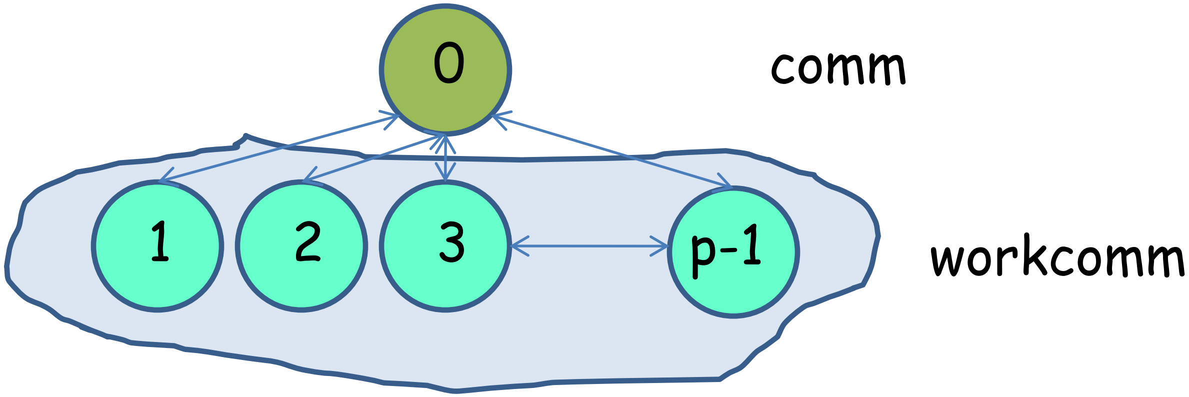
For workers NOT:
MPI_Barrier(comm), MPI_Allgather(comm), …

- master might be away, doing something else: deadlock!



```
MPI_Comm_split(comm,(rank>0 ? 1 : 0),0,&workcomm);
// workcomm on workers (rank>0 in comm): all workers
// workcomm on master (rank==0 in comm): only master
```

MPI_COMM_SELF:
communicator with only process itself, size==1



```
MPI_Comm_group(comm,&group); // get processes in comm
ranklist[0] = 0; // rank 0 to be excluded
MPI_Group_excl(group,1,ranklist,&workgroup); // exclude 0
MPI_Comm_create(comm,workgroup,&workcomm);
// rank 0 (in comm) not in workgroup
// workcomm==MPI_COMM_NULL for rank 0 in comm
// rank!=0 in workcomm
```

Communicator object maintains (for each process) the list of processes in the communicator in rank order: the group

Communicator:
a distributed, global object, can be manipulated through collective operations (MPI_Comm_split, MPI_Comm_dup, …)

Process group (MPI_Group):
local object, ordered set of processes, can be manipulated locally by a process

- `MPI_Group_union, MPI_Group_intersection`
- `MPI_Group_incl, MPI_Group_excl`
- `MPI_Group_Translate_ranks`
- `MPI_Group_compare`
- …

Not this lecture

Use:
Building special communicators, one-sided communication

```
MPI_Comm_free(comm);
```

frees created communicator comm

Note: MPI_COMM_WORLD and MPI COMM_SELF cannot be freed

Good MPI practice:
Free any allocated MPI object after use (communicator, window, datatype, …)

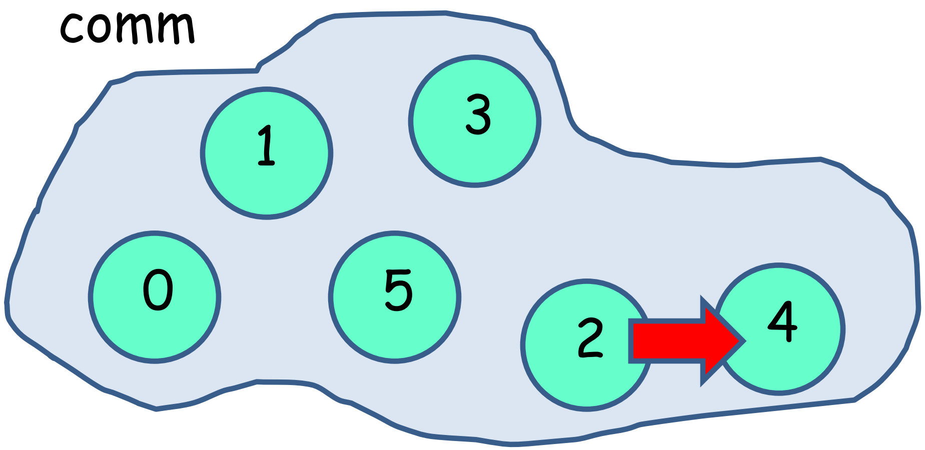
Communicators, summary

Predefined communicators:
- MPI_COMM_WORLD: all started processes
- MPI_COMM_SELF: singleton communicator for each process, only this process

A communicator is a static object, cannot change (processes coming and going); instead new communicators can be created from old:
- MPI_Comm_split
- MPI_Comm_create (+ MPI process groups)

Free after use:
- MPI_Comm_free

Point-to-point communication



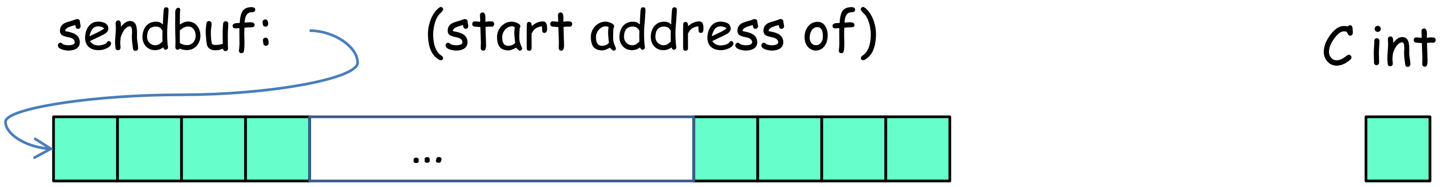
"Process 2 needs to send 500 integers to process 4 (in comm)"

```
int THISMSG=777; // the message TAG (integer type)
int count = 500;
if (rank==2) {
  int sendbuf[500] = {<the data>};
  MPI_Send(sendbuf,count,MPI_INT,4,THISMSG,comm);
} else if (rank==4) {
  int recvbuf[600]; // at least as large as message count
  MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,&status);
}
```

```
MPI_Send(sendbuf,count,datatype,dest,tag,comm);
```

```
int sendbuf[500] = {<the data>};
count = 500;

MPI_Send(sendbuf,count,MPI_INT,4,THISMSG,comm);
```

"Get message called THISMSG (int) stored in array sendbuf of 500 consecutive integers on the road to rank 4 in comm"



sendbuf: (start address of)

C int

Described by datatype MPI_INT

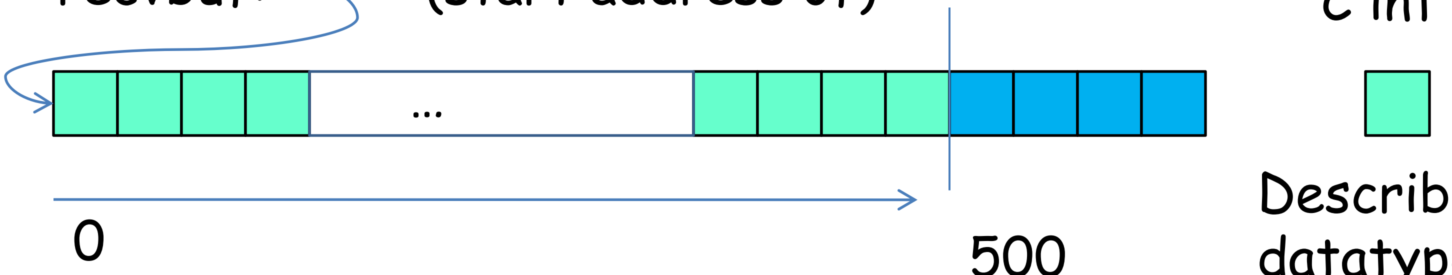
Only rank 4 in comm can ever receive this message

```
MPI_Recv(recvbuf,count,datatype,source,tag,comm,status);
```

```
int recvbuf[600]; // large enough
count = 600; // equal or larger to what is being sent
ok =
MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,&status);
```

"Start reception of message called THISMSG (int) from rank 2 in comm, store result in recvbuf, at most 600 consecutive integers (otherwise ok==MPI_ERR_TRUNCATE)

recvbuf: (start address of)

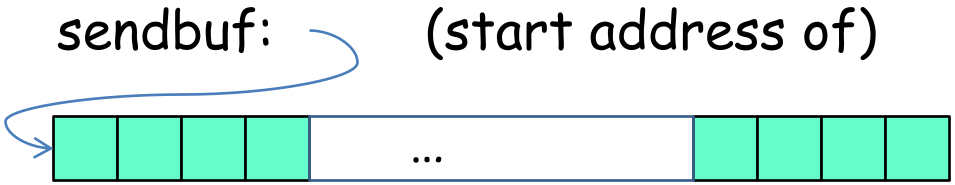


C int

Described by datatype MPI_INT

```
int sendbuf[500] = {<the data>};
count = 500;

MPI_Send(sendbuf,count,MPI_INT,4,THISMSG,comm);
sendbuf[27] = somenewdata; // setup for next operation
```

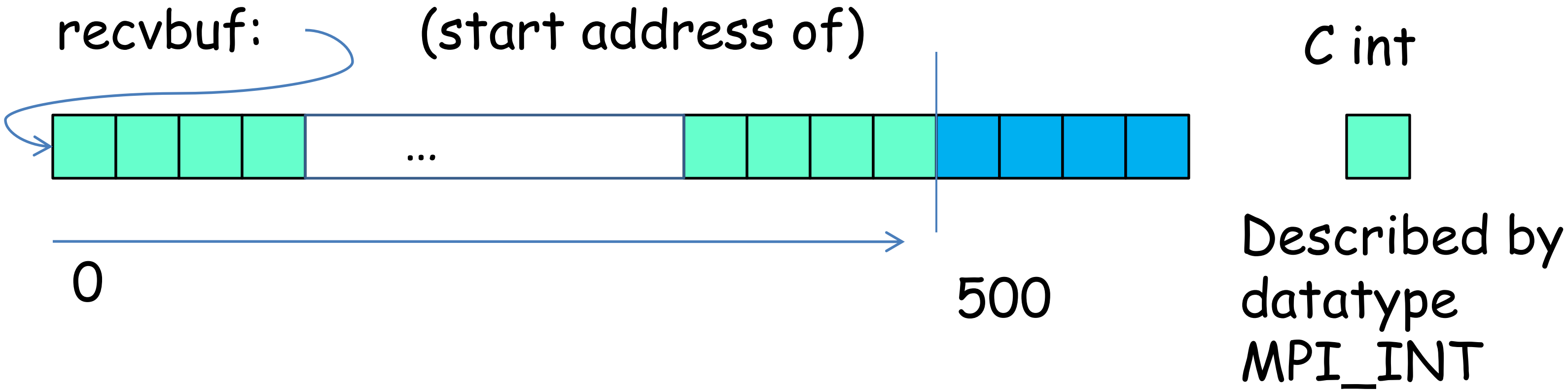
Call returns when it is safe to reuse sendbuf, all data have been taken care of – nothing guaranteed about what has happened on rank 4 (message received or not)



sendbuf: (start address of)

```
int recvbuf[600]; // large enough
count = 600; // equal or larger to what is being sent

MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,&status);
```

Returns when a message from rank 2 has been received; information about data in status object. Forever, if nothing is sent from 2!!



recvbuf: (start address of)

...

0

500

C int

Described by datatype MPI_INT

Status object (half opaque): information on communication

```
MPI_Status status;  // status handle
MPI_Recv(…,&status);
```

Status contains information on what was received:

Fixed fields in C:
status.MPI_SOURCE:
status.MPI_TAG
status.MPI_ERROR

Why?
Don't we
know this??

Fixed fields in FORTRAN:
Status(MPI_SOURCE)
Status(MPI_TAG)
Status(MPI_ERROR)

Status object (half opaque): information on communication

```
MPI_Status status;  // status handle
MPI_Recv(…,&status);
```

Status contains information on what was received:

Fixed fields in C:
status.MPI_SOURCE:
status.MPI_TAG
status.MPI_ERROR

Why?
Don't we
know this??

Fixed fields in FORTRAN:
Status(MPI_SOURCE)
Status(MPI_TAG)
Status(MPI_ERROR)

If so:
Consider
MPI_STATUS_IGNORE as
status argument in MPI_Recv

Status object (half opaque): information on communication

```
MPI_Get_count(status,datatype,count);
```

Returns (in count argument) number of "full datatypes" received; datatype equivalent to type used in receive call

```
MPI_Get_elements(status,datatype,count);
```

Returns (in count argument) number of basic elements received; datatype equivalent to type used in receive call

Note: with basic datatypes (MPI_INT etc.): same

Point-to-point communication succeeds if

1. Sender specifies a valid rank within communicator (0≤rank<size) – and a valid (allocated) send buffer!!
2. A receive with a matching source rank and tag is eventually posted on the same communicator
3. The amount of data sent is smaller or equal to the amount to be received (note: collectives have a different rule)
4. The type signature of the data sent match the type signature of the data to be received

Comments:

1. Mistakes normally caught by MPI_Send – error (abort)!
2. If not, deadlock
3. Otherwise, MPI_ERR_TRUNCATE or memory corruption (big trouble) at receiver!
4. MPI_INT matches MPI_INT, and so forth – see later – but this is rarely checked/enforced, be careful

Message in transit identified by “envelope”:

- Communicator (represented by unique, internal, non-accessible communication context identifier)
- Source (implicit)
- Destination
- Tag
- Other type information (header, part of message, error, …)

Implementation details; „envelope“ not accessible to application

```
MPI_Send(…,rank,tag, comm)
```

is determinate, message is always send to a specific rank (in comm) with a specific tag

```
MPI_Recv(…, rank/ANY,tag/ANY,comm,status)
```

receives from specific rank or non-determined (ANY) rank, with specific or non-dertermined (ANY) tag

Rule:
All messages sent must be received (*)

`MPI_Finalize();` may not terminate (deadlock) if there are pending communications (MPI_Send calls not matched by MPI_Recv)

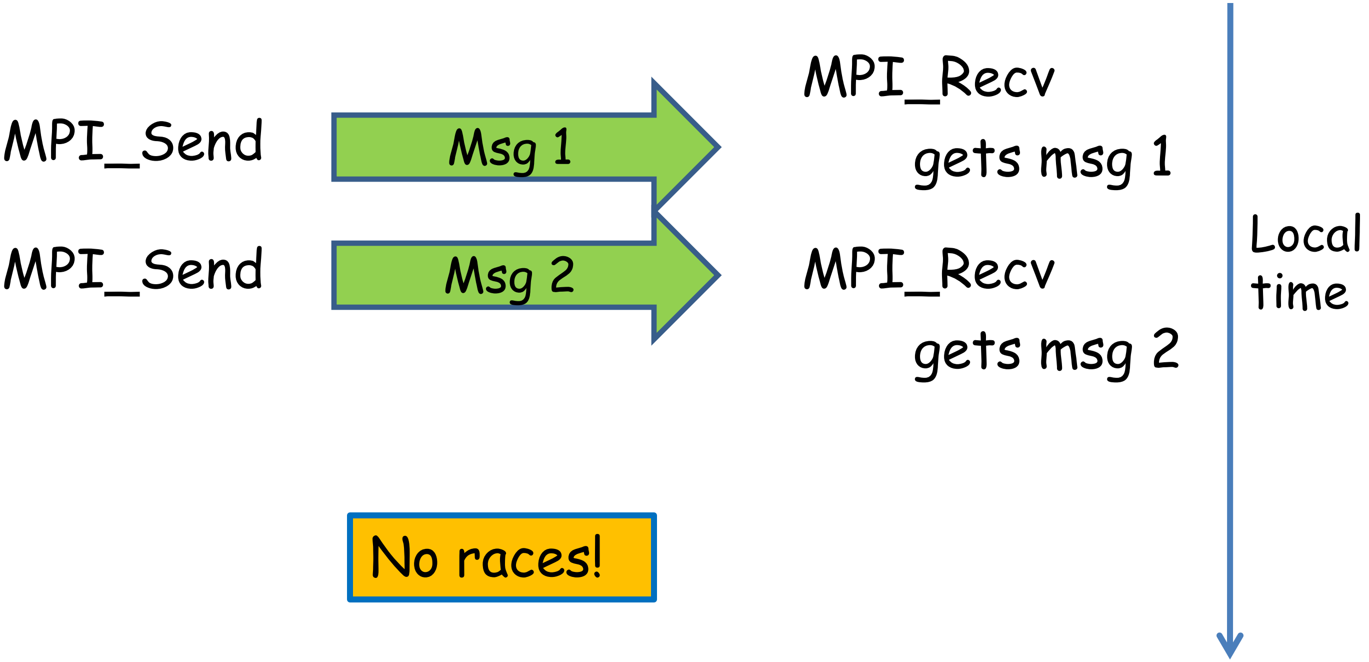
(* unless cancelled, but do not rely on this)

Not in this lecture

Reasoning about point-to-point communication

Deterministic – messages are non-overtaking (ALWAYS):

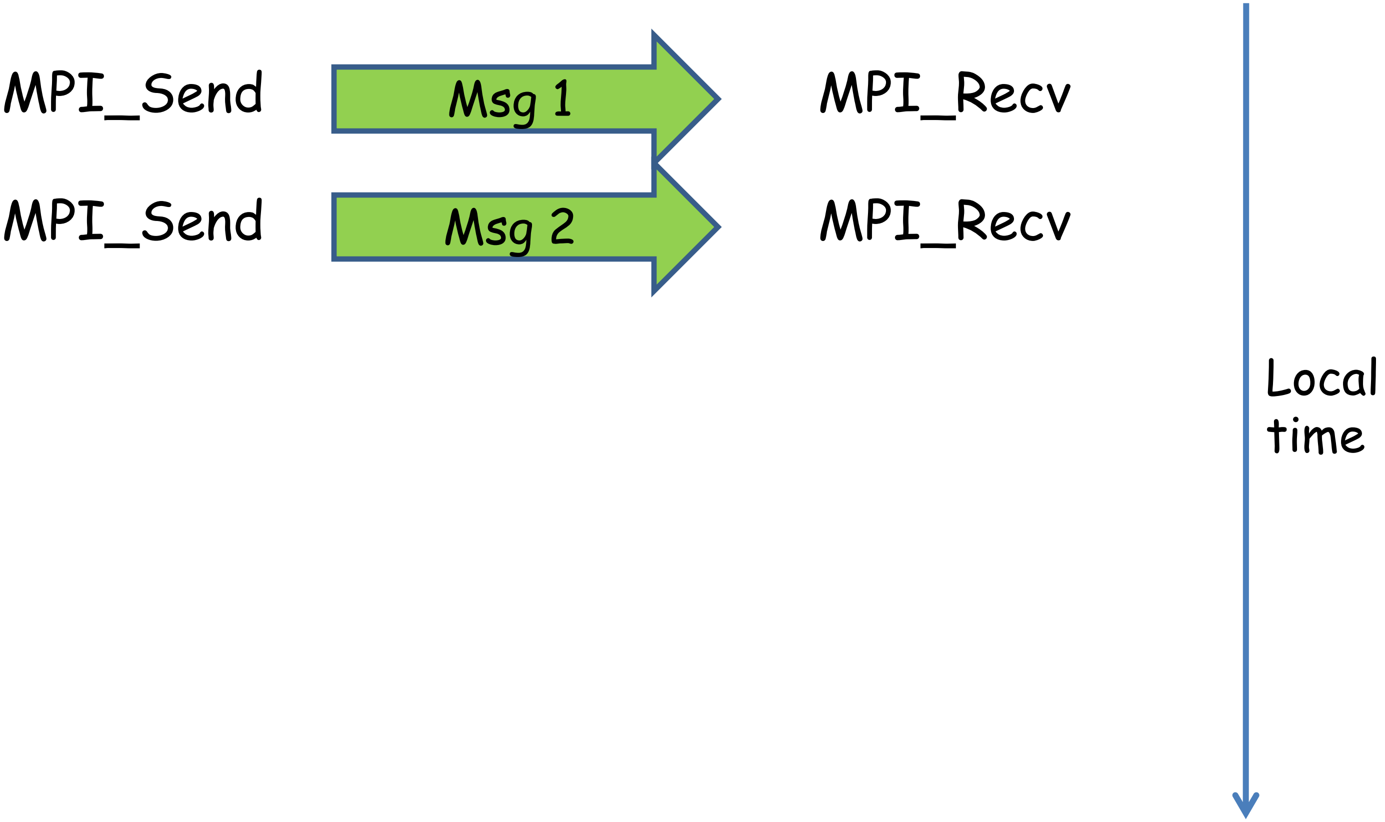
Messages sent with the same destination (rank) and the same tag arrive in sent order at destination



No races!

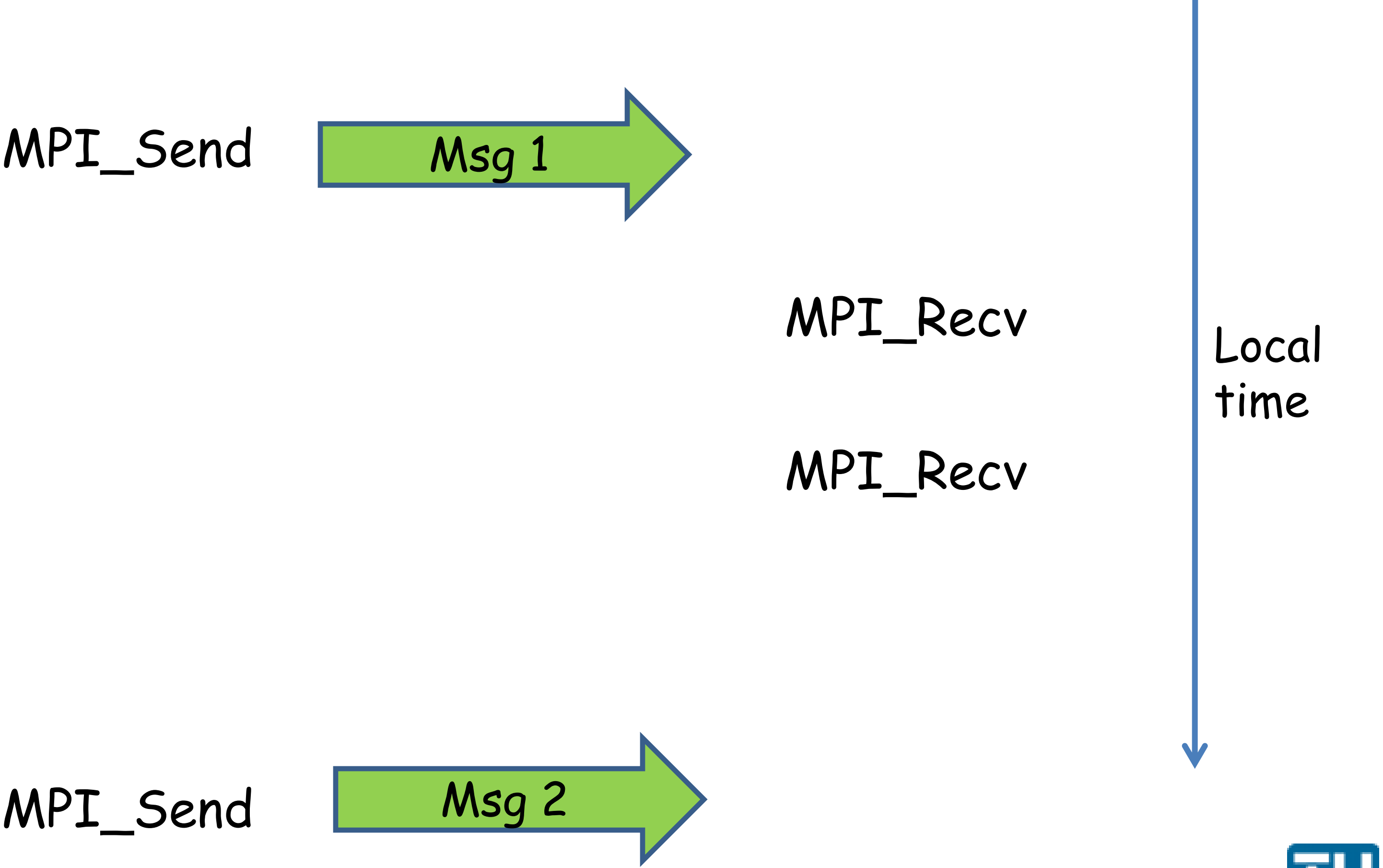
Message Passing Abstraction (reminder)

No global time, processes are not synchronized



Message Passing Abstraction (reminder)

In reality, processes not synchronized, may do different work



MPI_Send

Msg 1

MPI_Recv

MPI_Recv

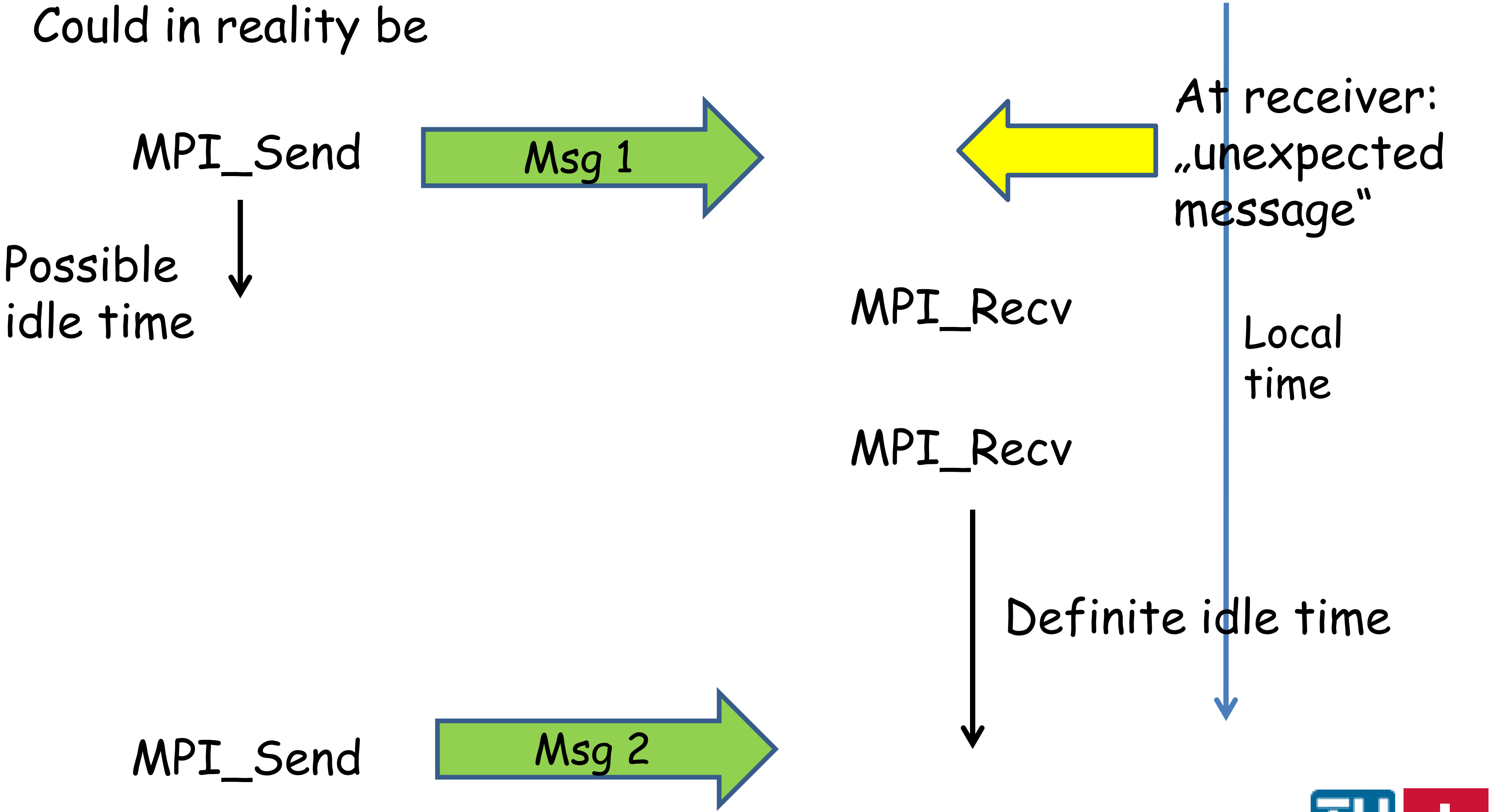
Local time

MPI_Send

Msg 2

Message Passing Abstraction (reminder)

Could in reality be



Sources of non-determinism (1)

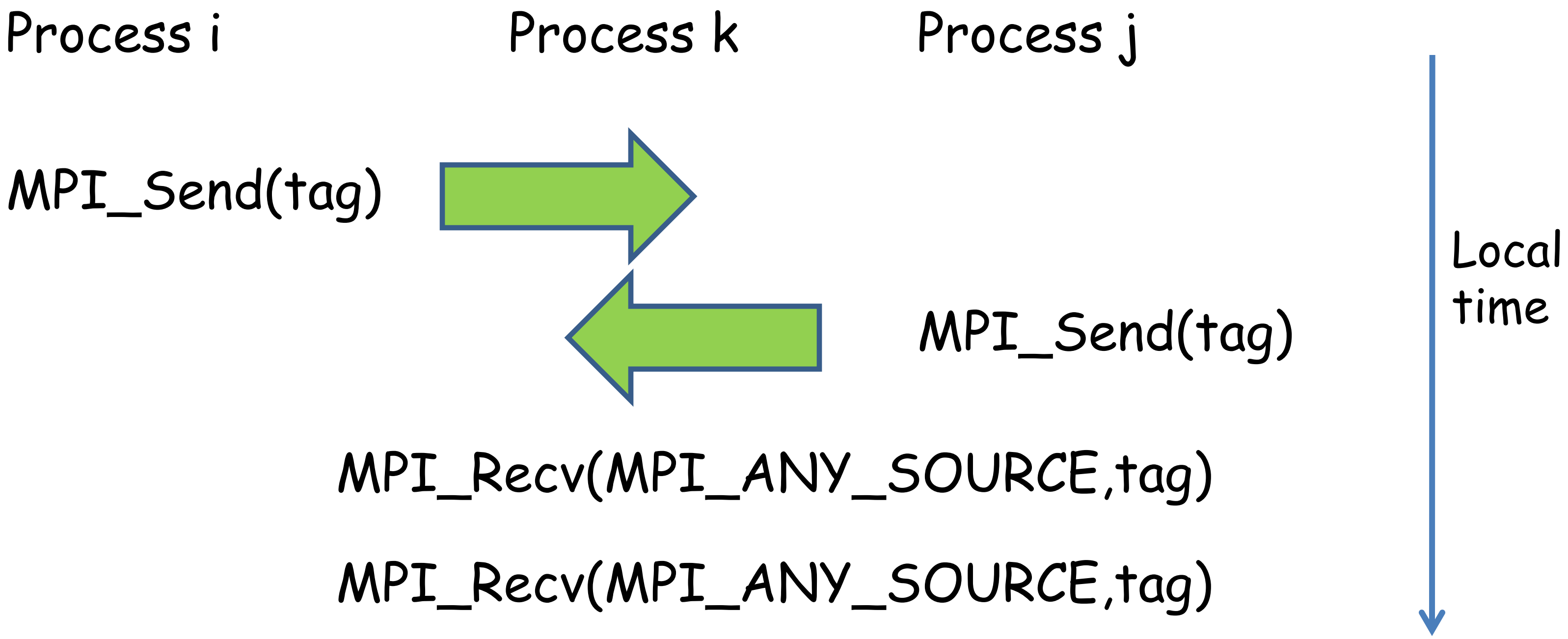
```
MPI_Recv(recvbuf,…,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,
         &status);
```

Wildcards:

- Receive some (ANY) message from somewhere (ANY, but within comm)
- Now, need to check status to find out source and tag!

Message ordering is still guaranteed (non-overtaking)

Sources of non-determinism (1)



Process i

Process k

Process j

MPI_Send(tag)

MPI_Send(tag)

MPI_Recv(MPI_ANY_SOURCE,tag)

MPI_Recv(MPI_ANY_SOURCE,tag)

Local time

Either messages may be received first; can cause problems if messages have different count/type

```
MPI_Probe(source,tag,comm,status);
```

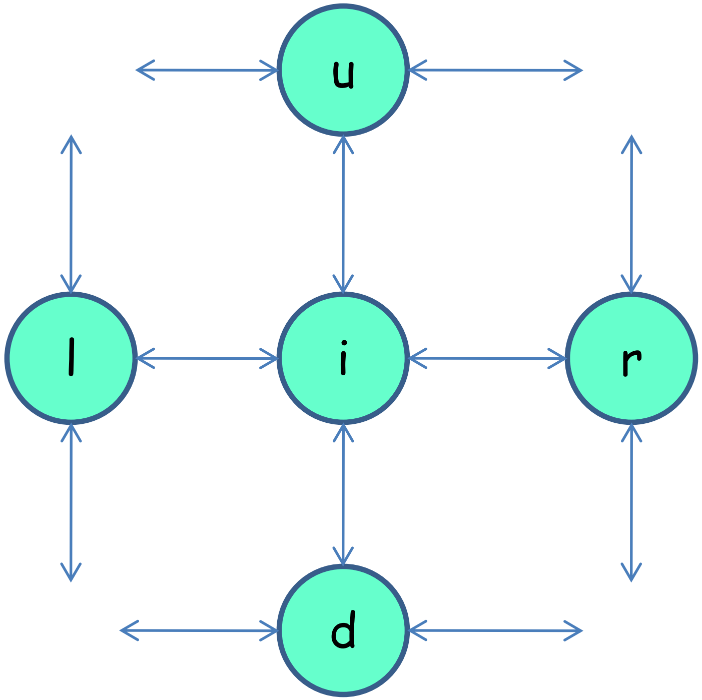
Return when a message with given source (or MPI_ANY_SOURCE) and tag (or MPI_ANY_TAG) in comm is ready for reception; count for message in status

After probe: receive message with MPI_Recv(buffer, count,…)

Advanced note: this can cause problems in multi-threaded MPI applications

Send semantics

Example: solution of Poisson PED by Jacobin method



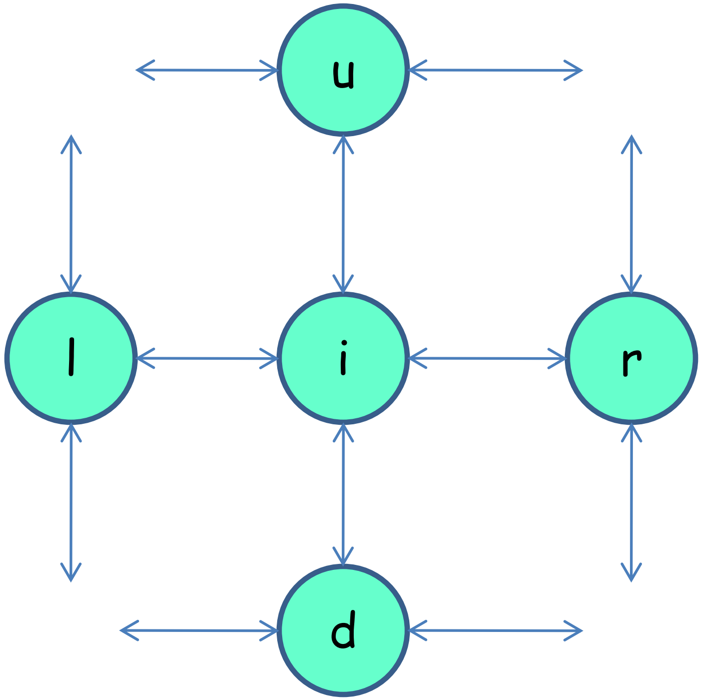
u[m,n]

$$\begin{pmatrix} & u[i-1,j] & \\ u[i,j-1] & u[i,j] & u[i,j+1] \\ & u[i+i,j] & \end{pmatrix}$$

Special conditions on borders, i=0, …

For all 0≤i<m, 0≤j<n, update
u[i,j] <- ¼(u[i,j-1]+u[i,j+1]+u[i-1,j]+u[i+1,j]-h^2f(i,j))

Send semantics



```
MPI_Send(up);
MPI_Send(down);
MPI_Send(left);
MPI_Send(right);

MPI_Recv(up);
MPI_Recv(down);
MPI_Recv(left);
MPI_Recv(right);
```

most likely deadlocks!

MPI_Send(sendbuf,…,rank,tag,comm);

starts sending a message – completion may depend on what receiver does; buffering not enforced by MPI standard

non-local completion semantics

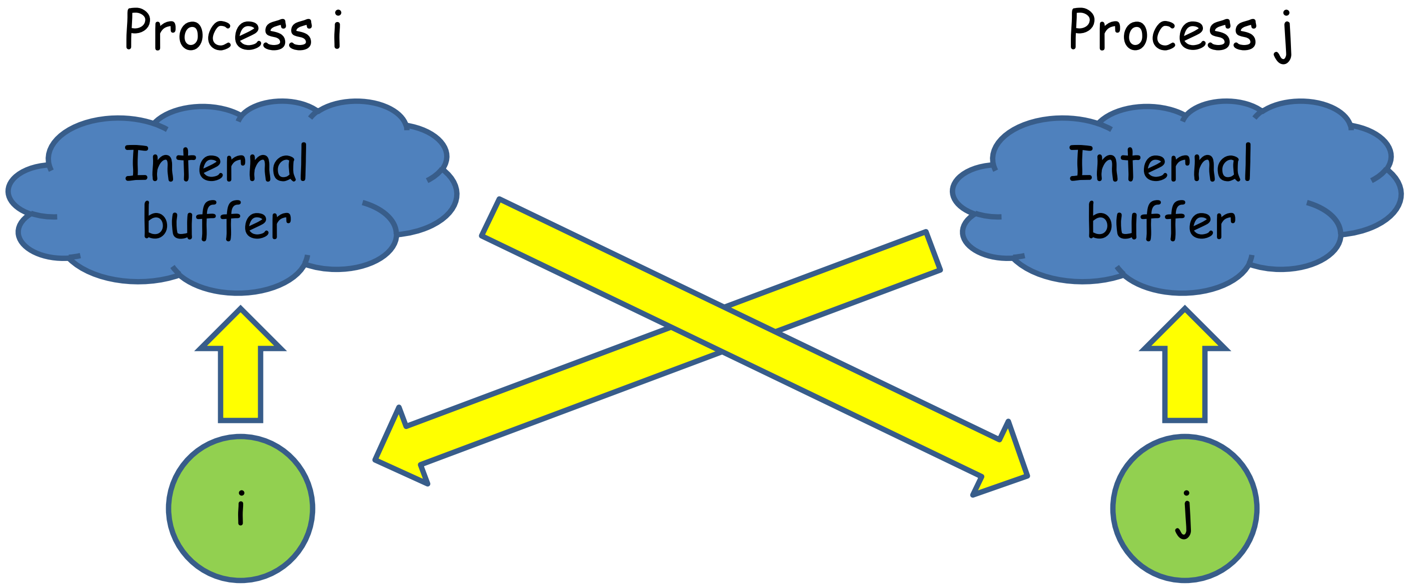
Blocking: returns when sendbuf can be reused

Freedom for MPI implementers:

- Short messages: usually just sent to some fixed address at receiver (to be processed later)
- Medium sized messages: may be buffered locally, and sent when receive has been posted (acknowledgement from receiving process)
- Long messages: participation of receiving process needed

Exact conditions of local-completion are MPI implementation dependent!

Template MPI_Send implementation, short messages

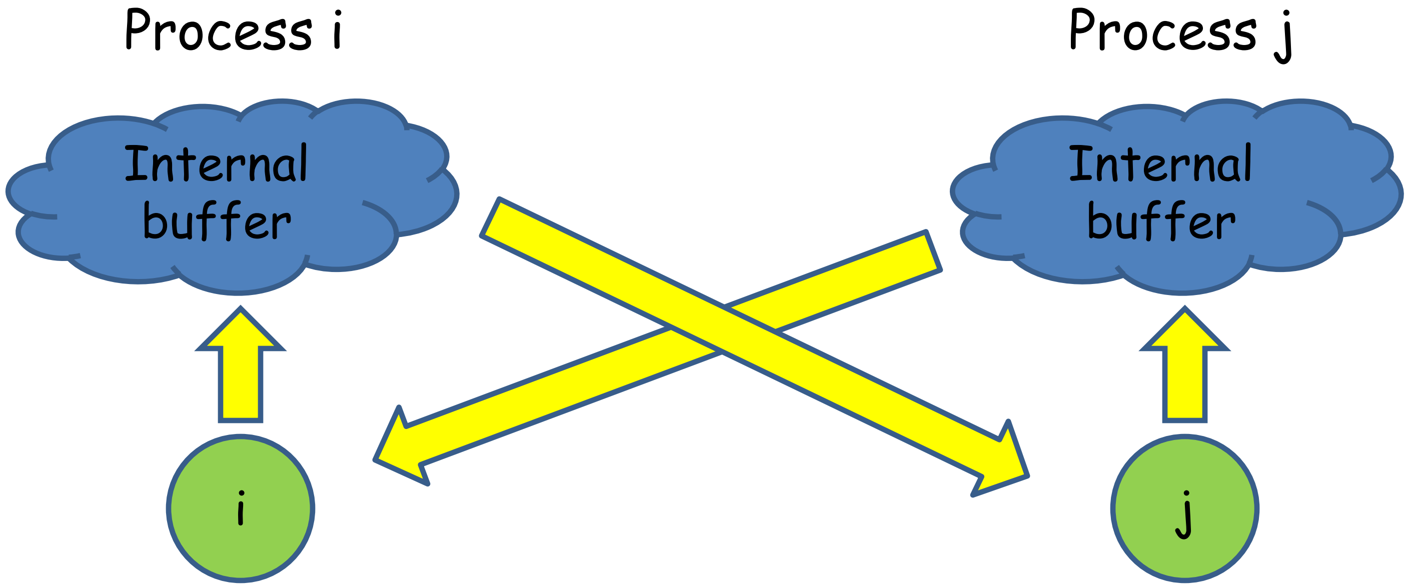


```
MPI_Send(buffer, …,j, …);
MPI_Recv(buffer, …, j, …);
```

```
MPI_Send(buffer, …,i, …);
MPI_Recv(buffer, …, i, …);
```

Succeeds if internal buffer is large enough. MPI does not require internal buffering

Template MPI_Send implementation, short messages



MPI_Send(buffer, …,j, …);

MPI_Recv(buffer, …, j, …);

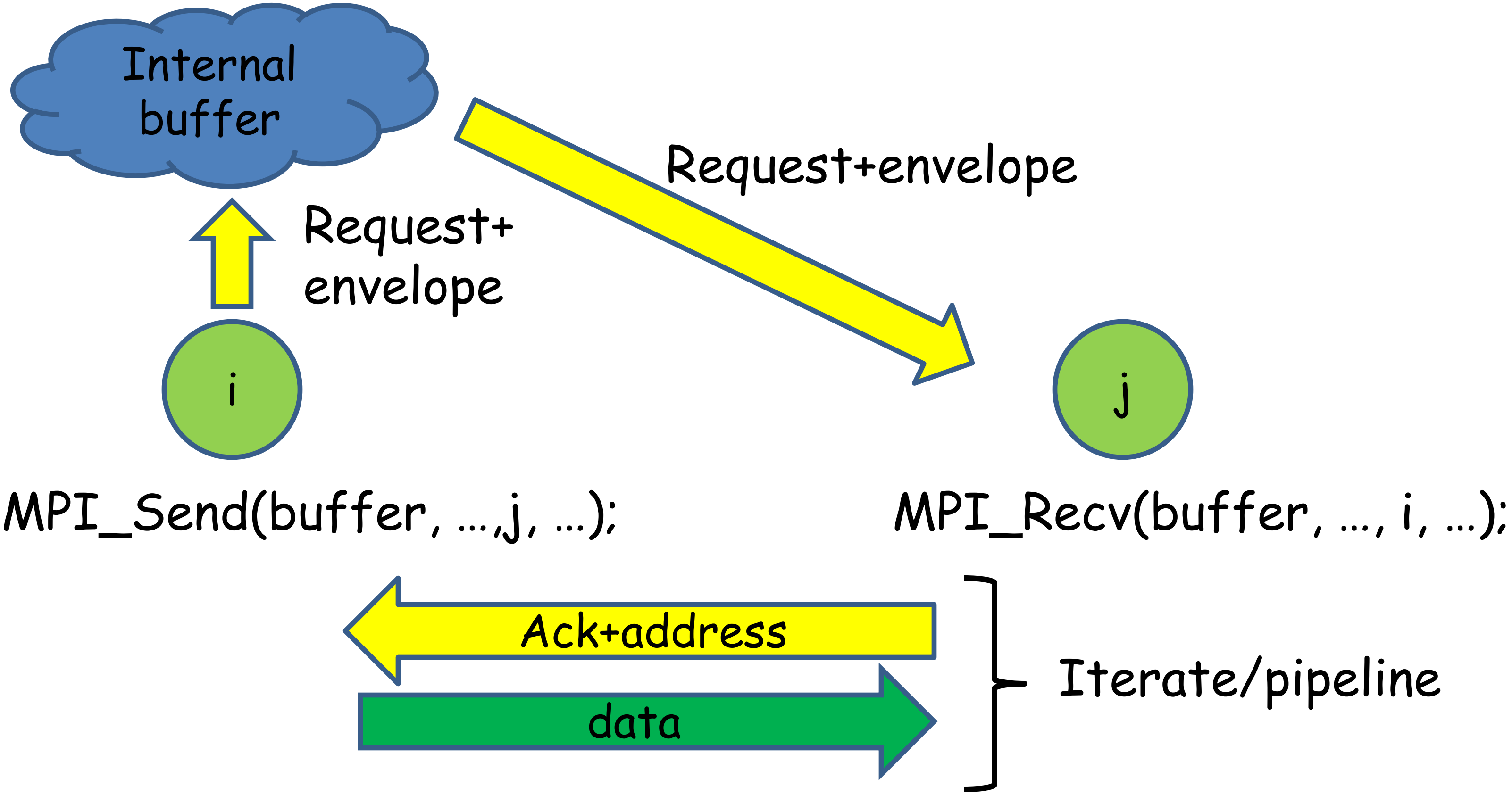
MPI_Send(buffer, …,i, …);

MPI_Recv(buffer, …, i, …);

Drawback: Extra copy – costly for large buffers

MPI design principle:
library should not allocate unbounded buffers

Template MPI_Send implementation, long messages



Internal buffer

Request+ envelope

Request+envelope

i

j

MPI_Send(buffer, …,j, …);

MPI_Recv(buffer, …, i, …);

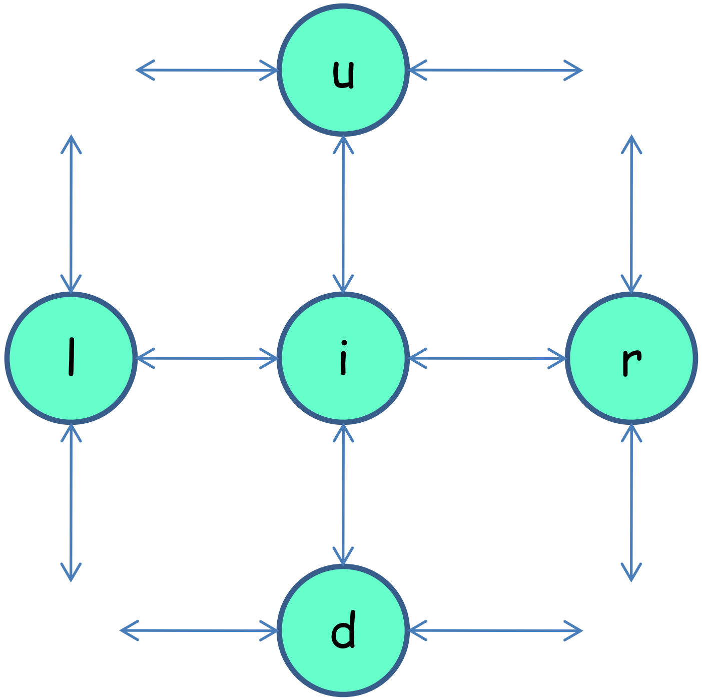
Ack+address

data

Iterate/pipeline

Send complete with last data

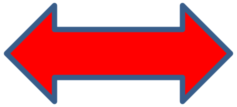
Send semantics (con't)



```
MPI_Send(up);
MPI_Send(down);
MPI_Send(left);
MPI_Send(right);

MPI_Recv(up);
MPI_Recv(down);
MPI_Recv(left);
MPI_Recv(right);
```

Program is unsafe: termination depends on MPI buffering and size of messages; implementation dependent!

Safe(r) programming



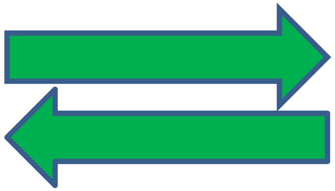
Process 0

```
MPI_Send
MPI_Recv
```

Process 1

```
MPI_Send
MPI_Recv
```

Unsafe, saved by scheduling – sometimes difficult



Process 0

```
MPI_Send
MPI_Recv
```

Process 1

```
MPI_Recv
MPI_Send
```

“even-odd” scheduling… (general: communication graph 1-factoring)

Safe(r) programming

```
Process 0          Process 1

MPI_Send           MPI_Send
MPI_Recv           MPI_Recv
```

Unsafe, saved by combined send-receive

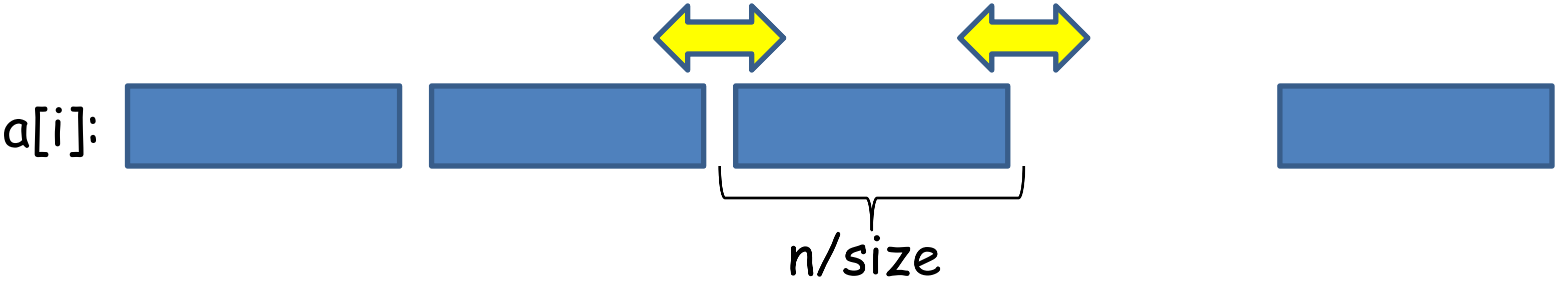
```
Process 0               Process 1

MPI_Sendrecv    ⟷      MPI_Sendrecv
```

```
MPI_Sendrecv(sendbuf,sendcount,sendtype,dest,sendtag,
             recvbuf,recvcount,recvtype,source,recvtag,
             comm,status);
```

Combined send-receive operation.

Note: `sendbuf` and `recvbuf` must be disjoint

Performance advantage:
can possibly better utilize bidirectional communication network (system dependent)

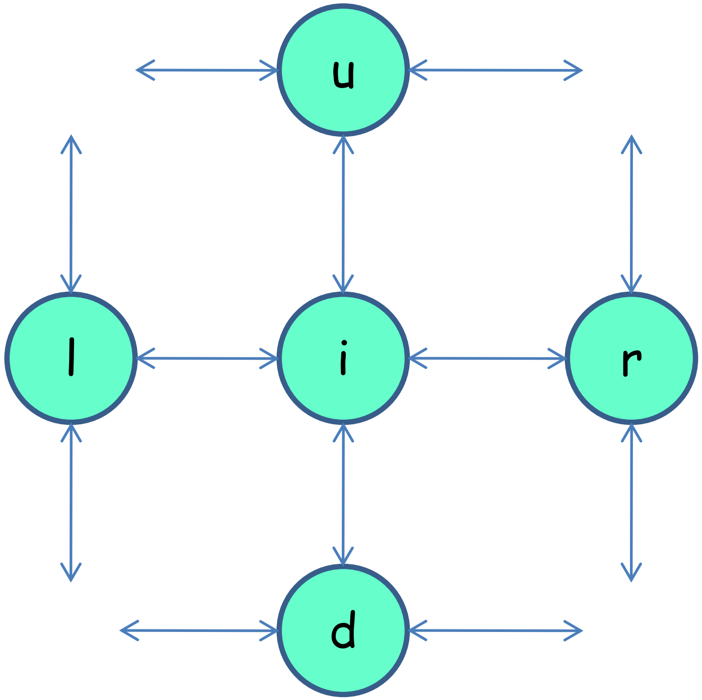


```
MPI_Sendrecv(…,rank-1,…,rank+1,…);
MPI_Sendrecv(…,rank+1,…,rank-1,…);
```

Exercise:
Implement and compare to other solutions

Parallel Computing

Safe programming – non-blocking communication



```
MPI_Isend(up,&req[0]);
MPI_Isend(down,&req[1]);
MPI_Isend(left,&req[2]);
MPI_Isend(right,&req[3]);

MPI_Irecv(up,req[4]);
MPI_Irecv(down,&req[5]);
MPI_Irecv(left,&req[6]);
MPI_Irecv(right,&req[7]);

MPI_Waitall(8,req,stats);
```

Safe: I(mmediate) operations have local completion semantics

```
MPI_Request request;
MPI_Isend(sendbuf,…,comm,request);
```

starts ("posts") send operation, returns immediately – local completion semantics, independent of receiving side – sendbuf should NOT be modified before operation is complete

"progress" information in request object:

```
MPI_Test(request,flag,status);
```

If flag==1 operation has completed, status set

```
MPI_Wait(request,status);
```

Wait; return when operation has completed, status set

```
 MPI_Isend(sendbuf,…,comm,&request);
 MPI_Wait(request,&status);
```

equivalent to `MPI_Send(sendbuf,…,comm);`

Note:
Again, semantics is non-local; sendbuf can be reused, receiver may or may not have started

Note:
for non-blocking send operations, status is undefined, except for MPI_ERROR field

Test and completion calls

- MPI_Wait
- MPI_Test

- MPI_Waitall(number,array_of_requests,array_of_statuses)
- MPI_Testall

- MPI_Waitany
- MPI_Testany

- MPI_Waitsome
- MPI_Testsome

For details, see MPI 2.2 Standard

Other send modes – send semantics

Mode		Remark	Semantics
MPI_Send	Standard Returns when sendbuf can be reused		Non-local (potentially)
MPI_Ssend	Synchronous Returns when sendbuf can be reused AND receiver has started reception		Strictly non-local
MPI_Bsend	Buffered, returns immediately, data may be copied into intermediate buffer	Intermediate buffer from user space must have been attached with MPI_Buffer_attach	local
MPI_Rsend	Ready, standard	Precondition: matching receive MUST have been posted	Non-local

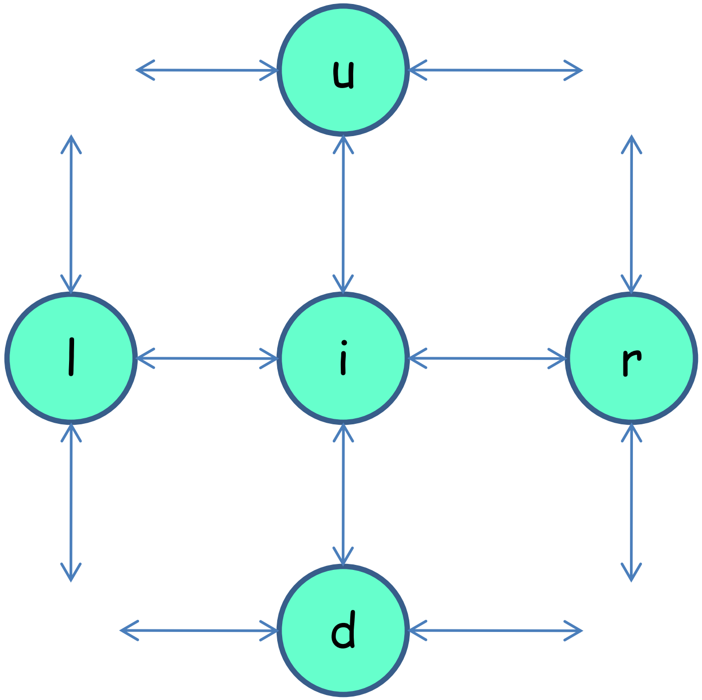
Only one receive mode (blocking and nonblocking)

MPI_Recv/MPI_Irecv

Blocking/non-blocking and modes are orthogonal, and can be arbitrarily combined

Non-blocking operations

Semantic advantages – easier to prevent deadlocks



```
MPI_Isend(up,&req[0]);
MPI_Isend(down,&req[1]);
MPI_Isend(left,&req[2]);
MPI_Isend(right,&req[3]);

MPI_Irecv(up,req[4]);
MPI_Irecv(down,&req[5]);
MPI_Irecv(left,&req[6]);
MPI_Irecv(right,&req[7]);

MPI_Waitall(8,req,stats);
```

Non-blocking operations

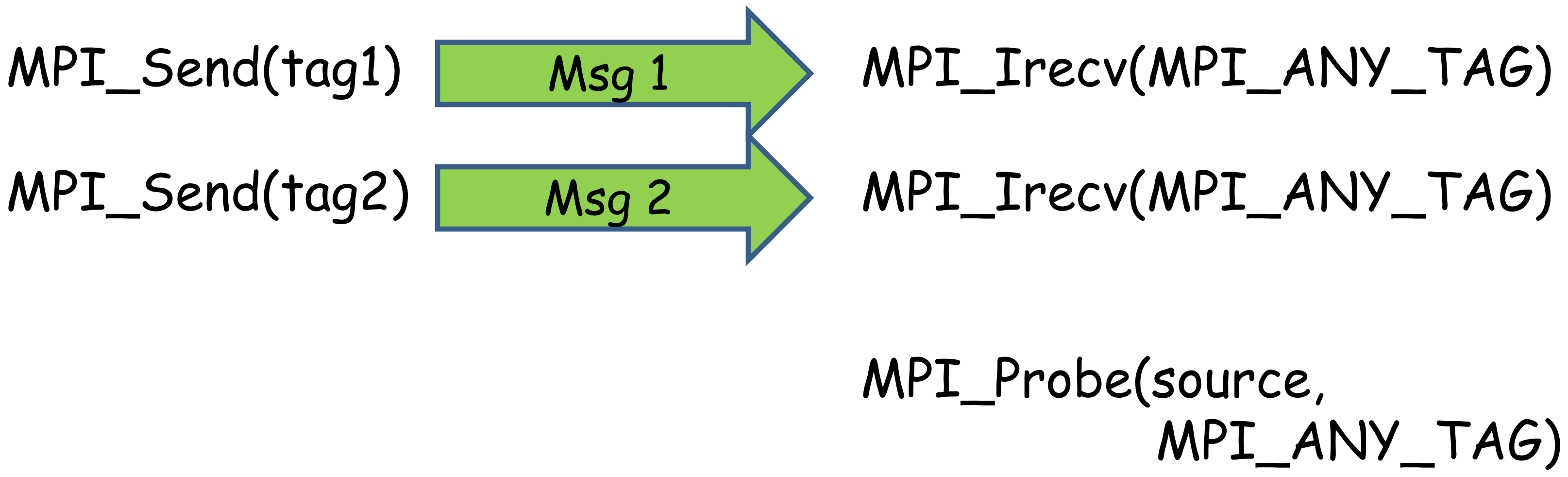
Performance advantages – may be possible to overlap communication with computation (eg. if other process is delayed)

```
          MPI_Isend                             MPI_Isend
          MPI_Irecv                             MPI_Irecv
<Compute>                             <Compute>
          MPI_Wait                              MPI_Wait
          MPI_Wait                              MPI_Wait
```

Note: implementation AND system dependent

Performance note: waiting too long with MPI_Wait call can slow down application (progress)

Sources of non-determinism (2)



MPI_Send(tag1)

MPI_Send(tag2)

MPI_Irecv(MPI_ANY_TAG)

MPI_Irecv(MPI_ANY_TAG)

MPI_Probe(source,
MPI_ANY_TAG)

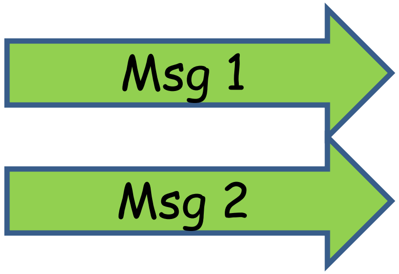
Messages are received in sent-order (tag1, tag2)

Note: MPI_ANY_TAG alone is not a source of non-determinism

Sources of non-determinism (2)

MPI_Send(tag1)

MPI_Send(tag2)



Msg 1

Msg 2

MPI_Irecv(tag2,&req2)

MPI_Irecv(tag1,&req1)

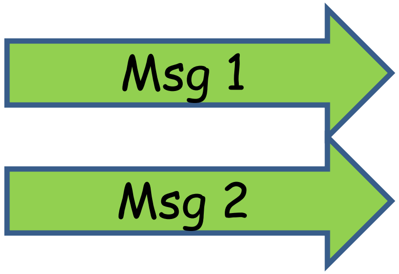
MPI_Wait(req2)

Enforce specific order

Sources of non-determinism (2)

MPI_Send(tag1)

MPI_Send(tag2)



Msg 1

Msg 2

MPI_Irecv(MPI_ANY_TAG)

MPI_Irecv(tag1,&req1)

MPI_Wait(req2)

DEADLOCK!

tag1 has matched MPI_ ANY_TAG

```
MPI_Iprobe(source,tag,comm,flag,status);
```

Non-blocking probe, flag==1 means message with source and tag ready for reception in comm

Point-to-point communication performance rules

Send operations: creating envelope in local buffer, initiating communication (e.g. $\alpha+\beta m$ transfer time)

Latency!

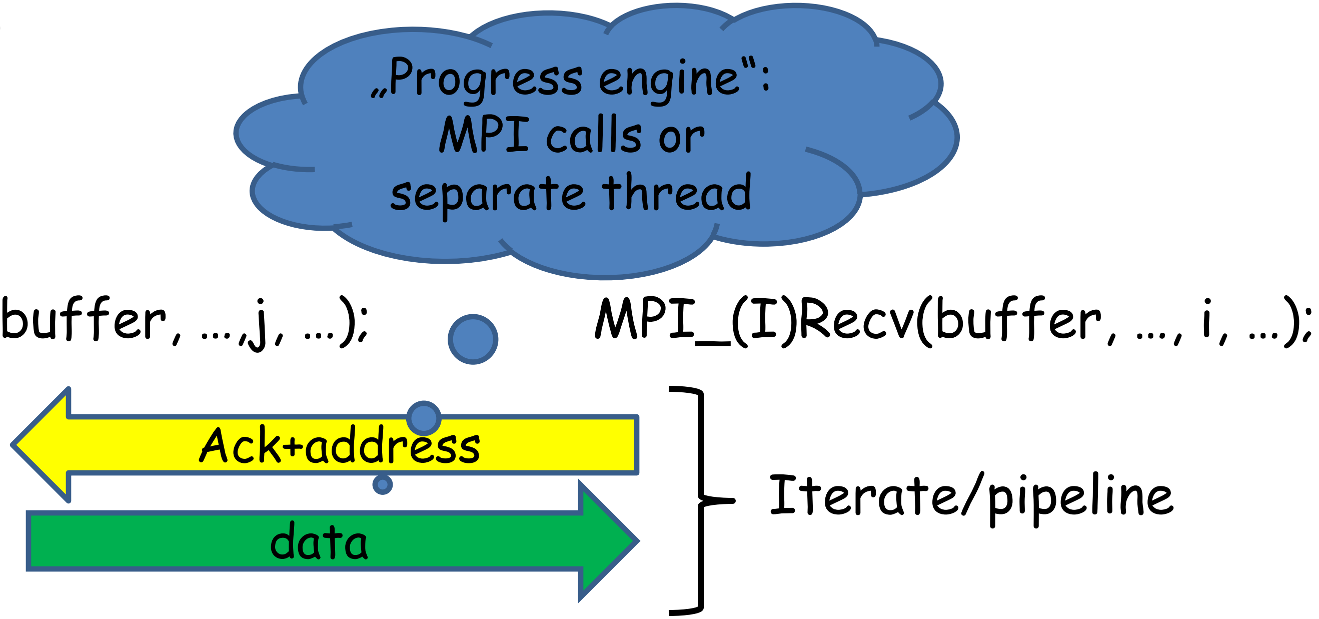
Rule-of-thumb: avoid many small messages, group into fewer, larger

MPI_Send: may or may not have to wait for acknowledgement; can sometimes be faster than other send operations

MPI_Send may (for large messages) depend on activity of receiving process

Point-to-point communication performance rules

MPI_Isend: can return immediately; progress and completion depends on activity of receiver AND often on activity/MPI calls by sender



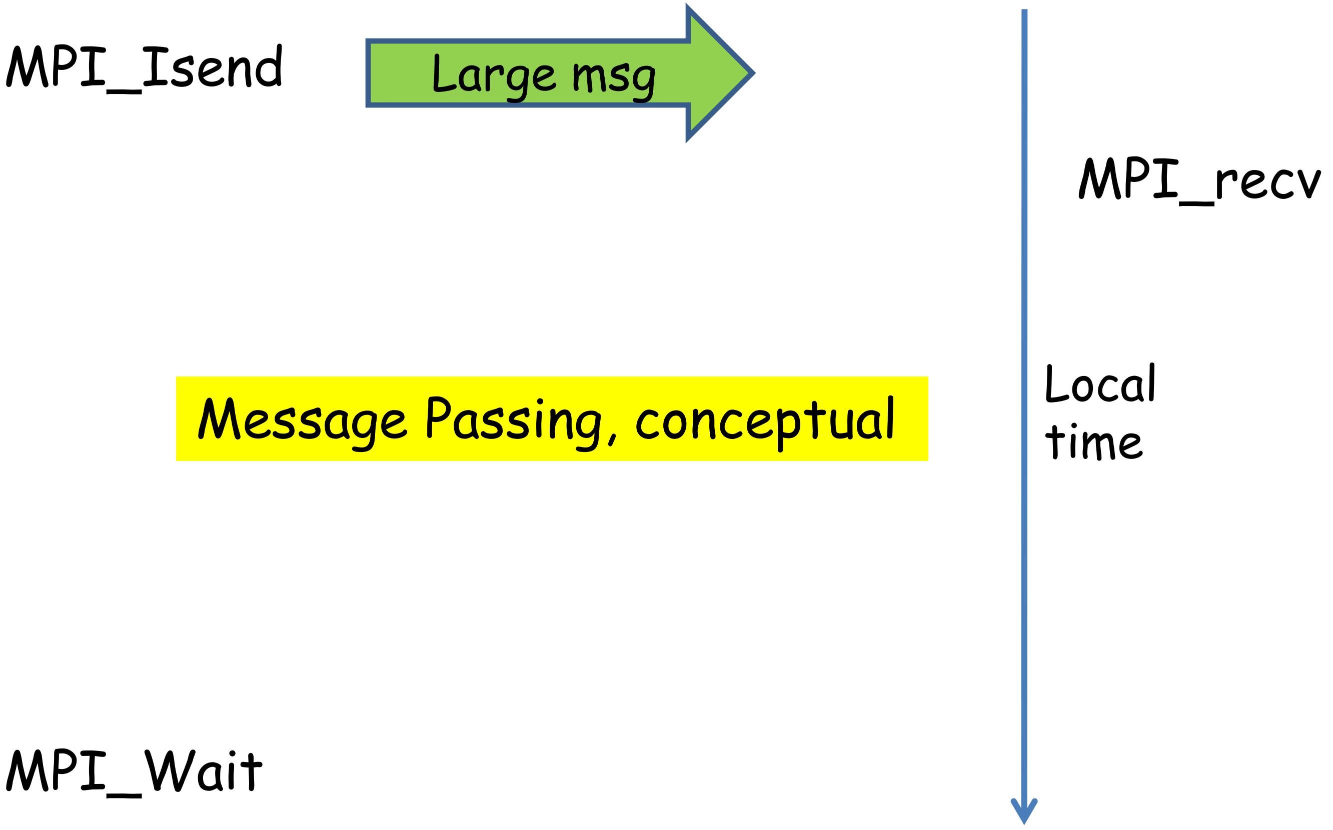
„Progress engine“: MPI calls or separate thread

MPI_(I)Send(buffer, …,j, …);

MPI_(I)Recv(buffer, …, i, …);

Completion of MPI_Send and MPI_Isend does not imply anything about receiving process

A note on progress



MPI_Isend

Large msg

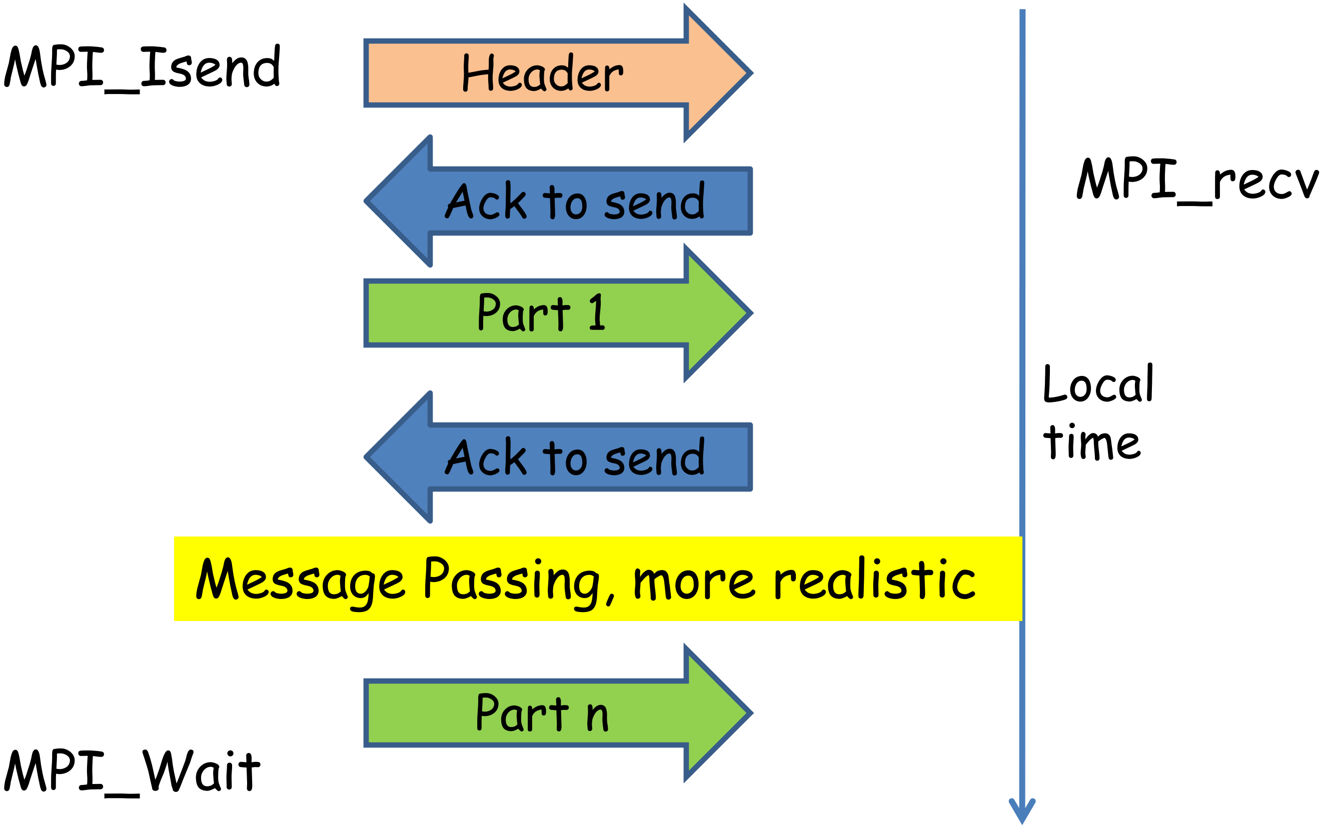
MPI_recv

Local time

Message Passing, conceptual

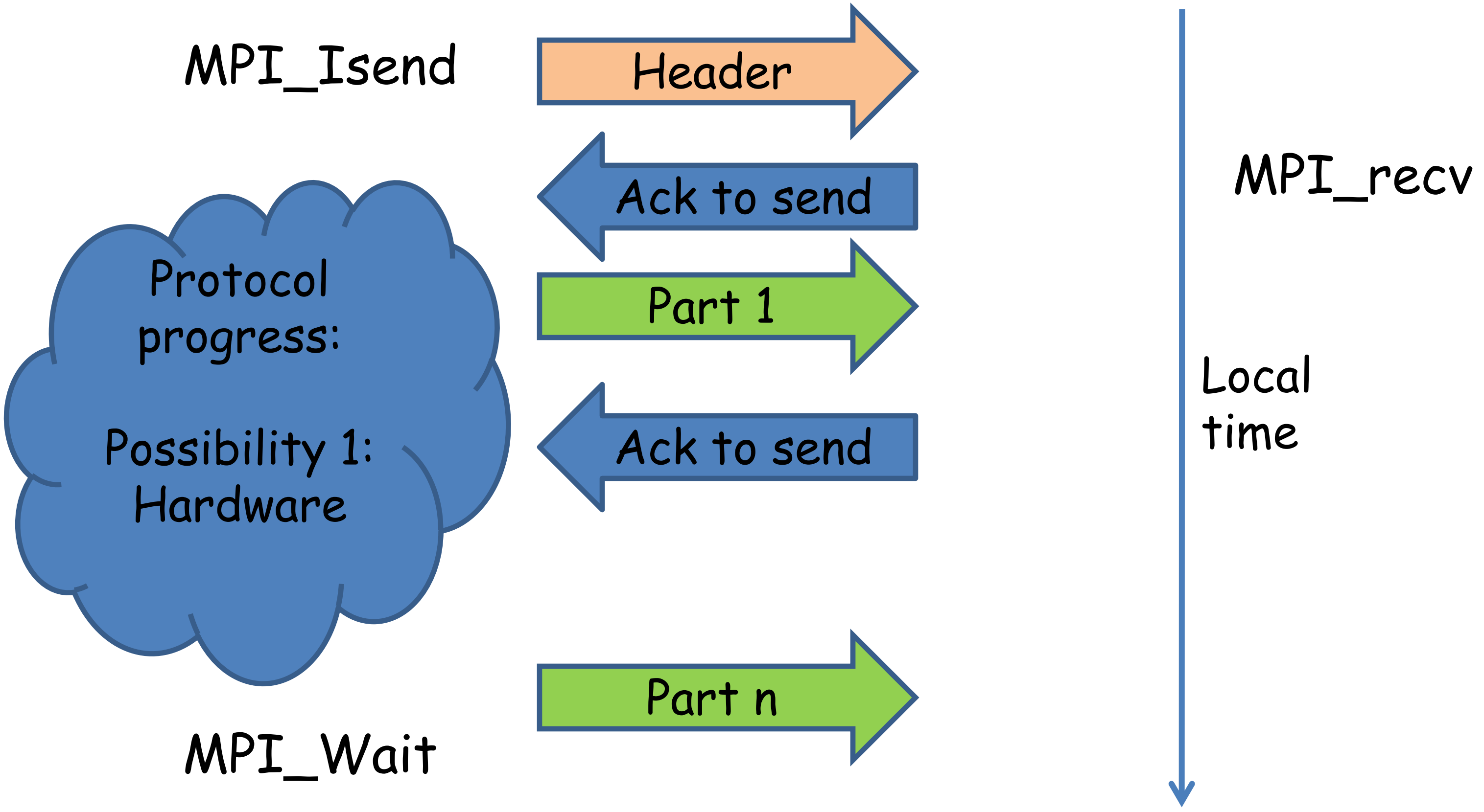
MPI_Wait

A note on progress

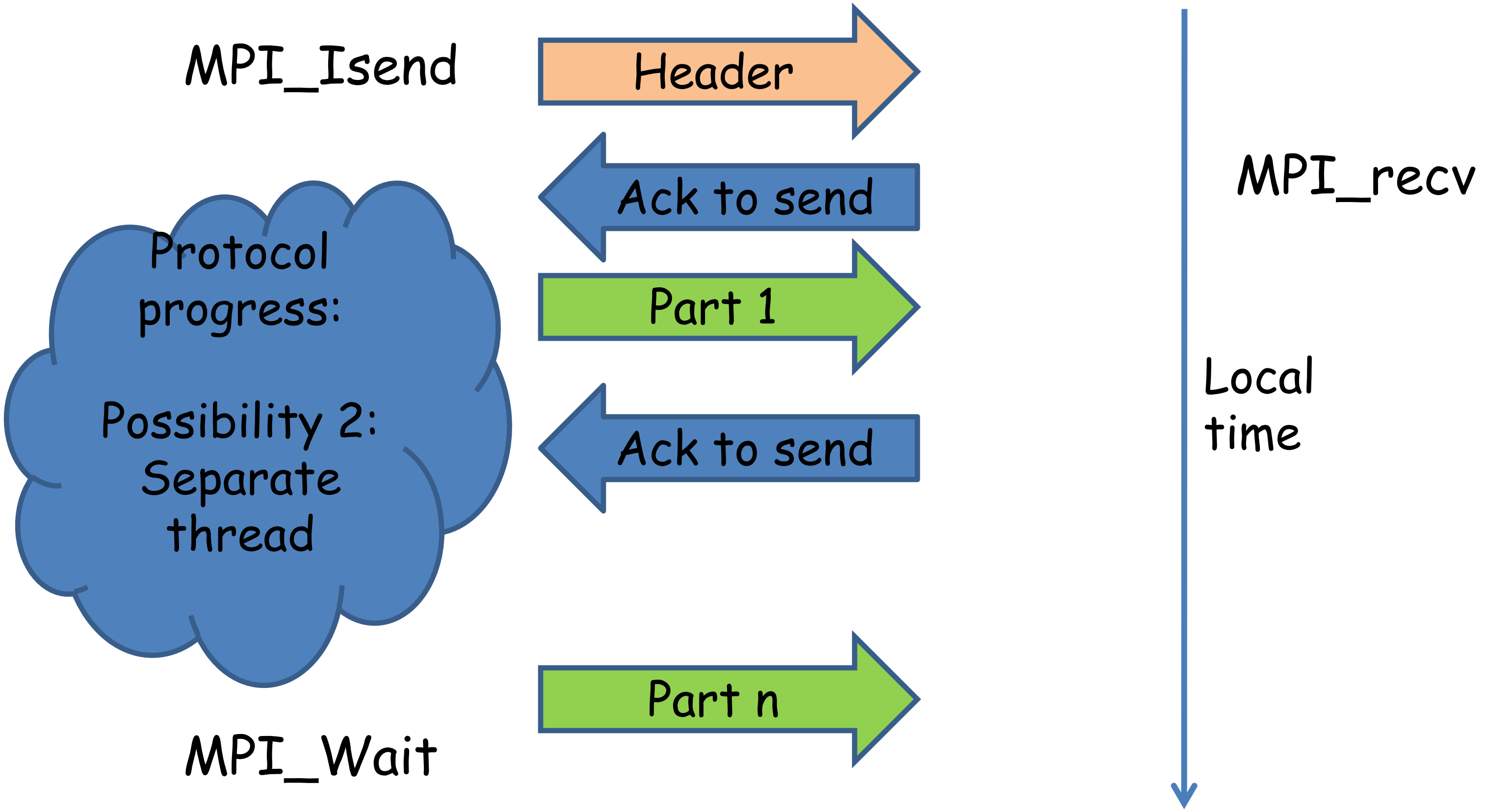


Message Passing, more realistic

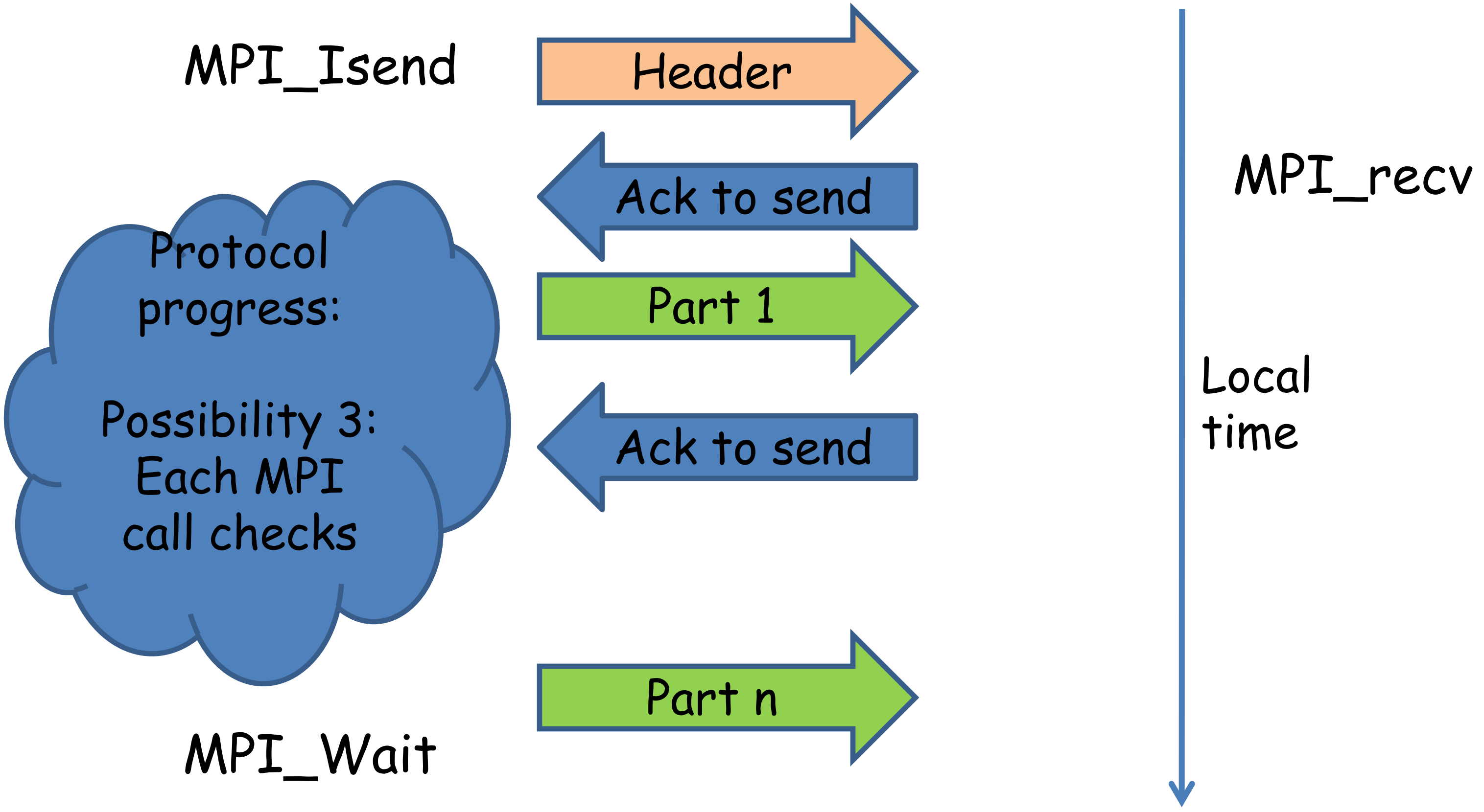
A note on progress



A note on progress



A note on progress



MPI_Isend

Header

Ack to send

Part 1

Ack to send

Part n

MPI_Wait

Protocol progress:

Possibility 3: Each MPI call checks

MPI_recv

Local time

A note on progress

MPI libraries often use mixed strategies:

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

Thread support often considered too expensive for HPC, sometimes not possible

Good practice: frequent MPI calls when using non-blocking operations

Principle: MPI standard is intentionally loose on progress

Point-to-point communication performance rules

MPI_Ssend: synchronous operation, returns when receive call has been posted (MPI_Recv, MPI_Irecv); always incur acknowledgement

MPI_Rsend: only legal when matching receive call has been posted; can save some ack's

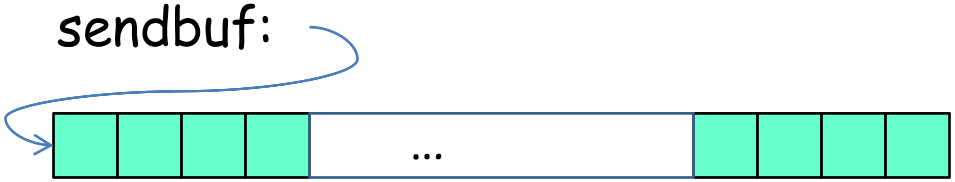
MPI_Bsend: data always copied to intermediate buffer; buffer supplied by user, in user space

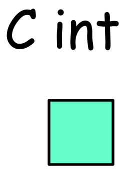
Datatypes, data layouts

```
MPI_Send(sendbuf,count,datatype,dest,tag,comm);
```

```
int sendbuf[500] = {<the data>};
count = 500;

MPI_Send(sendbuf,count,MPI_INT,4,tag,comm);
```

"Get message stored in array sendbuf of 500 consecutive integers on the road to rank 4 in comm"



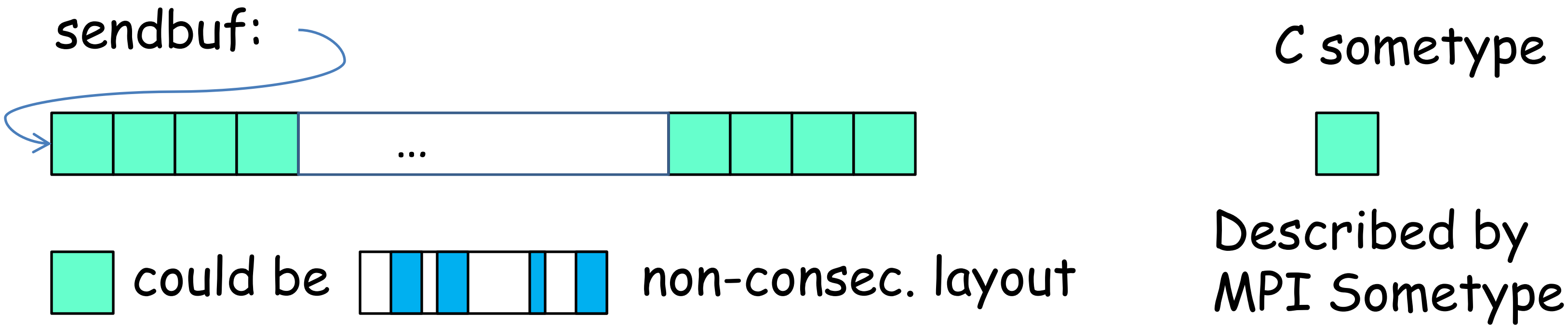


Described by datatype MPI_INT

```
MPI_Send(sendbuf,count,datatype,dest,tag,comm);
```

```
sometype *sendbuf;
sendbuf = malloc(count*sizeof(sometype));

MPI_Send(sendbuf,count,Sometype,dest,tag,comm);
```

"Get message stored in array sendbuf of count consecutive sometype's on the road to dest in comm"



sendbuf:

C sometype

🟩 could be non-consec. layout

Described by MPI Sometype

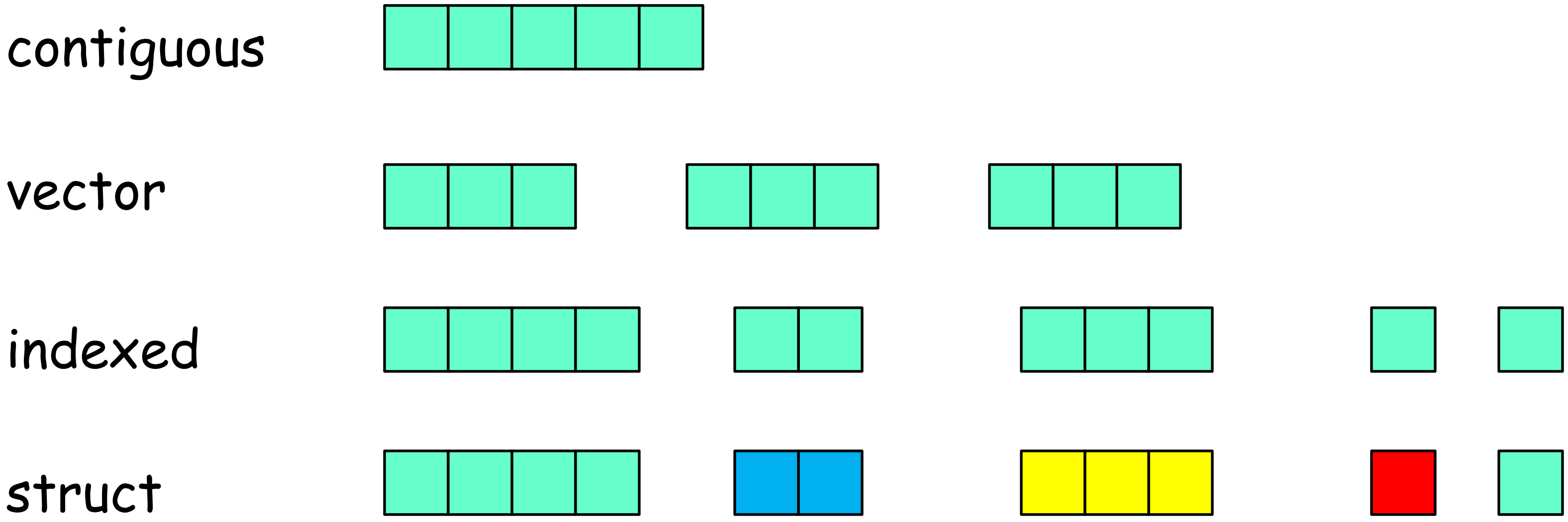
MPI datatypes

Describes unit of communication. Basic MPI datatypes correspond to basic datatypes of C and FORTRAN

New – user-defined or derived – datatypes can be constructed from previously described types as

•Contiguous : contigous blocks of element type
•Vectors: regularly strided blocks of element type
•Indexed: irregularly strided blocks of same type
•Structs: irregularly strided blocks of possibly different types

🟩 Basetype – basic or user-defined



contiguous

vector

indexed

struct

C integer datatypes

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

(*)

C integer datatypes(*)

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

(*)New with MPI 2.2, may not be implemented in your MPI version

C floating point datatypes

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

FORTRAN datatypes

Basic MPI_Datatype	FORTRAN type
MPI_INTEGER	INTEGER
MPI_REAL	REAL
MPI_DOUBLE_PRECISION	DOUBLE PRECISION
MPI_COMPLEX	COMPLEX
MPI_LOGICAL	LOGICAL
MPI_CHARACTER	CHARACTER(1)

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

Other point-to-point communication features

- MPI_PROC_NULL – „empty“ process to send to and receive from
- (MPI_Ssend, MPI_Bsend)
- Persistent requests
- MPI_Cancel – dangerous!
- MPI_Sendrecv_replace

Non-communication feature

```
double time = MPI_Wtime();
```

Get local time in number of seconds since some time in the past

```
stime = MPI_Wtime();

MPI_Send();

etime = MPI_Wtime();
// etime-stime is elapsed local time
```

MPI_WTIME_IS_GLOBAL: boolean attribute to MPI_COMM_WORLD, time is global (rare)

```
double time = MPI_Wtime();
```

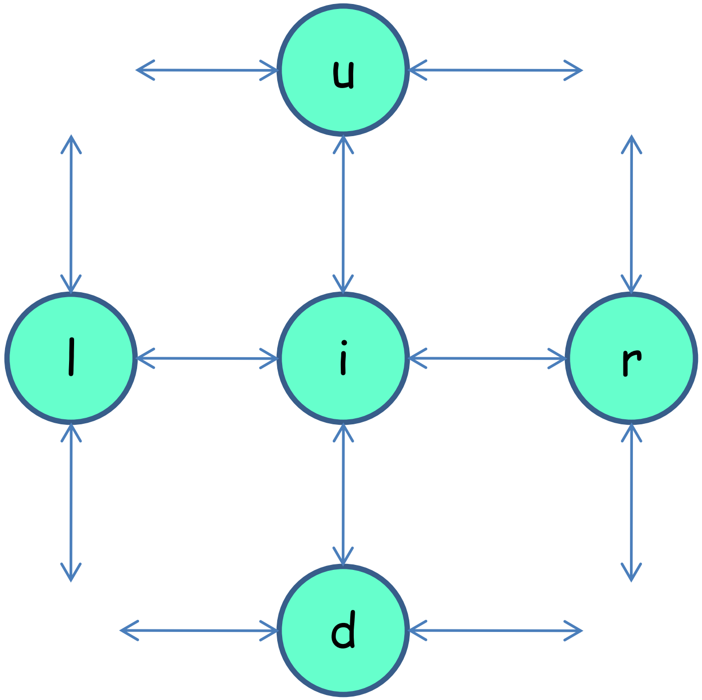
Get local time in number of seconds since some time in the past

```
MPI_Barrier(comm); // approx. Temporal synchronization

stime = MPI_Wtime();

MPI_Send();

etime = MPI_Wtime();
// etime-stime is elapsed local time
```

MPI_WTIME_IS_GLOBAL: boolean attribute to MPI_COMM_WORLD, time is global (rare)

One-sided communication – by example

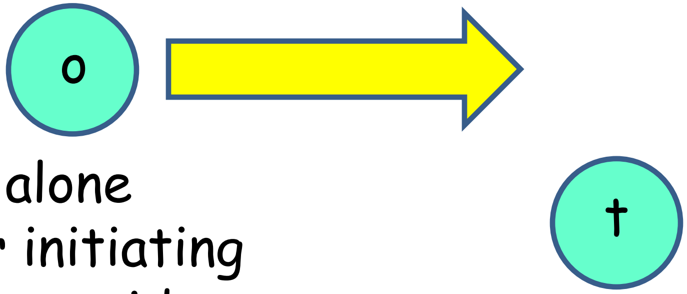
Safe neighbor exchange with one-sided (put) communication



```
MPI_Put(up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
```

- Where is the memory put to (and from)?
- When are data ready/operations complete?

One-sided communication decouples communication and synchronization



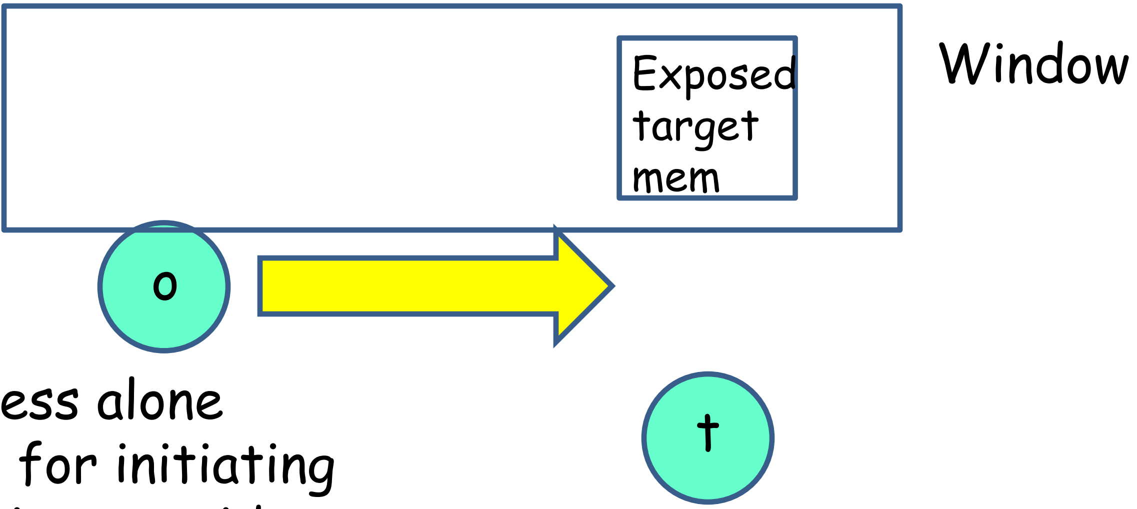
Origin process alone responsible for initiating communication, provides all arguments

Target process (semantically) not involved in communication

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

Communication calls are non-blocking, local completion semantics

Origin puts/get data from standard MPI buffer (buf,count,type)



Origin process alone responsible for initiating communication, provides all arguments

Target process (semantically) not involved in communication

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

Data on target exposed in window structure, addressed with relative displacement

Communication window:
Distributed, global object containing memory for each process that can be accessed in one-sided communication operations

```
MPI_Win_create(base,size,dispunit,info,comm,win);
```

Collective operation, all processes in comm provide a base address (size may be 0), displacement unit

info (special MPI (key,value) object) can influence window properties (use MPI_INFO_NULL)

MPI_Alloc_mem: special MPI memory allocator, sometimes beneficial (performance) for windows

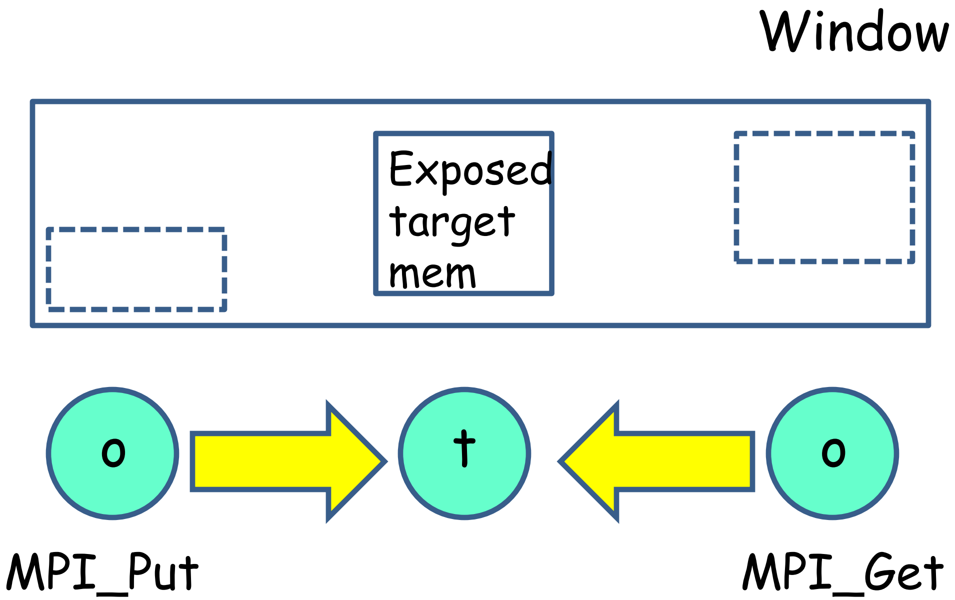
```
MPI_Put(obuf,…,target,targetdisp,…,win);
```

Data from obuf into target base+targetdispunit*targetdisp

NB: dispunit at target

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

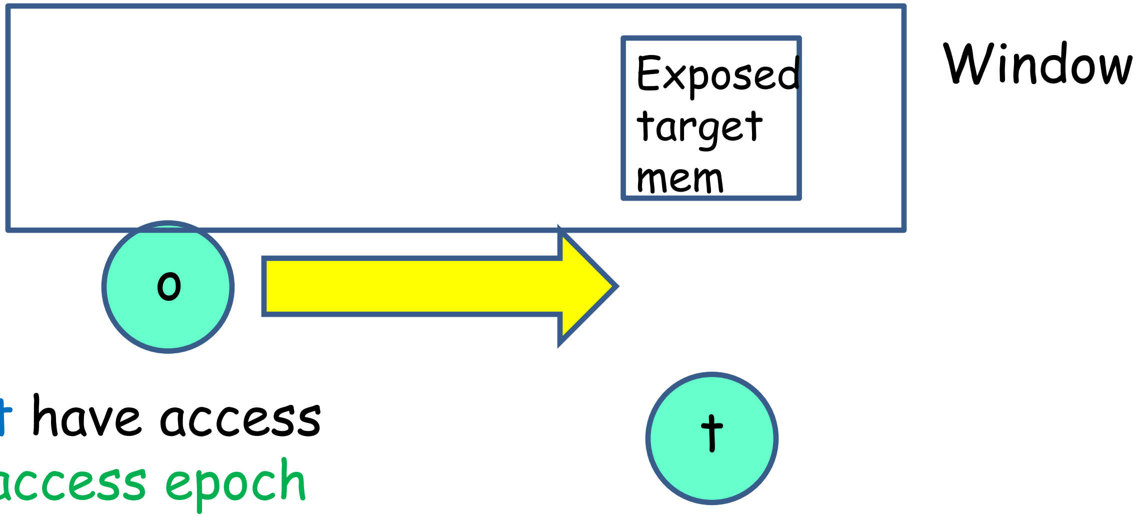
As for point-to-point communication



Concurrent gets/puts must access disjoint target addresses. Very strict rules, violation is erroneous (BUT usually not checked)

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

Communication epoch model



Origin must have access to target: access epoch

Target exposes memory: exposure epoch

End of epoch: access/exposure completed – data on origin processed (put or gotten), data on target arrived/accumulates complete

Synchronization, epochs

Active synchronization, both origin and target processes involved

```
MPI_Win_fence(assert,win)
```

Collective operation, all processes in comm of win must call. Closes previous epoch, opens access epoch to all processes, opens exposure epoch for all processes

Assertion can control opening/closure behavior

Synchronization, epochs

Active synchronization, both origin and target processes involved

```
MPI_Win_start(…,group)
MPI_Win_complete()
```

Opens/closes access epoch, targets as process group (MPI_Group)

```
MPI_Win_post(…,group)
MPI_Win_wait()
```

Opens/closes exposure epoch, origins as process group (MPI_Group)

„generalized“ pairwise synchronization…

Synchronization, epochs

Passive synchronization, only origin process involved

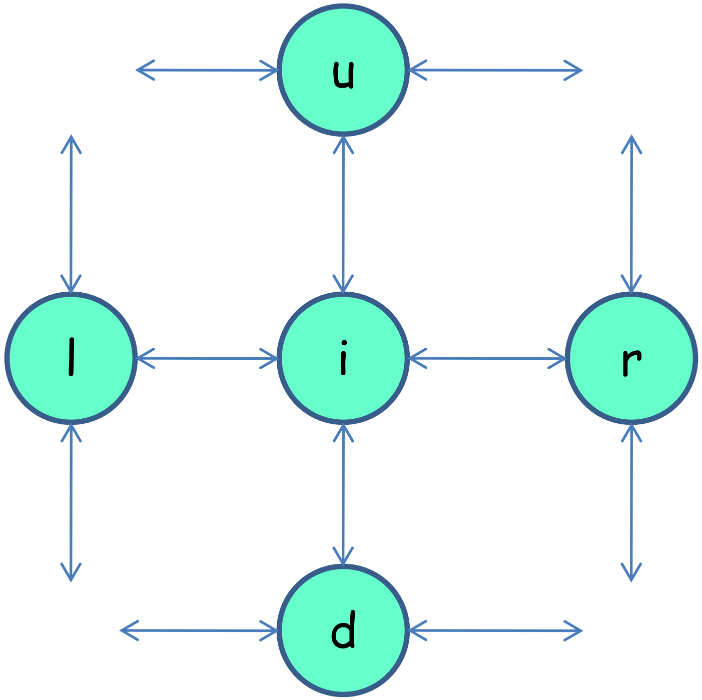
```
MPI_Win_lock(locktype,target,assertion,win);
MPI_Win_unlock(target,win);
```

Opens/closes exposure epoch at origin, access epoch at target

Note 1:
Not at all(!!) a lock – no test-and-set like operations, difficult to use for mutual exclusion. Very weak mechanism

Note 2:
Data at target may not be visible before target performs MPI_lock on itself (and other weirdness)

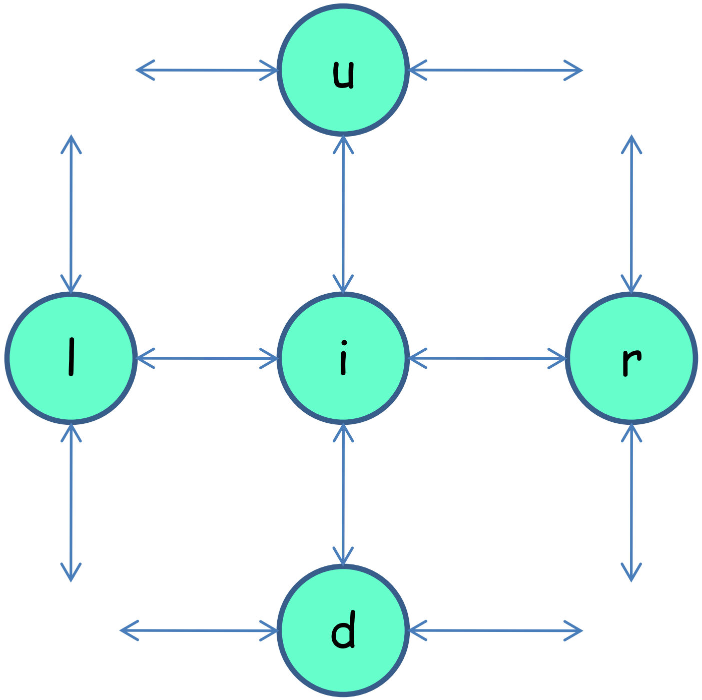
One-sided communication – by example



Safe neighbor exchange with one-sided (put) communication

```
// prepare neighbor data
MPI_Win_fence(win);
MPI_Put (up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
MPI_Win_fence(win);
// data from neighbors ready
```

Safe neighbor exchange with one-sided (put) communication



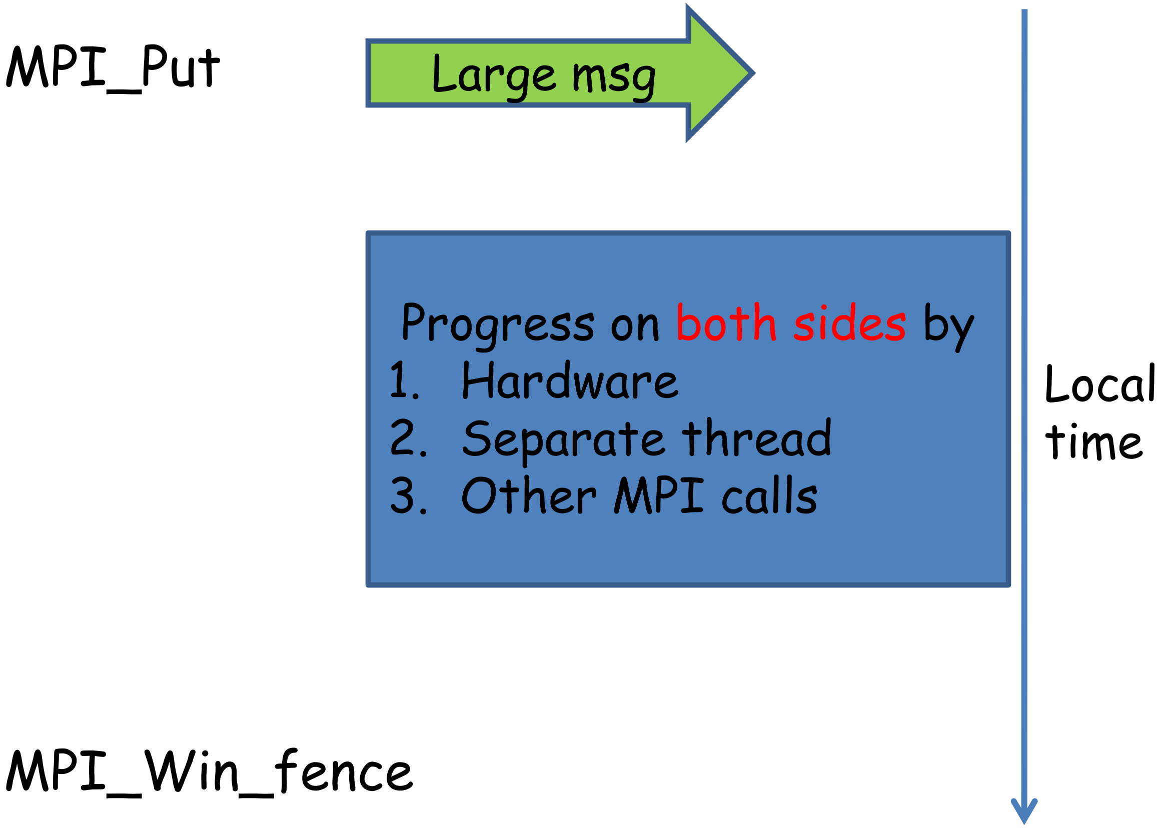
```
// prepare neighhbor data
MPI_Win_start ([l,u,r,d],win);
MPI_Win_post([l,u,r,d],win);
MPI_Put (up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
MPI_Win_wait(win);
MPI_Win_complete(win);
// data from neighbors ready
```

NB:
[l,u,r,d] is provided as process group (MPI_Group)

```
MPI_Win_free(win)
```

free after use... (like other MPI objects)

A note on progress



MPI_Put

Large msg

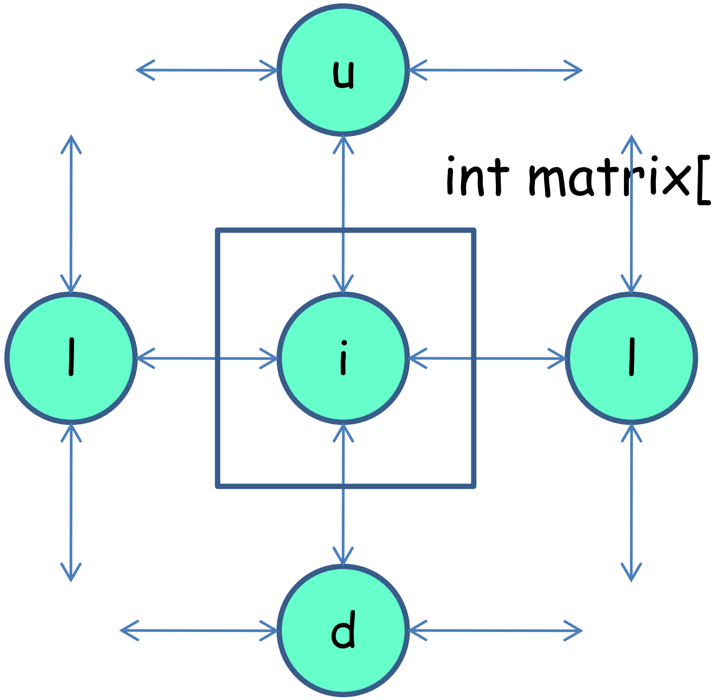
Progress on both sides by

1. Hardware
2. Separate thread
3. Other MPI calls

Local time

MPI_Win_fence

Example: datatypes for neighbor exchange



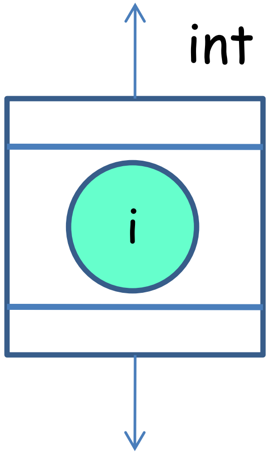
int matrix[d][d];

- Each MPI process has local dxd matrix
- n= dp
- n>>p

- Exchange upper row with lower row of upper process
- Exchange left column with right column of left process
- …

For all $0\le i<m$, $0\le j<n$, update

$u[i,j] <- \frac{1}{4}(u[i,j-1]+u[i,j+1]+u[i-1,j]+u[i+1,j]-h^2f(i,j))$



int matrix[d][d];

Rows:

```
MPI_Isend(m[0],d,MPI_INT,up,…);
MPI_Isend(m[d-1],d,MPI_INT,
          down,…);
```

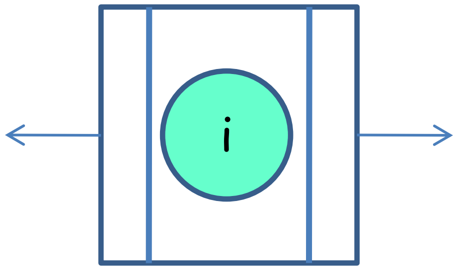
Or

```
MPI_Put(m[0],d,MPI_INT,up,…);
MPI_Put(m[d-1],d,MPI_INT,
        down, …);
```

In C, matrix is stored in row-major order. Rows can be sent/received as consecutive buffer

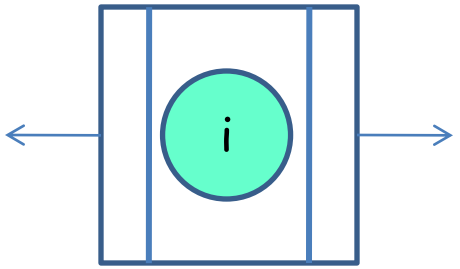
Columns:

int matrix[d][d];

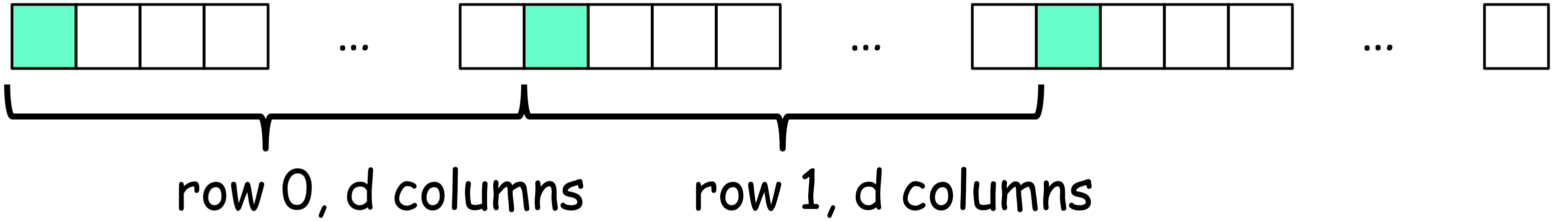


Columns:

```
int matrix[d][d];
```



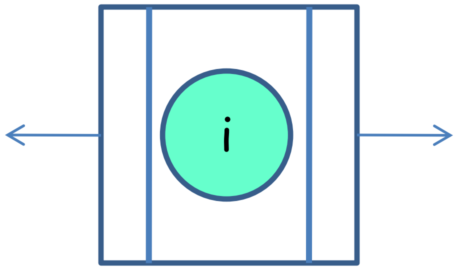
```
MPI_Datatype col;
MPI_Type_vector(d,1,d,MPI_INT,col);
MPI_Type_commit(&col);

MPI_Isend(&m[0][0],1,col,left,…);
MPI_Isend(&m[0][d-1],1,col,
          down,…);
```



```
MPI_Type_free(&col); // when done
```

Columns:

int matrix[d][d];



```
MPI_Datatype col;
MPI_Type_vector(d,1,d,MPI_INT,col);
MPI_Type_commit(&col);

MPI_Isend(&m[0][0],1,col,left,…);
MPI_Isend(&m[0][d-1],1,col,
          down,…);
```

```
MPI_Type_free(&col); // when done
```

Advice: use it! Should be at least as good as

a) Copying the row elements into intermediate, consecutive int buffer

b) Sending intermediate buffer

MPI: collective comm

- Collective communication

Collective operations - motivation

Task:
each process has a vector of elements, needs to compute the elementwise sum of all vectors, and store result vector at some root/all processes

x0+x1+x1+ … + x(p-1) = y

„Root“:
designated MPI process that receives/computes final result

Method 1: root receives and computes

```
MPI_Send(x,n,MPI_<type>,root,SUMTAG,comm);

if (rank==root) {
  void *z; // intermediate n element buffer
  z = malloc(n*sizeof(<type>);
  for (i=0; i<p; i++)  {
    MPI_Recv(z,n,MPI_<type>,i,SUMTAG,comm,&status);
    for (j=0; j<n; j++) {
      y[j] += z[j]; // type cast required
   }
  }
}
```

The program is unsafe. Tedious, if required to work for all possible C types.

Method 1: root receives and computes

```
MPI_Send(x,n,MPI_<type>,root,SUMTAG,comm);

if (rank==root) {
  void *z; // intermediate n element buffer
  z = malloc(n*sizeof(<type>);
  for (i=0; i<p; i++)  {
    MPI_Recv(z,n,MPI_<type>,i,SUMTAG,comm,&status);
    for (j=0; j<n; j++) {
      y[j] += z[j]; // type cast required
   }
  }
}
```

Performance: O(p), p(α+βn)+pγn, γ time of „+“ per element

No speedup possible - sequential summing of p vectors: pγn

Method 2: ring, all compute

```
prev = (rank-1+size)%size; next = (rank+1)%size;
if (rank==root) {
  void *z; // intermediate n element buffer
  MPI_Recv(z,n,MPI_<type>,prev,SUMTAG,comm,&status);
  for (j=0; j<n; j++) {
    y[j] = x[j]+z[j]; // type cast required
  }
} else {
  if (prev!=root) {
    MPI_Recv(z,n,MPI_<type>,prev,SUMTAG,comm,&status);
    for (j=0; j<n; j++) y[j] = x[j]+z[j]; // cast
  } else {
    for (j=0; j<n; j++) y[j] = x[j]; // cast
  }
  MPI_Send(y,n,MPI_<type>,next,SUMTAG,comm)
};
```

Method 2: ring, all computes

Ring: result y is computed in the order

x(root+1)+x(root+2)+…+x(size-1)+x0+…+x(root)

What if root≠size-1, and the operation „+“ is not commutative?

Performance: still no speedup

Method 2: ring, all computes

```
int RingReduce(void *sendbuf,
               void *recvbuf, int count,
               MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm)
{
   <insert method 2 or 1 here>
   return MPI_SUCCESS; // everything went fine…
}
```

MPI_Op: MPI type handle for binary „operators“
MPI_Datatype: handle for datatypes

Method 2: ring, all computes

```
int RingReduce(void *sendbuf,
               void *recvbuf, int count,
               MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm)
{
   <insert method 2 or 1 here>
   return MPI_SUCCESS; // everything went fine…
}
```

What happens here: (if i==j+1)

Process i:

```
RingReduce(x1,y1,…,root,…,
           comm);
MPI_Recv(a,…,j,SUMTAG,…);
```

Process j:

```
MPI_Send(a,…,i,SUMTAG,comm);
RingReduce(x1,y1,…,root,…,
           comm);
```

Unsafe parallel library function!

Method 2: ring, all computes

```
int RingReduce(void *sendbuf,
               void *recvbuf, int count,
               MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm)
{
   <insert method 2 or 1 here>
   return MPI_SUCCESS; // everything went fine…
}
```

And here:

Process i:
RingReduce(x1,y1,…,root0,…);
RingeReduce(x2,y2,…,root37,…);

Process j:
RingReduce(x2,y2,…,root37,…);
RingReduce(x1,y1,…,root0,…);

Unintended use; unsafe

Method 3: using properties of „+“ to improve performance

Since „+“ is associative

x0+x1+x2+ … + x(p-1) = y

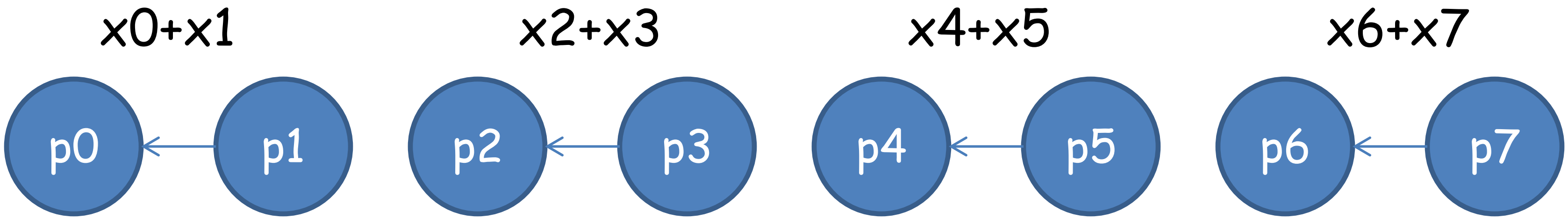
can be computed as

(x0+x1)+(x2+x3) + … + x(p-1) = y

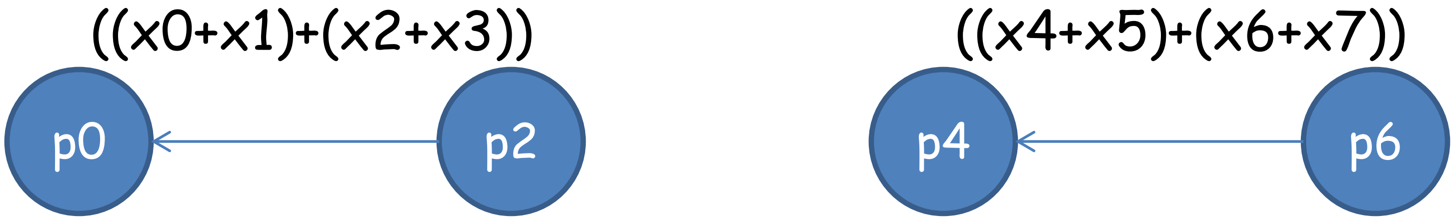
and

((x0+x1)+(x2+x3)) + …((x(p-2) + x(p-1)) = y

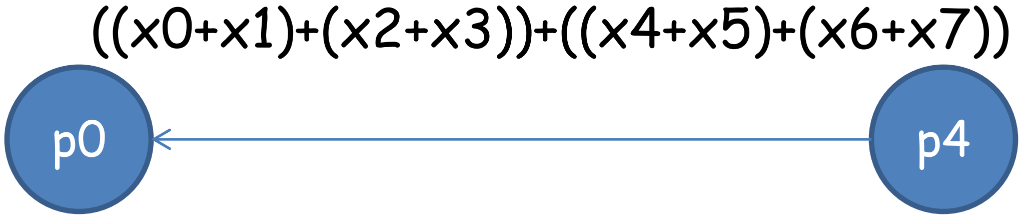
Step 1: in parallel



Step 2: in parallel



Step 3: in parallel



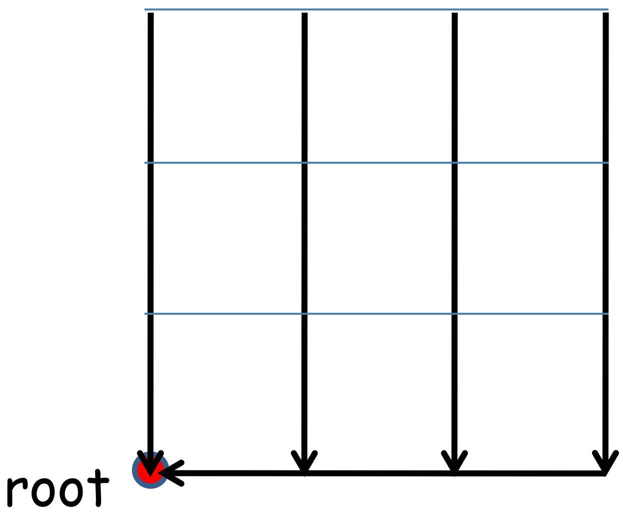
„Theorem“:
Sum can be computed log_2 p communication rounds with p processes by binomial tree algorithm

Time log_2(α+βn+γn)

Assumption:
Tree-like communication is efficiently supported by underlying communication network

Meets lower bound (as for broadcast), not possible to reduce in less than log_2 p rounds, even on fully connected network

Reduction on mesh/torus networks



Phase 1:
reduce vertic ally
Phase 2:
Reduce horizontally

Time: $\sqrt{p}(\alpha+\beta n)$

Collective operations - motivation

- Implementation of summation tedious: must to work for all combinations of datatypes, binary operators, …
- Performance dependent on communication network properties
- Different algorithms for different networks
- Different algorithms for different vector sizes, datatypes, …
- …

```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm);
```

as a „collective operation" in MPI

Collective operations - motivation

```
MPI_Reduce(sendbuf,recvbuf,count,datatype,op,root,comm);
```

- Saves work for application programmer: no need to implement complicated, own library functions
- Improves portability: part of MPI standard, available everywhere
- Improves performance portability: good MPI implementation will provide „best possible“ performance for given system

Collective communication (and reduction) operations

MPI_Bcast – data from root to all

MPI_Scatter – individual (personalized) data from root to all
MPI_Gather – individual data from all to root

MPI_Alltoall – individual (personalized) data from all to all, “transpose”

MPI_Allgather – data from all to all

MPI_Reduce – apply associative function (e.g. “+”) to data from each process, result at root

MPI_Allreduce – result to all
MPI_Reduce_scatter – result scattered (parts) to all

MPI_Barrier – (semantic) synchronization

Collective MPI operations

All functions of MPI requiring participation of all processes in communicator

- Many bookkeeping functions (MPI_Comm_split, …)
- Dynamic process spawning
- MPI-IO (collective and individual functionalities)
- Virtual topologies (MPI_Graph_create, …)

17 (16 in MPI 1) collective communication (and reduction) operations are called the „collectives“ of MPI

Collective MPI operations are called the same way by the participating processes, same arguments for all processes, but some arguments may be significant only at some processes (root)

Process i (non-root):

```
MPI_Reduce(sbuf,rbuf,…,root,comm);
```

Process j (root):

```
MPI_Reduce(sbuf,rbuf,…,root,comm);
```

Again: all processes in comm must participate

Example: reduction of single “scalar” (C int, MPI_INT)

```
if (rank==root) {
   x = rank;
   MPI_Reduce(&x,&y,1,MPI_INT,MPI_SUM,root,comm);
   if (y!=(size*(size-1))/2) printf(„Error!\n“);
   // y significant at root only
} else {
   x = rank;
   MPI_Reduce(&x,&y,1,MPI_INT,MPI_SUM,root,comm);
}
```

Collective operation semantics

Requirement:

If a process calls collective MPI_<A> on communicator C, then eventually all other processes in C must call MPI_<A> and no other collective inbetween (on that communicator)

Collective operations are safe: collective communication on communicator C will not interfere with other communication on C

Collective operation semantics

Requirement:

If a process calls collective MPI_<A> on communicator C, then eventually all other processes in C must call MPI_<A> and no other collective inbetween (on that communicator)

Collective functions are blocking. A process returns when locally complete, buffers etc. can be reused. Completion semantics are non-local (most likely dependent on what other processes do) (*)

Collective functions are not synchronizing. A process may leave MPI_<A> as soon as it is locally complete (required local data sent and received)

Like MPI_Send

Exception: `MPI_Barrier(comm);`

(*) nonblocking collectives will be part of MPI 3.0

Correct:

Process i:

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

Process j:

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

Process local time

MPI_Bcast is blocking:

root: does not return before data have left buffer

Non-root: does not return before data from root have been received in buffer

Correct:

Process i:

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

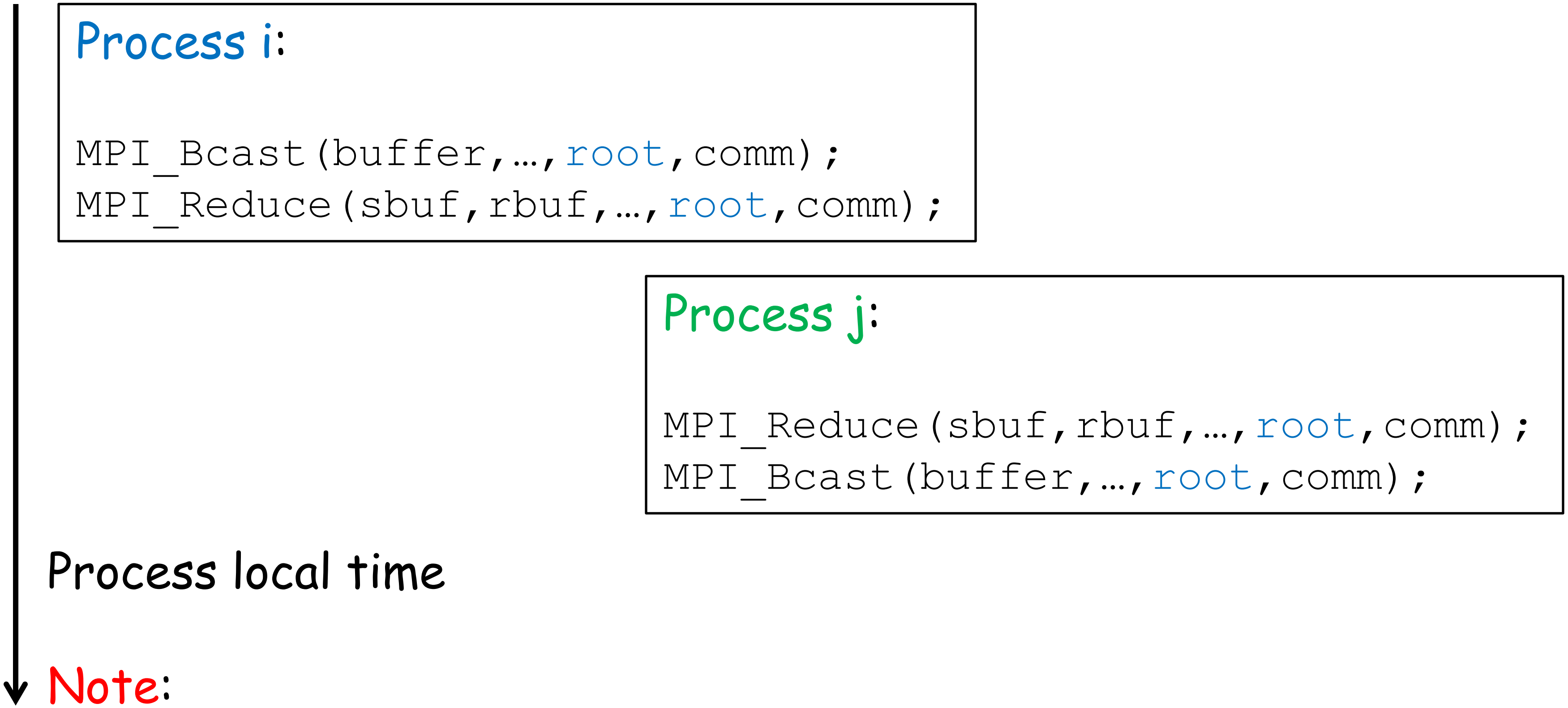
Process j:

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

Process local time

MPI_Bcast is not synchronizing:
root: may return as soon as data have left buffer (independent of non-roots)
Non-root: may return as soon as data from root have been received in buffer (independent of other non-roots)

Incorrect:



Process i:

```
MPI_Bcast(buffer,…,root,comm);
MPI_Reduce(sbuf,rbuf,…,root,comm);
```

Process j:

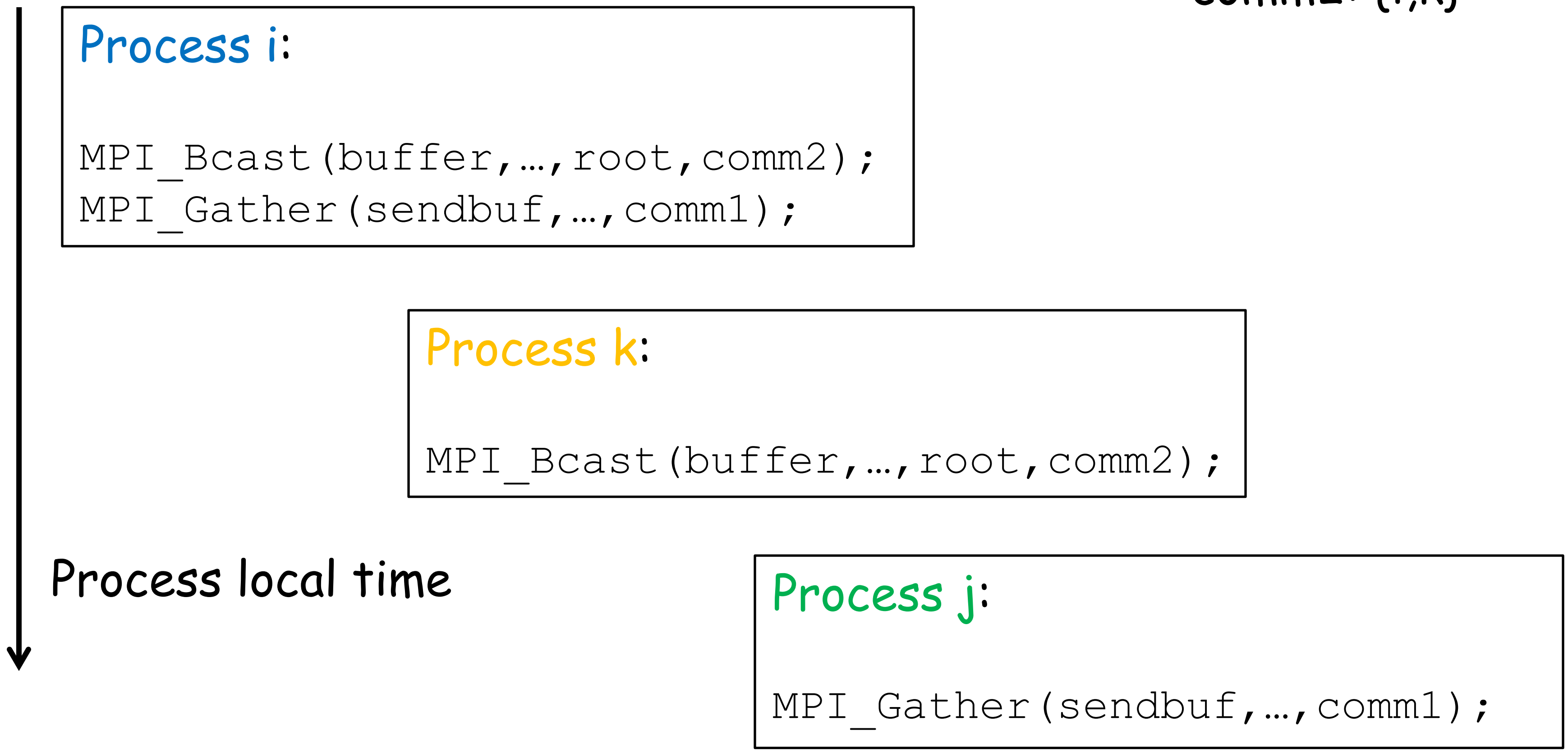
```
MPI_Reduce(sbuf,rbuf,…,root,comm);
MPI_Bcast(buffer,…,root,comm);
```

Process local time

Note:
“incorrect” means that MPI may crash, deadlock, give wrong results! Or even work (for small counts: unsafe)

Correct:

comm1: {i,j}
comm2: {i,k}



Process i:

```
MPI_Bcast(buffer,…,root,comm2);
MPI_Gather(sendbuf,…,comm1);
```

Process k:

```
MPI_Bcast(buffer,…,root,comm2);
```

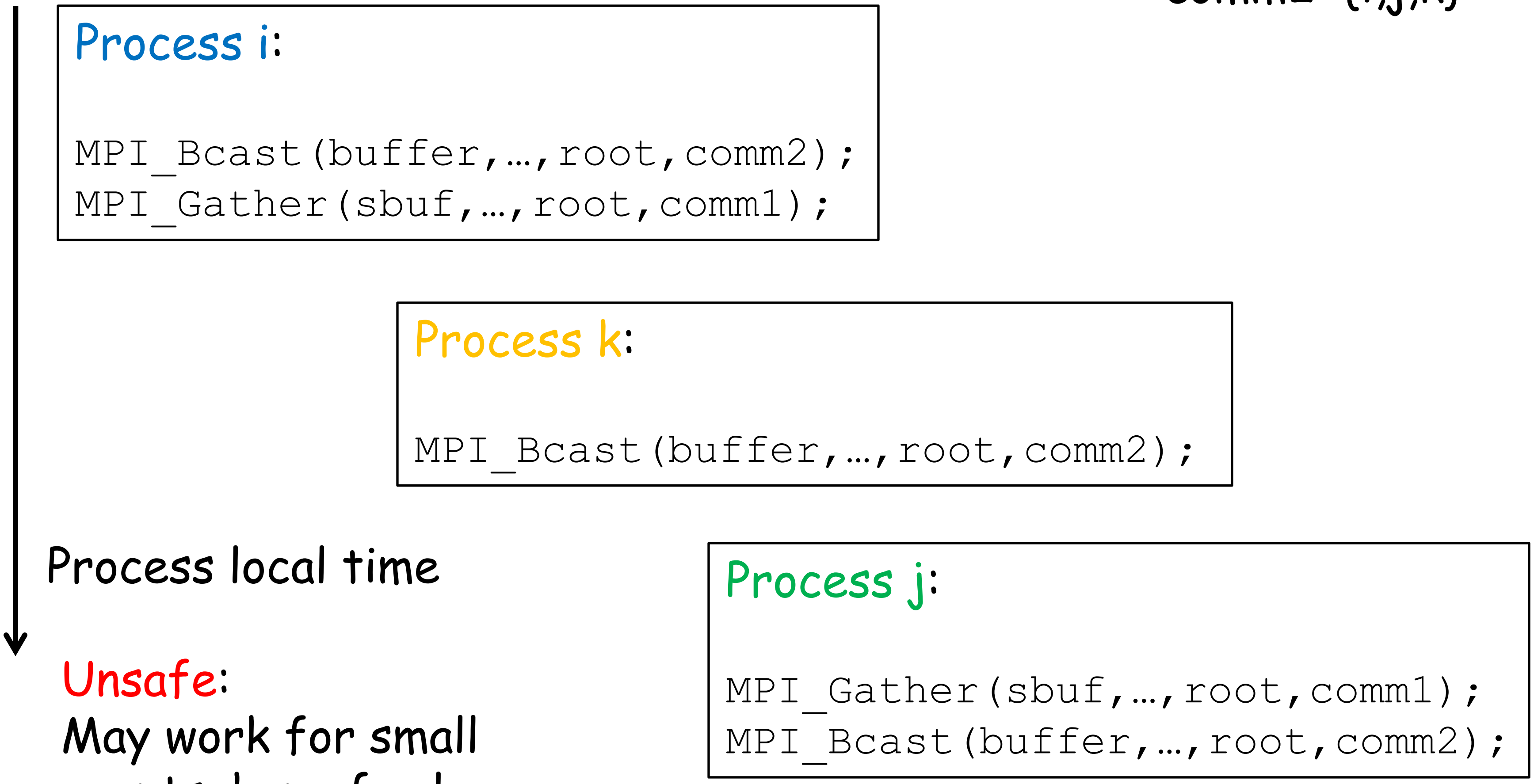
Process local time

Process j:

```
MPI_Gather(sendbuf,…,comm1);
```

Unsafe:

comm1: {i,j}
comm2: {i,j,k}



Process i:

```
MPI_Bcast(buffer,…,root,comm2);
MPI_Gather(sbuf,…,root,comm1);
```

Process k:

```
MPI_Bcast(buffer,…,root,comm2);
```

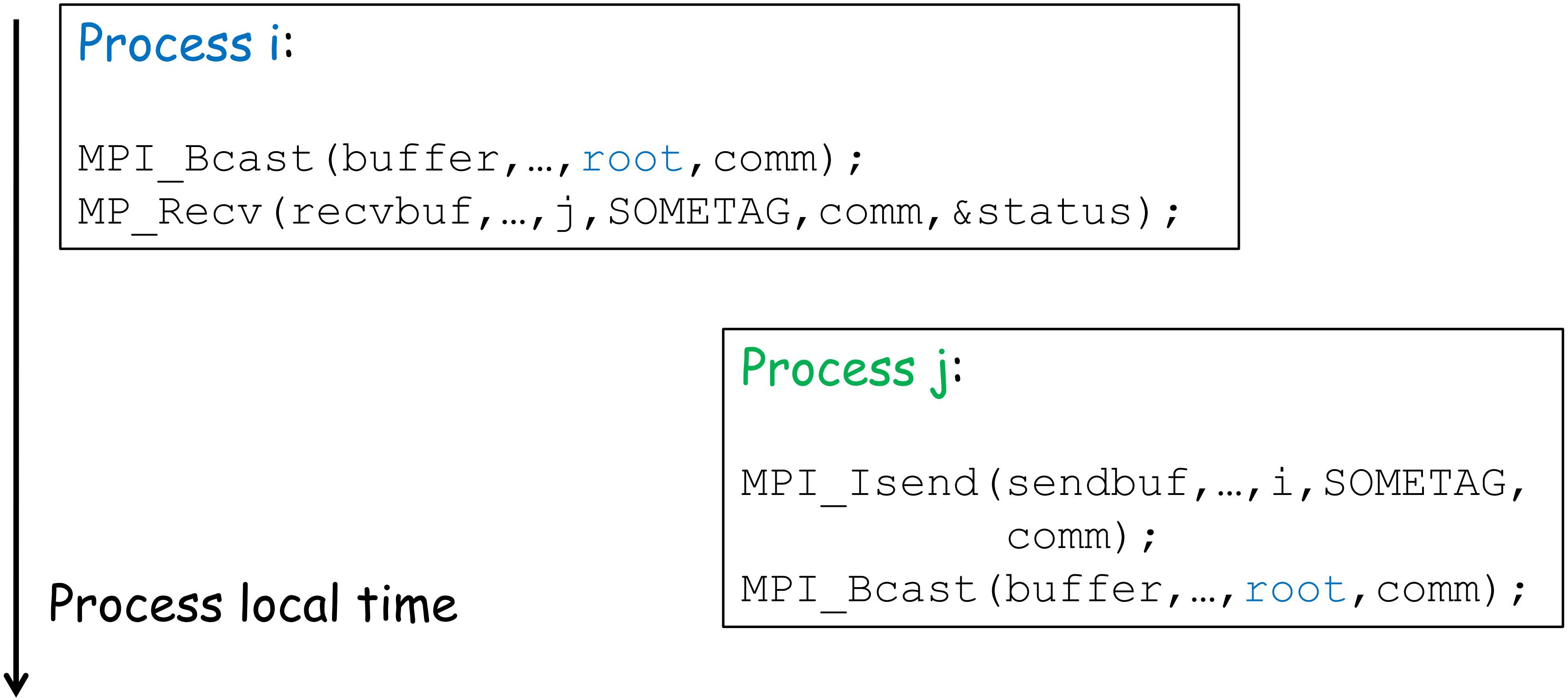
Process j:

```
MPI_Gather(sbuf,…,root,comm1);
MPI_Bcast(buffer,…,root,comm2);
```

Process local time

Unsafe:
May work for small counts, hang for large

Safe:



Process i:

```
MPI_Bcast(buffer,…,root,comm);
MP_Recv(recvbuf,…,j,SOMETAG,comm,&status);
```

Process j:

```
MPI_Isend(sendbuf,…,i,SOMETAG,
          comm);
MPI_Bcast(buffer,…,root,comm);
```

Point-to-point and one-sided and collective communication does not interfere

Process involvement in/blocking behavior of collective call MPI_<A> is implementation dependent

Unsafe collective programming: relying on synchronization properties

Observation:
Explicit MPI_Barrier calls are never (should never be) needed for correctness of MPI programs

If it seems so, there's probably something wrong

```
MPI_Barrier(comm);
```

Calling process waits for all other processes in `comm` to enter barrier, can leave when all others have performed call

Purely semantic definition; no requirement that barrier can be used to synchronize time (e.g. for benchmark purposes)

MPI libraries attempt to have a fast, accurate barrier, so that all processes leave barrier „more or less at the same time“

Sometimes HW support helps (atomic counters, collective network)

Example: timing a function

```
MPI_Barrier(comm);
// processes may be synchronized here
double start = MPI_Wtime();

<something to be timed>

double stop = MPI_Wtime();

double local_time = stop-start;
```

Example: benchmarking

Repeat measurement until stable, reproducible result has been achieved

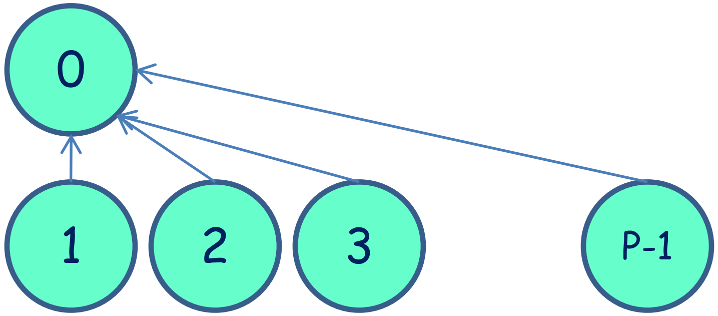
```
for (r=0; r<REPETITIONS; r++) {
  MPI_Barrier(comm);
  // processes may be synchronized here
  double start = MPI_Wtime();

  <something to be timed>

  double stop = MPI_Wtime();

  double local_time = stop-start;
  // compute local average time, max time, min time
}
```

A (legal) barrier implementation:

not suitable for timing!

MPI libraries do something better…

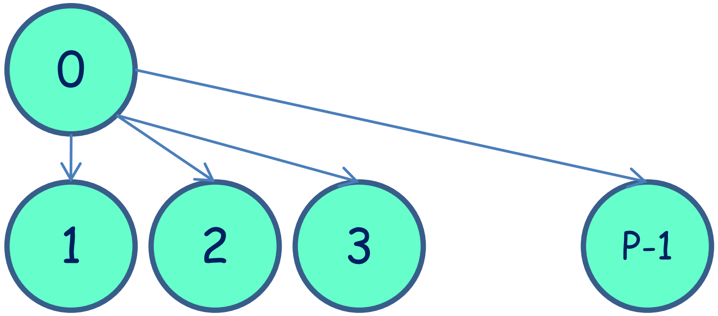
Phase 1: „gather“



```
for (i=1; i<p; i++)
MPI_Recv(NULL,0,MPI_BYTE,…,comm);
```

```
MPI_Send(NULL,0,…,comm);
```

Phase 2: „scatter“



```
for (i=1; i<p; i++)
MPI_Send (NULL,0,…,comm);
```

```
MPI_Recv(NULL,0,…,comm);
```

MPI „collectives“ classification

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	

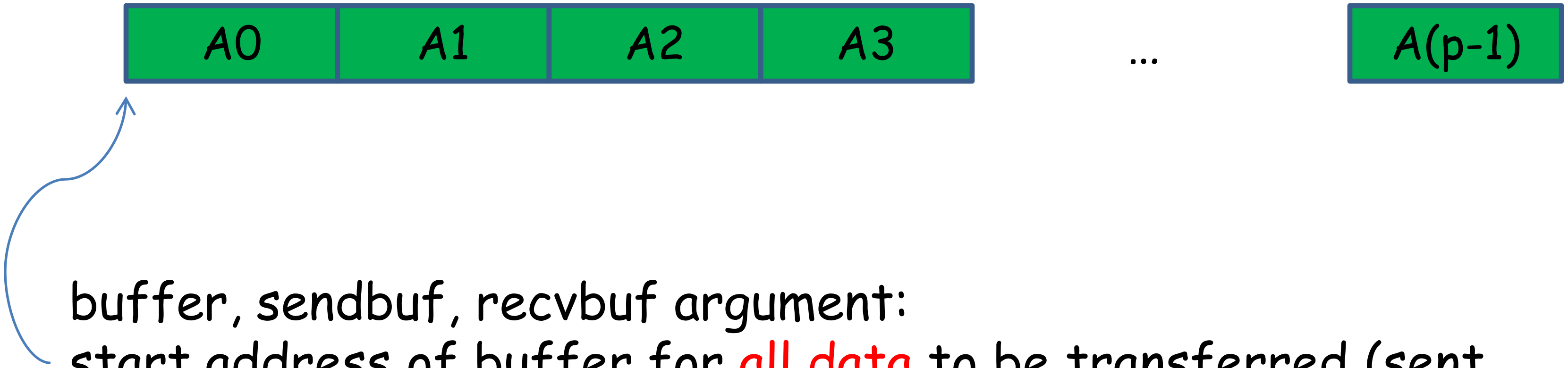
(*) MPI_Reduce_scatter_block: MPI 2.2 extension

Symmetric vs. non-symmetric: all processes lay the same role in collective vs. one/some process (root) is special

Regular vs. irregular: each process contributes or receives the same amount of data from/to each other process

Note:
As for all other types of MPI communication, data in collective operations can be structured, described by derived datatype

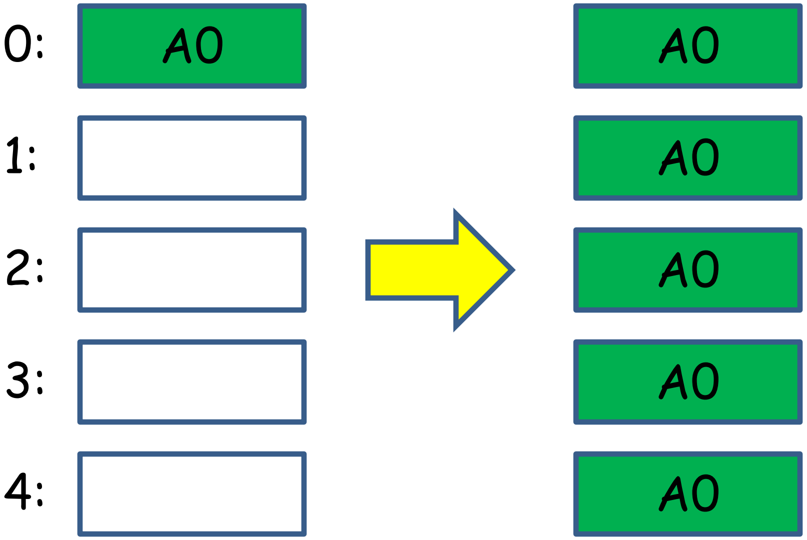
Regular collectives



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(buffer,count,datatype,root,comm);
```

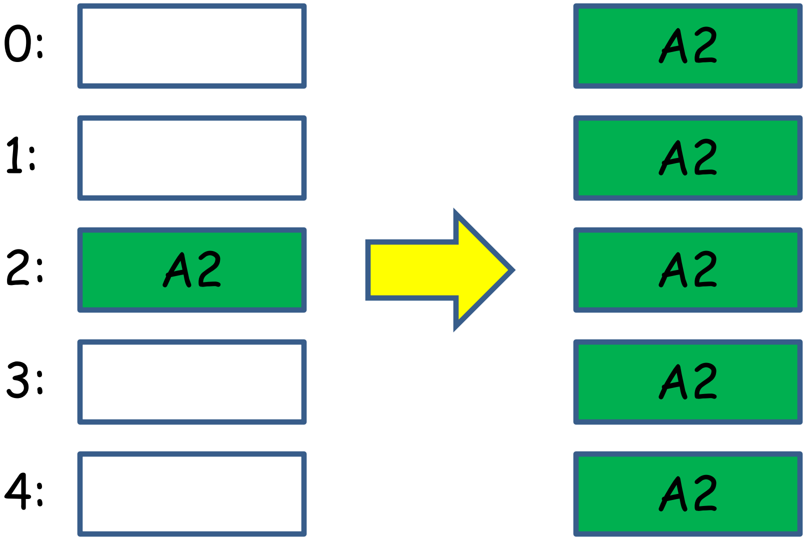


Example: root==0

Semantics: data from root buffer is transferred to buffer of all non-root processes

Use: All processes Bcast with same root, buffer with same type signature (e.g. same count for basic datatypes like MPI_FLOAT)

```
MPI_Bcast(buffer,count,datatype,root,comm);
```



Example: root==2

Semantics: data from root buffer is transferred to buffer of all non-root processes

Use: All processes Bcast with same root, buffer with same type signature (e.g. same count for basic datatypes like MPI_FLOAT)

MPI 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,3,MPI_INT,0,comm);
} else {
  // do something on non-root first
  MPI_Bcast(matrixdims,2,MPI_INT,0,comm);
  // uhuh, Bcast probably works, but later…
}
```

MPI requirement

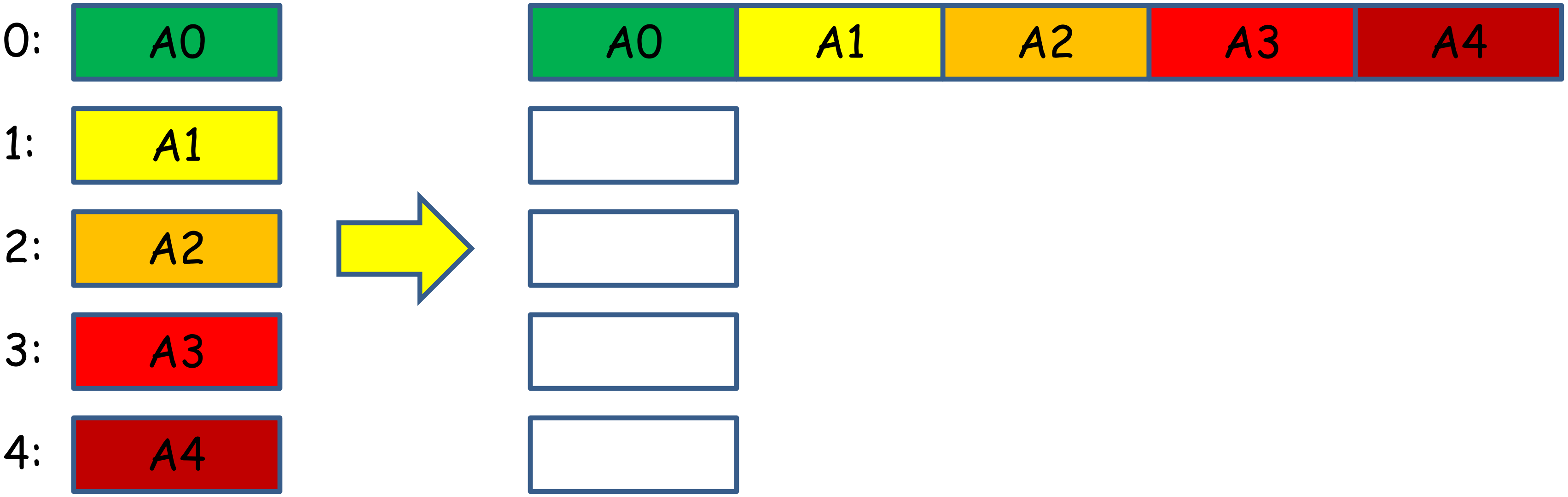
Collective functions MUST be called with consistent arguments

- 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.)

Calling with different roots probably just deadlocks

For efficiency reasons, MPI libraries do not check such things. User on his own when making mistakes. Consistency tools can help!

```
MPI_Gather(sbuf,scount,stype,rbuf,rcount,rtype,root,comm);
```

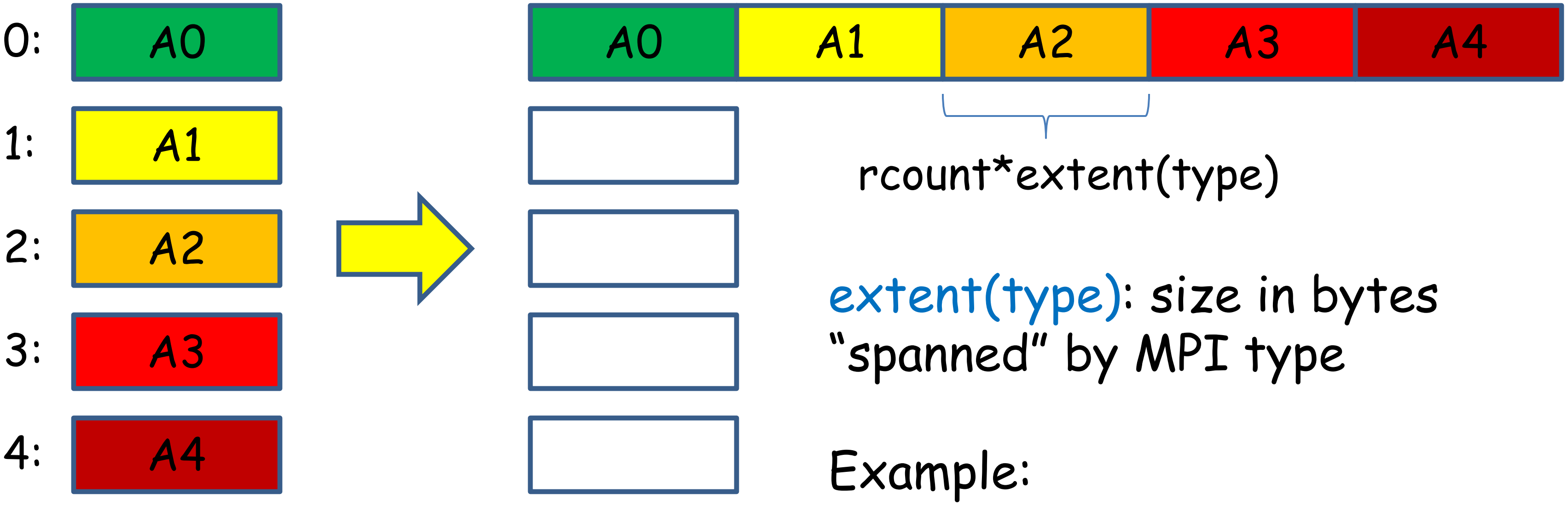


Semantics: each process contributes a block of data to rbuf at root, blocks end up stored consecutively in rank order at root

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

Note: rcount is count of one block, not of whole rbuf

```
MPI_Gather(sbuf,scount,stype,rbuf,rcount,rtype,root,comm);
```



extent(type): size in bytes "spanned" by MPI type

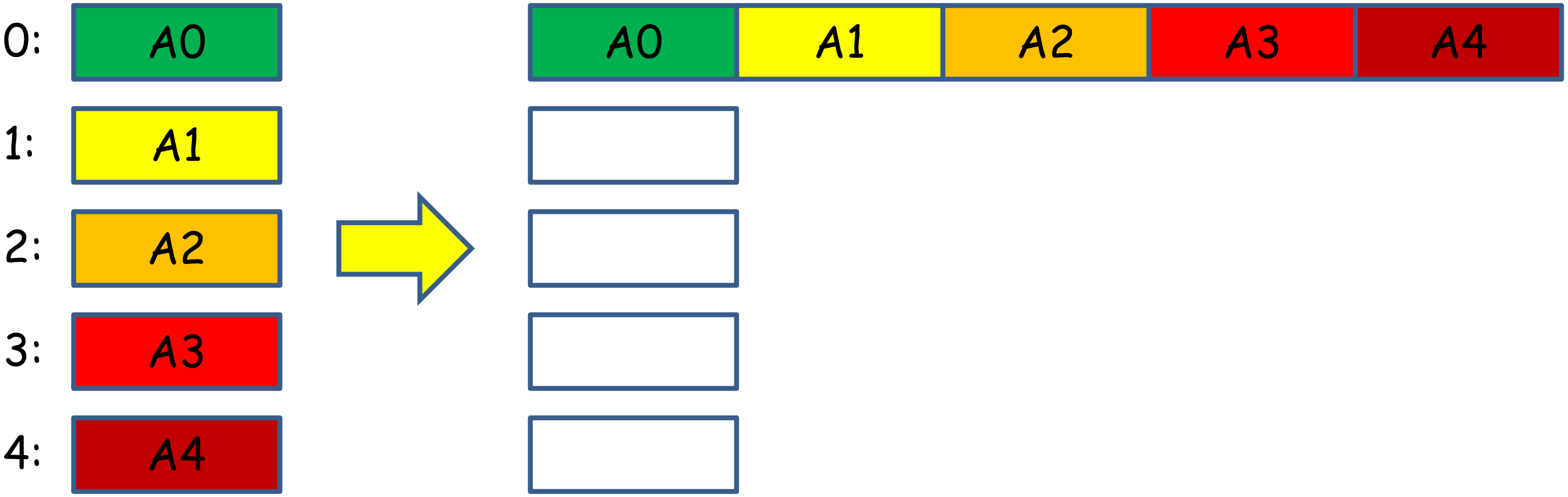
Example:
extent(MPI_INT) == sizeof(int)

Semantics: each process contributes a block of data to rbuf at root, blocks end up stored consecutively in rank order at root

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

Note: rcount is count of one block, not of whole rbuf

```
MPI_Gather(sbuf,scount,stype,rbuf,rcount,rtype,root,comm);
```

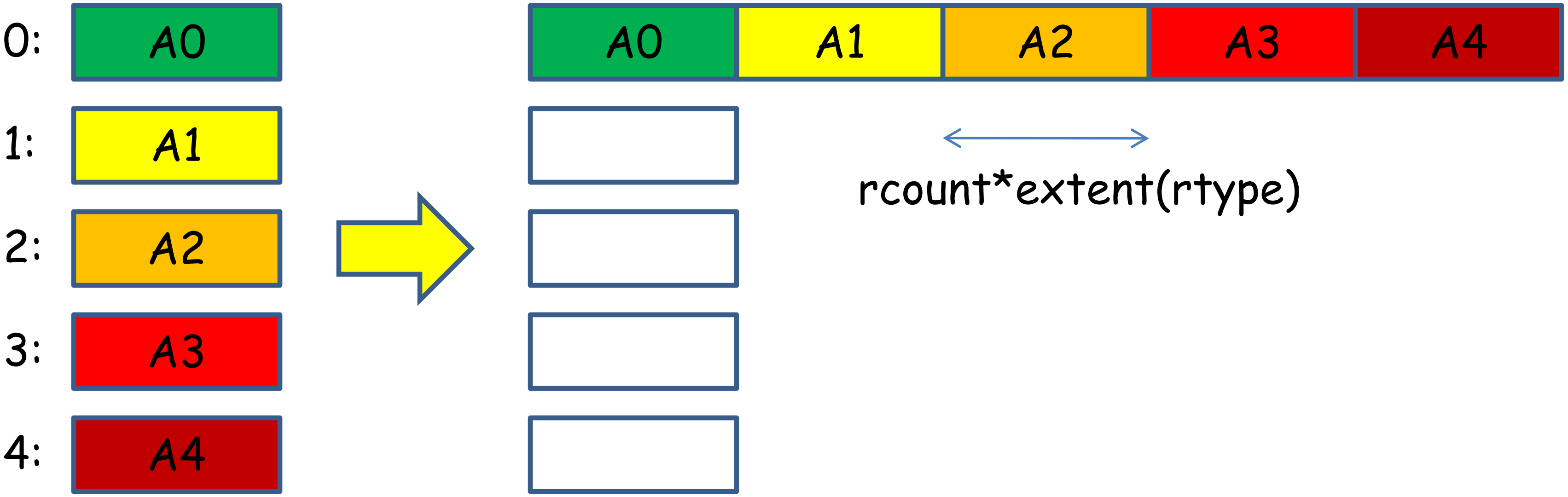


Result buffer (rbuf,rcount,rtype) significant only on root

Note: root also gathers from itself

Special MPI buffer argument MPI_IN_INPLACE can be used on root for sbuf to indicate that result from root is already „in place"

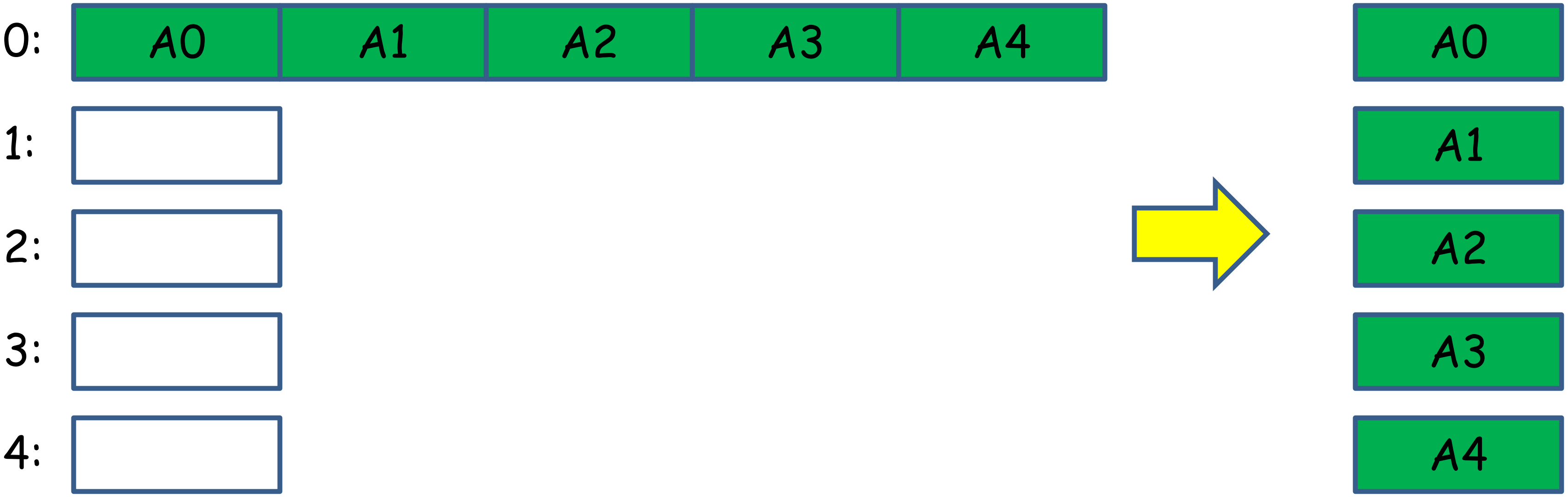
```
MPI_Gather(sbuf,scount,stype,rbuf,rcount,rtype,root,comm);
```



Semantics (only!), NOT implemented this way:

```
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);
```

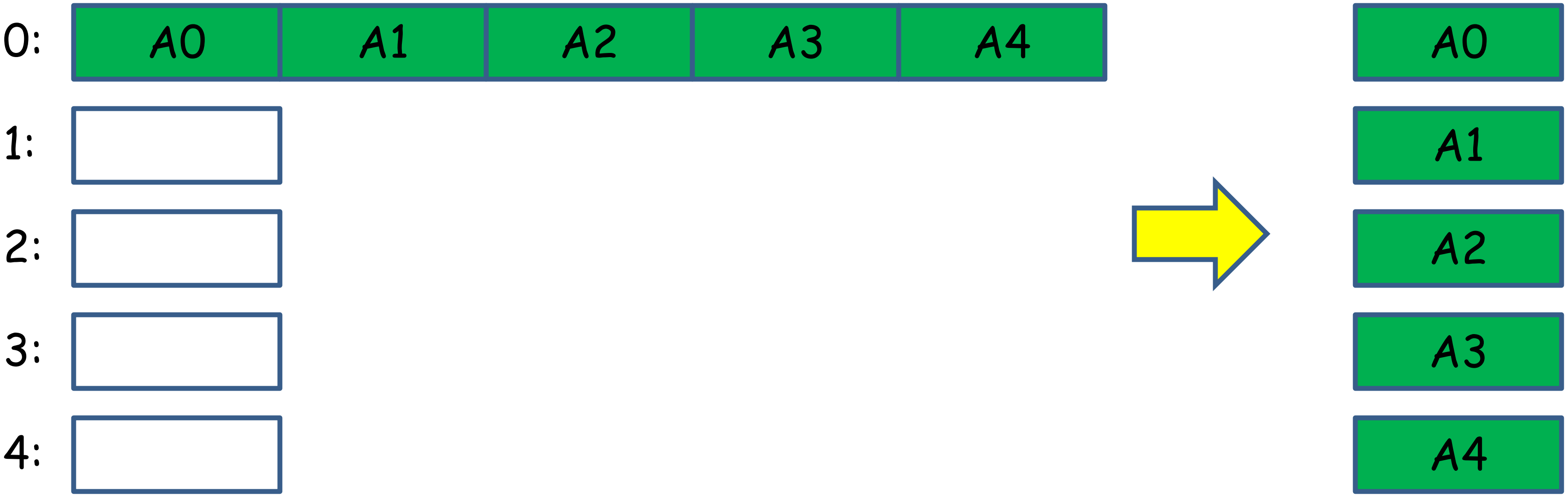
```
MPI_Scatter(sbuf,scount,stype,rbuf,rcount,rtype,root,comm);
```



Semantics: root contributes a different block of data to each process, blocks stored consecutively in rank order at root

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

```
MPI_Scatter(sbuf,scount,stype,rbuf,rcount,rtype,root,comm);
```



Send buffer (sbuf,scount,stype) significant only on root

MPI_IN_INPLACE can be used on root for rbuf to indicate that result from root is already „in place"

Further differences to point-to-point communication:

- Collective communication functions do not have a 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

Further differences to point-to-point communication:

- Collective communication functions do not have a 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

Further differences to point-to-point communication:

- Collective communication functions do not have a 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

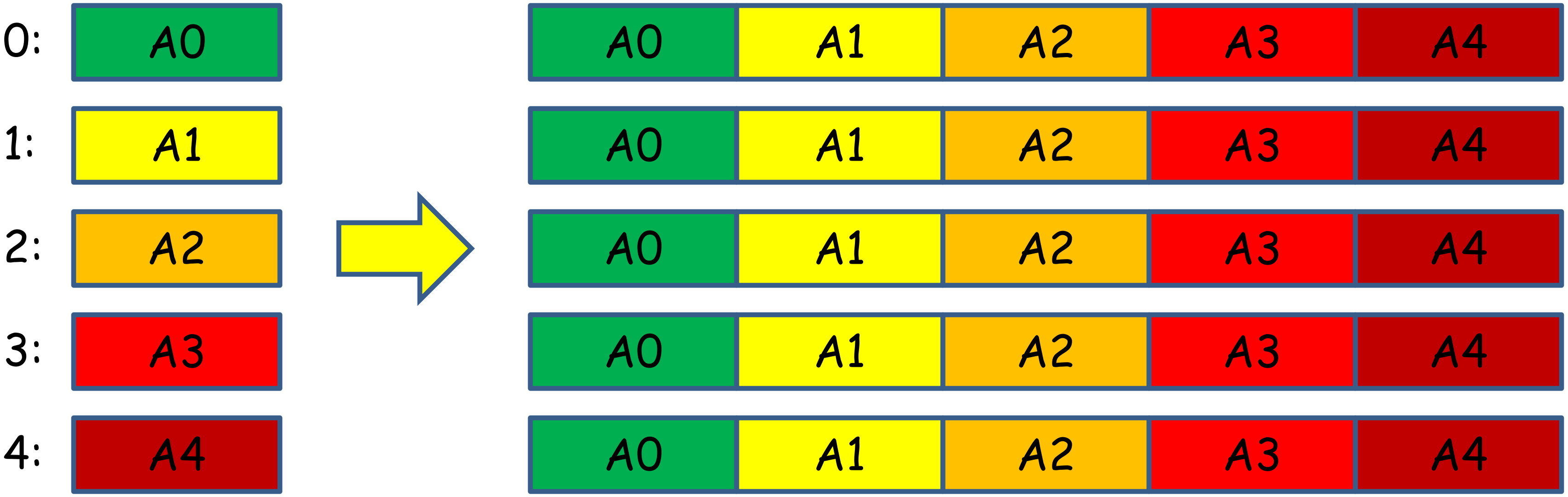
Does this barrier work?

```
MPI_Gather(NULL,0,MPI_BYTE,NULL,0,MPI_BYTE,0,comm);
MPI_Scatter(NULL,0,MPI_BYTE,NULL,0,MPI_BYTE,0,comm);
```

Well, depends, it may (performance wise better than send-recv implementation, but still bad) – but depends whether 0-buffers are gathered/scattered

Unsafe collective programming: relying on synchronization properties

```
MPI_Allgather(sbuf,scount,stype,rbuf,rcount,rtype,comm);
```



0: A0 A0 A1 A2 A3 A4

1: A1 A0 A1 A2 A3 A4

2: A2 A0 A1 A2 A3 A4

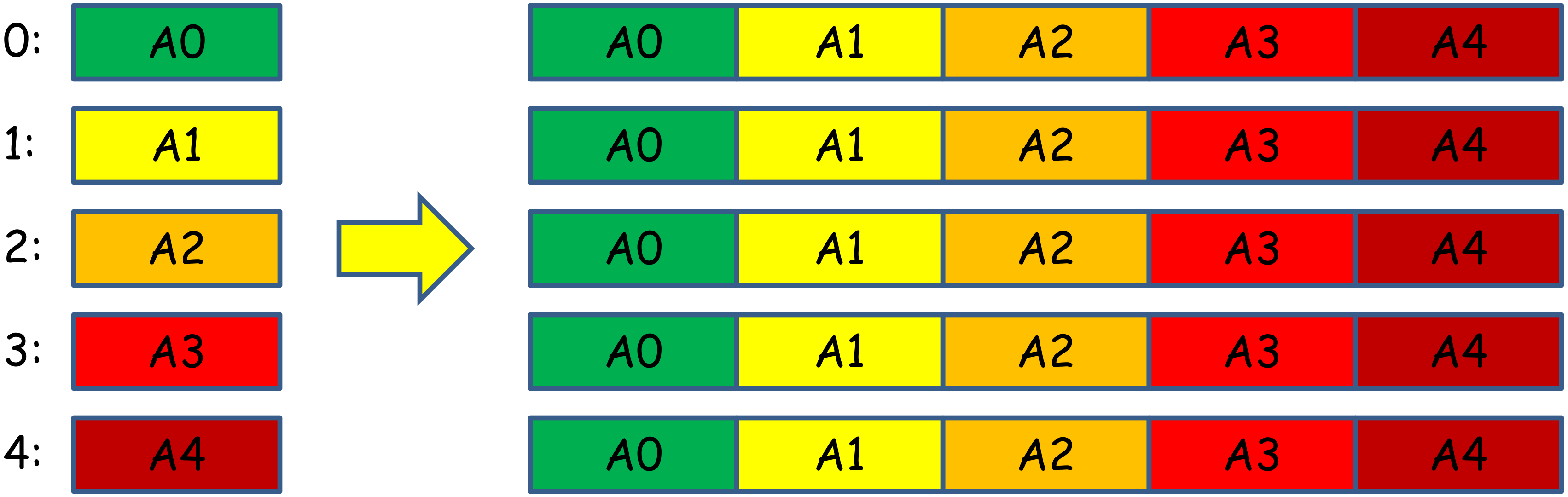
3: A3 A0 A1 A2 A3 A4

4: A4 A0 A1 A2 A3 A4

Semantics: each process contributes a block of data to rbuf at all processes, blocks end up stored consecutively in rank order

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

```
MPI_Allgather(sbuf,scount,stype,rbuf,rcount,rtype,comm);
```



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

aka all-to-all broadcast, all processes get result of gather

MPI_IN_INPLACE can be used for sbuf to indicate that local part of result is already „in place"

```
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) { // 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 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, and that indeed MPI_Allgather always (all other things being equal) performs better than MPI_Gather+MPI_Bcast

Golden rule:
Use collectives for conciseness and performance whereever possible

Complain to MPI library implementer, if performance anomalies are discovered

Example: parallel matrix-vector multiplication

nxn matrix M, n vector V, compute product n vector

W = MxV

where $W[j] = \sum(0\leq j<n): M[j][i]*V[i]$

Takes O(n^2) operations (sequential work)

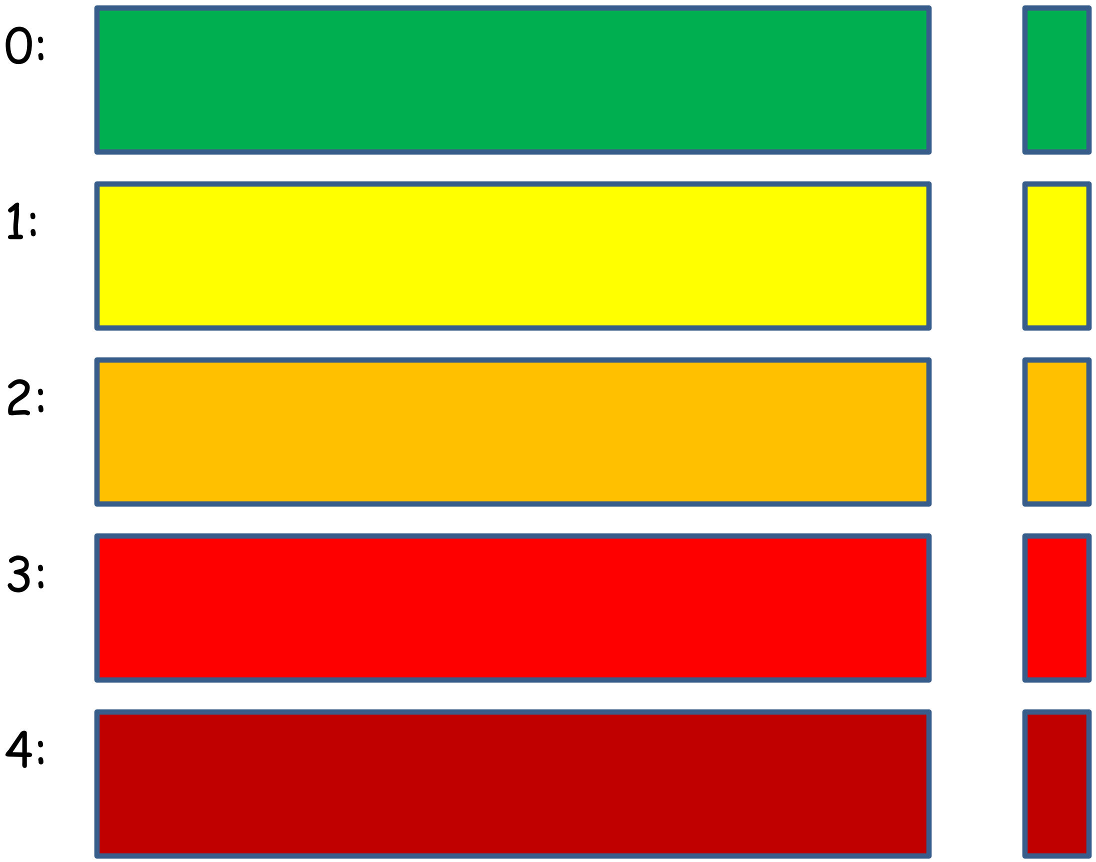
Both M and V should be distributed evenly over the MPI processes; result vector W should be distributed as V

Solution 1: Matrix-vector multiplication

Assume p divides n, distribute M row-wise, each process has n/p rows of M, n/p elements of V

Distribution

local M, V



0:

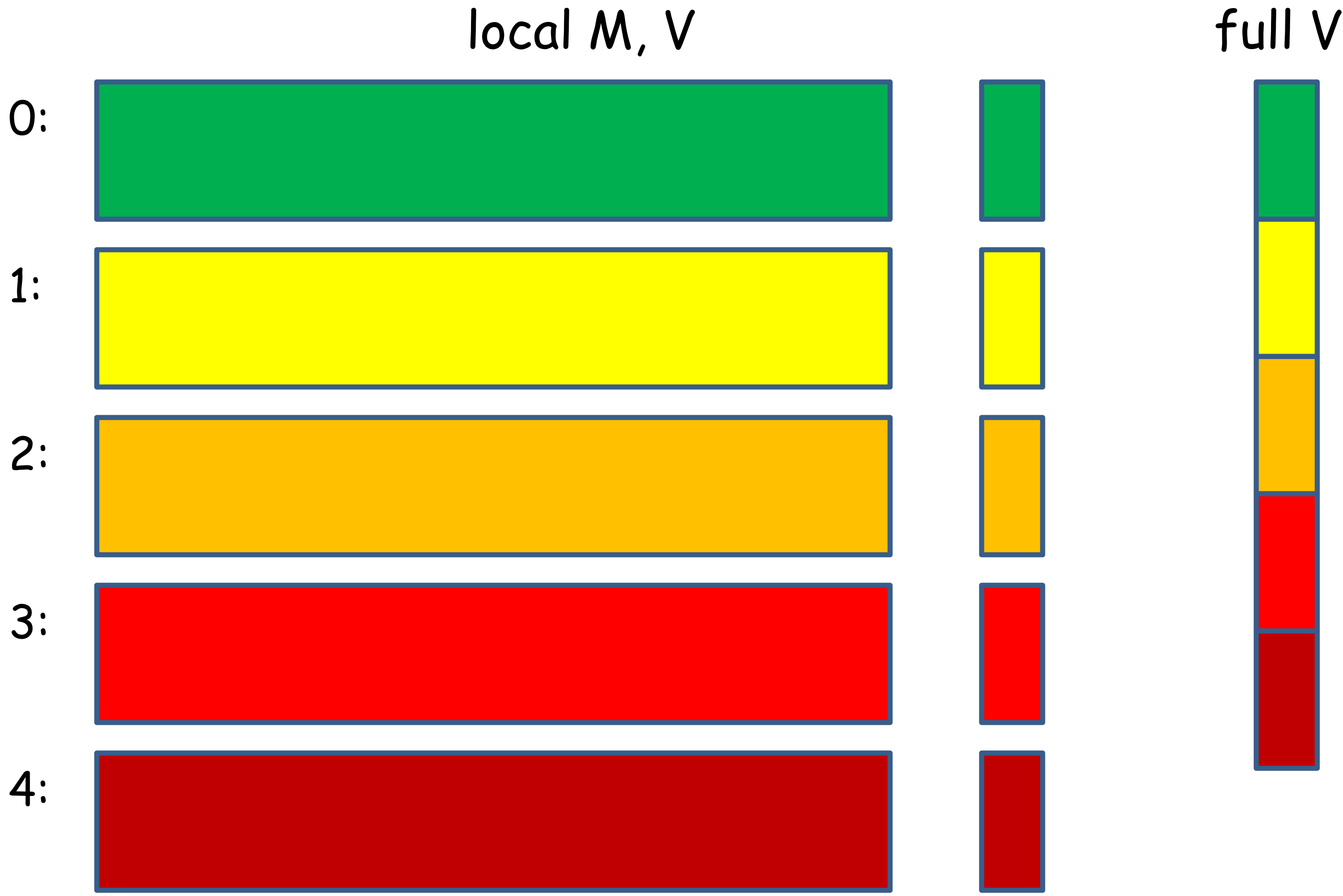
1:

2:

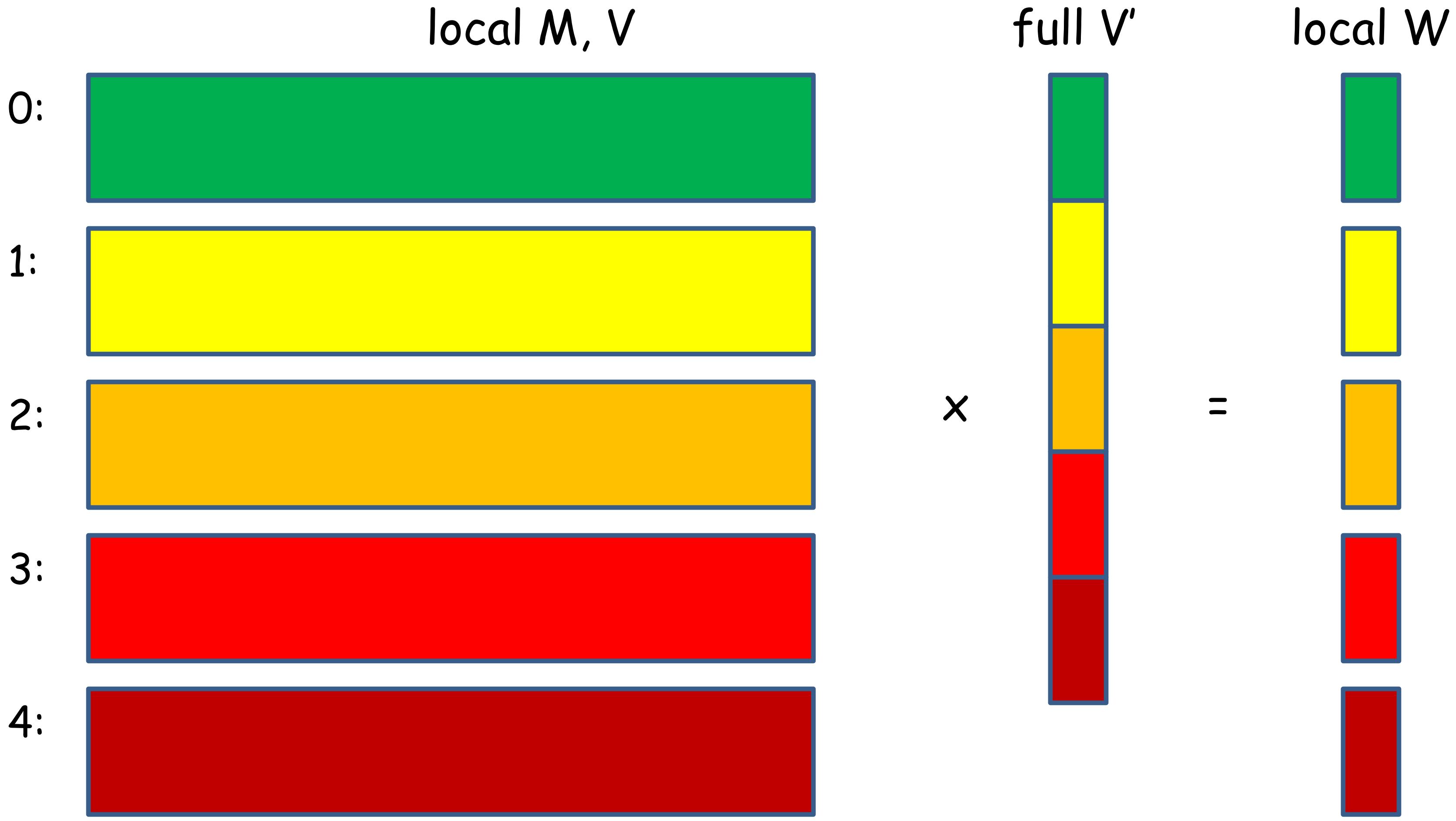
3:

4:

Step 1: gather V at all processes   MPI_Allgather



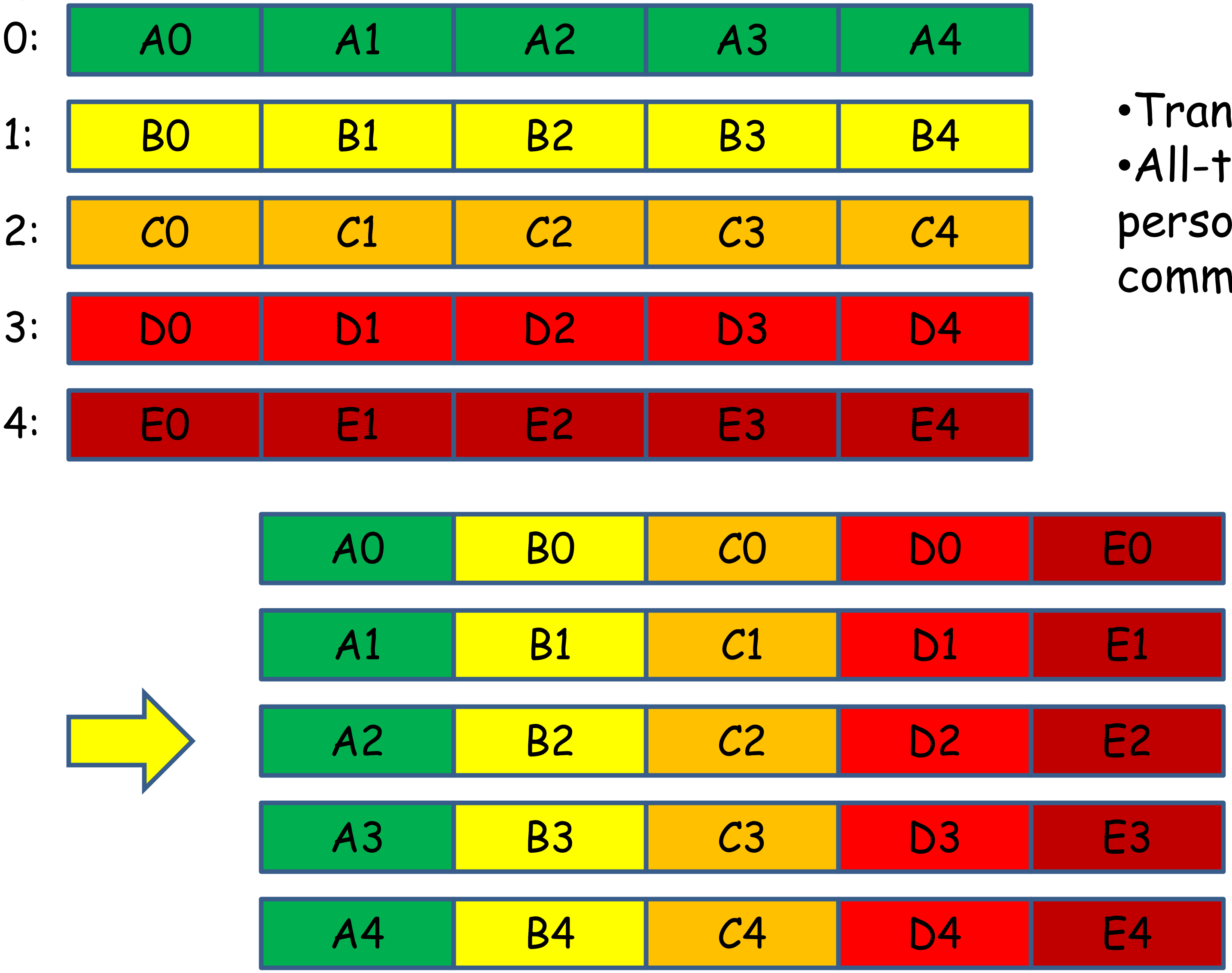
Step 2: locally compute MxV'   in parallel



$O(n^2/p)$ work for local multiplication, assuming MPI_Allgather can be done in $O(n+\log p)$ gives total parallel time $O(n^2/p+n)$

Linear speedup for $p \leq n$

```
MPI_Alltoall(sbuf,scount,stype,rbuf,rcount,rtype,comm);
```



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

- Transpose
- All-to-all personalized communication

```
MPI_Alltoall(sbuf,scount,stype,rbuf,rcount,rtype,comm);
```

Semantics: each process contributes an individual (personalized) block of data to each other process

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

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

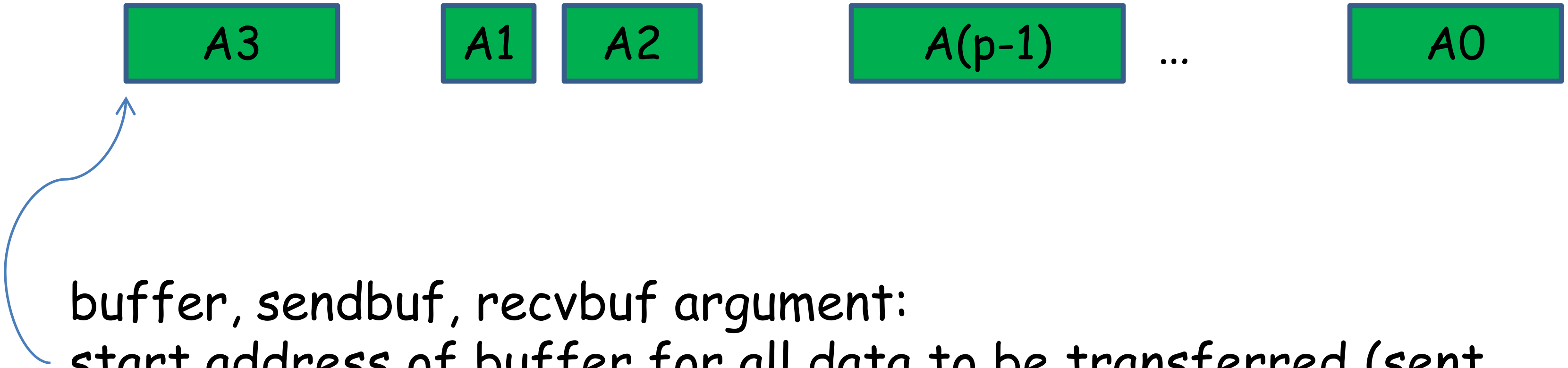
Irregular (vector, v-) collectives:
Possibly different amounts of data destined to different processes

- 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)

Irregular collectives

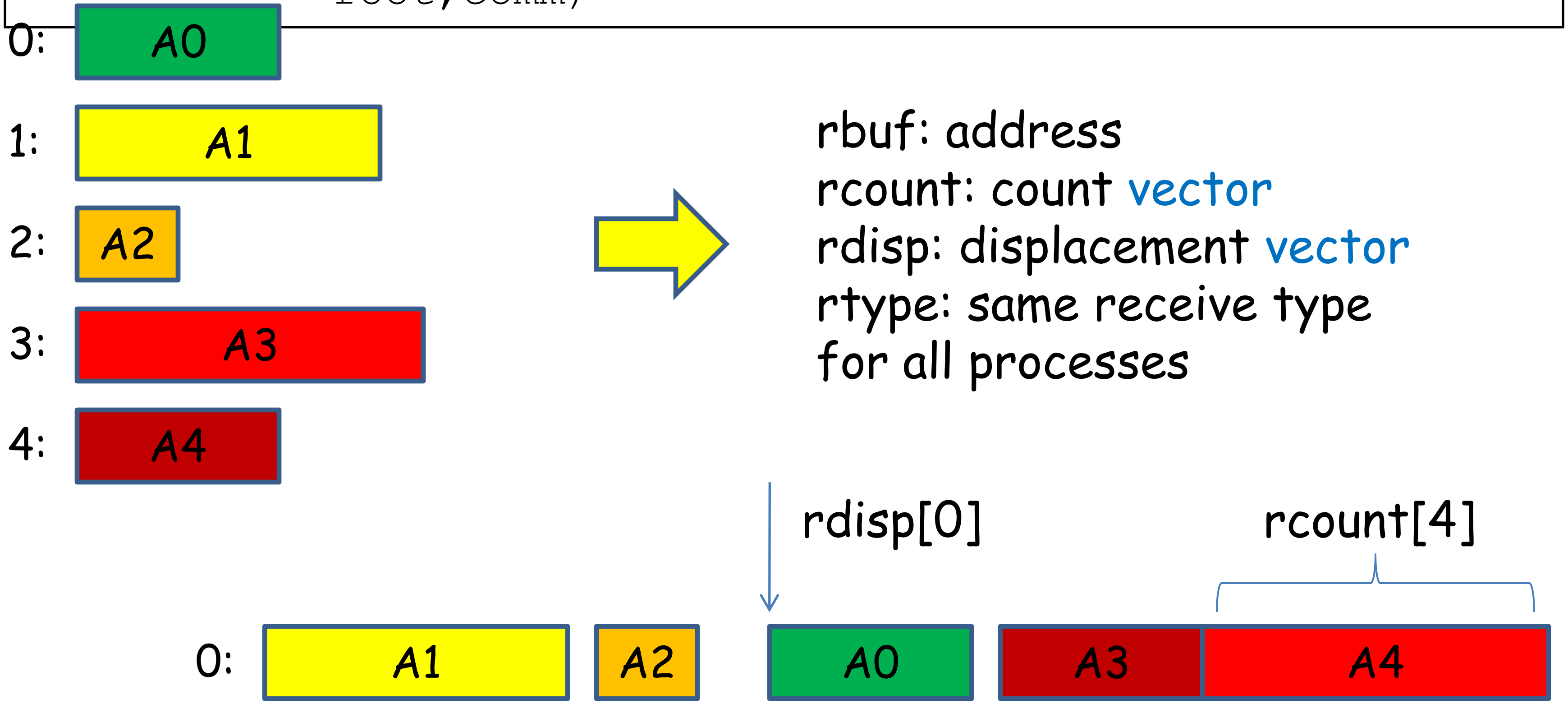


buffer, sendbuf, recvbuf argument:
start address of buffer for all data to be transferred (sent or received)

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(sbuf,scount,stype,rbuf,rcount,rdisp,rtype,
                    root,comm)
```



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

Received data must not overlap. Displacement significant only at root. Size/signature match pairwise

Example: root gathers unknown amount of data from all processes

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

Array of receive counts for all processes

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

Will not work if root does not know scount of other processes.

MPI_Gatherv requires that rcount[i] equals scount of process i (if stype and rtype are same)

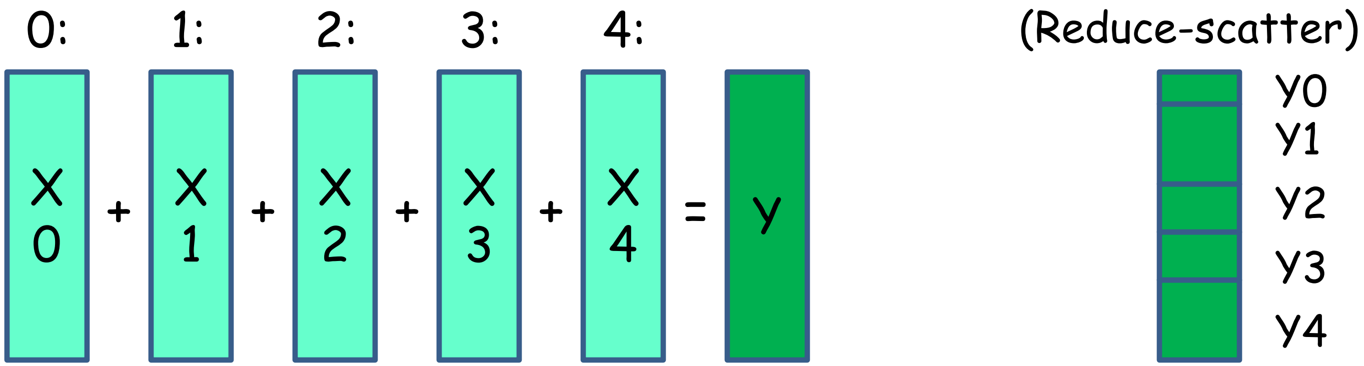
Example: root gathers unknown amount of data from all processes

```
if (rank==root) {
  MPI_Gather(scount,1,MPI_INT,rcounts,1,MPI_INT,comm);
  // compute displacements
  MPI_Gatherv(sbuf,…rbuf,rcounts,rdisp,…,comm);
} else {
  MPI_Gather(scount,1,MPI_INT,rcounts,1,MPI_INT,comm);
  MPI_Gatherv(sbuf,scount,…,comm);
}
```

Use regular MPI_Gather to gather rcount vector: each process transmits its scount to root

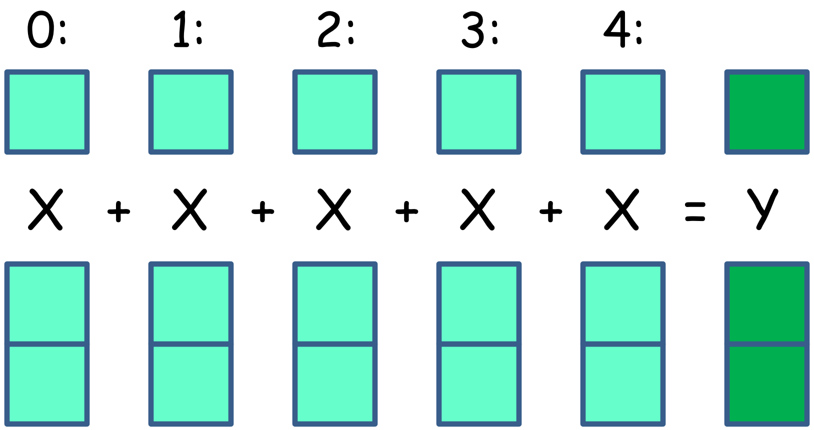
Then correct MPI_Gatherv call can be set up

Reduction collectives



- Each process has vector of data X (same size, same signature)
- Associative operation + (MPI builtin, MPI_SUM,…, or user def)
- Reduction result Y=X0+X1+X2+ … + X(p-1) is stored at
- Root – MPI_Reduce
- All processes – MPI_Allreduce
- Scattered in blocks (Y0 to 0, Y1 to 1, …) – MPI_Reduce_Scatter

Reductions are performed elementwise on the input vectors



Binary operation „+“ is required (and assumed by MPI implementation) to be associative:

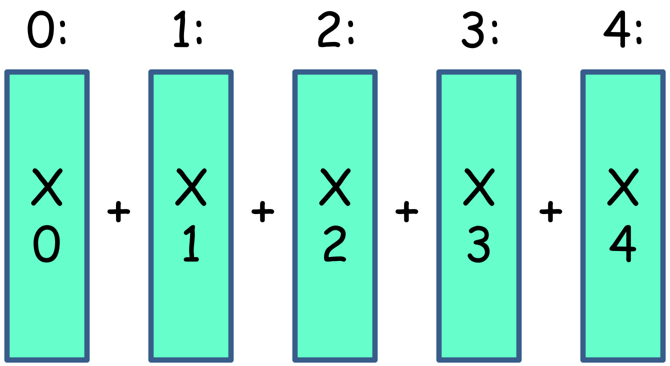
X1 + (X2 + (X3 + (X4 + X5))) = (X1+X2)+(X2+(X3+X4)) = X1 + X2 + X3 + X4 + X5

By associativity: Result independent of „bracketing“, partial results Xi+…Xj can be computed in parallel

If operation is commutative, this can also be exploited

Note: MPI functions are not mathematical functions, i.e. not really commutative (MPI_FLOAT) – good MPI implementations are careful with exploiting commutativity

Scan collectives



- Each process has vector of data X (same size, same signature)
- Associative operation + (MPI builtin, MPI_SUM,…, or user def)
- All prefix sums Yi=X0+…+Xi are computed and stored
- Yi at rank i – MPI_Scan
- Yi at rank i+1 – MPI_Exscan (rank 0 undefined)

```
MPI_Reduce(sendbuf,recvbuf,count,type,op,root,comm);
```

```
MPI_Allreduce(sendbuf,recvbuf,count,type,op,comm);
```

```
MPI_Reduce_scatter(sendbuf,recvbuf,counts,type,op,comm);
```

Here: counts is a vector

`MPI_IN_PLACE` can be used for sendbuf (on root), operand taken from recvbuf

```
MPI_Exscan(sendbuf,recvbuf,count,type,op,root,comm);
```

```
MPI_Scan(sendbuf,recvbuf,count,type,op,root,comm);
```

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

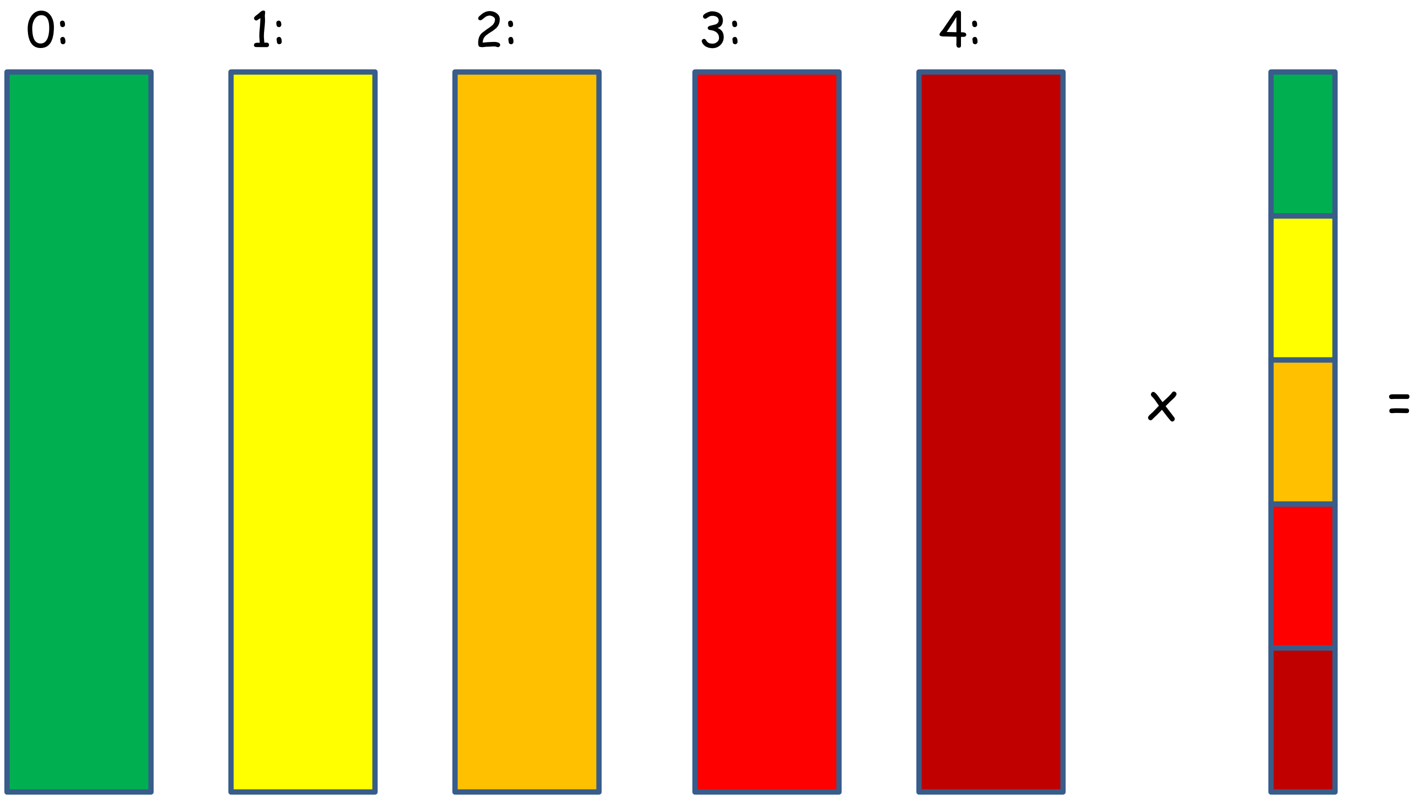
```
MPI_Op_create(MPI_User_function *function,
              int commutative, MPI_Op *op);
```

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

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

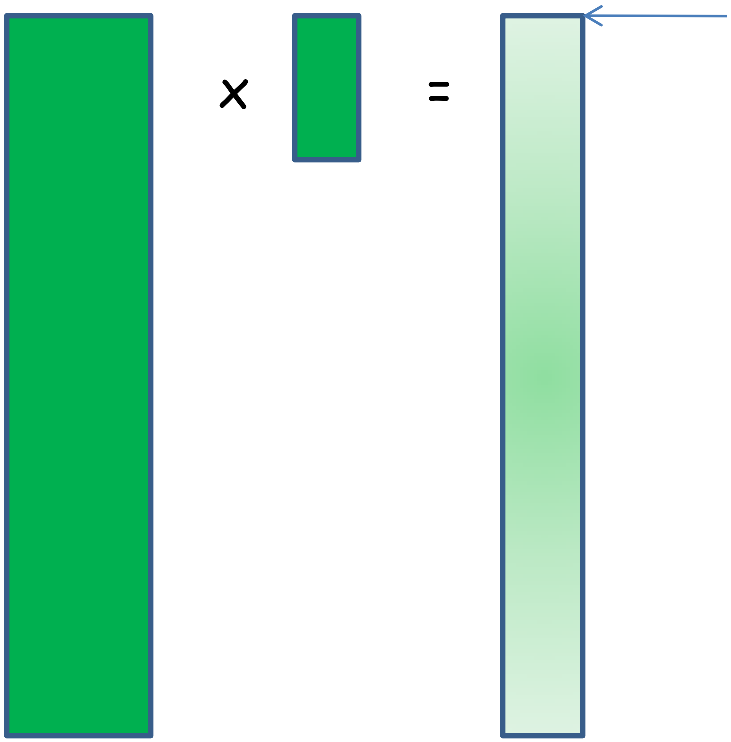
And free it again after use…

Solution 2: Matrix-vector multiplication



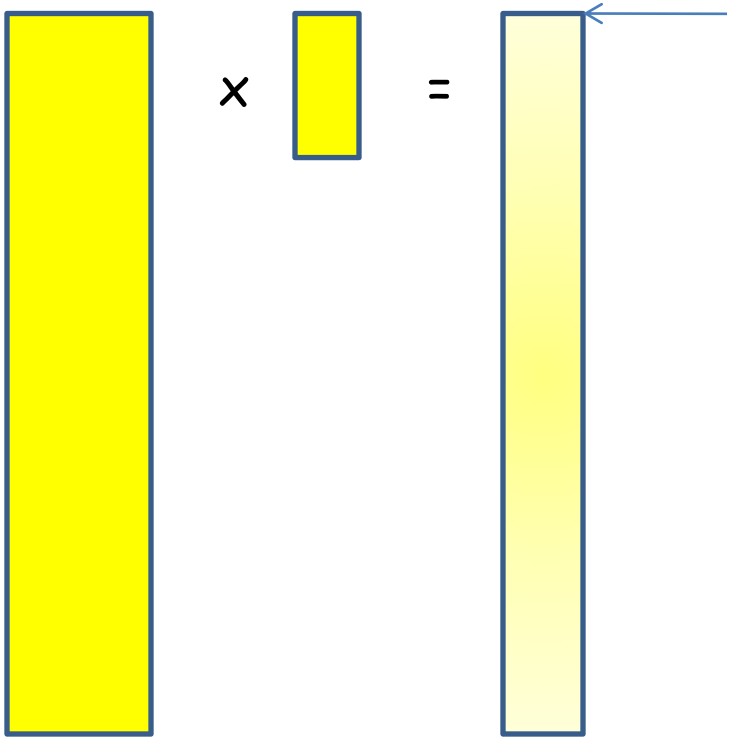
Each rank has n/p columns of (nxn) matrix, n/p rows of vector

1. Locally compute (nxn/p) matrix n/p vector product



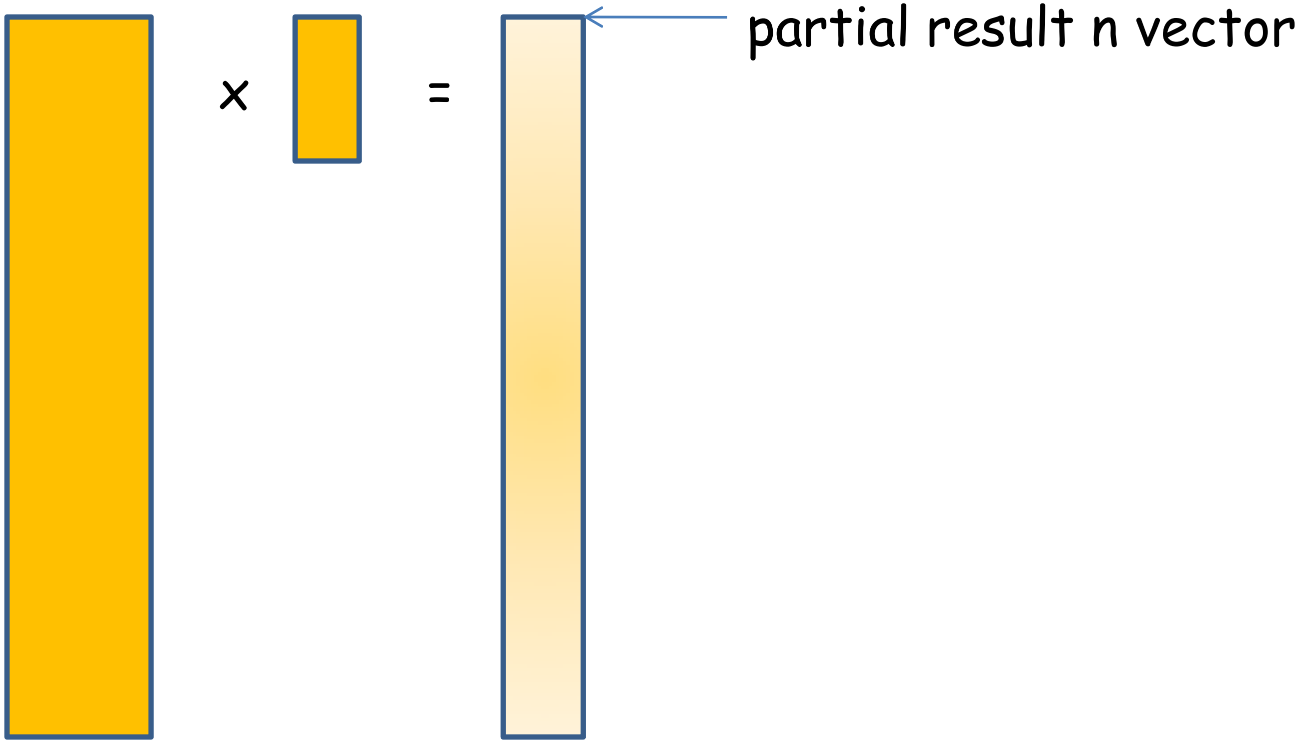
partial result n vector

1. Locally compute (nxn/p) matrix n/p vector product

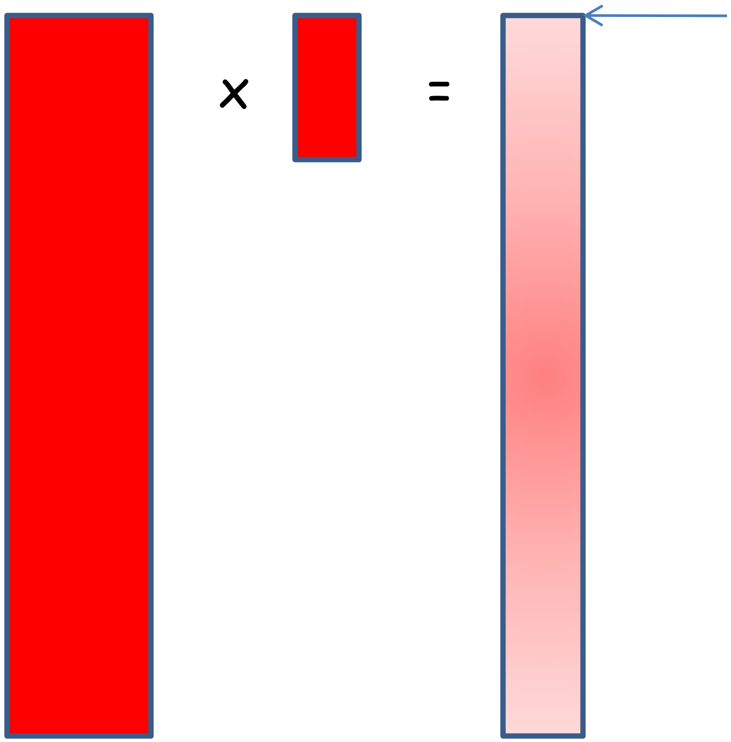


partial result n vector

1. Locally compute (nxn/p) matrix n/p vector product

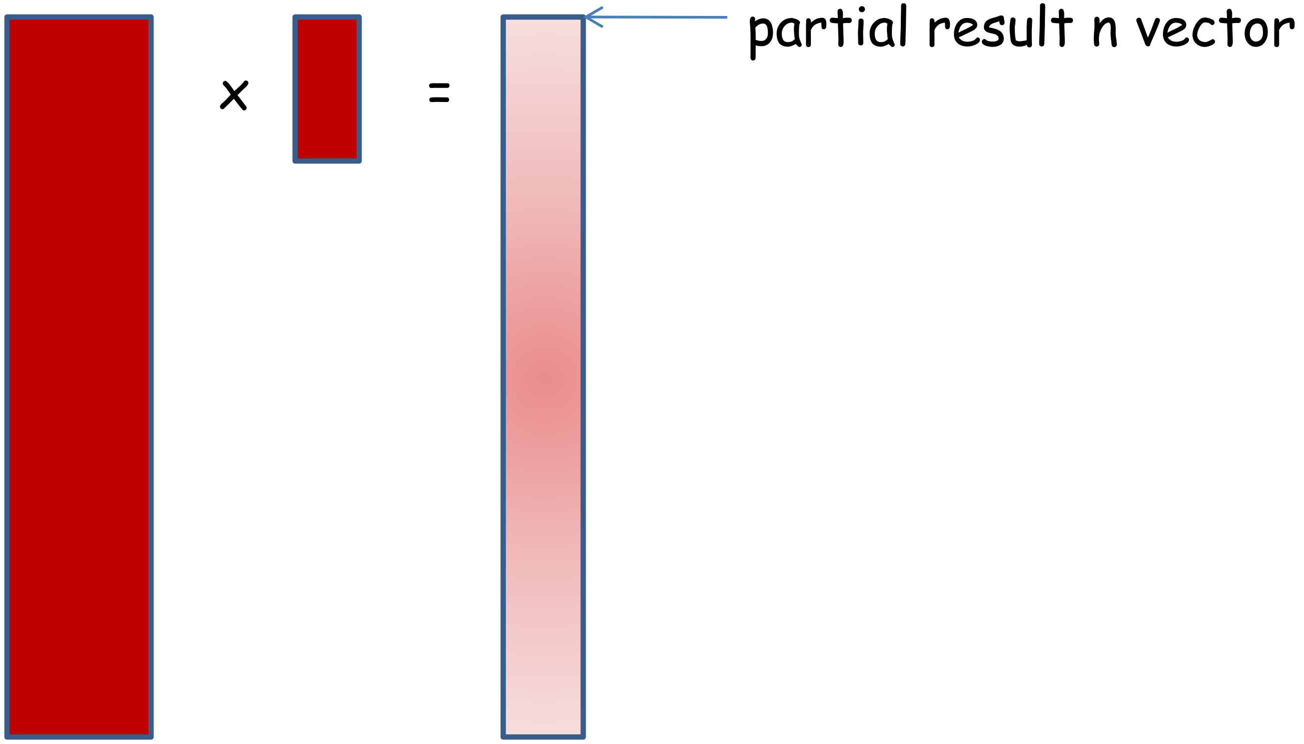


1. Locally compute (nxn/p) matrix n/p vector product

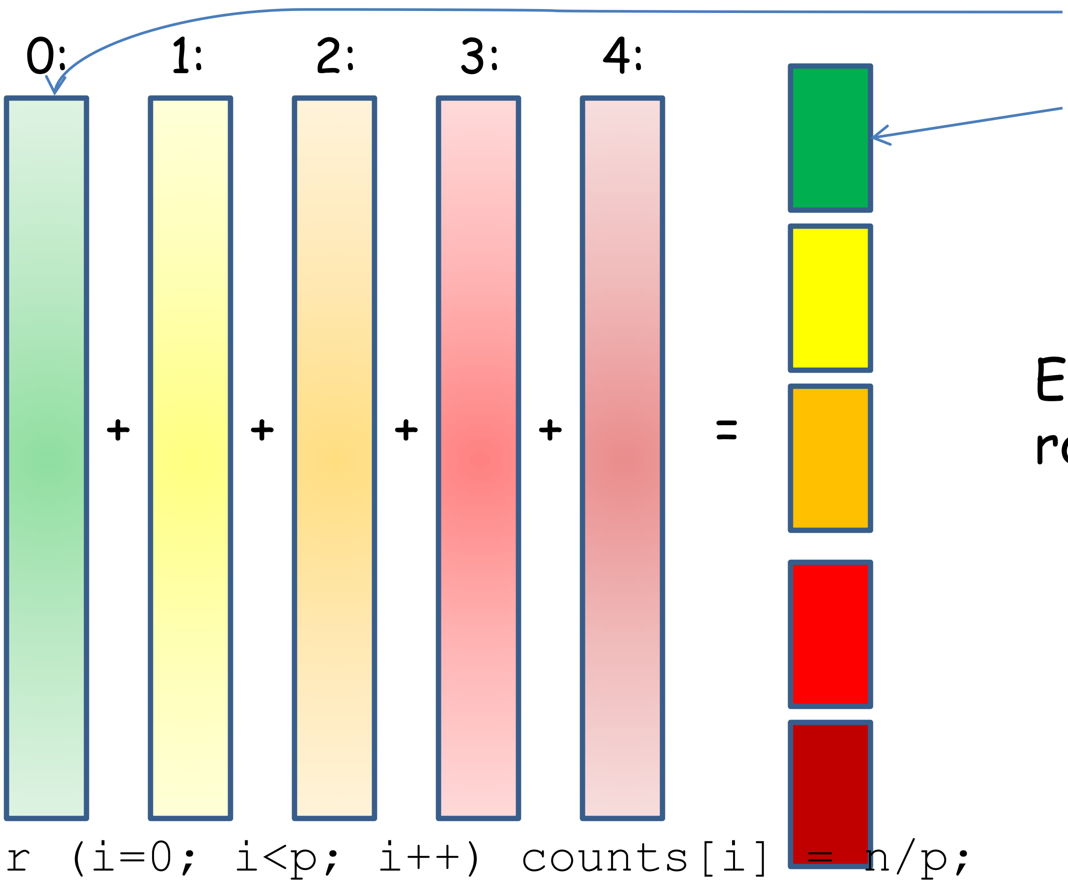


partial result n vector

1. Locally compute (nxn/p) matrix n/p vector product



2. Sum partial result n vectors and scatter n/p blocks



partial result buffer

(local) result buffer

Each rank stores n/p rows of result vector

```
for (i=0; i<p; i++) counts[i] = n/p;
MPI_Reduce_scatter(partial,result,counts,MPI_FLOAT,
                   MPI_SUM,comm);
```

O(n^2/p) work for local multiplication, assuming MPI_Reduce_scatter can be done in O(n+log p) gives total parallel time O(n^2/p+n)

Linear speedup for p≤n

Exercise:
Which method is better?