High Performance Computing
Advanced MPI: Topologies

Jesper Larsson Träff
traff@par. …
Institute of Computer Engineering, Parallel Computing, 191-4
Treitlstrasse 3, 5. Stock (DG)

High Performance Computing: MPI topics and algorithms

- "Advanced" features of MPI (Message-Passing Interface): Collective operations, non-blocking collectives, sparse collectives, datatypes, one-sided communication, process topologies, MPI I/O, …
- Efficient implementation of MPI collectives: Algorithms under (simplified) network assumptions
- Useful for implementation of own algorithms, libraries, etc.

MPI: the "Message-Passing Interface"

- Library with C and Fortran bindings that implements a message-passing model
- Current *de facto* standard in HPC and distributed memory parallel computing

PGAS competitors:
UPC, CaF, OpenSHMEM, …

MPI: the "Message-Passing Interface"

- Library with C and Fortran bindings that implements a message-passing model

- Current *de facto* standard in HPC and distributed memory parallel computing

Open source implementations:

- `mpich` from Argonne National Laboratory (ANL)
- `mvapich` from Ohio State
- OpenMPI, community effort

from which many special purpose/vendor implementations are derived

mpich(2) canonical references

William Gropp, Ewing L. Lusk, Nathan E. Doss, Anthony Skjellum: A High-Performance, Portable Implementation of the MPI Message Passing Interface Standard. Parallel Computing 22(6): 789-828 (1996)
William Gropp, Ewing L. Lusk: Sowing mpich: a Case Study in the Dissemination of a Portable Environment for Parallel Scientific Computing. IJHPCA 11(2): 103-114 (1997)
William Gropp, Ewing L. Lusk: A High-Performance MPI Implementation on a Shared-Memory Vector Supercomputer. Parallel Computing 22(11): 1513-1526 (1997)

Rajeev Thakur, Rolf Rabenseifner, William Gropp: Optimization of Collective Communication Operations in MPICH. IJHPCA 19(1): 49-66 (2005)

Open MPI canonical references

Richard L. Graham, Brian Barrett, Galen M. Shipman, Timothy S. Woodall, George Bosilca: Open MPI: a High Performance, Flexible Implementation of MPI Point-to-Point Communications. Parallel Processing Letters 17(1): 79-88 (2007)

Edgar Gabriel, Graham E. Fagg, George Bosilca, Thara Angskun, Jack J. Dongarra, Jeffrey M. Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine, Ralph H. Castain, David J. Daniel, Richard L. Graham, Timothy S. Woodall: Open MPI: Goals, Concept, and Design of a Next Generation MPI Implementation. PVM/MPI 2004: 97-104

Richard L. Graham, Timothy S. Woodall, Jeffrey M. Squyres: Open MPI: A Flexible High Performance MPI. PPAM 2005: 228-239

Message-passing model

- Processors (in MPI: processes – something executed by a physical processor/core) execute program on local data
- Processors exchange data and synchronize by explicit communication; only way to exchange information ("shared nothing")

Program: MIMD, SPMD

Communication:
- Asynchronous (non-blocking) or synchronous (blocking)
- In-order, out of order
- One-to-one, one-to-many, many-to-one, many-to-many

Strict message-passing (theoretical model):

Synchronous, in-order, one-to-one communication: Enforces event order, easier to reason about correctness, mostly deterministic execution, no race-conditions

Hoare: CSP

Practical advantages:

- Enforces locality (no cache sharing, less memory per process)
- Synchronous communication can be implemented space efficiently (no intermediate buffering)

Recent: GO

May: OCCAM

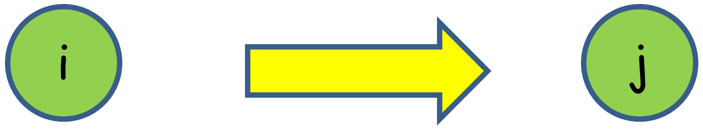
C. A. R. Hoare: Communicating Sequential Processes. Comm. ACM 21(8): 666-677 (1978)

MPI message-passing model

Three main communication models:

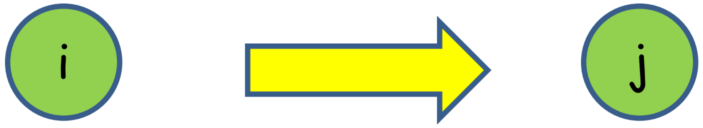
MPI_Send

MPI_Recv



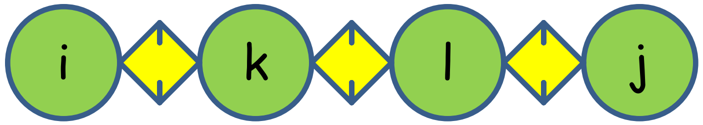
Point-to-point: Two processes explicitly involved

MPI_Put/Get/Accumulate



One-sided: One process explicitly involved

MPI_Bcast



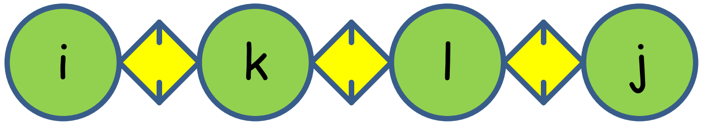
Collective: p≥1 processes explicitly involved

MPI message-passing model

All communication between named processes in same communication domain:

Communication domain (“communicator”): Ordered set of processes that can communicate (e.g., MPI_COMM_WORLD)

Name: For domain of p processes, rank between 0 and p-1

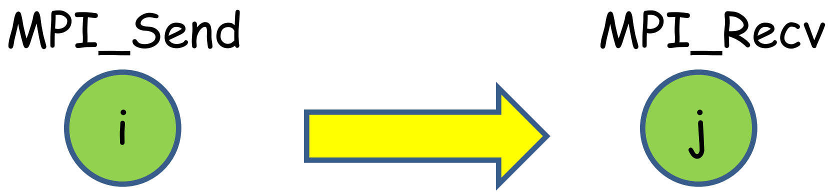


Note:
MPI is a specification; not an implementation (term "MPI library" refers to specific implementation of the standard)

MPI specification/standard prescribes (almost) nothing about the implementation, in particular:

- No performance model (thus no assumptions on communication network)
- No performance guarantees
- No prescribed algorithms (for collective communication, synchronization, datatypes etc.)
- But: Some assumptions about "High Quality implementation"

Point-to-point communication

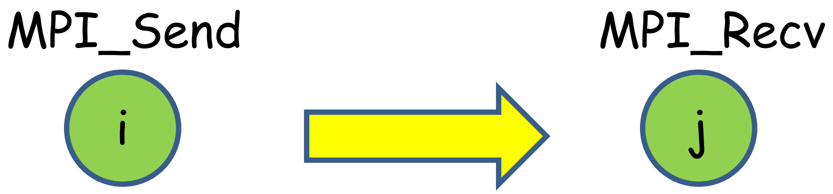


i, j: name ("rank") of process

- Both sender and receiver explicitly involved in communication
- Communication is reliable (implementation and underlying communication infrastructure must guarantee this)
- Communication is ordered: Messages with same destination and same tag arrive in order sent (implementation must guarantee this)

(*) Whether MPI_Send can complete may depend on MPI implementation

Point-to-point communication

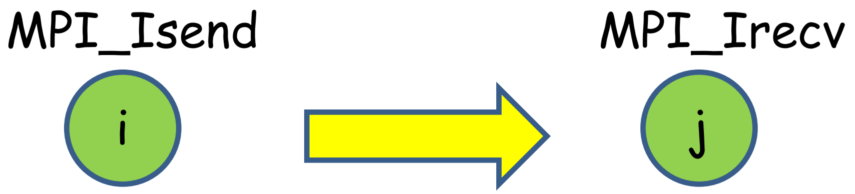


i, j: name ("rank") of process

- Largely asynchronous model: MPI_Send has semi-local completion semantics, when call completes buffer can be reused (*)
- Blocking and non-blocking ("immediate") semantics

Blocking MPI call:
Returns when operation locally complete, buffer can be reused, may or may not depend on actions of other processes, no guarantee regarding state of other processes

Point-to-point communication

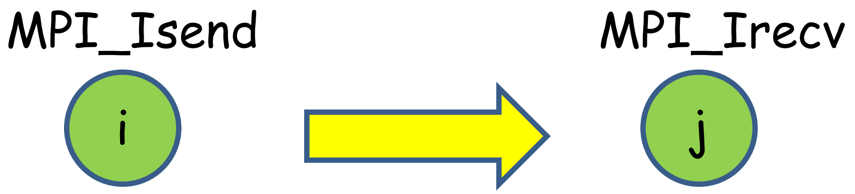


i, j: name ("rank") of process

- Largely asynchronous model: MPI_Send has semi-local completion semantics, when call completes buffer can be reused
- Blocking and non-blocking ("immediate") semantics

Non-blocking MPI call:
Returns immediately, independent of actions of other processes, buffer may be in use. Completion as in blocking call must be enforced (MPI_Wait, MPI_Test, …)

Point-to-point communication

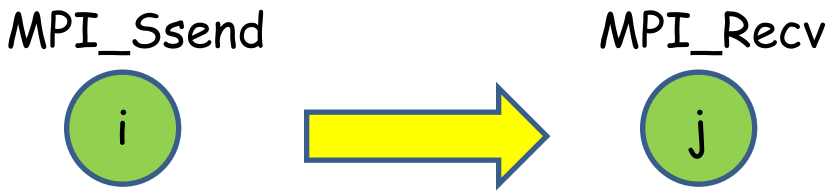


i, j: name ("rank") of process

- Largely asynchronous model: MPI_Send has semi-local completion semantics, when call completes buffer can be reused
- Blocking and non-blocking ("immediate") semantics

Terminology note:
Non-blocking in MPI is not a non-blocking progress condition, but means that the calling process can continue; progress of communication eventually depends on progress of other process

Point-to-point communication

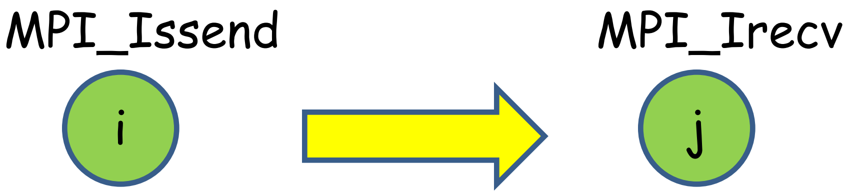


i, j: name ("rank") of process

- Largely asynchronous model: MPI_Send has semi-local completion semantics, when call completes buffer can be reused
- Blocking and non-blocking ("immediate") semantics
- Many send "modes": MPI_Ssend (MPI_Bsend, MPI_Rsend)

```
int x[count];
MPI_Ssend(x,count,MPI_INT,dest,tag,comm);
```

Point-to-point communication

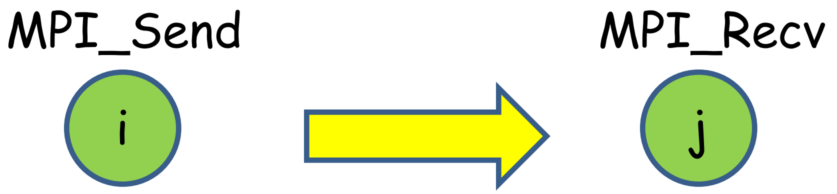


i, j: name ("rank") of process

- Largely asynchronous model: MPI_Send has semi-local completion semantics, when call completes buffer can be reused
- Blocking and non-blocking ("immediate") semantics
- Many send "modes": MPI_Ssend (MPI_Bsend, MPI_Rsend)

```
int x[count];
MPI_Issend(x,count,MPI_INT,dest,tag,comm,
            &request);
```

Point-to-point communication

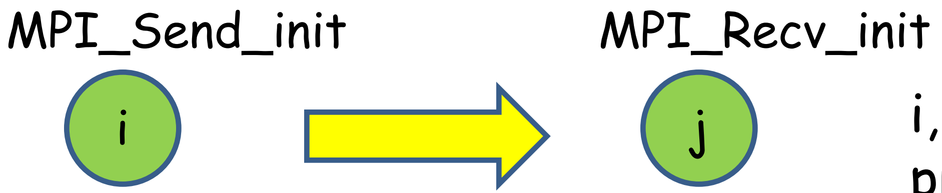


i, j: name ("rank") of process

- Largely asynchronous model: MPI_Send has semi-local completion semantics, when call completes buffer can be reused
- Blocking and non-blocking ("immediate") semantics
- Non-determinism only through "wildcards" in MPI_Recv

```
int x[count];
MPI_Recv(x,count,MPI_INT,MPI_ANY_SOURCE,
         MPI_ANY_TAG,comm,&status);
```

Point-to-point communication

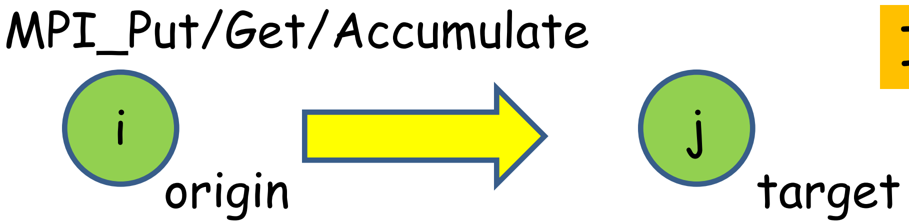


i, j: name ("rank") of process

- Persistent versions of all blocking point-to-point operations
- Optimization potential: All parameters bound at initialization call (in MPI_Request object), can be reused, algorithm can be adapted to use case (parameters, process mapping, etc.)

```
MPI_Request request[2];
MPI_Recv_init(x,count,MPI_INT,MPI_ANY_SOURCE,
              MPI_ANY_TAG,comm,&request[0]);
MPI_Ssend_init(y,count,MPI_INT,j,TAG,comm,
               &request[1]);
MPI_Startall(2,request);
```

One-sided communication



Introduced with MPI 2.0

News in MPI 3.0

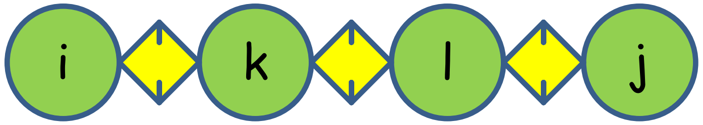
- Only one process explicitly involved in communication; origin supplies all information

- Communication is reliable, but not ordered
- Non-blocking communication, completion at synchronization point
- Different types of synchronization (active/passive; collective/localized)

- Target memory in exposed/accessible window

Collective communication

MPI_Bcast



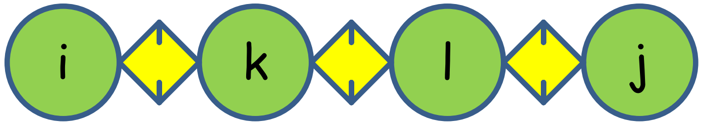
News in MPI 3.0

- All processes explicitly involved in data exchange or compute operation (one-to-many, many-to-one, many-to-many)
- All collective operations blocking in the MPI sense: Return when operation is locally complete, buffers can be reused
- Operations are reliable
- Collectives (except for MPI_Barrier) are not synchronizing

```
float *buffer = malloc(count*sizeof(float));
MPI_Bcast(buffer,count,MPI_FLOAT,root,comm);
```

Collective communication

MPI_Bcast



Synchronizing: MPI_Barrier

Exchange, regular, rooted

MPI_Bcast
MPI_Gather/MPI_Scatter

Reduction, regular, rooted

MPI_Reduce

Non-rooted (symmetric)

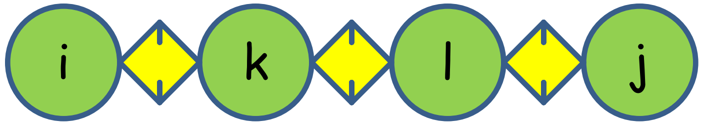
MPI_Allgather
MPI_Alltoall

Non-rooted (symmetric)

MPI_Allreduce
MPI_Reduce_scatter_block
MPI_Scan/MPI_Exscan

Collective communication

MPI_Bcast



Exchange, irregular (vector)

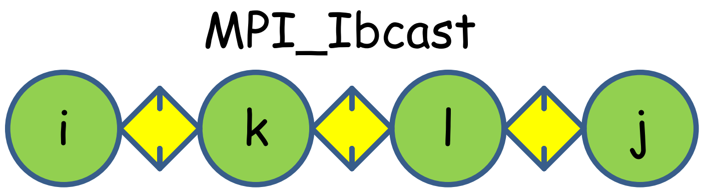
MPI_Gatherv/MPI_Scatterv
MPI_Allgatherv
MPI_Alltoallv/MPI_Alltoallw

Reduction, irregular (vector)

MPI_Reduce_scatter

Irregular collective: Size and basetype (technically: type signature) of buffers must match pairwise between processes, but can differ between different pairs

Collective communication

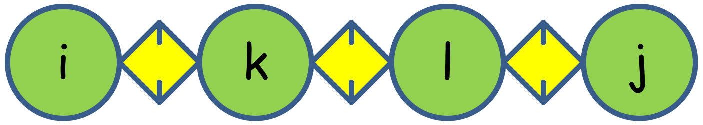


With MPI 3.0

- All 17 collectives in non-blocking versions
- Call returns immediately, independent of actions of other processes
- Completion with same semantics as blocking counterpart must be explicitly enforced (MPI_Wait, MPI_Waitall, …)

Collective communication

MPI_Bcast_init



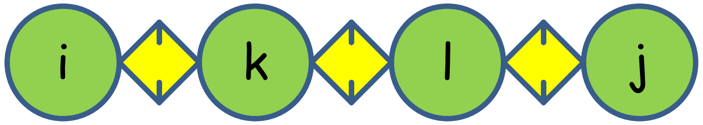
With MPI 4.0

- All 17 collectives in persistent versions
- Optimization potential: All parameters bound at initialization call (in MPI_Request object), can be reused, algorithm can be adapted to use case (parameters, process mapping, etc.)

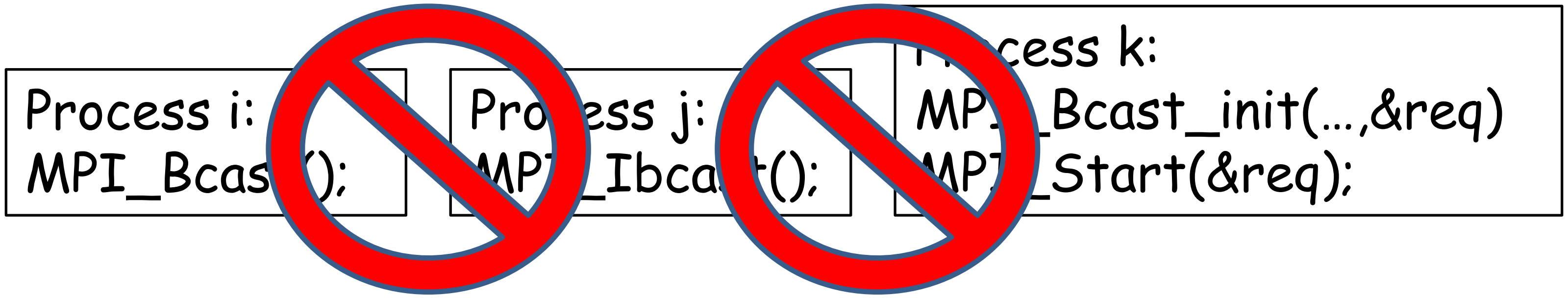
```
MPI_Barrier_init(comm,MPI_INFO_NULL,&request);
while (1) {
  MPI_Start(&request); // start the barrier
  MPI_Wait(&request);  // now enforce the barrier
}
MPI_Request_free(&request);
```

Collective communication

MPI_Bcast, MPI_Ibcast, MPI_Bcast_init



- Although semantically equivalent, blocking, non-blocking, and persistent collective cannot be used together
- Different from persistent point-to-point: modes can be mixed



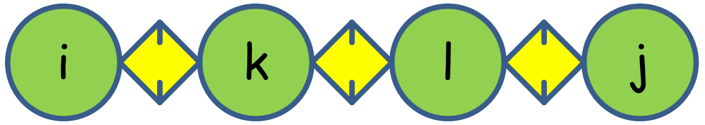
```
Process i:
MPI_Bcast();
```

```
Process j:
MPI_Ibcast();
```

```
Process k:
MPI_Bcast_init(…,&req)
MPI_Start(&req);
```

Collective communication

MPI_Neighbor_alltoall



With MPI 3.0 and MPI 4.0

- 5 sparse (neighbor) collective operations, in blocking, non-blocking and persistent versions

Non-rooted (symmetric)

MPI_Neighbor_allgather
MPI_Neighbor_alltoall

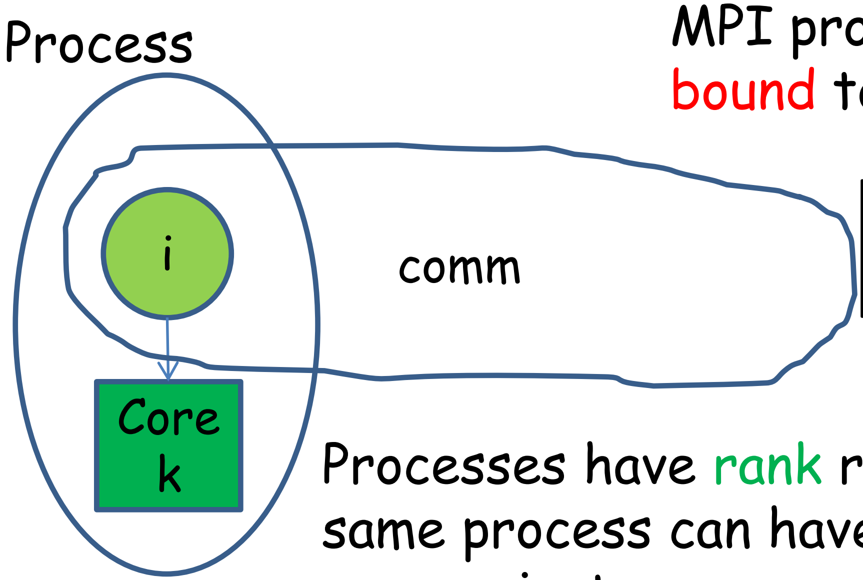
Exchange, irregular (vector)

MPI_Neighbor_allgatherv
MPI_Neighbor_alltoallv
MPI_Neighbor_alltoallw

All models

Communication is wrt. communicator: Ordered set of processes, mapped to physical nodes/processors/cores

In one-sided model, communicator is part of memory-window

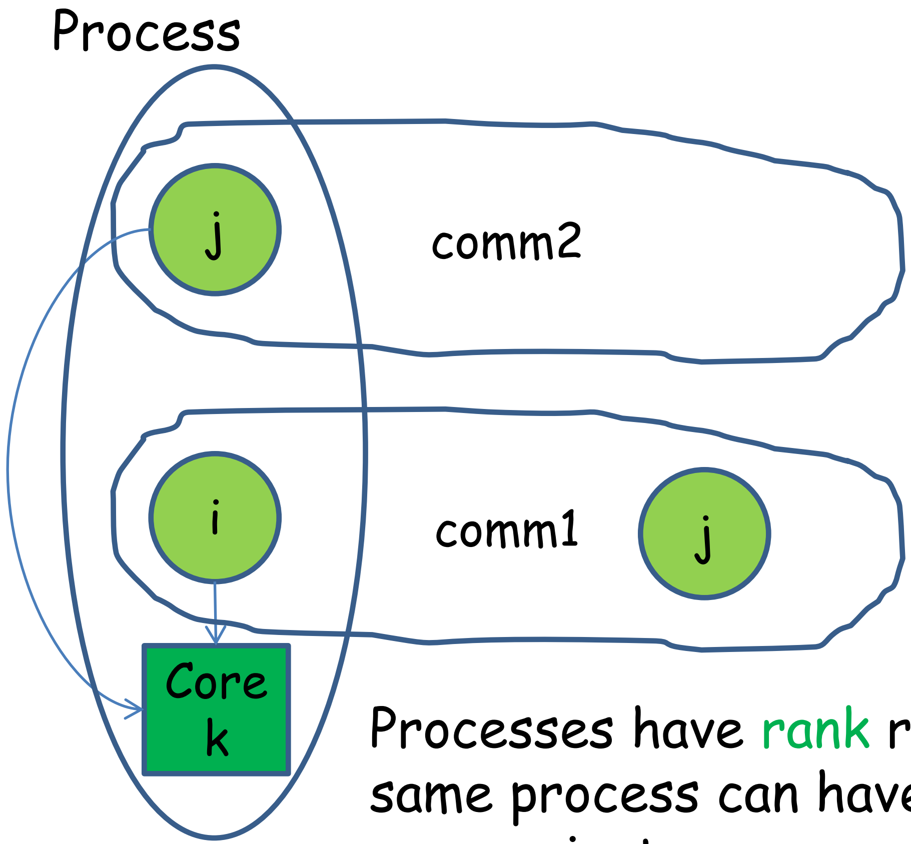


MPI processes (mostly) statically bound to processors/cores

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

Processes have rank relative to communicator; the same process can have different ranks in different communicators

MPI has multi-purpose (collective) operations for creating new communicators out of old ones. If a different mapping is needed, new communicator must be created (MPI objects are static)



Process with rank j in comm1 may have to send state/data to the process with rank j in comm2

Processes have rank relative to communicator; the same process can have different ranks in different communicators

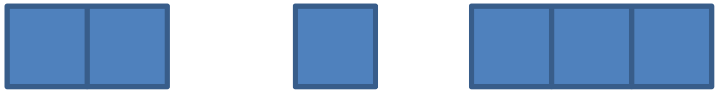
For discussion on collective interfaces, sparse (neighbor) and global communication, and the relation to communicators, see

Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, Daniel J. Holmes: MPI collective communication through a single set of interfaces: A case for orthogonality. Parallel Comput. 107: 102826 (2021)

Many interesting things to do for (paid) Master's thesis…

All models

- Data in buffers can be arbitrarily structured, not necessarily only consecutive elements
- Data structure/layout communicated to MPI implementation by datatype handle
- Communication buffers: Buffer start (address), element count, element datatype



e.g. indexed datatype

MPI terms: Derived datatypes, user-defined datatypes, … (MPI 3.1 standard, Chapter 4)

Beyond message-passing MPI features

- MPI-IO: Serial and parallel IO, heavy use of derived datatypes in specification
- MPI process management (important concept: intercommunicators): Coarse grained management (spawn, connect) of additional processes
- Topologies: Mapping application communication patterns to communication system
- Tools support: Profiling interface and library information

Concrete example (MPI code recap)

- C code (could also be "mild" C++)
- What to include, what to define
- Using assertions
- Good SPMD style: program works for all numbers of processes
- Compiling and running (environment dependent)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#include <mpi.h>
//#define NDEBUG
#include <assert.h>

#define SRTAG 777

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

  MPI_Init(&argc,&argv);

  // […] Here comes the code

  MPI_Finalize();

  return 0;
}
```

Good practice: Make assertions

MPI library must be initialized

… and finalized

The code, part 1: Two processes communicate (first and last)

```
MPI_Comm_dup(MPI_COMM_WORLD,&comm);
MPI_Comm_rank(comm,&rank);
MPI_Comm_size(comm,&size);
assert(rank<size);

int r;
if (rank==0) {
  MPI_Sendrecv(&rank,1,MPI_INT,size-1,SRTAG,
               &r,   1,MPI_INT,size-1,SRTAG,
               comm,MPI_STATUS_IGNORE);
  assert(r==size-1);
} else if (rank==size-1) {
  MPI_Sendrecv(&rank,1,MPI_INT,0,SRTAG,
               &r,   1,MPI_INT,0,SRTAG,
               comm,MPI_STATUS_IGNORE);
  assert(r==0);
}
```

New comm for this

Assert simple assumptions

Buffers always have this structure: Address,count,type

Good SPMD style: All sizes

The code, part 2: All processes communicate collectively

```
int *allrank;
allrank = (int*)malloc(size*sizeof(int));
for (root=0; root<size; root++) {
  allrank[root] = rank;
  MPI_Bcast(&allrank[root],1,MPI_INT,root,comm);
  assert(allrank[root]==root);
}
int *ar;
ar = (int*)malloc(size*sizeof(int));
MPI_Allgather(&rank,1,MPI_INT,
              ar,   1,MPI_INT,comm);
for (root==0; root<size; root++)
  assert(ar[root]==allrank[root]);

free(allrank);
free(ar);
```

No stack allocation

Buffer structure, but beware: one block

Allgather is Bcast from all

Free after use, don't forget

Running the code (on hydra):

```
> scp mpiintro.c <account>@hydra:~/
> ssh hydra
> module avail
> module load mpi/openmpiS
> mpicc -o intro -O3 mpiintro.c
> srun -N 10 --tasks-per-node=32 ./intro
>
```

Now on hydra

See doku.par.tuwein.ac.at for more

Example: Three algorithms for matrix-vector multiplication

mxn matrix A and n-element vector x distributed evenly across p MPI processes: compute

$y = Ax$

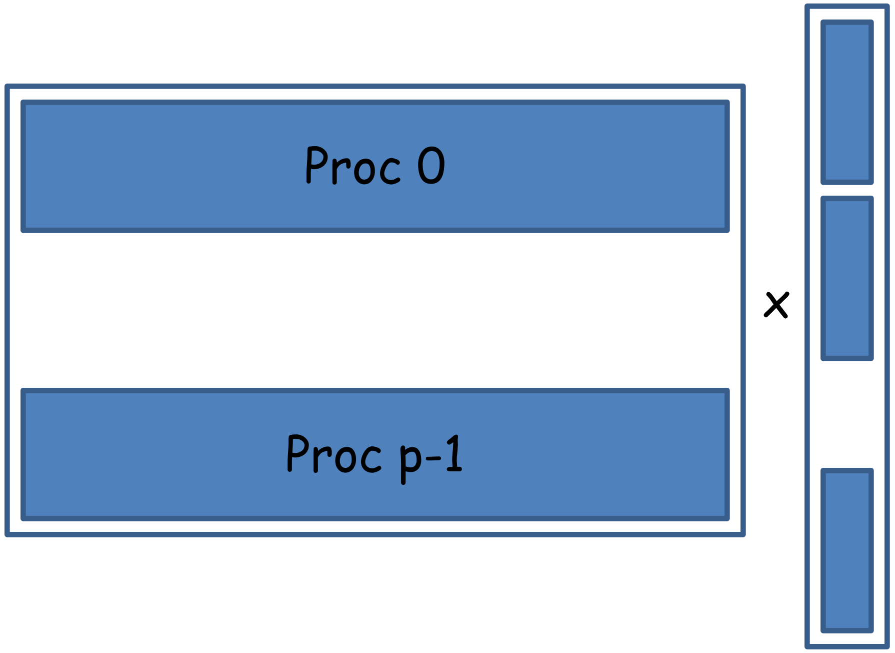
with y the m-element result vector

Even distribution:

- Each of p processes has an mn/p element submatrix, an n/p element subvector, and computes an m/p element result vector.
- Algorithms should respect/preserve distribution

$$\begin{pmatrix} y0 \\ y1 \\ y2 \end{pmatrix} = \begin{pmatrix} A0 \\ A1 \\ A2 \end{pmatrix} \begin{pmatrix} x0 \\ x1 \\ x2 \end{pmatrix}$$

Ai: (m/p) x n matrix



Algorithm 1:

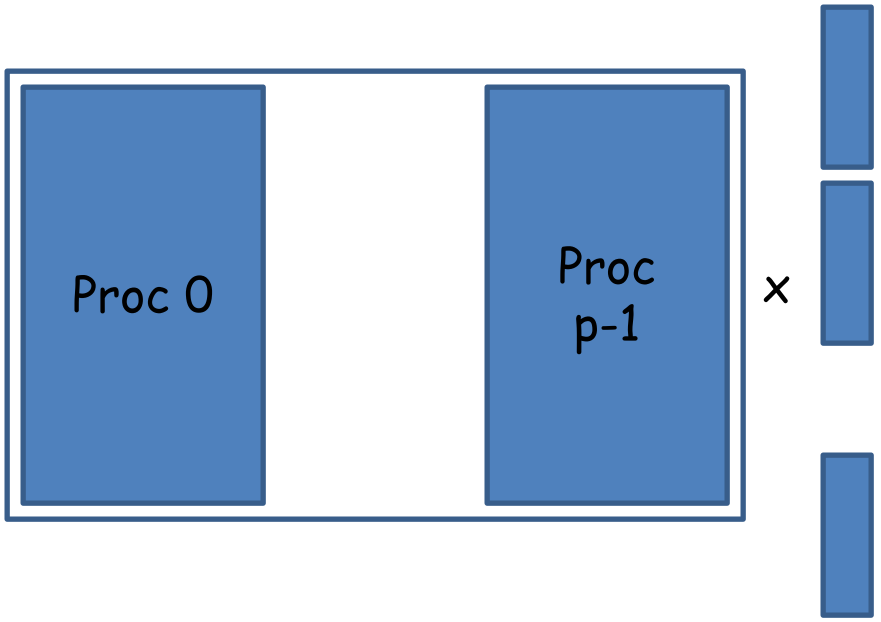
- Row-wise matrix distribution
- Each process needs full vector: MPI_Allgather(v)
- Compute blocks of result vector locally

Conjecture: $T(m,n,p) = O((m/p)n + n + \log p)$, $p \leq m$

Why?

$$\begin{pmatrix} y0 \\ y1 \\ y2 \end{pmatrix} = A0x0 + A1x1 + A2x2$$

A = (A0 | A1 | A2), Ai m x (n/p) matrix



Algorithm 2:

- Column-wise matrix distribution
- Compute local partial result vector
- MPI_Reduce_scatter to sum and distribute partial results

Conjecture: T(m,n,p) = O(m(n/p) + m + log p), p≤n

Why?

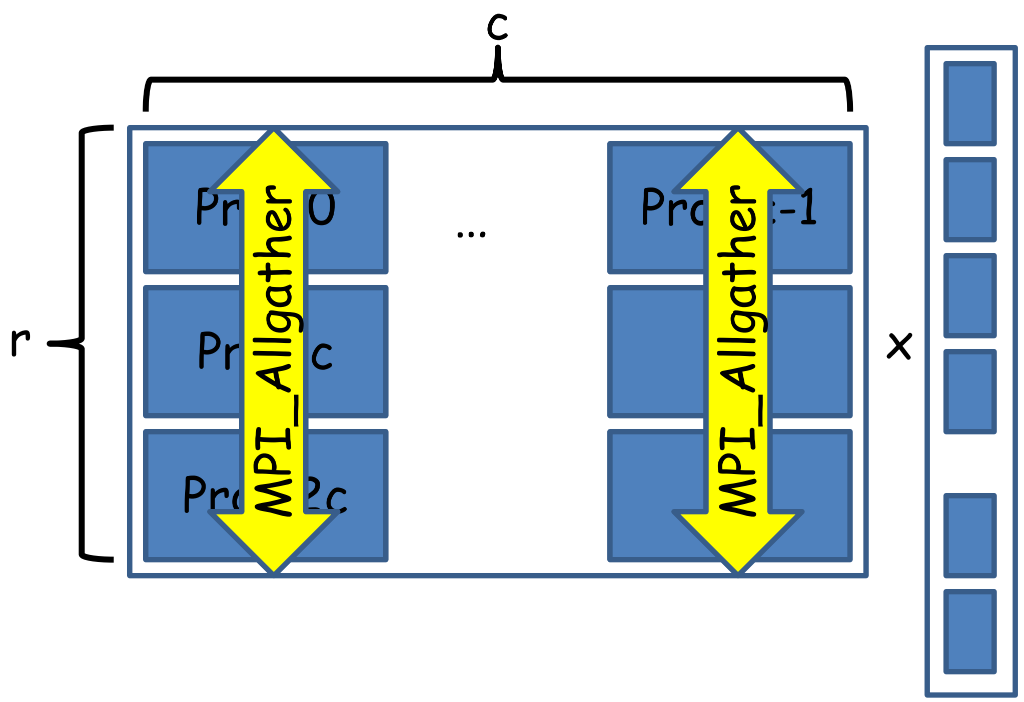
Conjecture:

- MPI_Allgatherv(n) can be done in O(n+log p) communication steps
- MPI_Reduce_scatter(n) can be done in O(n+log p) communication steps

for certain types of networks, and not better (regardless of network)

This lecture:
Will substantiate the conjectures (constructions and proofs)

Factor p = rc, each process has m/r x n/c submatrix and n/rc = n/p subvector

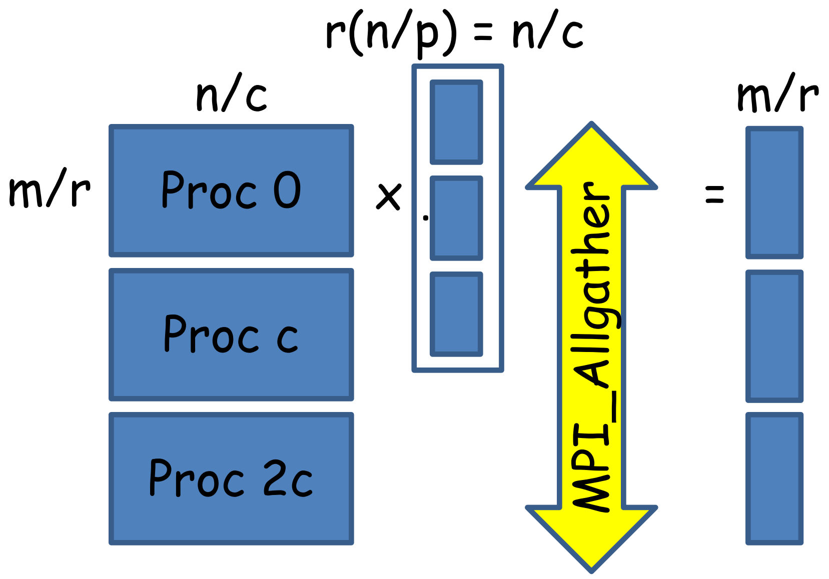


Algorithm 3:

- Matrix distribution into blocks of m/r x n/c elements
- Algorithm 1 on columns
- Algorithm 2 on rows

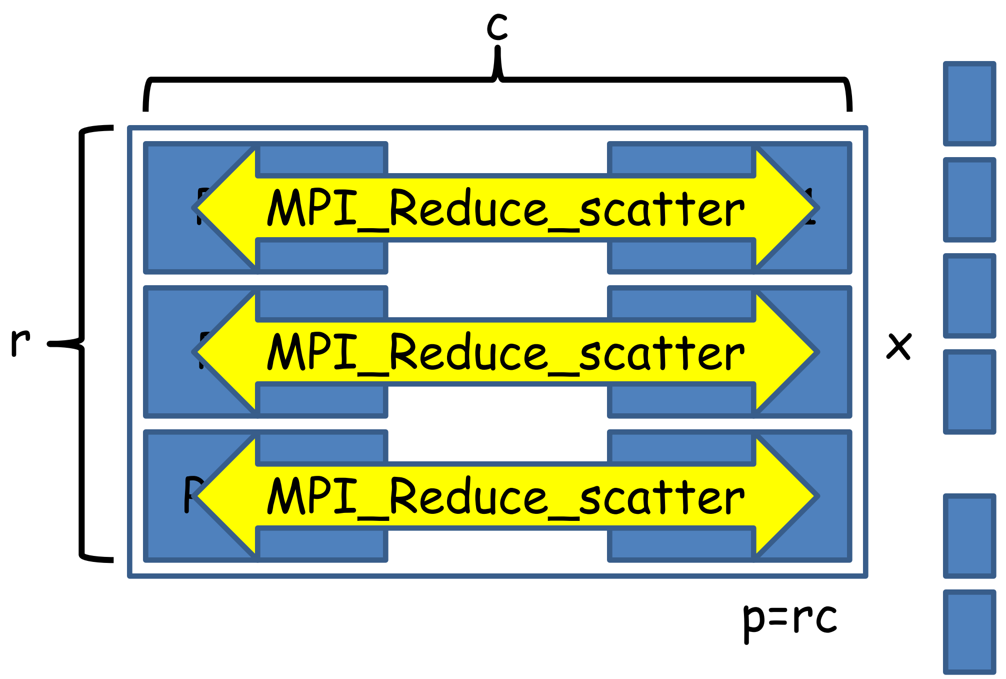
Step 1: Simultaneous MPI_Allgather(v) on all c process columns

Factor p = rc, each process has m/r x n/c submatrix and n/rc = n/p subvector



Step 1: Simultaneous MPI_Allgather(v) on all c process columns

Factor p = rc, each process has m/r x n/c submatrix and n/rc = n/p subvector

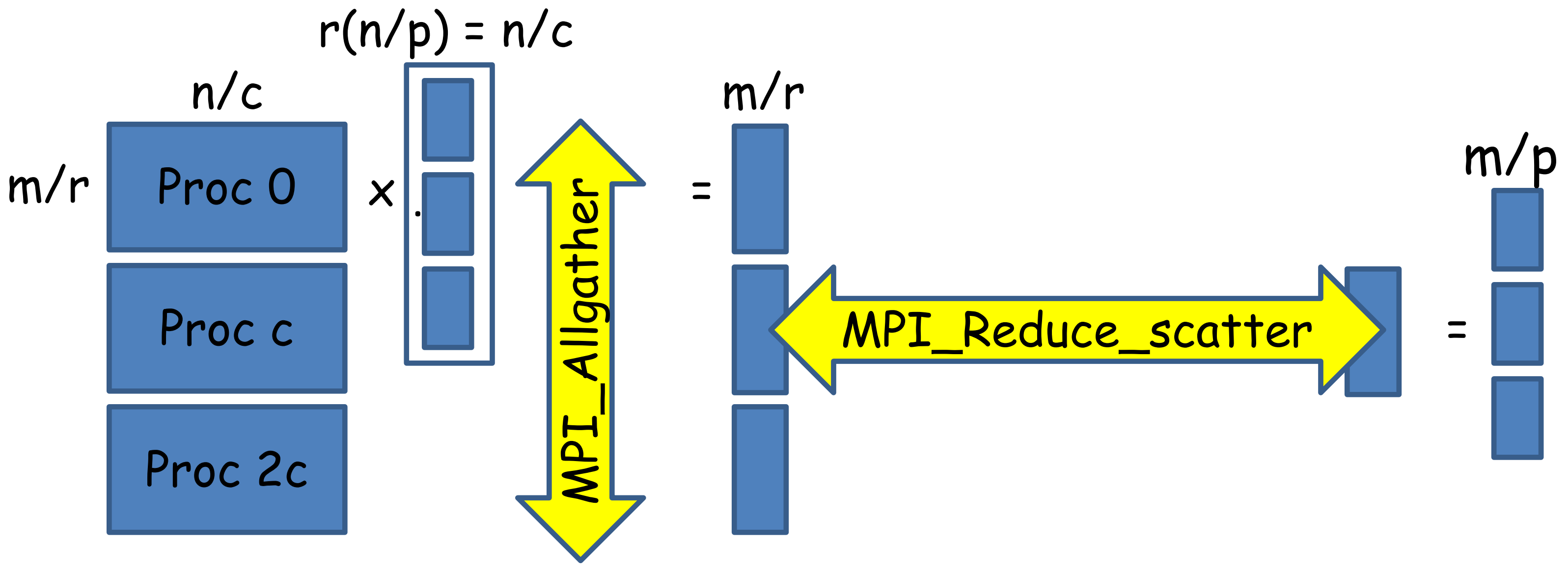


Algorithm 3:

- Matrix distribution into blocks of m/r x n/c elements
- Algorithm 1 on columns
- Algorithm 2 on rows

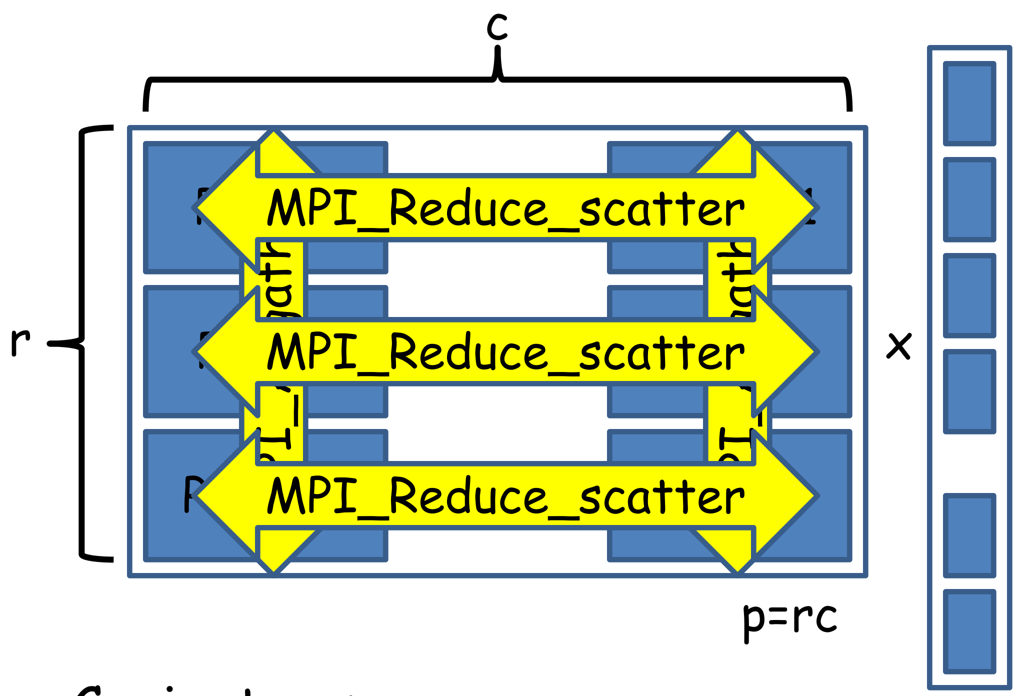
Step 4: Simultaneous MPI_Reduce_Scatter(_block) on process row communicators

Factor p = rc, each process has m/r x n/c submatrix and n/rc = n/p subvector



Step 4: Simultaneous MPI_Reduce_Scatter(_block) on process row communicators

Factor p = rc, each process has m/r x n/c submatrix and n/rc = n/p subvector

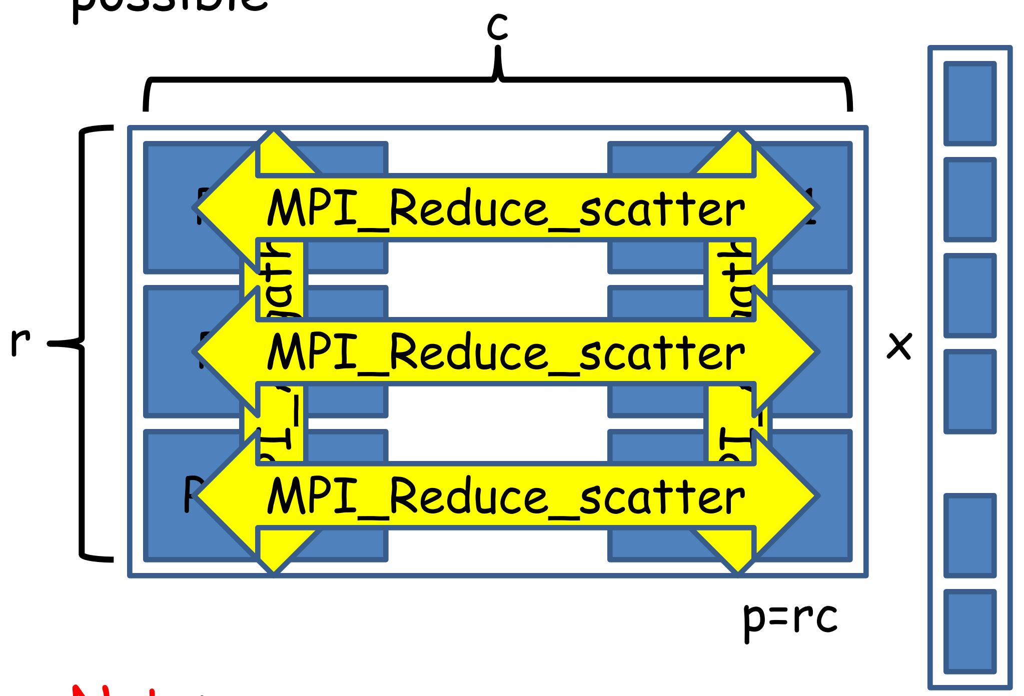


<u>Algorithm 3</u>:

- Matrix distribution into blocks of m/r x n/c elements
- Algorithm 1 on columns
- Algorithm 2 on rows

Conjecture:
T(m,n,p) = O(mn/p + n/c + m/r + log p), p≤min(mc,nr)

Algorithm 3 is more scalable. To implement the algorithm, essential that independent, simultaneous, collective communication on (column and row) subsets of processes is possible

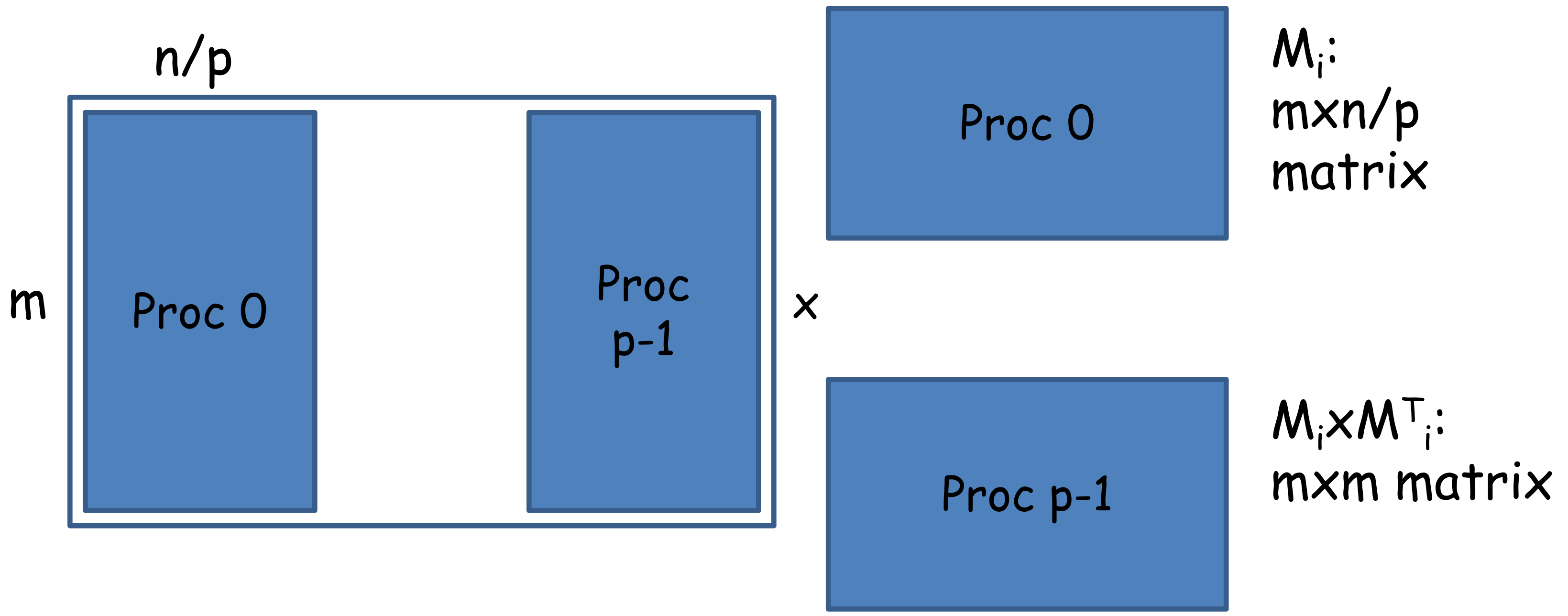


Algorithm 3:

- Matrix distribution into blocks of m/r x n/c elements
- Algorithm 1 on columns
- Algorithm 2 on rows

Note:
Interfaces that do not support collectives on subsets of processes cannot express Algorithm 3 (e.g., UPC, CaF)

Algorithm 2 can also be used for computing $M x M^T$, the product of mxn matrix M with its own transpose (also called SYRK: Symmetric Rank K update); result is a symmetric mxm matrix



Compute for processor i:
$(MxM^T)_i = \Sigma_{0 \le i < p} M_i x M^T_i$ with local SYRK and MPI_Reduce_scatter, result distributed column- or row-wise (use-case for datatypes)

This algorithm is good (optimal) for $m \leq n$ (short, wide matrices) and smaller p, $p \leq n/\sqrt{(m(m-1))}$

Cost analysis (conjecture): $T(m,n,p) = O(m^2n/p + m^2 + \log p)$

Real communication volume is $m(m+1)/2\ (p-1)/p$

MPI Process topologies (MPI 3.1, Chapter 7)

Observation:
For Algorithm 3, row and column subcommunicators are needed; must support concurrent collective operations

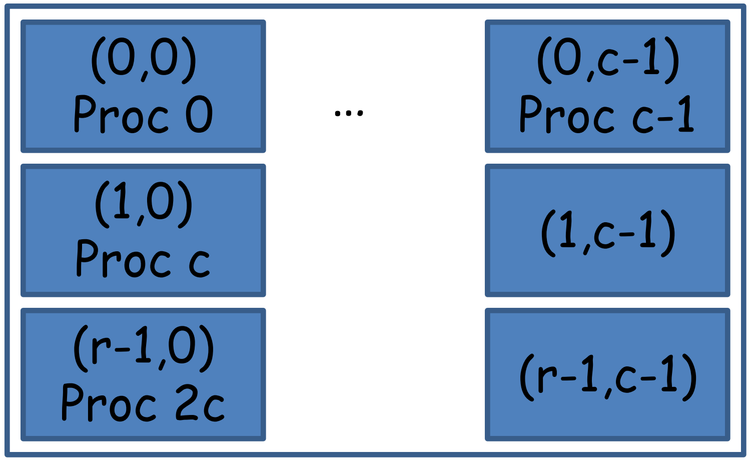
Convenient to organize processors into a Cartesian rxc mesh with MPI processes in row-major.
Use this Cartesian (x,y) naming to create subcommunicators for the matrix-vector example

Cartesian naming useful for torus/mesh algorithms, e.g., for d-dimensional stencil computations (Jacobi, Life, …)

```
rcdim[0] = c;  rcdim[1] = r;
period[0] = 0; period[1] = 0; // no wrap around

reorder = 0; // no attempt to reorder

MPI_Cart_create(comm,2,rcdim,period,reorder,&rccomm);
```

Collective communicator creation call:
All processes belonging to old communicator (`comm`) must perform call, each process becomes new communicator (`rccomm`, or `MPI_COMM_NULL`), possibly with new rank and size

```
rcdim[0] = c;  rcdim[1] = r;
period[0] = 0; period[1] = 0; // no wrap around

reorder = 0; // no attempt to reorder

MPI_Cart_create(comm,2,rcdim,period,reorder,&rccomm);
```

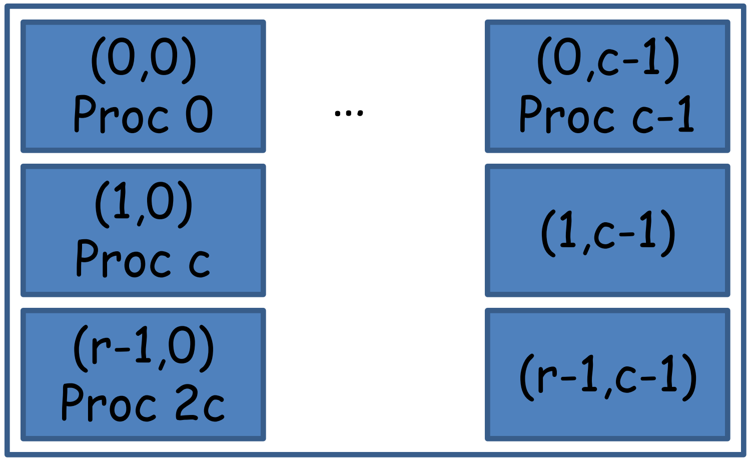


Row major process numbering

Same processes in rccomm communicator as in comm (rc must be at most size(comm)). But in rccomm, processes also have a d-dimensional coordinate as "name".

But: Ranks may not be bound to the same processors (if reorder=1)

```
rcdim[0] = c;  rcdim[1] = r;
period[0] = 0; period[1] = 0; // no wrap around

reorder = 0; // no attempt to reorder

MPI_Cart_create(comm,2,rcdim,period,reorder,&rccomm);
```



rank = $\sum_{0\leq i<d}(\mathrm{coord}[i]\prod_{0\leq j<i}d[j])$

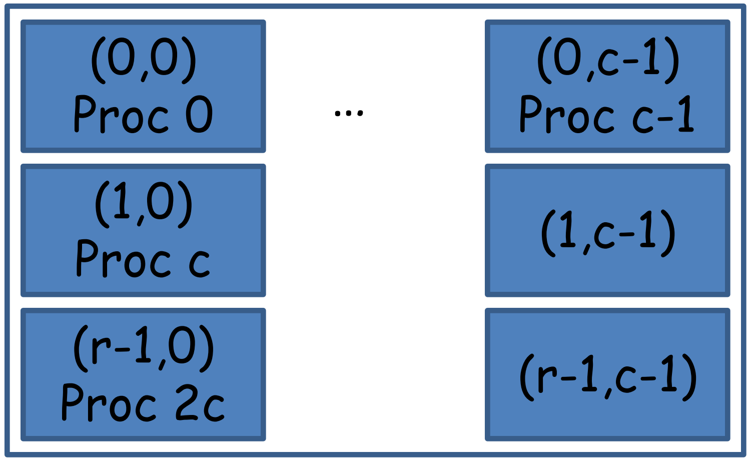
Translation functions

```
int rccoord[2];
MPI_Cart_coords(rccomm,
                rcrank,
                2,rccoord);
```

```
MPI_Cart_rank(rccomm,
              rccoord,
              &rcrank);
```

E.g., rank(2,c-2) = c-2+2c = 3c-2

```
rcdim[0] = c;  rcdim[1] = r;
period[0] = 0; period[1] = 0; // no wrap around

reorder = 0; // no attempt to reorder

MPI_Cart_create(comm,2,rcdim,period,reorder,&rccomm);
```



Communication in rccomm uses ranks (not coordinates).

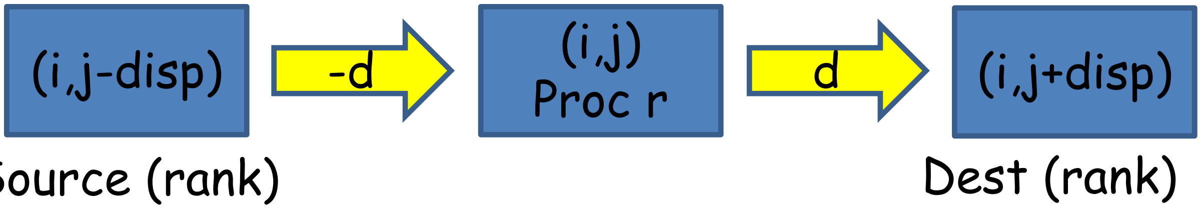
Communication between any ranks possible/allowed.

Processes are neighbors if they are adjancent in Cartesian mesh/torus

```
rcdim[0] = c;  rcdim[1] = r;
period[0] = 0; period[1] = 0; // no wrap around

reorder = 0; // no attempt to reorder

MPI_Cart_create(comm,2,rcdim,period,reorder,&rccomm);
```

Rank "shifting" function (no communication)

```
MPI_Cart_shift(rccomm,dimension,displacement,
                &source,&dest);
```



Shift along dimension 0, displacement 2 (e.g.). What if j-disp<0 or j+disp>=size? Answer: `MPI_PROC_NULL` if (`!period[0]`), otherwise cyclic shift

MPI convenience functionality for factoring p into “best” (closest??) r and c… (in d dimensions)

```
int dimensions[d] = {0, …, 0}; // all dimensions free
MPI_Dims_create(p,d,dimensions);
```

Factor p into d dimensions with order (size) as close to each other as possible… (MPI 3.1, p.293)

Much natural functionality is not in MPI (what can you think of?)

Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, Daniel J. Holmes: MPI collective communication through a single set of interfaces: A case for orthogonality. Parallel Comput. 107: 102826 (2021)

```
int dimensions[d] = {0, …, 0}; // all dimensions free
MPI_Dims_create(p,d,dimensions);
```

Factor p into d dimensions with order (size) as close to each other as possible… (MPI 3.1, p.293). Why should this be best?

Note: Seemingly innocent interface entails integer factorization. Complexity of factorization is unknown (probably high)

Note: This algorithm for 2-dimensional balancing is exponential:

```
f = (int)sqrt((double)p);
while (p%f!=0) f--;
```

Pseudo-polynomial: Polynomial in the magnitude, but not the size of p. It takes only log p bits to represent p.

But… factorization of small numbers (p in the order of millions) that fulfill the MPI specification (dimension sizes in increasing order) is not expensive (for p up to millions):

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

Ca. 2015:
Many (all?) MPI libraries had (severely) broken implementations of MPI_Dims_create

(SE) Problem: How can quality of MPI library implementation be assessed? Ensured?

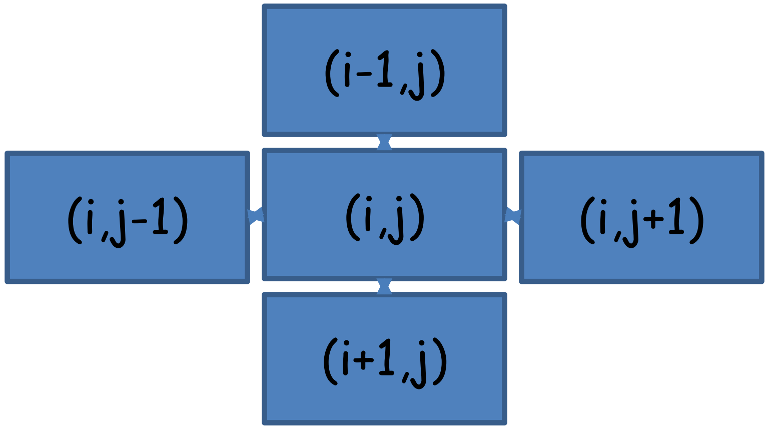
MPI_Cart_create operation

- Defines Cartesian naming, creates new communicator, and caches information with this (number and sizes of dimensions …)
- Row-major ordering of processes (dimension order 0, 1, 2, …)
- Neighbors: The 2d adjacent processes in each dimension
- Defines order of neighbors (dimension-wise), important for MPI 3.0 neighborhood collectives

See later

- May (reorder=1) attempt to remap processes, such that processes that are neighbors in virtual application topology are neighbors in physical system

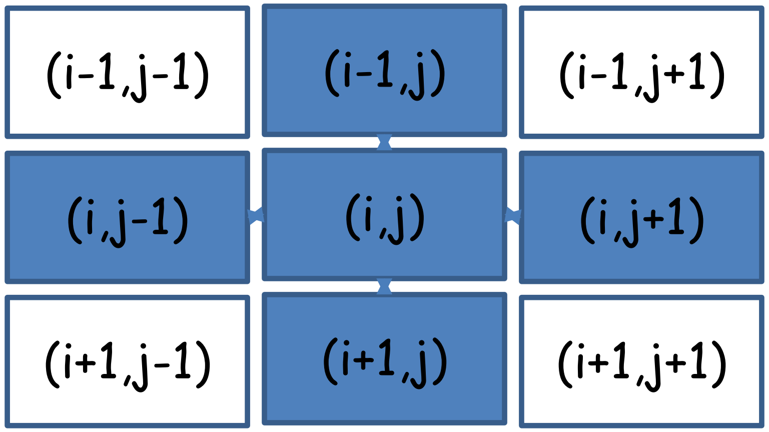
Cartesian neighborhoods in two dimensions



Pattern aka 5-point stencil

Process with coordinates (i,j) has 2d direct neighbors (for Cartesian communicator in d dimensions) with which it is implicitly supposed to communicate. This is the neighborhood (graph) used with MPI neighborhood collectives
Process (0,j) neighbor of process (r-1,j) if periodic in dimension

Cartesian neighborhoods in two dimensions



MPI restrictions:

- Not possible to specify other patterns, e.g., 9-point stencil
- Neighborhood is unweighted

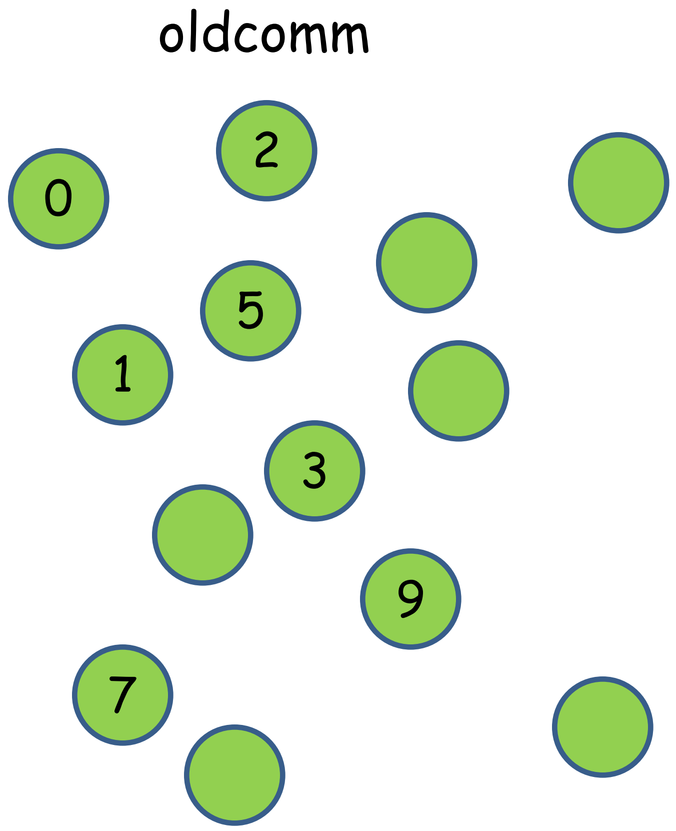
Process with coordinates (i,j) has 2d direct neighbors (for Cartesian communicator in d dimensions) with which it is implicitly supposed to communicate. This is the neighborhood (graph) used with MPI neighborhood collectives
Process (0,j) neighbor of process (r-1,j) if periodic in dimension

For discussion on communicators, naming schemes and topological information associated with communicators, see again

Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, Daniel J. Holmes: MPI collective communication through a single set of interfaces: A case for orthogonality. Parallel Comput. 107: 102826 (2021)

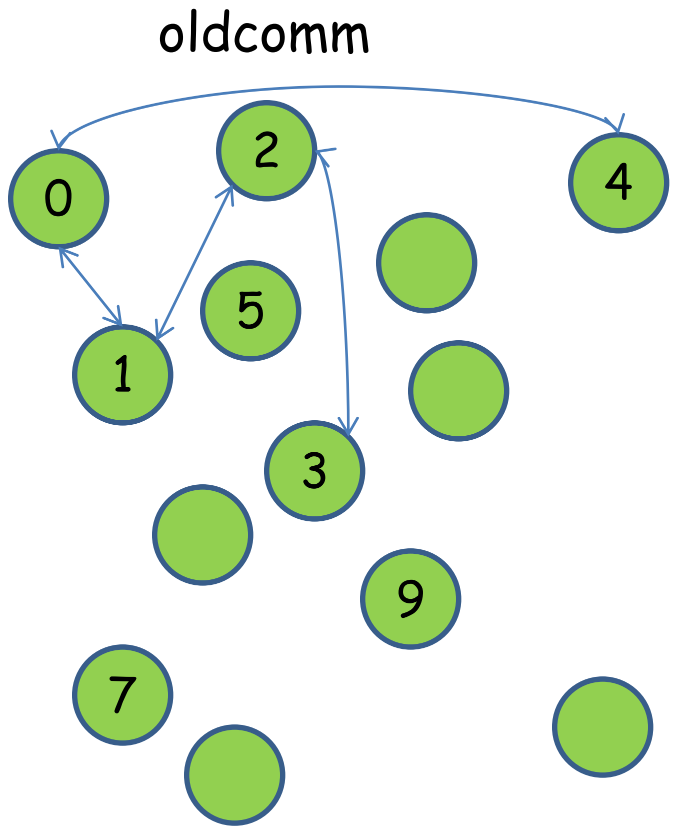
Many interesting things to do for (paid) Master's thesis…

Reordering processes via new communicators



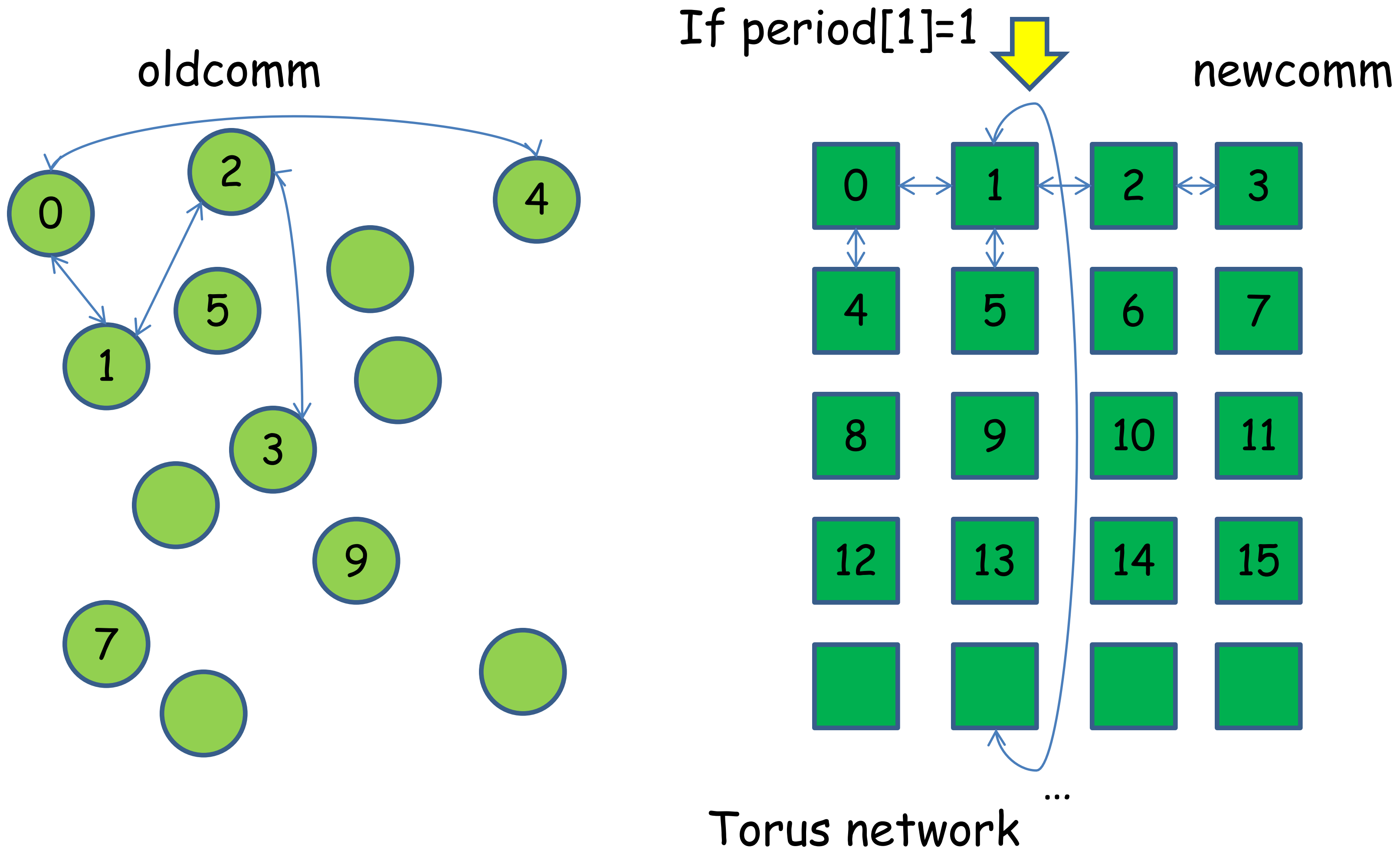
MPI assumption:
Application programmer knows which processes will communicate (heaviest communication, most frequent communication): Virtual topology

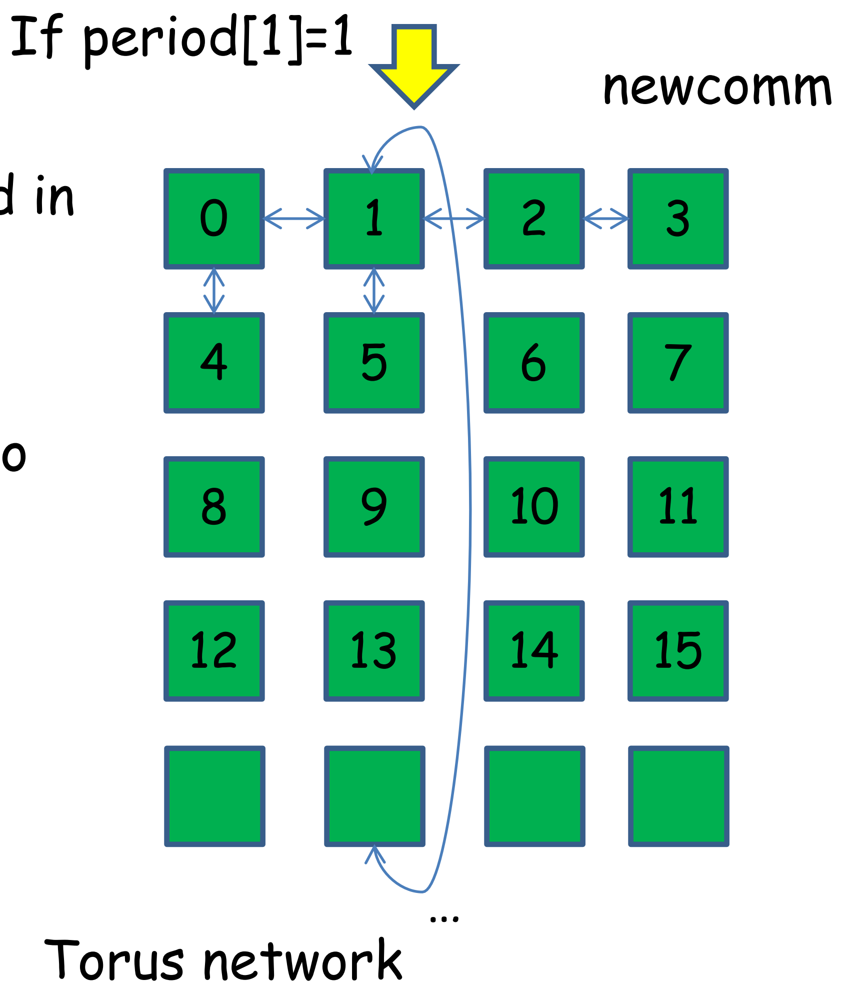
Idea:
Convey this information to MPI library; library can suggest a good mapping of MPI processes to processors that fits communication system



MPI_Cart_create:

Implicit assumption: Process with rank i in old communicator will likely communicate with neighboring processes in implicit Cartesian neighborhood


If period[1]=1
oldcomm
newcomm
0
1
2
3
4
5
7
9
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
…
Torus network

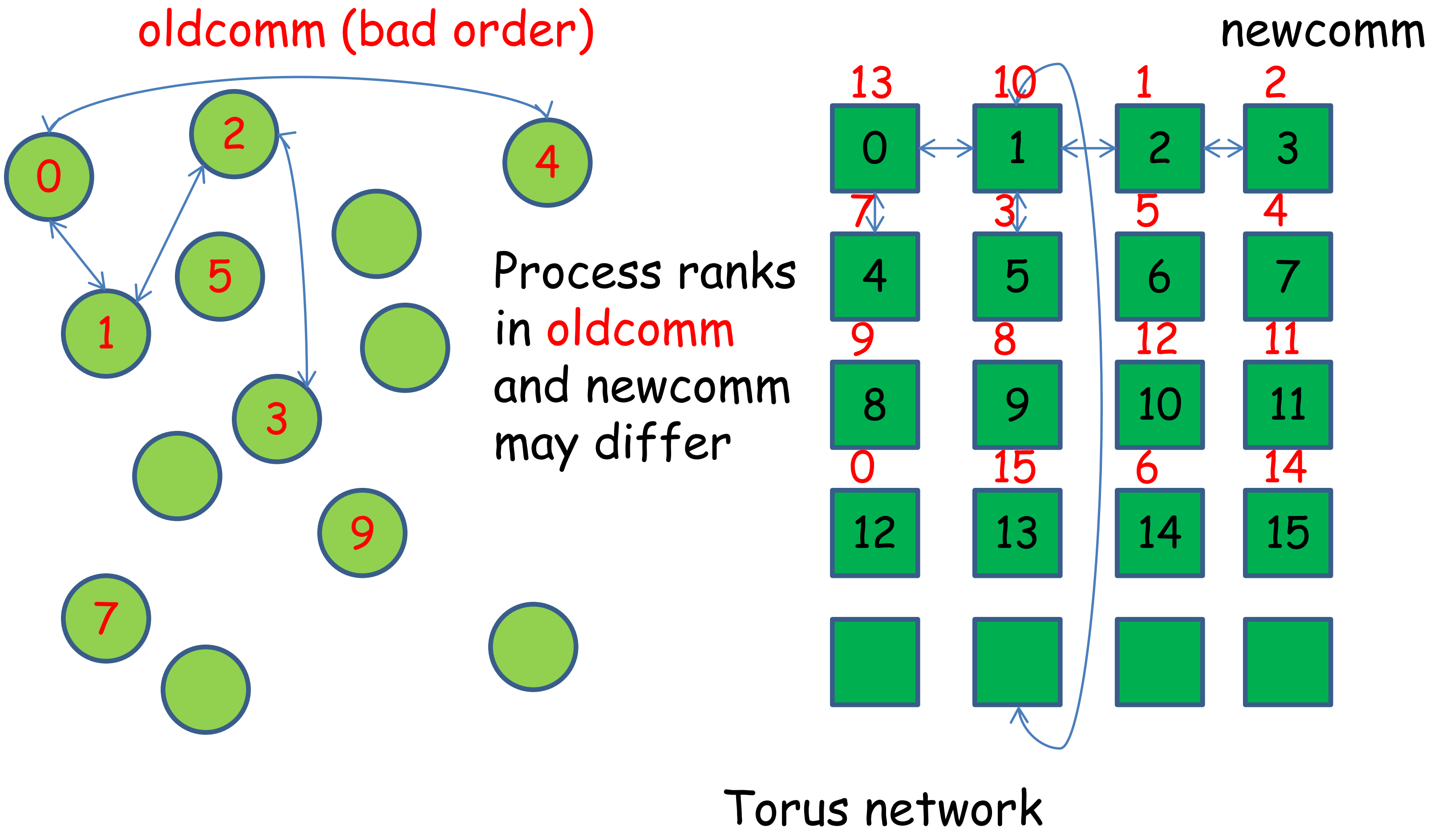


Ranks in newcomm are organized in a (virtual) d-dimensional mesh/torus

Good order: Ranks correspond to processor id's in physical torus system

Physical torus systems:
3-dim, 5-dim, 6-dim; BlueGene, Cray, Fujitsu K, Fugaku

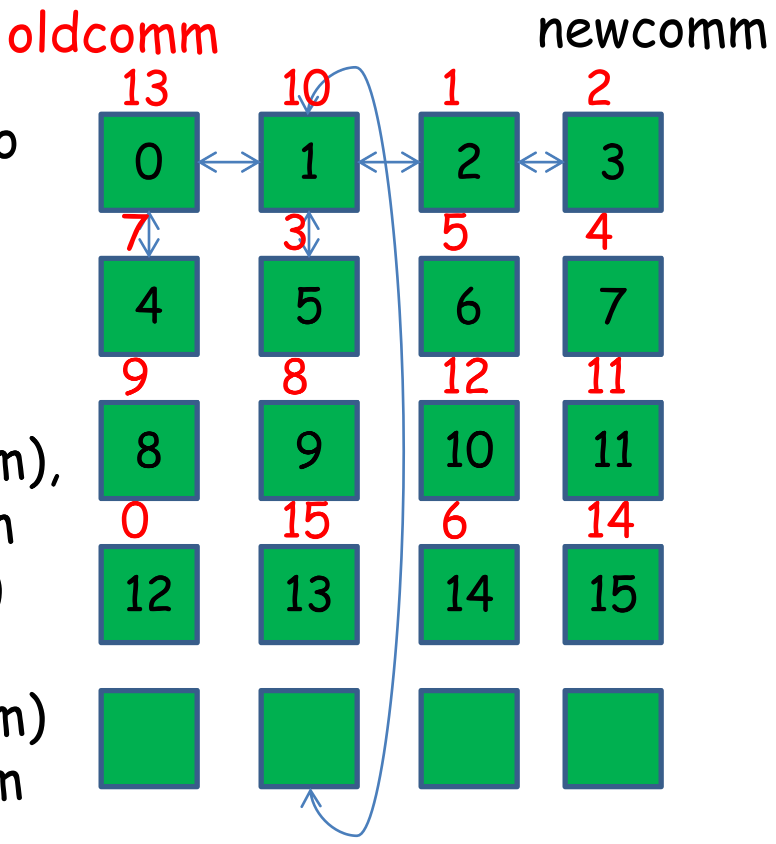
Topology creation with reorder=1: MPI library may attempt to map virtual topology to physical topology: newcomm



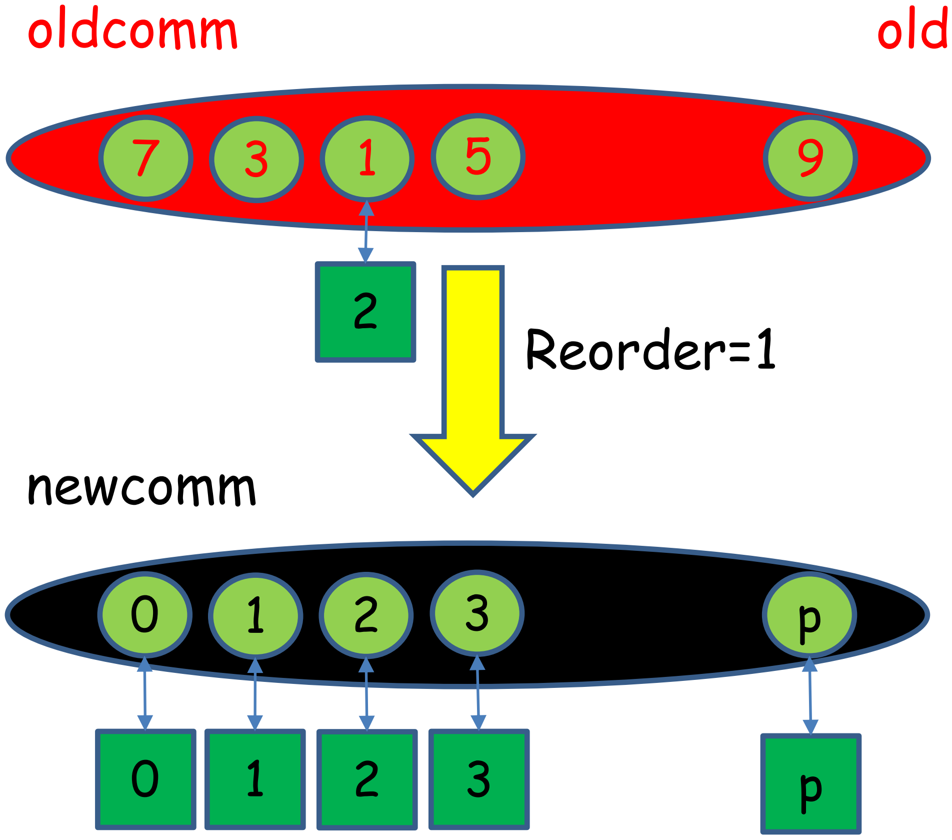
Processes in oldcomm remapped to other processes in newcomm

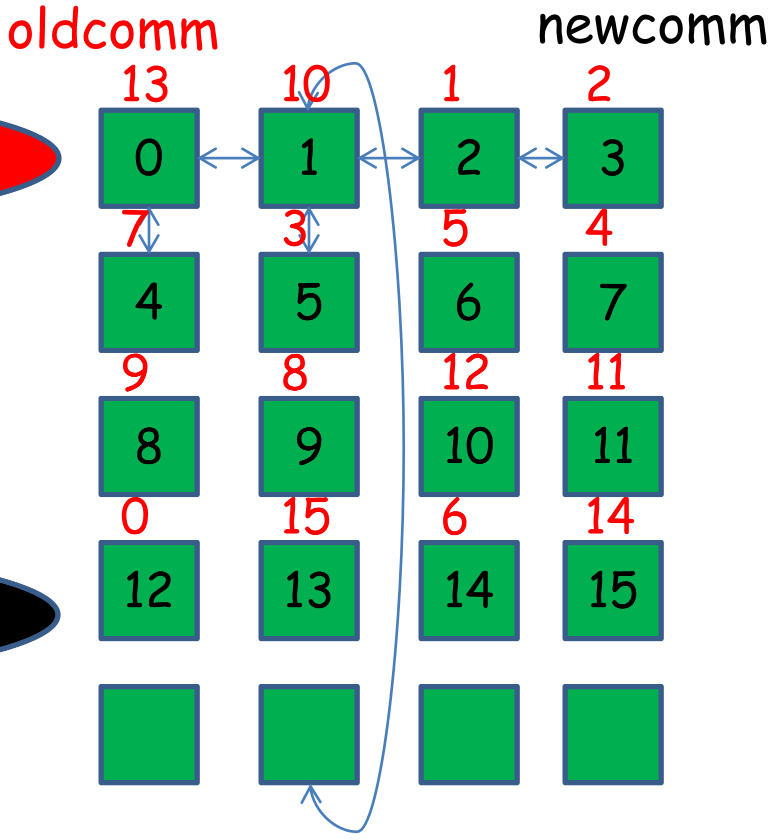
Example:

- Rank 1 in oldcomm mapped to processor 2 (rank 2 in newcomm), remapped to rank 1 in newcomm (which was rank 10 in oldcomm)
- Rank 4 in oldcomm mapped to processor 7 (rank 7 in newcomm) remapped to rank 4 in newcomm



Torus network





Torus network

If reordering has been done (ranks in newcomm ≠ ranks in oldcomm)

- Has reordering been done? All processes: Check whether rank in the two communicators differ, allreduce to inform all (or use MPI_Comm_compare)
- Application may have to transfer data to same rank in new communicator; no MPI support for this

- Need to be able to translate between ranks in different communicators

Cartesian communicators can capture only limited Cartesian communication patterns: 2d neighborhoods

```
MPI_Comm_compare(comm1,comm2,&result)
```

Result either of

- MPI_IDENT (communicators identical, same)
- MPI_CONGRUENT (same processes in same order, nevertheless different communicator)
- MPI_SIMILAR (same processes, different order)
- MPI_UNEQUAL (otherwise)

Interface ideas for general stencil patterns

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

Jesper Larsson Träff, Sascha Hunold: Cartesian Collective Communication. ICPP 2019: 48:1-48:11

Jesper Larsson Träff, Sascha Hunold, Guillaume Mercier, Daniel J. Holmes: MPI collective communication through a single set of interfaces: A case for orthogonality. Parallel Comput. 107: 102826 (2021)

Interesting Master thesis projects here

Recent improvements to MPI implementations

- Remapping for network topology: Place links in communication graph (e..g. Cartesian neighborhoods) on links in hardware network
- Remapping for hierarchical systems: Map as many neighbors as possible inside compute nodes

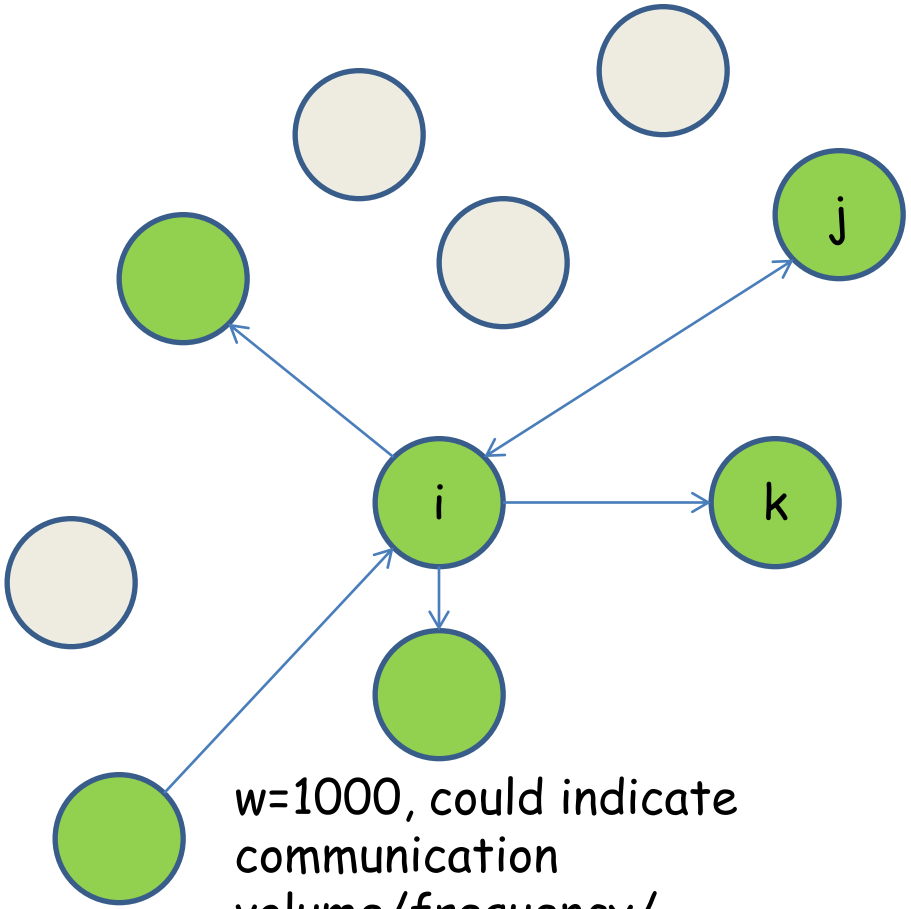
William D. Gropp: Using node and socket information to implement MPI Cartesian topologies. Parallel Computing 85: 98-108 (2019)

- Remapping for hierarchical systems: Heuristically map as many neighbors as possible inside compute nodes
- Challenge: balance bandwidth inside nodes with (multi-lane) bandwidth across nodes

Konrad von Kirchbach, Markus Lehr, Sascha Hunold, Christian Schulz, Jesper Larsson Träff: Efficient Process-to-Node Mapping Algorithms for Stencil Computations. CLUSTER 2020: 1-11

TU Wien: New FWF project on process mapping

Fully general application communication patterns



w=1000, could indicate communication volume/frequency/…

- Weighted, directed (multi)graph to specify application communication pattern (whom with whom, how much)
- Hints can be provided (info)
- Distributed description, any MPI process can specify any edges

```
MPI_Dist_graph_create(comm,
                      n,sources,degrees,destinations,
                      weights,
                      info,reorder,&graphcomm)
```

Example: Process i may specify (what it knows):

sources:	[0,5,7,2,…]
(out)degrees:	[2,2,2,2,…]
destinations:	[1,2,18,36,4,5,8,117,…]
weights:	[100,1000,10,10,1000,1000,100,100,…]

MPI (3.0) uses graph structure for descriptive purposes (neighborhood collectives). MPI library can attempt to remap communication graph to fit target communication system. Other optimizations are possible.

```
int reorder = 1;
MPI_Dist_graph_create(comm,
                      n,sources,degrees,destinations,
                      weights,
                      info,reorder,&graphcomm)
```

If reordering is requested, rank i in new graphcomm may be bound to different processor than rank i in comm

⇒ Data redistribution (from rank i in comm to rank i in graphcomm) can be necessary; must be done explicitly by application (first: Check if reordering has taken place)

Note:
It is legal for an MPI library to return a graphcomm with the same mapping as comm

If each process knows its incoming and outgoing communication edges

```
MPI_Dist_graph_create_adjacent(comm,
          indeg,sources,sourceweights,
          outdeg,destinations,destweights,
          info,reorder,&graphcomm)
```

may be more convenient and efficient

Furthermore, the operation does not require communication in order to support the neighborhood query functions:

```
MPI_Dist_graph_neighbors_count(graphcomm, …)
MPI_Dist_graph_neighbors(graphcomm, …)
```

MPI 3.1 Virtual topologies

- "info" can be provided; not (yet) specified in MPI standard, implementation dependent
- Interpretation of weights unspecified (when? How much? How often? …)

- Differences between Cartesian and distributed graph topologies: Cartesian creation function (MPI_Cart_create) takes no weights, no info

Do MPI libraries perform non-trivial mapping?

Try it

Distributed graph interface since MPI 2.2.

Old MPI 1.0 interface is badly non-scalable: Full graph required at each process

Don't use MPI_Graph_create!

Definition:
An MPI construct is non-scalable if it entails $\Omega(p)$ memory or time overhead(*)

(*): that cannot be accounted for/amortized in the application

T. Hoefler, R. Rabenseifner, H. Ritzdorf, B. R. de Supinski, R. Thakur, Jesper Larsson Träff: The scalable process topology interface of MPI 2.2. Concurrency and Computation: Practice and Experience 23(4): 293-310 (2011)

Implementing topology mapping

Step 1:
Application specifies communication patterns as weighted directed (guest) graph (implicitly: Unweighted, d-dimensional Cartesian pattern)

Step 2:
MPI library knows physical communication structure; model subgraph of system spanned by communicator as weighted, directed (host) graph

Step 3:
Map/embed communication (guest) graph onto system (host) graph, subject to… The map (embedding) is an injective function Γ from guest graph vertices to host graph vertices

Mapping function can be injective, surjective, bijective

Some optimization criteria:

- As many heavy communication edges on physical edges as possible
- Minimize communication costs (volume)
- Small congestion (bandwidth)
- Small dilation (latency)
- …

Meta-Theorem: Most graph embedding problems are NP-hard

Depending on physical system (e.g., clustered, hierarchical), problem can sometimes be formulated as a graph partitioning problem (e.g., clusters, partition into nodes, minimize weight of inter-node edges), …

Guest graph G=(V,E), let w(u,v) denote the volume of data from process (vertex) u to process (vertex) v

Host graph H=(V',E'), let c(u',v') denote the network capacity between processors u' and v'. Let R(u',v') be a function determining the path from processor u' to v' in the system (Routing algorithm)

Let Γ: V -> V' be an injective mapping (embedding). The congestion of link e in E' wrt. Γ is defined as

Cong(e) = [∑w(u,v) over (u,v) in E where e in R(Γ(u),Γ(v))]/c(e)

The congestion of Γ is defined as the maximum congestion of any link e

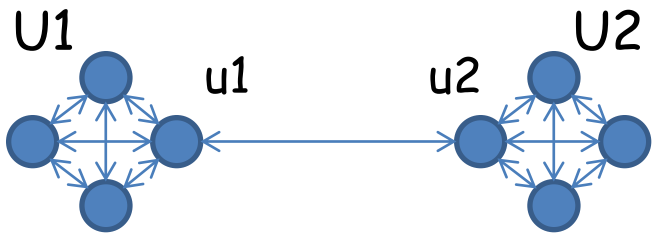
Call the problem of determining whether an embedding exists that has congestion less than C for CONG

Concrete Theorem: For given guest and host graphs G and H, determining whether an embedding exists that has congestion at most C is NP-complete.

Proof: CONG is in NP. Non-deterministically choose an embedding, and check whether its congestion is at most C; this can be done in polynomial time

Completeness is by reduction from MINIMUM CUT INTO BOUNDED SETS (Garey&Johnson ND17): Given a graph G=(V,E), two vertices s,t in V, integer L, determine whether there is a partition of V into subsets V1 and V2 with s in V1, t in V2, |V1|=|V2|, such that the number of edges between V1 and V2 is at most L

Let (G,s,t,L) be an instance of MINIMUM CUT INTO BOUNDED SETS. Let the host graph H be constructed as follows



Complete $K_{|V/2|}$ graphs

c(u1,u2) = |V|, c(e) =1 for all other edges of H. The routing function R selects for u,v in V' a shortest path from u to v

Let the guest graph be G with w(e)=1 for e in E, except for the edge (s,t) for which w(s,t) = $|V|^4$ if (s,t) is not in E, and w(s,t)=$|V|^4$+1 if (s,t) in E. That is, the guest graph may have one edge not in G, namely (s,t), and each edge in G contributes a volume of at least 1

Any injective mapping Γ from G to H determines a partition of V into V1 and V2 with |V1|=|V2|, namely Vi = {v | Γ(v) in Ui}

Any mapping that minimizes the congestion must map (s,t) to (u1,u2). This gives a congestion of at most $|V|^3+|V|$ (the volume on (u1,u2) is at most $|V|^4+|V|^2/4$); any other mapping has congestion at least $|V|^4$.

In such a mapping, the most congested edge is (u1,u2), with congestion

C = $|V|^3+|\{(v1,v2)$ in E | v1 in V1 and v2 in V2 $\}|/|V|$

fractional

Thus, a solution to CONG with congestion at most $|V|^3+L/|V|$ gives a solution to the MINIMUM CUT INTO BOUNDED SETS instance

A nicer construction (15.11.2021)

Host graph H as before (dumbbell)

Let the guest graph be G with w(e)=$|V|$ for e in E, except for the edge (s,t) for which w(s,t) = $|V|^4$ if (s,t) is not in E, and w(s,t)=$|V|^4+|V|$ if (s,t) in E. That is, the guest graph may have one edge not in G, namely (s,t), and each edge in G contributes a volume of at least $|V|$

Any injective mapping Γ from G to H determines a partition of V into V1 and V2 with $|V1|=|V2|$, namely $Vi = \{v \mid \Gamma(v) \text{ in } Ui\}$

Any mapping that minimizes the congestion must map (s,t) to (u1,u2). This gives a congestion of at most $|V|^3+|V|^2$ (the volume on (u1,u2) is at most $|V|^4+|V||V|^2/4$); any other mapping has congestion at least $|V|^4$.

In such a mapping, the most congested edge is (u1,u2), with congestion

$C = |V|^3+|\{(v1,v2) \text{ in } E \mid v1 \text{ in } V1 \text{ and } v2 \text{ in } V2\}|$

Thus, a solution to CONG with congestion at most $|V|^3+L$ gives a solution to the MINIMUM CUT INTO BOUNDED SETS instance

Conversely, from a solution to the MINIMUM CUT INTO BOUNDED SETS an embedding with this congestion can easily be constructed (check!)

Proof (simplified) from

T. Hoefler, M. Snir: Generic topology mapping strategies for large-scale parallel architectures. ICS 2011: 75-84

ND17, see

M. R. Garey, D. S. Johnson: Computer and Intractability. A guide to the theory of NP-Completeness. W. H. Freeman, 1979 (update 1991)

CONG is a special case of the simple graph embedding problem

For given (complete) guest graph G=(V,E) with edge weights w(u,v), and host graph H=(V',E') with edge costs c(u',v'), find an injective mapping (embedding) Γ: V -> V' that minimizes

Σ(u,v) in E: w(u,v)c(Γ(u) Γ(v))

This is the quadratic assignment problem (QAP), which is NP-complete (Garey&Johnson ND43)

w(u,v): volume of data to be transferred between processes u and v
c(u,v): cost per unit; could model pairwise communication costs (without congestion) in network

Some MPI relevant solutions to the topology mapping problem

S. H. Bokhari: On the Mapping Problem. IEEE Trans. Computers (TC) 30(3):207-214 (1981)
T. Hatazaki: Rank Reordering Strategy for MPI Topology Creation Functions. PVM/MPI 1998: 188-195
J. L. Träff: Implementing the MPI process topology mechanism. SC 2002:1-14
H. Yu, I-Hsin Chung, J. E. Moreira: Blue Gene system software - Topology mapping for Blue Gene/L supercomputer. SC 2006: 116
A. Bhatele: Topology Aware Task Mapping. Encyclopedia of Parallel Computing 2011: 2057-2062
A. Bhatele, L. V. Kalé: Benefits of Topology Aware Mapping for Mesh Interconnects. Parallel Processing Letters 18(4): 549-566 (2008)
Emmanuel Jeannot, Guillaume Mercier, Francois Tessier: Process Placement in Multicore Clusters: Algorithmic Issues and Practical Techniques. IEEE Trans. Parallel Distrib. Syst. 25(4): 993-1002 (2014)

Heuristics for Cartesian topology mapping (with arbitrary neighborhoods)

Sascha Hunold, Konrad von Kirchbach, Markus Lehr, Christian Schulz, Jesper Larsson Träff: Efficient Process-to-Node Mapping Algorithms for Stencil Computations. IEEE Cluster (2020)
William D. Gropp: Using node and socket information to implement MPI Cartesian topologies. Parallel Comput. 85: 98-108 (2019)

Graph partitioning software for process topology embedding

KaHIP, Karlsruhe High Quality Parititoning, recent, state-of-the art graph partitioning (heuristic, multi-level) for large graphs; see http://algo2.iti.kit.edu/documents/kahip/

Peter Sanders, Christian Schulz:
Think Locally, Act Globally: Highly Balanced Graph Partitioning.
SEA 2013: 164-175

VieM, Vienna Mapping and Quadratic Assignment, solves the embedding problem for hierarical systems as a special quadratic assignment problem; see http://viem.taa.univie.ac.at/

Christian Schulz, Jesper Larsson Träff: Better Process Mapping and Sparse Quadratic Assignment. SEA 2017: 4:1-4:15

Graph partitioning software for process topology embedding

METIS, ParMETIS, … : Well-known (early) software packages for (hyper)graph and mesh partitioning; see http://glaros.dtc.umn.edu/gkhome/projects/gp/products?q=views/metis

George Karypis: METIS and ParMETIS. Encyclopedia of Parallel Computing 2011: 1117-1124

Scotch, a "software package and libraries for sequential and parallel graph partitioning, static mapping and clustering, sequential mesh and hypergraph partitioning, and sequential and parallel sparse matrix block ordering"; see http://www.labri.fr/perso/pelegrin/scotch/

Cédric Chevalier, François Pellegrini: PT-Scotch. Parallel Computing 34(6-8): 318-331 (2008)

Graph partitioning software for process topology embedding

Jostle, originally for partitioning and load balancing for unstructructured meshes; no longer open source (2017); see http://staffweb.cms.gre.ac.uk/~wc06/jostle/

C. Walshaw and M. Cross. JOSTLE: Parallel Multilevel Graph-Partitioning Software - An Overview. In F. Magoules, editor, Mesh Partitioning Techniques and Domain Decomposition Techniques, pages 27-58. Civil-Comp Ltd., 2007. (Invited chapter)

Lots of possibilities for projects/Master's Thesis

Topology mapping: Discussion

- For scalability, distributed specification of communication graph
- Requires distributed algorithm (heuristic) to solve/approximate hard optimization problem
- Can the mapping overhead be amortized?
- Does it make sense to look for an optimum? Is the MPI interface sufficiently rich to provide all relevant information? Does the application have this information?
- Is the application programmer prepared to use the complex interface?

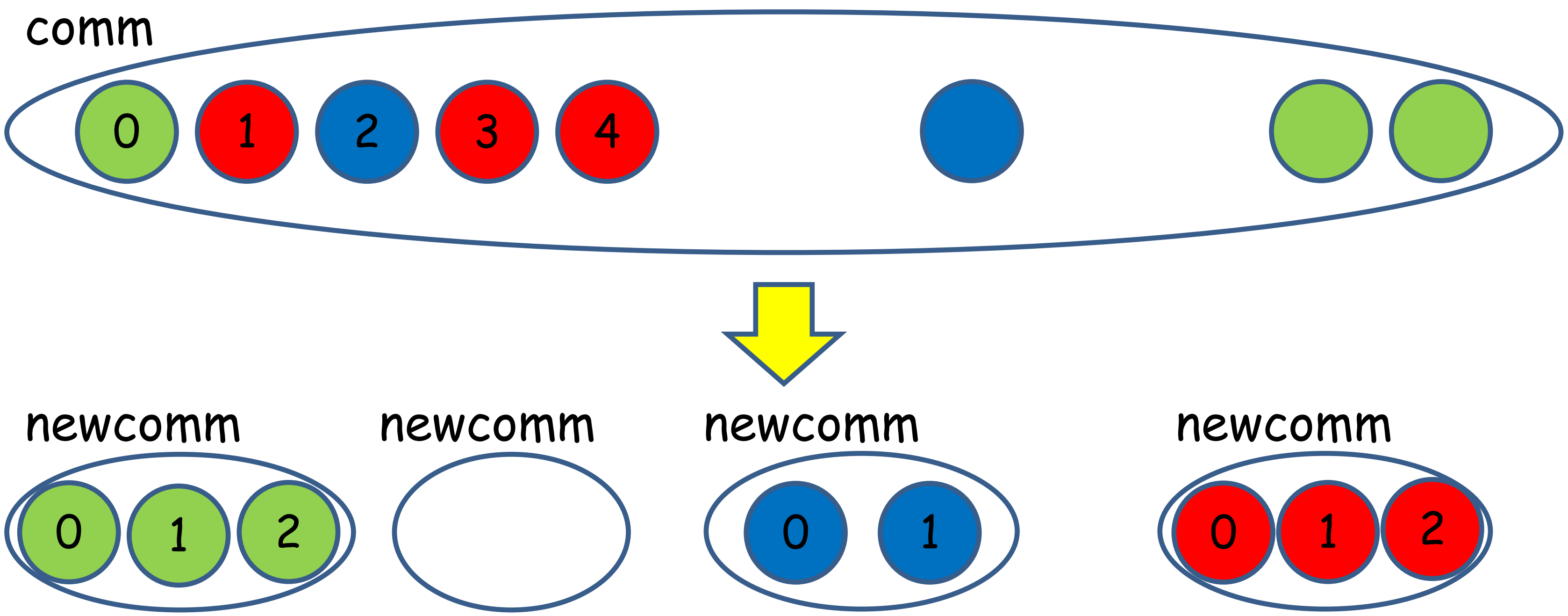
Creating the communicators for MV example (Method 1)

```
int rcrank;
int rccoord[2];

MPI_Comm_rank(rccomm,&rcrank);
MPI_Cart_coords(rccomm,rcrank,2,rccoord);

MPI_Comm_split(rccomm,rccoord[0],rccoord[1],&ccomm);
MPI_Comm_split(rccomm,rccoord[1],rccoord[0],&rcomm);
```

color: all processes calling with same color, will belong to same communicator

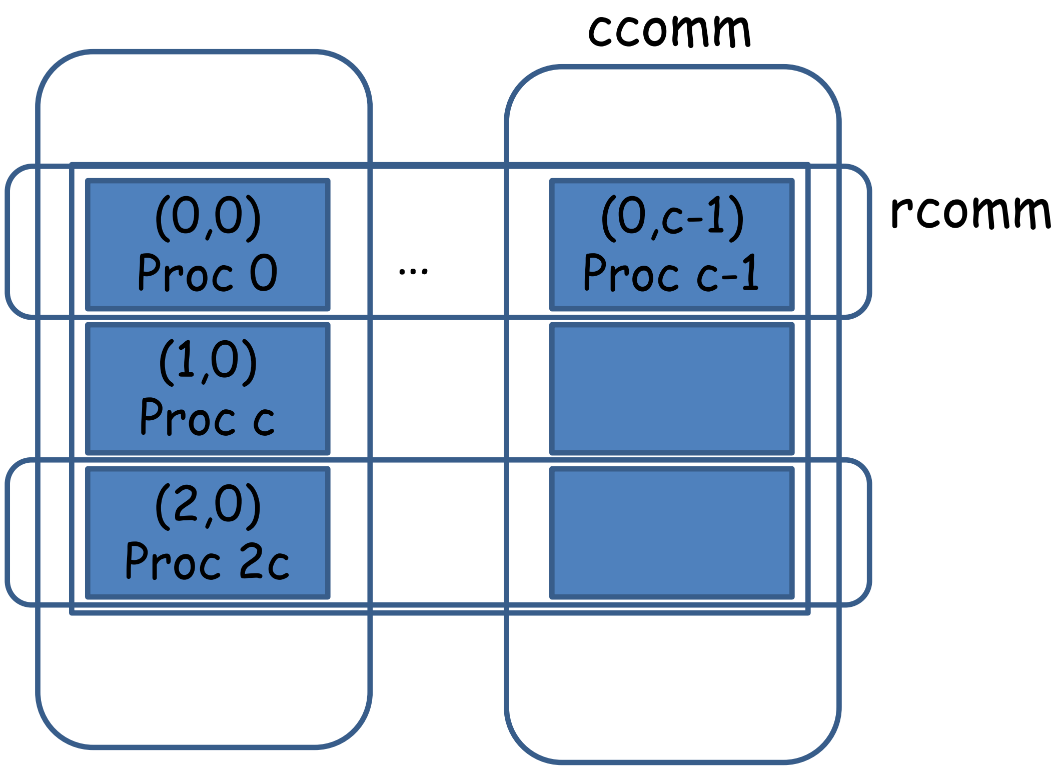
key: determines relative process ordering in new comminicator

```
MPI_Comm_split(comm,color,key,&newcomm);
```



Processes in comm are sorted after key; determines relative order in newcomm

MPI_Comm_split (color=rccoord[0]);
MPI_Comm_split (color=rccoord[1]);



Creating the communicators for MV example (Method 2)

MPI_Comm_split() is an expensive collective operation. MPI process groups can be used for possibly more efficient communicator creation

```
MPI_Comm_group(rccomm,&rcgroup); // get group out
MPI_Comm_rank(rccomm,&rcrank);
MPI_Cart_coords(rccomm,rcrank,2,rccoord);

for (i=0; i<rcsize; i++) {
  MPI_Cart_coords(rccomm,i,2,coord);
  if (coord[0]==rccoord[0]) cranks[c++] = i;
  if (coord[1]==rccoord[1]) rranks[r++] = i;
}
MPI_Group_include(rcgroup,c,cranks,&cgroup);
MPI_Group_include(rcgroup,r,rranks,&rgroup);
MPI_Comm_create(rccomm,cgroup,&ccomm);
MPI_Comm_create(rccomm,rgroup,&rcomm);
```

Process group operations

Better?

MPI process groups (MPI 3.1, Chapter 6)

Process local MPI objects representing ordered sets of processes

MPI_Comm_group: Get group of processes associated with communicator

Processes in group ranked from 0 to size of group-1, a process can locally determine if it's a member of some group, and its rank in this group

Operations:

- Comparing groups (same processes, in same order)
- Group union, intersection, difference, …
- Inclusion and exclusion of groups
- Range groups
- Translation of ranks between groups

Creating the communicators (Method 2')

MPI_Comm_split() is an expensive collective operation. MPI process groups can be used for possibly more efficient communicator creation

```
MPI_Comm_group(rccomm,&rcgroup); // get group out
MPI_Comm_rank(rccomm,&rcrank);
MPI_Cart_coords(rccomm,rcrank,2,rccoord);

for (i=0; i<rcsize; i++) {
  MPI_Cart_coords(rccomm,i,2,coord);
  if (coord[0]==rccoord[0]) cranks[c++] = i;
  if (coord[1]==rccoord[1]) rranks[r++] = i;

MPI_Group_include(rcgroup,c,cranks,&cgroup);
MPI_Group_include(rcgroup,r,rranks,&rgroup);
MPI_Comm_create_group(rccomm,cgroup,tag,&ccomm);
MPI_Comm_create_group(rccomm,rgroup,tag,&rcomm);
```

Even better?

MPI 3.1 standard: Many variations, many subtleties: Read it

- MPI_Comm_create(comm, …): Collective over all processes in comm
- MPI_Comm_create_group(comm,group, …): Collective over all processes in group, processes in group subset of processes in group of comm

MPI_Comm_create_group: Smaller groups can work independently to create their communicator. Tag arguments for multi-threaded uses (many news in MPI 3.1)

Why should MPI_Comm_create be better than MPI_Comm_split?

Reasonable to expect

"not slower than"

MPI_Comm_create(…,group,…) ≤ MPI_Comm_split(…,color…)

all other things (communicator) being equal

- MPI_Comm_create() straightforward by MPI_Comm_split(): take color as first process in group, key as rank in group, for processes in group (MPI_UNDEFINED otherwise). More specialized operation

Exercise: Implement

Reasonable to expect MPI_Comm_create() in O(log p)? Or O(log p+p)?

Probably not! Recall: MPI has no performance model or guarantee

MPI_Comm_create(…,group,…) ≤ MPI_Comm_split(…,color…)

Example of performance guideline that can be verified by benchmarking.

If not fulfilled, something wrong with MPI library

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

All group operations can be implemented in O(s) operations, where s is the size of the largest group in the operation. Space consumption is O(U), where U is the maximum number of MPI processes in the system (size of `MPI_COMM_WORLD`)

Questions:

- Group (and communicator) operations permit construction of unrestricted rank->process mappings; this is costly in space. Possible to define a useful set of more restricted operations?
- Which group operations can be implemented in o(s) operations?

Jesper Larsson Träff: Compact and Efficient Implementation of the MPI Group Operations. EuroMPI 2010: 170-178

Problem still to be solved:
Finding out where data from some rank (in oldcomm) have to be sent to: Where is the rank in newcomm?

```
MPI_Group_translate_ranks(MPI_Group group1,
                          int n, const int ranks1[],
                          MPI_Group group2,
                          int ranks2[])
```

Problem: Find `fromrank`, `torank`, such that process in (in oldcomm) can receive data from `fromrank`, and send its own data to `torank`

Recall: Cannot send from rank in oldcomm to rank in newcomm; communication is always relative to one, same communicator

```
int torank, fromrank; // to be computed, in oldcomm
int oldrank, newrank;

MPI_Group oldgroup, newgroup;
MPI_Comm_group(oldcomm,&oldgroup);
MPI_Comm_group(newcomm,&newgroup);

// where has rank been mapped to?
MPI_Comm_rank(oldcomm,&oldrank); // rank in old
MPI_Group_translate_rank(newgroup,1,&oldrank,
                         oldgroup,&torank);
// torank may be MPI_UNDEFINED
// if newcomm smaller than oldcomm

MPI_Comm_rank(newcomm,&newrank);
MPI_Group_translate_ranks(newgroup,1,&newrank,
                          oldgroup,&fromrank);
```

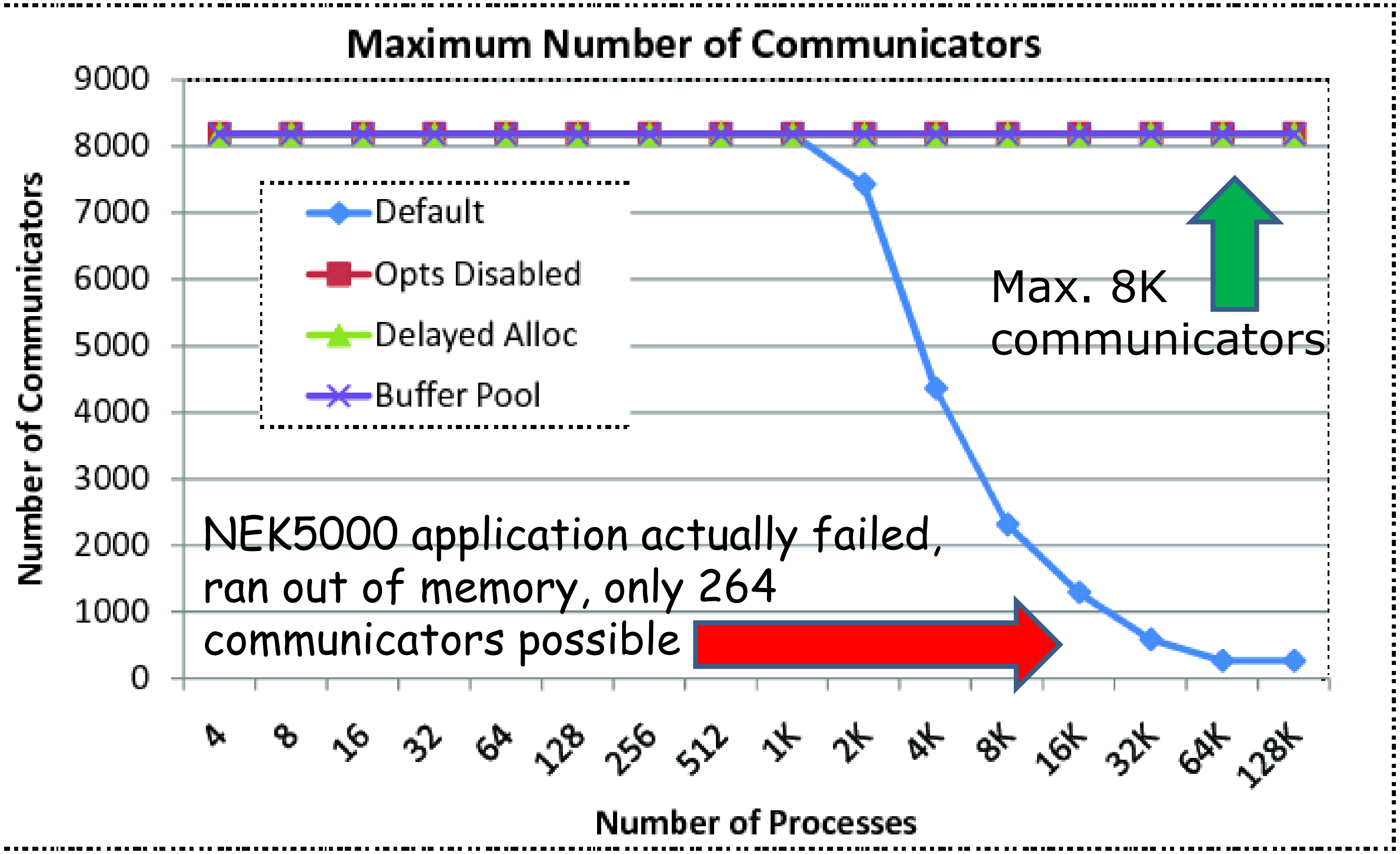
Scalability of MPI communicator concept

- A communicator must support translation from MPI rank to physical core in system
- Storing translations explicitly as p-element lookup array (per core) is a non-scalable implementation. But still the most common

Example: 250.000 cores, 4-Byte integers ≈ 1MByte space per communicator (per core). Applications on systems with limited memory, and many communicators can be trouble

Note:
MPI specification puts no restrictions on the number and kind of communicators that can be created

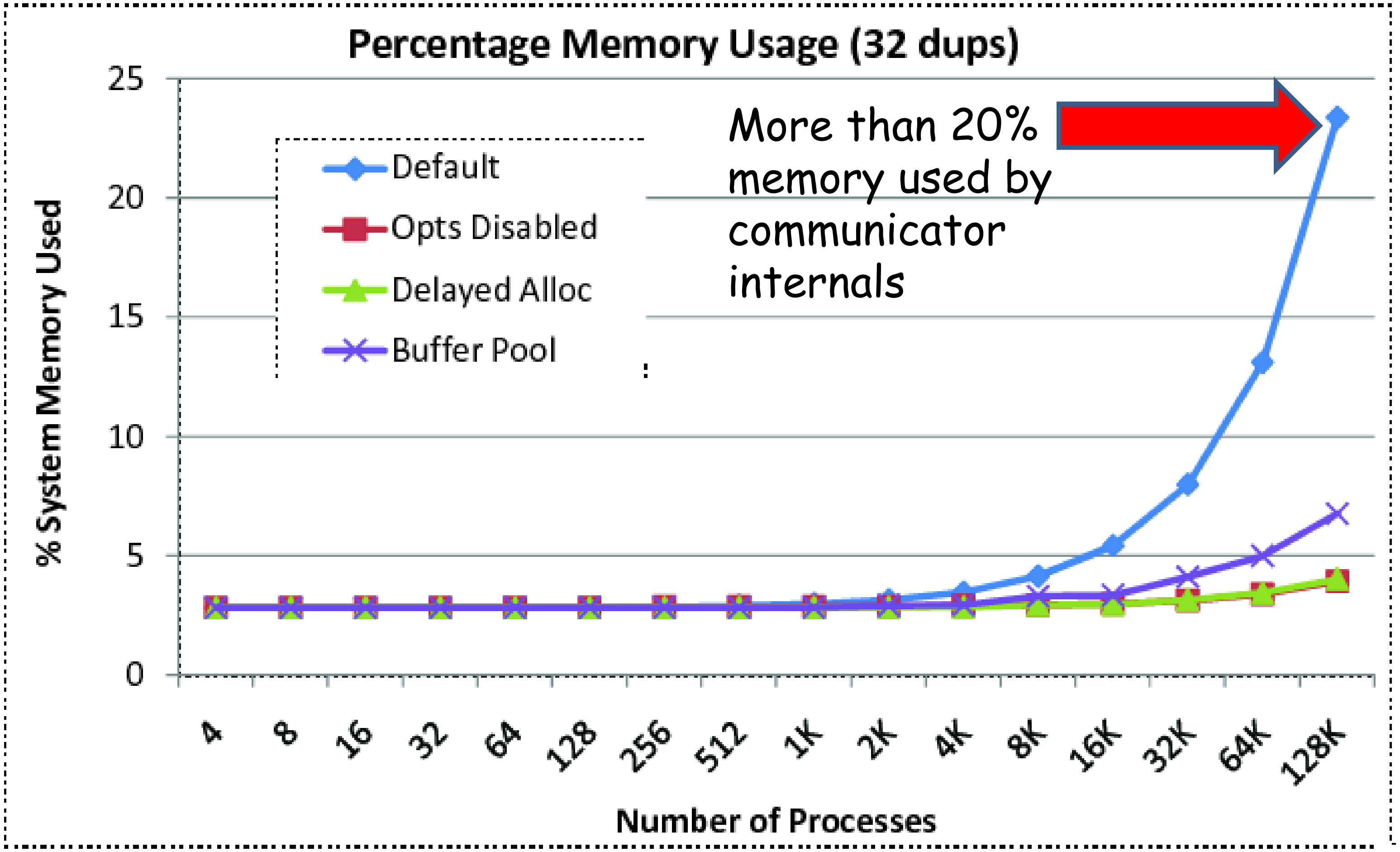
Experiment (1) at Argonne National Lab BlueGene system, 2011



Max. 8K communicators

NEK5000 application actually failed, ran out of memory, only 264 communicators possible

Experiment (2) at Argonne National Lab BlueGene system, 2011



NEK5000: Computational fluid dynamics code, often used (benchmark variant: NEKBONE), see https://nek5000.mcs.anl.gov/

On scalability of MPI, especially communicators and interface

P. Balaji, D. Buntinas, D. Goodell, W. Gropp, T. Hoefler, S. Kumar, E. L. Lusk, R. Thakur, Jesper Larsson Träff: MPI on millions of Cores. Parallel Processing Letters 21(1): 45-60 (2011)

H. Kamal, S. M. Mirtaheri, A. Wagner: Scalability of communicators and groups in MPI. HPDC 2010: 264-275

Somewhat recent (now routine) result with many, many MPI processes (little detail): MPI has so far been able to scale

P. Barnes, C. Carothers, D. Jefferson, J. LaPre: Warp speed: executing Time Warp on 1,966,080 cores. ACM SIGSIM-PADS, 2013

Topological awareness

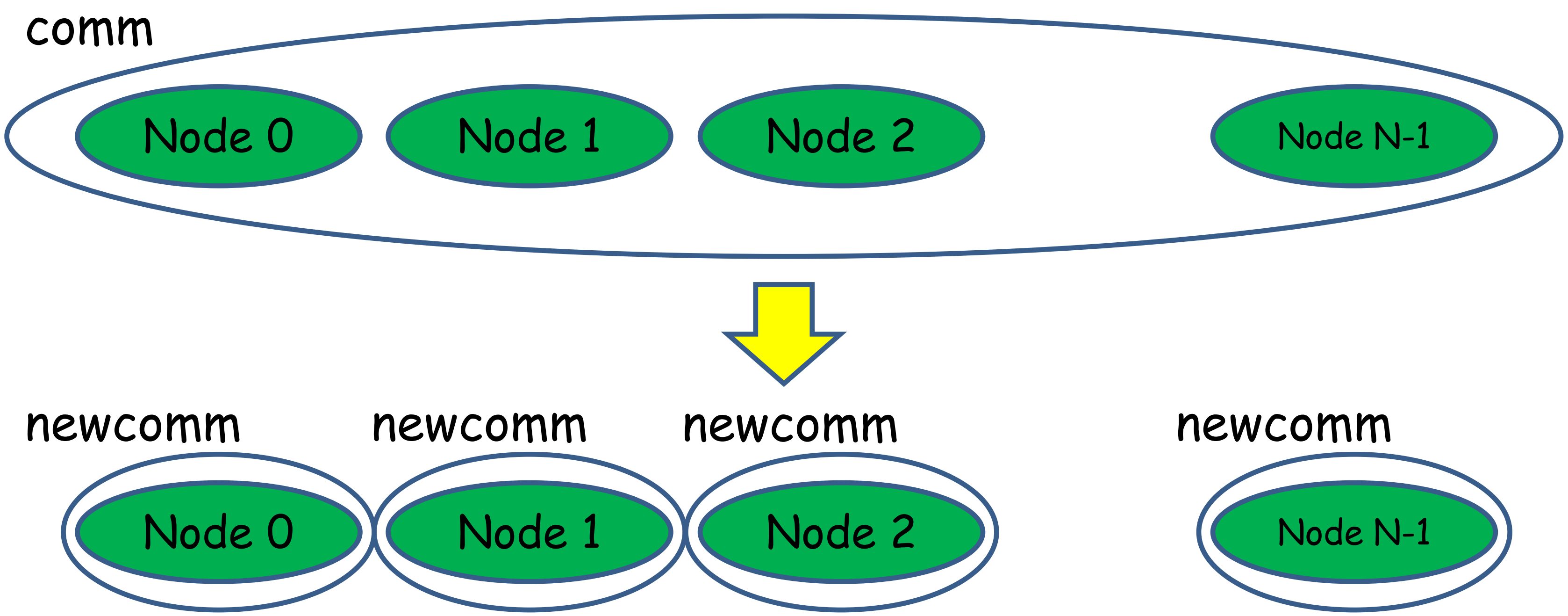
Orthogonal idea:
Instead of letting user specify communication requirements (communication topology), let system suggest good communicators subject to specified constraints

New in MPI 3.0, some support for better exploiting hierarchical systems/shared memory nodes

```
MPI_Comm_split_type(comm,MPI_COMM_TYPE_SHARED,key,
                    info,&newcomm);
```

`MPI_COMM_TYPE_SHARED`: Creates communicator for processes where a shared memory region can be created (processes on same shared-memory node)

```
MPI_Comm_split_type(comm,MPI_COMM_TYPE_SHARED,key,
                    info,&newcomm);
```



More to come with MPI 4.0

New split types with MPI 4.0

`MPI_COMM_TYPE_HW_UNGUIDED`, `MPI_COMM_TYPE_HW_GUIDED`: Splitting according to other hardware characteristics (NUMA nodes), either per default, or as specified in MPI_Info object