High Performance Computing
Advanced MPI implementations: Topologies

Jesper Larsson Träff
traff@par. ...
Institute of Computer Engineering, Parallel Computing, 191-4 Favoritenstrasse 16, 3. Stock
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
**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


## Open MPI canonical references


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

Practical advantages:

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

Hoare: CSP

Recent: GO

May: OCCAM

MPI message-passing model

Three main communication models:

- **Point-to-point**: Two processes explicitly involved
  - \( \text{MPI} \_\text{Send} \) \( i \rightarrow j \)
  - \( \text{MPI} \_\text{Recv} \) \( i \leftarrow j \)

- **One-sided**: One process explicitly involved
  - \( \text{MPI} \_\text{Put}/\text{Get}/\text{Accumulate} \) \( i \rightarrow j \)

- **Collective**: \( p \geq 1 \) processes explicitly involved
  - \( \text{MPI} \_\text{Bcast} \) \( i \leftrightarrow k \leftrightarrow l \leftrightarrow j \)
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

- 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)
Point-to-point communication

MPI_Send \rightarrow MPI_Recv

• 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

(*) Whether MPI_Send can complete may depend on MPI implementation
Point-to-point communication

- 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

- 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

- Largely asynchronous model: MPI_Ssend 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)

```c
int x[count];
MPI_Ssend(x,count,MPI_INT,tag,dest,comm);
```
Point-to-point communication

- 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)

```c
int x[count];
MPI_Issend(x, count, MPI_INT, tag, dest, comm, &request);
```
Point-to-point communication

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

```c
int x[count];
MPI_Recv(x,count,MPI_INT,MPI_ANY_TAG,
        MPI_ANY_SOURCE,comm,
        &status);
```
One-sided communication

- 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

**MPI_Put/Get/Accumulate**

- Introduced with MPI 2.0
- News in MPI 3.0
Collective communication

- 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**

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

MPI_Bcast

i <-> k <-> l <-> j

Synchronizing:

MPI_Barrier

Exchange, regular, rooted

MPI_Bcast
MPI_Gather/MPI_Scatter

MPI_Allggather
MPI_Alltoall

Non-rooted (symmetric)

Reduction, regular, rooted

MPI_Reduce

MPI_Allreduce
MPI_Reduce_scatter_block
MPI_Scan/MPI_Exscan

Non-rooted (symmetric)
Collective communication

**MPI_Bcast**

\[ i \leftrightarrow k \leftrightarrow l \leftrightarrow j \]

---

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.
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

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

```c
MPI_Comm_rank(comm, &rank);
MPI_Comm_size(comm, &size);
```
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 \( \text{comm1} \) may have to send state/data to the process with rank \( j \) in \( \text{comm2} \).

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

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
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
\[
\begin{pmatrix}
  y_0 \\
  y_1 \\
  y_2
\end{pmatrix} =
\begin{pmatrix}
  A_0 \\
  A_1 \\
  A_2
\end{pmatrix}
\begin{pmatrix}
  x_0 \\
  x_1 \\
  x_2
\end{pmatrix}
\]

$A_i$: $(m/p) \times n$ 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?**
Algorithm 2:
- Column-wise matrix distribution
- Compute local partial result vector
- \texttt{MPI\_Reduce\_scatter} to sum and distribute partial results

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

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 (proofs and constructions)
Factor $p = rc$, each process has $m/r \times n/c$ submatrix and $n/rc = n/p$ subvector.

Algorithm 3:
- Matrix distribution into blocks of $m/r \times n/c$ 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 $m/r \times n/c$ submatrix and $n/rc = n/p$ subvector

Step 1: Simultaneous `MPI_Allgather(v)` on all $c$ process columns
Factor $p = rc$, each process has $m/r \times n/c$ submatrix and $n/rc = n/p$ subvector

Algorithm 3:
- Matrix distribution into blocks of $m/r \times n/c$ 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 \( m/r \times n/c \) submatrix and \( n/rc = n/p \) subvector

Step 4: Simultaneous `MPI_Reduce_Scatter(_block)` on process row communicators
Factor \( p = rc \), each process has \( \frac{m}{r} \times \frac{n}{c} \) submatrix and \( \frac{n}{rc} = \frac{n}{p} \) subvector

Algorithm 3:
- Matrix distribution into blocks of \( \frac{m}{r} \times \frac{n}{c} \) elements
- Algorithm 1 on columns
- Algorithm 2 on rows

Conjecture:
\[
T(m,n,p) = O\left(\frac{mn}{p} + \frac{n}{c} + \frac{m}{r} + \log p\right), \quad p \leq \min(m,c,n) \]
Algorithm 3 is more **scalable**. To implement the algorithm, it is **essential** that independent, simultaneous, collective communication on (column and row) subsets of processes is possible.

Algorithm 3:
- Matrix distribution into blocks of $m/r \times n/c$ 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)
Observation:
For Algorithm 3, row and column subcommunicators are needed; must support concurrent collective operations.

Convenient to organize processors into a Cartesian $r \times c$ 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);

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);

Translation functions

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

MPI_Cart_rank(rccomm, rccoord, &rcrank);

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

```c
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: Collectives and Communicators: A Case for Orthogonality: (Or: How to get rid of MPI neighbor and enhance Cartesian collectives). EuroMPI 2020: 31-38
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:

```c
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):


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

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

---

See later
Cartesian neighborhoods in two dimensions

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**

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.
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.
oldcomm

If period[1]=1

newcomm

Torus network
If period[1]=1

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

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

oldcomm (bad order)

Process ranks in oldcomm and newcomm may differ

Torus network
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
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


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


TU Wien: New FWF project (with UniWien) on process mapping
Fully general application communication patterns

- 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

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

\[
\text{MPI\_Dist\_graph\_create\_adjacent}(\text{comm}, \text{indeg}, \text{sources}, \text{sourceweights}, \text{outdeg}, \text{destinations}, \text{destweights}, \text{info}, \text{reorder}, \&\text{graphcomm})
\]

may be more convenient and efficient

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

\[
\text{MPI\_Dist\_graph\_neighbors\_count}(\text{graphcomm}, ...) \\
\text{MPI\_Dist\_graph\_neighbors}(\text{graphcomm}, ...)
\]
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

**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

Don’t use MPI_Graph_create!

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 \( \Gamma \) from guest graph vertices to host graph vertices
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 $\Gamma: V \rightarrow V'$ be an injective mapping (embedding). The congestion of link $e$ in $E'$ wrt. $\Gamma$ is defined as

$$ Cong(e) = \left[ \sum_{(u,v) \in E} w(u,v) \text{ over } (u,v) \text{ in } E \text{ where } e \text{ in } R(\Gamma(u),\Gamma(v)) \right] / c(e) $$

The congestion of $\Gamma$ 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
- \(c(u_1,u_2) = |V|\), \(c(e) = 1\) 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 $\Gamma$ from $G$ to $H$ determines a partition of $V$ into $V_1$ and $V_2$ with $|V_1|=|V_2|$, namely $V_i = \{v \mid \Gamma(v) \in U_i\}$

Any mapping that minimizes the congestion must map $(s,t)$ to $(u_1,u_2)$. This gives a congestion of at most $|V|^3+|V|$; any other mapping has congestion at least $|V|^4$.

In such a mapping, the most congested edge is $(u_1,u_2)$, with congestion

$$C = |V|^3+|\{(v_1,v_2) \in E \mid v_1 \in V_1 \text{ and } v_2 \in V_2 \}|/|V|$$

Thus, a solution to $CONG$ with congestion at most $|V|^3+L/|V|$ 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 from

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

ND17, see

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) \( \Gamma: V \to V' \) that minimizes

\[
\sum_{(u,v) \in E} w(u,v)c(\Gamma(u), \Gamma(v))
\]

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

J. L. Träff: Implementing the MPI process topology mechanism. SC 2002:1-14
Heuristics for Cartesian topology mapping (with arbitrary neighborhoods)


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/](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/](http://viem.taa.univie.ac.at/)

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=view]s/metis


**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/]

Graph partitioning software for process topology embedding

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

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)

```c
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 communicator
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.

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

for (i=0; i<rsize; 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);
```

Better?

Process group operations
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.

```c
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

```
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…)\[1\]

Example of performance guideline that can be verified by benchmarking.

If not fulfilled, something wrong with MPI library

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-to-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?

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?

```c
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
```c
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 $\approx 1$MByte 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

NEK5000 application actually failed, ran out of memory, only 264 communicators possible.
Experiment (2) at Argonne National Lab BlueGene system, 2011

More than 20% memory used by communicator internals.
NEK5000: Computational fluid dynamics code, often used (benchmark variant: NEKBONE), see https://nek5000.mcs.anl.gov/
On scalability of MPI, especially communicators and interface


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

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

```c
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);