Introduction to Parallel Computing
Distributed memory systems and programming

Jesper Larsson Träff
Vienna University of Technology
Parallel Computing

©Jesper Larsson Träff
Distributed memory parallel architectures and models

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

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

Cost model:

- Communication
- Local vs. non-local memory access
Distributed memory parallel programming models

“Communication driven”: Explicit message passing between pairs or groups of processors

Examples: MPI, PVM, Occam, ...

“Data distribution driven”: Partitioned Global Address Space (PGAS) model specifies distribution of data (structures) over processors, communication implicit through data access

Examples: UPC, Titanium, Chapel, X10, Global Arrays, HPF, ...

Not in this lecture
„Pure“ distributed memory system architecture:

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

Network properties:

- Structure: topology
- Capabilities: one or several operations per network component
- Routing technique
- Switching strategy
- Cost: Latency and bandwidth

This lecture: A little bit about topology
Network properties: Latency and bandwidth

Network latency: time to get first data element from source to destination

Network bandwidth: How many data elements per time unit (seconds) can be transferred from source to destination

Not (much) in this lecture:
Cost models for network communication

Latency, bandwidth: one pair of communicating processes (no load)? Under load (congestion)? Which loads?
Network properties: Topology

**Topology** modeled as (un)directed graph $G=(V,E)$

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

$(u,v) \in E$: there is a direct link (cable, optical connection, ...) from element $u$ to element $v$
**Direct network:**
The only network elements are processors, no extra switch nodes

- Typical examples (see later):
  - mesh/torus, ring, hypercube, Kautz-network, ...

**Indirect network:**
Processors connected via dedicated switch nodes

- Examples:
  - Fat tree, Clos network, butterfly network, Dragonfly network...
diameter($G$): $\max(\{|\text{shortest path}(u,v)|\}$ over all $u,v$ in $V$)

| Lower bounds number of communication rounds for collective communication operations |

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

| „cost factor“. High-degree gives potential for more simultaneous communication (multi-port) |
bisection width($G$): minimum number of edges to remove to partition $V$ into two (roughly) equal-sized, disconnected parts

bisection width($G$): $\min(|\{(u,v) \in E, u \in V_1, v \in V_2\}|)$ over all partitions $V_1, V_2$ of $V$ with $|V_1| \approx |V_2|$

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

Note: Finding bisection width of arbitrary topology is NP-hard: Graph Partitioning
Broadcast (collective operations) in communication networks

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

- Assume all processors "know" which processor is root.
- Assume data are indivisible units (still no cost model).
The ideal case: Fully connected network

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

- **Diameter:** 1
- **Bisection width:** \((p/2)^2\)

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

**1-ported communication system:** In each communication operation/step, a processor can send/receive to/from at most one other processor.
The ideal case: Fully connected network

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

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

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

$k$-ported communication system: In each communication operation/step, a processor can send/receive to/from at most $k$ other processors
The ideal case: Fully connected network

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

- **Unidirectional**: A processor can either send OR receive in a communication step
- **Bidirectional**: Processor can send AND receive

\[
\text{diameter} = 1
\]

\[
\text{bisection width} = (p/2)^2
\]

**Expensive**: \( p^2 - p \) links (cables, switch-ports, …), degree = \( p-1 \)
The ideal case: Fully connected network

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

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

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

**Bidirectional**: A processor can send AND receive

- to/from same processor: telephone model
- to/from two different processors: send-receive bidirectional
Broadcast in fully connected, 1-port ed network

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

Algorithm:
1. If |V|=1 done
2. Partition processors into two roughly equal-sized sets V1 and V2
3. Assume root r in V1, choose local root rr in V2
4. Send data from r to rr
5. Recursively broadcast in V1 and V2
Algorithm:
1. If $|V|=1$ done
2. Partition processors into two roughly equal-sized sets $V_1$ and $V_2$
3. Assume root $r$ in $V_1$, choose local root $rr$ in $V_2$
4. Send data from $r$ to $rr$
5. Recursively broadcast in $V_1$ and $V_2$

Analysis:
Assume communication takes place in synchronized communication rounds. After Step 4, two problems of half the original size are solved independently. Algorithm therefore takes $\lceil \log_2 p \rceil$ rounds for all processors to have received data

Note: $\lceil \log_2 p \rceil > \text{diameter}(G)=1$. Can we do better?
Algorithm:
1. If $|V| = 1$ done
2. Partition processors into two roughly equal-sized sets $V_1$ and $V_2$
3. Assume root $r$ in $V_1$, choose local root $rr$ in $V_2$
4. Send data from $r$ to $rr$
5. Recursively broadcast in $V_1$ and $V_2$

Fundamental lower bound:
At least $\lceil \log_2 p \rceil$ communication rounds are needed for the broadcast problem.

Proof:
In each round the number of processors that have the data can at most double (namely when each processor sends to a processor that did not have data). Therefore, $\lceil \log_2 p \rceil$ rounds are required.
Algorithm:
1. If $|V|=1$ done
2. Partition processors into two roughly equal-sized sets $V_1$ and $V_2$
3. Assume root $r$ in $V_1$, choose local root $rr$ in $V_2$
4. Send data from $r$ to $rr$
5. Recursively broadcast in $V_1$ and $V_2$

Theorem:
Recursive algorithm matches lower bound on number of communication rounds and is therefore $\Theta(\text{ceil}(\log_2 p))$

Note:
Bidirectional communication is not used, and cannot speed up the broadcast operation
The worst case: Linear array, ring, tree

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

\[
diameter = p-1 \quad (p/2 \text{ for ring}) \\
\text{diameter} = 2 \log_2 \left(\frac{p+1}{2}\right)
\]

Both: Diameter determines broadcast complexity
The worst case: Linear array, ring, tree

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

degree = 2  
degree = 3
Mesh, torus

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

Both: diameter determines broadcast complexity

"wrap-around" for tori
Mesh, torus

"wrap-around" for tori

\[
\text{bisection width(mesh)} = p^{(d-1)/d} \\
\text{bisection width(torus)} = 2p^{(d-1)/d}
\]

Both: Bisection bandwidth determines transpose/alltoall communication complexity
Hypercube

- Diameter = $k (= \log_2 p)$
- Bisection width = $p/2$
- Degree = $k (= \log_2 p)$

Diameter determines broadcast complexity

$k$ dimensional hypercube composed from $2 \times (k-1)$ dimensional hypercubes
Examples

NEC Earth Simulator (2002-2004): fully connected, 1-ported, bidirectional send-receive

Blue Gene, Cray, Fujitsu K-1, (dead) Blue Waters: mesh/torus (+ collective tree network), d-ported, bidirectional send-receive

InfiniBand: Multi-stage fat-tree, 1-ported, bidirectional send-receive
Other topologies

Kautz-network: Directed graph, with fixed degree $d$ and dimension $n$, has $|V|=(d+1)d^n$ vertices; smallest possible diameter among all graphs with degree $d$ and $|V|$ vertices

SciCortex, until 2009

Many early parallel systems (90ties) were hypercubes (Intel iPSC, nCube, ...)

Butterfly-network essential for (experimental) parallel systems that emulate a PRAM (SBPRAM, Saarbrücken, mid-90ties)
Indirect multistage networks: (Fat) tree, Clos, Benes, Omega, ...

Switch: typically full n x m crossbar
Routing terminology

Processor u (source) needs to send message to processor w (destination)
• Since u and w not necessarily adjacent, path (with intermediate node v) has to be selected
• In the presence of other traffic in network

Accomplished by a routing algorithm/protocol
Routing terminology

**Correctness**: Routing protocol must be **deadlock free**: any message sent from some node to another must arrive, regardless of which other messages are sent between other nodes concurrently.

**Performance**: Minimize delays of messages (due to other messages that have to go through the same node); bound buffer usage at nodes (for storing parts of messages); avoid contention (“hot spots”); ...
Routing terminology

**Deterministic (oblivious) routing:** The path followed by a message determined only by source and destination and static network properties

**Adaptive routing:** Path determined dynamically (based on, e.g., network traffic)

**Minimal routing algorithm:** Routes along a shortest path from source to destination; a non-minimal algorithm may choose a longer path (typical for adaptive schemes; deflection routing at congested nodes)
Routing terminology

Circuit switching: Entire path from source to destination is reserved for the duration of the entire transmission.

Packet switching: Message partitioned into packets that can be transferred independently, possibly along different paths (note: may not arrive in order at destination; reassembly needed).

Store-and-forward: For each intermediate node on the path from source to destination, whole packet (message) is stored, before being sent along (forwarded to next node along) the path.

For concrete algorithms: Lecture HPC.
Transmission cost model

Simple, first assumption (more accurate models: LogP)

Cost of transmitting (indivisible) data of size $m$ along edge $(u,v)$ in communication network linear in $m$; both $u$ and $v$ involved during whole transmission

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

$\alpha$: „start-up“ latency
$\beta$: time per unit (Byte)

In this model:
Recursive/binomial tree broadcast: $\log_2 p(\alpha + \beta m)$
Lower bound on broadcast in linear cost, fully connected network model is

$$\max(a \log_2 p, a + \beta m)$$

- \(a \log_2 p\): \(\log_2 p\) communication rounds, each communication incurring one „start-up“
- \(\beta m\): the \(m\) data units have to leave the root

**Why not \((\log_2 p)(a + \beta m)\) ?**

**Answer:** \(m\) need not be sent as one unit, „pipelining“

**Question:** possible to achieve both lower bounds?

**Answer:** yes

**HPC lecture**
Source-destination transmission time

Path of length $l$

Store-and-forward: $l(a+\beta m)$

Packet switching (with same path): apply pipelining, packet size $b$, number of packets $\lceil m/b \rceil$

Delay of first packet

$T(m) = l(a+\beta b) + (m/b-1)(a+\beta b) = (l+m/b-1)a + (l+m/b-1)\beta b$

$= (l-1+m/b)a + (l-1)\beta b + \beta m$

When $m$ large, $l$ play smaller role
Hybrid/hierarchical HPC architectures

Processing elements organized hierarchically: “Nodes” with shared memory, interconnected by network

Example: Traditional SMP cluster

**Issue:** Many cores share single (a few) network ports
• Two- or more level hierarchy (nodes, racks, islands, …)
• Different communication capabilities at different levels (shared-memory/k-ported network, …), simple $\alpha+\beta m$ cost model may be (too) inaccurate

Network bottleneck: Number of ports to network (much) smaller than number of cores per node
Shared vs. distributed: A matter of degree...

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

Programming-model wise: It can make sense to treat as distributed memory system - to emphasize locality.
TU Wien parallel computing hybrid distributed memory machine

- 36 shared-memory nodes
- InfiniBand QDR switch,
- Node with 2x8-core AMD "magny cours" processor, 2.3GHz
- 32 GByte shared-memory/node
- 1TB local disk/node

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

Exercise:
What is the theoretical peak performance?
Mellanox InfiniBand switch MT4036

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

System configuration by NEC

Basic software:
- NEC MPI
- mpich2 MPI
- OpenMPI

Projects: Are there performance differences between these MPI libraries?
Message-passing parallel programming model

- Finite set of “processes” with local memory (and program)
- Communicate explicitly by sending and receiving messages, all processes explicitly involved (exception: one-sided communication)
- No implicit synchronization between processes
- No other means of communication (shared nothing)

Assumption: Each process allowed to communicate (exchange messages) with any other process. Processes are named (e.g., integer rank)
Advantages:

• All data exchange explicit (when, what, where): means to reason about correctness
• (largely) Deterministic; no race conditions
• Locality key to performance

Rule of thumb “performance model”:
Communication expensive, local computation cheap; optimize for locality and rare communication of large messages

Roots in Dijkstra/Hoare school: Communicating Sequential Processes (CSP)

Message passing abstraction/programming model

- Finite set of sequential processes communicate through a communication medium; communication between all processes possible: Abstracts physical network properties
- Processes communicate by (explicitly) sending and receiving messages: message-passing paradigm
- No implicit synchronization between processes, only communication
Communication medium realized by some physical communication network

Communication „medium“: some network

Routing protocol (algorithm) implements correct, reliable, deadlock-free communication between message passing processes. Not part of model but handled by “lower layers”
MPI: The Message-Passing Interface

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

Intended for:

- Single applications with non-trivial communication requirements for HPC systems and clusters with non-trivial communication capabilities
- HPC applications (almost) exclusively use MPI
- Many, many, many parallel applications for clusters, medium sized, and very large systems
- *Paradigmatic* realization of the message passing model
- Well-engineered standard, *lots to learn for other interfaces*
MPI design principles, imperatives and goals

• **High-performance**: Communication functions close to typical “hardware” functionality, low protocol stack overhead

• **Portability and Scalability**

• **Memory efficient**: little dynamic memory (\(O(1)\)?) needed by MPI functions, memory (communication buffers) in user-space

• Coexist with other parallel interfaces (OpenMP, threads, …)

• Support (not hinder) construction of tools

• Support heterogeneous systems (data representation)

• Support SPMD or MIMD paradigm

• Support library building, application specific libraries
SPMD: Same Program, Multiple Data

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

MIMD: Multiple Programs, Multiple Data

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

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

- **Library**, not a programming language!

- **Pros**: Can be developed independently of compiler support; bindings for C and Fortran (*not* C++ anymore), maximum freedom for library developer.

- **Cons**: Things that compiler knows cannot readily be exploited, user sometimes have to convey information from language (data layouts) to library.

**Same considerations for thread support (pthreads library)**
MPI is large:

- 306 C functions in MPI 2.2 (Sept. 2009)
- 406 C functions in MPI 3.0 (Sept. 2012)
- 416 C functions in MPI 3.1 (June 2015)

but centered around few basic concepts
- **Natural functionalities**: Consult standard for concrete details
- (almost) All MPI functions are there for good reasons and have applications

**Note**: Since ca. 2009 not good practice to distinguish between MPI 1 and MPI 2. MPI 3.0&3.1 still new enough that distinction should be made
MPI programming model

1. Set of **processes** (in communication domain) that can communicate
2. Processes identified by **rank** in communication domain
3. Ranks successive 0, ..., p-1; p number of processes in domain (**size**) 
4. More than one communication domain possible; created relative to default domain of all started processes
5. Processes operate on **local data**, all communication explicit
6. Communication **reliable** and **ordered**
7. Three basic communication models:
   i. Point-to-point communication – different modes, non-local and local completion semantics
   ii. One-sided communication – different synchronization mechanisms, local completion mechanisms
   iii. Collective operations, non-local completion semantics (*)

8. Structure of communicated data orthogonal to model/mode

9. Communication domains may reflect physical topology

10. No communication cost model

(*) MPI 3.0 features non-blocking collective operations
MPI communication models

- **Point-to-point**: `MPI_Send` ➔ `MPI_Recv`

- **One-sided**: `MPI_Put`

- **Collective**: `MPI_Bcast` ➔ `MPI_Bcast` ➔ `MPI_Bcast`
Extended “communication”

- Parallel I/O:

- Process management:

- Virtual topologies:

Not this lecture
Library building

- **Communicator management:** creating/freeing communicators
  - e.g., `MPI_Comm_create`

- **Attributes:** Additional information attached to MPI objects

- **Datatypes:**
  - e.g., `MPI_Type_vector`
Safe parallel libraries

Communication inside library independent of/isolated from communication outside library, no (semantic) interference

\texttt{MPI\_Comm\_dup(comm, newcomm);}

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

MPI attribute mechanism not in this lecture
Basic MPI concepts

1. **Communicators** (process groups, windows) represent **sets of processes** that can communicate; communication relative to one communicator **cannot (semantically) interfere** with communication relative to other communicator.

2. **Datatypes** describe layout/structure of data to be communicated.

3. Completion semantics of communication operations **local** (not depending on action of other process) or **non-local** (depending on action of other process).

4. Communication calls **blocking** (until communication has been locally completed) or **non-blocking**.
MPI standard

Not a formal specification, tries to be (and is quite) precise, but sometimes (intentionally) vague...:

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

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

(**) specific requirements might not be feasible for all communication systems; could limit portability of MPI
Before MPI (early 90ties)

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

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

The application portability problem

Lots of commonalities, need for a standard (ca. 1994)
**Code/application portability:**
Application developed on system A will run unchanged on system B; perhaps with recompilation/relinking. No code change/work-around needed

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

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

Shared-memory models (memory consistency, atomic operations, ... architecture dependency), GPU models, ... may not
"Performance portability": Could mean: No change in application needed to efficiently exploit system B with code developed on system A

Distributed memory programming model may provide some performance portability: All communication explicit and delegated to library (MPI)

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

Major (performance) portability HPC disruption: transition from "vector" to "scalar" systems late 90ties. Consult Top500
Evolution of the MPI Standard

MPI 1.0, 1.1, 1.2: 1994-1995
- Point-to-point and collective communication, datatypes, ...

MPI 2.0: 1997
- One-sided communication, parallel I/O, dynamic process management

MPI 2.1: 2008
- Consolidation

MPI 2.2: 2009
- Scalable topologies, new collectives

Implementations:
ANL: mpich, 1996
NEC: MPI/SX, 2000
mpich2, 2004
OpenMPI, 2006
Evolution of the MPI Standard

**MPI 3.0: 2012**
- Non-blocking collectives,
- topological (neighbor) collectives,
- extended one-sided communication,
- performance tool support
- C++ bindings removed
- Still no fault-tolerance support

**MPI 3.1, 2015**
- Fixes
- Non-blocking I/O

Implementations:

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

MPI 3.0: Vienna, Sept. 21, 2012

MPI 3.1
June 2015

MPI 3.0: Vienna, Sept. 21, 2012

© Jesper Larsson Träff
MPI Forum: Towards MPI 3.0, MPI 3.1 (MPI 4.0?)

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

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

- 30-50 organizations involved, about 30 participants at meetings
- All major MPI developers (mpich, openMPI, mvapich,…), all major vendors, major labs with applications
MPI Forum: MPI 3.0 (and onwards)

- 5 year long process
- Non-blocking collectives - for performance and convenience
- Topological (neighbor) collectives - for scalability and expressivity
- Extended one-sided communication support - for performance and convenience
- Performance tool support - controlled access to internal state of library
- C++ bindings have been **removed**
- New, correct Fortran bindings

Much to do for library implementers (and application programmers) for the coming years. **Not the last word**
Additional literature:

MPI standard, MPI 3.1 [www.mpi-forum.org/docs/mpi-3.1/mpi31-report.pdf]

Michael J. Quinn: Parallel Programming in C with MPI and OpenMP, McGraw-Hill 2003
Peter S. Pacheco: Parallel Programming with MPI, Morgan-Kaufmann, 1997
First MPI program

```c
#include <mpi.h>

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

    MPI_Init(&argc,&argv);

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

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

    MPI_Finalize();
    return 0;
}
```
First MPI program

```c
#include <mpi.h>

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

    MPI_Init(&argc,&argv);

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

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

    MPI_Finalize();
    return 0;
}
```

Standard MPI header
FORTRAN: INCLUDE „mpif.h“
#include <mpi.h>

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

    MPI_Init(&argc,&argv);

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

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

    MPI_Finalize();
    return 0;
}
First MPI program

```c
#include <mpi.h>

int main(int argc, char *argv[]) {
    int rank, size;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&size);
    MPI_Comm_rank(MPI_COMM_WORLD,&rank);
    fprintf(stdout,"Here is %d out of %d\n",rank,size);
    MPI_Finalize();
    return 0;
}
```

Initial communication context, set of processes

Who am I?
Compiling and running MPI programs

• `mpicc`, `mpif77`, `mpif90` - like `cc`, `f77`, `f90`
• `mpirun -np <procs>` ...

• Batch system: slurm

This year (2017): Either OpenMPI or mpvapich2.

For use with slurm: load the corresponding module
MPI conventions

“Namespace”, C

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

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

“Namespace”, Fortran

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

MPI constants (`MPI_SUCCESS`, `MPI_INT`, ...) allCAPS

`MPI_` - prefix reserved, don’t use in own programs (illegal)
**Good practice** to always check error status; MPI programmers most often don’t...

Error behavior can be controlled to some extent by error handlers

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

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

**BUT(!):** „text that states that errors *will* be handled, should be read as *may* be handled“, MPI 2.2, p. 276, MPI 3.1, p. 340

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

Major discussion in MPI Forum on error handling and fault tolerance

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

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_SUCCESS</td>
<td>Success</td>
</tr>
<tr>
<td>MPI_ERR_BUFFER</td>
<td>Buffer error</td>
</tr>
<tr>
<td>MPI_ERR_COUNT</td>
<td>Count error</td>
</tr>
<tr>
<td>MPI_ERR_TYPE</td>
<td>Type error</td>
</tr>
<tr>
<td>MPI_ERR_TAG</td>
<td>Tag error</td>
</tr>
<tr>
<td>MPI_ERR_COUNT</td>
<td>Count error</td>
</tr>
<tr>
<td>MPI_ERR_RANK</td>
<td>Rank error</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>MPI_ERR_UNKNOWN</td>
<td>Unknown error</td>
</tr>
<tr>
<td>MPI_ERR_TRUNCATE</td>
<td>Truncate error</td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>MPI_ERR_WIN</td>
<td>Window error</td>
</tr>
<tr>
<td>MPI_ERR_LASTCODE</td>
<td>Last code error</td>
</tr>
</tbody>
</table>

Sometimes returned in point-to-point

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

“language independent”:

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

- **IN** sendbuf
- **OUT** recvbuf
- **IN** count
- **IN** datatype (handle)
- **IN** op (handle)
- **IN** root
- **IN** comm (handle)
C prototype

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

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

FORTRAN binding

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

Handles are INTEGERS (problems with F90 typing)
The 6 basic functions

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

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

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

MPI_Comm_rank(MPI_COMM_WORLD,&rank);
MPI_Comm_size(MPI_COMM_WORLD,&size);
Process rank i:

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

Data transferred from i to j

Process rank j:

```c
int b[N];
float area;
MPI_Recv(b,N,MPI_INT,i,TAG1,MPI_COMM_WORLD,&status);
MPI_Recv(&area,1,MPI_FLOAT,i,TAG2,MPI_COMM_WORLD,
        &status);
```
Get time (in micro-seconds with suitably high resolution) since some time in the past:

```c
double point_in_time = MPI_Wtime();
```

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

```c
MPI_Barrier(MPI_COMM_WORLD);
```
Beware: Most often means on one node only!

Communication, processes, communicators

\texttt{mpirun -np `<procs>` `<program>` ... <options to program>}

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

Same `<program>` will run on all resources: SPMD

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

Note:
Options not standardized, see local installation&manpages, try mpirun -help
<program> executes

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

**MPI_COMM_WORLD**: Initial communicator containing all started processes. *Static: never changes!*

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

**MPI Rule**

- Communication in MPI is always relative to a communicator
- Processes are identified by their ranks in some communicator
Communicator: distributed, global object, communication context, finite set of ordered processes that can communicate.

Physical processors may run more than one MPI process (but most often only one).
Binding of MPI processes (static) to processors. Done outside of MPI; not part of standard.

MPI process identified by rank in comm

Communicator

MPI process ranks

Processor/core

Environment and options to mpirun control binding
Communicator object:

- All processes in a communicator can communicate
- All models (point-to-point, one-sided, collective; all other functionality)
- Has a size: number of processes
- Each process has a rank \((0 \leq \text{rank} < \text{size})\)

\[
\begin{array}{cccccccc}
0 & 1 & 2 & 3 & 4 & 5 & 6 \\
\end{array}
\]

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

**MPI process:** (normally) statically bound to some processor; can have different ranks in different communicators; canonically identified by rank in `MPI_COMM_WORLD`
**Good SPMD practice:**
Write programs to work correctly for any number of processes

```c
MPI_Comm_Size(MPI_COMM_WORLD, &size);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
if (size<10 || size>1000000) MPI_Abort(comm, errcode);
if (rank==0) {
   // code for rank 0; may be special
} else if (rank%2==0) {
   // remainder even ranks
} else if (rank==7) {
   // another special one
} else {
   // all other (odd) processes – perhaps do nothing?
}
```

**Dangerous practice (type error):** Don't rely on C convention for Boolean expressions if (rank) {...}
Communicator object:

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

```c
MPI_Comm_dup(comm, &comm1);
```
MPI_Comm_dup(comm, &comm1);

Duplicates communicator: Same processes, but different communication domain.
Communication wrt. comm cannot interfere with communication wrt. comm1

First example of collective operation: All processes in comm must call MPI_Comm_dup
**Good practice** for writing libraries

```c
int my_library_init(comm, &libcomm)
{
    MPI_Comm_dup(comm, &libcomm); // store somewhere

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

Communication inside library use libcomm, and will never interfere with communication using other communicators.

**MPI_Comm_dup:**
Collective operation, **MUST** be called by all processes in comm
MPI handles

**MPI_COMM_WORLD**, comm1, comm2, ...:
An MPI (predefined) **handle** used to access MPI objects (communicators, windows, datatypes, attributes)

- Handles are (almost always) **opaque**: Internal MPI data structures cannot be accessed; but only manipulated through the **operations** defined on them

- MPI does not define how handles are represented (index into table, or pointer, or ...)

- Handles in C and Fortran **may be different**

Example: `MPI_Comm_f2c(comm)`:
Returs C handle of Fortran communicator (**no error code here**)
Other MPI handles

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

- **MPI_Datatype**: description of data layouts (basic/primitive - or user-defined/derived)
- **MPI_Op**: binary operators (built-in or user-defined)

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

- **MPI_Errhandler**: ...
- ...
Splitting communicators

Partition set of MPI processes (communicator) into disjoint sets (communicators) that can communicate independently

```c
MPI_Comm_split(comm1, color, key, &comm2);
MPI_Comm_create(comm1, group, &comm2);
...
```

Calling MPI process may have different ranks in comm1 and comm2

- **color, key**: integers determine subsets and relative order
- **group**: `MPI_Group` of ordered processes
Example: Even-odd split

Even-numbered processes (in communicator comm1) shall work together, independently of the odd-numbered processes

General use: divide-and-conquer type algorithms (Quicksort-like)

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

// find process' new role in comm2
MPI_Comm_size(comm2,&newsize);
MPI_Comm_rank(comm2,&newrank);
```

Even numbered processes (in comm1) have color==0, odd processes color==1
Odd processes in comm1

Even processes in comm1
MPI_Comm_rank(comm1,&rank);

MPI_Comm_split(comm1, color, key,&comm2);

// process' role in comm2
MPI_Comm_size(comm2,&newsize);
MPI_Comm_rank(comm2,&newrank);

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

- Calling processes are sorted into groups of same color
- In each group, processes are sorted according to key, ties broken by rank in calling communicator
Example: Master-worker (\textit{non-scalable, use with care})

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

For workers: \texttt{MPI\_Barrier(comm)}, \texttt{MPI\_Allgather(comm)}, … (collective operations) \texttt{deadlock}, if master is doing something else

\textbf{Solution}: Split communicator, workers only communicator
Solution 1: Splitting

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

**MPI_COMM_SELF:** Communicator with only process itself, size==1

**MPI_COMM_NULL:** No or invalid communicator
Solution 2: Process groups

```
MPI_Comm_group(comm,&group); // get processes in comm
ranklist[0] = 0; // rank 0 to be excluded
MPI_Group_excl(group,1,ranklist,&workgroup);
MPI_Comm_create(comm,workgroup,&workers);
// rank 0 (in comm) not in workers
// workers==MPI_COMM_NULL for rank 0 in comm
```
Communicator: A distributed, **global object** that can be manipulated through **collective operations** (*MPI_Comm_split*, *MPI_Comm_dup*, …)

Process group (*MPI_Group*): A **local object**, ordered set of processes that can be manipulated locally by an MPI process

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

**Use:**
Building special communicators, one-sided communication
MPI_Comm_free(&comm);

frees created communicator comm

Note:
MPI_COMM_WORLD and MPI_COMM_SELF cannot be freed

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

©Jesper Larsson Träff
What is in a communicator (hidden in the implementation)?

Each process locally maintains:

1. List of processes (ordered set): The process group
2. Mapping from rank to process to processor (implicit or explicit)
3. Communication context: A tag to identify communication on this communicator
Communicators, summary

Predefined communicators:
- **MPI_COMM_WORLD**: All started processes
- **MPI_COMM_SELF**: Singleton communicator for each process, only this process

Communicator are static objects, cannot change, processes cannot come and go. New communicators can be created from old ones:
- **MPI_Comm_split**
- **MPI_Comm_create (+ MPI process groups)**
- ...  

Free after use:
- **MPI_Comm_free**
SPMD: All processes run same user program

User program

communicator

User program

MPI_Send

MPI_Recv

MPI run-time library

MPI run-time library

Communication network

©Jesper Larsson Träff
Point-to-point communication

• Point-to-point: MPI_Send → MPI_Recv

Communication between two processes. Both explicitly involved in communication, one as sender, one as receiver.
int THISMSG=777; // message TAG
int count = 500;
if (rank==2) {
    int sendbuf[500] = {<the data>};
    MPI_Send(sendbuf,count,MPI_INT,4,THISMSG,comm);
} else if (rank==4) {
    int recvbuf[600]; // as large as send count
    MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,&status);
}
MPI_Send(void *sendbuf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm);

sendbuf: address of (pointer to) data to be sent
count: number of units to be sent
datatype: structure of the communication unit
dest: rank of process to receive data
tag: tag (type) of message
comm: the communicator

The unit is often some simple datatype in C (int, float, char, double, ...), but can be objects with structure
MPI_Send(sendbuf, count, datatype, dest, tag, comm);

int sendbuf[500] = {<the data>};
count = 500;
MPI_Send(sendbuf, count, MPI_INT, 4, THISMSG, comm);

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

sendbuf: (start address of)  

Only rank 4 in comm can ever receive this message

Described by datatype MPI_INT

C int
MPI_Recv(void *recvbuf, int count, MPI_Datatype datatype,
            int source, int tag, MPI_Comm comm,
            MPI_Status *status);

recvbuf: address of (pointer to) data to be received
count: expected number of units to be received
datatype: structure of the communication unit
source: rank of process that sent data
tag: tag (type) of message
comm: the communicator
status: details of received message

The unit is often some simple datatype in C (int, float, char,
double, ...), but can be objects with structure
MPI_Recv(recvbuf,count,datatype,source,tag,comm,*status);

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

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

recvbuf: (start address of) C int

0 500

Described by datatype MPI_INT
int sendbuf[500] = {<the data>};
count = 500;

MPI_Send(sendbuf,count,MPI_INT,4,THISMSG,comm);
sendbuf[27] = ...; // data for next operation

Call returns when it is **safe** to reuse `sendbuf`, all data have been taken care of. **Nothing** guaranteed about what has happened on rank 4 (message received or not), but buffer can be reused.
int recvbuf[600]; // large enough
count = 600; // equal or larger to what is being sent
success =
MPI_Recv(recvbuf,count,MPI_INT,2,THISMSG,comm,&status);

Returns when a message from rank 2 has been received;
information about data in status object. Forever, if nothing is
sent from 2
Status object (half opaque): Information on communication

MPI_Status status; // status handle
MPI_Recv(...,&status);

Status contains information on what was received:

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

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

Why? Don’t we know this??

If so: Consider MPI_STATUS_IGNORE as status argument in MPI_Recv
Status object (half opaque): Information on communication

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

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

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

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

**Note:** With basic datatypes (MPI_INT etc.) `MPI_Get_count` and `MPI_Get_elements` are equivalent
#include <assert.h>

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

int realcount;
MPI_Get_count(status,MPI_INT,&realcount);
// possibly realcount<count;
assert(realcount<=count);
Point-to-point semantics

**MPI_Send(sendbuf,...,rank,tag,comm):**

Initiates and completes send to designated process rank:

- **Completion is non-local:** Call return may depend on receiving rank having initiated receive operation (*)
- **Operation is blocking:** When call returns all of buffer can be reused
- **Sent messages to same rank with same tag are non-overtaking**

**MPI_Recv(recvbuf,...,rank,tag,comm,&status):**

Completes receive operation from specific rank, or **ANY**

- **Operation is blocking:** Call returns when full message has been received

(*) but may also buffer message
Point-to-point communication succeeds if

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

Comments:

1. Mistakes normally caught by MPI_Send: error (abort)
2. If not, deadlock
3. Otherwise, MPI_ERR_TRUNCATE or memory corruption (big trouble) at receiver!
4. MPI_INT matches MPI_INT, and so forth... rarely checked/enforced, be careful
Process rank i:

double a;
MPI_Send(&a,1,MPI_DOUBLE,j,1111,MPI_COMM_WORLD);

Would likely succeed, but violates 4.

Process rank j:

double a;
MPI_Recv(a,sizeof(double),MPI_BYTE,i,111,MPI_COMM_WORLD,&status);

Very bad practice: Type information (double) lost (process i and j could be on different types of processors, little/big endian)
MPI_Send(sendbuf,...,rank,tag,comm):

**Determinate:** Message always sent to specific rank (in comm) with specific tag

MPI_Recv(recvbuf,..., rank/MPI_ANY_SOURCE,tag/MPI_ANY_TAG, comm,&status):

receives from specific rank or non-determined (ANY) rank, with specific or non-determined (ANY) tag
MPI_Send(sendbuf,…,rank,tag,comm):

**Determinate**: Message always sent to specific rank (in comm) with specific tag

MPI_Recv(recvbuf,…, rank/MPI_ANY_SOURCE,tag/MPI_ANY_TAG, comm,&status):

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

Source of non-determinism in MPI message-passing programs
Rule: All messages sent must be received (*)

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

(* unless cancelled, but do not rely on this)
Message in transit identified by “envelope” (meta-information):

1. Communication context (identifier): Distinguishes communication on different communicators
2. Source (implicit)
3. Destination
4. Tag
5. Message type information (header, data block, error, …)

Implementation: The “envelope” is not accessible to application programmer (and not specified by MPI standard)

Performance: MPI is designed to allow small message envelope
Implementation: The „envelope“ is not accessible to application programmer (and not specified by MPI standard)

Performance: MPI is designed to allow small message envelope

Not part of envelope:
1. Type and structure of message: What is being sent is a sequence of basic datatypes (int, double, char, …); and it is expected that receiving process has space for exactly the same sequence of basic datatypes
Reasoning about point-to-point communication

Deterministic messages are non-overtaking:

Messages sent to the same destination (rank) arrive in sent order at destination

\[
\begin{align*}
\text{MPI\_Send} & \quad \text{Msg 1} & \quad \text{MPI\_Recv} & \quad \text{msg 1} \\
\text{MPI\_Send} & \quad \text{Msg 2} & \quad \text{MPI\_Recv} & \quad \text{msg 2} \\
\end{align*}
\]

No races
But no global time, MPI processes are not synchronized

```plaintext
MPI_Send  Msg 1  MPI_Recv
MPI_Send  Msg 2  MPI_Recv
```

Local time
Therefore, possibly...

At receiver: 
„unexpected message“

Possibly idle time

Definitely idle time

MPI_Send

Msg 1

MPI_Recv

MPI_Recv

MPI_Send

Msg 2
Example: Synchronization with 0-count message

```c
MPI_Send(NULL, 0, MPI_INT, dest, tag, comm);

MPI_Recv(NULL, 0, MPI_INT, source, tag, comm, &status);
```

0-count message **must be sent!** Why?

Can be used for process synchronization: Receiving process (with rank dest) cannot proceed before sending process (rank source) has reached send operation. For pairwise synchronization, send 0-message back

Alternative: `MPI_Ssend` Synchronous send, not this lecture
MPI_Recv(recvbuf,...,MPI_ANY_SOURCE,MPI_ANY_TAG,comm,&status);

Wildcards:
• Receive some (ANY) message from somewhere (ANY, but within comm)
• Need to check status to find out source and tag

Message ordering (non-overtaking) is still guaranteed
Either message may be received first. Problems if messages have different count and/or type.
MPI_Probe(source, tag, comm, &status);

Return when a message with given source (or MPI_ANY_SOURCE) and tag (or MPI_ANY_TAG) in comm has arrived and is ready for reception; the count for message (and error code etc.) in status

After probe:
Receive message with MPI_Recv(buffer, count,...,comm)

Advanced note: Can cause problems in multi-threaded MPI applications (fix in MPI 3.0)
Wait until message has arrived from somewhere

```c
#include <assert.h>

MPI_Status status;
int tag = 777;

MPI_Probe(MPI_ANY_SOURCE, tag, comm, &status);
// blocking call
// returns when message with tag 777 has arrived

int source = status.MPI_SOURCE;
assert(status.MPI_TAG == tag);
int count;
MPI_Get_count(status, MPI_INT, &count); // get count
recvbuf = (int*)malloc(count*sizeof(int)); // allocate
MPI_Recv(recvbuf, count, MPI_INT, source, tag, comm,
         MPI_STATUS_IGNORE);
```
#include <assert.h>

MPI_Status status;
int tag = 777;

MPI_Probe(MPI_ANY_SOURCE, tag, comm, &status);
// blocking call
// returns when message with tag 777 has arrived

int source = status.MPI_SOURCE;
assert(status.MPI_TAG==tag);
int count;
MPI_Get_count(status, MPI_INT, &count); // get count
recvbuf = (int*)malloc(count*sizeof(int)); // allocate
MPI_Recv(recvbuf, count, MPI_INT, source, tag, comm, MPI_STATUS_IGNORE);

No type-field in status, no MPI_Get_type(). Why?
Properties of point-to-point communication

• Point-to-point communication is **deadlock free**: A message sent from a source process to a destination process is eventually received provided the destination posts a matching receive operation.

• Point-to-point communication is **non-overtaking**: Messages from a single source arrive in sent order at destination (different reception order can be enforced by tags; non-determinism with any-source).

• Point-to-point communication is **reliable**: Messages are not lost, content can be trusted.
Example: 2d-stencil

For all \(0 \leq i < m, 0 \leq j < n\), update

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

Special conditions on borders, \(i=0, i=m-1, j=0, j=n-1\)

Idea:
Each process takes care of \(d_1 \times d_2\) submatrix; communication with neighbors of border rows and columns
May deadlock: MPI_Send has non-local completion semantics, each send may block until receive operation starts. This may never happen.

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

- Non-local completion, may block for receive operation
- Exact conditions of local-completion are MPI implementation dependent

Freedom for MPI implementers:

- Short messages: Buffered at sender or receiver, processed later (MPI_Send can return immediately)
- Medium sized messages: May be buffered locally, and sent when receive has been posted (MPI_Send can return can return)
- Long messages: Participation of receiving process needed (MPI_Send cannot return before MPI_Recv active)
Template MPI_Send implementation, short messages

MPI_Send(buffer, ..., j, ...);
MPI_Recv(buffer, ..., j, ...);

MPI_Send(buffer, ..., i, ...);
MPI_Recv(buffer, ..., i, ...);

Succeeds if internal buffer is large enough. MPI does not require internal buffering
Template MPI_Send implementation, short messages

MPI_Send(buffer, ..., j, ...);
MPI_Send(buffer, ..., i, ...);

MPI_Recv(buffer, ..., j, ...);
MPI_Recv(buffer, ..., i, ...);

Drawback: Extra copy - costly for large buffers

MPI design principle: Library should not allocate unbounded buffers
Template MPI_Send implementation, long messages

MPI_Send(buffer, ..., j, ...);

Send complete with last data

MPI_Recv(buffer, ..., i, ...);

Iterate/pipeline

Request+envelope

Ack+address

data
Example: Loop with some dependencies (1d-stencil)

Process \( j \), \( 0 \leq j \lt p \)

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

Arrays \( a \) and \( b \) distributed in blocks over MPI processes

\( n[j] \approx j \times (n/size) \)

Arrays \( a \):

- Processor \( j \)

- \( n/size \)
float *a = malloc((n/p+2)*sizeof(float));
a += 1; // offset, such that -1 and n/p can be addressed

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

DEADLOCK! All processes attempt to receive, no progress
float *a = malloc((n/p+2)*sizeof(float));
a += 1; // offset, such that -1 and n/p can be addressed

if (rank>0) {
    MPI_Send(&a[0],1,MPI_FLOAT,rank-1,999,comm);
    MPI_Recv(&a[-1],1,MPI_FLOAT,rank-1,999,comm,&status);
}
if (rank<size-1) {
    MPI_Send(&a[n/p-1],1,MPI_FLOAT,rank+1,999,comm;
    MPI_Recv(&a[n/p],1,MPI_FLOAT,rank+1,999,comm,&status);
}
for (i=0; i<n/p; i++) {
    b[i] = a[i-1]+a[i]+a[i+1];
}
All processes trying to send, looks like **deadlock**

**In MPI**: Behavior depends on data size, so this style is **unsafe**
DEADLOCK:

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

All deadlock forms are possible with MPI programs

 Particularly problematic: Some deadlocks are context and MPI library implementation dependent; termed unsafe

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

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

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

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

MPI_Recv(up);
MPI_Recv(down);
MPI_Recv(left);
MPI_Recv(right);
Safe(r) programming

Process 0
MPI_Send
MPI_Recv

Process 1
MPI_Send
MPI_Recv

Unsafe, make safe by scheduling

Process 0 (even)
MPI_Send
MPI_Recv

Process 1 (odd)
MPI_Recv
MPI_Send
Even-odd scheduling for avoiding deadlocks

Idea:
Even numbered processes send first (then receive), odd numbered processors receive first (then send)

Problem: What if the number of processes is odd?
Problem: What if the number of processes is odd?

Example: Hillis-Steele prefix-sums algorithm

Round k:
Each process $i$ receives partial sum from process $i-2^k$, and sends partial result to process $i+2^k$

Is process $i$ odd or even in round $k$?
Next attempt to parallelize data parallel loop

```c
float *a = malloc((n/p+2)*sizeof(float));
a += 1;
if (rank>0) {
    MPI_Send(&a[0],1,MPI_FLOAT,rank-1,999,comm);
    MPI_Recv(&a[-1],1,MPI_FLOAT,rank-1,999,comm,&status);
}
if (rank<size-1) {
    MPI_Recv(&a[n/p],1,MPI_FLOAT,rank+1,999,comm);
    MPI_Send(&a[n/p-1],1,MPI_FLOAT,rank+1,999,comm,
             &status);
}
for (i=0; i<n/p; i++) {
    b[i] = a[i-1]+a[i]+a[i+1];
}
```
Unfortunate even-odd scheduling leads to serialization. Last process size-1 receives after p steps!
Safe(r) programming

Unsafe, make safe by combined send-receive

Process 0
- MPI_Send
- MPI_Recv

Process 1
- MPI_Send
- MPI_Recv

Process 0
- MPI_Sendrecv

Process 1
- MPI_Sendrecv
MPI_Sendrecv(void *sendbuf,
    int sendcount, MPI_Datatype sendtype,
    int dest, int sendtag,
    void *recvbuf,
    int recvcount, MPI_Datatype recvtype,
    int source, int recvtag,
    MPI_Comm comm, MPI_Status *status);

Combined send-receive operation, in one operation send to some process and receive from some (possibly different) process

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

**Performance advantage:**
Can possibly better utilize (fully) bidirectional communication network
Exercise: Implement and compare to other solutions

\[ \text{MPI\_Sendrecv(...,rank+1,...,rank-1,...);} \]
Exercise: Implement and compare to other solutions
Example: Hillis-Steele prefix-sums algorithm (solution)

Round k:
Each process $i$ receives partial sum from process $i-2^k$, and sends partial result to process $i+2^k$
Safe programming: Non-blocking communication

```
MPI_Request req[8];
MPI_Status stats[8];

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

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

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

Safe: I(mmediate), non-blocking operations with local completion semantics
Point-to-point semantics: immediate operations

MPI_Isend(sendbuf, ..., rank, tag, comm, request):
Locally initiates send to designated process rank
• Completion is **local**: Returns immediately, send has been initiated
• Operation is **non-blocking**: Buffers cannot be reused before completion has been checked/enforced
• Sent messages to same rank with same tag are **non-overtaking**

MPI_Irecv(recvbuf, ..., rank, tag, comm, request):
Initiates receive operation from specific rank, or ANY
• Operation is **non-blocking**: Call returns immediately, message received after completion
MPI_Status status;
MPI_Request request;
MPI_Isend(sendbuf,...,comm,&request);

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

"progress" information in `request` object:

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

If flag==1 operation has completed, information in status

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

Wait: Returns when operation has completed, information in status
MPI_Isend(sendbuf,...,comm,&request);
MPI_Wait(&request,&status);

equivalent to MPI_Send(sendbuf,...,comm);

Recall:
Semantics of MPI_Isend+MPI_Wait=MPI_Send is non-local; sendbuf can be reused, receiver may or may not have started

Note:
For non-blocking send operations, status is undefined, except for MPI_ERROR field
MPI_Sendrecv(sendbuf, sendcount, sendtype, dest, sendtag,
recvbuf, recvcount, recvtype, source, recvtag,
comm, &status);

equivalent to

MPI_Request request;
MPI_Irecv(recvbuf, recvcount, recvtype, source, recvtag,
comm, &request);
MPI_Send(sendbuf, sendcount, sendtype, dest, sendtag,
comm);
MPI_Wait(&request, status);
Test and completion calls

Completion checked/enforced for all non-blocking operations by

- `MPI_Wait`
- `MPI_Test`
- `MPI_Waitall(number, array_of_requests, array_of_statuses)`
- `MPI_Testall`
- `MPI_Waitany`
- `MPI_Testany`
- `MPI_Waitsome`
- `MPI_Testsome`

Details, see MPI Standard
<table>
<thead>
<tr>
<th>Mode</th>
<th>Remark</th>
<th>Semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>MPI_Send</strong></td>
<td><em>Standard</em> Returns when sendbuf can be reused</td>
<td>Non-local (potentially)</td>
</tr>
<tr>
<td><strong>MPI_Ssend</strong></td>
<td><em>Synchronous</em> Returns when sendbuf can be reused <strong>AND</strong> receiver has started reception</td>
<td>Strictly non-local</td>
</tr>
<tr>
<td><strong>MPI_Bsend</strong></td>
<td><em>Buffered</em>, returns immediately, data may be copied into intermediate buffer</td>
<td>local</td>
</tr>
<tr>
<td><strong>MPI_Rsend</strong></td>
<td><em>Ready, standard</em> Precondition: matching receive <strong>MUST</strong> have been posted</td>
<td>Non-local</td>
</tr>
</tbody>
</table>

**Summary: Send modes, send semantics**
Only one receive mode (blocking and nonblocking)

\texttt{MPI\_Recv}/\texttt{MPI\_Irecv}

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

Semantic advantages, easier to prevent deadlocks

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

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

MPI_Waitall(8,req,stats);
```
Non-blocking operations

**Performance advantages:** It may be possible to overlap communication with computation (e.g., if other process is delayed)

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

**Note:** Actual overlap is implementation AND system dependent

**Performance note:** Waiting too long with MPI_Wait call can slow down application (progress)
Sources of non-determinism (2)

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

Note: MPI_ANY_TAG alone not a source of non-determinism
Sources of non-determinism (2)

MPI_Send(tag1) → Msg 1 → MPI_Irecv(tag2,&req2)

MPI_Send(tag2) → Msg 2 → MPI_Irecv(tag1,&req1)

MPI_Wait(req2)

Enforce specific order
Sources of non-determinism (2)

\[
\begin{align*}
\text{MPI}_\text{Send}(\text{tag1}) & \quad \text{Msg 1} \quad \text{MPI}_\text{Irecv}(\text{MPI}_{-}\text{ANY}_{-}\text{TAG}) \\
\text{MPI}_\text{Send}(\text{tag2}) & \quad \text{Msg 2} \quad \text{MPI}_\text{Irecv}(\text{tag1},&\text{req1}) \\
\text{MPI}_\text{Wait}(\text{req1})
\end{align*}
\]

DEADLOCK!

tag1 message has matched MPI_{-} ANY_{-} TAG
MPI_Iprobe(source, tag, comm, &flag, &status);

Non-blocking probe, flag==1 means message with source and tag ready for reception in comm
Point-to-point communication performance rules

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

- **Latency!**

  **Rule-of-thumb**: Avoid many small messages, group into fewer, larger messages (a may be large)

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

  **Semantics**: `MPI_Send` may (for large messages) depend on activity of receiving process
Point-to-point communication performance rules

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

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

```
MPI_(I)Send(buffer, ..., j, ...);
MPI_(I)Recv(buffer, ..., i, ...);
```

“Progress engine”: MPI calls or separate thread

Iterate, pipeline
On progress of communication

MPI_Isend

Large msg

Message transmission, conceptual

MPI_Wait

MPI_Recv

Local time
MPI_Isend

Header

Ack to send

Part 1

Ack to send

MPI_Wait

Part n

Message transmission, for real

MPI_Recv

Local time
MPI_Isend

Protocol progress:
Possibility 1: Hardware

MPI_Wait

MPI_Isend

Header

Ack to send

Part 1

Ack to send

Part n

MPI_Recv

Local time
Protocol progress:
Possibility 2: Separate thread

$\text{MPI\textunderscore Isend}$

- Header
- Ack to send
- Part 1
- Ack to send
- Part n

$\text{MPI\textunderscore Recv}$

Local time

$\text{MPI\textunderscore Wait}$
MPI_Isend

Protocol progress:
Possibility 3: Each MPI call makes progress

MPI_Wait

Header
Ack to send
Part 1
Ack to send
Part n

Local time

MPI_Recv
Application enforced progress: Difficult to tune, may be non-portable

Possibility 3: Each MPI call makes progress

MPI_Isend

MPI_Test

Protocol progress:

MPI_Test

MPI_Wait

MPI_Recv

Local time

MPI_Test

Part 1

Ack to send

Part n

MPI_Test

Header

Ack to send

MPI_Test

MPI_Recv
Progress

MPI libraries often use mixed strategies:
1. Hardware, whenever possible ("offload to NIC")
2. MPI calls to make progress
3. Sometimes thread support ("progress thread")

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

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

Principle: MPI standard is intentionally loose on progress to allow different implementations
Other point-to-point send operations

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

**MPI_Rsend**: Only allowed when matching receive call has already been posted; can save some ack’s

**MPI_Bsend**: Data always copied to intermediate buffer; buffer supplied by user, in user space. **Local completion semantics**
Other point-to-point communication features

- `MPI_PROC_NULL` - "empty" process to send to and receive from
- `(MPI_Ssend, MPI_Bsend)`
- Persistent requests
- `MPI_Cancel` - dangerous!
- `MPI_Sendrecv_replace`
Non-communication feature

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

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

```c
stime = MPI_Wtime();
MPI_Send(sendbuf,...,comm);
etime = MPI_Wtime();
// etime-stime is elapsed local time
```

`MPI_WTIME_IS_GLOBAL`: Boolean attribute to `MPI_COMM_WORLD`, time is global (rare)
double time = MPI_Wtime();

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

MPI_BARRIER(comm); // approx. temporal synchronization
stime = MPI_Wtime();
MPI_Send(sendbuf,...,comm);
etime = MPI_Wtime();
// etime-stime is elapsed local time

MPI_WTIME_IS_GLOBAL: Boolean attribute to
MPI_COMM_WORLD, time is global (rare)
MPI Datatypes, data layouts

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

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

“Get message stored in array sendbuf of 500 consecutive integers on the road to dest in comm”

C int

Described by datatype MPI_INT
MPI_Send(sendbuf,count,datatype,dest,tag,comm);

basetype *sendbuf;
sendbuf = malloc(count*extent*sizeof(basetype));

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

“Get message stored in array sendbuf of count consecutive datatype’s on the road to dest in comm”

sendbuf:

C sometype

could be non-consec. layout

Described by datatype
MPI datatypes

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

Handle: MPI_Datatype

New (called user-defined or derived) datatypes can be constructed from previously described types as

- Contiguous: contiguous blocks of element type
- Vectors: regularly strided blocks of element type
- Indexed: irregularly strided blocks of same type
- Structs: irregularly strided blocks of possibly different types
Basetype: basic or user-defined

contiguous

vector

indexed

struct

Sequence numbers: Order in which basetype are sent and received
Derived/user-defined datatypes, quick tour

- **Basetype**: basic or derived, unit of communication

Constructors to form new units of communication out of existing ones (and imposes a sequence numbering)

\[ \text{MPI\_Bcast(buffer, count, root, comm)}: \]

New, derived (user-defined) datatypes can be used in all MPI communication operations

Units of communication have a **size** (number of bytes occupied), **extent** (number of bytes spanned), may have internal structure
Basetype: basic or derived

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

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

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

```c
MPI_Type_get_extent(datatype, &lb, &extent);
MPI_Type_size(datatype, &bytes);
```

Care: extent and lb are MPI_Aint, bytes is int
Basetype: basic or derived

```c
MPI_Type_contiguous(count, basetype, &newtype);
```

extent (for repetition)

```c
MPI_Type_commit(&newtype);
```

Cost of description: 1 word

newtype is now a new unit of communication (consisting of count successive basetypes)
Type signature:
List of basic datatypes (order is determined by constructor)

Type map:
List of basic datatypes with their displacement

**MPI_Send**(datatype):
A sequence of values of type corresponding to type signature is sent in the order imposed by the type

**MPI_Recv**(datatype):
A sequence of values corresponding to (prefix of) type signature is received and stored as described by type map

Types match if send signature is a prefix of receive signature
MPI_Type_vector(count, blocksize, stride, basetype, &newtype);

Basetype: basic or derived

extent

stride (in units of basetype)

Cost of description: 3 words
Example

```c
int m[d1][d2];
```

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

Diagram:

- Nodes: u, i, d, l, r
- Edges indicate communication among processes.
int m[d1][d2];

Rows:

MPI_Isend(m[0], d2, MPI_INT, up, ...);
MPI_Isend(m[d1-1], d2, MPI_INT, down, ...);
int m[d1][d2];

MPI_Datatype col;
MPI_Type_vector(d1,1,d2,MPI_INT,col);
MPI_Type_commit(&col);

MPI_Isend(&m[0][0],1,col,left,…);
MPI_Isend(&m[0][d2-1],1,col,down,…);

row 0, d2 columns  row 1, d2 columns

MPI_Type_free(&col);
Advice: use it! Should be at least as good as

1. Copying the row elements into intermediate, consecutive int buffer
2. Sending intermediate buffer

```c
int m[d1][d2];

MPI_Datatype col;
MPI_Type_vector(d1,1,d2,MPI_INT,col);
MPI_Type_commit(&col);

MPI_Isend(&m[0][0],1,col,left,…);
MPI_Isend(&m[0][d2-1],1,col,down,…);

MPI_Type_free(&col);
```
Type matching

int m[d1][d2];

Observation: Transpose operation performed by communication and datatype

int vec[d];

Columns:

MPI_Datatype col;
MPI_Type_vector(d1,1,d2,MPI_INT,col);
MPI_Type_commit(&col);

MPI_Isend(&m[0][0],1,col,left,...);
Basetype: basic or derived

MPI_Type_create_indexed_block(count, block, displs, basetype, &newtype);

Cost of description: 1 word, 1 array (displs)
Basetype: basic or derived

\[ \text{MPI\_Type\_indexed(count,blocks,displs,basetype,}\] 
\[\text{&newtype);} \]

Cost of description: 1 word, 2 arrays (blocks, displs)
Basetypes: basic or derived

```c
MPI_Type_create_struct(count, blocks, displs, etypes, &newtype);
```

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

Displacements in bytes

Cost of description: 1 word, 3 arrays
struct somestruct {
    int a[4];
    double b[4];
    int somepaddingthatweignore;
    double c;
    char d[2];
    long e[3];
} data[100];

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

- `MPI_Type_create_hvector` - vector with byte stride
- `MPI_Type_create_hindexed` - indexed with byte displacements
- `MPI_Type_create_hindexed_block` - indexed with same sized blocks and byte displacements

- `MPI_Type_create_subarray` - arrays, complex
- `MPI_Type_create_darray` - distributed arrays, not easy...
Performance of MPI derived datatypes

Applications may have non-consecutive data (example: d-dimensional stencil: vector)

Communication alternatives:
1. Communicate data elements one after the other. BAD: many communication operations (of small messages)
2. Copy data into larger consecutive buffer. Possibly tedious, but user knows where the data are. Always extra copy
3. Use MPI derived datatypes (example: vector) to describe where the data are, leave packing and efficient pipelining to MPI library. Setting up the datatype a bit tedious, overhead in type creation and MPI_Type_commit. MPI library can possibly avoid extra copy: zero-copy

Many modern MPI libraries have reasonable datatype engine
### C integer datatypes

<table>
<thead>
<tr>
<th>Basic MPI_Datatype</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_CHAR</td>
<td>char</td>
</tr>
<tr>
<td>MPI_SHORT</td>
<td>(signed) short (int)</td>
</tr>
<tr>
<td>MPI_INT</td>
<td>int</td>
</tr>
<tr>
<td>MPI_LONG</td>
<td>(signed) long (int)</td>
</tr>
<tr>
<td>MPI_LONG_LONG</td>
<td>signed long long int</td>
</tr>
<tr>
<td>MPI_SIGNED_CHAR</td>
<td>signed char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_CHAR</td>
<td>unsigned char</td>
</tr>
<tr>
<td>MPI_UNSIGNED_SHORT</td>
<td>unsigned short int</td>
</tr>
<tr>
<td>MPI_UNSIGNED</td>
<td>unsigned int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG</td>
<td>unsigned long int</td>
</tr>
<tr>
<td>MPI_UNSIGNED_LONG_LONG</td>
<td>unsigned long long int</td>
</tr>
<tr>
<td>MPI_C_BOOL</td>
<td>_Bool</td>
</tr>
<tr>
<td>MPI_WCHAR</td>
<td>wchar_t</td>
</tr>
</tbody>
</table>
### C integer datatypes (con't)

<table>
<thead>
<tr>
<th>Basic <code>MPI_Datatype</code></th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_INT8_T</code></td>
<td><code>int8__t</code></td>
</tr>
<tr>
<td><code>MPI_INT16_T</code></td>
<td><code>int16_t</code></td>
</tr>
<tr>
<td><code>MPI_INT32_T</code></td>
<td><code>int32_t</code></td>
</tr>
<tr>
<td><code>MPI_INT64_T</code></td>
<td><code>int64_t</code></td>
</tr>
<tr>
<td><code>MPI_UINT8_T</code></td>
<td><code>uint8__t</code></td>
</tr>
<tr>
<td><code>MPI_UINT16_T</code></td>
<td><code>uint16_t</code></td>
</tr>
<tr>
<td><code>MPI_UINT32_T</code></td>
<td><code>uint32_t</code></td>
</tr>
<tr>
<td><code>MPI_UINT64_T</code></td>
<td><code>uint64_t</code></td>
</tr>
</tbody>
</table>
### C floating point datatypes

<table>
<thead>
<tr>
<th>Basic MPI_Datatype</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>MPI_FLOAT</code></td>
<td>float</td>
</tr>
<tr>
<td><code>MPI_DOUBLE</code></td>
<td>double</td>
</tr>
<tr>
<td><code>MPI_LONG_DOUBLE</code></td>
<td>long double</td>
</tr>
<tr>
<td><code>MPI_C_COMPLEX</code></td>
<td>float _Complex</td>
</tr>
<tr>
<td><code>MPI_C_DOUBLE_COMPLEX</code></td>
<td>double _Complex</td>
</tr>
<tr>
<td><code>MPI_LONG_DOUBLE_COMPLEX</code></td>
<td>long double _Complex</td>
</tr>
</tbody>
</table>
## FORTRAN datatypes

<table>
<thead>
<tr>
<th>Basic MPI_Datatype</th>
<th>FORTRAN type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_INTEGER</td>
<td>INTEGER</td>
</tr>
<tr>
<td>MPI_REAL</td>
<td>REAL</td>
</tr>
<tr>
<td>MPI_DOUBLE_PRECISION</td>
<td>DOUBLE PRECISION</td>
</tr>
<tr>
<td>MPI_COMPLEX</td>
<td>COMPLEX</td>
</tr>
<tr>
<td>MPI_LOGICAL</td>
<td>LOGICAL</td>
</tr>
<tr>
<td>MPI_CHARACTER</td>
<td>CHARACTER(1)</td>
</tr>
</tbody>
</table>
## Special datatypes

<table>
<thead>
<tr>
<th>Basic MPI_Datatype</th>
<th>C type</th>
<th>Fortran type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_BYTE</td>
<td>Uninterpreted bytes</td>
<td></td>
</tr>
<tr>
<td>MPI_PACKED</td>
<td>Special, packed data (*)</td>
<td></td>
</tr>
</tbody>
</table>

(*) generated by MPI_Pack/MPI_Unpack only

<table>
<thead>
<tr>
<th>Basic MPI_Datatype</th>
<th>C type</th>
<th>Fortran type</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_AINT</td>
<td>MPI_Aint</td>
<td>INTEGER (KIND=MPI_ADDRESS_KIND)</td>
</tr>
<tr>
<td>MPI_OFFSET</td>
<td>MPI_Offset</td>
<td>INTEGER (KIND=MPI_OFFSET_KIND)</td>
</tr>
</tbody>
</table>

**MPI_Aint**: address sized int
One-sided communication

Communication between two processes, but only one process is explicitly involved with communication calls.

- One-sided: MPI_Put
  
  MPI_Get
One-sided communication: By example

Safe neighbor exchange with one-sided (put) communication

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

Issues:

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

One-sided communication decouples communication and synchronization
**MPI one-sided communication terminology**

**Origin** process alone responsible for initiating communication, provides all arguments

- `MPI_Put(obuf, ocount, otype, ..., win)`
- `MPI_Get(obuf, ocount, otype, ..., win)`
- `MPI_Accumulate(obuf, ocount, otype, ..., op, win);`

Communication calls are **non-blocking**, local completion semantics: no guarantee that data have arrived before synchronization operation

**Origin** puts/get data from standard MPI buffer `(buf, count, type)`
Data on target exposed in window structure, addressed with relative displacement

Origin process alone responsible for initiating communication, provides all arguments

- MPI_Put(..., target, tdisp, tcount, ttype, win);
- MPI_Get(..., target, tdisp, tcount, ttype, win);
- MPI_Accumulate(..., target, tdisp, tcount, ttype, op, win);

Target process (semantically) not involved in communication
**Communication window:**
Distributed, global object containing memory for each process that can be accessed in one-sided communication operations

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

Collective operation, all processes in `comm` provide a base address (size in Bytes may be 0), displacement unit `info` (special MPI (key,value) object) can influence window properties (use `MPI_INFO_NULL`)

**MPI_Alloc_mem**: Special MPI memory allocator, sometimes beneficial (performance) for windows
MPI_Put (obuf, ..., target, targetdisp, ..., win);

Data from obuf into target base + targetdisp unit * targetdisp

**NB:** dispunit at target

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

**Note:** Same rules as for point-to-point communication
Concurrent gets/puts must access disjoint target addresses. Very strict rules, violation is undefined (and usually not checked)

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

**Origin** must have access to target: **access epoch**

**Target** must expose memory: **exposure epoch**

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

Close epoch: synchronization, communication operations become visible
Synchronization, epochs

Active synchronization, both origin and target processes involved

```c
MPI_Win_fence(assert, win);
```

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

Assertion can control opening/closure behavior, use for tuning
Synchronization, epochs

Active synchronization, both origin and target processes involved

\[
\text{MPI\_Win\_start(..., group);} \\
\text{MPI\_Win\_complete();}
\]

Opens/closes access epoch, targets as process group (MPI\_Group)

\[
\text{MPI\_Win\_post(..., group);} \\
\text{MPI\_Win\_wait();}
\]

Opens/closes exposure epoch, origins as process group (MPI\_Group)

„generalized“ pairwise synchronization...
Synchronization, epochs

Passive synchronization, only origin process involved

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

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

**Note 1:** Not a lock (critical section), difficult to use for mutual exclusion (read-modify-write), weak mechanism

**Note 2:** Data at target may not be visible before target performs `MPI_lock` on itself (and other weirdness)
One-sided communication - by example

Safe neighbor exchange with one-sided (put) communication

```c
// prepare neighbor data
MPI_Win_fence(win);
MPI_Put(up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
MPI_Win_fence(win);
// data from neighbors ready
```
Safe neighbor exchange with one-sided (put) communication

// prepare neighbor data
MPI_Win_start ([l,u,r,d], win);
MPI_Win_post([l,u,r,d], win);
MPI_Put(up);
MPI_Put(down);
MPI_Put(left);
MPI_Put(right);
MPI_Win_wait(win);
MPI_Win_complete(win);

// data from neighbors ready

Note: [l,u,r,d] is provided as process group (MPI_Group)
MPI_Win_free(win);

free after use... (like other MPI objects); freeing the memory per process is **NOT** handled by MPI

```
base = (void*)malloc(size);
// or: MPI_Alloc_mem(size,MPI_INFO_NULL,&base);
MPI_Win_create(base,size,dispunit,info,comm,&win);

... // one sided communication epochs

MPI_Win_free(&win);
free(base);
// or: MPI_free_mem(base);
```
A note on progress

MPI_Put

Large msg

Progress on both sides by
1. Hardware
2. Separate thread
3. Other MPI calls

MPI_Win_fence

Local time
Example: Binary search with one-sided communication

A[i]:

\[ l = -1; u = n; \]
\[ \text{do } \{ \]
\[ \quad m = (l+u)/2; \]
\[ \quad \text{if } (x < A[m]) u = m; \text{ else } l = m; \]
\[ \} \text{ while } (l+1 < u); \]
\[ i = l; \]

Binary search for key \( x \) in array \( A[0,...,n-1] \) in at most \( \lceil \log n \rceil \) steps. Upon termination \( A[i] \leq x < A[i+1] \)

Task: implement sequential binary search in MPI (distributed memory, message passing), many processes search simultaneously

Side note: Binary search cannot be sped up (much) by parallel processing (Snir lower bound)
Array $A[i]$ distributed (roughly) evenly over $p$ MPI processes, process $r, 0 \leq r < p$, has block of (roughly) $n/p$ elements.
A[i]:  

\[
\begin{align*}
\text{Process r:} & \\
l = -1; & u = n; \\
do & \{ \\
& m = (l+u)/2; \\
& mA = \text{<get A[m]>} \\
& \text{if } (x<mA) \ u = m; \text{ else } l = m; \\
} & \text{while } (l+1<u); \\
i = l; 
\end{align*}
\]

Where is the middle element, and how to get it?

Process r can compute where A[m] is, but this process cannot know that r want to read an element: Fits one-sided model
Process $r$ can compute where $A[m]$ is, but this process cannot know that $r$ want to read an element: Fits one-sided model

All processes make their block of $A$ accessible to other processes in communication window

```c
A = (int*)malloc(n/p*sizeof(int));
... // init block (see later)

MPI_Win_create(A, (n/p)*sizeof(int), sizeof(int), MPI_INFO_NULL, comm, &win);
```
Process r:

```c
A[i]:

l = -1; u = n;

do {
    m = (l+u)/2;
    t = m/p; // target process
    tix = m%p; // index at target
    MPI_Get(&mA,1,MPI_INT,
            t,tix,1,MPI_INT,win);

    if (x<mA) u = m; else l = m;
} while (l+1<u);

i = l;
```

Problem: Need to make sure that data have been received.
Process r:

```c
l = -1; u = n;
while (l+1 < u) {
    m = (l+u)/2;
    MPI_Win_fence(0, win);
    t = m/p; // target process
    tix = m%p; // index at target
    MPI_Get(&mA, 1, MPI_INT,
            t, tix, 1, MPI_INT, win);
    MPI_Win_fence(0, win);
    if (x < mA) u = m; else l = m;
} i = l;
```

Solution 1:
Open/close access and exposure epochs with collective fence

Problem:
Fence is collective, all processes in win must perform call

Must ensure that all processes do same number of iterations (calls to fence)
A[i]:

\[ t = \frac{m}{p} \]

\[ t_i = m \% p \]

\[ \approx \frac{n}{p} \]

Process \#r:

\[ l = -1; u = n; \]

\[ \text{do} \{ \]

\[ m = \frac{l+u}{2}; \]

\[ \text{MPI}_\text{Win}_\text{fence}(0, \text{win}); \]

\[ \text{MPI}_\text{Win}_\text{fence}(0, \text{win}); \]

\[ l = m; \]

\[ \text{while} \ (l+1<u); \]

\[ // \text{All processes must perform} \]

\[ // \text{ceil}(\log n) \text{ iterations} \]

\[ i = l; \]

Solution 1:

Open/close access and exposure epochs with collective fence

Use upper bound \( \text{ceil}(\log n) \) to make sure all processes do same number of iterations

Still not correct
A[i]:

\[ t = \frac{l + u}{2} \]

\[ \text{MPI\_Win\_fence}(\text{MPI\_NO\_PRECEDE}, \text{win}); \]

\[ t = \frac{m}{p}; \quad \text{target process} \]

\[ \text{tix} = m \mod p; \quad \text{index at target} \]

\[ \text{MPI\_Get}(&mA, 1, \text{MPI\_INT}, \]
\[ t, \text{tix}, 1, \text{MPI\_INT}, \text{win}); \]

\[ \text{MPI\_Win\_fence}(\text{MPI\_NO\_SUCCEED}, \text{win}); \]

\[ \text{if} \ (x < mA) \ u = m; \text{ else } l = m; \]

\[ } \quad \text{while} \ (l + 1 < u); \]

\[ i = l; \]

\[ \approx \frac{n}{p} \]

**Process r:**

\[ l = -1; \ u = n; \]

\[ \text{do} \{ \]

\[ m = \frac{l + u}{2}; \]

\[ \text{MPI\_Win\_fence}(\text{MPI\_NO\_PRECEDE}, \text{win}); \]

\[ t = \frac{m}{p}; \quad \text{target process} \]

\[ \text{tix} = m \mod p; \quad \text{index at target} \]

\[ \text{MPI\_Get}(&mA, 1, \text{MPI\_INT}, \]
\[ t, \text{tix}, 1, \text{MPI\_INT}, \text{win}); \]

\[ \text{MPI\_Win\_fence}(\text{MPI\_NO\_SUCCEED}, \text{win}); \]

\[ \text{if} \ (x < mA) \ u = m; \text{ else } l = m; \]

\[ } \quad \text{while} \ (l + 1 < u); \]

\[ i = l; \]

**Tuning:**

Use assertions.

- **MPI\_NO\_PRECEDE** asserts that fence call does not complete any local Get/Put's
- **MPI\_NO\_SUCCEED** asserts that fence does not start any local Put/Get's
A[i]:

\[
t = \frac{m}{p} \\
tix = m \mod p \\
\approx \frac{n}{p}
\]

Process r:

\[
l = -1; \ u = n; \\
do \ {m = (l+u)/2; \\
\text{MPI}_\text{Win\_fence}(\text{MPI\_NO\_PRECED}, \text{win});} \\
t = \frac{m}{p}; // \text{target process} \\
tix = m \mod p; // \text{index at target} \\
\text{MPI\_Get}(&mA, 1, \text{MPI\_INT}, \\
\quad t, tix, 1, \text{MPI\_INT}, \text{win}); \\
\text{MPI\_Win\_fence}(\text{MPI\_NO\_SUCCEED}, \text{win}); \\
\text{if} \ (x < mA) \ u = m; \ \text{else} \ l = m; \\
\} \ \text{while} \ (l+1 < u); \\
i = l;
\]

Tuning:
Use assertions, see MPI Standard for more possibilities
A[i]:

\[ \text{tix } = \frac{m}{p} \]

\[ t = \frac{m}{p} \]

\[ \approx \frac{n}{p} \]

**Process r:**

```c
l = -1; u = n;
do 
    { 
        m = (l+u)/2;
        t = m/p;  // target process
        MPI_Win_lock(MPI_LOCK_SHARED, t, 0, win);
        tix = m%p; // index at target
        MPI_Get(&mA, 1, MPI_INT,
                t, tix, 1, MPI_INT, win);
        MPI_Win_unlock(t, win);
        if (x<mA) u = m; else l = m;
    } while (l+1<u);
i = l;
```

**Solution 2:**
Open/close access and exposure epochs with passive lock

**Drawback:** probably slow(er)?

**But**, recall that \( n >> \log n \)
Distributed merging

Given

A: < < < <

B: < < < <

Ordered input arrays A and B of n and m elements distributed in (roughly) even-sized blocks over the p MPI processes. In particular, elements for process \( r \) are smaller-equal than elements for process \( r+1 \)

Task: Compute blocks of ordered output array C, such that \( \text{merge}(A, n, B, m, C) \). Each process stores a block of C of size (roughly) \( (n+m)/p \)
1. Use co-ranking algorithm (MPI one-sided implementation) to determine partial blocks for the C block of process r. All p MPI processes co-rank in parallel.
2. Copy partial blocks into intermediate $C'$ array. One-sided communication; note that a partial block can span several processes.
3. Local, sequential merge of $C'$ array into resulting $C$ block

**Note:** Binary search parallel merge algorithm can also be used, but more tedious.
Convenience functionality: d-dimensional naming

For d-dimensional stencil computation:

MPI processes must be organized in a d-dimensional mesh: each process rank needs to be able to compute efficiently in $O(1)$ operations its 2d neighbor ranks.

By-hand solution: row-order numbering. Assume $d=2$, let $p=rc$ (row-column), rank $i$, $0 \leq i < p$, has coordinate $(i/c, i \% c)$, coordinate $(a, b)$, $0 \leq a < r$, $0 \leq b < c$, has rank $a*c + b$.
MPI functionality: Cartesian communicators

\[
\text{MPI\_Cart\_create}(\text{comm}, d, \text{dim}, \text{period}, \text{reorder}, &\text{newcomm});
\]

creates new communicator with helper functionality for \(d\)-dimensional Cartesian coordinate addressing. Collective operation, all processes in \text{comm} must call

\(d\): number of dimensions (1, 2, 3, ...)
\(\text{dim}\): \(d\)-element array, \(\text{dim}[i]\) is size in \(i\)'th dimension, must hold that \(\prod\text{dim}[i] \leq p\)
\(\text{period}\): \(d\)-dimensional flag-array, if \(\text{period}[i] == 1\) (true) the Cartesian grid is periodic in the \(i\)'th dimension

Let \text{reorder}=0 (false), otherwise MPI library may attempt to rerank processes to let virtual, Cartesian topology fit better with communication network topology
MPI_Comm_size(comm, &p);
r = sqrt(p); c = p/r; // or try MPI_Dims_create
dim[0] = c; dim[1] = r;
period[0] = 0; period[1] = 0;

reorder = 0;
MPI_Cart_create(comm, d, dim, period, reorder, &newcomm);
MPI_Comm_size(comm,&p);
\[ r = \sqrt{p}; \quad c = \frac{p}{r}; \] // or try MPI_Dims_create
\[ \text{dim}[0] = c; \quad \text{dim}[1] = r; \]
\[ \text{period}[0] = 1; \quad \text{period}[1] = 0; \]

Periodic in dimension 0, processes on fringe are neighbors

\[ \text{reorder} = 0; \]
\[ \text{MPI_Cart_create}(\text{comm},d,\text{dim},\text{period},\text{reorder},\&\text{newcomm}); \]
Computing coordinates of u, d, l, r (Solution 1)

```c
int icoord[2];
MPI_Cart_coords(newcomm,i,2, icoord);

ucoord[0] = icoord[0];
// if icoord[1]≠0
MPI_Cart_rank(newcomm,ucoord, &u);
```

Etc. MPI_Cart_rank requires all coordinate in dimension i to be in range, when period[i]==0
Computing coordinates of u, d, l, r (Solution 2)

\[
\begin{align*}
\text{MPI}_\text{Cart}_{-}\text{shift} & (\text{newcomm}, 0, 1, & r, & l) \\
\text{MPI}_\text{Cart}_{-}\text{shift} & (\text{newcomm}, 1, 1, & u, & d)
\end{align*}
\]

compute neighbor ranks for one hop in each dimension

r, l, u, d will be MPI\_PROC\_NULL when shifting out of range
Collective communication

Communication domain (communicator)  MPI processes

- Collective:  MPI_Bcast  MPI_Bcast  MPI_Bcast

Communication among many (all) processes in communicator, all processes in communicator are explicitly involved, and must invoke same collective communication operation
Collective operations: Motivation (I)

Distributed data structure

while (!global convergence) {
    compute locally
    check for convergence
}
Distributed data structure

while (!global convergence) {
    compute locally
    check for local convergence
    All processes:
    globalflag = AND(localflags)
}

Collective operation: All processes take part in AND-computation (reduction with associative operation)
Distributed data structure

while (!global convergence) {
    compute locally
    redistribute data
    check for local convergence

    All processes:
    globalflag = AND(localflags)
}

©Jesper Larsson Träff
Distributed data structure: Compute

```
while (!global convergence) {
    compute locally
    redistribute data
    check for local convergence

    All processes:
    globalflag = AND(localflags)
}
```
Distributed data structure: Redistribute

while (!global convergence) {
    compute locally
    redistribute data
    check for local convergence

    All processes:
    globalflag = AND(localflags)
}

Collective operation: All-to-all communication, each process may have data for each other process; all processes take part in exchange
Distributed data structure: Initial distribution

```
distribute data from “master”
while (!global convergence) {
    compute locally
    redistribute data
    check for local convergence

    All processes:
    globalflag = AND(localflags)
}
collect result, write to file
```

Collective communication operations (non-symmetric), or collective I/O
Collective operations: motivation (II)

**Task:** Each process $i$ has a vector of elements, compute the elementwise sum of all vectors, and store result vector at some root process, or at all processes

$$y = x_0 + x_1 + x_1 + \ldots + x(p-1)$$

$xi = \begin{pmatrix} x_0i \\ x_1i \\ x_2i \\ \vdots \\ x(n-1)i \end{pmatrix}$

„Root“: Designated MPI process that receives/computes final result
Method 1: root receives and computes

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

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

Problem 1: The program is unsafe.
Problem 2: Tedious, if required to work for all possible C types.
Performance: $O(p), p(\alpha+\beta n)+p\gamma n$, $\gamma$ time of "+" per element

No speedup possible: Sequential summing of $p$ vectors: $p\gamma n$
Method 2: Ring, all compute

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

Ring: Result $y$ is computed in the order

$$x(root+1)+x(root+2)+...+x(size-1)+x0+...+x(root)$$

What if $root \neq size-1$, and the operation "+" is not commutative?

Performance: $O(p)$, still no speedup
int RingReduce(void *sendbuf,
               void *recvbuf, int count,
               MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm)
{
    // <insert method 2 or 1 here>
    return MPI_SUCCESS;  // everything went fine...
}
What happens here:

If i==j+1: RingReduce receives a

Process i:
RingReduce(x1,y1,…,root,…,comm);
MPI_Recv(a,…,j,SUMTAG,comm);

Process j:
MPI_Send(a,…,i,SUMTAG,comm);
RingReduce(x1,y1,…,root,…,comm);
MPI_Reduce(a,…,j,SUMTAG,comm);

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

And here:  

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

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

Unintended use; unsafe
Method 3: Using properties of \( + \) to improve performance

Since \( + \) is associative

\[
x_0 + x_1 + x_2 + \ldots + x(p-1) = y
\]

can be computed as

\[
(x_0 + x_1) + (x_2 + x_3) + \ldots + x(p-1) = y
\]

and

\[
((x_0 + x_1) + (x_2 + x_3)) + \ldots + ((x(p-2) + x(p-1)) = y
\]
**Step 1:** in parallel

\[
(x_0 + x_1) + (x_2 + x_3) + (x_4 + x_5) + (x_6 + x_7)
\]

\[
\begin{align*}
p_0 & \rightarrow p_1 \rightarrow p_2 \rightarrow p_3 \\
p_4 & \rightarrow p_5 \rightarrow p_6 \rightarrow p_7
\end{align*}
\]

**Step 2:** in parallel

\[
((x_0 + x_1) + (x_2 + x_3)) + ((x_4 + x_5) + (x_6 + x_7))
\]

\[
\begin{align*}
p_0 & \rightarrow p_2 \\
p_4 & \rightarrow p_6
\end{align*}
\]

**Step 3:** in parallel

\[
((x_0 + x_1) + (x_2 + x_3)) + ((x_4 + x_5) + (x_6 + x_7))
\]

\[
\begin{align*}
p_0 & \rightarrow p_4
\end{align*}
\]
Binomial tree

Observation:
Root node 0 active in each communication round

Same idea as round-optimal broadcast: “divide processes in two equal-sized sets, local reduction to local root in each, send result from one set to the set in which the global root is...”
Theorem:
Sum can be computed in $\text{ceil}(\log_2 p)$ communication rounds with $p$ processes by binomial tree algorithm.

Time $(\log_2 p)(a+\beta n+\gamma n)$

Assumption:
Tree-like communication is efficiently supported by the communication network. Requirement: tree can be embedded in communication graph with no contention and no dilation.

Meets lower bound (as for broadcast), not possible to reduce in less than $\log_2 p$ rounds, even on fully connected network.
Binomial tree of $p$ nodes, $r = \log_2 p$, number of levels $r + 1$

Level 0

Level 1

Level 2

Level 3

Level 4

Number of nodes at level $k$, $0 \leq k \leq r$, is

$$\binom{r}{k} = \frac{r!}{((r-k)!k!)}$$

“binomial coefficient”
Binomial tree

0

bin(0)=000

1

bin(1)=001

2

bin(2)=010

4

bin(4)=100

5

bin(5)=101

6

bin(6)=110

3

bin(3)=011

7

bin(7)=111

bit = 0x1;
while ((rank&bit)==0x0&&... ) {
    <receive from rank|bit> // bit set
    bit <<= 1; // shift right
}
<send to rank|(~bit)> // mask out bit
Method 3: Binomial tree

```c
int TreeReduce(void *sendbuf, void *recvbuf, int count,
               MPI_Datatype type,
               MPI_Op op, int root, MPI_Comm comm)
{
    int bit = 0x1; // assert(root==0);
    int rank, size; // get with MPI_Comm_rank...
    while ((rank&bit)==0&&(rank&(~bit))<size) { // not set
        MPI_Recv(tmp,count,type,(rank|bit),REDTAG,comm,…);
        MPI_Reduce_local(res,tmp,count,type,op);
        bit <<= 1;
        void *tb = tmp; tmp = res; res = tb; // swap
    }
    if (rank!=root) {
        MPI_Send(tmp,count,type,rank&(~bit),…);
    } else … // root must have result in recvbuf
    return MPI_SUCCESS; // everything went fine…
}
```
Method 3: Binomial tree

Some details:

- Assumed that rank 0 is root (some difficulties to adapt to general case, depending on properties of \( \text{op} \))
- \( \text{tmp} \) and \( \text{res} \) are intermediate buffers, at first \( \text{sendbuf} \) argument has to be copied into \( \text{res} \); in each receive round, \( \text{tmp} \) and \( \text{res} \) swapped
- Be careful about (not to exploit) commutativity
- \( \text{MPI\_Reduce\_local} \) is an MPI operation that allows local reduction of structured MPI vectors with user defined operation
Reduction on mesh/torus networks

Phase 1:
reduce vertically
Phase 2:
Reduce horizontally

Time: $2 (\sqrt{p-1})(\alpha + \beta n)$

Note:
Both binomial tree and mesh algorithm assumed that the n-vectors are non-divisible
Reduction on mesh/torus networks

Phase 1: reduce vertically
Phase 2: Reduce horizontally

Time: $2(\sqrt{p-1})(\alpha+\beta n)$

Note:
Much better algorithms possible with pipelining, dividing n-vectors into blocks, reducing by iterating over the blocks
Collective operations: Motivation

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

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

as a „collective operation“ in MPI
MPI_Reduce(void *sendbuf, void *recvbuf,
    int count, MPI_Datatype datatype,
    MPI_Op op,
    int root,
    MPI_Comm comm);

• Saves work for application programmer: No need to implement complicated, own library functions
• Improves **portability**: Part of MPI standard, available everywhere
• Improves **performance portability**: Good MPI implementations will provide „best possible“ performance for given system
Collective communication (and reduction) operations

**MPI_Bcast:** Data from root to all

**MPI_Scatter:** Individual (personalized) data from root to all

**MPI_Gather:** Individual data from all to root

**MPI_Alltoall:** Individual (personalized) data from all to all, “transpose”

**MPI_Allgather:** Data from all to all

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

**MPI_Allreduce:** Result to all

**MPI_Reduce_scatter:** Result scattered (parts) to all

**MPI_Barrier:** (semantic) Synchronization
### Complexity of collective operations (fully connected network)

<table>
<thead>
<tr>
<th>MPI Collective</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier</td>
<td>$O(\log p)$</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>$O(n+\log p)$</td>
</tr>
<tr>
<td>MPI_Gather/Scatter</td>
<td>$O(n+\log p)$</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>$O(n+\log p)$</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>$O(n+p)$ (*)</td>
</tr>
<tr>
<td>MPI_Reduce</td>
<td>$O(n+\log p)$</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>$O(n+\log p)$</td>
</tr>
</tbody>
</table>

Important to find algorithms with small $O$-constants: see HPC lecture

(*) interesting tradeoffs possible

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

**MPI_Bcast is NOT $O(n \log p)$**
### Complexity of collective operations (general)

<table>
<thead>
<tr>
<th>MPI Collective</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_Barrier</td>
<td>$O(d)$</td>
</tr>
<tr>
<td>MPI_Bcast</td>
<td>$O(n+d)$</td>
</tr>
<tr>
<td>MPI_Gather/Scatter</td>
<td>$O(n+d)$</td>
</tr>
<tr>
<td>MPI_Allgather</td>
<td>$O(n+d)$</td>
</tr>
<tr>
<td>MPI_Alltoall</td>
<td>$O(n+p)$ (*)</td>
</tr>
<tr>
<td>MPIReduce</td>
<td>$O(n+d)$</td>
</tr>
<tr>
<td>MPI_Allreduce</td>
<td>$O(n+d)$</td>
</tr>
</tbody>
</table>

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

Important to find algorithms with small $O$-constants: see HPC lecture

(*) interesting tradeoffs possible
Collective MPI operations

All functions of MPI requiring participation of all processes in communicator

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

The 17 (16 in MPI 1) collective communication (and reduction) operations are called the "collectives" of MPI
Collective MPI operations are called the same way by the participating processes, same arguments for all processes, but some arguments may be significant only at some processes (root)

Process $i$ (non-root):

\[
\text{MPI\_Reduce}(\text{sbuf}, \text{rbuf}, \ldots, \text{root, comm});
\]

Process $j$ (root):

\[
\text{MPI\_Reduce}(\text{sbuf}, \text{rbuf}, \ldots, \text{root, comm});
\]

But: All processes in comm must participate

Significant at root only
Example: Reduction of single “scalar” (C int, MPI_INT)

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

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

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

Requirement:
Collective operations must be called with consistent arguments: same root, same op, exactly matching amounts of data (see individual functions)
Collective operation semantics

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

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

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

**Exception:** MPI_Barrier(comm);

(*) nonblocking collectives from MPI 3.0
**Correct:**

**Process i:**

\[ \text{MPI\_Bcast(buffer,\ldots,root,comm);} \]

**Process j:**

\[ \text{MPI\_Bcast(buffer,\ldots,root,comm);} \]

**Process local time**

**MPI\_Bcast** is **blocking**:
- **root**: does not return **before** data have left buffer
- **Non-root**: does not return **before** data from **root** have been received in buffer
Correct:

**Process i:**

MPI_Bcast(buffer,...,root,comm);

**Process j:**

MPI_Bcast(buffer,...,root,comm);

**MPI_Bcast is not synchronizing:**

root: may return as soon as data have left buffer (independent of non-roots)
Non-root: may return as soon as data from root have been received in buffer (independent of other non-roots)
Incorrect:

Process i:

MPI_Bcast(buffer, ..., root, comm);
MPI_Reduce(sbuf, rbuf, ..., root, comm);

Process j:

MPI_Reduce(sbuf, rbuf, ..., root, comm);
MPI_Bcast(buffer, ..., root, comm);

Process local time

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

Process i:

MPI_Bcast(buffer, ..., root, comm2);
MPI_Gather(sendbuf, ..., comm1);

Process k:

MPI_Bcast(buffer, ..., root, comm2);

Process j:

MPI_Gather(sendbuf, ..., comm1);

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

Process local time
Unsafe:

Process i:

MPI_Bcast(buffer,...,root,comm2);
MPI_Gather(sbuf,...,root,comm1);

Process j:

MPI_Gather(sbuf,...,root,comm1);
MPI_Bcast(buffer,...,root,comm2);

Process k:

MPI_Bcast(buffer,...,root,comm2);

Process local time

Unsafe:

May work for small counts, hang for large

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

Process i:

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

Process j:

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

Point-to-point and one-sided and collective communication does not interfere.
Process involvement in/blocking behavior of collective call MPI_{A} is implementation dependent

Unsafe collective programming: Relying on synchronization properties

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

If it seems so, there's probably something wrong
Calling process waits for all other processes in `comm` to enter barrier, can leave when all others have performed call

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

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

Sometimes HW support helps (atomic counters, collective network)
Example: Timing a function

```c
MPI_Barrier(comm);
// processes hopefully now synchronized
double start = MPI_Wtime();

<something to be timed>

double stop = MPI_Wtime();
double local_time = stop - start;

double spent_time; // time for slowest process
MPI_Allreduce(&local_time,
              &spent_time, 1, MPI_DOUBLE, MPI_MAX,
              comm);
```
Example: Benchmarking

1. Repeat measurement until stable, reproducible result has been achieved

2. Collect all results at root (or all processes)

3. Report average, best and worst
double start[REPETITIONS], stop[REPETITIONS];
double local_time[REPETITIONS];
for (int r=0; r<REPETITIONS; r++) {
    MPI_Barrier(comm);
    // processes hopefully now synchronized
    start[r] = MPI_Wtime();

    <something to be timed>

    stop[r] = MPI_Wtime();
    local_time[r] = stop[r] - start[r];
}
double spent_time[REPETITIONS];
MPI_Allreduce(local_time,
    spent_time,REPETITIONS,MPI_DOUBLE,MPI_MAX, comm);
A (legal) barrier implementation: not suitable for timing!

MPI libraries do something better...

Phase 1: "gather"

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

Phase 2: "scatter"

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

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

for (i=1; i<p; i++)
MPI_Recv(NULL,0,...,comm);
## MPI „collectives“ classification

<table>
<thead>
<tr>
<th>Class</th>
<th>regular</th>
<th>Irregular, vector</th>
</tr>
</thead>
<tbody>
<tr>
<td>Symmetric, no data</td>
<td>MPI_Barrier</td>
<td></td>
</tr>
<tr>
<td>Rooted</td>
<td>MPI_Bcast</td>
<td></td>
</tr>
<tr>
<td>Rooted</td>
<td>MPI_Scatter</td>
<td>MPI_Scatterv</td>
</tr>
<tr>
<td>Rooted</td>
<td>MPI_Gather</td>
<td>MPI_Gatherv</td>
</tr>
<tr>
<td>Symmetric, non-rooted</td>
<td>MPI_Allgather</td>
<td>MPI_Allgatherv</td>
</tr>
<tr>
<td>Symmetric, non-rooted</td>
<td>MPI_Alltoall</td>
<td>MPI_Alltoallv, MPI_Alltoallw</td>
</tr>
<tr>
<td>Rooted</td>
<td>MPI_Reduce</td>
<td></td>
</tr>
<tr>
<td>Non-rooted</td>
<td>MPI_Reduce_scatter_block</td>
<td>MPI_Reduce_scatter</td>
</tr>
<tr>
<td>Symmetric, non-rooted</td>
<td>MPI_Allreduce</td>
<td></td>
</tr>
<tr>
<td>Non-rooted</td>
<td>MPI_Scan</td>
<td></td>
</tr>
<tr>
<td>Non-rooted</td>
<td>MPI_Exscan</td>
<td></td>
</tr>
</tbody>
</table>

(*) MPI_Reduce_scatter_block: MPI 2.2 extension
Symmetric vs. non-symmetric: all processes have the same role in collective vs. one/some process (root) is special

Regular vs. irregular: each process contributes or receives the same amount of data from/to each other process vs. different pairs of processes may exchange different amounts of data

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

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

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

Rank: A0

before

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

Example: root==0

0: \( A_0 \) \( A_0 \)
1: \( A_0 \) \( A_0 \)
2: \( A_0 \) \( A_0 \)
3: \( A_0 \) \( A_0 \)
4: \( A_0 \) \( A_0 \)

**Semantics:** Data from root buffer is transferred to buffer of all non-root processes.

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

Example: root==2

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

Use: All processes Bcast with same root, buffer with same type signature (e.g. same count for basic datatypes like MPI_FLOAT)
Collective functions MUST be called with consistent arguments:

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

```c
int matrixdims[3]; // 3 dimensional matrix
if (rank==0) {
    MPI_Bcast(matrixdims,3,MPI_INT,0,comm);
} else {
    // do something on non-root first
    MPI_Bcast(matrixdims,2,MPI_INT,0,comm);
    // uhuh, Bcast probably works, but later...
}
```
**MPI requirement**

Collective functions **MUST** be called with **consistent arguments**:
- Same root
- Matching type signatures (in particular: pairwise same size)
- **Note**: Number of elements sent and received must **match exactly** (unlike Send-Recv: sent ≤ recv and Get/Put)
- Same **op** (MPI_Reduce etc.)

**Calling with different roots probably just deadlocks**

For efficiency reasons, MPI libraries **do not check**. User on his/her own when making mistakes. **Consistency tools** can help
MPI_Gather(sbuf, scount, stype, rbuf, rcount, rtype, root, comm);

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

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

Note: rcount is count of one block, not of whole rbuf
MPI_Gather(sbuf, scount, stype, rbuf, rcount, rtype, root, comm);

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

Block from process $i$ is stored at $rbuf + i \times rcount \times \text{extent}(rtype)$

Note: $rcount$ is count of one block, not of whole $rbuf$
MPI_Gather(sbuf, scount, stype, rbuf, rcount, rtype, root, comm);

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

Note: root also gathers from itself

Special MPI buffer argument MPI_IN_INPLACE can be used on root for sbuf to indicate that result from root is already „in place“
MPI_Gather(sbuf, scount, stype, rbuf, rcount, rtype, root, comm);

is semantically equivalent to

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

Note: Semantics only, typically not implemented this way
MPI_Scatter(sbuf, scount, stype, rbuf, rcount, rtype, root, comm);

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

Block from process root is stored at `sbuf+i*scount*extent(stype)`
Send buffer \((\text{sbuf}, \text{scount}, \text{stype})\) significant only on root

\textbf{MPI\_IN\_INPLACE} can be used on root for \text{rbuf} to indicate that result from root is already „in place“
Example: Distributing initial array, collecting result

A: \[ n \]

\[ \text{MPI}_{-}\text{Scatter}(A,\ldots); \]

B: \[ \text{MPI}_{-}\text{Gather}(B,\ldots); \]

C: \[ \text{MPI}_{-}\text{Gather/}\text{MPI}_{-}\text{Scatter: All blocks same size} \]
int root = 0;
MPI_Comm_size(comm, &p);
MPI_Comm_rank(comm, &r);
assert(n%p==0);
if (rank==root) {
    int *a = (int*)malloc(n*sizeof(int));
    int *c = (int*)malloc(n*sizeof(int));
    for (i=0; i<n; i++) a[i] = <init>;
}
int *b = (int*)malloc((n/p)*sizeof(int));
MPI_Scatter(a,n/p,MPI_INT,
    b,n/p,MPI_INT,root,comm);

... // compute on b: all processes

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

- $p$ divides $n$
- If not, use `MPI_Scatterv`, irregular collective, see later
Challenge: Scatter submatrices

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

2. MPI_Scatter will not work... (why?)

3. MPI_Type_create_resize with MPI_Scatterv can solve this problem

For project: Generate submatrices locally, or scatter manually with p times send and receive
Further differences to point-to-point communication:

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

Process i:

```c
MPI_Bcast(buffer, 0, MPI_CHAR, ..., root, comm);
```

Process j:

```c
MPI_Bcast(buffer, 0, MPI_CHAR, ..., root, comm);
```

**Correct!** May be implemented as no-op, nothing broadcast
Further differences to point-to-point communication:

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

Process \( i \):

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

Process \( j \):

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

**Correct! But** an empty message must be sent
Further differences to point-to-point communication:

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

```c
Process i:
MPI_Send(buffer, 0, MPI_CHAR, j, TAG, comm);
```

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

Correct! But an empty message must be sent, since receive count could be greater 0 and sender cannot know this
Does this barrier work?

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

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

Unsafe collective programming: Relying on synchronization properties
MPI_Allgather(sbuf, scount, stype, rbuf, rcount, rtype, comm);

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

Block from process i is stored at rbuf+i*rcount*extent(rtype)
MPI_Allgather(sbuf, scount, stype, rbuf, rcount, rtype, comm);

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

**MPI_IN_INPLACE** can be used for sbuf to indicate that local part of result is already „in place“
MPI_Allgather(sbuf,...rbuf,rcount,rtype,...comm);

is equivalent to

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

and

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

But: Performance of library function should be better!
Fact:
Much better algorithms for MPI_Allgather than MPI_Gather+MPI_Bcast exist

A good MPI implementation will ensure that “best possible” algorithm is implemented, and that indeed MPI_Allgather always (all other things being equal) performs better than MPI_Gather+MPI_Bcast

Golden MPI rule

• Use collectives for conciseness and performance wherever possible

• Complain to MPI library implementer if performance anomalies are discovered
Example: parallel matrix-vector multiplication

**Problem specification:** given nxn matrix $M$ and n vector $V$, compute product n vector $W$

$W = M \times V$

where $W[j] = \sum_{0 \leq i < n} M[j][i] \times V[i]$

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

Both $M$ and $V$ should be distributed evenly over the MPI processes; result vector $W$ should be distributed as $V$. 
**Distribution is part of problem specification**
Solution 1: Matrix-vector multiplication

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

For process \( k, 0 \leq k < p \), rows \( k(n/p) \leq j < (k+1)(n/p) \):

\[
W'[j] = \sum_{0 \leq i < n} M'[j][i] \times V[i]
\]

\( j \) index in global matrix and result vector, \( j-k(n/p) \) index in local row-matrix \( M' \) and local result vector \( W' \)

**Solution:** Distribute \( M \) row-wise into local parts \( M' \), all-gather \( V \) from pieces \( V' \) on each process, perform \( M'V \) locally
Distribution

local $M', V'$

0:

1:

2:

3:

4:
Step 1: gather $V$ at all processes

**MPI_Allgathers**

- Full $V$

<table>
<thead>
<tr>
<th>Process</th>
<th>$M'$, $V'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>
Step 2: locally compute $M \times V$ in parallel

0: local $M'$, $V'$
1: full $V$
2: $x$
3: $=$
4: local $W'$
Complexity summary:

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

Linear speedup for $p\leq n$
MPI_Alltoall(sbuf, scount, stype, rbuf, rcount, rtype, comm);

<table>
<thead>
<tr>
<th></th>
<th>0: A0</th>
<th>A1</th>
<th>A2</th>
<th>A3</th>
<th>A4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>B0</td>
<td>B1</td>
<td>B2</td>
<td>B3</td>
<td>B4</td>
</tr>
<tr>
<td>2:</td>
<td>C0</td>
<td>C1</td>
<td>C2</td>
<td>C3</td>
<td>C4</td>
</tr>
<tr>
<td>3:</td>
<td>D0</td>
<td>D1</td>
<td>D2</td>
<td>D3</td>
<td>D4</td>
</tr>
<tr>
<td>4:</td>
<td>E0</td>
<td>E1</td>
<td>E2</td>
<td>E3</td>
<td>E4</td>
</tr>
</tbody>
</table>

- Transpose
- All-to-all personalized communication
MPI_Alltoall(sbuf, scount, stype, rbuf, rcount, rtype, comm);

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

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

Block from process i is stored at rbuf+i*rcount*extent(rtype)
Irregular (vector, v-) collectives:
Possibly different amounts of data destined to different processes

- MPI_Gatherv, MPI_Scatterv
- MPI_Allgatherv
- MPI_Alltoally, MPI_Alltoallw

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

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

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

Segments to be transferred to/from different ranks may have different size (count[i]), and different displacement (displ[i]) relative to start address. Displacement is in datatype units
MPI_Gatherv(sbuf, scount, stype, rbuf, rcount, rdisp, rtype, root, comm)

Received data must not overlap. Displacement and count vectors significant only at root. Size/signature match pairwise.
MPI_Gatherv(sbuf, scount, stype, rbuf, rcount, rdisp, rtype, root, comm)

Block from process i stored at rbuf+disp[i]*extent(rtype)

Displacement in extent(rtype) units; same for all processes (one rtype only)
Example: Root gathers unknown amount of data from all processes

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

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

`MPI_Gatherv` requires that `rcount[i]` equals `scount` of process `i` (if `stype` and `rtype` are same)
Example: Root gathers unknown amount of data from all processes

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

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

Then correct `MPI_Gatherv` call can be set up
Reduction collectives

- Each process has a vector of data $X$ (same size, same type)
- Associative operation $+$ ($\text{MPI builtin, MPI\_SUM, etc., user def}$)
- Element-wise reduction result $Y = X_0 + X_1 + X_2 + \ldots + X(p-1)$ is stored at
  1. **Root:** MPI\_Reduce
  2. **All processes:** MPI\_Allreduce
  3. **Scattered in blocks** ($Y_0$ to 0, $Y_1$ to 1, ...): MPI\_Reduce\_Scatter
Reductions are performed elementwise on the input vectors
Binary operation "+" is required (and assumed by MPI implementation) to be associative:

\[ X_1 + (X_2 + (X_3 + (X_4 + X_5))) = (X_1+X_2)+(X_2+(X_3+X_4)) = X_1 + X_2 + X_3 + X_4 + X_5 \]

By associativity: Result independent of "bracketing", partial results \(X_i+...X_j\) can be computed in parallel

If operation is commutative, this can also be exploited

Note:

MPI functions are **not** mathematical functions, i.e., not really associative (MPI_FLOAT). Good MPI implementations are careful with exploiting associativity (and commutativity)
MPI_Reduce(sbuf, rbuf, count, type, op, root, comm);

Example: root==0

**Semantics:** All processes contribute sbuf (vector) of same size, elementwise result stored in rbuf at root. With MPI_IN_PLACE as sbuf, input is taken from rbuf.
MPI_Allreduce(sbuf, rbuf, count, type, op, comm);

Semantics: All processes contribute `sbuf` (vector) of same size, elementwise result stored in `rbuf` at all processes. With `MPI_IN_PLACE` as `sbuf`, input is taken from `rbuf`
MPI_Reduce_scatter_block(sbuf, rbuf, count, type, op, comm);

<table>
<thead>
<tr>
<th>0:</th>
<th>A0</th>
<th>B0</th>
<th>C0</th>
<th>D0</th>
<th>E0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1:</td>
<td>A1</td>
<td>B1</td>
<td>C1</td>
<td>D1</td>
<td>E1</td>
</tr>
<tr>
<td>2:</td>
<td>A2</td>
<td>B2</td>
<td>C2</td>
<td>D2</td>
<td>E2</td>
</tr>
<tr>
<td>3:</td>
<td>A3</td>
<td>B3</td>
<td>C3</td>
<td>D3</td>
<td>E3</td>
</tr>
<tr>
<td>4:</td>
<td>A4</td>
<td>B4</td>
<td>C4</td>
<td>D4</td>
<td>E4</td>
</tr>
</tbody>
</table>

**Semantics:** All processes contribute sbuf (vector) of same size, elementwise result is scattered in same sized blocks and stored in rbuf at each process. With MPI_IN_PLACE as sbuf, input is taken from rbuf.
MPI_Reduce_scatter(sbuf, rbuf, counts, type, op, comm);

**Semantics:** All processes contribute sbuf (vector) of same size, elementwise result is scattered in blocks and stored in rbuf at each process. With MPI_IN_PLACE as sbuf, input is taken from rbuf. Since rbuf may have different size at different processes, counts[] is a vector. All processes provide same counts[] vector.
<table>
<thead>
<tr>
<th><strong>MPI_Op</strong></th>
<th><strong>function</strong></th>
<th><strong>Operand type</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>MPI_MAX</td>
<td>max</td>
<td>Integer, Floating</td>
</tr>
<tr>
<td>MPI_MIN</td>
<td>min</td>
<td>Integer, Floating</td>
</tr>
<tr>
<td>MPI_SUM</td>
<td>sum</td>
<td>Integer, Floating</td>
</tr>
<tr>
<td>MPI_PROD</td>
<td>product</td>
<td>Integer, Floating</td>
</tr>
<tr>
<td>MPI_LAND</td>
<td>logical and</td>
<td>Integer, Logical</td>
</tr>
<tr>
<td>MPI_BAND</td>
<td>bitwise and</td>
<td>Integer, Byte</td>
</tr>
<tr>
<td>MPI_LOR</td>
<td>logical or</td>
<td>Integer, Logical</td>
</tr>
<tr>
<td>MPI_BOR</td>
<td>bitwise or</td>
<td>Integer, Byte</td>
</tr>
<tr>
<td>MPI_LXOR</td>
<td>logical exclusive or</td>
<td>Integer, Logical</td>
</tr>
<tr>
<td>MPI_BXOR</td>
<td>bitwise exclusive or</td>
<td>Integer, Byte</td>
</tr>
<tr>
<td>MPI_MAXLOC</td>
<td>max value and location of max</td>
<td>Special pair type</td>
</tr>
<tr>
<td>MPI_MINLOC</td>
<td>min value and location of min</td>
<td>Special pair type</td>
</tr>
</tbody>
</table>
makes it possible to define/register own, “user-defined”, binary, associative operators that can even work on derived datatypes

And free it again after use…
Solution 2: Matrix-vector multiplication

Assume p divides n, distribute $M$ column-wise, each process has $n/p$ columns of $M$, $n/p$ elements of $V$

For process $k$, $0 \leq k < p$, columns $c_i \leq j < (c_{i+1})$, where $c_i = k(n/p)$

$$W[j] = \sum_{c_0 \leq i < c_1} M'[j][i] \cdot V'[i] + \sum_{c_1 \leq i < c_2} M'[j][i] \cdot V'[i] + \ldots + \sum_{c_{(p-1)} \leq i < c_p} M'[j][i] \cdot V'[i]$$

Correct since $+$ is associative. Here $i$ is index in global matrix, $i$-ck index in local column-matrix $M'$ and local vector $V'$

Solution: distribute $M$ column-wise, perform local $M'V$, sum partial results
Solution 2: Matrix-vector multiplication

Each rank has \(\frac{n}{p}\) columns of \((nxn)\) matrix, \(\frac{n}{p}\) rows of vector

\[
\begin{bmatrix}
0: & 1: & 2: & 3: & 4: \\
\text{Green} & \text{Yellow} & \text{Orange} & \text{Red} & \text{Green}
\end{bmatrix}
\times
\begin{bmatrix}
\text{Green} & \text{Orange} & \text{Red}
\end{bmatrix}
= \begin{bmatrix}
\text{Yellow} & \text{Red}
\end{bmatrix}
\]
1. Locally compute $(nxn/p)$ matrix $n/p$ vector product, $M'V'$

\[ \text{partial result n vector} \]
1. Locally compute \((nxn/p)\) matrix \(n/p\) vector product, \(M'V'\)
1. Locally compute \((\text{n} \times \text{n}/\text{p})\) matrix \(n/p\) vector product, \(M'V'\)
1. Locally compute \((n \times n/p)\) matrix \(n/p\) vector product, \(M'V'\)
1. Locally compute \((n \times n/p)\) matrix \(n/p\) vector product, \(M'V'\)
2. Sum partial result n vectors and scatter n/p blocks

\[\begin{align*}
0: &+ 1: + 2: + 3: + 4: = \\
\end{align*}\]

```
MPI_Reduce_scatter_block(partial, result, n/p, MPI_FLOAT, MPI_SUM, comm);
```

Each rank stores n/p rows of result vector
2. Sum partial result \( n \) vectors and scatter \( n/p \) blocks

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

Each rank stores \( n/p \) rows of result vector
Complexity summary:

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

Linear speedup for $p\leq n$

Which method is better? Implement and test
Scan collectives

- Each process has vector of data X (same size, same type)
- Associative operation + (MPI builtin, MPI_SUM,..., user def)
- All prefix sums $Y_i = X_0 + ... + X_i$ are computed and stored
- $Y_i$ at rank $i$: MPI.Scan
- $Y_i$ at rank $i+1$: MPI.Exscan (rank 0 undefined)
MPI_Scan(sbuf, rbuf, count, type, op, comm);

Semantics: Elementwise, inclusive prefix sums over sbuf vectors. Rank i stores sum from rank 0 to rank i (included)
MPI_Exscan(sbuf, rbuf, count, type, op, comm);

**Semantics:** Elementwise, exclusive prefix sums over sbuf vectors. Rank \(i\) stores sum from rank 0 to rank \(i\) (excluded)
Example: Distributed memory Quicksort

Quicksort bottleneck: Partition

Idea:
Distribute pivot value across all processes, do local partition at each process; exclusive scan (prefix sums) to compute indices of elements smaller equal, and larger equal pivot value; redistribute into new distributed array

Split set of processes into those with small and those with large elements, recurse
Initial data distribution: Each process has local part of input of size roughly $n/p$ (last process may have less)

Desired result: Also distributed, each process has roughly $n/p$ elements in order
Initial data distribution: Each process has local part of input of size roughly $n/p$ (last process may have less)

Intermediate result might be unbalanced (but too unbalanced is undesirable):

Need to balance in the end (project: it is allowed to be unbalanced, but discuss this; or implement rebalancing step)
Step 1: Choose pivot, distribute to all processes

1. Process 0 chooses pivot, \texttt{MPI\_Bcast}
2. Better: Each process chooses pivot, \texttt{MPI\_Allgather}, median of p as pivot

Step 2: Local partition
Step 3: Global partition, pairwise (hypercube) exchange

Even numbered process $i$ sends larger than pivot elements to process $i+1$, and receives smaller than pivot elements from process $i+1$. Odd numbered process $i$ receives larger than pivot elements from process $i-1$ and sends smaller than pivot elements to process $i$: MPI_Sendrecv

Implementation: May need to exchange number of elements first
Step 4: Split communicator and recurse

Use `MPI_Comm_split` with color as `rank%2`: Even numbered processes will work on smaller than pivot elements, odd numbered processes on larger than pivot elements.

After $\log_2 p$ steps, each process is in communicator by itself: Sort locally.

Youran Lan, Magdi A. Mohamed: Parallel Quicksort in hypercubes. SAC 1992: 740-746
Drawbacks:

• Only for powers-of-two numbers of processors (hypercubed algorithm)
• Load balance might be arbitrarily bad, one process could do all the work if pivot is bad

...but ok for project

Analysis, assuming exact median pivot:

\[ T(n,p) = O(p) + O(n/p) + T(n/2, p/2) \]
\[ T(n,1) = O(n \log n) \]

Since \( (n/2)/(p/2) = n/p \), and \( 2^{\log p} = p \), we get

\[ T(n,p) = O(p \log p + (\log p)(n/p)) + O(n/p \log(n/p)) = \]
\[ O(p \log p) + O((\log p)(n/p) + (n/p)(\log n) - (n/p)(\log p)) = \]
\[ O(p \log p) + O(n/p \log n) \]

Speedup: \( O((n \log n)/(n/p \log n)) = O(p) \) for \( n \gg p \)
Improved step 3: Global partition by compaction

Smaller than pivot elements:
- For process $i$, compute how many smaller than pivot elements there are at process 0, 1, ..., $i-1$: `MPI_Exscan` with result $m_i$
- Elements have to be sent to process $m_i/(n/p)$ and (possibly) process $m_i/(n/p)+1$
- All processes need to inform their receiving processes (at most two) how many elements to receive: `MPI_Alltoall`
- Redistribution by `MPI_Alltoallv` or send and receive
Improved step 3: Global partition by compaction

Larger than pivot elements: Similar

Drawback/careful with elements in the middle. Each process can have only either small than pivot or larger than pivot elements; some load imbalance may occur.
Step 4: Split communicator and recurse

Use `MPI_Comm_split` with color=(rank<middle ? 0 : 1) to divide the processes into those with smaller than and those with larger than pivot elements.

After $\log_2 p$ steps, each process is in communicator by itself:
Sort locally

Same analysis applies, but good load balance irrespective of pivot choice.
Example: Integer (bucket, counting) sorting

Given n integers in range \([0,R[ (=\{0,1,2,\ldots,R-1\}) stored in array \(A\)

Bucket sort:
1. Count number of elements of each magnitude (“key”)
   
   \[
   \begin{align*}
   \text{for } (i=0; i<R; i++) \text{ bucket}[i] &= 0; \\
   \text{for } (i=0; i<n; i++) \text{ bucket}[A[i]] &= +;
   \end{align*}
   \]

2. Compute start index of each bucket: exclusive prefix-sums operation
   
   \[
   \begin{align*}
   \text{for } (i=1; i<R; i++) \text{ bucket}[i] &= + \text{ bucket}[i-1]; \\
   \text{for } (i=R-1; i>0; i--) \text{ bucket}[i] &= \text{ bucket}[i-1]; \\
   \text{bucket}[0] &= 0;
   \end{align*}
   \]
Given n integers in range \([0, R[ (=0, 1, 2, \ldots, R-1])\) stored in array \(A\)

3. Distribute elements of \(A\) into buckets (in new array \(B\))

```plaintext
for (i=0; i<n; i++) B[bucket[A[i]]++] = A[i];
```

4. Copy \(B\) back to \(A\), if needed

**Note:**
As implemented, this bucket-sort is stable: relative order or elements in \(A\) with same key is preserved
Example: Integer (bucket, counting) sorting in parallel

n integers in a given range \([0, R]\), distributed evenly across \(p\) MPI processes: \(m = \frac{n}{p}\) integers per process

\[
A = \begin{array}{c}
0 & 1 & 3 & 0 & 0 & 2 & 0 & 1 \\
\end{array} \\
B = \begin{pmatrix}
4 \\
2 \\
1 \\
3 \\
\end{pmatrix}
\]

Input array distributed over \(p\) processes, \(A\) and \(B\) process local arrays of input elements and bucket sizes
n integers in a given range \([0, R]\), distributed evenly across \(p\) MPI processes: \(m = n/p\) integers per process

\[
A = \begin{bmatrix} 0 & 1 & 3 & 0 & 0 & 2 & 0 & 1 & \ldots \end{bmatrix} \quad B = \begin{pmatrix} 4 \\ 2 \\ 1 \\ 3 \end{pmatrix}
\]

Step 1: bucket sort locally, \(B[i]\) number of elements with key \(i\)

Step 2: \texttt{MPI\_Allreduce}(B,AllB,R,MPI\_INT,MPI\_SUM,comm);

Step 3: \texttt{MPI\_Exscan}(B,RelB,R,MPI\_INT,MPI\_SUM,comm);

Now:
Vector \(\text{AllB}\) contains the sizes of all buckets over all processes;
\(\text{RelB}\) is for process rank the relative position of rank's elements in the buckets
n integers in a given range [0,R[, distributed evenly across p MPI processes: \( m = \frac{n}{p} \) integers per process

\[
A = \begin{bmatrix}
0 & 1 & 3 & 0 & 0 & 2 & 0 & 1 & \ldots
\end{bmatrix}
\quad B = \begin{bmatrix}
4 \\
2 \\
1 \\
3
\end{bmatrix}
\]

Step 1: bucket sort locally, \( B[i] \) number of elements with key \( i \)

Step 2: \texttt{MPI\_Allreduce}(B, AllB, R, MPI\_INT, MPI\_SUM, comm);

Step 3: \texttt{MPI\_Exscan}(B, RelB, R, MPI\_INT, MPI\_SUM, comm);

Step 4: local exclusive prefix-sums of AllB

Now:
Local element \( A[j] \) needs to go to position \( \text{AllB}[A[j]] + \text{RelB}[A[j]] + j' \) (for \( A[j] > 0 \))

\( j' \) number of local elements with key \( A[j] \) before \( j \)
n integers in a given range \([0,R]\), distributed evenly across \(p\) MPI processes: \(m = \frac{n}{p}\) integers per process

\[ A = \begin{bmatrix} 0 & 1 & 3 & 0 & 0 & 2 & 0 & 1 & \ldots \end{bmatrix} \quad B = \begin{bmatrix} 4 \\ 2 \\ 1 \\ 3 \end{bmatrix} \]

Step 5: compute number of elements to be sent to each other process, \(sendelts[i]\), \(i=0,\ldots,p-1\)

Step 6:
\[
\text{MPI\_Alltoall}(sendelts,1,\text{MPI\_INT},recvelts,1,\text{MPI\_INT},\text{comm});
\]

Step 7: redistribute elements
\[
\text{MPI\_Alltoallv}(A,sendelts,sdispls,\text{MPI\_INT},C,recvelts,\ldots,\text{comm});
\]
n integers in a given range \([0, R]\), distributed evenly across \(p\) MPI processes: \(m = \frac{n}{p}\) integers per process

\[A = \begin{pmatrix} 0 & 1 & 3 & 0 & 0 & 2 & 0 & 1 & \ldots \end{pmatrix} \quad B = \begin{pmatrix} 4 \\ 2 \\ 1 \\ 3 \end{pmatrix}\]

Step 7: redistribute elements
\(\text{MPI\textunderscore Alltoallv}(A, \text{sendelts}, \text{sdispls}, \text{MPI\textunderscore INT}, C, \text{recvelts}, \ldots, \text{comm});\)

Step 8: reorder elements from \(C\) back to \(A\)

Project: Work out steps 4-8

Possible optimization: Replace \(\text{MPI\textunderscore Allreduce}\) by \(\text{MPI\textunderscore Bcast}\)
Example: Integer (bucket, counting) sorting in parallel

The algorithm is stable \( \rightarrow \) Radixsort

Choice of radix \( R \) depends on properties of network (fully connected, fat tree, mesh/torus, ...) and quality of reduction/scan-algorithms

The algorithm is portable (by virtue of the MPI collectives), but tuning depends on systems: Concrete performance model needed, but analysis outside scope of MPI

Note: On strong network \( T(MPI\_Allreduce(m)) = O(m + \log p) \)

\( \text{NOT: } O(m \log p) \)
Quicksort and bucketsort data redistribution

In both cases, the redistribution has special structure, and it may be possible to do better with special algorithm than with **MPI_Alltoallv**

Worth trying:
- **MPI_Exscan** over block sizes; compute where data goes
- **MPI_Win_lock (MPI_LOCK_SHARED); MPI_Put; MPI_Win_unlock** to deliver data
**MPI summary**

- MPI very rich, extensive, versatile realization of message passing programming model/abstraction
- De facto standard for parallel HPC

- Few basic concepts: communicator, datatype, window, MPI handles...

- Point-to-point communication: Blocking and non-blocking sends and receives, combined send-receive, send modes with different semantics
- One-sided communication decouples communication and synchronization, different synchronization mechanisms
- Collective communication involves all processes in communicator, rich set of communication and compute operations, collective bookkeeping operations
MPI 3.0, MPI 3.1

- Significant extension to one-sided model (atomic operations, memory management)
- Non-blocking collectives (MPI_Ibcast, MPI_Ibarrier, ...)
- Sparse (neighbor) collectives (rely on virtual topology mechanism)
- Non-blocking I/O
MPI summary: Realizing the programming model

Programming model

Processes, point-to-point, one-sided, collective, IO, dynamic processes, datatypes

Implementation support:
Efficient communication algorithms

Architecture model

Multi-core processor nodes, communication network
Lecture summary, checklist

- Distributed memory architectures
- Communication networks: topology, routing
- Fundamental lower bound for broadcast (and similar collective operations)

- MPI

Bachelor thesis projects: See us