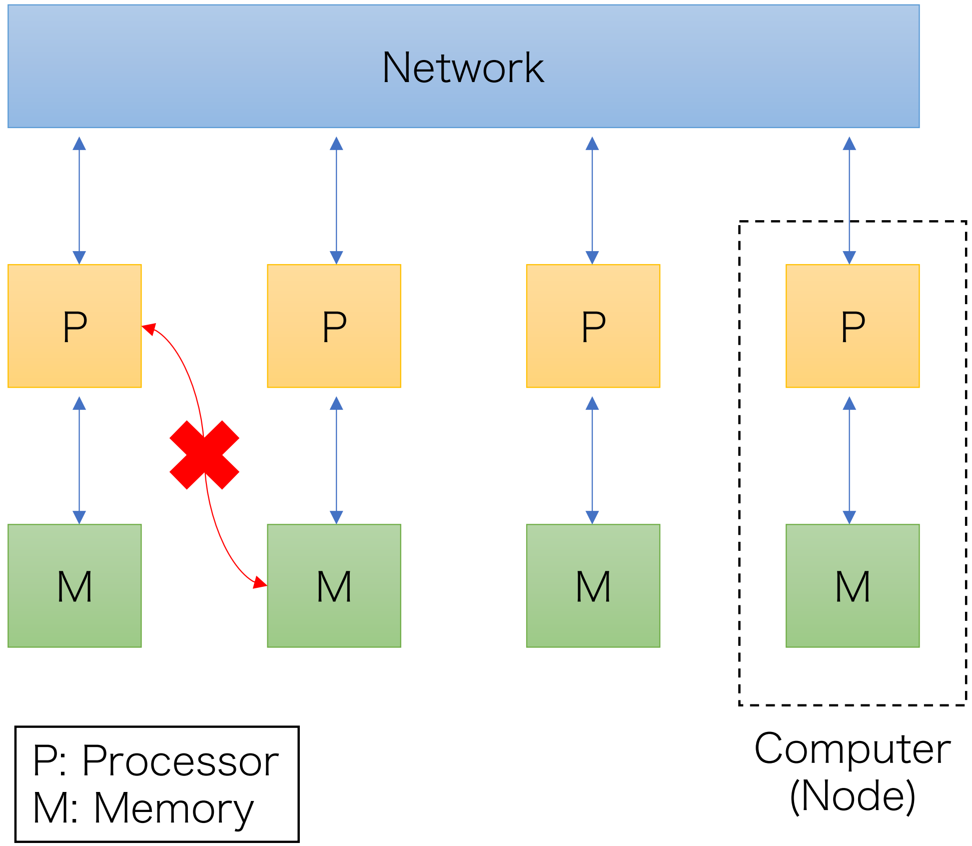
Japan-Korea HPC Winter School & High Performance Parallel Computing Technology for Computational Sciences 「MPI」

Norihisa Fujita

Center for Computational Sciences, University of Tsukuba

Distributed Memory System

- Computer
  - called as "Node"
- Network
  - High-performance network
  - Commodity high-speed network
  - or specialized design for supercomputers
- Distributed Memory
  - Each node has own memory
  - Cannot access memory on different node (not same as OpenMP)



Network for Parallel Computing

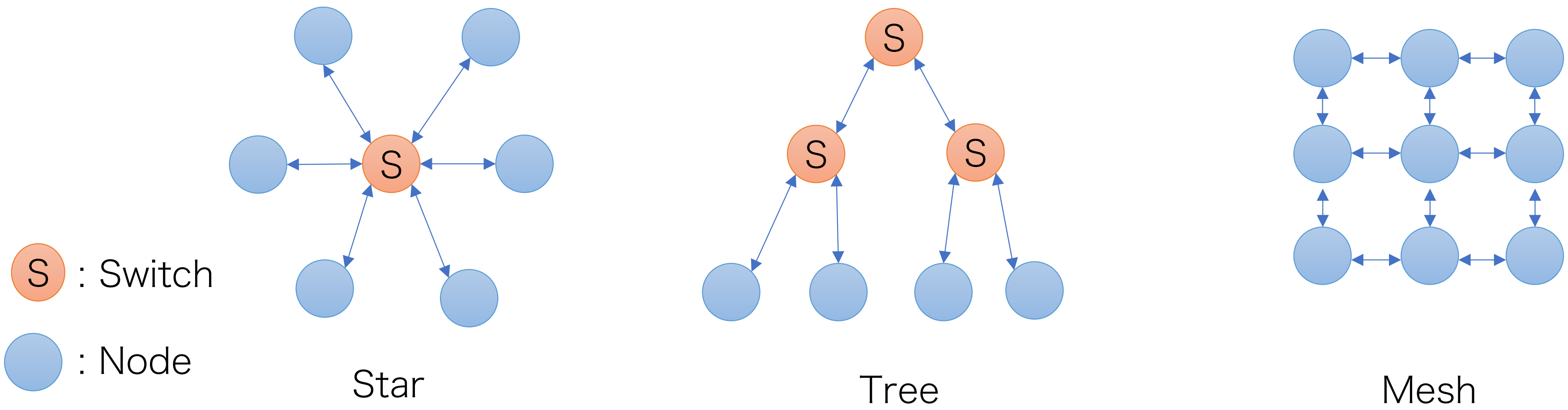
- Performance is important for parallel computing
  - Solving a problem on multiple nodes
  - → high frequent communication is required
- Communication reduces computation performance
  - Thus, keep communication time small as possible
- High-performance network is used in supercomputers
  - For example: Ethernet, InfiniBand
  - Network performance is essential for high-performance computing

Network Elements

- Nodes
  - Computers connected to the network
- Cables
  - node-to-node or node-to-switch
  - Material: copper or optical (Silicon/Glass)
  - Copper cable：Low cost, but only for short-range (~5m) comm.
  - Optical cable：for long-range comm. (~km), but higher cost
- Switches
  - Relay of communication device.
  - Used for constructing large network with multiple nodes
  - Wide-range of scale. from dozens to thousands of ports.
  - Large network consists of multiple switches

Topology

- Topology is abstract structure of connections in network
- Star and tree with switches are widely used
  - All nodes are connected via switches
- Specialized network for HPC may uses Mesh



Ethernet

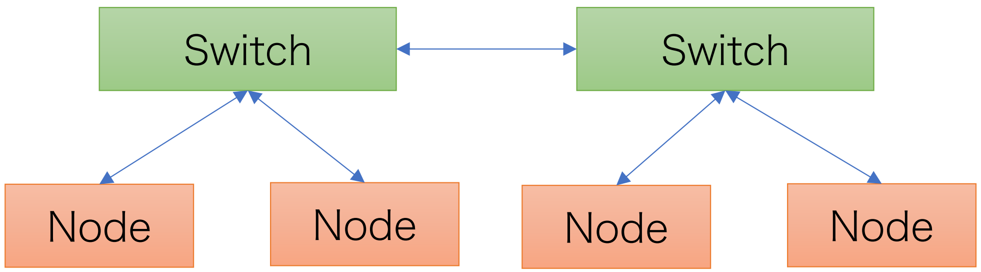
- Ethernet
  - Most widely used network
  - We often call it as “LAN” or “LAN Cable”
  - Gigabit Ethernet (1Gbps) is the most popular one
- Ethernet was born in 1980s
  - Many variations in the standard
  - Speed：1Mbps ~ 100Gbps
  - Cable Distance：100m ~ xx km
- High-speed Ethernet is widely used in supercomputers
  - 100Gbps, 200Gbps, …

Ethernet Switch and Cable

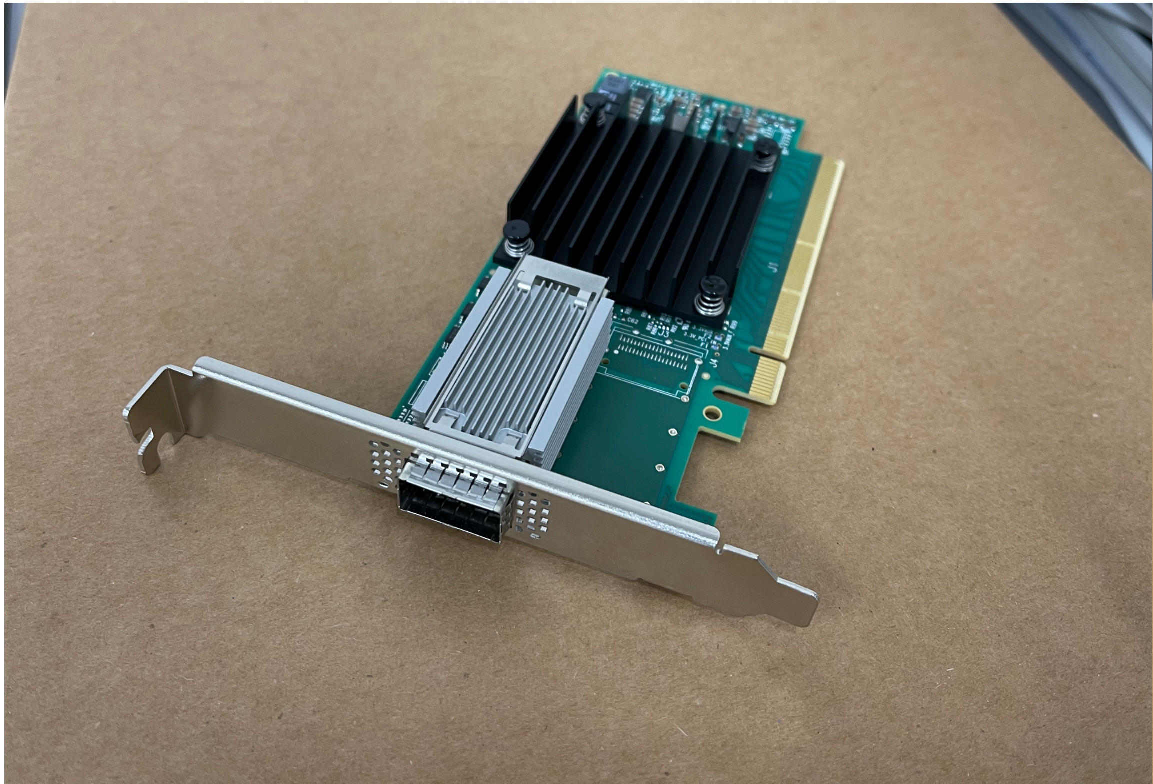
LAN Ports on Wall

InfiniBand

- InfiniBand (IB)
  - One of high-speed networks by NVIDIA (was Mellanox)
  - IB supports wide variety size of network
    - from few nodes to more than 10,000 nodes
  - up to 400 Gbps (IB NDR) is available
  - 800 Gbps is planned



IB network uses Tree topology.
Large network may use multiple stage network in the tree.



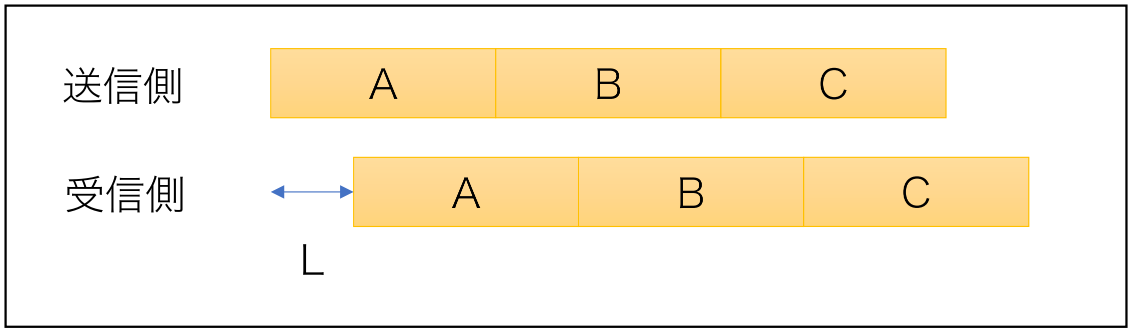
Inifiniband HCA (Host Card Adaptor)

Optical Cable (5m)


NVIDIA
V100

Bandwidth & Latency

- Bandwidth (Byte/s)
  - Amount of data [Byte] transferred in a second
- Latency (s)
  - Data transfer time between sender and receiver
  - Delay of communication
- If we have network bandwidth of B[B/s] with latency of L[s], transfer of N[B] data takes L + N/B [s]
  - If we send data successively, L affects only beginning



Bandwidth & Latency

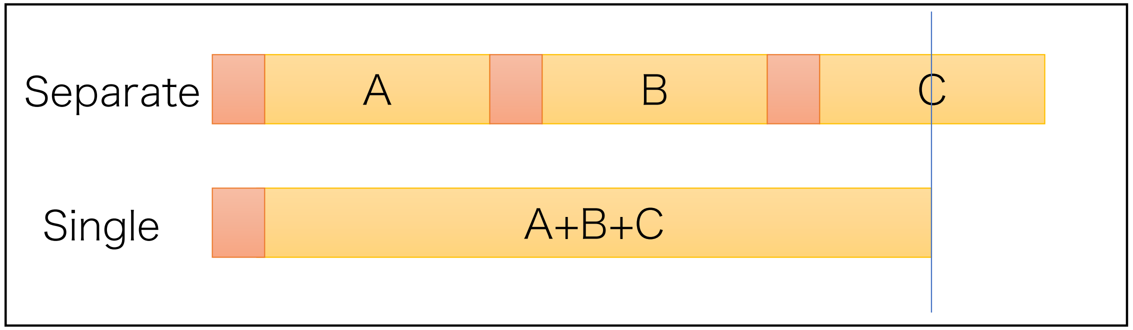
- Examples
  - Transfer of 1GB data takes 10 sec. Find bandwidth of this network.
    - 1 [GB] / 10 [s] = 0.1 [GB/s]

  - How long does it take to transfer 100MB of data at 5MB/s?
    - 100[MB] / 5[MB/s] = 20[s]

  - It takes 100ms to send 1 byte data between two nodes (round-trip). Find latency of this network. Ignore the time for sending data.
    - 100[ms] / 2 = 50[ms]

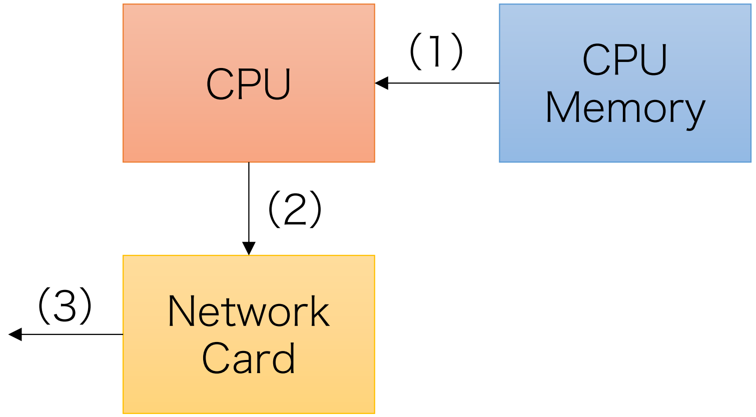
Overhead of Communication

- Communication includes overhead of preparation and control
  - Cost does not change between short and long communication
- Number of communication should be minimized
  - Considering data transfer of A, B, and C, single transfer A+B+C is better than separate transfer

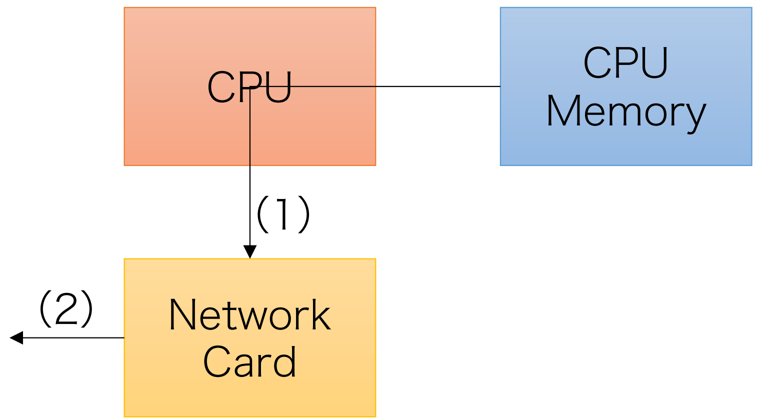


Direct Memory Access (DMA)

- Network cards used in HPC can access CPU memory directly
  - a.k.a. Direct Memory Access (DMA)
- CPU is released form data transfer for communication
  - Optimized data transfer for network
  - CPU can other computation rather than communication



without DMA

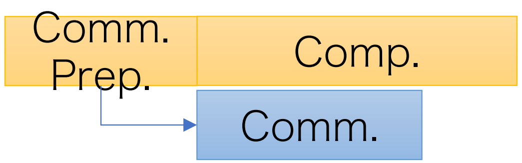


with DMA

Asynchronous Communication

- For CPU, communication is time consuming task
  - Waiting for communication wastes CPU computation time
- Computation and Computation at time same time
  - This is called as "Asynchronous Communication"
  - If DMA is supported, async. comm. is zero overhead
  - Ideally, communication is overlapped with communication completely.
- However, programming becomes complicated
  - Changing code or order of computation may be required
  - Must compute data required by communication before other data

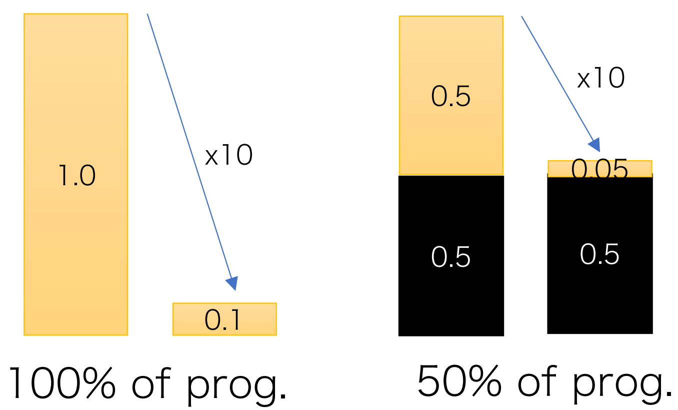
Synchronous Communication



Asynchronous Communication

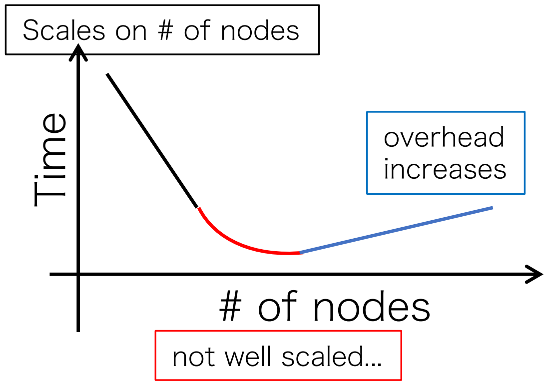
Amdahl's law

- Speed up of parallel computation is dominated by non-parallel part
  - If 10 times speedup is achieved 100% of program
    - 1 / (1 / 10) = x10
  - 90% of program
    - 1 / (0.1 + 0.9 / 10) = x5.26
  - 80% of program
    - 1 / (0.2 + 0.8 / 10) = x3.57
  - 50% of program
    - 1 / (0.5 + 0.5 / 10) = x1.82
  - Assuming infinity speedup…
    - 99%→100 times, 90%→10 times, 50%→2 times



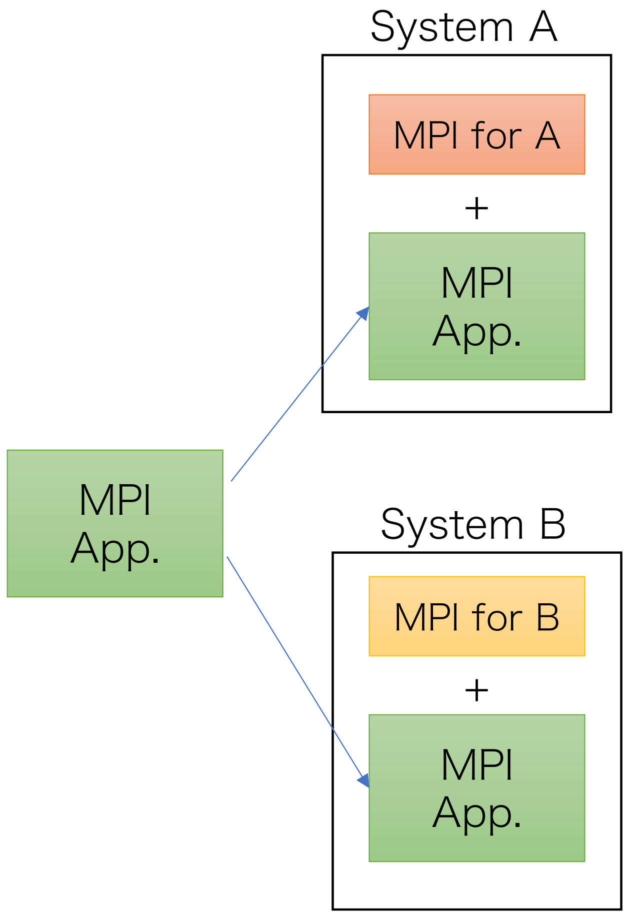
Amdahl's law

- Ideal : Using n nodes, speed is n times faster than 1 node
- Actual : Amdahl's law
  - To achieve x100 speedup on 100 nodes, at least 99% of program must be parallelizable
  - Moreover, communication overhead reduces performance
- per-node computation is 1/100
  - Amount of communication will also be 1/100
  - Time of communication will not be 1/100 due to overhead
- Research to improve communication efficiency is widely studied



Message Passing Interface (MPI)

- Communication library standard widely used in supercomputers
  - Many open-source and proprietary implementations
- De-facto standard for HPC applications
  - MPI is used for sending and receiving data
  - also supports communication patterns frequently used in scientific computation
- Many MPI implementations
  - We can run same applications on multiple systems using different MPI implementations
  - Large supercomputers often have proprietary network
    - → system specific and optimized MPI by system vendor

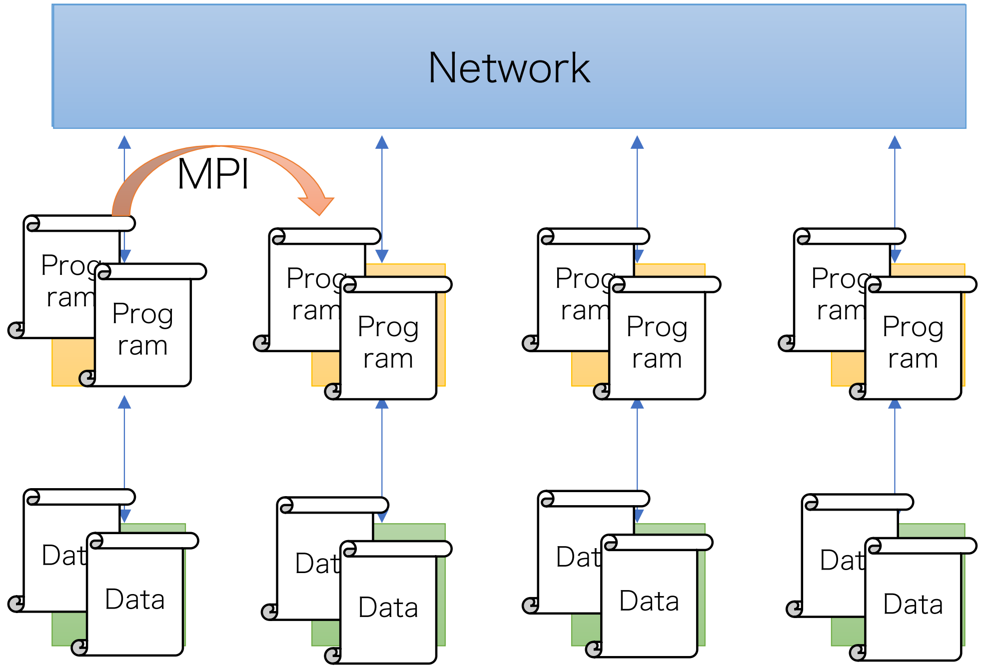


Message Passing Interface (MPI)

- was born in 1992
    - Specification：https://www.mpi-forum.org/
    - 1994: MPI-1.0 release
    - 2009: MPI-2.2 release， 647 pages
    - 2015: MPI-3.1 release， 868 pages
    - 2021: MPI-4.0 release， 1139 pages

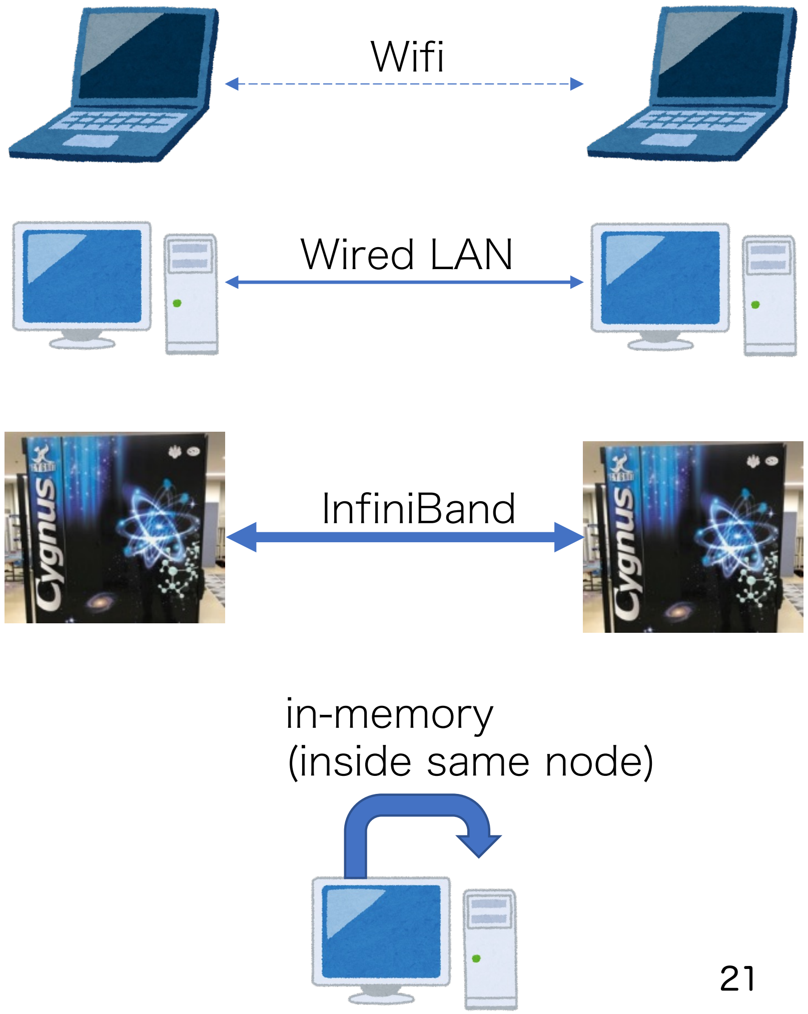
SPMD

- Single Program Multiple Data
  - Run same program
  - but computes on different data
- MPI is SPMD
  - Run same program on nodes
  - program called as “process” in MPI
    - 1 process on 1 node
    - or N procs. on 1 node
  - Describe (what) data transfer between processes using MPI
    - MPI abstracts “how to transfer”
    - We don’t need to care about that
  - Processes run independently without explicit synchronization using MPI



OSS MPIs

- OpenMPI
  - https://www.open-mpi.org/
- MPICH
  - https://www.mpich.org/
- MVAPICH
  - https://mvapich.cse.ohio-state.edu/
- Scalable implementation
  - from laptops to supercomputers
  - from memory to InfiniBand



Programming Language

- In this class, C is used for explanation
- The MPI standard uses C and Fortran
  - Previously, C++ was used but has been discontinued
- Many libraries on other languages
  - C++: Boost.MPI
  - Python: mpi4py
  - Java: OpenMPI
  - Go: go-mpi
  - Rust: rsmpi
  - etc.

MPI Initialization

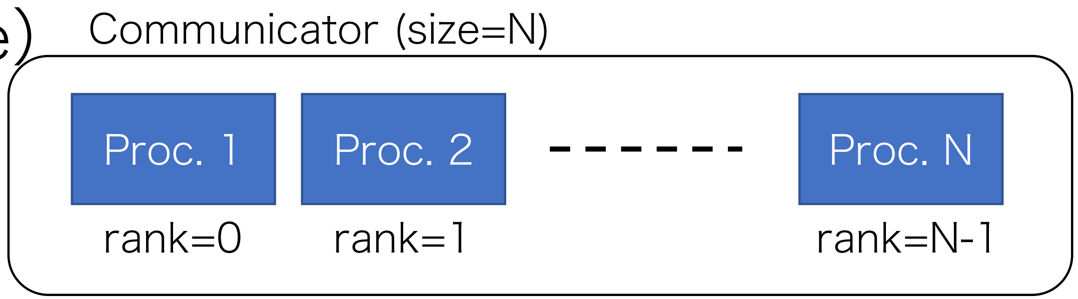
- `MPI_Init(int*, char***)`
  - Initializes the MPI library
    - Must call before other MPI function calls
  - Takes arguments (argc, argv in C) for program
    - to handle options for MPI library
    - プログラムへの引数を解釈して，MPI向け引数を除去するため
- `MPI_Init_thread(int*, char***, int req, int* provided)`
  - MPI_Init for multithread applications (OpenMP, pthread, etc.)
  - req specifies requested level, and provided returns actual level
    - major implementations supports MULTIPLE

MPI_THREAD_SINGLE	same as MPI_Init. Multithread is disallowed.
MPI_THREAD_FUNNELED	Only thread that called MPI_Init_thread can use MPI.
MPI_THREAD_SERIALIZED	Multiple threads can use MPI, however, calling MPI from multiple threads simultaneously is disallowed.
MPI_THREAD_MULTIPLE	Multiple threads can use MPI without any restriction.

MPI Finalization

- `MPI_Finalize()`
  - terminates the MPI library (successfully)
  - Do not call MPI after MPI_Finalize()
  - non-MPI program is allowed after MPI_Finalize()
- `MPI_Abort(MPI_Comm, int)`
  - terminates the MPI library with an error
    - All processes aborts execution
    - big difference from non-MPI functions (exit, abort, etc.)
  - Recovery from error is also supported for fault tolerance

Communicator



- Communicator (`MPI_Comm` type)
  - term representing "communication group" in MPI
- `MPI_COMM_WORLD`
  - All processes join MPI_COMM_WORLD at default
  - Special communicator available at MPI_Init()
- `MPI_Comm_rank(MPI_Comm, int*)`
  - Obtain rank number (0~) of called process
  - rank is unique number in the communicator
- `MPI_Comm_size(MPI_Comm, int*)`
  - Obtain how many processes are in the communicator

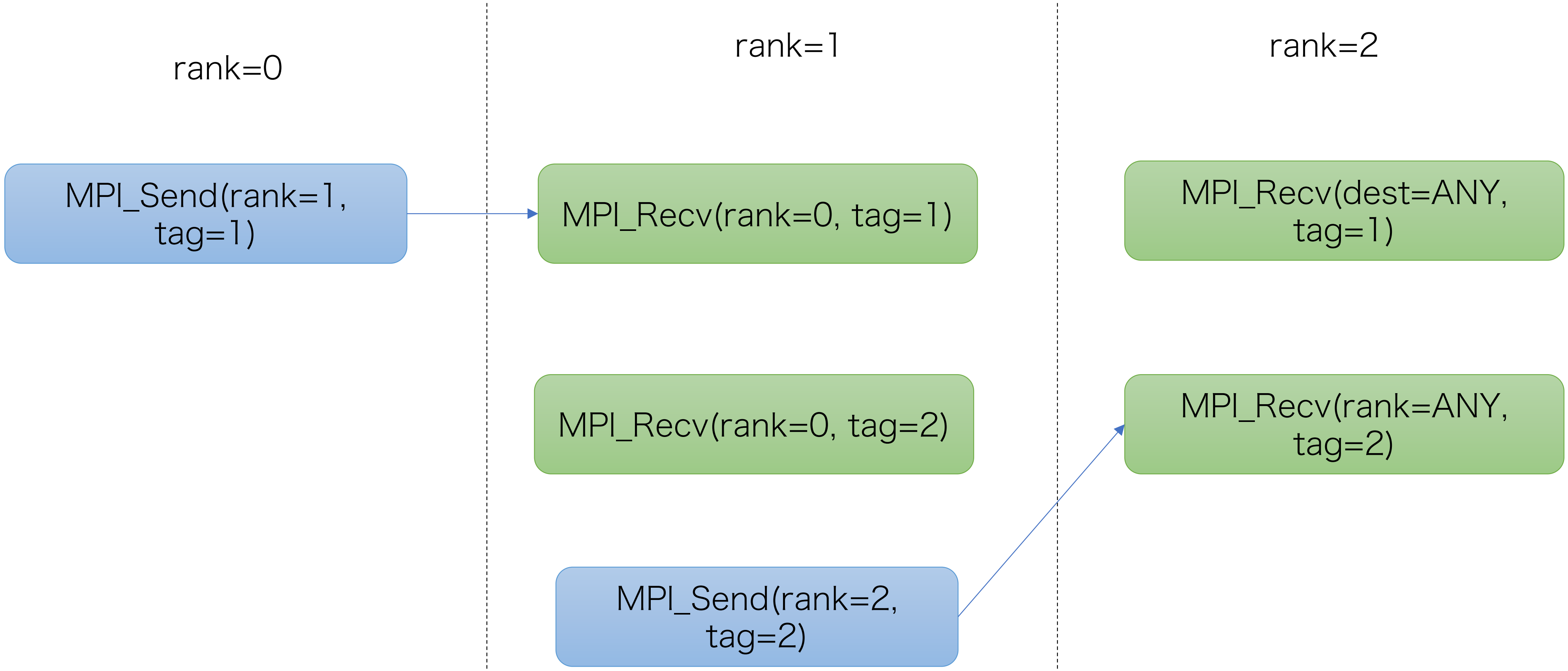
Kind of Comms.

1. point-to-point
   - Communication between proc. A and proc. B
   - Before communication, sender confirms receiver is ready
     - “handshake”
     - like telephone call (ringtone)
2. collective
   - Many procs. join it to accomplish objective of the communication
   - sum of array (reduction), data relocation (gather/scatter), transpose of matrix (Alltoall), synchronize among procs. (barrier), etc.
   - MPI has many kinds of collective comms.
3. one-sided
   - Send data from proc. A to proc. B
   - Sender does not confirm receiver’s status
     - like home delivery service
     - Japanese service does not check we are in home or not.
   - More effective than point-to-point because of no handshake
     - However, program must guarantee OK to be written from other procs

point-to-point

- `MPI_Send(void const* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)`
  - buffer: pointer to data to be sent
  - count: number of data (not byte!)
  - datatype: data type of data
  - dest: destination rank in comm
  - tag: tag for matching
  - comm: Communicator
- `MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)`
  - source: source rank in comm. (any is also supported)
  - status: result of receiving (length, source rank, etc.)

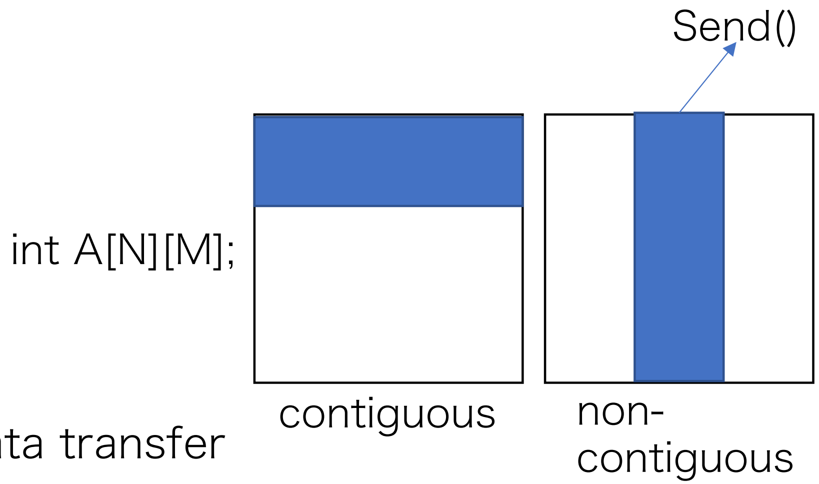
point-to-point



"tag" matches MPI_Send with MPI_Recv.
Data are transferred between MPI_Send and MPI_Recv that are have same tag.

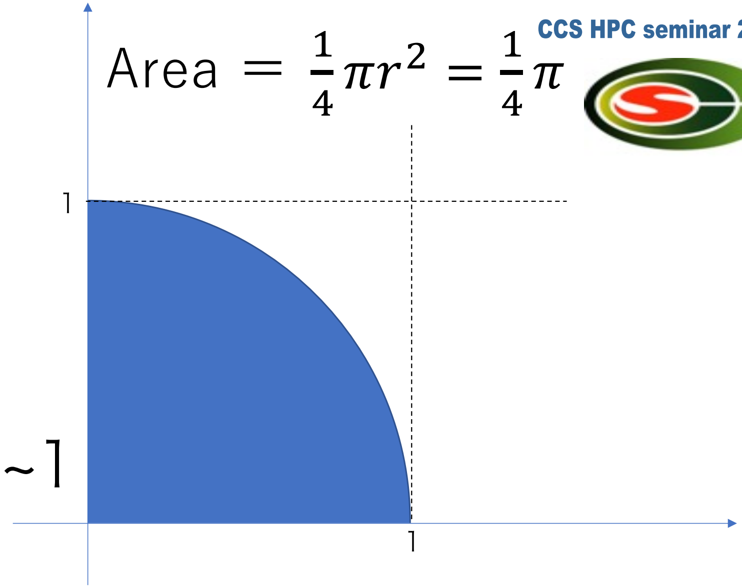
MPI DataType

- DataType
  - MPI_INT, MPI_FLOAT, MPI_DOUBLE, etc.
- Specifies what kind of data to communicate
  - MPI does not use byte in size
    - (# of elements) * (DataType)
    - 100 * MPI_INT
  - MPI is based on array
    - useful for collective communications
- Extended Data Type
  - Program defines new data type
    - combining basic data types into one
    - → for structures and array of structure
  - Non-contiguous data transfer
    - optimization for transfer a part of array
    - allows MPI library to optimize non-contiguous data transfer



Ex: Monte carlo

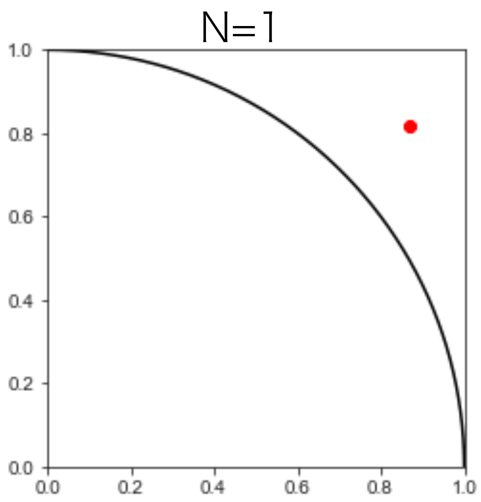
- Based on random numbers
- Consider 1/4 area of a circle (radius=1)
- Plot points randomly in range x=0~1, y=0~1
  - Check a point is inside the circle
  - Is distance from center less than 1?
  - $\sqrt{x^2+y^2} < 1 \iff x^2+y^2 < 1$
- If p points are inside circle, area is approximately $\frac{p}{N}$ and $\pi = 4\frac{p}{N}$.
  - Accuracy depends on N. Larger N is better.
- Very easy to implement using MPI
  - Split work into procs.

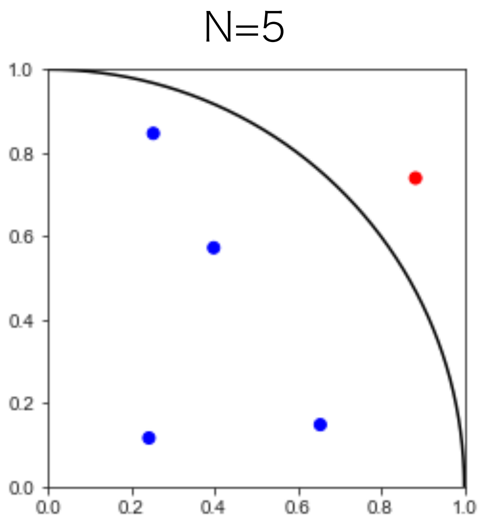


Proc. 1 (N=100)
0 – 99

Proc. 1	Proc. 2	Proc. 3	Proc. 4
0 – 24	25 – 49	50 – 74	75 - 99

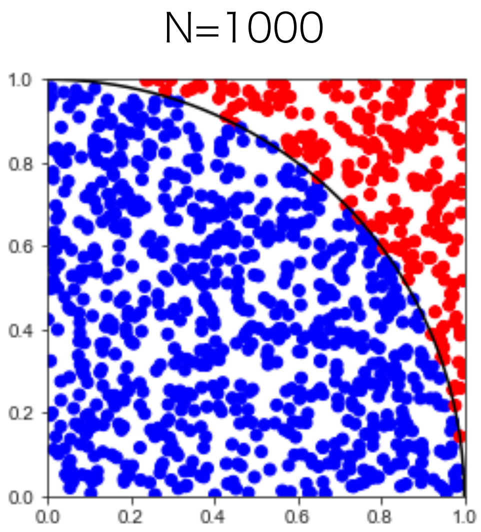
0 / 1 * 4 = 0

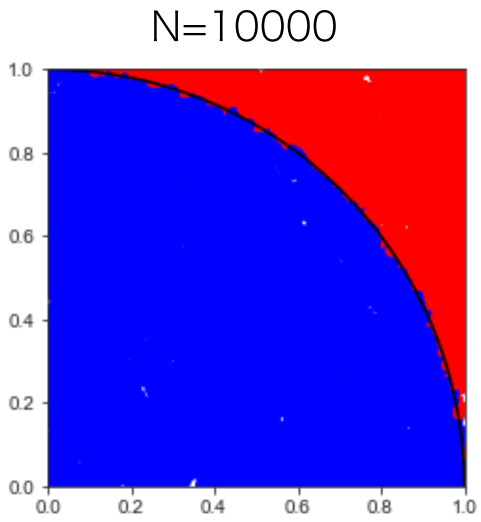




4 / 5 * 4 = 3.2

779 / 1000 * 4 = 3.116





7849 / 10000 * 4 = 3.1396

```c
#include <mpi.h>

int main(int argc, char** argv) {
    MPI_Init(&argc, &argv);

    MPI_Comm_rank(MPI_COMM_WORLD, &mpi_rank);
    MPI_Comm_size(MPI_COMM_WORLD, &mpi_size);
    printf("MPI(rank=%d, size=%d)¥n", mpi_rank, mpi_size);

    unsigned long loop = 1000000000lu;
    int repeat = 10;

    for (int r = 0; r < repeat; r++) {
        compute_main(loop / mpi_size);
    }

    MPI_Finalize();

    return 0;
}
```

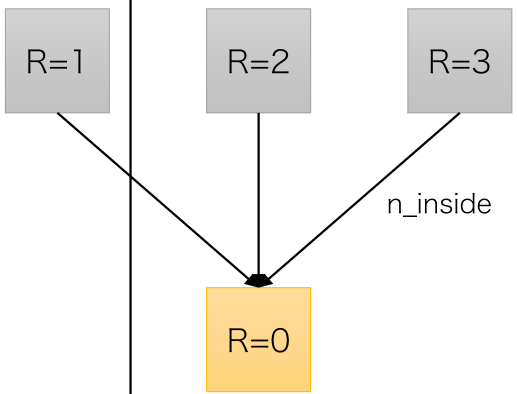
Each process computes (loop/mpi_size) points

```
void compute_main(unsigned long loop) {
    int n_inside = 0;

    for (unsigned long i = 0; i < loop; i++) {
        double x = random01();
        double y = random01();

        if (x * x + y * y < 1.0) {
            n_inside += 1;
        }
    }

    if (mpi_rank == 0) {
        for (int i = 1; i < mpi_size; i++) {
            unsigned long temp;
            MPI_Recv(&temp, 1, MPI_UNSIGNED_LONG, i, 0,
                     MPI_COMM_WORLD, MPI_STATUS_IGNORE);
            n_inside += temp;
        }
    } else
        MPI_Send(&n_inside, 1, MPI_UNSIGNED_LONG, 0, 0, MPI_COMM_WORLD);

    if (mpi_rank == 0)
        printf("result = %.10f¥n", compute_pi(n_inside, mpi_size * loop))
}
```

if (my rank is 0)
Receive other's result (Rank 1~3)

if (my rank is not 0)
send result to rank zero

Rank 0 prints the final result



```
result = 3.2000000000
    PI = 3.1415926535897 ...
  loop = 10
  time = 0.000 sec


result = 2.9600000000
    PI = 3.1415926535897 ...
  loop = 100
  time = 0.000 sec


result = 3.1240000000
    PI = 3.1415926535897 ...
  loop = 1000
  time = 0.000 sec


result = 3.1052000000
    PI = 3.1415926535897 ...
  loop = 10000
  time = 0.000 sec
```

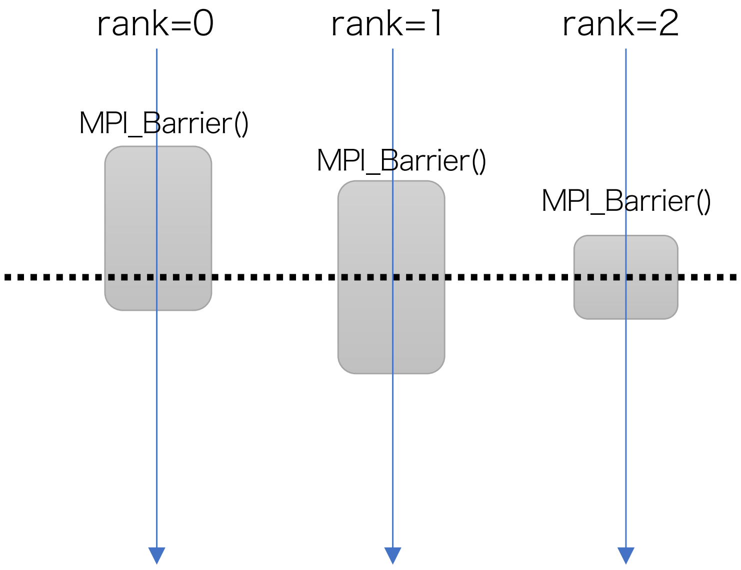
```
result = 3.1516000000
result = 3.1208000000
result = 3.1612000000
result = 3.1268000000
result = 3.1248000000
result = 3.1432000000
result = 3.1468000000
result = 3.1556000000
result = 3.1244000000
result = 3.1244000000
```

Collective

- Collective
  - performs N-to-N communications
  - All procs. in a communicator must be participate
  - Sub-communicators API for partial (not WORLD) collective
- Barrier synchronization, Broadcast, Gather, Scatter, Allgather, Alltoall, Reduce, Allreduce, …
  - I will explain them in following slides
- Why do we use them?
  - Collectives can be composed of sends and recvs
  - However, MPI library optimizes them in terms of algorithm and communication pattern

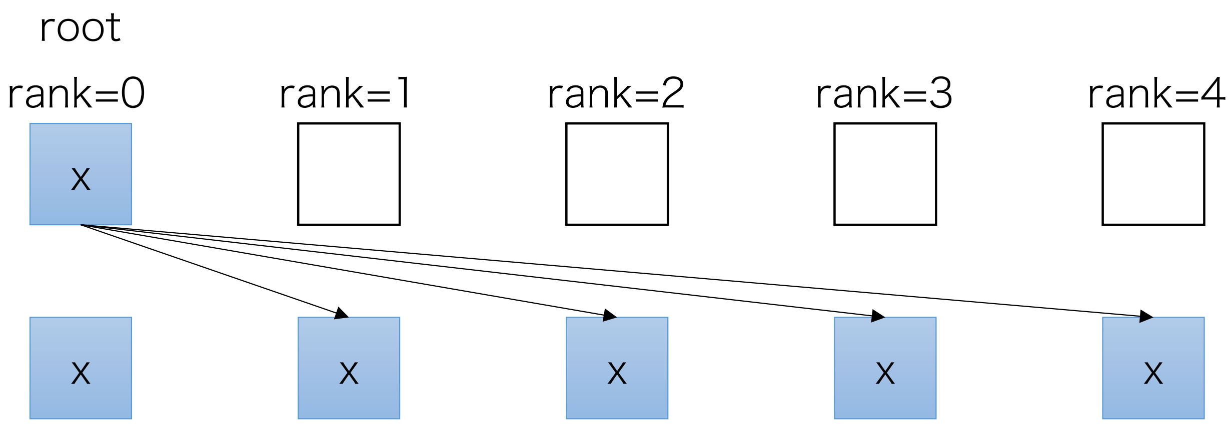
MPI_Barrier

- `MPI_Barrier(MPI_Comm comm)`
  - Synchronize among procs. in comm
  - MPI_Barrier guarantees all procs. have arrived the call
    - exit timing from MPI_Barrier may not be same across pros.
    - Due to delay of network and noise of execusion



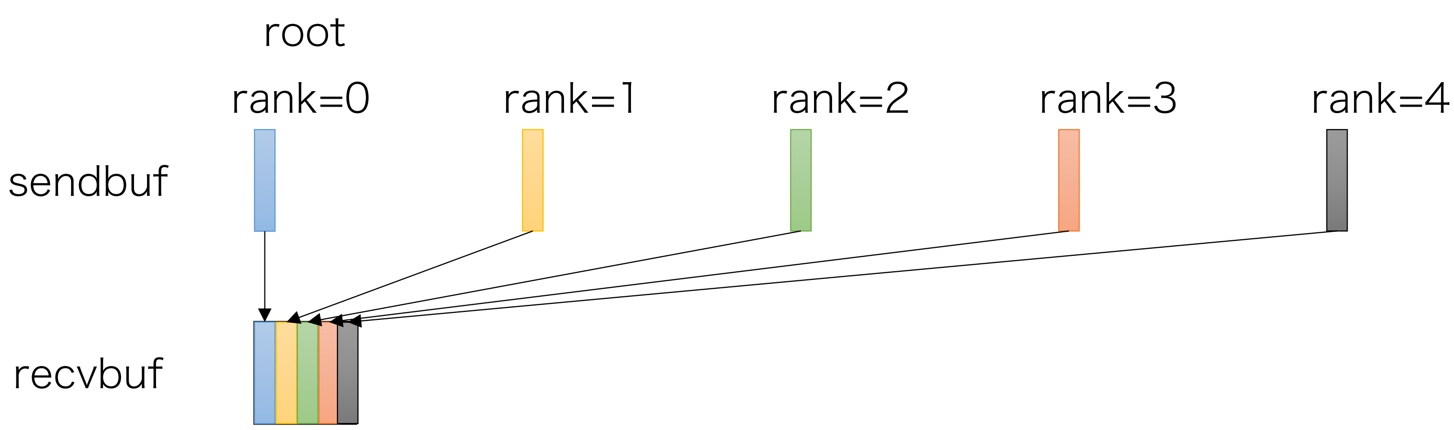
MPI_Bcast

- `MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)`
  - Data in buffer at rank=root are sent to other (rank != root) procs.
  - Bcast = Broadcast
  - Behavior is calling MPI_Send()s for each proc.
  - MPI_Bcast allows library to optimize
  - For example:
    - tree-based algorithms (O(log N))
    - Network supported Bcast



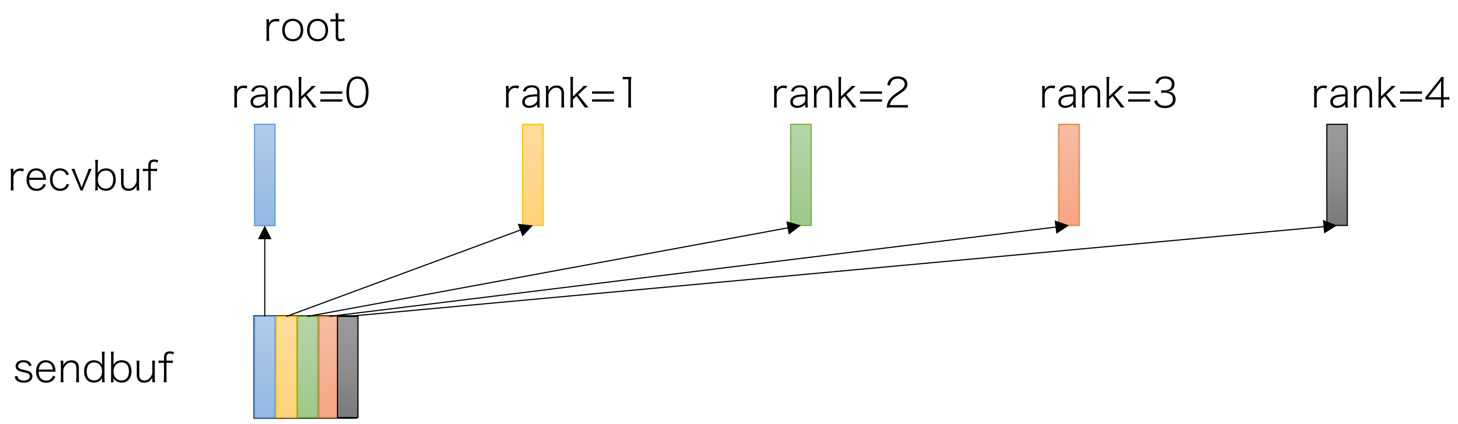
MPI_Gather

- `MPI_Gather(void const* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - Gathering data
    - data in sendbuf at all procs.
    - →
    - recvbuf at root proc.



MPI_Scatter

- `MPI_Gather(void const* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)`
  - Scattering data
    - data in sendbuf at root proc.
    - →
    - recvbuf at all procs.
  - Reverse of MPI_Gather



MPI_Alltoall

- `MPI_Alltoall(void const* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)`
  - Performs to transpose a matrix
  - all ranks must communicate with all ranks
    - So, this is called as “alltoall”
  - Optimizing alltoall is very difficult
    - optimization depends on topology of network

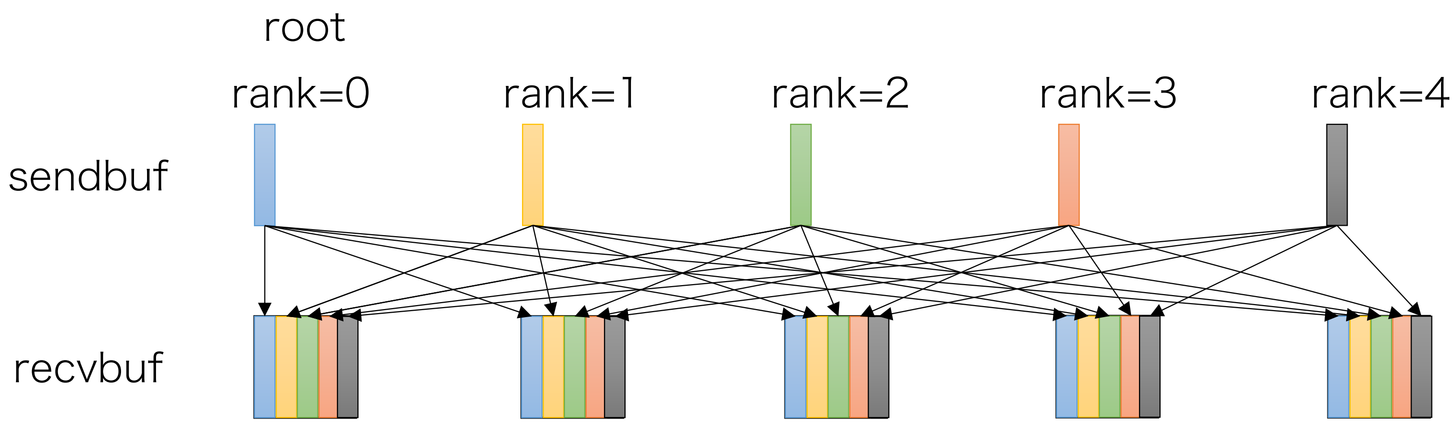
rank=0	1	2	3	4
rank=1	5	6	7	8
rank=2	9	10	11	12
rank-3	13	14	15	16

Alltoall

1	5	9	13
2	6	10	14
3	7	11	15
4	8	12	16

MPI_Allgather

- `MPI_Allgather(void const* sendbuf, int sendcount, MPI_Datatype datatype, void* recvbuf, int recvcount, MPI_Datatype datatype, MPI_Comm comm)`
  - Behavior is Gather() then Bcast()
    - Typical usage is to gather and to share the result of computation
  - Optimized algorithm is used better than just calling gather and bcast.



MPI_Reduce / Allreduce

- `MPI_Reduce(void const* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)`
  - Compute reduction of sendbuf at all procs using op
    - $\mathrm{recvbuf}_{\mathrm{root}} = \mathrm{sendbuf}_0 \ \mathit{op} \ \mathrm{sendbuf}_1 \ \mathit{op} \ \ldots \ \mathit{op} \ \mathrm{sendbuf}_N$
  - Behavior is gathering sendbuf to root and applying op them at root
    - MPI uses optimized algorithm, thus this will be O(log N)
  - Typical usage is backpropagation in AI learning
- Allreduce = Reduce + Bcast
  - Allreduce shares the result of Reduce on all procs.

		root rank=0		rank=1		rank=2		rank=3		rank=4
15	=	1	+	2	+	3	+	4	+	5

op=MPI_SUMの場合

Communication Model (1/3)

- How MPI guarantee communication is competed?
- When sender calls MPI_Send(), data is arrived at receiver?
  - You may think "MPI_Send sends data, so yes!"
  - → Not guaranteed
- This is very important to write correct program
  - Generally, we don't need to care about it
    - Understanding communication model helps you to optimize your program

Communication Model (2/3)

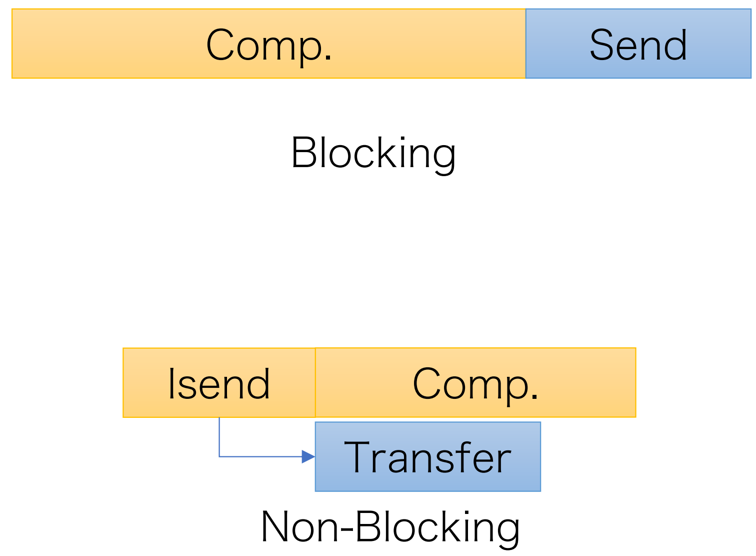
- When MPI_Send() returns,
  - it does not guarantee that data is arrived at receiver
  - e.g., for small messages, it just writes data to buffer at sender
    - Nothing happens on network
  - e.g., data are just arrived at receiver, but still in buffer
    - MPI_Recv() is under processing
- MPI does not guarantee completion of communication
  - If you want to know, need to check yourself
- Completion of MPI communication function
  - → The buffer given to the function is ready to use
  - MPI_Send: OK to modify the buffer for further comm./comp.
  - MPI_Recv: OK to read from the buffer

Communication Model (3/3)

- Most MPI implementations use two kind of protocols for point-to-point
- Eager
  - for short messages
  - Sender writes (small) eager buffer at receiver
  - MPI_Recv can be delayed
- Handshake
  - for large messages
  - MPI_Send and MPI_Recv wait each other (handshake)
  - and then, transfer the data
    - Eager buffer is not used because MPI_Recv know receive buffer given by application

Non-blocking

- Non-blocking functions
  - Immediately returns when preparation is completed
  - Do not write/read the buffer after returns
  - This allows program to overlap computation with communication
- MPI_Request
  - ticket for non-blocking communications
  - We use this to check if finished later
- MPI_Wait/Waitall/Waitany
  - Wait for completion of given requests
- MPI_Waitall/Waitany
  - Wait for multiple requests (as array)
  - Waitall: all of requests
  - Waitany: one of requests at least



Non-blocking Point-to-Point

- Add “I” to non-blocking functions
  - MPI_Send → MPI_Isend
  - MPI_Recv → MPI_Irecv
  - “I” for Immediate or Incomplete
  - MPI_Request is added in arguments
- `MPI_Isend(void const* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm, MPI_Request* request)`
- `MPI_Irecv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Request* request)`

MPI_Isend, Irecv

```
MPI_Request r[2];
MPI_Status status[2];

MPI_Isend(sendbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &r[0]);
MPI_Irecv(recvbuf, n, MPI_INT, 1, 0, MPI_COMM_WORLD, &r[1]);

MPI_Wait(&r[0], &s[0]);
MPI_Wait(&r[1], &s[1]);

do_computation();

MPI_Waitall(2, r, status);
```

Non-blocking Collective

- MPI-3 introduces non-blocking version of collectives
- Skip in this class due to many
  - Purpose is same as point-to-point
  - Usage is same manner as point-to-point, too
- Ibarrier, Ibcast, Igather, Iscatter, Ireduce, ..., etc.
  - Add "I" to non-blocking functions
  - MPI_Request is returned to check later
- `MPI_Ibarrier(MPI_Comm comm, MPI_Request* request)`
- `MPI_Ireduce(..., MPI_Comm, MPI_Request*)`
- `etc.`

Ex: Laplace Equation

$$\frac{\partial^2 f}{\partial x^2}+\frac{\partial^2 f}{\partial y^2} = 0$$

discretization

*$f(-1,0)$ means $f(x-\Delta x, y)$

$$f(0,-1) + f(-1,0) + f(1,0) + f(0,1) - 4f(0,0) = 0$$

- Explicit method of Laplace (2D) equation

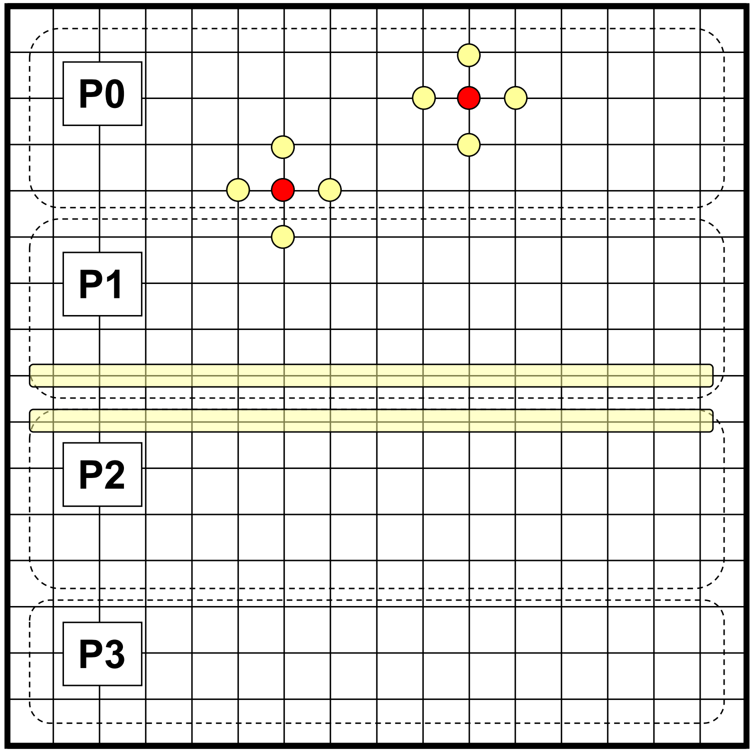
$$f(0,0)_{new} = \frac{1}{4}\left(f_{old}(0,-1) + f_{old}(-1,0) + f_{old}(1,0) + f_{old}(0,1)\right)$$

  - Update values with average of neighbor 4 points
    - Use two arrays(old, new)
    - Update “new” with values in “old”
    - Compute residual to check convergence
    - Copy “new” to “old”
  - Typical domain decomposition
    - Split large computation into small computation on multiple processes

Ex: Laplace Equation

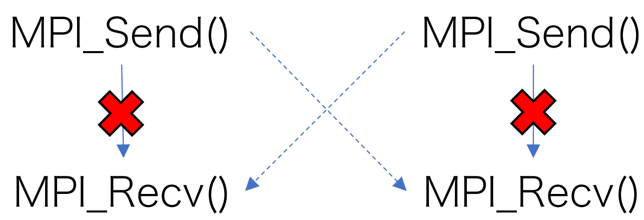
```
u[x][y] = 0.25 * (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])
```

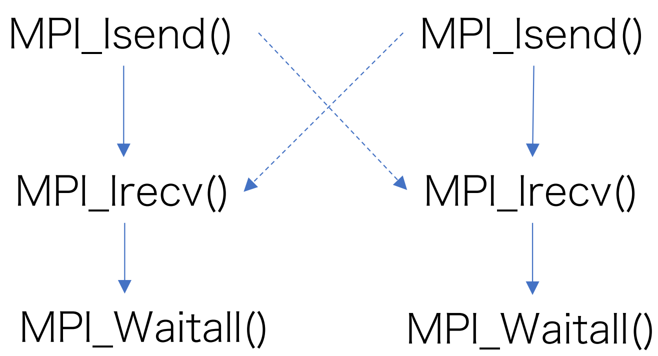
- Decompose 2D domain with 1D block
- Some area requires data on next process
  - if y-1 or y+1 are out of my domain
  - Yellow area in the right figure
  - We call the area as “boundary”
- To obtain boundary data on next process, we use MPI communication
  - P1 sends boundary to P0 and P2
  - P2 sends boundary to P1 and P3



Exchange Data

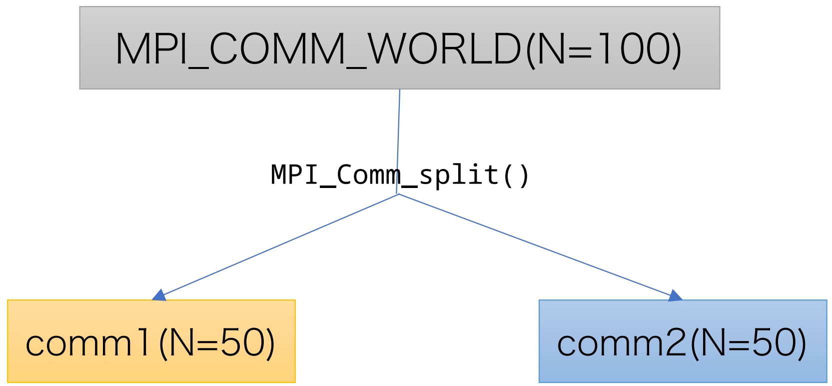
- Simple way
  - MPI_Send() & MPI_Recv()
    - Possibility of blocking by MPI_Send
    - MPI_Recv may never be executed
  - If and only if MPI_Send is eager, MPI_Recv can be executed
    - we cannot guarantee this
    - depending on implementation and size of message
- How to solve
  - Use MPI_Sendrecv()
    - MPI do send and receive at same time
  - Use Non-blocking comms.
    - MPI_Isend never blocks
    - We can handle multiple comms. simultaneously in any order





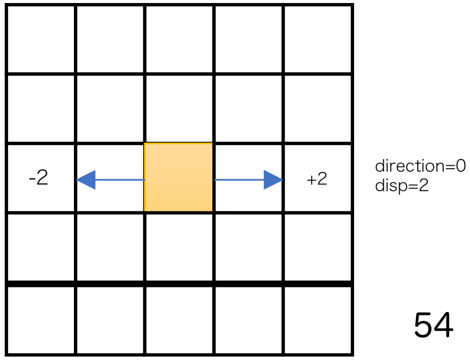
Communicator

- Communicator
  - Group of processes
  - Target of communication
- Program can create communicators as needed
- Typical Usage:
  - Reorder processes (rank number)
  - To split (MPI_Comm_split())



Cartesian topology

- API for topology of Cartesian coordinate
- `MPI_Cart_create(MPI_Comm comm_old, int ndims, int const* dims, int const* periods, int reorder, MPI_Comm* comm_cart)`
  - Create new MPI_Comm from *comm_old*
    - Split *comm_old* into *ndims* dimensions
  - *dims* and *periods* are array representing size of dimensions and boundary type
  - If *reorder* is true, order of rank will be reordered.
- `MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int* rank_source, int* rank_dest)`
  - Obtain neighbor rank on Cartesian topology
    - *direction* is dimension to shift (0 to *ndims-1*)
    - *disp* is how many ranks to shift
  - The results are stored into *rank_source* and *rank_dest*
    - If result is out of domain, MPI_PROC_NULL will be returned

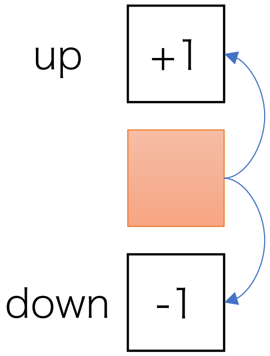


```c
/*
** Laplace equation with explicit method
**/

#include <math.h>
#include <mpi.h>
#include <stdio.h>
#include <stdlib.h>

/* square region */
#define XSIZE 256
#define YSIZE 256
#define PI 3.1415927
#define NITER 10000
double u[XSIZE + 2][YSIZE + 2], uu[XSIZE + 2][YSIZE + 2];
double time1, time2;
void lap_solve(MPI_Comm);
int myid, numprocs;
int namelen;
char processor_name[MPI_MAX_PROCESSOR_NAME];
int xsize;
```

Don’t forget to include mpi.h

u and uu are array for computation
u is old and uu is new.

```c
void initialize() {
    int x, y;
    /* Compute initial values */
    for (x = 1; x < XSIZE + 1; x++)
        for (y = 1; y < YSIZE + 1; y++)
            u[x][y] = sin((x - 1.0) / XSIZE * PI) + cos((y - 1.0) / YSIZE * PI);

    /* zero fill on boundaries */
    for (x = 0; x < XSIZE + 2; x++) {
        u[x][0] = u[x][YSIZE + 1] = 0.0;
        uu[x][0] = uu[x][YSIZE + 1] = 0.0;
    }

    for (y = 0; y < YSIZE + 2; y++) {
        u[0][y] = u[XSIZE + 1][y] = 0.0;
        uu[0][y] = uu[XSIZE + 1][y] = 0.0;
    }
}
```




```c
#define TAG_1 100
#define TAG_2 101
#ifndef FALSE
#define FALSE 0
#endif

void lap_solve(MPI_Comm comm) {
    int x, y, k;
    double sum;
    double t_sum;
    int x_start, x_end;
    MPI_Request req1, req2;
    MPI_Status status1, status2;
    MPI_Comm comm1d;
    int down, up;
    int periods[1] = {FALSE};

    /*
     * Create one dimensional cartesian topology with
     * nonperiodical boundary
     */
    MPI_Cart_create(comm, 1, &numprocs, periods, FALSE, &comm1d);
    /* calculate process ranks for 'down' and 'up' */
    MPI_Cart_shift(comm1d, 0, 1, &down, &up);
    x_start = 1 + xsize * myid;
    x_end = 1 + xsize * (myid + 1);
```

Create 1D cartesian topology.
Boundary is not periordical.
ndims=1, dims={numprocs}

My compute range
x_start <= x < x_end

Get ranks of up and down process.
MPI_PROC_NULL for boundaries.

```
void lap_solve(MPI_Comm comm) {
  ....
    for (k = 0; k < NITER; k++) {
        /* old <- new */
        for (x = x_start; x < x_end; x++)
            for (y = 1; y < YSIZE + 1; y++) uu[x][y] = u[x][y];

        /* recv from down */
        MPI_Irecv(&uu[x_start - 1][1], YSIZE, MPI_DOUBLE, down, TAG_1, comm1d, &req1);
        /* recv from up */
        MPI_Irecv(&uu[x_end][1], YSIZE, MPI_DOUBLE, up, TAG_2, comm1d, &req2);
        /* send to down */
        MPI_Send(&u[x_start][1], YSIZE, MPI_DOUBLE, down, TAG_2, comm1d);
        /* send to up */
        MPI_Send(&u[x_end - 1][1], YSIZE, MPI_DOUBLE, up, TAG_1, comm1d);
        MPI_Wait(&req1, &status1);
        MPI_Wait(&req2, &status2);

        /* update */
        for (x = x_start; x < x_end; x++)
            for (y = 1; y < YSIZE + 1; y++)
                u[x][y] = .25 * (uu[x - 1][y] + uu[x + 1][y] + uu[x][y - 1] + uu[x][y + 1]);
    }
```

Exchange boundary data

```c
void lap_solve(MPI_Comm comm) {
   ....
    /* check sum */
    sum = 0.0;
    for (x = x_start; x < x_end; x++)
        for (y = 1; y < YSIZE + 1; y++) sum += uu[x][y] - u[x][y];

    MPI_Reduce(&sum, &t_sum, 1, MPI_DOUBLE, MPI_SUM, 0, comm1d);

    if (myid == 0) {
        printf("sum = %g¥n", t_sum);
    }

    MPI_Comm_free(&comm1d);
}
```

Compute residual
Use MPI_Reduce for summation

MPI_Init

Initialize

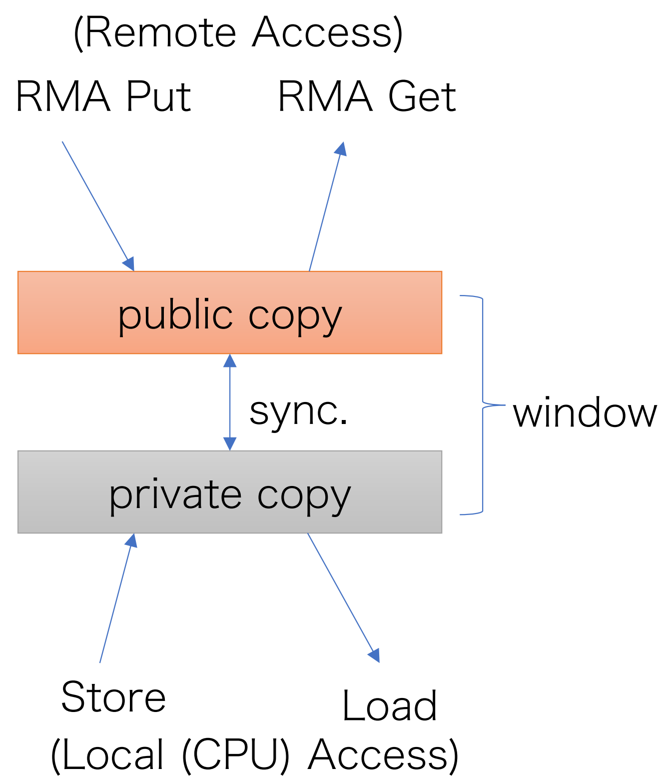
Main part

MPI_Finalize

```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name, &namelen);
    fprintf(stderr, "Process %d on %s¥n", myid, processor_name);

    xsize = XSIZE / numprocs;
    if ((XSIZE % numprocs) != 0)
        MPI_Abort(MPI_COMM_WORLD, 1);

    initialize();

    MPI_Barrier(MPI_COMM_WORLD);
    time1 = MPI_Wtime();
    lap_solve(MPI_COMM_WORLD);
    MPI_Barrier(MPI_COMM_WORLD);
    time2 = MPI_Wtime();

    if (myid == 0) {
        printf("time = %g¥n", time2 - time1);
    }

    MPI_Finalize();
    return (0);
}
```

One-Sided

- Brief overview of one-sided comm.
- "Window" object is used to represent memory region for one-sided communication
  - public copy & private copy
  - RMA separate memory model
  - Synchronization is required to match data between public and private
- RMA unified memory model (since MPI-3)
  - public and private are same (not-separate)
  - If network supports DMA and MPI supports unified, this model is very efficient



Report (MPI)

- Improve Laplace program shown in this class.
  - Report must contain
    - Program code
    - Description of improvement
      - Where and how did you modify
      - What is improved and how is it improved.
    - Output of program
  - Improve must relate to MPI
- Hints, but not limited to
  - Using OpenMP for hybrid parallelization
  - Use one-sided communication
  - 2D domain decomposition instead of 1D