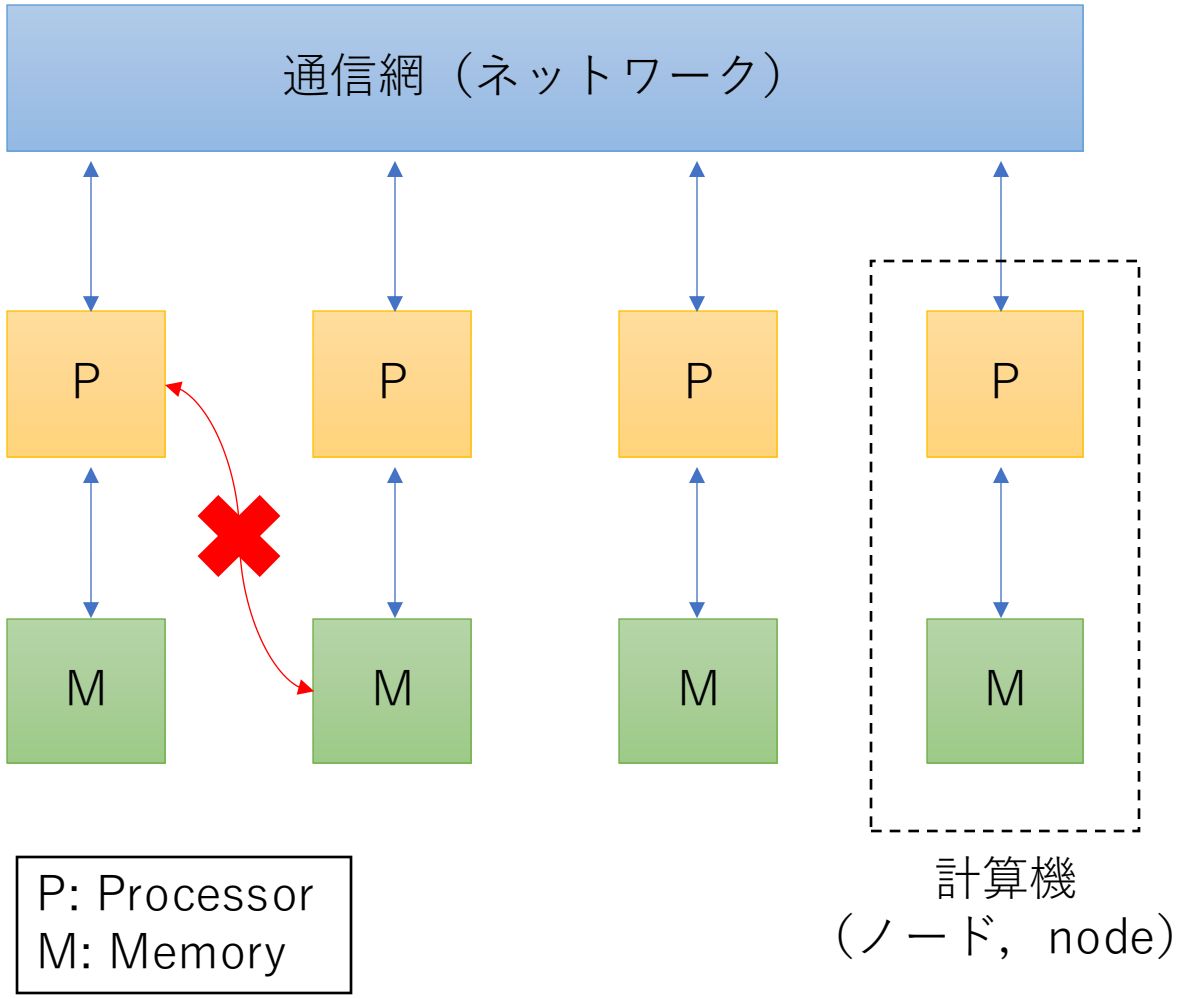
# 筑波大学 計算科学研究センター CCS HPCセミナー 「MPI」

藤田 典久

筑波大学 計算科学研究センター

# 分散メモリシステム

- 計算機
  - 一般的には「ノード」と呼ばれる
- 通信網
  - 非常に高性能なネットワーク
  - スパコン向け専用設計のネットワークも多い
- 分散メモリ
  - 計算機ごとに独立したメモリ
  - 異なるノードにあるメモリは自由に触れない（OpenMPとは違う）



# 並列計算に用いられるネットワーク

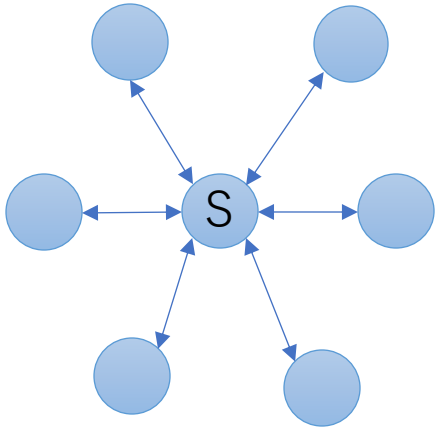
- 並列計算ではネットワークの性能がとても重要となる
  - 複数の計算機で1つの問題を扱うため，高頻度な通信が行われる
- 通信は計算の本質的な要素ではない
  - したがって，可能な限り，通信にかかる時間を短くしたい
  - 並列計算における通信最適化の手法はよく研究されているが，そもそもの通信性能が低いと限界がある
- そのため，非常に高性能なネットワークが用いられることが多い
  - →Ethernet，InfiniBand
  - ただし，高速な通信設備はコストがかかる

# ネットワークの構成要素

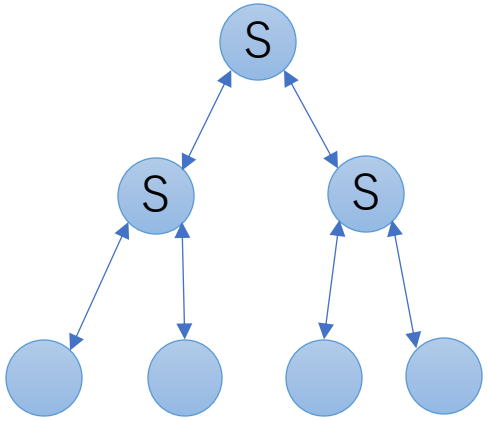
- ノード
  - ネットワークを構成する計算機
- 通信ケーブル
  - ノードーノード間，ノードースイッチ間を接続する
  - ケーブルによって最大通信速度や通信距離が異なる
    - 一般的に，高速な長距離通信は困難．→ケーブルなどが高価
  - 銅ケーブル：安価だが，距離を伸ばすことが難しい
  - 光ケーブル：通信距離を長くできるが，銅と比べると高価
- スイッチ
  - 通信を中継する機器．複数のノードでネットワークを構成するときに使う．
  - 数十台用から，～1000台用まで様々な製品が存在する
  - 構築するネットワークの規模によって選択する

# トポロジー

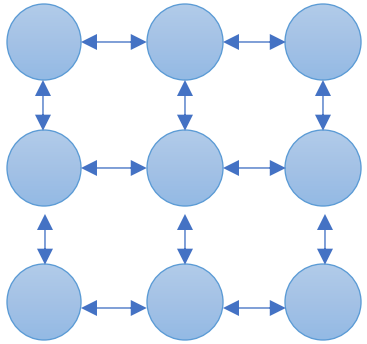
- ネットワークの接続を抽象化したものをトポロジーと呼ぶ
- スイッチを介したツリー型やスター型がよく用いられる
- 並列計算に特化したネットワークでは，メッシュ型が使われることもある




スター型




ツリー型



メッシュ型

# Ethernet

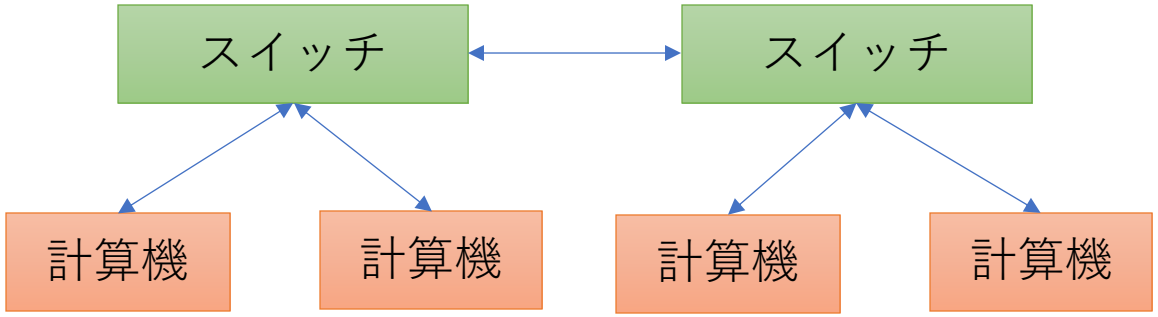
- Ethernet（イーサネット）
  - 最も広く用いられている有線通信規格
  - いわゆる「LAN」や「LANケーブル」などはEthernet規格の一部を指す
  - 1Gbpsの速度をもつ Gigabit Ethernet が有名
- 古くからある規格なため，様々な仕様が混在している
  - 速度：1Mbps ~ 100Gbps
  - 距離：100m ~ 数十km
- 高速な規格は並列計算に用いられることも多い


Ethernetスイッチとケーブル

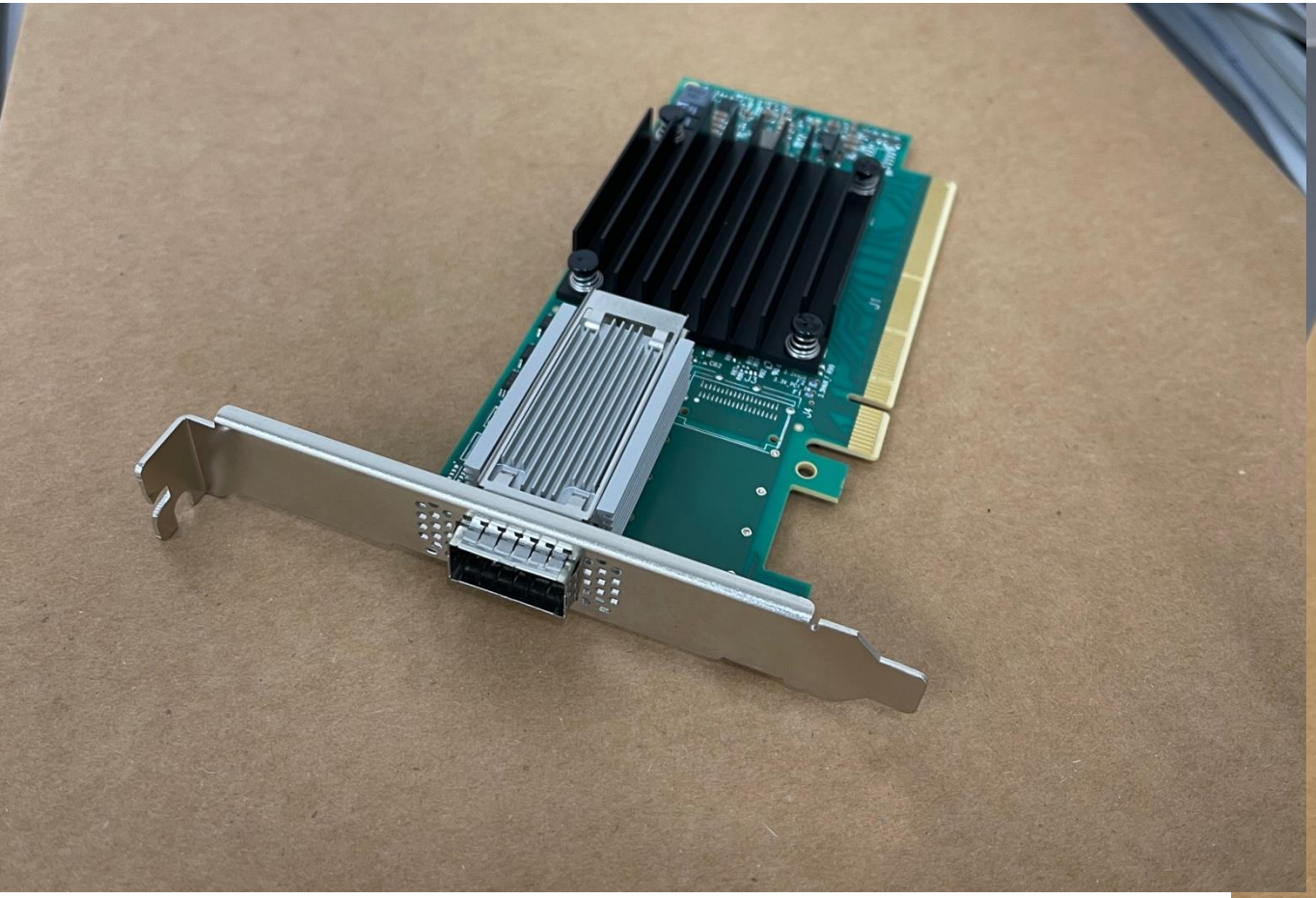

壁に設置されているLANポート

# InfiniBand

- Infiniband（インフィニバンド）
  - 高性能な汎用通信網の規格の一つ
  - NVIDIA社（旧Mellanox社）の製品が有名
  - 小規模な研究室レベル（一桁ノード）から，
    大規模なスーパーコンピュータレベル（1万ノードクラス）まで
    柔軟なネットワーク構築ができる
  - 200 Gbits/sec (Gbps) の速度まで製品があり，今後400Gbps以上のさらなる高速化が予定されている

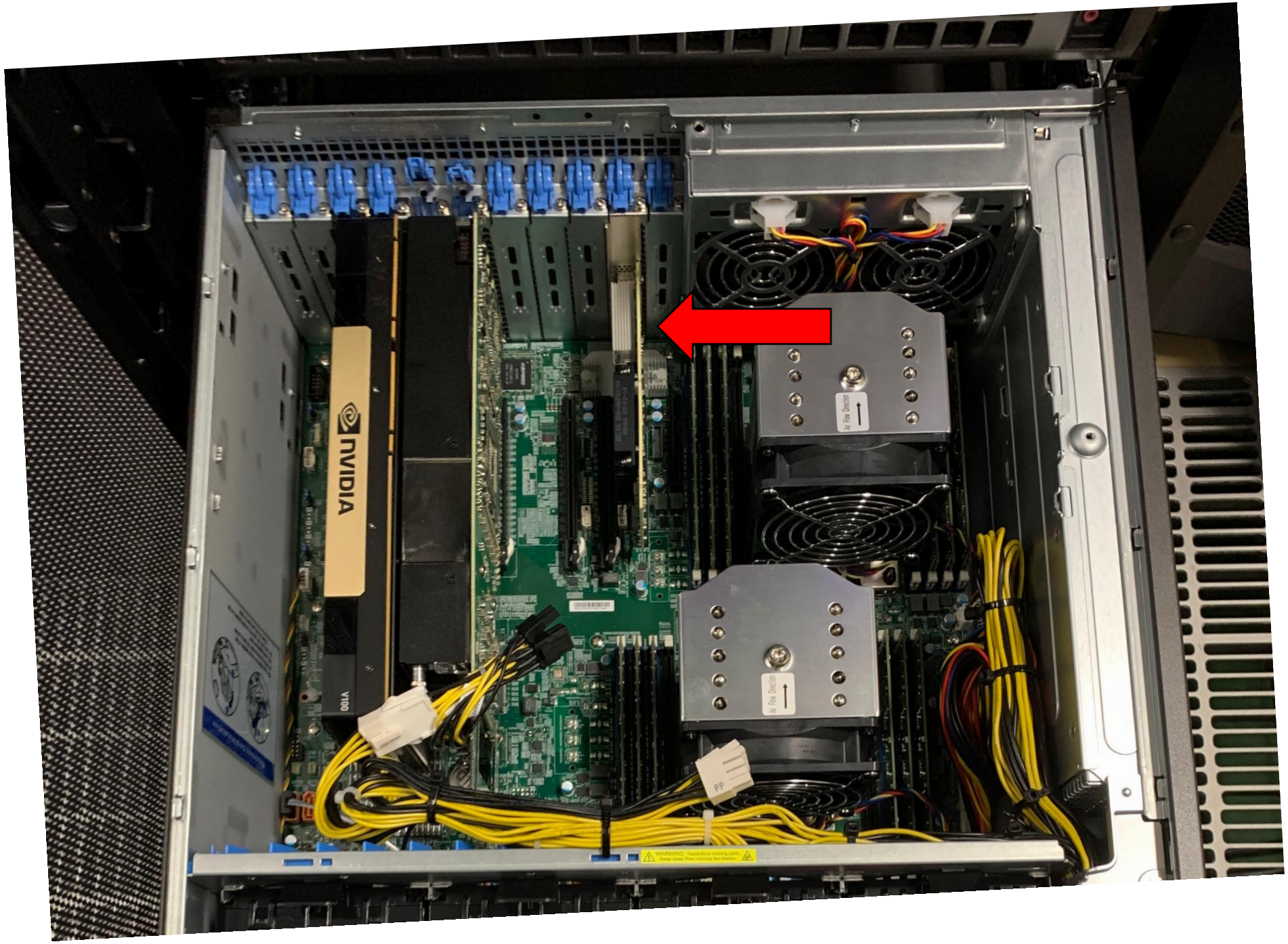



InfiniBandネットワークの例.
大規模なネットワークでは
複数のスイッチを多段で使う場合もある.



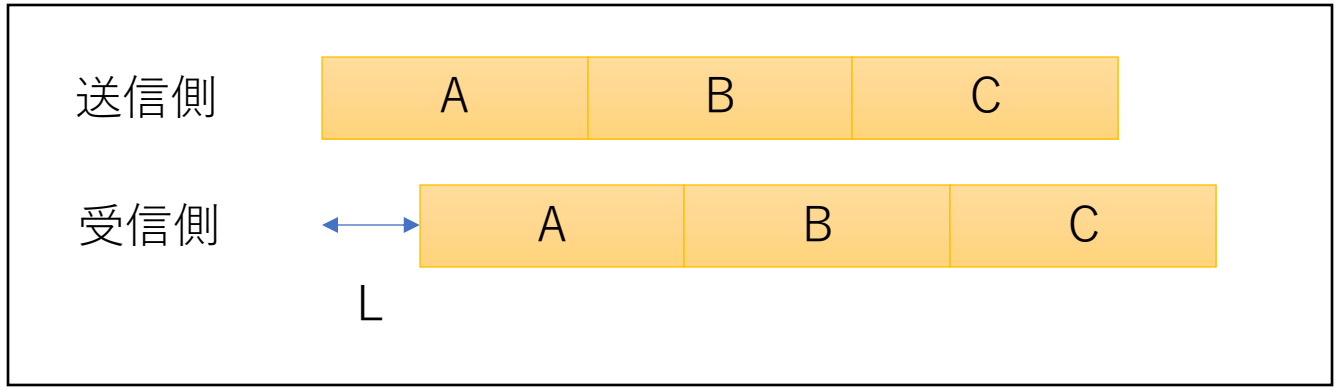
Inifiniband HCA (Host Card Adaptor)

光ケーブル (5m)


NVIDIA
V100
Air Flow Direction
Air Flow Direction
PP

# 通信の帯域とレイテンシ

- 帯域 (Byte/s)
  - 1秒あたりの通信できる通信量
- レイテンシ (s)
  - 送信側が送信して，受信側に届くまでの時間．通信遅延
- 帯域をB[B/s]，レイテンシをL[s]とすると，
  N[B]のデータを送るのにかかる時間は L + N/B [s]
  - ただし，連続して複数の通信を行う場合，Lがかかるのは最初の一回のみ

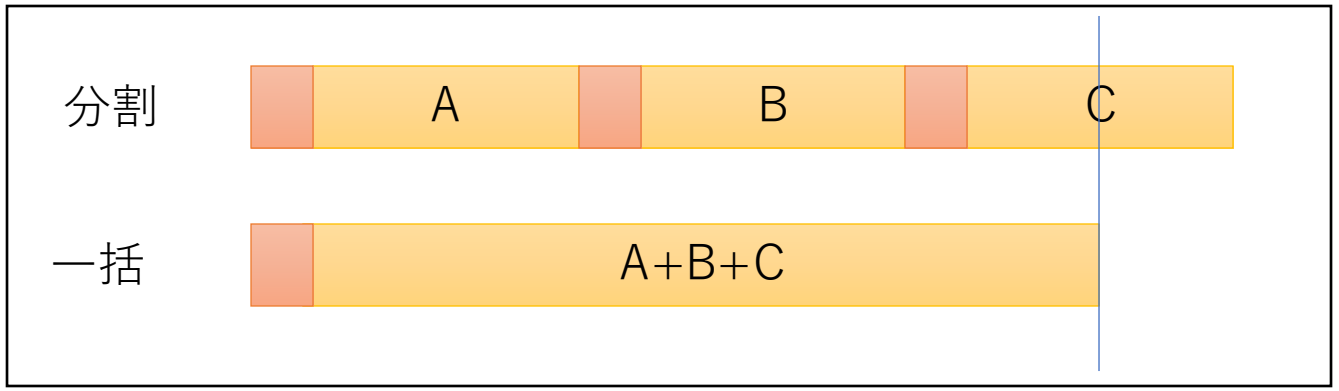


# 通信の帯域とレイテンシ

- 例題
  - 1GBのデータを送るのに10秒かかった．この通信路の実効帯域を求めよ
    - 1[GB] / 10[s] = 0.1[GB/s]
  - 100MBのデータを5MB/sの速度で送信すると何秒かかるか
    - 100[MB] / 5[MB/s] = 20[s]
  - 2点間で1バイトのデータを往復するのに100msかかった．通信遅延を求めよ．ただし，1バイトのデータを送信するのにかかる時間は無視する
    - 100[ms] / 2 = 50[ms]

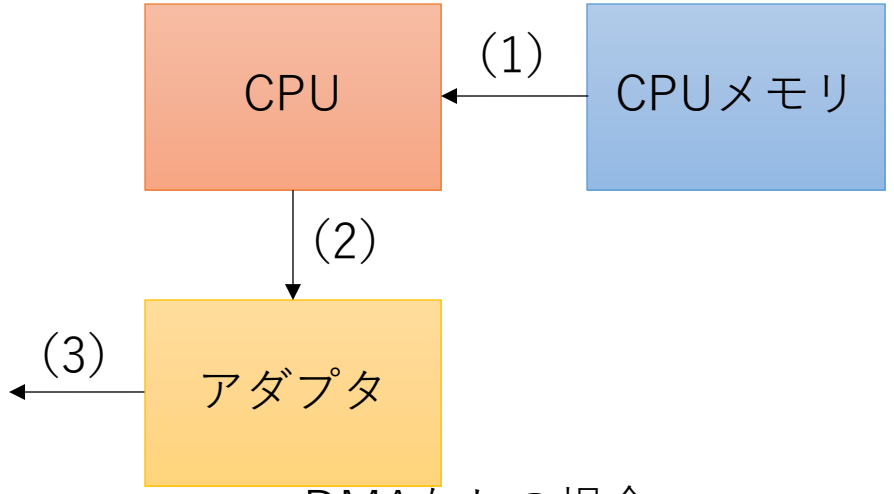
# 通信のオーバーヘッド

- 一般的に通信を行う際には，通信準備や制御のオーバーヘッドが発生する
  - 短い通信でも長い通信でも，このコストは変わらない
- 通信はできるだけまとめて行い，回数を減らしたほうが良い
  - A,B,Cの3データを送る場合，A+B+Cを1回で通信した方が，バラバラに3回通信するよりもオーバーヘッドが削減できる

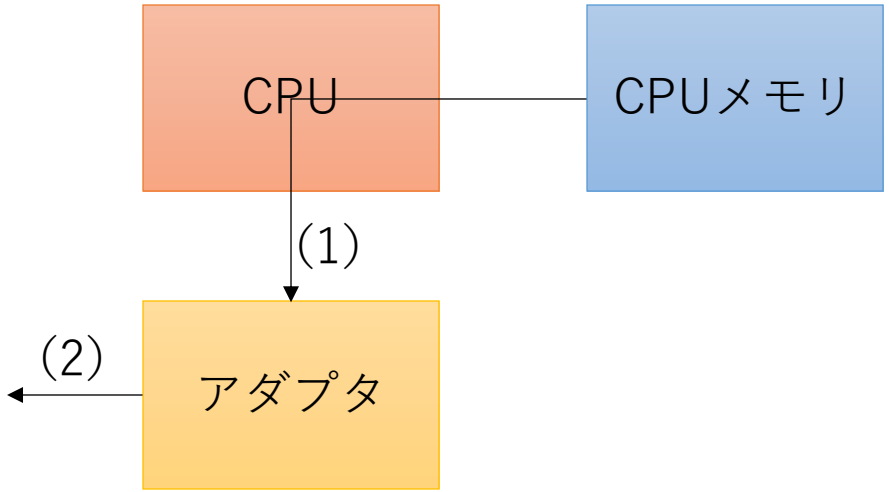


# Direct Memory Access (DMA)

- 高性能なネットワークでは，通信アダプタがCPUメモリに直接アクセスできるものがある
  - このように，外部デバイスがCPUのメモリに直接アクセスすることを，Direct Memory Access (DMA)と呼ぶ．
- データ転送中，CPUの介入が不要なため高性能
  - CPUが自由になるため，別の計算を行える




DMAなしの場合




DMAありの場合

# 非同期通信

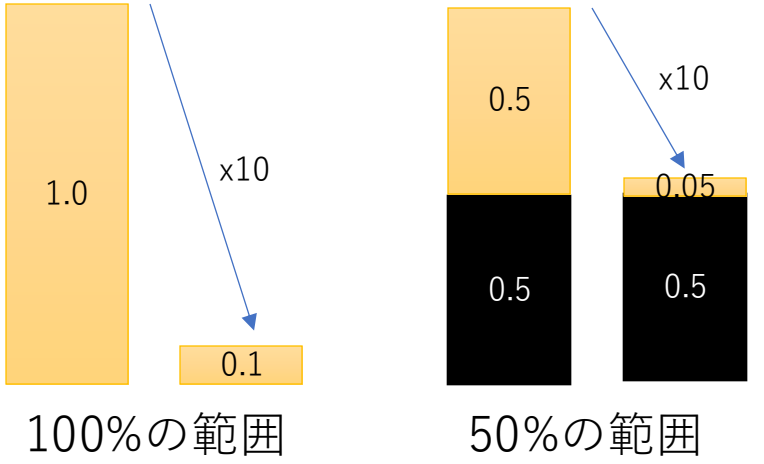
- CPUにとって通信はかなり時間のかかる作業である
  - 通信が終わるまで，CPUが何もしないのは無駄が大きい
- 計算を行っている間に，同時に通信を行う
  - このような通信形態を非同期通信と呼ぶ
  - 前述したDMAが使える場合，CPUにかかる負荷は無視できる
  - 理想的な状態では，通信にかかる時間を完全に隠蔽できる
- 効果は大きいが，プログラミングが複雑になる
  - 非同期通信を行うために，計算順序の変更やプログラム構造の変更が必要になる場合がある
  - 通信するための計算を先に行うなどの工夫が必要


同期通信の場合


非同期通信の場合

# アムダールの法則

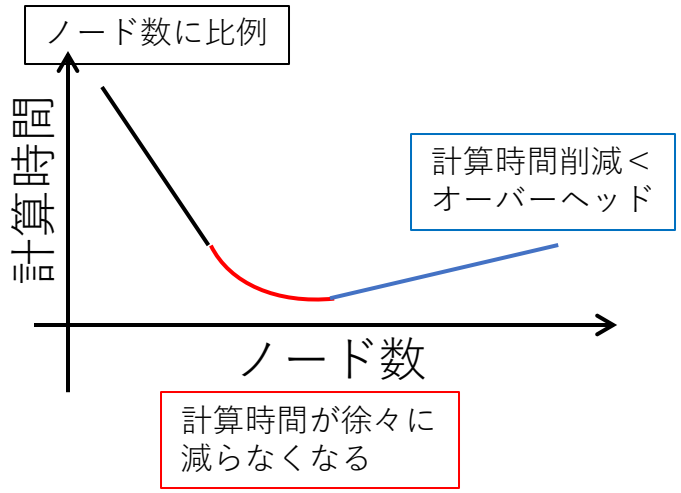
- プログラム全体の性能向上は，並列化できない部分に律速される
  - 100%の範囲を10倍高速化できた場合
    - 1 / (1 / 10) = 10
  - 90%
    - 1 / (0.1 + 0.9 / 10) = 5.26
  - 80%
    - 1 / (0.2 + 0.8 / 10) = 3.57
  - 50%
    - 1 / (0.5 + 0.5 / 10) = 1.82




- プログラムの90%を10倍高速化しても，全体では5.2倍にしかならない
- プログラムの半分を10倍に高速化しても，全体では1.8倍にしかならない
- 無限大に高速化したとしても・・・
  - 99%→100倍，90%→10倍，50%→2倍 にしかならない

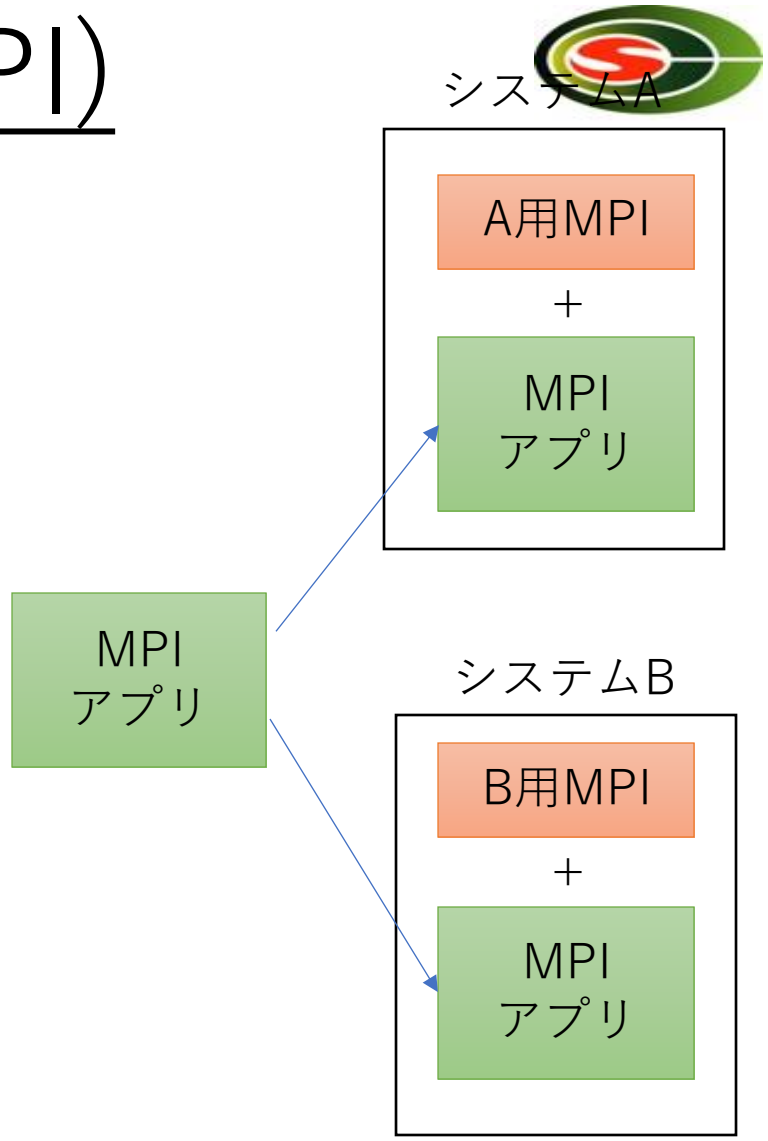
# アムダールの法則

- 理想：n台の計算機を使うと，1台の時よりn倍高速になってほしい
- 現実：アムダールの法則
  - 100台で100倍高速にするには，少なくとも99％の範囲が並列化可能でなければならない
  - しかし，通信はオーバーヘッドがある
- 1の計算を100台で分担するので，1台あたりの計算量は0.01になる
  - 通信量は線形(1/100)や2乗(1/10000)で減る
    - 通信データ長が小さい時，通信データ長を1/2にしても，通信時間は1/2にならない
- 並列計算の効率を高める研究は盛んに行われている



# Message Passing Interface (MPI)

- 高性能計算の分野で広く用いられている通信ライブラリの規格
  - https://www.mpi-forum.org/
  - MPI実装として，OpenMPIやMPICHなどがある
- 科学技術計算アプリケーションで並列計算を行う際のデファクトスタンダード
  - MPIができることは，ただ「データを送る」「データを受ける」だけではない
  - 科学技術計算で頻出する通信パターンをサポートする
- 仕様と実装が別れている
  - 同じアプリケーションを様々なシステムで実行可能
  - 大型の並列システムでは，独自のネットワークを持つシステムも多い（→システム特化MPI実装）

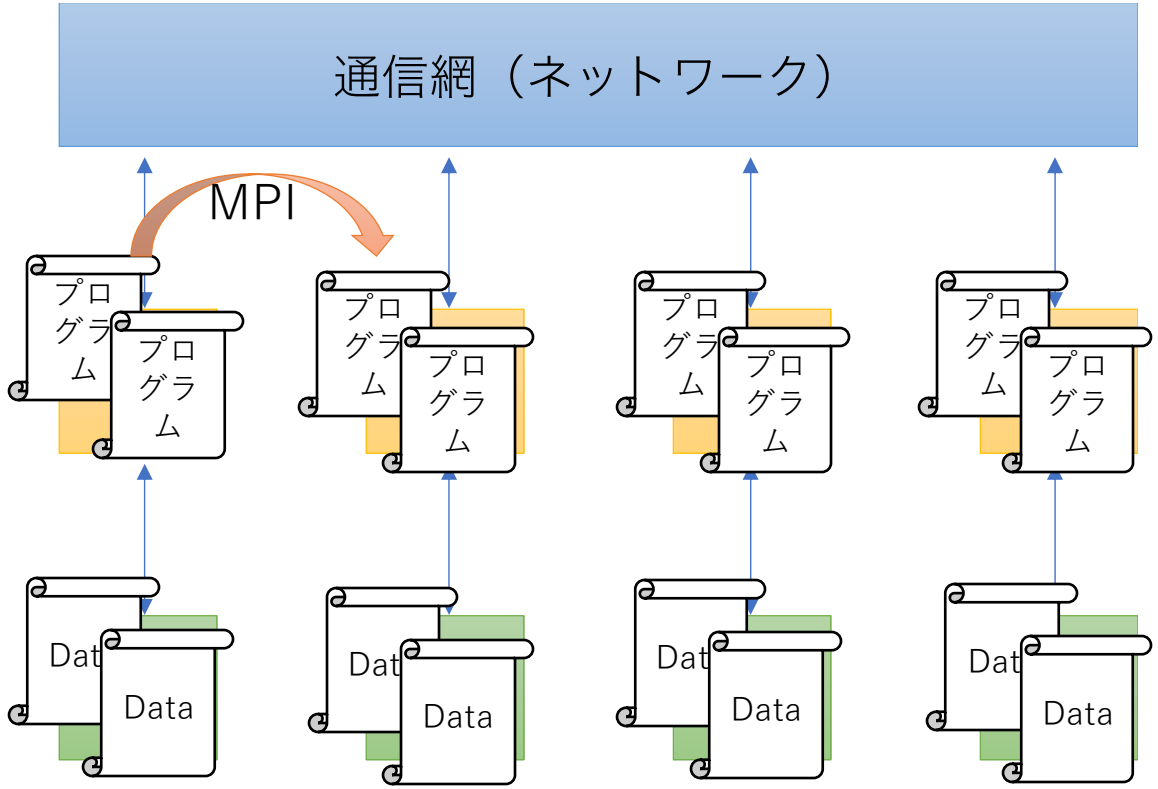


# Message Passing Interface (MPI)

- 1992年より標準化活動開始
  - 仕様書：https://www.mpi-forum.org/
  - 1994年 MPI-1.0リリース
  - 2009年 MPI-2.2リリース，647ページ
  - 2015年 MPI-3.1リリース，868ページ
  - 2021年 MPI-4.0リリース，1139ページ

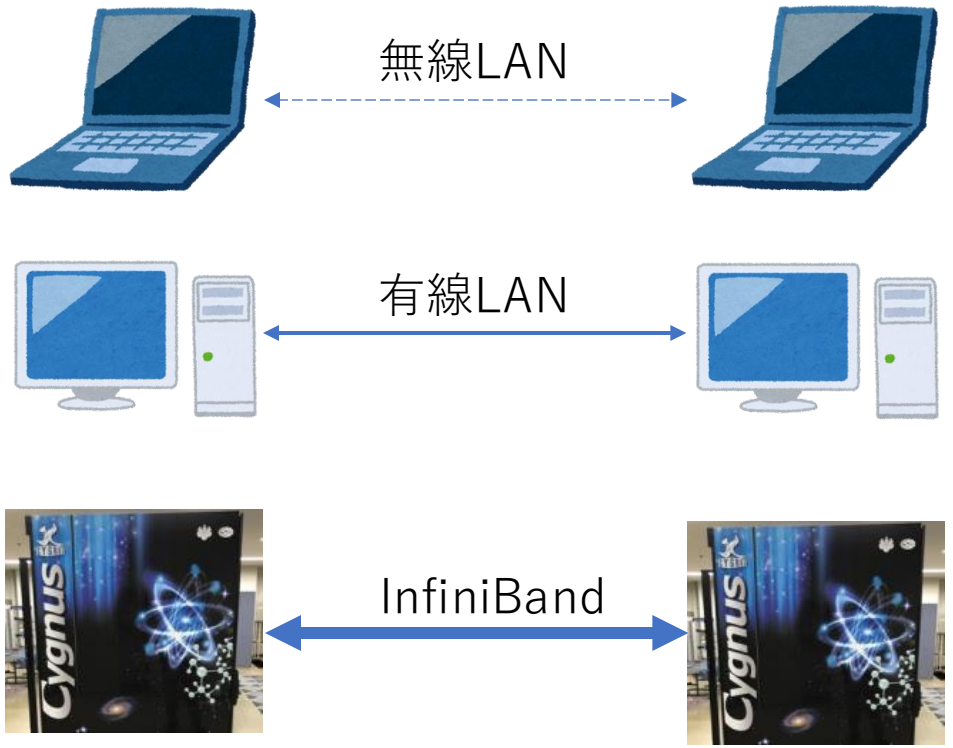
# SPMD

- Single Program Multiple Data
  - 同じプログラムで異なるデータを処理
- MPIの実行モデル
  - 同じプログラムを複数立ち上げる（プロセス）
    - 1つの計算機に2個以上のプロセスを配置しても良い
  - プロセス間の通信を記述する
    - どうすればデータが届くのかを抽象化するのがMPI
  - プロセス間は通信がなければ同期しない

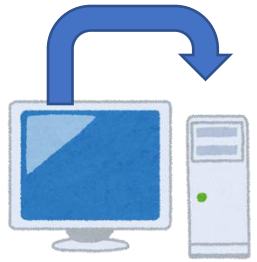


# OSSなMPI実装

- OpenMPI
  - https://www.open-mpi.org/
- MPICH
  - https://www.mpich.org/
- MVAPICH
  - https://mvapich.cse.ohio-state.edu/
- スケーラブルな実装
  - ノートPCからスパコンまで
  - TCP/IPからInfiniBandまで




in-memory
(同一計算機内)



# プログラミング言語について

- 以降，すべての説明はC言語に基づく
- MPIの仕様としては，CとFortranにおいて定義される
  - 以前はC++における仕様も定義されていたが廃止
- ラッパー経由で，他言語でも利用可能
  - C++: Boost.MPI
  - Python: mpi4py
  - Java: OpenMPI
  - Golang: go-mpi
  - Rust: rsmpi
  - など

# MPI初期化

- MPI_Init(int*, char***)
  - MPIの初期化関数
  - プログラムの引数をポインタで渡す (argc, argv)
    - 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)
  - OpenMPなど，マルチスレッドでMPIを扱いたい場合に必須
  - reqに必要なマルチスレッドレベル，providedに実際にMPIが対応しているレベルが帰る

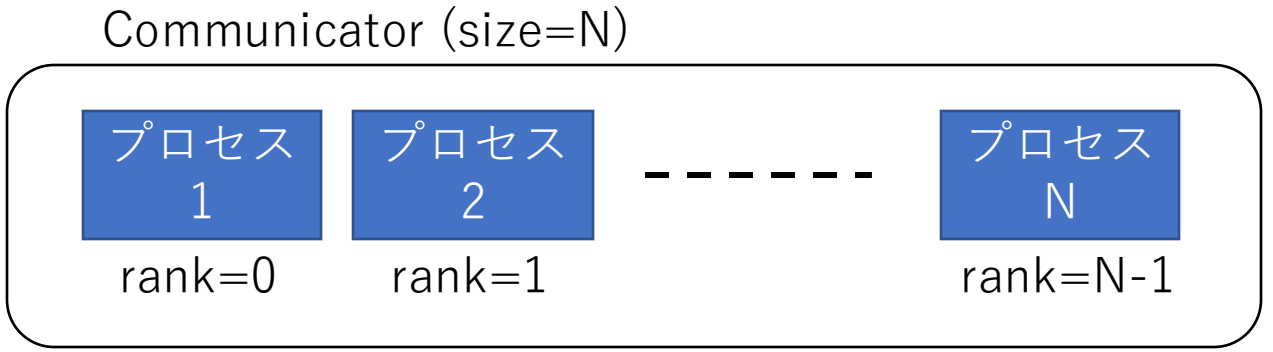
| | |
|---|---|
| MPI_THREAD_SINGLE | MPI_Initと等価．マルチスレッド不可． |
| MPI_THREAD_FUNNELED | マルチスレッド可能だが，MPI_Init_threadを呼んだスレッドのみMPIを使える |
| MPI_THREAD_SERIALIZED | 複数のスレッドからMPIを使えるが，同時に複数のスレッドからMPI関数を呼び出してはいけない |
| MPI_THREAD_MULTIPLE | 複数のスレッドからMPIを制限なく使える |

# MPI終了

- MPI_Finalize()
  - MPIの（正常）終了関数
  - これより後にMPI関数を呼んではいけない
  - （MPI以外の処理は記述しても良い）
- MPI_Abort(MPI_Comm, int)
  - 異常終了を表す
  - 「全体」が異常終了することを保証
  - MPIの関数を用いて，MPI_Abortを検出できる
    - → 異常状態からの回復を行うことも可能

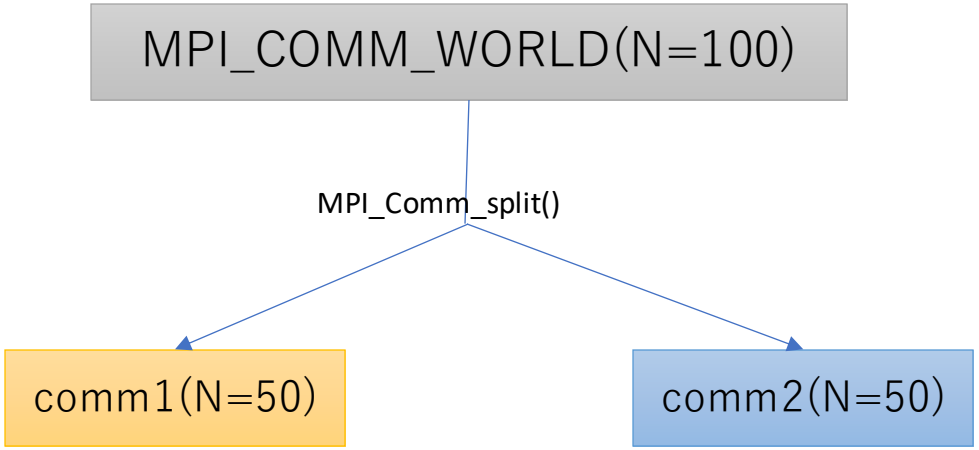
# Communicator

- Communicator（MPI_Comm型）
  - MPIにおいて「通信グループ」を指す用語
- MPI_COMM_WORLD
  - 全プロセスが参加しているCommunicator
  - 最初から作られている特殊なCommunicator
- MPI_Comm_rank(MPI_Comm, int*)
  - communicator内での自分の番号（1～，重複なし）を取得
- MPI_Comm_size(MPI_Comm, int*)
  - communicator内のrank総数を取得



# Communicator

- Communicator
  - 通信の相手・集団を表す
- 自由にプログラムで作成できる
  - 役割・目的に応じて作成
  - MPI_Comm_split 等（本講義では扱わない）
- タスク分割
  - 半分がシミュレーション，半分がデータ前・後処理など
- 一部プロセスだけが参加する通信
  - 集団通信（後述）に参加するプロセスを制限



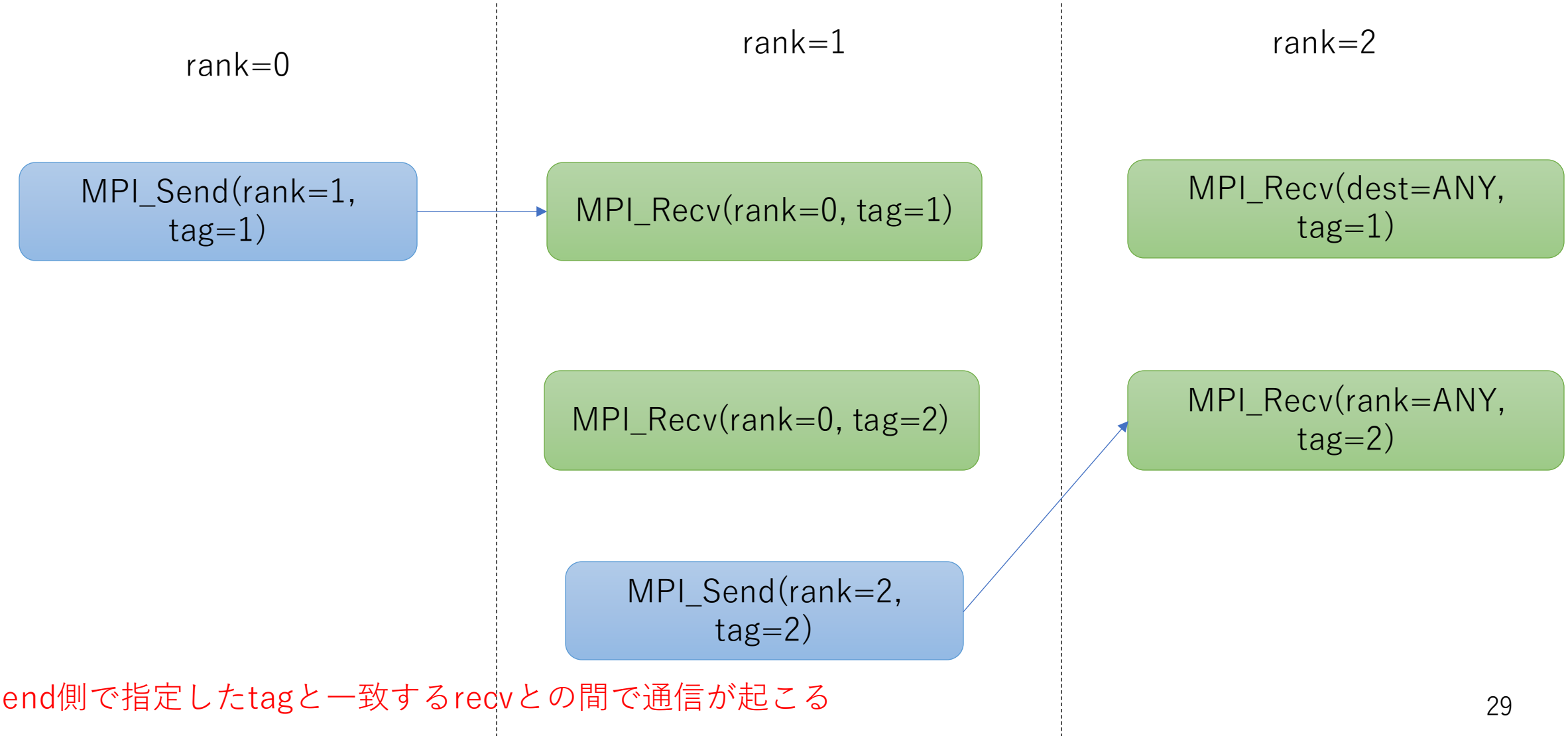
# 通信の種類について

1. 一対一通信 （point-to-point）
   - プロセスaとプロセスbの間で一対一に通信を行う
   - 送信側と受信側が互いに準備が出来ていることを保証してから通信を行う
     - handshake，電話のようなイメージ
2. 集団通信（collective）
   - 多数のプロセスが参加して，目的を達成するために通信を行う
   - 配列の総和を求める（Reduction），データの再配置（Gather/Scatter），配列の転地（Alltoall），待ち合わせ（Barrier）など
3. 片方向/単方向/一方向通信（one-sided）
   - プロセスaからプロセスbにデータを送信する
   - 受信側の都合を考慮せず，一方的に送信する
     - 宅配便でいきなり荷物を送りつけるイメージ，受信側が在宅かは考慮しない
     - 待ち合わせがないので高速だが，データ破損など起きないよう注意が必要になる
       - 受信側が使っているバッファを書き換えてはいけない
     - 送って良いことをプログラム側で保証する

# 一対一通信

- MPI_Send(void const* buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm)
  - buffer: 送るデータへのポインタ
  - count: 送信データの個数
  - datatype: 送信データ型
  - dest: 送信先Rank (in comm)
  - tag: タグ
  - comm: Communicator
- MPI_Recv(void* buf, int count, MPI_Datatype datatype, int source, int tag, MPI_Comm comm, MPI_Status* status)
  - source: 受信元Rank （ANYも可能）
  - status: 受信結果を返す．（長さや送信元など）

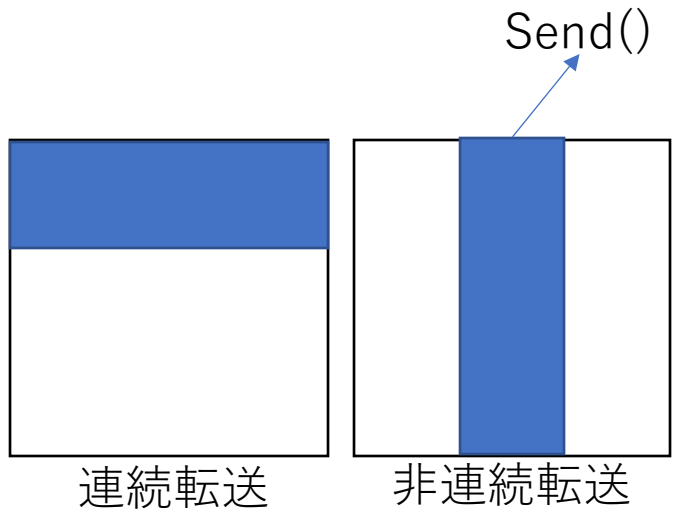
# 一対一通信




send側で指定したtagと一致するrecvとの間で通信が起こる

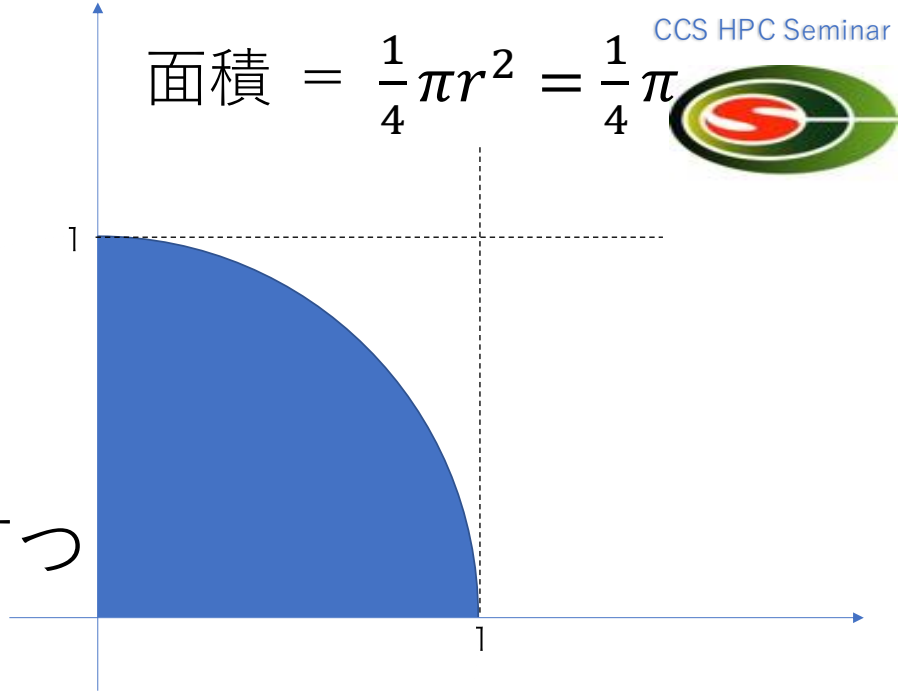
# MPIデータ型

- DataType
  - MPI_INT, MPI_FLOAT, MPI_DOUBLE など
- C言語のintは環境によってサイズが違う可能性があるため
  - 計算機が混在していても，正しくint型で通信できる
  - （計算機の種別が混在しているシステムは一般的ではないが・・・）
- 拡張データ型
  - 基本型を組み合わせて1つのデータ型として扱える
    - → 構造体，配列を表すのに用いる
  - 非連続なデータ転送
    - 配列の一部を転送する場合などに用いる
  - MPIの処理系に最適化の余地を与える



# 例題：モンテカルロ法

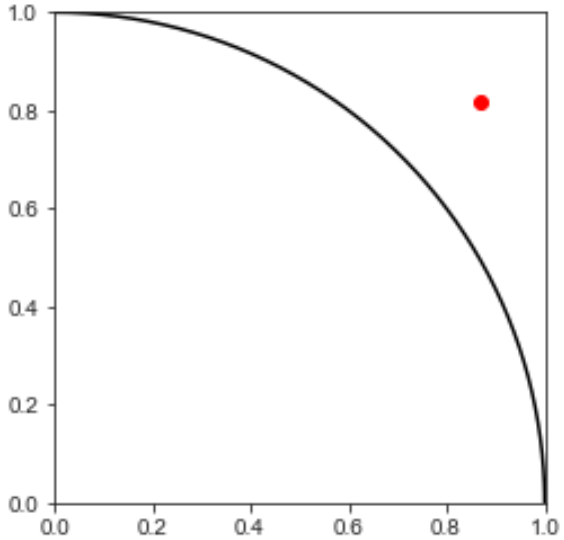
$$面積 = \frac{1}{4}\pi r^2 = \frac{1}{4}\pi$$



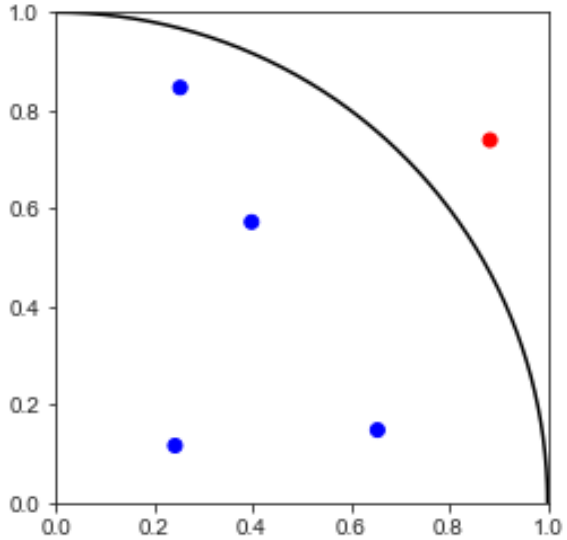

- 乱数に基づくアルゴリズム
- 半径1の円の1/4範囲を考える
- x=0~1, y=0~1の範囲にランダムに点をN個打つ
  - 点が円の中に入っているかを調べる
  - （原点からの距離）＜１か？
  - $\sqrt{x^2+y^2} < 1 \iff x^2+y^2 < 1$
- p個の点が円の中に入っていた場合，面積を $\frac{p}{N}$ と近似でき，$\pi = 4\frac{p}{N}$ と近似できる
  - Nが十分大きければ，精度良く近似できる
- 非常に並列化をしやすい

0 / 1 * 4 = 0

N=1

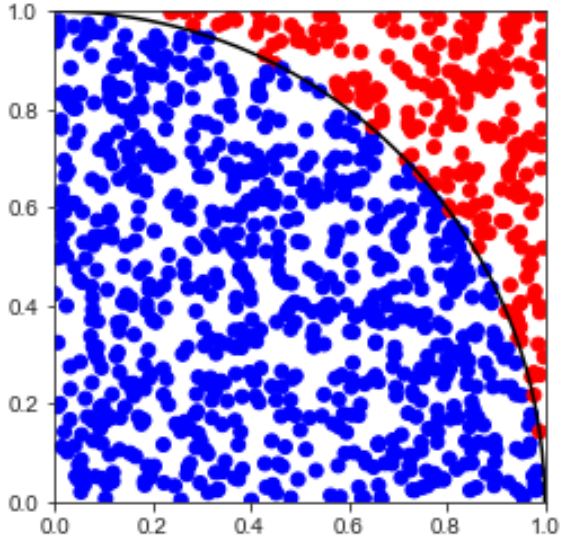



N=5



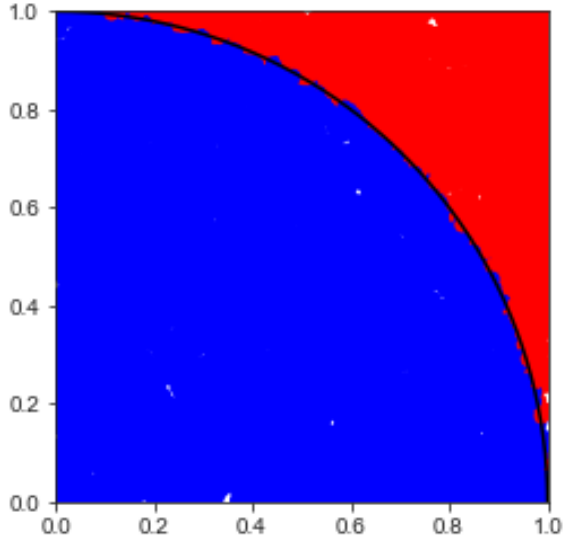

4 / 5 * 4 = 3.2

N=1000




779 / 1000 * 4 = 3.116

N=10000




7849 / 10000 * 4 = 3.1396

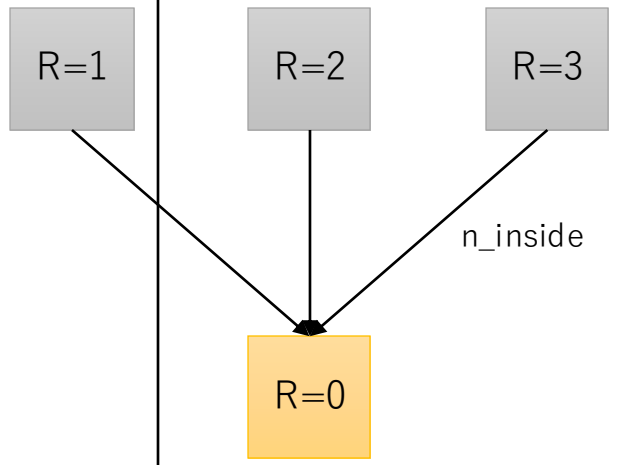
```
#include <mpi.h>

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;
}
```

loop回の計算をmpi_size個のプロセスで分割

```
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))
}
```

部分計算の結果をmpi_rank=0へ集める

最終結果を保有しているrank=0が結果を出力



```
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
```

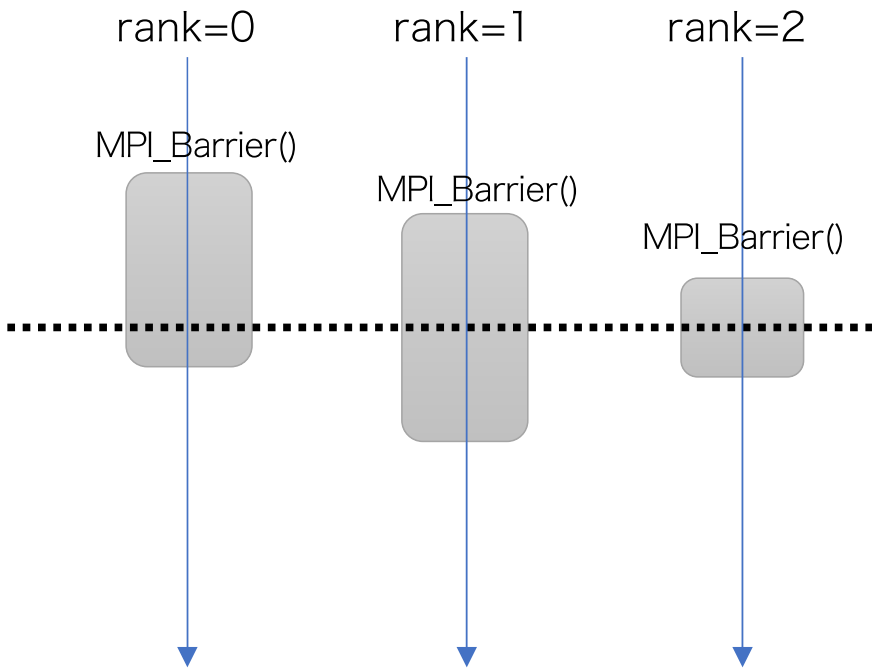
```
result = 3.1516000000
result = 3.1208000000
result = 3.1612000000
result = 3.1268000000
result = 3.1248000000
result = 3.1432000000
result = 3.1468000000
result = 3.1556000000
result = 3.1244000000
result = 3.1244000000
```

# Collective

- 集団通信
  - あるCommunicatorの全てのプロセスの参加が必要な通信
  - 1：1の通信ではなく，N：Nの通信を扱う
  - 通信を行うCommunicatorに属する全てのプロセスが関数を呼び出さなければならない
  - 部分的に行いたい場合は，それ用のCommunicatorを作る
- バリア同期, Broadcast, Gather, Scatter, Allgather, Alltoall, Reduce, Allreduce, ...

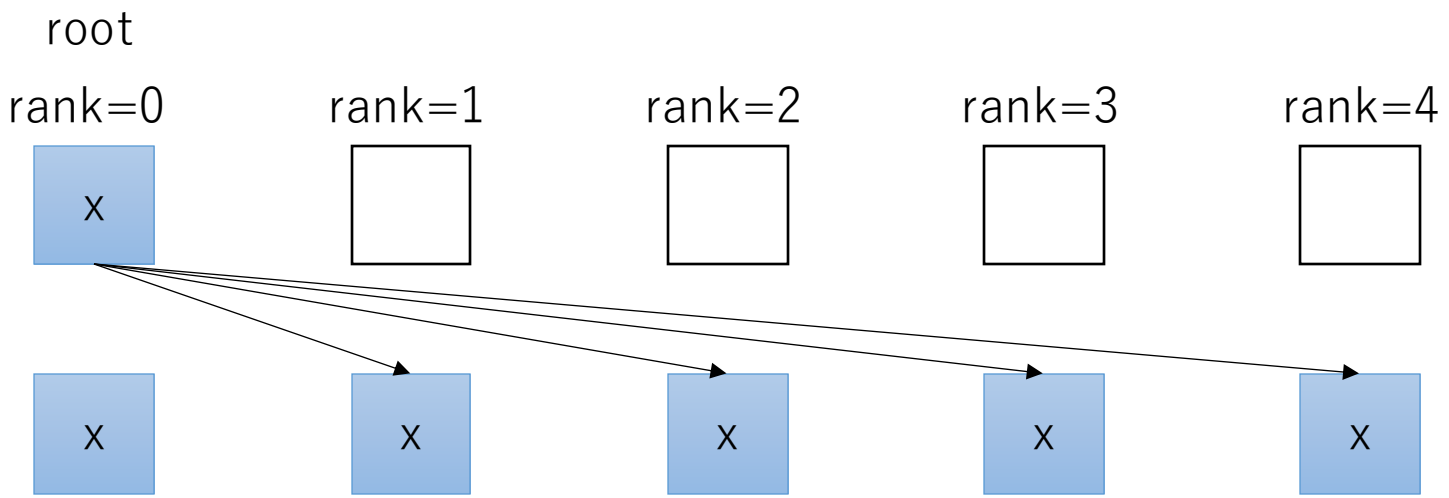
# MPI_Barrier

- MPI_Barrier(MPI_Comm comm)
  - commに属する全てのプロセス間で待ち合わせを行う
  - 全プロセスがMPI_Barrierへ到達したことを保証する
    - MPI_Barrierからの続きを実行するタイミングが同一であることは保証しない
    - 通信遅延，実行時ノイズなどの影響による



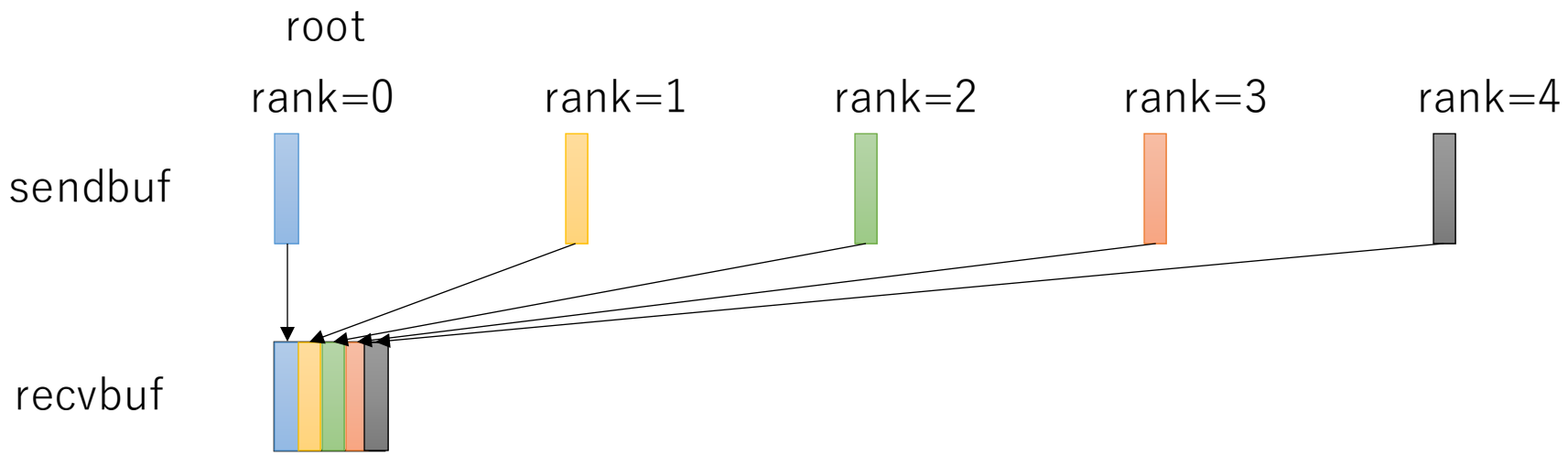
# MPI_Bcast

- MPI_Bcast(void* buffer, int count, MPI_Datatype datatype, int root, MPI_Comm comm)
  - rank=rootにあるbufferの中のデータを，他のランクに配る
  - 放送・Broadcast
  - 順にMPI_Sendをするよりも効率の良い実装が期待できる



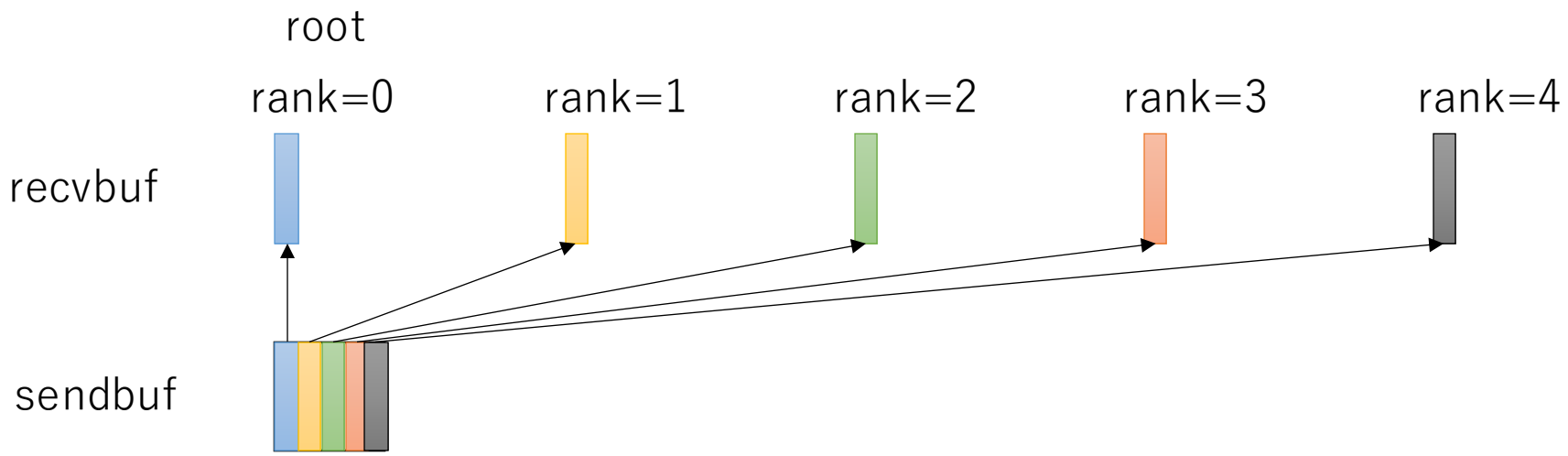
# MPI_Gather

- MPI_Gather(void const* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  - 各rankのsendbufに入っているデータをrank=rootのrecvbufへ集める



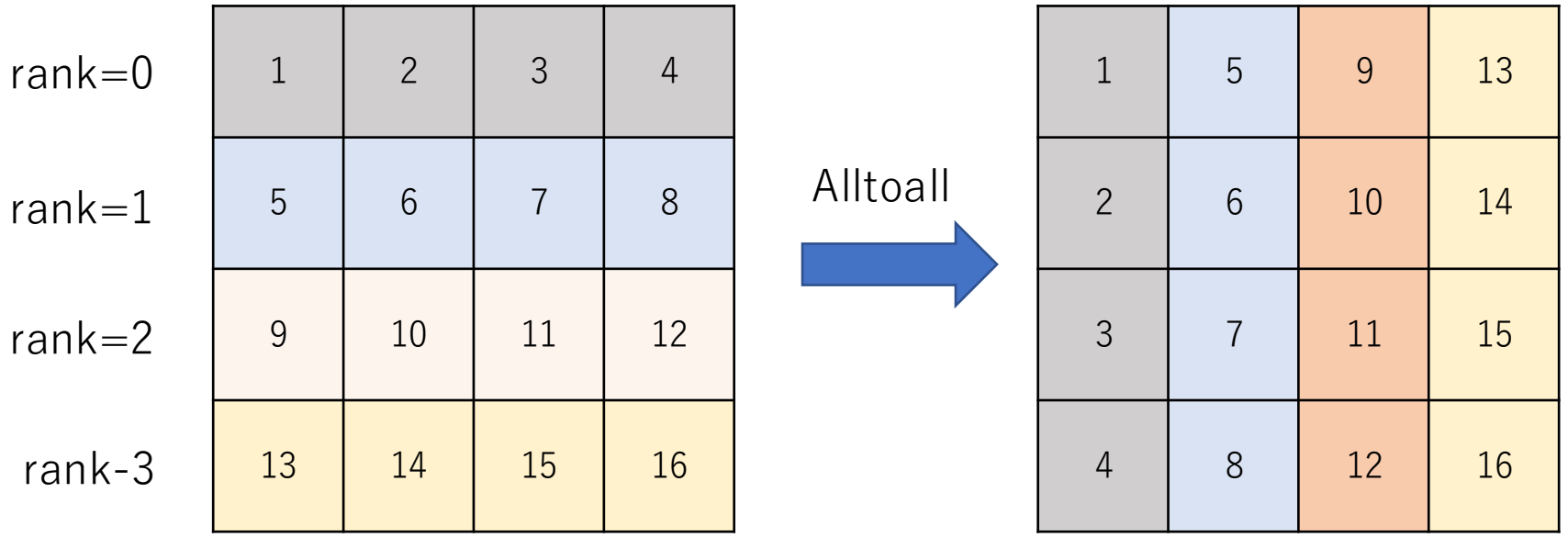
# MPI_Scatter

- MPI_Gather(void const* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm)
  - rank=rootのsendbufに入っているデータを，各rankのrecvbufへ配る
  - Gatherの逆



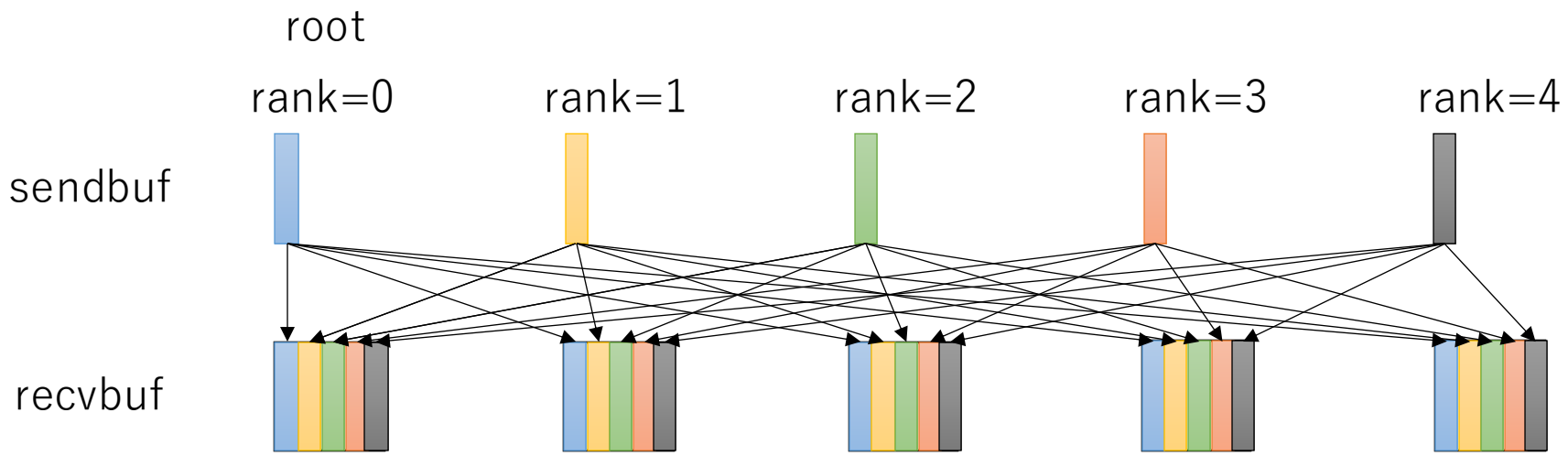
# MPI_Alltoall

- MPI_Alltoall(void const* sendbuf, int sendcount, MPI_Datatype sendtype, void* recvbuf, int recvcount, MPI_Datatype recvtype, MPI_Comm comm)
  - 列方向に分散して持っている行列の転置を行う
  - 全rankが全rankと通信を行う必要があるため，alltoallと呼ばれる
  - ネットワーク・トポロジーの影響を受けやすく，実装が難しい



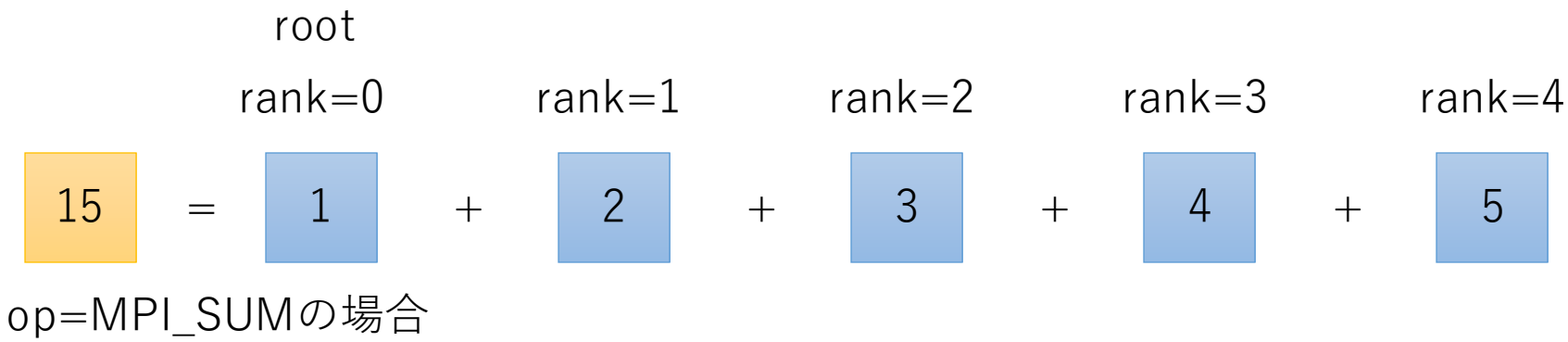
# MPI_Allgather

- MPI_Allgather(void const* sendbuf, int sendcount, MPI_Datatype datatype, void* recvbuf, int recvcount, MPI_Datatype datatype, MPI_Comm comm)
  - 挙動は Gather → Bcast と同等
    - 計算結果を集め，再び全員で共有するシーンで用いられる
  - ただし，より効率的なアルゴリズムが用いられる



# MPI_Reduce / Allreduce

- MPI_Reduce(void const* sendbuf, void* recvbuf, int count, MPI_Datatype datatype, MPI_Op op, int root, MPI_Comm comm)
  - 各rankのsendbufに入っているデータに対してopを適用して縮約を行う
  - $\mathrm{recvbuf}_{\mathrm{root}} = \mathrm{sendbuf}_0$ op $\mathrm{sendbuf}_1$ op … op $\mathrm{sendbuf}_N$
  - Gatherしてopを適用するのと同じだが，最適化された実装が用いられる
  - 使用例：配列の総和を求める時，rank毎に部分和を求めてからReduceする
- Allreduce = Reduce + Bcast



# MPI通信のモデル

- 送信側がMPI_Sendを呼んで戻った時，受信側にデータが必ず届いているだろうか？ → わからない
  - 送信側の通信バッファにデータを書いただけかもしれない
    - 何も通信されていないパターン
  - 送信側は全データを送ったが，途中のスイッチのバッファに溜まっているかもしれない
    - 送信完了したが，受信完了していないパターン
- MPI通信関数は，実際に通信完了したかどうかを保証しない
  - Barrierなどを用いて同期しないとわからない
- MPI通信の完了 → バッファが利用可能になった時と考えれば良い
  - MPI_Send: 送信用領域を書き換えても良い
  - MPI_Recv: 受信用領域から読み出して良い

# Non-blocking

- Non-blocking通信関数
  - 通信を開始したら直ちに制御が戻る関数郡
  - 関数から帰った時点でバッファに触ってはいけない
- MPI_Request
  - non-blocking通信を表すチケット
  - あとで通信が完了したかどうかを確認するために用いる
- MPI_Wait/Waitall/Waitany
  - Requestが完了するまで待機する
  - Waitall/Waitany
    - 複数のMPI_Requestを同時に監視する


Blocking通信の場合


Non-Blocking通信の場合

# Non-blocking Point-to-Point

- 普通の通信関数の名前にI（アイ）をつける
  - “I” for Immediate or Incomplete
  - MPI_Requestが出力に増えていることが特徴
- 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 s[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]);

// MPI_Waitall(2, r, s);
```

# Non-blocking Collective

- 数が多いので，詳細は省略
  - MPI-3で導入
  - 利用目的はpoint-to-pointと同じ
- Ibarrier, Ibcast, Igather, Iscatter, Ireduce, ..., etc.
  - 関数名にIを入れると，Non-blockingバージョン
  - point-to-point通信と同様にMPI_Requestが帰ってくるので，後でwaitを行う
- MPI_Ibarrier(MPI_Comm comm, MPI_Request* request)
- MPI_Ireduce(..., MPI_Comm, MPI_Request*)
- など

# 例題：ラプラス方程式

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

離散化

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

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

- ラプラス方程式の陽的解法

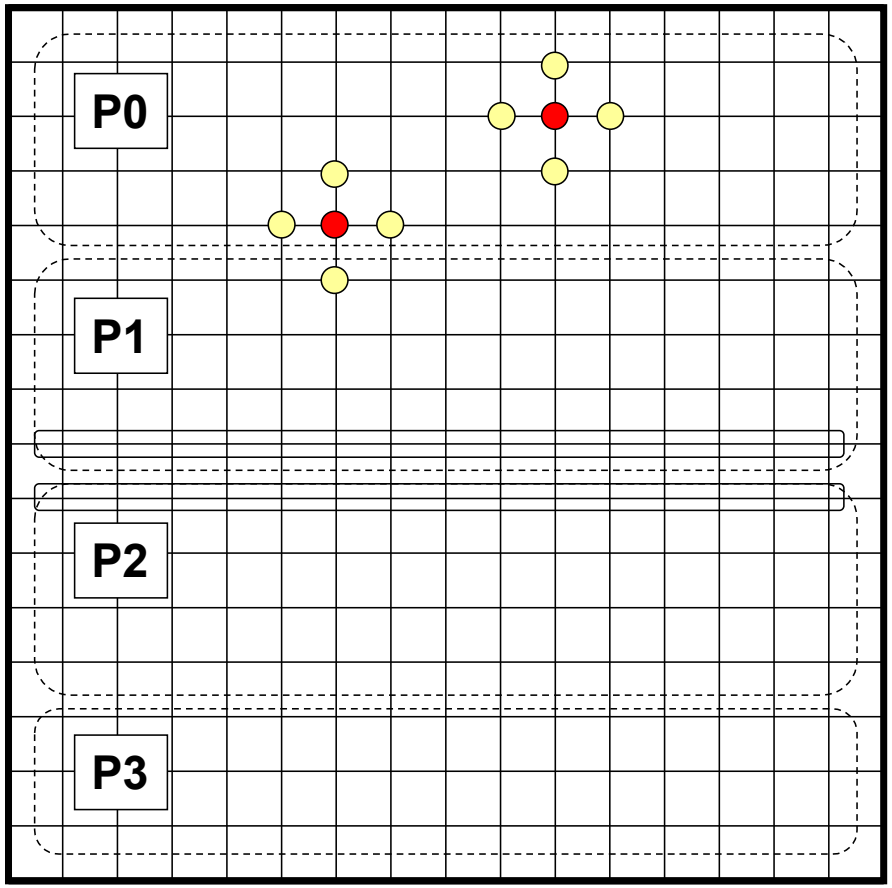
$$f(0,0)_{new}=\frac{1}{4}\left(f_{old}(0,-1)+f_{old}(-1,0)+f_{old}(1,0)+f_{old}(0,1)\right)$$

  - 上下左右の4点の平均で値を更新していく
  - 配列を2つ(old,new)用意して，new→oldへコピーしてからnewを更新していく
  - 典型的な領域分割
    - 大きな計算を複数のプロセスに分割して計算を行う
  - 最後に残差を取る

# 例題：ラプラス方程式

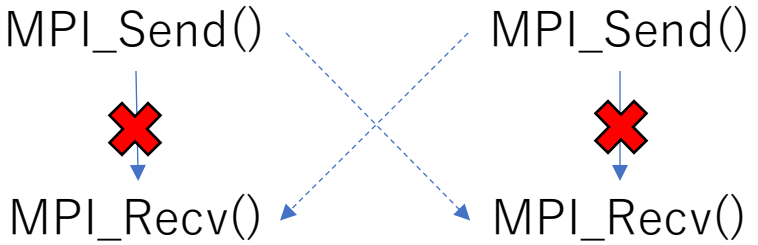
u[x][y] = 0.25 * (uu[x-1][y] + uu[x+1][y] + uu[x][y-1] + uu[x][y+1])

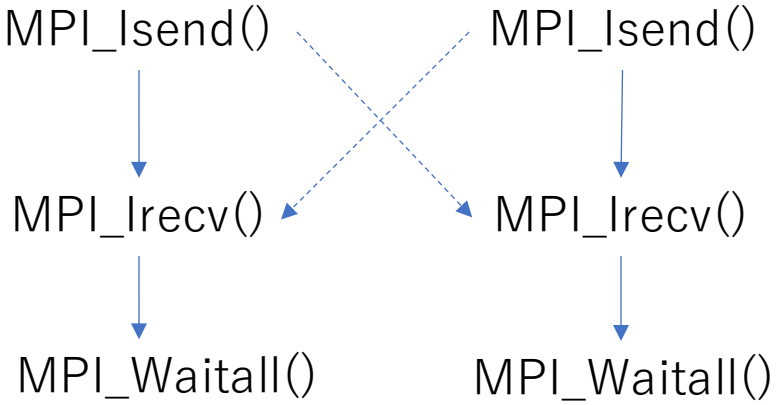
- 二次元領域を1次元にブロック分割
- 計算に必要な要素が隣のプロセスにある場合がある
  - y-1やy+1が自領域から外れる場合
  - 右図黄色の領域
  - 境界領域 (boundary) と呼ぶ
- 境界にあるデータを隣のプロセスに送信しなければならない



# データの交換

- 単純に考えると・・
  - MPI_Send() & MPI_Recv()
    - MPI_Recvがいつまでたっても実行できない可能性
  - MPI_Sendがバッファリングされる場合のみ実行可能
    - 一般的に保証できない
- 解決方法
  - MPI_Sendrecv()を使う
    - 送信と受信を不可分に扱う
  - Non-blocking通信を使う
    - 必ずSendとRecvを実行できる
    - 任意の順番で通信が実行できる



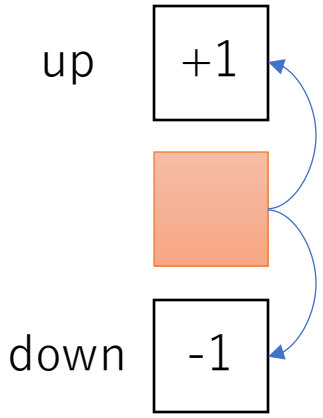
# Cartesian topology

- 直交座標系のトポロジー
- MPI_Cart_create(MPI_Comm comm_old, int ndims, int const* dims, int const* periods, int reorder, MPI_Comm* comm_cart)
  - comm_oldをndims次元に分割してcomm_cartを作成する
  - dims, periodsは配列を渡し，各次元の大きさ・周期境界かどうかを示す
  - reorderはrankの再割当てを行うかどうか．falseならばcomm_oldを引き継ぐ
- MPI_Cart_shift(MPI_Comm comm, int direction, int disp, int* rank_source, int* rank_dest)
  - directionはシフトする次元(0 ~ ndims-1)
  - disp個だけシフトした時の，送信元と送信先がrank_sourceとrank_destに格納される
  - 周期的ではない場合，境界を超えるとMPI_PROC_NULLが返される

```
/*
** 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;
```

二次元計算領域
uuは更新用配列

```
void initialize() {
   int x, y;
   /* 初期値を設定 */
   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);

   /* 境界をゼロクリア */
   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;
   }
}
```




```
#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);
    _start = 1 + xsize * myid;
    _end = 1 + xsize * (myid + 1);
```

comm1dを1次元トポロジーで作成
境界は非周期
ndims=1, dims={numprocs}

自プロセスの
計算範囲（index）
x_start <= x < x_end

上下のプロセス番号をup, downに取得
境界ではMPI_PROC_NULL

```
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]);
    }
```

境界領域の交換

```
void lap_solve(MPI_Comm comm) {
  ....
```

残差の計算.
Reduceで和を計算

```
    /* 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);
}
```

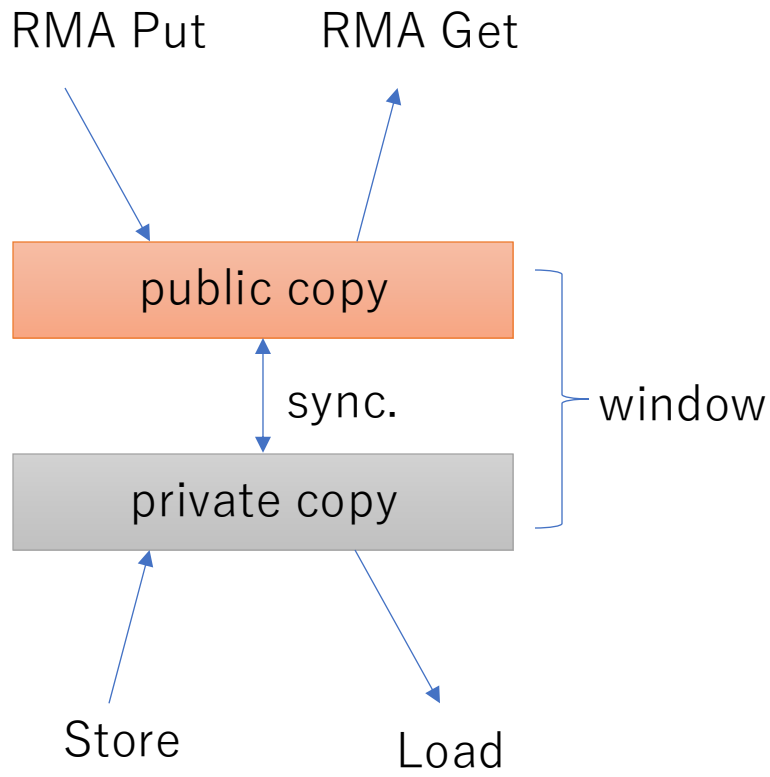
```
int main(int argc, char *argv[]) {
    MPI_Init(&argc, &argv);
    MPI_Comm_size(MPI_COMM_WORLD, &numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    MPI_Get_processor_name(processor_name, &namelen);
    fprintf(stderr, "Process %d on %s¥n", myid, processor_name);

    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);
}
```

初期データ計算

メイン計算

# One-Sided

- Windowと呼ばれるオブジェクトを通じて片方向通信を扱う
  - public copy & private copy
  - RMA separate memory model
- public copyとprivate copyを同期するための関数
  - 同期するまでpublicとprivateのデータが一致することが保証されない
- RMA unified memory model (since MPI-3)
  - public == private
  - 高性能な通信機構で扱う際の挙動に近いモデル




動作の定義上のために2つの領域に分かれているが，実際に分かれているかどうかは実装依存.

# 安全性

- 片方向通信は勝手にデータを送信・取得するモデル
  - 「いつ」アクセスが有効になるのかが重要
  - 誤ってデータを上書きしてしまったり，古いデータを読んでしまったりする危険性
  - Send/Recvよりも考える必要のある要素は多い
- MPIの片方向通信は比較的ゆるい制約の上で成り立つ
  - さまざまな計算機・通信機構上で実装を可能にするため
  - 特定の関数群を呼んだ時に，状態が更新されていることを保証する
    - 呼ぶまでは，状態が変化していないかもしれないし，変化していても良い
- 先進的な環境では，異なる計算機のメモリに直接アクセスができるものがある
  - Remote Direct Memory Access (RDMA)
    - こういった機構がある場合は，片方向通信は非常に効率が良い
  - InfiniBandの場合は，HCAが直接メモリへアクセスできる（CPUの能力を奪わない）

# MPI レポート課題

- 講義中で触れたLaplace方程式を解くプログラムを改良しなさい．レポートにはプログラム，プログラムの説明，実行結果，実行結果の説明を含めること．
  - ただし，MPI（通信，データ分割）に関する改良に限る
  - 具体的にどこをどう改良したかを記述すること

- 改良するポイントのヒントを以下に示すが，あくまで一例であり，以下にない内容でも構わない
  - 片方向通信を用いる
  - 配列の確保方法を最適化する
    - 計算する領域は $N^2/p$ なのに，配列は $N^2$ 全て確保している
  - 二次元分割を行う

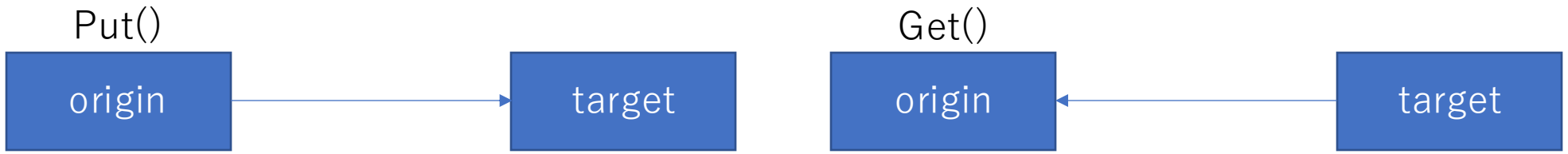
# laplace.c

```
/*
 * 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;

void initialize() {
    int x, y;
    /* 初期値を設定 */
    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);
    /* 境界をゼロクリア */
    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;
    }
}

#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
     * non periodical 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);

    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]);
    }
    /* 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("t_sum = %g\n", t_sum);
    }
    MPI_Comm_free(&comm1d);
}

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);
}
```

# 以下は補助資料

# RMA Epoch

- あるMPI_Winに対して，片方向通信を行える期間をRemote Memory Access (RMA) Epochと呼ぶ
  - origin: RMAを行うプロセス
  - target: RMAでアクセスされるプロセス
    - データの流れる方向とは無関係． != sender/receiver
  - access epoch: originがRMAを行う期間
  - exposure epoch: targetがRMAされる期間



# Window

- MPI_Win_create(void* base, MPI_Aint size, int disp_unit, MPI_Info info, MPI_Comm comm, MPI_Win* win)
  - baseからbase+sizeまでの範囲を指すWindowをcomm上に作りwinへ返す
  - disp_unitはアクセス位置を指定する時に用いる倍率
    - 1ならばバイトアクセス，sizeof(double)などを与えれば配列アクセス時に便利
  - infoはオプションを与えるために用いる
- MPI_Win_free(MPI_Win win)
  - winを破棄する

# Window

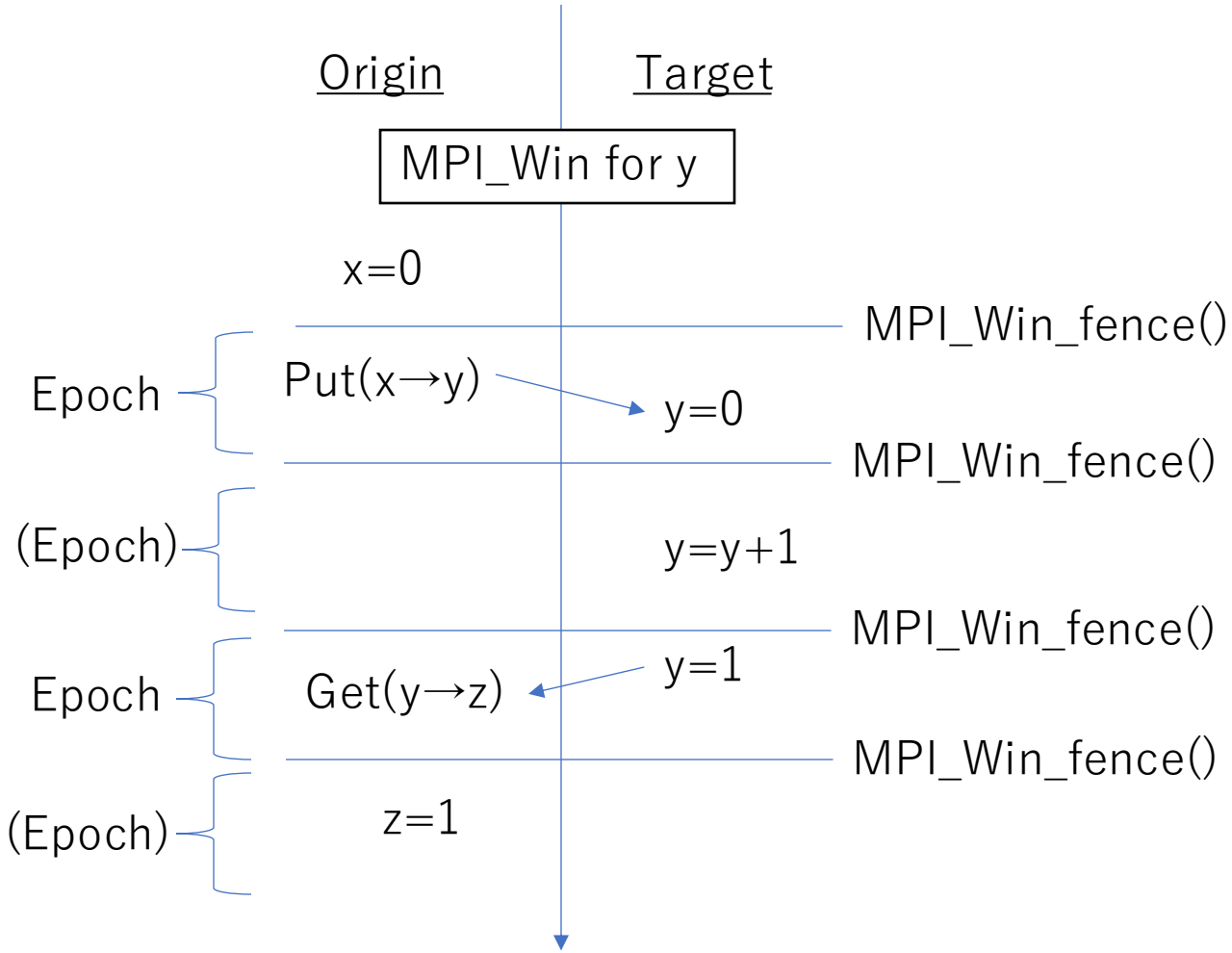
- MPI_Win_allocate(MPI_Aint size, ..., void* baseptr, MPI_Win* win)
  - sizeバイトの領域を確保した上でその領域に対するWindowを作る
    - 出力 baseptr: 先頭アドレス，win: Window
  - ただし，どういう手段で領域が確保されるかは実装依存
    - 片方向通信に最適化された領域が確保されるかもしれない
- MPI_Win_free(MPI_Win win)
  - winを破棄する

# Put/Get

- MPI_Get/Put(const void* origin_addr, int origin_count, MPI_Datatype origin_datatype, int target_rank, MPI_Aint target_disp, int target_count, MPI_Datatype target_datatype, MPI_Win win);
  - target_rankにあるwinとの通信を行う
  - origin_addrからorigin_datatype型のデータをorigin_count個の範囲と
  - target_dispの位置からtarget_datatype型のデータとしてtarget_count個の範囲の通信を行う
  - origin_addrはwinの範囲内でなくて良い
- Put: originがtargetへ書き込む
- Get: originがtargetから読み込む

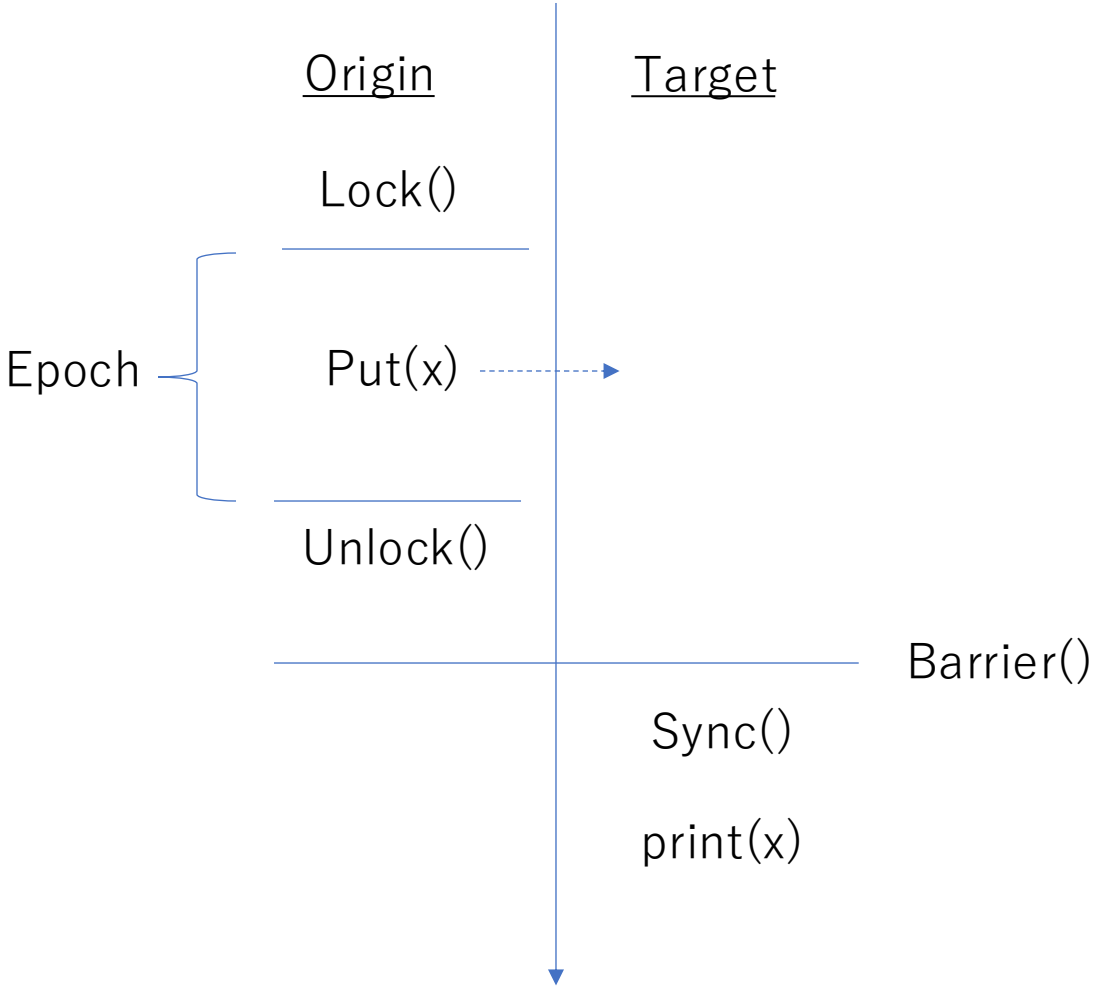
# MPI_Win_fence

- Fenceの前に発行されたRMAが全て完了することを保証する
  - 暗黙にBarrierを含む
- **MPI_Win_fence(int assert, MPI_Win win)**
  - winを持つ任意のプロセスに対してアクセス可能
  - assertは最適化のための補助情報
    - この区間で書き込みしない など



# MPI_Win_lock

- access epochを開始する
  - targetと同期を取らず，自由なタイミングでtargetへアクセスできる
- **MPI_Win_lock(int lock_type, int rank, int assert, MPI_Win win)**
  - rank上にあるwinさ指す領域へのaccess epochを開始する
  - lock_type
    - MPI_LOCK_EXCLUSIVE
    - MPI_LOCK_SHARED
  - assertは最適化のための補助情報
- **MPI_Win_unlock(rank, win)**



# MPI_Win_Start/Complete/Post/Wait

- Fenceを細分化して，同期する相手を選べるもの
- 本講義では省略

# RMA同期まとめ

- RMA通信で読み書きできることを保証するには同期関数が必要
  - Fenceが最もわかりやすいが，過剰な同期を行ってしまう可能性
  - Lockはtargetとの同期がないので注意が必要
    - BarrierやReduceなど他の手段で同期を取り，RMAしても良いことが保証されるときに用いる