並列有限要素法への道
―並列データ構造―

中島 研吾
東京大学情報基盤センター

- MPI超入門
- 並列有限要素法への道
  - 局所データ構造
- 並列有限要素法のためのMPI
  - Collective Communication（集団通信）
  - Point-to-Point (Peer-to-Peer) Communication（1対1通信）

MPIとは（1/2）

- Message Passing Interface
- 分散メモリ間のメッセージ通信APIの「規格」
  - プログラム，ライブラリ，そのものではない
    - http://phase.hpcc.jp/phase/mpi-j/ml/mpi-j-html/contents.html
- 歴史
  - 1992　MPIフォーラム
  - 1994　MPI-1規格
  - 1997　MPI-2規格（拡張版），現在はMPI-3が検討されている
- 実装
  - mpich　アルゴンヌ国立研究所
  - LAM
  - 各ベンダー
  - C/C++,FOTRAN,Java ; Unix，Linux，Windows，Mac OS

MPIとは（2/2）

- 現状では，mpich（フリー）が広く使用されている。
  - 部分的に「MPI-2」規格をサポート
  - 2005年11月から「MPICH2」に移行
  - http://www-unix.mcs.anl.gov/mpi/
- MPIが普及した理由
  - MPIフォーラムによる規格統一
    - どんな計算機でも動く
    - FORTRAN，Cからサブルーチンとして呼び出すことが可能
  - mpichの存在
    - フリー，あらゆるアーキテクチュアをサポート
- 同様の試みとしてPVM（Parallel Virtual Machine）があったが，こちらはそれほど広がらず

参考文献

- P.Pacheco「MPI並列プログラミング」，培風館，2001（原著1997）
- W.Gropp他「Using MPI second edition」，MIT Press, 1999.
- M.J.Quinn「Parallel Programming in C with MPI and OpenMP」，McGrawhill, 2003.
- W.Gropp他「MPI：The Complete Reference Vol.I, II」，MIT Press, 1998.
- http://www-unix.mcs.anl.gov/mpi/www/
  - API（Application Interface）の説明

MPIを学ぶにあたって(1/2)

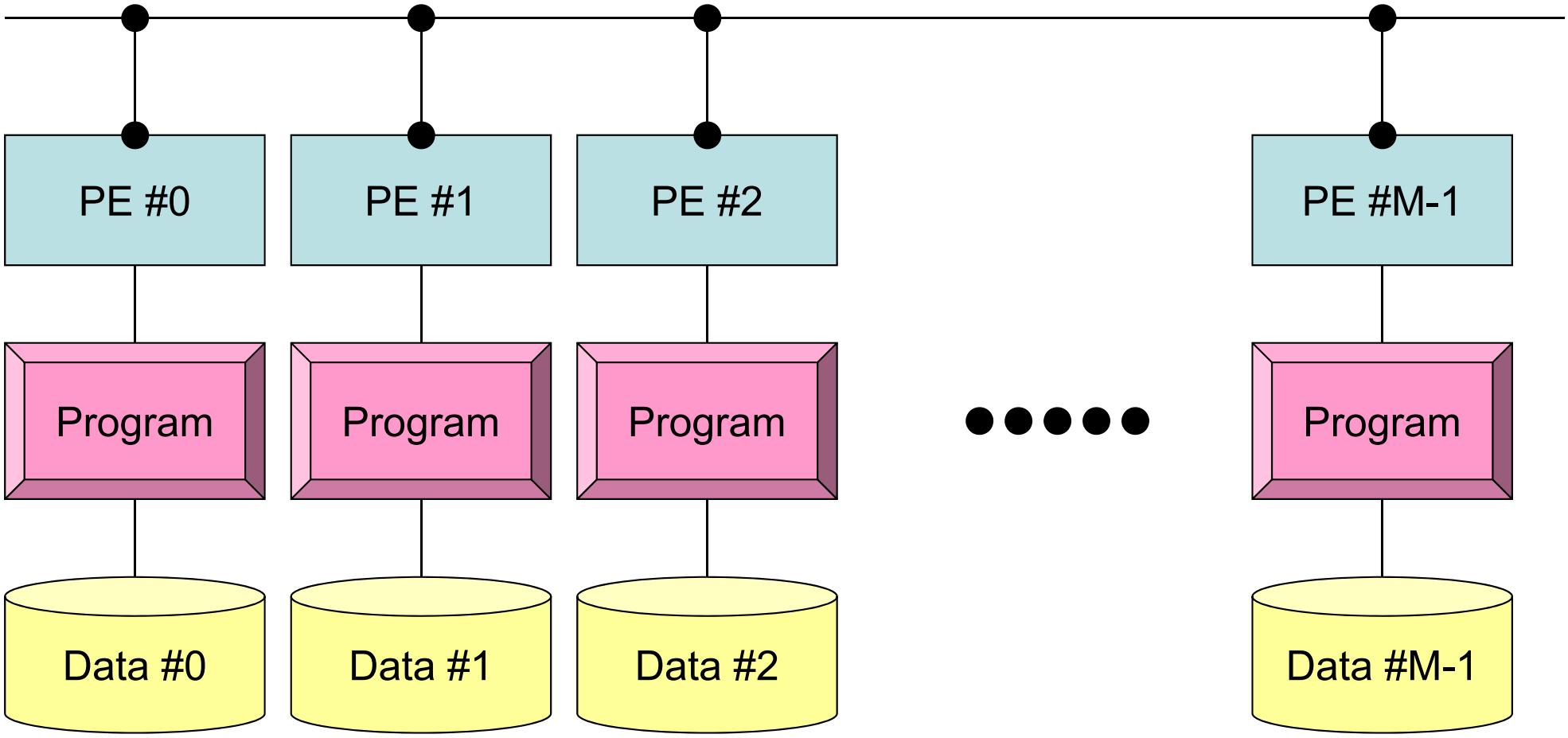
- 文法
  - 「MPI-1」の基本的な機能(10程度)について習熟する
    - MPI-2では色々と便利な機能があるが・・・
  - あとは自分に必要な機能について調べる，あるいは知っている人，知っていそうな人に尋ねる
- 実習の重要性
  - プログラミング
  - その前にまず実行してみること
- SPMD/SIMDのオペレーションに慣れること・・・「つかむ」こと
  - Single Program/Instruction Multiple Data
  - 基本的に各プロセスは「同じことをやる」が「データが違う」
    - 大規模なデータを分割し，各部分について各プロセス(プロセッサ)が計算する
  - 全体データと局所データ，全体番号と局所番号

PE: Processing Element
プロセッサ，領域，プロセス

SPMD

この絵が理解できればMPIは9割方理解できたことになる。

```
mpirun -np M <Program>
```



各プロセスは「同じことをやる」が「データが違う」
大規模なデータを分割し，各部分について各プロセス（プロセッサ）が計算する
通信以外は，単体CPUのときと同じ，というのが理想

用 語

- プロセッサ，コア
  - ハードウェアとしての各演算装置。シングルコアではプロセッサ＝コア
- プロセス
  - MPI計算のための実行単位，ハードウェア的な「コア」とほぼ同義。
  - しかし1つの「プロセッサ・コア」で複数の「プロセス」を起動する場合もある（効率的ではないが）。
- PE（Processing Element）
  - 本来，「プロセッサ」の意味なのであるが，本講義では「プロセス」の意味で使う場合も多い。次項の「領域」とほぼ同義でも使用。
    - マルチコアの場合は：「コア＝PE」という意味で使うことが多い。
- 領域
  - 「プロセス」とほぼ同じ意味であるが，SPMDの「MD」のそれぞれ一つ，「各データ」の意味合いが強い。しばしば「PE」と同義で使用。
- MPIのプロセス番号（PE番号，領域番号）は0から開始
  - したがって8プロセス（PE，領域）ある場合は番号は0～7

MPIを学ぶにあたって（2/2）

- 繰り返すが，決して難しいものではない。
- 以上のようなこともあって，文法を教える授業は2~3回（90分）程度で充分と考えている。
  - 今回はその時間は無いが・・・
- とにかくSPMDの考え方を掴むこと！

Actually, MPI is not enough ...

- Multicore Clusters
- Heterogeneous Clusters (CPU+GPU, CPU+Manycore)

- MPI + X (+ Y)
- X
  - OpenMP, Pthread
  - OpenACC
  - CUDA, OpenCL
- Y
  - Vectorization

まずはプログラムの例

hello.f

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h‘
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

hello.c

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

FORTRAN/Cの違い

- 基本的にインタフェースはほとんど同じ
  - Cの場合，「**MPI_C**omm_size」のように「MPI」は大文字，「MPI_」のあとの最初の文字は大文字，以下小文字

- FORTRANはエラーコード（ierr）の戻り値を引数の最後に指定する必要がある。
- Cは変数の特殊な型がある
  - MPI_Comm, MPI_Datatype, MPI_Op etc.

- 最初に呼ぶ「MPI_INIT」だけは違う
  - `call MPI_INIT (ierr)`
  - `MPI_Init (int *argc, char ***argv)`

何をやっているのか？

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;
    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);
    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

```
#!/bin/sh
#PJM -L “node=1“                ノード数
#PJM -L “elapse=00:10:00“       実行時間
#PJM -L “rscgrp=lecture“        実行キュー名
#PJM -g “gt61“                  グループ名（俗称：財布）
#PJM -j
#PJM -o “hello.lst“             標準出力ファイル名
#PJM --mpi “proc=4“             MPIプロセス数

mpiexec ./a.out                 実行ファイル名
```

- **mpiexec** により4つのプロセスが立ち上がる（今の場合は”proc=4”）。
  - 同じプログラムが4つ流れる。
  - データの値（myid）を書き出す。
- 4つのプロセスは同じことをやっているが，データとして取得したプロセスID（myid）は異なる。
- 結果として各プロセスは異なった出力をやっていることになる。
- まさにSPMD

mpi.h，mpif.h

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h‘
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

```
#include "mpi.h"
#include <stdio.h>
int main(int argc, char **argv)
{
    int n, myid, numprocs, i;

    MPI_Init(&argc,&argv);
    MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
    MPI_Comm_rank(MPI_COMM_WORLD,&myid);

    printf ("Hello World %d¥n", myid);
    MPI_Finalize();
}
```

- MPIに関連した様々なパラメータおよび初期値を記述。
- 変数名は「MPI_」で始まっている。
- ここで定められている変数は，MPIサブルーチンの引数として使用する以外は陽に値を変更してはいけない。
- ユーザーは「MPI_」で始まる変数を独自に設定しないのが無難。

Fortran

MPI_INIT

- MPIを起動する。他のMPIサブルーチンより前にコールする必要がある（必須）
- 全実行文の前に置くことを勧める。

- **call MPI_INIT (ierr)**
  - **ierr**　整数　0　完了コード

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

Fortran

MPI_FINALIZE

- MPIを終了する。他の全てのMPIサブルーチンより後にコールする必要がある（必須）。
- 全実行文の後に置くことを勧める
- これを忘れると大変なことになる。
  - 終わったはずなのに終わっていない・・・
- **call MPI_FINALIZE (ierr)**
  - **ierr**　　整数　　0　　完了コード

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

Fortran

MPI_COMM_SIZE

- コミュニケーター「comm」で指定されたグループに含まれるプロセス数の合計が「size」にもどる。必須では無いが，利用することが多い。

- **call MPI_COMM_SIZE (comm, size, ierr)**
  - **comm** 整数 I コミュニケータを指定する
  - **size** 整数 O comm.で指定されたグループ内に含まれるプロセス数の合計
  - **ierr** 整数 O 完了コード

```
implicit REAL*8 (A-H,O-Z)
include 'mpif.h'
integer :: PETOT, my_rank, ierr

call MPI_INIT      (ierr)
call MPI_COMM_SIZE (MPI_COMM_WORLD, PETOT, ierr )
call MPI_COMM_RANK (MPI_COMM_WORLD, my_rank, ierr )

write (*,'(a,2i8)') 'Hello World FORTRAN', my_rank, PETOT

call MPI_FINALIZE (ierr)

stop
end
```

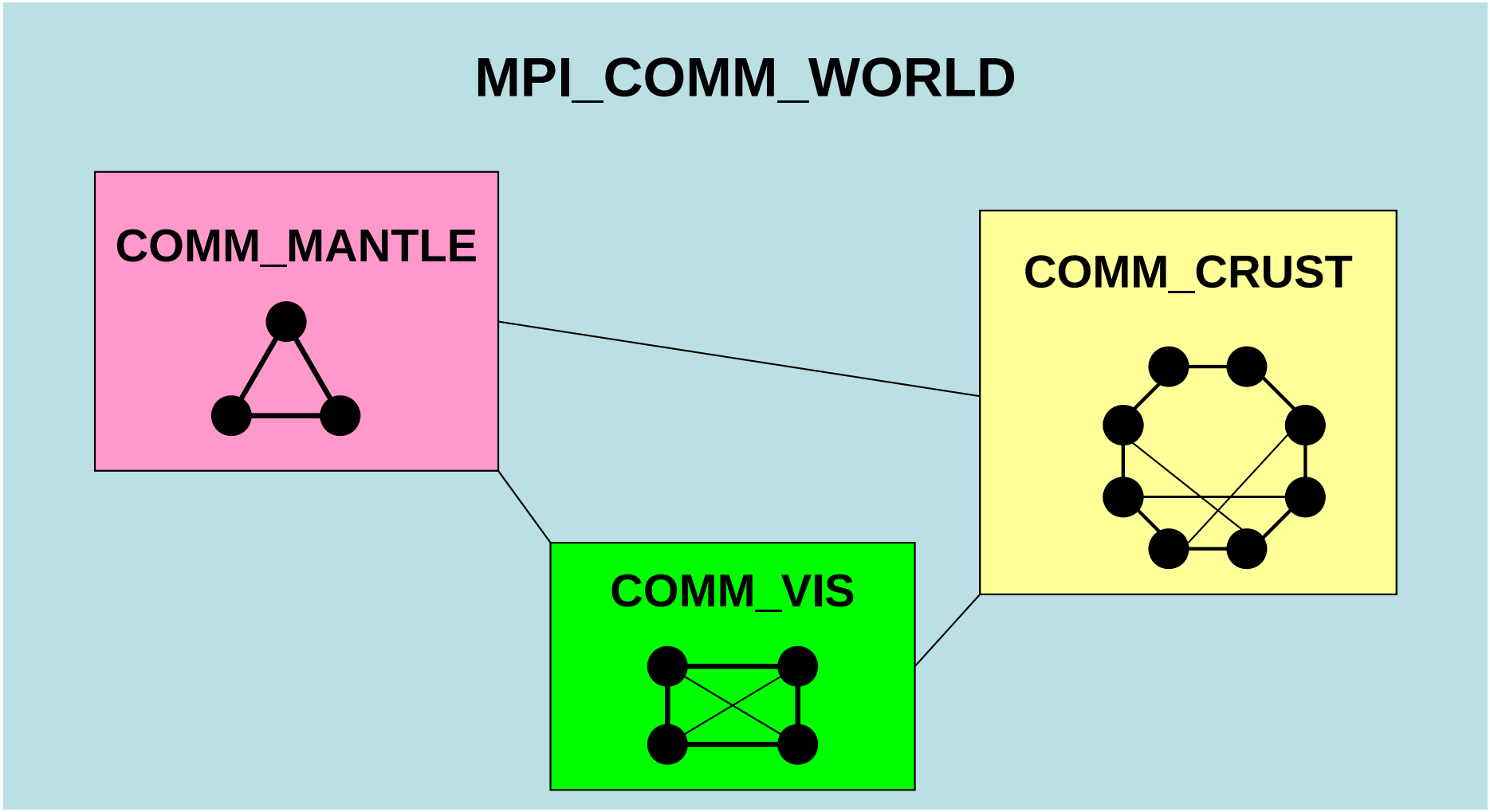
コミュニケータとは？

```
MPI_Comm_Size (MPI_COMM_WORLD, PETOT)
```

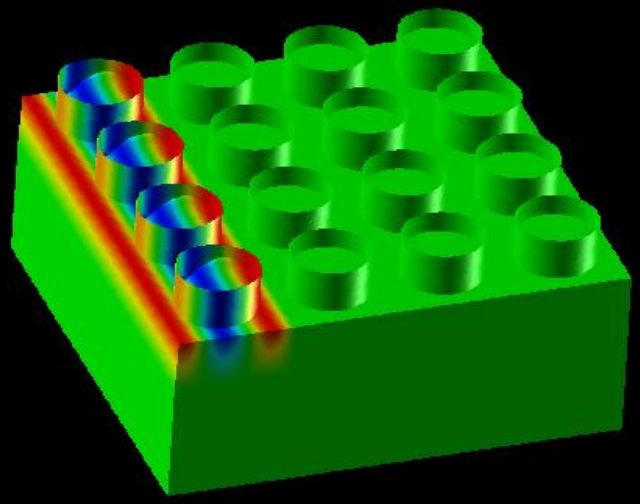
- 通信を実施するためのプロセスのグループを示す。
- MPIにおいて，通信を実施する単位として必ず指定する必要がある。
- mpirunで起動した全プロセスは，デフォルトで「**MPI_COMM_WORLD**」というコミュニケータで表されるグループに属する。
- 複数のコミュニケータを使用し，異なったプロセス数を割り当てることによって，複雑な処理を実施することも可能。
  - 例えば計算用グループ，可視化用グループ
- この授業では「**MPI_COMM_WORLD**」のみでOK。

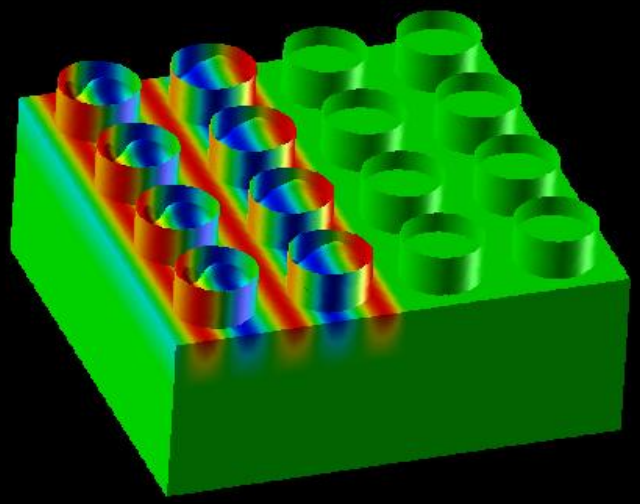
コミュニケータの概念

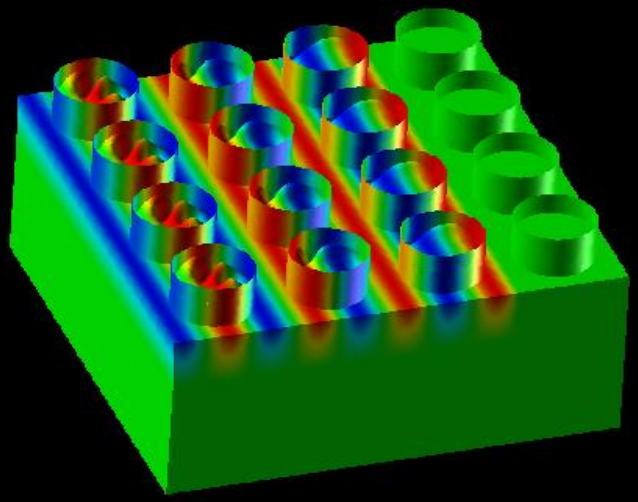
あるプロセスが複数のコミュニケータグループに属しても良い

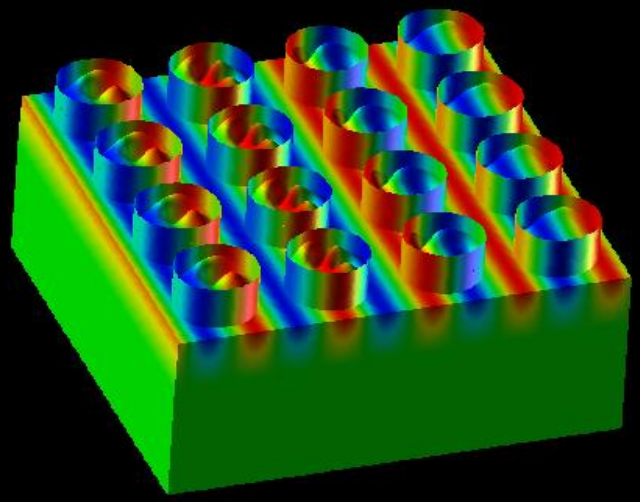


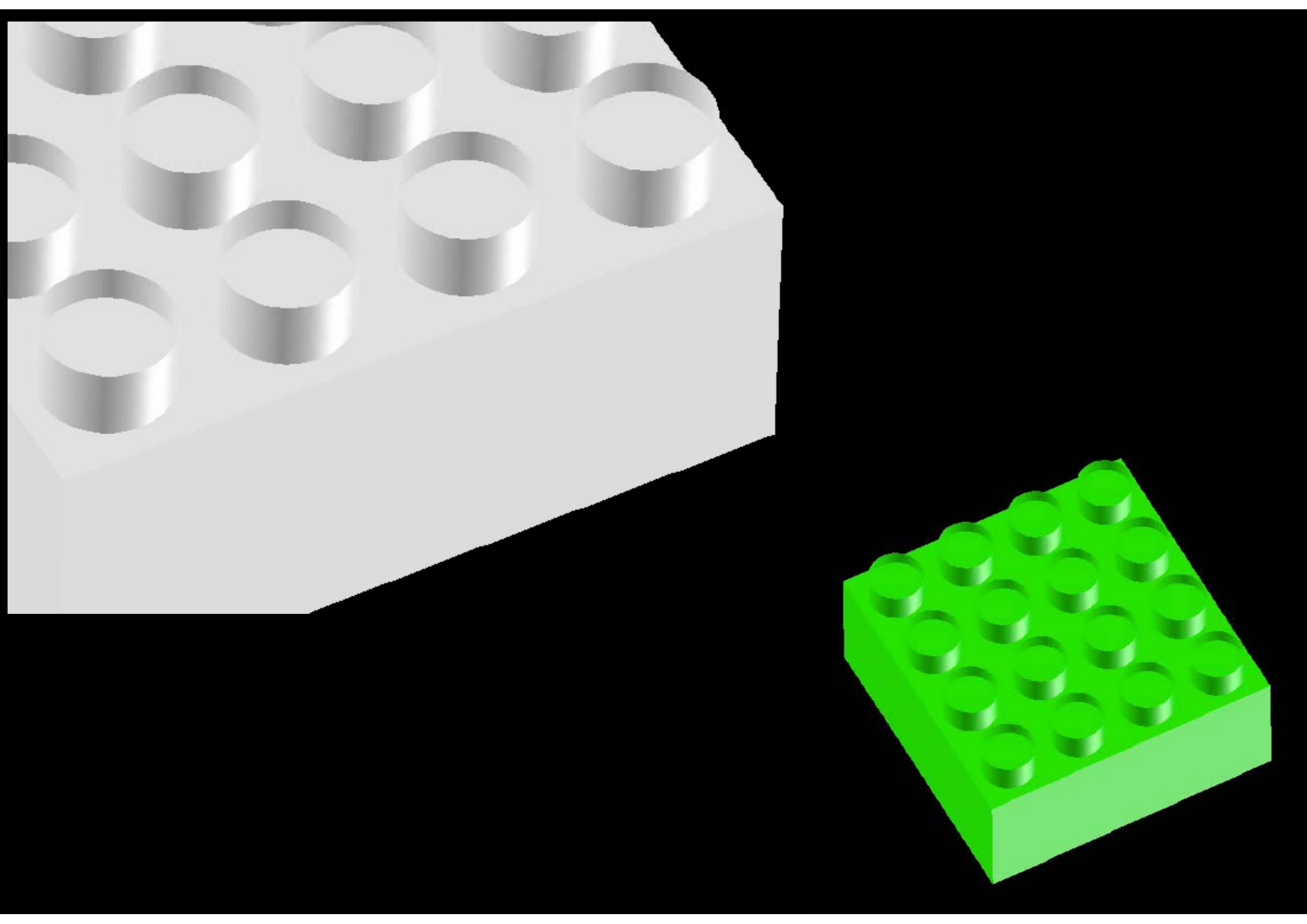
Coupling between “Ground Motion” and “Sloshing of Tanks for Oil-Storage”





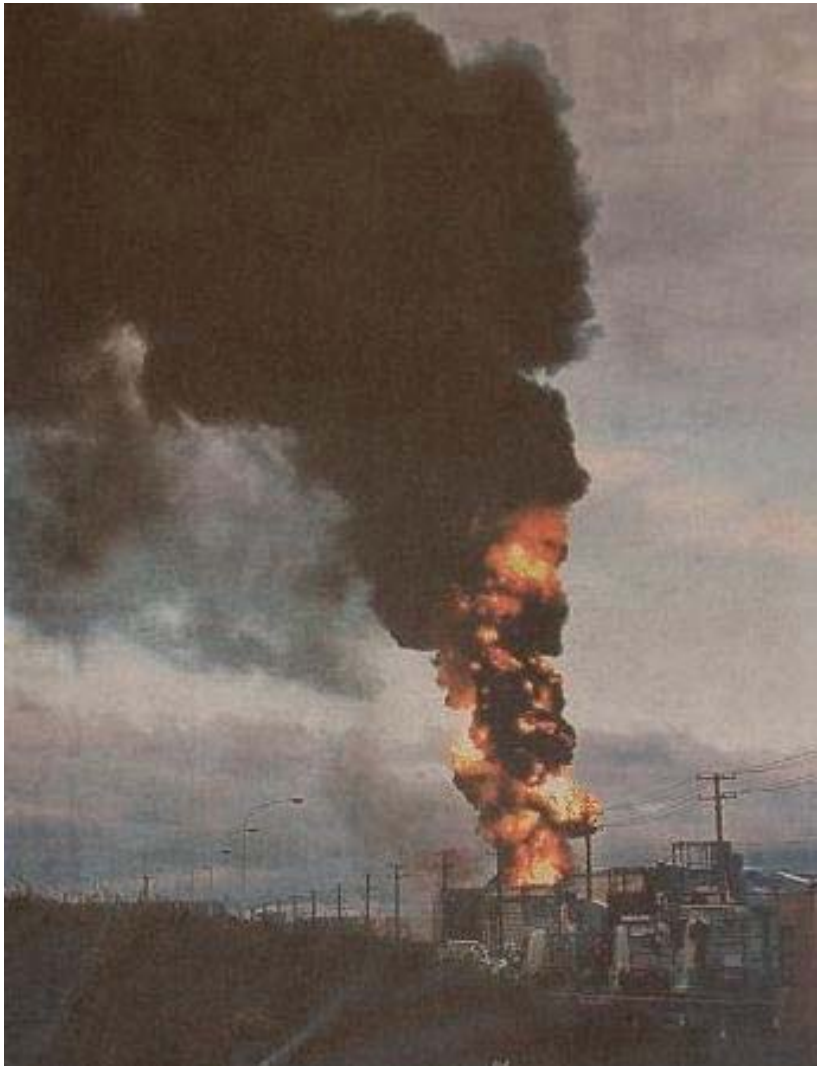


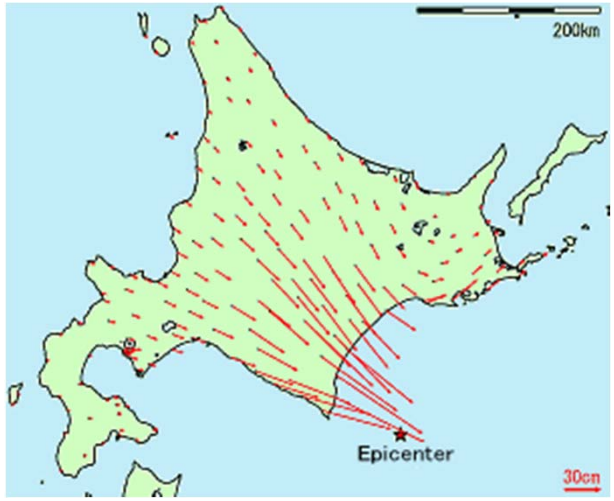




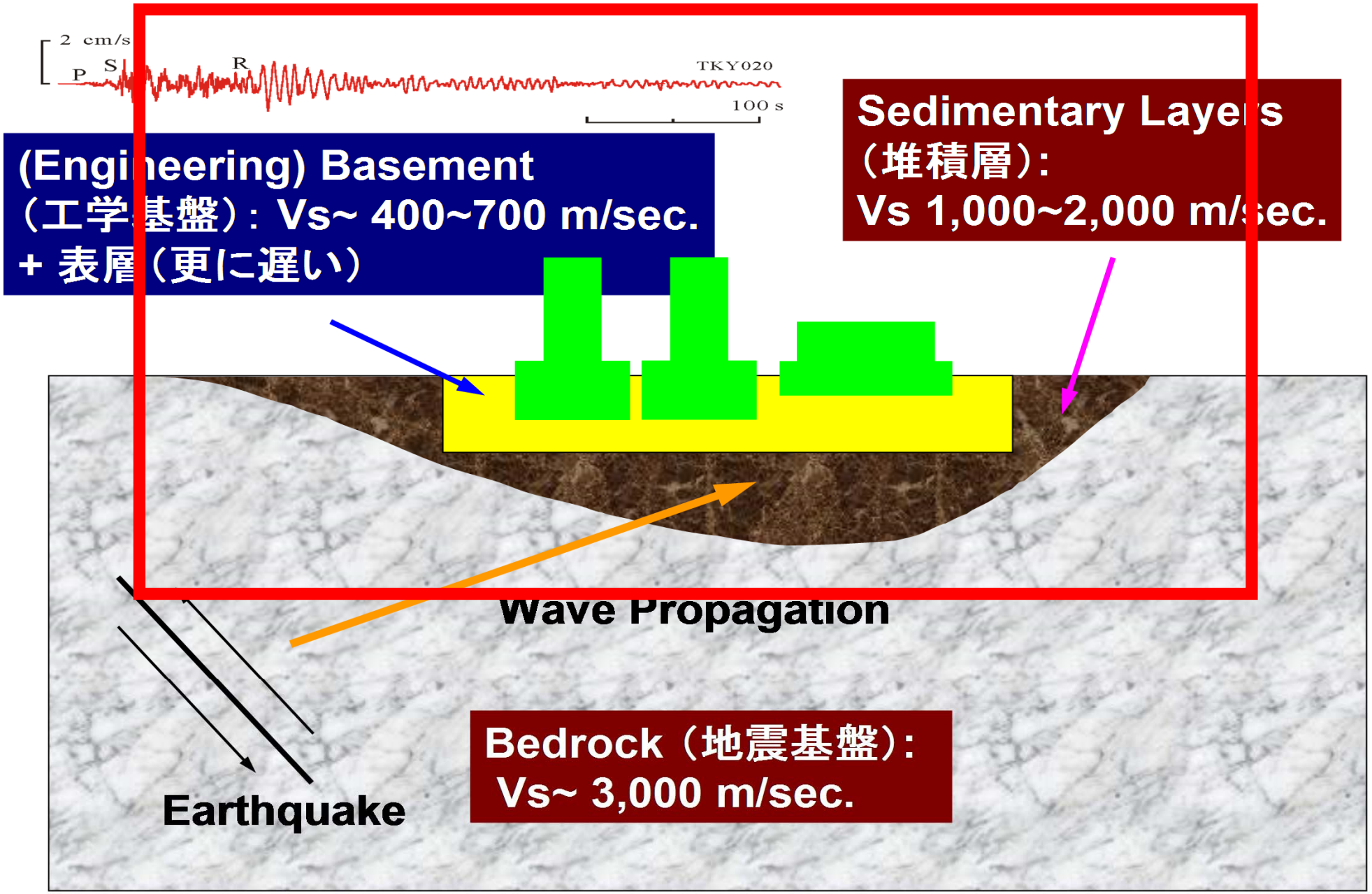
2003年　十勝沖地震

長周期地震波動（表面波）のために苫小牧の石油タンクがスロッシングを起こし火災発生



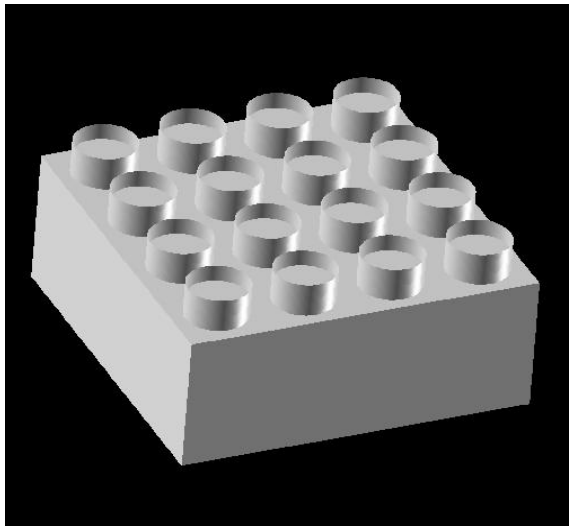


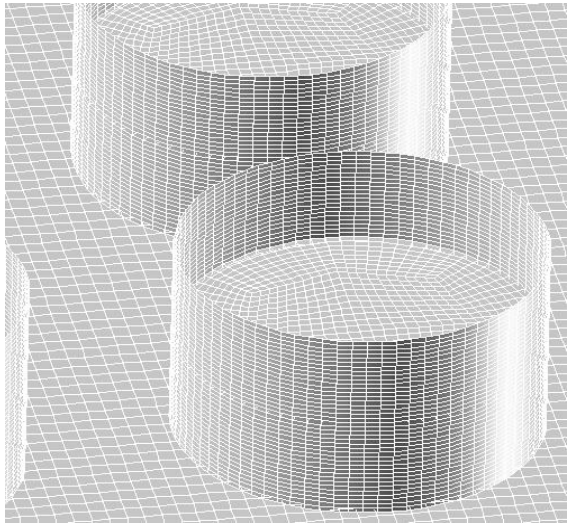
地盤・石油タンク振動連成シミュレーション



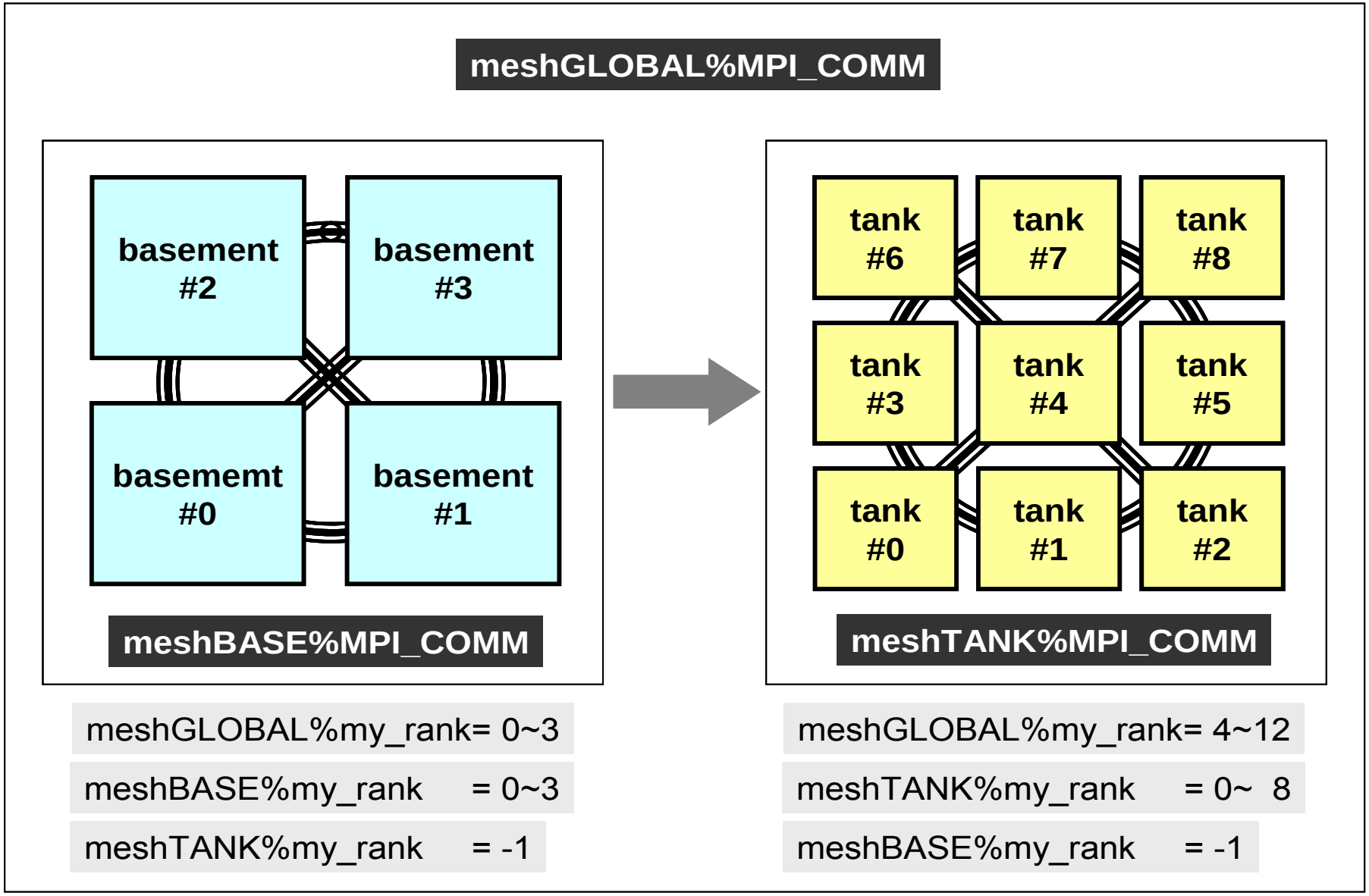
地盤，タンクモデル

- 地盤モデル（市村）FORTRAN
  - 並列FEM，三次元弾性動解析
    - 前進オイラー陽解法，EBE
  - 各要素は一辺2mの立方体
  - 240m×240m×100m
- タンクモデル（長嶋）C
  - シリアルFEM(EP)，三次元弾性動解析
    - 後退オイラー陰解法，スカイライン法
    - シェル要素＋ポテンシャル流(非粘性)
  - 直径:42.7m，高さ:24.9m，厚さ:20mm，液面:12.45m，スロッシング周期:7.6sec.
  - 周方向80分割，高さ方向:0.6m幅
  - 60m間隔で4×4に配置
- 合計自由度数:2,918,169





3種類のコミュニケータの生成



- MPI超入門
- 並列有限要素法への道
  - 局所データ構造
- 並列有限要素法のためのMPI
  - Collective Communication（集団通信）
  - Point-to-Point (Peer-to-Peer) Communication（1対1通信）

並列計算の目的

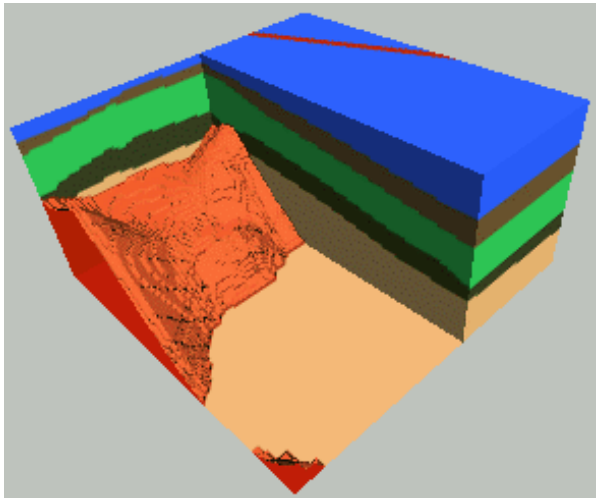
- 高速，大規模
  - 「大規模」の方が「新しい科学」という観点からのウェイトとしては高い。しかし，「高速」ももちろん重要である。
    - 細かいメッシュ
- +複雑
- 理想：Scalable
  - N倍の規模の計算をN倍のCPUを使って，「同じ時間で」解く（大規模性の追求：Weak Scaling）
    - 実際はそうは行かない
      - 例：共役勾配法⇒問題規模が大きくなると反復回数が増加
  - 同じ問題をN倍のCPUを使って「1/Nの時間で」解く・・・という場合もある（高速性の追求：Strong Scaling）
    - これも余り簡単な話では無い

並列計算とは？(1/2)

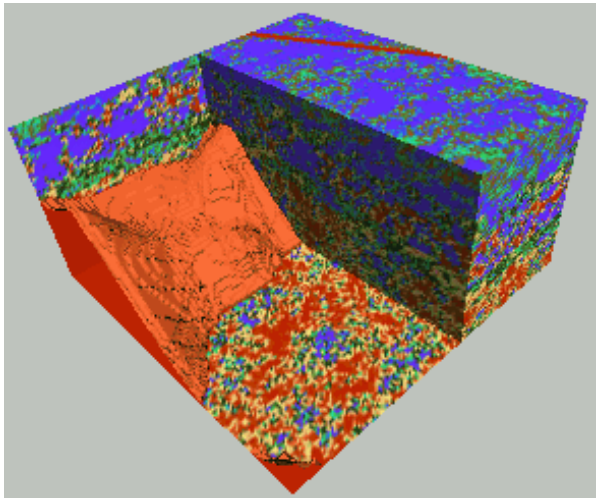
- より大規模で複雑な問題を高速に解きたい

Homogeneous/Heterogeneous Porous Media

Lawrence Livermore National Laboratory



Homogeneous

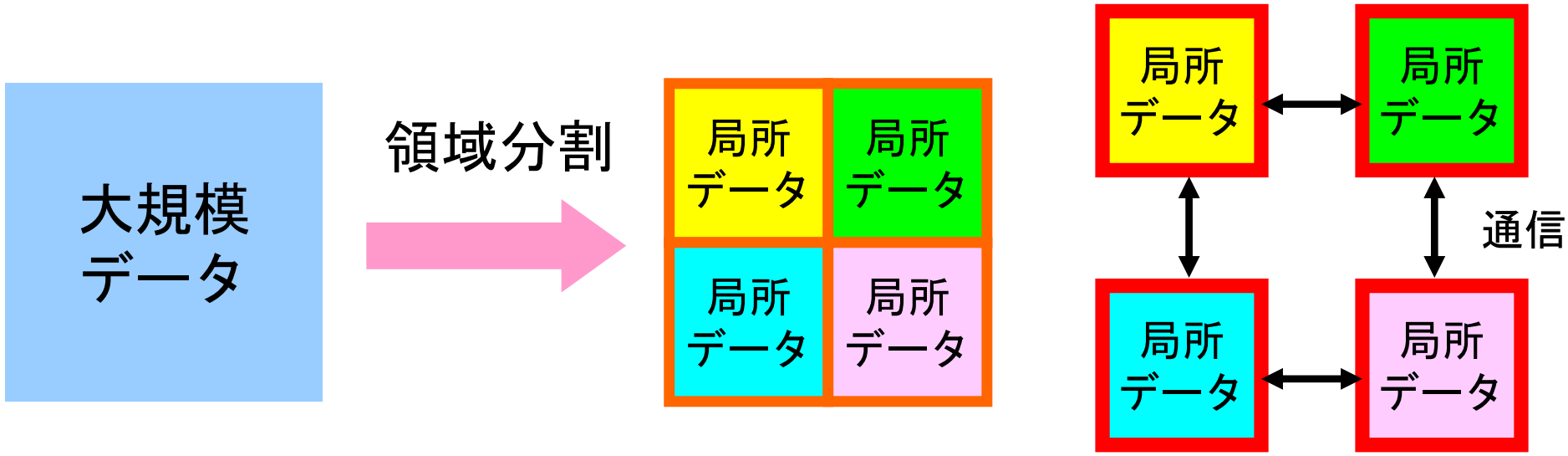


Heterogeneous

このように不均質な場を模擬するには非常に細かいメッシュが必要となる

並列計算とは？(2/2)

- 1GB程度のPC → <10^6メッシュが限界：FEM
  - 1000km×1000km×100kmの領域（西南日本）を1kmメッシュで切ると10^8メッシュになる
- 大規模データ → 領域分割，局所データ並列処理
- 全体系計算 → 領域間の通信が必要



通信とは？

- 並列計算とはデータと処理をできるだけ「局所的（local）」に実施すること。
  - 要素マトリクスの計算
  - 有限要素法の計算は本来並列計算向けである

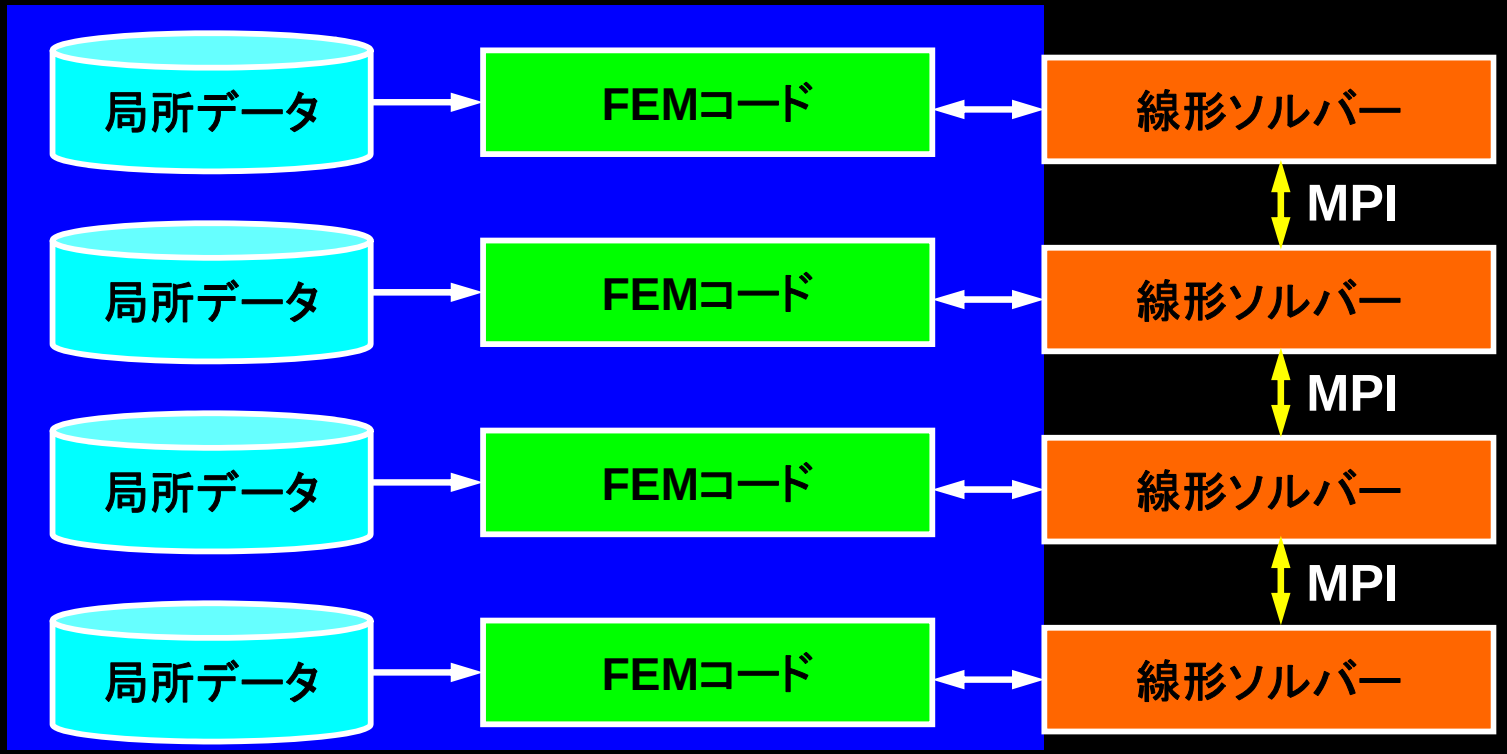
- 「大域的（global）」な情報を必要とする場合に通信が生じる（必要となる）。
  - 全体マトリクスを線形ソルバーで解く

並列有限要素法の処理：SPMD

巨大な解析対象 → 局所分散データ，領域分割（Partitioning）

有限要素コードは領域ごとに係数マトリクスを生成：要素単位の処理によって可能：シリアルコードと変わらない処理

グローバル処理，通信は線形ソルバーのみで生じる
　内積，行列ベクトル積，前処理

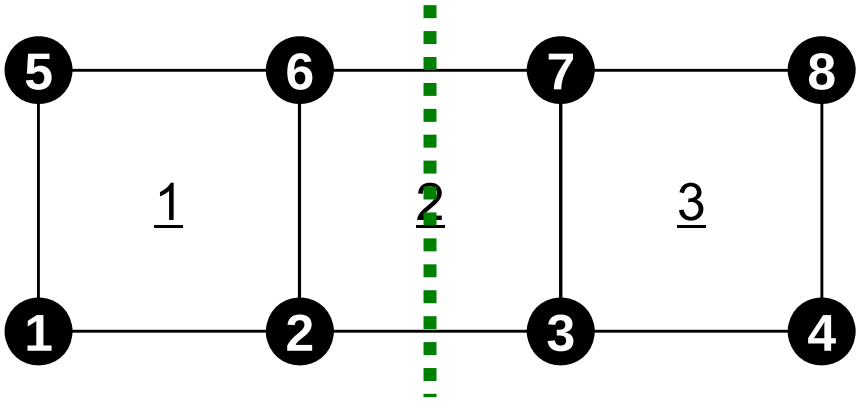


並列有限要素法プログラムの開発

- 前頁のようなオペレーションを実現するためのデータ構造が重要
  - アプリケーションの「並列化」にあたって重要なのは，適切な局所分散データ構造の設計である。

- 前処理付反復法
- マトリクス生成：ローカルに処理

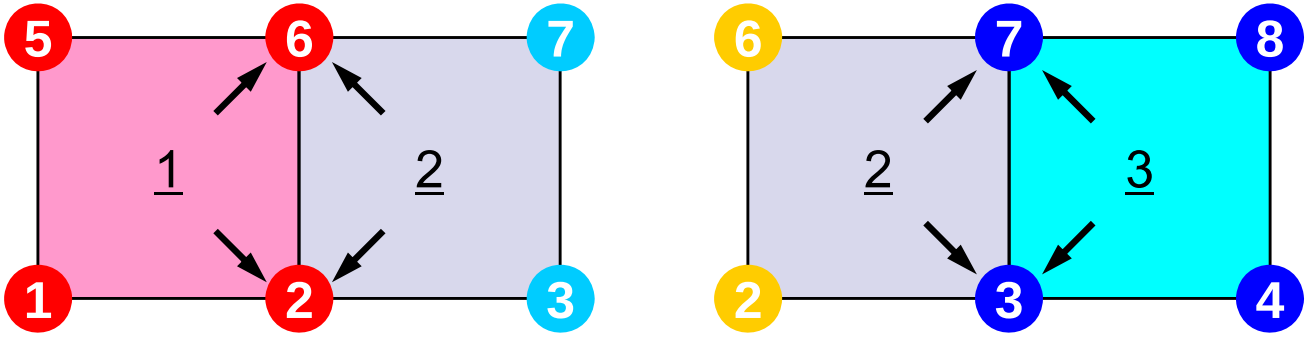
四角形要素



「節点ベース（領域ごとの節点数がバランスする）」の分割

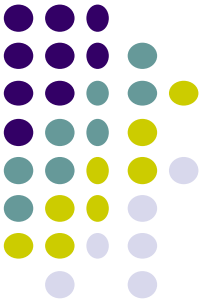
自由度：節点上で定義

これではマトリクス生成に必要な情報は不十分



マトリクス生成のためには，オーバーラップ部分の要素と節点の情報が必要

並列有限要素法の局所データ構造

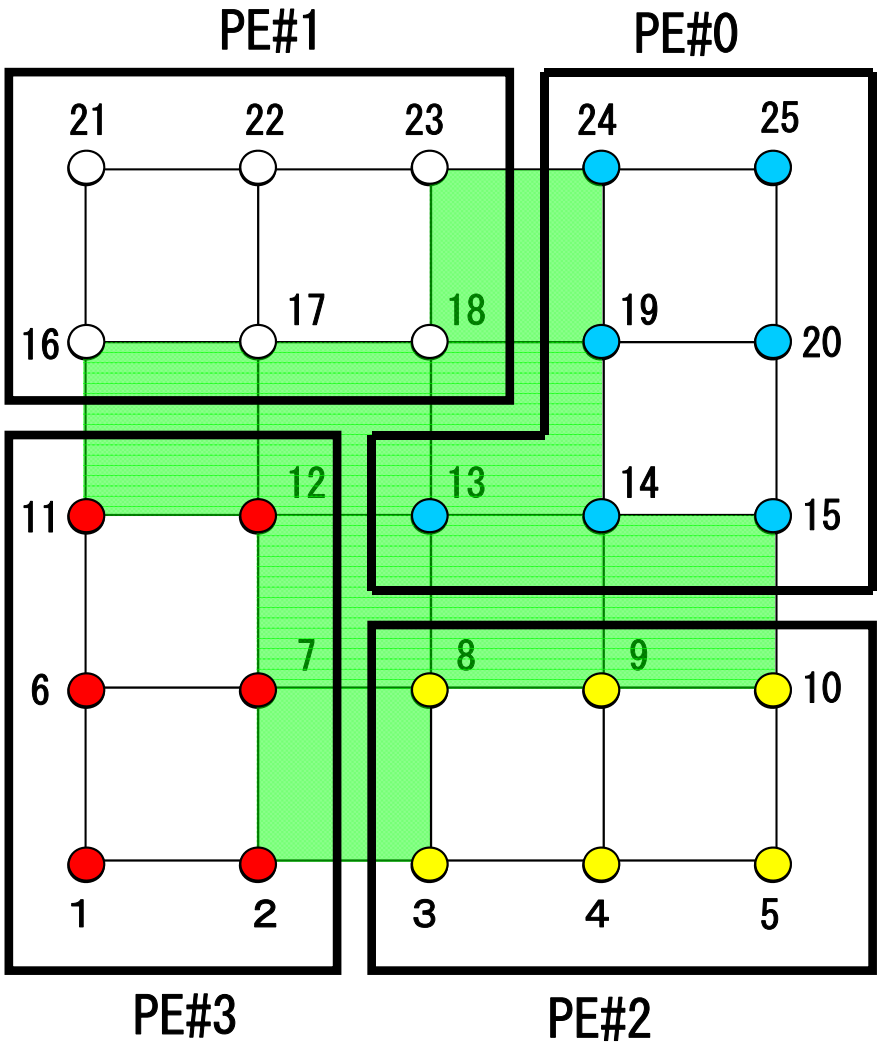


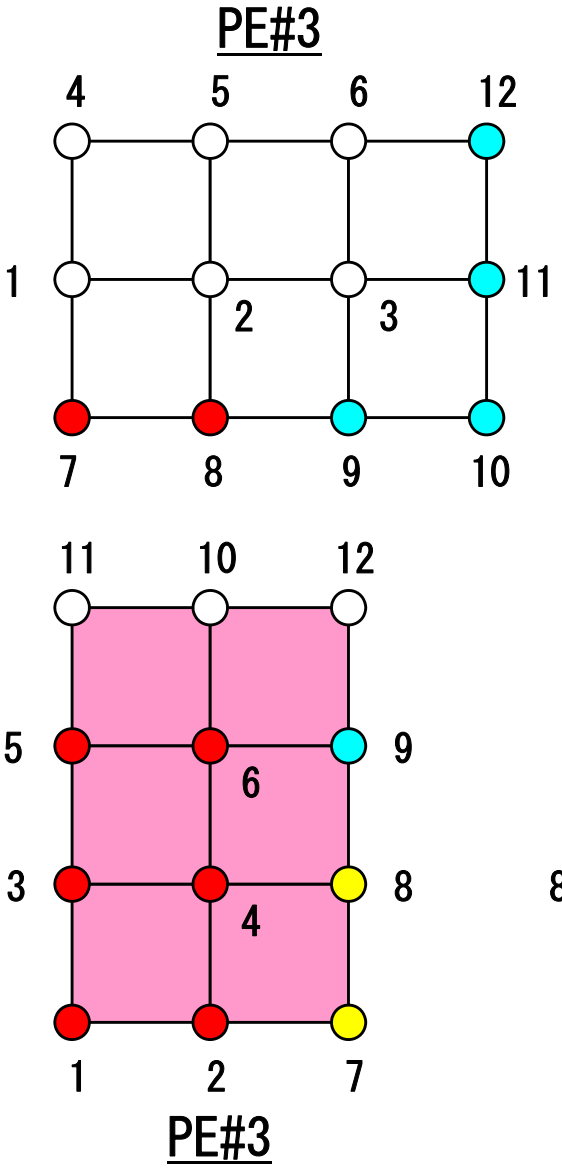
- 節点ベース：Node-based partitioning
- 局所データに含まれるもの：
  - その領域に本来含まれる節点
  - それらの節点を含む要素
  - 本来領域外であるが，それらの要素に含まれる節点
- 節点は以下の3種類に分類
  - 内点：Internal nodes　　その領域に本来含まれる節点
  - 外点：External nodes　　本来領域外であるがマトリクス生成に必要な節点
  - 境界点：Boundary nodes　　他の領域の「外点」となっている節点
- 領域間の通信テーブル

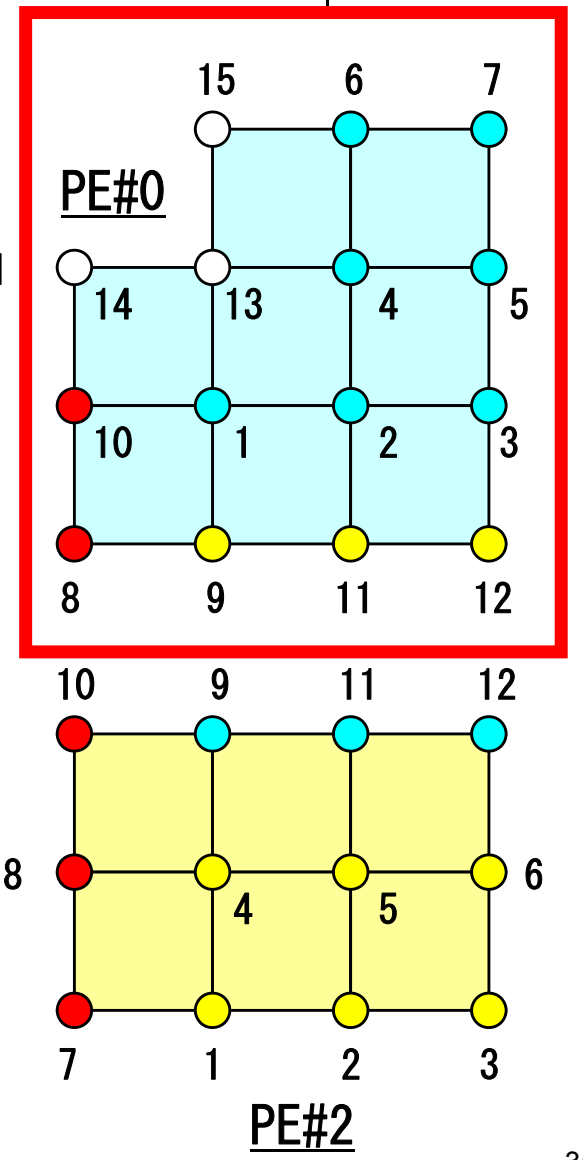
- 領域間の接続をのぞくと，大域的な情報は不要
  - 有限要素法の特性：要素で閉じた計算

Node-based Partitioning

internal nodes - elements - external nodes





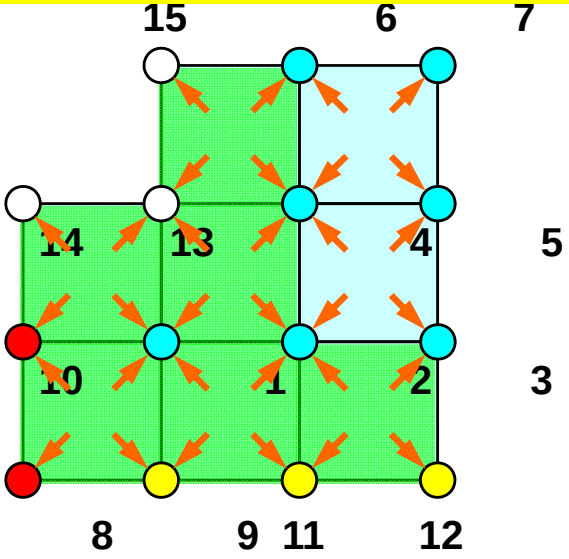


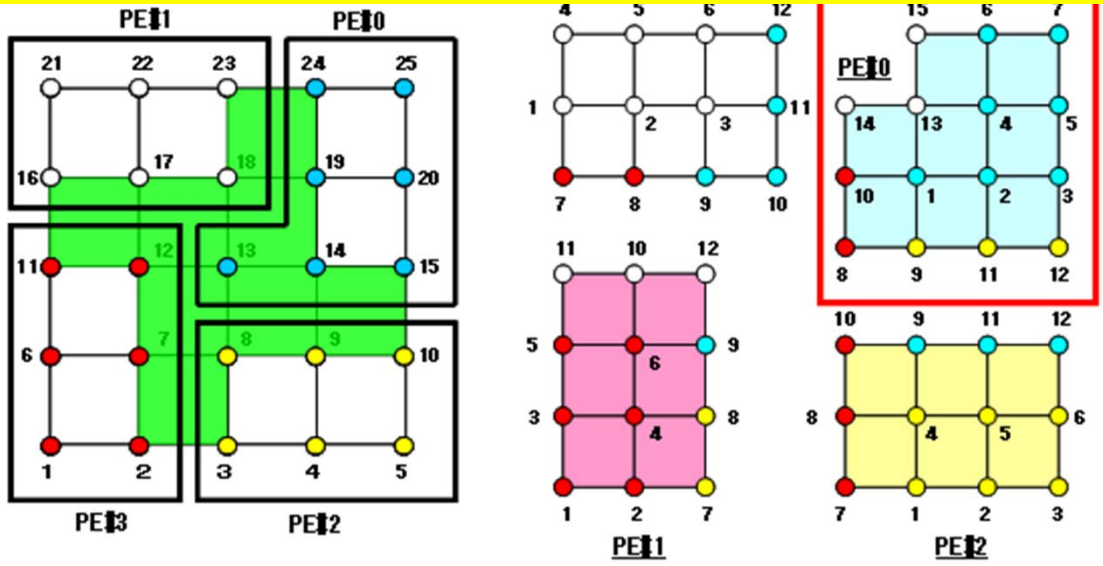
Node-based Partitioning

internal nodes - elements - external nodes

- Partitioned nodes themselves (Internal Nodes) 内点
- Elements which include Internal Nodes 内点を含む要素
- External Nodes included in the Elements 外点

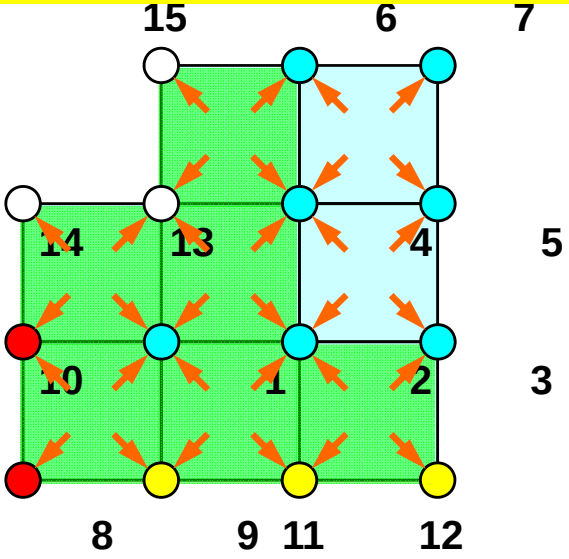
  in overlapped region among partitions.
- Info of External Nodes are required for completely local element–based operations on each processor.

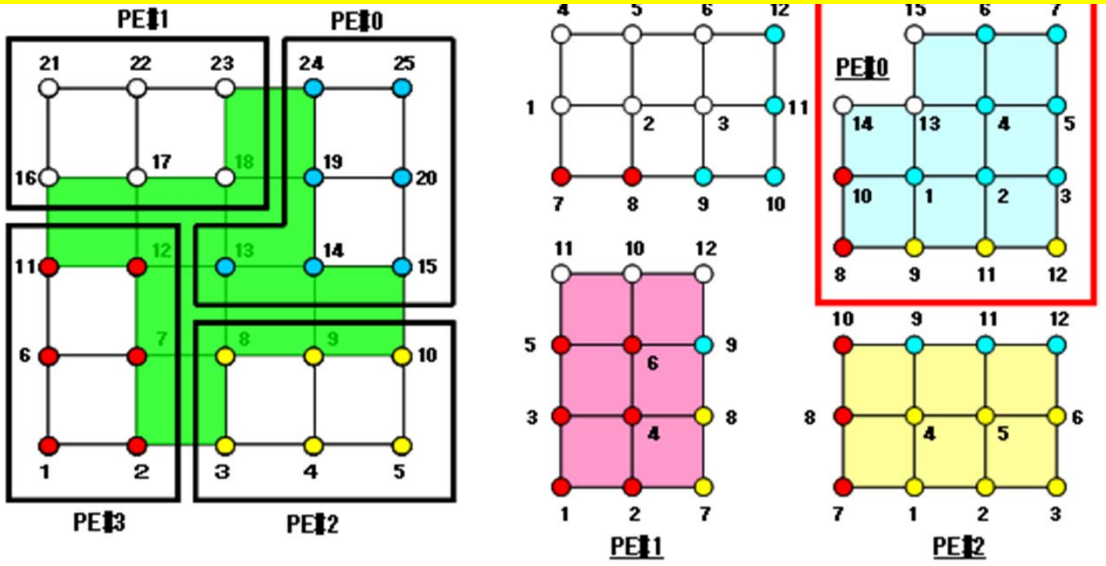




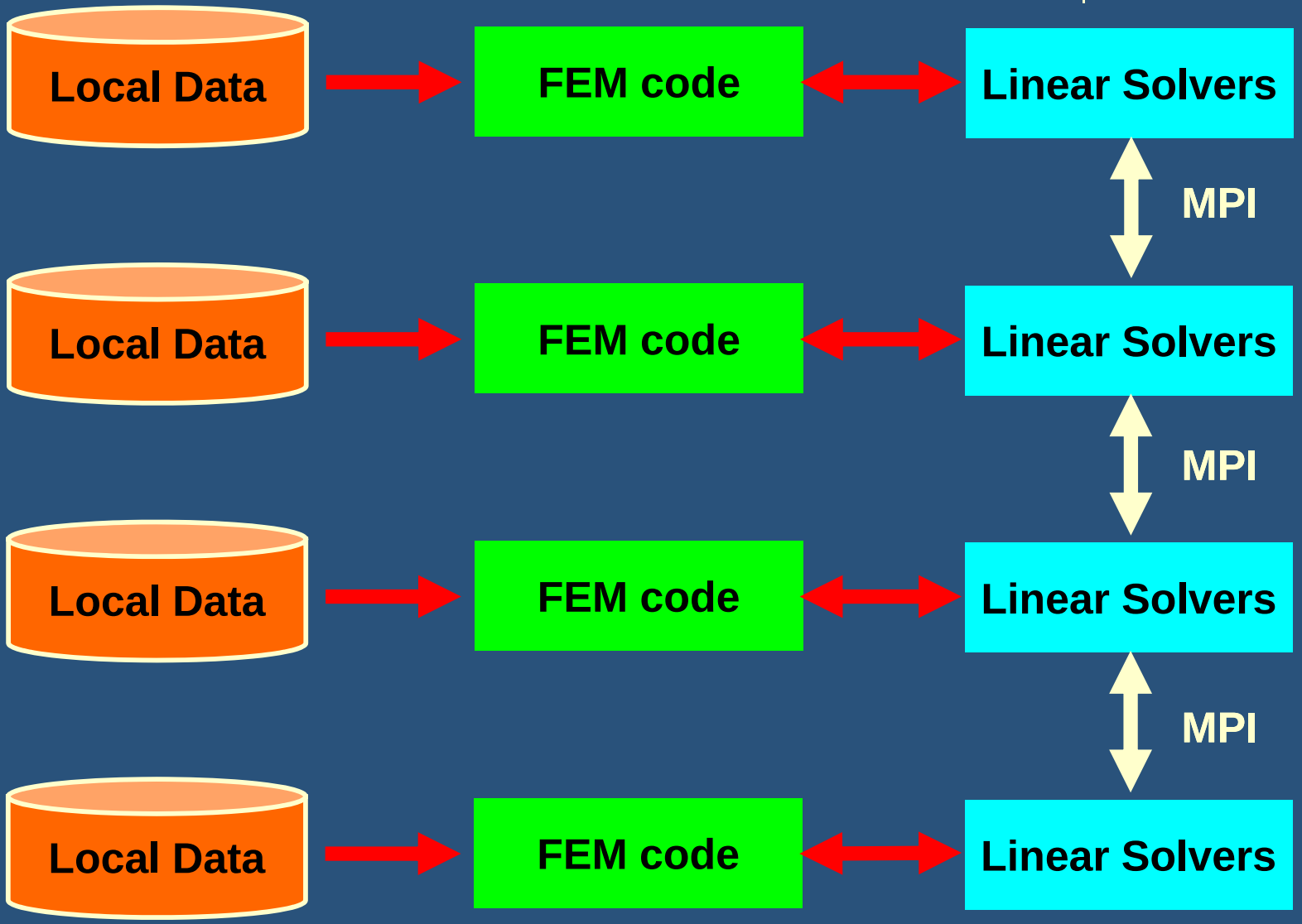
マトリクス生成時の通信は不要

- Partitioned nodes themselves (Internal Nodes)　内点
- Elements which include Internal Nodes　内点を含む要素
- External Nodes included in the Elements　外点
  in overlapped region among partitions.
- Info of External Nodes are required for completely local element–based operations on each processor.

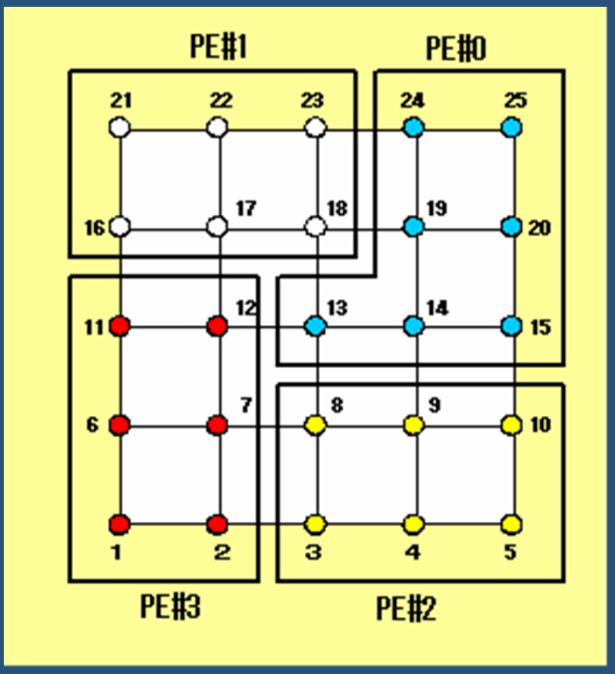




Parallel Computing in FEM
SPMD: Single-Program Multiple-Data



Parallel Computing in FEM
SPMD: Single-Program Multiple-Data



Local Data → FEM code ↔ Linear Solvers

MPI

Local Data → FEM code ↔ Linear Solvers

MPI

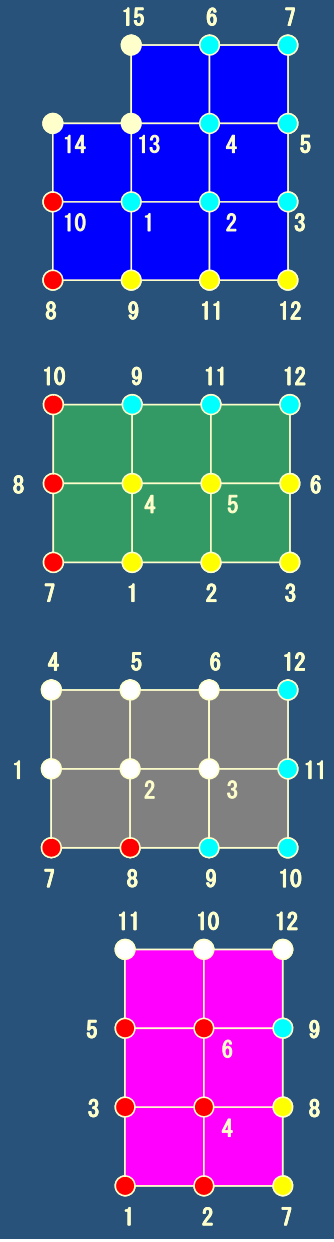
Local Data → FEM code ↔ Linear Solvers

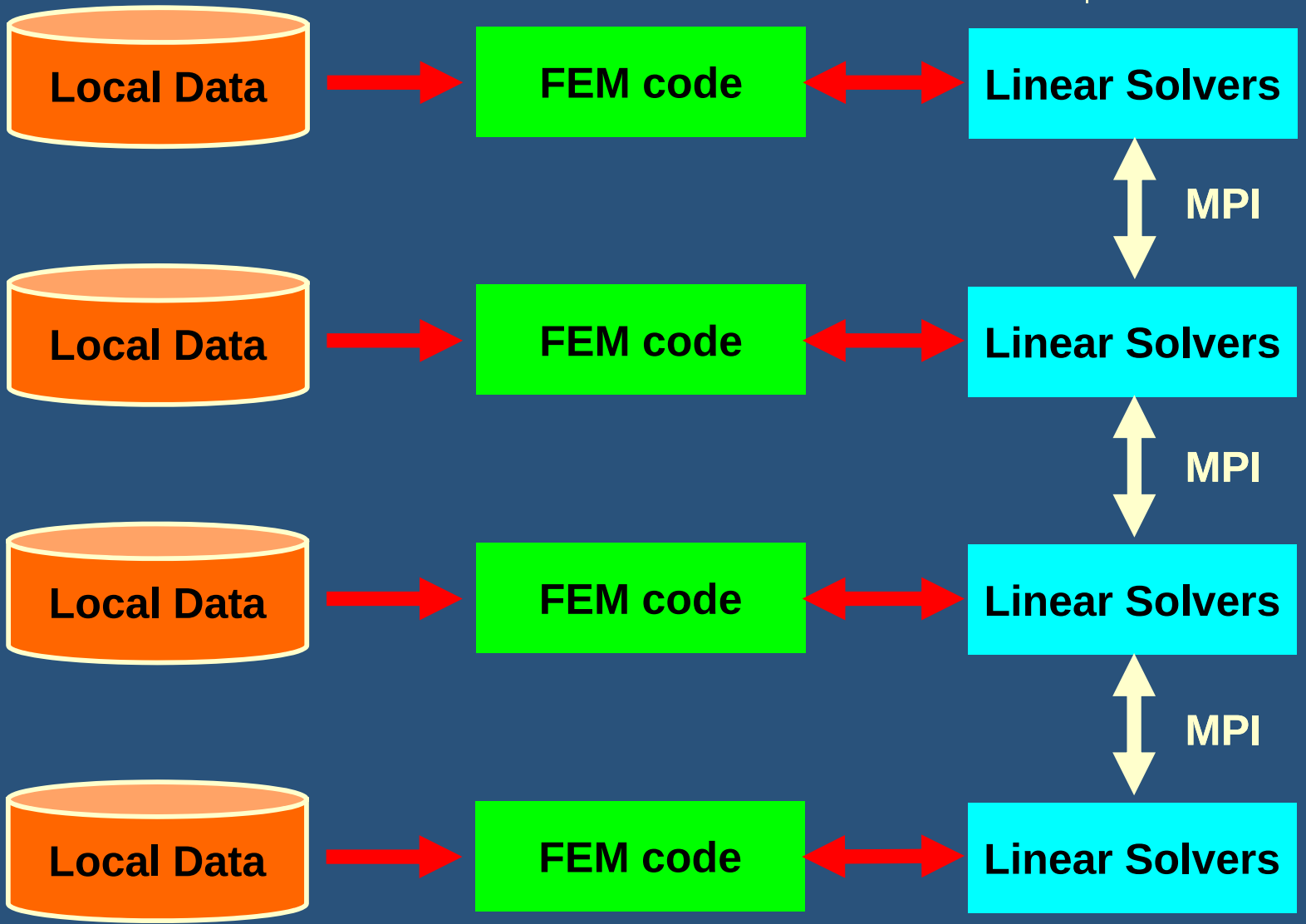
MPI

Local Data → FEM code ↔ Linear Solvers

Parallel Computing in FEM

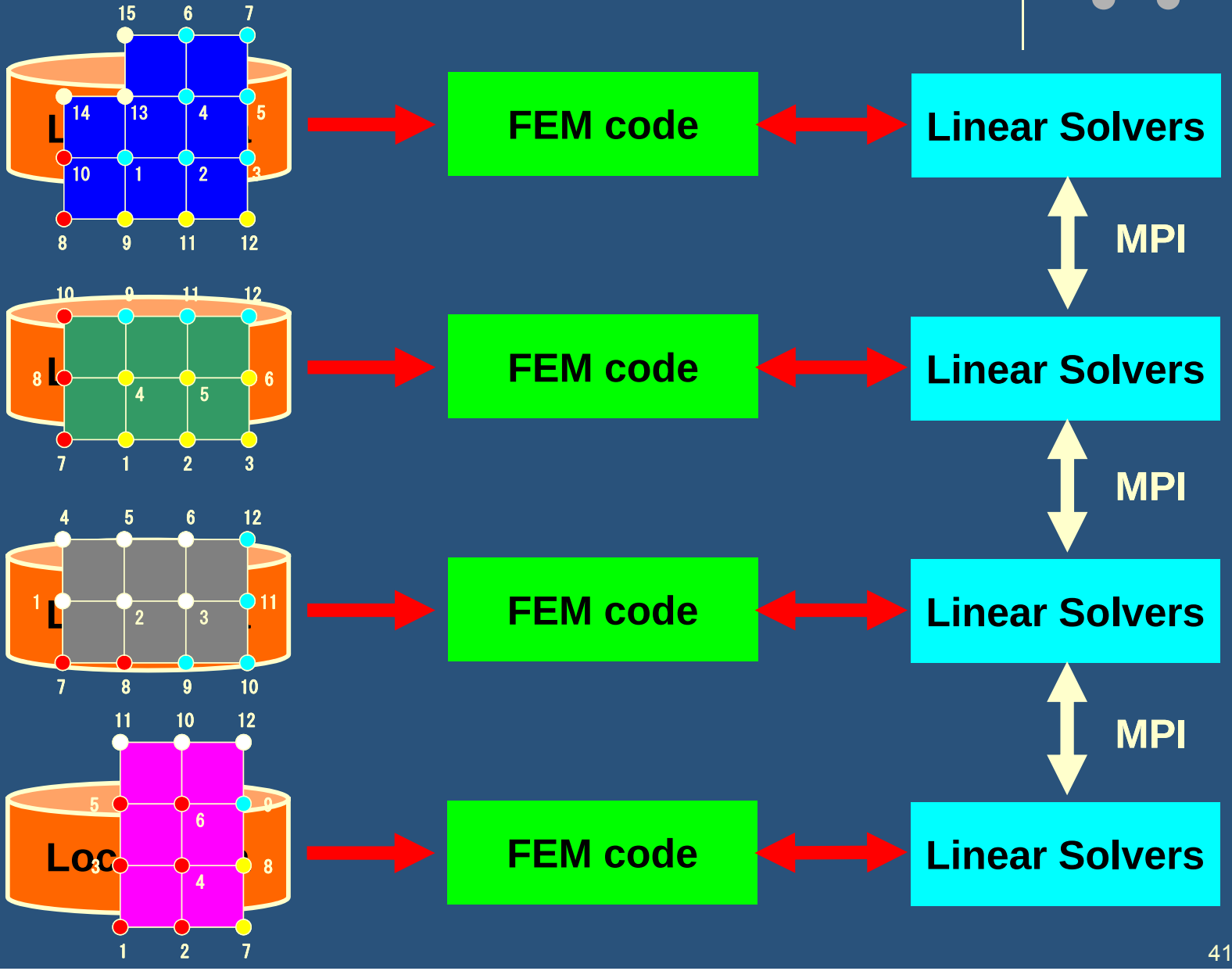
SPMD: Single-Program Multiple-Data





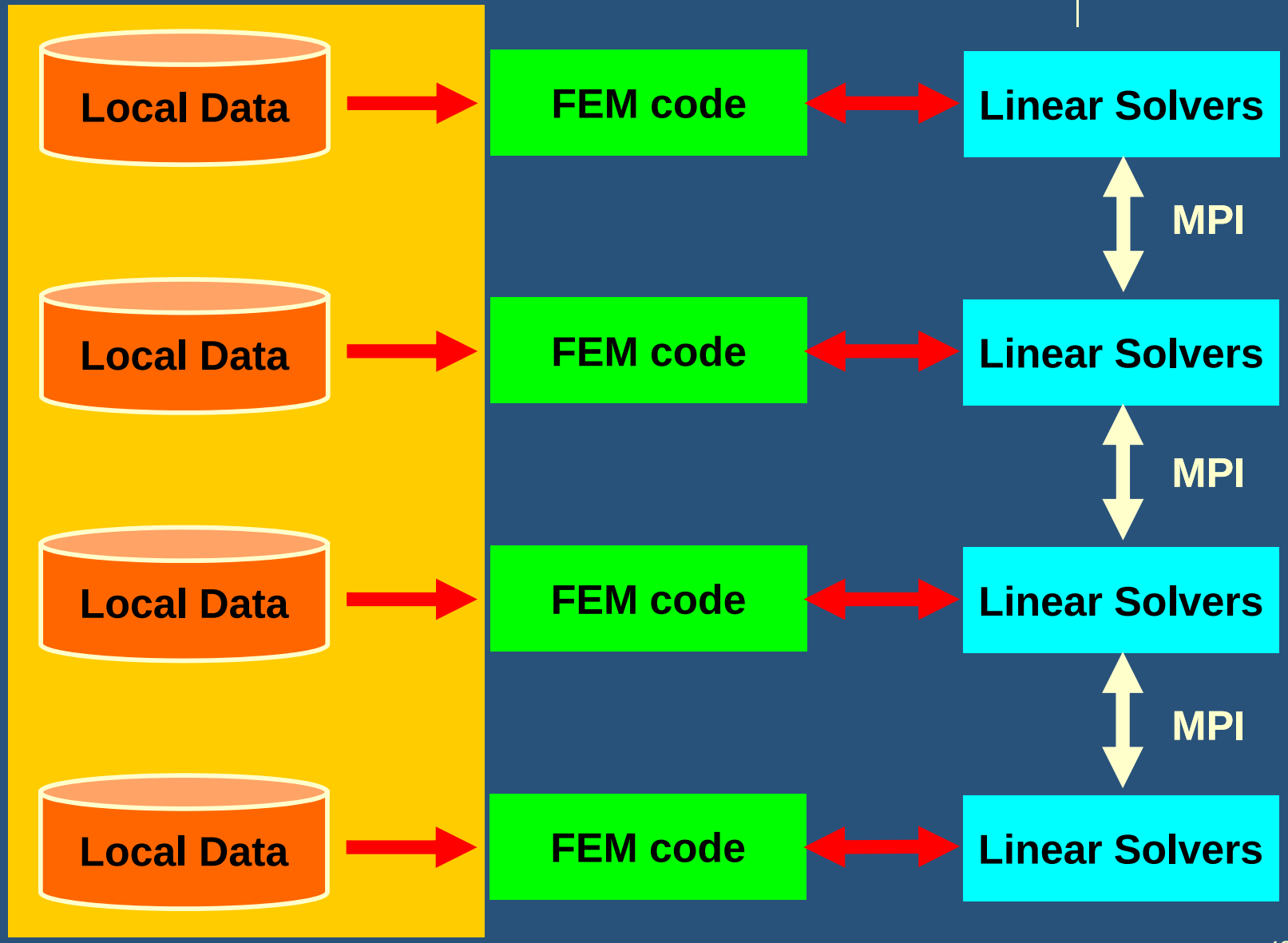
Parallel Computing in FEM

SPMD: Single-Program Multiple-Data

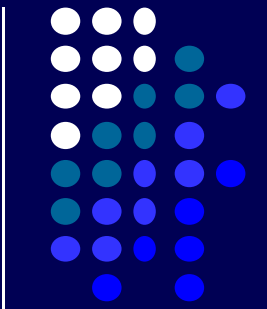


Parallel Computing in FEM

SPMD: Single-Program Multiple-Data

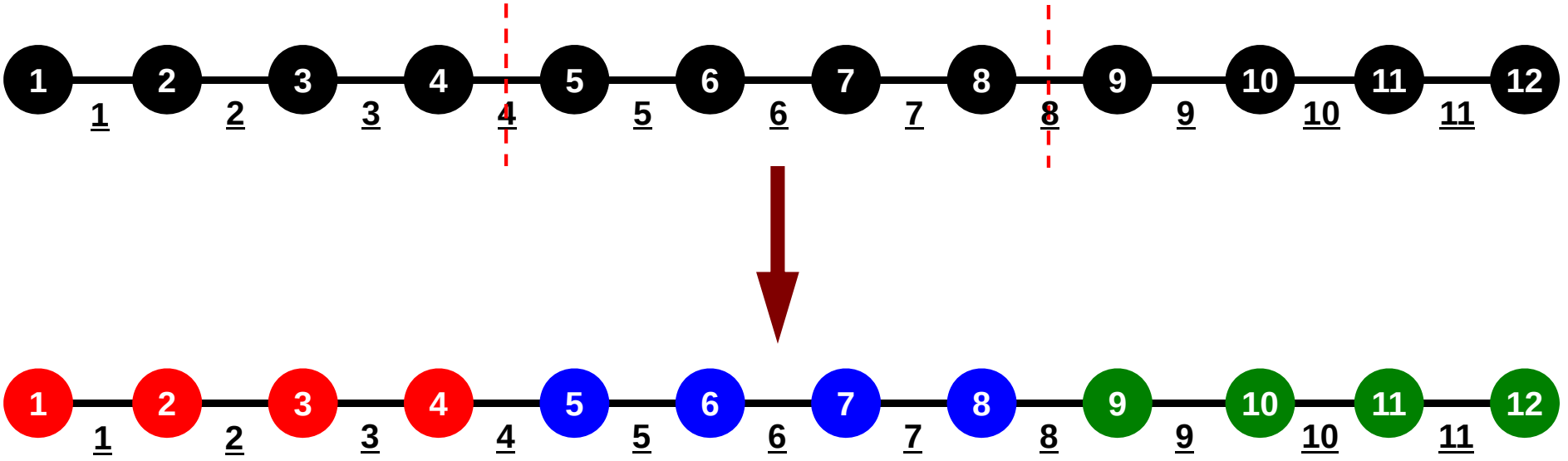


通信とは何か？

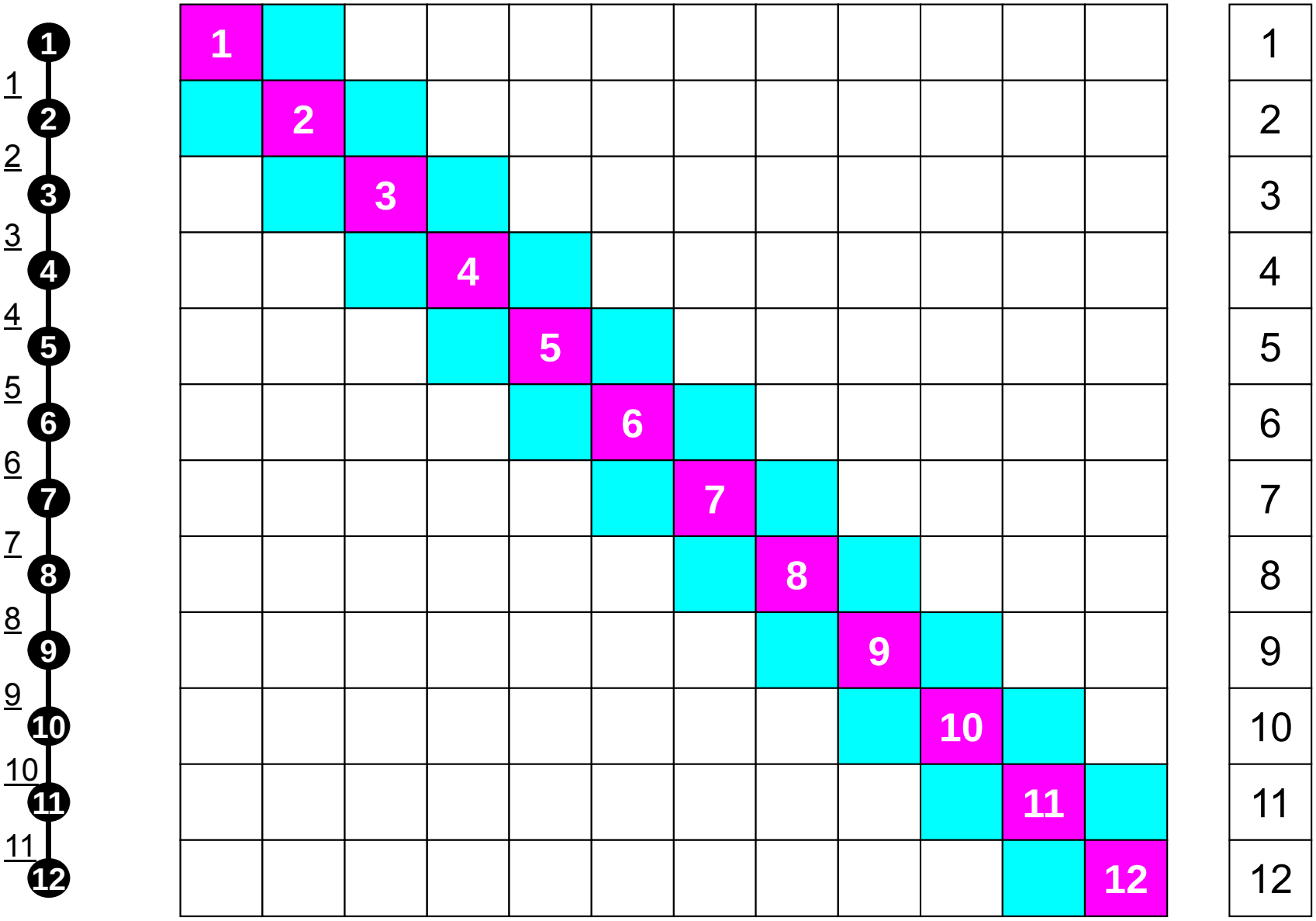


- 「外点」の情報を外部の領域からもらってくること
- 「通信テーブル」にその情報が含まれている

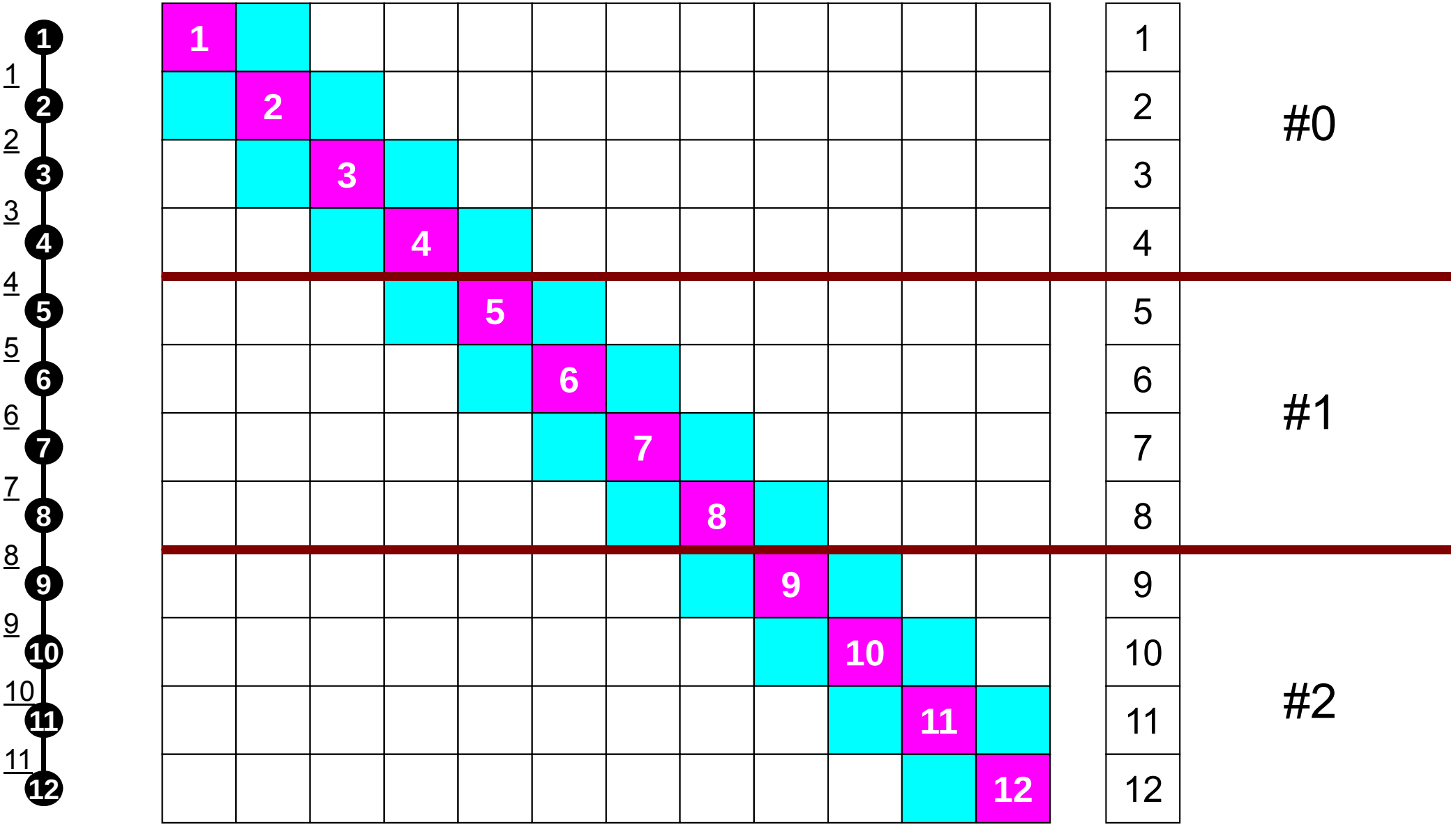
一次元問題:11要素，12節点，3領域



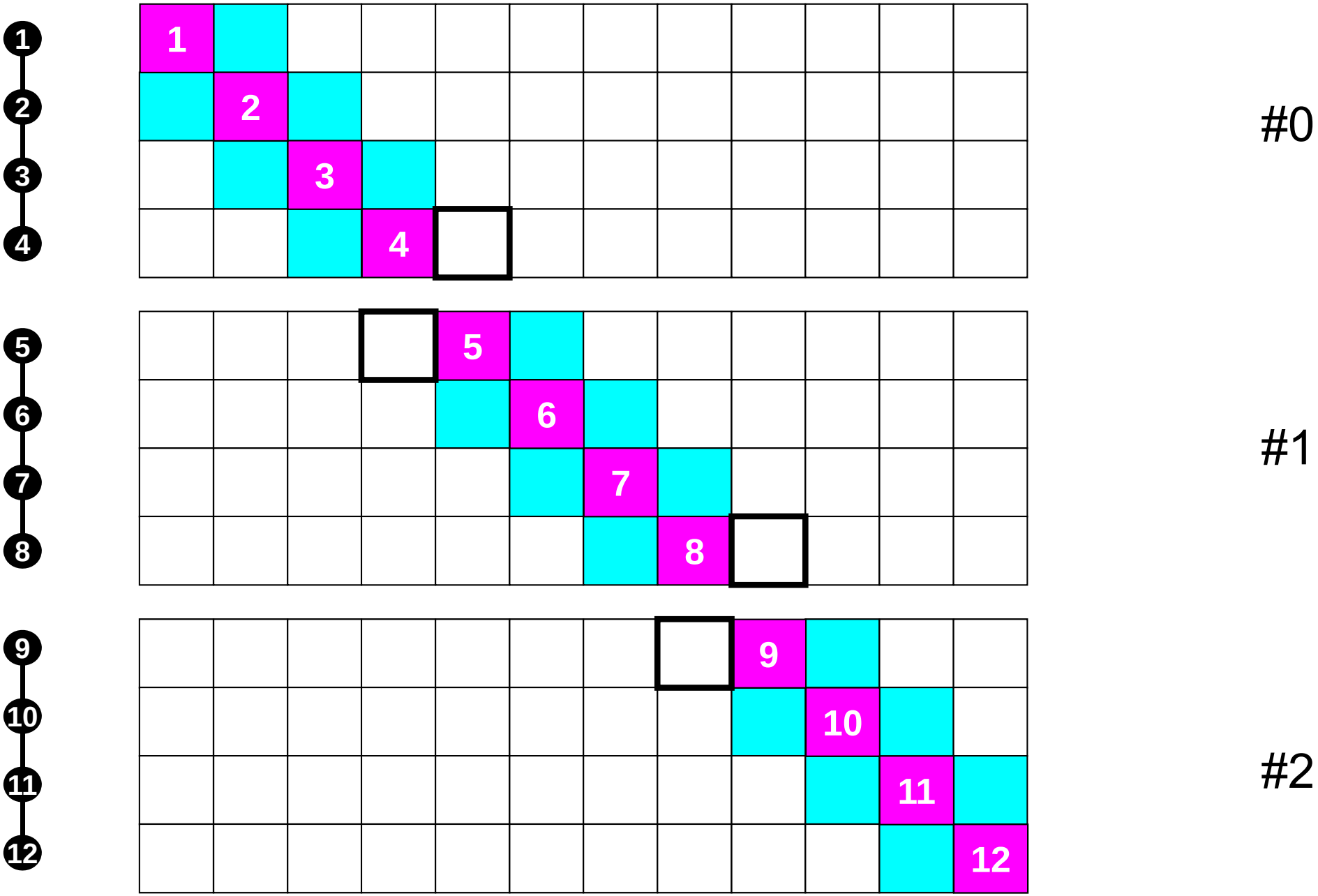
一次元問題:11要素，12節点，3領域



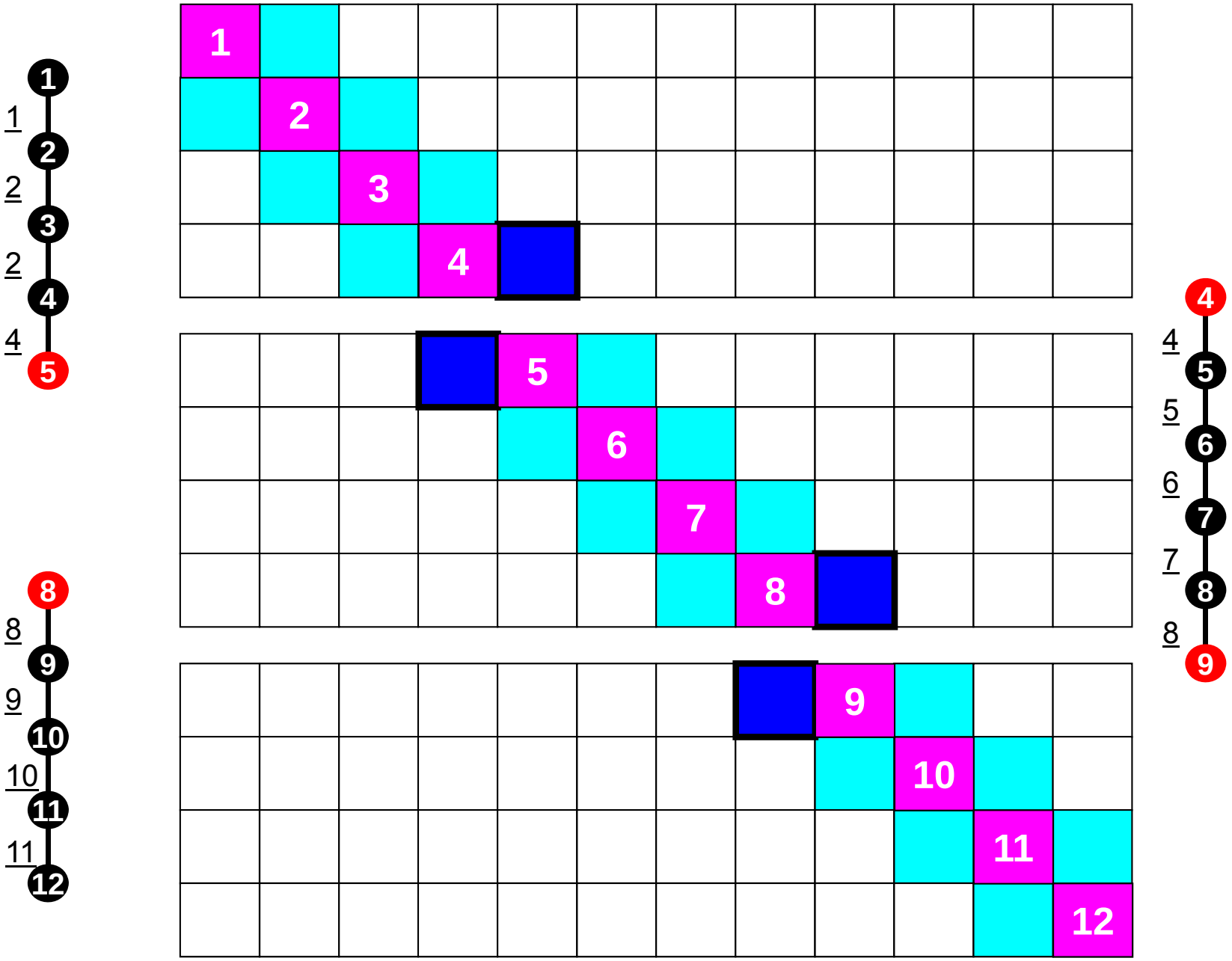
節点がバランスするよう分割：内点



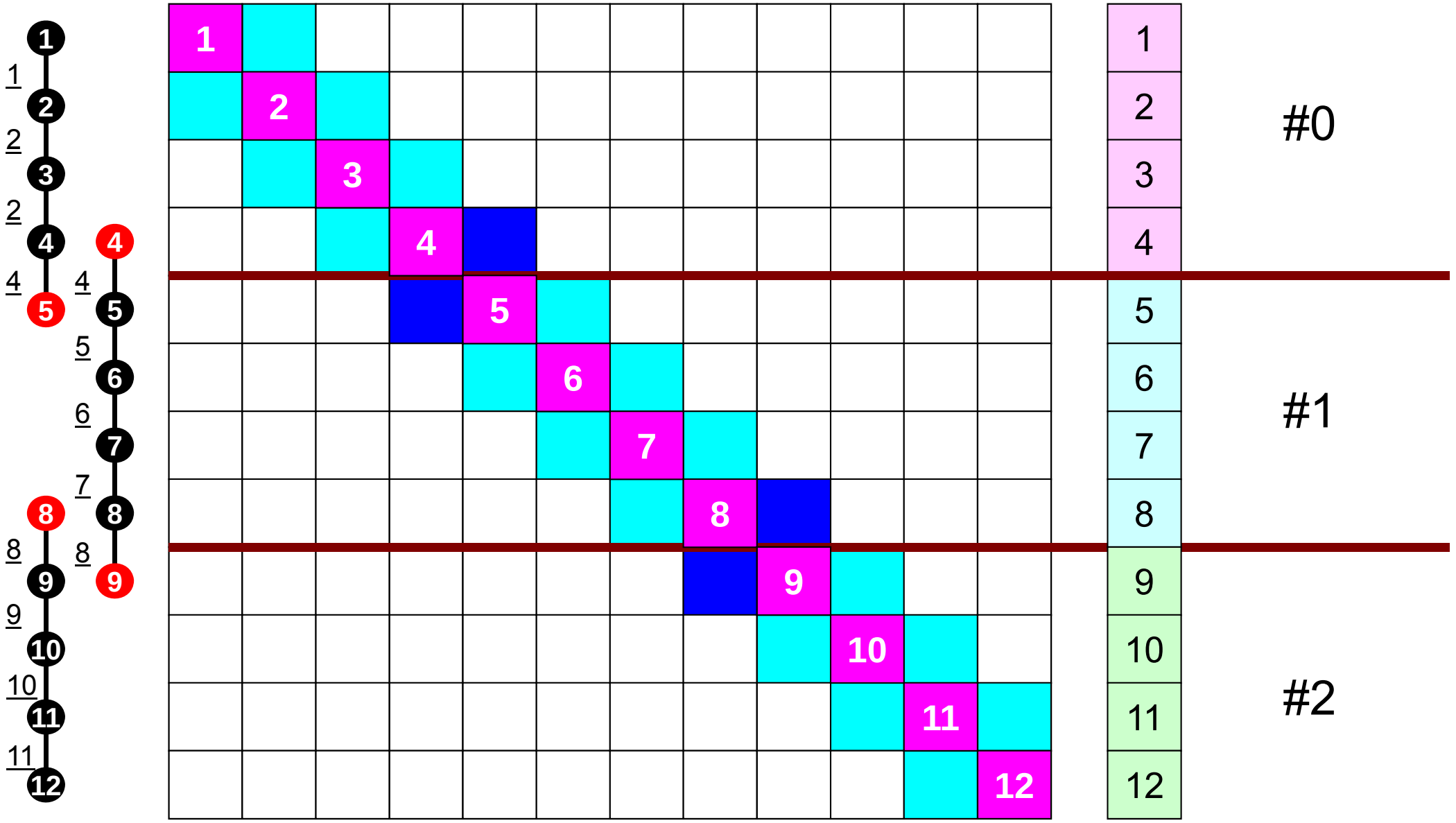
内点だけで分割するとマトリクス不完全



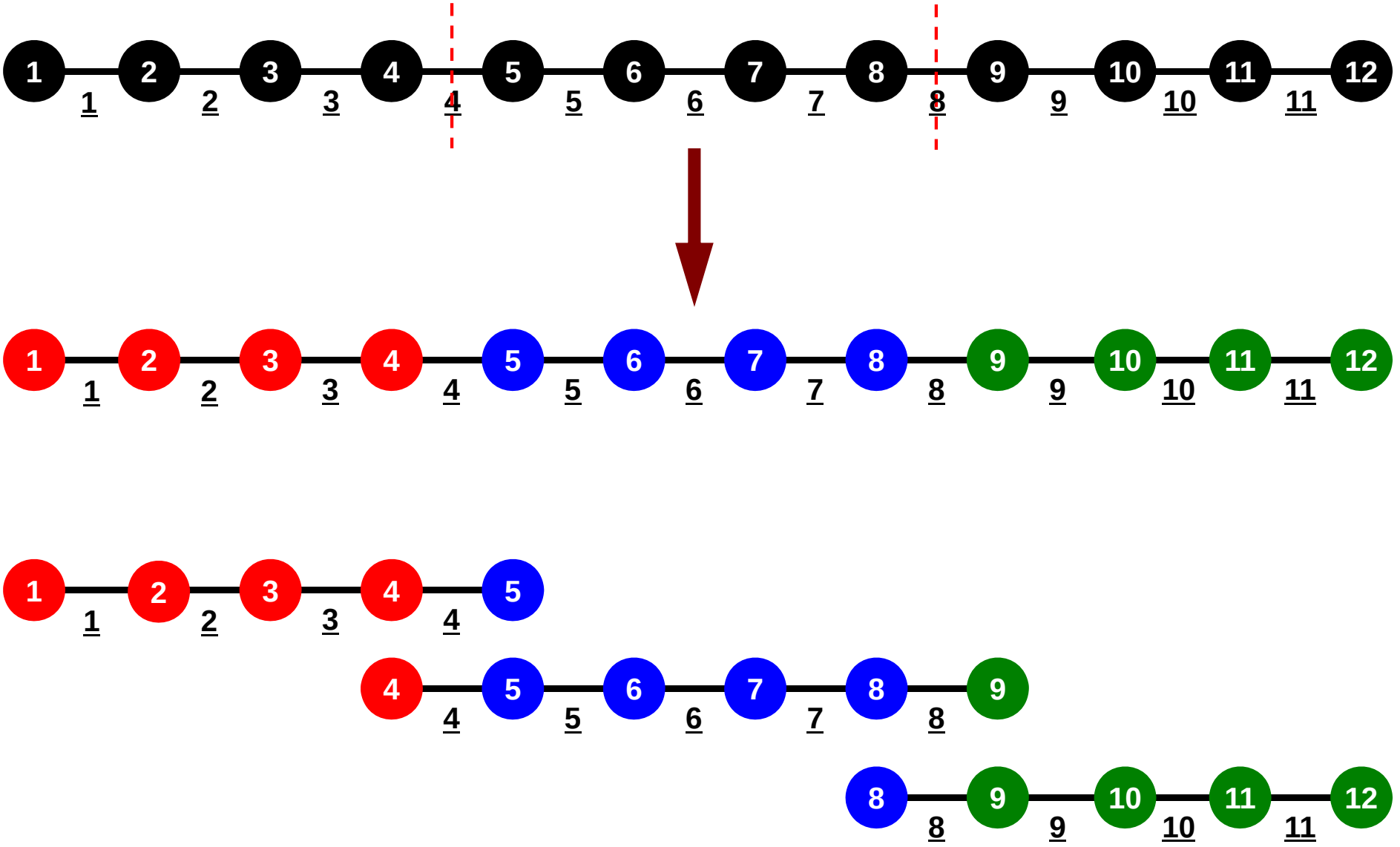
要素＋外点



一次元問題:11要素，12節点，3領域



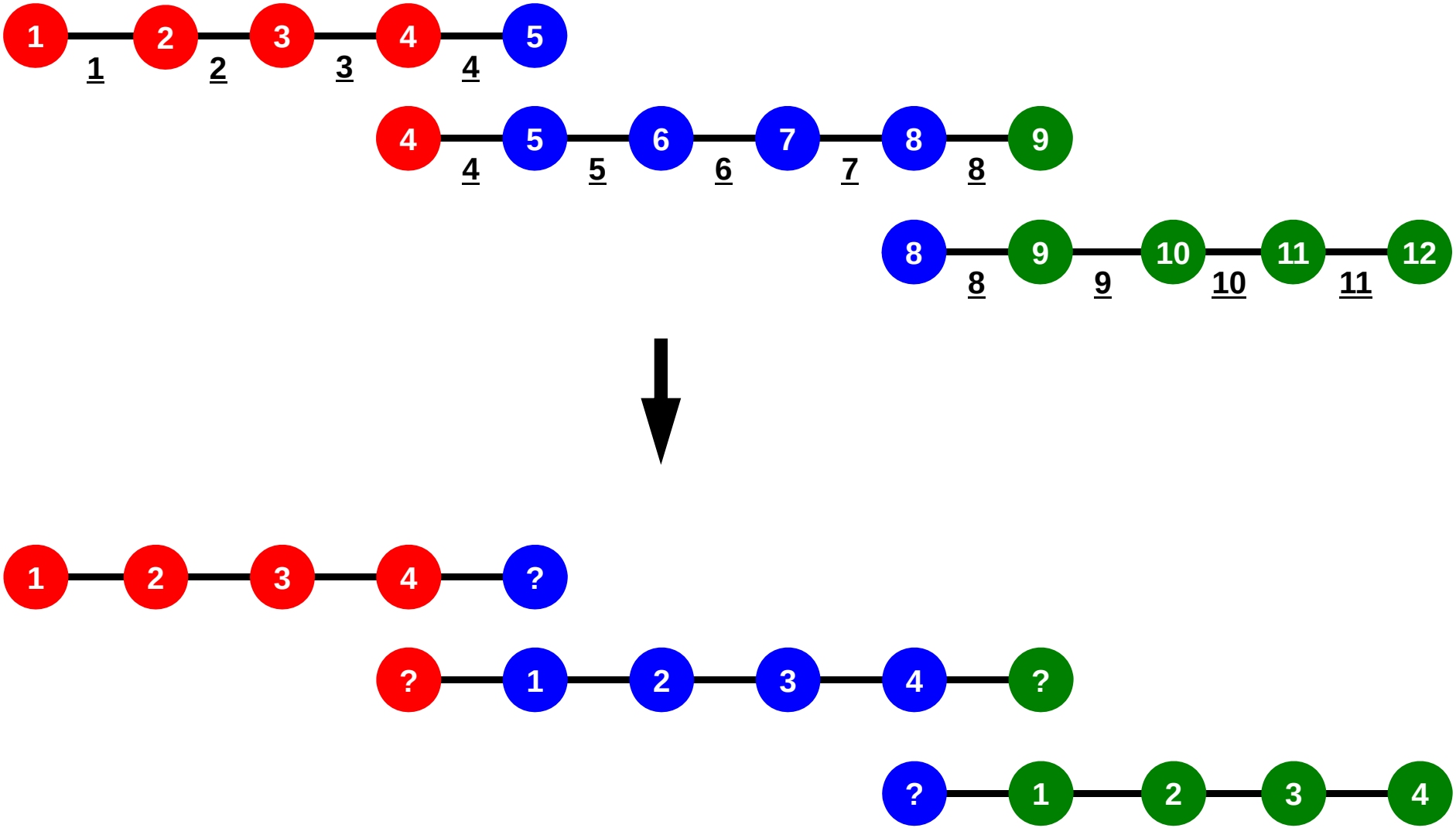
一次元問題:11要素，12節点，3領域



SPMD向け局所番号付け

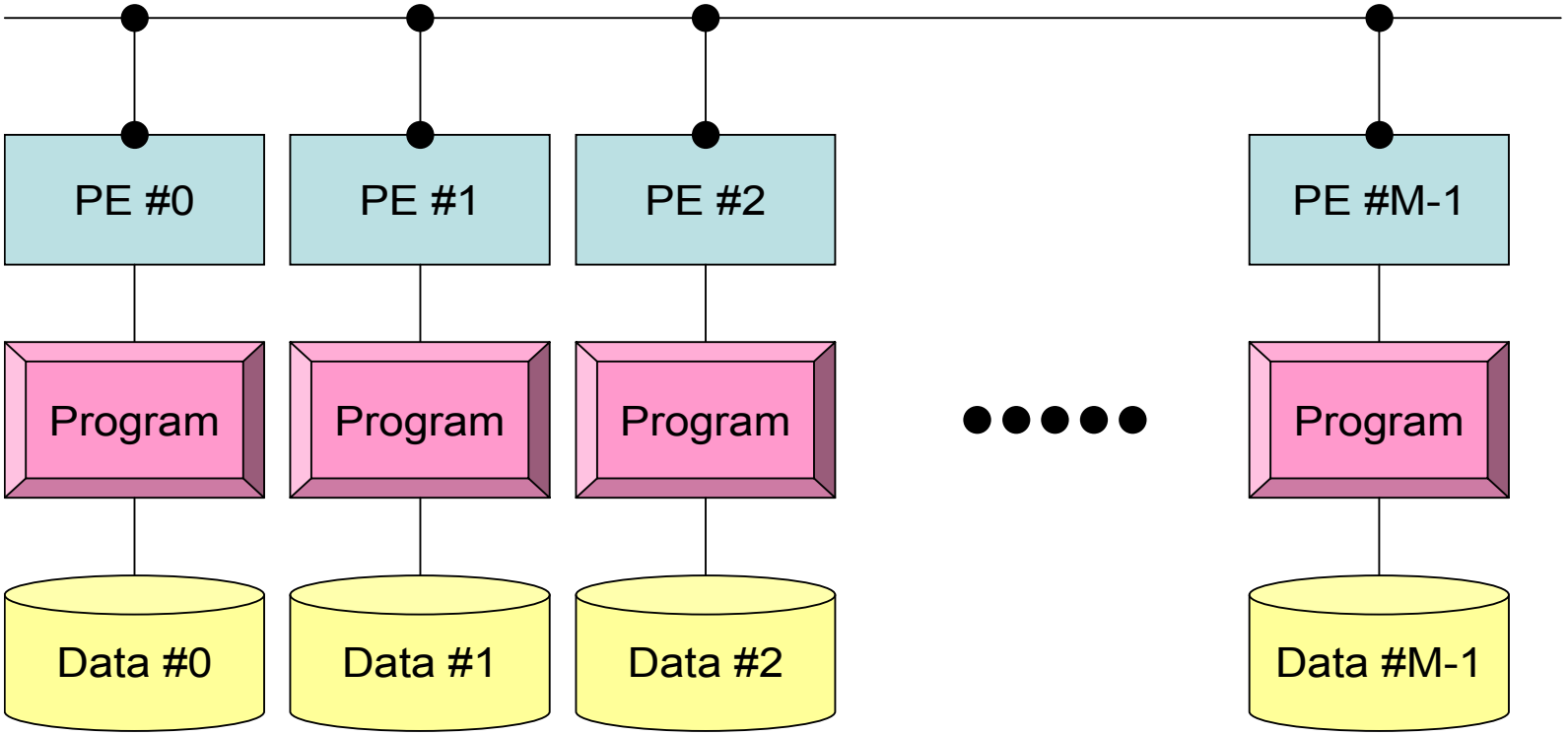
内点が1~Nとなっていれば，もとのプログラムと同じ

外点の番号は?



MPIによる並列化：SPMD

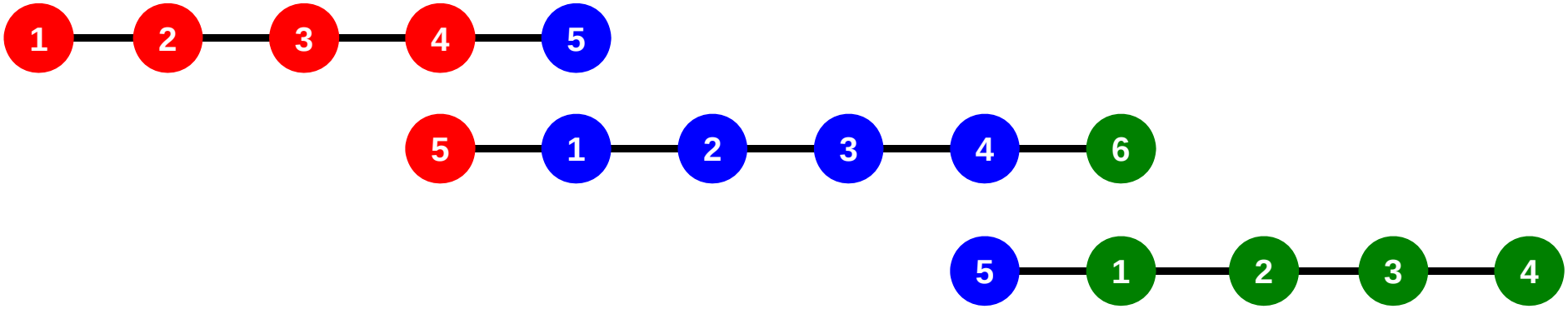
- Single Program/Instruction Multiple Data
- 基本的に各プロセスは「同じことをやる」が「データが違う」
  - 大規模なデータを分割し，各部分について各プロセス（プロセッサ）が計算する
- 全体データと局所データ，全体番号と局所番号
- 通信以外は単体CPUと同じ，というのが理想



SPMD向け局所番号付け

内点が1~Nとなっていれば，もとのプログラムと同じ

外点の番号は?，N+1,N+2



- MPI超入門
- 並列有限要素法への道
  - 局所データ構造
- 並列有限要素法のためのMPI
  - Collective Communication（集団通信）
  - Point-to-Point (Peer-to-Peer) Communication（1対1通信）

有限要素法の処理：プログラム

- 初期化：並列計算可
  - 制御変数読み込み
  - 座標読み込み⇒要素生成（N:節点数，NE：要素数）
  - 配列初期化（全体マトリクス，要素マトリクス）
  - 要素⇒全体マトリクスマッピング（Index，Item）
- マトリクス生成：並列計算可
  - 要素単位の処理（do icel= 1, NE）
    - 要素マトリクス計算
    - 全体マトリクスへの重ね合わせ
  - 境界条件の処理
- 連立一次方程式：?
  - 共役勾配法（CG）

前処理付き共役勾配法
Preconditioned Conjugate Gradient Method（CG）

Compute $\mathbf{r}^{(0)}= \mathbf{b}-[A]\mathbf{x}^{(0)}$
for i= 1, 2, …
  solve $[M]\mathbf{z}^{(i-1)}= \mathbf{r}^{(i-1)}$
  $\rho_{i-1}= \mathbf{r}^{(i-1)}\ \mathbf{z}^{(i-1)}$
  if i=1
    $p^{(1)}= z^{(0)}$
  else
    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
    $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$
  endif
  $\mathbf{q}^{(i)}= [A]\mathbf{p}^{(i)}$
  $\alpha_i\ = \rho_{i-1}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence $|\mathbf{r}|$
end

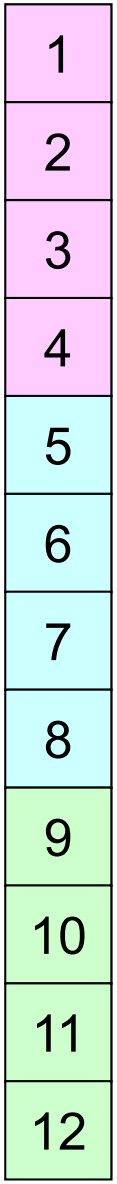
- 前処理
  - 対角スケーリング
- 並列処理が必要なプロセス
  - 内積
  - 行列ベクトル積

前処理，ベクトル定数倍の加減

局所的な計算（内点のみ）が可能⇒並列処理

```
!C
!C-- {z}= [Minv]{r}

      do i= 1, N
        W(i,Z)= W(i,DD) * W(i,R)
      enddo
```

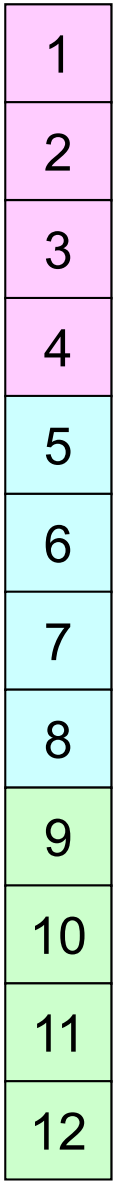
```
!C
!C-- {x}= {x} + ALPHA*{p}
!C   {r}= {r} - ALPHA*{q}

      do i= 1, N
        PHI(i)= PHI(i) + ALPHA * W(i,P)
        W(i,R)= W(i,R) - ALPHA * W(i,Q)
      enddo
```



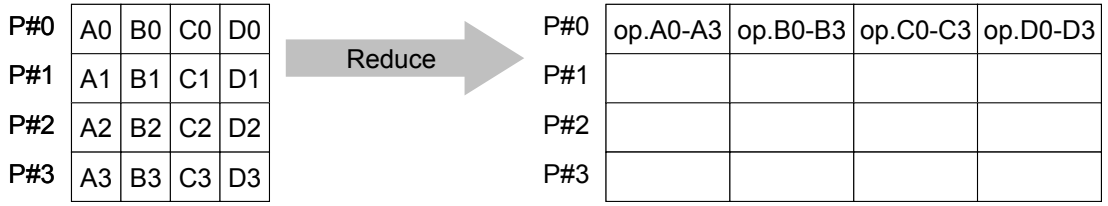
内積

全体で和をとる必要がある⇒通信?

```
!C
!C-- ALPHA= RHO / {p}{q}

      C1= 0.d0
      do i= 1, N
        C1= C1 + W(i,P)*W(i,Q)
      enddo
      ALPHA= RHO / C1
```



MPI_REDUCE



- コミュニケーター「comm」内の，各プロセスの送信バッファ「sendbuf」について，演算「op」を実施し，その結果を1つの受信プロセス「root」の受信バッファ「recbuf」に格納する。
  - 総和，積，最大，最小 他

- **call MPI_REDUCE (sendbuf,recvbuf,count,datatype,op,root,comm,ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **recvbuf** 任意 O 受信バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ

    ```
    FORTRAN  MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
    C        MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc
    ```

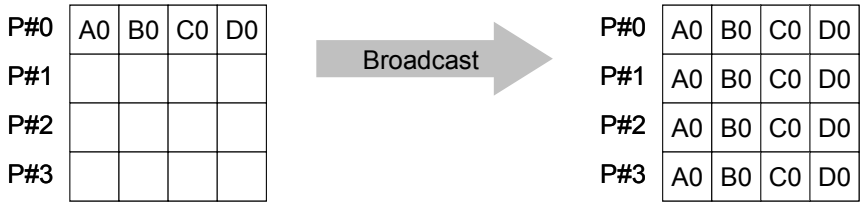
  - **op** 整数 I 計算の種類

    ```
    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
    ```

    ユーザーによる定義も可能：MPI_OP_CREATE
  - **root** 整数 I 受信元プロセスのID（ランク）
  - **comm** 整数 I コミュニケータを指定する
  - **ierr** 整数 O 完了コード

Fortran

MPI_BCAST



- コミュニケーター「comm」内の一つの送信元プロセス「root」のバッファ「buffer」から，その他全てのプロセスのバッファ「buffer」にメッセージを送信。

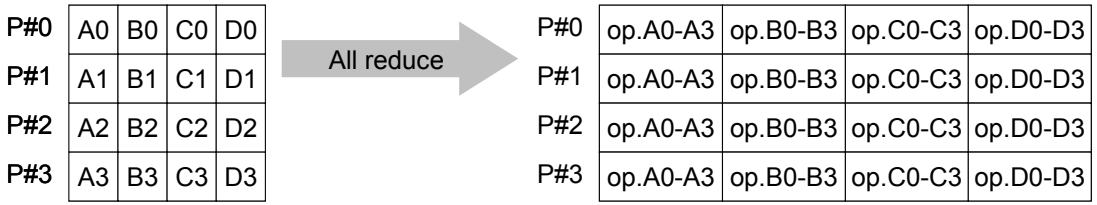
- **`call MPI_BCAST (buffer,count,datatype,root,comm,ierr)`**
  - **`buffer`** 任意 I/O バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **`count`** 整数 I メッセージのサイズ
  - **`datatype`** 整数 I メッセージのデータタイプ

```
FORTRAN  MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
C        MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc.
```

  - **`root`** 整数 I 送信元プロセスのID（ランク）
  - **`comm`** 整数 I コミュニケータを指定する
  - **`ierr`** 整数 O 完了コード

Fortran

MPI_ALLREDUCE



- MPI_REDUCE + MPI_BCAST
- 総和，最大値を計算したら，各プロセスで利用したい場合が多い

- **`call MPI_ALLREDUCE`**
  `(sendbuf,recvbuf,count,datatype,op, comm,ierr)`
  - **`sendbuf`** 任意 I 送信バッファの先頭アドレス，
  - **`recvbuf`** 任意 O 受信バッファの先頭アドレス，
    タイプは「datatype」により決定
  - **`count`** 整数 I メッセージのサイズ
  - **`datatype`** 整数 I メッセージのデータタイプ
  - **`op`** 整数 I 計算の種類
  - **`comm`** 整数 I コミュニケータを指定する
  - **`ierr`** 整数 O 完了コード

Fortran

MPI_Reduce/Allreduceの “op”

Fortran

```
call MPI_REDUCE
(sendbuf,recvbuf,count,datatype,op,root,comm,ierr)
```

- **MPI_MAX**, **MPI_MIN** 最大値，最小値
- **MPI_SUM**, **MPI_PROD** 総和，積
- **MPI_LAND** 論理AND

- MPI超入門
- 並列有限要素法への道
  - 局所データ構造
- 並列有限要素法のためのMPI
  - Collective Communication（集団通信）
  - Point-to-Point (Peer-to-Peer) Communication（1対1通信）

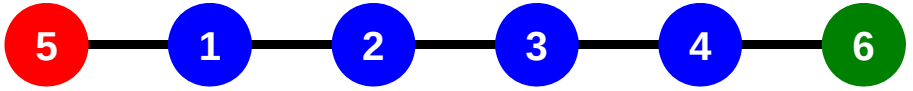
前処理付き共役勾配法
Preconditioned Conjugate Gradient Method（CG）

Compute $\mathbf{r}^{(0)}= \mathbf{b}-[A]\mathbf{x}^{(0)}$
for i= 1, 2, …
  solve $[M]\mathbf{z}^{(i-1)}= \mathbf{r}^{(i-1)}$
  $\rho_{i-1}= \mathbf{r}^{(i-1)}\ \mathbf{z}^{(i-1)}$
  if i=1
    $p^{(1)}= z^{(0)}$
  else
    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$
    $p^{(i)}= z^{(i-1)} + \beta_{i-1}\ p^{(i-1)}$
  endif
  $\mathbf{q}^{(i)}= [A]\mathbf{p}^{(i)}$
  $\alpha_i\ = \rho_{i-1}/\mathbf{p}^{(i)}\mathbf{q}^{(i)}$
  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
  check convergence $|\mathbf{r}|$
end

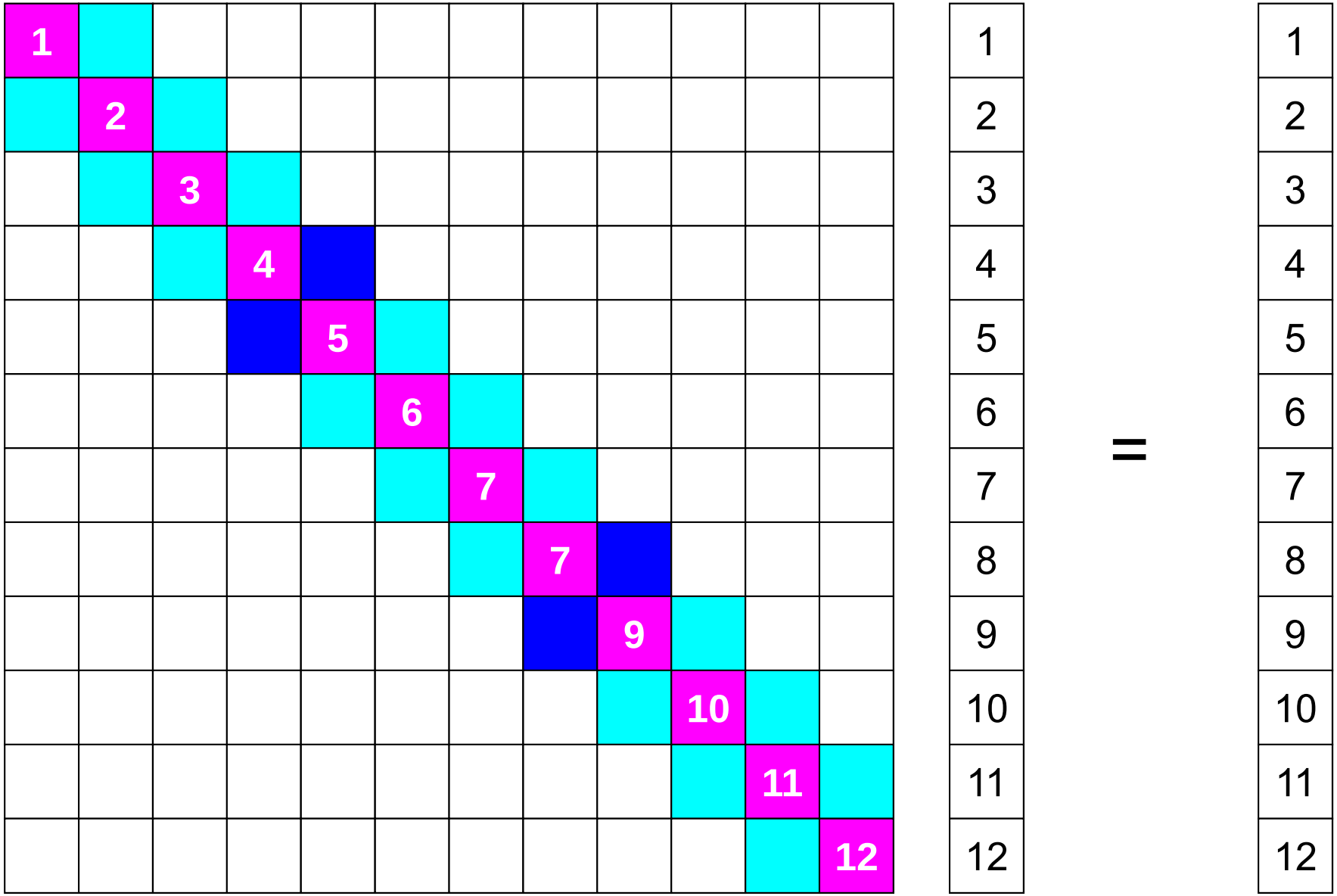
- 前処理
  - 対角スケーリング
- 並列処理が必要なプロセス
  - 内積
  - 行列ベクトル積

行列ベクトル積
外点の値が必要⇒通信?

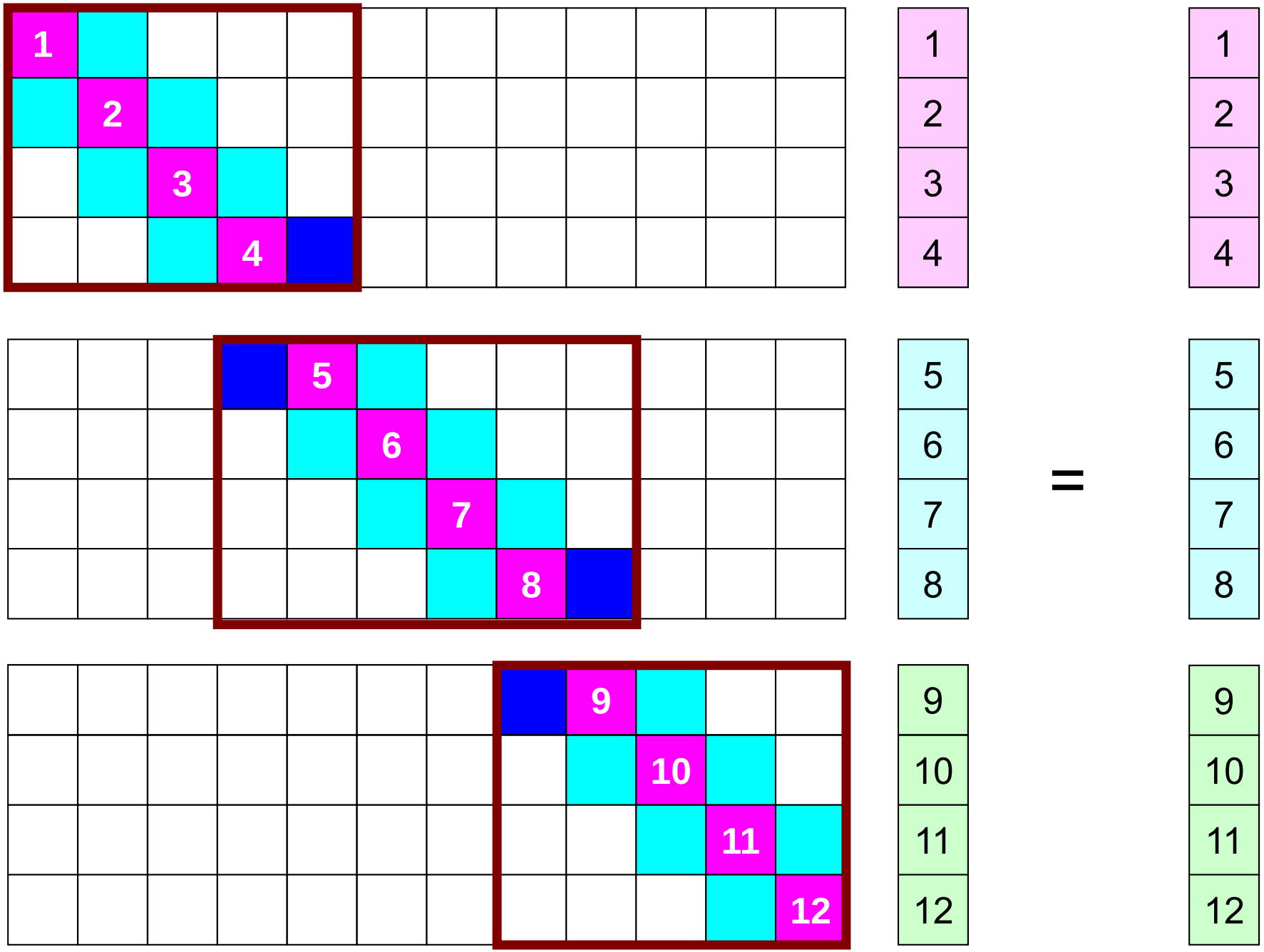
```
!C
!C-- {q}= [A]{p}

      do i= 1, N
        W(i,Q) = DIAG(i)*W(i,P)
        do j= INDEX(i-1)+1, INDEX(i)
          W(i,Q) = W(i,Q) + AMAT(j)*W(ITEM(j),P)
        enddo
      enddo
```



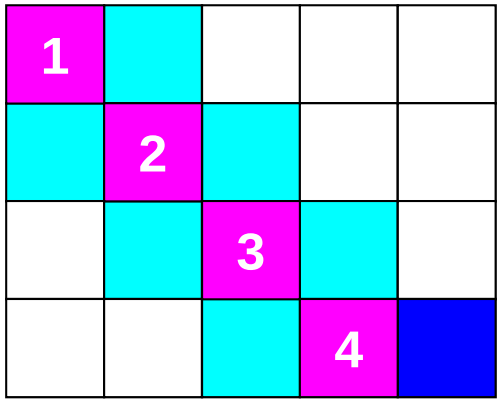
行列ベクトル積：ローカルに計算実施可能



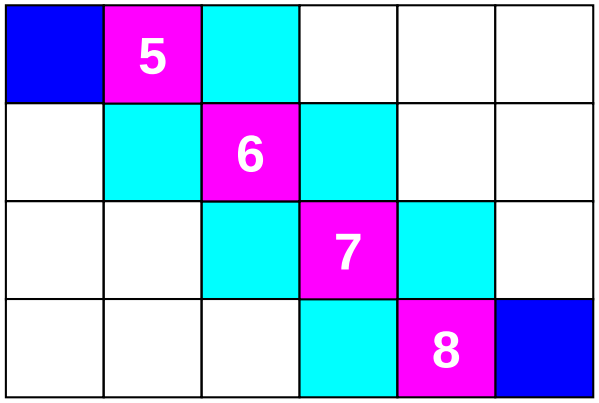
行列ベクトル積：ローカルに計算実施可能



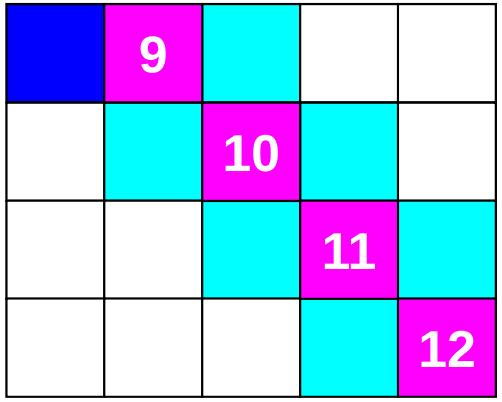
行列ベクトル積：ローカルに計算実施可能



1	1
2	2
3	3
4	4

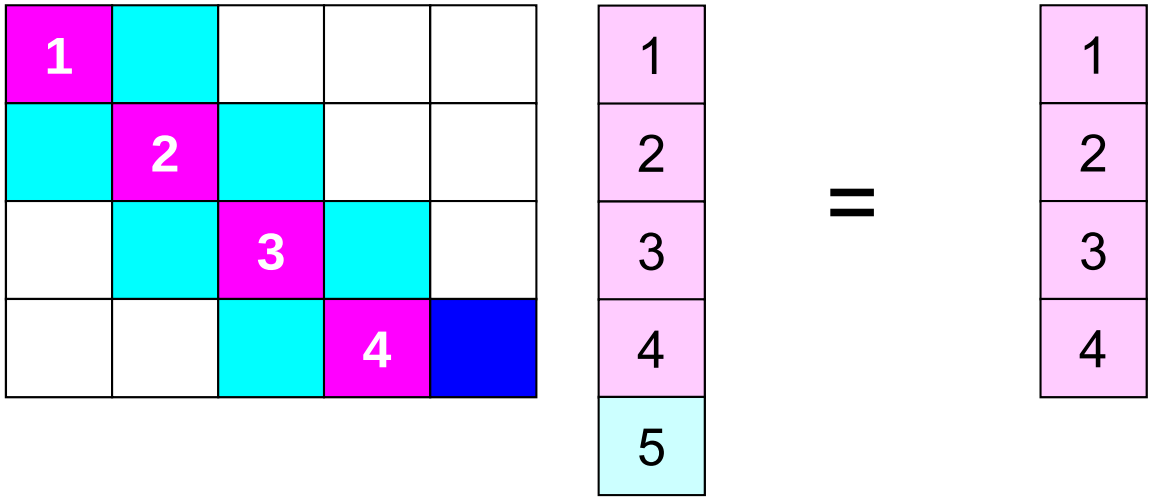


5	=	5
6		6
7		7
8		8

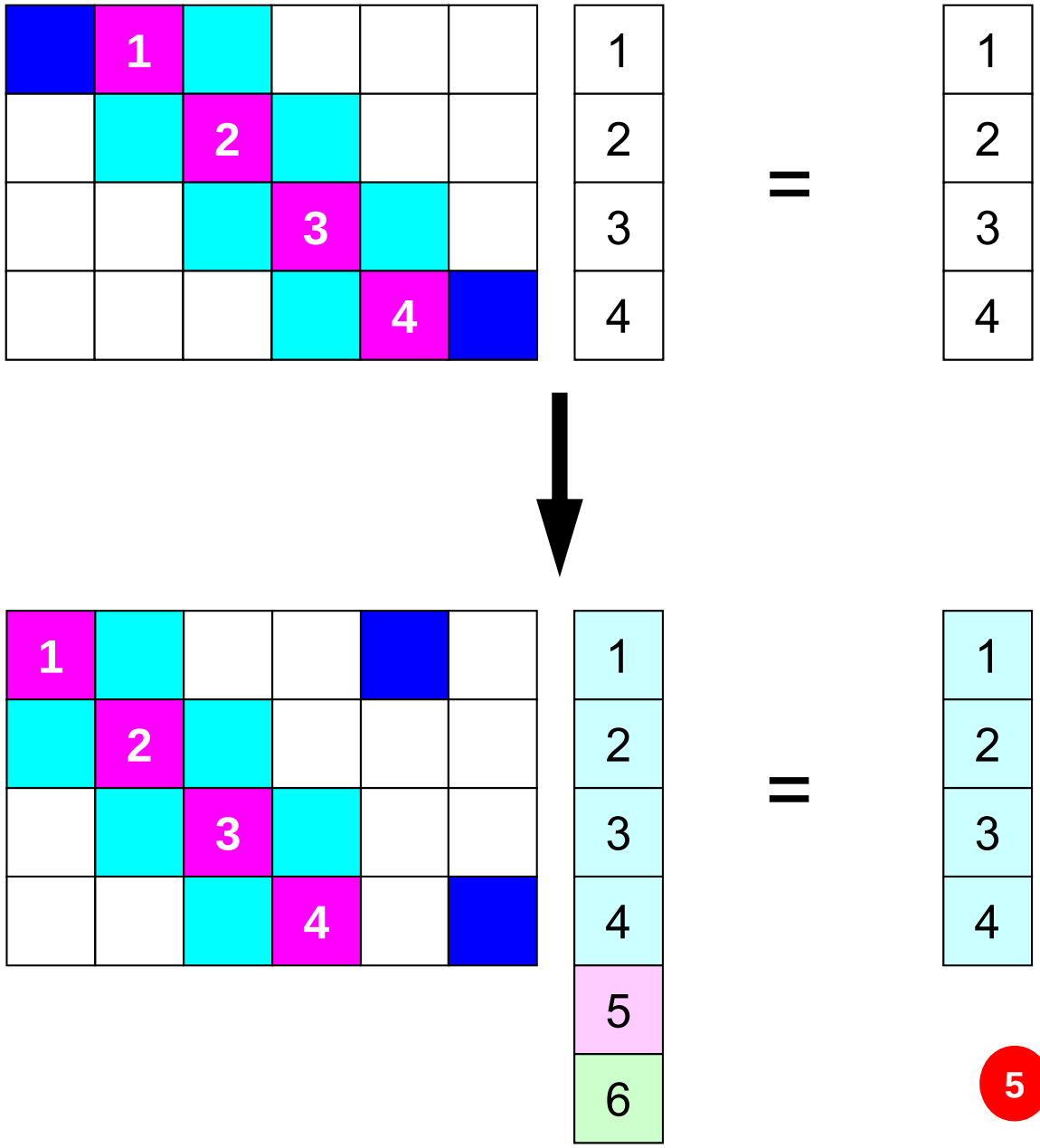


9	9
10	10
11	11
12	12

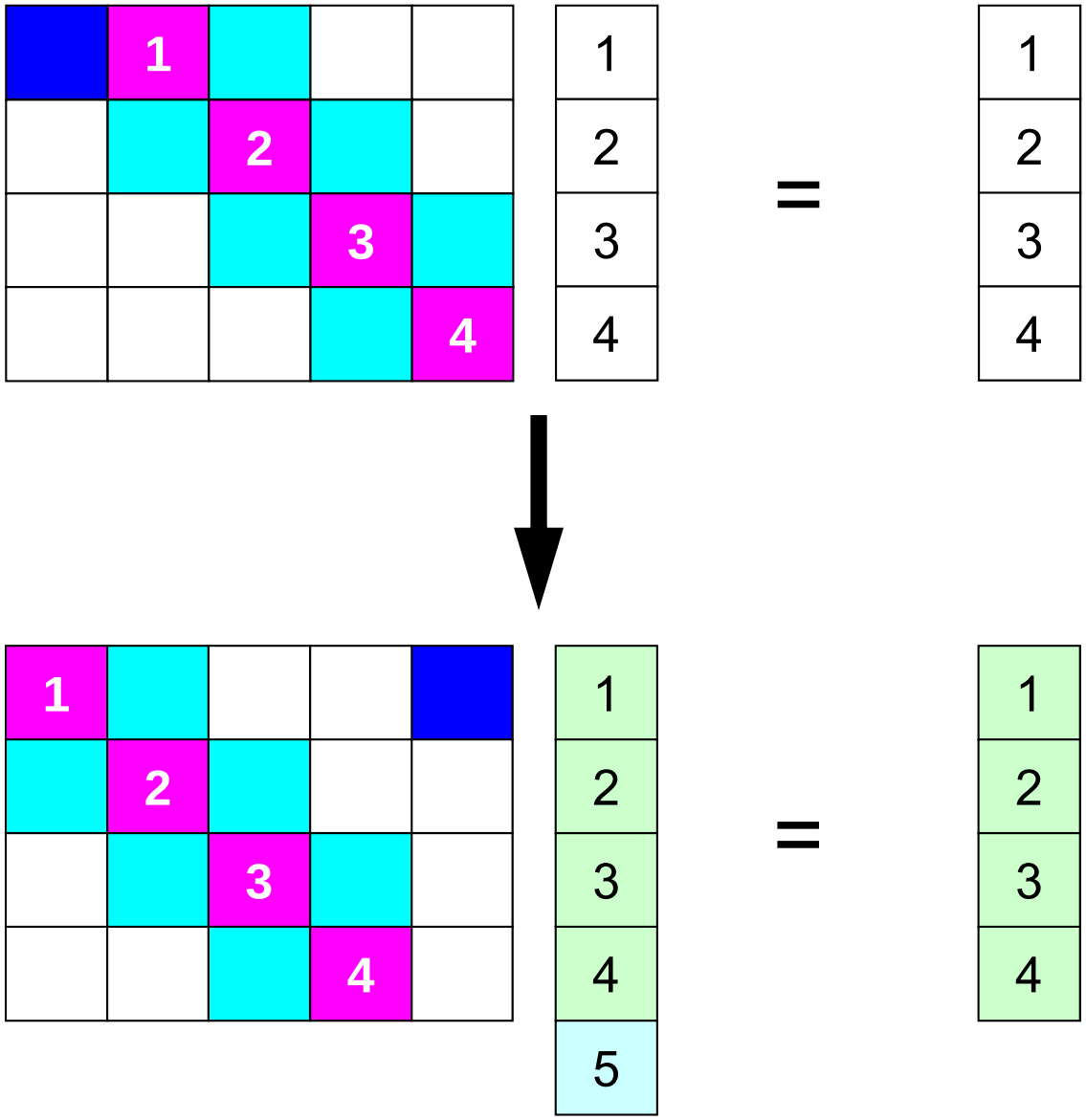
行列ベクトル積：ローカル計算 #0



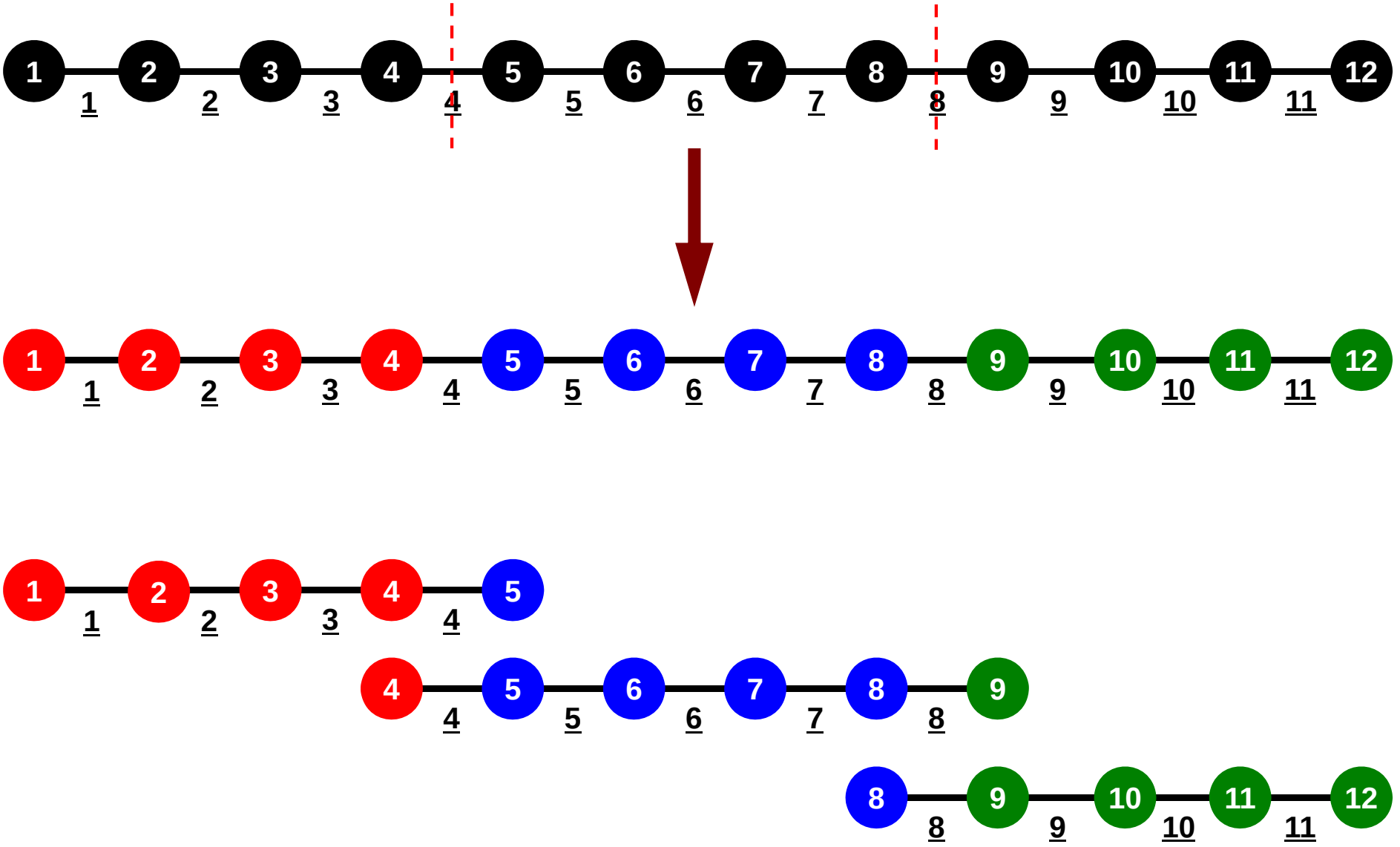
行列ベクトル積：ローカル計算 #1



行列ベクトル積：ローカル計算 #2

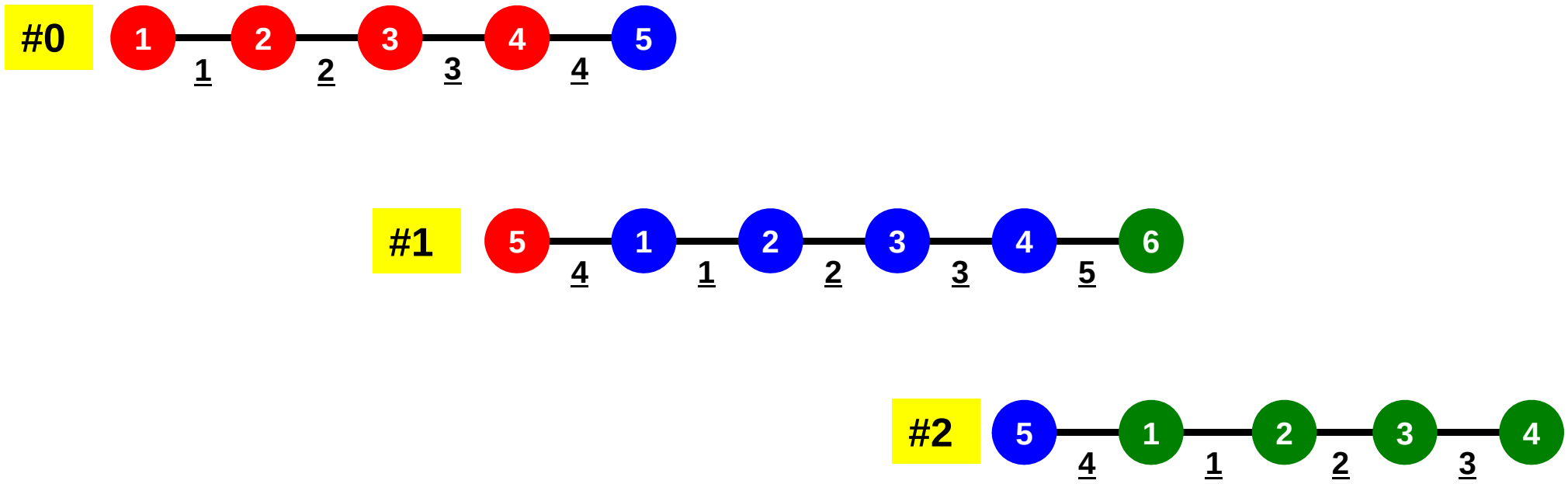


1D FEM: 12 nodes/11 elem's/3 domains



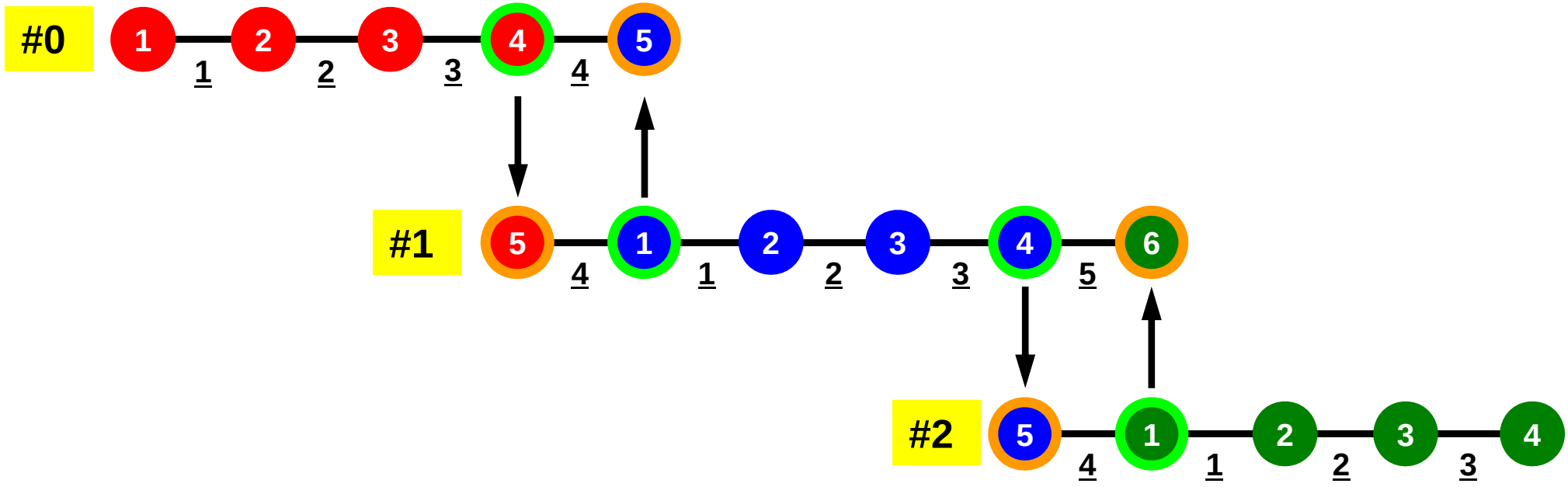
1D FEM: 12 nodes/11 elem's/3 domains

Local ID: Starting from 0 for node and elem at each domain



1D FEM: 12 nodes/11 elem's/3 domains

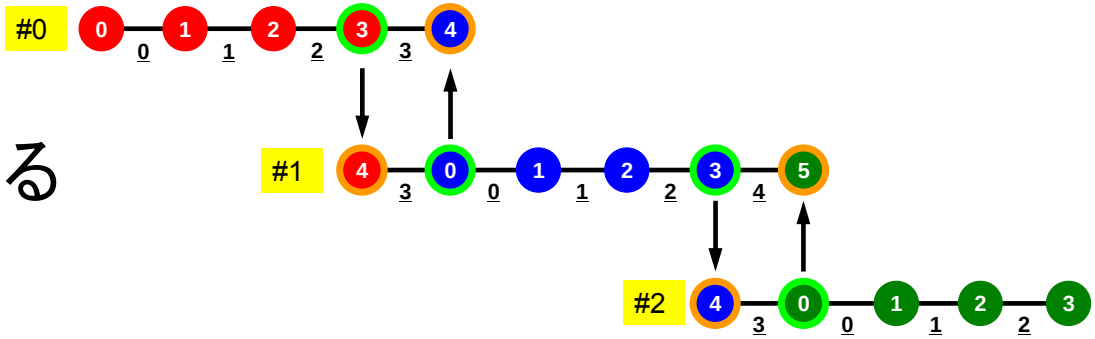
Internal/External Nodes



1対1通信とは？

- グループ通信：Collective Communication
  - MPI_Reduce, MPI_Scatter/Gather など
  - 同じコミュニケータ内の全プロセスと通信する
  - 適用分野
    - 境界要素法，スペクトル法，分子動力学等グローバルな相互作用のある手法
    - 内積，最大値などのオペレーション

- 1対1通信：Point-to-Point
  - MPI_Send, MPI_Receive
  - 特定のプロセスとのみ通信がある
    - 隣接領域
  - 適用分野
    - 差分法，有限要素法などローカルな情報を使う手法

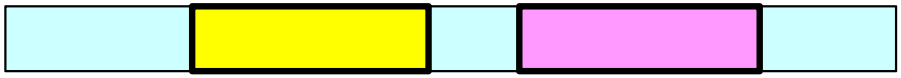


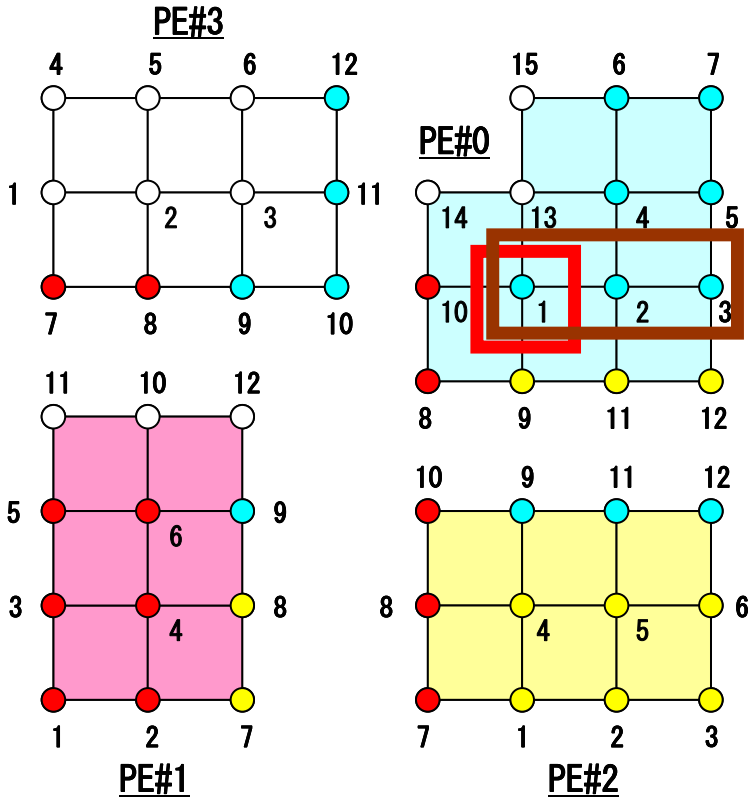
SEND（送信）：境界点の送信

送信バッファの連続したデータを隣接プロセスに送る

```
call MPI_ISEND
   (sendbuf,count,datatype,dest,tag,comm,request, ierr)
```

- **sendbuf** 任意 I 送信バッファの先頭アドレス，
- **count** 整数 I メッセージのサイズ
- **datatype** 整数 I メッセージのデータタイプ
- **dest** 整数 I 宛先プロセスのアドレス（ランク）





Fortran

MPI_ISEND

- 送信バッファ「sendbuf」内の，連続した「count」個の送信メッセージを，タグ「tag」を付けて，コミュニケータ内の，「dest」に送信する。「MPI_WAITALL」を呼ぶまで，送信バッファの内容を更新してはならない。

- **call MPI_ISEND**
  (sendbuf,count,datatype,dest,tag,comm,request, ierr)
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，送信メッセージの種類を区別するときに使用。通常は「0」でよい。同じメッセージタグ番号同士で通信。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_WAITALLで使用。（配列：サイズは同期する必要のある「MPI_ISEND」呼び出し数（通常は隣接プロセス数など））：C言語については後述
  - **ierr** 整数 O 完了コード

通信識別子（request handle）：request

- **call MPI_ISEND (sendbuf,count,datatype,dest,tag,comm,request, ierr)**
  - **sendbuf** 任意 I 送信バッファの先頭アドレス，
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，送信メッセージの種類を区別するときに使用。通常は「0」でよい。同じメッセージタグ番号同士で通信。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_WAITALLで使用。（配列：サイズは同期する必要のある「MPI_ISEND」呼び出し数（通常は隣接プロセス数など））
  - **ierr** 整数 O 完了コード

- 以下のような形で宣言しておく（記憶領域を確保するだけで良い：Cについては後述）

```
allocate (request(NEIBPETOT))
```

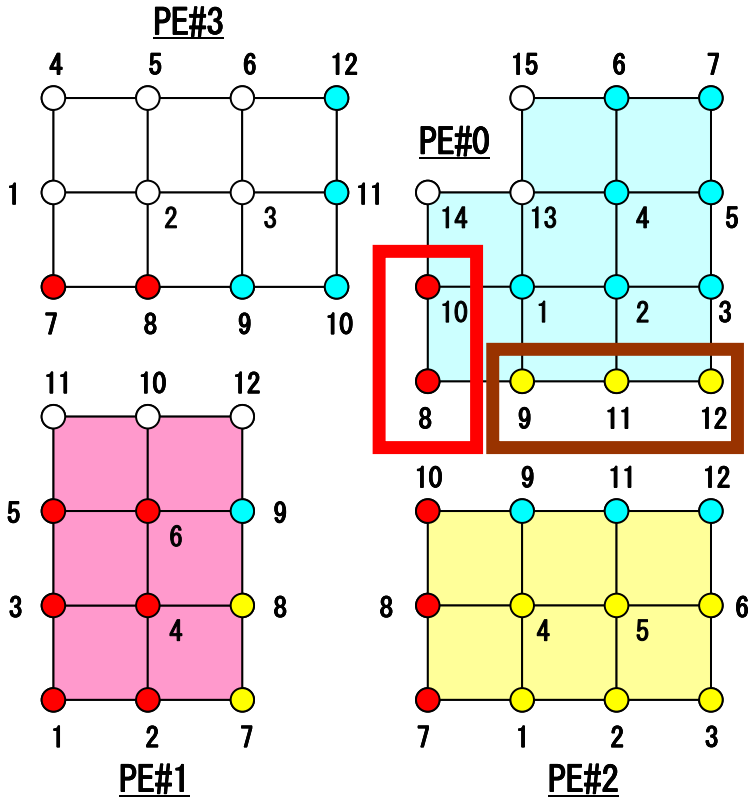
Fortran

RECV(受信):外点への受信

受信バッファに隣接プロセスから連続したデータを受け取る

```
call MPI_IRECV
   (recvbuf,count,datatype,dest,tag,comm,request, ierr)
```

- **recvbuf** 任意 I 受信バッファの先頭アドレス,
- **count** 整数 I メッセージのサイズ
- **datatype** 整数 I メッセージのデータタイプ
- **dest** 整数 I 宛先プロセスのアドレス(ランク)



Fortran

MPI_IRECV

- 受信バッファ「recvbuf」内の，連続した「count」個の送信メッセージを，タグ「tag」を付けて，コミュニケータ内の，「dest」から受信する。「MPI_WAITALL」を呼ぶまで，受信バッファの内容を利用した処理を実施してはならない。

- **call MPI_IRECV**
  (recvbuf,count,datatype,dest,tag,comm,request, ierr)
  - **recvbuf** 任意 I 受信バッファの先頭アドレス，
  - **count** 整数 I メッセージのサイズ
  - **datatype** 整数 I メッセージのデータタイプ
  - **dest** 整数 I 宛先プロセスのアドレス（ランク）
  - **tag** 整数 I メッセージタグ，受信メッセージの種類を区別するときに使用。通常は「0」でよい。同じメッセージタグ番号同士で通信。
  - **comm** 整数 I コミュニケータを指定する
  - **request** 整数 O 通信識別子。MPI_WAITALLで使用。（配列：サイズは同期する必要のある「MPI_IRECV」呼び出し数（通常は隣接プロセス数など））：C言語については後述
  - **ierr** 整数 O 完了コード

Fortran

MPI_WAITALL

- 1対1非ブロッキング通信サブルーチンである「MPI_ISEND」と「MPI_IRECV」を使用した場合，プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_WAITALL」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_WAITALL」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば，「MPI_ISEND」と「MPI_IRECV」を同時に同期してもよい。
  - 「MPI_ISEND/IRECV」で同じ通信識別子を使用すること
- 「MPI_BARRIER」と同じような機能であるが，代用はできない。
  - 実装にもよるが，「request」，「status」の内容が正しく更新されず，何度も「MPI_ISEND/IRECV」を呼び出すと処理が遅くなる，というような経験もある。

- **`call MPI_WAITALL (count,request,status,ierr)`**
  - **`count`** 整数 I 同期する必要のある「MPI_ISEND」，「MPI_RECV」呼び出し数。
  - **`request`** 整数 I/O 通信識別子。「MPI_ISEND」，「MPI_IRECV」で利用した識別子名に対応。（配列サイズ：（count））
  - **`status`** 整数 O 状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE,count））
    MPI_STATUS_SIZE：“mpif.h”，”mpi.h”で定められるパラメータ：C言語については後述
  - **`ierr`** 整数 O 完了コード

状況オブジェクト配列（status object）：status

- `call MPI_WAITALL (count,request,status,ierr)`
  - **count** 整数 I 同期する必要のある「MPI_ISEND」，「MPI_RECV」呼び出し数。
  - **request** 整数 I/O 通信識別子。「MPI_ISEND」，「MPI_IRECV」で利用した識別子名に対応。（配列サイズ：（count））
  - **status** 整数 O 状況オブジェクト配列（配列サイズ：（MPI_STATUS_SIZE,count））
    MPI_STATUS_SIZE：“mpif.h”,”mpi.h”で定められるパラメータ
  - **ierr** 整数 O 完了コード

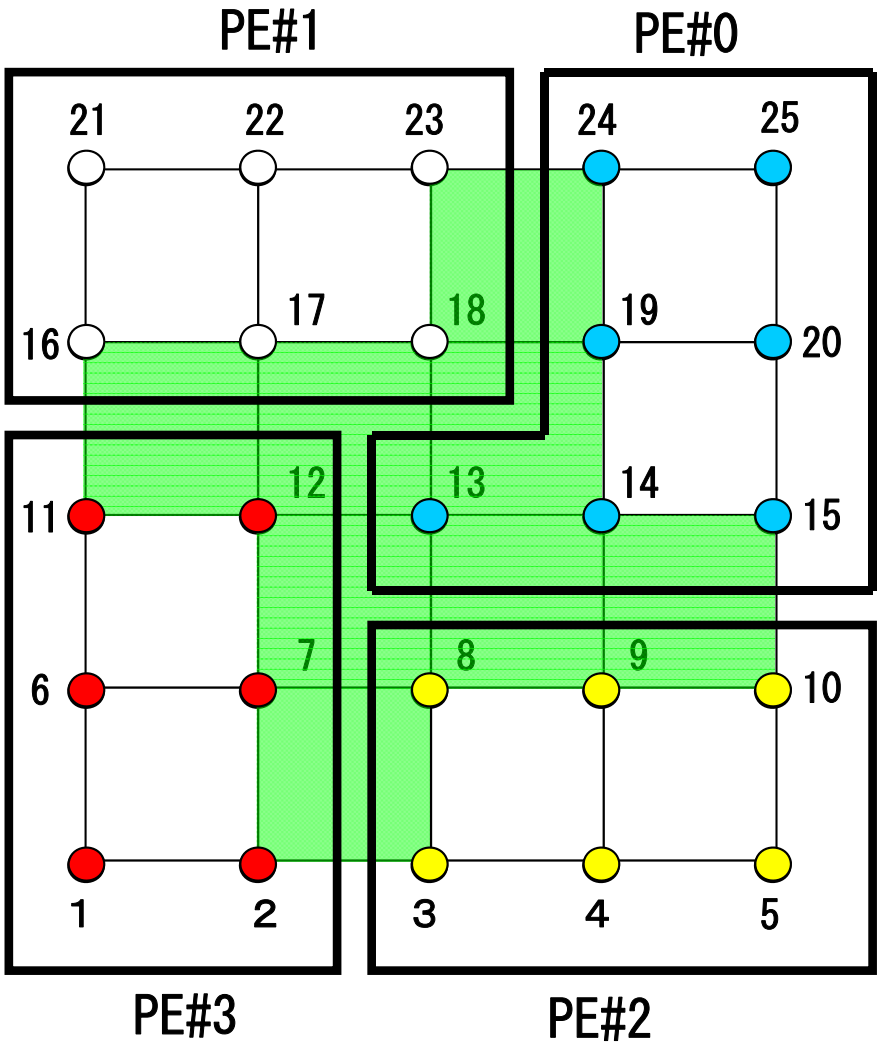
- 以下のように予め記憶領域を確保しておくだけでよい（Cについては後述）：

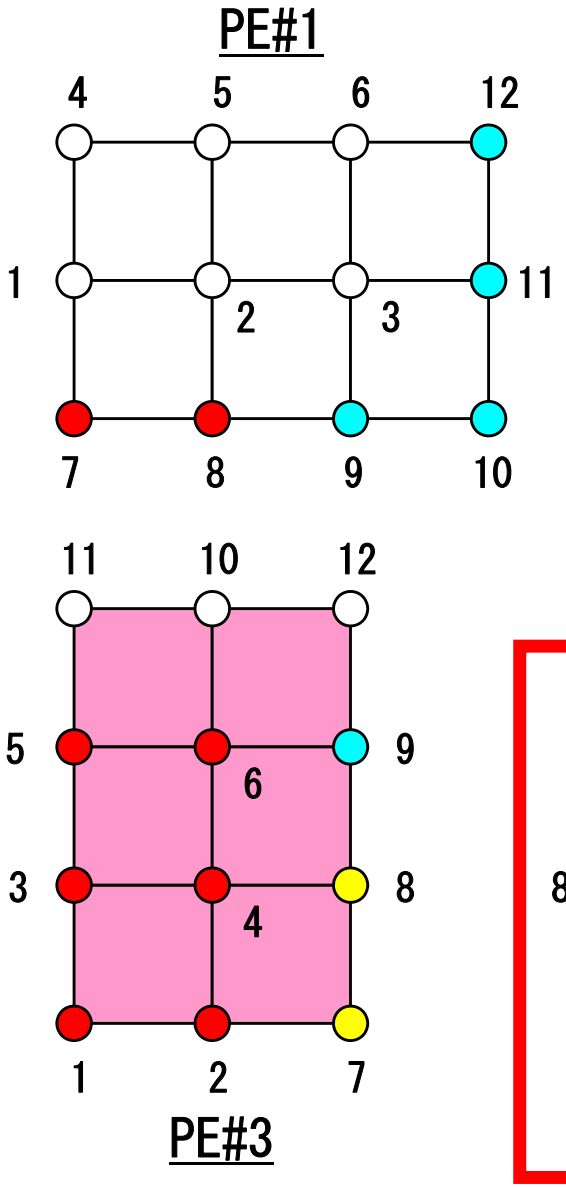
```
allocate (stat(MPI_STATUS_SIZE,NEIBPETOT))
```

Fortran

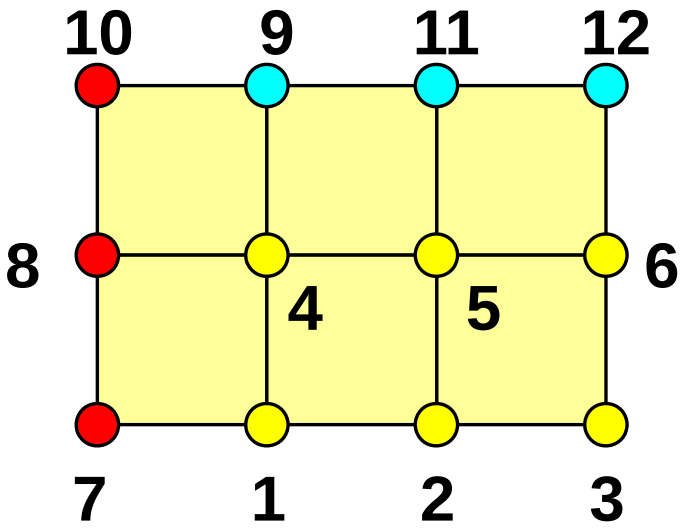
Node-based Partitioning

internal nodes - elements - external nodes





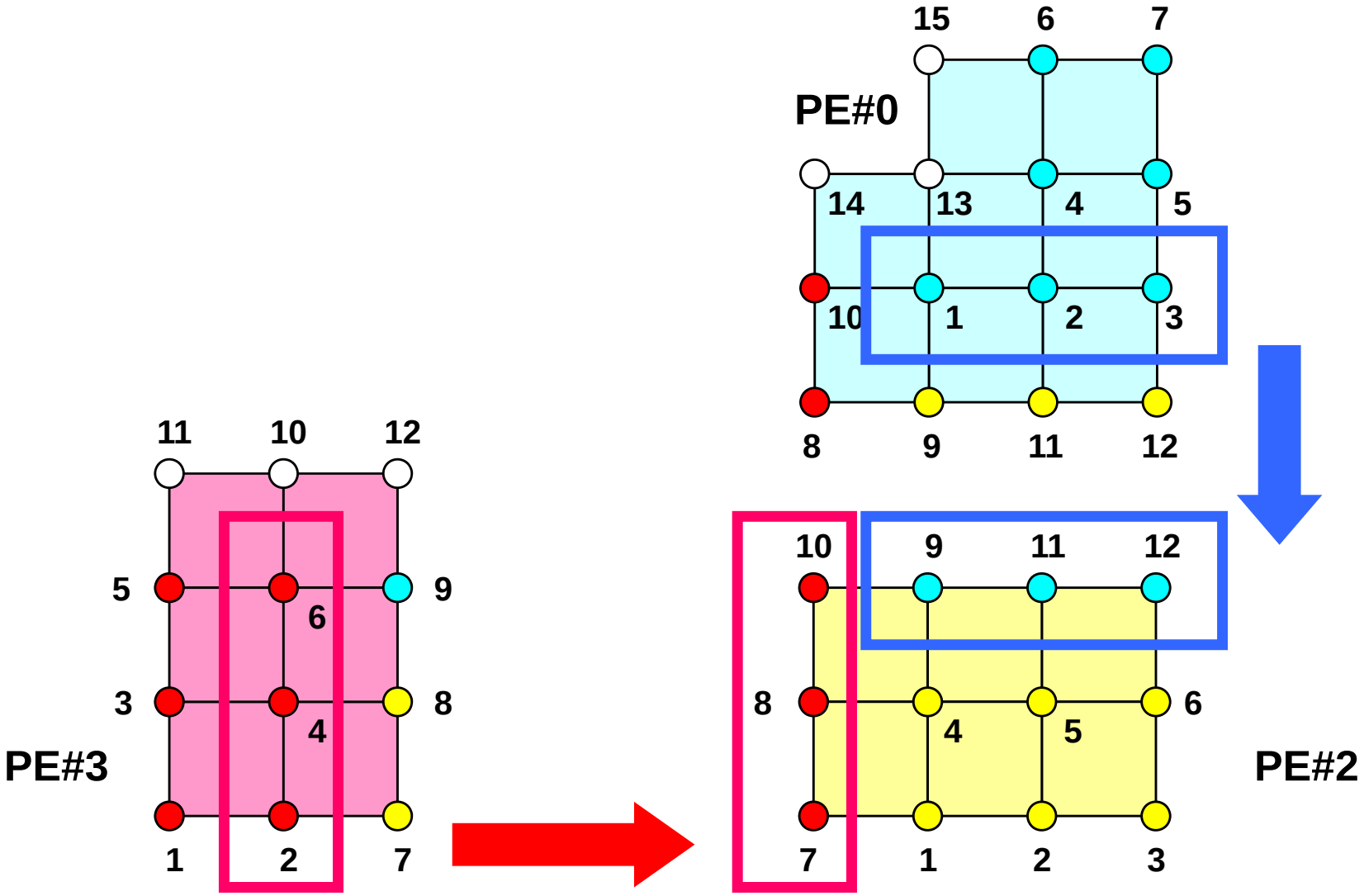
各領域データ(局所データ)仕様



- 内点，外点（internal/external nodes）
  - 内点～外点となるように局所番号をつける
- 隣接領域情報
  - オーバーラップ要素を共有する領域
  - 隣接領域数，番号
- 外点情報
  - どの領域から，何個の，どの外点の情報を「受信：import」するか
- 境界点情報
  - 何個の，どの境界点の情報を，どの領域に「送信：export」するか

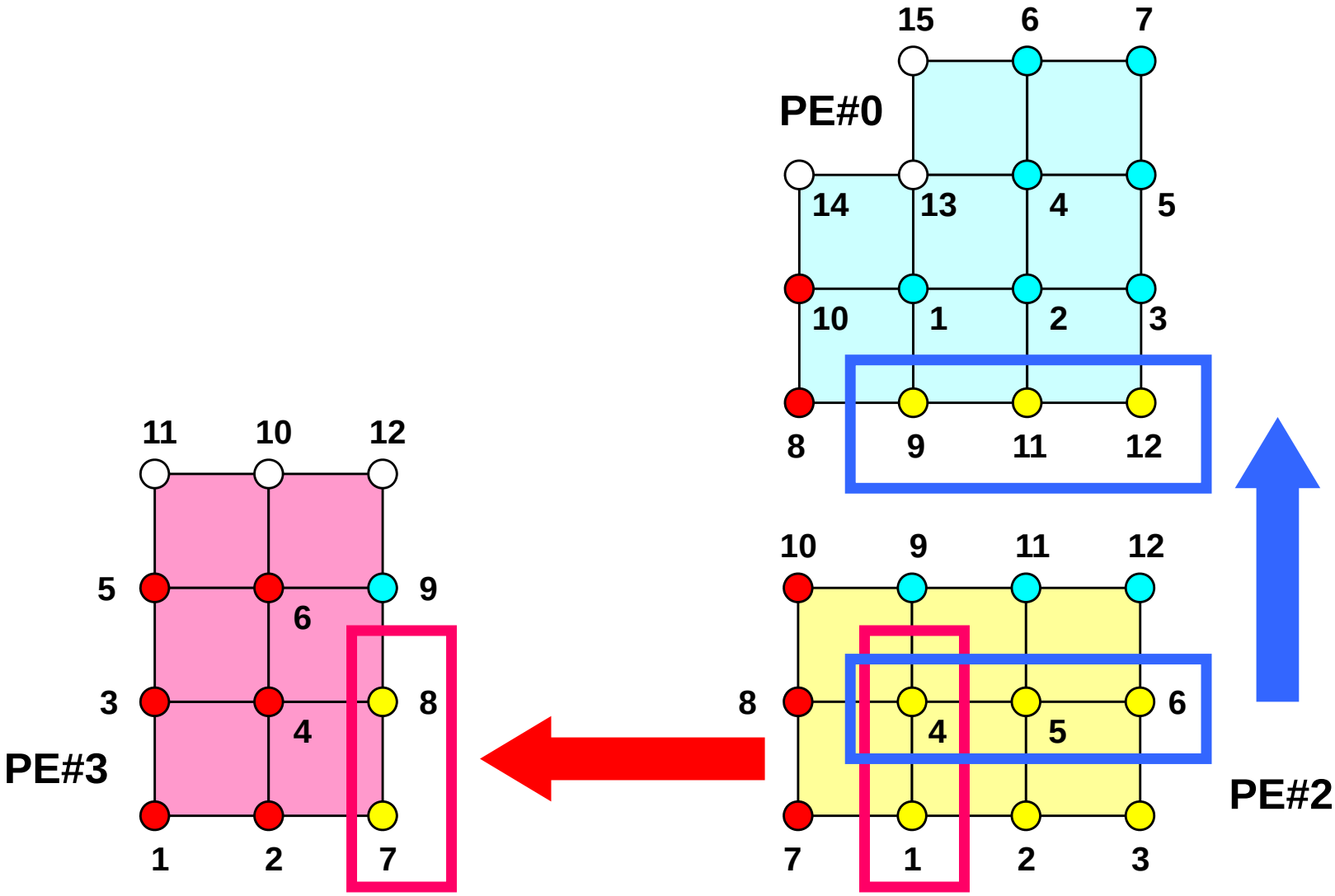
External Nodes（外点）：RECEIVE

PE#2 : receive information for “external nodes”



Boundary Nodes（内点）：SEND

PE#2 : send information on “boundary nodes”

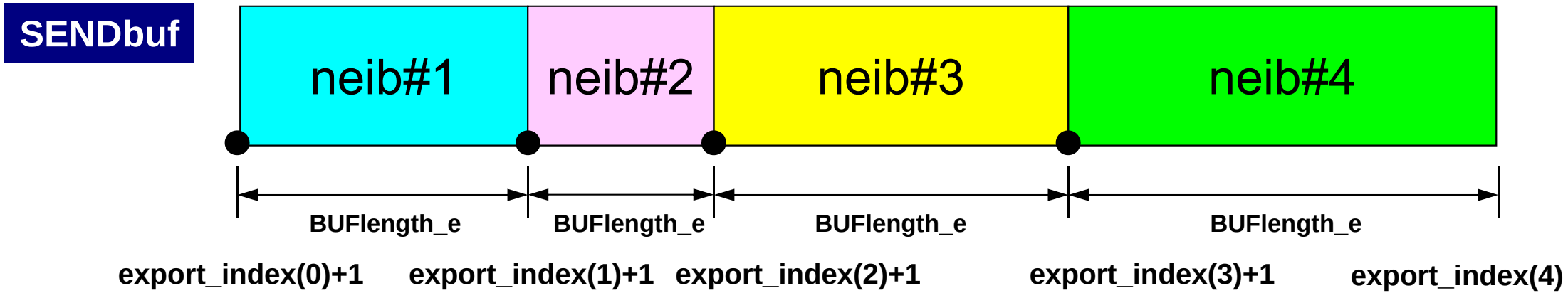


Fortran

一般化された通信テーブル：送信

- 送信相手
  - NEIBPETOT，NEIBPE(NEIBPETOT)
- それぞれの送信相手に送るメッセージサイズ
  - export_index(0:NEIBPETOT)
- 「境界点」番号
  - export_item(nn), nn= export_index(NEIBPETOT)
- それぞれの送信相手に送るメッセージ
  - SENDbuf(nn), nn=export_index(NEIBPETOT)

送信（MPI_Isend/Irecv/Waitall） Fortran




```
 do neib= 1, NEIBPETOT
   do k= export_index(neib-1)+1, export_index(neib)
     kk= export_item(k)
     SENDbuf(k)= VAL(kk)
   enddo
 enddo

 do neib= 1, NEIBPETOT
   iS_e= export_index(neib-1) + 1
   iE_e= export_index(neib  )
   BUFlength_e= iE_e + 1 - iS_e

   call MPI_ISEND                                                  &
&         (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&          MPI_COMM_WORLD, request_send(neib), ierr)
 enddo

 call MPI_WAITALL (NEIBPETOT, request_send, stat_recv, ierr)
```

送信バッファへの代入
温度などの変数を直接送信，受信に使うのではなく，このようなバッファへ一回代入して計算することを勧める。

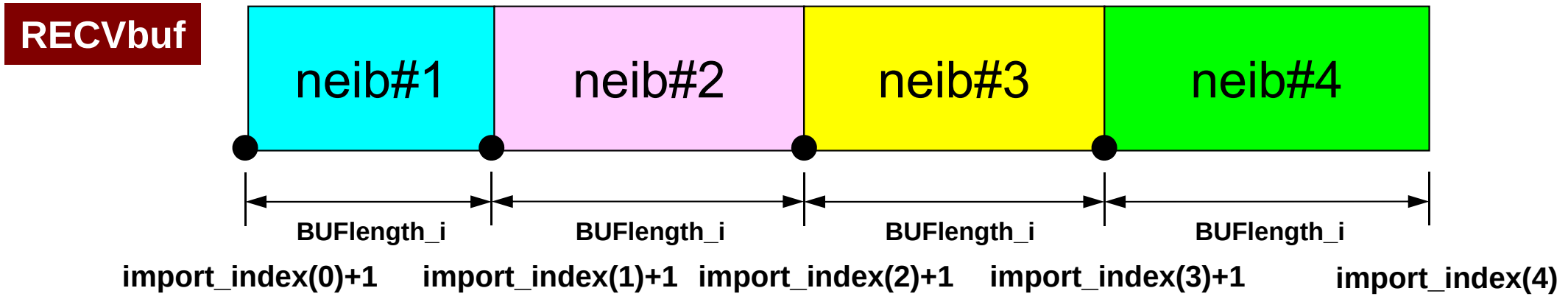
Fortran

一般化された通信テーブル：受信

- 受信相手
  - NEIBPETOT，NEIBPE(NEIBPETOT)
- それぞれの受信相手から受け取るメッセージサイズ
  - import_index(0:NEIBPETOT)
- 「外点」番号
  - import_item(nn), nn= import_index(NEIBPETOT)
- それぞれの受信相手から受け取るメッセージ
  - RECVbuf(nn), nn= import_index(NEIBPETOT)

受信（MPI_Isend/Irecv/Waitall） Fortran

```
  do neib= 1, NEIBPETOT
    iS_i= import_index(neib-1) + 1
    iE_i= import_index(neib  )
    BUFlength_i= iE_i + 1 - iS_i

    call MPI_IRECV                                                &
&         (RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&          MPI_COMM_WORLD, request_recv(neib), ierr)
  enddo

  call MPI_WAITALL (NEIBPETOT, request_recv, stat_recv, ierr)

  do neib= 1, NEIBPETOT
     do k= import_index(neib-1)+1, import_index(neib)
       kk= import_item(k)
       VAL(kk)= RECVbuf(k)                                  受信バッファから代入
     enddo
  enddo
```



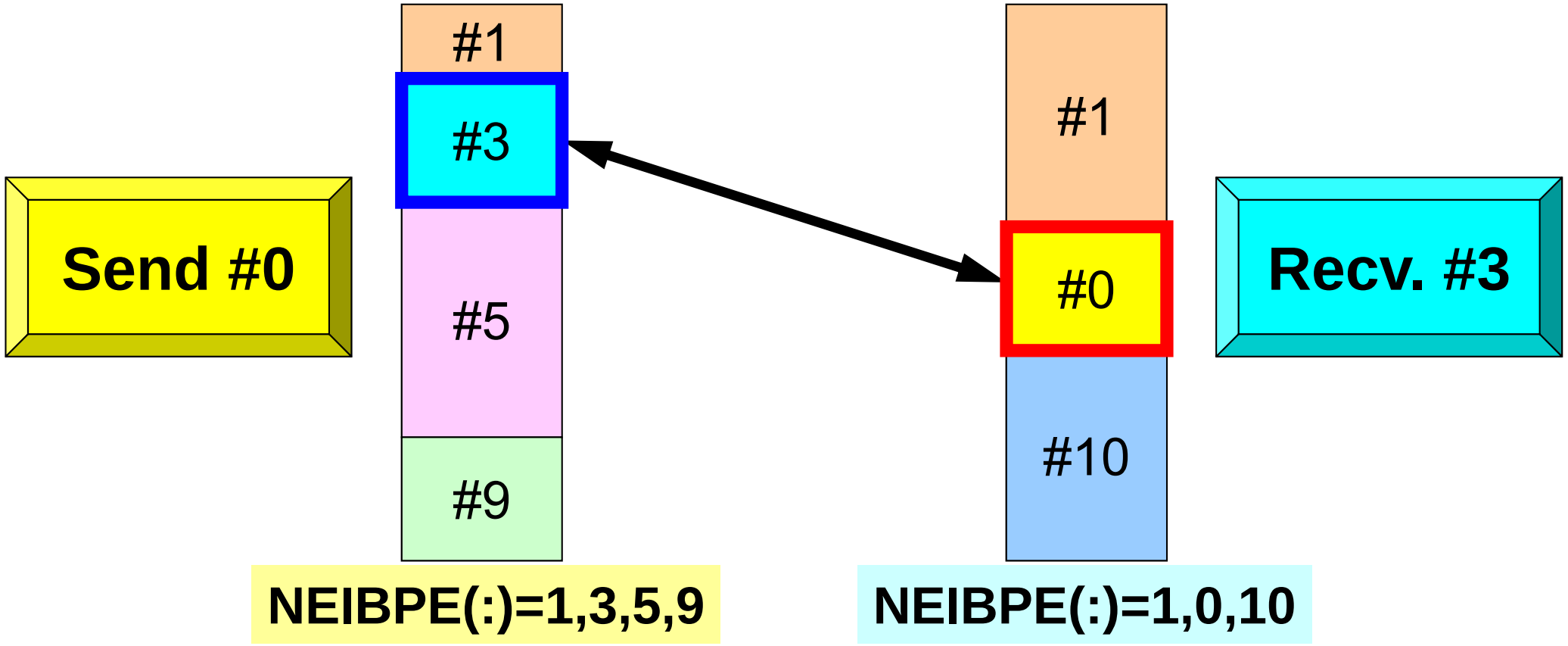
送信と受信の関係

```
 do neib= 1, NEIBPETOT
   iS_e= export_index(neib-1) + 1
   iE_e= export_index(neib  )
   BUFlength_e= iE_e + 1 - iS_e

   call MPI_ISEND                                                  &
&         (SENDbuf(iS_e), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&          MPI_COMM_WORLD, request_send(neib), ierr)
  enddo
```

```
 do neib= 1, NEIBPETOT
   iS_i= import_index(neib-1) + 1
   iE_i= import_index(neib  )
   BUFlength_i= iE_i + 1 - iS_i

   call MPI_IRECV                                                  &
&         (RECVbuf(iS_i), BUFlength_i, MPI_INTEGER, NEIBPE(neib), 0,&
&          MPI_COMM_WORLD, request_recv(neib), ierr)
  enddo
```

- 送信元・受信先プロセス番号，メッセージサイズ，内容の整合性 ！
- NEIBPE（neib）がマッチしたときに通信が起こる。

送信と受信の関係（#0⇒#3）

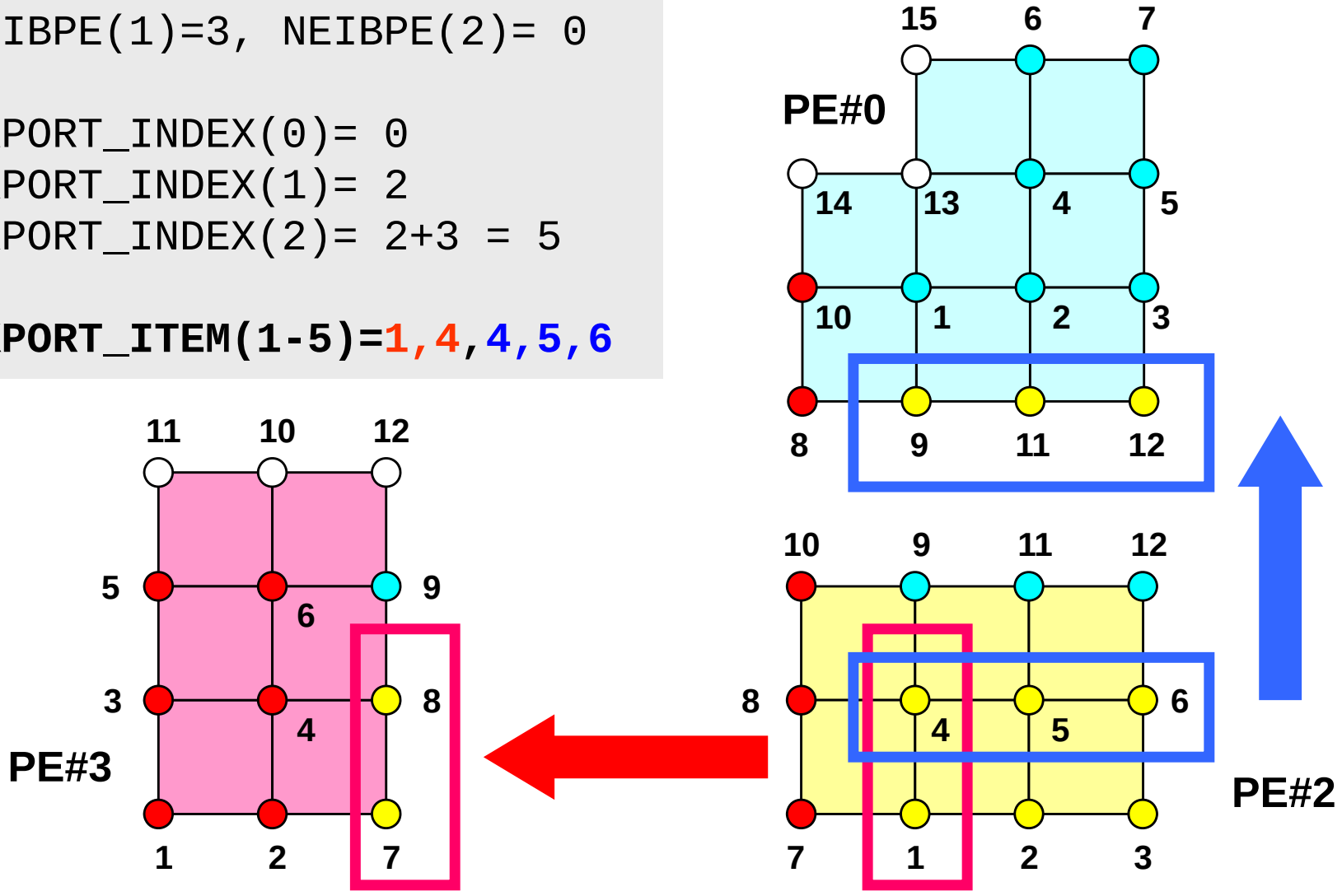


- 送信元・受信先プロセス番号，メッセージサイズ，内容の整合性 ！
- NEIBPE（neib）がマッチしたときに通信が起こる。

Example: SEND

PE#2 : send information on “boundary nodes”

```
NEIBPE= 2
NEIBPE(1)=3, NEIBPE(2)= 0

EXPORT_INDEX(0)= 0
EXPORT_INDEX(1)= 2
EXPORT_INDEX(2)= 2+3 = 5

EXPORT_ITEM(1-5)=1,4,4,5,6
```

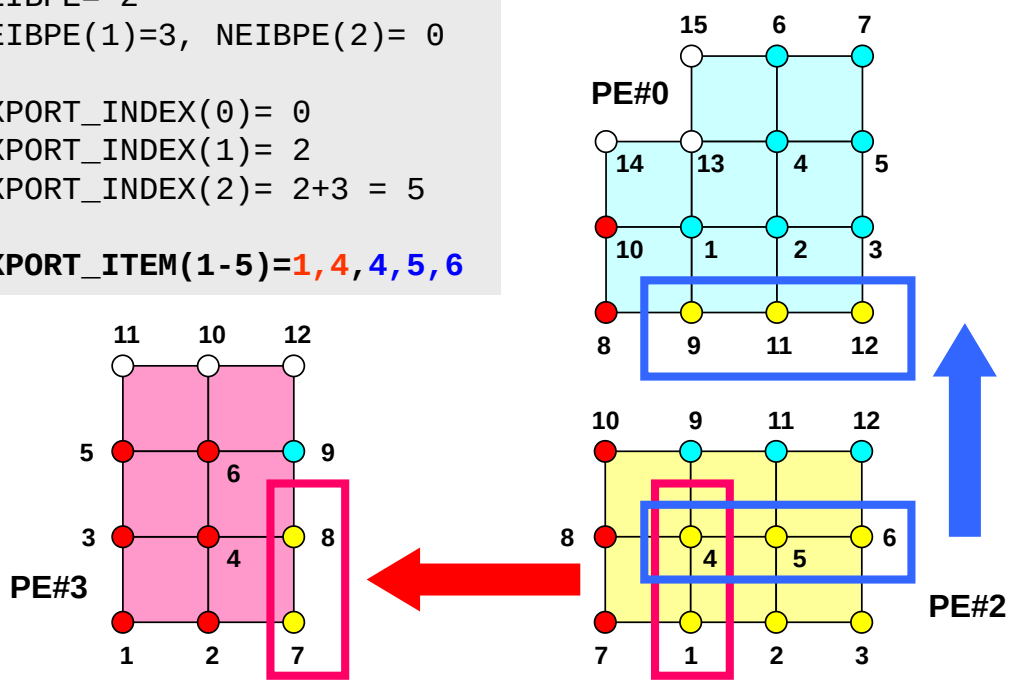


送信バッファの効用

Fortran

```
 do neib= 1, NEIBPETOT
   iS_e= export_index(neib-1) + 1
   iE_e= export_index(neib  )
   BUFlength_e= iE_e + 1 - iS_e

   call MPI_ISEND                                                  &
&         (VAL(...), BUFlength_e, MPI_INTEGER, NEIBPE(neib), 0,&
&          MPI_COMM_WORLD, request_send(neib), ierr)
 enddo
```

```
NEIBPE= 2
NEIBPE(1)=3, NEIBPE(2)= 0

EXPORT_INDEX(0)= 0
EXPORT_INDEX(1)= 2
EXPORT_INDEX(2)= 2+3 = 5

EXPORT_ITEM(1-5)=1,4,4,5,6
```



たとえば，この境界点は連続していないので，

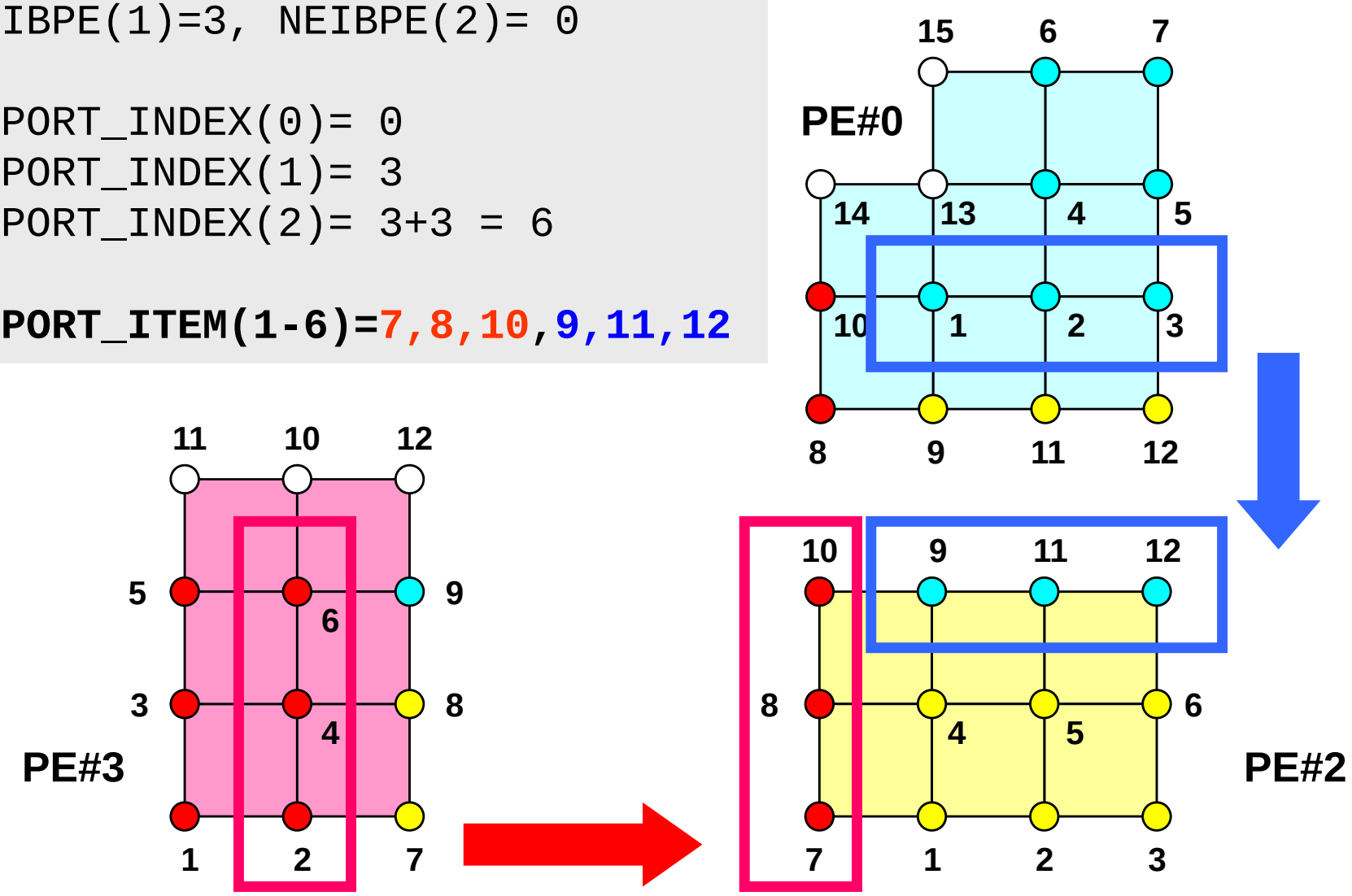
- 送信バッファの先頭アドレス
- そこから数えて●●のサイズのメッセージ

というような方法が困難

Example: RECEIVE

PE#2 : receive information for “external nodes”

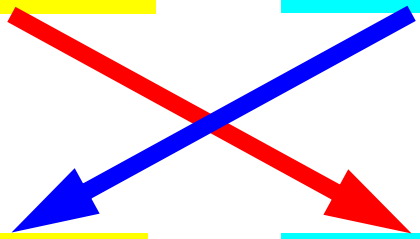
```
NEIBPE= 2
NEIBPE(1)=3, NEIBPE(2)= 0

IMPORT_INDEX(0)= 0
IMPORT_INDEX(1)= 3
IMPORT_INDEX(2)= 3+3 = 6

IMPORT_ITEM(1-6)=7,8,10,9,11,12
```



配列の送受信:注意

```
#PE0
  send:
    SENDbuf(iS_e)~
    SENDbuf(iE_e+BUFlength_e-1)
```

```
#PE1
  send:
    SENDbuf(iS_e)~
    SENDbuf(iE_e+BUFlength_e-1)
```

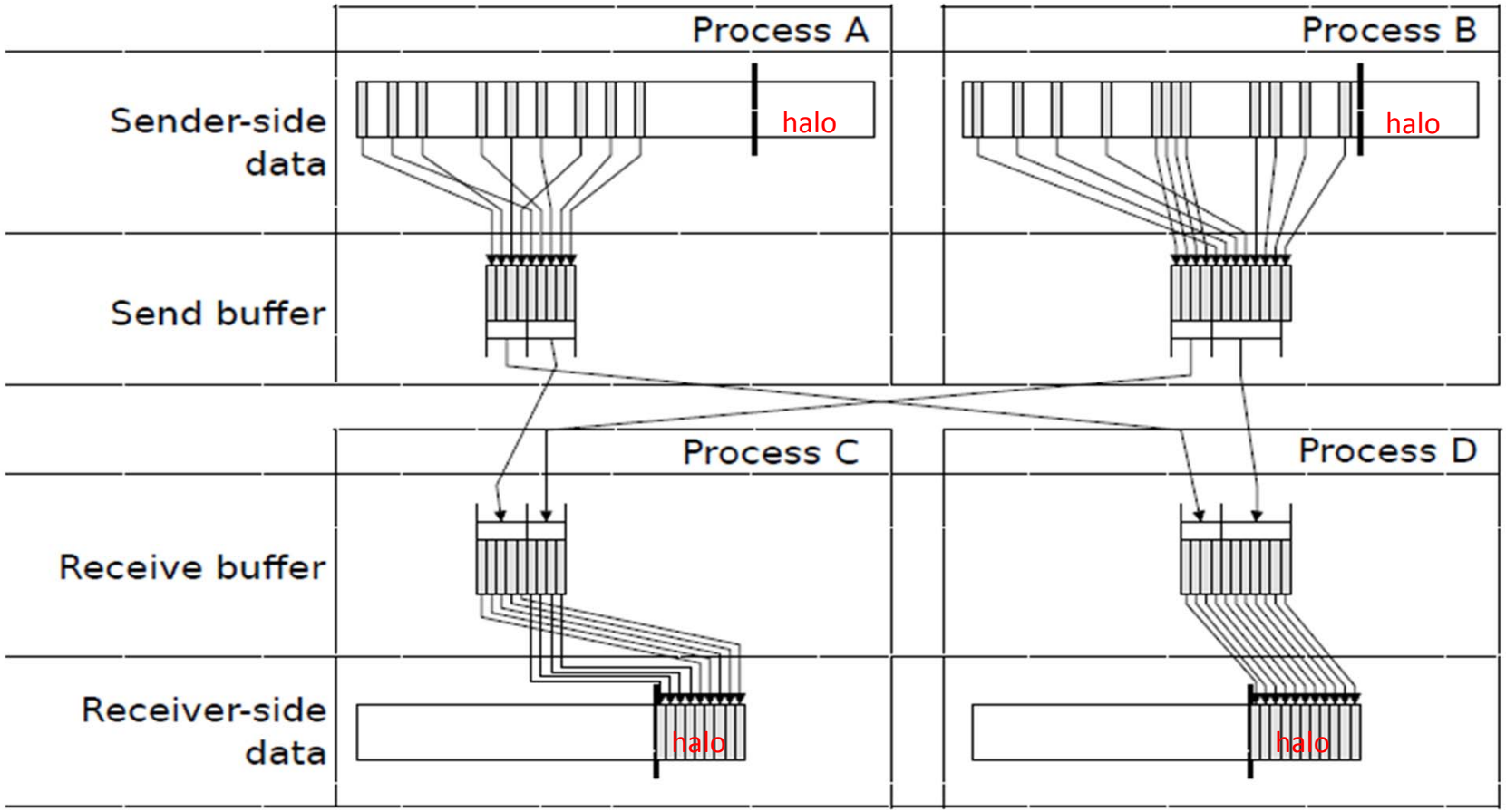


```
#PE0
  recv:
    RECVbuf(iS_i)~
    RECVbuf(iE_i+Buflength_i-1)
```

```
#PE1
  recv:
    RECVbuf(iS_i)~
    RECVbuf(iE_i+Buflength_i-1)
```

- 送信側の「BUFlength_e」と受信側の「BUFlength_i」は一致している必要がある。
  - PE#0⇒PE#1，PE#1⇒PE#0
- 「送信バッファ」と「受信バッファ」は別のアドレス

Communication Pattern using 1D Structure



Dr. Osni Marques
（Lawrence Berkeley National Laboratory）

並列計算向け局所(分散)データ構造

- 差分法，有限要素法，有限体積法等係数が疎行列のアプリケーションについては領域間通信はこのような局所(分散)データによって実施可能
  - SPMD
  - 内点～外点の順に「局所」番号付け
  - 通信テーブル：一般化された通信テーブル

- 適切なデータ構造が定められれば，処理は非常に簡単。
  - 送信バッファに「境界点」の値を代入
  - 送信，受信
  - 受信バッファの値を「外点」の値として更新