並列有限要素法への道
―並列データ構造―
C言語編

中島 研吾
東京大学情報基盤センター

並列計算の目的

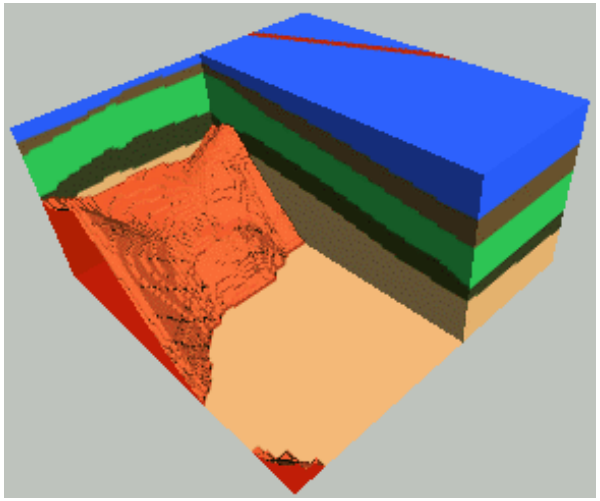
- 高速，大規模
  - 「大規模」の方が「新しい科学」という観点からのウェイトとしては高い。しかし，「高速」ももちろん重要である。
    - 細かいメッシュ
- +複雑
- 理想：Scalable
  - N倍の規模の計算をN倍のCPUを使って，「同じ時間で」解く（大規模性の追求：Weak Scaling）
    - 実際はそうは行かない
      - 例：共役勾配法⇒問題規模が大きくなると反復回数が増加
  - 同じ問題をN倍のCPUを使って「1/Nの時間で」解く・・・という場合もある（高速性の追求：Strong Scaling）
    - これも余り簡単な話では無い

並列計算とは？(1/2)

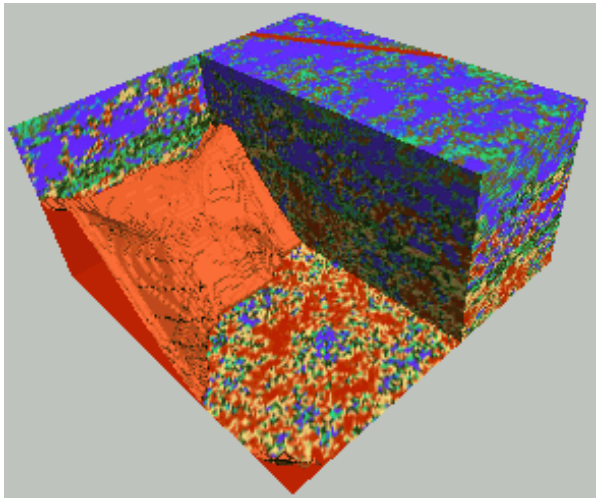
- より大規模で複雑な問題を高速に解きたい

Homogeneous/Heterogeneous Porous Media

Lawrence Livermore National Laboratory



Homogeneous

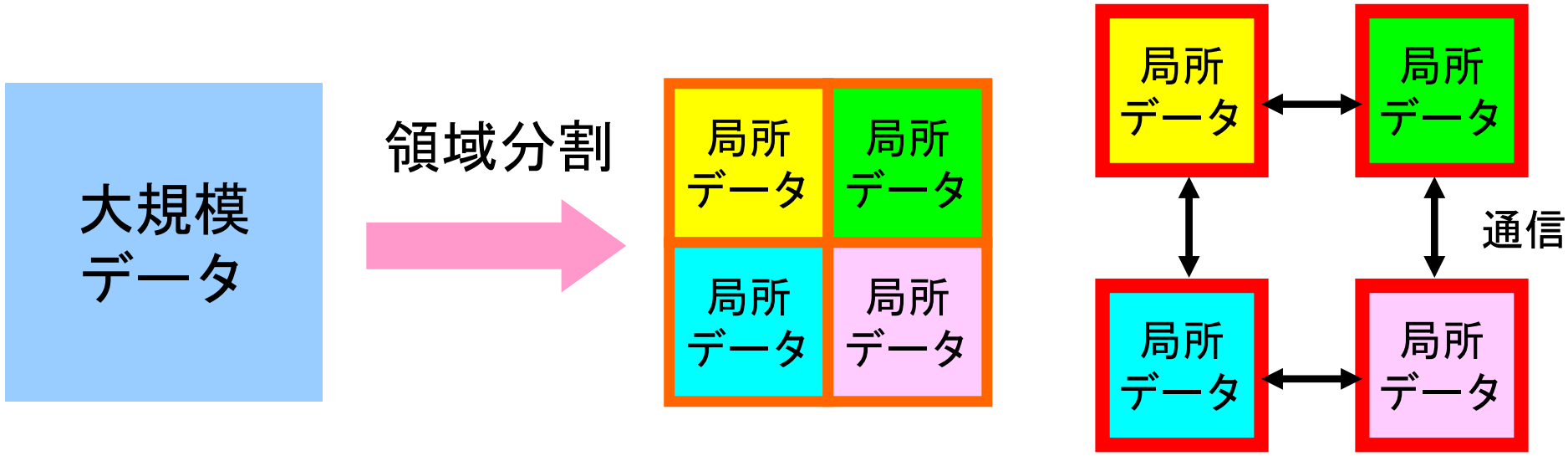


Heterogeneous

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

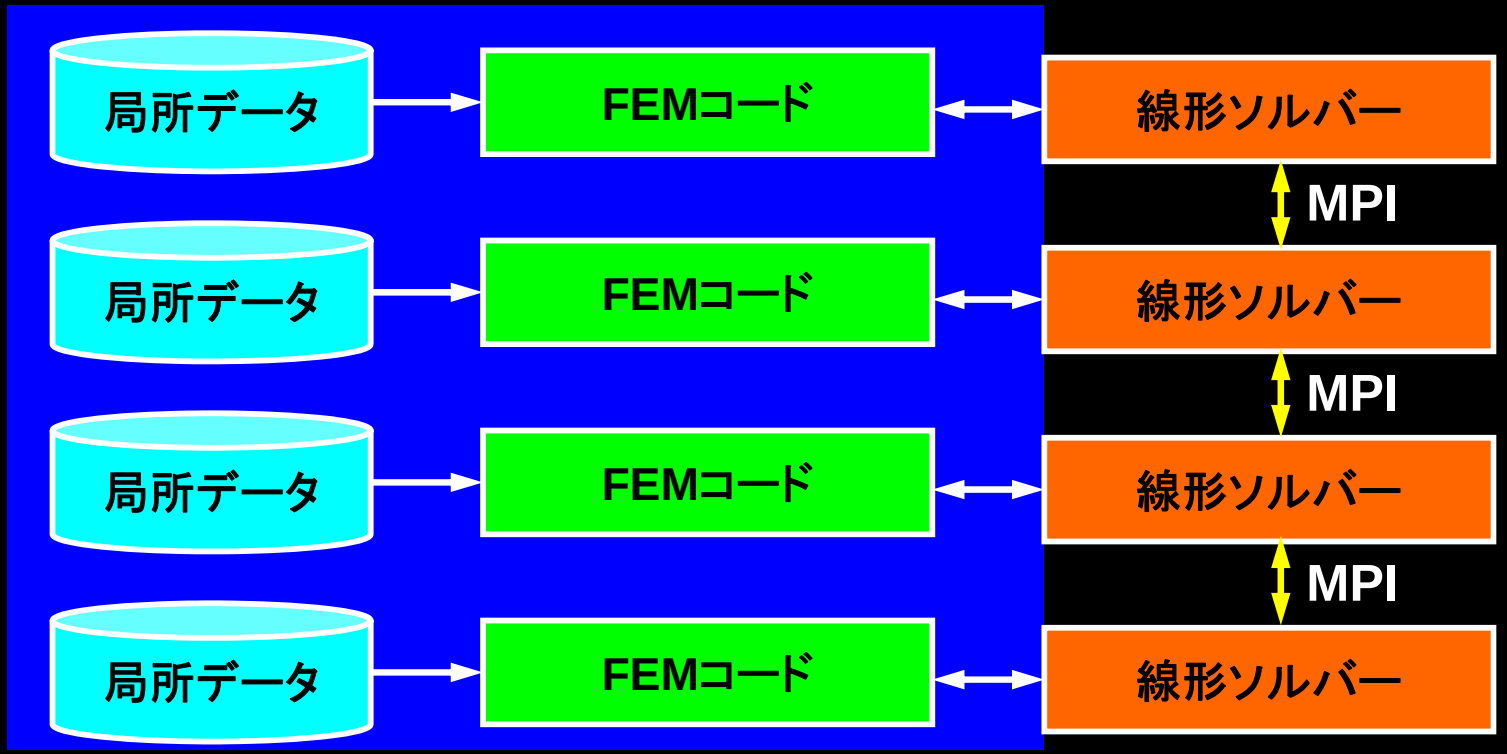
- 「大域的（global）」な情報を必要とする場合に通信が生じる（必要となる）。
  - 全体マトリクスを線形ソルバーで解く

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

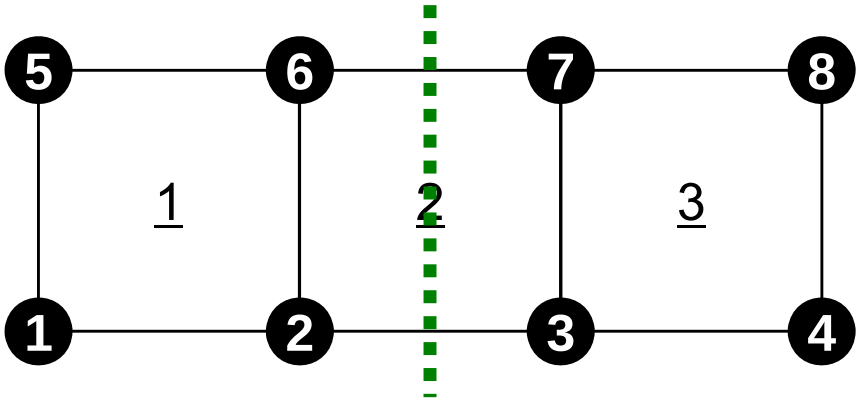


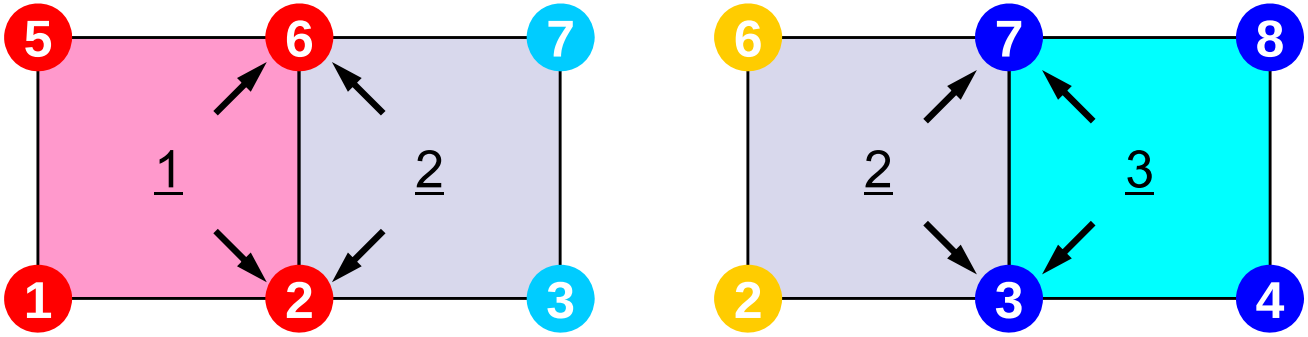
並列有限要素法プログラムの開発

- 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





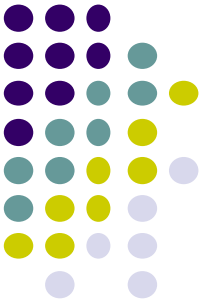
「節点ベース（領域ごとの節点数がバランスする）」の分割

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

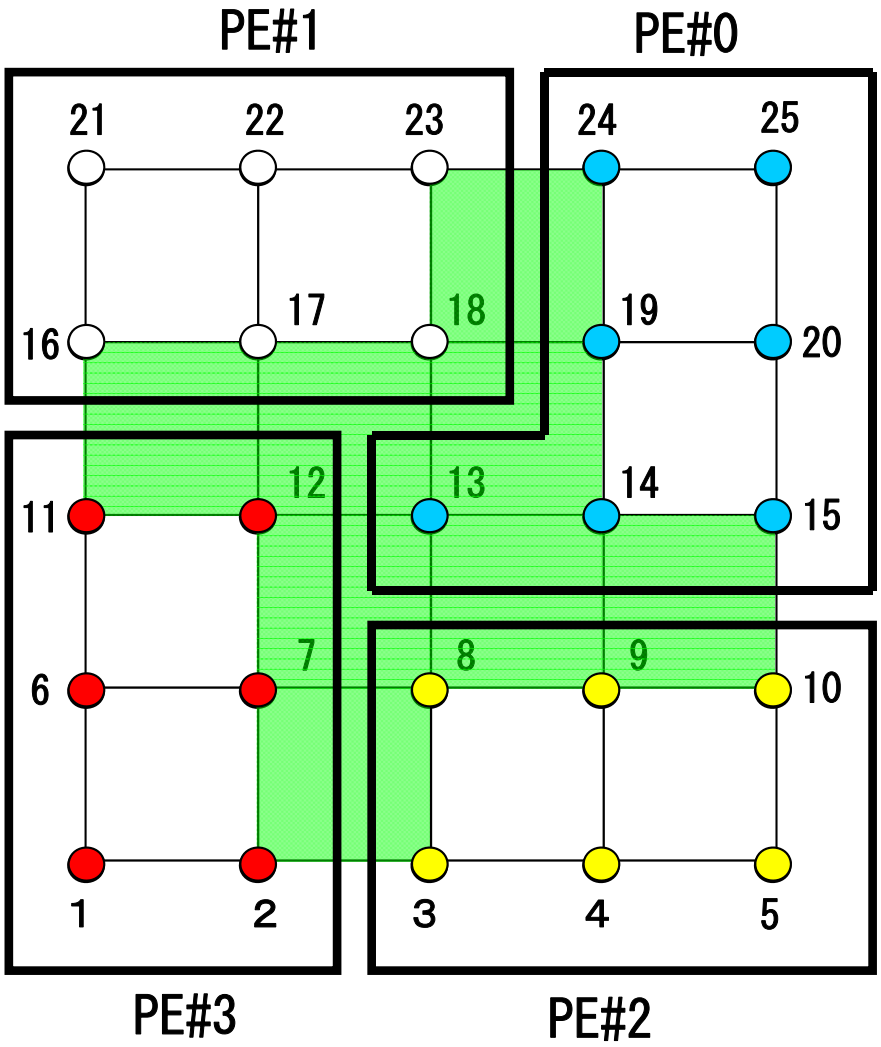


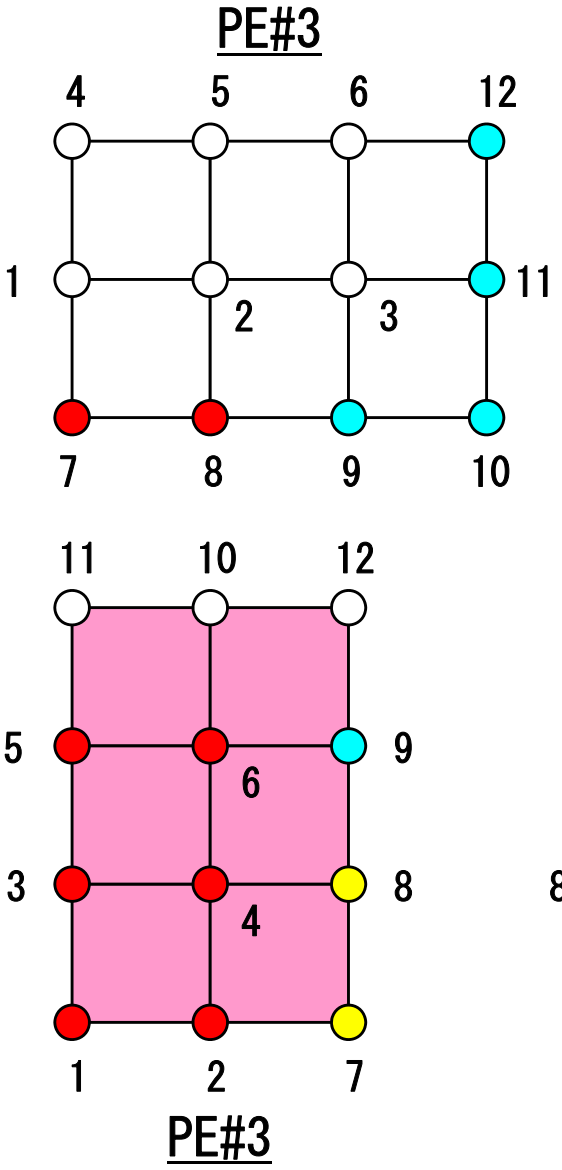
- 節点ベース：Node-based partitioning
- 局所データに含まれるもの：
  - その領域に本来含まれる節点
  - それらの節点を含む要素
  - 本来領域外であるが，それらの要素に含まれる節点
- 節点は以下の3種類に分類
  - 内点：Internal nodes　　その領域に本来含まれる節点
  - 外点：External nodes　　本来領域外であるがマトリクス生成に必要な節点
  - 境界点：Boundary nodes　　他の領域の「外点」となっている節点
- 領域間の通信テーブル

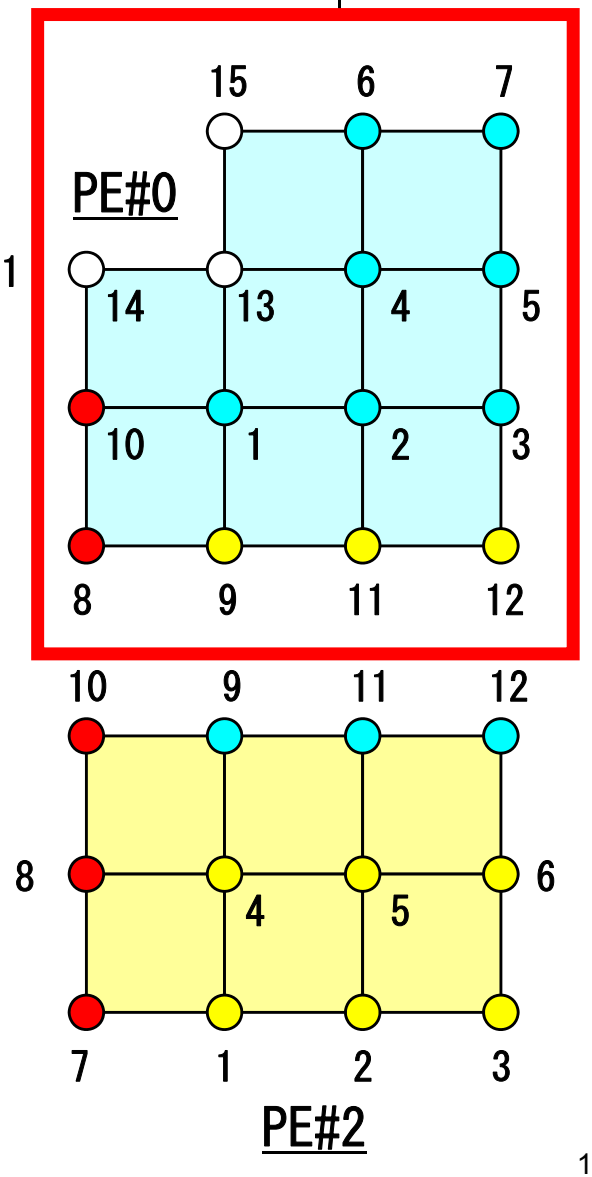
- 領域間の接続をのぞくと，大域的な情報は不要
  - 有限要素法の特性：要素で閉じた計算

Node-based Partitioning

internal nodes - elements - external nodes

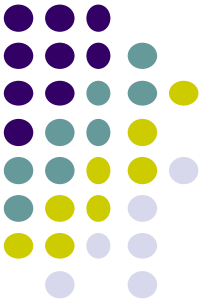






Node-based Partitioning

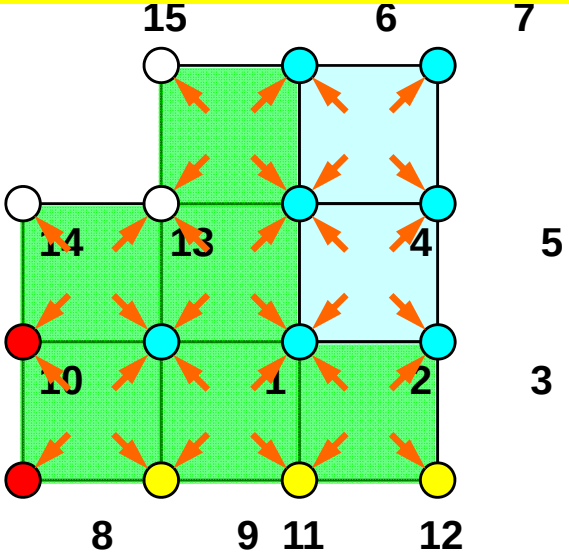
internal nodes - elements - external nodes

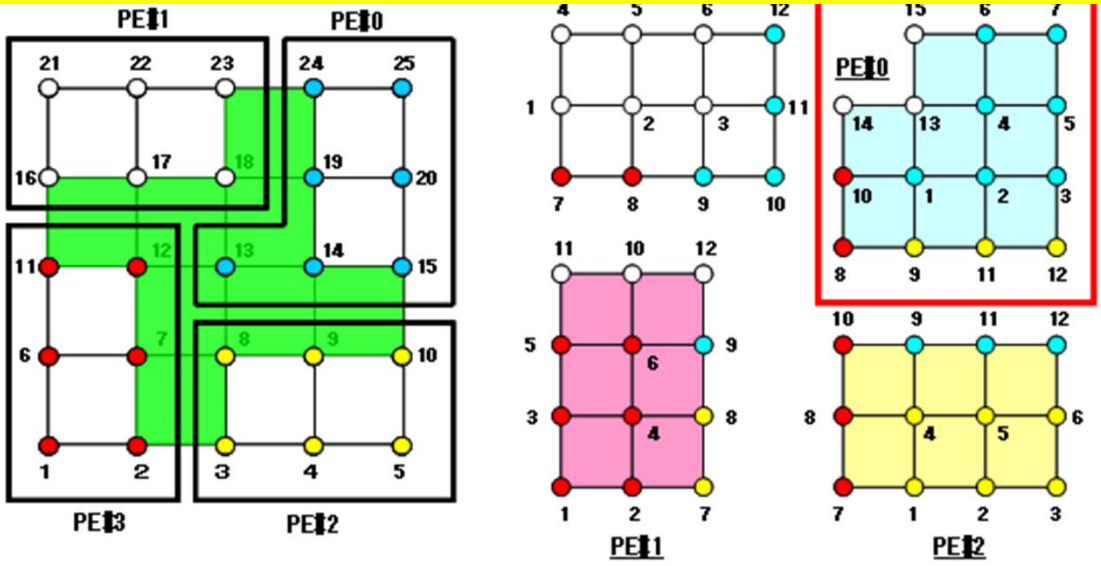


- Partitioned nodes themselves (Internal Nodes) 内点
- Elements which include Internal Nodes 内点を含む要素
- External Nodes included in the Elements 外点

  in overlapped region among partitions.

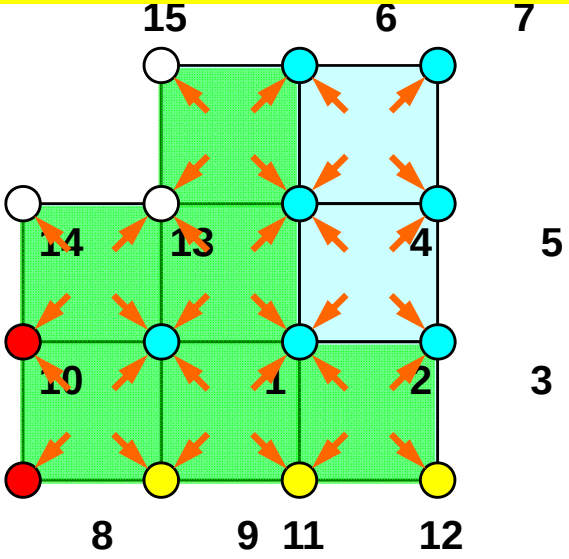
- Info of External Nodes are required for completely local element–based operations on each processor.

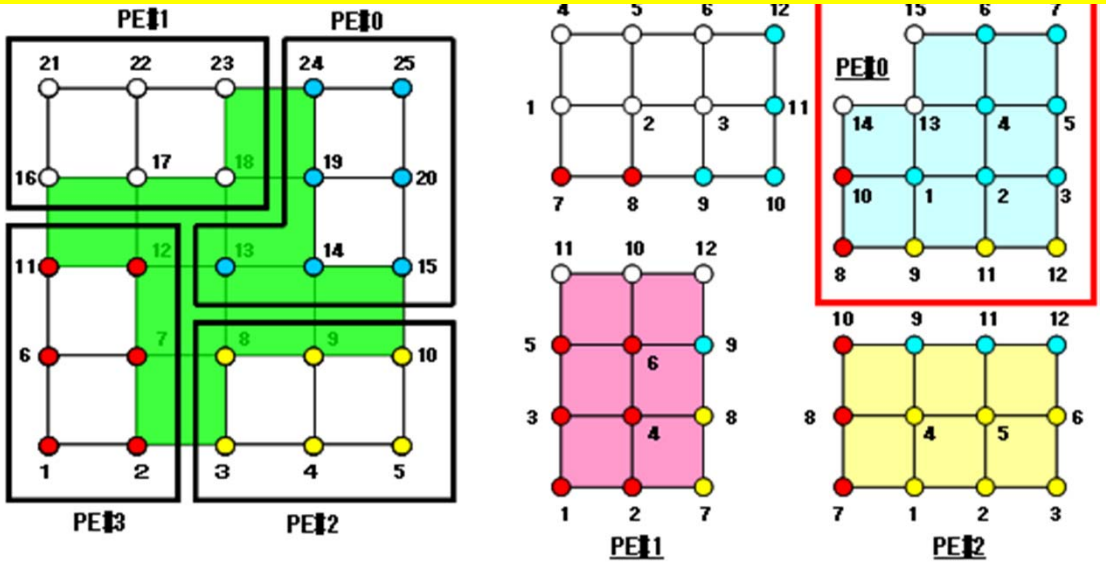




マトリクス生成時の通信は不要

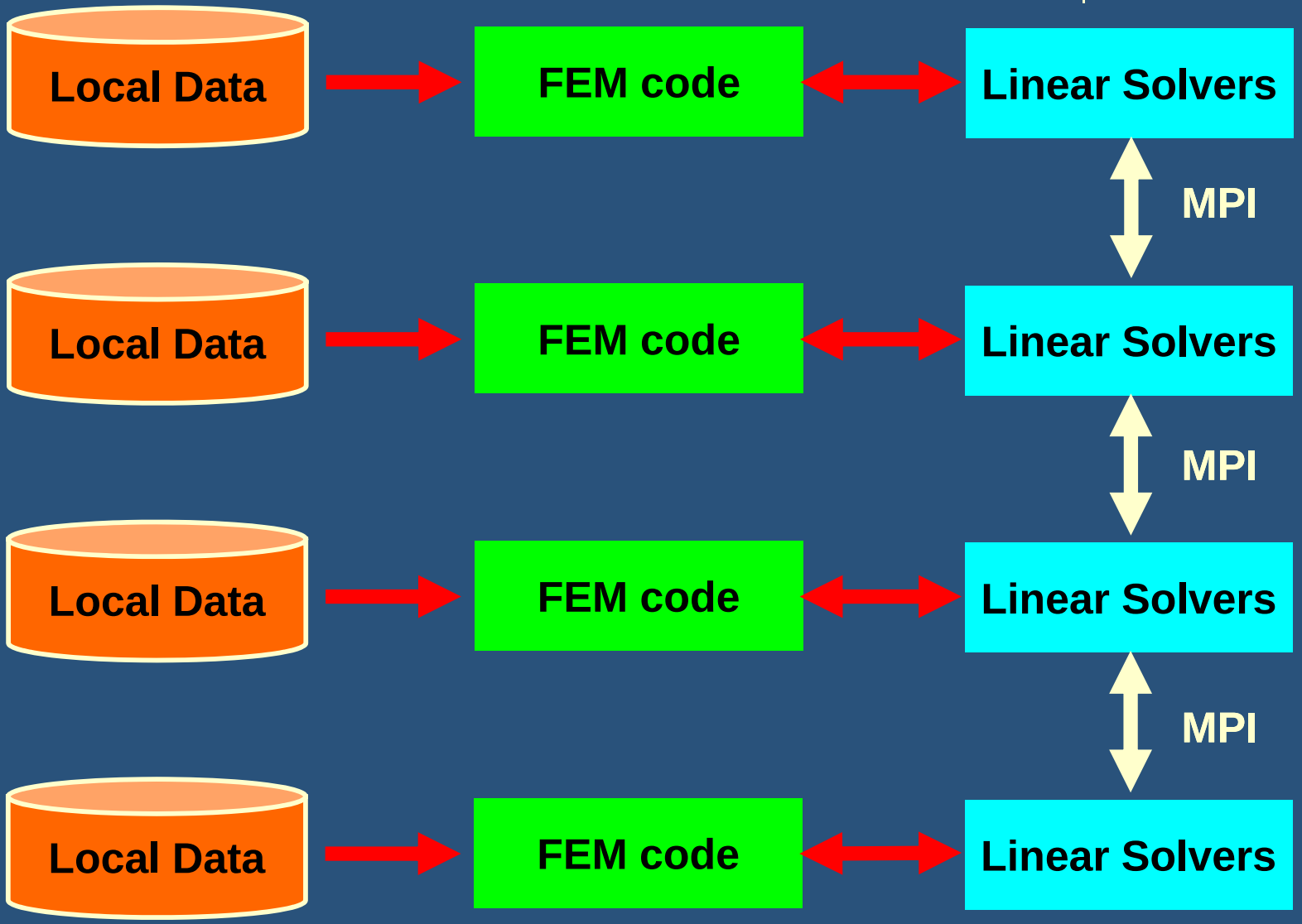
- Partitioned nodes themselves (Internal Nodes)　内点
- Elements which include Internal Nodes　内点を含む要素
- External Nodes included in the Elements　外点
  in overlapped region among partitions.
- Info of External Nodes are required for completely local element–based operations on each processor.



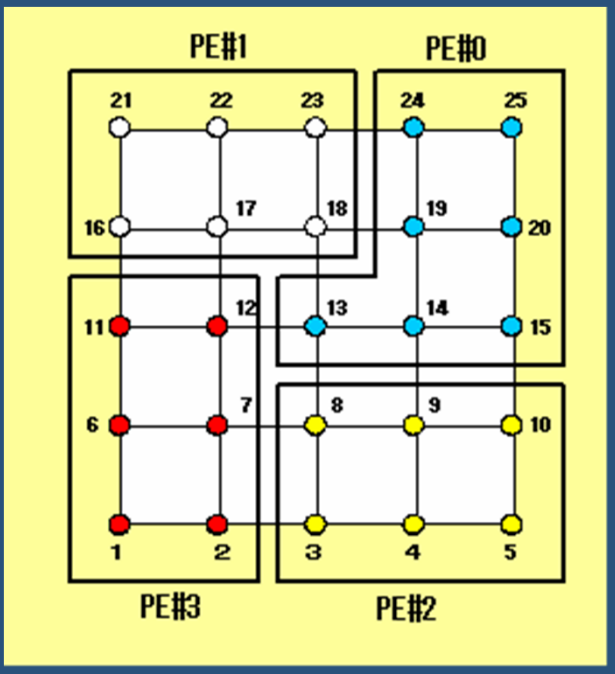


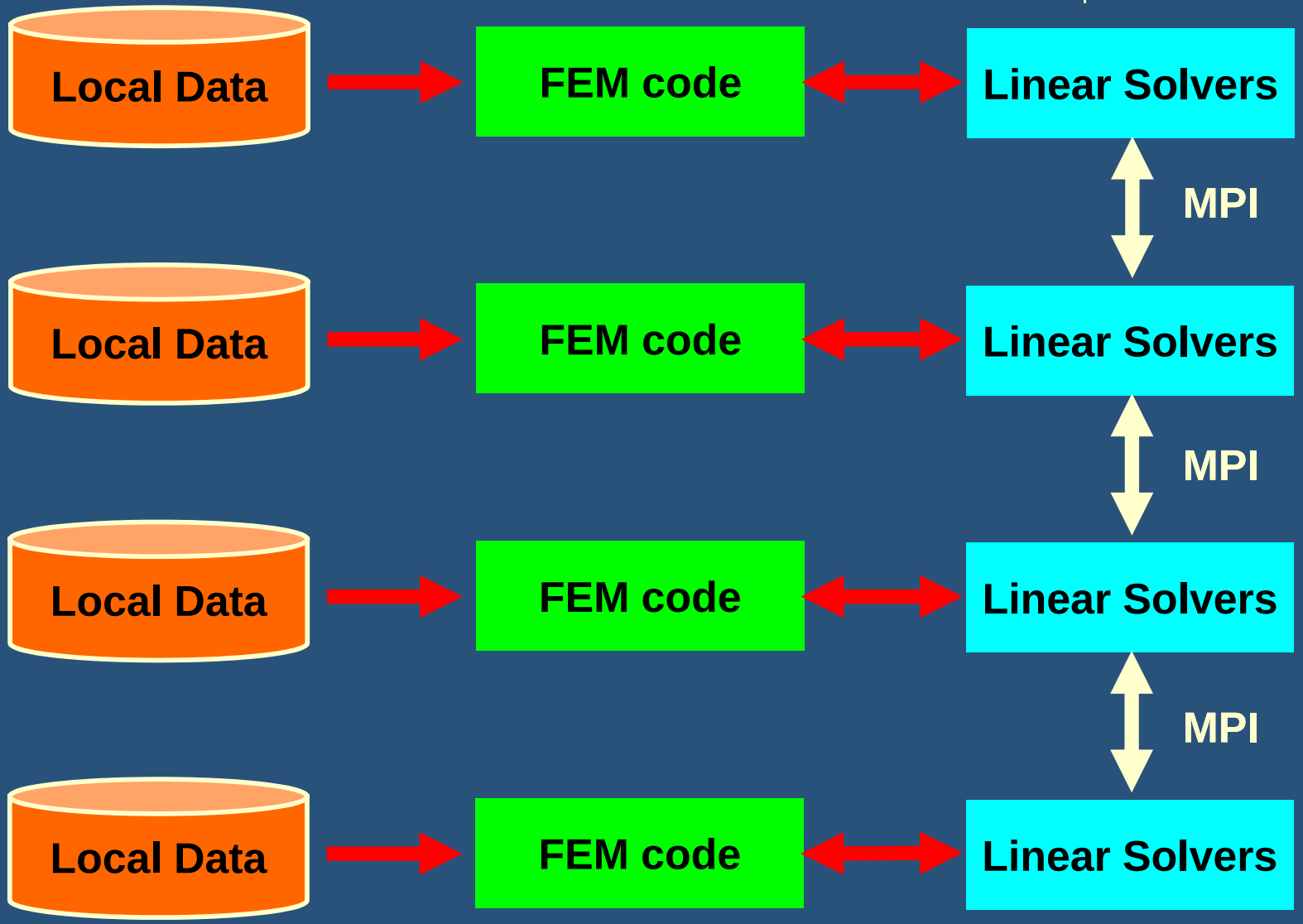
Parallel Computing in FEM

SPMD: Single-Program Multiple-Data



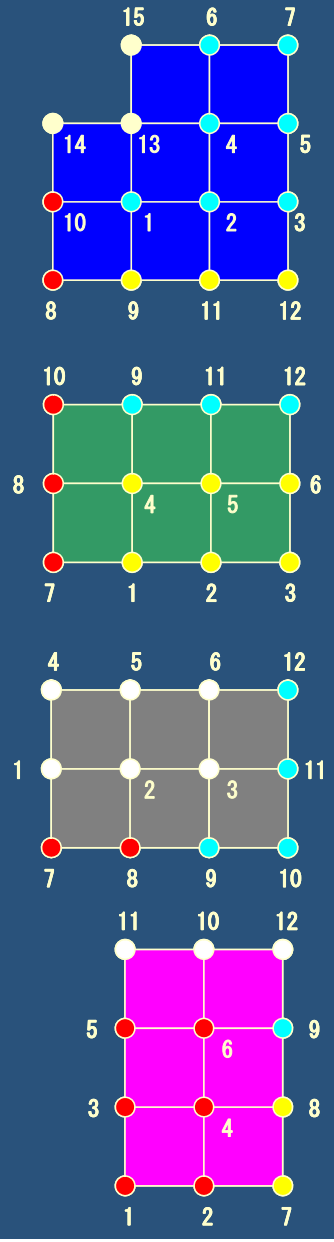
Parallel Computing in FEM
SPMD: Single-Program Multiple-Data

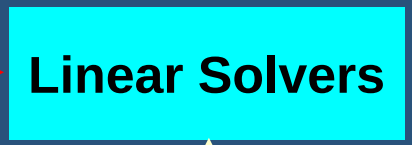


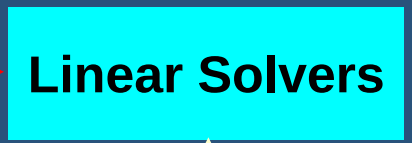


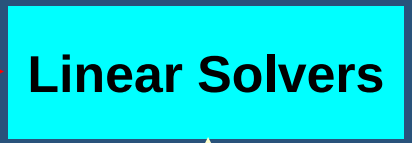
Parallel Computing in FEM

SPMD: Single-Program Multiple-Data



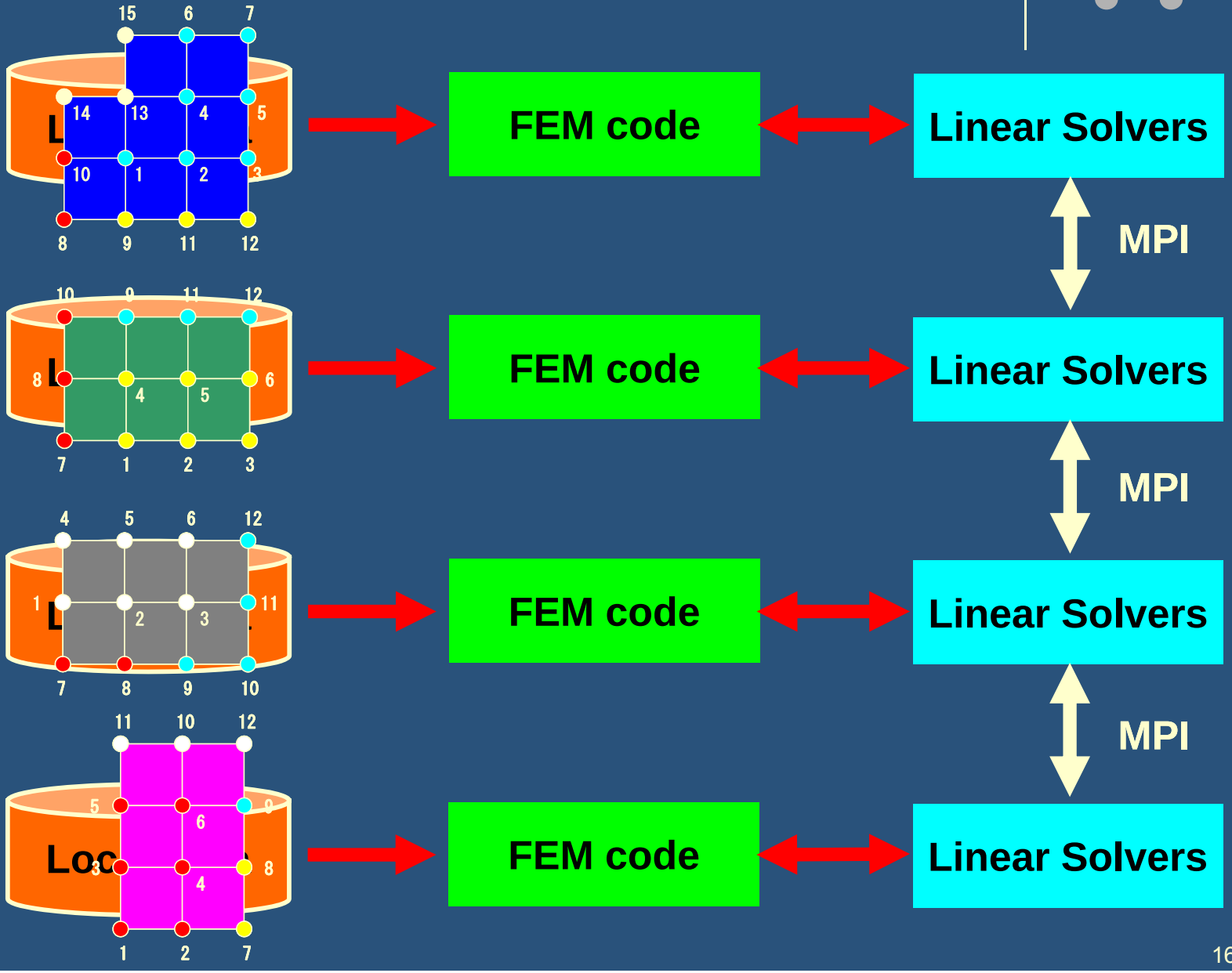






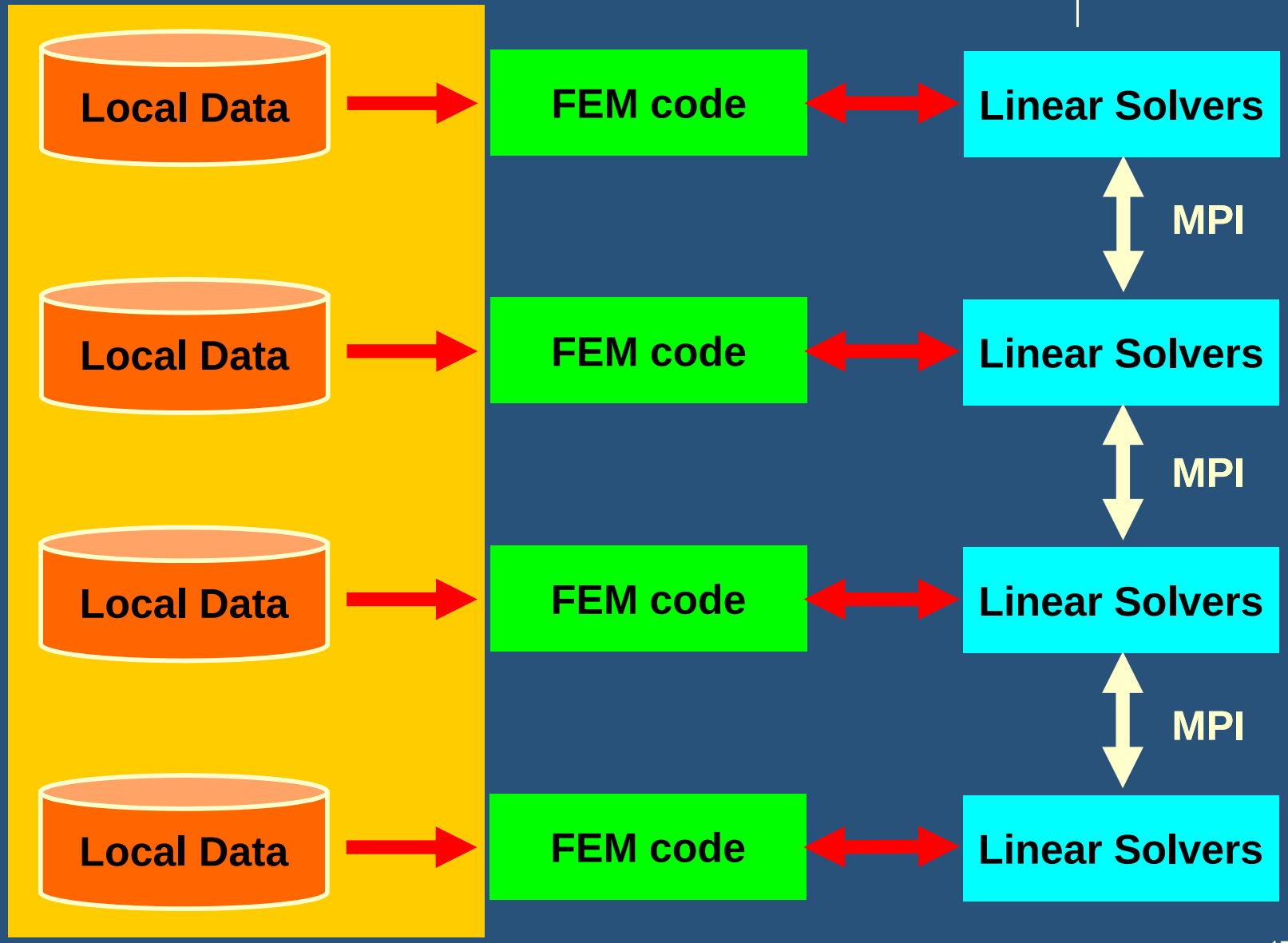
Parallel Computing in FEM

SPMD: Single-Program Multiple-Data

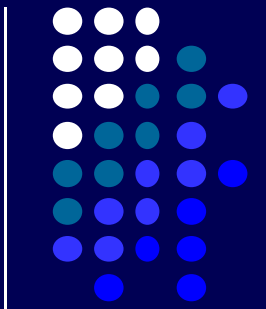


Parallel Computing in FEM

SPMD: Single-Program Multiple-Data

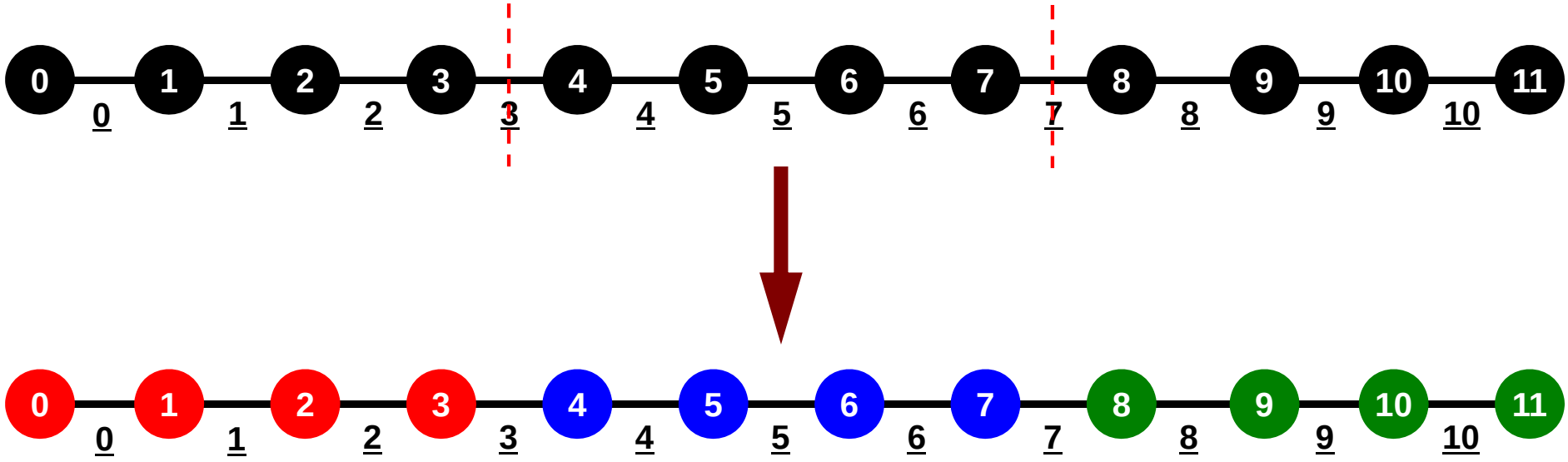


通信とは何か？

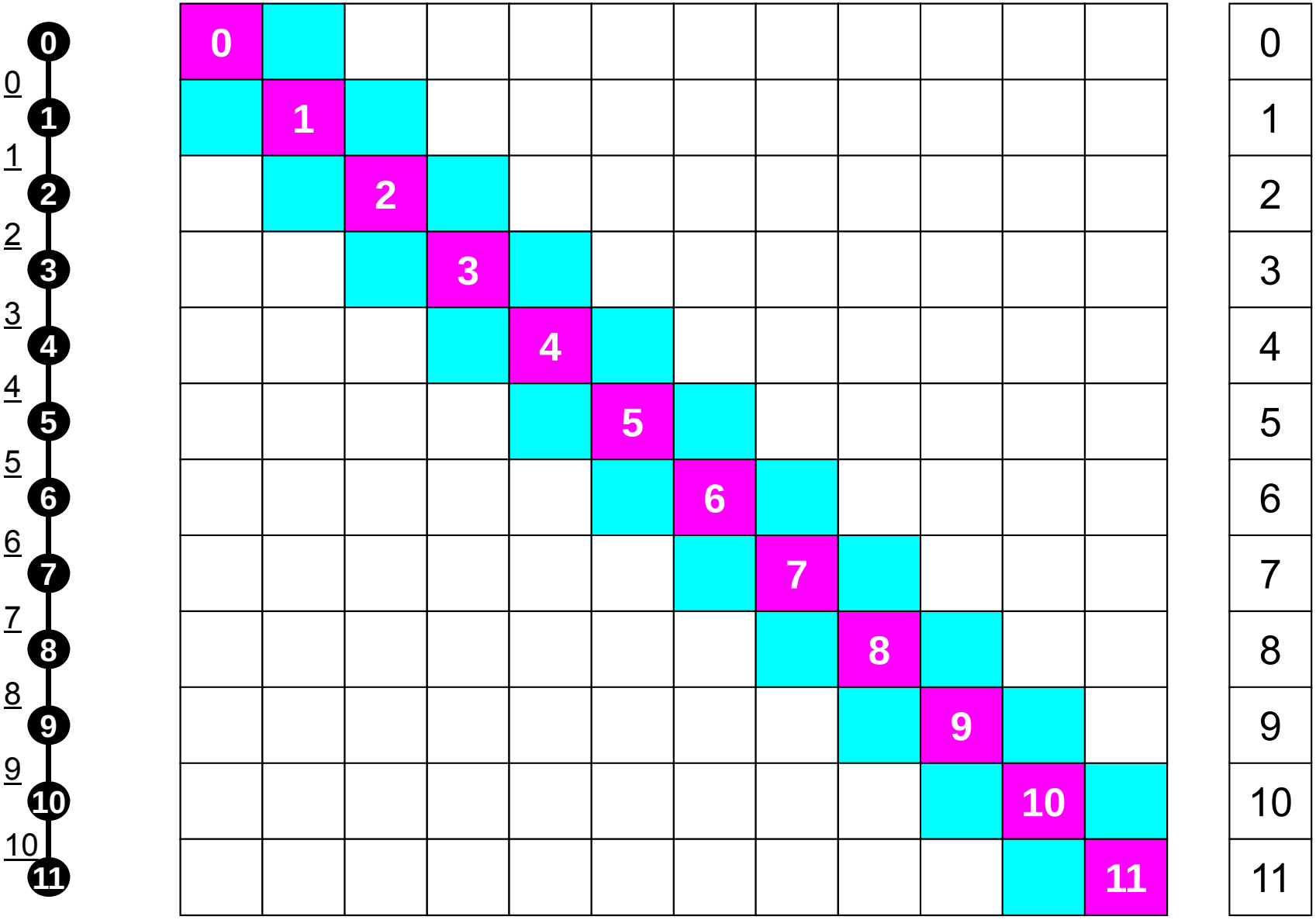


- 「外点」の情報を外部の領域からもらってくること
- 「通信テーブル」にその情報が含まれている

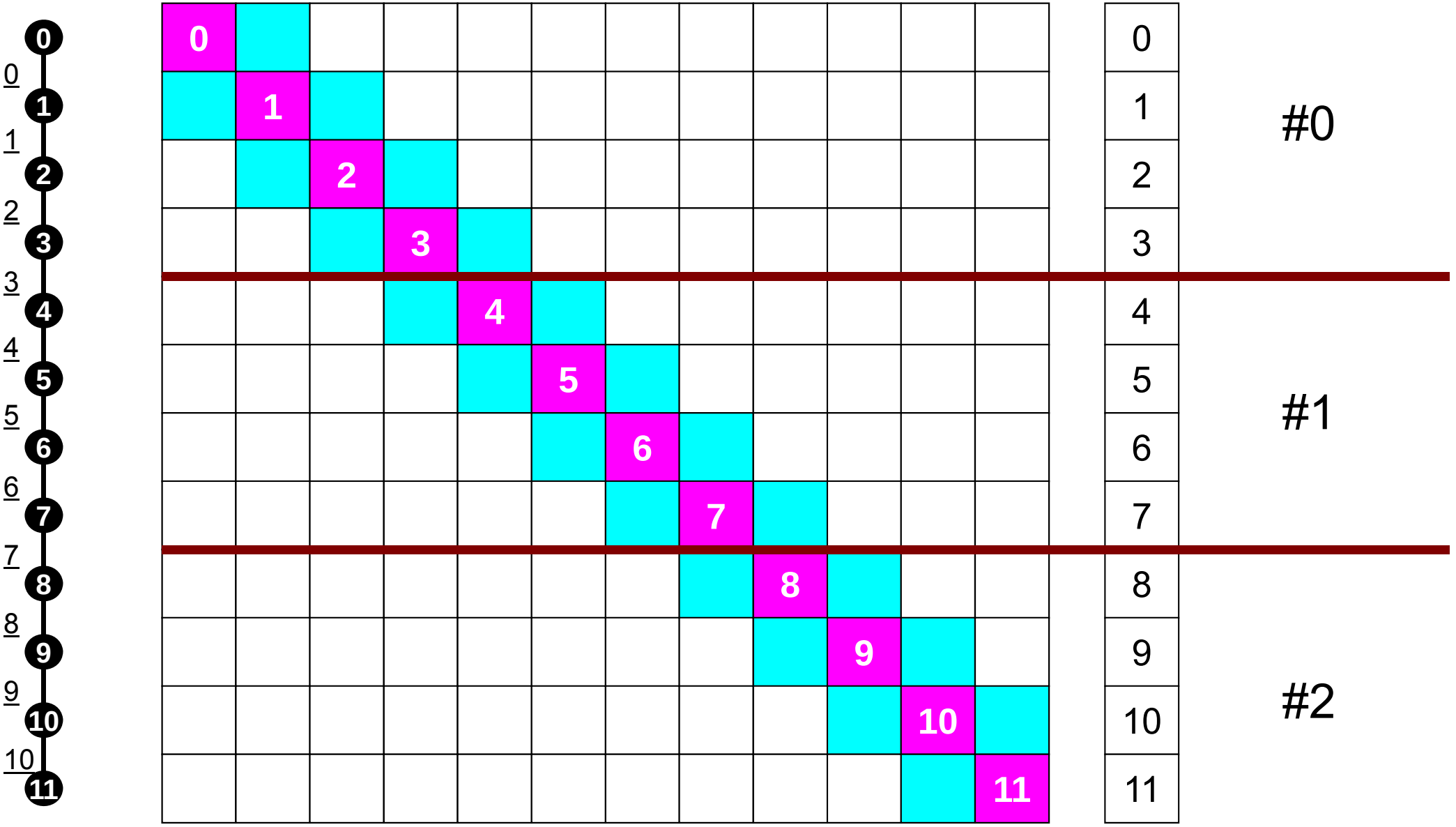
一次元問題:11要素，12節点，3領域



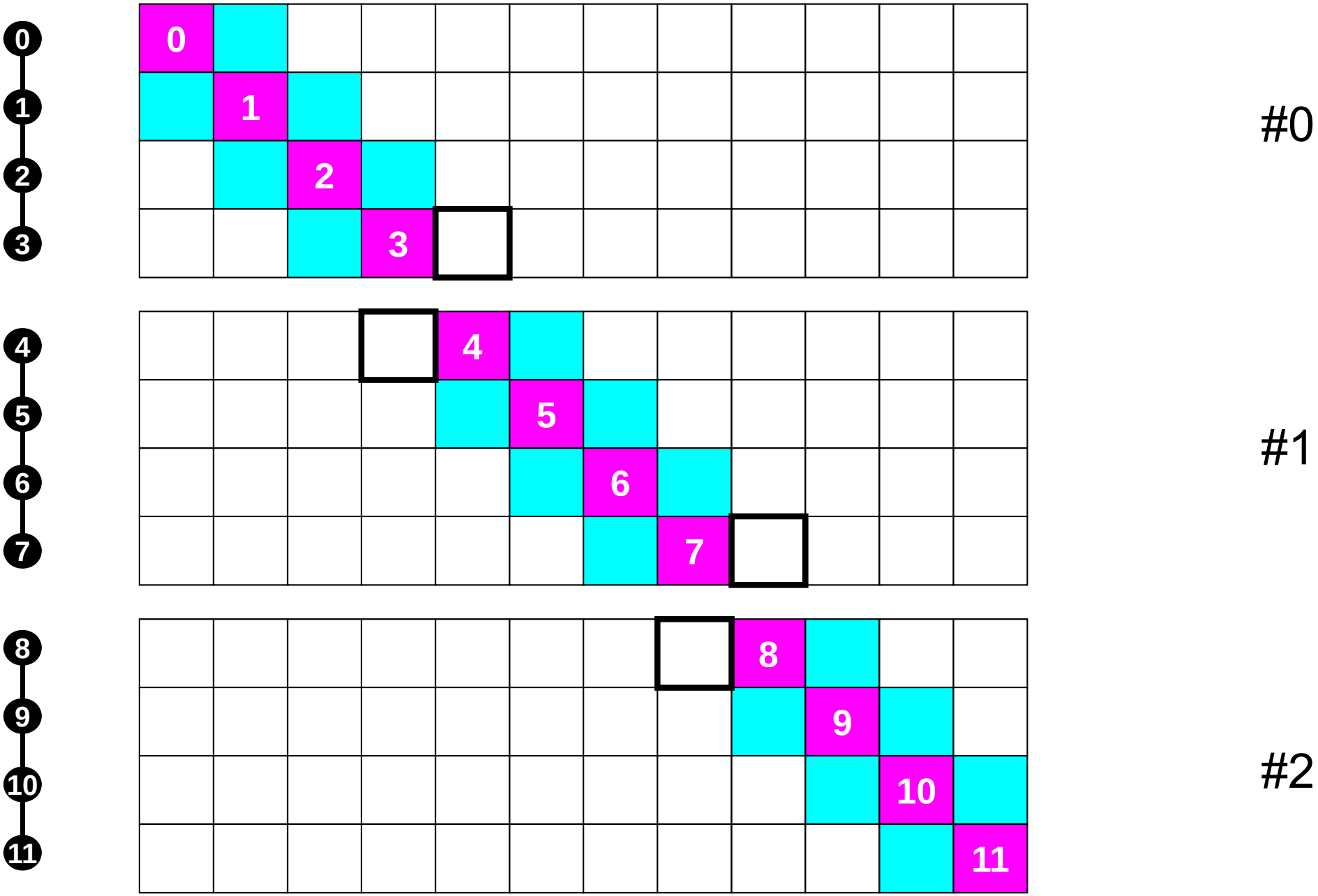
一次元問題：11要素，12節点，3領域



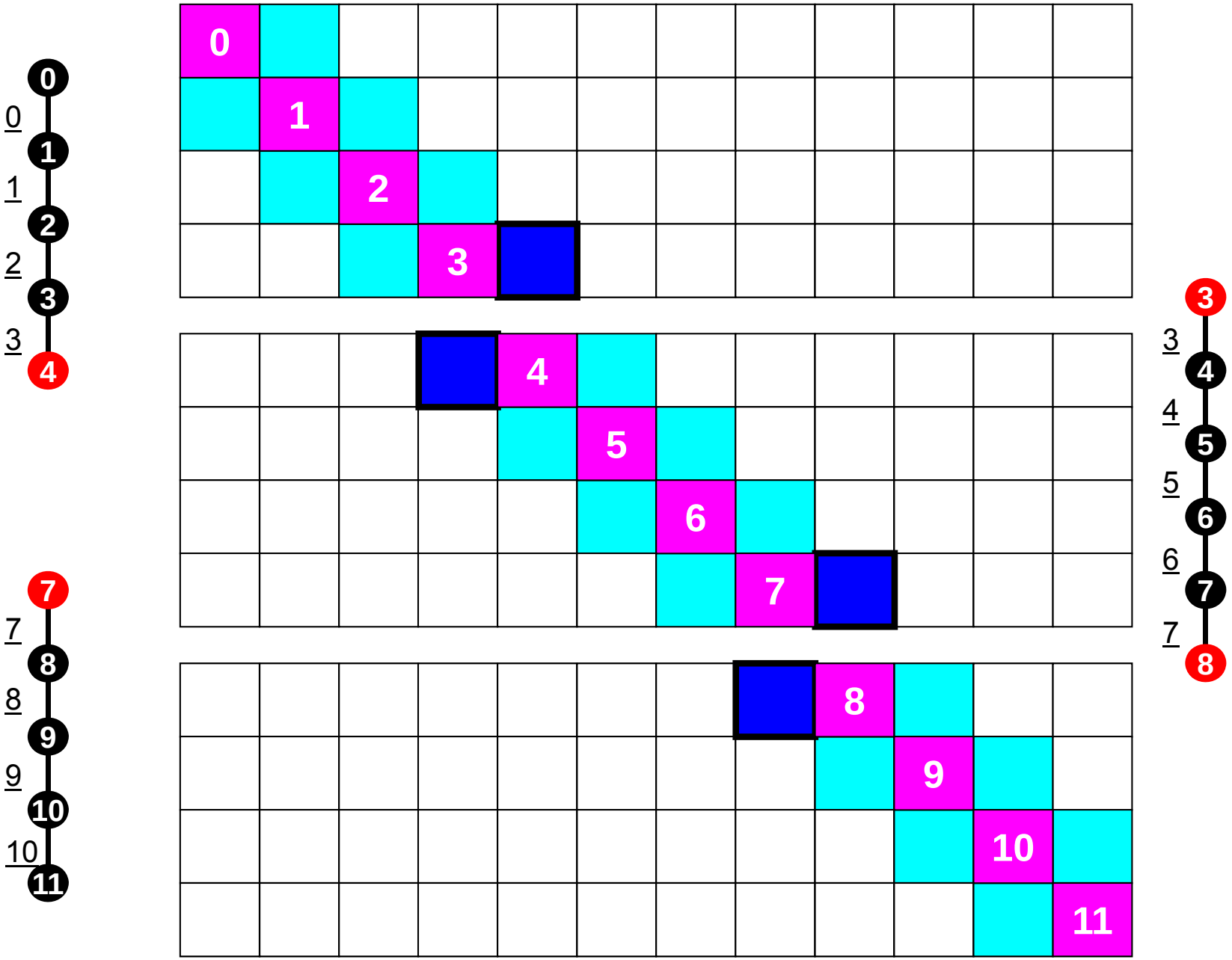
節点がバランスするよう分割：内点



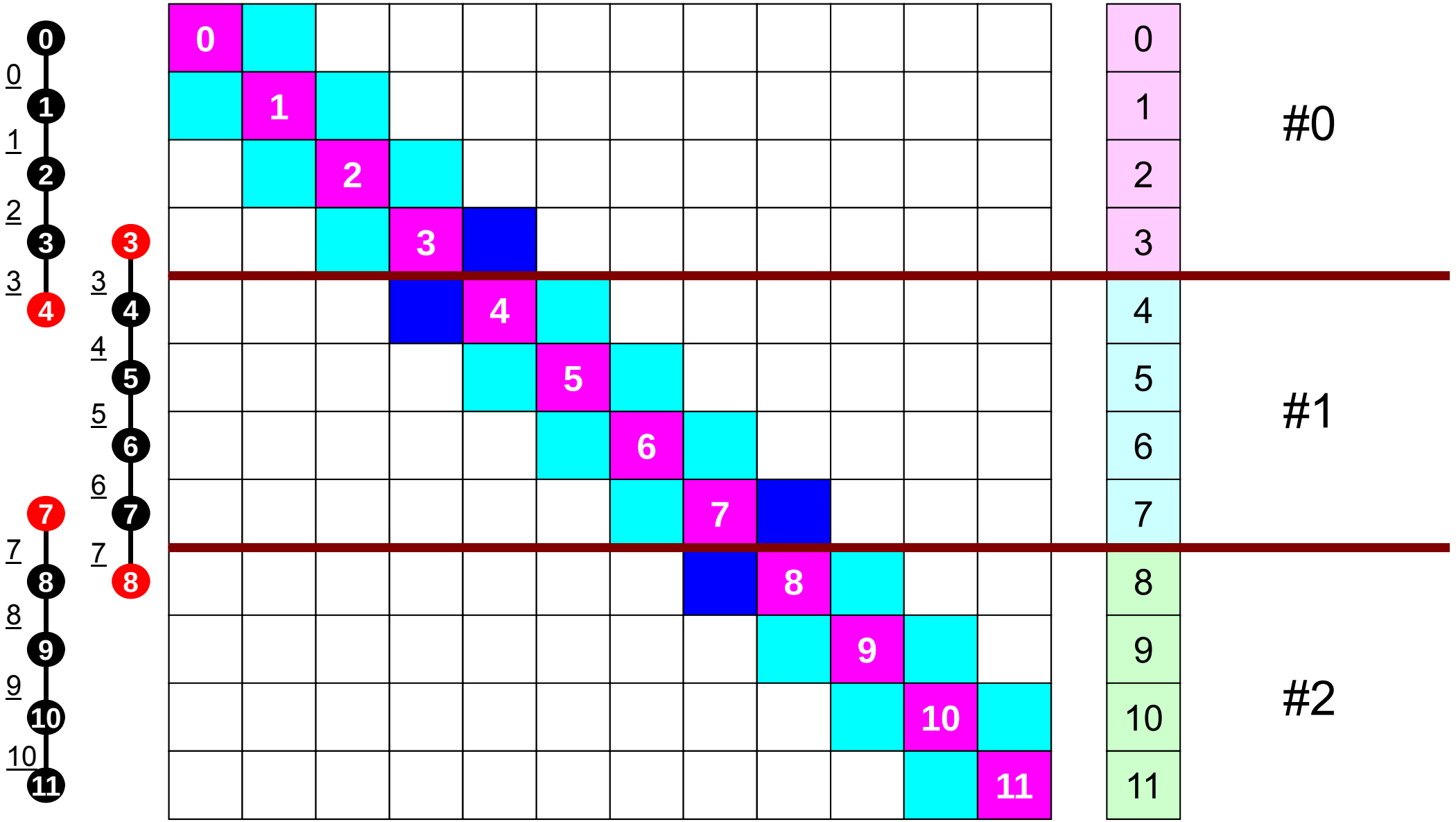
内点だけで分割するとマトリクス不完全



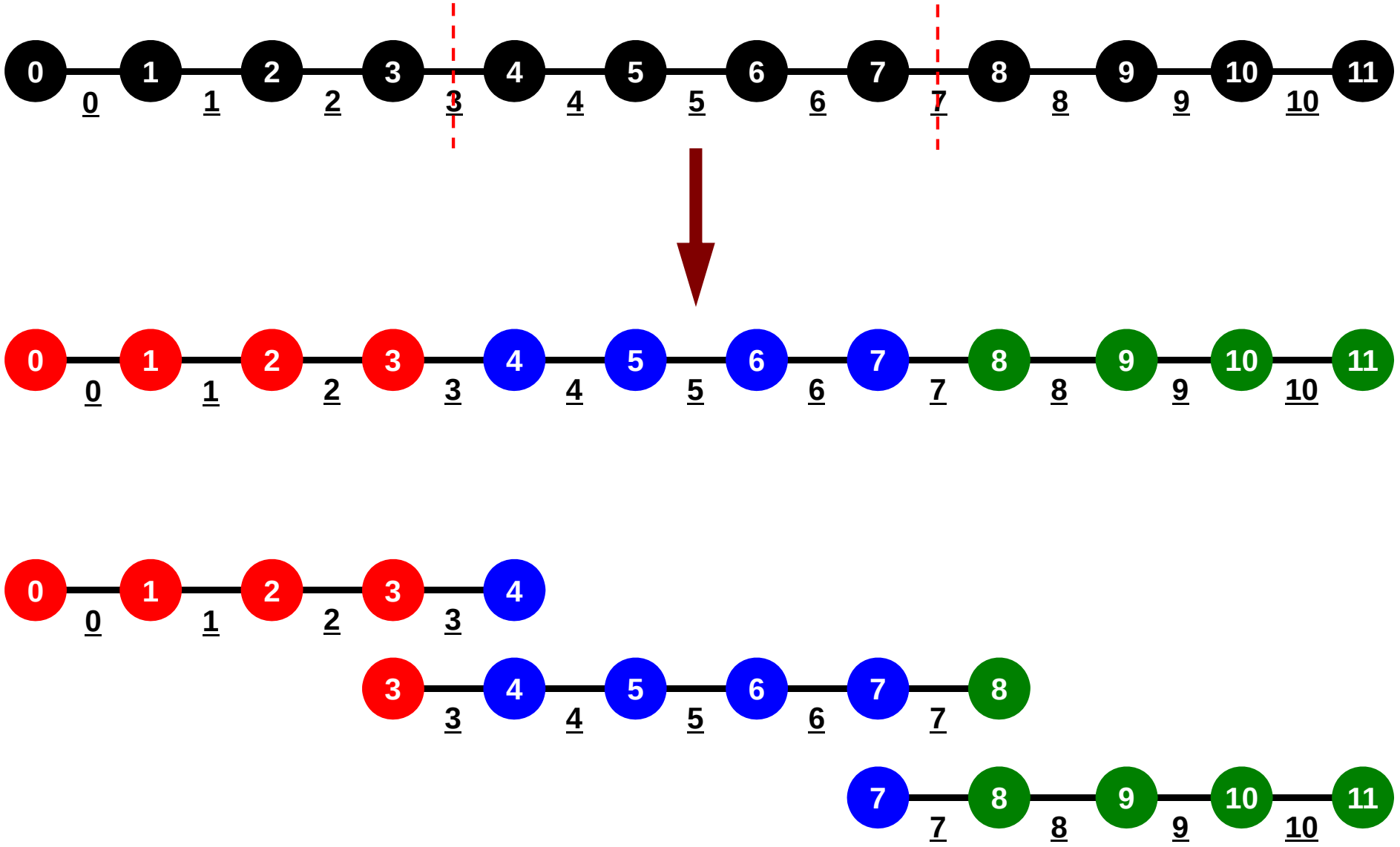
要素＋外点



一次元問題：11要素，12節点，3領域



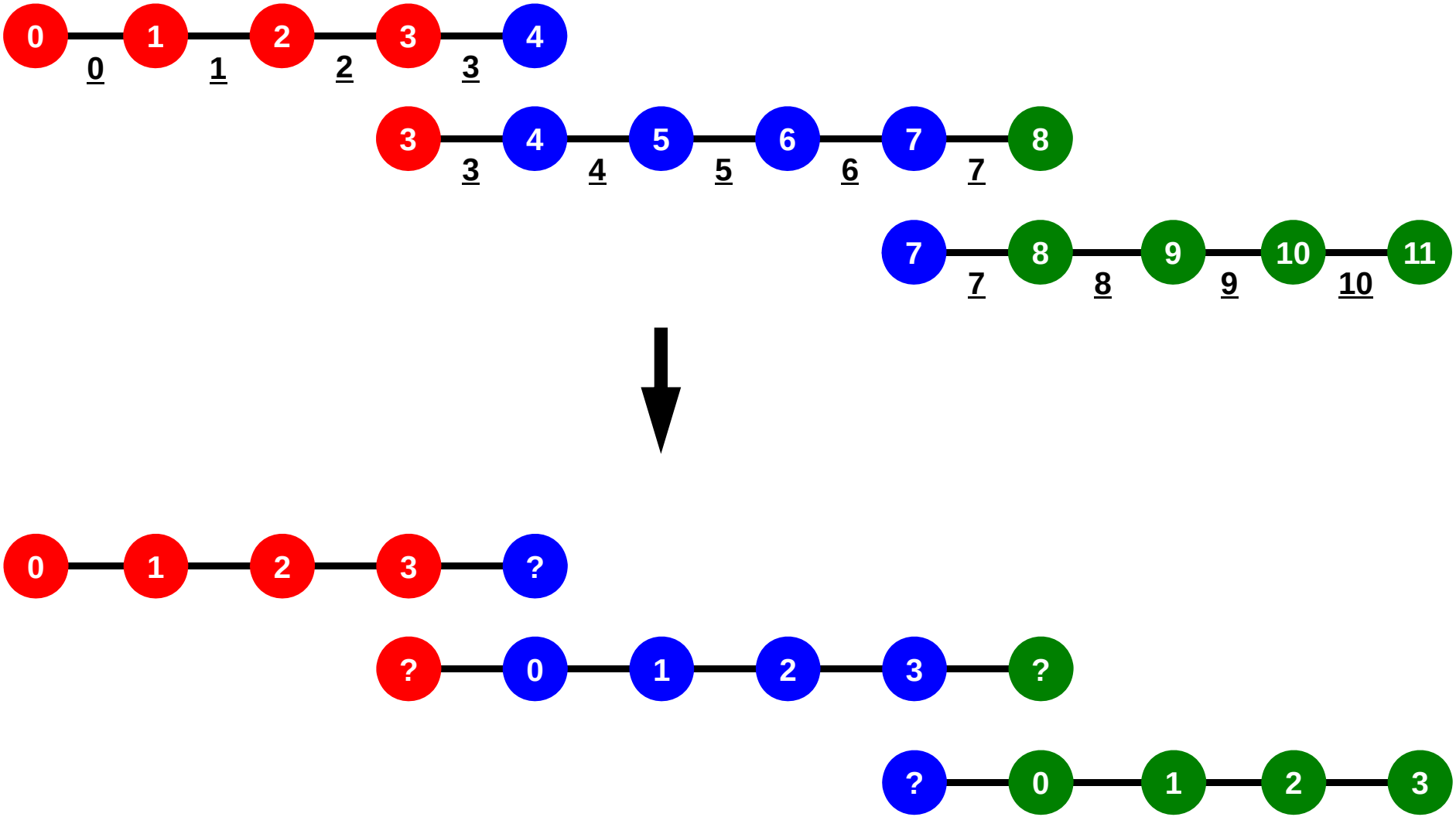
一次元問題:11要素，12節点，3領域



SPMD向け局所番号付け

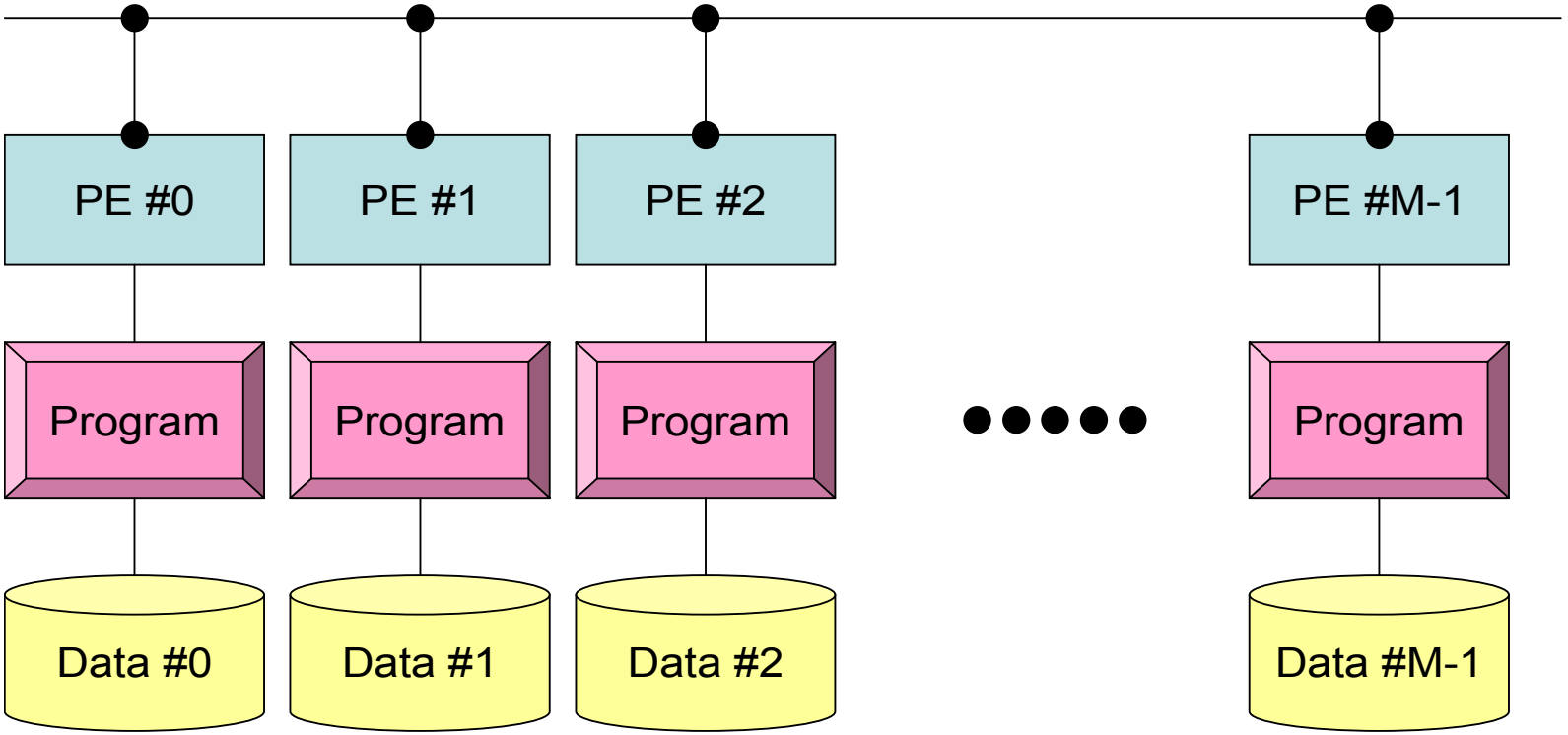
内点が1~N（0~N）となっていれば，もとのプログラムと同じ

外点の番号は?



MPIによる並列化：SPMD

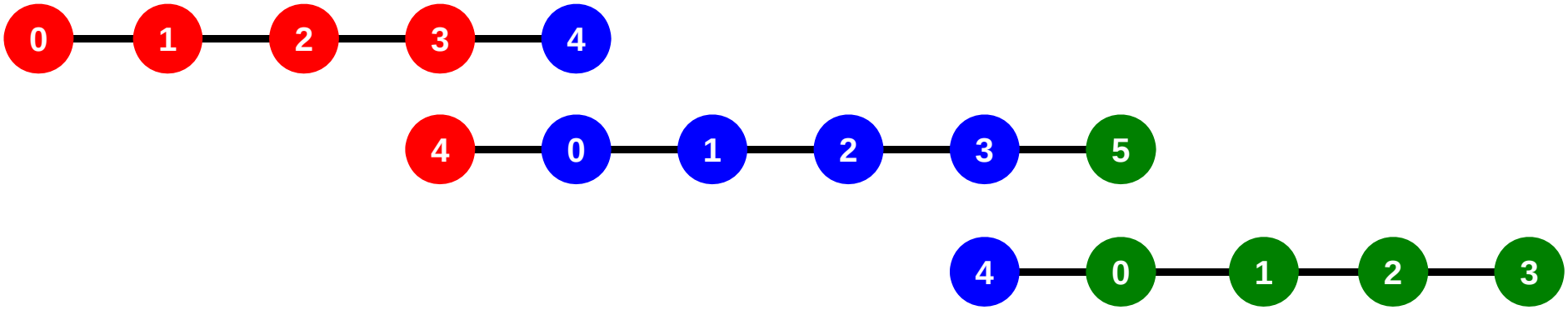
- Single Program/Instruction Multiple Data
- 基本的に各プロセスは「同じことをやる」が「データが違う」
  - 大規模なデータを分割し，各部分について各プロセス（プロセッサ）が計算する
- 全体データと局所データ，全体番号と局所番号
- 通信以外は単体CPUと同じ，というのが理想



SPMD向け局所番号付け

内点が1~N（0~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（N,N+1）



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

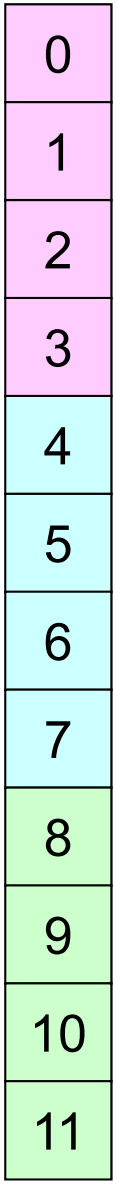
- 前処理
  - 対角スケーリング
- 並列処理が必要なプロセス
  - 内積
  - 行列ベクトル積

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

```
/*
//-- {z}= [Minv]{r}
*/
    for(i=0;i<N;i++){
        W[Z][i] = W[DD][i] * W[R][i];
    }
```

```
/*
//-- {x}= {x} + ALPHA*{p}
//   {r}= {r} - ALPHA*{q}
*/
  for(i=0;i<N;i++){
      U[i]    += Alpha * W[P][i];
      W[R][i] -= Alpha * W[Q][i];
  }
```

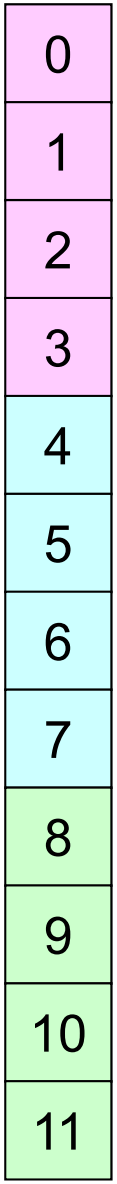


内積

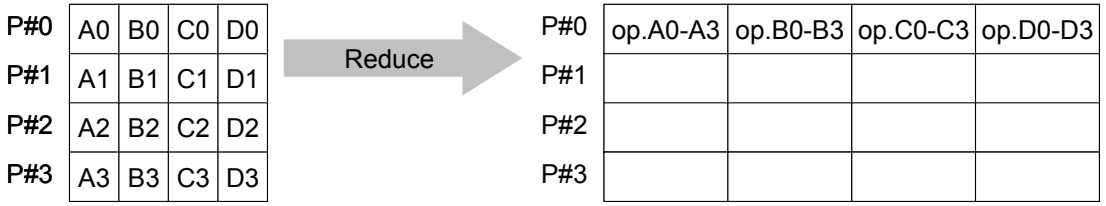
全体で和をとる必要がある⇒通信?

```
/*
//-- ALPHA= RHO / {p}{q}
*/
  C1 = 0.0;
  for(i=0;i<N;i++){
      C1 += W[P][i] * W[Q][i];
  }

  Alpha = Rho / C1;
```



MPI_Reduce



- Reduces values on all processes to a single value
  - Summation, Product, Max, Min etc.

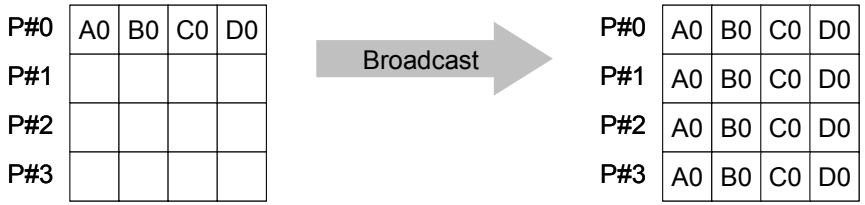
- **MPI_Reduce (sendbuf,recvbuf,count,datatype,op,root,comm)**
  - **sendbuf** choice I starting address of send buffer
  - **recvbuf** choice O starting address receive buffer
    type is defined by "**datatype**"
  - **count** int I number of elements in send/receive buffer
  - **datatype** MPI_Datatype I data type of elements of send/recive buffer
    FORTRAN MPI_INTEGER, MPI_REAL, MPI_DOUBLE_PRECISION, MPI_CHARACTER etc.
    C MPI_INT, MPI_FLOAT, MPI_DOUBLE, MPI_CHAR etc

  - **op** MPI_Op I reduce operation
    MPI_MAX, MPI_MIN, MPI_SUM, MPI_PROD, MPI_LAND, MPI_BAND etc
    Users can define operations by **MPI_OP_CREATE**

  - **root** int I rank of root process
  - **comm** MPI_Comm I communicator

C

MPI_Bcast



- Broadcasts a message from the process with rank "root" to all other processes of the communicator

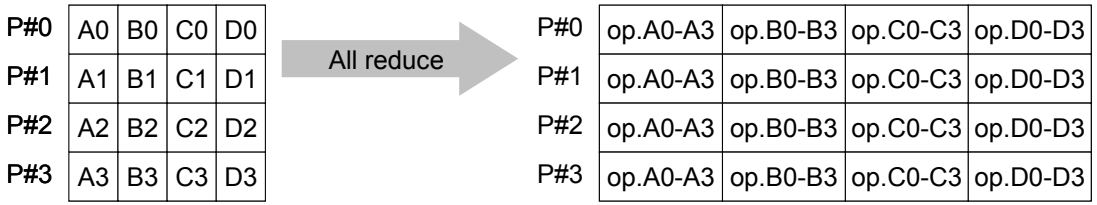
- **`MPI_Bcast (buffer,count,datatype,root,comm)`**
  - **`buffer`** choice I/O starting address of buffer
    type is defined by "**`datatype`**"
  - **`count`** int I number of elements in send/receive buffer
  - **`datatype`** MPI_Datatype I data type of elements of send/recive buffer

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

  - **`root`** int I rank of root process
  - **`comm`** MPI_Comm I communicator

C

MPI_Allreduce



- MPI_Reduce + MPI_Bcast
- Summation (of dot products) and MAX/MIN values are likely to utilized in each process

- **`call MPI_Allreduce (sendbuf,recvbuf,count,datatype,op, comm)`**
  - **`sendbuf`** choice I starting address of send buffer
  - **`recvbuf`** choice O starting address receive buffer
    type is defined by "**`datatype`**"
  - **`count`** int I number of elements in send/receive buffer
  - **`datatype`** MPI_Datatype I data type of elements of send/recive buffer
  - **`op`** MPI_Op I reduce operation
  - **`comm`** MPI_Comm I communicator

C

C

“op” of MPI_Reduce/Allreduce

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

- **MPI_MAX**, **MPI_MIN** Max, Min
- **MPI_SUM**, **MPI_PROD** Summation, Product
- **MPI_LAND** Logical AND

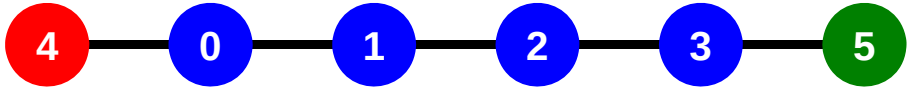
前処理付き共役勾配法
Preconditioned Conjugate Gradient Method（CG）

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

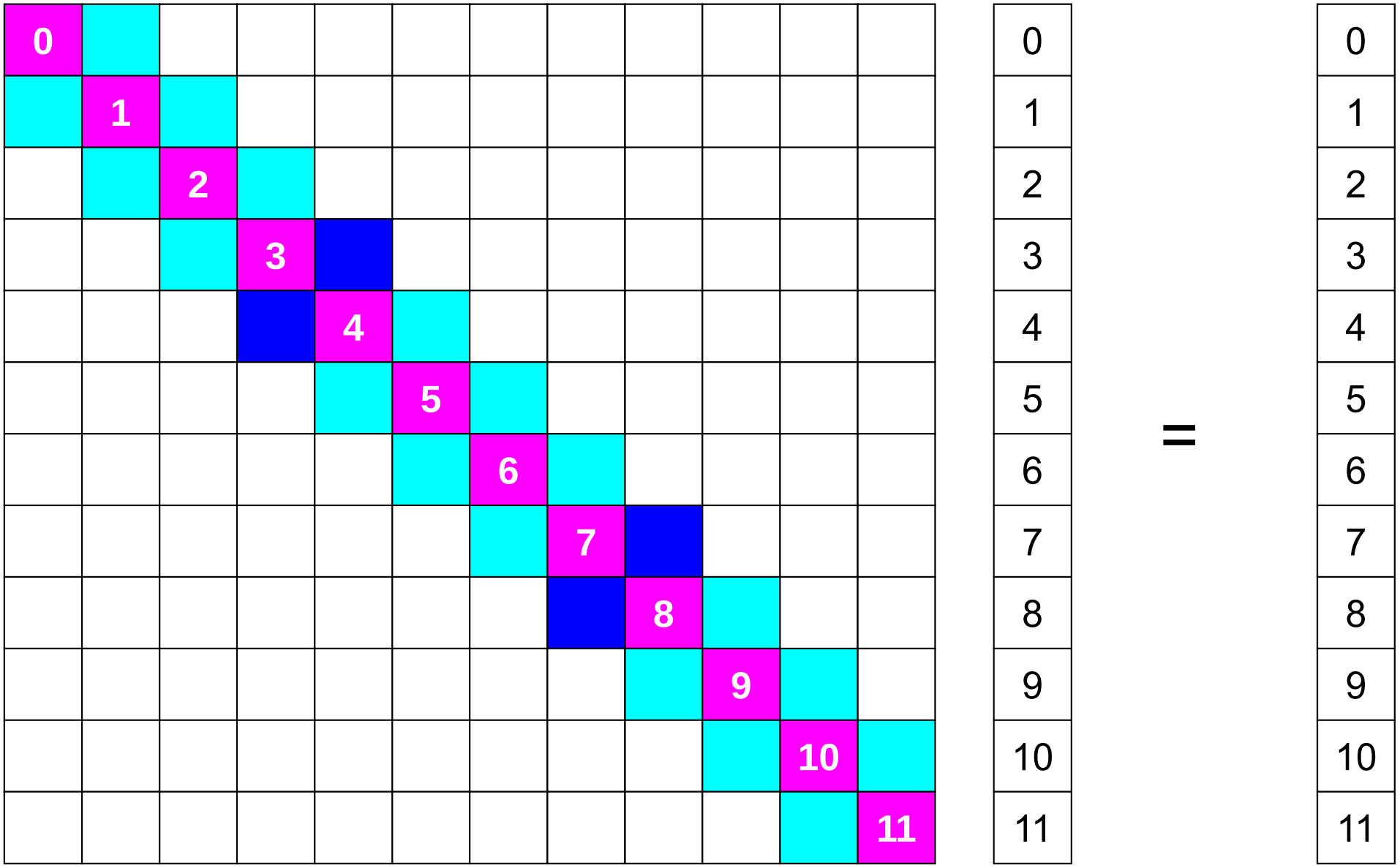
- 前処理
  - 対角スケーリング
- 並列処理が必要なプロセス
  - 内積
  - 行列ベクトル積

行列ベクトル積
外点の値が必要⇒通信?

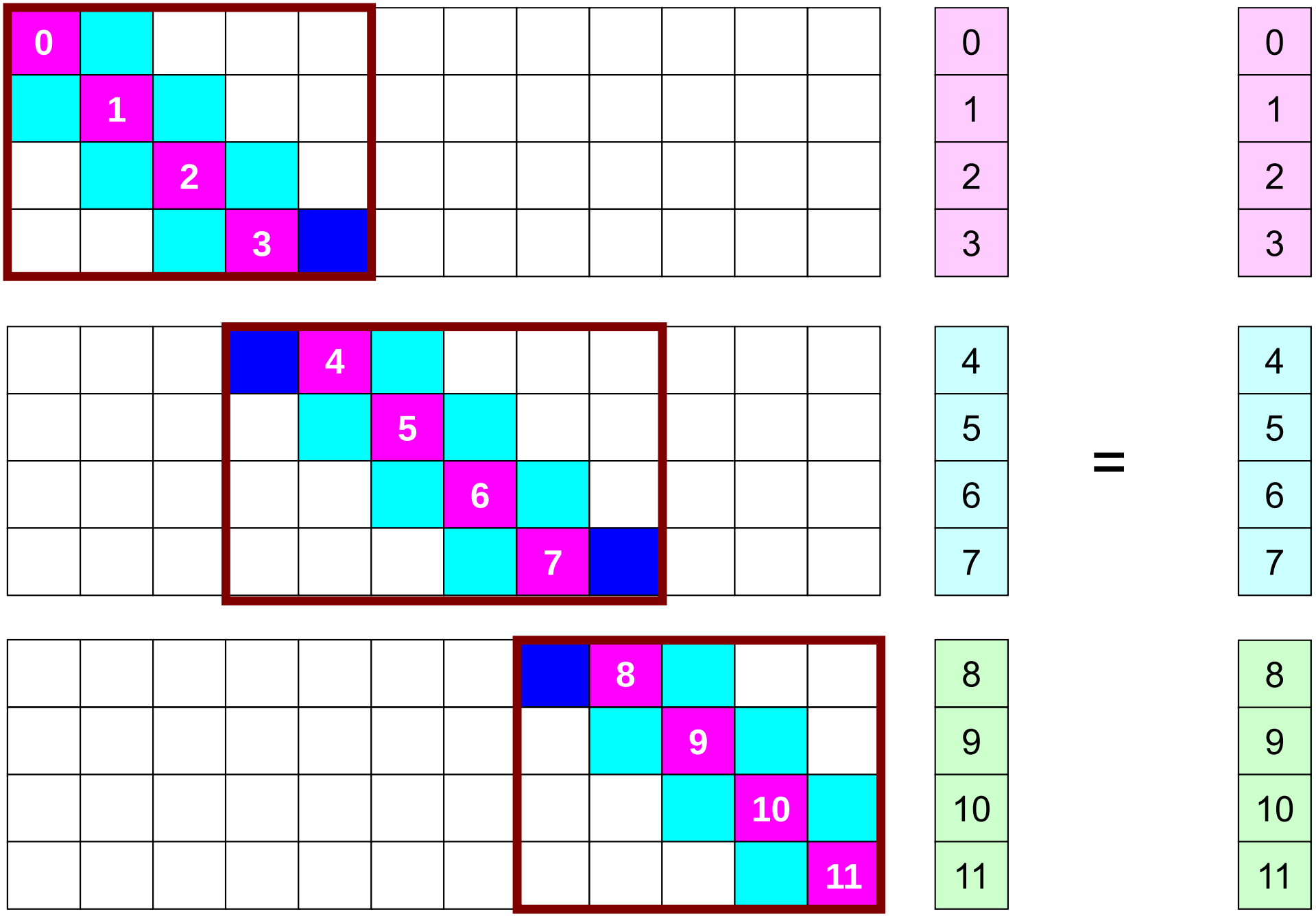
```
/*
//-- {q}= [A]{p}
*/
  for(i=0;i<N;i++){
      W[Q][i] = Diag[i] * W[P][i];
      for(j=Index[i];j<Index[i+1];j++){
          W[Q][i] += AMat[j]*W[P][Item[j]];
          }
      }
```



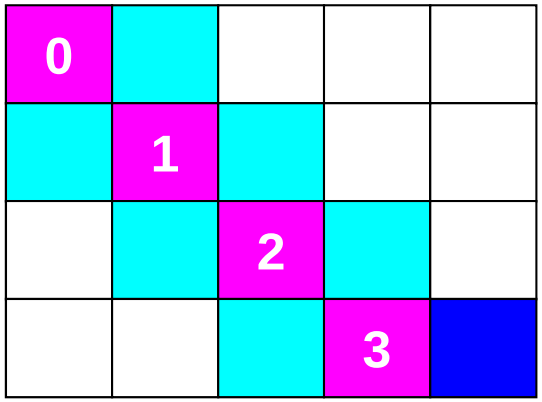
行列ベクトル積：ローカルに計算実施可能



行列ベクトル積：ローカルに計算実施可能

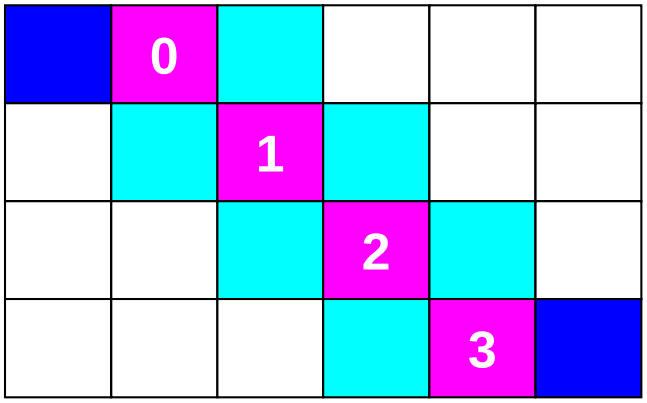


行列ベクトル積：ローカルに計算実施可能



0
1
2
3

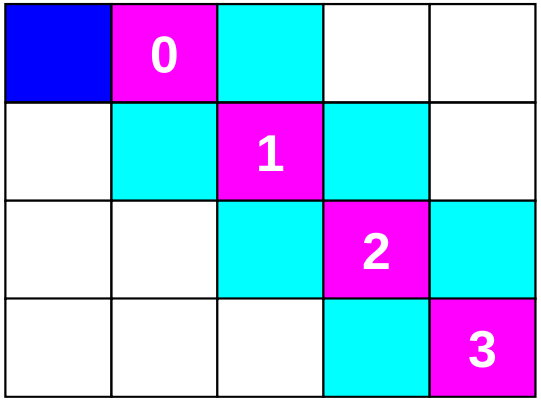
0
1
2
3



0
1
2
3

=

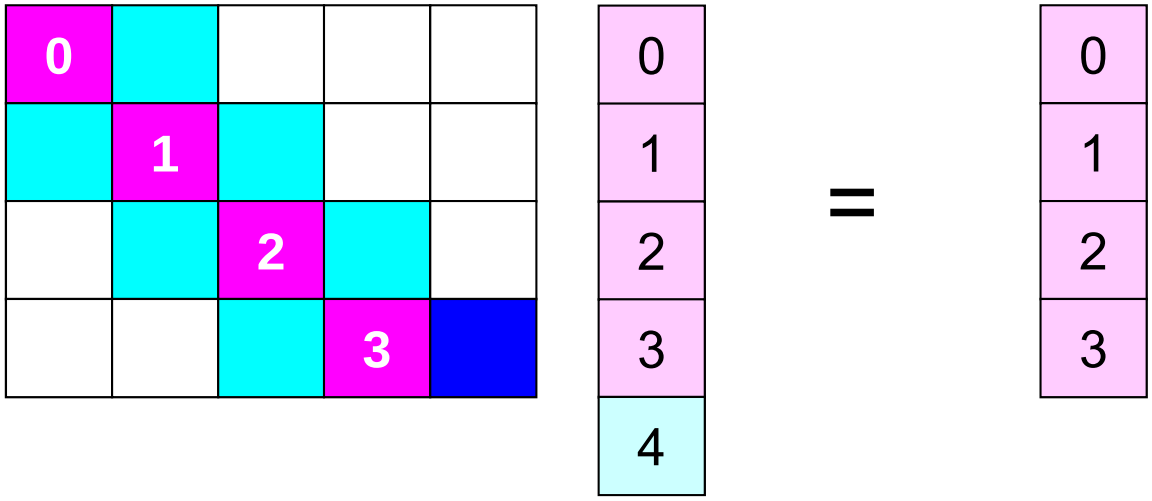
0
1
2
3

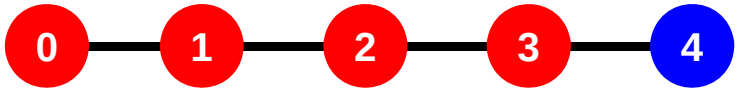


0
1
2
3

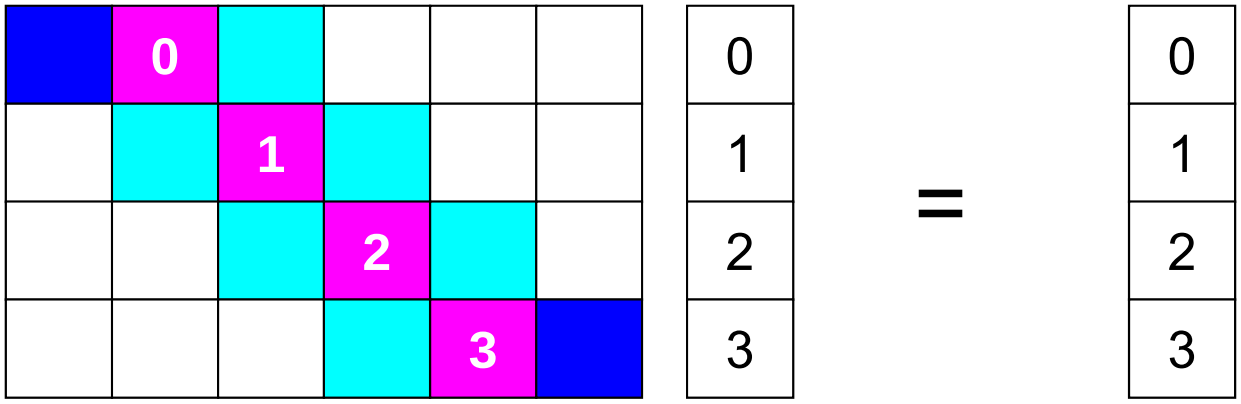
0
1
2
3

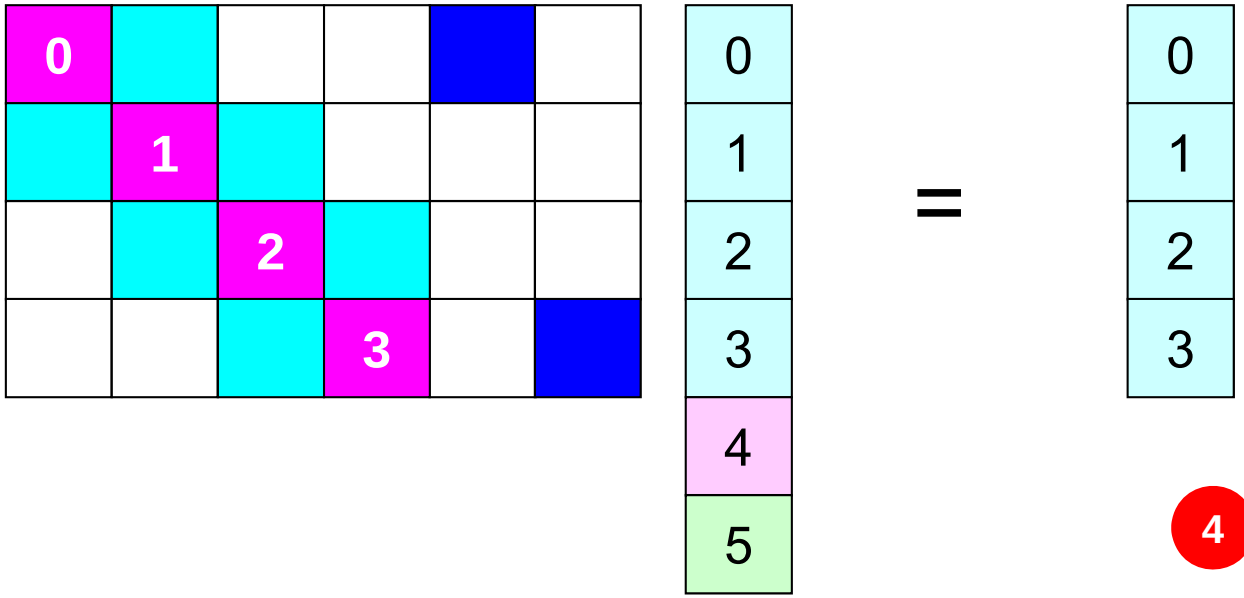
行列ベクトル積：ローカル計算 #0

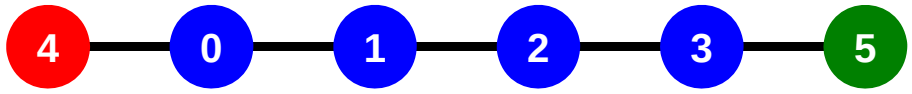




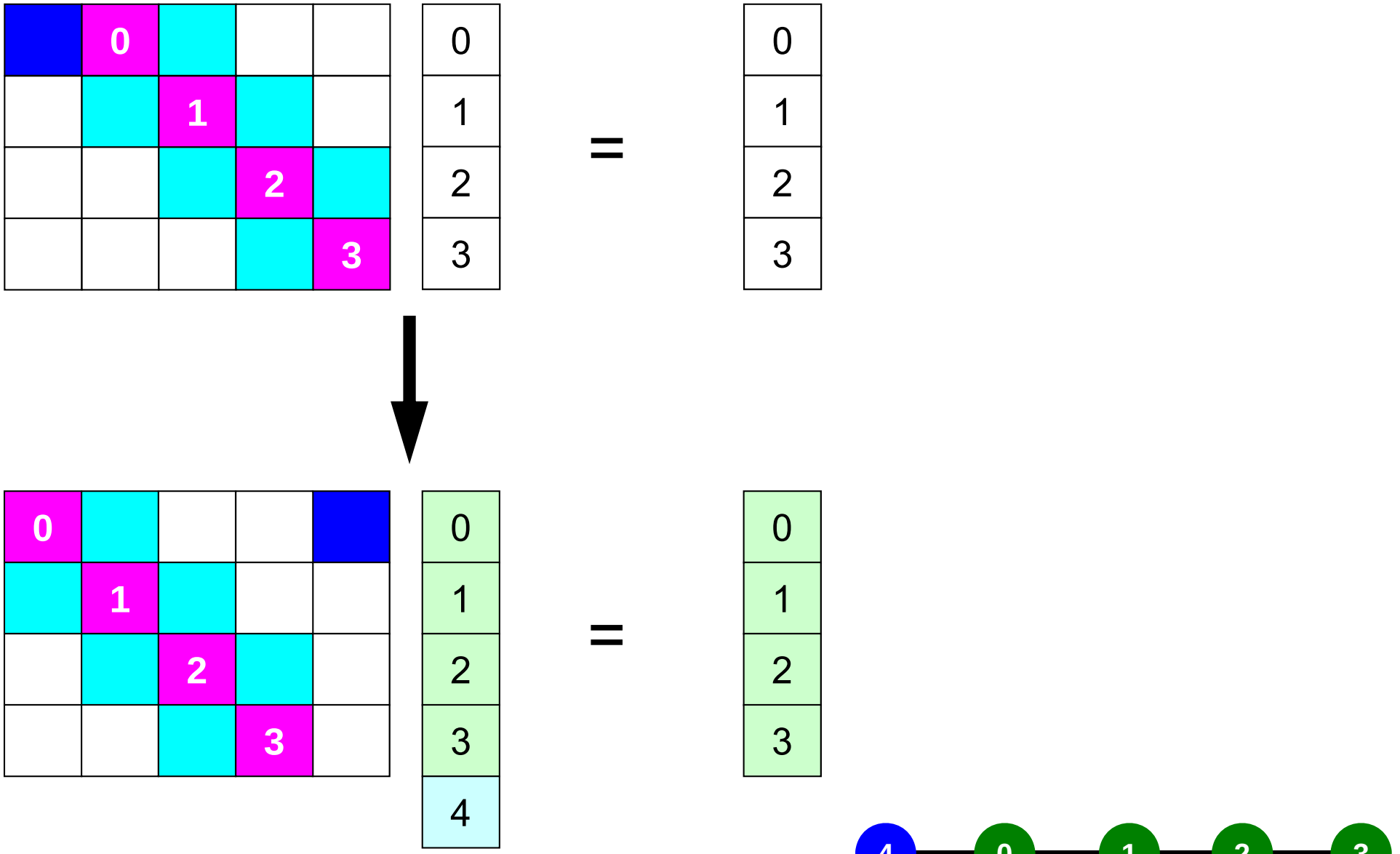
行列ベクトル積：ローカル計算 #1



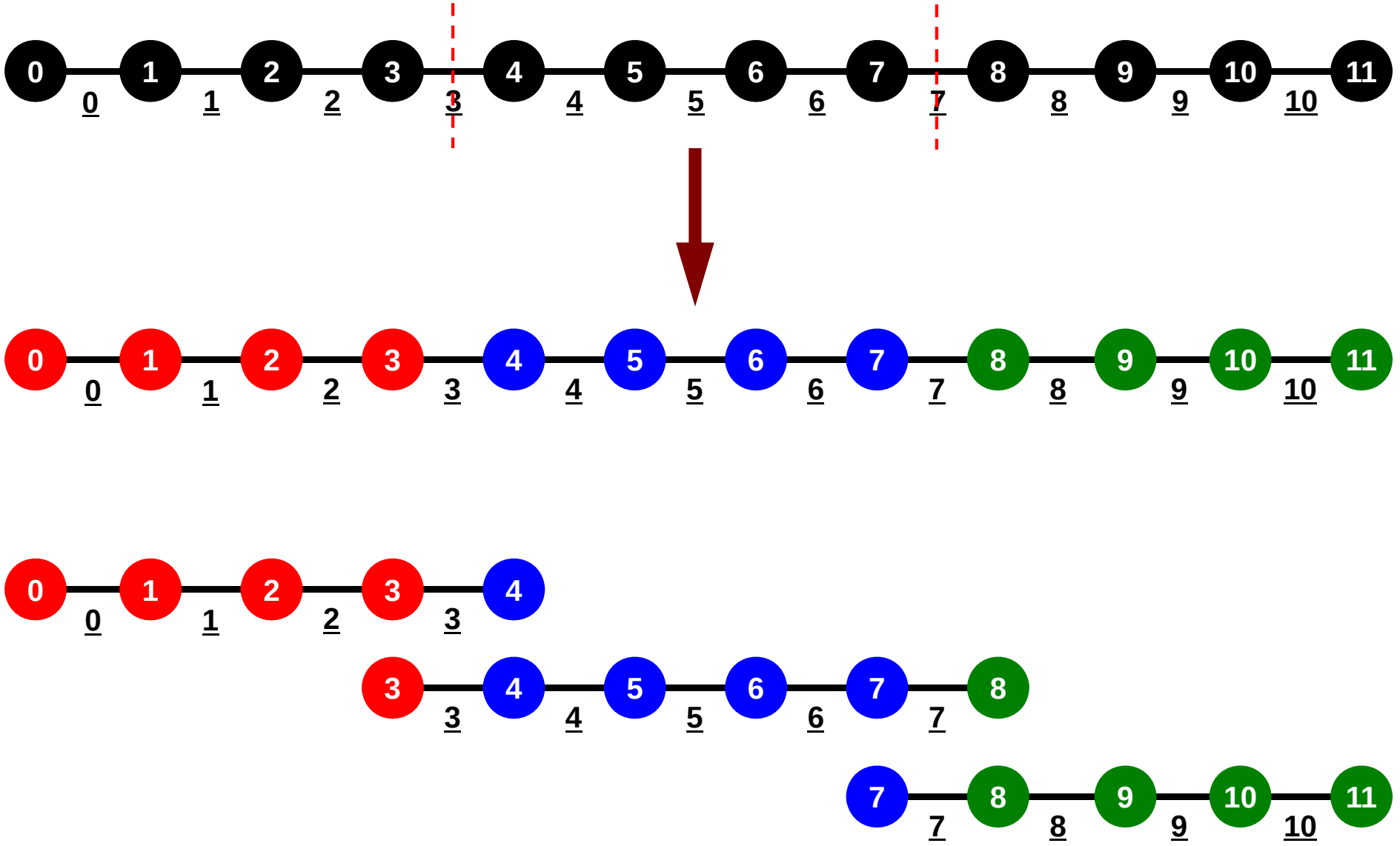




行列ベクトル積：ローカル計算 #2

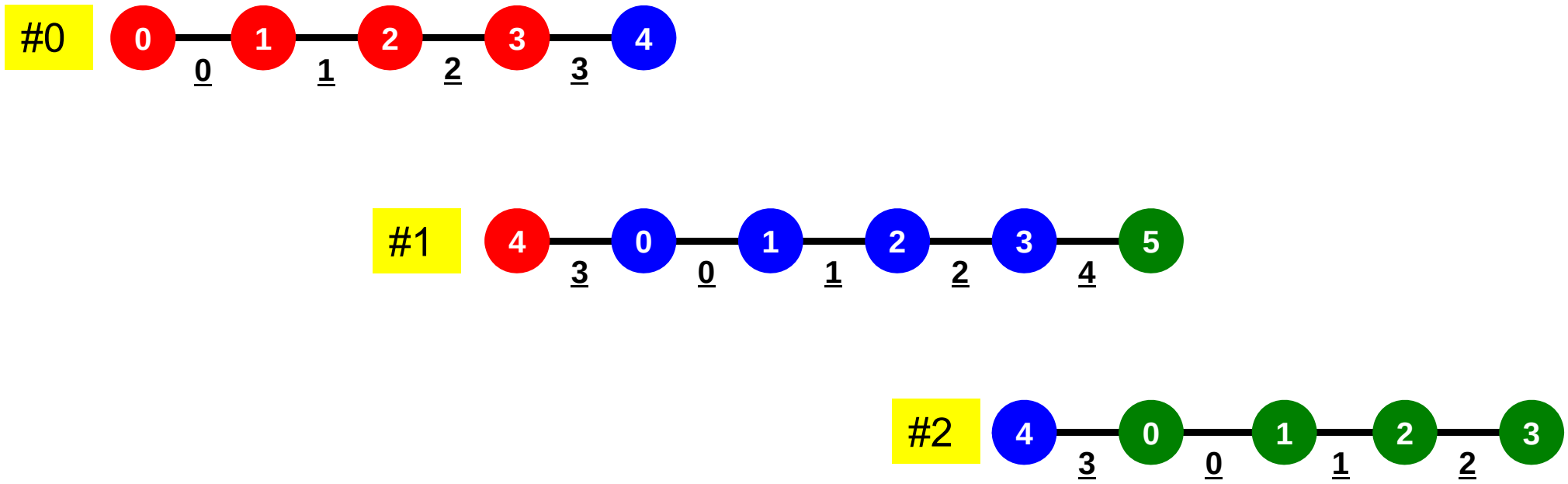


1D FEM: 12 nodes/11 elem's/3 domains



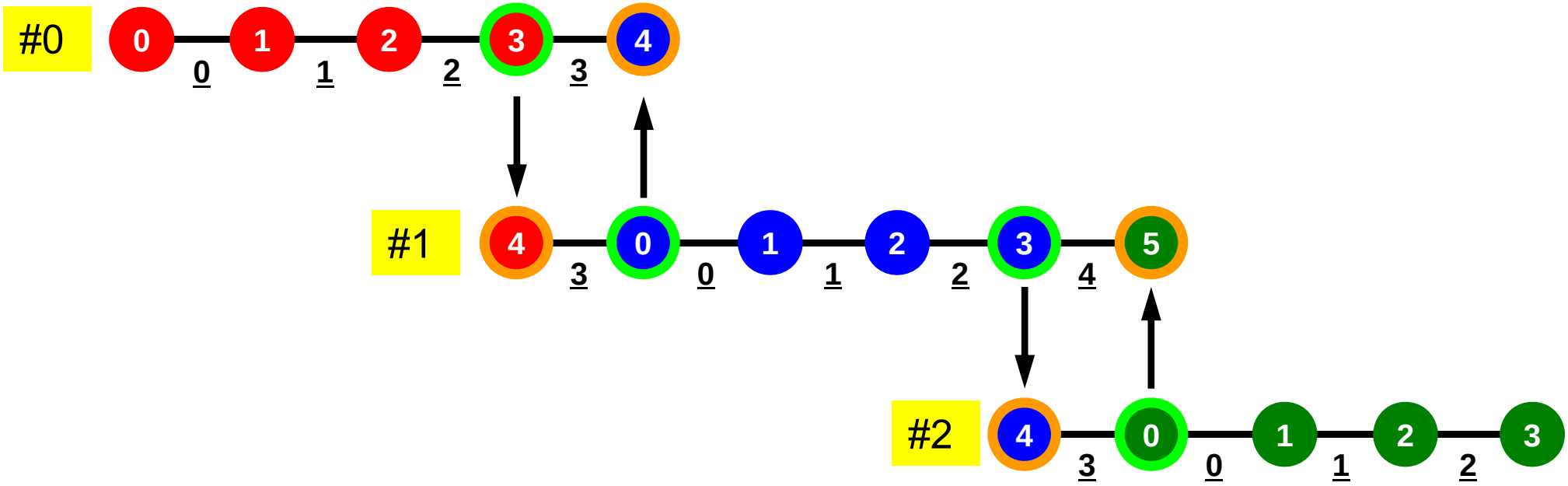
1D FEM: 12 nodes/11 elem's/3 domains

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



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

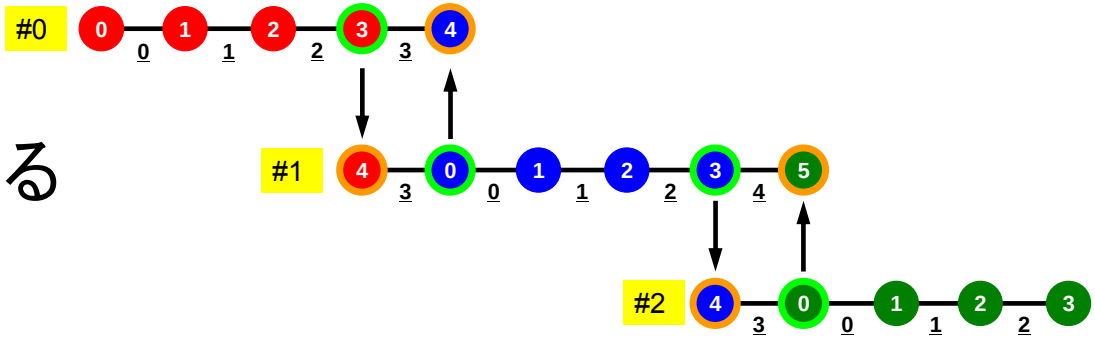
Internal/External Nodes



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

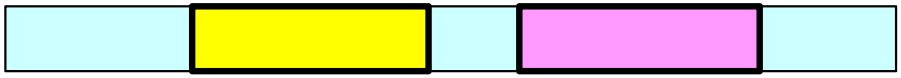
- 1対1通信：Point-to-Point
  - MPI_Send, MPI_Receive
  - 特定のプロセスとのみ通信がある
    - 隣接領域
  - 適用分野
    - 差分法，有限要素法などローカルな情報を使う手法

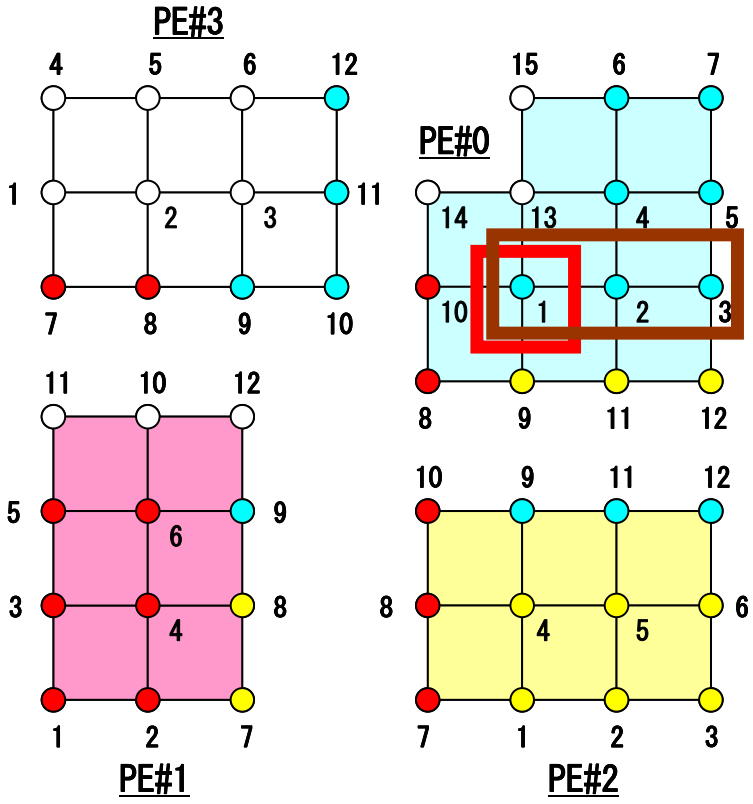


SEND（送信）：境界点の送信

送信バッファの連続したデータを隣接プロセスに送る

- **MPI_Isend**
  (sendbuf,count,datatype,dest,tag,comm,request)
  - **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）





C

MPI_Isend

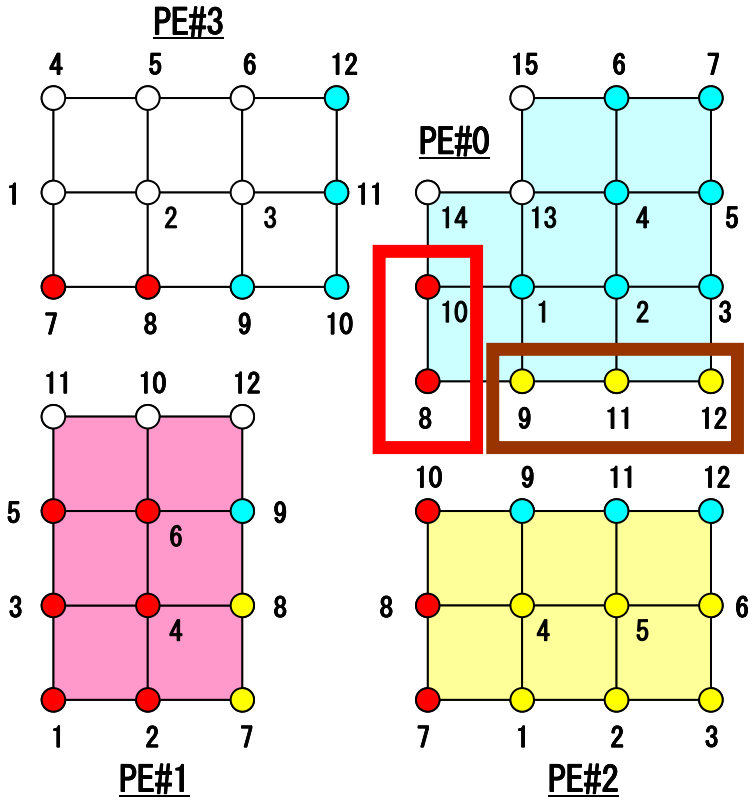
- 送信バッファ「sendbuf」内の，連続した「count」個の送信メッセージを，タグ「tag」を付けて，コミュニケータ内の，「dest」に送信する。「MPI_Waitall」を呼ぶまで，送信バッファの内容を更新してはならない。

- **MPI_Isend (sendbuf,count,datatype,dest,tag,comm,request)**
  - **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** MPI_Comm I コミュニケータを指定する
  - **request** MPI_Request 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

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

- `MPI_Irecv (recvbuf,count,datatype,dest,tag,comm,request)`
  - **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)



C

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。

- **MPI_Irecv (recvbuf,count,datatype,dest,tag,comm,request)**
  - **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** MPI_Comm I コミュニケータを指定する
  - **request** MPI_Request 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

C

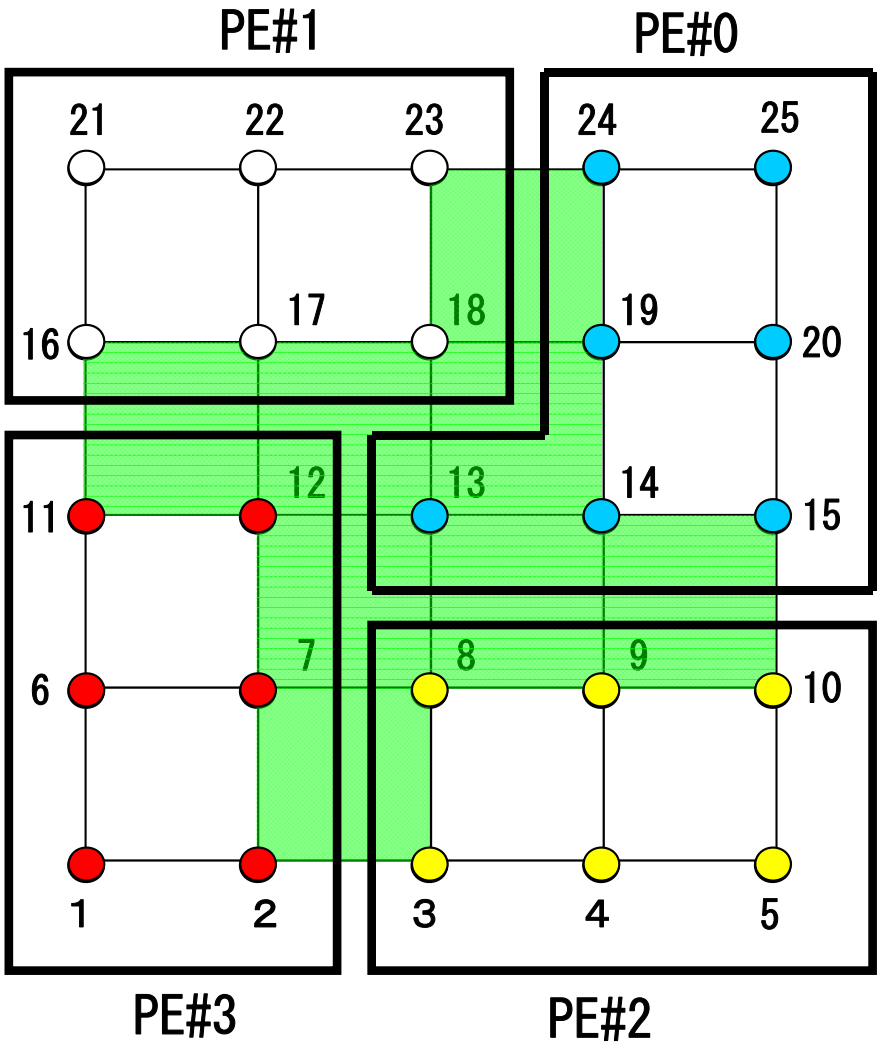
MPI_Waitall

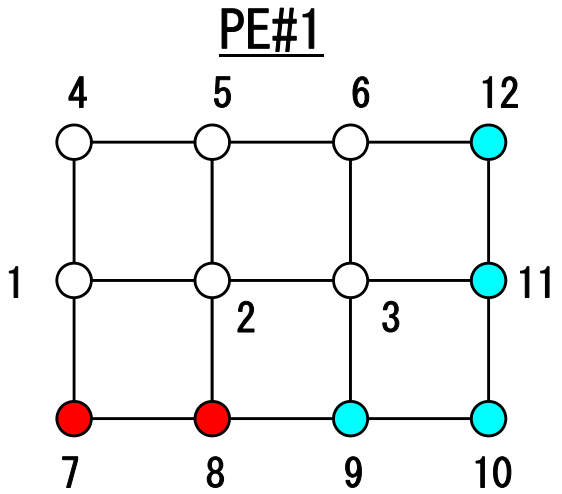
- 1対1非ブロッキング通信関数である「MPI_Isend」と「MPI_Irecv」を使用した場合，プロセスの同期を取るのに使用する。
- 送信時はこの「MPI_Waitall」を呼ぶ前に送信バッファの内容を変更してはならない。受信時は「MPI_Waitall」を呼ぶ前に受信バッファの内容を利用してはならない。
- 整合性が取れていれば，「MPI_Isend」と「MPI_Irecv」を同時に同期してもよい。
  - 「MPI_Isend/Irecv」で同じ通信識別子を使用すること
- 「MPI_Barrier」と同じような機能であるが，代用はできない。
  - 実装にもよるが，「request」，「status」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。

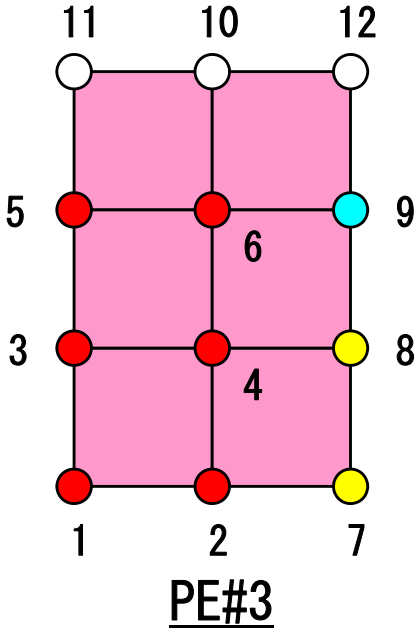
- **MPI_Waitall (count,request,status)**
  - **count** 整数 I 同期する必要のある「MPI_ISEND」，「MPI_RECV」呼び出し数。
  - **request** 整数 I/O 通信識別子。「MPI_ISEND」，「MPI_IRECV」で利用した識別子名に対応。（配列サイズ：（count））
  - **status** MPI_Status O 状況オブジェクト配列
    MPI_STATUS_SIZE：“mpif.h”，”mpi.h”で定められる
    パラメータ：C言語については後述

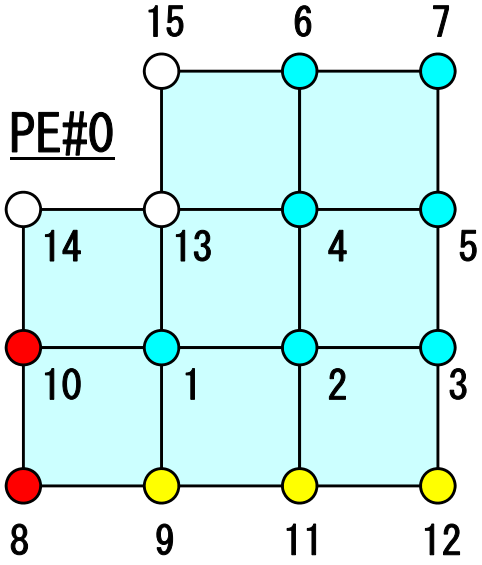
Node-based Partitioning

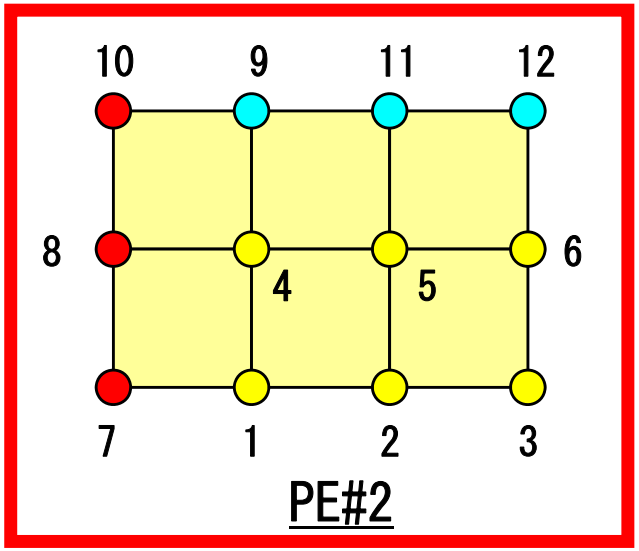
internal nodes - elements - external nodes



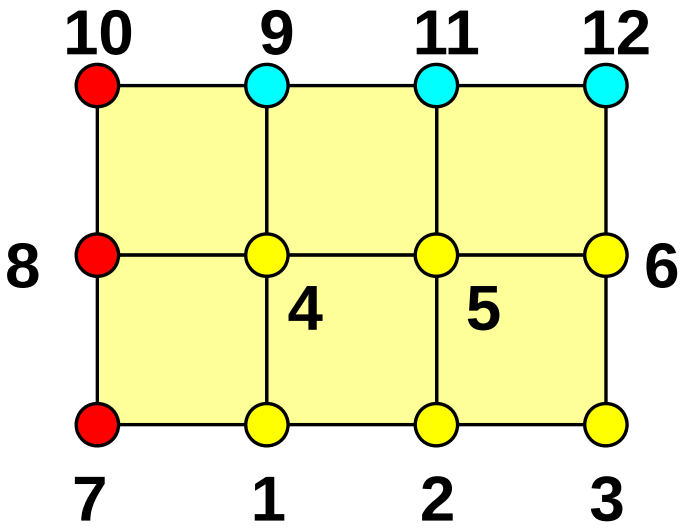








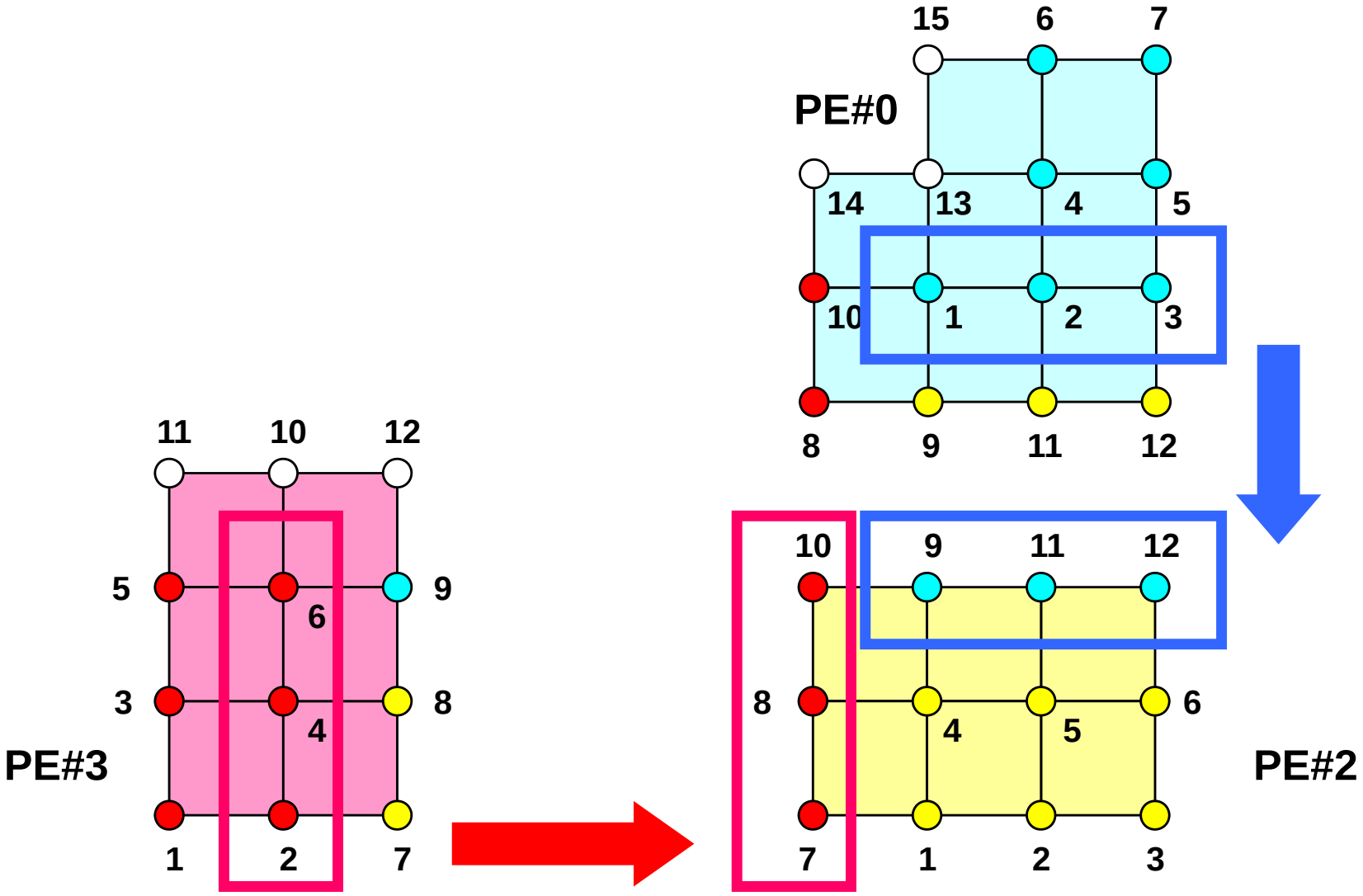
各領域データ(局所データ)仕様



- 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

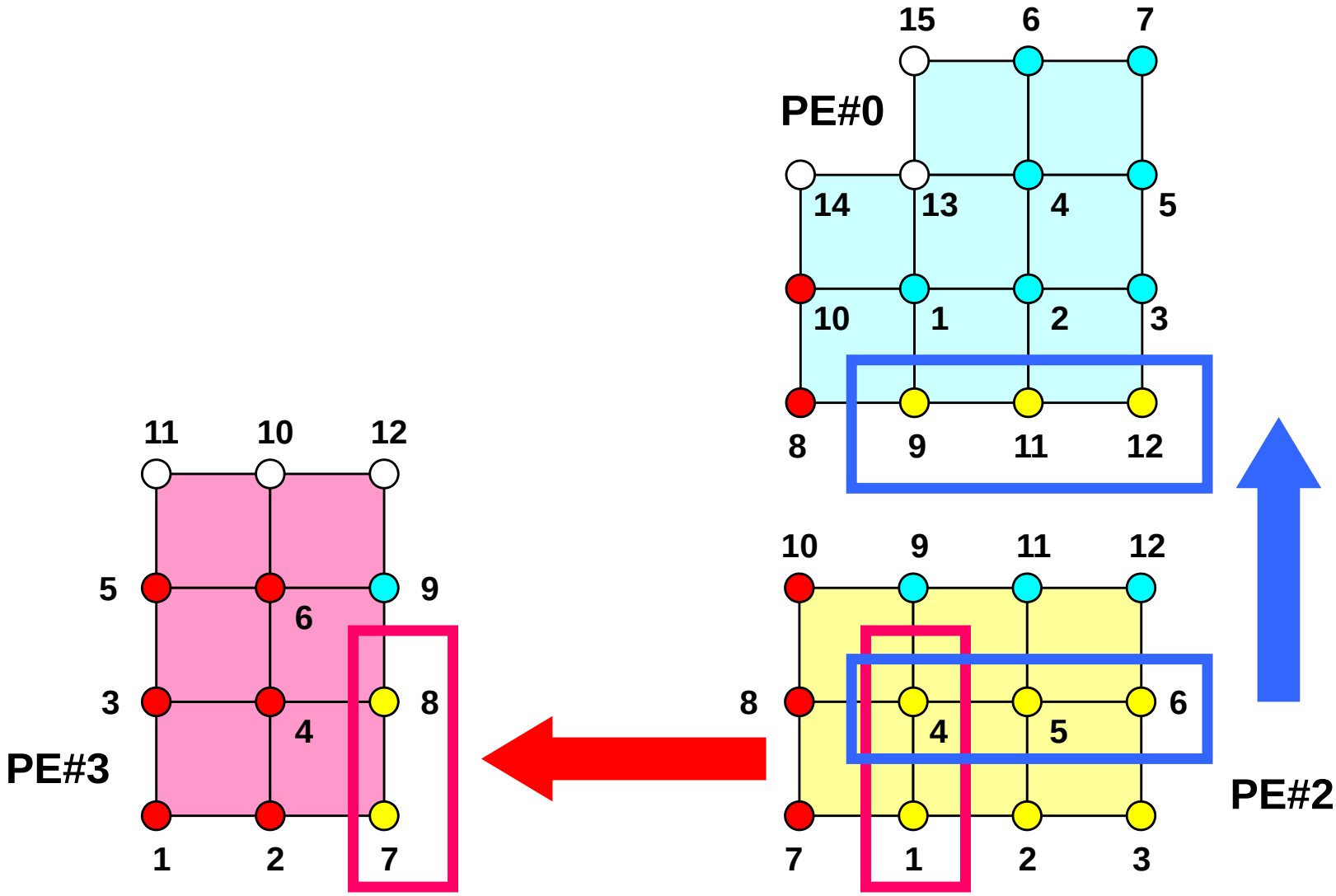
External Nodes（外点）：RECEIVE

PE#2 : receive information for “external nodes”



Boundary Nodes（内点）：SEND

PE#2 : send information on “boundary nodes”



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