並列有限要素法への道
―並列データ構造―
Fortran編

中島 研吾

東京大学情報基盤センター

- 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 Communication（1対1通信）

並列計算の目的

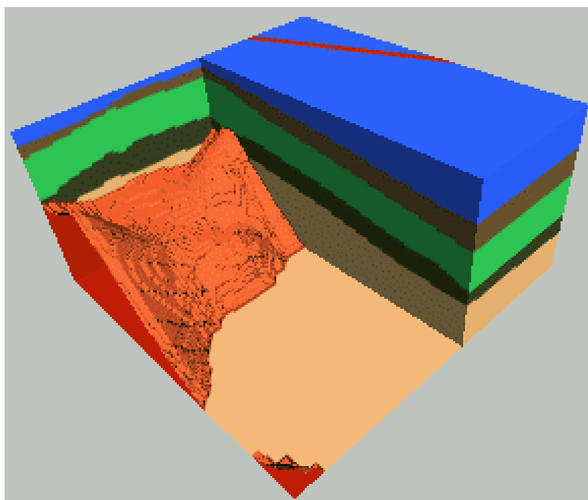
- 高速，大規模
  - 「大規模」の方が「新しい科学」という観点からのウェイトとしては高い。しかし，「高速」ももちろん重要である。
    - 細かいメッシュ
- +複雑
- 理想：Scalable
  - N倍の規模の計算をN倍のCPUを使って，「同じ時間で」解く（大規模性の追求：Weak Scaling）
    - 実際はそうは行かない
      - 例：共役勾配法⇒問題規模が大きくなると反復回数が増加
  - 同じ問題をN倍のCPUを使って「1/Nの時間で」解く・・・という場合もある（高速性の追求：Strong Scaling）
    - これも余り簡単な話では無い

並列計算とは？(1/2)

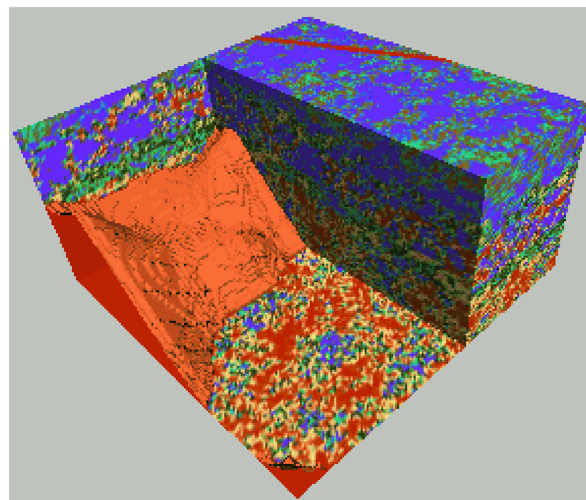
- より大規模で複雑な問題を高速に解きたい

Homogeneous/Heterogeneous Porous Media

Lawrence Livermore National Laboratory



Homogeneous

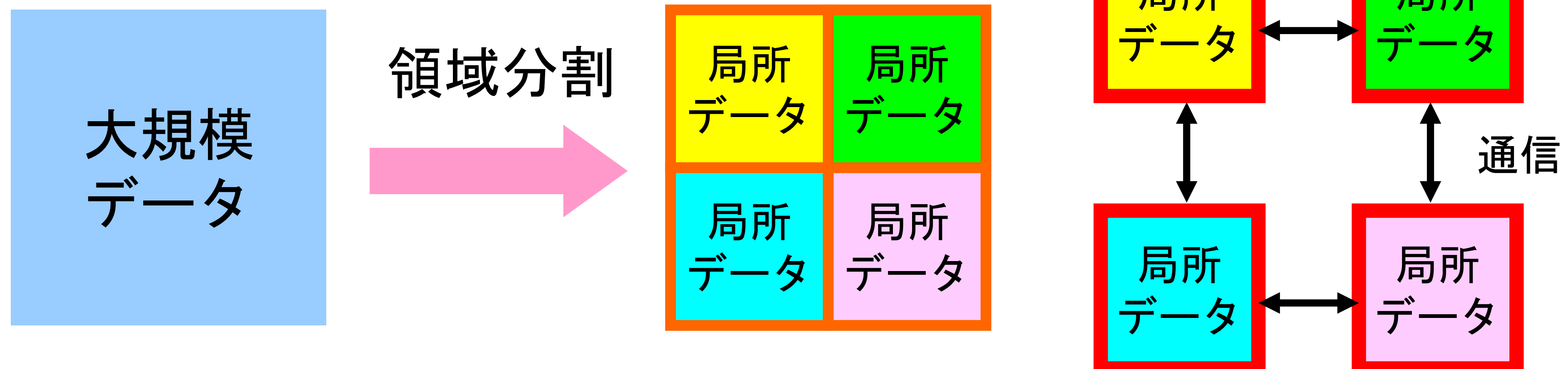


Heterogeneous

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

データ構造とアルゴリズム

- コンピュータ上で計算を行うプログラムはデータ構造とアルゴリズムから構成される。
- 両者は非常に密接な関係にあり，あるアルゴリズムを実現するためには，それに適したデータ構造が必要である。
  - 極論を言えば「データ構造＝アルゴリズム」と言っても良い。
  - もちろん「そうではない」と主張する人もいるが，科学技術計算に関する限り，中島の経験では「データ構造＝アルゴリズム」と言える。
- 並列計算を始めるにあたって，基本的なアルゴリズムに適したデータ構造を定める必要がある。

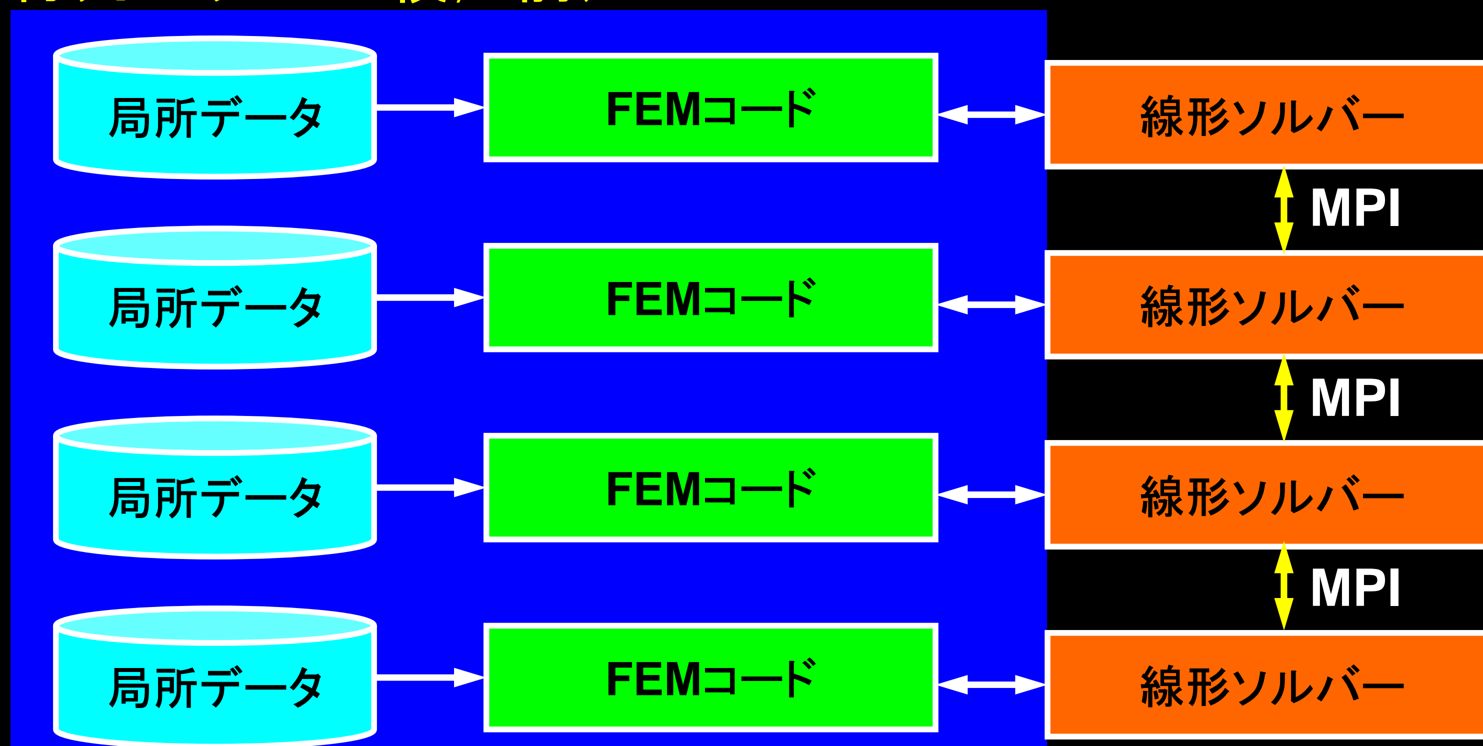
並列有限要素法の処理：SPMD

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

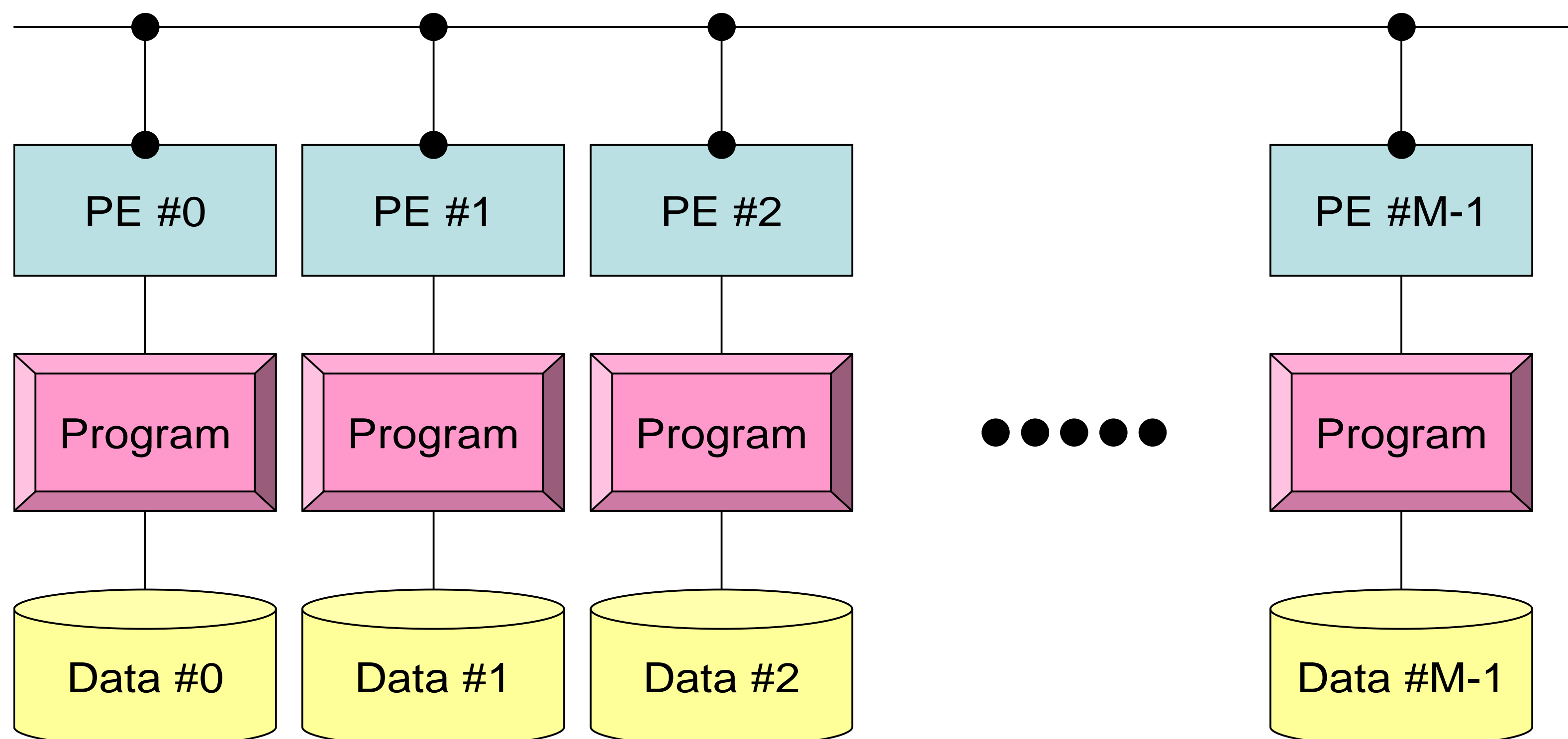
グローバル処理，通信は線形ソルバーのみで生じる

内積，行列ベクトル積，前処理



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

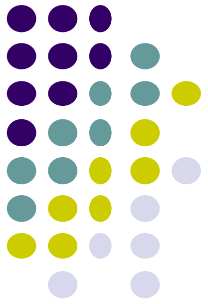


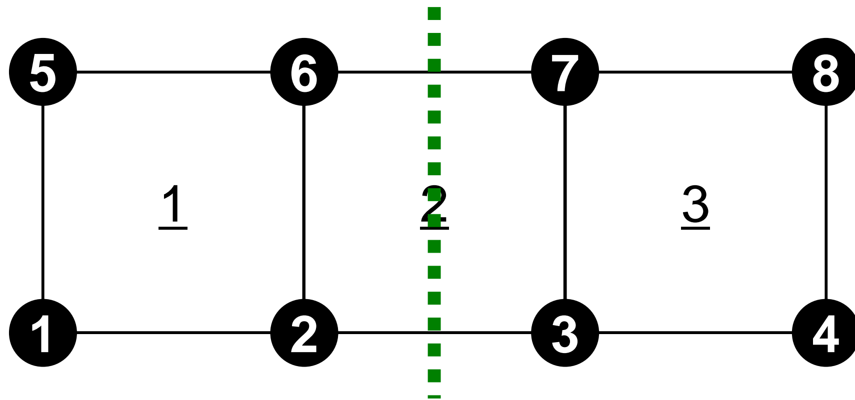
並列有限要素法プログラムの開発

- 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

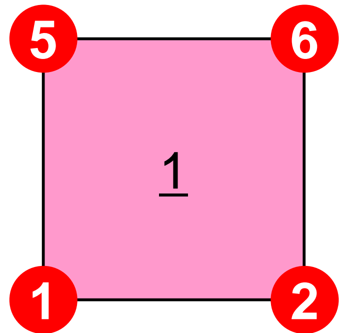
四角形要素

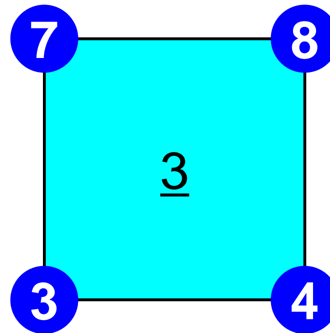




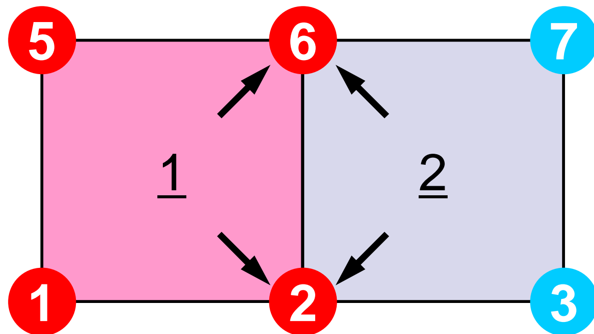
「節点ベース(領域ごとの節点数がバランスする)」の分割

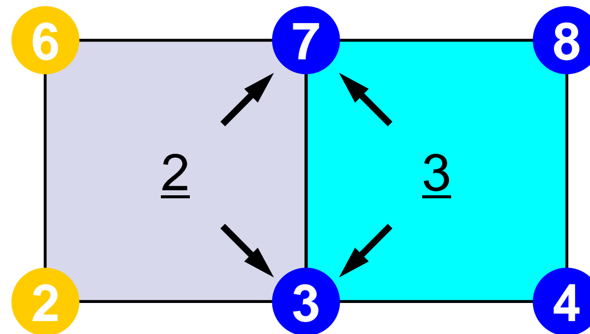
自由度:節点上で定義





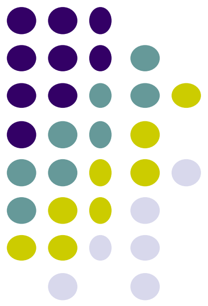
これではマトリクス生成に必要な情報は不十分





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

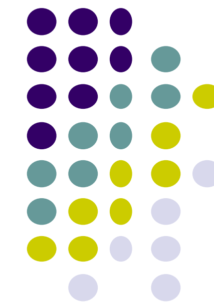
並列有限要素法の局所データ構造

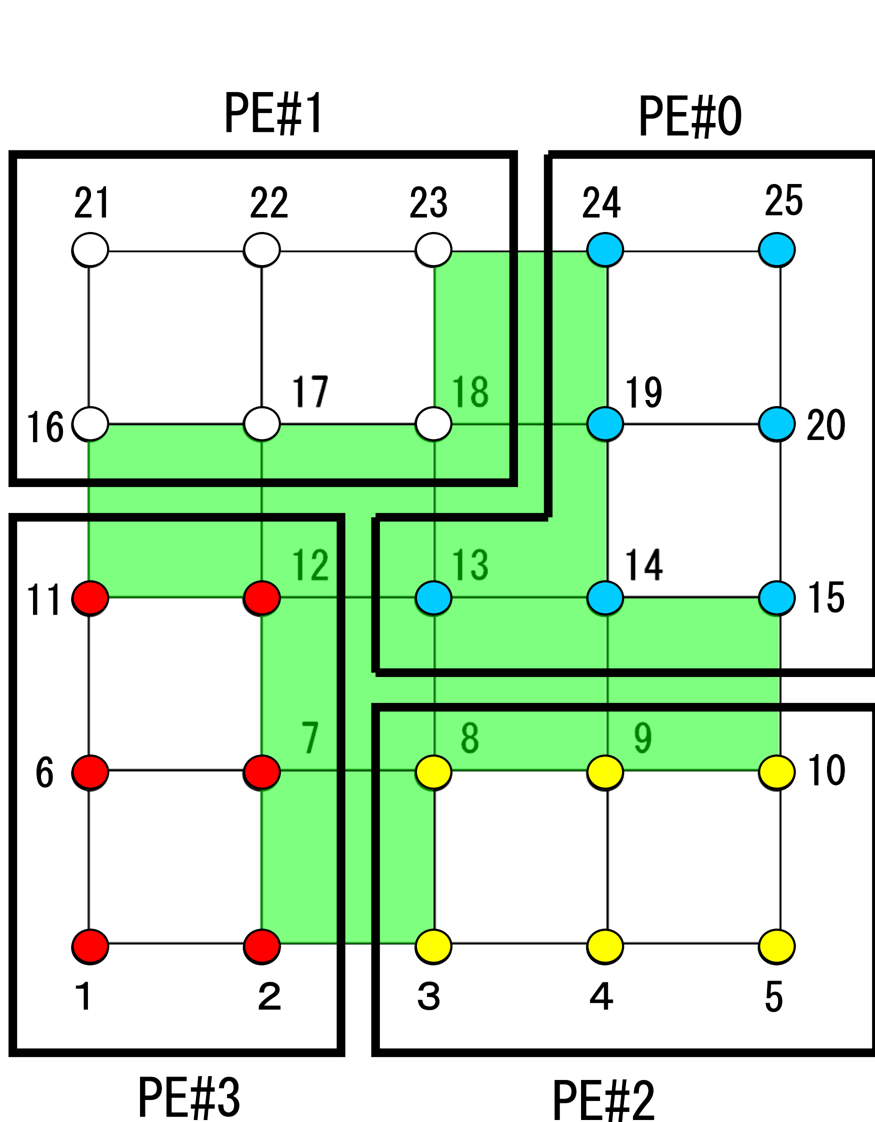


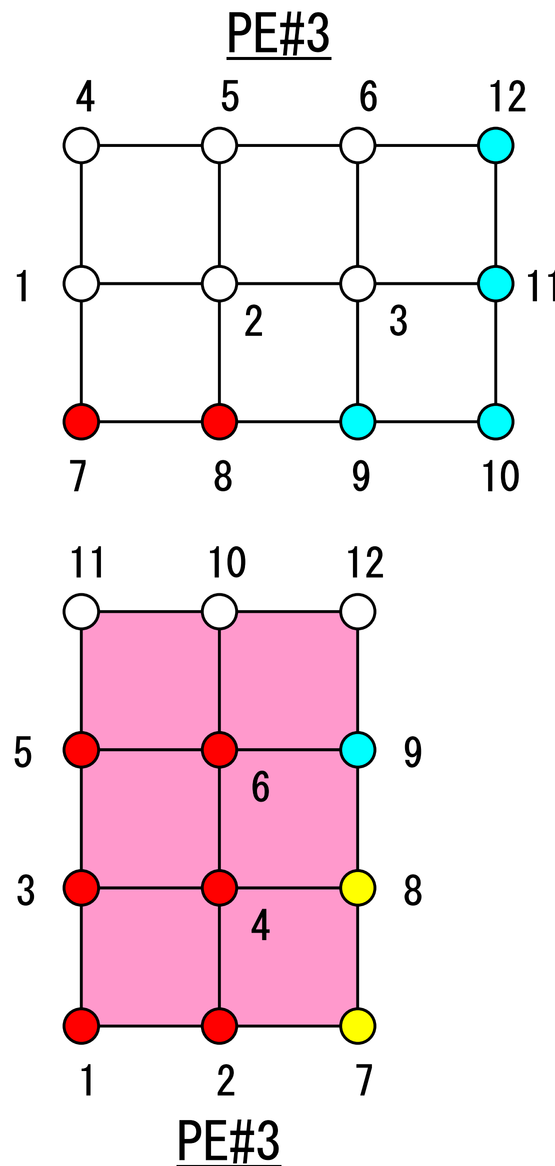
- 節点ベース：Node-based partitioning
- 局所データに含まれるもの：
  - その領域に本来含まれる節点
  - それらの節点を含む要素
  - 本来領域外であるが，それらの要素に含まれる節点
- 節点は以下の3種類に分類
  - 内点：Internal nodes　　その領域に本来含まれる節点
  - 外点：External nodes　　本来領域外であるがマトリクス生成に必要な節点
  - 境界点：Boundary nodes　　他の領域の「外点」となっている節点
- 領域間の通信テーブル
- 領域間の接続をのぞくと，大域的な情報は不要
  - 有限要素法の特性：要素で閉じた計算

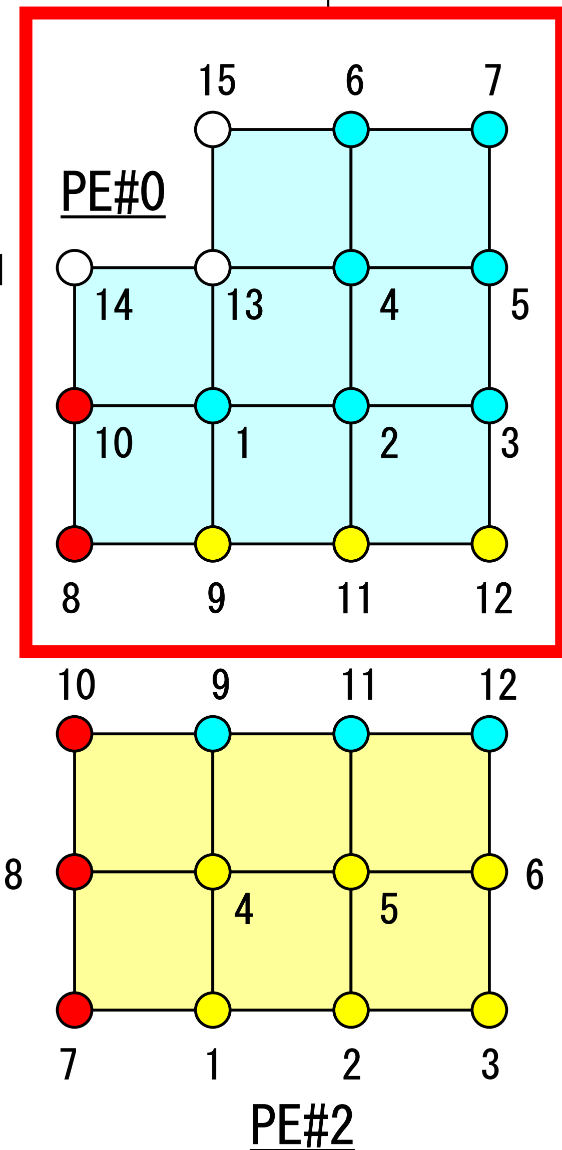
Node-based Partitioning

internal nodes - elements - external nodes



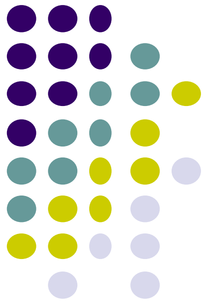






Node-based Partitioning

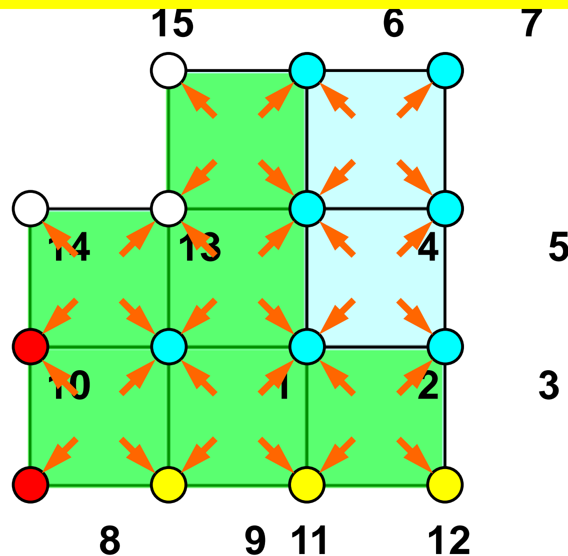
internal nodes - elements - external nodes

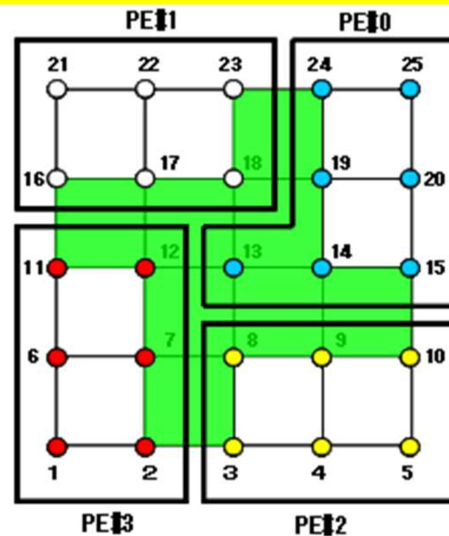


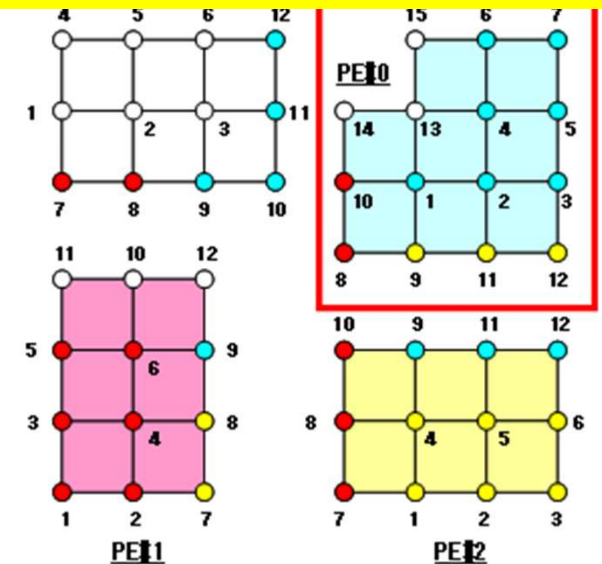
- Partitioned nodes themselves (<u>Internal Nodes</u>) 内点
- Elements which include Internal Nodes 内点を含む要素
- <u>External Nodes</u> included in the Elements 外点

  in overlapped region among partitions.

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

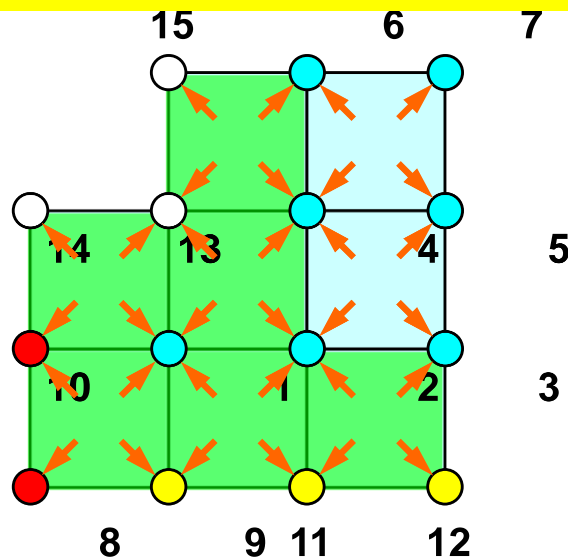


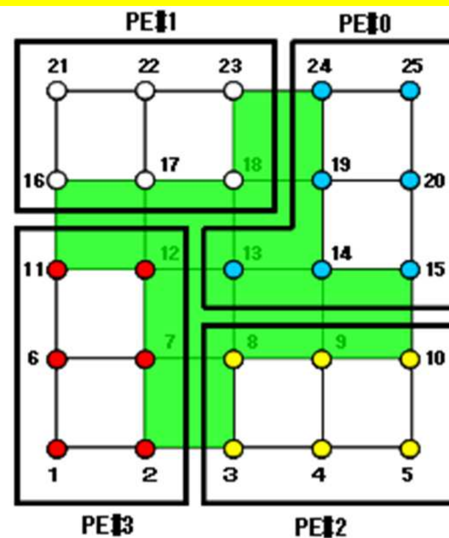


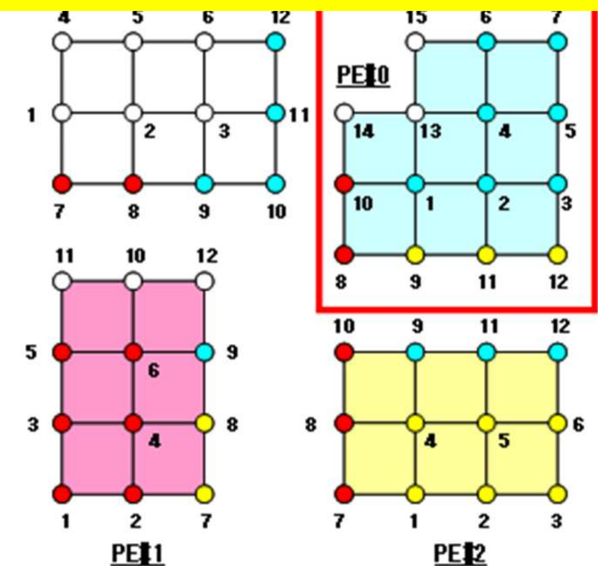


マトリクス生成時の通信は不要

- Partitioned nodes themselves (<u>Internal Nodes</u>)　内点
- Elements which include Internal Nodes　内点を含む要素
- <u>External Nodes</u> 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.







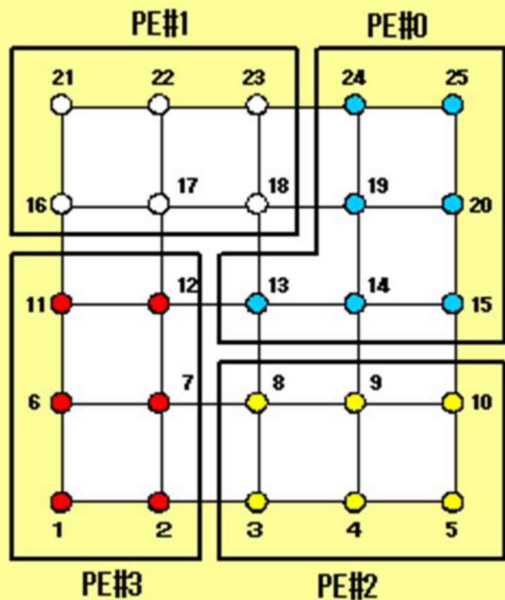
Parallel Computing in FEM

SPMD: Single-Program Multiple-Data

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



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

Mesh	Local Data	→	FEM code	↔	Linear Solvers
1	Local Data	→	FEM code	↔	Linear Solvers
2	Local Data	→	FEM code	↔	Linear Solvers
3	Local Data	→	FEM code	↔	Linear Solvers
4	Local Data	→	FEM code	↔	Linear Solvers

MPI (between Linear Solvers 1–2, 2–3, 3–4)

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

Local Data → FEM code ↔ Linear Solvers

MPI

Local Data → FEM code ↔ Linear Solvers

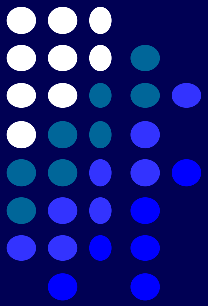
MPI

Local Data → FEM code ↔ Linear Solvers

MPI

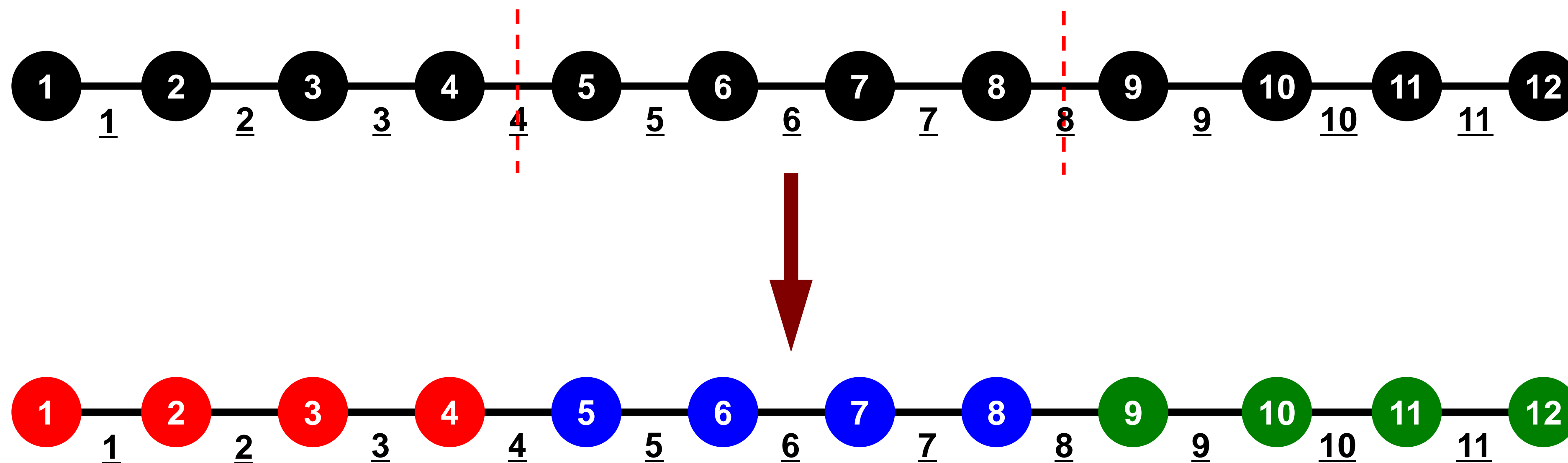
Local Data → FEM code ↔ Linear Solvers

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

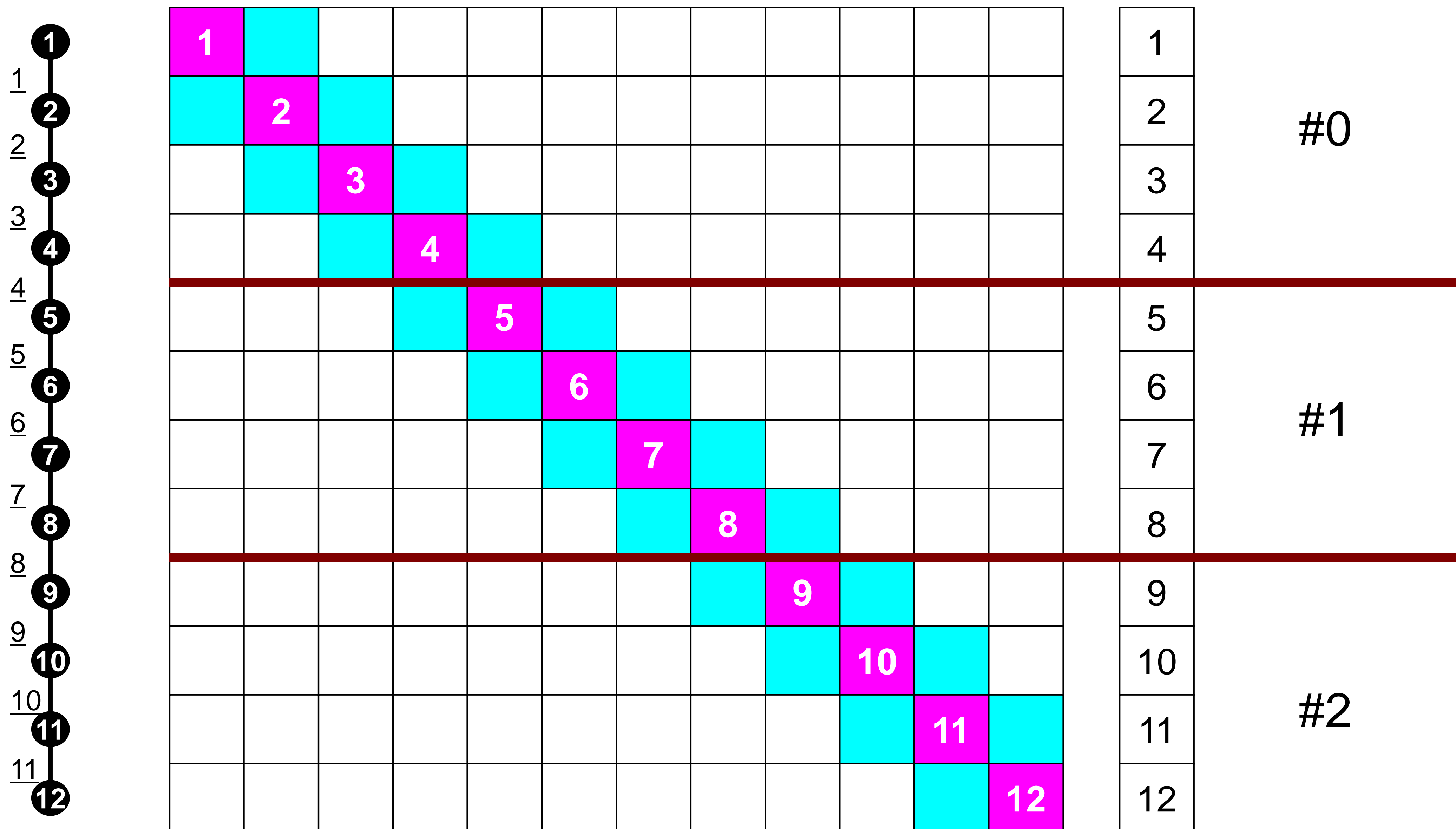
三重対角行列：Tri-Diagonal Matrix

1 1 2 2 3 3 4 4 5 5 6 6 7 7 8 8 9 9 10 10 11 11 12

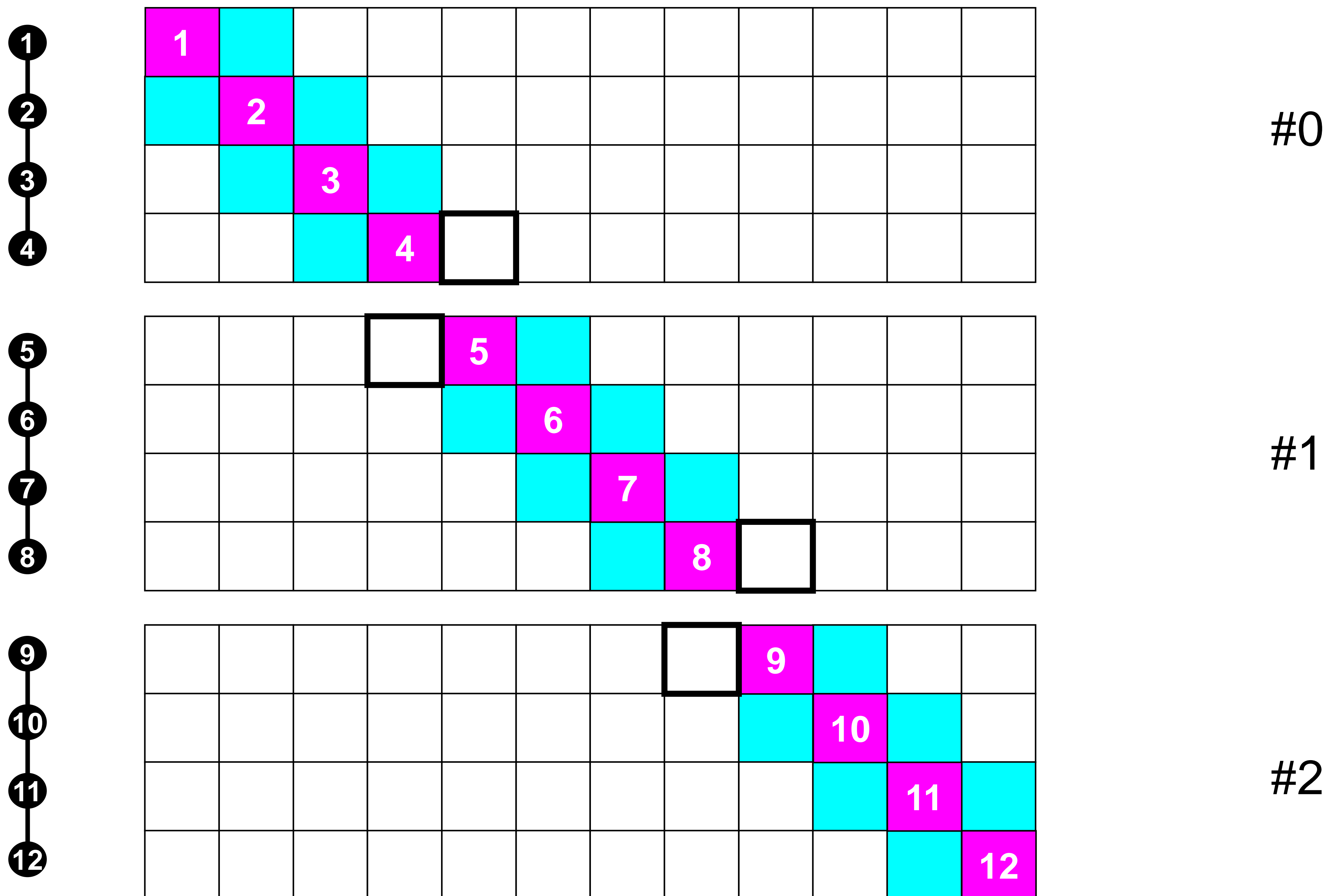
1											
	2										
		3									
			4								
				5							
					6						
						7					
							8				
								9			
									10		
										11	
											12

1
2
3
4
5
6
7
8
9
10
11
12

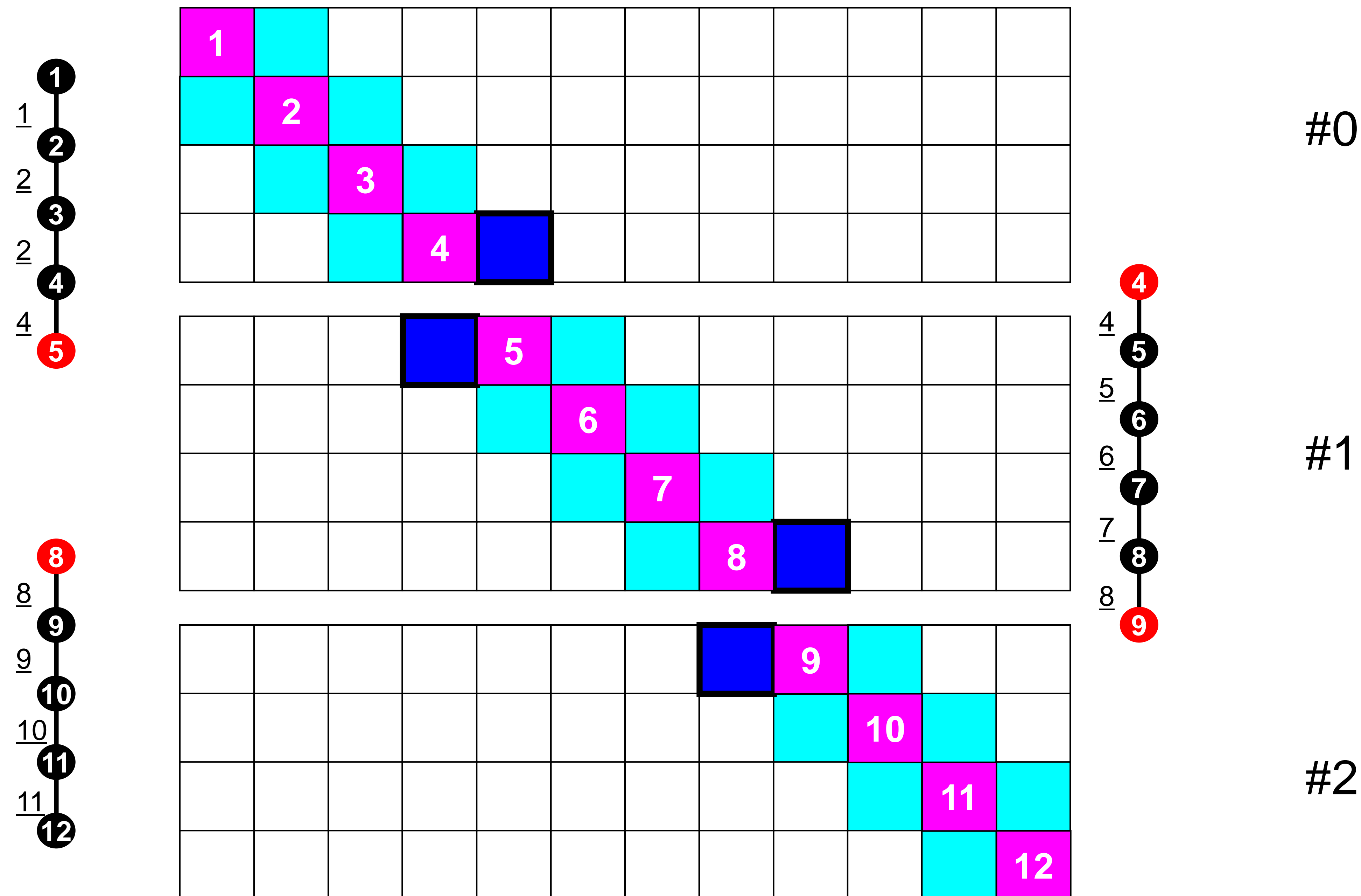
節点がバランスするよう分割：内点



内点だけで分割するとマトリクス不完全

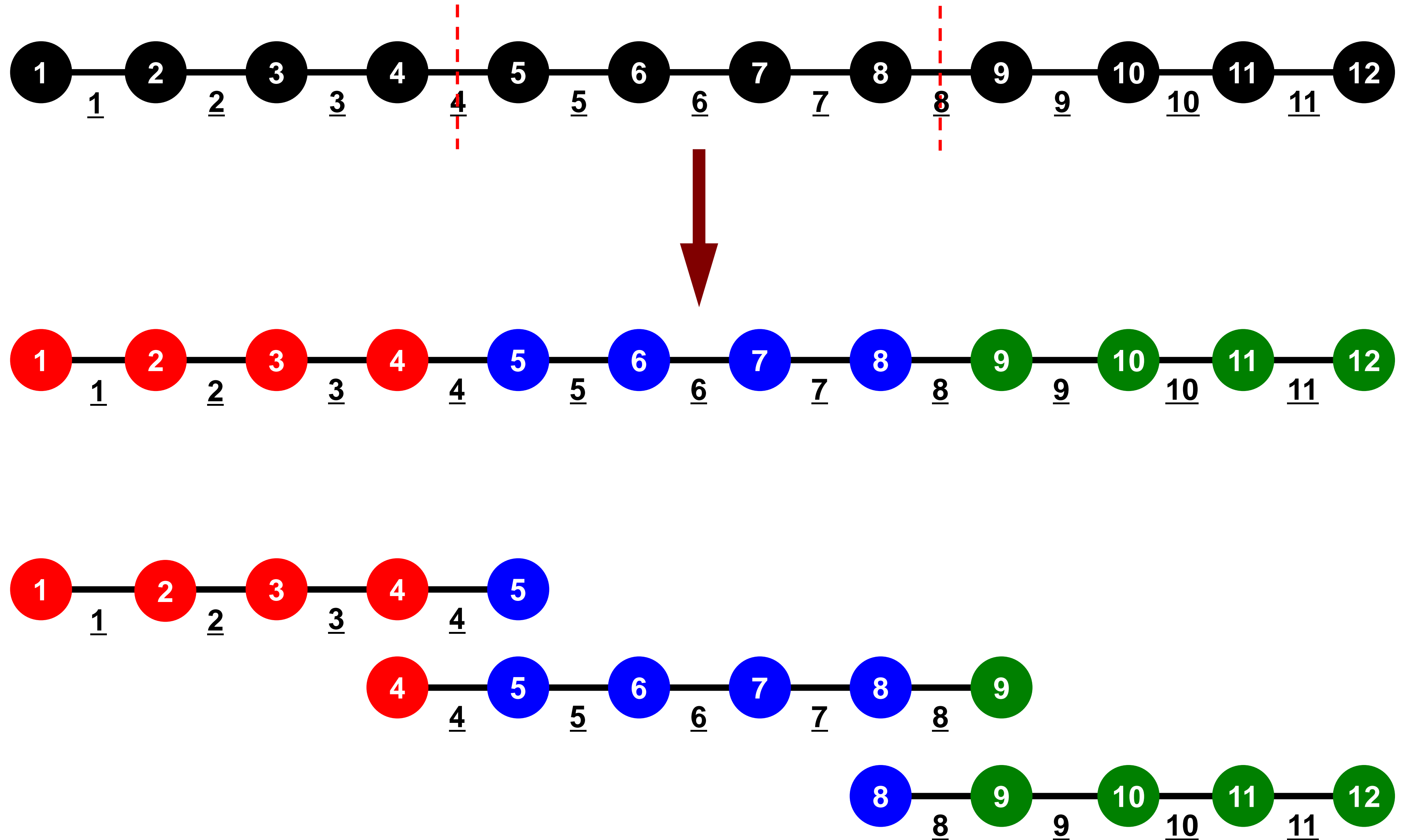


要素＋外点



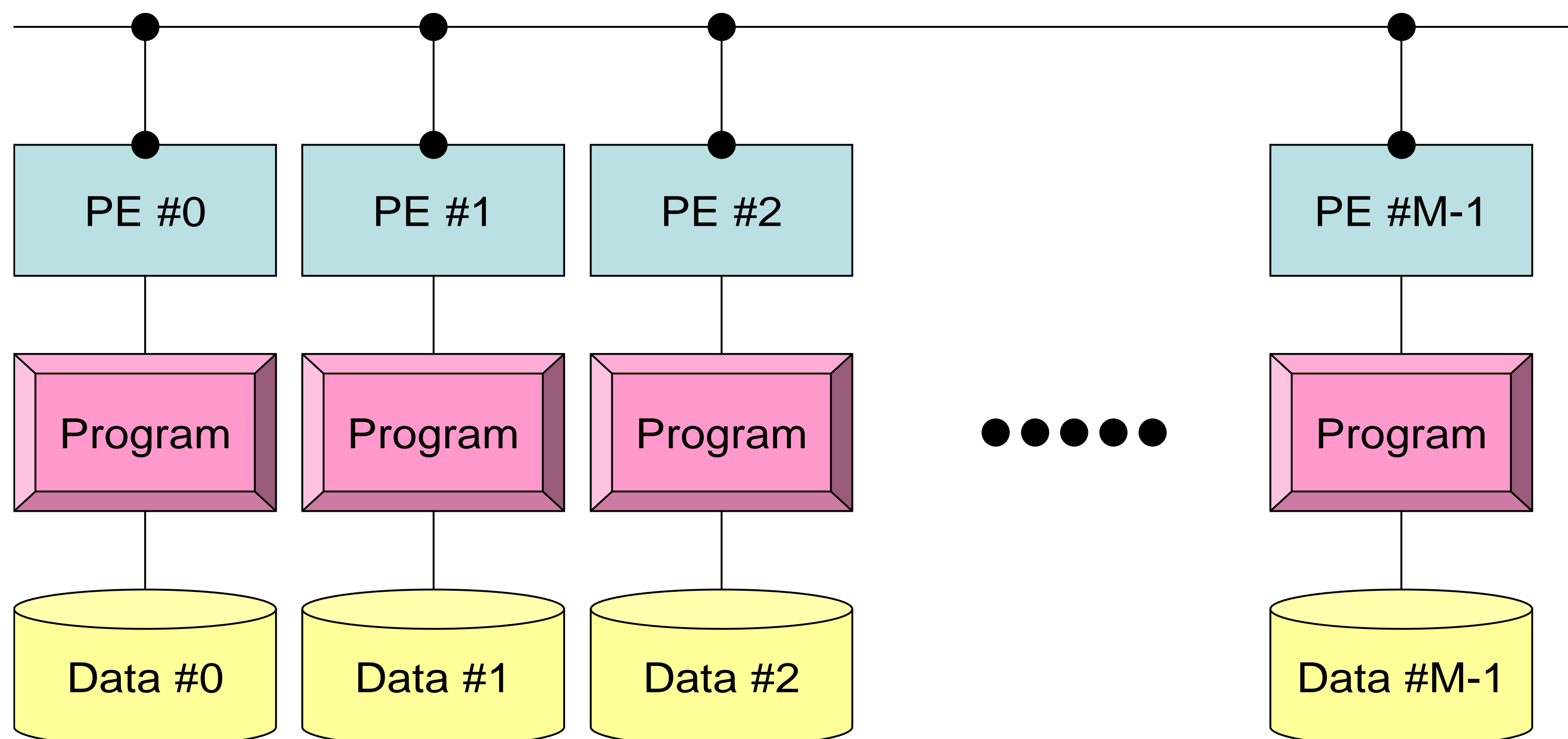
一次元問題：11要素，12節点，3領域

一次元問題：11要素，12節点，3領域



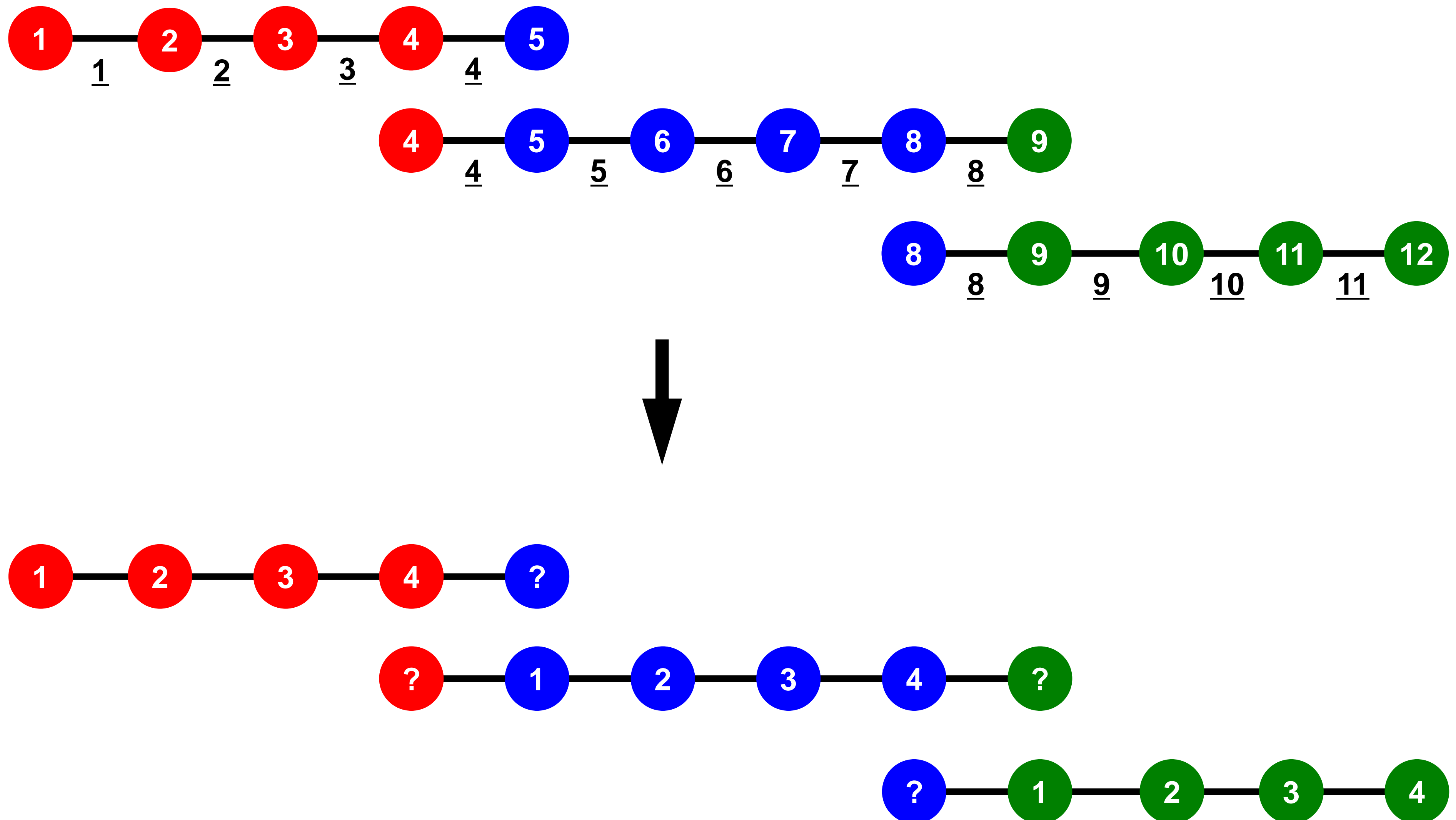
MPIによる並列化：SPMD

- 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



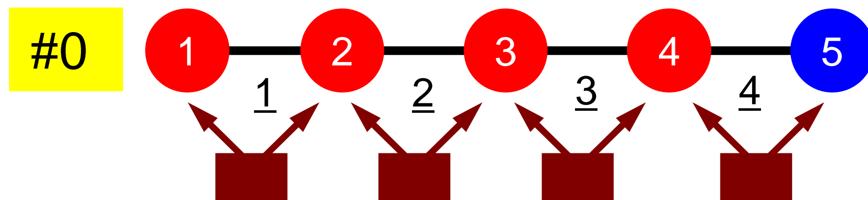
SPMD向け局所番号付け

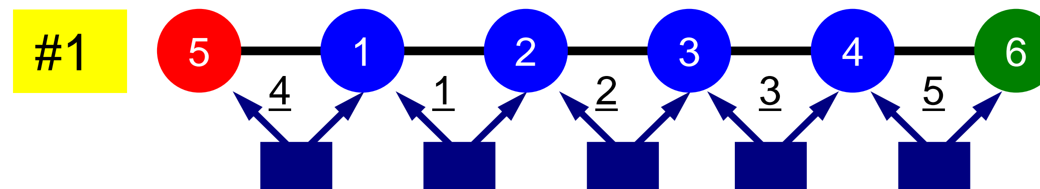
内点が1~Nとなっていれば，もとのプログラムと同じ
外点の番号は?

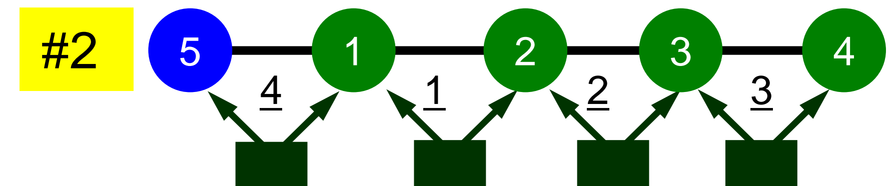


並列有限要素法：マトリクス生成

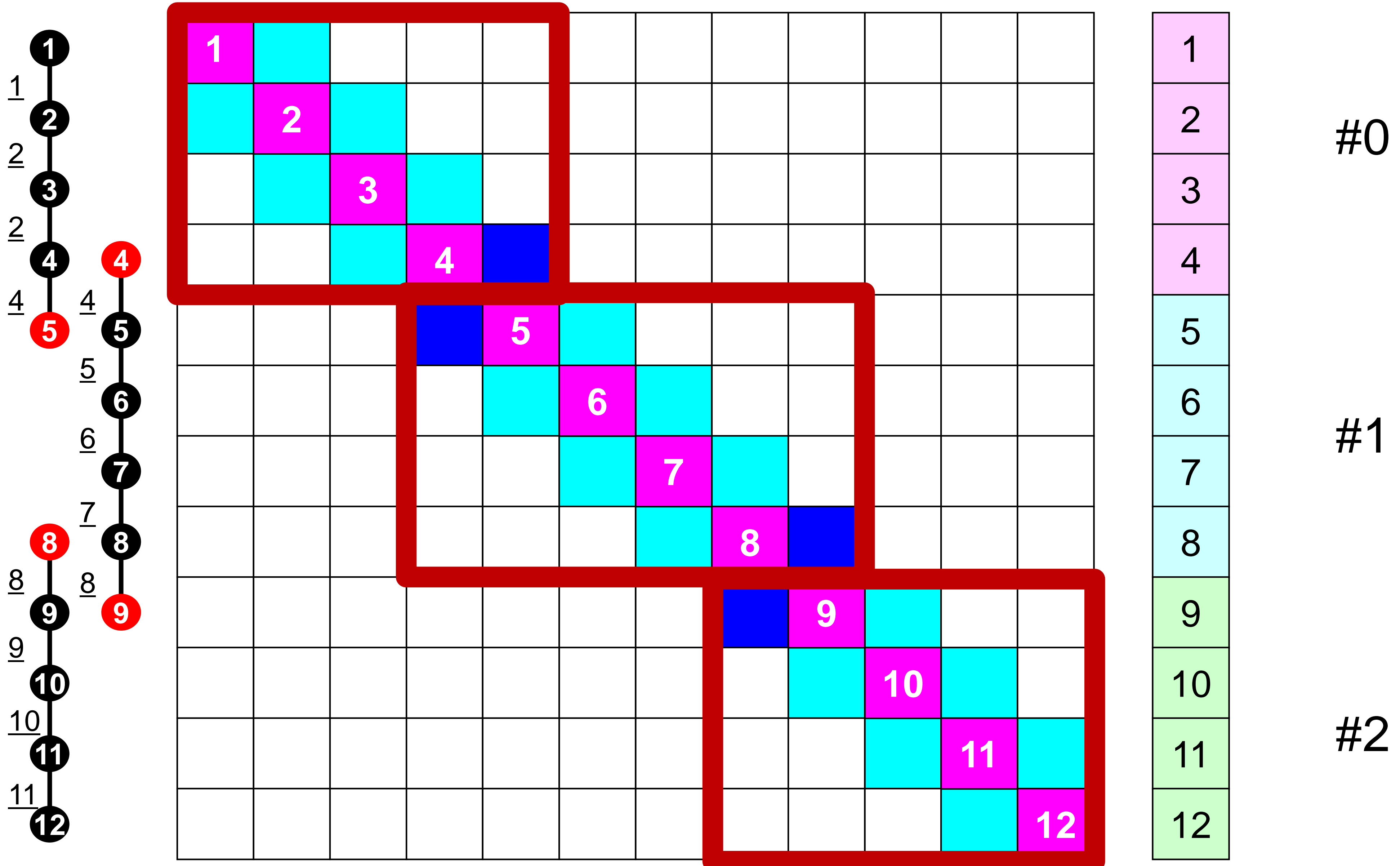
各要素での積分，要素マトリクス⇒（局所）全体マトリクス生成の計算は，内点・外点，それらを含む要素の情報があれば可能







係数行列が疎であるため，局所行列の和集合が全体行列となる



- 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 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 r(0)= b-[A]x(0)
for i= 1, 2, …
    solve [M]z(i-1)= r(i-1)
    ρi-1= r(i-1) z(i-1)
    if i=1
      p(1)= z(0)
     else
      βi-1= ρi-1/ρi-2
      p(i)= z(i-1) + βi-1 p(i-1)
    endif
    q(i)= [A]p(i)
    αi = ρi-1/p(i)q(i)
    x(i)= x(i-1) + αip(i)
    r(i)= r(i-1) - αiq(i)
    check convergence |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

$$[M]=\begin{bmatrix} D_1 & 0 & \dots & 0 & 0 \\ 0 & D_2 & & 0 & 0 \\ \dots & & \dots & & \dots \\ 0 & 0 & & D_{N-1} & 0 \\ 0 & 0 & \dots & 0 & D_N \end{bmatrix}$$

前処理，ベクトル定数倍の加減

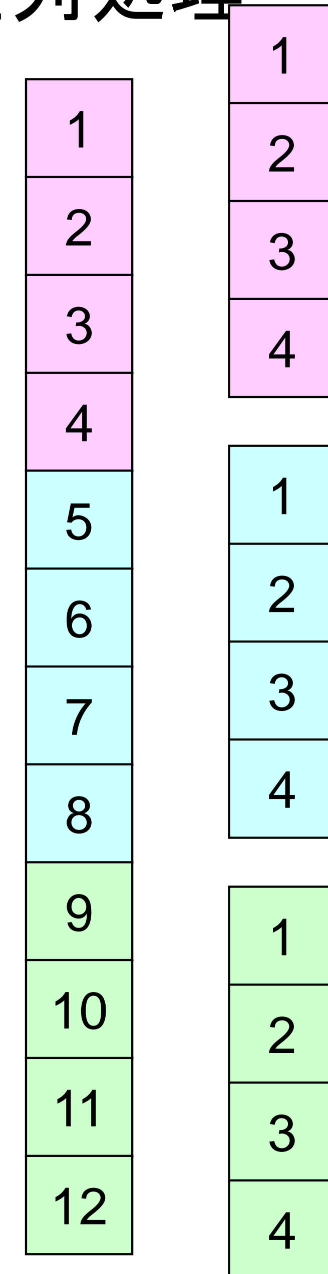
局所的な計算（内点のみ）が可能⇒並列処理

```
!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
```

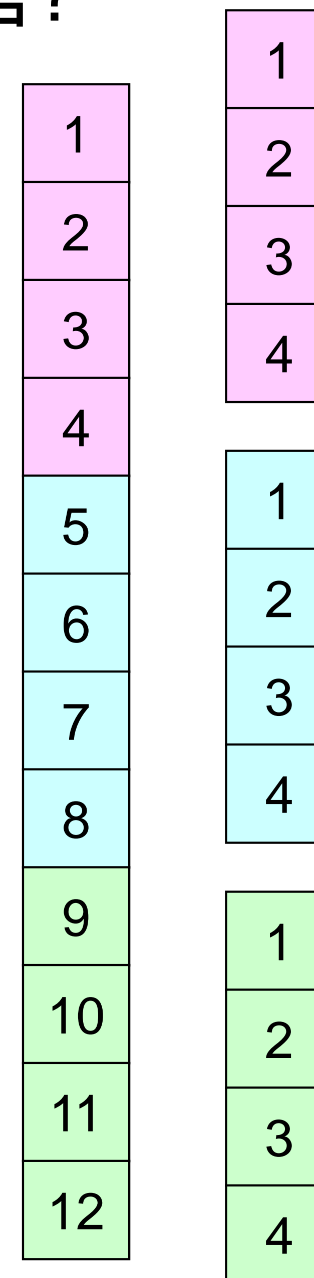


内積

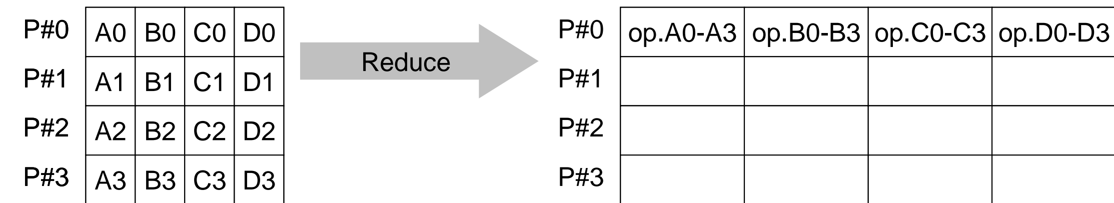
全体で和をとる必要がある⇒通信?

```
!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

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

```
Compute r(0)= b-[A]x(0)
for i= 1, 2, …
    solve [M]z(i-1)= r(i-1)
    ρi-1= r(i-1) z(i-1)
    if i=1
      p(1)= z(0)
     else
      βi-1= ρi-1/ρi-2
      p(i)= z(i-1) + βi-1 p(i-1)
    endif
    q(i)= [A]p(i)
    αi = ρi-1/p(i)q(i)
    x(i)= x(i-1) + αip(i)
    r(i)= r(i-1) - αiq(i)
    check convergence |r|
end
```

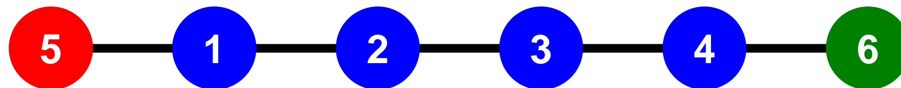
- 前処理
  - 対角スケーリング
- 並列処理が必要なプロセス
  - 内積
  - 行列ベクトル積

$$[M]=\begin{bmatrix} D_1 & 0 & \dots & 0 & 0 \\ 0 & D_2 & & 0 & 0 \\ \dots & & \dots & & \dots \\ 0 & 0 & & D_{N-1} & 0 \\ 0 & 0 & \dots & 0 & D_N \end{bmatrix}$$

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

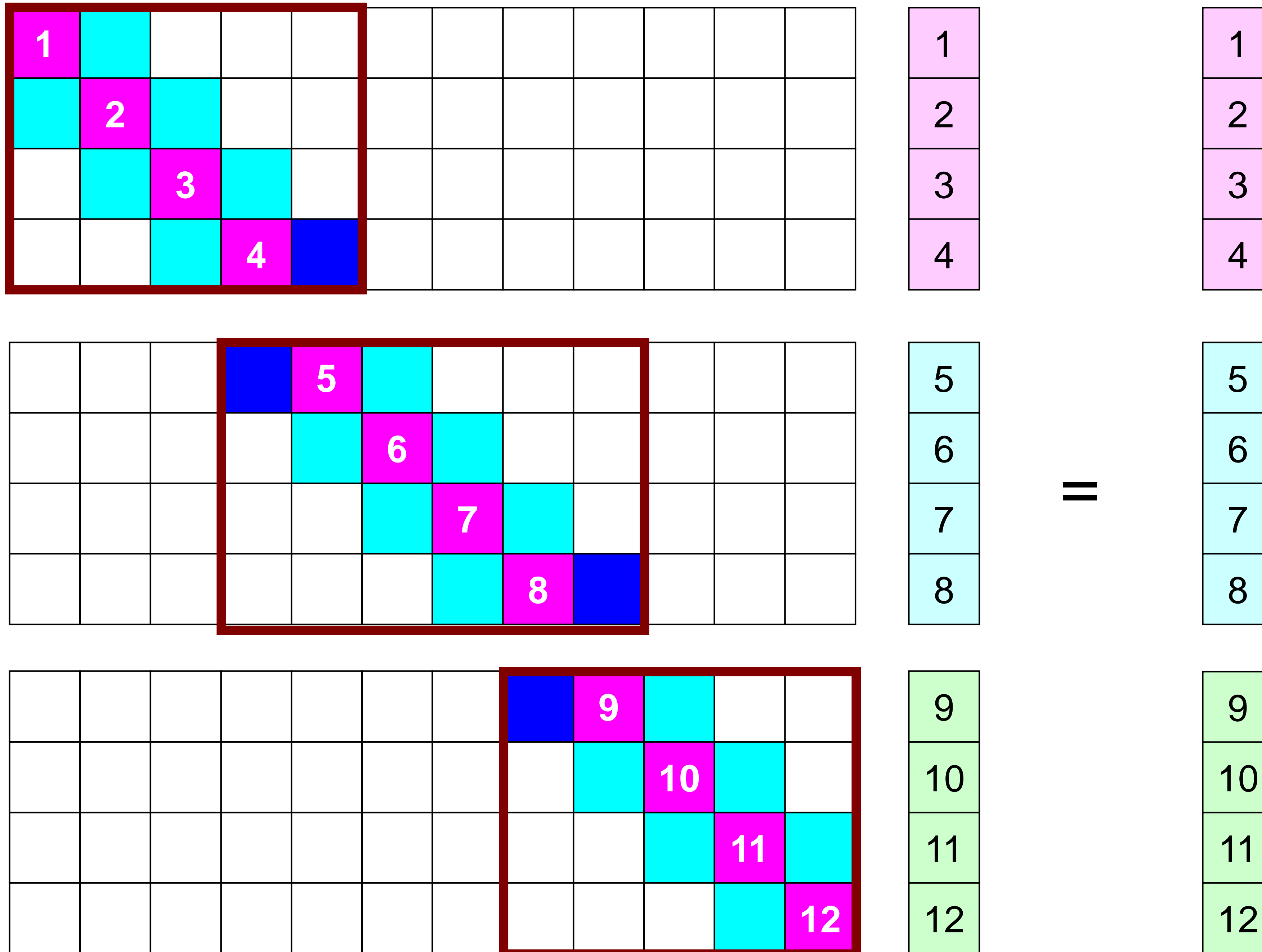
1 2 3 4 5 6 7 7 9 10 11 12

1 2 3 4 5 6 7 8 9 10 11 12

=

1 2 3 4 5 6 7 8 9 10 11 12

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
2
3
4

5
6
7
8

9
10
11
12

1
2
3
4

5
6
7
8

9
10
11
12

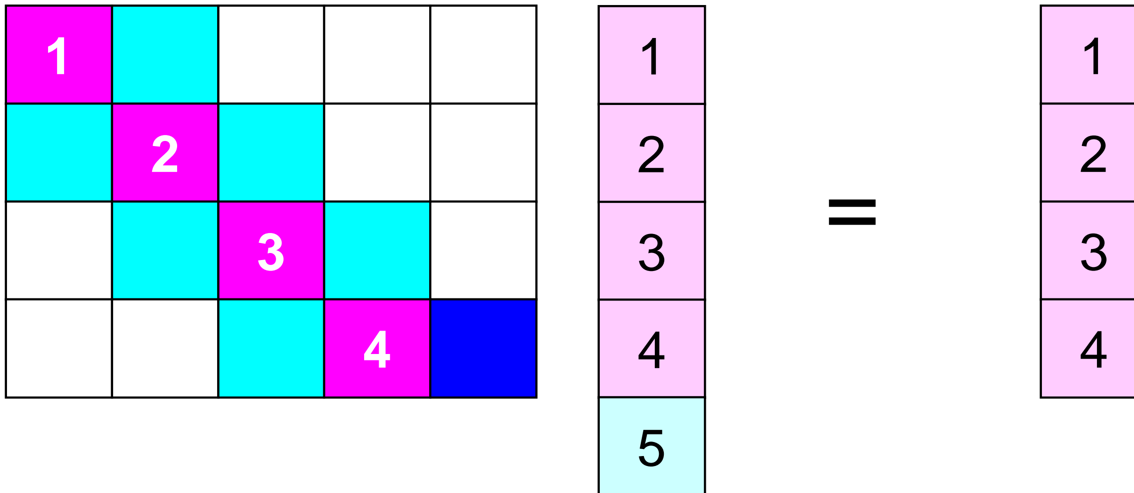
=

1
2
3
4

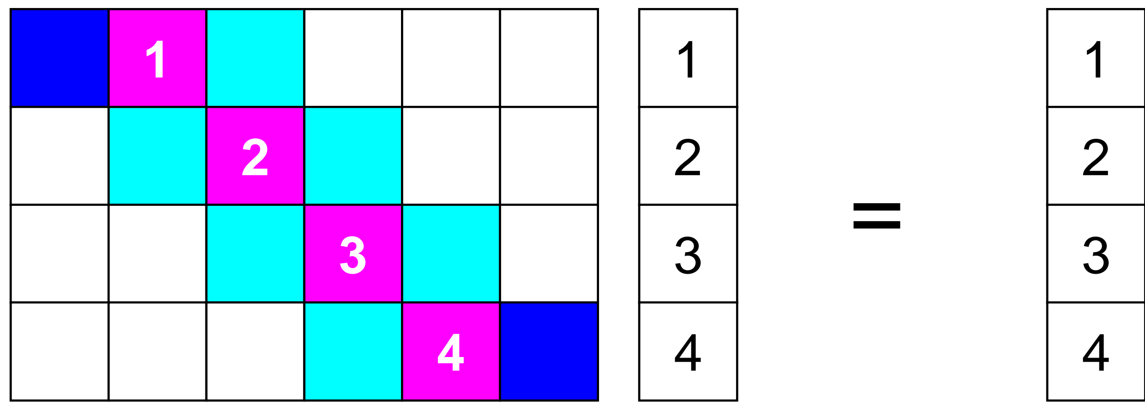
5
6
7
8

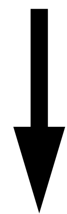
9
10
11
12

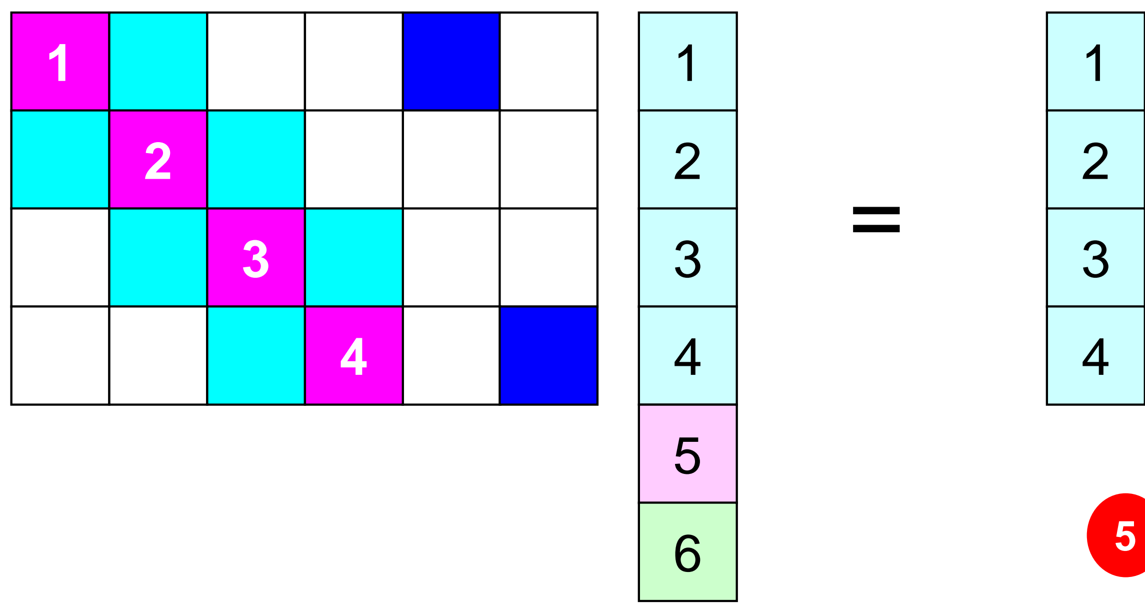
行列ベクトル積：ローカル計算 #0

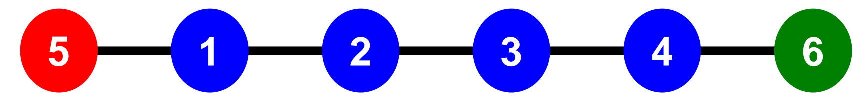


行列ベクトル積：ローカル計算 #1

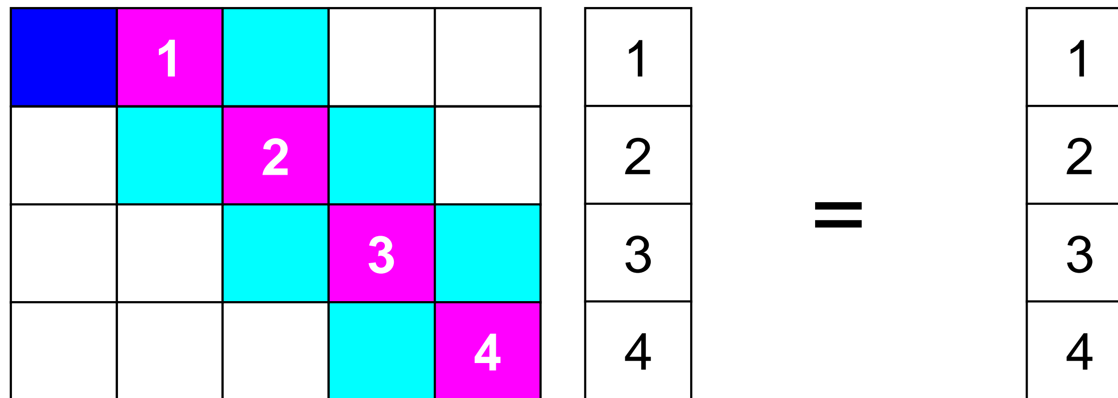


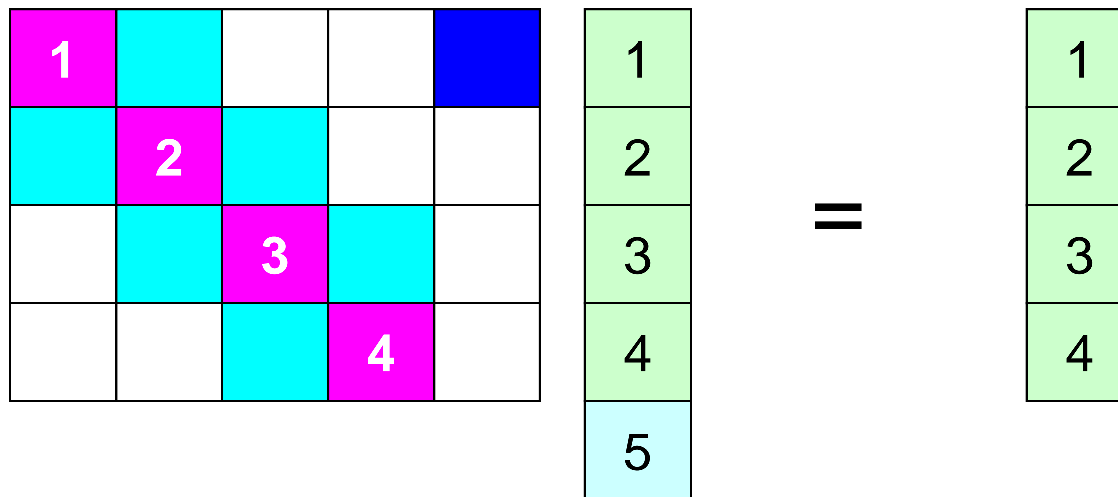




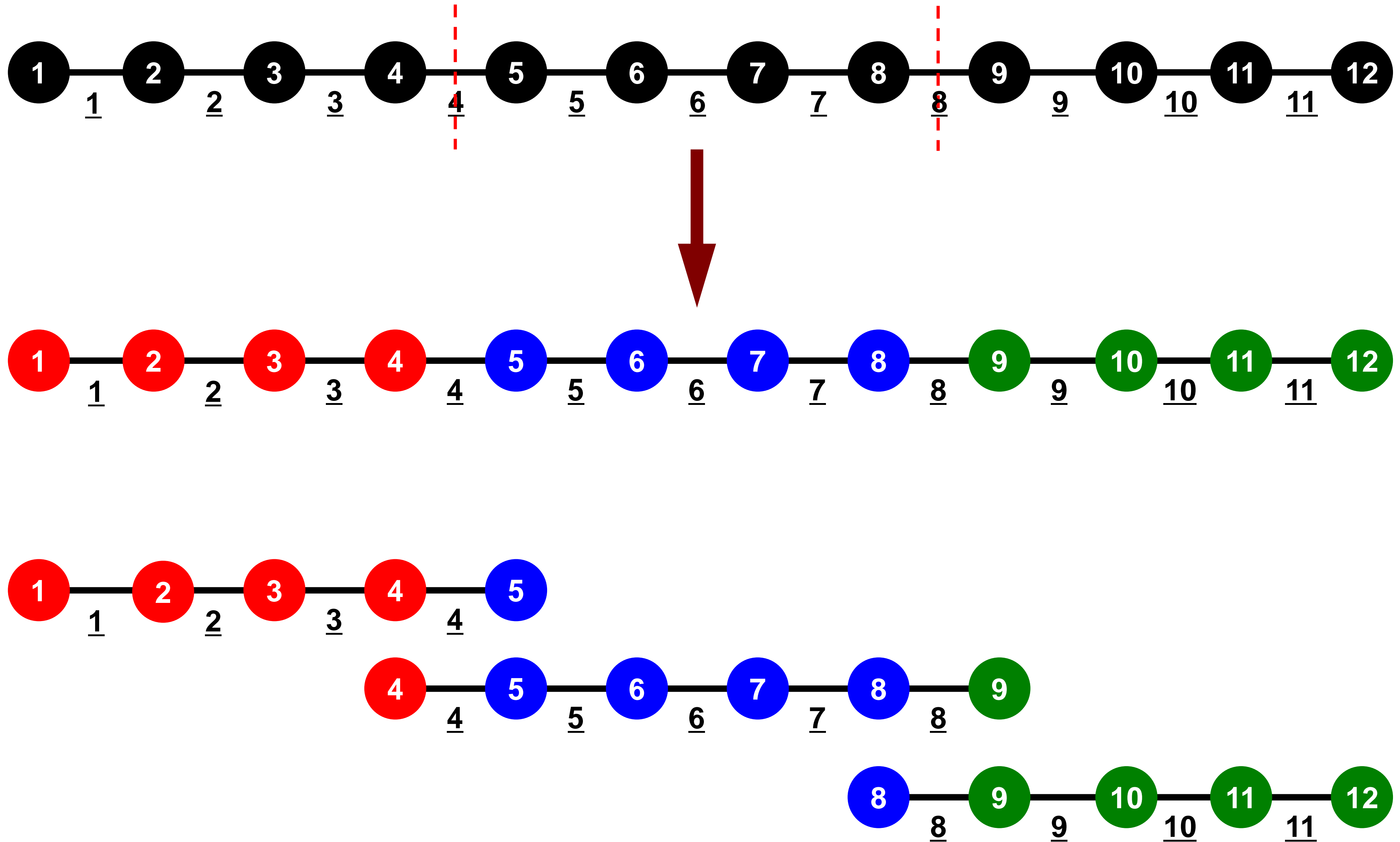


行列ベクトル積：ローカル計算 #2



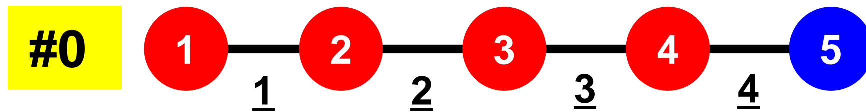


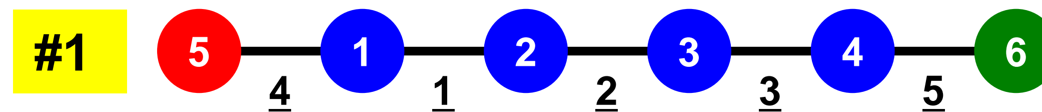
1D FEM: 12 nodes/11 elem's/3 domains

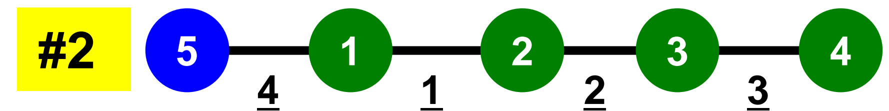


1D FEM: 12 nodes/11 elem's/3 domains

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

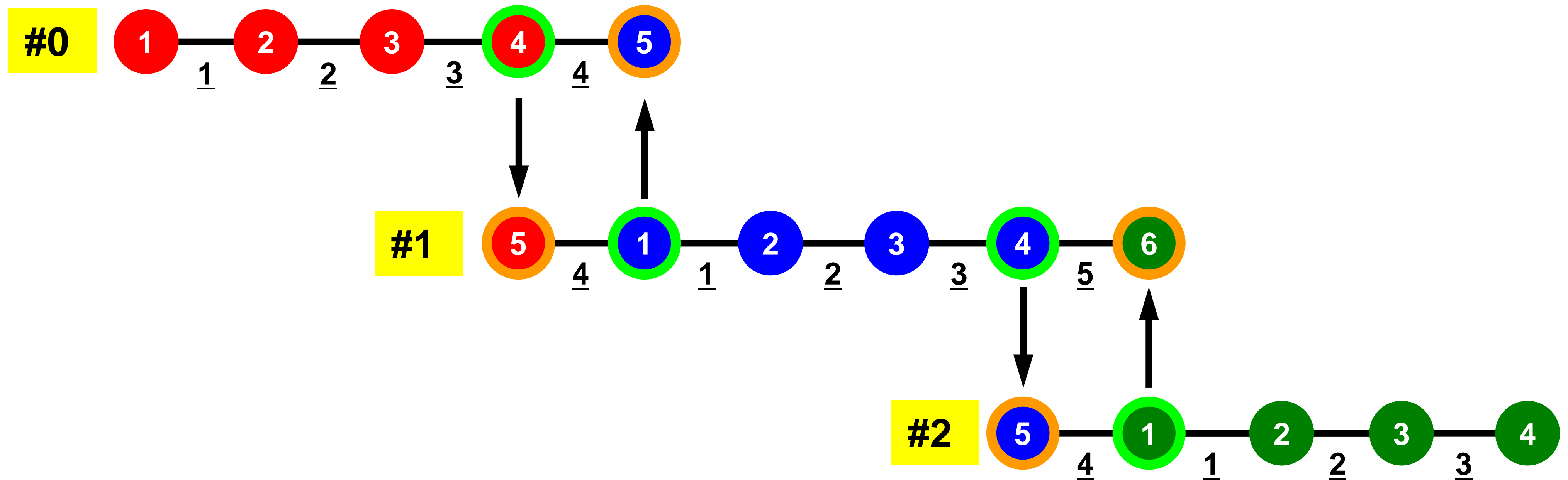






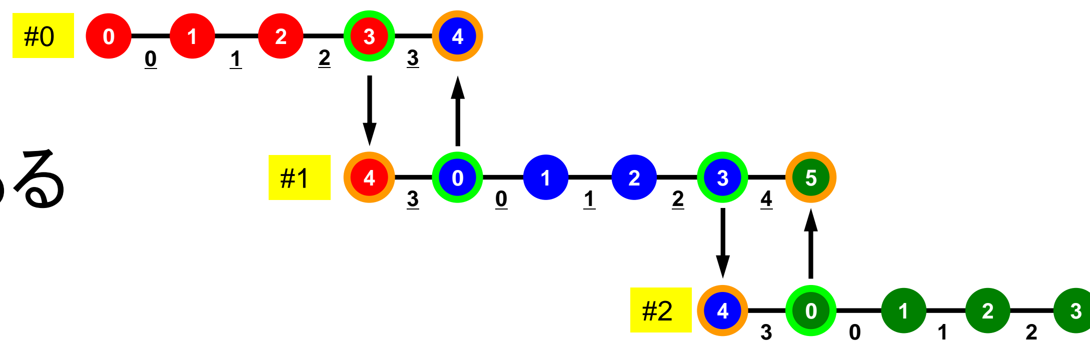
1D FEM: 12 nodes/11 elem's/3 domains

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_Recv
  - 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



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

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