Introduction to Parallel Programming for Multicore/Manycore Clusters

Part B2: Reordering

Kengo Nakajima
Information Technology Center
The University of Tokyo

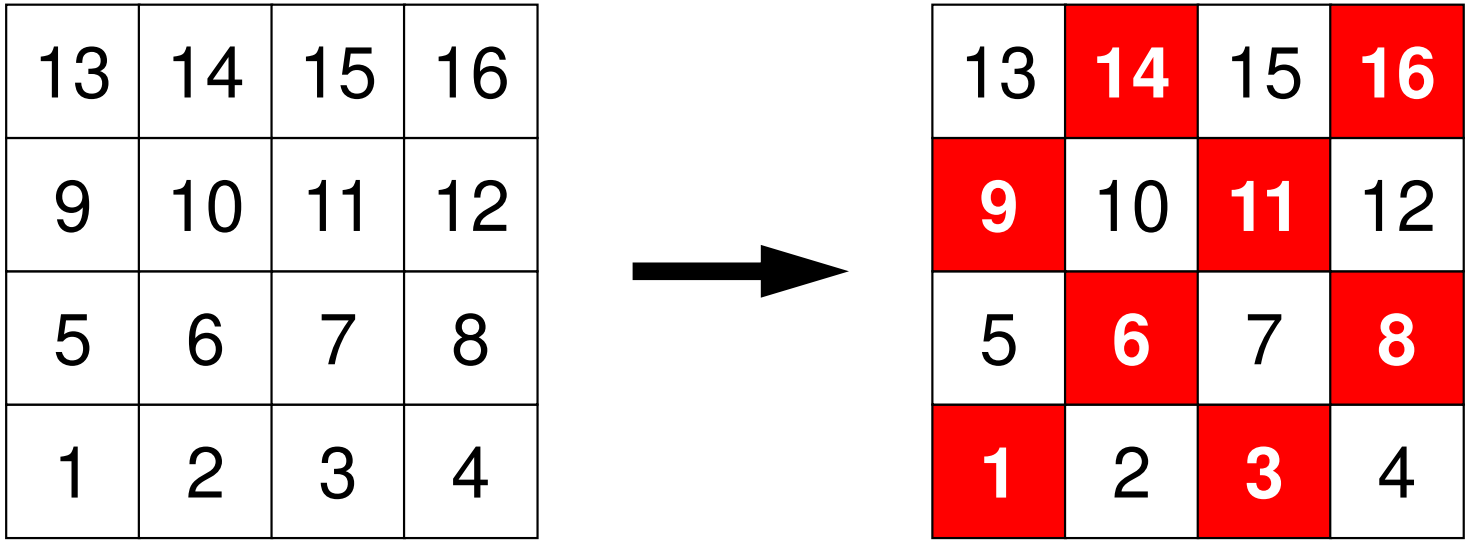
- Remedy for Data Dependency
- Ordering/Reordering
  - Red-Black, Multicoloring (MC)
  - Cuthill-McKee (CM), Reverse-CM (RCM)
  - Reordering and Convergence
- Implementation
- ICCG with Reordering

Parallelize ICCG Method

- Dot Product: OK
- DAXPY: OK
- Matrix-Vector Multiply: OK
- Preconditioning: Something special needed !
  - Just inserting OpenMP directive is not enough

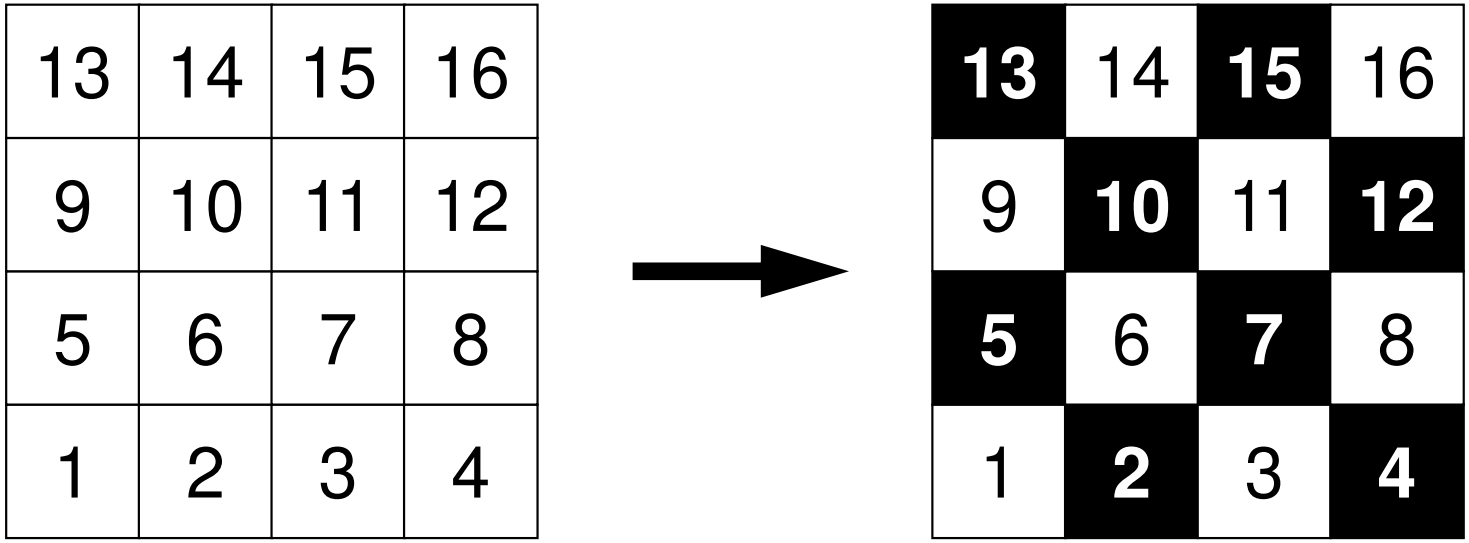
One Remedy for Data Dependency = Coloring

- Parallel (concurrent) processing is possible for independent meshes without dependency



One Remedy for Data Dependency = Coloring

- Parallel (concurrent) processing is possible for independent meshes without dependency



One Remedy for Data Dependency = Coloring

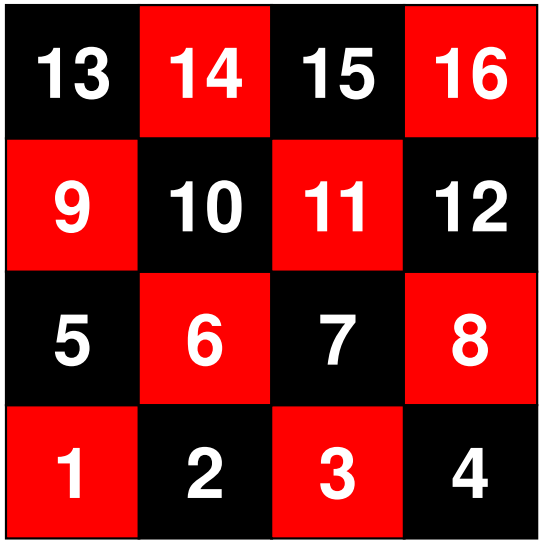
- Applying same “color” to independent meshes without dependency: Coloring
- Most simple case: Red-Black with 2 Colors

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

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

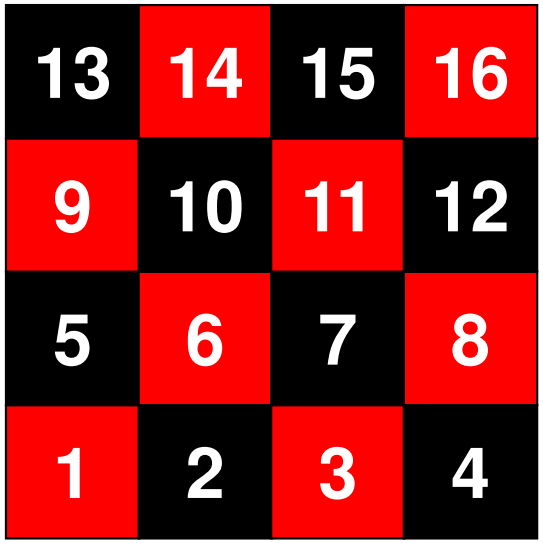
Numbering starts at 0 in the program, but I would like to use this one starting at 1. Please do not confuse !!

Red-Black (1/3)




```
#prgma omp parallel for private (ip,i,k,VAL)
for(ip=0; ip<4; ip++){
  for(i=INDEX[ip]; i<INDEX[ip+1]; i++){
    if (COLOR[i]== “RED” ){
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
  }
}

#prgma omp parallel for private (ip,i,k,VAL)
for(ip=0; ip<4; ip++){
  for(i=INDEX[ip]; i<INDEX[ip+1]; i++){
    if (COLOR[i]== “BLACK” ){
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
  }
}
```

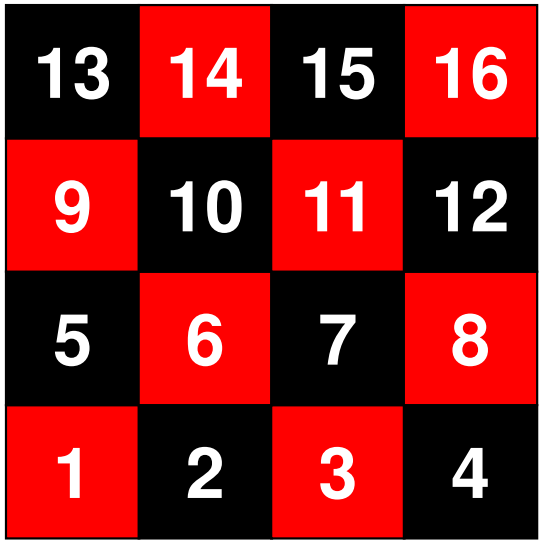
Red-Black (2/3)




```
#prgma omp parallel for private (ip,i,k,VAL)
for(ip=0; ip<4; ip++){
  for(i=INDEX[ip]; i<INDEX[ip+1]; i++){
    if (COLOR[i]== “RED” ){
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
  }
}
```

- During operations on “red” meshes, only “black” meshes appear in RHS.
  - “red”: writing, “black”: reading
- During operations on “red” meshes, values on “black” meshes do not change.
- Data dependency is avoided.

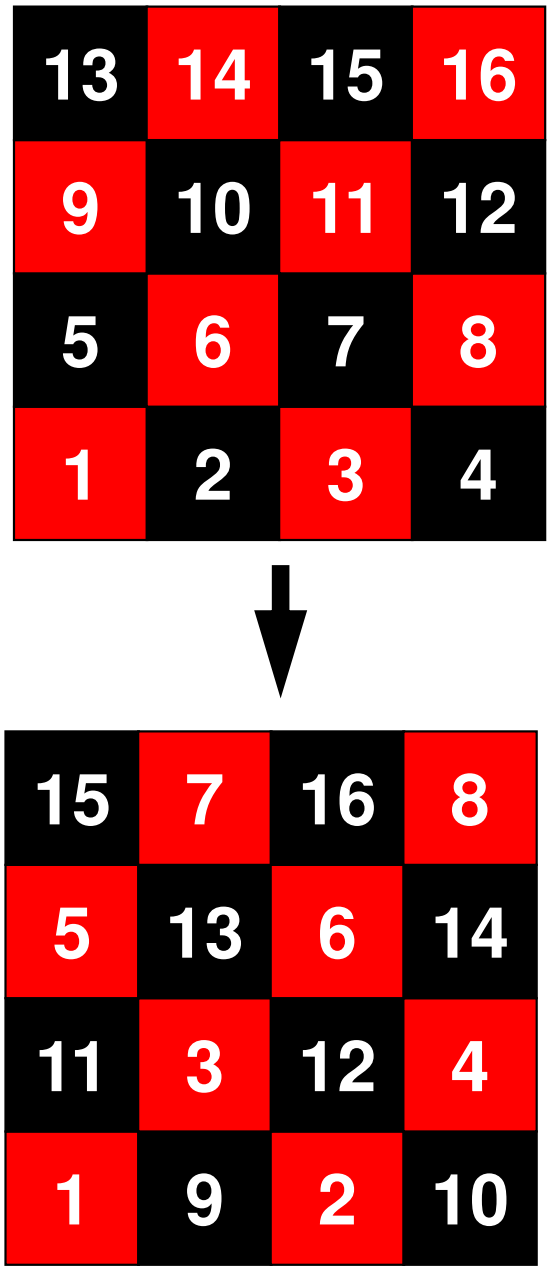
Red-Black (3/3)



- During operations on “black” meshes, only “red” meshes appear in RHS.
  - “black”: writing, “red”: reading
- During operations on “black” meshes, values on “red” meshes do not change.
- Data dependency is avoided.

```
#prgma omp parallel for private (ip,i,k,VAL)
for(ip=0; ip<4; ip++){
  for(i=INDEX[ip]; i<INDEX[ip+1]; i++){
    if (COLOR[i]== “BLACK” ){
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
  }
}
```

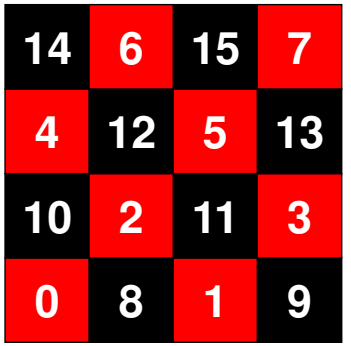
Red-Black Ordering/Reordering




```
for(icol=0; icol<2; icol++){
#pragma omp parallel for private (ip,i,j,VAL)
  for(ip=0; ip<4; ip++){
      for(i=INDEX[ip][icol]; i<INDEX[ip+1][icol]; i++){
          WVAL = W[Z][i];
          for(j=indexL[i]; j<indexL[i+1]; j++) {
              WVAL -= AL[j] * W[Z][itemL[j]-1];
          }
          W[Z][i] = WVAL * W[DD][i];
      }
  }
}
```

```
INDEX[0][0]= 0
INDEX[1][0]= 2
INDEX[2][0]= 4
INDEX[3][0]= 6
INDEX[4][0]= 8
```

```
INDEX[0][1]=  8
INDEX[1][1]= 10
INDEX[2][1]= 12
INDEX[3][1]= 14
INDEX[4][1]= 16
```



- Renumbering/reordering meshes from “red” to “black”
- Simpler, more efficient

- Remedy for Data Dependency
- **Ordering/Reordering**
  - **Red-Black, Multicoloring (MC)**
  - **Cuthill-McKee (CM), Reverse-CM (RCM)**
  - Reordering and Convergence
- Implementation
- ICCG with Reordering

Effect of Reordering

- **Extracting parallelism, removing dependency**
- Reducing
  - fill-in's, "bandwidth of matrix", "profile"
- Blocking

- Related to "four color problem", "travelling salesman problem" etc.
  - applied to numerical analysis

Ordering/Reordering Method for Parallel Computing

- Multicoloring (MC)
  - Parallelism
  - Red-Black with 2 colors
- CM (Cuthill-McKee), RCM (Reverse Cuthill-McKee)
  - Reducing fill-in's, matrix bandwidth, profiles
  - Parallelism

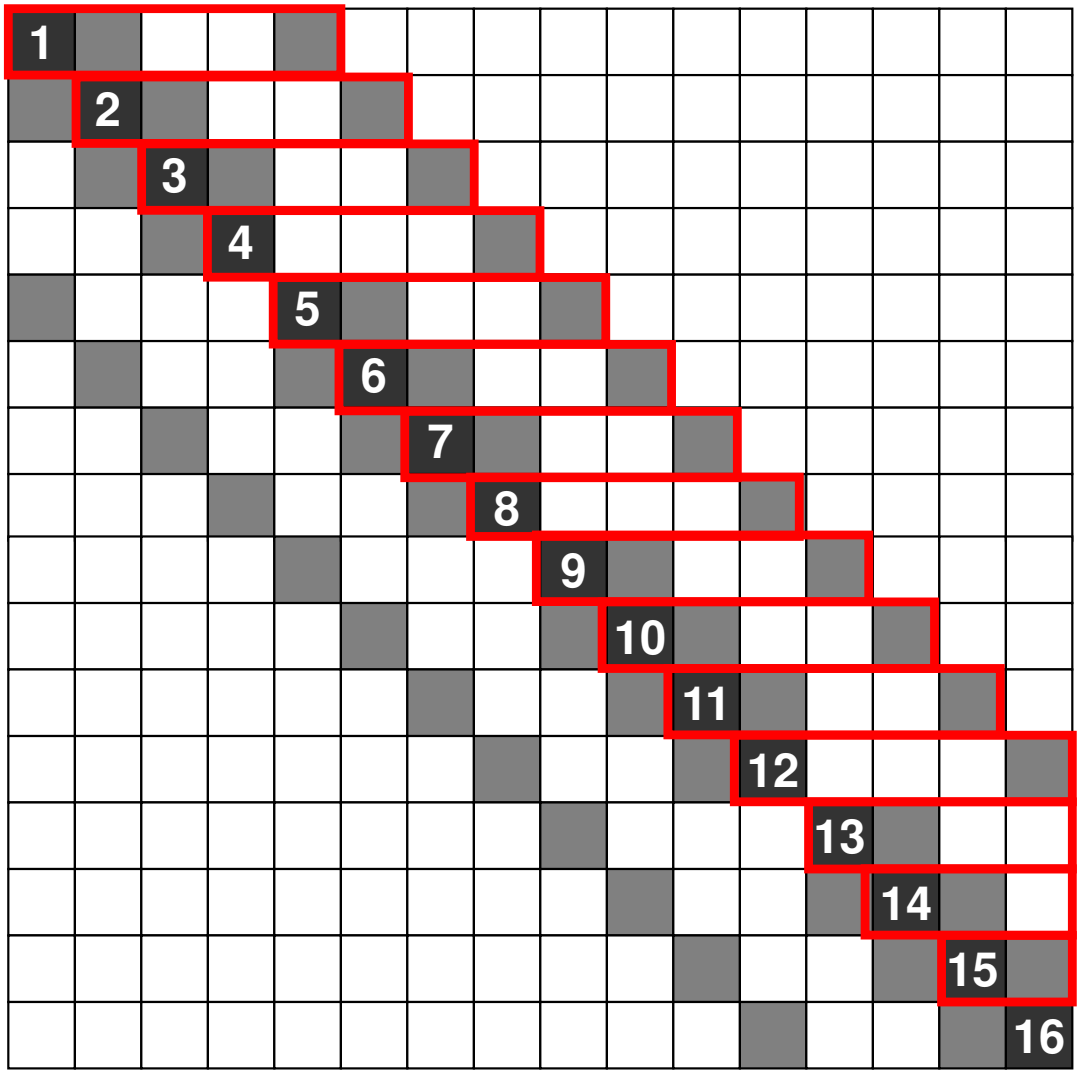
Technical Terms for Matrix

- β_i: $\beta_i = k - i$ where maximum ID number of non-zero column is k at i-th row of the target matrix
- Bandwidth: Maximum value of β_i
- Profile: Total sum of β_i

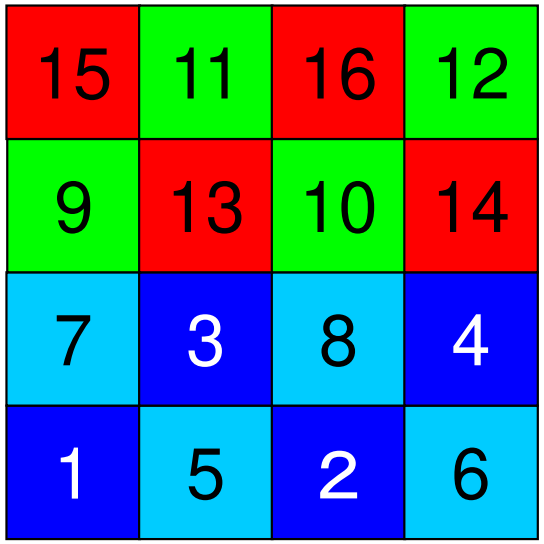
- Bandwidth, Profile, Fill-in
  - smaller is better
  - Both of “Bandwidth” and “Profile” of matrices affect convergence of preconditioned iterative solvers.

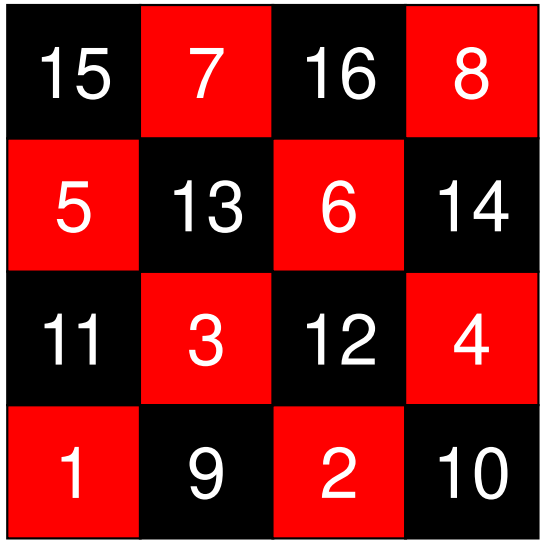
β_i

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



Multicoloring (MC), Multicolor Ordering





- Meshes in same color are independent, and renumbered according to the color ID.
  - Red-Black: MC with 2 colors
  - More colors needed for complicated geometries
- Parallel operations are possible for meshes in same color.
- A mesh and its neighboring meshes belong to different colors.

Fundamental Algorithm of MC Method

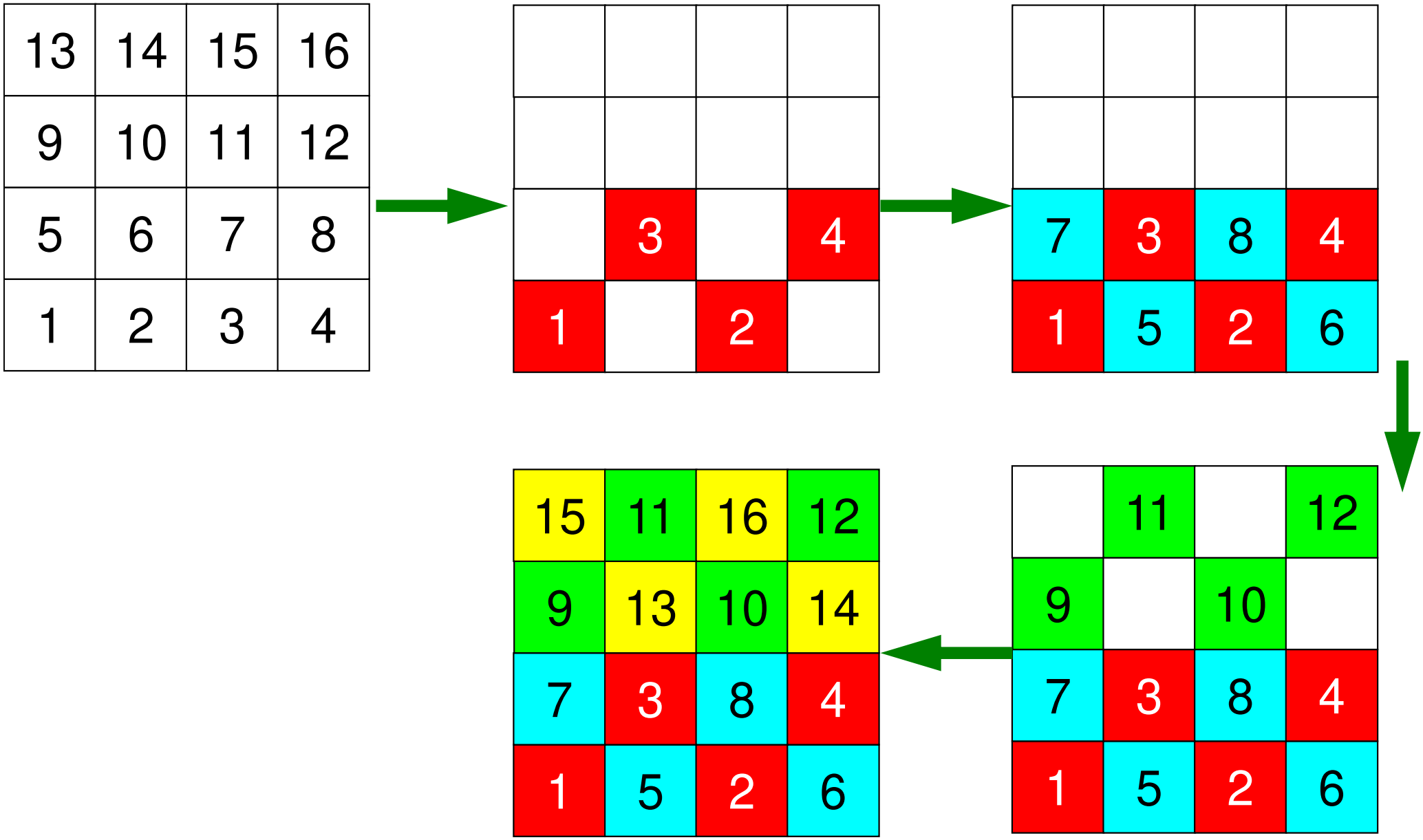
① m= mesh# / color#

② Color “m” independent meshes in ascending orders according to initial mesh ID, then proceed to the next “color”

③ Repeat ②, until every mesh is colored

④ Renumber meshes in ascending orders according to color ID. In each color, numbering is in ascending orders according to initial mesh ID.

MC with 4 Colors



Modified MC Method

① ONE mesh with minimum value of “degree” is set to “NEW mesh ID= 1”, “Color ID= 1”, and “counter for color number” is 1.

② Define “*ITEMcou= ICELTOT/NCOLORtot*”, where *ITEMcou* is maximum number of meshes in each color.

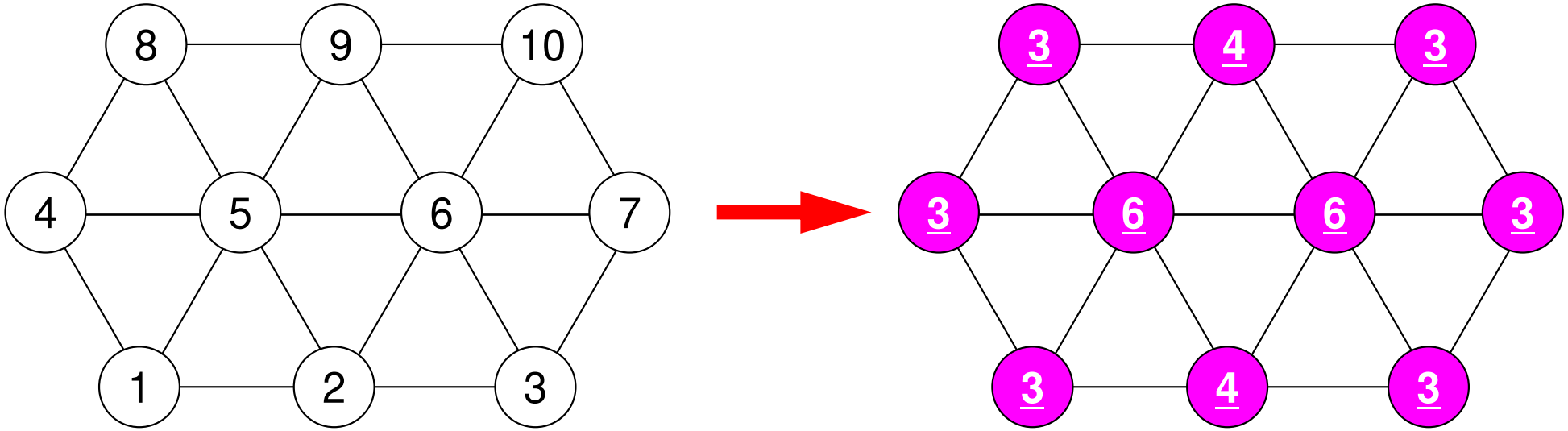
③ Color *ITEMcou* independent meshes in ascending order according to initial mesh ID.

④ If *ITEMcou* meshes are colored, or no more independent meshes do not exist, add “1” to the “counter for color number”, and proceed to the next color.

⑤ Repeat ③ and ④, until all meshes have been colored.

⑥ “Final counter for color” is *NCOLORtotF*. Renumber meshes in ascending orders according to color ID. In each color, numbering is in ascending orders according to initial mesh ID. In each color, new numbering of meshes is continuous.

“Degree”: Number of vertices adjacent to each vertex



Modified MC Method

① ONE mesh with minimum value of “degree” is set to “NEW mesh ID= 1”, “Color ID= 1”, and “counter for color number” is 1.

② Define “*ITEMcou= ICELTOT/NCOLORtot*”, where *ITEMcou* is maximum number of meshes in each color.

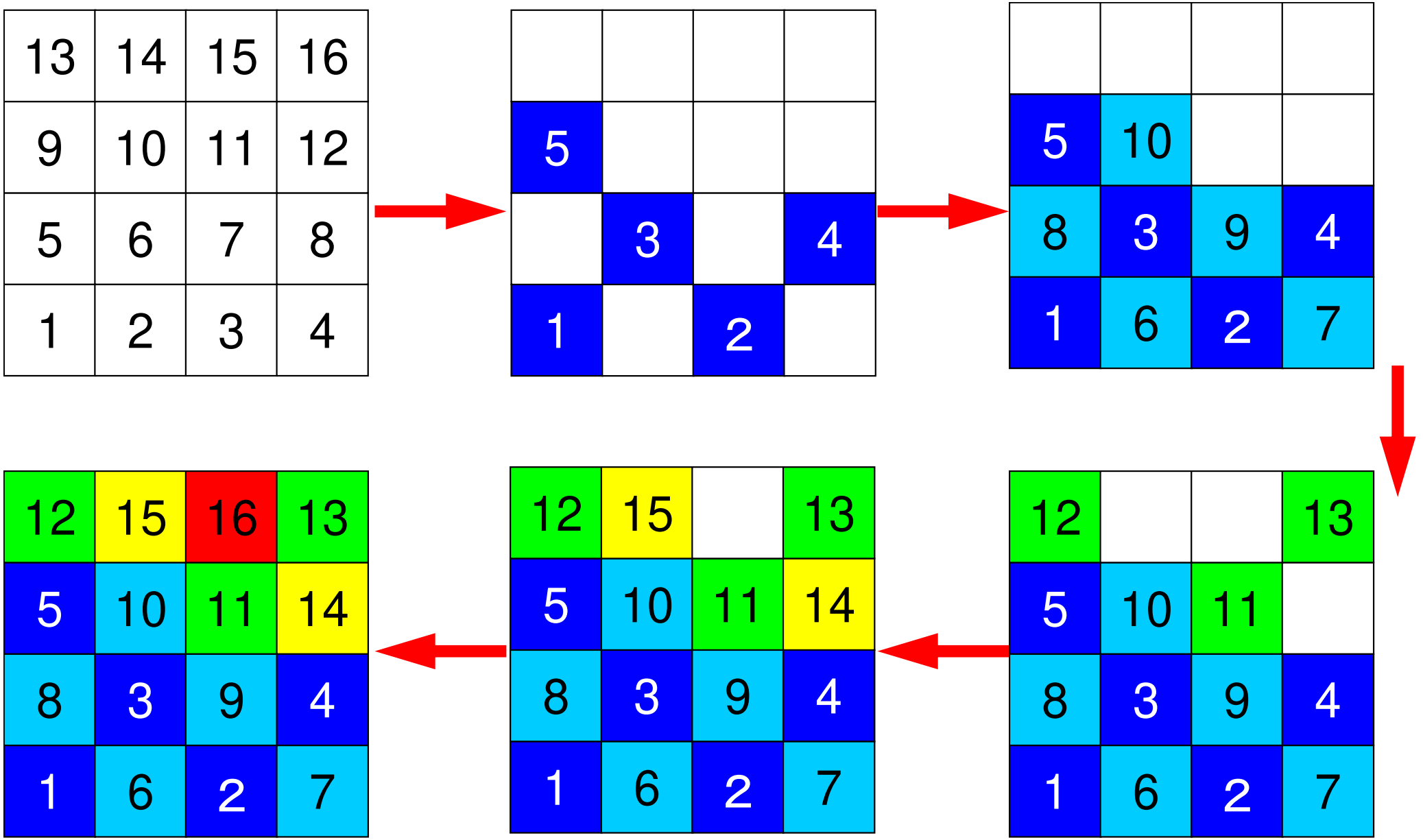
③ Color *ITEMcou* independent meshes in ascending order according to initial mesh ID.

④ If *ITEMcou* meshes are colored, or no more independent meshes do not exist, add “1” to the “counter for color number”, and proceed to the next color.

⑤ Repeat ③ and ④, until all meshes have been colored.

⑥ “Final counter for color” is *NCOLORtotF*. Renumber meshes in ascending orders according to color ID. In each color, numbering is in ascending orders according to initial mesh ID. **In each color, new numbering of meshes is continuous.**

MC with 3 colors, finally 5 colors



Ordering/Reordering Method for Parallel Computing

- Multicoloring (MC)
  - Parallelism
  - Red-Black with 2 colors

- CM (Cuthill-McKee), RCM (Reverse Cuthill-McKee)
  - Reducing fill-in's, matrix bandwidth, profiles
  - Parallelism

Fundamental Algorithm for CM Method (Cuthill-McKee)

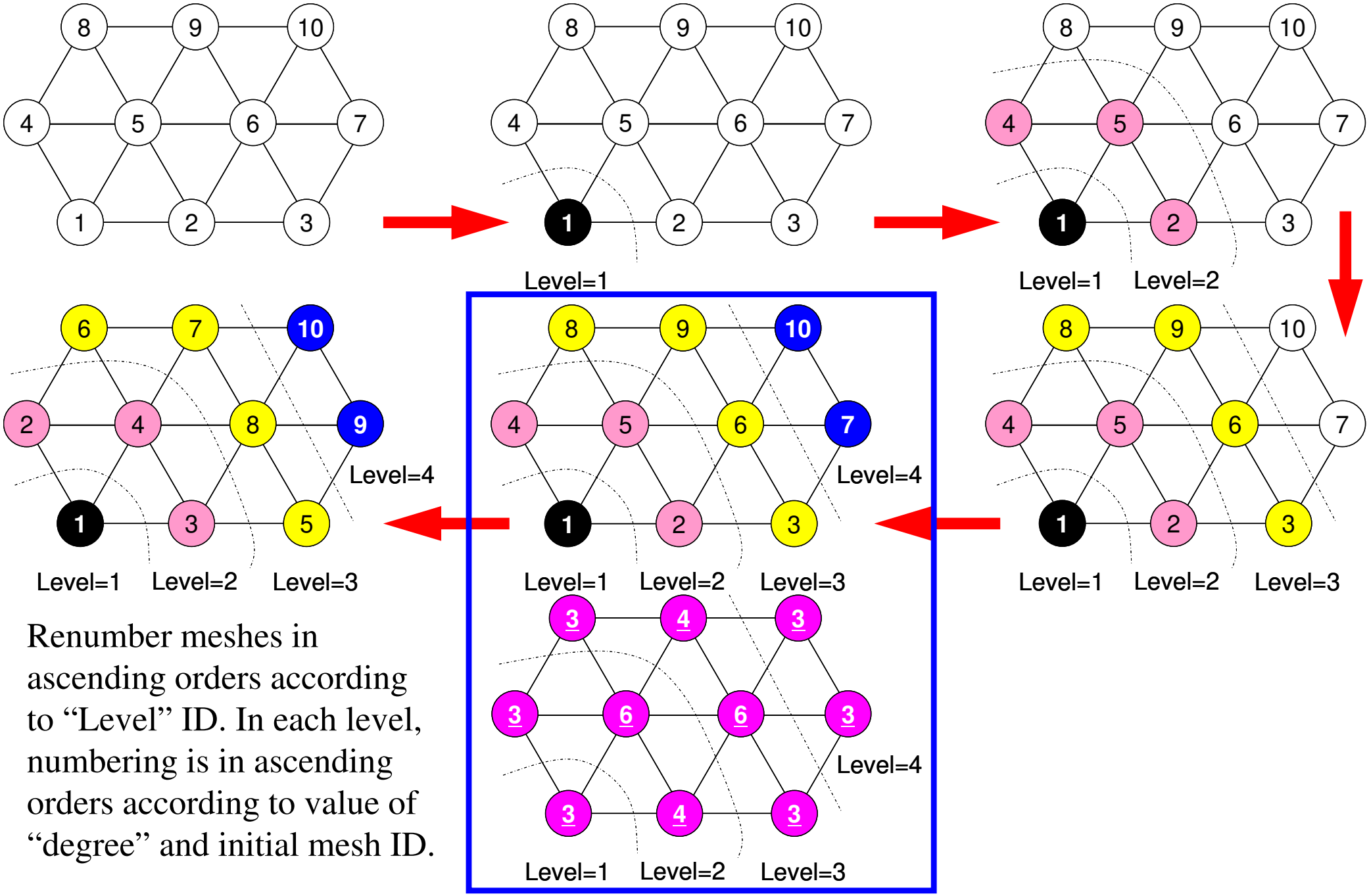
① ONE mesh with minimum value of “degree” is set to “Level=1”.

② Meshes adjacent to “Level=k-1” meshes are set to “Level=k”.

③ Repeat ②, until all meshes are flagged to “levels”

④ Renumber meshes in ascending orders according to “Level” ID. In each level, numbering is in ascending orders according to value of “degree” and initial mesh ID. In each level, new numbering of meshes is continuous.

Example of CM Method

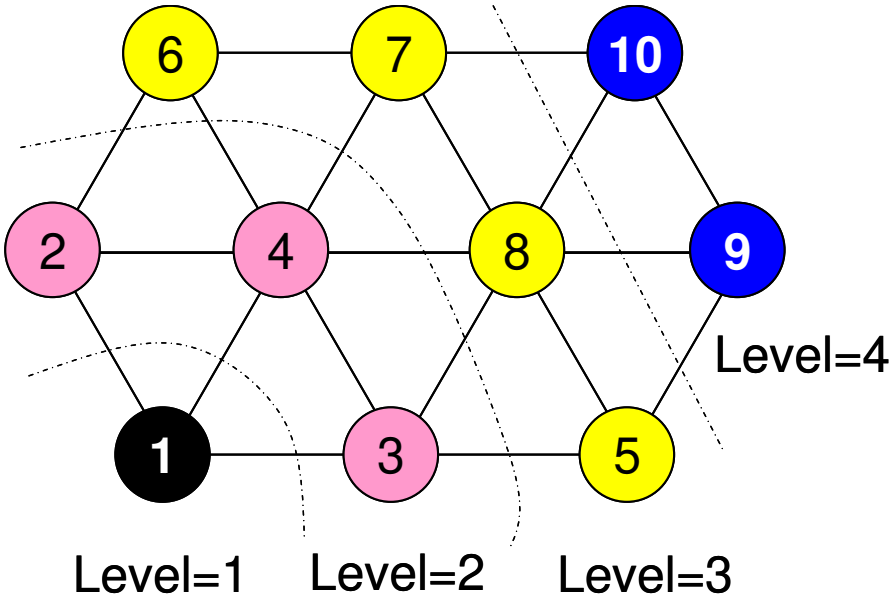


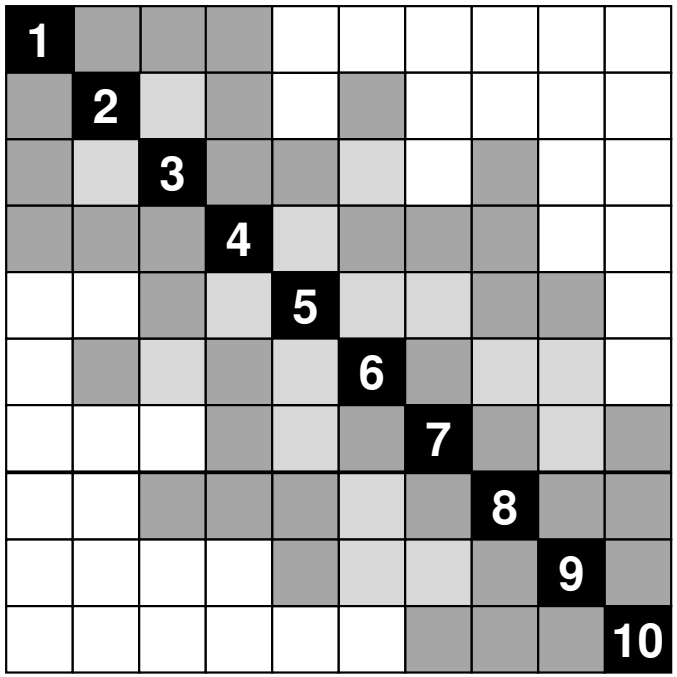
Renumber meshes in ascending orders according to "Level" ID. In each level, numbering is in ascending orders according to value of "degree" and initial mesh ID.

RCM: Reverse Cuthill-McKee

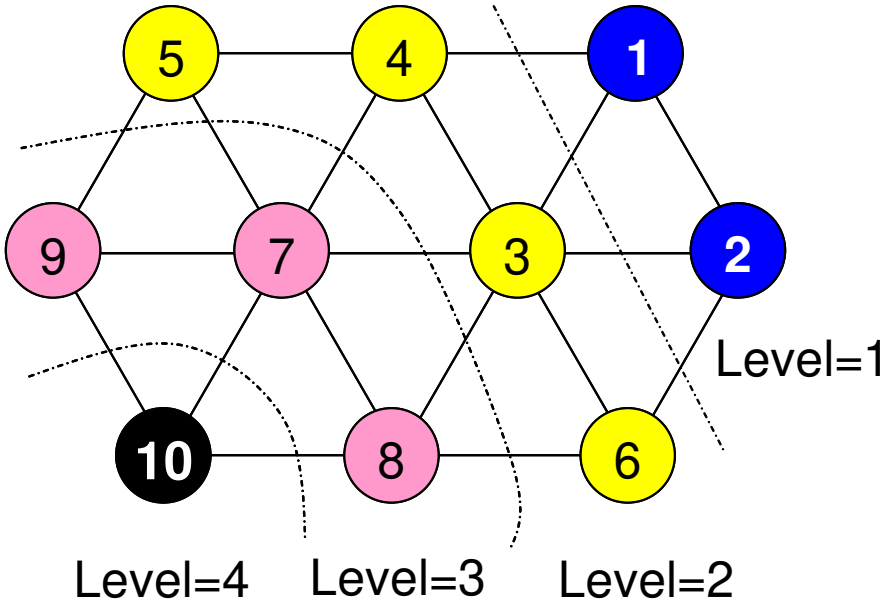
- Do operations for “CM” method
  - Calculate “degree” at each mesh
  - Flag “level k (1,2,...)” to meshes
  - Repeat processes, final renumbering
- Renumbering Again
  - Renumber meshes reordered by CM method in reverse order.
  - Fill-in’s (for full LU factorization) are fewer than CM

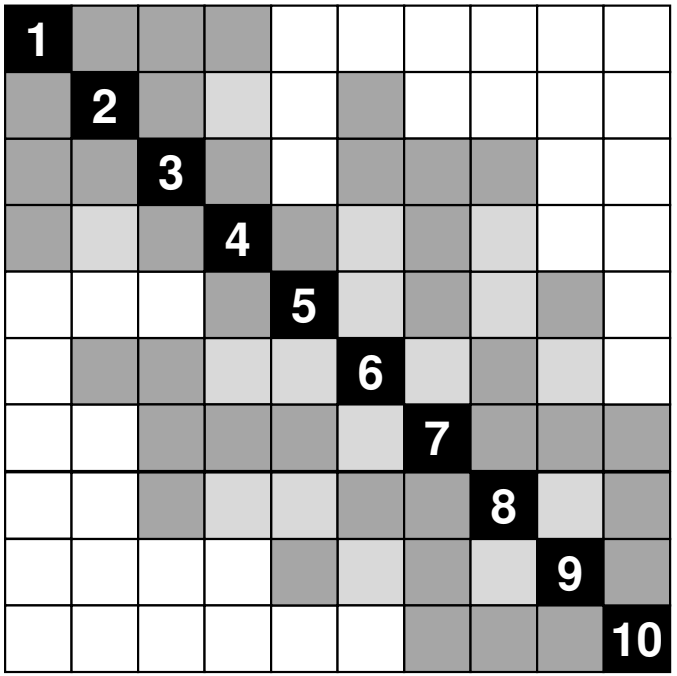
CM: 16 fill-in's





RCM: 14 fill-in's

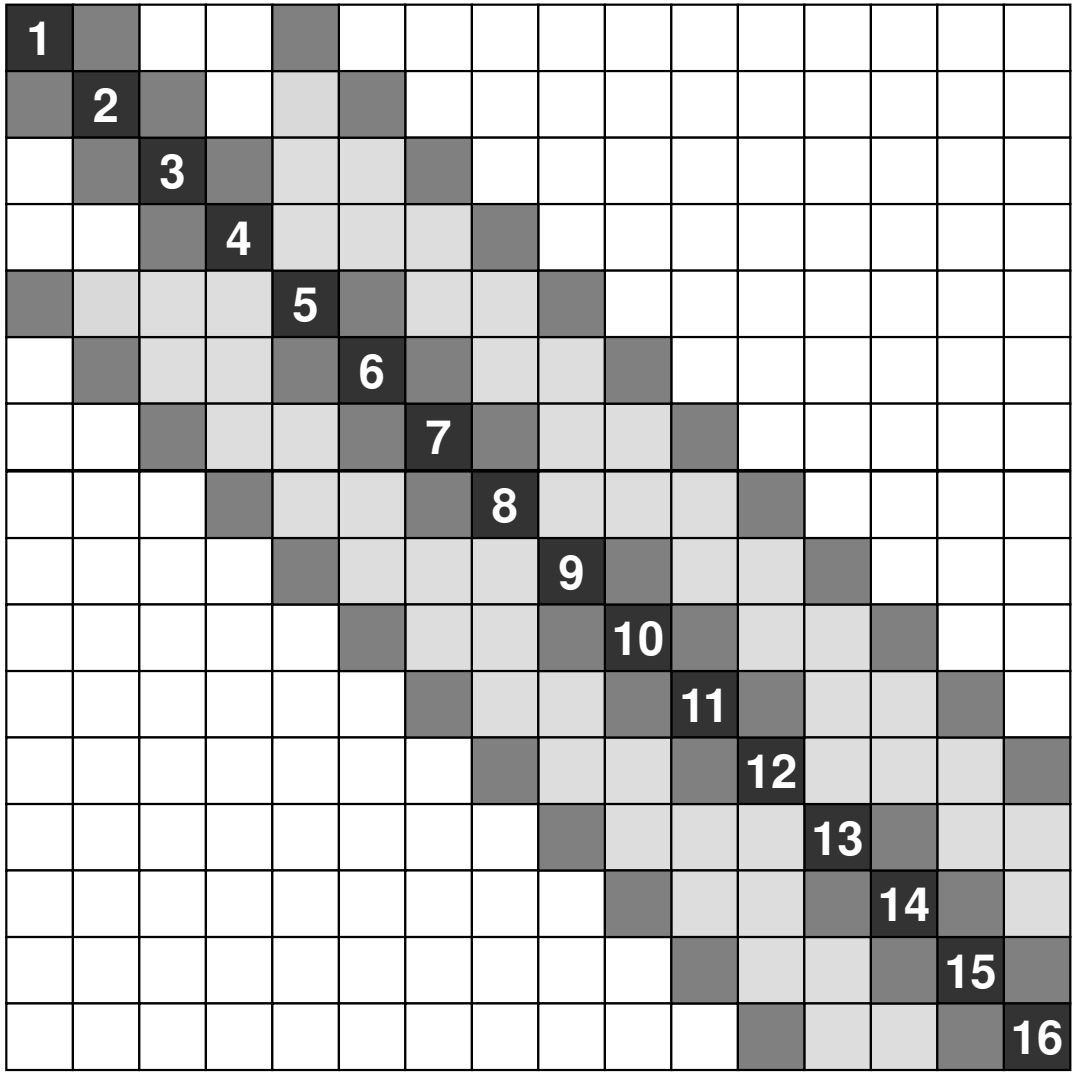




Initial Matrix

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

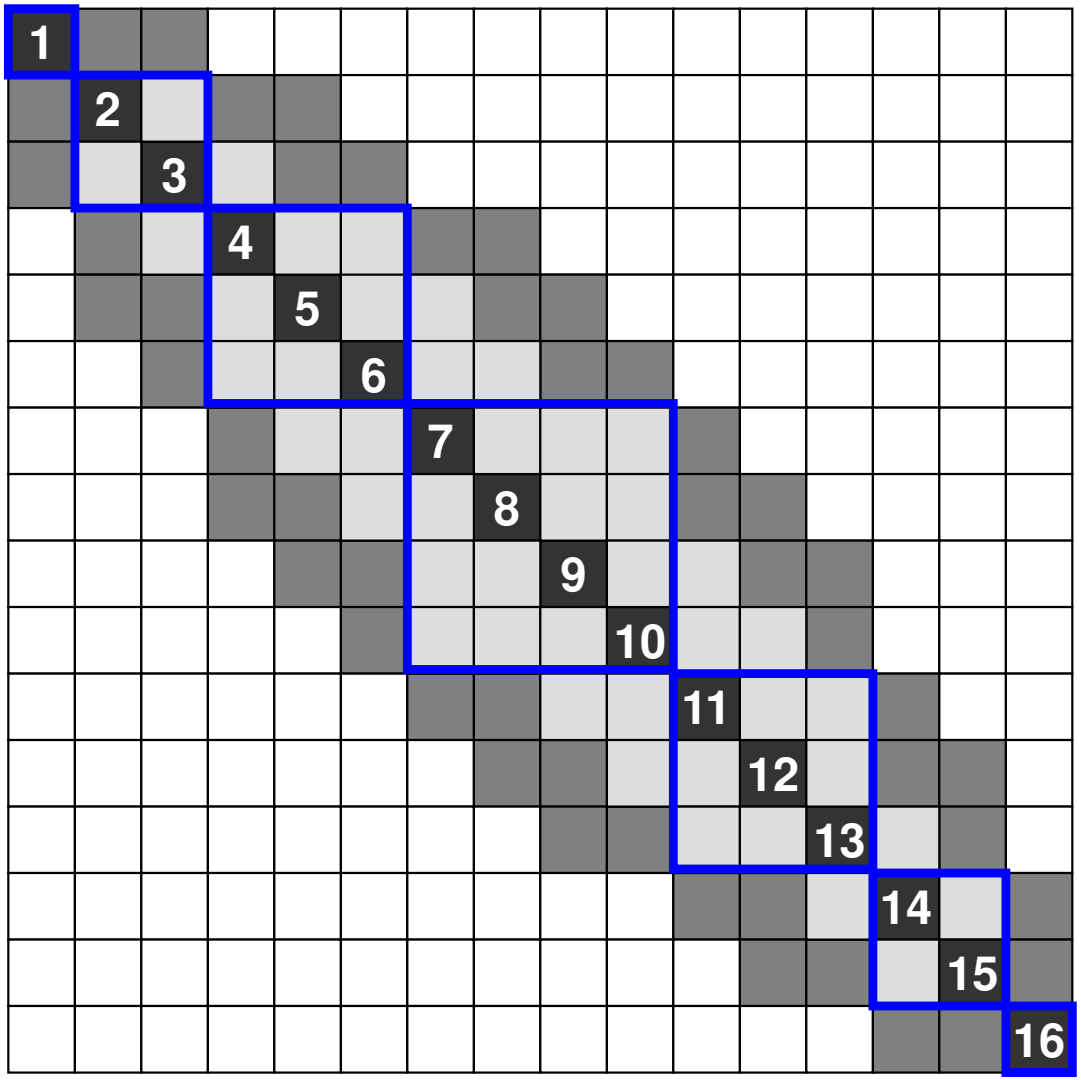
Bandwidth 4
Profile 51
Fill-in 54



Non-zero, Fill-in

CM

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



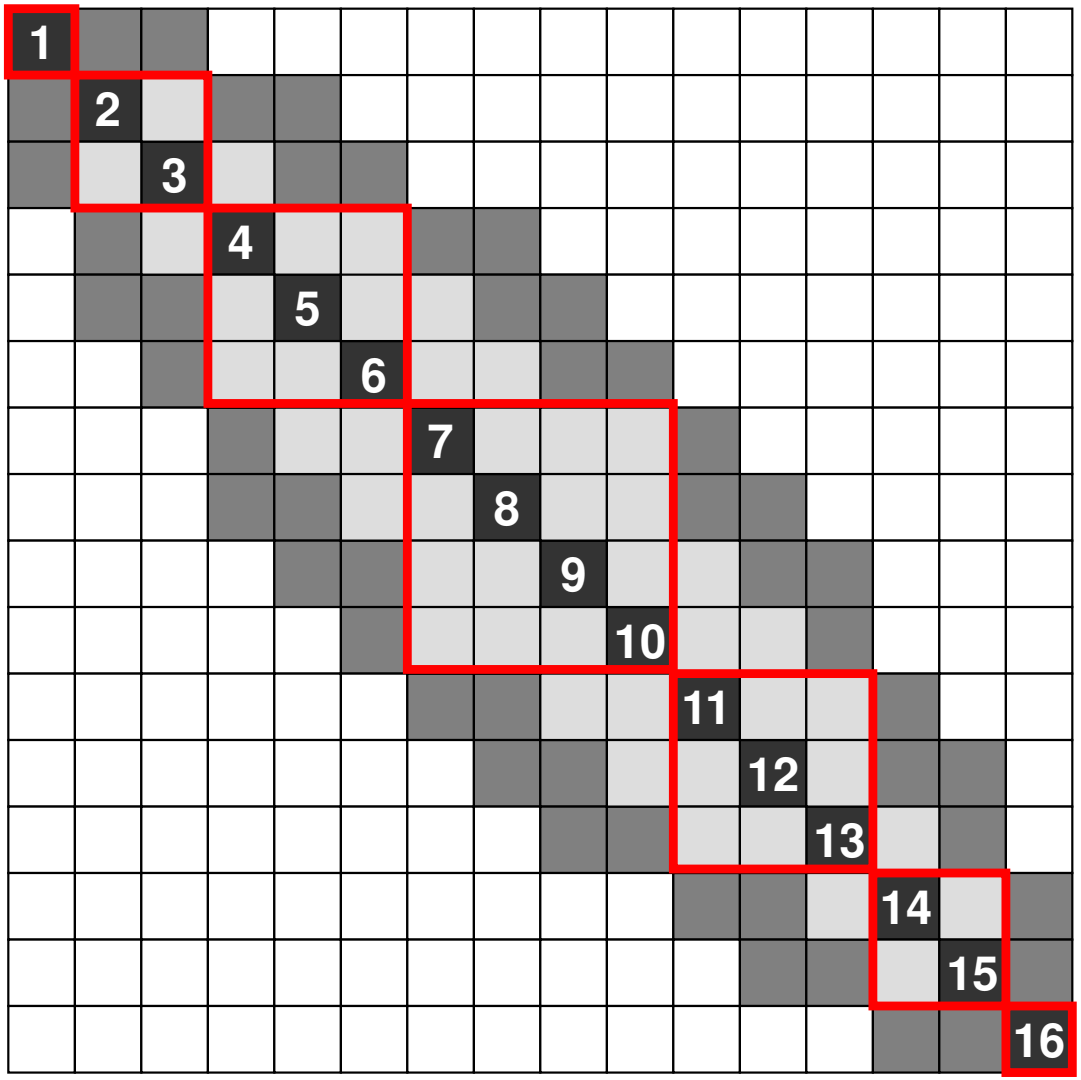
Bandwidth 4
Profile 46
Fill-in 44

Non-zero, Fill-in

RCM

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

Bandwidth 4
Profile 46
Fill-in 44



Non-zero, Fill-in

CM, RCM: Hyperline (i+j=const.)
3D: Hyperplane (i+j+k=cons.)

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

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

1,4	2,4	3,4	4,4
1,3	2,3	3,3	4,3
1,2	2,2	3,2	4,2
1,1	2,1	3,1	4,1

i+j=5

Modified CM Method for Parallel Computing

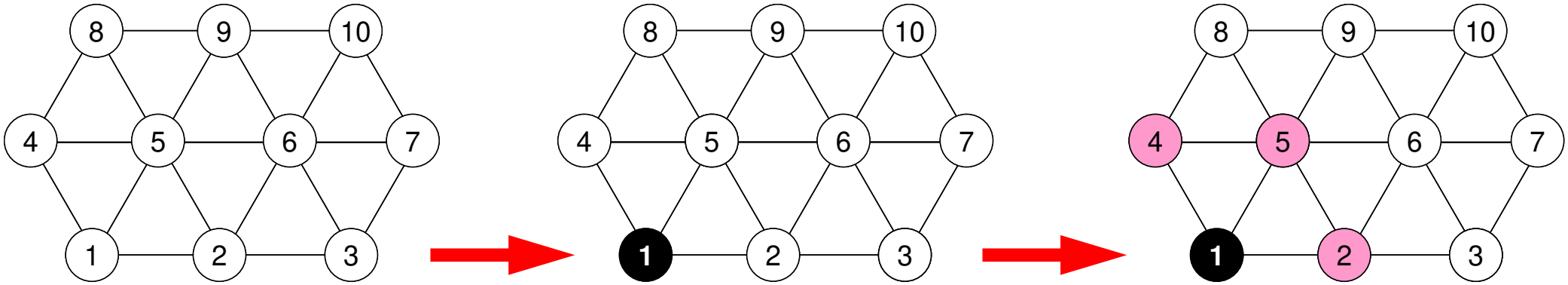
① ONE mesh with minimum value of “degree” is set to “Level=1”.

② Meshes adjacent to “Level=k-1” meshes are set to “Level=k”. In each level, meshes must be independent (not directly connected). If a dependent pair is found in same color, one mesh is removed (In current implementation, a mesh found later is removed).

③ Repeat ②, until all meshes are flagged to “levels”

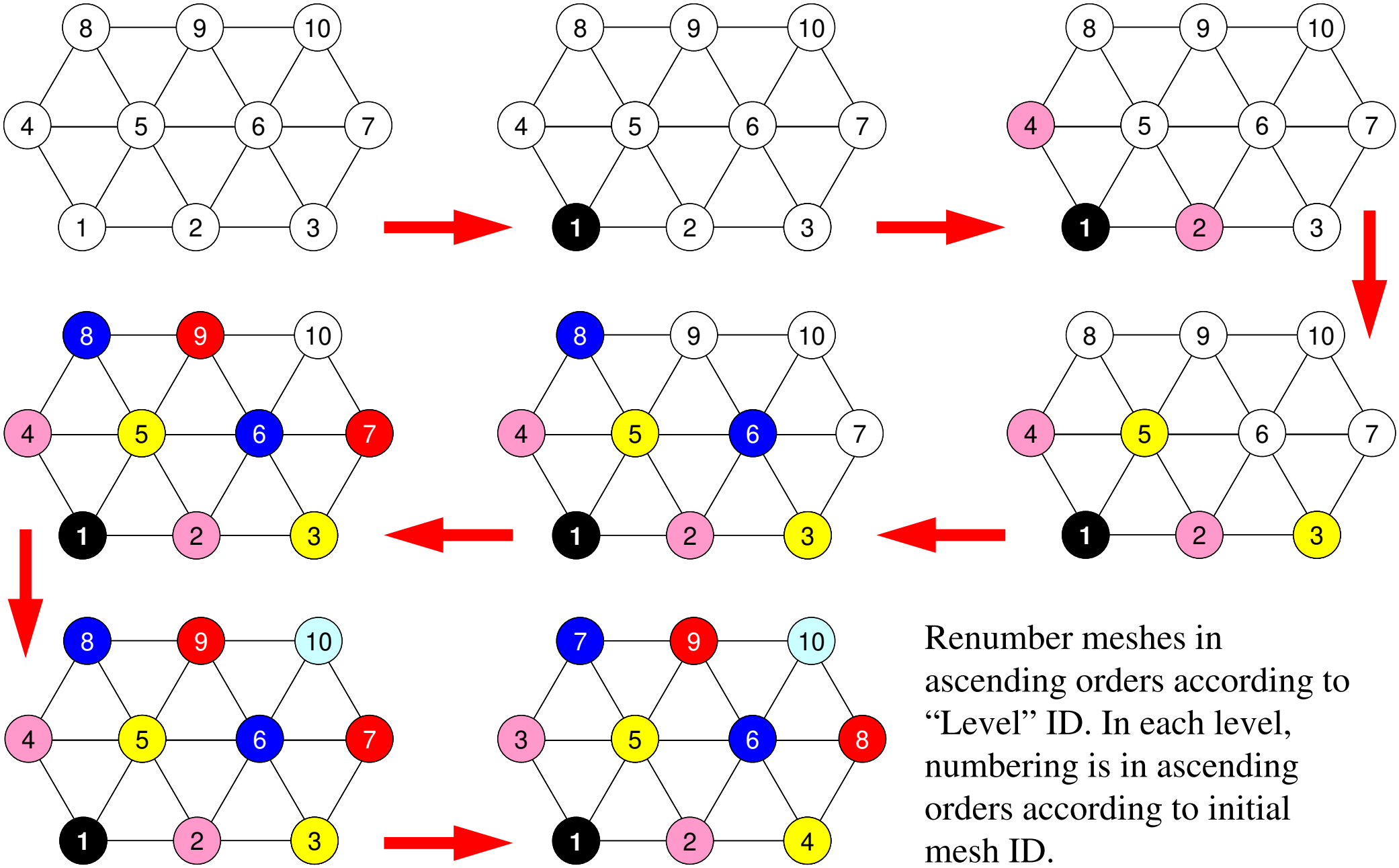
④ Renumber meshes in ascending orders according to “Level” ID. In each level, numbering is in ascending orders according to initial mesh ID. In each level, new numbering of meshes is continuous.

Modified CM Method



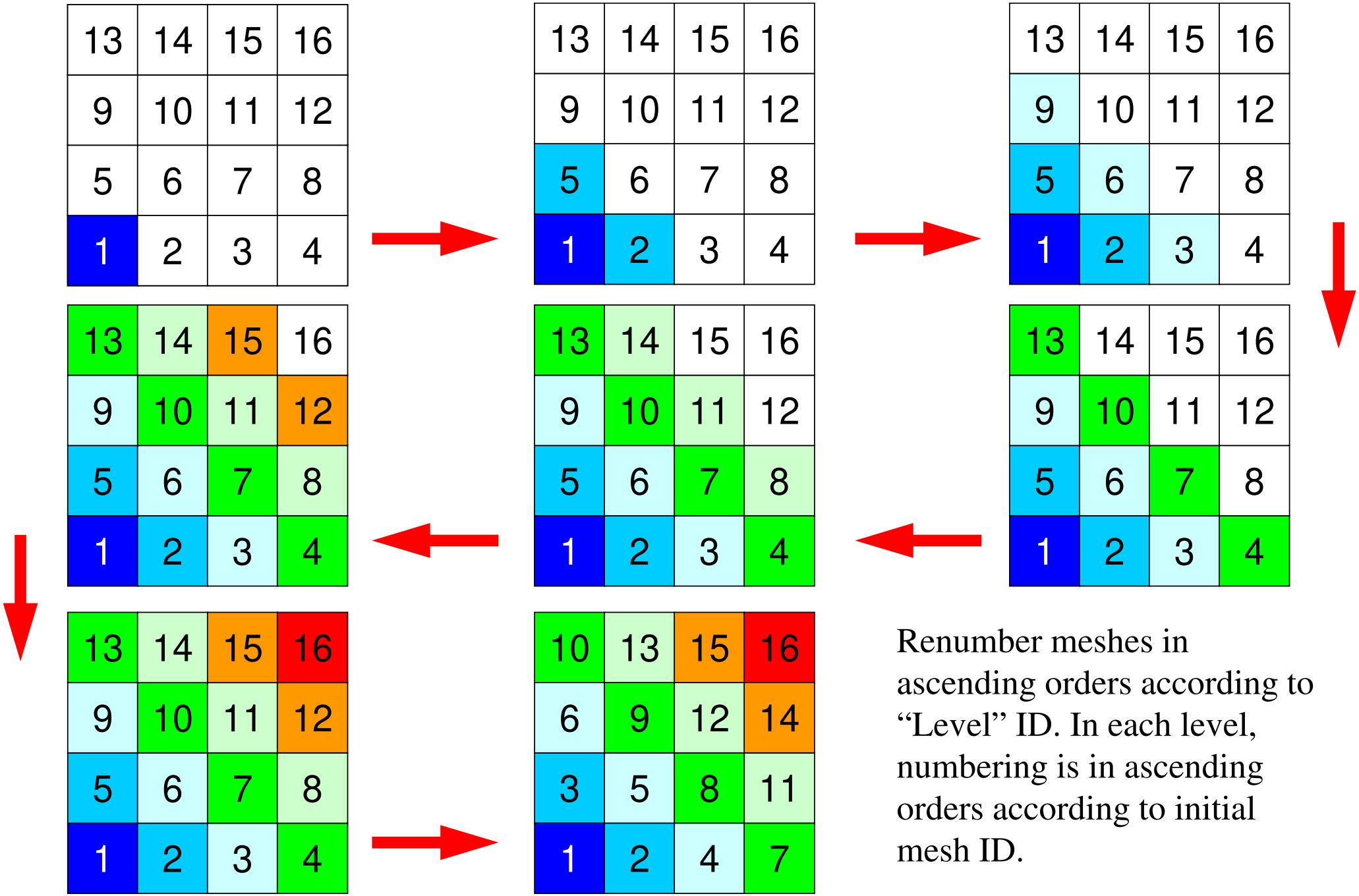
In each level, meshes are independent

Modified CM Method



Renumber meshes in ascending orders according to "Level" ID. In each level, numbering is in ascending orders according to initial mesh ID.

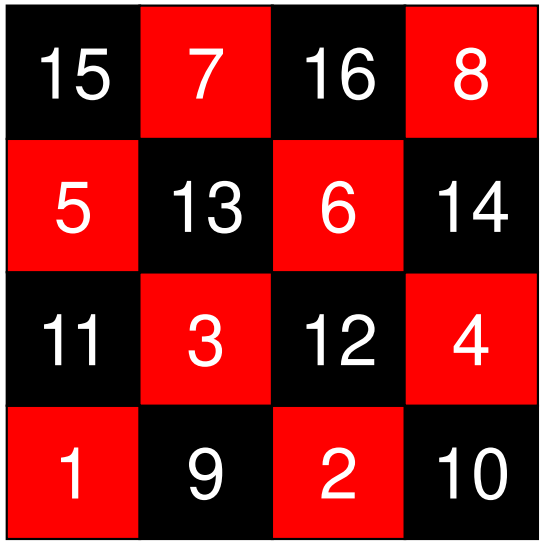
Modified CM Method

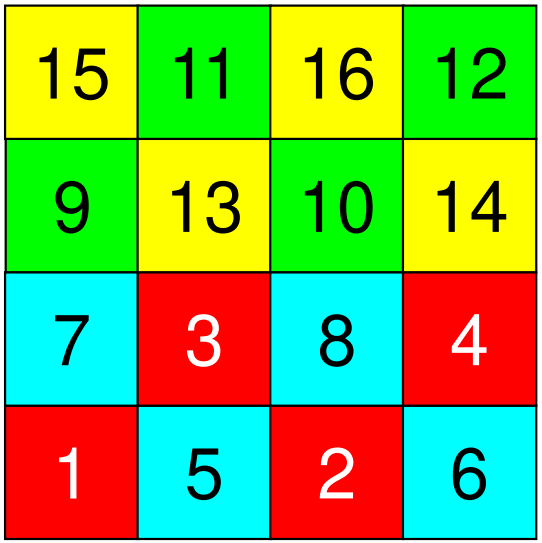


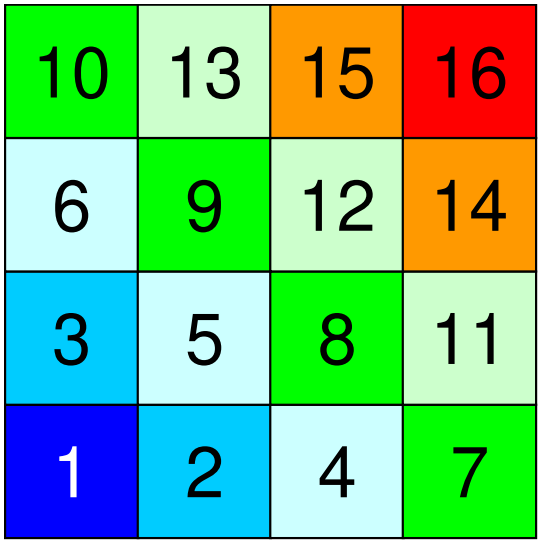
Renumber meshes in ascending orders according to "Level" ID. In each level, numbering is in ascending orders according to initial mesh ID.

MC and CM/RCM

- In CM/RCM, sequence of computations, and dependency between levels (color) are also considered.

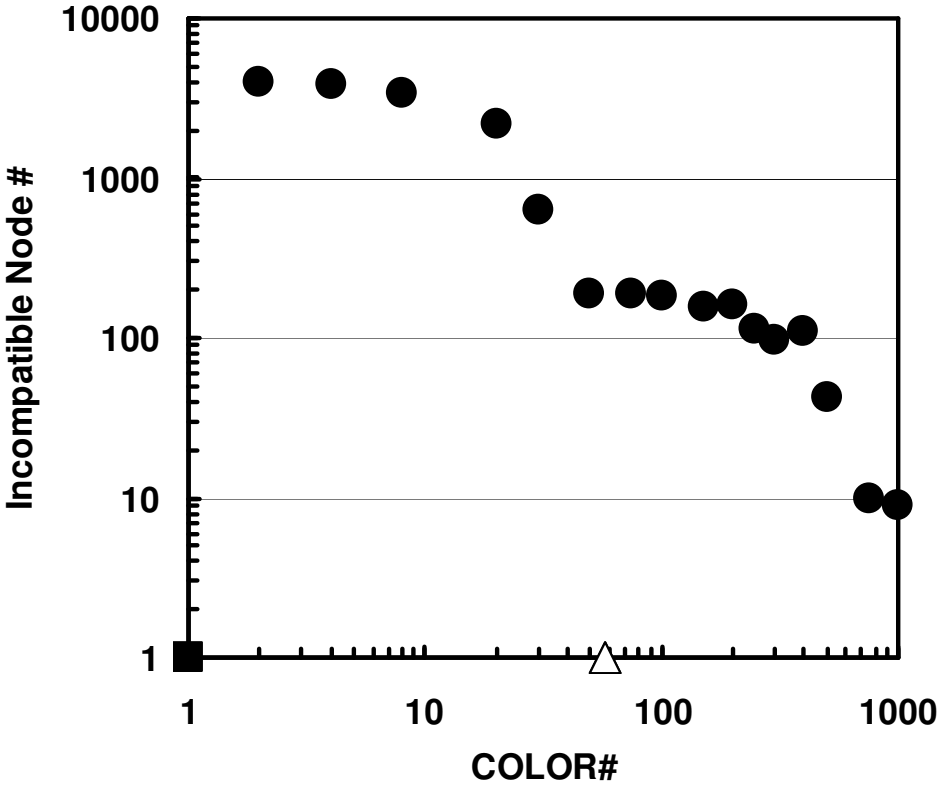


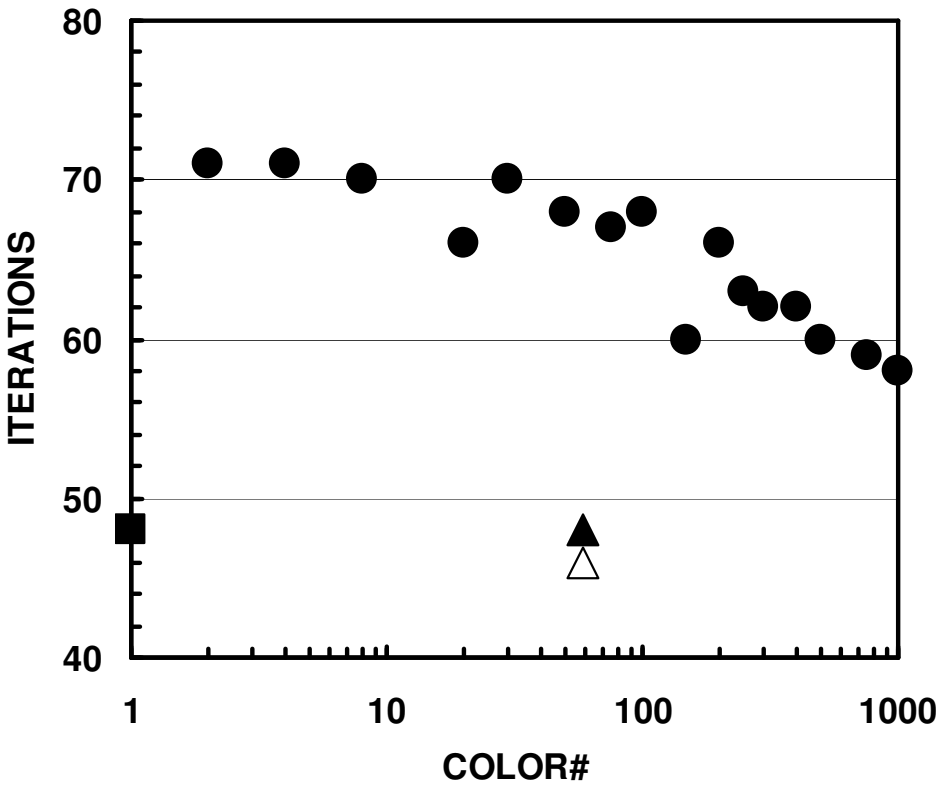




- Remedy for Data Dependency
- **Ordering/Reordering**
  - Red-Black, Multicoloring (MC)
  - Cuthill-McKee (CM), Reverse-CM (RCM)
  - **Reordering and Convergence**
- Implementation
- ICCG with Reordering

Effect of Color Number on Convergence of ICCG





(20^3=8,000 meshe, EPSICCG=10^{-8})

(■:ICCG(L1), ●:ICCG-MC, ▲:ICCG-CM, △:ICCG-RCM)

Effect of Color Number on Convergence of ICCG

- Number of Elements: 20^3
- Red-Black ~ 4-Colors < Initial Numbering ~ CM, RCM

Initial Numbering	
Bandwidth	4
Profile	51
Fill-in	54

Red-Black	
Bandwidth	10
Profile	77
Fill-in	44

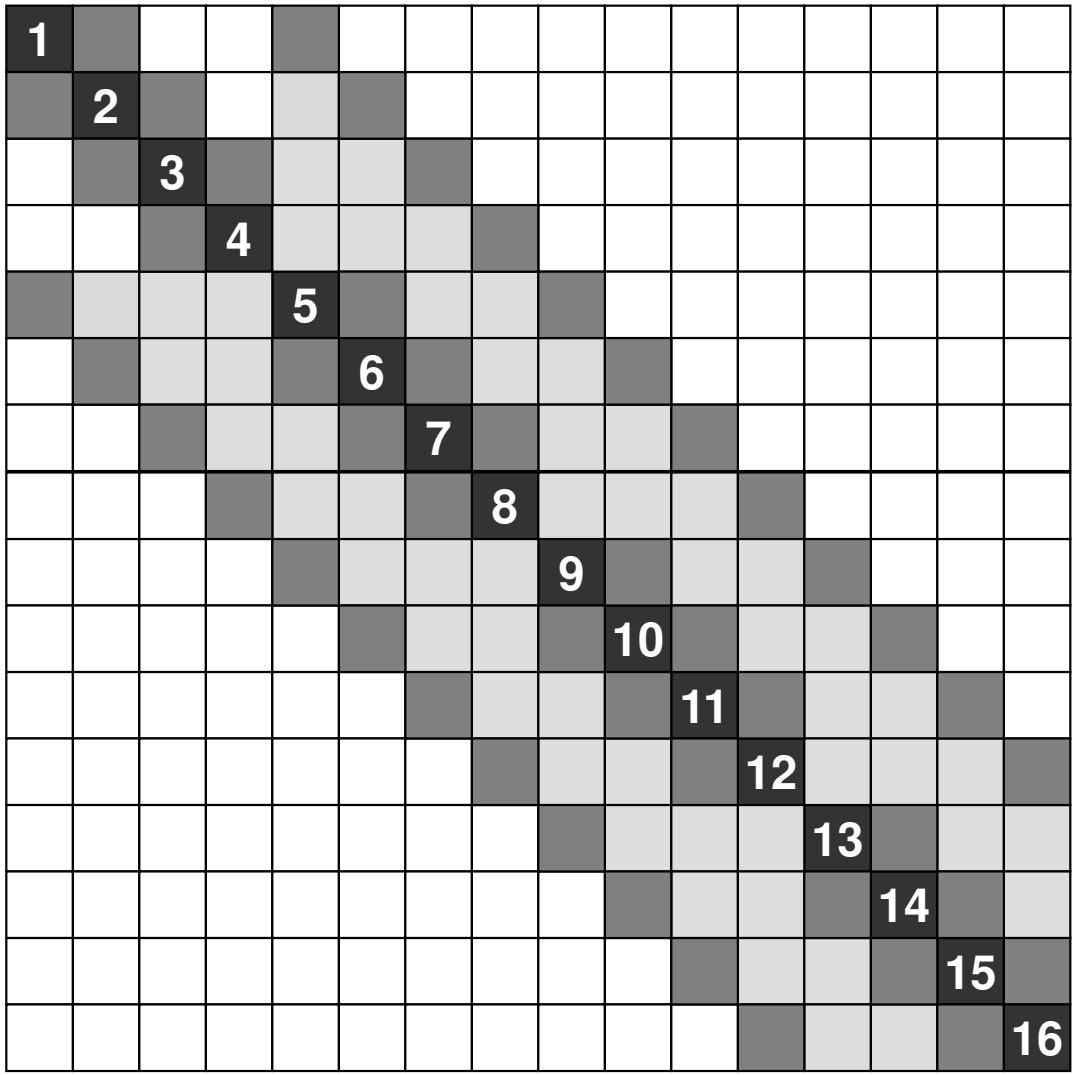
4-Colors	
Bandwidth	10
Profile	57
Fill-in	46

CM, RCM	
Bandwidth	4
Profile	46
Fill-in	44

Initial Matrix

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

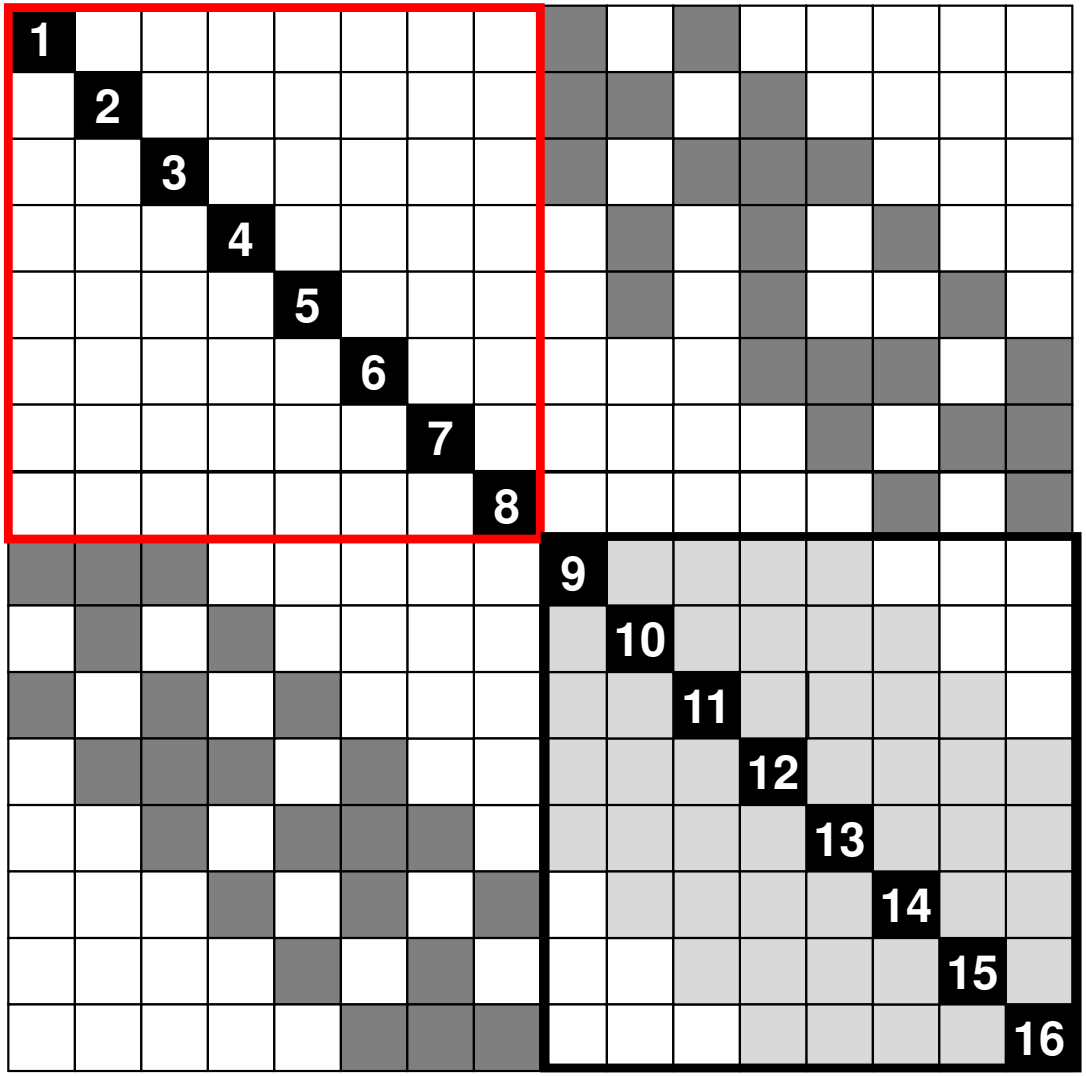
Bandwidth 4
Profile 51
Fill-in 54



Non-zero, Fill-in

Red-Black (2-Colors)

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

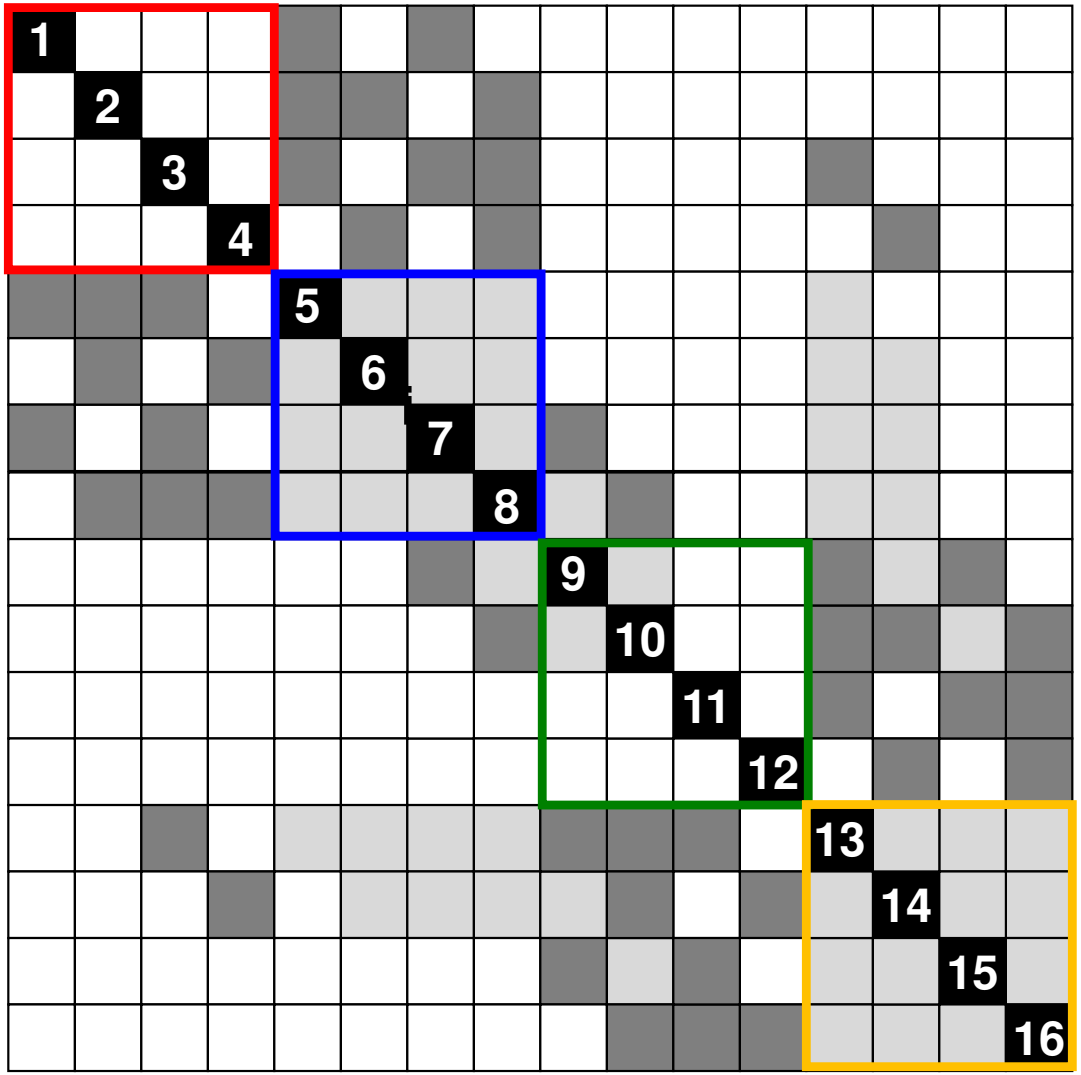


Bandwidth	10
Profile	77
Fill-in	44

Non-zero, Fill-in

MC (4-Colors)

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



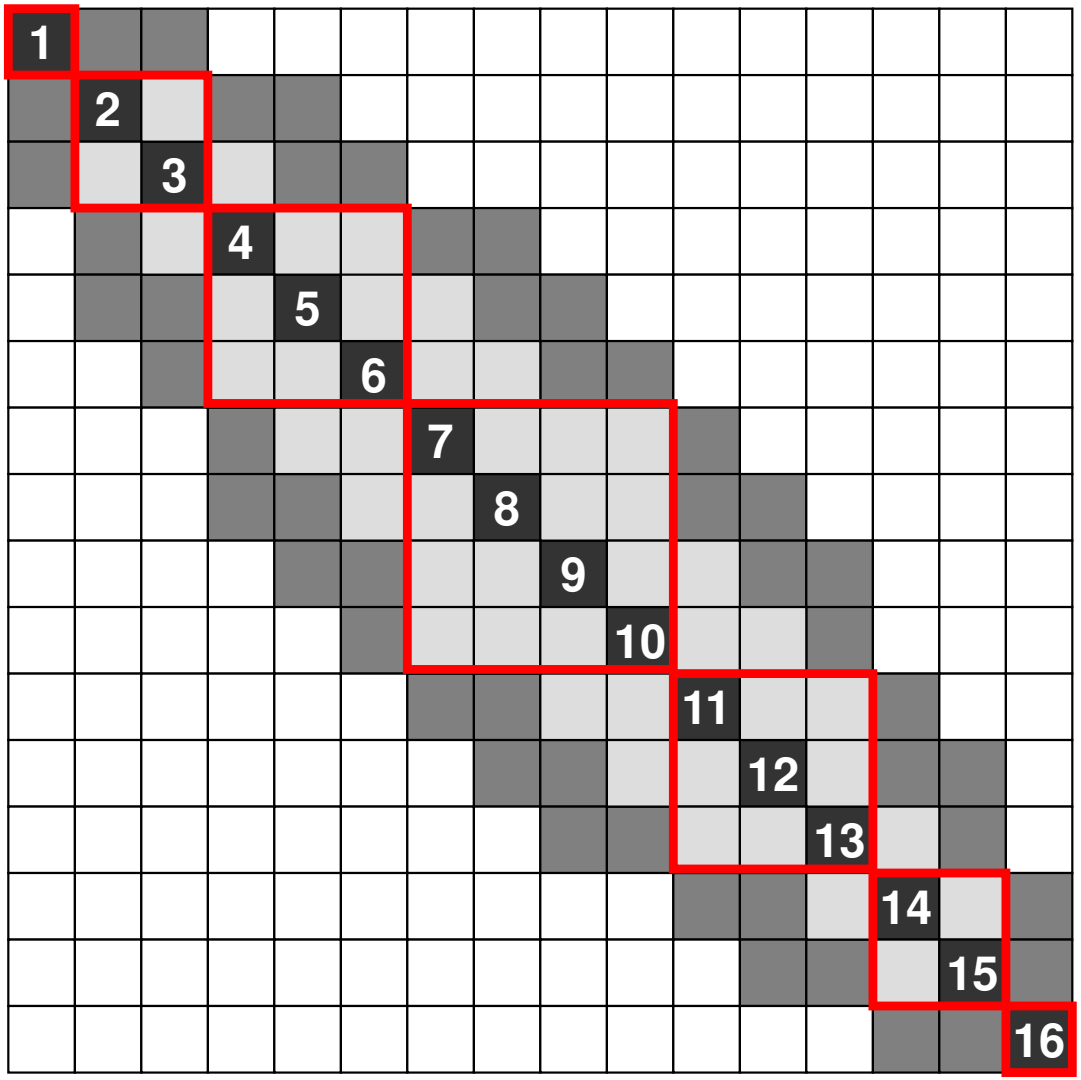
Bandwidth	10
Profile	57
Fill-in	46

Non-zero, Fill-in

RCM

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

Bandwidth 4
Profile 46
Fill-in 44

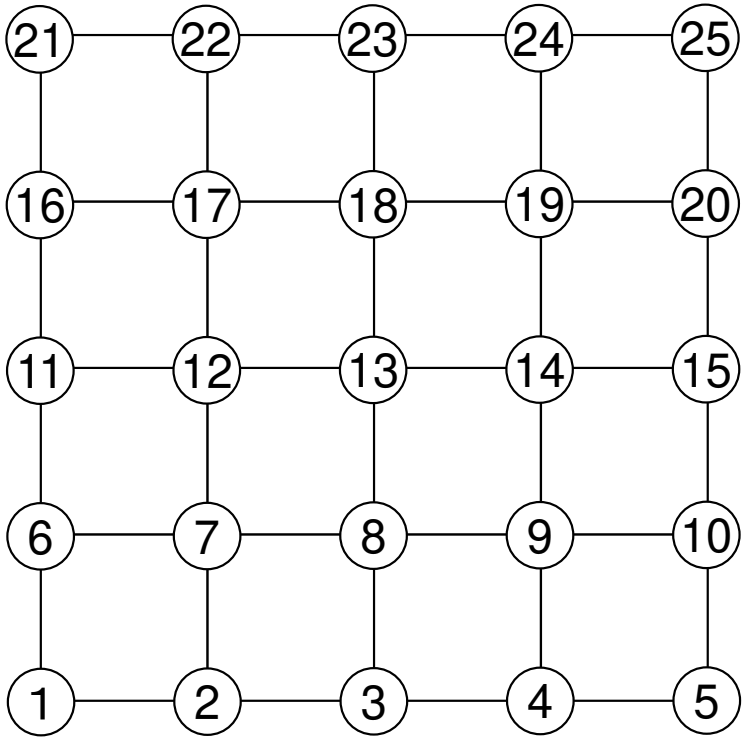


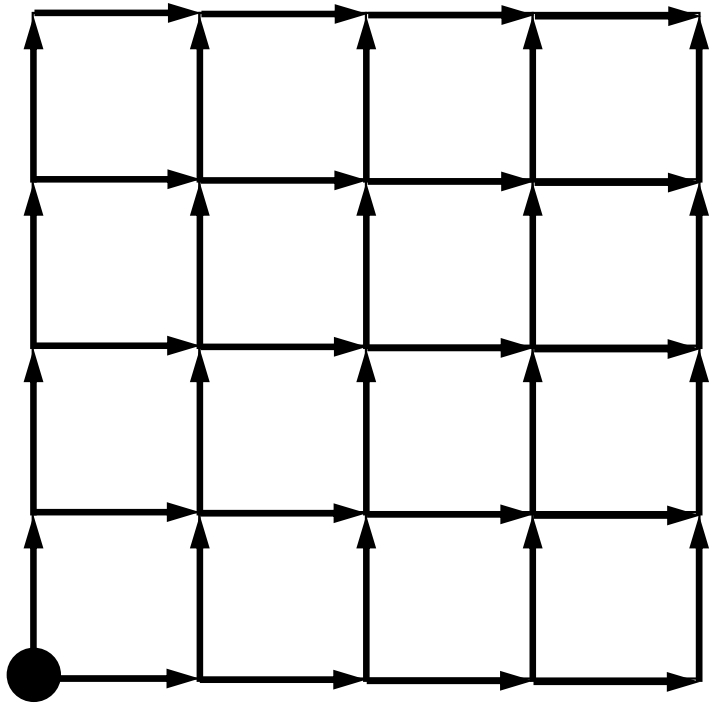
Non-zero, Fill-in

Color Number and Convergence Incompatible Nodes

Doi, S. (NEC) et al.

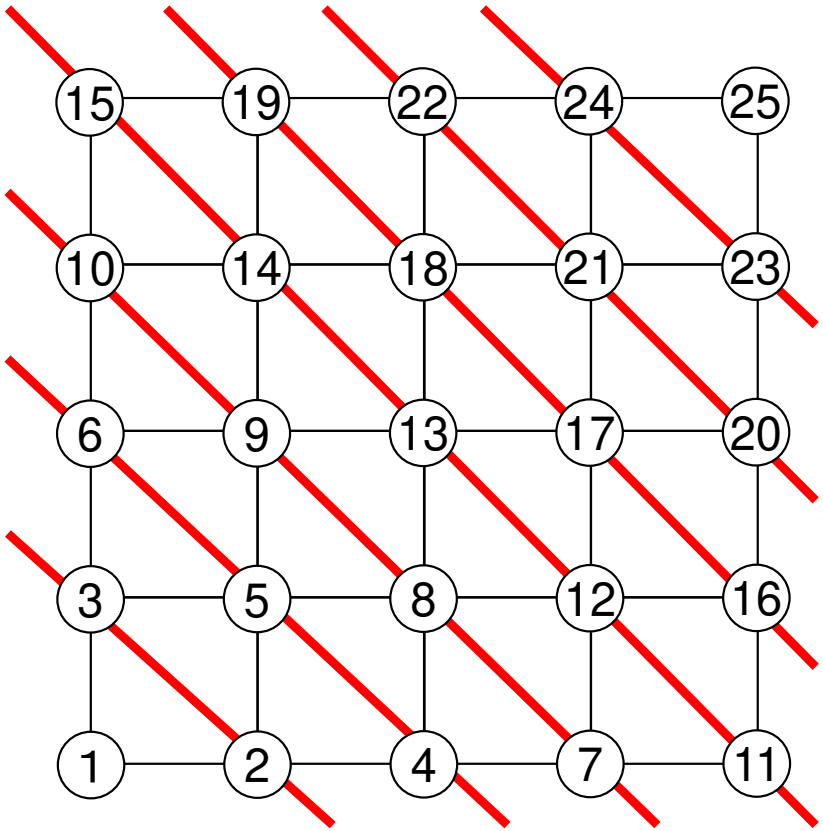
Propagation of effects in Forward Substitution

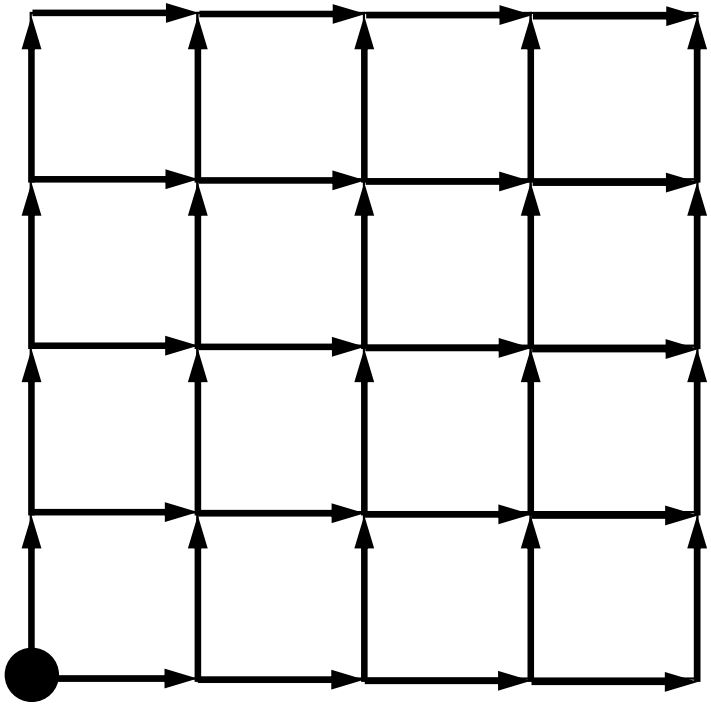




“Incompatible Nodes” not affected by other nodes. ID of such node is smaller than those of adjacent nodes. If we have smaller number of “incompatible nodes”, convergence is faster.

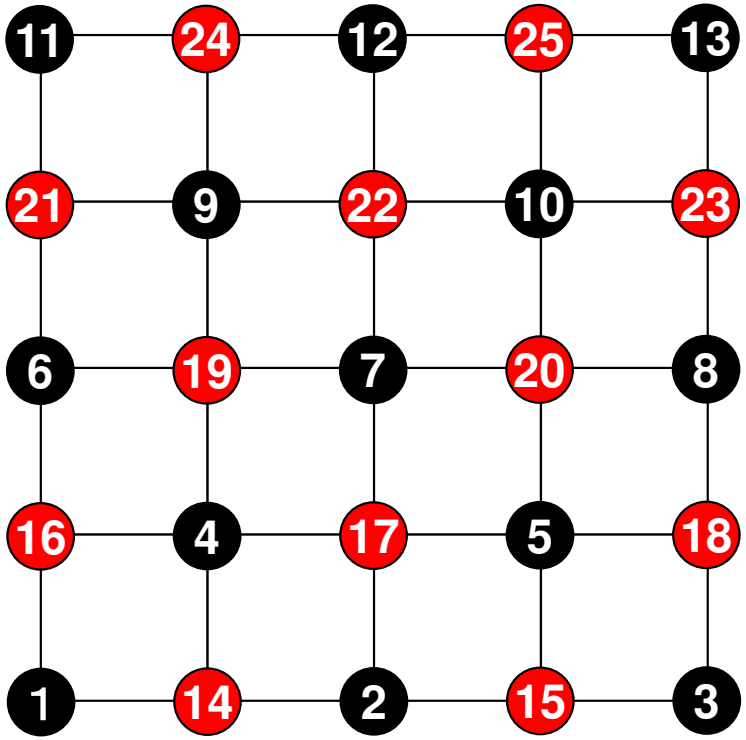
CM (Cuthill-McKee)

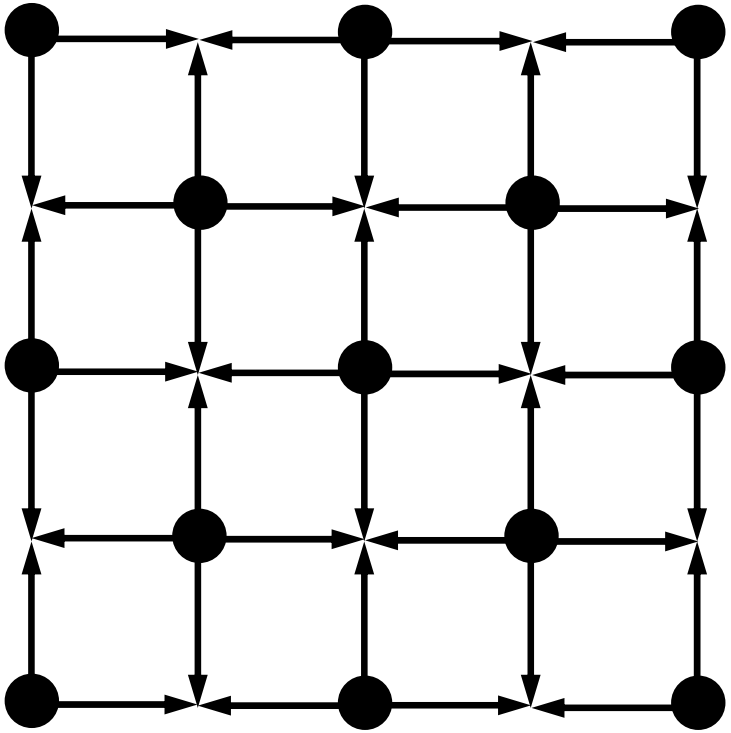




Red-Black

Higher parallelism, but many “incompatible nodes”
Slower convergence in ICCG, Gauss-Seidel etc.

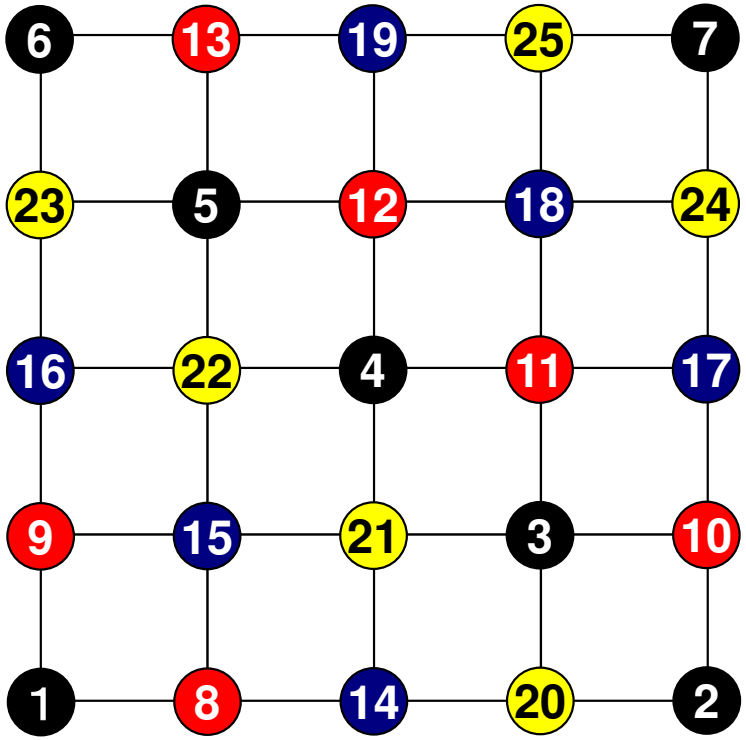


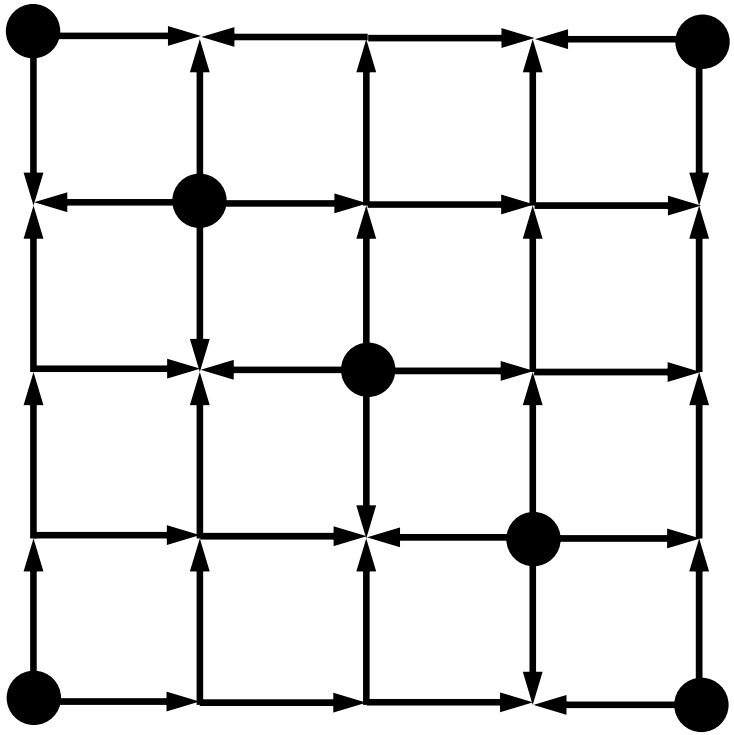


4-Colors

Still many “incompatible nodes”

Slower convergence in ICCG, Gauss-Seidel etc.





Generally speaking, convergence of ICCG is better for configurations with …

- More Colors

- Smaller Bandwidth
- Smaller Profile
- Fewer Fill-in’s
- Fewer Incompatible Nodes

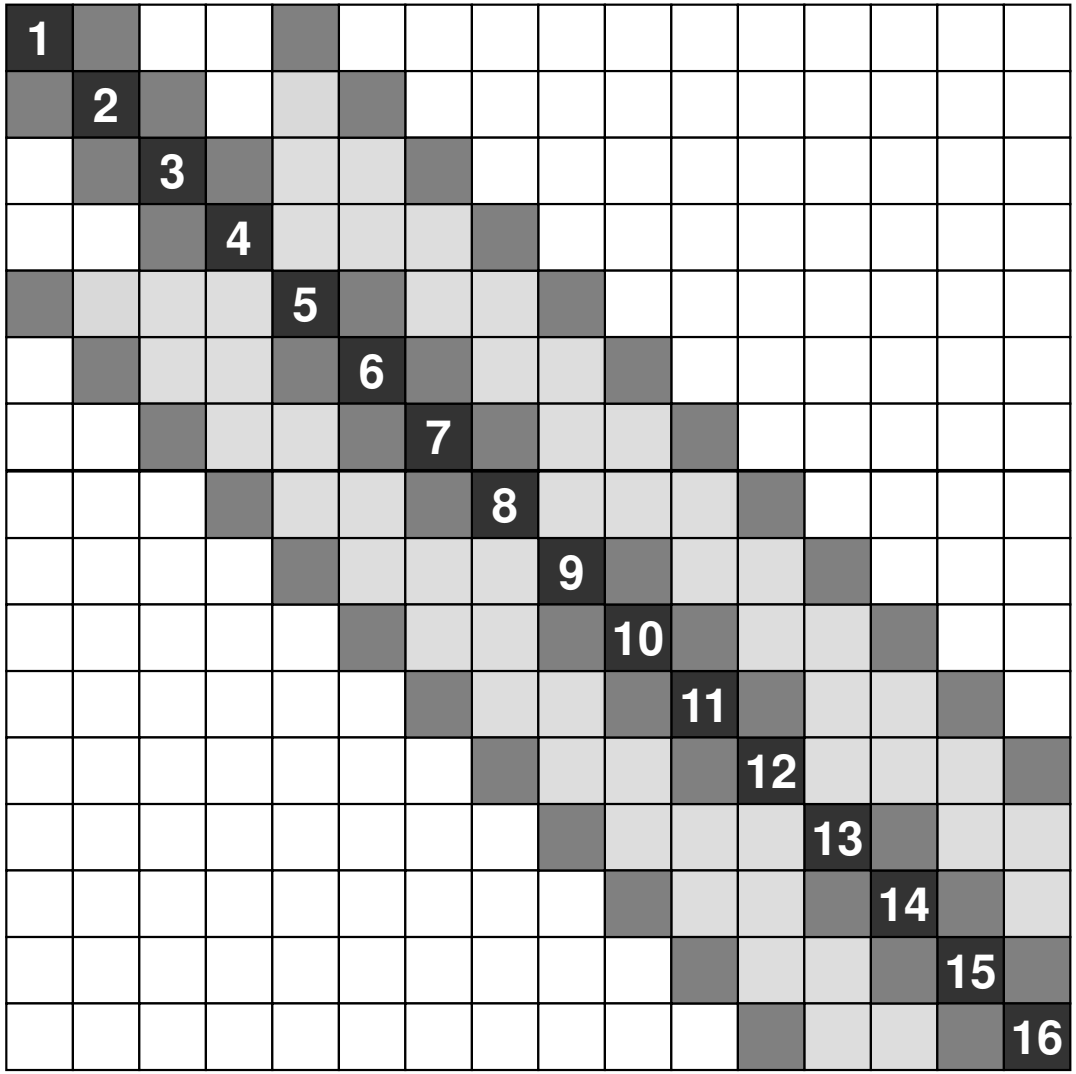
Generally speaking, convergence of ICCG is better for configurations with fewer fill-in's

- ICCG (IC(0)-CG) ignores fill-in's
- **More fill-in's are ignored for configurations (e.g. coloring) with more fill-in's**
  - **IC(0) for configurations with more fill-in's may be weaker than that with fewer fill-in's**
- **Distribution of fill-in's may affect the convergence**
  - **Initial Matrix**
  - **Red-Black (MC with 2-Colors)**

Initial Matrix

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

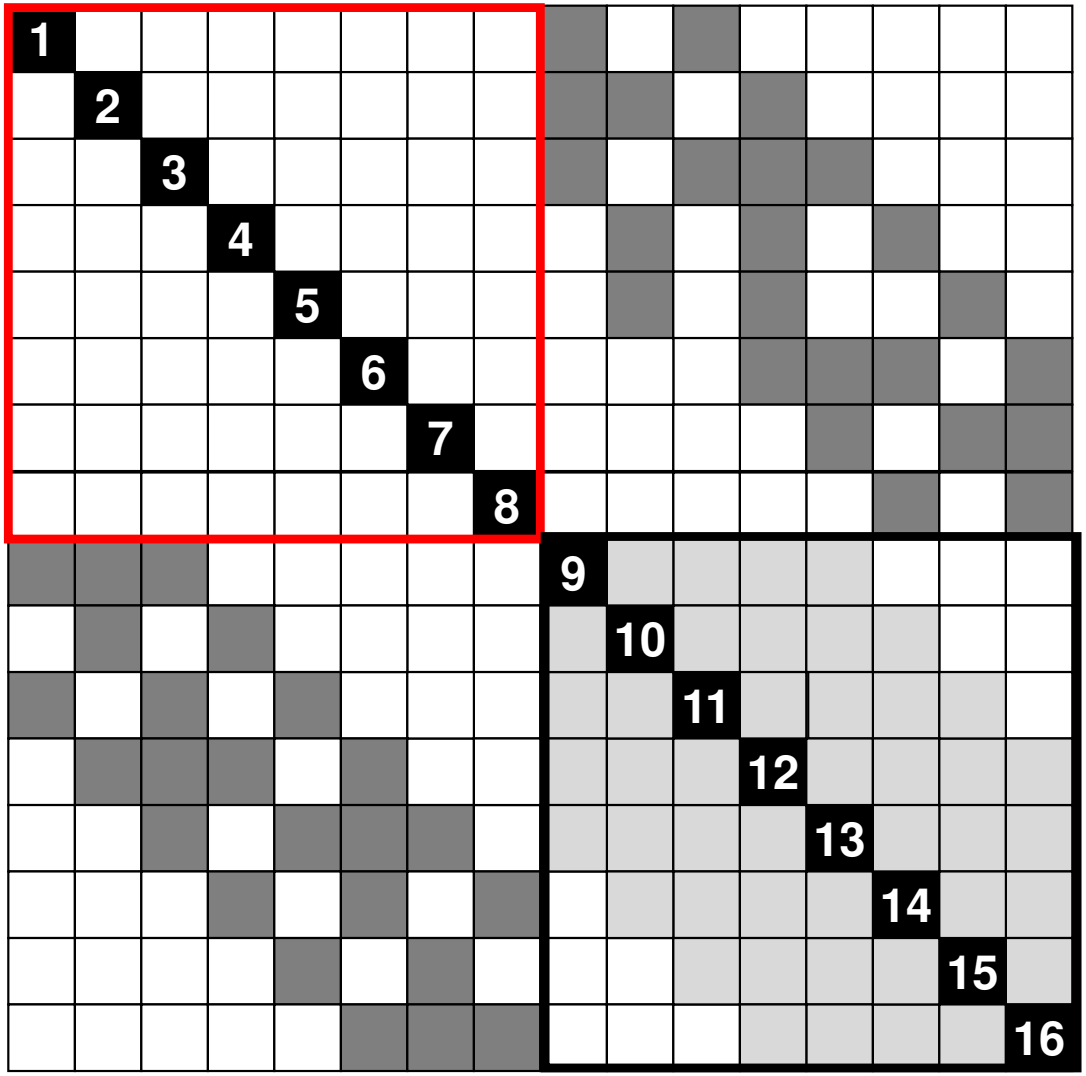
Bandwidth 4
Profile 51
Fill-in 54



Non-zero, Fill-in

Red-Black (2-Colors)

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



Bandwidth 10
Profile 77
Fill-in 44

Non-zero, Fill-in

Effect of Reordering/Color # on Convergence

- Other effects (e.g. B.C.) should be considered.
- It is difficult to provide very general remarks.
- e.g. RCM provides slightly faster convergence than CM, although parameters (Bandwidth, profile, fill-in's) are same.

Effect of Reordering

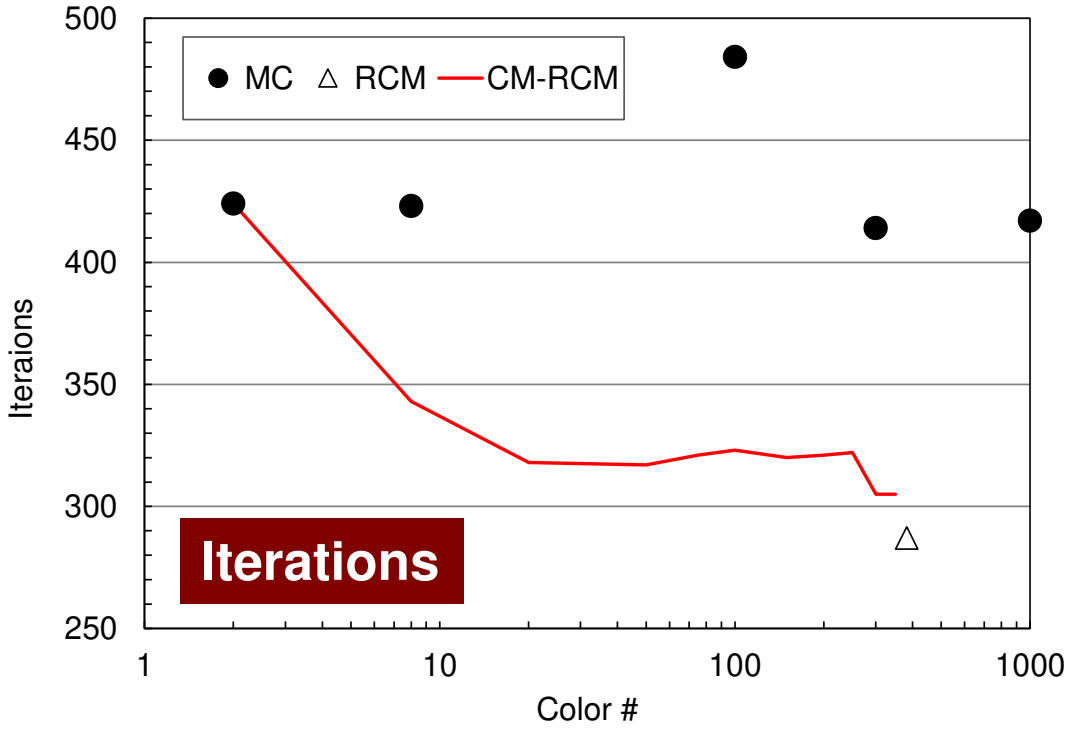
- Reordering changes sequence of matrix operations, and sometimes improves convergence.
  - Parallelism, Faster Convergence
- We need some kind of reordering for parallel ICCG on such simple meshes described in examples.

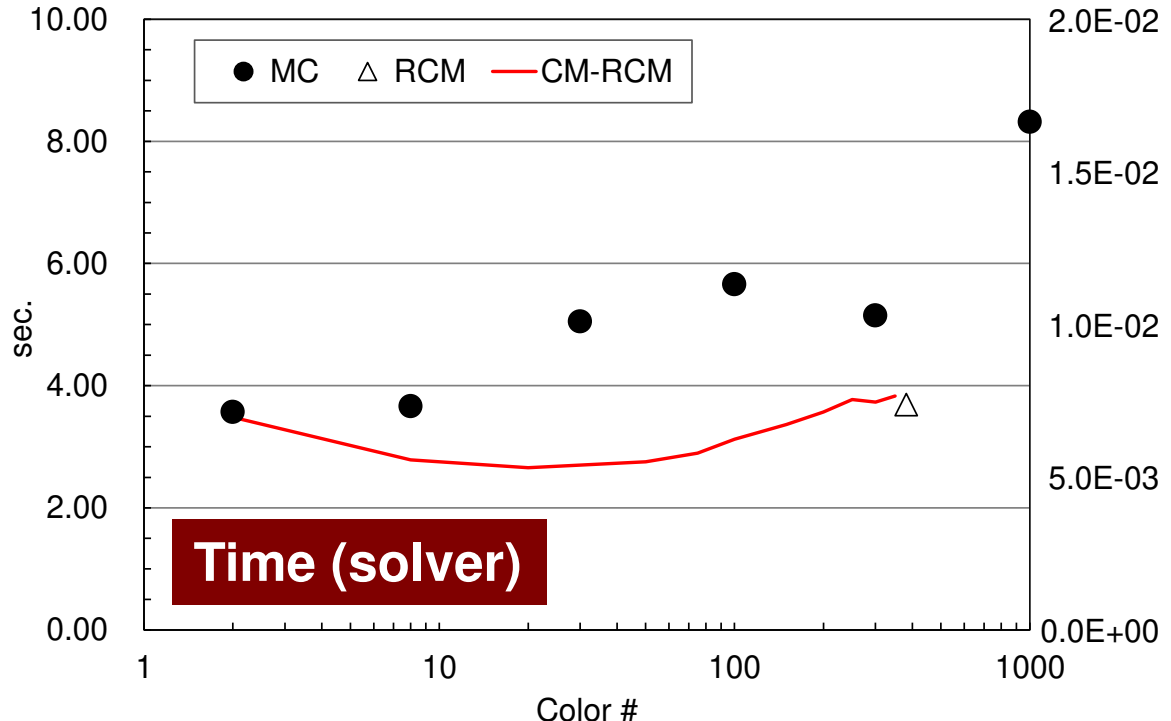
- Notice
  - Reordering may change results.
  - We need deep insight and understanding on background physics and mathematics.

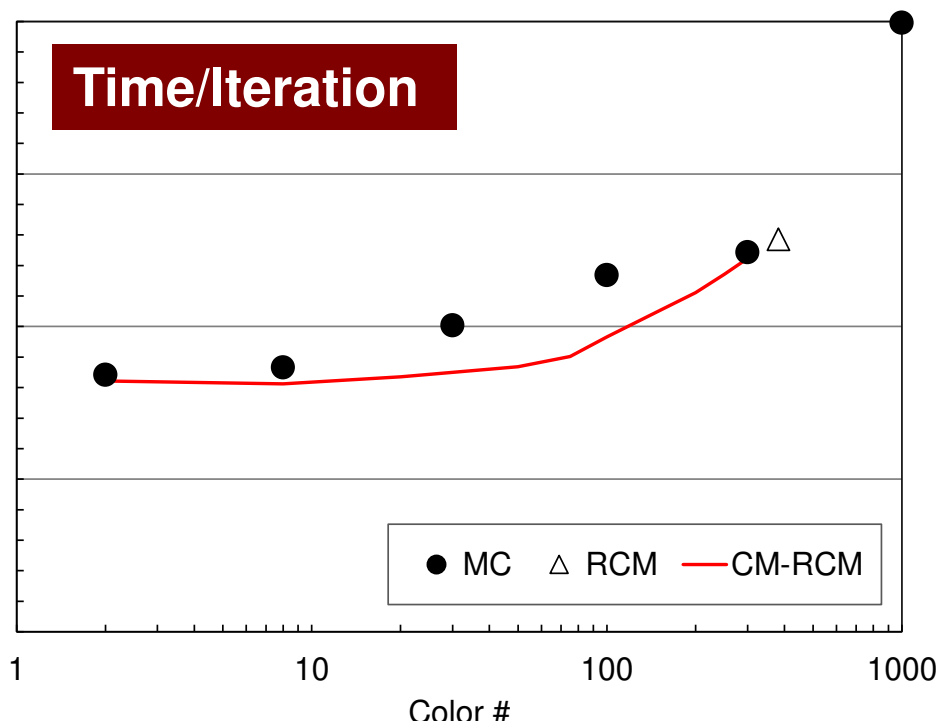
Odyssey

1-CMG/12-cores, 128^3

(●:MC, △:RCM, -:CM-RCM)

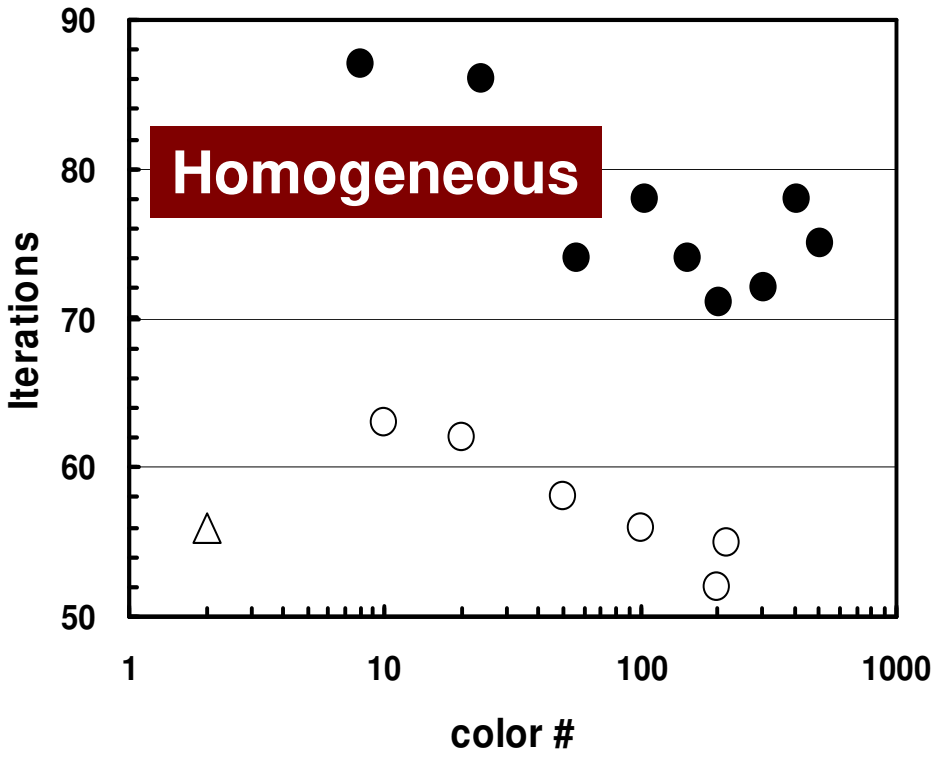


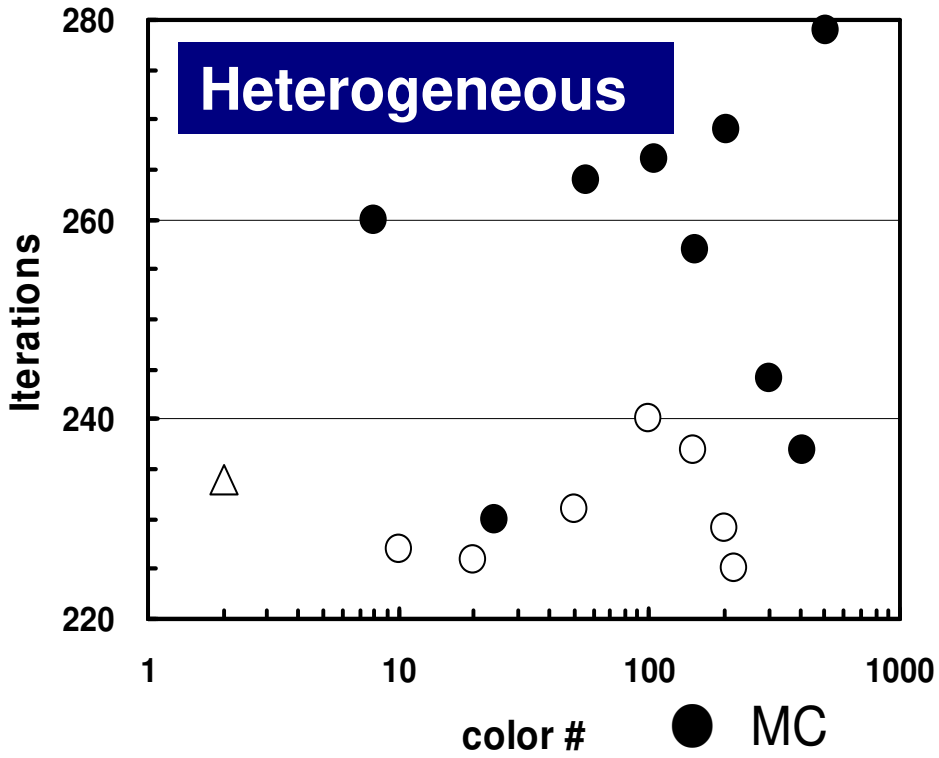




Comparison of Reordering Methods 3D Linear Elastic Problems

- MC: Slow convergence, unstable for heterogeneous cases (ill-conditioned problems).
- Cyclic-Mulricoloring + RCM (CM-RCM) is effective





MC
CM-RCM
No reordering

3D Linear-Elastic Problems with 32,768 DOF

- Remedy for Data Dependency
- Ordering/Reordering
  - Red-Black, Multicoloring (MC)
  - Cuthill-McKee (CM), Reverse-CM (RCM)
  - Reordering and Convergence
- **Implementation**
- ICCG with Reordering

Implementation: L2-color (1/2)

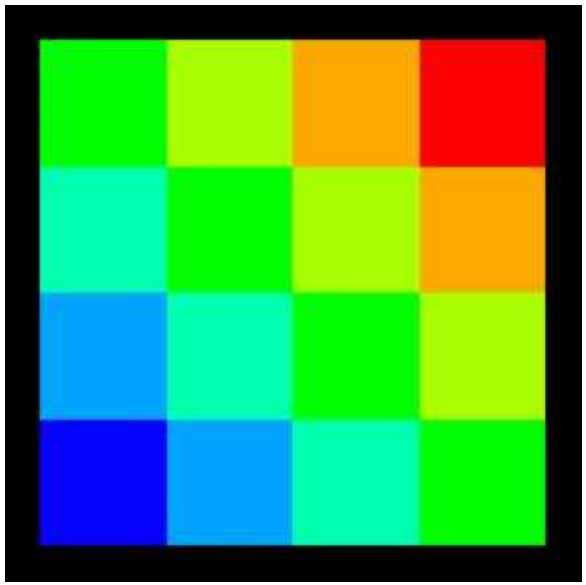
- Program for Coloring
  - MC, CM, RCM, and CM-RCM

```
$ cd multicore-c/L2/coloring/src
$ make
$ cd ../run
$ ./L2-color
   NX/NY/NZ ?

4    4    1        2D geometry with 16 meshes
You have        16 elements.
How many colors do you need ?
   #COLOR must be more than 2 and
   #COLOR must not be more than       16
    CM if #COLOR .eq. 0
   RCM if #COLOR .eq.-1
CMRCM if #COLOR .le.-2
=>
```

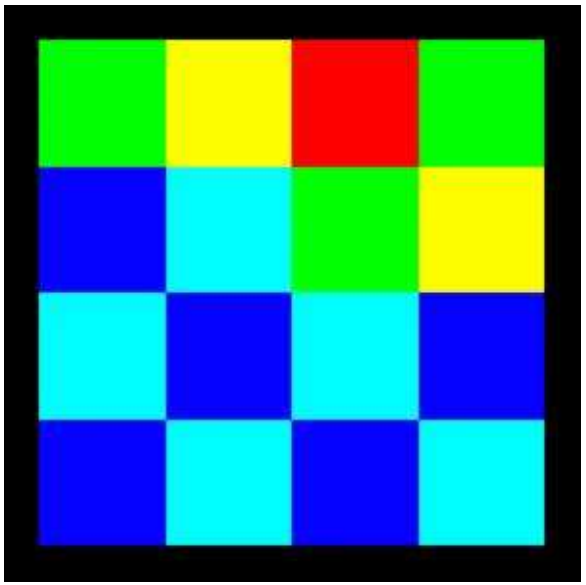
13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

Results: L2-color

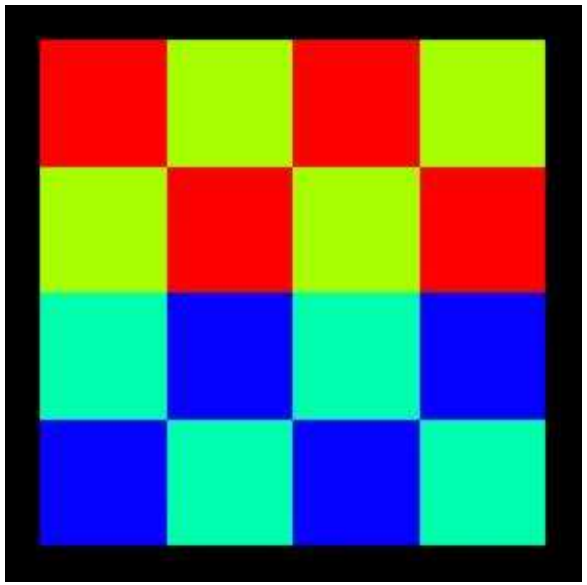
- 2 files are created
  - color.log Table of OLD-to-NEW mesh ID
    Information of the matrix
  - color.inp Colors/levels of meshes (ParaView)



INPUT: 0
(CM, 7 levels)



INPUT: 3
(MC, 5 colors)



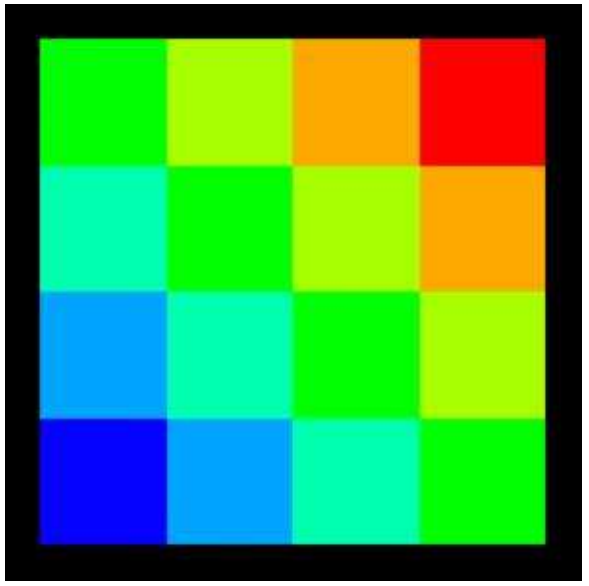
INPUT: 4
(MC, 4 colors)

INPUT=0: CM, 7-Levels

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

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

```
#new     1   #old     1   color     1
#new     2   #old     2   color     2
#new     3   #old     5   color     2
#new     4   #old     3   color     3
#new     5   #old     6   color     3
#new     6   #old     9   color     3
#new     7   #old     4   color     4
#new     8   #old     7   color     4
#new     9   #old    10   color     4
#new    10   #old    13   color     4
#new    11   #old     8   color     5
#new    12   #old    11   color     5
#new    13   #old    14   color     5
#new    14   #old    12   color     6
#new    15   #old    15   color     6
#new    16   #old    16   color     7
```

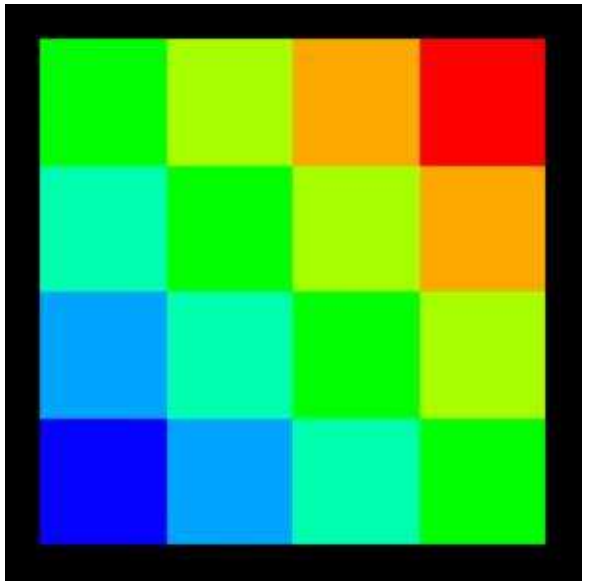


INPUT=0: CM, 7-Levels

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

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

```
#new     1   #old     1   color     1
#new     2   #old     2   color     2
#new     3   #old     5   color     2
#new     4   #old     3   color     3
#new     5   #old     6   color     3
#new     6   #old     9   color     3
#new     7   #old     4   color     4
#new     8   #old     7   color     4
#new     9   #old    10   color     4
#new    10   #old    13   color     4
#new    11   #old     8   color     5
#new    12   #old    11   color     5
#new    13   #old    14   color     5
#new    14   #old    12   color     6
#new    15   #old    15   color     6
#new    16   #old    16   color     7
```

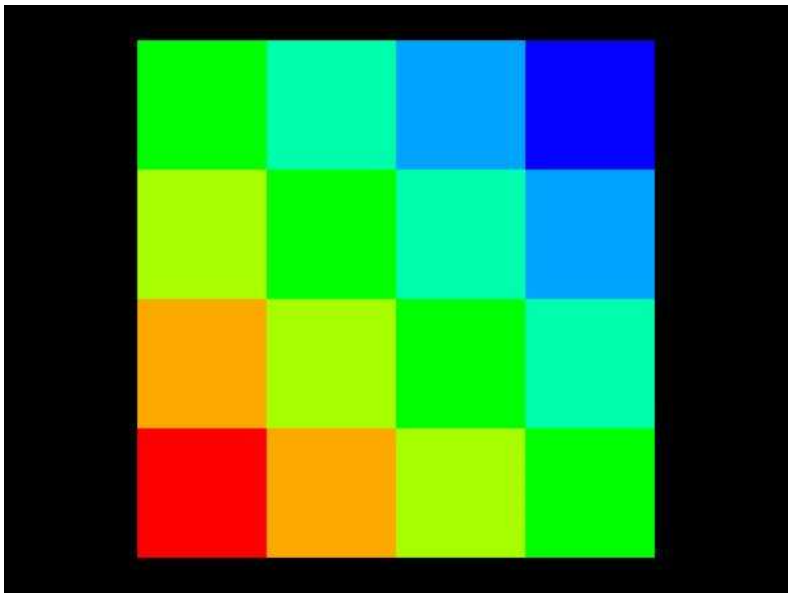


INPUT=-1: RCM, 7-Levels

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

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

```
#new    1   #old   16   color    1
#new    2   #old   15   color    2
#new    3   #old   12   color    2
#new    4   #old   14   color    3
#new    5   #old   11   color    3
#new    6   #old    8   color    3
#new    7   #old   13   color    4
#new    8   #old   10   color    4
#new    9   #old    7   color    4
#new   10   #old    4   color    4
#new   11   #old    9   color    5
#new   12   #old    6   color    5
#new   13   #old    3   color    5
#new   14   #old    5   color    6
#new   15   #old    2   color    6
#new   16   #old    1   color    7
```

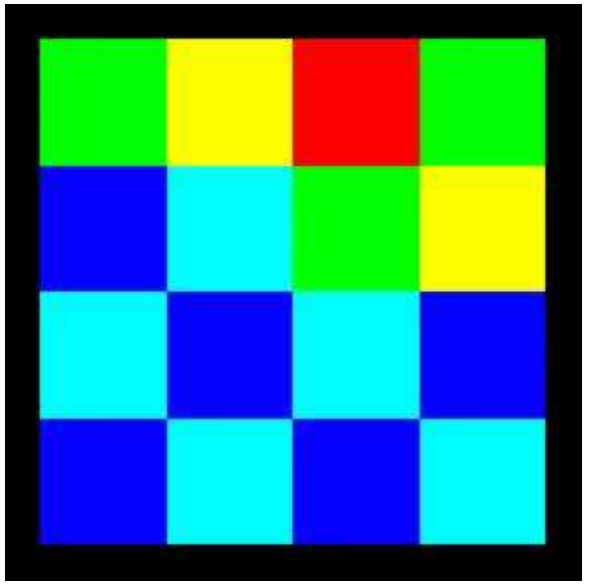


INPUT=3: MC, 5-Colors

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

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

```
#new     1   #old     1   color     1
#new     2   #old     3   color     1
#new     3   #old     6   color     1
#new     4   #old     8   color     1
#new     5   #old     9   color     1
#new     6   #old     2   color     2
#new     7   #old     4   color     2
#new     8   #old     5   color     2
#new     9   #old     7   color     2
#new    10   #old    10   color     2
#new    11   #old    11   color     3
#new    12   #old    13   color     3
#new    13   #old    16   color     3
#new    14   #old    12   color     4
#new    15   #old    14   color     4
#new    16   #old    15   color     5
```



INPUT=3: MC, 5-Colors

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

5					
	3		4		
1		2			

```
#new      1   #old      1   color      1
#new      2   #old      3   color      1
#new      3   #old      6   color      1
#new      4   #old      8   color      1
#new      5   #old      9   color      1
#new      6   #old      2   color      2
#new      7   #old      4   color      2
#new      8   #old      5   color      2
#new      9   #old      7   color      2
#new     10   #old     10   color      2
#new     11   #old     11   color      3
#new     12   #old     13   color      3
#new     13   #old     16   color      3
#new     14   #old     12   color      4
#new     15   #old     14   color      4
#new     16   #old     15   color      5
```

16/3=5= **ITEMcou**

5 independent meshes

INPUT=3: MC, 5-Colors

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

5	10				
8	3	9	4		
1	6	2	7		

```
#new      1   #old      1   color      1
#new      2   #old      3   color      1
#new      3   #old      6   color      1
#new      4   #old      8   color      1
#new      5   #old      9   color      1
#new      6   #old      2   color      2
#new      7   #old      4   color      2
#new      8   #old      5   color      2
#new      9   #old      7   color      2
#new     10   #old     10   color      2
#new     11   #old     11   color      3
#new     12   #old     13   color      3
#new     13   #old     16   color      3
#new     14   #old     12   color      4
#new     15   #old     14   color      4
#new     16   #old     15   color      5
```

16/3=5= **ITEMcou**

5 independent meshes

INPUT=3: MC, 5-Colors

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

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

```
#new      1   #old      1   color      1
#new      2   #old      3   color      1
#new      3   #old      6   color      1
#new      4   #old      8   color      1
#new      5   #old      9   color      1
#new      6   #old      2   color      2
#new      7   #old      4   color      2
#new      8   #old      5   color      2
#new      9   #old      7   color      2
#new     10   #old     10   color      2
#new     11   #old     11   color      3
#new     12   #old     13   color      3
#new     13   #old     16   color      3
#new     14   #old     12   color      4
#new     15   #old     14   color      4
#new     16   #old     15   color      5
```

16/3=5= **ITEMcou**

Proceed to the next color, if no more independent meshes.

INPUT=3: MC, 5-Colors

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

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

```
#new      1   #old      1   color      1
#new      2   #old      3   color      1
#new      3   #old      6   color      1
#new      4   #old      8   color      1
#new      5   #old      9   color      1
#new      6   #old      2   color      2
#new      7   #old      4   color      2
#new      8   #old      5   color      2
#new      9   #old      7   color      2
#new     10   #old     10   color      2
#new     11   #old     11   color      3
#new     12   #old     13   color      3
#new     13   #old     16   color      3
#new     14   #old     12   color      4
#new     15   #old     14   color      4
#new     16   #old     15   color      5
```

16/3=5= **ITEMcou**

Proceed to the next color, if no more independent meshes.

INPUT=3: MC, 5-Colors

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

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

```
#new     1   #old     1   color     1
#new     2   #old     3   color     1
#new     3   #old     6   color     1
#new     4   #old     8   color     1
#new     5   #old     9   color     1
#new     6   #old     2   color     2
#new     7   #old     4   color     2
#new     8   #old     5   color     2
#new     9   #old     7   color     2
#new    10   #old    10   color     2
#new    11   #old    11   color     3
#new    12   #old    13   color     3
#new    13   #old    16   color     3
#new    14   #old    12   color     4
#new    15   #old    14   color     4
#new    16   #old    15   color     5
```

16/3=5= **ITEMcou**

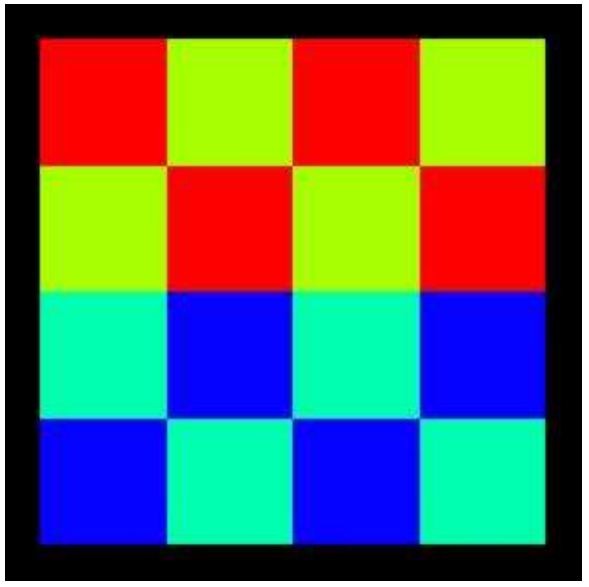
Finally, 5 colors are needed.

INPUT=4: MC, 4-Colors

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

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

```
#new     1   #old     1   color     1
#new     2   #old     3   color     1
#new     3   #old     6   color     1
#new     4   #old     8   color     1
#new     5   #old     2   color     2
#new     6   #old     4   color     2
#new     7   #old     5   color     2
#new     8   #old     7   color     2
#new     9   #old     9   color     3
#new    10   #old    11   color     3
#new    11   #old    14   color     3
#new    12   #old    16   color     3
#new    13   #old    10   color     4
#new    14   #old    12   color     4
#new    15   #old    13   color     4
#new    16   #old    15   color     4
```



INPUT=3: MC, 5-Colors

color.log: matrix info.

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

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

```
### INITIAL connectivity
I=    1      INL(i)=    0      INU(i)=    2
  IAL:
  IAU:    2    5
I=    2      INL(i)=    1      INU(i)=    2
  IAL:    1
  IAU:    3    6
I=    3      INL(i)=    1      INU(i)=    2
  IAL:    2
  IAU:    4    7
I=    4      INL(i)=    1      INU(i)=    1
  IAL:    3
  IAU:    8
I=    5      INL(i)=    1      INU(i)=    2
  IAL:    1
  IAU:    6    9
I=    6      INL(i)=    2      INU(i)=    2
  IAL:    2    5
  IAU:    7   10
I=    7      INL(i)=    2      INU(i)=    2
  IAL:    3    6
  IAU:    8   11
I=    8      INL(i)=    2      INU(i)=    1
  IAL:    4    7
  IAU:   12
I=    9      INL(i)=    1      INU(i)=    2
  IAL:    5
  IAU:   10   13
I=   10      INL(i)=    2      INU(i)=    2
  IAL:    6    9
  IAU:   11   14
I=   11      INL(i)=    2      INU(i)=    2
  IAL:    7   10
  IAU:   12   15
I=   12      INL(i)=    2      INU(i)=    1
  IAL:    8   11
  IAU:   16
I=   13      INL(i)=    1      INU(i)=    1
  IAL:    9
  IAU:   14
I=   14      INL(i)=    2      INU(i)=    1
  IAL:   10   13
  IAU:   15
I=   15      INL(i)=    2      INU(i)=    1
  IAL:   11   14
  IAU:   16
I=   16      INL(i)=    2      INU(i)=    0
  IAL:   12   15
  IAU:


COLOR number        5

  #new       1   #old       1  color     1
  #new       2   #old       3  color     1
  #new       3   #old       6  color     1
  #new       4   #old       8  color     1
  #new       5   #old       9  color     1
  #new       6   #old       2  color     2
  #new       7   #old       4  color     2
  #new       8   #old       5  color     2
  #new       9   #old       7  color     2
  #new      10   #old      10  color     2
  #new      11   #old      11  color     3
  #new      12   #old      13  color     3
  #new      13   #old      16  color     3
  #new      14   #old      12  color     4
  #new      15   #old      14  color     4
  #new      16   #old      15  color     5
```

INPUT=3: MC, 5-Colors

color.log: matrix info.

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

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

```
### FINAL connectivity
I=    1     INL(i)=    0     INU(i)=    2
  IAL:
  IAU:    6    8
I=    2     INL(i)=    0     INU(i)=    3
  IAL:
  IAU:    6    7    9
I=    3     INL(i)=    0     INU(i)=    4
  IAL:
  IAU:    6    8    9   10
I=    4     INL(i)=    0     INU(i)=    3
  IAL:
  IAU:    7    9   14
I=    5     INL(i)=    0     INU(i)=    3
  IAL:
  IAU:    8   10   12
I=    6     INL(i)=    3     INU(i)=    0
  IAL:    1    2    3
  IAU:
I=    7     INL(i)=    2     INU(i)=    0
  IAL:    2    4
  IAU:
I=    8     INL(i)=    3     INU(i)=    0
  IAL:    1    3    5
  IAU:
I=    9     INL(i)=    3     INU(i)=    1
  IAL:    2    3    4
  IAU:   11
I=   10     INL(i)=    2     INU(i)=    2
  IAL:    3    5
  IAU:   11   15
I=   11     INL(i)=    2     INU(i)=    2
  IAL:    9   10
  IAU:   14   16
I=   12     INL(i)=    1     INU(i)=    1
  IAL:    5
  IAU:   15
I=   13     INL(i)=    0     INU(i)=    2
  IAL:
  IAU:   14   16
```

```
I=   14     INL(i)=  3     INU(i)=  0
  IAL:    4   11   13
  IAU:
I=   15     INL(i)=  2     INU(i)=  1
  IAL:   10   12
  IAU:   16
I=   16     INL(i)=  3     INU(i)=  0
  IAL:   11   15   13
  IAU:
```

Source Files: L2-color

```
$ cd multicore-c/L2/coloring/src
$ ls
```

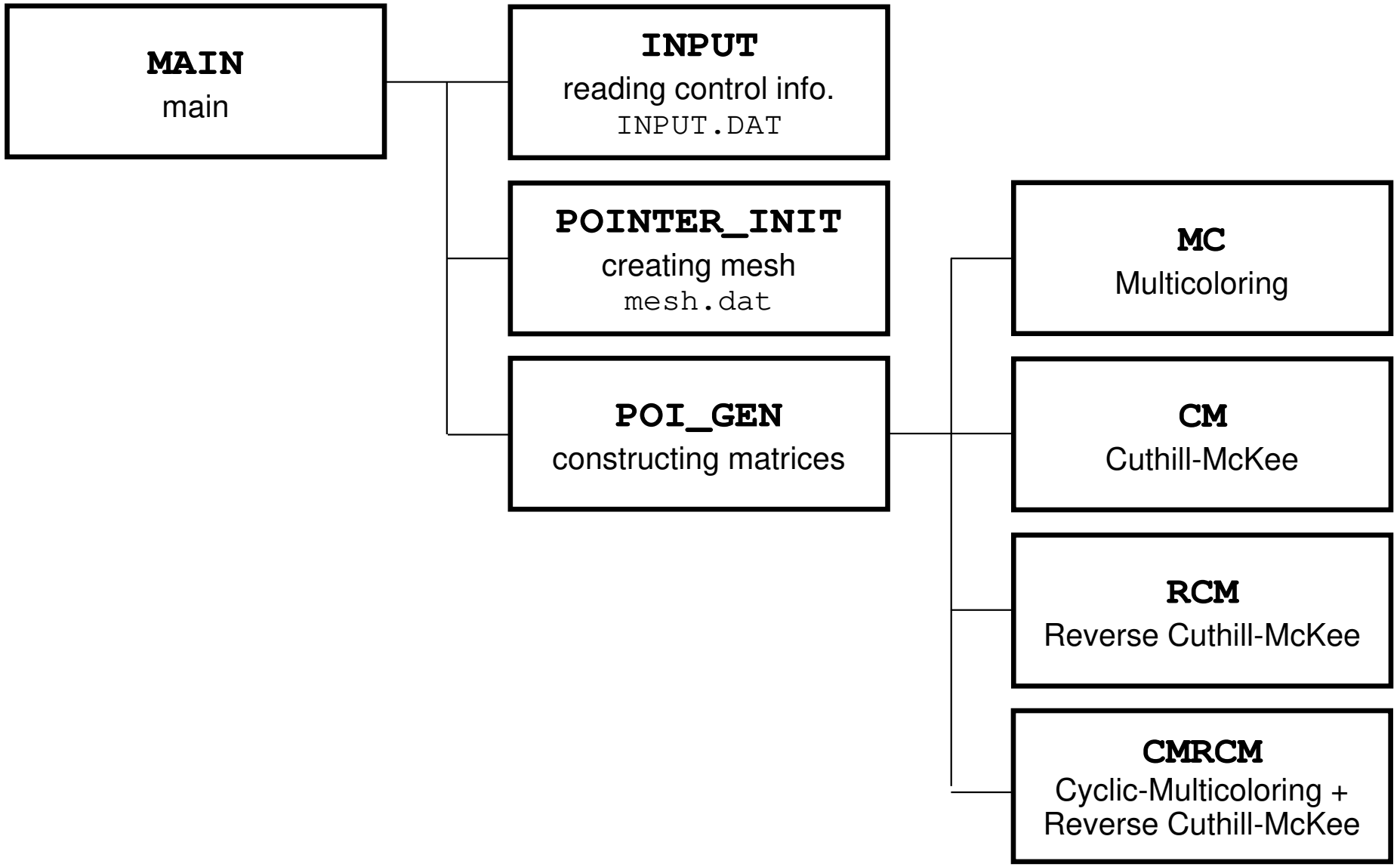
13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

Target geometry to be colored

Main Program

```
#include <stdio.h> ...

int
main(int argc, char **argv)
{
        FILE *fp21;
        int i, ic, k;

        if(POINTER_INIT()) goto error;
        if((fp21 = fopen("color.log", "w")) == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        if(POI_GEN(fp21)) goto error;
        if(OUTUCD()) goto error;

        fprintf(fp21, "¥n¥nCOLOR number%8d¥n¥n", NCOLORtot);
        for(ic=1; ic<=NCOLORtot; ic++) {
                for(i=COLORindex[ic-1]+1; i<=COLORindex[ic]; i++) {
                        fprintf(fp21, "  #new%8d   #old%8d  color%8d¥n", i, NEWtoOLD[i-1]+1, ic);
                }
        }

        (...)

        fclose(fp21);
        return 0;

error:
        return -1;
}
```

Structure of L2-color



Main Program

```
#include <stdio.h> ...

int
main(int argc, char **argv)
{
        FILE *fp21;
        int i, ic, k;

        if(POINTER_INIT()) goto error;
        if((fp21 = fopen("color.log", "w")) == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        if(POI_GEN(fp21)) goto error;
        if(OUTUCD()) goto error;

        fprintf(fp21, "¥n¥nCOLOR number%8d¥n¥n", NCOLORtot);
        for(ic=1; ic<=NCOLORtot; ic++) {
                for(i=COLORindex[ic-1]+1; i<=COLORindex[ic]; i++) {
                        fprintf(fp21, "  #new%8d   #old%8d  color%8d¥n", i, NEWtoOLD[i-1]+1, ic);
                }
        }

        (...)

        fclose(fp21);
        return 0;

error:
        return -1;
}
```

struct.h

```
#ifndef __H_STRUCT
#define __H_STRUCT

#include <omp.h>

int ICELTOT, ICELTOTp, N;
int NX, NY, NZ, NXP1, NYP1, NZP1, IBNODTOT;
int NXc, NYc, NZc;

double DX, DY, DZ, XAREA, YAREA, ZAREA;
double RDX, RDY, RDZ, RDX2, RDY2, RDZ2, R2DX, R2DY, R2DZ;
double *VOLCEL, *VOLNOD, *RVC, *RVN;

int **XYZ, **NEIBcell;

int ZmaxCELtot;
int *BC_INDEX, *BC_NOD;
int *ZmaxCEL;

int **IWKX;
double **FCV;

int my_rank, PETOT, PEsmpTOT;

#endif /* __H_STRUCT */
```

ICELTOT:
Number of meshes (NX x NY x NZ)

N:
Number of modes

NX,NY,NZ:
Number of meshes in x/y/z directions

NXP1,NYP1,NZP1:
Number of nodes in x/y/z directions

IBNODTOT:
= NXP1 x NYP1

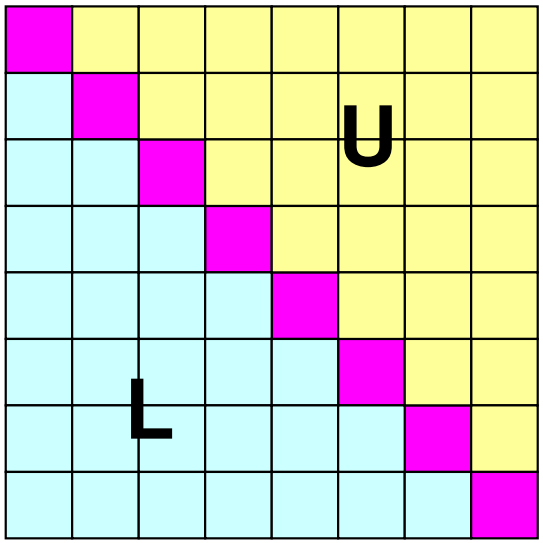
XYZ[ICELTOT][3]:
Location of meshes

NEIBcell[ICELTOT][6]:
Neighboring meshes

pcg.h

```
#ifndef __H_PCG
#define __H_PCG
        static int N2 = 256;
        int NUmax, NLmax, NCOLORtot, NCOLORk, NU,
NL;
        int METHOD, ORDER_METHOD;
        double EPSICCG;

        double *D, *PHI, *BFORCE;
        double *AL, *AU;

        int *INL, *INU, *COLORindex;
        int *indexL, *indexU;
        int *OLDtoNEW, *NEWtoOLD;
        int **IAL, **IAU;
        int *itemL, *itemU;
        int NPL, NPU;
#endif /* __H_PCG */
```

- Sparse Matrix
- Only non-zero off-diagonal components (CRS)
- Diagonal/Lower/Upper components are stored separately



pcg.h

```
#ifndef __H_PCG
#define __H_PCG
        static int N2 = 256;
        int NUmax, NLmax, NCOLORtot, NCOLORk, NU, NL;
        int METHOD, ORDER_METHOD;
        double EPSICCG;

        double *D, *PHI, *BFORCE;
        double *AL, *AU;

        int *INL, *INU, *COLORindex;
        int *indexL, *indexU;
        int *OLDtoNEW, *NEWtoOLD;
        int **IAL, **IAU;
        int *itemL, *itemU;
        int NPL, NPU;
#endif /* __H_PCG */
```

Auxiliary Arrays

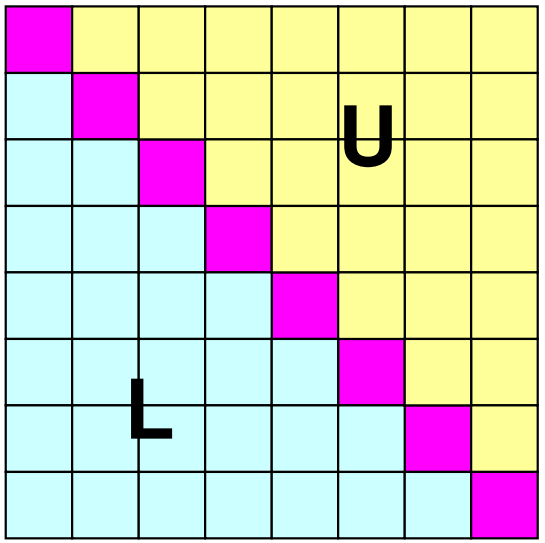
Lower Part (Column ID)

`IAL[i][icou] < i`

Upper Part (Column ID)

`IAU[i][icou] > i`

`INL[ICELTOT]`	# Non-zero off-diag. components (lower)
`IAL[ICELTOT][NL]`	**Col. ID: non-zero off-diag. comp. (lower)**
`INU[ICELTOT]`	# Non-zero off-diag. components (upper)
`IAU[ICELTOT][NU]`	**Col. ID: non-zero off-diag. comp. (upper)**
`NU,NL`	Max # of L/U non-zero off-diag. comp.s (=6)
`indexL[ICELTOT+1]`	# Non-zero off-diag. comp. (lower, CRS)
`indexU[ICELTOT+1]`	# Non-zero off-diag. comp. (upper, CRS)
`NPL, NPU`	Total # of L/U non-zero off-diag. comp.
`itemL[NPL],itemU[NPU]`	Col. ID: non-zero off-diag. comp. (L/U, CRS)



```
#ifndef __H_PCG
#define __H_PCG
        static int N2 = 256;
        int NUmax, NLmax, NCOLORtot, NCOLORk, NU, NL;
        int METHOD, ORDER_METHOD;
        double EPSICCG;

        double *D, *PHI, *BFORCE;
        double *AL, *AU;

        int *INL, *INU, *COLORindex;
        int *indexL, *indexU;
        int *OLDtoNEW, *NEWtoOLD;
        int **IAL, **IAU;
        int *itemL, *itemU;
        int NPL, NPU;
#endif /* __H_PCG */
```

pcg.h

Auxiliary Arrays

Lower Part (Column ID)

`IAL[i][icou] < i`

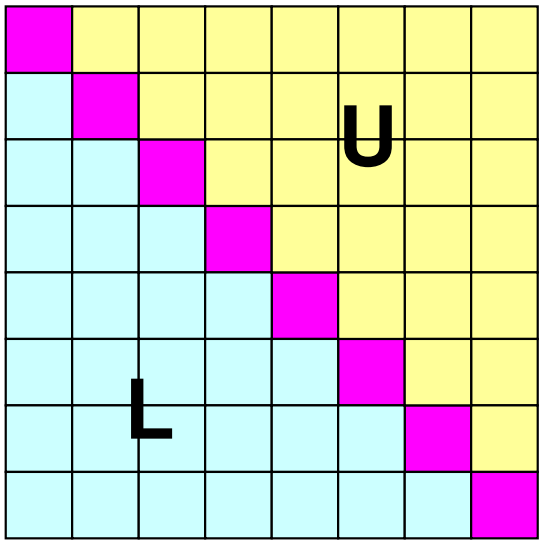
`INL[i]`: Number@each row

Upper Part (Column ID)

`IAU[i][icou] > i`

`INU[i]`: Number@each row

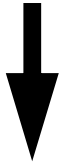
`INL[ICELTOT]`	**# Non-zero off-diag. components (lower)**
`IAL[ICELTOT][NL]`	Col. ID: non-zero off-diag. comp. (lower)
`INU[ICELTOT]`	**# Non-zero off-diag. components (upper)**
`IAU[ICELTOT][NU]`	Col. ID: non-zero off-diag. comp. (upper)
`NU,NL`	Max # of L/U non-zero off-diag. comp.s (=6)
`indexL[ICELTOT+1]`	# Non-zero off-diag. comp. (lower, CRS)
`indexU[ICELTOT+1]`	# Non-zero off-diag. comp. (upper, CRS)
`NPL, NPU`	Total # of L/U non-zero off-diag. comp.
`itemL[NPL],itemU[NPU]`	Col. ID: non-zero off-diag. comp. (L/U, CRS)



INPUT=3: MC, 5-Colors

color.log: matrix info.

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



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

```
### INITIAL connectivity
I=    1      INL(i)=    0      INU(i)=    2
  IAL:
  IAU:    2    5
I=    2      INL(i)=    1      INU(i)=    2
  IAL:    1
  IAU:    3    6
I=    3      INL(i)=    1      INU(i)=    2
  IAL:    2
  IAU:    4    7
I=    4      INL(i)=    1      INU(i)=    1
  IAL:    3
  IAU:    8
I=    5      INL(i)=    1      INU(i)=    2
  IAL:    1
  IAU:    6    9
I=    6      INL(i)=    2      INU(i)=    2
  IAL:    2    5
  IAU:    7   10
I=    7      INL(i)=    2      INU(i)=    2
  IAL:    3    6
  IAU:    8   11
I=    8      INL(i)=    2      INU(i)=    1
  IAL:    4    7
  IAU:   12
I=    9      INL(i)=    1      INU(i)=    2
  IAL:    5
  IAU:   10   13
I=   10      INL(i)=    2      INU(i)=    2
  IAL:    6    9
  IAU:   11   14
I=   11      INL(i)=    2      INU(i)=    2
  IAL:    7   10
  IAU:   12   15
I=   12      INL(i)=    2      INU(i)=    1
  IAL:    8   11
  IAU:   16
I=   13      INL(i)=    1      INU(i)=    1
  IAL:    9
  IAU:   14
```

```
I=   14      INL(i)=  2      INU(i)=  1
  IAL:   10   13
  IAU:   15
I=   15      INL(i)=  2      INU(i)=  1
  IAL:   11   14
  IAU:   16
I=   16      INL(i)=  2      INU(i)=  0
  IAL:   12   15
  IAU:


COLOR number        5

  #new       1   #old       1  color     1
  #new       2   #old       3  color     1
  #new       3   #old       6  color     1
  #new       4   #old       8  color     1
  #new       5   #old       9  color     1
  #new       6   #old       2  color     2
  #new       7   #old       4  color     2
  #new       8   #old       5  color     2
  #new       9   #old       7  color     2
  #new      10   #old      10  color     2
  #new      11   #old      11  color     3
  #new      12   #old      13  color     3
  #new      13   #old      16  color     3
  #new      14   #old      12  color     4
  #new      15   #old      14  color     4
  #new      16   #old      15  color     5
```

INPUT=3: MC, 5-Colors

color.log: matrix info.

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

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

```
### FINAL connectivity
I=    1     INL(i)=    0     INU(i)=    2
  IAL:
  IAU:    6    8
I=    2     INL(i)=    0     INU(i)=    3
  IAL:
  IAU:    6    7    9
I=    3     INL(i)=    0     INU(i)=    4
  IAL:
  IAU:    6    8    9   10
I=    4     INL(i)=    0     INU(i)=    3
  IAL:
  IAU:    7    9   14
I=    5     INL(i)=    0     INU(i)=    3
  IAL:
  IAU:    8   10   12
I=    6     INL(i)=    3     INU(i)=    0
  IAL:    1    2    3
  IAU:
I=    7     INL(i)=    2     INU(i)=    0
  IAL:    2    4
  IAU:
I=    8     INL(i)=    3     INU(i)=    0
  IAL:    1    3    5
  IAU:
I=    9     INL(i)=    3     INU(i)=    1
  IAL:    2    3    4
  IAU:   11
I=   10     INL(i)=    2     INU(i)=    2
  IAL:    3    5
  IAU:   11   15
I=   11     INL(i)=    2     INU(i)=    2
  IAL:    9   10
  IAU:   14   16
I=   12     INL(i)=    1     INU(i)=    1
  IAL:    5
  IAU:   15
I=   13     INL(i)=    0     INU(i)=    2
  IAL:
  IAU:   14   16
```

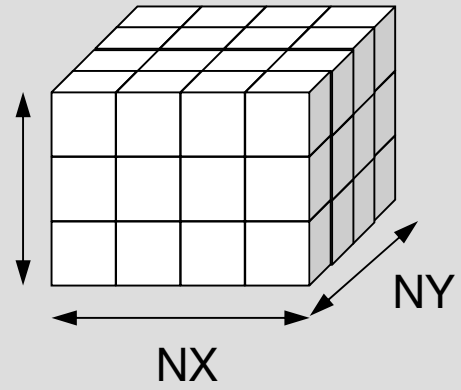
```
I=   14     INL(i)=  3     INU(i)=  0
  IAL:    4   11   13
  IAU:
I=   15     INL(i)=  2     INU(i)=  1
  IAL:   10   12
  IAU:   16
I=   16     INL(i)=  3     INU(i)=  0
  IAL:   11   15   13
  IAU:
```

Main Program

```
#include <stdio.h> ...

int
main(int argc, char **argv)
{
        FILE *fp21;
        int i, ic, k;

        if(POINTER_INIT()) goto error;
        if((fp21 = fopen("color.log", "w")) == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        if(POI_GEN(fp21)) goto error;
        if(OUTUCD()) goto error;

        fprintf(fp21, "¥n¥nCOLOR number%8d¥n¥n", NCOLORtot);
        for(ic=1; ic<=NCOLORtot; ic++) {
                for(i=COLORindex[ic-1]+1; i<=COLORindex[ic]; i++) {
                        fprintf(fp21, "  #new%8d   #old%8d  color%8d¥n", i, NEWtoOLD[i-1]+1, ic);
                }
        }

        (...)

        fclose(fp21);
        return 0;

error:
        return -1;
}
```

pointer_init (1/3)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include "struct_ext.h"
#include "pcg_ext.h"
#include "pointer_init.h"
#include "allocate.h"

extern int
POINTER_INIT(void)
{

        int icel, ipe, i, j, k;

 fprintf(stderr, "input NX NY NZ=>¥n");
 fscanf(stdin, "%d%d%d", &NX, &NY, &NZ);
 fprintf(stderr, "NX=%d NY=%d NZ=%d¥n", NX, NY, NZ);

/*
 * INIT.
 */

  ICELTOT = NX * NY * NZ;

  NXP1 = NX + 1;
  NYP1 = NY + 1;
  NZP1 = NZ + 1;


  NEIBcell =
    (int **)allocate_matrix(sizeof(int),ICELTOT,6);

  XYZ      =
    (int **)allocate_matrix(sizeof(int),ICELTOT,3);
```



NX,NY,NZ:

Number of meshes in x/y/z directions

NXP1,NYP1,NZP1:

Number of nodes in x/y/z directions (for visualization)

ICELTOT:

Number of meshes (NX x NY x NZ)

XYZ[ICELTOT][3]:

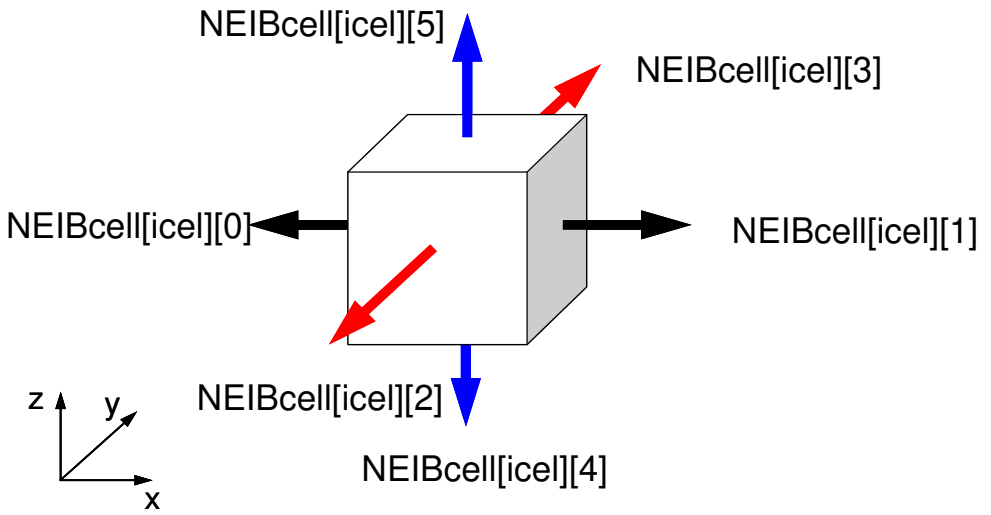
Location of meshes

NEIBcell[ICELTOT][6]:

Neighboring meshesc

pointer_init (2/3)

```
for(k=0; k<NZ; k++) {
    for(j=0; j<NY; j++) {
        for(i=0; i<NX; i++) {
            icel = k * NX * NY + j * NX + i;
            NEIBcell[icel][0] = icel - 1       + 1;
            NEIBcell[icel][1] = icel + 1       + 1;
            NEIBcell[icel][2] = icel - NX      + 1;
            NEIBcell[icel][3] = icel + NX      + 1;
            NEIBcell[icel][4] = icel - NX * NY + 1;
            NEIBcell[icel][5] = icel + NX * NY + 1;
            if(i == 0)    NEIBcell[icel][0] = 0;
            if(i == NX-1) NEIBcell[icel][1] = 0;
            if(j == 0)    NEIBcell[icel][2] = 0;
            if(j == NY-1) NEIBcell[icel][3] = 0;
            if(k == 0)    NEIBcell[icel][4] = 0;
            if(k == NZ-1) NEIBcell[icel][5] = 0;

            XYZ[icel][0] = i + 1;
            XYZ[icel][1] = j + 1;
            XYZ[icel][2] = k + 1;
        }
    }
}
```



i= XYZ[icel][0]
j= XYZ[icel][1], k= XYZ[icel][2]
icel= k*NX*NY + j*NX + i

“icel” starts at 0

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

“NEIBcell” starts at 1

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

NEIBcell[icel][0]= icel – 1 + 1
NEIBcell[icel][1]= icel + 1 + 1
NEIBcell[icel][2]= icel – NX + 1
NEIBcell[icel][3]= icel + NX + 1
NEIBcell[icel][4]= icel – NX*NY + 1
NEIBcell[icel][5]= icel + NX*NY + 1

pointer_init (3/3)

```
        if(DX <= 0.0) {
                DX = 1.0 / (double)NX;
                DY = 1.0 / (double)NY;
                DZ = 1.0 / (double)NZ;
        }

        NXP1 = NX + 1;
        NYP1 = NY + 1;
        NZP1 = NZ + 1;

        IBNODTOT = NXP1 * NYP1;
        N        = NXP1 * NYP1 * NZP1;

        return 0;
}
```

Main Program

```
#include <stdio.h> ...

int
main(int argc, char **argv)
{
        FILE *fp21;
        int i, ic, k;

        if(POINTER_INIT()) goto error;
        if((fp21 = fopen("color.log", "w")) == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        if(POI_GEN(fp21)) goto error;
        if(OUTUCD()) goto error;

        fprintf(fp21, "¥n¥nCOLOR number%8d¥n¥n", NCOLORtot);
        for(ic=1; ic<=NCOLORtot; ic++) {
                for(i=COLORindex[ic-1]+1; i<=COLORindex[ic]; i++) {
                        fprintf(fp21, "  #new%8d   #old%8d  color%8d¥n", i, NEWtoOLD[i-1]+1, ic);
                }
        }

        (...)

        fclose(fp21);
        return 0;

error:
        return -1;
}
```

poi_gen (1/4)

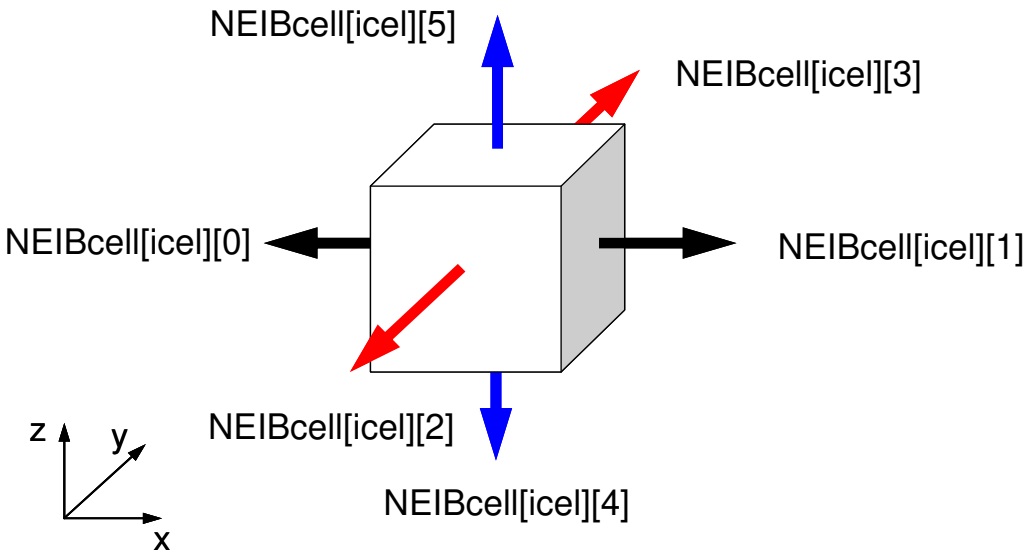
```
#include "allocate.h"
extern int
POI_GEN(void)
{
  int nn;
  int ic0, icN1, icN2, icN3, icN4, icN5, icN6;
  int i, j, k, ib, ic, ip, icel, icou, icol, icouG;
  int ii, jj, kk, nn1, num, nr, j0, j1;
  double coef, VOL0, S1t, E1t;
  int isL, ieL, isU, ieU;
  NL=6; NU= 6;

  IAL = (int **)allocate_matrix(sizeof(int),ICELTOT,NL);
  IAU = (int **)allocate_matrix(sizeof(int),ICELTOT,NU);
  INL = (int *)allocate_vector(sizeof(int),ICELTOT);
  INU = (int *)allocate_vector(sizeof(int),ICELTOT);

  for (i = 0; i <ICELTOT ; i++) {
       BFORCE[i]=0.0;
       D[i]  =0.0; PHI[i]=0.0;
       INL[i] = 0; INU[i] = 0;
       for(j=0;j<6;j++){
           IAL[i][j]=0; IAU[i][j]=0;
       }
  }
  for (i = 0; i <=ICELTOT ; i++) {
       indexL[i] = 0; indexU[i] = 0;
  }
```

allocate.c

```
/*****************************
  allocate matrix
*****************************/
void** allocate_matrix(int size,int m,int n)
{
  void **aa;
  int i;
  if ( ( aa=(void ** )malloc( m * sizeof(void*) ) ) == NULL ) {
     fprintf(stdout,"Error:Memory does not enough! aa in matrix ¥n");
     exit(1);
  }
  if ( ( aa[0]=(void * )malloc( m * n * size ) ) == NULL ) {
     fprintf(stdout,"Error:Memory does not enough! in matrix ¥n");
     exit(1);
  }
        for(i=1;i<m;i++) aa[i]=(char*)aa[i-1]+size*n;
  return aa;
}
```

```
for(icel=0; icel<ICELTOT; icel++) {

     icN1 = NEIBcell[icel][0];
     icN2 = NEIBcell[icel][1];
     icN3 = NEIBcell[icel][2];
     icN4 = NEIBcell[icel][3];
     icN5 = NEIBcell[icel][4];
     icN6 = NEIBcell[icel][5];

     if(icN5 != 0) {
        icou = INL[icel] + 1;
        IAL[icel][icou-1] = icN5;
        INL[icel]         = icou;
      }

      if(icN3 != 0) {
        icou = INL[icel] + 1;
        IAL[icel][icou-1] = icN3;
        INL[icel]         = icou;
      }

      if(icN1 != 0) {
        icou = INL[icel] + 1;
        IAL[icel][icou-1] = icN3;
        INL[icel]         = icou;
      }
```

poi_gen (2/4)



Lower Triangular Part

NEIBcell[icel][4]= icel – NX*NY + 1
NEIBcell[icel][2]= icel – NX + 1
NEIBcell[icel][0]= icel – 1 + 1

“icel” starts at 0

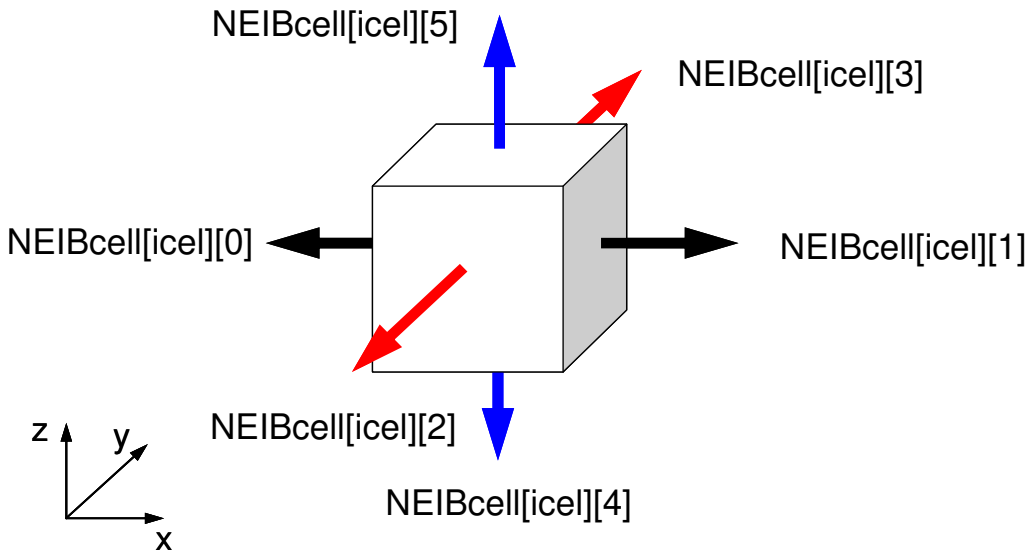
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

“IAL” starts at 1

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

poi_gen (3/4)

```
 for(icel=0; icel<ICELTOT; icel++) {

      icN1 = NEIBcell[icel][0];
      icN2 = NEIBcell[icel][1];
      icN3 = NEIBcell[icel][2];
      icN4 = NEIBcell[icel][3];
      icN5 = NEIBcell[icel][4];
      icN6 = NEIBcell[icel][5];
....
      if(icN2 != 0) {
        icou = INU[icel] + 1;
        IAU[icel][icou-1] = icN2;
        INU[icel]         = icou;
      }

      if(icN4 != 0) {
        icou = INU[icel] + 1;
        IAU[icel][icou-1] = icN4;
        INU[icel]         = icou;
      }

      if(icN6 != 0) {
        icou = INU[icel] + 1;
        IAU[icel][icou-1] = icN6;
        INU[icel]         = icou;
      }
}
```



Upper Triangular Part

NEIBcell[icel][1]= icel + 1 + 1
NEIBcell[icel][3]= icel + NX + 1
NEIBcell[icel][5]= icel + NX*NY + 1

“icel” starts at 0

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

“IAU” starts at 1

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

poi_gen (4/4)

Reading “initial” color number

```
N111:
        fprintf(stderr, "¥n¥nYou have%8d elements¥n", ICELTOT);
        fprintf(stderr, "How many colors do you need ?¥n");
        fprintf(stderr, "  #COLOR must be more than 2 and¥n");
        fprintf(stderr, "  #COLOR must not be more than%8d¥n", ICELTOT);
        fprintf(stderr, "  if #COLOR= 0 then CM  ordering¥n");
        fprintf(stderr, "  if #COLOR=-1 then RCM ordering¥n");
        fprintf(stderr, "  if #COLOR<-1 then CMRCM ordering¥n");
        fprintf(stderr, "=>¥n");
        fscanf(stdin, "%d", &NCOLORtot);
        if(NCOLORtot == 1 && NCOLORtot > ICELTOT) goto N111;

        OLDtoNEW = (int *)calloc(ICELTOT, sizeof(int));
        if(OLDtoNEW == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        NEWtoOLD = (int *)calloc(ICELTOT, sizeof(int));
        if(NEWtoOLD == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        COLORindex = (int *)calloc(ICELTOT+1, sizeof(int));
        if(COLORindex == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }

        if(NCOLORtot > 0) {
                MC(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot == 0) {
                CM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot ==-1) {
                RCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot <-1) {
                CMRCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        }

        fprintf(stderr, "¥n# TOTAL COLOR number%8d¥n", NCOLORtot);
        return 0;
}
```

poi_gen (4/4)

Allocate matrices

```
N111:
        fprintf(stderr, "¥n¥nYou have%8d elements¥n", ICELTOT);
        fprintf(stderr, "How many colors do you need ?¥n");
        fprintf(stderr, "  #COLOR must be more than 2 and¥n");
        fprintf(stderr, "  #COLOR must not be more than%8d¥n", ICELTOT);
        fprintf(stderr, "  if #COLOR= 0 then CM  ordering¥n");
        fprintf(stderr, "  if #COLOR=-1 then RCM ordering¥n");
        fprintf(stderr, "  if #COLOR<-1 then CMRCM ordering¥n");
        fprintf(stderr, "=>¥n");
        fscanf(stdin, "%d", &NCOLORtot);
        if(NCOLORtot == 1 && NCOLORtot > ICELTOT) goto N111;

        OLDtoNEW = (int *)calloc(ICELTOT, sizeof(int));
        if(OLDtoNEW == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        NEWtoOLD = (int *)calloc(ICELTOT, sizeof(int));
        if(NEWtoOLD == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        COLORindex = (int *)calloc(ICELTOT+1, sizeof(int));
        if(COLORindex == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }

        if(NCOLORtot > 0) {
                MC(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot == 0) {
                CM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot ==-1) {
                RCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot <-1) {
                CMRCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        }

        fprintf(stderr, "¥n# TOTAL COLOR number%8d¥n", NCOLORtot);
        return 0;
}
```

poi_gen (4/4)

```
        if(NCOLORtot > 0) {
                MC(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot == 0) {
                CM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot ==-1) {
                RCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot <-1) {
                CMRCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        }

        fprintf(stderr, "¥n# TOTAL COLOR number%8d¥n", NCOLORtot);
        return 0;
```

INL, INU, IAL, IAU	Info. after renumbering
OLDtoNEW, NEWtoOLD	Reference table before/after renumbring
NCOLORtot	Final number of colors (g.e. initial number)
COLORindex[NCOLORtot+1]	Meshes from **COLORindex[ic]** to **COLORindex[ic+1]-1** are in (ic+1)-th color.
	Meshes in same color are independent: Parallel processing can be applied.
	(ic)-th color: if color ID starts at 0
	(ic+1)-th color: if color ID starts at 1
	In the program, color ID starts at 1 !!

COLORindex[NCOLORtot+1] Meshes from **COLORindex[ic]** to **COLORindex[ic+1]-1** are in (ic+1)-th color.

Meshes in same color are independent: Parallel processing can be applied.

(ic)-th color: if color ID starts at 0
(ic+1)-th color: if color ID starts at 1
In the program, color ID starts at 1 !!

```
for(ic=1; ic<=NCOLORtot; ic++) {
    for(i=COLORindex[ic-1]+1; i<=COLORindex[ic]; i++) {
        fprintf(fp21, "#new%8d #old%8d  color%8d¥n",
                       i, NEWtoOLD[i-1]+1, ic);
    }
}
```

COLORindex

```
COLOR number        5
  #new       1   #old       1  color       1
  #new       2   #old       3  color       1
  #new       3   #old       6  color       1
  #new       4   #old       8  color       1
  #new       5   #old       9  color       1
  #new       6   #old       2  color       2
  #new       7   #old       4  color       2
  #new       8   #old       5  color       2
  #new       9   #old       7  color       2
  #new      10   #old      10  color       2
  #new      11   #old      11  color       3
  #new      12   #old      13  color       3
  #new      13   #old      16  color       3
  #new      14   #old      12  color       4
  #new      15   #old      14  color       4
  #new      16   #old      15  color       5
```

Modified MC Method

① ONE mesh with minimum value of "degree" is set to "NEW mesh ID= 1", "Color ID= 1", and "counter for color number" is 1.

② Define "*ITEMcou= ICELTOT/NCOLORtot*", where *ITEMcou* is maximum number of meshes in each color.

③ Color *ITEMcou* independent meshes in ascending order according to initial mesh ID.

④ If *ITEMcou* meshes are colored, or no more independent meshes do not exist, add "1" to the "counter for color number", and proceed to the next color.

⑤ Repeat ③ and ④, until all meshes have been colored.

⑥ "Final counter for color" is *NCOLORtotF*. Renumber meshes in ascending orders according to color ID. In each color, numbering is in ascending orders according to initial mesh ID. In each color, new numbering of meshes is continuous.

mc (1/8)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include "mc.h"

int
MC(int N, int NL, int NU, int *INL, int **IAL, int *INU, int **IAU,
        int *NCOLORtot, int *COLORindex, int *NEWtoOLD, int *OLDtoNEW)
{
        int NCOLORk;
        int *IW, *INLw, *INUw;
        int **IALw, **IAUw;
        int INmin, NODmin, ITEMcou;
        int i, j, k, icon, icou, icouK, icoug, icol, ic, ik, jL, jU;
```

mc (2/8)

```
IW = (int *)calloc(N, sizeof(int));
if(IW == NULL) {
        fprintf(stderr, "Error: %s¥n",
        strerror(errno));
              return -1;
}

NCOLORk = *NCOLORtot;

for(i=0; i<N; i++) {
    NEWtoOLD[i] = i;
    OLDtoNEW[i] = i;
}

INmin = N;
NODmin = -1;

for(i=0; i<N; i++) {
    icon = INL[i] + INU[i];
    if (icon < INmin) {
        INmin = icon;
        NODmin = i;
     }
}

OLDtoNEW[NODmin] = 1;
NEWtoOLD[0     ] = NODmin+1;

memset(IW, 0, sizeof(int)*N);
IW[NODmin] = 1;

ITEMcou = N / NCOLORk;
```

IW:
Work array
"Color ID" of each mesh
IW=0 at initial stage

NODmin:
ID of the mesh with minimum value of "degree"

mc (2/8)

```
IW = (int *)calloc(N, sizeof(int));
if(IW == NULL) {
         fprintf(stderr, "Error: %s¥n",
         strerror(errno));
                return -1;
}

NCOLORk = *NCOLORtot;

for(i=0; i<N; i++) {
    NEWtoOLD[i] = i;
    OLDtoNEW[i] = i;
}

INmin = N;
NODmin = -1;

for(i=0; i<N; i++) {
    icon = INL[i] + INU[i];
    if (icon < INmin) {
        INmin = icon;
        NODmin = i;
     }
}

OLDtoNEW[NODmin] = 1;
NEWtoOLD[0     ] = NODmin+1;

memset(IW, 0, sizeof(int)*N);
IW[NODmin] = 1;

ITEMcou = N / NCOLORk;
```

New mesh ID of **NODmin** is set to 0
Color ID of **NODmin** is set to 1

OLDtoNEW[NODmin]= 1
NEWtoOLD[0]= NODmin+1

IW[NODmin]=1: Color ID

mc (2/8)

```
IW = (int *)calloc(N, sizeof(int));
if(IW == NULL) {
         fprintf(stderr, "Error: %s¥n",
         strerror(errno));
                return -1;
}

NCOLORk = *NCOLORtot;

for(i=0; i<N; i++) {
    NEWtoOLD[i] = i;
    OLDtoNEW[i] = i;
}

INmin = N;
NODmin = -1;

for(i=0; i<N; i++) {
    icon = INL[i] + INU[i];
    if (icon < INmin) {
        INmin = icon;
        NODmin = i;
     }
}

OLDtoNEW[NODmin] = 1;
NEWtoOLD[0     ] = NODmin+1;

memset(IW, 0, sizeof(int)*N);
IW[NODmin] = 1;

ITEMcou = N / NCOLORk;
```

ITEMcou= N/NCOLORk:
(Maximum) number of meshes in each color

INPUT=3: MC, 5-Colors

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

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

```
#new     1   #old     1   color     1
#new     2   #old     3   color     1
#new     3   #old     6   color     1
#new     4   #old     8   color     1
#new     5   #old     9   color     1
#new     6   #old     2   color     2
#new     7   #old     4   color     2
#new     8   #old     5   color     2
#new     9   #old     7   color     2
#new    10   #old    10   color     2
#new    11   #old    11   color     3
#new    12   #old    13   color     3
#new    13   #old    16   color     3
#new    14   #old    12   color     4
#new    15   #old    14   color     4
#new    16   #old    15   color     5
```

16/3=5= **ITEMcou**

Finally, 5 colors are needed.
Affected by
revaluation/devaluation

mc (3/8)

Initialization of Counters

icou : Global Counter
icouK: Intra-Color Counter

```
icou  = 1;
icouK = 1;
for(icol=0; icol<N; icol++) {
    NCOLORk = icol + 1;
    for(i=0; i<N; i++) {
        if(IW[i] <= 0) IW[i] = 0;}
           for(i=0; i<N; i++) {
/* already COLORED*/
                if(IW[i] == icol+1) {
                   for(k=0; k<INL[i]; k++) {
                       ik = IAL[i][k];
                       if(IW[ik-1] <= 0) IW[ik-1] = -1;
                   }
                   for(k=0; k<INU[i]; k++) {
                       ik = IAU[i][k];
                       if(IW[ik-1] <= 0) IW[ik-1] = -1;
                   }
                }

/* not COLORED */
                if(IW[i] == 0) {
                   icou++; icouK++;
                   IW[i] = icol + 1;
                   for(k=0; k<INL[i]; k++) {
                       ik = IAL[i][k];
                       if(IW[ik-1] <= 0) IW[ik-1] = -1;
                   }
                   for(k=0; k<INU[i]; k++) {
                       ik = IAU[i][k];
                       if(IW[ik-1] <= 0) IW[ik-1] = -1;
                   }
                }
                if(icou == N) goto N200;
                if(icouK == ITEMcou) goto N100;
    }
N100:
            icouK = 0;
}
N200:
```

mc (3/8)

Loop on Colors

```
icou  = 1;
icouK = 1;
for(icol=0; icol<N; icol++) {
    NCOLORk = icol + 1;
    for(i=0; i<N; i++) {
        if(IW[i] <= 0) IW[i] = 0;}
           for(i=0; i<N; i++) {
/* already COLORED*/
               if(IW[i] == icol+1) {
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }

/* not COLORED */
               if(IW[i] == 0) {
                  icou++; icouK++;
                  IW[i] = icol + 1;
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }
               if(icou == N) goto N200;
               if(icouK == ITEMcou) goto N100;
           }
N100:
           icouK = 0;
}
N200:
```

mc (3/8)

```
icou  = 1;
icouK = 1;
for(icol=0; icol<N; icol++) {
    NCOLORk = icol + 1;
    for(i=0; i<N; i++) {
        if(IW[i] <= 0) IW[i] = 0;}
           for(i=0; i<N; i++) {
/* already COLORED*/
               if(IW[i] == icol+1) {
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }

/* not COLORED */
               if(IW[i] == 0) {
                  icou++; icouK++;
                  IW[i] = icol + 1;
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }
               if(icou == N) goto N200;
               if(icouK == ITEMcou) goto N100;
           }
N100:
           icouK = 0;
}
N200:
```

NCOLORk:
Current number of colors

IW[i]=0:
If i-th mesh (original numbering) is not colored.

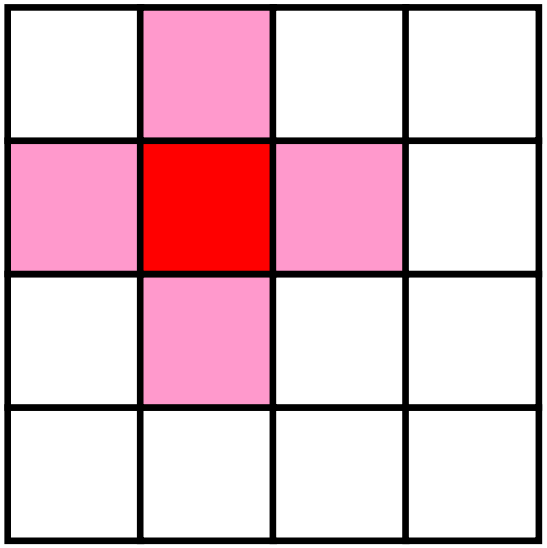
mc (3/8)

```
icou  = 1;
icouK = 1;
for(icol=0; icol<N; icol++) {
    NCOLORk = icol + 1;
    for(i=0; i<N; i++) {
        if(IW[i] <= 0) IW[i] = 0;}
           for(i=0; i<N; i++) {
/* already COLORED*/
               if(IW[i] == icol+1) {
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }

/* not COLORED */
               if(IW[i] == 0) {
                  icou++; icouK++;
                  IW[i] = icol + 1;
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }
               if(icou == N) goto N200;
               if(icouK == ITEMcou) goto N100;
           }
N100:
           icouK = 0;
}
N200:
```

Loop on Colors

If mesh is already assigned to the “current color”, components of **IW** array for adjacent meshes are set to “-1”.

Remove meshes connected to meshes assigned to the “current color”, because they cannot be into the “current color”.



mc (3/8)

```
icou  = 1;
icouK = 1;
for(icol=0; icol<N; icol++) {
    NCOLORk = icol + 1;
    for(i=0; i<N; i++) {
        if(IW[i] <= 0) IW[i] = 0;}
           for(i=0; i<N; i++) {
/* already COLORED*/
               if(IW[i] == icol+1) {
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }

/* not COLORED */
               if(IW[i] == 0) {
                  icou++; icouK++;
                  IW[i] = icol + 1;
                  for(k=0; k<INL[i]; k++) {
                      ik = IAL[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
                  for(k=0; k<INU[i]; k++) {
                      ik = IAU[i][k];
                      if(IW[ik-1] <= 0) IW[ik-1] = -1;
                  }
               }
               if(icou == N) goto N200;
               if(icouK == ITEMcou) goto N100;
           }
N100:
           icouK = 0;
}
N200:
```

if **IW[i]=0:**

- **icou =icou+1**
- **icouK=icouK+1**
- **IW[i]=icol+1:** Color ID

- **IW[ik]=-1** where ***ik-th*** mesh is adjacent to ***i-th*** mesh

mc (3/8)

```
icou  = 1;
icouK = 1;
for(icol=0; icol<N; icol++) {
    NCOLORk = icol + 1;
    for(i=0; i<N; i++) {
        if(IW[i] <= 0) IW[i] = 0;}
        for(i=0; i<N; i++) {
/* already COLORED*/
            if(IW[i] == icol+1) {
               for(k=0; k<INL[i]; k++) {
                   ik = IAL[i][k];
                   if(IW[ik-1] <= 0) IW[ik-1] = -1;
               }
               for(k=0; k<INU[i]; k++) {
                   ik = IAU[i][k];
                   if(IW[ik-1] <= 0) IW[ik-1] = -1;
               }
            }

/* not COLORED */
            if(IW[i] == 0) {
               icou++; icouK++;
               IW[i] = icol + 1;
               for(k=0; k<INL[i]; k++) {
                   ik = IAL[i][k];
                   if(IW[ik-1] <= 0) IW[ik-1] = -1;
               }
               for(k=0; k<INU[i]; k++) {
                   ik = IAU[i][k];
                   if(IW[ik-1] <= 0) IW[ik-1] = -1;
               }
            }
            if(icou == N) goto N200;
            if(icouK == ITEMcou) goto N100;
        }
N100:
        icouK = 0;
}
N200:
```

icou : Global Counter
icouK: Intra-Color Counter

if **icou=N (ICELTOT)**:
- All meshes are colored (completed).

if **icouK=ITEMcou**:
- **icouK=0**
- Proceed to the next color.

if **icouK<ITEMcou.and.i=N**:
- No more independent meshes.
- Proceed to the next color.

INPUT=3: MC, 5-Colors

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

5					
	3		4		
1		2			

```
#new      1   #old      1   color      1
#new      2   #old      3   color      1
#new      3   #old      6   color      1
#new      4   #old      8   color      1
#new      5   #old      9   color      1
#new      6   #old      2   color      2
#new      7   #old      4   color      2
#new      8   #old      5   color      2
#new      9   #old      7   color      2
#new     10   #old     10   color      2
#new     11   #old     11   color      3
#new     12   #old     13   color      3
#new     13   #old     16   color      3
#new     14   #old     12   color      4
#new     15   #old     14   color      4
#new     16   #old     15   color      5
```

16/3=5= **ITEMcou**

5 independent meshes

INPUT=3: MC, 5-Colors

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

5	10				
8	3	9	4		
1	6	2	7		

```
#new      1   #old      1   color      1
#new      2   #old      3   color      1
#new      3   #old      6   color      1
#new      4   #old      8   color      1
#new      5   #old      9   color      1
#new      6   #old      2   color      2
#new      7   #old      4   color      2
#new      8   #old      5   color      2
#new      9   #old      7   color      2
#new     10   #old     10   color      2
#new     11   #old     11   color      3
#new     12   #old     13   color      3
#new     13   #old     16   color      3
#new     14   #old     12   color      4
#new     15   #old     14   color      4
#new     16   #old     15   color      5
```

16/3=5= **ITEMcou**

5 independent meshes

INPUT=3: MC, 5-Colors

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

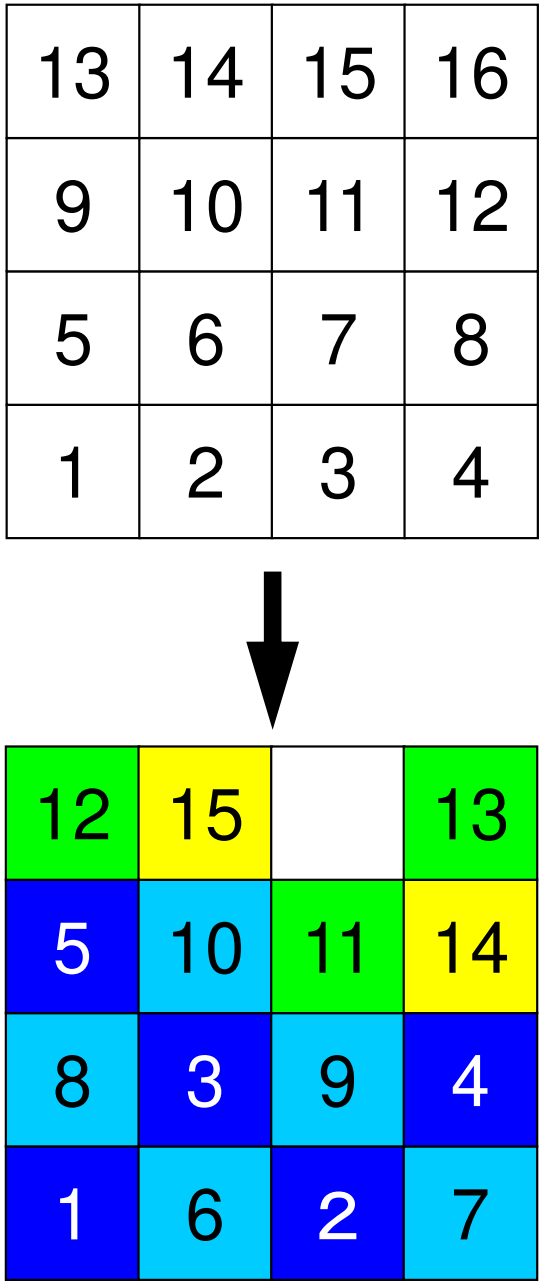
12			13		
5	10	11			
8	3	9	4		
1	6	2	7		

```
#new       1   #old       1   color       1
#new       2   #old       3   color       1
#new       3   #old       6   color       1
#new       4   #old       8   color       1
#new       5   #old       9   color       1
#new       6   #old       2   color       2
#new       7   #old       4   color       2
#new       8   #old       5   color       2
#new       9   #old       7   color       2
#new      10   #old      10   color       2
#new      11   #old      11   color       3
#new      12   #old      13   color       3
#new      13   #old      16   color       3
#new      14   #old      12   color       4
#new      15   #old      14   color       4
#new      16   #old      15   color       5
```

16/3=5= **ITEMcou**

Proceed to the next color, if no more independent meshes.

INPUT=3: MC, 5-Colors




```
#new      1   #old      1   color      1
#new      2   #old      3   color      1
#new      3   #old      6   color      1
#new      4   #old      8   color      1
#new      5   #old      9   color      1
#new      6   #old      2   color      2
#new      7   #old      4   color      2
#new      8   #old      5   color      2
#new      9   #old      7   color      2
#new     10   #old     10   color      2
#new     11   #old     11   color      3
#new     12   #old     13   color      3
#new     13   #old     16   color      3
#new     14   #old     12   color      4
#new     15   #old     14   color      4
#new     16   #old     15   color      5
```

16/3=5= **ITEMcou**

Proceed to the next color, if no more independent meshes.

mc (5/8)

```
/******************
 * Final Coloring *
 ******************/

*NCOLORtot = NCOLORk;
memset(COLORindex, 0, sizeof(int)*(N+1));

icoug = 0;
for(ic=1; ic<=(*NCOLORtot); ic++) {
    icou = 0;
    for(i=0; i<N; i++) {
        if(IW[i] == ic) {
           NEWtoOLD[icoug] = i+1;
           OLDtoNEW[i]     = icoug+1;
           icou++;
           icoug++;
         }
    }
    COLORindex[ic] = icou;
}

for(ic=1; ic<=(*NCOLORtot); ic++) {
    COLORindex[ic] += COLORindex[ic-1];
}
```

NCOLORtot= NCOLORk:
Final number of colors.

NCOLORtot g.e. (Initial number of colors provided by user)

mc (5/8)

```
/******************
 * Final Coloring *
 ******************/

*NCOLORtot = NCOLORk;
memset(COLORindex, 0, sizeof(int)*(N+1));

icoug = 0;
for(ic=1; ic<=(*NCOLORtot); ic++) {
    icou = 0;
    for(i=0; i<N; i++) {
        if(IW[i] == ic) {
           NEWtoOLD[icoug] = i+1;
           OLDtoNEW[i]     = icoug+1;
           icou++;
           icoug++;
         }
    }
    COLORindex[ic] = icou;
}

for(ic=1; ic<=(*NCOLORtot); ic++) {
    COLORindex[ic] += COLORindex[ic-1];
}
```

Renumber meshes in ascending orders according to color ID.

OLDtoNEW[OldID]= NewID+1
NEWtoOLD[NewID]= OldID+1
starting@”1”

COLODindex[ic]:
At this stage, number of meshes in each color (ic+1) is stored.

mc (6/8)

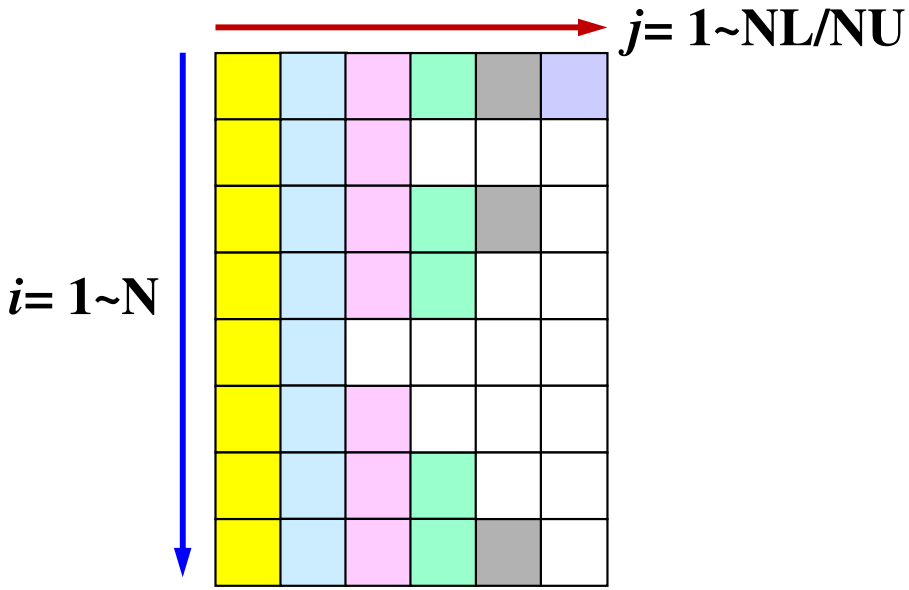
```
/******************
 * Final Coloring *
 ******************/

*NCOLORtot = NCOLORk;
memset(COLORindex, 0, sizeof(int)*(N+1));

icoug = 0;
for(ic=1; ic<=(*NCOLORtot); ic++) {
    icou = 0;
    for(i=0; i<N; i++) {
        if(IW[i] == ic) {
           NEWtoOLD[icoug] = i+1;
           OLDtoNEW[i]     = icoug+1;
           icou++;
           icoug++;
         }
    }
    COLORindex[ic] = icou;
}

for(ic=1; ic<=(*NCOLORtot); ic++) {
    COLORindex[ic] += COLORindex[ic-1];
}
```

COLODindex[ic]:

Now it is 1D index.

mc (6/8)

```
INLw = (int *)calloc(N, sizeof(int));
if(INLw == NULL) {
            fprintf(stderr, "Error: %s¥n", strerror(errno));
            return -1;
}...

for(j=0; j<NL; j++) {
    for(i=0; i<N; i++) {
        IW[i] = IAL[NEWtoOLD[i]-1][j];
    }
    for(i=0; i<N; i++) {
        IAL[i][j] = IW[i];
    }
}

for(j=0; j<NU; j++) {
    for(i=0; i<N; i++) {
        IW[i] = IAU[NEWtoOLD[i]-1][j];
    }
    for(i=0; i<N; i++) {
        IAU[i][j] = IW[i];
    }
}
```



Arrays for Work

- **INLw(N)**
- **INUw(N)**
- **IALw(NL,N)**
- **IAUw(NU,N)**

Lower/upper components (column ID) are reordered according to new numbering.

ID's of column ID are by old numbering.

mc (7/8)

```
for(i=0; i<N; i++) {
    IW[i] = INL[NEWtoOLD[i]-1];
}
for(i=0; i<N; i++) {
    INLw[i] = IW[i];
}
for(i=0; i<N; i++) {
    IW[i] = INU[NEWtoOLD[i]-1];
}
for(i=0; i<N; i++) {
    INUw[i] = IW[i];
}

for(j=0; j<NL; j++) {
    for(i=0; i<N; i++) {
        if(IAL[i][j] == 0) {
           IALw[i][j] = 0;
        } else {
           IALw[i][j] = OLDtoNEW[IAL[i][j]-1];
        }
    }
}

for(j=0; j<NU; j++) {
    for(i=0; i<N; i++) {
        if(IAU[i][j] == 0) {
           IAUw[i][j] = 0;
        } else {
           IAUw[i][j] = OLDtoNEW[IAU[i][j]-1];
        }
    }
}
```

Information for number of lower/upper components based on new numbering is stored into:

- **INL -> INLw**
- **INU -> INUw**

mc (7/8)

```
for(i=0; i<N; i++) {
    IW[i] = INL[NEWtoOLD[i]-1];
}
for(i=0; i<N; i++) {
    INLw[i] = IW[i];
}
for(i=0; i<N; i++) {
    IW[i] = INU[NEWtoOLD[i]-1];
}
for(i=0; i<N; i++) {
    INUw[i] = IW[i];
}

for(j=0; j<NL; j++) {
    for(i=0; i<N; i++) {
        if(IAL[i][j] == 0) {
           IALw[i][j] = 0;
        } else {
           IALw[i][j] = OLDtoNEW[IAL[i][j]-1];
        }
    }
}

for(j=0; j<NU; j++) {
    for(i=0; i<N; i++) {
        if(IAU[i][j] == 0) {
           IAUw[i][j] = 0;
        } else {
           IAUw[i][j] = OLDtoNEW[IAU[i][j]-1];
        }
    }
}
```

"Renumbered" lower/upper components (column ID) are stored into:

- **IAL -> IALw**
- **IAU -> IAUw**

mc (8/8)

```
memset(INL, 0, sizeof(int)*N);
memset(INU, 0, sizeof(int)*N);
for(i=0; i<N; i++) {
    memset(IAL[i], 0, sizeof(int)*NL);}
for(i=0; i<N; i++) {
    memset(IAU[i], 0, sizeof(int)*NU);}

for(i=0; i<N; i++) {
    jL = 0;
    jU = 0;
    for(j=0; j<INLw[i]; j++) {
        if(IALw[i][j] > i+1) {
           IAU[i][jU] = IALw[i][j];
           jU++;
        } else {
           IAL[i][jL] = IALw[i][j];
           jL++;
        }
     }

     for(j=0; j<INUw[i]; j++) {
         if(IAUw[i][j] > i+1) {
            IAU[i][jU] = IAUw[i][j];
            jU++;
         } else {
            IAL[i][jL] = IAUw[i][j];
            jL++;
         }
      }
      INL[i] = jL;
      INU[i] = jU;
}
        free(IW);
        free(INLw);          free(INUw);
        free(IALw);          free(IAUw);

        return 0;
}
```

Operation for lower triangular components (column ID) in the original matrix. “Renumbered” components (column ID) are stored into:

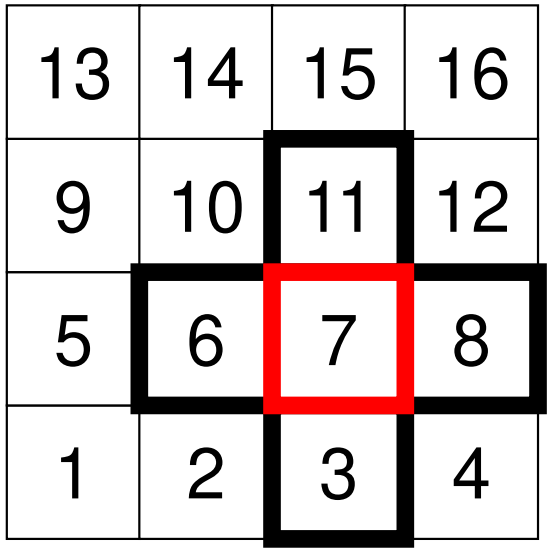
- **IALw -> IAL,IAU**

```
for(i=0; i<N; i++) {
    jL = 0;
    jU = 0;
    for(j=0; j<INLw[i]; j++) {
        if(IALw[i][j] > i+1) {
           IAU[i][jU] = IALw[i][j];
           jU++;
        } else {
           IAL[i][jL] = IALw[i][j];
           jL++;
        }
     }
}
```

Because **IALw[i][j]** could be larger than **i** according to new numbering.

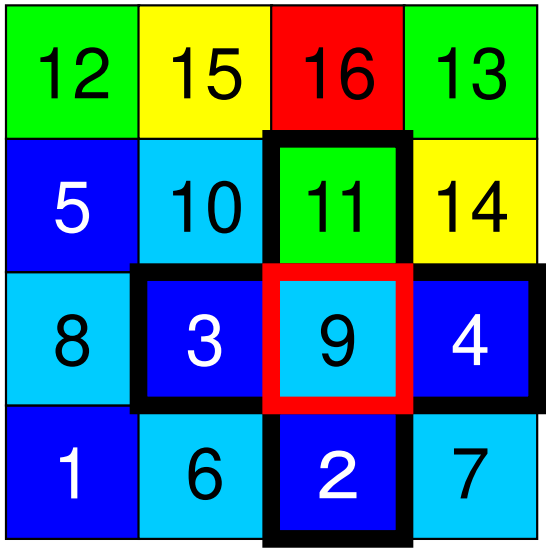
Why do we need these operations ?

Original




```
INL[7-1]= 2
IAL[6][0]= 3,  IAL[6][1]= 6

INU[7-1]= 2
IAU[6][0]= 8,  IAU[6][1]=11
```

5 Color




```
INL[9-1]= 3
IAL[8][0]= 2,  IAL[8][1]= 3
IAL[8][2]= 4

INU[9-1]= 1
IAU[8][0]=11
```

Magnitude correlation with adjacent meshes could change after renumbering.

mc (8/8)

```
memset(INL, 0, sizeof(int)*N);
memset(INU, 0, sizeof(int)*N);
for(i=0; i<N; i++) {
    memset(IAL[i], 0, sizeof(int)*NL);}
for(i=0; i<N; i++) {
    memset(IAU[i], 0, sizeof(int)*NU);}

for(i=0; i<N; i++) {
    jL = 0;
    jU = 0;
    for(j=0; j<INLw[i]; j++) {
        if(IALw[i][j] > i+1) {
           IAU[i][jU] = IALw[i][j];
           jU++;
        } else {
           IAL[i][jL] = IALw[i][j];
           jL++;
        }
     }

     for(j=0; j<INUw[i]; j++) {
         if(IAUw[i][j] > i+1) {
            IAU[i][jU] = IAUw[i][j];
            jU++;
         } else {
            IAL[i][jL] = IAUw[i][j];
            jL++;
         }
      }
      INL[i] = jL;
      INU[i] = jU;
}
        free(IW);
        free(INLw);          free(INUw);
        free(IALw);          free(IAUw);

        return 0;
}
```

Operation for upper triangular components (column ID) in the original matrix. “Renumbered” components (column ID) are stored into:

- **IAUw -> IAL, IAU**

mc (8/8)

```
memset(INL, 0, sizeof(int)*N);
memset(INU, 0, sizeof(int)*N);
for(i=0; i<N; i++) {
    memset(IAL[i], 0, sizeof(int)*NL);}
for(i=0; i<N; i++) {
    memset(IAU[i], 0, sizeof(int)*NU);}

for(i=0; i<N; i++) {
    jL = 0;
    jU = 0;
    for(j=0; j<INLw[i]; j++) {
        if(IALw[i][j] > i+1) {
           IAU[i][jU] = IALw[i][j];
           jU++;
        } else {
           IAL[i][jL] = IALw[i][j];
           jL++;
        }
     }

     for(j=0; j<INUw[i]; j++) {
         if(IAUw[i][j] > i+1) {
            IAU[i][jU] = IAUw[i][j];
            jU++;
         } else {
            IAL[i][jL] = IAUw[i][j];
            jL++;
         }
      }
      INL[i] = jL;
      INU[i] = jU;
}
        free(IW);
        free(INLw);          free(INUw);
        free(IALw);          free(IAUw);

        return 0;
}
```

"Final" number of upper/lower components (column ID) in the renumbered new matrix.

- **INL**
- **INU**

Modified CM Method for Parallel Computing

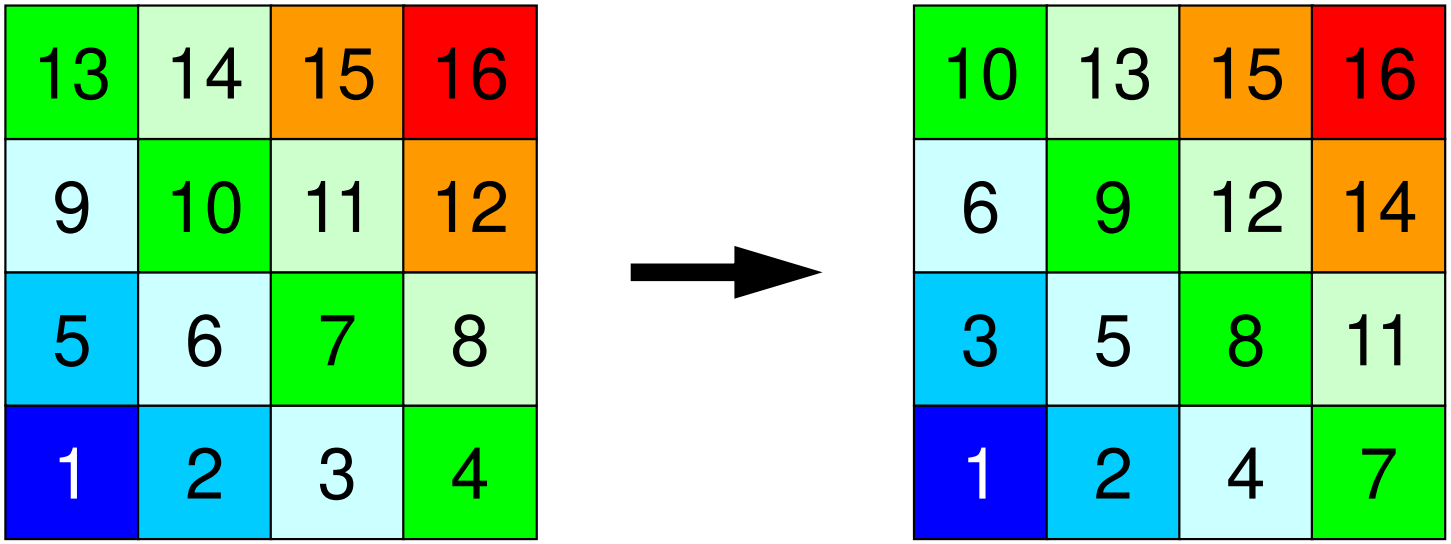
① ONE mesh with minimum value of “degree” is set to “Level=1”.

② Meshes adjacent to “Level=k-1” meshes are set to “Level=k”. In each level, meshes must be independent (not directly connected). If a dependent pair is found in same color, one mesh is removed (In current implementation, a mesh found later is removed).

③ Repeat ②, until all meshes are flagged to “levels”

④ Renumber meshes in ascending orders according to “Level” ID. In each level, numbering is in ascending orders according to initial mesh ID. In each level, new numbering of meshes is continuous.

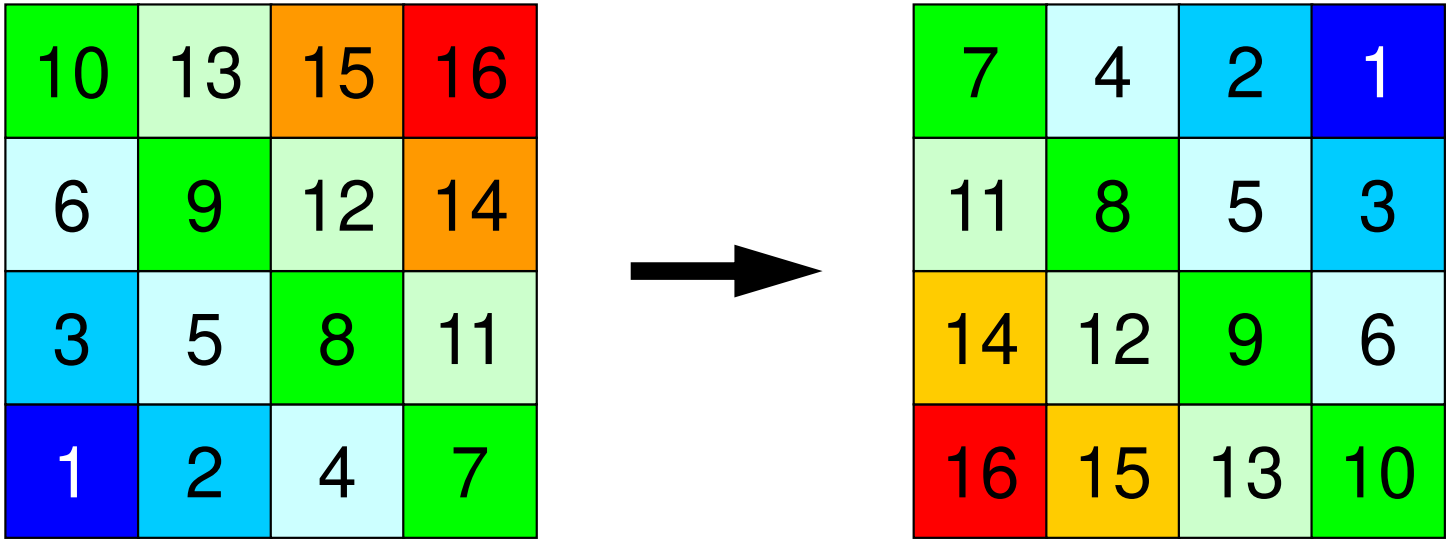
Procedure of CM Method



- Meshes adjacent to “Level=k-1” meshes are set to “Level=k” (Repeat until all meshes are flagged into “levels”)
  - In each level, meshes must be independent (not directly connected). If a dependent pair is found in same color, one mesh is removed (In current implementation, a mesh found later is removed).
- Renumber meshes in ascending orders according to “Level” ID.

RCM: Reverse Cuthill-McKee

- Do operations for “CM” method
  - Calculate “degree” at each mesh
  - Flag “level k (1,2,...)” to meshes
  - Repeat processes, final renumbering
- Renumbering Again
  - Renumber meshes reordered by CM method in reverse order.
  - Fill-in’s are fewer than CM



cm (1/5)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#include "rcm.h"

extern int
CM(int N, int NL, int NU, int *INL, int **IAL, int *INU, int **IAU,
   int *NCOLORtot, int *COLORindex, int *NEWtoOLD, int *OLDtoNEW)
{
        int **IW;
        int *INLw, *INUw;
        int **IALw, **IAUw;
        int KC, KCT, KCTO, KMIN, KMAX;
        int JC, JN;
        int II;
        int i, j, k, ic, jL, jU;
        int INmin, NODmin;
        int icon, icol, icouG, icou, icou0, in, inC;
```

cm (2/5)

```
IW = (int **)calloc(2, sizeof(int *));
if(IW == NULL) {
   fprintf(stderr, "Error: %s¥n", strerror(errno));
   return -1;
}

for(i=0; i<2; i++) {
    IW[i] = (int *)calloc(N, sizeof(int));
}

INmin  = N;
NODmin = -1;

for(i=0; i<N; i++) {
    icon = 0;
    for(k=0; k<INL[i]; k++) {
        icon++;
    }
    for(k=0; k<INU[i]; k++) {
        icon++;
    }

    if(icon < INmin) {
       INmin  = icon;
       NODmin = i;
    }
}

if(NODmin == -1) NODmin = 0;

IW[1][NODmin] = 1;

NEWtoOLD[0]      = NODmin+1;
OLDtoNEW[NODmin] = 1;

icol = 0;
```

IW[0][i]:
Work array

IW[1][i]:
“Level ID” of each mesh starting at “1”

cm (2/5)

```
IW = (int **)calloc(2, sizeof(int *));
if(IW == NULL) {
   fprintf(stderr, "Error: %s¥n", strerror(errno));
   return -1;
}

for(i=0; i<2; i++) {
    IW[i] = (int *)calloc(N, sizeof(int));
}

INmin  = N;
NODmin = -1;

for(i=0; i<N; i++) {
    icon = 0;
    for(k=0; k<INL[i]; k++) {
        icon++;
    }
    for(k=0; k<INU[i]; k++) {
        icon++;
    }

    if(icon < INmin) {
       INmin  = icon;
       NODmin = i;
    }
}

if(NODmin == -1) NODmin = 0;

IW[1][NODmin] = 1;

NEWtoOLD[0]      = NODmin+1;
OLDtoNEW[NODmin] = 1;

icol = 0;
```

NODmin:
ID of the mesh with minimum value of “degree”

cm (2/5)

```
IW = (int **)calloc(2, sizeof(int *));
if(IW == NULL) {
   fprintf(stderr, "Error: %s¥n", strerror(errno));
   return -1;
}

for(i=0; i<2; i++) {
    IW[i] = (int *)calloc(N, sizeof(int));
}

INmin  = N;
NODmin = -1;

for(i=0; i<N; i++) {
    icon = 0;
    for(k=0; k<INL[i]; k++) {
        icon++;
    }
    for(k=0; k<INU[i]; k++) {
        icon++;
    }

    if(icon < INmin) {
       INmin  = icon;
       NODmin = i;
    }
}

if(NODmin == -1) NODmin = 0;

IW[1][NODmin] = 1;

NEWtoOLD[0]      = NODmin+1;
OLDtoNEW[NODmin] = 1;

icol = 0;
```

New mesh ID of **NODmin**
is set to 0.
Level ID of **NODmin** is set to 1

OLDtoNEW[NODmin]= 1
NEWtoOLD[0]= NODmin+1

IW[1][NODmin]=1: Level ID

cm (3/5)

```
icouG = 1;
for(icol=1; icol<N; icol++) {
    icou  = 0;
    icou0 = 0;
    for(i=0; i<N; i++) {
        if(IW[1][i] == icol) {
           for(k=0; k<INL[i]; k++) {
               in = IAL[i][k];
               if(IW[1][in-1] == 0) {
                  IW[1][in-1] = -(icol + 1);
                  icou++;
                  IW[0][icou-1] = in;
               }
           }
           for(k=0; k<INU[i]; k++) {
               in = IAU[i][k];
               if(IW[1][in-1] == 0) {
                  IW[1][in-1] = -(icol + 1);
                  icou++;
                  IW[0][icou-1] = in;
               }
           }
        }
    }

    if(icou == 0) {
       for(i=0; i<N; i++) {
           if(IW[1][i] == 0) {
              icou++;
              IW[1][i     ] = -(icol + 1);
              IW[0][icou-1] = i + 1;
             break;
           }
        }
     }
```

`icouG`: Global Counter
`icou` : Intra-Level Counter

Loop on Levels

cm (3/5)

```
icouG = 1;
for(icol=1; icol<N; icol++) {
    icou  = 0;
    icou0 = 0;
    for(i=0; i<N; i++) {
        if(IW[1][i] == icol) {
           for(k=0; k<INL[i]; k++) {
               in = IAL[i][k];
               if(IW[1][in-1] == 0) {
                  IW[1][in-1] = -(icol + 1);
                  icou++;
                  IW[0][icou-1] = in;
               }
           }
           for(k=0; k<INU[i]; k++) {
               in = IAU[i][k];
               if(IW[1][in-1] == 0) {
                  IW[1][in-1] = -(icol + 1);
                  icou++;
                  IW[0][icou-1] = in;
               }
           }
        }
    }

    if(icou == 0) {
       for(i=0; i<N; i++) {
           if(IW[1][i] == 0) {
              icou++;
              IW[1][i     ] = -(icol + 1);
              IW[0][icou-1] = i + 1;
              break;
           }
        }
     }
```

icouG: Global Counter
icou : Intra-Level Counter

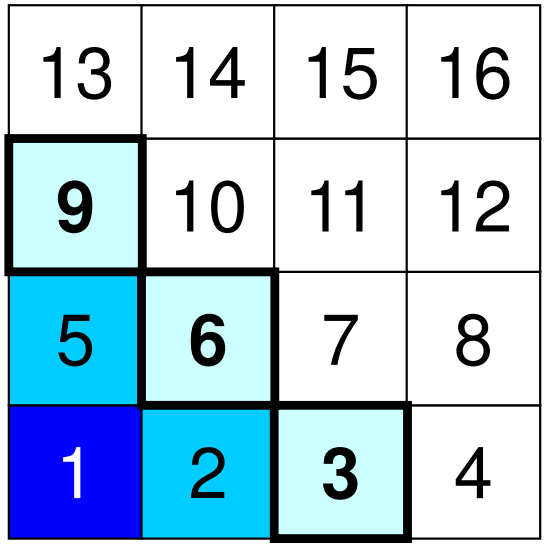
Loops for Each Element

If (**in-1**)th mesh is adjacent to **i**th mesh where **IW[1][i]=icol**, and level of (**in-1**)th mesh is not finalized, (**in-1**)th mesh could be a candidate for meshes in (**icol+1**)th level.

- **IW[1][in-1]= -(icol+1)**
- **icou= icou + 1**
- **IW[0][icou-1]= in:** This array indicates “when” this mesh was found, and nominated as a candidate.

mesh, color: starting from 0
IAL[:][:]: starting from 1

What does it mean ?



icol+1=4
IW[1][i]= icol=3: i=3,6,9

icouG: Global Counter
icou : Intra-Level Counter

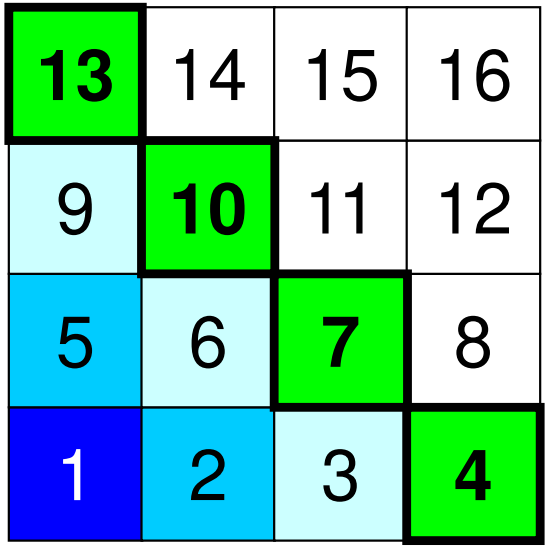
Loops for Each Element

If (**in-1**)th mesh is adjacent to **i**th mesh where **IW[1][i]=icol**, and level of (**in-1**)th mesh is not finalized, (**in-1**)th mesh could be a candidate for meshes in (**icol+1**)th level.

- **IW[1][in-1]= -(icol+1)**
- **icou= icou + 1**
- **IW[0][icou-1]= in:** This array indicates “when” this mesh was found, and nominated as a candidate.

mesh, color: starting from 0
IAL[:][:]: starting from 1

What does it mean ?




```
icol+1=4
 IW[1][i]= icol=3: i=3,6,9
```

```
IW[1][ 3(= 4-1)]= -4
IW[1][ 6(= 7-1)]= -4
IW[1][ 9(=10-1)]= -4
IW[1][12(=13-1)]= -4

IW[0][0]=  4
IW[0][1]=  7
IW[0][2]= 10
IW[0][3]= 13
```

icouG: Global Counter
icou : Intra-Level Counter

Loops for Each Element

If (**in-1**)th mesh is adjacent to **i**th mesh where **IW[1][i]=icol**, and level of (**in-1**)th mesh is not finalized, (**in-1**)th mesh could be a candidate for meshes in (**icol+1**)th level.

- **IW[1][in-1]= -(icol+1)**
- **icou= icou + 1**
- **IW[0][icou-1]= in:** This array indicates “when” this mesh was found, and nominated as a candidate.

mesh, color: starting from 0
IAL[:][:]: starting from 1

cm (3/5)

```
icouG = 1;
for(icol=1; icol<N; icol++) {
    icou  = 0;
    icou0 = 0;
    for(i=0; i<N; i++) {
        if(IW[1][i] == icol) {
           for(k=0; k<INL[i]; k++) {
               in = IAL[i][k];
               if(IW[1][in-1] == 0) {
                  IW[1][in-1] = -(icol + 1);
                  icou++;
                  IW[0][icou-1] = in;
               }
           }
           for(k=0; k<INU[i]; k++) {
               in = IAU[i][k];
               if(IW[1][in-1] == 0) {
                  IW[1][in-1] = -(icol + 1);
                  icou++;
                  IW[0][icou-1] = in;
               }
           }
        }
    }

    if(icou == 0) {
       for(i=0; i<N; i++) {
           if(IW[1][i] == 0) {
              icou++;
              IW[1][i     ] = -(icol + 1);
              IW[0][icou-1] = i + 1;
              break;
           }
        }
     }
```

`icouG`: Global Counter
`icou` : Intra-Level Counter

Loops for Each Element

If **`icou=0`**, a mesh, which satisfies the following conditions, is the candidate (usually, this does not happen):

- Level of this mesh is not finalized.
- Mesh ID according to the initial numbering is the smallest.

cm (4/5)

```
for(icol=1; icol<N; icol++) {
....

for(ic=0; ic<icou; ic++) {
    inC = IW[0][ic];
    if(IW[1][inC-1] != 0) {
       for(k=0; k<INL[inC-1]; k++) {
           in = IAL[inC-1][k];
           if(IW[1][in-1] <= 0) {
              IW[1][in-1] = 0;
           }
       }
       for(k=0; k<INU[inC-1]; k++) {
           in = IAU[inC-1][k];
           if(IW[1][in-1] <= 0) {
              IW[1][in-1] = 0;
           }
       }
    }
}

for(ic=0; ic<icou; ic++) {
    inC = IW[0][ic];
    if(IW[1][inC-1] != 0) {
       icouG++;
       IW[1][inC-1] = icol + 1;
    }
}

if(icouG == N) break;
}
```

Candidates for **icolth** level are stored in:

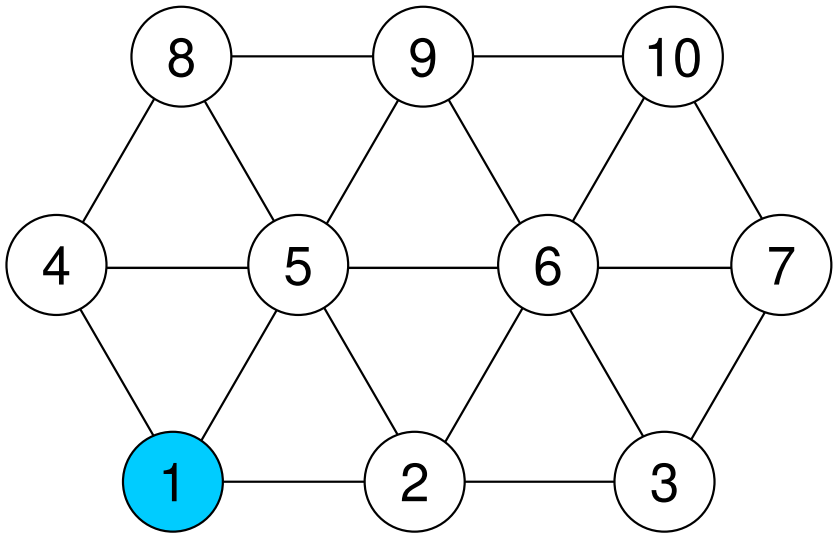
- **IW[0][ic], ic= 0~icou**

Remove such meshes that are adjacent to other candidates, because neighboring meshes cannot belong to same level.

If we have such removed mesh **in-1**, apply the following operations:

- **IW[1][in-1]= 0**

What does is mean ?



e.g.
Mesh (1) belongs to **(icol)th** level

Candidates for (**icol+1)th** level are stored in:

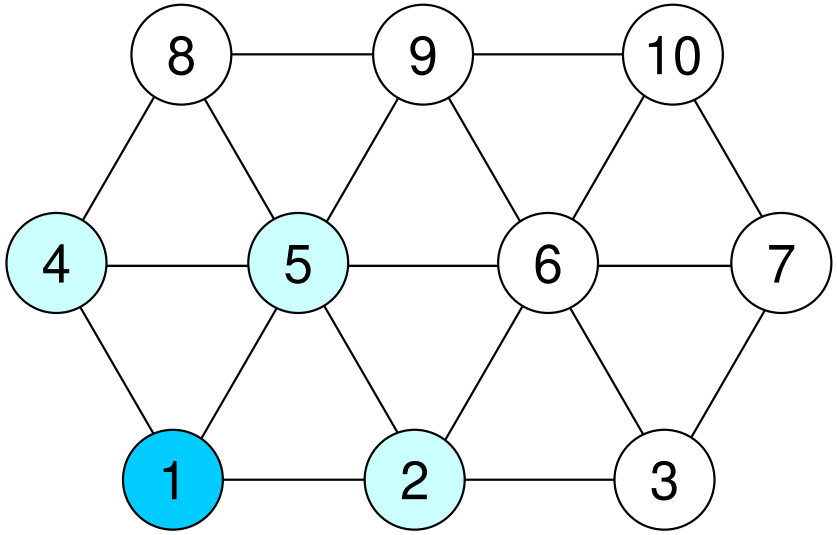
- **IW[0][ic], ic= 0~icou**

Remove such meshes that are adjacent to other candidates, because neighboring meshes cannot belong to same level.

If we have such removed mesh **in-1**, apply the following operations:

- **IW[1][in-1]= 0**

What does is mean ?



(2),(4) and (5) are candidates for **(icol+1)th** level

```
IW[1][2-1]= -(icol+1)
IW[1][4-1]= -(icol+1)
IW[1][5-1]= -(icol+1)

IW[1][0]= 2
IW[1][1]= 4
IW[1][2]= 5
```

Candidates for (**icol+1)th** level are stored in:

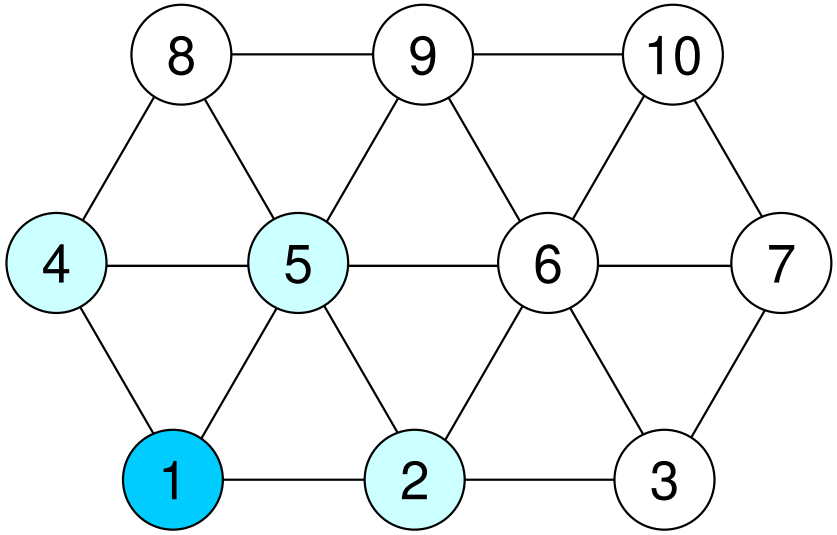
- **IW[0][ic], ic= 0~icou**

Remove such meshes that are adjacent to other candidates, because neighboring meshes cannot belong to same level.

If we have such removed mesh **in-1**, apply the following operations:

- **IW[1][in-1]= 0**

What does is mean ?



Considering dependency:

```
IW[1][2-1]= -(icol+1)
IW[1][4-1]= -(icol;1)
IW[1][5-1]= 0

IW[1][0]= 2
IW[1][1]= 4
IW[1][2]= 5
```

Candidates for (**icol+1)th** level are stored in:

- **IW[0][ic], ic= 0~icou**

Remove such meshes that are adjacent to other candidates, because neighboring meshes cannot belong to same level.

If we have such removed mesh **in-1**, apply the following operations:

- **IW[1][in-1]= 0**

cm (4/5)

```
for(icol=1; icol<N; icol++) {
....

for(ic=0; ic<icou; ic++) {
    inC = IW[0][ic];
    if(IW[1][inC-1] != 0) {
       for(k=0; k<INL[inC-1]; k++) {
            in = IAL[inC-1][k];
            if(IW[1][in-1] <= 0) {
               IW[1][in-1] = 0;
            }
       }
       for(k=0; k<INU[inC-1]; k++) {
            in = IAU[inC-1][k];
            if(IW[1][in-1] <= 0) {
               IW[1][in-1] = 0;
            }
       }
    }
}

for(ic=0; ic<icou; ic++) {
    inC = IW[0][ic];
    if(IW[1][inC-1] != 0) {
       icouG++;
       IW[1][inC-1] = icol + 1;
    }
}

if(icouG == N) break;
}
```

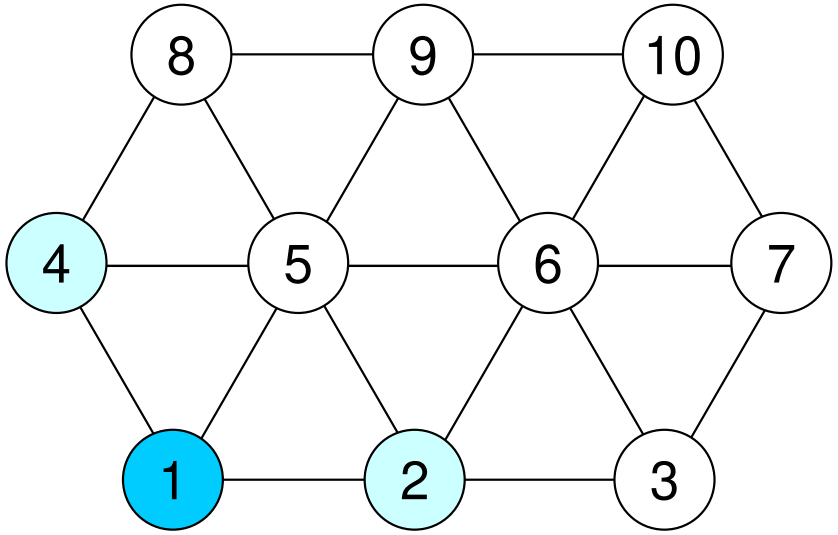
icouG: Global Counter
icou : Intra-Level Counter

Finally, meshes which satisfies **IW[1][inC-1]=-(icol+1)**, belong to **(icol+1)th** level.

For such meshes, apply **IW[1][inC-1]= icol + 1**.

Finally, **icouG= icouG + 1**

What does is mean ?



Considering dependency:

```
IW[1][2-1]= -(icol+1)
IW[1][4-1]= -(icol+1)
IW[1][5-1]= 0
IW[1][0]= 2
IW[1][1]= 4
IW[1][2]= 5
```

Finally:

```
IW[1][2-1]= icol+1
IW[1][4-1]= icol+1
```

Finally, meshes which satisfies **`IW[1][inC-1]=-(icol+1)`**, belong to (**`icol+1)`**th level.

For such meshes, apply **`IW[1][inC-1]= icol + 1`**.

Finally, **`icouG= icouG + 1`**

cm (4/5)

```
for(icol=1; icol<N; icol++) {
....

for(ic=0; ic<icou; ic++) {
     inC = IW[0][ic];
     if(IW[1][inC-1] != 0) {
        for(k=0; k<INL[inC-1]; k++) {
             in = IAL[inC-1][k];
             if(IW[1][in-1] <= 0) {
                IW[1][in-1] = 0;
             }
        }
        for(k=0; k<INU[inC-1]; k++) {
             in = IAU[inC-1][k];
             if(IW[1][in-1] <= 0) {
                IW[1][in-1] = 0;
             }
        }
     }
}

for(ic=0; ic<icou; ic++) {
     inC = IW[0][ic];
     if(IW[1][inC-1] != 0) {
        icouG++;
        IW[1][inC-1] = icol + 1;
     }
}

if(icouG == N) break;
}
```

icouG: Global Counter
icou : Intra-Level Counter

if **icouG=N (ICELTOT)**:
- All meshes are colored (completed).

Otherwise, proceed to the next level.

cm (5/5)

```
/*******************/
/* FINAL COLORING */
/*******************/

*NCOLORtot = icol + 1;
icouG = 0;

for(ic=1; ic<=(*NCOLORtot); ic++) {
    icou = 0;
    for(i=0; i<N; i++) {
        if(IW[1][i] == ic) {
           NEWtoOLD[icouG] = i+1;
           OLDtoNEW[i    ] = icouG+1;
           icou++;
           icouG++;
        }
    }
    COLORindex[ic] = icou;
}

COLORindex[0] = 0;

for(ic=1; ic<=(*NCOLORtot); ic++) {
    COLORindex[ic] = COLORindex[ic-1] +
                     COLORindex[ic];
}

/*******************
 * MATRIX transfer *
 *******************/
```

NCOLORtot= NCOLORk:
Final number of colors.

NCOLORtot g.e. (Initial number of colors provided by user)

Renumber meshes in ascending orders according to level ID.

OLDtoNEW[OldID]= NewID+1
NEWtoOLD[NewID]= OldID+1

COLODindex[ic]:
At this stage, number of meshes in each level (ic+1) is stored.

cm (5/5)

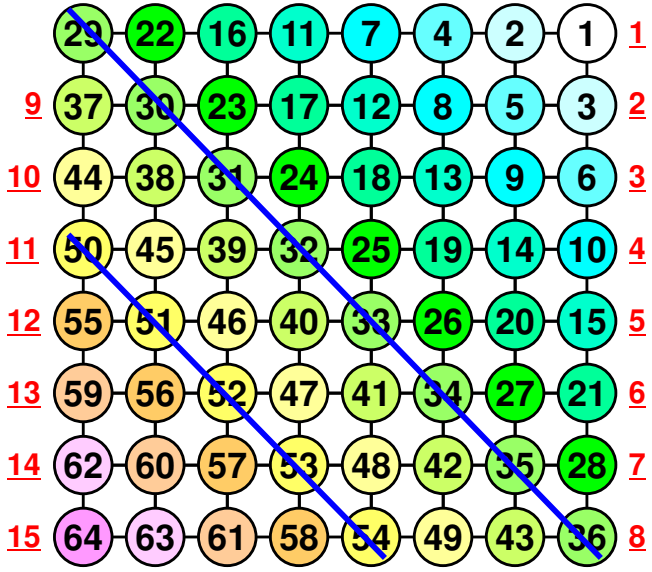
```
/******************/
/* FINAL COLORING */
/******************/

*NCOLORtot = icol + 1;
icouG = 0;

for(ic=1; ic<=(*NCOLORtot); ic++) {
    icou = 0;
    for(i=0; i<N; i++) {
        if(IW[1][i] == ic) {
           NEWtoOLD[icouG] = i+1;
           OLDtoNEW[i    ] = icouG+1;
           icou++;
           icouG++;
        }
    }
    COLORindex[ic] = icou;
}

COLORindex[0] = 0;

for(ic=1; ic<=(*NCOLORtot); ic++) {
    COLORindex[ic] = COLORindex[ic-1] +
                     COLORindex[ic];
}

/********************
 * MATRIX transfer *
 ********************/
```

COLODindex[ic]:

Now it is 1D index.

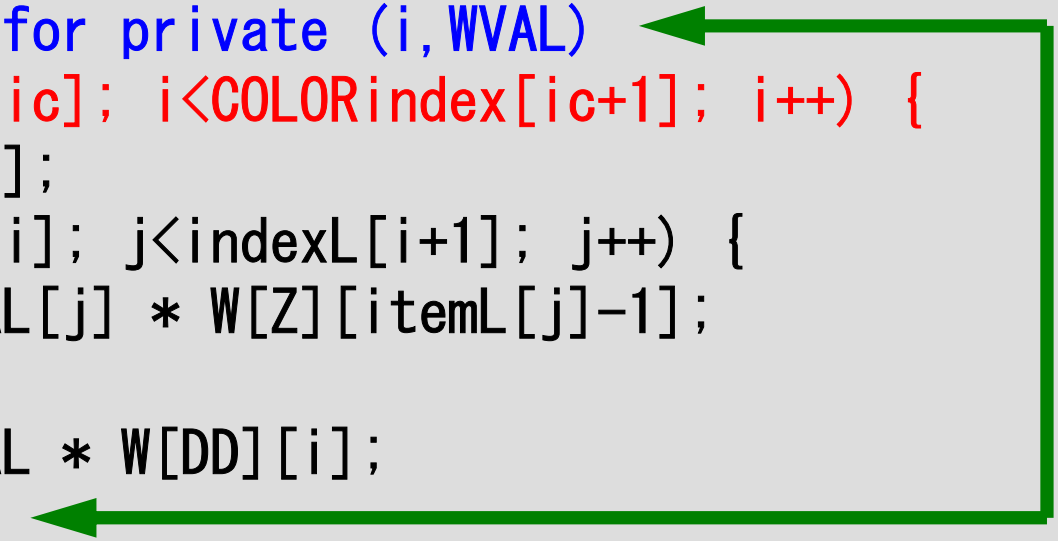
Features of MC & CM/RCM

- MC
  - Good parallel performance & load balancing
  - More Colors -> Better Convergence
    - Smaller number of meshes per color, and per thread
    - Small Granularity（粒度）, Larger Synchronization Overhead
    - Finally, lower parallel performance
- CM/RCM
  - Faster convergence than MC.
  - Generally, many levels (and number is unknown before computation)
    - Same problems on parallel efficiency as MC
    - Number of meshes in each level is random
      - At the 1st/Last Level: Only One Mesh/Level
- Convenient method needed
  - Fast convergence
  - Low overhead with smaller number of colors



More Colors: Synch. Overhead

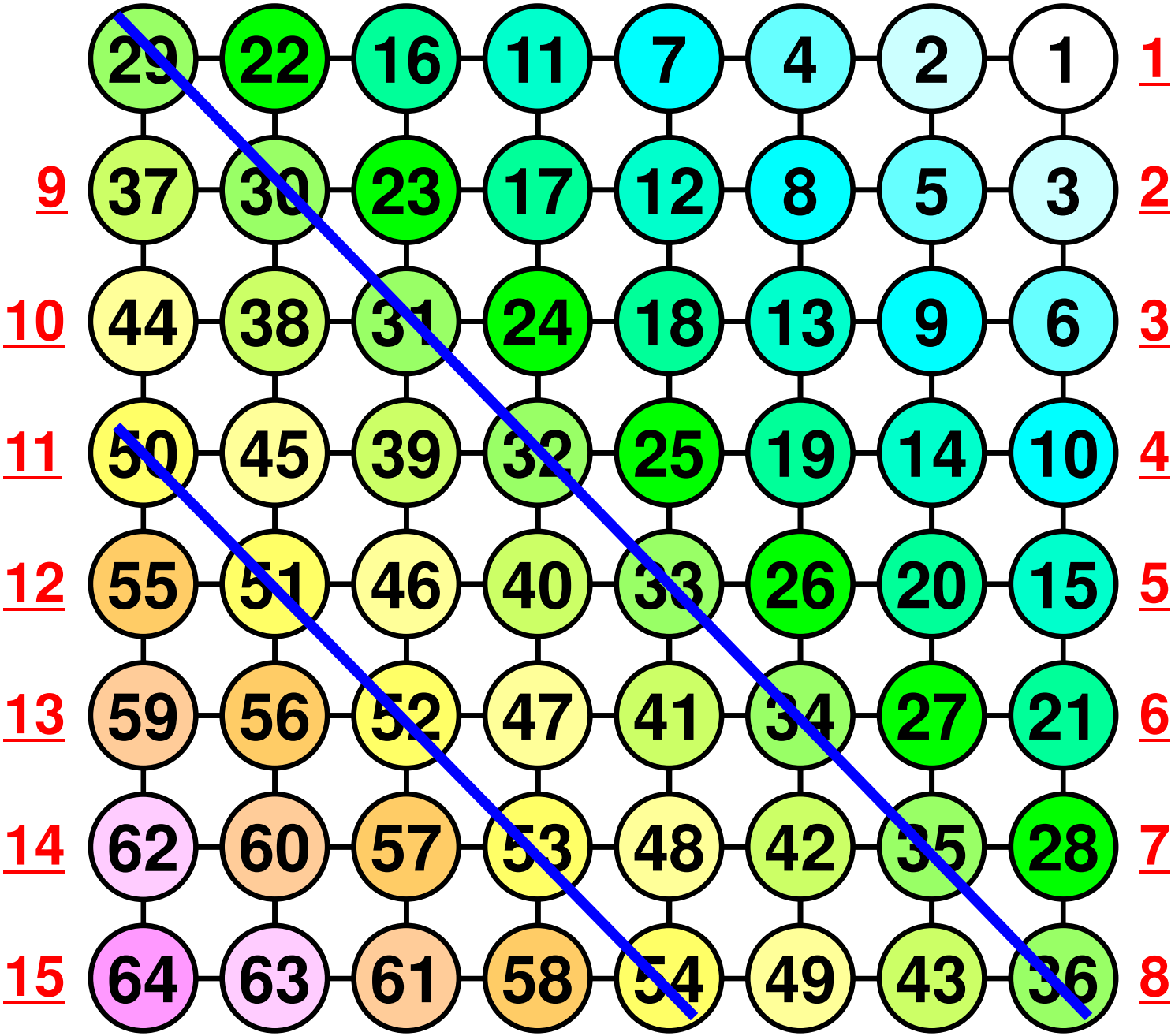
```
for(ic=0; ic<NCOLORtot; ic++) {
#pragma omp parallel for private (i,WVAL)
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}
```



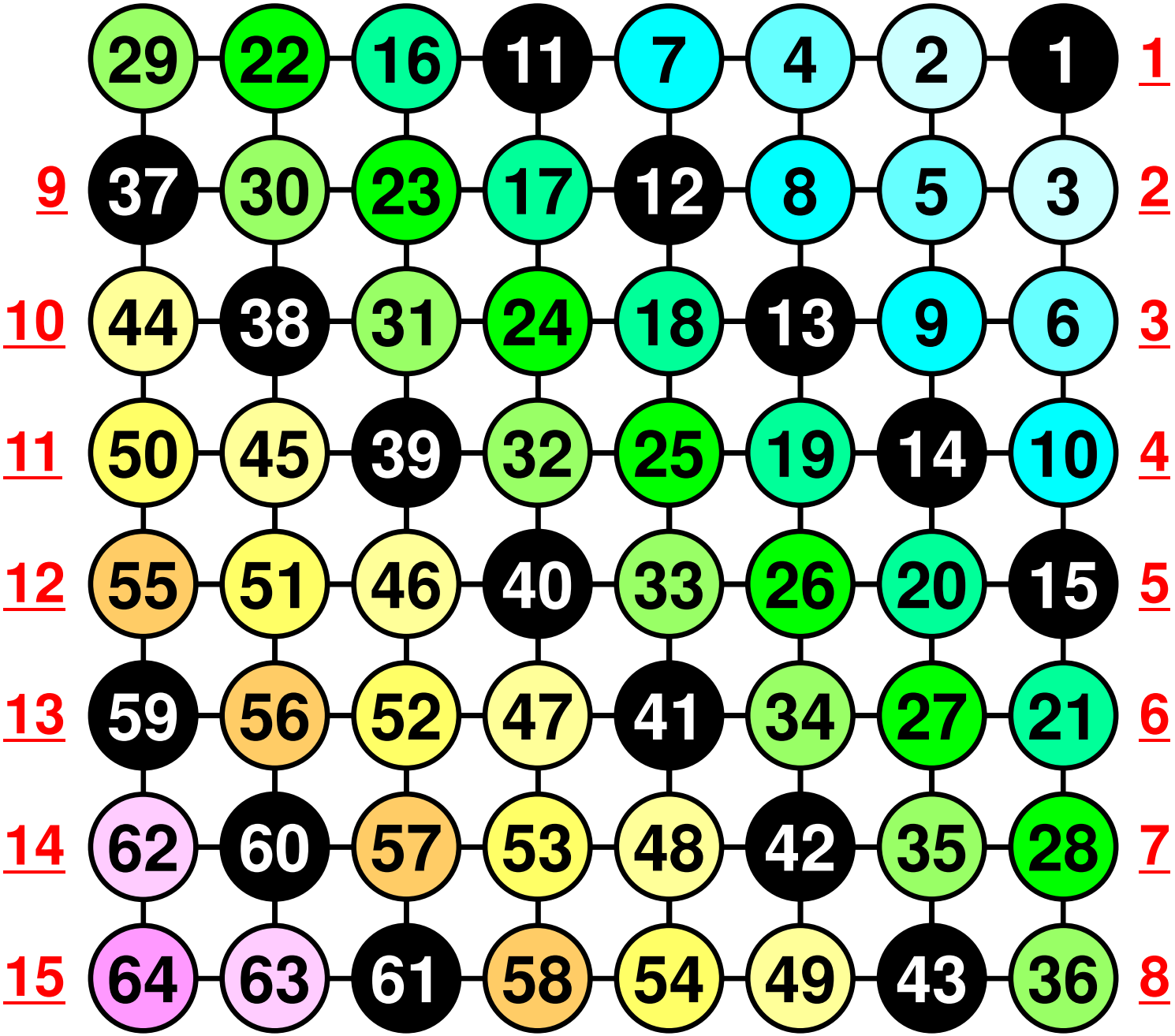
Remedy: CM-RCM

- RCM + Cyclic Mulricoloring [Doi, Osoda, Washio]
- Procedures
  - Apply RCM
  - Define “Nc” (Color number of Cyclic Multicoloring (CM))
    - 1^{st}-Color in CM-RCM: 1^{st}, $(Nc+1)^{th}$, $(2Nc+1)^{th}$ ... levels in RCM
    - 2^{nd}-Color in CM-RCM: 2^{nd}, $(Nc+2)^{th}$, $(2Nc+2)^{th}$ ... levels in RCM
    - k^{th}-Color in CM-RCM: k^{th}, $(Nc+k)^{th}$, $(2Nc+k)^{th}$ ... levels in RCM
    - Each level of RCM is colored in cyclic manner (cycle=Nc).
  - If “k” reaches “Nc”, and all levels of RCM are colored, it’s completed.
    - Renumber meshes in ascending orders according to “Color” ID.
  - If dependency between levels in same color, start from the beginning of the cyclic multicoloring with Nc=Nc+1.

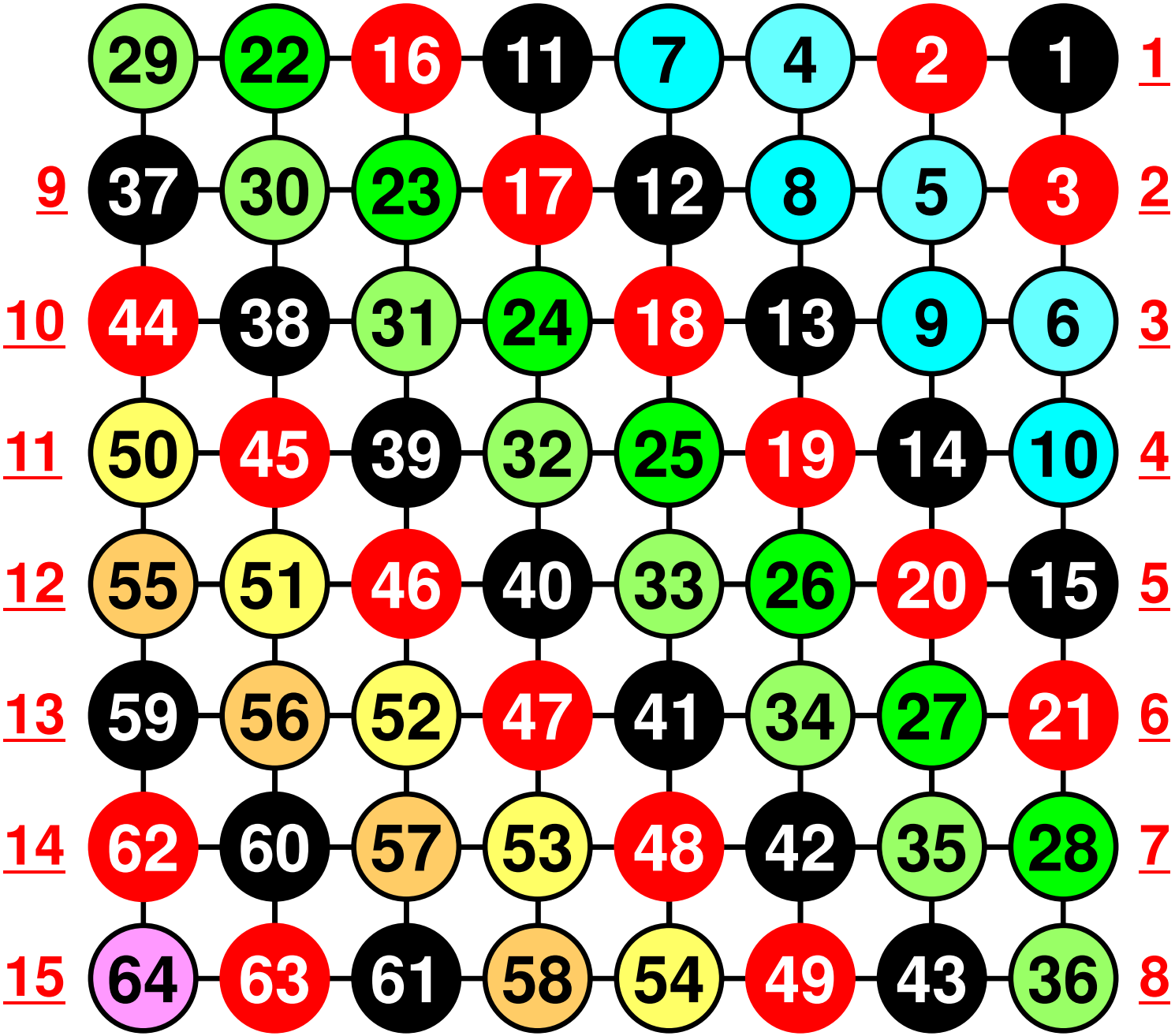
RCM: 1st/Last Level: Only 1 Mesh/Level



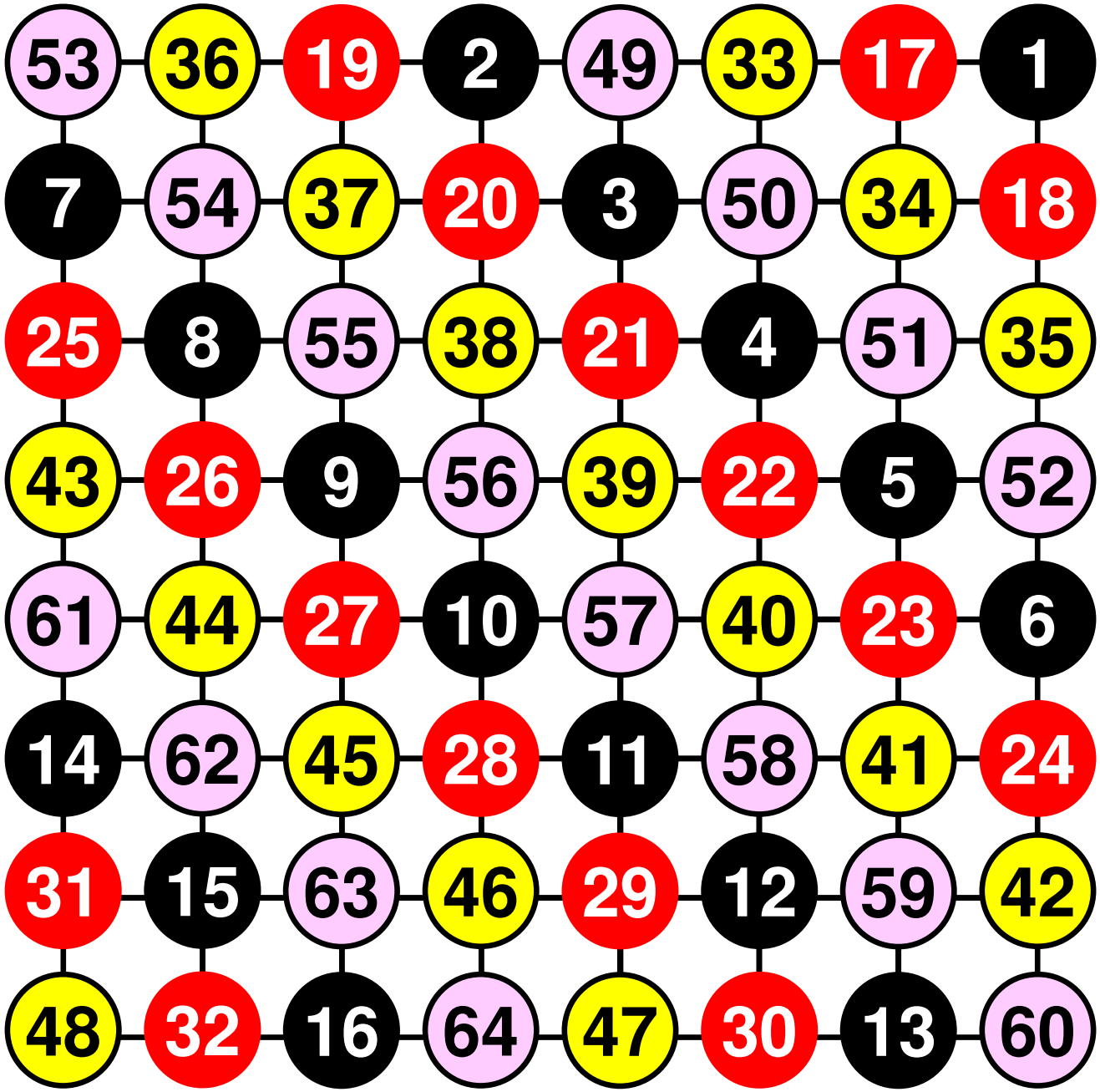
Nc=4, k=1, Level: 1,5,9,13



Nc=4, k=2, Level: 2,6,10,14



CM-RCM(Nc=4): Renumbering



k=1: 16
k=2: 16
k=3: 16
k=4: 16

CM-RCM

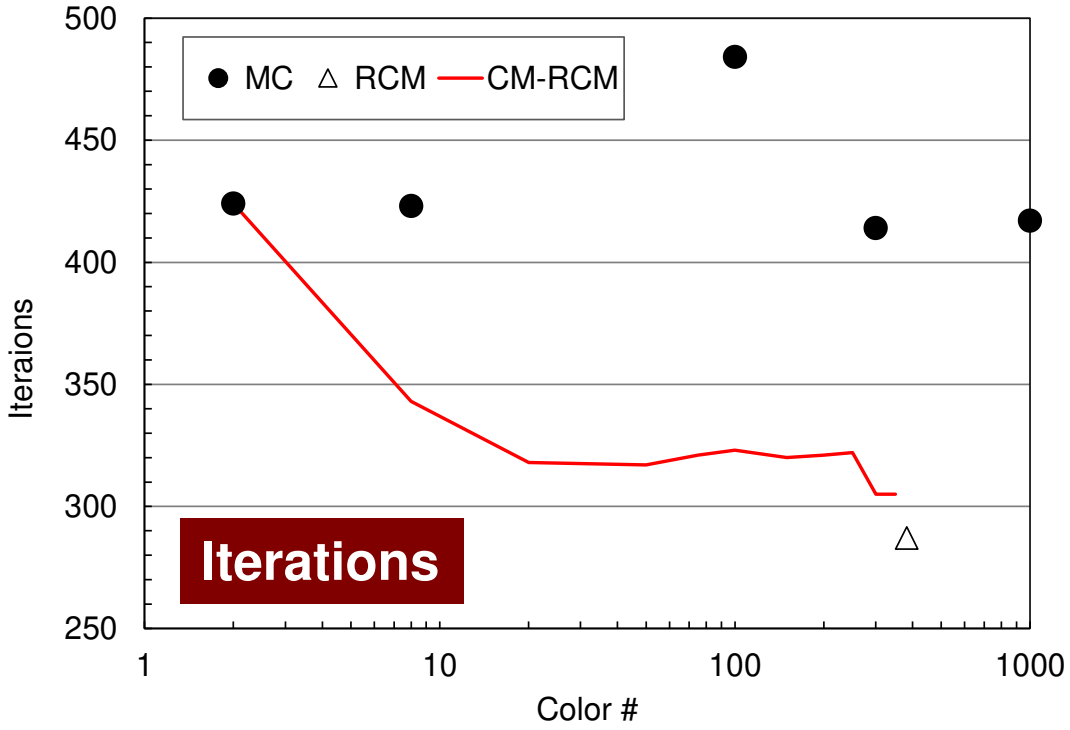
- How to run
  - "**NCOLORtot=-Nc**" in INPUT.DAT
  - Already implemented in L2

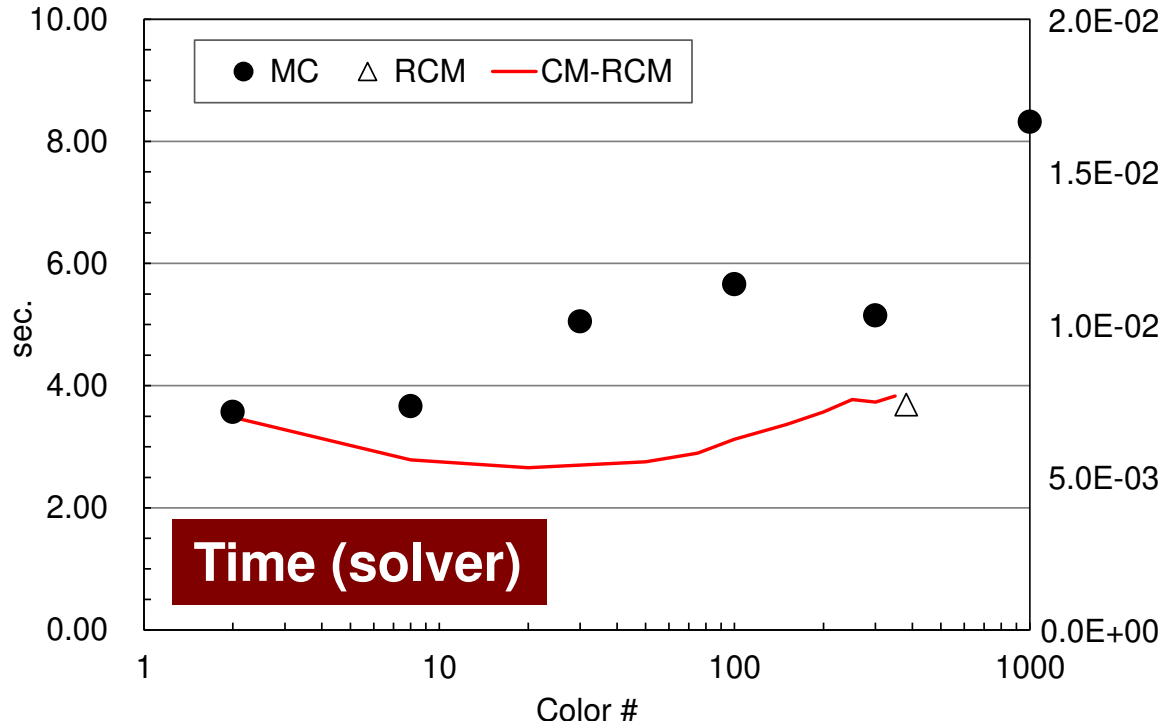
- cmrcm.f, cmrcm.c

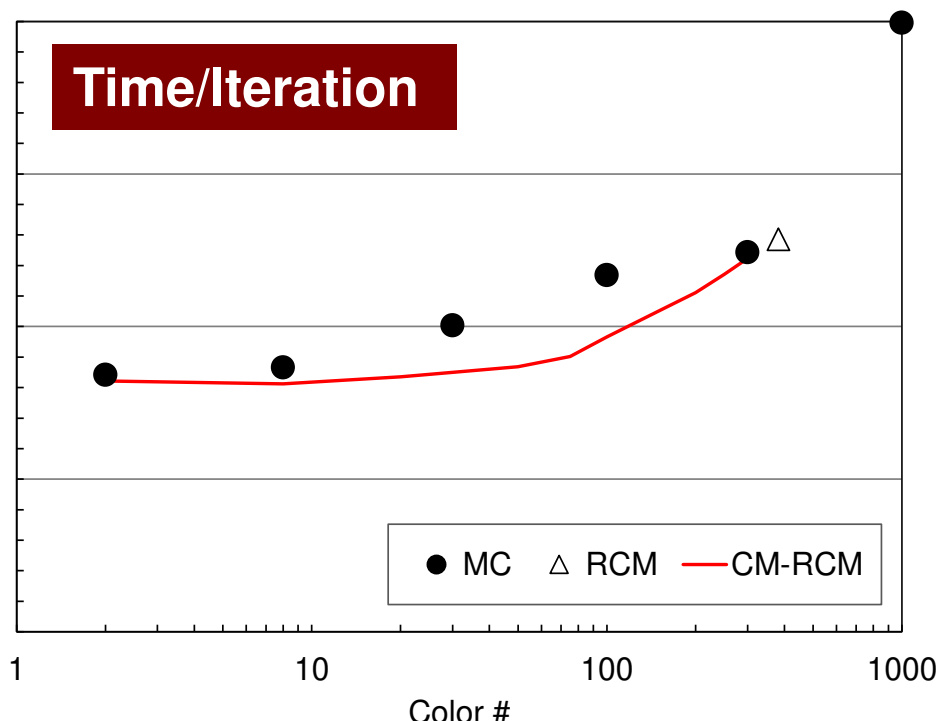
Odyssey

1-CMG/12-cores, 128^3

(●:MC, △:RCM, -:CM-RCM)

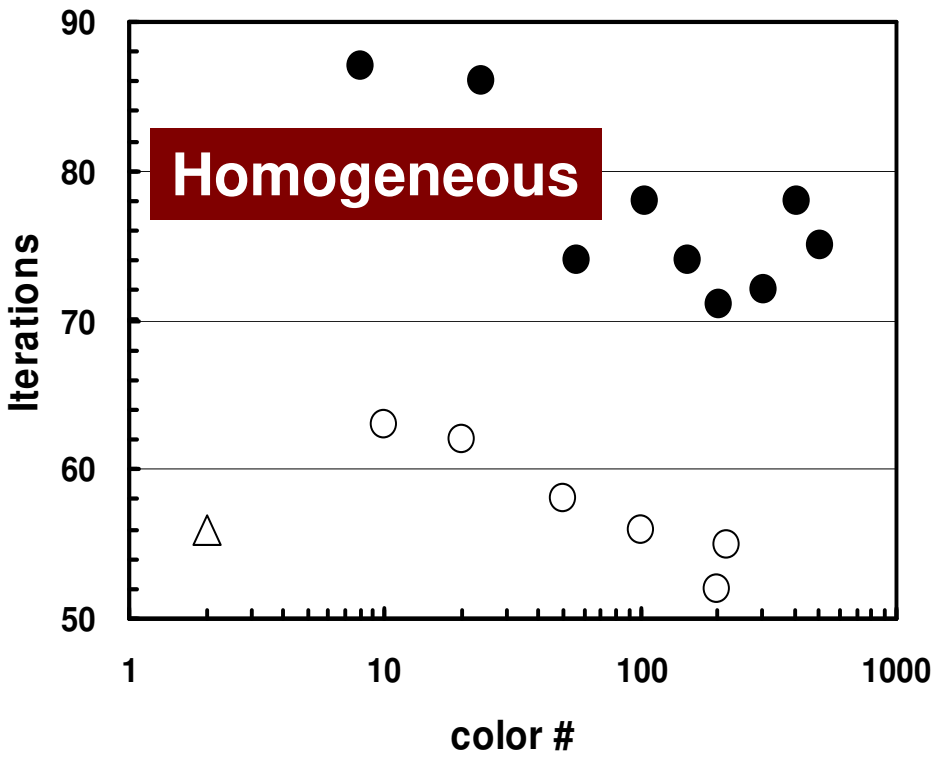


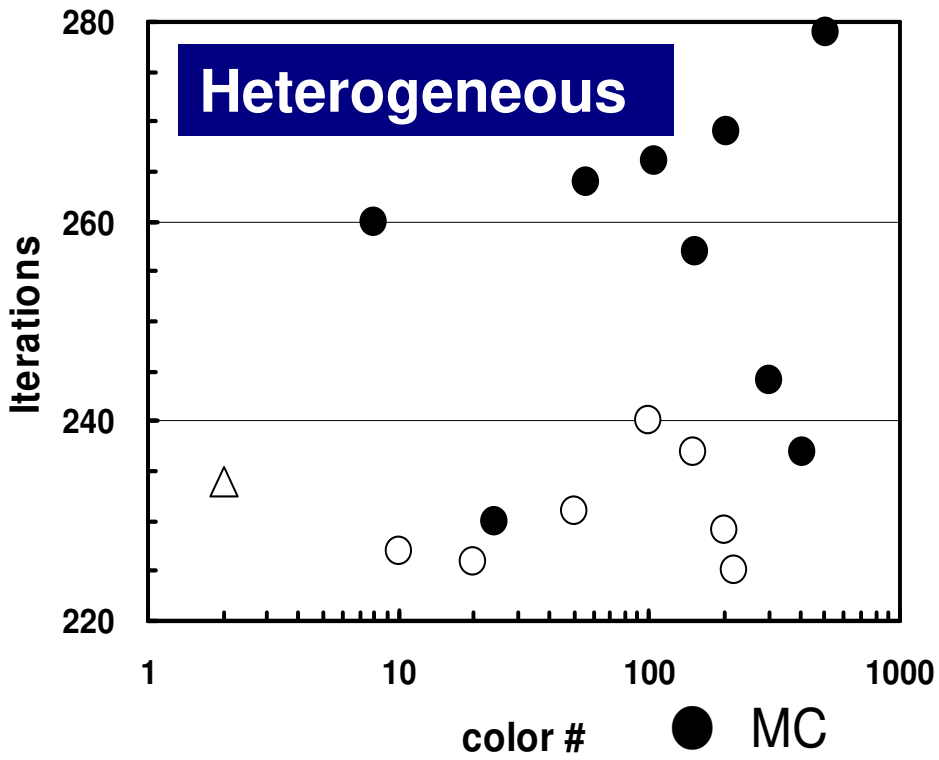




Comparison of Reordering Methods 3D Linear Elastic Problems

- MC: Slow convergence, unstable for heterogeneous cases (ill-conditioned problems).
- Cyclic-Mulricoloring + RCM (CM-RCM) is effective





MC
CM-RCM
No reordering

3D Linear-Elastic Problems with 32,768 DOF

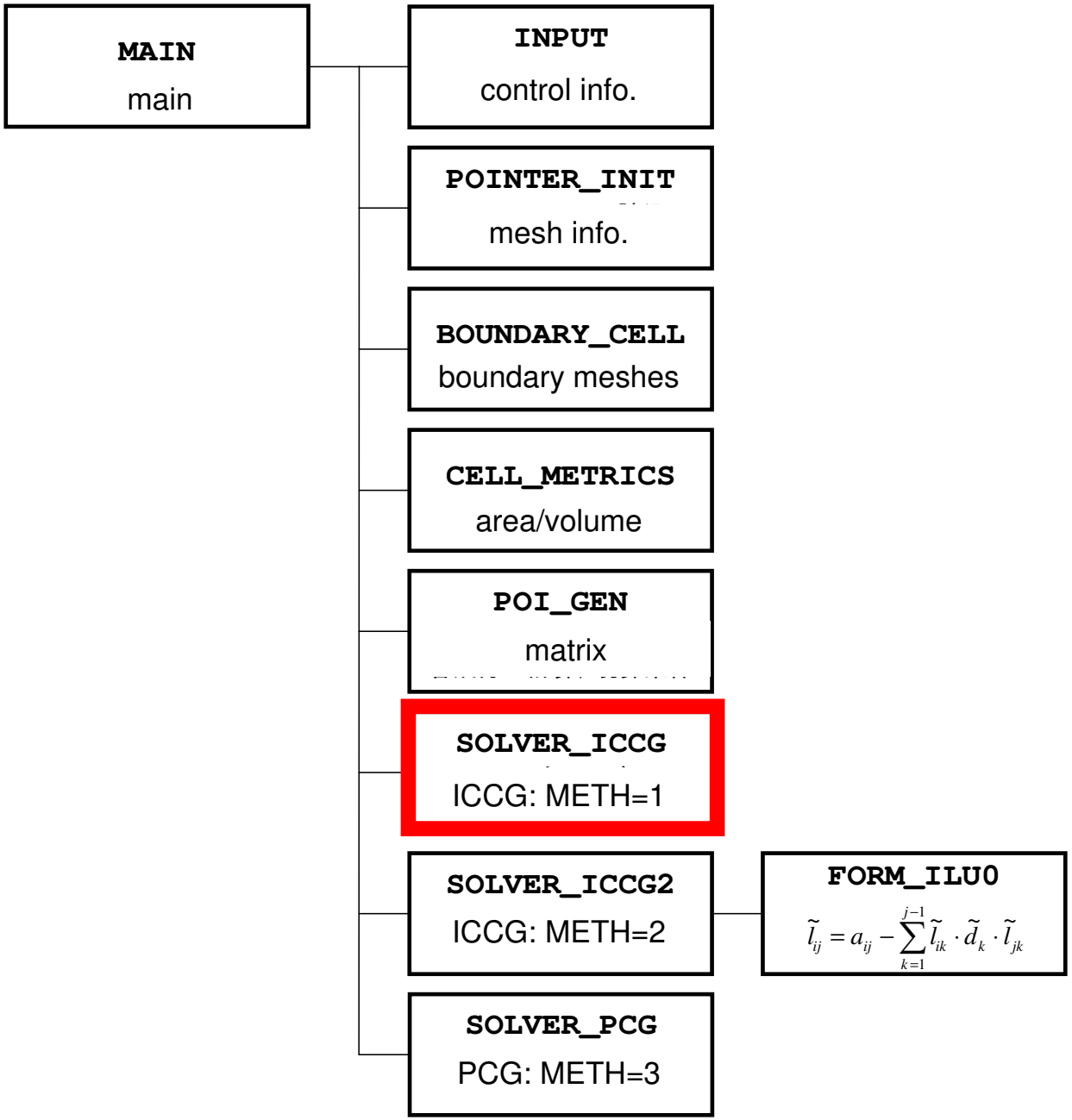
- Remedy for Data Dependency
- Ordering/Reordering
  - Red-Black, Multicoloring (MC)
  - Cuthill-McKee (CM), Reverse-CM (RCM)
  - Reordering and Convergence
- Implementation
- **ICCG with Reordering**

Implementation of Reordering to ICCG

- Apply “L2-color” to “L1-sol”

- Calling “mc”, “cm”, “rcm” and “cmrcm” after computation of “INU, INL, IAL, IAU” in “poi_gen”.
- Computing “D,AL,AU” by new numbering.
- B.C., and RHS are applied by new numbering.
- Calling “ICCG”
- Renumbering components of “PHI (results)” into initial numbering.
- OUPUT_UCD (UCD file)

L1-sol


MAIN
main
INPUT
control info.
POINTER_INIT
mesh info.
BOUNDARY_CELL
boundary meshes
CELL_METRICS
area/volume
POI_GEN
matrix
SOLVER_ICCG
ICCG: METH=1
SOLVER_ICCG2
ICCG: METH=2
FORM_ILU0
$\tilde{l}_{ij} = a_{ij} - \sum_{k=1}^{j-1} \tilde{l}_{ik} \cdot \tilde{d}_k \cdot \tilde{l}_{jk}$
SOLVER_PCG
PCG: METH=3

Minv{r}={z} (1/2)

Forward Substitution $(L)\{z\}=\{r\}$ $(M)\{z\}=(LDL^T)\{z\}=\{r\}$

```
for(i=0; i<N; i++) {
    WVAL = W[Z][i];
    for(j=indexL[i]; j<indexL[i+1]; j++) {
        WVAL -= AL[j] * W[Z][itemL[j]-1];
    }
    W[Z][i] = WVAL * W[DD][i];
}
```

Backward Substitution $(DL^T)\{z\}=\{z\}$

```
for(i=N-1; i>=0; i--) {
    SW = 0.0;
    for(j=indexU[i]; j<indexU[i+1]; j++) {
    SW += AU[j] * W[Z][itemU[j]-1];
    }
    W[Z][i]= W[Z][i]  - W[DD][i] * SW;
}
```

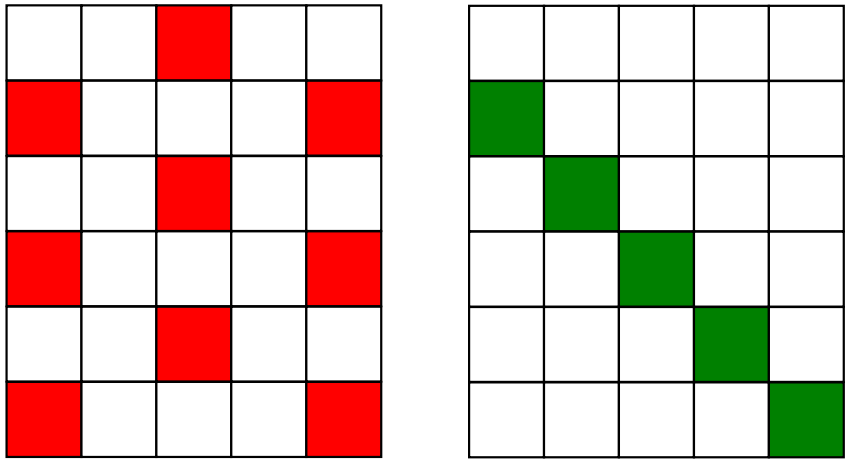
Data Dependency
Z appears in both of LHS and RHS.

Reordering may eliminate this data dependency.

Minv{r}={z} (2/2)

Forward Substitution $(L)\{z\}=\{r\}$ $(M)\{z\}=(LDL^T)\{z\}=\{r\}$

```
for(icol=0; icol<NCOLORtot; icol++){
    for(i=COLORindex[icol]; i<COLORindex[icol+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}
```



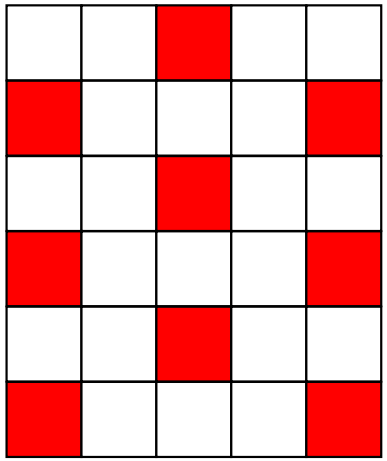
"Z" components in RHS do not belong to "icol-th" color.

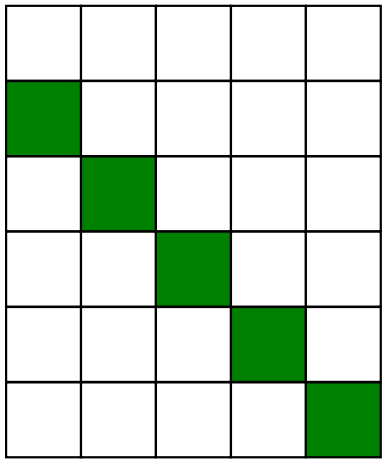
Meshes in same color are independent. (No Data Dependency)

Minv{r}={z} (2/2)

Forward Substitution $(L)\{z\}=\{r\}$ $(M)\{z\}=(LDL^T)\{z\}=\{r\}$

```
for(icol=0; icol<NCOLORtot; icol++){
    for(i=COLORindex[icol]; i<COLORindex[icol+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}
```



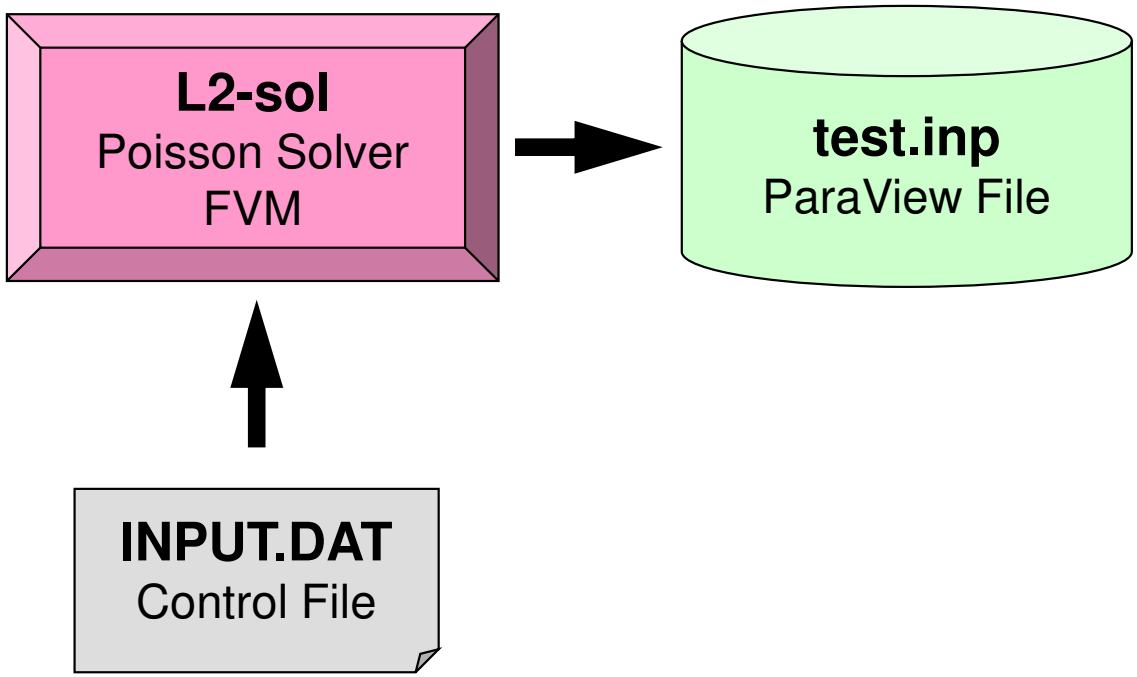


Parallel processing can be applied to these loops.

Files

```
$> cd multicore-c/L2/solver/src


$> make
$> ls ../run/L2-sol
   L2-sol
```

Running the Program <$P-L2>/solver/run

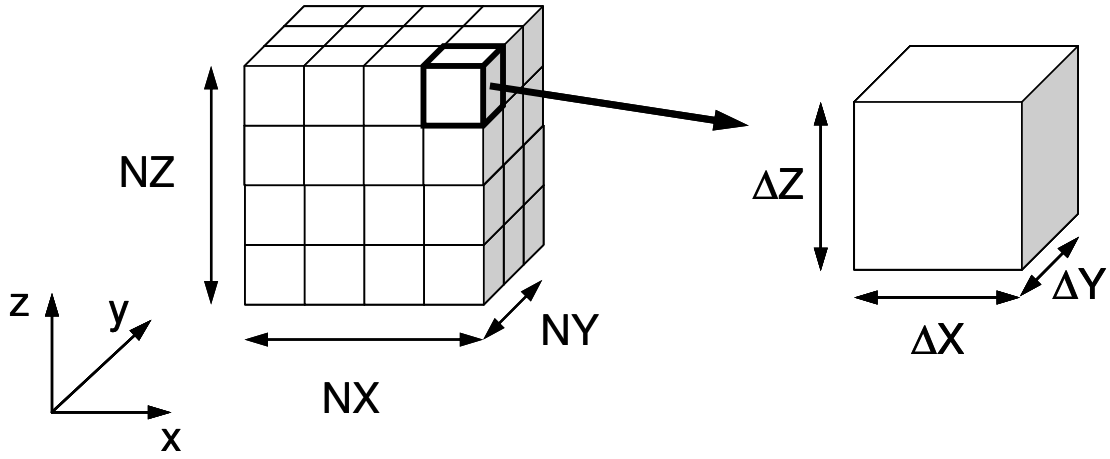


Running the Program

Control Data: <$P-L2>/solver/run/INPUT.DAT

```
32 32 32                                NX/NY/NZ
1.00e-00 1.00e-00 1.00e-00              DX/DY/DZ
1.0e-08                                 EPSICCG
```

- NX, NY, NZ
  - Number of meshes in X/Y/Z dir.
- DX, DY, DZ
  - Size of meshes
- EPSICCG
  - Convergence Criteria for ICCG



Running the Program
<$P-L2>/solver/run/

```
$ cd <$P-L2>/solver/run
$ ./L2-sol

You have    8000 elements.
How many colors do you need ?
  #COLOR must be more than 2 and
  #COLOR must not be more than    8000
   CM if #COLOR .eq. 0
  RCM if #COLOR .eq.-1
CMRCM if #COLOR .le.-2

=> XXX

$ ls test.inp
```

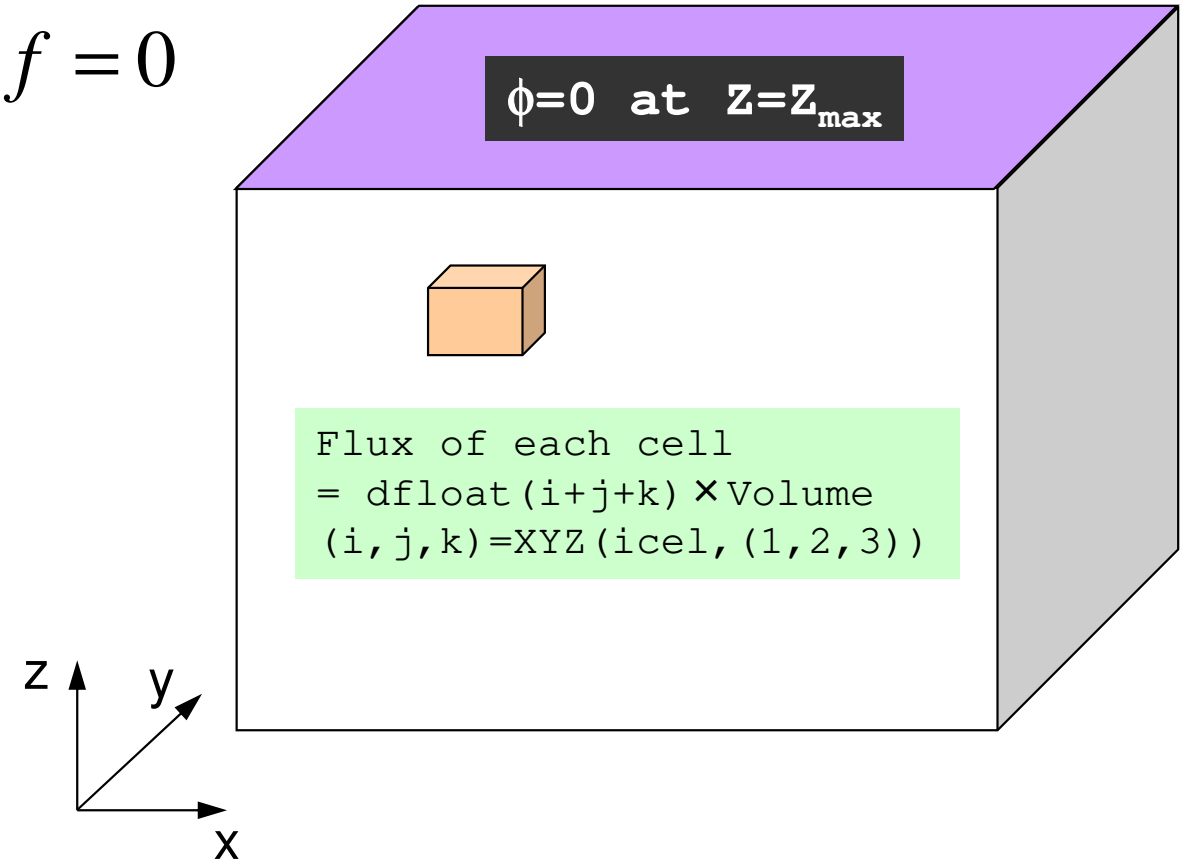
Target Problem: Variables are defined at cell-center’

Poisson Equation

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

Boundary Conditions (B.C.) etc.

- Volume Flux
- $\phi = 0 @ Z = Z_{max}$

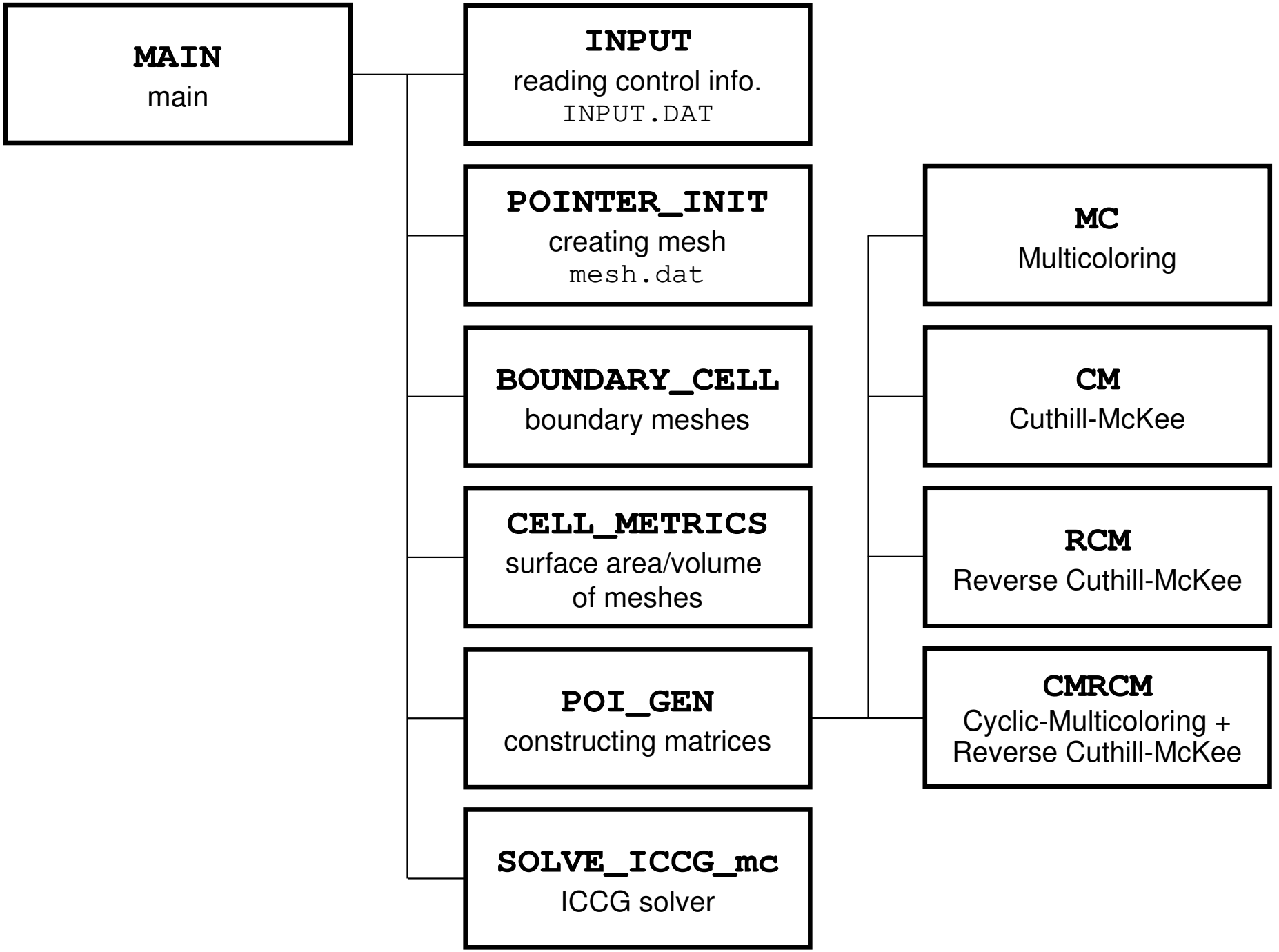


Main Program

```
#include <stdio.h> ...

int
main()
{
        double *WK;
        int NPL, NPU; ISET, ITR, IER; icel, ic0, i;
        double xN, xL, xU; Stime, Etime;

        if(INPUT()) goto error;
        if(POINTER_INIT()) goto error;
        if(BOUNDARY_CELL()) goto error;
        if(CELL_METRICS()) goto error;
        if(POI_GEN()) goto error;

        ISET = 0;
        WK = (double *)malloc(sizeof(double)*ICELTOT);
        if(WK == NULL) {
           fprintf(stderr, "Error: %s¥n", strerror(errno));
           goto error;}
        if(solve_ICCG_mc(ICELTOT, NL, NU, indexL, itemL, indexU, itemU,
                         D, BFORCE, PHI, AL, AU, NCOLORtot, COLORindex,
                         EPSICCG, &ITR, &IER)) goto error;

        for(ic0=0; ic0<ICELTOT; ic0++) {
            icel = NEWtoOLD[ic0];
            WK[icel-1] = PHI[ic0];
        }
        for(icel=0; icel<ICELTOT; icel++) {
            PHI[icel] = WK[icel];
        }
        if(OUTUCD()) goto error;
        return 0;
error:
        return -1;
}
```

Renumbering of “PHI” to original numbering

Structure of L2-sol



Variables/Arrays for Matrix (1/2)

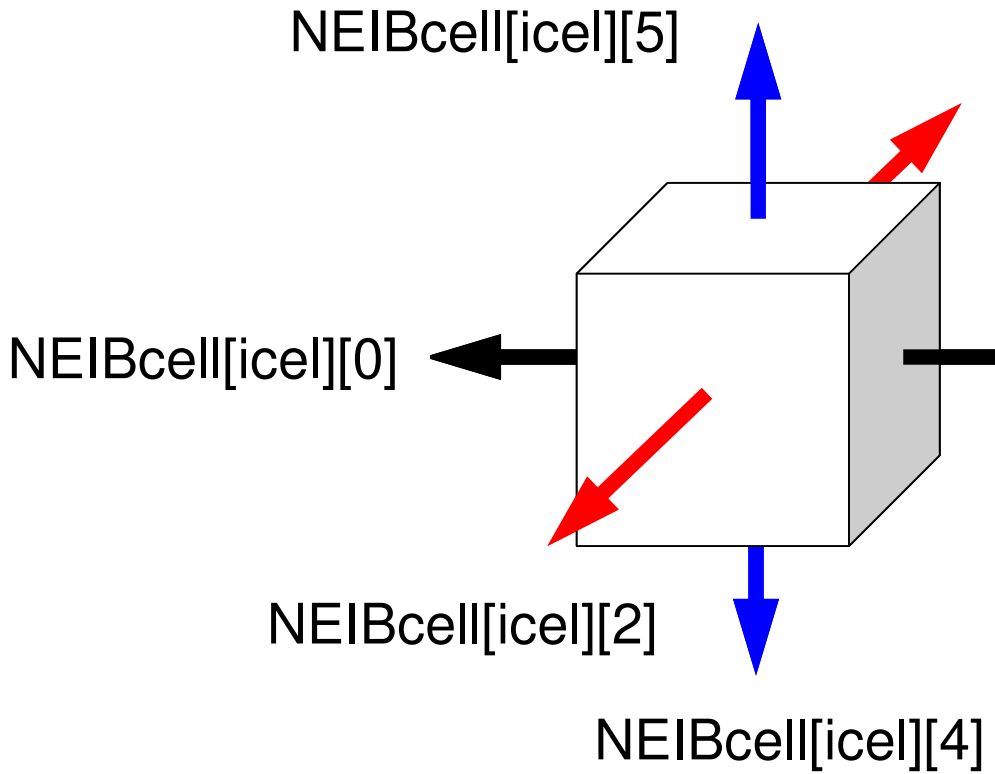
Name	Type	Content
D[N]	**R**	Diagonal components of the matrix (N= ICELTOT)
BFORCE[N]	**R**	RHS vector
PHI[N]	**R**	Unknown vector
indexL[N+1] **indexU[N+1]**	**I**	# of L/U non-zero off-diag. comp. (CRS)
NPL, NPU	**I**	Total # of L/U non-zero off-diag. comp. (CRS)
itemL[NPL] **itemU[NPU]**	**I**	Column ID of L/U non-zero off-diag. comp. (CRS)
AL[NPL] **AU[NPU]**	**R**	L/U non-zero off-diag. comp. (CRS)

Name	Type	Content
NL,NU	**I**	MAX. # of L/U non-zero off-diag. comp. for each mesh (=6)
INL[N] **INU[N]**	**I**	# of L/U non-zero off-diag. comp.
IAL[N][NL] **IAU[N][NU]**	**I**	Column ID of L/U non-zero off-diag. comp.

Variables/Arrays for Matrix (2/2)

Name	Type	Content
NCOLORtot	**I**	**Input:** reordering method + initial number of colors/levels ≧2: MC, =0: CM, =-1: RCM, -2≧: CMRCM **Output:** Final number of colors/levels
COLORindex [NCOLORtot+1]	**I**	Number of meshes at each color/level 1D compressed array Meshes in **icolth** color/level are stored in this array from **COLORindex[icol]** to **COLORindex[icol+1]-1**
NEWtoOLD[N]	**I**	Reference array from New to Old numbering
OLDtoNEW[N]	**I**	Reference array from Old to New numbering

NEIBcell: ID of Neighboring Mesh/Cell =0: for Boundary Surface



i= XYZ[icel][0]
j= XYZ[icel][1], k= XYZ[icel][2]
icel= k*NX*NY + j*NX + i

NEIBcell[icel][0]= icel – 1 + 1
NEIBcell[icel][1]= icel + 1 + 1
NEIBcell[icel][2]= icel – NX + 1
NEIBcell[icel][3]= icel + NX + 1
NEIBcell[icel][4]= icel – NX*NY + 1
NEIBcell[icel][5]= icel + NX*NY + 1

Main Program

```
#include <stdio.h> ...

int
main()
{
        double *WK;
        int NPL, NPU; ISET, ITR, IER; icel, ic0, i;
        double xN, xL, xU; Stime, Etime;

        if(INPUT()) goto error;
        if(POINTER_INIT()) goto error;
        if(BOUNDARY_CELL()) goto error;
        if(CELL_METRICS()) goto error;
        if(POI_GEN()) goto error;

        ISET = 0;
        WK = (double *)malloc(sizeof(double)*ICELTOT);
        if(WK == NULL) {
           fprintf(stderr, "Error: %s¥n", strerror(errno));
           goto error;}
        if(solve_ICCG_mc(ICELTOT, NL, NU, indexL, itemL, indexU, itemU,
                         D, BFORCE, PHI, AL, AU, NCOLORtot, COLORindex,
                         EPSICCG, &ITR, &IER)) goto error;

        for(ic0=0; ic0<ICELTOT; ic0++) {
            icel = NEWtoOLD[ic0];
            WK[icel-1] = PHI[ic0];
        }
        for(icel=0; icel<ICELTOT; icel++) {
            PHI[icel] = WK[icel];
        }
        if(OUTUCD()) goto error;
        return 0;
error:
        return -1;
}
```

poi_gen (1/8)

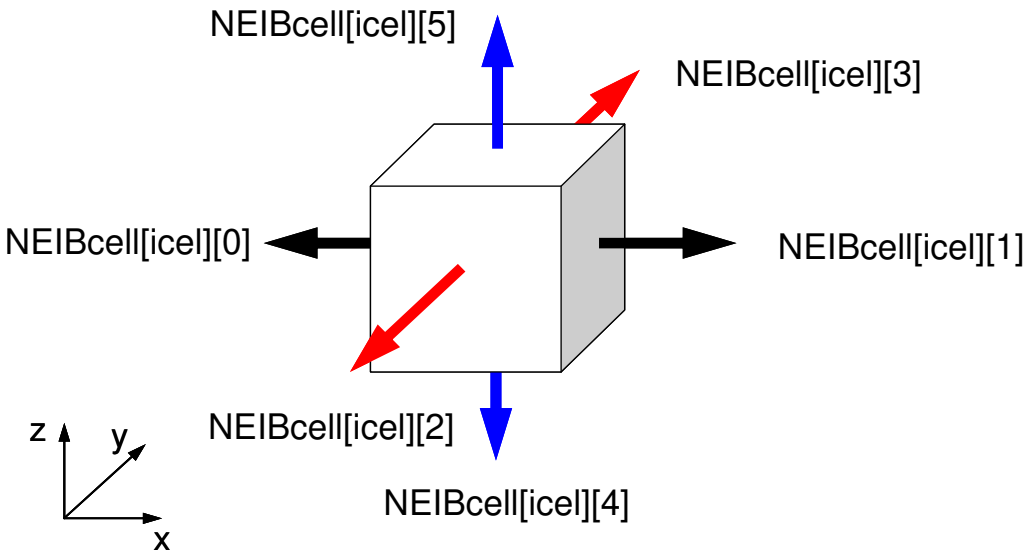
```
#include "allocate.h"
extern int
POI_GEN(void)
{ int nn;
  int icO, icN1, icN2, icN3, icN4, icN5, icN6;
  int i, j, k, ib, ic, ip, icel, icou, icol, icouG;
  int ii, jj, kk, nn1, num, nr, j0, j1;
  double coef, VOLO, S1t, E1t;
  int isL, ieL, isU, ieU;
  NL=6; NU= 6;
  IAL = (int **)allocate_matrix(sizeof(int),ICELTOT,NL);
  IAU = (int **)allocate_matrix(sizeof(int),ICELTOT,NU);
  BFORCE = (double *)allocate_vector(sizeof(double),ICELTOT);
  D      = (double *)allocate_vector(sizeof(double),ICELTOT);
  PHI    = (double *)allocate_vector(sizeof(double),ICELTOT);
  INL = (int *)allocate_vector(sizeof(int),ICELTOT);
  INU = (int *)allocate_vector(sizeof(int),ICELTOT);

  for (i = 0; i <ICELTOT ; i++) {
       BFORCE[i]=0.0;
       D[i]  =0.0; PHI[i]=0.0;
       INL[i] = 0; INU[i] = 0;
       for(j=0;j<6;j++){
           IAL[i][j]=0; IAU[i][j]=0;
       }
  }
  for (i = 0; i <=ICELTOT ; i++) {
       indexL[i] = 0; indexU[i] = 0;
  }
```

allocate.c

```
/*************************
  allocate matrix
*************************/
void** allocate_matrix(int size,int m,int n)
{
  void **aa;
  int i;
  if ( ( aa=(void ** )malloc( m * sizeof(void*) ) ) == NULL ) {
     fprintf(stdout,"Error:Memory does not enough! aa in matrix ¥n");
     exit(1);
  }
  if ( ( aa[0]=(void * )malloc( m * n * size ) ) == NULL ) {
     fprintf(stdout,"Error:Memory does not enough! in matrix ¥n");
     exit(1);
  }
        for(i=1;i<m;i++) aa[i]=(char*)aa[i-1]+size*n;
  return aa;
}
```

```
for(icel=0; icel<ICELTOT; icel++) {

      icN1 = NEIBcell[icel][0];
      icN2 = NEIBcell[icel][1];
      icN3 = NEIBcell[icel][2];
      icN4 = NEIBcell[icel][3];
      icN5 = NEIBcell[icel][4];
      icN6 = NEIBcell[icel][5];

      if(icN5 != 0) {
        icou = INL[icel] + 1;
        IAL[icel][icou-1] = icN5;
        INL[icel]         = icou;
      }

      if(icN3 != 0) {
        icou = INL[icel] + 1;
        IAL[icel][icou-1] = icN3;
        INL[icel]         = icou;
      }

      if(icN1 != 0) {
        icou = INL[icel] + 1;
        IAL[icel][icou-1] = icN3;
        INL[icel]         = icou;
      }
```

poi_gen (2/8)



Lower Triangular Part

NEIBcell[icel][4]= icel – NX*NY + 1
NEIBcell[icel][2]= icel – NX + 1
NEIBcell[icel][0]= icel – 1 + 1

“icel” starts at 0

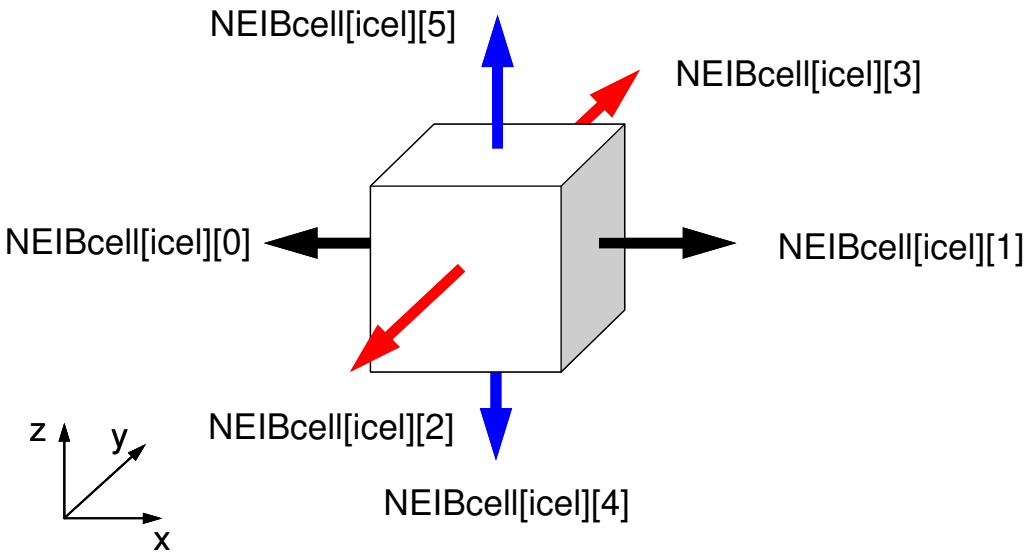
12	13	14	15
8	9	10	11
4	5	6	7
0	1	2	3

“IAL” starts at 1

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

poi_gen (3/8)

```
 for(icel=0; icel<ICELTOT; icel++) {

      icN1 = NEIBcell[icel][0];
      icN2 = NEIBcell[icel][1];
      icN3 = NEIBcell[icel][2];
      icN4 = NEIBcell[icel][3];
      icN5 = NEIBcell[icel][4];
      icN6 = NEIBcell[icel][5];
....
      if(icN2 != 0) {
        icou = INU[icel] + 1;
        IAU[icel][icou-1] = icN2;
        INU[icel]         = icou;
      }

      if(icN4 != 0) {
        icou = INU[icel] + 1;
        IAU[icel][icou-1] = icN4;
        INU[icel]         = icou;
      }

      if(icN6 != 0) {
        icou = INU[icel] + 1;
        IAU[icel][icou-1] = icN6;
        INU[icel]         = icou;
      }
}
```



Upper Triangular Part

NEIBcell[icel][1]= icel + 1 + 1
NEIBcell[icel][3]= icel + NX + 1
NEIBcell[icel][5]= icel + NX*NY + 1

“icel” starts at 0

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

“IAU” starts at 1

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

poi_gen (4/8)

```
N111:
        fprintf(stderr, "¥n¥nYou have%8d elements¥n", ICELTOT);
        fprintf(stderr, "How many colors do you need ?¥n");
        fprintf(stderr, "  #COLOR must be more than 2 and¥n");
        fprintf(stderr, "  #COLOR must not be more than%8d¥n", ICELTOT);
        fprintf(stderr, "  if #COLOR= 0 then CM  ordering¥n");
        fprintf(stderr, "  if #COLOR=-1 then RCM ordering¥n");
        fprintf(stderr, "  if #COLOR<-1 then CMRCM ordering¥n");
        fprintf(stderr, "=>¥n");
        fscanf(stdin, "%d", &NCOLORtot);
        if(NCOLORtot == 1 && NCOLORtot > ICELTOT) goto N111;

        OLDtoNEW = (int *)calloc(ICELTOT, sizeof(int));
        if(OLDtoNEW == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        NEWtoOLD = (int *)calloc(ICELTOT, sizeof(int));
        if(NEWtoOLD == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }
        COLORindex = (int *)calloc(ICELTOT+1, sizeof(int));
        if(COLORindex == NULL) {
                fprintf(stderr, "Error: %s¥n", strerror(errno));
                return -1;
        }

        if(NCOLORtot > 0) {
                MC(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot == 0) {
                CM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot ==-1) {
                RCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        } else if(NCOLORtot <-1) {
                CMRCM(ICELTOT, NL, NU, INL, IAL, INU, IAU,
                                &NCOLORtot, COLORindex, NEWtoOLD, OLDtoNEW);
        }

        fprintf(stderr, "¥n# TOTAL COLOR number%8d¥n", NCOLORtot);
        return 0;
}
```

poi_gen (5/8)

New numbering is applied after this point

```
indexL =
   (int *)allocate_vector(sizeof(int),ICELTOT+1);
indexU =
   (int *)allocate_vector(sizeof(int),ICELTOT+1);

for(i=0; i<ICELTOT; i++){
    indexL[i+1]=indexL[i]+INL[i];
    indexU[i+1]=indexU[i]+INU[i];
}
NPL = indexL[ICELTOT];
NPU = indexU[ICELTOT];

itemL = (int *)allocate_vector(sizeof(int),NPL);
itemU = (int *)allocate_vector(sizeof(int),NPU);
AL = (double *)allocate_vector(sizeof(double),NPL);
AU = (double *)allocate_vector(sizeof(double),NPU);

memset(itemL, 0, sizeof(int)*NPL);
memset(itemU, 0, sizeof(int)*NPU);
memset(AL, 0.0, sizeof(double)*NPL);
memset(AU, 0.0, sizeof(double)*NPU);

for(i=0; i<ICELTOT; i++){
    for(k=0;k<INL[i];k++){
        kk= k + indexL[i];
        itemL[kk]= IAL[i][k];
    }
    for(k=0;k<INU[i];k++){
        kk= k + indexU[i];
        itemU[kk]= IAU[i][k];
    }
}

free(INL); free(INU);
free(IAL); free(IAU);
```

Name	Type	Content
`D[N]`	R	Diagonal components of the matrix (N= ICELTOT)
`BFORCE[N]`	R	RHS vector
`PHI[N]`	R	Unknown vector
`indexL[N+1]` `indexU[N+1]`	I	# of L/U non-zero off-diag. comp. (CRS)
`NPL, NPU`	I	Total # of L/U non-zero off-diag. comp. (CRS)
`itemL[NPL]` `itemU[NPU]`	I	Column ID of L/U non-zero off-diag. comp. (CRS)
`AL[NPL]` `AU[NPU]`	R	L/U non-zero off-diag. comp. (CRS)

“itemL” /”itemU” start at 1

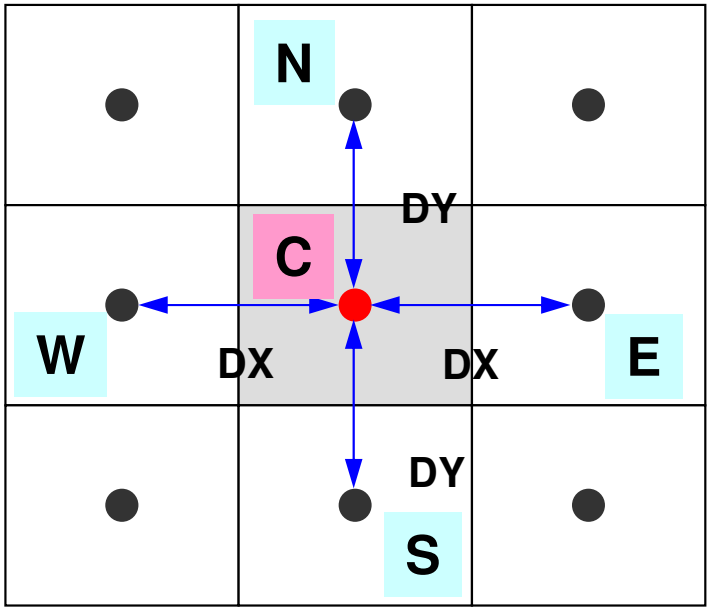
13	14	15	16
9	10	11	12
5	6	7	8
1	2	3	4

```
for (i=0; i<N; i++) {
    q[i]= D[i] * p[i];
    for (j=indexL[i]; j<indexL[i+1]; j++) {
        q[i] += AL[j] * p[itemL[j]-1];
    }
    for (j=indexU[i]; j<indexU[i+1]; j++) {
        q[i] += AU[j] * p[itemU[j]-1];
    }
}
```

```
for(icel=0;  icel<N; icel++) {
    ic0  = NEWtoOLD[icel];
    icN1 = NEIBcell[ic0-1][0];
    icN2 = NEIBcell[ic0-1][1];
    icN3 = NEIBcell[ic0-1][2];
    icN4 = NEIBcell[ic0-1][3];
    icN5 = NEIBcell[ic0-1][4];
    icN6 = NEIBcell[ic0-1][5];
    VOL0 = VOLCEL[ic0];

    isL =indexL[icel  ];    ieL =indexL[icel+1];
    isU =indexU[icel  ];    ieU =indexU[icel+1];

    if(icN5 != 0) {
       icN5 = OLDtoNEW[icN5-1];
       coef = RDZ * ZAREA;
       D[icel] -= coef;

       if(icN5-1 < icel) {
          for(j=isL; j<ieL; j++) {
              if(itemL[j] == icN5) {
                 AL[j] = coef;
                 break;
              }
          }
       } else {
          for(j=isU; j<ieU; j++) {
              if(itemU[j] == icN5) {
                 AU[j] = coef;
                 break;
              }
           }
        }
      }
...
```

poi_gen (6/8)

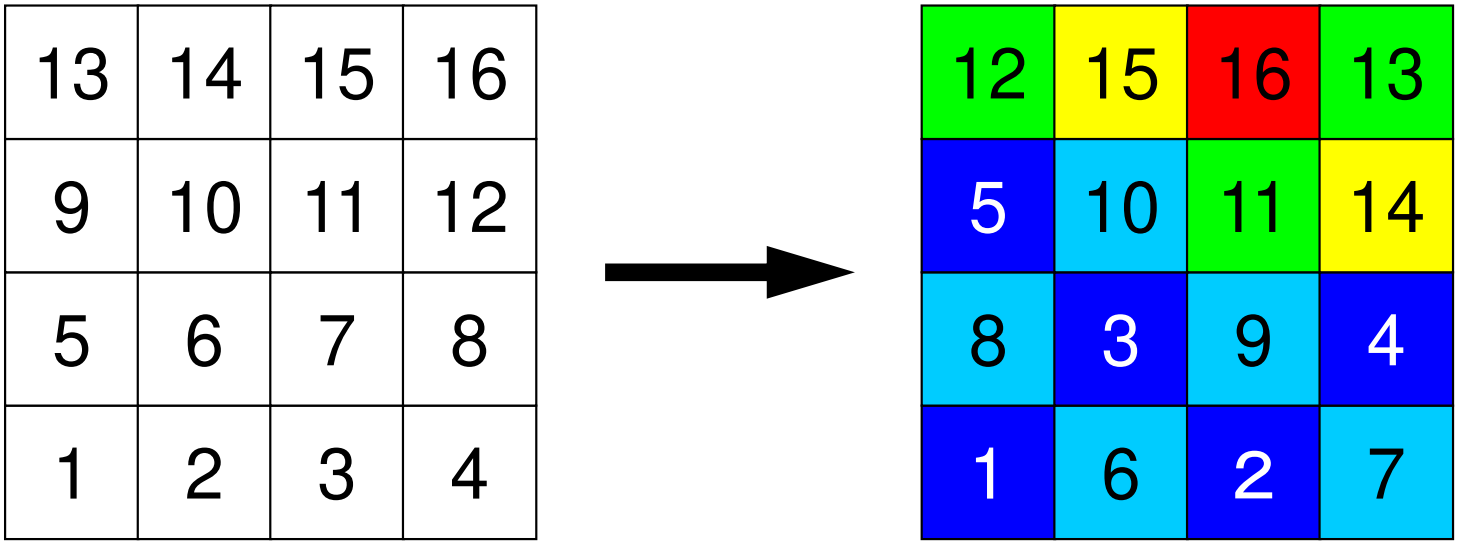
Calculation of Coefficients



$$\frac{\phi_E-\phi_i}{\Delta x}\Delta y+\frac{\phi_W-\phi_i}{\Delta x}\Delta y+$$

$$\frac{\phi_N-\phi_i}{\Delta y}\Delta x+\frac{\phi_S-\phi_i}{\Delta y}\Delta x=f_c\Delta x\Delta y$$

New Numbering



- Coloring by MC/CM/RCM/CM-RCM
- Renumber meshes in ascending orders according to “Level/Color” ID.
  - 1st-Color: 1,2,3,4,5 (Original: 1,3,6,8,9)
  - 2nd-Color: 6,7,8,9,10 (2,4,5,7,10)
  - 3rd-Color: 11,12,13 (11,13,16)
  - 4th-Color: 14,15 (12,14), 5th-Color: 16 (15)

New Numbering (cont.)

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

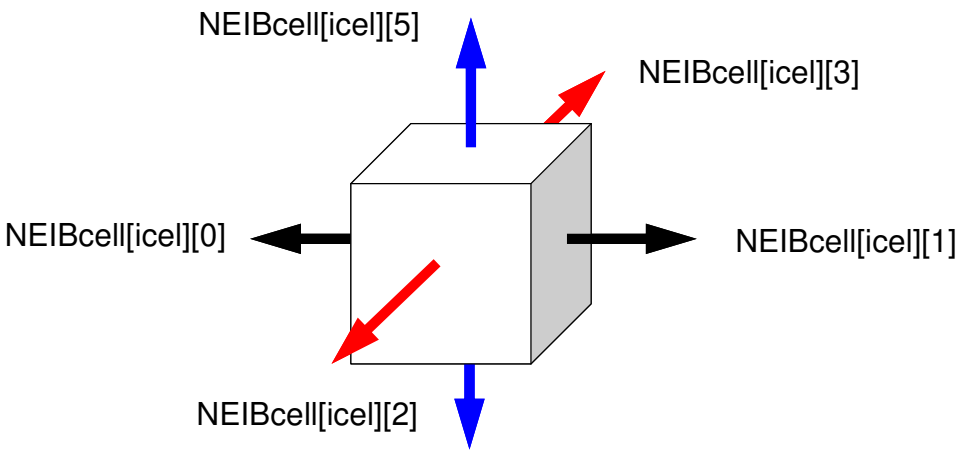
12	15	16	13
5	10	11	14
8	3	9	4
1	6	2	7

```
NCOLORtot= 5
COLORindex[0]=  0, COLORindex[1]=  5, COLORindex[2]= 10
COLORindex[3]= 13, COLORindex[4]= 15, COLORindex[5]= 16
```

- NEWtoOLD, OLDtoNEW
  - **`OLDtoNEW[6-1]=3, NEWtoOLD[3-1]=6`**

poi_gen (6/8)

```
for(icel=0;  icel<N; icel++) {
    ic0  = NEWtoOLD[icel];
    icN1 = NEIBcell[ic0-1][0];
    icN2 = NEIBcell[ic0-1][1];
    icN3 = NEIBcell[ic0-1][2];
    icN4 = NEIBcell[ic0-1][3];
    icN5 = NEIBcell[ic0-1][4];
    icN6 = NEIBcell[ic0-1][5];
    VOL0 = VOLCEL[ic0];

    isL =indexL[icel  ];    ieL =indexL[icel+1];
    isU =indexU[icel  ];    ieU =indexU[icel+1];

    if(icN5 != 0) {
       icN5 = OLDtoNEW[icN5-1];
       coef = RDZ * ZAREA;
       D[icel] -= coef;

       if(icN5-1 < icel) {
          for(j=isL; j<ieL; j++) {
              if(itemL[j] == icN5) {
                 AL[j] = coef;
                 break;
              }
          }
       } else {
          for(j=isU; j<ieU; j++) {
              if(itemU[j] == icN5) {
                 AU[j] = coef;
                 break;
              }
           }
        }
      }
...
```



$$\frac{\phi_{neib[icel][0]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][1]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][2]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

$$\frac{\phi_{neib[icel][3]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

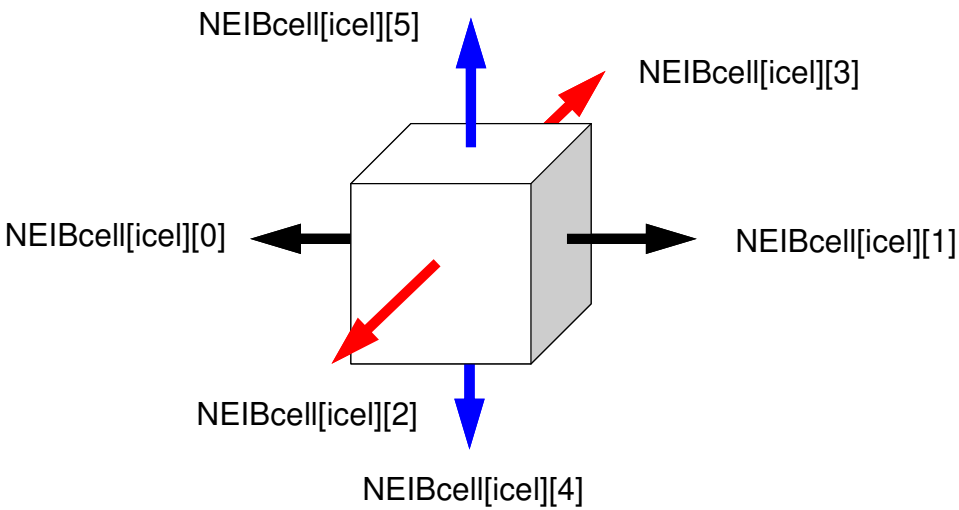
$$\frac{\phi_{neib[icel][4]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+$$

$$\frac{\phi_{neib[icel][5]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+f_{icel}\Delta x\Delta y\Delta z=0$$

poi_gen (6/8)

```
for(icel=0;  icel<N; icel++) {
    ic0  = NEWtoOLD[icel];
    icN1 = NEIBcell[ic0-1][0];
    icN2 = NEIBcell[ic0-1][1];
    icN3 = NEIBcell[ic0-1][2];
    icN4 = NEIBcell[ic0-1][3];
    icN5 = NEIBcell[ic0-1][4];
    icN6 = NEIBcell[ic0-1][5];
    VOL0 = VOLCEL[ic0];

    isL =indexL[icel  ];    ieL =indexL[icel+1];
    isU =indexU[icel  ];    ieU =indexU[icel+1];

    if(icN5 != 0) {
       icN5 = OLDtoNEW[icN5-1];
       coef = RDZ * ZAREA;
       D[icel] -= coef;

       if(icN5-1 < icel) {
          for(j=isL; j<ieL; j++) {
              if(itemL[j] == icN5) {
                 AL[j] = coef;
                 break;
              }
          }
       } else {
          for(j=isU; j<ieU; j++) {
              if(itemU[j] == icN5) {
                 AU[j] = coef;
                 break;
              }
           }
        }
      }
...
```

icN5 < icel
Lower Part



$$\frac{\phi_{neib[icel][0]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][1]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][2]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

$$\frac{\phi_{neib[icel][3]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

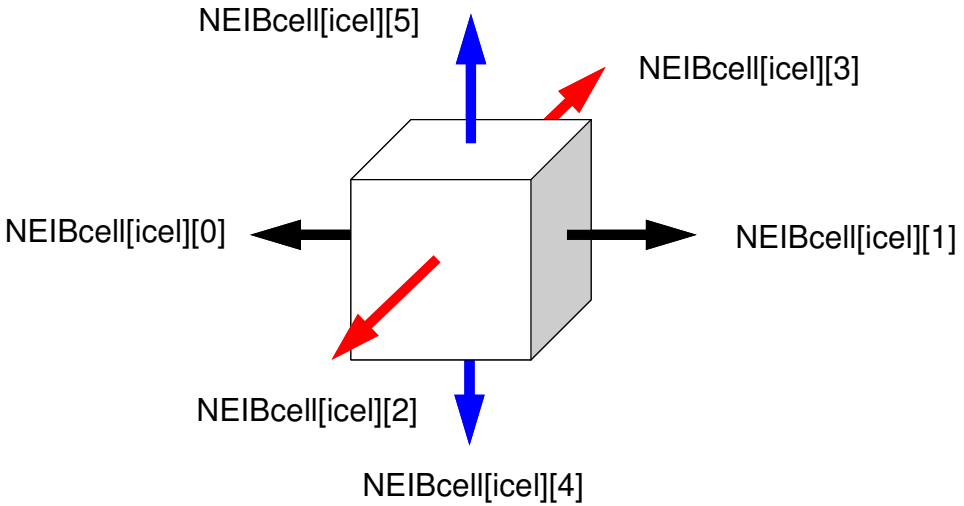
$$\frac{\phi_{neib[icel][4]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+$$

$$\frac{\phi_{neib[icel][5]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+f_{icel}\Delta x\Delta y\Delta z=0$$

poi_gen (6/8)

```
for(icel=0;  icel<N; icel++) {
    ic0  = NEWtoOLD[icel];
    icN1 = NEIBcell[ic0-1][0];
    icN2 = NEIBcell[ic0-1][1];
    icN3 = NEIBcell[ic0-1][2];
    icN4 = NEIBcell[ic0-1][3];
    icN5 = NEIBcell[ic0-1][4];
    icN6 = NEIBcell[ic0-1][5];
    VOL0 = VOLCEL[ic0];

    isL =indexL[icel  ];    ieL =indexL[icel+1];
    isU =indexU[icel  ];    ieU =indexU[icel+1];

    if(icN5 != 0) {
       icN5 = OLDtoNEW[icN5-1];
       coef = RDZ * ZAREA;
       D[icel] -= coef;

       if(icN5-1 < icel) {
          for(j=isL; j<ieL; j++) {
              if(itemL[j] == icN5) {
                 AL[j] = coef;
                 break;
              }
          }
       } else {
          for(j=isU; j<ieU; j++) {
              if(itemU[j] == icN5) {
                 AU[j] = coef;
                 break;
              }
          }
       }
    }
...
```

icN5 > icel
Upper Part



$$\frac{\phi_{neib[icel][0]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][1]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][2]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

$$\frac{\phi_{neib[icel][3]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

$$\frac{\phi_{neib[icel][4]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+$$

$$\frac{\phi_{neib[icel][5]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+f_{icel}\Delta x\Delta y\Delta z=0$$

```
    if(icN6 != 0) {
       icN6 = OLDtoNEW[icN6-1];
       coef = RDZ * ZAREA;
       D[icel] -= coef;

       if(icN6-1 < icel) {
          for(j=isL; j<ieL; j++) {
              if(itemL[j] == icN6) {
                 AL[j] = coef;
                 break;
              }
          }
       } else {
          for(j=isU; j<ieU; j++) {
              if(itemU[j] == icN6) {
                 AU[j] = coef;
                 break;
              }
           }
        }
      }

      ii = XYZ[icO-1][0];
      jj = XYZ[icO-1][1];
      kk = XYZ[icO-1][2];

      BFORCE[icel]= -(double)(ii+jj+kk) * VOLO;

}
```

BFORCE using original mesh ID

poi_gen (7/8)

$$\frac{\phi_{neib[icel][0]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][1]}-\phi_{icel}}{\Delta x}\Delta y\Delta z+$$

$$\frac{\phi_{neib[icel][2]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

$$\frac{\phi_{neib[icel][3]}-\phi_{icel}}{\Delta y}\Delta z\Delta x+$$

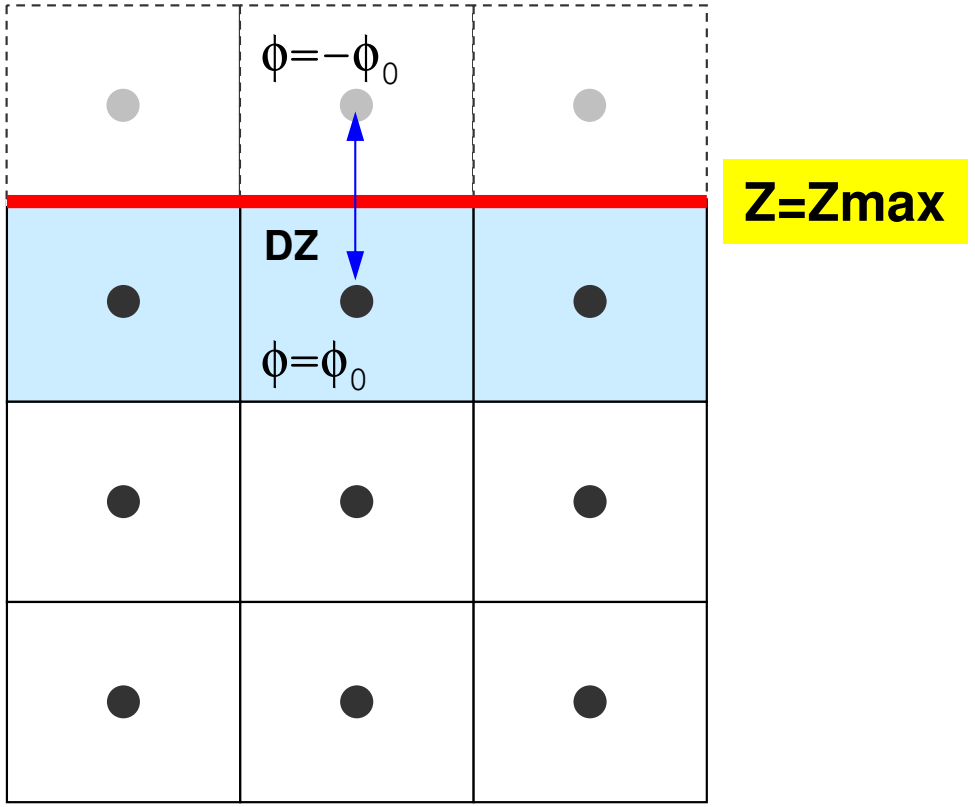
$$\frac{\phi_{neib[icel][4]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+$$

$$\frac{\phi_{neib[icel][5]}-\phi_{icel}}{\Delta z}\Delta x\Delta y+f_{icel}\Delta x\Delta y\Delta z=0$$

```
        for(ib=0; ib<ZmaxCELtot; ib++) {
            ic0  = ZmaxCEL[ib] - 1;
            coef = 2.0 * RDZ * ZAREA;
            icel = OLDtoNEW[ic0];
            D[icel-1] -= coef;
        }


        return 0;
}
```

poi_gen (8/8)

Calculation of Coefficients on Boundary Surface @ $Z=Z_{max}$



<u>1st Order Approximation:</u>

Mirror Image according to $Z=Z_{max}$ surface.

$\phi=-\phi_0$ at the center of the (virtual) mesh

$\phi=0$@$Z=Z_{max}$ surface

Main Program

```
#include <stdio.h> ...

int
main()
{
        double *WK;
        int NPL, NPU; ISET, ITR, IER; icel, ic0, i;
        double xN, xL, xU; Stime, Etime;

        if(INPUT()) goto error;
        if(POINTER_INIT()) goto error;
        if(BOUNDARY_CELL()) goto error;
        if(CELL_METRICS()) goto error;
        if(POI_GEN()) goto error;

        ISET = 0;
        WK = (double *)malloc(sizeof(double)*ICELTOT);
        if(WK == NULL) {
           fprintf(stderr, "Error: %s¥n", strerror(errno));
           goto error;}
        if(solve_ICCG_mc(ICELTOT, NL, NU, indexL, itemL, indexU, itemU,
                         D, BFORCE, PHI, AL, AU, NCOLORtot, COLORindex,
                         EPSICCG, &ITR, &IER)) goto error;

        for(ic0=0; ic0<ICELTOT; ic0++) {
            icel = NEWtoOLD[ic0];
            WK[icel-1] = PHI[ic0];
        }
        for(icel=0; icel<ICELTOT; icel++) {
            PHI[icel] = WK[icel];
        }
        if(OUTUCD()) goto error;
        return 0;
error:
        return -1;
}
```

Matrix, RHS are calculated according to new numbering

solve_ICCG_mc (1/7)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>
#include <math.h> etc.

#include "solver_ICCG.h"

extern int
solve_ICCG_mc(int N, int NL, int NU, int *indexL, int *itemL, int *indexU,
              int *itemU,
              double *D, double *B, double *X, double *AL, double *AU,
              int NCOLORtot, int *COLORindex, double EPS, int *ITR, int *IER)
{

        double **W;
        double VAL, BNRM2, WVAL, SW, RHO, BETA, RHO1, C1, DNRM2, ALPHA, ERR;
        int i, j, ic, ip, L, ip1;
        int R = 0;
        int Z = 1;
        int Q = 1;
        int P = 2;
        int DD = 3;
```

solve_ICCG_mc (2/7)

```
W = (double **)malloc(sizeof(double *)*4);
  if(W == NULL) {
  fprintf(stderr, "Error: %s¥n", strerror(errno));
  return -1;
  }

  for(i=0; i<4; i++) {
      W[i] = (double *)malloc(sizeof(double)*N);
      if(W[i] == NULL) {
         fprintf(stderr, "Error: %s¥n",
         strerror(errno)); return -1;
      }
  }

  for(i=0; i<N; i++) {
      X[i] = 0.0;
      W[1][i] = 0.0;
      W[2][i] = 0.0;
      W[3][i] = 0.0;
  }

  for(ic=0; ic<NCOLORtot; ic++) {
      for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++){
          VAL = D[i];
          for(j=indexL[i]; j<indexL[i+1]; j++) {
              VAL-= AL[j]*AL[j]*W[DD][itemL[j] - 1];
          }
          W[DD][i] = 1.0 / VAL;
       }
  }
```

Incomplete "Modified" Cholesky Factorization

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …
- solve $[M]z^{(i-1)}= r^{(i-1)}$
- $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
- 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
- $q^{(i)}= [A]p^{(i)}$
- $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$
- $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$
- $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$
- check convergence $|r|$

end

solve_ICCG_mc (2/7)

```
W = (double **)malloc(sizeof(double *)*4);
  if(W == NULL) {
  fprintf(stderr, "Error: %s¥n", strerror(errno));
  return -1;
  }

  for(i=0; i<4; i++) {
      W[i] = (double *)malloc(sizeof(double)*N);
      if(W[i] == NULL) {
         fprintf(stderr, "Error: %s¥n",
         strerror(errno)); return -1;
      }
  }

  for(i=0; i<N; i++) {
      X[i] = 0.0;
      W[1][i] = 0.0;
      W[2][i] = 0.0;
      W[3][i] = 0.0;
  }

  for(ic=0; ic<NCOLORtot; ic++) {
      for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++){
          VAL = D[i];
          for(j=indexL[i]; j<indexL[i+1]; j++) {
              VAL-= AL[j]*AL[j]*W[DD][itemL[j] - 1];
          }
          W[DD][i] = 1.0 / VAL;
       }
  }
```

Incomplete "Modified" Cholesky Factorization

$$d_i = \left(a_{ii} - \sum_{k=1}^{i-1} l_{ik}^{\ 2} \cdot d_k \right)^{-1} = l_{ii}^{\ -1}$$

$$d_i = \left(a_{ii} - \sum_{k=1}^{i-1} a_{ik}^{\ 2} \cdot d_k \right)^{-1} = l_{ii}^{\ -1}$$

W[DD][i]:	d_i
D[i]:	a_{ii}
itemL[j]:	k
AL[j]:	a_{ik}

Incomplete “Modified” Cholesky Factorization

```
for(i=0; i<N; i++) {
    VAL = D[i];
    for(j=indexL[i]; j<indexL[i+1]; j++) {
        VAL-= AL[j]*AL[j]*W[DD][itemL[j] - 1];
    }
    W[DD][i] = 1.0 / VAL;
}
```

```
for(ic=0; ic<NCOLORtot; ic++) {
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++){
        VAL = D[i];
          for(j=indexL[i]; j<indexL[i+1]; j++) {
              VAL-= AL[j]*AL[j]*W[DD][itemL[j] - 1];
          }
          W[DD][i] = 1.0 / VAL;
     }
}
```

Mesh “i” and “itemL(k)” in RHS belong to different “colors”.

NO data dependency.

solve_ICCG_mc (3/7)

```
for(i=0; i<N; i++) {
    VAL = D[i] * X[i];

    for(j=indexL[i]; j<indexL[i+1]; j++) {
        VAL += AL[j] * X[itemL[j]-1];
    }

    for(j=indexU[i]; j<indexU[i+1]; j++) {
        VAL += AU[j] * X[itemU[j]-1];
    }
    W[R][i] = B[i] - VAL;
}

BNRM2 = 0.0;
for(i=0; i<N; i++) {
    BNRM2 += B[i]*B[i];
}
```

Compute $r^{(0)}= b-[A]x^{(0)}$

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

solve_ICCG_mc (4/7)

```
*ITR = N;
for(L=0; L<(*ITR); L++) {


for(i=0; i<N; i++) {
    W[Z][i] = W[R][i];
}

for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}

for (ic=NCOLORtot-1; ic>=0; ic--){
     for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
         SW = 0.0;
         for(j=indexU[i]; j<indexU[i+1]; j++) {
             SW += AU[j] * W[Z][itemU[j]-1];
         }
         W[Z][i]= W[Z][i]  - W[DD][i] * SW;
     }
}
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

  solve $[M]z^{(i-1)}= r^{(i-1)}$

  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$

  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

  $q^{(i)}= [A]p^{(i)}$

  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$

  check convergence $|r|$

end

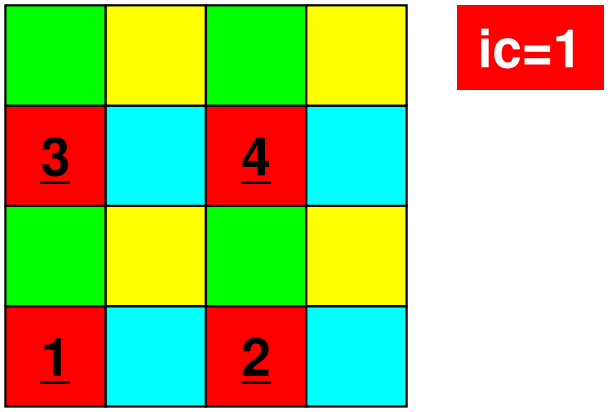
solve_ICCG_mc (4/7)

```
*ITR = N;
for(L=0; L<(*ITR); L++) {


for(i=0; i<N; i++) {
    W[Z][i] = W[R][i];
}

for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}

for (ic=NCOLORtot-1; ic>=0; ic--){
     for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
         SW = 0.0;
         for(j=indexU[i]; j<indexU[i+1]; j++) {
             SW += AU[j] * W[Z][itemU[j]-1];
         }
         W[Z][i]= W[Z][i]  - W[DD][i] * SW;
     }
}
```

$$(M)\{z\}=\left(LDL^T\right)\{z\}=\{r\}$$

$$(L)\{z\}=\{r\}$$

Forward Substitution



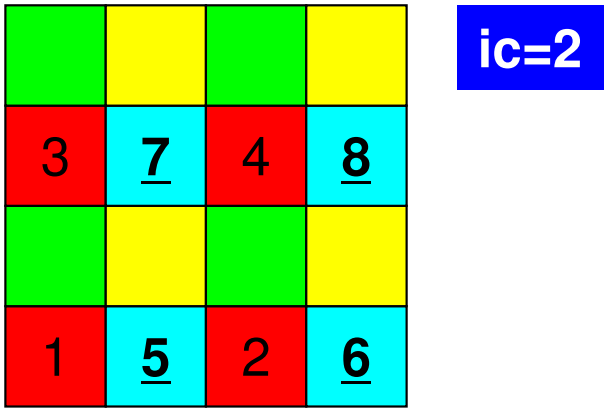
solve_ICCG_mc (4/7)

```
*ITR = N;
for(L=0; L<(*ITR); L++) {


for(i=0; i<N; i++) {
    W[Z][i] = W[R][i];
}

for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}

for (ic=NCOLORtot-1; ic>=0; ic--){
     for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
         SW = 0.0;
         for(j=indexU[i]; j<indexU[i+1]; j++) {
             SW += AU[j] * W[Z][itemU[j]-1];
         }
         W[Z][i]= W[Z][i]  - W[DD][i] * SW;
     }
}
```

$$(M)\{z\}=\left(LDL^T\right)\{z\}=\{r\}$$

$$(L)\{z\}=\{r\}$$

Forward Substitution



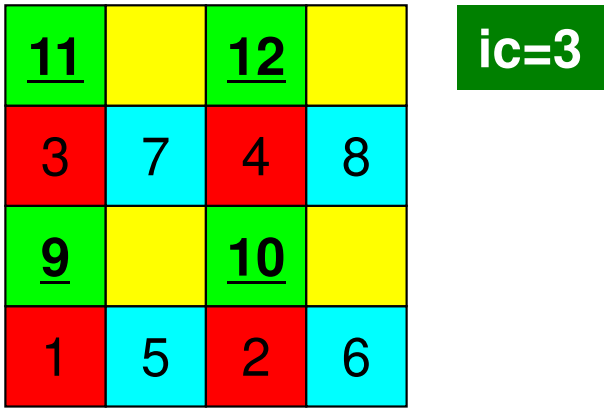
solve_ICCG_mc (4/7)

```
*ITR = N;
for(L=0; L<(*ITR); L++) {


for(i=0; i<N; i++) {
    W[Z][i] = W[R][i];
}

for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}

for (ic=NCOLORtot-1; ic>=0; ic--){
     for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
         SW = 0.0;
         for(j=indexU[i]; j<indexU[i+1]; j++) {
             SW += AU[j] * W[Z][itemU[j]-1];
         }
         W[Z][i]= W[Z][i]  - W[DD][i] * SW;
     }
}
```

$$(M)\{z\}=\left(LDL^T\right)\{z\}=\{r\}$$

$$(L)\{z\}=\{r\}$$

Forward Substitution



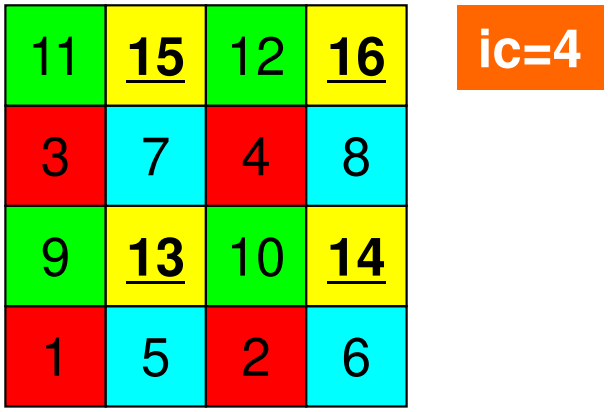
solve_ICCG_mc (4/7)

```
*ITR = N;
for(L=0; L<(*ITR); L++) {


for(i=0; i<N; i++) {
    W[Z][i] = W[R][i];
}

for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}

for (ic=NCOLORtot-1; ic>=0; ic--){
     for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
         SW = 0.0;
         for(j=indexU[i]; j<indexU[i+1]; j++) {
             SW += AU[j] * W[Z][itemU[j]-1];
         }
         W[Z][i]= W[Z][i]  - W[DD][i] * SW;
     }
}
```

$$(M)\{z\}=\left(LDL^T\right)\{z\}=\{r\}$$

$$(L)\{z\}=\{r\}$$

Forward Substitution



solve_ICCG_mc (4/7)

```
*ITR = N;
for(L=0; L<(*ITR); L++) {


for(i=0; i<N; i++) {
    W[Z][i] = W[R][i];
}

for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}

for (ic=NCOLORtot-1; ic>=0; ic--){
     for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
         SW = 0.0;
         for(j=indexU[i]; j<indexU[i+1]; j++) {
             SW += AU[j] * W[Z][itemU[j]-1];
         }
         W[Z][i]= W[Z][i]  - W[DD][i] * SW;
     }
}
```

$$(M)\{z\}=(LDL^T)\{z\}=\{r\}$$

$$(L)\{z\}=\{r\}$$

$$(DL^T)\{z\}=\{z\}$$

Forward Substitution

Backward Substitution

solve_ICCG_mc (4/7)

```
*ITR = N;
for(L=0; L<(*ITR); L++) {


for(i=0; i<N; i++) {
    W[Z][i] = W[R][i];
}

for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
                                                  ; j++) {
                                                 [j]-1];

for (ic=NCOLORtot-1; ic>=0; ic--){
     for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++) {
         SW = 0.0;
         for(j=indexU[i]; j<indexU[i+1]; j++) {
             SW += AU[j] * W[Z][itemU[j]-1];
         }
         W[Z][i]= W[Z][i]  - W[DD][i] * SW;
     }
}
```

$$(M)\{z\}=(LDL^T)\{z\}=\{r\}$$

If order of computations in same color is changed: NO effect

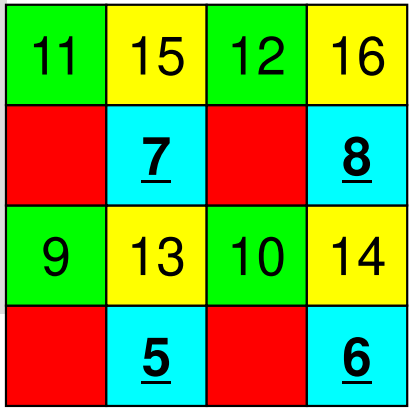
i= COLOR[ic], COLOR(ic+1)
i= COLOR[ic+1], COLOR[ic], -1

$$(L)\{z\}=\{r\}$$

$$(DL^T)\{z\}=\{z\}$$

Forward Substitution

Backward Substitution



ic=2

Forward/Backward Substitution
前進後退代入

```
for(i=0; i<N; i++) {
    WVAL = W[Z][i];
    for(j=indexL[i]; j<indexL[i+1]; j++) {
        WVAL -= AL[j] * W[Z][itemL[j]-1];
    }
    W[Z][i] = WVAL * W[DD][i];
}

for(i=N-1; i>=0; i--) {
    SW = 0.0;
    for(j=indexU[i]; j<indexU[i+1]; j++) {
        SW += AU[j] * W[Z][itemU[j]-1];
    }
    W[Z][i]= W[Z][i]  - W[DD][i] * SW;
}
```

```
for(ic=0; ic<NCOLORtot; ic++){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++){
        WVAL = W[Z][i];
        for(j=indexL[i]; j<indexL[i+1]; j++) {
            WVAL -= AL[j] * W[Z][itemL[j]-1];
        }
        W[Z][i] = WVAL * W[DD][i];
    }
}

for (ic=NCOLORtot-1; ic>=0; ic--){
    for(i=COLORindex[ic]; i<COLORindex[ic+1]; i++){
        SW = 0.0;
        for(j=indexU[i]; j<indexU[i+1]; j++) {
            SW += AU[j] * W[Z][itemU[j]-1];
        }
        W[Z][i]= W[Z][i]  - W[DD][i] * SW;
    }
}
```

solve_ICCG_mc (5/7)

```
/****************
 * RHO = {r}{z} *
 ****************/

RHO = 0.0;
for(i=0; i<N; i++) {
    RHO += W[R][i] * W[Z][i];
}

/*********************************
 * {p}  = {z} if      ITER=0    *
 * BETA = RHO / RHO1  otherwise *
 *********************************/

if(L == 0) {
   for(i=0; i<N; i++) {
       W[P][i] = W[Z][i];
   }
   } else {
   BETA = RHO / RHO1;
   for(i=0; i<N; i++) {
       W[P][i] = W[Z][i] + BETA * W[P][i];
   }
}
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

  solve $[M]z^{(i-1)}= r^{(i-1)}$

  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$

  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

  $q^{(i)}= [A]p^{(i)}$

  $\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$

  check convergence $|r|$

end

solve_ICCG_mc (6/7)

```
/****************
 * {q} = [A]{p} *
 ****************/

for(i=0; i<N; i++) {
    VAL = D[i] * W[P][i];
    for(j=indexL[i]; j<indexL[i+1]; j++) {
        VAL += AL[j] * W[P][itemL[j]-1];
    }
    for(j=indexU[i]; j<indexU[i+1]; j++) {
        VAL += AU[j] * W[P][itemU[j]-1];
    }
    W[Q][i] = VAL;
}
```

Compute $r^{(0)}= b-[A]x^{(0)}$

for i= 1, 2, …

  solve $[M]z^{(i-1)}= r^{(i-1)}$

  $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$

  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

  $q^{(i)}= [A]p^{(i)}$

  $\alpha_i\ = \rho_{i-1}/p^{(i)}q^{(i)}$

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$

  check convergence $|r|$

end

solve_ICCG_mc (7/7)

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

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

DNRM2 = 0.0;
for(i=0; i<N; i++) {
    DNRM2 += W[R][i]*W[R][i];
}

ERR = sqrt(DNRM2/BNRM2);
if((L+1)%100 ==1) {
   fprintf(stderr, "%5d%16.6e¥n", L+1, ERR);
}
if(ERR < EPS) {
   *IER = 0; goto N900;
  } else {
   RHO1 = RHO;
  }
}
*IER = 1;
```

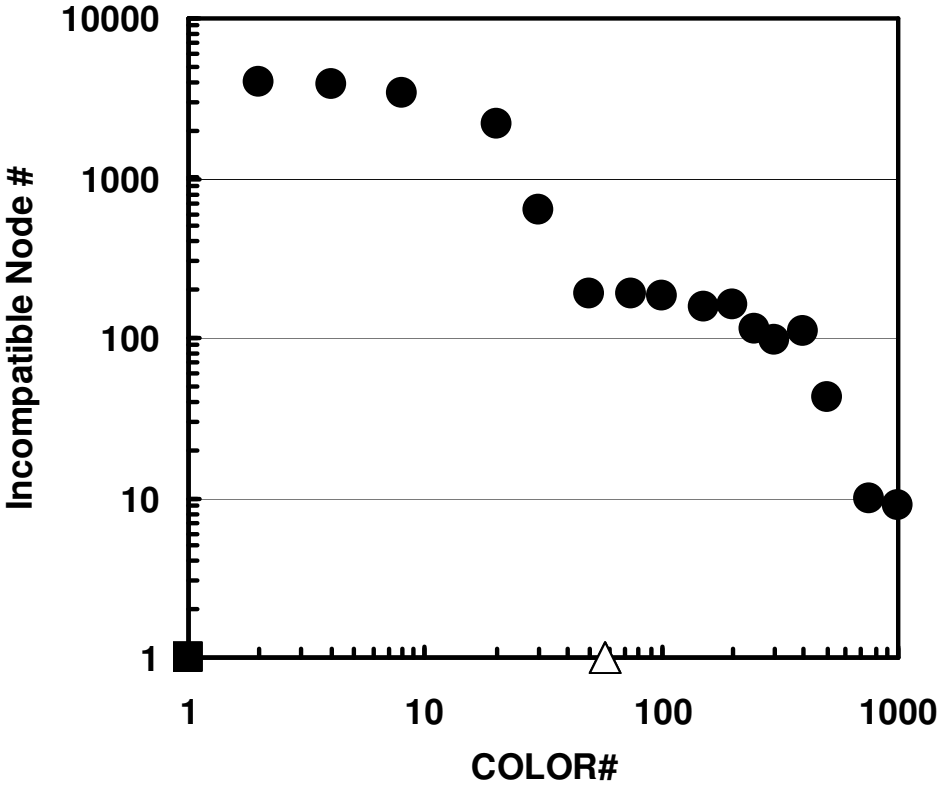
Compute $r^{(0)}= b-[A]x^{(0)}$

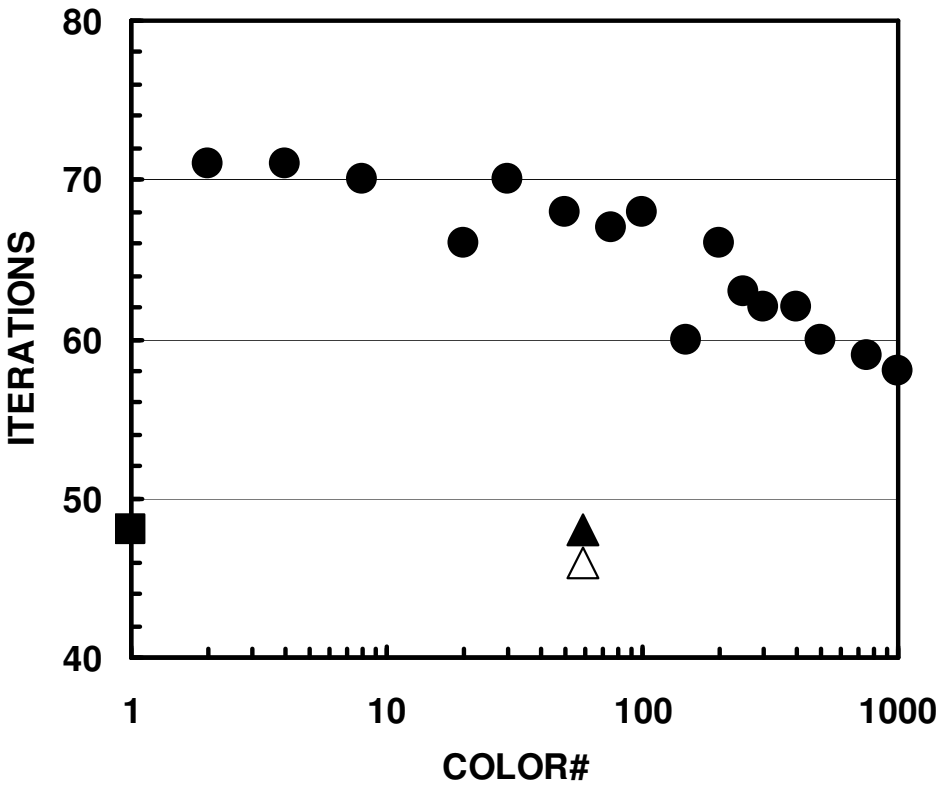
for i= 1, 2, …

- solve $[M]z^{(i-1)}= r^{(i-1)}$
- $\rho_{i-1}= r^{(i-1)}\ z^{(i-1)}$
- 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
- $q^{(i)}= [A]p^{(i)}$
- **$\alpha_i = \rho_{i-1}/p^{(i)}q^{(i)}$**
- **$x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$**
- **$r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$**
- **check convergence |r|**

end

Effect of Color Number on Convergence of ICCG





(20^3=8,000 meshe, EPSICCG=10^{-8})

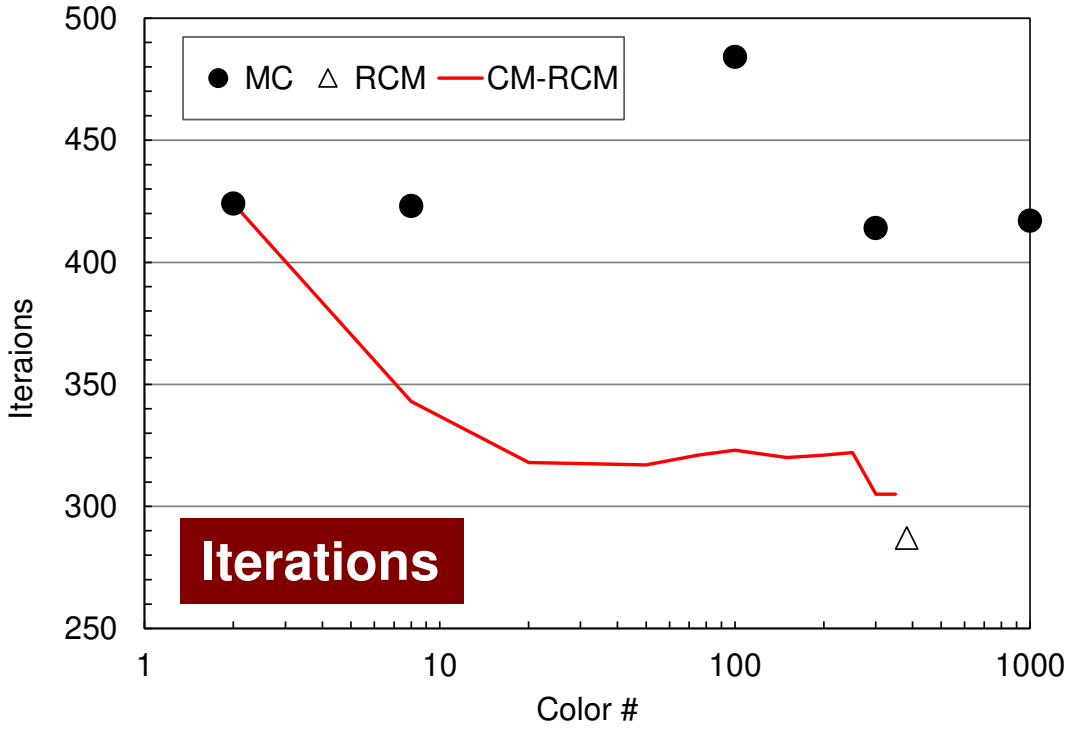
(■:ICCG(L1), ●:ICCG-MC, ▲:ICCG-CM, △:ICCG-RCM)

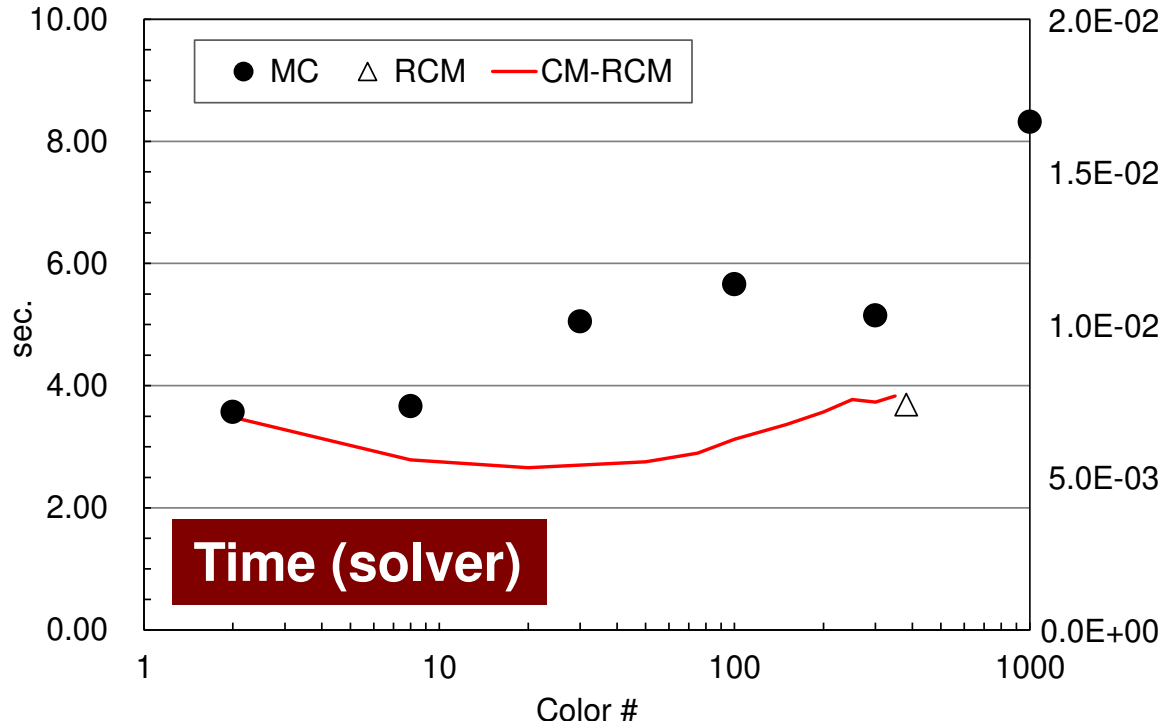
- Remedy for Data Dependency
- Ordering/Reordering
  - Red-Black, Multicoloring (MC)
  - Cuthill-McKee (CM), Reverse-CM (RCM)
  - Reordering and Convergence
- Implementation
- ICCG with Reordering
- **ICCG with Reordering on Multicores**
  - **Just apply OpenMP to L2-sol**

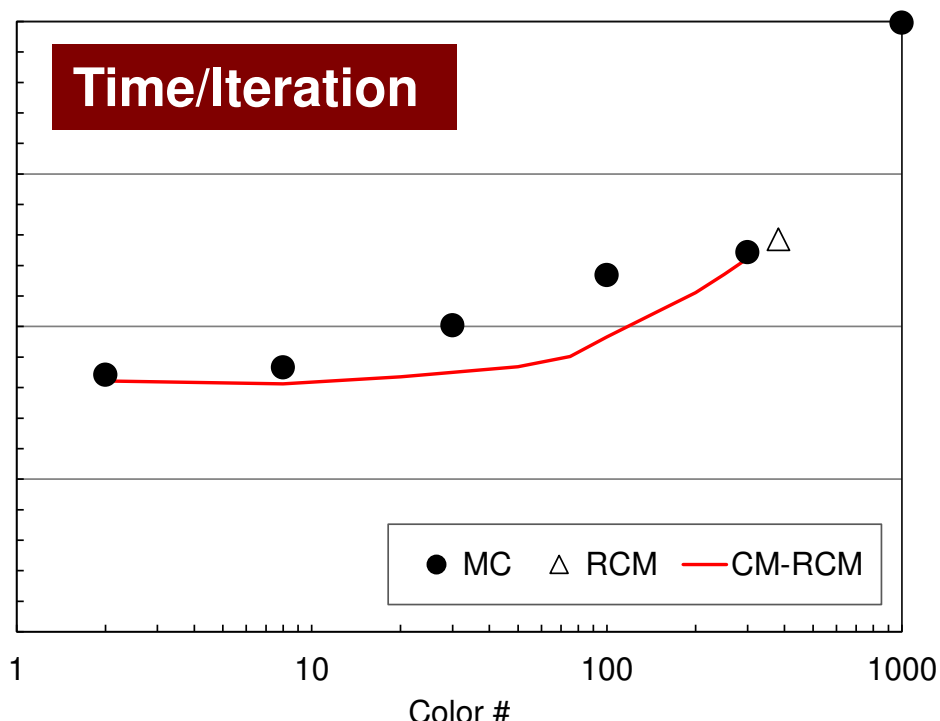
Odyssey

1-CMG/12-cores, 128^3

(●:MC, △:RCM, -:CM-RCM)







OBCX

Intel Xeon CLX 1-soket/24-cores, 128^3

(●:MC, △:RCM, -:CM-RCM)