OpenMP（Only Solver）（F・C）

```
>$ cd ~/pfem/pfem3d/src1
>$ make
>$ cd ../run
>$ ls sol1
      sol1

>$ cd ../pmesh

<Parallel Mesh Generation>

>$ cd ../run

<modify go1.sh>

>$ pjsub go1.sh
```

OpenMP (Solver + Matrix Assembly)（Fortran Only）

```
>$ cd ~/pfem/pfem3d/src2
>$ make
>$ cd ../run
>$ ls sol2
      sol2

>$ cd ../pmesh

<Parallel Mesh Generation>

>$ cd ../run

<modify go1.sh>

>$ pjsub go1.sh
```

HB 16×1, 1-node

Time for CG Solver (Iterations)

```
NX  NX  NX
 1   1   1
pcube
```

	NX=128 2,097,152 nodes	NX=129 2,146,689 nodes
Original	14.7 (392)	6.28 (396)

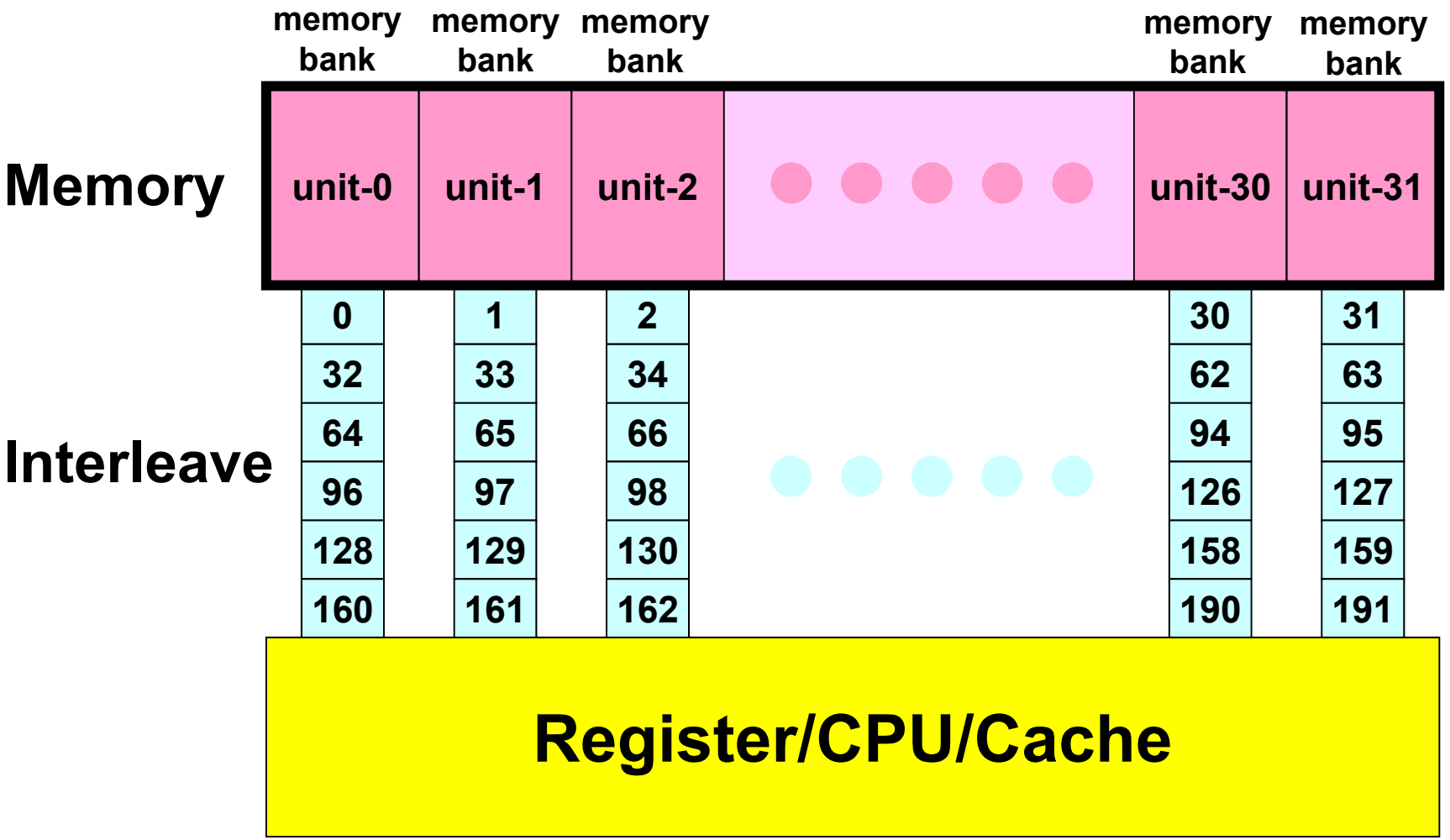
- Why ?
  - Bank Conflict
  - Cache Thrashing

Memory Interleaving/Bank Conflict

- Memory Interleaving
  - Method for fast data transfer to/from memory.
  - Parallel I/O for multiple memory banks.
- Memory Bank
  - Unit for memory management, small pieces of memory
    - Usually, there are 2^n independent modules.
  - Single bank can execute a single reading or writing at one time. Therefore, performance gets worse if data components on same bank are accessed simultaneously.
  - For example, “bank conflict” occurs if off-set of data access is 32 (in next page).
    - Remedy: Change of array size, loop exchange etc.

Bank Conflict

If off-set of data access is 32, only a single bank is utilized



Avoiding Bank Conflict

×

```
REAL*8 A(32,10000)

k= N
do i= 1, 10000
   A(k,i)= 0.d0
enddo
```

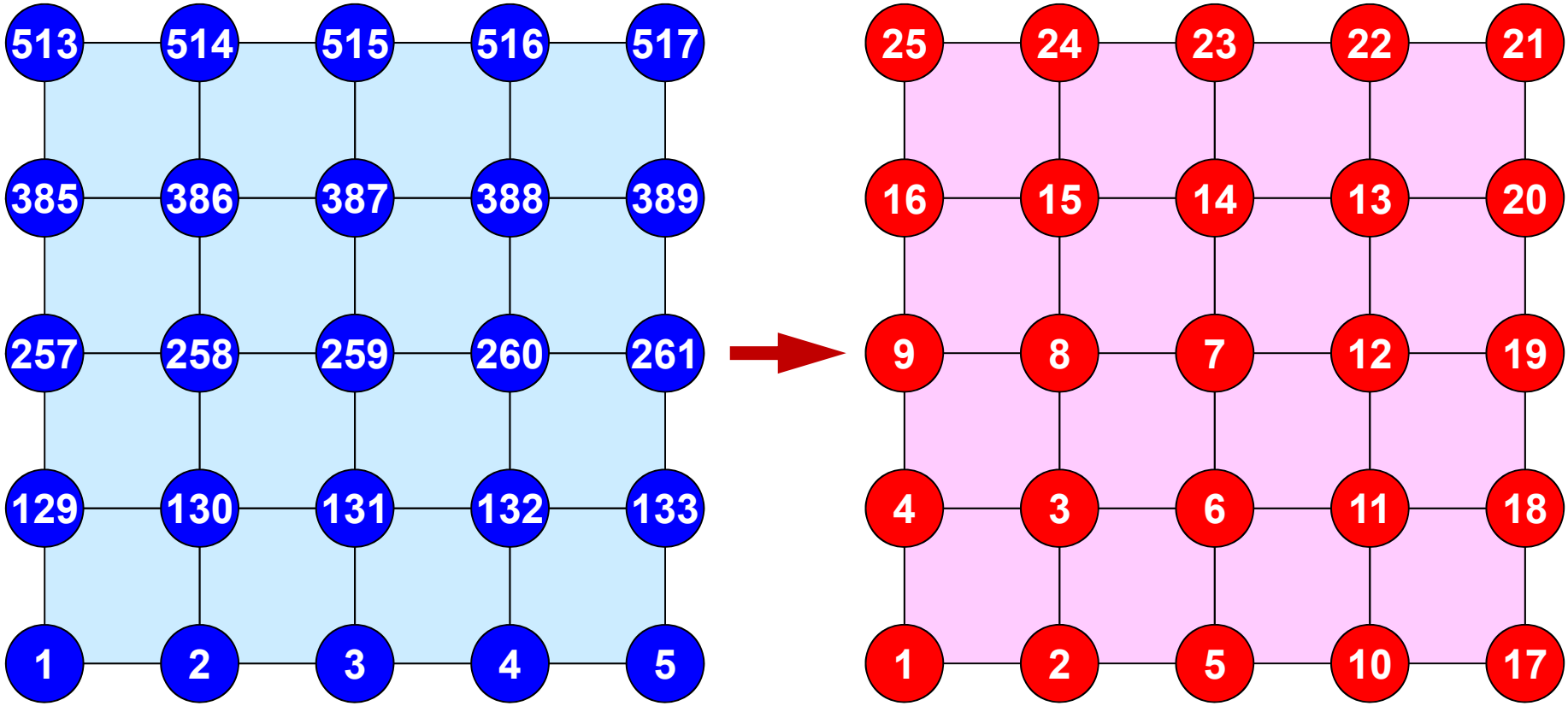
○

```
REAL*8 A(33,10000)

k= N
do i= 1, 10000
   A(k,i)= 0.d0
enddo
```

- Arrays with size of 2^n should be avoided.

- Bank conflict always occurs when non-zero off-diagonals are accessed, 16 threads
- Remedy for Bank Conflict
  - Padding by compiler
  - Reordering, such as CM, is effective
    - Hyperline, Hyperplane



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
  - Bandwidth and profile of matrices affects convergence.

Fundamental Algorithm for CM Method (Cuthill-McKee)

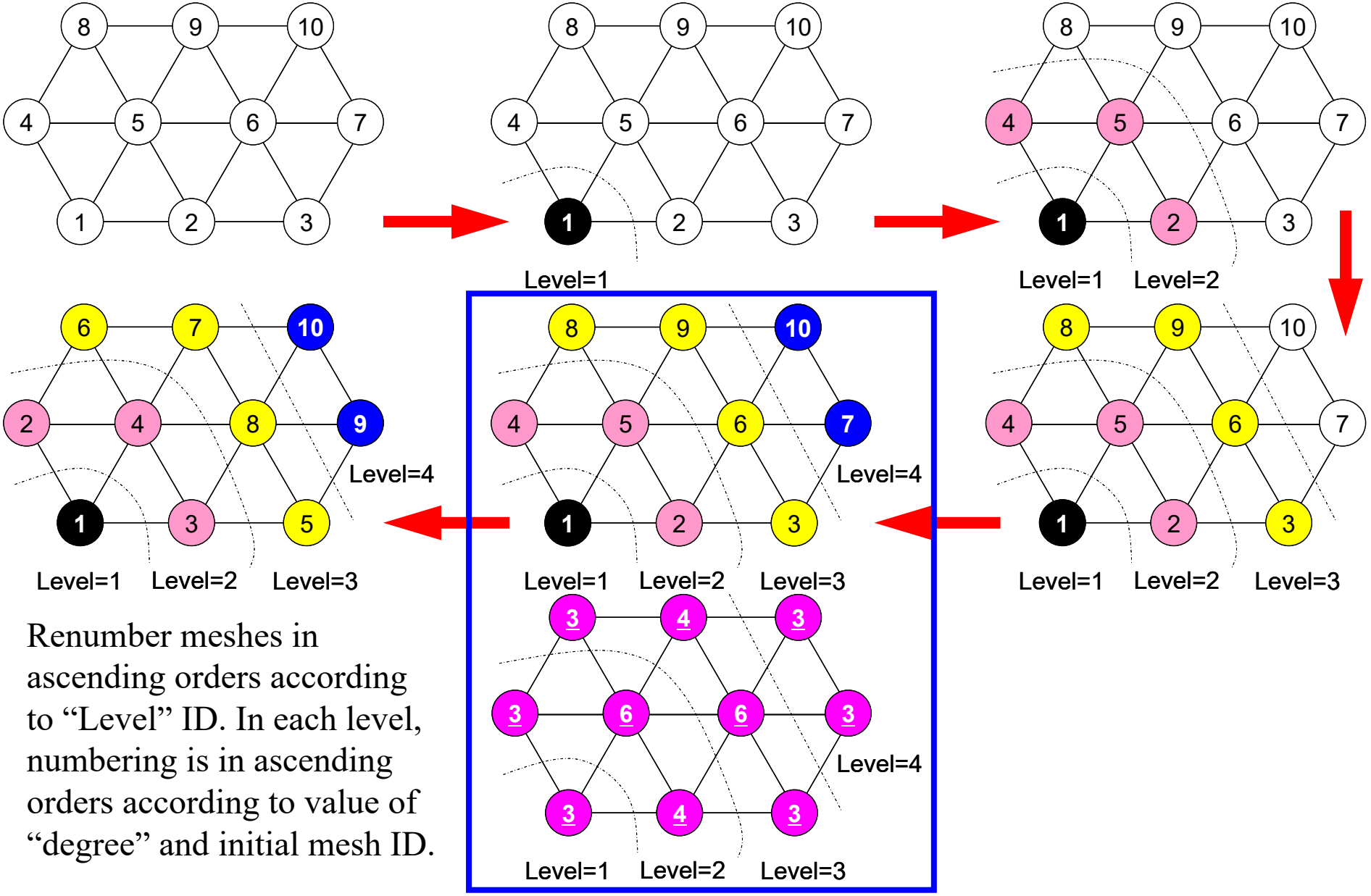
① The mesh with minimum value of “degree” is set to “Level=1”.

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.

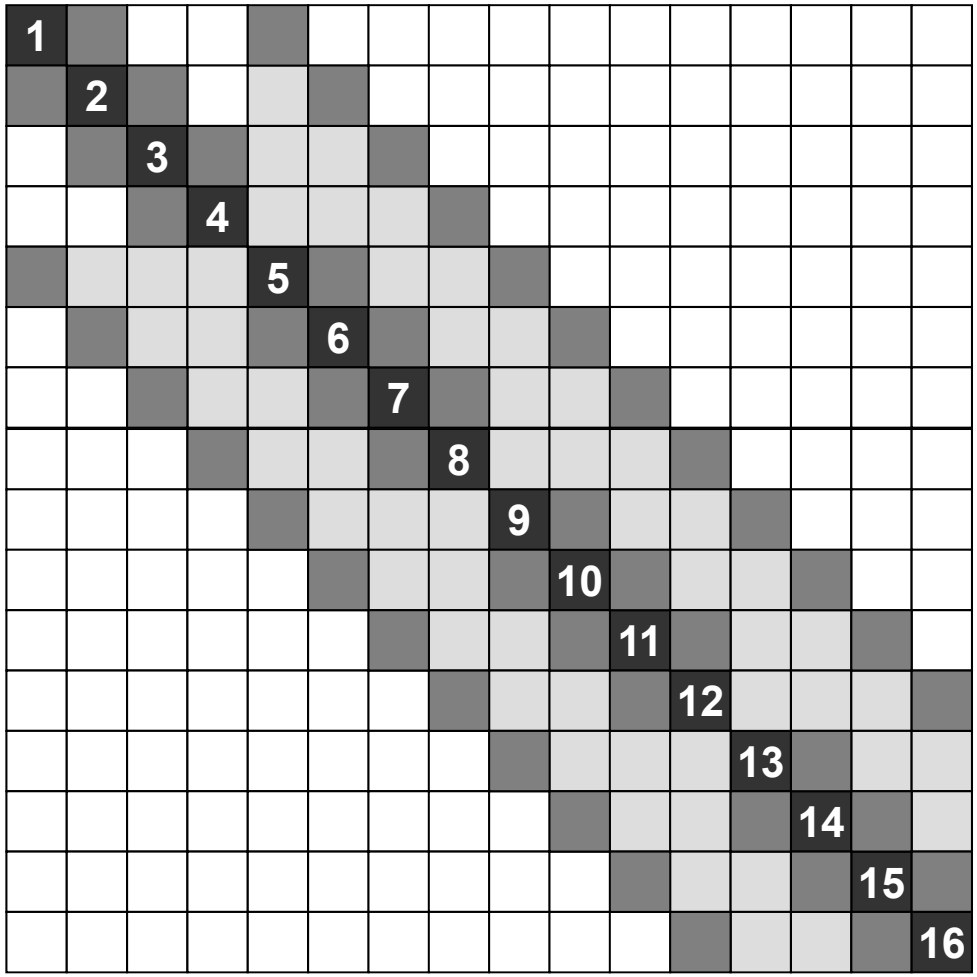
RCM: Reverse Cuthill-McKee

- 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 are fewer than CM

Initial Matrix

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

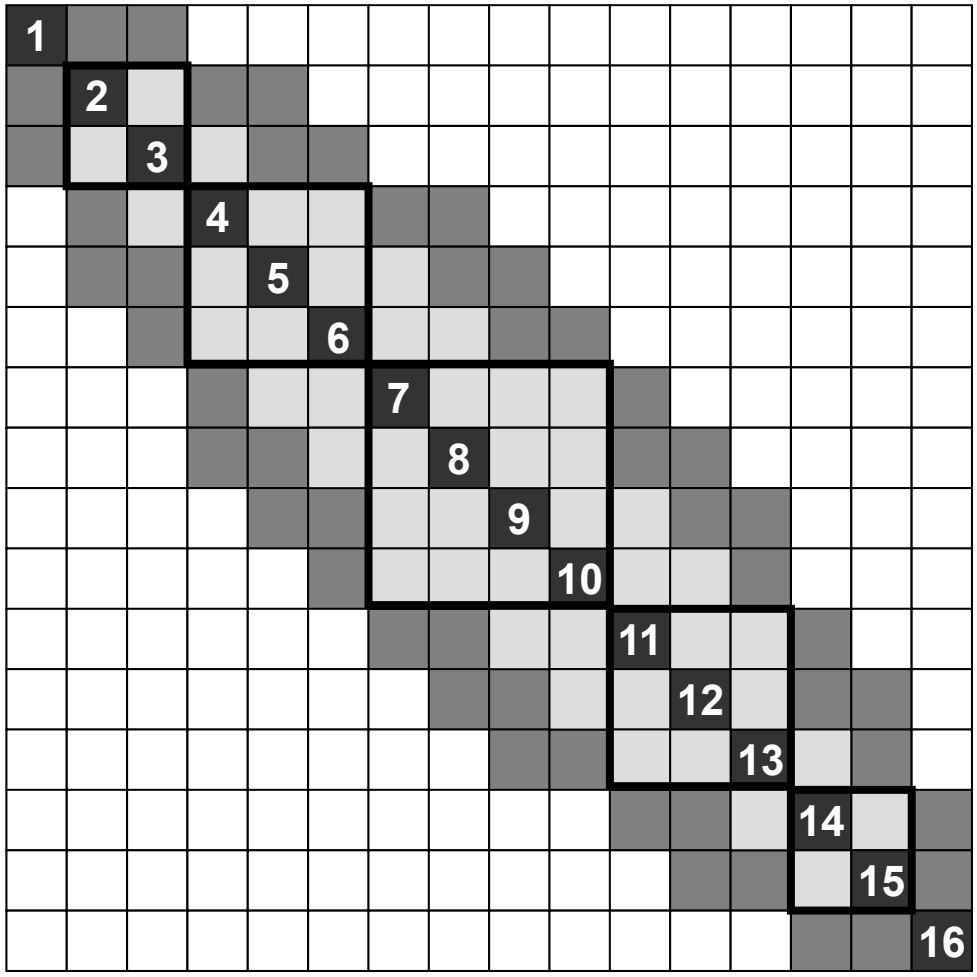
Bandwidth 4
Profile 51
Fill-in 54



CM

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

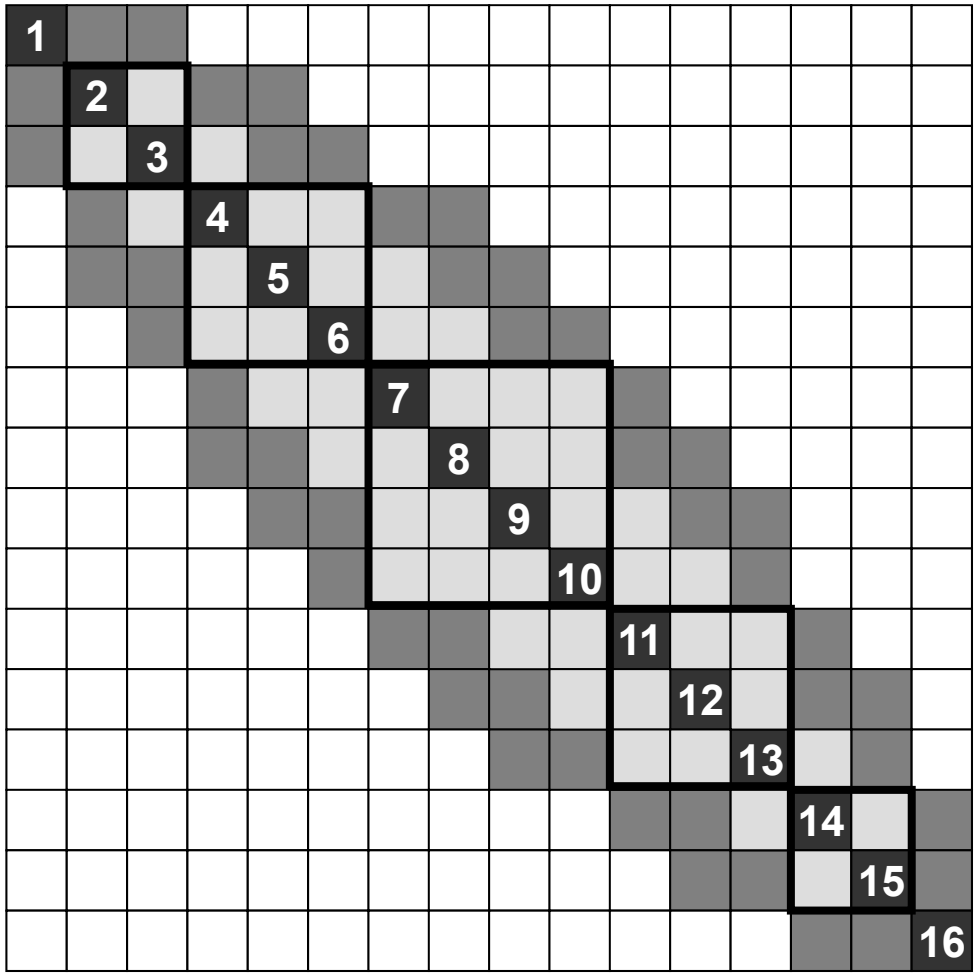
Bandwidth 4
Profile 46
Fill-in 44



RCM

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

Bandwidth 4
Profile 46
Fill-in 44



Implementation of CM (1/3)

```
!C
!C +-------+
!C | INIT. | The node with minimum degree
!C +-------+ New ID= 1
!C===
      allocate (IW(NP))
      IW = 0

       INmin= NP
      NODmin= 0

      do i= 1, N
        if (INLU(i).lt.INmin) then
          INmin = INLU(i)
          NODmin= i
        endif
      enddo
 200  continue


      if (NODmin.eq.0) NODmin= 1

      IW(NODmin)= 1

      NEWtoOLD(1     )= NODmin
      OLDtoNEW(NODmin)= 1

      icol= 1
!C===
```

NODmin: Node ID with MIN. degree (Old)

OLDtoNEW(Old ID)= New ID
NEWtoOLD(New ID)= Old ID

IW(Old ID)= Level for CM

```
!C
!C +---------------+
!C | CM-redordering |
!C +---------------+
!C===
      icouG= 1
      do icol= 2, N
        do i= 1, N
          if (IW(i).eq.icol-1) then
            do k= 1, INLU(i)
              in= IALU(i,k)
              if (IW(in).eq.0) then
                IW(in)= icol
                icouG= icouG + 1
              endif
            enddo
          endif
        enddo
        if (icouG.eq.N) exit
      enddo

      NCOLORtot= icol
      icoug    = 0
      do ic= 1, NCOLORtot
        do i= 1, N
          if (IW(i).eq.ic) then
            icoug= icoug + 1
            NEWtoOLD(icoug)= i
            OLDtoNEW(i    )= icoug
          endif
        enddo
      enddo
!C===
```

Implementation of CM (2/3)

```
!C
!C +-------+
!C | INIT. |
!C +-------+
!C===
      allocate (IW(NP))
      IW = 0

       INmin= NP
      NODmin= 0

      do i= 1, N
        if (INLU(i).lt.INmin) then
          INmin = INLU(i)
          NODmin= i
        endif
      enddo
 200  continue


      if (NODmin.eq.0) NODmin= 1

      IW(NODmin)= 1

      NEWtoOLD(1     )= NODmin
      OLDtoNEW(NODmin)= 1

      icol= 1
!C===
```

NODmin: Node ID with MIN. degree (Old)

OLDtoNEW(Old ID)= New ID
NEWtoOLD(New ID)= Old ID

IW(Old ID)= Level for CM

```
!C
!C +---------------+
!C | CM-redordering |
!C +---------------+
!C===
      icouG= 1                #Node, leveled
      do icol= 2, N
        do i= 1, N
          if (IW(i).eq.icol-1) then
            do k= 1, INLU(i)
              in= IALU(i,k)
              if (IW(in).eq.0) then
                IW(in)= icol
                icouG= icouG + 1
              endif
            enddo
          endif
        enddo
        if (icouG.eq.N) exit
      enddo

      NCOLORtot= icol
      icoug    = 0
      do ic= 1, NCOLORtot
        do i= 1, N
          if (IW(i).eq.ic) then
            icoug= icoug + 1
            NEWtoOLD(icoug)= i
            OLDtoNEW(i    )= icoug
          endif
        enddo
      enddo
!C===
```

Implementation of CM (3/3)

```
!C
!C +-------+
!C | INIT. |
!C +-------+
!C===
      allocate (IW(NP))
      IW = 0

       INmin= NP
      NODmin= 0

      do i= 1, N
        if (INLU(i).lt.INmin) then
          INmin = INLU(i)
          NODmin= i
        endif
      enddo
 200  continue


      if (NODmin.eq.0) NODmin= 1

      IW(NODmin)= 1

      NEWtoOLD(1     )= NODmin
      OLDtoNEW(NODmin)= 1

      icol= 1
!C===
```

NODmin: Node ID with MIN. degree (Old)

OLDtoNEW(Old ID)= New ID
NEWtoOLD(New ID)= Old ID

IW(Old ID)= Level for CM

```
!C
!C +---------------+
!C | CM-redordering |
!C +---------------+
!C===
      icouG= 1                #Node, leveled
      do icol= 2, N
        do i= 1, N
          if (IW(i).eq.icol-1) then
            do k= 1, INLU(i)
              in= IALU(i,k)
              if (IW(in).eq.0) then
                IW(in)= icol
                icouG= icouG + 1
              endif
            enddo
          endif
        enddo
        if (icouG.eq.N) exit
      enddo

      NCOLORtot= icol         Renumbering by Level
      icoug    = 0             "Degree"  is not
      do ic= 1, NCOLORtot     considered
        do i= 1, N
          if (IW(i).eq.ic) then
            icoug= icoug + 1
            NEWtoOLD(icoug)= i
            OLDtoNEW(i    )= icoug
          endif
        enddo
      enddo
!C===
```

HB 16×1, 1-node

Time for CG Solver (Iterations)

```
NX  NX  NX
 1   1   1
pcube
```

	NX=128 2,097,152 nodes	NX=129 2,146,689 nodes
Original	14.7 (392)	6.28 (396)
with CM	6.33 (392)	6.26 (396)

- Much better than before.
- Still the case with “N=128^3” is faster than that with “N=129^3”.

Cache Thrashing

- FX10: L1D cache with 32KB for each core, 2-way
  - n-way set associative（n群連想記憶式）
  - Cache is divided into “n” banks
  - Each bank is divided into “cache lines”
    - Number of Cache Lines, Size of Cache Line (128 bytes for FX10) ⇒ 2^m
- This “2-way” cache is very harmful
  - If “$N=2^m$“, memory addresses of `W(i,P)`, `W(i,Q)`, `W(i,R)` map to the same cache address in CG computation.
  - Cache Thrashing: Lower Performance
    - `R=1, P=2, Q=3`
    - `X(i)` is not affected

```
!$omp parallel do private(i)
      do i= 1, N
        X(i)   = X(i)   + ALPHA * W(i,P)
        W(i,R)= W(i,R) - ALPHA * W(i,Q)
      enddo
```

Remedy

- If the loop is split into 2 loops, up to 2 cache lines of W are referred. Therefore, cache thrashing does not occur (Remedy-1).

```
!$omp parallel do private(i)
      do i= 1, N
         X(i)  = X (i)   + ALPHA * W(i,P)
      enddo

!$omp parallel do private(i)
      do i= 1, N
        W(i,R)= W(i,R) - ALPHA * W(i,Q)
      enddo
```

- **If "N=2^m", certain numbers (e.g. 64, 128 …) can be added to N. Thus, size of the array is not equal to 2^m, and cache thrashing is avoided (Remedy-2).**
  - **No such operation is needed for X**

```
N2=128
allocate (W(N+N2,4))
```

HB 16×1, 1-node

Time for CG Solver (Iterations)

```
NX   NX   NX
 1    1    1
pcube
```

	NX=128 2,097,152 nodes	NX=129 2,146,689 nodes
Original	14.7 (392)	6.28 (396)
+CM	6.33 (392)	6.26 (396)
+ Remedy-2	6.08 (392)	6.24 (396)

- Reasonable
- Problem is that the cache is 2-way on FX10
- Most of modern architectures based on 4- or 8-way
- FX-100 (successor of FX10) has 4-way cache

Final Version
（Fortran Only）

```
>$ cd ~/pfem/pfem3d/src3
>$ make
>$ cd ../run
>$ ls sol3
      sol3

>$ cd ../pmesh

<Parallel Mesh Generation>

>$ cd ../run

<modify go1.sh>

>$ pjsub go1.sh
```