三次元並列有限要素法への OpenMP/MPIハイブリッド 並列プログラミングモデル適用

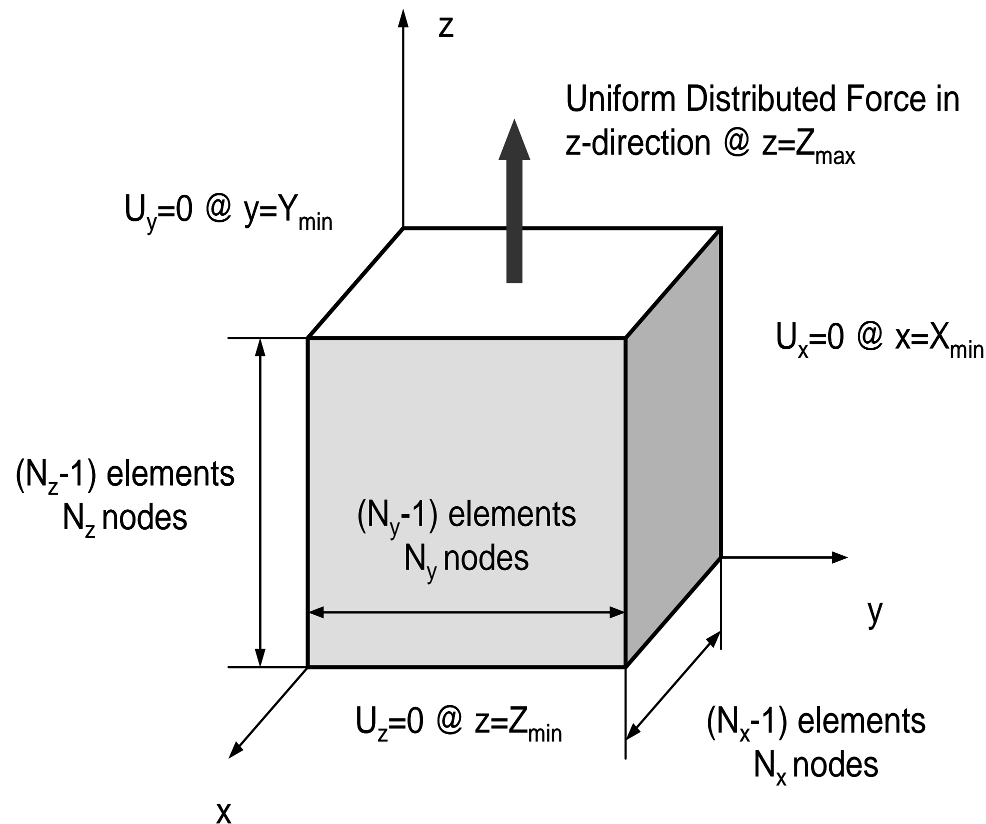
中島 研吾
東京大学情報基盤センター

RIKEN AICS Spring School 2014

Hybrid並列プログラミング

- スレッド並列＋メッセージパッシング
  – OpenMP+ MPI
  – CUDA + MPI, OpenACC + MPI
- 個人的には自動並列化＋MPIのことを「ハイブリッド」とは呼んでほしくない
  – 自動並列化に頼るのは危険である
  – 東大センターでは現在自動並列化機能はコンパイラの要件にしていない（加点すらしない）
    - 利用者にももちろん推奨していない
- OpenMPがMPIより簡単ということはない
  – データ依存性のない計算であれば，機械的にOpenMP指示文を入れれば良い
  – NUMAになるとより複雑：First Touch Data Placement

対象とする問題



- 境界条件
  - 対称条件
    - U_X=0@X=0
    - U_Y=0@Y=0
    - U_Z=0@Z=0
  - 等分布荷重
    - F_Z=1@Z=Z_{max}

- 弾性体
  - ヤング率：E (=1.00), ポアソン比：ν (=0.30)
- 直方体
  - 一辺長さ1の立方体（六面体）要素
  - 各方向にNX・NY・NZ個の節点

プログラムの概要

- 三次元弾性問題
  - 3×3ブロック処理
- 前処理無しCG法
- Flat MPI，Hybrid

- 並列分散メッシュをプログラム内で自動生成
  - 予めメッシュ生成，領域分割等の必要ナシ

ファイルコピー on FX10

Fortranのみ

```
>$ cd <$O-TOP>
>$ cp /home/ss/aics60/2014SpringSchool/GeoFEM.tar .
>$ tar xvf GeoFEM.tar

>$ ls
     flat hybrid
>$ ls GeoFEM/flat
     run  src
>$ ls GeoFEM/hybrid
     run  src  src2
>$ cd GeoFEM/flat/src
>$ make

>$ cd ../../hybrid/src
>$ make
>$ cd ../src2
>$ make
```

実行方法 on FX10

Fortranのみ

Flat MPI

```
>$ cd <$O-TOP>/GeoFEM/flat/run
>$ <modify "go.sh", "mesh.inp">
>$ pjsub go.sh
```

Hybrid

```
>$ cd <$O-TOP>/GeoFEM/hybrid/run
>$ <modify "go.sh", "mesh.inp">
>$ pjsub go.sh
```

Hybrid（マトリクス生成部スレッド並列化）

```
>$ cd <$O-TOP>/GeoFEM/hybrid/run
>$ <modify "go2.sh", "mesh.inp">
>$ pjsub go2.sh
```

“mesh.inp”の中身：Flat MPI

(値)	(変数名)	(変数内容)
80 80 80	npx,npy,npz	p.2のNx，Ny，Nz
4 2 2	ndx,ndy,ndz	X，Y，Z軸方向の分割数
1 1	PEsmpTOT,(unused)	各MPIプロセスにおけるスレッド数(=1)，未使用（1を入れる）
200000	ITERmax	CG法の反復回数

- npx,npy,npzはndx,ndy,ndzで割り切れる必要あり
- ndx×ndy×ndzが総MPIプロセス数
  - 上記の場合は1ノード，16コア，16プロセス

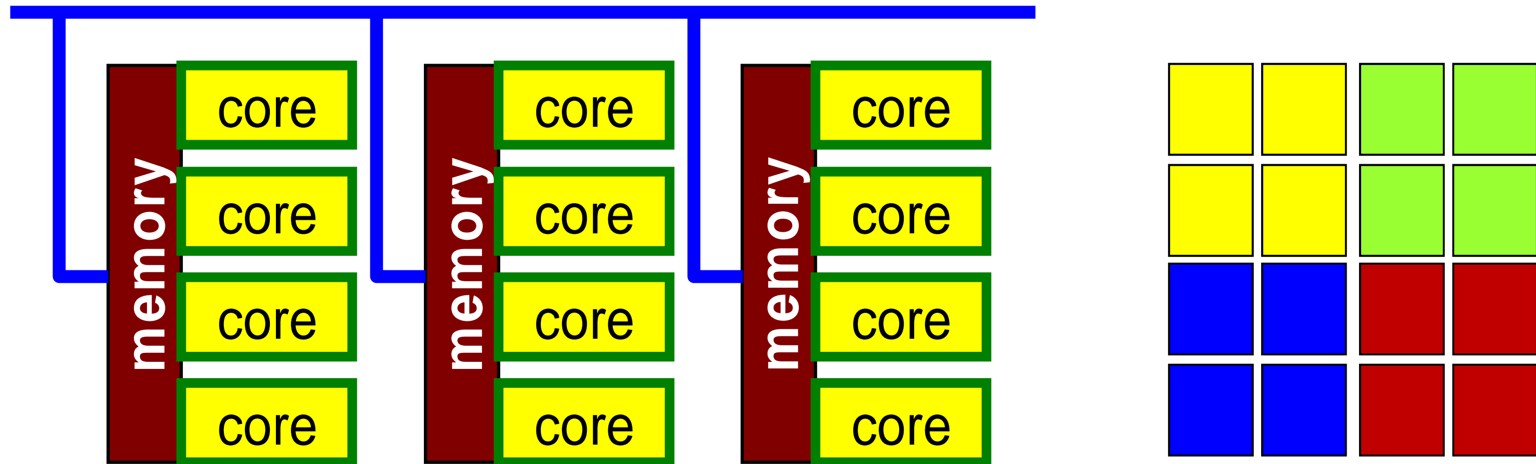
“mesh.inp”の中身：Hybrid

(値)	(変数名)	(変数内容)
80 80 80	npx,npy,npz	p.2のNx，Ny，Nz
1 1 1	ndx,ndy,ndz	X，Y，Z軸方向の分割数
16 1	PEsmpTOT,Ftflag	各MPIプロセスにおけるスレッド数(=1)，First Touch (=0:無し，=1:有り)
200000	ITERmax	CG法の反復回数

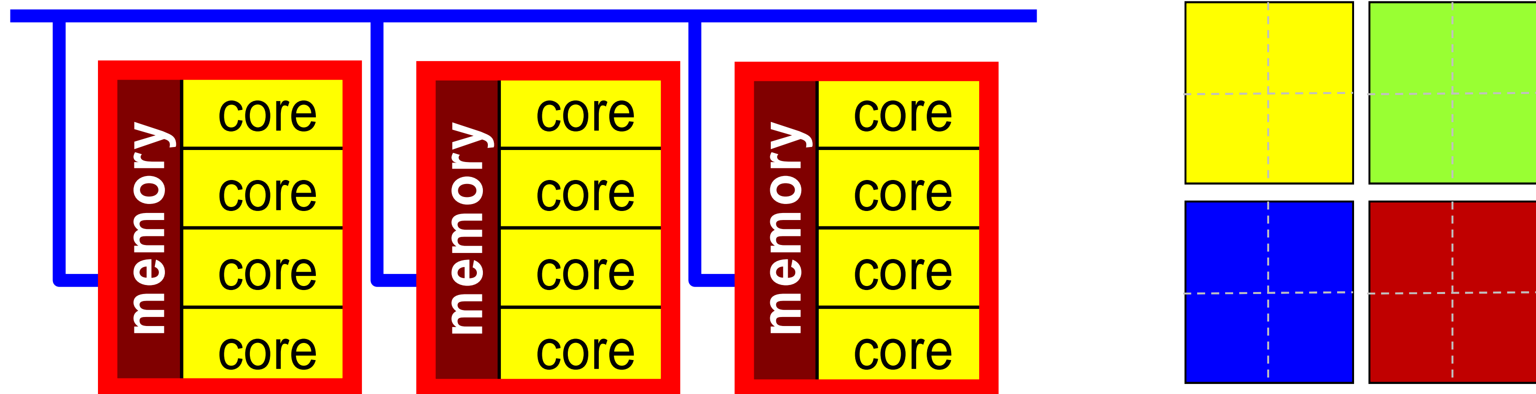
- npx,npy,npzはndx,ndy,ndzで割り切れる必要あり
- ndx×ndy×ndzが総MPIプロセス数
  - 上記の場合は1ノード，16コア，1プロセス
- First Touchの有無はFX10では関係ナシ
  - Multi Socket，NUMAだと関係ある
    - 詳細はサマースクール資料

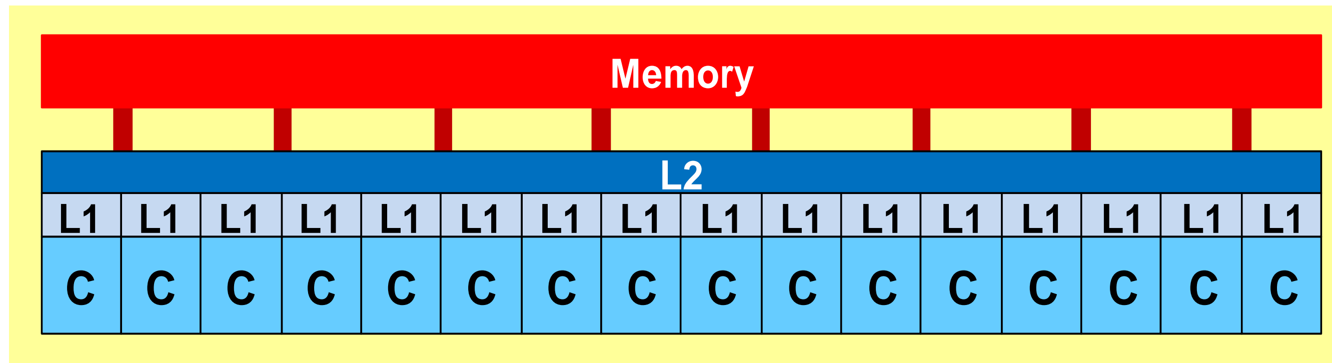
Flat MPI vs. Hybrid

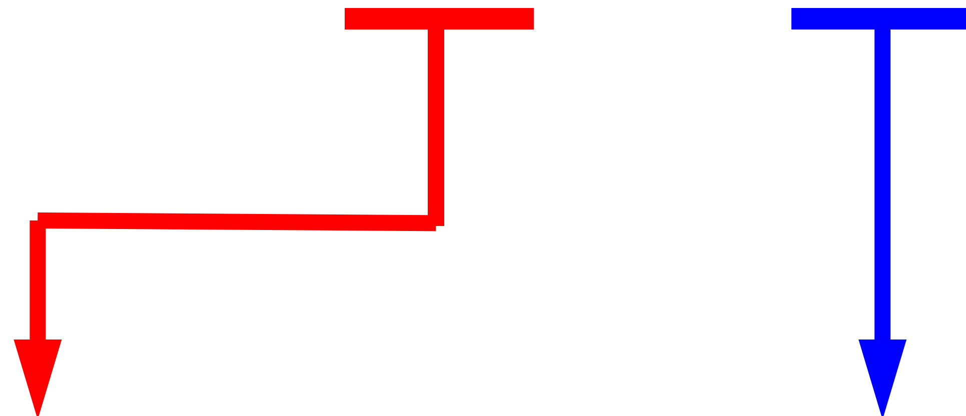
Flat-MPI : Each Core -> Independent



Hybrid : Hierarchal Structure







Number of OpenMP threads per a single MPI process

Number of MPI process per a single node

go.sh, go2.sh

Flat MPI

```
#PJM -L "node=1"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=small"
#PJM -o "test.lst"
#PJM --mpi "proc=16"

mpiexec ./sol

rm wk.*
```

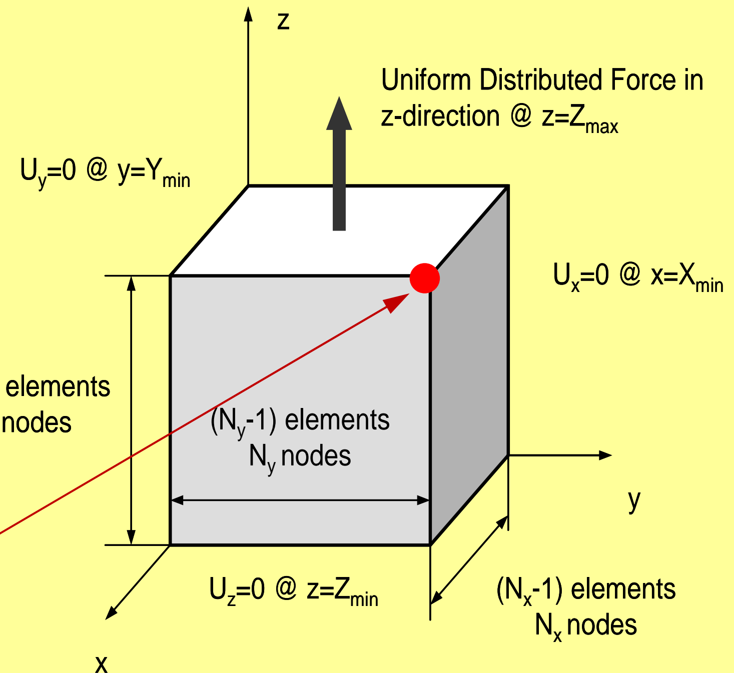
Hybrid

```
#!/bin/sh
#PJM -L "node=1"
#PJM -L "elapse=00:05:00"
#PJM -j
#PJM -L "rscgrp=small"
#PJM -o "test.lst"
#PJM --mpi "proc=1"

export OMP_NUM_THREADS=16
mpiexec ./sol

rm wk.*
```

全体処理

```
program SOLVER33_TEST_SMP

use solver33
use hpcmw_all
implicit REAL*8(A-H,O-Z)
integer, dimension(:), allocatable :: OLDtoNEWpe

call HPCMW_INIT
call INPUT_CNTL

allocate (OLDtoNEWpe(PETOT))

call INPUT_GRID (OLDtoNEWpe, ITERkk)
call MAT_CON0 (ITERkk)
call MAT_CON1

S1_time= MPI_WTIME()
call MAT_ASS_MAIN
E1_time= MPI_WTIME()
call MAT_ASS_BC
E2_time= MPI_WTIME()

t1= E1_time - S1_time
t2= E2_time - E1_time
if (my_rank.eq.0) write (*,'(2(1pe16.6))') t1, t2

call SOLVE33 (hpcmwIarray, hpcmwRarray, ITERkk)

if (my_rank.eq.PETOT-1) then
  i= N
  write (*,'(i8,3(1pe16.6))') i,X(3*i-2),X(3*i-1),X(3*i)
endif

call HPCMW_FINALIZE

end program SOLVER33_TEST_SMP
```

この点の変位



計算結果（Flat MPI）

```
        80        80        80
         4         2         2
         1
### NORMAL

color number:          1

    6.311270E-01      1.540459E-02

    1     1.001078E+00
  101     4.844414E-01
  201     1.161330E-01
  301     2.332553E-02
  401     3.149446E-03
  501     2.317633E-04
  601     1.145272E-05
  701     1.245789E-06
  801     2.954170E-08
  833     9.582782E-09

elapsed      833     1.892079E-02
   32000   -2.370E+01   -2.370E+01    7.900E+01

jwe0002i stop  * normal termination
```

mesh.inpのエコー

処理時間
（mat_ass_main,
mat_ass_bc）

反復回数（ε=10^{-9}），
1反復あたり計算時間
●点の3方向変位

計算結果（Hybrid）

```
        80         80         80
         1          1          1
        16
         1
### NORMAL

color number:         0

    6.585308E+00     1.859461E-01

  833     9.589950E-09
elapsed      833     1.825187E-02

  512000    -2.370E+01    -2.370E+01     7.900E+01

jwe0002i stop  * normal termination
```

mesh.inpのエコー

処理時間（mat_ass_main, mat_ass_bc）

反復回数（ε=10^{-9}），1反復あたり計算時間

●点の3方向変位

三次元弾性問題：1節点3成分

$$\frac{\partial \sigma_x}{\partial x} + \frac{\partial \tau_{xy}}{\partial y} + \frac{\partial \tau_{zx}}{\partial z} + X = 0$$

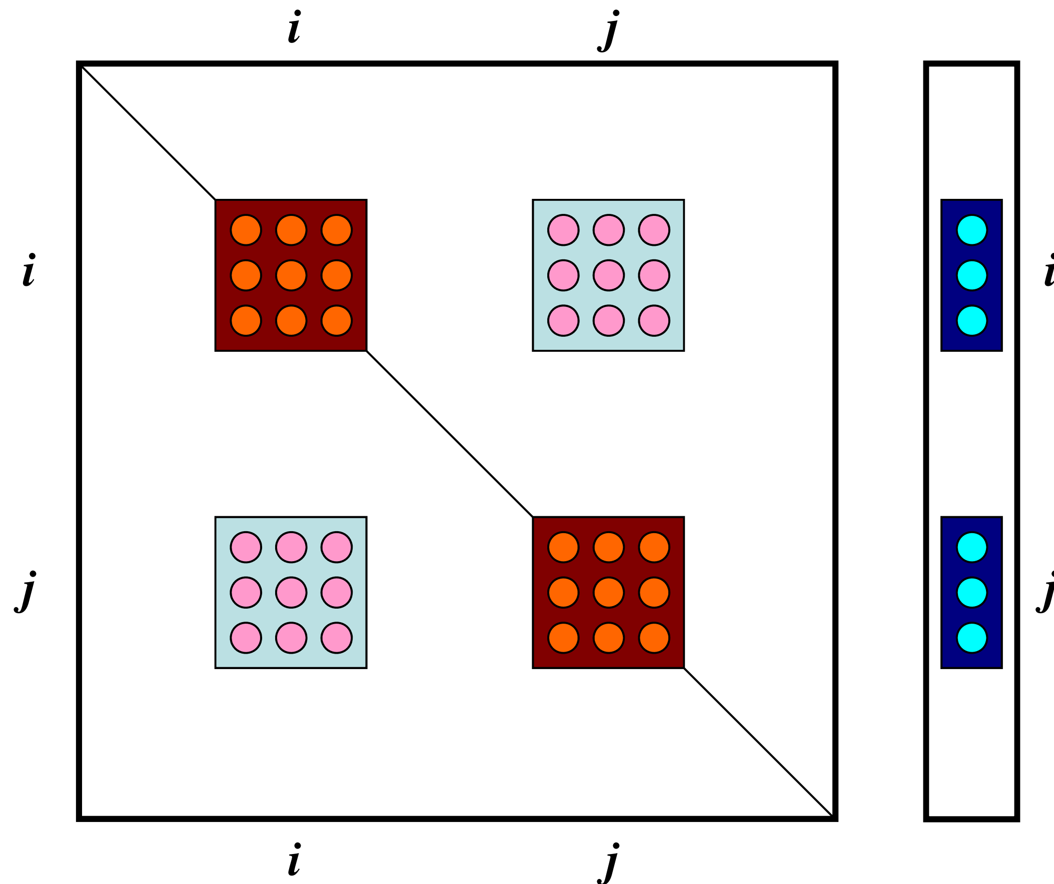
$$\frac{\partial \tau_{xy}}{\partial x} + \frac{\partial \sigma_y}{\partial y} + \frac{\partial \tau_{yz}}{\partial z} + Y = 0$$

$$\frac{\partial \tau_{zx}}{\partial x} + \frac{\partial \tau_{yz}}{\partial y} + \frac{\partial \sigma_z}{\partial z} + Z = 0$$

- ブロックとして記憶
  - ベクトル：1節点3成分（3方向変位成分）
  - 行列：各ブロック9成分
  - 行列の各成分ではなく，節点上の3変数に基づくブロックとして処理する

ブロックとして記憶（1/3）

- 記憶容量が減る
  - index, itemに関する記憶容量を数十分の1に削減できる



ブロックとして記憶（2/3）

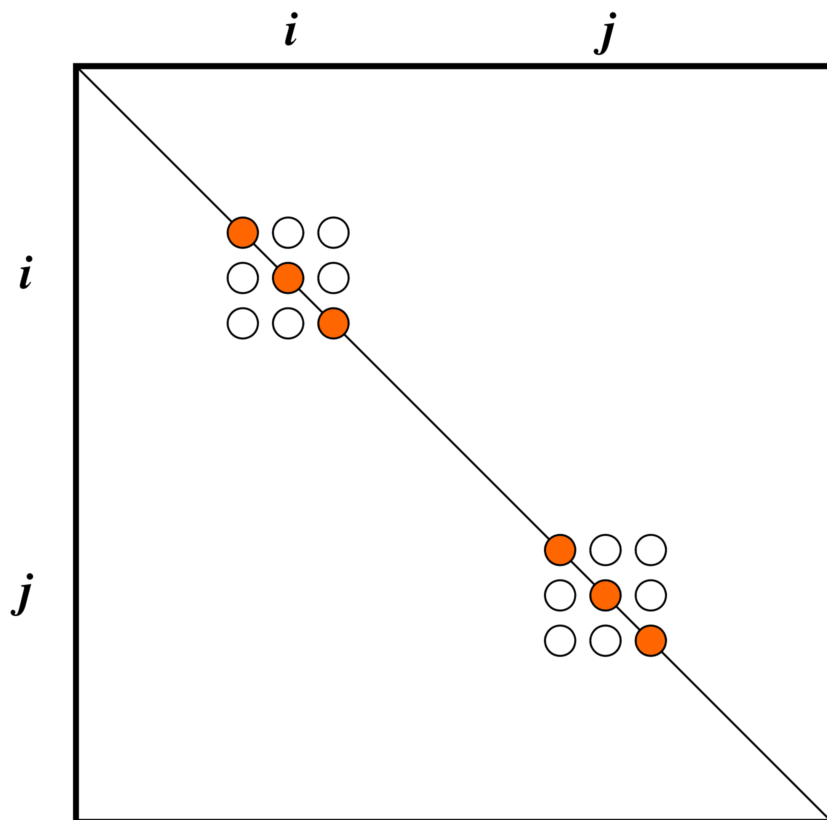
- 計算効率
  - 間接参照（メモリに負担）と計算の比が小さくなる
  - ベクトル，スカラー共に効く：2倍以上の性能

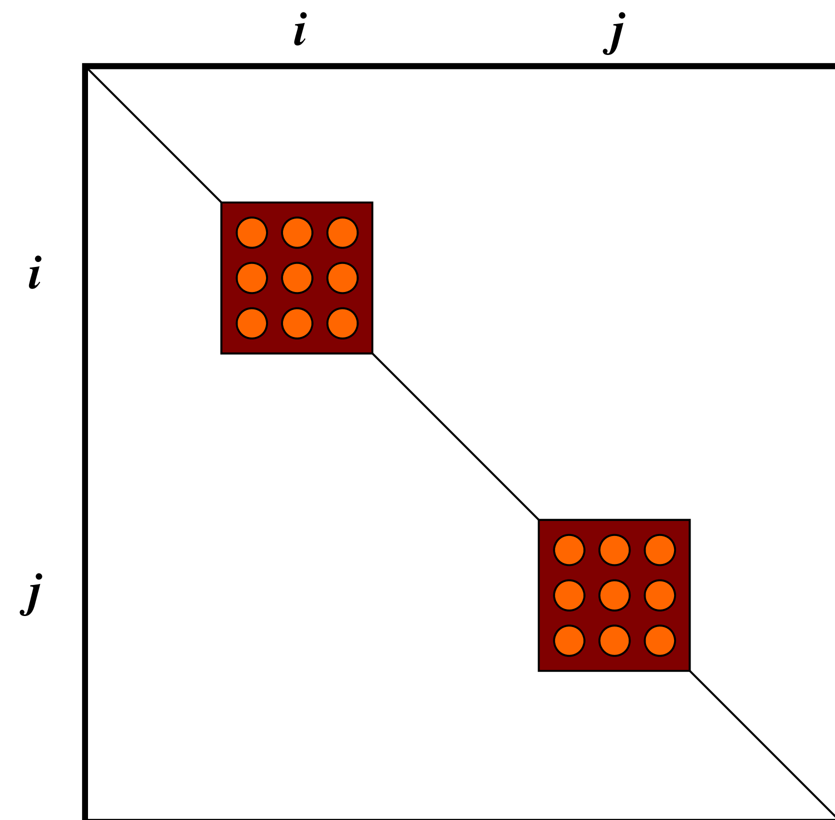
```
do i= 1, 3*N
  Y(i)= D(i)*X(i)
  do k= index(i-1)+1, index(i)
    kk= item(k)
    Y(i)= Y(i) + AMAT(k)*X(k)
  enddo
enddo
```

```
do i= 1, N
  X1= X(3*i-2)
  X2= X(3*i-1)
  X3= X(3*i)
  Y(3*i-2)= D(9*i-8)*X1+D(9*i-7)*X2+D(9*i-6)*X3
  Y(3*i-1)= D(9*i-5)*X1+D(9*i-4)*X2+D(9*i-3)*X3
  Y(3*I  )= D(9*i-2)*X1+D(9*i-1)*X2+D(9*I  )*X3
  do k= index(i-1)+1, index(i)
    kk= item(k)
    X1= X(3*kk-2)
    X2= X(3*kk-1)
    X3= X(3*kk)
    Y(3*i-2)= Y(3*i-2)+AMAT(9*k-8)*X1+AMAT(9*k-7)*X2 &
                                 +AMAT(9*k-6)*X3
    Y(3*i-1)= Y(3*i-1)+AMAT(9*k-5)*X1+AMAT(9*k-4)*X2 &
                                 +AMAT(9*k-3)*X3
    Y(3*I  )= Y(3*I  )+AMAT(9*k-2)*X1+AMAT(9*k-1)*X2 &
                                 +AMAT(9*k  )*X3
  enddo
enddo
```

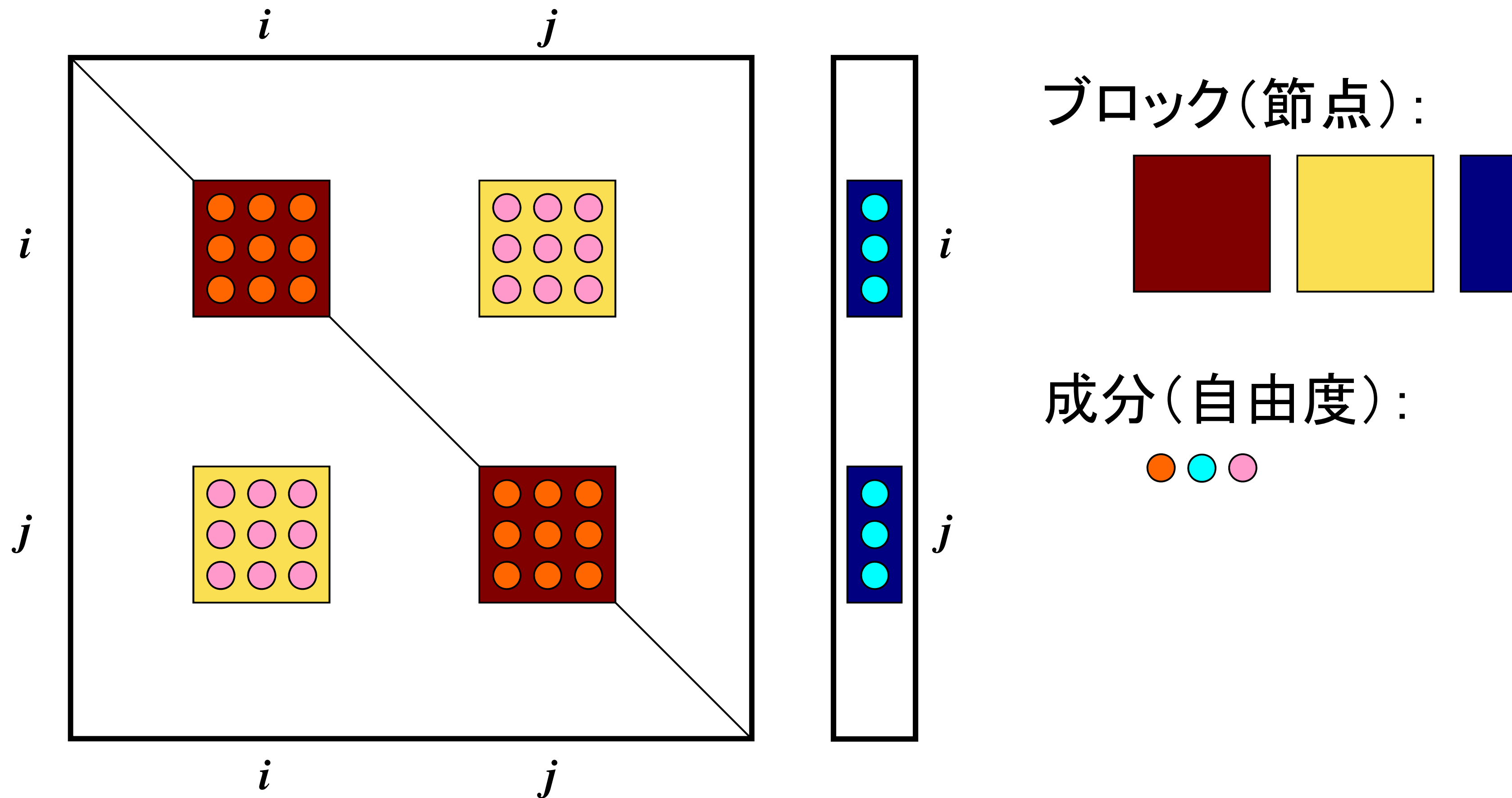
ブロックとして記憶（3/3）

- 計算の安定化
  - 対角成分で割るのではなく，対角ブロックの完全LU分解を求めて解く
  - 特に悪条件問題で有効：本問は簡単なので前処理ナシ





用語の定義



DAXPY in CG

```
!C
!C +----------------------+
!C | {x}= {x} + ALPHA*{p} |
!C | {r}= {r} - ALPHA*{q} |
!C +----------------------+
!C===
      do i= 1, N
         X(3*i-2)  = X (3*i-2)   + ALPHA * WW(3*i-2,P)
         X(3*i-1)  = X (3*i-1)   + ALPHA * WW(3*i-1,P)
         X(3*i  )  = X (3*i  )   + ALPHA * WW(3*i  ,P)
        WW(3*i-2,R)= WW(3*i-2,R) - ALPHA * WW(3*i-2,Q)
        WW(3*i-1,R)= WW(3*i-1,R) - ALPHA * WW(3*i-1,Q)
        WW(3*i  ,R)= WW(3*i  ,R) - ALPHA * WW(3*i  ,Q)
      enddo
```

```
!$omp parallel do private(iS,iE,i)
!$omp&            shared (ALPHA)
      do ip= 1, PEsmpTOT
        iS= STACKmcG(ip-1) + 1
        iE= STACKmcG(ip  )
      do i= iS, iE
         X(3*i-2)  = X (3*i-2)   + ALPHA * WW(3*i-2,P)
         X(3*i-1)  = X (3*i-1)   + ALPHA * WW(3*i-1,P)
         X(3*i  )  = X (3*i  )   + ALPHA * WW(3*i  ,P)
        WW(3*i-2,R)= WW(3*i-2,R) - ALPHA * WW(3*i-2,Q)
        WW(3*i-1,R)= WW(3*i-1,R) - ALPHA * WW(3*i-1,Q)
        WW(3*i  ,R)= WW(3*i  ,R) - ALPHA * WW(3*i  ,Q)
      enddo
      enddo
```

STACKmcGはナシでもOK

```
!$omp parallel do private(i) shared (ALPHA)
      do i= 1, N
         X(3*i-2)   = X (3*i-2)    + ALPHA * WW(3*i-2,P)
         X(3*i-1)   = X (3*i-1)    + ALPHA * WW(3*i-1,P)
         X(3*i  )   = X (3*i  )    + ALPHA * WW(3*i  ,P)
        WW(3*i-2,R)= WW(3*i-2,R) - ALPHA * WW(3*i-2,Q)
        WW(3*i-1,R)= WW(3*i-1,R) - ALPHA * WW(3*i-1,Q)
        WW(3*i  ,R)= WW(3*i  ,R) - ALPHA * WW(3*i  ,Q)
      enddo
```

```
      allocate (STACKmcG(0:PEsmpTOT)); STACKmcG= 0
      icon= N/PEsmpTOT; ir= N - icon*PEsmpTOT
      do ip= 1, PEsmpTOT
        STACKmcG(ip)= icon
      enddo
      do ip= 1, ir
        STACKmcG(ip)= icon + 1
      enddo
      do ip= 1, PEsmpTOT
        STACKmcG(ip)= STACKmcG(ip-1) + STACKmcG(ip)
      enddo
```

```
!$omp parallel do private(iS,iE,i)
!$omp&            shared (ALPHA)
      do ip= 1, PEsmpTOT
        iS= STACKmcG(ip-1) + 1
        iE= STACKmcG(ip  )
      do i= iS, iE
         X(3*i-2)   = X (3*i-2)    + ALPHA * WW(3*i-2,P)
         X(3*i-1)   = X (3*i-1)    + ALPHA * WW(3*i-1,P)
         X(3*i  )   = X (3*i  )    + ALPHA * WW(3*i  ,P)
        WW(3*i-2,R)= WW(3*i-2,R) - ALPHA * WW(3*i-2,Q)
        WW(3*i-1,R)= WW(3*i-1,R) - ALPHA * WW(3*i-1,Q)
        WW(3*i  ,R)= WW(3*i  ,R) - ALPHA * WW(3*i  ,Q)
      enddo
      enddo
```

内積

```
      DNRM20= 0.d0

      do i= 1, N
        DNRM20= DNRM20 + WW(3*i-2,R)**2 + WW(3*i-1,R)**2                &
     &                                  + WW(3*i  ,R)**2
      enddo

      call MPI_Allreduce (DNRM20, DNRM2, 1, MPI_DOUBLE_PRECISION,       &
     &                    MPI_SUM, SOLVER_COMM, ierr)
```

```
      DNRM20= 0.d0
!$omp parallel do private(iS,iE,i)
!$omp&            reduction(+:DNRM20)
      do ip= 1, PEsmpTOT
        iS= STACKmcG(ip-1) + 1
        iE= STACKmcG(ip  )
      do i= iS, iE
        DNRM20= DNRM20 + WW(3*i-2,R)**2 + WW(3*i-1,R)**2                &
     &                                  + WW(3*i  ,R)**2
      enddo
      enddo

      call MPI_Allreduce (DNRM20, DNRM2, 1, MPI_DOUBLE_PRECISION,       &
     &                    MPI_SUM, SOLVER_COMM, ierr)
```

行列ベクトル積（Flat MPI）

```
 call SOLVER_SEND_RECV_3                                                 &
&   ( N, NP, NEIBPETOT, NEIBPE, STACK_IMPORT, NOD_IMPORT,               &
&     STACK_EXPORT, NOD_EXPORT, WS, WR, WW(1,P) , SOLVER_COMM, my_rank)
 do j= 1, N
      X1= WW(3*j-2,P)
      X2= WW(3*j-1,P)
      X3= WW(3*j  ,P)
   WVAL1= D(9*j-8)*X1 + D(9*j-7)*X2 + D(9*j-6)*X3
   WVAL2= D(9*j-5)*X1 + D(9*j-4)*X2 + D(9*j-3)*X3
   WVAL3= D(9*j-2)*X1 + D(9*j-1)*X2 + D(9*j  )*X3
   do k= INL(j-1)+1, INL(j)
      i= IAL(k)
     X1= WW(3*i-2,P)
     X2= WW(3*i-1,P)
     X3= WW(3*i  ,P)
     WVAL1= WVAL1 + AL(9*k-8)*X1 + AL(9*k-7)*X2 + AL(9*k-6)*X3
     WVAL2= WVAL2 + AL(9*k-5)*X1 + AL(9*k-4)*X2 + AL(9*k-3)*X3
     WVAL3= WVAL3 + AL(9*k-2)*X1 + AL(9*k-1)*X2 + AL(9*k  )*X3
   enddo
   do k= INU(j-1)+1, INU(j)
      i= IAU(k)
     X1= WW(3*i-2,P)
     X2= WW(3*i-1,P)
     X3= WW(3*i  ,P)
     WVAL1= WVAL1 + AU(9*k-8)*X1 + AU(9*k-7)*X2 + AU(9*k-6)*X3
     WVAL2= WVAL2 + AU(9*k-5)*X1 + AU(9*k-4)*X2 + AU(9*k-3)*X3
     WVAL3= WVAL3 + AU(9*k-2)*X1 + AU(9*k-1)*X2 + AU(9*k  )*X3
   enddo
   WW(3*j-2,Q)= WVAL1
   WW(3*j-1,Q)= WVAL2
   WW(3*j  ,Q)= WVAL3
 enddo
```

行列ベクトル積（Hybrid）

```
      call SOLVER_SEND_RECV_3                                              &
     &   ( N, NP, NEIBPETOT, NEIBPE, STACK_IMPORT, NOD_IMPORT,             &
     &     STACK_EXPORT, NOD_EXPORT, WS, WR, WW(1,P) , SOLVER_COMM, my_rank)
!$omp parallel do private (ip,jS,jE,j,k,i,X1,X2,X3,WVAL1,WVAL2,WVAL3)
      do ip= 1,PEsmpTOT
        jS= STACKmcG(ip-1) + 1; jE= STACKmcG(ip  )
      do j= jS, jE
           X1= WW(3*j-2,P)
           X2= WW(3*j-1,P)
           X3= WW(3*j  ,P)
        WVAL1= D(9*j-8)*X1 + D(9*j-7)*X2 + D(9*j-6)*X3
        WVAL2= D(9*j-5)*X1 + D(9*j-4)*X2 + D(9*j-3)*X3
        WVAL3= D(9*j-2)*X1 + D(9*j-1)*X2 + D(9*j  )*X3
        do k= INL(j-1)+1, INL(j)
            i= IAL(k)
          X1= WW(3*i-2,P)
          X2= WW(3*i-1,P)
          X3= WW(3*i  ,P)
          WVAL1= WVAL1 + AL(9*k-8)*X1 + AL(9*k-7)*X2 + AL(9*k-6)*X3
          WVAL2= WVAL2 + AL(9*k-5)*X1 + AL(9*k-4)*X2 + AL(9*k-3)*X3
          WVAL3= WVAL3 + AL(9*k-2)*X1 + AL(9*k-1)*X2 + AL(9*k  )*X3
        enddo
        do k= INU(j-1)+1, INU(j)
            i= IAU(k)
          X1= WW(3*i-2,P)
          X2= WW(3*i-1,P)
          X3= WW(3*i  ,P)
          WVAL1= WVAL1 + AU(9*k-8)*X1 + AU(9*k-7)*X2 + AU(9*k-6)*X3
          WVAL2= WVAL2 + AU(9*k-5)*X1 + AU(9*k-4)*X2 + AU(9*k-3)*X3
          WVAL3= WVAL3 + AU(9*k-2)*X1 + AU(9*k-1)*X2 + AU(9*k  )*X3
        enddo
        WW(3*j-2,Q)= WVAL1
        WW(3*j-1,Q)= WVAL2
        WW(3*j  ,Q)= WVAL3
      enddo
      enddo
```

SEND

```
      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
        do k= istart+1, istart+inum
               ii   = 3*NOD_EXPORT(k)
           WS(3*k-2)= X(ii-2)
           WS(3*k-1)= X(ii-1)
           WS(3*k  )= X(ii  )
        enddo

        call MPI_ISEND (WS(3*istart+1), 3*inum,MPI_DOUBLE_PRECISION,     &
     &                  NEIBPE(neib), 0, SOLVER_COMM, req1(neib), ierr)
      enddo
```

```
      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
!$omp parallel do private (ii)
        do k= istart+1, istart+inum
               ii   = 3*NOD_EXPORT(k)
           WS(3*k-2)= X(ii-2)
           WS(3*k-1)= X(ii-1)
           WS(3*k  )= X(ii  )
        enddo

        call MPI_ISEND (WS(3*istart+1), 3*inum,MPI_DOUBLE_PRECISION,     &
     &                  NEIBPE(neib), 0, SOLVER_COMM, req1(neib), ierr)
      enddo
```

SEND/RECV (Original)

```
!C
!C-- INIT.
      allocate (sta1(MPI_STATUS_SIZE,NEIBPETOT), sta2(MPI_STATUS_SIZE,NEIBPETOT))
      allocate (req1(NEIBPETOT), req2(NEIBPETOT))

!C
!C-- SEND
      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
        do k= istart+1, istart+inum
          WS(k)= X(NOD_EXPORT(k))
        enddo
        call MPI_ISEND (WS(istart+1), inum, MPI_DOUBLE_PRECISION,           &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, req1(neib), ierr)
      enddo

!C
!C-- RECEIVE
      do neib= 1, NEIBPETOT
        istart= STACK_IMPORT(neib-1)
        inum  = STACK_IMPORT(neib  ) - istart
        call MPI_IRECV (WR(istart+1), inum, MPI_DOUBLE_PRECISION,           &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, req2(neib), ierr)
      enddo
      call MPI_WAITALL (NEIBPETOT, req2, sta2, ierr)

      do neib= 1, NEIBPETOT
        istart= STACK_IMPORT(neib-1)
        inum  = STACK_IMPORT(neib  ) - istart
        do k= istart+1, istart+inum
          X(NOD_IMPORT(k))= WR(k)
        enddo
      enddo
      call MPI_WAITALL (NEIBPETOT, req1, sta1, ierr)
```

If numbering of external nodes is continuous in each neighboring process ...

	84	81	85	82	83	86	88	87	
96	57	58	59	60	61	62	63	64	73
95	49	50	51	52	53	54	55	56	74
94	41	42	43	44	45	46	47	48	80
93	33	34	35	36	37	38	39	40	79
92	25	26	27	28	29	30	31	32	78
91	17	18	19	20	21	22	23	24	77
90	9	10	11	12	13	14	15	16	76
89	1	2	3	4	5	6	7	8	75
	65	66	67	68	69	70	71	72	

SEND/RECV (NEW:1)

```
!C
!C-- INIT.
        allocate (sta1(MPI_STATUS_SIZE,2*NEIBPETOT))
        allocate (req1(2*NEIBPETOT))

!C
!C-- SEND
      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
        do k= istart+1, istart+inum
          WS(k)= X(NOD_EXPORT(k))
        enddo
      enddo

      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
        call MPI_ISEND (WS(istart+1), inum, MPI_DOUBLE_PRECISION,          &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, req1(neib), ierr)
      enddo

!C
!C-- RECEIVE
      do neib= 1, NEIBPETOT
        inum  = STACK_IMPORT(neib) - STACK_IMPORT(neib-1)
        istart= NOD_IMPORT(STACK_IMPORT(neib-1)+1)

        call MPI_IRECV (X(istart), inum, MPI_DOUBLE_PRECISION,             &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, req1(NEIBPETOT+neib), ierr)
      enddo

      call MPI_WAITALL (2*NEIBPETOT, req1, sta1, ierr)
```

SEND/RECV (NEW:2), N0: int. node

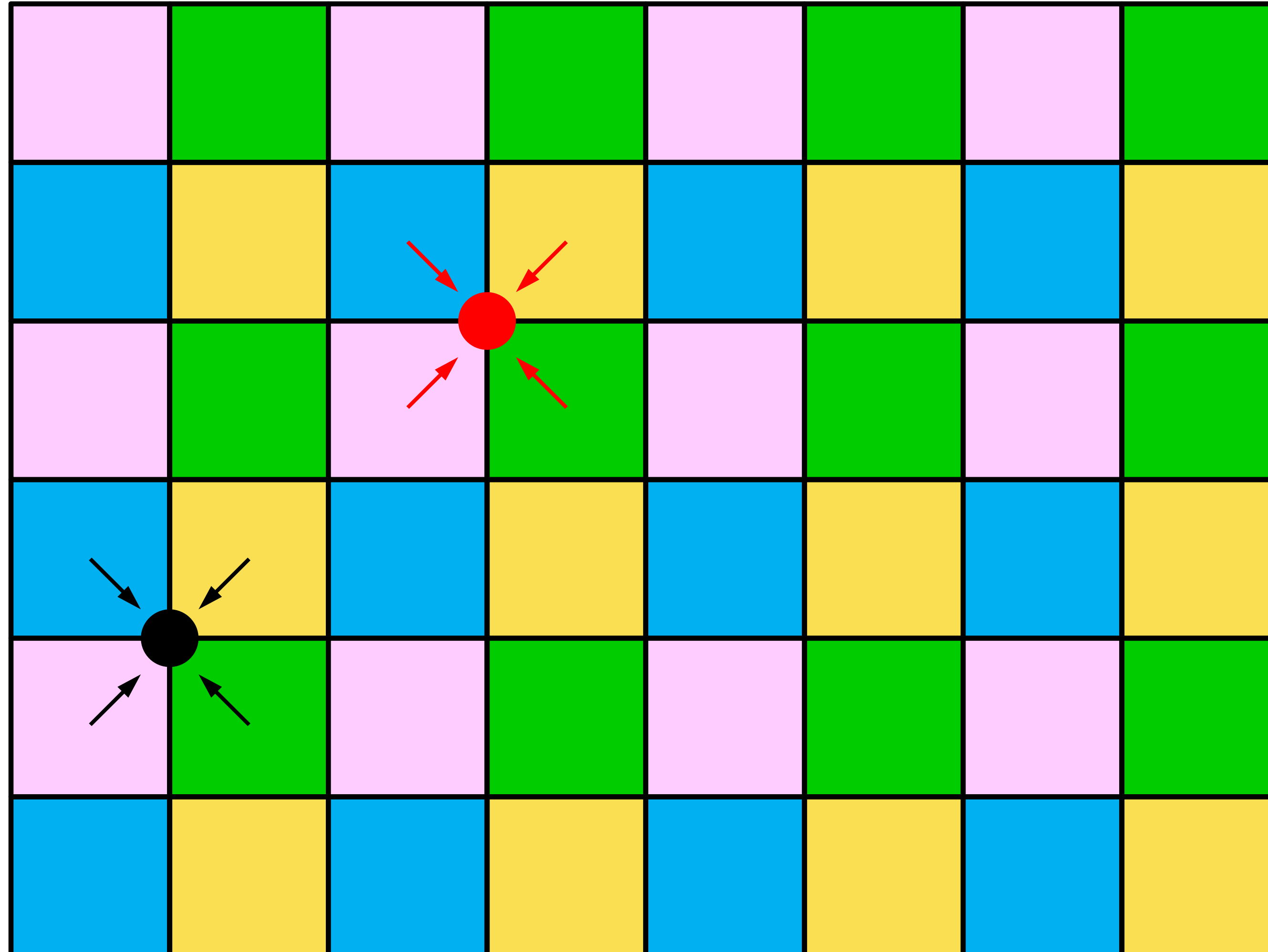
```
!C
!C-- INIT.
        allocate (sta1(MPI_STATUS_SIZE,2*NEIBPETOT))
        allocate (req1(2*NEIBPETOT))

!C
!C-- SEND
      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
        do k= istart+1, istart+inum
          WS(k)= X(NOD_EXPORT(k))
        enddo
      enddo

      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
        call MPI_ISEND (WS(istart+1), inum, MPI_DOUBLE_PRECISION,              &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, req1(neib), ierr)
      enddo

!C
!C-- RECEIVE
      do neib= 1, NEIBPETOT
        inum  = STACK_IMPORT(neib) - STACK_IMPORT(neib-1)
        istart= STACK_IMPORT(neib-1) + N0 + 1

        call MPI_IRECV (X(istart), inum, MPI_DOUBLE_PRECISION,                 &
     &                  NEIBPE(neib), 0, MPI_COMM_WORLD, req1(NEIBPETOT+neib), ierr)
      enddo

      call MPI_WAITALL (2*NEIBPETOT, req1, sta1, ierr)
```

スレッド並列化

- CG法
  - ほぼOpenMPの指示文（directive）を入れるだけで済む
  - 前処理がILU系になるとそう簡単ではない（Summer School資料参照）
- 行列生成部（mat_ass_main, mat_ass_bc）
  - 複数要素から同時に同じ節点に足し込むことを回避する必要がある
    - 計算結果が変わってしまう
    - 同時に書き込もうとして計算が止まってしまう場合もある（環境依存）
  - 色分け（Coloring）
    - 色内に属する要素が同じ節点を同時に更新しないように色分けすれば，同じ色内の要素の処理は並列にできる
    - 現在の問題は規則正しい形状なので，8色に塗り分けられる（1節点を共有する要素数は最大8，要素内節点数8）

行列生成部スレッド並列化

同じ色の要素の処理は並列に実行可能



要素色分け（1/2）

```
      allocate (ELMCOLORindex(0:NP))     各色に含まれる要素数（一次元圧縮配列）
      allocate (ELMCOLORitem (ICELTOT))  色の順番に並び替えた要素番号
      if (allocated (IWKX)) deallocate (IWKX)
      allocate (IWKX(NP,3))

      IWKX= 0
      icou= 0
      do icol= 1, NP
        do i= 1, NP
          IWKX(i,1)= 0
        enddo
        do icel= 1, ICELTOT
          if (IWKX(icel,2).eq.0) then
            in1= ICELNOD(icel,1)
            in2= ICELNOD(icel,2)
            in3= ICELNOD(icel,3)
            in4= ICELNOD(icel,4)
            in5= ICELNOD(icel,5)
            in6= ICELNOD(icel,6)
            in7= ICELNOD(icel,7)
            in8= ICELNOD(icel,8)

            ip1= IWKX(in1,1)
            ip2= IWKX(in2,1)
            ip3= IWKX(in3,1)
            ip4= IWKX(in4,1)
            ip5= IWKX(in5,1)
            ip6= IWKX(in6,1)
            ip7= IWKX(in7,1)
            ip8= IWKX(in8,1)
```

要素色分け（2/2）

```
          isum= ip1 + ip2 + ip3 + ip4 + ip5 + ip6 + ip7 + ip8
          if (isum.eq.0) then                 要素各節点が同色内でアクセスされていない
            icou= icou + 1                    カウンターを1つ増やす
            IWKX(icol,3)= icou                各色内に含まれる要素数の累積
            IWKX(icel,2)= icol
            ELMCOLORitem(icou)= icel          icou番目の要素をicelとする

            IWKX(in1,1)= 1                    各節点は同色内でアクセス不可, Flag立てる
            IWKX(in2,1)= 1
            IWKX(in3,1)= 1
            IWKX(in4,1)= 1
            IWKX(in5,1)= 1
            IWKX(in6,1)= 1
            IWKX(in7,1)= 1
            IWKX(in8,1)= 1
            if (icou.eq.ICELTOT) goto 100     全要素が色づけされたら終了
          endif
        endif
      enddo
    enddo

100 continue
    ELMCOLORtot= icol                         色数
    IWKX(0         ,3)= 0
    IWKX(ELMCOLORtot,3)= ICELTOT

    do icol= 0, ELMCOLORtot
      ELMCOLORindex(icol)= IWKX(icol,3)
    enddo

    write (*,'(a,2i8)') '### Number of Element Colors',
   &                    my_rank, ELMCOLORtot

    deallocate (IWKX)
```

スレッド並列化されたマトリクス生成部

```
      X1= 0.d0
      Y1= 0.d0
      Z1= 0.d0
....
      X8= 0.d0
      Y8= DY
      Z8= DZ

        call JACOBI (DETJ, PNQ, PNE, PNT, PNX, PNY, PNZ,                 &
     &              X1, X2, X3, X4, X5, X6, X7, X8,                      &
     &              Y1, Y2, Y3, Y4, Y5, Y6, Y7, Y8,                      &
     &              Z1, Z2, Z3, Z4, Z5, Z6, Z7, Z8 )
```

（全要素同じ形状なのでヤコビアンの計算部分はループの外に出してしまう）

```
      do icol= 1, ELMCOLORtot
!$omp parallel do private (icel0,icel,in1,in2,in3,in4,in5,in6,in7,in8)   &
!$omp&            private (nodLOCAL,ie,je,ip,jp,kk,iiS,iiE,iDlu,k)       &
!$omp&            private (PNXi,PNYi,PNZi,PNXj,PNYj,PNZj,a11,a12)        &
!$omp&            private (a13,a21,a22,a23,a31,a32,a33,ipn,jpn,kpn,coef)
      do icel0= ELMCOLORindex(icol-1)+1, ELMCOLORindex(icol)
        icel= ELMCOLORitem(icel0)
        in1= ICELNOD(icel,1)
        in2= ICELNOD(icel,2)
        in3= ICELNOD(icel,3)
        in4= ICELNOD(icel,4)
        in5= ICELNOD(icel,5)
        in6= ICELNOD(icel,6)
        in7= ICELNOD(icel,7)
        in8= ICELNOD(icel,8)
...
```

余談：First Touch Data Placement

“Patterns for Parallel Programming” Mattson, T.G. et al.

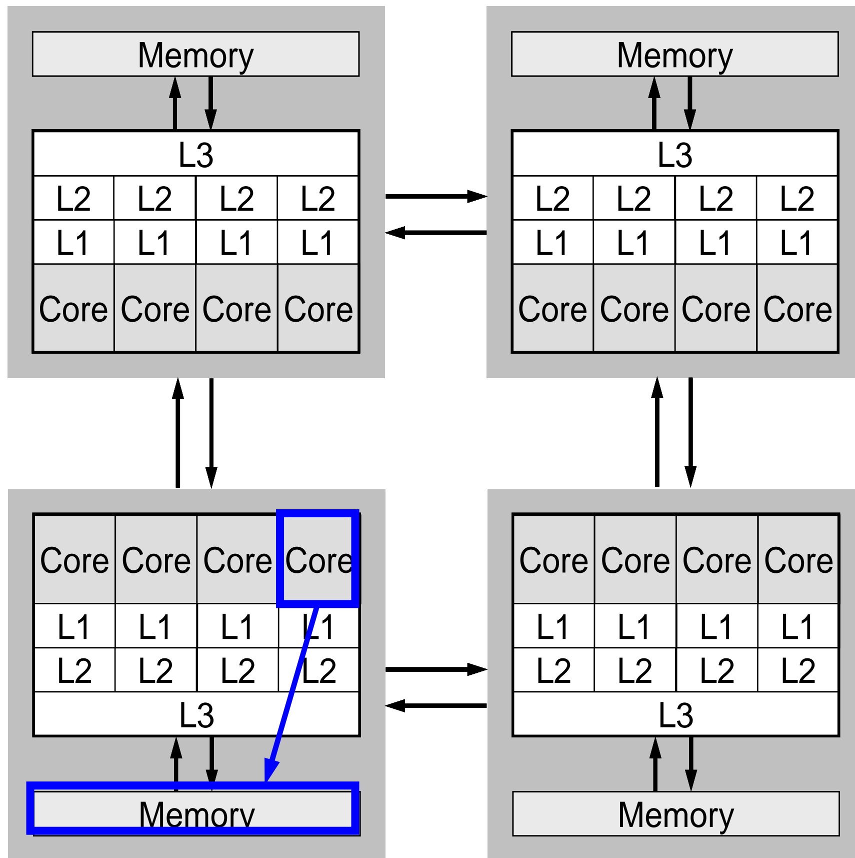
To reduce memory traffic in the system, it is important to keep the data close to the PEs that will work with the data (e.g. NUMA control).

On NUMA computers, this corresponds to making sure the pages of memory are allocated and “owned” by the PEs that will be working with the data contained in the page.

The most common NUMA page-placement algorithm is the “first touch” algorithm, in which the PE first referencing a region of memory will have the page holding that memory assigned to it.

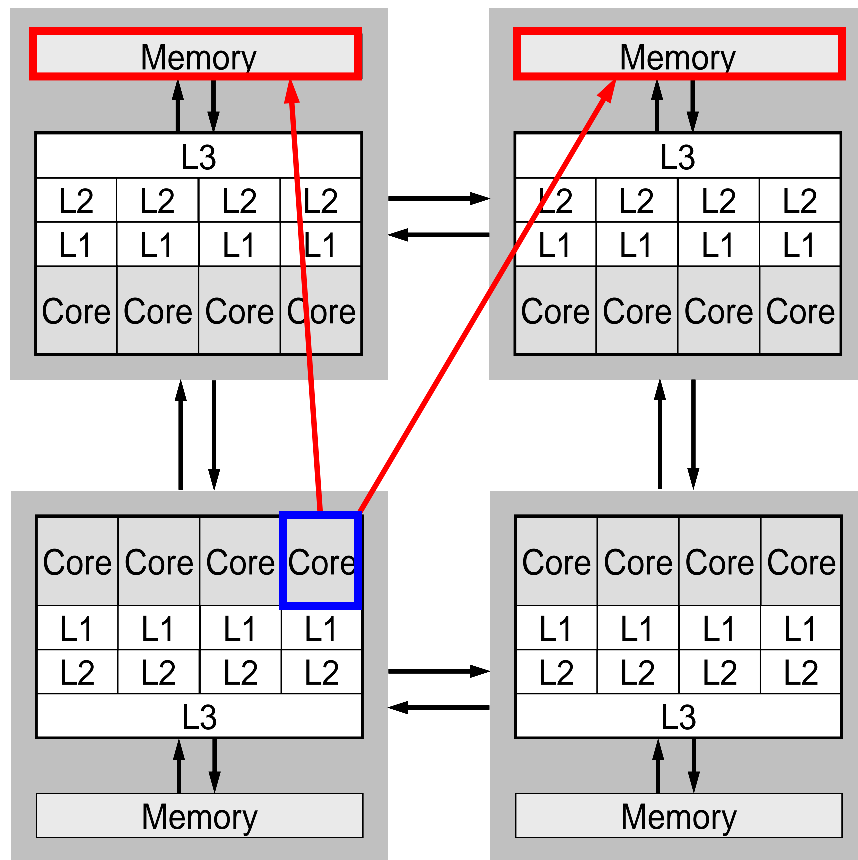
A very common technique in OpenMP program is to initialize data in parallel using the same loop schedule as will be used later in the computations.

NUMA アーキテクチャ



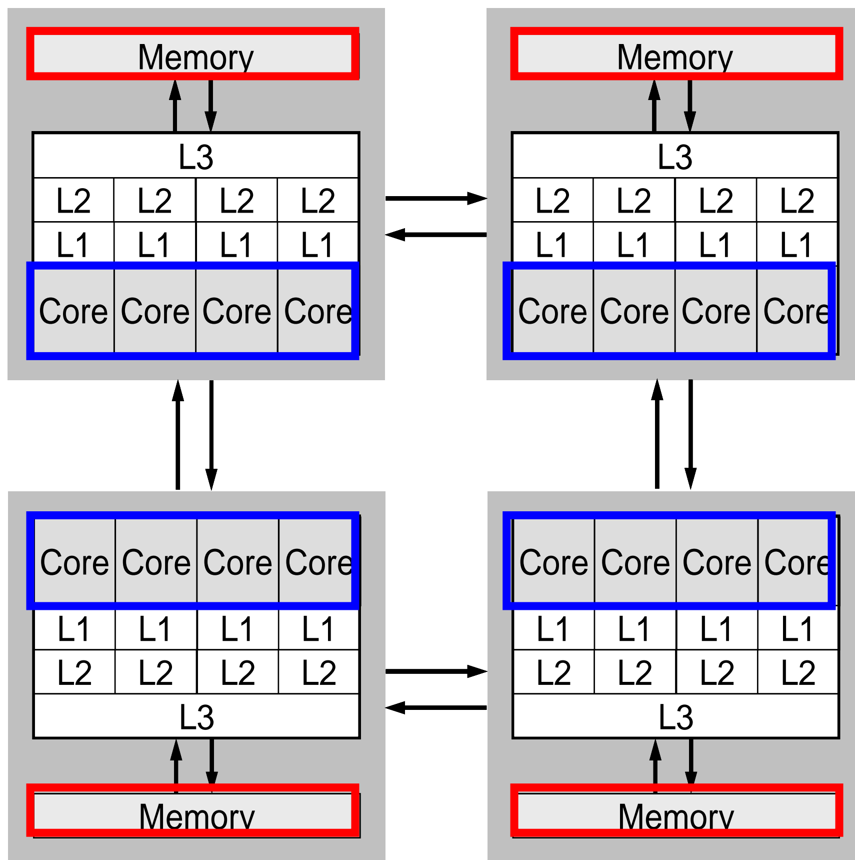
- コアで扱うデータはなるべくローカルなメモリ（コアの属するソケットのメモリ）上にあると効率が良い。

NUMA アーキテクチャ



- 異なるソケットにある場合はアクセスに時間がかかる。

NUMA アーキテクチャ



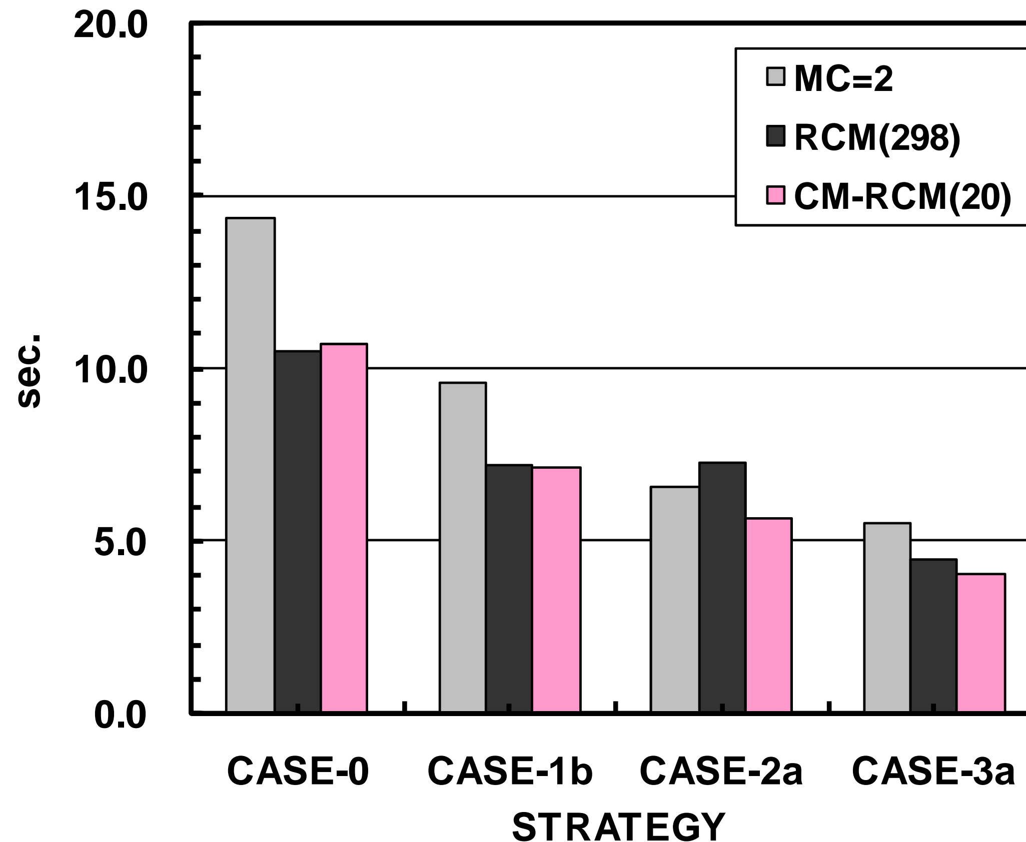
- First-touchによって，できるだけローカルなメモリ上にデータを持ってくる。
- NUMAアーキテクチャでは，ある変数を最初にアクセスしたコア（の属するソケット）のローカルメモリ上にその変数の記憶領域（ページファイル）が確保される。
  - 配列の初期化手順によって大幅な性能向上が期待できる。

First Touchの有無の例

```
      if (FTflag.eq.1) then
!$omp parallel do private(jS,jE,j,jsL,jeL,jsU,jeU,k)
        do ip= 1, PEsmpTOT
          jS= STACKmcG(ip-1) + 1
          jE= STACKmcG(ip  )
          do j = jS, jE
            jsL= indexL(j-1)+1
            jeL= indexL(j)
            do k= jsL, jeL
              AL(9*k-8)= 0.d0
              AL(9*k-7)= 0.d0
...
              AL(9*k-1)= 0.d0
              AL(9*k  )= 0.d0
            enddo

            jsU= indexU(j-1)+1
            jeU= indexU(j)
            do k= jsU, jeU
              AU(9*k-8)= 0.d0
              AU(9*k-7)= 0.d0
...
              AU(9*k-1)= 0.d0
              AU(9*k  )= 0.d0
            enddo
          enddo
        enddo
       else
        AL= 0.d0
        AU= 0.d0
      endif
```

First Touchの効果

T2K東大1ノード16コア，16スレッド
三次元ポアソン方程式計算時間（ICCG法）



Case-0：
初期

Case-1b：
NUMAコントロール（実行時）

Case-2a：
+ First Touch

Case-3a：
+ Sequential Reordering（これはNUMAでなくても有効，Summer School資料参照）