2015年4月24日（金）
第18回FrontISTR研究会

FrontISTRの並列計算の基礎

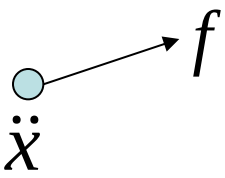
奥田洋司
okuda@k.u-tokyo.ac.jp
東京大学大学院・新領域創成科学研究科・人間環境学専攻

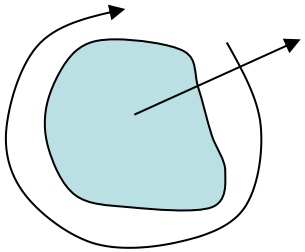
目次

- 導入
  - 計算力学とは
  - 連続体の力学、連立１次方程式
  - FEM構造解析の概要
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FEM計算におけるノード間並列
  - 領域分割とMPI
- FEM計算におけるノード内並列（単体性能）
  - ループ分割とOpenMP
- 実効性能について

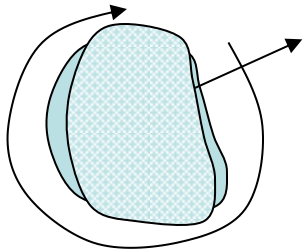
連続体の力学

- 運動の記述　　質点系→剛体系→連続体





大きさをもつが変形しない



変形しながら運動

- 保存則

  質量保存、運動量保存（並進と回転）、エネルギー保存

- 境界条件と初期条件
- 構成則

これら偏微分方程式により、固体/液体/気体の物理現象が記述される

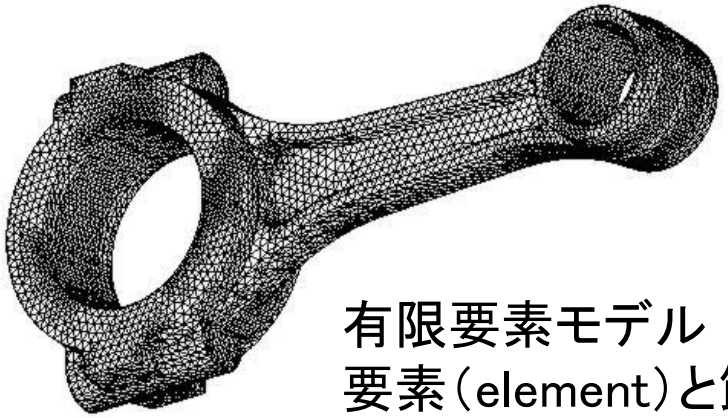
微分方程式をコンピュータで解くには？

微分方程式 → 連立1次方程式 → （各点での物理量が求まる）

（連続体）　（とびとびの点で表現）

$$[\ K\ ]\{\ u\ \} = \{\ f\ \}$$

CADモデル



有限要素モデル
要素（element）と節点（node）

節点に物理量が定義されている
（例）節点変位、節点温度

連立1次方程式を高速に解く

直接解法（ Direct method ）

- ガウスの消去法に基づく
- 決まった演算数で解が求まる
- 行列の分解の際にフィルインを考慮しなければならないため、多大な記憶容量を必要とする

反復解法（ Iterative method ）

- 解の候補を修正しながら反復的に収束解を求める
- 非ゼロ成分だけを記憶すればよいため大規模問題を扱うことができる
- 強力な前処理（ Preconditioning ）が必須

直接法・反復法

（直接法＋解の反復修正）

- 直接法が破たんしないような実装
- 直接法により得られた解を、反復法で修正

（強力な前処理＋反復法）

- 直接法に近い処理
- 少ない反復回数

→結局、上の2つは同じこと

目次

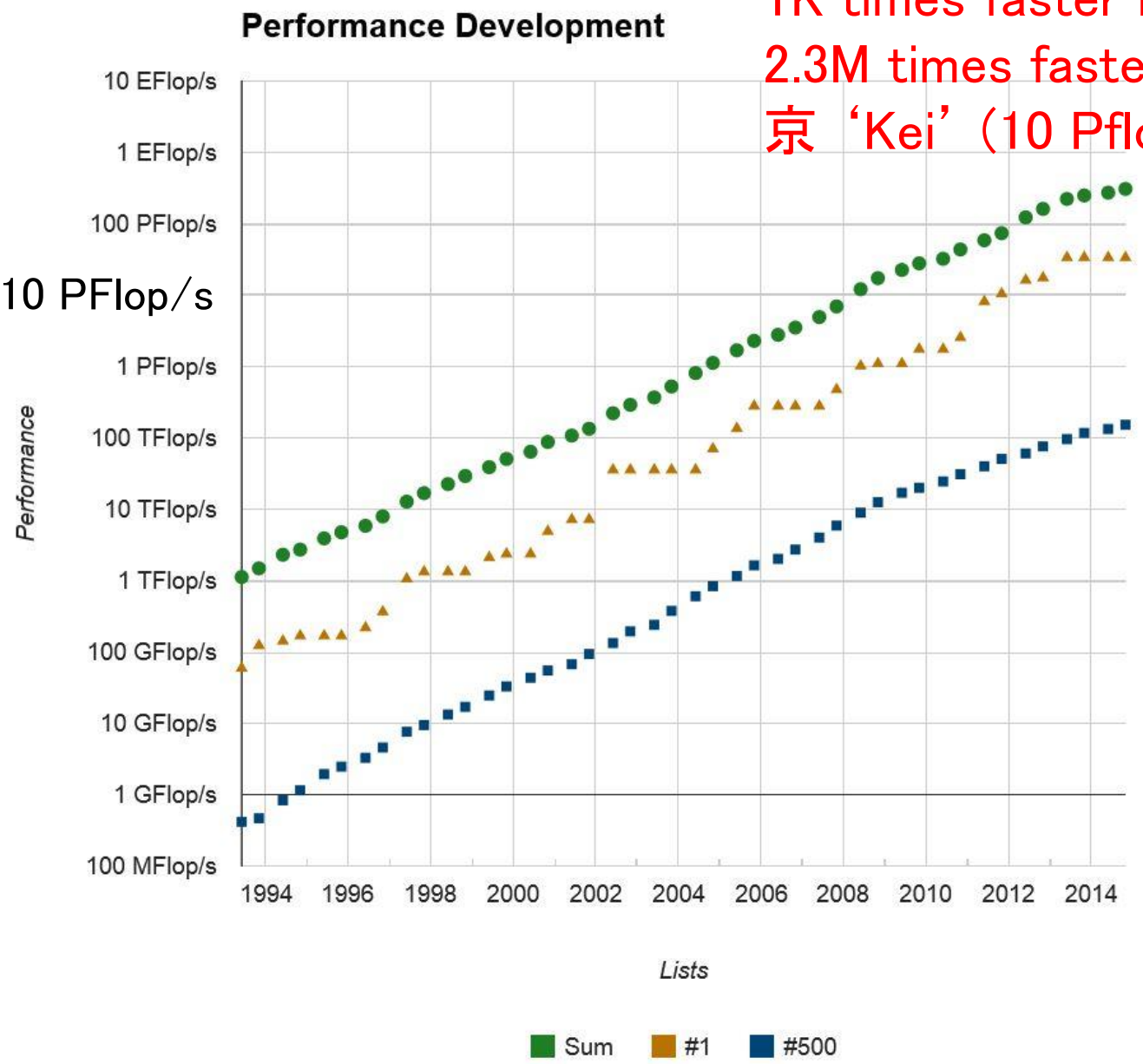
- 導入
  - 計算力学とは
  - 連続体の力学、連立１次方程式
  - FEM構造解析の概要
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FEM計算におけるノード間並列
  - 領域分割とMPI
- FEM計算におけるノード内並列（単体性能）
  - ループ分割とOpenMP
- 実効性能について

1K times faster in 10 years

2.3M times faster in 30 years

京 'Kei' (10 Pflop/s, 2012)

10 PFlop/s



http://www.top500.org

並列処理の重要性

計算機ハードウェアの性能向上には２つの寄与

＜クロック高速化＞

半導体技術と高密度実装技術の高度化　→　限界

＜アーキテクチャにおける並列処理＞　→　近年の性能向上にはこの寄与が大きい

・ベクトル計算

・命令の並列実行

・マルチプロセッサ化、マルチコア化

など

逐次(serial) と 並列(parallel)

逐次処理しなければならない例

```
DO   J=1, N
A(J+1)=A(J)+B(J)
END DO
```

配列Aに関して前のループの演算結果が必要なため、インデックスJについて並列処理することができない。

並列処理できる例

```
DO   J=1, N
A(J)=B(J)+C(J)
END DO
```

プログラムの中には、並列処理できる部分とできない部分がある。

「最適化」「チューニング」

並列化は必須

- 理論性能と実効性能

  アナロジー）自動車の燃費

- 理論性能において、並列処理が大きい寄与を占める

実効性能向上のための工夫

並列化プログラミング

並列化支援の通信ライブラリや並列処理・ベクトル処理の指示文を挿入する　など

オーダリング

演算順序やデータ配置の並べ替えによる、演算の依存性の排除

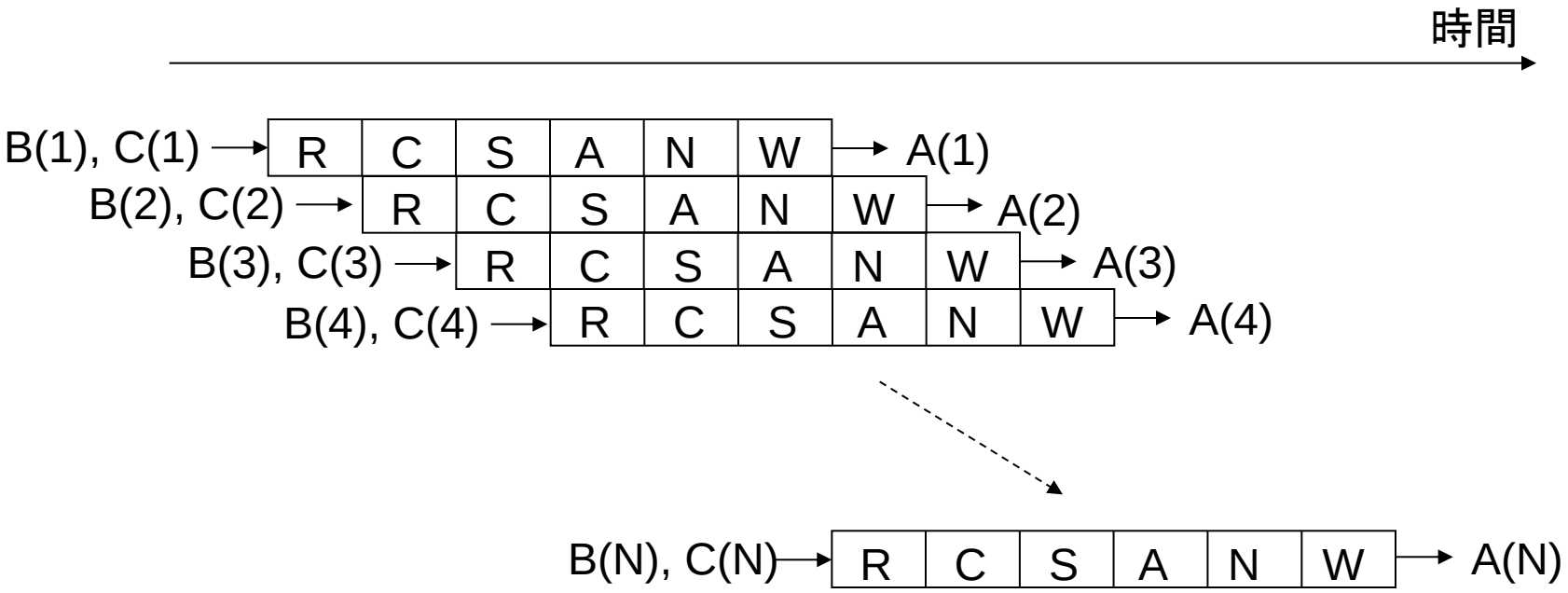
並列アーキテクチャ(1/3)

ベクトル処理　　地球シミュレータ、SIMD

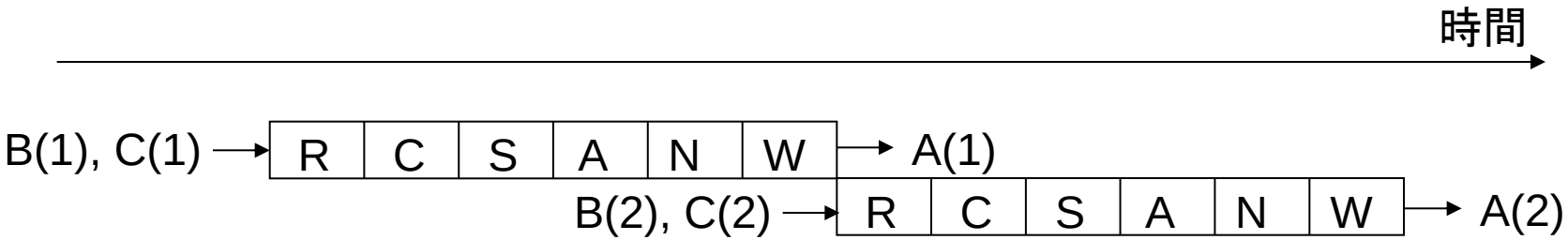
⇔スカラ処理

- データレベルでの並列化のひとつ
- ベクトルデータを同時に処理できるベクトルレジスタとパイプライン(セグメント)化されたベクトル演算器との組み合わせによって高速演算が実現される

ベルトコンベアに沿って並べて置かれた装置によって、演算操作はパイプライン上でオーバーラップしてベクトルデータに次々と実行される(パイプライン実行される)ため、並んだ装置の数だけ速度が向上する。ベクトル型スーパーコンピュータでは、複数個のベクトル演算機を装備してさらに高速化が図られている。



(a) ベクトル処理



(b) スカラ処理

並列アーキテクチャ(2/3)

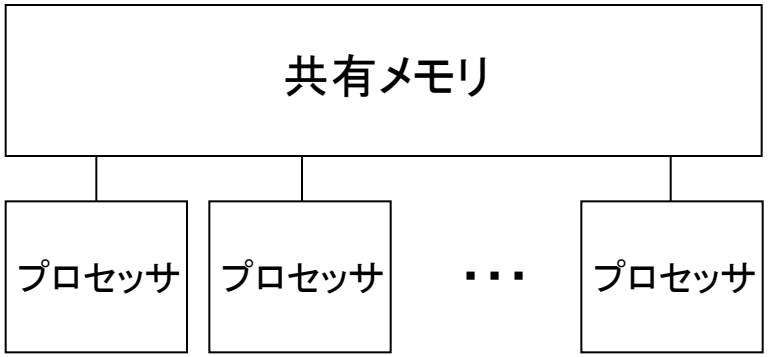
マルチプロセッサ・マルチコア

並列（パラレル）⇔逐次（シリアル）

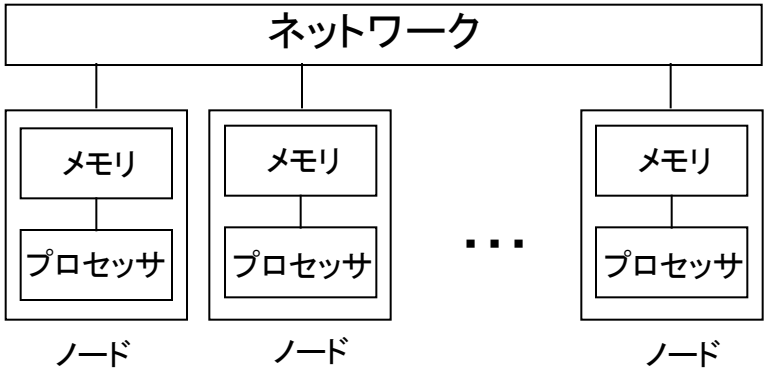
- もう一段階上の並列化レベル
- プロセッサを複数のコアで構成する
- メモリの割り当て方によって3方式

クラスタ計算機、PCクラスタ

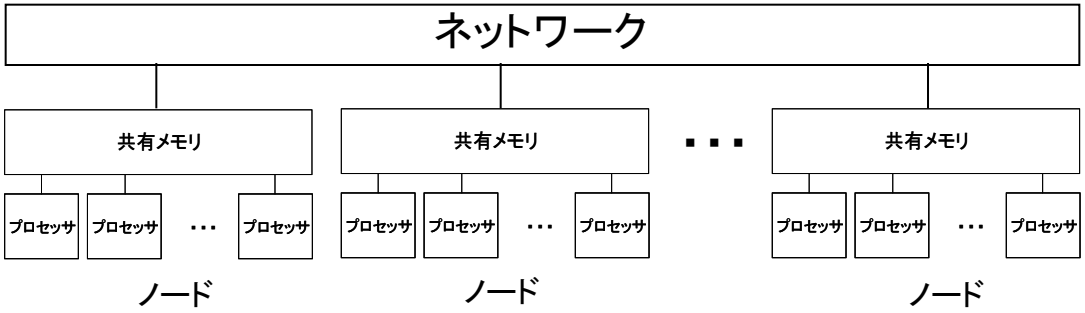
コモディティプロセッサを比較的安価で低速なLANでネットワーク結合



(a) 共有メモリ型



(b) 分散メモリ型



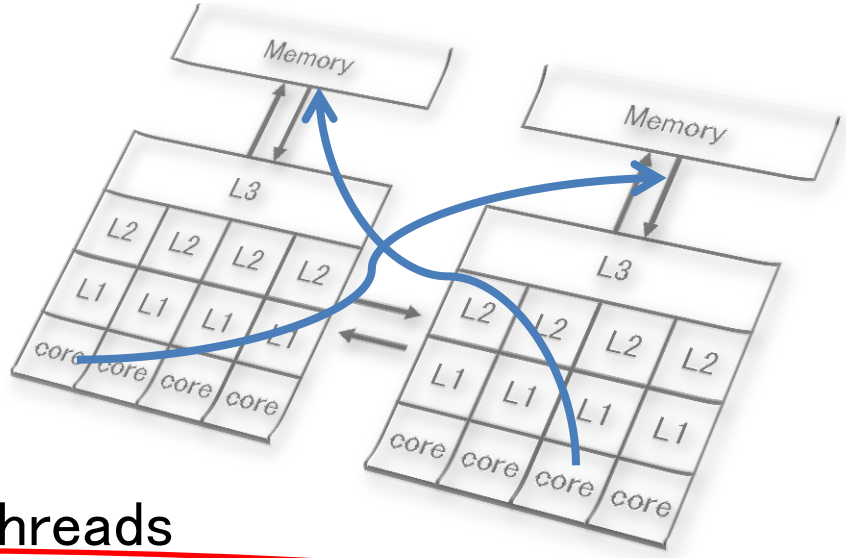
(c) 共有・分散メモリ型

プロセッサはマルチコア化

並列計算機におけるネットワーク、メモリ、プロセッサの構成

Multicore and NUMA (Non-Uniform Memory Access)

- Today's common architecture in consumer and/or enterprise processors



- Programming model
  - ✓ Distributed memory : MPI
  - ✓ Shared memory : Multi-threads
  - ✓ NUMA : Hybrid = MPI + Multithreads

- Performance of core & memory, vs. Intensity Flop/Byte

逆数 BPF

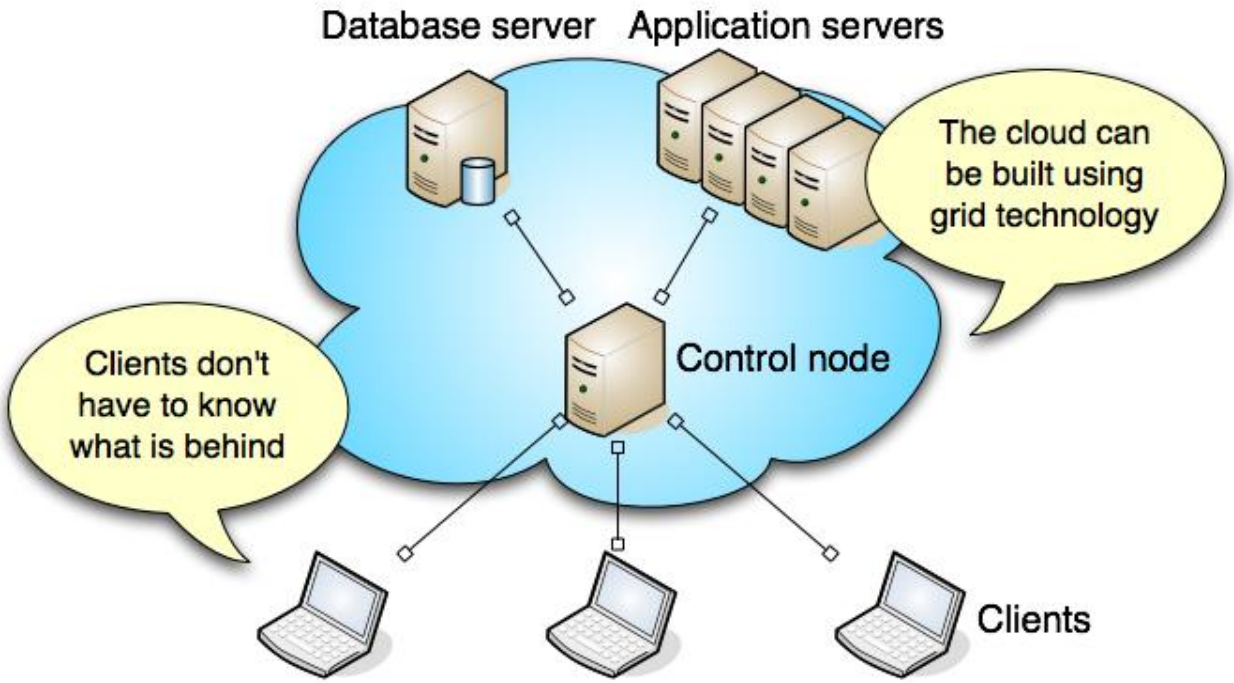
The “K-computer”

- 10 PFLOPS
- # of CPUs　　＞80,000
- # of cores　　＞640,000
- Total memory　＞1PB

- Parallelism
  - Inter-node (node⇔node)　　: MPI
  - Intra-node (core⇔core)　　: OpenMP
  - Flat MPI is NOT recommended

- Hybrid programming is crucial for “K”.

並列アーキテクチャ(3/3)

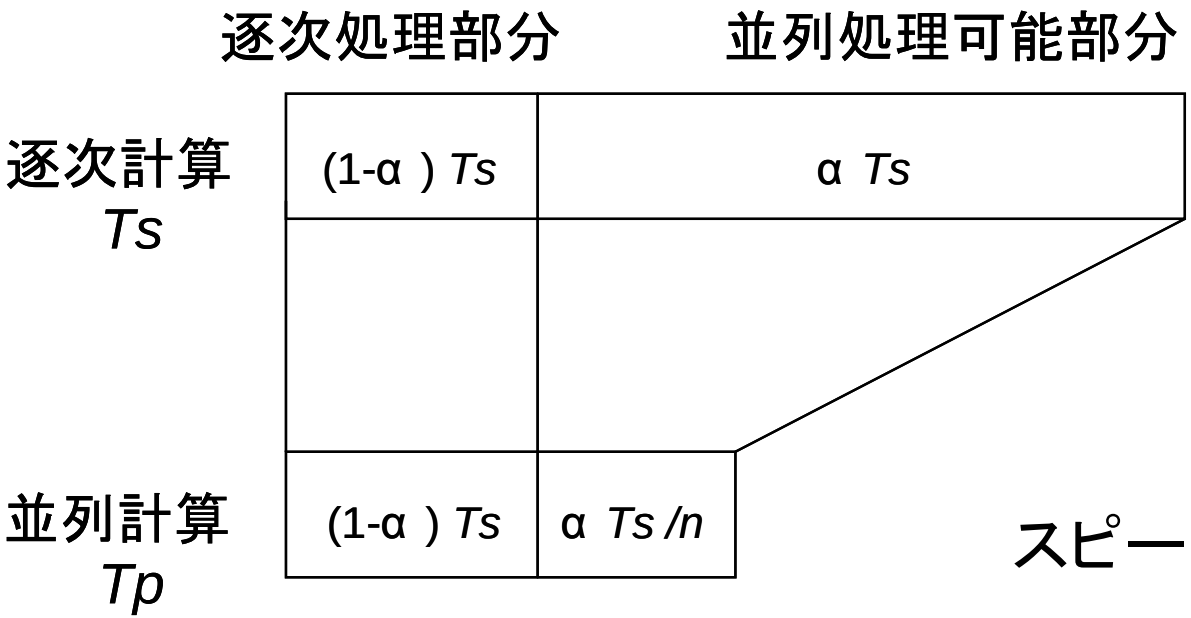


クラウド

「データもプログラムも雲(クラウド)の上に置かれている。ネットに接続するブラウザがあれば、どんな端末からでも雲に届く」(Eric Schmidt, Google CEO)

計算機を並べただけでは速くならない

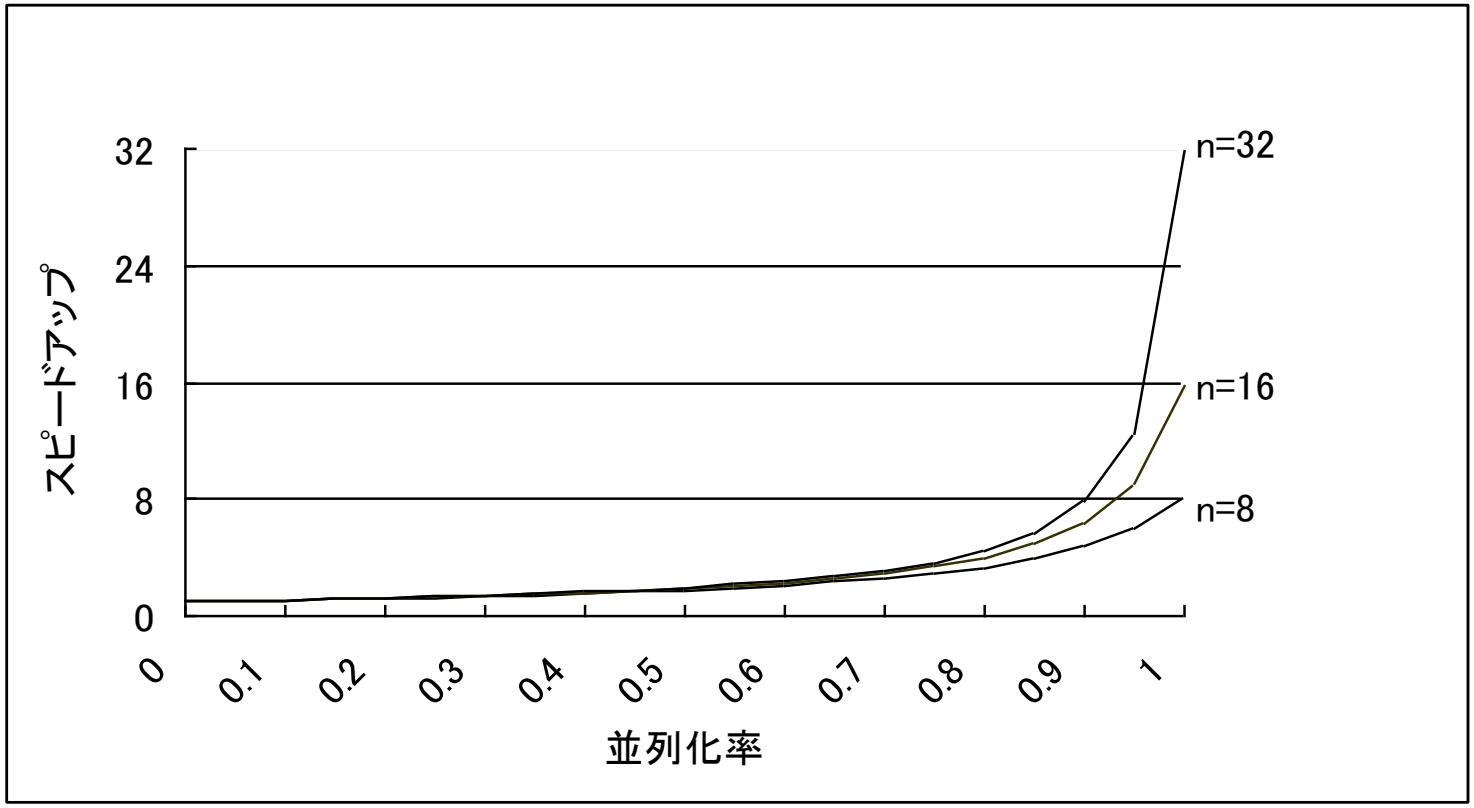
- アムダールの法則
  - プログラムの並列化率と並列化による実効性能向上の関係

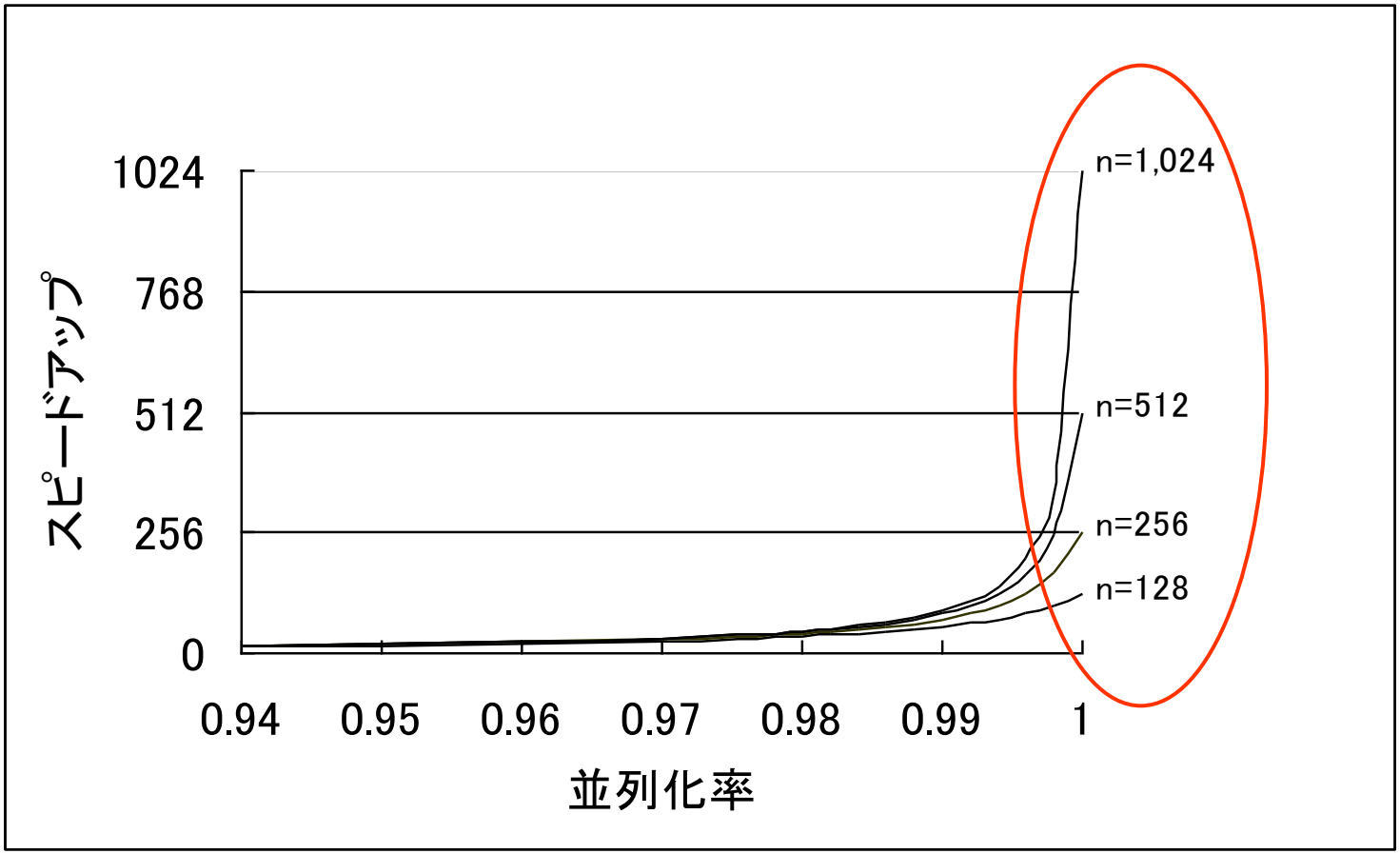


スピードアップ（$= Ts / Tp$）

α：並列化率（並列処理可能な部分に要した時間の割合）
n：プロセッサ数

スピードアップ＝*Ts*／*Tp*
理想値は *n*





横軸に注意（α>0.94以上を表示）

計算全体の演算量（問題規模）は一定に固定されていることに注意。
実際の並列計算においては、多くの場合、プロセッサ（あるいはノード）の数を増やすにつれて計算規模も大きくして実行することが多い。

目次

- 導入
  - 計算力学とは
  - 連続体の力学、連立１次方程式
  - FEM構造解析の概要
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FEM計算におけるノード間並列
  - 領域分割とMPI
- FEM計算におけるノード内並列（単体性能）
  - ループ分割とOpenMP
- 実効性能について

アプリケーションレベルにおける並列化の対象(1/2)

並列プログラミングとは

- ワークやデータを、どのように分散し、どのように同時実行するかをプログラムの中で指示すること

ワークシェアリング、データシェアリング

- 複数のプロセッサでループを分担して処理
- 複数のプロセッサで異なるプログラムを同時に実行
- ネットワーク結合されたノードのメモリにデータを分散し、ノードごとに異なるデータに対して同じ演算を施す

アプリケーションレベルにおける並列化の対象(2/2)

プログラム中での指示方法

- MPI などのメッセージ・パッシング・ライブラリをアプリとリンクして用いる　分散メモリ、共有メモリ
- アプリの中に並列化指示文(OpenMPなど)やベクトル化の指示文を挿入する　共有メモリ
- 並列言語(HPFなど)を用いる

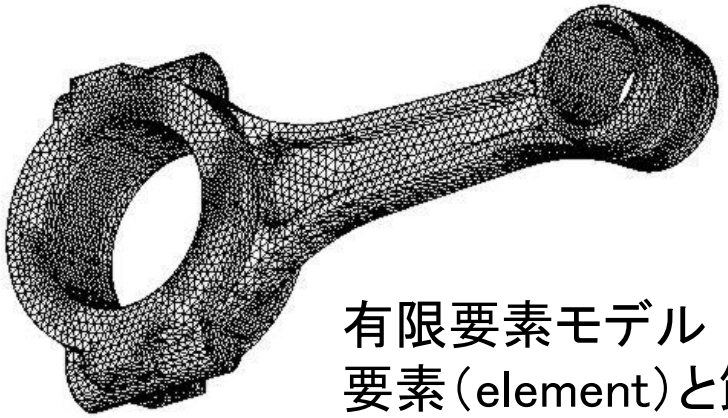
微分方程式をコンピュータで解くには？

微分方程式 → 連立1次方程式 → （各点での物理量が求まる）

（連続体）　（とびとびの点で表現）

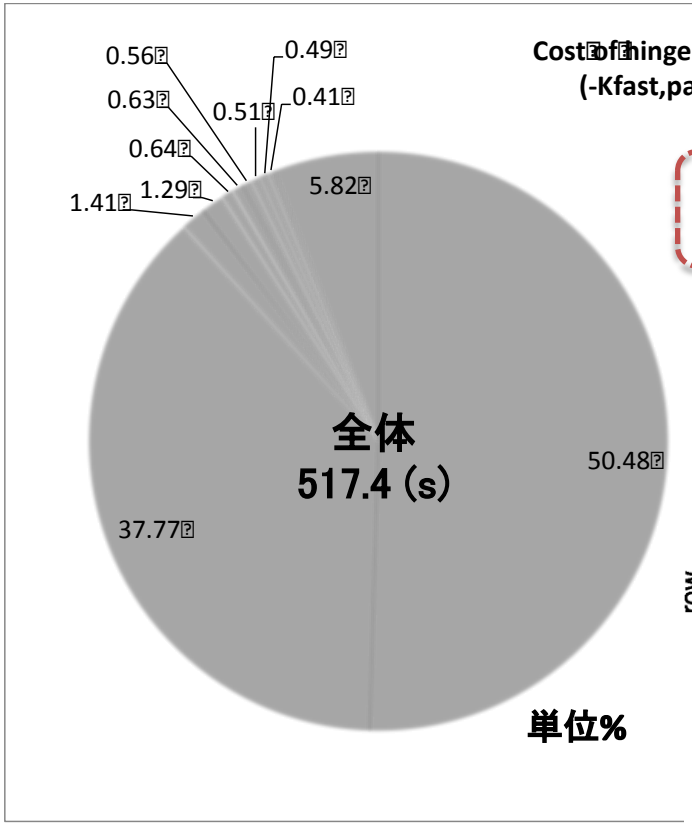
$$[\ K\ ]\{\ u\ \} = \{\ f\ \}$$

CADモデル

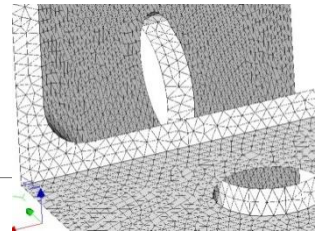


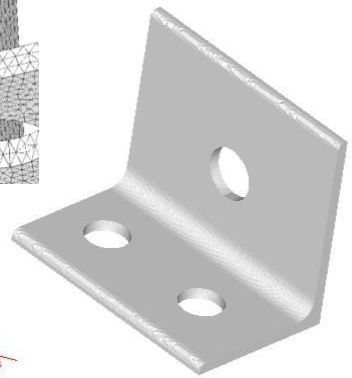
有限要素モデル
要素（element）と節点（node）

節点に物理量が定義されている
（例）節点変位、節点温度



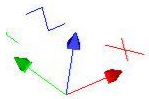
Cost of hingeS1 8 cores
(-Kfast,parallel)



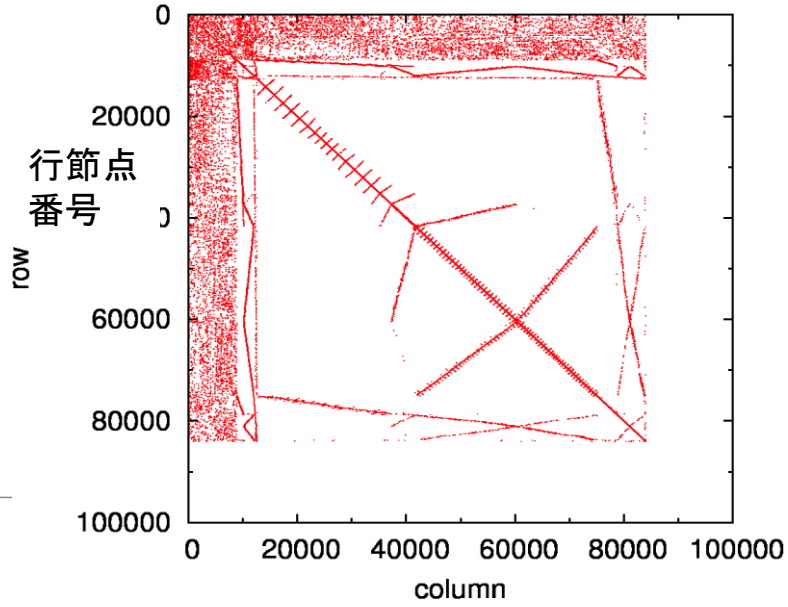


precond_33_&

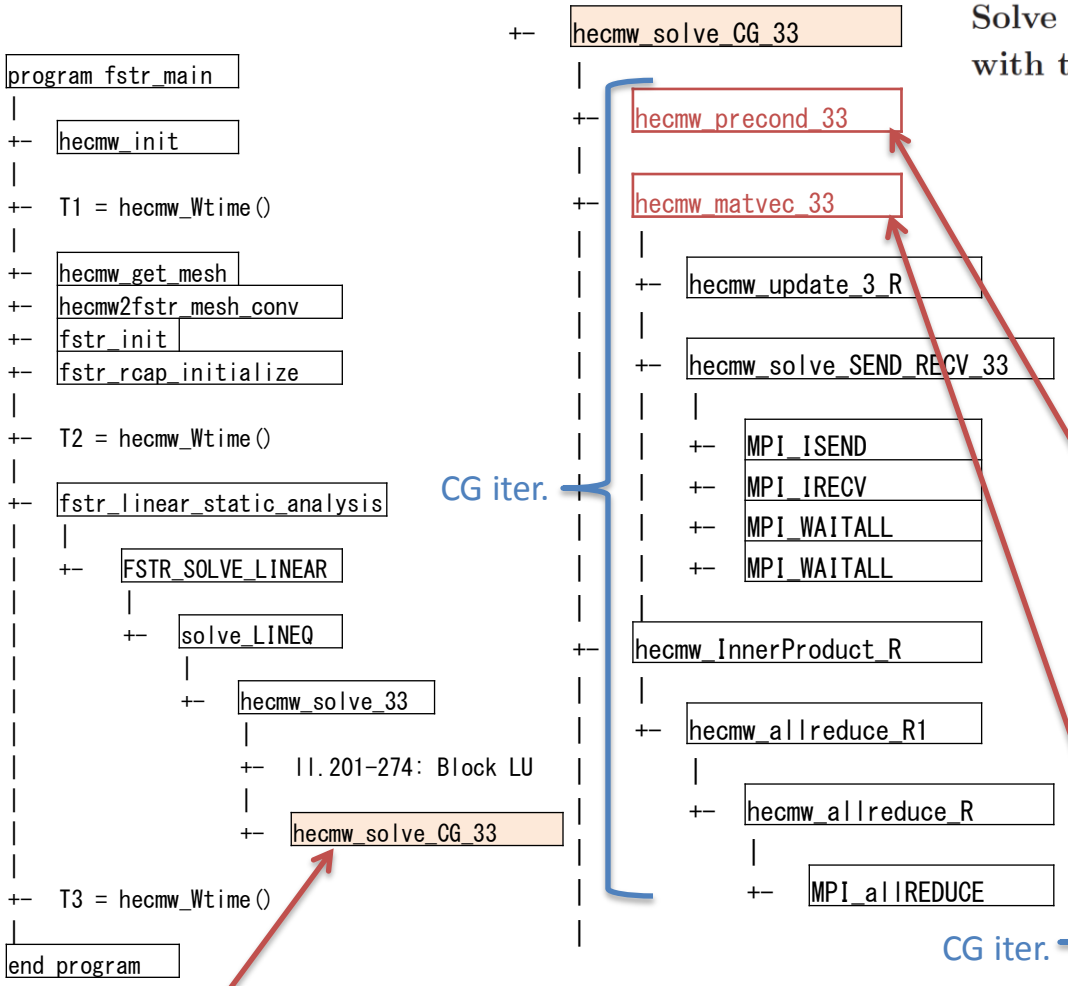
matvec_33_&



hingeS1



hecmw_precond_33とhecmw_matvec_33が高コストなルーチン（全体の90%近くを占める）



Solve $b = Ax$ by the Conjugate Gradient (CG) method with the left preconditioner:

0. Choose the preconditioner M such that $M \simeq A$
   0-a. by Incomplete LU factorization, $M = LU$,
   0-b. by Diagonal block scaling,
   or others
1. Set $\boldsymbol{x} = \boldsymbol{x}_{\rm ini}$: initial guess, and $\boldsymbol{b}$: rhs vector.
2. Set MAXIT: maximum number of iterations, and TOL: tolerance of convergence.
3. $\boldsymbol{r} = \boldsymbol{b} - A\boldsymbol{x}$
4. do iter = 1, MAXIT
5. $\quad \boldsymbol{z} = M^{-1}\boldsymbol{r}$ 前進・後退代入
6. $\quad \rho = (\boldsymbol{r}, \boldsymbol{z})$
7. $\quad$ if (iter .eq. 1) then
8. $\quad\quad \boldsymbol{p} = \boldsymbol{z}$
9. $\quad$ else
10. $\quad\quad \beta = \rho/\rho_1$
11. $\quad\quad \boldsymbol{p} = \boldsymbol{z} + \beta\boldsymbol{p}$
12. $\quad$ endif
13. $\quad \boldsymbol{q} = A\boldsymbol{p}$ 行列ベクトル積
14. $\quad \alpha = \rho/(\boldsymbol{p}, \boldsymbol{q})$
15. $\quad \boldsymbol{x} = \boldsymbol{x} + \alpha\boldsymbol{p}, \quad \boldsymbol{r} = \boldsymbol{r} - \alpha\boldsymbol{q}$
16. $\quad$ Compute the scaled residual norm, RESID = ||r||/||b||
17. $\quad$ if (RESID .le. TOL) exit do-loop
18. $\quad \rho_1 = \rho$
19. enddo

FrontISTRのプログラム構造

高コストルーチンprecond_33とmatvec_33は、剛性方程式の解を求めるCG法においてコールされる。

領域分割

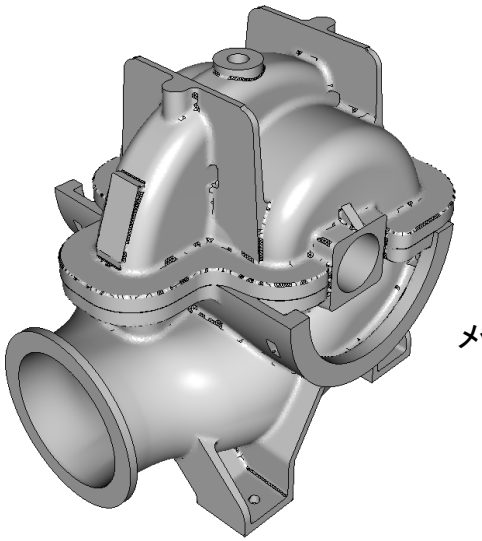
部分領域への分割

- 部分領域のデータを分散メモリに割り当て、各部分領域の計算を並列に実施する。
- 一般に部分領域ごとの計算は完全には独立ではなく、行列ベクトル積や内積計算において、領域全体の整合性をとるために通信が必要。
- 部分領域間での通信ができるだけ少なくてすむように、データが局所化されている必要がある。

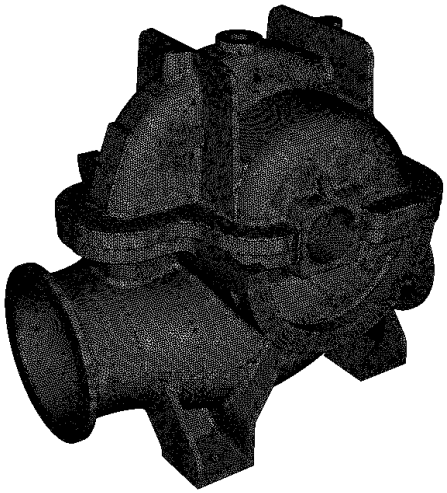
通信テーブル

- 隣接する部分領域との間の節点や要素の接続情報
- 領域分割のツール　METIS、Scotch

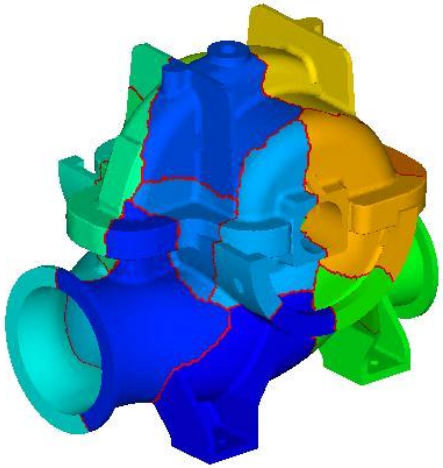
部分領域ごとに行列ベクトル積を実行する．全体領域での行列ベクトル積と同じ結果になるように、部分領域間で通信する．



メッシュ分割

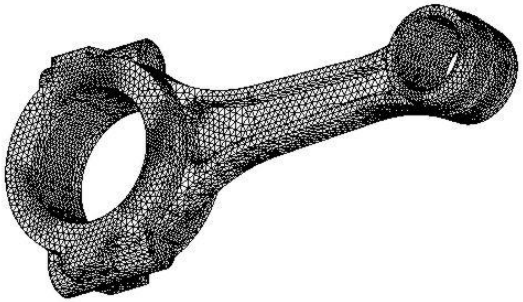


領域分割

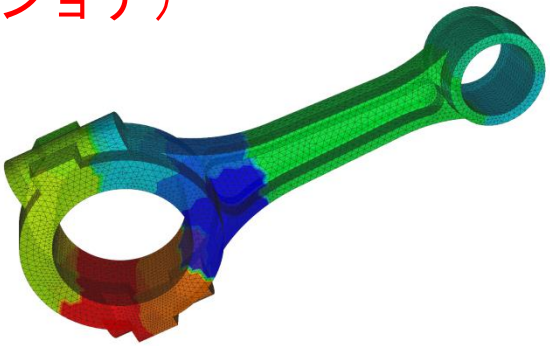


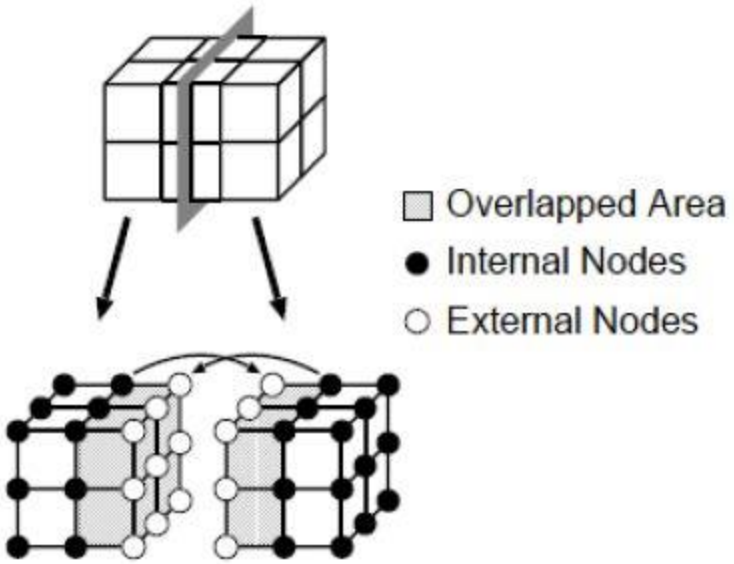
領域分割ツール
（パーティショナ）

CADモデル

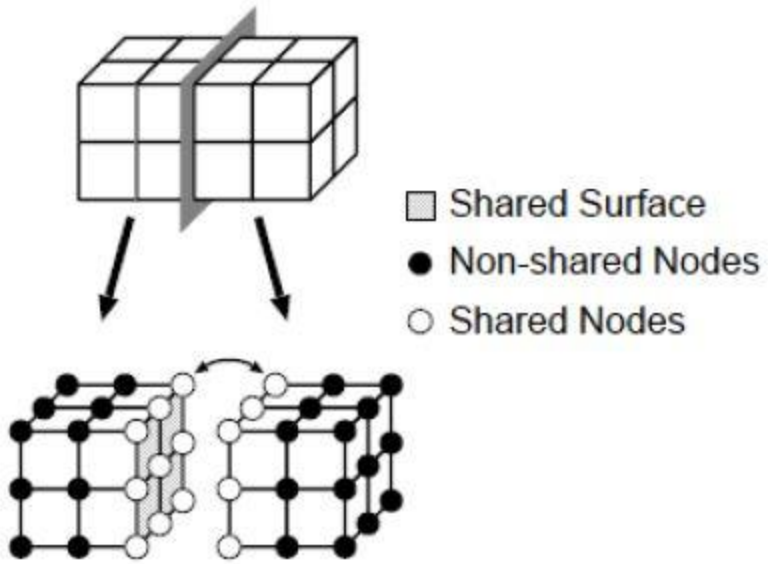


有限要素モデル
要素（element）と節点（node）





(a) Node-based



(b) Element-based

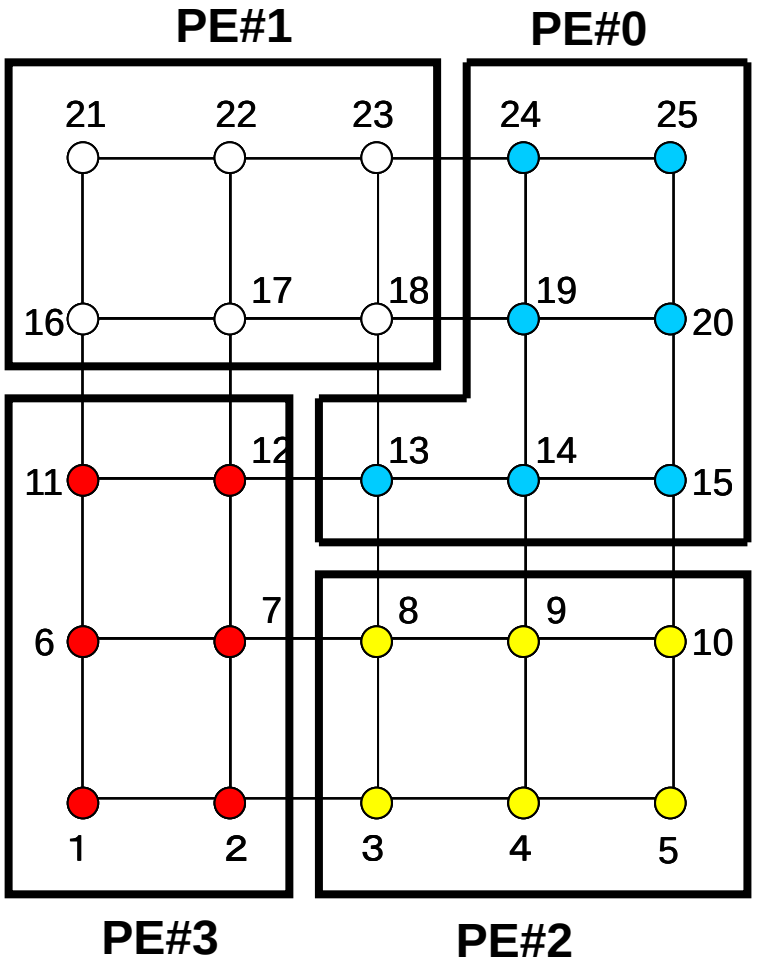
並列プログラミング方法(1/3)

メッセージ・パッシング・ライブラリの利用

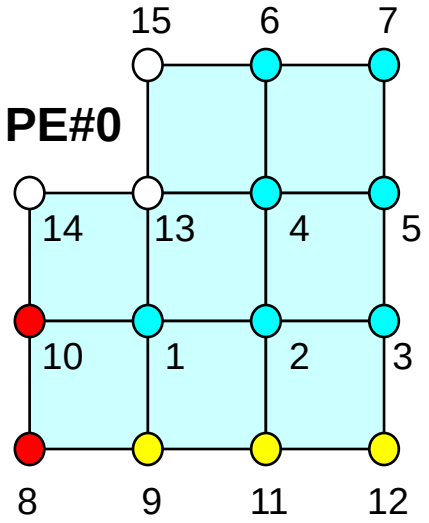
- メッセージ・パッシング・ライブラリ： 分散(共有)メモリ間でネットワークを介して、データを送受信、プロセスの起動や同期などの制御、を行うライブラリ群
- FortranやCなどのAPI
- 逐次プログラムからコールすることで並列計算が可能
- MPI 「MPI」は単に規格を指す。その実装系であるmpichはほとんどのメーカーのプロセッサに対応したものが準備されている。商用の汎用並列計算機にはそれぞれのアーキテクチャに最適化されたMPIが実装されている。
- MPIは共有メモリ、分散メモリ、共有・分散メモリのどの形態の計算機システムにおいても用いられる。

関数名	機　能
MPI_INIT	MPIの起動
MPI_COMM_SIZE	コミュニケータの立ち上げ
MPI_COMM_RANK	コミュニケータ内のプロセスの認識
MPI_FINALIZE	MPIの終了
MPI_BARRIER	各プロセスの同期
MPI_WAITALL	各プロセスの同期
MPI_BCAST	１つの送信元から全プロセスにメッセージを送信する
MPI_ALLREDUCE	すべてのプロセスからメッセージを受信し、それらの算術計算結果を全プロセスに送信する
MPI_SEND, MPI_RECV	１対１ブロッキング通信（送信、受信）
MPI_ISEND, MPI_IRECV	１対１非ブロッキング通信（送信、受信）．MPI_WAITと共に用いる

MPIのサポートする機能の例



(a) 4領域への分割

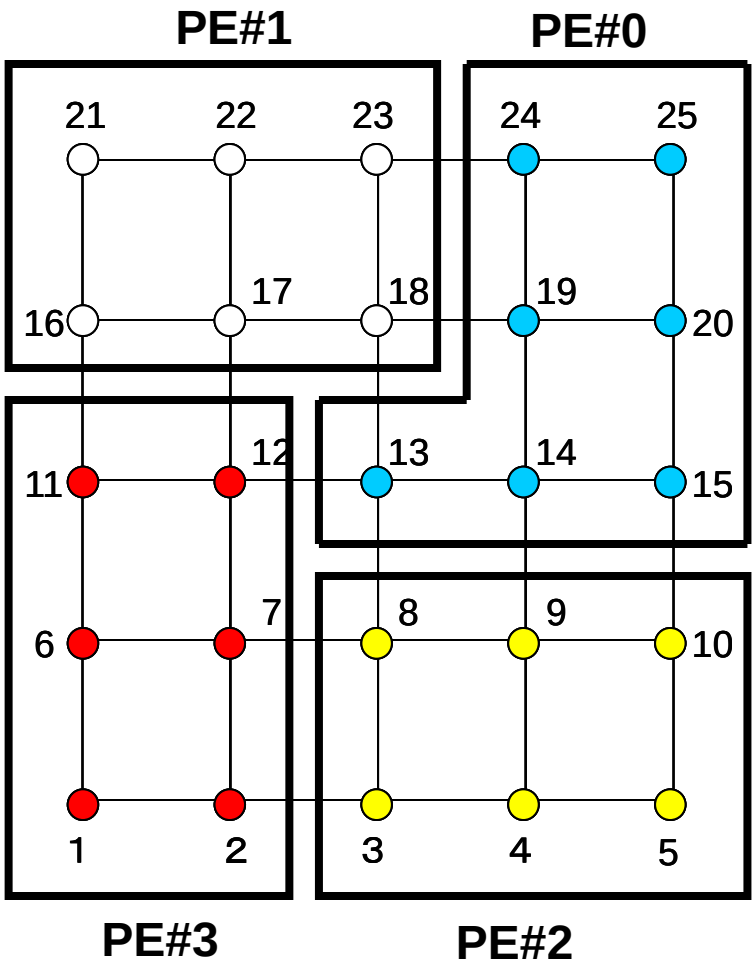


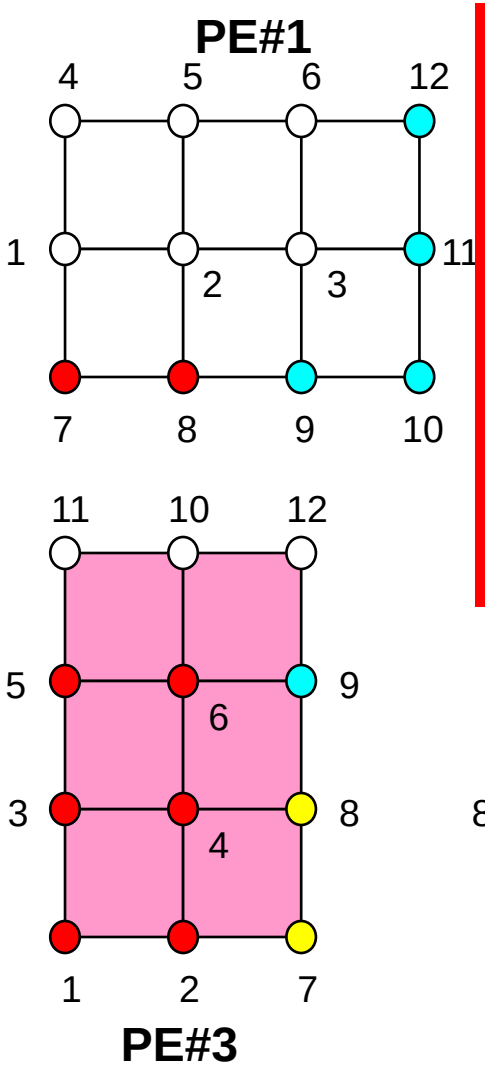
(b) PE#0が保持する情報
（隣接PEとのオーバーラップあり）

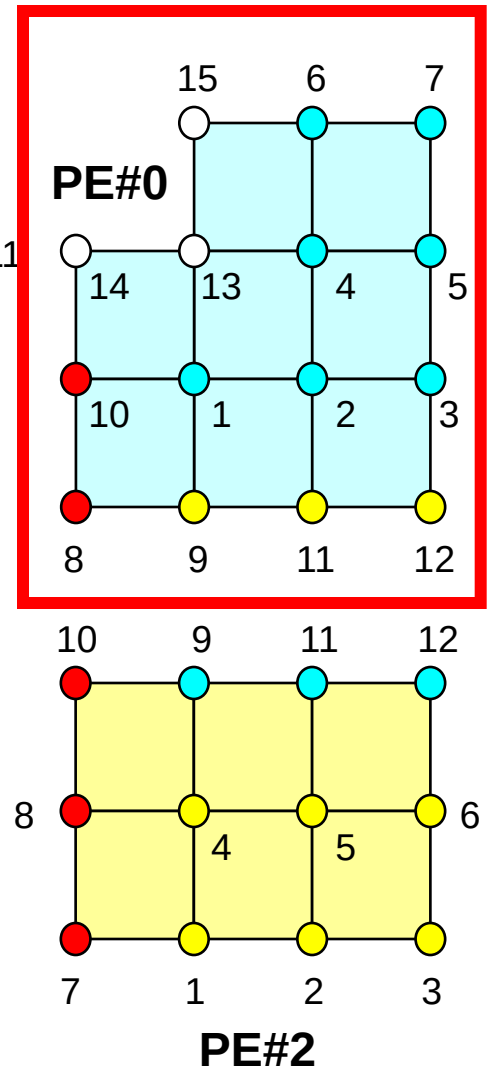
FrontISTRにおける有限要素法データの領域分割例

Local Data Structure

Node-based Partitioning
internal nodes - elements - external nodes

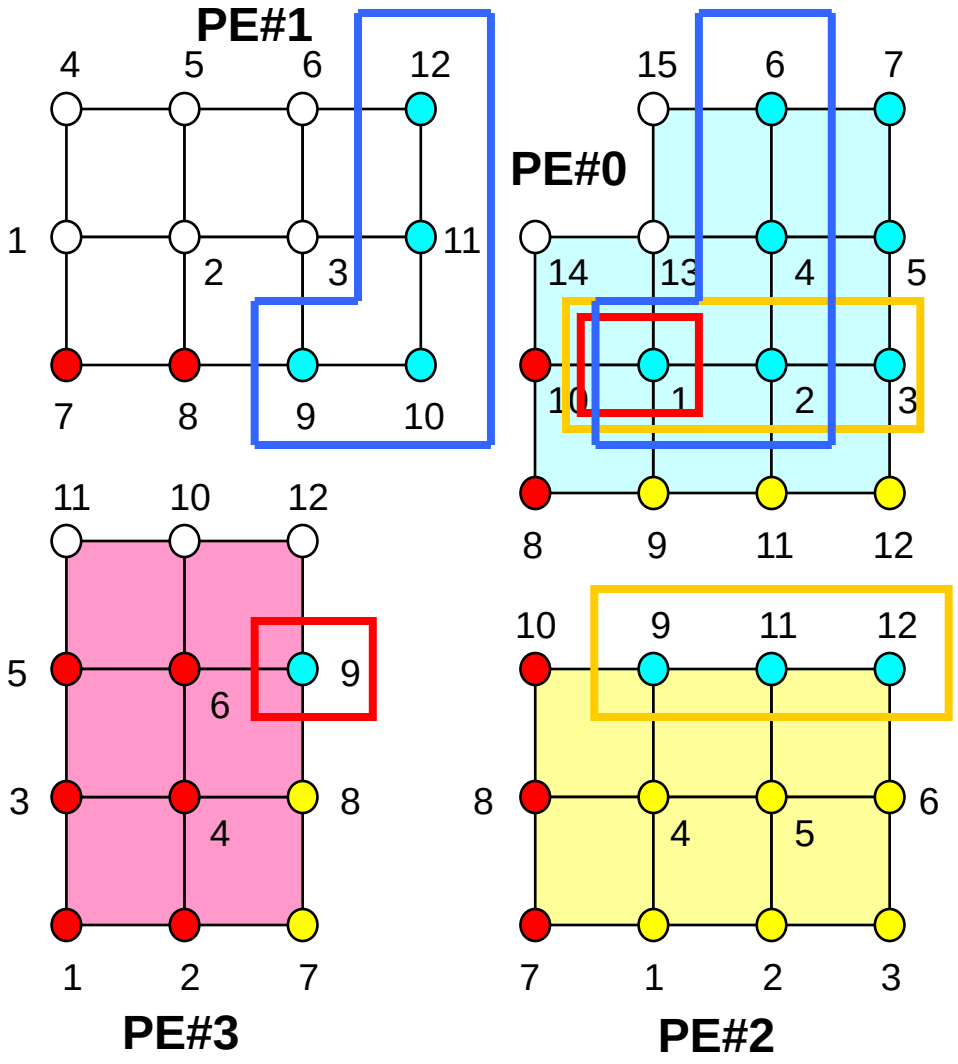






Local Data Structure : PE#0

internal nodes - elements - external nodes



Partitioned nodes themselves
internal nodes

Elements which include “internal nodes”
Provide data locality in order to carry out element-by-element operation in each partition

Nodes included in the elements
external nodes
Numbering : internal -> external

Internal nodes which are “external nodes” for other partitions
boundary nodes

Communication table provides boundary~external node relationship

場所　FrontISTR_V44/hecmw1/src/solver/solver_33

ファイル名　　hecmw_solver_CG_33.f90
　サブルーチン名　　hecmw_solve_CG_33
（↑を使っている限りは、逐次プログラムと同じ）

ファイル名　　hecmw_solver_las_33.f90
　サブルーチン名　　hecmw_matvec_33

場所　FrontISTR_V44/hecmw1/src/solver/communication

ファイル名　　hecmw_common_f.f90
　サブルーチン名　　hecmw_update_3_R

ファイル名　　hecmw_solver_SR_33.f90
　サブルーチン名　　hecmw_solve_send_recv_33

（ hecmw_solve_CG_33 より下層（アプリ開発者には見せない）では、部分領域間での通信が行われている）

ここで、プログラム4個（抜粋）を参照

<送信部分>
do neib=1,NEIBPE　　隣接PE数
istart=INDEX_EXPORT(neib-1)
inum=INDEX_EXPORT(neib)-istart
do k=istart+1, istart+inum
WS(k)=X(NOD_EXPORT(k))　　送信データのWSへの格納
end do
call **MPI_ISEND**(WS(istart+1), inum, …)　　送信
end do

<受信部分>
do neib=1,NEIBPE　　隣接PE数
istart=INDEX_IMPORT(neib-1)
inum=INDEX_IMPORT(neib)-istart
call **MPI_IRECV**(WR(istart+1), inum, …)　　受信
end do
call MPI_WAITALL (NEIBPETOT, …)
do neib=1,NEIBPE　　隣接PE数
istart=INDEX_IMPORT(neib-1)
inum=INDEX_IMPORT(neib)-istart
do k=istart+1, istart+inum
X(NOD_IMPORT(k))=WR(k)　　受信データのXへの格納
end do
end do

目次

- 導入
  - 計算力学とは
  - 連続体の力学、連立１次方程式
  - FEM構造解析の概要
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FEM計算におけるノード間並列
  - 領域分割とMPI
- FEM計算におけるノード内並列（単体性能）
  - ループ分割とOpenMP
- 実効性能について

並列プログラミング方法（2/3）

並列化指示文

- ワークシェアリングやデータシェアリングのための指示文（ディレクティブ）をプログラムの中に挿入する。
- 主に、共有メモリの並列計算機システム。共有・分散メモリのノード内並列化も同様。
- 指示文はプログラム中で見かけ上コメント行のように書かれるが、コンパイル時にオプションを指定することによって、その指示文が解釈されるようになる。
- ポータビリティ（可搬性、移植性）を考慮して指示文を統一化した規格がOpenMP, OpenCL, OpenACCnなど
- ベクトルプロセッサの場合、ベクトルベクトル計算に関する指定も指示文の挿入によって行われる。

```
SUBROUTINE DAXPY(Z,A,X,Y)
INTEGER  I
DOUBLE PRECISION  Z(1000), A, X(1000), Y

!$OMP PARALLEL DO SHARED(Z, A, X, Y) PRIVATE(I)
DO I=1, 1000
Z(I) = A * X(I) + Y
END DO
RETURN
END
```

OpenMPの記述例

!$OMP で始まる文がOpenMPの指示文。DOループが分割され複数のスレッドによって同時実行される。

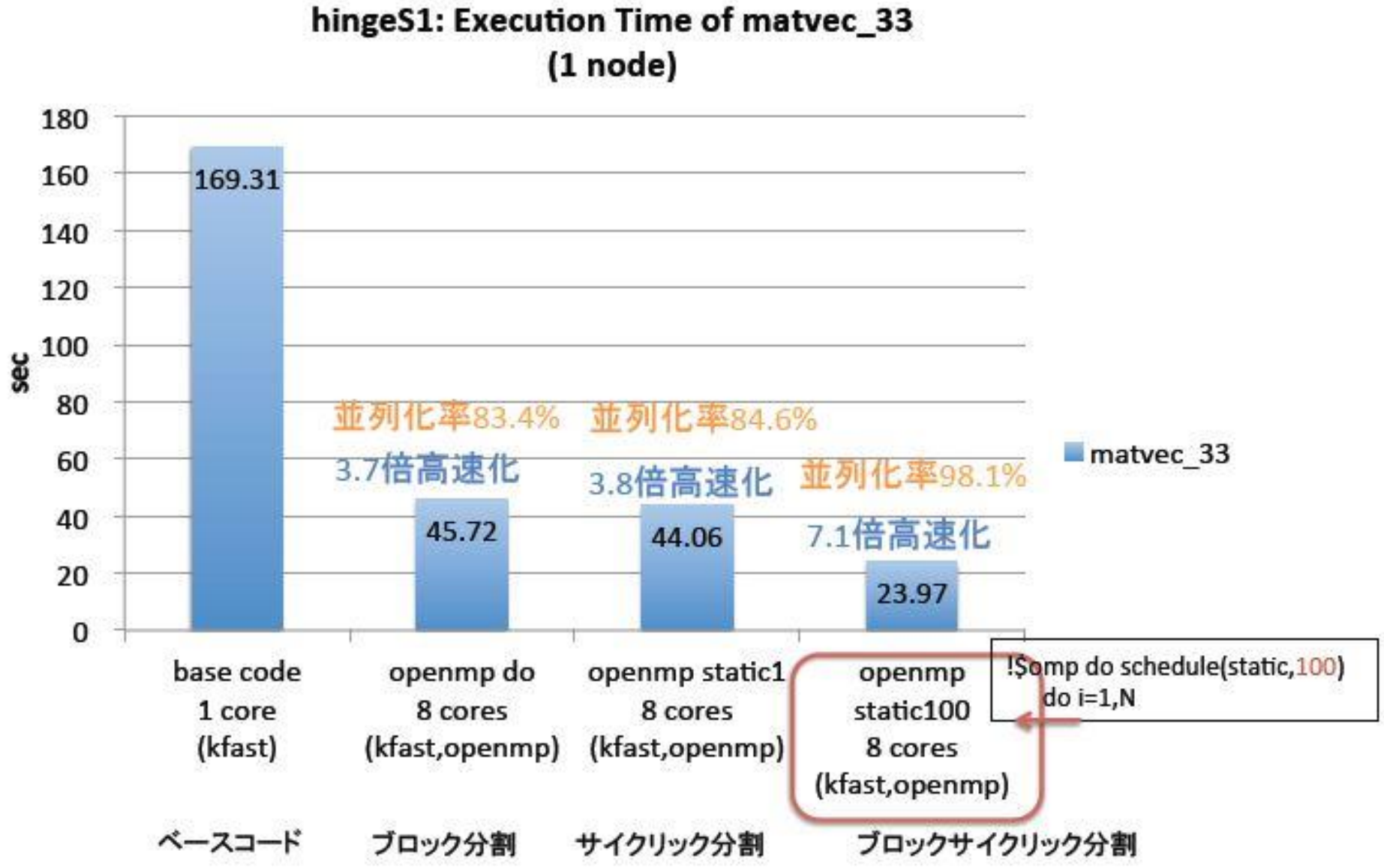
matvec_33のOpenMP化　（「京」でのプログラミング例）

外側の変数 i のループは各 i について配列の参照・代入の依存関係が無い。
i のループをOpenMPでスレッド並列化することが可能。

コンパイルオプション -Kfast,openmp でコンパイルする。

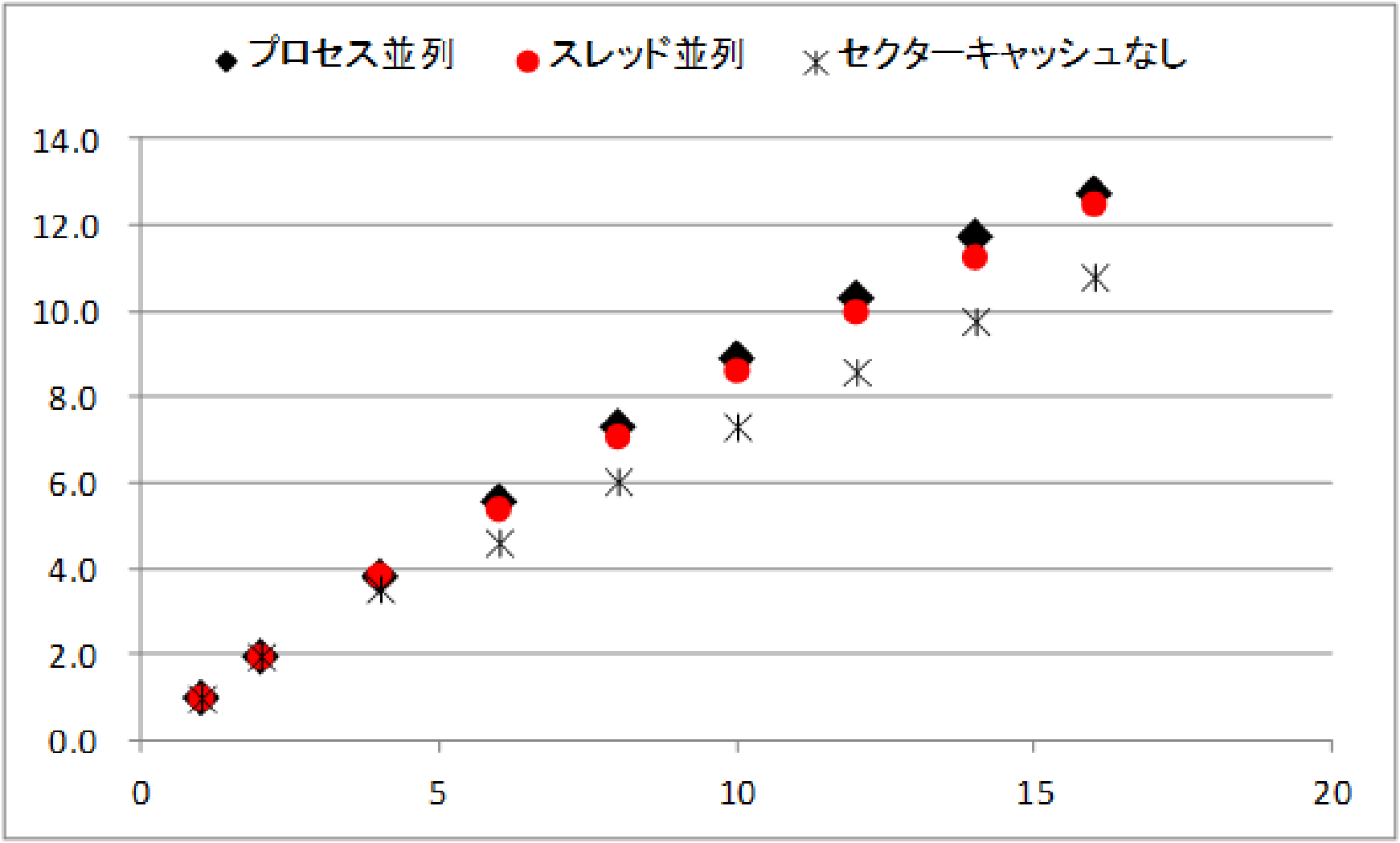
外側のループが
スレッド並列化された

```
32                    !$OMP PARALLEL DEFAULT(NONE) PRIVATE(i,X1,X2,X3,YV1,YV2,YV3,jS,jE,j,in) &
                      !$OMP    SHARED(hecMAT,X,Y)
                      !$OMP DO
33    1  p            do i= 1, hecMAT%N          外側のループ
34    1  p              X1= X(3*i-2)             配列Xは参照のみ。
          (中略)
37    1  p              YV1= hecMAT%D(9*i-8)*X1 + hecMAT%D(9*i-7)*X2                &
38    1                &                         + hecMAT%D(9*i-6)*X3
          (中略)
46    2  p  v           do j= jS, jE             内側のループ
47    2  p  v             in  = hecMAT%itemL(j)
48    2  p  v             X1= X(3*in-2)          配列Xは参照のみ。
          (中略)
51    2  p  v             YV1= YV1 + hecMAT%AL(9*j-8)*X1 + hecMAT%AL(9*j-7)*X2      &
52    2                &                               + hecMAT%AL(9*j-6)*X3
          (中略)
57    2  p  v           enddo
          (中略)
72    1  p              Y(3*i-2)= YV1            配列Yは代入のみ。
          (中略)
75    1  p            enddo
76                    !$OMP END DO
77                    !$OMP END PARALLEL
```



実行時間は7.1倍高速化。
高速化の理由を精密PAで調べる。

（「FX10」での単体性能）



ソルバー部分のノード内並列処理性能の比較（Hingeモデル）

並列プログラミング方法(3/3)

- 並列言語
  - HPF　Fortran90にいくつかの指示文を加え、Fortranの拡張として定義された言語。分散メモリにおける並列化を対象としている。HPFでは、指示文によってデータシェアリングを指定すれば、残るワークシェアリングは分散メモリ間の通信を含めてコンパイラが自動的に並列化を行う。

目次

- 導入
  - 計算力学とは
  - 連続体の力学、連立１次方程式
  - FEM構造解析の概要
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FEM計算におけるノード間並列
  - 領域分割とMPI
- FEM計算におけるノード内並列（単体性能）
  - ループ分割とOpenMP
- 実効性能について

Flop/Byte

SpMV with CSR:

Flop/Byte = 1/{6*(1+m/nnz)} = **0.08～0.16**

SpMV with BCSR:

Flop/Byte = 1/{4*(1+fill/nnz) + 2/c + 2m/nnz*(2+1/r)}

= **0.18～0.21**

nnz: number of non-zero components

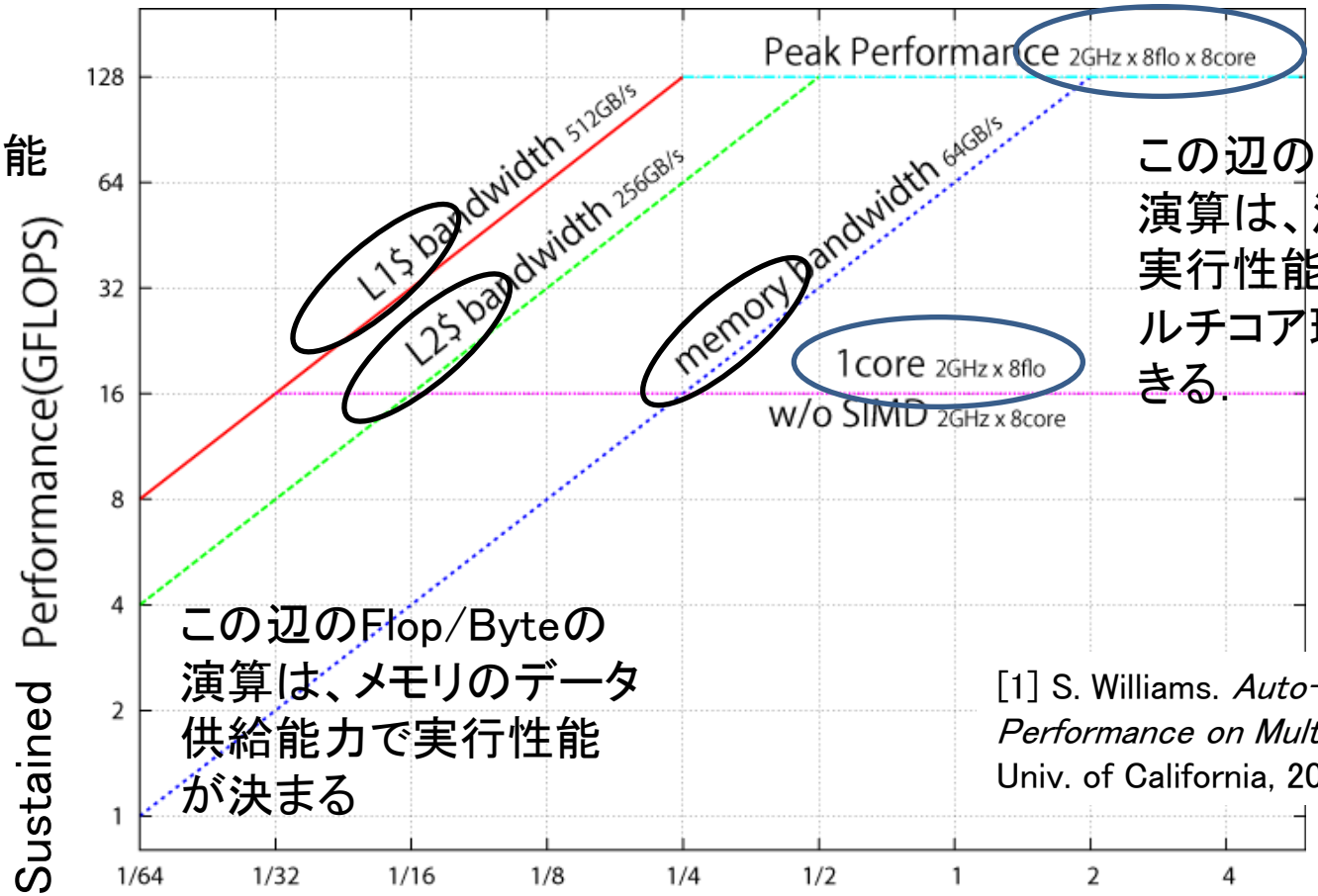
m: number of columns,

r, c: block size,

fill: number of ‘zero’s for blocking

Performance Model (1/2)

- The K-computer's roofline model based on William's model[1].
- Sustained performance can be predicted w.r.t. applications' Flop/Byte ratio.



[1] S. Williams. *Auto-tuning Performance on Multicore Computers*. Univ. of California, 2008.

Performance Model (2/2)

SpMV with CSR
B/F = 6.25～12.5
SpMV with BCSR:
B/F = 4.76～5.56

Machine	Node performance	BW (catalog)	BW (STREAM)	B/F	B/F of FISTR	To-peak
K	128 Gflops	64 GB/s	46.6 GB/s	0.36		
FX10	236.5 Gflops	85 GB/s	64 GB/s	0.27		

Measured performance by profiler
on FX10
4.2 %

SpMV with CSR
2.9～5.8 %
SpMV with BCSR:
4.9～7.6 %

SpMV with CSR
2.2～4.3 %
SpMV with BCSR:
3.7～5.7 %

オーダリング(Ordering)

- ループ依存性
  - i番目の結果がi+1番目以降の計算結果に影響を与えるような場合には、並列処理(あるいはベクトル処理、以下も)してしまうと誤った結果となってしまう。
- オーダリング
  - 配列データの順序を並べ替えるなどの総称。
  - ループ依存性をなくすことができる場合には、コンパイラに強制的に並列処理を指示できる。
  - オーダリングによって、依存性のない演算部がグループ化され、それらに対して並列計算が可能となる。
  - 演算が節点や要素についてのループである場合には、依存性の有無は節点や要素の接続関係から判断することができる。

オーダリング(Ordering)

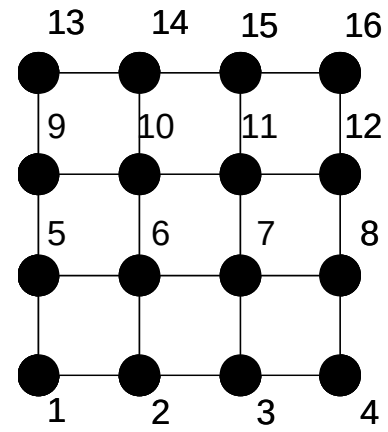
例えば、次式のような演算を考える。

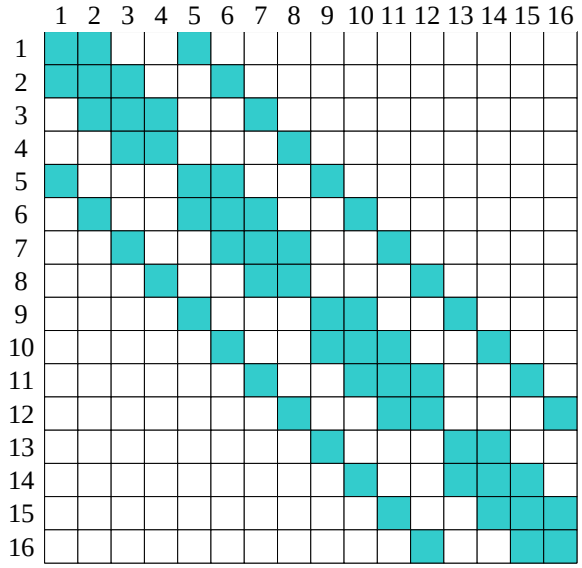
$$x_i = y_i - \sum_{k=1}^{i-1} L_{ik} x_k$$

- ここで、添字は節点のインデックスを表す。このような演算は、連立一次方程式の解法など多くの行列演算に現れる。

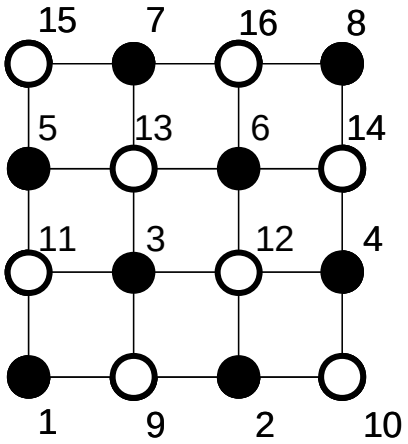
- オーダリング前の節点番号付けの場合、節点iの演算の際にそれ以前に演算済みの情報が必要であることがわかる。
- 依存性のない節点を2色に色分けて(黒と白)番号を付け替えた場合、同一色に属する節点に関する演算は互いに依存性がないことがわかる。すなわち、ノード内並列処理やベクトル処理が可能となる。

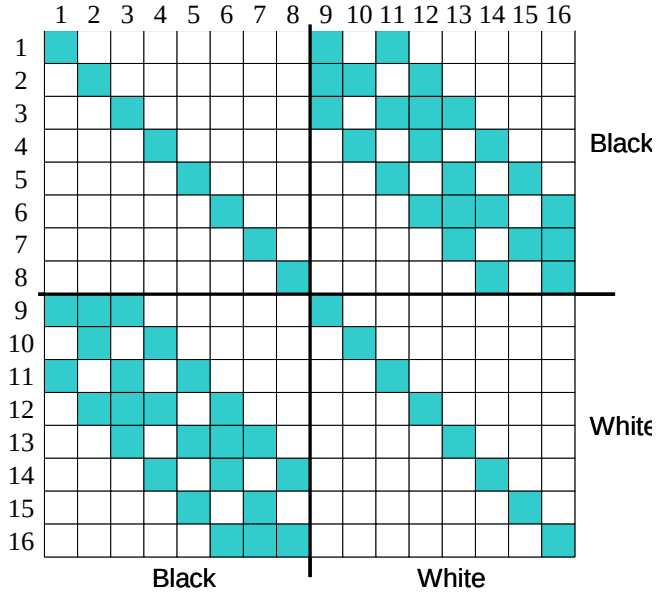
- red and black法、マルチカラー法
- 演算に依存性のない節点をハイパープレーンと呼ばれるグループに分類し、各ハイパープレーン上の節点についてノード内並列処理やベクトル処理を行うことも多い。





オーダリング前





オーダリング前
（2色）

節点番号

行列のプロファイル

◇ 「ノード間並列」について詳しく解説した

◇ Work Ratio を高くとることで、一般に、ノード間並列性能、Weak Scalingは良好な値が得られる

———

◇ 「ノード内並列」以降の部分は、時間不足のため後日にあらためて解説予定

◇ 対ピーク性能（＝実効性能／理論性能）を上げるには、「ノード内並列（＝スレッド並列＝CPU単体性能）」が重要