2016年6月10日（金）
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
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FrontISTRにおける並列計算
- 実効性能について

- ノード間並列
  - 領域分割とMPI
- ノード内並列（単体性能）
  - ループ分割とOpenMP

今回はこの話が中心です

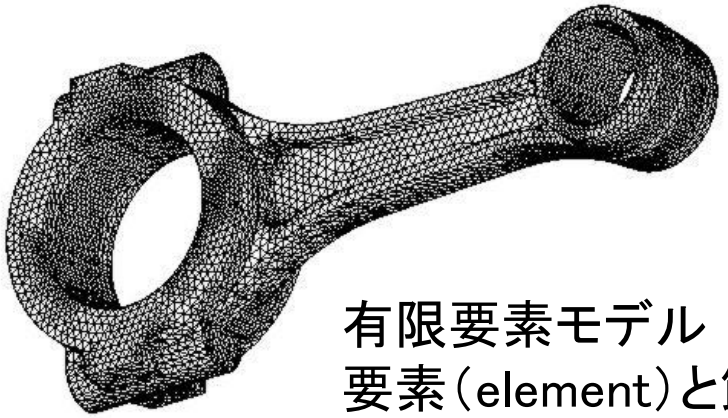
微分方程式をコンピュータで解くには？

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）

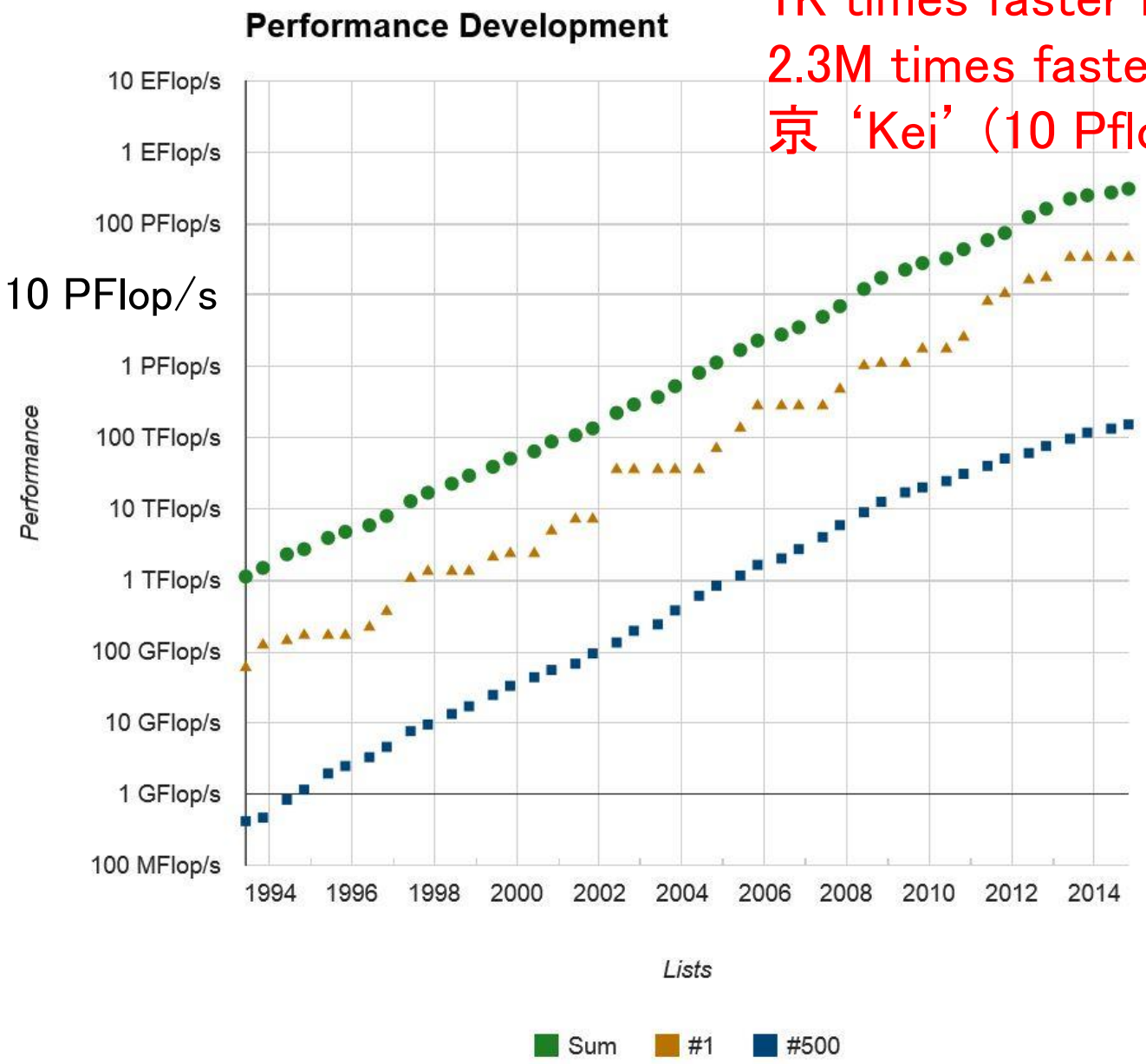
- 直接法に近い処理
- 少ない反復回数

→結局、上の2つは同じこと

1K times faster in 10 years

2.3M times faster in 30 years

京 'Kei' (10 Pflop/s, 2012)



http://www.top500.org

なぜ構造解析の並列計算コードが必要か？

- 計算機はハード並列化によって高速化と大容量化を実現.
- CPUは演算を複数のコアで同時実行. データは階層的に配置されたキャッシュメモリによって効率的に処理.
- PCクラスタは、こうしたCPUを10～100個程度、ネットワークで結合して並列に実行.
- スパコンは、より高速なメモリやネットワークを用いてCPUを数千個かそれ以上結合し、計算の規模と速度をスケールアップ.

- こうした潮流は今後も続くと思われる. 計算機の本来の性能を発揮し、構造解析の能力を高度化してゆくためには、演算とデータ転送の並列化、階層化を考慮したソフトウェア設計の見直しが必要.

Trends in Parallel Architecture and Parallel Programing Strategies (1/2)

<table>
<tr><th>Parallelism</th><th colspan="6">Points of concern</th></tr>
<tr><td>Inter-node via network</td><td></td><td></td><td></td><td colspan="2">Memory distribution over network</td><td>InfiniBand, Ethernet, Myrinet</td></tr>
<tr><td rowspan="3">Intra-node</td><td></td><td rowspan="2">Number of cores</td><td rowspan="2">Programability</td><td colspan="3">Memory</td></tr>
<tr><td></td><td>Size (GB)</td><td>Throughput (GB/s)</td><td rowspan="3">MSU:
Large and slow

L1～L3:
Small and fast</td></tr>
<tr><td>CPU</td><td>O(1) ↗</td><td>good</td><td>O(100)</td><td>O(10)</td></tr>
<tr><td>GPU</td><td>O(100)</td><td>↗</td><td>O(1)</td><td>O(100)</td></tr>
</table>

Between CPU-GPU : PCIe O(1)

Trends in Parallel Architecture and Parallel Programing Strategies (2/2)

Parallelism	Points of concern			
	Parallel efficiency E1 x E2	Programing model	Strategy	Scalability
Inter-node via network	E1	MPI	High work ratio (Localized mesh)	Weak scale
Intra-node	E2	MPI, Thread, OpenMP, OpenCL, OpenACC	Appropriate B/F & Long vector (Blocking, Padding, Reordering)	Strong scale

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

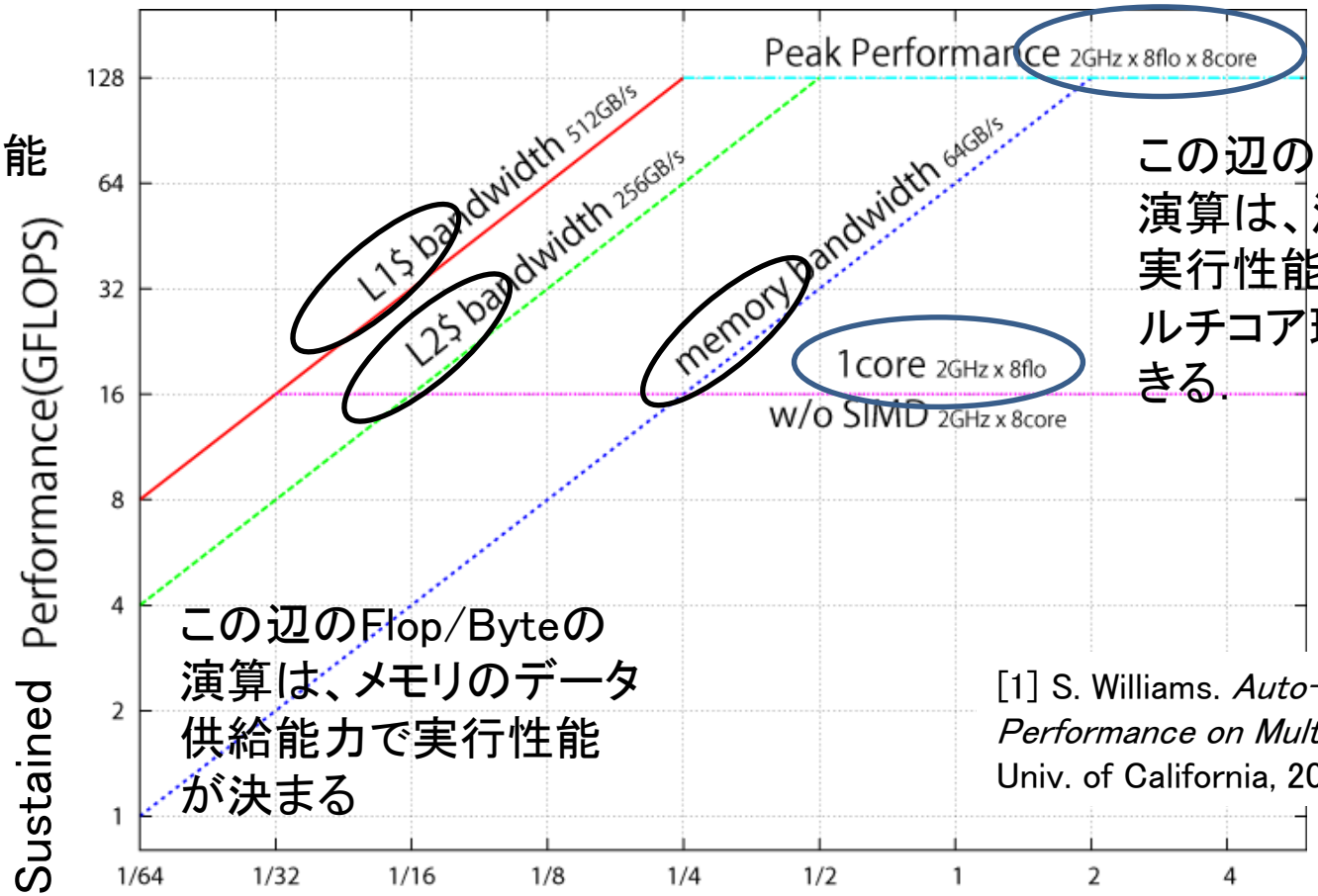
m: number of columns,

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 %

◇　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

◇　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

「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

リオーダリング

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(1/2)

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。

並列アーキテクチャ(2/2)

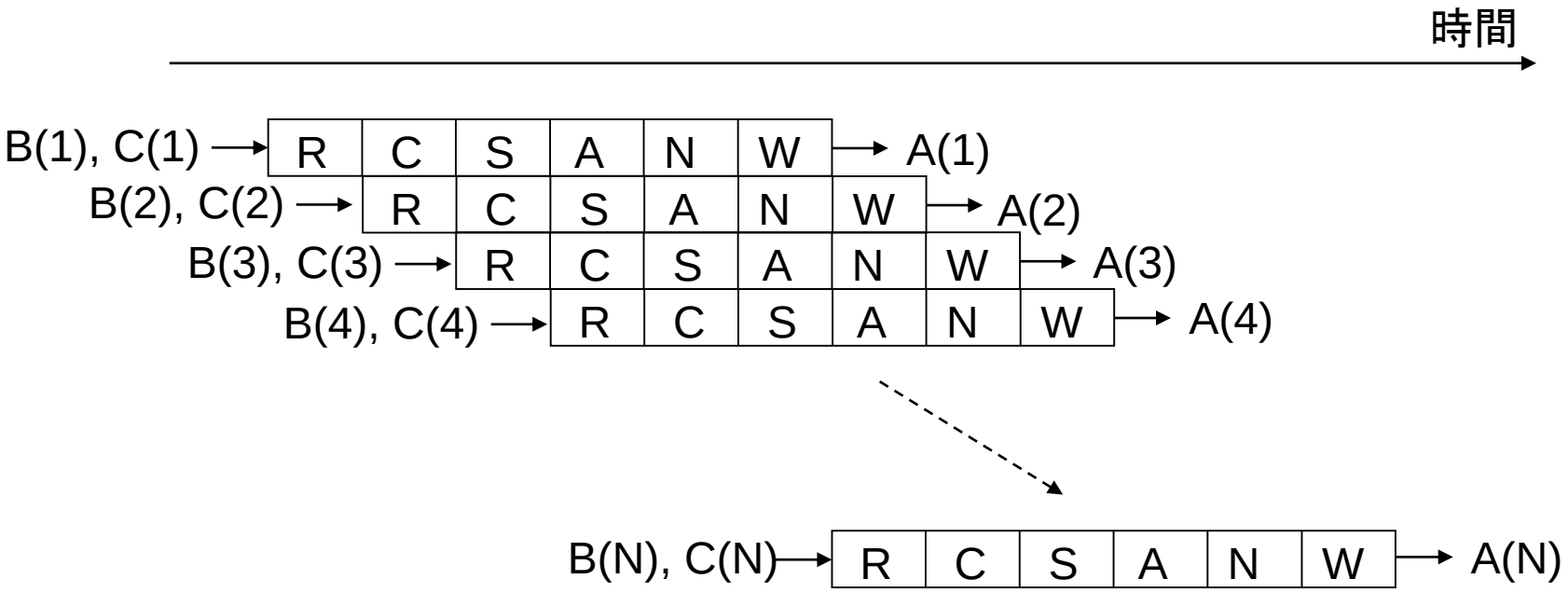
マルチプロセッサ・マルチコア

並列(パラレル)⇔逐次(シリアル)

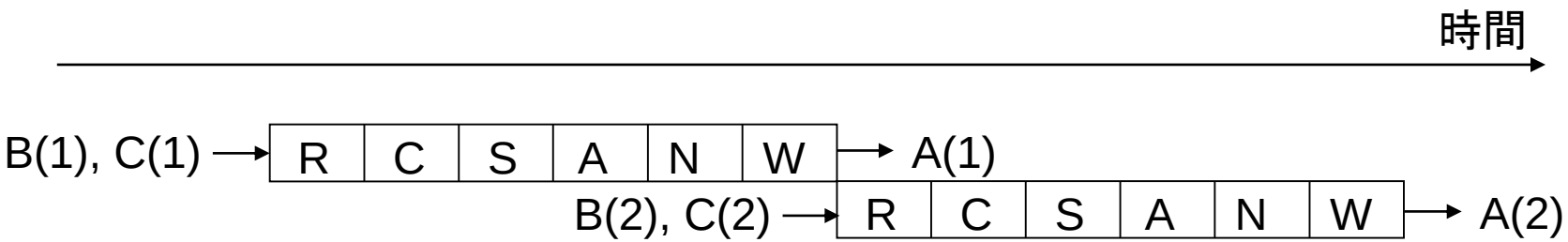
- もう一段階上の並列化レベル
- プロセッサを複数のコアで構成する
- メモリの割り当て方によって3方式

クラスタ計算機、PCクラスタ

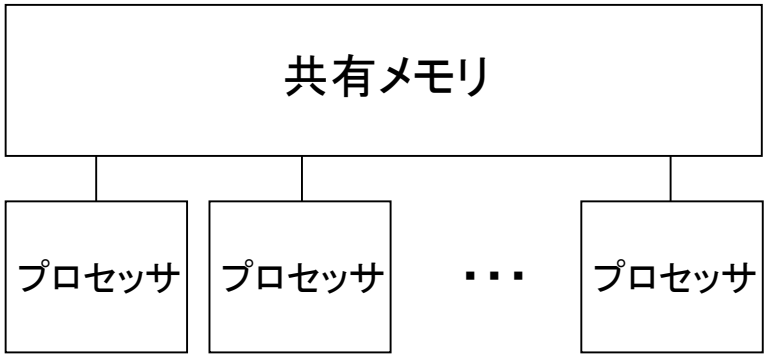
コモディティプロセッサを比較的安価で低速なLANでネットワーク結合



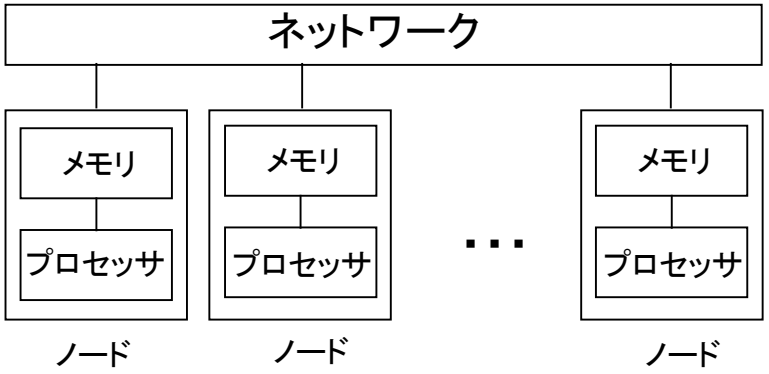
(a) ベクトル処理



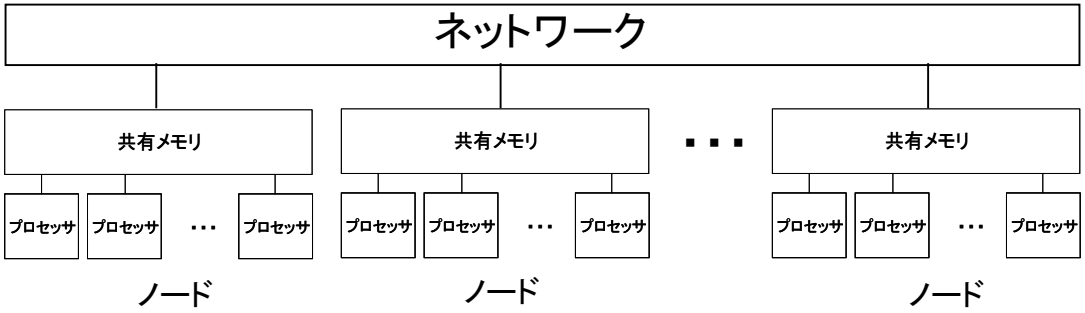
(b) スカラ処理



(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

The “K-computer”

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

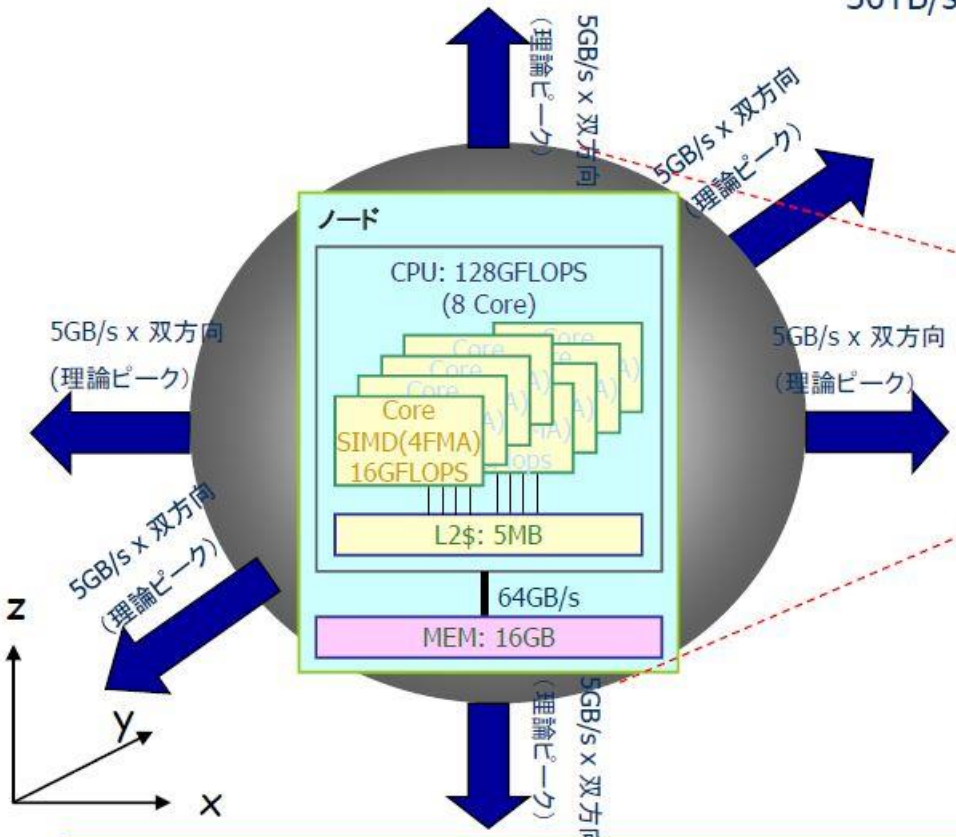
- Parallelism
  - Inter-node (node⇔node) : MPI
  - Intra-node (core⇔core) : OpenMP
  - Flat MPI is NOT recommended

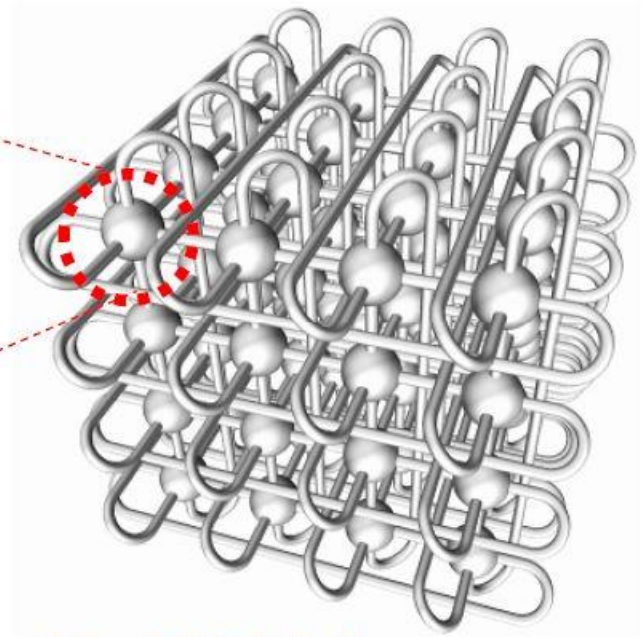
- Hybrid programming is crucial for “K”.

本体システムの構成図

- 計算ノード数：8万以上
  - CPU数：　8万以上
  - コア数：64万以上
- ピーク演算性能：10PFLOPS以上
- メモリ総容量：1PB以上（ノード当り16GB）

- ネットワーク：Tofuインターコネクト（ユーザービューは3Dトーラス）
  - 帯域幅：3次元の正負各方向にそれぞれ5GB/s x 2（双方向、理論ピーク）
  - 同時通信数:4
  - バイセクションバンド幅：30TB/s（双方向、理論ピーク）以上

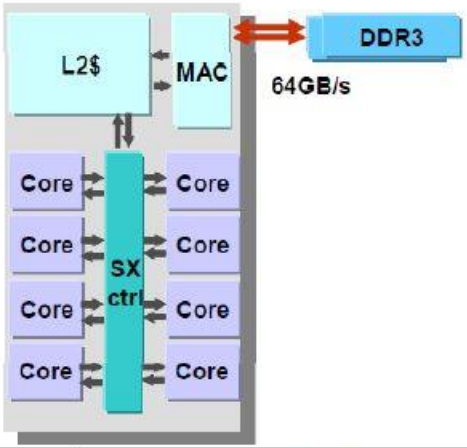




3Dトーラスのイメージ

プロセッサ構成

- 8コア構成，各コア256本の浮動小数点レジスタを備えたスーパースカラ方式
  - SIMD拡張（積和演算4個，逆数近似演算4個など）
  - コア当り16GFLOPS，CPU当り128GFLOPS
- 大容量コア共有キャッシュ（5MB）
  - ハードウェアバリア機構
  - プリフェッチ機構
  - キャッシュを2つのセクタに分け，データの再利用性に応じた使い分けをソフトウェアから制御（次スライド参照）
- データ供給能力
  - レジスタ-L1キャッシュ間：4B/FLOP
  - L1キャッシュ-L2キャッシュ間：2B/FLOP
  - L2キャッシュ-主記憶間：0.5B/FLOP



	仕　様
CPU性能	128GF（16GFx8コア）
コア数	8個
浮動小数点演算器構成（コア当り）	積和演算器：2×2個（SIMD）拡張 逆数近似演算器：2×2個（SIMD）拡張 除算器：2個 比較器：2個 ビジュアル演算器：1個
	浮動小数点レジスタ(64ビット)：256本 グローバルレジスタ(64ビット)：188本
キャッシュ構成	1次命令キャッシュ：32KB(2way) 1次データキャッシュ：32KB(2way) 2次キャッシュ：5MB(10way)コア間共有
メモリバンド幅	64GB/s(0.5B/F)

より詳細な情報は、「SPARC64™ VIIIfx Extensions」を参照のこと
http://img.jp.fujitsu.com/downloads/jp/jhpc/sparc64viiifx-extensions.pdf

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

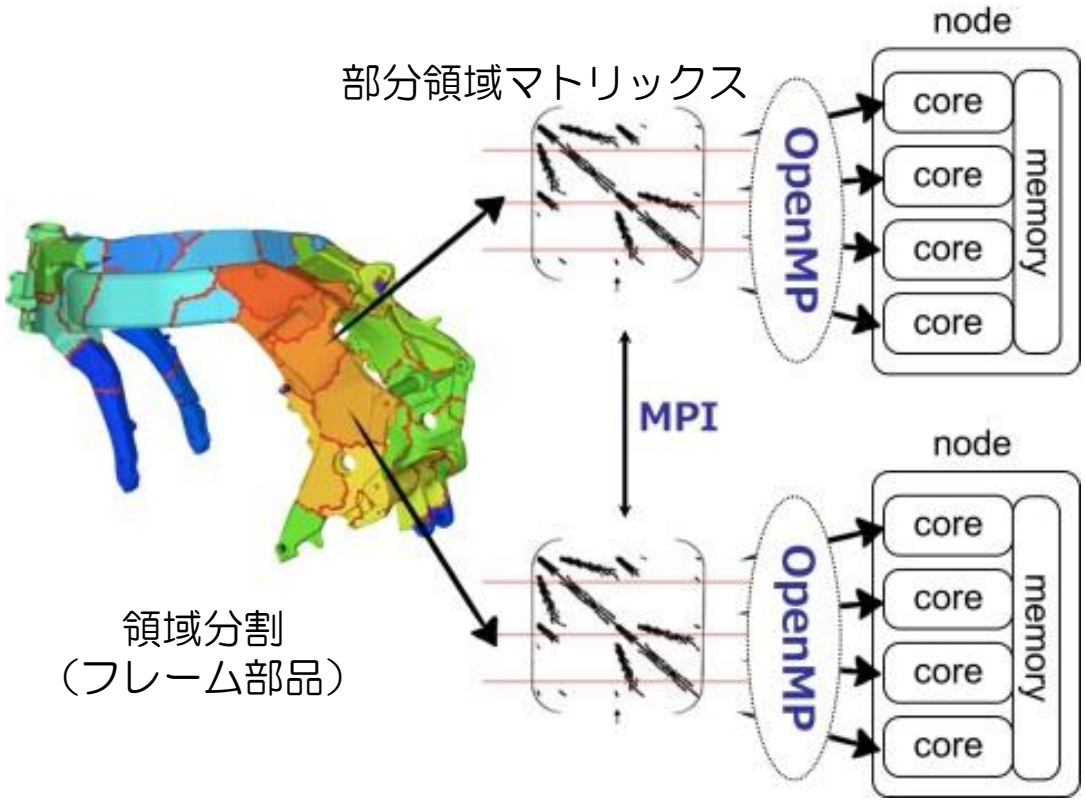
アプリケーションレベルにおける並列化の対象(2/2)

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

参考： FrontISTRにおける 並列計算のしくみ ＜ハイブリッド並列＞

注意：本日の話題は OpenMP並列 が中心です



MPI－OpenMP ハイブリッド並列

目次

- 導入
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FrontISTRにおける並列計算

- ノード間並列
  - 領域分割とMPI
- 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

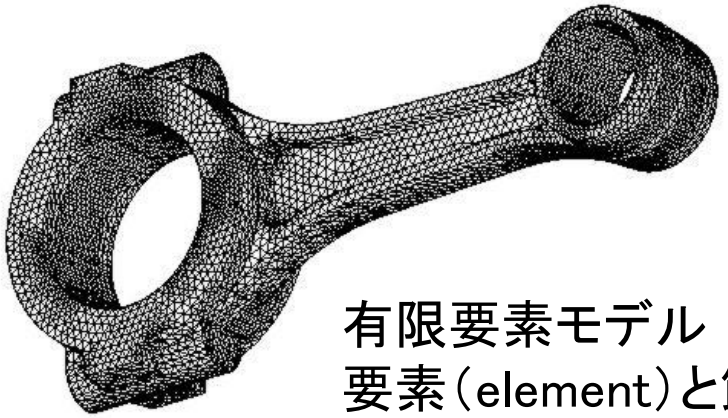
微分方程式をコンピュータで解くには？

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）

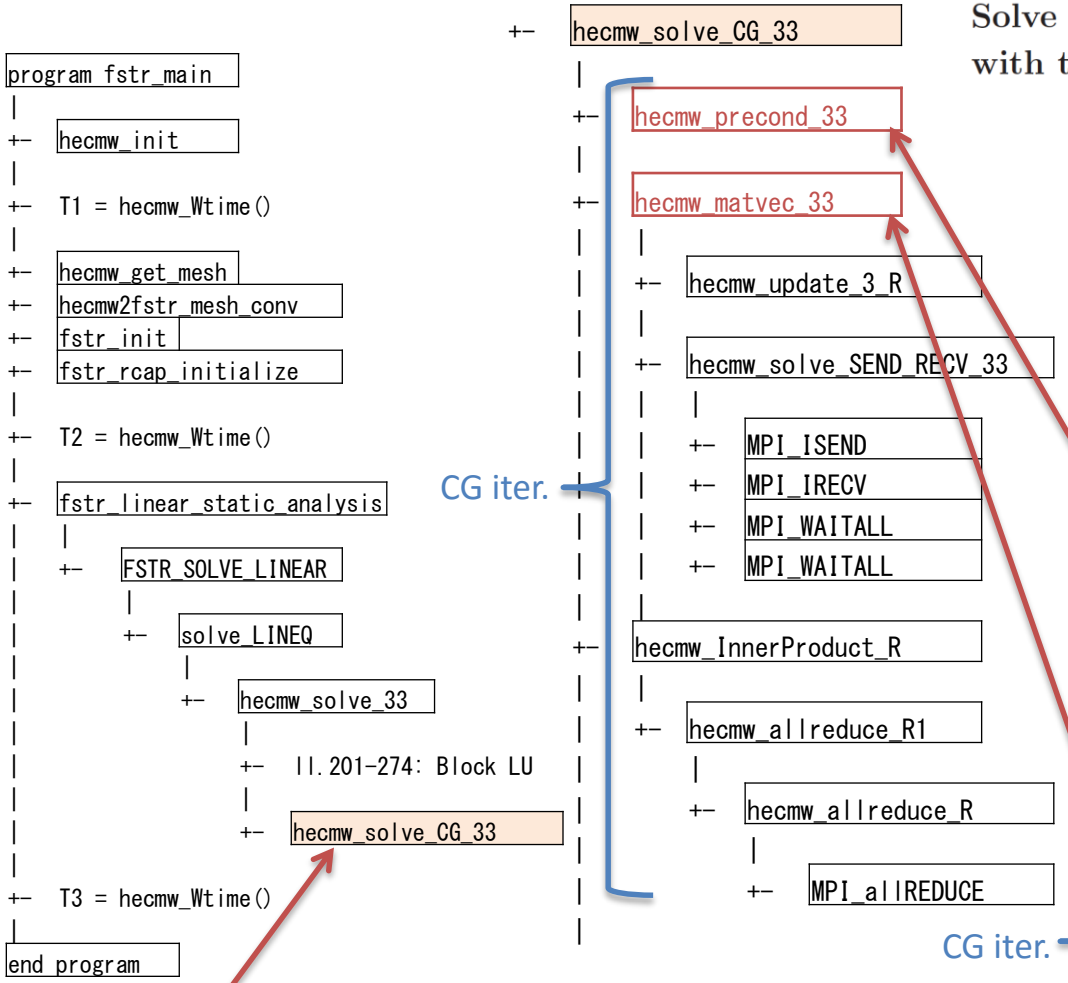
$$[\ K\ ]\{\ u\ \} = \{\ f\ \}$$

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

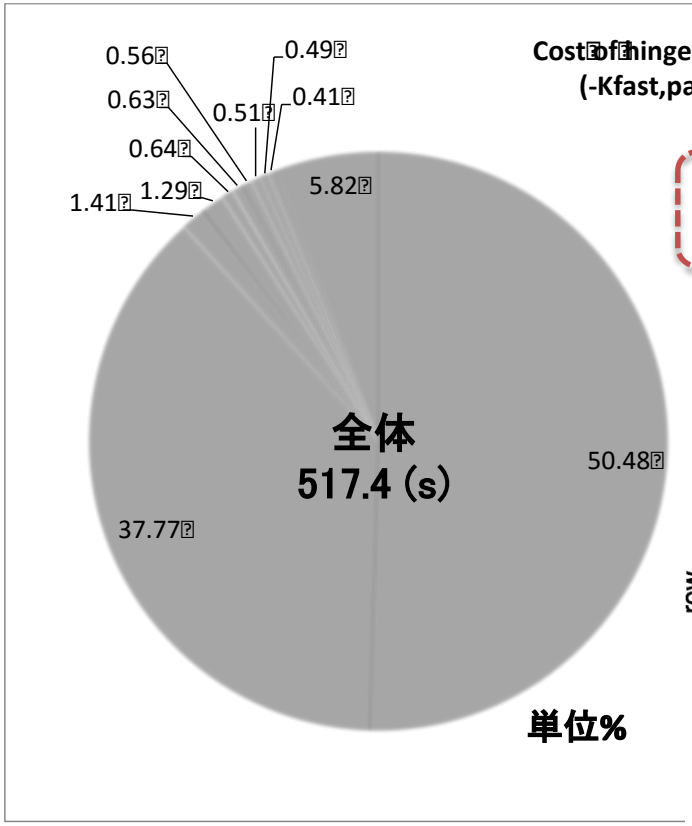


Solve $b = Ax$ by the Conjugate Gradient (CG) method with the left preconditioner:

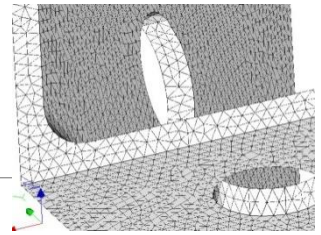
0. Choose the preconditioner M such that $M \simeq A$
   0-a. by Incomplete LU factorization, $M = LU$,
   0-b. by Diagonal block scaling,
   or others
1. Set $\boldsymbol{x} = \boldsymbol{x}_{\rm ini}$: initial guess, and $\boldsymbol{b}$: rhs vector.
2. Set MAXIT: maximum number of iterations, and 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

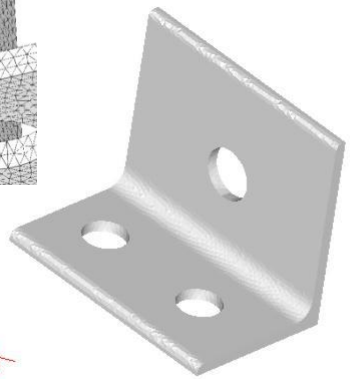
FrontISTRのプログラム構造

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。

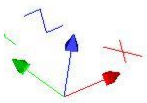


Cost of hingeS1 8 cores
(-Kfast,parallel)

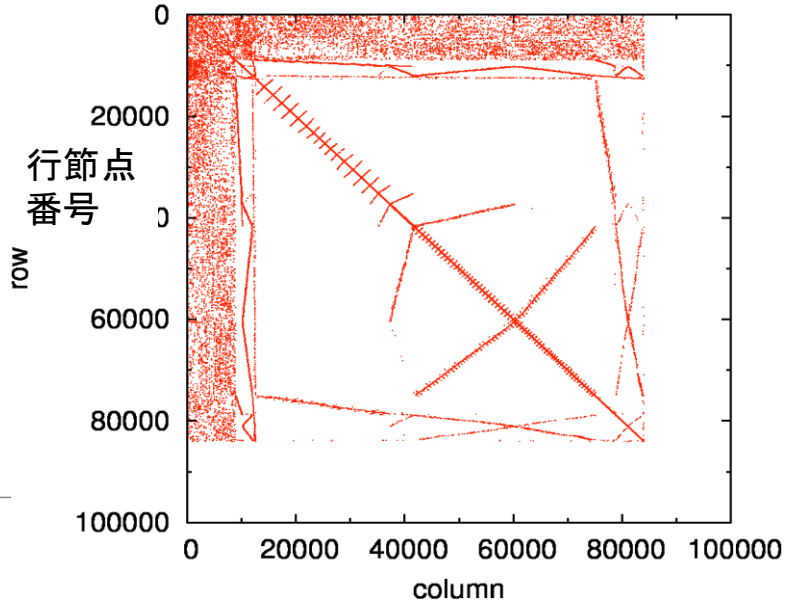




precond_33_&
matvec_33_&



hingeS1



行節点番号

列節点番号

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）

前処理付きCG法のアルゴリズム

compute $r^{(0)}= b - Ax^{(0)}$ for some initial guess $x^{(0)}$

for i= 1,2,...

  solve M $z^{(i-1)}= r^{(i-1)}$ (M: preconditioning matrix) — Preconditioning

  $\rho_{i-1}= r^{(i-1)T} z^{(i-1)}$ — Dot Product (1)

  if i=1

    $p^{(1)}= z^{(0)}$

  else

    $\beta_{i-1}= \rho_{i-1}/\rho_{i-2}$

    $p^{(i)}= z^{(i-1)} + \beta_{i-1} p^{(i-1)}$ — DAXPY (1)

  endif

  $q^{(i)}= A p^{(i)}$ — MATVEC

  $\alpha_i= \rho_{i-1}/(p^{(i)T} q^{(i)})$ — Dot Product (2)

  $x^{(i)}= x^{(i-1)} + \alpha_i p^{(i)}$ — DAXPY (2)

  $r^{(i)}= r^{(i-1)} - \alpha_i q^{(i)}$ — DAXPY (3)

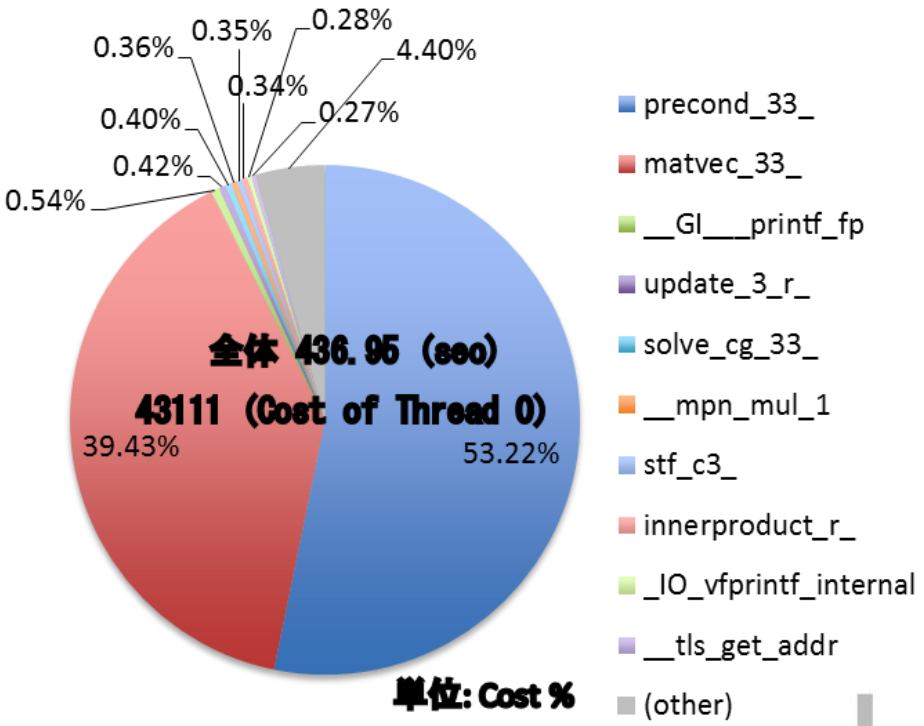
  check convergence; continue if necessary

end

FrontISTRにおける 並列計算のしくみ ＜領域分割に基づく並列FEM＞

- FEMの主な演算
  - 剛性マトリックスの作成 →部分領域（要素）ごとに並列処理可
  - 剛性行列の求解 {反復法ソルバー, 直接法ソルバー}

- 反復法ソルバー
  - 4種類の演算からなる
    - （1）疎行列・ベクトル積
    - （2）ベクトル・ベクトル内積
    - （3）ベクトルの加減（DAXPY）
    - （4）前処理



FrontISTRにおける 並列計算のしくみ ＜領域分割に基づく並列FEM＞

- 反復法ソルバーの並列処理
  - 4種類の演算からなる →通信しながら部分領域ごとに並列処理可
    - (1) 疎行列・ベクトル積
    - (2) ベクトル・ベクトル内積
    - (3) ベクトル(およびその実数倍)の加減(DAXPY)　通信不要
    - (4) 前処理

目次

- 導入
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FrontISTRにおける並列計算
- 実効性能について

- ノード間並列
  - 領域分割とMPI
- ノード内並列（単体性能）
  - ループ分割とOpenMP

今回はこの話が中心です

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。

領域分割

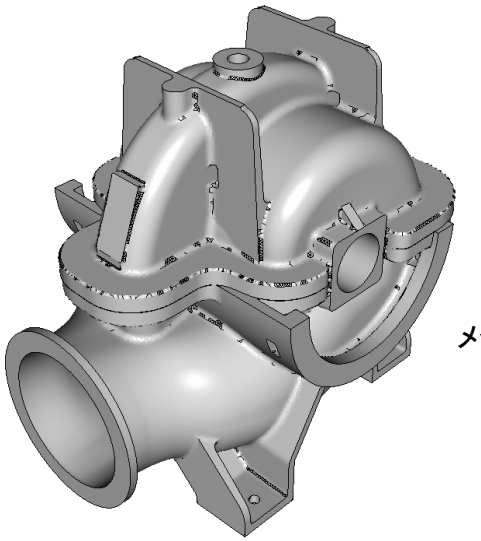
部分領域への分割

- 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。

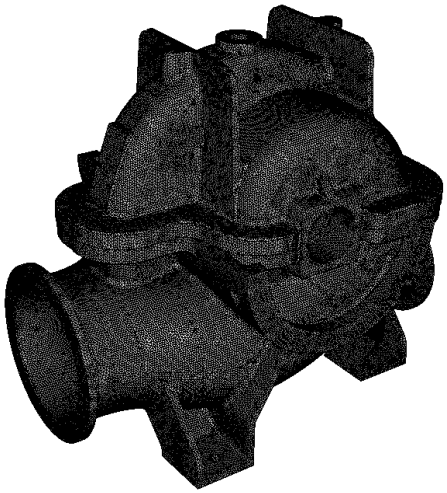
通信テーブル

- 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

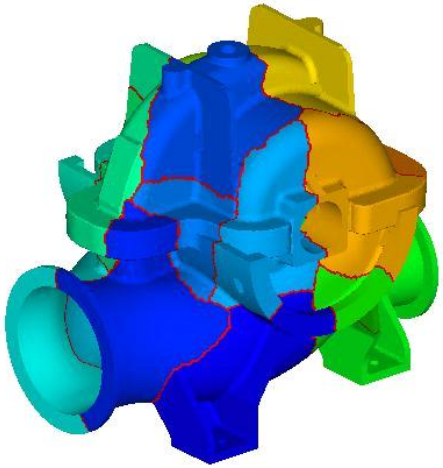
部分領域ごとに行列ベクトル積を実行する．全体領域での行列ベクトル積と同じ結果になるように、部分領域間で通信する．



メッシュ分割

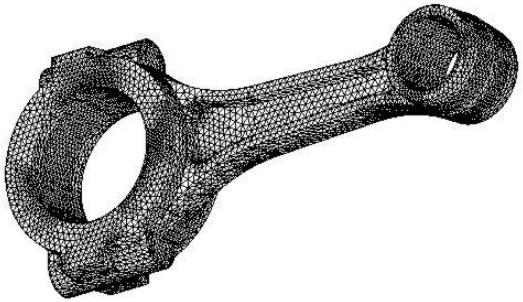


領域分割

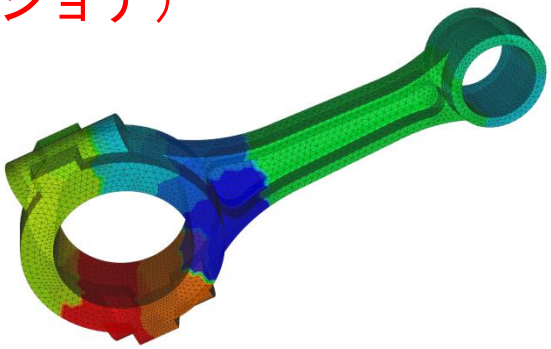


領域分割ツール
（パーティショナ）

CADモデル

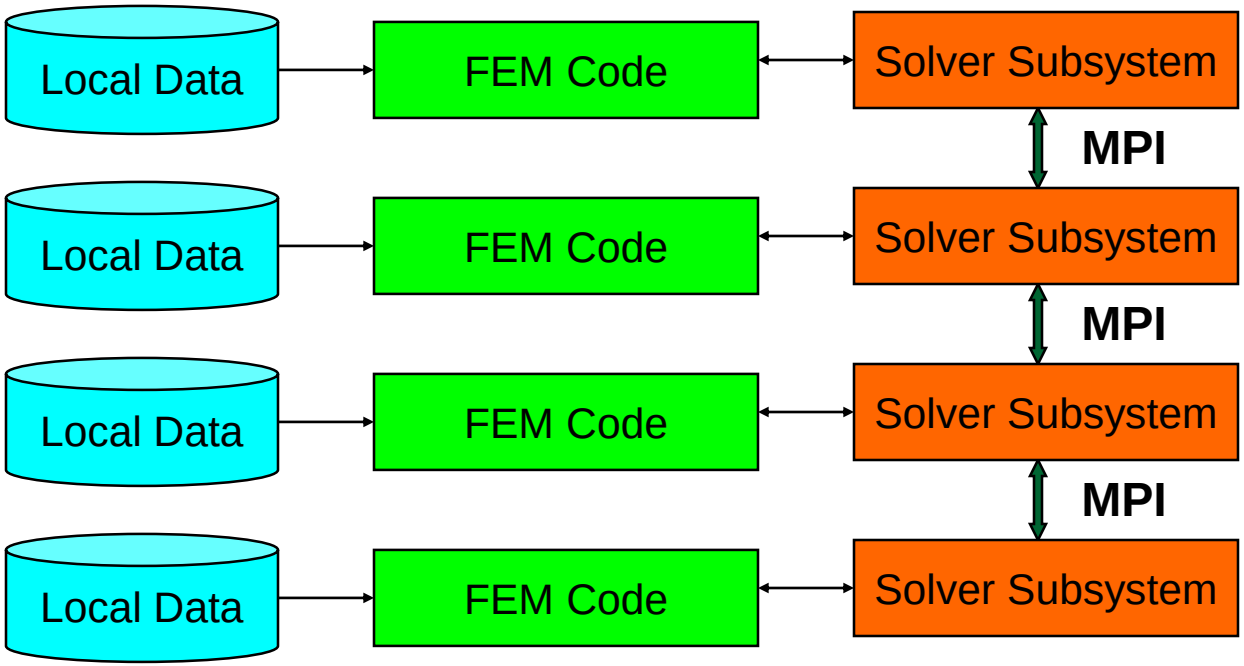


有限要素モデル
要素（element）と節点（node）



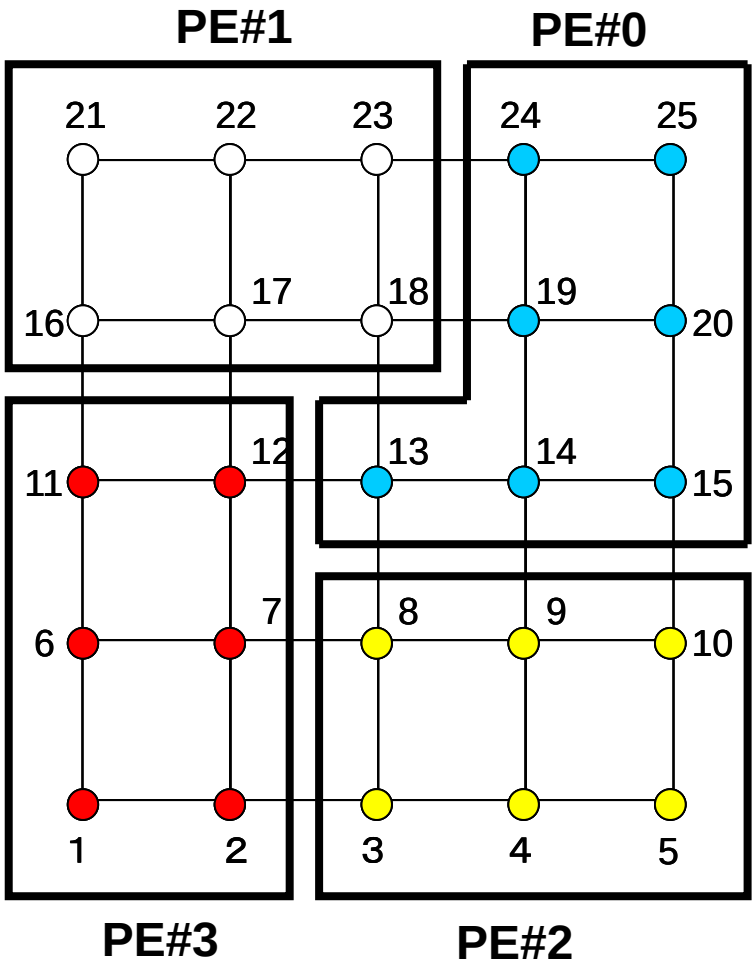
SPMD Programming Style

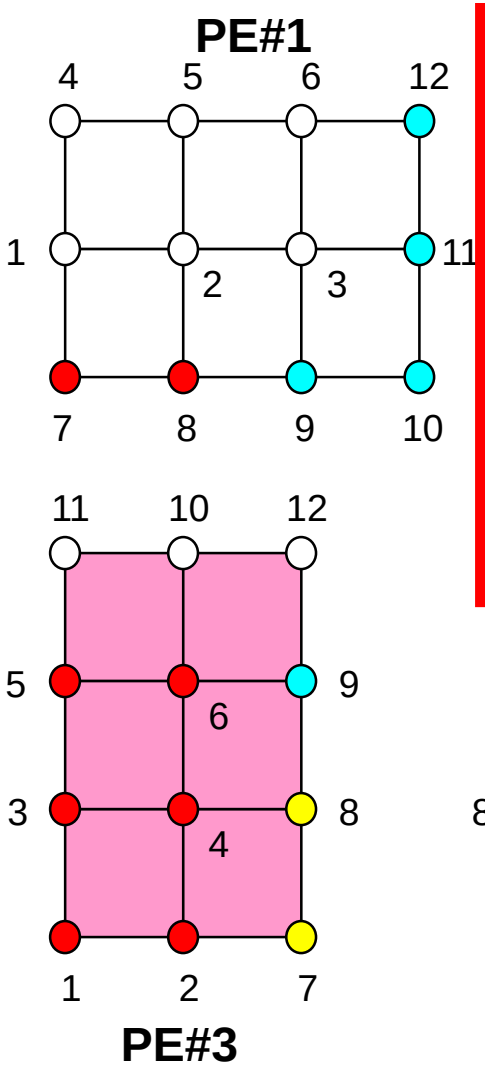
- Large file handling → Local distributed data
- FE analysis modules just consider local operation (element matrix assemble)
- Global operation occurs only in linear solver.

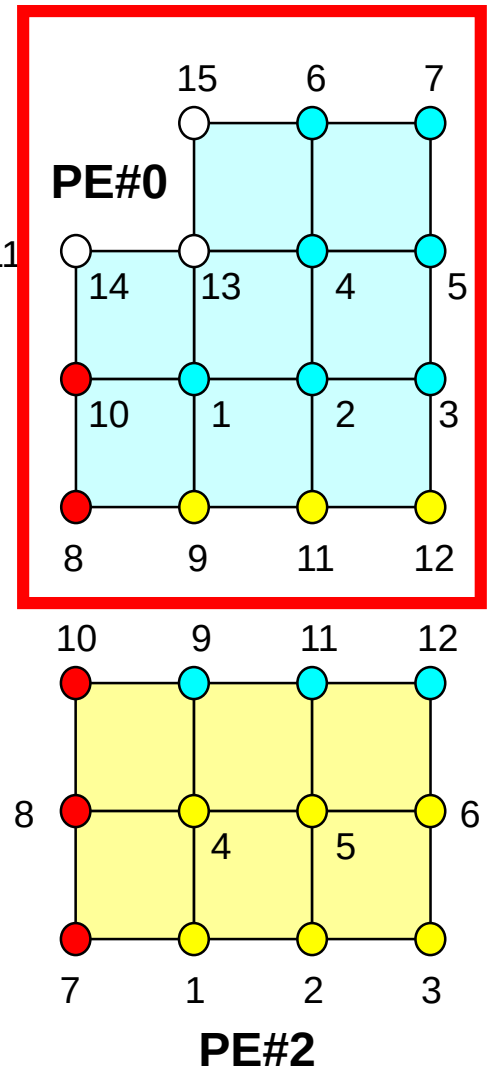


Local Data Structure

Node-based Partitioning
internal nodes - elements - external nodes







<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
- なぜ並列化か？
  - 並列アーキテクチャ、並列プログラミング
- FrontISTRにおける並列計算
- 実効性能について

- ノード間並列
  - 領域分割とMPI
- ノード内並列（単体性能）
  - ループ分割とOpenMP

今回はこの話が中心です

並列プログラミング方法（2/3）

並列化指示文

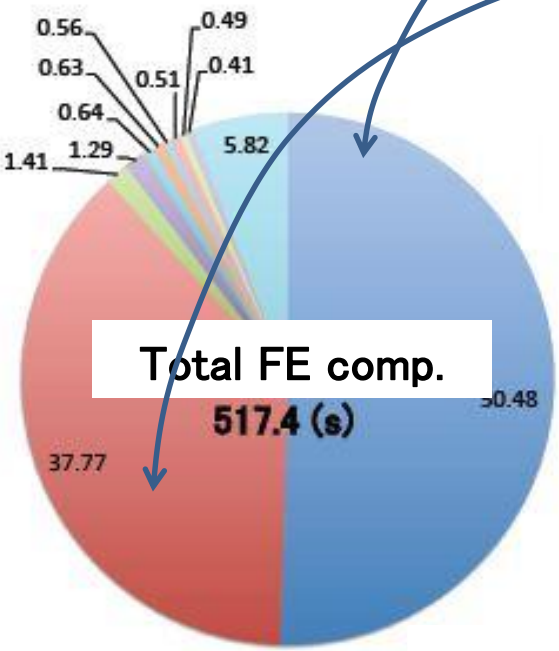
- ワークシェアリングやデータシェアリングのための指示文（ディレクティブ）をプログラムの中に挿入する。
- 主に、共有メモリの並列計算機システム。共有・分散メモリのノード内並列化も同様。
- 指示文はプログラム中で見かけ上コメント行のように書かれるが、コンパイル時にオプションを指定することによって、その指示文が解釈されるようになる。
- ポータビリティ（可搬性、移植性）を考慮して指示文を統一化した規格がOpenMP, OpenCL, OpenACCnなど
- ベクトルプロセッサの場合、ベクトルベクトル計算に関する指定も指示文の挿入によって行われる。

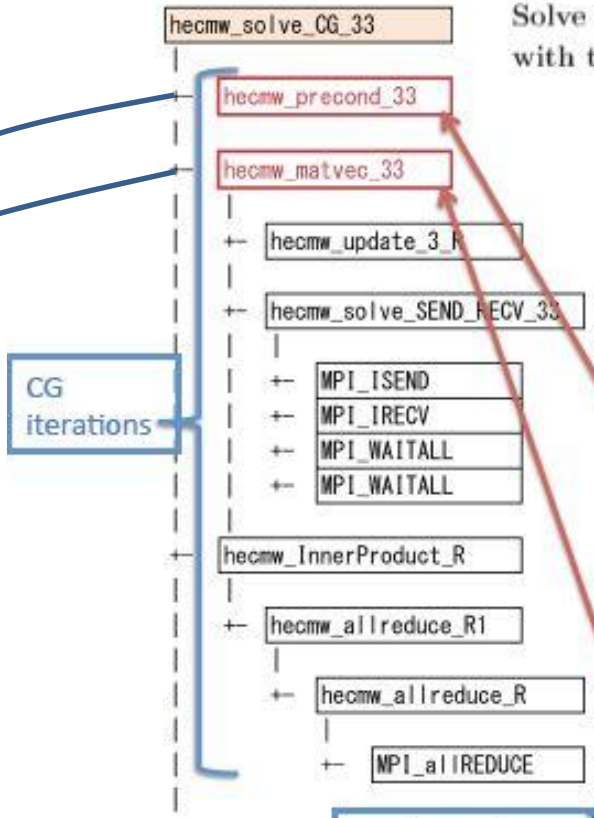
```
SUBROUTINE DAXPY(Z,A,X,Y)
INTEGER  I
DOUBLE PRECISION  Z(1000), A, X(1000), Y

!$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。





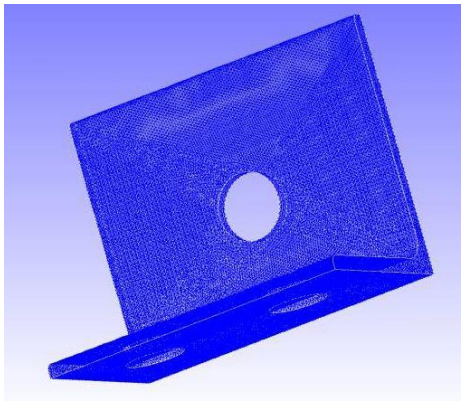
Solve $b = Ax$ by the Conjugate Gradient (CG) method with the left preconditioner:

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. $\quad \boldsymbol{z} = M^{-1}\boldsymbol{r}$
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. $\quad \boldsymbol{q} = A\boldsymbol{p}$
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

CG iterations

Acknowledgements:
Research Organization for Information Science and Technology, RIKEN AICS

Example ‘hinge’



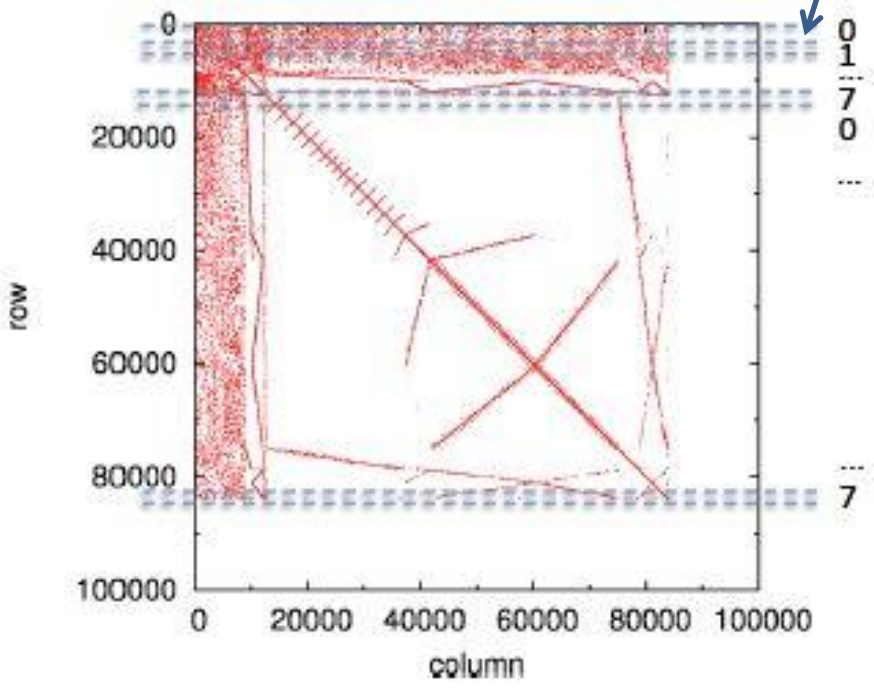
- 252,168 DOFs
- 2,115,968 non-zeros
- Density of non-zero : 0.03%

Simple cyclic

```
!$omp do schedule(static,1)
    do i=1,N
```

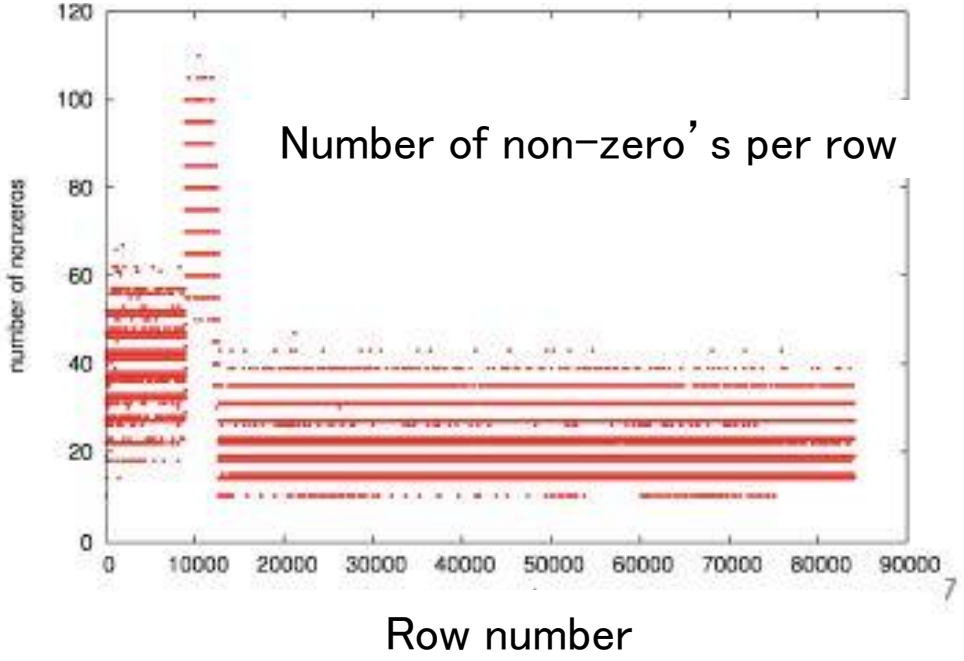
Simple block cyclic

```
!$omp do schedule(static,100)
    do i=1,N
```

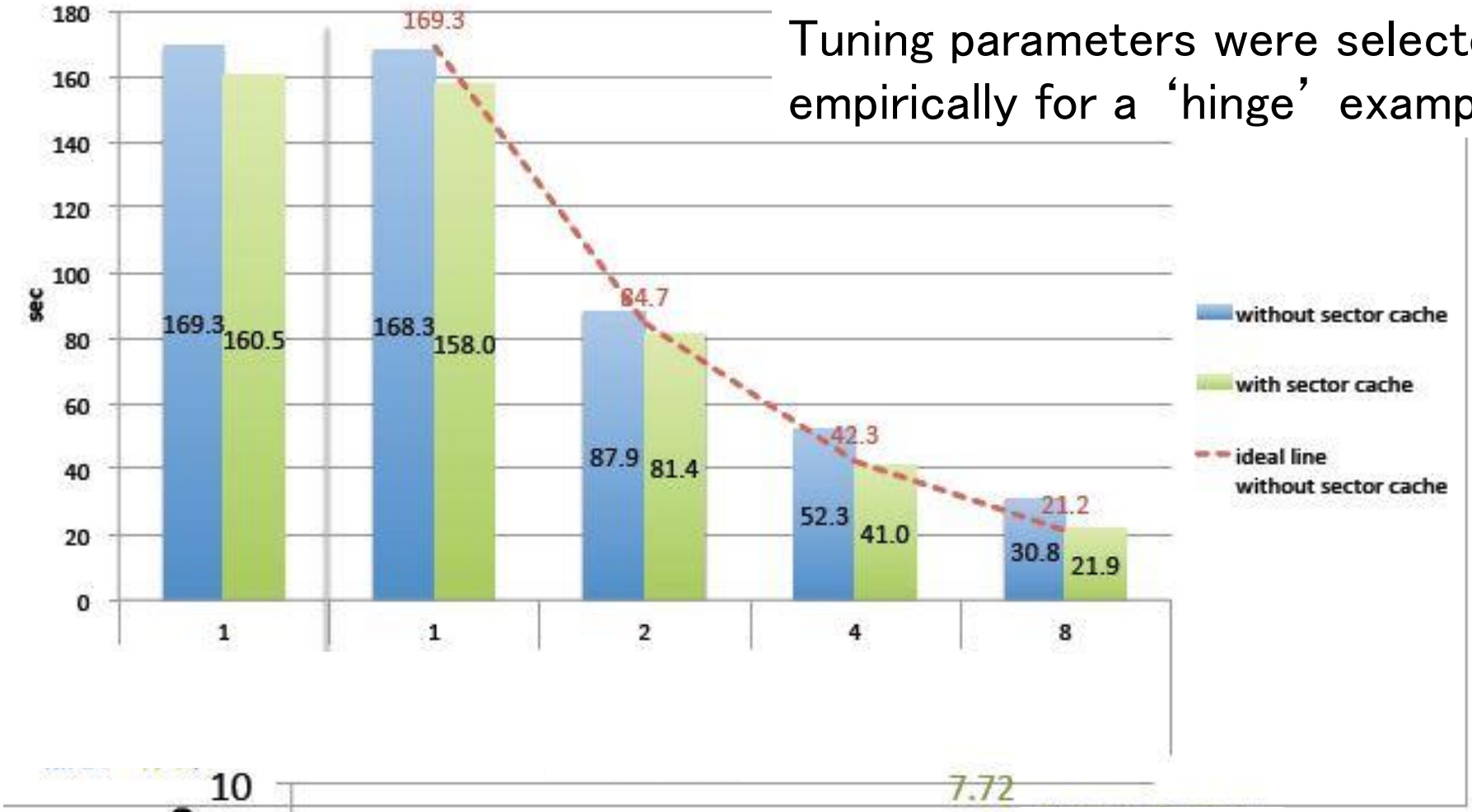


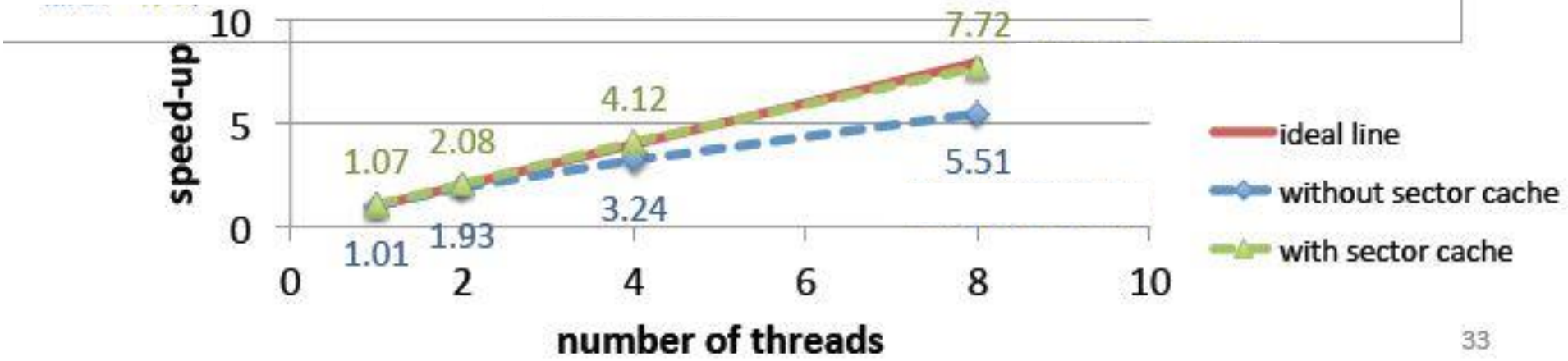
Distribute non-zeros among threads

```
threadNum = omp_get_thread_num()
do i= startPos(threadNum), endPos(threadNum)
```

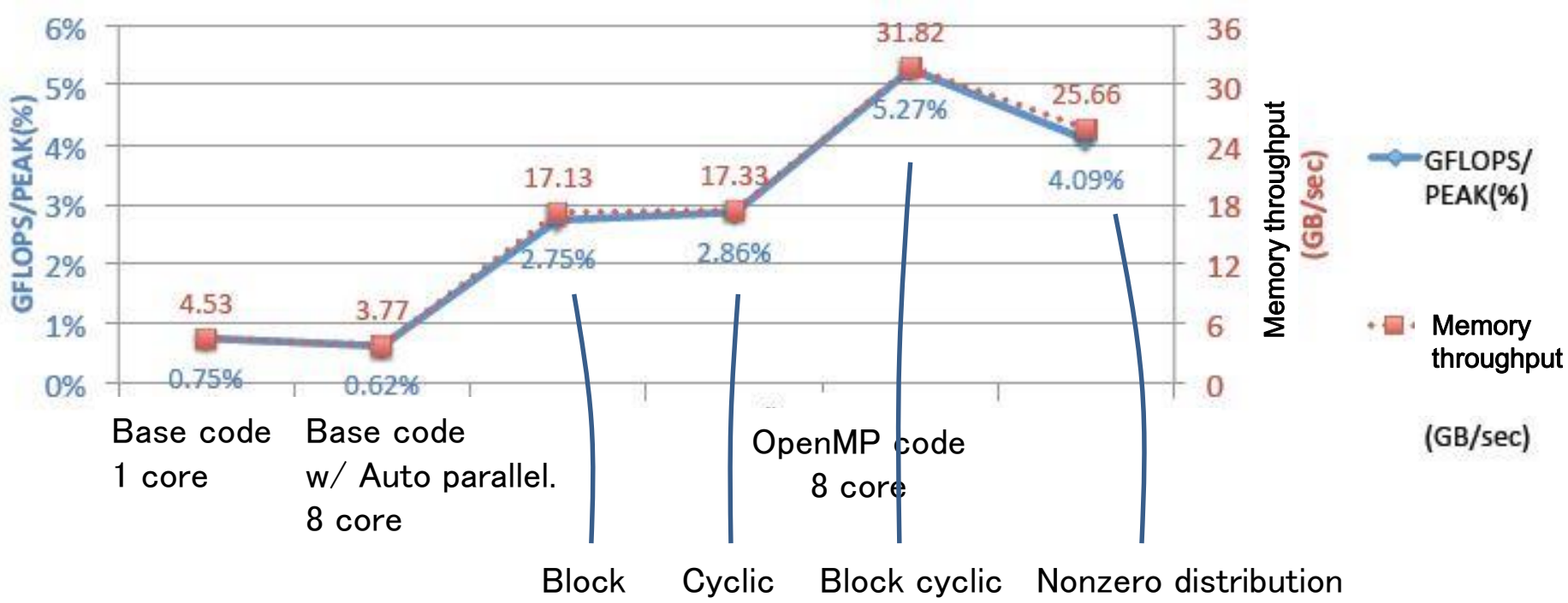


Tuning parameters were selected empirically for a ‘hinge’ example.



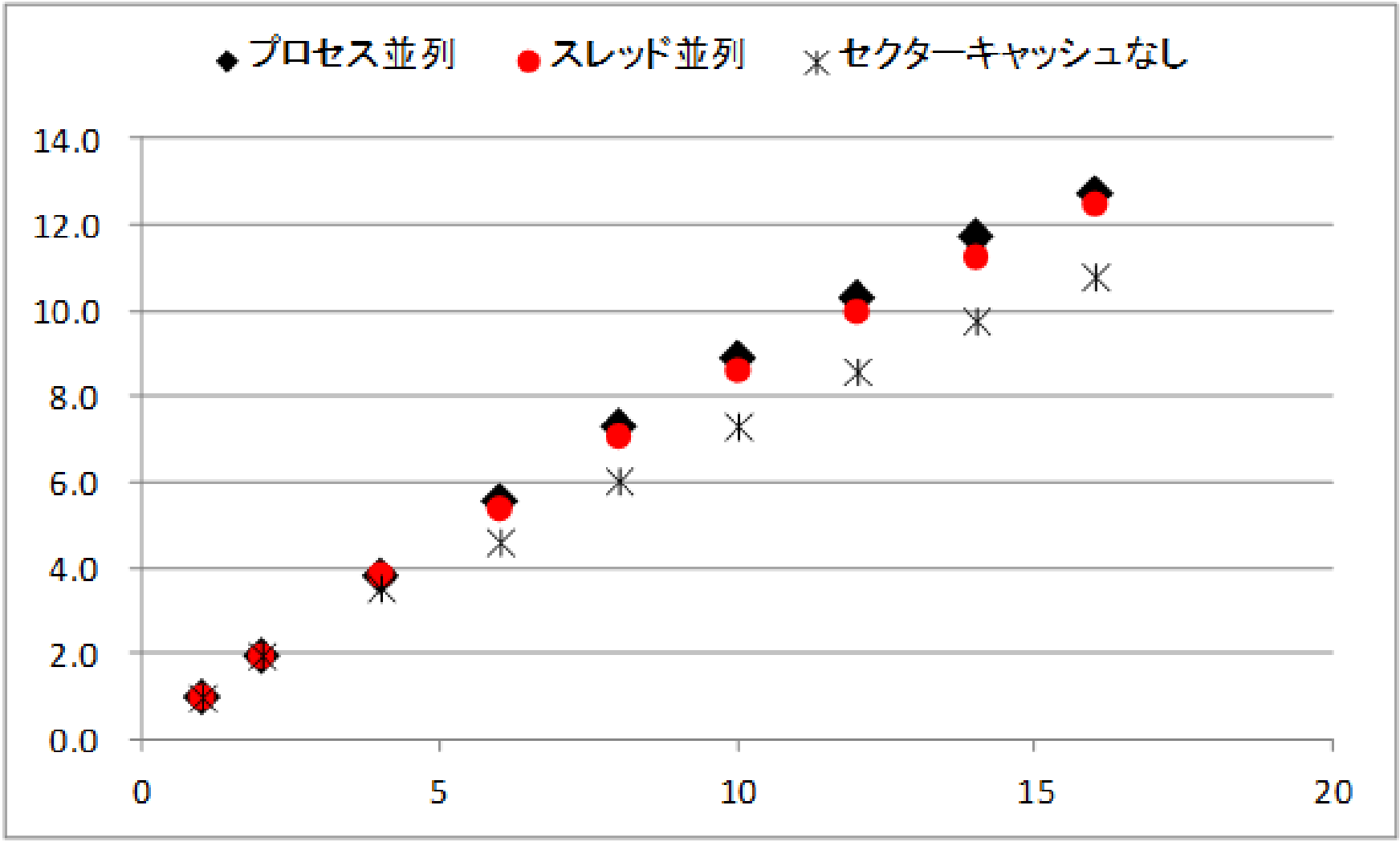


Observed memory wall feature



Acknowledgements:
Research Organization for Information Science and Technology, RIKEN AICS

（「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。

precond_33 のOpenMP並列化

matvec_33 とは違って、precond_33 は「依存性」のあるアルゴリズム。

並列計算の際には、リオーダリング（番号付替え）によってあらかじめ依存性を排除する必要がある。

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)

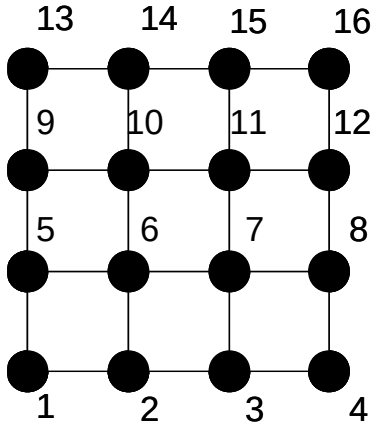
例えば、次式のような演算を考える。

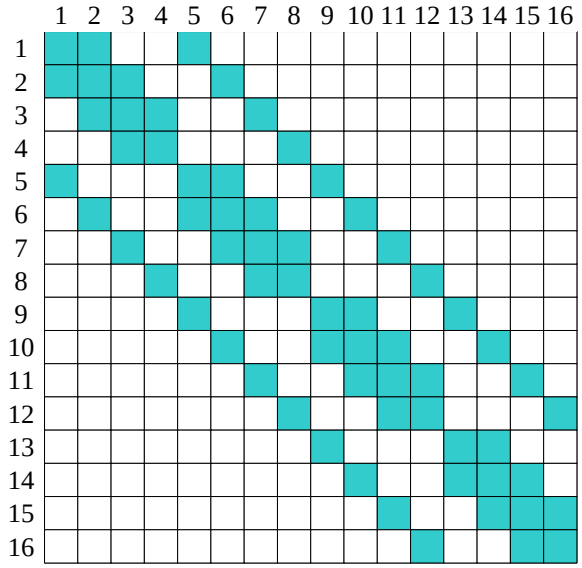
$$x_i = y_i - \sum_{k=1}^{i-1} L_{ik} x_k$$

- 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。

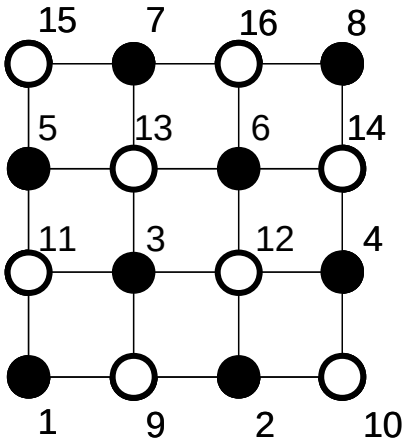
- オーダリング前の節点番号付けの場合、節点iの演算の際にそれ以前に演算済みの情報が必要であることがわかる。
- 依存性のない節点を2色に色分けて(黒と白)番号を付け替えた場合、同一色に属する節点に関する演算は互いに依存性がないことがわかる。すなわち、ノード内並列処理やベクトル処理が可能となる。

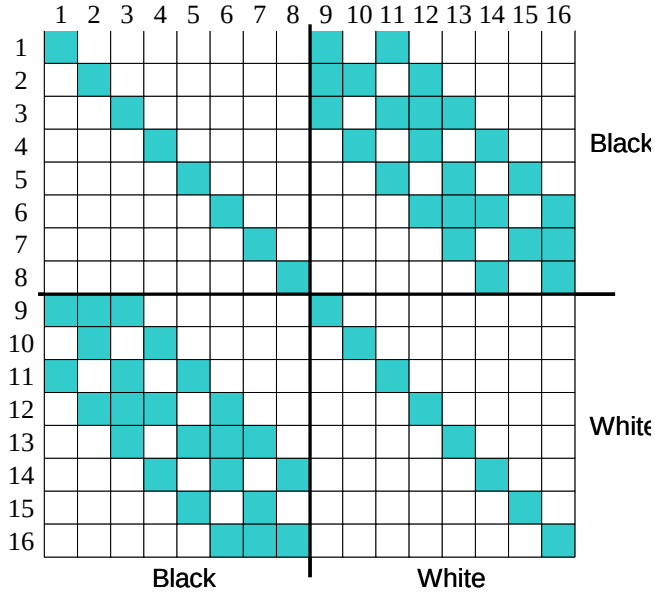
- red and black法、マルチカラー法
- 演算に依存性のない節点をハイパープレーンと呼ばれるグループに分類し、各ハイパープレーン上の節点についてノード内並列処理やベクトル処理を行うことも多い。





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