並列・マルチコア・GPU

2013年10月18日（金）
第8回FrontISTR研究会
神戸大学
システム情報学研究科棟

- なぜ並列？
- 並列アーキテクチャ
- 並列プログラミング
- 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

奥田洋司

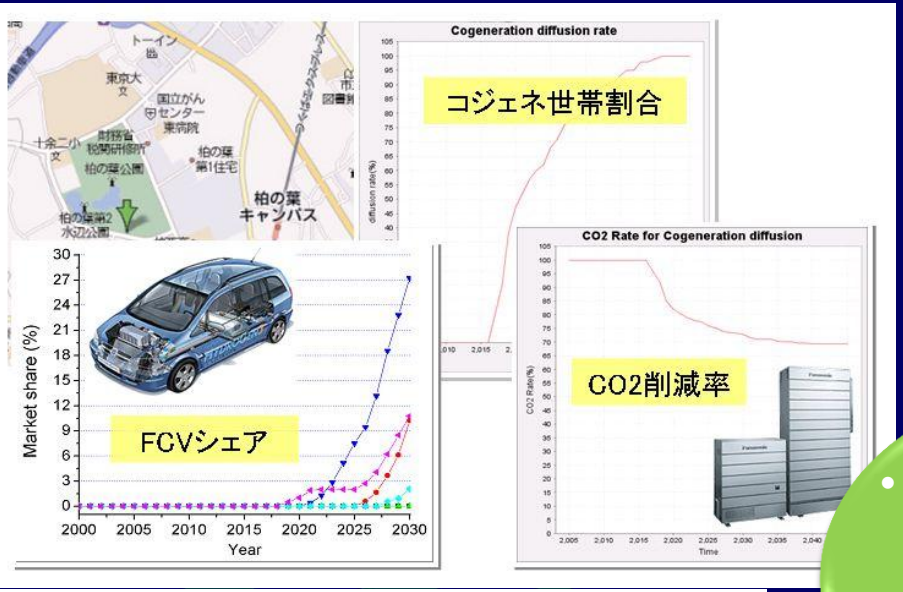
okuda@k.u-tokyo.ac.jp

東京大学

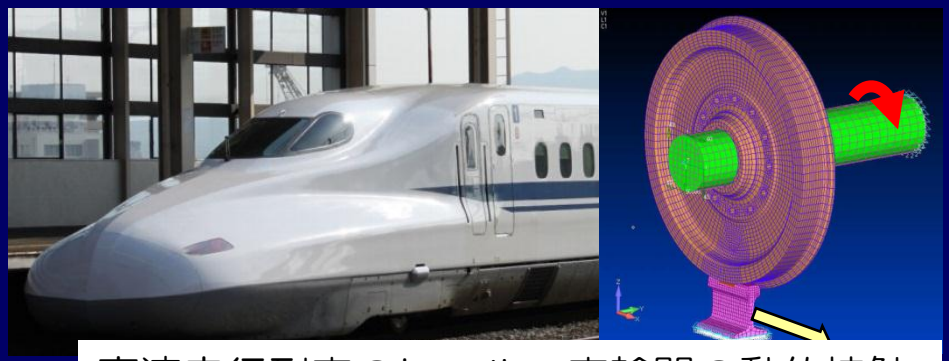
新領域創成科学研究科・人間環境学専攻

マルチシナリオシミュレーション環境学分野

ー計算科学を駆使した人工物シミュレータに関する研究ー



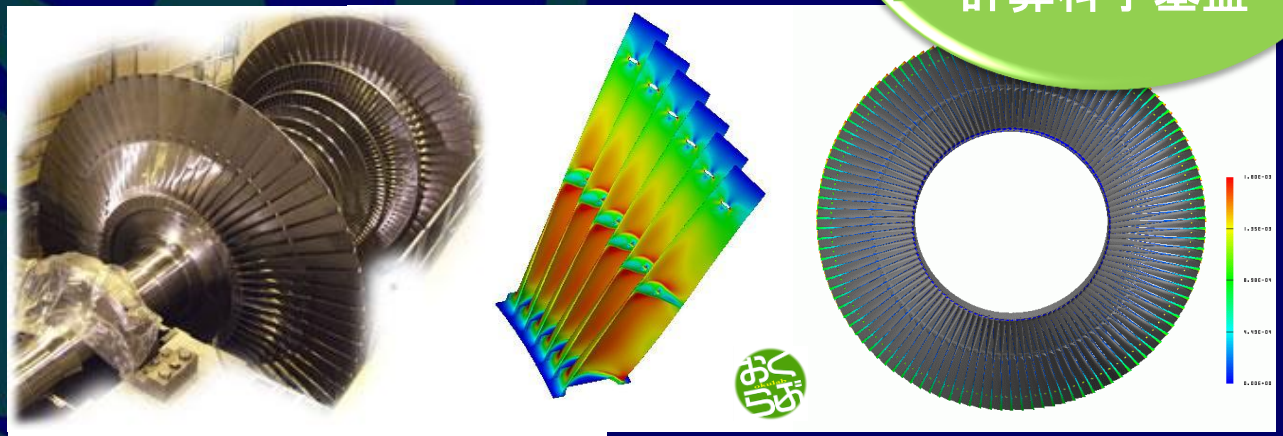
水素社会構築過程のシミュレーション



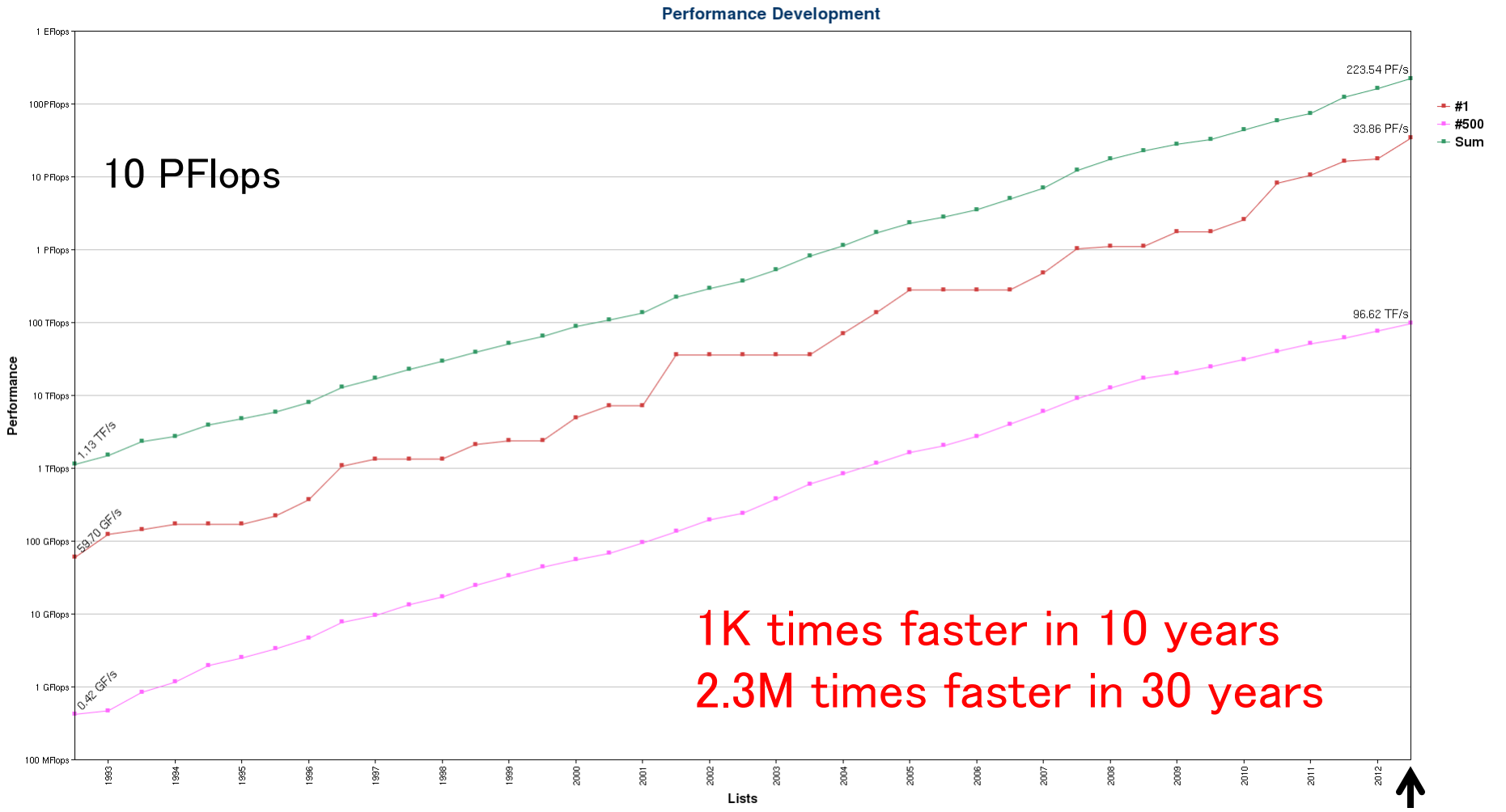
高速走行列車のレール・車輪間の動的接触

- 人工物の導入シナリオ
- 機能予測
- 計算科学基盤

GPUクラスタ計算機



高性能タービンブレード設計



1K times faster in 10 years

2.3M times faster in 30 years

Jun.1,2013

http://www.top500.org/statistics/perfdevel/

並列処理の重要性

- 計算機ハードウェアの性能向上　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
```

インデックスJについて独立に計算できる

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

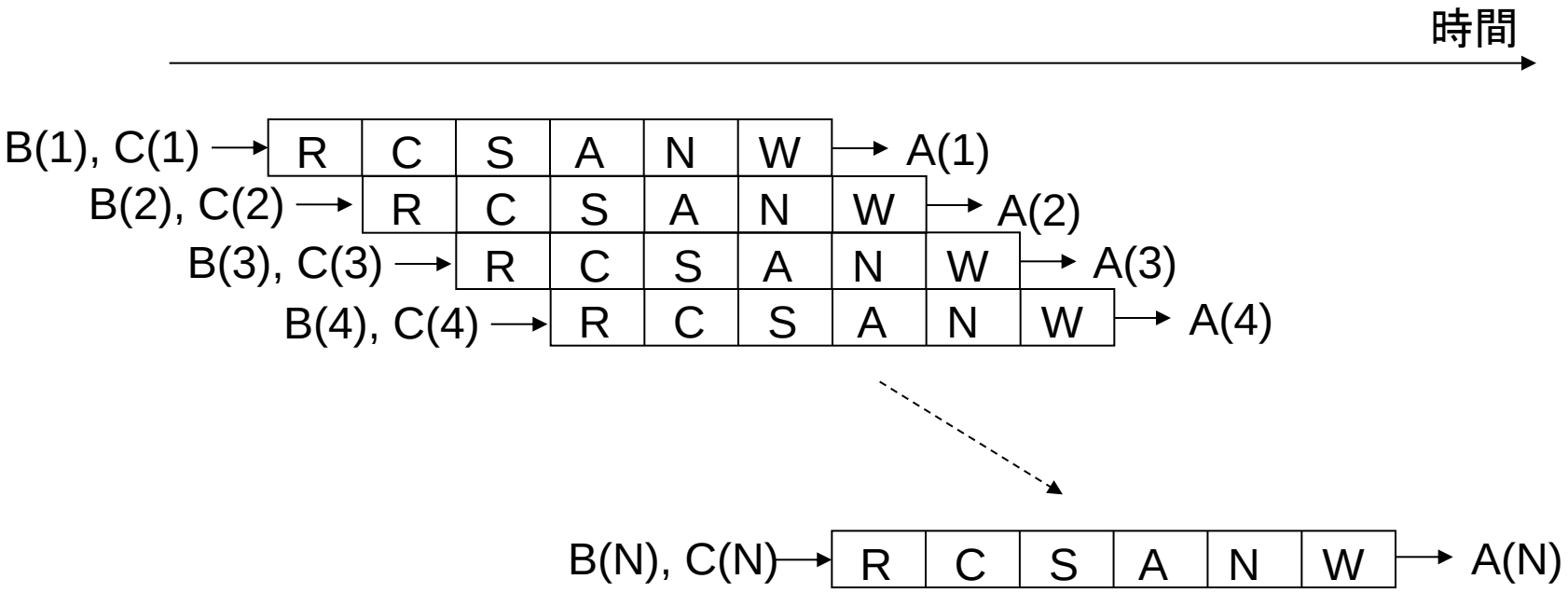
並列アーキテクチャ(1/3)

ベクトル処理　　地球シミュレータ、SIMD

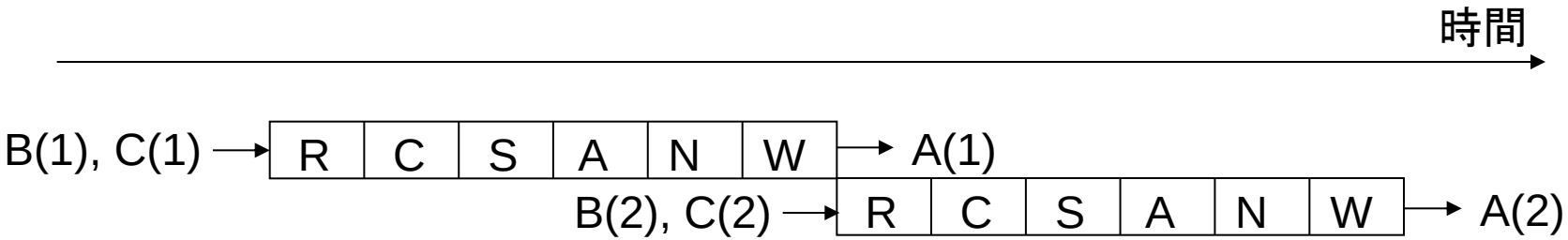
⇔スカラ処理

- 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

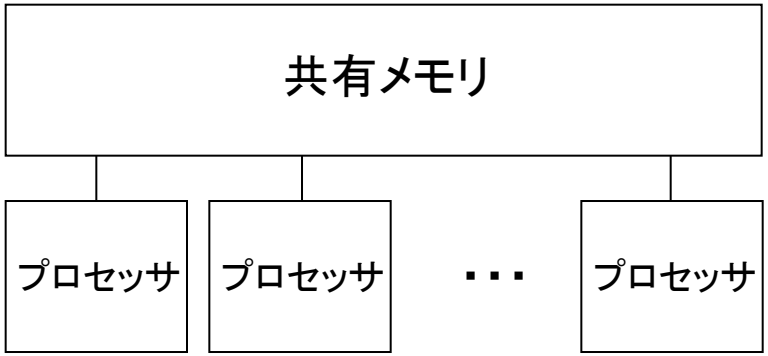
並列アーキテクチャ(2/3)

マルチプロセッサ・マルチコア

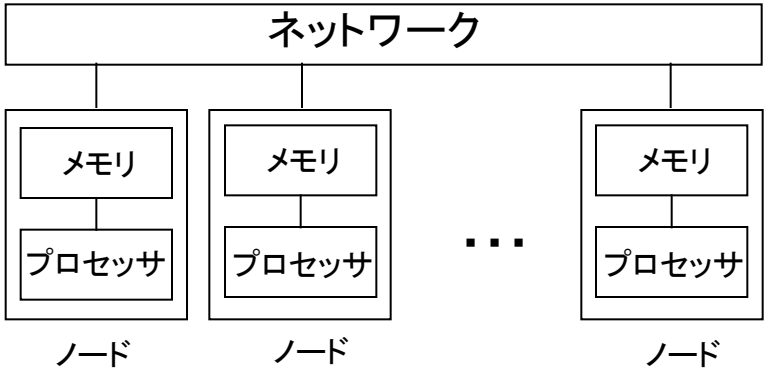
並列(パラレル)⇔逐次(シリアル)

- アーキテクチャ面での並列化レベルをもう一段階上げ、プロセッサを並べる
- 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
  - 共有メモリ型(SMP(Symmetrical Multi Processor))
  - 分散メモリ型
  - 共有・分散メモリ型

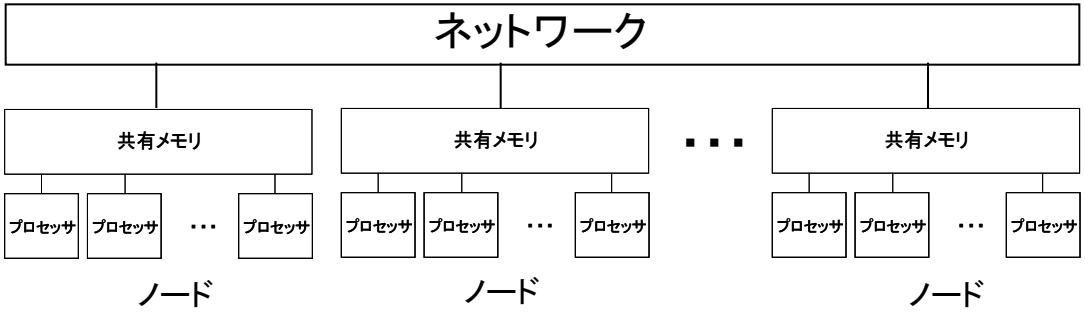
コモディティプロセッサを比較的安価で低速なLANでネットワーク結合したものを一般にクラスタ計算機あるいはPCクラスタと呼び、並列計算機として多く用いられている。



(a) 共有メモリ型



(b) 分散メモリ型



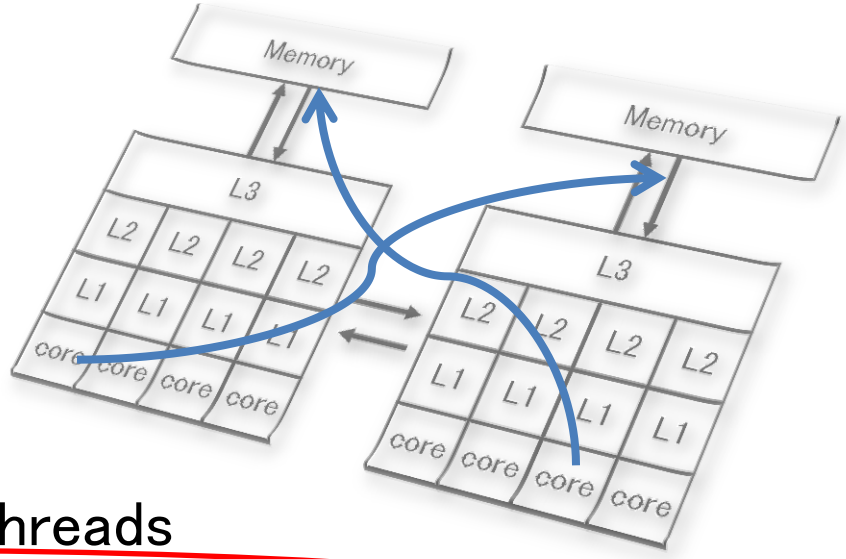
(c) 共有・分散メモリ型

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 and memory, vs. Flop/Byte

The “K-computer”

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

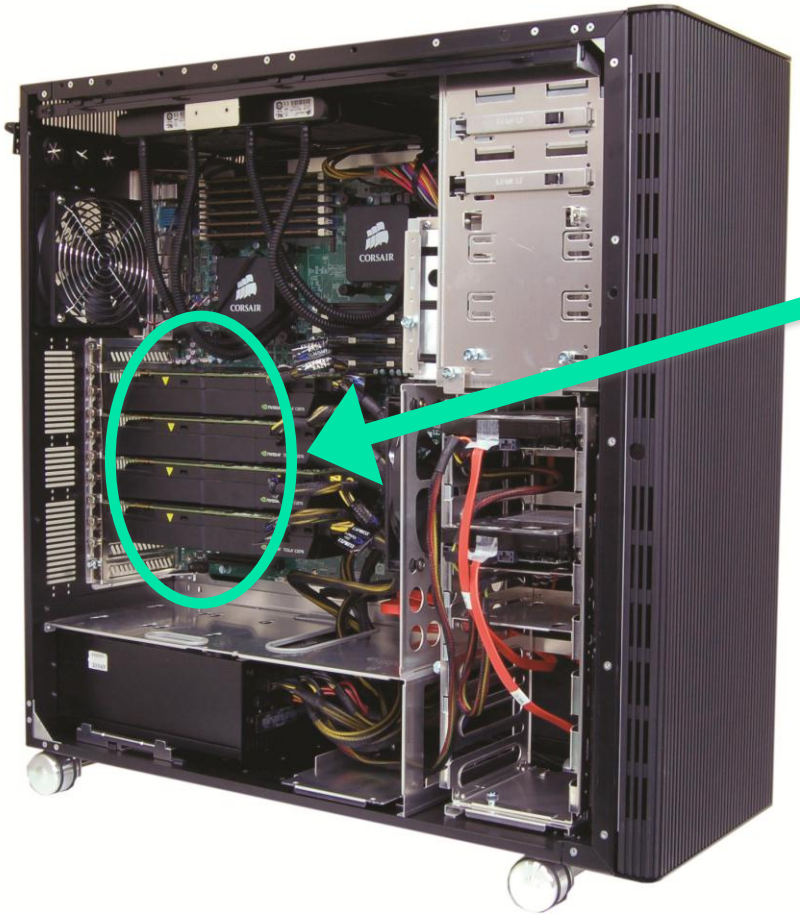
- Parallelism
  - Inter-node (node⇔node) : MPI
  - Intra-node (core⇔core) : OpenMP
  - Flat MPI is NOT recommended

- Hybrid programming is crucial for “K”.

GPUを使って計算する

（2012年8月作成のスライド）

PC



GPU ボード

- 大きさに注意
- 6ピン と 8ピン 電源ケーブル
- 電力: 1枚 250W 程度

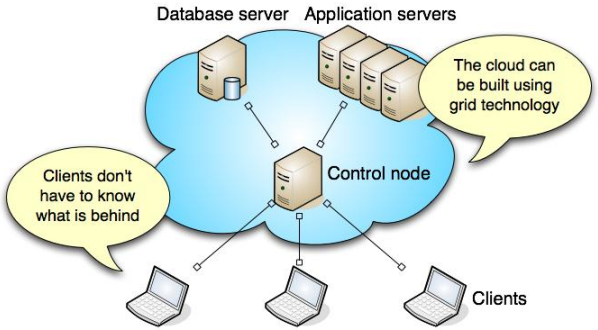
GPUを使って計算する

（2012年8月作成のスライド）

GPUのトレンド

	Graphics用 （倍精度 200Gflops程度）	HPC用 （倍精度 500Gflops程度）
NVIDIA 社	 GeForce （シェア 60%）	 Tesla （13台/Top100）
AMD 社	 FirePro （シェア 40%）	 Radeon （1台/Top100）

並列アーキテクチャ(3/3)

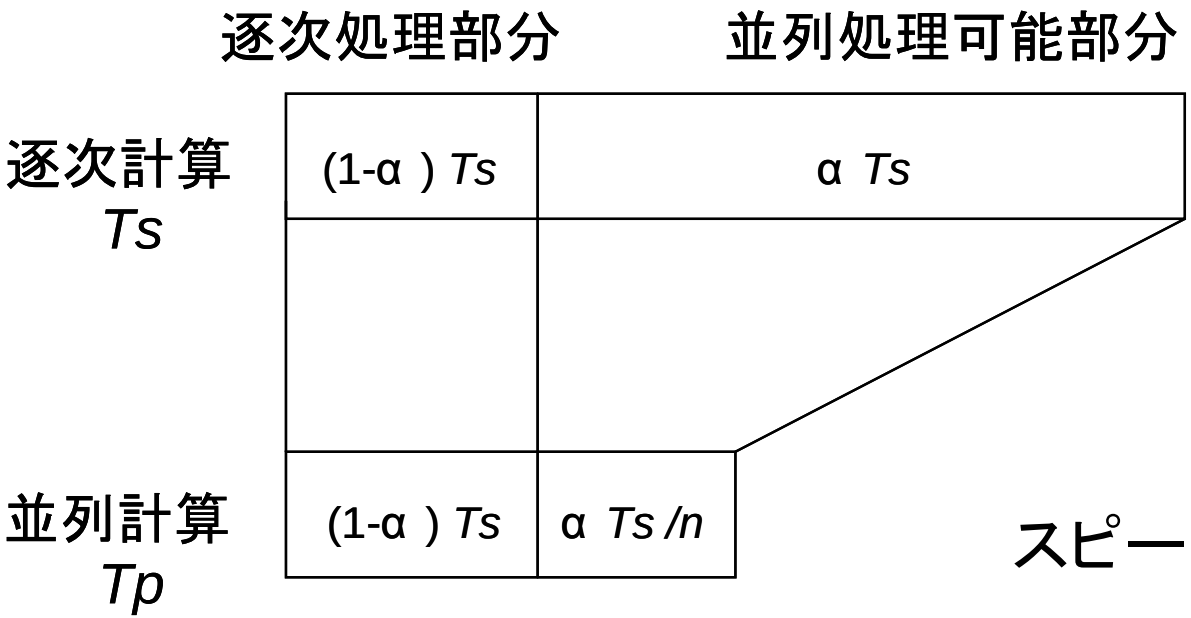


- グリッド
  - Power Grid（電力網）とのアナロジー
  - 多様なOS、計算機、ネットワーク、データベース等を統一的なツールやインターフェイスで取り扱うための、さまざまなミドルウェアやプロトコル。 例）Globus、Ninfなど
  - グリッドの利用形態

    メタコンピューティング／データ・インテンシブ・コンピューティング／ボランティア・コンピューティング／テレ・イマージョン／グリッドASP
- 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）
  - サービス内部の実装にグリッド技術が利用可能
  - コンピュータ、ネットワーク、WebService、データベース技術の高度化

計算機を並べただけでは速くならない

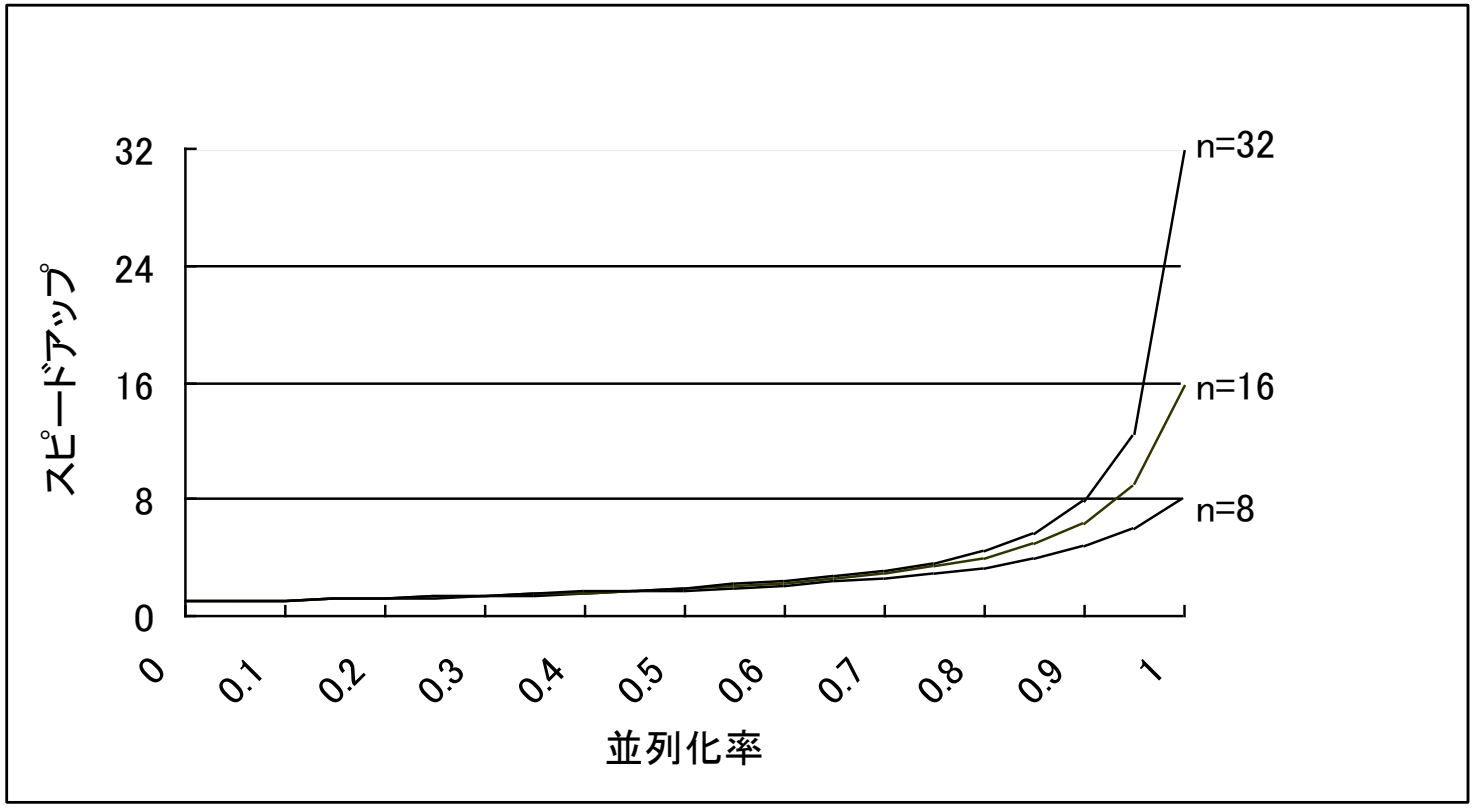
- アムダールの法則
  - プログラムの並列化率と並列化による実効性能向上の関係

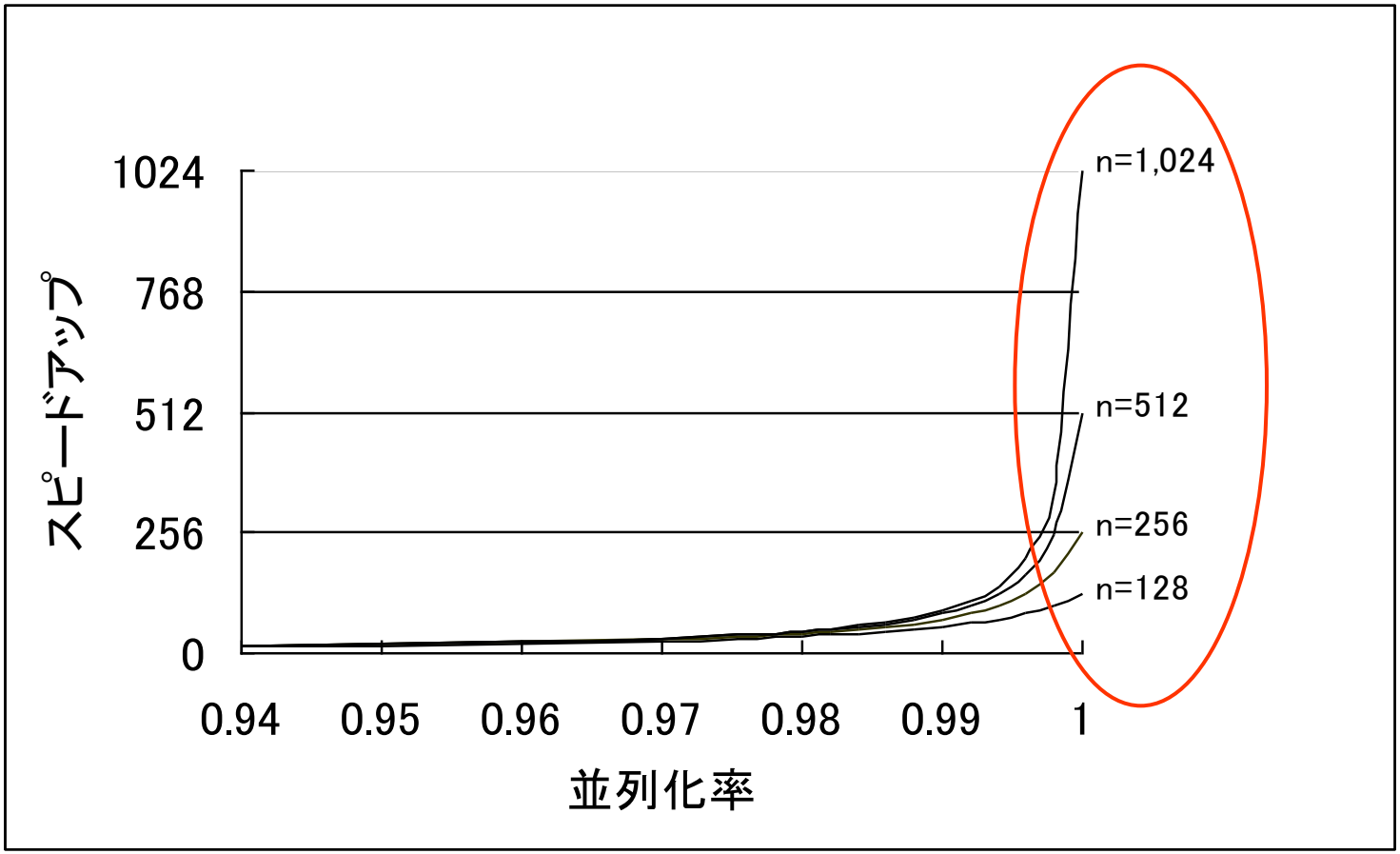


スピードアップ（$= Ts / Tp$）

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。

2013/10/18 はここまで

- なぜ並列？
- 並列アーキテクチャ
- **並列プログラミング**
- 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

アプリケーションレベルにおける並列化

- 自動並列化は可能？

  最初は逐次処理のプログラム

  → コンパイラによって各種ハードウェア環境に適応したロードモジュールに変換

  - コンパイラが並列ハードウェア環境の違いや並列処理をFortranやCなどの高級言語から完全に隠蔽し、高い効率で並列実行できるようなロードモジュールを生成するということはない。
  - アプリケーションプログラムのレベルにおいて、何らかの並列化を意識した手続きを加えなければならない。

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DO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
  - 分散配置されたノードのメモリ応じてデータを分散し、ノードごとに異なるデータに対して同じ演算を施す

  など

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

  メモリ分散方式の違いやプロセッサタイプの違いなどに応じて、いくつかの様式がある。

  - 通信ライブラリ(MPI などのメッセージパッシングライブラリ)をアプリケーションプログラムとリンクして用いる。
  - アプリケーションプログラムの中に並列化の指示文(OpenMPなど)やベクトル化の指示文を挿入する。
  - 並列言語(HPFなど)を用いる。

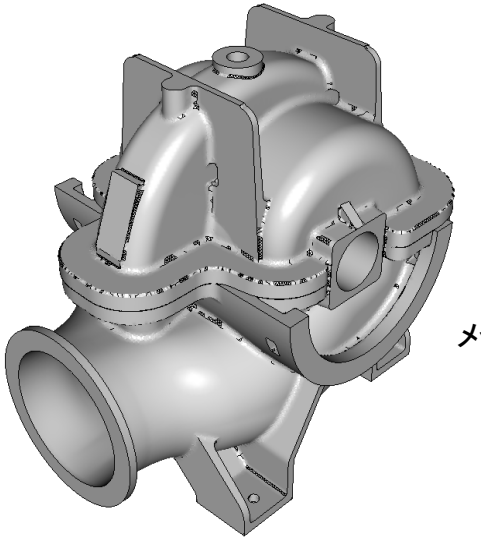
    など

並列プログラミング方法（1/3）

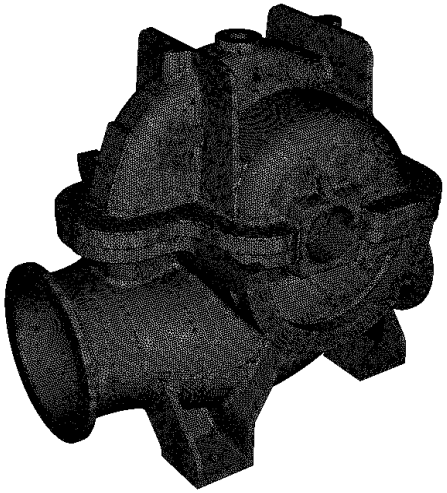
- メッセージパッシングライブラリの利用
  - メッセージパッシングライブラリ：　分散されたメモリ間でネットワークを介してデータを送受信したり、プロセスの起動や同期などの制御を行う機能をサポートしたライブラリ群。
  - FortranやCなどのライブラリとして、逐次プログラムの中からコールすることで並列計算が可能となる。
  - MPI　「MPI」は単に規格を指す。その実装系であるmpichはほとんどのメーカーのプロセッサに対応したものが準備されている。商用の汎用並列計算機にはそれぞれのアーキテクチャに最適化されたMPIが実装されている。
  - MPIは共有メモリ、分散メモリ、共有・分散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

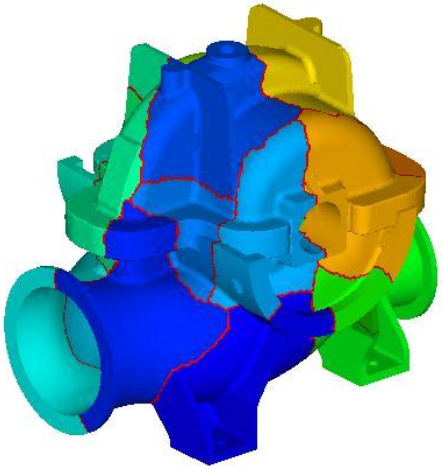
MPIのサポートする機能の例



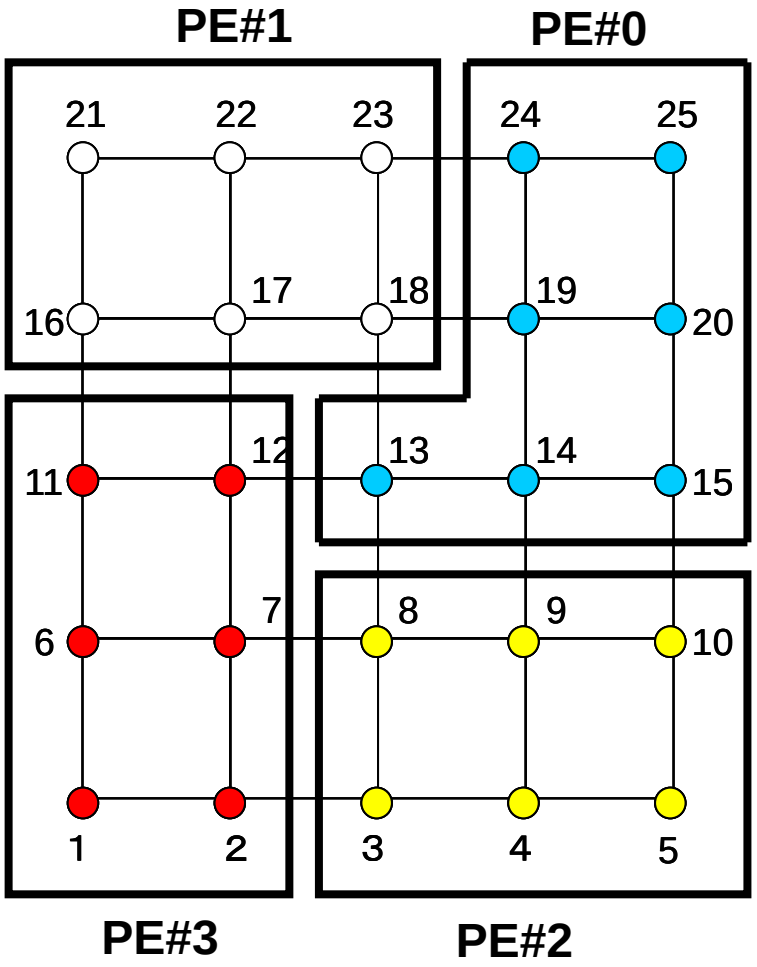
メッシュ分割



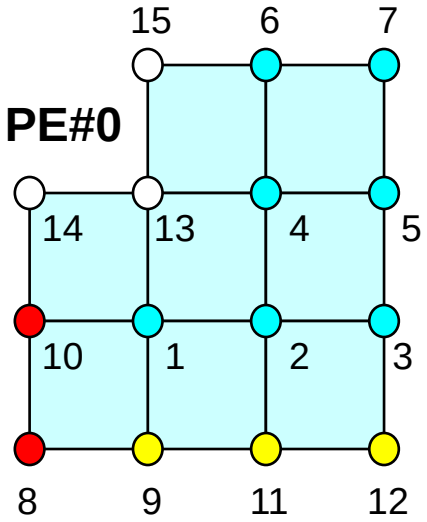
領域分割



領域分割ツール
（パーティショナ）



(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）

有限要素法データの領域分割例

<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
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
```

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。

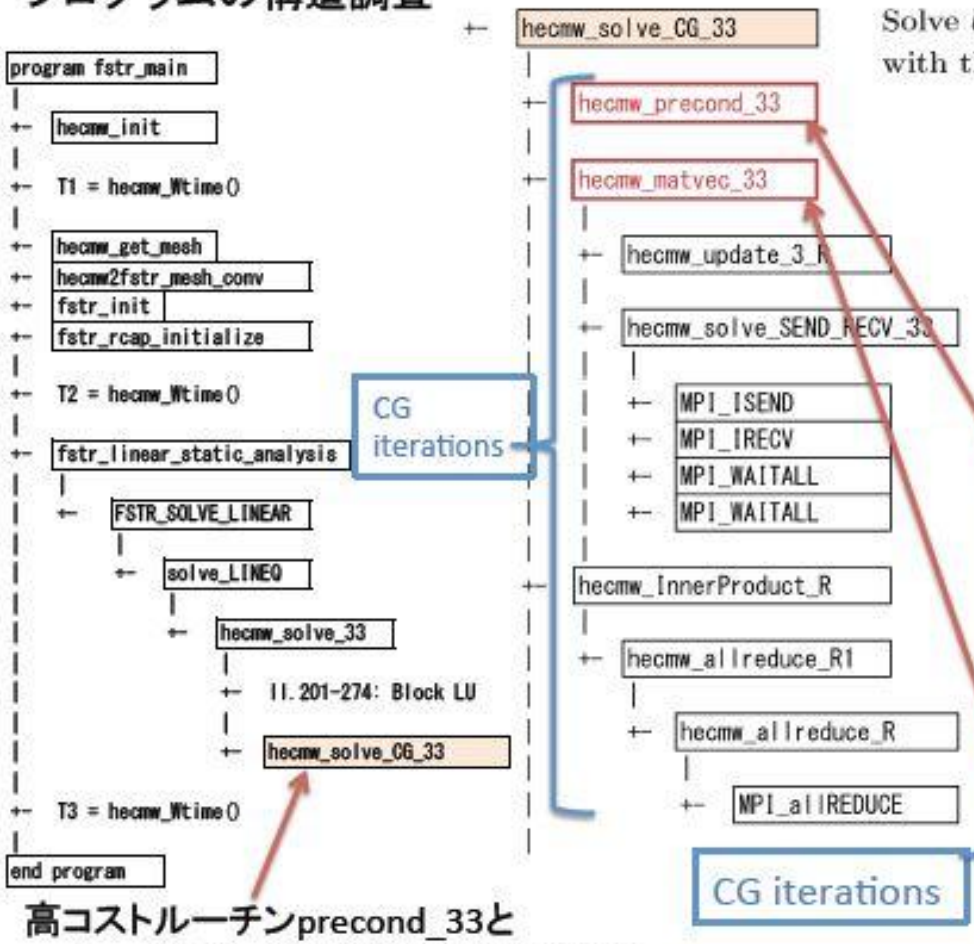
```
SUBROUTINE DAXPY(Z,A,X,Y)
INTEGER  I
DOUBLE PRECISION  Z(1000), A, X(1000), Y

!$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。

（「京」でのプログラミング例）

プログラムの構造調査



高コストルーチンprecond_33とmatvec_33は、有限要素法により離散化した連立一次方程式の解を求めるCG法においてコールされる。

有限要素法により離散化した連立一次方程式の解を求めるCG法の計算がホットスポットである。

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

高コストなルーチン matvec_33 と precond_33 のOpenMP化可能性の検討

ハイブリッド並列化のためには、高コストなルーチン matvec_33 と precond_33 に
手でOpenMPの指示文を挿入して、スレッド並列化できたらいいと期待する。

matvec_33

```
      do i= 1, hecMAT%N
        X1= X(3*i-2)
        X2= X(3*i-1)
        X3= X(3*i  )
        YV1= hecMAT%D(9*i-8)*X1 + hecMAT%D(9*i-7)*X2  &
      &                         + hecMAT%D(9*i-6)*X3
(中略)
        jS= hecMAT%indexL(i-1) + 1
        jE= hecMAT%indexL(i  )
        do j= jS, jE
          in  = hecMAT%itemL(j)
          X1= X(3*in-2)
          X2= X(3*in-1)
          X3= X(3*in  )
          YV1= YV1 + hecMAT%AL(9*j-8)*X1 + hecMAT%AL(9*j-7)*X2  &
        &                                + hecMAT%AL(9*j-6)*X3
(中略)
        enddo
        jS= hecMAT%indexU(i-1) + 1
        jE= hecMAT%indexU(i  )
        do j= jS, jE
(中略)
        enddo
        Y(3*i-2)= YV1
        Y(3*i-1)= YV2
        Y(3*i  )= YV3
      enddo
```

外側のループ

内側のループ

配列Xは参照のみ。

配列Yは代入のみ。

matvec_33には参照・代入の
依存関係は無いことが確認できた。

precond_33

```
!C-- FORWARD
      do i= 1, hecMAT%N
        SW1= ZP(3*i-2)
        SW2= ZP(3*i-1)
        SW3= ZP(3*i  )
        isL= hecMAT%indexL(i-1)+1
        ieL= hecMAT%indexL(i)
        do j= isL, ieL
          k= hecMAT%itemL(j)
          X1= ZP(3*k-2)
          X2= ZP(3*k-1)
          X3= ZP(3*k  )
          SW1= SW1 - hecMAT%AL(9*j-8)*X1 - hecMAT%AL(9*j-7)*X2  &
        &                                - hecMAT%AL(9*j-6)*X3
(中略)
        enddo
(中略)
        X1= hecMAT%ALU(9*i-8)*( X1 - hecMAT%ALU(9*i-6)*X3  &
      &                            - hecMAT%ALU(9*i-7)*X2)
        ZP(3*i-2)= X1
        ZP(3*i-1)= X2
        ZP(3*i  )= X3
      enddo
```

外側ループに関する
以前の行番号を参照

前の行のZPの値が
参照されている！

precond_33には
参照・代入の依存関
係がある。

参照・代入に依存関係があるので、
スレッド並列化できない。
OpenMP化には、節点ループに
~~依存関係がないように~~
節点番号のリオーダリング処理が必要。

今回の作業では、matvec_33のOpenMP化を実施することとした。

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)
                  !$OMP DO
33    1   p           do i= 1, hecMAT%N
34    1   p             X1= X(3*i-2)
              (中略)
37    1   p           YV1= hecMAT%D(9*i-8)*X1 + hecMAT%D(9*i-7)*X2                &
38    1             &                         + hecMAT%D(9*i-6)*X3
              (中略)
46    2   p   v       do j= jS, jE
47    2   p   v         in  = hecMAT%itemL(j)
48    2   p   v         X1= X(3*in-2)
              (中略)
51    2   p   v         YV1= YV1 + hecMAT%AL(9*j-8)*X1 + hecMAT%AL(9*j-7)*X2        &
52    2             &                                  + hecMAT%AL(9*j-6)*X3
              (中略)
57    2   p   v       enddo
              (中略)
72    1   p           Y(3*i-2)= YV1
              (中略)
75    1   p           enddo
76                3OMP END DO
77                !$OMP END PARALLEL
```

外側のループ

配列Xは参照のみ。

内側のループ

配列Xは参照のみ。

配列Yは代入のみ。

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。

GPUマシン 略図

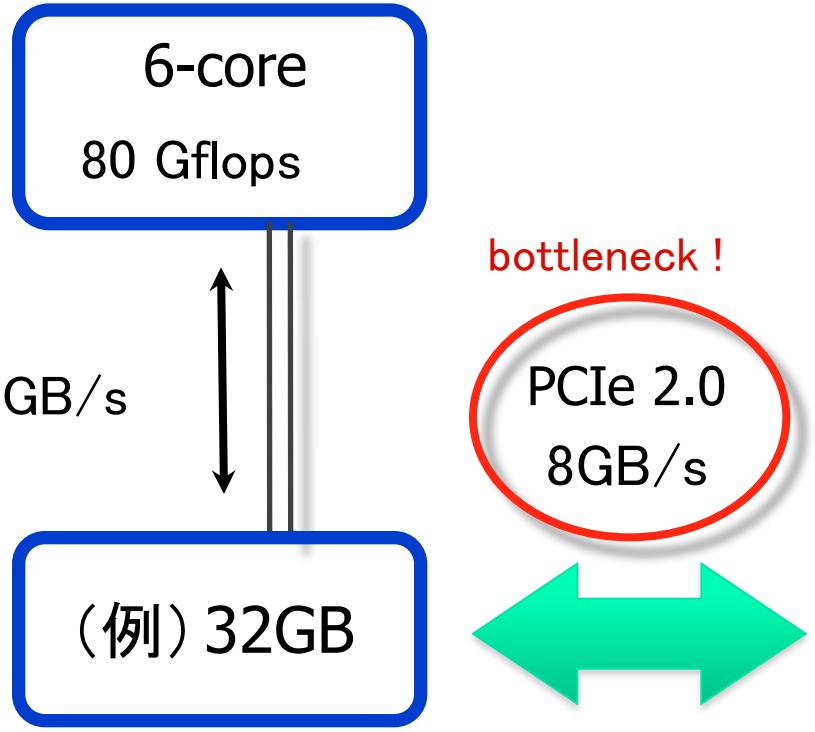
GPU

（Tesla C2075）

CPU

（Xeon X5680）

メインメモリ



MPIと同じで、CPU⇆GPU 間の転送が必要（※遅い）

CUDAプログラミング (2/3)

CPUの場合を復習 （Intel MKL, CSR formatでSpMV）

```
#include “mkl.h”
#define N 10000
        ⋮
mkl_dcsrgemv(&transa, &N, val, row, col, x, y)
        ⋮
```

```
# icc spmv.c -lmkl_intel_lp64 -Wl,--start-group ･･･
```

CUDAプログラミング（3/3）

GPUでCUDAの場合　（CUSPARSEライブラリ, CSR formatでSpMV）

```
#include “cusparse_v2.h”
#define N 10000
        :
cudaMalloc((void**)&val, nz*sizeof(double));
cudaMalloc((void**)&row, (N+1)*sizeof(int));
cudaMalloc((void**)&col, nz*sizeof(int));

cudaMemcpy(val, dval, nz*sizeof(double), cudaMemcpyHostToDevice);
cudaMemcpy(row, drow, (N+1)*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(col, dcol, nz*sizeof(int), cudaMemcpyHostToDevice);
cudaMemcpy(x, dx, N*sizeof(double), cudaMemcpyHostToDevice);

cusparseDcsrmv(handle, CUSPARSE_OPERATION_NON_TRANSPOSE, N, N, nz,
                    &alpha, descr, val, row, col, x, &beta, y)

cudaMemcpy(x, dx, N*sizeof(double), cudaMemcpyDeviceToHost);
        :
```

GPUメモリを malloc

CPU → GPU

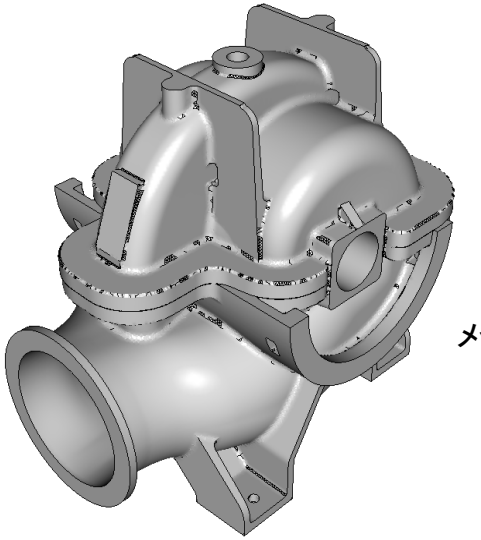
CPU ← GPU

```
# nvcc spmv.cu -lcusparse
```

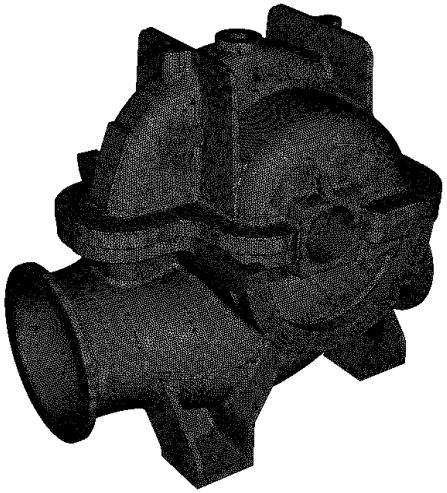
- なぜ並列？
- 並列アーキテクチャ
- 並列プログラミング
- **ノード間並列　領域分割とMPI**
- ノード内並列（単体性能）　ループ分割とOpenMP
- 実効性能について

領域分割

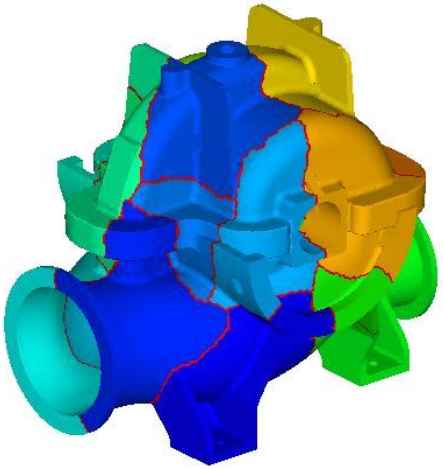
- 部分領域への分割
  - 部分領域のデータを分散メモリに割り当て、各部分領域の計算を並列に実施する。
  - 一般に部分領域ごとの計算は完全には独立ではなく、行列ベクトル積や内積計算において、領域全体の整合性をとるために通信が必要。
  - 部分領域間での通信ができるだけ少なくてすむように、データが局所化されている必要がある。
- 通信テーブル
  - 隣接する部分領域との間の節点や要素の接続情報
  - 領域分割のツール　METIS



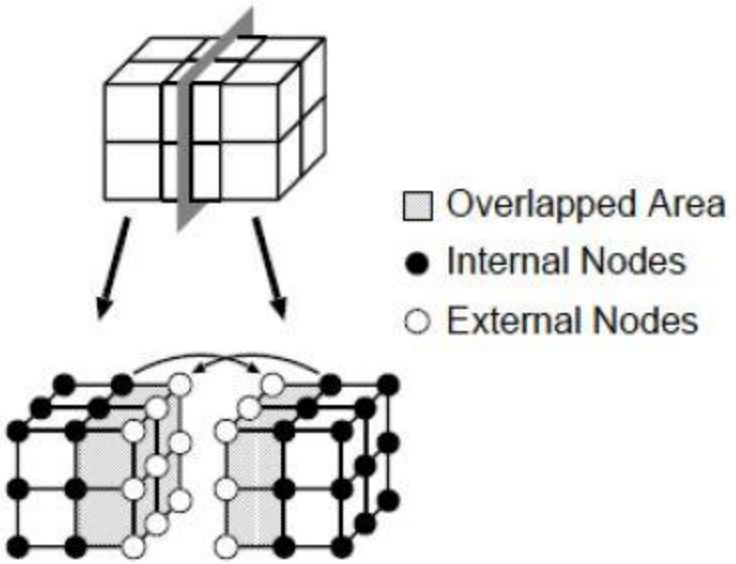
メッシュ分割



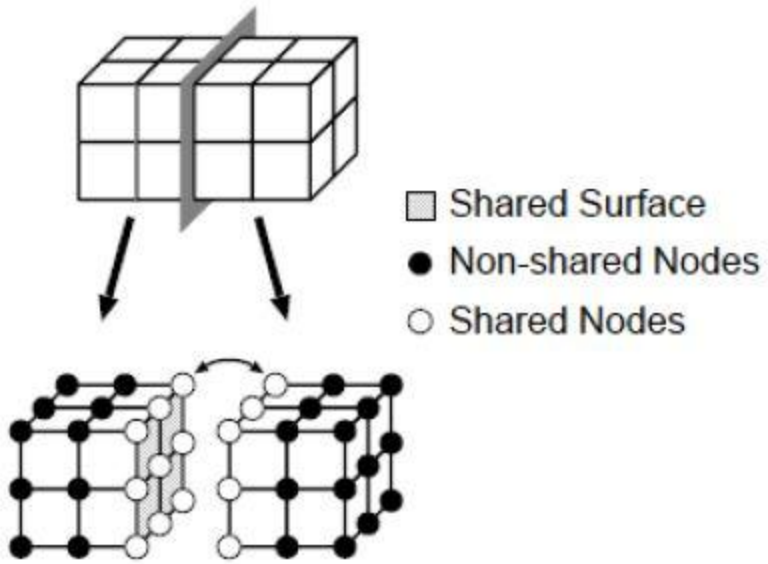
領域分割



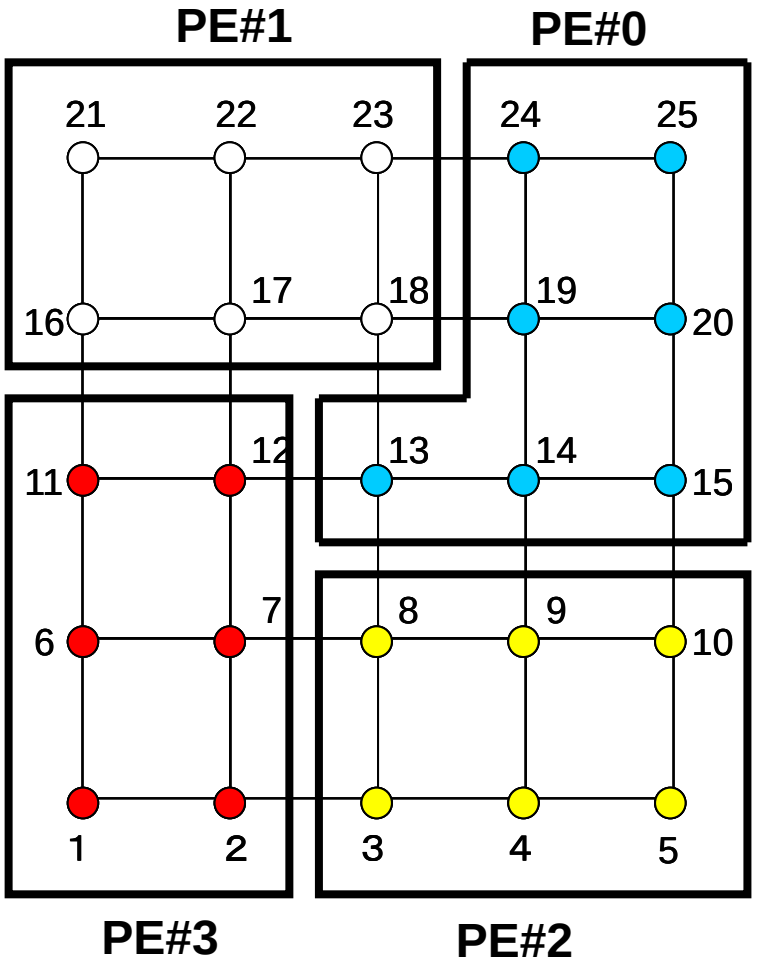
領域分割ツール
（パーティショナ）



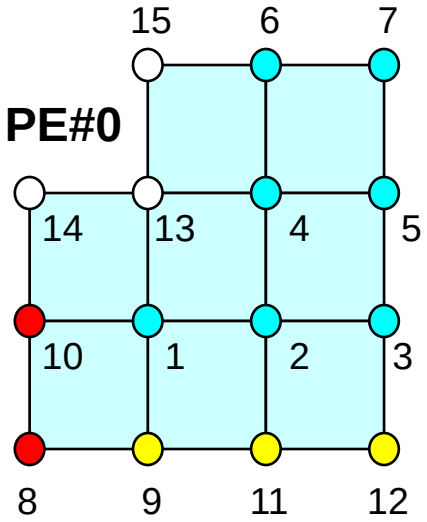
(a) Node-based



(b) Element-based



(a) 4領域への分割



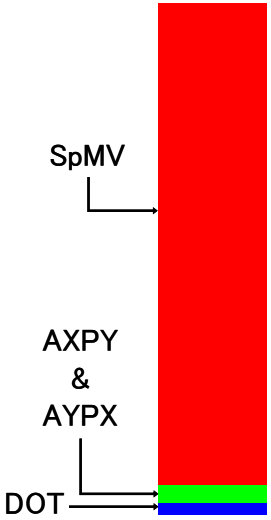
(b) PE#0が保持する情報
（隣接PEとのオーバーラップあり）

有限要素法データの領域分割例

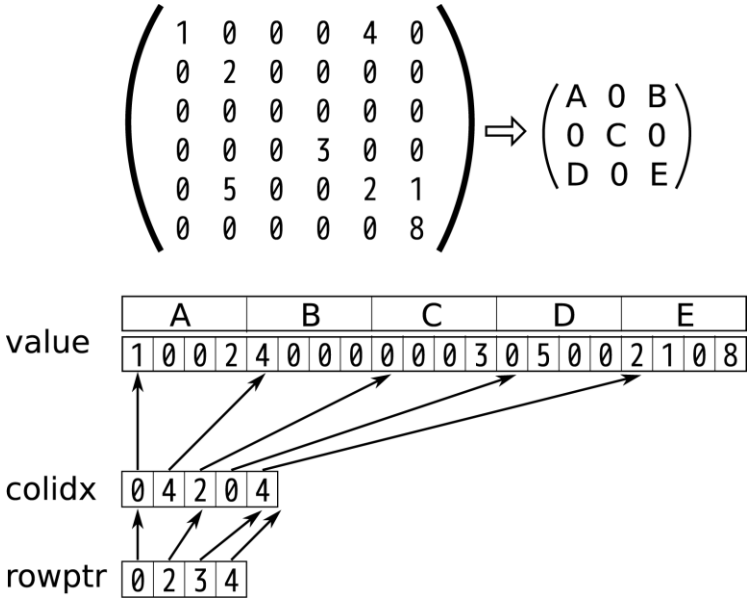
- なぜ並列？
- 並列アーキテクチャ
- 並列プログラミング
- ノード間並列　領域分割とMPI
- **ノード内並列（単体性能）　ループ分割とOpenMP**
- 実効性能について

Blocking with padding for multicore CPU and GPUs

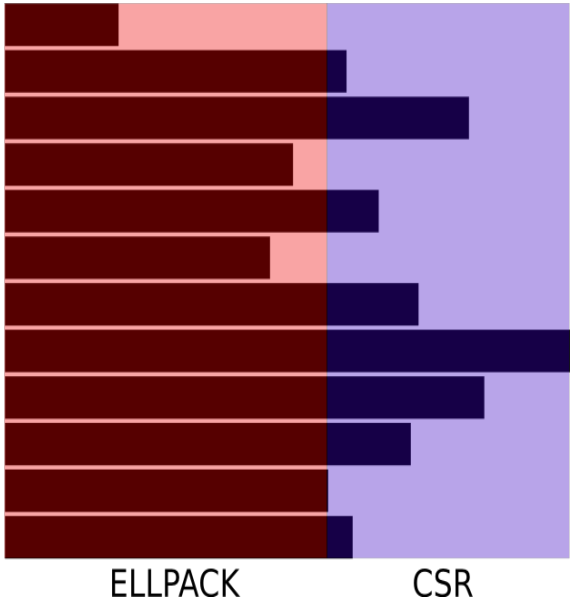
- SpMV in iterative solvers is crucial for unstructured grid.
- For improving B/F ratio
  - Blocking, Padding, ELL+CSR.



Breakdown of CPU in CG operations

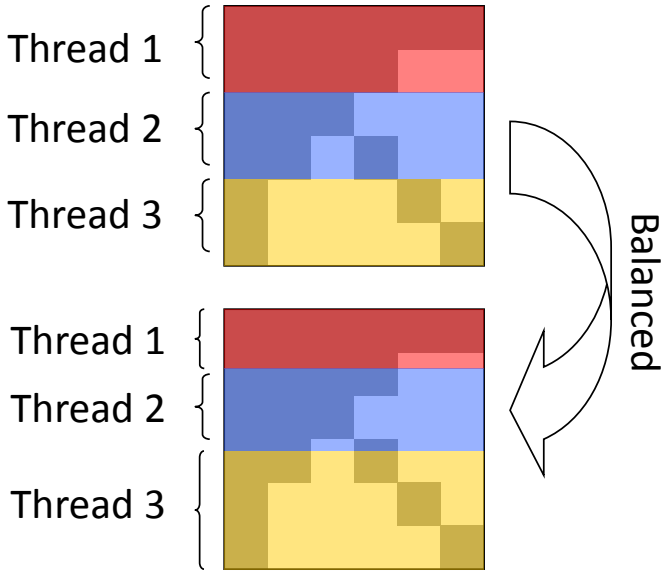


Blocked CSR

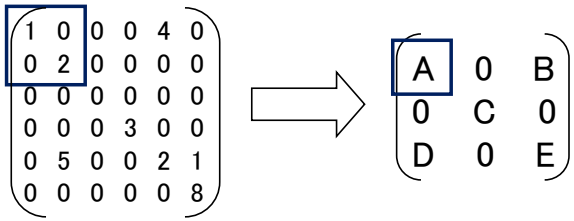


HYB format = ELL + CSR4

Acceleration of SpMV Computation



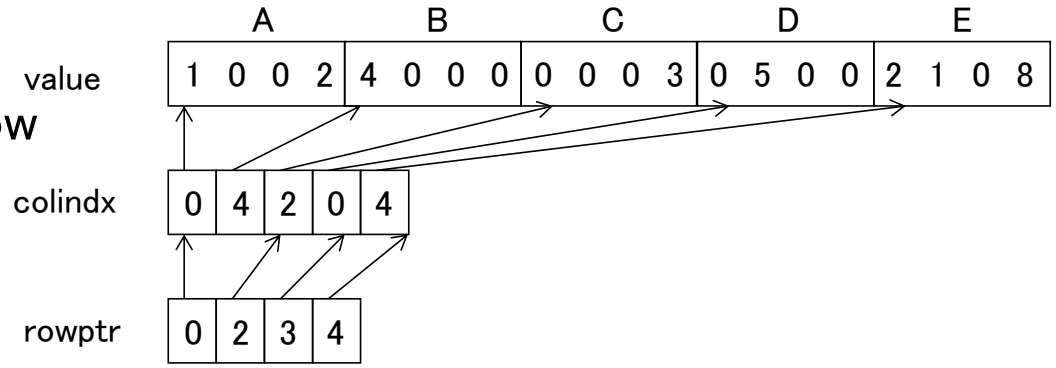
- Parallelization
  - ✓ Rows are distributed among threads.
- Load balancing
  - ✓ Reallocate rows to balance loads.



- Blocking
  - ✓ Matrix format is crucial.
  - ✓ CSR: Compressed Sparse Row

    B/F = 6.25～12.5
  - ✓ BCSR: Blocked CSR

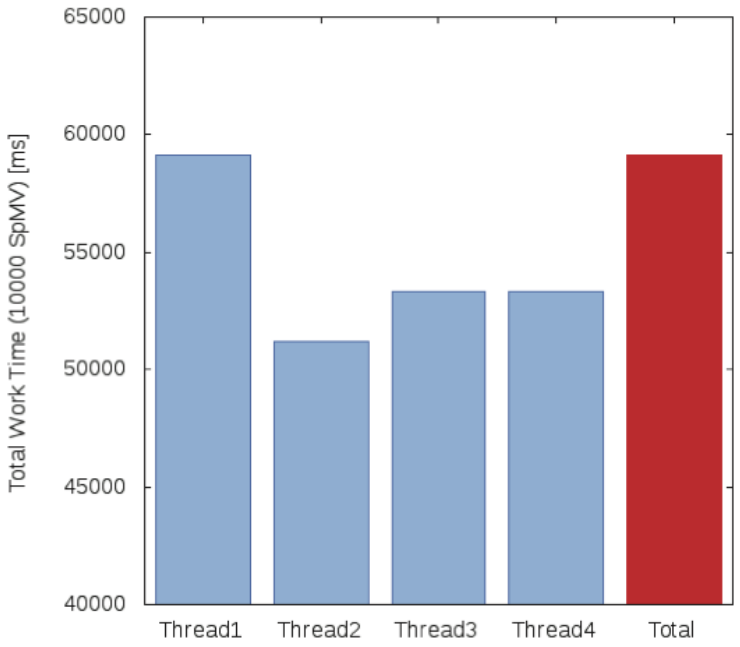
    B/F = 4.76～5.56

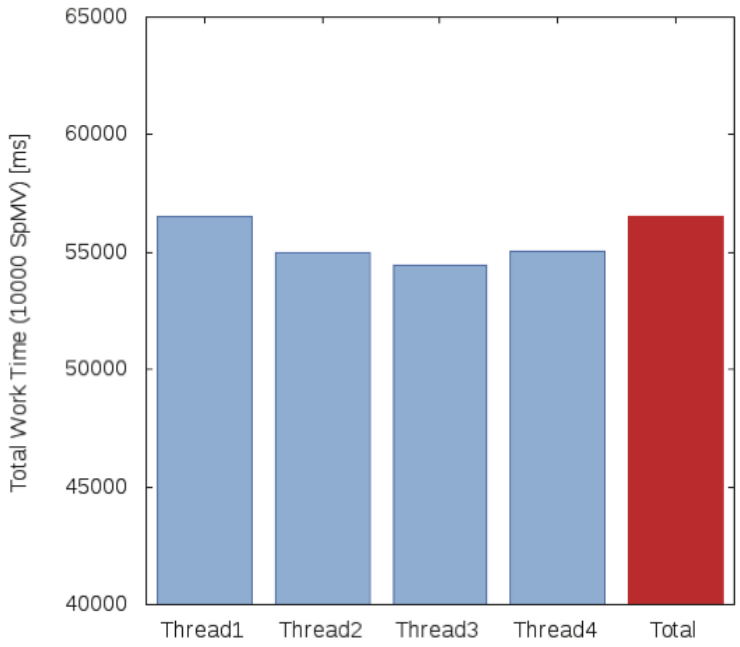


Performance Test (1/3)
Load Balancing

on Nehalem (Core i7 975)

- x10,000 SpMV of unbalanced matrices from the library[2]
- Left: w/o. load balancing
- Right: with load balancing

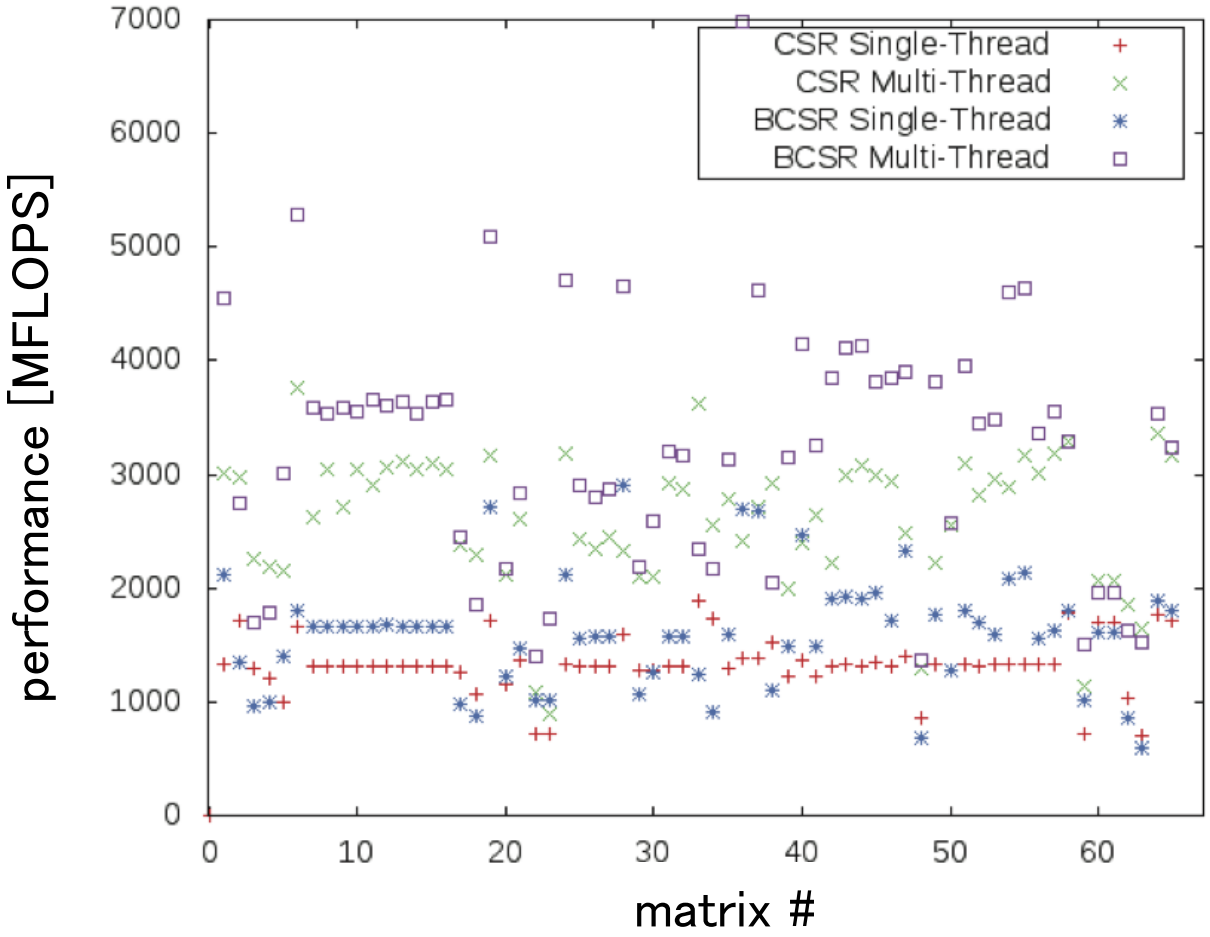




[2] T. A. Davis. University of Florida sparse matrix collection, 1997.

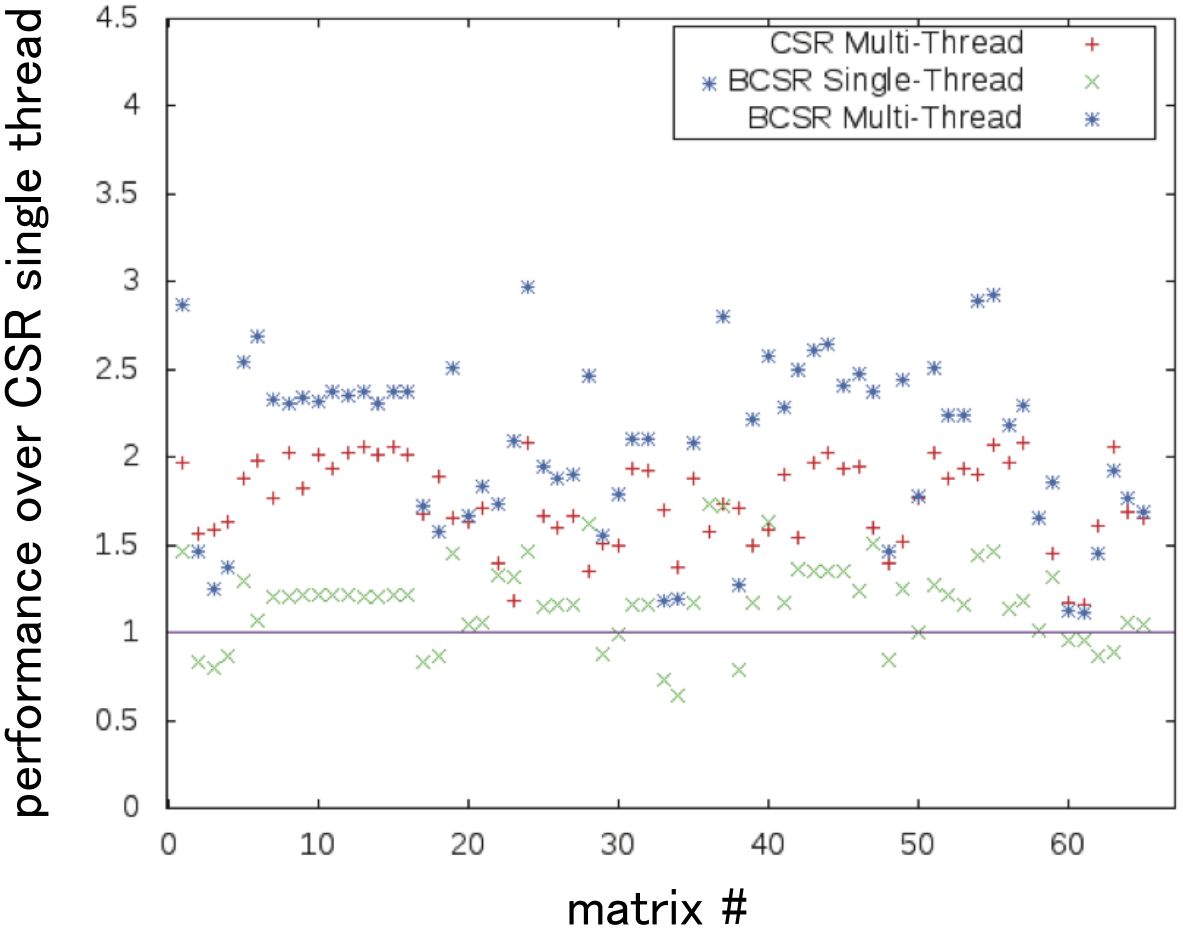
Performance Test (2/3) Parallelization and Matrix Format

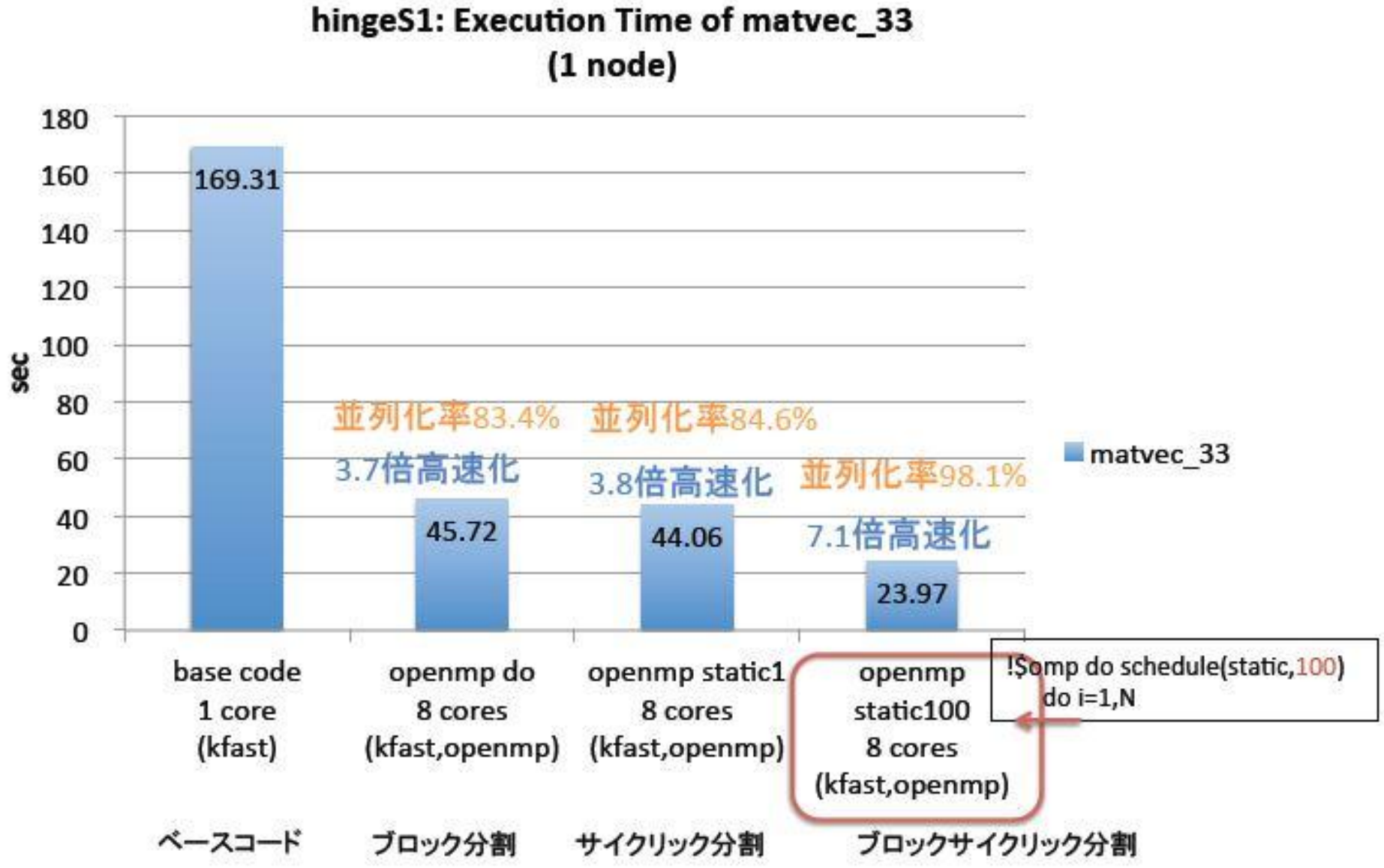
- Performance of SpMV on Nehalem (Core i7 975)
- CSR / BCSR format



Performance Test (3/3) Overall CG Solver

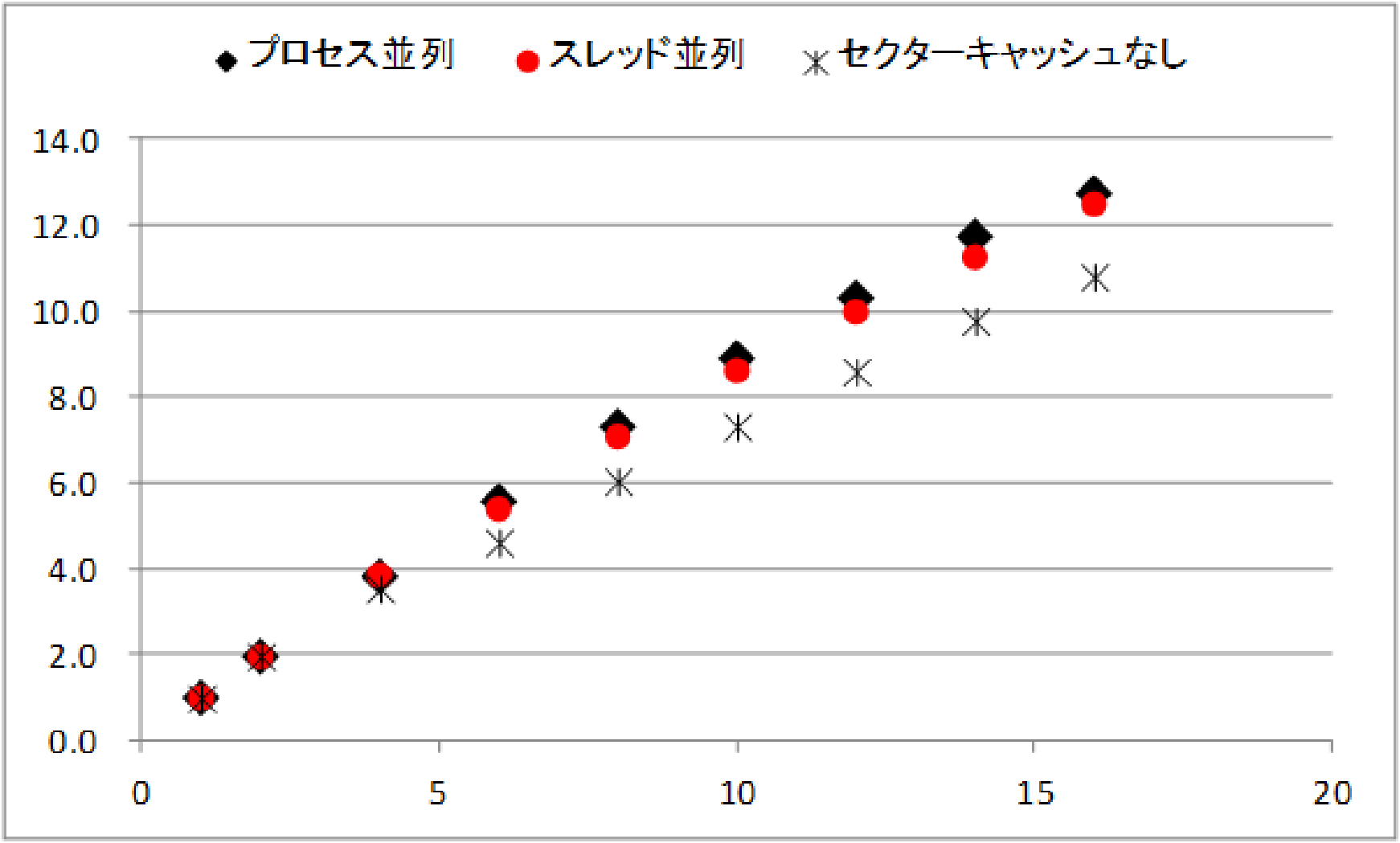
- CG solver's performance on CSR single thread on Nehalem (Core i7 975)





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）

- なぜ並列？
- 並列アーキテクチャ
- 並列プログラミング
- 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**

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

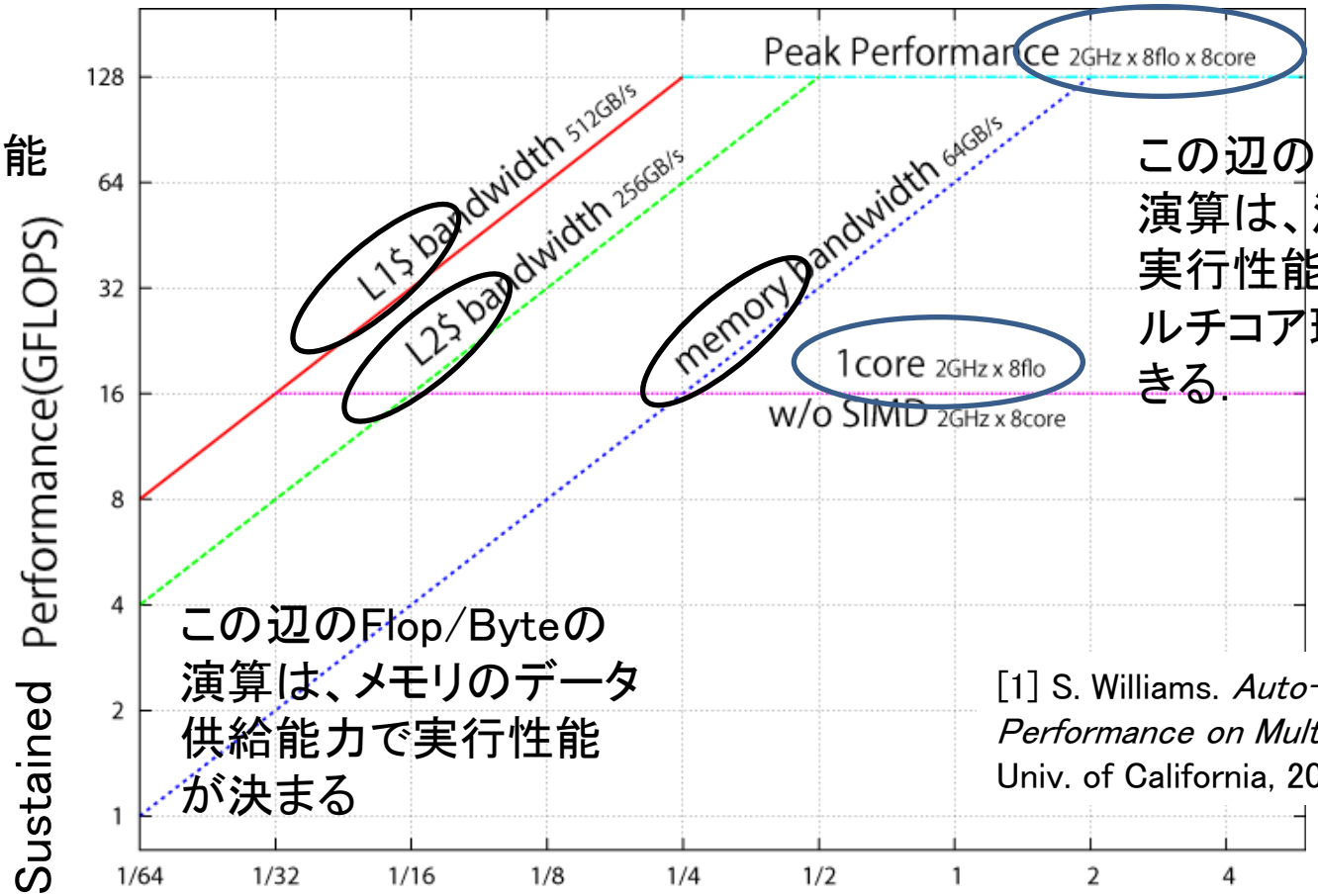
m: number of columns,

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)

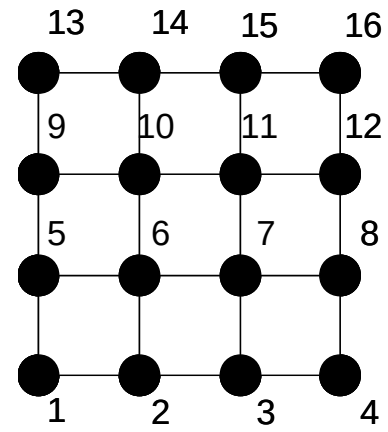
例えば、次式のような演算を考える。

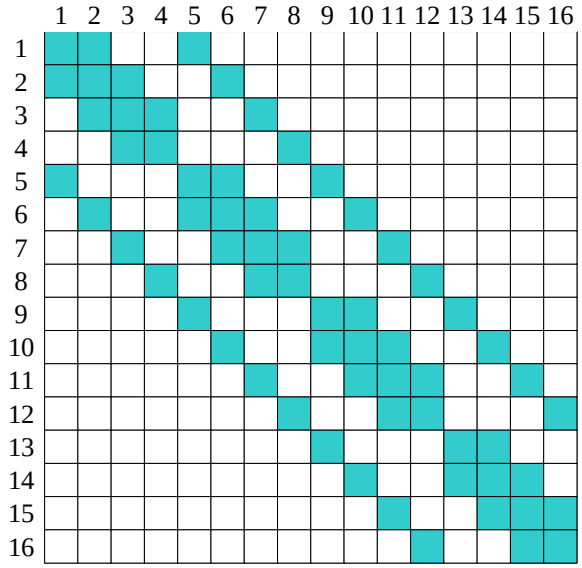
$$x_i = y_i - \sum_{k=1}^{i-1} L_{ik} x_k$$

- 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。

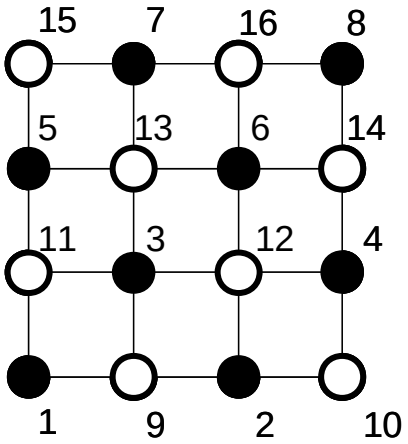
- オーダリング前の節点番号付けの場合、節点iの演算の際にそれ以前に演算済みの情報が必要であることがわかる。
- 依存性のない節点を2色に色分けて(黒と白)番号を付け替えた場合、同一色に属する節点に関する演算は互いに依存性がないことがわかる。すなわち、ノード内並列処理やベクトル処理が可能となる。

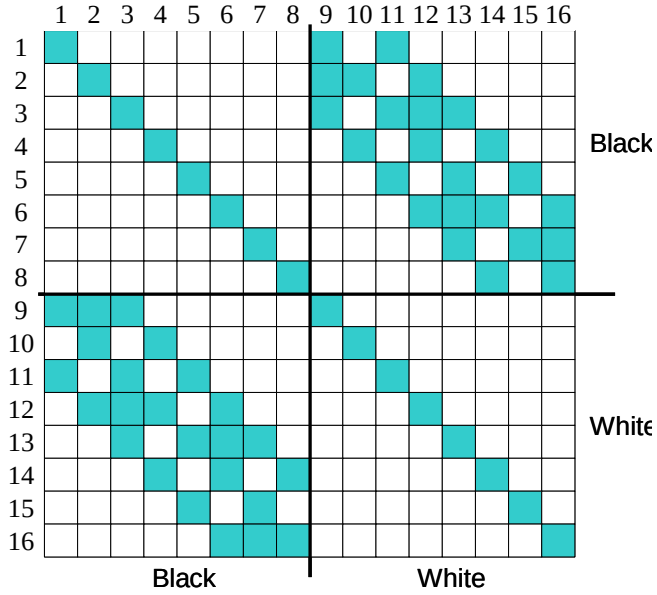
- red and black法、マルチカラー法
- 演算に依存性のない節点をハイパープレーンと呼ばれるグループに分類し、各ハイパープレーン上の節点についてノード内並列処理やベクトル処理を行うことも多い。





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