OpenFOAMのMPI+OpenMP+GPGPUによる高速計算

株式会社 EQN

松村　茂

2014.11.14

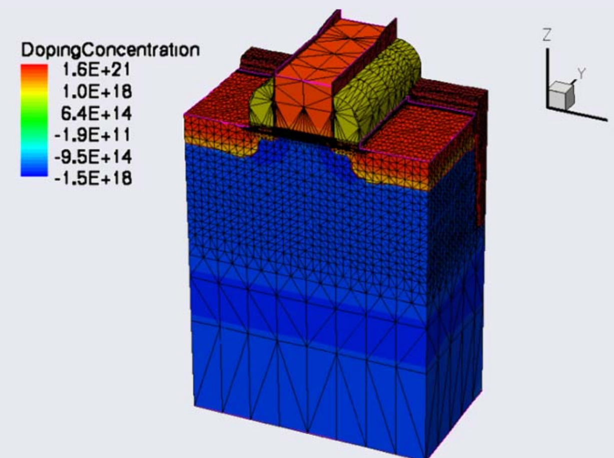
紹介

株式会社EQN

代表取締役 松村 茂

主に半導体シミュレーション(プロセス･デバイス)関係

- 物理モデル
- 線形ソルバー
- Delaunayメッシュ
- 形状
- etc...



目次

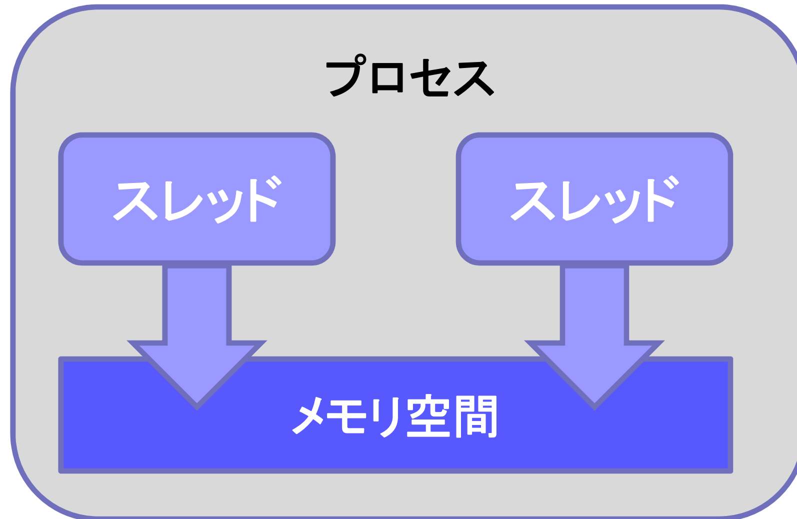
1. 並列化・線形ソルバー基礎
2. OpenFOAMへの導入
3. ベンチマーク結果

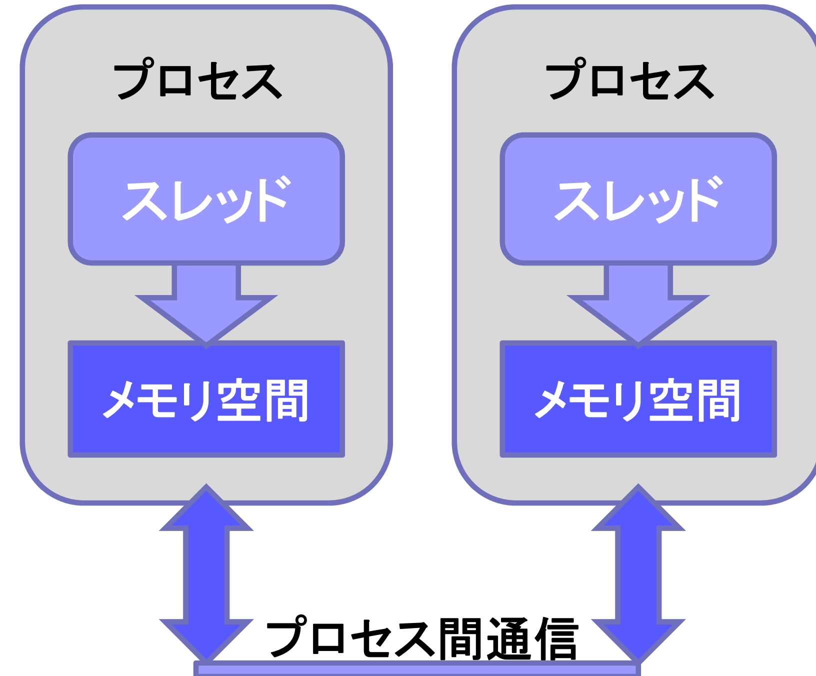
目次

1. 並列化・線形ソルバー基礎
2. OpenFOAMへの導入
3. ベンチマーク結果

並列化・線形ソルバー基礎

■ メモリの取り扱いから見た並列化の種類



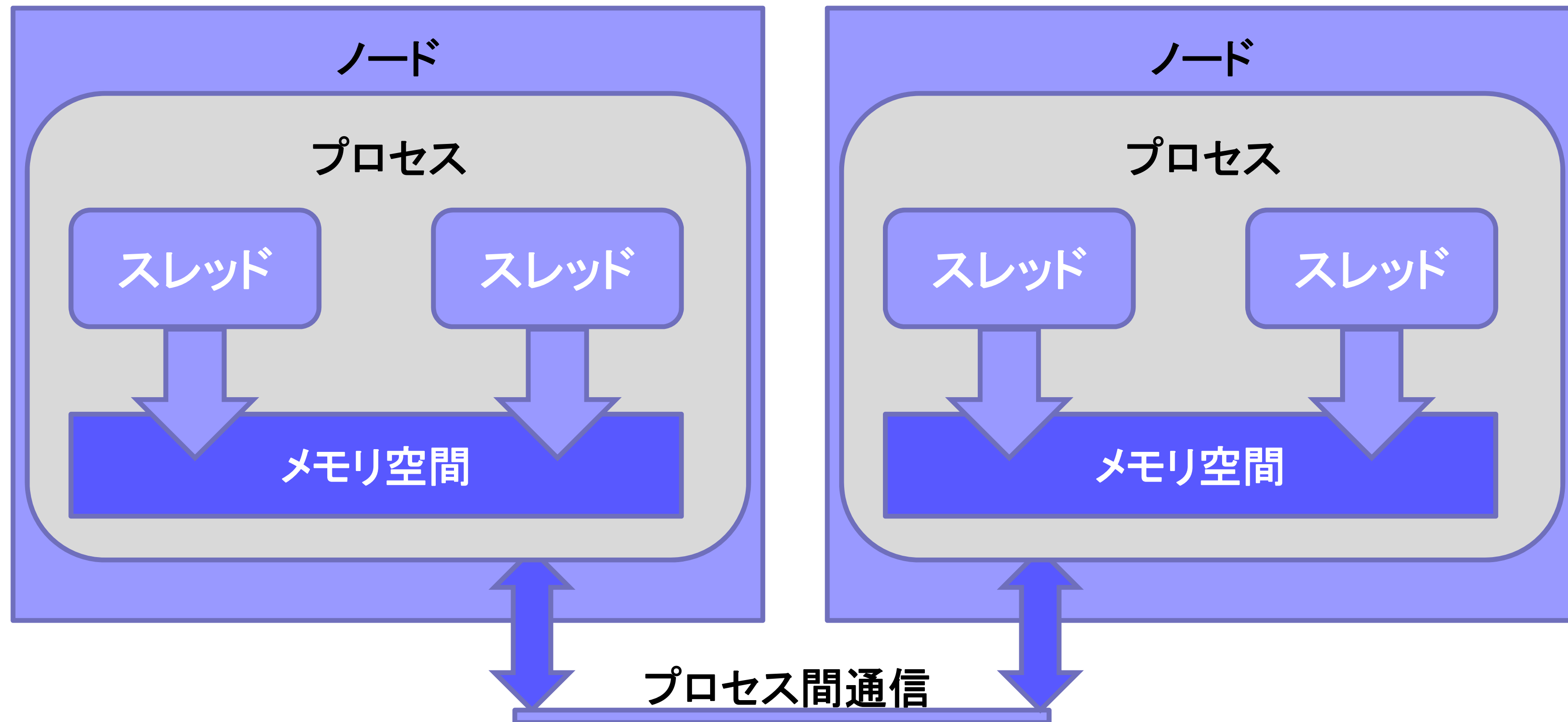


- OpenMPやpthread等
- 一般にノード内での並列化

- MPIやPVM等
- 一般に複数ノードでの並列化
- **OpenFOAMが採用**

並列化・線形ソルバー基礎

- メモリの取り扱いから見た並列化の種類



- ノード内または複数ノードにおいて共有メモリ型、分散メモリ型並列を使う
- **今回の計算で採用**

並列化・線形ソルバー基礎

- プログラムから見た並列化技術

MPI
- 分散メモリ型並列
- **OpenFOAMが採用**

OpenMP
- 共有メモリ型並列

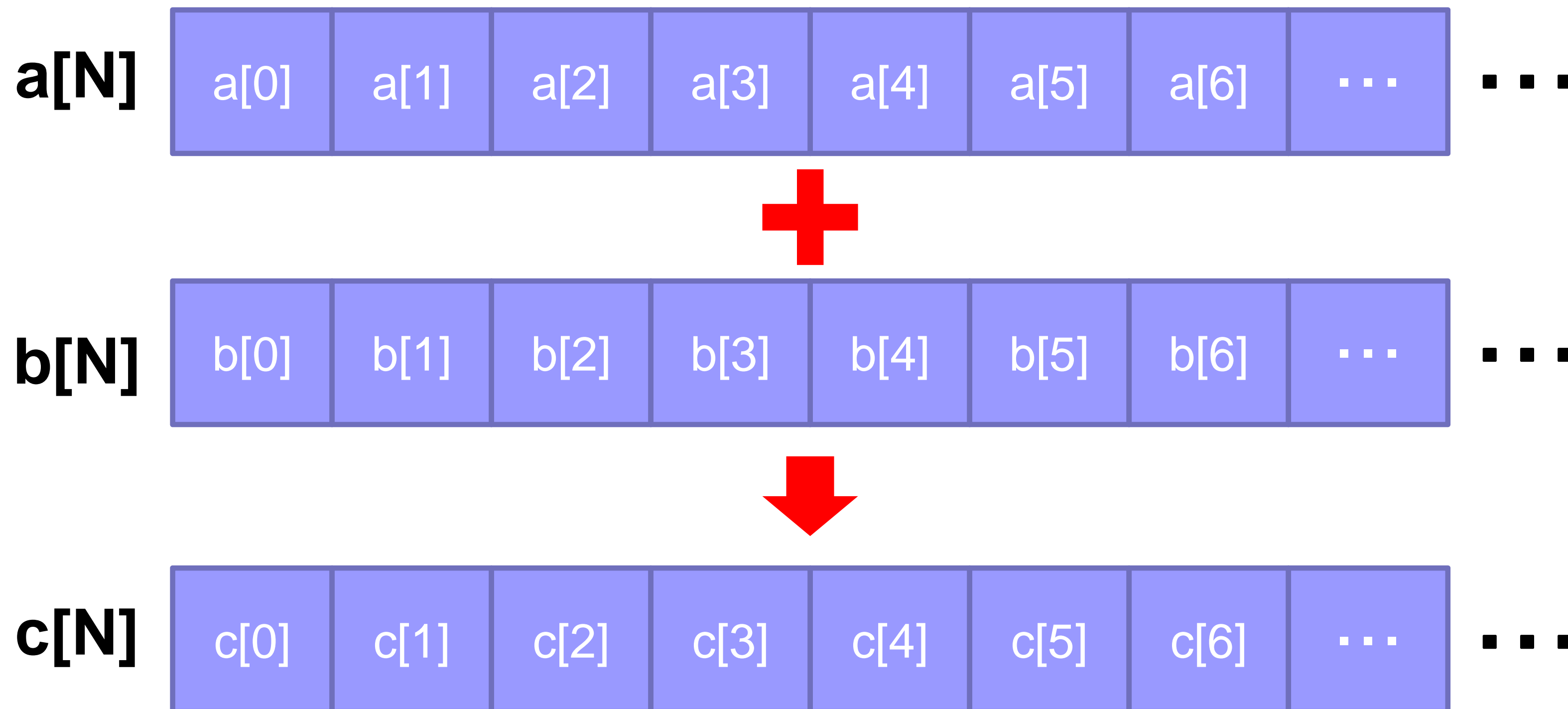
CUDA
- GPU上で並列計算を実行

今回の計算に採用

並列化・線形ソルバー基礎

- プログラムから見た並列化技術の例

ベクトル和の計算の例



並列化・線形ソルバー基礎

- 各手法によるベクトル和の計算の違い

シングルCPU

```
void add( int *a, int *b, int *c ) {
        for ( int tid = 0; tid < N; tid++ ) {
                c[tid] = a[tid] + b[tid];
        }
}
```

c配列=a配列+b配列

OpenMP

```
void add( int *a, int *b, int *c ) {
        #pragma omp parallel for
        for ( int tid = 0; tid < N; tid++ ) {
                c[tid] = a[tid] + b[tid];
        }
}
```

このループがCPU数に同時に展開される

CUDA

```
__global__ void add( int *a, int *b, int *c) {
        int tid = threadIdx.x + blockIdx.x * blockim.x;
        for( ; tid < N; tid += blockDim.x * gridDim.x ) {
                c[tid] = a[tid] + b[tid];
        }
}
```

この関数がGPUコアに同時に展開される

並列化・線形ソルバー基礎

線形行列ソルバーとは

$$\mathbf{A} \times \vec{x} = \vec{b} \quad \text{(Aは行列)}$$

- Aと$\vec{b}$が既知の時$\vec{x}$を計算する。
- 陰解法を用いるシミュレーターはこれに依存する。
- 計算時間のほとんど全てがこの計算に消費される。

行列の種類

密行列

- 行列全体のほとんどが0以外の要素からなる。

疎行列

- 行列要素のほとんどが0からなる。
- 隣接する要素間の関係のみ必要とする計算行列は疎行列
- **OpenFOAMの流体・構造計算は疎行列計算**

並列化・線形ソルバー基礎

- 線形行列ソルバーの種類

直接法(LU分解)

- 必ず何らかの解が得られる。
- 精度は保証されない。
- メモリ量O(n^2) (密行列)
- 計算時間O(n^3) (密行列)

  行列サイズn=1万(1万x1万の行列)の計算に1秒かかる場合n=1億を計算するのに3万年以上かかる。

⇒ 大規模計算に向かない

反復法

- 発散等、解が得られる保証がない。
- 解の精度は保証される。
- 膨大なアルゴリズムがある。

  CG、BiCG、BiCGStab、GMRES、AMG…等。

  ⇒ OpenFOAMではCG, BiCG, GAMGなど採用
- 前処理を用いて収束性を上げる。

  ヤコビ、SOR、IC、ILU前処理等

  ⇒OpenFOAMではDILU,DICなど採用(簡易的に対角のみILU,ICを計算する手法)

目次

1. 並列化・線形ソルバー基礎
2. OpenFOAMへの導入
3. ベンチマーク結果

OpenFOAMへの導入

TESTsolver.H ヘッダーファイル

```
namespace Foam
{
class TESTsolver : public lduMatrix::solver
{
        TESTsolver(const TESTsolver&);
        void operator=(const TESTsolver&);
public:
    TypeName("TESTsolver");

    TESTsolver (
          const word& fname,
          const lduMatrix& mtx,
          const FieldField<Field, scalar>& ifBC,
          const FieldField<Field, scalar>& ifIC,
          const lduInterfaceFieldPtrsList& iface,
          const dictionary& ctrl
    );
    virtual ~TESTsolver(){}
    virtual solverPerformance solve ( scalarField& x,
          const scalarField& b, const direction cmpt=0 ) const;
};
}
```

驚くほど簡単に実装可能！

ソルバ名定義

コンストラクタ

デストラクタ

solve関数

OpenFOAMへの導入

TESTsolver.C ソースコード(その1)

```
namespace Foam
{
    defineTypeNameAndDebug(TESTsolver, 0);

    lduMatrix::solver::addasymMatrixConstructorToTable<TESTsolver>
        addTESTsolverAsymMatrixConstructorToTable_;
    lduMatrix::solver::addsymMatrixConstructorToTable<TESTsolver>
        addTESTsolverSymMatrixConstructorToTable_;
}
```

非対称行列対応の場合

対称行列対応の場合

両方書くと対称行列、非対称行列両方対応のソルバーとして認識される。

OpenFOAMへの導入

TESTsolver.C ソースコード(その2)

```
Foam::TESTsolver::TESTsolver (
    const word& fname,
    const lduMatrix& mtx,
    const FieldField<Field, scalar>& ifBC,
    const FieldField<Field, scalar>& ifIC,
    const lduInterfaceFieldPtrsList& iface,
    const dictionary& ctrl
): lduMatrix::solver ( fname, mtx, ifBC, ifIC, iface,ctrl ) {}

Foam::solverPerformance Foam::TESTsolver::solve (
    scalarField& x, const scalarField& b, const direction cmpt) const
{
    solverPerformance solverPerf ( typeName, fieldName_ );

    //
    // ここに何かソルバーのコードを書く
    //

    return solverPerf;
}
```

コンストラクタ

中身なしでもOK

solve関数
A･x = b を解く

OpenFOAMへの導入

線形ソルバーの例

PCG(Preconditioned Conjugate Gradient)

$$\mathbf{r}_0 := \mathbf{b} - \mathbf{A}\mathbf{x}_0$$
$$\mathbf{z}_0 := \mathbf{M}^{-1}\mathbf{r}_0$$
$$\mathbf{p}_0 := \mathbf{z}_0$$
$$k := 0$$

repeat

$$\alpha_k := \frac{\mathbf{r}_k^T\mathbf{z}_k}{\mathbf{p}_k^T\mathbf{A}\mathbf{p}_k}$$
$$\mathbf{x}_{k+1} := \mathbf{x}_k + \alpha_k\mathbf{p}_k$$
$$\mathbf{r}_{k+1} := \mathbf{r}_k - \alpha_k\mathbf{A}\mathbf{p}_k$$

if $\mathbf{r}_{k+1}$ is sufficiently small then exit loop end if

$$\mathbf{z}_{k+1} := \mathbf{M}^{-1}\mathbf{r}_{k+1}$$
$$\beta_k := \frac{\mathbf{z}_{k+1}^T\mathbf{r}_{k+1}}{\mathbf{z}_k^T\mathbf{r}_k}$$
$$\mathbf{p}_{k+1} := \mathbf{z}_{k+1} + \beta_k\mathbf{p}_k$$
$$k := k + 1$$

end repeat

The result is $\mathbf{x}_{k+1}$

行列・ベクトル積計算

前処理計算

計算コスト≒高速化の余地

行列ベクトル積・前処理　>　内積　>　ベクトル加減・定数倍

特にILU前処理などは高コスト

OpenFOAMへの導入

使い方

1. ソルバーディレクトリのMake/filesに

   TESTsolver.C

   を記述してwmake

2. caseディレクトリのsystem/fvSolutionから

   solver TESTsolver;

   が選択可能になっている！

OpenFOAMへの導入

- 疎行列格納フォーマットの例

$$\begin{bmatrix} 1 & 2 & 0 & 0 \\ 0 & 3 & 4 & 0 \\ 0 & 0 & 5 & 0 \\ 0 & 0 & 6 & 7 \end{bmatrix}$$

CSR

ia	1	3	5	6	8		
ja	1	2	2	3	3	3	4
val	1	2	3	4	5	6	7

iaは左上から非零要素を数えた時各行の左端の要素が何番目かを表す。

COO

ia	1	1	2	2	3	4	4
ja	1	2	2	3	3	3	4
val	1	2	3	4	5	6	7

ia、jaがそれぞれの要素の行・列の座標を表す。valは要素値。

OpenFOAMへの導入

■ 疎行列格納フォーマットの例

1	2
3	4

メッシュ構造と要素番号

$$\begin{bmatrix} 1 & 2 & 3 & 0 \\ 4 & 5 & 0 & 6 \\ 7 & 0 & 8 & 9 \\ 0 & 10 & 11 & 12 \end{bmatrix}$$

OpenFOAM内部形式

upper	2	3	6	9
lower	4	7	10	11
diag	1	5	8	12
lowerAddr	1	1	2	3
upperAddr	2	3	4	4

- upper, lower, diagはそれぞれ上三角、下三角、対角の行列要素。
- lowerAddr, upperAddrはそれぞれメッシュ構造の面番号における、隣接要素の要素番号。
- 対角対称構造のみ表現可能。

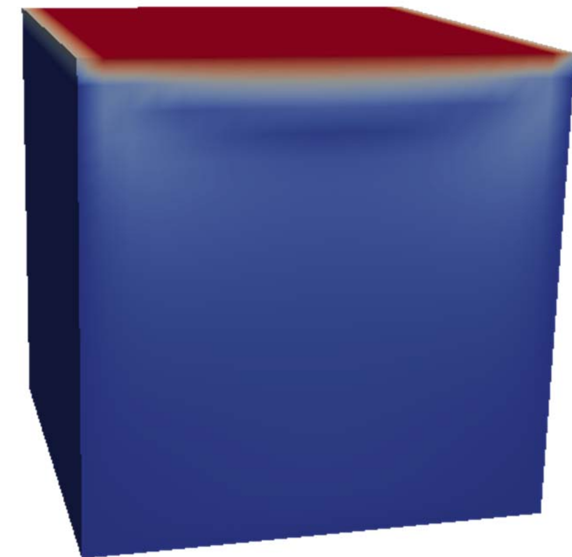
目次

1. 並列化・線形ソルバー基礎
2. OpenFOAMへの導入
3. ベンチマーク結果

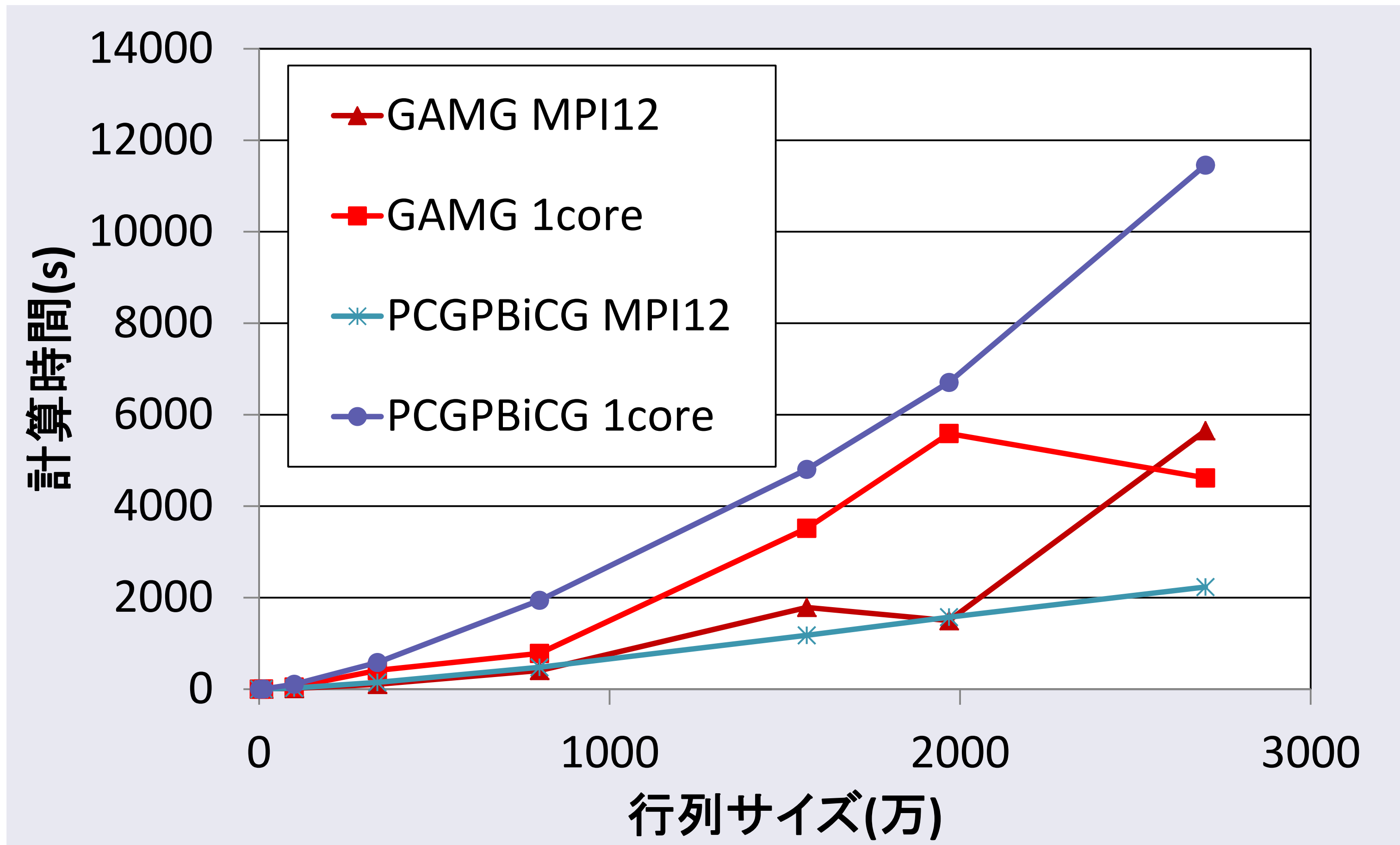
ベンチマーク結果

ベンチマーク条件

項目	
CPU	Intel Xeon E5-2650 2.0GHz
GPU	NVIDIA GeForce GTX Titan
OS	CentOS 6.5
CUDA	CUDA Ver 6.5
OpenFOAM	OpenFOAM-2.3.0
Case	incompressible/icoFoam/cavity
Ref. Solver	p: DIC-PCG U: DILU-PBiCG
変更点	領域を(0,0,0)-(1,1,1)の立方体に deltaT 1e-7 endTime 5e-7
GPU計算方式	全て倍精度
計測時間	全て線形ソルバーのみの計算時間



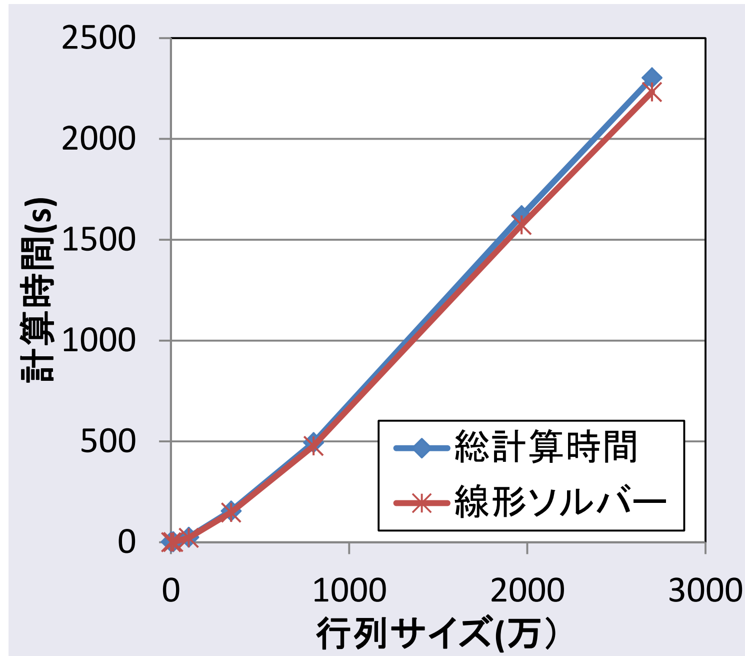
ベンチマーク結果

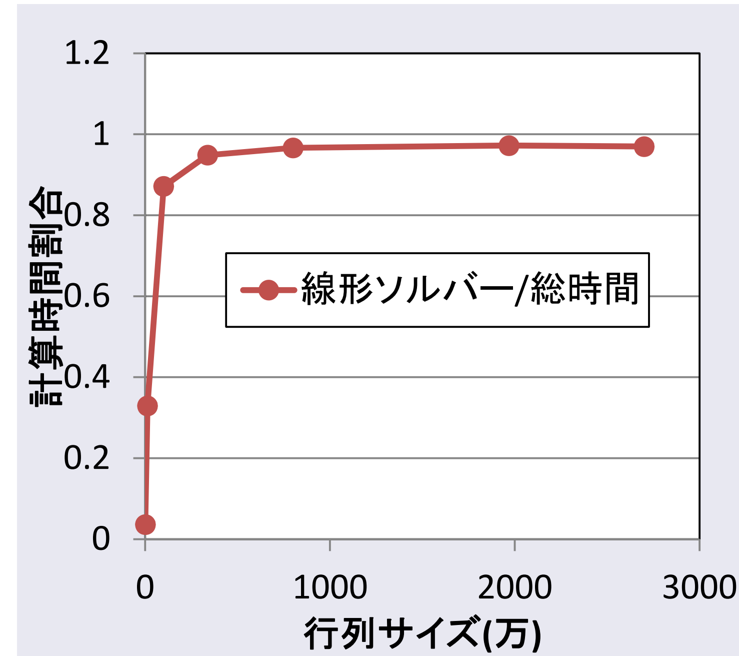


リファレンスソルバー計算時間

GAMGはサイズによって計算時間に振動や1コアとの逆転があるため
今回のリファレンスにはPCG,PBiCGソルバーを用いる

ベンチマーク結果



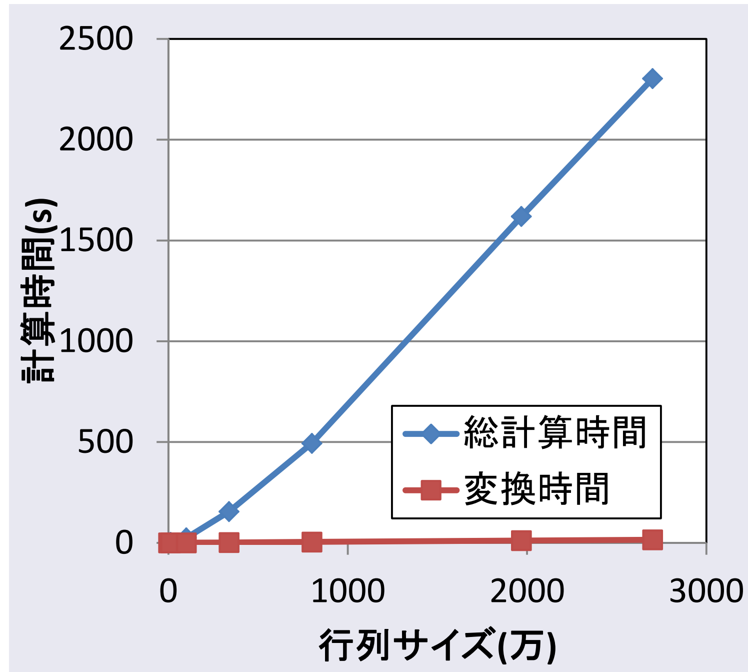


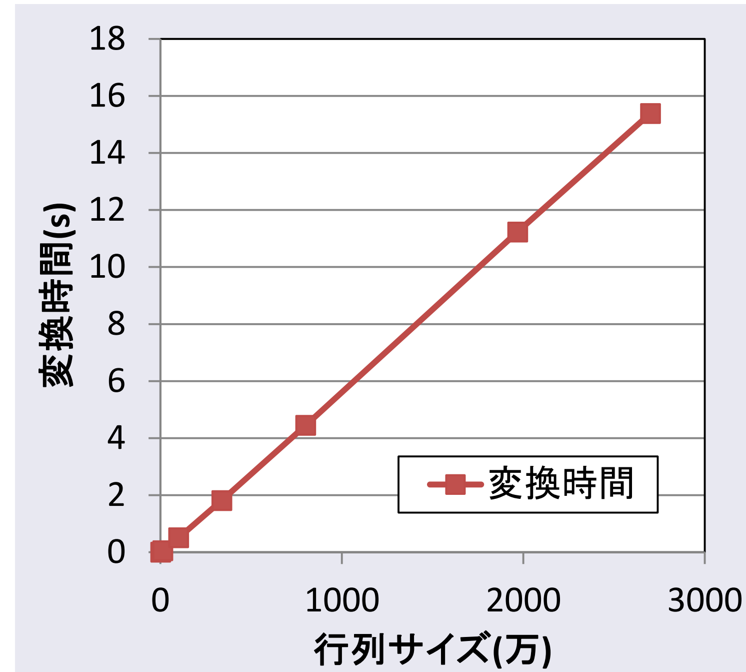
線形ソルバー計算時間割合
(PCG,PBiCG MPI12計算)

ファイルアクセス等含む総計算時間に占める
線形ソルバーの計算時間は95%以上

➡線形ソルバーさえ高速化されればOK

ベンチマーク結果





CSRフォーマットへの計算時間
(PCG,PBiCG MPI12計算)

OpenFOAM形式からCSR形式への変換時間は
ファイルアクセス等含む総計算時間のうちの1%未満程度

➡ **線形ソルバーが高速化しやすい形式を選択すればOK**

ベンチマーク結果

EQNソルバーシステム概要

OpenMP並列化

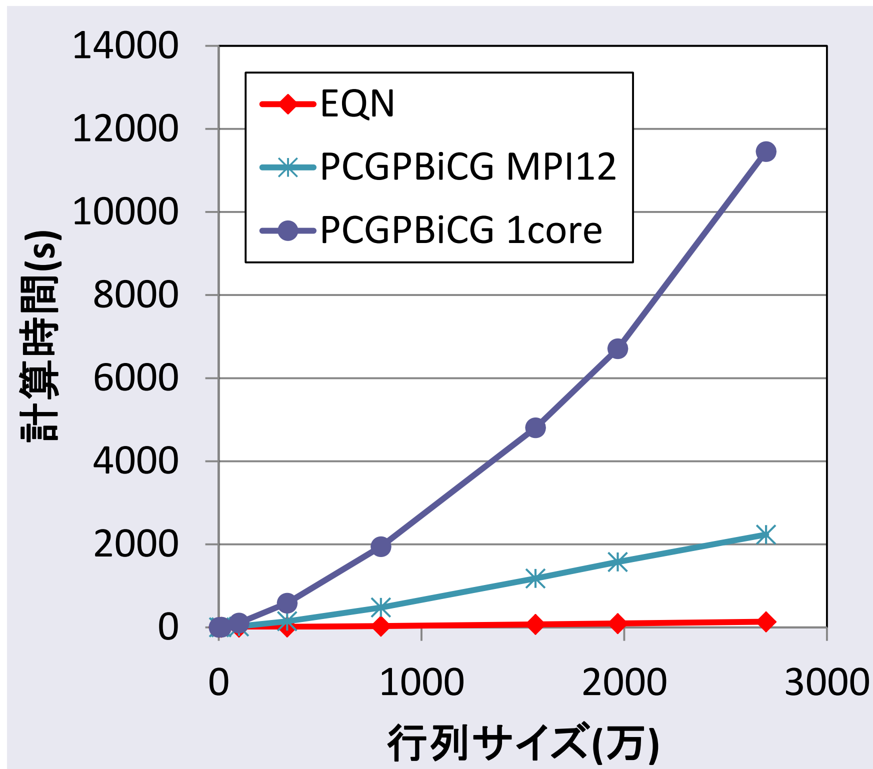
MPI並列化

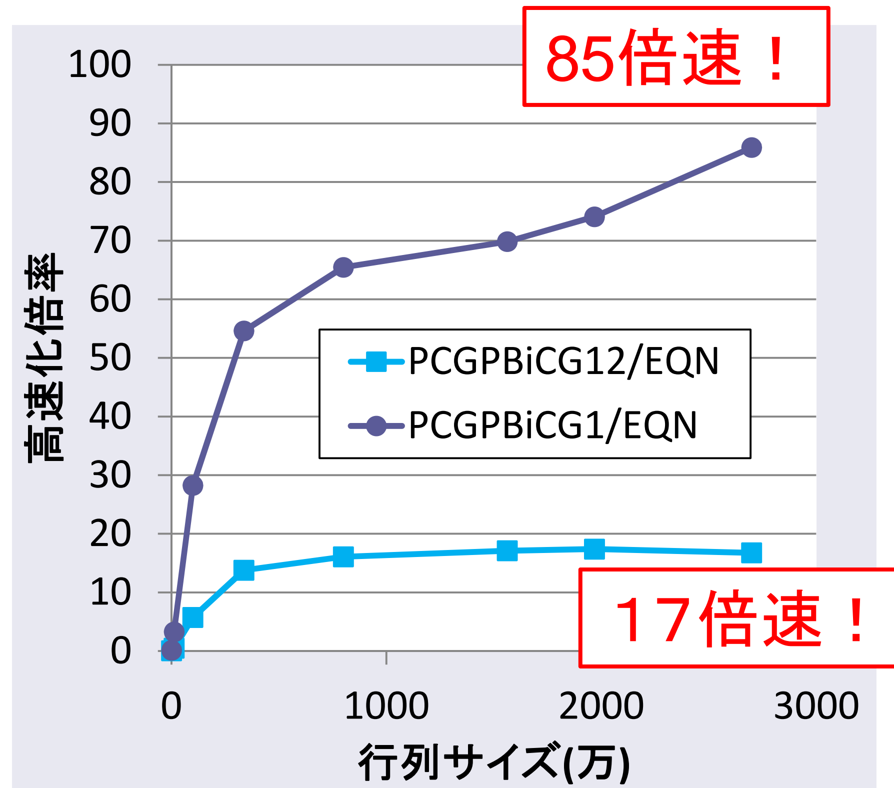
倍精度演算性能
1.7T Flops

GTX Titan

Xeon 4 Core

ベンチマーク結果

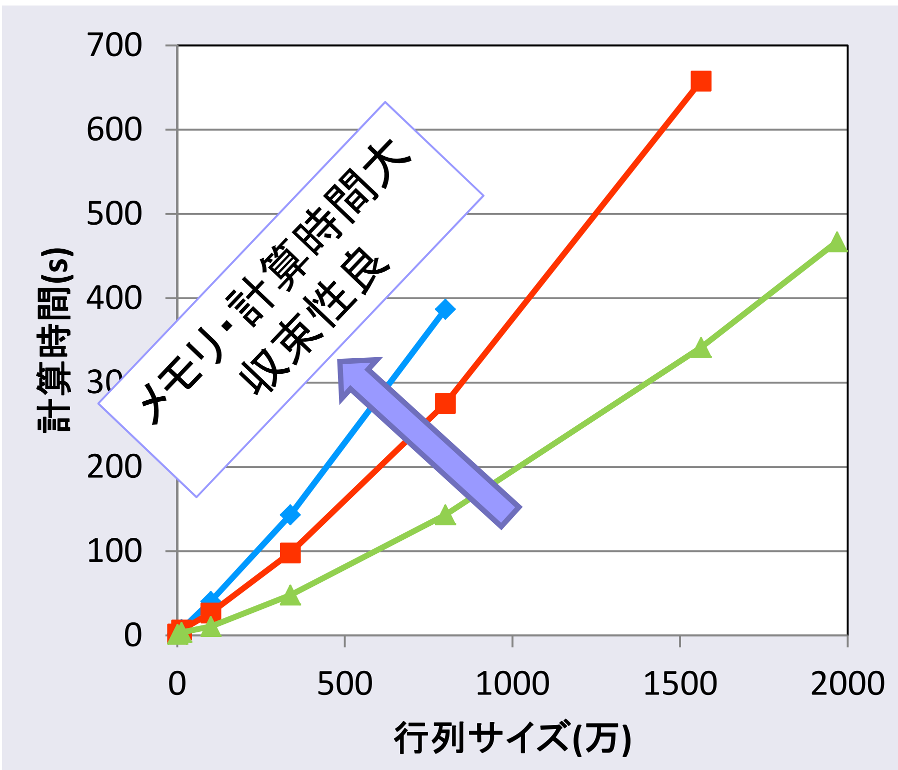




MPI+OpenMP+GPGPUソルバー高速化倍率

ベンチマーク結果

ILU(不完全LU分解)の
さまざまな並列化組み合わせ



ハイブリッド並列化により
色々な並列化の組み合わせで
リソースと収束性のトレードオフを
探すことができる

BiCG系の計算を1GPU
ILUの計算を1GPU

BiCG計算の
GPUへの割り当て

ILU計算の
GPUへの割り当て

BiCG系の計算を1GPU
ILUの計算を3GPU+12Core

ILU計算の
割り当て
ブロック分解し
GPU,CPUを
ロードバランス

BiCG計算の
GPUへの割り当て

BiCG系の計算を3GPU
ILUの計算を3GPU

BiCG計算の
GPUへの割り当て

ILU計算の
GPUへの割り当て

行列の計算部分割り当てイメージ

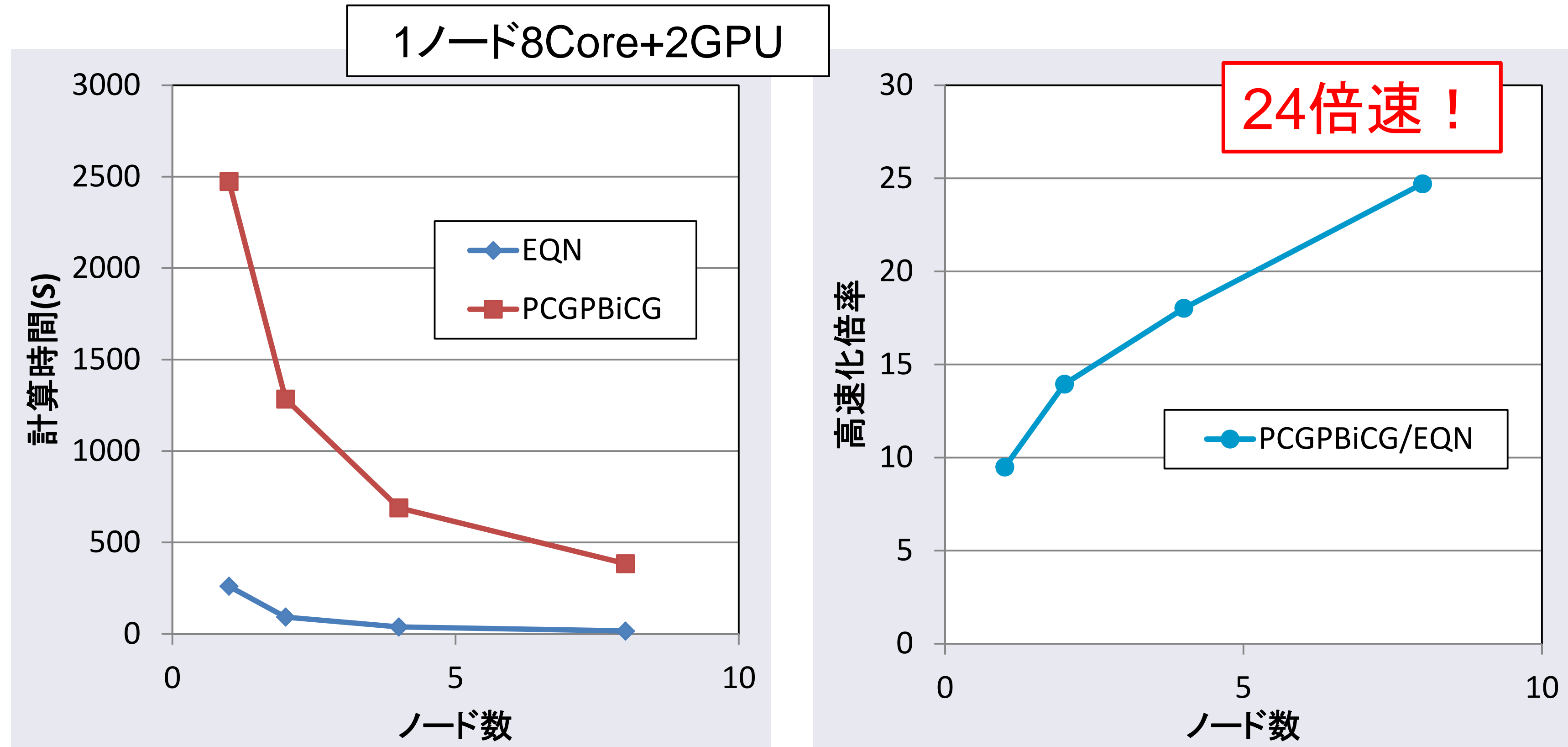
ベンチマーク結果

Amazon EC2 GPGPUクラスタースペック一覧

インスタンスタイプ	g2.2xlarg	cg1.4xlarge
CPU	Xeon E5-2670	Xeon X5570 ⇒今回採用
メモリ	15GiB	22.5GiB
ECUs	26	33.5
vCPUs	8	16
イーサーネット I/O	高い	非常に高い(10Gbps)
GPU	Grid K520*1	Tesla M2050*2
GDDR5	4096 MB	2687 Mb
CUDA Core	1536	448
GPU Clock	797 MHz	1147 MHz
Memory Clock	2.5 GHz	1546 MHz
単精度演算性能x1	1128 GFlops	585 GFlops
単精度演算性能x2		1155 GFlops
倍精度演算性能x1	**83 GFlops**	286 GFlops
倍精度演算性能x2		**585 GFlops**
料金(Virginia, RHEL)	**$0.780 / 1時間**	**$2.230/ 1時間**

nbodyベンチ
実測

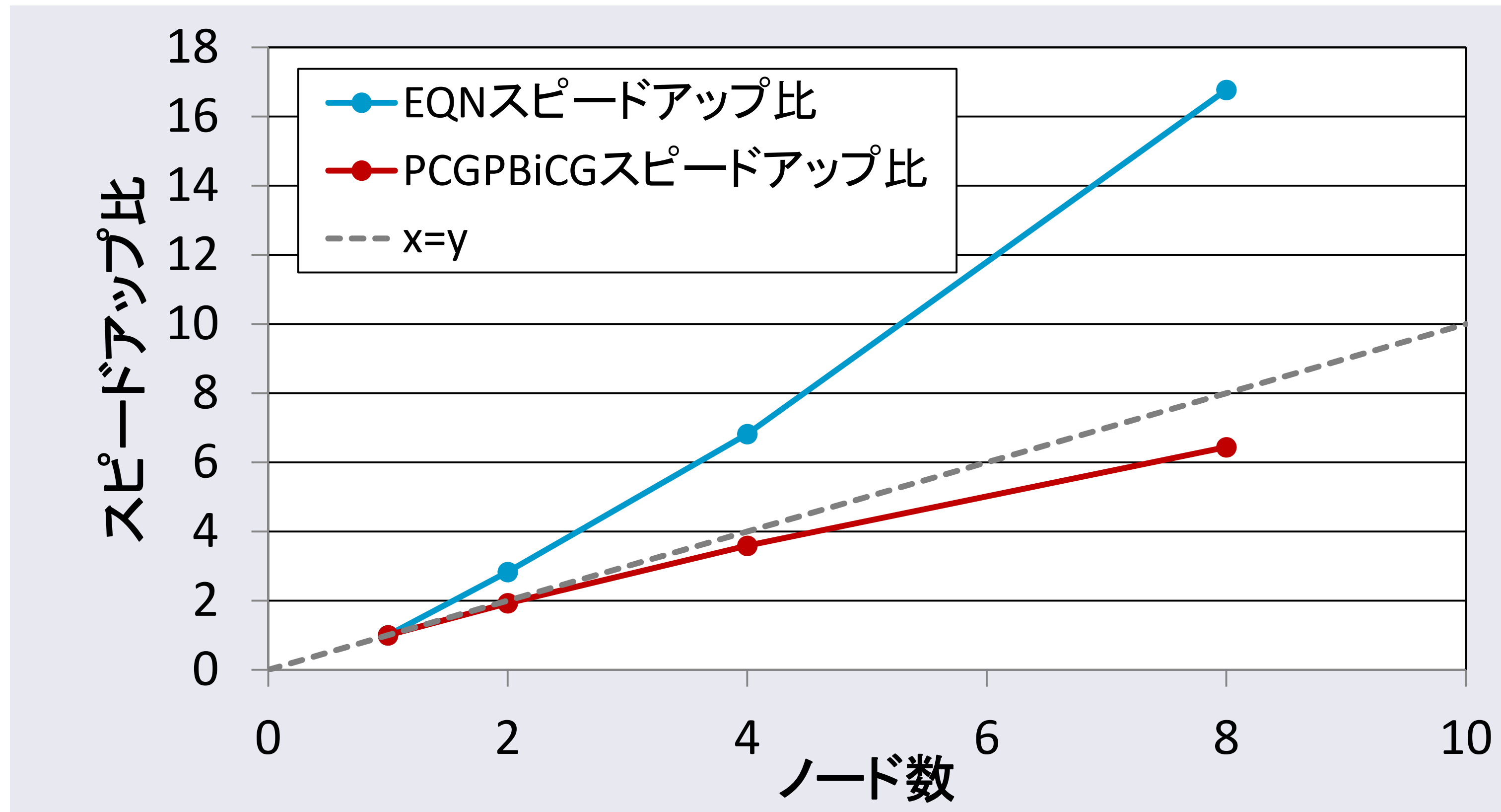
ベンチマーク結果



ハイブリッドソルバー高速化倍率(1500万メッシュ計算)

8ノード(64CPU Core)のPCG,PBiCG MPI計算に対して
8ノード(16GPU+64CPU Core) ハイブリッド並列化ソルバーが**24倍速**

ベンチマーク結果



スピードアップ比の比較(1500万メッシュ計算)

スピードアップ比＝1ノード計算時間/Nノード計算時間

まとめ

- OpenFOAMでの線形行列ソルバーの差し替えは容易
- 行列形式の変換は低コスト
- ハイブリッド並列の分解手法は問題に合わせてさまざまに変更可能
- GPGPUを組み合わせたハイブリッド並列化は**極めて効果的**