OpenFOAMを用いた
超大規模計算モデル作成とその性能の評価

清水建設 株式会社

PHAM VAN PHUC

内山　学

「京」でのOpenFOAMに関する取組み

- 第1回　OpenFOAMワークショップ (2013)
  - コード移植、10億格子計算の壁・解決策（プリ・ポスト）
- 第2回　OpenFOAMワークショップ（2014）
  - 1万並列計算の壁・解決策（MPIプラットフォーム）

コードの課題・改良策

- 第3回　OpenFOAMワークショップ (2015)
  - 超並列・超大規模解析（10万並列、1千億格子）
- 第4回　OpenFOAMワークショップ（2016）
  - 超大規模ポスト処理（1千億格子）

改良コードの超大規模解析・ポスト処理

- 第2回CAEワークショップ（本日）
  - **超大規模計算モデル作成とその性能の評価**

⇩

超大規模プリ処理＋最新改良・性能分析

内容

- 超大規模のプリ処理　（フック）
  - メッシュの作り方
- コードの最新改良と性能評価　（内山）
  - OpenFOAMのthread 並列化

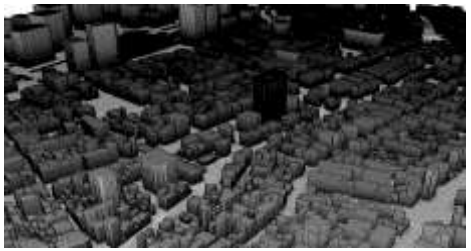
超大規模のプリ処理

プリ・ポスト処理

数億格子規模でほぼ限界

- シリアル処理の限界
  - データの分割・結合には時間・手間は非常にかかる

10億格子データ



1TBメモリの利用

京のプリ・ポスト処理
PC：1TB

モデル作成（プリ処理）

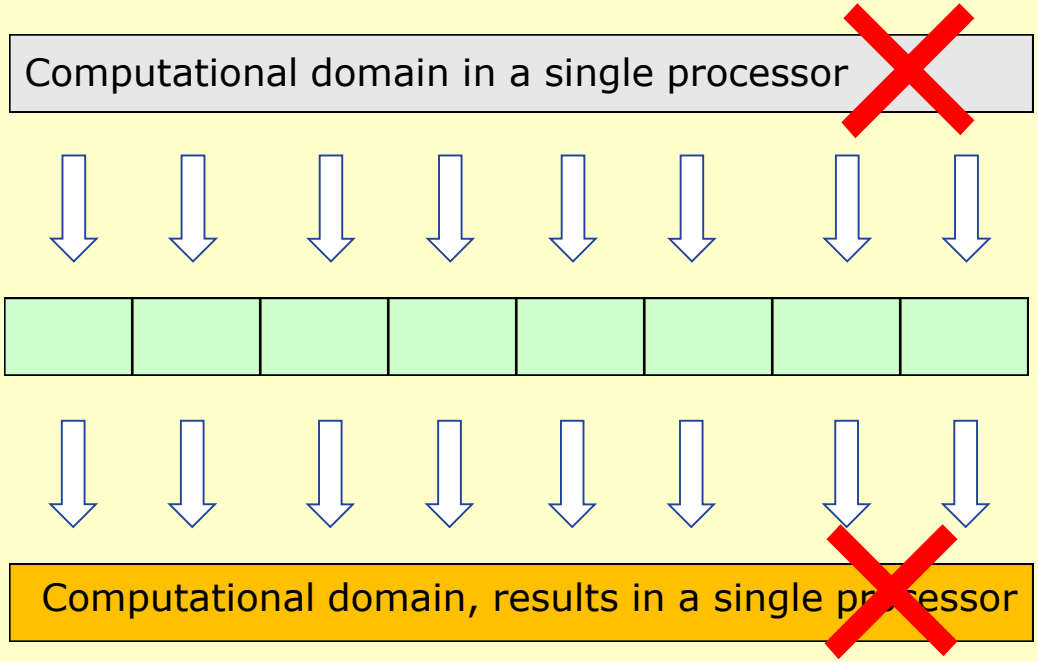
初期化

領域分割

シミュレーション

データ結合
ポスト処理

出力・可視化



プリ・ポスト処理

- 分散処理の重要性

本日、紹介

モデルの作成（プリ）

初期化・領域分割（分散）

・ロードバランス

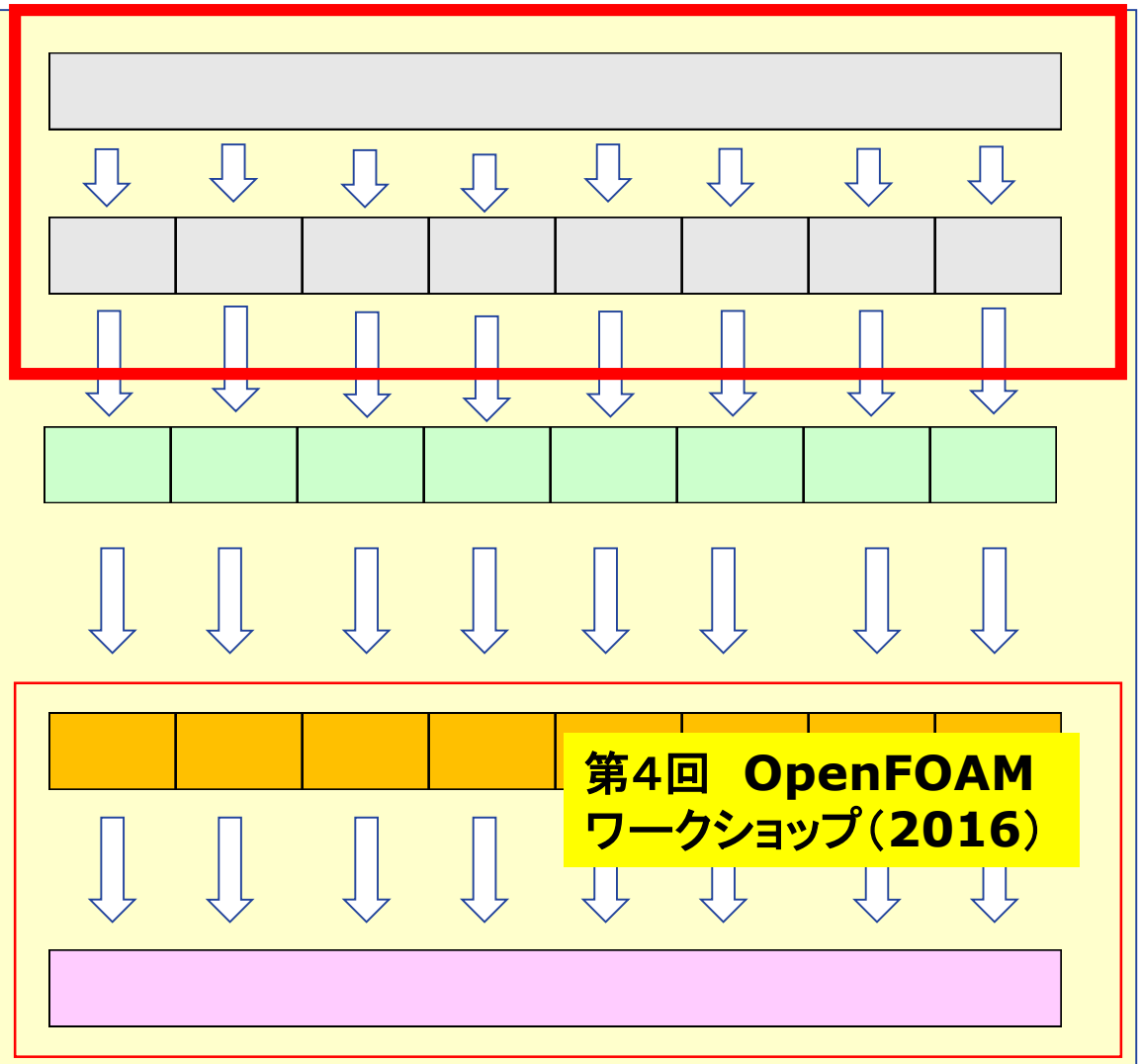
シミュレーション

ポスト処理

データ出力・可視化

データ処理等

画像処理



**第4回　OpenFOAM
ワークショップ（2016）**

大規模プリ処理手法の分類

1.細分化の手法

- 表面の比較的大きいモデル
  （粗い格子で計算格子を作成できる）

2.分散処理の手法

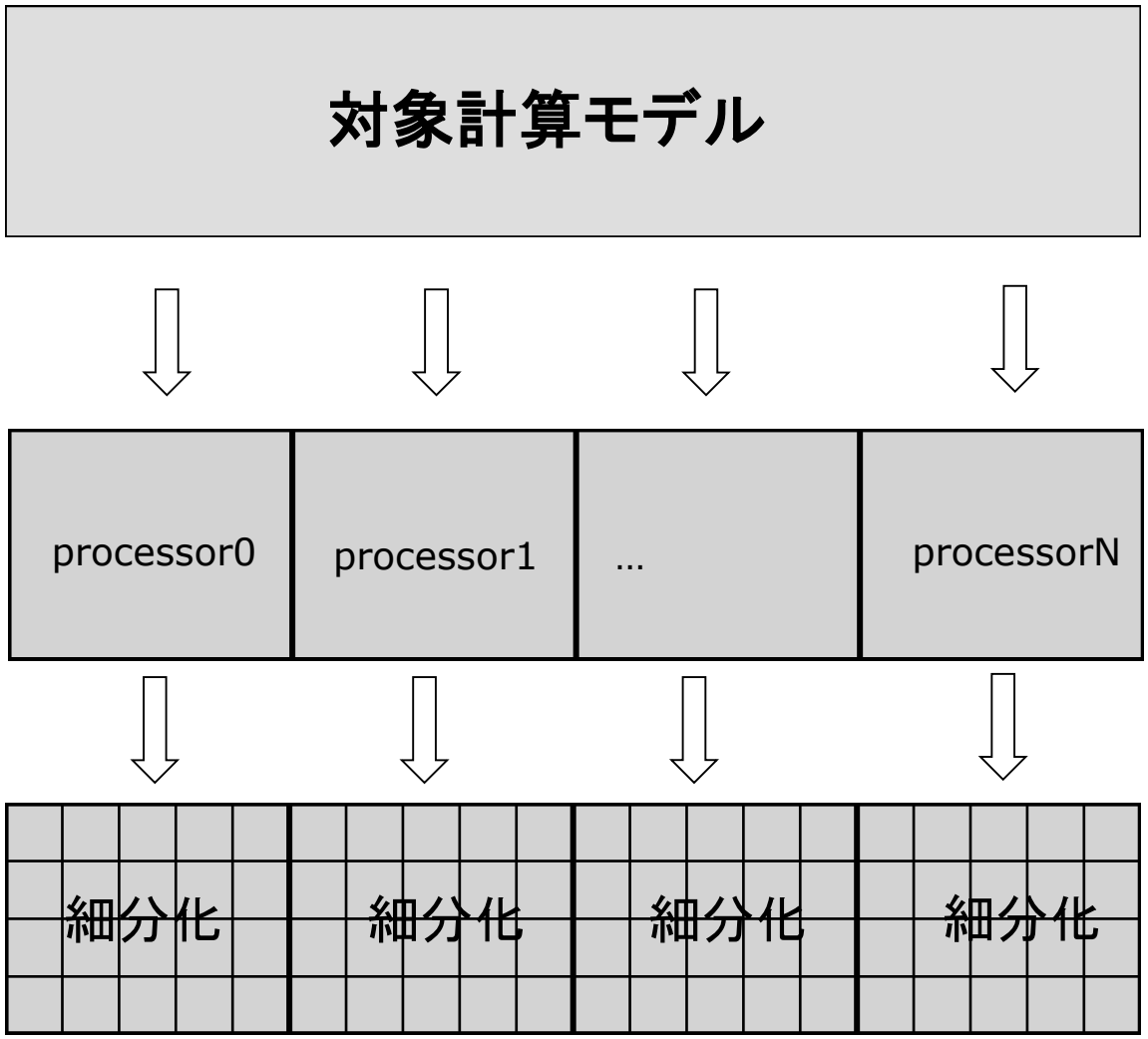
- 独立性のあるモデル（小規模の複数ケースの計算）

3.マルチカラー処理の手法

- 少ない計算リソースのある場合

1. 細分化の手法

手法のコンセプト



1）数千万の粗い格子でのメッシュの作成
(blockMesh/SnappyHexMesh
/市販メッシュ作成ソフト)

2）領域分割
(decomposePar)

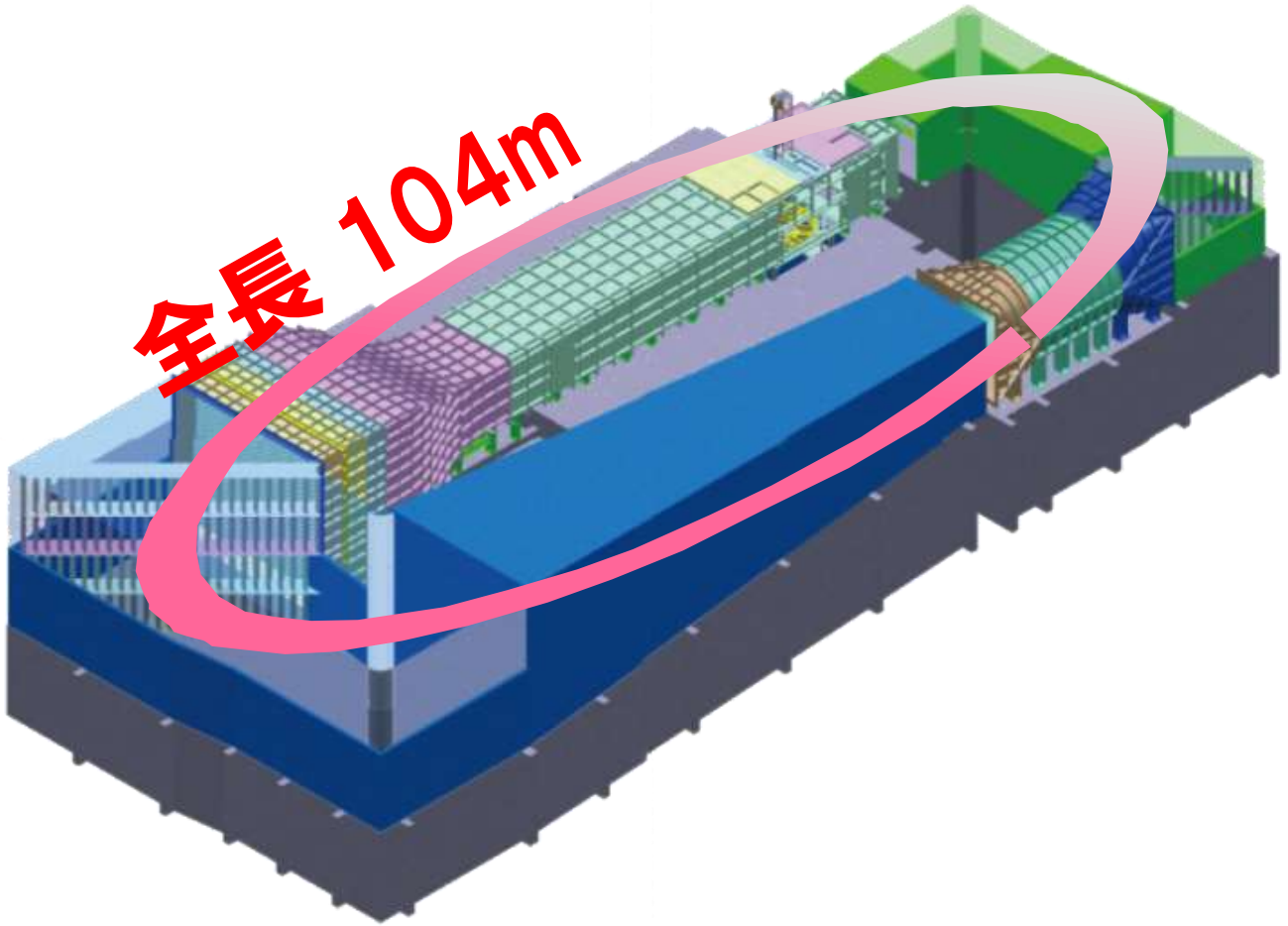
・ロードバランス

3）計算格子の細分化
(refineHexMesh/refineMesh)

大規模計算モデル
（数十～数百億格子）

1. 細分化の手法

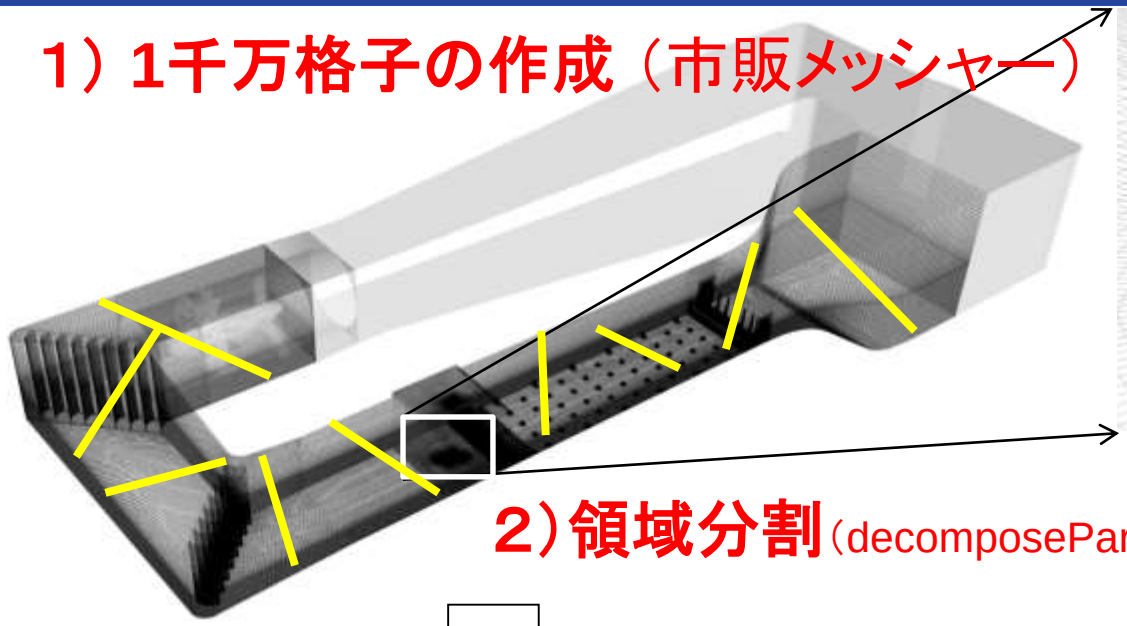
風洞丸ごとの再現計算事例

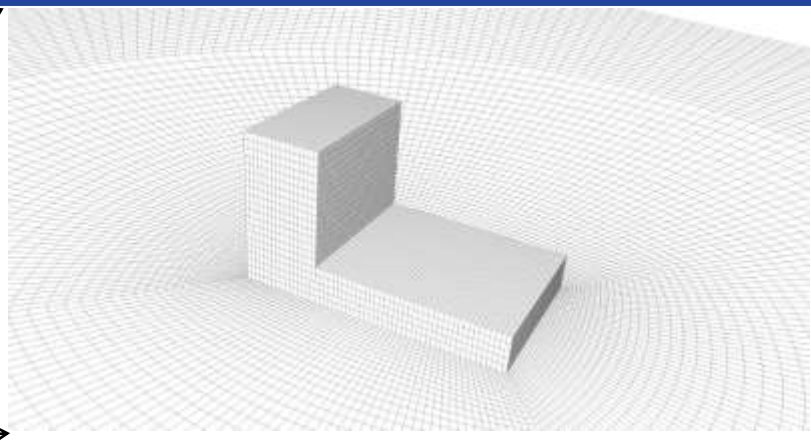


1. 細分化の手法

風洞丸ごとの再現計算事例
（イメージ図）

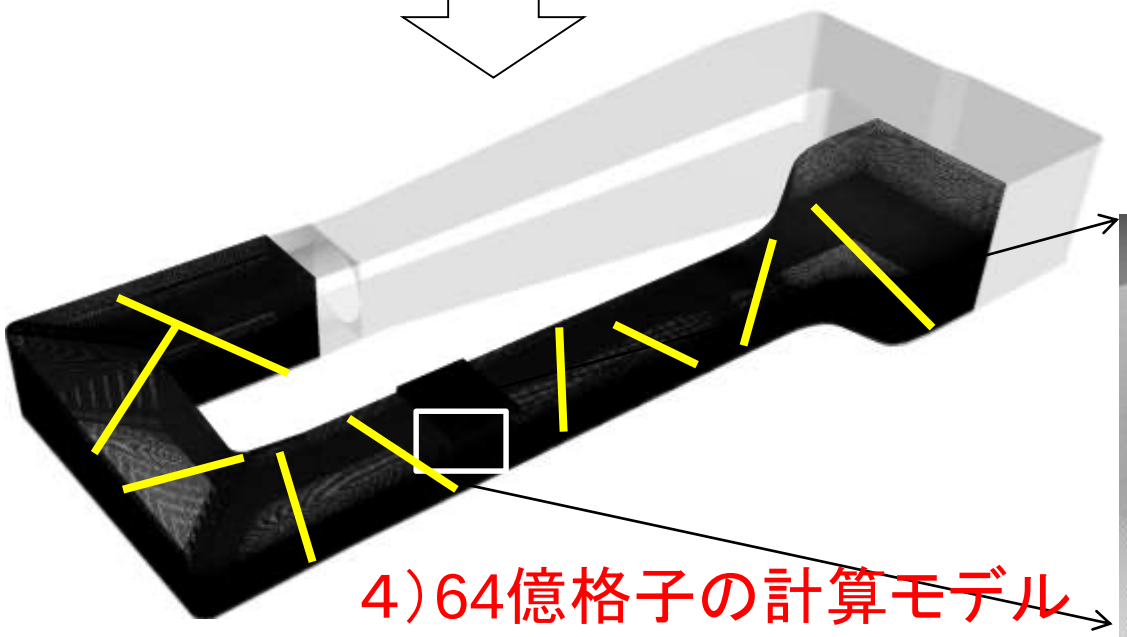
1）1千万格子の作成（市販メッシャー）

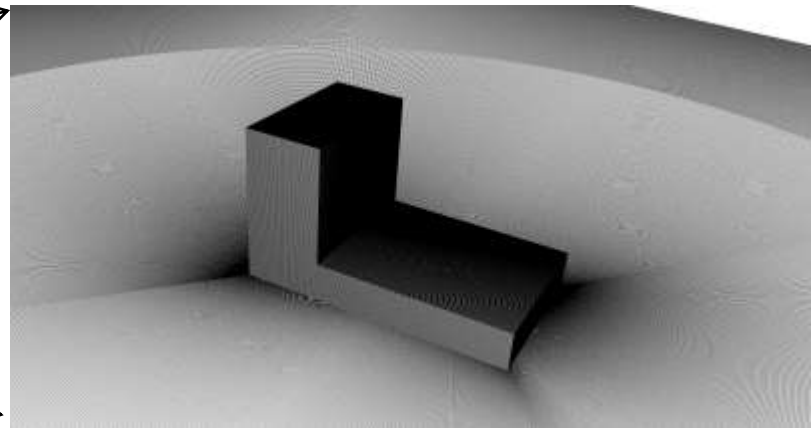




2）領域分割（decomposePar: ロードバランス）

3）計算格子の細分化
(refineHexMesh/refineMesh)

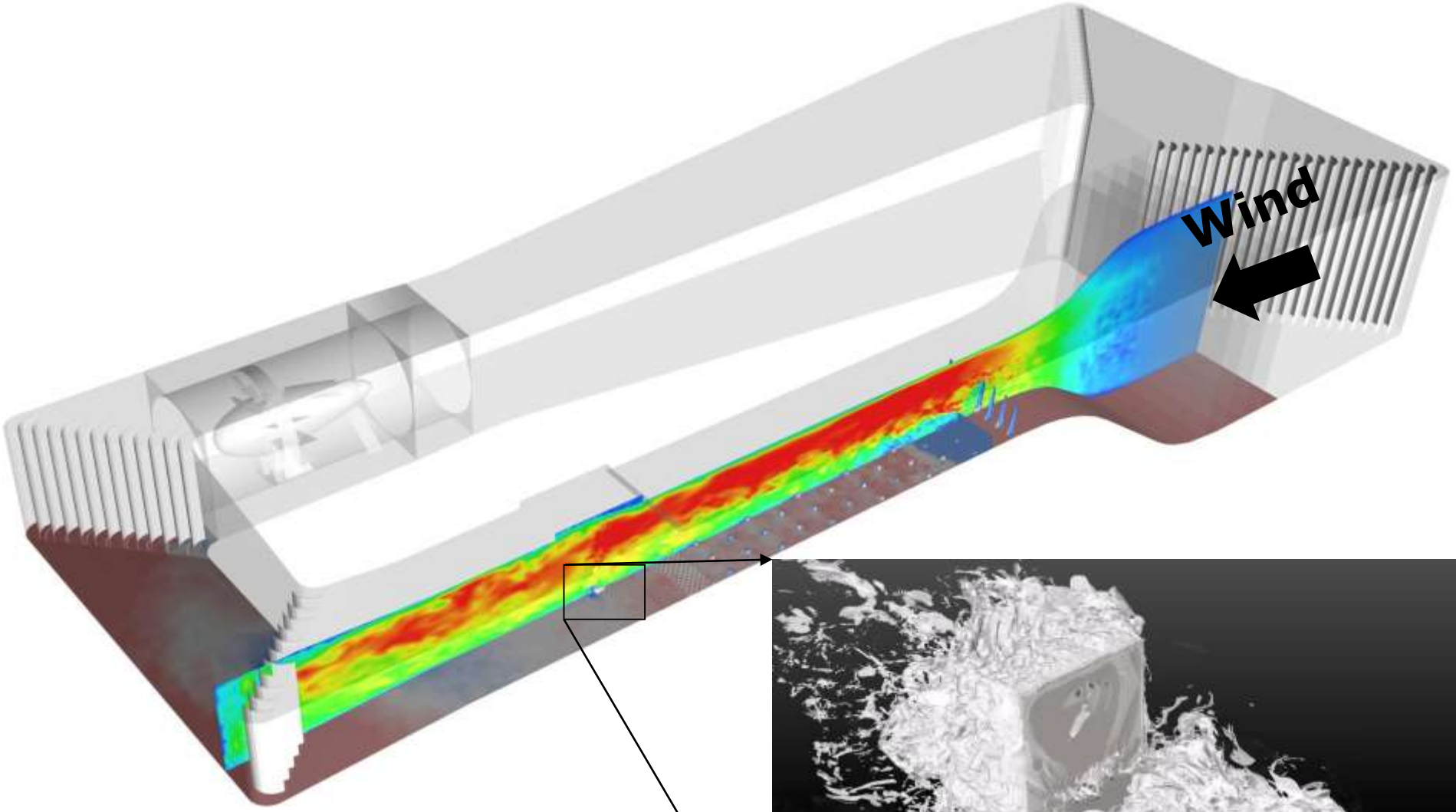




4）64億格子の計算モデル

1. 細分化の手法

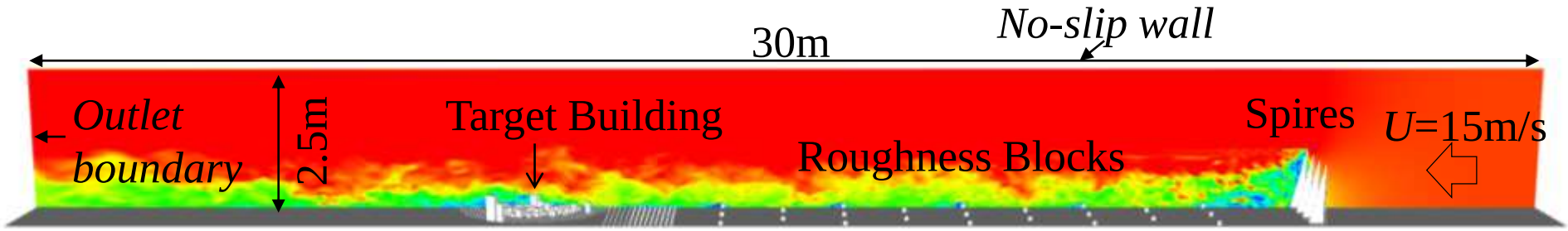
風洞丸ごとの再現計算事例



大規模計算（64億格子）

1. 細分化の手法

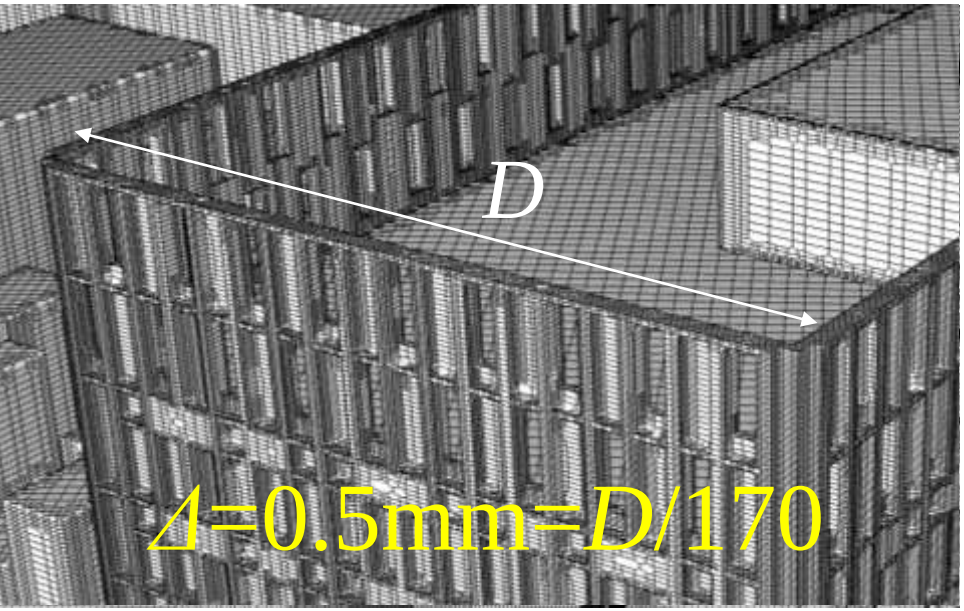
複雑形状建物の計算事例

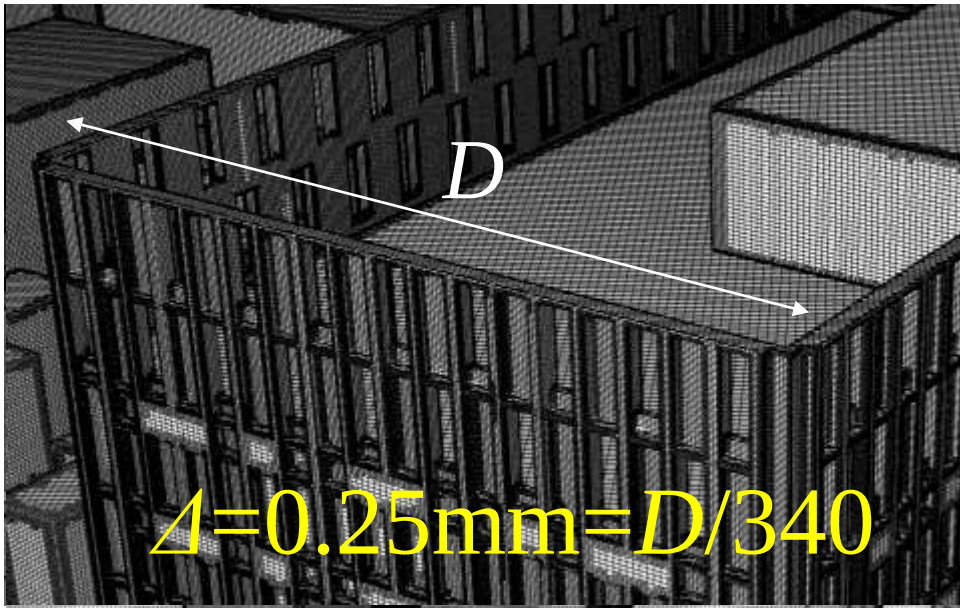


1) SnappyHexMeshでのメッシュ作成

2) 領域分割（6144領域：ロードバランス）

3）計算格子の細分化



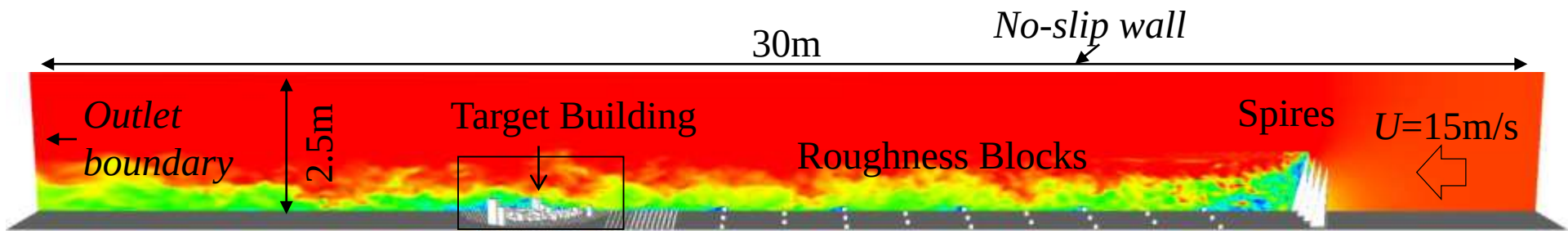


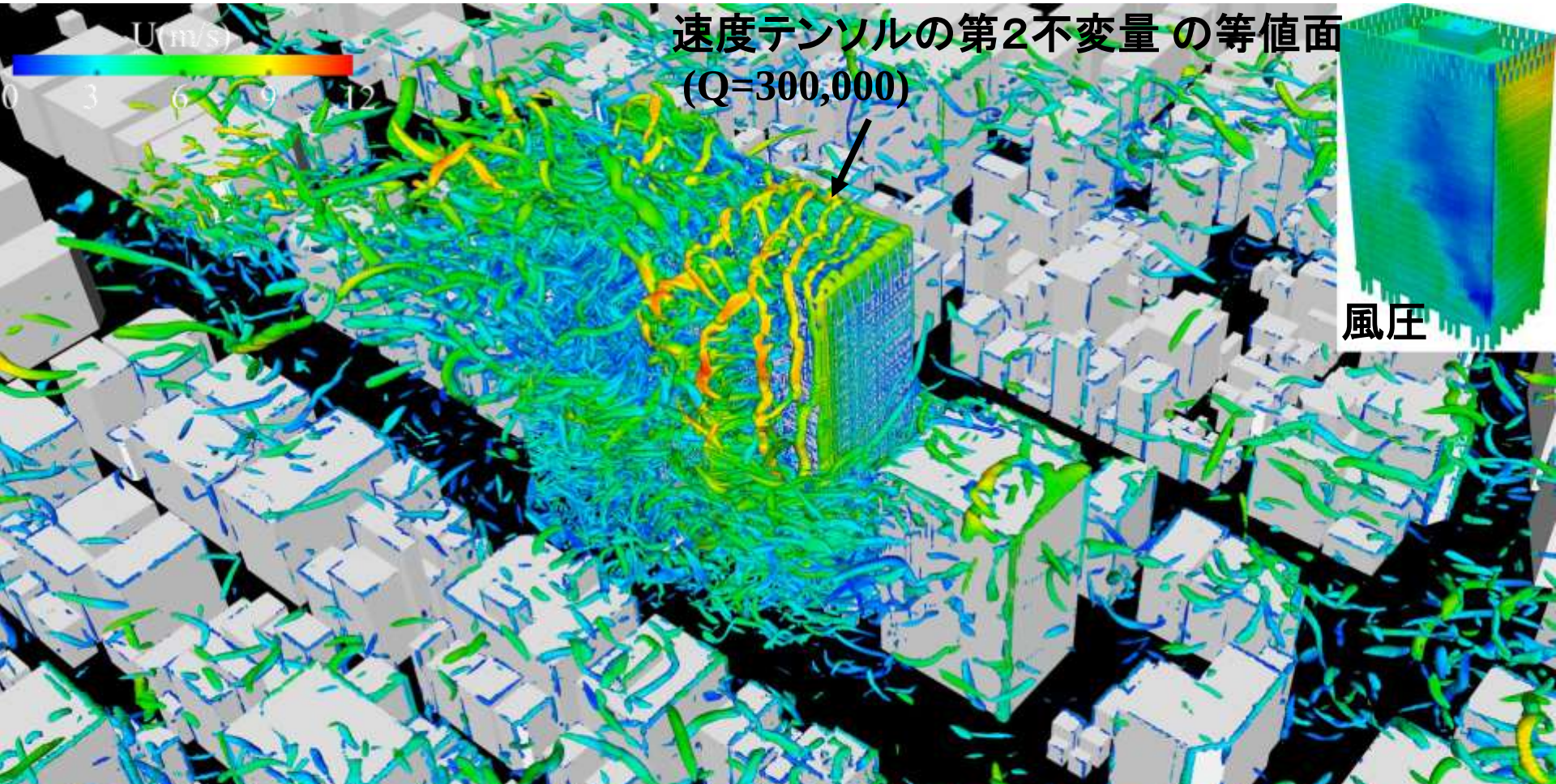
格子 A: 1.4億格子

格子 B: 11億格子

1. 細分化の手法

複雑形状建物の計算事例



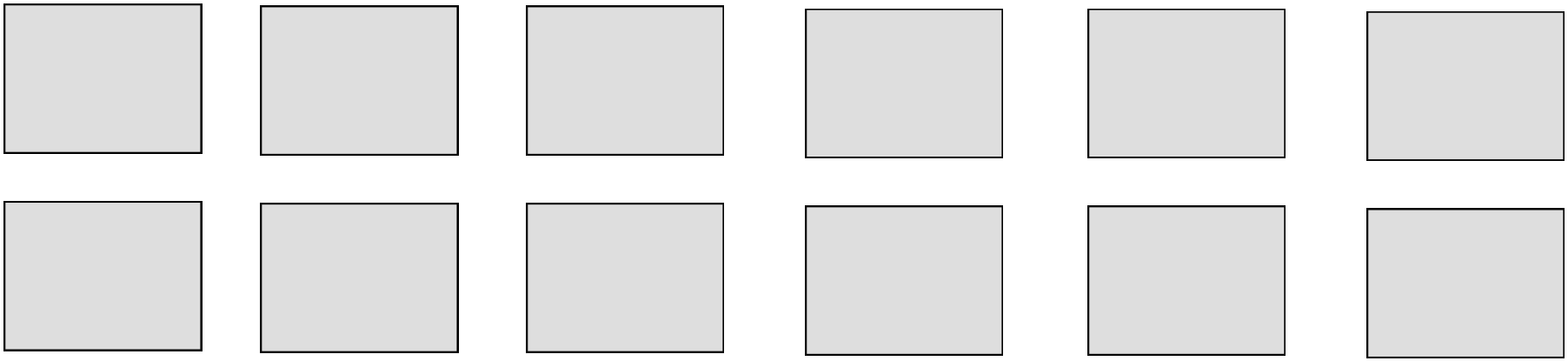


2. 分散処理の手法

手法のコンセプト

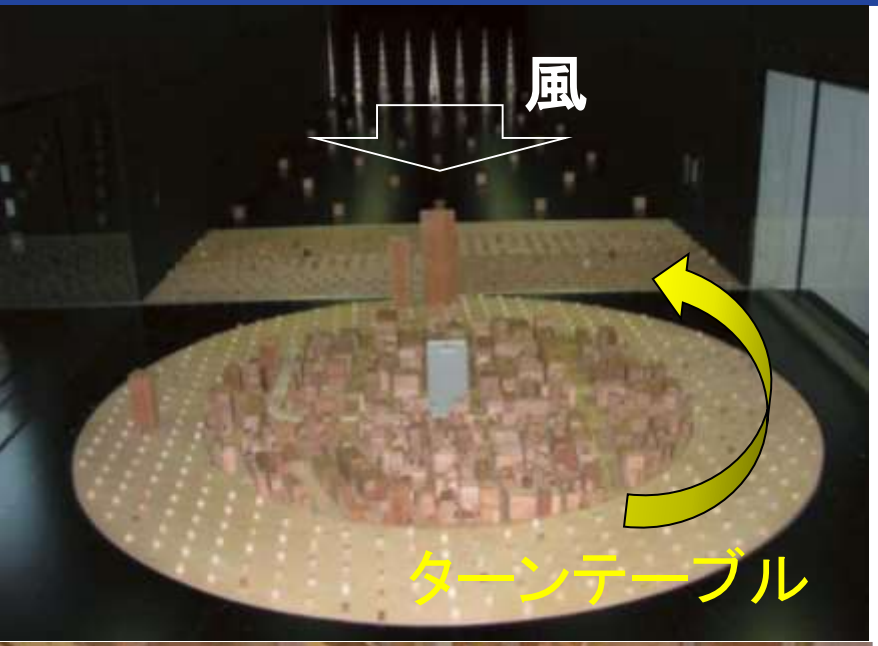
（独立性のある）対象計算モデル

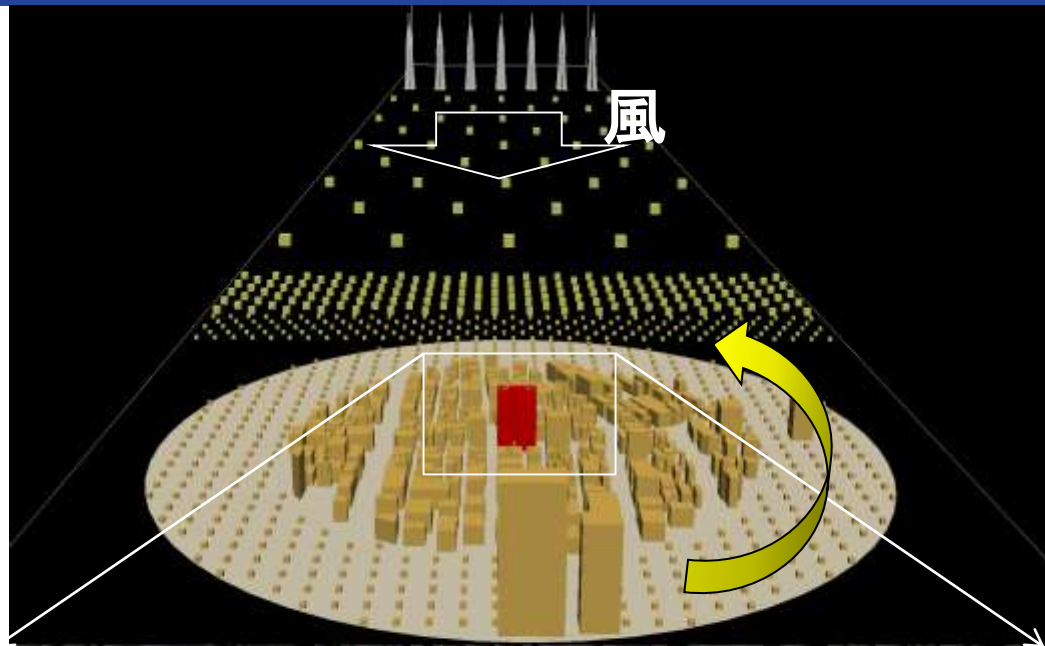
～　同様の小規模の複数ケースの計算



2. 分散処理の手法

36風向の風洞実験の再現
独立性のある計算モデル

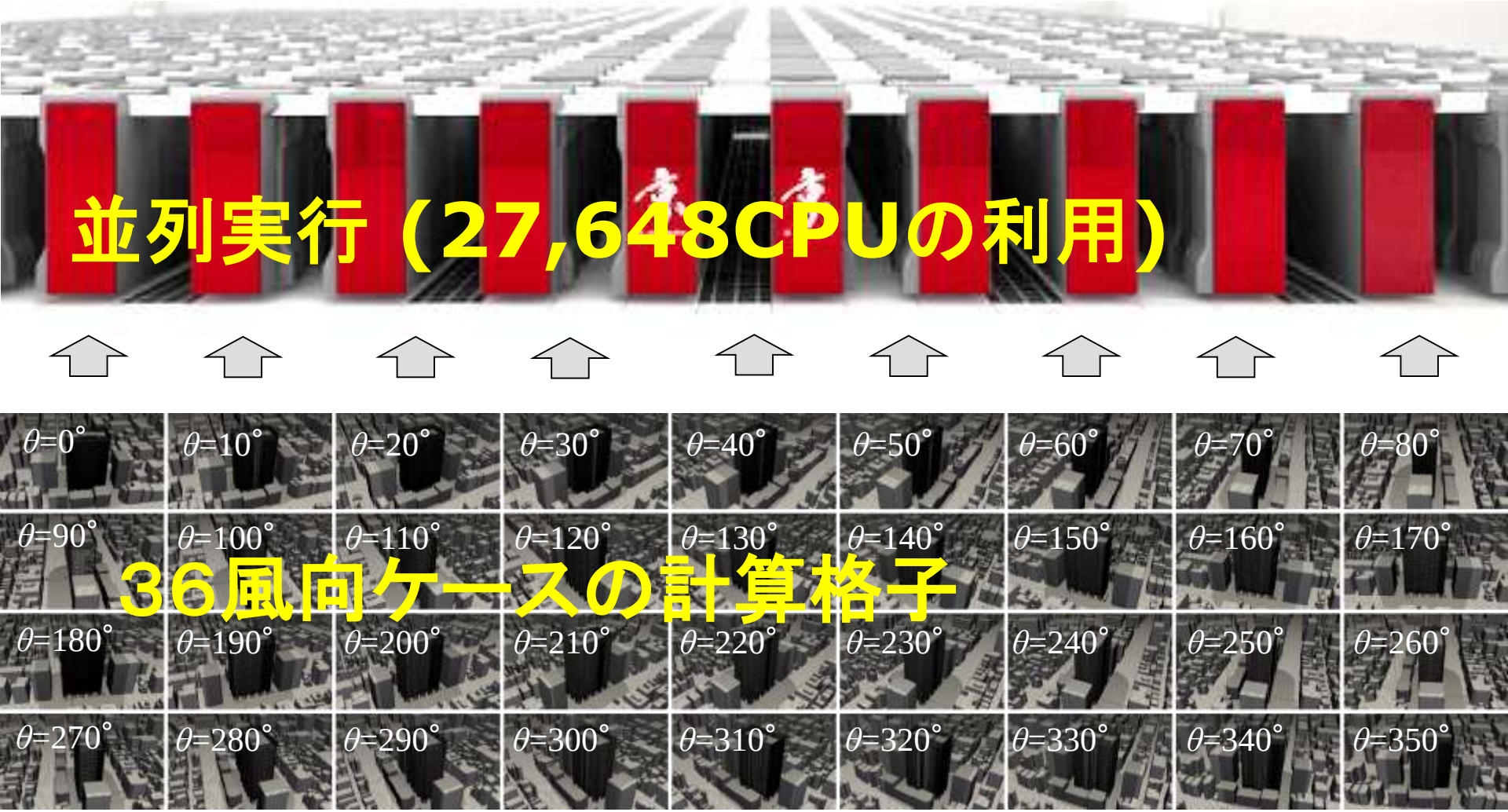




2. 分散処理の手法

36風向の風洞実験の再現
独立性のある計算モデル

「京」コンピュータ



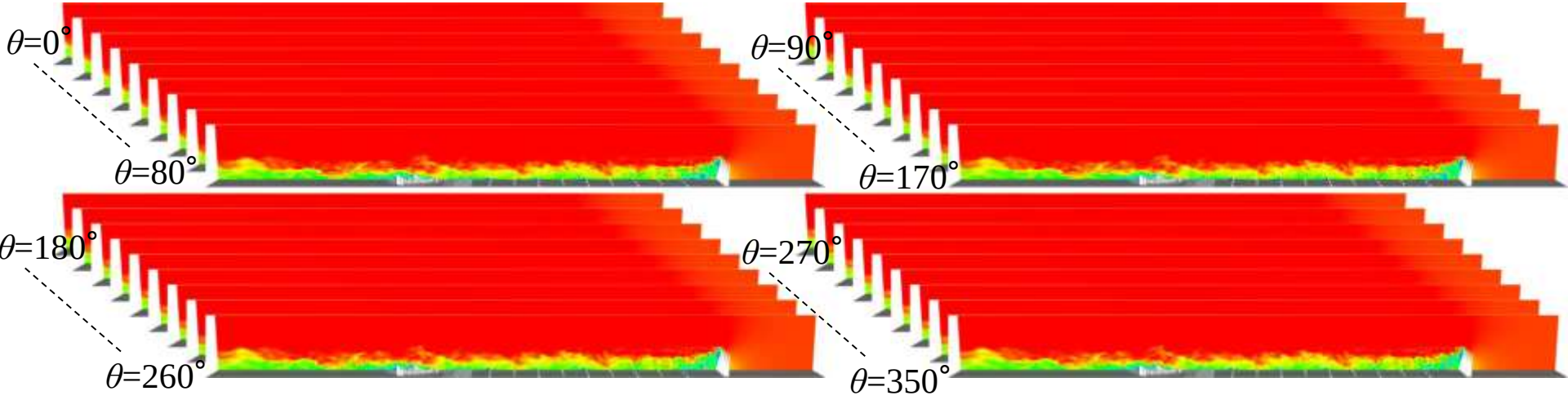
総格子数：1.4億 x 36風向 ~ 50億格子

2. 分散処理の手法

36風向の風洞実験の再現
独立性のある計算モデル

「京」コンピュータ

36風向ケースの並列実行

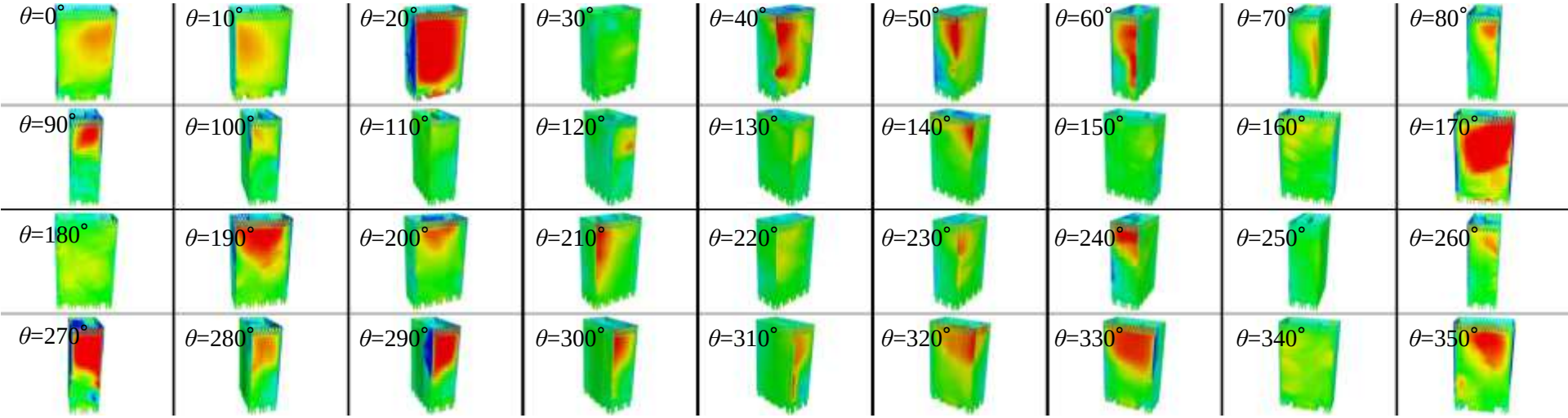


2. 分散処理の手法

36風向の風洞実験の再現
独立性のある計算モデル

「京」コンピュータ

36風向ケースの並列実行

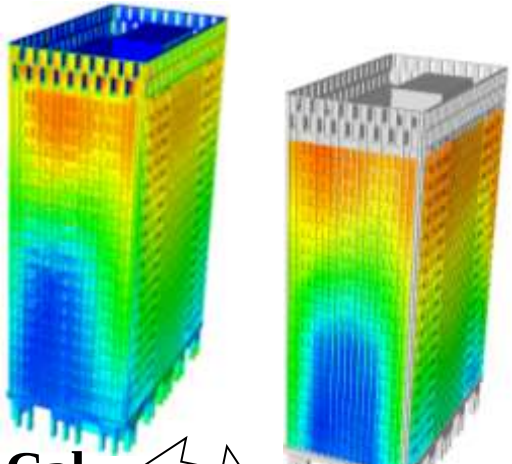


風圧分布の算出

2. 分散処理の手法

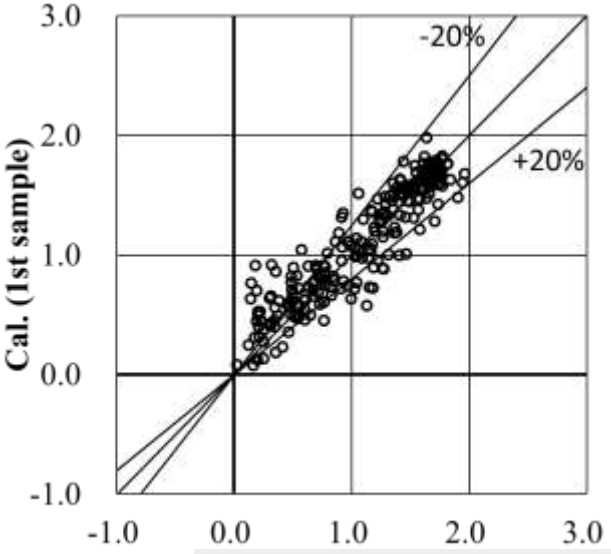
36風向の風洞実験の再現
独立性のある計算モデル

「京」コンピュータ

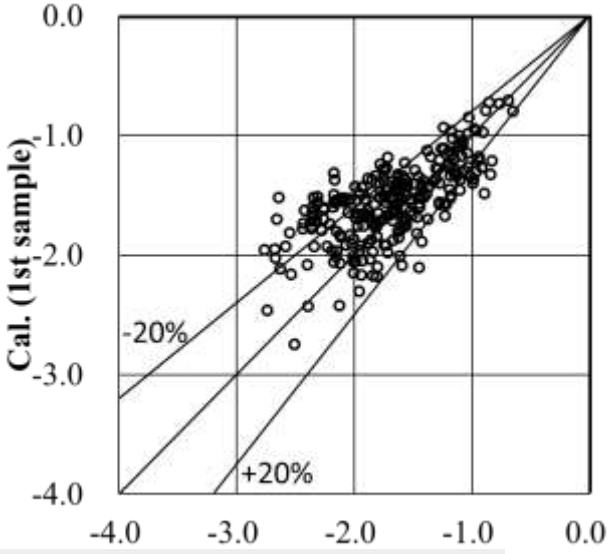


全風向における
ピーク外圧係数

正側ピーク外圧係数



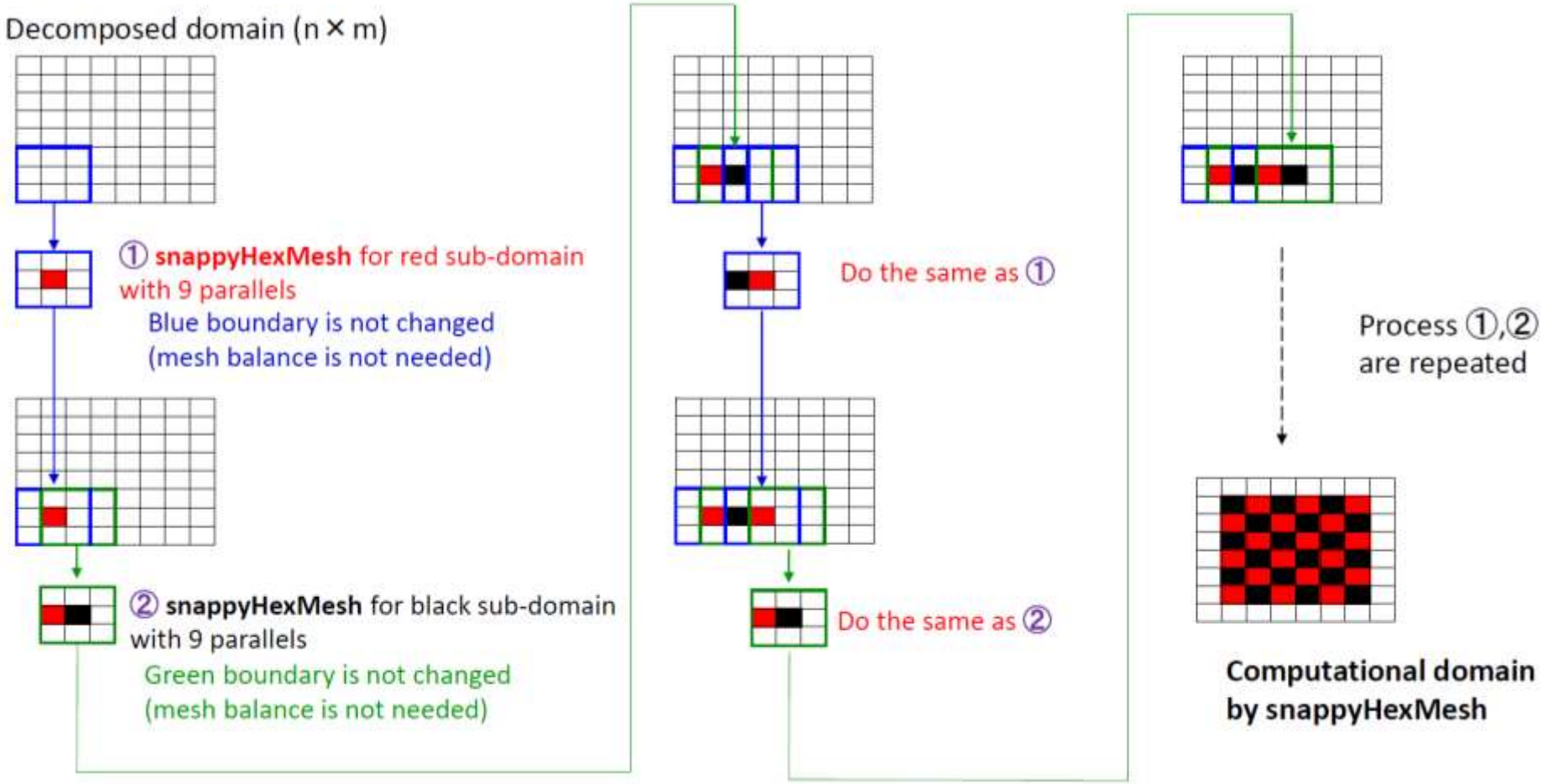
負側ピーク外圧係数



実験結果と同等な精度を確認

3.マルチカラー処理の手法　手法のコンセプト

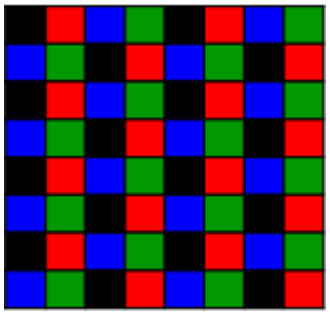
少ない計算リソースのある場合の対応　（既存ツールの改良）



※9並列処理、Red-Blackの2色処理法

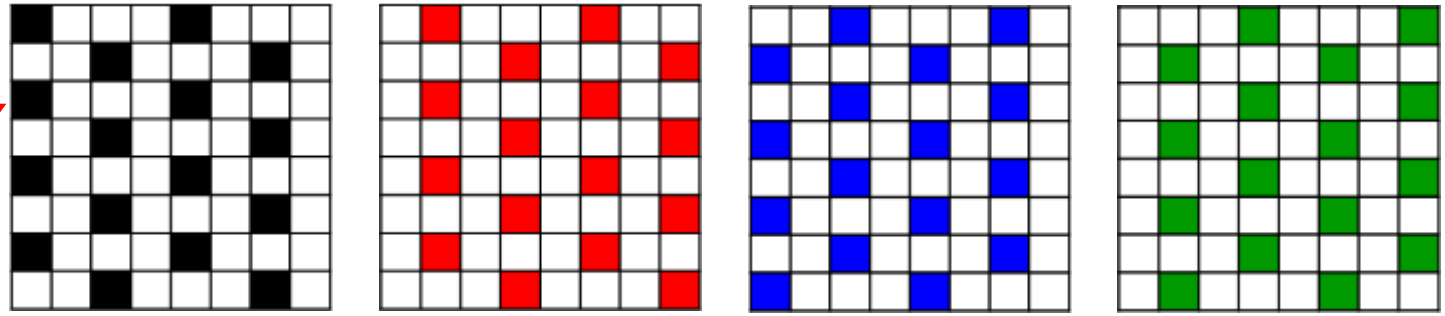
3.マルチカラー処理の手法　手法のコンセプト

少ない計算リソースのある場合の対応　（既存ツールの改良）

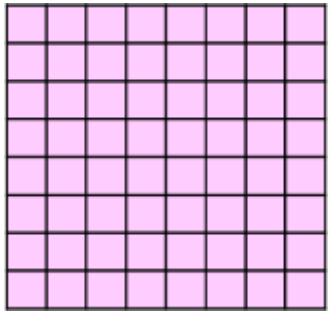


8×8=64領域
4色で色付け（黒、赤、青、緑）

独立で
メッシュを作成



4ステップで、順次に処理

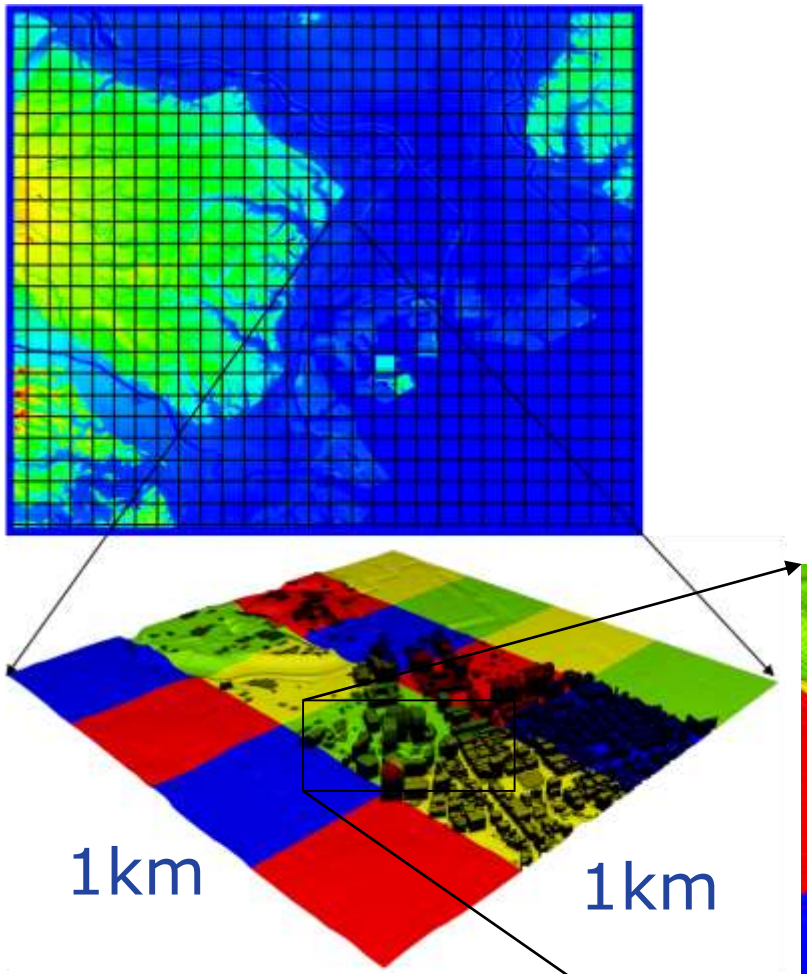


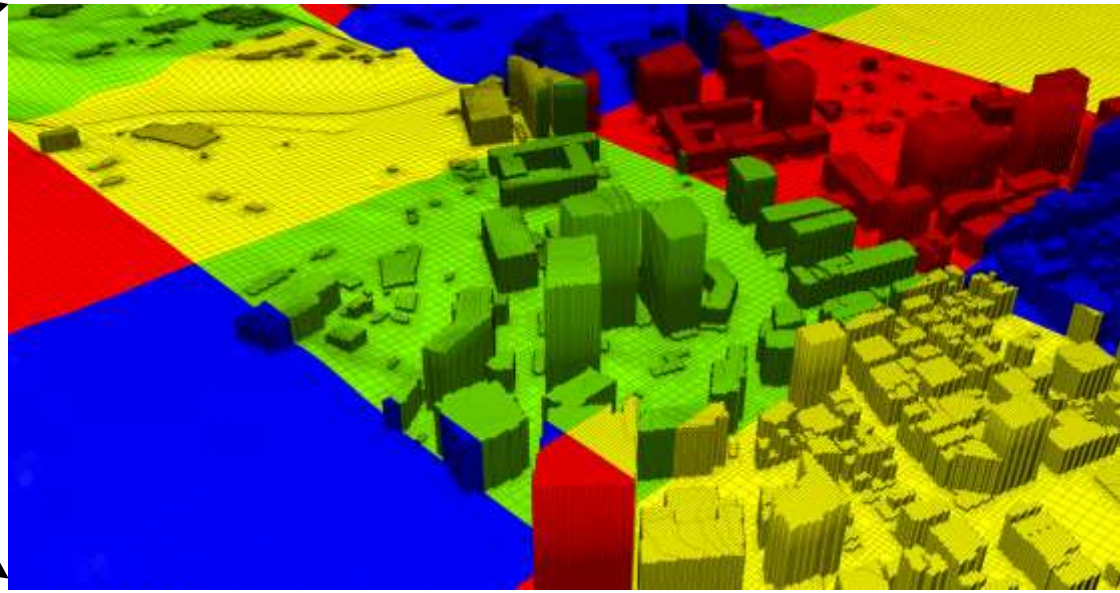
全領域のメッシュ作成完了

空いている
少ない計算リソースの利用

3.マルチカラー処理の手法

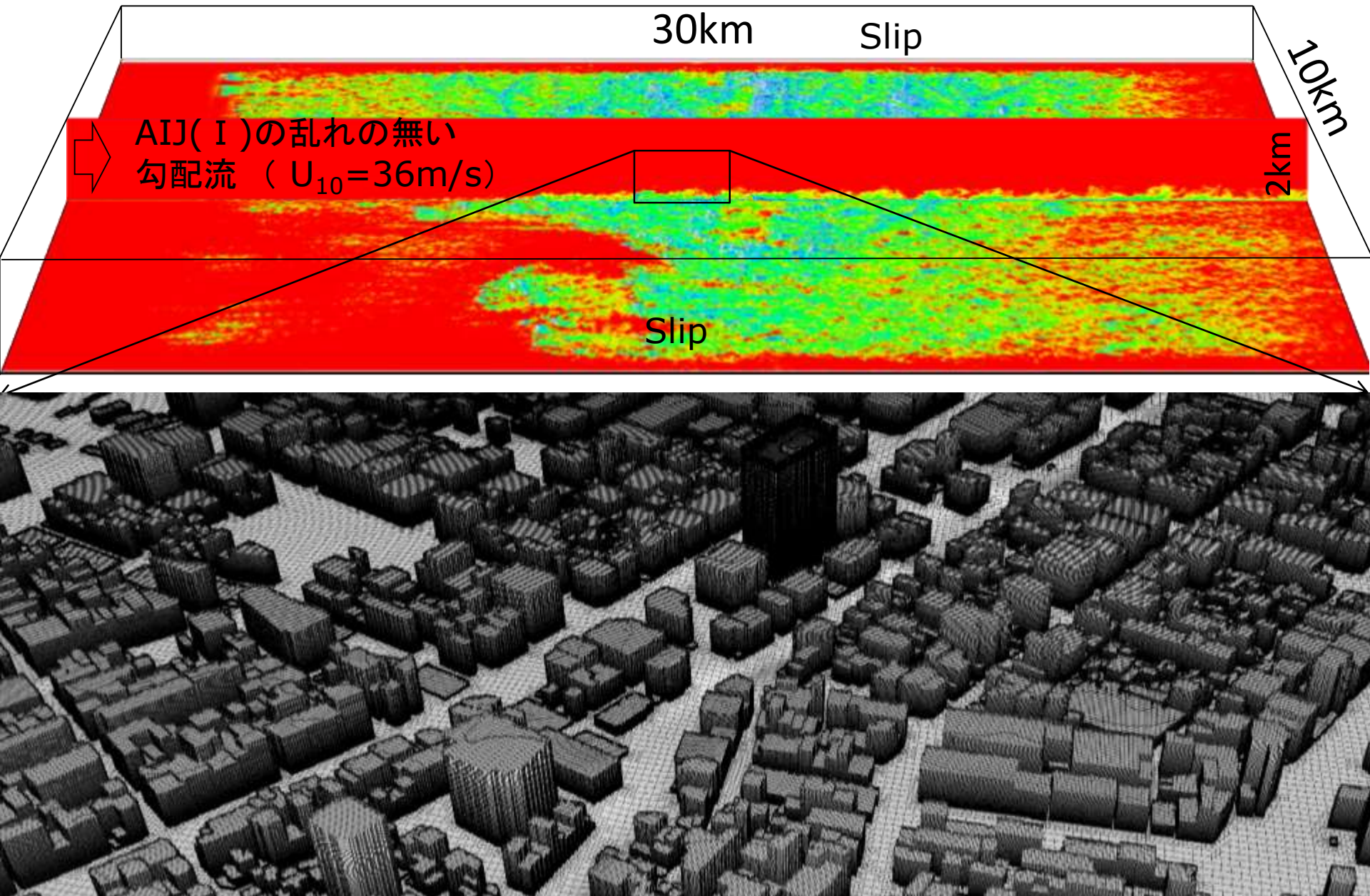
計算格子の作成





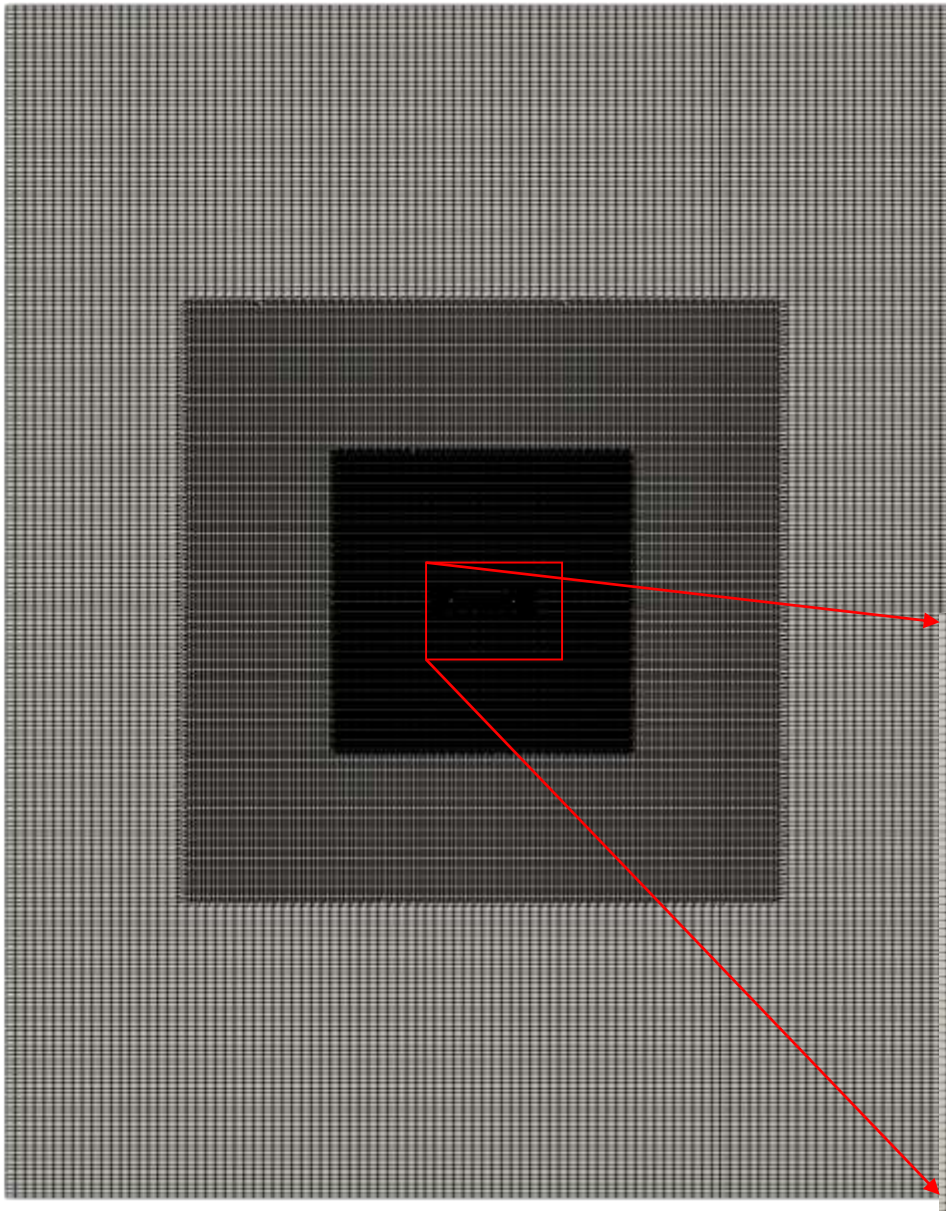
3.マルチカラー処理の手法

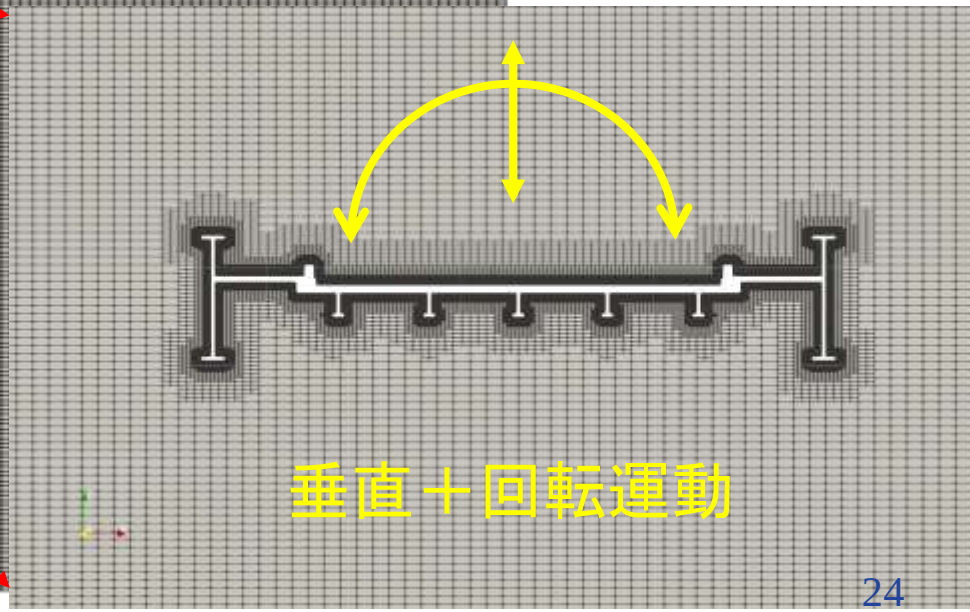
計算事例



3.マルチカラー処理の手法

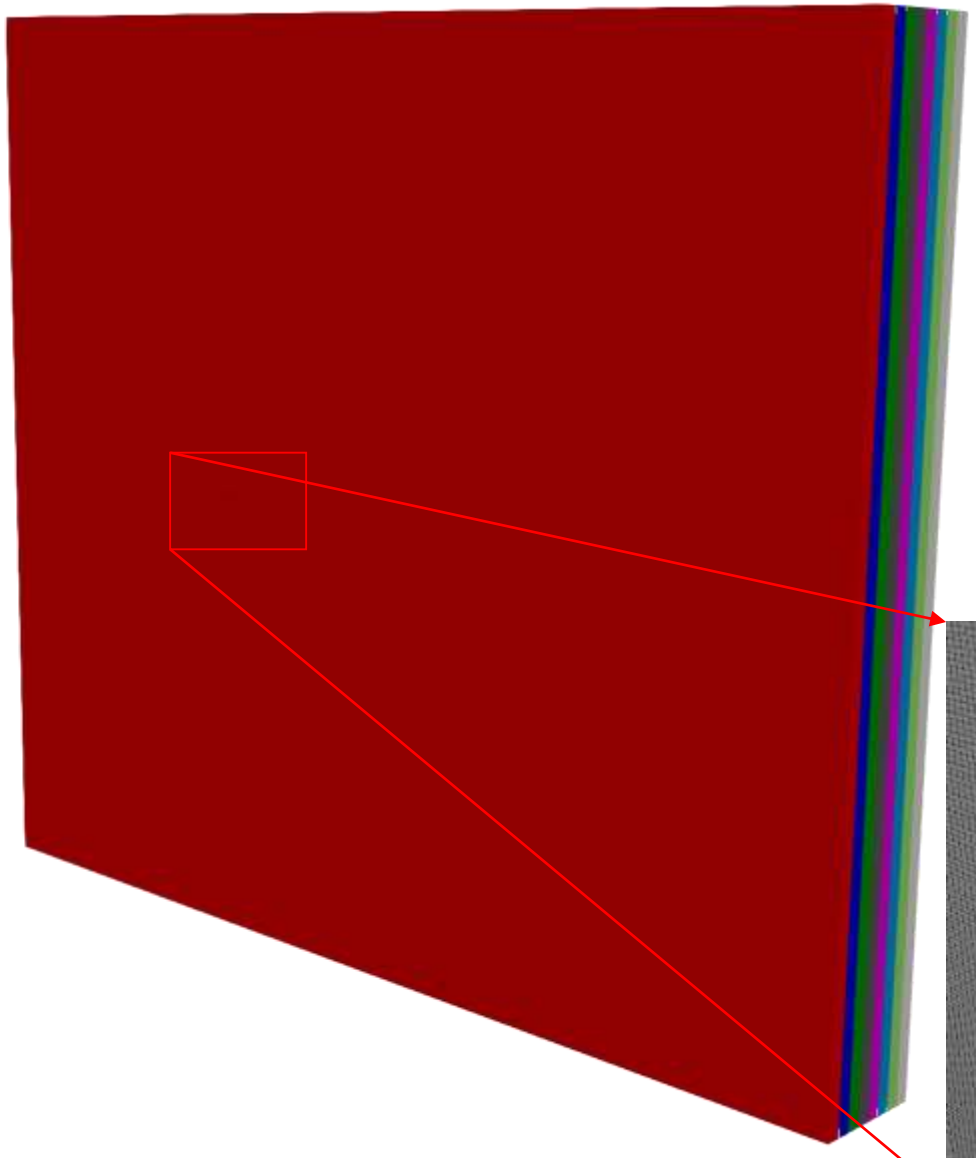
計算格子の作成

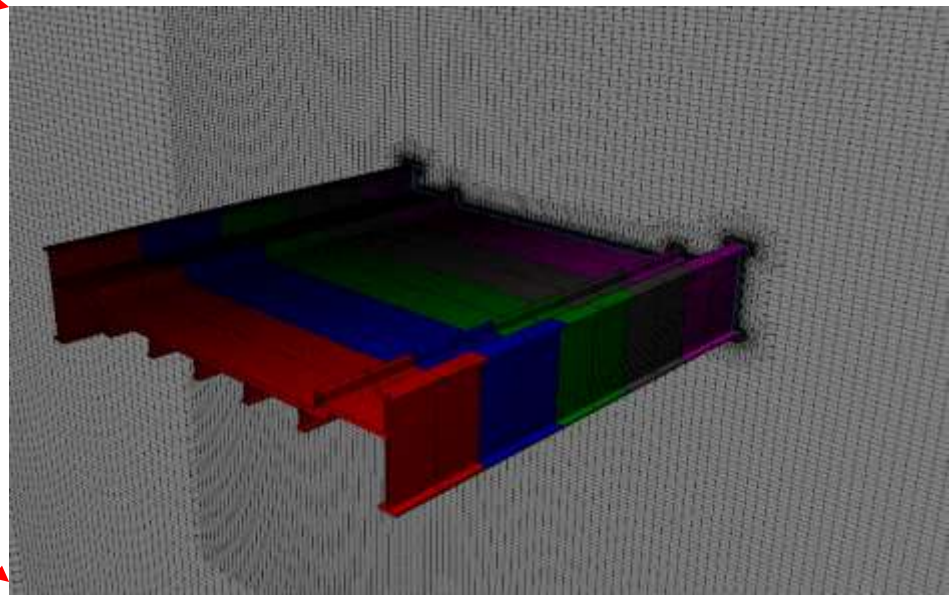




3.マルチカラー処理の手法

計算格子の作成





内容

- 超大規模のプリ処理　（フック）
  - メッシュの作り方
- コードの最新改良と性能評価　（内山）
  - OpenFOAMのthread 並列化

Xeon-phi上での非構造格子のthread 並列化

OpenFOAM本体の高速化とスレッド並列化

本報告ではthread並列化まで

- 入り組んだC++コードの展開とチューニング（2倍高速化）
- 連立方程式解法の変更，改良
  - 流速：BiCG → Additive Schwartzに変更（演算量1/2）
  - 圧力：AMG-CG → CG法のアルゴリズム改変（安定化）
  - 独自のblock multicolorでILU smootherをスレッド並列化＊
- 連立方程式以外の部分も細部まで並列化
  - loopタイプは数種類しかない　⇒　数種類の方法を考えれば良い
  - 行列イメージで，バンド幅の狭い問題と広い問題で手法を開発＊
  - 各境界領域内も再帰参照しないようにリオーダリング＊

⇒ストロングスケールで40倍以上の性能（64スレッド）

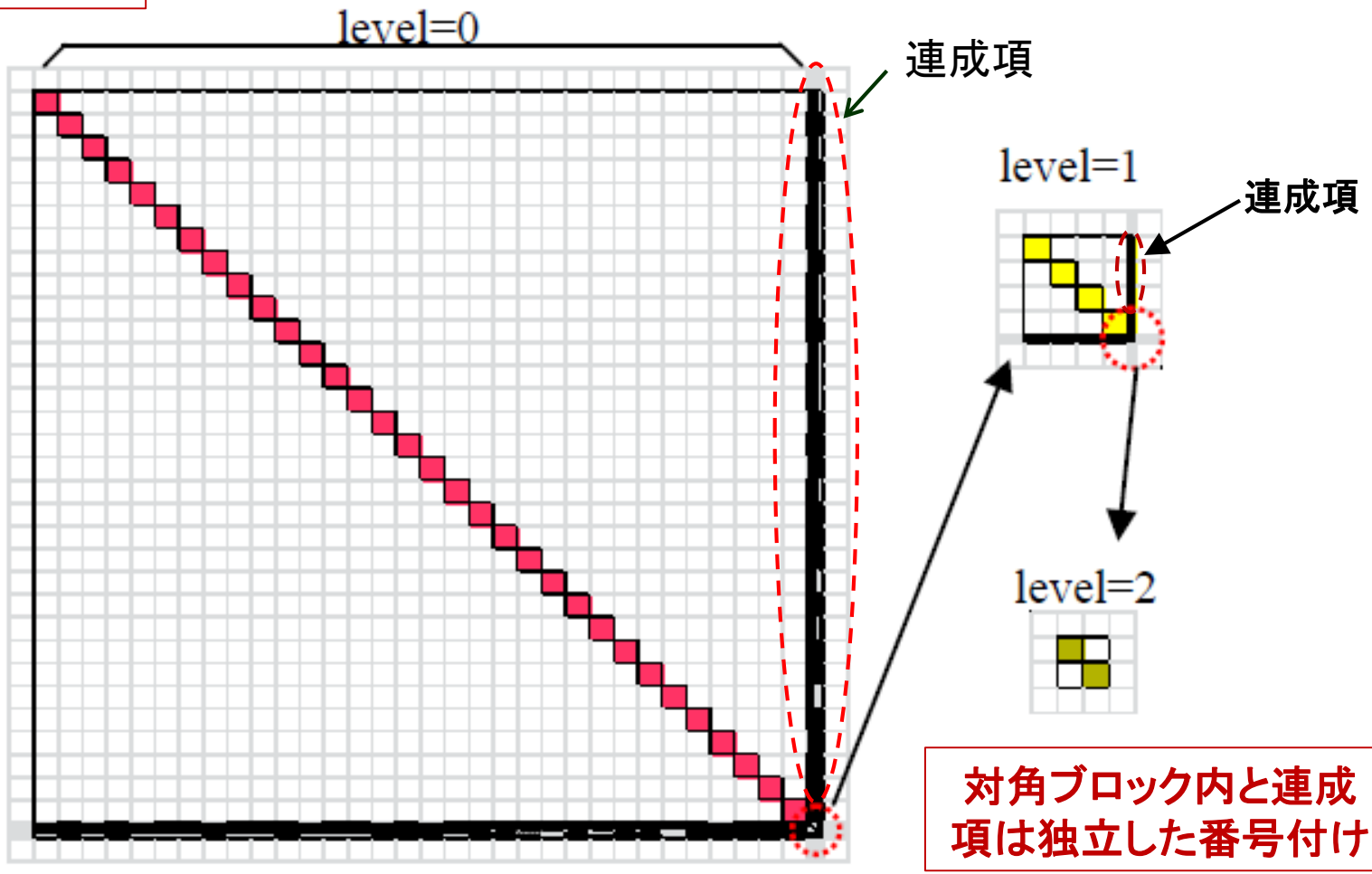
対象コード：OpenFOAM-1606+ (pisoFoam)

使用する計算機：Oakforest-PACS（東大・筑波）

格子のオーダリング（連立方程式解法）

METISでグラフ分割
連成項を後ろに回す

例：グループ数：（32, 4, 2）



係数行列の非零項の分布イメージ

対角ブロック内と連成
項は独立した番号付け

連立方程式解法前後で
並び替えを行う

AMG法のCoarse Grid
にも適用

格子の99%以上がlevel=0, 1に含まれる

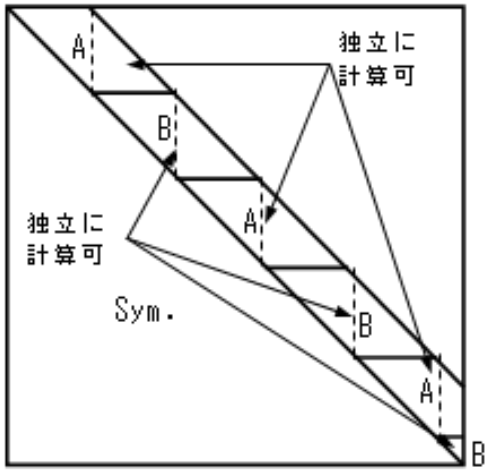
良く現れるループの並列化

良く現れるループの形：faceの数で回るループ

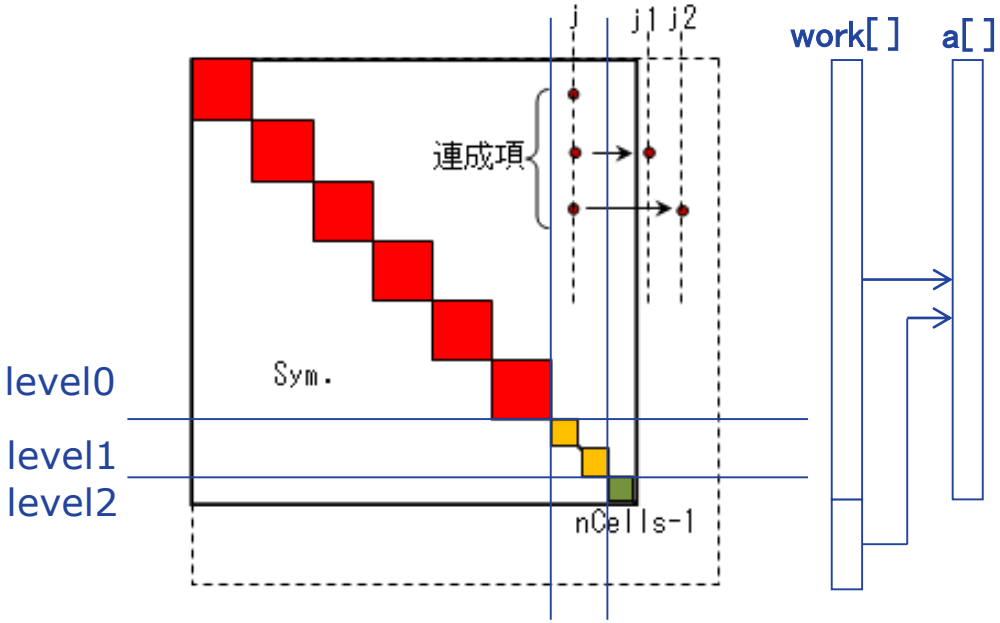
```
for(label fi=0; fi<nFaces; fi++) {
  label i=owner[fi], j=neighbor[fi];
  scalar tmp = (some computation);
  a[i] += tmp;
  a[j] += tmp;
}
```

行列イメージだと，
非対角項に関する計算を行って，
非対角項の位置 iと jに対応する2つの項に加算

バンド幅が狭い場合



バンド幅が広い場合



非零項の配置は対称で、Compressed Row Storage（上三角）

良く現れるループの並列化

境界領域

境界の数

```
for(label ipt=0; ipt<nPatches; ipt++) {
  const LabelUList& faceCells = boundary[ipt].faceCells();
```

faceの数

```
  for(label fi=0; fi<patchSizes[ipt]; fi++) {
    label i=faceCells[ipt][fi];
    scalar tmp = (some computation);
    a[i] += tmp;
  }
}
```

再帰参照される

並び替えて,
再帰参照されるものを後ろに回せば良い

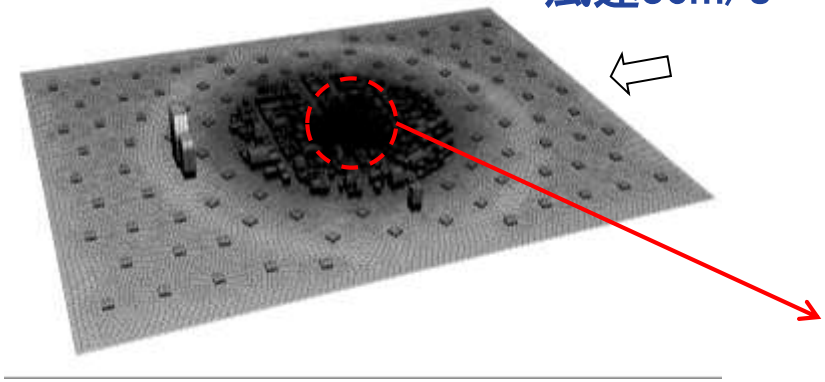
```
for(label ipt=0; ipt<nPatches; ipt++) {
  const LabelUList& faceCells = boundary[ipt].faceCells();
  const label               fi1  = cp1[ipt].fi1;
  const label * __restrict__ old1 = cp1[ipt].old1;
  #pragma omp for
  #pragma ivdep
  for (label ffi=0; ffi<fi1; ffi++) {
    label  fi  = old1[ffi];
    label  i   = faceCells[fi];
    scalar tmp = (some computation);
    a[i] += tmp;
  }
  #pragma omp single
  for (label ffi=fi1; ffi<patchSizes[ipt]; ffi++) {
    label  fi  = old1[ffi];
    label  i   = faceCells[fi];
    scalar tmp = (some computation);
    a[i] += tmp;
  }
}
```

再帰参照しない

前loopの残り

計算例：高層ビル：バンド幅狭い

風速36m/s



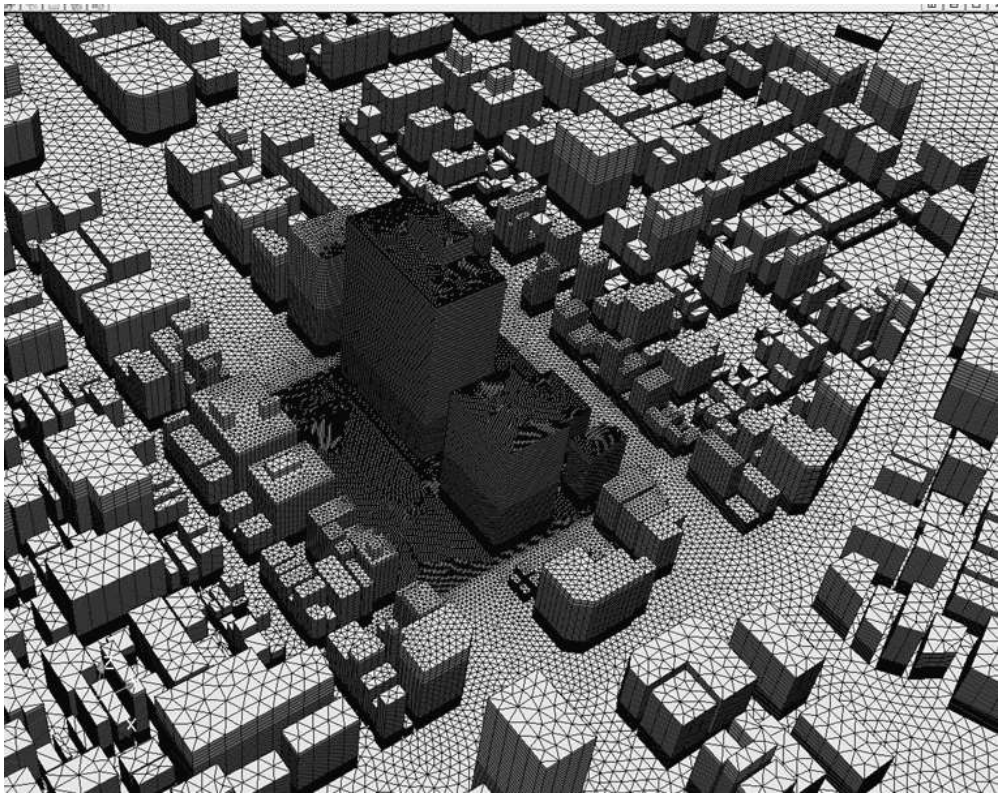


Table 1 計算領域と格子数

計算領域 (m)	1,500×2,000×500	
格子数	9,973,802	
格子種類と数	hexahedra	237,952
	prisms	9,735,850

t=5.0−6.0 s, Δt=0.001 s

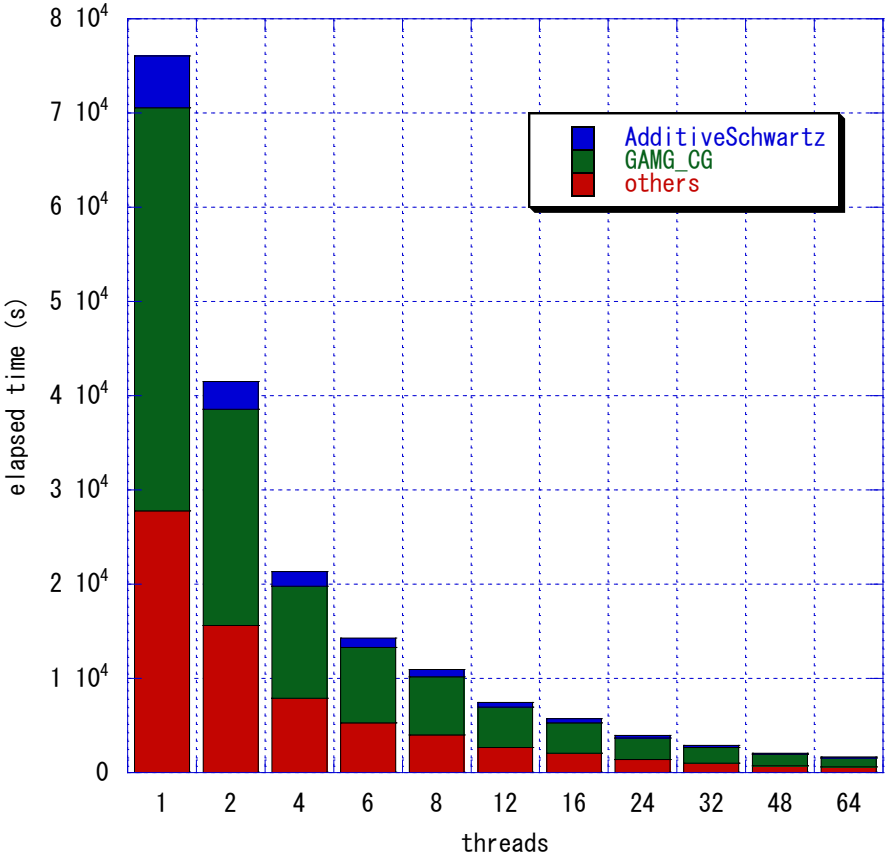
圧力の修正回数=3

計算例：高層ビル：バンド幅狭い

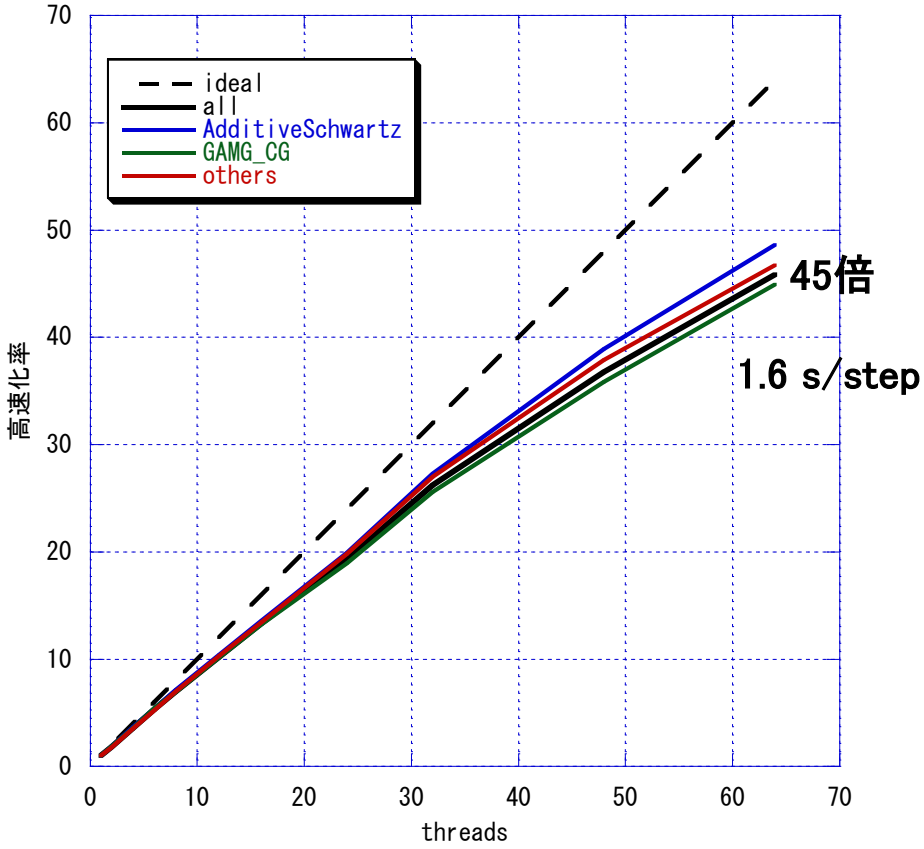
others（連立方程式解法以外）が 40%程度

連立方程式解法だけ並列化や高速化をしても駄目

Xeon-phi 7250 @ 1.4GHz
MCDRAM 16GBをcacheとして使用



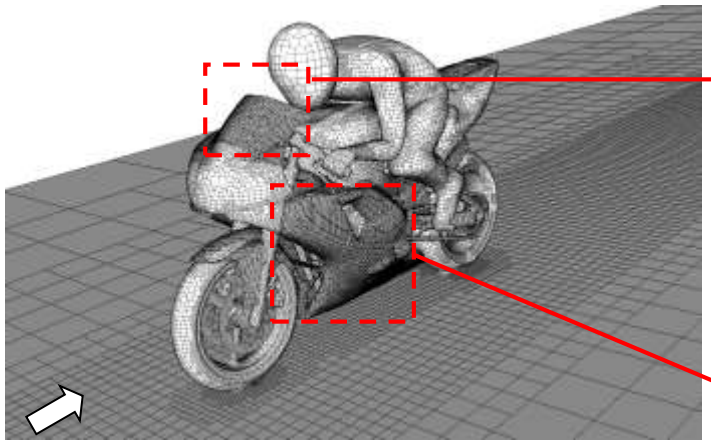
計算時間（5 s － 6 s）



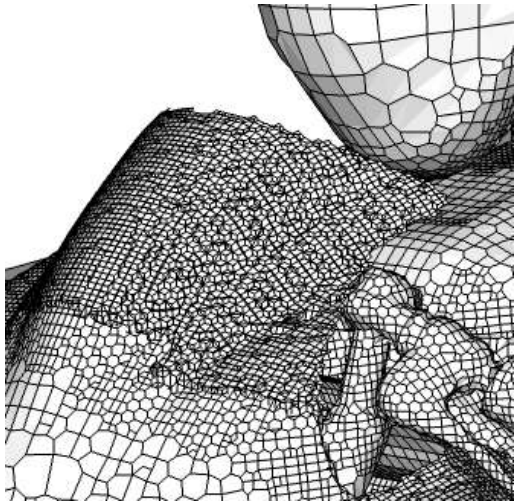
高速化率

バンド幅が狭い場合：436ブロック(最少ブロックサイズ=3000)

計算例：motorBike：バンド幅広い



風速20m/s



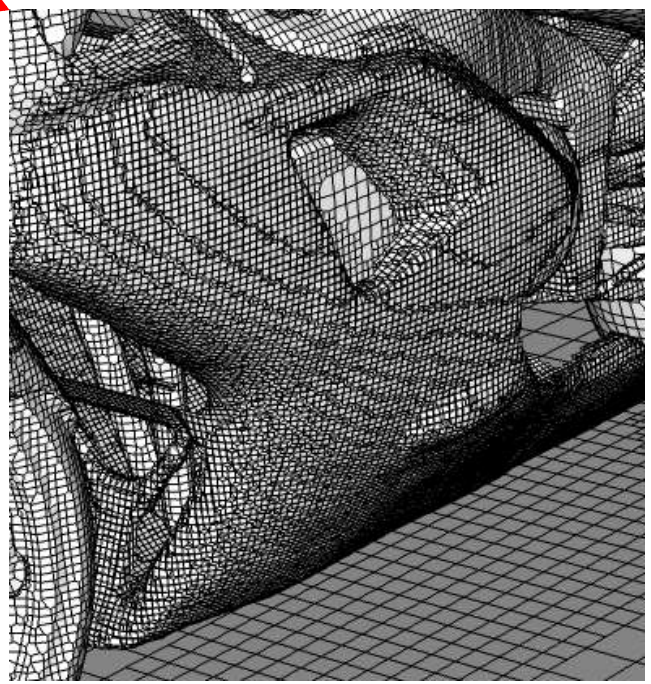


Table 2 計算領域と格子数

計算領域 (m)	20×8×8	
格子数	4,601,581	
格子種類と数	hexahedra	3,897,137
	prisms	107,368
	wedges	19,160
	pyramids	535
	tet wedges	21,499
	tetrahedral	444
	polyhedra	555,438

t=0.2−0.25 s, Δt=0.00005 s

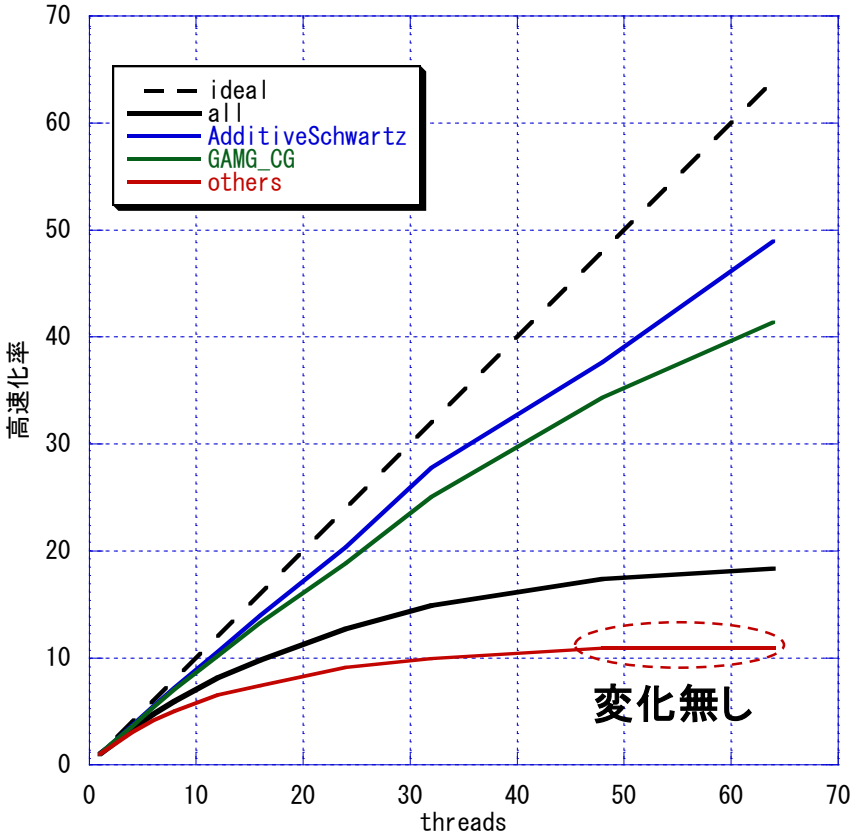
.

計算例：motorBike：バンド幅広い

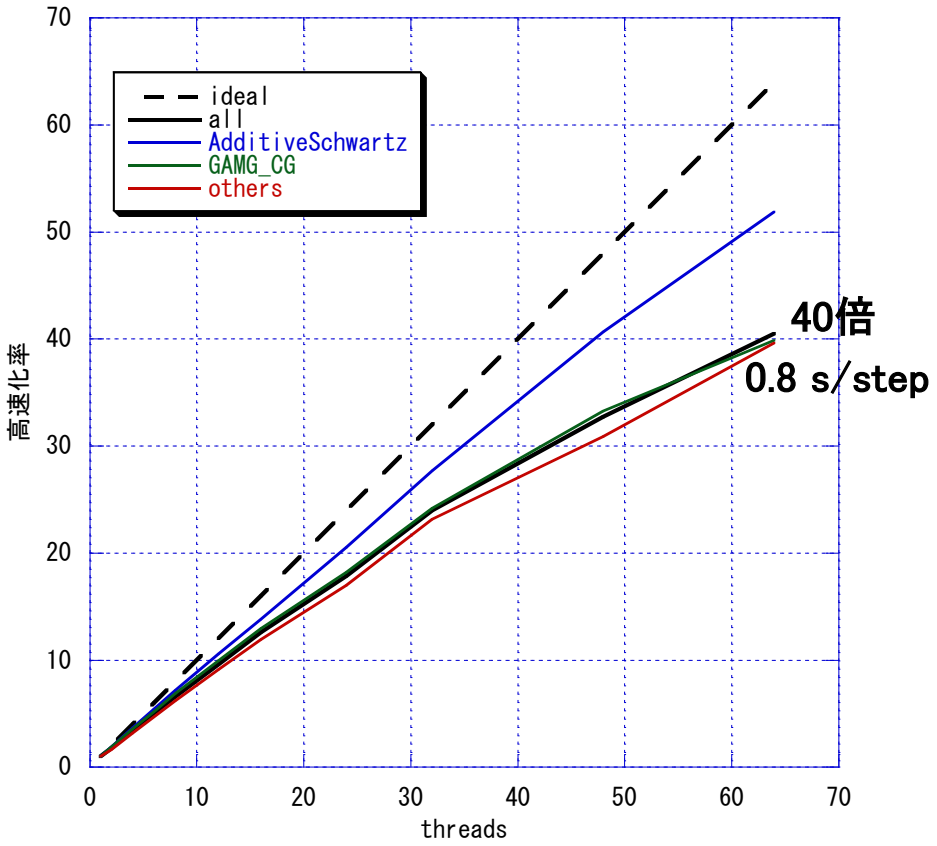
others（連立方程式解法以外）が 40%程度

連立方程式解法だけ並列化や高速化しても駄目

Xeon-phi 7250 @ 1.4GHz
MCDRAM 16GBをcacheとして使用



移植直後の高速化率



改良後の高速化率

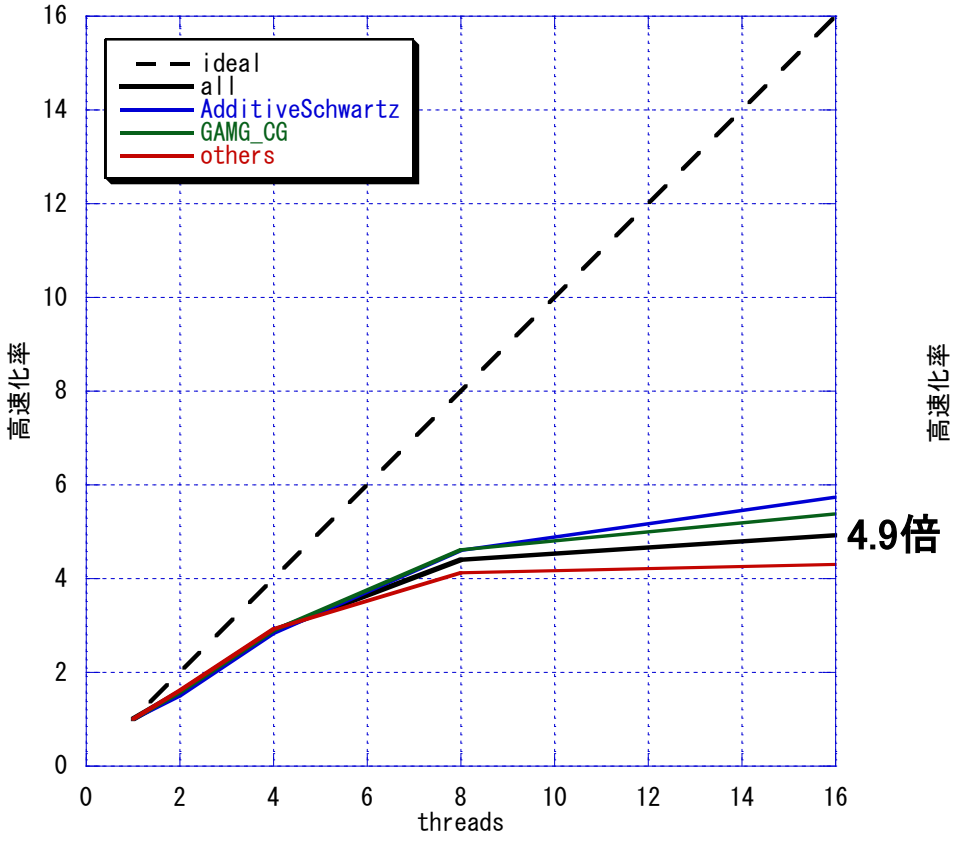
バンド幅が狭い場合の方法：117ブロックしかない

バンド幅が広い場合の方法：各256グループ(level=0,1)

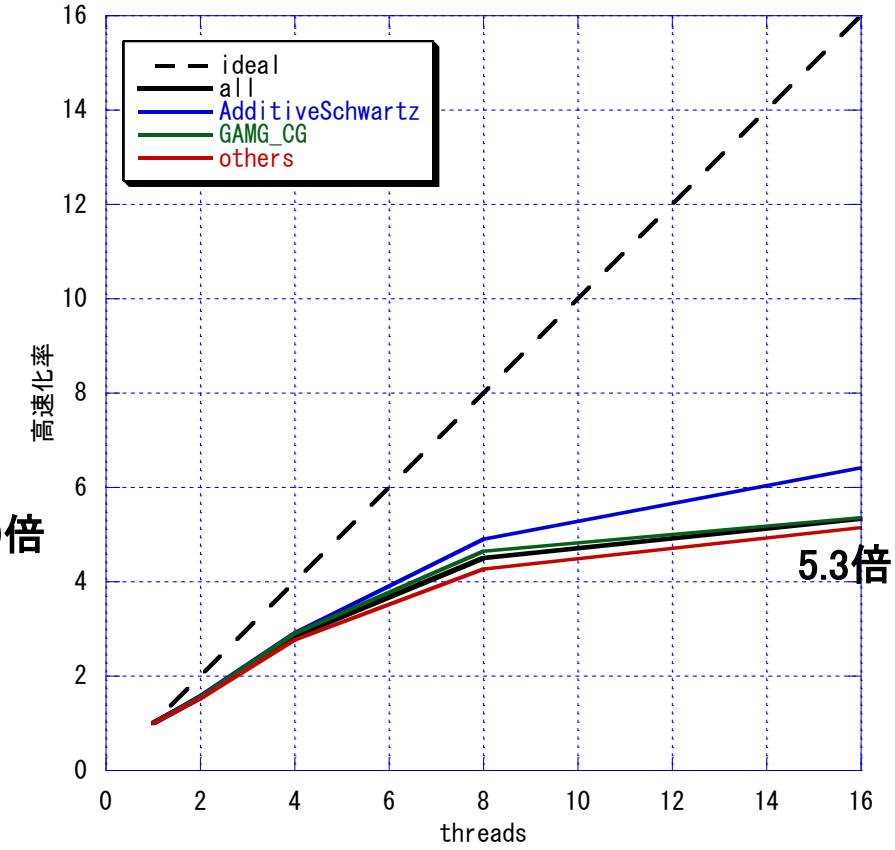
Xeon(Broadwell)で計算してみたが...

スレッドを増やしても性能が上がりにくい
- メモリー帯域が足りない？
- cache memoryが競合？

Xeon E5-2687W v4 @ 3.00GHz
2 CPU, 12 core/CPU, 30MB Cache



高速化率：高層建物



高速化率：motorBike

結語

■Oakforest-PACS上でpisoFoamのthread並列化を行った

■連立方程式解法以外の部分も細部に渡ってthread並列化
■細部まで並列化し，ストロングスケールで40倍以上の性能

課題

■SIMD化