BCM データ管理フレームワーク BCM Tools (仮称) ガイド

2012 年 9 月 16 日

目次

1 はじめに

本フレームワークは，BCM(Building-Cube Method) データ構造の管理を目的としている．扱う対象は，各キューブに付随するデータとキューブ間の接合情報である．それらのデータを管理・操作するための機能を提供する．

本フレームワークは，主に流体シミュレーションプログラムの開発において利用することを想定している．今後は，さらに規模の大きなフレームワーク (例えば「流体ソルバ開発用フレームワーク」) に，その一部として組み込まれる可能性がある．

1.1 用語

本フレームワークおよび本ドキュメントで用いる用語について整理する．

ブロック BCM におけるキューブのこと
フェイス ブロックの 6 つの境界面
サブフェイス フェイスを 4 等分した正方形領域
仮想セル ブロック内をセル分割した 3 次元データにおける隣接ブロックにまたがった袖領域
ノード 特にことわらない場合は，並列計算ノードではなく，ツリーデータ構造におけるノードを指すものとする

1.2 フレームワークの機能

■できること

- ブロック単位でのデータ管理
- 仮想セル同期

■できないこと

- 周期境界以外の境界条件の取り扱い
- データのファイル IO
- BCM 構造 (ブロックの空間配置) 生成 → すでに生成済みであること
- 領域分割処理 → すでに分割済みであること

これらは，本フレームワークを組み込んだブログラム側で処理する必要がある．ただし，フレームワークの使用例として，可視化用データ出力クラス (6 章)，BCM ブロック配置決定とその領域分割 (7 章)，および，ブロックデータ作成用ユーティリティクラス群のサンプル実装を，パッケージ内に含めてある．

2 インストール方法

2.1 ディレクトリ構成

```
Makefile.inc       ... make パラメータファイル
lib/               ... ライブラリファイル
include/           ... ヘッダファイル
src/               ... ソースコード
  Makefile
  DataClass.list     ... DataClass リストファイル
  Block/
  CommBuffer/
  DataClass/
example/           ... サンプルプログラム
  util/              ... サンプルプログラムで共用するユーティリティクラス
    include/           ... ヘッダファイル
    src/               ... ソースファイル
  ParametricDivider/
  PolygonDivider/
  SOR1/
  SOR2/
  ParallelMeshGeneration/
  Polylib_2_0_3_Rel/                 ... Polylib ライブラリ
  Polylib_2_0_3_Rel_search_nearest/  ... パッチを当てた Polylib ライブラリ
  STL_DATA/        ... STL データファイル
doc/
  BCM_guide.pdf    ... このドキュメント
  guide/           ... TeX ソースファイル
  doxygen/html/    ... Doxygen 出力
```

2.2 コンパイル・リンク方法

以下の手順により，lib ディレクトリにライブラリファイル libbcm.a を作成する．

1. Makefile.inc の書き換え
2. src/DataClass.list の書き換え
3. src ディレクトリで make を実行

■Makefile.inc の書き換え　次の 2 つのマクロに値を設定する．

`CXX`　C++ コンパイラ名

`OPTIONS`　C++ コンパイラに渡すオプション (`-I`, `-L`, `-l` 以外のもの)

また，可視化データ出力用の SiloWrite クラスを利用するには，以下のマクロも設定する必要がある．

`SILO_INC`　Silo ライブラリのインクルードファイルディレクトリ指定 (`-I` オプション)

`SILO_LIB`　Silo および HDF5 ライブラリのリンクオプション (`-L` と`-l` オプション)

■src/DataClass.list の書き換え　このファイルには，使用を予定するデータクラスおよび仮想セルアップデータを次の形式で列挙する．

```
DATACLASS_SCALAR_XXX     # スカラー型データクラス
DATACLASS_VECTOR_XXX     # ベクトル型データクラス
UPDATER_SCALAR_XXX       # スカラー型仮想セルアップデータ
UPDATER_VECTOR_XXX       # ベクトル型仮想セルアップデータ
```

ここで，XXX には INT, DOUBLE など，型を表すキーワードが入る．詳しくは，同ファイル中のコメントを参照のこと．また，指定可能な型については，次節の制限事項を参照．

■外部ライブラリ　サンプルプログラムをコンパイル・リンクするには，以下のライブラリが必要である．

HDF5 **および** Silo　ParametricDivider, PolygonDivider, SOR2, ParallelMeshGeneration に必要
Polylib_2_0_x　ParallelMeshGeneration に必要
search_nearest **パッチを当てた** Polylib_2_0_x　PolygonDivider に必要

HDF5 および Silo ライブラリのインストール方法については，付録 A を参照のこと．

2.3　制限事項

現バージョンの実装には，以下の制限がある．

1. スカラー型データクラス (Scalar3D<T>) に対応した仮想セルアップデータ (Scalar3DUpdater<T>) で利用可能なものは，実数型のみ (T=float または double)．
2. ベクトル型データクラス (Vector3D<T>) に対応した仮想セルアップデータ (Vector3DUpdater<T>) は未実装．
3. 京でのチューニングは，これから．

3　フレームワーク構成

実行環境に合わせたチューニング，アルゴリズム改良，機能追加などを容易にするために，本フレームワークは，単純な機能を持つ複数のコンポーネントから構成し，各コンポーネント間の依存関係を最小限にとどめるように設計した．

本フレームワークは C++ を用いて実装され，以下のクラス群により構成される (図 1)．

ブロック (Block)　個々のキューブに属するデータを管理
データクラス (DataClass)　ブロック内の 3 次元配列データのラッパクラス
ブロックマネージャ (BlockManager)　並列計算ノード内の全ブロックを管理
仮想セルアップデータ (VCUpdater)　データクラスに付随して，仮想セル同期計算を担当
通信バッファクラス (CommBuffer)　MPI 通信用ユーティリティクラス

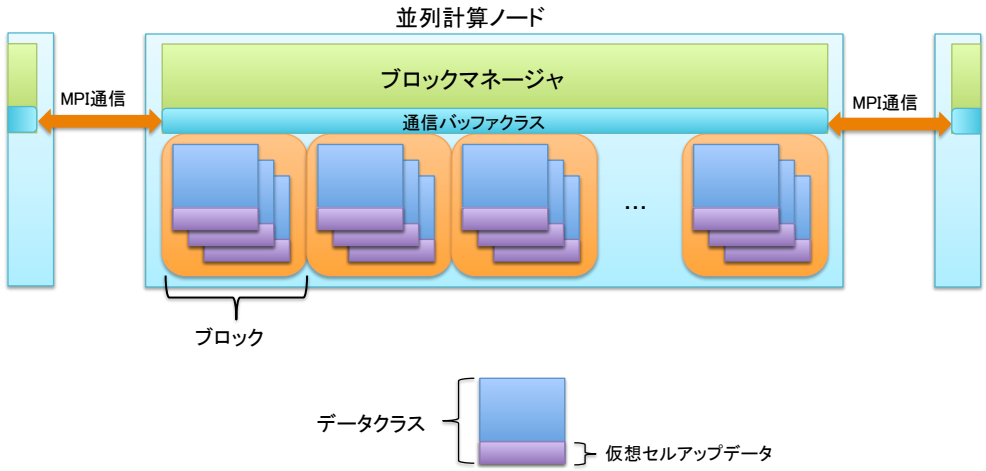

図 1　フレームワーク構成

3.1　**ブロック** (Block)

個々のブロックに属するデータを管理するクラス．ブロック固有のデータ，物理量などを格納した 3 次元配列データ (データクラス) のテーブル，隣接ブロックとの接合情報，これらを管理するためのメソッドよりなる．

フレームワークとしては，最低限の基本機能を実装した基底クラス (BlockBase クラス) のみを提供する．この基底クラスを継承によりカスタマイズすることにより，開発するソルバに合わせて，任意の変数やメソッドを追加できる．

3.2 データクラス (DataClass)

ブロック内の各セルにおける圧力，速度などの物理量データや，内部境界条件に対応したフラグ値などを格納するための，3 次元配列のラッパクラス．これらのデータは，実際には，データクラス内部の 1 次元配列に格納される．この内部データは，Fortran や C で書かれたサブルーチンに渡せる形式でエキスポート可能である．

フレームワークとしては，基底 DataClass クラス，仮想セル同期可能なデータクラスのインタフェースを規定した UpdatableDataClass クラス，サンプル実装としての Scalar3D テンプレートクラスと Vector3D テンプレートクラスを提供する (図 2)．

仮想セル同期計算は，データクラス内部に保持した仮想セルアップデータクラス (updater メンバ) が担当する．仮想セルアップデータクラスを入れ替えることにより，仮想セルの同期方法やレベル間補間計算方法をカスタマイズできる．

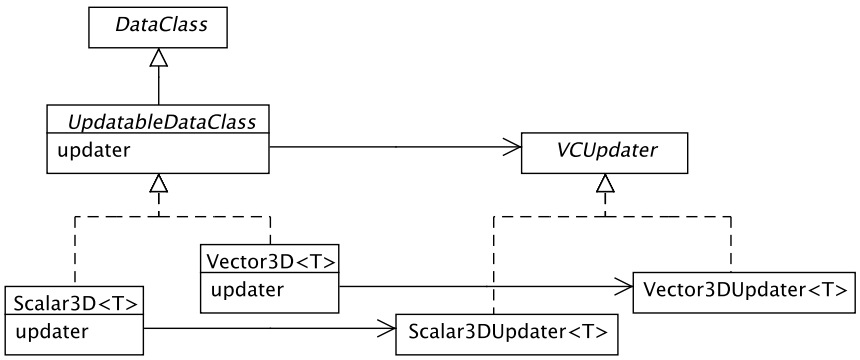

図 2　データクラスと仮想セルアップデータ

3.3 ブロックマネージャ (BlockManager)

並列計算ノード内の全ブロックを管理するクラス．ブロックマネージャに登録された全ブロックに対して以下の処理を行える．

- データクラスの生成と登録
- 指定したデータクラスの仮想セル同期計算

将来的には，AMR(Adaptive Mesh Refinement) 実装時におけるキューブの再分割と統合，動的ロードバランスを考慮した場合のキューブデータの並列計算ノード間での移動，などの機能もブロックマネージャに担当させる．

3.4 仮想セルアップデータ (VCUpdater)

仮想セル同期計算と，それにともなう異なるレベル間での仮想セル補間計算を担当する．ただし，MPI 通信部分は通信バッファクラスに丸投げしている．このクラスをカスタマイズすることにより，セル表面配置 (staggered) ベクトルデータの仮想セル同期や，斜め方向ステンシルを含む仮想セル同期にも対応可能．

フレームワークとしては，インタフェースを規定した基底 VCUpdater クラス，スカラおよびベクトル配列用のサンプル実装 Scalar3DUpdater テンプレートクラス，Vector3DUpdater テンプレートクラス[*1]を提供する (図 2)．

3.5 通信バッファクラス (CommBuffer)

MPI 通信用のユーティリティクラス．図 1 のようにこのクラスをブロックマネージャに持たせることにより，指定したデータクラスの仮想セル同期通信を全ブロックまとめて行うことができる．また，このクラスをブロックごとに持たせることにより，ブロック単位で仮想セル同期を行うことも可能となる．

図 3 に，仮想セル同期における，データクラス，仮想セルアップデータ，通信バッファクラスの役割分担をまとめる．

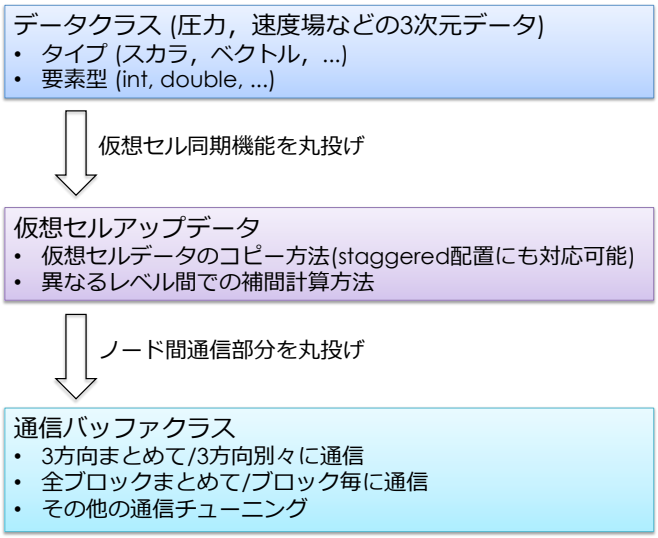

図 3　仮想セル同期における役割分担

[*1] Vector3DUpdater クラスは，まだ未実装．

4　フレームワークを使ってソルバを書く

ここでは，本フレームワークを使ってソルバを書くために必要な説明を行う．

- フレームワークの使い方の概略
- ブロッククラスの使い方およびカスタマイズ方法
- ブロックマネージャクラスの使い方
- Scalar3D，Vector3D データクラスの使い方

4.1　フレームワークの使い方の概略

■フレームワーク初期化以前に行なっておくこと

1. BCM キューブ配置 (ツリー構造) の構築
2. キューブ単位での領域分割 (並列計算ノードへの各キューブの分配)

各並列計算ノードに属するキューブには，0 から始まるローカル ID 番号を振っておく．また，それらをランク 0 の計算ノードから順番に並べて，0 からの通し番号を振り，これをグローバル ID と呼ぶことにする[*2]．

■フレームワークの初期化　以下の手順にしたがいフレームワークの初期化する．

1. ブロックマネージャの取得
2. 各ブロックの生成とブロックマネージャへの登録
3. ブロックマネージャをとおして:
   (1) 各ブロックにおけるデータクラスの生成と登録
   (2) 必要な内部テーブルの準備

個々のブロックの生成時には，BlockBase クラスのコンストラクタに，6 つのブロック境界面 (フェイス) ごとに隣接情報をパックした NeighborInfo オブジェクトをわたす．NeighborInfo に含まれる情報は以下のとおりで，ブロックの生成時にこれらが確定している必要がある[*3]．

- その面方向に隣接ブロックが存在するかどうか
- 隣接ブロックが存在する場合:
  - 隣接ブロックとのレベル差
  - レベル差が 0 の場合:
    - 隣接ブロックの所属計算ノードのランク番号
    - 隣接ブロックのグローバル ID 番号
  - レベル差が −1 の場合:
    - 隣接ブロックの所属計算ノードのランク番号

[*2] 逆に，先に Z-ordering 等で全キューブにグローバル ID を振り，これをノード数で等分割することにより，領域分割を行なってもよい．

[*3] ツリー構造については，隣接ブロック間のレベル差が 1 以内であることのみを要求しているので，マルチルートなツリー構造にもそのまま対応可能である．また，現実装では，各ブロックの pedigree 値は必要としていない．

    * 隣接ブロックのグローバル ID 番号
    * 隣接ブロックのどのサブフェイスに接しているか
  – レベル差が +1 の場合，4 つのサブフェイスごとに:
    * 隣接ブロックの所属計算ノードのランク番号
    * 隣接ブロックのグローバル ID 番号

■全ブロックに対する処理の概要　以下の例では，ブロックマネージャ blockManager をとおして，全ブロックに対して，仮想セルアップデータ Scalar3DUpdater<double>を持つデータクラス Scalar3D<double>を生成し登録している．

```
int dacaClassID; // データクラスを識別するための ID

// ブロックマネージャを取得
BlockManager& blockManager = BlockManager::getInstance();

// データクラスの生成・登録
const int vc = 2; // 仮想セル幅
dataClassID
  = blockManager.setDataClass<Scalar3D<double>, Scalar3DUpdater<double> >(vc);

// 仮想セル同期の準備
const int tag = 1234; // MPI 通信タグ値
blockManager.prepareForVCUpdate(dataClassID, tag);

// 仮想セル同期計算
blockManager.updateVC(dataClassID);
```

■ブロック単位での処理の概要　以下の例では，個々のブロック内で，指定したデータクラスを取得して，それに対して処理を行っている．

```
// ブロックマネージャを取得
BlockManager& blockManager = BlockManager::getInstance();

// ローカルブロック ID に対するループ
for (int id = 0; id < blockManager.getNumBlock(); id++) {

  // ブロックの取得
  Bloc* block = dynamic_cast<Block*>(blockManager.getBlock(id));

  // データクラスの取得
  Scalar3D<double>* dataClass
      = dynamic_cast<Scalar3D<double>*>(block->getDataClass(dataclassID));

  // データクラス内のデータ配列を取得
  double* data = dataClass->getData();

  // Fortran サブルーチンなどに処理をわたす
  sub_(nx, ny, nz, vc, data, ...);
}
```

4.2 ブロッククラスの使い方およびカスタマイズ方法

ブロッククラスは，個々のブロックに属するデータを管理する．フレームワークとしては，最低限の基本機能のみを実装した基底クラス (BlockBase クラス) のみを提供する．

4.2.1 BlockBase クラス

■ヘッダファイル BlockBase.h

■private 変数

```
Vec3i size;          // ブロック内の分割数（セル数）
Vec3r origin;        // ブロック原点の空間座標
Vec3r blockSize;     // ブロックサイズ
Vec3r cellSize;      // セルサイズ

int level;           // ツリーレベル

NeighborInfo* neighborInfo;    // 隣接情報（各面に対応した 6 要素の NeighborInfo 配列）

typedef std::vector<DataClass*> DataClassTable;
DataClassTable dataClassTable; // データクラスのポインタテーブル
```

■コンストラクタ

```
BlockBase(const Vec3i& size, const Vec3r& origin, const Vec3r& blockSize,
          int level, NeighborInfo* neighborInfo);
```

ブロック分割数 (セル数)size は，各次元とも偶数である必要がある．

■データクラス登録

```
int setDataClass(DataClass* dataClass);
```

データクラス ID(登録されたデータクラスの通し番号) が返る．

■データクラス取得

```
DataClass* getDataClass(int dataClassId);
```

実際には，取得したデータクラスは，以下の例のように登録したデータクラスにキャストして使用する必要がある．

```
Scalar3D<double>* dataClass
          = dynamic_cast<Scalar3D<double>*>(block->getDataClass(dataClassId));
```

■位置情報，分割情報，レベル値の取得

```
const Vec3i& getSize() const;
```

```
const Vec3r& getOrigin() const;
const Vec3r& getBlockSize() const;
const Vec3r& getCellSize() const;
int getLevel() const;
```

■隣接情報の取得

```
const NeighborInfo* getNeighborInfo() const;
```

各面に対応した要素数 6 の NeighborInfo クラス配列が返る．

4.2.2 NeighborInfo クラス

面ごとの隣接ブロック情報を保持するクラス．ブロッククラスの生成時に，各面に対応した 6 要素からなる NeighborInfo 配列をコンストラクタにわたす必要がある[*4]．

■ヘッダファイル NeighborInfo.h

■private 変数

```
int neighborID[4];      // 隣接ブロックのグローバル ID
int neighborRank[4];    // 隣接ブロックが所属する計算ノードのランク番号

bool outerBoundary;     // 外部境界フラグ

int levelDiffarence;    // 隣接ブロックとのレベル差（-1，0，+1 のどれか）

int neighborSubface;    // 隣接ブロック接触面のサブフェイス番号
```

ブロック境界面が周期境界以外の外部境界面である場合には，隣接ブロックは存在しない．その場合には，グローバル ID には −1, ラング番号には MPI::PROC_NULL を入れる．フラグ outerBoundary には，隣接面が外部境界 (周期境界条件も含む) の場合に true をセットする．

配列 neighborID[4] と neighborRank[4] は，levelDiffarence=1 以外の場合には，最初の要素のみを用いる．

neighborSubface には，levelDiffarence=-1 以外の場合には，常に-1 を入れる．

4.2.3 ブロッククラスのカスタマイズ

BlockBase クラスを継承によりカスタマイズすることにより，任意のメンバ変数やメソッドを追加することができる．以下の例では，新に boundaryInfo クラスの配列をメンバ変数に追加した Block クラスを定義している．

```
#include "BlockBase.h"
#include "BoundaryInfo.h"

class Block : public BlockBase {
```

[*4] 将来的には，NeighborInfo 配列への値の設定は，本フレームワークの上位に位置するフレームワーク/ライブラリにより，入力データより自動的に行われるべきである．

```
  // 追加したメンバ変数
  BoundaryInfo* boundaryInfo;

public:

  // コンストラクタ
  // 初期化リスト内で，基底クラス BlockBase のコンストラクタを呼んでいる
  Block(const Vec3i& size, const Vec3r& origin, const Vec3r& blockSize,
        int level, NeighborInfo* neighborInfo, BoundaryInfo* boundaryInfo)
   : BlockBase(size, origin, blockSize, level, neighborInfo),
     boundaryInfo(boundaryInfo) {}

  // デストラクタ
  virtual ~Block() {
    delete[] boundaryInfo;
  }

  // 追加したメソッド
  const BoundaryInfo* getBoundaryInfo() const { return boundaryInfo; }

};
```

4.3 ブロックマネージャクラスの使い方

ブロックマネージャクラスに対する操作は，

- ブロックマネージャの初期化
- ブロックの登録
- ブロックの取得
- 全ブロックに対してのデータクラス操作

からなる．

4.3.1 BlockManager クラス

■ヘッダファイル BlockManager.h

■インスタンスの呼び出し

```
BlockManager& getInstance();
```

BlockManager クラスはシングルトンとして実装されているので，コンストラクタによりインスタンスを生成する必要はない．BlockManager クラスでは，このメソッドのみスタテックメソッドである．実際のコードでは以下のように使用する．

```
BlockManager& blockManager = BlockManager::getInstance();
```

■コミュニケータの設定

```
setCommunicator(const MPI::Intracomm& comm);
```

MPI::COMM_WORLD 以外のコミュニケータを使用するには，ブロックの登録前に，このメソッドによりコミュニケータを変更する．

■ブロックの登録

```
void registerBlock(BlockBase* block);
```

並列計算ノード内の全ブロックに対して，この操作を行う必要がある．登録した順に，0 から始まる (ローカル) ブロック ID がふられる．登録されたブロックのオブジェクトは，BlockManager クラスのデストラクタにより，メモリ解放される．

■ブロック登録の終了処理

```
void endRegisterBlock();
```

全ブロックの登録後にこのメソッドを呼び出す必要がある．このメソッドでは，登録されたブロック間の整合性 (各ブロックのセル分割数が等しいこと，など) のチェックも行う．

■コミュニケータの取得

```
const MPI::Intracomm& getCommunicator() const;
```

■先頭ブロックのグローバル ID を取得

```
int getStartID() const;
```

■並列計算ノード内のブロック総数の取得

```
int getNumBlock() const;
```

■ブロックの取得

```
BlockBase* getBlock(int localID);
```

ローカル ID(0〜numBlock-1) を指定してブロックを取得．基底クラス BlockBase の提供する機能ではなく，ユーザがカスタマイズしたブロッククラスの機能を使う場合には，以下の例のように，返り値をキャストする必要がある．

```
Block* block = dynamic_cast<Block*>(blockManager->getBlock(id));
```

■ブロック内のセル分割数の取得

```
const Vec3i& getSize() const;
```

■全ブロックに対して，共通したデータクラスを生成して登録

```
int setDataClass<D>(int vc);     // 仮想セル同期なし
int setDataClass<D, U>(int vc);  // 仮想セル同期あり
```

両メソッドともテンプレートメソッドで，D にはデータクラスの型，U には仮想セルアップデータの型を指定する．vc は仮想セル幅．登録後は，返り値のデータクラス ID をとおして，このデータクラスを操作できる．

■仮想セル同期の準備

```
void prepareForVCUpdate(int dataClassID, int tag, bool separate = false);
```

仮想セル同期可能なデータクラスに対して，内部テーブルの作成などの準備を行う．dataClassID には，setDataClass メソッドの返り値を指定する．tag は，MPI 通信で用いるタグ値．xyz 三方向別々に仮想セル同期を行わせるには，フラグ separate に true を指定する．

■仮想セル同期 (三方向同時)

```
// ブロッキング通信版
void updateVC(int dataClassID);

// ノンブロッキング通信による同期開始
void beginUpdateVC(int dataClassID);

// ノンブロッキング通信による同期終了待ち
void endUpdateVC(int dataClassID);
```

■仮想セル同期 (三方向別々)

```
// ブロッキング通信版
void updateVC_X(int dataClassID);
void updateVC_Y(int dataClassID);
void updateVC_Z(int dataClassID);

// ノンブロッキング通信による同期開始
void beginUpdateVC_X(int dataClassID);
void beginUpdateVC_Y(int dataClassID);
void beginUpdateVC_Z(int dataClassID);

// ノンブロッキング通信による同期終了待ち
void endUpdateVC_X(int dataClassID);
void endUpdateVC_Y(int dataClassID);
void endUpdateVC_Z(int dataClassID);
```

4.4 Scalar3D，Vector3D データクラスの使い方

4.4.1 共通するメソッド

■セル数，仮想セル幅の取得

```
const Vec3i& getSize() const;  // セル数 (nx,ny,nz)
int getSizeX() const;          // x 方向セル数 (nx)
int getSizeY() const;          // y 方向セル数 (ny)
int getSizeZ() const;          // z 方向セル数 (nz)
int getVCSize() const;         // 仮想セル幅 (vc)
```

4.4.2 Scalar3D クラス

■ヘッダファイル Scalar3D.h

■コンストラクタ

```
Scalar3D<T>(const Vec3i& size, int vc);
```

T は double, int などの基本型．size はセル分割数を納めた 3 次元整数ベクトル，vc は仮想セル幅．

■データ領域の取得

```
T* getData() const;
```

要素数 (nx+2*vc)×(ny+2*vc)×(nz+2*vc) の T 型配列へのポインタが返る．

■インデックスファンクタ (関数オブジェクト) の取得

```
Index3DS getIndex() const;
```

4.4.3 Vector3D クラス

■ヘッダファイル Vector3D.h

■コンストラクタ

```
Vector3D<T>(const Vec3i& size, int vc);
```

T は double, int などの基本型．size はセル分割数を納めた 3 次元整数ベクトル，vc は仮想セル幅．

■データ領域の取得

```
T* getData() const;
```

要素数 3×(nx+2*vc)×(ny+2*vc)×(nz+2*vc) の T 型配列へのポインタが返る．

■インデックスファンクタ (関数オブジェクト) の取得

```
Index3DV getIndex() const;
```

4.4.4 セルデータへのアクセス

Scalar3D クラスでは，3 次元データを Fortran 的なメモリ配置で 1 次元配列に格納している．3 次元添字 (i,j,k) により指定されたデータにアクセスするために，次の 3 つの方法を用意してある．内部セル領域の添字は，0 から始めるものとしている．

1. データクラス変数に直接 3 次元添字 (i,j,k) を付けてアクセス
2. エクスポートした 1 次元配列に対して，自前で 1 次元添字を計算してアクセス
3. エクスポートした 1 次元配列に対して，インデックスファンクタ (関数オブジェクト)Index3DS によりアクセス

Vector3D クラスでは，3 次元データの配置は Scalar3D クラスと同様であるが，個々のベクトルの 3 成分はメモリ上に連続するように配置される．また，Vector3D 用のインデックスファンクタ Index3DV では，3 次元添字 (i,j,k) による x 成分のインデックスの取得に加えて，4 次元添字 (i,j,k,l) による l 成分 (l=0,1,2) のインデックス取得が可能である[*5]．

```
Scalar3D<double>(size, vc) p;
Vector3D<double>(size, vc) v;

// 直接 3 次元添字でアクセス
p(i,j,k) = 0.0;
v(i,j,k,0) = v(i,j,k,1) = v(i,j,k,2) = 0.0;

// 1 次元配列データをエクスポート
double* pData = p.getData();
double* vData = v.getData();

// 自前で 1 次元添字を計算してアクセス
int nx0 = size[0] + 2 * vc;
int ny0 = size[1] + 2 * vc;
int nz0 = size[2] + 2 * vc;
#define INDEX(i,j,k) ((i)+vc + nx0*((j)+vc) + nx0*ny0*((k)+vc))
pData[INDEX(i,j,k)] = 0.0;
vData[3*INDEX(i,j,k)] = vData[3*INDEX(i,j,k)+1] = vData[3*INDEX(i,j,k)+2] = 0.0;

// インデックスファンクタによるアクセス
Index3DS pIndex = p.getIndex();
Index3DV vIndex = v.getIndex();
pData[pIndex(i,j,k)] = 0.0;
vData[vIndex(i,j,k)] = vData[vIndex(i,j,k)+1] = vData[vIndex(i,j,k)+2] = 0.0;
// または
vData[vIndex(i,j,k,0)] = vData[vIndex(i,j,k,1)] = vData[vIndex(i,j,k,2)] = 0.0;
```

図 4　Scalar3D, Vector3D クラスのおけるセルデータへのアクセス方法

現在[*6]の「京」では，直接 3 次元添字を付けてアクセスする方法では SIMD などの最適化が効かないため，この方法は推奨されない．他の 2 つの方法については，SIMD を含めて (うまく書くと) 同程度の最適化が効

[*5] Index3DV の 4 次元添字は (i,j,k,l) であるが，メモリ配置を Fortran 配列的に表すと (l,i,j,k) である．
[*6] 2012 年 4 月

いている.

5 フレームワークをチューニング・カスタマイズする

この章では，フレームワークをチューニング・カスタマイズするために必要な内部情報について説明する．

5.1 通信バッファクラス (CommBuffer)

通信バッファクラスは，パック/アンパック型通信を行うためのユーティリティクラスである．通信バッファクラスは，送信用の SendBuffer クラスと受信用の RecvBuffer クラスよりなる．

通信バッファクラスでは，通信対象となるデータを MPI_Pack/MPI_Unpack のようにスタック的にパック/アンパックするのではなく，ユーザは，パック時には予め指定されていた送信バッファ内のアドレス位置にデータをコピーし，アンパック時には予め指定されていたた受信バッファ内のアドレス位置からデータをコピーする．このため，MPI_Pack/MPI_Unpack と違い，パック/アンパックにおけるデータ操作を任意の順番に行える．

5.1.1 通信バッファクラスの使い方の概略

1. 送信対象となるデータ変数と，送信バッファ中のアドレス位置を格納する「バッファポインタ変数」を用意する．
2. 受信データを格納する変数と，受信バッファ中のアドレス位置を格納する「バッファポインタ変数」を用意する．
3. 通信バッファクラスの初期化
   (1) 送信用通信バッファクラスに，送信先ランク，送信データ変数のバイトサイズ，「バッファポインタ変数」へのポインタを登録．
   (2) 受信用通信バッファクラスに，送信元ランク，受信データ変数のバイトサイズ，「バッファポインタ変数」へのポインタを登録．
   (3) 全ての送受信データの登録後，送信用バッファクラス，受信用バッファクラスそれぞれに対して内部バッファ領域の確保を命じる．この時点で，各バッファポインタ変数に実際のバッファ中のアドレス位置が格納される．
4. データ送信
   (1) バッファポインタ変数に格納されているアドレスに送信データをコピー．
   (2) 全ての送信データのコピー後，送信用通信バッファクラスにデータ送信を命じる．
5. データ受信
   (1) 受信用通信バッファクラスにデータ受信を命じる．
   (2) バッファポインタ変数に格納されているアドレスから受信データをコピー．

初期化時の，「バッファポインタ変数」へのポインタ登録では，「バッファポインタ変数」へのポインタそのものでなく，それをラップした PointerSetter クラスのインスタンスをわたす．

■重要 送信側バッファクラスと受信用バッファクラス間での登録データの整合性 (データの数，サイズ，登録順) は，ユーザ側で責任を持つ必要がある．

5.1.2 PointerSetter クラス

任意の基本型 T のバッファポインタ変数にアドレス値をキャストするための，補助的なテンプレートクラス．基底クラス PointerSetterBase を多態的に継承している (図 5)．

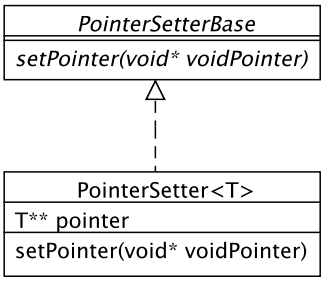

図 5　PointerSetter クラス

■ヘッダファイル　PointerSetter.h

■コンストラクタ

```
PointerSetter(T** pointer);
```

コンストラクタをとおして，内部にバッファポインタ変数 (T 変数のポインタ) へのポインタを格納する．

■バッファポインタ変数の設定

```
void setPointer(void* voidPointer);
```

void 型ポインタとして与えられたアドレス値を，T*型にキャストしてからバッファポインタ変数に設定する．このメソッドは，通信バッファクラス内で内部バッファ領域確保後に呼ばれる．

5.1.3 SendBuffer クラス

送信用通信バッファポインタクラス．

■ヘッダファイル　CommBuffer.h

■コンストラクタ

```
SendBuffer(int tag, const MPI::Comm& comm = MPI::COMM_WORLD);
```

タグ tag には，対応する受信用通信バッファクラスと同じ値を指定すること．

■送信データの登録

```
void setData(int rank, size_t size, PointerSetterBase* pointerSetter);
```

rank は送信先ランク番号．size はバイト単位での送信データ量．

■内部バッファ領域の確保

```
int allocateBuffer();
```

通信相手ノード数が返る．このメソッド内で，PointerSetter クラスをとおして，各バッファポインタ変数に値が設定される．

■データ送信

```
void send();        // ブロッキング送信
void sendBegin();   // ノンブロッキング送信開始
void sendEnd();     // ノンブロッキング送信終了待ち
```

5.1.4 RecvBuffer クラス

受信用通信バッファポインタクラス．

■ヘッダファイル CommBuffer.h

■コンストラクタ

```
RecvBuffer(int tag, const MPI::Comm& comm = MPI::COMM_WORLD);
```

タグ tag には，対応する送信用通信バッファクラスと同じ値を指定すること．

■送信データの登録

```
void setData(int rank, size_t size, PointerSetterBase* pointerSetter);
```

rank は送信元ランク番号．size はバイト単位での受信データ量．

■内部バッファ領域の確保

```
int allocateBuffer();
```

通信相手ノード数が返る．このメソッド内で，PointerSetter クラスをとおして，各バッファポインタ変数に値が設定される．

■データ受信

```
void recv();        // ブロッキング受信
void recvBegin();   // ノンブロッキング受信開始
void recvEnd();     // ノンブロッキング受信終了待ち
```

5.2 仮想セルアップデータ

仮想セルアップデータを作成するには，基底クラス VCUpdater を継承して，以下のメソッドを実装する必要がある．これらのメソッドは，VCUpdater 内では完全仮想メソッドとしてインタフェースのみが定義されている．

5.2.1 VCUpdate クラス

■ヘッダファイル VCUpdater.h

■データクラスの登録

```
// 仮想セルアップデータの持ち主データクラスの登録
void setDataClass(DataClass* dataClass);

// 隣接ブロックのデータクラスの登録
void setNeighbor(Face face, Subface subface, DataClass* dataClass);

// 隣接ブロックのデータクラスの抹消
virtual void clearNeighbor(Face face, Subface subface);
```

レベル差が 0 または -1 の場合は，subface=0 のみを使用する．

■通信バッファサイズの計算 [*7]

```
// 送信用
size_t getSendBufferByteSize(Face face) const;                      // レベル L → L
size_t getSendBufferByteSizeC2F(Face face, Subface subface) const; // レベル L → L+1
size_t getSendBufferByteSizeF2C(Face face, Subface subface) const; // レベル L+1 → L

// 受信用
size_t getRecvBufferByteSize(Face face) const;                      // レベル L → L
size_t getRecvBufferByteSizeC2F(Face face, Subface subface) const; // レベル L → L+1
size_t getRecvBufferByteSizeF2C(Face face, Subface subface) const; // レベル L+1 → L
```

レベル差 -1 の場合には，subface は隣接ブロック側のサブフェイスの指定に使用する．(以降のメソッドも同様)

■バッファポインタ変数へのポインタ (PointerSetter クラス) の取得

```
// 送信用
PointerSetterBase* getSendBufferPointerSetter(Face face, Subface subface);

// 受信用
PointerSetterBase* getRecvBufferPointerSetter(Face face, Subface subface);
```

■隣接データクラスからのコピー

```
void copyFromNeighbor(Face face);                      // レベル L → L
void copyFromNeighborC2F(Face face, Subface subface); // レベル L → L+1
void copyFromNeighborF2C(Face face, Subface subface); // レベル L+1 → L
```

[*7] 以下の説明では，末尾に F2C が付くメソッドはデータがレベル $L+1$ のブロック (Fine グリッド) からレベル L のブロック (Coarse グリッド) の方向に流れ，C2F が付くメソッドはデータがレベル L のブロック (Coarse グリッド) からレベル $L+1$ のブロック (Fine グリッド) の方向に流れることを示している．

■送信用バッファへのコピー

```
virtual void copyToCommBuffer(Face face);                      // レベル L → L
virtual void copyToCommBufferC2F(Face face, Subface subface); // レベル L → L+1
virtual void copyToCommBufferF2C(Face face, Subface subface); // レベル L+1 → L
```

■受信用バッファからのコピー

```
virtual void copyFromCommBuffer(Face face);                      // レベル L → L
virtual void copyFromCommBufferC2F(Face face, Subface subface); // レベル L → L+1
virtual void copyFromCommBufferF2C(Face face, Subface subface); // レベル L+1 → L
```

6　可視化用データ出力クラス

BCM データを可視化するための，簡単なデータ出力クラスを example/util ディレクトリに用意した．出力ファイルに Silo フォーマットを採用し，以下の機能を提供する．

- セル中心で定義された実数変数のスカラデータ，ベクトルデータを出力できる．
- 領域分割の状態をブロック単位で出力できる．
- 出力は，Silo の Multi-Block 出力機能を利用して，並列計算ノード単位での分散出力となる．
- 内部フォーマットの HDF5 によるデータ圧縮に対応

Silo ファイルは，VisIt により可視化することができる．ParaView も Silo ファイルの読み込みは可能だが，Multi-Block 形式の Silo ファイルには対応していないようである．

Silo ライブラリのインストール方法については，付録 A を参照のこと．

また，Silo ライブラリの使用方法については，Silo Users'Guide(https://wci.llnl.gov/codes/silo/documentation.html) を参照のこと．

6.1　SiloWriter クラス

BCM データの Silo フォーマットファイルへの出力機能を提供する．VisIt による実際の可視化方法については，サンプルプログラム (8 章) での具体例を参考のこと．

■ヘッダファイル　SiloWriter.h

■コンストラクタ

```
SiloWriter(const std::string& fileName, const std::string& meshName);
```

出力用に Silo フォーマットファイル fileName をオープンする．ファイル名の拡張子は `silo` とすること．`fileName` として `xxx.silo` を指定した場合，ランク 0 の並列計算ノードでマスターファイル xxx.silo を出力し，一般のノード (ランク 0 も含む) では分散出力データファイル xxx_n.silo(n はランク番号) を出力する．

`meshName` は，セル中心データ出力時に参照される，(セル単位での) メッシュ名．メッシュデータは，コンストラクタ呼び出し時に出力データファイルに出力される．

■デストラクタ

```
~SiloWriter();
```

Silo フォーマットファイルのクローズ．

■領域分割情報の出力

```
void writeDomain(const std::string& blockMeshName, const std::string& name);
```

ブロック境界を定めるメッシュデータをメッシュ名 `blockMeshName` で出力し，そのメッシュ上のスカラーデータとして担当計算ノードのランク番号をデータ名 `name` で出力する．

■スカラデータの出力

```
template <typename T>
void writeScalar(int dataClassID, const std::string& name);
```

セル中心で定義された，dataClassID をデータクラス ID に持つ Scalar3D<T>型データを，データ名 name で出力する．現バージョンでは，型 T が int などの整数型の場合も，float として出力される．

■ベクトルデータの出力

```
template <typename T>
void writeVector(int dataClassID, const std::string& name);
```

セル中心で定義された，dataClassID をデータクラス ID に持つ Vector3D<T>型データを，データ名 name で出力する．現バージョンでは，型 T が int などの整数型の場合も，float として出力される．

7　ブロック配置ツール

ここでは，サンプルプログラムで用いているブロック配置アルゴリズムと，その実装クラス群について説明する．

7.1　Octree と分割判定クラスによる BCM ブロック配置決定

計算対象領域を Octree(八分木) により分割することにより，BCM のブロック配置を構成することを考える．Octree の各リーフノードが BCM ブロックに対応する．以下では，簡単のために，計算対象領域は立方体形状とし，シングルルート (単一の Octree) によるブロック配置決定問題について説明する．

7.1.1　用語

分割レベル　Octree の階層レベル (ルートノードをレベル 0 とする)

ルートブロック　全計算領域 (分割レベル 0 に対応)

7.1.2　BCM ブロックの配置方針

次のデータとパラメータを入力とし，隣接 BCM ブロック (リーフノード) 間の分割レベル差が最大 1 に収まるように (2to1 制限)，Octree を構築する．

境界面データ　パラメトリックな幾何形状，または，STL ファイルによるポリゴンデータ

最大分割レベル　境界面と交わる BCM ブロックの分割レベル

最小分割レベル　このパラメータにより，BCM ブロックの最大サイズを制御

7.1.3　Octre 構築アルゴリズム

ルートブロックから始めて，以下の処理を再帰的に行う．

- 次のいずれかの条件を満たせば，リーフノードと判定して，処理を親ノードのもどす
  - すでに最大分割レベルに達した
  - そのノードに対応するブロック領域 (＋マージン) 内に境界面が存在しない
- 次のいずれかの条件を満たせば，子ノードを 8 つ生成 (ブロックを 8 分割) して，順番に子ノードに処理をわたす
  - 最小分割レベルにまだ達していない
  - そのノードに対応するブロック領域 (＋マージン) 内に境界面が存在する

7.1.4　ブロック分割判定クラス

このアルゴリズムでは，問題に依存するのは，再帰処理中に行われるブロックの分割判定部分だけである．そこで，この判定処理部分を「ブロック分割判定クラス」にまとめることする．ユーザは，このブロック分割判定クラスをカスタマイズすることにより，個々の問題 (いろいろな境界形状) に対応できるようになる．

以下に，ブロック分割判定クラスのインタフェースを規定する基底クラス Divider を示す．実際のブロック

分割判定クラスは，このクラスを継承して，分割判定を行うクラスをファンクタ (関数オブジェクト) として実装する．

```
/// ブロック分割判定クラス（基底クラス）.
class Divider {

public:

  /// ブロック（ノード）タイプ型
  enum NodeType {
    BRANCH,           ///< 枝（分割を続ける）
    LEAF_ACTIVE,      ///< アクティブなリーフノード（分割を終了）
    LEAF_NO_ACTIVE,   ///< 非アクティブなリーフノード（分割を終了）
  };

public:

  /// コンストラクタ.
  Divider() {}

  /// デストラクタ.
  virtual ~Divider() {}

  /// ブロックを分割するかどうかを判定.
  ///
  ///   @param[in] pedigree 対象ブロックの位置
  ///   @return ブロックタイプ
  ///
  virtual NodeType operator() (const Pedigree& pedigree) = 0;

};
```

図 6　ブロック分割判定 Divider クラス (基底クラス)

リーフノードのタイプに，アクティブ (LEAF_ACTIVE) と非アクティブ (LEAF_NO_ACTIVE) の 2 種類を設けた．これを利用することにより，固体領域中に完全に埋もれて流体計算の対象にならない BCM ブロックを非アクティブとするなどの使い方が可能である (図 7)．

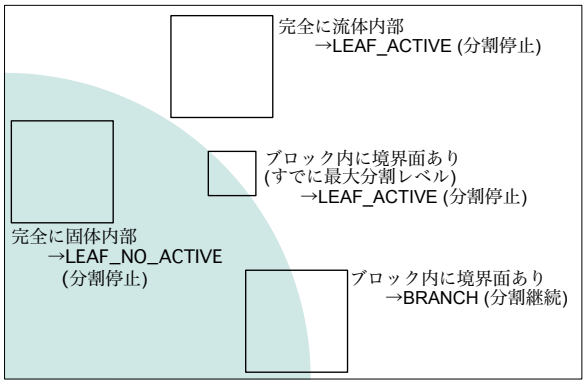

図 7　アクティブと非アクティブの使い分け

7.1.5　2to1 制限対応

前節のアルゴリズムを単純に使用すると，2to1 制限に対して，次の例のような問題が生じる．

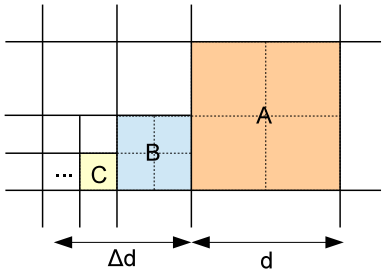

図 8　2to1 制限とマージン幅 Δd

Octree 構築途中のある時点で，図 8 のようなブロック配置になったとする．ここで，ブロック A，ブロック B，ブロック C，…，となるにしがたって，分割レベルがひとつずつ大きくなっている．このようなブロック配置に対して，ブロック A の分割判定を行うものとする．

- ブロック A 内に境界面が存在しなくても，ブロック B が分割される場合には，ブロック A も分割する必要がある．したがって，ブロック B 内での境界面の有無も調べる必要あある．
- ブロック B 内に境界面が存在しなくても，ブロック C が分割される場合には，ブロック B も分割する必要がある．したがって，ブロック C 内での境界面の有無も調べる必要あある．
- ブロック C 内に境界面が存在しなくても，…
- …
- …

したがって，2to1 制限を満たすためには，ブロック A の分割判定を行う時に，単純にブロック A 内だけではなく，その周りにマージン幅 Δd をとった領域内での境界面の有無をチェックする必要がある．ブロック A のサイズを d とすると，ブロック B のサイズは $d/2$，ブロック C のサイズは $d/4$，…，となるので，無限等比

級数の和により，$\Delta d = d$ とすれば十分であることが分かる．

実際には，最大分割レベルが指定されているので，Δd はもう少し小さくできる．

$$\Delta d = (1 - 2^{-s})d$$

ここで，パラメータ s は次式で与えられる．

$$s = (\text{最大分割レベル}) - (\text{ブロック A の分割レベル}) - 1$$

この，ブロックにマージン Δd をもたせた領域を考え，その中での境界面の有無をチェックすることによる分割判定法には，以下の利点がある．

1. Octree 構築とともに自動的に 2to1 制限が満たされる
2. マルチルートな Octree 構築にも，そのまま使える
3. 分割判定に仮想セル領域内での境界面の存在も考慮したい場合には，最大分割レベルでの仮想セル幅分を Δd に追加するだけでよい

7.2 ブロック配置ユーティリティクラス群

前節のアルゴリズムをマルチルート Octree に拡張し，領域分割まで行うユーティリティクラス群を，exmaples/util ディレクトリに用意した．

RootGrid クラス　ルートブロックの空間配置を管理
Node クラス　Octree ノードクラス
Pedigree クラス　Octree ノードのツリー内での位置
BoundingBox クラス　直方体領域クラス
BCMOctree クラス　マルチルート Octree の構築，管理
MultiRootDivider クラス　マルチルート Octree 用 Divider 継承クラス
Partition クラス　領域分割管理クラス

なお，個々の BCM ブロックデータ作成時には，そのブロックの隣接情報が必要になる．この隣接情報計算を最も簡単に行う方法は，各並列計算ノードで，全 Octree 情報を重複に保持しておくことである．そのためには，Octree 構築を各並列計算ノードで重複して行う，あるいは，ランク 0 の並列計算ノードのみで Octree 構築を行い，他の計算ノードに Octree データをブロードキャストする方法が考えられる．BCMOctree クラスでは後者の方法にも対応できるように，Octree データのブロードキャストメソッドも備えている．

以下にブロック配置ユーティリティクラスの主要クラスについて説明する．詳細については，Doxygen 出力の HTML 文書を参照のこと．

7.2.1 RootGrid クラス

ルートブロックの空間配置を管理するクラス．

■ヘッダファイル　RootGrid.h

■コンストラクタ

```
RootGrid(int nx, int ny, int nz);
RootGrid(const Vec3i& n);
```

ルートブロック空間配置を 3 次元整数ベクトル n=(nx,ny,nz) で指定する.

■各方向に周期境界条件を設定

```
void setPeriodicX();
void setPeriodicY();
void setPeriodicZ();
```

■各方向の周期境界条件を解除

```
void clearPeriodicX();
void clearPeriodicY();
void clearPeriodicZ();
```

■ルートブロック位置インデクスをルート ID に変換

```
int index2rootID(int ix, int iy, int iz) const;
```

ルート ID とは，個々の Octree にふった 0 から始まる通し番号.

■ルート ID から位置インデクスを計算

```
int rootID2indexX(int rootID) const;
int rootID2indexY(int rootID) const;
int rootID2indexZ(int rootID) const;
```

■指定した面が外部境界 (周期境界も含む) かどうかチェック.

```
bool isOuterBoundary(int rootID, Face face) const;
```

指定した面が外部境界なら true を返す.

■隣接するルートブロックの ID を取得

```
int getNeighborRoot(int rootID, Face face) const;
```

隣接ルートが存在しない場合は -1 を返す.

7.2.2 BCMOctree クラス

マルチルート Octree の構築，管理を行うクラス.

■ヘッダファイル BCMOctree.h

■コンストラクタ

```
BCMOctree(RootGrid* rootGrid, Divider* divider, Ordering ordering);
```

rootGrid でルートブロックの空間配置を指定し，divider にはブロック分割判定メソッドを実装した

Divider 継承クラスを指定する．ordering には，領域分割時に使用するリーフノードのリストアップ順を決める以下のいずれかの定数を指定する．

BCMOctree::Z　Z(Morton) オーダリング
BCMOctree::HILBERT　ヒルベルトオーダリング
BCMOctree::RANDOM　ランダムシャッフル

現実装では，このオーダリング指定が効くのは個々の Octree 内だけである．ルートノード間は，オーダリング指定によらず，常に rootGrid 内に登録されたルート ID 順に巡ることになっている．

このコンストラクタが呼ばれると，内部で Octree データの構築，リーフノードリストの作成までを行う．

■デストラクタ

```
~BCMOctree();
```

デストラクタ呼び出し時に，rootGrid と divider も解放される．

■Octree データの他並列計算ノードへのブロードキャスト

```
void broadcast(MPI::Intracomm& comm = MPI::COMM_WORLD);
```

■Octree データをマスター並列計算ノードから受信

```
static BCMOctree* ReceiveFromMaster(MPI::Intracomm& comm = MPI::COMM_WORLD);
```

ランク 0 計算ノードで broadcast メソッドを呼び，それ以外の計算ノードでこのメソッドを呼ぶ．

■リーフノード総数を取得

```
int getNumLeafNode() const;
```

アクティブなリーフノードの総数が返る．

■リーフノードリストを取得

```
std::vector<Node*>& getLeafNodeArray();
const std::vector<Node*>& getLeafNodeArray() const;
```

このリーフノードリストには，非アクティブなリーフノードは含まれない．

■指定したノードの面が外部境界 (周期境界も含む) かどうかチェック

```
bool checkOnOuterBoundary(const Node* node, Face face) const;
```

外部境界なら true を返す．

■指定されたノードに対応するブロックの原点位置を取得

```
Vec3r getOrigin(const Node* node) const;
```

■指定されたノードの隣接情報を計算

```
NeighborInfo* makeNeighborInfo(const Node* node,
                               const Partition* partition) const;
```

Partition は，後述する領域分割管理クラス．隣接面で 2to1 制限が満たされているかどうかのチェックも行う．

7.2.3 MultiRootDivider クラス

マルチルート Octree 用 Divider 継承クラス (抽象クラス)．このクラスを継承して，ブロック分割判定クラスをカスタマイズする．

■コンストラクタ

```
MultiRootDivider(const Vec3r& origin, double rootLength,
                 const RootGrid* rootGrid);
```

origin には $(0,0,0)$ に位置するルートブロック (ルート ID が 0 のルートブロック) の原点座標を，rootLength にはルートブロックの辺長を，rootGrid ではルートブロックの空間配置を指定する．

■境界探査領域の決定

```
BoundingBox defineSearchRegion(const Pedigree& pedigree, int maxLevel);
```

マージン幅 Δd を考慮した，境界面探査領域を決定する．このメソッドは protected で，このクラスを継承したブロック分割判定の実装クラスから利用することができる．

7.2.4 Partition クラス

1 次元領域分割用ユーティリティクラス．

■コンストラクタ

```
Partition(int nProcs, int nItems);
```

要素数 nItems からなる 1 次元集合を，nProcs 個の領域に等分割する．

■先頭要素番号の取得

```
int getStart(int rank) const;
```

■末尾要素番号の取得

```
int getEnd(int rank) const;
```

実際には，末尾要素番号 $+1$ が返る．

■担当要素数を取得

```
int getNum(int rank) const;
```

■担当プロセスを取得

```
int getRank(int i) const;
```

i 番目の要素を担当している並列計算ノードのランク番号が返る．

■分割内容を出力

```
void print() const;
```

7.3 使用例

7.3.1 Octree 構築を重複に行う場合

以下のコードで，UserDefinedDivider はブロック分割判定クラスの実装クラス．UserDefinedBlockFactoryMethod はノード情報から BCM ブロックデータを作成するファクトリメソッドである．これらは，問題依存なため，
具体的な実装例は，exmaples ディレクトリ内のサンプルコードを参照のこと．

```
int main(int argc, char** argv)
{
  MPI::Init(argc, argv);
  MPI::Comm& comm = MPI::COMM_WORLD;
  int myRank = comm.Get_rank();
  int numProcs = comm.Get_size();

  // ルートブロック（単位立方体を 2x2x2 に配置）
  int rootLength = 1.0;
  Vec3r origin(0.0, 0.0, 0.0);
  Vec3i rootN(2,2,2);

  RootGrid* rootGrid = new RootGrid(rootN);

  // カスタマイズしたブロック分割判定クラス
  Divider* divider = new UserDefinedDivider(...);

  // ヒルベルトオーダリング
  BCMOctree::Ordering ordering = BCMOctree::HILBERT

  // Octree 構築
  BCMOctree* tree = new BCMOctree(rootGrid, divider, ordering);

  // リーフノード総数を取得
  int numLeafNode = tree->getNumLeafNode();

  // 領域分割を実施
  Parition* partition = new Partition(numProcs, numLeafNode);
  if (myRank == 0) {
    std::cout << std::endl << "Partitioning" << std::endl;
    partition->print();
  }

  // リーフノードリストの取得
  std::vector<Node*>& leafNodeArray = tree->getLeafNodeArray();

  // ブロックマネージャの取得
  BlockManager& blockManager = BlockManager::getInstance();

  // BCM ブロックの生成と登録
  for (int id = partition->getStart(myRank); id < partition->getEnd(myRank); id++) {
    Node* node = leafNodeArray[id];
```

```
    Block* block = UserDefinedBlockFactoryMethod(node);
    blockManager.registerBlock(block);
  }
  blockManager.endRegisterBlock();

  // ブロック配置情報の表示
  blockManager.printBlockLayoutInfo();

  delete tree;
  delete partition;

  /* 以下略 */
}
```

7.3.2 マスター並列計算ノードのみが Octree 構築を行う場合

```
int main(int argc, char** argv)
{
  MPI::Init(argc, argv);
  MPI::Comm& comm = MPI::COMM_WORLD;
  int myRank = comm.Get_rank();
  int numProcs = comm.Get_size();

  BCMOctree* tree;

  if (myRank == 0) {

    // ルートブロック（単位立方体を 2x2x2 に配置）
    int rootLength = 1.0;
    Vec3r origin(0.0, 0.0, 0.0);
    Vec3i rootN(2,2,2);

    RootGrid* rootGrid = new RootGrid(rootN);

    // カスタマイズしたブロック分割判定クラス
    Divider* divider = new UserDefinedDivider(...);

    // ヒルベルトオーダリング
    BCMOctree::Ordering ordering = BCMOctree::HILBERT

    // Octree 構築
    tree = new BCMOctree(rootGrid, divider, ordering);

    // Octree 情報をブロードキャスト
    tree->broadcast();

  }
  else { // rank0 以外

    // Octree 情報を受信
    tree = BCMOctree::ReceiveFromMaster();

  }

  /* 以降は，重複に Octree を構築した場合と同じ */

  // リーフノード総数を取得
  int numLeafNode = tree->getNumLeafNode();
```

```
  // 領域分割を実施
  Parition* partition = new Partition(numProcs, numLeafNode);
  if (myRank == 0) {
    std::cout << std::endl << "Partitioning" << std::endl;
    partition->print();
  }

  // リーフノードリストの取得
  std::vector<Node*>& leafNodeArray = tree->getLeafNodeArray();

  // ブロックマネージャの取得
  BlockManager& blockManager = BlockManager::getInstance();

  // BCM ブロックの生成と登録
  for (int id = partition->getStart(myRank); id < partition->getEnd(myRank); id++) {
    Node* node = leafNodeArray[id];
    Block* block = UserDefinedBlockFactoryMethod(node);
    blockManager.registerBlock(block);
  }
  blockManager.endRegisterBlock();

  // ブロック配置情報の表示
  blockManager.printBlockLayoutInfo();

  delete tree;
  delete partition;

  /* 以下略 */
}
```

8 サンプルプログラム

samples ディレクトリに以下のサンプルプログラムを収録した.

ParametricDivider　パラメトリックな境界面指定によるブロック配置例
PolygonDivider　ポリゴンデータ境界面指定によるブロック配置例
SOR1　SOR による拡散方程式ソルバ (線形問題による厳密解との比較)
SOR2　SOR による拡散方程式ソルバ (計算結果の可視化)
ParallelMeshGeneration　メッシュ生成プロトタイプ

8.1 ParametricDivider (**パラメトリックな境界面指定によるブロック配置**)

このサンプルプログラムでは，以下の 4 つのブロック分割判定クラスを実装している．

FlatDivider **クラス**　最大分割レベルまで均等に分割 (図 9)
SimpleDivider **クラス**　外部境界に接するブロックのみ分割していく (図 10)
SphereDivider **クラス**　球の外部
CylinderDivider **クラス**　Z 方向に延びた円筒の内部

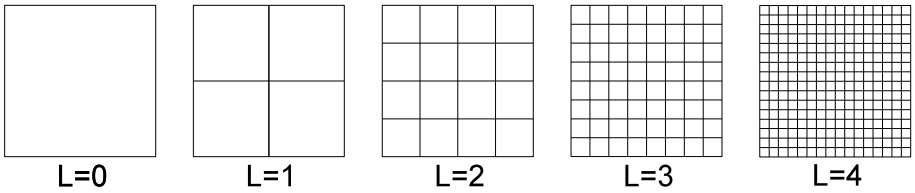

図 9　FlatDivider クラスによるブロック配置

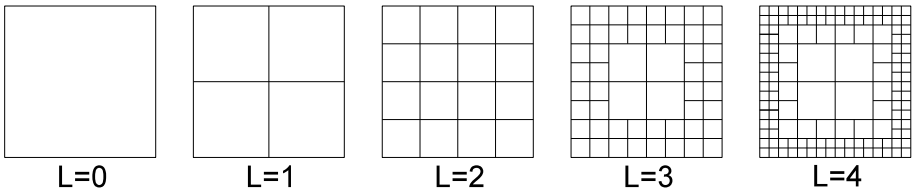

図 10　SimpleDivider クラスによるブロック配置

なお，このプログラムのコンパイル・リンクには，Silo および HDF5 ライブラリが必要である．
マルチルートに対応しているが，ルートブロックの辺長は 1.0，原点は $(0, 0, 0)$ に固定されている．

■設定パラメータ　プログラム実行時に引数として指定する設定ファイル内に，「キーワード = 値」の形式で，以下のパラメータを指定する．

`rootGrid`　ルートブロック配置 (`nx ny nz`)
`minLebel`　最小分割レベル
`maxLebel`　最大分割レベル
`treeType`　`flat`, `simple`, `sphere`, `cylinder` のいずれか
`ordering`　`Z`, `Hilbert`, `random` のいずれか
`sphereCenter`　球の中心座標 (`x y z`)
`sphereRadius`　球の半径
`cylinderCenter`　円柱の中心座標 (`x y`)
`cylinderRadius`　円柱の半径
`size`　ブロック内のセル分割数

`output`　結果出力ファイル名 (拡張子を「`silo`」とすること)

■可視化方法　VisIt による可視化時には，出力された Silo ファイルは全て同じディレクトリ内に置く必要がある．設定ファイルの `output` キーワードで指定したファイルがマスターファイルで，このファイルを VisIt で開く．可視化可能なデータは以下のとおり．

メッシュ mesh　セル単位でのメッシュ
メッシュ block_mesh　ブロック単位でのメッシュ
スカラー domain　各ブロックの担当ランク番号
スカラー scalar　原点からの距離
ベクトル vector　位置ベクトル

■計算結果例

`rootGrid = 1 1 1`, `treeType = simple`, `minLevel = 0`, `maxLebel = 4` を 126 ノードで計算．

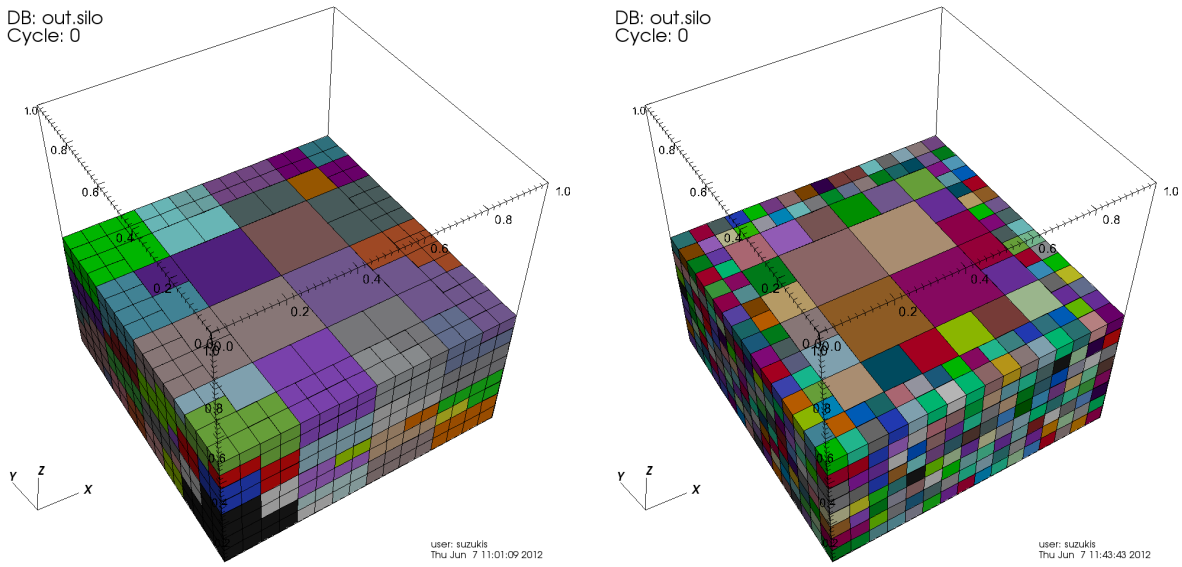

図 11　SimpleDivider ブロック配置と領域分割の様子．左図は Z オーダリング，右図はランダムシャッフル．block_mesh を”Mesh” 表示，domain を”Pseudocolor” 表示し，$z = 0.5$ でクリッピングしたもの．

`rootGrid = 2 1 1`, `treeType = sphere`, `sphereCenter = 0.8 0.5 0.5`, `shhereRadius = 0.25`, `ordering = Hilbert`, `minLevel = 0`, `maxLebel = 5` を 8 ノードで計算．

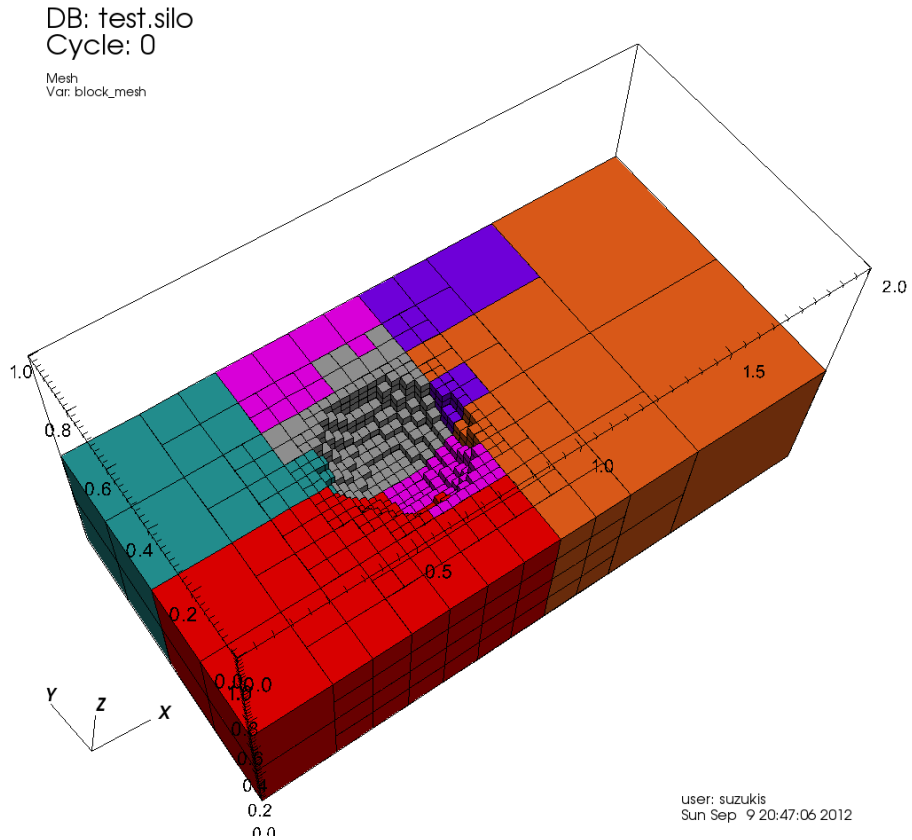

図 12　SphereDivider ブロック配置と領域分割の様子．block_mesh を”Mesh” 表示，domain を”Pseudocolor” 表示し，$z = 0.5$ でクリッピングしたもの．

`rootGrid = 1 1 2`, `treeType = cylinder`, `cylinderCenter = 0.5 0.5`, `cylinderRadius = 0.4`, `minLevel = 0`, `maxLebel = 5` を 7 ノードで計算．

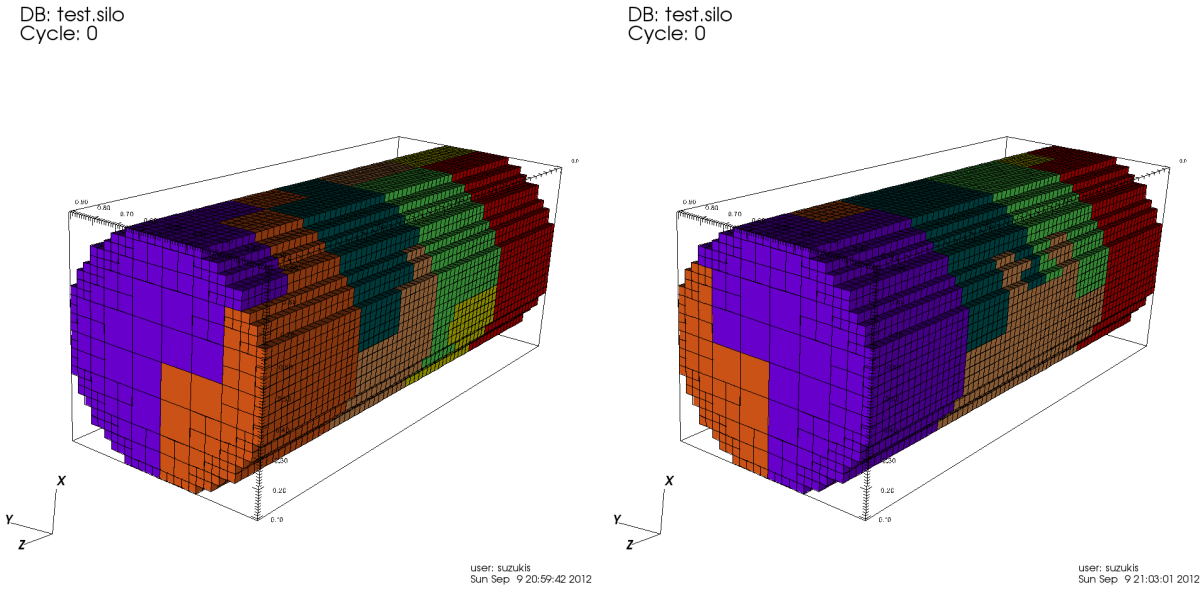

図 13　CylinderDivider ブロック配置と領域分割の様子．左図は Z オーダリング，右図はヒルベルトオーダリング．block_mesh を”Mesh” 表示，domain を”Pseudocolor” 表示．

8.2 PolygonDivider (ポリゴンデータ境界面指定によるブロック配置)

ポリゴンデータ境界面指定によるブロック配置のサンプルプログラム．

このプログラムのコンパイル・リンクには，Silo および HDF5 ライブラリが必要である．

また，search_nearest パッチを当てた Polylib-2.0.x ライブラリが必要である．このパッチにより，Plylib に，指定された点に最も近いポリゴンを検索する search_nearest_polygon を追加している．PolygonDivider クラスでは，このメソッドを利用して，あるブロックが境界面の表裏どちらがにいるかを判定し，完全に固体側に埋もれたブロックは非アクティブとして，計算対象から除外している．

この表裏判定には，ポリゴンの法線ベクトルを用いている．したがって，PolygonDivider クラスが機能するには，入力 STL ファイル中の各ポリゴンの放線ベクトルに有効な値 (少なくとも表裏の向きに整合性があること) が格納されている必要がある．

■設定パラメータ　プログラム実行時に引数として指定する設定ファイル内に，「キーワード = 値」の形式で，以下のパラメータを指定する．

`origin`　ルートブロック原点 (`ox oy oz`)
`rootLength`　ルートブロック辺長
`rootGrid`　ルートブロック配置 (`nx ny nz`)
`minLebel`　最小分割レベル
`maxLebel`　最大分割レベル
`polylibConfig`　Polylib 設定 XML ファイル
`polygonGroup`　境界面とするポリゴングループ名
`polygonInsideOut`　ポリゴンの表裏を逆転するフラグ (true/fase)
`ordering`　`Z`, `Hilbert`, `random` のいずれか
`size`　ブロック内のセル分割数
`output`　結果出力ファイル名 (拡張子を「`silo`」とすること)

■可視化方法　VisIt による可視化時には，出力された Silo ファイルは全て同じディレクトリ内に置く必要がある．設定ファイルの `output` キーワードで指定したファイルがマスターファイルで，このファイルを VisIt で開く．可視化可能なデータは以下のとおり．

メッシュ mesh　セル単位でのメッシュ
メッシュ block_mesh　ブロック単位でのメッシュ
スカラー domain　各ブロックの担当ランク番号
スカラー scalar　原点からの距離
ベクトル vector　位置ベクトル

■計算結果例

STL ファイル STL_DATA/pato.stl, `origin = -50.0 -50.0 0.0`, `rootLength = 100.0`, `rootGrid = 1 1 1`, `minLevel = 0`, `maxLebel = 4` を 4 ノードで計算．

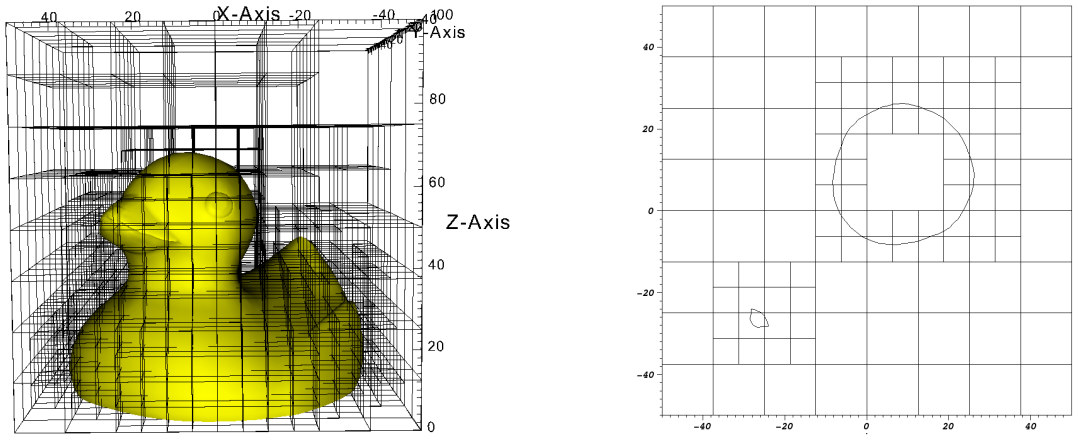

図 14　PolygonDivider ブロック配置 (pato.stl)．block_mesh を”Mesh” 表示，STL ファイルを読み込み”Subset” から”STL_Mesh” を表示したもの．右図は $z = 40$ でクリッピング．

8.3 SOR1 (SOR による拡散方程式ソルバ — 線形問題による厳密解との比較)

単位立方体領域 $(0 \leq x, y, z, \leq 1)$ において，ソース項 0 の Poisson 方程式を中心差分で差分化し，SOR で解く．

ブロックの配置は，flat(図 9) と simple(図 10) から選択できる．

境界条件は，指定された次元方向のみ Dirichlet 条件 (値は負側 0，正側 1) とし，その他の面は周期境界条件としている．

SOR 計算における反復数は，2 つのパラメータ `nLoopOuter` と `nLoopInner` により制御される (図 15)．各ブロック毎に独立に反復数 `nLoopInner` の SOR を行い，その後に仮想セル同期を行う．このステップを `nLoopOuter` 回繰り返すことになる．

```
for (int i = 0; i < nLoopOuter; i++) {
  for (ブロックに対するループ) {
    for (int j = 0; j < nLoopInner; j++) {
      ブロック内の SOR 計算 1 ステップ
    }
  }
  仮想セル同期
}
```

図 15　SOR 計算における反復制御

■設定パラメータ　プログラム実行時に引数として指定する設定ファイル内に，「キーワード = 値」の形式で，以下のパラメータを指定する．

- `minLebel`　最小分割レベル
- `maxLebel`　最大分割レベル
- `treeType`　`flat`, `simple` のいずれか
- `ordering`　`Z`, `Hilbert`, `random` のいずれか
- `type`　`x`, `y` または `z`．Dirichlet 境界条件面を指定する
- `nLoopOuter`,`nLoopInner`　SOR の反復制御パラメータ (1 以上の整数)
- `omega`　SOR の加速係数 (正の実数)
- `size`　ブロック内のセル分割数 (偶数であること)
- `vc`　仮想セル幅 (1 以上で `size/2` 以下の整数)
- `separateVCUpdate`　`true` または `false`(ディフォルト)．`true` の場合，仮想セル同期を xyz 方向別々に行う

■計算結果例　以下に計算時の出力例を示す．末尾の `errorMax` の値が，厳密解との差 (絶対値) の全計算セルにおける最大値である．

```
# of MPI processes = 4
```

```
# of OpenMP threads = 1

Configuration file: test.conf
  min level:          0
  max level:          3
  tree type:          simple
  type:               x
  ordering:           Hilbert
  block size:         4
  vc width:           1
  boundary values:    0.00000, 1.00000
  SOR oemga:          1.20000
  inner loop length:  10
  outer loop length:  400
  output file:        out.dat
  verbose message:    off

Partitioning
0: [0:113] #114
1: [114:227] #114
2: [228:341] #114
3: [342:455] #114

Block Layout Information
  Min level: 2
  Max level: 3
  Number of blocks
    L=2: 8
    L=3: 448
    total: 456
  Number of total blocks / node
    ave: 114.000
    min: 114
    max: 114
     sd: 0.00000
  Number of faces
    intra-node
        dLevel=-1: 96
        dLevel= 0: 2024
        dLevel=+1: 96
            total: 2216
    inter-node
        dLevel=-1: 0
        dLevel= 0: 464
        dLevel=+1: 0
            total: 464
  Number of total inter-node faces / node
    ave: 116.000
    min: 116
    max: 116
     sd: 0.00000

errorMax = 0.0329891
```

8.4 SOR2 (SOR による拡散方程式ソルバ — 計算結果の可視化)

SOR1 と同様な計算領域，ブロック配置に対して，適当なソース項を与えて以下の解を持つようにした Poisson 方程式を中心差分で差分化し，SOR で解く．

$$f(x,y,z) = \sin(2\pi x)\sin(2\pi y)\sin(2\pi z), \qquad 0 \le x,y,z \le 1$$

境界条件は，全ての外部境界面で $f(x,y,z) = 0$ とする．

このプログラムのコンパイル・リンクには，Silo および HDF5 ライブラリが必要である．

■設定パラメータ　プログラム実行時に引数として指定する設定ファイル内に，「キーワード = 値」の形式で，以下のパラメータを指定する．

`minLebel`　最小分割レベル
`maxLebel`　最大分割レベル
`treeType`　`flat`, `simple` のいずれか
`ordering`　`Z`, `Hilbert`, `random` のいずれか
`nLoopOuter,nLoopInner`　SOR の反復制御パラメータ (1 以上の整数)
`omega`　SOR の加速係数 (正の実数)
`size`　ブロック内のセル分割数 (偶数であること)
`vc`　仮想セル幅 (1 以上で `size/2` 以下の整数)
`separateVCUpdate`　`true` または `false`(ディフォルト)．`true` の場合，仮想セル同期を xyz 方向別々に行う
`output`　結果出力ファイル名 (拡張子を「`silo`」とすること)

■可視化方法　VisIt による可視化時には，出力された Silo ファイルは全て同じディレクトリ内に置く必要がある．設定ファイルの `output` キーワードで指定したファイルがマスターファイルで，このファイルを VisIt で開く．可視化可能なデータは以下のとおり．

メッシュ mesh　セル単位でのメッシュ
メッシュ block_mesh　ブロック単位でのメッシュ
スカラー domain　各ブロックの担当ランク番号
スカラー result　計算結果
スカラー error　計算誤差

■計算結果例　ブロック配置 simple, ツリーレベル 4，ブロック内セル分割数 16 で，126 ノードを用いて計算．

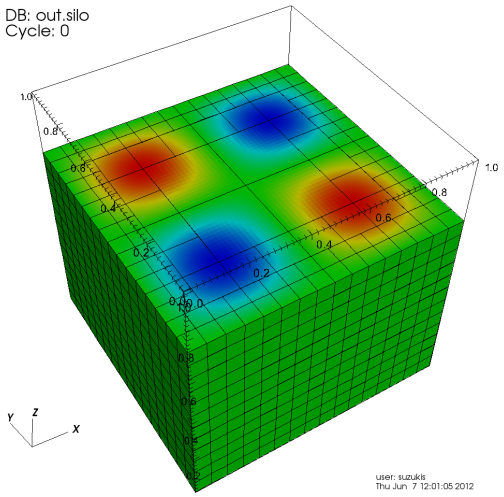

図 16　計算結果．block_mesh を”Mesh” 表示，result を”Pseudocolor” 表示し，$z = 0.75$ でクリッピング．

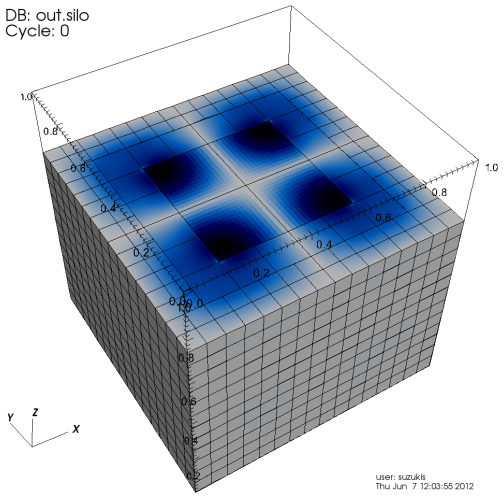

図 17　計算誤差．block_mesh を”Mesh” 表示, error を”Pseudocolor” 表示し，$z = 0.75$ でクリッピング．色の濃さ＝絶対誤差．

8.5 ParallelMeshGeneration (メッシュ生成プロトタイプ)

このサンプルプログラムは，大規模並列計算時におけるソルバ内部でのメッシュデータ作成のプロトタイプとなることを意図している．図 18 に計算手順を示す．全 Octree データは全ての計算ノードで重複して保持するが，入力データはランク 0 ノードだけが読み込み，他の計算ノードには最小限のポリゴンデータを送信するようにしている[*8]．

本プログラムでは，図 18 の計算手順のうち，ポリゴンデータの各計算ノードへの送信までを行っている．各ノードでの担当ブロック内のボクセル化は行っていない．プログラム終了時に Polylib の save_parallel メソッドにより，各計算ノードが保持しているポリゴンデータを出力している．この出力結果をチェックすることにより，並列メッシュ生成に必要な手順が正しく行われていることの確認ができる．

なお，このプログラムのコンパイル・リンクには，Polylib-2.0.x，Silo および HDF5 ライブラリが必要である．

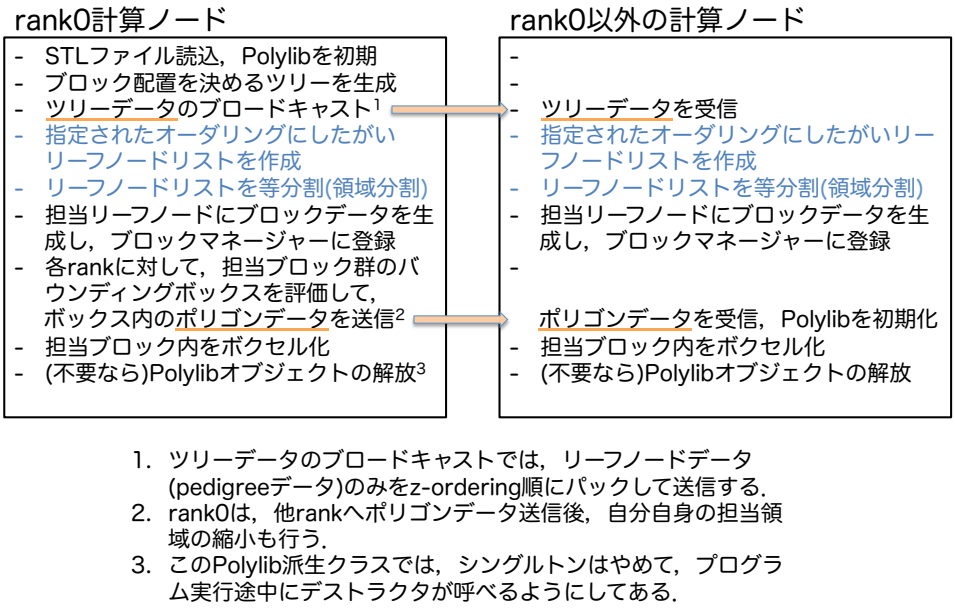

図 18　メッシュ生成計算手順．青字の箇所では，全計算ノードで重複したデータに対して重複した計算を行う．

■PolygonBBoxDivider **クラス**　本サンプルプログラムでは，ブロック分割判定クラスとして PolygonDivider クラスを改良した PolygonBBoxDivider クラスを用いている．PolygonBBoxDivider クラスでは，2 種類のパラメータにより，ブロック配置をコントロールできる．

1.「ポリゴングループ，分割レベル」ペアのリスト
2.「バウンディングボックス (直方体領域)，分割レベル」ペアのリスト

[*8] 現バージョンのプログラムでは，各計算ノードの担当領域をバウンディングボックス (直方体領域) として扱っているので，実際には不必要なポリゴンデータも送信している．

「ポリゴングループ，分割レベル」ペアリストにより，複数のポリゴングループからなる境界面を指定できる．また，ポリゴングループごとに異なる最大分割レベルをもたせることが可能である．「バウンディングボックス，分割レベル」ペアリストでは，境界ポリゴンの存在に関係なく，任意の領域をバウンディングボックスとして指定することにより，その領域と交わるブロックの最大分割レベルを調節することが可能となる．なお，バウンディングボックスとして，座標値に同一値を指定することにより，面領域，線分，点上での最大分割レベルを指示することもできる．

PolygonBBoxDivdier クラスでは，PolygonDivider クラスと違い，リーフノードブロック位置の境界面に対する表裏判定は行っていない．しがたって，全てのリーフノードブロックはアクティブとして扱われる．

■BCMPolylib クラス　並列化された Polylib の MPIPolylib クラスは，3 次元ブロック領域分割のみに対応している．そこで，新たに BCM ブロック配置の領域分割に対応した BCMPolylib クラスを，MPIPolylib クラスの派生クラスとして実装した．また，BCMPolylib クラスでは，メッシュ生成後に不要なポリゴンデータとともに解放できるように，シングルトンは廃し，コンストラクタおよびデストラクタを公開メソッドとした．BCMPolylib クラスの詳細については，Doxygen 出力の HTML 文書を参照のこと．

■設定パラメータ　プログラム実行時に引数として指定する設定ファイル内に，「キーワード = 値」の形式で，以下のパラメータを指定する．

`origin`　ルートブロック原点 (`ox oy oz`)
`rootLength`　ルートブロック辺長
`rootGrid`　ルートブロック配置 (`nx ny nz`)
`minLebel`　最小分割レベル
`polylibConfig`　Polylib 設定 XML ファイル
`polygonGroupList`　「ポリゴングループ名，分割レベル」ペアのリスト
`boundingBoxList`　「バウンディングボックス，分割レベル」ペアのリスト
`ordering`　`Z`, `Hilbert`, `random` のいずれか
`size`　ブロック内のセル分割数
`vc`　仮想セル幅
`output`　結果出力ファイル名 (拡張子を「`silo`」とすること)

「ポリゴングループ名，分割レベル」ペアリストの指定は次のように記述する．

```
polygonGroupList =
          root/ducky  3
          root/bunny  4
```

この例では，ポリゴングループ `root/ducky` 境界面の分割レベルを 3 に，ポリゴングループ `root/bunny` 境界面の分割レベルを 4 にに指定している．

「バウンディングボックス，分割レベル」ペアリストの指定は次のように記述する．

```
boundingBoxList =
        -50.0 -50.0 0.0  150.0 50.0 0.0   4
        125.0 25.0 75.0  125.0 25.0 75.0  5
```

この例では，点 (−50.0, −50.0, 0.0) と点 (150.0, 50.0, 0.0) で張られるバウンディングボックス (実際には面) の分割レベルを 4 に，点 (125.0, 25.0, 75.0) と点 (125.0, 25.0, 75.0) で張られるバウンディングボックス (実際には点) の分割レベルを 5 に指定している．

■可視化方法　VisIt による可視化時には，出力された Silo ファイルは全て同じディレクトリ内に置く必要がある．設定ファイルの `output` キーワードで指定したファイルがマスターファイルで，このファイルを VisIt で開く．可視化可能なデータは以下のとおり．

メッシュ mesh　セル単位でのメッシュ
メッシュ block_mesh　ブロック単位でのメッシュ
スカラー domain　各ブロックの担当ランク番号

■計算結果例　STL ファイル STL_DATA/pato.stl, `origin = -50.0 -50.0 0.0`, `rootLength = 100.0`, `rootGrid = 1 1 1`, `minLevel = 0` を 4 ノード，ヒルベルトオーダリングで計算．

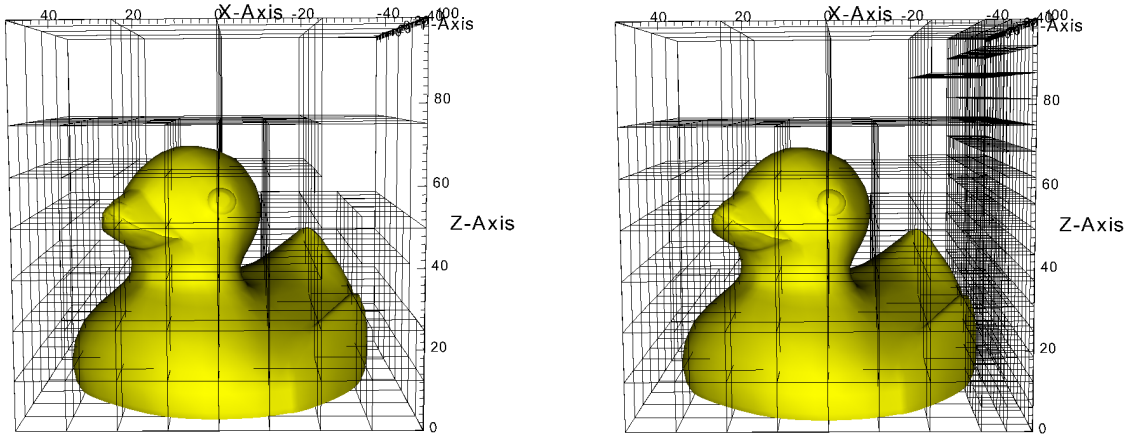

図 19　ポリゴン表面の分割レベルを 3 に指定．右図では，さらにバウンディングボックスにより $z = 0$ 面 (右面) での分割レベルを 4 に指定

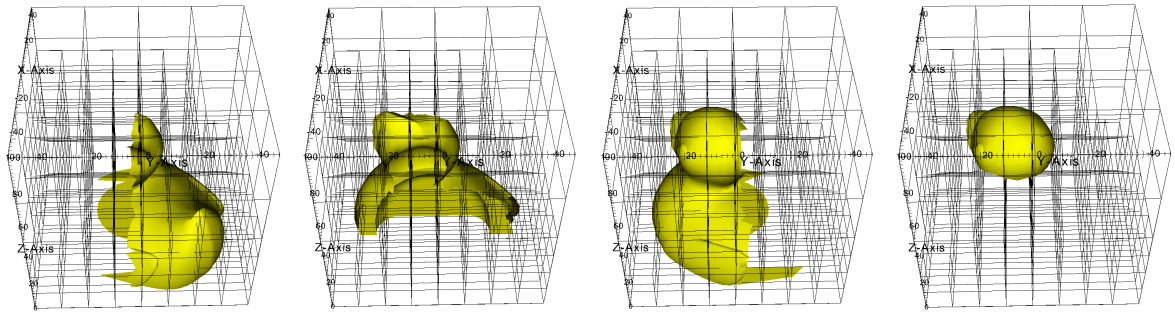

図 20　各計算ノードが保持するポリゴンデータ

付録 A　Silo の FX10, 京へのイントール方法

ここでは，BCM データ出力用 SiloWriter クラスの使用時に必要な，Silo ライブラリ[*9]の Mac OS X, FX10，京へのインストール方法について説明する．

A.1　HDF5 のインストール

まず，Silo が内部フォーマットとして利用する HDF5 ライブラリ[*10]をインストールする．なお，HDF5 なしでも Silo ライブラリのインストールは可能だが，その場合にはデータ圧縮機能が使用できない．

A.1.1　Mac OS X へのインストール

http://hpc.sourceforge.net/ で配布されている gcc, gfortran を利用していることを前提とする (環境変数 PATH において，/usr/local/bin が/usr/bin の前に記述されていること)．/home_direcotory/opt ディレクトリ以下に，ヘッダファイル，ライブラリファイル等をインストールするものとする．

```
tar zxvf hdf5-1.8.9.tar.gz
cd hdf5-1.8.9
./configure --prefix=/home_directory/opt --enable-fortran -enable-cxx
make
make install
```

A.1.2　FX10 へのインストール

次の Web ページを参考にした．ただし，このページに書いてあるような cmake のインストールは必要ない．

http://todo.issp.u-tokyo.ac.jp/ja/members/wistaria/log/1wqum2

以下では，/home_direcotory/FX10 ディレクトリにクロスコンパイル用の環境 (include ディレクトリ, lib ディレクトリ) をインストールするものとする．

```
tar zxvf hdf5-1.8.9.tar.gz
cd hdf5-1.8.9
echo 'hdf5_cv_have_lfs=yes' > myconfig.cache
echo 'hdf5_cv_gettimeofday_tz=yes' >> myconfig.cache
echo 'hdf5_cv_vsnprintf_works=yes' >> myconfig.cache
echo 'hdf5_cv_system_scope_threads=yes' >> myconfig.cache
echo 'hdf5_cv_direct_io=no' >> myconfig.cache
echo 'hdf5_cv_ldouble_to_integer_works=yes' >> myconfig.cache
echo 'hdf5_cv_ulong_to_float_accurate=yes' >> myconfig.cache
echo 'hdf5_cv_fp_to_ullong_accurate=yes' >> myconfig.cache
echo 'hdf5_cv_fp_to_ullong_right_maximum=yes' >> myconfig.cache
echo 'hdf5_cv_ldouble_to_uint_accurate=yes' >> myconfig.cache
echo 'hdf5_cv_ullong_to_ldouble_precision=yes' >> myconfig.cache
echo 'hdf5_cv_fp_to_integer_overflow_works=yes' >> myconfig.cache
```

[*9] https://wci.llnl.gov/codes/silo/
[*10] http://www.hdfgroup.org/HDF5/

```
echo 'hdf5_cv_ldouble_to_long_special=yes' >> myconfig.cache
echo 'hdf5_cv_long_to_ldouble_special=yes' >> myconfig.cache
echo 'hdf5_cv_ldouble_to_llong_accurate=yes' >> myconfig.cache
echo 'hdf5_cv_llong_to_ldouble_correct=yes' >> myconfig.cache
cp  myconfig.cache config.cache
./configure --prefix=/home_directory/FX10 --cache-file=config.cache \
    --target=sparc64-unknown-linux --host=x86 CC="fccpx -Xg"
cd src
make H5make_libsettings H5detect
pjsub --interact
./H5make_libsettings > H5lib_settings.c
./H5detect > H5Tinit.c
exit
cd ..
make
make install
```

configure 時のコンパイラ指定は，「`CC=fccpx CFLAGS=-Xg`」だとダメなので注意．

A.1.3 京へのインストール

FX10 の場合とほぼ同様であるが，「会話ジョブ」が使えない場合がある．その場合には，「`make H5make_libsettings H5detect`」後に次のいずれかの方法を取ること．

方法 1 通常ジョブの中で H5make_libsettings と H5detect を実行して，H5lib_settings.c と H5Tinit.c を作成する．この時，H5make_libsettings の実行には，ファイル libhdf5.settings も必要となるので注意．

方法 2 FX10 で作成した H5lib_settings.c と H5Tinit.c をコピーして用いる．

A.2 Silo のインストール

A.2.1 Mac OS X へのインストール

HDF5 と同様に，Silo ライブラリも/home_direcotory/opt ディレクトリ以下にインストールするものとする．

```
./configure --prefix=/home_directory/opt \
     --with-hdf5=/home_directory/opt/include,/home_directory/opt/lib
make
make install
```

A.2.2 FX10 へのインストール

HDF5 と同様に，Silo ライブラリも/home_direcotory/FX10 ディレクトリ以下にインストールするものとする．

```
tar zxvf silo-4.8-bsd.tar.gz
cd silo-4.8-bsd
echo 'ac_cv_type_off64=yes' > myconfig.cache
echo 'ac_cv_sizeof_off64_t=8' >>  myconfig.cache
cp myconfig.cache config.cache
CC="fccpx -Xg" ./configure --cache-file=config.cache --prefix=/home_directory/FX10 \
```

```
    --target=sparc-linux --host=x86 --disable-fortran --disable-browser \
    --with-hdf5=/home_directory/FX10/include,/home_directory/FX10/lib
cd src/pdb
make detect
pjsub --interact
make pdform.h
exit
cd ../..
make
make install
```

A.2.3 京へのインストール

FX10 の場合とほぼ同様であるが，やはり「会話ジョブ」が使えない場合があるので，「`make detect`」の後に，通常ジョブとして「`make detect`」を行うか，FX10 で作成した pdform.h をコピーすることにより対処する．

付録 B　京での C++ プログラミング

ここでは，京でのコンパイラ最適化を効かすために必要な，C++ プログラミング上の注意点について記す．以下の内容は，2012 年 4 月現在の状況を元にしている．

B.1　テンプレートとコンパイル時ソース出力

テンプレートを使用したコードは，一般にはヘッダファイルに記述するため (図 21)，コンパイル時に-Nsrc オプションを指定しても，最適化の結果をソースリスト上で確認できない．このような場合には，使用予定のある型について明示的な実体化を用いることにより，実装ファイル側にテンプレート定義を記述することにより (図 22)，最適化結果のソース出力が可能となる．

```
// Array3D.h

template <typename T>
class Array3D {
  int n;
  T* data;

public:
  Array3D(int n)
   : n(n), T(new T[n*n*n]) {}

  ~Array3D() { delete[] data; }

  // このメソッドの最適化のかかり具合が知りたい
  void calc() {
    for (int i = 0; i < n; i++) {
      for (int j = 0; j < n; j++) {
        for (int k = 0; k < n; k++) {
          data[i+n*j+n*n*k] = ···
        }
      }
    }
  }

  ···
};
```

図 21　ヘッダファイル中のテンプレート (最適化結果のソースコード出力はなし)

```
// Array3D.h

template <typename T>
class Array3D {
  int n;
  T* data;

public:
  Array3D(int n)
   : n(n), T(new T[n*n*n]) {}

  ~Array3D() { delete[] data; }

  // このメソッドの最適化具合が知りたい
  void calc();

  …
};
```

```
// Array3D.cpp

#include "Array3D.h"

template <typename T>
void Array3D<T>::calc()
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        data[i+n*j+n*n*k] = …
      }
    }
  }
}

// 明示的な実体化
// (これがないとコンパイルできない)
template class Array3D<double>;
```

図 22　明示的な実体化による実装ファイルでのテンプレート定義

B.2　クラスメンバ変数の排除

図 22 のコード例では，calc メソッド内の変数 n はクラス Array3D のメンバ変数である．現在の京では，このようにループ制御部分にクラスのメンバ変数があると，SIMD 化が効かない傾向がある．そのため，プライベートメソッドによりラップする必要がある (図 23).

```
// Array3D.h

template <typename T>
class Array3D {
  int n;
  T* data;

public:
  Array3D(int n)
   : n(n), T(new T[n*n*n]) {}

  ~Array3D() { delete[] data; }

  void calc(){
    calc0(data, n);
  }

  …

private:
  void calc0(T* data, int n);
};
```

```
// Array3D.cpp

#include "Array3D.h"

template <typename T>
void Array3D<T>::calc0(T* data, int n)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        data[i+n*j+n*n*k] = …
      }
    }
  }
}

// 明示的な実体化
// (これがないとコンパイルできない)
template class Array3D<double>;
```

図 23　プライベートメソッドによるラッピング

B.3　「参照渡し」より「実体渡し」

前節のコード例を改良して，3 次元インデックス計算をインデックスファンクタ Index3D(図 24) を用いて行うことを考える．

```
// Index3D.h

class Index3D {
  int n;
public:
  Index3D(int n) : n(n) {}
  int operator() (int i, int j, int k) const {
    return i + n*j + n*n*k;
  }
};
```

図 24　3 次元インデックスファンクタ Index3D

```
// Array3D.h

#include "Index3D.h"

template <typename T>
class Array3D {
  int n;
  T* data;
  Index3D index;

public:
  Array3D(int n)
   : n(n), T(new T[n*n*n]), index(n) {}

  ~Array3D() { delete[] data; }

  void calc(){
    calc0(data, n);
  }

  …

private:
  void calc0(T* data,
             const Index3D& index, int n);
};
```

```
// Array3D.cpp

#include "Array3D.h"

template <typename T>
void Array3D<T>::calc0(T* data,
             const Index3D& index, int n)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        data[index(i,j,k)] = …
      }
    }
  }
}

// 明示的な実体化
// (これがないとコンパイルできない)
template class Array3D<double>;
```

図 25　インデックスファンクタの参照渡し

図 25 のコードでは，インデックスの計算自体はインライン展開されるが，最適化のかかり具合は，図 23 のコードに比べて悪い (3 倍程度遅くなることがある).

一方，オブジェクトのコピーのコストがかかるが，図 26 のコードのように，インデックスファンクタを実体渡しすると，図 23 のコードとほぼ同程度の最適化がかかる (それでも 2〜3% 遅い).

```
// Array3D.h

#include "Index3D.h"

template <typename T>
class Array3D {
  int n;
  T* data;
  Index3D index;

public:
  Array3D(int n)
   : n(n), T(new T[n*n*n]),
     index(n) {}

  ~Array3D() { delete[] data; }

  void calc(){
    calc0(data, n);
  }

  …

private:
  void calc0(T* data,
             Index3D index, int n);
};
```

```
// Array3D.cpp

#include "Array3D.h"

template <typename T>
void Array3D<T>::calc0(T* data,
                       Index3D index, int n)
{
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      for (int k = 0; k < n; k++) {
        data[index(i,j,k)] = …
      }
    }
  }
}

// 明示的な実体化
// (これがないとコンパイルできない)
template class Array3D<double>;
```

図 26　インデックスファンクタの実体渡し

B.4　やっぱり Fortran

それでもまだ遅い場合は，ホットスポットとなるメソッドを Fortran で記述する．