BCM データ管理フレームワーク「BCM Tools(仮称)」ガイド

2012 年 6 月 7 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，その一部として取り入れられる可能性がある．

1.1 用語

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

1.2 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[*1]
- 領域分割処理 → すでに分割済みであること
- BCM 構造 (ツリー構造) 生成 → すでに生成済みであること

[*1] 簡単な可視化データ出力クラス (5 章) をオマケにつけました．

2　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．

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

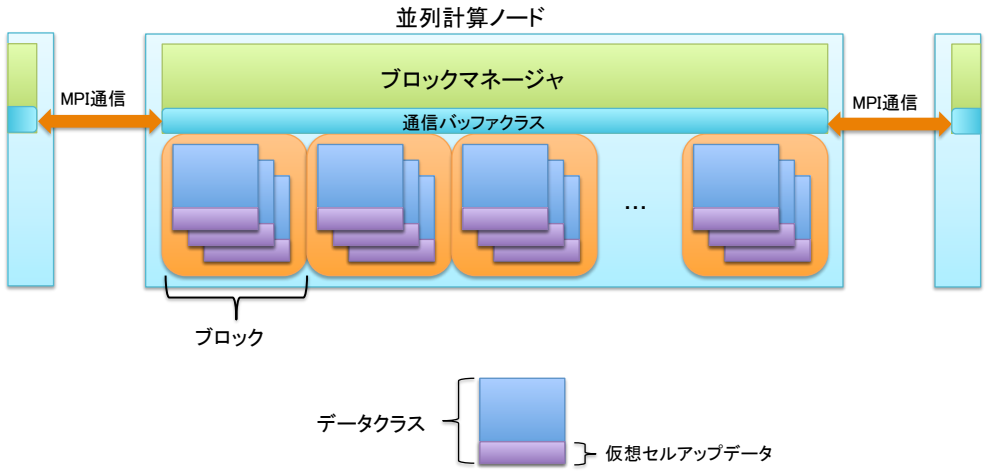

図 1　フレームワーク構成

2.1　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，最低限の基本機能のみ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．

2.2 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．格子空間内のセルに対応した配列中の各値は，仮想セル幅を考慮した 3 次元添字によりアクセスできる．また，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のカスタマイズが可能となる．

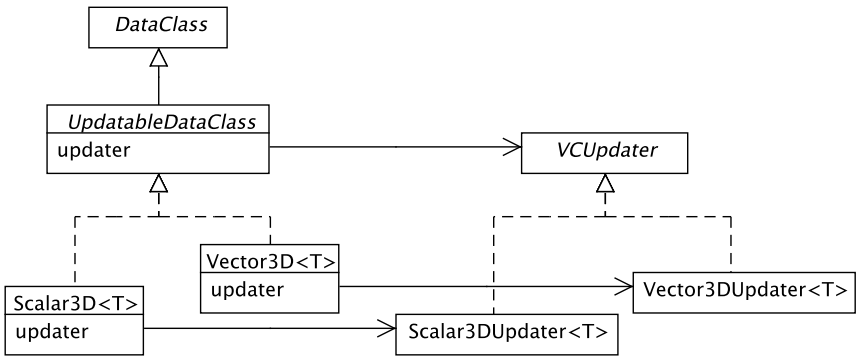

図 2　データクラスと仮想セルアップデータ

2.3 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う．

- 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．

2.4 仮想セルアップデータ (VCUpdater)

仮想セル同期計算，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[*2]を提供する (図 2)．

2.5 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．

[*2] Vector3DUpdater クラスは，まだ未実装．

3　フレームワークを使ってソルバを書く

ここでは，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

3.1　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[*3]．

■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

各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[*4]．

- その面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ランク
    - 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ランク

[*3] 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．

[*4] 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，サブフェイスごとに:
    * 隣接ブロックの所属ランク
    * 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，全ブロックで，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
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う．

```
// ブロックマネージャ
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, ...);
}
```

3.2 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．

3.2.1 BlockBase クラス

■ヘッダファイル BlockBase.h

■private 変数

```
Vec3i size;          // ブロック分割数（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．

3.2.2 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，各ブロック面に対応した 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．

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．

■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．

3.2.3 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ブロッククラスに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 {
```

```
  // 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; }

};
```

3.3 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．

3.3.1 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) のチェックを行う．

■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;
```

■総ブロック数の取得

```
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．BlockBase クラスの提供する機能ではなく，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));
```

■ブロックのセル分割数の取得

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

■全ブロックに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同期開始
void beginUpdateVC(int dataClassID);

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

3.4 Scalar3D，Vector3D データクラスの使い方

3.4.1 共通するメソッド

■セル数，仮想セル幅の取得

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

3.4.2 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;
```

3.4.3 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;
```

3.4.4 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．内部セル領域の添字は，0 から始める．3 次元添字 (i,j,k) により指定されたセルにおけるデータにアクセスする方法は，次の 3 つを用意してある．

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 クラスでの 3 次元データ配置も Scalar3D クラスと同様であるが，ベクトルの 3 成分はメモリ上に連続配置されるように置かれる．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 3　Scalar3D, Vector3D クラスのおけるセルデータへのアクセス方法

現在の「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いている．

[*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．

4 フレームワークをチューニング・カスタマイズする

以下では，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．

4.1 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．

通信対象となるデータは，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を行い，アンパック時には指定され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．

4.1.1 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，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，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のポインタを登録では，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 PointerSetter クラスでラップしたものをわたす．

■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こと．

4.1.2 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 4)．

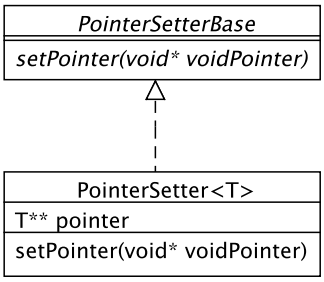

図 4　PointerSetter クラス

■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．

4.1.3 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
```

4.1.4 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
```

4.2 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．

4.2.1 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 [*6]

```
// 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 の場合の 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
```

■送信用バッファへのコピー

[*6] 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．

```
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
```

5　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用意してある．出力ファイルは Silo フォーマットを採用して，次の機能を持つ．

- 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．
- ブロック単位での領域分割の状態を出力できる．
- 出力は，Silo の Multi-Block 出力機能を利用して，ノード単位の分散出力となる．
- 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．

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．

5.1　SiloWrite 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．

■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，ノード 0 でマスターファイル xxx.silo を出力し，一般のノード (ノード 0 も含む) では分散出力データファイル xxx_n.silo(n はノード番号) 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ノード番号をデータ名 `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，データクラス IDdataClassID の Scalar3D<T>型データを，データ名 name で出力する．型 T は，float または double にのみ対応．

■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，データクラス IDdataClassID の Vector3D<T>型データを，データ名 name で出力する．型 T は，float または double にのみ対応．

6 インストール方法

6.1 ディレクトリ構成

```
Makefile.inc       ... make パラメータファイル
lib/               ... ライブラリ置場
include/           ... インクルードファイル
src/               ... ソースコード
  Makefile
  DataClass.list   ... DataClass リストファイル
  Block/
  CommBuffer/
  DataClass/
example/           ... サンプルコード
  sor/
  sor_vis/
  vis/
doc/
  guide.pdf        ... このドキュメント
  doxygen/
    hmtl/          ... Doxygen 出力
```

6.2 コンパイル方法

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 が作成される．

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 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．

6.3 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．

7　サンプルプログラム

7.1　sor (Poisson 方程式を SOR で解く)

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 5) と simple(図 6) から選択できる．flat ブロック配置では，外部境界に接するブロックのみ再分割するという規則を再帰的に繰り返すことにより，ツリーを構成している．並列計算ノードへの領域分割は，いずれのブロック配置も，全ブロックに対して Z-ordering によりグローバル ID をふり，それをノード数により等分割することにより行なっている．

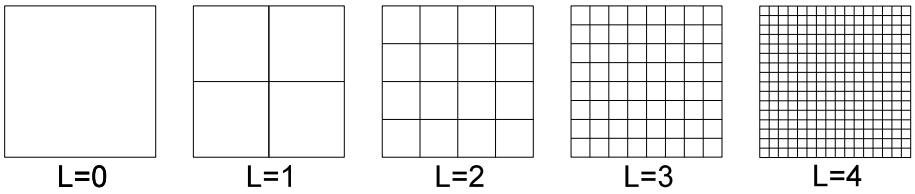

図 5　ブロック配置 flat (level=0〜4)

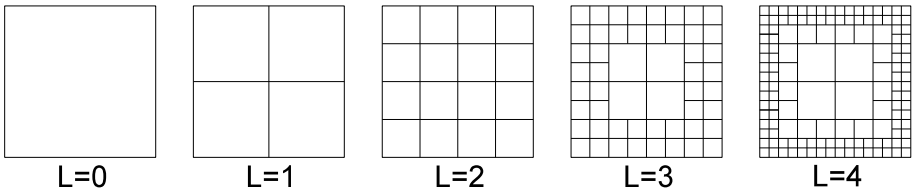

図 6　ブロック配置 simple (level=0〜4)

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 7)．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` 回繰りかえすことになる．

```
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 7　SOR 計算における反復制御

サンプルプログラムの実行においては，設定ファイルをその引数で指定する．設定ファイルには，「キーワード = 値」の形式で，以下のパラメータを指定する．

`level`　ブロック配置のツリーレベル (0 以上の整数)
`treeType`　ブロック配置タイプ (`flat` または `simple`)
`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)
`randomShuffle`　`true` または `false`(ディフォルト)．`true` の場合，ブロック ID をランダムにシャッフルしてから等分割し，各ノードに配分する．
`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

7.2　sor_vis (Poisson 方程式を SOR で解き，可視化用データを出力)

sor と同様な計算領域，ブロック配置に対して，適当なソース項により以下の厳密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．

設定ファイルで指定するパラメータは以下のとおり．

`level`　ブロック配置のツリーレベル (0 以上の整数)
`treeType`　ブロック配置タイプ (`flat` または `simple`)
`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)
`randomShuffle`　`true` または `false`(ディフォルト)．`true` の場合，ブロック ID をランダムにシャッ

フルしてから等分割し，各ノードに配分する．

`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．

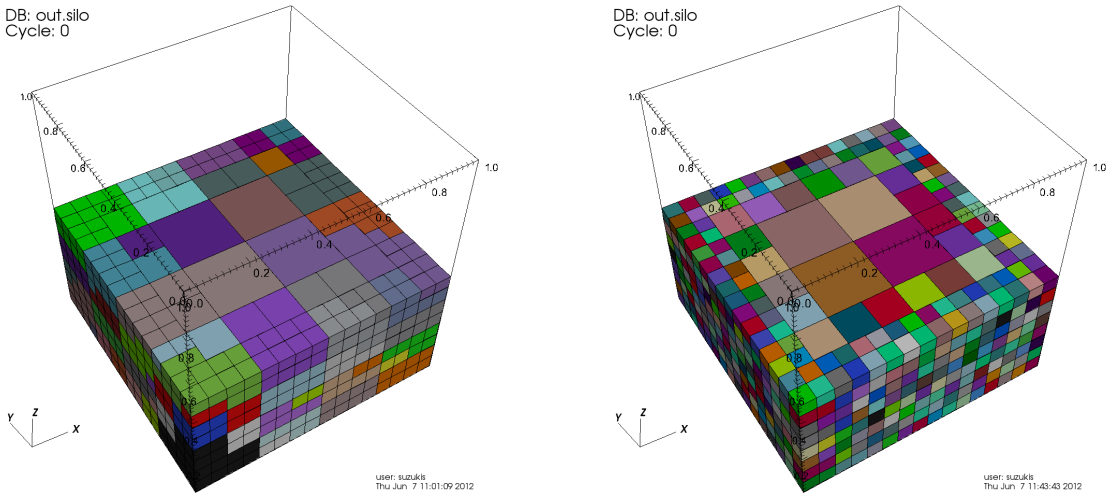

図 8　領域分割の様子．左図はランダムシャッフルなし，右図はランダムシャッフルあり．block_mesh を”Mesh” 表示，domain を”Pseudocolor” 表示し，$z = 0.5$ でクリッピングしたもの．

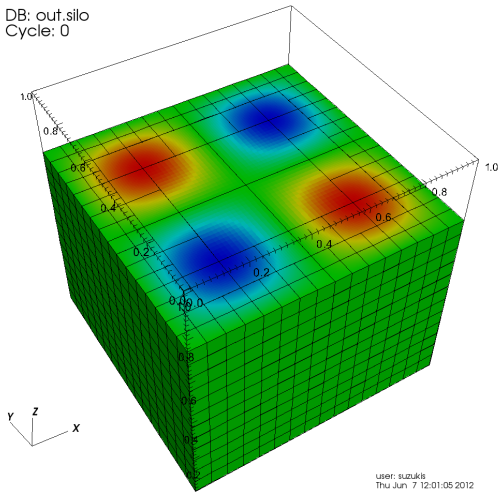

図 9　計算結果．block_mesh を”Mesh” 表示，result を”Pseudocolor” 表示し，$z = 0.75$ でクリッピング．

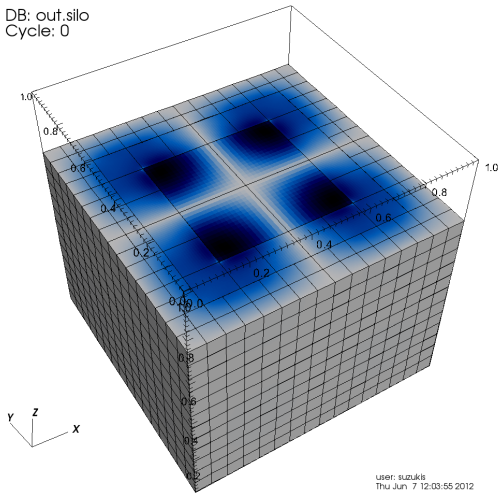

図 10　計算誤差．block_mesh を”Mesh” 表示，error を”Pseudocolor” 表示し，$z = 0.75$ でクリッピング．色の濃さ＝絶対誤差．

7.3　vis (可視化用データ出力のテストプログラム)

単位立方体領域 $(0 \leq x, y, z, \leq 1)$ に対して，可視化テスト用に，スカラーデータとして原点からの距離，ベクトルデータとして位置ベクトルを出力する．

設定ファイルで指定するパラメータは以下のとおり．

`level`　ブロック配置のツリーレベル (0 以上の整数)
`treeType`　ブロック配置タイプ (`flat` または `simple`)
`size`　ブロック内のセル分割数 (偶数であること)
`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　ブロック配置 simple, ツリーレベル 4，ブロック内セル分割数 2．

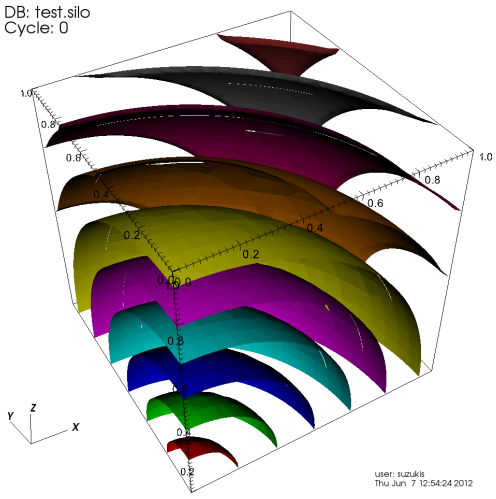

図 11　scalar を”Contour” 表示.

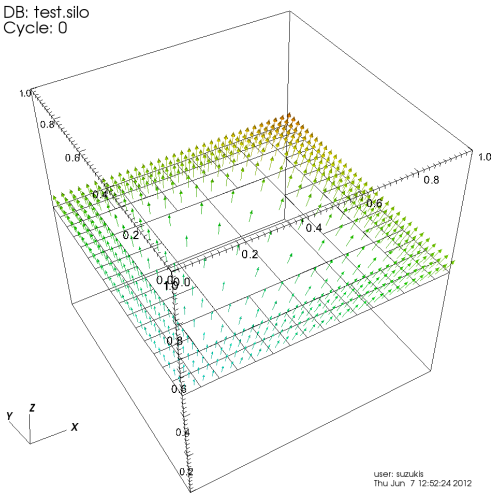

図 12　vecotr を”Vector” 表示し，$z = 0.5$ でスライス.

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[*7]の FX10 および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[*8]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　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
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
```

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

```
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.1.2 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 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.2.2 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 13)，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 14)，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 13　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 14　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 14 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 15).

```
// 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 15　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 16) 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 16　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 17　インデックスファンクタの参照渡し

図 17 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 15 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 18 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 15 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 18　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．