User Guide of PFClib

Parallel File Compression Library

Ver. 1.0.0

Advanced Institute for Computational Science
RIKEN

http://www.aics.riken.jp/

Febrary 2014

Version 1.0.0 19 Feb. 2014

7-1-26, Minatojima-minami-machi, Chuo-ku, Kobe, 650-0047, JAPAN.

目次

表目次

図目次

第 1 章

PFClib の概要

PFClib の概要と本ユーザガイドについて説明します．

1.1　PFClib

PFClib(Parallel File Library) は，大規模並列シミュレーションの結果ファイルのサイズを小さくするデータ圧縮クラスライブラリを行う C++ クラスライブラリです．ユーザーは，C++ で本ライブラリを利用できます．

PFClib は，以下の機能を有します．

- POD(Proper Orthogonal Decomposition) を用いたデータの圧縮および展開

1.2　この文書について

1.2.1　書式について

次の書式で表されるものは，Shell のコマンドです．

```
$ コマンド（コマンド引数）
```

または，

```
# コマンド（コマンド引数）
```

“$” で始まるコマンドは一般ユーザーで実行するコマンドを表し，“#” で始まるコマンドは管理者（主に root）で実行するコマンドを表しています．

1.2.2　動作環境

PFC ライブラリは，以下の環境について動作を確認しています．

- Linux/gnu コンパイラ
- Linux/Intel コンパイラ
- 京コンピュータ

第 2 章

パッケージのビルド

この章では，PFClib のコンパイルについて説明します.

2.1　パッケージのビルド

2.1.1　パッケージの構造

PFC ライブラリのパッケージは次のようなファイル名で保存されています.
(*X.X.X* にはバージョンが入ります)
`PFClib-`*X.X.X*`.tar.zip`

このファイルの内部には，次のようなディレクトリ構造が格納されています.

PFClib-*X.X.X*/
- AUTHORS
- COPYING
- ChangeLog
- INSTALL
- LICENSE
- Makefile.am
- Makefile.in
- NEWS
- README
- aclocal.m4
- compile
- config.h.in
- configure
- configure.ac
- depcomp
- install-sh
- missing
- doc/
  - PFClib_UserGuide.pdf
  - Reference.pdf
- doxygen/
  - Doxyfile
  - makepdf.sh
- example/
  - compress/
  - compress_cmd/
  - restration_read_at_index/
  - restration_read_at_index_on_the_fly/
  - restration_read_in_range/
- include/
- src/
- utility/
  - staging/
    - include/
    - src/

これらのディレクトリ構造は，次の様になっています．

- doc
  この文書を含む PFClib ライブラリの文書が収められています．

- doxygen
  ドキュメントの元となったファイルが収められています．

- include
  ヘッダファイルが収められています．ここに収められたファイルは make install で$prefix/include にインストールされます．

- src
  ソースが格納されたディレクトリです．ここにライブラリ libPFC.a が作成され，make install で $prefix/lib にインストールされます．

- utility
  ステージングを行うユーティリティが収められています．

2.1.2　パッケージのビルド

いずれの環境でも shell で作業するものとします．以下の例では bash を用いていますが，shell によって環境変数の設定方法が異なるだけで，インストールの他のコマンドは同一です．適宜，環境変数の設定箇所をお使いの環境でのものに読み替えてください．

以下の例では，作業ディレクトリを作成し，その作業ディレクトリに展開したパッケージを用いてビルド，インストールする例を示しています．

1. 作業ディレクトリの構築とパッケージのコピー
   まず，作業用のディレクトリを用意し，パッケージをコピーします．ここでは，カレントディレクトリに work というディレクトリを作り，そのディレクトリにパッケージをコピーします．

   ```
   $ mkdir work
   $ cp [パッケージのパス] work
   ```

2. 作業ディレクトリへの移動とパッケージの解凍
   先ほど作成した作業ディレクトリに移動し，パッケージを解凍します．

   ```
   $ cd work
   $ unzip zxvf PFClib-X.X.X.zip
   ```

3. PFClib-X.X.X ディレクトリに移動
   先ほどの解凍で作成された PFClib-X.X.X ディレクトリに移動します．

   ```
   $ cd PFClib-X.X.X
   ```

4. configure スクリプトを実行
   次のコマンドで configure スクリプトを実行します．

   ```
   $ ./configure [option]
   ```

configure スクリプトの実行時には，お使いの環境に合わせたオプションを指定する必要があります．configure オプションに関しては，2.1.3 章を参照してください．configure スクリプトを実行することで，環境に合わせた Makefile が作成されます．

5. make の実行
   make コマンドを実行し，ライブラリをビルドします．

   ```
   $ make
   ```

   make コマンドを実行すると，次のファイルが作成されます．

   ```
   src/libPFC.a
   ```

   ビルドをやり直す場合は，`make clean` を実行して，前回の make 実行時に作成されたファイルを削除します．

   ```
   $ make clean
   $ make
   ```

   また，configure スクリプトによる設定，Makefile の生成をやり直すには，`make distclean` を実行して，全ての情報を削除してから，configure スクリプトの実行からやり直します．

   ```
   $ make distclean
   $ ./configure [option]
   $ make
   ```

6. インストール
   次のコマンドで configure スクリプトの`--prefix` オプションで指定されたディレクトリに，ライブラリ，ヘッダファイルをインストールします．

   ```
   $ make install
   ```

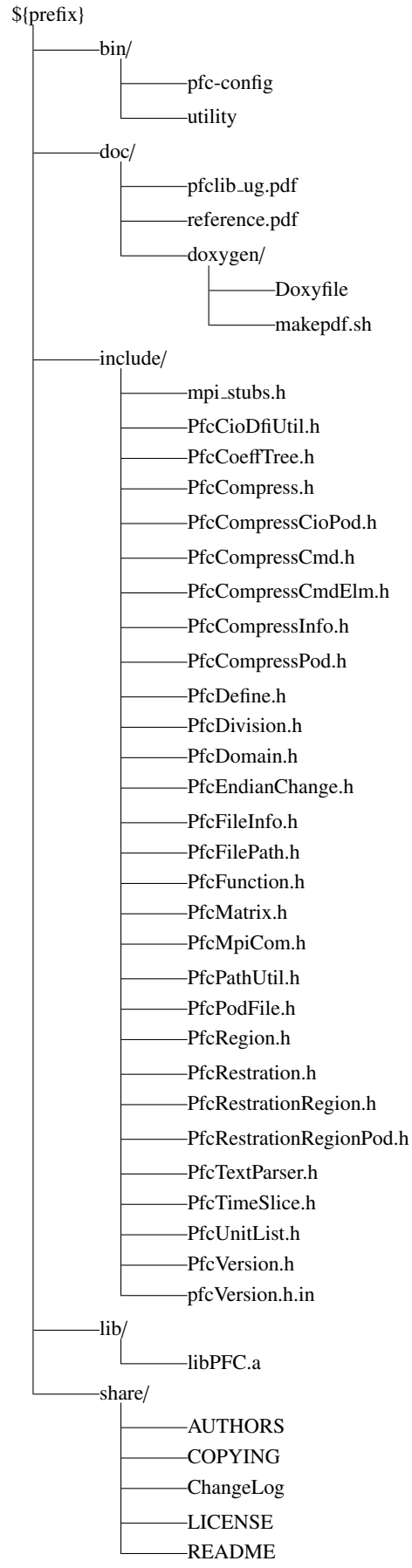
   ただし，インストール先のディレクトリへの書き込みに管理者権限が必要な場合は，sudo コマンドを用いるか，管理者でログインして `make install` を実行します．

   ```
   $ sudo make install
   ```

   または，

   ```
   $ su
   password:
   # make install
   # exit
   ```

   インストールされる場所とファイルは以下の通りです．



7. アンインストール

   アンインストールするには，書き込み権限によって，

```
$ make uninstall
```

または,

```
$ sudo make uninstall
```

または,

```
$ su
passward:
# make uninstall
# exit
```

を実行します.

2.1.3　configure スクリプトのオプション

- --host=*hostname*
  host は，クロスコンパイルの場合に指定します.

- --prefix=*dir*
  prefix は, パッケージをどこにインストールするかを指定します．prefix で設定した場所が--prefix=/usr/local/PFClib の時,

  ライブラリ：/usr/local/PFClib/lib
  ヘッダファイル：/usr/local/PFClib/include

  にインストールされます.
  prefix オプションが省略された場合は，デフォルト値として/usr/local/PFClib が採用され，インストールされます.

- --with-example=*yes*|*no*
  with-example は，サンプルプログラムをインストールする場合に'yes' を指定します.

- --with-ompi=*dir*
  with-ompi は，OpenMPI ライブラリへのパスを指定します.

- --with-parser=*dir*
  with-parser は，TextParser ライブラリへのパスを指定します.

- --with-cio=*dir*
  with-cio は，CIO ライブラリへのパスを指定します.

- コンパイラ等のオプション
  コンパイラ，リンカやそれらのオプションは，configure スクリプトで半自動的に探索します．ただし，標準ではないコマンドやオプション，ライブラリ，ヘッダファイルの場所は探索出来ないことがあります．また，標準でインストールされたものでないコマンドやライブラリを指定して利用したい場合があります．そのような場合，これらの指定を configure スクリプトのオプションとして指定することができます.

CXX
C++ コンパイラのコマンドパスです．

CXXFLAGS
C++ コンパイラへ渡すコンパイルオプションです．

LDFLAGS
リンク時にリンカに渡すリンク時オプションです．例えば，使用するライブラリが標準でないの場所 <libdir> にある場合，-L<libdir> としてその場所を指定します．

LIBS
利用したいライブラリをリンカに渡すリンク時オプションです．例えば，ライブラリ <library> を利用する場合，-l<library> として指定します．

・ ライブラリ指定のオプション
PFC ライブラリを利用する場合，コンパイル，リンク時に，MPI ライブラリと TextParser ライブラリと CIOlib ライブラリが必ず必要になります．これらのライブラリのインストールパスは，次に示す configure オプションで指定する必要があります．

--with-ompi=*dir*
MPI ライブラリとして OpenMPI を使用する場合に，OpenMPI のインストール先を指定します．

--with-parser=*dir*
TextParser ライブラリのインストール先を指定します．

--with-cio=*dir*
CIOlib ライブラリのインストール先を指定します．

なお、mpic++ 等の mpi ライブラリに付属のコンパイララッパーを使用する場合は，mpi に関する設定がラッパー内で自動的に設定されるため，--with-ompi の指定は必要ありません．

なお，configure オプションの詳細は，./configure --help コマンドで表示されますが，PFC ライブラリでは，上記で説明したオプション以外は無効となります．

2.1.4　configure 実行時オプションの例

・ Linux/gnu コンパイラ

PFC ライブラリの prefix：/home/userXXXX/PFClib
MPI ライブラリ：OpenMPI，/usr/local/openmpi
TextParser ライブラリ：/home/userXXXX/textparser
CIOlib ライブラリ：/home/userXXXX/CIOlib
C++ コンパイラ：g++

の環境の場合，次のように configure コマンドを実行します．

```
$ ./configure --prefix=/home/userXXXX/PFClib \
              --with-ompi=/usr/local/openmpi \
              --with-parser=/home/userXXXX/textparser \
              --with-cio=/home/userXXXX/CIOlib
              CXX=g++
```

・京コンピュータの場合

PFC **ライブラリの** prefix：/home/userXXXX/PFClib
TextParser **ライブラリ**：/home/userXXXX/textparser
CIOlib **ライブラリ**：/home/usreXXXX/CIOlib
C++ **コンパイラ**：mpiFCCpx

の環境の場合，次のように configure コマンドを実行します．

```
$ ./configure --host=sparc64-unknown-linux-gnu \
              --prefix=/home/userXXXX/PFClib \
              --with-parser=/home/userXXXX/textparser \
              --with-cio=/home/usreXXXX/CIOlib \
              CXX=mpiFCCpx
```

2.1.5　pfc-config コマンド

PFC ライブラリをインストールすると，$prefix/bin/pfc-config コマンド（シェルスクリプト）が生成されます．
このコマンドを利用することで，ユーザーが作成したプログラムをコンパイル，リンクする際に，PFC ライブラリを参照するために必要なコンパイルオプション，リンク時オプションを取得することができます．

pfc-config コマンドは，次に示すオプションを指定して実行します．

--cxx
PFC ライブラリの構築時に使用した C++ コンパイラを取得します．

--cflags
C++ コンパイラオプションを取得します．

--libs
PFC ライブラリのリンクに必要なリンク時オプションを取得します．

ただし，pfc-config コマンドで取得できるオプションは，PFC ライブラリを利用する上で最低限必要なオプションのみとなります．
最適化オプション等は必要に応じて指定してください．

また，具体的な pfc-config コマンドの使用方法は，2.2 章を参照してください．

2.1.6　フロントエンドでステージングツールを使用する場合のビルド方法

京コンピュータ等のクロスコンパイル環境でステージングツールを使用する場合，フロントエンド用のネイティブコンパイラを用いて CIO ライブラリをビルドする必要があります．
また，フロントエンドに MPI ライブラリがインストールされていない場合，CIO ライブラリの configure スクリプト実行時に MPI ライブラリを未実行とするオプション「–without-MPI」を付けてビルドする必要があります．

・京コンピュータフロントエンド用の configure 実行例

```
$ ./configure --prefix=/home/userXXXX/PFClib_frontend \
              --without-MPI \
              --with-parser=/home/userXXXX/textparser \
              --with-cio=/home/userXXXX/CIOlib \
              CXX=g++
```

なお，この場合リンクする textparser もフロントエンドのネイティブコンパイラでビルドしておく必要があります．

2.2　PFC ライブラリの利用方法

PFC ライブラリは，C++ プログラム内で利用できます．以下に，ユーザーが作成する PFC ライブラリを利用するプログラムのビルド方法を示します．

以下の例では，configure スクリプトで”--prefix=/usr/local/PFClib” を指定して PFC ライブラリをビルド，インストールしているものとして示します．

```
$ ./configure --prefix=/usr/local/PFClib \
              --with-ompi=/usr/local/openmpi \
              --with-parser=/usr/local/textparser \
              --with-cpm=/usr/local/CIOlib \
              CXX=icpc
```

第 3 章

API 利用方法

この章では，PFClib の API の利用方法について説明します．

3.1　ユーザープログラムでの利用方法

以下に，PFC ライブラリの C++ API の説明を示します．

3.1.1　ヘッダーファイルのインクルード

PFC ライブラリの C++ API 関数群は, PFC ライブラリが提供するヘッダファイル PfcCompress.h PfcCompressCmd.h PfcRestration.h で定義されています．PFC ライブラリの API 関数を使う場合は，これらヘッダーファイルをインクルードします．

PfcCompress.h には，圧縮に必要な CPfcCompress クラスのインターフェイスが記述されています．PfcCompress-Cmd.h には，圧縮の制御に必要な CPfcCompressCmd クラスのインターフェイスが記述されています．PfcRestration.h には，展開に必要な CPfcRestration クラスのインターフェイスが記述されています．ユーザープログラムから本ライブラリを使用する場合，これらのクラスのメソッドを用います．

Pfc*.h は，configure スクリプト実行時の設定 `prefix` 配下の`${prefix}`/include に `make install` 時にインストールされます．

3.1.2　マクロ，列挙型，エラーコード

PFC ライブラリ内で使用されるマクロ，列挙型，エラーコードについては，PfcDefine.h に定義されています．

・ D_PFC マクロ
D_PFC マクロは，pfcDefine.h で表 3.1 のように定義されています．

表 3.1　D_PFC マクロ

マクロ名	内容
D_PFC_FLOAT32	”Float32”
D_PFC_FLOAT64	”Float64”
D_PFC_IJNK	”ijkn”
D_PFC_NIJK	”nijk”
D_PFC_LITTLE	”little”
D_PFC_BIG	”big”
D_PFC_EPSILON	”(1.0e-9)”
D_PFC_COMPRESS_ERROR_DEFAULT	”(0.01)”

・ E_PFC_COMPRESS_FORMAT 列挙型
E_PFC_COMPRESS_FORMAT 列挙型は，pfcDefine.h で表 3.2 のように定義されています．
圧縮形式を指定するフラグとして使われます．

・ E_PFC_DTYPE 列挙型
E_PFC_DTYPE 列挙型は，pfc_Define.h で表 3.3 のように定義されています．
フィールドデータのデータ形式を指定するフラグとして使われます．

表 3.2　E_PFC_COMPRESS_FORMAT 列挙型

E_PFC_COMPRESS_FORMAT 要素	値	意味
E_PFC_COMPRESS_FMT_UNKNOWN	-1	未定
E_PFC_COMPRESS_FMT_POD	1	POD format

表 3.3　E_PFC_DTYPE 列挙型

E_PFC_DTYPE 要素	値	意味
E_PFC_DTYPE_UNKNOWN	0	未定義
E_PFC_INT8	1	char
E_PFC_INT16	2	short
E_PFC_INT32	3	int
E_PFC_INT64	4	long long
E_PFC_UINT8	5	unsigned char
E_PFC_UINT16	6	unsigned short
E_PFC_UINT32	7	unsigned int
E_PFC_UINT64	8	unsigned long long
E_PFC_FLOAT32	9	float
E_PFC_FLOAT64	10	double

・ E_PFC_ARRAYSHAPE 列挙型

E_PFC_ARRAYSHAPE 列挙型は，pfcDefine.h で表 3.4 のように定義されています.
フィールドデータの配列形式を指定するフラグとして使われます.

表 3.4　E_PFC_ARRAYSHAPE 列挙型

E_PFC_ARRAYSHAPE 要素	値	意味
E_PFC_ARRAYSHAPE_UNKNOWN	-1	未定義
E_PFC_IJKN	0	(i,j,k,n)
E_PFC_NIJK	1	(n,i,j,k)

・ E_PFC_ENDIANTYPE 列挙型

E_PFC_ENDIANTYPE 列挙型は，pfc_Define.h で表 3.5 のように定義されています.
フィールドデータのエンディアン形式を指定するフラグとして使われます.

表 3.5　E_PFC_ENDIANTYPE 列挙型

E_PFC_ENDIANTYPE 要素	値	意味
E_PFC_ENDIANTYPE_UNKNOWN	-1	未定義
E_PFC_LITTELE	0	リトルエンディアン形式
E_PFC_BIG	1	ビッグエンディアン形式

・E_PFC_ERRORCODE 列挙型

E_PFC_ERRORCODE 列挙型は，pfcDefine.h で表 3.6，3.7 のように定義されています．

PFC ライブラリの API 関数のエラーコードは，全てこの列挙型で定義されています．

表 3.6　E_PFC_ERRORCODE 列挙型 その 1

E_PFC_ERRORCODE 要素	値	意味
E_PFC_SUCCESS	1	正常終了
E_PFC_ERROR	-1	エラー終了
E_PFC_ERROR_READ_CNTLFILE_OPENERROR	500	制御ファイルオープンエラー
E_PFC_ERROR_READ_CNTL_DOMAINDIVISION	501	PfcCompressCntl/DomainDivision 読込エラー
E_PFC_ERROR_READ_CNTL_NO_ITEM	510	制御ファイル未定義
E_PFC_ERROR_READ_CNTL_ITEMCNTL	511	PfcCompressCntl/ItemCntl 読込エラー
E_PFC_ERROR_READ_CNTL_DFI_PATH	512	PfcCompressCntl/ItemCntl/InputDfiPath 読込エラー
E_PFC_ERROR_READ_CNTL_OUTDIR_PATH	513	PfcCompressCntl/ItemCntl/OutputDirectoryPath 読込エラー
E_PFC_ERROR_READ_CNTL_COMPRESS_FMT	514	PfcCompressCntl/ItemCntl/CompressFormat 読込エラー
E_PFC_ERROR_READ_CNTL_PROCFILE_SAVE	515	PfcCompressCntl/ItemCntl/ProcFileSave 読込エラー
E_PFC_ERROR_READ_INDEXFILE_OPENERROR	1000	Index ファイルオープンエラー
E_PFC_ERROR_READ_FILEINFO	1010	FileInfo 読込エラー
E_PFC_ERROR_READ_PFC_DIRECTORYPATH	1011	FileInfo/DirectoryPath 読込エラー
E_PFC_ERROR_READ_PFC_PREFIX	1012	FileInfo/Prefix 読込エラー
E_PFC_ERROR_READ_PFC_FILEFORMAT	1013	FileInfo/FileFormat 読込エラー
E_PFC_ERROR_READ_PFC_GUIDECELL	1014	FileInfo/GuideCell 読込エラー
E_PFC_ERROR_READ_PFC_DATATYPE	1015	FileInfo/DataType 読込エラー
E_PFC_ERROR_READ_PFC_ENDIAN	1016	FileInfo/Endian 読込エラー
E_PFC_ERROR_READ_PFC_ARRAYSHAPE	1017	FileInfo/ArrayShape 読込エラー
E_PFC_ERROR_READ_PFC_COMPONENT	1018	FileInfo/Component 読込エラー
E_PFC_ERROR_READ_COMPRESSINFO	1030	CompressInfo 読込エラー
E_PFC_ERROR_READ_PFC_COMPRESSFORMAT	1031	CompressInfo/CompressFormat 読込エラー
E_PFC_ERROR_READ_PFC_COMPRESSERROR	1032	CompressInfo/CompressError 読込エラー
E_PFC_ERROR_READ_PFC_CALCULATEDLAYER	1033	CompressInfo/CalculatedLayer 読込エラー
E_PFC_ERROR_READ_PFC_VERSION	1034	CompressInfo/Version 読込エラー
E_PFC_ERROR_READ_PFC_STARTSTEP	1035	CompressInfo/StartStep 読込エラー
E_PFC_ERROR_READ_PFC_ENDSTEP	1036	CompressInfo/EndStep 読込エラー
E_PFC_ERROR_READ_FILEPATH	1040	FilePath 読込エラー
E_PFC_ERROR_READ_PFC_DFIPATH	1041	FilePath/DfiPath 読込エラー
E_PFC_ERROR_READ_PFC_PFCPROCESS	1042	FilePath/PfcProcess 読込エラー
E_PFC_ERROR_READ_UNITLIST	1050	UnitList 読込エラー
E_PFC_ERROR_READ_TIMESLICE	1060	TimeSlice 読込エラー
E_PFC_ERROR_WRITE_INDEXFILENAME_EMPTY	1100	インデックスファイル名未定義
E_PFC_ERROR_WRITE_INDEXFILE_OPENERROR	1101	インデックスファイルオープンエラー
E_PFC_ERROR_WRITE_FILEINFO	1110	FileInfo 出力エラー
E_PFC_ERROR_WRITE_PFC_DATATYPE	1111	FileInfo/DataType 出力エラー
E_PFC_ERROR_WRITE_COMPRESSINFO	1130	CompressInfo 出力エラー
E_PFC_ERROR_WRITE_FILEPATH	1140	FilePath 出力エラー
E_PFC_ERROR_WRITE_UNITLIST	1150	UnitList 出力エラー
E_PFC_ERROR_WRITE_TIMESLICE	1160	TimeSlice 出力エラー
E_PFC_ERROR_READ_PROCFILE_OPENERROR	1200	Proc ファイルオープンエラー
E_PFC_ERROR_READ_DOMAIN	1210	Domain 読込エラー
E_PFC_ERROR_READ_PFC_GLOBALORIGIN	1211	Domain/GlobalOrigin 読込エラー
E_PFC_ERROR_READ_PFC_GLOBALREGION	1212	Domain/GlobalRegion 読込エラー

表 3.7　E_PFC_ERRORCODE 列挙型 その 2

cpm_ErrorCode 要素	値	意味
E_PFC_ERROR_READ_PFC_GLOBALVOXEL	1213	Domain/GlobalVoxel 読込エラー
E_PFC_ERROR_READ_PFC_GLOBALDIVISION	1214	Domain/GlobalDivision 読込エラー
E_PFC_ERROR_READ_DIVISION	1220	Division 読込エラー
E_PFC_ERROR_READ_PFC_NO_REGION	1221	Division/Region 読込エラー
E_PFC_ERROR_READ_PFC_REGION_ID	1222	Division/Region/ID 読込エラー
E_PFC_ERROR_READ_PFC_REGION_VOXELSIZE	1223	Division/Region/VoxelSize 読込エラー
E_PFC_ERROR_READ_PFC_REGION_HEADINDEX	1224	Division/Region/HeadIndex 読込エラー
E_PFC_ERROR_READ_PFC_REGION_TAILINDEX	1225	Division/Region/TailIndex 読込エラー
E_PFC_ERROR_WRITE_PROCFILE_OPENERROR	1300	Proc ファイルオープンエラー
E_PFC_ERROR_WRITE_DOMAIN	1310	ProcDomain 出力エラー
E_PFC_ERROR_WRITE_DIVISION	1320	ProcDivision 出力エラー
E_PFC_ERROR_READ_PODBASE_OPENERROR	2000	基底ファイルオープンエラー
E_PFC_ERROR_READ_PODBASE_HEADER	2005	基底ファイルヘッダー読込エラー
E_PFC_ERROR_READ_PODBASE_DATA	2010	基底ファイルデータ読込エラー
E_PFC_ERROR_WRITE_PODBASE_OPENERROR	2100	基底ファイルオープンエラー
E_PFC_ERROR_WRITE_PODBASE_DATA	2110	基底ファイル出力エラー
E_PFC_ERROR_READ_PODCOEF_OPENERROR	2200	係数ファイルオープンエラー
E_PFC_ERROR_READ_PODCOEF_DATA	2210	係数ファイル読込エラー
E_PFC_ERROR_WRITE_PODCOEF_OPENERROR	2300	係数ファイルオープンエラー
E_PFC_ERROR_WRITE_PODCOEF_DATA	2310	係数ファイル出力エラー
E_PFC_ERROR_PFC_COMPRESSFORMAT	3000	圧縮形式出力エラー
E_PFC_ERROR_OUT_OF_RANGE	3100	

3.2　圧縮機能

3.2.1　機能概要

3.2.2　圧縮処理手順

PFC では以下の手順で，フィールドデータの圧縮処理を行います.

3.2.3 圧縮処理のサンプルコード

1. 圧縮処理のサンプルコードを以下に示します.

```
#include "mpi.h"
#include "PfcCompress.h"

int main( int argc, char **argv )
{
  PFC::E_PFC_ERRORCODE ret;


  //MPI Initialize
  if( MPI_Init(&argc,&argv) != MPI_SUCCESS )
  {
      std::cerr << "MPI_Init error." << std::endl;
      return false;
  }
  int numRank;
  int rankID;
  MPI_Comm_size( MPI_COMM_WORLD, &numRank );
  MPI_Comm_rank( MPI_COMM_WORLD, &rankID );

  if( rankID == 0 ) {
    cout << "##### Compression sample start #####"<< endl;
  }

  CPfcCompress  compress;

  std::string dfiFilePath    = "IN_Cio/vel.dfi";
  std::string outDirPath     = "field_data";
  std::string compressFormat = "pod";
  //double      compressError  = 3.0;  // 誤差率 (%)
  double      compressError  = 1.0;  // 誤差率 (%)
  int         domainDivision[3];
  int         startStep      = 0;
  int         endStep        = 30; int numStep=4;
  int         optFlags       = 0;

  //ステップ数よりステップ方向の圧縮並列数を取得
  int numParallel = CPfcFunction::GetPodParallel( numStep );
  int numRegion   = numRank/numParallel;

  if( numRegion == 1 ) {
    domainDivision[0] = 1; domainDivision[1] = 1; domainDivision[2] = 1;
  } else if( numRegion == 2 ) {
    domainDivision[0] = 2; domainDivision[1] = 1; domainDivision[2] = 1;
  } else if( numRegion == 4 ) {
    domainDivision[0] = 2; domainDivision[1] = 2; domainDivision[2] = 1;
  } else if( numRegion == 8 ) {
    domainDivision[0] = 2; domainDivision[1] = 2; domainDivision[2] = 2;
  } else if( numRegion == 16 ) {
    domainDivision[0] = 4; domainDivision[1] = 2; domainDivision[2] = 2;
  } else {
    domainDivision[0] = 2; domainDivision[1] = 2; domainDivision[2] = 2;
  }

  // 初期化
  if( rankID == 0 ) {
    cout << "##### Compression Init()  start #####"<< endl;

    cout << "  numRank     = "<<numRank<< endl;
    cout << "  rankID      = "<<rankID<< endl;
    cout << "  numStep     = "<<numStep<< endl;
    cout << "  numParallel = "<<numParallel<< endl;
    cout << "  numRegion   = "<<numRegion<< endl;
    cout << "  domainDivision[3] = "<<domainDivision[0]<<" "
```

```
        <<domainDivision[1]<<" "<<domainDivision[2]<< endl;
    cout << "  compressError     = "<<compressError<< endl;
  }

  ret = compress.Init(
            MPI_COMM_WORLD,  // MPI コミュニケータ
            dfiFilePath,     // DFI ファイルパス
            outDirPath,      // フィールドデータ出力ディレクトリパス
            compressFormat,  // 圧縮フォーマット
            compressError,   // 誤差率 (%)
            domainDivision,  // 領域の分割数
            startStep,       // 開始ステップ
            endStep,         // 終了ステップ
            optFlags         // オプション flags
          );
  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Compression Init() Error  #####"<< endl;
    return -1;
  }

  // proc.pfc ファイル出力
  if( rankID == 0 ) {
    cout << "##### Compression WriteProcFile()  start #####"<< endl;
  }

  ret = compress.WriteProcFile();
  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Compression WriteProcFile() Error  #####"<< endl;
    return -1;
  }

  // 圧縮＆圧縮データファイル＆ index.pfc 出力処理
  if( rankID == 0 ) {
    cout << "##### Compression compress.WriteData()  start #####"<< endl;
  }

  ret = compress.WriteData();
  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Compression WriteProcFile() Error  #####"<< endl;
    return -1;
  }

  if( rankID == 0 ) {
    cout << "##### Compression sample end #####"<< endl;
  }

  MPI_Finalize();

  return 0;
}
```

[**実行例**]

```
mpiexec -np 8 ./pfcCompress
```

(1). **入力データ**

./IN_Cio **ディレクトリ配下を参照して下さい**

・ボクセルサイズと領域分割

```
GlobalVoxel         = (8, 8, 8)
GlobalDivision      = (2, 2, 2)
```

・タイムステップ

```
0step〜490step (50slice)
```

(2). **圧縮実行条件**

・領域方向分割数

```
domainDivision      = (2, 2, 1)
```

・圧縮対象タイムステップ

```
0step〜30step (4slice)
```

→　ステップ方向並列数=2

・実行並列数

領域分割数４×ステップ方向並列数２

→　MPI **８並列**

・ユーザ指定誤差

```
compressError  = 1.0;  // 誤差率 (%)
```

(3). **圧縮結果**

./OUT_Compress_sample **ディレクトリ配下を参照して下さい**

・ボクセルサイズと領域分割

```
GlobalVoxel         = (8, 8, 8)
GlobalDivision      = (2, 2, 1)
```

・タイムステップ

```
0step〜30step (4slice)
```

・圧縮情報

```
CompressInfo {
  CompressFormat     = "pod"
  CompressError      =  1.000
  CalculatedLayer    = 1
  Version            = "1.0.0"
  StartStep          = 0
  EndStep            = 30
}
```

3.3　展開機能

3.3.1　機能概要

3.3.2　展開処理手順

PFC では以下の手順で，フィールドデータの展開処理を行います.

3.3.3 展開処理のサンプルコード

1. 展開処理のサンプルコードを以下に示します.

```
#include "mpi.h"
#include "PfcRestration.h"

int main( int argc, char **argv )
{
  PFC::E_PFC_ERRORCODE ret;

  cout << "##### Restration sample start #####"<< endl;

  CPfcRestration  restration;

  std::string pfcFilePath    = "IN_Compress/vel.pfc";

  // 初期化
  ret = restration.Init(
            pfcFilePath      // index.pfc のファイルパス
          );
  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Restration Init() Error  #####"<< endl;
    return -1;
  }

  int head[3];
  int tail[3];

  // 計算領域の決定
  {
    int regionID  = 0; // 担当する領域 ID
    int gDiv[3] = { 1, 1, 1 }; // 各方向の分割数

    ret = restration.GetHeadTail (
                      gDiv,     // [in]  計算空間の領域分割数
                      regionID, // [in]  領域 ID
                      head,     // [out] 計算領域の開始位置
                      tail      // [out] 計算領域の終了位置
                );
  }

  // 圧縮データがメモリ上にロード可能か確認
  int    memUseMax = 1024; // 1GiB  使用メモリ MAX  単位 (Mib)
  double loadRatio;

  ret = restration.CheckCompressDataOnMem(
              memUseMax,  // [in]  使用メモリ MAX  単位 (Mib)
              head,       // [in]  計算領域の開始位置
              tail,       // [in]  計算領域の終了位置
              loadRatio   // [out] ロード可能な割合 ( 0.0 - 1.0 )
           );

  cout << "##### Restration CheckCompressDataOnMem() ret="<< ret
                              <<" loadRatio="<<loadRatio<< endl;

  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Restration can not loaded  #####"<< endl;
    return -1;
  }

  // 圧縮データをメモリ上にロード
  ret = restration.LoadCompressDataOnMem( head, tail );

  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Restration LoadCompressDataOnMem() Error  #####"<< endl;
```

```
    return -1;
  }

  // タイムステップリスト取得
  vector<int> timeStepList;
  ret = restration.GetTimeStepList(
                      timeStepList // [out] タイムステップリスト
                    );

  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Restration GetTimeStepList() Error  #####"<< endl;
    return -1;
  }
  int numStep = timeStepList.size();
  //int numStep = 2;  // 読みだすステップ数

  printf("##### Restration Result Dump Start #######\n");
      // 成分数 = 3
  // タイムステップループ
  for(int istep=0; istep<numStep; istep++ ) {
    printf("  ----- step = %d -------\n",istep);
    for(int iz=head[2]; iz<=tail[2]; iz++ ) {
      for(int iy=head[1]; iy<=tail[1]; iy++ ) {
        for(int ix=head[0]; ix<=tail[0]; ix++ ) {
          double dv[3];

          // データ読み込み（位置指定）
          ret = restration.ReadData (
                    dv,                   // [out] 読み込み領域
                    timeStepList[istep], // [in]  タイムステップ番号
                    ix,                   // [in]  x position ( >= 1 )
                    iy,                   // [in]  y position ( >= 1 )
                    iz                    // [in]  z position ( >= 1 )
               );
          if( ret != PFC::E_PFC_SUCCESS ) {
            cout << "##### Restration GetTimeStepList() Error  #####"<< endl;
            return -1;
          }

          int iwk = istep + iz + iy + ix;
          int ic;
          ic = 0;
          printf("istep=%3d ic=%3d iz=%3d iy=%3d ix=%3d orig=%d  dv=%15.4lf\n",
                  istep,ic,iz,iy,ix,(iwk+ic),dv[ic]);
          ic = 1;
          printf("istep=%3d ic=%3d iz=%3d iy=%3d ix=%3d orig=%d  dv=%15.4lf\n",
                  istep,ic,iz,iy,ix,(iwk+ic),dv[ic]);
          ic = 2;
          printf("istep=%3d ic=%3d iz=%3d iy=%3d ix=%3d orig=%d  dv=%15.4lf\n",
                  istep,ic,iz,iy,ix,(iwk+ic),dv[ic]);
        }
      }
    }

  } // ! タイムスタップループ
  printf("##### Restration Result Dump End  #######\n");

  cout << "##### Restration sample end #####"<< endl;


  return 0;
}
```

[実行方法]

```
./pfcRestoreIndex
```

(1). **入力データ（POD 圧縮結果）**

./IN_Compress **ディレクトリ配下を参照して下さい**

・ボクセルサイズと領域分割

```
GlobalVoxel         = (8, 8, 8)
GlobalDivision      = (2, 2, 1)
```

・タイムステップ

```
0step〜30step (4slice)
```

・圧縮情報

```
CompressInfo {
  CompressFormat     = "pod"
  CompressError      =  1.000
  CalculatedLayer    = 1
  Version            = "1.0.0"
  StartStep          = 0
  EndStep            = 30
}
```

(2). **展開実行条件**

圧縮データをメモリに展開後、
インデックス (i,j,k,step) **を指定して値を取得します。**

2. 展開処理 インデックス指定 On The Fly のサンプルコード

```
#include "mpi.h"
#include "PfcRestration.h"

int main( int argc, char **argv )
{
  PFC::E_PFC_ERRORCODE ret;

  cout << "##### Restration sample start #####"<< endl;

  CPfcRestration  restration;

  std::string pfcFilePath    = "IN_Compress/vel.pfc";

  // 初期化
  ret = restration.Init(
            pfcFilePath      // index.pfc のファイルパス
          );
  if( ret != PFC::E_PFC_SUCCESS ) {
```

```
  cout << "##### Restration Init() Error  #####"<< endl;
  return -1;
}

int head[3];
int tail[3];

// 計算領域の決定
{
  int regionID  = 0; // 担当する領域 ID
  int gDiv[3] = { 1, 1, 1 }; // 各方向の分割数

  ret = restration.GetHeadTail (
                    gDiv,     // [in]  計算空間の領域分割数
                    regionID, // [in]  領域 ID
                    head,     // [out] 計算領域の開始位置
                    tail      // [out] 計算領域の終了位置
                );
}

// タイムステップリスト取得
vector<int> timeStepList;
ret = restration.GetTimeStepList(
                     timeStepList // [out] タイムステップリスト
                   );

if( ret != PFC::E_PFC_SUCCESS ) {
  cout << "##### Restration GetTimeStepList() Error  #####"<< endl;
  return -1;
}
int numStep = timeStepList.size();
//int numStep = 2;  // 読みだすステップ数

printf("##### Restration Result Dump Start #######\n");
    // 成分数 = 3
// タイムステップループ
for(int istep=0; istep<numStep; istep++ ) {
  printf("  ----- step = %d -------\n",istep);
  for(int iz=head[2]; iz<=tail[2]; iz++ ) {
    for(int iy=head[1]; iy<=tail[1]; iy++ ) {
      for(int ix=head[0]; ix<=tail[0]; ix++ ) {
        double dv[3];

        // データ読み込み（位置指定）
        ret = restration.ReadData (
                  dv,                 // [out] 読み込み領域
                  timeStepList[istep], // [in]  タイムステップ番号
                  ix,                 // [in]  x position ( >= 1 )
                  iy,                 // [in]  y position ( >= 1 )
                  iz                  // [in]  z position ( >= 1 )
             );
        if( ret != PFC::E_PFC_SUCCESS ) {
          cout << "##### Restration GetTimeStepList() Error  #####"<< endl;
          return -1;
        }

        int iwk = istep + iz + iy + ix;
        int ic;
        ic = 0;
        printf("istep=%3d ic=%3d iz=%3d iy=%3d ix=%3d orig=%d  dv=%15.4lf\n",
                istep,ic,iz,iy,ix,(iwk+ic),dv[ic]);
        ic = 1;
        printf("istep=%3d ic=%3d iz=%3d iy=%3d ix=%3d orig=%d  dv=%15.4lf\n",
                istep,ic,iz,iy,ix,(iwk+ic),dv[ic]);
        ic = 2;
        printf("istep=%3d ic=%3d iz=%3d iy=%3d ix=%3d orig=%d  dv=%15.4lf\n",
                istep,ic,iz,iy,ix,(iwk+ic),dv[ic]);
      }
    }
  }
```

```
  } // ! タイムスタップループ
  printf("##### Restration Result Dump End  #######\n");

  cout << "##### Restration sample end #####"<< endl;


  return 0;
}
```

[実行方法]

```
./pfcRestoreOTF
```

(1). 入力データ（POD 圧縮結果）

./IN_Compress ディレクトリ配下を参照して下さい

・ボクセルサイズと領域分割

```
GlobalVoxel         = (8, 8, 8)
GlobalDivision      = (2, 2, 1)
```

・タイムステップ

```
0step〜30step (4slice)
```

・圧縮情報

```
CompressInfo {
  CompressFormat     = "pod"
  CompressError      =  1.000
  CalculatedLayer    = 1
  Version            = "1.0.0"
  StartStep          = 0
  EndStep            = 30
}
```

(2). 展開実行条件

圧縮データをメモリに展開せずに、
インデックス (i,j,k,step) を指定して値を取得します。
（読み出す度にファイルを読みに行きます）

3. 展開処理 インデックス指定のサンプルコード

```
#include "mpi.h"
#include "cio_DFI.h"
#include "PfcRestration.h"

#define ERROR               0.000001

double speed_length(double x, double y, double z){
  return(sqrt(x*x+y*y+z*z));
}

//*************************************************
int main( int argc, char **argv )
{
  PFC::E_PFC_ERRORCODE ret;

  cout << "##### Restration sample start #####"<< endl;

  //MPI Initialize  for CIO
  if( MPI_Init(&argc,&argv) != MPI_SUCCESS )
  {
      std::cerr << "MPI_Init error." << std::endl;
      return false;
  }

  CPfcRestration  restration;

  std::string pfcFilePath    = "IN_Compress/vel.pfc";
  std::string dfiFilePath    = "IN_Cio/vel.dfi";


  // PFClib 展開処理初期化
  ret = restration.Init(
             pfcFilePath      // index.pfc のファイルパス
           );
  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### Restration Init() Error  #####"<< endl;
    return -1;
  }

  int head[3];
  int tail[3];
  int numRegion = 1; // 全領域数
  int regionID  = 0; // 担当する領域 ID
  int gDiv[3]   = { 1, 1, 1 }; // 各方向の分割数
  int gVoxel[3]; // 各方向の voxel 数

  // 計算領域の決定
  {
    restration.GetGlobalVoxel( gVoxel );

    ret = restration.GetHeadTail (
                      gDiv,     // [in]  計算空間の領域分割数
                      regionID, // [in]  領域 ID
                      head,     // [out] 計算領域の開始位置
                      tail      // [out] 計算領域の終了位置
                );
    printf("head[3] = %d %d %d\n",head[0],head[1],head[2]);
    printf("tail[3] = %d %d %d\n",tail[0],tail[1],tail[2]);
  }

  // CIO 初期化
  int guideCell = 0;
  double  f_time;
  unsigned int i_dummy;
  double  f_dummy;
  CIO::E_CIO_ERRORCODE ret_cio;

  cio_DFI* pDfiIN = cio_DFI::ReadInit(
                        MPI_COMM_WORLD,
                        dfiFilePath,
                        gVoxel,
```

```
                      gDiv,
                      ret_cio );


// タイムステップリスト取得
vector<int> timeStepList;
ret = restration.GetTimeStepList(
                    timeStepList // [out] タイムステップリスト
                  );

if( ret != PFC::E_PFC_SUCCESS ) {
  cout << "##### Restration GetTimeStepList() Error  #####"<< endl;
  return -1;
}
int numStep = timeStepList.size();

// 領域アロケート
int size = (tail[0]-head[0]+1)*(tail[1]-head[1]+1)*(tail[2]-head[2]+1);
int numComponent = 3;   // 成分数
   // CIO Sph original data
double* dv_orig = new double[size*numComponent*numStep];

double* dv = new double[size*numComponent*numStep];


// タイムステップループ
for(int istep=0; istep<numStep; istep++ ) {
  int ip = size*numComponent*istep;

  // PFC Pod データ読み込み（範囲指定）
  ret = restration.ReadData (
              &dv[ip],             // [out] 読み込み領域先頭
              timeStepList[istep], // [in]  タイムステップ番号
              head,                // [in]  計算領域の開始位置
              tail                 // [in]  計算領域の終了位置
            );
  if( ret != PFC::E_PFC_SUCCESS ) {
    cout << "##### restration.ReadData() Error ret="<<ret<<"  #####"<< endl;
    return -1;
  }

  // CIO Sph original データ読み込み

  pDfiIN->ReadData(
                 &dv_orig[ip],
                 timeStepList[istep],
                 guideCell,
                 gVoxel,
                 gDiv,
                 head,
                 tail,
                 f_time,
                 true,
                 i_dummy,
                 f_dummy
               );

} // ! タイムスタップループ

// 検証
int ix_size = (tail[0]-head[0]+1);
int iy_size = (tail[1]-head[1]+1);
int iz_size = (tail[2]-head[2]+1);
int nsize   = ix_size*iy_size*iz_size;
double max_abs_diff = 0.0;

// NIJK の場合のダンプ出力
printf("##### Restration Result Dump Start #######\n");
for(int istep=0; istep<numStep; istep++ ) {
printf("  ----- step = %d -------\n",istep);
  for(int iz=head[2]; iz<=tail[2]; iz++ ) {
```

```
    for(int iy=head[1]; iy<=tail[1]; iy++ ) {
      for(int ix=head[0]; ix<=tail[0]; ix++ ) {
        for(int ic=0; ic<numComponent; ic++ ) {
          int ip =   numComponent*ix_size*iy_size*iz_size*istep
                   + numComponent*ix_size*iy_size*(iz-head[2])
                   + numComponent*ix_size*(iy-head[1])
                   + numComponent*(ix-head[0])
                   + ic;

          double diff = dv_orig[ip] - dv[ip];
          printf("istep=%3d ic=%3d iz=%3d iy=%3d ix=%3d  orig=%15.4lf  val=%15.4lf  diff=%15.4lf\n",
                  istep,ic,iz,iy,ix,dv_orig[ip],dv[ip],diff);
          if( fabs(diff) > max_abs_diff ) {
            max_abs_diff = fabs(diff);
          }
        }
      }
    }
  }
}
printf("\n");
printf("**** max_abs_diff =%15.4lf\n",max_abs_diff);

double* error_co = new double[nsize];
double* error    = new double[numStep];
double* sum_orig = new double[numStep];
double* fc       = new double[numStep];
double* largest_error = new double[numStep];
double* sum_compression = new double[numStep];
double temp_data1, temp_data2;
int    ip;

for(int i = 0 ; i < numStep; i++){
  sum_orig[i] = 0.0;
  sum_compression[i] = 0.0;
  largest_error[i] = -1.0;
  fc[i] = 0.0;
}

for(int istep=0; istep<numStep; istep++ ) {
  double average = 0.0;
  double fangcha = 0.0;
  double ave1 = 0.0;
  double ave2 = 0.0;
  double ave = 0.0;
  double temp;

  for(int i=0; i<nsize; i++ ) {
    ip = nsize*istep + numComponent*i;
    temp_data1 = speed_length(dv_orig[ip], dv_orig[ip+1], dv_orig[ip+2]);
    temp_data2 = speed_length(dv[ip], dv[ip+1], dv[ip+2]);

    if(fabs(temp_data1) < ERROR ) {
      temp = 0.0;
    } else {
      temp = 1.0/(temp_data1*temp_data1);
    }
    ave1 = ave1 + (temp_data1 - temp_data2)*temp;
    ave2 = ave2 + temp;

  }
  ave = ave1/ave2;

  for(int i=0; i<nsize; i++ ) {
    ip = nsize*istep + numComponent*i;
    temp_data1 = speed_length(dv_orig[ip], dv_orig[ip+1], dv_orig[ip+2]);
    temp_data2 = speed_length(dv[ip], dv[ip+1], dv[ip+2]) + ave;

    if(fabs(temp_data1) < ERROR ) {
     error_co[i] = 0.0;
    } else {
```

```
        error_co[i] = fabs(temp_data1-temp_data2)/temp_data1/nsize;
       average = average + error_co[i];
      }
    }


    for(int i=0; i<nsize; i++ ) {
      ip = nsize*istep + numComponent*i;
      temp_data1 = speed_length(dv_orig[ip], dv_orig[ip+1], dv_orig[ip+2]);
      temp_data2 = speed_length(dv[ip], dv[ip+1], dv[ip+2]) + ave;

      if(fabs(temp_data1) < ERROR ){
        temp_data1 = 0.0;
        temp_data2 = 0.0;
      }

      sum_orig[istep] = sum_orig[istep] + temp_data1/nsize;
      sum_compression[istep] = sum_compression[istep] + fabs(temp_data1-temp_data2)/nsize;
    }

    error[istep] = sum_compression[istep]/sum_orig[istep]*100.0;
    for(int i = 0; i < nsize; i++) {
      fangcha = fangcha + pow(error_co[i]*nsize-average,2);
    }
    fangcha = fangcha / nsize;
    fangcha = sqrt(fangcha);
    fc[istep] = fangcha;

    cout<<endl;
    cout<<"#### finish the "<<istep<<"th timestep ######"<<endl;
    cout<<"   error:"<<error[istep]<<"  fangcha:"<<fangcha<<endl;
    cout<<endl;
  }

  delete [] dv;
  delete [] dv_orig;
  delete [] error_co;
  delete [] error;
  delete [] sum_orig;
  delete [] fc;
  delete [] largest_error;
  delete [] sum_compression;
  delete pDfiIN;

  // CIO
  MPI_Finalize();

  cout << "##### Restration sample end #####"<< endl;


  return 0;
}
```

[**実行例**]

MPI 1 **並列**

```
./pfcRestoreRange
```

(1). 入力データ (original)

./IN_Cio ディレクトリ配下を参照して下さい

・ボクセルサイズと領域分割

```
GlobalVoxel         = (8, 8, 8)
GlobalDivision      = (2, 2, 2)
```

・タイムステップ

0step〜490step (50slice)

(2). 入力データ（POD 圧縮結果）

./IN_Compress ディレクトリ配下を参照して下さい

・ボクセルサイズと領域分割

```
GlobalVoxel         = (8, 8, 8)
GlobalDivision      = (2, 2, 1)
```

・タイムステップ

0step〜30step (4slice)

・圧縮情報

```
CompressInfo {
  CompressFormat     = "pod"
  CompressError      =  1.000
  CalculatedLayer    = 1
  Version            = "1.0.0"
  StartStep          = 0
  EndStep            = 30
}
```

(3). 展開実行条件

・領域方向分割数

```
domainDivision      = (1, 1, 1)
```

・展開処理そのものは MPI 実行する必要はありませんが
CIOlib が MPI を要求するため、並列数１で MPI 実行しています。

第 4 章

ステージングツール

この章では，PFClib のステージングツールについて説明します．

4.1　ステージングツール

4.1.1　機能概要

ステージングツール pfcfrm(PFC FileRankMapper) は，大規模並列計算機で PFC ライブラリを使用する上で，各計算ノード (MPI ランク) 毎に必要なファイルを，ランク番号で命名したディレクトリにコピーする，ステージング対応用のバッチプログラムです．

(1) 領域分割対応

領域分割方向の対応と時間軸（タイムステップ）方向の並列化対応を行います．

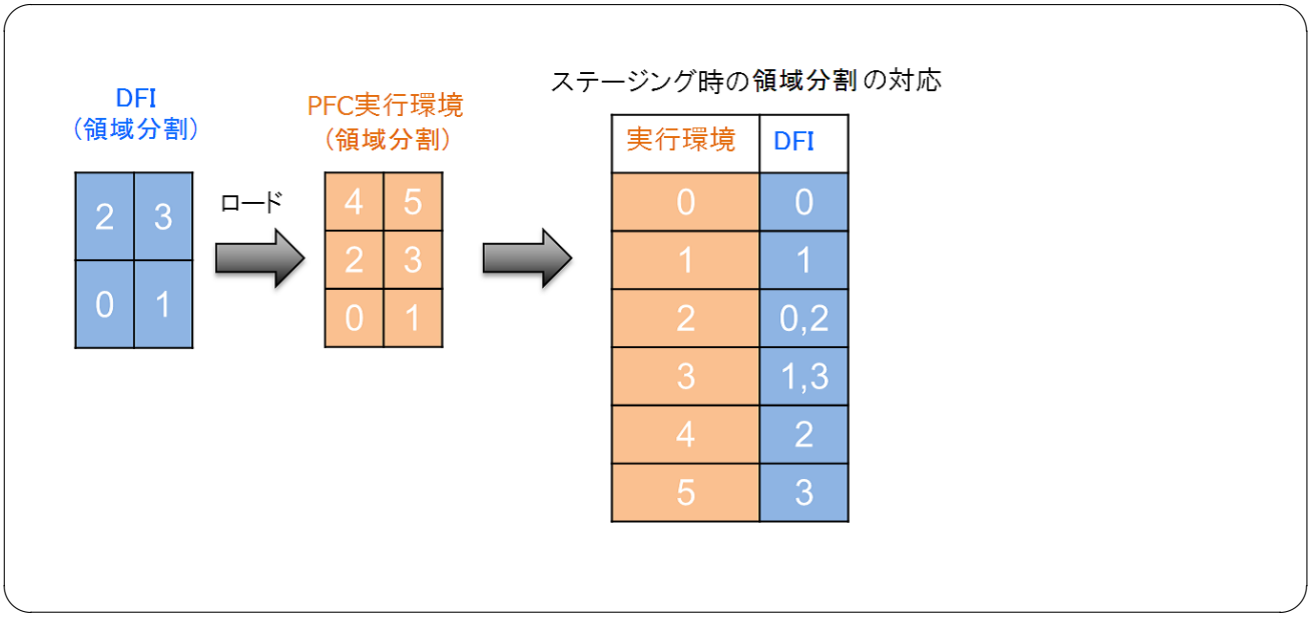


図 4.1　領域方向の分割

(2) 時間軸方向の並列化対応

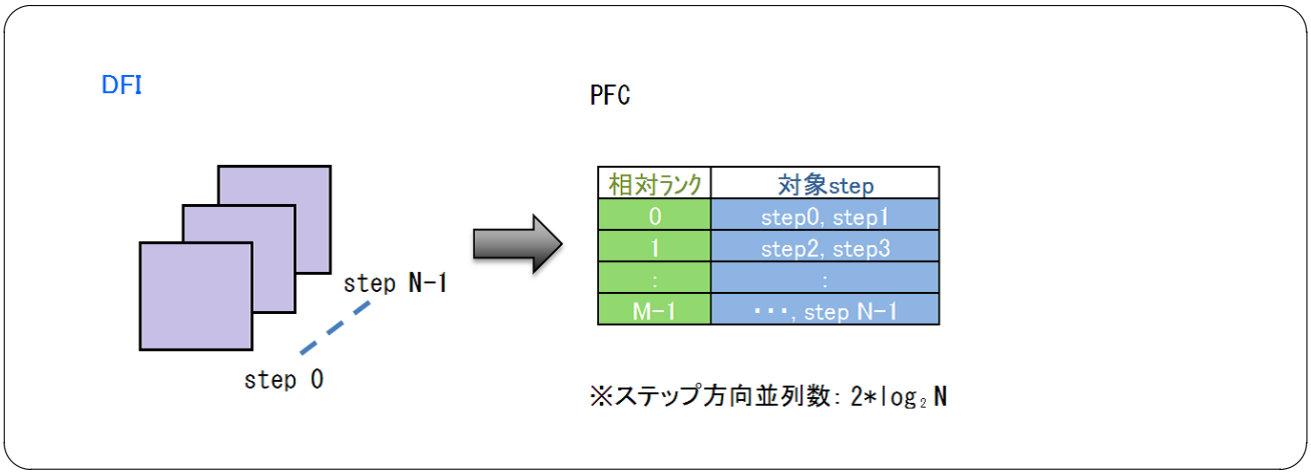


図 4.2　時間軸方向の分割

実際のステージングの機能は (1) と (2) を合わせた形となる．

領域分割数 4，step 数 10 の例を示す．

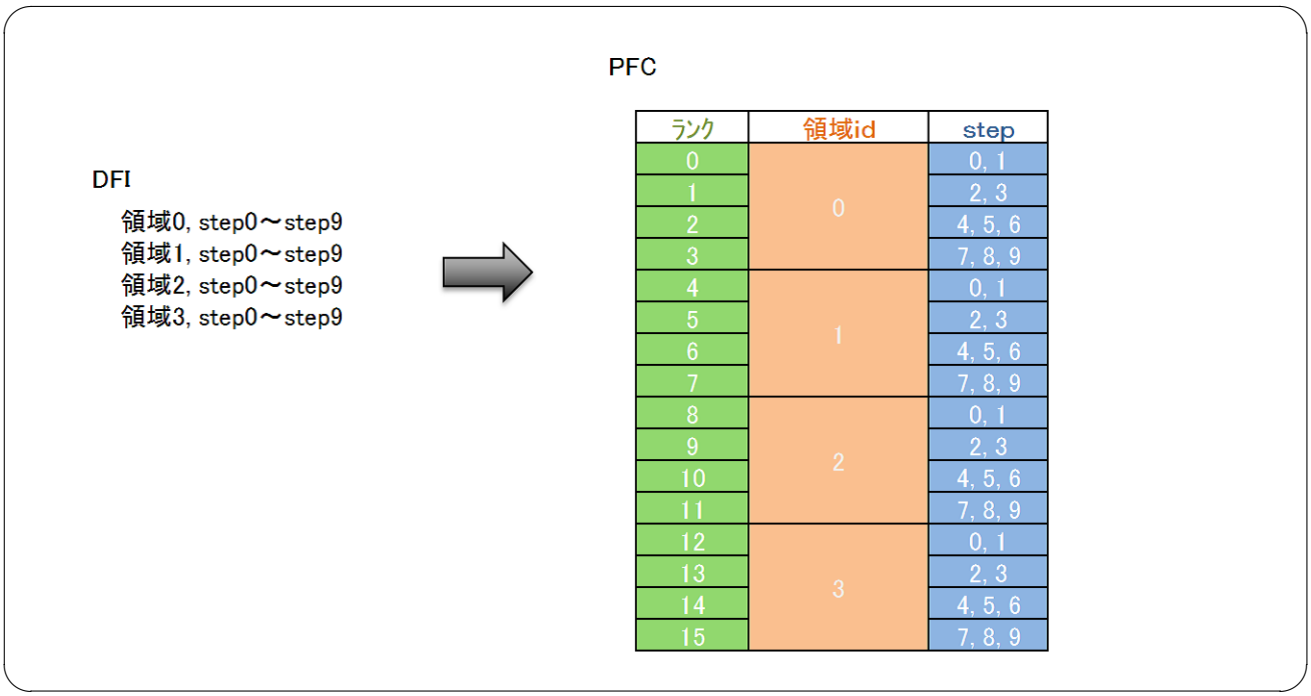


図 4.3　時間軸方向の分割例

4.1.2　ステージングツールのインストール

pfcfrm は，CIO パッケージのビルド (configure，make，makeinstall) が行われるときに同時にビルドされ，configure スクリプト実行時の設定 `prefix` 配下の`${prefix}`/bin に `make install` 時にインストールされます．

4.1.3　使用方法

pfcfrm はコマンドを実行して使用します．

コマンド引数

以下の引数を指定します．([] は省略可能なオプション)

```
$ pfcfrm -i proc.txt [-s StartStepNo] [-e EndStepNo] [-o outDir] PFCfile
```

引数の説明

-i proc.txt (必須)

これから計算するソルバーの領域分割情報が記述されたファイル名を指定します．

proc.txt にソルバーの Domain 情報が入ったファイル名 (TextParser 形式) を指定します．

領域分割情報が記述されたファイル proc.txt の仕様は**??**参照．

-s StartStepNo (省略可)

開始ステップ番号を指定します．

StartStepNo に対象とする開始ステップ番号を指定します．

省略した場合は先頭ステップが対象となります．

(例) -s 100
PFCfile で指定したファイル中の step100 以降のファイルについて各ランクのディレクトリにコピーされます.

-e EndStepNo (省略可)
終了ステップ番号を指定します.
EndStepNo に対象とする終了ステップ番号を指定します.
省略した場合は最終ステップが対象となります.
(例) -e 200
PFCfile で指定したファイル中の step200 以前のファイルについて各ランクのディレクトリにコピーされます.

-o outDir (省略可)
振り分け結果のコピー先のディレクトリ名を指定します.
outDir にディレクトリ名を指定します.
省略した場合はカレントディレクトリが出力先となります.
(例 1) -o hoge
カレントディレクトリに hoge/ディレクトリが生成され，そのディレクトリ配下に各ランク用の 000000/, 000001/,... ディレクトリが生成されます.
(例 2) 省略時
カレントディレクトリに各ランク用の 000000/,000001/,... ディレクトリが生成されます.

PFCfile... (必須)
振り分け対象とする PFC ファイル名 (index.pfc) を指定します.

実行例

4 分割 (2,1,2) の結果を 8 分割 (2,2,2) でリスタートする例

・ソルバーの Domain 情報格納ファイル (solvproc.txt)

```
Domain {
  GlobalVoxel=(64,64,64)
  GlobalDivision=(2,2,2)
  ActiveSubdomainFile=""
}
```

・振り分け対象の DFI ファイル
old ディレクトリ配下の prs.dfi,vel.dfi
実体の sph ファイルは SPH/ディレクトリに存在.

```
old/
  prs.dfi           <--DirectoryPath="SPH"
  vel.dfi           <--DirectoryPath="SPH"
  proc.dfi          <--prs.dfi, vel.dfi から参照
```

```
SPH/
  prs_0000000000_id000000.sph
  prs_0000000000_id000001.sph
  prs_0000000000_id000002.sph
  prs_0000000000_id000003.sph
  prs_0000000100_id000000.sph
  prs_0000000100_id000001.sph
  prs_0000000100_id000002.sph
  prs_0000000100_id000003.sph
  vel_0000000000_id000000.sph
  vel_0000000000_id000001.sph
  vel_0000000000_id000002.sph
  vel_0000000000_id000003.sph
  vel_0000000100_id000000.sph
  vel_0000000100_id000001.sph
  vel_0000000100_id000002.sph
  vel_0000000100_id000003.sph
```

・振り分け対象ステップ番号
ステップ 100 のファイル

・出力先ディレクトリ
hoge/

・実行コマンド

```
$ frm -i solvproc.txt -s 100 -o hoge old/prs.dfi old/vel.dfi
```

・出力結果
hoge/ディレクトリが生成され，その配下に 6 桁のランク番号ディレクトリが生成されます．各ランク用ディレクトリ配下にそれぞれ必要なファイルがコピーされます．

```
hoge/000000/
  prs.dfi                          <--DirectoryPath="./"
  prs_0000000100_id000000.sph
  prs_proc.dfi                     <--proc.dfi からコピーされる
  vel.dfi                          <--DirectoryPath="./"
  vel_0000000100_id000000.sph
  vel_proc.dfi                     <--proc.dfi からコピーされる

hoge/000001/
  prs.dfi
  prs_0000000100_id000001.sph
  prs_proc.dfi
  vel.dfi
  vel_0000000100_id000001.sph
  vel_proc.dfi
```

```
hoge/000002/
  prs.dfi
  prs_0000000100_id000000.sph
  prs_proc.dfi
  vel.dfi
  vel_0000000100_id000000.sph
  vel_proc.dfi

hoge/000003/
  prs.dfi
  prs_0000000100_id000001.sph
  prs_proc.dfi
  vel.dfi
  vel_0000000100_id000001.sph
  vel_proc.dfi

hoge/000004/
  prs.dfi
  prs_0000000100_id000002.sph
  prs_proc.dfi
  vel.dfi
  vel_0000000100_id000002.sph
  vel_proc.dfi

hoge/000005/
  prs.dfi
  prs_0000000100_id000003.sph
  prs_proc.dfi
  vel.dfi
  vel_0000000100_id000003.sph
  vel_proc.dfi

hoge/000006/
  prs.dfi
  prs_0000000100_id000002.sph
  prs_proc.dfi
  vel.dfi
  vel_0000000100_id000002.sph
  vel_proc.dfi

hoge/000007/
  prs.dfi
  prs_0000000100_id000003.sph
  prs_proc.dfi
```

```
vel.dfi
vel_0000000100_id000003.sph
vel_proc.dfi
```

第 5 章

ファイル仕様

PFClib で使用しているファイルの仕様について説明します.

5.1　ファイル仕様

5.1.1　インデックスファイル（index.pfc）仕様

index.pfc ファイルはファイル情報（FileInfo），圧縮情報（CompressInfo），ファイルパス情報（FilePath），単位系（UnitList），時系列データ（TimeSlice）の５つのブロックで構成されています．

以下に，index.pfc ファイルの仕様とサンプルをブロック毎に示します．

ファイル情報（FileInfo）の仕様

```
FileInfo
{
  DirectoryPath = "./hoge"   // フィールドデータの存在するディレクトリ
  Prefix        = "vel"      // ベースファイル名                (※ 1)
  FileFormat    = "pod"      // ファイルタイプ，拡張子          (※ 1)
  GuideCell     = 0          // 仮想セル数 ( =0 固定 )
  DataType      = "Float64"  // データタイプ                    (※ 2)
  Endian        = "little"   // データのエンディアン            (※ 3)
  ArrayShape    = "ijkn"     // 配列形状                        (※ 4)
  Component     = 3          // 成分数（スカラーは不要）        (※ 4)
}
```

(※ 1) ファイル名
　　並列時 [Prefix]_id[領域 ID:6 桁].[ext]
　　逐次時 [Prefix].[ext]
　　pod の係数ファイルの拡張子は coef とする
(※ 2) Float64
　　圧縮前データが Float32 の場合は圧縮時の誤差が大きくなるので Float64 に変換する
(※ 3) little,big
(※ 4) ijkn,nijk
　　ijkn:(imax,jmax,kmax,Component)
　　nijk:(Component,imax,jmax,kmax)

圧縮情報（CompressInfo）の仕様

```
CompressInfo
{
  CompressFormat  = "pod"    // 圧縮形式              (※ 5)
  CompressError   = 0.01     // 利用者指定誤差率（%）
  CalculatedLayer = 4        // 計算レイヤー数        (※ 6)
  Version         = "1.0.0"  // 圧縮形式のバージョン
  StartStep       = 0        // 開始ステップ
  EndStep         = 90       // 終了ステップ
}
```

(※ 5) pod pod 以外は reserve
(※ 6) pod の時のみ有効
　　精度確保のため，指定された誤差率より計算レイヤー数が変更される可能性あり

ファイルパス（FilePath）の仕様

```
FilePath
{
  DfiPath    = "vel.dfi"      // CIO DFI ファイルパス
  PfcProcess = "proc.pfc"     // PFC proc ファイル名
                              //   並列情報，領域情報（全体，分割）
}
```

単位系（UnitList）の仕様

```
UnitList
{
  Length {
    Unit       = "NonDimensional"    // (NonDimensional, m, cm, mm)
    Reference  = 1.000000e+00
  }
  Pressure {
    Unit       = "NonDimensional"
    Reference  = 0.000000e+00
    Difference = 1.176300e+00        // 圧力差 (Pa)
  }
  Velocity {
    Unit       = "NonDimensional"
    Reference  = 1.000000e+00
  }
}
```

時系列データ（TimeSlice）の仕様

```
TimeSlice
{
  Slice[@] {            // ファイル出力回数分
    Step = 0                    // 出力ステップ
    Time = 0.0                  // 出力時刻
    AverageTime =               // 平均時間（必要に応じて出力）
    AverageStep =               // 平均化したステップ数（必要に応じて出力）
    VectorMinMax {
      Min = 0.000000e+00
      Max = 0.000000e+00
    }
    MinMax[@] {         // Component 個
      Min  = -1.56e-2           // 最小値
      Max  = 8.2e-01            // 最大値
    }
  }
  Slice[@] {
    Step = 10                   // 出力ステップ
    Time = 3.125e-2             // 出力時刻
    AverageTime =               // 平均時間（必要に応じて出力）
    AverageStep =               // 平均化したステップ数（必要に応じて出力）
    VectorMinMax {
      Min = 0.000000e+00
      Max = 0.000000e+00
    }
    MinMax[@] {         // Component 個
      Min  = -4.000938e-05      // 最小値
      Max  = 2.169153e-04       // 最大値
    }
  }
  Slice{@} {
    :
    :
  }
}
```

5.1.2　プロセス情報ファイル（proc.pfc）仕様

proc.pfc ファイルはドメイン情報（Domain），分割領域情報（Division）の２つのブロックで構成されています．

以下に，proc.pfc ファイルの仕様とサンプルをブロック毎に示します．

ドメイン情報（Domain）の仕様

```
Domain
{
  GlobalOrigin   = (-3.00, -3.00, -3.00)      // 計算空間の起点座標
  GlobalRegion   = ( 6.00,  6.00,  6.00)      // 計算空間の各軸方向の長さ
  GlobalVoxel    = (64, 64, 64)               // 計算領域全体のボクセル数
  GlobalDivision = (2, 2, 2)                  // 計算領域の部分領域の分割数
}
```

分割領域情報（Division）の仕様

```
Division {{
  Region[@] { //  分割領域の数あり（上記の例では 2x2x2->8 個 データあり）
    ID        = 0                    // 分割した領域の ID
    VoxelSize = (32, 32, 32)         // ボクセルサイズ
    HeadIndex = (1, 1, 1)            // 始点インデクス　グローバルで (1,1,1) からスタート
    TailIndex = (32, 32, 32)         // 終点インデクス　グローバルで (64, 64, 64) が終端
  }
  Region[@] {
  ...
}
```

5.1.3　圧縮制御情報ファイル（pfc_cntl）仕様

pfc_cntl ファイルは圧縮制御情報（PfcCompressCntl）のブロックで構成されています.

以下に，pfc_cntl ファイルの仕様とサンプルを示します.

圧縮制御情報（PfcCompressCntl）の仕様

```
PfcCompressCntl
{
  DomainDivision  = (2, 2, 2)   // 計算領域の部分領域の分割数　※必須

  ItemCntl[@] { // 属性（変数）ごとに指定する

    InputDfiPath    = "vel.dfi" // CIO の index.dfi のパス     ※必須
                                // Input となる DFI パスを指定
    OutputDirectoryPath = "./"  // 出力ディレクトリパス

    CompressFormat  = "pod"     // 圧縮形式　※必須
    CompressError   = 0.01      // 許容誤差率（%）      ※省略可（省略時：0.01）

    StartStep       = 100       // 圧縮開始ステップ　※省略可（省略時：dfi ファイルに従う）※ 1
    EndStep         = 200       // 圧縮終了ステップ　※省略可（省略時：dfi ファイルに従う）※ 1

    ProcFileSave    = “ON”    // proc.pfc ファイル出力の有無　(ON/OFF) ）　※必須　※ 2
    OptSave         = "ON"      // 検証用：全ての圧縮段階の基底ベクトルと係数の組を残す
                                // ※省略可（省略時： "OFF" ）
  }

  ItemCntl[@] {
    ・・・
  }
  ・・・
}
```

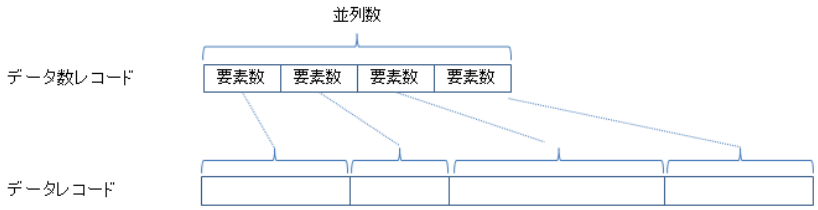
(※ 1) 同一ファイル内で同じ属性（変数）に対して，複数の開始ステップ〜終了ステップを指定することは出来ません.
(※ 2) 全体で proc.pfc ファイルは 1 ファイルあれば良い．複数 Save 指定された場合など，proc.pfc ファイルが既に存在する場合は上書きされる.

表 5.1　基底ファイル

レコード名	型	サイズ	出力数	内容
エンディアンチェックレコード	整数	4byte	1	整数 1 固定 Read 時に 1 として読み込めたかどうかをチェックして エンディアンを判定する
データタイプレコード	整数	4byte	1	整数 2 固定 2：倍精度浮動小数点
タイムステップ数レコード	整数	4byte	1	タイムステップ数
並列数レコード	整数	4byte	1	圧縮時の並列数
レイヤー数レコード	整数	4byte	1	圧縮時の計算レイヤー数
データ数レコード	整数	4byte	並列数	各並列ごとのデータ（要素）数
データレコード	倍精度	8*データ数	並列数	各並列ごとのデータ

5.2　PFC ファイル仕様

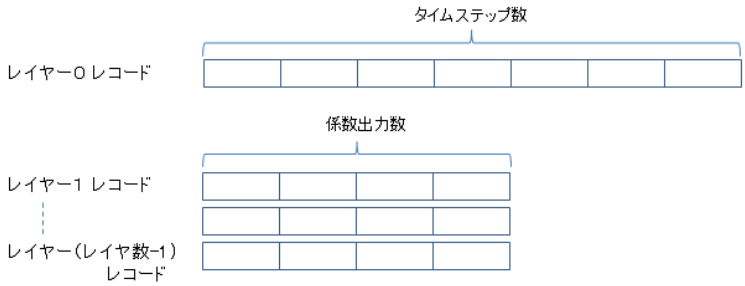
5.2.1　基底ファイル



5.2.2　係数ファイル

表 5.2　係数ファイル

レコード名	型	サイズ	出力数	内容
エンディアンチェックレコード	整数	4byte	1	整数 1 固定 Read 時に 1 として読み込めたかどうかをチェックして エンディアンを判定する
データタイプレコード	整数	4byte	1	整数 2 固定 2：倍精度浮動小数点
タイムステップ数レコード	整数	4byte	1	タイムステップ数
係数出力数レコード	整数	4byte	1	係数出力数=2^((int)log2(タイムステップ数）) （並列数× 2）
レイヤー数レコード	整数	4byte	1	圧縮時の計算レイヤー数
データレコード（レイヤ 0）	倍精度	8byte	数	レイヤー 0 のデータ
データレコード（レイヤ 1 以降）	倍精度	8byte	係数出力数	レイヤー 1 以降のデータ



5.3　PFC ファイル仕様 (デバッグ用)

5.3.1　基底ファイル (デバッグ用)

表 5.3　基底ファイル (デバッグ用)

レコード名	型	サイズ	出力数	内容
エンディアンチェックレコード	整数	4byte	1	整数 1 固定 Read 時に 1 として読み込めたかたどうかをチェックして エンディアンを判定する
データタイプレコード	整数	4byte	1	整数 2 固定 2：倍精度浮動小数点
タイムステップ数レコード	整数	4byte	1	タイムステップ数
並列数レコード	整数	4byte	1	圧縮時の並列数
レイヤー数レコード	整数	4byte	1	圧縮時の計算レイヤー数
データ数レコード	整数	4byte	1	出力データ（要素）数
データレコード	倍精度	8*データ数	1	圧縮データ

ファイル名 並列時 [Prefix]_id[領域 ID:6 桁]-[領域内の連番:5 桁]_layer[レイヤー No:2 桁].pod
・各レイヤー毎に出力する
・各ランク毎に出力する（ランクで集めるとサイズが大きくなりすぎるため）
・領域数が 0 であっても、ファイル名に領域 ID を含む。

第 6 章

アップデート情報

アップデート情報について記します.

6.1　アップデート情報

本文書のアップデート情報について記します.

Version 1.0　　2014/02/20

- リリース

第 7 章

Appendix

7.1　API メソッド一覧

以下に，PFC ライブラリが提供する API メソッドの一覧を示します．（表 7.1）

表 7.1　メソッド一覧（クラス名の無い C++ メソッドは cio_DFI クラスメンバ）

機能	C++ API	備考
読込み用インスタンスの生成	ReadInit	static メソッド
出力用インスタンスの生成	WriteInit	float 版，static メソッド
	WriteInit	double 版，static メソッド
cio_FileInfo クラスポインタの取得	GetcioFileInfo	
cio_FilePath クラスポインタの取得	GetcioFilePath	
cio_Unit クラスポインタの取得	GetcioUnit	
cio_Domain クラスポインタの取得	GetcioDomain	
cio_MPI クラスポインタの取得	GetcioMPI	
cio_TimeSlice クラスポインタの取得	GetcioTimeSlice	
cio_Process クラスポインタの取得	GetcioProcess	
フィールドデータの読込み	ReadData	読込んだデータの配列ポインタが戻される
	ReadData	引数で渡された配列ポインタに読み込まれる
フィールドデータの出力	WriteData	
proc.dfi ファイル出力	WriteProcDfiFile	float 版
	WriteProcDfiFile	double 版
DFI の配列形状を取得	GetArrayShapeString	文字列を取得
	GetArrayShape	列挙型を取得
DFI のデータタイプ取得	GetDataTypeString	文字列を取得
	GetDataType	列挙型を取得
DFI の成分数取得	GetNumComponent	
データタイプを文字列から列挙型に変換	ConvDatatypeS2E	static メソッド
データタイプを列挙型から文字列に変換	ConvDatatypeE2S	static メソッド
DFI の GlobalVoxel の取得	GetDFIGlobalVoxel	
DFI の GlobalDivision の取得	GetDFIGlobalDivision	
単位系を追加	AddUnit	
単位系を取得（クラス単位）	GetUnitElem	
単位系を取得（メンバ変数）	GetUnit	
FileInfo の成分名を登録する	setComponentVariable	
FileInfo の成分名を取得する	getComponentVariable	
DFI の MinMax の合成値を取得する	getVectorMinMax	
DFI の MinMax を取得する	getMinMax	
読込みランクリストの生成	CheakReadRank	
インターバルステップの登録	setIntervalStep	
インターバルタイムの登録	setIntervalTime	
インターバルの時間を無次元化する	normalizeTime	base_time,interval_time,start_time, last_time 全て無次元化する
インターバルの base_time を無次元化	normalizeBaseTime	
インターバルの interval を無次元化	normalizeIntervalTime	
インターバルの start_time を無次元化	normalizeStartTime	
インターバルの last_time を無次元化	normalizeLastTime	
インターバルの DetlaT を無次元化	normalizeDelteT	
CIO のバージョン No の取り出し	getVersionInfo	static メソッド