Wisteria/BDEC-01 利用事例（6）
マルチプログラム連成ライブラリ h3-Open-UTIL/MP (2/2)

荒川隆（CliMTech）
八代尚（国立環境研究所）
住元真司（東京大学情報基盤センター）
中島研吾（東京大学情報基盤センター）

1　はじめに

前回（2022 年 5 月号）[1]と今回の 2 回に分けて複数のシミュレーションプログラムを連成実行し「計算・データ・学習」融合を推進するためのライブラリ h3-Open-UTIL/MP について紹介する。前回は一般的な単一アーキテクチャ環境で複数プログラムを連成するケースについて説明したが，今回は h3-Open-UTIL/MP の特徴的な機能である異機種間連成，Python APP 連成，アンサンブル連成機能などについて紹介する。連成計算の基本的な概念や実行方法については前号で詳述しているので，これらについて未経験の読者は先に前号の記事を参照されたい。また，異機種間連成を実現するために h3-Open-UTIL/MP はライブラリ h3-Open-SYS/WaitIO を利用しているが，h3-Open-SYS/WaitIO についてはスーパーコンピューティングニュース前々号と前号に紹介記事が掲載されている[2][3]ので併せて参照いただけると幸いである。h3-Open-SYS/WaitIO は記事掲載後も開発が続けられ，インターフェースなどに若干の仕様変更がなされている。そこで，本稿では h3-Open-SYS/WaitIO の最新情報についても記述する。なお前回と同様，このような計算方法に対しては基本的に「連成計算」の語を用い，一部はコミュニティの慣習に従って結合（例：大気海洋結合）を用いることとする。また以下では h3-Open-SYS/WaitIO は単に WaitIO と表記する。

2　異機種間連成機能

2.1 異機種間連成の概要

東京大学情報基盤センターの Wisteria/BDEC-01 をはじめ，名古屋大学の不老や海洋研究開発機構の地球シミュレータなど，近年はアーキテクチャの異なる複数のサブシステムで構成される計算機が増えている。これは現代のスーパーコンピューティングの対象領域がシミュレーションにとどまらず，機械学習や大規模データ処理など要求性能の異なる多様なタスクに拡大していることに対応するためである。これらのサブシステム同士は基本的にはファイル共有によって計算＋データ＋学習連携機能を実現しているが，サブシステム間でのリアルタイムな連成計算が可能になれば，シミュレーションと解析の同時実行やシミュレーションへの機械学習の反映など，ファイルを経由した逐次的な処理では対応できなかった計算手法が実現できる。このような背景から WaitIO が開発され，別掲記事で紹介されているように観測データ同化による長周期地震動リアルタイム予測などに利用されている。この異機種間通信機能を h3-Open-UTIL/MP に組み込む事により時空間構造の異なる複数のプログラムを異機種間で連成することが可能となる。

2.2 WaitIO の最新情報

前節で述べたように WaitIO は現在も精力的に開発が進んでおり[3]の記事掲載後も使用方法に若干の変更がある。そこで本節では WaitIO の最新情報について説明する。前号記事掲載後に行われたアップデートのうち利用者に直接関係するのは WaitIO が Environmental Modules に対応したことである。Wisteria/BDEC-01 では環境設定に Environment Modules を用いており，コンパイラの切り替えなどを module コマンドで行うようになっている。WaitIO の設定を Environment Modules で行う場合の具体的なコマンドはごく一般的な module load waitio であるが，注意すべき点として，異機種間通信ライブラリという WaitIO の役割に対応するため使用されるコンパイラや MPI に応じてコマンドで設定される内容が変化するということがある。Wisteria/BDEC-01 の Odyssey で WaitIO を用いる場合のコマンドは図１のようになる[1]。このときに設定される環境変数は図２に示すとおりである。一方，Aquarius で WaitIO を用いる場合のコマンドと設定される環境変数は図３，図４に示すようになる。WaitIO を利用する際は複数の環境でプログラムをコンパイル・リンクする必要があるため，コンパイル・リンク環境に応じて適切なコマンドを用いなければならない。また，リンクすべきライブラリも環境に応じて複数あるため適切なライブラリを選択する必要がある。具体的には Odyssey では A64FX 用のライブラリである libwaitio_a64fx.so を，Aquarius では用いるコンパイラや MPI が Intel+impi か gnu+ompi かに応じて libwaitio_intel.so もしくは libwaitio.so をリンクするようにする。

```
$module purge
$module load odyssey
$module load waitio
```

図１　Odyssey での WaitIO の module load 手順

```
WAITIO_INC=/work/opt/local/aarch64/apps/fj/1.2.35/fjmpi/1.2.35/waitio/1.0/include
WAITIO_SERVER_HOST=wisteria01
WAITIO_DIR=/work/opt/local/aarch64/apps/fj/1.2.35/fjmpi/1.2.35/waitio/1.0
WAITIO_LIB=/work/opt/local/aarch64/apps/fj/1.2.35/fjmpi/1.2.35/waitio/1.0/lib
```

図２　Odyssey で設定される WaitIO の環境変数

```
$module purge
$module load intel
$module load impi
$module load waitio
```

図３　Aquarius での WaitIO の module load 手順

```
WAITIO_INC=/work/opt/local/x86_64/apps/intel/2021.2.0/impi/2021.2.0/waitio/1.0/include
WAITIO_SERVER_HOST=wisteria01
WAITIO_DIR=/work/opt/local/x86_64/apps/intel/2021.2.0/impi/2021.2.0/waitio/1.0
WAITIO_LIB=/work/opt/local/x86_64/apps/intel/2021.2.0/impi/2021.2.0/waitio/1.0/lib
```

図４　Aquarius で設定される WaitIO の環境変数

[1] なお，コンパイラや MPI 環境が load されていない状態で module load waitio コマンドを実行しても無効となる。

実行に際しても図５のように module load コマンドで適切な WatiIO 環境を設定してやる必要がある。なお図５右図の例では, 実行バイナリを gcc+ompi で生成しているため, intel+impi で WaitIO 環境を設定した後，改めて gcc+ompi を load している。

```
#!/bin/bash
#PJM -L rscgrp=regular-o
省略

module load fj
module load fjmpi
module load waitio

export WAITIO_MASTER_HOST=`hostname`
export WAITIO_MASTER_PORT=7100
export WAITIO_PBID=0
export WAITIO_NPB=2

waitio-serv-a64fx -d -m $WAITIO_MASTER_HOST

mpiexec -np 160 ./nicam
```

```
#!/bin/bash
#PJM -L rscgrp=regular-a
省略

module unload aquarius
module unload gcc
module load intel
module load impi
module load waitio

export WAITIO_MASTER_HOST=`waitio-serv -c`
export WAITIO_MASTER_PORT=7100
export WAITIO_PBID=1
export WAITIO_NPB=2

module unload intel
module unload impi
module load gcc ompi

mpiexec -n 20 ./ada
```

図５　WaitIO を用いた異機種間連成の実行ジョブスクリプト（左図：Odyssey，右図：Aquarius）

2.3 異機種間連成の実現方法

本節では h3-Open-UTIL/MP がどのようにして WaitIO と連携し異機種間連成を実現しているかを説明する。WaitIO は異機種間通信用の基本関数として waitio_isend, waitio_irecv, waitio_wait の３種類を提供している。これらは MPI の同等の関数と同じインターフェースを持ち，容易に相互変換可能である。一方，h3-Open-UTIL/MP はモデル間のデータ交換を mpi_isend, mpi_irecv, mpi_wait の３種の関数のみで実行しているため，これらを WaitIO の上記関数に置き換えるだけで異機種間データ交換に対応できる。一方，データ交換前の初期化プロセスではアプリケーション間の大域通信を行っているため，これを WaitIO で置き換える必要がある。そこで大域通信についてはアプリケーション内の通信を MPI で, アプリケーション間（＝異機種間）の通信を WaitIO で行うようにした。全体のプログラム構造を図６に示す。図で右側の柱は Aquarius 上で Python のアプリケーションを実行する際のプログラム構造である。これについては次章で詳述する。異機種間連成に関わるプログラム構造は図の下部，青系の四角で示した３つの層である。最下層の MPI+WaitIO は大域通信を行うレイヤで，ここでは MPI と WaitIO を組み合わせて通信を実現している。具体的な方法は次に説明する。下から２段目の層は局所通信を行うレイヤである。ここでは, 送受信の相手が別プリケーションの場合は WaitIO を, 同一アプリケーションの場合は MPI を用いて通信を行う。下から３段目の層は MPI と WaitIO を統合し単一の API を提供する Wrapper レイヤである。これによって，h3-Open-UITL/MP 本体のコードを修正することなく，異機種間連成に対応させることが可能となり, 同機種間連成専用プログラムとも容易に切り替え可能となる。

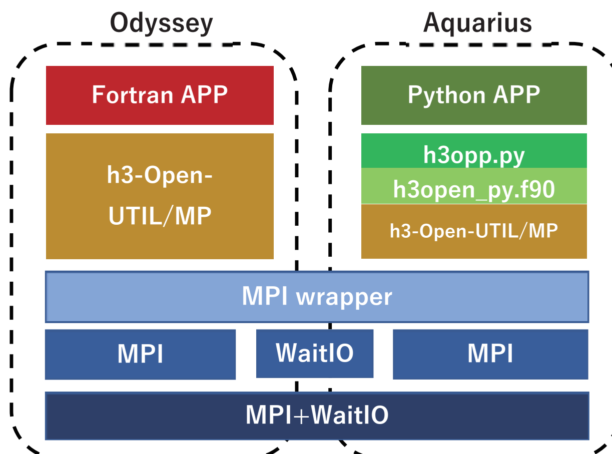

図６　h3-Open-UTIL/MP+WaitIO のプログラム構造

次に上図最下段の MPI+WaitIO がどのようにして実現されているかを説明する。h3-Open-UTIL/MP が用いている大域通信用 MPI ルーチンは MPI_Bcast, MPI_Gather, MPI_Reduce, MPI_Allreduce の４種類に限られており，異機種間連成ではこれらのルーチンを WaitIO の基本通信関数と MPI の大域通信関数を組み合わせて実現すればよいことになる。この際，異機種間の大域通信を waitio_isend/irecv を用いた多対多通信で実装すると並列規模が大きくなった際に対応できなくなる懸念があったため，異機種間の通信は各アプリケーションの代表プロセス間のみの１対１通信で行うようにした。実装された通信方法を MPI_Gether に相当する大域通信を例として図７に示す。図で横に並んだ３つの四角がひとつのアプリケーションを，個々の四角が個々のプロセスを表している。左端の四角は各アプリケーションの代表プロセス(ゼロ番プロセス＝King）である。図の黄色い四角が Gather 先のプロセス(root)とすると，はじめに MPI 関数で King に情報を集める。次いでアプリケーションの King 間を WaitIO で通信し，最後に MPI 関数で対象プロセスにデータを送る。図の例では Gahter 先のプロセスを代表プロセス以外のプロセスとしたが，現実的には Gather 先プロセスは代表プロセスとなることがほとんどであるため最後の手続きは不要になる。

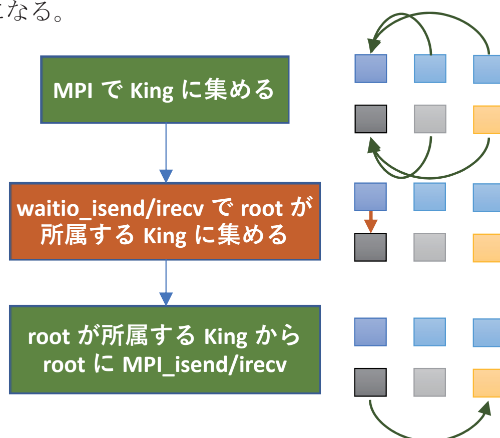

図７　MPI+WaitIO を用いた大域通信の例（Gather）

2.4 異機種間連成の具体例

本節ではサンプルプログラムを参照しつつ，異機種間連成の具体的な手順について説明する。なお単一機種内連成については前号の記事[1]で詳述しているので，ここでは異機種間連成に関連する部分のみを取り上げる。h3-Open-UTIL/MP のライブラリと API は Wisteria/BDEC-01 上で公開されているが，単一機種用と異機種用ではライブラリが異なるので注意が必要である。また異機種用は Odyssey 用と Aquerius 用でも異なるライブラリを用いる必要がある。ライブラリの名称はいずれも同じ（libjcup4.a と libh3ou.a）でディレクトリによって使い分けるようになっている。異機種関連成用のライブラリとインクルードファイルのディレクトリ構成は図８に示すとおりで，include と lib ディレクトリの下に odyssey と aquarius ディレクトリがあり，各々のシステム用のイングルードファイルとライブラリが格納されている。従って，Odyssey 用のバイナリを生成する際は odyssey ディレクトリを，Aquarius 用のバイナリを生成する際は aquarius ディレクトリをそれぞれ参照する。加えて，2.1 節で述べたように適切な WaitIO モジュールを load する。

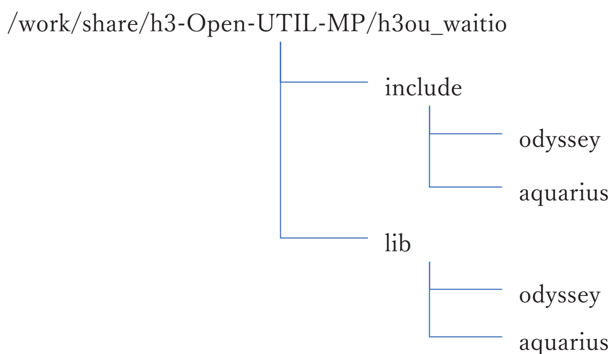

図８　h3-Open-UTIL/MI+WaitIO の格納ディレクトリ

異機種間連成のサンプルプログラムは次章で述べる Python APP 連成のサンプルプログラムとともに/work/share/h3-Open-UTIL-MP/src/sample/h3ou_waitio 以下に置かれている。これを適当なディレクトリのコピーし test_waitio/src に移動して図９に示すように Odyssey と Aquarius のモジュール設定を行った上で make コマンドを実行する。プログラムの実行は test_waitio/run 以下である。go_nicam.sh が Odyssey 用の，go_ada_aquarius.sh が Aquarius 用のジョブスクリプトである。これらを続けてジョブに投入するとプログラムが実行される。実行ディレクトリに h3ou*log.PE*ファイルが出力されファイルの末尾に，

>>>>>>>>>>>>>>>> h3ou_coupling_end IN

<<<<<<<<<<<<<<<< h3ou_coupling_end OUT

のように連成終了サブルーチンのメッセージが出力されていれば実行は成功である。

```
#How to make nicam and ada
cd src
export SYSTEM=odyssey
module purge
module load odyssey
module load waitio
make clean
make nicam

export SYSTEM=aquarius
module purge
module load intel impi
module load waitio
module unload intel impi
module load gcc ompi
make clean
make ada
```

図９　異機種間連成サンプルプログラムのコンパイル手順

3　Python APP 連成

本章では h3-Open-UTIL/MP の拡張機能の一つである Python アプリケーションの連成機能について説明する。はじめに Python API の構造と仕様について説明し，次いで Python アプリケーションを連成するために必要な手順を具体例に基づきつつ説明する。

3.1 Python API の構造と仕様

h3-Open-UTIL/MP のコードは Fortran で記述されており，ネイティブな API は Fortran のモジュール形式で提供されている。Python は C 言語の関数を呼び出す機能を持ち C 言語との連携は容易であるが Fortran のモジュールを直接使用することはできない。しかしながら Fortran2003 以降では C 言語の互換インターフェースを提供する仕組みが標準装備されているため，この機能を使うことで Python から C 言語の関数として Fortran のサブルーチン（や関数）をコールする事ができる。具体的な方法を h3-Open-UTIL/MP の初期化サブルーチンを例として説明する。h3-Open-UTIL/MP の初期化サブルーチン h3ou_init はモデル名と設定ファイル名（および optional 引数としてアンサンブル数）を与える仕様になっている。このサブルーチンを Python から呼び出せるように，モジュールに含まれない Fortran サブルーチンを定義する。初期化サブルーチンの例を図 10 に示す。C 言語は文字列の扱いが Fortran と異なっているため文字列を渡す際は先頭位置と長さの２つの引数を渡すことになる。この２つの値から関数 get_char_str で Fortran の文字列に変換し，h3-Open-UTIL/MP のネイティブ API である h3ou_init をコールする。このサブルーチンには bind(C)属性が付加され，これによって Python からは C 言語の関数として呼び出すことが可能となる。初期化を含む各種 API ルーチンは図６の h3open_py.f90 に格納されている。次に呼び出し側である Python の関数を図 11 に示す。はじめに argtypes でインターフェースを定義し，ついで restypes で関数の返値を定義する。ここでは返値なし（c_void_p）が与えられている。渡されてきた引数は C 言語のポインタ型に変換され図 10 に示された Fortran サブルーチンに渡されるとともに，文字列の長さもポインタ引数として渡される。

```
subroutine h3oup_init(comp_name, comp_name_len, config_name, &
                                                          config_name_len) bind(C)
  use h3ou_py_base, only : get_char_str
  use h3ou_api, only : h3ou_init
  implicit none
  character(len=1)        :: comp_name(*)
  integer, intent(IN) :: comp_name_len
  character(len=1)        :: config_name(*)
  integer, intent(IN) :: config_name_len

  call h3ou_init(trim(get_char_str(comp_name, comp_name_len)), &
                        trim(get_char_str(config_name, config_name_len)))

end subroutine h3oup_init
```

図 10　PythonAPP 連成用 Fortran API の内容例

```
def h3ou_init(comp_name, config_file_name):

    h3oupf.h3oup_init.argtypes = [
            ctypes.POINTER(ctypes.c_char),
            ctypes.POINTER(ctypes.c_int32),
            ctypes.POINTER(ctypes.c_char),
            ctypes.POINTER(ctypes.c_int32)
            ]
    h3oupf.h3oup_init.restype = ctypes.c_void_p

    comp          = ctypes.create_string_buffer(comp_name.encode("utf-8"))
    comp_len    = len(comp_name)
    config         = ctypes.create_string_buffer(config_file_name.encode("utf-8"))
    config_len   = len(config_file_name)


    h3oupf.h3oup_init(comp, ctypes.byref(ctypes.c_int32(comp_len)), ¥
                            config, ctypes.byref(ctypes.c_int32(config_len)))
```

図 11　APython APP 連成用 API の内容例

3.2 Python APP 実行のための環境構築

Python の利点は，豊富なライブラリ群が全世界の開発者から提供されており，これらのライブラリを有効利用することで高度な機能を簡単に実現することができる，という点にある。Python アプリケーションを連成する際も目的に応じて適切なライブラリをインストールして利用することになるが，その際，システム環境にはインストールできずユーザー領域に環境を構築することになる。そのための手順は Wisteria/BDEC-01 システム利用手引き書，最新版の手引き書（Version 1.1.9）では 192 ページ「9.2.5.追加パッケージのインストール方法」に記載されている。この手順に従って/work/groupname/username/packagename にインストールされたライブラリ群はジョブスクリプト内で，

source /work/groupname/username/packagename/bin/activate

と記述することでジョブに投入された Python プログラムから参照可能となる。

3.3 Python プログラムの記述方法

h3-Open-UTIL/MP の Python API コードは/work/h3-Open-UTIL/MP/h3ou_waitio/include/aquarius にファイル名 h3ou.py で格納されている。従って，この API を用いるには図 12 に示すように当該ディレクトリにパスを設定した上で h3ou を import する。インポートしたライブラリに対して h3opp.h3ou_init(comp_name, config_name)のように API をコールする事で h3-Open-UTIL/MP を Python から利用することができる。

```
import os
import sys
sys.path.append(os.path.join(os.path.dirname(__file__), ¥
                    "/work/share/h3-Open-UTIL-MP/h3ou_waitio/include/aquarius"))

import h3ou as h3opp

省略

print("H3Open Adaptive Data Analyzer initialization start, comp name = ", comp_name)

h3opp.h3ou_init(comp_name, config_name)
```

図 12 Python API の利用例

3.4 Python APP 連成の具体例

Python APP 連成のサンプルプログラムは次章で述べる異機種間連成のサンプルプログラムとともに/work/share/h3-Open-UTIL-MP/src/sample/h3ou_waitio 以下に置かれている。これを適当なディレクトリのコピーし test_python/src に移動して図 9 の前半に例示された Odyssey のモジュール設定を行った上で make コマンドを実行する。Python APP については当然ながらコンパイル・リンクの必要がないため make コマンドの対象は Odyssey 側のプログラムのみとなる。プログラムの実行は test_python/run 以下である。go_nicam.sh が Odyssey 用の，go_ada_aquarius.sh が Aquarius 用のジョブスクリプトである。これらを続けてジョブに投入するとプログラムが実行される。実行ディレクトリに h3ou*log.PE*ファイルが出力されファイルの末尾に，

```
>>>>>>>>>>>>>>>> h3ou_coupling_end IN
<<<<<<<<<<<<<<<< h3ou_coupling_end OUT
```

のように連成終了サブルーチンのメッセージが出力されていれば実行は成功である。

4 アンサンブル連成機能

4.1 アンサンブル連成の概要

アンサンブル計算とは入力データやパラメータなどの数値をわずかに変えた多数のシミュレーションを実行し，結果の統計的な描像を得る手法である。google で「アンサンブル計算」や「アンサンブルシミュレーション」で検索するとヒットするのはほぼ気象・気候シミュレーションに関連するページで占められる。このことから推測できるように，アンサンブル計算は当

該分野では一般的な計算手法である。気象現象は非線形性が強く，バタフライエフェクトという言葉に象徴されるように初期値などのわずかな違いが結果に大きな相違をもたらす。従って個々のシミュレーションが不可避的に内在する不確実性を低減し，あるいは不確実性の程度を定量的に評価するためにもアンサンブル計算は不可欠の手法である。h3-Open-UTIL/MP は連成されたモデルのペアを複数起動し，同時に多数の連成計算を実行することができる。もちろんこれだけなら複数のジョブを同時に投入することでも実現可能であるが，h3-Open-UTIL/MP は実行中の連成モデル群に対して，連成間通信を行う機能を有している。通信の模式図を図 13 に示す。図のように h3-Open-UTIL/MP のアンサンブル連成は 2 通りの実行方法をサポートしている。ひとつは左図のように，連成されたペア（例えば大気海洋結合モデル）を複数起動する方法であり，もう一つは右図のように複数のモデル A に対してひとつのモデル B を連成するような方法である。後者の具体的な適用例を図 14 に示す。ここでは低解像度大気モデルのアンサンブルと高解像度大気モデルが連成されている。この手法により，高解像度モデルのアンサンブル計算に比してより少ない計算資源で，確度の高い高解像度計算が実現できる。

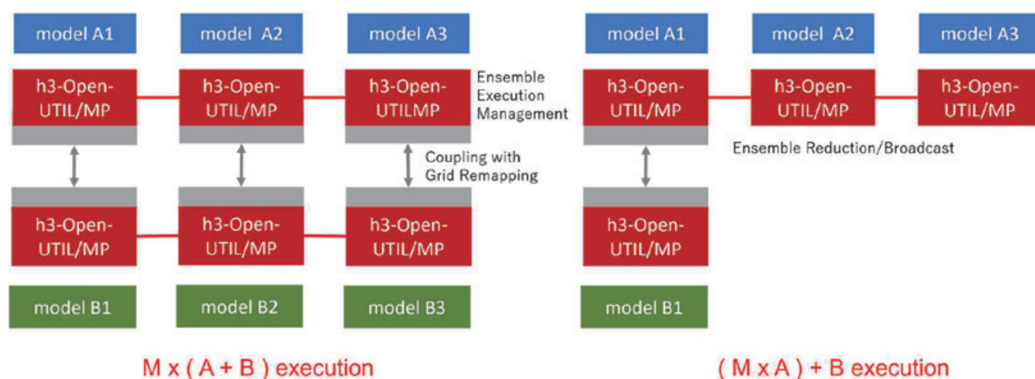

図 13　アンサンブル連成の実行パタン

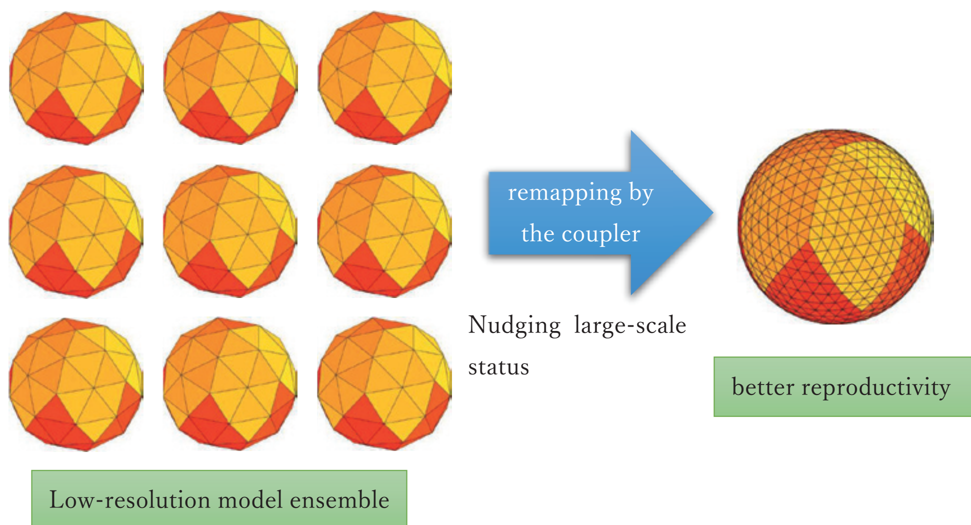

図 14　多対一アンサンブル連成の適用例

4.2 アンサンブル連成の方法

本節では h3-Open-UTIL/MP においてアンサンブル連成がどのように実現されているかを解説する。説明の便のために，ここでアンサンブル方向と連成方向という用語を導入する。アンサンブル方向とは図 13 では横方向になる。アンサンブル方向ではアンサンブル対象となる各モデルコンポーネントは当然ながら同一のモデルであり，更に同じ時空間設定を持つ，すなわち同一の格子，同一の領域分割，同一の積分時間ステップ等を持つものとする。一方連成方向は図 13 では縦方向となり，対象となるモデルコンポーネントは相互に異なる時空間構造を持つ異なるモデル（同一モデルも含む）であることが前提となる。アンサンブル計算におけるアンサンブル数は目的や物理的条件，あるいは利用可能な計算資源量などで異なるが，大規模な計算になると 1000 を越える[4]。このアンサンブル数を MPI で起動する際に mpiexec -np N ensemble1 : -np N ensemble2 : -np N ensemble3 のようにひとつずつ記述していくのは現実的ではない。そこで h3-Open-UTIL/MP では mpiexec -np MxNA ModelA : -np MxNB ModelB のように各モデルで実行される全プロセス数を mpi への引数とする。ここで M はアンサンブル数，NA は ModelA における個々のアンサンブルのプロセス数，NB は ModelB での個々のアンサンブルのプロセス数である。アンサンブル数 M は h3-Open-UTIL/MP の初期化 API である h3ou_init で引数として与える。h3ou_init はデフォルトでは引数としてモデル名称と設定ファイル名の２つを与えるが，３番目に optional 引数としてアンサンブル数(num_of_ensemble)を与えることができるようになっている。２つのモデルで num_of_ensemble は同数かもしくは片方が１の場合のみが許され，同数の場合は図 13 左図の形式，片方が１の場合は図 13 右図の形式と判定される。また各モデルの全プロセス数はアンサンブル数で割り切れることが条件である。h3ou_init の引数として与えられるモデルの名称および MPI 環境から得られる全プロセス数から，各モデルが合計いくつのプロセスで実行されるかが求められ，この値とアンサンブル数 M からアンサンブル毎のプロセス数 NA,NB も求められる。引数として与えるモデル名はカプラ内部ではモデル識別コードとして扱われ，実際のモデル名は ModelA0001，ModelA0002 のようにアンサンブル番号が付加された文字列となる。これが個々のモデル名としてログファイル名などに用いられるようになっている。個々のプロセスが何番のアンサンブルに所属しているかは h3-Open-UTIL/MP の API を通じて取得できるので，アンサンブル毎にデータやパラメータを変える際に，所属番号に応じてデータのディレクトリを変える，あるいは設定ファイル名を変えるなどの対応が可能となる。アンサンブル連成に際して h3-Open-UTIL/MP が生成し使用している MPI コミュニケータの種類を図 15 に示す。最外周の黒線は MPI のデフォルトコミュニケータ MPI_COMM_WORLD である。各モデルプロセス全体を囲む青線がアンサンブル方向（モデル毎）のグローバル通信を行うコミュニケータである。個々のモデルを囲む緑線が，各モデルコンポーネントが通信（例えば袖領域の交換）に用いるローカルコミュニケータである。連成方向に２つのモデルコンポーネントを囲む黄色線が個々のアンサンブルで連成に用いられるコミュニケータである。最後に，アンサンブル方向でモデル内の同一プロセス同士を繋いでいる赤線がアンサンブル方向のプロセス間通信を行うコミュニケータである。これらのコミュニケータの内，利用者が参照することができ，かつ参照する必要があるのはローカルコミュニケータだけであり，残りのコミュニケータはすべてカプラ内部で使用される。

アンサンブル連成では，個々の連成計算の途中でアンサンブル方向の統計処理を行いその結果を連成計算に反映させる，といった計算方法が想定されている。現在実装されている統計処理は平均値の計算のみであるが，これについては必要に応じて今後追加される予定である。

: global = MPI_COMM_WORD
: ecomp = same component name : アンサンブル方向の通信に用いる
: local = local model process : 各モデルコンポーネントが使う
: couple = coupled processes : Jcupにthe worldとして与えられる
: glocal = same rank of the component : アンサンブル方向の通信に用いる

図 15　アンサンブル連成に用いられるコミュニケータ

4.3 アンサンブル連成の具体例

アンサンブル連成のサンプルプログラムは Wisteria/BEDEC-01 の/work/share/h3-Open-UTIL-MP/h3ou/src/sample/h3ou 以下にある。h3ou ディレクトリには Mkinlcude ファイルと sysdep, test_simple, test_ensemble ディレクトリが置かれており，test_ensemble がアンサンブル連成のテスト環境になっている。h3ou ディレクトリ以下を適当なディレクトリにコピーし, test_ensemblle/src ディレクトリで make コマンドを実行するとプログラムがコンパイル・リンクされる。test_ensemble/run ディレクトリで pjsub. go.sh によりジョブに投入するとプログラムが実行されログファイル h3ou.app*log*が出力される。ログファイルの末尾に

```
>>>>>>>>>>>>>>>> h3ou_coupling_end IN
<<<<<<<<<<<<<<<< h3ou_coupling_end OUT
```

のように連成終了サブルーチンのメッセージが出力されていれば実行は成功となる。アンサンブル連成の設定ファイルの形式は[1]で解説した通常の連成の場合と同じで，実行方法も通常連成と変わらない。唯一の相違は前節で説明したように，初期化 API である h3ou_init でオプショナル引数としてアンサンブル数を渡すことである。アンサンブル数が渡されることで h3-Open-UTIL/MP はこの実行がアンサンブル連成であると判断しコミュニケータの生成やモデル名の生成などの処理を行う。サンプルプログラムでは図 16 に示すように，プログラムの引数としてア

ンサンブル数を取得して h3ou_init の引数として渡している。アンサンブルの情報はログファイルの冒頭付近に出力される。サンプル実行のログを図 16 に示す。サンプル実行は多対一アンサンブルであるため 0 番アンサンブルとそれ以外で動作が異なる。図 16 上図は 0 番アンサンブルの出力，下図は 0 番以外の出力である。0 番は相手モデルとの結合フラグ(coupling flag)が T になっているが，その他のアンサンブルは結合フラグが F となっており直接連成されていないことがわかる。

```
subroutine init_common(comp_name, comp_id)
  use h3ou_api, only : h3ou_init, h3ou_get_mpi_parameter
  implicit none
  character(len=*), intent(IN) :: comp_name
  integer, intent(IN)          :: comp_id
  integer, parameter :: MAX_GRID = 8
  integer :: ierror
  integer :: num_of_arg
  character(len=128) :: arg_ensemble
  integer :: num_of_ensemble
  intrinsic :: command_argument_count, get_command_argument

  my_name = trim(comp_name)
  my_comp_id = comp_id

  num_of_arg = command_argument_count()

  if (num_of_arg == 0) then
     num_of_ensemble = 0
  else
    call get_command_argument(1, arg_ensemble)
    read(trim(arg_ensemble), *) num_of_ensemble
  end if

  if (num_of_ensemble >= 1) then
     call h3ou_init(my_name, "coupling.conf", num_of_ensemble)
  else
     call h3ou_init(my_name, "coupling.conf")
  end if
```

図 16　アンサンブル連成の初期化プログラム

```
>>>>>>>>>>>>>>> h3ou_init IN
 -------------- ensemble information
 Ensemble coupling : my name =     app10000, target name =     app20000
 ensemble type = 2, coupling flag =   T
```

```
>>>>>>>>>>>>>>> h3ou_init IN
 -------------- ensemble information
 Ensemble coupling : my name =     app10001, target name =   NO_TARGET
 ensemble type = 2, coupling flag =   F
```

図 17　アンサンブル連成のログ（上図：0 番アンサンブル，下図：0 番以外のアンサンブル）

5　まとめ

本稿では Wisteria/BDEC-01 利用事例として汎用連成ソフトウェア h3-Open-UTIL/MP の拡張機能について解説した。h3-Open-SYS/WaitIO との協調機能を用いることで Odyssey と Aquarius の２つのシステム間での連成計算が可能となり，更に Python インターフェースを供用することによってシミュレーションプログラムと機械学習プログラムなど従来はファイルを経由して逐次的に実行されていた処理をリアルタイムに並列処理することができるようになる。本稿では Toy モデルを用いた簡単な計算事例を紹介したが，これらの機能を実アプリケーションに適用した計算も試みられている[5]。今後も機能拡張やユーザインターフェースの改良といったカプラ本体の開発を継続するとともに，実アプリケーションへの適用も推進してゆく予定である。

参考文献

[1] 荒川隆，八代尚，中島研吾, Wisteria/BDEC-01 利用事例(5) マルチプログラム連成ライブラリ h3-Open-UTIL/MP(1/2)，スーパーコンピューティングニュース Vol.24 No3, https://www.cc.u-tokyo.ac.jp/public/VOL24/No3/13_202205-Wisteria-2.pdf

[2] 住元真司，坂口吉生，松葉浩也，中島研吾, Wisteria/BDEC-01 利用事例(3) データ受け渡しライブラリ h3-Open-SYS/WaitIO(1/2)，スーパーコンピューティングニュース Vol.24 No.2, https://www.cc.u-tokyo.ac.jp/public/VOL24/No2/10_202203Wisteria-1.pdf

[3] 住元真司，坂口吉生，松葉浩也，中島研吾, Wisteria/BDEC-01 利用事例(4) データ受け渡しライブラリ h3-Open-SYS/WaitIO(2/2)，スーパーコンピューティングニュース Vol.24 No.3, https://www.cc.u-tokyo.ac.jp/public/VOL24/No3/12_202205-Wisteria-1.pdf

[4] H. Yashiro, H., Terasaki, K., Kawai, Y., Kudo, S., Miyoshi, T., Imamura, T., Minami, K., Inoue, H., Nishiki, T., Saji, T., Satoh, M., and Tomita, H.: "A 1024- Member Ensemble Data Assimilation with 3.5-Km Mesh Global Weather Simulations," in SC20: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), Atlanta, GA, US, 2020 pp. 1-10. doi: 10.1109/SC41405.2020.00005

[5] Arakawa, T., Yashiro, H., Nakajima, K., Development of a coupler h3-Open-UTIL/MP, ACM Proceedings of the International Conference on High Performance Computing in Asia-Pacific Region(HPC Asia 2022), 2022, https://doi.org/10.1145/3492805.3492809