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)

住元真司・坂口吉生（富士通株式会社）
松葉浩也・中島研吾（東京大学情報基盤センター）

1　はじめに

今回と次回（2022 年 5 月号）、次々回（2022 年 7 月号）の 3 回に分けて，Wisteria/BDEC-01［1,2,3］のようなヘテロジニアスな環境で「計算・データ・学習」融合を実現するためのツールである，h3-Open-SYS/WaitIO［4］と h3-Open-UTIL/MP［5］について紹介する。今回と次回については h3-Open-SYS/WaitIO と h3-Open-UTIL/MP の基本知識と協調動作の事例を中心に紹介し、次々回では応用例として h3-Open-UTIL/MP を活用した GP-GPU 連携を含めた多機能カップラーとして「計算・データ・学習」融合の事例も紹介する。

2　Wisteria/BDEC-01

東京大学情報基盤センターで 2021 年 5 月 14 日に運用を開始した「Wisteria/BDEC-01」〔1,2,3〕は，「シミュレーション（Simulation）・データ（Data）・学習（Learning）（S+D+L）」融合を目指す BDEC システム構想（Big Data & Extreme Computing）に基づくシステムの第 1 号機であり，シミュレーションノード群（Odyssey（オデッセイ））とデータ・学習ノード群（Aquarius（アクエリアス））の 2 つの計算ノード群を有したシステムである（図 1）。

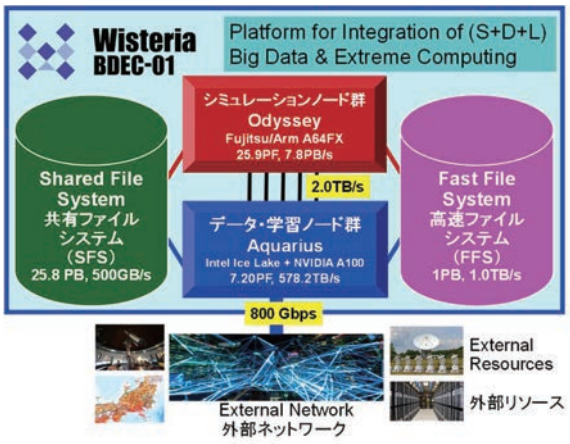

図1　Wisteria/BDEC-01の概要〔1,2,3〕

シミュレーションノード群（Odyssey）は「FUJITSU Supercomputer PRIMEHPC FX1000」20 ラックから構成され，「A64FX」を 7,680 ノード（368,640 コア）搭載する。各ノードは倍精度浮動小数点演算で 3.3792 TFLOPS の理論ピーク性能を実現する。合計ピーク性能は 25.9 PFLOPS であり，各ノードは 32 GiB の HBM2 メモリを搭載し，シミュレーションノード群（Odyssey）の総メモリ容量は 240 TiB，総メモリバンド幅は 7.8 PB／秒である。各ノードはバイセクションバンド幅が 13.0 TB／秒のノード間相互結合ネットワーク（Tofu インターコネクト D）で結合されている。

データ・学習ノード群（Aquarius）各ノードは汎用 CPU 2 基（Intel Xeon Platinum 8360Y（Ice Lake），36core, 2.4GHz)），演算加速装置（GPU）8 基（NVIDIA A100 Tensor コア（SXM4，40GB））から構成されており，ノード間インターコネクトには NVIDIA Mellanox HDR InfiniBand ネットワークが採用されている。データ・学習ノード群（Aquarius）の合計ピーク性能は 7.2 PFLOPS，総メモリ容量は 36.5 TiB，総メモリバンド幅は 578.2 TB／秒である。各ノードは，データ転送速度が 200 Gbps の帯域を有する InfiniBand HDR を 4 リンク用いて，フルバイセクションバンド幅を持つノード間相互結合ネットワークで結合されている。さらに，外部接続のために 25 Gbps Ethernet インタフェースも有している。

FEFS（Fujitsu Exabyte File System）による，共有ファイルシステム（容量：25.8 PB，データ転送速度：0.504 TB／秒）および SSD を搭載した高速ファイルシステム（容量：1.0 PB，データ転送速度：1.00 TB／秒）を有し，それぞれシミュレーションノード群（Odyssey），データ・学習ノード群（Aquarius）からアクセスし，大規模なデータを高速に処理することが可能である。

シミュレーションノード群（Odyssey）とデータ・学習ノード群（Aquarius）は，合計 160 本の InfiniBand EDR（100Gbps）を用いて 2.0 TB／秒のネットワークバンド幅で結合されている。また，データ・学習ノード群（Aquarius）は合計 800 Gbps のネットワーク転送速度で外部との通信が可能である。Aquarius の一部のノードは SINET 等の外部ネットワークを介して，サーバー，ストレージ，センサーネットワークを含む様々な外部リソースに直接アクセス可能であり，観測データをリアルタイムに取り込んで解析，シミュレーションに利用することも可能である。現在は，Odyssey と Aquarius は別々に運用されている。

3　h3-Open-BDEC

3.1 概要

「計算・データ・学習（S+D+L）」融合のためには，Wisteria/BDEC-01 のようなこれまでにない革新的なハードウェアが必要であるが，様々なアプリケーション，ワークロードを Wisteria/BDEC-01 上で開発，実行していくためのソフトウェア群も重要である。

当センターで開発した「ppOpen-HPC（自動チューニング機構を有するアプリケーション開発・実行環境）」〔6〕，「h3-Open-BDEC（「計算・データ・学習」融合のための革新的ソフトウェア基盤）」〔7,8〕を利用し，高性能なアプリケーションを容易に開発することが可能である。

当センターでは，センター内外の計算科学，計算機科学，数値アルゴリズム，データ科学，機械学習の専門家と協力して，エクサスケール時代のスパコンの能力を最大限活用し，科学的発見を持続的に促進するために，（計算・データ・学習）融合による革新的シミュレーション手法を提案し，最小限の計算量・消費電力で融合シミュレーションを実現する研究開発，ソフトウェア基盤実装を実施している〔7,8〕。本研究では，Wistria/BDEC-01 を（計算・データ・学習）融合のためのプラットフォームと位置付け，①変動精度演算・精度保証・自動チューニング（Automatic Tuning，AT）による新計算原理に基づく革新的高性能・高信頼性・省電力数値解法，②機械学習に基づく革新的手法である階層型データ駆動アプローチ（hDDA）の 2 項目を中心に研究開発を実施し，革新的ソフトウェア基盤「h3-Open-BDEC」を開発する。（図 2）〔7,8〕

革新的ソフトウェア基盤「h3-Open-BDEC」（図 2）を構成する h3-Open-MATH（数値アルゴリズム），h3-Open-VER（精度保証），h3-Open-AT（自動チューニング），h3-Open-APP（アプリ開発），h3-Open-DATA（データ科学），h3-Open-DDA（データ駆動アプローチ），h3-Open-UTIL（並列ユ

ーティリティ），h3-Open-SYS（統合・制御）は複数の構成要素を含み，緊密に関連し，以下の 3 層を構成している：

①「変動精度演算に基づく新計算原理」層（h3-Open-MATH，h3-Open-VER，h3-Open-AT）
②「(計算・データ・学習）融合」層（h3-Open-APP，h3-Open-DATA，h3-Open-DDA）
③「統合・通信・ユーティリティ」層（h3-Open-SYS，h3-Open-UTIL）

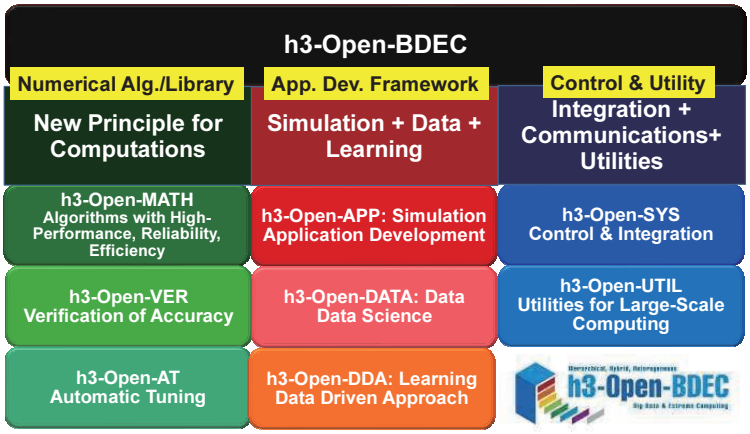

図 2　h3-Open-BDEC の概要〔7,8〕

エクサスケールシステムにおける高性能数値アルゴリズム実現には，メモリ・ネットワークの階層の深化に対応した通信最適化（Serial，Parallel），省電力・省エネルギに向けた検討が必要である。数値計算による近似解（数値解）は様々な計算誤差を含み，計算結果の信頼性の観点から，数値解の正しさを数学的に保証する必要があり，低精度・混合／変動精度使用時，悪条件問題には重要であるが，実問題で現れる大規模疎行列・H 行列への応用例はほとんどない。本研究では，「変動精度演算に基づく新計算原理」確立を目指して，高性能アルゴリズム，精度保証，最適精度選択のための自動チューニング手法の研究開発を実施する。

シミュレーションに機械学習を適用して異なるパラメータでの解を予測するデータ駆動アプローチ（Data Driven Approach，DDA）では，計算を繰り返して教師データを生成する必要がある。本研究で提案する階層型 DDA（hDDA）は，特徴検知，MOR（Model Order Reduction），UQ（Uncertainty Quantification），スパースモデリング，適応格子等の諸機能を駆使して，計算量（メッシュ数，粒子数）を削減した簡易モデルを，機械学習により自動生成し，教師データ生成用モデルとして利用する。「統合・通信・ユーティリティ」層は Wisteria/BDEC-01 のようなヘテロジニアスなシステム上で，「計算・データ・学習」融合を容易に実現するための環境を提供する。

h3-Open-BDEC はエクサスケール時代のスパコンで（計算・データ・学習）融合を実現する世界初の革新的ソフトウェア基盤であり，計算科学の専門家のみで（計算・データ・学習）融合を容易に実現できる。ソースコード，マニュアル類も含めて一般に公開し，様々な環境で利用できるよう，普及に努める。h3-Open-BDEC 利用による（計算・データ・学習）融合シミュレーションにより，従来手法と同等の正確さを保ちつつ，大幅な計算量・消費電力削減を目指す。

計算科学シミュレーションは多くの場合，非線形な問題を扱うため，多数のパラメータスタディが必要である。Wisteria/BDEC-01 では，機械学習による最適パラメータ推定を，外部から取り

込んだ実験・観測データによる同化と組み合わせて，正確な解をより短時間で求めることを目指している。

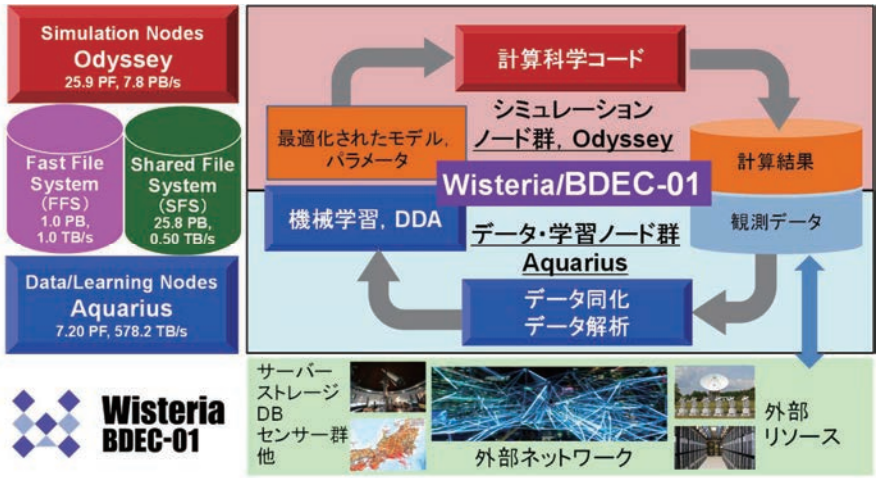

図3　Wisteria/BDEC-01利用による「計算・データ・学習」融合のイメージ〔7,8〕

図 3 は，Wisteria/BDEC-01 における「計算・データ・学習（S+D+L)」融合のイメージである。h3-Open-BDEC を使用することによって，シミュレーションノード群で計算科学シミュレーションコードを実行し，データ・学習ノード群では外部から取り込んだ観測データや，機械学習による推論等に基づきパラメータを最適化し，更に計算を実施するというサイクルを容易に実現することができ，またパラメータ最適化によって計算時間を全体として短縮できることが期待される。

3.2 h3-Open-SYS

h3-Open-SYS は Wisteria/BDEC-01 のようなヘテロジニアスなシステムにおいて，シミュレーションとデータ処理実行を統合するソフトウェア群である。h3-Open-SYS/WaitIO〔4,7,8〕はその中核機能として，ファイルシステムを通じて複数の並列プログラムがデータの受け渡しを行うライブラリである。多くのスパコンで提供されている共有ファイルシステムをプログラム間のデータ連携手段として使用することで高い汎用性を確保し，ファイルを通信手段として用いる際に一般的に問題となる同期の問題を WaitIO ライブラリで解決する。WaitIO ライブラリは元々ファイルシステム経由で通信を前提としていたが，複数の並列プログラム間での通信を実施するために，Odyssey と Aquarius 間の通信ライブラリとしても開発が進められている［4］。h3-Open-SYS/WaitIO は，h3-Open-BDEC の多機能カプラ（Coupler）である h3-Open-UTIL/MP［5］と組み合わせることによって，Wisteria/BDEC-01 のようなヘテロジニアスなシステム上で「S+D+L」融合を実現できる。h3-Open-SYS/WaitIO と h3-Open-UTIL/MP の使用によって，Odyssey と Aquarius を連携し，「計算・データ・学習」融合の実現が始まっており，観測データ同化による長周期地震動リアルタイム予測シミュレーション等へ適用されている［9,10,11］。

4　h3-Open-SYS/WaitIO

h3-Open-SYS/WaitIO(以下 WaitIO)は Odyssey と Aquarius を連携し「S+D+L」融合を実現するため

に開発された２つ以上の並列プログラム間で通信を行う通信ライブラリである。各並列プログラムは典型的には MPI プログラムであるが，MPI 以外の形の並列プログラムからも利用できるよう汎用的に設計されている。

WaitIO は多様な並列計算機環境を想定している。特に，通信を行う並列プログラムが同一の並列計算機で動作する場合はもちろん，前述の Odyssey と Aquarius のように各々の並列プログラム異なる並列計算機で動作する場合にも対応する。そのために通信経路としてファイルと TCP/IP を想定する。

4.1 WaitIO-File

ファイル経由での通信を行う WaitIO を特に WaitIO-File（以下，簡単に WaitIO-File）と呼ぶ。WaitIO-File は同時実行される並列プログラムの間の通信手段が共有ファイルシステムに限られる状況を想定し，共有ファイルを通信路として使用するものである。一般にスーパーコンピュータは，並列プロセスに対して割り当てたノード群から外への通信を許さない運用が多いため，共有ファイルを用いたデータの受け渡しは非効率であるものの現実的な手段である。WaitIO-File は通信を行うプロセスのペアそれぞれに対して通信方向別に 1 個ずつのファイルを準備する。送信側は送信データをファイル末尾に追記し続け，受信側はそのファイルを読み込む動作が基本であるが，特に受信側はファイルの末尾以降を読み込もうとした場合には新たなデータの追記を待つ点が通常のファイル I/O と異なる（WaitIO はこの特徴を表した名前である）。このファイルの追記を効率的に待つ手法，また，送受信が完了したデータを削除する方法について，ファイルシステムに依存した細かな最適化が必要であり現在検討を進めている。

4.2 WaitIO-Socket

TCP/IP を用いて通信を行う WaitIO を特に WaitIO-Socket と呼ぶ。WaitIO-Socket は並列プログラムを構成する各プロセスが，別の並列プログラムの各プロセスに直接通信できる環境を想定している。前述のように現在のスーパーコンピュータの通常の運用ではこのような通信は許されていないことが多いが，並列計算機を構成するインターコネクトは TCP/IP での通信をサポートしていることが多いため，セキュリティー上の問題を考慮の上，運用方針を明確にすれば技術的には容易に実現可能な環境と言える。WaitIO-Socket の基本アイディアは，並列プログラムを構成する各プロセスが通信相手の並列プログラムの全プロセスに対して TCP/IP のコネクションを張り通信を行うものである。この基本アイディアはシンプルであるが，各並列プロセスが 1 万を超える数のプロセスから構成されるような大規模環境を想定する際に，通信ソケットだけでメモリ資源を使い尽くすようなことがないよう，設計，実装上の工夫が必須である。

本稿では，WaitIO-Socket についてその動作と使い方を含めて詳細に説明する。

5　WaitIO の動作仕様

WaitIO は複数の MPI アプリケーション間で通信を行うライブラリであり既存プログラムに WaitIO によるデータ通信を追加することにより複数アプリケーション間のデータ交換を実現するものである。図 4 に WaitIO の動作イメージを示す。

各アプリケーションは Parallel Block (以降，PB)と呼ばれる複数のプロセスグループで管理される。1 PB は MPI の場合一回の mpiexec コマンド実行に対応する。

複数の PB から構成される WaitIO 全体は WaitIO Instance と呼ばれ，各 PB は PBID と呼ばれる識別子で管理されている。PBID=0 の PB を MASTER PB と呼ばれ，プログラムの起動時に初期

化され，構成される PB 間で同期して各プロセス情報を交換するサーバーとして機能する。初期化に必要な情報はプログラム実行時のシェルの環境変数で定義され，WAITIO_MASTER_HOST, WAITIO_MASTER_PORT, WAITIO_PBID, WAITIO_NPB (構成される PB 数)を各プログラム実行時に設定する。

WaitIO を用いて通信を行うには WaitIO Group(WG)を作成する必要がある。WG を構成するメンバープロセスは開発者が任意に選択することが可能である。WG は MPI の Communicator と同じ概念である。WG が作成されると各メンバープロセスには MPI と同様な概念のランク番号が与えられ，このランク番号を宛先に指定した通信が実行可能になる。

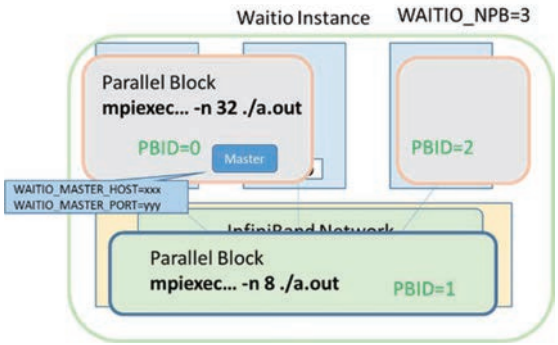

図 4　WaitIO の動作イメージ

5.1 実装 API の概要

表 1 に WaitIO の API を示す。初期化とサポート関数以外は 基本通信関数 Non-Blocking Send/Recv と Wait 関数だけである。

表 1. WaitIO の API

WatiIO API	概要
waitio_isend	Non-Blocking送信
waitio_irecv	Non-Blocking受信
waitio_wait	送受信完了待ち合わせ
waitio_init	WaitIO初期化
waitio_get_nprocs	PB毎の参加プロセス数獲得
waitio_create_group waitio_create_group_wranks	PB間通信グループ生成（メンバ配列指定、関数指定）
waitio_group_rank	グループ内Rankの獲得
waitio_group_size	グループサイズの獲得
waitio_pb_size	全PBのサイズ獲得
waitio_pb_rank	全PB内Rankの獲得

各関数のうち特徴的な関数について説明する。

(1)　初期化終了 API: int waitio_init(int timeout);

- 初期化を実施する。MPI における MPI_Init()と同様に全プロセスが必ず最初に呼ぶ必要がある。本関数の実行によりマスタープロセスが立ち上がり，全員が参加するまで timeout 秒待つ。

(2)　各 PB を構成するプロセス数獲得 API：int waitio_get_nprocs(int ary[]);

- 返り値は WAITIO Instance に含まれる並列プロセス数(環境変数 WAITIO_NPB で設定した数)
- ary に各 PB に含まれるプロセス数が返る

(3) 初期化終了 API ：int waitio_finalize();
- 終了処理を実施する。実行完了後，再び waitio_init()を呼び出すことはできない。

(4) グループ作成 API：waitio_group_t waitio_create_group(ing gid, waitio_filter_func_t[], int order[]);
- 本関数は構成される各 PB に対して PB 内プロセスを選別する関数(waitio_filter_func_t)を与え選別した結果を order の順に並べて WG を作成する関数である。

(5) 通信 API：int waitio_isend(waitio_group_t group, int dst, char *buf, size_t len,
unsigned long tag, waitio_req_t *req);
- group 内のランク dst に送信する。

(6) 通信 API：int waitio_irecv(waitio_group_t group, int src, char *buf, size_t len,
unsigned long tag, waitio_req_t *req);
- group 内のランク src から受信する。ただし MPI の持つ MPI_ANY_SOURCE, MPI_ANY_TAG の概念はない。

(7) 通信 API：int waitio_wait(waitio_req_t *req);
- req で指定した isend, irecv 処理の完了を待つ

6 WaitIO プログラミング

図 5 に WaitIO の API を用いた PingPong プログラムを示す。このプログラムは 0 から始まる WaitIO のランク番号が偶数の場合は送信後に受信，奇数の場合は受信後に送信する。

図 5 において左半分は初期化部分である。13 行目の MPI の初期化の後 17 行目で WaitIO の初期化関数を呼んでいる。17 行目において引数の-1 は無限大の待ち時間を示す。

22 行目に WaitIO の複数 MPI プログラムを結合するプロセスグループを生成する関数 waitio_create_group()の引数は 8 行目と 9 行目で定義されている。8 行目は 2 つまでの PB 結合をする際のプロセスを選択する関数を指定しており，ture()関数が指定されている。これは PB を構成するすべてのプロセスが生成されるグループのメンバになることを示している。9 行目は PB 間を結合した場合のランク番号の順序を指定する配列で，PB 番号 0 番の後 PB 番号 1 番の PB の順にランク番号が割り当てられる。

22 行目で生成されたプロセスグループの WaitIO Group(WG)におけるランク番号は 28 行目の waitio_group_rank()関数により取得することができる。

29 行目以降は通信を行うコードであるが，28 行目で取得したランク番号を元に偶数奇数で送受信の順序を変更する。ランク番号が奇数の場合は自己のランク番号より 1 つ小さなランク番号からの受信処理後送信，ランク番号が偶数の場合は自己のランク番号より 1 つ大きなランク番号への送信処理後に受信するコードとなっている。

WaitIO API が提供する通信は Non-Blocking 通信のみであるので，送受信処理の完了待ちのために waitio_wait()関数を発行している。

```
001 /* waitio-test.c */
002 #include <mpi.h>
003 #include "waitio.h"

004 int truef(int pbid, int n) { return 1; }

006 int main (int argc, char *argv[]) {
007   int ret, wrank;
008   waitio_filter_func_t func[4]= {truef, truef, NULL, NULL};
009   int  array[4] = {1, 2, 0, 0};
010   int  data[2], *buf = data;
011   waitio_req_t req;
012   waitio_group_t grp1;

013   if((ret = MPI_Init( &argc, &argv)) != MPI_SUCCESS) {
014     fprintf(stderr, "MPI_Init failed code %d¥n", ret);
015     exit(ret);
016   }
017   if((ret = waitio_init(-1)) != 0) {
018     fprintf(stderr, "waitio_init failed code %d¥n", ret);
019      MPI_Abort(MPI_COMM_WORLD, ret);
020     exit(ret);
021   }

022   grp1 = waitio_create_group(0, func, array);
023   if(grp1 == NULL) {
024     fprintf(stderr, "waitio_create_group failed code %d¥n", ret);
025     MPI_Finalize();
026   exit(ret);
027    }
028   waitio_group_rank(grp1, &wrank);
```

```
029   if((wrank%2) == 1) {
030     waitio_irecv(grp1, wrank-1, (char *)buf, 4, 0, &req);
031     if((ret = waitio_wait(&req)) != 0) {
032       fprintf(stderr, "%d waitio_irecv error %d¥n", wrank, ret);
033     }
034     else {
035       fprintf(stderr, "rank%d: waitio_irecv from rank%d¥n", wrank,  wrank-1);
036     }
037     waitio_isend(grp1, wrank-1, (char *)buf, 4, 0, &req);
038     if((ret = waitio_wait(&req)) != 0) {
039       fprintf(stderr, "%d waitio_isend error %d¥n", wrank, ret);
040     }
041     else {
042       fprintf(stderr, "rank%d: waitio_isend to rank%d¥n", wrank,  wrank-1);
043     }
044   }
045   else {
046     waitio_isend(grp1, wrank+1, (char *)buf, 4, 0, &req);
047     if((ret = waitio_wait(&req)) != 0) {
048       fprintf(stderr, "%d waitio_isend error %d¥n", wrank, ret);
048     }
050     else {
051       fprintf(stderr, "rank%d: waitio_isend to rank%d¥n", wrank,  wrank+1);
052     }
053     waitio_irecv(grp1, wrank+1, (char *)&buf, 4, 0, &req);
054     if((ret = waitio_wait(&req)) != 0) {
055       fprintf(stderr, "%d waitio_irecv error %d¥n", wrank, ret);
056     }
057     else {
058       fprintf(stderr, "rank%d: waitio_irecv from rank%d¥n", wrank,  wrank+1);
059     }
060   }
061   waitio_finalize();
062   MPI_Finalize();
063 }
```

図 5　WaitIO-Socket の API を用いた PingPong プログラム例

図 6 に図 5 のプログラムの実行例を示す。

この例では Intel MPI を用いた wisteria01 でのインタラクティブな実行例と Odyssey と Fujitsu MPI を用いたバッチシステムでの実行例を示す。

図 6 左は Intel MPI を用いた実行例である。WaitIO で利用するライブラリは利用する MPI 毎に提供されており，Intel MPI の場合は_intel，Fujitsu MPI の場合は_a64fx の post fix をライブラリに追加する。Aquarius の Open MPI の場合は Post Fix は不要である。また，実行時に図にある環境変数 WAITIO_MASTER_HOST, WAITIO_MASTER_PORT, WAITIO_PBID, WAITIO_NPB の設定が必要である。この他，WaitIO 関連のライブラリが環境変数 LD_LIBRARY_PATH の指定にある $(WAITIOLIB)にある想定としている。ここで，環境変数 WAITIO_MASTER_PORT については，同一ホストを MASTER とするプログラムが他に存在し PORT 番号が同じ場合誤動作するため WAITIO_MASTER_PORT の数値は他のプログラムと同じにならないよう注意する必要がある。図 6 左の例では続いて，これらの環境変数を実行環境に設定し mpiexec コマンドで 2 プロセスの場合，8 プロセスの場合を実行している。この実行においては，WAITIO_NPB の値が 1 であるために他の PB を待つことなく実行が開始される。

図 6 右上は Fujitsu MPI でコンパイルして Odyssey システムにてバッチ実行した実行例である。この例では図 6 左と同様にバッチスクリプトに必要な環境変数を設定し，pjsub でバッチ処理を開始する。実行結果は図 6 右下のようにファイルに出力される。

```
[z30xxx@wisteria01 waitio]$ mpiicc test.c  -o test –L$(WAITIOLIB)/ -lwaitio_intel

[z30xxx@wisteria01 waitio]$ cat env.sh
  export LD_LIBRARY_PATH= $(WAITIOLIB)/ :$LD_LIBRARY_PATH
  export WAITIO_MASTER_HOST=wisteria01
  export WAITIO_MASTER_PORT=7100
  export WAITIO_PBID=0
  export WAITIO_NPB=1
[z30xxx@wisteria01 waitio]$ . env.sh

[z30xxx@wisteria01 waitio]$ mpiexec -np 2  ./test
wisteria01:0:0:sock_create master_port 7100
wisteria01:0:0: WAITIO_MASTER socket bind port 7100
rank0: waitio_isend to rank1
rank1: waitio_irecv from rank0
rank0: waitio_irecv from rank1
rank1: waitio_isend to rank0

[z30xxx@wisteria01 waitio]$ mpiexec -np 8  ./test
wisteria01:0:0:sock_create master_port 7100
wisteria01:0:0: WAITIO_MASTER socket bind port 7100
rank0: waitio_isend to rank1
rank4: waitio_isend to rank5
rank1: waitio_irecv from rank0
rank2: waitio_isend to rank3
rank5: waitio_irecv from rank4
rank5: waitio_isend to rank4
rank0: waitio_irecv from rank1
rank1: waitio_isend to rank0
rank3: waitio_irecv from rank2
rank4: waitio_irecv from rank5
rank6: waitio_isend to rank7
rank6: waitio_irecv from rank7
rank7: waitio_irecv from rank6
rank7: waitio_isend to rank6
rank2: waitio_irecv from rank3
rank3: waitio_isend to rank2
[z30zzz@wisteria01 waitio]$
```

Intel MPIを用いたインタラクティブ実行例

```
[z30xxx@wisteria01 waitio]$ mpifccpx test.c  -o test-a64fx - L$(WAITIOLIB)/ -lwaitio_a64fx
[z30xxx@wisteria01 waitio]$ cat test-a64fx.sh
#!/bin/sh
#------ pjsub option --------#
#PJM -L rscgrp=debug-o
#PJM -L node=1
#PJM --mpi proc=2
#PJM -L elapse=00:02:00
#PJM -g jhxxxxxxo
#PJM -j
#------- Program execution -------#

export LD_LIBRARY_PATH= $(WAITIOLIB)/ :$LD_LIBRARY_PATH
module load fj
module load fjmpi
export WAITIO_MASTER_HOST=`hostname`
export WAITIO_MASTER_PORT=7100
export WAITIO_PBID=0
export WAITIO_NPB=1

mpiexec  ./test-a64fx
exit
[z30xxx@wisteria01 waitio]$ pjsub test-a64fx.sh
[INFO] PJM 0000 pjsub Job 244072 submitted.
[z30xxx@wisteria01 waitio]$
```

```
wo0001:0:0:sock_create master_port 7100
wo0001:0:0: WAITIO_MASTER socket bind port 7100
rank0: waitio_isend to rank1
rank1: waitio_irecv from rank0
rank1: waitio_isend to rank0
rank0: waitio_irecv from rank1
```

Fujitsu MPIを用いたバッチシステム実行例

図 6　WaitIO-Socket の API を用いた PingPong プログラムの実行例(単一プログラム実行)

```
[z30xxx@wisteria01 waitio]$ cat test-a64fx-1.sh
#!/bin/sh
#PJM -L rscgrp=debug-o
#PJM -L node=1
#PJM --mpi proc=2
#PJM -L elapse=00:02:00
#PJM -g jhxxxxxxo
#PJM -j
hostname
export LD_LIBRARY_PATH= $(WAITIOLIB)/ :$LD_LIBRARY_PATH
module load fj
module load fjmpi
export WAITIO_MASTER_HOST=`hostname`
export WAITIO_MASTER_PORT=7100
export WAITIO_PBID=0
export WAITIO_NPB=2
mpiexec  ./test-a64fx
exit

[z30xxx@wisteria01 waitio]$ cat test-intel-2.sh
#!/bin/sh
#PJM -L rscgrp=debug-a
#PJM -L node=1
#PJM --mpi proc=2
#PJM -L elapse=00:05:00
#PJM -g jhxxxxxxa
#PJM -j
module unload aquarius
module unload gcc
module load intel
module load impi
export LD_LIBRARY_PATH= L$(WAITIOLIB)/ :$LD_LIBRARY_PATH
export WAITIO_MASTER_HOST=wo0001
export WAITIO_MASTER_PORT=7100
export WAITIO_PBID=1
export WAITIO_NPB=2
mpiexec -n 2  ./test
exit

[z30xxx@wisteria01 waitio]$ pjsub test-a64fx-1.sh
[INFO] PJM 0000 pjsub Job 244102 submitted.
[z30xxx@wisteria01 waitio]$ pjsub test-intel-2.sh
[INFO] PJM 0000 pjsub Job 244104 submitted.
[z30zzz@wisteria01 waitio]$
```

```
wo0001
wo0001:0:0:sock_create master_port 7100
wo0001:0:0: WAITIO_MASTER socket bind port 7100
wo0001:0:0 waitio_init Procs:
  PB No.0 rank=0 IP-addr:port#=10.11.101.1:8000 nprocs=2, nGIO=0
  PB No.1 rank=0 IP-addr:port#=10.0.6.1:8000 nprocs=2, nGIO=0
wo0001:0/2(85): Multiple WaitIO Connecter now ready NPB=2!
 wo0001:0/2(85):PBserver: waitio_fetch_PBdata accepted from 10.0.6.1 port=38607 fd=33
rank0: waitio_isend to rank1
rank1: waitio_irecv from rank0
rank1: waitio_isend to rank0
rank0: waitio_irecv from rank1
```

Odysseyでの実行結果例

```
wa01:1:0/2(87):PBclient:trying connect host 10.11.101.1,port 48155
wa01:1/2(87):PBclient: connected to host 10.11.101.1,port 48155
wa01:1/2(87):PBclient: upstream to WaitIO master Nproc=2 done!
wa01:1:0 waitio_init Procs:
  PB No.0 rank=0 IP-addr:port#=10.11.101.1:8000 nprocs=2, nGIO=0
  PB No.1 rank=0 IP-addr:port#=10.0.6.1:8000 nprocs=2, nGIO=0
wa01:1/2(87): Multiple WaitIO Connecter now ready NPB=2!
rank2: waitio_isend to rank3
rank2: waitio_irecv from rank3
rank3: waitio_irecv from rank2
rank3: waitio_isend to rank2
```

Aquariusでの実行結果例

Wisteria/BDEC01(Odyssey+Aquarius)を用いた試行例

図 7　WaitIO-Socket の API を用いた PingPong プログラムの実行例(バッチによる複数プログラムの結合実行)

図 7 は Odyssey と Aquarius を用いて図 6 の実行バイナリを協調動作した例である。図 6 で生成した実行バイナリをそのまま利用してバッチスクリプトにて環境変数を設定するだけで協調動作することが可能である。バッチ処理の実行は WAITIO_PBID が 0 であるスクリプトから開始する。

図 7 左では Odyssey でのバッチ処理である test-a64fx-1.sh が WAITIO_PBID=0 であるためこのファイルから実行する。バッチ処理では実行時にノードが割り当てられるために，どのノードで実行されたかを知るためにスクリプト内の hostname コマンドを用いて環境変数 WAITIO_MASTER_HOST を設定するほか，Aquarius で実行する WAITIO_PBID=1 のバッチスクリプトに設定するために hostname コマンドでバッチ処理出力に出力している。

Aquarius 向けのバッチスクリプトである test-intel-2.sh ではバッチ処理 test-a64fx-1.sh の出力の hostname コマンド出力結果を test-intel-2.sh の WAITIO_MASTER_HOST と WAITIO_MASTER_PORT 環境変数に設定した後 pjsub コマンドにてバッチ処理を実行する。

図 7 右に Odyssey(上)，Aquarius(下)のバッチジョブ出力結果を示す。それぞれのバッチ処理出力では最初に 2 つのバッチジョブが結合されて 4 プロセスの WaitIO Instance が生成され Odyssey の 2 プロセスに WaitIO Gourp(WG)のランク番号 0,1 と Aquarius の 2 プロセスにランク番号 2,3 が割り当てられ 4 プロセスによるプログラム実行が行われたことを示している。この例ではそれぞれ 2 プロセスとしたがそれぞれのプロセス数を増減しても，実行時に反映される他，3 つ以上のバッチジョブについてもスクリプトを準備することで実行可能である。

7　WaitIO-MPI Conversion ライブラリの概要と動作仕様

WaitIO はシンプルな通信に特化するため MPI の Datatype に相当する概念がない。このため，MPI で書かれたプログラムを WaitIO に書き直すのはひと手間かかる。プログラムを書き直すとプログラム実行の不具合につながる。したがって，可能な限り機械的に書き直せることが望ましい。

このため, MPI とのインタフェース変換を担う WaitIO-MPI Conversion ライブラリを準備した。この WaitIO-MPI Conversion ライブラリは，必要になった関数を随時加えることとしている。API としては FORTRAN 向け API も実装している。

現状，実装されている関数を表 2 に示す。同期・非同期送受信関数(send, recv,isend, irecv)の他，集団通信関数(bcast, reduce, allreduce, gather, allgather, scatter, scatter, gather, battier)であり，扱う Datatype は数種の基本 Datatype(CHAR, INT, FLOAT, DOUBBLE)に限定している。アプリケーションの必要に応じて拡張する予定である。現状，関数名と datatype, op などの MPI オブジェクト表記は WAITIO_の prefix を機械的に追加することで WaitIO 向けに書き換えるようにしている。

実装されている集団通信アルゴリズムは動作確認と通信負荷を目的としたシンプルな 2 階層のシーケンシャル実装(2 Layered:以降 2L)と Binary(Binominal) Tree ベース(以降 BT)である。シーケンシャル実装の Bcast は root プロセスから各 worker プロセスに送信，reduce は各 worker プロセスから root プロセスに送信，root プロセスで演算する原始的なものである。また，Allreduce は reduce+bcast 実装である。　ただし 1 プロセスに同時通信できるソケット数が限られるため，一定プロセス数以上の場合は階層的に通信する実装としている。Binary(Binominal) Tree ベースは Binary Tree 上にプロセスをマップしメッセージ交換を実施する。

表.2 WaitIO-MPI Conversion ライブラリの API 概要

WaitIO-MPI API	概要
waitio_mpi_send	WaitIO実装版MPI_Send
waitio_mpi_recv	WaitIO実装版MPI_Recv
waitio_mpi_isend	WaitIO実装版MPI_Isend
waitio_mpi_irecv	WaitIO実装版MPI_Irecv
waitio_mpi_reduce	WaitIO実装版MPI_Reduce
waitio_mpi_bcast	WaitIO実装版MPI_Bcast
waitio_mpi_allreduce	WaitIO実装版MPI_Allreduce
waitio_mpi_waitall	WaitIO実装版MPI_Waitall
waitio_create_universe	Waitio初期化サポート関数:全ランク
waitio_create_universe_pbhead	Waitio初期化サポート関数:PB先頭
waitio_mpi_gather	WaitIO実装版MPI_Gather
waitio_mpi_allgather	WaitIO実装版MPI_Allgather
waitio_mpi_scatter	WaitIO実装版MPI_Scatter
waitio_mpi_scatter	WaitIO実装版MPI_Scatterv
waitio_mpi_gather	WaitIO実装版MPI_Gatherv
waitio_mpi_barrier	WaitIO実装版MPI_Barrier
waitio_mpi_type_size	WaitIO実装版MPI_Tyep_size

8　WaitIO-MPI Conversion ライブラリプログラミング

```
000 /* test-mpi.c*/
001 #include <mpi.h>
002 #include "waitio.h"
003 #include "waitio_mpi.h"

004 int main (int argc, char *argv[]) {
005   int  data[2], *buf = data;
006   waitio_group_t grp1;
007   int ret;
008   int wrank;

009   if((ret = MPI_Init( &argc, &argv)) != MPI_SUCCESS) {
010     fprintf(stderr, "MPI_Init failed code %d¥n", ret);
011     exit(ret);
012   }
013
014   waitio_create_universe (&grp1);
015   if(grp1 == NULL) {
016     fprintf(stderr, "waitio_create_universe failed code %d¥n", ret);
017     MPI_Finalize();
018     exit(ret);
019   }
```

waitio_mpi_recv はWAITIO_MPI_Recv
waitio_mpi_send はWAITIO_MPI_Send
としても定義されているため、MPIプログラムの機械的な
置き換えも可能である。

```
020   waitio_group_rank(grp1, &wrank);
021   if((wrank%2) == 1) {
022     if((ret = waitio_mpi_recv(buf, 4, WAITIO_MPI_CHAR, wrank-1, 0, grp1)) != 0) {
023       fprintf(stderr, "%d waitio_mpi_recv error %d¥n", wrank, ret);
024     }
025     else {
026       fprintf(stderr, "rank%d: waitio_mpi_recv from rank%d¥n", wrank,  wrank-1);
027     }
028     if((ret = waitio_mpi_send(buf, 4, WAITIO_MPI_CHAR, wrank-1, 0, grp1)) != 0) {
029       fprintf(stderr, "%d waitio_send error %d¥n", wrank, ret);
030     }
031     else {
032       fprintf(stderr, "rank%d: waitio_mpi_send to rank%d¥n", wrank,  wrank-1);
033     }
034   }
035   else {
036     if((ret = waitio_mpi_send(buf, 4, WAITIO_MPI_CHAR, wrank+1, 0, grp1)) != 0) {
037       fprintf(stderr, "%d waitio_send error %d¥n", wrank, ret);
038     }
039     else {
040       fprintf(stderr, "rank%d: waitio_mpi_send to rank%d¥n", wrank,  wrank+1);
041     }
042     if((ret = waitio_mpi_recv(buf, 4, WAITIO_MPI_CHAR, wrank+1, 0, grp1)) != 0) {
043       fprintf(stderr, "%d waitio_recv error %d¥n", wrank, ret);
044     }
045     else {
046       fprintf(stderr, "rank%d: waitio_mpi_recv from rank%d¥n", wrank,  wrank+1);
047     }
048   }

049   waitio_finalize();
050   MPI_Finalize();
051 }
```

図 8.　WaitIO-MPI Conversion ライブラリの API を用いた PingPong プログラム例

図 8 に WaitIO-MPI Conversion ライブラリの API を用いた PingPong プログラムを示す。このプログラムは図 5 のプログラムをそのまま WaitIO-MPI Conversion ライブラリに置き換えたもので，0 から始まる WaitIO のランク番号が偶数の場合は送信後に受信，奇数の場合は受信後に送信する。

図 8 において左半分は初期化部分である。9 行目の MPI の初期化の後 14 行目で WaitIO-MPI

Conversion ライブラリの初期化関数を呼んでいる。ここでは，WaitIO Instance すべてのプロセスを利用する waitio_create_universe()関数を用いている。現在のところ本関数と各 PB の先頭プロセスをグループ化する waitio_create_universe_pbhead()関数が用意されている。

14 行目で生成されたプロセスグループの WaitIO Group(WG)におけるランク番号は 20 行目の waitio_group_rank()関数により取得することができる。

21 行目以降は通信を行うコードであるが，20 行目で取得したランク番号を元に偶数奇数で送受信の順序を変更する。ランク番号が奇数の場合は自己のランク番号より 1 つ小さなランク番号からの受信処理後送信，ランク番号が偶数の場合は自己のランク番号より 1 つ大きなランク番号への送信処理後に受信するコードとなっている。WaitIO ライブラリと異なるのはデータタイプの指定が必要である点である。

WaitIO-MPI Conversion ライブラリ API は Non-Blocking 通信の他 Blocking 通信も準備されているので，本プログラムでは Blocking 通信関数を用いている。

```
[z30xxx@wisteria01 waitio]$ mpiicc test-mpi.c -o test-mpi –L${WAITIOLIB}/ -
lwaitio_mpi_intel -lwaitio_intel
[z30xxx@wisteria01 waitio]$ mpifccpx test-mpi.c -o test-mpi-a64fx -
L${WAITIOLIB}/ -lwaitio_mpi_a64fx -lwaitio_a64fx
[z30xxx@wisteria01 waitio]$ cat test-mpi-a64fx-1.sh
#!/bin/sh
#PJM -L rscgrp=debug-o
#PJM -L node=1
#PJM --mpi proc=2
#PJM -L elapse=00:02:00
#PJM -g jh2xxxxxxo
#PJM -j
hostname
export LD_LIBRARY_PATH= ${WAITIOLIB}/ :$LD_LIBRARY_PATH
module load fj
module load fjmpi
export WAITIO_MASTER_HOST=`hostname`
export WAITIO_MASTER_PORT=7100
export WAITIO_PBID=0
export WAITIO_NPB=2
mpiexec ./test-mpi-a64fx
exit

[z30xxx@wisteria01 waitio]$ cat test-mpi-intel-2.sh
#!/bin/sh
#PJM -L rscgrp=debug-a
#PJM -L node=1
#PJM --mpi proc=2
#PJM -L elapse=00:05:00
#PJM -g jh2xxxxxa
#PJM -j
module unload aquarius
module unload gcc
module load intel
module load impi
export LD_LIBRARY_PATH= ${WAITIOLIB}/ :$LD_LIBRARY_PATH
export WAITIO_MASTER_HOST=wo0009
export WAITIO_MASTER_PORT=7100
export WAITIO_PBID=1
export WAITIO_NPB=2
mpiexec -n 2 ./test-mpi
exit

[z30xxx@wisteria01 waitio]$ pjsub test-mpi-a64fx-1.sh
[INFO] PJM 0000 pjsub Job 244535 submitted.
###< Check the output of Job 24435 and set the hostname of the job output
###  to test-mpi-intel-2.sh >
[z30xxx@wisteria01 waitio]$ pjsub test-mpi-intel-2.sh
[INFO] PJM 0000 pjsub Job 244537 submitted.
```

```
wo0009
wo0009:0:0:sock_create master_port 7100
wo0009:0:0: WAITIO_MASTER socket bind port 7100
wo0009:0:0 waitio_init Procs:
  PB No.0 rank=0 IP-addr:port#=10.11.101.9:8000 nprocs=2, nGIO=0
  PB No.1 rank=0 IP-addr:port#=10.0.6.1:8000 nprocs=2, nGIO=0
wo0009:0/2(83): Multiple WaitIO Connecter now ready NPB=2!
 wo0009:0/2(83):PBserver: waitio_fetch_PBdata accepted from 10.0.6.1 port=30403 fd=33
rank0: waitio_mpi_send to rank1
rank1: waitio_mpi_recv from rank0
rank1: waitio_mpi_send to rank0
rank0: waitio_mpi_recv from rank1
```

Odysseyでの実行結果例

```
wa01:1:0/2(87):PBclient:trying connect host 10.11.101.9,port 48155
wa01:1/2(87):PBclient: connected to host 10.11.101.9,port 48155
wa01:1/2(87):PBclient: upstream to WaitIO master Nproc=2 done!
wa01:1:0 waitio_init Procs:
  PB No.0 rank=0 IP-addr:port#=10.11.101.9:8000 nprocs=2, nGIO=0
  PB No.1 rank=0 IP-addr:port#=10.0.6.1:8000 nprocs=2, nGIO=0
wa01:1/2(87): Multiple WaitIO Connecter now ready NPB=2!
rank2: waitio_mpi_send to rank3
rank2: waitio_mpi_recv from rank3
rank3: waitio_mpi_recv from rank2
rank3: waitio_mpi_send to rank2
```

Aquariusでの実行結果例

Wisteria/BDEC01(Odyssey+Aquarius)を用いた試行例

図 9. WaitIO-MPI Conversion ライブラリの API を用いた PingPong プログラムの実行例
(バッチによる複数プログラムの結合実行)

図 9 に Odyssey と Aquarius を用いて図 8 のプログラムをバイナリ生成と協調動作した例である。図 7 に示した WaitIO ライブラリのプログラムとコンパイル，実行方法はほとんど同じであるが，コンパイル時に-lwaitio_mpi ライブラリの追加が異なる。

9　WaitIO-MPI Conversion の適用例

本章では WaitIO-MPI Conversion の適応例として MPI+FORTRAN で書かれた pHEAT-3D について述べる。pHEAT-3D は GeoFEM[12]プロジェクトで開発された並列有限要素法アプリケーションを元に開発した三次元定常熱伝導解析コードである。pHEAT-3D の主要な通信部分は MPI_Isend()+MPI_Irecv()で書かれた通信部分と MPI_Allreduce()で書かれた部分である。

図 10 に主要部分のソース変更を示す。赤字で書かれたところが修正部分である。基本的に MPI プログラムの MPI_のプレフィックスを WAITIO_MPI_のプレフィックスに置換することでソースコード修正できるように設計している。

一つ MPI と異なる点としては，WaitIO では MPI_Status に相当するオブジェクトが存在しないのと MPI_Request に相当するオブジェクトが MPI はスカラー定義であるが WaitIO は構造体である点に注意が必要である。このため，プログラムにおいては Request オブジェクトを構造体として確保する必要がある。例えば FORTRAN では WAITIO_REQUEST_SIZE の大きさをもつ配列として定義する必要がある。

```
      include 'mpif.h'
      include 'waitio_mpif.h'
      integer(kind=kint ), dimension(:,:), save,allocatable :: req1
      integer, save :: NFLAG
      data NFLAG/0/

!C
!C-- INIT.
      if (allocated(sta1)) deallocate (sta1)
      if (allocated(req1)) deallocate (req1)
      allocate (sta1(WAITIO_STATUS_SIZE,2*NEIBPETOT+4))
      allocate (req1(WAITIO_REQUEST_SIZE,2*(NEIBPETOT+4)))
!C
!C-- SEND
      do neib= 1, NEIBPETOT
        istart= STACK_EXPORT(neib-1)
        inum  = STACK_EXPORT(neib  ) - istart
!$omp parallel do private (k,ii)
        do k= istart+1, istart+inum
          ii= NOD_EXPORT(k)
          WS(k)= X(ii)
        enddo
        call WAITIO_MPI_Isend (WS(istart+1), inum,
     &       WAITIO_MPI_DOUBLE_PRECISION,
     &       NEIBPE(neib), 0, WAITIO_SOLVER_COMM, req1(1,neib), ierr)
      enddo
```

```
!C-- RECV
      do neib= 1, NEIBPETOT
        istart= STACK_IMPORT(neib-1)
        inum  = STACK_IMPORT(neib  ) - istart
        call WAITIO_MPI_Irecv (X(istart+N0+1),inum,
     &                WAITIO_MPI_DOUBLE_PRECISION,
     &                NEIBPE(neib), 0, WAITIO_SOLVER_COMM,
     &                req1(1,neib+NEIBPETOT), ierr)
      enddo
!C
      call WAITIO_MPI_Waitall (2*NEIBPETOT, req1, sta1, ierr)

      end subroutine SOLVER_SEND_RECV
      end module    solver_SR
```

```
!$omp parallel do private(i) reduction(+: RHO0)
      do i= 1, N
        RHO0= RHO0 + WW(i,R)*WW(i,Z)
      enddo

      call WAITIO_MPI_Allreduce (RHO0, RHO, 1,
     &                WAITIO_MPI_DOUBLE_PRECISION,
     &                WAITIO_MPI_SUM, WAITIO_SOLVER_COMM, ierr)
```

図 10. WaitIO-MPI Conversion ライブラリにおける従来プログラム pHEAT-3D の修正例

10. WaitIO-Socket 通信性能評価

本章は WaitIO-Socket 通信ライブラリが Wisteria システム上でどの程度通信性能を引き出せるかを検証することを目的として基本通信性能評価を実施した結果を述べる。

10.1 評価環境と評価プログラム

WaitIO-Socket の評価においては，より広いシステムでも稼働することを確認するため Wisteria システムの他に Oakbridge-CX, Oakforest-PACS を用いて実施した。評価環境の仕様を表 3 に示す。

表 3　Wisteria/BDEC-01, Oakbridge-CX, Oakforest-PACS の各構成システム仕様

	Wisteria/Odyssey	Wisteria/Aquarius
総理論演算性能	25.9 PFLOPS	7.2 PFLOPS
総ノード数	7,680	45
インターコネクト	Tofu-D (6次元メッシュ/トーラス)	InfiniBand HDR (200Gbps) x 4HCA (Full Fat Tree)
プロセッサ/ノード	A64FX(Armv8.1+SVE), 2.2GHz, 1 socket (48 Core+2 or 4アシスタントコア)	Intel Xeon Platinum 8360Y, 2.4GHz 2 sockets(36+36)
メモリ量/ノード	32 GB	512GiB
メモリバンド幅/ノード	1024GB/s	409.6GB/s
GPU/ノード	-	NVIDIA A100 x8
コンパイラ、MPI	富士通コンパイラ、富士通MPI	Intelコンパイラ、インテルMPI、(Open MPI)
オペレーティングシステム	Red Hat Enterprise Linux 8	Red Hat Enterprise Linux 8

	Oakbridge-CX (OBCX)	Oakforest-PACS (OFP)
総理論演算性能	6.61 PFLOPS	25 PFLOPS
総ノード数	1368	8208
インターコネクト	Omni-Path (100Gbps)	Omni-Path (100Gbps)
プロセッサ/ノード	Intel Xeon Platinum 8280 2.7GHz, 2 socket (28+28)	Intel Xeon Phi 7250 1.4GHz 1 socket (68)
メモリ量/ノード	192 GB	96(DDR4)+16(MCDRAM) GB
メモリバンド幅/ノード	281.6GB/s	115(DDR4)+490(MCDRAM) GB/s
コンパイラ、MPI	Intelコンパイラ、インテルMPI、(Open MPI)	Intelコンパイラ、インテルMPI、(Open MPI)
オペレーティングシステム	Red Hat Enterprise Linux 7, CentOS 7	Red Hat Enterprise Linux 7, CentOS 7

評価プログラムはシステム間のデータ通信性能を評価するため複数ストリームでの PingPong 通信の評価プログラムを作成した。これは，全 PB 内プロセスで 2 プロセスペアを作り PingPong 通信を行うものである。すべてのペアの PingPong 通信終了を MPI_Barrier()で待ち合わせて測定している。

実行評価プログラムはそれぞれのシステムにインストールされているベンダ製コンパイラ＋ベンダ製 MPI で評価してグラフ化したが Xeon プロセッサ搭載システムでは Open MPI+GCC の組み合わせでも動作確認している。

10.2 1PB 内 1 対 1 通信評価(multi-stream)

本節では 1 PB 内でのマルチストリーム 1 対 1 通信評価について述べる。

10.2.1 PingPong 1/2 RTT 評価

本節では各システムで測定した PingPong1/2 RTT の性能評価について述べる。表 4. に各システムの 8 Byte の 1/2 RTT(ノード内，ノード間)結果のサマリを示す。各データはノード数を固定，ノードあたりのプロセス数を増加させ，複数プロセスによる複数ストリームを 1000 回実行し MPI_Barrier()で同期後の時間を回数分で割り算した平均値を示している。すなわち，各回のストリーム実行の最悪値の平均である。

表 4　Wisteria/BDEC-01, Oakbridge-CX, Oakforest-PACS の 1/2RTT 測定結果

usec(8B)	1/2 RTT(ノード内)	1/2 RTT(ノード間)
Odyssey	26.8	37.1 (Tofu-D)
Aquarius	6.57	21.1 (IB)
OBCX	6.85	14.5 (OPA)
OFP	49.7	83.7 (OPA)

表 4. より，Intel Xeon プロセッサシステムのノード内の測定結果は，ほぼ同等の実行時間であ

る。一方，Odyssey, OFP については Xeon 搭載システムに比べ 4 倍弱(Odyssey)，9 倍弱(OFP)遅い結果となっている。相対的なプロセッサ処理の差であるほか，Odyssey の結果に関して，カーネル内処理はアシスタントコアで実行されるためにその遅延も含まれていると想定される。

10.2.2 PingPong バンド幅評価

本節では各システムで測定した PingPong バンド幅の性能評価について述べる。表 5.に各システムの PingPong バンド幅(ノード内，ノード間)の最大値と 1 ノードあたりの PingPong バンド幅のサマリを示す。各データは複数プロセスによる複数ストリームを 1000 回実行し MPI_barrier()で同期後の時間を回数分で割り算した平均値を示している。

表 5　Wisteria/BDEC-01, Oakbridge-CX, Oakforest-PACS の PingPong バンド幅性能の測定結果

PingPong BW GB/s	BW ノード内	BW ノード間
Odyssey	21.3	0.35 (Tofu-D)
Aquarius	259.7	27.1 (IB)
OBCX	45.7	9.3 (OPA)
OFP	8.2	1.12 (OPA)

表 5.より，各システムのメモリ帯域，インターコネクト帯域，そしてコピー処理，通信プロトコルを実行するプロセッサの処理性能の影響が見られる結果となっている。

表 5 の結果の中で最も高い性能を記録したのは，Aquarius である。ノード内性能ではノードあたり 260GB/秒，ノード間性能でもノードあたり 27GB/秒を観測した。PingPong 処理においては WaitIO-Socket ライブラリ-カーネル間コピーが 2 回発生する。このため Aquarius の 960 プロセス結果のメッセージ長が 8MB までの区間で大きく山が出ているのはキャッシュの影響であると想定される。

次にノード間のインターコネクト毎の性能差に着目すると，やはり最新の InfiniBand HDR 4 本を搭載する Aquarius がプロセッサ性能とのバランスに優れ ノードあたり 27.1GB/秒の通信性能を記録している。物理性能が半分の OmniPath を搭載している OBCX は 9.3GB/秒を記録している。その性能差は 2.9 倍であった。

一方，同じ OmniPath を搭載する OFP の結果については 1.1GB/秒と OBCX に比べ 8.5 倍の性能差である。OFP に関してはノード内，ノード間とも他のシステムに比べ大きな性能差がないために，プロセッサコアの処理性能で律速していると想定される。

ここでノード間の通信性能が 0.35GB/秒の Odyssey について調べてみる。図 11 に Odyssey のノード内とノード間の測定結果を示す。図 11 左のノード内の測定結果に比べ図 11 右のノード間の結果を比較すると，ノード内の通信性能に比べ，ノード間の性能が 61 倍低い結果となった。これはノード内通信が各計算コアによるシステムコール経由のメモリコピーで済むのに対して，ノード間通信はバックグラウンドで実行される TCP/IP 処理が 1 ノードあたり 2 or 4 コアのアシスタントコアで実行されるために，このアシスタントコア処理がボトルネックとなり性能が律速されていると考えられる。

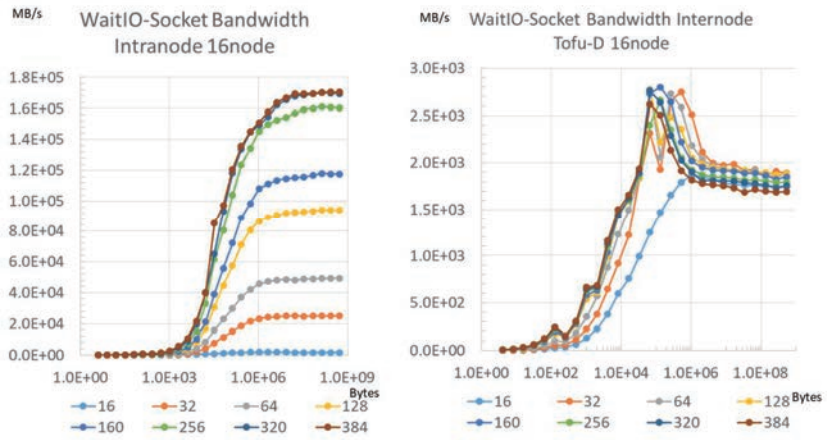

図 11.　Wisteria/BDEC-01(Odyssey)の PingPong バンド幅性能の測定結果

同様に Aquarius についても図 12 にノード内とノード間の測定結果を示す。Odyssey の測定結果と異なり，ノード間の通信性能についてもノード内プロセスの増加と共に通信性能が向上していることがわかる。

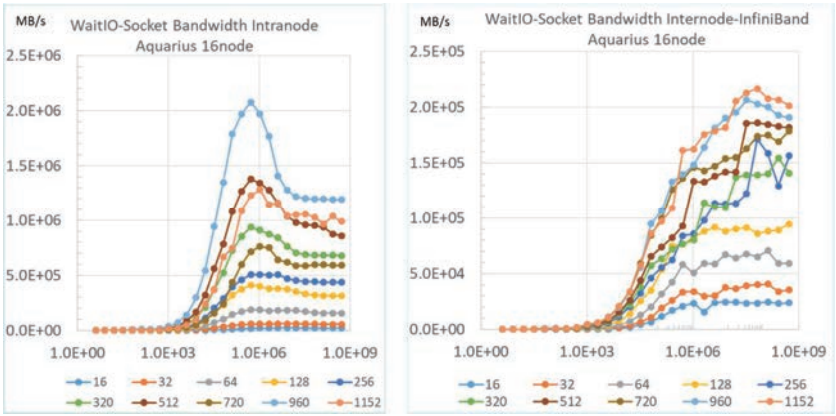

図 12　Wisteria/BDEC-01(Aquarius)の PingPong バンド幅性能の測定結果

10.3 2PB 結合での通信性能評価

本節では複数ジョブを WaitIO-Socket で結合した場合の通信性能について評価する。前節の大規模環境評価を 1 ジョブ(1 PB:Parallel Block)と 2 ジョブ(2 PB)で実行し性能差を評価する。

図 13.左に Odyssey 1 PB(2048node) 2 PB (1024+1024node)，図 13.右に Aquarius 1 PB(36node)と 2 PB(18+18node)のノード間の通信性能の比較結果を示す。

図 13.左より，Odyssey のノード間通信性能については少し性能にバタつきが見られるが傾向は類似という結果となった。

Aquarius については図 13.右より 1PB より 2PB の方が高い性能となっているが概ね同様の傾向の性能結果となった。実装上 1PB でも 2PB でも相手の IP アドレスと Port 番号をベースに相手と通信する仕組みは変わらないので，1 ジョブ 1PB と 2 ジョブ 2PB の性能差が出る可能性としては，ジョブの割り当て場所，ランク割り当てノードとインターコネクト上の位置の違いが考えられる。

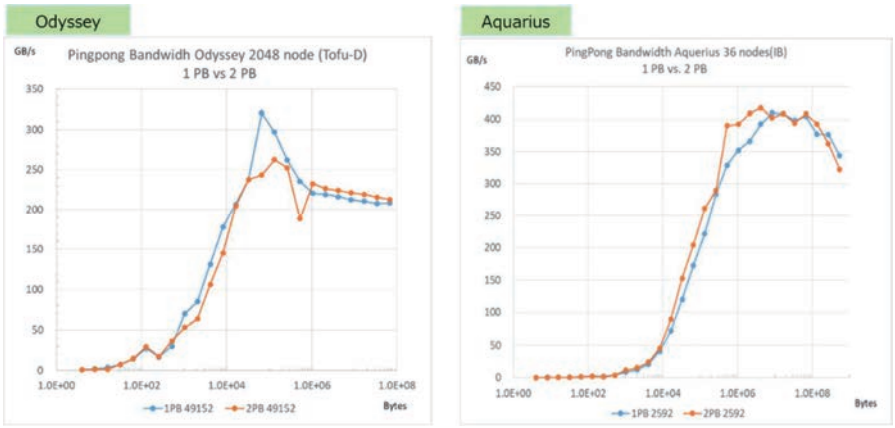

図 13.　Wisteria/BDEC-01(Aquarius)の PingPong バンド幅性能の測定結果

10.4 アプリケーションを用いた WaitIO-MPI Conversion ライブラリの評価

WaitIO-MPI Conversion ライブラリを用いて pHEAT-3D アプリケーション[12]を移植し実行時間を評価した。実行環境は OBCX であり比較対象は Intel MPI(OmniPath)である。評価結果はノードあたり 8 プロセス，1 プロセスあたり 6 スレッド，1000STEP の実行時間の結果である。移植作業はエディタ上で機械的に実施可能で実際には WAITIO_MPI_Isend(), WAITIO_MPI_Irecv(), WAITIO_MPI_Allreduce() を用いて置き換えている。集団通信の利用アルゴリズムは BT である。

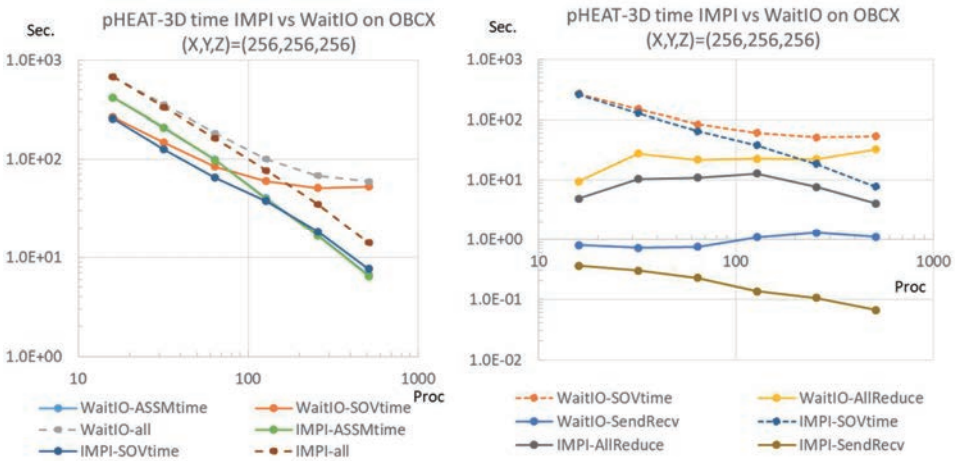

図 14. WaitIO-Socket の pHEAT-3D の性能比較(vs. Intel MPI)

図 14.左に配列サイズ(X,Y,Z)=(256,256,256)の測定結果を示す。図 14.左の結果より , Intel MPI(IMPI) は 512 プロセスまで性能向上しているが，WaitIO-Socket の測定結果では 256 プロセス以上では性能が頭打ちであった。

図 14.右に pHEAT-3D で利用している 4 回の Allreduce，iSend/iRecv で実装された SendRecv 通信時間と Solver 全体の実行時間を示す。図 14. 右の結果より AllReduce 通信が支配的であることが観測された。

なお，WaitIO-ASSEtime と Intel MPI-ASSEtime の性能差がほとんどないのは計算のみで通信を実施していないからである。

10.5 Wisteria/BDEC-01 全系を用いた WaitIO-Socket の通信性能評価

本節では Wisteria/BDEC-01 全系を用いた WaitIO-Socket の通信性能評価を行う。Wisteria を構成する Odyssey と Aquarius システム間は InfiniBand EDR で結合されている。この InfiniBand EDR は Odyssey の 1 ラックあたり 8 つ存在する GIO に結合されている。Wisteria の各ノードへのネットワークアクセスについて，Aquarius システムは各計算ノードに InfiniBand HDR が搭載されているが, Odyssey システムの各計算ノードは GIO と呼ばれる Gateway 用の計算ノードを除き Tofu-D インターコネクトのみを搭載している。GIO 以外の計算ノードは GIO を経由した TCP/IP 通信によりアクセス可能である。図 15 に Wisteria/BDEC-01 のインターコネクトの接続イメージ図を示す。

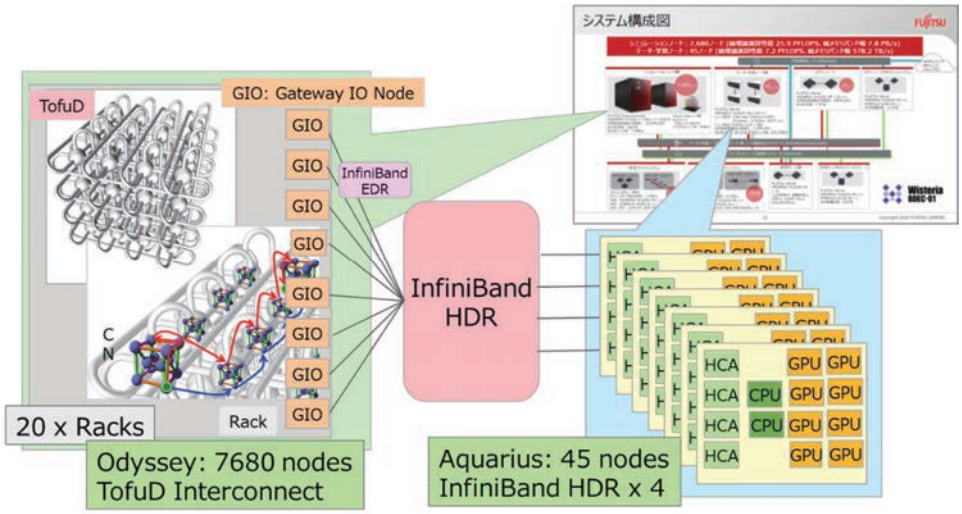

図 15. Wisteria/BDEC-01 の Odyssey-Aquarius 間のインターコネクト接続構成図

図 15 において Odyssey のノードから Aquarius のノードへの通信経路は 2 種類あり，一つは GIO から InfiniBand を用いた直接的な通信と GIO 以外のノードからの GIO を経由した通信の 2 種類である。このため，性能測定も GIO 直接の場合と GIO を経由した場合の 2 種測定した。

Odyssey 計算ノード- Aquarius計算ノード間
1ノードあたり転送性能: 350MB/s
GIO Routing性能1/2: 175MB/s
1ラックあたり 175 x 8: 1.4GB/s
全系通信性能20ラック： 28 GB/s

Odyssey GIOノード- Aquarius計算ノード間
GIO vs Aquarius計算ノード間通信性能: 370MB/s
(iperf3)
1ラックあたり 370 x 8: 2.96GB/s
全系通信性能20ラック： 59.2 GB/s

図 16. Wisteria/BDEC-01 の Odyssey-Aquarius 間の TCP/IP 転送性能ベースの通信性能見積もり

図 16 に GIO 直接と GIO 経由の場合の転送性能を元にした Odyssey-Aquarius 間の総転送性能見積もりを示す。GIO におけるルーティング処理でのオーバヘッドにより GIO のみを用いた場合に比べ約半分の通信性能となっている。

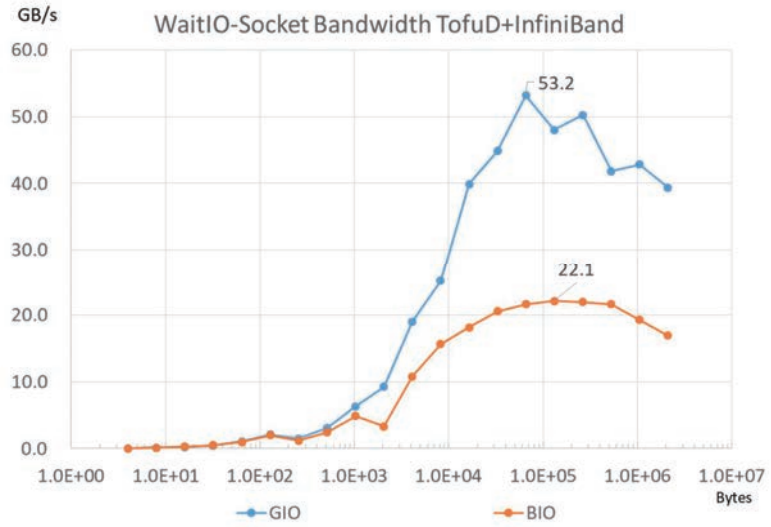

図 17. Wisteria/BDEC-01 の Odyssey-Aquarius 間の WaitIO-Socket の PingPong 転送性能

図 17 に全系を用いた通信性能の測定結果を示す。保守時の試験環境のため Aquarius は 8 ノードに限定されている他，測定時間の関係で Odyssey は全系ノードではなく 1 ラックに 24 ノード搭載されている BIO(Boot IO ノード)からの測定となっている。この GIO, BIO を選別する WaitIO Group の作成には，waitio_create_group()関数の評価関数として GIO 並びに BIO の IP アドレスが一致したものを WG のメンバとして選択する関数を作成してグループ化している。

図 17 の結果より，BIO を用いた場合(GIO のルーティングあり)は 22.1GB/秒，GIO 直接の場合は 53.2GB/秒の通信性能を得た。予測性能に比べて GIO ルーティングありの場合で 79%の性能，GIO 直接の場合で 90%の性能であった。ラックを増加するごとに通信性能が増しているが GIO 単体の通信性能の向上が今後の課題である。アプリケーション実行上は実行ノード数と配置を分散させることである程度所要の通信性能を確保できる場合もあるため，適用事例ごとに通信性能高速化の必要性を見極めていく。

11. まとめ

本稿では Wisteria/BDEC-01 利用事例として h3-Open-BDEC と h3-Open-BDEC の一つのモジュールの一つである h3-Open-SYS/WaitIO の概要(1/3)について，サンプルを含めたプログラミングと実行例を交えて紹介した。次号(2/3)では簡単なプログラムを使用したより詳細な実装例と h3-Open-UTIL/MP について紹介する。

参考文献

[1] Wisteria/BDEC-01（「計算・データ・学習」融合スーパーコンピュータシステム）:https://www.cc.utokyo.ac.jp/supercomputer/wisteria

[2] 中島研吾，塙敏博，下川辺隆史，伊田明弘，芝隼人，三木洋平，星野哲也，有間 英志，河合直

聡，坂本龍一，岩下武史，八代尚，長尾大道，松葉浩也，荻田武史，片桐孝洋，古村孝志，鶴岡弘，市村強，藤田航平，近藤正章，「計算・データ・学習」融合スーパーコンピュータシステム「Wisteria/BDEC- 01」の概要，情報処理学会研究報告（2020-HPC-179-01），2021

[3] 塙 敏博，中島 研吾，下川辺隆史，芝隼人，三木洋平，星野哲也，河合直聡，似鳥啓吾，今村俊幸，工藤周平，中尾 昌広，「計算・データ・学習」融合スーパーコンピュータシステム「Wisteria/BDEC- 01」の性能評価，情報処理学会研究報告（2021-HPC-180-22），2021

[4] 住元真司，荒川隆，坂口吉生，松葉浩也，八代尚，塙敏博，中島研吾，WaitIO-Socket：異種システム上の複数 MPI プログラムを結合する通信ライブラリの試作，情報処理学会研究報告（2021-HPC-181-07），2021

[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

[6] ppOpen-HPC: https://github.com/Post-Peta-Crest/ppOpenHPC

[7] h3-Open-BDEC：https://h3-open-bdec.cc.u-tokyo.ac.jp/

[8] Iwashita, T, Nakajima, K., Shimokawabe, T., Nagao, H., Ogita, T., Katagiri, T., Yashiro, H., Matsuba, H., h3-Open-BDEC: Innovative Software Platform for Scientific Computing in the Exascale Era by Integrations of (Simulation + Data + Learning), Project Poster, ISC-HPC 2020

[9] Nakajima, K., Matsuba, K., Hanawa, T., Furumura, T., Tsuruoka, H., Nagao, H., Integration of 3D Earthquake Simulation & Real-Time Data Assimilation on h3-Open-BDEC, MS290: Progress & Challenges in Extreme Scale Computing & Data SIAM Conference on Computational Science & Engineering (CSE21) (Online, March 4, 2021)

[10] SIAM News (March 10, 2021), Supercomputer Simulations of Earthquakes in Real Time (by Jillian Kunze), https://sinews.siam.org/Details-Page/supercomputer-simulations-of-earthquakes-in-real-time

[11] 中島研吾，古村孝志，鶴岡弘，坂口吉生，松葉浩也，住元真司，八代尚，荒川隆，塙敏博，観測データ同化による長周期地震動リアルタイム予測へ向けた試み，情報処理学会研究報告（2021-HPC-182-08），2021

[12] Nakajima, K., Parallel Iterative Solvers of GeoFEM with Selective Blocking Preconditioning for Nonlinear Contact Problems on the Earth Simulator. ACM/IEEE Proceedings of SC'03, https://doi.org/10.1145/1048935.1050164, 2003