Vol.2015-HPC-151 No.19
2015/10/1

「京」コンピュータでの C++型流体コードにおける MPI の評価

ファム　バン　フック[†1]　井上義昭[†2]　浅見暁[†2]　内山学[†1]　千葉修一[†3]

本研究では C++オープンソース OpenFOAM を対象として，利用しているデータ交換形態，C++テンプレートおよび MPI プラットフォームの特徴とその課題を述べた．また，「京」コンピュータの Tofu 高機能バリア通信機能を活用して，データ型に合わせたテンプレートの追加による全体実行時間の軽減を確認した．また，OpenFOAM 特有の PstreamBuffer 全体データ交換形態を必要最小限の隣接データ交換形態に改良し，通信バッファサイズおよび通信時間が減少した．これらにより大規模並列処理を可能にして，アプリケーション全体の実行効率が大幅に向上した

Evaluation of MPI Optimization of C++ CFD Code on the K computer

PHAM VAN PHUC[†1]　YOSHIAKI INOUE[†2]
AKIRA AZAMI[†2]　MANABU UCHIYAMA[†1]　SHUICHI CHIBA[†3]

In this study, a CFD open-source called OpenFOAM using C++ language in a large scale simulation has been investigated. Its MPI platform and C++ templates has been discussed to clarify the problems and advantage characteristics of a CFD code using C++. Some efforts have been made to reduce the execution time by adding the C++ templates with specific data type to utilize the Tofu highly functional barrier communication of "K" computer. A new data exchange method has also been proposed to minimize data transfers basing on the adjacent data exchange form. It successes to reduce the communication buffer size and the communication time and improve the performance of the entire application in massively parallel solution.

1. はじめに

C や FORTRAN 等の伝統的な言語は構造化された言語であり，数十万行までの中程度の複雑なプログラムを記述することができた．しかし，コードの行数がある規模に達すると，プログラムが過度に複雑化し，プログラマが全体を掌握することが難しくなる．より大規模なプログラムを取り扱えるように，1979 年にベル研究所のコンピュータ科学者 Bjarne Stroustrup によって C を拡張したオブジェクト指向プログラミング言語である C++が考案された．

C++は，C の構造化プログラミングの考え方を最大限に生かし，より効果的にプログラムを組織化するためのカプセル定義，多重継承，仮想関数，テンプレート等の多種多様な機能を備え，プログラムの生産性や柔軟性を高める言語である．これより，近年の商用ソフトウェアからオープンソース,更に HPC 分野のソフトウェアでも C++を利用した開発が増加している．特に，数値流体解析の CAE 分野では，現在，最も普及しているオープンソースのライブラリーの一つである OpenFOAM(Open source Field Operation And Manipulation)[1]が挙げられる．

OpenFOAM では C++のプログラミングに基づく乱流，燃焼,および混相流など様々な物理モデルが用意されており，対象に合わせたソルバ，クラスライブラリーを利用することができる．これより，研究者から CAE 利用者まで，C++のオブジェクト指向によって，コードの生産性を確保しながら，高度なシミュレーションを実現できる [2]．また，OpenFOAM では C++コードとして命令レベルの高速化を実現するために，数多くの技術を有している．その一つが Expression Template 技術の利用である．OpenFOAM を構成する基底クラスでこの技術を利用することで，コンパイラの最適化に頼ることなく一時的なメモリ利用を省略し，ソルバ演算の演算密度を向上させる利点が得られる．

しかしながら，C++は C や FORTRAN に比べて歴史が浅く，科学技術計算用の数値計算ライブラリーが少ない．OpenFOAM のような計算コードも「京」コンピュータで実行されるような大規模並列処理の実績が少なく，プロセスレベルでの高速化が検証できていない．そのため，コードチューニングの観点からは高速化がしにくく並列計算の性能も出しにくいと思われている．

本論では OpenFOAM を対象として，コードのテンプレート構造や利用している並列処理プラットフォームの特徴とその課題について述べる．また，テンプレートの活用により，データ型やデータ交換形態を考慮したプラットフォームに改良し，最適化を行う．更に，実際の流体解析アプリケーションを実装して「京」コンピュータでの大規模並列計算によりその性能特性を検証し，評価結果を報告する．

†1 清水建設株式会社
SHIMIZU Corporation
†2 一般財団法人高度情報科学技術研究機構
Research Organization for Information Science and Technology
†3 富士通株式会社
Fujitsu Ltd.

ⓒ2015 Information Processing Society of Japan

①初期化

```
int main(int argc, char *argv[])
{
    #include "setRootCase.H"

    #include "createTime.H"
    #include "createMesh.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"

    // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.loop())
    {
        Info<< "Time = " << runTime.timeName() << nl << endl;

        #include "readPISOControls.H"
        #include "CourantNo.H"

        // Pressure-velocity PISO corrector
        {
            // Momentum predictor
```

②運動方程式

```
            fvVectorMatrix UEqn
            (
                fvm::ddt(U)
              + fvm::div(phi, U)
              + turbulence->divDevReff(U)
            );

            UEqn.relax();

            if (momentumPredictor)
            {
                solve(UEqn == -fvc::grad(p));
            }

            // --- PISO loop
```

③圧力方程式

```
            for (int corr=0; corr<nCorr; corr++)
            {
                volScalarField rAU(1.0/UEqn.A());

                volVectorField HbyA("HbyA", U);
                HbyA = rAU*UEqn.H();
                surfaceScalarField phiHbyA
                (
                    "phiHbyA",
                    (fvc::interpolate(HbyA) & mesh.Sf())
                  + fvc::ddtPhiCorr(rAU, U, phi)
                );

                adjustPhi(phiHbyA, U, p);

                // Non-orthogonal pressure corrector loop
                for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
                {
                    // Pressure corrector

                    fvScalarMatrix pEqn
                    (
                        fvm::laplacian(rAU, p) == fvc::div(phiHbyA)
                    );

                    pEqn.setReference(pRefCell, pRefValue);

                    if
                    (
                        corr == nCorr-1
                     && nonOrth == nNonOrthCorr
                    )
                    {
                        pEqn.solve(mesh.solver("pFinal"));
                    }
                    else
                    {
                        pEqn.solve();
                    }

                    if (nonOrth == nNonOrthCorr)
                    {
                        phi = phiHbyA - pEqn.flux();
                    }
                }

                #include "continuityErrs.H"

                U = HbyA - rAU*fvc::grad(p);
                U.correctBoundaryConditions();
            }
        }

        turbulence->correct();

        runTime.write();

        Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
            << "  ClockTime = " << runTime.elapsedClockTime() << " s"
            << nl << endl;
    }

    Info<< "End\n" << endl;

    return 0;
}
```

Figure 1 OpenFOAM 代表的な解析ソルバ（**pisoFOAM**）

2. 対象 C++型流体コードの概要

2.1 コードの概要とその課題

OpenFOAM は，OpenCFD 社が中心に開発した物理場の演算コード群である．その機能は，メッシュ作成等の前処理から流体・温熱・分子動力学・電磁流体・固体応力解析等の解析ソルバ群，結果処理や可視化まで多岐にわたっている．例えば，単層流非圧縮性解析ソルバ pisoFOAM（図 1）は①初期化，②運動方程式，③圧力方程式として構成されているが，解くべき物理方程式は，C++ オブジェクト指向により僅かな数十行のコードで記述されている．これより，利用者にとって非常に分かりやすく，必要な物理量の追加等も容易である．そのため，生産性の高い数値流体解析コードとして知られている．

この C++で書かれた OpenFOAM コードでは，膨大なクラス群を使うことにより，柔軟性および生産性が高くなっている．しかしながら，チューニングの観点ではCやFortran と比較して複雑となっている．例えば，境界処理や並列処理，大規模計算において，負荷の大きい箇所を性能ツールで分析しても，対象となるオブジェクトの関係が多岐に渡りホットスポットの特定が困難となっている．既存の HPC 分野の経験も十分に活かされない現状である．

図 2 には境界処理の多い実アプリケーションの並列性能の一例を示す．並列数 200MPI ではそこそこの性能を得られるが，並列数 800MPI ではその性能が急激に悪化している [3]．そのため，OpenFOAM で保有している並列処理プラットフォームや，使用している数値モデル等 [4]の詳細な検討が求められている．本論では，前者の並列処理プラットフォームについて検討を行う．

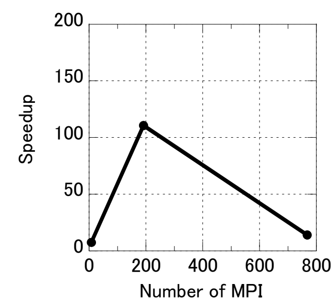


Figure 2 実アプリケーション性能の例

2.2 並列処理プラットフォーム

OpenFOAM の並列処理は空間的な領域分割手法を採用している．この手法は解析領域を小領域に分割して，それぞれの MPI プロセスに与えて並列分散処理を行い，計算高速化を行う．しかし，並列分散処理は，必ずしもそれぞれの小領域内の処理が独立に行われない．対象としているアプリケーションや使用している数値モデル，数値解法，処理の依存関係によって，それぞれの小領域から適宜に全ての小領域，または隣接の小領域とデータ交換を行うことが

ⓒ2015 Information Processing Society of Japan

必要である．データ交換量や交換回数はプロセス間の通信量や通信回数に比例し，アプリケーションの並列性能に大きく影響する．

そのため，領域分割手法では，比較的データ交換回数の多い全て小領域へのデータ交換（以下，全体データ交換）は，収斂残差値や理論値等の小さいデータ（短いメッセージ長）のみを扱う．一方，比較的大きいデータ（長いメッセージ長）を扱うのは，依存関係の強い隣接小領域からの境界面データ交換（以下，隣接データ交換）に限定する等，アルゴリズムを工夫することが重要である．なお，全体データ交換を実施する通信手法は集団通信に限定するものではなく，1 対 1 通信でも実現できる．データ交換形態とメッセージ通信形態の対応は一意ではない．

OpenFOAM の並列処理に関連するクラスと関数の定義は以下のフォルダーパスから参照できる．

(1) src/OpenFOAM/db/IOstreams/Pstreams
OpenFOAM クラス群に使用される特有の領域間のデータ交換形態クラス **UPstream**，**Pstream**, **PstreamBuffer** を定義している．

(2) src/Pstream
クラス **UPstream**，**Pstream**，**PstreamBuffer** とメッセージ通信ミドルウェア mpi や gamma 等と関係する関数を定義する（本論では MPI ミドルウェアのみを検討）．

また，解析ソルバ，コア部分（OpenFOAM-Core），データ交換形態とメッセージ通信ミドルウェアとの関係は階層構造として配置されている（図 4）．これより，OpenFOAM は利用者がミドルウェアを意識せずに，並列プログラムを書くことが可能な汎用性の高い並列プラットフォームである．ただし，大規模計算の性能に影響しやすい境界処理等はミドルウェアより高い階層にあるため，並列処理のチューニングや最適化が難しくなる点が挙げられる．

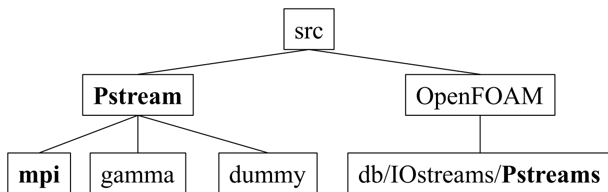


Figure 3 OpenFOAM の並列処理に関連するパス

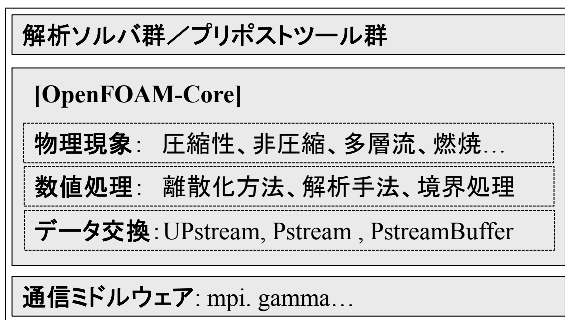


Figure 4 OpenFOAM の階層構造

2.3 並列処理の必要な領域間のデータ交換形態

表 1 に OpenFOAM で用いた領域間のデータ交換関数と機能，MPI プロセス間のメッセージ通信形態を示す．

隣接データ交換については，まず，基底クラス **UPstream** から派生クラス **UOPstream・UIPstream** が作られる．次に，**UOPstream**,・**UIPstream** が write・read 関数を定義する (UOPwrite.C，UIPread.C ファイルを参照)．これらの関数は「1 対 1 通信」のメッセージ通信機能に相当している．ただし，実際に Nonblocking/Blocking/Scheduler の変数によって MPI ミドルウェアの MPI_Send/MPI_Recv，または MPI_ISend/MPI_Irecv メッセージ通信モードを行う．

一方で，全体データ交換については，主にクラス **Pstream** とその派生クラス **PstreamBuffer** が使われる．**Pstream** は gatther/scatter 関数（**Pstream**::gatther/scatter）を定義している．関数名の通りに「集団通信」機能に相当している．ただし，実際に **Pstream**::gather/scatter は，MPI_Send/MPI_Recv の OpenFOAM で定義された特有なリニアまたはバイナリツリー通信形態で行われている．また，クラス **Pstream** は関数 exchange（**Pstream**::exchange）も定義しており，この関数は派生クラス **PstreanBuffer** から呼び出されている．**PstreamBuffer** と **Pstream**::exchange の特徴は節 3.3.2 で詳細に紹介するが，その機能は主に MPI_Alltoall に相当するものである．

さらに，クラス型とデータオペレーター形態を格納する reduce テンプレート関数が独立に定義されている．このテンプレート関数は MPI_Allreduce 機能に相当するものであり，実際のデータ交換時にデフォルトモードとして **Pstream**::gather/scatter の形態で行う．ただし，その一部のデータ型は MPI_Allreduce として行われている．

Table 1　OpenFOAM の領域間データ交換関数の概要

領域間データ交換関数	機能相当 MPI 関数
隣接データ交換	**1 対 1 通信**
UOPstream::write	MPI_Send/ MPI_Isend
UIPtream::read	MPI_Recv/ MPI_Irecv
全体データ交換	集団通信
Pstream::gather[1)]	MPI_gather
Pstream::scatter[1)]	MPI_scatter
Pstream::exchange[2)]	MPI_Alltoall
reduce, returnReduce[3)]	MPI_Allreduce[3)]
MPI プロセス間メッセージ通信形態	
1) MPI_Send/Recv のリニア/バイナリツリー通信	
*2) **Pstream**::gather/scatter[1)]+**UOPstream**::write/**UIPstream**::read*	
***PstreamBuffer** クラスの finishedSends 関数から呼び出し*	
*3) デフォルト：**Pstream**::gather/scatter[1)] の形態*	
一部のデータ型は MPI_Allreduce	

3. 実アプリケーションによる評価

本章では実アプリケーションを用いて評価した結果を述べる.

3.1 対象アプリケーションの概要

(1) 解析モデル

対象コードの性能特性を調べるために，建築分野に特有の問題である風洞測定部を再現したベンチマークモデルを用いた．解析領域は長さ 16m ×幅 3m×高さ 2m である．対象コードの Weak Scaling を調べるために，一つの MPI プロセスにおける計算格子数を 262,144 (=64×64×64) として，表 1 に示す 5 ケースの計算を実施した．なお，本論ではケース 2~4 の結果を中心に紹介する.

Table 2 解析モデル

No	分割	MPI	Tofu 座標	格子数
1	32×6×4	768	4×6×4	2.013E+08
2	64×6×4	1,536	8×6×4	4.027E+08
3	64×12×4	3,072	8×12×4	8.053E+08
4	64×12×8	6,144	8×12×8	1.611E+09
5	128×24×16	49,152	16×24×16	1.288E+10

(2) 対象解析ソルバと解析条件

解析は OpenFOAM-2.2.x の解析ソルバ pisoFOAM を用いた．数値解法は速度場で BiCG 法，圧力場で CG 法を採用した．また，計算ケースによって数値の収斂性が異なっており，相対的な比較が難しくなる．そのため，経験に基づいて BiCG 法の反復回数を 3 回，CG 法の反復回数を 100 回に固定した．これより全てのケースの MPI プロセスにおける計算量は同じになる．なお，解析の最初の 10 ステップの結果を評価する．解析スキームは文献 [4]を参照されたい.

(3) 使用計算機と計測方法

解析は「京」コンピュータで行った.「京」は，6 次元メッシュ／トーラスネットワークで構成された Tofu インターコネクトで計算ノード間を繋ぐ．この 6 次元で与えられた座標を Tofu 座標として，大きさ 2×3×2 の Tofu 単位で構成されている [5]．本論では，この単位に合わせて解析領域を分割した（表 2)．また，1 ノードあたり 8MPI プロセスとして完全 Flat-MPI 計算を実施した(「京」:8cores/node).

表 3 は実行環境である．アプリケーションの基本情報や MPI 情報等の測定には,「京」で整備された詳細プロファイラツール [6]を使用した．なお，本論では主に MPI 情報の平均値を用いて，アプリケーションの特性を評価した.

Table 3 実行環境

Compiler	**Fujitsu C/C++ Version K-1.2.0-16**
Build Option	C++ flag: -O3 -Xg
MCA	Execution option: Default

3.2 コード最適化の基本的な考え方

3.2.1 最適化の指針

図 4 より，OpenFOAM-Core と MPI ミドルウェアとのやり取りは OpenFOAM 特有のデータ交換クラスを通じて間接的に行われている．そのコアの高層部分から直接的にインライン展開を行い，MPI 関数を埋め込む等，個別のデータ交換や並列処理の最適化を行うことは，技術的に不可能な方法ではない．しかし，このような方法は OpenFOAM 本体の階層構造を壊し，コードのメンテナンス性や拡張性を失う．C++コードの利便性，また生産性という観点からこれは望ましくない.

本論では，OpenFOAM-Core の最低層部の「データ交換クラス」に着目し，柔軟性のある C++テンプレートの追加により，利用計算機や解析モデルに対応した MPI ミドルウェアを提供する．その性能を向上することで OpenFOAM 全体の並列性能を改善できると考える.

3.2.2 ハードウェア機能と解析ソルバを考慮した通信実装の指針

「京」コンピュータでは，MPI_Barrier 関数， MPI_Reduce 関数および MPI_Allreduce 関数の実行時に，Tofu インターコネクトのハードウェア機能として提供される高機能バリア通信機能を利用して，通信高速化を実現することができる．この機能は，メッセージの要素が 1 個であり，理論型・整数型・浮動小数点型・複素数型の限定的なデータ型ではなければならない．そのため，OpenFOAM での短いデータ長さの交換は高機能バリア通信機能を適用できるように，MPI ミドルウェアの利用関数とデータ型を合わせる必要がある．固有コードを明示的に変更せずに，特定のデータ型に対応したテンプレートの追加によって利用関数に合わせることが，C++では容易に実現できると考えられる.

一方で，対象としたアプリケーション（解析の目的）によって解析ソルバが既に選定されてあり，解析領域が変化する場合の条件等，境界処理にも必要最小限のデータ交換プロセスが既に定められている．そのため，解析ソルバに応じて適切な並列プロセスを与える必要があり，領域間のデータ交換形態の最適化等に，検討の余地がある.

これらにより，本論ではまず，Tofu 高機能バリア通信機能を最大限に活かすために，表 1 に示すように OpenFOAM の低層部にある reduce テンプレート群にデータ型に合わせたテンプレートを追加した．また，解析ソルバに適切なデータ交換形態を与えるために，OpenFOAM の高層部にある解析ソルバに新たなテンプレートを加えて，低層部で行われているメッセージ通信プロセスを直接呼び出した.

これより，表 3 には本論に用いた解析コードの概要を示す．STD 版は OpenFOAM の標準版である．STD+版は Tofu の高速バリア通信を意識したテンプレートを追加したものである．また，NEW 版はデータ交換形態を改良したテン

プレートを新たに追加したものである．これらの解析コードの性能特性を次節で評価する．

Table 4 対象解析コードの概要

コード	備考
STD 版	・OpenFOAM 標準版
STD+版	・STD 版の利用 ・Reduce テンプレートの追加 ・Tofu の高速バリア通信利用
NEW 版	・STD+版の利用 ・PstreamBuffer データ交換形態の変更 ・通信形態改良（MPI::Send/Receive）

3.3 コードの測定結果および分析

3.3.1 Tofu 高機能バリア通信機能の効果

a) 実施方法

OpenFOAM のデータ型は整数型・浮動小数点型から配列やテンソル等までクラスとして定義されている．その一部のデータ基本型は表 5 である．MPI_Allreduce 機能に相当する全体データ交換については，格納されるデータ型 T とデータオペレーター形態 BinaryOp の reduce テンプレート（図 5）として行われている．

OpenFOAM のデータ交換は,「京」の Tofu 高機能バリア通信機能を意識していないテンプレート構文で実装されている．その通信アルゴリズムは OpenFOAM の特有なリニアまたはバイナリツリー通信として行われていることが，図 5 の実装コード（linearCommunication, treeCommunication 変数名）から読み取れる．

Tofu 高機能バリア通信機能を最大限に活用するために，当該ソースに理論型・整数型・浮動小数点型・複素数型等，高機能バリア通信機能に適合する 1 個のデータを格納したテンプレートを追加した．図 6 は最大値を求める浮動小数点型（scalar）のテンプレートの一例である．このような記述構文により allReduce 関数を通じて浮動小数点型データの 1 個の MPI_Allreduce を実現できる．

b) 結果と考察

表 6 には，ケース 4 (6,144MPI)を対象として，OpenFOAM の標準版（**STD** 版）と，Tofu 高機能バリア通信機能を意識して新たなテンプレートを追加した **STD+**版の結果を示す．比較情報は Allreduce, Send/Recv の呼び出された回数，Tofu 高機能バリア通信の回数と全体実行時間である．

STD+版の Allreduce 回数の増加，Send/Recv 回数の減少を確認できる．これはデータ型に合わせた新たな reduce テンプレートの追加により，OpenFOAM の一部の Send/Recv 通信が Allreduce の通信に移された．また，Tofu バリア通信情報より Allreduce 通信は全て高機能バリア通信機能として実施されていることが分かる．

Table 5 OpenFOAM 一部の基本データ型

データ型	意味	相当 C++データ型
label	整数	int
scalar	浮動小数点型	double
vector	3 次元配列	{double, double, double}
vector2D	2 次元配列	{double, double}
bool	理論型	—

```
//src/OpenFOAM/db/IOstream/Pstream/PstreamReduceOps.H//
template<class T, class BinaryOp>
void reduce
(
    T& Value,
    const BinaryOp& bop,
    const int tag = Pstream::msgType()
)
{
    if (UPstream::nProcs() < UPstream::nProcsSimpleSum)
    {
        reduce(UPstream::linearCommunication(), Value, bop, tag);
    }
    else
    {
        reduce(UPstream::treeCommunication(), Value, bop, tag);
    }
}
```

*※**T**: 格納されるデータ型*

***BinaryOp**: 格納されるオペレーター形態*

Figure 5 デフォルト reduce テンプレート構文

```
//src/OpenFOAM/db/IOstream/Pstream/PstreamReduceOps.H//
void reduce
(
    scalar& Value,
    const maxOp<scalar>& bop,
    const int tag = Pstream::msgType()
);
//src/Pstream/UPstream.C//
void Foam::reduce(scalar& Value, const maxOp<scalar>& bop, const int tag)
{
    allReduce(Value, 1, MPI_DOUBLE, MPI_MAX, bop, tag);
}
```

Figure 6 浮動小数点型の **reduce** テンプレートの追加例

Table 6 比較検討（ケース 4, 6,144MPI）

コード	**STD** 版	**STD+**版
呼び出された回数		
Allreduce	AVG: 6,581 MAX: 6,581 MIN: 6,581	AVG: 6,641 MAX: 6,641 MIN: 6,641
Send	AVG: 130 MAX: 6,975 MIN: 65	AVG: 8 MAX: 6,182 MIN: 4
Recv	AVG: 130 MAX: 6,975 MIN: 65	AVG: 8 MAX: 6,182 MIN: 4
Tofu Barrier Communication		
Allreduce	6,581	6,641
アプリケーションの全体実行時間		
経過時間(s)	324.58	319.90

結果的に，アプリケーションの全体実行時間は少なくなり，計算の高速化を実現できた．ちなみに，**STD+**版の Allreduce 数は **STD** 版の数と比べてその差が小さいと見られる．これは，本論で対象とした OpenFOAM バージョンでは，一部のオペレーター形態で既に同様なテンプレート構文が記述されていることによるものである．また，本検討ケースでは倍精度浮動小数点型データが支配的であり，その型が STD 版に対応していると言える．バージョンや検討対象モデルによってその差が大きく変化すると思われる．

3.3.2 領域間データ交換形態の改良とその効果

a) 実施方法

OpenFOAM は主に **UIPstream・UOPstream**，**Pstream**，**PstreamBuffer** の 3 種類のクラスを通じて領域間のデータ交換を行う（節 2.3 また表 1 を参照）．

特に，クラス **PstreamBuffer** は全体データ交換に適用し，**Pstream**::exchange 関数を用いて実装されている．図 7 はデータ交換の構造である．転送元と転送先が格納された 2D 配列テーブル「sizes」（サイズは N^2，N : MPI プロセス数）があり，本テーブルにより **Pstream**::exchange 関数を用いて，全プロセスに対してデータを転送するループ構造となっている．ちなみに，このデータ交換の実装方法は，他領域への依存関係データサイズをまず 2D 配列テーブル sizes に格納して，combineReduce 関数を通じて全プロセスにデータを転送した後，他領域から受信した依存関係データサイズによってデータ有無，受信を行うか否かは判断する．

このようなデータ交換方法は MPI_Alltoall の通信形態と相当するデータ交換方法である（図 8）．このデータ交換は解析領域の多種多様な変化や不特定な領域境界についても対応できるように実装されている万能なデータ交換である．

しかし，2D 配列テーブル「sizes」は N^2（N: MPI プロセス数）のサイズに比例しているため，そのテーブルサイズは大規模な MPI 数になると著しく増加すると推測される．また，N^2 と全体プロセスを跨ぐ通信であり，並列数の 2 乗に比例して通信時間も増大する．

しかしながら，利用者側から解析ソルバを選定する時，例えば，本論で対象とした解析ソルバ pisoFOAM の選定については既に解析領域や分割された小領域が不変とする条件で行われており，隣接依存関係が定められ，領域間は PstreamBuffer の全体データ交換形態を行う必要がない．

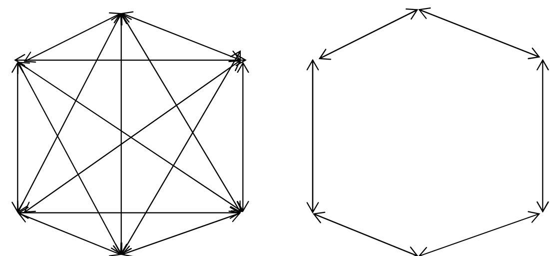
従って，本論では，クラス **PstreamBuffer** を用いた **Pstream**::exchange 関数の新たなテンプレートを加えることにより，必要最小限の隣接データ交換形態としたクラス **PstreamBuffer** を構築した．具体的には図 7 の 2D 配列テーブル「sizes」を使わず，sendBuf サイズから recvBuf サイズに算出する等， **PstreamBuffer** は現コードの全体データ交換形態から隣接データ交換形態に変更することで実装した（図 8 を参照）．

```
//src/OpenFOAM/db/IOstream/Pstream/PstreamBuffers.C//
template<class Container, class T>
void Pstream::exchange
(
    const List<Container>& sendBufs, // Send buffer data    O(N)
    List<Container>& recvBufs,       // Receive buffer data O(N)
    labelListList& sizes,            // 2D Array of Send/Recv
                                     //      buffer sizes O(N^2)
    ...
)
{
    ................................
    // Collective Communication using Pstream::gather/scatter
    combineReduce(sizes, ..., ...);
    ................................
    {
        forAll(sizes, procI) //~ procI=0,N-1
        {
            label nRecv = sizes[procI][...];
            if (... nRecv > 0)      // Do action if available
            {
                .........................
                UIPstream::read(...);
            }
        }

        forAll(sendBufs, procI)  //~ procI=0,N-1
        {
            //Send data if available
            if (... sendBufs[procI].size() > 0)
            {
                if(!UOPstream::write(...)){}
            }
        }
        ..........................
    }

    ............................
}
```

Figure 7 クラス **PstreamBuffer** の **Pstream:**:exchange 関数



(a) 全体データ交換（現）　(b) 隣接データ交換（**NEW**）

Figure 8 **PstreamBuffer** データ交換形態のイメージ図

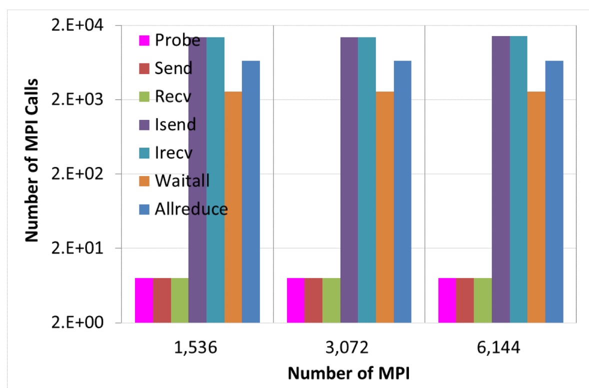
b) 結果と考察

まず，領域間データ交換形態を変更しない既存コードの性能特性について述べる．

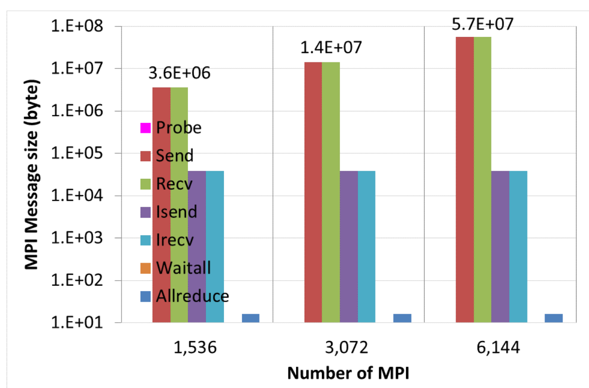
図 9 には，**STD+**版のケース 2,3,4（1,536MPI; 3,072MPI; 6,144MPI）を対象として，アプリケーション全体の MPI 命令の特性を示す．なお，**STD** 版は **STD+**版と同様な特性を持っているため，ここでは省略する．

図 9(a)より，**STD+**版では MPI プロセス当たりの Probe, Send/Recv, Isend/Irecv, Waitall，Allreduce 関数が呼び出された回数はいずれも MPI 数に依存せず，一定値となっている．一方，図 9(b)から，Allreduce や Isend/Irecv のメッセージの平均サイズは一定値であるが，呼び出された回数の少ない Send/Recv の平均サイズは，1,536mpi で 3.6E+6byte，3,072mpi で 1.4E+07byte，6,144mpi で 5.7E+07byte である．

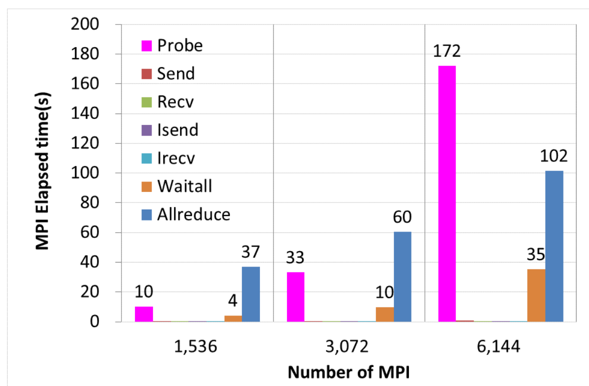
ⓒ2015 Information Processing Society of Japan



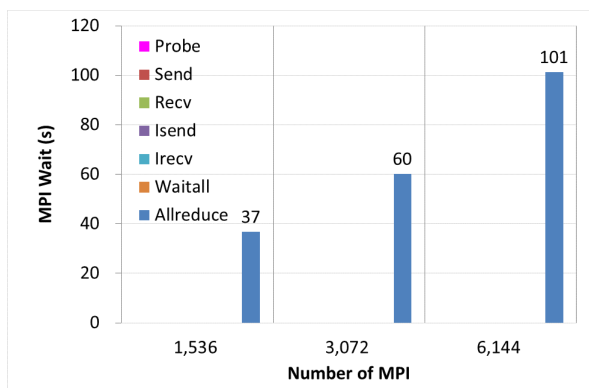
(a) 呼び出された平均回数



(b) メッセージの平均サイズ

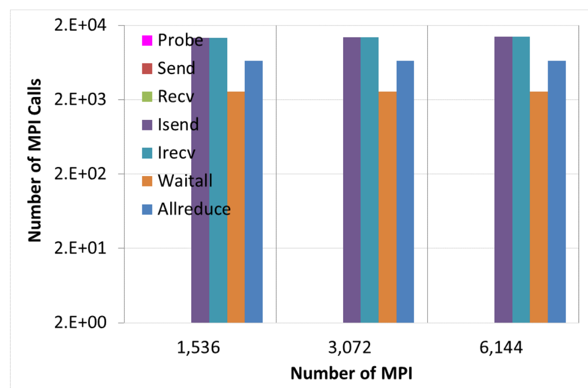


(c) 平均経過時間

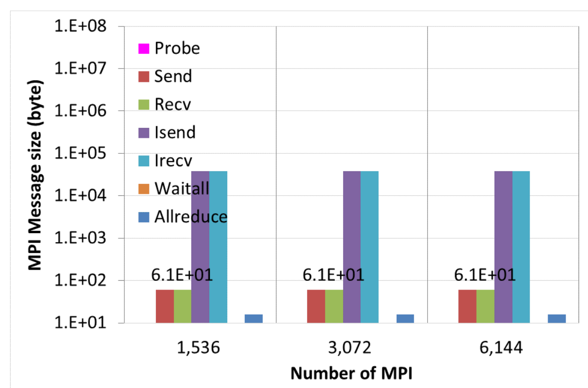


(d) 平均待ち時間

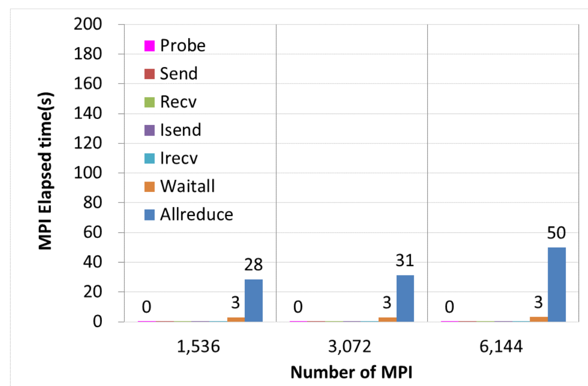
Figure 9 **STD+**版の全体 MPI 通信特性



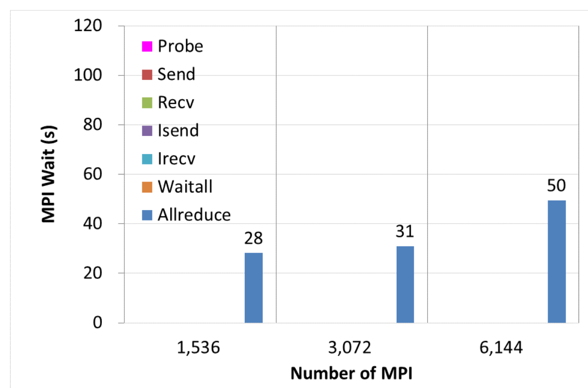
(a) 呼び出された平均回数



(b) メッセージの平均サイズ



(c) 平均経過時間



(d) 平均待ち時間

Figure 10 **NEW** 版の全体 MPI 通信特性

ⓒ2015 Information Processing Society of Japan

その平均サイズが爆発的に 増大しており，前節に述べた 2D 配列テーブル「sizes」のサイズである N^2（N: MPI プロセス）に比例している．これより現状の OpenFOAM はより大規模並列計算に適していないと言える．

図 9(c)，(d)は STD+版のそれぞれの MPI 命令の平均経過時間（MPI Elapsed time）と待ち時間 (MPI Wait)である．経過時間と待ち時間との差から Allreduce の処理時間は殆どないことが分かる．また，Allreduce の待ち時間と Probe，Waitall の経過時間は MPI 数に従い増加している．特に，Probe 経過時間は Waitall 経過時間や Allreduce の待ち時間と比べてその増加勾配が大きい．これらの時間の増大はアプリケーションの並列性能を悪化させる傾向となる．

なお，節 3.3.2 よりクラス **PstreamBuffer** は 2D 配列テーブル「sizes」のデータ交換を行う時に **Pstream**:gather/scatter を通じて MPI_Send/ MPI_Recv 関数を呼び出している．また，ソースコード UIPread.C ファイルから，MPI_Recv 関数を行う前に MPI_Probe 関数を実行し，受信完了を持ち合わせ，MPI_Get_count で受信データサイズを取得して受信バッファーメモリを確保する等を特定できる．すなわち，Probe 経過時間は 2D 配列テーブル「sizes」のサイズの爆発的増大によるものが分かる．

次に，図 10 は **NEW** 版の結果である．図 10(a)より MPI の呼び出された回数は **STD+**版の結果と同じである．しかしながら，**STD+**版の平均メッセージサイズの爆発的な増大に対して **NEW** 版のメッセージサイズは激減した（図 10b）．これは **STD+**版の **PstreamBuffer** データ交換形態における複数 MPI プロセスによる相互通信に対して，**NEW** 版の新たな通信形態では必要最小限隣接通信を行うことによるものである．

図 10(c)，(d)は **NEW** 版のそれぞれの MPI 命令の平均経過時間（MPI Elapsed time）と待ち時間 (MPI Wait)である．**STD+**版の結果と比べていずれの時間も減少している．特に，Probe の経過時間はほぼ無くなっている．また，Waitall の経過時間はほぼ一定になっているため，ISend/IRecv による通信はコード内の演算量によって隠ぺいされると推測している．さらに，Allreduce 回数とメッセージサイズは **STD+**の結果と同じであるが，Allreduce の待ち時間は **STD+**版と比べて半減した．これは，**STD+**版において非常に大きなメッセージサイズの Send/Recv が行われており，その完了を持つ Probe の経過時間が大きくなることと，メッセージ長の不均一や送受信待ち合わせ（文献 7 を参照）による処理全体の Imbalance も大きくなることが要因であると推測される．**NEW** 版ではその分が消されて，Imbalance も解消された等，全体的に Allreduce の待ち時間も軽減したと考えられている．いずれもこれらの関係は今後に詳細に調べる予定である．

図 11 には **STD+**版と **NEW** 版の計算ノードにおけるメモリ使用量を示す．**STD+**版においてはメッセージサイズの増大によりそのメモリ使用量が増加している．結果的に 6,144MPI モデルより大きな解析モデルは，2D 配列テーブル「sizes」がメモリ容量を圧迫にして，計算が困難になってくる．

また，図 12 はそれぞれの解析コードの実行効率比率を示す．ここで，STD+版(runT)は PstreamBuffer データ交換形態を除いた結果（複雑な境界処理が少ないアプリケーションに相当）である．本論で提案した NEW 版は OpenFOAM の並列性能を向上させることが確認できた．

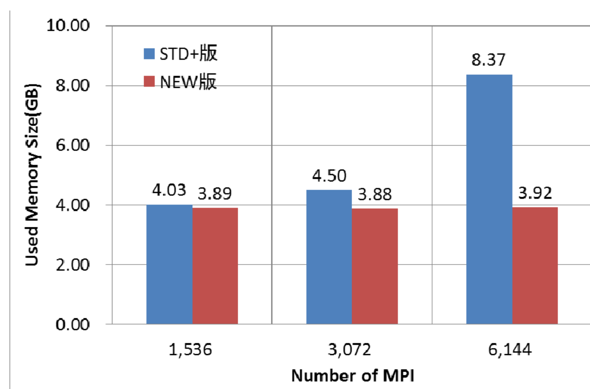


Figure 11 計算ノードにおけるメモリ使用量

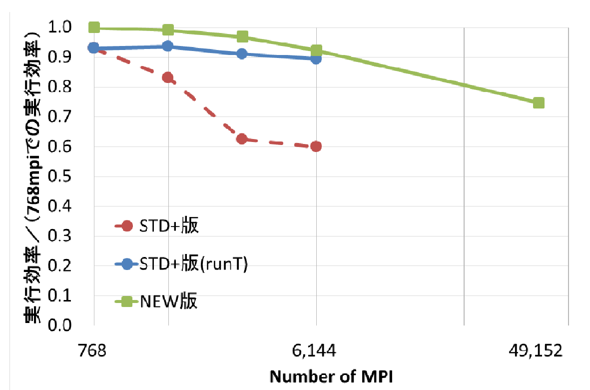


Figure 12 実行効率の変化

4. まとめ

本論では C++言語オープンソース OpenFOAM を対象として，利用している MPI プラットフォームの特徴とその課題を述べた．まず，「京」コンピュータの Tofu 高機能バリア通信機能を活用して，データ型に合わせたテンプレートの追加による全体実行時間の軽減を確認した．次に，OpenFOAM 特有の **PstreamBuffer** データ交換形態を必要最小限の隣接通信に改良し，送信・受信バッファサイズおよび通信時間が減少した．上記 2 点の改良によって，アプリケーション全体の実行効率が大幅に向上した．

謝辞
本検討は，理化学研究所のスーパーコンピュータ「京」を利用 して得られたものである（課題番号: hp150031）．ここに記して謝意を表する．

ⓒ2015 Information Processing Society of Japan

参考文献

1) OpenFOAM：http://www.openfoam.com/
2) ファム　バン　フック，野津剛，菊池浩利，日比一喜：建築分野の数値流体解析における大規模計算，TSUBAME ESJ Vol.8 , pp.15-20, 2013
3) ファム　バン　フック，その他：超大規模数値流体解析による建物局部風圧の予測とその制御システムの開発，第 1 回成果報告会，「京」を含む産業利用枠（実証利用）課題，2014
4) ファム　バン　フック，その他：LES の SGS モデルによる一様流中のセットバックした建物の局部風圧の検討，風工学シンポジウム論文集，Vol.23，pp.463-468 ,2014.
5) Fujitsu: Parallelnavi Technical Computing Language, MPI 使用手引書，2015
6) Fujitsu: Parallelnavi Technical Computing Language,プロファイラ使用手引書，2015
7) 井上義昭 :「京」における OpenFOAM の性能評価，平成 26 年度「京」における高速化ワークショップ発表資料，2014

ⓒ2015 Information Processing Society of Japan