Vol.2019-HPC-168 No.6
2019/3/5

MYX：マルチSPMDプログラミングモデルにおける実行時正当性チェック

辻 美和子[1,a)] 村井 均[1,b)] 佐藤 三久[1,c)] 朴 泰祐[2,d)] Petiton, Serge[3,4,5,e)] Emad, Nahid[6,f)]
Protze, Joachim[7,g)] Terboven, Christian[7,h)] Müeller, Matthias S.[7,i)]

概要：分散並列プログラミングの複雑化にともない，事前もしくは実行時に並列プログラムの正当性をチェックするソフトウエアやライブラリが提案されている．MUST はスケイラブルな正当性チェックライブラリであり，MPI に加えて，指示文ベースの分散並列プログラミング言語 XcalableMP (XMP) をサポートする．本稿では，MUST による XMP プログラムの正当性チェックを，複数の XMP プログラムを協調動作させるマルチ SPMD (mSPMD) プログラミングモデルにおいて適用する．本稿では，mSPMD プログラミング開発・実行環境を拡張し，mSPMD プログラミングモデルで実行されるワーカ・プログラム内のユーザ記述のタスク部に対して，MUST による正当性チェックをサポートする．予備実験により，mSPMD に対する MUST の正当性チェックを動作確認するとともに，MUST 適用のオーバーヘッドについても検証した．

1. はじめに

分散並列プログラミングの複雑化にともない，事前もしくは実行時に並列プログラムの正当性をチェックするソフトウエアやライブラリが提案されている．MUST はスケイラブルな正当性チェックライブラリであり，MPI に加えて，指示文ベースの分散並列プログラミング言語 XcalableMP (XMP) をサポートする．本稿では，MUST による MPI や XMP プログラムの正当性チェックを，複数の SPMD プログラムを協調動作させるマルチ SPMD プログラミングモデルにおいて適用する．

図 1 にマルチ SPMD プログラムおよびその正当性チェックがどのように実現されるかを示す．複数の SPMD プログラムを制御するワークフローのために，科学技術計算のためのワークフロー開発実行環境である YML[1], [2] が用いられる．タスクの記述のために，MPI に加えて，指示文に基づく分散並列プログラミング言語である XcalableMP (XMP) [5] を用いることができる．従来の YML では逐次言語で記述されたタスクを対象としていたが，われわれの YML の拡張により，逐次言語に XMP 指示文を挿入することでタスクを SPMD で実行することが可能となった [9], [12]．MUST は [3], [4], [8] は，MPI におけるプログラムエラーを検出し，ユーザに報告するライブラリである．MPI のみならず XMP についてプログラムエラーの検出を行うために，XMP は MUST に対して XMP ツールイ

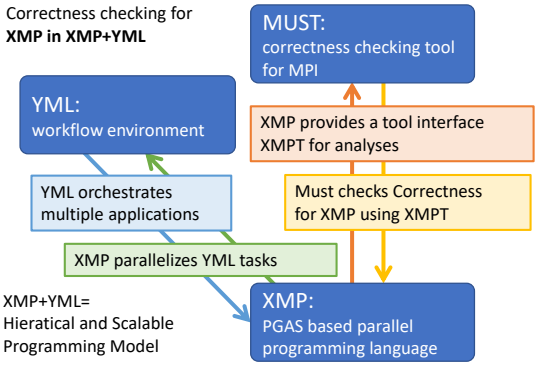


図 1 マルチ SPMD プログラムに対する正当性チェックの概要

1 理化学研究所計算科学研究センター
RIKEN Cenetr for Computational Science
2 筑波大学計算科学研究センター
Cenetr for Computational Sciences, University of Tsukuba
3 Maison de la Simulation
4 Centre National de la Recherche Scientifique
5 Université de Lille
6 Université de Versailles
7 RWTH Aachen University
a) miwako.tsuji@riken.jp
b) h-murai@riken.jp
c) msato@riken.jp
d) taisuke@cs.tsukuba.ac.jp
e) Serge.Petiton@univ-lille1.fr
f) nahid.emad@uvsq.fr
g) protze@itc.rwth-aachen.de
h) terboven@itc.rwth-aachen.de
i) mueller@itc.rwth-aachen.de

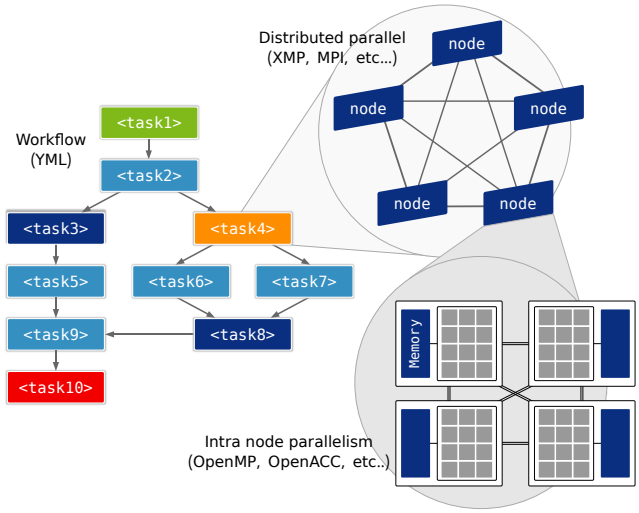


図 2　マルチ SPMD (mSPMD) プログラミングモデルの概要

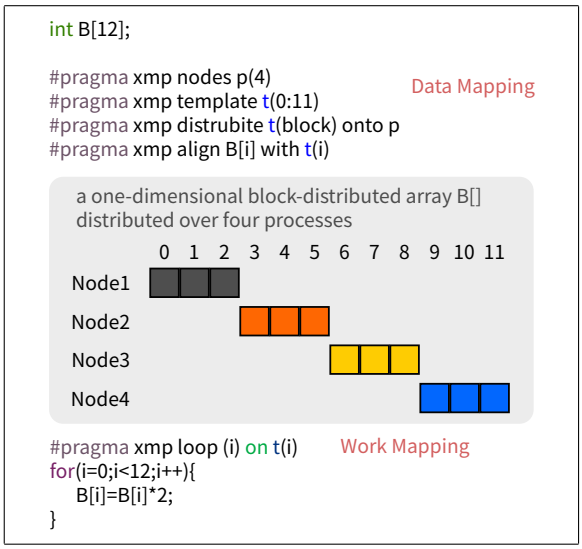


図 3　XMP により記述されたソースコードの例

ンターフェスを提供する．MUST はこれを用いて XMP のエラーを検出する [6]．

2. 背景 ： マルチ SPMD プログラミング開発実行環境と正当性チェックライブラリ MUST

2.1 マルチ SPMD プログラミング開発実行環境

OpenMP/MPI ハイブリッドプログラミングなどの MPI + X は，現在の大規模 HPC アプリケーション開発において，通信の効率化のための重要な概念となっている．しかしながら，ノード内にも NUMA などの複雑なトポロジを持つ数十コアものメニーコアノードからなる大規模システムをより効率的に利用するためには，MPI + X のみならず，X_1 + MPI + X_2 などのさらなる階層性を持つプログラミングモデルが必要である．

著者らは，大規模かつ階層的なアーキテクチャのための X_1 + MPI + X_2 プログラミングモデルとして，ワークフロー + 分散並列 + スレッド並列を組み合わせて利用するマルチ SPMD (mSPMD) プログラミングモデルを提案し，mSPMD プログラミングモデルに基づくプログラムの開発・実行環境を開発してきた [9], [10], [12]．

図 2 に mSPMD プログラミングモデルの概要を示す．従来のワークフロープログラミングモデルにおいては，ワークフローのタスクは逐次プログラムとして実行されるが，mSPMD プログラミングモデルにおいては，ワークフローのタスクは分散並列プログラムもしくは分散並列とスレッド並列のハイブリッドプログラムとして実行される．ワークフローを実行するために YML が用いられ，ワークフロー内のタスクを記述するために MPI に加えて指示文に基づく並列プログラミング言語 XcalableMP (XMP) がサポートされる．

YML[1], [2] は，Delannoy らによって開発された科学技術計算のためのワークフロー開発実行環境であり，タスクどうしの依存関係を記述するワークフロー記述言語 YvetteML をサポートする．YML は YvetteML ワークフロー・ソースコードを解釈してタスクどうしの依存関係を示す DAG を生成し，DAG に従ってタスクを実行する．オリジナルの YML は，逐次言語で記述されたタスクを P2P 環境や小規模クラスタで実行することを想定していたが，著者らによる拡張によって並列言語で記述されたタスクの大規模システムでの実行が可能になった．また，これをサポートするミドルウェアについても開発した．

mSPMD プログラミングモデルで用いられるミドルウエア OmniRPC-MPI[11] は，Remote Procedure Call (RPC) をサポートするライブラリ OmniRPC [7] の拡張である．OmniRPC-MPI ミドルウェアは，ワークフロー・スケジューラからリモートプログラムの起動やリモートプログラムでのタスク実行などの依頼を受け，それらを MPI_Comm_spawn や MPI_Send などの MPI 関数を用いて実現する．

XcalableMP (XMP) [5] は，指示文に基づく分散並列プログラミング言語である．XMP コンパイラは，ソース to ソースのコンパイラであり，XMP 指示文とベース言語 (C もしくは Fortran) で記述されたソースコードを，XMP ランタイムライブラリ呼出しを含むベース言語のソースコードに変換する．XMP ランタイムライブラリは，通信レイヤとして主に MPI を用いて通信を行う．**図 3** に，XMP により記述された分散並列のソースコードの例を示す．XMP では，**template** と呼ばれる仮想配列の **nodes** 上への分散を定義し，**align** 文を用いて配列と template と対応付けることで，配列のノード上への分散を定義する．また，処理の分散は，**loop** と template を用いて定義される．

mSPMD プログラミングモデルの利点として，

- 大規模分散並列プログラムをワークフローの枠組で中規模プログラムに分割し，後に結果を統合することで，

通信のオーバーヘッドを防ぐことができる
- ワークフローにおいてボトルネックとなっていたタスクを分散並列を用いて並列化し，全体を高速化する
- 既存の並列プログラムや並列ライブラリをワークフローを用いて組み合わせることで，複雑なアプリケーションを平易に実装できる

などが挙げられる．

2.2 正当性チェックライブラリ MUST

MUST[3], [4], [8] は，MPI におけるプログラムエラーを検出し，ユーザに報告するライブラリである．複雑な MPI アプリケーションにおいては，プログラムエラーが頻発すると考えられることから，MUST ライブラリはアプリケーション開発者の負担を大幅に軽減することができる．

MUST は，PnMPI ライブラリを用いて MPI 関数呼出をインターセプトし，それらを MPI スタンダードに基づいて解釈することでエラーの検出を行う．引数の不正などのローカルエラー検出に加えて，データタイプの非一致やデッドロックなどのノンローカルなエラー検出もサポートされる (**図 4**) ．デッドロックなどの依存関係の検出は，AND ⊕ OR と呼ばれる AND-OR モデルを簡略化したグラフモデルを用いて行われる．

MUST では，スケーラブルな正当性チェックを実現するために，Tree-Based Ovewrlay Network (TBON) を提案し，分散してチェックを行っている [3]．木構造を用いることで，MPI プロセス数に対するチェックの回数を減らすことができる．**図 5** に，4 プロセスで実行される MPI の集合通信の整合性チェックに対する MUST の TBON 上での動作を示す．木の葉は MPI プロセスを，中の数字はプロセス番号を示す．中間の T_0, T_1, T_2 は MUST ツールを示す．円で示された $R_0 \sim R_3$ は各 MPI プロセスにおける MPI イベントを示す．MPI 関数が呼び出されると，図の右にあるように，各 MPI プロセスはイベントをツール T_0 および T_1 にフォワードする．左図のように，T_0, T_1 は，受け取った MPI イベントの整合性 — たとえばルートランクや型の一致 — を調査し，問題がなければ受け取ったイベントのうち 1 つを Representative として，T_2 にフォワードし，エラーがあれば報告する．同様にして，T_2 も受け取ったイベントを比較し，エラーがあれば報告する．

MUST を利用して実行時にアプリケーションの正当性チェックを行うためには，MPI の実行コマンド `mpiexec` を `mustrun` に置き換える：

```
mpiexec -np 4 application.exe
→ mustrun -np 4 application.exe
```

`mustrun` は以下を行うスクリプトファイルである：

(1) 指定されたアプリケーションに MUST を適用するためのコード生成し，これらをコンパイルしてダイナミックライブラリを作成
(2) 環境変数の設定
(3) `mpiexec` の実行

(2) で `mustrun` が指定する主な環境変数としては以下がある:

`LD_LIBRARY_PATH`: MUST ライブラリへのパス
`DYLD_LIBRARY_PATH`: MUST ライブラリへのパス
`PNMPI_LIB_PATH`:PnMPI ライブラリへのパス，および (1) でつくられたライブラリへのパス
`XMP_TOOL_LIBRARIES`:XMP ツールを使用するためのライブラリ `libxmpAdapter.so` を絶対パスで指定
`LD_PRELOAD`:実行時にプレロードされる PnMPI ライブラリ `libpnmpi.so` を絶対パスで指定
`PNMPI_CONF`:(1) でつくられた設定ファイルを絶対パスで指定

MUST は正当性チェックのために 1 プロセスを占有する．以降は，これを MUST プロセスと呼ぶ．ユーザが `mustrun` で n プロセスを指定した場合，`mustrun` は $(n+1)$ プロセスで `mpiexec` を実行する．PnMPI ライブラリによって，ユーザ定義の MPI_COMM_WORLD は，新たな n プロセスのコミュニケータに置き換えられる．以降は，この n プロセスからなるコミュニケータを USER_COMM_WORLD と呼ぶ.[*1]

`mustrun` が終了すると，MUST は `MUST_Output.html` ファイルと `MUST_Output-files` フォルダを生成する．前者はエラーの概要であり，後者には各エラーの詳細な情報と MPI 関数呼出の依存関係グラフが納められる．いずれも web ブラウザから確認することができる．MUST による出力 `MUST_Output.html` の例を**図 6** 示す．

2.3 XMP ツールインターフェスと MUST による XMP の正当性チェック

Protze らは，XMP に対する正当性チェックをサポートするよう，XMP および MUST をそれぞれ拡張した [6].

XMP においては，実行時に MUST をはじめとする解析ツールに対して必要な情報を送るためのインターフェース・XMPT が導入された．XMPT は，対応する指示文に関するすべての情報 (ディスクリプタ) を引数として渡すインターフェースである．MUST などの解析ツールの開発者は，各指示文に対してディスクリプタを用いて解析を行うコールバック関数を実装する．XMP の初期化関数 XMP_Init は，これらのコールバック関数を，対応する XMP 指示文に登録する．アプリケーションの実行時に，対象となる指示文に到達した場合，XMP ランタイムライブラリは登録されたコールバック関数を呼び出す．

[*1] USER_COMM_WORLD は説明の簡単のために用いた仮名である．実際には，MUST が解析用に用いるコミュニケータは，専用のインターフェスを用いて取得する

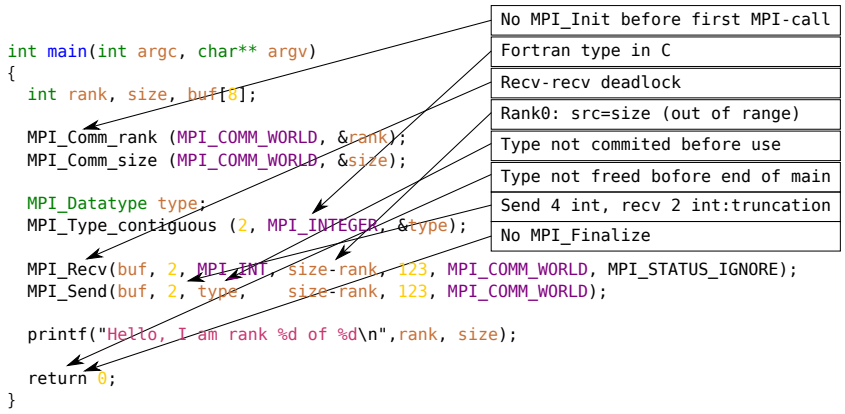


図 4　MUST で検出できる MPI プログラムの誤りの例

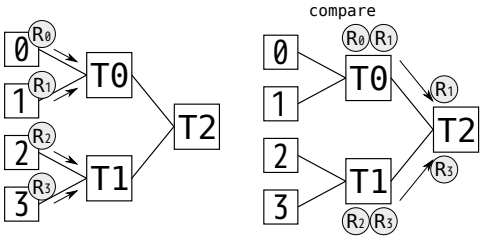


図 5　MUST の TBON での動作イメージ

MUST Output, starting date: Tue Jan 22 19:28:40 2019.

Rank(s)	Type	Message
1	Error	A send and a receive operation use datatypes that do not match! Mismatch occur…

Details:

Message	From	References
A send and a receive operation use datatypes that do not match! Mismatch occurs at (MPI_INTEGER) in the send type and at (MPI_INT) in the receive type (consult the MUST manual for a detailed description of datatype positions). A graphical representation of this situation is available in the file named "MUST_Output-files/MUST_Typemismatch_1.dot". Use the dot tool of the graphviz package to visualize it, e.g. issue "dot -Tps MUST_Output-files/MUST_Typemismatch_1.dot -o mismatch.ps". The graph shows the nodes of the involved Datatypes that form the root cause of the type mismatch. The send operation was started at reference 1, the receive operation was started at reference 2. (Information on communicator: MPI_COMM_WORLD) (Information on send of count 1 with type:MPI_INTEGER) (Information on receive of count 1 with type:MPI_INT)	Representative location: call MPI_Recv (1st occurrence)	References of a representative process: reference 1 rank 0: call MPI_Send (1st occurrence) reference 2 rank 1: call MPI_Recv (1st occurrence)

3	Error	A send and a receive operation use datatypes that do not match! Mismatch occur…
0	Error	A send and a receive operation use datatypes that do not match! Mismatch occur…
0-3	Warning	You requested 12 threads by OMP_NUM_THREADS but used MPI_Init to start your ap…
0-3	Error	Argument 1 (comm) is an unknown communicator where a valid communicator was ex…
3	Error	There are 16 communicators that are not freed when MPI_Finalize was issued, a …
1	Error	There are 16 communicators that are not freed when MPI_Finalize was issued, a …
2	Error	There are 16 communicators that are not freed when MPI_Finalize was issued, a …
0-3	Error	There are 2 operations that are not freed when MPI_Finalize was issued, a qual…
0	Error	There are 16 communicators that are not freed when MPI_Finalize was issued, a …

MUST has completed successfully, end date: Tue Jan 22 19:28:41 2019.

図 6　MUST の出力のスクリーンショット

MUST には，各 XMP 指示文について，XMPT より提供される情報を用いて，正当性チェックを行うコールバック関数が実装されている．XMP ランタイムライブラリにより呼び出された MUST のコールバック関数は，XMP ディスクリプタから必要な情報を取得し，XMP プログラムを解析する．例えば，図 3 で示したループ指示文については，**template** の大きさとループの回転数などがチェックされる．

3. マルチ SPMD プログラミングモデルにおける MUST による実行時正当性チェックの実装

本稿では，マルチ SPMD プログラミングモデルにおいて，各 SPMD に対する MUST を用いた実行時正当性チェックを提供する．

3.1 提供する機能

mSPMD プログラミングモデルは，ワークフロー・スケジューラ，ミドルウエア，リモートプログラムなどから構成され，リモートプログラムはワークフロー・スケジューラと通信を行う制御部とユーザ記述のタスク部からなる．これらの要素間および要素内の通信は，基本的にすべて MPI もしくは XMP で記述されている (**図 7**)．本稿では，アプリケーション開発者が記述したタスクの振る舞いについてのみ，MUST による正当性チェックを実施するものとした．図 7 において，黒文字で記されている MPI 関数 —リモートプログラムを起動するための MPI_Comm_spawn や，リモートプログラムに実行すべきタスクを指示する MPI_Send など —には正当性チェックを実施しない．一方，図で赤文字で書かれたタスク内部の通信や XMP によるワークロード分散のための指示文などの正当性は，MUST によって解析される．

3.2 実装の詳細

3.1 で述べた機能を実現するためには

- 各 SPMD タスクに MUST によるチェックを適用する
- 上記以外に MUST によるチェックを適用しないようにする

する必要がある．

一般的な MPI ライブラリの実装においては，MPI_Send などの MPI ではじまる関数 (以下，MPI 関数と呼ぶ) は，実際に通信を行う PMPI ではじまる関数 (以下 PMPI 関数と呼ぶ) のラッパーである．MPI 関数はウィークシンボル

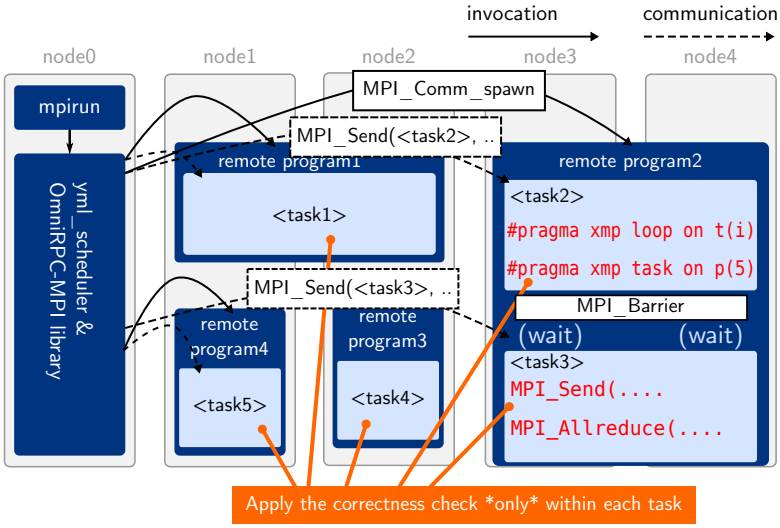


図 7　MUST による mSPDM プログラミングモデルの正当性チェックの適用範囲．赤文字で書かれた通信にのみ正当性チェックが行われ，黒文字の MPI 関数には適用されない

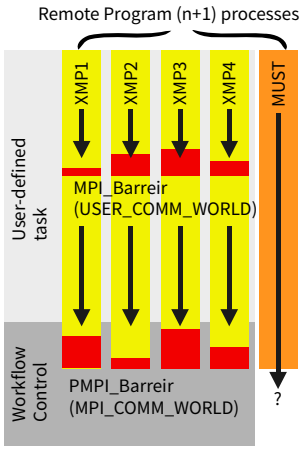


図 8　ワークフロー制御部の同期がとれない例

```
#define MYX

#ifdef MYX
#define MPI_Comm_rank(comm, rank)\
        PMPI_Comm_rank(comm, rank)

#define MPI_Bcast(buf, count, type, root, comm)\
        PMPI_Bcast(buf, count, type, root, comm)

#define MPI_Barrier(comm)\
        PMPI_Barrier(comm)

#endif // #ifdef MYX
```

図 9　MUST による正当性チェックを無効にするためのマクロ

として定義され，リンク時もしくは実行時にユーザ定義の同名の関数があった場合には，MPI ライブラリの関数ではなくユーザ定義の関数が優先して使用される．ユーザやツール開発者は，PMPI 関数の前後になんらかの処理を行う MPI 関数を自ら定義し，従来の MPI 関数と置き換えることで，通信プロファイリングや解析を行うことができる．

MUST が MPI 関数呼出をインターセプトするために用いている PnMPI も，MPI 関数の再定義による置き換えを用いている．よって，特定の MPI 関数呼出を MUST の解析の対象外に置くためには，MPI 関数ではなく PMPI 関数を用いればよい．図 7 に示したように，mSPMD のリモートプログラムは，

- ユーザ定義のタスク
- OmniRPC-MPI ミドルウエアによる制御部

からなる．本稿では，ミドルウエア OmniRPC-MPI について，**図 9** に示すようなマクロを定義し，マクロの使用/不使用を切り換えることで，必要な関数を置き換える．

ただし，PMPI 関数が用いられる制御部において MPI_COMM_WORLD に対する集合通信を用いている場合は，MUST プロセスではこれが呼び出されないため，集合通信が終了せず，プログラムがそれ以降に進むことができなくなってしまう．例えば，図 7 で示したように，リモートプログラムではタスクの終了を確実にするためにバリア同期を行っている．**図 8** に示すように，ユーザ定義のタスク内部に存在するバリア同期・MPI_Barrier(MPI_COMM_WORLD) は，MUST により自動的に MPI_Barrier(USER_COMM_WORLD) に置き換わり n プロセスで実行される．しかし，MUST 解析の対象外の制御関数である PMPI_Barrier(MPI_COMM_WORLD) は，PMPI_Barrier(MPI_COMM_WORLD) のままであり，本来 $n+1$ プロセスから呼び出さなければならないが，実際には MUST プロセス以外の n プロセスのみがこれを実行するため，同期が完了しない．そこで，本拡張では，リモートプログラム内部で使用される OmniRPC-MPI のコミュニケータを新たに定義し，ユーザが MUST による解析を行うときは USER_COMM_WORLD を複製し，そうでないときは MPI_COMM_WORLD を複製することで，この問題を回避した．あわせて，リモートプログラムのジェネレータも拡張され，USER_COMM_WORLD を得るための関数を含む MUST や PnMPI のライブラリなどがリモートプログラムにリンクされる．

SPMD タスクを含むリモートプログラムは，ワークフロースケジューラにリンクされた OmniRPC-MPI ミドルウエアによって起動される．リモートプログラムの起動には MPI_Comm_spawn 関数が用いられ，`mustrun` コマンドは使用できない．よって別の方法で，2.2 で述べた `mustrun` により行われる (1) アプリケーションに MUST を適用するためのコード生成とそのコンパイル，(2) 環境変数の設定，および (3) $n+1$ プロセスでのリモートプログラムの起動，を行う必要がある．

(1) について，MUST はアプリケーションを実行する環

境とコンパイルする環境が異なる場合を想定して，準備のみを行うオプション --must:mode prepare を用意している:

```
mustrun --must:mode prepare
    -np 4 application.exe
```

(2) については，現在の実装ではユーザがマニュアルで指定する．

(3) については，前述の MPI 関数の PMPI 関数へのマクロ定義を用いた置き換えにおいて，以下のように置き換える：

```
#define MPI_Comm_spawn(command, argv, maxprocs,\
 info, root, comm, intercomm, array_of_errcodes)\
       PMPI_Comm_spawn(command, argv, maxprocs+1,\
 info, root, comm, intercomm, array_of_errcodes)
```

オリジナルの MUST は単一の MPI アプリケーションを解析するためのツールであり，実行後に解析結果 MUST_Output.html を出力する．しかし，mSPMD プログラミングモデルでは，同時に複数のリモートプログラムが実行される．複数のプログラムに対して，同じフォルダから MUST による解析を実行した場合，最後に解析を終了したプログラムの結果のみが保存され，ほかは上書きされてしまう．そこで，本稿では，MUST_Output.html を MUST_Output_プロセス ID.html に変更した．yml はタスク名やプロセス ID，プロセス数などを

```
=======================
Task 1: Finished

Task 1: SUCCESS
component    : test1 (pid=17306)
start   time: Fri Jan 25 16:47:23 2019
              (1548402443.012190)
.....
=======================
```

のように記録しており，MUST の出力とタスクとの対応を確認することが可能である．

よりユーザフレンドリなインターフェスについては今後の課題として検討する．

4. 実験

予備実験として，mSPMD プログラミングモデルで簡単な通信を繰り返すタスクを実行し，MUST を適用するときとしないときの動作や実行時間などを調査した．

本実験では，以下のマルチコアノード 1 ノードを用いた：

```
Intel Xeon CUP E5-2680 v3 @ 2.5GHz (24 core)
DDR4-2133 Reg ECC (2GBx6)
```

タスクは，各コアで 1 プロセスを実行するフラット MPI モデルとした．

```
<?xml version="1.0"?>
<component type="impl" name="allreduce" abstract="allreduce">
<impl lang="XMP" nodes="CPU:(4)">
<header>
<![CDATA[
#include<unistd.h> //sleep
]]>
</header>
<source>
<![CDATA[
  int    myrank, nprocs;
  int    i, n;
  long   buf[1], rbuf[1];

  MPI_Comm_rank(MPI_COMM_WORLD, &myrank);
  MPI_Comm_size(MPI_COMM_WORLD, &nprocs);

  for(i=0; i<100; i++){
    MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_SUM, MPI_COMM_WORLD);
    usleep(100000);
  }

]]>
</source>
<footer />
</impl>
</component>
```

```
  for(i=0; i<100; i++){
    if(myrank==0)
    MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_MAX, MPI_COMM_WORLD);
    else
    MPI_Allreduce(buf, rbuf, 1, MPI_LONG, MPI_MIN, MPI_COMM_WORLD);
    usleep(100000);
  }
```

```
  for(i=0; i<100; i++){
    if(myrank%2==0)
    MPI_Send(buf, 1, MPI_LONG, dest, tag, MPI_COMM_WORLD);
    else
    MPI_Recv(buf, 1, MPI_LONG, src , tag, MPI_COMM_WORLD, &stat);
    usleep(100000);
    if(myrank%2==0)
    MPI_Recv(buf, 1, MPI_LONG, src , tag, MPI_COMM_WORLD, &stat);
    else
    MPI_Send(buf, 1, MPI_LONG, dest, tag, MPI_COMM_WORLD);
    usleep(100000);
  }
```

```
  for(i=0; i<100; i++){
    if(myrank%2==0)
    MPI_Send(buf, 1, MPI_LONG, dest, tag, MPI_COMM_WORLD);
    else
    MPI_Recv(buf, 1, MPI_FLOAT, src , tag, MPI_COMM_WORLD, &stat);
    usleep(100000);
    if(myrank%2==0)
    MPI_Recv(buf, 1, MPI_LONG, src , tag, MPI_COMM_WORLD, &stat);
    else
    MPI_Send(buf, 1, MPI_LONG, dest, tag, MPI_COMM_WORLD);
    usleep(100000);
  }
```

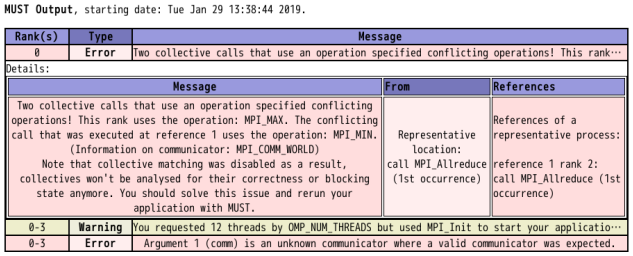
図 10 実験に用いられたタスク．上から，allreduce (エラーなし)，allreduce (エラーあり)，pingpong (エラーなし), pingpong (エラーあり)．なお，allreduce (エラーなし) 以外はカーネル部のみ示した．

4.1 MPI タスク

allreduce および pingpong を繰り返す 2 種類のタスクが考慮された．それぞれの場合で，誤りを含むコードと含まないコードを用意した．図 10 に，実験に用いられたタスクを示す．上から，allreduce (エラーなし)，allreduce (エラーあり)，pingpong (エラーなし) , pingpong (エラーあり) である．なお，allreduce (エラーなし) 以外はカーネル部のみ示した．allreduce (エラーあり) では，プロセス 0 が他と異なる MPI_Op を実行している．pingpong (エラー

表 1　各タスクに対する実行可能性とエラーの報告状況

	MUST あり	MUST なし
allreduce エラーあり	途中終了・エラーのレポート	途中終了
allreduce エラーなし	完了・エラーのレポート	完了
pingpong エラーあり	完了・エラーのレポート	完了
pingpong エラーなし	完了・エラーのレポート	完了


MUST Output, starting date: Tue Jan 29 13:38:44 2019.

Rank(s)	Type	Message
0	Error	Two collective calls that use an operation specified conflicting operations! This rank···

Details:

Message	From	References
Two collective calls that use an operation specified conflicting operations! This rank uses the operation: MPI_MAX. The conflicting call that was executed at reference 1 uses the operation: MPI_MIN. (Information on communicator: MPI_COMM_WORLD) Note that collective matching was disabled as a result, collectives won't be analysed for their correctness or blocking state anymore. You should solve this issue and rerun your application with MUST.	Representative location: call MPI_Allreduce (1st occurrence)	References of a representative process: reference 1 rank 2: call MPI_Allreduce (1st occurrence)

0-3	Warning	You requested 12 threads by OMP_NUM_THREADS but used MPI_Init to start your applicatio···
0-3	Error	Argument 1 (comm) is an unknown communicator where a valid communicator was expected.

図 11　Allreduce の MPI_Op 不一致に対する MUST の出力のスクリーンショット

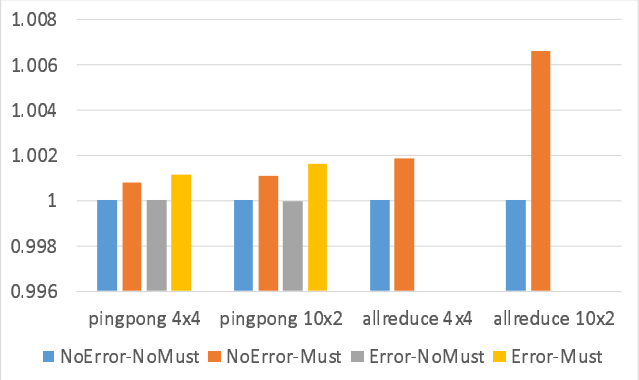


図 12　実行時間

あり) では，受信送信のデータ型が異なっている．

表 1 に，図 10 の各タスクを行う mSPMD プログラミング開発実行環境で，MUST を適用あり・なしでの動作とエラーの報告状況を示す．MUST を適用した場合，異常終了する場合も含めてすべての場合でチェックの結果のレポートが出力される．エラーがない場合でもエラーレポートが出ているのは，現在の実装では，XMP のランタイムライブラリを MUST の解析対象外としておらず，これに含まれるクリティカルでないバグが報告されているからである．

図 11 に，MPI_Allreduce に対する MPI_Op の不一致があった場合の MUST の出力を示す．MUST を用いない場合，このタスクは異常終了するだけであるが，MUST を用いる場合にはエラー内容を出力したのちに異常終了する．以上のように，本稿の拡張によって，マスター・プログラムから呼び出された，mSPMD ワーカ・プログラムに対しても MUST のチェックを適用可能になった．

図 12 に，pingpong および allreduce を行うタスクを，4 プロセス × 4 タスク，および 10 プロセス × 2 タスクを実

```xml
<?xml version="1.0"?>
<component type="impl" name="allreduce" abstract="allreduce">
<impl lang="XMP" nodes="CPU:(4)">
<source>
<![CDATA[
   int sum;
   sum = 1;

#pragma xmp task on _XMP_default_nodes(1)
   {
#pragma xmp reduction (+:sum) on _XMP_default_nodes(3)
   }
]]>
</source>
<footer />
</impl>
</component>
```

図 13　検証に用いられたエラーのある XMP タスク

行した場合の相対実行時間を示す．基準値はエラーのないタスクに対して MUST を適用しなかった場合である．なお，ワークフロー・スケジューラも 1 プロセスを使用するため，実際に使用されるプロセス数は，MUST を適用しない場合は 1 + (プロセス数) × (タスク数) であり，MUST を適用する場合は 1 + (プロセス数 + 1) × (タスク数) である．すべての場合で，MUST を適用すると実行時間が増加し，またタスク毎のプロセス数が大きいほうが増分が大きい．しかし，増加はごく僅かである．poigpong のケースを見ると，MUST を適用しない場合，エラーの有無に拘わらず，実行時間がほぼ同様なのに対して，MUST を適用する場合には，エラーのあるタスクのほうが実行時間が大きい．

4.2　XMP タスク

続いて，タスク内の XMP におけるエラー検出について動作を確認した．なお，現在の MUST の XMP 向けの拡張は，一部の XMP エラーの検出のみしか実装されていないため，本節では実行時間などには焦点を当てず，動作確認にとどめた．

図 13 にエラーのある XMP プログラムの例を示す．`_XMP_default_nodes ()` は，YML にタスクの使用ノード数を知らせる `nodes="CPU:(4)"` から自動生成されたデフォルトのノード指示文である．このプログラムでは，`#pragma xmp task on _XMP_default_nodes(1)` 定義されたブロックはノード 1 のみで実行されるが，その内部では`#pragma xmp reduction (+:sum) _XMP_default_nodes(3)` としてノード 3 のみで実行されるリダクションを呼び出しているため矛盾が生じる．

図 14 に，エラーのある XMP タスクを mSPMD プログラム+MUST を用いて実行したときのスクーンショットを示す．現在の MUST の実装では XMP に対するエラーは MUST_Output.html ではなく，標準出力／標準エラー出力に表示される．スクリーンショットは，エラーを表示する部分をマウスクリックして強調した瞬間を撮影した：

```
[CurrentNodesTrack 0 ] is no subset: \
```

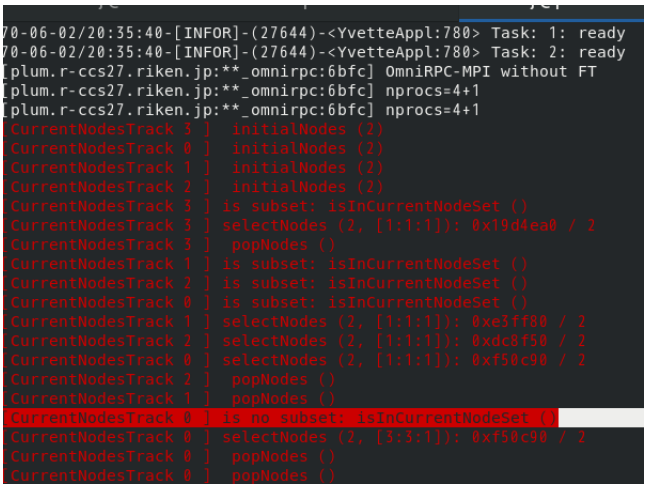


図 14 mSPMD プログラムでエラーのある XMP タスクを実行した場合のスクリーンショット

`isInCurrentNodeSet ()`

これは，XMP 指示文が実行時のコンテクストのノード集合の外であることを示す．

5. 終わりに

本稿では複数の SPMD プログラムをワークフローのタスクとして協調動作させてアプリケーションを実行するマルチ SPMD プログラミングモデル XMP/YML において，タスク部に MPI や XMP などの並列プログラムの正当性チェックを行うライブラリである MUST を適用した．このために，XMP/YML およびミドルウエア OmniRPC-MPI を拡張した．ミドルウエアにより行われる制御のための通信については MUST によるチェックを行わないものとし，そのための手法を示した．本稿の拡張によって，SPMD プログラミングモデルで実行されるワーカ・プログラム内部で実行されるユーザ記述のタスク部について MUST 解析を適用しすることが可能になった．実験を行い，タスクに MPI もしくは XMP の記述エラーがあった場合に，MUST によって検出可能であることを示した．

今後の課題とては，

- マルチノード環境での性能評価
- ユーザインターフェースやエラーの報告手法の改善

などがあげられる．

参考文献

[1] O. Delannoy, N. Emad, and S. Petiton. Workflow global computing with yml. In *The 7th IEEE/ACM International Conference on Grid Computing*, pp. 25–32, 2006.

[2] O. Delannoy and S. Petiton. A peer to peer computing framework: Design and performance evaluation of yml. In *Third International Workshop on Algorithms, Models and Tools for Parallel Computing on Heterogeneous Networks*, pp. 362–369, 2004.

[3] T. Hilbrich, F. Hasel, M. Schulz, B. R. de Supinski, M. S. Muller, and W. E. Nagel. Runtime MPI collective checking with tree-based overlay networks. In *Proceedings of the 20th European MPI Users' Group Meeting (EuroMPI 13)*, pp. 129–134. ACM, 2013.

[4] T. Hilbrich, J. Protze, M. Schulz, B. R. de Supinski, and M. S. Muller. MPI runtime error detection with MUST: Advances in deadlock detection. In *International Conference on High Performance Computing, Networking, Storage and Analysis (SC12)*. IEEE, 2012.

[5] J. Lee and M. Sato. Implementation and performance evaluation of XcalableMP: A parallel programming language for distributed memory systems. In *39th Annual International Conference on Parallel Processing*, pp. 413–420, 2010.

[6] J. Protze, C. Terboven, M. S. Muller, S. Petiton, N. Emad, H. Murai, and T. Boku. Runtime correctness checking for emerging programming paradigms. Proceedings of the First International Workshop on Software Correctness for HPC Applications (Correctness' 17). IEEE, 2017.

[7] M. Sato, M. Hirano, Y. Tanaka, and S. Sekiguchi. Omnirpc: A grid rpc facility for cluster and global computing in openmp. In *International Workshop on OpenMP Applications and Tools, 2001*, pp. 130–136, 2001.

[8] The MUST Project. https://www.itc.rwth-aachen.de/must.

[9] M. Tsuji, M. Sato, M. Hugues, and S. Petiton. Multiple-SPMD programming environment based on PGAS and workflow toward post-petascale computing. In *Proceedings of the 2013 International Conference on Parallel Processing (ICPP-2013)*, pp. 480–485. IEEE, 2013.

[10] 辻美和子, G. Jérôme, S. Petiton, 佐藤三久. マルチ SPMD プログラミング環境における線形ソルバの性能評価. 情報処理学会研究報告, No. 2018-HPC-165, pp. –. 情報処理学会, 2018.

[11] 辻美和子, 佐藤三久. マルチ SPMD 環境における耐故障性実現に向けた OmniRPC-MPI の拡張. 情報処理学会研究報告, No. 2014-HPC-146, pp. –. 情報処理学会, 2014.

[12] 辻美和子, 佐藤三久, M. Hugues, S. Petiton. 並列コンポーネントを統合する階層型並列プログラミングモデル. 情報処理学会研究報告, No. 2012-HPC-135, pp. 1–10. 情報処理学会, 2012.