Vol.2022-HPC-183 No.19
2022/3/18

分散メモリ並列計算機用スクリプト言語 SXMP の実装と評価

新井 正樹[1,a)]　津金 佳祐[1]　前田 宗則[1]　吉川 隆英[1]

概要：本報告では，Partitioned Global Address Space（PGAS）言語 XcalableMP （XMP）と同等の機能をもち，かつプログラムを Python 言語ベースで記述可能な Scripting XMP（SXMP）を提案する．SXMP は動的型付け言語とインタプリタを利用したプログラム開発の容易性と，コンパイラを利用した高性能コード生成というふたつの特徴をもつ．また，XMP の機能に加えて，分散並列化関連の指示を実行時に指定可能とする動的機能と，データ依存解析に基づくソースコードの自動変換機能を導入している．予備評価として，himeno ベンチマークプログラムを SXMP と XcalableMP/C で記述した場合の性能を比較し，SXMP のプログラムを C 言語に変換して実行するコンパイラモードでは，SXMP が XcalableMP/C に対して 10% 程度の性能低下で実行できることを確認した．

1. はじめに

近年，分散メモリ並列計算機のノード数は増大し，各計算ノードの構成が複雑化しているため，その上で高い利用効率を実現するプログラムの開発が困難となっている．大規模並列計算機を利用する場合には，様々なレベルでの並列性抽出とデータ配置の工夫が必要であるため，ハードウェアを意識したプログラムが記述可能であるプログラミング言語 C/C++/Fortran を使用し，計算ノード間の通信には Message Passing Interface（MPI）ライブラリを利用することが主流となっている．一方で，機械学習・深層学習の分野では，高速実行が必要な主要部分はプログラミング言語 C/C++ で記述し，それらを Python 言語のインタプリタから利用することで，プログラム開発の負担を軽減することを意図したツール [1][20] が開発されている．HPC の分野でも Python 言語のインタプリタから MPI を利用するためのモジュールとして mpi4py[9] が開発されている．抽象度の高いレベルのデータ構造の利用とインタプリタを利用した対話的開発方法によって，mpi4py を利用した場合は C/C++/Fortran 言語で MPI ライブラリを利用する場合と比較して，プログラムのサイズを小さくできることとプログラム開発が容易であるという利点がある．しかし，mpi4py を利用する場合には，計算ノード間でのデータの分散指定と通信を明示的に記述しなけらばならないため，依然としてプログラム開発者の負担は大きい．

分散メモリ並列計算機上のプログラム開発を容易にするためのプログラミング言語として，指示文ベースの Partitioned Global Address Space（PGAS）言語である XcalableMP（XMP）[26][15] が提案されている．XMP は逐次実行が可能である C/Fortran 言語のプログラムに指示文を付加することでプログラムの分散並列化を可能とし，グローバルビューによる記述の容易性とプログラムの段階的分散並列化の手段をプログラム開発者に提供している．XMP は，指示文を利用することで逐次プログラムと分散並列化プログラムのふたつを同時に開発できるという利点をもつ一方で，指示文はコンパイル時に解釈する必要があるため，一部の拡張を除いてプログラムの実行時に分散並列化の適用方法を決定することができないという問題がある．さらに，XMP では，プログラムのもつ並列性の抽出と抽出した並列性をデータと共に計算ノードに分配する方法を決定することはプログラム開発者の責任であり，これらの作業を軽減する仕組みや指示文の誤りを検出する手段は提供していないため，プログラムを分散並列化するための開発コストは依然として小さくない．

これらの問題を解決するための手段として，我々は Python 言語をベースとした分散メモリ並列計算機用スクリプト言語 Scripting XMP（SXMP）を開発している．SXMP は次の特徴をもつ．

- XMP のグローバルビューに関連する機能は基本的に継承する．
- 分散並列化に関する指定は指示文ではなく，関数とし

1 富士通株式会社
FUJITSU LIMITED
a) arai.masaki@fujitsu.com

ⓒ 2022 Information Processing Society of Japan

て実現する．

- 配列に関する処理は NumPy[17] ベースとする．
- Python 言語のインタプリタ上で実行するモードと，C 言語のプログラムに変換してコンパイル・実行するモードをもつ．
- ソースコードレベルでのデータの依存解析，自動並列化，ループの最適化機能をもつ．

本稿の構成を以下に示す．第 2 章では SXMP の言語の概要について述べる．第 3 章では SXMP の実装について述べ，第 4 章では SXMP の性能評価について記述する．第 5 章では今後の課題を，第 6 章では関連研究について記述し，第 7 章ではまとめを行う．

2. SXMP の概要

SXMP は，XMP と同等の機能を実現する言語 SXMP-1 と，各種の自動化機能によって，SXMP-1 のプログラム開発を支援する言語 SXMP-0 のふたつのレベルをもつ分散並列計算機用のスクリプト言語である．

2.1 SXMP-1

SXMP-1 は，基本的に XMP のグローバルビューに関する機能を継承している．XcalableMP/C（XMPC）のプログラムの例を図 1 に示す．XMPC では，分散並列実行に関連する指示文は

```
#pragma xmp
```

で始まる pragma として記述する．XMPC コンパイラは，コンパイル時にこれらを解釈して，MPI を利用して分散並列実行を行う Single Program Multiple Data（SPMD）形式の実行可能プログラムを生成する．また，コンパイル時に pragma による分散並列指示を無視することで，逐次実行形式の実行可能ファイルを作成することができる．このようにして，XMP では，ひとつのプログラムで逐次実行形式のプログラムと分散並列実行形式のプログラムの両方を同時に保持することにより，段階的に分散並列実行形式のプログラムの開発を行うことを可能にしている．

図 1 の XMPC プログラムに相当する，SXMP-1 プログラムを図 2 に示す．SXMP-1 は Python 言語をベースとしたプログラミング言語であり，配列の処理については NumPy と同等の記述を採用している．XMP とは異なり，分散並列実行に関する指示は，Python 言語の通常の関数として実現している．また，XMP でループの並列実行を指示する `loop` 指示文（図 1 の 12 行目）と，タスクの実行を指示する `task` 指示文（図 1 の 18 行目）は，SXMP-1 ではそれぞれ `parallel_on` 文（図 2 の 12 行目）と，`only_on` 文（図 2 の 18 行目）によって実現している．Python 言語の if 文や for 文などと同様に，`parallel_on` と `only_on` による指示が影響する範囲はインデントによって表現する．

```
1  #pragma xmp nodes p[4]
2  #pragma xmp template t[20]
3  #pragma xmp distribute t[block] onto p
4
5  double a[20];
6
7  #pragma xmp align a[i] with t[i]
8
9  int main() {
10   double tmp[20];
11
12 #pragma xmp loop on t[i]
13   for (int i = 0; i < 20; i++)
14     a[i] = func(i);
15
16 #pragma xmp gmove
17   tmp[0:20] = a[0:20];
18 #pragma xmp task on p[0]
19   printf("%f\n", tmp[0]);
20 }
```

図 1 XcalableMP/C のプログラム

```
1  p = nodes((4))
2  t = template((20))
3  distribute(t, ('block'), p)
4
5  a = make_global_array((20), 'float64')
6
7  align_with(a[i], t[i])
8
9  def main():
10     tmp = make_array((20), 'float64')
11
12     parallel_on((i), t[i]):
13         for i in range(0, 20):
14             a[i] = func(i)
15
16     gmove(tmp[0:20], a[0:20])
17
18     only_on(p[0]):
19         print(tmp[0])
```

図 2 SXMP-1 のプログラム

XMPC では，pragma を指示文とする性質上，一部の拡張を除いて，指示文に関連するパラメータは基本的にコンパイル時に定数値でなければならない．しかし，SXMP-1 では分散並列実行に関する指示は通常の関数であるため，それらに関するパラメータは実行時に動的に変更することが可能である．

SXMP-1 のプログラムは，XMP と同様に逐次実行形式のプログラムとして実行することができる．SXMP-1 の逐

次実行モードでは，分散並列実行に関連する関数の動作をすべて実行時に何もせず None を返す関数として実行する．また，逐次実行モードでは，parallel_on 文と only_on 文を実行時に Python 言語の

```
if True:
```

に置き換えることによって，逐次実行プログラムとしての動作を実現している．

2.2 SXMP-0

SXMP-0 は，プログラム開発者に各種の自動化機能を提供することで，SXMP-1 のプログラム開発を支援することを目的とする言語である．SXMP-0 は，SXMP-1 から計算ノードを定義する nodes 関数以外の分散並列指示の関数を取り除き，各種自動化を適用するための対象範囲を指定する auto_begin と auto_end を追加した言語である．SXMP-0 のプログラムの例を図 3 に，また図 3 の解析対象範囲に対して，ソースコード変換を行った結果のプログラムの例を図 4 に示す．図 4 では，解析対象範囲のコードに対して，自動並列化とループタイル化を適用している．

```
1  p = nodes((N))
2  auto_begin()
3  for i in range(0, N):
4      for j in range(0, N):
5          for k in range(0, N):
6              C[i,j] = C[i,j] + A[i,k] * B[k,j]
7  auto_end()
```

図 3 SXMP-0 のプログラム

```
1  p = nodes((N))
2  auto_begin()
3  if (N >= 0):
4   parallel_on((c5), p[0:floor(N,32)]):
5    for c5 in range(0, floor(N,32)):
6     for c4 in range(0, floor(N,32)):
7      for c3 in range(32*c5, min(N,32*c5+31)):
8       for c2 in range(32*c4, min(N,32*c4+31)):
9        for c1 in range(0, N):
10        C[c3,c2] = (C[c3,c2]
11                  + (A[c3,c1] * B[c1,c2]))
12 auto_end()
```

図 4 SXMP-0 のプログラム（自動変換結果）

SXMP-0 では，凸多面体モデルを利用した最適化 [2][6][12][11] により，対象範囲のコードのデータ依存解析を行い，対象範囲のコードに対して各種の source-to-source 変換を適用することで，SXMP-1 のプログラム開発や性能チューニング作業を容易にすることを目的としている．凸多面体モデルを利用した最適化は，自動並列化，ループの分割・融合，ループのタイル化，ベクトル化，SIMD 化などを統合して適用することができる手法であり，近年，OSS のコンパイラ [22][13] でも採用されている．最適化の対象とすることができるコードは，Static Control Part（SCoP）形式 [12][10] に限定されるが，典型的な HPC 系のアプリケーションのカーネル部分の多くは SCoP 形式に分類できる．

SXMP-1 の parallel_on 文は XMP の loop 指示文と同様に完全入れ子形式のループのみを分散並列化指示の対象とすることができる．したがって，対象範囲のループに多次元の並列性が存在する場合であっても，完全入れ子ループとして記述しなければ，その処理を計算ノードに分散させることができない．また，多次元の並列化可能ループが完全入れ子形式であっても，計算ノードに分散させるループ集合の選択に合わせて，ループ交換によってループの入れ子の順序を調整する必要がある．さらに，データの分散方法やループの回転数が計算ノードの構成の次元数や各次元のノード数に合わない場合には，分散並列化の指示を適用する前に，対象ループに対してストリップマイニング [28] やループタイル化の最適化を適用して，ループの回転数を調整しなければならない場合がある．凸多面体モデルを利用した最適化はデータ依存解析結果に基づいて，これらの条件に合う形状へループを自動変換する機能を実現するため，ユーザによるプログラムの書き換えの手間を軽減し，性能チューニングのために複数の分散並列化方法を試す作業を補助することができる．

2.3 SXMP プログラムの実行

SXMP-0 と SXMP-1 のプログラムはどちらも SXMP のインタプリタモードかコンパイラモードで実行することができる．インタプリタモードでは，Python 言語のインタプリタから MPI ライブラリ機能を利用するためのモジュールである mpi4py のためのコマンド mpiexec 上で，利用する計算ノード数を指定して動作させることができる．コンパイラモードでは，SXMP-0 と SXMP-1 のプログラムを C 言語のプログラムに変換して，MPI ライブラリを利用するための C コンパイラ mpicc によってコンパイルし，MPI ライブラリに付属のコマンドである mpirun 上で実行することができる．したがって，SXMP のユーザは，小規模の計算ノード数とデータ数を指定し，まずはインタプリタ上で対話的にプログラムを開発し，実行性能が重要であるか，あるいは大量の計算ノードを利用する場合には，C 言語に変換して効率良い実行を行う形態の開発手法を採ることができる．

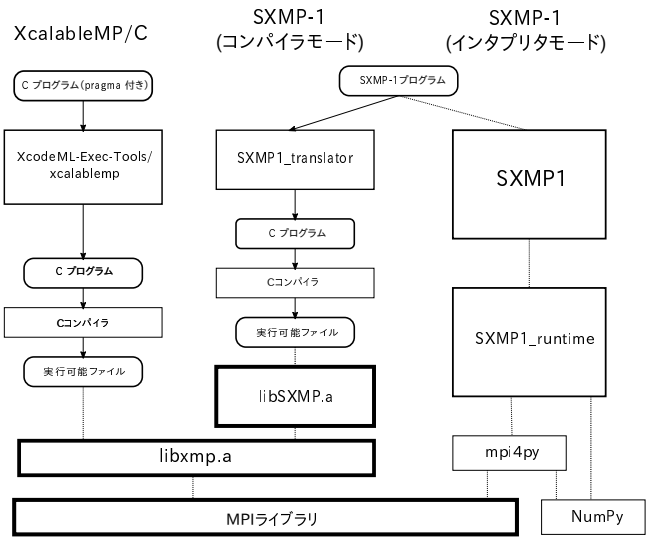

図 5 実行環境の概観

3. 実装

SXMP は，XMP のリファレンス実装である Omni Compiler[19] を利用して実装している．図 5 に，Omni Compiler による XMPC と，SXMP のコードを C 言語に変換して実行するコンパイラモードと，SXMP のコードを Python 言語のインタプリタ上で実行するインタプリタモードの実装の概観を示す．

図 5 の左側の列が Omni Compiler による XMPC プログラムの処理の概要である．Omni Compiler は XMPC プログラムを次の手順で処理する．

- pragma 付きの C プログラムを `libxmp.a` のライブラリ関数を呼び出す C プログラムに source-to-source 変換する．XMPC プログラム内の pragma による指示文の解釈は `XcodeML-Exec-Tools` の `xcalablemp` 部分が実施する．
- 変換結果の C プログラムは C コンパイラ（`mpicc`）によって，分散メモリ並列計算機上で動作する実行可能ファイルにコンパイルされる．MPI ライブラリの機能は実行時に `libxmp.a` を通して利用される．

上記手順で生成した実行可能ファイルは，MPI ライブラリに付属のコマンドである `mpirun` 上で実行することができる．

3.1 SXMP-1 のインタプリタモード

図 5 の右側の列が SXMP-1 プログラムをインタプリタモードで処理する場合の概要である．SXMP-1 プログラムはインタプリタモードで次のように実行される．

- SXMP-1 プログラムは，mpi4py のためのコマンド `mpiexec` 上で動作させるプログラム `SXMP1_run` に SXMP-1 プログラムを含むファイルを入力ファイルとして指定することで Python 言語のインタプリタ上で実行することができる．
- `SXMP1_run` 内で利用する `SXMP1` モジュールによって，実行時に分散並列化に関係する関数を `SXMP1_runtime` モジュールの関数呼び出しに変換する．この処理は Omni Compiler の `XcodeML-Exec-Tools` の `xcalablemp` 部分の処理に相当する．
- 変換した関数呼び出しを `SXMP1_runtime` モジュールを利用して解釈実行する．この処理は Omni Compiler の `libxmp.a` の関数呼び出しの処理に相当する．MPI ライブラリの機能は `mpi4py` モジュールを通して利用される．

インタプリタモードでは配列の処理には NumPy モジュールを利用する．

3.2 SXMP-1 のコンパイラモード

図 5 の中央の列が SXMP-1 のコンパイラモードによる処理の概要である．SXMP-1 のコンパイラモードはプログラムを次のように処理する．

- `SXMP1_translator` コマンドによって，SXMP-1 プログラムを C 言語のプログラムに source-to-source 変換する．このとき，分散並列化に関係する関数は実行時に `libSXMP.a` のライブラリ関数を呼び出す処理に変換する．ライブラリ `libSXMP.a` の行う処理は Omni Compiler の `XcodeML-Exec-Tools` の `xcalablemp` 部分の処理に相当する．
- 変換結果の C 言語のプログラムは C コンパイラ（`mpicc`）によって分散メモリ並列計算機上で動作する実行可能ファイルにコンパイルされる．SXMP-1 の分散並列化に関係する関数は，実行時に `libSXMP.a` を通して Omni Compiler の `libxmp.a` への関数呼び出しとして実行される．MPI ライブラリの機能は実行時に `libxmp.a` を通して利用される．

上記手順で生成した実行可能ファイルは，MPI ライブラリに付属のコマンドである `mpirun` 上で実行することができる．

コマンド `SXMP1_translator` は，SXMP-1 のプログラムを C プログラムに変換する時に分散並列化に関係する関数の引数が定数値であることを確定できる場合は，それらの関数を `libSXMP.a` ではなく，`libxmp.a` を直接呼び出す関数に変換する．その場合には，結果として Omni Compiler の変換結果と同等の C プログラムに変換することが可能となる．

3.3 SXMP-0

現在の SXMP-0 の実装では，凸多面体モデルを利用した最適化のアルゴリズムとして，Pluto+[5] 相当のデータの依存解析と source-to-source 変換機能を実現している．また，Pluto+ のアルゴリズムが生成した多次元スケジュー

ルから，最適化結果のソースコードを生成するコード生成処理についてはISL[25] の機能を利用している．Pluto+ のアルゴリズムは対象範囲のコードに対して，抽出可能な並列化可能ループを可能な限り外側ループとして配置する多次元スケジュールを作成する．コード生成で利用する凸多面体スキャンアルゴリズム [21] は，凸多面体モデルを利用した最適化の結果である多次元スケジュールから，実行時にループ内に条件分岐を配置しないことを目的としたコードを生成するため，一般に結果のソースコードでは並列化ループは完全入れ子形式のループにはならない．この問題を解決するために，SXMP-0 の実装では，最適化結果の多次元スケジュール上で並列化可能の次元が連続する部分については，その部分は完全入れ子ループとしてコードを作成し，それ以外の部分には凸多面体スキャンアルゴリズムを適用することによって，分散並列化の指示を直接付加することが可能であるループを生成している．また，生成した多次元スケジュール上で並列化可能の次元が連続する場合には，これらの次元は任意の入れ子の順序で実行することができるという性質がある．この性質を利用して，SXMP-0 では，自動変換を適用する場合のオプション指定によって，並列化可能次元の実行順序を選択できるようにしている．これらの機能を利用することで，SXMP のプログラム開発者は，並列化するループ集合の選択と選択したループ集合を完全入れ子形式のループとして望みの順序で配置することが可能となっている．

3.4 C 言語プログラムへの変換

この節では，Python 構文で記述した SXMP プログラムを C 言語のプログラムに変換する機能の実装について述べる．この処理で主に問題となるのは，型情報の抽出，配列の次元数の情報の抽出である．

Python 言語では C 言語と異なり，プログラム中に変数や関数の型を記述する必要性はなく，変数や関数の型がプログラムの実行時に一意に決定する必要性もない．Python 言語にはプログラム中の変数や関数に任意で型情報を指定する型ヒント機能がある．SXMP ではこの型ヒント機能を採用しているので，それによって変数や関数の型を一意に指定した場合には，その型情報を C 言語へのプログラム変換に利用する．型ヒント機能では変数や関数が実行時に複数の型を取ることを記述可能であり，また Python 言語と同様に SXMP では型ヒント機能による型の記述は強制ではないため，変数や関数の型を一意に決定できない場合の対処が必要となる．さらに，SXMP では，配列の記述方法や実装は NumPy の ndarray をベースとしているため，配列データを表す変数の型は ndarray であるが，現在の Python 言語の型ヒント機能では配列の要素のデータ型や配列の形状を記述する方法が存在しないため，C 言語のプログラムに変換する場合は問題となる．

これらの問題を解決するために，SXMP では，C 言語に変換する場合にはプログラムの全体解析を行い，変数や関数の型の情報と，配列の要素のデータ型と配列の形状を決定する．プログラムの全体解析を実施しても，プログラム中にこれらの情報が不在である場合や，情報が存在する場合でも一意に決定できない場合があり得る．このような場合には，現在の SXMP の実装ではすべての可能な状況に対応する複数のコードを生成し，実行時に適切なコードを選択して実行するマルチバージョニングによって対応している．任意の Python 言語のプログラムを，この方法で C 言語のプログラムに変換することは一般には困難であるが，SXMP は主として HPC 系のアプリケーションを記述するためのデータ構造と構文に限定した Python 言語のサブセットであるため，現時点では実質的には問題とはなっていない．

例として，図 6 に示す SXMP のプログラムを，図 7 に示す C 言語のプログラムに変換する場合の処理について述べる．図 6 の関数 `matmul` の引数 `N`，`A`，`B`，`C` とループ変数 `i`，`j`，`k` の型は前述のプログラムの全体解析から決定する．また，引数 `A`，`B`，`C` の型が ndarray の配列であると解析した後で，それぞれの配列の要素の型も同様にプログラムの全体解析から決定する．次に，図 6 のプログラムが，関数 `matmul` の内部では配列 `A`，`B`，`C` をループ内で 2 次元配列として参照しているという情報を利用し，これらの配列が C 言語では double 型の 2 次元配列であることを定義するマクロ `VARRAY2` を生成している．この例に示した変換結果で使用する C 言語の型やマクロ定義を図 8 に示す．図 7 では NumPy の ndarray 型の配列データを，C 言語の可変長配列として処理するコードに変換している．

ここでは，図 6 に示していない外側のプログラムの情報から，配列 `A`，`B`，`C` の要素の型を double に決定できると仮定したが，プログラムの全体解析を行っても配列の要素の型を決定できない場合もあり得る．その場合には，関数 `matmul` の配列 `A`，`B`，`C` の要素の型が NumPy の ndarray でサポートしている型のどの型であっても実行できるように，関数 `matmul` のマルチバージョニングを行い，配列 `A`，`B`，`C` の要素の型の任意の組み合わせに対応したコードを生成し，実行時に配列 `A`，`B`，`C` の要素の型をチェックした後でそれらの型に合った適切な関数を呼び出すコードを生成する必要がある．

```
1 def matmul(N, A, B, C):
2     for i in range(0, N):
3         for j in range(0, N):
4             for k in range(0, N):
5                 C[i,j] = C[i,j]
6                         + A[i,k] * B[k,j]
```

図 6 SXMP プログラム（C 言語への変換の入力）

```
1  void matmul(int N, NDARRAY_P A,
2                     NDARRAY_P B,
3                     NDARRAY_P C) {
4    VARRAY2(double, B, B_);
5    VARRAY2(double, A, A_);
6    VARRAY2(double, C, C_);
7    for (int i = 0; i < N; i++) {
8      for (int j = 0; j < N; j++) {
9        for (int k = 0; k < N; k++) {
10         (*C_)[i][j] = ((*C_)[i][j]
11                     + ((*A_)[i][k]
12                     * (*B_)[k][j]));
13       }
14     }
15   }
16 }
```

図 7 C 言語への変換結果

```
1  struct NDARRAY {
2    int n_dim;
3    int dim0;  int dim1;  int dim2;  int dim3;
4    int dim4;  int dim5;  int dim6;
5    void *a;
6  };
7  typedef struct NDARRAY* NDARRAY_P;
8
9  #define VARRAY2(TYPENAME,NAME,NEWNAME) \
10     typedef TYPENAME (* const _MT_ ## NAME)\
11       [NAME->dim0][NAME->dim1];\
12     _MT_ ## NAME NEWNAME = NAME->a
```

図 8 C 言語への変換で利用する定義

4. 性能評価

himeno ベンチマークを利用して SXMP の実行性能に関する予備評価を行った．比較対象として，XMP のリファレンス実装である Omni Compiler を使用した．評価で使用した himeno ベンチマークプログラムは，Omni Compiler の sample に付属の XMPC のプログラムを利用した．himeno ベンチマークは，32 ビット浮動小数点（C 言語の float）を処理する C 言語のプログラムである．SXMP では，配列のデータ構造については NumPy を利用しているため，32 ビット浮動小数点（float32）のデータを扱うことが可能であるが，Python 言語の float 型は 64 ビットの浮動小数点である．このため，評価を公平にするために，himeno ベンチマークのデータ型を 64 ビットの浮動小数点（C 言語の double）に書き換えたプログラムを作成し，それを評価に使用した．

性能評価を行った環境の情報を表 1 に示す．また，逐次実行の性能評価結果を表 2 に，並列実行の性能評価結果を表 3 に，それぞれ示す．表 2 と表 3 で，SMALL と MIDDLE はぞれぞれ himeno ベンチマークのデータセットの種類を表す．また，表の評価結果の値は himeno ベンチマークが出力するスコアの値（単位は MFLOPS）である．

表 1 測定環境

CPU	AMD Ryzen 5 5600X（6 コア，12 スレッド）
Memony	64GB
OS	Debian GNU Linux 11.2
C	GNU C Compiler 10.2.1（-O2 を使用）
Python	Python 3.9.2 NumPy 1.19.5 mpi4py 3.0.3
MPI	OpenMPI 4.1.0
XcalableMP/C	Omni Compiler 1.3.4

表 2 逐次実行性能（単位は MFLOPS）

実装	データセット	
	SMALL	MIDDLE
C	2086.47	1848.94
SXMP-1（インタプリタモード）	7.98292	7.96759
SXMP-1（コンパイラモード）	2130.46	1814.76

表 3 並列実行性能（単位は MFLOPS）

実装	データセット	
	SMALL	MIDDLE
XcalableMP/C	8818.03	7076.71
SXMP-1（インタプリタモード）	12.1738	11.3577
SXMP-1（コンパイラモード）	8122.01	6386.19

表 2 の評価では，分散並列指示の記述のない，C 言語のプログラムと SXMP のプログラムの実行性能を比較している．この表が示すように，SXMP のプログラムをインタプリタモードで実行した場合は，C 言語のプログラムと比較して，それぞれ SMALL の時に 261 倍，MIDDLE の時に 233 倍遅くなっている．しかし，SXMP のコンパイルモードで SXMP のプログラムを C 言語のプログラムに変換した場合には，元の C 言語のプログラムとほぼ同等の性能になっている．

表 3 の評価では，himeno ベンチマークの分散並列化の指示を有効化し，4 コアを使用して実行した場合の実行性能を比較している．比較に使用した SXMP プログラムは，XMPC と同等の分散並列化を実現するように作成した．この表が示すように，SXMP のプログラムをインタプリタモードで実行した場合は，XMPC のプログラムと比較して，それぞれ SMALL の時に 724 倍，MIDDLE の時に 623 倍遅くなっている．逐次実行の場合以上に遅くなっている理由は，分散並列化したループ内での配列のアクセスが元の配列アクセスよりも複雑なコードになっているためである．SXMP のコンパイルモードで SXMP のプログラムを C 言語のプログラムに変換した場合には，XMPC の

プログラムと比較して約 10 % 程度の性能低下で済んでいる．この性能差の主な原因は，XMPC が配列の要素を 1 段階のポインタ参照でアクセスしている所を，SXMP は NDARRAY を経由したことにより 2 段階のポインタを利用してアクセスしているためである．

5. 今後の課題

5.1 言語仕様

現在の SXMP-1 では，XMP のローカルビューとタスク並列化の機能を実装していない．これらの機能の実装は今後の課題である．また，SXMP では，Python 言語のクラス定義，文字列処理，リスト，タプル，辞書，集合などのデータ構造は部分的にしかサポートしていない．これらを分散並列実行時にどのように扱うかについても検討する予定である．

SXMP では，配列の記述方法や実装は NumPy をベースとしているが，現時点では配列を操作する関数や数学関数をサポートしていない．これらを分散並列実行時に扱う方法に関しても今後の課題である．

5.2 自動化

現在の SXMP-0 の実装では，指定した対象範囲から抽出した並列性は分散並列環境でのみ利用している．Pluto+ のアルゴリズムでは，多次元スケジュールによって，ループのスレッド並列化や，ベクトル化，SIMD 化で利用する並列性も同時に抽出することができる．今後は，これらの並列性を利用したコードを自動生成することを検討する予定である．また，Pluto+ のアルゴリズムの応用として，計算ノード間の通信用のコードの生成も含めた自動分散並列化手法 [4] が提案されている．このアルゴリズムでは，すべての計算ノードがローカル配列をコピーしたメモリ領域をもつことを前提としており，XMP や SXMP のように計算ノード間に配列を分散配置することを考慮していない．我々はこのアルゴリズムを応用して，SXMP-1 の分散並列化の指示を完全自動化する手法を実装する予定である．

5.3 実行性能の改善

SXMP のインタプリタモードは，現時点では実行結果の確認にのみ利用している．インタプリタモードでの実行性能の改善は今後の課題である．SXMP のコンパイルモードに関しては，XMPC と比較した時に，ループ内に関しては同じ記述になり，差分はループ外でのみ発生するようにする，source-to-source 変換の後処理の導入を検討している．この改善によって，SXMP で XMPC とほぼ同等の性能を達成できると予想している．

6. 関連研究

PGAS モデルの分散メモリ並列計算機用プログラミング言語として，既存言語を拡張した言語と独自の言語が存在する．

Fortran 言語を拡張した PGAS モデルのプログラミング言語としては，High Performance Fortran（HPF）[14] がある．HPF は，OpenMP や XMP と同様に逐次実行形式のプログラムに指示文を付加することで分散並列化を実現することができる．HPF はノード間通信をコンパイラが自動的に生成する点で XMP とは異なる．また，Fortran 言語の拡張である Co-Array Fortran（CAF）[18] の機能は，Fortran 2008 の言語仕様に採用され，XcalableMP/-Fortran からも利用することができる．C/C++ 言語をベースとした PGAS モデルのプログラミング言語として，Unified Parallel C（UPC）[23]，UPC++[24]，XMPC が存在する．XMPC では，Fortran 2008 と同様に Co-Array 機能を利用できるように C 言語を拡張している．分散並列環境向けの Python 言語の拡張として，Data-Centric（DaCe） Python[27] が提案されている．DaCe Python では，Python 言語のプログラムをデータフロー形式のプログラムに変換し，データをストリーム処理として最適化し，かつコンパイラを利用することで高速化と移植性を実現している．この言語では，アノテーションと拡張データ型を利用することで生産性と高めているが，データの参照と定義のパターンが複雑でデータフロー形式への変換が難しいプログラムについては高性能化の困難が予想される．XMP では，Python 言語から XMP のプログラムを利用する機能 [16] が実装されている．SXMP-1 とは異なり，この機能では Python 言語のレベルで分散並列化の指定を行うことはできない．

PGAS プログラミングモデルをサポートする独自言語としては，Chapel[7]，Fortress[3], X10[8] が提案されている．

7. おわりに

本報告では, PGAS 言語 XMP と同等の機能をもち, かつプログラムを Python 言語ベースで記述可能な分散メモリ並列計算機用スクリプト言語 SXMP を提案した．SXMP は基本的に XMP の機能を継承しており，かつ動的型付け言語とインタプリタを利用したプログラムの開発が可能であるという利点をもつ．さらに，SXMP では，XMP の機能に加えて，ソースコードレベルで自動並列化やループの最適化によるプログラム変換を利用可能であり，それらを活用して分散並列プログラムの開発を支援できることを示した．himeno ベンチマークプログラムを利用した予備評価では，SXMP のプログラムを Python インタプリタ上で実行した場合は XMPC と比較して大幅に遅いけれども，SXMP のプログラムを C 言語に変換した場合は XMPC と比較して 10% 程度の性能低下で実行できることを示した．今後は，実行性能の改善と，XMP の機能で SXMP では未実装であるローカルビューとタスク並列化の機能の実

装を行う予定である．また，自動分散並列化と自動タスク並列化についても実装を検討する予定である．

謝辞 SXMP に関して貴重なご意見を頂いた理化学研究所計算科学研究センターの佐藤三久先生に感謝いたします．

参考文献

[1] Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., Kudlur, M., Levenberg, J., Monga, R., Moore, S., Murray, D. G., Steiner, B., Tucker, P., Vasudevan, V., Warden, P., Wicke, M., Yu, Y. and Zheng, X.: TensorFlow: A System for Large-Scale Machine Learning, *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, Savannah, GA, USENIX Association, pp. 265–283 (online), available from ⟨https://www.usenix.org/conference/osdi16/technical-sessions/presentation/abadi⟩ (2016).

[2] Aho, A. V., Lam, M. S., Sethi, R. and Ullman, J. D.: *Compilers-Principles, Techniques, and Tools*, Addison-Wesley (2007).

[3] Allen, E., Chase, D., Hallett, J., Luchangco, V., Maessen, J.-W., Ryu, S., Steele Jr, G. L., Tobin-Hochstadt, S., Dias, J., Eastlund, C. et al.: The Fortress language specification, *Sun Microsystems*, Vol. 139, No. 140, p. 116 (2005).

[4] Bondhugula, U.: Compiling affine loop nests for distributed-memory parallel architectures, *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pp. 1–12 (online), DOI: 10.1145/2503210.2503289 (2013).

[5] Bondhugula, U., Acharya, A. and Cohen, A.: The Pluto+ Algorithm: A Practical Approach for Parallelization and Locality Optimization of Affine Loop Nests, *ACM Trans. Program. Lang. Syst.*, Vol. 38, No. 3 (online), DOI: 10.1145/2896389 (2016).

[6] Bondhugula, U., Hartono, A. and Ramanujam, J.: A practical automatic polyhedral parallelizer and locality optimizer, In PLDI ' 08: Proceedings of the ACM SIGPLAN 2008 conference on Programming language design and implementation (2008).

[7] Chamberlain, B., Callahan, D. and Zima, H.: Parallel Programmability and the Chapel Language, *The International Journal of High Performance Computing Applications*, Vol. 21, No. 3, pp. 291–312 (online), DOI: 10.1177/1094342007078442 (2007).

[8] Charles, P., Grothoff, C., Saraswat, V., Donawa, C., Kielstra, A., Ebcioglu, K., von Praun, C. and Sarkar, V.: X10: An Object-Oriented Approach to Non-Uniform Cluster Computing, *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, New York, NY, USA, Association for Computing Machinery, p. 519–538 (online), DOI: 10.1145/1094811.1094852 (2005).

[9] Dalcín, L., Paz, R. and Storti, M.: MPI for Python, *Journal of Parallel and Distributed Computing*, Vol. 65, No. 9, pp. 1108–1115 (online), DOI: https://doi.org/10.1016/j.jpdc.2005.03.010 (2005).

[10] Feautrier, P.: Dataflow analysis of array and scalar references, *International Journal of Parallel Programming*, Vol. 20, No. 1, pp. 23–53 (1991).

[11] Feautrier, P.: Some efficient solutions to the affine scheduling problem: I. One-dimensional time, *Int. J. Parallel Program.*, Vol. 21, pp. 313–348 (1992).

[12] Girbal, S., Vasilache, N., Bastoul, C., Cohen, A., Parello, D., Sigler, M. and Temam, O.: Semi-Automatic Composition of Loop Transformations for Deep Parallelism and Memory Hierarchies, *International Journal of Parallel Programming*, Vol. 34, No. 3, pp. 261–317 (2006).

[13] Grosser, T., Zheng, H., Aloor, R., Simbürger, A., Größlinger, A. and Pouchet, L.-N.: Polly-Polyhedral optimization in LLVM, *Proceedings of the First International Workshop on Polyhedral Compilation Techniques (IMPACT)*, Vol. 2011, p. 1 (2011).

[14] High Performance Fortran: http://hpff.rice.edu/. Accessed: Feb. 10, 2022.

[15] Lee, J. and Sato, M.: Implementation and Performance Evaluation of XcalableMP: A Parallel Programming Language for Distributed Memory Systems, *2010 39th International Conference on Parallel Processing Workshops*, pp. 413–420 (online), DOI: 10.1109/ICPPW.2010.62 (2010).

[16] Nakao, M.: *Mixed-Language Programming with XcalableMP*, pp. 147–163, Springer Singapore (2021).

[17] NumPy: https://numpy.org/. Accessed: Feb. 10, 2022.

[18] Numrich, R. W. and Reid, J.: Co-Array Fortran for Parallel Programming, *SIGPLAN Fortran Forum*, Vol. 17, No. 2, p. 1–31 (online), DOI: 10.1145/289918.289920 (1998).

[19] Omni Compiler: https://github.com/omni-compiler/. Accessed: Feb. 10, 2022.

[20] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L. and Lerer, A.: Automatic differentiation in pytorch (2017).

[21] Quilleré, F., Rajopadhye, S. and Wilde, D.: Generation of Efficient Nested Loops from Polyhedra, *International Journal of Parallel Programming*, Vol. 28, p. 2000 (2000).

[22] Trifunovic, K., Cohen, A., Edelsohn, D., Li, F., Grosser, T., Jagasia, H., Ladelsky, R., Pop, S., Sjödin, J. and Upadrasta, R.: GRAPHITE Two Years After: First Lessons Learned From Real-World Polyhedral Compilation, *GCC Research Opportunities Workshop (GROW'10)*, Pisa, Italy, (online), available from ⟨https://hal.inria.fr/inria-00551516⟩ (2010).

[23] Unified Parallel C: https://upc.lbl.gov/. Accessed: Feb. 10, 2022.

[24] Unified Parallel C++: https://upcxx.lbl.gov/. Accessed: Feb. 10, 2022.

[25] Verdoolaege, S.: isl: An Integer Set Library for the Polyhedral Model, *Mathematical Software - ICMS 2010* (Fukuda, K., Hoeven, J., Joswig, M. and Takayama, N., eds.), Lecture Notes in Computer Science, Vol. 6327, Springer, pp. 299–302 (2010).

[26] XcalableMP: https://xcalablemp.org/. Accessed: Feb. 10, 2022.

[27] Ziogas, A. N., Schneider, T., Ben-Nun, T., Calotoiu, A., De Matteis, T., de Fine Licht, J., Lavarini, L. and Hoefler, T.: Productivity, Portability, Performance: Data-Centric Python, *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '21, New York, NY, USA, Association for Computing Machinery, (online), DOI: 10.1145/3458817.3476176 (2021).

[28] 中田育男：コンパイラの構成と最適化（第 2 版），朝倉書店 (2009).