

# Pythonと連携するPGAS言語XcalableMPの プログラミングモデル

## - Graph Order/degree 問題への適用 -

中尾 昌広<sup>1,a)</sup> 村井 均<sup>1</sup> 朴 泰祐<sup>2,3</sup> 佐藤 三久<sup>1</sup>

**概要:** 分散メモリ環境におけるアプリケーションの開発には、高い生産性と性能を発揮できる PGAS (Partitioned Global Address Space) 言語が利用されることがある。しかしながら、効率的にアプリケーションの開発を行うには、PGAS 言語のみでアプリケーションを開発するのではなく、PGAS 言語と他の言語とを組合せて用いることも重要である。そこで、本稿では PGAS 言語の 1 つである XcalableMP (XMP) に着目し、XMP のコンパイラ実装である Omni Compiler に対して C・Fortran・Python と XMP との連携機能を作成した。本稿では、その機能の実装について述べる。さらに、Python と XMP との連携機能の実例として、Graph Order/degree 問題に対するアプリケーションを作成した。Graph Order/degree 問題の中で計算コストが最も高い全頂点間最短経路探索を XMP を用いて並列化することで、アプリケーションの高速化を行った。XMP で記述したアプリケーションとオリジナルの Python で記述したアプリケーションとを比較した結果、1CPU コアを用いた場合は、21%の性能向上を達成した。また、XMP で記述したアプリケーションを 1280CPU コア (64 計算ノード) を用いた結果、1CPU コアを利用した場合と比較して 921 倍の高速化を達成した。

## Cooperation of Python and XcalableMP languages - Application to Graph Order/degree Problem -

MASAHIRO NAKAO<sup>1,a)</sup> HITOSHI MURAI<sup>1</sup> TAISUKE BOKU<sup>2,3</sup> MITSUHIISA SATO<sup>1</sup>

### 1. はじめに

分散メモリ環境におけるアプリケーションの開発に、仮想的な大域アドレス空間を用いることで高い生産性を発揮できる PGAS (Partitioned Global Address Space) 言語が利用されることがある [1–4]。PGAS 言語は片側通信と親和性が高いため、ハードウェアに近い通信レイヤを直接利用することにより、MPI ライブラリで記述されたアプリケー

ションよりも高い性能を発揮する場合もある [4, 5]。PGAS 言語の例としては、XcalableMP (XMP) [4, 6–8], XcalableACC (XACC) [9–11], Coarray Fortran (CAF) [12], PCJ [13], Unified Parallel C (UPC) [14], UPC++ [15], HabaneroUPC++ [16], X10 [17], Chapel [18], DASH [19] などがある。

PGAS 言語には多くの利点が存在するが、既存の MPI ライブラリで記述されたアプリケーションを PGAS 言語を用いて再実装することは、次の理由により現実的ではない場合が多い。(1) プログラミングコストが大きい、(2) PGAS 言語によって生産性や性能が向上するケースは限定的である、(3) PGAS 言語の中で今後もサポートが続くと考えられるのは、Fortran 2008 から導入された CAF の一部の機能のみであり、他の PGAS 言語のサポートの今後について

<sup>1</sup> 理化学研究所 計算科学研究機構  
RIKEN Advanced Institute for Computational Science  
<sup>2</sup> 筑波大学 計算科学研究センター  
Center for Computational Sciences, University of Tsukuba  
<sup>3</sup> 筑波大学 大学院 システム情報工学研究科  
Graduate School of Systems and Information Engineering,  
University of Tsukuba  
<sup>a)</sup> masahiro.nakao@riken.jp

は不明である。また、一般に1つのプログラミング言語には得手・不得手があるため、1つの汎用的なプログラミング言語やフレームワークのみで、すべての並列アプリケーションを作成することは困難である。

分散メモリ環境におけるアプリケーション開発においてPGAS言語の利点を活かすためには、PGAS言語と他言語との連携が重要であると考えられる。例えば、MPIライブラリで記述されたあるアプリケーションの性能もしくはコードの見通しが良くなる箇所のみをPGAS言語で記述する、などである。既存のアプリケーションコードに対して部分的にコード変更を行うことは、全体的にコード変更を行うことよりもコストは小さいため、前段落で挙げた3つの問題点も軽減できる。

我々は、PGAS言語XMPとその処理系であるOmni Compiler [20]を開発している。本稿ではOmni Compilerに対して、MPIライブラリを利用しているC・Fortran・Pythonの各言語とXMPとの連携機能を作成する。また、PythonとXMPとの連携機能の実例として、Graph Order/degree問題に対するアプリケーションを作成する。Pythonは科学技術計算で利用される多くのライブラリを持つため、PythonとPGAS言語との連携を行うことは、HPCアプリケーション開発におけるプログラミングコストの大きな削減につながると考えている。

以下、本稿は次のような構成になっている。2章と3章では、それぞれXMPの概要とOmni Compilerについて述べる。4章では、Omni CompilerにおけるC・Fortran・Pythonの各言語とXMPとの連携機能の開発について述べる。5章では、Graph Order/degree問題に対するアプリケーション開発について述べる。6章では、本稿のまとめと今後の課題について述べる。

## 2. XcalableMP

### 2.1 概要

XMPはPC Cluster Consortium [21]の並列プログラミング言語XMP規格部会によって仕様策定が行われている並列言語である。XMPはHPC分野でよく用いられているC言語およびFortranに対して、並列化のための指示文およびCoarrayのための拡張構文を提供している。現在、C++への対応も進めている。本稿では、C言語版のXMPをXMP/C、Fortran版のXMPをXMP/Fortranと呼称する。XMPの仕様の一部は、CAFとHigh Performance Fortran [22,23]がベースとなっており、XMP/FortranはFortran 2008の上位互換である。

図1にXMPの実行モデルを示す。XMPの実行モデルはSingle Program Multiple Data (SPMD)である。実行ユニットを“ノード”と呼び、各ノードは同じプログラムを実行する。各ノードは指示文やCoarray構文の行において、他のノードとの通信や同期を行う。

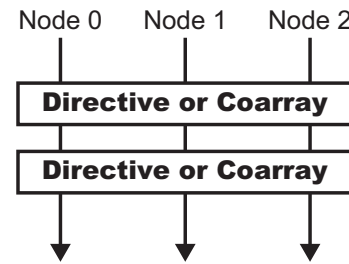


図 1: XcalableMP の実行モデル

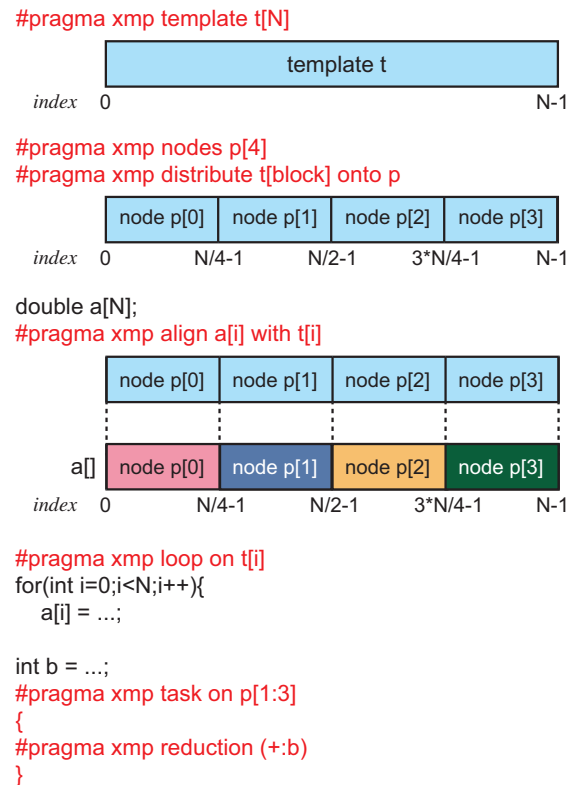


図 2: 分散配列の定義と並列処理の例

### 2.2 プログラム例

XMPが提供する指示文は、ノード集合の定義、分散配列の定義、ループ文の分散などを行うことができる。XMPでは分散配列を定義するために、仮想インデックス集合である“テンプレート”を用いる。図2にXMP/Cにおける分散配列の定義と並列処理の例を示す。

- (1) **template** 指示文はテンプレート  $t$  を定義する。図2の  $t$  のインデックスは  $0 \sim N-1$  である。
- (2) **node** 指示文はノード集合  $p$  を定義する。図2では、 $p$  は  $p[0] \sim p[3]$  の4ノードで構成されている。もし、 $p[*]$  のように角カッコ内に“\*”が指定されていた場合は、実行時にノードの数が決定する。
- (3) **distribute** 指示文はテンプレート  $t$  をノード集合  $p$  に指定した分散方式で割り当てる。図2の場合はブロック分散を指定しており、可能な限り同じブロックサイズ ( $\lceil (\text{template size}) / (\text{size of node set}) \rceil$ ) でテンプレートを分割する [8]。

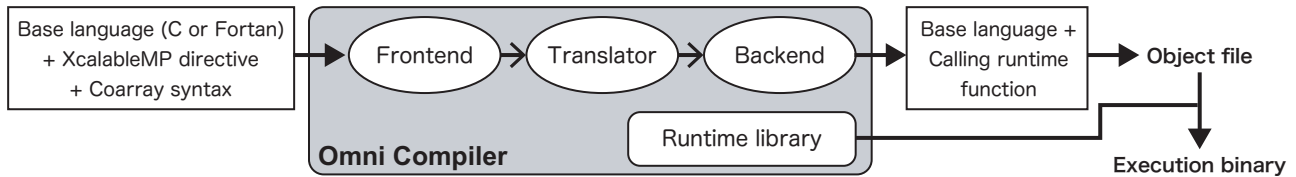


図 3: Omni Compiler のコンパイルフロー

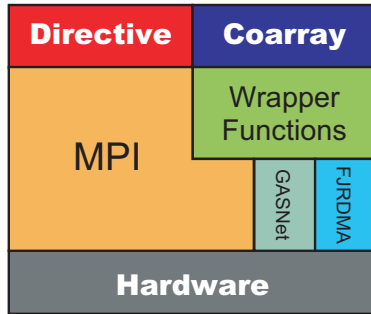


図 4: Omni Compiler の通信ランタイム

- (4) **align** 指示文は分散配列  $a[]$  を、分散されたテンプレート  $t$  に整列させる。図 2 で  $N=16$  の場合、各ノードは分散配列  $a[]$  の 4 要素ずつを持つ。
- (5) **loop** 指示文は分散されたテンプレート  $t$  に従って、ループ文を分割する。図 2 で  $N=16$  であるならば、ノード  $p[0]$  はインデックス  $i$  の 0~3 を実行する。
- (6) **task** 指示文はある特定のノード集合のみを処理する。図 2 では、**task** 指示文で囲まれた括弧内は、 $p[1] \sim p[3]$  の 3 ノードだけが実行する。
- (7) **reduction** 指示文は集約演算を行う。図 2 は **task** 指示文の中で **reduction** 指示文が用いられているため、 $p[1] \sim p[3]$  の 3 ノードの間でのみ、ローカル変数  $b$  に対する集約演算が発生する。

### 3. Omni Compiler

#### 3.1 概要

Omni Compiler は理化学研究所と筑波大学で開発が行われている source-to-source コンパイラである。Omni Compiler は、Linux クラスタおよび様々なクラスタシステム (e.g. スーパーコンピュータ「京」[24], 富士通 FX10・FX100, NEC SX-9, 日立 SR16000, Cray XE6, IBM BlueGene/Q など) で動作する。

#### 3.2 コンパイルフロー

図 3 に Omni Compiler のコンパイルフローを示す。Omni Compiler により、XMP 指示文と Coarray 構文で記述されたコードはランタイムの呼び出しに変換される。その変換されたコードはネイティブコンパイラ (gcc など) によってオブジェクトファイルが生成され、最後に Omni Compiler が用意しているランタイムとリンクされた後、実行バイナリが生成される。

```

1 void foo(){
2     printf("Hello\n");
3 }
4
5 int main(){
6     foo();
7     return 0;
8 }

```

(a) 変換前のコード

```

1 void foo(){
2     printf("Hello\n");
3 }
4
5 static int xmpc_main(int argc, char **argv){
6     foo();
7     return 0;
8 }
9
10 int main(int argc, char **argv){
11     xmpc_init_all(argc, argv);
12     int r = xmpc_main(argc, argv);
13     xmpc_finalize_all();
14     return r;
15 }

```

(b) 変換後のコード

図 5: Omni Compiler の変換例

図 4 に通信に関するランタイムの構成図を示す。XMP 指示文に関するランタイムは、MPI ライブラリを用いて実装されている。Coarray に関するランタイムは片側通信ライブラリ毎に下記の 3 つの実装があり、Omni Compiler のインストール時にユーザが選択する。

- MPI (MPI Version 3 以降の片側通信関数)
- GASNet [25]
- FJRDMA (富士通の一部のクラスタシステムが提供している拡張 RDMA インタフェース)

なお、トランスレータの実装を簡易化するため、どの片側通信ライブラリを用いても、Omni Compiler は Coarray 構文を同じ関数 (ラッパー関数) に変換する。Omni Compiler では、そのラッパー関数を介して片側通信ライブラリの内の 1 つを利用する。

#### 3.3 変換例

XMP/C における Omni Compiler によるコードの変換例を図 5 に示す。Omni Compiler は図 5a のコードを図 5b

表 1: XMP プログラムから MPI プログラムを利用するための関数

| Language       | Return Value Type  | Function                                      | Description                  |
|----------------|--------------------|-----------------------------------------------|------------------------------|
| XMP/C<br>XMP/F | void<br>(None)     | xmp_init_mpi(int*, char***)<br>xmp_init_mpi() | MPI 環境の初期化処理                 |
| XMP/C<br>XMP/F | MPIComm<br>integer | xmp_get_mpi_comm(void)<br>xmp_get_mpi_comm()  | 実行中のノード集合に対する MPI コミュニケータの取得 |
| XMP/C<br>XMP/F | void<br>(None)     | xmp_finalize_mpi(void)<br>xmp_finalize_mpi()  | MPI 環境の完了処理                  |

のコードに変換する。XMP/Fortran においても、ほぼ同等の操作が行われるため、本稿では XMP/Fortran の例は省略する。

変更後のコードに自動挿入される図 5b の 11 行目の **xmp\_init\_all** 関数と 13 行目の **xmp\_finalize\_all** 関数は、それぞれ XMP に対する初期化処理、終了処理を行う関数であり、ランタイム内で定義されている。**xmp\_init\_all** 関数の中では MPI ライブラリの初期化関数である **MPI\_Init** 関数が利用されているため、コマンドライン引数が必要になる。そのため、Omni Compiler は、変換前の **main** 関数の引数にコマンドラインの情報がなければ、変換後の **main** 関数にコマンドラインの情報を自動的に追加する。また、**xmp\_finalize\_all** 関数の中では、**MPI\_Finalize** 関数が呼び出されている。

**xmp\_init\_all** 関数の中では、**MPI\_Comm\_dup** 関数を用いることで、**MPI\_COMM\_WORLD** の複製を行い、その複製されたコミュニケータを用いて XMP が提供する通信を行う<sup>\*1</sup>。XMP プログラムの中では、図 2 の **task** 指示文を用いることで、ノード集合を分割することも可能である。**task** 指示文の実装では、**MPI\_Comm\_split** 関数を用いることで、複製されたコミュニケータをもとに新しいコミュニケータを生成している。図 2 のように **task** 指示文の中で通信が発生する場合は、その新しいコミュニケータが利用される。

Omni Compiler は変換前の **main** 関数を **xmpc\_main** 関数にリネームし、**xmpc\_main** 関数を変換後の **main** 関数の中から呼ぶように変更する。上記のような特別なコード変換は **main** 関数についてのみ行われる。それ以外の関数（例えば、図 5 中の **foo** 関数）については、変換は行われない。

#### 4. Omni Compiler における他言語との連携機能

Omni compiler に対し、下記の機能を作成した。それぞれについて説明する。

- XMP プログラムから MPI プログラムを利用する機能

<sup>\*1</sup> この手順を踏んでいるため、XMP プログラムの中でユーザが **MPI\_COMM\_WORLD** を用いた MPI の通信を行っても、ユーザの通信と XMP の通信とは衝突しない。

```

1 #include <xmp.h>
2 #include <mpi.h>
3 #pragma xmp nodes p[*]
4
5 int main(int argc, char **argv){
6     xmp_init_mpi(&argc, &argv);
7     MPI_Comm comm = xmp_get_mpi_comm();
8     call_mpi(comm);
9     xmp_finalize_mpi();
10
11     return 0;
12 }
```

(a) XMP のコード (xmp.c)

```

1 #include <mpi.h>
2
3 void call_mpi(MPI_Comm comm){
4     int rank, size;
5     MPI_Comm_rank(comm, &rank);
6     MPI_Comm_size(comm, &size);
7     :
8 }
```

(b) MPI のコード (mpi.c)

図 6: XMP プログラムから MPI プログラムを利用するプログラム例

- MPI プログラムから XMP プログラムを利用する機能
- Python プログラムから XMP プログラムを利用する機能

なお、XMP プログラムから Python プログラムを呼び出す機能は、利用頻度が少ないと考えられるため、本稿では作成しなかった。

#### 4.1 XMP プログラムから MPI プログラムを呼び出す機能

本機能を実現するため、表 1 の関数を開発した。表 1 の関数については、XMP 仕様書 Version 1.3 [8] の Appendix A で定義されている。これらの関数を利用した XMP/C のプログラム例を図 6 に示す。図 6a の 1 行目は、表 1 の関数を利用するために、XMP のヘッダファイルをインクルードしている。2 行目は、7 行目で取得する MPI コミュニケータの型の情報を得るために、MPI のヘッダファイルをインクルードしている。6 行目は、MPI 環境の初期化

表 2: MPI プログラムから XMP プログラムを利用するための関数

| Language | Return Value Type | Function           | Description  |
|----------|-------------------|--------------------|--------------|
| XMP/C    | void              | xmp_init(MPI_Comm) | XMP 環境の初期化処理 |
| XMP/F    | (None)            | xmp_init(Integer)  |              |
| XMP/C    | void              | xmp_finalize(void) | XMP 環境の完了処理  |
| XMP/F    | (None)            | xmp_finalize()     |              |

処理を行っている。7 行目は、現在実行しているノード集合の情報から MPI コミュニケータを作成し、それを返している。8 行目は、図 6b に示した MPI 関数の呼び出しを行っている。9 行目は、MPI 環境の完了処理を行っている。なお、`xmp_get_mpi_comm` 関数と MPI 関数の呼び出しは、`xmp_init_mpi` 関数と `xmp_finalize_mpi` 関数の間で行う必要がある。

実装的には、`xmp_init_mpi` 関数と `xmp_finalize_mpi` 関数は、空の関数である。なぜなら、図 5b と 3.3 節に示した通り、プログラムの最初と最後では必ず `xmp_init_all` 関数と `xmp_finalize_all` 関数が呼び出され、各関数の中で MPI 環境の初期化処理・完了処理が行われるからである。次に、`xmp_get_mpi_comm` 関数の実装について述べる。3.3 節で述べた通り、`task` 指示文は新しいコミュニケータを生成する。そのため、MPI コミュニケータは、ランタイム内ではスタックの形で保持されている。`xmp_get_mpi_comm` 関数では、そのスタックの一番上にあるコミュニケータを返すことにより実現している。

Omni Compiler を用いたコンパイルと実行例は下記の通りである\*2。Omni Compiler の XMP/C コンパイラのコマンド名は `xmpcc` である。

```
$ mpicc mpi.c -c -o mpi.o
$ xmpcc xmp.c -c -o xmp.o
$ xmpcc mpi.o xmp.o -o a.out
```

なお、Omni Compiler は内部で MPI コンパイラを用いているので、下記のように MPI プログラムも `xmpcc` コマンドでコンパイルすることができる。

```
$ xmpcc mpi.c -o mpi.o
```

そのため、下記のようにすべてのコンパイル作業を 1 行で実行することも可能である。

```
$ xmpcc mpi.c xmp.c -o a.out
```

実行バイナリはユーザの MPI 環境が提供している実行コマンド (`mpirun` など) を用いて実行する。

\*2 リンカを `mpicc` コマンドで行うことも可能であるが、その場合は Omni Compiler が用意しているライブラリを指定する必要がある。

```
1 #include <xmp.h>
2 #include <mpi.h>
3
4 int main(int argc, char **argv){
5     MPI_Init(&argc, &argv);
6
7     xmp_init(MPI_COMM_WORLD);
8     call_xmp();
9     xmp_finalize();
10
11     MPI_Finalize();
12     return 0;
13 }
```

(a) MPI のコード (mpi.c)

```
1 void call_xmp(){
2     #pragma xmp nodes p[*]
3     :
4 }
```

(b) XMP のコード (xmp.c)

図 7: MPI プログラムから XMP プログラムを利用するプログラム例

```
$ mpirun -np 4 a.out
```

## 4.2 MPI プログラムから XMP プログラムを呼び出す機能

本機能を実現するため、表 2 の関数を開発した。これらの関数を利用した XMP/C のプログラム例を図 7 に示す。図 7a の 1 行目は、表 2 の関数を利用するために、XMP のヘッダファイルをインクルードしている。7 行目は、XMP 環境の初期化処理を行っている。`xmp_init` 関数の引数には、MPI コミュニケータを指定する。8 行目は、図 7b に示した XMP 関数の呼び出しを行っている。9 行目は、XMP 環境の完了処理を行っている。なお、XMP 関数の呼び出しは、`xmp_init` 関数と `xmp_finalize` 関数の間で行う必要がある。また、呼び出される XMP 関数のノード集合は、`xmp_init` 関数で指定されたコミュニケータをもとに作成される。

`xmp_init` 関数の実装は、基本的には `xmp_init_all` 関数を呼び出しているが、下記の工夫を行っている。

- `xmp_init` 関数の前では、ユーザプログラム側で必ず

表 3: Python プログラムから XMP プログラムを利用するための関数

| Return Value Type | Function                                   | Description  |
|-------------------|--------------------------------------------|--------------|
| (None)            | init_py(ctypes.CDLL, mpi4py.MPI.Intracomm) | XMP 環境の初期化処理 |
| (None)            | finalize_py(ctypes.CDLL)                   | XMP 環境の完了処理  |

```

1 from mpi4py import MPI
2 import ctypes
3 import xmp
4
5 lib = ctypes.CDLL('test.so')
6 xmp.init_py(lib, MPI.COMM_WORLD)
7 lib.test()
8 xmp.finalize_py(lib)

```

(a) Python のコード (test.py)

```

1 void test(){
2 #pragma xmp nodes p[*]
3 :
4 }

```

(b) XMP のコード (test.c)

図 8: Python プログラムから XMP プログラムを利用するプログラム例

MPI.Init 関数がすでに呼び出されている。そのため、xmp\_init 関数から xmp\_init\_all 関数を呼び出した場合のみ、xmp\_init\_all 関数中の MPI\_Init 関数を呼び出さない処理を追加した。

- 3.3 節で説明した通り、xmp\_init\_all 関数の中では、MPI\_Comm\_dup 関数を用いて MPI\_COMM\_WORLD を複製している。そこで、xmp\_init から xmp\_init\_all 関数を呼び出す場合は、複製されるコミュニケータは xmp\_init で指定したコミュニケータにする処理を追加した。

xmp\_finalize 関数の実装も、基本的には xmp\_finalize\_all 関数を呼び出していることで実現している。xmp\_init 関数と同様に、xmp\_finalize 関数の後では、ユーザプログラム側で必ず MPI\_Finalize 関数が呼び出されているため、xmp\_finalize 関数から xmp\_finalize\_all 関数を呼び出した場合のみ、xmp\_finalize\_all 関数中の MPI\_Finalize 関数は呼び出さない処理を追加した。

Omni Compiler を用いたコンパイルと実行例は、4.1 節と同じである。

#### 4.3 Python プログラムから XMP プログラムを呼び出す機能

本機能を実現するため、Python のパッケージとして表 3 の関数を開発した。これらの関数を利用した Python のプログラム例を図 8 に示す。“ctypes”という共有ライ

```

1 def init_py(lib, comm):
2     fcomm = comm.py2f()
3     lib.xmp_init_py(fcomm)
4
5 def finalize_py(lib):
6     lib.xmp_finalize()

```

図 9: Python の XMP パッケージ

```

1 void xmp_init_py(MPI_Fint comm) {
2     xmp_init(MPI_Comm_f2c(comm));
3 }

```

図 10: xmp\_init\_py 関数

リ内の関数呼び出しを可能する外部関数ライブラリと、“mpi4py”という Python 用の MPI ライブラリの利用を想定している。

図 8a の 3 行目は、今回作成した Python の XMP パッケージをインポートする。5 行目は、図 8b のファイルから作成される共有ライブラリ test.so を読み込んでいる。6 行目は、XMP 環境の初期化処理を行っている。init\_py 関数の引数には、共有ライブラリと MPI コミュニケータを指定する。7 行目は、図 8b の呼び出しを行っている。8 行目は、XMP 環境の完了処理を行っている。finalize\_py 関数の引数には、共有ライブラリを指定する。なお、XMP 関数の呼び出しは、init\_py 関数と finalize\_py 関数の間で行う必要がある。また、呼び出される XMP 関数のノード集合は、init\_py 関数で指定されたコミュニケータをもとに作成される。

init\_py 関数と finalize\_py 関数が定義してある XMP パッケージの Python プログラムを図 9 に示す。mpi4py パッケージでは、MPI コミュニケータは Fortran 形式に変換することはできるが、C 言語の形式に変換することはできない。そこで、まず init\_py 関数において、MPI コミュニケータ comm を py2f メソッドを使って Fortran 形式の MPI コミュニケータ (MPI\_Fint 型) に変換する。次に、Omni Compiler のランタイムに作成した xmp\_init\_py 関数に処理を引き継ぐ。xmp\_init\_py 関数を図 10 に示す。xmp\_init\_py 関数では、MPI\_Comm\_f2c 関数を利用することにより、与えられた Fortran 形式の MPI コミュニケータを C 言語の形式に変換し、表 2 の xmp\_init 関数を利用して XMP 環境の初期化を行う。

Python から Omni Compiler の提供するランタイムを利用するためには、そのランタイムは共有ライブラリで

ある必要がある．そのため，Omni Compiler をコンパイルする際に，共有ライブラリも生成するためのオプションを作成した．下記のように，`./configure` スクリプトに`--enable-shared` を付加すると，共有ライブラリを生成できるようにした．

```
$ ./configure --enable-shared
```

Omni Compiler を用いたコンパイルおよび実行例は下記の通りである．まず，`xmpcc` コマンドにより，共有ライブラリを作成する．共有ライブラリを作成するためのコンパイルオプションはネイティブコンパイラに依存する．ネイティブコンパイラが `gcc` の場合は`-fPIC -shared` である．実行は，Python を介して MPI の実行コマンドを用いる．

```
$ xmpcc -fPIC -shared test.c -o test.so
$ mpirun -np 4 python test.py
```

## 5. Graph Order/degree 問題への適用

### 5.1 Graph Order/degree 問題とは

Graph Order/degree 問題とは，与えられた頂点数と次数を持つ無向グラフの直径と直径間の平均距離（ASPL : Average Shortest Path Length）を最小化することであり，低遅延な相互結合網の設計に役立つとされている [26]．頂点数 ( $n$ ) と次数 ( $d$ ) から，理論的な直径の下界 ( $K_{n,d}$ ) と平均距離の下界 ( $L_{n,d}$ ) は，下記のように計算できる [26,27]．

$$K_{n,d} = \begin{cases} \left\lceil \frac{n-1}{2} \right\rceil & \text{if } d = 2 \\ \left\lceil \log_{d-1} \left( \frac{(n-1)(d-2)}{d} \right) + 1 \right\rceil & \text{if } d > 2 \end{cases} \quad (1)$$

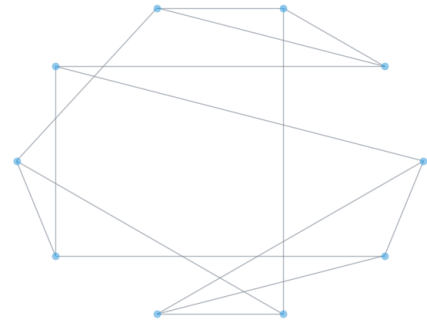
$$L_{n,d} = \begin{cases} 1 & \text{if } K_{n,d} = 1 \\ \frac{S_{n,d} + K_{n,d} R_{n,d}}{n-1} & \text{if } K_{n,d} \geq 2 \end{cases} \quad (2)$$

$$S_{n,d} = \sum_{i=1}^{K_{n,d}-1} id(d-1)^{i-1} \quad (3)$$

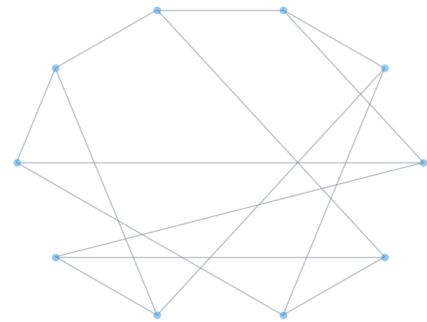
$$R_{n,d} = n-1 - \sum_{i=1}^{K_{n,d}-1} d(d-1)^{i-1} \quad (4)$$

$n = 10, d = 3$  の場合のグラフ例を図 11 に示す．図 11a はランダムにエッジを配置したグラフであり，図 11b は 5.2 節で述べるアルゴリズムを用いて最適化した結果である．図 11b の直径と平均距離は下界である．

Graph Order/degree 問題をオープンサイエンスの対象に取り上げる試みとして，国立情報学研究所は，その直径と平均距離の小ささを競うコンペティション “Graph Golf” を開催している [28]．このコンペティションは 2015 年から毎年開催されており，年ごとにいくつかの頂点数と次数



(a) 直径 3, 平均距離=1.89



(b) 直径 2, 平均距離=1.67

図 11:  $n = 10, d = 3$  の場合の例

の組合せを出題している．出題は，頂点が自由に配置できる問題である “General Graph Category” とグリッド上に配置される問題である “Grid Graph Category” がある．本稿では General Graph Category について扱う．2017 年の General Graph Category の頂点数と次数の組合せを表 4 に示す．

Graph Golf の公式サイト [28] で，Graph Order/degree 問題に対する Python プログラムである “create-random.py” が公開されている<sup>\*3</sup>．この Python プログラムは頂点数と次数を引数に指定することで，下記を出力する．

- ランダムな初期解（図 11a のグラフはこの機能を用いて作成した）
- 初期解の直径と平均距離の出力
- 初期解の図を PNG 形式で保存（図 11 の 2 つのグラフの図はこの機能を用いて作成した）

これらの計算には，Python の `networkx` パッケージが利用されている [29]．

### 5.2 実装

Graph Golf が公開している `create-random.py` は，グラフの初期解は生成するが，その最適化は行わない．また，直径と平均距離の計算には，全頂点間最短経路を求める

<sup>\*3</sup> <http://research.nii.ac.jp/graphgolf/py/create-random.py>

表 4: 2017 年の General Graph Category の頂点と次数の組合せ

| 頂点数 ( $n$ ) | 32 | 256 | 576 | 1344 | 4896 | 9344 | 88128 | 98304 | 100000 | 100000 |
|-------------|----|-----|-----|------|------|------|-------|-------|--------|--------|
| 次数 ( $d$ )  | 5  | 18  | 30  | 30   | 24   | 10   | 12    | 10    | 32     | 64     |

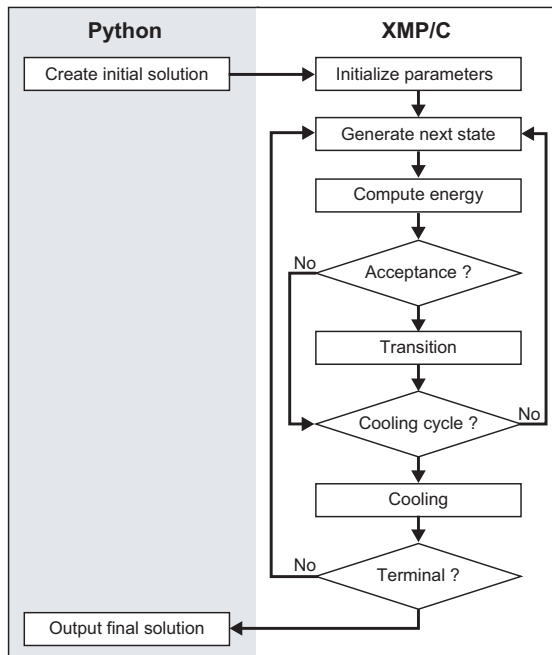


図 12: 作成したアルゴリズムのフローチャート

必要がある。create-random.py においては、networkx の `shortest_path_length` メソッドを用いて全頂点間最短経路を求めているが、その計算には多くの時間を要する。

そこで、グラフの最適化を行うコードを create-random.py と XMP/C を用いて作成する。最適化アルゴリズムには Simulated Annealing (SA) [30,31] を用いた。また、SA で必要になる全頂点間最短経路の計算には、XMP/C による並列化を行った。図 12 に、そのアルゴリズムのフローチャートを示す。初期解の生成とグラフの表示は create-random.py を用いており、それ以外の箇所は XMP/C を用いて作成した。

作成したコードの一部を図 13 に示す\*4。図 13a の 7～11 行目は、引数に与えられた頂点数と次数を整形している。13～16 行目は初期解を生成し、18～21 行目は初期解の情報を XMP/C で作成した `xmp_graphgolf` 関数に渡し、最適化を行う。なお、最適化後の解は、引数と同じ `arr` に格納される。23 行目以降は、解を図に変換し、保存している。図 13b の 1～3 行目は、XMP のテンプレート、ノード集合を定義し、そのテンプレートをノード集合にブロック分散している。template 指示文で “[:]” としているのは、定義の時点ではテンプレートの大きさは未定であることを示している。そして、8 行目の `template_fix` 指示文で、そのテンプレートの大きさを `nodes` と決定している。

\*4 プログラムの全コードは <https://github.com/mnakao/GraphGolf> にある。

表 5: coma システムのスペック

|          |                                                                                 |
|----------|---------------------------------------------------------------------------------|
| CPU      | Intel Xeon-E5 2670v2 2.8 GHz x 2 Sockets                                        |
| Memory   | DDR3 1866MHz 59.7GB/s 64GB                                                      |
| Network  | InfiniBand FDR 7GB/s                                                            |
| Software | intel/16.0.2, intelmpi/5.1.1, Omni Compiler 1.2.1<br>Python 2.7.9, networkx 1.9 |

10～12 行目は、頂点ごとの直径と平均距離を計算している(図 12 の “Compute Energy” の箇所であるが、図 13b では省略している)。これらの計算にはトップダウンアプローチの幅優先探索 [32] を用いた。なお、全頂点間最短経路探索の計算量のオーダーは  $O(n^2d)$  であり、通信量のオーダーは  $O(nd)$  である。9 行目の `loop` 指示文は、各ノードで並列に for ループ内の計算を行っている。13～15 行目は、集約演算を行うことで、直径と平均距離の計算を行っている。

### 5.3 評価

表 5 に示す筑波大学のクラスタシステム coma を用いて、全頂点間最短経路の計算の速度評価を行う。用いた頂点数と次数は、表 4 の中で中規模の  $n = 9344, d = 10$  である。

XMP で用いるノード数を変えて、1 回の全頂点間最短経路探索に要する時間の計測を行った。1 計算ノードにつき、XMP の実行ユニットは 20 ノードを割り当てた。結果を図 14 に示す。図中の棒グラフは計算時間を示しており、線グラフは 1 ノードに対する並列化効率を示している。

1 ノードの計算時間は 123.17 秒であるのに対し、1280 ノード (64 計算ノードを利用) の計算時間は 0.13 秒であり、921 倍の高速化を達成した。なお、Python の networkx パッケージを用いた場合の時間は 148.83 秒であり、1CPU コアを用いた場合の比較では、XMP は 21% の性能向上を達成した。

図 14 より、ノード数が増えるにつれ、並列化効率が悪くなるのがわかる。その理由は下記の 2 つが考えられる。

- 計算時間に対する通信の割合が増えるから。計算時間に対する通信割合を図 15 に示す。1280 ノード時の通信を占める割合は、全体の 21% である。
- 並列化対象のループ長が短くなるから。図 13b の 9 行目のループ文の長さは頂点数と同じ 9344 である。その長さをノード数で割るため、1 つのノードの最大長は 8 ( $= \lceil 9344/1280 \rceil$ ) になる。そのため、仮に通信時間が 0 であったとしても、並列化効率は  $0.91 (= 9344/8/1280)$  になる。

```

1 from mpi4py import MPI
2 import ctypes
3 import xmp
4 import networkx as nx
5 import argparse
6
7 argumentparser = argparse.ArgumentParser()
8 argumentparser.add_argument('vertex', type=int)
9 argumentparser.add_argument('degree', type=int)
10 vertex = args.vertex
11 degree = args.degree
12
13 arr = ((c_int * 2) * (vertex * degree / 2))()
14 for i, l in enumerate(nx.generate_edgelist(g, data=False)):
15     arr[i][0] = int(l.split()[0])
16     arr[i][1] = int(l.split()[1])
17
18 lib = ctypes.CDLL("xmp_graphgolf.so")
19 xmp.init_py(lib, MPI.COMM_WORLD)
20 lib.xmp_graphgolf(c_int(vertex*degree/2), arr, c_int(vertex))
21 xmp.finalize_py(lib)
22
23 if (MPI.COMM_WORLD.Get_rank() == 0):
24     image_name = "n"+str(vertex)+"d"+str(degree)+".png"
25     lines = []
26     for line in arr:
27         lines.append(str(line[0]) + " " + str(line[1]))
28     G = nx.parse_edgelist(lines, nodetype = int)
29     save_image(G, image_name)
30
31 def save_image(g, filepath):
32     import matplotlib as mpl
33     mpl.use('Agg')
34     import matplotlib.pyplot as plt
35     layout = nx.circular_layout(g)
36     nx.draw(g, with_labels=False, node_size=50,
37             linewidths=0, alpha=0.5, node_color='#3399ff',
38             edge_color='#666666', pos=layout)
39     plt.draw()
40     plt.savefig(filepath)

```

(a) Python プログラム

```

1 #pragma xmp template t[:];
2 #pragma xmp nodes p[*]
3 #pragma xmp distribute t[block] onto p
4
5 void xmp_graphgolf(int lines, int edge[lines][2], int vertices
6 )
7 {
8     :
9     #pragma xmp template_fix t[vertices]
10    #pragma xmp loop on t[i]
11        for(int i=0;i<vertices;i++){
12            : // Calculate diameter and ASPL
13        }
14    #pragma xmp reduction(+:ASPL)
15    #pragma xmp reduction(max:diameter)
16    :
17 }

```

(b) XMP/C プログラム

図 13: GraphGolf プログラム

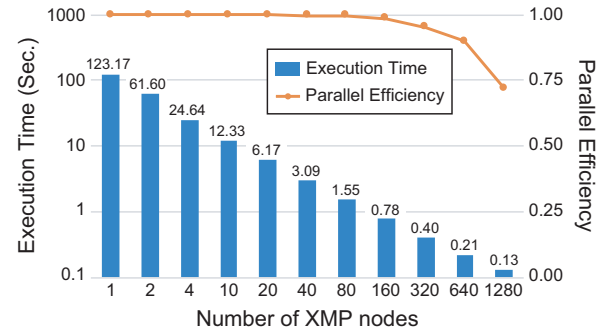


図 14: 性能評価

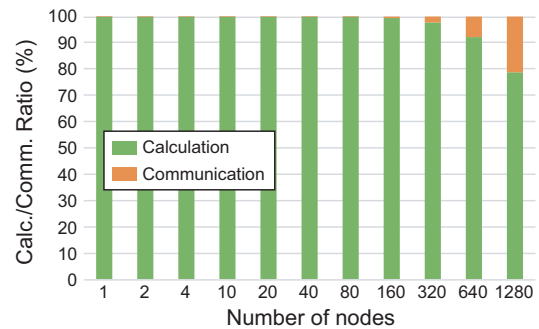


図 15: 計算時間に対する通信割合

## 6. まとめと今後の課題

本稿では、PGAS 言語の 1 つである XMP と C・Fortran・Python との連携を行うための機能の実装を Omni Compiler に対して行った。実装した機能の 1 つである Python との連携機能を用いて、Graph Order/degree 問題に対するアプリケーションの開発を行った。Graph Order/degree 問題の中で最も計算コストが大きい全頂点間最短経路探索に要する時間を計測した結果、オリジナルの Python のプログラムと比較して、我々が作成した Python と XMP で作成したプログラムは、1CPU コアを用いた場合に 21% の性能向上を達成した。また、1280 CPU コアを用いた場合は、1CPU コアを用いた場合と比較して 921 倍の高速化を達成した。

今後の課題として、並列化効率の改善が挙げられる。現在の実装では、flat-MPI のように、各 CPU コアに対して 1 ノードを割り当てている。そこで、頂点毎の幅優先探索（図 13b の 10～12 行目）をスレッド並列化することにより、通信時間の削減とノードに対するタスクの割当の均一化を達成できると考えている。

## Acknowledgements

This research used the coma system provided by Interdisciplinary Computational Science Program in the Center for Computational Sciences, University of Tsukuba. The work was supported by the Japan Science and Technology Agency, Core Research for Evolutional Science and

Technology program entitled “Research and Development on Unified Environment of Accelerated Computing and Interconnection for Post-Petascale Era” in the research area of “Development of System Software Technologies for Post-Peta Scale High Performance Computing.”

## 参考文献

- [1] F. Cantonnet and Y. Yao and M. Zahran and T. El-Ghazawi. Productivity analysis of the UPC language. In *18th International Parallel and Distributed Processing Symposium, 2004. Proceedings.*, pp. 254–260, April 2004.
- [2] Katherine Yelick et. al. Productivity and Performance Using Partitioned Global Address Space Languages. PASC0 '07, Proceedings of the 2007 international workshop on Parallel symbolic computation, 2007.
- [3] Andrew I. Stone et al. Evaluating coarray fortran with the cgpops miniapp. In *Proceedings of the Fifth Conference on Partitioned Global Address Space Programming Models (PGAS)*, October 2011.
- [4] Masahiro Nakao and et al. Implementation and evaluation of the HPC challenge benchmark in the XcalableMP PGAS language. *The International Journal of High Performance Computing Applications*, pp. 1–14, 2017.
- [5] Jose Jithin and et al. Unifying UPC and MPI Runtimes: Experience with MVAPICH. In *Proceedings of the Fourth Conference on Partitioned Global Address Space Programming Model*, PGAS '10, pp. 5:1–5:10, New York, NY, USA, 2010. ACM.
- [6] Masahiro Nakao et al. Productivity and Performance of the HPC Challenge Benchmarks with the XcalableMP PGAS Language. In *7th International Conference on PGAS Programming Model*, pp. 157–171, 2013.
- [7] Masahiro Nakao et al. Productivity and Performance of Global-View Programming with XcalableMP PGAS Language. In *Proceedings of the 2012 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID '12*, pp. 402–409, 2012.
- [8] XcalableMP Specification. <http://xcalablemp.org/specification>.
- [9] Masahiro Nakao et al. XcalableACC: Extension of XcalableMP PGAS Language Using OpenACC for Accelerator Clusters. In *Proceedings of the First Workshop on Accelerator Programming Using Directives, WACCPD '14*, pp. 27–36, 2014.
- [10] XcalableACC Specification. <http://xcalablemp.org/XACC.html>.
- [11] Masahiro Nakao and et al. Implementing Lattice QCD Application with XcalableACC Language on Accelerated Cluster. In *2017 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 429–438, Sept 2017.
- [12] Numrich Robert W. and Reid John. Co-array Fortran for parallel programming. *SIGPLAN Fortran Forum*, Vol. 17, No. 2, pp. 1–31, August 1998.
- [13] Nowicki M. and Gorski L. and Grabarczyk P. and Bala P. PCJ - Java library for high performance computing in PGAS model. In *High Performance Computing Simulation (HPCS), 2014 International Conference on*, pp. 202–209, July 2014.
- [14] A publication of the UPC Consortium, November 2013. <https://upc-lang.org/assets/Uploads/spec/upc-lang-spec-1.3.pdf>.
- [15] Yili Zheng and Kamil, A. and Driscoll, M.B. and Hongzhang Shan and Yelick, K. UPC++: A PGAS Extension for C++. In *Parallel and Distributed Processing Symposium, 2014 IEEE 28th International*, pp. 1105–1114, May 2014.
- [16] Kumar Vivek and et al. HabaneroUPC++: A Compiler-free PGAS Library. In *Proceedings of the 8th International Conference on Partitioned Global Address Space Programming Models*, PGAS '14, pp. 5:1–5:10, 2014.
- [17] Charles Philippe and Grothoff Christian and Saraswat Vijay and Donawa Christopher and Kielstra Allan and Ebcioğlu Kemal and von Praun Christoph and Sarkar Vivek. X10: An Object-oriented Approach to Non-uniform Cluster Computing. *SIGPLAN Not.*, Vol. 40, No. 10, pp. 519–538, October 2005.
- [18] Chamberlain B.L. and Callahan D. and Zima H.P. Parallel Programmability and the Chapel Language. *Int. J. High Perform. Comput. Appl.*, Vol. 21, No. 3, pp. 291–312, August 2007.
- [19] Karl Förlinger and et al. DASH: A C++ PGAS library for distributed data structures and parallel algorithms. *CoRR*, Vol. abs/1610.01482, , 2016.
- [20] Omni Compiler. <http://omni-compiler.org>.
- [21] PC Cluster Consortium. <http://www.pccluster.org/en/>.
- [22] Charles H. Koelbel et al. *The High Performance Fortran Handbook*. MIT Press, 1994.
- [23] Kennedy Ken and Koelbel Charles and Zima Hans. The rise and fall of High Performance Fortran: an historical object lesson. In *Proceedings of the third ACM SIGPLAN conference on History of programming languages*, HOPL III, pp. 7–17–22, 2007.
- [24] Hiroyuki Miyazaki, Yoshihiro Kusano, Naoki Shinjou, Fumiyoshi Shoji, Mitsuo Yokokawa, Tadashi Watanabe. Overview of the K computer. *FUJITSU SCIENTIFIC and TECHNICAL JOURNAL*, Vol. 48, No. 3, pp. 255–265, 2012.
- [25] Dan Bonachea. GASNet Specification. Technical Report CSD-02-1207, University of California, Berkeley, October 2002.
- [26] 藤原一毅, 藤田聡, 中野浩嗣, 井上武, 鯉淵道紘. みんなで order/degree 問題を解いて究極の低遅延相互結合網をつくろう. 電子情報通信学会技術研究報告 信学技報, Vol. 115, No. 174, pp. 223–228, aug 2015.
- [27] V. G. Cerf, D. D. Cowan, R. C. Mullin, and R. G. Stanton. A lower bound on the average shortest path length in regular graphs. *Networks*, Vol. 4, No. 4, pp. 335–342, 1974.
- [28] Graph Golf: The Order/degree Problem Competition. <http://research.nii.ac.jp/graphgolf>.
- [29] NetworkX. <https://networkx.github.io>.
- [30] Nicholas Metropolis, Arianna W. Rosenbluth, Marshall N. Rosenbluth, Augusta H. Teller, and Edward Teller. Equation of state calculations by fast computing machines. *The Journal of Chemical Physics*, Vol. 21, No. 6, pp. 1087–1092, 1953.
- [31] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, Vol. 220, No. 4598, pp. 671–680, 1983.
- [32] Beamer et al. Direction-optimizing breadth-first search. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pp. 12:1–12:10, Los Alamitos, CA, USA, 2012. IEEE Computer Society Press.