Vol.2017-HPC-160 No.23
2017/7/27

Omniコンパイラ基盤における HPC向けメタプログラミング機能の構想

村井 均[1]　佐藤 三久[1]　李 珍泌[1]　中尾 昌広[1]　岩下英俊[1]

概要：Omni は，Fortran および C をターゲットとする，ソース to ソース変換に基づくコンパイラ基盤である．Omni は，理研と筑波大によって開発されており，Omni XcalableMP コンパイラの基盤としても用いられている．現在，我々は，主にプログラム生産性の向上を目的として，Omni における HPC 向けメタプログラミング機能の設計を進めている．本報告では，本機能の設計および予備的実装を示すとともに，本機能の実現可能性および有効性を検証する．ループ・アンローリングおよび構造型のデータレイアウト最適化に関する適用例を用いた評価の結果，本機能により種々の変換を記述することが可能であることを確認できた．

1. はじめに

従来，ハイパフォーマンスコンピューティング（HPC）の分野では，プログラミングにおける高生産性よりも高性能の方が重視される場合が多かったと言える．そのため，ターゲットとする計算機環境に応じて，アプリケーション・プログラムに（HPC 向けの）様々な高度なチューニングまたは最適化を施すことが必要不可欠であった．

本来，このようなチューニング/最適化は，コンパイラによって自動的に適用されることが理想ではあるが，以下の 2 つの理由により，プログラマが自身でコードを修正しなければならないことは多い．

- 解析能力の限界から，対象となるコード変形の安全性および有効性をコンパイラが正しく判断できるとは限らない．
- 望む最適化を任意のコンパイラがサポートしているとは限らない．

1 点目についてはコンパイラに対する指示文またはプラグマを使用することによって解決可能だが，2 点目は依然として問題となる．結果として，ユーザは，自身でアプリケーション・プログラムのソースコードにチューニングを施すことを強いられてきた．

近年，HPC の分野で用いられる計算機環境がますます多様化するとともに，ユーザがチューニングを施したソースコードも多様化・細分化の一途を辿ることは不可避であり，この分野におけるプログラム生産性の低下が重大な問題となりつつある現在，高性能と高生産性を両立するプログラミング環境の研究開発は急務である．我々は，メタプログラミングの技法によってこの問題を解決できると考えている．

プログラムを処理対象とするプログラムを，**メタプログラム**と定義する [*1]．ここで，処理対象のプログラム（ターゲットプログラム）は自分自身であってもよい．メタプログラムを利用するプログラミング技法を，**メタプログラミング**と呼ぶ．一般に，メタプログラミングを活用することにより得られる恩恵は，次の 2 点である．

- コンパイル時評価による性能向上
- 定形コード（いわゆるボイラープレートコード）の抽象化（共通化）による生産性向上

以上に加え，HPC 向けの高度な最適化をメタプログラミングの技法で記述することにより，複雑なコード変形をユーザプログラムから切り離すことが可能となり，高性能と高生産性を両立する理想のプログラミング環境を実現することができる．

一方，我々は，Fortran および C をターゲットとする，ソース to ソース変換に基づくコンパイラ基盤 Omni[1] の開発を行ってきた．Omni は，Omni XcalableMP[2], [3] や Omni XcalableACC[4] の基盤としても用いられている．我々は，現在，Omni における HPC 向けメタプログラミング機能の設計を進めている．本報告は，本機能の概要を示すとともに，ループ・アンローリングおよび構造型のデータレイアウト最適化に対する適用例によってその実現可能性と有効性を検証することを目的とする．

[1] 国立研究開発法人理化学研究所 計算科学研究機構

[*1] この定義に従うなら，コンパイラは，一種の（典型的な）メタプログラムであると言える

以下，2 章で関連研究について述べ，3 章で Omni コンパイラ基盤を概観する．続いて，4 章でメタプログラミング機能の設計，5 章でその実装方式を説明した後，6 章でループ・アンローリングおよび構造型のデータレイアウト最適化に対する適用例を示す．最後に 7 章で本報告を総括する．

2. 関連研究

メタプログラミングを実現するための技術として，以下のようなものが挙げられる．

- マクロ
  C プリプロセッサ（cpp）や m4 に代表される，主に文字列置換に基づくプログラム変換の機能．
- テンプレート
  C++を始めとする言語が備える総称的プログラミングのための機能であり，「テンプレート」と呼ばれる雛形コードを，コンパイル時に置換（実体化）する処理に基づく．上記のマクロを発展させたものだが，プリプロセッサではなくコンパイラによって処理されるため，データの「型」を明示的に扱うことができる．
- メタクラス
  Python や Ruby などの言語が備える，クラスそのものの挙動を定義するための機能．

特に，テンプレート・メタプログラミングの一種である式テンプレートが，作業配列の削減などの目的で HPC アプリケーションでも利用される [5], [6].

また，HPC アプリケーションのための，種々の埋め込みドメイン専用言語（Domain-Specific Language: DSL）が開発されており，これら埋め込み DSL の実現にはメタプログラミングの技法が用いられる場合が多い．HPC の分野では，例えば，Coarray C++[7] や DASH[8] は，テンプレート・メタプログラミングに基づく埋め込み DSL として実現されている．

一方，HPC の分野で広く用いられている Fortran は，C++におけるテンプレートや Python におけるメタクラスのような，メタプログラミングを実現するための組込みの機能を持たない．これに対し，Yue らは，Fortran においてメタプログラミングを実現する仕組みとして，OpenFortran[9] および SPOT[10] を提案している．OpenFortran は，Fortran ソースコードに対するコード変換器（トランスレータ）として機能する．OpenFortran では，ターゲットのソースコードがオブジェクトとして表現されるとともに，それらに対する操作を定義するメタオブジェクト・プロトコル（Metaobject Protocol: MOP）が提供される．ユーザは，この MOP に基づくオブジェクト操作を記述することにより，コード変換を行う．SPOT は，OpenFortran における変換規則を定義するための DSL であり，MOP に基づきオブジェクトを直接に操作するのに比べ，ユーザは，より簡単にコード変換の動作を記述できる．

Xevolver[11] は，本研究と同じく，アプリケーションコードと，それに対する最適化を切り分けて管理するためのプログラミングフレームワークである．最適化の手順を，既存のアプリケーションから抽出された「再利用可能な処理とプログラム中の記述」のカタログから得る点が，本研究とは異なる．山田らは，本報告でも適用事例として取り上げる構造型のデータレイアウト最適化に対し Xevolver を適用している [12].

3. Omni コンパイラ基盤

Omni[1] は，Fortran および C をターゲットとする，ソース to ソース変換に基づくコンパイラ基盤であり，Omni XcalableMP[2], [3]，Omni XcalableACC[4]，CLAW[13]，Xevolver[11] の基盤として用いられている．

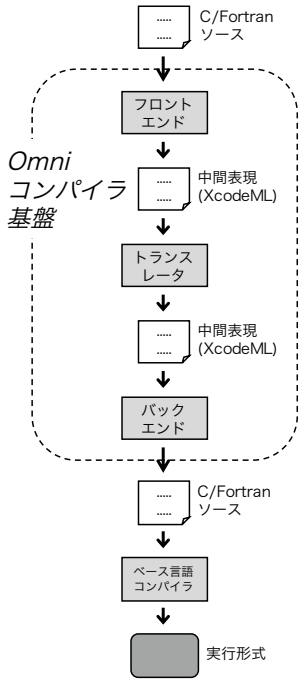


図 1 Omni コンパイラ基盤の構成

図 1 に示す通り，Omni は，フロントエンド，トランスレータ，バックエンドの 3 つのコンポーネントから成り，それらはそれぞれ次のような機能を持つ．

- フロントエンド
  C または Fortran[*2] のソースコードをパースし，XcodeML/C 形式 [14] または XcodeML/Fortran 形式 [15] の中間表現へ変換する．
- トランスレータ
  XcodeML 形式を Xobject 形式へ変換した後，各種の変換（e.g. XcalableMP における分散メモリ向け並列

[*2] 現在，C++のフロントエンドを開発中である

化，XcalableACC におけるアクセラレータ向けオフロード処理）を適用する．

- バックエンド（デコンパイラ）
  XcodeML 形式を C または Fortran のソースプログラムへ変換する．

ここで，Omni における抽象構文木（abstract syntax tree: AST）の XML 表現が XcodeML（**図 2**）であり，Java オブジェクト表現が Xobject である．

```
1 <FassignStatement lineno="2" file="foo.f90">
2   <Var type="Freal" scope="local">a</Var>
3   <plusExpr type="Freal">
4     <Var type="Freal" scope="local">b</Var>
5     <Var type="Freal" scope="local">c</Var>
6   </plusExpr>
7 </FassignStatement>
```

図 2 XcodeML/Fortran の例（a = b + c）

バックエンドが出力したソースプログラムは，ターゲット環境に応じたベース言語コンパイラによりコンパイルされ，必要に応じて各種のランタイムとリンクされて，実行形式を得る．現在のところ，トランスレータは並列化（XMP の場合）やオフロード処理（XACC の場合）に必要な変換のみを担い，各種の標準的な最適化（e.g. レジスタ割付け，SIMD 化）はベース言語コンパイラに委ねられる．

4. 基本設計

1 章で述べた目的を達成するために，本研究において開発するメタプログラミング機能が満たすべき要件は以下の通りである（**図 3**）．

- ユーザが開発するユーザプログラムは，アプリケーションプログラムと **Omni メタプログラム**を含む．
- アプリケーションプログラムは，適用すべき変換の種類や対象を指示する **Omni メタプログラミング指示文**を含む．
- Omni メタプログラムは，アプリケーションプログラムに対する操作を定義する．
- Omni は，アプリケーションプログラムおよび Omni メタプログラムを入力とし，指定された操作を施されたアプリケーションプログラムを出力とする（コンパイル時メタプログラミング）．

ここで，Omni メタプログラムは，ソースコードレベルで任意の変形を定義できなければならない．すなわち，本機能は，任意の効果を持つ指示文を定義できる機能であると言え換えられる．

以下，Omni メタプログラミング指示文および Omni メタプログラムについて，順に説明する．

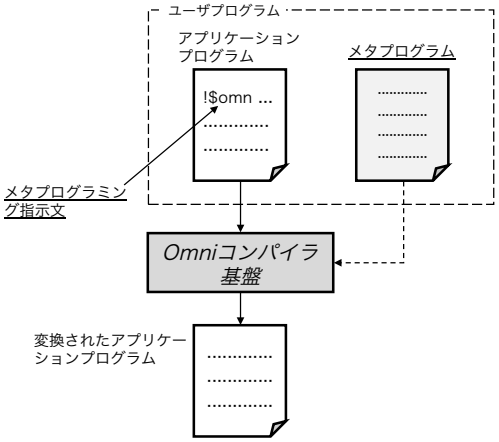


図 3 Omni メタプログラミング機能の基本設計

4.1 Omni メタプログラミング指示文

Omni メタプログラミング指示文として，次の 2 種類を提供する．

- 実行指示文（omn 指示文）
  プレフィクス omn から始まり，実行部に置かれる．直後の構文（e.g. DO ループ）に効果を持つ（変換の対象とする）．
- 宣言指示文（omd 指示文）
  プレフィクス omd から始まり，宣言部に置かれる．有効域全体に効果を持つ（変換の対象とする）．

図 4 に，Omni メタプログラミング指示文の文法を示す．*omn_name* および *omd_name* は，指示文の名前を示す文字列であり，*arg* はターゲット言語における任意の式である．

```
1 !$omn omn_name ( arg , ... )
```

(a) omn 指示文

```
1 !$omd omd_name ( arg , ... )
```

(b) omd 指示文

図 4 Omni メタプログラミング指示文の文法

4.2 Omni メタプログラム

Omni メタプログラムは，**Omni メタ言語**によって記述される．

前述した Omni メタプログラムの要件を満たすため，Omni メタ言語は，ソースコードレベルの任意の変形を表現できるものでなければならない．すなわち，Omni メタ言語は，ターゲットプログラムの AST に相当するデータ構造と，それに対する操作を扱う．

上記の要件を満たす Omni メタ言語の候補として，以下の 3 種類が考えられる．

(a) Xobject ベース

Omni トランスレータが備えるコード変換機能（Xobject とそれを操作するための各種メソッド）を利用する方法．Omni コンパイラ基盤が既に備える柔軟かつ高度な操作が可能だが，ユーザは Omni の利用法に習熟する必要がある．

(b) XML ベース
XSLT などの一般的な XML 変換言語により，XcodeML を XML として変換する方法．Omni トランスレータが保持する解析情報（e.g. 変数の有効範囲（スコープ）に関する情報）を利用できないため，別途の解析が必要になる．

(c) DSL
Omni メタプログラムの記述に特化した DSL（およびその専用処理系）を用いる方法（cf. 関連研究で述べた SPOT）．

これら 3 種類の Omni メタ言語のうち，上に挙げたものほど，処理系実装の手間は小さいが，ユーザが払う労力は大きくなると考えられる．

5. プロトタイプ実装

4 章で示した設計に対し，**図 5** に示す構成により実装を行う．すなわち，**Omni メタトランスレータ**は，Omni メタプログラミング指示文と Omni メタプログラムの記述に従い，アプリケーションプログラムに変換を加える．

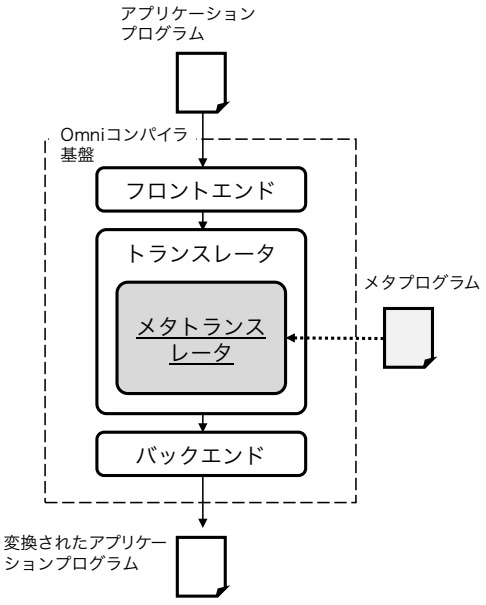


図 5 Omni メタプログラミング機能の構成

4.2 節で述べた Omni メタ言語の種類によって，Omni メタトランスレータの実体は次のように実現される．

(a) Xobject ベース → Omni トランスレータまたはその拡張
(b) XML ベース → XML 変換言語処理系（e.g. XSLT 処理系）
(c) DSL → DSL 処理系

本報告では，プロトタイプとして，Fortran をターゲット言語として，「(a) Object ベース」のメタ言語を処理するメタトランスレータを実装する．

図 6 はプロトタイプの構成を示す．

以下，5.1 節の「Omni メタプログラムの作成」は，本機能を利用するユーザとしての作業であり，5.2 節の「Omni メタトランスレータの実装」は本機能の開発者としての作業であることに注意されたい．

5.1 Omni メタプログラムの作成

ユーザは，Omni が提供するインタフェース `METAXexec`（`omn` 指示文に対応する場合）または `METAXdecl`（`omn` 指示文に対応する場合）を実装するクラス（メタプログラム・クラスと呼ぶ）として Omni メタプログラムを作成する．そのクラス名は，対応する Omni メタプログラミング指示文の名前に一致させる [*3]．さらに，Omni トランスレータが備えるコード変換機能に基づいて，`run` メソッドおよび `rundecl` メソッド（`omd` 指示文に対応する場合）を実装する．これらのメソッドの詳細については，次節で述べる．

5.2 Omni メタトランスレータの実装

既存の Omni トランスレータ（およびフロントエンド）を拡張することにより，以下のように Omni メタトランスレータの機能を実現する．

- フロントエンドは，ターゲットプログラムに含まれる Omni メタプログラミング指示文をパースし，OMDPragma 型（宣言指示文）または OMNPragma 型（実行指示文）の XcodeML 要素を生成する．
- 上述の XcodeML 要素に対し，Omni トランスレータは，METAXblock 型オブジェクトを生成する．その際，Java のリフレクションの機能によって，当該指示文の名前に等しい名前のクラス（前節で述べたメタプログラム・クラス）をインスタンス化する．
- Omni トランスレータは，各 METAXblock 型オブジェクトに対し次の処理を行う．
  - 実行指示文
    後続する Block オブジェクトを引数として，メタプログラム・クラスの `run` メソッドを呼び出す．
  - 宣言指示文
    当該指示文が含まれるプログラム単位に相当する Block オブジェクトを引数として，メタプログラム・クラスの `rundecl` メソッドおよび `run` メソッドを呼び出す．前者は各宣言文に対する変換を実施し，後者は各実行文に対する変換を実施する．さらに，当

[*3] したがって，現在の実装では，`omn` 指示文および `omd` 指示文の名前は，Java のクラス名として許される文字列に限られる．

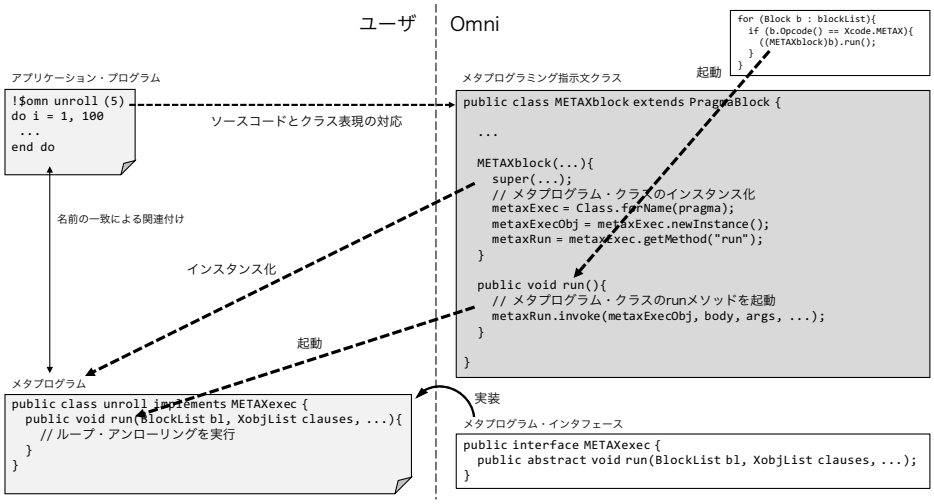


図 6 プロトタイプ実装の構成

該指示文がモジュール中に含まれていた場合，そのモジュールを参照する他の全てのプログラム単位に対しても，run メソッドを呼び出す．

6. ケーススタディ

6.1 ループ・アンローリング

ループ・アンローリングは，よく知られたループ最適化手法であり，ループ処理自体のオーバヘッド（e.g. ループカウンタのインクリメント，終了条件の判定）の削減や，ループボディに含まれる演算の数が増えることによってコンパイラによる命令スケジューリングの自由度を増やせる，といった利点が得られる．n 段のループ・アンローリングは，次の手順から成る．

- ループの繰り返し空間の変形
  対象のループの制御変数の繰返し範囲に，ストライド n を設定する．
- ループボディの複製
  ループボディを n 回だけ複製する．
- 剰余ループの生成
  対象のループの繰り返し回数が，n で割り切れない場合，剰余に当たるループを生成する．

本機能の利用に際して，ユーザが行うべき作業は次の 2 つである．

- Omni メタプログラミング実行指示文の設計
  ここでは，次のような unroll 指示文を用いることとする．

```
1 !$omn unroll ( n )
```

  ここで，引数 n はループ・アンローリングの段数である．
- Omni メタプログラムの作成
  上述したループ・アンローリングの手順を実現する Omni メタプログラムとして，unroll クラスを作成する．

作成した unroll クラスのコード（抜粋）を，付録 A.1 に掲載する．

5 章で実装した機能を用いて，**図 7**(a) のループに 5 段ループ・アンローリングを適用した結果，同図 (b) のループが得られた．

```
1 !$omn unroll (5)
2 do i = 1, n
3    a(i) = ...
4 end do
```

(a) アンロール前

```
1  DO i = 1 , n , 5
2   a ( i ) = ...
3   a ( i + 1 ) = ...
4   a ( i + 2 ) = ...
5   a ( i + 3 ) = ...
6   a ( i + 4 ) = ...
7  END DO
8  DO i = 1 + 5 * ( ( n - 1 + 1 ) / 5 ) , n , 1
9   a ( i ) = ...
10 END DO
```

(b) アンロール後

図 7 5 段ループ・アンローリングの例

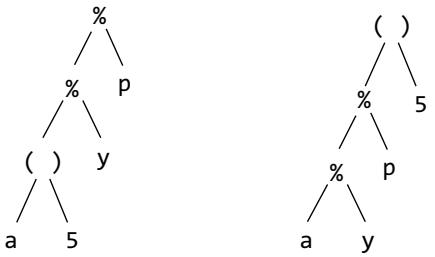
6.2 構造型のデータレイアウト最適化

本最適化は，Array-of-Structure（AOS）型の配列を，Structure-of-Array （SOA）型の変数に置き換えることにより，データのメモリレイアウトを変更する変換である．この変換により，多くの場合，メモリアクセスの連続性は

向上し，実行速度の向上が見込める．

本最適化は，次の手順から成る．

- 構造型宣言文の変形
  対象の AOS 構造型を，SOA 構造型に置き換える．
- 構造型配列の出現の変形
  実行部における対象の AOS 構造型配列の出現を，等価な SOA 構造型変数に置き換える．例えば，**図 8**(a) の AST で表現される AOS 構造型配列 `a(5)%y%p` は，同図 (b) の通り SOA 構造型変数 `a%y%p(5)` に置き換えられる．



(a) AOS 形式　　(b) SOA 形式

図 8 AOS 形式および SOA 形式の AST 表現

本機能の利用に際して，ユーザが行うべき作業は次の 2 つである．

- Omni メタプログラミング宣言指示文の設計
  ここでは，次のような `aos_to_soa` 指示文を用いることとする．

```
!$omd aos_to_soa ( a, ... )
```

  ここで，引数 `a`，`...` は，対象の構造型配列の名前の列である．
- Omni メタプログラムの実装
  上述した最適化の手順を実現する Omni メタプログラムとして，`aos_to_soa` クラスを作成する．

5 章で実装した機能を用いて，**図 9**(a) の構造型配列 `a` にデータレイアウト最適化を適用した結果，同図 (b) が得られた．

7. おわりに

本報告では，Omni コンパイラ基盤における HPC 向けメタプログラミング機能の概要を示すとともに，ループ・アンローリングおよび構造型に対するデータレイアウト最適化に関する適用例を用いて本機能の実現可能性および有用性を予備的に評価した．

その結果，メタプログラミング指示文を含むアプリケーションプログラムおよびメタプログラムを，メタトランスレータが処理するという実装により，種々の変換を適用することが可能であることを確認した．本機能は，高性能と

```
1  type t0
2     integer p
3  end type t0
4  type t1
5     real x
6     type(t0) y
7  end type t1
8  type(t1) a(100)
9  !$omd aos_to_soa (a)
10 a(5)%y%p = 0.
```

(a) 変換前

```
1  TYPE :: t0
2   INTEGER :: p ( 1 : 100 )
3  END TYPE t0
4
5  TYPE :: t1
6   REAL :: x ( 1 : 100 )
7   TYPE ( t0 ) :: y
8  END TYPE t1
9
10 TYPE ( t1 ) :: a
11
12 a % y % p ( 5 ) = 0.
```

(b) 変換後

図 9 構造型のデータレイアウト最適化の例

高生産性を両立するプログラミング環境の実現に際し，重要な機能の一つになると考えられ，現在，仕様の策定が進められている XcalableMP 次期仕様（XMP2.0）の新機能の一つとして提案することも検討している．

一方，本報告における予備実装で用いた，Xobject ベースのメタ言語によってメタプログラムを記述する方式は，ユーザが Omni の利用法に習熟していることを必要とするため現実的ではない．実際には，Xobject を操作するための，より明確で扱いやすい API（i.e. メタオブジェクト・プロトコル）を整備・提供することが望ましい．あるいは，DSL ベースのメタ言語および DSL 処理系としてのメタトランスレータを提供するという方法も考えられる．これらの設計および実装は今後の課題である．

また，本報告では，メタプログラミングの技法を用いて，ターゲットプログラム中のコードを「変換する」ことを扱ったが，新たに生成したコードをターゲットプログラム中の任意の箇所に「埋め込む」という方式も考えられる．この方式の検討も今後の課題である．

参考文献

[1] Omni Compiler Project: Omni Compiler Project, `http://omni-compiler.org/`.
[2] XcalableMP Specification Working Group: XcalableMP

Specification Version 1.2.1, http://www.xcalablemp.org/download/spec/xmp-spec-1.2.1.pdf (2014).
[3] Omni Compiler Project: Omni XcalableMP Compiler, http://omni-compiler.org/xcalablemp.html.
[4] Nakao, M., Murai, H., Shimosaka, T., Tabuchi, A., Hanawa, T., Kodama, Y., Boku, T. and Sato, M.: XcalableACC: Extension of XcalableMP PGAS language using OpenACC for accelerator clusters, *Accelerator Programming using Directives (WACCPD), 2014 First Workshop on*, IEEE, pp. 27–36 (2014).
[5] Iglberger, K., Hager, G., Treibig, J. and Rde, U.: High performance smart expression template math libraries., *HPCS* (Smari, W. W. and Zeljkovic, V., eds.), IEEE, pp. 367–373 (2012).
[6] Iglberger, K., Hager, G., Treibig, J. and Rüde, U.: Expression Templates Revisited: A Performance Analysis of the Current ET Methodology, *CoRR*, Vol. abs/1104.1729 (online), available from ⟨http://arxiv.org/abs/1104.1729⟩ (2011).
[7] Johnson, T. A.: Coarray C++, *Proc. 7th Int'l Conf. on Partitioned Global Address Space Programming Models (PGAS2013)* (2013).
[8] Fürlinger, K., Fuchs, T. and Kowalewski, R.: DASH: A C++ PGAS Library for Distributed Data Structures and Parallel Algorithms, *Proceedings of the 18th IEEE International Conference on High Performance Computing and Communications (HPCC 2016)*, Sydney, Australia, pp. 983–990 (online), DOI: 10.1109/HPCC-SmartCity-DSS.2016.0140 (2016).
[9] Yue, S. and Gray, J.: OpenFortran: Extending Fortran with Meta-programming, *Proc. Supercomputing 2013* (2013).
[10] Yue, S. and Gray, J.: SPOT: A DSL for Extending Fortran Programs with Metaprogramming, *Adv. Soft. Eng.*, Vol. 2014, pp. 9:9–9:9 (online), DOI: 10.1155/2014/917327 (2014).
[11] Takizawa, H., Hirasawa, S., Hayashi, Y., Egawa, R. and Kobayashi, H.: Xevolver: An XML-based Code Translation Framework for Supporting HPC Application Migration, *Proc. IEEE Int'l Conf. High Performance Computing (HiPC)*, pp. 1–11 (2014).
[12] 山田剛史，平澤将一，須田礼仁，滝沢寛之：データレイアウト最適化のためのコード変換規則の自動生成，情報処理学会研究報告，Vol. 2017–HPC–158, No. 28, pp. 1 – 8 (2017).
[13] C2SM (Center for Climate Systems Modeling), ETH Zurich: CLAW FORTRAN Compiler, https://github.com/C2SM-RCM/claw-compiler.
[14] XcalableMP/Omni Compiler Project: XcodeML/C 仕様書 Version 0.9J, http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/xcodeml/XcodeML-C-0.9J.pdf (2009).
[15] XcalableMP/Omni Compiler Project: XcodeML/Fortran 仕様書 Version 0.9J, http://www.hpcs.cs.tsukuba.ac.jp/omni-openmp/xcodeml/XcodeML-Fortran-0.9J.pdf (2009).

付　　録

A.1 unroll クラスのコード（抜粋）

```
1  public class unroll implements METAXexec {
2
3    public void run(BlockList body, XobjList clauses, METAXblock metaxBlock){
4
5      Xobject factor = clauses.getArg(0);
6
7      ...
8
9      FdoBlock do_block = (FdoBlock)body.getHead();
10     FdoBlock reminder_do_block = do_block.copy();
11
12     //
13     // ループの繰り返し空間の変形
14     //
15
16     do_block.setStep(factor);
17
18     //
19     // ループボディの複製
20     //
21
22     ...
23
24     // 段数分だけ複製する
25     for (int i = 0; i < factor.getInt(); i++){
26       ...
27       // ループ制御変数iの出現を，i , i+1, i+2, ...で置き換える.
28       for (j.init(); !j.end(); j.next()) {
29         Xobject x = j.getXobject();
30         if (x != null && x.Opcode() == Xcode.VAR &&
31             x.getSym().equals(ctl_var)){
32           j.setXobject(Xcons.plusOp(x, Xcons.IntConstant(i)));
33         }
34       }
35       newBody.add(Bcons.buildBlock(newObj));
36     }
37
38     do_block.setBody(newBody);
39
40     //
41     // 剰余ループの生成
42     //
43
44     Xobject reminder_lb = ...
45     reminder_do_block.setLowerBound(reminder_lb);
46     body.add(reminder_do_block);
47
48   }
49 }
```