プログラミング 8－14
（1996. 8. 28）

メタレベル機能による並列プログラミング

高橋俊行[†] 石川 裕[††]
佐藤三久[††] 米澤明憲[†]

MPC++ はC++にメタレベル機能を導入した言語である。MPC++ はメタレベル機能を用い言語機能の追加、変更が可能である。本稿では、代表的なデータ並列構文の一つであるforall構文を例にMPC++のメタレベル機能を用いた構文木の変換の概要を示す。また、メタプログラムの記述においてテンプレートの使用がコードの再利用性を高める例を示す。

Parallel Programming using Meta-level Architecture

Toshiyuki TAKAHASHI,[†] Yutaka ISHIKAWA,[††]
Mitsuhisa SATO[††] and Akinori YONEZAWA[†]

MPC++ is an extended C++ that has a meta-level architecture. The architecture makes it possible to change and extend its language features. Using this architecture, we are implementing the 'forall' syntax for an data-parallel extension. In this paper, we give an overview of the implementation and an example of meta-level description using C++ template. The example shows template programming improves reusability of meta-level code.

1. はじめに

並列計算機環境のためのプログラミング言語では、並列化のための構文など、言語の構成要素の実装を環境やアプリケーションに応じてカスタマイズしたい場合が往々にしてある。データ並列構文を例にすると、分散共有メモリ環境での実装は、メモリのコンシステンシをとるためのバリア同期コードの挿入、メッセージパッシング型分散メモリ環境では、計算に必要なデータの通信コードの挿入について考慮しなくてはならない。また、通信の戦略はアプリケーションごとに変更が可能であることが望ましい。

このように言語機能の実装が、環境によって数多く考えられる場合、すべての場合を網羅している言語処理系を最初から提供することは非常に困難である。また、環境の差異をクラスライブラリによって吸収するアプローチも考えられるが、クラスライブラリにはコンパイル時の大域的な解析を記述することはできない。ゆえにこのアプローチは最適化のためにこのような解析を必要とするデータ並列機能などの実装には不向きである。

環境やアプリケーションプログラム毎に最適化されたコードを生成するようなプログラムを、メタレベル機能によって提供し、ライブラリ化する方法は有望である[4),6)]。メタレベル機能は、ターゲットプログラムが「どのように実行されるか」を、プログラマに操作させることを可能にしており、既存の言語機能を変更したり、新しい言語機能を定義することを可能にする。この操作が、メタプログラムとして、ターゲットプログラムから分離して記述できるため、言語機能をモジュール化することができる。言語機能のモジュール化については以下の利点が挙げられる。

- 言語機能の追加が可能になる。これにより、並列構文などの言語機能があたかも組み込みの機能であるかのように使える。
- 言語機能が提供する最適化を、アプリケーションプログラムの性質や実行環境に応じて、適切なものを選択することができる。

本稿では、C++[3)]にメタレベル機能を導入した言語MPC++[4)]を用い、メタレベルのプログラミングについて考える。例として分散メモリ環境のためのデータ並列構文 `forall` を実装するメタレベルプログラミングについて述べる。ここでは、単に並列実行するだけではなく、ソースコードの大域的な解析を行い、inspector/executor[5)]アルゴリズムに基づく最適化されたコードを生成する。

2節ではMPC++のメタレベル機能について説明し、3節では実装する我々の `forall` 構文の仕様についてあ

† 東京大学大学院理学系研究科
Faculty of Science, University of Tokyo
†† 新情報処理開発機構
Real World Computing Partnership

らわす。4節では forall 構文の inspector/executor による実行方式について述べ、5節では forall 構文を4節の方式へ変換するメタプログラムについて紹介する。

2. MPC++ のメタレベル機能

2.1 概　　要

MPC++ ではメタレベル機能を用い、ソースプログラム中に「MPC++ がある構文単位をどのように処理するか」を記述することが可能である。この記述は、MPC++ コンパイラの実装の一部とみなすことができる。

ソースプログラム中に記述される MPC++ コンパイラの実装を、MPC++ コンパイラのメタプログラムと呼ぶ。MPC++ ではコンパイル対象となるソースプログラム (ベースプログラム) とメタプログラムを分離するために C++ のシンタックスを拡張した。$meta で始まる宣言はメタレベルの記述である。また、$meta 構文に *compound-statement* を記述した場合、この *compound-statement* は、これを構文解析した直後に評価される。

MPC++ コンパイラはベースプログラムを構文解析し、構文木を構成するが、メタプログラムにはこの構文木を解析したり、別の構文木へ変換したりするプログラムを記述することができる。プログラマは構文木の変換プログラムを適当な場所へ挿入することでコンパイラの実装を変更する。

2.2 拡張構文の導入

MPC++ では *block-statement* の追加を可能にするため、以下に示すような構文規則の雛型を準備している。

```
block-statement :
    _MPCXX_STMT_NEWSTMT0  compound-statement
  | _MPCXX_STMT_NEWSTMT1 (expression)
                          compound-statement
  | _MPCXX_STMT_NEWSTMT2 (expression;
              expression) compound-statement
```

5パラメータ *block-statement* の新しい構文 forall を登録するには、以下のように keybind を用いる。

```
$meta {
    keybind("forall", _MPCXX_STMT_NEWSTMT5,
                              "ForallStmt"); }
```

上記 keybind が評価されると、キーワード forall が字句解析器に登録され、その後のソースコード中において forall 構文が使用可能になる。構文解析によって forall 構文が得られると ForallStmt クラスのインスタンスが生成される。

2.3 構文木の構成

ベースプログラムは MPC++ コンパイラの構文解析によって、各構文要素をノードとした構文木に構成される。このノードを Syntax Tree Object(STO) と呼ぶ。STO は以下に示すクラス SynObj を基底クラスとする派生クラスで構成される (図1)。2.2の ForallStmt も

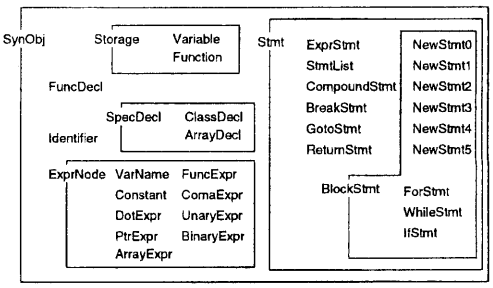

図1　Syntax Tree Objct のクラス階層 (一部)

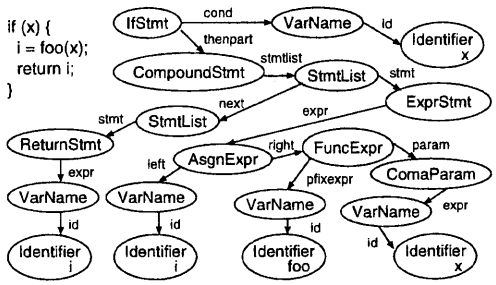

図2　構文解析の例

派生クラスのひとつである。

```
$meta class SynObj {
public:
    Lineno*          lineno;
    SynObj*          delegator;
    Environment*     environment;
    virtual NodeType getType();
    virtual SynObj*  trans0();
    virtual SynObj*  trans1();
};
```

図2には、プログラムの断片と、それを構文解析した結果得られる STO の木を示した。

MPC++ コンパイラは、大域的な宣言を一つ構文解析終了するたびに、構文木構成が終了したばかりの各 STO について trans0 メソッドを呼ぶ。trans0 メソッドは STO を返す。メソッド呼出後、STO は、返された STO と置き換えられる。ここで、trans0 メソッドに構文木の変換を施すコードを記述することにより、構文木の変換が可能になる。

3. 実装するデータ並列構文の仕様

3.1では本稿で使用する分散行列クラスの仕様を示し、3.2では、実装するデータ並列構文 forall の仕様を示す。

3.1 分散行列クラスの仕様

行列は Matrix クラステンプレートで表現することにする。たとえば double 型の要素をもつ 1000 × 1000 行

列 A は次のように宣言する。

```
Matrix<double> A(1000, 1000);
```

コンストラクタへのパラメータは行列のサイズである。

Matrix<double> クラスでは、演算子 () を再定義しており、行列 A の i, j 要素へのアクセスは A(i,j) で与えられる。

分散行列は DistMatrix クラステンプレートで表現する。DistMatrix は前述の Matrix を継承するクラステンプレートである。分散の方式はクラスで指定する。以下の例で行列 B は block 分割で分散、行列 C は cyclic 分割で分散する。分割は行方向に限定する。

```
DistMatrix<double, block> B(1000, 1000);
DistMatrix<double, cyclic> C(1000, 1000);
```

DistMatrix で表される行列はメンバ変数 owner を持つ。owner には分散方式をあらわすクラスのインスタンスが代入されている。owner に行インデックスを与えたとき、「そのインデックスの要素を所有する PE の番号」が返されるものとする。

3.2 forall 構文の仕様

forall はイテレーションを独立に実行するループ構文である☆。本稿で導入する forall のシンタックスを以下に示す。

forall(loopvar; lower; upper;
step; distmatrix) compound-statement;

loopvar はループ制御変数である。これは int 型の変数でなければならない。*lower*, *upper*, *step* はそれぞれ *expression* で、*loopvar* がとる値の下限、上限、増分をあらわす。

forall は全ての PE 上で実行される。イテレーションは各 PE に分散される。イテレーション内部では DistMatrix 型の分散行列を使用する計算を行う。各 PE は行列の分割をそれぞれ所有しているが、計算によってはローカル PE が所有していない行列要素を参照するかもしれない。その際に発生する通信については forall 構文を実装するメタプログラムが通信コードを生成するので forall の使用者は考慮する必要はない。

イテレーションは *distmatrix*.owner(*loopvar*) で表される PE で実行される。*distmatrix* を省略した場合、すべてのイテレーションはローカルの PE で実行される。この場合forall 文は以下の for 文と等価である。

for(loopvar = lower; loopvar <= upper;
loopvar += step) compound-statement;

forall の使用例として緩和計算の一部分を示す。

```
DistMatrix<double, block> A(N,N);
forall(i; 0; upper-1; 1; A) {
  forall(j; 1-(i%2); upper; 2;) {
    A(i,j) = (omega/4.0*(A(i-1,j)+A(i,j-1)
     +A(i+1,j)+A(i,j+1)))+((1.0-omega)*A(i,j));
  }
}
```

☆ HPF における INDEPENDENT ループに相当する

4. forall 構文の実行方式

分散メモリ環境においては、データ並列処理の際に発生する通信のレイテンシ隠蔽が重要である。このため、データ並列構文の実装は対象となる計算について解析を行い、通信量の削減／通信フラグメンテーションの解消／通信と計算のオーバラップなどについて検討を行う必要がある。

forall 構文ではイテレーション間に依存がないため、イテレーションの実行を開始する前に、イテレーション内で必要となる「他の PE が所有するデータの参照」を、あらかじめまとめて解決しておくことが可能である。これにより通信のフラグメンテーションを解消することができる。また、あらかじめイテレーションを、実行において通信を必要とするイテレーションと必要としないイテレーションに分割すると、通信処理を行っている最中に通信を必要としないイテレーションを実行させることが可能になる。これにより通信と計算がオーバラップする。

本稿で紹介する forall 構文の実装は、上記の手法にもとづき、通信レイテンシを隠蔽した、効率のよいコードを展開する。

ループ制御変数 *loopvar* がとりうる値の集合をイテレーション集合 *Iter* とする。このとき、*Iter*(*pe*) を「*pe* であらわされる PE で実行されるイテレーション集合」とする。*Iter*(*pe*) は *Iter* の分割である。*Iter*(*pe*) をさらに「通信が不必要なイテレーション集合」および「通信が必要なイテレーション集合」へ分割し、それぞれ *LocalIter*(*pe*) と *NonlocalIter*(*pe*) で表すことにする。また、*SendSet*(*pe*) と *RecvSet*(*pe*) をそれぞれ「*pe* が他の PE に送らなければならないデータ集合」「*pe* が他の PE から受け取らなければならないデータ集合」を表すものとする。

forall 構文の実装であるメタプログラムはループの計算対象である *compound-statement* の構文木を解析し、*SendSet*(*pe*), *RecvSet*(*pe*), *NonlocalIter*(*pe*), *LocalIter*(*pe*) を求める。これらの集合はコンパイル時に静的解析を行って求められる場合もあるが、実行時にしか求められない場合もある。実行時に求める場合、集合を計算するコードを inspector と呼び、実際の計算を executor と呼ぶ。

ここで 3.2で示した forall の使用例の展開について考える。この例では以下の DistMatrix 型変数が参照されている。

```
A(i-1, 1-(i%2)..upper)
A(i, (1-(i%2))-1..upper-1)
A(i+1, 1-(i%2)..upper)
A(i, (1-(i%2))+1..upper+1)
```

イテレーションから PE への写像が A で与えられているので、行インデックスが i であるものについては、

常にローカルPEにデータがある。ゆえに通信を考慮する必要はない。一方、行インデックスが i-1 であるもの、i+1 であるものについては、通信を考慮する必要がある。

この場合、行インデックスがループ変数の一次式であるため、静的解析によって必要な通信を前もって求めることも容易であるが、今回はinspectorによってこれを実行時に計算する例を紹介する。以下に展開されたinspectorの疑似コードを示す。

```
if (flagFirstTime) {
  flagFirstTime = false;
  for (i = 0; i <= upper-1; i += 1) {
    flagLocalIter = true;
    if (A.owner(i) == myPE) {
      if (A.owner(i-1) != myPE) {
        add (i-1,i-(i%2)..upper,A.owner(i-1))
            to RecvSet(myPE);
        flagLocalIter = false;
      }
      if (A.owner(i+1) != myPE) {
        add (i+1,i-(i%2)..upper,A.owner(i+1))
            to RecvSet(myPE);
        flagLocalIter = false;
      }
      if (flagLocalIter)
        add i to LocalIter(myPE);
      else
        add i to NonlocalIter(myPE);
    }
  }
  compute SendSet(myPE) using ∪pe RecvSet(pe);
}
```

executorは以下のように展開される。

```
// send part
foreach (row, range, pe) ∈ SendSet(myPE) {
  send(pe, &(A(row, range.lower)),
  (range.upper-range.lower)*sizeof(double));
}
// local iteration part
foreach i ∈ LocalIter(myPE) {
  for (j = 1-(i%2); j <= upper; j += 2)
    A(i,j) = (omega/4.0 *
      (A(i-1,j) + A(i,j-1) + A(i+1,j) +
      A(i,j+1))) + ((1.0 - omega) * A(i,j));
}
// receive part
foreach (row, range, pe) ∈ RecvSet(myPE) {
  receive(pe, tmp_A[row]);
  offset_A[row] = range.lower;
}
// nonlocal iteration part
foreach i ∈ NonlocalIter(myPE) {
  for (j = 1-(i%2); j <= upper; j += 2)
    A(i,j) = (omega/4.0 *
    (tmp_A[i-1][j-offset_A[i-1]] + A(i,j-1)
   + tmp_A[i+1][j-offset_A[i+1]] + A(i,j+1))
   + ((1.0 - omega) * A(i,j));
}
```

5. forall構文のメタプログラム

5.1では、forall構文を4節で紹介した実行方式へ変換するメタプログラムの概要を説明する。また5.2では、このような変換を行うメタプログラム中に多く現われる構文木の走査についてその記述の一手法について述べる。

5.1 構文木の変換

forall構文は以下のクラスのSTOで実現する。

```
$meta class ForallStmt : public NewStmt5 {
    CompoundStmt* genInspector(AnalysisEnv&);
    CompoundStmt* genExecutor(AnalysisEnv&);
public:
    virtual SynObj* trans0();
}
```

ここで、NewStmt5は拡張構文のための雛型で、5つの *expression* をパラメータ式としてもつ *block-statement* のクラスである。パラメータ式はメンバ変数 expr1 ～ expr5、*compound-statement* はメンバ変数 body によってアクセス可能である。

forallの展開を行うコードは trans0 メソッドに記述される。

```
SynObj* ForallStmt::trans0()
{
    CompoundStmt* compStmt;
    // analyze body
    AnalysisEnv aenv(expr1);
    Analyzer analyzer(aenv);
    analyzer.traverse(body);
    // generate inspector & executor
    compStmt = new CompoundStmt(environment);
    compStmt->addStmt(`{
        $stmt; $stmt;
    }.attach(genInspector(aenv),
             genExecutor(aenv)));
    return compStmt->trans0();
}
```

analyzer オブジェクトは body の構文木を走査し、DistMatrix 型変数の参照を調べあげる。その結果は aenv オブジェクトに蓄えられる。

genInspector および genExecutor は inspector, executor の構文木を生成するメソッドである。構文木生成には aenv に蓄えられた情報を使用する。

`{} はメタプログラムのためのマクロ記法で Syntax Tree Constant(STC) と呼ぶ。attach メソッドは STC の中に書かれた $stmt を引数に置き換えた *compound-statement* を返す。

CompoundStmt クラスの addStmt メソッドは、その *compound-statement* のブロックの最後に引数で与えられた *statement* を追加する。上の例では、genInspector メソッドおよび genExecutor メソッドから返る2つの *compound-statement* から成る *compound-statement* が compStmt へ追加される。

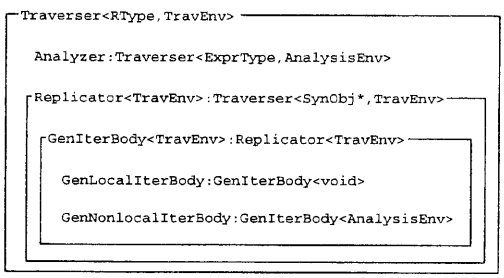

図3 構文木走査クラス階層

5.2 構文木の走査

前述の **analyzer** オブジェクトは与えられた構文木に含まれる各 *expression* についてループ不変性を調べる。また、出現する **DistMatrix** 型変数の参照について、その行インデックス式ごとに、列インデックスがとる値の範囲を調べる。この解析を行う為に、構文木の走査が必要である。

また、**genExecutor** メソッドでは、local iteration part の生成時および nonlocal iteration part の生成時に構文木を走査する。local iteration part の生成時には、走査によって **body** で与えられるオリジナルの構文木の複製を作る。ただしこのとき **body** の内部にあらわれる **forall** 文については **for** 文への変換を施す。nonlocal iteration part の生成時には、それに加え **DistMatrix** 型変数の参照を、リモート PE から受信したデータの参照に書き換える。

このようにメタプログラムにおける構文木変換には構文木の走査が多く出現する。構文木の走査に関連する処理には共用可能なコードも多く、プログラムの再利用性向上の見地からもクラスライブラリ化することが望ましい。

forall 構文の展開に必要な構文木走査クラスの階層を図3に示す。**Traverser** は全ての走査クラスの基底クラスとなる。**Traverser** の定義の一部を以下に示す。

```
$meta template <class RType, class TravEnv>
class Traverser {
public:
  TravEnv& travEnv;
  Traverser(TravEnv& env) { travEnv = env; }
  virtual RType traverse(SynObj* obj) {
    switch (SynObj::obj->getType()) {
    case STORAGE:
      return traverse((Storage*)obj);
    case EXPRNODE:
      return traverse((ExprNode*)obj);
    case STMT:
      return traverse((Stmt*)obj);
    }
    return (RType)NULL; // type error
  }
              ;              ;
};
```

走査は **traverse** メソッドで行う。**RType** は **traverse** メソッドの帰り値の型、**TravEnv** 型は、走査によって得られた情報の蓄積に使用するオブジェクトの型である。このクラスで定義している構文要素のクラスによるディスパッチコードは全ての構文木走査クラスで共用する。

analyzer オブジェクトのクラス **Analyzer** はクラステンプレート **Traverser** を継承する。**ExprType** は構文要素のループ不変性に関する情報を表現する型である。また **AnalysisEnv** は走査による解析結果を蓄積するオブジェクトのクラスである。**Analyzer** クラスでは STO を表現するすべてのクラスについて **traverse** メソッドを再定義する。これらのクラスの基底クラスについての **traverse** メソッド(クラスによるディスパッチコード) は **Traverser** での定義を継承する。

local/nonlocal iteration part 生成の手続きのほとんどは構文木の複製をつくることである。構文木の複製は **forall** 構文の実装だけに限らず、様々なメタプログラムで使用される可能性があるため、クラステンプレート **Replicator** を定義した。**Replicator** の **traverse** メソッドは引数に与えられた構文木の複製を返す。**Replicator** は **Analyzer** と同様、STO を表現するすべてのクラスについて **traverse** メソッドを再定義し、基底クラスについての **traverse** メソッドは継承する。

local iteration part の生成には **GenLocalIterBody** クラスを用い、nonlocal iteration part の場合には **GenNonlocalIterBody** クラスを使用する。**GenIterBody** クラステンプレートには local/ nonlocal に共通する **forall** 文から **for** 文への変換を記述する。

このように構文木を走査をクラステンプレートを用いて記述すると、走査の共通性を容易に表現でき、コードの再利用性が非常に高い。本節で挙げた例はメタレベルプログラミングにおいてもテンプレートが十分に有用であることを示している。

構文木を走査するコードは、クラステンプレートを用いずに表現することも可能である。例えば関数オーバロードを利用する方法、全ての STO のクラスを走査メソッドを追加する派生クラスで置き換える方法などが考えられる。しかし、いずれの方法もクラステンプレートを用いる手法に比べコードの簡潔性・再利用性に乏しい。

6. 関連研究

実行時メタレベル機能を用いて、並列・分散プログラムの最適化を行う研究がある[6)~8)]。AL-1/D[8)] や CodA MOP[7)] 等ではメタはベースレベルにおける局所的プログラムの意味を変更することができない。例えばオブジェクトに対するメッセージ通信式の意味を変更するようなことは出来ても複数の式からなるプログラム全

体の意味を変更することは不可能である。ABCL/R3[6]では、実行時にメタレベルでベースレベルのプログラムを操作することが出来、部分計算を可能としている。今回導入した `forall` 構文はABCL/R3でも実現が可能である。しかし、多くの場合 `forall` 構文で実行時部分計算による性能向上の可能性は少ないと考えられる。

OpenC++[1),2)]ではMPC++同様、ベースレベルプログラムの構文木の変換を行うメタプログラムを記述できる。しかし、MPC++では全ての構文要素の変換を変更可能であったのに対して、OpenC++ではベースレベルのクラス定義部とそのインスタンス式に限られるという違いがある。そのため、OpenC++における拡張構文は、拡張したクラスを定義し、そのクラスのインスタンス式という形で記述しなければならないという制約がある。

Anibus[9]はSchemeコンパイラにメタレベル機能を導入した初期の研究で、プログラムの並列化をメタレベルで記述するような応用が報告されている。しかし、(1) メタプログラムがベースレベルプログラムと異なる言語(CLOS)で記述されている、(2) メタプログラムにquasi-quoteのようなものがなく、明示的な構文木の操作を記述する必要があるため繁雑になるといった違いがある。

7. おわりに

`forall` に代表されるデータ並列構文の実装において、inspector/executorに基づく最適化されたコードを生成するためには構文木の解析が必要である。そのためC++のような言語ではこのような `forall` の機能をクラスライブラリで提供することはできない。MPC++はメタレベル機能を持ち、言語機能の追加、変更が可能である。このメタレベル機能を用いた `forall` 構文の実装について述べた。

MPC++はメタプログラムにターゲットプログラムの構文木の操作を記述する。本稿では構文木の変換の概要を示すことにより、MPC++のメタレベル機能が `forall` 構文の記述に十分であることを示した。また、構文木の変換に不可欠である構文木の走査について、メタプログラムにおける記述の一手法について述べた。この記述ではコードの再利用性を高めるためにクラステンプレートを使用した。これによりメタプログラミングにおけるクラステンプレートの有用性を示した。

本稿で示した `forall` 構文の実装はメッセージパッシング型分散メモリ環境を対象としたものである。他の並列計算機環境を対象とした実装もこれと同様に可能である。また、本稿で示した `forall` 構文の解析手法や展開手法は非常にナイーブであるが、より複雑な解析を施し、効率のよいコードに展開することも可能である。

MPC++が提供するメタレベル機能では、構文木に対するあらゆる操作が可能である一方、メタプログラマは構文木の結合関係を張るコードを意識的に記述する必要があるのため、メタプログラミングが簡単であるとはいい難い。現在、メタレベル機能をもちいた様々な言語機能の実装を試み、知見を集め、よりよいメタレベル機能の提供方法について検討を行っている。

謝辞 東京大学の増原英彦、田浦健次朗両氏には数々の貴重なアドバイスを頂いた。ここに感謝の意を表す。

参考文献

1) Chiba, S.: *OpenC++ Programmer's Guide for Version 2*. http://www.parc.xerox.com/spl/projects/oi-at-parc/openc++/.

2) Chiba, S.: A Metaobject Protocol for C++, *Proceedings of OOPSLA'95 (SIGPLAN Notices Vol.30, No.10)*, Austin, TX, ACM, pp. 285–299 (1995).

3) Ellis, M. A. and Stroustrup, B.: *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company (1990).

4) Ishikawa, Y., Hori, A., Sato, M., Matsuda, M., Nolte, J., Tezuka, H. and Konaka, H.: Design and Implementation of Metalevel Architecture in C++ — MPC++ Approach —, *Reflection Symposium'96*, San Francisco, CA (1996).

5) Koelbel, C. and Mehrotra, P.: Compiling Global Name-Space Parallel Loops for Distributed Execution, *IEEE Transaction on Parallel and Distributed Systems*, Vol. 2, No. 4, pp. 440–451 (1991).

6) Masuhara, H., Matsuoka, S. and Yonezawa, A.: Implementing Parallel Language Constructs Using a Reflective Object-Oriented Language, *Reflection Symposium'96*, San Francisco, CA (1996).

7) McAffer, J.: Meta-level Programming with CodA, *Proceedings of ECOOP95*, LNCS, Springer-Verlag (1995).

8) Okamura, H., Ishikawa, Y. and Tokoro, M.: AL-1/D: Distributed Programming System with Multi-Model Reflection Framework, *Proceedings of IMSA: Reflection and Meta-Level Architecture*, Tama City, Tokyo, pp. 36–47 (1992).

9) Rodoriguez Jr., L.: A Study on the Viability of a Production-Quality Metaobject Protocol-Based Statically Parallelizing Compiler, *Proceedings of IMSA: Reflection and Meta-Level Architecture*, Tama City, Tokyo, pp. 107–112 (1992).

(E-mail:tosiyuki@is.s.u-tokyo.ac.jp)