プログラミング 14－7
（1997. 8. 21）

メタレベル機能が支援するハイパフォーマンスオブジェクト指向計算

高橋俊行[†] 石川 裕[††]
佐藤三久[††] 米澤明憲[†]

我々はメタレベル機能を持つC++処理系であるMPC++上に、クラスライブラリに特化したオプティマイザを容易に記述するためのメタレベルライブラリを構築している。本ライブラリの一部として、オブジェクトのデータ依存を解析するデータフロー解析器クラスライブラリを提案する。また、メタレベルによるオブジェクトのデータフロー解析が可能となるように、ユーザ定義クラスのデータ依存情報を提供する手法を提案する。これら提案する枠組みによる並列クラスライブラリのためのオプティマイザの作成例により、その有効性を示す。

High Performance Object-Oriented Computing supported by Meta-level Architecture

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

We have been designing and developing a metalevel library that enables a class library developer to implement a class specific optimization using the MPC++ metalevel architecture. The data flow analyzers class library to analyze data dependencies of objects is proposed. The method, that tells the data dependency information of a user-defined class to the metalevel, is proposed to proceed the metalevel dependency analysis. To demonstrate the ability of the proposed framework, a parallel class library and its optimizer using the analyzers class library is shown.

1. はじめに

近年、オブジェクト指向プログラミング技術を並列数値計算などの高性能を要求するアプリケーションの作成にもちいる研究が盛んである。C++を基にした高性能な並列オブジェクト指向プログラミング環境を提供する研究では、主に次の2つのアプローチがとられている。ひとつは並列のための拡張構文やディレクティブ、プラグマなどを言語に導入するアプローチである。pC++[6]、CC++[3]、ICC++[5]やHPC++ level 1[2]などがこの方式である。このアプローチではコンパイラに並列機能についてのオプティマイザを組み込むことによって高い性能を得ることは可能である。しかし、並列機能が固定されているため、並列記述方式についての柔軟性が損なわれている。もうひとつのアプローチはC++の言語仕様を変更せず、ライブラリで並列機能を提供するアプローチである。ABC++[8]、POOMA[9]などがこの方式を用いている。この方式は並列機能についての柔軟性が高いものの、性能をひきだすことは難しい。なぜならコンパイラではオブジェクト間の最適化を行わない為に、並列機能を実現しているオブジェクトを利用しているプログラムが最適化されないからである。

我々が研究を進めているMPC++は2つのlevelをもつ。level 0ではマルチスレッド、リモート呼び出し、同期構造体、グローバルポインタなどの基本的な並列プリミティブをC++のテンプレートライブラリで提供する[7]。level 1ではメタレベル機能をもつC++コンパイラを提供する。MPC++のメタレベル機能は、コンパイラの構成部品の容易な記述を可能にし、さらにその構成部品をコンパイラにプラグインすることを可能にする機能である。構成部品はC++で記述され、その記述はメタレベルコードと呼ばれる。メタレベルコードは入力ソースコードと同じファイル中に記述することができる。MPC++コンパイラはソースコードの入力とともに、メタレベルコードとして記述されたコンパイラの構成部品を組み込む。組み込まれた構成部品は、続いて入力されるソースコードの解釈に使用される。

メタレベル機能を使用すれば、クラスライブラリ提供

† 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

者はクラスライブラリと共に、これに特化したオプティマイザをメタレベルコードとして提供することが可能になる。クラスに特化したオプティマイザは、クラスの実装および使用法についての知識を用いることで従来の最適化コンパイラでは実現できなかった特殊な最適化コードを生成するものである。

クラスに特化したオプティマイザは、level 0 のアプローチにおける欠点を克服する。並列機能の設計者は、MPC++ が提供する高性能並列オブジェクト指向プログラミング環境を用い、並列機能クラスライブラリと、そのクラスライブラリに特化したオプティマイザの組みを提供することが可能になり、提供される並列機能は柔軟性と性能を両立させたものとなる。この並列機能を使用するアプリケーションは MPC++ コンパイラを用いることで高い性能が引き出される。

我々は、クラスに特化したオプティマイザを容易に記述するため、メタレベルコードのライブラリ (以降、単にメタライブラリと呼ぶ) の構築している。これまでに以下のメタライブラリを提案、実装してきた[13),14)]。

- 構文木の走査を行うトラバーサクラス
- 構文木の複製を行うレプリケータクラス
- 構文木のマッチングを行うクラスライブラリ
- コード特殊化を支援するクラスライブラリ
- 簡易データフロー解析を行うクラスライブラリ

そして、これらのメタライブラリの有用性を確かめるために、分散配列クラスおよび分散共有クラスに特化したオプティマイザを作成した。オプティマイザは記述が簡素であるにもかかわらず、高い性能が得られるコードを生成する。最適化されたコードは C 言語で記述した同機能のプログラムと同等な性能を得ることができる。

クラスに特化した最適化では、オブジェクトが使用されている文や式に対して、クラスが持つ性質に応じたコード変換を行う。これを実現するためには、オブジェクトのメソッド呼び出し関係を調べることによる、オブジェクト間のデータ依存解析を行う必要がある。本稿ではこのような解析を行うデータフロー解析器の枠組を提案し、この枠組に基づいたメタライブラリを設計する。本メタライブラリの有用性を示すために、"依存する式の抽出" を行うデータフロー解析器クラスを実現し、これを用いた分散共有配列のキャッシュ更新最適化プログラムを示す。

2. MPC++ のメタレベル機能

2.1 メタレベルコードの記述

図 1に MPC++ level 1 コンパイラの構成を示す。MPC++ コンパイラは C++ ソースコードを入力とする。入力されたソースコードは構文解釈を経て、構文木となる。ひとつの宣言につきひとつの構文木が構成される。構成された構文木は図のように順次トラバーサオブジェクト群にわたされる。構文木の解析、変換はトラバーサオブジェクト群によってなされる。そして、最後のフェーズで構文木は C++ コードに再度戻される。出力された C++ コードは GNU G++ などの一般的な C++ コンパイラに用いて実行コードにコンパイルする。

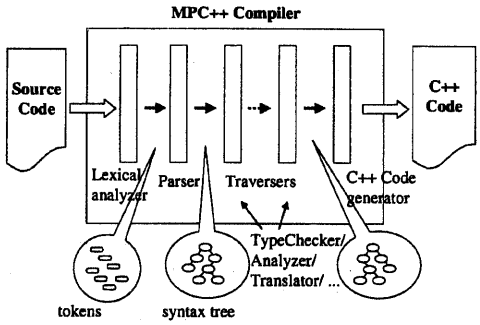

図 1 MPC++ level 1 コンパイラの構成

MPC++ のメタレベル機能は、トラバーサオブジェクトの記述を入力ソースコード中に入れることを可能にする。MPC++ コンパイラは **$meta** ではじまる宣言をメタレベルコードとして扱う。メタレベルコードは他のコードとは分離され、コンパイラ組み込みのインタプリタによって解釈される。メタレベルコードには関数やクラスなど、C++ の構文が使用可能である。構文 "**$meta** *compound-statement* " は特殊で、*compound-statement* 中のコードが即時解釈される。

図 2に、メタレベルコードが含まれる入力ソースコード例を示した。各クラスの詳細な記述は省略した。図 2のコード中のメタレベルコードは **DistArray** クラスのオプティマイザを定義するものである。メタレベルのクラス **DistArrayOpt** は **DistArray** クラスの最適化を行うトラバーサオブジェクトの定義である。2.2 節で説明するトラバーサクラスを継承してつくられる。関数 **registTraverser** の呼び出しは即時に解釈される。この関数はメタレベルプリミティブで、引数に与えられたトラバーサオブジェクトをトラバーサオブジェクト群へメンバとして追加するものである。ここで追加された新しいトラバーサオブジェクトは、続く関数 **foo** のコンパイルに使用される。関数 **foo** に含まれる **DistArray** クラスの使用は、メタレベルのクラス **DistArrayOpt** によって最適化される。

2.2 トラバーサクラス

MPC++ コンパイラは入力ソースコードから図 4のような構文木を生成する。この構文木のノードを表現するクラス階層は図 3に示した。トラバーサクラスには構文木を走査するオブジェクトに必要不可欠な次のコードが含まれる。

```
class DistArray;
$meta class DistArrayOpt : public Traverser;
$meta {
    registTraverser(new DistArrayOpt);
}
foo(DistArray &A, DistArray &B) {
    for(int i = A.getLower();
        i < A.getUpper(); i++)
        A[i] += B[i] + 1;
}
```

図2 トラバーサオブジェクトを登録するメタレベルコード

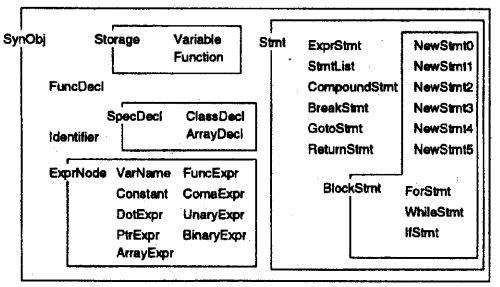

図3 構文要素のクラス階層 (一部)

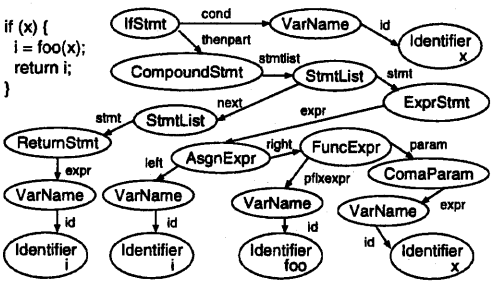

図4 構文解析の例

- 与えられたノードがどのクラスであるかを知り、クラス別の処理を呼び出すコード。
- 子ノードの走査を再帰的に行うコード。

図5にトラバーサクラスの宣言の一部を示す。名前が **Visit** で終っている仮想関数は、各ノードを訪れた際に呼ばれる関数である。実際に構文木を走査するクラスを記述する際には、この仮想関数群のうち必要な関数を再定義し、ノードクラス別の処理を記述する。それぞれの関数の引数と返り値の意味は次の通りである。

第1引数 "訪れているノード" へのポインタ。

第2引数 親ノードがもつ "訪れているノード" へのポインタへのポインタ。"訪れているノード" を別のノードへ置き換える場合に使用する。

返り値 true が返された場合のみ、"訪れているノード" の子ノードへの走査が行われる。

トラバーサクラスで定義されている各 **Visit** 関数は true を返すのみで他の処理は行わない。すなわち、子ノードへのトラバースを行うのみである。

```
$meta class Traverser {
public:
 Node *traverse(Node*);
 virtual bool compStmtVisit(CompStmt*, Stmt**);
 virtual bool forStmtVisit(ForStmt*, Stmt**);
 virtual bool funcExprVisit(FuncExpr*,ExprNode**);
 virtual bool asgnExprVisit(AsgnExpr*,ExprNode**);
 ....
};
```

図5 トラバーサクラス宣言 (の一部)

構文木を走査するクラスは、トラバーサクラスを継承することによって容易に記述することができる。

3. クラスに特化したオプティマイザ

クラスに特化したオプティマイザでは、クラスの実装や使用法についての知識を利用した次のような最適化が可能である。

- データフロー解析の複雑性から従来のコンパイラが行わなかった最適化。たとえば、メソッド呼び出し間にまたがったループ結合[4] や引数の値域についての知識を用いた部分計算[14] など。
- コードの変換にメソッドの意味知識を用いる最適化。たとえば、メソッド実行のプロファイリングを行うコードの生成など。

とくに後者の例では、プロファイリング対象となるメソッドとプロファイリング結果を処理するメソッドの双方についての知識が不可欠であるため、汎用オプティマイザでこれを実現することは不可能である。5節で示す分散共有配列クラスに特化したオプティマイザは、プロファイリングによってキャッシュ更新の最適化を行う。

このような最適化は、文脈に応じメソッド呼び出しの意味変更を行うものである。最適化のフェーズでは、オブジェクト間のデータの伝搬について知り、各メソッド呼び出しの依存関係をあきらかにすること、すなわち、オブジェクト間データフロー解析が重要な役割を示す。

4. オブジェクト間データフロー解析

本節ではオブジェクト間のデータフロー解析 (ODFA) のための枠組を提案し、その枠組に基づいたメタライブラリを設計する。

4.1 ODFA における問題

C++ の仕様では ODFA が難しい理由を以下に示す。

- メソッド呼び出しの影響解析に必要とされるポインタ解析および手続き間解析の計算量が、コード量に対し指数関数的に増える[11]。
- コントロールフローがつかめないコードによるポインタの伝搬は解析が不可能。

とくにマルチスレッド実行のための並列ライブラリは、後者の理由からデータフロー解析が不可能である。

```
$meta class ODFA {
  ODFA(FlowFunc, Meet, FlowEq);
  analyze(Node* tree, int Iter);
};
$meta class FlowVar {
  virtual int equal(FlowVar, FlowVar);
};
$meta class Meet {
  virtual FlowVar meet(FlowVar, FlowVar);
};
$meta class FlowFunc {
  virtual FlowVar flow(Node*, FlowVar);
};
$meta class FlowEq {
  FlowEq(FlowFunc, Meet);
  virtual void init(FlowVar&);
  virtual int solve(Node*, FlowVar&, FlowVar&);
};
```

図6 データフロー解析器のためのメタライブラリ

4.2 ODFA の枠組の提案

そこで我々は、以下の要素からなる ODFA の枠組を提案する。

オブジェクト依存情報の記述 解析にさきだって、i) オブジェクトのデータフローはメソッド呼び出し間の依存関係として明示的に指定し、ii) エイリアスをもたないオブジェクト群を明示的に指定する、ことによって単純な解析を可能にする。

ODFA 問題の記述 データフロー解析問題の構成要素である、フロー値、ミート演算、フロー関数およびフロー方程式それぞれをクラスライブラリで構成。このことは、ODFA 問題の記述コードの再利用性を高める。

オブジェクト依存情報は、クラスの提供者がクラスの実装や使用法についての知識をもとに記述する。この情報はすべての ODFA から共有される。

ODFA 問題は従来の DFA 問題[1)12)10)] と同様の枠組でとらえる。従来の DFA では対象となるデータ構造は基本データ型であったが、ODFA では対象はユーザ定義型 (クラス) である。DFA における USE(値の参照)/DEF(値の代入) 操作は、ODFA ではクラスメソッドに対応する。オブジェクト間のデータの伝搬は、各オブジェクトの依存関係によってとらえる。

4.3 ODFA のメタライブラリ

このような枠組に基づく ODFA のためのメタライブラリを設計した (図 6)。クラス **ODFA** はデータフロー解析器のクラスである。フロー関数、ミート演算、フロー方程式を表現するオブジェクトをコンストラクタ引数にとる。解析は **analyze** メソッドによって行われる。第 1 引数には解析対象となる構文木、第 2 引数にはフロー解析の繰り返しの最大値を与える。

オブジェクト依存情報の記述は 4.3.1節に示すインターフェースで与える。

ODFA 問題は、フロー値、ミート演算、フロー関数およびフロー方程式を **FlowVar**, **Meet**, **FlowFunc**, **FlowEq** 各クラスの継承クラスに記述する。記述例を 4.3.2節で示す。

```
$meta{
methodDepends("DistArray::init(int)",
              "DistArray::operator[](int)");
methodDepends("Gvalue::operator=(double)",
              "Gvalue::operator double()");
methodArgDepends("X::foo(Y&, Z&)", 2, "Z::bar()");
noAliasedClass("CachedArray");
retInternalObj("CachedArray::operator[](int)");
}
```

図7 ODFA メタライブラリへのオブジェクト依存情報の提供例

4.3.1 オブジェクト依存情報の記述

オブジェクトのデータフローをメソッド呼び出し間の依存関係として指定するために、各クラスのメソッド実行が与える以下の影響についての記述が必要である。

- オブジェクトの状態が変更される場合、その変更が、どのメソッド実行に影響を与えるのか。
- 引数として渡されるオブジェクトの状態が変更される場合、その変更が (引数オブジェクトの) どのメソッド実行に影響を与えるのか。

また、エイリアスをもたないオブジェクト群を明示的に指定するため、オブジェクトのメモリ上での表現についての記述が必要である。

- 別々の名前で指される同一クラスの複数オブジェクトが、互いに素なメモリブロックで構成されているかどうか。
- メソッド呼び出しの帰り値がオブジェクトの内部状態を指す領域へのポインタかどうか。

以下にこれらの情報を **ODFA** メタライブラリへ提供するためのインターフェースを示す。

methodDepends 引数で与えるメソッド実行に依存があることを指定。

methodArgDepends メソッド実行の引数オブジェクトに対する依存を指定。

noAliasedClass 指定されたクラスのオブジェクトはエイリアスをもたない。

retInternalObj 指定されたメソッドの、返り値はオブジェクトの内部状態。

このインターフェースの使用例を図 7に示す。

4.3.2 ODFA 問題の記述

ここで、ODFA 問題の例として、"依存のある式の抽出" を行う解析を示す。(この解析の使用例は 5.2 節で示す。) この ODFA は、メソッド呼び出し式の集合を与えたときに、与えられた構文木から、そのメソッド呼び出し式と依存関係のあるメソッド呼び出し式をすべて抽出し、その集合を返すものである。

フロー値 「メソッド呼び出し式の集合」である。変数名とメソッド名のペアのリストで表現する。

ミート演算 「メソッド呼び出し式の集合」の和集合を

計算する。

フロー関数　注目しているノードの式が、引数のフロー値(メソッド呼び出し式の集合)のどれかの式に依存のあるメソッド呼び出しであれば、そのメソッド呼び出しをフロー値に加え、それを返り値とする。

フロー方程式　先頭ノードにおけるフロー変数の初期値は、このODFAの開始にあたって与えられた"メソッド呼び出し式の集合"。各ノードにおけるフロー方程式は、入力フロー値＝前ノード群の出力フロー値にミート演算を施したもの。出力フロー値＝入力フロー値にフロー関数を適用したもの。

5. 分散共有配列クラスの最適化

本節では、分散共有配列クラスに特化したオプティマイザの実装例を示す。

5.1 分散共有配列クラス

MPC++ level 0で提供されている並列プリミティブを使用した分散共有配列クラスライブラリを示す。分散共有配列クラス **CachedArray** では、要素は各プロセッサに分割されている。値の変更はローカルプロセッサにある要素のみ許される。各プロセッサはリモートプロセッサが所有する全要素を保存するだけのキャッシュ領域をもち、プロセッサ間コヒーレンシは **CachedArray** クラスの **update** メソッドによって保たれる。**update** メソッドはローカルプロセッサが所有する全ての要素値を他の全てのプロセッサへ配布する。しかし、各々のリモートプロセッサは必ずしも、ローカルプロセッサのすべての要素を参照するとは限らないので、このようなキャッシュ更新は効率が悪い。

そこで、実際の計算に先だって、あらかじめ参照されるインデックスについてのプロファイリングを行い、キャッシュ更新の際には、リモートプロセッサから参照される要素のみを配布する機能をもつ分散共有配列クラス **InspectedArray** を実装した。このクラスは **CachedArray** クラスを継承している。

プロファイリング処理を行うコードをインスペクタと呼ぶ。インスペクタでは、まずローカルプロセッサが参照するインデックスを引数にして、**ref** メソッドを呼び、ローカルプロセッサが参照するインデックスの集合をつくる。このインデックス集合は **updateRef** メソッドで全リモートプロセッサへ配布される。リモートプロセッサから受け取った、各リモートプロセッサが参照するインデックス集合をもとに、各プロセッサは、リモートプロセッサが参照する要素インデックスの表を作成する。図8の計算のためのインスペクタを図9に示した。

5.2 インスペクタ生成器

InspectedArray クラスを使用している関数についてのインスペクタを自動生成するトラバーサオブジェクト **InspectGen** を実現する。**InspectGen** ではインスペクタ関数を生成するタイミングをとるために、**inspect**

```
void matvec(InspectedArray<double, Block> &x,
            CachedArray<double, Block> &y)
{
  int i, j, k, n;
  k = a.getLower();
  for (i = y.getLower();
       i < y.getUpper(); i++) {
    y[i] = 0.0;
    n = rowstr[i+1] - rowstr[i];
    for (j = 0; j < n; j++) {
      y[i] += a[k] * x[colidx[k]];
      k++;
    }
  }
}
```

図8　オリジナル関数

```
void matvec_i(InspectedArray<double, Block> &x,
              CachedArray<double, Block> &y)
{
  int i, j, k, n;
  k = a.getLower();
  for (i = y.getLower();
       i < y.getUpper(); i++) {
    n = rowstr[i+1] - rowstr[i];
    for (j = 0; j < n; j++) {
      x.ref[colidx[k]];
      k++;
    }
  }
  x.updateRef();
}
```

図9　インスペクタ関数

という特別な関数呼び出しをユーザに記述してもらう。**inspect** 関数の引数の意味は以下の通りである。

第1引数　オリジナルの関数へのポインタ

第2引数以降　インスペクタ関数への実引数

InspectGen は例えば、**inspect(matvec, p, q)** という関数呼び出しに出会うと、図10で示した **funcExprVisit** メソッドを実行する。**funcExprVisit** メソッドは、その関数呼び出しを **matvec_i(p, q)** という関数呼び出しに変更すると共に、図9に示すインスペクタ関数を作成する。以下に **funcExprVisit** メソッドのアルゴリズムを具体的に説明する。

(1)　**inspect** 関数の実引数から、オリジナルの関数定義とインスペクタ関数への実引数を得る。

(2)　新しい関数名でオリジナル関数の複製をつくる。これをインスペクタ関数にする(**replicateAs**)。

(3)　「依存のある式の集合」を保持するオブジェクト **depExpr** をつくる。

(4)　インスペクタ関数を走査し、**InspectArray** クラスの **operator[]** メソッド呼び出しを探す。その引数式(インデックス)を **depExpr** に加える(**findInspectIndex**)。

(5)　"依存のある式の抽出(4.3.2節)"を行うデータフロー解析を行う(**depAnalysis**)。初期値は

```
int
InspectGen::funcExprVisit(FuncExpr *expr,
                          ExprNode **ret)
{
  FuncDecl *fdecl = expr->getFuncDecl();
  if (fdecl != fdeclInspect) return TRUE;
  ComaParam *param = expr->getParam();
  fdecl = findFuncDecl(param->getParam());
  Identifier *name = genSym();
  fdecl = replicateAs(name, fdecl);
  SetExpr depExpr;
  findInspectIndex(&depExpr, fdecl);
  depAnalysis(&depExpr, fdecl);
  fdecl = codeElimination(fdecl);
  insertUpdateRef(fdecl);
  *ret = newFuncExpr(fdecl, param->getNext());
  return FALSE;
}
```

図10 インスペクタ生成器

depExpr、解析対象はインスペクタ関数である。解析結果は depExpr に返される。

(6) depExpr に含まれない式をインスペクタ関数から消去する。その際に、InspectArray クラスの operator[] メソッド呼び出しは ref メソッドの呼び出しに変更する (codeElimination)。

(7) インスペクタ関数の出口に updateRef メソッドの呼び出しを挿入する (insertUpdateRef)。

(8) funcExprVisit の帰り値をインスペクタ関数の呼び出しに変更する。

6. まとめ

クラスライブラリ作成者がクラスに特化したオプティマイザの記述を容易にする MPC++ メタライブラリを実現する為に、データフロー解析を行うメタライブラリの枠組を提案した。一般に、オブジェクト間にまたがるデータフロー解析は不可能な場合が多いが、我々は、i) オブジェクトのデータフローはメソッド呼び出し間の依存関係として明示的に指定し、ii) エイリアスをもたないオブジェクト群を明示的に指定する、ことによって解析を可能にした。これらの指定はクラスを定義したプログラマによってなされる。このような枠組を用いたデータフロー解析器の例として、依存のある式を抽出するデータフロー解析器クラスを実現した。そして、これを用いた分散共有配列のキャッシュ更新最適化プログラムを示すことによって我々のメタライブラリの有用性を示した。

参考文献

1) Aho, A. V., Sethi, R. and Ullman, J. D.: *Compilers: Principles, Techniques, and Tools.*, Addison-Wesley (1988).

2) Beckman, P., Gannon, D. and Johnson, E.: *Portable Parallel Programming in HPC++.* http://www.extreme.indiana.edu/hpc++/.

3) Chandy, K. M. and Kesselman, C.: CC++: A Declarative Concurrent Object-Oriented Programming Notation, *Research Directions in Concurrent Object Oriented Programming* (G. Agha, P. Wegner, A. Yonezawa(ed.)), MIT press (1993).

4) Chiba, S.: *A Study of a Compile-time Metaobject Protocol*, PhD Thesis, Graduate School of Science, The University of Tokyo, Japan (1996).

5) Chien, A., Reddy, U., Plevyak, J. and Dolby, J.: ICC++ – A C++ Dialect for High Performance Parallel Computing, *Object Technologies for Advanced Software*, Lecture Notes in Computer Science, Vol. 1049, Springer-Verlag, pp. 76–95 (1996).

6) DennisGannon, J. K.L.: Object Oriented Parallelism: pC++ Ideas and Experiments, *Proceedings of 1991 Joint Symposium of Parallel Processing*, pp. 13–22 (1991).

7) Ishikawa, Y.: *Multiple Threads Template Library.* http://www.rwcp.or.jp/lab/mpslab/mpc++/mpc++.html.

8) O'Farrell, W. G., Eigler, F. C., Pullara, S. D. and Wilson, G. V.: ABC++, *Parallel Programming Using C++*, The MIT Press, pp. 1–42 (1996).

9) Reynders, J. V. W., Hinker, P. J., Cummings, J. C., Atlas, S. R., Banerjee, S., Humphrey, W. F., Karmesin, S. R., Keahey, K., Srikant, M. and Tholburn, M.: POOMA, *Parallel Programming Using C++*, The MIT Press, pp. 547–588 (1996).

10) Tjiang, S. W. K.: *Automatic Generation of Data-flow Analyzers: A Tool for Building Optimizers*, PhD Thesis, Computer Systems Laboratory, Stanford University (1993).

11) Wilson, R. P. and Lam, M. S.: Efficient Context-Sensitive Pointer Analysis for C Programs, *ACM SIGPLAN'95 Conference on Programming Language Design and Implementation*, La Jolla, CA, ACM, pp. 1–12 (1995).

12) Zima, H. and Chapman, B.: *Supercompilers for Parallel and Vector Computers*, Addison-Wesley (1991).

13) 高橋俊行, 石川裕, 佐藤三久, 米澤明憲: メタレベル機能による並列プログラミング, *IPSJ SIG Notes*, Vol. 96, No. 82, pp. 79–84 (1996).

14) 高橋俊行, 石川裕, 佐藤三久, 米澤明憲: メタレベル機能によるクラスライブラリ最適化手法, 並列処理シンポジウム JSPP'97 (1997).

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