メタレベル機能によるクラスライブラリ最適化手法

高橋俊行[†] 石川 裕[†]
佐藤三久[†] 米澤明憲[††]

クラスに代表されるオブジェクト指向言語機能は，ソフトウェアを構成する部品のライブラリ化を促進するという利点がある．しかし，多くのC++処理系では，クラスライブラリのメンバ関数呼び出しコードの最適化が不可能であるため，このようなコードは，クラスを用いずに記述したコードと比べ，高い性能を得ることができない．我々はコンパイル時メタレベル機能を持つC++処理系でクラスライブラリごとに特化したオプティマイザを用いることで，この問題を回避する．本稿では並列計算のための分散配列クラスライブラリを例題に選び，このクラスライブラリに特化したオプティマイザの記述方法を提案する．オプティマイザはメタレベル機能を用いて記述され，コンパイラに組み込まれる．オプティマイザの作成はメタライブラリを用いた差分プログラミングで行うため容易である．我々は，このオプティマイザによる最適化結果を確認するためにいくつかの評価実験を行った．最適化されたコードはクラスライブラリを用いないコードとほぼ同じ実行速度で動くことを確認している．

Optimization Technique for Class Libraries using Meta-level Architecture

TOSHIYUKI TAKAHASHI,[†] YUTAKA ISHIKAWA,[†] MITSUHISA SATO[†]
and AKINORI YONEZAWA[††]

Object-oriented language features such as *class* make it easier to break down software into re-usable components. However, C++ compilers could not optimize a program that calls member functions in class libraries, the program using class libraries has disadvantage of performance, compared with the program that does not use class libraries. To solve such a problem, we build optimizers specific to class libraries using a C++ compiler that has a compile-time meta-level architecture. In this paper, we propose an optimization technique specific to a class library that implements distributed array for a parallel environment. The optimizer is implemented and is incorporated into the compiler using the meta-level architecture. By using Meta-library, an object-oriented library for optimizers it is easy to build such optimizers. To demonstrate effectiveness of the optimizer specific to the distributed array library, we have implemented some benchmark program. The experimental results show that the optimized code runs as fast as the code that does not use class libraries.

1. はじめに

ソフトウェアの開発においては再利用性，可搬性および実行性能が重要視される．再利用性や可搬性を高めるためにユーザ定義型，継承，型多相といった抽象化を施すことが有効である．C++[4]はクラスやテンプレートなどの言語機能によって，このような抽象化をサポートしている．

数値計算ソフトウェアの記述にC++を利用すると，並列アルゴリズムを部品化し，ライブラリとして提供することが可能になる．その一例として，本稿では，データ並列計算のための分散配列ライブラリ作成例を紹介する．この分散配列ライブラリはクラステンプレートを利用して実装されており，要素型と要素の分散方式を型パラメータとする．ライブラリ使用者は要素の分散アルゴリズムを自由にカスタマイズすることが可能である．

このように，C++が提供するクラスなどの言語機能はライブラリの記述性を高め，ソフトウェアの再利用性や可搬性を高める．その一方で，これらの言語機能はソフトウェアの実行性能を低下させる欠点がある．本稿で紹介する分散配列ライブラリも，ライブラリを使用しなかった場合に比べ2倍近い実行性能の低下がみられた．そこで我々は，コンパイル時メタレベル機

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

能を持つC++のうえで，このメタレベル機能を用い，同ライブラリに特化したオプティマイザを設計した．このオプティマイザは，同ライブラリを使用するコードの文脈を解析し，最適なコードを生成する．生成されたコードはライブラリを使用しなかった場合のコードに匹敵する実行性能を発揮する．

本稿の構成は以下のとおりである．2章ではC++でのクラス使用による実行性能低下の問題について述べる．3章でクラスライブラリに特化した最適化を提案し，オプティマイザの記述に有用であるメタレベル機能について説明する．そして，4章で紹介する分散配列ライブラリのオプティマイザを5章で示す．6章では現実のアプリケーションを用いた評価実験結果を，そして7章では関連研究を紹介する．

2. クラスライブラリ化による問題

クラスやテンプレートなど，オブジェクト指向言語機能はソフトウェアの部品化を進め，高度なライブラリの構成を可能にする．しかし，一般に，このようなライブラリを用いたソフトウェアは，そうでないソフトウェアに比べ実行性能が劣る．ここではC++のテンプレートおよびクラスについて，その使用が実行性能に与える影響について述べる．

2.1 テンプレートの使用による影響

テンプレートの実装方式は，C++処理系によって異なる．一般的な処理系は，構文解析のフェーズにおいて，型パラメータに，実際に使用される型を代入した結果のクラスや関数，すなわち具体化されたクラスや関数を自動生成する．多くのC++処理系は，具体化されたクラス群や関数群を，プログラマがテンプレートを使用せずに記述したクラス群や関数群と同様に扱うので，テンプレートの使用が実行性能に影響することはない．

2.2 クラスの使用による影響

2つ以上の名前が同一のメモリ領域を指すように使用することをエイリアシングという．エイリアシングはコードの最適化を難しくする．なぜなら，C++が持つキャストや多彩なポインタ演算機能が「2つのポインタが同一のメモリ領域を指していること」をコンパイル時に判定することを難しくしており，エイリアシングによって共有されるメモリ領域への副作用をコンパイル時に解析することを困難にするからである．一般に，クラスを使用したC++プログラムはクラスを使用しないプログラムと比べ実行性能が劣るが，これは，エイリアシングを考慮し，オプティマイザがクラスメンバ変数を読み書きするコードの最適化を行わないことに起因する．

```
class SimpleArray {
private:
    double *data;
    int size;
public:
    void init(int n)
        { data = new double[n]; size = n;}
    double& operator[](int i)
        { return data[i]; }
    int getSize()
        { return size; }
};
```

図1 SimpleArrayクラス
Fig. 1 Class SimpleArray.

```
bar(SimpleArray& A, SimpleArray& B, int N) {
    for (int i = 0; i < N; i++)
        A[i] = B[i];
}
```

図2 SimpleArrayクラス使用例（関数bar）
Fig. 2 Example using SimpleArray (bar).

```
        mov 0,%o3           ! i = 0
    L46:
        cmp %o3,%o2         ! i < N
        bge L54             ! exit loop
        sll %o3,3,%o0
        ld [%o4],%o1        ! load A.data
        ld [%o5],%g2        ! load B.data
        ldd [%g2+%o0],%g2   ! load B[i]
        add %o3,1,%o3       ! i++
        b L46               ! continue
        std %g2,[%o1+%o0]   ! store A[i]
    L54:
```

図3 関数barのコンパイル結果
Fig. 3 Output of a C++ Compiler (bar).

メンバ変数の読み書きは，とくにループ内では実行性能に非常に大きく影響する．ここで簡単な配列クラスを例に説明する．**図1**に示すSimpleArrayクラスは，**double**型の配列クラスの一表現である．**図2**に示す関数をG++ 2.7.2（sparc-sun-sunos4.1.4）を用い-O3でコンパイルした結果の一部分（for文に相当する部分）を**図3**に示した．

図2における計算ではイテレーション内の計算が**data**メンバの値に副作用を及ぼすことはない．ゆえに，**data**メンバはループ不変値であり，その読み込みはループの前に1度であるようなコードが生成されることが望ましい．しかし，図3のコードでは，イテレーションごとに**A.data**，**B.data**がメモリから読み込まれており，この読み込みが実行性能に大きく影響している．**data**メンバの値をイテレーションごとに

読む理由は，イテレーション内の計算が，data メンバの値に副作用を及ぼさないことをコンパイラが保証しなかったからである．プロシージャ間解析を施すことで，これを保証することも可能であるが，この解析は複雑であるため，多くのC++コンパイラは，これを行わない．

3. メタレベル機能を用いたオプティマイザ

3.1 クラスライブラリに特化した最適化

メンバ変数のアクセスについて最適化を施す方法には，次の2種類が考えられる．

- 言語仕様を変更し，エイリアシングを制限
- クラスに特化したオプティマイザを提供

言語仕様の変更手法には，(1) エイリアシングを制限するコンパイルオプションやプラグマを導入する方法，(2) エイリアシングを制限するポインタ修飾子を導入する方法があげられる．

エイリアシングを制限するコンパイルオプションやプラグマは，これで指定されるソースプログラム内でエイリアシングが行われていないことを，プログラマがコンパイラに対し保証するものである．この方法では，パフォーマンス向上のために，内部でエイリアシングを使用しているインライン関数ライブラリが使用できなくなるという問題がある．

また，エイリアシングを制限するポインタ修飾子は，「この修飾子がついたポインタ型が指すオブジェクトはエイリアシングがなされていない」ということを，プログラマがコンパイラに対し保証するのに使用する．このポインタ型は，循環構造を表現するポインタには使用できない．また，ポインタが配列を指しているとき，その配列の各要素についてエイリアシングの許可不許可を制御することもできない．

一方，クラスに特化したオプティマイザはライブラリプログラマとライブラリユーザの間で交わされたクラスライブラリの使用についての取決めを利用して，そのクラスに特有の最適化を施すものである．クラス使用についての取決めは，主にエイリアシングを制限するものである．エイリアシングが制限されたもとで，オプティマイザはメソッドの呼び出しをインライン展開するが，そのときオプティマイザは，あらかじめオプティマイザに与えられている，メソッド内部で使用される変数の（メソッド呼び出しについての）依存関係を利用し，共通部分式やループ不変値の発見が容易なコードを生成する．この生成されたコードを既存のオプティマイザ——共通部分式削除を施したり，ループ不変式のつりあげを施すオプティマイザ——で処理することによって，パフォーマンスの高いコードを得ることができる．またコード生成時には，あらかじめオプティマイザが知っている「メソッド間の依存関係」を利用すれば，一連のメソッド呼び出し式についての最適化も可能である．

クラスに特化したオプティマイザを使用することの利点に，最適化に必要な計算コストが少ないことがあげられる．汎用的なオプティマイザは，最適化に先立ちソースを走査しながら，メソッドで使用される変数の依存関係を抽出する必要がある．一方，クラスに特化したオプティマイザは，すでに依存関係を知っているので，コンパイル時に抽出する必要がない．ゆえに，コンパイル時間が短縮される．

その半面，クラスに特化したオプティマイザを作成することの負担は無視できない．ライブラリプログラマはクラスライブラリに加えて，オプティマイザのコーディングをする必要がある．

3.2 MPC++ のメタ機能

MPC++[7] はコンパイル時メタレベル機能をサポートするC++処理系である．コンパイル時メタレベル機能はコンパイラ構成部品の必要に応じた組替えを可能にする．MPC++ ではメタレベル機能をC++のsource-to-sourceトランスレータで実現している．このトランスレータをMPC++ フロントエンドコンパイラとも呼ぶ．フロントエンドコンパイラの出力は標準C++プログラムである．フロントエンドコンパイラによって変換されたC++プログラムは，バックエンドコンパイラ（GNU G++やSun C++などの既存のC++処理系）を使用し，オブジェクトコードへコンパイルされる．

フロントエンドコンパイラでの変換を実行するプログラムをメタプログラムと呼ぶ．クラスに特化したオプティマイザはメタプログラムで記述することができる．メタプログラムは，C++の構文木を操作するプログラムであり，入力として与えられた構文木の走査を何回か行い，適当な組替えを施して出力する．メタプログラムの記述にはC++が使用される．構文要素はすべてC++のクラスを用いて表現されている．この構文要素とは構文木のノードの構成要素で，C++の文法ルールに現れる，すべての終端，非終端記号を指す．

構文木の走査はプログラム変換に不可欠な作業である．たとえば，一般にオプティマイザは走査を通じ，共通部分式やループ不変式を探す．そして走査で集められた情報を用い，最適化されたプログラムを生成する．この生成の際にも走査が必要である．走査を行い，

コードの複製をつくる一方で必要な所でコードの置換えを施すのである．走査のためのコードには再利用できる部分が多い．そこで，我々はクラステンプレートを用い，走査アルゴリズムをライブラリ化した[9]．オプティマイザの記述には，このメタライブラリを使用する．

4. 分散配列ライブラリ

4.1 基盤となる並列環境

紹介する分散配列ライブラリは，新情報処理開発機構で開発されたMTTL（Multi Thread Template Library）[6]を基盤にする．MTTLは，分散メモリ型並列計算環境にSPMDスタイルのマルチスレッドプログラミングを提供するC++クラスライブラリである．

MTTLでは同期/非同期リモート呼び出し関数，同期オブジェクト，グローバルポインタなどの並列アブストラクションが用意されている．しかし，MTTLが提供する並列機能はプリミティブである．実際のアプリケーションを作成する際には，より高度なアブストラクションを使用することが望ましい．本稿で紹介する分散配列ライブラリは，数値計算ソフトウェアを作成する場合に有用な，データ並列計算のためのアブストラクションである．

4.2 分散配列ライブラリの設計

この分散配列ライブラリでは，分散メモリ環境上での配列の表現手段として，2種類の配列クラス，分散配列クラスDistArrayおよびキャッシュ付き分散配列クラスCachedArrayを用意した．各配列クラスは，要素型と，分散配置アルゴリズムをそれぞれ型パラメータとしてうけとるクラステンプレートである．

分散配列クラスは，配列を分散配置アルゴリズムに従って分割し，各プロセッサに配置する．ローカルメモリに配置されていない配列要素の読み書きは，読み書きごとの通信によって解決される．

キャッシュ付き分散配列クラスでは，すべてのプロセッサが配列全体を保持できるだけのメモリを持つ．このメモリのうち，分散配置アルゴリズムによってローカルに指定される領域以外は，キャッシュとして使用される．`update`メソッドを起動すると，ローカルに指定されている領域の要素が他のプロセッサへ配布される．

主に前者（分散配列クラス）は計算中に更新がほとんどない行列等の表現に使用し，後者（キャッシュ付き分散配列クラス）は更新が頻繁に行われる行列等の表現に使用する．後者を使用する場合，アプリケーションプログラムはキャッシュされている値に矛盾が生じないように，適当な箇所で`update`メソッドを呼ばなくてはならない．

2つの配列クラスに共通なメソッドを以下に示す．

init イニシャライズ．要素数，初期値および分散配置アルゴリズムのインスタンスオブジェクトを引数にとる．

operator[] インデックスを引数としてとり，要素への参照を返す．

getUpper ローカルメモリに配置されている要素のインデックスの上限を返す．

getLower ローカルメモリに配置されている要素のインデックスの下限を返す．

getLocalDist initメソッドで設定された分散配置アルゴリズムのインスタンスオブジェクトを返す．

分散配置アルゴリズムは要素の分割を表現するクラスである．分散配置アルゴリズムには次のメソッドが定義されていなければならない．

init イニシャライズ．要素数を引数にとる．

owner インデックスを引数としてとり，要素が配置されているプロセッサ番号を得る．

own 引数で与えるインデックスが表す要素がローカルメモリに配置されているか否かを知る．

index インデックスを引数としてとり，ローカルメモリ上でのインデックスに変換する．

これらのメソッドは配列クラスとのインタフェースに用いられる．

現在のインタフェースでは，1つのプロセッサに与えられる配列の分割は，連続したインデックスを持つ1つの塊である必要がある．ゆえにブロックサイクリック分割などの表現はできない．これらの表現のためには，配列クラスに塊の数を保持するメンバ変数を追加し，`index`メソッドや`getUpper`，`getLower`メソッドの引数に塊の番号を追加すればよい．

5. 配列ライブラリに特化したオプティマイザ

5.1 SimpleArrayクラスのオプティマイザ

まず，図1で示したSimpleArrayクラスについてのオプティマイザを例に，我々の最適化手法を説明する．このオプティマイザは`operator[]`メソッドのインライン展開を行う．その際，メソッドがアクセスするprivateメンバ変数のうち，メソッド呼び出しによる副作用がみられないものを，なるべくレジスタに配置させるコードを展開する．

5.1.1 オプティマイザの仕様

まずSimpleArrayクラスのユーザは，このオプティマイザが持つ次の制限を認識しておかなければならな

い．以下の制限は最適化の効果を最大限に得，その一方でオプティマイザの実装を容易にするための妥協である．ここでブロックとは *compound-statement* のことを指す．また，**init** メソッドはこのクラスの private メンバ変数の値を変更する唯一のメソッドである．

(1) ユーザはキャスト等を利用してこのクラスの private メンバ変数にクラス外からアクセスするコードを記述してはならない．private メンバのクラス外からのアクセスは禁止されているが，対象となるオブジェクトを，アクセス制限の緩い型にキャストすれば，そのメンバ変数へのアクセスも可能となる．しかし，このようなアクセスは，その private メンバ変数がどこで変更されうるか解析することを難しくする．

(2) 入口が先頭に限られているブロックで，なおかつ **init** メソッドの呼び出しが絶対に発生しないことを，オプティマイザがフロー解析を施すことによって判定できるもののみが，最適化の対象となる．**init** メソッド呼び出しを含むブロック，ソースコードがコンパイル時に得られない関数☆の呼び出しを含むブロック，およびラベルを持つブロックは最適化されない．

図 2 で示されたプログラムは，このオプティマイザによって**図 4** で示されるプログラムへ変換される．関数 bar のボディブロックからは **init** メソッドが呼ばれないこと，および，このボディブロックへの入口は，このブロックの先頭のみであることから，このボディブロックに含まれる **operator[]** メソッドはオプティマイザが展開する．その際，private メンバ変数 **data** はブロック内で値が変更されないから，一時変数 **_A_data**，**_B_data** に，その値が保持される．この一時変数は，for 文のブロック内ではループ不変式である．ゆえに既存の多くの C++コンパイラに実装されているオプティマイザによって，図 4 のプログラムは図 2 のプログラムに比べ実行性能が高いアセンブラコードに変換される．

5.1.2 オプティマイザの実装

SimpleArray クラスに特化したオプティマイザは 100 行に満たない C++コードである．**図 5** に SimpleArray クラスについてのオプティマイザのメタプログラムのメインルーチンを示した．関数 **optimizeSimpleArray** は引数に与えられた構文木に対し最適化を施したものを返す．

```
bar(SimpleArray& A, SimpleArray& B, int N) {
    double* _A_data = &A[0];
    double* _B_data = &B[0];
    for (int i = 0; i < N; i++)
        _A_data[i] = _B_data[i];
}
```

図 4 関数 bar のオプティマイズ結果
Fig. 4 Output of the Optimizer (bar).

```
SynObj* optimizeSimpleArray(SynObj* src) {
    SynObj* dst;
    OptSimpleArray osa;
    AnaSrcCode ascSimpleArray(osa);
    GenDstCode gdcSimpleArray(osa);
    ascSimpleArray.traverse(src);
    dst = gdcSimpleArray.traverse(src);
    return dst;
}
```

図 5 SimpleArray オプティマイザ：メインルーチン
Fig. 5 Main routine of the SimpleArray Optimizer.

最適化は 2 つのフェーズからなる．1 つめのフェーズではコードの走査による解析が行われる．**AnaSrcCode** クラスには，この解析のための走査アルゴリズムが記述されている．このクラスはメタライブラリにある走査アルゴリズムのクラステンプレートを継承して記述する．

解析は，解析対象となる構文木を引数に **traverse** メソッドを呼ぶことで開始される．この解析では各ブロックに含まれる関数呼び出しを調べ，ブロックごとに最適化を行うか否かを決定する．解析結果は **osa** オブジェクトに集められる．**osa** オブジェクトには集めた解析結果を次のフェーズに引き渡す役割がある．ここでは最適化可能ブロックの集合が次のフェーズに渡される．

2 つめのフェーズでは最適化されたコードの生成が行われる．**GenDstCode** クラスには，このコード生成のための走査アルゴリズムが記述されている．このアルゴリズムは，もとの構文木の複製をつくる走査の拡張である．そこで，このクラスもまたメタライブラリにある構文木複製アルゴリズムのクラステンプレートを継承して記述する．

走査が最適化可能なメソッド呼び出し（最適化可能ブロックに含まれる **operator[]** の呼び出し）に到達した際には複製ではなくインライン展開が施される．このインライン展開は A-part，B-part に分けられる．A-part は展開対象となるメソッド呼び出しを含む最適化可能ブロックのうち，最も外側のブロックの先頭に挿入されるコードである．また，B-part は展開対象となるメソッド呼び出しと置換されるコードである．

☆ オプティマイザが「**init** メソッドの呼び出しが発生しない」ことをすでに知っている関数はこのような関数から除外する．このオプティマイザはそのような関数名を，あらかじめ登録する機能を持つ．

```
SynObj* inlineSimpleArray(GenDstCode* t,
            CompoundStmt* cs, FuncExpr* mc) {
  Identifier* id = genSym();
  cs->insertStmt(
    '{double* $id = &$expr[0];}.attach(
    id, getObjVar(mc))); // A-part
  VarName* vn = cs->newVarName(id);
  return '($expr[$expr]).attach(
    vn, t->traverse(mc->getParam(0))); // B-part
}
```

図6 SimpleArray オプティマイザ：operator[] の展開
Fig. 6 Translation of operator[] in the SimpleArray Optimizer.

図2から図4への変換における，式 A[i] の A-part, B-part を以下に示す．

A-part:

```
double* _A_data = &A[0];
```

B-part:

```
_A_data[i]
```

一時変数 _A_data の名前は，実際には「その時点で使用されていなかった新しいシンボル」が割り振られる．本稿では，アンダーラインで始まるシンボルは，処理系が自動生成するシンボルを表す．

この展開を施す部分のメタプログラムを図 6 に示した．関数 inlineSimpleArray は，走査アルゴリズムのオブジェクト，A-part を挿入するブロック（*compound-statement*）および展開対象となるメソッド呼び出し式を引数として受け，ブロックに A-part の挿入を施し，B-part を返す．

ここで '{}および '() 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．STC に対する attach メソッドは STC の中に書かれた $expr（*expression*），$id（*identifier*）などを引数に置き換えた構文木を返す．STC が '{}で表されているときは *compound-statement* を返し，'() で表されているときは *expression* を返す．

関数 genSym は未使用のシンボルを生成し，それを指す *identifier* オブジェクトを返す．関数 getObjVar は引数として与えられたメソッド呼び出し式について，そのメソッドを呼ぶオブジェクトを表現する変数式を返す．CompoundStmt クラスの insertStmt メソッドは引数として与えられた *statement* を自ブロックの先頭に挿入する．また，newVarName メソッドは引数として与えられた *identifier* の名前を持つ変数式を生成する．FuncExpr クラスの getParam メソッドは関数呼び出し式の引数部を抽出するメソッドである．ここでは 1 番目の引数のみを抽出しているが，これは operator[] メソッドは引数を 1 つしか持たないからである．

```
daxy(double a, DistArray<double, Dist>& x,
             DistArray<double, Dist>& y) {
  for (int i = y.getLower();
       i < y.getUpper(); i++)
    y[i] = a * x[i];
}
```

図7 DistArray クラス使用例（関数 daxy）
Fig. 7 Example using DistArray (daxy).

5.2 分散配列クラスのオプティマイザ

次に DistArray クラスのオプティマイザについて述べる．DistArray クラスの operator[] メソッドは，与えられるインデックスの値域がコンパイル時に判明するならば，性能の良いコードに展開できる場合がある．値域から，参照するデータがローカルメモリ上にあると前もって判明しているならば，ローカルメモリを直接参照するコードに変換できるからである．

5.2.1 オプティマイザの仕様

DistArray クラスのオプティマイザでは，5.1.1 項で示した，SimpleArray クラスのオプティマイザが持つ制限に以下の制限を加える．ここで getLower, getUpper メソッドは，DistArray クラスがローカルメモリに所有する配列のインデックスの下限，上限を得るメソッドである．

(1) DistArray クラスのユーザは，プロセッサ 0 以外で init メソッドを呼ぶようなコードを記述してはならない☆．また，init メソッドが終了を待たずに他の（DistArray などのクラスを使用する）スレッドが動くコードを記述してはならない．このオプティマイザは，クラスの間違った使用や，複数のスレッドが与えるメモリへの副作用の自動検出は行わない．

(2) for 文のループ変数 i の値域指定の際に getLower, getUpper メソッドを使用した場合かつ，インデックスが i であることが，オプティマイザが行うパターンマッチで判定された場合のみ，インデックス値域による最適化が施される．なお，この場合 i は単調増加であるとオプティマイザは仮定する．

図 7 で示すプログラムは，このオプティマイザによって図 8 で示すプログラムへ変換される．ここで，for 文のループ変数 i の値域はオブジェクト y がローカルメモリに持つ要素のインデックス値域と一致するので，y の operator[] は，つねにローカルメモリをアクセスするように書き換えられる．一方，オブジェクト x については，オブジェクト y と，同じ型の分散

☆ このようなコードは init メソッドの間違った使用である．

```
daxy(double a, DistArray<double, Dist>& x,
                DistArray<double, Dist>& y) {
  int _yl = y.getLower();
  int _yu = y.getUpper();
  Dist* _y_dist = y.getLocalDist();
  Dist* _x_dist = x.getLocalDist();
  double* _y_data = y.getData();
  double* _x_data = x.getData();
  for (int i = _yl; i < _yu; i++)
    _y_data[i - _yl] = a *
    (_x_dist == _y_dist) ?
      _x_data[i - _yl] : x[i];
}
```

図8 関数daxyのオプティマイズ結果
Fig. 8 Output of the Optimizer (daxy).

```
bool AnaSrcCodeDA::traverse(ForStmt* obj) {
  if (`($expr = $expr.getLower())
    .match(obj->expr1, loopVar1, loopObj1) &&
    `($expr < $expr.getUpper())
    .match(obj->expr2, loopVar2, loopObj2) &&
    isVarEqual(loopVar1, loopVar2) &&
    isVarEqual(loopObj1, loopObj2) &&
    hasBaseType(loopObj1, "DistArray")) {
    travEnv->registLoopVar(loopVar1, loopObj1);
  }
  return AnaSrcCode::traverse(obj);
}
```

図9 DistArrayオプティマイザ：for文の解析
Fig. 9 Analysis of for statement in the DistArray Optimizer.

配置アルゴリズムを持つが，まったく同じ分割を行っているかどうかは，同じ分散配置のオブジェクトを使用しているかどうか実行時に判定しないと分からない．この判定が，for文のループボディ内にある条件分岐である．この条件分岐は `_x_dist`，`_y_dist` がループ不変式であることから，既存の最適化手法[1]であるhoisting☆を用いループの外に括り出すことができる．

5.2.2 オプティマイザの実装

DistArrayクラスに特化したオプティマイザは200行程度のC++コードである．オプティマイザの基本的な構成は5.1.2項で紹介したものとほぼ同じである．DistArrayのオプティマイザは，最初の（解析の）フェーズにおいて，for文におけるループ変数の値域に関する情報も集める．

図9に解析のための走査アルゴリズム`AnaSrcCodeDA`の一部を示した．`AnaSrcCodeDA`はSimpleArrayのときに使用した`AnaSrcCode`を継承している．図9に示すfor文についての走査メソッドは`AnaSrcCode`における定義を上書きする．このメソッドではfor文の3つのパラメータ式のうち最初の2つの式についてのパターンマッチを行う．STCの`match`メソッドは，1番目の引数で与えられる構文木とSTCとでパターンマッチを行う．その結果，マッチングに成功した場合は真を返し，失敗した場合は偽を返す．`match`メソッドの2番目以降の引数はマッチングの結果を返すために使用される．たとえば図9の最初の`match`メソッドでは，左辺の`$expr`にあてはまる構文木は`loopVar1`に代入され，右辺の`$expr`にあてはまる構文木は`loopVar2`に代入される．以下のfor文を解析対象とすると，

```
for(i = y.getLower();
    i < y.getUpper(); i++) { ... }
```

`loopVar1`には変数`i`が代入され，`loopObj1`には変数`y`が代入される．

`isVarEqual`は2つの引数がいずれも変数式で，同じ記憶域を持つかどうか判定する．`hasBaseType`は1番目の引数で与えられた変数式が2番目の引数で与えられた型の派生型であるか判定する．そして，`registLoopVar`メソッドは解析結果を集めるオブジェクトに`loopVar1`と`loopObj1`の組を登録する．上の例では`i`と`y`の組が登録される．

コード生成のフェーズではインデックス式の値域，分散配置アルゴリズムなどに応じ，`operator[]`メソッドが展開される．

for文のループボディにおける`operator[]`の展開を紹介する．さきの解析によってDistArrayオブジェクト`y`と，ループ変数`i`の組が登録されているならば，

```
y.getLower() <= i < y.getUpper()
```

が，成り立つと考える．そして`y[i]`は次のように展開される．

A-part:
```
int _yl = y.getLower();
double* _y_data = y.getData();
```
B-part:
```
_y_data[i - _yl]
```

ここで，DistArrayの要素型を仮に`double`にしたが，実際にはこの型はDistArrayクラステンプレートの型パラメータから決定される．

また，DistArrayオブジェクト`x`について，その`x`が`y`と同じ型の分散配置アルゴリズムであり，かつ，インデックス式が`i`であれば，`x`のR-valueを参照する式は次のように展開される．

A-part:
```
int _yl = y.getLower();
Dist* _x_dist = x.getLocalDist();
Dist* _y_dist = y.getLocalDist();
```

☆ ループ外側で分岐判定を行ってからどちらかの版のループを実行するようなコードを生成すること．

表 1 SimpleArray クラスを用いた実験
Table 1 Results using SimpleArray.

GNU G++	daxy	daxpy	dotpro
no-opt	790 (2.09)	1112 (1.54)	809 (1.27)
optimized	378 (1.00)	726 (1.00)	635 (1.00)
no-class	378 (1.00)	724 (1.00)	635 (1.00)

Sun C++	daxy	daxpy	dotpro
no-opt	986 (1.60)	1344 (1.36)	718 (1.06)
optimized	614 (1.00)	1000 (1.01)	672 (0.99)
no-class	617 (1.00)	987 (1.00)	677 (1.00)

単位：μ秒（no-class との比）

```
double* _x_data = x.getData();
```

B-part:

```
(_x_dist == _y_dist) ?
 _x_data[i - _yl] : x[i]
```

ここで，DistArray の分散配置アルゴリズムを仮に `Dist` にしたが，実際にはこの型は DistArray クラステンプレートの型パラメータから決定される．

6. 評価実験

最適化による効果を確認するため，いくつかの実験を行った．現在のところ SimpleArray クラスに特化したオプティマイザは実装されているが，分散配列クラスに特化したオプティマイザは実装されていない．そこで，後者についての実験は，ハンドコンパイルで行った．

まず最初に，SimpleArray クラスに特化した最適化を，daxy，daxpy，dotpro 計算それぞれについて施した結果を示す．ここで daxy 計算は，`double` 型のベクトル X，Y，および `double` 型の数値 a について，$Y = aX$ を計算する．daxpy 計算は $Y = Y + aX$ を計算し，dotpro 計算は内積 $X \cdot Y$ を求める．いずれも数値計算アプリケーションのなかで頻繁に必要とされる計算である．

表 1 に，それぞれの計算について，SimpleArray クラスを用いたが（我々の）最適化をかけなかった場合（no-opt），SimpleArray クラスを用い，最適化をかけた場合（optimized），SimpleArray クラスを用いなかった場合（no-class）それぞれの実行時間を示した．計測したマシンは Sun Ultra 1（167 MHz）で，使用したバックエンドコンパイラは GNU G++ 2.7.2 (sparc-sun-solaris2.5) および Sun C++ 4.1 である．いずれの実験においても，これらコンパイラに組込みのオプティマイザは使用している．表の各欄における数値はベクトル長が 10000 であるときの各計算の実行時間（単位は μ秒）を表す．

表 2 分散配列ライブラリを用いた実験（CG 法）
Table 2 Results using DistArray, CachedArray (NPB CG).

	16PE	32PE
no-opt	27.630 (1.72)	15.663 (1.77)
optimized	20.124 (1.26)	11.045 (1.25)
unrolled	16.886 (1.05)	9.609 (1.09)
no-class	16.030 (1.00)	8.840 (1.00)

単位：秒（no-class との比）

SimpleArray クラスを用いることは，実行時間を最悪の場合，2 倍程度遅くしていたが，最適化を施した結果，このクラスを用いることの不利益がまったくないコードが生成されていることが，実験結果からも読みとれる．

表 2 では，より現実的なアプリケーション，NAS Parallel Benchmark[2] の CG 法プログラム（クラス A）を用いて行った実験結果である．計測は RWC ワークステーションクラスタ（36 台の Sun SS20（75 MHz）を Myrinet で結合）[10]☆で行った．使用したバックエンドコンパイラは GNU G++ 2.7.2 (sparc-sun-sunos4.1) である．

no-class は分散配列ライブラリを用いずに MTTL ライブラリを直接呼び出すコーディングで作成した CG 法プログラムである．これに対し，no-opt，optimized，unrolled は 4 章で紹介した分散配列ライブラリ（DistArray および CachedArray）を用いて作成した CG 法プログラムである．計算対象となる疎行列の表現には DistArray を使用した．疎行列の，要素配列およびインデックス配列それぞれの DistArray で分散配置オブジェクトを共有している．また，計算の途中結果を保持する行列の表現には CachedArray を使用した．他のプロセッサで作られた計算結果が頻繁にアクセスされるからである．ここでもすべての CachedArray で分散配置オブジェクトを共有している．

no-opt は，なんら最適化を施さなかった場合，optimized は本稿で紹介した最適化を施した場合，unrolled は，さらに loop unrolling[1] を施した場合の実験結果である（no-class も loop unrolling が施されている）．また，いずれの場合においても loop unswitching が施されている．CachedArray に対する最適化には SimpleArray に対する最適化手法がそのまま適用された．表の各欄の数値は実行時間（単位は秒）を表す．

分散配列ライブラリを使用した場合，使用しなかった場合に比べその実行時間は最悪で 1.8 倍程度に増加

☆ http://www.rwcp.or.jp/lab/pdslab/

していたが，我々の最適化を施すことによって，その増加は1.1倍以下におさえられることが分かった．この1.1倍の増加原因は，最適化を施されたコードに現れるA-partであると予想される．

7. 関連研究

7.1 SUIF

SUIF[8]は，抽象構文木のフォーマットであるSUIFをベースに，自動並列化等のオプティマイザについて研究を行っているプロジェクトである．SUIFでは，解析の簡略化のために構文要素の表現を限られたものにしている．SUIFでは抽象構文木をBlock, If, Loopおよび Instructionで表現する．データ型にはC言語に用意されている型に加え，ARRAY型およびmodifierとしてCALL-BY-REF型が用意されている．

SUIFにはC++のクラスやテンプレートを表現する抽象構文要素がない．そのため，C++をSUIFに変換した場合，クラスは構造体で表現することになるだろう．その際，継承関係についての情報や，型パラメータについての情報など多くの情報が失われる．これは最適化のためのヒントを失うことに等しい．

7.2 OpenC++

OpenC++[3]はMPC++同様，コンパイル時メタレベル機能を持ったC++処理系である．OpenC++のメタレベル機能は，C++のクラスシステムの実装をライブラリプログラマに公開するものである．メタレベル機能を用いると，ライブラリとなる個々のクラスに特化したクラスシステムを適用することができる．特化したクラスシステムはライブラリプログラマが記述するメタクラスによって定義される．メタクラスにはメソッド呼び出し式の変換コードが記述される．このメタレベル機能を用い，OpenC++でもクラスに特化した最適化を行うことが可能であるが，OpenC++では変換対象となるメソッド呼び出し式の文脈を解析することができないため，本稿で紹介した最適化を記述することはできない．

7.3 pC++

pC++[5]は分散配列を実現するための特殊な言語仕様を持つC++である．要素型を型パラメータとする`SuperKernel`クラステンプレートのオブジェクトをコレクションと呼び，分散方式はコレクションのコンストラクタ引数で指定する．拡張された言語仕様により，コレクションは要素型のメンバを分散メモリ透過的にアクセスすることが可能である．しかしpC++では，(1) シングルスレッドのデータ並列モデルのみを想定しており，(2) クラス使用によるコストを解決する手法が提案されていないため，多くのアプリケーションにおいて，高性能なコードを得ることが難しい．

我々は，言語仕様を拡張することなく，ユーザに対して分散配列を実現しているだけでなく，Cプログラムで記述したコードとほぼ同等の性能を引き出すことを可能としている．

8. まとめ

クラスやテンプレートなどのC++のオブジェクト指向言語機能は，ソフトウェアの部品化，ライブラリ化をすすめ，再利用性や可搬性を高める．しかし，C++にはメモリのエイリアシングを許す言語仕様があるため，クラスのメンバ関数呼び出しコードの最適化が不可能である．そのため，クラスを多用したアプリケーションは，クラスを用いずに記述したアプリケーションと比較し，実行性能が劣るという問題があった．

我々は，コンパイル時メタレベル機能を持つC++処理系MPC++のもと，クラスライブラリに特化したオプティマイザを用いることで，この問題を回避した．

本稿では並列計算のための分散配列クラスライブラリを例題に選び，このクラスライブラリに特化したオプティマイザの記述方法を提案した．オプティマイザはメタレベル機能を用いて記述され，コンパイラに組み込まれる．オプティマイザの作成はメタライブラリを用いた差分プログラミングで行うため容易である．ライブラリプログラマが実際にクラスごとに記述するのは，メソッド呼び出し相互関係などによる最適化の可否判断コードと，メソッドのインライン展開を施す際に使用する，展開コードの雛型のみである．

我々は最適化結果が期待どおりであることを確認するためにいくつかの評価実験を行った．特にNAS Parallel BenchmarkのCG法プログラムを用いた結果は，十分満足のできるものである．

本稿で提案されたオプティマイザのうち分散配列クラスについてのオプティマイザはいまだ実装されていない．これは，現時点においてMPC++がテンプレートの構文解析をサポートしていないからである．現在，我々はMPC++にテンプレート機能を実装中である．分散配列クラスについてのオプティマイザは，この完了を待って実装される予定である．

参考文献

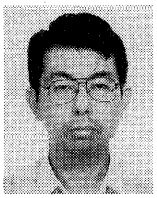
1) Bacon, D.F., Graham, S.L. and Sharp, O.J.: Compiler Transformations for High-Performance Computing, *ACM Computing Surveys*, Vol.26, No.4, pp.345–420 (1994).

2) Bailey, D.H., Barton, J.T., Lasinski, T.A. and Simon, H.D.: The NAS Parallel Benchmarks, NASA Technical Memorandum 103863, NASA Ames Research Center (1993).
3) Chiba, S.: A Study of a Compile-time Metaobject Protocol, PhD Thesis, Graduate School of Science, The University of Tokyo, Japan (1996).
4) Ellis, M.A. and Stroustrup, B.: *The Annotated C++ Reference Manual*, Addison-Wesley Publishing Company (1990).
5) Gannon, D. and Lee, J.K.: Object Oriented Parallelism: PC++ Ideas and Experiments, 並列処理シンポジウム JSPP'91, 情報処理学会 (1991).
6) Ishikawa, Y.: Multiple Threads Template Library, http://www.rwcp.or.jp/lab/mpslab/mpc++/mpc++.html.
7) 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).
8) Stanford Compiler Group: The SUIF Library, http://suif.stanford.edu/.
9) 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).
10) 堀 敦史, 手塚宏史, 石川 裕, 曽田哲之, 小中裕喜, 前田宗則：並列プログラム実行環境のワークステーションクラスタ上での実装, 並列処理シンポジウム JSPP'96, 情報処理学会 (1996).

(平成 9 年 11 月 4 日受付)
(平成 10 年 4 月 3 日採録)

高橋 俊行（正会員）

昭和 46 年生．平成 7 年東京理科大学大学院理工学研究科情報科学専攻修士課程修了．平成 10 年東京大学大学院理学系研究科情報科学専攻博士課程単位取得退学．同年技術研究組合新情報処理開発機構に入社．現在，同機構並列分散システムソフトウェアつくば研究室研究員．理学修士．並列・分散システム，オブジェクト指向言語，コンパイラ，リフレクション，音楽情報科学，等に興味を持つ．ソフトウェア科学会会員．

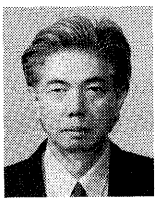


石川 裕（正会員）

昭和 35 年生．昭和 62 年慶應義塾大学大学院理工学研究科博士課程修了．工学博士．同年電子技術総合研究所入所．昭和 63～平成元年カーネギー・メロン大学客員研究員．平成 2 年日本ソフトウェア科学会高橋奨励賞を受賞．平成 5 年から技術研究組合新情報処理開発機構に出向．並列・分散システム，適応可能並列プログラミング言語/環境/処理系，リアルタイム処理等に興味を持つ．ソフトウェア科学会，ACM，IEEE 各会員．

佐藤 三久（正会員）

昭和 57 年東京大学理学部情報科学科卒業．昭和 61 年同大学院理学系研究科博士課程中退．同年新技術事業団後藤磁束量子情報プロジェクトに参加．平成 3 年，通産省電子技術総合研究所入所．平成 8 年より，新情報処理開発機構つくば研究センターに出向．現在，同機構並列分散システムパフォーマンス研究室室長．理学博士．並列処理アーキテクチャ，言語およびコンパイラ，計算機性能評価技術等の研究に従事．日本応用数理学会会員．



米澤 明憲（正会員）

昭和 22 年生．昭和 52 年 Ph.D. in Computer Science（MIT）．平成元年より東京大学理学部情報科学科教授．超並列・分散ソフトウェアアーキテクチャ等に興味を持つ．共著書「算法表現論」,「モデルと表現」(岩波書店)，編著書「ABCL: An Object-Oriented Concurrent System」(MIT Press) 等がある．平成 4～8 年ドイツ国立情報処理研究所（GMD）科学顧問，ACM Transaction on Programming Languages and Systems 副編集長，IEEE Concurrency Parallel & Distributed Technology および Computer 編集委員などを歴任，元日本ソフトウェア科学会理事長．