C++テンプレートを使ったデータ並列ライブラリの効率化手法

松田元彦† 石川 裕† 佐藤三久†

C言語のデータ並列拡張であるC*言語をC++言語の機能を利用することでコンパイラに遜色ないライブラリとして実装することを試みた．一般にアレイ演算等の低レベルなデータ並列ライブラリは実装の容易さや移植性の点で優れるが，演算に一時変数が必要になるなどコンパイラによる実装よりも性能が悪い．本研究ではコンパイラに対する性能差の主要因と考えられる演算中の一時変数除去と規則的通信パターン検出を可能にした．一時変数除去にはC++言語のテンプレート機能を使用するテンプレート・クロージャの手法を提案する．この手法では個々の演算は式を表現する木構造を生成するとともにその構造をタイプ情報に反映させる．このタイプ情報によりコンパイラは式全体の値を効率良く評価するコードを生成可能である．規則的通信パターン検出には規則的パターンを表す式に固有のタイプを与えることで，C++言語のタイプ・システムによる検出を可能にした．実装したライブラリをCM-5上で3つの例題に対して性能比較した結果，C*コンパイラに対して同等か1.5倍程度の実行時間に抑えることが可能となった．

Efficient C++ Data-parallel Library Utilizing C++ Templates

MOTOHIKO MATSUDA,† YUTAKA ISHIKAWA† and MITSUHISA SATO†

This paper describes a C++ template library which implements data-parallel extended C language C* efficiently using the C++ template and type-dispatch features. In general, programs written in low level array library tend to be inefficient compared to code generated by compilers. The proposed library deals with this problem from two efficiency issues: elimination of temporaries in expression evaluation, and detection of regular communication patterns such as cshift in communication expressions. Temporaries are eliminated by evaluating an expression as a whole by creating a tree-like structure representing it. We propose a C++ template technique which enables this evaluation being expanded at compile time. Regular communication patterns are detected by assigning particular types to expressions of potentially regular patterns. We implemented the library that imitates C* syntax/semantics and compared the performance to the C* compiler on three examples. The library performance is improved to about the same or 1.5 times slower than the C* compiler on the CM-5.

1. はじめに

C++言語ではデータ並列計算をSPMD処理により実装するライブラリとして提供することができる．このとき，演算子オーバロードの機能により `+` や `*` 等を使用した自明な記述をとることが可能である．たとえば，既存のデータ並列C言語であるC*[3]やNCX[12]といった言語にシンタックス上近いものを提供することができる．また，デファクト標準を狙うHPC++(High Performance C++)[9]に含まれるアレイ処理はC++のシンタックスの範疇にあるので，ライブラリとして提供可能である．

このようなライブラリによる実装は，通常の言語処理系に変更を加える必要がなく移植性が高いといった点でコンパイラに比べて有利である．しかし一般に要素ごとの加算や乗算といったプリミティブによる低レベルのアレイ処理を，ライブラリにより実装したのでは性能が悪い．たとえば3つのアレイ `A`, `B`, `C` を要素ごとに加算する場合 `A + (B + C)` を考える．これはライブラリでは2項演算子ごとに別々の2つのループにより処理される．一方，コンパイラではプログラム・テキストを調べることでこれを1つのループとして処理することが可能である．アレイ処理を提供するライブラリではこのようなオーバヘッドが問題となる．

そこで本研究ではライブラリである利点を生かしたまま効率良く実装するため，C++のテンプレートや多相型によるディスパッチ機能を利用することにより非効率部分を改善する．データ並列計算のモデルやシンタックスは既存の言語から借りることとしたが，本研

† 新情報処理開発機構
Real World Computing Partnership

究では比較の便からC*を選んだ.

一般にデータ並列計算の実行は仮想プロセッサのエミュレーションという方法をとる[4]. これは各アレイ要素に個別の仮想プロセッサが関連付けられており, アレイ要素への演算はその各仮想プロセッサが独立に計算するものと考える. 実際の処理は1つの物理プロセッサが複数の仮想プロセッサの働きをエミュレーションする. つまりデータ並列処理の実装はエミュレーションされる仮想プロセッサ数にあたるデータに対する処理であり, これは大きなベクタに対する処理と考えることができる.

ライブラリとコンパイラで性能差を生じる主な原因は次の2点と考えられる:

- 演算子ごとのループと一時変数の導入

```
X = A + (B + C);
```

C++では演算子をオーバロードする場合適用ごとに結果を生成しなければならない. さらに演算が結果を返すためには, 明示的に一時変数を導入しそこに中間結果を生成するように記述する必要がある. また演算子ごとに結果を生成するために演算子と同数のエミュレーション・ループが必要になる.

- 規則的な通信パターン等の検出

```
X = [pcoord(0) + 1] A;
```

これはC*における通信の記述であり, アレイの添字が1だけ大きい要素をとってくるシフト通信である. ここで **pcoord()** は仮想プロセッサのインデックスを返す関数であるので, この場合要素ごとにソース位置が指定されていることになる. ライブラリのようにプログラム・テキストを調べることのできない実装では個々の要素に関して送り先を決定しなければならない. 一方コンパイラであればこのような簡単なパターンを発見し効率の良い規則的通信コードを生成することが可能である.

一時変数の存在はコピーのための記憶域アロケーションと余分な代入の実行という非効率をともなう. これはレジスタやキャッシュの利用に悪影響を与える. また大きなアレイを扱う場合, コピーの存在は余分なメモリを使用し仮想記憶等の性能悪化をまねく. 演算子オーバロードに対する一時変数の発生はC++言語の設計時点から問題として取り上げられているが, 言語設計としての解決は与えられていない[2].

式評価の最適化に関して, ステートメント間に渡るものや共通部分式の除去などは本研究の対象外である.

このあと2章で予備知識としてANSI C++の機能であるテンプレートとSTLの関数オブジェクトについて簡単に説明する. その後, 3章でライブラリで実装した一時変数の除去について, 4章で通信パターンの検出について述べる. 5章ではライブラリの主な構文について説明し, 6章でCM-5上での性能評価について述べる. 7章では関連研究に簡単に触れる.

2. C++テンプレートと関数オブジェクト

この章ではC++言語のテンプレート機能と, STL (Standard Template Library)[8]で定義される関数オブジェクトについて概略説明を行う. これらはANSI C++標準化ドラフト[1]に記述されている.

2.1 C++テンプレート

テンプレートはC++言語にパラメータ化タイプを導入するもので, クラスに対するものと関数に対するものがある. それぞれクラス・テンプレート, 関数テンプレートという. **図1**にテンプレートの例を示す.

vec はタイプ引数 **T** の要素を10個持つベクタを定義している. テンプレートはタイプ引数をあるタイプでインスタンス化して使用する. 例では, タイプ引数 **T** を **double** 型にインスタンス化している. ここで **vec<double>** は, タイプ引数 **T** が **double** に置き換えられたコードに展開されていると考えられる.

関数テンプレートも同様であり, 図1の **add1** はどのようなタイプの引数に対しても1を加算する関数を定義している. 関数テンプレートのインスタンス化の場合は, コンパイラが引数からタイプ情報を取り出すことができるので, 明示的にタイプ引数を書く必要がなく **add1(v)** と書くことができる.

テンプレートに対してはインスタンス化されたタイプごとにコードが生成されるため, テンプレート化による実行時オーバヘッドはない.

```
/* クラス・テンプレートの例 */
/* 要素のタイプをタイプ引数とするベクタと
   double 型を要素に持つベクタの宣言 */
template <class T>
class vec {
    T v[10];  // ベクタ要素
};
vec<double> x;

/* 関数テンプレートの例 */
/* 引数タイプによらず1を加える関数テンプレートと
   double 型に対する関数テンプレートの適用 */
template <class T>
T add1 (T x) {
    return x + 1;
}
double v = 3.0;
v = add1<double>(v);
```

図1 クラス・テンプレートと関数テンプレートの例
Fig. 1 Examples of class template and function template.

2.2 STL 関数オブジェクト

STL[8]はリストやベクタといった「列」データ構造を操作するためのジェネリックな枠組みであり，container，iterator，algorithm 等の構成要素からなる．container は列のインタフェースの抽象化，iterator は container 中の要素をたどるためのポインタの抽象化である．algorithm はソート等の列に対する基本的な操作を定義している．

さらに STL では**関数オブジェクト**と呼ばれる演算子相当のものを定義する．関数オブジェクトは適用 `operator()` をメンバ関数に持つクラスのインスタンスと定義される．関数オブジェクトには適用 `operator()` が定義されるので，関数へのポインタと同様に振る舞うことができる．

図 2 は関数オブジェクトの例である．コンストラクタ呼び出し `op_add_n(3)` は，$\lambda x.(x+3)$ という関数閉包に相当する関数オブジェクトを作る．適用 `op(10)` は関数オブジェクトに対する単項演算 `operator()` の呼び出しであり，インライン展開され次のようになる：

```
x = 10 + op.N; // この場合 op.N の値は 3
```

ここで関数オブジェクトの利用はインラインに展開されるので，関数へのポインタの場合と異なり関数呼び出し等のオーバヘッドはない．

STL ではこのオーバヘッドのない関数オブジェクトを使う高階関数によって，インタフェースを抽象化する．高階関数には関数テンプレートを使用し，関数引数に関数オブジェクトを使用する．このような高階関数では，引数である関数オブジェクトのタイプ情報が高階関数の定義にまで伝播するので，関数オブジェクト呼び出しをコンパイル時にインライン展開することができる．

以上のように STL は列の抽象化とともに，呼び出しがインラインに展開される関数オブジェクトを定義する枠組みを与えている．

```
/* 定数を加える関数オブジェクト */
class op_add_n {
    int N;
    op_add_n(int n) {N = n;} // コンストラクタ
    int operator()(int x) { return x + N; }
};

/* 関数オブジェクトの使用例 */
op_add_n op = op_add_n(3);
x = op(10);  // x は 10 + 3 になる
```

図 2 関数オブジェクトの例
Fig. 2 Example of function object.

3. 一時変数の除去

STL の関数オブジェクトを使った関数適用は静的なタイプ情報によりコンパイル時に展開されるのでオーバヘッドは微少である．この関数オブジェクトを利用して一時変数を除去する．

C++では演算に対して演算の評価結果を返す代わりに演算を含む式の構造を保持したオブジェクトを生成することができる．このようなオブジェクトを使い評価を遅延することで，複数の演算を一度に処理し一時変数を不要にすることが可能である[5]．ここでこの遅延に関数オブジェクトを利用することができれば処理オーバヘッドをほとんどなくすことができる．

以下，一般的なライブラリ実装における一時変数の存在についての説明に続いて，関数オブジェクトを使用することでオーバヘッドなく複数の演算を一度に処理することを可能にする手法について説明する．

3.1 エミュレーション・ループと一時変数

仮想プロセッサのエミュレーション・ループを実装する場合物理プロセッサは複数の仮想プロセッサのデータを持つが，それはベクタとして表すことができる．エミュレーション・ループはこのベクタへの逐次処理である．一時変数の問題はこの逐次処理に関して発生する問題であるので，単一の物理プロセッサ上での処理に注目して説明する．ここでは，並列データ変数 `A` について個々の物理プロセッサに割り当てられたデータセットを `A[i]` で参照する．

図 3 に一般的なアレイ・クラスの定義を示す．ここでは並列データ型を `PARRAY<double>` 等と表記する．`PARRAY` は要素の型をタイプ引数とするクラス・テンプレートである．加算演算子は一時変数に加算結果を代入し，それを返すようにオーバロードされている．

次に，`PARRAY<double>` 型の変数に対する加算に続く代入という式について考える：

```
X = A + (B + C);
```

図 4 に一般のアレイ・クラス実装における加算の模式的な処理を示す．ここで `t1`，`t2` は一時変数である．アグレッシブな最適化コンパイラならループ・フュージョン等によりこれら一時変数を除去可能かもしれないが，一般には除去されないことが多い．

3.2 テンプレート・クロージャ

式評価を表現する関数閉包を作ることができれば，実際の値が必要になる時点（代入やキャスト等）まで式評価は遅延することができる．我々は複数の演算子を含む式に対して遅延を行うため関数オブジェクトを利用するが，それを**テンプレート・クロージャ**と呼ぶ．

```
/* タイプTの要素を持つアレイ・クラスの定義 */
template <class T>
class PARRAY {
    T *storage;
    T& operator[](int i)        // 要素への参照
        { return storage[i]; }
};

/* 加算演算子のオーバロード */
template <class T>
PARRAY<T>&
operator+(PARRAY<T>& X, PARRAY<T>& Y) {
    PARRAY<T> t; // 一時変数
    for ( … i … )
        t[i] = X[i] + Y[i];
    return t;
}

/* 代入演算子のオーバロード */
template <class T>
PARRAY<T>&
PARRAY<T>::operator=(PARRAY<T>& X) {
    for ( … i … )
        (*this)[i] = X[i];
    return *this;
}
```

図3 一般的なアレイ・クラスの定義
Fig. 3 Ordinary definition of an array class.

```
/* 加算に続く代入処理 */
PARRAY<double> X, A, B, C;
X = A + (B + C);

/* 上記式の模式的な処理 */
PARRAY<double> t1, t2; // 一時変数
for ( … i … )
    t1[i] = B[i] + C[i];
for ( … i … )
    t2[i] = A[i] + t1[i];
for ( … i … )
    X[i] = t2[i];
```

図4 一般的なアレイ・クラスによる加算の処理
Fig. 4 Addition of arrays by an ordinary array class.

テンプレート・クロージャは演算オペランドの型をタイプ引数とするように関数オブジェクトを拡張したものである．このことによりオペランドの型がタイプ引数としてテンプレート・クロージャ自身のタイプ情報に反映される．

図5に加算演算子を遅延するためのテンプレート・クロージャの定義を示す．加算演算子はテンプレート・クロージャを返すようにオーバロードされ，代入演算子は一般の場合と同様に要素ごとの代入として定義される．

図6に式評価 **X = A + (B + C)** に対して生成されるテンプレート・クロージャを示す．変数 **X, A, B, C** はすべて **PARRAY<double>** 型とする．部分式 **B + C** に対してクロージャ*E0*が，**A + (B + C)** に対して*E1*が生成される．

ここで演算のネストに従って，クロージャのタイプ情報がパラメータ化タイプとしてネストしていることに注意してほしい．つまり，各クロージャはそのタイプ情報として結果のタイプだけでなく，部分式のツリー構造を持っていることになる．テンプレート・クロージャに対する処理は，そのすべての部分式のタイプ情

```
/* 加算を遅延するクロージャ */
template <class EXPR1, class EXPR2>
class expr_add {
    EXPR1& L;  // 左オペランド
    EXPR2& R;  // 右オペランド
    typedef EXPR1::T T;
    T operator[](int i)
        {return L[i] + R[i];}
};

/* 加算演算子のオーバロード */
template <class EXPR1, class EXPR2>
expr_add<EXPR1, EXPR2>
operator+(EXPR1 e1, EXPR2 e2) {
    expr_add<EXPR1, EXPR2> e;
    e.L = e1; e.R = e2;
    return e;
}

/* 代入演算子のオーバロード */
template <class T> template <class EXPR1>
PARRAY<T>&
PARRAY<T>::operator=(EXPR1 E) {
    for ( … i … )
        (*this)[i] = E[i];
    return *this;
}
```

図5 加算を遅延するテンプレート・クロージャの定義
Fig. 5 Definition of a template closure for addition.

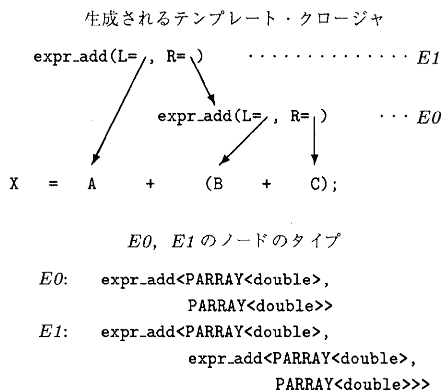


図6 加算に対して生成されるクロージャ
Fig. 6 Template closures generated by an expression.

報をコンパイル時に得ることができる．つまり，式評価の遅延に相当するオブジェクトとしてテンプレート・クロージャを使用した場合，式全体として計算するライブラリ呼び出しをコンパイル時に展開できることになる．

3.3 テンプレート・クロージャによる一時変数除去

次に図6の具体例を使って，テンプレート・クロージャにより式評価を遅延し一時変数を除去する方法について説明する．

まず，代入演算は要素ごとの代入であると定義される：

```
for ( … i … )
    (*this)[i] = E[i];
```

ここで E はテンプレート・クロージャ*E1*である．クロージャに対する [] 演算子の適用は左右オペランドに対する適用の和と定義されるので，これは次のように展開される：

```
for ( … i … )
    (*this)[i] = E.L[i] + E.R[i];
```

さらに E.R[i] はクロージャ*E0*であり，このタイプも分かっているので続けて展開される．結局，X への代入は次のように展開される：

```
for ( … i … )
    (*this)[i] = E.L[i] + ((E.R).L[i]
                          + (E.R).R[i]);
```

ここで元の式と比べると同じ計算になっていることが分かる．この展開から分かるように一時変数が除去されるとともに，演算に対するループが1つとなるループ・フュージョンの効果も得られている．この展開はタイプ情報よりコンパイル時に行われるので実行にオーバヘッドはない．

3.4 一時変数除去の効果

表1 に一時変数除去の効果を示す．ここでは2つの一時変数について保持場所のアロケーションとコピーが取り除かれている．カッコ内は人手で展開した最適なコーディングとの比である．比較は各物理プロセッサが1M個の仮想プロセッサをエミュレーションする場合を想定した．計算は次の式を使用し，これを逐次のC++コードとしてコーディングして比較した：

表1 一時変数除去の効果
Table 1 Effect of temporary elimination.

コード	実行時間
一時変数除去	0.60 sec (1.09)
一時変数あり	1.04 sec (1.89)
最適化したコード	0.55 sec (1.00)

（カッコ内は最適なコードとの比）

```
PARRAY<double> X, A, B, C;
X = A + (B + C);
```

この比較はSPARC Station 20/71（SunOS）上で行い，C++コンパイラにはGNU g++（コンパイル・オプション-O）を使用した．

この式では2つの一時変数が除去されているが，一時変数を除去したコードと最適なコードとの差はテンプレート・クロージャ内のポインタたどりだと考えられる．1つまり前節の展開式中の R や L といったメンバへの参照が余分なポインタ参照になっている．

4. 規則的通信パターンの検出

4.1 通信の構文

C*の通信構文はレフト・インデッキシングと呼ばれ，以下のように記述される：

```
X = [I]A;
```

この構文は C++にはないので，ライブラリ作成にあたってはメンバ関数のシンタックスで記述することにする：

```
X = A.at(I);
```

I は int 型アレイの式でも単なる int 型の式でもよい．int 型アレイによるインデックスの場合は並列データ参照であり，int 型によるインデックスの場合は要素への参照である．

次にこの通信式の意味を説明する．X, I, A はすべて並列データで，特に I は int 型のアレイであるとする．このとき，I でインデックスされる A の要素が X に代入される．この式の意味は模式的には次のようになる：

```
for ( … i … )
    X[i] = A[I[i]];
```

ここで X と I はアレイの形が同じ（シェイプが同じ）である必要がある．これは汎用のget-通信である．get-通信は個々の要素ごとにソース位置が指定される通信である．get-通信では送り先の決定にコストがかかり，さらに分散環境ではインデックスを与えるアレイも分散されているので実際のデータ通信以外に送り先情報交換のための通信も必要になる．

4.2 規則的通信のパターン

規則的通信パターンとして重要なのはcshift，eoshiftである．cshift，eoshift はメッシュに対するシフト通信のうち，メッシュ端が閉じているものと開いているものにあたる．

C*では cshift 等は関数 pcoord(*axis*) を使って記述される．pcoord(*axis*) は仮想プロセッサのインデックスを返す関数である．*axis* で示される次元のインデッ

クスを，0から（次元のサイズ）-1までの int 型アレイとして返す．A を1次元アレイでその大きさを N とすると，cshift，eoshift は次のように記述される：

```
X = A.at((pcoord(0) + 1) % N); //cshift
X = A.at(pcoord(0) + 1);       //eoshift
```

このままコード生成したのでは要素ごとにソースを指定するので効率が悪い get-通信である．しかし C*コンパイラは pcoord() を含む式の簡単なパターンを見つけ，効率の良い cshift，eoshift に相当するコードを生成することができる．

4.3 通信パターンの検出

C*コンパイラ同様の cshift，eoshift パターン検出を C++の多相型を使い実現する．pcoord() に int 型アレイではなく特別なタイプを与えることで，pcoord() を式のパターンとして使うことを可能にする．つまり，通信はこの pcoord() を含む式のタイプに従ってタイプ・ディスパッチすることで規則的な通信を呼び出すことが可能になる．また pcoord() の返すタイプにはキャストを定義しているので，アレイとしての値が必要になる場合には int 型アレイに変換される．以下では説明上一次元アレイの場合を想定するが，多次元の場合にも容易に拡張できる．

通信として見つけるべきパターンは，pcoord() を使った次の形である：

pcoord(0) + M

(pcoord(0) + M) % N

ここで M, Nは（並列データでない）整数とする．Mはシフトの大きさで，Nはラップ・アラウンドの大きさである．

pcoord() および上記の pcoord() を含む式に対して，MやNの情報を持った特別のクラスを与える：

```
class expr_pcoord1 { int M; };
class expr_pcoord2 { int M, N; };
```

つまり cshift 等を表すパターンである式が expr_pcoord1 等の型を持つように演算子を定義する．

図7 に上記のタイプに対する at() のオーバロードを示す．実際に cshift 等が使えるかは MやNの値に左右される．このチェックは実行時に行い，条件に合わない場合は汎用の get-通信を呼び出している．

```
/* cshift の表記 */
X = A.at((pcoord(0) + 1) % N);

/* cshift に対応する通信の定義 */
template <class T>
PARRAY<T>&
PARRAY<T>::at(expr_pcoord2 & e) {
    if ( cshift が使えるか条件チェック… )
        return _cshift(*this, e.M);
    else
        return _get(*this, (PARRAY<int>) e);
}
```

図7　cshift に対応する通信のオーバロード
Fig. 7　Overloading for cshift communication.

5. ライブラリの構文

通信構文については4章で述べたが，この章ではそれ以外の主なライブラリのシンタックスについて説明する．

5.1 変数宣言と簡単な式

C*では並列アレイのサイズ（つまり仮想プロセッサのサイズ）をシェイプと呼び，shape というタイプで宣言する．C*では $N \times N$ の並列アレイの宣言を次のように記述する：

```
shape [N][N]s0;
double:s0 X;
```

これに対し本ライブラリでは C*にならって次のように記述する：

```
shape_t *s0 = new shape_t(N, N);
PARRAY<double> X(s0);
```

演算子はオーバロードにより，C*同様の自明な記述になる：

```
PARRAY<double> X(s0), A(s0), B(s0), C(s0);
X = A + B * C;
```

5.2 where 構文

C*ではアクティブな仮想プロセッサのみを選択的に実行させるために where 等の構文が用意されている．これらはアクティブな仮想プロセッサのみが実行を行うようにするためのマスクを生成する．たとえば，where 文は次のように記述される：

```
where ( X > 0 ) { … }
```

この例では X の値が正である仮想プロセッサのみで，where 文の本体が実行される．

where 等のシンタックスは C++では定義できないので，C-プリプロセッサのマクロを使って見かけを繕った．where(…) は for 文に展開される．for 文に展開することにより，ブロック脱出時にマスクを元に戻す処理を行うことができるからである．この for 文はループするためのものではないので，一度で抜けるように条件を記述してある．

6. CM-5 上での評価

性能評価のため，CM-5 上で C*コンパイラと比較した．C++コンパイラは CM-5 のベクタユニットを使用できないので，C*においてもベクタユニットなしオ

```
ライブラリを使ったコード
#define CR      0.320
#define CI      0.043
PARRAY<double> zr(s), zi(s);
PARRAY<double> zrs(s), zis(s);
PARRAY<unsigned char> ittr(s);
...
int i;
for ( i = 0 ; i < RES ; i++ ) {
    where ( zrs + zis <= 4.0 ) {
        zrs = zr * zr;
        zis = zi * zi;
        zi  = zr * 2.0 * zi + CI;
        zr  = zrs - zis + CR;
        ittr = i;
    }
}
```

図8 ジュリア・セットの計算
Fig. 8 Calculation of Julia set.

表2 ジュリア・セット計算実行時間
Table 2 Execution time of calculation of Julia set.

処理系	実行時間
本ライブラリ	22.285 sec (1.59)
一時変数除去なし	50.726 sec (3.63)
C*コンパイラ	13.977 sec (1.00)

(カッコ内はC*との比)

プションを指定した．使用したCM-5は32プロセッサでC*のバージョンは7.2である．C++コンパイラにはGNU g++（コンパイル・オプション-O）を使用した．

本ライブラリはCM-5のメッセージパッシング用ライブラリCMMDを使用してSPMD実行で実装した．つまりプログラムはすべてのプロセッサで実行が開始され，その後はリダクションや通信といった同期点を除いては独立に処理される．並列アレイの分散としてはブロック–サイクリック分割をサポートするが，ここではC*との比較のためブロック分割を使用した．

プログラムとしては，ジュリア・セットの計算，マトリックスの積，エラトステネスの篩（ふるい）をとった．ジュリア・セットの計算とエラトステネスの篩はCM-5付属のC*の例題を使用した[10]．

計測時間はCM-5のタイマによるelapse timeである．

6.1 ジュリア・セットの計算

一時変数コピー除去の効果を見るために通信を含まないジュリア・セットの計算を選んだ．ジュリア・セットはフラクタルの一種である．図8にプログラムを示す．C*のプログラムは変数宣言のシンタックスを除けば同一であるので省略する．

```
ライブラリを使ったコード
shape_t *NN = new shape_t ...;
PARRAY<double> A (NN), B (NN), C (NN);
...
C = 0;
for ( i = 0 ; i < N ; i++ ) {
    C = C + (A * B);
    A = A.at(pcoord(NN,0), ((pcoord(NN,1)+1)%N));
    B = A.at(((pcoord(NN,0)+1)%N), pcoord(NN,1));
}

C*によるコード
shape [N][N]NN;
double:NN A, B, C;
...
C = 0;
for ( i = 0 ; i < N ; i++ ) {
    C = C + (A * B);
    A = [pcoord(0)][(pcoord(1)+1)%%dimof(NN,1)]A;
    B = [(pcoord(0)+1)%%dimof(NN,0)][pcoord(1)]B;
}
```

図9 シストリックなマトリックス積
Fig. 9 Systoric matrix multiply.

表3 マトリックス積の実行時間
Table 3 Execution time of systoric matrix multiply.

処理系	実行時間
本ライブラリ	3.869 sec (1.07)
一時変数除去なし	5.314 sec (1.47)
C*コンパイラ	3.623 sec (1.00)

(カッコ内はC*との比)

表2にC*とライブラリとの実行時間を示す．問題のサイズは512 × 512で計測した．時間計測はこのループ内のみでかつI/Oは省いた．

この問題に関してはステートメント間での最適化が効くため，C*コンパイラが有利になっている．しかし一時変数を除去しない実装に比較すると大きく改善されており，本手法に効果があることが分かる．

6.2 マトリックス積

規則的な通信に対する比較としてシストリックなマトリックス積の計算を選んだ．これはよく知られているように前処理として行/列を適当にずらせておけば，その後は一方を行方向に他方を列方向にずらしつつ要素を積和することでマトリックス積が得られるというアルゴリズムである．図9にプログラムを示す（前処理は含んでいない）．プログラム中の `(A * B)` は要素ごとの積でありマトリックス積でないことに注意．ライブラリの `pcoord()` はシェイプを第一引数にとるように若干変更している．

表3にそれぞれの実行時間を示す．マトリックスの大きさは256 × 256とした．時間計測に前処理部分は

```
      ライブラリを使ったコード
PARRAY<int> is_candidate (s);
...
PARRAY<int> pcoord0 = pcoord(s, 0);
do {
    where ( is_candidate ) {
        int p0 = reduce(min<int>(), pcoord0);
        where ( ! (pcoord0 % p0) )
            is_candidate = FALSE;
        (*is_prime_p).at(p0) = TRUE;
    }
} while (reduce(bitior<int>(), is_candidate));

      C*によるコード
bool:s is_candidate;
...
do {
    where ( is_candidate ) {
        int p0 = <?= pcoord(0);
        where ( ! (pcoord(0) % p0) )
            is_candidate = FALSE;
        [p0]*is_prime_p = TRUE;
    }
} while ( |= is_candidate );
```

図 10 エラトステネスの篩
Fig. 10 Sieve of Eratosthenes.

表 4 エラトステネスの篩実行時間
Table 4 Execution time of seive.

処理系	実行時間
本ライブラリ	58.332 sec (0.67)
C*コンパイラ	87.295 sec (1.00)

(カッコ内は C*との比)

含んでいない．ライブラリは C*コンパイラとほぼ同等性能が出ている．

6.3 エラトステネスの篩

図 10 は CM-5 付属の C*の例題からとったエラトステネスの篩（ふるい）である．リダクションとレフト・インデッキシングのシンタックスが異なっている．本ライブラリでは **pcoord()** の計算が遅いのであらかじめコピーをとってループの外に出した．一方 C*コンパイラでは **pcoord()** がループ内にあっても速度に違いはない．

表 4 に C*とライブラリとの実行時間を示す．64 K 個の整数に対して篩をかけた．

この問題に関しては今回の最適化は効果がないが実際には C*コンパイラよりも性能が良い実装になっている．これは C*の SIMD 実行と本ライブラリの SPMD 実行での差に由来すると考えられる．C*は SIMD 実行であり逐次部分は 1 つのプロセッサで実行し，並列部分のみプロセッサ・エレメントを起動する．C*では通信や制御構文だけでなく **where** 構文もプロセッサ・エレメントの実行が中断されるので，このプログラムでは数多くの同期点が入ってしまうことになる．

7. 関連研究

7.1 A++/P++

A++/P++は Fortran 90 ライクなアレイ記述を提供する C++ライブラリである[5]．A++は逐次版で P++は並列版であり，どちらも同一のインタフェースを与えている．

A++/P++では一時変数の問題への対処として複数の演算を一括処理するため，演算をノードとする式ツリーを実行時に生成する．式ツリーの形の判断を実行時に行い，その形に適当なルーチンを呼び出すことで演算を一括処理する．式ツリーの形に従ったルーチン群をあらかじめ用意しておかなければならないため，限られた式に対してのみ最適化が適用される．

7.2 AVTL

C++のテンプレートを使用したデータ並列ライブラリに AVTL[7]がある．AVTL はネストしたデータ並列計算も取り扱うことができる．AVTL は代入時のコピーを記憶域のシェアによって最適化している．記憶域はリファレンス・カウントで管理されている．

AVTL はベクタに対して要素ごとに演算を行うマップ処理を基本としている．つまり，ベクタ要素に対するマップ関数に関数引数として関数オブジェクトを渡すことでデータ並列計算を行う．これは STL のスタイルを直接データ並列計算に拡張したものといえる．この場合ユーザが明示的に関数オブジェクトを定義するので無駄な一時変数が現れることはない．ただし新たな関数オブジェクトの定義以外に関数合成の手段を与えないので簡単な式の場合も記述が煩雑になる．

7.3 Expression Template

C++のテンプレートを利用して，通常の算術式の記述を関数引数に使う手法がある．これは **x** を特別なタイプとして **(x + 3)** 等の記述を関数引数として使用しようとするものである．この応用としてここで述べたのと同様のアレイ演算中の一時変数除去について述べられている[11]．

7.4 CHAOS++

通信の効率化という面では，不規則な通信を扱うために Inspector/Executor モデルを使う CHAOS++ ライブラリがある[6]．Inspector/Executor モデルでは不規則な通信を扱うために通信パターンを通信スケジュールという形で保持しそれを繰り返し利用する．本研究での通信は，本来規則的な通信であるものがシンタックス上不規則な通信と区別不可能になっている

点を問題としているので目的が異なっている．

8. おわりに

C++言語では演算子オーバロード機能によってC*ライクな記述をライブラリとして提供することができる．さらにそこにC++テンプレート機能などによる最適化のテクニックを使うことで，データ並列言語コンパイラの性能に近づけることができることを示した．ここでの結果は，ライブラリ実装が予備的なのでコードのチューニングによりまだ性能改善できる余地が残っている．

このデータ並列ライブラリは通常のC++言語で記述されている．よってMPI等の通信ライブラリとともに使用することで，ワークステーション・クラスタや一般的な並列計算機上でデータ並列計算を行う場合に適用できる．さらに，ここで使ったテンプレート・クロージャの技法は関数の合成をコンパイル時にインライン展開可能にするので，一般のC++プログラムにも応用できると考えられる．

謝辞 本研究の機会を与えていただいた島田潤一研究所長，ならびに有益なご討論をいただいた並列分散システムパフォーマンス研究室と並列分散システムソフトウェア研究室のメンバに感謝いたします．

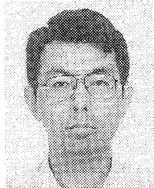
参考文献

1) ANSI: Working Paper for Draft Proposed International Standard for Information Systems – Programming Language C++ (1995).
2) Ellis, M.A. and Stroustrup, B.: *The Annotated C++ Reference Manual*, pp.299–303, §12.1c: Temporary Elimination, Addison-Wesley (1994).
3) Frankel, J.L.: A Reference Description of the C* Language, Thinking Machines Corporation, TR-253 (1991).
4) Hatcher, P.J. and Quinn, M.J.: *Data-Parallel Programming on MIMD Computers*, MIT Press (1991).
5) Parsons, R. and Quinlan, D.: A++/P++ Array Classes for Architecture Independent Finite Difference Computations, *Proc. 2nd Annual Object-oriented Numerics Conference* (1994).
6) Chang, C., Sussman, A., and Saltz, J.: CHAOS++, *Parallel Programming Using C++*, MIT Press (1996).
7) Sheffler, T.J.: A Portable MPI-based Parallel Vector Template Library, RIACS TR-95.04 (1995).
8) Stepanov, A. and Lee, M.: The Standard Template Library, HP Technical Report, HPL-94-34 (1994).
9) The HPC++ Working Group: HPC++ White Papers, Center for Reseach on Parallel Computation, Technical Report, TR-95633 (1995).
10) Thinking Machines Corporation: Getting Started in C* (1993).
11) Veldhuizen, T.: Expression Templates, *C++ Report*, Vol.7, No.5, pp.26–31 (1995).
12) 湯淺太一，貴島寿郎，小西 浩：データ並列計算のための拡張C言語NCX，電子情報通信学会論文誌，Vol.J78-D-1, No.2, pp.200–209 (1995).

(平成8年9月18日受付)
(平成9年7月1日採録)

松田 元彦（正会員）

1988年京都大学理学部卒業．同年住友金属工業（株）入社．1995年より新情報処理開発機構に出向．データ並列計算等の研究に従事．



石川 裕

1987年慶應義塾大学理工学部電気工学科博士課程修了．工学博士．同年電子技術総合研究所入所．1988～1989年カーネギー・メロン大学客員研究員．1990年日本ソフトウエア科学会高橋奨励賞を授賞．1993年から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．ソフトウエア科学会，ACM，IEEE各会員．

佐藤 三久（正会員）

1959年生．1982年東京大学理学部情報科学科卒業．1986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．1991年，通産省電子技術総合研究所入所．1996年より，新情報処理開発機構つくば研究センタに出向．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．情報処理学会，日本応用数理学会会員．