最適化機構を持つ C++ 並列スケルトンライブラリ

明石 良樹　松崎 公紀　岩崎 英哉　筧 一彦　胡 振江

並列プログラムを作成する難しさを緩和するため，頻繁に用いられる並列処理のパターンをあらかじめ実装しておき，そのパターンの組み合わせでプログラムを作成する「スケルトン並列プログラミング」が考案されている．プログラマは並列スケルトンと呼ばれる関数を組み合わせることによって，並列プログラムを容易に記述することができるようになる．しかし，並列スケルトンで書かれたプログラムは，必ずしも実行効率が良くないという問題点がある．本研究では，BMF という計算モデルに基づいた並列スケルトンライブラリを C++上に実装した．その上で，融合変換と呼ばれる，複数の関数呼出しを単一の関数呼出しに融合する手法を適用し，並列スケルトンで書かれたプログラムを最適化する機構を実現した．本論文では，並列スケルトンライブラリの実装を提案し，最適化の効果について報告する．

Skeletal parallel programming enables programmers to build a parallel program from ready-made components called skeletons (parallel primitives) for which efficient implementations are known to exist, making both the parallel program development and the parallelization process easier. Parallel programs in terms of skeletons are, however, not always efficient, because intermediate data structures which do not appear in the final result may be produced and passed between skeletons. To overcome this problem and make the skeletal parallel programming more practical, this paper proposes a new parallel skeleton library in C++. This system have an optimization mechanism which transforms successive calls of parallel skeletons into a single function call with the help of fusion transformation. This paper describes the implementation of the skeleton library and reports the effects of the optimization.

1 はじめに

大規模な問題を解く上で，並列計算は非常に大きな役割を果たしてきた．しかし，計算機間の通信や同期制御などを行わなければならないため，並列プログラミングは非常に難しい．

この問題を解決するため，頻繁に用いられる並列処理のパターンをあらかじめ実装しておき，そのパターンの組み合わせでプログラムを作成する「スケルトン並列プログラミング」[6] が考案されている．プログラマは並列スケルトンと呼ばれる関数を組み合わせることによって，並列プログラムを容易に記述することができるようになる．並列プログラミングの知識がない人でも，逐次プログラムを書く感覚で並列プログラムを記述できる．

しかし，並列スケルトンを用いて書かれたプログラ

A Parallel Skeleton Library in C++ with Optimization Mechanism.

Yoshiki Akashi, 電気通信大学大学院電気通信学研究科, Graduate School of Electro-Communications, The University of Electro-Communications.

Kiminori Matsuzaki, Kazuhiko Kakehi, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo.

Hideya Iwasaki, 電気通信大学情報工学科, Department of Computer Science, The University of Electro-Communications.

Zhenjiang Hu, 東京大学大学院情報理工学系研究科, Graduate School of Information Science and Technology, The University of Tokyo. 科学技術振興機構さきがけ研究 21, PRESTO 21, Japan Science and Technology Agency.

コンピュータソフトウェア, Vol.22, No.4 (2005), pp.78–83.
[小論文] 2005 年 2 月 18 日受付.

ムは必ずしも実行効率が良くないという問題がある．並列スケルトンひとつひとつが効率の良いものであっても，それらを組み合わせた全体は必ずしも効率の良いものになるとは限らないからである．その原因の一つに，ある並列スケルトンによって計算された値が，別の並列スケルトンへの入力として使われるだけで最終結果に現れないような場合に，はじめの並列スケルトンによる値の計算コストが結果的に高くついてしまうことがあげられる．この問題は並列スケルトンの性質上避けられないものであるが，これを改善する機構は従来の並列スケルトンシステムではほとんど考慮されていなかった．本研究では以上の問題を解決し，実用に耐えうる並列スケルトンライブラリを実現することを目指す．

本研究では，BMF [3] という計算モデルに基づいて並列スケルトンライブラリを実現した．本ライブラリでは，リストや木などの代表的なデータ構造を扱うことができる．一般に広く使われている C++上に実装し，構文の拡張を行わずに純粋な C++のライブラリとして提供することで，一般ユーザにとって利用しやすい，実用的な並列スケルトンプログラミングシステムを目指している．

さらに，本ライブラリを用いて書かれたプログラムの実行効率を改善するために，ソースコードを変換して最適化を行う機構を実現した．本機構では，融合変換と呼ばれる，複数の関数呼出しを単一の関数呼出しに融合する手法を用いて，並列スケルトン間で受け渡される中間データを生成しないようにした．

本論文では，並列スケルトンライブラリの実装を提案し，最適化の効果について述べる．

2 並列スケルトン

本研究では，BMF というデータ並列プログラミングモデルに基づいて並列スケルトンを設計した．BMF を用いると，プログラムとその変換を簡潔に記述することができる．

2.1 表記法

本論文では，並列スケルトンを表現するために，関数プログラム的な表記法を用いる．ここで，本論文で使う表記法についてまとめておく．

- 関数適用は，関数と引数との間に空白を入れることによって示し，他の演算子と比較して最も結合の優先順位が高いものとする．
- 関数合成は $\circ$ で表す．すなわち，$(f \circ g)\ x = f\ (g\ x)$ である．
- 二項演算子 $\oplus$ は，$a \oplus b = (a\,\oplus)\ b = (\oplus\, b)\ a = (\oplus)\ a\ b$ といったように関数として扱うことができる．
- 空リストは $[\,]$，a を要素に持つリストは $[a]$ で表す．また，$[\cdot]$ を a から $[a]$ を生成する関数とする．
- $x +\!\!+\, y$ はリスト x，y を連結したリストであり，$[1] +\!\!+ [2] +\!\!+ [3]$ を $[1,2,3]$ と略して表記する．また，$[a] +\!\!+\, x$ を $a : x$ と書くこともある．

2.2 並列スケルトン

BMF では，並列スケルトンに適した高階関数をいくつか定義している．重要な高階関数は map，reduce，scan，zip の 4 つである．

map はリストの各要素に関数を適用するスケルトンである．f を関数とすると，定義は次のようになる．

$$\mathsf{map}\ f\ [x_1, x_2, \ldots, x_n] = [f\ x_1, f\ x_2, \ldots, f\ x_n]$$

reduce はリストの各要素を結合的な二項演算子で結合した値を返すスケルトンである．$\oplus$ を二項演算子とすると，定義は次のようになる．

$$\mathsf{reduce}\ (\oplus)\ [x_1, x_2, \ldots, x_n] = x_1 \oplus x_2 \oplus \cdots \oplus x_n$$

scan は reduce の全ての途中結果を返すスケルトンである．$\oplus$ を結合的な二項演算子，$\oplus$ の単位元を e とすると，定義は次のようになる．

$$\begin{aligned}\mathsf{scan}\ (\oplus)\ &[x_1, x_2, \ldots, x_n] \\ &= [e, e \oplus x_1, \cdots, e \oplus x_1 \oplus \cdots \oplus x_n]\end{aligned}$$

zip は 2 つのリストの要素を組にして，1 つのリストにするスケルトンである．定義は次のようになる．

$$\begin{aligned}\mathsf{zip}\ &[x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] \\ &= [(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)]\end{aligned}$$

上の 4 つの他，Hu ら [7] [9] は accumulate という，リストに再帰関数を適用する並列スケルトンを提案

している．g，p，q を関数とし，$\oplus$，$\otimes$ を二項演算子とすると，定義は次のようになる．

$$\begin{aligned}&\mathsf{accumulate}\ [\,]\ e = g\ e\\&\mathsf{accumulate}\ (a:x)\ e\\&\qquad = p\ (a,e)\ \oplus\ \mathsf{accumulate}\ x\ (e\otimes q\ a)\end{aligned}$$

以降，本論文では accumulate を $[\![g,(p,\oplus),(q,\otimes)]\!]$ と書く．

3 並列スケルトンライブラリ

本研究では，C++と MPICH という分散並列計算用のライブラリを用いて，並列スケルトンライブラリを実装した．本ライブラリでは，いかにユーザに並列処理のための操作を隠蔽し，効率の良い並列プログラムが書けるか，といった点に注目した．本ライブラリはリストや木などのデータ構造をサポートするが，本論文ではリストの並列スケルトンに焦点をあてて，本ライブラリの特徴を説明する．

3.1 データの実現

分散並列計算では，リストなどのデータを各計算機に均等に分散して，データに対する操作を同時に行うことで，並列処理を実現している．本ライブラリでは，リストを `dist_array` クラスとして実現し，データの分散や収集といった，分散並列特有の操作を隠蔽している．ユーザはデータが分散されていることを意識することなく，逐次プログラムにおけるリストと同じ感覚で分散されたリストを扱うことができる．

例えば，整数の配列 `array` の要素を各計算機に分散させてリストを生成する場合，次のように記述する．

```
dist_array<int> *as
  = new dist_array<int>(array, size);
```

この 1 文だけで，`array` の要素を均等に分割し，各計算機に分散させることができる．このようにして生成されたリストに対して，並列スケルトンを呼び出すことによって，並列プログラムを記述していく．

3.2 並列スケルトンの実現

2.2 節で述べた並列スケルトンは，`dist_array` クラスのメンバ関数として実装した．各並列スケルトンは，ユーザの利便性を考えて，様々な種類のものを用意している．例えば map の場合，本ライブラリでは，

1. 計算結果のリストを戻り値として返すもの
2. 引数に与えたリストに計算結果を代入するもの
3. 呼出し元のインスタンスに計算結果を上書きするもの

を用意している．`A` を呼出し元のインスタンスが保持するデータの型とすると，map は図 1 のようになる．図 1 のメンバ関数は，上から順に 1，2，3 に対応している．例えば，`map_ow` は引数に関数ポインタを取り，その関数をリストの全要素に適用し，呼出し元のインスタンスに計算結果を上書きするものである．`map_ow` を使う場合，ユーザは適用したい関数を作成し，リストのインスタンスに対してその関数ポインタを引数にして `map_ow` を呼び出す．例えば，リスト `as` に関数 `f` を適用する場合は次のようにする．

```
template<typename B>
dist_array<B>* map(B (*f)(const A&)) const;

template<typename B>
void map(void (*f)(B*, const A*),
         dist_array<B> *bs) const;

void map_ow(A (*f)(const A&));
```

図 1 map の定義

```
as->map_ow(f);
```

他のスケルトンも同様にして利用することができる．基本的に，ユーザは関数を作成し，その関数ポインタをスケルトンに与えることによって並列プログラムを記述していく．

次に，各並列スケルトンの実装と，計算コストについて説明する．なお，リストの長さを n，プロセッサ数を p とし，スケルトンに与える関数及び二項演算子の計算コストを O(1) とする．

map は，ループを使ってリストの要素に関数を適用する．各プロセッサが保持するリストの長さは n/p で，通信は発生しないので，map の計算コストは $\mathrm{O}(n/p)$ である．

reduce は，各プロセッサにおいてリストの要素を二項演算子で結合し，その結果を二分木構造に基づく

通信を用いて結合していく．最終的な結果は 1 つのプロセッサに集約されるので，最後にその値をブロードキャストする．二項演算子による結合が $O(n/p)$，木構造通信が $O(\log p)$，ブロードキャストが $O(\log p)$ なので，reduce の計算コストは $O(n/p + \log p)$ である．

scan は，各プロセッサにおいてリストの要素を二項演算子で結合し，その結果をバタフライ通信を用いて分配する．そして，分配された値を初期値として，リストの要素を二項演算子で結合した途中結果をリストにする．二項演算子による結合が $O(n/p)$，バタフライ通信が $O(\log p)$，途中結果の計算が $O(n/p)$ なので，scan の計算コストは $O(n/p + \log p)$ である．

zip は，2 つのリストの要素を C++ 標準テンプレートライブラリの pair で組にする．zip の計算コストは，map と同様に $O(n/p)$ である．

3.3 プログラム例

簡単な例として，リスト $as = [a_1, a_2, \ldots, a_n]$ が与えられたとき，その分散 var を求めるプログラムを考える．分散は次の式で求められる．

$$\begin{aligned} var &= \sum_{i=1}^{n}(a_i - ave)^2/n \\ ave &= \sum_{i=1}^{n} a_i/n \end{aligned}$$

この計算を BMF で記述すると図 2 (a) のようになる．一方，本ライブラリで記述すると図 2 (b) のようになり，BMF とほぼ同様に記述できることがわかる．このように，本ライブラリを用いると，並列プログラムを簡潔に記述することができる．

もう一つの例として，統計解析手法における分散分析のプログラム例を示す．分散分析は，データの持つばらつき（分散）を要因別に分解し，比較する手段である．ここでは，n 個のデータを一つの群とし，a 個の群の間に有意な差があるかを分散分析を用いて検定する．以下に示す並列スケルトンによるプログラムでは，群の数 a はそれほど大きい値ではなく，1 群の要素数 n が大きいと考えて，n の大きさに関係する部分に対して並列化を行っている．したがって，各群を dist_array で表現し，これに対して並列スケルトンを適用する．本ライブラリによる記述の一部を，図 3 に示す．ここで，number は群の数，size は要素数である．

$var\ as = sqSum/n$
where
  sum = reduce (+) as
  $ave = sum/n$
  $sqSum$ = reduce (+)
      (map $square$ (map $(-ave)$ as))

(a) BMF による記述

```
sum = as->reduce(add);
ave = sum / n;
as->map_ow(sub_ave);
as->map_ow(square);
sq_sum = as->reduce(add);
var = sq_sum / n;
```

(b) 本ライブラリによる記述

図 2 分散を求める計算の記述

```
...
for(int i = 0; i < number; i++){
  ave_a[i] = a[i].reduce(add) / size;
  ave += ave_a[i];
}
...
for(int i = 0; i < number; i++){
  a[i].map_ow(sub_ave);
  a[i].map_ow(square);
}
for(int i = 0; i < number; i++)
  st += a[i].reduce(add);
...
```

図 3 分散分析のプログラムの一部

このように，ある程度規模の大きい問題に対しても，本ライブラリを用いて記述することができる．

4 スケルトン並列プログラムの最適化

「はじめに」で述べたように，スケルトン並列プログラムはそのままでは必ずしも効率が良いとは限らない．例えば，map f (map g x) は map を 2 回実行しなければならない．これを map $(f \circ g)$ x と最適化すれば，map を 1 回実行するだけで済む．このよう

な最適化は，従来の並列スケルトンシステムではほとんど考慮されていなかった．本研究では，融合変換と呼ばれる，複数の関数呼出しを単一の関数呼出しに融合する手法を用いてソースコードを変換し，効率が良いプログラムを生成する機構を実現した．

4.1 最適化規則

最適化を行う際に，全てのスケルトンの組み合わせを考えると，非常に大きな最適化規則が必要になってしまう．本研究では Hu らによる方法[8] に基づき，集約された最適化規則を用いる．

スケルトンプログラムを扱うにあたって，並列スケルトンを accumulate および cataJ，buildJ という関数を用いた中間表現に変換する．cataJ と buildJ の定義を次に示す．

定義 (cataJ)**.** cataJ は accumulate の特別な場合である．p を関数，$\oplus$ を e を単位元としてもつ結合的な二項演算子として，次のように定義する．

$$\begin{aligned}\mathsf{cataJ}\ [\,] &= e \\ \mathsf{cataJ}\ (a : x) &= p\ a\ \oplus\ \mathsf{cataJ}\ x\end{aligned}$$

以降，本論文では cataJ を $(\!|\oplus, p, e|\!)$ と書く．

定義 (buildJ)**.** buildJ は関数の引数にリストの構成子を与えるものであり，次のように定義する．

$$\mathsf{buildJ}\ gen = gen\ (+\!\!+)\ [\cdot]\ [\,]$$

cataJ は，リストを append リストとして捉え，構成子 $[\,]$ を e に，$[\cdot]$ を p に，: を二項演算子 $\oplus$ に置きかえる関数である．その結果，リストの各要素に関数 p を適用した値を $\oplus$ で結合した値を作り出す．reduce は cataJ の特別な場合（p が恒等関数のとき）として捉えることができる．

一方，buildJ の引数は，リストの 3 個の構成子を引数として受け取りリストを生成するような関数である．buildJ は，このような関数に対し，実際にリストの構成子を与えてリストを作る．buildJ によって作られたリストが cataJ に渡される場合，buildJ の引数の関数にはじめから e，p，$\oplus$ を与えれば，リストを作ることなく直接結果を得ることができる．これが，後で述べる CataJ-BuildJ 規則である．

accumulate，cataJ，buildJ を用いて並列スケルトンを表現すると，次のようになる．ここで，id を恒等関数とする．

$$\begin{aligned}&\mathsf{map}\ f \quad = \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|c, s \circ f, e|\!)) \\ &\mathsf{reduce}\ (\oplus) = (\!|\oplus, id, e|\!) \\ &\mathsf{scan}\ (\oplus)\ x = \\ &\quad \mathsf{buildJ}\ (\lambda c\ s\ e.\ [\![s, (\lambda(a, e).\ s\ e, c), (id, \oplus)]\!])\ x\ e\end{aligned}$$

このように表現された並列スケルトンに対し，次の最適化規則を用いてスケルトンプログラムを融合変換し，最適化する．

CataJ-BuildJ：

$$(\!|c, s, e|\!) \circ \mathsf{buildJ}\ gen = gen\ c\ s\ e$$

この規則の適用例として，map の結果に reduce を実行するプログラムを単一の cataJ に融合する例を示す．

$$\begin{aligned}&\mathsf{reduce}\ (\oplus) \circ \mathsf{map}\ f \\ &= (\!|\oplus, id, e|\!) \circ \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|c, s \circ f, e|\!)) \\ &= ((\lambda c\ s\ e.\ (\!|c, s \circ f, e|\!))\ (\oplus)\ id\ e) \\ &= (\!|\oplus, f, e|\!)\end{aligned}$$

しかし，この規則だけでは，map $f \circ$ map g を融合することができない．そこで，次の規則を定義する．

BuildJ(CataJ-BuildJ)：

$$\begin{aligned}&\mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|\phi_1, \phi_2, \phi_3|\!)) \circ \mathsf{buildJ}\ gen = \\ &\qquad \mathsf{buildJ}\ (\lambda c\ s\ e.\ gen\ \phi_1\ \phi_2\ \phi_3)\end{aligned}$$

この規則を適用すると，map $f \circ$ map g は次のように融合することができる．

$$\mathsf{map}\ f \circ \mathsf{map}\ g = \mathsf{buildJ}\ (\lambda c\ s\ e.\ (\!|c, s \circ f \circ g, e|\!))$$

上の規則を一般化したのが，次の規則である．ここで，fst を組の第一要素を返す関数とする．

BuildJ(Acc-BuildJ)：

$$\begin{aligned}&\mathsf{buildJ}\ (\lambda c\ s\ e.\ [\![g, (p, \oplus), (q, \otimes)]\!]) \circ \\ &\quad (\mathsf{buildJ}\ gen\ x)\ e \\ &\qquad = fst\ (\mathsf{buildJ}\ (\lambda c\ s\ e.\ gen\ (\odot)\ f\ d)\ x\ e) \\ &\mathbf{where} \\ &\quad (u \odot v)\ e = \mathbf{let}\ (r_1, s_1, t_1) = u\ e \\ &\qquad\qquad\qquad\qquad (r_2, s_2, t_2) = v\ (e \otimes t_1)\end{aligned}$$

$$\textbf{in}\ (s_1 \oplus r_2, s_1 \oplus s_2, t_1 \otimes t_2)$$
$$f\ a\ e = (p\ (a,e) \oplus g\ (e \otimes q\ a), p\ (a,e), q\ a))$$
$$d\ e = (g\ e, _, _)$$

4.2　実装

本研究では，最適化機構を OpenC++[5] を用いて実装した．本システムでは，スケルトンプログラムのソースコードを cataJ，buildJ などを用いた中間表現に変換し，最適化を行い，中間表現を元の並列スケルトンに還元することで最適化を実現している．

OpenC++において，ソースコードには解析木としてアクセスすることができる．中間表現への変換は，ソースコードを始めから順に見ていき，並列スケルトンがあれば，それを対応する中間表現に置換する．最適化は，並んだ式を 2 つずつ見ていき，対応する組み合わせがあれば，最適化規則に従って融合する．中間表現からの還元は，中間表現への変換の逆を行う．map $f \circ$ map g を map $(f \circ g)$ に最適化したときなど，場合によっては関数を合成する必要がある．本機構では，合成する関数を連続して呼び出すような関数を新しく定義することで，関数合成を実現している．

3.3 節での分散のプログラム例を解析木で表し，map_ow と reduce を中間表現に変換すると次のようになる．

```
[[‘1‘ as -> sum ‘cataJ‘ [[add]] nil nil] ;]
[[ave = [sum / size]] ;]
[[‘3‘ as -> as ‘buildJ‘
         ‘cataJ‘ nil [[sub_ave]] nil] ;]
[[‘3‘ as -> as ‘buildJ‘
          ‘cataJ‘ nil [[square]] nil] ;]
[[‘1‘ as -> sq_sum
              ‘cataJ‘ [[add]] nil nil] ;]
[[var = [sq_sum / size]] ;]
```

ここで，BuildJ(CataJ-BuildJ) 規則と CataJ-BuildJ 規則を順に適用すると次のようになる．

```
[[‘1‘ as -> sum ‘cataJ‘ [[add]] nil nil] ;]
[[ave = [sum / size]] ;]
[[‘1‘ as -> sq_sum ‘cataJ‘ [[add]]
          [[sub_ave] [square]] nil] ‘;‘]
[[var = [sq_sum / size]] ;]
```

この解析木を，元の並列スケルトンに還元すると次のようになる．

表 1　実験環境

CPU	Pentium4 2.4GHz
メモリ	512MB
ネットワーク	1Gbps
OS	Linux 2.4.20
コンパイラ	g++2.96
MPICH	mpich 1.2.6

```
sum = as->reduce(add);
ave = sum / size;
sq_sum = as->cataj(_sym11086_2, add);
var = sq_sum / size;
```

ここで，_sym11086_2 は sub_ave と square を連続して呼び出すように自動的に定義した関数である．このようにして，並んだ 2 つの map と reduce を融合して 1 つの cataJ にすることができ，$2(n/p)$ 回分のループを削減できた．

5　実験

3.3 節で示したプログラムについて，最適化していないスケルトン並列プログラム，最適化したスケルトン並列プログラム，C++と MPI のみで書いたプログラムの 3 種類プログラムの実行時間を測定した．分散を求めるプログラムでは，リストの大きさを 1000 万，分散分析のプログラムでは，要素数を 10 万，群の大きさを 100 とし，ともに計算回数を 100 回とした．実験は，表 1 に示す同性能の計算機 10 台からなる PC クラスタで行った．

図 4 に分散のプログラムの実験結果のグラフを示す．最適化した方がしないものと比べて 29.7%実行速度が速く，最適化の効果が得られていることが確認できた．このうち，BuildJ(CataJ-BuildJ) 規則による効果が 16.0%，CataJ-BuildJ 規則による効果が 13.7%であった．また，10 台の実行で 8.20 倍速くなっていることから，台数効果が得られていることも確認できた．

図 5 に分散分析のプログラムの実験結果のグラフを示す．最適化した方がしないものと比べて 7.8% 実行速度が向上した．分散のプログラムと比較して，実行速度の向上が小さいのは，プログラムの規模の割

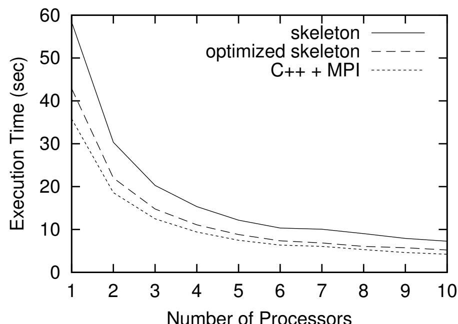

図 4　分散の実験結果

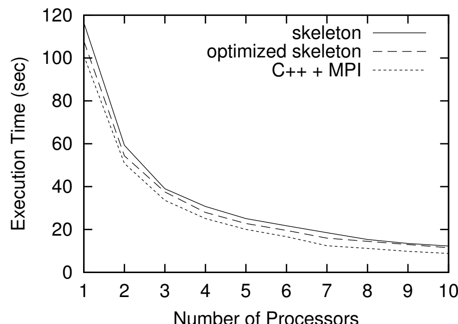

図 5　分散分析の実験結果

に，最適化できる部分が非常に少なかったことが原因と考えられる．

また，最適化したプログラムは，C++と MPI のみで書いたプログラムと比較すると，分散，分散分析ともに 15% 程度のオーバーヘッドがあった．†1

6　関連研究

今まで，スケルトン並列プログラミングに関する多くの研究がなされ，システム・言語も提案されている．

P3L [2] は，スケルトンの概念に基づいた並列プログラミング言語であり，map，reduce，scan などのデータ並列スケルトンに加え，pipe（パイプライン処理に用いる）等のタスク並列スケルトンが提供されている．P3L では，逐次的な部分は C 言語で記述し，データ型やスケルトンの部分は C 言語に似た構文で記述する．

Skil [4] は，C 言語をベースとした上に関数型言語の特徴を持つ並列プログラミング用手続き型言語である．スケルトンを組み合わせて記述した Skil プログラムを C 言語に変換して実行する．プログラマはデータが分散されていることを意識してプログラミングする必要があるため，スケルトンによる並列部分の隠蔽が十分とはいえない．

HPC++ [10] は，並列スケルトンという立場ではなく，標準テンプレートライブラリの並列化という立場から開発されたシステムであり，分散的な多次元配列，map，reduce，scan に相当する機能を提供している．HPC++ では，並列実行のためのコンパイラに対する指示を行う必要がある．

これらの研究と本研究を比較すると，P3L，Skil はともに C 言語にスケルトン用の構文拡張，HPC++ では並列化指示な構文拡張を行っているが，本ライブラリは，C++ に対する拡張は一切行っていないため，新たな構文を修得する必要なく純粋な C++ のプログラムとして並列プログラムを記述し利用することができる．また，上の研究はいずれも融合変換のようなプログラム最適化を考慮していないのに対し，本研究のシステムは，ソースコード変換による最適化を行う点が大きく異なっている．

7　おわりに

本研究では，BMF に基づいた並列スケルトンライブラリを C++上に実装した．その上で，融合変換を用いて並列スケルトンで書かれたプログラムを最適化する機構を実現した．実験では，最適化の効果が得られ，台数効果も得られていることが確認できた．

本論文は，リストに対する並列スケルトンに焦点をあてて説明したが，木に対する並列スケルトンについては，現在，二分木に対して map，zip，reduce，scan に相当する 2 つのスケルトン，の 5 つの基本スケルトン [12]，および，データ分散を行うスケルトンが提供されている．これらは，Tree Contraction [1] [11] という手法に基づいて，分散メモリ型並列計算機においても効率が良くなるよう実装されている．

本ライブラリはまだ初期段階であり，実用的なライブラリに向けて研究中である．zip の最適化規則や，

†1　投稿時の実装であり，現在の実装では分散は 5% 程度，分散分析は 7% 程度のオーバヘッドである．

データの依存関係を考慮した最適化機構を実現することが今後の課題である.

参 考 文 献

[1] Abrahamson, K., Dadoun, N., Kirkpatrik, D., and Przytycka, T.: A Simple Parallel Tree Contraction Algorithm, *Journal of Algorithms*, Vol. 10, No. 2 (1989), pp. 287–302.

[2] Bacci, B., Danelutto, M., Orlando, S., Pelagatti, S., and Vanneschi, M.: P3L: A Structured High Level Programming Language and its Structured Support, *Concurrency: Practice and Experience*, Vol. 7, No. 3 (1995), pp. 225–255.

[3] Bird, R.: An Introduction to the Theory of Lists, *Proc. NATO Advanced Study Institute on Logic of Programming and Calculi of Discrete Design*, Springer-Verlag, 1987, pp. 5–42.

[4] Botorog, G. and Kuchen, H.: Skil: An Imperative Language with Algorithmic Skeletons for Efficient Distributed Programming, *Proc. 5th International Symposium on High Performance Distributed Computing* (*HPDC-5*), IEEE Computer Society Press, 1996, pp. 243–252.

[5] Chiba, S.: OpenC++. `http://opencxx.sourceforge.net/`.

[6] Cole, M.: *Algorithmic Skeletons: A Structured Approach to the Management of Parallel Computation*, Research Monographs in Parallel and Distribute Computing, Pitman, 1989.

[7] Hu, Z., Iwasaki, H., and Takeichi, M.: Diffusion: Calculating Efficient Parallel Programs, *Proc. 1999 ACM SIGPLAN International Workshop on Partial Evaluation and Semantics-Based Program Manipulation* (*PEPM'99*), 1999, pp. 85–94.

[8] Hu, Z., Iwasaki, H., and Takeichi, M.: An Accumulative Parallel Skeleton for All, *Proc. 2002 European Symposium on Programming* (*ESOP'2002*), Lecture Notes in Computer Science 2305, Springer-Verlag, 2002, pp. 83–97.

[9] Iwasaki, H. and Hu, Z.: A New Parallel Skeleton for General Accumulative Computations, *International Journal of Parallel Programming*, Vol. 32, No. 5 (2004), pp. 389–414.

[10] Johnson, E. and Gannon, D.: HPC++: Experiments with the Parallel Standard Template Library, *Proc. 11th International Conference on Supercomputing*, ACM Press, 1997, pp. 124–131.

[11] Miller, G. and Reif, J.: Parallel Tree Contraction and its Application, *Proc. 26th Annual Symposium on Foundations of Computer Science*, IEEE Computer Society Press, 1985, pp. 478–489.

[12] Skillicorn, D.: Parallel Implementation of Tree Skeletons, *Journal of Parallel and Distributed Computing*, Vol. 39, No. 2 (1996), pp. 115–125.