助っ人：構成的な並列スケルトンによる並列プログラミングライブラリ

SkeTo: A Library for Parallel Programming with Constructive Skeletons

松崎 公紀 [1], 明石 良樹 [2], 江本 健斗 [1], 岩崎 英哉 [3], 胡 振江 [1]
Kiminori MATSUZAKI, Yoshiki AKASHI, Kento EMOTO, Hideya IWASAKI, Zhenjiang HU

[1] 東京大学大学院情報理工学系研究科
Graduate School of Information Science and Technology, University of Tokyo
{kmatsu,Kento_Emoto,hu}@mist.i.u-tokyo.ac.jp

[2] 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
yoshiki@ipl.cs.uec.ac.jp

[3] 電気通信大学情報工学科
Department of Computer Science, The University of Electro-Communications
iwasaki@cs.uec.ac.jp

並列プログラミングは, プロセッサ間通信, 資源の配置, 同期などの点で逐次プログラミングより難しいと言われている. これらの問題を解消する一つのパラダイムとして, スケルトン並列プログラミングが提案された. これまで, 多くの研究者によってスケルトン並列プログラミングの理論に関する研究が行われており, このような成果を実用化する並列スケルトンライブラリの構築は重要である.

本論文では, 構成的アルゴリズム論に基づく並列スケルトンライブラリ「助っ人」を提案する. 「助っ人」の特長は, 大きく次の二点である. (1) リスト (一次元配列), 二次元配列, および, 木に対する並列スケルトンを統一的な枠組みの下で提供する. (2) 構成的アルゴリズム論における融合変換による自動的な最適化機構を実現した. 本論文では, 本ライブラリを使うことでどのように並列プログラムを作成することができるかを, 具体例を交えながら説明する.

1 はじめに

近年, PC クラスタなどの並列計算機環境が非常に身近なものになっているが, その一方で, それらを利用するために効率的な並列プログラミングを行うことはまだ難しいと言われている. この理由として, 並列プログラミングではプロセッサ間通信, 資源の配置, 同期などの点を考慮する必要があり, 逐次プログラミングよりプログラマが考えなければならないことが多いことが挙げられる. また, ある並列計算機環境で効率の良いプログラムは, 別の並列計算機環境で効率が悪い, もしくは, 実行できないこともある.

このような問題点を解決するためのひとつのアプローチとして, スケルトン並列プログラミング (Skeletal Parallel Programming) [11, 29] が提案された. スケルトン並列プログラミングでは, 並列スケルトンと呼ばれる並列計算パターンを組み合わせることで並列プログラムを作成する. この並列スケルトンは, 並列計算における頻出計算パターンを抽象化したものであり, また, 多くの並列計算機環境において効率良く実装できることが示されている. これまで, スケルトン並列プログラミングに関する多くの理論的な研究が行われており [7, 12, 18, 34], これらの成果をユーザに提供するスケルトン並列プログラミング環境の構築は非常に重要である.

著者らは, 並列プログラミングに関する知識をあまり持たない人でも並列プログラムを作成することができることを目標として, スケルトン並列プログラミングのためのライブラリ「助っ人 (英語名 SkeTo)」を開発してきた. 「助っ人」は標準の C++と MPI を用いて実装されており, 複数のデータ構造の上での並列スケルトンを, 構成的アルゴリズム論に基づく統一的な枠組みの上で提供する. また, これらの並列スケルトンを提供するだけでなく, MPI の記法を隠蔽

することにより, 並列プログラミングに不慣れな人でも逐次プログラムを作成するようにプログラミングできるようになっている. 「助っ人」では, 提供する並列スケルトンに対して構成的アルゴリズム論に基づく融合変換を実装しており, これによってスケルトン並列プログラミングにおいて問題になりやすい効率の問題を改善することが可能となっている.

例えば, mpich のサンプルにも含まれている, モンテカルロ法によって π の近似値を求める並列プログラムを考えてみよう. この並列プログラムでは, p 個のプロセッサが, 合計で n 個の点をランダムに生成し, 単位円の中に入っている個数を求める. その後, 各プロセッサの結果を集計し, π の近似値とする. 「助っ人」の並列スケルトンを用いると, この並列計算の中心的な部分は次のように書くことができる.

```
// 点の個数を表すリストの生成
dist_list<int> *as
  = new dist_list<int>(gen, p);
// 各プロセッサでモンテカルロ法の実行
list_skeletons::map_ow(monte, as);
// 結果の集計と近似値を求める
int sum
  = list_skeletons::reduce(add, 0, as);
double pi = (double) sum / n;
```

ここで, `map_ow`, `reduce` はそれぞれ並列スケルトン (第 2 節) であり, その引数である, `gen`, `monte`, `add` はそれぞれ逐次プログラムとして定義された関数オブジェクトである. このように, 「助っ人」の並列スケルトンを用いると, それほど並列性を意識せずに逐次プログラムのように並列プログラムを作成することができる.

「助っ人」の特長は, 次の通りである.

構成的アルゴリズム論に基づく並列スケルトン

これまでの並列スケルトンライブラリでは, 主にリスト (一次元配列) が対象とされてきた. 「助っ人」では, 構成的アルゴリズム論に基づいて定式化された並列スケルトンを実現しており, 統一的な枠組みの中で, リスト, 二次元配列, および, 木に対する並列スケルトンを提供している.

融合変換による最適化機構

スケルトン並列プログラミングでは, 並列スケルトンによる比較的小さな処理を組み合せることでプログラムを作成するため, 小さな関数呼び出しによるオーバヘッドや, 関数間で受け渡される中間的なデータ構造の生成や操作によるオーバヘッドなどが問題になりやすい. 「助っ人」では, 構成的アルゴリズム論の分野で研究されてきた融合変換を並列スケルトンに対して定式化, 実現することにより, これらのオーバヘッドを軽減し効率を改善させている.

標準の C++と MPI による実装

これまでの多くの並列スケルトンライブラリは, 並列スケルトンのための特別な言語を導入するものが多くあった. これに対して, 「助っ人」の提供する並列スケルトンは, 標準の C++ の言語仕様のもとで定義されており, C++ と MPI を用いて実装されている. これにより, ユーザは特別な記法を新しく覚える必要がなく, また, 様々な並列計算機環境において使用することができる.

MPI や並列計算の知識をあまり要求しない設計

「助っ人」は, 各データ構造に対する並列スケルトンと同時に, 与えられるデータを複数のプロセッサに分配・収集するための機能を提供する. これによって, ユーザは資源の配置についてそれほど意識する必要がない. また, MPI をあまり良く知らない人でも使用できるよう, 直接 MPI の関数を呼ばなくても適切に動作するように実現されている.

本論文では, 「助っ人」の特長, および, それを用いた並列プログラミングについて述べる.

本論文の構成は次の通りである. 続く第 2 節では, 表記法を簡単に述べた後, スケルトン並列プログラミングについて, 理論的な定式化を概観する. 第 3 節では, 「助っ人」を用いて並列プログラムを作成する方法, および, 「助っ人」の実装について, 基本的なデータ構造であるリストの場合について説明する. リスト以外のデータ構造に対する並列スケルトンとその実装については, 第 4 節でそれらの概要を示す. 「助っ人」による並列プログラミングの評価を第 5 節で示す. 最後に, 第 6 節で関連研究について触れ, 本論文のまとめと今後の課題について第 7 節で述べる.

2 スケルトン並列プログラミング

本節では, スケルトン並列プログラミングについて, その理論的な内容を概観する. まず本論文中で用いる表記法について簡単に述べ, 続いてリストに対する基本的な並列スケルトンとそれを用いてプログラミングを行う方法を例を用いて説明する.

2.1 表記法

本論文では, 基本的に関数型言語 Haskell [5] の表記法を用いてアルゴリズムを表現する.

関数

関数適用は空白によって表現し, 関数の引数の括弧は省略する. つまり, $f(a)$ のことを $f\ a$ と表記する. 関数はカリー化され, 関数適用は左結合であるとする. よって, $f\ a\ b$ は $(f\ a)\ b$ を意味する. 関数適用は他の演算より優先順位が高い. 例えば, $f\ a \oplus b$ は $(f\ a) \oplus b$ を意味し, $f\ (a \oplus b)$ ではない. 関数合成は $\circ$ を用いて $(f \circ g)\ a = f\ (g\ a)$ と定義する.

本論文では二項演算子として $\oplus, \otimes$ などを用いる. 二項演算子は括弧で囲むことによりセクション化され, $a \oplus b = (a\ \oplus)\ b = (\oplus\ b)\ a = (\oplus)\ a\ b$ のように単項もしくは二項の関数として扱うことができる.

リスト

リストは同じ型の要素からなる有限の列であり, 空リスト *Nil*, もしくは, 値をリストの先頭に追加する *Cons* によって生成される.

$$\mathbf{data}\ \mathit{List}\ \alpha = \mathit{Nil} \mid \mathit{Cons}\ \alpha\ (\mathit{List}\ \alpha)$$

本論文では $\mathit{List}\ \alpha$ を $[\alpha]$ と, *Nil* を $[\,]$ と, $\mathit{Cons}\ a\ as$ を $a : as$ と, それぞれ表記する. 演算子 : は右結合であり, 次のようなリストの略記法を使用する.

$$1 : 2 : 3 : 4 : [\,] = [1, 2, 3, 4]$$

2.2 リストに対する基本並列スケルトン

スケルトン並列プログラミングでは, 並列スケルトンと呼ばれる並列計算における基本的な処理を抽象化したものを適切に組み合わせることによって並列プログラミングを行う. 我々は, BMF (Bird-Meertens Formalism) [6, 33] を基礎として, 次に挙げる 4 つのデータ並列スケルトンを中心に考える.

並列スケルトン map は, 一引数関数 f とリストをひとつ受け取り, リストの各要素に対して関数 f を適用したリストを返す.

$$\mathsf{map}\ f\ [a_1, a_2, \ldots, a_n] = [f\ a_1, f\ a_2, \ldots, f\ a_n]$$

並列スケルトン reduce は, 結合的な二項演算子 $\oplus$, 初期値 e, および, リストをひとつ受け取り, リストの各要素を二項演算子 $\oplus$ で畳み込んだ値を返す.

$$\begin{aligned}&\mathsf{reduce}\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n]\\ &\quad = e \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_n\end{aligned}$$

並列スケルトン scan は, reduce と同様に, 結合的な二項演算子 $\oplus$, 初期値 e, および, リストをひとつ受け取り, reduce の計算を前から行ったときの中間結果をリストとして返す. これは prefix-sum と呼ばれる並列計算を抽象化したものである.

$$\begin{aligned}&\mathsf{scan}\ (\oplus)\ e\ [a_1, a_2, \ldots, a_n]\\ &\quad = [e, e \oplus a_1, \ldots, e \oplus a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}]\end{aligned}$$

並列スケルトン zip は, ふたつの同じ長さのリストを受け取り, 対応する要素を組にしたリストを返す.

$$\begin{aligned}&\mathsf{zip}\ [a_1, a_2, \ldots, a_n]\ [b_1, b_2, \ldots, b_n]\\ &\quad = [(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)]\end{aligned}$$

これらのスケルトンは, 多くの並列計算機環境で効率的に実装できることが知られており [7, 32], その並列計算におけるコストを簡単な式で与えることができる. 例えば, ハイパーキューブを含む多くの並列計算機において, プロセッサの数を p, リストの要素数を n, 関数 f の逐次計算のコストを t_f とすると, map f の並列コストは $\lceil n/p \rceil \times t_f$, zip の並列コストは $\lceil n/p \rceil \times t_{(,)}$, reduce $(\oplus)$ の並列コストは $(\lceil n/p \rceil + \log p) \times t_{(\oplus)}$, scan $(\oplus)$ の並列コストは, $2 \times (\lceil n/p \rceil + \log p) \times t_{(\oplus)}$, とそれぞれ与えられる. ただし, ここで $(,)$ は組を返す演算とする.

2.3 スケルトン並列プログラミングの例

本節では, 基本並列スケルトンを用いたスケルトン並列プログラミングについて, 分散の計算と括弧対応問題のふたつの具体例を使って説明する.

2.3.1 分散の計算

入力として n 個の値 $a_1, a_2, \ldots, a_n$ が与えられたとき, その分散 *var* は次の式で与えられる.

$$var = \frac{1}{n}\sum_{i=1}^{n}(a_i - ave)^2 \quad \mathbf{where}\ ave = \frac{1}{n}\sum_{i=1}^{n} a_i$$

この式に基づいて分散を求める場合, まず平均値 *ave* を求め, 次にそれぞれの値に対して平均値からの差, および, その二乗を求め, そして分散 *var* を求めることになる. 以下では, 入力の n 個の値はリスト *as* として与えられるとする. このとき, 上記のアルゴリズ

ムは, 基本並列スケルトンの map と reduce を用いて次に示すように直感的に記述することができる.

$$
\begin{aligned}
var\ as\ n = \textbf{let}\ & ave = \mathsf{reduce}\ (+)\ 0.0\ as\ /\ n \\
& as' = \mathsf{map}\ (\lambda x.x - ave)\ as \\
& as'' = \mathsf{map}\ (\lambda x.x^2)\ as' \\
\textbf{in}\ & \mathsf{reduce}\ (+)\ 0.0\ as''\ /\ n
\end{aligned}
$$

2.3.2 括弧対応問題

括弧対応問題とは, 入力として '(', ')', およびそれ以外の文字からなる文字列が与えられ, これに対して, すべての括弧'(', ')' が対応しているかどうかを求める問題である.

これを解く並列アルゴリズムを設計することは容易ではないが, スタックの考え方を用いれば, リスト上の再帰関数として次に示すように記述できる.

$$
\begin{aligned}
& bm\ [\,]\ s = s == 0 \\
& bm\ (a : as)\ s = g_1\ (a, s) \wedge bm\ as\ (g_2\ a\ s) \\
& \textbf{where}\ g_1\ (a, s) = \textbf{if}\ a == \text{'('}\ \textbf{then}\ True \\
& \qquad\qquad \textbf{else if}\ a == \text{')'}\ \textbf{then}\ s > 0 \\
& \qquad\qquad \textbf{else}\ True \\
& \qquad g_2\ a\ s = \textbf{if}\ a == \text{'('}\ \textbf{then}\ s + 1 \\
& \qquad\qquad \textbf{else if}\ a == \text{')'}\ \textbf{then}\ s - 1 \\
& \qquad\qquad \textbf{else}\ s
\end{aligned}
$$

このような再帰関数は, Diffusion 定理 [17] を用いれば次に示すように基本並列スケルトンの組み合わせとして記述することができる.

$$
\begin{aligned}
bm\ as\ s = \textbf{let}\ & as' = \mathsf{map}\ g_2'\ as \\
& ss = \mathsf{scan}\ (+)\ s\ as' \\
& s' = \mathsf{reduce}\ (+)\ s\ as' \\
& ass' = \mathsf{map}\ g_1\ (\mathsf{zip}\ as\ ss) \\
\textbf{in}\ & \mathsf{reduce}\ (\wedge)\ True\ ass' \wedge s' == 0 \\
\textbf{where}\ g_2'\ a = \textbf{if}\ & a == \text{'('}\ \textbf{then}\ 1 \\
& \textbf{else if}\ a == \text{')'}\ \textbf{then} - 1 \\
& \textbf{else}\ 0
\end{aligned}
$$

このように, スケルトン並列プログラミングは, 並列アルゴリズムを直感的に, もしくは系統的に導出することができるという利点がある.

3 「助っ人」による並列プログラミング

我々は, 構成的アルゴリズム論に基づいた並列スケルトンライブラリ「助っ人」を開発, 公開[1] した.

[1] SkeTo Project Homepage. http://www.ipl.t.u-tokyo.ac.jp/sketo/

「助っ人」では並列スケルトンのための特別な記法を導入せず, 各並列スケルトンは標準の C++の文法で定義されている. したがって, ユーザが並列スケルトンを使用するための敷居が比較的小さくて済むようになっている. また, 「助っ人」は C++と MPI を用いて実装されており, 並列スケルトンを用いて記述されたプログラムは多くの並列計算機環境で実行できるという利点がある.

「助っ人」のおおまかな構成を図 1 に示す. 「助っ人」は, MPI のラッパー関数や関数オブジェクトの型定義などの共通に使用されるユーティリティ, 並列データ構造とそれに対する並列スケルトン, および, 融合変換による最適化機構, の大きく 3 つの機能から構成されている. リスト, 二次元配列, 木に対する重要な並列スケルトンの一覧を表 1 に示す.

本節では, 「助っ人」の構成についてリストに対するスケルトンを中心に述べる. また, 本ライブラリを用いてどのように並列プログラムを作成することができるかを, 具体例として第 2.3.1 節の分散を求める問題を解く並列プログラム (図 2) を使って説明する.

3.1 共通ユーティリティ

共通ユーティリティは, 「助っ人」の並列スケルトンを用いてプログラムを作成する際に共通する機能として, MPI を隠蔽する機能と, 関数オブジェクトに関する機能を提供する.

「助っ人」では, MPI にあまり詳しくない人でも使えるように, MPI の関数を直接呼び出さなくてもプログラムを作成できるようにしてある. 「助っ人」を用いたプログラムは, `SketoMain` という関数 (25 行目) から実行される. 通常, MPI を用いた並列プログラムは `MPI_Init` や `MPI_Finalize` などの関数を呼び出さなくてはならないが, これらの呼び出しをは `SketoMain` によって隠蔽され, ユーザは通常の main 関数と同じようにプログラムを作成できる. `skeleton::rank` (42 行目) は MPI における `MPI_Comm_rank` 関数の結果を与える.

図 2 のプログラムにおいて, MPI 専用のデータ型, 例えば `double` に対応する `MPI_DOUBLE`, が指定されていないことに注意してもらいたい. 「助っ人」では, C++のテンプレート機能を利用して, C++の基本型や `std::pair` などから MPI におけるデータ型を自動的に補完するようになっている. これらの機能によって, ユーザは MPI の詳細な仕様などを特に意識することなくプログラムを作成することができる.

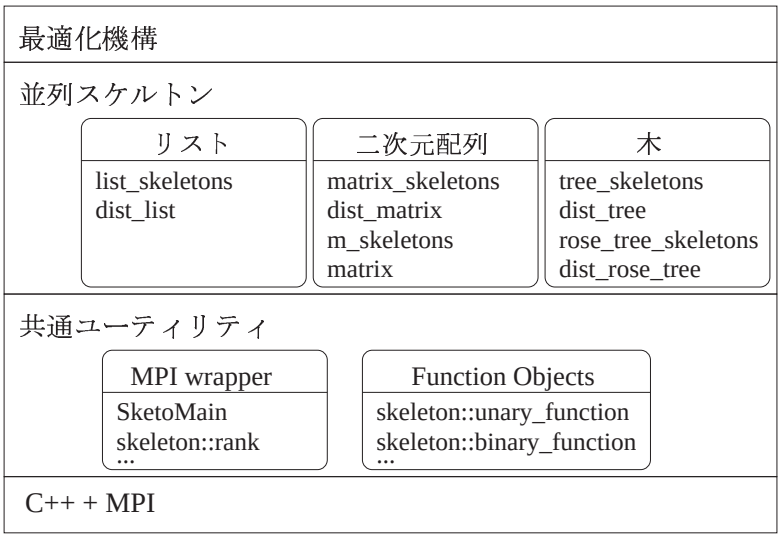

図 1: 「助っ人」の構造

表 1: 「助っ人」における重要な並列スケルトン

リストスケルトン

map	すべての要素に同じ関数を適用する
zip	ふたつのリストのそれぞれの要素を組にする
reduce	結合的な二項演算子で畳み込む
scan	reduce の中間結果を保持するリストを返す
accumulate	特定の形の再帰関数を Diffusion 定理に基づき計算する

二次元配列スケルトン

map	すべての要素に同じ関数を適用する
zip	ふたつの二次元配列のそれぞれの要素を組にする
reduce	Abide な演算子の組を使って行列の要素を畳み込む
scan	左上から reduce を計算した中間結果を二次元配列として返す
rotateRows	分割された各ブロックを横方向に回転させる
rotateCols	分割された各ブロックを縦方向に回転させる

木スケルトン (二分木)

map	二分木のすべての要素に同じ関数を適用する
zip	ふたつの二分木のそれぞれの要素を組にする
reduce	木をボトムアップに畳み込みひとつの値を返す
uAcc	各ノードの値をボトムアップに累積させ, 同じ形の二分木を返す
dAcc	各ノードの値をトップダウンに累積させ, 同じ形の二分木を返す

木スケルトン (薔薇木)

map	薔薇木のすべての要素に同じ関数を適用する
reduce	演算子の組を使って薔薇木をボトムアップに畳み込む

```
1  #include <iostream>
2  #include "list_skeletons.h"
3
4  const int SIZE = 1000;
5
6  struct Gen : public skeleton::unary_function<int, double> {
7    double operator()(int index) const { return static_cast<double>(index); }
8  } gen;
9
10 struct Add : public skeleton::binary_function<double, double, double> {
11   double operator()(double x, double y) const { return x + y; }
12 } add;
13 const double add_unit = 0.0;
14
15 struct Sqr : public skeleton::unary_function<double, double> {
16   double operator()(double x) const { return x * x; }
17 } sqr;
18
19 struct Sub : public skeleton::unary_function<double, double> {
20   double val;
21   Sub(double val_) : val(val_){ }
22   double operator()(double x) const { return x - val; }
23 };
24
25 int SketoMain(int argc, char **argv)
26 {
27   // generate data
28   dist_list<double> *as = new dist_list<double>(gen, SIZE);
29
30   // calculate average
31   double sum1 = list_skeletons::reduce(add, add_unit, as)
32   double ave = sum1 / SIZE;
33
34   // calculate variance
35   Sub sub(ave);
36   list_skeletons::map_ow(sub, as);
37   list_skeletons::map_ow(sqr, as);
38   double sum2 = list_skeletons::reduce(add, add_unit, as) / SIZE;
39   double var = sum2 / SIZE;
40
41   // output results
42   if(skeleton::rank == 0){
43     std::cout << "average:" << ave << "\n" << "variance:" << var << "\n";
44   }
45
46   return 0;
47 }
```

図 2: 「助っ人」による分散を求めるプログラム

「助っ人」を用いたスケルトン並列プログラミングでは, 小さな関数を多数呼ぶようなプログラミングスタイルをとるため, 関数呼び出しのオーバヘッドを小さくすることは効率を考える上で重要である. 「助っ人」では, この問題を解決するために関数オブジェクトを利用している (6 行目〜23 行目). 関数オブジェクトを利用して高階関数を実現することにより, その関数オブジェクトに対応する関数の呼出しをインライン展開することが可能となり, 関数呼び出しのオーバヘッドを取り除くことができる. また, 関数オブジェクトを利用することにより, 関数合成や引数の束縛などを可能にすることができる. これらは, 後の第 3.3 節に示す自動的な最適化機構を実現する上でも重要となる.

3.2 リストスケルトン

「助っ人」における各データ構造に対する並列スケルトンのライブラリは, それぞれのデータ構造の並列計算機上での実装を定義するクラスと, その並列データ構造に対する並列スケルトンの実装を与えるクラスからなる.

リストに対する並列データ構造は, C++のテンプレートを用いて実装された `dist_list` クラス (28 行目) によって定義され, リストをプロセッサに分配, 収集する関数を提供する. データの分散の方法について, 現在の実装では, 分散メモリ並列計算機環境における実装の効率のため, ブロック分割によるデータ配置を採用している.

一方, リストに対する並列スケルトンは, 別の `list_skeletons` クラスに実装されている. Haskell などによる定式化から直接的に「助っ人」のプログラムを得ることができるように, 各スケルトンは Haskell による定式化と同様の形で使用できるように定義されている. 例えば, reduce スケルトンによってリストの和を求める

$$\mathsf{reduce}\ (+)\ 0.0\ as$$

というアルゴリズムは,

```
list_skeletons::reduce(add, 0.0, as);
```

のように記述される (31 行目). ここで, `add` は (+) に対応する関数オブジェクトである. 関数オブジェクトを用いることにより, 上記の並列スケルトンの実装は適切にインライン展開される.

ユーザが効率の良い並列プログラムを作成できるように, いくつかの並列スケルトンについては特殊化した実装を与えている. 例えば, 通常の `map` スケルトンは新しくリストを生成して返すが, 入力のリストを上書きするよう特殊化した `map_ow` スケルトンがある. 分散を求めるプログラムの 36, 37 行目のように入力のリストを上書きしても構わない場合には, この `map_ow` スケルトンを用いることで効率を良くすることができる.

さらに, 基本並列スケルトン以外にも, これまでの理論的な研究成果を反映する並列スケルトンが実装されている. 第 2.3.2 節において, ある形の再帰関数が Diffusion 定理によって基本並列スケルトンの組み合わせに分解することができることを述べたが, 「助っ人」ではこの結果となる基本並列スケルトンをまとめた accumulate スケルトンとして提供している. このスケルトンの実装では繰り返しの融合などの最適化を行っており, ユーザはより直接的に, また, 効率の良いプログラムを得ることができる. 例えば, 括弧対応問題の場合, 基本並列スケルトンのみを使って

```
as_ = list_skeletons::map(g2_, as);
ss  = list_skeletons::scan(add, s, as_);
s_  = list_skeletons::reduce(add, s, as_);
ass = list_skeletons::zip(as, ss);
ass_= list_skeletons::map(g1, ass);
result = (s_ == 0) &&
  list_skeletons::reduce(and, true, ass_);
```

という並列プログラムを記述することができるが, accumulate スケルトンを利用することによって, これと同じ結果を得るプログラムを

```
result = list_skeletons::accumulate
           (equal0,
            g1, and, true,
            g2, add, s, as);
```

と簡潔に記述することができる.

3.3 最適化機構

並列スケルトンを組み合わせて作成したプログラムは, 小さな処理を組み合わせることによるオーバヘッドが大きく, 効率があまり良くないことがある. 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り, これを改善する機構は従来の並列スケルトンライブラリではほとんど考慮されていなかった. Aldinucci ら [3] はプログラム変換に基づきスケルト

ンの組み合わせを効率のよいものへと変換する手法を提案しているが, 彼らの方法は非常にアドホックであり, 複数のデータ構造に対するスケルトンや新規スケルトンの追加などに対応するのが困難であった.

そこで, 我々は構成的アルゴリズム論における融合変換 [15] を並列スケルトンの上で定式化し [16, 19], さらに C++に対するメタプログラミング言語である OpenC++ [10] を用いてこの変換機構を実現した [27, 37].

例えば, 分散を求めるプログラムの次の部分について考えてみよう.

```
list_skeletons::map_ow(sub, as);
list_skeletons::map_ow(sqr, as);
double sum2 = list_skeletons::reduce
                    (add, add_unit, as);
```

このプログラムのふたつの `map_ow` は, `sub` と `sqr` を関数合成すれば, ひとつの `map_ow` で実行することができる. さらに, `map_ow` と `reduce` は, 関数を適用しながら二項演算子で畳み込んでいく `cataj` という関数に融合することができる. したがって, このプログラムは以下のように最適化することができる.

```
double sum2;
sum2 = list_skeletons::cataj(add,
  compose_unary_functions(sqr, sub),
                          add_unit, as);
```

ここで, `compose_unary_functions` は合成関数を返す関数であり, C++のテンプレートを利用して実装されている.

このプログラムについて, 本機構によって自動的に最適化を行い, 最適化の前後のプログラムの実行速度を計測した. 図 3 に示すように, 最適化したプログラムは約 1.3 倍速く計算を行うことができ, また, 最適化の前後で同様の台数効果が得られ, 最適化の効果を確認することができた.

本最適化機構は, 「助っ人」で書かれた様々なプログラムに適用することができ, 上記のような最適化を自動的に行うことができる.

4 その他のデータ構造に対するスケルトン

「助っ人」では, リストの他に二次元配列 (行列) や木に対する並列スケルトンが実装されている. 本節ではこれらの並列スケルトンについて述べる.

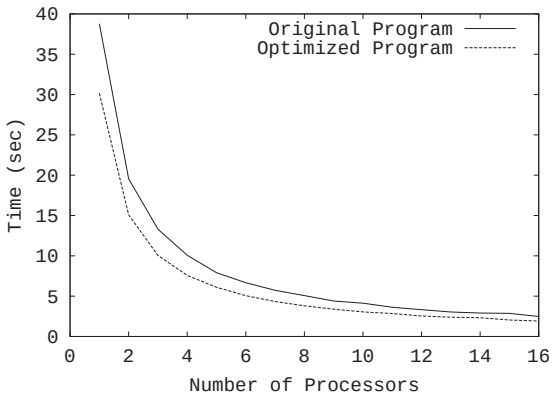

図 3: 最適化前後の実行速度

4.1 二次元配列スケルトン

二次元配列に対する並列スケルトンを実装するスケルトン並列プログラミング環境はいくつか存在している [21, 31]. しかしながら, これらの環境は定式化が不十分なために, 並列化可能な二次元配列上の計算が制限されてしまう. さらに, 「助っ人」の最適化機構のようなスケルトンの組み合わせを最適化する機能は無く, 効率的なプログラムを得るためにはユーザが適切にスケルトンを組み合わせなければならない.

そこで, 「助っ人」では, 二次元配列上の並列スケルトンを構成的アルゴリズム論に基づき根本から定式化し直すことで, これらの問題を解決している [13]. これにより, 実用上重要である二次元配列上のアルゴリズムを容易に効率的に並列化することが可能である.

この定式化においては *abide* 性と呼ばれる性質が重要となっている. この性質を使うことで, 分散並列計算機環境における二次元配列の分配の仕方にとらわれることなく並列プログラムを記述することが可能となり, また, 実際にプログラムを動作させる際にその計算機環境に最も適したデータの分配の仕方で計算を行うことができる. すなわち, 「助っ人」上で開発されたプログラムは, 従来のネストしたリストによる表現 [20] や, 四分木による表現 [14] に適した環境においては, それらの表現を用いて効率的に実行することが可能となる.

「助っ人」における, 二次元配列に対するライブラリは, 分散メモリ並列計算機環境上で二次元配列を分配, 収集するクラスと, その分配されたデータに対する並列スケルトンを提供するクラスから構成される. さらに, 実行される並列アルゴリズムの特性に合うよ

うに, ユーザが二次元配列の分配の仕方をコントロールすることも可能である. 例えば, ある種の行列乗算アルゴリズムでは行列を縦方向と横方向で同じ数に分割して分配しなければならない. この場合, ユーザはライブラリに対して “縦横同数の分割”（正方形分割）の戦略を二次元配列の分配戦略として指定することで, 二次元配列の分配の仕方をアルゴリズムに適した形にコントロールすることができる. そのほかの戦略としては, “縦方向分割” と “横方向分割”, 及び縦横の分割数が同じとは限らない “縦横方向分割” が指定可能である. これらの二次元配列の分配戦略を指定することで, ユーザは実際の並列計算機環境のプロセッサ数を気にすることなく, アルゴリズムに適した分配を行ってアルゴリズムを正しく動作させることが可能となる.

二次元配列スケルトンを使ってプログラムを作成する例として, 行列のフロベニウスノルムを求める問題を考えてみよう. 行列のフロベニウスノルムとは, 行列の要素の二乗和の平方根をとったものである. 二次元配列スケルトン map, reduce を用いるとこのアルゴリズムは,

$$norm = (\mathsf{reduce}\ ((+),(+))) \circ (\mathsf{map}\ (\lambda x.\ x^2))$$

と記述できる. 「助っ人」を用いることで次のような並列プログラムを作成することができる.

```
dist_matrix<double> *dmat,*dmat2;
dmat = new dist_matrix<double>(mat);
dmat2 = matirx_skeletons::map(sq, dmat);
double *ss = matrix_skeletons::reduce
                      (add, add, dmat2);
double ret = sqrt(*ss);
```

上記の例では, デフォルトの分割戦略が用いられるが, これに対して縦方向だけを分割したい場合には, コンストラクタの前に

```
matrix_skeletons::setDistributionPolicy
        (matrix_skeletons::DPOLICY_ROW);
```

と記述して分配の戦略を縦方向分割に変更するだけでよい.

4.2 木スケルトン

「助っ人」では二分木, および, 一般の木に対する並列スケルトンを提供している. 木は XML などのように構造化されたデータを表現するのに適したデータ構造であるが, その一方でその不均等な構造のため効率的に並列化することは難しく, 著者らの知る限りでは木に対する汎用の並列スケルトンライブラリはこれが初めてである.

これまで, 木に対する並列スケルトンとしては, Skillicorn [35] が BMF に基づいて二分木に対する 5 つの並列スケルトンを定式化している. これらの並列スケルトンは, 木に対する重要な並列アルゴリズムである Tree Contraction アルゴリズム [1, 28] を用いて実装できることが示されている [35].

「助っ人」における二分木に対するライブラリは, 分散メモリ並列計算機環境でも効率が良くなるように適切に木のノードを分配, 収集するクラスと, その分配されたデータに対する並列スケルトンを提供するクラスから構成される. 木に対する並列スケルトンは, もともと共有メモリ上のアルゴリズムである Tree Contraction アルゴリズムを, 分散メモリ並列計算機環境でも効率的に動作するように応用して実装されている. したがって, ユーザは [24, 25] などの手法によって導出した Tree Contraction アルゴリズム用の関数を用いて並列スケルトンを呼び出すことにより, 効率的な並列プログラムを得ることができる.

例として, 二分木の高さを求める問題を考えてみよう. 非常に単純な計算方法として, まずそれぞれのノードに 1 を割り当て, 次にルートノードから値を足しながら葉に伝搬させていくことで深さを求め, 最後にその深さの最大値を求める, という方法が考えられる. このアルゴリズムの各ステップはそれぞれ, map, dAcc, reduce という並列スケルトンによって実現できる. 「助っ人」のスケルトンを使うと, このアルゴリズムは次に示すプログラムのように記述される.

```
dist_tree<char> *tree1
 = dist_tree<char>::read_from_file(filename);
dist_tree<int> *tree2
 = tree_skeletons::map(f_one, f_one, tree1);
dist_tree<int> *tree3
 = tree_skeletons::dAcc(f_plus, 0, f_id,
                        f_id, 0, tree2);
int result = tree_skeletons::reduce
             (f_id, f_max, f_max_p, f_max_l,
              f_max_r, f_max_G, tree3);
```

上記のプログラムは, 葉と内部ノードに対する関数, 右と左の再帰に対する関数を指定しなければならないため, リストなどに比較すると関数が多くなっている. また, reduce スケルトンについては, 効率の良い実装を保証するため, より多くの関数が必要になって

表 2: リストスケルトンを用いた並列プログラムの台数効果

問題例	$P=1$		$P=2$		$P=4$		$P=8$		$P=16$	
	時間	比	時間	比	時間	比	時間	比	時間	比
Variance	38.8	1	19.5	1.95	10.1	3.85	5.07	7.64	2.48	15.6
Bracket Matching	22.0	1	16.5	1.34	13.6	1.62	12.1	1.82	11.5	1.91
N-Queen	13.9	1	7.59	1.83	3.87	3.60	1.94	7.16	1.04	13.4
Heat Equation	86.7	1	43.0	2.02	20.4	4.25	7.38	11.8	5.36	16.2

いる. それでも, Tree Contraction を実装するのと比較すると, 非常に簡潔に記述できる.

木の高さの計算は, やや複雑な Tree Contraction アルゴリズムを利用することでボトムアップな 1 パスの計算が可能である. 「助っ人」の reduce スケルトンでも, 適切な関数を使うことで効率的に計算することができる.

一方, 子の数が任意であるような木 (薔薇木) に対する並列スケルトンもいくつか実装されている. 本ライブラリでは, 薔薇木を内部では二分木として表現しているが, ユーザは薔薇木の上で定義される関数のみを指定すれば良い. 薔薇木に対する並列スケルトンは, 内部表現の二分木の上での演算を薔薇木上の関数から自動的に導出し, それらを使って二分木に対するスケルトンを呼び出すことで実現されている. 薔薇木上の関数から二分木上の関数への変換については, [24] に示されているものを実現している.

5 実験

「助っ人」を用いて多くの問題を解く並列プログラムを作成することができる. 我々はこれまでに, さまざまな問題を解く並列スケルトンプログラムを作成し, その台数効果を調べた.

なお, 実験環境としては, 16 台の均質な PC から構成される PC クラスタを用いた. 各 PC は Pentium4 3.0GHz (Hyper Threading ON) の CPU, 1GB のメモリを持ち, これらがギガビットイーサネットで接続されている. OS は Linux 2.6.8, コンパイラは gcc 2.95, MPI ライブラリは mpich 1.2.6 を用いた.

リストスケルトンによる並列プログラム

リストスケルトンを使って, 次に示すような問題を解く並列プログラムを作成した. これらの並列プログラムの実行時間を表 2 に示す. 実行時間には, 初期データの分配と最終結果の収集の時間を含まない.

Variance

第 2.3.1 節の分散を求める図 2 のプログラム. ただし, 要素数を 10000000 とし, 100 回連続して実行した.

Bracket Matching

4 種類の括弧を許す括弧対応問題を解くプログラム. 文字列の長さを 1000000 とし, 100 回連続して実行した.

N-Queen

N-Queen 問題を解くプログラム. $N = 16$ とした.

Heat Equation

一次元の熱伝導方程式を解くプログラム. 要素数を 100000 とし, 10 単位時間のシミュレーションを行った.

全体的には, これらの実験結果は良い台数効果を得ている. Heat Equation の例において, 超線形な結果が得られているが, これはプログラム使用するメモリが大きいことが原因で, そのようなプログラムの場合に多く見られる現象である. Bracket Matching の例において台数効果が得られていないが, これはアルゴリズムにおいて複数の CPU で計算したデータをまとめる際のコストが非常に高いことが大きな原因である.

二次元配列スケルトンによる並列プログラム

二次元配列スケルトンを使って, 次に示すような問題を解く並列プログラムを作成した. これらの並列プログラムの実行時間を表 3 に示す.

Matrix Multiplication

科学計算の基本である行列乗算を行うプログラム. 演算に用いた行列の大きさは 1000×1000 である.

Maximum Segment Sum

二次元配列のあらゆる部分配列（矩形）を考え, その内部の要素和が最大となる部分配列を求めるプログラム. パターン認識などの応用を持つ. ここでは 400×400 の二次元配列に対して最大値のみを求めた.

表 3: 二次元配列スケルトンを用いた並列プログラムの台数効果

問題例	$P=1$		$P=2$		$P=4$		$P=8$		$P=16$	
	時間	比	時間	比	時間	比	時間	比	時間	比
Matrix Multiplication	14.1	1	5.21	2.71	2.52	5.60	1.05	13.40	0.530	26.62
Maximum Segment Sum	6.21	1	3.51	1.77	2.30	2.70	1.72	3.61	1.50	4.15
F-Norm	0.21	1	0.104	1.97	0.053	3.88	0.026	7.72	0.013	15.01
QR Decomposition	297.1	1	-	-	96.60	3.09	64.76*	4.64*	70.47	4.22

(* は $P=9$ で実行)

表 4: 木スケルトンを用いた並列プログラムの台数効果

問題例	$P=1$		$P=2$		$P=4$		$P=8$		$P=16$	
	時間	比	時間	比	時間	比	時間	比	時間	比
Height of Tree	0.547	1	0.214	2.55	0.128	4.28	0.104	5.25	0.080	6.83
XPath Query	1.92	1	0.762	2.52	0.694	2.77	0.476	4.04	0.360	5.35
Party Planning	1.04	1	0.476	2.19	0.230	4.54	0.374	2.79	0.216	4.82
Party Planning (Rose)	0.143	1	0.143	1.00	0.094	1.51	0.069	2.07	0.040	3.66

F-Norm

第 4.1 節に示した, 行列の簡単かつ基本的な演算のひとつであるフロベニウスノルム（要素の自乗和の平方根）を求めるプログラム. 入力には 4000×4000 の行列を与えている.

QR Decomposition

行列の演算の一つである QR 分解を行うプログラム. 分解のアルゴリズムには Frens and Wise[14] の四分木上のアルゴリズムを用いている. 実行にはプロセッサ数がある数の二乗数でなければならない.

行列乗算の例において超線形な結果が得られているが, これは計算機の持つメモリ量に比較して使用するメモリ量が大きい時にしばしば見られる現象であり, 「助っ人」の有効性を示すものである. 一方で, Maximum Segment Sum の実行においては台数が増した場合の台数効果があまり得られていない. これは Maximum Segment Sum を解くプログラムがスケルトンをネストさせて書かれることと, 現在のライブラリの実装ではネストした内側のスケルトンを並列に実行することができないことによる. このネストしたスケルトンの並列実行は将来の「助っ人」で実現される. QR 分解の例に関しては, 今回はその再帰構造の全体を並列化せずに一部の演算の並列化にとどめているため, 台数が多くなった場合に台数効果が小さくなってしまっている.

木スケルトンによる並列プログラム

木スケルトンを使って, 次に示すような問題を解く並列プログラムを作成した. これらの並列プログラムの実行時間を表 4 に示す.

Height of Tree

与えられた二分木の高さを求めるプログラム. 二分木のノード数は 2000001 とした.

XPath Query

XPath クエリを並列に実行するプログラム [26]. 二分木に変換する前のノード数は 2000001 であり, XPath に含まれる軸の数 (長さ) は 5 とした.

Party Planning

パーティ計画問題は木上の動的計画法の一つの問題であり, 二分木に対して最適となる組み合わせを求めるプログラムを作成した. 二分木のノード数は 2000001 とした.

Party Planning (Rose)

薔薇木に対してパーティ計画問題の最適値を求めるプログラム. 薔薇木のノード数は 1000000 とした.

実験結果において, リスト, 二次元配列と同様に超並列な台数効果を得ていることろが見られる. また, 台数効果が比較的小さくなっているのは, リストなどと違って木構造を均等に分割することが困難であるためである.

6 関連研究

これまでにも, スケルトン並列プログラミングのためのライブラリや環境が開発されている. 本節では, それらについて述べ, 「助っ人」との比較を行う.

スケルトン並列プログラミングに関する理論的な研究が主に関数型言語を利用して行われてきたため, 関数型言語の上で動作する並列スケルトンライブラリが自然に実現されている. 例えば, Eden [9] は Haskell 上で実装された並列スケルトンライブラリであり, 関数型言語の特長をいかして並列スケルトンの実装を行っている.

一方, 手続き型言語に基づく並列スケルトンライブラリも提案されている. Skil [8, 30] や P^3L [4] では, C 言語に対して多様性や高階関数などの関数型言語の特徴を扱うための拡張を施し, 並列スケルトンを実現している. しかし, このような言語の拡張はライブラリの使用の敷居を高くしてしまう.

Kuchen [21, 22] は, 標準の C++を用いた並列スケルトンライブラリのプロトタイプを実装した. このライブラリでは, C++のテンプレート機能や関数オブジェクトを利用することによって, 多様性や高階関数などを扱っており [23], 「助っ人」はこれを多く参考にしている. Kuchen のライブラリでは, コントロール並列のためのスケルトンとデータ並列のためのスケルトンを同時に利用することができる. しかし, 個別のスケルトンについてはアドホックに効率的な実装のものを提供しているが, 最適化機構などの効率化は実装されていない.

スケルトン並列プログラミングの最適化を行うようなライブラリの研究はまだ少ない. Aldinucci ら [3] は, スケルトンの組み合わせを最適化することが有効であることを示し, そのためのフレームワークを提案しているが, 彼らの方法では新しいスケルトンや, 複数のデータ構造に対応するのが非常に困難であった. 我々の最適化機構は, 汎用的な融合変換 [15] を基礎としているため, 拡張が比較的容易にできることが期待される.

また, グリッド環境を対象としたスケルトン並列プログラミング環境として ASSIST [2, 36] が有名である. ASSIST では, 並列計算機環境よりもグリッド環境を対象とするため, タスク並列を中心に研究, 実装が行われている.

7 まとめ

本論文では, 構成的アルゴリズム論に基づく並列スケルトンライブラリ「助っ人」について述べた. 「助っ人」では, 複数のデータ構造に対する並列スケルトンが統一的な枠組みの下で提供されており, また, 構成的アルゴリズム論における融合変換を基礎とする最適化機構を実現している. これにより, 並列計算や MPI の知識をあまり持たないユーザでも, 直感的にある程度効率の良い並列プログラムを作成することができる. また, 「助っ人」のスケルトンは, 標準の C++と MPI を用いて実現されており, ユーザが作成した並列プログラムは, 幅広い並列計算機環境の上で動作させることができる.

今後の課題としては, 次のようなことが挙げられる. まず, リストに対してその効果を示した融合変換による最適化機構を, 二次元配列や木に対して定式化し, それを実現する. また, 現在の「助っ人」が提供するデータ並列スケルトンだけではうまく記述しにくい問題もあり, これを解決するためにコントロール並列スケルトンを同様の枠組みの下で提供することに取り組んでいる.

参考文献

[1] K. Abrahamson, N. Dadoun, D. G. Kirkpatrik, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.

[2] M. Aldinucci, S. Campa, P. Ciullo, M. Coppola, S. Magini, P. Pesciullesi, L. Potiti, R. Ravazzolo, M. Torquati, M. Vanneschi, and C. Zoccolo. The implementation of ASSIST, an environment for parallel and distributed programming. In *EuroPar 2003 Conference, LNCS 2790*, pages 712–721, Klagenfurt, Austria, Aug 2003. Springer.

[3] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: The FAN skeleton framework. *Parallel Algorithms and Applications*, 16:87–121, March 2001. Gordon & Breach (Taylor & Francys group).

[4] B. Bacci, M. Danelutto, S. Orlando, S. Pelagatti, and M. Vanneschi. P^3L: a structured high-level parallel language and its structured support. *Concurrency: Practice and Experience*, 7(3):225–255, May 1995.

[5] R. Bird. *Introduction to Functional Programming using Haskell*. Prentice Hall, January 1998.

[6] R. S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, volume 36 of *NATO ASI Series F*, pages 5–42. Springer-Verlag, 1987.

[7] G. E. Blelloch. Scans as primitive operations. *IEEE Transactions on Computers*, 38(11):1526–1538, November 1989.

[8] G. H. Botorog and H. Kuchen. Skil: An imperative language with algorithmic skeletons for efficient distributed programming. In *Proceedings of the Fifth International Symposium on High Performance Distributed Computing (HPDC5)*, pages 243–252. IEEE Computer Society Press, 1996.

[9] S. Breitinger, R. Loogen, Y. Ortega-Mallén, and R. Peña. The Eden Coordination Model for Distributed Memory Systems. In *High-Level Parallel Programming Models and Supportive Environments (HIPS), LNCS 1123*, pages 120–124. IEEE Press, 1997.

[10] S. Chiba. A metaobject protocol for C++. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'95)*, SIGPLAN Notices 30(10), pages 285–299, Austin, Texas, USA, Oct. 1995.

[11] M. Cole. *Algorithmic skeletons : A structured approach to the management of parallel computation.* Research Monographs in Parallel and Distributed Computing, Pitman, London, 1989.

[12] M. Cole. Parallel programming, list homomorphisms and the maximum segment sum problems. Report CSR-25-93, Department of Computing Science, The University of Edinburgh, May 1993.

[13] K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi. A Compositional Framework for Developing Parallel Programs on Two Dimensional Arrays. Technical Report METR2005-09, Department of Mathematical Informatics, University of Tokyo, 2005.

[14] J. D. Frens and D. S. Wise. QR Factorization with Morton-Ordered Quadtree Matrices for Memory Re-use and Parallelism. In *Proc. 2003 ACM Symp. on Principles and Practice of Parallel Programming*, pages 144–154, 2003.

[15] A. Gill, J. Launchbury, and S. Peyton Jones. A short cut to deforestation. In *Proc. Conference on Functional Programming Languages and Computer Architecture*, pages 223–232, Copenhagen, June 1993.

[16] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *11th European Symposium on Programming (ESOP 2002), LNCS 2305*, pages 83–97, Grenoble, France, April 2002. Springer Verlag.

[17] Z. Hu, M. Takeichi, and H. Iwasaki. Towards polytypic parallel programming. Technical Report METR 98-09, University of Tokyo, 1998.

[18] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '99)*, pages 85–94, San Antonio, Texas, January 1999. BRICS Notes Series NS-99-1.

[19] H. Iwasaki and Z. Hu. A new parallel skeleton for general accumulative computations. *International Journal of Parallel Programming*, 32(5):389–414, 2004.

[20] J. Jeuring. *Theories for Algorithm Calculation.* PhD thesis, Utrecht University, 1993. Parts of the thesis appeared in the Lecture Notes of the STOP 1992 Summerschool on Constructive Algorithmics.

[21] H. Kuchen. A skeleton library. In *Proceedings of EuroPar 2002, LNCS 2400*, pages 620–629. Springer-Verlag, August 2002.

[22] H. Kuchen and M. Cole. The integration of task and data parallel skeletons. In *Proceedings of 3rd International Workshop on "Constructive Methods for Parallel Programming" (CMPP 2002)*, pages 3–16, TU Berlin, 2002.

[23] H. Kuchen and J. Striegnitz. Higher-order functions and partial applications for a C++ skeleton library. In *Proceedings of the International Symposium on Computing in Object-oriented Parallel Environments (ISCOPE 2002)*, 2002.

[24] K. Matsuzaki, Z. Hu, K. Kakehi, and M. Takeichi. Systematic derivation of tree contraction algorithms. In S. Gorlatch, editor, *4th International Workshop on "Constructive Methods for Parallel Programming" (CMPP 2004)*, pages 109–123, July 2004.

[25] K. Matsuzaki, Z. Hu, and M. Takeichi. Parallelization with tree skeletons. In *Proceedings of the 9th EuroPar Conference (EuroPar 2003), LNCS 2790*, pages 789–798, Klagenfurt, Austria, Aug 2003. Springer-Verlag.

[26] K. Matsuzaki, K. Kakehi, Z. Hu, and M. Takeichi. Parallelizing xpath queries with tree homomorphisms. Submitted to the Third Asian Symposium on Programming Languages and Systems (APLAS 2005), 2005.

[27] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A fusion-embedded skeleton library. In M. Danelutto, D. Laforenza, and M. Vanneschi, editors, *Proceedings of the 10th International EuroPar Conference, LNCS 3149*, pages 644–653, Pisa, Italy, August/September 2004. Springer-Verlag.

[28] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *26th Annual Symposium on Foundations of Computer Science*, pages 478–489, Portland, OR, October 1985. IEEE Computer Society Press.

[29] F. Rabhi and S. Gorlatch. *Patterns and Skeletons for Parallel and Distributed Computing.* Springer-Verlag New York Inc., 2002.

[30] T. Richert. Skil: Programming with algorithmic skeletons – a practical point of view. In *Proceedings of the 12th International Workshop on Implementation of Functional Languages*, pages 15–30, Aachen, Germany, July 2000. Aachener Informatik Bericht.

[31] J. Sérot and D. Ginhac. Skeletons for parallel image processing: an overview of the SKIPPER project. *Parallel Computing*, 28(12):1685–1708, 2002.

[32] D. Skillicorn. Architecture-independent parallel computation. *IEEE Computer*, 23(12):38–51, December 1990.

[33] D. B. Skillicorn. The Bird-Meertens formalism as a parallel model. In J. S. Kowalik and L. Grandinetti, editors, *NATO ASI Workshop on Software for Parallel Computation, NATO ARW "Software for Parallel Computation"*, volume 106 of *F*, Cetraro, Italy, June 1992. Springer-Verlag NATO ASI.

[34] D. B. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[35] D. B. Skillicorn. Parallel implementation of tree skeletons. *Journal of Parallel and Distributed Computing*, 39(2):115–125, 1996.

[36] M. Vanneschi. The programming model of ASSIST, an environment for parallel and distributed portable applications. *Parallel Computing*, 28(12):1709–1732, 2002.

[37] 明石 良樹, 松崎 公紀, 岩崎 英哉, 筧 一彦, 胡 振江. 最適化機構を持つ C++並列スケルトンライブラリ. **コンピューターソフトウェア**, 22(3):214–222, 2005.