並列スケルトンライブラリ「助っ人」の実現

松崎 公紀　　江本 健斗

東京大学 大学院情報理工学系研究科

{kmatsu,emoto}@ipl.t.u-tokyo.ac.jp

概要. マルチコア CPU の普及などにより並列プログラミングがますます重要となってきている. しかし, 並列プログラミングは逐次プログラミングと比べてより複雑で困難である. これを解決する 1 つの手法がスケルトン並列プログラミングである. この手法は, 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プログラミングを行うというものである.

著者らのグループでは, これまで並列スケルトンライブラリ「助っ人」の開発を行ってきた. 「助っ人」は, C++と MPI による分散メモリ並列計算環境を対象としたスケルトンライブラリであり, 新しい「助っ人」の実現として次の 2 つの改良を行った. 1 つは, C++のライブラリとしてより使いやすいようにインターフェイスを変更したことである. もう 1 つは, 式テンプレートの利用による, 逐次計算部分の融合変換を実装したことである. 本論文では, これらの点を中心に, 「助っ人」の設計と実装を示す.

Implementation of Parallel Skeleton Library "SkeTo"

Kiminori Matsuzaki　　Kento Emoto

Graduate School of Information Science and Technology, University of Tokyo

Abstract. Though parallel programming is getting more and more important according to the evolution of the hardware, it is still a complicated and hard task. *Skeletal parallel programming* is a promising way to parallel programming, in which we build parallel programs by composing ready-made computational patterns called parallel skeletons.

The authors' group has developed a parallel skeleton library "SkeTo," which is implemented in C++ and MPI for distributed-memory parallel environments. Recently, the implementation of the SkeTo library was improved and the two important changes are as follows. First, the interface of the parallel skeleton functions was polished. Second, the fusion-optimization mechanism was implemented based on so-called expression template techniques. In this paper, we show the design and implementation of the new SkeTo parallel skeleton library.

1　はじめに

近年, 計算機やネットワークの高性能化と低価格化によって, 並列計算を行うことができる計算機環境を容易に手に入れることができるようになった. 特に, マルチコア CPU の普及により, 計算機を効率良く利用するために並列プログラミングがますます重要となってきている. しかし, 並列プログラミングは, データおよび計算資源の分配, プロセッサ間の通信や同期など, 逐次プログラミングと比べてより複雑で困難なものとなっている.

これを解決するための 1 つの手法に, 「スケルトン並列プログラミング」 [6] がある. 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で良く使われる計算パターンを組み合わせることでプログラムを作成する. 並列性が並列スケルトンに隠蔽されることにより, ユーザは逐次プログラムを作成するように並列プログラムを作成することができるという利点がある.

著者らのグループでは 1990 年代の後半より, スケルトン並列プログラミングに関する研究を行ってきた. 特に, 構成的アルゴリズム論 [4] という理論に基づき, 並列スケルトンプログラ

ムが逐次的な再帰プログラムから導出できること [11] や, 複数の並列スケルトンの呼び出しを融合するプログラム変換ルール [10] などの成果を得た. これらの理論的な研究成果を実現し実際に使えるような並列スケルトンライブラリを構築することを目的として, 並列スケルトンライブラリ「助っ人」(SkeTo) [1] の開発を行ってきた [15].

並列スケルトンライブラリ「助っ人」の特徴は大きく次の 3 つである.

- PC クラスタなどの分散メモリ並列計算機環境で動作し, 標準的な C++と MPI によって実装されている. 特に, C++の言語機能のみを用いたライブラリであり, ユーザは C++に慣れていれば新しい言語や言語拡張を習得する必要がない.
- 構成的アルゴリズム論と呼ばれる理論に基づいて定式化されたデータ並列計算を表現する並列スケルトンを提供する. 特に, これらの並列スケルトンは構成的アルゴリズム論の理論に基づいて定式化されている. リスト (一次元配列) だけでなく, 行列 (二次元配列) [7], 木 [14] に対する並列スケルトンを提供する.
- データの分散/収集やプロセッサ間の通信は並列スケルトンの実装に隠蔽されている. そのため, ユーザは逐次プログラムと同じようにプログラミングを行うことができる.

並列スケルトンライブラリ「助っ人」の前のバージョンを 2007 年 1 月に公開したが, その後いくつかの問題点や改良点がでてきたため, 現在, 新しいバージョンのライブラリを作成している. 特に, 新しいバージョンでは次の 2 点が大きく改良されている.

インターフェイス これまでの「助っ人」の実装ではテンプレートや関数オブジェクトなどの C++の機能を利用していたが, 変数やメモリの確保/解放についてユーザがある程度気にする必要があった. 新しい実装では, これらは自動的に行われるようした. また, 効率のために用意されていた複数の関数について, 基本的に 1 つのものを利用できるように改良した.

融合変換による最適化 これまでの「助っ人」の実装では, OpenC++と呼ばれる C++のメタプログラミング言語を用いて融合変換による最適化を行っていた. しかし現在, OpenC++はメンテナンスされていないためこの機能が利用しにくくなっていた. そこで, 融合変換による最適化を行う対象とその方法について再考し, より現実的な範囲の最適化を式テンプレートというプログラミングテクニックにより実現した.

本論文では, 新しいバージョンで改良された点を中心に, リスト (一次元配列) に対する並列スケルトンに限定して「助っ人」の設計と実装を示す. 本論文の構成は次のとおりである. 2 節では,「助っ人」における並列リストスケルトンの定義を示し, 融合変換による効率化をどの範囲で行うべきかを議論する. 3 節では, 並列リストスケルトンが実際にどのように実装されているかを説明する. 4 節では, 2 つの例を用いた実験の結果を示し,「助っ人」の性能についての議論する. 5 節で関連研究を述べ, 最後に 6 節で本論文をまとめる.

2 「助っ人」におけるリストスケルトンライブラリの設計

「助っ人」で提供される並列スケルトンは, Bird-Meertens Formalism (BMF) [4] と呼ばれる枠組みで定義されている 4 つの計算パターンを元にしている [2]. 本節では, その 4 つの並列スケルトンについて, ユーザの観点からの逐次的な定義と実装の観点からの並列計算の定義とを与える. また, スケルトンプログラムの効率化を行うために融合変換をどのようにして組み込むかについて議論を行う.

[1] 英語名は, Skeleton Library in Tokyo の下線部より名前がとられている.

[2] もともとの (逐次の) BMF の枠組みでは十分な記述力がある. しかし, 並列計算の枠組みとした際に, リストの長さが変化するような計算などで効率の問題が起こる. その効率の問題を解決する理論と実現に関する研究は, 現在進行中である.

2.1 リストスケルトンの定義 (逐次)

まず, 「助っ人」が提供する 4 つの重要な並列リストスケルトン, map, zip, reduce, scan について, その逐次的な意味を説明する. これらの定義を図 1 に示す. 「助っ人」のユーザは, この逐次的な定義に基づいてプログラムを作成すれば良い.

並列スケルトン map と zip は, 各要素について独立した計算を表現する. 並列スケルトン map は, 関数とリストを受け取り, その関数をリストの各要素に適用したリストを返す. 並列スケルトン zip は, 関数と同じ長さの 2 つのリストを受け取り, 2 つのリストの対応する各要素に対して関数を適用して得られるリストを返す.

並列スケルトン reduce は, 結合的な二項演算子とリストを受け取り, リストの要素をその二項演算子によって畳み込んだ結果の値を返す. 並列スケルトン scan は, reduce 同様に結合的な二項演算子とリストを受け取るが, リストの先頭から畳み込んでいく途中結果を表現するリストを返す. リストの先頭要素は二項演算子の単位元とし, リストの最終要素は scan の計算では利用されない [3].

2.2 リストスケルトンの定義 (並列)

次に, リストスケルトンがどのように並列に計算されるかについて説明する. 「助っ人」では分散メモリ並列計算機を対象としているので, 複数のプロセスが独立した値を持った計算モデルを考える. プロセスのリストとデータのリストを区別するためにプロセスのリストの表記法を導入する. p 個のプロセスがそれぞれデータ $a_1, a_2, \ldots, a_p$ を持つとき, $\langle a_1, a_2, \ldots, a_p \rangle$ と記述する. ただし, すべてのプロセスで同じ値を持つ場合には, 括弧 $\langle\rangle$ を省略する.

「助っ人」では, リストはブロック分割されてプロセスに割り当てられる. すなわち, n 個の要素からなる大きなリスト $a_1, a_2, \ldots, a_n$ は, プロセス数 p だけの連続した小さなリスト $\langle [a_1, \ldots, a_{n/p}], \ldots, [a_{(p-1)*n/p+1}, \ldots, a_n] \rangle$ に分割される. (要素数がプロセス数で割り切れない場合には適切に振り分けられる.)

リストスケルトンの並列計算は, 各プロセスで独立に行われる計算 (ローカル計算) と, プロセス間で行われる計算 (グローバル計算) とを組み合わせることで実現される. 図 2 に, リストスケルトンの並列計算の定義を示す.

並列スケルトン map と zip は, それぞれ要素ごとの独立した計算であるのでローカル計算で map や zip を行えば良い. 並列スケルトン reduce は, まずローカル計算として各部分リストについて畳み込みの計算を行い, 次に得られた結果をプロセス間で畳み込む. 並列スケルトン scan の計算は 3 ステップからなる: (1) 各プロセスの持つ部分リストに対してローカルに scan を行い, 同時に畳み込みの結果を求める, (2) 畳み込みの結果に対して, グローバルに scan を行う, (3) 各プロセスの持つローカルな scan の結果に対して, グローバルな scan の結果を追加で適用する.

なお, 並列スケルトン scan の実装は, ローカル reduce, グローバル scan, ローカル scan という別の 3 ステップの方法がある. この実装は, リストの要素への書き込み回数が少ないという利点があるが, 後に説明する融合変換による効率化が難しくなってしまう. そのため, 新しい「助っ人」の実現では, 基本的には図 2 に示した 3 ステップの実装法をとった.

2.3 融合変換の適用範囲

スケルトン並列プログラミングのように計算パターンを組み合わせてプログラミングを行う手法では, 小さな部品ごとに処理を組み立てていくことができるという利点がある一方, 中間データの生成や処理の呼び出しのオーバーヘッドがあるという欠点もある. 融合変換は, このようなオーバーヘッドを除去する重要な手法である. 並列スケルトンに対する融合変換に関して, これまでにも複数の研究がある [2,8,10].

Aldinucci ら [2] は, 連続する複数の並列スケルトンを別の (複数の) スケルトンに置き換える変換ルールに基づくフレームワークを提案している. 複数の map を 1 つに融合するといった簡単なルールだけでなく, 連続する scan と reduce

[3] この定義 (prescan とも呼ばれる) は, BMF での scan と少し異なる. 「助っ人」では, 単位元を先頭に追加せず最終要素までの計算結果を含む定義 (postscan とも呼ばれる) も提供している.

$$\mathsf{map}(f, [a_1, a_2, \ldots, a_n]) = [f(a_1), f(a_2), \ldots, f(a_n)]$$
$$\mathsf{zip}(f, [a_1, a_2, \ldots, a_n], [b_1, b_2, \ldots, b_n]) = [f(a_1, b_1), f(a_2, b_2), \ldots, f(a_n, b_n)]$$
$$\mathsf{reduce}(\oplus, [a_1, a_2, \ldots, a_n]) = a_1 \oplus a_2 \oplus \cdots \oplus a_n$$
$$\mathsf{scan}(\oplus, [a_1, a_2, \ldots, a_n]) = [\iota_\oplus, a_1, \ldots, a_1 \oplus a_2 \oplus \cdots \oplus a_{n-1}]$$

図 1: 並列スケルトンの逐次的な定義. ただし, $\iota_\oplus$ は二項演算子 $\oplus$ の単位元を表す.

$$\mathsf{map}(f, \langle as_1, \ldots, as_p \rangle) = \textbf{let for each } i \in [1..p]\text{: } bs_i = \mathsf{map}_l(f, as_i) \textbf{ in } \langle bs_1, \ldots, bs_p \rangle$$
$$\mathsf{zip}(f, \langle as_1, \ldots, as_p \rangle, \langle bs_1, \ldots, bs_p \rangle) = \textbf{let for each } i \in [1..p]\text{: } cs_i = \mathsf{zip}_l(f, as_i, bs_i) \textbf{ in } \langle cs_1, \ldots, cs_p \rangle$$
$$\mathsf{reduce}(\oplus, \langle as_1, \ldots, as_p \rangle) = \textbf{let for each } i \in [1..p]\text{: } b_i = \mathsf{reduce}_l(\oplus, as_i) \textbf{ in } \mathsf{reduce}_g(\oplus, \langle b_1, \ldots, b_p \rangle)$$
$$\mathsf{scan}(\oplus, \langle as_1, \ldots, as_p \rangle) = \textbf{let for each } i \in [1..p]\text{: } bs_i = \mathsf{scan}_l(\oplus, as_i); c_i = \mathsf{reduce}_l(\oplus, as_i); \langle d_1, \ldots, d_p \rangle = \mathsf{scan}_g(\oplus, \langle c_1, \ldots, c_p \rangle) \textbf{ for each } i \in [1..p]\text{: } es_i = \mathsf{map}_l((d_i \oplus), bs_i) \textbf{ in } \langle es_1, \ldots, es_p \rangle$$

図 2: リストスケルトンの並列計算の定義. 定義にあたって, 図 1 のリストスケルトンの逐次的な定義を利用している. 区別しやすくするため, ローカル計算として利用される場合は添字 l を, グローバル計算として利用される場合は添字 g を付けた.

をある条件のもとで 1 つの reduce に変換するといったルールも扱われているが, そのようなルールは数が膨大になってしまうため全てを実装するのは困難である. 既存の「助っ人」の実装では, Hu ら [10] によって提案されたデータ生成/消費に着目した標準形による融合変換を実装していた [16]. しかしこの方法では, 融合した結果が並列スケルトンとして提供されていないようなものとなる可能性があり, 融合変換が適用できない場面もあった.

そこで, 新しい「助っ人」の実装では, より現実的な融合変換を実装することにした. そのアイデアは, ローカルの計算部分だけ融合するというものである. 実際, これまでの「助っ人」の融合変換によって効率化されるのは, ローカルな計算部分がほぼ全てであった. そこで, 複数の並列スケルトン全体を融合するのではなく, グローバルな計算ではさまれたローカルな計算部分だけに対して融合変換を適用する. 前節で示したスケルトンの並列計算の定義より, グローバルな計算ではさまれたローカルな計算部分は, map や zip という要素ごとに独立した計算がいくつか行われた後, reduce もしくは scan によるリストの走査が行われる (行われないこともある). このようなローカルな計算部分の独立した計算とその後の走査は比較的容易に融合することができる. 融合変換を行う実装については, 3.3 節にて示す.

なお, このリストスケルトンに対する融合変換は, コンパイラなどの分野ではループ融合として知られている最適化手法である. 残念ながら, 使用するレジスタを圧迫してしまう場合など, 必ず効率が良くなるという保証はないが, 多くの場合に効率が向上するので現在の実装では適用できる場合には融合変換を必ず適用するようになっている.

3 「助っ人」におけるリストスケルトンライブラリの実装

「助っ人」のリストスケルトンライブラリでは, 分散されたリスト (一次元配列) はテンプレートクラス `dist_list` を通じて提供される.

```
#include <iostream>
using namespace std;
const int n = 10000000;

int main(int, char**) {
  int *as = new int[n];
  double ave = 0;
  for (int i = 0; i < n; ++i) {
    as[i] = i*i*i*i*i % 100;
    ave += as[i];
  } ave /= n;

  double var = 0;
  for (int i = 0; i < n; ++i) {
    var += (as[i]-ave) * (as[i]-ave);
  } var /= n;

  cout << var << endl;
  delete [] as;
}
```

図 3: 単純な for ループによるプログラム

また, この分散されたリストを操作する並列スケルトンは, 名前空間 `sketo::list_skeletons` にて定義されている.

具体的な実装の方法について示す前に, プログラムの例を示しておく. 例として, n 個の値 $[a_0, \ldots, a_{n-1}]$ (ただし, $a_i = i^5 \bmod 100$ とする) の分散 var を,

$$ave = \sum_{i=0}^{n-1} a_i/n$$
$$var = \sum_{i=0}^{n-1} (a_i - ave)^2/n$$

という定義に沿って求める問題を使う. 図 3 にはこの問題を解く単純な for ループによるプログラムを, 図 4 にはこの問題を解く助っ人によるプログラムを, 図 5 にはこの問題を解く STL によるプログラムを示す.

3.1 分散データ構造

これまでの分散リストの実装と同様に, `dist_list` クラスのコンストラクタなどの中で, 操作の対象リストは均等な要素数となるようにブロック分割され各プロセスに分配される. ユーザはリストがどのように分配されるかについて意識する必要はない.

これまでの実装から大きく変わった点は, 分散リストの実体を `dist_list` クラスから分離して管理するようにした点である. 分散リストを効率的に操作するため, 分散リストの実体 (メモリの確保) は別のテンプレートクラス `dist_list_buffer` で行い, クラス `dist_list`

```
#include <iostream>
#include <sketo/sketo.h>
#include <sketo/list_skeletons.h>
const int n = 10000000;

using namespace std;
using namespace sketo;
using namespace sketo::list_skeletons;

struct gen
  : public functions::base<int (int)> {
  int operator()(int i) const {
    return i*i*i*i*i % 100;
  }
};

int sketo::main(int, char**) {
  dist_list<int> as;
  as = generate(n, gen());
  double ave
    = reduce(plus<double>(), as) / n;

  double var
    = reduce(plus<double>(),
        map(functions::square<double>(),
          map(bind2nd(minus<double>(), ave),
            as))) / n;

  sketo::cout << var << endl;
}
```

図 4: 助っ人を用いたプログラム

は実体へのポインタだけを持つようにしてある. クラス `dist_list_buffer` が確保されたメモリの他にリファレンスカウンタを持つことで, 不必要なメモリの確保や解放が行われないようなっている.

例えば, 次のコードを考える.

```
1.   dist_list<int> as(n), bs(n);
2.   {
3.     dist_list<int> cs = as;
4.     as = bs;
5.   }
```

1 行目で 2 つの分散リストの実体が確保され, それぞれのリファレンスカウンタが 1 となる. 3 行目では `cs` は `as` と同じ実体を指し, リファレンスカウンタが 2 となる. 4 行目の時点で, `as` は `bs` と同じ実体を指すようになり, `cs` の指している (もともと `as` であった) 実体のリファレンスカウンタは 1 となる. 5 行目で `cs` がスコープから抜ける時, `cs` の指す実体のリファレンスカウンタが 0 となり解放され, `as` と `bs` の両方が指している 1 つの実体だけが残ることになる. このように, 必要な分散リストの実体だけが確保

```
#include <iostream>
#include <vector>
#include <functional>
#include <algorithm>
#include <numeric>
using namespace std;

const int n = 10000000;

struct gen {
  mutable int index;
  gen() : index(0) {};
  double operator()() const {
    const int i = index++;
    return i*i*i*i*i % 100;
  }
};

struct sqr {
  double operator()(double x) const {
    return x * x;
  }
};

int main(int, char**) {
  vector<double> as(n);

  generate(as.begin(), as.end(), gen());
  double ave
    = accumulate(as.begin(), as.end(),
                 0.0, plus<double>()) / n;

  transform(as.begin(),as.end(),as.begin(),
            bind2nd(minus<double>(), ave));
  transform(as.begin(),as.end(),as.begin(),
            sqr());
  double var
    = accumulate(as.begin(), as.end(),
                 0.0, plus<double>()) / n;

  cout << var << endl;
}
```

図 5: STL を用いたプログラム

されることになり, またユーザは特に意識することなく扱うことができるわけである.

この仕組みは, 並列スケルトンから返される分散リストを変数に代入する際にもうまく働くようにしている. 例えば,

```
as = map(f, as);
```

というコードがあった時, as の指す実体のリファレンスカウンタが 1 であれば, map の計算結果は as の指す実体に上書きされる. さらに, 左辺の変数と右辺の変数が違う場合であっても, 実体への上書きが可能であれば行うようにしている.

3.2 並列スケルトン

並列スケルトンのインターフェイスはこれまでのものから大きくは変更されていない.

並列スケルトンが入力として受けとる関数には, 関数オブジェクトを用いる. 関数ポインタではなく関数オブジェクトを利用することで, 引数として渡される処理をコンパイラが具体的に知ることができ, インライン展開によって関数呼び出しのオーバーヘッドが削減できる. 特に, 小さな処理を行うスケルトンを複数組み合わせてプログラムを行う場合には, このインライン展開による効率化がよく効く.

スケルトンに渡される関数オブジェクトは, テンプレートクラス `sketo::functions::base` の 1 つを継承し, 引数や返り値などの型をコンパイラに教える. このテンプレートクラス `sketo::functions::base` は, Boost ライブラリの `boost::function` を参考に新たに定義したものである. 新たに定義した理由は, `boost::function` を使うとうまくインライン展開できなかったためである. 同じ理由で, Boost ライブラリの `boost::lambda` なども効率の点で問題がある. STL の`<functional>`にて提供されている関数オブジェクトについてはインライン展開されるので, 問題なく「助っ人」でも利用可能である.

これまでの「助っ人」では, それぞれのスケルトンについて上書きを行うために特殊化された実装が与えられていた. 例えば, 入力のリストに上書きを行う map スケルトンとして `map_ow` などが提供されていた. 新しい実装では, 上記の分散データ構造の中で説明したように, リファレンスカウンタを用いて分散リストに対する上書きなどを管理するようにした. これに伴い, 上書きを行うための特殊化が必要なくなり, その副産物として提供されている関数の数が減少したという利点を得た.

3.3 式テンプレートによる融合変換

新しい「助っ人」の実装では, 2.3 節にて議論した融合変換を, 式テンプレート [19] と呼ばれるプログラミングテクニックを用いて実現している.

まず, 式テンプレート [19] について簡単に説明する. 式テンプレートとは, 部分式を評価する際に, 通常通り値を計算するのではなく部分式を表現する構造情報を求めておき, 最終的に計算結果が要求される時 (代入の場合など) まで計算を遅延させるプログラミングテクニックである. 計算を遅延させることにより, 式全体に渡る効率の良いプログラムをテンプレートで生成することができるという利点がある.

式テンプレートは, 行列やベクトルに対する線形計算のライブラリなどにおいてよく用いられている. ここでは, 具体例としてベクトル (vec 型) の要素 A, B, C, D に対する次の計算を考える.

```
D = A + B - C;
```

演算子+を vec に対して単純にオーバーロードした場合, 右辺ではまず A + B の計算結果が一時変数として確保され, 次にそれと C の差が求められた後, 最終的に得られたベクトルが D に代入される. これに対して, 式テンプレートを用いた場合では, A+B の結果としてまず plus<vec,vec>型が作られ, 次に右辺全体を表現する minus<plus<vec,vec>,vec>型が作られ, 最終的には代入を表現するメンバ変数 vec::operator=(minus<plus<vec,vec>,vec>) が作られ呼び出される. このメンバ変数の中で, 適切な実装を行うことにより, 単純に演算子オーバーロードを行った場合に比べて効率の良い計算が得られることになる.

2.3 節では, 融合変換が適用できる場所とて, グローバルな計算に囲まれたローカルな計算部分であると述べた. しかし, C++の言語仕様の制約により, 実際には

- 1 つの式として記述された計算のうち,
- グローバルな計算に囲まれた, もしくはその外側の, ローカルな計算部分,

に対して融合変換が適用される.

実装では, ローカルな map, ローカルな zip, ローカルな generate を表現するような型を定義し, 計算を遅延させるようにしている. 例えば, 図 4 のプログラム中の

```
double var
  = reduce(plus<double>(),
      map(functions::square<double>(),
        map(bind2nd(minus<double>(), ave),
          as))) / n;
```

では, reudce の引数として

```
MapObj<G,MapObj<F,dist_list> >
```

(F は bind2nd(minus<double>(),ave) の型,
G は functions::square<double>() の型を表す)

という型の値が渡される. その後, reduce の計算の中で, 2 つの map の計算とローカルの reduce をひとつのループで行う. これにより, 上記の 3 つのスケルトンからなる式は, 実際には 1 つのループに相当する計算に置き換えられることになる.

この融合変換がどの程度効率に影響するかについては, 4 節にて議論する.

3.4 その他の工夫

前節で見たように, 式テンプレートは効率の良いプログラムを生成する上で重要なテクニックである. しかし, 式テンプレートによる融合変換を並列スケルトンライブラリに用いる場合には, 少し問題が発生した. それは, プログラムが間違っていた場合のエラーが非常に読み難いという問題である.

線形代数の計算を置き換える場合には, 使用する演算子が+や*など, プログラムを作成する上であまりミスを犯しにくいものであった. しかしながら, スケルトン並列プログラミングでは, 並列スケルトンが関数を引数にとるため, 計算の自由度が非常に高く, ミスがおきやすい. 例えば, 図 4 のコード中で, bind2nd の第 1 引数に間違えて一引数関数の square<double>() を渡してしまうなどのミスは起こりうるものであろう. しかし, この場合, たった 1 箇所のミスで 30 行ほどのエラーが表示され, そのうちの 1 行を取ってくると

```
dist_list.h: In member function ‘typename F::
result_type sketo::impl::dist_list_mapobj_loc
al_iterator<F, B>::operator*() const [with F
= sketo::functions::square<double>, BIT = ske
to::impl::dist_list_mapobj_local_iterator<std
::binder2nd<sketo::functions::square<double>
>, sketo::impl::dist_list_local_iterator<int>
>]’:
```

という非常に解読しにくいものとなってしまっている. 内部実装を知らなければ, このエラーがなぜに起こったのか判別は困難である.

したがって, このようなエラーが発生した時などのために, 式テンプレートを使わない単純なスケルトンライブラリとの切り替えを簡単に行うことができるようにした. 具体的には, プリプロセッサで__SKETO_NO_FUSION__というマクロを定義しておけば良い. 式テンプレートを使わない場合には, エラーの箇所を比較的簡単に見つけることができる.

4 実験

新しい「助っ人」の実装の性能評価として, 図 3, 4, 5 に示した分散の計算と, N-Queens 問題の解を数える計算とを用いて, 実験を行った.

まず, リストスケルトンの実装の台数効果やオーバーヘッドを調べるため, 分散の計算を用いて実験を行った. 実験では, 要素数を 200,000,000 とした. また実験に用いた計算機は, CPU に Xeon E5430 (2.66GHz, 4 コア) を 2 つ, メモリを 8GByte 持つものであり, コンパイラには GCC 4.4.0 の開発バージョン, MPI ライブラリとして mpich 1.2.7p1 を用いた. この実験の結果を, 表 1 と図 6 に示す. 凡例は, Loop は図 3 の for 文による単純なループを OpenMP によって並列化したもの, SkeTo は図 4 の助っ人によるもの, SkeTo NO は助っ人で融合変換を行わなかった場合, STL は図 5 の STL によるプログラムを GCC の libstdc++ parallel mode [18] にて並列化したものである.

この結果より, 式テンプレートによる融合変換によって, for ループによるプログラムと同じ程度まで効率化されることが分かる. 一方, 融合変換を行わない助っ人によるプログラムと STL のプログラムは, 何度もリストや配列を走査するオーバーヘッドが大きく, 効率の良い 2 つのプログラムに比べて 2〜3 倍程度遅くなってしまっている. なお, この実験において台数効果が十分に出ていない理由は, メモリアクセスが多いためにメモリアクセスの帯域を全て使ってしまっているからであると考える.

次に, リストスケルトンのスケーラビリティを調べるため, 18-Queens 問題を用いて実験を行った. 実験には, 「助っ人」のリストスケルトンで記述したプログラムと, Kise らによる MPI と C によるプログラム [12] (nq24) とを用いた.

表 1: 分散の計算に対する実験結果 (単位は秒)

コア数	1	2	4	8
Loop	1.10	1.11	0.614	0.505
SkeTo	2.38	1.25	0.676	0.495
SkeTo NO	3.93	2.25	1.43	1.26
STL	3.65	3.18	2.47	2.45

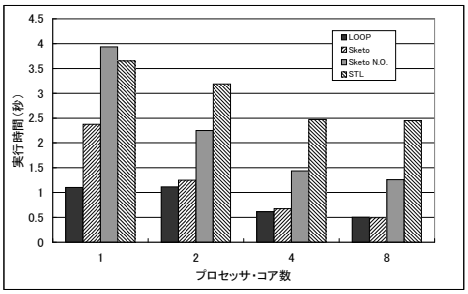

図 6: 分散の計算に対する実験結果

実験に用いた計算機は, Xeon 2.4GHz の CPU を 2 つとメモリを 2GByte 持つ各ノードが, ギガビットイーサネットで接続されたクラスタである. コンパイラは GCC 4.1.2, MPI ライブラリは mpich 1.2.7p1 である. 実験の結果を表 2 と図 7 に示す.

図 7 の結果よりいずれのプログラムとも良い台数効果を得ている. SkeTo を用いたプログラムは台数効果が Kise らによるプログラムと比べて少し悪くなっているが, これは, SkeTo が静的にデータ分配を行っているため, 1CPU の場合により速く, 逆に 32CPU の場合に負荷分散が不十分になっているためであろう. OpenMP や他の並列計算ライブラリのように, 動的な負荷分散などの複数のデータ分配法を提供することも効率を追求する上では重要となると考える.

以上の 2 つの実験により, 「助っ人」のリストスケルトンはオーバーヘッドが小さく, また比較的良い台数効果を得ることができていると言える.

5 関連研究

これまでにも, 様々な並列スケルトンライブラリが実現されてきている. 比較的新しいものの例をあげると, Muskel [1], eSkel [3], Muesli [13] などがある. Muskel や eSkel は, 主にタスク並

表 2: 18-Queens 問題に対する実験結果 (単位は秒)

CPU 数	1	4	16	32
SkeTo	554	157	39.9	20.9
nq24	596	149	37.3	18.7

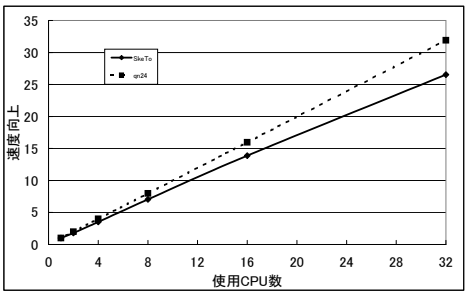

図 7: 18-Queens 問題の計算に対する実験結果

列計算のための並列スケルトンを提供する並列スケルトンライブラリである. Muesli は,「助っ人」の開発にあたって非常に参考にしたスケルトンライブラリである. Muesli は「助っ人」同様, リスト (一次元配列), 行列 (二次元配列) に対する並列スケルトンを提供する標準的な C++ によるライブラリであり, さらにタスク並列計算を実現するスケルトンも提供している. 逆に, Muesli と比較して「助っ人」は, 行列スケルトンの定式化 [7], 木スケルトン [14], 融合変換による最適化機構 [16] などの長所を持つ. 近年, Intel によって Intel Thread Building Blocks [17] が開発された. これを利用することで共有メモリ計算機に対して並列計算を比較的容易に開発ができるようになってきている.

一方, 新しいライブラリを実現するのではなく, 既存の標準テンプレートライブラリ STL を並列計算に対応したものに置き換えることで, STL を用いたプログラムを並列に計算できるようにするという方針の研究もある. そのようなものには, DatTel [5] や MCSTL [18] がある. 特に, 後者の MCSTL は libstdc++ parallel mode という名前で GCC に取り込まれている.

「助っ人」の新しい実装では, 融合変換機能を式テンプレートによって実現した. 式テンプレートは, 特に行列やベクトルなどの線形計算を効率良く実装する手法として広く利用されており, 代表的なものには Blitz++ [20] や Boost の uBLAS ライブラリ[4]などがある. また, 式テンプレートによる効率的な並列実装を行っているシステムとして NT2 [9] などがある.

6 まとめと今後の課題

本論文では, 並列スケルトンライブラリ「助っ人」におけるリストスケルトンライブラリの新しい設計と実装を示した. 並列スケルトンの定義については, ユーザ側からの直感的な定義に加えて, 並列計算のための定義を与え, その定義をもとに融合変換を行う範囲をプロセス毎に独立したローカルな計算と定めた. 実装に関しては, 分散リストの実体をリファレンスカウンタによって管理し, 式テンプレートによる融合変換を実装した. また実験により, 並列スケルトンを組み合わせて得られる並列プログラムは十分に効率の良いものであることを確認した.

今後の課題は, 並列スケルトンを拡充し表現力を向上することと, 融合変換機構のさらなる実装である. 現在の「助っ人」のリストスケルトンで扱えない, リストの長さが変わるような計算やソートなどの理論的にはまだ定式化ができていないが実用上重要な操作を取り込むことが必要であろう. また, 融合変換機構の実装については, Emoto ら [8] によって提案された, 近傍要素を利用するような計算パターンに対する最適化についても実装することは応用上大事であると考えている.

参考文献

[1] M. Aldinucci, M. Danelutto, and P. Dazzi. Muskel: an expandable skeleton environment. *Scalable Computing: Practice and Experience*, 8(4):325–341, 2007.

[2] M. Aldinucci, S. Gorlatch, C. Lengauer, and S. Pelagatti. Towards parallel programming by transformation: the FAN skeleton framework. *Parallel Algroithms and Applications*, 16(2–3):87–121, 2001.

[3] A. Benoit, M. Cole, S. Gilmore, and J. Hillston. Flexible skeletal programming with eSkel. In *Euro-Par 2005, Parallel Processing, 11th International Euro-Par Conference, Proceedings*, Vol. 3648 of *LNCS*, pp. 761–770. Springer, 2005.

[4] `http://www.crystalclearsoftware.com/cgi-bin/boost_wiki/wiki.pl?Effective_UBLAS`

[4] R. S. Bird. An introduction to the theory of lists. In *Logic of Programming and Calculi of Discrete Design*, Vol. 36 of *NATO ASI Series F*, pp. 5–42. Springer, 1987.

[5] H. Bischof, S. Gorlatch, and R. Leshchinskiy. Generic parallel programming using C++ templates and skeletons. In *Domain-Specific Program Generation, International Seminar, Revised Papers.*, Vol. 3016 of *LNCS*, pp. 107–126. Springer, 2004.

[6] M. Cole. *Algorithmic Skeletons: Structural Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. MIT Press, 1989.

[7] K. Emoto, Z. Hu, K. Kakehi, and M. Takeichi. A compositional framework for developing parallel programs on two-dimensional arrays. *International Journal of Parallel Programming*, 35(6):615–658, 2007.

[8] K. Emoto, K. Matsuzaki, Z. Hu, and M. Takeichi. Domain-specific optimization strategy for skeleton programs. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Proceedings*, Vol. 4641 of *LNCS*, pp. 705–714. Springer, 2007.

[9] J. Falcou, J. Sérot, L. Pech, and J.-T. Lapresté. Meta-programming applied to automatic smp parallelization of linear algebra code. In *Euro-Par 2008 - Parallel Processing, 14th International Euro-Par Conference, Proceedings*, Vol. 5168 of *LNCS*, pp. 729–738. Springer, 2008.

[10] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *Programming Languages and Systems, 11th European Symposium on Programming, ESOP 2002, Proceedings*, Vol. 2305 of *LNCS*, pp. 83–97. Springer, 2002.

[11] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating efficient parallel programs. In *Proceedings of the 1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pp. 85–94. University of Aarhus, 1999. Technical report BRICS-NS-99-1.

[12] K. Kise, T. Katagiri, H. Honda, and T. Yuba. Solving the 24-queens problem using MPI on a PC cluster. Technical Report UEC-IS-2004-6, Graduate School of Information Systems, The University of Electro-Communications, 2004.

[13] H. Kuchen. A skeleton library. In *Euro-Par 2002, Parallel Processing, 8th International Euro-Par Conference Paderborn, Proceedings*, Vol. 2400 of *LNCS*, pp. 620–629. Springer, 2002.

[14] K. Matsuzaki. Efficient implementation of tree accumulations on distributed-memory parallel computers. In *ICCS 2007: 7th International Conference, Proceedings, Part II*, Vol. 4488 of *LNCS*, pp. 609–616. Springer, 2007.

[15] K. Matsuzaki, H. Iwasaki, K. Emoto, and Z. Hu. A library of constructive skeletons for sequential style of parallel programming. In *InfoScale '06: Proceedings of the 1st international conference on Scalable information systems*, Vol. 152 of *ACM International Conference Proceeding Series*. ACM Press, 2006.

[16] K. Matsuzaki, K. Kakehi, H. Iwasaki, Z. Hu, and Y. Akashi. A fusion-embedded skeleton library. In *Euro-Par 2004 Parallel Processing, 10th International Euro-Par Conference, Proceedings*, Vol. 3149 of *LNCS*, pp. 644–653. Springer, 2004.

[17] J. Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism*. O'Reilly Media, Inc, 2007.

[18] J. Singler, P. Sanders, and F. Putze. The multi-core standard template library. In *Euro-Par 2007, Parallel Processing, 13th International Euro-Par Conference, Proceedings*, Vol. 4641 of *LNCS*, pp. 682–694. Springer, 2007.

[19] T. L. Veldhuizen. Expression templates. *C++ Report*, 7(5):26–31, 1995. Reprinted in *C++ Gems* edited by Stanley Lippman.

[20] T. L. Veldhuizen. Arrays in Blitz++. In *Proceedings of the 2nd International Scientific Computing in Object-Oriented Parallel Environments (ISCOPE'98)*, Vol. 1505 of *LNCS*, pp. 223–230. Springer, 1998.