# 分散メモリ型並列計算機上での木に対する並列スケルトンの実現

Efficient Implementation of Parallel Tree Skeletons on Distributed Systems

松崎 公紀 † 胡 振江 †‡ 武市 正人 †
Kiminori MATSUZAKI Zhenjiang HU Masato TAKEICHI

† 東京大学大学院 情報理工学系研究科 数理情報学専攻
Dept. of Mathematical Informatics, Graduate School of
Information Science and Technology, University of Tokyo
{ki-matsuzaki-7,hu,takeichi}@mist.i.u-tokyo.ac.jp

‡ さきがけ研究 21 科学技術振興事業団
PRESTO 21, JST

データ構造 “木” は, 式の解析などで用いる構文木や, XML などの構造をもった文書などで使用される有用なデータ構造であるが, 構造が不規則・不均等になりうるために, 均衡がとれていない木は簡単には並列プログラムに組み込むことができなかった. 本稿では, “木” に対する並列スケルトンについて, 分散システム上でも効率よく実行できる実装法を提案する. 本稿の実装法では, $m$-bridge を利用して木を分割し, 分割された先の値を表す変数を含む式の形で計算することにより, 並列に計算を行うことが可能になっている. また, 実験を行い, 均衡のとれていない木に対しても効率性が確認された.

## 1 はじめに

近年では, PC クラスタなど分散メモリ型の並列計算機を利用できる環境が身近になってきている. しかし, 並列に効率よく動作するプログラムを作成することは非常に難しい仕事である. なぜなら, 効率のよい並列プログラムを作成するためには, 資源の配置・プロセッサ間の通信・処理の同期など, 背後にあるアーキテクチャを意識してユーザがプログラミングを行う必要があるからである. さらに, 分散メモリ型並列計算機では通信時間が大きいという問題があるので, 効率のよい実装のためには, 資源をうまく配置してプロセッサ間通信のコストを減らすことが必要で, より並列プログラミングが難しいものになっている.

これらの困難を解決する一つの手法として, 「並列スケルトンプログラミング」が提案された [3]. 並列スケルトンプログラミングは, 並列性を持つスケルトンを組み合わせることで, プログラミングを行う手法である. これにより, ユーザは背後にあるアーキテクチャをそれほど意識する必要がないという利点がある.

これまで, 並列スケルトンに関する研究は, 主にデータ構造 “リスト” に対して行われてきた [2]. リストはよく利用される, 非常に規則的なデータ構造である. このリストの上での並列スケルトンとして, 基本的な 3 つの並列スケルトン map, reduce, scan が研究されてきた. また, 並列スケルトンの適用性の問題を解決するため, これらの基本的な並列スケルトンを組み合わせてより広い範囲の問題に適用することができる, 逆融合変換 [1] なども提案されている.

“木” は, 式の解析などで用いる構文木や, XML などの構造をもった文書などで使用される有用なデータ構造である. しかし, “木” は構造が不規則・不均等になりうるために, 均衡がとれていない木に対しては, これまで簡単に並列プログラムに組み込むことができなかった. データ構造 “木” に対する並列計算の研究としては, Tree Contraction が過去に提案されている [4, 5]. しかし, Tree Contraction は主に共有メモリ型並列計算機を対象とした実装しかない.

本稿では, “木” に対する並列スケルトンについて, 分散システム上でも効率よく実行できる実装法を提案する. 本稿の実装法では, $m$-bridge を利用して木を分割し, その木の分割に基づいて並列スケルトンの計算を行う. その際, 分割された先の値を変数とし, その変数を含む式の形のまま計算することにより, 各プロセッサが並列に計算を行うことが可能になっている. また, 実験を行い, 均衡のとれていない木に対しても効率よく計算されることが確認された.

$$
\begin{array}{lcl}
map\ f_L\ f_N\ (Leaf\ n) & = & Leaf\ (f_L\ n) \\
map\ f_L\ f_N\ (Node\ n\ l\ r) & = & Node\ (f_N\ n)\ (map\ f_L\ f_N\ l)\ (map\ f_L\ f_N\ r)
\end{array}
$$

$$
\begin{array}{lcl}
reduce\ f\ (Leaf\ n) & = & n \\
reduce\ f\ (Node\ n\ l\ r) & = & f\ n\ (reduce\ f\ l)\ (reduce\ f\ r)
\end{array}
$$

$$
\begin{array}{lcl}
uAcc\ f\ (Leaf\ n) & = & Leaf\ n \\
uAcc\ f\ (Node\ n\ l\ r) & = & Node\ (f\ n\ (root\ lt)\ (root\ rt))\ lt\ rt \\
 & & \quad \textbf{where}\ \ lt = uAcc\ f\ l \\
 & & \qquad\qquad\ \ rt = uAcc\ f\ r
\end{array}
$$

$$
\begin{array}{lcll}
dAcc\ f_N\ f_L\ f_R\ (Leaf\ n)\ acc & = & Leaf & (f_N\ n\ acc) \\
dAcc\ f_N\ f_L\ f_R\ (Node\ n\ l\ r)\ acc & = & Node & (f_N\ n\ acc) \\
 & & & (dAcc\ f_N\ f_L\ f_R\ l\ (f_L\ n\ acc)) \\
 & & & (dAcc\ f_N\ f_L\ f_R\ r\ (f_R\ n\ acc))
\end{array}
$$

図 1: 木に対する 4 つの並列スケルトンの定義

## 2 木に対する並列スケルトンプログラミング

二分木に対する重要な 4 つの並列スケルトンとして map, reduce, upwards accumulate, downwards accumulate が提案されている [5]. これらの形式的な定義を図 1 に示し, 以下では直感的な意味を説明する.

**map** $map\ f_L\ f_N$ はすべての葉, 内部ノードに対して, それぞれのノードの情報だけから計算することができる関数 $f_L, f_N$ を適用する操作である.

**reduce** $reduce\ f$ はノードの値, 左の部分木の値, 右の部分木の値から計算される関数 $f$ をボトムアップに適用して, 木を 1 つの値にまとめる操作である.

**upwards accumulate** $uAcc\ f$ は reduce 同様に関数 $f$ をボトムアップに適用して, 途中の結果を残した木を生成する操作である. したがって, 部分木の値をボトムアップに累積させた木を求めることになる.

**downwards accumulate** $dAcc\ f_N\ f_L\ f_R$ は根から各ノードまでの経路に沿って, 各ノードには $f_N$, 左の子には $f_L$, 右の子には $f_R$ をトップダウンに適用し, 途中の結果を残した木を生成する操作である.

この 4 つの並列スケルトンを用いたプログラムとして, 構文木上の計算の例を挙げて説明する.

葉には実数, 内部ノードには二項演算子 $+$, $-$, $\times$ のいずれかが入っている構文木があったとする. この構文木から, 部分式を評価したときにその結果が閾値 $\theta$ を超える部分式の個数を求めたいとする. このような並列プログラムは並列スケルトンを用いて次のように簡潔に書ける.

$$
\begin{array}{rl}
 & reduce\ sum3 \circ map\ (fil\ \theta) \circ uAcc\ eval \\
\textbf{where} & sum3\ n\ l\ r = n + l + r \\
 & fil\ \theta\ n = (n > \theta)\ ?\ 1\ :\ 0 \\
 & eval\ \text{'}+\text{'}\ l\ r = l\ +\ r \\
 & eval\ \text{'}-\text{'}\ l\ r = l\ -\ r \\
 & eval\ \text{'}\times\text{'}\ l\ r = l\ \times\ r
\end{array}
$$

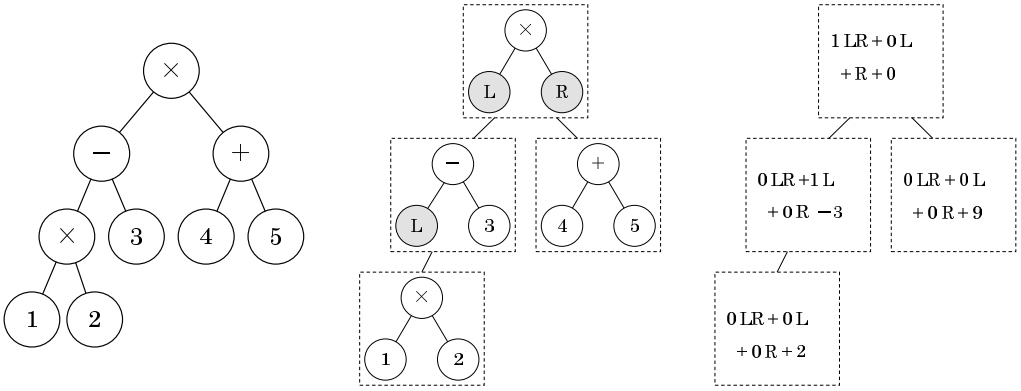
まず, 途中の結果を残して部分式を評価し, 次に, 閾値 $\theta$ を超えるものを 1 にする. 最後に和をとることで個数を求めることができる.

## 3 木の分割

分散メモリ型の並列計算機ではプロセッサ間通信のコストが高いため, データをどのように分配させるかが重要な問題の一つである. データ構造 “木” を分割する際には, ノード数ができるだけ均等になるだけではなく, それらの各プロセッサに割り当てられたノードがつながっているという大きく 2 つの性質を満たすことが望まれる.

このような性質を満たす木の分割として, $m$-bridge [4] を利用して分割する. この $m$-bridge を利用した木の分割には次のような利点がある.

- 各プロセッサに割り当てられるノードがつながっている. さらに, もとのデータが二分木であるとき, この二分木を分割した際のプロセッサ間の関係もまた二分木となる. この性質により, プログラミングが容易になる.
- 分割された各部分木のノード数は, どのような二分木に対しても $4n/p$ ($n$ はノード数, $p$ はプロ




(a) 木の初期状態　(b) 木を分割した結果　(c) 内部で reduce を行った結果

図 2: reduce の実装: 構文木を評価する関数の計算過程

セッサ数) を超えない. この性質により, 実装の効率性が保証される.

実装では, 分割された部分木同士の関係 (すなわち, プロセッサ間の関係) と, 子の部分木がどこに接続していたかを示すフラグ $L$, $R$ を用いることで木の分割を表現している.

本実装における木の分割の例を図 2 (a), (b) に示す.

## 4 木の分割に基づいた並列スケルトンの実現

本章では, 第 3 章のように分割された木に対して, 基本的な 4 つの並列スケルトン map, reduce, upwards accumulate, downwards accumulate の実装を示す.

reduce, upwards accumulate, downwards accumulate の各計算は, 子または親の値が求まらないと計算できないため, そのままでは各プロセッサが並列に計算を行うことができない. 本手法では, 各部分木の計算において Tree Contraction[4, 5] で用いられている考え方を利用することで, この問題を解決した. 分割された先の計算結果を変数とし, 各計算をこの変数を用いた式を用いて行うことで, 各プロセッサが並列に計算を行うことができる.

これらの並列スケルトンの計算は, 大きく次の 3 つの段階から構成される.

1. **前プロセッサ内計算**: 分割された各部分木の内部の計算を, 各プロセッサ上で行う. この計算では, 分割された先の値を変数として, この変数を用いた式を求める. この計算では, プロセッサ間通信を行わない.
2. **プロセッサ間計算**: 部分木同士の関係に基づいて, プロセッサ間で通信を行い, 1. で求めた式に含まれる変数に値を代入していくことで木全体の計算を行う.
3. **後プロセッサ内計算**: 2 によって, 分割された先の値が求まっているので, この値を用いて各部分木の内部を再計算する.

本稿では, reduce の実装についてその詳細を述べ, 残りの 3 つのスケルトンについては概要のみ示す.

### 4.1 reduce の実装

分割された木に対する reduce は次の 2 つの段階から計算される. ここでは, 構文木を評価する関数の例 (図 2) を用いて説明する.

1. **部分木の内部の reduce**

各部分木について, その内部を reduce する. 分割された先の計算結果はこの段階では分からないため, それらを 2 つの変数 $L, R$ で表し, これらの変数を含む式として計算する. この計算は, 各プロセッサが並列に行うことができ, $m$-bridge の分割の性質より, この計算時間はノード数 $n$, プロセッサ数 $p$ の時に, $O(n/p)$ となる.

構文木の例では, 部分木の内部を reduce すると, $aLR + bL + cR + d$ (ただし, $a, b, c, d$ は実数) と表現することができ, 計算結果は図 2 (c) のようになる.

表 1: 並列スケルトンの計算量

| | 部分木の計算 | 通信コスト |
|---|---|---|
| map | $O(n/p)$ | 0 |
| reduce | $O(n/p)$ | $O(\log p)$ |
| upwards | $O(n/p)$ | $O(\log p)$ |
| downwars | $O(n/p)$ | $O(\log p)$ |

$n$:木のノード数, $p$:プロセッサ数

2. **木全体の reduce**

ボトムアップに, 1 で求めた各部分木の式の変数 $L, R$ に子の値を代入して計算を行う. これにより求める値を得ることができる. この計算には逐次では $O(p)$ で, Tree Contraction を用いると $O(\log p)$ の計算時間で行うことができる.

### 4.2 その他の並列スケルトンの実装

本節では, 残りの 3 つの並列スケルトンの実装法について, その概要を示す.

map は各ノードが独立に計算される. したがって, 各プロセッサが独立に各部分木に対して map の計算を行うことで計算される.

upwards accumulate は reduce に似た操作であり, reduce の計算の後, さらに部分木の内部で upwards accumulate することで求める木を得ることができる.

downwards accumulate では, ルートの親のノードがわからないので, それを変数でおいて, upwards accumulate と同様に計算することで, 求める木を得ることができる.

本実装の場合の計算量は表 1 の通りである. 本実装の並列スケルトンの適用条件は Tree Contraction のそれと同じである.

## 5 実験と考察

本稿の実装法の有効性の検証として, 第 2 章で例として挙げた並列スケルトンプログラムを実際に C++ と MPI ライブラリを用いて実装し, 実験を行った. 1,000,000 ノードからなる木に対する実験結果は図 3 のようになった. 横軸はプロセッサ数, 縦軸は 1 プロセッサに対する相対時間である. また, 実線は実験結果, 下の破線は理想的な場合の計算時間の上限 $(n/p)$, 上の破線は理想的な場合の計算時間の上限の 4 倍の値 $(4n/p)$ である.

$m$-bridge による木の分割はノード数が $4n/p$ を超えることはなく, 実際にグラフでも理想的な場合の計算時間の上限の 4 倍 $(4n/p)$ より良い結果となっている. したがって, 全体的には台数効果が得られていると結論付けることができる.

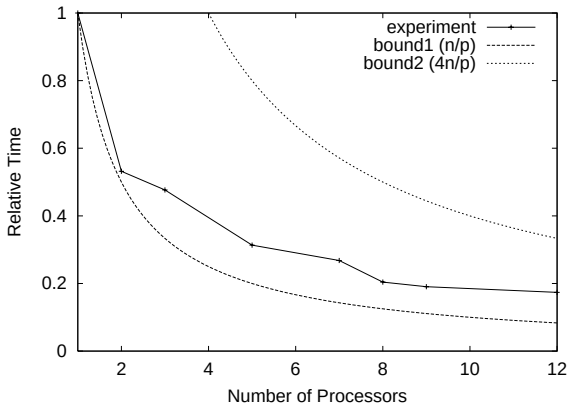



図 3: 実行結果

## 6 まとめと今後の課題

本稿では, データ構造 “木” に対する並列スケルトンについて, 分散システムでも効率よく計算することが可能な実装法を提案した. 特に, 本稿の実装法では, 木が均衡がとれていない場合でも効率よく計算することができる. また, 実際に並列スケルトンプログラムを実装して有効性の検証も行った.

今後の課題としては, 逐次再帰プログラムから並列プログラムを導出する手法 [1] を木に拡張すること, 実装をコンパイラに組み込んでより容易に並列スケルトンを利用できるようにすることを考えている.

### 参考文献

[1] Z. Hu, M. Takeichi, H. Iwasaki, Diffusion: Calculating Efficient Parallel Programs, *1999 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*. San Antonio, Texas, January 22-23, 1999, pp. 85-94.

[2] R. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design,* pages 5–42. Springer-Verlag, 1987.

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

[4] M. Reid-Miller, G. L. Miller, and F. Modugno. *Synthesis of Parallel Algorithms.* Morgan Kaufmann Publishers, 1996.

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