ブラックボックステストによる軽量なリダクションループ自動並列化

森畑 明昌　佐藤 重幸

リダクションループは多数のデータを少数の結果へと集約する処理を表し、計数・平均値・最大値などの計算に広く用いられている。リダクションループは多くの場合効率的に並列計算できる。しかし、演算子の代数的性質を注意深く確認し計算の依存関係を寸断することが必要となるため、自動並列化は容易ではない。本発表では、リダクションループが半環上の線形式をなすかを、ブラックボックステストによって判定し、自動並列化を行う手法を提案する。半環上の線形式に基づく既存の自動並列化手法は記号実行などの解析に基づいており、プログラムの末節の変更の影響を受けやすく、大きなプログラムの扱いにも難があった。これに対し、提案手法では、リダクション変数の推測・ループ分解・線形式の係数の推測を全てブラックボックステストとして行う。これにより、静的解析はほぼ不要となり、また複雑なプログラム自動合成も避けることができる。提案手法のプロトタイプ実装を用いた実験により、既存のリダクションループ自動並列化の論文で扱われていた例のほとんどが提案手法で容易に並列化可能であることが確認できた。

Reduction loops, which summarize huge data into a few values, are widely used for counting, average calculation, maximum-value finding, etc. While their efficient parallel evaluation is commonly possible, their automatic parallelization tends to be difficult because of the necessity of using algebraic properties for breaking value dependencies. In this presentation, we propose an automatic reduction parallelization method based on black-box testing for recognizing the loop as a system of linear polynomials over semirings. There exist proposals for using linear polynomials for reduction parallelization. However, they rely on static analyses such as symbolic evaluations, and hence a small syntactic modification tends to affect their parallelizability. The current proposal instead uses black-box testing for every component, including reduction variable inference, loop decomposition, and coefficient inference. It is little susceptible to program detail; moreover, it avoids complex program synthesis. Our proof-of-concept implementation could immediately parallelize nearly all test programs, which was exhaustively collected from the literature on automatic reduction parallelization.

1　はじめに

リダクションループとは多数のデータを少数の結果へと集約する処理であり、データの個数のカウントや総和・平均値・最大値などの計算に広く用いられている。特に並列計算の文脈では、計算に逐次的な依存関係があるにもかかわらず効率的な並列計算が可能であることがよく知られており、「並列リダクション」と呼ばれ重要な構成要素となっている。しかし、リダクションループの自動並列化は一般には容易でない。並列化のためには計算の依存関係を寸断する必要があるが、これには演算子の代数的性質を注意深く確認しつつ計算の依存関係を認識することが求められる。そのため、既存のフレームワーク（例えば OpenMP、MPI、Inted TBB など）では、自明に自動並列化が可能なリダクションループ（例えば総和や最大値などの計算）のみを対象とするか、またはプログラマが明示的に並列評価に必要な情報を与えることを求めている。

Lightweight Reduction-loop Parallelization by Black-box Testing.

Akimasa Morihata, 東京大学大学院総合文化研究科, Graduate School of Arts and Sciences, the University of Tokyo.

Shigeyuki Sato, 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.

複雑なリダクションループや関連する構造（リダクションを行う再帰関数など）の自動並列化については多くの既存研究がある [1–5, 8, 9, 11–16]. これらはいずれも以下のような方針を採用している。

- 効率的な並列リダクションが可能なプログラムのパターンを与える。
- 並列化対象のリダクションループがそのパターンに変形できるかを、静的解析（特に記号実行）や自動プログラム合成を用いて確認する。

これら 2 点の要素はトレードオフのような関係にある。並列リダクションパターンの一般性の高いほど多くのリダクションループが並列化できる可能性があるが、そのパターンへの具体的な変換手順の構成が難しくなり、複雑な自動プログラム合成などが必要となりやすい。一方、並列リダクションパターンとして一般性の低いものを採用すると、表現力は劣るが、そのパターンへの変換に際しては特長を活かし効率的に行える可能性がある。

本研究では「半環上の線形式」を並列リダクションが可能なプログラムのパターンとして採用する（詳細は 2 節で解説する）。この方針は既存研究 [9,10,13,15] でも採用されているが、並列リダクションパターンとしての一般性が低いと思われること、また実際にそのパターンに変換可能なリダクションループであったとしても、それが明示的に半環の演算子を用いて記述されていないと並列化できないことが問題だとされてきた。本研究はこれらの問題が杞憂であることを示す。つまり、「半環上の線形式」は（少なくとも自動並列化が可能なリダクションループに対する抽象化としては）一般性が高く、また注意深く自動並列化アルゴリズムを構成すれば、プログラマが半環の演算子を使わなくても、またプログラムが複雑になろうとも、頑健に並列化が可能である、ということを示す。

本研究の主要な着想は、「半環上の線形式」を並列リダクションのパターンとして採用すると、ブラックボックステストによって簡単に並列化可能性を推測できる、という点である。標準的なアプローチでは、適切な解析によってプログラム並列化に有益な情報を集め、それを用いることで自動並列化を行う。しかし、一般に静的解析は構文上の差異（例えば関数呼び出しの有無）などの影響を受けやすい。これに対し本研究では、プログラムの詳細を全く無視し、プログラムの入出力関係だけから、適切な並列プログラムを構成する。このアプローチでは、静的解析ほど自由に情報を得ることはできないものの、入出力関係だけはどんなプログラムでも均一に得ることができる。よって、入出力関係が十分な情報を含んでいれば、これは頑健な並列化手法となる。

例として、配列 `array` の総和を求める簡単なループを考えよう [†1]。

```
s = 0
for x in array:
    s += x
```

このループの本体は `s += x` という文である。この文を「`s` と `x` を含む環境を更新する」プログラムだとみなし、ブラックボックステストを行う。テストする内容は「プログラム終了後の `s` と `x` は、プログラム開始前の `s` と `x` の線形式として表現できるか？」だ。テストによって、例えば以下のような結果が得られるだろう。左側が実行前の環境、右側が実行後の環境である。

$$\begin{aligned}
\{\mathtt{s}=0,\ \mathtt{x}=0\} &\quad\to\quad \{\mathtt{s}=0,\ \mathtt{x}=0\}\\
\{\mathtt{s}=3,\ \mathtt{x}=0\} &\quad\to\quad \{\mathtt{s}=3,\ \mathtt{x}=0\}\\
\{\mathtt{s}=0,\ \mathtt{x}=4\} &\quad\to\quad \{\mathtt{s}=4,\ \mathtt{x}=4\}\\
\{\mathtt{s}=2,\ \mathtt{x}=4\} &\quad\to\quad \{\mathtt{s}=6,\ \mathtt{x}=4\}
\end{aligned}$$

以上たった 4 回のテストの結果だけからであっても「プログラム終了後の `s` は、プログラム開始前の `s` と `x` の和、終了後の `x` は開始前の `x` と等しい」と推測するのは自然だろう。さらに多く、例えば 100 回ほどのテストを行えば、この推測がほぼ間違いないと確信することができるだろう（3 節では具体的なアルゴリズムを示す）。

当然のことながら、上記テストは完全ではなく、推測は間違っているかも知れない。そもそも、ブラックボックステストは、原理的に完全にはなり得ない。しかし、以下の 2 つの理由から、このことは自動並列化そのものの本質的な困難に比べればマイナーな問題であると言える。

†1 本研究を通して Python 風の擬似コードををプログラムの記述に用いる

まず、ほとんどの場合、ループ自動並列化は、プログラマの「このループを並列化せよ」という指示に基づいて行われるものだ。ブラックボックステストによるアプローチが失敗するのは、典型的には特定の値を閾値としてプログラムの挙動が本質的に変化するようなケースだが、並列化対象として指示されるループは通常そのような物ではないし、またプログラマ自身がそのようなループへの適用を避けることも難しくない。また、プログラマが閾値を自覚しているなら、その前後の値を両方システムにテストさせることも難しくない。

加えて、ブラックボックステスト完了後に 1 度だけ、特定の具体的な線形式とプログラムが等価かどうかを既存のプログラム検証手法で確認するのは、コストとして十分許容できると思われる。少なくとも、既存の少なくない自動並列化手法がプログラム検器（例えば SAT や SMT のソルバー）を何度も呼び出すのに比べれば、大きな改善であるとも言えよう。

以上の洞察に基づき、実際にプログラム自動並列化手法を構成した。これに際して、本研究では多数の技術的な貢献を行っている。

まず、半環上の線形式によって並列化を行うには、適切な半環とその線形式の係数を推測することが必要になる。本研究では、これをブラックボックステストにより実現する方法を提案する（3.3 節）。提案手法は数値の加算・乗算のなす半環だけでなく、真偽値に対する論理和・論理積のなす半環など、多くの半環に対して係数推測方法を与えた。

次に、リダクション並列化に際してはループ分解が効果的である一方、ループ分解は並列プログラムの効率を大きく低下させる要因ともなるため闇雲に行うのは現実的ではなく、適切な方針がよく分かっていなかった。これに対し、本研究ではブラックボックステストよってループ分解を行う手法（3.2 節）と推測された半環からループ再融合を行う手法を提案する（3.4 節）。提案手法は、並列化可能性の判定時には on-the-fly で可能な限りループ分解を行い、その後に並列可能性を失わない範囲で可能な限りループ再融合を行う。

半環上の線形式によってリダクションループを並列化する指針は既存研究で既に与えられていたが、多重ループの並列化についてはほとんど議論がなかった。これに対し、本研究では多重ループに対しても同様に容易な並列化が可能であることを論じ、並列化アルゴリズムを提案した（4 節）。

また、複雑なリダクションループでは配列に一時的な結果を格納することがあるが、この場合に配列の添字アクセスによって計算の依存関係が生じうるため、並列化が難しくなる。この問題に対し、ブラックボックステストに基づき、ループ内で読み出される・書き込まれる配列の添字を推測する手法を提案する（5 節）。この手法は線形式係数の推測に基づいており、提案手法の応用ともなっている。

さらに、提案手法に対しプロトタイプ実装を与え、既存のリダクションループ自動並列化の研究で扱われていた例に対し網羅的な実験評価を行った（6 節）。76 例中 74 例については本実装によってごく短時間で並列化が可能であった。並列化に失敗した 2 例は適切な半環を未実装だったために並列化に失敗したものであった。この結果は、「半環上の線形式」が非常に広い範囲の、特に既存手法で自動並列化が可能であると報告されているもののほとんど全てに近いリダクションループを捉えられることを示している。もちろん、「半環上の線形式」が全ての自動並列化可能なリダクションループを捉えられると主張するつもりはない。しかし、提案手法が軽量であることもあり、これはリダクション自動並列化の 1 つのベースラインとなりうるだろう。

2 半環とリダクションループ並列化

2.1 半環

本研究では半環に基づいてリダクションループの並列化を行う。半環は整数上の加算・乗算の関係を抽象化したものである。形式的には半環は $(S, \oplus, \otimes, \overline{0}, \overline{1})$ からなる 5 つ組である。S は計算対象となる値の集合である。$\otimes$ と $\oplus$ は S に対する 2 項演算であり、それぞれ「加算」「乗算」と呼ばれる。$\overline{0}$ と $\overline{1}$ は S の要素である。これらは以下の性質を満たすことが要求される。

$a \oplus \overline{0} = \overline{0} \oplus a = a$	$\overline{0}$ は $\oplus$ の単位元
$a \oplus (b \oplus c) = (a \oplus b) \oplus c$	$\oplus$ の結合性
$a \oplus b = b \oplus a$	$\oplus$ の可換性
$a \otimes \overline{1} = \overline{1} \otimes a = a$	$\overline{1}$ は $\otimes$ の単位元
$a \otimes (b \otimes c) = (a \otimes b) \otimes c$	$\otimes$ の結合性
$a \otimes (b \oplus c) = (a \otimes b) \oplus (a \otimes c)$	$\otimes$ の左分配性
$(b \oplus c) \otimes a = (b \otimes a) \oplus (c \otimes a)$	$\otimes$ の右分配性
$a \otimes \overline{0} = \overline{0} \otimes a = \overline{0}$	$\overline{0}$ は $\otimes$ の吸収元

実用上重要な半環の大部分は $\otimes$ が可換性をもつ。これを可換半環と呼ぶ。$\otimes$ が可換性をもつと議論が簡単になることもあり、本稿では「半環」で単に「可換半環」のことを指す。ただし、提案手法が本質的に $\otimes$ の可換性を要求するわけではない。

半環の例としては以下のものが挙げられる。整数の加算と乗算のなす半環 $(\mathbb{Z}, +, \times, 0, 1)$、真偽値の論理積と論理和のなす半環 $(\{0,1\}, \vee, \wedge, 0, 1)$、真偽値の論理和と論理積のなす半環 $(\{0,1\}, \wedge, \vee, 1, 0)$、整数の二項最大値演算と加算のなす半環 $(\mathbb{Z} \cup \{-\infty\}, \max, +, -\infty, 0)$、整数の二項最大値演算と二項最小値演算のなす半環 $(\mathbb{Z} \cup \{\infty, -\infty\}, \min, \max, \infty, -\infty)$、非負整数の二項最大値演算と乗算のなす半環 $(\mathbb{Z}_+, \max, \times, 0, 1)$、集合の和集合と積集合のなす半環 $(2^S, \cup, \cap, \emptyset, S)$。これらいずれの例についても、上記性質は簡単に確認できる。

変数集合 $\{x_1, x_2, \ldots, x_n\}$ が与えられたとき、半環 $(S, \oplus, \otimes, \overline{0}, \overline{1})$ 上の線形式とは、係数 $a_0, a_1, \ldots, a_n \in S$ で特定される以下の形式の式を指す。

$$a_0 \oplus (a_1 \otimes x_1) \oplus \cdots \oplus (a_n \otimes x_n)$$

以降、誤解のない範囲で半環 $(S, \oplus, \otimes, \overline{0}, \overline{1})$ を $(\oplus, \otimes)$ と略記する。

2.2 代数的単純化による分割統治計算

リダクションループが半環上の線形式で特定できる場合には並列化が可能であることが知られている [9,10,13,15]。例として以下のリダクションループを考える。なお、これは最大部分列和問題 [4,12] の計算に対応する。

```
gm = 0
lm = 0
for x in array:
    lm, gm = max(0, x + lm), max(gm, x + lm)
```

この並列化を論じるためにいくつかの用語を準備する。**リダクション変数**とは、ループを繰り越す依存関係のある値を保持している変数である。上記例では `lm` と `gm` がリダクション変数にあたる。一方、**非リダクション変数**とは、リダクション変数ではない変数である。リダクション変数はループの実行に従って計算を進めないと値が判明しないため、並列計算の妨げになる。一方、非リダクション変数は、ループの繰り返し毎にどの値を取るかがそれ以前の繰り返しを計算せずとも確定するため、並列計算の妨げにはならない。なお、ループカウンタの類（例えば上記ループの `x`）は、形式上はループを繰り越す依存関係を持つが、値がループを計算せずとも確定するため、非リダクション変数として扱う。

さて、上記ループの本体の計算は、リダクション変数に対応する変数集合 $\{lm, gm\}$ をとり、半環 $(\max, +)$ を考えることで以下の線形式系とみなすことができる。なお、ここでは $\uparrow$ を二項最大値演算子として使う。

$$\begin{aligned} lm &:= 0 \uparrow (\mathtt{x} + lm) \uparrow (-\infty + gm) \\ gm &:= -\infty \uparrow (\mathtt{x} + lm) \uparrow (0 + gm) \end{aligned}$$

なお、左辺の lm と gm はプログラム中では次のループでの lm と gm に対応する。

これをふまえこのプログラムを並列化する。並列プログラムでは、各プロセスはそれぞれループの連続した繰り返しの処理を担当する。例として、あるプロセスがループの s 回目から $t-1$ 回目、つまり `array[`s`]` から `array[`$t-1$`]` までの処理を担当するとする。このプロセスは、本来ループの $s-1$ 回目で計算されたリダクション変数の値なしに計算を進めることができない。そこで、並列プログラムは代わりに線形式（正確にはその係数）を計算する。以降の繰り返しについても、具体的な値ではなく線形式の形で計算を続ける。このとき、半環の代数的性質によって線形式を単純化できる点が重要である。そのため、繰り返しを経ても式が肥大化することなく、係数を求め続ける形で計算を続けることができる。

例として再び最大部分列和問題を考える。いま、`a[`s`]` の値が 3、`a[`$s+1$`]` の値が -4 だとすると、$s+1$ 番目のループまで処理した時の線形式は以下のよう

に求められる。

$$\begin{aligned} lm &:= 0\uparrow(-4+(0\uparrow(3+lm))) \\ &= 0\uparrow(-1+lm) \\ gm &:= (-4+(0\uparrow(3+lm)))\uparrow((3+lm)\uparrow gm) \\ &= -4\uparrow(3+lm)\uparrow gm \end{aligned}$$

上記の処理を繰り返すと、各プロセスが担当部分の計算を線形式の形まで集約できる。最後に、ループの計算の進行に従い初期値から順に、各プロセスが求めた線形式のリダクション変数へ値を代入し左辺値を求めることを繰り返すことで、最終的なリダクションの結果を求めることができる。

この並列計算の計算量は、プロセッサ数 p がループの回数 N に比べて非常に小さいとして、$O(N/p)$ であり、漸近的に理想的な並列速度向上を示す。

2.3 リダクション変数の依存関係によるループ分解

前節で述べたアプローチだけでは様々な演算子が関係したループを並列化できない。例として括弧対応 [1,8] を考える。このプログラムは、正しく括弧が対応しているかどうかを判定する。

```
count = 0
valid = true
for i in len(1, len(array)):
  if array[i] == '(':
    count += 1
  elif array[i] == ')':
    valid = valid and count > 0
    count -= 1
```

変数 count は現状の括弧のネストの深さを、変数 valid はそこまでの括弧対応が正しいかどうかを保持する。閉じ括弧があるときに count が 0 以下の場合、つまり対応する開き括弧がない場合に valid は偽となる。このプログラムでは、整数に対する加算と真偽値に対する論理積演算が使われており、1 つの半環での計算には明らかに対応しない。

実は、変数 valid の値は変数 count の値次第で変化する（この状況を「valid が count に依存する」と呼ぶことにする）ものの、変数 count は変数 valid には依存しない。そこで、count の値を事前に全て計算するループと、count の値をふまえ valid を求めるループの 2 つに分解する。

```
count = 0
counts = [None] * len(array)
for i in len(1, len(array)):
  counts[i] = count
  if array[i] == '(':
    count = count + 1
  elif array[i] ==')':
    count = count - 1
valid = true
for i in len(1, len(array)):
  if array[i] == ')':
    valid = valid and counts[i] > 0
```

前半のループでは、ループの各繰り返しごとの count の値を計算し、配列に記憶する。後半のループはその結果を用いて valid を求める。前半のループは整数の加算のみを、後半のループは真偽値の論理積のみを用いており、それぞれ半環上の線形式で捉えることができる。また、前半のループは単一の値ではなく配列を計算するが、この計算パターンは並列 scan（または並列 prefix-sum）と呼ばれ、並列リダクションと同様に並列計算できることが知られている。よって、このプログラム全体は効率的に並列計算できる。

以上の過程は以下のように一般化することができる。複数のリダクション変数がある場合には、変数の依存関係に基づきループ分解を行うことができる。その結果、相互に値が依存する変数に関する計算のみを（つまりそれ以外の変数の値は全て確定していると仮定した場合に）半環上の線形式によって捉えることができれば、プログラム全体が並列化できる。このことは先行研究、例えば [4,11–13] などで既に広く利用されている。

ループの分解は並列化を簡単にするが、得られる並列プログラムには無視できないオーバーヘッドを課す。並列 scan は、多数の値を計算・記憶する必要がありアルゴリズムも複雑になるため、並列リダクションに比べ数倍程度遅い。また、メモリの読み書きの回数が増えるため、メモリ帯域により並列速度向上が律速されやすくなる。そのため、並列化に必須でないな

らループ分解は可能な限り避けたい。例えば、前小節の最大部分列和問題でも、`lm` は `gm` に依存しないためループ分解は可能であるが、ループ分解をせずとも並列化できるため避けている。

3 ブラックボックステストによる並列化可能性判定

3.1 入力プログラム

提案手法の入力としては以下のようなループを想定する。

```
for a in array:
  stmt
```

つまり、ループの本体が *stmt* であるループである。*stmt* の実行は `array` の値を変更しないと仮定する。この条件は、3.2 節の解析時に同時に確認できる。これに加え、*stmt* が使用する変数の集合 X が既知であるものとする。変数集合を得るには簡単な静的解析を行っても良いが、プログラム全体をテスト実行し、このループ開始時・終了時の変数環境を確認しても良い。

各変数 $x_i \in X$ の値が v_i の状況で *stmt* を実行すると $x_j \in X$ の値が v'_j となる、という状況を Hoare 論理にならい

$$\{x_0 = v_0, x_1 = v_1, \ldots\}\ stmt\ \{x_j = v'_j\}$$

と記述することにする。また、$E = \{x_0 = v_0, x_1 = v_1, \ldots\}$ に対し、$E[x_{i_1} = w_1, \ldots, x_{i_m} = w_m]$ は E 中の $x_{i_1}, \ldots, x_{i_m}$ についての等式をそれぞれ $x_{i_1} = w_1, \ldots, x_{i_m} = w_m$ に置換したものを表すとする。

3.2 依存性解析とループ分解

提案手法では、2.3 節に従い、まずリダクション変数間の依存性解析を行い、ループの分解を行う。このために以下のブラックボックステストを行う。

手続き 3.1 (変数依存性の確認)**.**

1. 依存関係を空集合で初期化する。
2. 各変数 $x_i, x_j \in X$ に対し以下を行う。
   (a) 十分な回数以下を確認する。
      i. 各 $x_k \in X$ に対し値 v_k をランダムに生成し、$E = \{x_0 = v_0, x_1 = v_1, \ldots\}$ とする。
      ii. $E\ stmt\ \{x_j = v'_j\}$ となる v'_j を求める。
      iii. $v_i \neq w_i$ となる値 w_i をランダムに生成する。
      iv. $E[x_i = w_i]\ stmt\ \{x_j = v''_j\}$ となる v''_j を求める。
      v. $v'_j \neq v''_j$ ならば依存関係に $x_i \mapsto x_j$ を追加し、最内の繰り返しを終了する。
3. 依存関係の推移閉包をもとめ、最終的な依存関係とする。

ここでは、ランダムな値を何度も生成しながら、x_i の値のみを変更した場合に x_j の値が変化しうるかどうかを確認している。x_j の値が変化した場合、x_j は x_i の値に依存している。最終的な依存関係は、テストによって求まった推移閉包に対応する。これは以下のようなケースを適切に扱うためである。

```
for a in array:
  z = y + 1
  y = x + 1
```

このプログラムでは、上述のテストでは `x` を変更しても `z` の値は変化しない。これは、上述のテストではループの 1 回分の計算しか行わないためである。しかし、ループの複数回の計算を行えば、`z` の値は `x` の値の変化の影響を受ける。そのため、これは「依存関係がある」と判定されるのが望ましい。

依存関係の解析はリダクション変数・非リダクション変数の区別をせずに行っている。解析の結果として、自分自身に依存している変数がリダクション変数だと判明する。これ以外の変数は、本質的には非リダクション変数である。特に、どの変数にも依存していない変数は定数値である。なお、前述のとおり、変数 `array` は定数値でなければならない。他の変数に依存していても、自身に依存してない場合、ループをまたぐ依存関係はないため、自明に並列化できる。リダクション変数に依存していたとしても、ループ分解を行えばループをまたぐ依存関係は消失する。

3.3 半環・係数の推測と検証

3.3.1 半環・係数推測の概要

依存関係が解析できると、次は各リダクション変数 x_i について、その変数の計算がどの半環に対応するかを推測する。このためには以下のブラックボックステストを行う。以下は、変数 $x_i \in X$ の計算に対する半環 $\mathcal{R} = (S, \oplus, \otimes, \overline{0}, \overline{1})$ のテストとなっており、必要に応じて様々な変数・半環に対してテストを行う。また、係数の推測は半環毎に手法が異なるため、後で詳しく論じる。

手続き 3.2 (半環と係数の推測)**.**

1. $\{x_j \mid x_j \in X, x_i$ と x_j は互いに依存 $\} = X' = \{x_{i_1}, \ldots, x_{i_m}\}$ とする。
2. 十分な回数以下を確認する:
   (a) 各 $x_k \in X$ に対し値 v_k をランダムに生成する。
   (b) $\{x_0 = v_0, x_1 = v_1, \ldots\}$ *stmt* $\{x_i = v_i'\}$ となる v_i' を求める。
   (c) 半環 $\mathcal{R}$ について、$v_1, v_2, \ldots$ と v_i' をもとに X' を変数とする線形式の係数 $a_0, a_1, \ldots, a_m$ を推測する。
   (d) $a_0 \oplus (a_1 \otimes v_{i_1}) \oplus \cdots \oplus (a_m \otimes v_{i_m}) \neq v_i'$ ならば半環 $\mathcal{R}$ は不適切である。手続きを終了する。
3. x_i の半環の候補に $\mathcal{R}$ を加える。

ここでは、単純に *stmt* を実行した場合と、半環 $\mathcal{R}$ の線形式の係数を推測し、その線形式を計算した場合とで、結果が一致するかどうかを確認する。結果が一致しない場合、半環 $\mathcal{R}$ の線形式で計算を表すことはできない。なお、相互に依存する変数は同時に計算する必要があるため、これらをリダクション変数（つまり線形式の変数）として扱う。様々な変数・半環についてこのテストをを行い、相互に依存する変数間で共通に利用できる半環が見つかった場合、それが実際に並列化に使われることになる。

このテストでは、各半環でどのように線形式 $a_0 \oplus (a_1 \otimes x_{i_1}) \oplus \cdots \oplus (a_m \otimes x_{i_m})$ の係数を推測するかが重要となる。以下の説明では、手続き 3.2 の通り、$E = \{x_0 = v_0, x_1 = v_1, \ldots\}$ を前提として、$x_{i_1}, \ldots, x_{i_m}$ を変数とする線形式のために係数を $m+1$ 個推測することを考える。

3.3.2 定数項の推測

定数項 a_0 の推測方法は半環によらず同じである。

$$E[x_{i_1} = \overline{0}, \ldots, x_{i_m} = \overline{0}]\ stmt\ \{x_i = a_0\}$$

となる値 a_0 を求めればよい。これは、*stmt* の計算がもし線形式で表されるなら、各変数に $\overline{0}$ を代入すれば他の係数の影響が消え定数項のみが残るという洞察に基づいている。

定数項以外の係数の推測方法は半環の性質毎に異なる。

3.3.3 加算の逆元が存在する場合

値 a^{-1} が値 a の加算 $\oplus$ の逆元であるとは、$a^{-1} \oplus a = \overline{0}$ であることを意味する。$(+, \times)$ や行列の加算・乗算からなる半環では加算の逆元が存在する。この場合、係数を簡単に推測できる。係数 a_j を推測したい場合、$E[x_{i_1} = \overline{0}, \ldots, x_{i_j} = \overline{1}, \ldots, x_{i_m} = \overline{0}]$ *stmt* $\{x_i = v_i'\}$ となる v_i' を求め、$a_j = a_0^{-1} \oplus v_i'$ とすればよい。

これは以下の洞察に基づいている。線形式 $a_0 \oplus (a_1 \otimes x_{i_1}) \oplus \cdots \oplus (a_m \otimes x_{i_m})$ に対し、x_{i_j} のみに $\overline{1}$、他の変数に $\overline{0}$ を代入すると、$a_0 \oplus a_j$ となる。これが *stmt* の計算と等しいならば $v_i' = a_0 \oplus a_j$ であるから、a_j を求めるには v_i' から a_0 を引けばよい。

3.3.4 加算が乗算に分配する場合

しばしば（可換）半環の加算と乗算を交換しても半環をなす。具体的には $(\vee, \wedge)$ と $(\wedge, \vee)$、$(\min, \max)$ と $(\max, \min)$、$(\cup, \cap)$ と $(\cap, \cup)$ などである。この場合、加算の単位元が乗算の吸収元であることより、乗算の単位元 $\overline{1}$ が加算の吸収元となる。すなわち、$\overline{1} \oplus a = \overline{1}$ を満たす。この場合も簡単に係数を求めることができる。具体的には、係数 a_j を推測したい場合、

$E[x_{i_1} = \overline{0}, \ldots, x_{i_j} = \overline{1}, \ldots, x_{i_m} = \overline{0}]$ *stmt* $\{x_i = v_i'\}$

となる v_i' を求め、$a_j = v_i'$ とすればよい。

これは以下の洞察に基づく。前述の通り $v_i' = a_0 \oplus a_j$ である。しかし、以下に示すとおり、任意の b に対し、$a_0 \oplus (a_j \otimes b) = a_0 \oplus ((a_0 \oplus a_j) \otimes b)$ である。

$$
\begin{aligned}
& a_0 \oplus ((a_0 \oplus a_j) \otimes b) \\
= \;& (a_0 \otimes \overline{1}) \oplus (a_0 \otimes b) \oplus (a_j \otimes b) \\
= \;& (a_0 \otimes (\overline{1} \oplus b)) \oplus (a_j \otimes b) \\
= \;& (a_0 \otimes \overline{1}) \oplus (a_j \otimes b) \\
= \;& a_0 \oplus (a_j \otimes b)
\end{aligned}
$$

よって $a_j = v_i'$ として問題がない。

3.3.5 乗算の逆元が存在する場合

最後に加算ではなく乗算が逆元を持つ場合、すなわち $a \otimes a^{-1} = a^{-1} \otimes a = \overline{1}$ の場合を考える。ただし、$\overline{0} = \overline{1}$ でない限り、$\overline{0}$ には逆元が定義できない。なぜなら、$\overline{0}^{-1} \otimes \overline{0}$ が $\overline{0}$ なのか $\overline{1}$ なのか定まらないからである。

乗算に逆元が存在するだけでは係数を求めることは難しいが、ここでゼロ元 $\overline{0}$ に「十分近い」値 $r \neq \overline{0}$ の存在を仮定する。r に求めるのは、**ほとんどの** a および c について、$(a \otimes r) \oplus c = c$ となることである。この時、以下の手順で係数 a_j を推測できる。$E' = E[x_{i_1} = \overline{0}, \ldots, x_{i_j} = r^{-1}, \ldots, x_{i_m} = \overline{0}]$ とし、

$$E' \ stmt \ \{x_i = v_i'\}$$

となる v_i' を求め、$a_j = v_i' \otimes r$ とする。

これは以下の洞察に基づいている。線形式 $a_0 \oplus (a_1 \otimes x_{i_1}) \oplus \cdots \oplus (a_m \otimes x_{i_m})$ に対し、x_{i_j} のみに r^{-1}、他の変数に $\overline{0}$ を代入すると、$a_0 \oplus (a_j \otimes r^{-1})$ となる。つまり $v_i' = a_0 \oplus (a_j \otimes r^{-1})$ であるはずである。これを以下のように式変形する。

$$
\begin{aligned}
& a_0 \oplus (a_j \otimes r^{-1}) \\
= \;& (a_0 \otimes r \otimes r^{-1}) \oplus (a_j \otimes r^{-1}) \\
= \;& ((a_0 \otimes r) \oplus a_j) \otimes r^{-1}
\end{aligned}
$$

つまり、$((a_0 \otimes r) \oplus a_j) = a_j$ であれば、$v_i' \otimes r = a_j \otimes r^{-1} \otimes r = a_j$ である。よって、ほとんどの a_0, a_j に対して、$v_i' \otimes r = a_j$ である。

この手法による推測は不完全である。実際の計算で用いられる a_0, a_j では $((a_0 \otimes r) \oplus a_j) = a_j$ が満たされないかもしれない。つまり、テストをパスしたとしても、実際にはこのリダクションループはこの半環の線形式では表現できないかもしれない。しかし、この問題はこの係数推測手法にのみあるわけではない。提案手法がブラックボックステストに基づいているため、全ての係数推測は完全ではない。よって、軽量に高精度の推測を行えれば実用上十分である。

この手法は、$(\max, +)$、$(\min, +)$、$(\max, \times)$、$(\min, \times)$ などの半環に有用である。例えば $(\max, +)$ であれば、ゼロ元 $-\infty$ に十分近い値として、絶対値の非常に小さな数、例えば -2^{30} などを用いることで、高い精度で係数を推測することができる。

3.3.6 加算のみを用いる場合

依存関係によってループ分解をした後であれば、少なくないリダクションループでは 1 種類の可換な演算子のみが計算に使われる。さらに、各変数が高々1 回しか使われないのであれば、線形式は必ず $a_0 \oplus x_1 \oplus \cdots \oplus x_m$ の形式となる。つまり、係数は $\overline{0}$ か $\overline{1}$ のどちらかである。かつ、変数 x_j の値が最終結果に直接依存する（つまり、手続き 3.1 で推移閉包を取る前の依存関係に含まれるなら）ならば、係数は $\overline{0}$ ではありえず、逆に直接依存しないなら $\overline{0}$ である。よって、1 種類の演算子のみが使われ、変数が高々1 回しか使われない場合には常に係数を適切に推測できる。

3.3.7 その他の半環の場合

以上の手法で、実用上有用と思われるほとんどの半環を扱うことができる。しかし全てではない。例えば、文字列集合に対する（可換でない）半環 $(\cup, \cdot, \emptyset, \{\varepsilon\})$（$X \cdot Y = \{vw \mid v \in X, w \in Y\}$、$\varepsilon$ は空文字列）に対しては、上記いずれの条件も満たさないため、係数を推測することはできない。これは今後の課題である。

3.4 ループ再融合

ここまでは可能な限りループを分解した上で各変数がどの半環に対応するかを推測してきた。しかし、2.3 節でも議論したとおり、ループ分解はオーバーヘッドが大きいため最小限にしたい。そこで、推測された半環から、並列化可能性を失わない範囲でループをできる限り再融合する。

ループを融合する際には、変数の依存関係に沿ったものでなければならない。加えて、複数の異なる半環による線形式が依存関係のある変数の計算の表現に使われてはならない。異なる半環の線形式が混ざると係数を適切に求められなくなるためである。以上をまとめると、ループが融合できる・できないを判断する条件は以下の通りである。

変数 x_i と変数 x_j を（$x_i = x_j$ を含む）1 つのループで計算する場合、x_i が x_j に依存するならば、x_j に依存し x_i に依存しない全ての変数は x_i や x_j と同じループで計算される必要があり、かつそれらは全て同じ半環で計算が表現できなければならない。

ここから導かれるいくつかの有用な結論を挙げる。

- 依存関係がある変数同士を 1 つのループで計算するときは同じ半環でなければならない。
- 依存関係が全くない変数同士は半環が異なっても 1 つのループで計算できる。
- 相互に依存する変数同士は 1 つのループで計算されなければならない。

以上の状況をもとに、融合できる範囲でループを融合してゆくことになる。

発見的にループ融合を行うのではなく、最適なループ融合を目指したいと考えるかもしれない。しかし、最適なループ融合の方針を与えるのは簡単ではない。例えば、リダクション変数 x_1, x_2, x_3 について、x_1 と x_2、x_2 と x_3、の計算はそれぞれ 1 つの並列化可能なループにまとめることができるが、x_1 と x_3（ひいては x_1, x_2, x_3 の全て）を 1 つのループにまとめると半環で表現できなくなってしまうとする。この場合、x_2 を x_1 と x_3 のどちらと同じループで計算するのが、大域的にプログラムの効率を最善とするかの判断は自明ではない。詳しい議論は今後の課題である。

3.5 コード生成

各リダクション変数の計算について、対応する半環が発見できた場合、コード生成を行う。コード生成の方針は、基本的には既存の線形式に基づく並列リダクション・並列 scan の生成方法 [10,13,15] に従う。提案手法で特徴的となるのは線形式の係数を算出するコードの生成方法である。

線形式の各係数は、3.3 節の方針に従って生成する。3.3 節では *stmt* 文実行時の各変数の値が具体的に分かっている状況で係数の値を求めていたが、コード生成時には同じ方法で係数を求めるプログラムを出力することになる。例えば、x_i の計算の定数項の生成に対応するコードは以下となる。

$x_{i_1} = \overline{0}$
$\vdots$
$x_{i_m} = \overline{0}$
stmt
$a_0 = x_i$

また、加算の逆元が存在する場合に、x_i の計算での変数 x_{i_j} に対応する係数 a_j の生成に対応するコードは以下となる。

$x_{i_1} = \overline{0}$
$\vdots$
$x_{i_j} = \overline{1}$
$\vdots$
$x_{i_m} = \overline{0}$
stmt
$a_j = a_0^{-1} \oplus x_i$

以上の方針によって得られるコードには多数の *stmt* の計算が含まれることになり、非効率であると感じられるかもしれない。しかし、その大部分は不要ないし共通である。そのため、既存のコンパイラ最適化によって非効率性の大部分は取り除かれると期待できる。

3.6 計算量

提案手法に要するコストは、変数の数、試す半環の数、およびテストを行う回数に依存する。いま、変数の数を V、半環の数を R、テストの実行回数を n とする。また、テストを 1 回行うのに要するコストは、仮に変数の数 V に比例するものとする。

依存関係の解析に際しては、各変数の組に対してテストを行う。このコストは $\mathrm{O}(nV^3)$ である。推移閉包計算のコストはアルゴリズムによるが、単純なアルゴリズムを用いるとして $\mathrm{O}(V^3)$ である。

半環の線形式係数の推測については、各変数の係数推測のコストが $\mathrm{O}(V)$ であり、係数を $V(V+1)$ 回推測することでテストを行う。そのため、コストは $\mathrm{O}(nRV^3)$ である。以上をまとめると、並列化可能性の推測に要するコストは $\mathrm{O}(nRV^3)$ である。

なお、実際にテストを行う回数は n よりかなり少なくなることに注意してほしい。依存性解析であれば

依存性が発見された時点で、半環の推測であればその半環が不適切であると判明した時点で、テストは終了する。そのため、変数同士が複雑に関連し、複雑な計算を行っているようなプログラムの場合、テストを行う回数はむしろ少なくなる。逆に、変数は沢山あるが値を受け渡しているだけで、変数同士にはあまり関係がなく、実質的な計算も行っていない（そのためどの半環にも対応しうる）ようなプログラムでは、多数のテストを行う必要が生じ、並列化可能性の推測にむしろ時間がかかる。この点は提案手法の特徴である。

4 半環に基づく多重ループの並列化

前節まででは 1 重のリダクションループに対する半環と線形式に基づく並列化を考えた。本節では前節のアプローチが多重ループにも適用できることを示す。

本節では以下の形式の 2 重ループを考える。

```
for a1 in array1:
  stmt1
  for a2 in array2:
    stmt2
  stmt3
```

この形式の 2 重ループを考えるのは本質的な制限ではない。本節の手法は、複数の内部ループをもつループ構造や 3 重以上のループ構造に対して自明に拡張できる。

4.1 デザイン

多重ループに対しても、3 節の方法をそのまま使って並列化を行うことはできる。そもそも 3 節の方法はブラックボックステストに基づいているため、ループ内部のプログラムがどのような構造をしているかに依存しない。しかし、以下の理由から、このアプローチで常に十分であるとは言い難い。

まず、多くの場合、多重ループを伴うプログラムのテストには一重ループの場合に比べかなりの実行時間がかかる。そのため、ブラックボックステストを繰り返す手法は現実的ではないかもしれない。また、ループ内部のプログラムが複雑になると、それに伴いそこで処理される配列や添字等の間に一定の制約（不変条件）が課される可能性が高い。これは、入力環境のランダム生成を難しくする。さらに、多重ループの場合、最外ループを並列化するのが最善の方針であるとは限らない。詳細は 4.4 節で論じるが、内部ループを並列化するほうが良い効率となる場合もあるし、そもそも内部ループは並列化できるが最外ループは並列化できない場合もある。

以上の状況をふまえ、本節では、プログラムが多重ループ構造であることが分かっていることを前提として、以下の方針に従って特化した並列化手法を与える。

- ループ処理を伴うプログラムに対するブラックボックステストは避ける。上記プログラムであれば、$stmt_1$・$stmt_2$・$stmt_3$ に対してブラックボックステストを行い、$stmt_2$ を含むループに対するテストは避ける。
- 各ループに対してモジュラーな解析を行う。これによって、どのループを並列化するかの指針を与える。

4.2 内部ループの独立計算

並列化の基本的な方針は前節までの手法と同じである。つまり、各 i について、外側のループの i 回目の繰り返し処理が線形式の形で表現できればよい。このとき、「外側のループの i 回目の繰り返し処理」には内側のループが含まれることに注意が必要である。一般には、内側のループの計算は、それ以前の外側のループ（やそれに伴う内側のループ）の計算結果に依存しうる。しかし、一重のループの場合と同様の観察により、内側のループの計算、具体的には $stmt_2$ の計算が半環上の線形式によって記述できていれば、それ以前の処理とは独立に内側のループの「計算結果」を求めることができる。

さらに、外側のループの i 回目の繰り返し処理の結果と $i+1$ 回目の結果が適切に併合できるためには、内側のループの計算結果にあたる線形式が、$stmt_1$ および $stmt_3$ の計算と適切に併合できる必要がある。これは、$stmt_2$ と $stmt_1$・$stmt_3$ が共通のリダクション変数と半環をもつ線形式としてそれぞれ記述できていれば十分である。

以上の洞察に基づき、2 重ループの並列化の方針は

以下となる。

1. $stmt_2$ に対して 3 節の方法を適用し、半環とその係数を推測する。
2. $stmt_1$・$stmt_3$ に対して 3 節の方法を適用し、半環とその係数を推測する。
3. 推測された半環・リダクション変数が同一であれば並列化は成功であり、各ループについて、3 節の方針に従いコード生成を行う。

これに際してはいくつかの注意点がある。以下の小節ではこれらについて述べる。

4.3 変数依存性とリダクション変数

まず、リダクション変数は、当該ループのみから判断されるリダクション変数だけでなく、他のループのリダクション変数も考慮する必要がある。例として以下のループを考える。

```
s = 0
for i in range(0, n):
  for j in range(0, m):
    s += a[i][j]
```

このループでは、外側のループだけを見るとリダクション変数は存在しない。しかし、内側のループでは s がリダクション変数であり、そのため、内側のループの「計算結果」が s に対するループを繰り越す依存関係を導入する。そのため、s は外側のループでもリダクション変数とみなされなければならない。

上記例からも分かるとおり、原則として、いずれかのループでリダクション変数である変数は、どのループでもリダクション変数であると判断せざるをえない。もちろん、これはやや保守的な判断ではある。例として以下のループを考える。

```
k = 0
for i in range(0, n):
  s = 0
  for j in range(0, m):
    s += a[i][j]
  k = max(k, s)
```

このプログラムでは、内側のループでは s がリダクション変数となっているが、外側のループから見た場合 s はループを繰り越す依存関係をもたない。このことをブラックボックステストで検証することは不可能ではない。s を変化させながらプログラムを実行し、結果が s に依存するかどうかを確認すれば良い。しかし、前提としてこのような解析は避ける方針である。また、リダクション変数を多めに見積もることにはデメリットが少ない。後の係数の推測等で多少コストはかかるが、最終的に係数が $\overline{0}$ になるだけである。よって、各ループについてリダクション変数を推測し、その総和をとるアプローチの方が現実的である。

変数についての依存関係も同様である。いずれかのループで依存関係がある変数同士は、全てのループで依存関係があると判断すべきである。やはり、これも保守的な判断である。そして、依存関係を保守的に見積もることによって、本来可能であったループ分解が不可能になり、ひいては並列化に失敗する可能性もある。現時点でこの問題が深刻となるような現実的な例は見つかっていないが、場合によってはより詳細な解析を行うことも有意義かも知れない。

4.4 コード生成

コード生成の方針は基本的には 3.5 節のものに従う。しかし、多重ループの場合、どの深さのループで分割統治計算を行うかという選択肢がある。以下、プロセッサ数を p、外側のループの繰返し回数を n、内側のループの繰返し回数を m とする。

まず、ループ全体について並列化が可能だと判断されたと仮定しよう。この場合、以下の 3 通りの並列化の方針がある。

1. 外側のループを（例えば n/p 回の繰り返し毎に分割し）並列に処理する。
2. 外側のループの各繰り返しに伴う n 個の内側のループを各プロセッサに割当てて並列に計算し、その結果を外側のループで逐次的に集約する。
3. 外側のループの各繰り返しごとに、内側のループをそれぞれ（例えば m/p 回の繰り返し毎に分割し）並列計算しつつ、全体としては逐次的に計算を行う。

アルゴリズムは異なるが、n や m が p に比べて十分大きければ、どの方法でも理論上の漸近計算量は $\mathrm{O}(nm/p)$ となり理想的な並列速度向上を達成する。

一方で、実際上の性能は多少異なる。一般に、(1) の方が並列計算タスクの生成回数が少なく（粒度が荒い）、そのため並列計算に伴う仕事量のオーバーヘッドが小さくなる可能性がある。一方、(3) の方が並列計算タスクを多く生成し（粒度が細かい）、そのため仕事量は多くなるものの負荷分散に成功しやすい。(2) は両者の中間的な性質となる。どのコード生成方針が適切かは具体的な処理内容やデータの大きさ・並列計算機環境等に依存するため、一概にはいえない。

また、 $stmt_2$ に対しては適切な半環と線形式が推測できたが、$stmt_1$・ $stmt_3$ については上手くいかなかった、というような場合もありうる。この場合、(1) の方針による並列化は不可能だが、 (2)・(3) の方針による並列化は可能である。

提案手法は、各ループについてリダクション変数や半環を推測してゆき、それを最終的に集約して並列化方針を判断するという方針を採用している。これによって、どのループまでであれば並列化が可能なのかを解析結果から判断できる。さらに、半環上の線形式を用いたリダクション並列化ではリダクション変数の数（つまり係数の数）に比例したオーバーヘッドを要するため、並列化に伴うオーバーヘッドがどの程度であるのかも見積もることができる。そのため、提案手法では、これら見積もりをふまえフレキシブルにコード生成を行ったり、複数の方針でコードを生成し実験的に性能を比較したりといったことが可能である。

5 中間データ構造として配列を用いる場合

特に多重ループからなる並列リダクションでは、中間的な値を配列に格納する場合が少なくない。例として以下のプログラムを考える。これは最長共通部分列 [2, 13] の計算に対応する。

```
for i in range(0, n):
  p = 0
  pp = 0
  for j in range(0, m):
    tmp = pp + 1 if a[i] == b[j] else 0
    m = max(tmp, r[j], p)
    pp = r[j]
    r[j] = m
    p = m
```

このプログラムは整数の配列 r を用いている。この配列は実質的には多数の変数の集合として機能している。

提案手法は半環上の線形式に基づいている。この時、線形式の各変数に対応するのは配列ではなく値（この場合は数値）であるし、また線形式の計算結果も配列ではなく値である。そのため、配列に対しては、配列そのものではなく配列の各インデックスをリダクション変数とみなし並列化を行う必要がある。しかし一方で、本研究ではブラックボックステストに基づいているため、具体的に配列のどのインデックスに注目すれば良いかを標準的な静的解析によって求めることはできない。

以下では、この問題を解決する。すなわち、配列を含むプログラムに対して、ブラックボックステストに基づき、配列を用いたプログラムの並列化を本研究で扱えるものに帰着する方法を示す。

5.1 配列変数に対するアクセスの推測

基本的な着想は、具体的に配列のどの変数に対して読み書きが行われたかを記録することである。3 節と同様の形式のループの並列化を考え、このループが配列 x を扱っていると仮定しよう。この時、以下のような手法によって、配列に対する読み出し・書き込みを判定できる †2。

- *stmt* 文実行前の x[i] のみを変化させると *stmt* 文実行後のいずれかの変数値が変化する場合、x[i] に対する読み出しがあったと判断できる。
- *stmt* 文実行前の x[i] と実行後の x[i] の値が異なっている場合、x[i] に対する書き込みがあったと判断できる。

これらにより、特定の実行における読み出し元・書き込み先を推測することができる。

しかし、特定の実行におけるアクセスを確認するだけでは不十分である。並列化を行うためには、ルー

†2 並列化対象の言語によっては、多少のメタプログラミングの機構（例えば配列の添字アクセスや代入に対するオーバーロード）を使い、配列に対するアクセス・書き込みを直接監視する方法もありうる。

プの何回目の繰り返し時には配列のどのインデックスにアクセスするかを知る必要がある。一般に、ほとんどの配列の添字アクセスは、他の変数値の（$(+, \times)$ 半環上の）線形式をなす。本研究でもこれを仮定し、3.3 節の手法によってこの線形式を推測する。具体的には以下の手順による。

手続き 5.1 (配列インデックスアクセスの推測)**.**

1. *3.2* 節の方法によって、配列の添字アクセスに影響する変数集合を得る。
2. 添字アクセス先が $(+, \times)$ 上の線形式であると仮定し、係数を推測する。具体的な推測方法は *3.3* 節の通りである。
3. 十分な回数テストを行い、推測が正しいことを確認する。推測が正しくなかった場合、並列化は失敗する。

前述の最長共通部分列のプログラムを例に説明を行う。まず、各変数値を変えつつ配列 `r` の読み出し元・書き込み先を確認することで、変数 `j` の値が読み出し元・書き込み先に影響していることが分かる。次に、$\mathtt{j} = 0$ および $\mathtt{j} = 1$ の場合の読み出し元・書き込み先から、線形式はどちらも $0 + 1 \times \mathtt{j}$ と推測できる。最後に、十分な回数を行いテストを行い、推測が正しいことを確認する。

以上によって、読み出し元・書き込み先が推測できたら、これらを配列の代わりに変数集合に加え、3 節の方法に基づき並列化を行う。例えば最長共通部分列の例であれば、配列 `r` ではなく `r[j]` を整数値変数とみなし、並列化を行う。

5.2 配列に対する依存性解析

配列を中間的に用いている場合、変数を用いる場合とは異なる形で「変数」間の依存関係が生じる場合がある。例として以下のプログラムを考える。

```
for i in range(0,n):
  x[i] = x[i+1]
```

このループに対し前小節の手法に基づき配列のアクセスを推測すると、`x[i]` および `x[i+1]` という「変数」を考える必要があることが分かる。これに続いて変数依存関係の解析を行うと、`x[i]` は `x[i+1]` に依存するが `x[i+1]` は `x[i]` には依存しない、という結果が得られる。しかしこれは正しくない。`x[i]` の値は次のループでの `x[i+1]` の値になる。このように、配列の場合、添字アクセスの変化による依存関係が発生しうる。これを無視することは、正しくない並列化を行うことに繋がりかねない

上記問題を解消するには、配列の各添字に対する依存関係は同一視すればよい。つまり、いずれかの添字に対して依存関係があれば、どの添字に対しても依存関係があると判断するのである。これは保守的であり、実際には依存関係がない場合もあり得る。配列の添字アクセスの形式とループカウンタの変化を注意深く確認すればより精密な解析も可能であろうが、ブラックボックステストに基づく本手法では、精密な解析を行う方法は少なくとも非自明であり、またそれによってどの程度の利得があるかも不明瞭である。

6 評価実験

本節では提案手法の有効性を調べるために行った評価実験について報告する。

6.1 実装

本稿で提案した並列化可能性判定手法を Python 言語で実装した。なお、実装は 350 行ほどである。

半環候補としては $(+, \times)$、$(\max, +)$、$(\max, \min)$、$(\min, \max)$、$(\wedge, \vee)$、$(\vee, \wedge)$、$(\max, \times)$ (非負値のみを扱う) を用意した。

入力プログラムはループ内の計算を表す Python の関数とし、これを `exec` 関数を用いて実行する。さらに、各引数・返値の型を追加の情報として与えている。型としては整数・真偽値・整数値配列を許している。浮動小数点数を入力とするプログラムについては、入力のランダム生成と結果の同値判定が複雑となるため、原則として整数を扱うとして処理している。

実装はおおむね本稿の内容に従っているが、コード生成（3.5 節）・ループ再融合（3.4 節）・配列の添字の推測（5.1 節）の機能は実装していない。

提案手法では入力プログラムの引数としてランダムな値を与えるが、プログラムによっては引数値に一定の制約がある場合がある。今回の実装では、制約は `assert` 文としてプログラム中に記述することにし

表 1　実験結果: 数値の計算に関するリダクション

プログラム	ループ分割	演算子	並列化可能	実行時間（秒）
総和 [1,3,4,12,14]		+	○	0.05
偶数和 [3]		+	○	0.04
正要素和 [3]		+	○	0.05
平均 [1,3]	○	+, +	○	0.12
正要素数 [3]		+	○	0.04
特定要素数 [3]		+	○	0.07
不連続部分数 [8]	○	+	○	0.09
最大値 [1,3,4,14]		max	○	0.06
2 番目最大値 [3]	○	max, max	○	0.17
最大絶対値 [3]		max	○	0.08
最小値 [1,3]		min, min	○	0.04
2 番目最小値 [1,8]	○	min	○	0.12
最大最小差 [3]	○	max, min	○	0.06
最大要素数 [3]	○	max, +	○	0.16
最小要素数 [3]	○	min, +	○	0.11
内積 [14]		+	○	0.05
ハミング距離 [1]		+	○	0.05
多項式 [1,8,13]	○	(+, ×), ×	△	0.09
複素数積 [14]		(+, ×)	○	0.08
二重指数平滑化 [8]		(+, ×)	△	0.09
三重対角行列 LU 分解 [13]	○	(+, ×)	△	0.08
差分法 [13]		(+, ×)	△	0.13

た。システムは、`AssertionError` を受け取った場合には入力を生成し直す †3。

並列化可能性のテストに要する時間は主にいくつの入力を試すかに依存する。今回の実験では各ステップで 1000 通りの入力を試すようにした。

6.2　実装上の効率化

本稿で述べた内容に加え、2 点実装上の効率化を行った。

1 つは、実質的な計算を行わずただ値を受け渡すだけのリダクション変数の検出である。このような計算は任意の半環で表現できる。提案手法では不適切な半環はテストですぐに発見される一方、適切な半環は規定回テストする必要があり、よりコストがかかる。値を受け渡すだけの変数は、適当な半環（実装上は $(+,\times)$）の係数を推測する際に、追加で「$\overline{0}$ でない係数がたかだか 1 つであり、それが $\overline{1}$ であること」をテストすることで確認できる。確認に成功した場合、以降はテストを行わず「全ての半環で表現できる」として処理する。

もう 1 つは、多重ループ時のテストの順番である。4 節では、各ループに対して独立に依存性解析と適切な半環の推測を行う手法を提案した。しかし、多重ループ全体を並列化する前提であれば、それぞれの変数依存性や半環の推測において、各ループを巡回的にテストする方が効率が良い。例えば、2 重ループにお

†3　ただし、係数推測のために単位元等を入力した際に、それが制約に反してしまい `AssertionError` となってしまう場合もある。この場合にはシステムは入力を再生成せず、その半環での並列化が失敗したものとして扱う。

表 2　実験結果: 最適化問題・判定問題・クエリに関するリダクション

プログラム	ループ分割	演算子	並列化可能	実行時間（秒）
最大連続 1 [1]	○	+, max	○	0.17
最大 1 間距離 [3, 8]		+, max	○	0.13
最大 0 間要素和 [3]		+, max	○	0.15
最大接頭辞和 [1, 8, 12]	○	+, max	○	0.23
最大接尾辞和 [8, 13]	○	(max, +)	○	0.07
最大部分列和 [1, 3, 4, 8, 12, 13]	○	(max, +), max	○	0.52
最大部分列積 [8]	○	(max, ×), max	○	0.26
全て同じ [3]	○	∧	○	0.06
01 数一致 [3]		+	○	0.05
括弧対応 [1, 8]	○	+, ∧	○	0.09
可視地点判定 [12]	○	max	○	0.11
dropwhile [1]	○	∨, max	○	0.07
要素検索 [3]	○	∨, max	○	0.06
整列済 [1, 3]	○	∧	○	0.06
0*1* [1]	○	∧	○	0.06
(01)* [3]	○	∧	○	0.06
0 が先頭以外になし [3]		∧	○	0.02
0 が 1 の後にある [1]	○	∧,∧	○	0.06
1*のマッチ数 [3]	○	+, +	○	0.12
1*2 のマッチ数 [3]	○	+, +	○	0.14
10*2 のマッチ数 [3]	○	+, +, +	○	0.28
1*2*3 のマッチ数 [3]	○	+, +, +	○	0.31
10*20*3 のマッチ数 [3]	○	+, +, +, +, +	○	0.66

いて、外側のループでは変数に依存性がないが内側のループでは依存性がある場合、外側のループを何度テストしても依存性は見つからないが、外側のループと内側のループを交互にテストすればすぐに依存性を発見できる。

以上 2 つの効率化は、いずれも並列化に比較的時間のかかるプログラムの効率を改善する。ただし、改善はせいぜい数倍程度であり、効率化なしでは並列化が事実上不可能なプログラムの並列化が可能になるわけではない。

6.3　実験環境と並列化対象プログラム

実験環境の CPU は Intel Core i7-7600U CPU (2.80 GHz)、メモリ 16GB、OS は Windows Subsystem Linux 上の Ubuntu 18.04、処理系は CPython 3.6.5（最適化オプション `-OO` 使用）である。

並列化対象プログラムとしては、リダクションループの自動並列化を行った先行研究 [1–4, 8, 12–14] で評価実験に用いられていたプログラムを全て用意した †4。ただし、明らかに重複のある例については省いている。例えば、「最大部分列和」については、最大和だけでなくその部分列の開始インデックス・終了インデックスも求めるプログラムのみを対象とし、最大和のみを求めるプログラムは省いている。

†4　「行最小値の最大値」については、[2] のプログラムが明らかに正しくなかったため、少なからずプログラムを修正している。なおこれにより条件分岐が増え、並列化はより難しくなっている。

表 3　実験結果: 多重ループのリダクション

プログラム	ループ分割	演算子	並列化可能	実行時間（秒）
2D 総和 [2]		+	○	0.05
2D 整列済 [2]	○	∧	○	0.05
4D 最大値座標 [14]	○	min	○	0.63
上下整列済 [2]		∧	○	0.02
対角整列済 [2]		∧	○	0.03
行値範囲単調増加 [2]	○	max, ∧	○	0.19
行値範囲 ovelap [2]	○	max, min, ∧	○	0.55
行値範囲単調縮小 [2]	○	max, min, ∧	○	0.55
全行値範囲 overlap [2]		∧	○	0.02
行最小値の最大値 [2]	○	min, max	○	0.16
列最小値の最大値 [2]	○	min, max	○	0.23
鞍点 [2]	○	min, max, min, max	○	0.51
2D 最大接頭辞和 [2]	○	+, max	○	0.14
2D 最大接尾辞和 [2]		(max, +)	○	0.03
2D 最大部分列和 [2]	○	(max, +), max	○	0.11
最大左上部分配列和 [2]	○	+, +, max	○	0.27
最大左下部分配列和 [2]	○	+, (max, +), max	○	0.24
最大右上部分配列和 [2]	○	+, (max, +), max	○	0.27
3D 最大接頭辞和 [2]	○	+, max	○	0.14
3D 最大接尾辞和 [2]		(max, +)	○	0.03
3D 最大部分列和 [2]	○	(max, +), max	○	0.11
3D 最大左接頭辞和 [2]	○	+, +, +, max	○	0.43
括弧対応列数 [2]	○	+, ∧, +	○	0.21
最頻出値 [2]	○	+, max	○	0.41
2 配列最大差 [2]	○	max	○	0.10
最長括弧対応 [2]	○	+, ∧, max	○	0.33
最長共通部分列 [2, 13]		(max, +)	○	0.19
独立要素数 [3]			−	
2D ヒストグラム [14]			−	

プログラムはできる限り自然な記述となるようつとめた。特に、最大値・最小値に関するプログラムでは、必要に応じて条件分岐も用いるなど、必ずしも半環に対応させることを意図しないプログラムとしている。

6.4　実験結果

実験の結果を表 1・表 2・表 3 に示す。「ループ分割」の列はループ分割が可能であると判定されたかどうかを表す。「演算子」の列は、どの演算子が使われていると判断されたかを表す。ループ分割が行われた場合、分割後の各ループについて使われた演算子がそれぞれ示されている。例えば「平均」の例では、2 つのループのそれぞれで + が使われていることを表す。

なお、提案手法は使われている半環を推測するが、演算子が 1 種類しか使われいない場合、その演算子を含む半環を列挙することになる。そのような場合には半環ではなく演算子のみを挙げている。「並列化可能」の列は、最終的に並列化が可能であると判定された場合に「○」、並列化可能と判定されるためにプログラムの書き方等に注意が必要だった場合に「△」、並列化可能性の判定に失敗した場合に「–」をそれぞれ記載している。「実行時間」の列には、並列化可能性の判定に要した時間を記載している。システムが行った判定の正しさについては著者が目視で確認したが、誤りはなかった。

実行時間は全ての例について 1 秒を大きく下回った。Python インタプリタでの実行であること、効率化等を全く考慮していないことを考えると、十分な性能であると判断できる。

並列化に失敗したのは「独立要素数」「2D ヒストグラム」の 2 例であった。理由は、「独立要素数」は集合、「2D ヒストグラム」はベクトルを扱っているためであった。これは今回の実装上の問題であり、適切な型と半環を加えれば並列化が可能であると推測している。

並列化が最も困難だったのは「三重対角行列 LU 分解」であった。この計算では数値の加算・乗算に加えて除算を用いているため、半環で捉えることができない。Sato ら [13] はこのような問題もうまく式変形することで半環上の線形式として定式化できることを論じているが、これはリダクション変数集合を変化させるものであり、提案手法の範囲で行うことはできない。今回は手作業でこの変形を行った。さらに、これを本実装で扱うには手作業でループ分解を行う必要があった。並列化には $(+, \times)$ 半環の係数を推測する必要があり、そのためにはリダクション変数に 0 を代入することになるのだが、これが除算の分母に使われてエラーとなってしまう。この除算は相互依存関係のない変数の値を求める計算であり、係数推測には関係がないのだが、ブラックボックステストである以上、プログラムのどの部分でゼロ除算がなされているかわからず、しかもその部分の処理を取り除くこともできない。そのため、手作業でループ分解を行うことにより、このゼロ除算を排除した。

その他の例についても、提案手法での並列化を成功させる上で配慮が必要であったものがいくつかあった。

- 「多項式（係数等に実数が現れる場合）」・「二重指数平滑化」・「差分法」では、本質的に実数を扱う。しかし、現状の実装では数値誤差を考慮していないため、推測した線形式と実際の計算の値が微妙に異なるため並列化に失敗する可能性がある。今回は数値誤差が生じない値（例えば 1/4）を用いて実験した。実用上は、並列化が数値誤差にどう影響するかも含め、より踏み込んだ議論が必要かもしれない。
- 「最大部分列積」の計算では、最大値となる部分列だけではなく最小値となる部分列も覚える必要がある。これは、負数が現れたときに大小が反転するためである。これをプログラムとして記述する際、負値を用いると $(\max, \times)$ の半環で扱えなくなるため、代わりにその絶対値を記憶する必要がある。

しかし、配慮が必要だった例は多くなかった。プログラム解析・変換ではプログラムの書かれ方によって利用可能性が変化するのは一般的である。少なくとも、他のリダクション自動並列化手法で必要な配慮と比べ、より注意深い配慮が必要であったとは考えていない。

7 関連研究

リダクションループおよび関連する構造の自動並列化については少なくない研究がある。

Fisher と Ghuloum [4] は複雑なリダクションループの自動並列化について先駆的な研究を行っている。彼らの手法は、ループ本体のプログラムに対し、リダクション変数を「穴」と見なしたテンプレートを考え、それらの複数回の合成が同じテンプレートに単純化できるかをプログラムの等価変形を繰り返して確認するものである。後に Morihata と Matsuzaki [11] は同様の手続きを限定記号除去を用いて定式化し直している。また、Farzan と Nicolet [1] は構文主導合成（Syntax Guided Synthesis）の枠組みに基づき、同様の自動並列化手法を提案している。これら手法は

一般性は高いものの、適切なテンプレートをどう得るかや合成時の等価変形をどう行うかの指針がないために、並列化手続きで多くの可能性を発見的に探索する必要が生じやすく、複雑なプログラムの並列化は現実的ではなかった。そのため、多くの研究では、並列化の際に考えるプログラムのパターンをある程度限定することで、現実的な自動並列化を目指している。

Xu ら [15] は計算が半環（およびそれの拡張にあたる代数構造）上の線形式であることを確認する静的解析を提案し、これによってリストを集約する再帰関数が並列化できることを示した。Mastuzaki ら [9] は木構造を集約する再帰関数について、Sato と Iwasaki [13] はリダクションループについて、よく似た手法を提案している。1 節でも述べたとおり、これらの研究は静的解析に依存しており、明示的に半環の演算子を用いてプログラムが記述されていなければ適用が難しかった。この問題を軽減するため、Sato と Iwasaki [13] は条件分岐で記述されたプログラムから SMT ソルバーを使って max 演算子を抽出する手法も与えている。本研究は、これらの研究の欠点をブラックボックステストを用いることで改善している。これにより、SMT ソルバー等に依存することなく、明示的に半環の演算子を使っていないプログラムに対しても並列化を行うことができる。

Suriana ら [1] は Halide という高性能配列計算用領域限定言語のためのリダクション並列化手法について論じている。彼らの手法はリダクション並列化が可能な演算子群を事前に探索的に生成しておき、プログラムがその演算子群によるリダクションに対応するかどうかを記号的に確認するものである。これは、半環の候補を事前に列挙しておき、その演算子でプログラムが記述されているかどうか確認する提案手法と類似している。しかし、提案手法では探索的な生成を避け、代わりに半環に注目することでより系統的に演算子群の特定を行っている。また提案手法では、明示的にその演算子でプログラムが記述されていなくても、意味上等価なプログラムは同様に並列化できる。

Jiang ら [8] のリダクション自動並列化手法では、実行時に複数の初期値からループの処理を行い、その結果を用いてループ処理の要約結果（例えば線形式）を再構成する。このアプローチは、複数回のテストケース実行の結果から係数を推測する本研究のアプローチによく似ている。しかし、両者には 2 点の重要な違いがある。まず、彼らのアプローチでは、ループの処理結果から要約結果を得る方針を静的解析によって決定している。これに対し本研究では、どの半環が適切かのブラックボックステストによって、要約結果を得る方針を決定している。また、彼らの手法は数値の加算・乗算・最大値計算に特化した手法と、真偽値などの有限値域の値に特化した手法から構成されており、例えば和集合・積集合からなる半環やベクトル計算などには適用できない。これに対し、本研究は半環に注目することで、より広い範囲のプログラムを系統的に並列化することができる。

Farzan と Nicolet [2] は、多重ループに対する自動並列化手法を提案している。彼らの手法は、外側のループの各繰り返しに対する内部ループの計算の集約結果を独立に求めることで、それら集約結果を集約するネストのないリダクションループの並列化に帰着する、というアプローチである。特に、内部ループの集約結果に注目し、それと外側のループとの関係を確認する点で、我々の並列化手法との類似性がある。しかし、彼らの手法が「外側のループを並列化するために内部ループの構造を調べる」という、いわばトップダウンのものであるのに対し、我々の手法は「**全て**のループの構造を独立に解析し、それらをもとに各ループの並列化可能性を調べる」という、いわばボトムアップの構造をもつ点に違いがある。ボトムアップにしたことにより、各ループに対する解析はよりモジュラーになり、またフレキシブルな並列化・コード生成が可能になっている。このボトムアップな構造は、半環という代数構造に着目したことの大きな利点である。

Fedyukovich ら [3] は反例主導帰納的合成（Counter Example Guided Inductive Synthesis）の枠組みに基づき、SMT ソルバーを用いた自動並列化手法を提案している。より具体的には、簡単な並列計算パターンから始め、並列化対象プログラムがそのパターンで記述できるかを SMT ソルバーに問合わせ、不可能なら徐々に複雑で表現力の高い並列計算パターンを

試す、という方針で並列プログラムを合成する。我々のアプローチで得られる並列プログラムは彼らの考えた並列計算パターンには直接は合致しないため、両者の能力の比較は難しい。ただし、SMT ソルバーを何度も呼び出すという戦略上、彼らの手法が複雑で大きなプログラムにもスケールするという見通しは乏しい。なお、単純な並列計算パターンから始め徐々に複雑なパターンを複雑化するアプローチは、本研究でもより複雑なプログラムを並列化する際には有用かもしれない。

リダクションループの自動並列化については、これらの他にも第三リスト準同形定理 [6] に基づく手法がある。Geser と Gorlatch [5] は第三リスト準同形定理と反単一化を組み合わせることで、Morita ら [12] は第三リスト準同形定理とプログラム逆化を組み合わせることで、森畑ら [16] は第三リスト準同形定理を並列プログラム候補の生成検査と組み合わせることで、それぞれ自動並列化を行う手法を提案している。第三リスト準同形定理に基づく場合、プログラマは通常のループだけではなく、通常とは逆順に処理するループのプログラムも同時に与える必要がある。そのためプログラマへの負担が重い。一方で、少なくとも今回の実験で確認できた範囲では、リダクションループの自動並列化に際して 2 種類のループを与えることの恩恵はそれほど大きくないように見える。なお、森畑ら [16] の手法で用いている候補生成手法は、並列化対象プログラムの構文的な書き換えに基づくものであり、半環に基づく本研究の手法よりはむしろ Farzan と Nicolet [1] の手法に近いものとなっている。

リダクションループの自動並列化は自動プログラム合成 [7] の一種とみなすことができる。自動プログラム合成は、特定の仕様（例えば与えられたプログラムと等価で、並列化可能なパターンに合致する）を満たすプログラムを自動的に構成する技法である。実際、少なくない自動並列化手法 [1–3, 11, 16] は明示的に自動プログラム合成に由来する手法を用いている。

自動プログラム合成の観点からみた場合、我々の手法は入出力例によるプログラミング（Programming by Example）の一種だとみなすことができる。ただし、入出力例はユーザが与えるのではなく、並列化対象のプログラムを実行することで得る。特に、半環の係数推測時には、半環の性質に基づき注意深く入力を与えている。これにより、プログラム合成を軽量かつ効果的に実現できている。

8 まとめと今後の課題

本稿ではブラックボックステストによってリダクションループの並列化を行う手法を提案した。この手法は、ループでの計算が半環上の線形式に対応するかどうかを推測することを要旨としている。また、ループの分解や配列へのアクセスの推測など、関連する問題についても同様にブラックボックステストによって行う手法を提案している。提案手法は多重ループに対しても適用可能である。

コンセプト確認のための実装を用いた実験では、既存のリダクションループの並列化で用いられたほとんど全ての例を並列化可能であると正しく判定できた。このことは、半環上の線形式という抽象化が、リダクションループの自動並列化に際しては極めて有効な抽象化であることの一つの証左であると考えている。

現状の実装はコンセプト確認のためのものであり、さらなる議論にはより緻密な実装が必要であろう。特に、3.5 節で論じたとおり、提案手法で得られる並列プログラムにはかなりの冗長性が含まれるため、それが本当に効率に影響を与えないかは確認する必要があろう。また、適切なループ融合の方針についてもより詳細な検討が求められる。

謝辞 東北大学電気通信研究所共同プロジェクト研究集会での議論に感謝する。本研究は JSPS 科研費 18K18032 の助成を部分的に受けたものである。

参 考 文 献

[1] Farzan, A. and Nicolet, V.: Synthesis of divide and conquer parallelism for loops, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Cohen, A. and Vechev, M. T.(eds.), ACM, 2017, pp. 540–555.

[2] Farzan, A. and Nicolet, V.: Modular divide-and-conquer parallelization of nested loops, *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2019, Phoenix, AZ, USA, June 22-26, 2019*,

McKinley, K. S. and Fisher, K.(eds.), ACM, 2019, pp. 610–624.

[3] Fedyukovich, G., Ahmad, M. B. S., and Bodík, R.: Gradual synthesis for static parallelization of single-pass array-processing programs, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, Cohen, A. and Vechev, M. T.(eds.), ACM, 2017, pp. 572–585.

[4] Fisher, A. L. and Ghuloum, A. M.: Parallelizing Complex Scans and Reductions, *Proceedings of the ACM SIGPLAN'94 Conference on Programming Language Design and Implementation (PLDI), Orlando, Florida, June 20-24, 1994*, 1994, pp. 135–146.

[5] Geser, A. and Gorlatch, S.: Parallelizing functional programs by generalization, *Journal of Functional Programming*, Vol. 9, No. 6(1999), pp. 649–673.

[6] Gibbons, J.: The Third Homomorphism Theorem, *Journal of Functional Programming*, Vol. 6, No. 4(1996), pp. 657–665.

[7] Gulwani, S., Polozov, A., and Singh, R.: *Program Synthesis*, NOW, 2017.

[8] Jiang, P., Chen, L., and Agrawal, G.: Revealing parallel scans and reductions in recurrences through function reconstruction, *Proceedings of the 27th International Conference on Parallel Architectures and Compilation Techniques, PACT 2018, Limassol, Cyprus, November 01-04, 2018*, Evripidou, S., Stenström, P., and O'Boyle, M. F. P.(eds.), ACM, 2018, pp. 10:1–10:13.

[9] Matsuzaki, K., Hu, Z., and Takeichi, M.: Towards automatic parallelization of tree reductions in dynamic programming, *SPAA 2006: Proceedings of the 18th Annual ACM Symposium on Parallel Algorithms and Architectures, Cambridge, Massachusetts, USA, July 30 - August 2, 2006*, Gibbons, P. B. and Vishkin, U.(eds.), ACM, 2006, pp. 39–48.

[10] Morihata, A.: Lambda calculus with algebraic simplification for reduction parallelization by equational reasoning, *Proc. ACM Program. Lang.*, Vol. 3, No. ICFP(2019), pp. 80:1–80:25.

[11] Morihata, A. and Matsuzaki, K.: Automatic Parallelization of Recursive Functions Using Quantifier Elimination, *Functional and Logic Programming, 10th International Symposium, FLOPS 2010, Sendai, Japan, April 19-21, 2010. Proceedings*, Blume, M., Kobayashi, N., and Vidal, G.(eds.), Lecture Notes in Computer Science, Vol. 6009, Springer, 2010, pp. 321–336.

[12] Morita, K., Morihata, A., Matsuzaki, K., Hu, Z., and Takeichi, M.: Automatic inversion generates divide-and-conquer parallel programs, *Proceedings of the ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation, San Diego, California, USA, June 10-13, 2007*, Ferrante, J. and McKinley, K. S.(eds.), ACM, 2007, pp. 146–155.

[13] Sato, S. and Iwasaki, H.: Automatic parallelization via matrix multiplication, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, Hall, M. W. and Padua, D. A.(eds.), ACM, 2011, pp. 470–479.

[14] Suriana, P., Adams, A., and Kamil, S.: Parallel associative reductions in Halide, *Proceedings of the 2017 International Symposium on Code Generation and Optimization, CGO 2017, Austin, TX, USA, February 4-8, 2017*, Reddi, V. J., Smith, A., and Tang, L.(eds.), ACM, 2017, pp. 281–291.

[15] Xu, D. N., Khoo, S.-C., and Hu, Z.: PType System: A Featherweight Parallelizability Detector, *Programming Languages and Systems: Second Asian Symposium, APLAS 2004, Taipei, Taiwan, November 4-6, 2004. Proceedings*, Chin, W.-N.(ed.), Lecture Notes in Computer Science, Vol. 3302, Springer, 2004, pp. 197–212.

[16] 森畑明昌, 松崎公紀, 胡振江, 武市正人: 並列プログラムの候補生成と適合性検査による並列化, 情報処理学会論文誌：プログラミング, Vol. 2, No. 2(2009), pp. 132–143.