並列KLIC処理系上での配列演算の最適化

坂本幸司[†] 松宮志麻[†] 上田和紀[††]

本研究の目標は，並行論理型言語 KL1 の UNIX 上の処理系 KLIC 上で，単一代入変数の特徴を生かしたまま効率の良い並列配列計算を実現することである．KL1 の変数の特性を実現するために，KLIC では様々な工夫がなされており，配列はマルチバージョンベクタとして実装されている．この配列は任意のデータを要素として保持できるが，その代償として数値演算向きの最適化は十分なされていない．浮動小数点数がジェネリック・オブジェクトであるため，時間的・空間的に非効率である．特に共有メモリ型並列計算機（SMP）上の数値並列処理を考えると，KLIC のベクタは SMP 上で配列の共有がなされていない．そのためノード間で配列の大きさに比例する量の通信が発生して並列効果があまり望めない状態であった．そこで本研究では，破壊的代入，タグ判別の省略などで高速に読み書きできる数値演算用の配列をジェネリック・オブジェクトとして実装した．制約ベースの静的解析系による変数のモード /型/参照数情報と具体化状態に関する性質を併用して，プログラム中のベクタを数値演算用配列に安全に置き換えることにより効率改善を行った．また，新たに多次元配列を導入することで数値演算プログラムのより自然な記述が可能となった．これらの配列は，本体を SMP の共有メモリ上に配置することで，配列に対する並列アクセスの安全性をプログラムレベルで明示したまま高度な並列化が実現できた．さらに，末尾呼び出しループに対して最適化の技法を併用することでさらに 5〜8 倍の性能向上を実現でき，手続き型プログラムに近い目的コードの生成を可能とした．

Optimizing Array Processing of Parallel KLIC

Koji Sakamoto,[†] Shima Matsumiya[†] and Kazunori Ueda[††]

The aim of this research is to establish an efficient parallel array processing paradigm on KLIC, an implementation of concurrent logic programming language KL1 for UNIX, keeping the characteristics of single-assignment variables. In order to implement the characteristics of single-assignment variables efficiently, various techniques are used in KLIC. One of the notable techniques is arrays implemented as multi-version vectors. However, multi-version vectors accommodate arbitrary data as their elements, and are not efficient enough for numerical computation because floating-point numbers are represented as generic objects. Another problem with KL1 vectors is that they do not allow efficient parallel operations on shared-memory multiprocessors. In this study, we have implemented a new array construct specialized for numerical processing as generic object, in which array elements are overwritten and tag checking is omitted. By static analysis of variable types, linearity and instantiation states, we have safely replaced vectors with our new arrays thereby gaining more efficiency. We also introduced multi-dimensional arrays and made it possible to program numerical computation more naturally. Moreover, by allocating array elements on the shared memory of a symmetrical multiprocessor (SMP) system, we provided a framework for writing efficient parallel programs naturally. The application of loop optimization lead to 5–8 times of further speedup and allowed us to generate code closer to that of procedural programs.

1. はじめに

並行論理型言語 KL1[1] は論理型言語に同期機構を導入して並行処理記述のための言語としたものである．簡潔な通信・同期機構を持ちながら，構成が動的に変化する並行プロセス系通信プロトコルを自然に記述できる柔軟性もあわせ持つ．

論理型言語の特徴として，記号アトムの機構や自動メモリ管理機構を持ち，並行プロセスの記述のために，データフロー同期機構や物理的並列性のプラグマによる指定などの特徴を持つ．

本研究では KL1 の実装である KLIC[2] 並列処理系において，制約概念に基づく静的解析によって得られるモードや型情報と，抽象解釈によって得られる変数の具体化状態に関する解析情報を併用することで，動

† 早稲田大学大学院理工学研究科
Graduate School of Science and Engineering, Waseda University
†† 早稲田大学理工学部
School of Science and Engineering, Waseda University

的な判断の多くを省略する．あわせて制約概念に基づく静的解析によって得られる参照数情報を用いて，配列の単一参照性を保証したまま配列要素への破壊的代入を実現する．以上の手法による配列演算の最適化を提案・評価する．

KLIC 逐次処理系における同様の最適化については，文献 3)，4) が本研究の先行研究であり，並列化以外に関する基本方針はそれらに立脚している．

1.1 KL1 と KLIC 処理系

KL1 プログラムは以下の形をしたガードつき節の集合である．ここで，h を一階述語論理における原子論理式，G と B を原子論理式のマルチ集合とする．

h :- G | B.

h を節の頭部，G をガード，B をボディと呼ぶ．

KL1 プログラムの実行の単位はゴールであり，ゴールとは述語名と引数並びを合わせた原子論理式である．ガードつき節はゴールの書換え（リダクション）規則を表している．

頭部に同一の述語記号 p を持つすべての節の集合を p の定義という．頭部はその節を用いてリダクションできるゴールの形を指定している．ガードはその節を選んでよいかどうかの付帯条件を指定する条件部であり，ボディはその節を選んだ後に生成されるゴール群を指定する部分である．

プログラムの実行は 1 つ以上の初期ゴールをゴールプール（書き換えるべきゴールのマルチ集合）に入れて開始される☆．その中から 1 つのゴールを取り出し，対応するヘッドを持つ節のうちガード条件を満たす節を 1 つだけ☆☆選んで選ばれた節のボディに書き換え，書き換えられたゴールをゴールプールに戻す（リダクション）．これを繰り返し，ゴールプールに書き換えるべきゴールがなくなれば実行を終える．異なるゴールのリダクションは並列に実行してもよい．

t_1 = t_2 のような等号（“=”）の両辺に値を書いたゴールを単一化（unification）と呼ぶ．ガードでは条件指定，ボディでは値の決定を意味する．

述語には，言語仕様で定義される組込み述語と，プログラム中で定義するユーザ定義述語の 2 種類がある．

複数のゴールで同じ変数を共有し，変数に値が入らなければガードの条件が判断できないときにはゴールの実行を見合わせ，中断（suspension）させる．他のゴールの実行により中断原因の変数の値が決まれば，実行を再開させる．これが通信と同期の機構である．

☆ KLIC では引数を持たないゴール `main` を初期ゴールとする．
☆☆ 複数あれば，どれが選ばれるかは分からない．

KLIC 処理系は KL1 から C へのコンパイラと実行時支援からなる．論理型言語の実装として比較的高速であり，次節で述べるジェネリックオブジェクトと呼ばれる言語拡張機能を持つことを特徴とする．

本実験のプラットフォームは，共有メモリを通信路として用いる並列 KLIC 処理系（Distributed KLIC と呼ばれている）である．並列処理系はデータを共有メモリ上に配置しノード間で共有する版（Shared-Memory KLIC と呼ばれている）もあるが，Distributed KLIC を選んだのは代表的な共有メモリ並列計算機上で安定して稼働し，かつバージョンが新しいからである．

1.2 配列を用いたプログラム例

KL1 では，ベクタ（vector）と呼ばれる言語機能が 1 次元配列の役割を持つ．

たとえば次のプログラムは配列 `V` の `V[K]` から `V[N]` まで（配列の添字は `0` から始まる）の要素（浮動小数点数）の和を求める述語 `sigma/5` を定義したモジュールである．

```
:- module vsig.

sigma(K, N, V, Tmp, Out):- K > N |
        Out $:= Tmp.
sigma(K, N, V, Tmp, Out):- K =< N |
        K1 := K + 1,
        vector_element(V, K, F, V1),
        Tmp1 $:= F + Tmp,
        sigma(K1, N, V1, Tmp1, Out).
```

`sigma(0,2,{1.0,2.0,3.0},0.0,Out)` なる呼び出しが行われると，述語 `sigma/5` の 2 節目を用いてリダクションされ，4 つのボディゴールの実行が試みられる．KLIC 処理系においては “書かれた順番” で実行を試みてその結果，`K1` に `0 + 1` が代入され，配列 `{1.0,2.0,3.0}` の `0` 番目の要素 `1.0` と `F` が単一化し，配列 `V1` に `{1.0,2.0,3.0}` が返り，`Tmp1` に `1.0 + 0.0` が代入される．そして，`sigma(1,2,{1.0,2.0,3.0},1.0,Out)` が呼び出される．同様にあと 2 回 `sigma/5` の 2 節目を用いてリダクションされる．最後に `sigma/5` の 1 節目を用いてリダクションされ，`Tmp` を `Out` に代入して実行が終了する．

KLIC では組込みの基本データ型をタグを使って表現しているので，それ以外のデータ型はジェネリックオブジェクト[2] という拡張機能で実装されている．ジェネリックオブジェクトの機構を利用するとユーザは容易にシステムを拡張することができる．ジェネリックオブジェクトのうち具体化されているデータを実現するオブジェクトがデータオブジェクトである．組込みデータ型であるベクタや浮動小数点数（float）はデー

タオブジェクトで実現されている．本研究でもこの機構を利用して数値演算用の配列を実装している．

データオブジェクトは，自らの振舞いを記述したメソッドの表へのポインタと，オブジェクトのデータ領域（もしくはデータ領域へのポインタ）からなる構造体である．この構造体は，自動メモリ管理の対象となるヒープ領域に置かれる．

上の例の sigma/5 中の vector_element/4☆はベクタを操作する組込み述語☆☆で，ベクタ V の K 番目の要素を F と単一化し，V と同じベクタを V1 に返す．

ほかにもベクタを扱う以下の組込み述語がある．

new_vector(V, S) 引数 S が整数型の場合要素数 S のベクタを生成し，全要素の初期値を 0 として，引数 V に返す．引数 S がリストであった場合，リストをベクタに変換して引数 V に返す．

set_vector_element(V, I, E, NE, NV) ベクタ V の引数 I によって指定される要素 E（省略可能）を引数 NE に書き換え，新しいベクタを引数 NV に返す．

vector_split(V, I, VL, VU) ベクタ V を I 番目より前と以降とに分割し，2 つのベクタ VL，VU を返す．

vector_join(VL, VU, V) 2 つのベクタ VL，VU を結合したベクタ V を返す．

この中で，split，join は配列を論理的に分割・結合を行うもので配列の 2 つの部分に対する並列更新の安全性をプログラム上で明示する効果を持っているが，実装レベルではベクタの全要素のコピーによって実現されており，効率面での問題がある．

論理変数の単一参照性を実現するために，ベクタの要素を更新する場合，元のベクタを残しておくために shallow binding を用いるマルチバージョン管理を行っている．ベクタ 1 要素の参照/更新は時間・空間計算量 O(1) で行え，古いベクタの参照は更新回数を k とするとき O(k) で行える．

2. 静的解析情報による処理系最適化

本章では，制約をベースとするモード /型/参照数解析，および抽象解釈による具体化状態の解析手法について述べ，さらにそれらに基づくコントロールフローの最適化について述べる．ここで紹介する最適化手法は，主として配列と末尾呼び出しループを用いた数値演算のために導入するものである．配列機能の最適化については 3 章で詳述する．

本章の 2.2 節以降の内容は，文献 4) に基づいていくつかの改良と詳細化を行ったものである．

2.1 静的解析

KL1 プログラム静的解析系 klint 第 2 版[5),6)] によって以下の解析が実装されている．

- モード解析
  プログラムのデータの流れ（モード）に関する解析を行う．解析結果は述語の各引数の入出力モードを判定する．
- 型解析
  変数のとりうる型に関する解析を行う．解析結果は述語の各引数のとりうる型を限定する．
- 参照数解析
  複数の参照ポインタを持ちうる（nonlinear な）データと，単一の参照ポインタしか持たない（linear な）データとを区別する．

例として 1.2 節で取り上げたモジュール vsig を klint で解析した解析結果を提示する．以下が klint による解析結果の出力である．

```
%%% Mode %%%
:- mode vsig:sigma(-1,2,++,-3,4).
:- modedef 1 = (-,[]).
:- modedef 2 = (+,[]).
:- modedef 3 = (-,[]).
:- modedef 4 = (-,[]).

%%% Linearity %%%
:- lin vsig:sigma(1,?,2,4,5).
:- lindef 1 = (?,[]).
:- lindef 2 = (?,vector(**,[])).
:- lindef 3 = (?,vector(**,[])).
:- lindef 4 = (?,[]).
:- lindef 5 = (?,[]).
:- lindepend 2 => 3.
:- lindepend 3 => 2.

%%% Type %%%
:- type vsig:sigma(1,2,3,5,6).
:- typedef 1 = ([integer],[]).
:- typedef 2 = ([integer],[]).
:- typedef 3 = ([vector],vector(4,[])).
:- typedef 4 = ([float],[]).
:- typedef 5 = ([float],[]).
:- typedef 6 = ([float],[]).
```

klint の解析結果はプログラムが生成するデータ構造のすべての部分構造に言及するので多少複雑な形式をとる．以下に解析結果を簡単に説明する．詳細は文献 5)，6) を参照してほしい．

Mode，Linearity，Type がそれぞれモード解析，

☆ KLIC においては vector_element は 3 引数のものしかないが，本研究ではベクタの単一参照性の保証を容易にするために第 4 引数を追加している．4 引数の vector_element は，同じ目的で並列推論マシン用の KL1 の実装には存在していた．

☆☆ データオブジェクトを扱う組込み述語はコンパイル時にメソッド呼び出しにマクロ展開される．

参照数解析，型解析の解析結果である．`:- mode`, `:- lin`, `:- type` の行は述語の各引数のモード，参照数，型を番号および記号で表現している．番号で表現したモード，参照数，および型は，`:- modedef`, `:- lindef`, `:- typedef` で定義したものを参照している．

モード解析では，`:- modedef 1 = (-,[])`. は，トップレベル（主関数記号）が出力（“-”）であり，トップレベル以外の関数記号の入出力には制約がない（“[]”）ようなモードに番号 1 を与えて定義している．`:- mode` 行の負の番号 `-n` は，モード `n` と入出力が正反対のモードを表す．よってモード情報をまとめると，`:- mode vsig:sigma(-1,2,++,-3,4)`. は，第 1，2，4 引数のトップレベル（主関数記号）が入力，第 5 引数のトップレベルが出力であることを示している．第 3 引数の “`++`” は，すべてのレベルの関数記号が入力であることを示している．

参照数解析では，`:-lindef 1 = (?,[])` の “`?`” は，この述語が原因となってトップレベル構造体が複数参照になることがないことを表し，“`[]`” はトップレベル以外の構造体も複数参照になることがないことを表している．`:-lin` の行の第 2 引数の “`?`” も，この述語が原因でトップレベル以下が複数参照になることがないことを表す．参照数定義 2 と 3 の中の `vector(**,[])` は，データがベクタに具体化している場合，その要素が複数参照（“`**`”）になりうることを表す．

`:- lindepend 2 => 3`. は，含意形式の制約であり，参照数定義 2 を参照している引数のデータ全体もしくはその一部が（他の述語が原因で）複数参照であるならば，参照数定義 3 を参照している引数のデータが対応する部分も複数参照と見なす必要があることを示す．

結局 `sigma/5` では，複数参照となりうる唯一のデータはベクタの各要素だけであることを，klint の解析結果は表している．ベクタの要素が複数参照となりうるのは，`vector_element/4` によって取り出した要素 `F` が，`vector_element/4` の返すベクタ `V1` からも間接的に参照されていて，別の `vector_element/4` の呼び出しによって取り出すことが可能だからである．

型解析では，`:- type vsig:sigma(1,2,3,5,6)`. は，第 1，2 引数が整数値型，第 4，5 引数が浮動小数点数型，第 3 引数は浮動小数点数型を要素とする配列型と解析されたことを表している．

2.2 データフローに沿ったボディゴール順序

モード解析により得られる述語の引数の入出力モード情報から得られるデータフローに沿って述語定義のボディゴールを並べ替えることで，冗長な中断ゴールの生成を抑制する方法を示す．

冗長な中断ゴールの生成・操作を抑えるために，以下のようにボディゴールを並べ替える．

ボディ内に出現するゴールは，モード解析によって判明する各引数の入出力モード情報および共有変数の出現に基づき，順序関係をつけることができる．つまり，ボディゴール中の出力出現の変数に注目し，これら出力出現の変数を入力として受け取るゴールを検出することにより，ゴール間の依存関係を生成することができる．

また，引数の具体化状態により inline 実行が可能となる組込み述語は，この依存関係を保つ範囲で優先的に実行するようにする．さらに，ゴール間の依存関係が循環するような場合，つまり先に観測されるゴール間の依存に逆らう依存が後に観測される場合は，先に観測される依存を優先することにする．

このようにしてボディゴールの半順序集合を作成し，トポロジカルソートにより並べ替えることにより，ボディゴールの全順序関係を得ることができる．プログラム中に依存関係の循環がない限り，この全順序関係はデータフローに沿った実行順序となっている．

2.3 具体化状態の抽象解釈

KLIC 処理系では，変数の値を参照する場合には必ず型と具体化状態をタグ判別により動的に検査している．しかし，述語呼び出し時の引数の型は型解析によりある程度静的に解析可能であり，具体化状態もこの節で述べる手法である程度静的に解析することが可能である．この型と具体化状態の情報をコンパイル時に実行コードに反映させることで，動的なタグ判別がある程度省略できるようになる．

具体化状態の解析は以下で述べる抽象解釈により行うが，対象のプログラムは 2.2 節で述べたデータフローに沿ったボディ実行順序を仮定するものとする．

抽象解釈を行うプログラム内の述語群の各引数に対し具体化状態を表す抽象領域 $\{ground,nonground\}$ を定義する．各引数は抽象実行の過程においてこの領域上の *ground* か *nonground* かどちらかの状態をとる．ベクタやファンクタなどの構造を持った引数に対しては，すべてのレベルが具体化している状態を *ground*，そうとは限らないとき *nonground* であるとする．

抽象領域 $\{ground,nonground\}$ に対して，呼び出し時の引数の具体化状態から，終了時の出力引数の具体化状態を抽象実行により求めてゆく．単一代入変数であるから，一度 *ground* になると *nonground* になることはありえない．組込み述語は呼び出し時の引数の

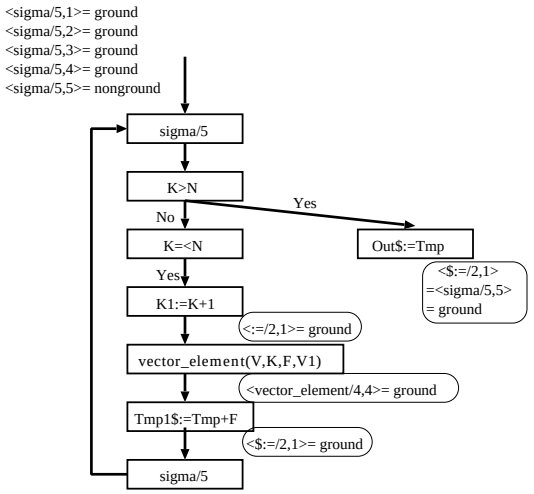


図 1 抽象解釈による引数の具体化情報の流れ
Fig. 1 Flow of the groundness information of arguments.

具体化状態と終了時の引数の具体化状態との関係が既知であるのでその情報を与えておく．ユーザ定義述語に関しては終了時の引数の具体化状態を求めるのに，対象述語のボディ内で出現するゴールの実行終了時の引数の具体化状態を用いる．

抽象実行を行う順序は，実際に KLIC 処理系で実行される順序に従う．ボディ内のゴールから呼び出されるユーザ定義述語に関しても同様で，ゴールが中断した場合は，その述語の呼び出し時の引数の具体化状態が中断時の具体化状態にあたる．

抽象解釈は，再帰を行う述語に関して不動点計算を適用する．停止性については，抽象領域の大きさと述語の個数がともに有限であることから保証できる．

2.3.1 具体化状態の解析例

1.2 節で例示したプログラムに対して，具体化状態に関する抽象解釈を適用した例を示す．このプログラムから得られる述語呼び出し間の引数具体化情報の流れは**図 1** のようになる．

ボディ内における引数情報はフローに沿って伝搬する．また，述語 `sigma/5` 内の自己再帰呼び出しは，外から呼び出された場合の引数情報と具体化状態が変わらないため，不動点に達している．

具体化状態の情報および型情報をまとめると，次のようになる．

- `sigma/5` の外からの呼び出し時に入力引数は 4 つすべて具体値であり，順に，整数型，整数型，配列型，浮動小数点数型である．
- `sigma/5` の 1 節目のガードを通過した場合，入力である 4 引数が順に，整数型，整数型，配列型，浮動小数点数型の具体値であるので，終了時には出力引数の第 5 引数も浮動小数点数型の具体値である．
- `sigma/5` の 2 節目のガードを通過した場合，組込み述語の出力引数はすべて具体値をとり，`sigma/5` への自己再帰呼び出しの入力引数はすべて具体値であり，順に，整数型，整数型，配列型，浮動小数点数型である．
- `sigma/5` の述語内では，配列型は要素まで具体化している *ground* である．

2.3.2 タグ判別の省略

ゴール g が述語 p を呼び出す際の引数の具体化状態と型情報が静的に解析できれば，述語 p はゴール g からの呼び出し時に動的な検査である引数のタグ判別を省略できる．

KLIC 処理系においてタグ解析を省略した実行コードを生成するには，述語呼び出し間の引数情報に関する抽象解釈結果を，KL1 コンパイラが生成する中間コードに反映させる必要がある．引数情報を反映する中間コードとして，1 つの述語定義に対して，引数情報を動的に判別する従来のコードと，呼び出し時の引数の具体化状態と型が解析された場合に述語ごとに動的な判別を省いたコードの 2 種類を生成し呼び出し元の述語によって使い分ける．

2.4 末尾再帰呼び出しループの最適化

KL1 言語における末尾再帰（tail recursive call）もしくは，相互末尾再帰（mutually tail recursive call）の形で記述されるループ処理の最適化について述べる．

述語ボディの最後のゴールから再帰を行うような述語に関して，静的解析による引数の具体化状態および型情報を用いて，タグ付き整数型や，浮動小数点数型などで box 化されている KL1 の数値データをループ処理内部で unbox 化したまま扱うことで，繰り返しごとの unbox 化，box 化の操作を削減する．同時に，メソッド呼び出しを inline 展開することで，メソッド呼び出しにかかるオーバヘッドを削減できる．特に浮動小数点数を要素として持つ配列のループ最適化は効果が期待できる．

この最適化の対象とする述語は，

- ボディの末尾において再帰呼び出しを行う述語，もしくは相互末尾再帰を行う述語の組で，
- 相互末尾再帰の場合は，外部から呼び出されるのはそのうちの 1 つの述語に限定され，もう片方の述語の末尾再帰呼び出しによる中断の可能性がなく，
- これらの述語（の組）のボディを構成する他のゴー

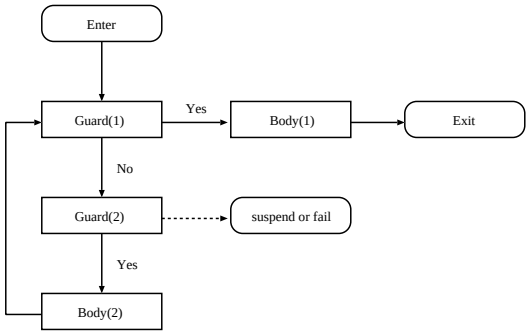


図 2 一般的な自己再帰型述語のコントロールフロー
Fig. 2 Control flow of general self-recursive predicates.

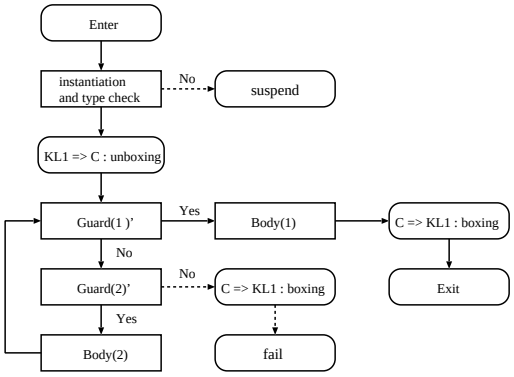


図 3 最適化した自己再帰型述語のコントロールフロー
Fig. 3 Control flow of optimized self-recursive predicates.

ルも中断する可能性がないもの
とする．

ここで，中断する可能性がないゴールとは，入力引数がすべて具体化しており，ガード検査で必ず 1 つ以上の節が選択されリダクションが行われ，その結果得られるゴールも，すべて中断する可能性がないゴールであるという意味である．

2.4.1 中間コードにおけるコントロールフローの変更

KLIC 処理系において，自己再帰型のループ処理を行う述語のコントロールフローの例として**図 2** のようなものがあるとする．この例では，述語宣言の 2 節目において実際のループ条件とループ処理が記述されている．

そこで，ループ内部のガードで行われている具体化状態と型に関する検査をループの直前に移動させることができれば，ループ内部で引数を unbox 化したまま扱うことができるようになる．そのために，**図 3** のようにコントロールフローを変換する．

図 3 では，図 2 中の `Guard(1)`，`Guard(2)` の条件の一部である具体化状態と型の検査をループの直前に移動させており，図 3 中の `Guard'` はそれぞれ図 2 中の `Guard` から具体化状態と型の検査をのぞいたものである．ループの入り口で unbox 化を行い，ループの中では高速な処理を実現している．ループから出るときに box 化を行う必要がある．

図 3 中の `Guard(2)'` からの破線のフローは 2 つのガード条件のどちらかを必ず満たすループであれば存在しない．

2.4.2 同期ポイントの移動

上に述べたコントロールフローの変更は，具体化状態の検査のタイミングをループ内からループの実行直前に移動している．これを同期ポイントの移動[7] という．

一般には同期ポイントの移動を行うことにより，コンパイラが付加した動的具体化検査によってリダクションが妨げられ，プログラム自体が正常に動作しない場合がある．しかし文献 7) の技法を用いることにより，プログラムの意味を変えることなく同期ポイントを移動することのできる変数の集合を，不動点計算によって求めることができる．

2.5 ま と め

本章では，制約をベースとした静的解析と抽象解釈によって得られる引数の具体化情報とを併用する最適化手法について述べた．KLIC 処理系にこれらの最適化を施した場合，次のような順序でコンパイル処理が行われることになる．

(1) モード /型/参照数解析
述語引数の入出力モード情報，データの型情報，データの参照数が分かる．

(2) データフローに沿ったボディゴールの配置
モード解析の結果により，データフローに従ってボディゴールを入れ替える．

(3) 具体化状態の解析
抽象解釈により述語呼び出し間の変数と，ボディ実行中の変数の動的な具体化状態が分かる．動的なタグ判別が省略できる．

(4) ループ処理最適化
抽象解釈の結果から，末尾呼び出しのループ処理を行う述語について，変更可能ならばコントロールフローを変更する．ループ外に移動できる具体化状態の検査をループ外に移動させる．

(5) 中間コードの生成
以上の最適化を施した中間コードを生成する．

3. 数値演算用配列を用いた最適化

KL1 で数値演算を行うときには配列として vector

を使用してきたが，KLIC 処理系での実装では，vector は任意のデータを要素として保持できる反面，数値演算向きの最適化は十分になされていない．また，浮動小数点数（float）自身がデータオブジェクトであるため，時間的・空間的に非効率である．

特に Distributed KLIC では，共有メモリ型並列計算機（SMP）上でさえも配列の共有がなされていないため，ノード間で配列の大きさに比例する量の通信が発生して並列効果があまり望めない状態であった．一方，Shared-Memory KLIC ではデータを共有メモリ上に置いてノード間で共有するが，共有メモリ実装は分散メモリ実装と比較して記憶管理が複雑となり，安定した実装が困難である．しかし，我々はあらゆるデータを共有メモリ上に配置し，共有する必要はさほどないと考える．共有することで大変効果が期待できる配列のようなデータを共有し，残りは分散管理する方が実装が容易であり，効率も良いと考える．

よって本研究では，vector の全要素に float をとるデータ構造に代わる最適化されたデータ構造として，Distributed KLIC へ新たな数値演算用の配列（float_array）を実装した．また，制約ベースの静的解析による型/参照数の情報により，vector の要素の型を特定するとともに vector の単一参照性を保証することで破壊的更新の安全性を保証して，vector を float_array に置き換えることで配列演算の効率化に成功した．

本章では SMP 上の Distributed KLIC での float_array 型の実装について述べ，さらに配列の多次元への拡張を提案する．

3.1 実装方針

従来の配列の要素として浮動小数点数をとり数値演算を行う場合の問題点をまとめる．

- vector はマルチバージョン管理を行い，また不完全データ構造を格納できるが，これらは数値演算にはほとんど必要がなく，オーバヘッドの原因となる．
- Processor Element（PE）の局所ヒープ領域に配列本体を置くため，copying garbage collection（GC）ごとに配列本体のコピーが起きる．
- 添字検査をメソッド内で行っているが，その多くは静的解析に基づいて省略できると期待される．
- 配列の分割・結合に配列要素のコピーを用いている．
- SMP 上の並列処理系でも配列本体が PE 間で共有されていない．

配列を用いた数値演算を行う場合を考えるとほとんどの場合は，vector の各要素は数値のみの即値であることが分かっており，vector が持つ特徴であるマルチバージョン管理や不完全データ構造などは実行効率を悪くする要因になることが多い．

たとえば，巨大な行列の 2 項演算を配列を用いて行うときは，演算の対象となる行列を格納した配列 `ProbA`，`ProbB` と，初期値（たとえば `0`）で初期化された演算結果を格納するための配列 `Ans` を用意し，計算結果は配列 `Ans` の要素を書き換えて格納してゆく．2 項演算のプロセスでは配列 `ProbA`，`ProbB` の要素の更新が行われず，`Ans` は単一参照性を保つようなプログラミングが可能である．このようなスタイルをとると，各配列はマルチバージョン管理を行う必要がなく，不完全データ構造も許す必要がない．

ところが，巨大な配列を vector として扱うと，KLIC 処理系がメモリ管理を行うヒープ領域を大量に消費して GC 起動回数を増加させ，GC 起動ごとに全要素のコピーを行うので効率が悪くなる．

また，vector において提供されている要素参照および書き換えを行うメソッドは，内部で必ず上限下限の添字検査をする．けれどもメソッド内で必ず行われる添字検査の多くは，適切な場所で添字のチェックを行うようプログラムを組むことで無駄になる場合が多い．添字検査を削除する手法[8),9)] も研究されており，メソッド内で必ずしも逐一検査する必要はない．

これらの問題点を解決するために数値演算用配列（float_array）を次の方針により実装する．

- 要素はすべて即値の 64 bit 浮動小数点数．
- 要素の書き換えは破壊的に行う．
- 配列本体はヒープ外の共有メモリ上専用領域に配置．
- 要素の参照/書き換えごとにメソッドの中では添字検査は行わない．
- 分割・結合の際可能な限り要素のコピーを行わない．
- 配列本体をすべての PE から参照できる共有メモリ上に確保し，共有可能にする．

破壊的代入による要素書き換えを行うので，参照数解析により単一参照性が保証された配列に対してのみ安全に使える．ただし単一参照を保証しなければならないのは配列の本体のみであり，配列の要素は単一参照である必要はない．

添字検査に関して，我々の方針は，

- メソッドの中には添字検査は含めない，
- 必要な添字検査は目的コードの適切な場所に別途挿入する，

表 1 vector と float_array の特徴
Table 1 Features of vector and float_array.

	vector	float_array	安全性の解析
配列の要素	任意データ （未定義変数可）	浮動小数点数に限定 （未定義変数不可）	型解析， 参照数解析
要素の書き換え	shallow binding による マルチバージョン管理	破壊的代入	参照数解析
添字検査	参照/書き換えごとに メソッド内で逐一検査	プログラム中の適切な箇所で 最小限の検査を別途挿入	添字検査系
配列本体の配置場所	局所ヒープ	ヒープ外の共有メモリ上専用領域	—
分割・結合	配列本体のコピーで行う	可能な限りポインタ操作で行う	参照数解析
配列本体のノード間共有	共有不可能	共有可能	—

というものである．KL1 用の添字検査系の実装はまだないが，近い将来解析手法が確立できることを期待しての実装方針である．本論文での性能評価ではメソッドの中に添字検査を含めたデータを測定した．

分割・結合に関しても，vector では分割・結合前のデータをそれぞれ残したまま新しい領域を確保して要素のコピーを行うが，float_array の分割・結合は配列本体へのポインタを操作するのみで，できる限り要素のコピーは行わない破壊的な分割・結合を行う．要素の書き換えと同様に配列の単一参照性を保証する必要がある．

vector の要素は他の PE から参照できない各 PE で独立した局所ヒープ領域に配置されるため，他 PE へ配列を渡すには全要素のコピーを行わざるをえない．float_array は配列の要素をコピーせずに配列の分割・結合，PE 間の受け渡しが可能なので，分割・結合，PE 間の受け渡しの手間が O(1) となる．

vector と float_array の比較を**表 1** にまとめた．

3.2 データ構造とメモリ配置

データオブジェクトとして実装した float_array の構造体定義を示す．

```
struct float_array_object {
  struct data_object_method_table
      *method_table;
  int size;
  double *body;
};
```

構造体メンバ method_table が float_array のメソッド表へのポインタであり，size は配列の要素数を表し，body は共有メモリ上の配列要素へのポインタである．float_array のデータ構造を**図 4** に示す．

float_array のメモリへの配置方法は，オブジェクト本体（構造体）は KLIC の管理する各 PE 個別のヒープ領域に配置するが，構造体メンバの body が指す double 型配列は共有メモリ上に確保する．そのためにあらかじめ（プログラム起動時に）共有メモリ中に double 型配列専用のメモリ領域を mmap 関数により確保しておき，オブジェクトの生成時に共有メモリ中に配列領域を確保する．つまり，複数 PE 間でオブジェクトを移動させる場合，float_array_object は移動させるが，double 型配列は共有メモリ上で共有されている．

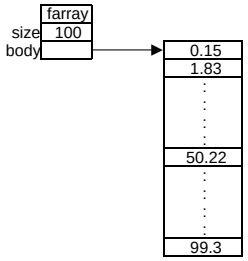


図 4 float_array 型のデータ構造とメモリ配置
Fig. 4 Data structure and memory allocation of the float_array type.

mmap 関数により確保した領域は，KLIC のヒープ領域外であり GC の起動に影響を与えない．けれども，double 型配列がゴミになった場合 GC により領域を解放されないという欠点がある．しかし次のような対策によって領域の解放が行えることを提案しておく．

(1) 参照数解析により領域が不要になるタイミングが分かる場合，コンパイル時に領域を解放するコードを付加する．
(2) 動的な参照数カウンタによって領域を解放できるかつねに監視する．

(1) の方法により，少なくとも参照数解析で linear である配列は処理系により解放を行うことができる．参照数カウンタは，MRB（multiple reference bit）と呼ばれる 1 ビット参照カウンタを動的に管理する方式が提案され，並列推論マシン上の KL1 処理系にも実装された．KLIC 処理系においては，時間的・空間的効

率の問題から動的な参照数カウンタは導入せず，GC によってメモリ管理を行っている．しかし，大規模なメモリ消費が起こりやすい配列型に関しては空間的効率を優先させ，共有メモリ上の配列データのみを対象とした動的な参照数カウンタの導入も考えられる．

3.3 提供するメソッド

本節では，float_array を操作する組込み述語について説明する．処理系内部ではこれまでに述べたように明確な違いがあるが，表記上 vector の操作述語に類似している．データオブジェクトであるため組込み述語の実体はメソッド呼び出しである．

`new_float_array`(A, S)

引数 S が整数型の整数値の場合は，要素数 S の配列を生成し，配列の全要素の初期値を 0.0 として，引数 A に返す．

引数 S がリストであった場合は，リストを配列に変換して引数 A に返す．ただし，リストの構成要素はすべて float 型である必要があり条件を満たさない要素がリスト内にあった場合，実行を中止する．

`float_array_element`(A, I, E, NA)

配列 A の引数 I によって指定される要素を引数 E に返す．引数 NA は配列 A と同じものを返すが省略可能．

`set_float_array_element`(A, I, NE, NA)

配列 A の引数 I によって指定される要素を引数 NE に書き換え，新しい配列を引数 NA に返す．vector と違い不完全データ構造を許さないため，NE が変数参照の場合は中断する．また，具体化されていても，float 型でない場合は実行を中止する．

`float_array_split`(A, I, LA, UA)

配列 A を引数 I 番目で分割し，2 つの配列 LA, UA を返す．配列 LA には 0 番から $I-1$ 番目までの要素が入り，UA にはその残りが入る．

実装レベルでは，オブジェクト内の `body` ポインタと `size` によって分割させる．共有メモリ上での `split` を**図 5** に示す．

`float_array_join`(LA, UA, NA)

2 つの配列 LA，UA を結合し，配列 NA へ返す．

実装レベルでは，LA と UA の `double` 型配列が連続したメモリ領域にマップされているときにはオブジェクト LA の `size` の変更によって結合する．それ以外の場合は，`double` 型配列をそれぞれ連続したメモリ領域にコピーすることで物理的に結合させる．つまり，あらかじめ `split` した配列を `join` するのであれば，`split/join` のコストは O(1) である．

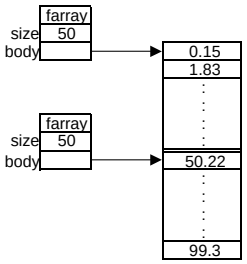


図 5 `float_array` の split
Fig. 5 Split opereation of `float_array`.

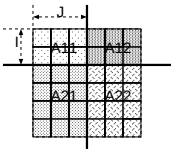


図 6 `float_2d_array` の split
Fig. 6 Split operation of `float_2d_array`.

3.4 多次元配列への拡張

KL1 では多次元配列は入れ子の配列として扱われてきた．

多次元配列の各要素にアクセスするのに，n 次元の配列ならば n 回，配列の要素を参照しなければならなくなり，配列の `split/join` を行うにも，内側の要素配列をそれぞれ `split/join` する必要があり大変効率が悪い．

そこで，配列 `float_array` を拡張して多次元配列の中間表現としての要素配列を使わずに直接，各要素にアクセスできる `float_nd_array`（n 次元）を提案する．今回実装した 2 次元配列のメソッドを以下に示す．

`new_float_2d_array`(A, M, N)

引数 M，N を整数型で与えると，要素数 $M \times N$ の 2 次元配列を A に返す．

以下同様に 2 次元に拡張する．

`float_2d_array_element`(A, I, J, NE)

`set_float_2d_array_element`(A, I, J, E, NA)

`float_2d_array_split`$(A, I, J, A11, A12, A21, A22)$

A は I，J で指定された要素番号で**図 6** のように分割された 2 次元配列 ($A11$，$A12$，$A21$，$A22$) を生成する．

`float_2d_array_join`$(A11, A12, A21, A22, A)$

2 次元配列 $A11$，$A12$，$A21$，$A22$ をつなげて A とする．

4. 評　　価

本研究で提案する手法の有効性を確かめるために，以下の 2 種類のプログラムを用いて，共有メモリ並列計算機上で性能評価を行った．

qsort　配列要素のクイックソート．入力は 400,000 個の浮動小数点数を要素とする配列．

mul_mat　浮動小数点数を要素とする 200×200 の 2 次元配列を 3 面用いた行列積演算．

評価は Sun Enterprise 4000（MPU 167 MHz（512 KB external cache）× 10 PE，Memory 1,280 MB）上で計測し，gcc コンパイラの最適化レベルはすべて `-O2` を用いた．それぞれ 5 回ずつ計測した平均値である．

4.1 クイックソートの最適化効果

はじめに float_array 型の実装による実装効果を測定する．クイックソートのアルゴリズムはおおむね次のとおり．

(1) 配列 $A[i], 0 \leq i < n$ の中から枢軸（Pivot）要素を選ぶ．

(2) Pivot の値未満の要素が $A[0], \ldots, A[j-1]$ に，Pivot の値以上の要素が $A[j], \ldots, A[n-1]$ にあるように配列要素を入れ換え，$A[j]$ の前で A を split する．

(3) 分割後の 2 つの配列を再帰的にクイックソートする．

(4) ソート終了後，配列を join で結合する．

クイックソートの並列化は，(3) でまだ空いている PE があれば，片方の配列をゴールごとその PE に渡して処理を委託し（この委託は再帰的に行われる），両 PE でのソート結果をもとの PE に集めて join で結合することで実現している．

測定結果は**表 2** のようになった．実際に測定した時間は，ソートを行うループに入ってから出るまでの時間である．問題を読み込む時間などは含まれていない．実行時オプションは，with vector が `-h1m -B4m -e`☆，その他は，`-A4m`☆☆で測定した．

表 2 中のプログラムはそれぞれ以下の版である．静的解析手法のうち klint は実装済であるがそれ以外は現時点で未実装であり，klint も KLIC とはまだ統合されていない．そこで，これらの評価用プログラム各版は，KLIC の出力する C コードを，静的解析結果に基づいて手作業で調整することで得た．

with vector　モード解析によりデータフローに沿ったボディゴール順序を定め，配列に vector を使用した版．

with array　with vector に参照数解析と型解析を施して安全性を確かめて，配列 float_array を使用した版．クイックソートでは配列は単一参照となる．

loop-opt　with array に具体化状態の情報を併用してアルゴリズム中のステップ (2) で Pivot と配列要素を比較する述語（相互再帰呼び出し）のループ最適化を行い，あわせて引数のタグ検査を省略しジェネリックメソッド呼び出しをインライン展開した版．

最適化前の with vector と比較すると，with array は，PE 数=1 のときでも 14.2%の速度向上が得られており，さらに並列化したときの台数効果も改善されている．with array と loop-opt では，クイックソートのアルゴリズムから求めた 400,000 要素の台数効果の理論値に近い台数効果がでており，理想的な並列化が行われたといえる．

表 2　クイックソートにおける並列/最適化効果（qsort）上段は実行時間（秒）下段は台数効果（倍）
Table 2　Effect of parallelization and optimization of the quicksort program: execution time (sec.) and parallel speedup.

PE 数	1	2	4	8
with vector	29.47	17.31	11.65	8.22
（最適化前）	1	1.70	2.53	3.58
with array	25.28	13.38	7.39	4.78
	1	1.89	3.42	5.29
loop-opt	5.01	2.62	1.54	0.96
	1	1.91	3.25	5.12

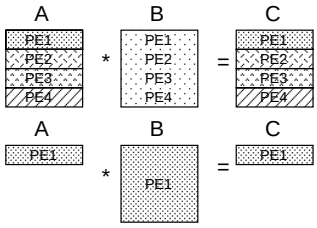


図 7　mul_mat プログラムにおける配列データの分割
Fig. 7　Dividing array data in the mul_mat program.

4.2 行列積の最適化効果

次に行列積の演算（mul_mat）における実行時間を測定した．mul_mat におけるデータ並列処理の方法を**図 7** で示す．配列 `C` は単一参照となる．

実行時オプションなしで測定した．測定結果は**表 3** のようになった．

表 3 中の計測対象は以下のプログラムの実行時間

☆ `-h` はワード単位の初期ヒープ領域量，数字の後の `m` は 2^{10} 単位の指定，`-B` はバイト単位の通信路バッファ領域量，`-e` はバッチ転送モードの指定．

☆☆ `-A` はバイト単位の float_array 用共有メモリ領域量の指定．

表 3 行列積の演算における並列/最適化効果（mul_mat）上段は実行時間（秒）下段は台数効果（倍）
Table 3 Effect of parallelization and optimization of the matrix multiplication program: execution time (sec.) and parallel speedup.

PE 数	1	2	3	4	5	6	7	8	9	10
with 2d_array	32.96	16.58	11.07	8.36	6.66	5.67	4.83	4.24	3.86	3.42
	1	1.99	2.82	3.94	4.94	5.81	6.82	7.77	8.52	9.63
loop-opt	4.11	2.08	1.41	1.04	0.85	0.72	0.62	0.54	0.51	0.44
	1	1.97	2.91	3.95	4.83	5.71	6.63	7.61	8.06	9.34
loop-opt no check	3.55	1.80	1.21	0.91	0.73	0.63	0.54	0.47	0.44	0.38
	1	1.97	2.93	3.90	4.89	5.63	6.57	7.55	8.07	9.34

（秒）と台数効果（倍）を示している．

with 2d_array モード解析によりデータフローに沿ったボディゴール順序に定め，参照数解析と型解析により安全性を確かめて，2 次元配列に float_2d_array を使用した版．

loop-opt with 2d_array に具体化状態の情報を併用して最内ループの最適化を行い，あわせて，引数のタグ検査を省略しジェネリックメソッド呼び出しをインライン展開した版．

loop-opt no check loop-opt に配列の添字検査系で解析が可能になった場合を仮定し，メソッドでの冗長な添字検査を省略した版．

行列積演算もほぼ理想的な台数効果が現れている．このプログラムにループ最適化を行うと台数効果を理想的に保ったまま，ループ最適化前の約 8 倍の性能が測定できた．

その結果 1 PE では 4.11 秒，10 PE では 0.44 秒となった．添字検査を省くとさらに 15%程度性能が上がる．ちなみに C 言語で記述した行列積演算の逐次プログラムの実行時間は 0.62 秒であり，KL1 プログラムを並列実行した場合にはこれを上回る性能が実測できた．KL1 プログラムがループ内で行う演算内容は C プログラムのそれにかなり近い．それにもかかわらず逐次性能には 5 倍以上の差があるが，この差を縮めるにはさらに細かなチューニングが必要である．

5. 関 連 研 究

Commited-Choice 型言語における配列処理最適化に関する研究としては，文献 10) で Fleng において，コピー除去とデータ並列化の研究がなされている．我々は，並列実行したいプログラムを divide and conquer 的に記述する方法と，それを並列実行する方法に焦点をあてている．並列性の抽出を目標とするのでなく，安全に並列実行できることが容易に見てとれるようなプログラム記述の方法を確立することを大きな目標としている点が異なる．

並行制約言語 Janus[11] は，変数の出現回数をちょうど 2 回に制限して破壊的更新を実現している．これは，構文的な制限によって，単一参照性が保証されるようなプログラムに限定していることに相当する．Janus はまた配列機能も提案しているが，要素アクセスを制約の枠組みで実現している．このため，配列本体の参照数については容易に推論・検査できるが，配列要素の参照数を推論・検査するのは容易でない．これに対して KL1 の `set_vector_element`/5 をはじめとするベクタ操作述語は，配列本体やその要素のモードと参照数の推論が容易になるように設計されている．

論理型言語においては，抽象解釈に基づく compile-time garbage collection としての研究は数多い（最近のものでは文献 12) など）が，言語レベルで破壊的更新を扱っている代表例は Mercury[13] である．Mercury では，unique mode 宣言とその静的検査によって，単一代入言語の枠組みの中で，破壊的更新を実現している[14]．Mercury 処理系の性能は非常に高いが，モードや型を詳細に宣言しなければならないので，プログラマの負担が懸念される．

純関数型言語においても配列の破壊的更新に関する研究は多いが，並列更新を扱った研究として文献 15) がある．配列の分割併合操作と，分割された配列に対する並列操作に基づいている点は，本論文およびその先行研究 16) と同様である．しかし，破壊的更新を実現するのに，データフロー解析に基づいて配列要素の参照が更新に先立つように評価順序を決めている点が，参照数解析を基本的な道具としている本論文の方式と異なる．参照数解析が広義の型解析であることを考えると，本論文の研究の立場はむしろ文献 17) の技法に近い．文献 18) は集合制約に基づく破壊的更新の解析を行っているが，ここでの制約の利用法は，データフロー解析の制約による定式化であって，型やモードの解析おける制約の利用法とは異なっている．

6. 結 論

本研究では，並行論理型言語 KL1 の並列 KLIC 処理系における配列を用いた数値演算の最適化手法を示

した．

静的解析を利用した逐次 KLIC 処理系での最適化技術を拡張し，共有メモリ並列計算機上での KLIC 処理系において，効率の良い並列配列演算の実装が行えた．

我々の並列化方式は，配列の split，join 操作に基づいている．これは，配列に対する並列更新の安全性をプログラムレベルで明示する効果を持っており，また参照数解析という広義の型解析によって単一参照性を保証することができる．並列更新の安全性を明示するコンストラクトを導入する代わりに，並列プロセス間のメモリアクセス競合を静的解析によって保証する方法も提案されているが[8]，我々は，並列性を明示するアプローチにおいて，見通しの良いプログラム記述を行うことを重視している．また，これは高度な最適化を見通し良く行うのに有利であると考えている．

最後に，今後の課題をいくつかあげる．

(1) 手続き間最適化．並行論理型言語の手続きは粒度が小さいため，高度な最適化を図るには手続き間にまたがる最適化が必要である．unbox 化されたままのデータの授受，引数の並びの最適化による無駄なデータ移動の削減などを図ってゆく必要がある．

(2) 配列操作の拡充．配列計算においては，配列の端と内部に対する演算を統一的に記述するために，上下限の少し外側の領域にアクセスできると好都合なことが少なくない．このような領域を guard wrapper[19] という．guard wrapper つきの配列の作成，分割，併合操作は現在設計中である．分散メモリ並列計算機上では，並列処理は配列のコピーをともなうので，guard wrapper つき配列の実現は比較的容易であるが，共有メモリに置かれた guard wrapper つき配列とその並列処理の実装は興味深い課題である．

(3) 添字検査系の実装．配列の添字検査は，Java の効率的な実装などのために研究が加速され，多くの研究（最近のものでは文献 8)，9) など）がある．我々の知る限り，並行論理プログラムのための添字検査系の実装はまだない．手続き型言語のために提案されている添字解析技法のかなりの部分が適用可能と考えるが，単一代入の並行言語であることを考慮した定式化を目下検討している．

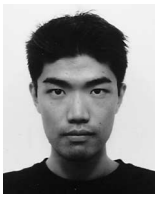
謝辞 本研究の内容に関して議論をしてくださった上田研究室の皆様，とりわけ加藤紀夫氏に感謝いたします．有用なコメントをくださった査読者の方に心より感謝申し上げます．本研究の一部は，文部省科学研究費補助金（C）（2）11680370 の助成を得て行った．

参 考 文 献

1) Ueda, K. and Chikayama, T.: Design of the Kernel Language for the Parallel Inference Machine, *Comput. J.*, Vol.33, No.6, pp.494–500 (1990).
2) Chikayama, T., Fujise, T. and Sekita, D.: A Portable and Efficient Implementation of KL1, *Proc. 6th Int. Symp. on Programming Language Implementation and Logic Programming* (*PLILP'94*), LNCS, Vol.844, pp.25–39, Springer-Verlag, Berlin (1994).
3) Ueda, K. and Tsuchiyama, R.: Optimizing KLIC Generic Objects by Static Analysis, *Proc. 11th Int. Conf. on Applications of Prolog* (*INAP'98*), pp.27–33, Prolog Association of Japan (1998).
4) 土山了士：KLIC 処理系における静的解析を用いた数値演算の最適化，早稲田大学理工学研究科修士論文 (1998).
5) Ueda, K.: Linearity Analysis of Concurrent Logic Programs, *Proc. Int. Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications*, Ito, T. and Yuasa, T. (Eds.), pp.253–270, World Scientific (2000).
6) 上田和紀：klint—KL1 プログラム自動解析系（2.2 版），available from http://www.ueda.info.waseda.ac.jp/software.html (2000).
7) 加藤紀夫，上田和紀：並行論理型言語における同期ポイントの移動の安全性について，情報処理学会論文誌：プログラミング，Vol.41, No.SIG 2 (PRO 6), pp.13–28 (2000).
8) Rugina, R. and Rinard, M.: Symbolic Bounds Analysis of Pointers, Array Indices, and Accessed Memory Regions, *Proc. ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation*, pp.182–195 (2000).
9) Bodik, R., Gupta, R. and Sark, V.: ABCD: Eliminating Array Bounds Checks on Demand, *Proc. ACM SIGPLAN '00 Conf. on Programming Language Design and Implementation*, pp.321–333 (2000).
10) 荒木拓也，坂井修一，田中英彦：Committed-Choice 型言語 Fleng における配列処理の最適化，情報処理学会論文誌，Vol.41, No.4, pp.1146–1161 (2000).
11) Saraswat, V.A., Kahn, K. and Levy, J.: Janus: A Step Towards Distributed Constraint Programming, *Proc. 1990 North American Conf. on Logic Programming*, Debray, S. and Hermenegildo, M. (Eds.), pp.431–446, MIT

Press (1990).
12) Gudjnsson, G. and Winsborough, W.H.: Compile-time Memory Reuse in Logic Programming Languages Through Update in Place, *ACM Trans. Prog. Lang. Syst*, Vol.21, No.3, pp.430–501 (1999).
13) Somogyi, Z., Henderson, F. and Conway, T.: The Execution Algorithm of Mercury: An Efficient Purely Declarative Logic Programming Language, *J. Logic Programming*, Vol.29, No.1-3, pp.17–64 (1996).
14) Henderson, F.: Strong Modes Can Change the World! Technical Report 93/25, Department of Computer Science, University of Melbourne (1993).
15) Sastry, A.V.S. and Clinger, W.: Parallel Destructive Updating in Strict Functional Languages, *Proc. 1994 ACM Conf. on LISP and Functional Programming*, pp.263–272 (1994).
16) Ueda, K.: Moded Flat GHC for Data-Parallel Programming, *Proc. FGCS'94 Workshop on Parallel Logic Programming*, pp.27–35, ICOT, Tokyo (1994).
17) Turner, D.N., Wadler, P. and Mossin, C.: Once Upon a Type, *Proc. 7th Int. Conf. on Functional Programming Languages and Computer Architecture* (*FPCA'95*), pp.1–11, ACM (1995).
18) Wand, M. and Clinger, W.D.: Set Constraints for Destructive Array Update Optimization, *Proc. IEEE Computer Society Int. Conf. on Computer Languages 1998*, pp.184–193 (1998).
19) Hatcher, P.J. and Quinn, M.J.: *Data-Parallel Programming*, The MIT Press, Cambridge, Mass. (1991).

(平成 12 年 7 月 14 日受付)
(平成 13 年 1 月 22 日採録)



坂本 幸司

1976 年生．1999 年早稲田大学理工学部情報学科卒業．現在，同大学院理工学研究科情報科学専攻修士課程在学中．並行論理型言語の最適化技術に関する研究に従事．

松宮 志麻

1977 年生．2000 年早稲田大学理工学部情報学科卒業．現在，同大学院理工学研究科情報科学専攻修士課程在学中．並列処理系，特にメモリ管理システムに興味を持つ．

上田 和紀（正会員）

1956 年生．1978 年東京大学工学部計数工学科卒業．1986 年同大学院情報工学博士課程修了．工学博士．1983 年日本電気（株）入社．1985〜1992 年（財）新世代コンピュータ技術開発機構へ出向．1993 年早稲田大学理工学部情報学科助教授．1997 年同教授．プログラミング言語の設計と実装，並行・並列処理，論理プログラミング，インタラクティブシステムなどの研究に従事．第 7 回日本 IBM 科学賞．日本ソフトウェア科学会，人工知能学会，ACM，IEEE Computer Society，Association for Logic Programming 各会員．Theory and Practice of Logic Programming 言語・システム分野エディタ．