Vol. 41 No. SIG 8(HPS 2) 情報処理学会論文誌：ハイパフォーマンスコンピューティングシステム Nov. 2000

# 分散メモリシステム上での細粒度 non-strict データフロー構造データの生産者・消費者間パイプライン実行

稲永 健太郎[†] 日下部 茂[††] 雨宮 真人[†††]

並列・分散処理における細粒度データフロー計算の有効性を実証するため，我々は non-strict なデータフロー言語およびその処理系の開発を進めている．この種の言語は，明示的な並列・分散処理の実行制御の記述が不要で，抽象度の高いプログラム記述が容易であるという魅力的な特徴を持つ．特に構造データ処理の記述において，この特徴が非常に有効である．しかしながら，既存の分散メモリシステム上でのこの種のプログラム実行において，データフロー計算モデルとノイマン型計算モデルとの違いから，頻繁な細粒度レベルの動的スケジューリングと構造データへの非同期なアクセスのオーバヘッドが生じる．本稿では，この種のオーバヘッドを削減するため，構造データ間の依存関係を解析し，構造データの生産者・消費者を静的に実行順序付け，生産者・消費者実行をパイプライン化する手法を提案する．複数の分散メモリシステム上での評価の結果，本手法が，分散メモリシステム上での細粒度 non-strict データフロー構造データ処理および細粒度マルチスレッド実行のオーバヘッドを削減し，プログラムの実行効率向上に有効であることを示す．

# Producer-consumer Pipelining for Fine-grain Non-strict Structured-data in a Dataflow Language on Distributed Memory Systems

Kentaro Inenaga,[†] Shigeru Kusakabe[††]
and Makoto Amamiya[†††]

Fine-grain non-strict data structures such as I-structure provide high level abstraction to easily write programs with potentially high parallelism due to the eager evaluation (lenient evaluation) of non-strict functions and non-strict structured-data such as an array. Non-strict data structures require frequent dynamic scheduling at a fine-grain level, which offsets the gain of latency hiding. In addition to the dynamic scheduling at a fine-grain level, asynchronous accesses to structured-data using non-strict data structures also cause heavy overhead on distributed memory systems. In order to reduce overhead of fine-grain non-strict structured-data, we propose a compilation technique to analyze dependencies among the structured-data and to schedule producers and consumers of the structured-data. The performance evaluation results indicate that the technique is effective to improve the performance of fine-grain non-strict programs with structured-data on distributed memory systems.

## 1. はじめに

一般に，並列プログラミングではプロセッサ間またはプロセス間同期をはじめとする煩雑な実行制御の記述を必要とする．手続型の言語を用いた場合，煩雑な並列処理制御の記述は，専門的な知識を必要とするため非常に困難である．一方，データフロー計算モデルに基づく言語を用いた場合，データ依存に従って自動的に同期がとられるため，煩雑な並列実行制御を意識することなくデータ依存にのみ着目するだけで，高い並列性を持つプログラムを簡潔に記述することができる．

この種の言語の中でも，non-strict な言語は strict な言語に比べ柔軟性の高いプログラムが記述でき，また，高い並列性の抽出が可能となる．我々は，このような non-strict 言語のプログラムを eager に評価（lenient 評価）し，細粒度マルチスレッド実行方式によって高効率なプログラム実行方式の実現を目指している．

† 九州大学大学院システム情報科学研究科知能システム学専攻
Department of Intelligent Systems, Graduate School of Information Science and Electrical Engineering, Kyushu University
†† 九州大学大学院システム情報科学研究院情報工学部門
Department of Computer Science and Communication Engineering, Graduate School of Information Science and Electrical Engineering, Kyushu University
††† 九州大学大学院システム情報科学研究院知能システム学部門
Department of Intelligent Systems, Graduate School of Information Science and Electrical Engineering, Kyushu University

分散・並列処理における細粒度 non-strict データフロー計算の有用性を実証するため，既存の計算機において non-strict データフロー言語 V [5),8),9),14)~16),18)] の実装を進めている．V 言語処理系は，マルチスレッド実行方式に基づいた細粒度なコード DVMC（Datarol Virtual Machine Code）を生成した後，そのコードから実装対象計算機上での実行可能なコードを生成する．ここで DVMC は，さまざまな種類の計算機への実装のために想定された，ハードウェア構造（ネットワーク構成，メモリ階層，プロセッサ数）を特定しない仮想計算機 DVM（Datarol Virtual Machine）[1)] 用のコードである．

一般に，細粒度なプログラム実行におけるオーバヘッドの主な原因となる実行中断には，スレッドの動的な順序付けにより生じる実行中断と構造データへの非同期なアクセスにより生じる実行中断がある．細粒度マルチスレッド実行の特別な支援環境を持たない既存の計算機では，この種の言語を単純に実装しただけでは，並列実行制御や同期処理をソフトウェアで実現するため，処理系が出力したコードの実行効率は大きく低下してしまう．これまでの細粒度マルチスレッド実行の静的な実行順序付けに関する研究の多くは，スレッドの動的な順序付けにより生じる実行中断の回数を削減することを目的としている．

我々は，これまでの V 言語処理系の研究において，strict 実行に対する効率的並列実行の可能性をを明らかにし [18)]，また，スレッドの動的な順序付けにより生じる実行中断に対する問題の解決 [5)] を行ってきた．本稿では，構造データへの非同期なアクセスにより生じる実行中断の回数を削減することを目的とし，構造データの生産者・消費者間の静的な実行順序付けを行いパイプライン化する手法を提案する．この手法を適用することで，分散メモリシステム上での細粒度 non-strict 構造データ処理に関する細粒度レベルの動的な順序付けおよび非同期アクセスのオーバヘッドを削減することが可能である．この静的な順序付け手法は，前者のスレッドの動的な順序付けにより生じる実行中断の回数を削減する手法と組み合わせて適用でき，より効率的な細粒度実行を可能とする．

ここで，スレッドの動的な順序付けにより生じる実行中断の回数を削減する手法では，スレッドからなる関数のレベルの並列性を用いることとし，関数内のスレッドに共通のコンテクストを持たせ，それら同一関数のスレッドをプログラムの動作を保証しつつできる限り結合し，同期処理の回数を削減している．構造データへの非同期なアクセスにより生じる実行中断の回数を削減する手法でも，同様に，過度な並列性を抑制することで同期処理の回数を削減している．

以下本稿では，2 章で，細粒度 non-strict 分散構造データと分散メモリシステム上でのこの種の構造データの実行モデルを示し，既存の計算機上での細粒度 non-strict 並列構造データの効率的な実装を阻害する問題点を指摘する．3 章で，構造データ間の依存関係を解析し，構造データの生産者・消費者間の実行順序を静的に決定するためのコンパイル手法を提案する．4 章では，構造データの 1 つである配列を用いたプログラムを複数の分散メモリシステム上で実行し，本手法の性能評価を行う．最後に，5 章で，他言語での関連研究との比較を行い本手法の優位性を述べる．

## 2. 細粒度 non-strict 分散構造データ

### 2.1 細粒度 non-strict 構造データ

Non-strict な構造データとは，すべての要素の値が決定する前に，その構造データの要素に対して読み出しアクセスが可能であるという特徴を持つ構造データを指す．この性質を用いることで，要素のアクセス順序を意識することなく，簡潔にプログラム記述できる．この種の構造データの処理を実現するため，基本的に細粒度（要素あるいは要素ブロック）レベルの処理が行われる．このような細粒度 non-strict 構造データの例として配列やリストがあげられるが，本稿では配列を例にとり話を進める．

細粒度 non-strict 構造データの生成では，領域確保と要素値生成の計算が分離される．ここで，**図 1** に細粒度 non-strict 配列生成の計算モデルを示す．

細粒度 non-strict 配列生成は，V 言語プログラム中の `mkarray` 式で表現される．この計算モデルでは，細粒度 non-strict 配列生成のための 2 種類の関数インスタンス（プロセス）が存在する．1 つは，`mkarray` 式を実行する `mkarray` インスタンスであり，もう 1 つは，各要素値を生成する `pf`（`parallel filling`

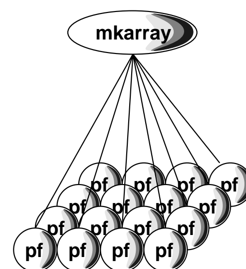


**図 1** 細粒度 non-strict 配列生成の計算モデル
Fig. 1 Computation model of creation of fine-grain non-strict array.

function）インスタンスである．mkarray インスタンスは，配列の領域を確保した後，要素値を生成する pf を並列に生成し，返り値として，生成する配列（へのポインタ）を親インスタンスに返す．一方，pf インスタンスは，mkarray インスタンスにより要素の生産者として並列に生成され，他の pf インスタンスとは非同期に要素値を生成する．

### 2.2 分散メモリシステム上での細粒度 non-strict 構造データ

分散メモリシステム上において，pf インスタンスは，各プロセッサ上に分配される．我々の実装では，基本的に owner computes rule に従い pf インスタンスが分配される．**図 2** に分散メモリシステムにおける細粒度 non-strict 配列生成の計算モデルを示す．

この計算モデルでは，2.1 節で示した 2 種類のインスタンス，mkarray インスタンスおよび pf インスタンスに加え，新たに mksubarray インスタンスが用意されている．mksubarray インスタンスは，部分配列を生成するためのものである．ここで部分配列とは，同一のプロセッサ上に存在する配列要素の集まりを指す．mkarray インスタンスは，mksubarray インスタンスを各プロセッサ上に並列に生成し，mksubarray インスタンスから返り値として各プロセッサ上の部分配列（へのポインタ）を受け取る．mksubarray インスタンスは，各プロセッサ上で pf インスタンスを並列に生成する．mksubarray インスタンスを用いることで，分散メモリシステム上での実装に特有の問題点であるインスタンス生成（ここでは pf インスタンス生成）の遠隔呼び出しの回数を削減し，mkarray インスタンスへのアクセスを分散させる．

### 2.3 細粒度 non-strict 構造データのオーバヘッド

既存の計算機上での細粒度 non-strict 構造データ（以下，本節では単に構造データ）処理におけるオーバヘッドの多くは，非同期なアクセスを実現するための機構を用いることにより生じる．さらに，この非同期なアクセスのために，マルチスレッド実行におけるスレッド間・インスタンス間同期処理を必要とする．並列実行においては，各プロセッサ上で上記の 2 種類の同期処理のオーバヘッドが，各プロセッサ上での実行時間に影響を及ぼし，結論としてプログラム全体の実行時間に影響を及ぼす．そこで本節では，並列実行における各プロセッサ上での構造データに関する non-strict な処理のオーバヘッドを示すために，次に示す 2 つの場合について，単体の計算機上で逐次処理的にプログラム実行した場合の実行時間を測定する．

(1) 静的順序付け
生産者・消費者を静的に順序付ける．すなわち，A および B の生産者は消費者よりも先に実行され，A，B ともに細粒度レベルの同期機構を必要としない．

(2) 動的順序付け
生産者・消費者のスケジュールを動的に行う．すなわち，A および B への生産者・消費者のアクセスは並列に行われ，A および B は，実行時に細粒度レベルの同期機構を必要とする．

ここで，配列 A，B の内積 $\mathtt{A}\cdot\mathtt{B} = \sum_i \mathtt{A}[i] * \mathtt{B}[i]$ を計算するプログラムを，単一プロセッサの Sun ワークステーション SS10（25 MHz，128 MB メモリ）上で実行する．この実験で用いるプログラムは，細粒度マルチスレッド実行のための独自のライブラリと実行時システムを用いて実行される．

V 言語で記述されたこのプログラムを**図 3** に示す．このプログラム中の let 式では，配列 A と B を生成するための 2 つの mkarray 式が存在する．

**表 1** に，前述の 2 つの場合それぞれの実行時間を示す．この表では，プログラム全体の実行時間を，構造データ特有の処理（I-structure を用いた構造データ領域管理および非同期アクセス）の時間と，マルチス

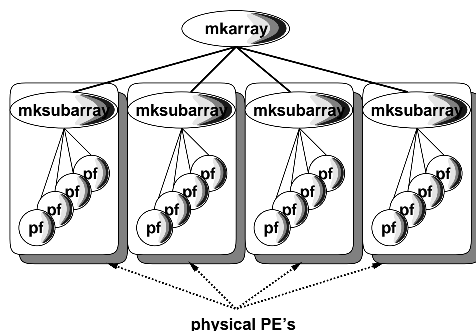


**図 2** 分散メモリシステムにおける細粒度 non-strict 配列生成の計算モデル
Fig. 2 Computation model of creation of fine-grain non-strict array on distributed memory systems.

```
program inner_product;
function main() return (integer)
= { let
     A:array of integer = mkarray i in [1..n] body i,
     B:array of integer = mkarray j in [1..n] body n-j
    in
     foreach k in [1..n] sum A[k]*B[k] };
```

**図 3** V 言語で記述された内積プログラム
Fig. 3 An inner product program written in the V language.

**表 1** 内積プログラムの実行時間（秒）
Table 1 Elapsed time of inner product program (sec.).

| | 細粒度 non-strict 配列特有の処理 | その他の処理 |
|---|---|---|
| 静的順序付け | 0 | 0.12 |
| 動的順序付け | 6.13 | 45.58 |

配列サイズ：64,000

レッド実行のための動的なスケジューリングを含むその他の処理の時間，の 2 つに分けて示す．

表 1 から，構造データ処理およびマルチスレッド実行のオーバヘッドの削減に，構造データあるいは要素レベルの並列性を制御し構造データの生産者・消費者の実行を順序付けする手法が有効であることが確認できる．また，表 1 の細粒度 non-strict 配列特有の処理の時間が静的な順序付けにより 0 となっている．つまり，構造データの生産者・消費者を静的に順序付けすることで，細粒度 non-strict 配列特有の処理が不要となりそのオーバヘッドを削減できることが確認できる．

また，細粒度マルチスレッド実行のオーバヘッドを含むその他の処理の時間について，2 つの場合に大きな差が生じていることが分かる．このことは，ある要素について，生産者である pf インスタンスが，値を書き込む以前に消費者が値の読み出し要求を発した場合，消費者側は値が書き込まれるまでその実行が中断され動的な順序付けを必要とし，構造データ処理が単にそれ自体のオーバヘッドだけでなく，マルチスレッド実行のための動的スケジューリングのオーバヘッドを生じるという副作用を起こしていることを示している．つまり，構造データの生産者・消費者を静的に順序付けすることで，マルチスレッド実行のための動的スケジューリングのオーバヘッドを削減できる．

次章では，構造データの生産者・消費者の実行を静的に順序付けし，非同期なアクセスを同期機構を必要としない同期付けされたアクセスに変換する手法を示す．

## 3. 細粒度 non-strict 構造データの生産者・消費者のパイプライン実行

本章で述べる手法では，処理系が扱うことのできるデータフローグラフで表現可能な中間表現を用い，細粒度 non-strict 構造データの生産者と消費者間の静的な順序付けを行う．ここで用いる中間表現は，DVMC に構造データの依存情報を付加した拡張中間表現である．この拡張中間表現は，他のデータフロー言語の中間表現に類似しており，スレッド結合[5] をはじめとする最適化を施す際にも使われる．この拡張中間表現を用いることで，処理系は本手法のために新たな内部表現を特別に用意することなく，コンパイル時の時間的・空間的コストを軽減することが可能である．

### 3.1 データフローグラフ

本稿で提案する手法で用いる，DVMC の拡張版中間表現であるデータフローグラフ $DG$ を次のように定義する．

**定義 1（データフローグラフ $DG$）** データフローグラフ $DG$ は，頂点と依存アーク $(V, A)$ からなる．ここで，$V$ は頂点の集まりを表し，$A \subseteq V \times V$ は，頂点間の依存関係の集まりを表す．

$V$ の各頂点は，DVMC 命令からなる．命令の例として，算術/論理演算命令，構造データ生成/アクセス命令，メッセージ送/受信命令がある．ここで，データフローグラフ $DG$ 中の non-strict な構造データ X の生成に関する頂点 $v_{\mathtt{X}}$ を定義する．

**定義 2（生成頂点 $v_{\mathtt{X}}$）** ある non-strict 構造データ X について，生成頂点 $v_{\mathtt{X}}$ は X を生成する命令 `mkarray` を持つ．

$DG$ における生成頂点は，2.1 節に述べた計算モデルにおける `mkarray` インスタンスに相当し，また V 言語における `mkarray` 式に相当する．

また，$A$ の各アークは頂点間の依存関係を表し，頂点 $v$ から頂点 $w$ への依存関係を $(v, w) \in A$ と表現する．遅延を持たない依存関係を CDD（Certain Direct Dependency），split-phase な構造データへの読み出し要求/目的の値の受け取りのような遅延を持つ可能性がある依存関係を CID（Certain Indirect Dependency）と呼ぶ．これらの依存関係は，DVMC において明示的に表現されている．$DG$ には，CDD，CID のほかに PID（Potential Indirect Dependency）と呼ばれる依存関係が存在する．PID は，Schauser ら[11] により提案された頂点間の依存関係の 1 つであり，関数呼び出しにおける引数の受け渡しや構造データへの読み書きなどの潜在的かつ間接的な依存関係を表す．実行時のデッドロックを避けるため，PID は CDD や CID とともに循環依存関係を形成することはない．ただ本手法において，処理系はこれら 3 つの依存関係を区別せず同じ $DG$ の依存関係として取り扱うこととする．

### 3.2 Non-strict 構造データ間の依存関係の探索

Non-strict な構造データの生産者・消費者の静的順序付けにおいて，コンパイラはまず構造データ間の依存関係を知る必要がある．$DG$ とその中にある生成頂点を用いて，2 つの non-strict な構造データ間の依存関係を定義する．ここで，Non-strict な構造データ間

(1) $V_{scope} := V$ ; $V_{search} := V_{init}$ ; $L_{search} := \{(x, \emptyset) | x \in V_{search}\}$ ; $L_{result} := \emptyset$ ;
(2) **repeat until** $V_{scope} = \emptyset$
  (a) $V_{creation-in-search} := \{v_{\mathtt{Y}} \mid v_{\mathtt{Y}} \in V_{search}\}$ ;
  (b) **if** $V_{creation-in-search} \neq \emptyset$ **then**
    **forall** $v_{\mathtt{Y}} \in V_{creation-in-search}$
      **if** $(v_{\mathtt{Y}}, L) \in L_{search} \ \wedge \ v_{\mathtt{X}} \in L$ **then**
        $L_{result} := L_{result} \ \cup \ \{\mathtt{X} \rightarrow \mathtt{Y}\}$ ;
  (c) $V_{scope} := V_{scope} - V_{search}$ ;
  (d) $V_{search'} := \{w \mid w \in V_{scope} \wedge v \in V_{search} \wedge (v, w) \in A\}$ ; $L_{search'} := \emptyset$ ;
  (e) **forall** $m \in V_{search}$
    **forall** $n \in V_{search'}$
      **if** $(m, n) \in A$ **then**
        **if** $m \in V_{creation-in-search}$ **then**
          $L_{search'} := L_{search'} \cup \{(n, L_m \cup \{m\})\}$ ;
        **else**
          $L_{search'} := L_{search'} \cup \{(n, L_m)\}$ ;
  (f) $V_{search} := V_{search'}$ ; $L_{search} := L_{search'}$ ;
(3) **return** $L_{result}$

**図 4** Non-strict 構造データ間の依存関係探索アルゴリズム
Fig. 4 Algorithm to search dependencies among non-strict structured-data.

の依存関係は，$DG$ 中では生成頂点間の依存関係として表現される．

**定義 3（Non-strict 構造データ間の依存関係）** $DG$ 中の生成頂点 $v_{\mathtt{X}}$ と $v_{\mathtt{Y}}$ との間に依存関係が存在するとき，non-strict 構造データ間の依存関係 X→Y が存在する．生成頂点間に依存関係が存在しないときは，X と Y 間に依存関係は存在しない（X↛Y と表される）．

この定義は，Y が X の要素値を用いて定義されれば，X→Y という依存関係が存在することを意味している．

上記の定義を用いて，non-strict 構造データ間の依存関係を探索するアルゴリズムを導入する．このアルゴリズムでは，データフローグラフ $DG = (V, A)$ を入力とする．$DG$ 内の MAIN 関数（プログラム実行で一番最初に実行される関数）の初期頂点（関数実行の開始時点で実行可能な頂点）を起点として，生成頂点からの依存関係を $DG$ に付加しながら，non-strict 構造データ間の依存関係を探索する．探索により発見された non-strict 構造データ間の依存関係を出力とする．このアルゴリズムの詳細を**図 4** に示す．

このアルゴリズム中の $V_{init}$ は MAIN 関数の初期頂点の集合を表現する．$L_{search}$ は，検索対象の頂点の集合 $V_{search}$ の要素である頂点 $m$ と，その要素への依存が存在する生成頂点の集合 $L_m$ との組 $(m, L_m)$ からなる集合を表現する．$L_{result}$ は，このアルゴリズムの出力となる生成頂点間の依存関係，すなわち non-strict 構造データ間の依存関係，を要素とする集合である．このアルゴリズムにおいて，探索するデータフローグラフの頂点の数を $N$，生成頂点の数を $c$ とすると，全頂点を 1 度ずつ探索し，すべての生成頂点間に依存関係が存在する場合を考慮しても計算量は $O(N + c^2)$ であり，$c$ がその頂点の性質上 $N$ に比べ非常に小さいことから，高速な依存関係の探索が可能である．

### 3.3 細粒度 non-strict 構造データの生産者・消費者間パイプライン化

3.2 節での non-strict 構造データ間の依存関係を探索した後，さらに要素単位の依存関係を調べ，その結果をもとに構造データの生産者・消費者間の静的な順序付け（パイプライン化）を行う．この手法では，生産者と消費者の間の実行順序を静的に決定することで，過度な並列性を抑制しつつ，生産者同士あるいは消費者同士の並列性は保持される．

(1) 構造データ間依存関係を全順序関係に変換
半順序関係を満たす構造データ間の依存関係を，トポロジカルソートにより全順序関係を満たすよう変換する．ただし，与えられる構造データ間依存関係には，直接的な依存関係だけからは把握できない，複数の構造データが介在する間接的な循環依存関係が存在する場合が考えられる．この種の循環依存関係が存在する場合，その循環依存関係ごとに，属する構造データをひとまとめにし，あたかも 1 つの構造データとして取り扱う．

(2) 生成パターンによる non-strict 構造データの論理的な要素集合への分割
ソースコードあるいは中間表現（データフローグラフ $DG$）内の要素値を生成する pf には，要素値を生成するパターンが記述されている．例として，即値のみを使った計算を行う生成パターンや，他の構造データ要素の値を参照しそ

の値を使った計算を行う生成パターンなどがあげられる．ここで，要素値の生成パターンが静的に決定されることを条件に，要素値の生成パターンごとに構造データを要素の論理的な要素集合☆に分割する．

(3) 論理的な要素集合の分類
前ステップで生成された論理的な要素集合レベルでの生成順序付けのために，論理的な要素集合を次の 2 種類に分類する．
- 始点集合
即値あるいは添字変数のみを用いて要素の値が生成される論理的な要素集合
- 参照集合
始点集合に属さない論理的な要素集合

(4) 論理的な要素集合内での要素間依存関係の探索
構造データの生産者・消費者間の順序付けを行うために必要な，論理的な要素集合内の要素間依存関係の情報として，論理的な要素集合ごとに，要素参照における各次元の参照方向を表す方向ベクトル $\vec{\theta}$ を計算する[16]．方向ベクトル $\vec{\theta}$ は，次元ごとに $>$（添字の昇順に要素生成可能の意），$<$（添字の降順に要素生成可能の意），$=$（参照する論要的な要素集合と同じ順序で要素生成可能の意），$*$（任意の順で要素生成可能の意）のいずれかの値をとる．

(5) 論理的な要素集合内での要素間の生成順序付け
論理的な要素集合ごとに生成された方向ベクトル $\vec{\theta}$ について，そのベクトルの方向が静的に一定である場合に限り，その論理的な要素集合内での要素の生成順序がベクトル方向に従って決定される．ベクトルの方向が静的に一定でない場合，生産者・消費者間の実行順序付けが不可能であり，従来用いてきた動的な要素間の生成順序付けを行うこととする．

(6) 論理的な要素集合間の生成開始タイミングの順序付け
前ステップで静的に一定であると判定された方向ベクトルを持つ分類された論理的な要素集合に対し，構造データ間の依存関係と論理的な要素集合ごとに生成された方向ベクトル $\vec{\theta}$ で表現された要素間の依存関係に従い，論理要素集合間の要素値生成順序付けを行う．まず，論理的な要素集合間の生成順序は，始点集合をはじめとして方向ベクトルに従い決定される．ここで，論理的な要素集合間の依存関係は半順序関係であることから，依存関係が存在しない論理要素集合間の生成開始タイミングの順序付けには，次に示す優先順位を用いる．
(a) 始点要素集合
(b) 直接自己参照する構造データ内の参照集合
(c) 他の構造データを参照する構造データ内の参照集合
本ステップで決定される生成順序は，論理的な要素集合内で最初に生成される要素の生成開始のタイミングを示すものであり，ある論理的な要素集合の全要素が生成された後に，次順序の論理的な要素集合の要素の生成が開始されることを意味するものではない．

(7) 要素の生産者・消費者間の静的順序付け
論理的な要素集合ごとの方向ベクトル $\vec{\theta}$ と論理的な要素集合間の生成順序に従い，要素の生産者・消費者間の実行順序付けを行う．この時点での，実行順序付けされた生産者・消費者の振舞いの例を**図 5** に示す．

(8) 非同期アクセスを同期付けされたアクセスに変換
生産者・消費者間の順序付けがなされた場合，生産者・消費者からの要素アクセスには細粒度な同期機構は不要であるため，この種の同期機構が不要な同期付けされたアクセスに変換する．

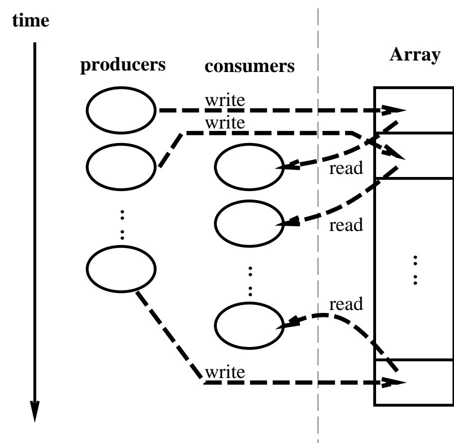


**図 5**　生産者・消費者間のパイプライン実行の振舞い
Fig. 5　Behavior of producer-consumer pipelining.

☆ ここでの論理的な要素集合は，各プロセッサ上に分散配置された物理的な要素集合と区別される．

```
function foo_1(n:integer) return (array of integer)
= {let
    A:array of integer
     = mkarray i with ([1..n]) body if i=1 then i
                                   else B[i-1],
    B:array of integer
     = mkarray j with ([1..n]) body A[j]
   in B };
function foo_2(n:integer; X:array of integer)
                          return (array of integer)
= {let
    Y:array of integer
      = mkarray k with ([1..n]) body X[n-k+1]
   in Y };
function foo_3(n:integer) return (array of integer)
= foo_2(n,foo_1(n));
```

**図 6** V 言語で記述された例題プログラム
Fig. 6 A sample program written in the V language.

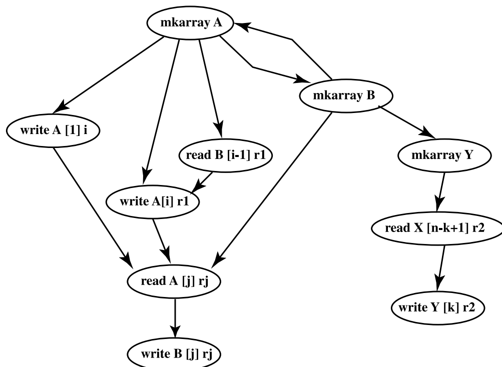


**図 7** 図 6 例題プログラムの $DG$（一部）
Fig. 7 A part of $DG$ of the sample program in Fig. 6.

上記の順序付けにおいて示された条件を満たさない論理要素集合については，属する要素の生産者・消費者間の静的な順序付けは困難であるため，要素レベルの同期機構により動的に実行順序が決定される．

ここで，**図 6** の例題プログラムを用いて本手法の適用例を示す．

図 6 の例題プログラムにおける A，B，X，Y に関するデータフローグラフの一部を**図 7** に示す．この V 言語プログラムには，A，B，Y に関する 3 つの mkarray 式が存在し，それぞれの mkarray 式の body 部に要素値の生成パターン（A には 1 と B[i-1]，B には A[j]，Y には X[n-k+1]）が記述されている．この V 言語プログラムの $DG$ には，プログラム中の mkarray 式に対して mkarray 命令が，body 部の各生成パターンに対して write 命令が，そして構造データ参照式に対して read 命令が存在する．

この図から，A→B と B→A，すなわち A$\xrightarrow{+}$A という循環依存関係と，B→Y，A$\xrightarrow{+}$Y という依存関係が存在する．A→B と B→A に関して，構造データレベルでの循環依存関係が存在するため，グラフ自体の変化はない．また，B→Y に関して，B に関する write 命令の後に，Y に関する read 命令（B の要素を参照）が実行されるよう順序付けすることになり，B に関する write 命令から Y に関する read 命令へのアークが張られる．さらに，B に関する mkarray 命令から C に関する mkarray 命令までのアークが削除される．

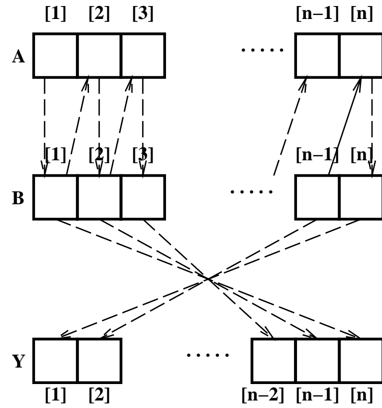


**図 8** 例題プログラム内の non-strict 構造データ要素のアクセス順序
Fig. 8 Access order to non-strict sturctured-data in Fig. 6.

A については，添字変数 i から要素値が生成される論理的な要素集合（以下，集合 A1）と，B[i-1] を参照する論理的な要素集合（以下，集合 A2）との，2 つの論理的な要素集合が存在する．B については，A[j] を参照する論理的な要素集合（以下，集合 B1）がただ 1 つ存在する．Y については，X[n-k+1] を参照する論理的な要素集合（以下集合 Y1）がただ 1 つ存在する．

また，論理的な要素集合ごとの方向ベクトル $\vec{\theta}_{LogicalSetName}$ が，次のように生成される．

$$\vec{\theta}_{A1}=(*), \vec{\theta}_{A2}=(>), \vec{\theta}_{B1}=(=), \vec{\theta}_{Y1}=(<)$$

上記の構造データ間依存関係と論理的な要素集合の性質から，論理的な要素集合間かつ要素間（論理的な要素集合内）の生成開始タイミングの順序が，次のように決定される（括弧内は論理要素集合内での生成順序を示す）．

(1) 集合 A1 の要素
(2) 集合 A2 の要素（添字変数の昇順）
(3) 集合 B1 の要素（添字変数の昇順）
(4) 集合 Y1 の要素（添字変数の降順）

上記の生成順序に従い，処理系は要素の生産者・消費者間の実行順序を決定する．**図 8** に図 6 の例題プログラム内での non-strict 構造データ要素のアクセス順

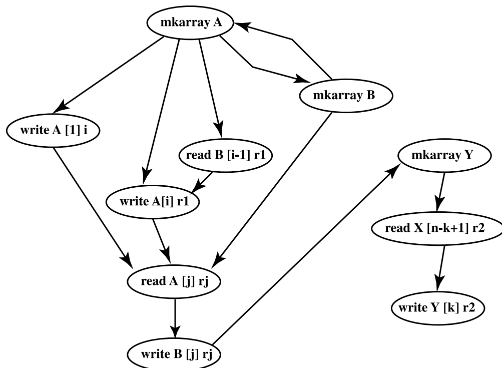


**図 9**　図 7 の *DG* から変換された *DG*（一部）
Fig. 9　A part of *DG* transformed from the *DG* in Fig. 7.

序を示す．また，**図 9** に，本手法により図 7 の *DG* から変換された後の *DG* を示す．

## 4. 評　　価

本章では，行列積，クイックソート，ウェーブフロント法の 3 種類のベンチマークプログラムを用いて，前章で提案した静的な順序付け手法を評価する．本評価では，分散メモリ型並列計算機富士通 AP1000（Sparc 25 MHz，FPU，16 MB メモリ搭載のセルプロセッサ 64 台）と PC クラスタマシン（Intel Celeron 300 MHz，256 MB メモリ搭載のノード PC 16 台を Myrinet[7] 接続）を用いた．この 2 種類の計算機は，ハードウェア，主にネットワーク部分，の異なる分散メモリ計算環境であり，これらの環境下で各プログラムを実行させその速度を評価した．

Non-strict データフロー言語 V による行列積，クイックソート，ウェーブフロント法のプログラム例を付録に示す．これらのプログラムのサイズ（行数）は，それぞれ 8，34，17 である☆．このことは，non-strict データフロー言語を用いることで，プログラマが並列プログラムを簡潔に記述でき高い実行性能を得ることができるという，プログラム記述の優位性を示している．

各プログラムとも高い並列性を内在している一方，それぞれに異なる規則性のアクセスパターンを持つため，生産者・消費者間の順序付けの自由度に違いが見られる．行列積については，構造データ間の循環依存関係を持たず，生産者・消費者間の順序付けの自由度が高い．またクイックソートについては，複数の構造データを介した循環依存関係を持ち，行列積プログラムに比べ生産者・消費者間の順序付けの自由度が低い．ウェーブフロント法については，自構造データ内で要素レベルの循環依存関係を持ち，行列積およびクイックソートプログラムに比べ生産者・消費者間の順序付けの自由度が低い．これらの異なる依存関係の特徴を持つプログラムを用いて，本稿で提案した手法の有効性を確認することを試みる．

**表 2** に，AP1000 上での行列積，クイックソート，ウェーブフロント法の各プログラムの実行時間をを示す．ここで，V 言語プログラムのコンパイル時に，本稿で提案した手法を適用しない場合の AP1000/C コード（a）の実行時間と，手法を適用した場合の AP1000/C コード（b）との実行時間を比較した場合，（a）の実行時間に比べ，（b）の実行時間が大幅に短縮されていることが確認できる．このことは，本手法により細粒度 non-strict 構造データ処理のオーバヘッドおよびマルチスレッド実行のオーバヘッドが削減され，プログラム全体の実行時間の短縮に貢献していることを示している．

次に，PC クラスタ上での行列積，クイックソート，ウェーブフロント法の各プログラムの実行時間を，それぞれ**表 3**，**表 4**，**表 5** に示す．すべてのプログラムにおいて，用いる配列のサイズを変化させて実行時間を測定した．AP1000 の場合と同様，V 言語プログラムのコンパイル時に，本稿で提案した手法を適用しない場合の MPC++[6] コード（a），手法を適用した場合の MPC++コード（b）との実行時間の比較した結果，本稿で提案した手法を適用することで約 4〜20 倍の実行効率の向上が見られる．さらに，コンパイル時に静的順序付け手法が適用されたコード（b）の実行時間と手書きによるコード（c）の実行時間差が，1 桁以内に収まっていることが確認できる．このことは，本手法により細粒度 non-strict 構造データアクセスのオーバヘッドが削減され，さらに静的な順序付けにより頻繁な細粒度の動的順序付けのコストが軽減され，細粒度マルチスレッドを用いたプログラムが，実用的な効率で実行可能であることを示している．

また，本実験で使用した AP1000 のように並列処理専用の計算機だけでなく，PC クラスタのようなプロセッサ処理速度に対するスループットの小さいネットワークを利用した計算機を用いた場合であっても，本稿で提案した手法が，細粒度マルチスレッドを用いたプログラムの実行において有効であると考える．

同じプログラムを AP1000 上で測定した結果と PC クラスタ上で測定した結果を比較した場合，PC クラ

☆ ちなみに，同様のプログラムを AP1000/C で記述したときのコードサイズ（行数）は 401，539，643，また，Myrinet PC クラスタマシン用の MPC++で記述した場合は，251，357，438 であった．

**表 2** AP1000 での静的順序付け手法の未適用/適用/手書きコードの実行時間（秒）
Table 2 Elapsed time of Non-applied/Applied/Hand-Coded Codes on AP1000 (sec.).

| プログラム名 | 行列積 | ウェーブフロント法 | クイックソート |
|---|---|---|---|
| 配列サイズ | 1024×1024 | 512×512 | 64,000 |
| (a) 未適用 | 184 | 32.3 | 19.4 |
| (b) 適用 | 33.0 | 6.92 | 1.82 |
| (c) 手書き | 1.84 | 2.63 | 0.130 |
| (a)/(b) | 5.58 | 4.67 | 10.7 |
| (b)/(c) | 17.9 | 2.64 | 14.1 |

**表 3** 行列積プログラムの実行時間（秒）
Table 3 Elapsed time of matrix multiplication program (sec.).

| 配列サイズ | 512 × 512 | 1024 × 1024 | 2048 × 2048 | 4096 × 4096 |
|---|---|---|---|---|
| (a) 未適用 | 51.2 | 305 | 2106 | 16230 |
| (b) 適用 | 5.10 | 38.7 | 240 | 1558 |
| (c) 手書き | 1.04 | 5.44 | 28.4 | 183 |
| (a)/(b) | 10.0 | 7.88 | 8.78 | 10.4 |
| (b)/(c) | 4.90 | 7.11 | 8.44 | 8.49 |

**表 4** クイックソートプログラムの実行時間（秒）
Table 4 Elapsed time of quicksort program (sec.).

| 配列サイズ | 8192 | 16384 | 32768 | 65536 |
|---|---|---|---|---|
| (a) 未適用 | 22.8 | 40.0 | 74.5 | 99.7 |
| (b) 適用 | 1.19 | 2.45 | 4.21 | 6.02 |
| (c) 手書き | 0.467 | 1.03 | 1.684 | 2.47 |
| (a)/(b) | 19.2 | 16.3 | 17.7 | 16.6 |
| (b)/(c) | 2.55 | 2.39 | 2.50 | 2.44 |

**表 5** ウェーブフロント法プログラムの実行時間（秒）
Table 5 Elapsed time of wavefront-based program (sec.).

| 配列サイズ | 512 × 512 | 1024 × 1024 | 2048 × 2048 | 4096 × 4096 |
|---|---|---|---|---|
| (a) 未適用 | 6.28 | 24.6 | 103 | 385 |
| (b) 適用 | 1.39 | 5.65 | 23.0 | 91.7 |
| (c) 手書き | 1.02 | 4.00 | 15.9 | 61.4 |
| (a)/(b) | 4.52 | 4.35 | 4.49 | 4.20 |
| (b)/(c) | 1.36 | 1.41 | 1.45 | 1.49 |

スタの場合の方が，(b)/(c) の値が小さい．これは，AP1000 と PC クラスタとの間のハードウェアの面での通信メカニズムの違いによると考える．AP1000 は，並列処理用計算機として開発されており，専用ネットワークのスループットの実効値が，PC クラスタに比べ理論値（100 Mbps）の 50%程度となる．これに対し，PC クラスタでは，従来の PC が外部ネットワークで接続されており，PC 間通信において外部バスを経由し，スループットの実効値が，AP1000 に比べ理論値（約 1 Gbps）の 25%程度となる．つまり，AP1000 に比べ PC クラスタの方が，プロセッサ処理速度に対するスループットが相対的に低く，プログラムの実行時間全体に占める通信処理に要する時間の割合が高いことが考えられる．このことは，処理系の出力するコードと手書きのコードとの，通信処理以外の処理の効率化の度合いが，プログラムの実行時間全体に及ぼす影響が小さいことを意味する．

ただ，依然としてコンパイルされたコードと手書きのコードとの間に実行時間の差が存在する．この時間差は，実装における分散構造データの領域管理およびアクセス計算にかかる計算量の違いによるものと考える．V 言語の実行時システムでは，block-cyclic 分散の規則に従い構造データを分散しており，そのような構造データの領域管理に必要な分散情報を各プロセッサが保持し，動的にその情報を利用する．これに対し手書きの場合，プログラマが構造データの分散情報を織り込んだ領域管理，たとえば同一領域の複数回の利用あるいはインクリメントによるアドレス計算の効率

化，の方針を持ち，暗黙的にコード内に記述することで，その分だけ動的に行う処理の回数を少なく抑えることができる．また分散構造データの要素へのアクセスに関して，V 言語プログラム実行では，プロセッサの識別子とそのプロセッサ内での局所アドレスの組が必要であり，それらの値は実行時に決定される．一方，手書きの場合，分散構造データへのアクセスに必要な値の組が明示的に記述されていることにより，我々の処理系が出力するコードに比べ，実行時のアドレス計算の回数を少なく抑えることができる．この分散メモリシステム上での構造データ管理のオーバヘッドに関しては，構造データに必要な領域サイズを静的に計算し，領域を静的に確保する手法と文献 8) による推移的アドレス計算手法を本手法と組み合わせて用いることで，より効率的なプログラム実行が可能になると考える．

## 5. 関連研究

細粒度実行の静的な実行順序付けに関する研究として，さまざまな種類の言語において細粒度マルチスレッド実行のオーバヘッドを削減するための，静的な順序付け手法が提案されている．その例として，関数型言語 Id における Separation Constraint Partitioning[11] と呼ばれるスレッド分割や，並列論理型言語 Fleng[10]，GHC[3]，KL1[13] などの処理系における，スレッドを用いた効率化技術であるゴール融合などがあげられる．我々の関数型言語においても，non-strict な実行を保証しつつ関数内の細粒度のスレッドをできる限り結合していくという効率化手法の研究が行われている[15]．この種の効率化手法は，スレッドの静的な順序付けを行うことで，スレッドの動的な順序付けにより生じる実行中断の回数の削減を実現している．

これらの効率化手法に対し，本稿で提案した静的な順序付け手法は，構造データへの非同期なアクセスにより生じる不要なスレッド実行の中断要因を取り除き，スレッドの集まりで表現される関数レベルの静的な順序付けが可能である．構造データへの非同期なアクセスによるスレッド実行の中断は，動的な順序付けによるスレッド実行の中断に比べ，その中断要因が，構造データへのアクセスという間接的なスレッド間の依存関係によるものであり，一般的にその解決は困難である．本手法では，既存の構造データのアクセス解析手法を用いて間接的なスレッド間の依存関係を解析し，その結果を用いて静的な順序付けを実現している．本手法は，スレッド実行の中断要因の違いから，我々が従来用いてきたスレッド結合と組み合わせて適用することができ，より効率的な実行が可能である．

細粒度 non-strict 構造データ処理のオーバヘッドを削減するための静的手法として，GPH（Glasgow Parallel Haskell）[12] と呼ばれる non-strict な関数型言語では，構造データへの非同期アクセスのオーバヘッドを削減するため，ソースコード内の let 式や for 式などの逐次処理記述部分からその計算順序を残しつつ，並列実行可能な部分を並列実行用の特別な記述に変換する手法を提案している[4]．この変換は，言語仕様において並列実行のための記述が許されているために可能である．一方，V 言語では並列実行制御の記述が不要であるという点が異なり，本稿で提案した手法では，並列性を内在するプログラムから逐次化可能な部分を抽出することが，GPH とは逆の発想に基づいている．また，配列処理の効率化の研究としては，V 言語における配列のコピー除去[14] がある．この研究では，配列更新のオーバヘッドを削減するためにコピー除去を行うが，この手法において配列参照の逐次化を行うという観点から，本稿で提案した静的順序付け手法と類似している．同様の研究として，Fleng[17] におけるコピー除去がある．

また，我々の言語と同じくデータフロー計算モデルに基づいた strict な言語に SISAL[2] がある．この言語は，すべての要素の値が生成された後にアクセスが可能となるような，strict な構造データを前提としている．したがって，本質的に strict な構造データだけを用いる場合には，V 言語と同等の実行効率を得ることが可能である[5]．さらに，我々の言語が持つ non-strict な構造データを用いる場合には，簡潔なプログラム記述ができるとともに，本稿で提案した構造データの要素レベルの静的な実行順序付けにより，効率の良いプログラム実行が可能である．

## 6. 結論

我々は既存の分散メモリ型計算機上での細粒度 non-strict 構造データに関するオーバヘッドを削減するための生産者・消費者間の静的順序付け手法を提案した．この手法では，処理系が non-strict なプログラム実行の正しさを保持しつつ，依存解析により構造データの生産者・消費者を順序付けし，さらに構造データへの非同期なアクセスを同期付けされたアクセスに変換する．分散メモリ型並列計算機 AP1000 および Myrinet 接続の PC クラスタ上での本手法の評価の結果，non-strict 構造データへの非同期なアクセスと頻繁な細粒度の動的スケジューリングによるオーバヘッドが削減され，本手法が分散メモリシステム上で non-strict な

言語のプログラムの実行効率の向上に有効であることが示された．

今後は，分散構造データの効率的な領域管理・アクセスのための手法の開発と，さまざまな機種の計算機を対象としたソフトウェアサポートによる細粒度 non-strict 構造データの効率的な実装を進めていく．

## 参考文献


1) Amamiya, M. and Taniguchi, R.: Datarol: A Massively Parallel Architecture for Functional Language, *Proc. SPDP*, pp.726–735 (1990).
2) Cann, D.C.: Retire Fortran? A Debate Rekindled, *Comm. ACM*, Vol.35, No.8, pp.81–89 (1992).
3) Fuchi, K., Furukawa, K. and Mizoguchi, F.：並列論理型言語 GHC とその応用，共立出版 (1987).
4) Hill, J.M.D., Clarke, K.M. and Bornat, R.: Vectorising a non-strict data-parallel functional language 'Spinless (not so) Tagless G-Machine, *Proc. 5th international workshop on the implementation of functional languages* (1993).
5) Inenaga, K., Kusakabe, S., Morimoto, T. and Amamiya, M.: Hybrid Approach for Non-strict Dataflow Program on Commodity Machine, *International Symposium on High Performance Computing* (*ISHPC*), LNCS, Vol.1336, pp.243–254, Springer-Verlag (1997).
6) Ishikawa, Y., Hori, A., Tezuka, H., Matsuda, M., Konaka, H., Maeda, M., Tomokiyo, T. and Nolte, J.: MPC++, *Parallel Programming Using C++*, Wilson, G.V. and Lu, P. (Eds.), pp.429–464, MIT Press (1996).
7) Boden, J.N., Cohen, D., Felderman, R.E., Kulawik, A.E., Seitz, C.L., Seizovic, J.N. and Su, W.-L.: Myrinet – A Gigabit-per-Second Local-Area Network, *IEEE-Micro*, Vol.15, No.1, pp.29–36 (1995).
8) Kusakabe, S., Nagai, T., Inenaga, K. and Amamiya, M.: Reducing Overhead in Implementing Fine-grain Parallel Data-structures of a Dataflow Language on Off-the-shelf Distributed-memory Parallel Computers, *Proc. HICSS-30* (*30th Hawaii International Conference on System Sciences*), Maui, HI, Vol.1, pp.234–243 (1997).
9) Kusakabe, S., Takahashi, E., Taniguchi, R. and Amamiya, M.: Dataflow-Based Lenient Implementation of a Functional Language, *Valid*, on Conventional Multi-Processors, *Parallel Architectures and Compilation Techinuqes* (*PACT '94*), pp.279–288 (1994).
10) Nilsson, M. and Tanaka, H.: Fleng Prolog – the language which turns supercomputers into Prolog machines, *Logic Programming '86*, LNCS Vol.264, pp.170–179, Springer-Verlag (1989).
11) Schauser, K.E., Culler, D.E. and Goldstein, S.C.: Separation Constraint Partitioning – A New Algorithm for Partitioning Non-strict Programs into Sequential Threads, *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (*POPL*) (1995).
12) Trinder, P.W., Barry Jr., E., Davis, M.K., Hammond, K., Junaidu, S.B., Klusik, U., Loidl, H.-W. and Jones, S.L.P.: GpH: An Architecture-Independent Functional Language, *IEEE Trans. Software Engineering, special issue on Architecture-Independent Languages and Software Tools for Parallel Processing* (1998).
13) Ueda, K. and Chikayama, T.: Design of the kernel language for the parallel interface machine, *The Computer Journal*, Vol.33, No.6, pp.494–500 (1990).
14) 日下部茂，岡崎芳希，谷口倫一郎，雨宮真人：データフロー関数型言語の並列化コンパイラにおける配列の静的コピー除去，並列処理シンポジウム JSPP '95 予稿集，福岡，pp.161–168 (1995).
15) 日下部茂，森本徹夫，雨宮真人：非ストリクトデータフロープログラム実行におけるスタックフレームの利用，情報処理学会研究報告，97-PRO-14, pp.73–78 (1997).
16) 稲永健太郎，日下部茂，雨宮真人：再帰的非ストリクト構造データ生成の効率化，第 55 回情報処理学会全国大会論文集，Vol.1, pp.333–334 (1997).
17) 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).
18) 日下部茂，高橋英一，谷口倫一郎，雨宮真人：データフローモデルに基づく超並列 V 言語とその商用並列計算機上の実装について，情報処理学会論文誌，Vol.36, No.7, pp.1529–1541 (1995).

## 付録 V 言語プログラム

```
program Matrix_Multiplication;
function matmul(n,integer; a,b:array of real)
                                  return (array of real)
= mkarray (i,j) in ([1..n],[1..n]) body
    foreach k in [1..n] sum a[i,k]*b[k,j];
function main(n:integer) return (array of real)
= {let
    a:array of real
      = mkarray (i,j) in ([1..n],[1..n]) body 1.0*(i+j),
    b:array of real
      = mkarray (i,j) in ([1..n],[1..n]) body 1.0*(i-j)
   in matmul(n,a,b) };
```

**図 10** V 言語で記述された行列積プログラム
Fig. 10 A matrix multiplication program written in the V language.

```
program wave;
function wave(A:array of integer; n:integer)
                               return (array of integer)
= { let
     ! = foreach (i,j) in ([1..n],[1..n])
           body if (i = 1) or (j = 1) then
                  update(A,[i,j],rand())
                else
                  ! = update(A,
                             [i,j],
                             A[i-1,j-1]+A[i-1,j]+A[i,j-1])
    in A};
function main(m:integer) return (integer)
= { let
     B:array of integer
       = mkarray (i,j) in ([1..n],[1..n]) body rand(),
     Result:array of integer
       = for (C:array of integer; k:integer) init (B,1)
           if k = m then C
           else recur(wave(C,m),k+1)
    in Result[n,n]};
```

**図 11** V 言語で記述されたウェーブフロント法プログラム
Fig. 11 A wavefront program written in the V language.

```
program Quicksort;
type AR = array of real;
function qsort(A0,A1:AR; left,right:integer) return (AR,AR)
= if left = right then (A0,A1)
  else
   { let
      pivot:real = A0[left],
      next_right:integer = for (l0,l1:integer) init (left,left)
                             if l0 = right then l1
                             else if A0[l0] < pivot then
                               {let
                                  ! = update(A1,[l1],A0[l0])
                                in recur(l0+1,l1+1)}
                             else recur(l0+1,l1),
      next_left:integer = for (r0,r1:integer) init (right,right)
                            if r0 = left then r1
                            else if A0[r0] >= pivot then
                              {let
                                 ! = update(A1,[r1],A0[r0])
                               in recur(r0-1,r1-1)}
                            else recur(r0-1,r1),
   in {let
         ! = qsort(A1,A0,left,next_right),
         ! = qsort(A1,A0,next_left,right)
       in (A1,A0)}
  };
function main(s:integer) return (AR)
= { let
     A:AR = mkarray i in [1..s] body rand(),
     B:AR = mkarray i in [1..s] body 0.0,
     (X,Y:AR) = qsort(A,B,1,s)
    in X };
```

**図 12** V 言語で記述されたクイックソートプログラム
Fig. 12 A quicksort program written in the V language.

(平成 12 年 5 月 7 日受付)
(平成 12 年 8 月 24 日採録)

**稲永健太郎**（学生会員）
1973 年生．1996 年九州大学工学部情報工学科卒業．1998 年同大学大学院システム情報科学研究科知能システム学専攻修士課程修了．現在同専攻博士後期課程在学中．工学修士．並列処理記述言語およびその処理系，データフローモデルに基づく細粒度構造データ処理の研究に従事．

**日下部 茂**（正会員）
1966 年生．1989 年九州大学工学部情報工学科卒業．1991 年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．同年より同専攻助手．1998 年より九州大学大学院システム情報科学研究科情報工学専攻助教授．2000 年マサチューセッツ工科大学計算機科学科客員研究員．関数型言語，細粒度並列言語処理系と実行時システムの研究に従事．工学博士．ACM，IEEE 各会員．

**雨宮 真人**（正会員）
1942 年生．1967 年九州大学工学部電子工学科卒業．1969 年同大学大学院工学研究科修士課程修了．同年日本電信電話公社武蔵野電気通信研究所入所．以来，プログラミング言語・処理系，自然言語理解，データフロー・アーキテクチャ，並列処理，関数型/論理型言語，知能処理アーキテクチャ等の研究に従事．現在九州大学大学院システム情報科学研究院知能システム学部門教授．工学博士．電子情報通信学会，ソフトウェア科学会，人工知能学会，IEEE，ACM，AAAI 各会員．