共有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
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

日下部，茂
九州大学大学院システム情報科学研究院情報工学部門

雨宮，真人
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

https://doi.org/10.15017/1525441

出版情報：九州大学大学院システム情報科学紀要. 7 (1), pp.29-36, 2002-03-26. 九州大学大学院システム情報科学研究院
バージョン：
権利関係：

九州大学大学院
システム情報科学紀要
第7巻 第1号 平成14年3月

Research Reports on Information Science and
Electrical Engineering of Kyushu University
Vol.7, No.1, March 2002

共有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***

Producer-consumer Pipelining for Fine-grain Non-strict Dataflow Structured-data on Shared Memory Systems

Kentaro INENAGA, Shigeru KUSAKABE and Makoto AMAMIYA

(Received December 14, 2001)

Abstract: 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 arrays. Non-strict data structures require frequent dynamic scheduling at fine-grain level, which offsets the gain of latency hiding. Not only the dynamic scheduling but also asynchronous accesses to structured-data using non-strict data structures cause heavy overhead on stock machines. Mutual exclusion of structured-data on shared memory systems also causes overhead.

In order to reduce overhead of fine-grain non-strict structured-data, we propose a compilation technique to analyze dependencies between 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 shared memory systems.

Keywords: Pipelining, Non-strict, Fine-grain, Multi-thread, Dataflow

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，データの依存関係にしたがって自動的に同期が取られるため，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 な言語では，strict な言語に比べ柔軟性の高いプログラムが記述でき，高い並列性の抽出も可能である．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，かつ細粒度マルチスレッド実行方式を用いることで高効率なプログラム実行を目指している．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[7),16),6),5)] (以下，V 言語) 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)]用のコードを指す．

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，non-strict 実行についての効率的並列実行の可能性を明らかにし[16)]，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．

一方，構造データへの非同期なアクセスにより生じる

平成13年12月14日受付
* 知能システム学部門
** 情報工学部門
*** 知能システム学部門

実行中断に関しては，分散メモリシステム上でのその実行中断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すでに提案している[15]．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．

共有メモリシステム上でこの静的な順序付け手法を用いる場合，構造データの排他制御を考慮する必要がある．排他制御には，共有資源へのアクセス順序の問題と，複数の同時アクセスの問題がある．これらの問題を解決するため，これまでにさまざまな解決方法が考えられている．それらは，1) 共通資源に対するアクセスの順序とそのタイミングを静的に決定する方法，2) 共有資源側に排他制御のためのデータ構造を用意し，動的に制御を行う方法の2つに大別できる．本稿では，上記の1)に該当する，共有メモリシステムにおいて適用可能な構造データの生産者・消費者間の静的順序付け手法を提案する．この手法を用いることで，アクセスする生産者・消費者間のスケジューリングにより，生産者の実行が終了後に消費者の実行が可能になることから，アクセスの順序とタイミングが静的に決定され，排他制御用データ構造が不要となる．その結果，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や非同期アクセス，排他制御のオーバヘッドが削減できることを示す．

以下本稿では，2.節で，細粒度 non-strict 構造データの特徴と共有メモリシステム上でのこの種の構造データの計算モデルを示し，共有メモリシステム上での細粒度 non-strict 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.節で，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.節では，構造データの一つである配列を用いたプログラムを Sun Ultra Enterprise 10000 Server (Ultra Sparc-I 250MHz×24CPU, 9GB メモリ,以下 Starfire と呼ぶ) で実行し，本手法の性能評価および考察を行う．最後に，5.節で，他言語での関連研究との比較を行ない本手法の優位性を述べる．

2. 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．この特徴により，要素のアクセス順序を意識することなく，簡潔にプログラムの記述ができる．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．細粒度 non-strict 構造データの例として配列やリストが挙げられるが，本稿では配列を例に取り話を進める．

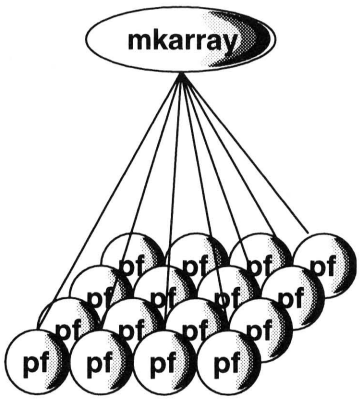

Fig.1 Computation model of creation of fine-grain non-strict array.

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．**Fig.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．ひとつは，`mkarray`式を実行する`mkarray`インスタンスであり，もう一つは，各要素値を生成する`pf` (`parallel filling 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．

2.1 共有メモリシステム上でのオーバヘッド

共有メモリシステムにおける各プロセッサでの細粒度 non-strict 構造データの主な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，配列を用いて2次元行列`A`, `B`の積を計算するプログラムを Starfire 上で実行し，その実行時間を測定する．

- 静的順序付け (static scheduling)
  生産者・消費者を静的に順序付ける．つまり，`A` および `B` の生産者は消費者よりも先に実行され，`A`, `B` ともに要素レベルのデータ同期機構を必要としない．また，排他制御を必要としない．
- 動的順序付け (dynamic scheduling)
  生産者・消費者のスケジュールを動的に行う．つまり，`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，実行時に要素レベルのデータ同期機構を必要とする．加えて，排他制御用のデータ構造を必要とする．

```
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)
  };
```

Fig.2 A matrix multiplication program written in the V language.

Table 1 Elapsed time of matrix multiplication program (sec.).

	fine-grain non-strict array processing	mutual exclusion	otherwise
static scheduling	0	0.59	8.23
dynamic scheduling	6.20	2.31	17.0

array size: 512×512

この実験で用いるプログラムでは，細粒度マルチスレッド実行のための独自のライブラリを用い，Solaris スレッドにより各プロセッサの振る舞いを実現している．

V言語で記述されたこのプログラムを**Fig.2**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，およびプログラムの解である配列を生成するための`mkarray`式が存在する．**Table–1**に，前述の2つの場合の実行時間をそれぞれ示す．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（要素レベルのデータ同期機構 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制御）の時間 (fine-grain non-strict array processing)，構造データおよび細粒度マルチスレッド実行で用いる資源の排他制御にかかる時間 (mutual exclusion)，そしてマルチス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 (otherwise)，の3つに分けて示す．

Table–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．具体的には，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，また構造データアクセスのための排他制御が不要となることを示している．

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わかる．この処理時間の差は，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する．このために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が生じ，加えて，マルチスレッド実行のための排他制御が生じる起因となる．このようなマルチスレッド実行のための動的スケジューリングおよび排他制御のオーバヘッドは，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削減できる．

次節では，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

本節では，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手法について述べる．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

本稿では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は頂点の集合を表し，$A \subseteq V \times V$は，頂点間の依存関係の集合を表す．

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．次に，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_X) 生成頂点 v_X は，*non-strict* 構造データ Xを生成する命令mkarrayを持つデータフローグラフDGの頂点である．

DGにおける生成頂点は，2.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 の他に PID (Potential Indirect Dependency) と呼ばれる依存関係が存在する．PIDは，Schauserら[9]により提案された頂点間の依存関係で，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しない．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_Xとv_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との間に依存関係は存在しない (X↛Yと表す)．

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**Fig.3**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，全頂点を一度ずつ探索し，全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・消費者間のパイプライン化

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，一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挙げられる．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要素集合に分割する．
3. 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，要素集合を次の2種類に分類する．

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) $V_{creation-in-search} := \{v_Y \mid v_Y \in V_{search}\}$;
   (b) **if** $V_{creation-in-search} \neq \emptyset$ **then**
   forall $v_Y \in V_{creation-in-search}$
   if $(v_Y, L) \in L_{search} \wedge v_X \in L$ **then**
   $L_{result} := L_{result} \cup \{X \rightarrow Y\}$;
   (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}

Fig.3　Algorithm to search dependencies between non-strict structured-data.

- 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要素集合
- 参照集合
  始点集合に属さない要素集合

4. 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，要素集合内の要素間依存関係の情報として，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[14]．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）のいずれかの値を取る．
5. 要素集合内での要素間の生成順序付け
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，その要素集合内での要素の生成順序がベクトル方向にしたがって決定される．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. 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要素集合に対し，構造データ間の依存関係と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したがい，要素集合間の要素値生成順序付けを行う．まず，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したがい決定される．ここで，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，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，ある要素集合の全要素が生成された後に，次順序の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
要素集合ごとの方向ベクトル$\vec{\theta}$と要素集合間の生成順序にしたがい，要素の生産者・消費者間の実行順序付けを行なう．この時点での，実行順序付けされた生産者・消費者の振る舞いの例を**Fig.4**に示す．
8. 同期付けアクセスへの変換
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．同期付けされたアクセスについては，排他制御用のデータ構造を用いないようアクセスを変換する．

上記の順序付けにおいて示された条件を満たさない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．

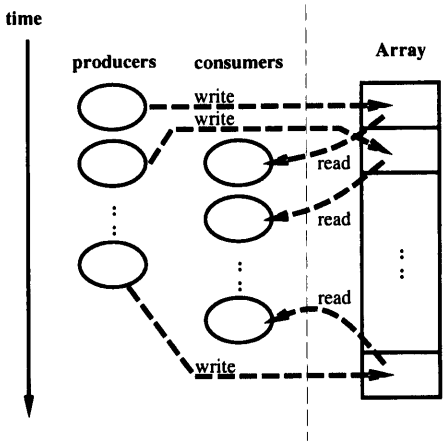

Fig.4　Behavior of producer-consumer pipelining.

4. 評　　価

本節では，行列積，クイックソート，ウェーブフロント法のV言語およびC言語プログラムを用いて，前節で提案した静的な順序付け手法をStarfire（Ultra Sparc-I 250MHz×24CPU, 9GBメモリ）上で評価する．ここで，V言語処理系は，Starfire上で実行可能なC言語プログラムを出力する．

3種類のベンチマークプログラムともに並列性を内在しているが，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定義-参照関係を持たず，生産者-消費者間の順序付けの自由度が3種類のベンチマークプログラムの中で最も高い．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定義-参照関係を持ち，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定義-参照関係を持ち，行列積やクイックソートのプログラムに比べ，生産者-消費者間の順序付けの自由度がより低い．これらの異なる特徴を持つ3種類のベンチマークプログラムを用いて，ノンストリクト構造データの生産者-消費者間の静的順序付け手法の有効性を確かめる．

Table–2に，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C言語コード (a) Non-appliedの実行時間と，手法を適用した場合のC言語コード (b) Applied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構造データ処理や排他制御，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．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) Hand-Codedの実行時間差が，一桁以内に収まっていることが確認できる．このことは，本手法により上記の3種類のオーバヘッドが削減され，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．

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，マルチスレッド実行に不可欠である，実行時システムの初期化や排他制御，Solarisスレッド処理，および独自ライブラリのオーバヘッドによるものと考える．

これらのオーバヘッドを削減するには，従来我々が開発を進めてきたスレッドレベルの効率化に加え，よりメタなレベルである関数レベルの効率化，例えば関数同士の結合，が必要である．この種の効率化では，本稿で用いたデータフローグラフDGの再構築あるいはDGの仕様の変更が不可欠である．その例として，逐次的実行を考慮したループや複数の構造データ要素への一括アクセスの導入，が考えられる．データフローグラフDGの再構築に関しては，データフロー関数型言語Idの共有メモリシステムへの実装において類似の研究が行われている[10]．この実装では，コンパイル時にループ化などのデータフローグラフ（DAG）の変換により，逐次的な実行を利用し実行効率の向上を図っている．このコンパイル時におけるデータフローグラフの再構築の考え方は，我々の研究対象であるnon-strictデータフロー言語に有効であると考える．

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．

スレッドの動的な順序付けにより生じる実行中断を対象とした研究では，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．例としては，関数型言語Idにおける Separation Constraint Partitioning[9]と呼ばれるスレッド分割や，並列論理型言語Fleng[8]，GHC[3]，KL1[12]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挙げられる．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を提案している[13]．

一方，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

Table 2 Elapsed time of Non-applied/Applied/Hand-Coded Codes on Starfire (sec.).

program name	matrix multiplication		quicksort		wave-front	
array size	512×512	1024×1024	4096	8192	256×256	512×512
(a) Non-applied	25.5	147	17.2	49.6	6.15	20.0
(b) Applied	8.83	38.9	3.97	9.01	1.20	3.33
(c) Hand-Coded	1.74	8.40	1.28	2.62	0.62	2.04
(a) / (b)	2.89	3.78	4.33	5.50	5.13	6.01
(b) / (c)	5.07	4.63	3.10	3.44	1.94	1.63

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る．本稿で提案した手法では，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レベルおよびスレッドの集合で表現される関数レベルの依存関係を解析する．その結果を用いて静的な関数レベルの生産者・消費者間の順序付けを行い，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くことができる．また，本手法では，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．

他の言語における細粒度 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，non-strict な関数型言語 GpH (Glasgow parallel Haskell) [11]において，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，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，non-strict な構造データを用いる V 言語の場合には，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

本論文では，共有メモリシステム上での細粒度 non-strict 構造データの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．Sun Ultra Enterprise 10000 Server (Starfire) 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や排他制御，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，本手法が共有メモリシステム上で non-strict な言語プログラムの実行効率の向上に有効であることを示した．

今後は，データフローグラフの再構築を含むコンパイル手法の開発と，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) M. Amamiya and R. Taniguchi. Datarol: A Massively Parallel Architecture for Functional Language. In *Proc. of SPDP*, pp. 726–735, 1990.
2) D. C. Cann. Retire Fortran? A Debate Rekindled. *Communications of the ACM*, Vol.35, No. 8, pp. 81–89, 1992.
3) K. Fuchi, K. Furukawa, and F. Mizoguchi. 並列論理型言語 GHC とその応用. 共立出版, 1987.
4) J. M. D. Hill, K. M. Clarke, and R. Bornat. Vectorising a non-strict functional language for a data-parallel 'Spineless (not so) Tagless G-Machine'. In *5th International Workshop on the Implementation of Functional Languages*, 1993.
5) K. Inenaga, S. Kusakabe, T. Morimoto, and M. Amamiya. Hybrid Approach for Non-strict Dataflow Program on Commodity Machine. In *International Symposium on High Performance Computing (ISHPC), LNCS, Springer-Verlag*, Vol. 1336, pp. 243–254, 1997.
6) S. Kusakabe, T. Nagai, K. Inenaga, and M. Amamiya. Reducing Overhead in Implementing Fine-grain Parallel Data-structures of a Dataflow Language on Off-the-shelf Distributed-memory Parallel Computers. In *Proc.*

of HICSS-30 (30th Hawaii International Conference on System Sciences), Vol. 1, pp. 234–243, 1997.

7) S. Kusakabe, E. Takahashi, R. Taniguchi, and M. Amamiya. Dataflow-Based Lenient Implementation of a Functional Language, *Valid*, on Conventional Multi-Processors. In *Parallel Architectures and Compilation Techniques (PACT'94)*, pp. 279–288, 1994.
8) M. Nilsson and H Tanaka. Fleng Prolog - the language which turns supercomputers into Prolog machines. In *Logic Programming '86, LNCS Vol. 264, Springer-Verlag*, pp. 170–179, 1986.
9) K. E. Schauser, D. E. Culler, and S. C. Goldstein. Separation Constraint Partitioning – A New Algorithm for Partitioning Non-strict Programs into Sequential Threads. In *Conference Record of POPL '95: 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pp. 259–271, 1995.
10) A. Shaw. *Compiling for Parallel Multithread Computation on Symmetric Multiprocessors.* PhD thesis, Massachusetts Institute of Technology, 1998.
11) P. W. Trinder, E. Barry Jr., M. Kei Davis, K. Hammond, S. B. Junaidu, U. Klusik, H. Loidl, and S. L. Peyton Jones. GpH: An Architecture-Independent Functional Language. *IEEE Transactions on Software Engineering, special issue on Architecture-Independent Languages and Software Tools for Parallel Processing*, 1998.
12) K. Ueda and T. Chikayama. Design of the kernel language for the parallel interface machine. *The Computer Journal*, Vol. 33, No. 6, pp. 494–500, 1990.
13) 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.
14) 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講演論文集, 第 1 巻, pp. 333–334, 1997.
15) 稲永健太郎, 日下部茂, 雨宮真人. 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. 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, Vol. 41, No. SIG 8 (HPS 2), pp. 73–84, 2000.
16) 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, 第 36 巻, pp. 1529–1541, 1995.