データ並列言語における擬似ベクトル処理のための実行方式

渡邊誠也[†,☆] 横山亮[††] 湯淺太一[††]

データ並列言語で用いられるデータの数は，通常，利用可能なプロセッサ数を大きく上回るため，個々のプロセッサに対して複数のデータが割り当てられ，繰り返しによりSIMD実行が実現される．この繰り返し実行は，プログラムのコンテキスト制御を行なうために，単純な繰り返しではなく，これまでにいくつかの実行方式が提案されている．一方，スカラプロセッサにおいて，ループ長の長い繰り返し実行の際に生じるキャッシュミスによる性能低下を防ぐ機構に，擬似ベクトル処理機構がある．本論文では，データ並列言語の実行方式について考察し，擬似ベクトル処理に適する実行方式を提案する．また，擬似ベクトル化の成功率を高めるループ分割方法を提案する．提案する手法をデータ並列C言語NCXの処理系に実装し，並列計算機SR2201を用いて性能評価を行なった．その結果，本手法によりデータ並列言語で擬似ベクトル処理が利用可能となり，プログラムの実行速度を向上できることを確認した．

An Execution Method of Data-parallel Languages for Pseudo Vector Processing

NOBUYA WATANABE,[†,☆] RYO YOKOYAMA[††] and TAIICHI YUASA[††]

Since the number of data used in data-parallel programs is generally larger than the number of available processors, multiple data are allocated to each processor and the SIMD execution is realized by iteration. This iterative execution is not obvious because it requires context control of the program, and some execution control methods have been proposed. On the other hand, *pseudo vector processing* is a mechanism which prevents performance degradation caused by cache fault during execution of a long iteration on scalar processors. In this paper, we analyze execution methods of data-parallel programs and propose a method that is appropriate for pseudo vector processing. Furthermore, we present a loop partition technique which enables to promote pseudo vectorization. We implemented the proposed techniques on our language system for the data-parallel C language NCX and evaluated the performance by using a parallel computer Hitachi SR2201. The result of evaluation shows that the proposed techniques make it possible to use pseudo vector processing on data-parallel languages and to increase the performance of programs.

1. はじめに

計算機のプロセッサの演算性能は，半導体技術の進歩により飛躍的に向上してきた．スカラプロセッサでは，プロセッサ内部の処理速度と比較してメモリアクセスの速度が遅く，このメモリアクセスレイテンシが性能に与える影響が大きい．このレイテンシを隠蔽する方法として，キャッシュメモリを用いた記憶の階層化が用いられてきた．キャッシュメモリは年々大容量化してきており，キャッシュミスによる性能低下も少なくなってきている．それでも，キャッシュメモリにデータが収まらないほどの大規模計算になると，キャッシュミスが頻発し，十分な実効性能が得られなくなってしまう．このような問題を改善するために提案されているプロセッサに擬似ベクトルプロセッサPVP-SW(Pseudo Vector Processor based on Slide-Windowed Registers)[4]がある．これは，既存のプロセッサに対して，大量のレジスタ，データのプリロード/ポストストア命令，レジスタウィンドウをソフトウェアで制御するための命令を追加したプロセッサである．演算で用いるデータのロードを演算に対して十分先行して開始し，メモリ－レジスタ間のデータ転送と，プロセッサ内部での演算を並列に行なうことで，メモリアクセスレイテンシの隠蔽を行な

† 豊橋技術科学大学情報工学系
Department of Information and Computer Sciences, Toyohashi University of Technology
☆ 現在，岡山大学工学部情報工学科
Department of Information Technology, Faculty of Engineering, Okayama University
†† 京都大学大学院情報学研究科通信情報システム専攻
Department of Communications and Computer Engineering, Graduate School of Informatics, Kyoto University

う．この手法は，擬似ベクトル処理と呼ばれる．

一方，科学技術計算といった大規模な計算処理には，従来ベクトルプロセッサが用いられてきた．近年，商用の並列計算機が登場してきており，この種の計算に並列計算機が用いられるようになってきた．並列処理を実現するための様々な計算モデルが提唱されているが，その1つにデータ並列モデルがある．多くの実用並列プログラムがデータ並列によって記述できることから，実用化が大いに期待されている．

データ並列計算のためのプログラミング言語は，様々な並列計算機アーキテクチャに対して実装されてきたが，それらの中には，一部のアーキテクチャに特化した記述方法を採用しているものがあり[8)14)]，応用プログラムの移植が困難であるなど，多くの問題を引き起こしている．近年では，ターゲットの並列計算機アーキテクチャに依存しない記述が可能な言語が提案されてきた[1)15)]．

これらのデータ並列言語では，実プロセッサ数を意識しないプログラミングが可能である．プログラムで使われる膨大な数のデータは，各プロセッサのメモリに分散して割り当てられる．各プロセッサでは，割り当てられたデータのサイズに一致する反復回数の繰り返しによりプログラムの各ステップが実行される．この繰り返し実行は，プログラムのコンテキスト制御を必要とするために，単純なループでは実現されない．これまでに，いくつかの実行方式が提案されている[5)]．これらの方式は擬似ベクトル処理に対応しておらず，擬似ベクトル処理を有効に利用することはできない．

本論文では，データ並列言語の実行方式について考察し，擬似ベクトル処理に適する実行方式を提案する．また，擬似ベクトル処理を有効に利用するためのループ分割手法を提案する．以下，まず第2節で，擬似ベクトル処理の概要について説明する．第3節では，これまでに提案されているデータ並列言語の実行方式について説明し，擬似ベクトル処理に適さないことを示す．第4節にて，擬似ベクトル処理に対応する実行方式について述べる．第5節では，擬似ベクトル化の成功率を高めるためのループの分割手法について述べる．第6節で，提案する手法の性能評価を行なった結果を示し，結果に対する考察を行なう．最後に，第7節でまとめる．

2. 擬似ベクトル処理の概要

2.1 擬似ベクトル処理機構

一般に，多くの科学技術計算のプログラムでは

(1) 演算に用いるオペランドデータのメモリからレジスタへのロード
(2) 演算
(3) 演算結果のメモリへのストア

の処理が，ループ文により繰り返し実行される．

ベクトルプロセッサでは，パイプライン化された主記憶アクセスにより，演算に用いる大量のデータを多数のベクトルレジスタにパイプライン的に供給することができ，主記憶アクセスレイテンシが性能に与える影響は少ない．一方，スカラプロセッサでは，プロセッサ内部の処理はパイプライン化されているのにもかかわらず，一般に，主記憶との間のデータ転送はパイプライン化されていない．そのため，プロセッサ内部での処理速度と比較してデータ供給能力が低く，主記憶アクセスレイテンシが性能に与える影響が大きい．多くのスカラプロセッサでは，このレイテンシを隠蔽するためにキャッシュメモリによる記憶の階層化が行なわれているが，膨大な数のデータを扱う大規模計算においては，データ領域が非常に大きく，データの時間的局所性が少ないという性質からキャッシュミスが頻発し，キャッシュを有効に利用できない．

この問題を解決する方法として，スカラプロセッサに大量のレジスタを追加し，演算とレジスタ－主記憶間のデータ転送を並列に行なうプロセッサアーキテクチャが提案されている[4)10)]．ベクトルプロセッサにおける処理と同様な処理をスカラプロセッサで行ない，主記憶アクセスのレイテンシを隠蔽するハードウェアが，**擬似ベクトル処理機構**である．

2.2 擬似ベクトル化

擬似ベクトルプロセッサをターゲットとするコンパイラは，擬似ベクトル化可能なループ文に対して，次のようなコード生成を行なう．

まずループ本体の演算に用いるデータの初期プリロードを行なう．演算に十分先行してデータのプリロード命令を発行することにより，メモリアクセスレイテンシを隠蔽する．

ループ本体の命令の実行において，レジスタには，先行して発行されているプリロード命令によってデータが到着しているため，データ待ちのストールを生じさせることなく演算命令を実行できる．その後，以前に発行された演算命令の結果を，メモリへ格納するポストストア命令を実行する．さらに，後の反復で使われるデータのプリロードを行なう．

各反復の最後で，レジスタのウィンドウをスライドさせることによって，レジスタリネーミングを行なう．これにより，同じ論理レジスタ番号でも反復毎に異なる物理レジスタを使うことができる．

このようなプリロード命令，ポストストア命令，レジ

スタウィンドウスライド命令を使用したループのオブジェクト生成のことを，擬似ベクトル化と呼ぶ．

2.3 擬似ベクトル化の条件

本稿ではターゲットマシンとして，擬似ベクトルプロセッサを要素プロセッサとする分散メモリ型並列計算機 Hitachi SR2201[2] を用いる．SR2201 の C コンパイラは，以下の条件をすべて満たすループを擬似ベクトル化の対象とし，自動的に擬似ベクトル化を行なう[3]．

条件 1 最内側の for 文，while 文．すなわち，本体に他の for 文，while 文を含まない for 文，while 文．

条件 2 ループ外飛び出し文，I/O，関数呼出しを含まない．ただし，数学関数 `exp`, `log`, `log10`, `sin`, `asin`, `cos`, `acos`, `tan`, `atan` の呼出しは擬似ベクトル化の対象とする．

条件 3 浮動小数点配列の演算を含む．

条件 4 for 文に関しては，ループの終値，増分値がループ内で不変であり，かつループの増分値が 0 でない．

条件 5 do-while 文を含まない．

条件 6 コンパイラが解析不可能な添字式（例えば，配列アクセスを含む式，ループ制御変数以外の変数を含む式）を持つ配列アクセスを含まない．

条件 7 ループ繰り返し回数が十分大きい．

3. データ並列言語の実行方式

本節では，これまでに提案されているデータ並列言語の実行制御方式の概要を簡単に説明し，そのままでは擬似ベクトル処理に対応できないことを示す．

3.1 並列実行におけるコンテキスト

まず，以降の議論を分かりやすくするために，仮想プロセッサ (virtual processor, 以下 VP と略す) の概念を導入する．プログラムで用いられているデータ集合の要素数と同数の VP からなる VP 集合を想定し，各 VP に 1 つずつデータ集合の要素を配置する．演算は，基本的に，データ集合と対応付けされた全 VP が同期的に各自のデータをオペランドとして実行する．例えば，変数 `x`, `y`, `z` が各 VP 上に割り当てられているとき，並列実行のコンテキストにおける次の代入文

```
x = y * z;
```

は，各 VP 上の変数 `y` と変数 `z` の値に対する演算，変数 `x` への結果の代入が，各 VP で並列実行されることを意味する．

データ集合の一部に対して演算を行なう場合，VP 集合の部分集合を選択して実行を行なう．データ並列言語では，データ集合の一部に対してのみ演算を行なうための構文が用意されている．例えば，C*[14]，Dataparallel-C[1] の where 文，MPL[8]，NCX[15] の if 文がそれにあたる．これらの構文により，プログラムのコンテキスト制御が行なわれる．このプログラムのコンテキスト，すなわち各 VP の並列実行への参加状況は，**アクティビティ** (activity) と呼ばれる．実行中のある時点で，並列実行に参加している VP をアクティブ (active) な VP，参加していない VP をインアクティブ (inactive) な VP と呼ぶ．

3.2 VP 実行の実現

データ並列プログラムで使用している VP 数に一致する数の実プロセッサを用意することは，通常，困難である．一般的に，1 台のプロセッサに複数の VP を対応付けて，各プロセッサは，割り当てられた複数の VP の処理を担当する．担当 VP 集合上に配置されているデータは，実プロセッサ上のメモリに配列データとして割り当て，ループによる繰り返し実行により，並列実行をエミュレートする．このループは，**VP エミュレーションループ**と呼ばれる．このエミュレーションループは，その時点でアクティブな VP についてのみ処理を実行するようにコード化される必要がある．

以下では，従来までに提案されている VP エミュレーション方式を実現するプログラムコードを示し，各方式を説明する．なお，プログラムは C 言語で記述し，プロセッサに割り当てられた VP の数を n で表記する．各 VP に対して 0 から $n-1$ の番号 (VP 番号) を付加することで，プロセッサに割り当てられた個々の VP を識別する．以降，各 VP を $v_0, v_1, \ldots, v_{n-1}$ と表記する．

3.3 アクティビティベクタによる方式

この方式では，VP のアクティビティを論理値のベクタ（アクティビティベクタと呼ぶ）で表現する．アクティビティベクタを長さ n の配列変数 `a` とすると，`a[`i`]` が 1 のとき v_i はアクティブ，0 のときインアクティブであることを表す．エミュレーションループは，図 1 に示すコードで実現される．ループの各反復でアクティビティベクタの要素の値を調べ，その結果に基づいて演算を行なうかどうかを決定する．E(`i`, S)で，入力プログラムの文 S をエミュレートするコードを表記する．例えば，入力が 3.1 節で述べた代入文の場合，具体的には次のコードとなる．

```
x[i] = y[i] * z[i];
```

この方式では，エミュレーションループの本体にアクティビティベクタの値をチェックするための if 文を含むために，擬似ベクトル化は不可能である．

3.4 VP リストによる方式

この方式では，実行に参加する VP をリストの形で

```
for (i = 0; i < n; i++)
    if (a[i])
        E(i, S)
```

図1 アクティビティベクタ方式のエミュレーションループ
Fig. 1 VP emulation loop using activity vector

```
for (i = head; i != END; i = vplist[i])
    E(i, S)
```

図2 VPリスト方式のエミュレーションループ
Fig. 2 VP emulation loop using VP list

管理する[5]．このリストは，VPリストと呼ばれる．エミュレーションループは，図2に示すコードで実現される．図2で，変数headはVPリストの先頭に対応するVP番号を，配列変数vplistはVP番号を値とし，vplist[i]でv_iの次のVP番号を保持する．VPリストの終端は，VP番号の値域外の値ENDとなっているものとする．

この方式では，ループ制御変数がインデックスを扱っているために，擬似ベクトル化の条件4を満たしておらず，擬似ベクトル化は不可能である．

4. 擬似ベクトル処理のための実行方式

本節では，従来の実行方式における問題解決方法について考察し，擬似ベクトル処理に対応する実行方式について述べる．

4.1 基本方針

アクティビティベクタ方式を採用している処理系の中には，すべてのVPがアクティブとなることをコンパイル時に解析できた場合，アクティビティベクタの値をチェックするコードを出力しないものがある．この場合，擬似ベクトル化を行なうことが可能である．しかし，この最適化は入力プログラムに大きく依存するため，広範囲のプログラムに対しては効果が期待できない．

また，並列計算機を構成する要素プロセッサがスカラプロセッサの場合，アクティビティベクタ方式と比較して，使用メモリ量，実行速度の点でVPリスト方式の方が優れているという報告[5]がある．したがって，本稿ではVPリスト方式をベースに，擬似ベクトル処理に対応する実行方式を考案する．

4.2 提案するVPエミュレーション方式

VPリスト方式では，VPリストの終端に達したかどうかによりループ実行の終了判定が行なわれる．つまり，ループの各反復でVPリストをたどり，リストの終端に達した時点でエミュレーションループの実行を終了する．そのために，エミュレーションループのコードを実行する直前において，アクティブなVP数，すなわち，ループの繰り返し回数は確定されていない．ループの繰り返し回数が確定されていないために，擬似ベクトル化を行なうことができない．エミュレーションループを実行する直前で，アクティブなVP数を確定できれば，擬似ベクトル化を行なえると考えられる．ここでは，このアクティブなVP数が確定できることを仮定して議論を進める．

```
for (j = 0, i = head;
     j < m; j++, i = vplist[i])
    E(i, S)
```

図3 擬似ベクトル処理に対応したエミュレーションループ
Fig. 3 VP emulation loop adapted to pseudo vector processing

VPリスト方式では，条件4が擬似ベクトル化を妨げていた．この条件4を満たすように，新たなループ制御変数を導入する．図3に，擬似ベクトル処理に対応したエミュレーションループのコードを示す．図3で，変数jが新たに導入したループ制御変数であり，変数mは，エミュレーションループを実行する直前における，アクティブなVPの数を保持する変数である．

4.3 エミュレーションループの繰り返し回数の求め方

ここでは，図3のエミュレーションループにおける繰り返し回数（すなわち，変数mの値）をどのように決定するのかについて述べる．

VPのアクティビティは，3.1節で述べた選択実行を行なう構文(if文や，where文など)により変更される．例えば，次のif文が与えられたとする．

if (C) S_1 else S_2

ここでは，if文を実行する全VPで常に，条件式Cの評価結果が同一とならないものとする．SIMD意味論では，上述のif文は次の3つのステップで実行される．

(1) if文を実行した全VPが，一斉に条件式Cを評価する．
(2) 条件式Cの評価結果が真となったVPがS_1を実行する．
(3) 条件式Cの評価結果が偽となったVPがS_2を実行する．

ステップ(1)の際に，VP集合の分割操作が行なわれる．このVP集合の分割の過程において，VPリストの走査が行なわれる．この際に，それぞれのVP集合の要素数をカウントすることにより，分割後の2つのVP集合の要素数を決定することができる．

ステップ(3)の後に，分割されたVP集合の併合操作が行なわれる．VP集合を併合する際も，併合する2つのVP集合の要素数を足し合わせることにより，併合後のVP集合の要素数を求めることができる．

以上のように，VP集合の分割の際に要素数をカウントし，併合操作の際に要素数を足し合わせることで常に，VP集合の数を保持することができる．VP集合を実現するデータ構造の中に，データとして“現在のVP集合の要素数”を保持することで，エミュレーションループの繰り返し回数を与えることが可能となる．

5. ループの分割手法

第4節で，擬似ベクトル処理に対応するエミュレーションループについて述べたが，単にエミュレーションループのコードの置き換えだけでは，擬似ベクトル化に失敗する場合がある．本節では，擬似ベクトル化の成功率を高めるためのループ分割手法について述べる．

5.1 同期ポイント

多くのデータ並列言語で採用しているSIMD意味論においては，演算毎に各VPで同期的実行が行なわれる．VP間で同期をとる箇所を，同期ポイントと呼ぶ．例えば，隣接する2つのVPの変数xの値の平均を求める次のプログラムが与えられたとする．

```
x = (x_{i+1} + x_{i-1}) / 2.0;
```

ここで，x_{i-1}，x_{i+1}は，隣接VPの変数xの参照を意味する．各VPにおける変数xへの代入は，すべてのVPが右辺値の計算を終了した後に行なう必要がある．つまり，右辺値の計算と変数xへの代入との間において，各VP間での同期を必要とする．

同期ポイントは，エミュレーションループの分割点に相当する．先ほどの例の場合，右辺値の計算結果を格納する一時変数tempを導入し，次のように2つのエミュレーションループによりVP実行が実現される．

```
VP_EMULATION_LOOP(i)
    temp[i] = (x[i-1] + x[i+1]) / 2.0;
VP_EMULATION_LOOP(i)
    x[i] = temp[i];
```

以降，エミュレーションループを実現するコードの詳細を省略し，上記のように記述する．

5.2 これまでのループ分割方法における問題点

入力プログラムの意味を保存したVPエミュレーションという観点から言えば，演算の各ステップに同期ポイントを設定すれば良い．しかし，この方法では，エミュレーションループの数が多くなり，ループ実行のオーバヘッドが増し効率的でない．ターゲットマシンがスカラプロセッサの場合，同期ポイントの設定は必要最小限とし，エミュレーションループの数をできるだけ少なくするコードが望ましい．しかし，擬似ベクトル処理への対応を考慮する場合，次の問題が生じる．

```
VP_EMULATION_LOOP(i) {
    S0;
    for (j = 0; j < n; j++) {
        S1;
    }
    S2;
}
```

図4 ループを含むエミュレーションループ
Fig. 4 VP emulation containing a loop

1つ目の問題は，エミュレーションループの本体にループ文が含まれる場合である（図4参照）．この場合，擬似ベクトル化の条件1より，最も内側のループが擬似ベクトル化の対象となり，エミュレーションループ自体が擬似ベクトル化の対象とならない．このエミュレーションループ本体に，擬似ベクトル処理の対象となるコードが含まれている場合，擬似ベクトル化率の低下につながる．

2つ目の問題は，エミュレーションループ本体に擬似ベクトル化を妨げるコード（すなわち，擬似ベクトル化の条件2で示したコード）と，擬似ベクトル処理の対象となるコードの双方が含まれている場合である．前者のコードのために，後者のコードの擬似ベクトル化が行なえなくなる．

以上の問題を解決するエミュレーションループの分割手法を次節以降で述べる．

5.3 ループ分割の基本方針

ループ分割の基本方針は，エミュレーションループ本体の擬似ベクトル化を妨げるコードと，擬似ベクトル処理の対象となるコードを別々のエミュレーションループとして構成することである．しかし，エミュレーションループを細かく分割しすぎると，逆にエミュレーションループ実行にともなうオーバヘッドが増して，実行効率の低下を招くことになる．したがって，擬似ベクトル処理の対象となるコードと擬似ベクトル化を妨げるコードを，同一のエミュレーションループ本体に含めないという制約の中で，エミュレーションループの数を最小とすることを目標に，分割を行なう．

5.4 エミュレーションループの分割方法

分割の対象となるエミュレーションループは，ループ本体に擬似ベクトル化の対象となる浮動小数点変数の参照を含むループである．そのようなコードを含まないループに対しては，分割操作を行なう必要はない．

まず，エミュレーションループの本体に現れるコードを次のように分類する．

PV 擬似ベクトル化の対象となる浮動小数点変数の参照を含むコード

NPV 擬似ベクトル化を妨げるコード

U 上記以外のコード

上記の PV, NPV, U をコードの種類と呼ぶ．

連続するコード列 $C_1, C_2, \cdots, C_n$ が与えられたときに，エミュレーションループの分割点，すなわち，同期ポイントを設定する箇所は表1により決定される．現在のコード列 $C_1, \cdots, C_i$ の状態を S，そのコード列の次に現れるコード C_{i+1} の種類を K とする．表1は，S と K により決定される次の状態を表している．K により枠で囲われた状態に遷移する時，K との間に同期ポイントを設定する．図5に状態遷移図を示す．図5の太い点線で示す矢印（PV から NPV，あるいは NPV から PV への矢印）を遷移するコードとの間に同期ポイントを設定する．

例えば，入力コード列の種類が，

PV,U,PV,NPV,U,PV

の場合，3番目までのコードの状態は PV であり，4番目のコードの種類が NPV であるために，3番目と4番目のコードの間に同期ポイントが設定される．また，5番目までのコードの状態は NPV で，6番目のコードの種類が PV であるため，5番目と6番目のコードの間に同期ポイントが設定される．したがって，次のように3つのエミュレーションループに分割される．

[PV,U,PV] [NPV,U] [PV]

ここでは，'[···]' で1つのエミュレーションループを表している．エミュレーションループの分割を行なわない場合，4番目のコードにより擬似ベクトル化を行なうことが不可能であるが，ループ分割により最初と最後のループが擬似ベクトル化の対象となる．

表1 状態遷移表
Table 1 State table

現在の状態 S	次のコードの種類 K PV	NPV	U
PV	PV	[NPV]	PV
NPV	[PV]	NPV	NPV
U	PV	NPV	U

6. 性能評価

本節では，提案する手法の有効性を示すための性能評価について述べる．提案する手法は，データ並列C言語 NCX の次の2つの処理系に実装した．

- NCX プログラムを C 言語プログラムに変換する EWS 版コンパイラ[12]．

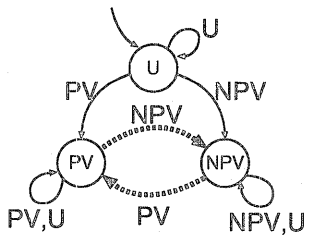


図5 状態遷移図
Fig. 5 State diagram

- MPI[9] ライブラリ呼出しを含む SPMD の C 言語プログラムに変換する MPI 版コンパイラ[6]

実験は，まず，各実行方式の基本性能を測定するプログラムを作成し，エミュレーションループ実行に要する時間の測定を行なった．さらに，実装した処理系を用いていくつかのベンチマークプログラムをコンパイルし，プログラムの実行時間を測定した．プログラム実行環境には，擬似ベクトルプロセッサを要素プロセッサに持つ分散メモリ型 MIMD 並列計算機 Hitachi SR2201 を用いた．SR2201 のプロセッサ性能を，表2に示す．

表2 SR2201 のプロセッサ性能
Table 2 Processor performance of SR2201

理論性能		300MFLOPS
レジスタ数		128
キャッシュ容量	1次	16KB(命令),16KB(データ)
	2次	512KB(命令), 512KB(データ)
メモリスループット		1.2GB/s

6.1 各実行方式の基本性能

エミュレーションループのループ長を変化させて各実行方式の実行速度を測定した結果を，図6に示す．測定した実行方式は，アクティビティベクタ方式 (AV)，VP リスト方式 (VPL)，提案する方式 (VPL-PVP) である．また，比較のためにアクティビティチェックを行なわない自明なループ文の実行速度も測定した．LOOP/PVP は，擬似ベクトル化を行なったコード，LOOP/noPVP は，擬似ベクトル化を行なわなかったコードをそれぞれ表している．実行速度は，1秒当たりの実行ループ回数 (単位は，10^6 ループ / 秒) で表している．ここでの測定結果には，アクティビティベクタ，および，VP リストのデータの初期化に要する時間は含まれていない．なお，ループ本体で行なう計算コードは，次の浮動小数点の乗算を用いた．

```
x[i] = y[i] * z[i];
```

図6より，提案する方式は，アクティビティベクタ方式，VP リスト方式に比べ最大で 2.8 倍高速であること

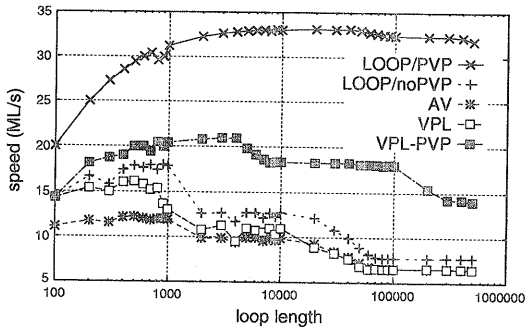


図6 実行方式による基本性能
Fig. 6 Performance of each execution method

が確認できる．一次キャッシュに，データが収まりきる範囲(ループ長1000以下)においては，速度向上は20%～60%程度である．一方，データ数がキャッシュ容量を越えると，擬似ベクトル処理に対応していない方式では性能が急激に低下するが，それに比べると提案する方式の性能低下は少ない．

提案方式でも，VPリストは固定小数点配列で保持されているために，擬似ベクトル処理の対象とはならない．VPリストデータの参照は，キャッシュを介したアクセスであり，リストデータが大きい場合，キャッシュミスが発生する．これが提案する方式の性能低下の原因と言える．

アクティビティチェックを行なわない自明なループと比較すると，提案方式は，ループ長の大きい時に擬似ベクトル化されたループの44%程度の速度となっている．この速度差は，アクティビティ制御のためのオーバヘッドに起因する．この結果より，すべてのVPがアクティブであるとコンパイル時に解析できた場合，アクティビティのチェックを行なわない自明なループのコードを生成することで，擬似ベクトル処理の効果を最大限に発揮できると言える．

6.2 各実行方式の実効性能

実装したEWS版NCXコンパイラを用いて，次のベンチマークプログラムをコンパイルして，その実行時間の測定を行なった．

pi 区分求積法による円周率の近似値計算
jacobi ヤコビ法による2次元平面の温度分布計算
shallow 有限差分法によるshallow-water方程式の解法

実験結果を**表3**に示す．表3より，実際のプログラムにおいても，提案する方式により速度向上がはかられていることが確認できる．6.1節で示した程度の速度向上がはかられていない原因は，実際のプログラムにおいては，選択実行といったVPのアクティビティ制御の処理によるオーバヘッドが加わるためである．VPのアクティビティを変更する処理では固定小数点データを扱っているために，擬似ベクトル処理の対象となっていない．

6.3 並列実行時の性能

6.2節で用いたベンチマークプログラム**jacobi**(使用VP数: 512×512)を並列実行した際の，各方式の実行速度と速度向上を**表4**に示す．提案方式(VPL-PVP)は，そのベースとなっているVPリスト方式(VPL)に対してあらゆるプロセス数において24%～37%の速度向上を確認できる．

一方，プロセス数が1の時，擬似ベクトル化されていないアクティビティベクタ方式(AV)よりも実行速度が遅くなっていることが分かる．この理由は，次のように考えられる．512×512のVPのアクティビティを保持するために，VPリスト方式では，$4(\mathrm{B/VP}) \times 512^2 = 1(\mathrm{MB})$のメモリを必要とするが，アクティビティベクタ方式では，$1(\mathrm{B/VP}) \times 512^2 = 256(\mathrm{KB})$のメモリで済み，これは2次キャッシュに格納可能な容量である．また，アクティビティベクタへのアクセスは連続しているために，キャッシュのヒット率がVPリスト方式と比較して高く，メモリアクセスレイテンシによる性能低下がVPリスト方式に比べ低いと考えられる．

使用プロセス数を2倍にした時，速度向上が2倍以上になっている場合がある．これは，同じ問題サイズのプログラムで使用プロセス数を増加させると，1つのプロセスが担当するVP数が減少し，キャッシュのヒット率が向上することが原因であると考えられる．

6.4 考　　察

今回のターゲットマシンのプロセッサは，浮動小数点演算のみを擬似ベクトル処理の対象とする．そのため，VPリストのデータアクセスに関しては，従来のプロセッサと同様，キャッシュを介したアクセスとなる．擬似ベクトル化できない従来までの方式と比較すると，浮動小数点配列へのアクセスがキャッシュを介されない分，キャッシュを有効に利用できるため性能向上がはかられている．一方，提案する方式においても，ループ長が長くなりVPリストデータがキャッシュの容量を越えると，性能低下を招いてしまうことが分かった．実験結果において，擬似ベクトル化されていないアクティビティベクタ方式との性能差が小さい場合があるが，これは前節で考察したように，アクティビティを保持するために要するメモリ容量の違いに起因することと考えられる．また，実験に用いたベンチマークプログラムは，比較的，VPのアクティビティに変動がなく，また，インアクティブとなるVPの数もそれほど多くないために，

表3 各実行方式の実行時間
Table 3 Execution time of each method

プログラム (使用 VP 数)	実行時間 (sec) AV	VPL	VPL-PVP	速度向上 対 AV	対 VPL
pi (1000000)	0.31	0.22	0.14	55%	36%
jacobi (100×100)	1.68	1.24	1.10	35%	11%
jacobi (200×200)	7.44	5.56	4.82	35%	13%
shallow (64×64)	15.11	16.90	13.28	12%	21%
shallow (128×128)	59.89	69.11	57.76	4%	16%

AV: アクティビティベクタ方式 VPL: VP リスト方式
VPL-PVP: 擬似ベクトル処理対応の VP リスト方式

表4 並列実行時の性能
Table 4 Performance of parallel execution

プロセス数	実行時間 (sec) AV	VPL	VPL-PVP	速度向上 対 AV	対 VPL
1	111.535	158.917	119.869	-7%	25%
2	49.318	55.804	42.469	14%	24%
4	18.295	18.408	12.420	32%	33%
8	9.108	7.790	4.927	46%	37%
16	4.515	3.910	2.616	42%	33%
32	2.468	2.105	1.513	39%	28%

AV: アクティビティベクタ方式 VPL: VP リスト方式
VPL-PVP: 擬似ベクトル処理対応の VP リスト方式

文献 5) に示されている VP リスト方式の優位性が現れてこないことも考えられる.

提案する方式では, 固定小数点データを扱うプログラムに対する性能向上ははかれない. これは, ターゲットマシンのプロセッサアーキテクチャの制約に起因する問題である.

浮動小数点の配列演算のみならず, 固定小数点配列の演算をも擬似ベクトル処理の対象とするプロセッサアーキテクチャ PVP-SWSW (PVP-SW with Slide-Windowed General Registers)[11] が提案されているが, まだ実現されてはいない. このアーキテクチャに基づくプロセッサを用いた場合, 上記の問題を解決できると考えられる. また, 提案する実行方式における性能低下を回避可能であると予想される. さらに, 選択実行において VP アクティビティが変更される際の処理, つまり, VP リストの分割処理も擬似ベクトル化することで, さらなる性能向上が期待できる.

6.5 ベクトルプロセッサへの適用について

本稿で提案した VP エミュレーション手法をそのままベクトルプロセッサに適用した場合, ベクトル化は不可能であると考えられる. これは, 各反復で次にエミュレートする VP 番号を VP リストのアクセスで取得するコード (図 3 における `i = vplist[i]` の部分) が, ベクトル処理に向かないためである. ベクトルプロセッサに対しては, ループキャリー依存を有する VP リスト方式ではなく, アクティビティベクタ方式を用いる必要がある. アクティビティベクタ方式の場合, ベクトルプロセッサのベクトルマスク制御機能を用いることによりベクトル実行が可能である[7)13)].

一方, エミュレーションループの分割に関しては, 本稿で提案する分割方式を用いることで, ベクトル化率を向上させることができると考えられる.

7. おわりに

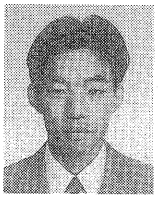
本稿では, データ並列言語の実行方式について考察し, 擬似ベクトル処理に適した実行方式を提案した. アクティブな VP をリストで保持する VP リスト方式のエミュレーションループに, 反復回数を特定するためのループ制御変数を 1 つ追加することで, 擬似ベクトル化を行なうことができることを示した. また, 擬似ベクトル化の成功率を高めるためのループ分割手法を提案した. さらに, 提案した手法をデータ並列 C 言語 NCX の言語処理系に実装した.

実装した処理系を用いて, いくつかのベンチマークプログラムを擬似ベクトルプロセッサ上で実行させ性能評価を行ない, 提案した手法の有用性を示した. 性能評価の結果より, 単一プロセッサを用いた場合で, 従来の手法に比べ最大で 55% の速度向上を確認できた. また, 並列実行を行った場合でも, 最大 46% の速度向上を確認できた.

参考文献

1) Hatcher, P. J. and Quinn, M. J.: Data-Parallel Programming on MIMD Computers, The MIT Press (1991).
2) 日立製作所: SR2201 ハードウェアの紹介, 東京大学計算機センターニュース, Vol. 29, No. 3, pp. 83–97 (1997).
3) 日立製作所: SR2201 各種言語コンパイラーの紹介, 東京大学計算機センターニュース, Vol. 29, No. 3, pp. 123–130 (1997).
4) 位守弘充, 中村宏, 朴泰祐, 中澤喜三郎: スライドウィンドウ方式による擬似ベクトルプロセッサ, 情報処理学会論文誌, Vol. 34, No. 12, pp. 2612–2623 (1993).
5) 貴島寿郎, 湯淺太一: VP リストを用いたデータ並列言語のアクティビティ制御, 電子情報通信学会論文誌, Vol. J80–D–I, No. 12, pp. 954–962 (1997).
6) 河内隆仁, 渡邊誠也, 小宮常康, 湯淺太一: MPI を用いたデータ並列 C 言語 NCX の実装, 情報処理学会 第 55 回 全国大会講演論文集 (1), pp. 337–338 (1997).
7) 幕田好久: データ並列言語のベクトルプロセッサ上での実現と実問題での性能評価, 修士論文, 豊橋技術科学大学 (1998).
8) MasPar Computer Corp.: MasPar Parallel Application Language (MPL) Reference Manual, Ver. 2.0 (1991).
9) Message Passing Interface Forum: MPI: A Message-Passing Interface Standard (1995).
10) Nakamura, H., Imori, H., Nakazawa, K., Boku, T., Nakata, I., Yamashita, Y., Wada, H. and Inagami, Y.: A Scalar Architecture for Pseudo Vector Processing based on Slide-Windowed Registers, *Proc. Int'l Conf. on Supercomputing*, pp. 298–307 (1993).
11) Nakamura, H., Wakabayashi, T., Nakazawa, K., Boku, T., Wada, H. and Inagami, Y.: Pseudo Vector Processor for High-Speed List Vector Computation with Hiding Memory Access Latency, *Proc. TENCON '94*, pp. 338–342 (1994).
12) 岡田徹也, 湯淺太一: データ並列 C 言語 NCX の EWS 用処理系の実装, 情報処理学会 システムソフトウェアとオペレーティング・システム研究会報告 96–OS–72–1 (1996).
13) 高橋大介: 超並列 C 言語 NCX のベクトル化コンパイラの実現, 修士論文, 豊橋技術科学大学 (1995).
14) Thinking Machines Corporation: *C* Programming Guide* (1993).
15) 湯淺太一, 貴島寿郎, 小西浩: データ並列計算のための拡張 C 言語 NCX, 電子情報通信学会論文誌, Vol. J78–D–I, No. 2, pp. 200–209 (1995).

(平成10年6月 4 日受付)
(平成10年9月14日採録)



渡邊 誠也 (正会員)

1970 年生. 1991 年鹿児島工業高等専門学校情報工学科卒業. 1993 年豊橋技術科学大学工学部情報工学課程卒業. 1995 年同大学院工学研究科情報工学専攻修士課程修了. 1998 年同大学院工学研究科電子・情報工学専攻博士後期課程単位取得退学. 現在, 岡山大学工学部情報工学科助手. 並列計算とプログラミング言語に興味を持ち, 超並列計算用プログラミング言語に関する研究に従事.

横山 亮

1973 年生. 1998 年京都大学工学部情報工学科卒業. 現在, 同大学院情報学研究科通信情報システム専攻修士課程に在学中. 並列処理に興味を持っている. ACM 会員.

湯淺 太一 (正会員)

1952 年神戸生. 1977 年京都大学理学部卒業. 1982 年同大学理学研究科博士課程修了. 同年京都大学数理解析研究所助手. 1987 年豊橋技術科学大学講師. 1988 年同大学助教授, 1995 年同大学教授, 1996 年京都大学大学院工学研究科教授, 1998 年同大学院情報学研究科教授となり現在に至る. 理学博士. 記号処理と超並列計算に興味を持っている. 著書「Common Lisp 入門 (共著)」, 「Scheme 入門」, 「C 言語によるプログラミング入門」他. 情報処理学会, ソフトウェア科学会, 電子情報通信学会, IEEE, ACM 各会員.