プログラミング 10－5
（1996. 11. 1）

# 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

貴島 寿郎
kijima@tutics.tut.ac.jp
豊橋技術科学大学 情報工学系
〒441 豊橋市天伯町雲雀ヶ丘 1-1

湯淺 太一
yuasa@kuis.kyoto-u.ac.jp
京都大学大学院 工学研究科 情報工学専攻
〒606-01 京都市左京区吉田本町

データ並列言語の多くは，SIMD 的に動作する多数の仮想プロセッサ（VP）を並列実行の主体と考えることができる．一般に，プログラムで使用する VP 数は，並列計算機の実プロセッサ数よりはるかに多いために，各実プロセッサが複数の VP の実行をエミュレートする．その際に，担当する VP のうち，アクティブなもののみを選択して処理を行なう必要がある．VP のアクティビティは条件分岐や繰り返しなどの制御構文によって動的に変化する．従来の実現では，アクティブな VP 集合をマスクベクタの形で表現していた．しかしこの方式では，アクティビティが頻繁に変化する場合，実行効率やメモリ効率の点で問題がある．本稿ではこれらの問題を解決するため，アクティブな VP 集合をリスト構造で表現する方式を提案し，その有用性を示す．

# Activity Control using VP Lists for Data-parallel Languages

Toshiro KIJIMA* and Taiichi YUASA**
* kijima@tutics.tut.ac.jp
Department of Information and Computer Sciences,
Toyohashi University of Technology, Toyohashi 441, Japan
** yuasa@kuis.kyoto-u.ac.jp
Department of Information Science,
Faculty of Engineering, Kyoto University, Kyoto 606-01, Japan

In data-parallel languages, programs are regarded as being executed by a number of virtual processors (VPs) that run in a SIMD fasion. On actual parallel machines, the number of VPs used in a program is in general much larger than the number of physical processors. Therefore, each physical processor emulates the behavior of multiple VPs. Among such VPs, each physical processor performs operations on only active VPs. Control constructs, such as conditional branches and iterations, may change VP activities dynamically. In conventional implementations, the set of active VPs is represented by a mask vector, but this method is extremely inefficient and requires a large amount of memory space. To solve these problems, we propose a new method for activity control, in which the set of active VPs is represented as a linked list. We show that our method actually increases the performance of data-parallel programs.

# 1 はじめに

データ並列モデルに基づく超並列プログラミング言語の多くは実行制御の意味論としてSIMD (Single-Instruction, Multiple-Data) モデルを採用している．その代表例としてはMPL[1], 並C[2], Dataparallel-C[3], C*[5], NCX[7]などがある．これらの言語では，同数の要素を持つデータ集合間での，対応する要素同士の演算が並列処理の基本となる．またデータ集合の各要素の配置を明示的に指定するための手段が提供されている．

並列実行の主体として仮想プロセッサ (virtual processor, 以下VPと略記) の概念を導入すると，データの配置と並列実行の意味づけが容易になる．データ集合の要素数と同数のVP集合を想定し，各VPには1つずつ要素を配置する．複数のデータ集合については，対応する要素を同一のVPに配置する．データ集合間の演算は全VPが各自の担当する要素の演算を同期的に実行する．

このようなプログラムを実際の並列計算機上で実行するためには，言語処理系は各VPをそれぞれ実際のプロセッサ (physical processor, 以下PPと略記) に対応づける必要がある．しかし多くの場合VP数はPP数を大きく上回る．したがって一般的に各PPは複数のVPの処理を担当し，それらの並列実行を逐次的にエミュレートすることになる．基本的には並列演算の各ステップを逐次ループに変換することで容易にエミュレートできる．このループをエミュレーションループと呼ぶ．

エミュレーションの際には各VPのアクティビティ (activity) も考慮する必要がある．アクティビティとはある時点での各VPの実行参加状態を意味する．また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．

通常，アクティビティはプログラムの実行中に動的に変化する．例えば並列条件文では，全VPで条件式を評価した後，結果が真であったVPのみがアクティブとなり，then節の実行に参加する．大抵の場合，ある並列演算ステップでどのVPがアクティブであるかは実行するまでわからない．したがって各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する必要がある．

先に挙げたデータ並列言語に関しては，エミュレーション手法がいくつか報告されている[6][3][4][9]．これらの手法におけるアクティビティ制御方式はいずれもベクトル計算機で用いられているマスクベクタと同様の手法を継承している．すなわちアクティビティは論理値を要素とするベクタの形で表現され，ベクタの各要素は1つのVPのアクティビティを表現している．

しかしこのアクティビティベクタ方式には次の2つの問題がある．

1. アクティビティ制御のための処理時間が各PPに割り当てられたVP数に比例して増加する．
2. アクティビティの保存に必要な領域の大きさは割り当てVP数と選択的実行のネストの深さに比例する．

本論文ではアクティビティベクタを用いないアクティビティ管理方式を提案する．提案する方式では実行に参加するVPをリストの形で管理する．この方式ではアクティビティ制御のための処理時間がアクティブなVP数に比例する．したがって選択的並列実行の多いプログラムにおいても実行効率を低下させないという特長を持っている．またアクティビティの管理に必要な記憶領域は割り当てVP数のみに比例し，選択的実行のネストの深さには依らない．

# 2 従来のアクティビティ制御

## 2.1 アクティビティベクタ

いま並列実行に参加するVP数を$n$とする．初期状態で$n$個全てのVPがアクティブであるものとする．それらのVPをPP内で識別するために各VPに0から$n-1$の番号 (VP番号) を割り振る．以下，個々のVPを$v_0, v_1, \ldots, v_{n-1}$と表記する．個々のVPの現在のアクティビティを表すために，$\{0,1\}$を要素とする長さ$n$の配列を用いる．この配列をアクティビティベクタと呼ぶ．$a$をアクティビティベクタとすると，$a[i]$が1のとき$v_i$はアクティブ，0のときはインアクティブであることを表す．図1に$n=10$の場合のアクティビティベクタの例を示す．初期状態では全VPがアクティブであるので，$a$の全要素は1に初期化されている．以降，$a$で表わされるアクティブなVP集合を単に$a$と表記する．

## 2.2 エミュレーションループ

アクティビティベクタ方式におけるエミュレーションループは，全VPについての繰り返しとなる．各反復では最初にそのVPがアクティブかどうかをチェッ

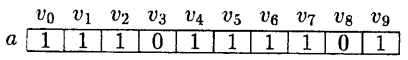


図 1: アクティビティベクタの例

クする．例えば文 $S$ の並列実行をエミュレートするコードは，以下のようになる．

```
for (i = 0; i < n; i = i + 1)
  if (a[i] == 1)
    E(v_i, S)
```

ここで $E(v,S)$ は各 $v \in a$ による $S$ の実行をエミュレートするコードを表す．上記のコードで注意すべき点は，ループの反復回数が常に全 VP 数 $n$ となることである．$i$ の更新や $a[i]$ の判定などのループの制御に関する処理時間はアクティブな VP の数に関係ない．このことは，アクティブな VP が少なくなると，全実行時間中でループの処理時間の占める割合が増加することを意味する．

## 2.3 アクティビティ制御

アクティビティベクタ方式において VP の選択的並列実行を行なう場合，アクティビティベクタに対して分割と併合の操作が必要となる．例えば if 文

```
if ( C ) S1 else S2
```

をエミュレートするためのコードは次のようになる．

1. アクティビティベクタ $a_1$ および $a_2$ を生成する．
2. 条件式 $C$ を評価し，$S_1$ 実行時のアクティビティを $a_1$ に，$S_2$ 実行時のアクティビティを $a_2$ にセットする．

```
for (i = 0; i < n; i = i + 1)
  if (a[i] == 0)
    a1[i] = a2[i] = 0;
  else if (E(v_i, C)) {
    a1[i] = 1; a2[i] = 0;
  } else {
    a1[i] = 0; a2[i] = 1;
  }
```

3. $a_1$ について $S_1$ を実行する．
4. $a_2$ について $S_2$ を実行する．
5. $a_1$ と $a_2$ を併合する．

```
for (i = 0; i < n; i = i + 1)
  a[i] = a1[i] || a2[i];
```

6. $a_1$ および $a_2$ の領域を解放する．

図 2にアクティビティベクタの分割例を示す．

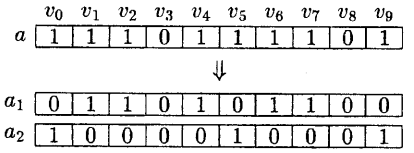


図 2: アクティビティベクタの分割例

基本的なエミュレーションループと同様，アクティビティベクタの分裂および併合に要する時間も $n$ に比例する．このように，アクティビティベクタ方式ではアクティビティ制御に関する処理時間が全 VP 数 $n$ に比例する．したがってアクティブな VP が少なくなると，アクティビティ制御の処理時間が相対的に増加し，実行効率が低下する．

また $S_1$ あるいは $S_2$ の中に if 文が現れた場合，その部分でさらにアクティビティベクタの分割が行なわれる．分割の際には，元のベクタと同じ大きさの領域を新たに必要とするため，メモリ使用量は $n$ と if 文のネストの深さに比例する．

# 3 VP リストを用いたアクティビティ制御

本節ではアクティビティベクタ方式における問題点をふまえ，より適切なアクティビティ制御方式を提案する．

まずエミュレーションループについて考える．本来エミュレーションループで行なうべきことは，アクティブな各 VP について順次処理を行なうことである．したがってある VP についてのエミュレーションの後，次のアクティブな VP が直ちに特定できることが望ましい．しかしアクティビティベクタ方式の場合，次のアクティブな VP を特定するためにはベクタの走査が必要である．これは実行効率の面で不利である．そこで，アクティブな VP のみの集合を VP の列として扱うことができるデータ構造が望ましい．

アクティビティ制御における基本操作は VP 集合の分割と併合である．したがって VP 集合の表現は分割および併合が高速に行なえるデータ構造である

ことが望ましい．これらの要件を満たすデータ構造としてはリスト構造があげられる．そこで今回リスト構造を採用したアクティビティの表現とその制御方式を提案する．これをVPリスト方式と呼ぶことにする．

## 3.1 データ構造

アクティビティをVPリスト方式で表現すると図3のようになる．VP集合は，VPを要素とする1本のリストで表現される．テーブル$L$はVP間のリンクを表現する．$L$の各要素$L[i]$はリスト上で$v_i$の次に位置するVPの番号を格納している．リストの終端となる$L$の要素にはVPの番号の範囲外の値$\perp$が格納される．初期状態では全VPが1本のリストを形成する．

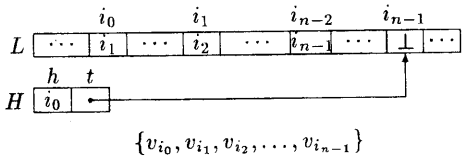


図 3: VP リスト

リストの先頭はテーブル$L$の外に用意されたヘッダ$H$で示される．ヘッダは$h$部と$t$部からなる．$h$部はリストの先頭にあるVPの番号を保持する．$t$部はリストの末尾となっている$L$の要素を指すポインタである．

特別な場合として，リストが空である場合，ヘッダの$h$部には$\perp$が格納される．また$t$部はそのヘッダの$h$部を指すように設定される．空リストを表現するこのようなヘッダを**空ヘッダ**と呼ぶ（図4参照）．

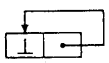


図 4: 空ヘッダ

$t$部はリストの連結や要素の追加の際に参照される．リスト$l_1$の後にリスト$l_2$を連結する場合を考える．この操作を**append**$(H_1, H_2)$と表記する．ここで$l_1$と$l_2$のヘッダをそれぞれ$H_1$と$H_2$とする．連結は$H_1$の$t$部の指す位置に$H_2$の$h$部の値を格納し，$H_2$の$t$部の値を$H_1$の$t$部に格納すればよい（図5参照）．空ヘッダの構造を上記のようにすることによって，同一の手続きが空リストの場合にも適用できる（図6参照）．

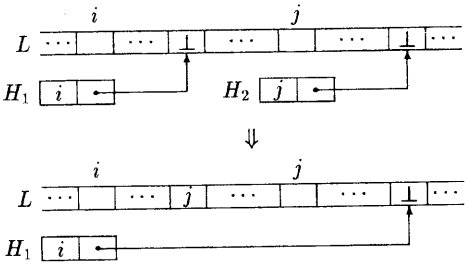


図 5: VP リストの連結 append$(H_1, H_2)$

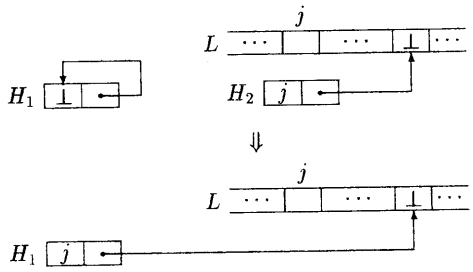


図 6: 空リストへの連結

VP集合の分割・併合の際にリストは破壊的に再構成される．分割された各集合の間ではVPが重複して含まれることはない．したがって各ヘッダに対応したリストは$L$上で互いに干渉することなく表現できる．よってテーブル$L$は実行全体を通じて1つだけあればよい．

## 3.2 エミュレーションループ

前節の例における文$S$の並列実行をVPリスト方式に基づいてエミュレートするコードは以下のようになる．

```
for (i = H.h; i != ⊥; i = L[i])
  E(v_i, S)
```

アクティビティベクタ方式におけるエミュレーションループと異なり，ループの反復回数はアクティブなVP数と一致し，しかも各繰り返しごとのアクティビティの判定は不要となる．

### 3.3 条件文によるVP集合の分割

条件文によるアクティブなVPの集合の分割はリストの各要素を先頭から順に切り離し，条件に応じて2本のリストに再構成する操作となる．前述のif文の例における条件式の処理の部分は以下のようなコードとなる（図7参照）．

```
make_empty(H1); make_empty(H2);
for (i = H.h; i != ⊥; i = L[i])
  if (E(vi,C)) {
    *H1.t = i; H1.t = &L[i];
  } else {
    *H2.t = i; H2.t = &L[i];
  }
*H1.t = ⊥; *H2.t = ⊥;
```

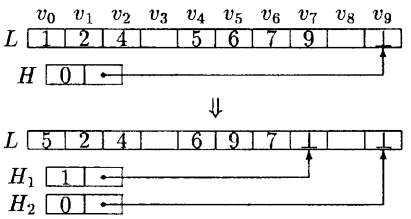


図 7: VP リストの分割例

ここで$H$を空ヘッダにする操作をmake_empty($H$)と表記する．

この操作では，元のリストをループの本体で変更しながらたどる，ということを行なっている．したがって変更しながらでも正しく元のリストの要素をたどることができることを保証する必要がある．それには$L$の各要素が参照前に変更されることはないことを示せばよい．$L$の要素が参照されるのは次の反復における$i$を求める時のみである．また$L$の要素の変更は$H_1$あるいは$H_2$の$t$部のポインタを介してのみ行なわれる．$H_1$あるいは$H_2$の$t$部に格納されるポインタの指す要素の内容は，その直後に必ず次の$i$の値として参照されている．元のリストに巡環はないことを前提とすると，$L$の各要素はたかだか1回しか参照されない．したがって$L$の各要素が参照前に変更されることはない．

### 3.4 VP集合の併合

アクティビティベクタ方式の場合，選択実行の前後でアクティビティが同一となることが明らかである場合には併合操作を必要とはしないが，VPリスト方式の場合は分割した時点で元のリストを破壊しているため，併合操作は必須となる．

```
make_empty(H);
append(H, H1);
append(H, H2);
```

VP集合の併合は単に一方のリストの末尾にもう一方のリストを接合するだけなので，実行時間のコストは数ステップですむ．またリスト中の各VPの順序は選択実行の前後で変化してしまうが，エミュレーションループは各サイクルの実行順序に実行結果が依存しないように構成されているため，プログラムの意味は保存される．

## 4 比較・評価

実際にエンジニアリングワークステーション上でアクティビティベクタ方式とVPリスト方式における処理時間を測定し両方式の効率を比較した．測定はサンプルプログラムにおける実行速度比ついて行なった．測定用の計算機としてはSun SPARC station 5を用いた．

$N$以下の整数に含まれる素数をエラトステネスのふるいで求めるプログラムを図8に示す．このプログラムは，解候補として2から$N$までの数を1つずつ$N-1$個のVPに割り当てておき，各VPは担当する解候補について素数か否かの判定が済むまでテストを繰り返すものである．プログラム中の$minimum(x)$はアクティブなVP間での$x$の最小値を求める演算を意味する．小さな数の解候補を担当しているVPはすぐに判定が終了するため，繰り返しの初期段階でインアクティブとなる．しかし全ての解候補についての判定が終了するまでには$\lfloor\sqrt{N}\rfloor$以下の素数についてのテストが行なわれる必要がある．また初期段階で多くの候補が棄却され，処理が進むにつれて棄却の頻度は減少する．したがって実行期間中はほぼ過半数のVPがインアクティブとなっている．

両方式についてこのプログラムの実行時間を測定し，アクティビティベクタ方式に対するVPリスト方式の速度比を求めた．各$N$について10回ずつ試行を行ない，その平均を求めた．結果を図9にグラフで示す．ただしアクティビティベクタの各要素は1バイトで表現した．全体的には1.6倍から2.7倍と確実に速度が向上している．また傾向としては$N$が大きくなるほど速度比が大きくなっている．

$N$の上昇にともなう速度比の上昇の度合には不規

```
for each v_i ∈ {v_0, v_1, ..., v_{N-2}} in parallel do
  prime = i + 2;
  min = minimum(prime);
  while (min * min < N && prime > min) {
    if (prime % min == 0)
      prime = 0;
    min = minimum(prime);
  }
```

図 8: サンプルプログラム：エラトステネスのふるい

則性が見られるが，これはプログラムの本来の特性によるものと思われる．前述の通り，アクティブな VP の割合によってアクティビティ制御に関する処理時間の比は変動する．しかしながらこのアルゴリズムにおいてアクティブな VP の割合の時間的変化は $N$ に対して線形ではない．そのためこのような不規則性が生じたものと思われる．またこの不規則性には VP リスト方式におけるメモリアクセスの不規則性に起因する部分も含まれていることが推測される．しかしながら１０回の試行における実行時間のばらつきはほとんどなかった．したがってこの測定においてはその影響は無視できる程度であったと思われる．

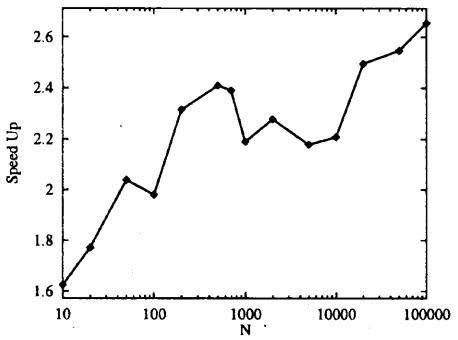


図 9: サンプルプログラムにおける速度向上比

## 5 まとめ

本稿ではデータ並列言語向きのアクティビティ制御方式として従来のアクティビティベクタ方式に対し VP リスト方式を提案し，その有用性を示した．本方式はスカラプロセッサを PP とするほとんどの並列計算機において適用可能である．また，スカラプロセッサを CPU とする通常の逐次計算機上でデータ並列言語のプログラムを実行する際にも有効である．

今回提案した VP リスト方式の実用的な一例に，筆者らの研究室にて開発された EWS 向けの NCX コンパイラ [9] がある．このコンパイラが生成するコードのアクティビティ制御部分に本方式が実際に用いられている．

## 参考文献


[1] MasPar Computer Corp. : "MasPar Parallel Application Language (MPL) Reference Manual", Ver. 2.0 (1991).

[2] Taiichi Yuasa, Motohiko Matsuda and Toshiro Kijima : " SM-1 and its Language Systems", Parallel Language and Compiler Research in Japan, Kluwer Academic Press (1994).

[3] Philip J.Hatcher, Michael J.Quinn : " Data-parallel Programming on MIMD Computers", MIT Press (1991).

[4] A.J.Lapadula : "An Optimizing Dataparallel C Cross-Compiler for Hypercube MultiComputers", Master's thesis, University of New Hampshire, 1991.

[5] Thinking Machines Corp. : "C* Programming Guide", Ver. 6.0 (1990).

[6] Thinking Machines Corp. : "Introduction to Programming in C/Paris", Ver. 5 (1989).

[7] 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, D-I Vol.J78-D-1, No.2, pp.200-209 (1995).

[8] 湯淺太一他 : "超並列 C 言語 NCX 言語仕様書 (Version 3)", 豊橋技術科学大学湯淺研究室 (1993).

[9] 湯淺太一，岡田徹也 : "データ並列言語 NCX の EWS 用処理系の実装", 情報処理学会研究会資料, 96-OS-72-1, pp. 1-6 (1996).