ハイパフォーマンス
コンピューティング 74−14
（1998. 12. 11）

データ並列言語における多重ループの計算分散方式

太田 寛　西谷 康仁
新情報日立研究室

分散メモリ向けのデータ並列言語のコンパイラは，データの分散に基づいて計算を分散する必要がある。計算分散については，従来から1次元配列と1重ループを対象として多くの研究が行われてきた。しかし，多次元配列や多重ループに対する計算分散は，まだ十分に研究が進んでいない。本報告では，配列添字がループ制御変数と1対1に対応しない場合などを含む一般的な多重ループに対して，統一的に計算分散を行う方式を提案する。提案方式は，マッピング標準形による計算マッピングの表現方法，与えられた多重ループに対する計算マッピングの決定方法，計算マッピングに対するループ変換方法から構成される。提案方式を実装し，様々な配列添字を含むループを用いて評価を行った。その結果，生成されたプログラムの性能が従来方式に比べて最小で1.4倍，最大で2.9倍向上した。

Computation Distribution of Nested Loops in Data-Parallel Languages

Hiroshi OHTA, Yasunori NISHITANI
RWCP Hitachi Laboratory

Compilers for data-parallel languages for distributed memory should have features to distribute computation in accordance with data distribution. Computation distribution has been intensively studied for a 1-dimensional array and a single loop. On the other hand, it is not yet sufficiently studied for a multidimensional array and nested loops. In this study, we present a unified computation distribution method for a general loop nest, including the case where array subscripts and loop control variables do not correspond uniquely to each other. Our method consists of representation of the computation mapping by *mapping normal form*, determining computation mapping for given nested loops, and transforming the loops based on the computation mapping. We have implemented the method and evaluated it using loops with various array subscripts. The results show that the performance of the generated programs has improved by a factor of 1.4 - 2.9.

1. はじめに

分散メモリ型マルチプロセッサ向けのプログラミング言語として，HPF(High Performance Fortran)[7]等の様々なデータ並列言語が提案されている。データ並列言語のコンパイラは，データマッピング(データの各プロセッサへの割り当て)に基づいて，計算マッピング(計算処理の各プロセッサへの割り当て)を決定し，並列コードに変換する計算分散機能を備えている必要がある。

1次元配列と1重ループを対象とする計算分散については，従来から多くの研究が行われてきている[1][2][3]。しかし，多次元配列や多重ループを対象とした計算分散は，まだ十分に研究が進んでいない。多重ループには，単なる1重ループの組み合わせでは対処できない様々な場合が存在する。例えば以下のような場合がある。

(1) 配列のある次元の添字がループ制御変数を含まない。

(2) 配列の各次元の添字がループ制御変数と1対1に対応しない。
(3) 配列があるプロセッサ次元に沿って，複数のプロセッサ上に複製されている(複製マッピング)，または，単一のプロセッサ上にのみ存在する(シングルマッピング)。

これらのうち，(2)については，Kennedy ら[4]やRamanujam ら[5]の研究が知られている。しかし，これらの研究では様々な多重ループを統一的に扱うことはできなかった。また，Adve ら[6]は線形不等式系ソルバを利用した多重ループの変換方法を提案しているが，コンパイル時間が長い，プロセッサ数が可変のときに特殊な扱いが必要になるなどの問題点があった。

コンパイル時に計算分散できない場合は，実行時解決法(runtime resolution)という方式に基づくコードが生成されてしまう。これは，全プロセッサが元のループの繰返し範囲全体を実行し，配列参照の度に毎回，参照される要素が自プロセッサに存在するかどうかを判定するというものであり，非常に実行性能が悪い。

本報告では，このような問題を解決するため，一般的な多重ループに対する計算分散方式を提案する。本方式では，配列の分散次元添字がループ制御変数の線形関数で表される場合には，ネストしたループのうち最低でも一つは分散可能となり，実行時解決法を回避することができる。

以下，第2章では配列や計算のマッピングを表現するためのマッピング標準形を導入する。第3章では与えられた多重ループに対して計算マッピングを決定するアルゴリズムを述べ，第4章では決定された計算マッピングに基づいてループを変換するアルゴリズムを述べる。第5章では性能評価によって本方式の有効性を示す。

2. マッピング標準形

本研究では，配列マッピングのパターンとしてはHPF2.0の核部分で規定されているものを対象とする[7]。HPFの公認拡張で規定されているGEN_BLOCK分散やINDIRECT分散は対象外とする。HPFでの一般的な配列マッピングは，テンプレート(メモリ上に領域を取らない仮想的な配列)を介して2段階で行う。すなわち，まず配列をテンプレートに対して整列(align)させ，次にテンプレートを多次元のプロセッサ構成の上に分散(distribute)させる。この2段階マッピングはプログラム記述の際には便利なこともあるが，テンプレートを介するために多少の冗長性を含み，コンパイラ内部でのマッピングの表現方法としては適当でない。

そこで本方式では，配列マッピングの表現方法として，**マッピング標準形**を導入する。これは，配列マッピングを表す様々なパラメタの中から，冗長な情報をできるだけ除去し，ループ分散のために本質的なパラメタだけを抽出して整理したものである。マッピング標準形は表1に示すパラメタから構成される。また，表2，表3にproc_axis_type およびproc_axis_info パラメタの意味を示す。

表1 マッピング標準形

(a) プロセッサ構成の形状		
	proc_rank	プロセッサ構成の次元数
	proc_size	プロセッサ構成の各次元の寸法(次元毎に1つ)
(b) 配列の形状		
	rank	配列の次元数
	size	配列の各次元の寸法(次元毎に1つ)
(c) プロセッサ次元毎マッピング情報 (各次元毎に1セットの情報を持つ)		
	proc_axis_type	NORMAL, REPLICATED, SINGLE のいずれかの値 意味は表2を参照
	proc_axis_info	proc_axis_type の値により，表3の意味を持つ
(d) 配列次元毎マッピング情報 *) (各次元毎に1セットの情報を持つ)		
	is_collapsed	当該次元が分散されていなければTRUE,分散されていればFALSE
	axis_map	当該次元のマッピング先であるプロセッサ次元
	align_lb	当該次元の最初の要素が整列するテンプレート要素のインデックス。ただしテンプレートの下限を0とする
	align_stride	当該次元のテンプレートへの整列ストライド
	blocksize	当該次元に対するテンプレート次元の分散ブロックサイズ

*) is_collapsed がTRUEのときは(d)に分類される他の情報は使用しない。

表2 proc_axis_type の値と意味

値	意味
NORMAL	当該プロセッサ次元に沿って，配列のある次元が分散されている
REPLICATED	当該プロセッサ次元に沿って，配列が複製マッピングされている
SINGLE	当該プロセッサ次元上の単一のプロセッサに，配列がシングルマッピングされている

表3 proc_axis_info の意味

proc_axis_type の値	proc_axis_info の意味
NORMAL	対応する配列次元
REPLICATED	使用せず
SINGLE	配列を持つプロセッサのインデックス

本方式では，計算マッピングを表すのにもマッピング標準形を用いる。すなわち，多重ループ内のある文に着目したとき，その文のインスタンス空間を配列のように見なすことによって，その文の計算マッピングをマッピング標準形で表すことができる。表 1において，

配列　　→　多重ループ全体

配列次元　→　文を囲む個々のループ

と読み替えれば良い。表 2の REPLICATED は重複実行，SINGLE は単一プロセッサでの実行を意味することになる。

さらに計算マッピングに対するマッピング標準系では，以下の拡張を許すことにする。

(1) ある内側ループに対する align_lb，および align_stride は，その内側ループで不変である限り，多重ループ全体では可変であっても良い。

(2) proc_axis_type が SINGLE である場合の proc_axis_info(マッピング先プロセッサインデックス)は，文のインスタンス毎に可変であっても良い。

この拡張の効果は，次節以降で明らかになる。

3. 計算マッピングの決定方法

本章では，与えられた多重ループに対して，計算マッピングを決定する方法を説明する。

まず，計算マッピング決定の基準となる配列参照を一つ選択する。この基準配列参照のオーナプロセッサがその計算を実行するように，計算マッピングを決定する。以降の例では，いわゆる Owner-Computes Rule に基づいて左辺の配列参照を基準として選択しているが，他の方法でも構わない。例えば HPF の ON HOME 指示文で指定された配列参照を選択する方法であっても，本方式は同様に適用できる。

なお，基準配列参照以外の配列参照に対しては一般に通信が必要となるが，これについては今後の別報告にて述べる予定である。

図 1に計算マッピング決定アルゴリズムを示す。アルゴリズムの入力は多重ループおよび基準配列参照，出力は計算マッピング標準形である。

アルゴリズム中で，is_collapsed(*ds*)のように *ds* を添字とするパラメタは，インスタンス空間の第 *ds* 次元に対するものである。また，*dp* を添字とするパラメタは，プロセッサ構成の第 *dp* 次元に対するものである。

また，アルゴリズムの 4 行目のマッピング情報設定は，具体的には以下のようなものである。

```
axis_map(ds) = axis_map(da);
align_lb(ds) = align_stride(da) *
          (F*L_loop + D - L_ary) + align_lb(da);
align_stride(ds) = align_stride(da) * F * S_loop;
blocksize(ds) = blocksize(da);
```

ここで，*da* は当該添字(F*Is+D)を持つ配列次元であり，axis_map(*da*)などは配列マッピング標準形における第 *da* 次元のパラメタを表す。L_loop，S_loop はループの下限と増分，L_ary は配列の第 *da* 次元の宣言下限である。

```
 1: for(基準配列参照を含む文のインスタンス空間の
      各次元 ds につき) do
 2:   if(添字が F*Is+D の形であるような配列分散
      次元がある) then
      /* ここで,
        Is: 次元 ds に対するループの制御変数
        F: 非ゼロ定数
        D: そのループで不変な式
      */
 3:     is_collapsed(ds) = FALSE;
 4:     該当する分散次元から 1 個を選び，当該添字
      に従って次元 ds のマッピング情報を設定;
 5:   else
      /* Is が配列分散次元添字に現れない,
        または現れるが F*Is+D の形でない */
 6:     is_collapsed(ds) = TRUE;
 7:   endif
 8: endfor
 9: for(プロセッサ構成の各次元 dp につき) do
10:   if(上の for ループで既にインスタンス空間次元
      と対応がついている) then
11:     proc_axis_type(dp) = NORMAL;
12:     proc_axis_info(dp) =
          対応するインスタンス空間次元;
13:   else if(配列のどの分散次元とも
      対応しない) then
      /* このとき配列の proc_axis_type(dp)は
        REPLICATED か SINGLE */
14:     proc_axis_type(dp) =
          配列マッピングの proc_axis_type (dp);
15:     proc_axis_info(dp) =
          配列マッピングの proc_axis_info(dp);
16:   else
      /* 配列のある分散次元 da と対応する。*/
17:     proc_axis_type(dp) = SINGLE;
18:     proc_axis_info(dp) = 配列次元 da の添字値の
          マッピング先プロセッサインデックス;
19:   endif
20: endfor
```

図 1 計算マッピング決定アルゴリズム

以下，例を用いてアルゴリズムの要点を説明する。

[例 1]

```
  real a(0:99, 0:99, 0:99)
!HPF$ processors p(0:3, 0:3)
!HPF$ distribute a(block,*,block) onto p
  do i1 = 0,98
    do i2 = 0,99
      do i3 = 2,50
        a(i1+1,i2,2*i3-1) = ...
      enddo
    enddo
  enddo
```

最も普通の場合として，全てのループ制御変数が配列次元と1対1に対応し，かつ添字が線形である例を考える。i1 は配列の一つの分散次元(第 1 次元)に一次式の形で現れるので，図 1の2行目で then 節が適用され，i1 ループはプロセッサ構成の第 1 次元上に分散される。なおアルゴリズムの 4 行目では「該当する分散次元から 1 個を選び」となっているが，この場合はそのような次元が一つしかないので，選択の余地はないことに注意。同様に i3 ループは，プロセッサ構成の第 2 次元上に分散される。i2 ループは，i2 を添字に含む配列次元はあるが(第 2 次元)，それが非分散なので図 1 の 5 行目以降の else 節が適用され，i2 ループも非分散(is_collapsed(*ds*) = TRUE)となる。

アルゴリズムの後半の for において，プロセッサ構成の第 1，第 2 次元に対して 11,12 行目が適用され，それぞれ，i1 ループ，i3 ループに対する NORMAL となる。

結局，次の計算マッピング標準形が得られる。プロセッサ構成やインスタンス空間の各次元のパラメタは第 1 次元を先頭にして横に並べてある。

proc_rank	2		
proc_size	4	4	
proc_axis_type	NORMAL	NORMAL	
proc_axis_info	1	3	
rank	3		
size	99	100	49
is_collapsed	FALSE	TRUE	FALSE
axis_map	1	-	2
align_lb	1	-	3
align_stride	1	-	2
blocksize	25	-	25

[例 2]

```
   real a(0:99, 0:99, 0:99)
!HPF$ processors p(0:3, 0:3)
!HPF$ distribute a(block,*,block) onto p
   do i1 = 0,98
     do i2 = 0,99
        a(i1,i2,99) = ...
     enddo
   enddo
```

配列参照のある分散次元の添字が定数の場合は，シングルマッピングとして扱われる。この例では，i1, i2 ループは，[例 1]と同様にアルゴリズム前半の for ループによって，それぞれ分散，非分散となる。プロセッサ構成の第 2 次元はインスタンス空間次元と対応せず，かつ配列の分散次元(第 3 次元)と対応しているので，アルゴリズムの 16 行目の else 節が適用されシングルマッピングとなる。

計算マッピング標準形は次のようになる。網かけの部分がシングルマッピングを表している。proc_axis_info はマッピング先プロセッサインデックスが 99/25=3 であることを示している。ここで 25 は配列マッピングの blocksize である。

proc_rank	2	
proc_size	4	4
proc_axis_type	NORMAL	SINGLE
proc_axis_info	1	3
rank	2	
size	99	100
is_collapsed	FALSE	TRUE
axis_map	1	-
align_lb	0	-
align_stride	1	-
blocksize	25	-

[例 3]

```
   real a(0:99, 0:99, 0:99)
!HPF$ processors p(0:3, 0:3)
!HPF$ distribute a(block,*,block) onto p
   do i1 = 0,98
     do i2 = 0,99
        a(i1, i2, i1+1) = ...
     enddo
   enddo
```

この例は，あるループの制御変数が複数の分散次元に現れる場合である。この場合，アルゴリズムの 3,4 行目によって，i1 ループは配列の第 1 次元の添字 'i1' または第 3 次元の添字 'i1+1' のいずれかにしたがって分散される。どちらを選ぶかはアルゴリズムでは規定されていないが，ここでは仮に第 1 次元が選ばれたとする。選ばれなかった配列第 3 次元のマッピング先であるプロセッサ第 2 次元は，アルゴリズム後半の for ループにおいて，インスタンス空間次元と対応しなくなる。そのため，アルゴリズムの 17 行目によってシングルマッピングとなる。proc_axis_info は配列インデックス'i1+1'のマッピング先プロセッサインデックスであるから，(i1+1)/25 となる。

計算マッピング標準形は，[例 2]のものと次の2箇所のみが異なる。

proc_axis_type	NORMAL	SINGLE
proc_axis_info	1	(i1+1)/25

このようなマッピングが表現可能となることが，第 2 章の最後で述べた，SINGLE のときに proc_axis_info が文のインスタンス毎に可変であるのを許すという拡張の効果である。

[例 4]

```
   real a(0:99, 0:99)
!HPF$ processors p(0:3, 0:3)
!HPF$ distribute a(block,block) onto p
   do i1 = 1,49
     do i2 = 0, 49
       a(i1-1, i2+i1) = ...
     enddo
   enddo
```

この例では，配列の第 2 次元の添字に複数のルー

プの制御変数が混在している。アルゴリズム前半のforループで，まずi1ループについて，第1次元添字 'i1-1' によって3,4行目が適用される。このとき，第2次元添字 'i2+i1'は，内側ループ制御変数i2がi1ループ不変ではないので,F*Is+Dの形とは見なされないことに注意。次に，i2ループについて考えるときは，制御変数i1はi2ループ不変式なので,第2次元添字 'i2+i1' はF*Is+Dの形と見なされ，やはりアルゴリズムの3,4行目が適用される。したがって，両方のループが分散される。

計算マッピング標準形は次のようになる。第2次元のalign_lbがi1となっているが，これはi2=0のときに配列の第2次元添字がi1となることによるものである。これによって，i2ループのマッピングが表現可能となっている。これは，第2章の最後で述べた，内側ループに対するalign_lbが，多重ループ全体では可変である場合を許すことの効果である。

proc_rank	2	
proc_size	4	4
proc_axis_type	NORMAL	NORMAL
proc_axis_info	1	2
rank	2	
size	49	50
is_collapsed	FALSE	FALSE
axis_map	1	2
align_lb	0	i1
align_stride	1	1
blocksize	25	25

なお，この例からも分かるように，配列の分散次元添字がループ制御変数の線形結合である場合，最低でも，それらのループの内で最内側のものは分散できることになる。なぜなら，最内側ループにおいては，外側ループ制御変数はすべて不変なので,添字はF*Is+Dの形と見なされるからである。

4. 計算マッピングに基づくループ変換

本節では,計算マッピングが与えられたときの，ループの変換方法を述べる。図2のアルゴリズムにしたがって変換することにより，各プロセッサが実行するSPMDプログラムが得られる。

以下，前章の[例3]を用いてアルゴリズムを説明する。

まずアルゴリズムの前半のforループを考える。インスタンス空間の第1次元のis_collapsedはFALSEであるから，アルゴリズムの5行目によりi1ループの範囲が縮小される。一方，第2次元のis_collapsedはTRUEであるから，アルゴリズムの3行目によりi2ループの範囲は元のままとなる。

次に後半のforループを考える。プロセッサ構成の第1次元のproc_axis_typeはNORMALであるから，この次元については何もしない。一方，第2次元のproc_axis_typeはSINGLEであるから,第2次元のプロセッサインデックスが(i1+1)/25 のプロセッサのみが実行するように，if文を生成してループボディに保護を掛ける。

```
1:  for(インスタンス空間の各次元 ds につき) do
2:    if(is_collapsed(ds) == TRUE) then
3:        ループ範囲を元のままとする;
4:    else
5:        当該次元 ds のマッピングにしたがって
          ループ範囲を縮小する;
6:    endif
7:  endfor
8:  for(プロセッサ構成の各次元 dp につき) do
9:    if(proc_axis_type(dp) == SINGLE) then
10:       proc_axis_info(dp)で指定される
          プロセッサだけが実行するように
          ループボディに保護を掛ける;
11:   else  /* NORMAL または REPLICATED */
12:       何もしない;
13:   endif
14: endfor
```

図2 ループ変換アルゴリズム

変換後のプログラムを以下に示す。

```
real a(0:24, 0:99, 0:24)
do i1 = L1,U1,S1
  do i2 = 0,99
    if(my_pindex(2) == (i1+1)/25) then
      a(i1, i2, mod(i1+1,25)) = ...
    endif
  enddo
enddo
```

ここで，L1,U1,S1は縮小されたループ範囲であり，1重ループに対する各種の従来方法[1][2]によって求められる。また，if文内のmy_pindex(2)は自プロセッサの第2次元のインデックスを表す。なお，配列第3次元の添字はmod演算によりローカルインデックス化されている。

シングルマッピングに対する簡単な最適化として，マッピング先プロセッサが多重ループ全体で不変ならば，if文による保護を多重ループの外へ移動することができる。例えば，前章の[例2]ではproc_axis_info(2)=3 で不変であるから，保護移動が適用できる。このとき，変換後のプログラムは以下のようになる。

```
real a(0:24, 0:99, 0:24)
if(my_pindex(2) == 3) then
  do i1 = L1,U1,S1
    do i2 = 0,99
      a(i1, i2, 24) = ...
    enddo
  enddo
endif
```

5. 評価

提案方式のプロトタイプを我々の開発した HPF コンパイラ[8]上に実装し評価を行った。評価用プログラムとしては，以下の 3 種類のカーネルループを用いた。

ループ1	ループ2	ループ3

real a(N,N)
do i = 1,N
  a(i,i)= ...
enddo
	real a(N,N)
	do j = 1,N/2
	do i = 1,N/2
	  a(i+j,j)= ...
	enddo
	enddo
		real a(N,N)
		do i = 1,N
		  a(1,i) = ...
		enddo

図 3 評価用プログラム

いずれの場合も配列は(block,block)で分散した。また，Nの値は，ループ1,ループ3では5000，ループ2では2000とした。代入文の右辺の計算は，文の1インスタンス当りの演算量が保護のオーバヘッドと同程度になるように調節した。通信の影響を除くため，右辺の変数は全プロセッサ上に複製し，通信が発生しないようにした。

上記プログラムを，実行時解決法および提案方法の2通りでコンパイルし，生成されたプログラムの性能を測定した。またループ3については，提案方法において保護のループ外移動を行ったものも測定した。使用したマシンはSR2201で，プロセッサ構成は2*2および4*4とした。

図 4に元のプログラムを逐次実行した場合に対する性能向上比を示す。すべての場合において，提案方法は実行時解決法に比べて良い性能が得られている。その比率は最小で1.4倍(ループ1,2*2)，最大で2.9倍(ループ2, 4*4)である。また，ループ3では保護の移動によりさらに20%～25%の性能向上が得られている。これらのプログラムでは，ループの性質上，プロセッサ数がP*Pのときには P 倍の性能向上が理論的限界であるが，ループ2やループ3では，ほぼその限界に到達する性能が得られている。以上により，提案方式の有効性が確認できた。

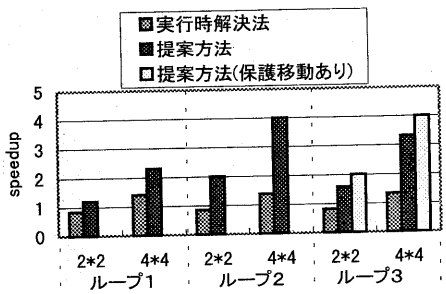

図 4 逐次実行に対する性能向上比

6. おわりに

分散メモリ向けデータ並列言語のコンパイラにおける，一般的な多重ループに対する計算分散方式を提案した。本方式により，配列添字がループ制御変数と 1 対 1 に対応しない場合などを含む様々な場合が統一的に取り扱えるようになる。

本方式のプロトタイプを我々の開発した HPF コンパイラ上に実装し，様々な配列添字を含むループを用いて評価を行った。本方式により生成されたプログラムの実行性能は，従来の実行時解決法に比べて，最小で 1.4 倍，最大で 2.9 倍向上した。また，簡単な最適化によって，さらに 20%～25%の性能向上が得られた。評価プログラム 3 本中の 2 本については，ほぼ理論的限界の性能が達成された。

今後は，実アプリケーションや代表的ベンチマークなどに本方式を適用し，有効性を検証する予定である。

参考文献

[1] S. K. S. Gupta, S. D. Kaushik, C.-H. Huang, and P. Sadayappan, "Compiling Array Expressions for Efficient Execution on Distributed-Memory Machines," J. of Parallel and Distributed Computing, Vol. 32, pp. 155-172, 1996.

[2] K. van Reeuwijk, W. Denissen, H. J. Sips, and E. M. R. M. Paalvast, "An Implementation Framework for HPF Distributed Arrays on Message-Passing Parallel Computer Systems," IEEE Trans. on Parallel and Distributed Systems, Vol. 7, No. 9, pp. 897-914, 1996.

[3] J. Ramanujam, S. Dutta, A. Venkatachar, and A. Thirumalai, "Advanced Compilation Techniques for HPF," Seventh International Workshop on Compilers for Parallel Computers, 1998.

[4] K. Kennedy, N. Nedeljkovic, and A. Sethi, "Efficient Address Generation for Block-Cylic Distributions," Int'l Conf. on Supercomputing '95, pp. 180-184.

[5] J. Ramanujam, S. Dutta, and A. Venkatachar, "Code Generation for Complex Subscripts in Data-Parallel Programs," Tenth Workshop on Languages and Compilers for Parallel Computing, 1997.

[6] V. Adve and J. Mellor-Crummey, "Using Integer Sets for Data-Parallel Program Analysis and Optimization," SIGPLAN '98 Conf. on Programming Language Design and Implementation, pp. 186-198.

[7] High Performance Fortran Forum, "High Performance Fortran Language Specification Version 2.0", Rice University, 1997.

[8] 佐藤真琴, 太田寛, 布広永示, "HPF トランスレータ Parallel FORTRAN の開発と評価," 情報処理, Vol. 38, No. 2, pp. 105-108, 1997.