システムソフトウェアと
オペレーティング・システム 67－20
(1994. 12. 9)

データ依存不確定ループの最内ループ並列化

小笠原 武史, 小松 秀昭

日本アイ・ビー・エム（株）東京基礎研究所

神奈川県大和市下鶴間1623-14, {takeshi,komatsu}@trl.ibm.co.jp

あらまし

分散メモリ型並列計算機のためのデータ並列言語コンパイラでは、メッセージベクタ化が重要である。しかし右辺の配列添字に登場するコンパイル時に未知の変数が原因で、anti あるいは true dependence のどちらかに固定しない場合があり、その時メッセージベクタ化ができない。従来研究の成果に versioning があるが、SPMD コード生成における場合分けによるアプローチはコードエクスプロージョンを引き起こす。本稿では versioning を使わずに最内ループが持つ並列性を引き出す手法を提案する。

和文キーワード　コンパイラ、並列化、データ並列言語、分散メモリ並列計算機

Innermost loop pipelining: a method for exploiting parallelism of innermost loops with dynamic dependences

Takeshi Ogasawara and Hideaki Komatsu

Tokyo Research Laboratory, IBM Japan,Ltd.

1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242, Japan

Abstract

Message vectorization is an important optimization technique for distributed-memory compilers, but it cannot be performed if there is any unresolved data dependence in the target loop nest. The existing approach, called "versioning", solves this problem. However, it has a serious drawback due to its code explosion problem, particularly when SPMD code is generated. In this paper, we propose a new technique that exploits innermost loop parallelism without using versioning. In this new approach, message vectorization can be performed even when there is an unresolved data dependence in the innermost loop.

英文 key words　compiler, parallelization, data parallel language, distributed memory machine

1 はじめに

一般にデータ並列言語用のコンパイラ [1, 2] は、ループ実行に必要なデータをループ実行の前に各プロセッサがベクタメッセージ通信 [3] によりプリフェッチし、通信や同期なしにループを実行させることが重要である。そのためには各プロセッサが読み込むデータをどのプロセッサが所有し、それらを所有するプロセッサがどの時点で定義した値を読み込むのかを解析し、そのデータ依存を維持するように通信コードを生成することが重要になる。これらはデータ依存解析の結果に基づいて行なわれる [4, 3]。

しかし dynamic dependences[5] と分類される図 1の例のようなループでは、データ依存 [6] が true dependence (書き込みその後読み込み) あるいは anti dependence (読み込みその後書き込み) のどちらにも実行時まで決まらないような場合、データ依存に基づくベクタ化メッセージを使用したループ並列化 [7] は不可能であった。

```
read(n)
do i=1, 20
  a[i] = a[n]+3
end do
```

図 1: シンプルな例

このような場合に対するアプローチとしては versioning が提案されている [8]。versioning はデータ依存が不明な時に場合わけをするコードをインライン展開する手法である。例えば図 1 の例では、図 2 のように展開される。

```
if (n >= 20) then
  do i=1, 20
    a[i] = a[n]+3
  end do
else
  do i=1, n-1
    a[i] = a[n]+3
  end do
  a[n] = a[n]+3
  do i=n+1, 20
    a[i] = a[n]+3
  end do
endif
```

図 2: versioning されたシンプルな例

versioning された結果のコードに含まれる 3 つのループはそれぞれ loop-carried なデータ依存がなくなり、プリフェッチによるループ並列化が可能になる (図 3)。

しかしながら図 3のように versioning はコード・エクスプロージョンを起こす。前述の例では右辺が 1 オペランドであったが、右辺の数が増えるに従ってさらに複雑なコードで場合わけを行なわなくてはならない。

```
if (n >= 20) then
  call compute_lis((1:20),a[i],lb1,ub1)
  call compute_receivers((1:20),a[i],a[n],PG_recv1)
  if (i_own(a[n])) then
    call send_to_grp(a[n],PG_recv1)
  else
    call receive_from(buf,owner(a[n]))
  endif
  do i=lb1, ub1
    a[i] = buf+3
  end do
else
  call compute_lis((1:n-1), a[i], lb2, ub2)
  call compute_receivers((1:n-1),a[i],a[n],PG_recv2)
  if (i_own(a[n])) then
    call send_to_grp(a[n],PG_recv2)
  else
    call receive_from(buf,owner(a[n]))
  endif
  do i=lb2, ub2
    a[i] = buf+3
  end do
  if (i_own(a[n])) then
    a[n] = a[n]+3
  endif
  call compute_lis((n+1:20), a[i], lb3, ub3)
  call compute_receivers((n+1:20),a[i],a[n],PG_recv3)
  if (i_own(a[n])) then
    call send_to_grp(a[n],PG_recv3)
  else
    call receive_from(buf,owner(a[n]))
  endif
  do i=lb3, ub3
    a[i] = buf+3
  end do
endif
```

図 3: versioning されたシンプルな例の SPMD コード

一般的に SPMD プログラムは 1 つのコードで複数プロセッサに対応するため、単一プロセッサ用のコードに比べてかなり大きくなる。versioning のような詳細な場合分けによる最適化を一様に適用すると、コードサイズをさらに増大させてしまう。そのため、できるだけコードサイズを大幅に増やさずに並列化する手法が重要である。

本稿で提案する最内ループパイプライン化方式は、dynamic dependences を持ったループについて versioning を用いずにループ並列化を行なう新しいアプローチである。

この方式が利用している特性を、図 1を例にとって、説明する。まず最初に、このループでは、インデックスセット

- $i = 1..n$ の実行では、ループ実行前の $a[n]$ の値、
- $i = n+1..20$ の実行では、ループ実行後の $a[n]$ の値

を必要とする。n が実行時に決まれば、$a[n]$ を所有するプロセッサは、ループ実行前にインデックスセット $i = 1..n$ を実行するプロセッサ群に、ループ実行後に $i = n+1..20$

を実行するプロセッサ群に $a[n]$ の値を送信すれば、データ依存を保証できる。

さらに配列 a がプロセッサによってブロック状に分割されていれば、それらのプロセッサそれぞれを、$i = 1..n$ を実行するプロセッサか、あるいは $i = n+1..n$ を実行するプロセッサかのどちらか分類することができる。すると $a[n]$ を受信するプロセッサはそれぞれ、

1. ループ実行前に送信された $a[n]$ の値を受信待ちか、
2. ループ実行後に送信された $a[n]$ の値を受信待ち

のどちらかに分類される。同期が $a[n]$ を所有するプロセッサに対してのみ必要なので、分類1に属すプロセッサ同士は並列に実行される。分類2についても同様である。以下、分類1、2に分類されるプロセッサ群をそれぞれ *PG.pre*、*PG.post* と呼ぶ。

以上述べてきたように、図1のループの実行では、各プロセッサはループ実行の前に $a[n]$ を所有するプロセッサと同期を1回とるだけで、並列に実行できる。

一般に n 重ループ、m 次元配列がブロック状に分割されている場合でも、dynamic dependence の要因が最内ループ i_n と j 番目の配列添字（$j = 1..m$）であったときに、同じ議論が成り立つ。

以下、2章で dynamic dependences の存在でループ並列化ができない場合の影響を説明する。3章で最内ループパイプライン化方式を説明する。最後に実装効果等を考察する。

2 ループ並列化における Dynamic Dependences の影響

本章では最初にデータ並列プログラムにおいて重要なループ並列化とその条件を説明する。次に dynamic dependences を持つ複雑な実際例を示し、並列化されないSPMD コードの効率の悪さを示す。

2.1 メッセージベクタ化によるループ並列化

ループのインデックス変数のとる値の集合をインデックスセットと呼ぶ。ループ並列化では *owner computes rule*[4] に従ってインデックスセット分配 [3] が行なわれるのが一般的である。ループを並列に実行するためには：

1. インデックスセット分配が可能であること、
2. そのループの本体に含まれる代入文の右辺について発生するすべての通信を、そのループの外側で行なうこと（メッセージベクタ化）

が必要である。

コンパイラはデータ依存解析で判明した loop-carried なデータ依存に基づき、true dependence ならばループ前プリフェッチのための、anti dependence ならばパイプラインのためのベクタメッセージ通信コードを生成する [7]。

つまりコンパイル時に (i) インデックスセットが分配可能かどうかが決まり、(ii) 代入文右辺のデータ依存が1つに決められれば、コンパイラはループを並列実行するコードが生成できる。

2.2 Dynamic Dependences と SPMD コード

1章の図1を再度考える。前述のように versioning を適用しないとコンパイル時にデータ依存が anti dependence あるいは true dependence のどちらか一方に決まらず、ループ並列化できない。この場合に生成される疑似 SPMD コードを、図1の例を使用して図4に示す。

```
do i=1, 20
  if (i_own(a[i])) then
    if (i_own(a[n])) then
      buf = a[n]
    else
      call receive_from(buf,owner(a[i]))
    endif
  else
    if (i_own(a[i])) then
      call send_to(a[n],owner(a[n]))
    endif
  endif
  if (i_own(a[i])) then
    a[i] = buf+3
  endif
end do
```

図 4: 並列化されていない SPMD コード

図4に登場する関数は以下のようなものである。*i_own* は指定された配列要素を自分が所有しているかどうかの真偽を返し、自分が所有しない配列要素へのアクセスを防ぐための IF 文（プロセッサガードと呼ぶ）を構成するために使われる。*owner* は指定された配列要素を所有するプロセッサを返し、*send_to* と *receive_from* とで指定された配列要素を通信する。

図4の SPMD コードを実行すると、それぞれのプロセッサはシングルプロセッサでの実行と同等の実行を行う。違いは、自分が所有しない配列への書き込みをガードすることと、自分が所有しない配列の読み込みを、所有するプロセッサからの通信で実現することである。

図4の SPMD コードの実行性能を下げている主な要因は、シングルプロセッサ実行でのセマンティクスを保持

するために

1. 全プロセッサがインデックスセット全域を実行し、
2. 通信がアクセスされる配列要素ごとに行われる

ことである。特に分散メモリ型計算機ではデータが使用可能になるまでの latency が非常に長いことを考慮すると、このSPMDコードの実行はシングルプロセッサで実行するのと比べてかえって遅くなる。

一般的にHPFのようなデータ並列言語が対象としているプログラムでは配列サイズが非常に大きいため、通信回数が配列サイズのオーダーであることは大きな問題となる。

3 最内ループパイプライン化

本章では最初に図1のループが持つ並列性を示す。次に図1を一般化して最内ループパイプライン化が対象とするループの定義をする。最後に、最内ループが持つ並列性を引き出すパイプライン化コード生成の説明をし、生成される疑似SPMDコードを示す。

3.1 最内ループの並列性

図1は n の値に依存して、loop-independent なデータ依存の場合もあれば、インデックスセットの一部が anti dependence で残りの一部が true dependence の場合もある。

しかし $a[n]$ の所有者プロセッサが、$PG.pre$ に属するプロセッサへ代入文実行前に $a[n]$ の値を、$PG.post$ に属するプロセッサへ代入文実行後に $a[n]$ の値を送信すれば、$PG.pre$、$PG.post$ それぞれの中で並列に実行でき、$PG.pre$ の実行と $PG.post$ の実行とで順序が発生するのみである。

例えば配列 a が2要素ずつ10台のプロセッサにブロック状に分割されていたとすると、$n=10$ のとき、$PG.pre$={プロセッサ1..4}、$PG.post$={プロセッサ6..10}となり、プロセッサ1から5までが互いに並列に、プロセッサ6から10までが互いに並列に実行される。プロセッサ1から5までのグループの実行開始と、プロセッサ6から10までのグループの実行開始は、プロセッサ5の実行の前と後と順序つけされる。図5は、これらの時間関係の概略を示したものである。図5のpiはプロセッサ、実線矢印は同期関係、影線矢印は実行時間推移、を示す。

以上の議論から、ループ実行前に $PG.pre$、$PG.post$ を計算し通信するコードを生成すれば図5で示す並列性を引き出すことができる。これは図2のversioningされたプログラムの各ループをプリフェッチで処理するのと同じ効果を持つが、versioningプログラムよりも9で述べるようなメッセージベクタ化を行い易い。以下、$PG.pre$、$PG.post$ の計算と通信による並列化を最内ループパイプライン化と呼ぶ。

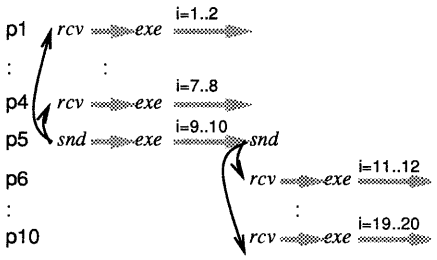

図5: プロセッサ数10,n=10,block分割の時の実行

3.2 対象ループの定義

図1で示されたループを、最内ループにおけるdynamic dependence、という特性を持つ n 重ループ、m 次元配列に一般化したものを図6に示す。$i_1..i_n$はそれぞれループインデックス変数、$UB_1..UB_n$はループ上限である。$Slhs_i$、$Srhs_i$はそれぞれ左辺、右辺の $i(i=1..m)$ 次元の添字式である。$Expr(i_n)=C1*i_n+C2$ であり $InvExpr=C3$ である。ここで $C1, C2, C3$ はそれぞれ i_nについてループ不変数から構成される式であり、$i_1..i_{n-1}$も含んでもよい。

図6のように、左辺と右辺の配列参照が1つの次元を残してすべて同じ添字式を持つ場合を、シンプルな参照と呼ぶ[9]。

シンプルな参照のみを含むループでは、とくに図6のように $Expr(i_n)$ が $C1*i_n+C2$ のような特定のループインデックスの一次式の場合、そのループに関するデータ依存のみしか発生しない。そのため3.1で述べたループ並列性が第 n ループについても同じく存在する。

図6における右辺の a のようにdynamic dependenceを発生させているオペランドが複数登場する場合にも、図6に関する最内ループの並列性を一般性を失わずに拡張できるので、以下図6を標準ループ形と呼びこのループに対して議論を進める。

```
do i1=1, UB1
  :
  do in=1, UBn
    a(Slhs1,..,Slhsj-1,Expr(in),Slhsj+1,..,Slhsm)=
    F(a(Srhs1,..,Srhsj-1,InvExpr,Srhsj+1,..,Srhsm))
  end do
  :
end do
```

図6: 標準ループ

表 1: $PG.pre$, $PG.post$ の計算

	条件	$IS.pre$	$IS.post$
1	$mod(C3-C2,\|C1\|)!=0$	$i_n=1,UB_n$	なし
2	$mod(C3-C2,\|C1\|)=0 \wedge (C1>0) \wedge (C1+C2>C3)$	$i_n=1,UB_n$	なし
3	$mod(C3-C2,\|C1\|)=0 \wedge (C1>0) \wedge (C3>C1*UB_n+C2)$	$i_n=1,UB_n$	なし
4	$mod(C3-C2,\|C1\|)=0 \wedge (C1>0) \wedge (C1+C2 \leq C3 \leq C1*UB_n+C2)$	$i_n=1,(C3-C2)/C1$	$i_n=(C3-C2)/C1+1,UB_n$
5	$mod(C3-C2,\|C1\|)=0 \wedge (C1<0) \wedge (C1*UB_n+C2>C3)$	$i_n=1,UB_n$	なし
6	$mod(C3-C2,\|C1\|)=0 \wedge (C1<0) \wedge (C3>C1+C2)$	$i_n=1,UB_n$	なし
7	$mod(C3-C2,\|C1\|)=0 \wedge (C1<0) \wedge (C1*UB_n+C2 \leq C3 \leq C1+C2)$	$i_n=(C3-C2)/C1,UB_n$	$i_n=1,(C3-C2)/C1-1$

ここで 3.1で行った議論を標準ループについて再度行っておく。標準ループは最内ループに関して、添字式 $Expr(i_n), InvExpr$の関係によりデータ依存が anti にも true にも確定しない。そのためデータ依存に基づくメッセージベクタ化を使ったループ並列化はできない。一方、$InvExpr$がコンパイル時に未定な値でもループ不変値である限り、実行時においては定数とみなすことができる。配列の j次元目がブロック状に分割されていれば、$PG.pre$、$PG.post$ を求めることによって最内ループを並列化できる。

すなわち (i) 標準ループで (ii) 配列の j次元目がブロック分割されているという 2 つの条件を満たせば最内ループパイプライン化を適用して並列化ができる。

3.3 コード生成

問題としているループが標準ループの形で、配列 a はブロック状に分割されているとコンパイラが認識した時、以下のような手順でコード生成する。

3.3.1 インデックスセット分割コード

インデックスセットをプロセッサ毎に分配する。インデックスセットの分配は、loop bounds reduction として [3] などに詳しいため、詳細は省略する。この計算を行なう手続きを

$call\ compute\_1D\_lis(a(..,Expr,..),lb,ub)$

とする。ここで lb,ub は、この手続きを呼んだプロセッサの分配されたインデックスセットのそれぞれ下限、上限である。

3.3.2 $PG.pre, PG.post$ の計算コード

$PG.pre$ と $PG.post$ を計算する手続きを

$call\ compute\_pre\_post(UB_n,$
$a(..,Expr,..),a(..,InvExpr,..),PG_{pre},PG_{post})$

とする。$compute\_pre\_post$ は、$Expr=C1*i_n+C2$ と $InvExpr=C3$ の関係と配列分割情報に基づいて、ループ本体である代入文で実行前の値を使うインデックスセットと、実行後の値を使うインデックスセットとに分類する。前者を $IS.pre$、後者を $IS.post$ と呼ぶ。表 1は $C1,C2,C3$ の関係と、対応するインデックスセット分類である。

$IS.pre, IS.post$ を計算後、さらに配列 a の分割情報 dsc_a と j次元目の添字式 $Expr(i_n)=C1*i_n+C2$ を使って、$IS.pre, IS.post$ の全てあるいは一部をインデックスセットとして持つプロセッサ群を計算し、PG_{pre}, PG_{post} として返す。

3.3.3 通信コード

問題となる右辺 $a(..,InvExpr,..)$ を所有するプロセッサはループの実行前に

$call\ send\_to\_grp(a(..,InvExpr,..),PG_{pre})$

ループの実行後に

$call\ send\_to\_grp(a(..,InvExpr,..),PG_{post})$

を実行することによって $a(..,InvExpr,..)$ の値を送信する。

それ以外の $PG.pre \cup PG.post$ に属すプロセッサは、ループ実行前に送信された $a(..,InvExpr,..)$ の値を

$call\ receive\_from(buf,owner(a(..,InvExpr,..)))$

によって受信する。

3.4 疑似 SPMD コード

図 7は最内ループパイプライン化された標準ループの SPMD コードである。関数 *in_PG* は、関数を呼んだプロセッサが指定されたプロセッサグループに入っているかどうかを返す。

```
call compute_1D_lis((1:UB_n),a(..,Expr,..),lb,ub)
if (i_own(a(..,InvExpr,..))) then
  call compute_pre_post(
       UB_n,a(..,Expr,..),a(..,InvExpr,..),PG_pre, PG_post)
  call send_to_grp(a(..,InvExpr,..),PG_pre)
else
  call receive_from(buf,owner(a(..,InvExpr,..)))
endif
do in=lb, ub
  a(..,Expr,..) = F(buf)
end do
if (i_own(a(..,InvExpr,..))) then
  call send_to_grp(a(..,InvExpr,..),PG_post)
endif
```

図 7: 最内ループパイプライン化された標準ループに対する SPMD コード

4 考察

4.1 非並列化コードに対する効果

最内ループパイプライン化を行なわない時の SPMD コードが呼び出す実行時ライブラリとその回数と、行なった場合とを比較する。

まず行なわない場合では、左辺のガードのために UB_n 回 *i_own* を呼ぶ。左辺ガードが true のとき、右辺読み込みのガードのために *i_own* を $|LIS|$ 回呼ぶ。$|LIS|$ は各プロセッサに分配されたインデックスセットのサイズである。左辺ガードが false のとき、右辺送信のガードのために $UB_n - |LIS|$ 回 *i_own* が呼ばれる。すなわち、ガードのために合計 $2 * UB_n$回の *i_own* が呼ばれる。

また右辺を所有するプロセッサでは $UB_n - |LIS|$ 回の *send_to*, *owner*が、それ以外のプロセッサでは $|LIS|$ 回の *receive_from* が呼ばれる。

配列 a の j次元目を分割しているプロセッサ数を P_jとする。$|LIS| = UB_n/P_j$であるので、実行時ライブラリのプロセスブロックする時間を除く実行時間をそれぞれ $Time_{i\_own}, Time_{send}, Time_{owner}, Time_{recv}$ とすると、実行時ライブラリで消費される時間は次のようになる。右辺を所有するプロセッサの時間を $Time_s$、それ以外のプロセッサを $Time_r$とする。

$$
\begin{aligned}
Time_s(UB_n, P_j) &= 2 * UB_n * Time_{i\_own} + \\
&\quad (1 - 1/P_j) * UB_n * Time_{send} + \\
&\quad (1 - 1/P_j) * UB_n * Time_{owner} \\
Time_r(UB_n, P_j) &= 2 * UB_n * Time_{i\_own} + \\
&\quad (1 - 1/P_j) * UB_n * Time_{owner} + \\
&\quad UB_n/P_j * Time_{recv}
\end{aligned}
$$

さらに 1 プロセッサ当たり持つ次元要素の数である UB_n/P_jを N_{epp}とおき P_jが十分に大きいとすると、式は次のように変形できる。

$$
\begin{aligned}
&Time_s(UB_n) \approx \\
&\qquad UB_n * (2 * Time_{i\_own} + Time_{send} + Time_{owner}) \\
&Time_r(UB_n, N_{epp}) \approx \\
&\qquad UB_n * (2 * Time_{i\_own} + Time_{owner}) + N_{epp} * Time_{recv}
\end{aligned}
$$

$Time_s, Time_r$は UB_n, N_{epp}に比例する、一方、最内ループ最適化に必要なライブラリの実行時間 $Time_i$は UB_n, N_epp と無関係である。

4.2 適用例

図 8に示すのは Perfect ベンチマークに登場する、pivot を用いた Gauss-Jordan 法による逆行列計算ルーチンに見られるループで、誘導変数処理、ループ正規化を終えた後のものである。

```
do iv1=1, n
  do iv2=1, n
    if (iv1 != k) then
      if (iv2 != k) then
        kj = n*iv2 - n + iv1 - iv1 + k
        a(iv1+n*j-n) = a(iv1+n*iv2-n)+
                       a(iv1+nk)*a(kj)
      endif
    endif
  end do
  i = 1 + iv2
end do
```

図 8: Perfect ベンチマークに見られるループ

配列 a がブロック状に分割されている時、図 8は $C1 = n, C2 = iv1 - n, C3 = iv1 + nk$のときの標準ループの形であるため、最内ループ最適化による並列化が可能である。

4.3 ベクタメッセージ

最内ループ並列化とメッセージベクタ化と組み合わせることによって、ループ実行における同期数、通信オーバーヘッドをさらに減少させることができる。

communication dependence vector[7] を使用すると、標準ループはシンプルな参照のため、ループ $i_1..i_{n-1}$について通信なしと判断できる。一般に、n 重の標準ループにおける最内ループにおける通信はベクタ化可能である。図 9は、標準ループにおいて $n = 2, C1 > 0$ の場合にメッ

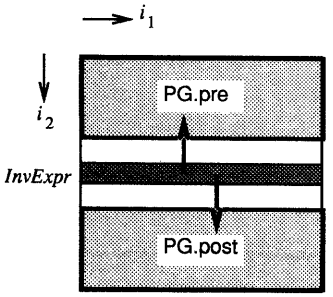

図 9: 最内ループメッセージのベクタ化

セージベクタ化をした場合の様子である。グレイの帯がベクタ化されたメッセージである。

5 まとめ

本稿では dynamic dependence が存在する時に versioning をせずに、最内ループパイプライン化による並列化によって、通信ライブラリ呼びだし回数減少、ループ内通信同期削除、プロセッサガード削除が実現された。またライブラリ実行時間についても、標準ループについて、特に dynamic dependence の原因となる配列次元を分割するプロセッサ数が多い場合、ループ並列化されないループでは実行時ライブラリの処理時間 $Time_s, Time_r$ が $O(UB_n)$ であったのに対し、最内ループパイプライン化後では UB_n によらない一定実行時間が実現された。

メッセージベクタ化との組合せで、さらに同期単位を大きくし、並列度をあげることも可能であることを示した。

現在この技術を TRL HPF コンパイラ [10, 11, 12, 7] に実装しており、実際のアプリケーション等で効果を評価する予定である。

謝辞

本研究にあたり、日本 IBM 東京基礎研究所の、中谷登志男氏、郷田修氏、大澤暁氏、菅沼俊男氏、石崎一明氏に感謝致します。

参考文献

[1] Philip J.Hatcher and Michael J.quinn. *Data-Parallel Programming on MIMD Computers*. The MIT Press, 1991.

[2] Charles H.Kowlbel, David B.Loveman, Robert S.Schreiber, Guy L.Steele Jr., and Mary E.Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[3] Chau-Wen Tseng. An Optimizing FORTRAN D Compiler for MIMD Distributed Memory Machines. Technical Report CRPC-TR93291-S, Rice University, Jan 1993.

[4] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines. Technical Report CRPC-TR91162, Rice University, Aug 1991.

[5] Dror E. Maydan. *Accurate Analysis of Array References*. PhD thesis, Stanford University, September 1992.

[6] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M.J.Wolfe. Dependence graphs and compiler optimizations. In *Eight ACM Symposium on the Principles of Programming Languages*, January 1981.

[7] 石崎一明, 小松秀昭. 分散並列計算機のための並列性抽出法. 電子情報通信学会技術研究報告, Vol. 94, No. 163, pp. 97–104, 1994.

[8] Mark Byler, James R.B.Davies, Christopher Huson, Bruce Leasure, and Michael Wolfe. Multiple version loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, New York, August 1987.

[9] Jingke Li and Marina Chen. Compiling communication-efficient programs for massively parallel machines. *IEEE Transactions on Parallel and Distributed Systems*, Vol. 2, No. 3, pp. 361–376, July 1991.

[10] 中谷登志男. Compiling HPF for A Cluster of Workstations. 並列処理シンポジウム JSPP'93, pp. 1–6, 5 月 1993.

[11] 大澤暁, 小松秀昭. ワークステーション・クラスタにおける動的なデータ交信／実行管理方法. 情報処理学会第 48 回全国大会講演論文集 (4), pp. 73–74, 3 月 1994.

[12] 菅沼俊夫, 小松秀昭. マルチプロセッサシステムにおけるリダクションオペレーション. 情報処理学会第 48 回全国大会講演論文集 (5), pp. 69–70, 3 月 1994.