

分散メモリ並列計算機のためのコンパイラによる 通信遅延隠蔽アルゴリズム

石 崎 一 明[†] 小 松 秀 昭[†]

本論文では、分散メモリ並列計算機において通信と計算のオーバーラップによって通信遅延を隠蔽するコンパイラのアルゴリズムについて述べる。本アルゴリズムでは、通信解析の結果と配列の添字式から得られる reuse 情報を用いて、tiling と loop interchange によるループ変換をコンパイラが行う。生成された tile 単位の通信と計算をオーバーラップするための通信生成方法を述べる。さらに、我々の HPF コンパイラに実装し、RISC システム/6000 SP 上で行った実験によってその有効性を示す。

A Compiler Algorithm for Overlapping Communication with Computation on Distributed Memory Machines

KAZUAKI ISHIZAKI[†] and HIDEAKI KOMATSU[†]

This paper presents a compiler algorithm to hide communication overhead by overlapping communication with computation on distributed memory machines. The algorithm applies loop transformations, such as tiling and loop interchange, based on the result of communication analysis and the reuse information of the operands obtained by subscript analysis. The paper also presents a method of generating communication to overlap communication with computation of tiles. The algorithm presented here has been implemented in our HPF compiler, and experimental results have shown its effectiveness on the RISC System/6000 SP.

1. はじめに

分散メモリ並列計算機では、自動並列化コンパイラ^{1)~4)}により、ユーザは分散メモリ並列計算機にかかわるプロセッサ間通信を考慮することなくグローバルアドレス空間上でプログラミング可能である。分散メモリ並列計算機においてプロセッサ間通信はオーバーヘッドが大きいので、コンパイラによる通信のベクトル化は必須である。

プログラム中の1つのループに着目して、通信のベクトル化を適用したとき、ループ前後の通信時間+ループ実行時間が消費される。通信時間は、1) ネットワークを経由する転送時間、2) ネットワークとメッセージバッファ間の転送時間の2つからなる。

1) の時間を隠蔽するために、通信と計算をオーバーラップする最適化が提案されている⁵⁾。また、ネットワークとメッセージバッファ間の直接 DMA 転送が可能ならば、2) の時間も隠蔽されるので、より効果的

に通信と計算のオーバーラップ可能である⁶⁾。

ソフトウェアから見ると、従来提案されている通信と計算のオーバーラップは、ユーザが直接通信を記述することによるものが多い^{11),13)}。

本論文では、コンパイラが自動的に通信と計算をオーバーラップするためのループ変形を行うアルゴリズムを提案する。これにより、High Performance Fortran (HPF)⁷⁾のようにユーザが通信を記述しない言語においても、通信と計算のオーバーラップを可能にする。具体的には、通信解析の結果と配列アクセスの reuse 情報を用いて、通信と計算のオーバーラップを行うための tiling⁸⁾と loop interchange⁸⁾によるループ変換を行う。さらに、tiling を適用して生成されたループに関する通信生成方法を述べる。最後に、コンパイラに実装しその効果を示す。

従来、tiling は uniprocessor においてデータの locality を改善するために用いられている。つまり、最内ループで同一データの reuse が発生するように、ループ変換を行う。分散メモリ並列計算機において同様の変換を行った場合、最内ループで通信をともしないうメモリアクセスが発生すると、同一メモリ領域への通信が

[†] 日本アイ・ビー・エム株式会社東京基礎研究所
IBM Tokyo Research Laboratory, IBM Japan Ltd.

複数回発生する。本アルゴリズムでは、reuseがあるループの外側で tiling を適用し、tile ループの前に通信を生成する。これによって、オーバーラップのための通信が同一領域へ複数回行われなくようにする。

以下、2章で関連研究を述べる。3章で分散メモリ並列計算機における reuse の利用方法について述べる。4章では、通信と計算をオーバーラップするためのループ変換のアルゴリズムを述べ、5章では、通信と計算をオーバーラップするための通信生成方法を述べる。6章ではプログラムを実行した結果を示す。7章でまとめを述べる。

2. 関連研究

従来行われてきた通信と計算のオーバーラップに関する研究を以下に示す。

- (1) true dependence があるループネストで、イタレーション空間を tile 単位に分割して wavefront 法で実行し、通信と計算をオーバーラップする方法^{9),10)}。これらは tile size の決定方法を議論している。
- (2) 行列積や SCG 法のように prefetch communication が発生するループにおいて、次に実行されるブロックの通信を計算とオーバーラップする方法¹¹⁾。ここでは、ユーザによって最適化されたプログラムに関する性能を議論している。
- (3) Jacobi 法のように stencil communication が発生する場合、計算にともなって通信が発生するループと、通信不要なループに分割して、通信と計算をオーバーラップする方法^{2),12)}。ここでは forall 文、または配列の分割が1次元のみなど、限られた場合のコンパイラによるコード生成方法を述べている。
- (4) ガウスの消去法に対して、次ピボット選択のための通信と行列消去の計算をオーバーラップする方法¹³⁾。ここでは、ユーザがクリティカルパスを短縮するために、通信と計算をオーバーラップする最適化方法を述べている。
- (5) データの生産者と消費者に着目して、データの send と receive の位置を移動し、不必要な通信を消去し、通信遅延を隠蔽するコンパイラのフレームワーク¹⁴⁾。ここでは、主に不必要な通信の消去に重点を置き、実際の効果を示していない。

(1)~(4)の研究では、コンパイラに実装可能なオーバーラップのためのループ変換方法を議論していない。

本論文では、上記の(1)と(2)の場合において、コ

ンパイラによって通信と計算のオーバーラップの最適化を行うループ変形アルゴリズムを提案する。我々のアルゴリズムを適用した結果得られるループは、従来の論文^{9)~11)}で示されたものと同じである。本論文ではコンパイラによる最適化アルゴリズムを提案・実装することにより、従来通信を直接記述する言語で行われていたオーバーラップによる最適化を、HPF 等ユーザが通信を記述しない言語において適用可能にする。

3. 通信遅延隠蔽のための reuse 情報の利用

分散メモリ並列計算機において、通信と計算のオーバーラップをするための tiling を適用する場合、reuse 情報を考慮する必要がある。なぜならば、reuse が存在するループインデックスで tiling を適用したとき、その値の変化によって通信が発生するオペランドの通信が、不要な同一配列領域への転送を複数回行うためである。たとえば、図2で、ループインデックス変数 K の内側で I と J に tiling を適用し、tile ループの前で通信を生成すると、同一領域への不要な通信が複数回行われる。

本章では、tiling を適用するループインデックスを決定する際に利用する reuse について分類し、分散メモリ並列計算機において、上記のような変換を避ける通信と計算のオーバーラップを行う tiling 方法を述べる。

3.1 ループ表現

まず、reuse をベクタを用いて表現するために、プログラム中のループネストを以下のように表現する。
定義 1: 図1において、ループネストの深さを n 、配列 l と r の次元数を m とする。 $\vec{i} = (i_1, \dots, i_n)$ はループインデックスを示し、 $\vec{l}$, $\vec{a}$ はそれぞれイタレーションの実行範囲の下限と上限を示す n 次元のベクタである。関数 $f(\vec{i}) = H\vec{i} + \vec{a}$ は n 次元のループインデックス $\vec{i}$ から、 m 次元の配列インデックス $\vec{j} = (j_1, \dots, j_m)$ への写像を表す関数である。 H は整数要素を持つ行列で、 $\vec{a}$ は整数要素からなるベクタである。関数 $g(\vec{j})$ は、 m 次元の配列インデックス $\vec{j}$ から、プログラムを実行するプロセッサ集合 P の要素であるプロセッサインデックス $\vec{p}$ への写像を表す関数である。

```
DO  $\vec{i} = \vec{l}$  TO  $\vec{u}$ 
  1 ( $f_r(\vec{i})$ ) =  $x(f_r(\vec{i}))$ 
ENDDO
```

図1 ループ表現

Fig. 1 Representation of the loop nest.

```

REAL (8,8) :: A, B, C
*HPFS PROCESSORS P(2)
*HPFS DISTRIBUTE (*,BLOCK) onto P :: A, B, C
DO 10 J = 1, 8
  DO 10 I = 1, 8
    DO 10 K = 1, 8
      A(I, J) = A(I, J) + B(I, K) * C(K, J)
    
```

図2 プログラム例

Fig. 2 Example of the program.

図2の配列Aでは、 $f(\vec{i}) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \vec{i}$, $g(\vec{j}) = \left(\left\lfloor \frac{j_2 - 1}{4} \right\rfloor \right)$ である。

3.2 reuse の分類

ループネスト内で2回以上同じデータがアクセスされる時 reuse があるという。Wolfe は reuse を以下のように分類している⁸⁾。reuse には、同じデータのアクセスによって発生する temporal reuse と、同一キャッシュライン上のデータの参照によって発生する spatial reuse がある。また、異なるイタレーションで同一オペランドの参照によって発生する self reuse と、異なるオペランドのアクセスによって発生する group reuse がある。この結果、reuse を以下の4種類に分類している。

- Self-temporal Reuse：同一メモリ位置のデータを同じオペランドがアクセスする。
- Self-spatial Reuse：同一キャッシュラインのデータを同じオペランドがアクセスする。
- Group-temporal Reuse：同一メモリ位置のデータを異なるオペランドがアクセスする。
- Group-spatial Reuse：同一キャッシュラインのデータを異なるオペランドがアクセスする。

3.3 temporal reuse

分散メモリ並列計算機では、 $A(I)$ と $A(I+1)$ のようにプログラム上連続する配列要素であっても、メモリの配置上連続する保証がない。配列の分散方法を考慮しなければならないためである。配列の分散方法を考慮して spatial reuse を一般的に扱うのは困難であるので、ここでは temporal reuse のみ議論する。

3.3.1 Self-temporal Reuse

self-temporal reuse は同じオペランドが異なるイタレーション $\vec{i}_1, \vec{i}_2$ において同じデータをアクセスする、つまり $H\vec{i}_1 + \vec{a} = H\vec{i}_2 + \vec{a}$, $H(\vec{i}_1 - \vec{i}_2) = 0$ と表現可能である。この方程式の解は $\ker H$ であり、self-temporal reuse vector space R_{ST} と呼ばれる。図2のプログラムで、配列Bの self-temporal reuse vector space は $\text{span}\{\vec{e}_1\}$ である。 $\text{span}\{\vec{e}_1\}$ は1次

元目の単位ベクタを表す。

uniprocessor では、self-temporal reuse を利用するために R_{ST} が0であるループインデックスを最内ループネストとする loop interchange を行う。最内ループネストに移動されたループインデックスに対して tiling を適用する¹⁵⁾。このループ変換により、内側のループネストで同一データがアクセスされるので reuse される可能性が高くなり、性能が改善される。

3.3.2 Group-temporal Reuse

group-temporal reuse は、異なるオペランドの添字式 $f_1(\vec{i}) = H\vec{i} + \vec{a}_1$ と $f_2(\vec{i}) = H\vec{i} + \vec{a}_2$ によって、異なるイタレーション $\vec{i}_1, \vec{i}_2$ で同じデータをアクセスする、つまり、 $H\vec{i}_1 + \vec{a}_1 = H\vec{i}_2 + \vec{a}_2$, $H(\vec{i}_1 - \vec{i}_2) = H\vec{r} = \vec{a}_2 - \vec{a}_1$ と表現可能である。この $\vec{r}$ が存在するかどうか、部分解 $\vec{r}_p$ を求める。一般解は $\ker H + \vec{r}_p$ であり、group-temporal reuse vector space R_{GT} と呼ばれている。uniprocessor では、self-temporal reuse と同様に R_{GT} が0であるループインデックスを最内ループネストへ移動する loop interchange を行う。最内ループネストへ移動したループインデックに対して tiling を適用する¹⁵⁾。この結果メモリアクセスの回数が減少し、性能が改善される。

3.4 通信遅延隠蔽のための reuse 情報の利用

分散メモリ並列計算機において、uniprocessor の場合と同様の条件で通信と計算をオーバラップするために tiling を行うと仮定する。このとき、tiling を適用したループインデックスにおいて通信が発生するオペランドに関する通信は、同一配列領域への不要な転送を複数回行う。

したがって、並列分散メモリ計算機において self-temporal reuse R_{ST} と group-temporal reuse R_{GT} を利用して通信と計算をオーバラップする tiling を行うとき、tile ループの直前で行う通信が同一配列領域へ複数回転送することを防ぐために、条件1を設定する。

条件1：オーバラップのための tiling 可能条件は、インデックス変数の値の変化により通信が発生するループインデックスのうち、self-temporal reuse と group-temporal reuse のないインデックスである。

さらに、並列分散メモリ計算機において異なるオペランドが同一データをアクセスする group-temporal reuse を利用するためには、複数のオペランドの通信にわたり同一領域を転送する通信を一括化する message coalescing²⁾を適用する。

4. ループ変換アルゴリズム

本章では、通信と計算をオーバラップするための tiling と loop interchange を用いたループ変換アルゴリズムを述べる。

アルゴリズムの概要は以下のとおりである。まず、通信情報と reuse 情報から条件 1 を満たすループインデックスを求めて、tiling を適用する候補インデックスとする。次に、並列性を引き出すためにループの実行方法を考慮し、実際に tiling を適用するループインデックスを決定する。最後に、loop interchange によって外側にループインデックスを移動し、tiling を適用する。このとき、元のループ前後で通信を一括で行う代わりに、tile ループを単位とした小さな通信に分割して行う。

アルゴリズムは、以下のステップからなる。

step 1: 配列の通信を求めるための、各プロセッサが代入を実行するループインデックス空間である Local Iteration Set²⁾と、通信を必要とする配列領域である In Set²⁾の計算

step 2: 配列に関する通信の有無を、ループインデックスの次元で表現した communication vector¹⁶⁾の計算

step 3: R_{ST} と R_{GT} を用いた tiling 適用可能なループインデックスの決定

step 4: ループの実行方法を考慮した tiling 適用ループインデックスの選択とループネストの外側に移動する loop interchange

図 3 に、step 1~3 によって tiling 適用可能なループインデックスを決定するアルゴリズムを示す。最後に得られた、tiling vector $\vec{t}$ が非零の次元に対応するループインデックスに対して tiling を適用可能である。また、4.4 節で述べる loop interchange 可能な最外ループインデックスが top に得られる。

以下、図 2 のプログラムを例にアルゴリズムを述べる。図 3 のアルゴリズムに入力されるパラメータは、図 4 のとおりである。 D はループネスト内に存在する data dependence vector の集合である。

4.1 In Set の計算 (step 1)

本節では通信を必要とする配列領域を表す In Set を求める。

まず、元のループのイタレーション空間のうち、各プロセッサ $\vec{p}$ で実行するイタレーション空間である Local Iteration Set $Q(\vec{p})$ を求める。owner-computes-rule を適用するとプロセッサ $\vec{p}$ の Local Iteration Set $Q(\vec{p})$ は、元のループのイタレーション空間 $(\vec{l} : \vec{u})$

と代入文の左辺の配列 l の添字式 $f_l(\vec{i})$ によってアクセスされる配列の領域のうち、プロセッサ $\vec{p}$ に属する配列領域をアクセスするイタレーションの集合で表される。

次に Local Iteration Set $Q(\vec{p})$ を実行する際に右辺の配列 r の添字式 $f_r(\vec{i})$ にアクセスされる配列領域のうち、プロセッサ $\vec{p}$ が持たない配列領域 In Set $Y(\vec{p})$ を求める。

本論文では owner-computes-rule を仮定しているが、以下のステップは特定の計算分割方法に依存しない。図 3 の $Q(\vec{p})$ 、 $Y(\vec{p})$ を求める式を変えることによって、同一イタレーションを同一 PE で実行する等の計算分割方法を適用可能である。

例：図 2 のオペランド $A(I, J)$ について $f_l(\vec{i}) = \begin{pmatrix} 0 & 1 & 0 \\ 1 & 0 & 0 \end{pmatrix} \vec{i}$, $g_l(\vec{j}) = \left(\left\lfloor \frac{j_2 - 1}{4} \right\rfloor \right)$, オペランド $B(I, K)$ に関して関数 $f_r(\vec{i}) = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \vec{i}$, $g_r(\vec{j}) = \left(\left\lfloor \frac{j_2 - 1}{4} \right\rfloor \right)$ である。

In Set の解析は右辺のすべてのオペランドに対して行う。以下では紙面の都合上、通信が発生する配列 B の結果のみを示す。よって、図 2 において $r = R = \{B(I, K)\}$ と考える。In Set は図 5 のとおりである。

4.2 communication vector の計算 (step 2)

配列領域ベクタで表されている通信情報を、ループインデックスベクタで表された dependence vector や reuse vector と演算するために、ループインデックスベクタで表す communication vector を定義する。In Set のある配列次元に着目した場合、通信が発生することは、各プロセッサから異なる配列領域を読み出すこと、によって表される。配列領域における通信の有無をループインデックス空間へ写像するために、ループインデックスベクタから配列領域ベクタへの写像を行う H_r の転置行列を用いる。その後、右辺の配列をアクセスしないループインデックスがある場合、 H_r の転置行列では写像不可能な通信情報を付加する。この結果が、通信情報をループインデックスで表したベクタである。

定義 2: communication vector $\vec{b}$ は、ループインデックスの各次元において各要素が、通信がある場合非零、ない場合零、の値をとるベクタである。

通信が発生させるデータ依存の種類を表すために、まずデータ依存に関するベクタを定義し、その後プロセッサ通信をデータ依存の種類によって分類したベクタを定義する。

algorithm

IN : ($\vec{l}$: set of loop index vector, /* lower bound */
 $\vec{u}$: set of loop index vector, /* upper bound */
 l : left hand side operand,
 R : set of right hand side operand,
 F : set of access function for operands,
 G : set of distribution function for operands,
 P : set of processor index vector,
 D : set of dependence vector,
 n : integer /* depth of a loop nest */
OUT : ($\vec{l}$: index vector /* tiling vector */,
top : integer)

$i, m, inner$: integer;
is_comm : boolean;
 $\vec{l}, \vec{d}$: loop index vector; /* $\vec{d}$ is dependence vector */
 $\vec{y}, \vec{z}$: array index vector;
 $Q(P)$: set of loop index vector; /* Local Iteration Set */
 r : right hand side operand; /* ex. B(I,K) */
 $Y_r(P)$: set of array index vector for operand r ; /* In Set */
 $\vec{b}_r$: loop index vector for operand r ; /* comm. vector */
 $f_r(\cdot)$: access function for operand r ; /* $\in F$ */
 H_r : mapping matrix for operand r ; /* $\in f_r(\cdot)$ */
 $g_r(\cdot)$: distribution function for operand r ; /* $\in G$ */

/* step 1: calculate In Set */

foreach $\vec{p} \in P$ do
 $Q(\vec{p}) := \emptyset$;
foreach $\vec{l} \in [\vec{l}; \vec{u}]$ do
if $g_l(f_l(\vec{l})) = \vec{p}$ then $Q(\vec{p}) := Q(\vec{p}) \cup \vec{l}$;
end foreach
end foreach
is_comm := false;
foreach $r \in R$ do
foreach $\vec{p} \in P$ do
 $Y_r(\vec{p}) := \emptyset$;
foreach $\vec{l} \in Q(\vec{p})$ do
if $g_r(f_r(\vec{l})) \neq \vec{p}$ then $Y_r(\vec{p}) := Y_r(\vec{p}) \cup f_r(\vec{l})$;
end foreach
if $Y_r(\vec{p}) \neq \emptyset$ then is_comm := true;
end foreach
end foreach
if not is_comm then return (0, 0);

/* step 2: calculate Communication Vector */

foreach $r \in R$ do
/* check communication in array dimension */
 $\vec{z} := 0$;
 $m = \text{rank of operand } r$;

for $i := 1$ to m do

if $\left(\bigcap_{\vec{p} \in P} \left\{ y_i | \vec{y} = (y_1, \dots, y_m), \vec{y} \in Y_r(\vec{p}) \right\} \right) = \emptyset$ then

$\vec{z} = \vec{z} \cup \text{span}\{\vec{e}_i\}$

end if

end for

/* generate communication vector */

$\vec{b}_r := H_r^T \vec{z}$;

if $\vec{b}_r \neq 0$ then

for $i := 1$ to n do

if $\exists \text{span}\{\vec{e}_i\} \in \ker f_r$ then $\vec{b}_r = \vec{b}_r \cup \text{span}\{\vec{e}_i\}$;

end foreach

endif

/* check condition 1 */

if $\exists \vec{d} \in D$ ($\vec{d}$ is anti dependence by operand r and $\vec{d} \cap \vec{b}_r \neq \emptyset$) and

$\exists \vec{d} \in D$ ($\vec{d}$ is true dependence by operand r and $\vec{d} \cap \vec{b}_r \neq \emptyset$) then

return (0, 0);

end if

end foreach

/* step 3: calculate $\vec{l}$ */

$\vec{l} := 0$;

foreach $r \in R$ do

for $i := 1$ to n do

if $(\ker f_r \cap \text{span}\{\vec{e}_i\} = \emptyset)$ and $((\ker f_r + \vec{b}_r) \cap \text{span}\{\vec{e}_i\} = \emptyset)$

and $(\vec{b}_r \cap \text{span}\{\vec{e}_i\} \neq \emptyset)$ then

$\vec{l} := \vec{l} \cup \text{span}\{\vec{e}_i\}$; /* check condition 2 */

end if

end foreach

end foreach

if $\vec{l} = 0$ then return (0, 0);

/* find fully permutable nest from top to n */

top := 1;

if $D = \emptyset$ then return ($\vec{l}$, top);

for inner := n to 1 by -1 if $t_{inner} \neq 0$ then break;

while (top < inner) do

if $\forall \vec{d} \in D: \left(\begin{array}{l} (d_1, \dots, d_{top-1}) \succ 0 \text{ or} \\ \forall top \leq i \leq n: d_i \geq 0 \end{array} \right)$ then break;

top := top + 1;

end while

if (top = inner) return (0, 0);

for $i := 1$ to top-1 do $t_i := 0$; /* reset invalid elements */

return($\vec{l}$, top);

図3 tiling 可能なループ決定アルゴリズム

Fig. 3 Algorithm to detect tilable loop indices.

$$n=3, \bar{u}=(8,8,8), P=\{(0),(1)\}, D=\emptyset$$

$$I=\{A(I,J)\}, R=\{A(I,J), B(I,K), C(K,J)\}$$

図4 アルゴリズムに入力されるパラメータ
Fig. 4 Parameters for our algorithm.

$$Q((0))=(1,4,1,8,1,8), Q((1))=(5,8,1,8,1,8)$$

$$Y_r((0))=(1,8,5,8), Y_r((1))=(1,8,1,4)$$

図5 In Set の結果
Fig. 5 Result of In Set.

$$D=D_t \cup D_a$$

$$D_t = \{\vec{d}_t \mid \vec{d}_t = \vec{i}' - \vec{i}, \text{ true data dependence from } \vec{i} \text{ to } \vec{i}'\}$$

$$D_a = \{\vec{d}_a \mid \vec{d}_a = \vec{i}' - \vec{i}, \text{ anti data dependence from } \vec{i} \text{ to } \vec{i}'\}$$

$$\vec{d}_t = (d_{t1}, \dots, d_{tn}), \vec{d}_a = (d_{a1}, \dots, d_{an})$$

$$d_i = [d_i^{\min}, d_i^{\max}], d_i^{\min} \in \mathbb{Z} \cup -\infty, d_i^{\max} \in \mathbb{Z} \cup \infty$$

図6 true data dependence vector と anti data dependence vector の定義

Fig. 6 Definition of true and anti data dependence vector.

定義 3: オペランド間に存在する true dependence によって制約されるイタレーション間の依存距離 $\vec{d}_t$, anti dependence によって制約されるイタレーション間の依存距離 $\vec{d}_a$ と, ループネスト内の dependence vector の集合 D を図 6 のように定義する。

定義 4: 通信によって制約されるイタレーション間の依存距離を communication dependence vector と定義する。communication dependence vector は, anti communication dependence vector $\vec{c}_a$ と true communication dependence vector $\vec{c}_t$ の 2 つからなり, それぞれ, $\vec{c}_a = \vec{b} \cap \vec{d}_a$, $\vec{c}_t = \vec{b} \cap \vec{d}_t$ で求められる。

このとき, 条件 2 を満たすならばオペランドに関する通信をベクトル化可能である¹⁶⁾。

条件 2: オペランドに関する通信をベクトル化可能の条件は以下の 1 つのみを満たすことである。

- (1) オペランドが $\vec{c}_a$ のみを持つ。このとき prefetch communication が発生する。
- (2) オペランドが $\vec{c}_t$ のみを持つ。このとき pipeline communication が発生する。
- (3) オペランドにデータ依存関係が存在しない。

もしベクトル化できない通信が存在するならば, 通信と計算のオーバーラップを行うことはできない。なぜならば, この最適化はベクトル化された通信を分割して計算とオーバーラップするためである。

例: 配列 B の In Set に関して, 1 次元目は各プロセッサにおいて通信領域に重複があり, 2 次元目は各プロセッサで異なる領域を読み込むので, 配列 B の 2 次元目で通信が発生する。

配列 B の 2 次元目をループインデックス変数 K でアクセスすることは, 図 3 では $\vec{b}_r = \begin{pmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}^T \text{span}\{\vec{e}_2\} = \text{span}\{\vec{e}_1, \vec{e}_3\}$ で示される。また, $D = \emptyset$ であるので条件 2 を満たす。

4.3 tiling 適用可能なループの決定 (step 3)

条件 2 を満たす通信が発生し reuse が無いループインデックスを求めるために, self-temporal reuse vector R_{ST} と group-temporal reuse vector R_{GT} が零で, 4.2 節で求めた communication vector $\vec{b}$ が非零の次元を求める。このループインデックスのうち fully permutable⁸⁾ なループインデックスに対して tiling を適用可能と決定する。

例: $R_{ST} = \text{span}\{\vec{e}_1\}$, $R_{GT} = \emptyset$ であり, $\vec{b}_r = \text{span}\{\vec{e}_1, \vec{e}_3\}$ と対応をとると $\vec{t} = \text{span}\{\vec{e}_3\}$ となる。 $D = \emptyset$ より, ループインデックスが top = 1 から inner = 3 まで fully permutable である。この結果, 図 3 より, $\vec{t} = \text{span}\{\vec{e}_3\}$, top = 1 を得る。

4.4 tiling 適用ループの選択とループ変形

(step 4)

ループ内の代入文を実行する際に発生する通信を条件 2 を用いて分類した結果を用いて, ループネストを定義 5 によって分類する¹⁶⁾。

定義 5: 並列実行可能なループを以下のように分類する。

DO PARALLEL: prefetch communication のみが発生するループネスト。

DO PIPELINE: 少なくとも pipeline communication が 1 つ発生するループネスト。

以下, 4.3 節で決定した tiling 適用可能なループインデックスから, ループの実行方法に基づいて tiling を実際に適用するループインデックスの選択方法を示す。その後, tile ループにおいて発生する通信を可能な限りループネストの外側で行うために, 選択したインデックスを fully permutable なループインデックスの範囲内で loop interchange によって移動する。その後, 実際に tiling を適用する。

4.4.1 DO PARALLEL ループネストの場合

prefetch communication のみが発生するループネストでは, ループ直前での通信終了後ループネストを同期なしで実行可能である。ループインデックスの選択方法によって並列度は変化しないので, すべてのループインデックスに対して tiling を適用する。最適な tile size は, 通信のオーバーヘッド時間と tile の計算時間を 1:1 にするところである。tile のサイズが小さ

くなるとプロセッサ内のデータの locality が失われるので、ある程度の粒度をとる必要がある。一般に tile を大きくすると、ループ実行中のプロセッサ間通信の数が減少し通信遅延を隠蔽しやすい。

4.4.2 DO PIPELINE ループネストの場合

pipeline communication が発生する場合、wavefront 法による実行が可能である。並列性を引き出すためには wavefront 面となる tile の数を増やし、通信と計算をオーバーラップする。wavefront 面を増やすためには、分散数が少ない配列次元の計算単位を積極的に縮小する必要がある。よって、配列を分散するプロセッサ数が少ない配列次元をアクセスするループインデックスに対して tiling を適用する。tile を大きくすると通信オーバーヘッドが減るが tile 数も減少し並列度が小さくなる。一方、tile を小さくすると並列度は増加するが、通信オーバーヘッドが増加する。通信や計算時間等のマシンパラメータから、最適な tile size を決定する方法は、文献 9), 10) 等で議論されている。

例: $D = \emptyset$ であるので pipeline communication を含まない。よって、DO PARALLEL ループネストの場合の方法に基づく。したがって、ループインデックス変数 K に tiling を適用する。

このループインデックスと、4.3 節で得られた loop interchange 可能である一番外側のループインデックスを loop interchange する。最後に、一番外側に移動したループインデックス K に tiling を適用する。

5. コード生成

本章では、通信と計算をオーバーラップするためのコード生成について、オペランドの通信の種類が prefetch communication の場合、pipeline communication の場合に分けて述べる。

以下、send/recv は non-blocking send/non-blocking receive の発行を意味する。また wait は、メッセージがすべて到着したことを確認後、処理を次に進める関数である。

5.1 prefetch communication のオーバーラップ

prefetch communication 可能と判断されたオペランドでは、ループ実行前に定義されたデータがループ内で参照される。したがって、任意の時点で参照されるデータを通信可能である。

tile 単位で通信と計算をオーバーラップするために、tile ループの直前でイタレーションの実行順に従って次の tile で参照されるデータを別バッファへ送信し、tile の計算を行う。次の tile を実行する直前に、先行して送信したデータをすべて受信したことを確認する

```
DO 10 K=1,8,KB
  if (K=1) send & recv B' for first tile
  wait
  swap B' and B
  if (K<=8-KB) send & recv B' for next tile
  endif
DO 10 KK=K,MIN(K+KB-1,N)
  DO 10 J=1,8/2
    DO 10 I=1,8
      A(I,J)=A(I,J)+B(I,KB)*C(KK,J)
    
```

図 7 prefetch communication のオーバーラップ

Fig. 7 Overlapping computation with prefetch communication.

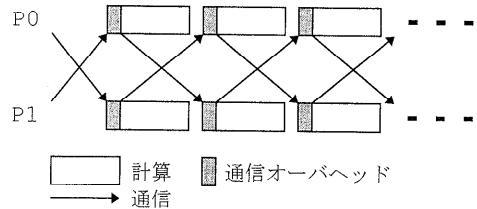


図 8 prefetch communication のオーバーラップの動作

Fig. 8 Execution of overlapping computation with prefetch communication.

ことで、データの送信者と受信者の間で同期をとり、通信バッファと計算バッファを切り替える。この結果、現在の tile の計算と次の tile のための通信をオーバーラップ可能である。

図 2 のプログラムに対して生成されるコードの例を、図 7 に示す。図 8 に生成コードにおける通信と計算の動作を示す。

5.2 pipeline communication のオーバーラップ

pipeline communication 可能と判断されたオペランドでは、あるイタレーションで定義されたデータが、dependence vector に従って後のイタレーションで参照される。よって、データを定義した tile の実行が終わった直後に、データが参照される tile を実行するプロセッサへ送る。その間に次の tile の計算を行い、通信と計算をオーバーラップする。データを参照する tile の実行前にデータを受信したことの確認によってデータの送信者と受信者の間で同期をとる。

プログラムと生成されるコード例を図 9 に示す。図 10 に生成コードにおける通信と計算の動作を示す。

6. 評価

我々の HPF コンパイラ¹⁷⁾にアルゴリズムを実装し、2つのプログラムを実行することで評価を行った。プログラムとして、定義 5 のそれぞれの分類に属する代表的な数値計算を選んだ。1つは行列積を求めるプロ

```

*HPFS DISTRIBUTE      DO 11 J=1,N,JB
*HPF$c A(BLOCK,*)      receive A
                        wait
                        DO 10 JJ=J,MIN(J+JB-1,N)
                        DO 10 I=1,M/n_procs
10  A(I,J)=A(I,J-1) 10  A(I,JJ)=A(I,JJ-1)
                        11 send A
a) ソースコード      b) 生成されるコードイメージ

```

図9 pipeline communication のオーバラップ

Fig. 9 Overlapping computation with pipeline communication.

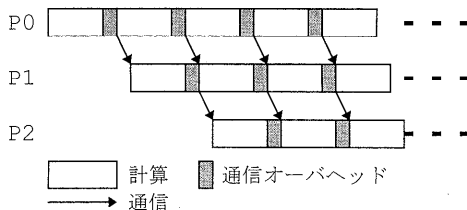


図10 pipeline communication のオーバラップの動作

Fig. 10 Execution of overlapping computation with pipeline communication.

グラムで、DO PARALLEL に分類される。もう1つは Successive Over Relaxation (SOR) 法によって偏微分方程式を解くプログラムで、DO PIPELINE に分類される。

実験は、IBM 社の RISC システム/6000 SP¹⁸⁾ (SP) thin ノードの 32 台構成にハイパフォーマンススイッチ (HPS) を接続し、通信ライブラリは Message Passing Library (MPL)¹⁹⁾ の non-blocking communication のライブラリを使用したシステム上で行った。最適化に関するコンパイルオプションはすべて -O を指定した。

6.1 行列積

行列の大きさは 1600×1600 で、2 次元目を与えられたプロセッサ数で BLOCK 分割した。実行プロセッサ数を 8, 16, 32 台とした実行結果を図 11 に示す。縦軸はループ直前で一括して通信を実行したときを 1 とした速度向上率を表す。横軸は tile size を表す。

6.2 Successive Over Relaxation 法

行列の大きさは 2000×2000 で、1 次元目を与えられたプロセッサ数で BLOCK 分割した。実行プロセッサ数を 9, 16 台とした実行結果を図 12 に示す。縦軸は 1 台で実行したときを 1 とした速度向上率を表し、横軸は tile size を表す。グラフには、プロセッサ構成を 3×3 と 4×4 台で 1 次元目と 2 次元目の両方を BLOCK 分割し、tiling しない場合の性能も示した。

6.3 考察

行列積では、prefetch communication が発生する。このとき、プロセッサ台数が増加するにつれて、広い

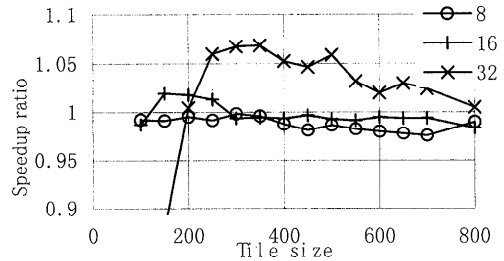


図11 行列積の性能

Fig. 11 Performance of matrix multiply.

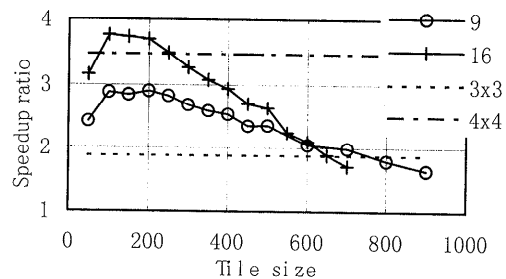


図12 SOR の性能

Fig. 12 Performance of SOR.

範囲の tile size で効果がある。また、速度向上率も大きくなる。

SOR では、prefetch communication と pipeline communication が発生し、wavefront 法による実行が可能である。このとき、コンパイラが通信と計算のオーバラップを適用することによって、ユーザが指定した配列分散で実行した場合よりも性能が向上する。特に、多次元配列において 1 次元しか分割できなかった場合、他に比べ性能向上が非常に大きい。

最適な tile size は、プロセッサ間通信の時間と tile の計算時間の関係から決められる。

行列積のように DO PARALLEL に分類されるループの場合、通信時間が tile の計算時間に完全に隠蔽されるならば、実行時間の中に占める通信時間の割合が非常に小さくなる。つまり最適な tile size は、プロセッサ間通信の時間と tile の計算時間を 1:1 にするところである。

SOR のように DO PIPELINE に分類されるループの場合、tile size を小さくすると wavefront 面が大きくなり並列度が増加し 1 回の通信量が減少するが、通信回数の増加にともないオーバヘッドが増えるので総通信時間が増加する。tile size を大きくすると 1 回の通信量は増えるが、通信回数の減少により通信にかかわるオーバヘッドが減り、総通信時間を減らすことができる。しかし、並列度が減少する。

図 11 の 32 プロセッサの場合、また図 12 では最適な tile size より大きくなると Speedup ratio の低下が顕著である。これは tile size の増加による計算時間の増大によって、プロセッサ間通信の待ちが発生しているためと考えられる。

現在の SP では、ネットワーク経由の転送時間と計算をオーバーラップ可能である。しかし、DMA 転送は HPS とシステム領域間でのみ可能であるので、HPS とユーザメッセージバッファ間の転送時間と計算はオーバーラップできない、non-blocking communication を用いて通信と計算のオーバーラップを行う場合、1) HPS からシステム領域へ転送が終了したときにおきる CPU への割込み、2) システム領域からユーザメッセージバッファへのコピーの 2 つの CPU オーバヘッドが存在する¹⁹⁾。1) は 60 μ s、2) は 28 μ s/KB の時間がかかる。また、これらによってキャッシュメモリへ計算と無関係のデータが読み込まれ、キャッシュメモリによる locality の改善を妨げる。これらのオーバヘッドが存在するにもかかわらず、tiling のサイズを適切に選んだ場合、従来より高い性能を得ることができる。これは、我々のアルゴリズムの有効性を示す。HPS からメッセージバッファへの直接 DMA 転送が実装された場合、上記の 2 つのオーバヘッドがなくなり、よりよい結果が得られることが予想される⁶⁾。

7. ま と め

本論文では、従来ユーザによって行われていた通信遅延を隠蔽するための計算と通信のオーバーラップによる最適化を、コンパイラによって行うアルゴリズムを示した。また、オーバーラップのためのコード生成方法を示した。さらに、アルゴリズムを我々の HPF コンパイラに実装し、その有効性を示した。

謝辞 日頃からご指導、ご助言いただく東京基礎研究所並列分散システムズの中谷登志男氏、寒川光氏、郷田修氏、菅沼俊夫氏、安江俊明氏、小笠原武史氏に感謝いたします。

参 考 文 献

- 1) Stanford SUIF Compiler Group: SUIF: A Parallelizing and Optimizing Research Compiler, Technical Report, Stanford University, CSL-TR-94-620 (1994).
- 2) Tseng, C.W.: An Optimizing Fortran D Compiler for MIMD Distributed-memory Machines, Ph.D. Thesis, Rice University, CRPC-TR93291 (1993).
- 3) Shindo, T., Iwashita, H., Doi, T., Hagiwara, J. and Kaneshiro, S.: HPF Compiler for the AP1000, *Proc. International Conference on Supercomputing*, pp.190-194 (1995).
- 4) 蒲池, 草野, 末広, 妹尾, 田村, 左近, 渡辺, 白戸: HPF 処理系の実現と Cenju-3 での評価, 並列処理シンポジウム JSPP'95, pp.361-368 (1995).
- 5) Eicken, T., Culler, D.E., Goldstein, S.C. and Schauser, K.E.: Active Messages: A Mechanism for Integrated Communication and Computation, *Proc. 19th Annual International Symposium on Computer Architecture*, pp.256-266 (1992).
- 6) 堀江, 小柳, 今村, 林, 清水, 石畑: メッセージ通信の分散メモリ型並列計算機性能への影響, 情報処理学会論文誌, Vol.35, No.4, pp.609-618 (1994).
- 7) High Performance Fortran Forum: High Performance Fortran Language Specification, Version 1.0, Technical Report, Rice University, CRPC-TR92225 (1992).
- 8) Wolfe, M.: *High Performance Compiler for Parallel Computing*, Addison-Wesley (1995).
- 9) Palermo, D.J., Su, E., Chandy, J.A. and Banerjee, P.: Communication Optimizations Used in the PARADIGM Compiler for Distributed-memory Multicomputers, *Proc. 23rd International Conference on Parallel Processing*, pp.11:1-10 (1994).
- 10) Ohta, H., Saito, Y., Kainaga, M. and Ono, H.: Optimal Tile Size Adjustment in Compiling General DOACROSS Loop Nests, *Proc. International Conference on Supercomputing*, pp.270-279 (1995).
- 11) 武本, 松本, 平木: プログラム最適化技法適用下における並列計算機結合形状の性能評価, 並列処理シンポジウム JSPP'94, pp.137-144 (1994).
- 12) Koelbel, C., Mehrotra, P. and Rosendale, J.V.: Supporting Shared Data Structures on Distributed Memory Architectures, *Proc. ACM SIGPLAN '90 Symposium on Principles and Practice of Parallel Programming*, pp.177-186 (1990).
- 13) Hiranandani, S., Kennedy, K. and Tseng, C.W.: Preliminary Experiences with the Fortran D Compiler, *Proc. Supercomputing '93*, pp.338-350 (1993).
- 14) Hanxleden, R. and Kennedy, K.: GIVE-N-TAKE A Balanced Code Placement Framework, *Proc. ACM SIGPLAN '94 Conference on Program Language Design and Implementation*, pp.107-120 (1994).
- 15) Wolfe, M.E. and Lam, M.S.: A Data Locality Optimizing Algorithm, *Proc. ACM SIGPLAN*

'91 Conference on Program Language Design and Implementation, pp.30-44, 1991

- 16) Ishizaki, K. and Komatsu, H.: A Loop Parallelization Algorithm for HPF Compilers, *8th Workshop on Language and Compilers for Parallel Computing*, pp.12.1-15 (1995).
- 17) 郷田, 大澤, 小松, 菅沼, 小笠原, 石崎, 中谷: HPF 処理系の実現と評価, 情報処理研究会報告, HPC57-20, pp.115-120 (1995).
- 18) Agewala, T., Martin, J.L., Mirza, J.H., Sadler, D.C., Dias, D.M. and Snir, M.: SP2 System Architecture, *IBM Systems Journal*, Vol.34, No.2. pp.152-184 (1995).
- 19) Snir, M., Hochschild, P., Fryer, D.D. and Gildea, K.J.: The Communication Software and Parallel Environment of the IBM SP2, *IBM Systems Journal*, Vol.34, No.2. pp.205-221 (1995).

(平成 8 年 9 月 17 日受付)

(平成 9 年 6 月 3 日採録)



研究に従事.



石崎 一明 (正会員)

1992 年早稲田大学大学院理工学研究科修士課程修了. 同年日本 IBM (株) 入社. 以来, 東京基礎研究所において, HPF コンパイラ, Java Just-In-Time コンパイラに関する

小松 秀昭 (正会員)

1985 年早稲田大学大学院理工学研究科電気工学専攻修了. 同年日本 IBM (株) 東京基礎研究所入社. コンパイラ, アーキテクチャの研究に従事.