計算機アーキテクチャ 97－6
（1992. 12. 3）

分散メモリ型並列計算機の自動並列化コンパイラ
―Inspector/Executor アルゴリズムの高速化―

窪田昌史[†]，大野和彦[†]，三吉郁夫[†]
森 眞一郎[†]，中島 浩[†]，富田眞治[†]
†：京都大学 工学部 情報工学教室

我々は，メッセージ交換型の分散メモリ型並列計算機のための自動並列化コンパイラを開発している．本稿では，配列の添字式に配列を含むループのコンパイル技法について述べる．このようなコードに対するコンパイル技法として，Inspector/Executor という戦略がある．本稿では，従来提案されている inspector のアルゴリズムにおいて，高速化を行う上で問題点となるプロセッサ間通信を不要にしたアルゴリズムを２種類提案する．まず最初に提案するアルゴリズムでは，添字式の逆関数を利用してプロセッサ間通信を不要にしている．また，配列の逆関数が求められない場合についても，添字式内の配列を全て検査することで通信を不要とするアルゴリズムも提案する．

Automatic Parallelizing Compiler for Distributed Memory Parallel Computers
—New Algorithms to Improve the Performance of the Inspector/Executor—

Atsushi KUBOTA[†], Kazuhiko OHNO[†], Ikuo MIYOSHI[†],
Shin-ichiro MORI[†], Hiroshi NAKASHIMA[†], Shinji TOMITA[†]

†: Department of Information Science
Faculty of Engineering, Kyoto University
Yoshida-hon-machi, Sakyo-ku, Kyoto 606-01 Japan

E-mail: {kubota, ohno, miyo, moris, nakasima, tomita}@kuis.kyoto-u.ac.jp

We have been constructing an automatic parallelizing compiler for message passing distributed memory computers. In this paper, compilation of loops where arrays with arrays in their subscript functions exist are focused. The Inspector/Executor strategy is a technique to compile such codes. In this paper, we propose two inspector algorithms. During inspector parts, our algorithms don't need inter-processor communications which was the drawback to get the faster codes. The first inspector algorithm proposed here utilizes inverse subscript functions in order to eliminate the inter-processor communications. To cope with the case when the inverse subscript functions cannot be defined, we also propose another algorithm where all arrays in subscript functions are examined to eliminate communications.

1 はじめに

現在我々はデータ並列性を内在する問題のプログラムを対象として，その自動並列化，並列化支援の研究を行っている．

並列計算機の主要なアプリケーションである科学技術計算においては，実行時間の大部分が，行列に対する操作，演算をループによって実行する部分に費やされる．ループ部分は，細粒度で規則性をもつ並列性を内在するため，SPMD(Single Program Multiple Data stream) モデルに基づいて並列化を行うシステムが数多く発表されている [1][2][3][5][6].

SPMD のコードの生成においては，並列性を最大限に引き出せるように，大規模な配列を分割して各プロセッサ (PE) に割付ける必要がある．

しかし，メッセージ交換型の分散メモリ型並列計算機において稼働するプログラムでは，データ空間に共有部分がなく，PE 間のデータの共有はメッセージ通信によって行わなければならない．そのため，メッセージ通信によるオーバーヘッドが，プログラムの性能に対して多大な影響を与える．

それゆえ，データ分割法を決定するには，PE 内の処理量の粒度 (granularity)，PE 間のメッセージ通信量，同期に要する待ち時間，負荷分散などの諸要素を勘案しなければならず，データ分割法を自動的に決定するのは困難である．

そこで我々は，

- 逐次プログラム，並列プログラムのトレースとデータ依存解析によって，最適なデータ分割の決定のための情報をユーザに与える．
- ユーザから与えられたデータ分割情報に基づき，最適な並列化コードを生成する．

という2つのフェーズに分けて研究を進めており，将来はこれらのフェーズを統合して，自動並列化コンパイラを完成させることを目標にしている．

本稿では，後者のコード生成におけるコンパイル技法について議論を行う．ソースプログラムとして対象とするのは,「FORTRAN,C などの手続き型言語で記述された逐次プログラムで，データ分割はディレクティブなどで指定されている」というものである．また，プログラム中のデータ依存関係による並列性の抽出などは，ソースプログラムの解析によって行われ，これらの情報がすでに得られていることを前提とする．

本稿では，2章で，我々が開発中の SPMD コード生成システムにおいて採用している基本方針と，DOALL 型ループ内の不規則なアクセスパターンに対する実行時解析を行う Inspector/Executor の手法について述べる．3章では，inspector の生成に的を絞り，その高速化，生成可能なソースコードについて議論する．最後に，4章で総括を行う．

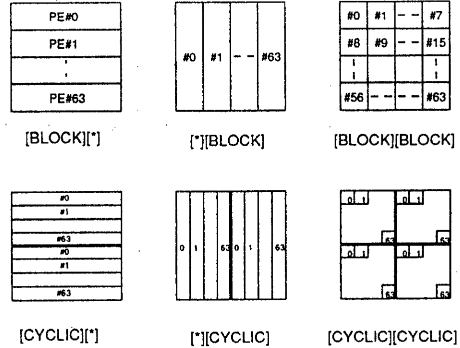

図 1: データ分割法 (プロセッサ数 64)

2 並列化コードの生成

2.1 並列化コードの生成の基本戦略

並列化コードの生成は，

- データ分割に基づく PE へのデータ割り付け
- 割り付けられたデータに対応するコードの生成

という2つのフェーズに分かれる．これを次のような例で説明する．

```
integer a(100),b(100),c(100)
for i=1,100 do
  a(i) = b(i) * c(i);
endfor
```

この DOALL 型ループを，4 PE で並列に実行するコードを生成するには，まず，配列 a,b,c を 25 要素づつ各 PE に割り付け，次に，この分割に従って，ループのイタレーションが 25 個のコードを生成する．

```
integer a(25),b(25),c(25)
for i=1,25 do
  a(i) = b(i) * c(i);
endfor
```

これを，本稿では，イタレーション番号が 1 から 25 までのイタレーションが 25 回繰り返されるというように表現する．

配列の分割法としては，図 1に示すように，BLOCK,CYCLIC 分割が代表的であり，本コンパイラにおいても，当面はこの2つを対象とする．このような規則的な分割法を採用すると，データの PE への割り付けはコンパイル時に行うことができる．コンパイラへの指定は，Fortran D[2] のよう

に，配列に対するディレクティブによって行う．部分ピボッティング付き LU 分解のコードの例を図 2 に示す．

```
integer i,j,k,l,pivot(N)
double precision a(N,N)

decomposition d(N,N)
align a(i,j) with d(i,j)
align pivot(i) with d(i,:)
distribute d(BLOCK)(BLOCK)

for k=1,N do

  /* ピボット行の選択，交換
     (pivot の値が変更される) */

  for i=k+1,N do
    a(pivot(i),k) = a(pivot(i),k)
                  / a(pivot(k),k)
  endfor
  for i=k+1,N do
    for j=k+1,N do
      a(pivot(i),j) = a(pivot(i),j)
        - a(pivot(i),k) * a(pivot(k),j)
    endfor
  endfor
endfor
```

図 2: 部分ピボッティング付き LU 分解のコード

割り付けられたデータに対応するコード生成には，Data Owner computes Rule に基づく手法を採用する．これは，分割された配列データを保持する PE(*owner*) が，そのデータをアクセスする文を実行するようにコードが生成するという戦略である．代入文では，原則として左辺のデータを所有する PE が実行する．

アクセスすべきデータが PE 内にあるか，他の PE とのメッセージ通信によって得られるかの判定は，コンパイル時に行う．その結果，必要であればメッセージ通信のコードが生成される．

しかし，アクセスすべきデータの位置が実行時に決定されるため，コンパイル時の解析だけでは不十分なプログラムも存在する．これらのプログラムに現れる，不規則なアクセスを *irregular computaion* と呼ぶ [4].

部分ピボッティング付きの LU 分解[1]，疎行列の分解など，非常によく使われるプログラムの中にも，irregular computation を含むものが多い．その要因の多くは，配列の添字内の整数型配列によるインデックス参照である．

次節では，このような irregular computation が，DOALL 型ループ内に現れる場合の高速化手法である Inspector/Executor 戦略を用いたコード生成法について述べる．

[1] 以下，LU 分解という場合，特に断らない限り部分ピボッティング付きのものを指す．

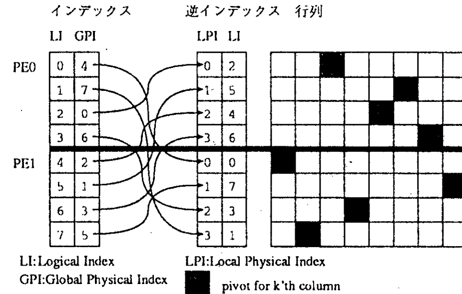

図 3: LU 分解におけるインデックス参照

2.2 Inspector/Executor 戦略

PE 間通信を伴うデータ参照では，データの送信者，受信者の PE が確定しなければならない．インデックス配列の値は実行時に変更されている可能性があるため，インデックス配列を添字に含む配列データへのアクセスが起こると，送受信の両 PE は実行時でなければ確定できない．

図 3は，8 × 8 の行列を PE 2 個で行方向に BLOCK 分割し，インデックス配列も BLOCK 分割した場合の LU 分解を示している．この図から明らかなように，インデックス参照が PE 間通信を伴う場合が存在している．さらに，このインデックス配列は分解の過程で変更されるため，インデックス参照が通信を伴うかどうかは，実行時にならなければ確定できないことがわかる．

実行時のデータアクセスの度に送受信 PE を確定すると，オーバーヘッドが大きい．そのため，DOALL 型ループに対して，先に PE 間通信を伴うデータ参照を検査する inspector を実行し，引き続いて演算そのものを行う executor を実行するというコード生成法が提案されている [5]．以下に，PEp におけるコードの概要を示す．なお，この中で，PEqは，$p \neq q$となる任意の PE である．

1. inspector では，以下のリストが作成される．
   (a) PE 間通信のためのデータ参照関係を示すリスト
      - p がアクセスを必要としているが，q が所有している配列データの変数名とその添字のリスト *recv_list* (p,q)
      - p が所有しており，qがアクセスを必要としている配列データの変数名とその添字のリスト *send_list* (p,q)

(b) executor でのイタレーションの実行順序を決めるためのリスト

- p において，ローカルに割り付けられているデータアクセスのみで実行されるイタレーションのリスト *local_iter*(p)
- p 以外の PE からのデータもアクセスする必要があるイタレーションのリスト *nonlocal_iter*(p)

2. executor では，データの送受信，ループの実行が以下の順で行われる．

(a) *send_list*(p,q) に基づいて，qへデータを送信する．

(b) イタレーション *local_iter*(p) のループを実行する．

(c) *recv_list*(p,q) のデータを qから受信する．

(d) 受信したデータを参照するイタレーション *nonlocal_iter*(p) を実行する．

executor においては，送信すべきデータを先送りし，通信が行われている間に *local_iter*(p) を実行する．ついで必要とするデータを受信し，*nonlocal_iter*(p) を実行する．このように，executor において，ある PEp から qへの通信を一度にまとめて，通信オーバヘッドを低減するとともに，ループのイタレーションの実行順序を変更して，通信のレイテンシの隠蔽を図っている．inspector はこれらの実現のための前処理という役割を持つ．

3 inspector の生成

本章では，inspector の生成について述べる．その準備として，3.1節において，データの参照関係，PE 間通信が行われるデータの定式化を行う．3.2節では，従来の inspector のアルゴリズムを考察し，それを改良した2種のアルゴリズムを提案する．3.3節では，提案したアルゴリズムと従来のアルゴリズムとの比較を行う．さらに，3.4節において，実際のプログラムのコード生成への inspector/executor アルゴリズムの適用について述べ，最後に3.5節で，DOALL 型ループ以外のコードへの Inspecotor/Executor 戦略の拡張の可能性について述べる．

3.1 集合の定義

この節では，[4][5] に沿って，図4に示す並列実行可能なコードについて，添字とイタレーションの集合を定義する．

```
for k⃗=l⃗,m⃗,s⃗
  X(g(k⃗)) = Y1(h1(k⃗))∘Y2(h2(k⃗))··· Yn(hn(k⃗))
endfor
```

図 4: 並列ループ

本稿では，以下，$X(g(\vec{k}))$, $Y_i(h_i(\vec{k}))$ をそれぞれ左辺，右辺の項と呼ぶ．XやY_iと略記することもある．$g(\vec{k}), h_i(\vec{k})$ は，添字内の表現式であり，以下添字式と呼び，g, h_iのように略す場合もある．これらは，イタレーション番号$\vec{k}$から添字を求める関数である．実際には，インデックス配列と呼ぶ整数型配列と，それに$+1, -1$ などの演算を施したものである．また，g, h_iの逆関数を g^{-1}, h_i^{-1}で表す．g, h_i内のインデックス配列に対しても，逆関数の概念を導入し，**逆インデックス配列**と呼ぶ．**添字**という用語は，イタレーション$\vec{k}$を添字式h_iなどに代入して得られる$\vec{j}=h_i(\vec{k})$ を指す．

図4のコードにおいて，$\vec{k}$などのようにベクトル表記となっているのは，多重ループを表現するためである．$\vec{l}$から$\vec{m}$まで，ステップ$\vec{s}$で取り得るイタレーション番号の集合を *IterationSet*$(\vec{k})$ と定義する．また，$\circ$は，右辺の項の和や積などの演算を表す．

データ分割を表す関数は次のようになる．

$$\mu_X(i) = (\delta_X(i), \alpha_X(i)) \quad (1)$$

配列 Xの大域的な添字を与えると，δは，$X(\vec{i})$ を所有する PE 番号を返し，αはその PE 内での添字を返す．配列 Xのサイズを N, PE 数を Pとすると，BLOCK 分割では，

$$\delta_X(i) = \lfloor i/\lceil N/P \rceil \rfloor, \alpha_X(i) = i \bmod P \quad (2)$$

となり，CYCLIC 分割では，

$$\delta_X(i) = i \bmod \lceil N/P \rceil, \alpha_X(i) = \lfloor i/P \rfloor \quad (3)$$

となる[2]．右辺 Y_iについても，同様に定義される．これらの関数は，多次元配列に対しては，それぞれの次元についてのこれらの関数のベクトル関数として定義される．

左辺の配列 Xのうち，PEp に所有される添字の集合は

$$local_X(p) = \{\vec{i} | \delta_X(\vec{i}) = p\} = \delta_X^{-1}(p) \quad (4)$$

である．右辺 Y_jについても，同様に定義される．

代入文の左辺のデータを所有する PE が，その代入文を実行するという Data Owner Computes

[2] これは C 言語流に配列の添字が0から始まる場合である．

Rule の原則を適用すると，上に示したループ中でPEp において実行されるイタレーションの集合は，

$$exec(p) = g^{-1}(local_X(p)) \cap IterationSet(\vec{k}) \quad (5)$$

である．

PEp で実行されるイタレーションで参照される右辺の添字の集合は，

$$ref_{Y_j}(p) = h_j(exec(p)) \quad (6)$$

である．$ref_X(p)$ は，$local, exec$の定義より，

$$ref_X(p) = g(exec(p)) = local_X(p) \quad (7)$$

となる．

また，右辺の表現式について，PEp に所有されているデータを参照するイタレーションの集合は，

$$deref_{Y_j}(p) = h_j^{-1}(local_{Y_j}(p)) \cap IterationSet(\vec{k}) \quad (8)$$

である．全ての右辺の表現式 $deref_{Y_j}$についての積集合を

$$deref(p) = \bigcap_j deref_{Y_j} \quad (9)$$

と定義する．

すると，2.2節で示した，ispector において求めるべき添字，イタレーションの集合は以下の通りになる．

PEp から，PE$q(\neq p)$ に送信すべきデータを表す添字の集合は，

$$send_list_{Y_j}(p,q) = local_{Y_j}(p) \cap ref_{Y_j}(q) \quad (10)$$

であり，PEp が，PE$q(\neq p)$ から受信すべきデータを表す添字の集合は，

$$recv_list_{Y_j}(p,q) = ref_{Y_j}(p) \cap local_{Y_j}(q) \quad (11)$$

となる．

PEp が所有しているデータのみで実行できるイタレーションの集合は，

$$local_iter(p) = exec(p) \cap deref(p) \quad (12)$$

であり，他の PE$p \neq q$が所有するデータも必要とするイタレーションの集合

$$nonlocal_iter(p) = exec(p) - nonlocal_iter(p) \quad (13)$$

となる．

3.2 inspector のアルゴリズム

本節では，図 4に対して生成される inspector のアルゴリズムを述べる．

まず，inspector/executor 戦略が対象とするコードを明確にしておく．対象とするのは DOALL 型のループであり，ループ内で，図 4の g, h_iの値が変更される場合は対象としていない．従って，ループ内で，インデックス配列が変更されるループに対しては生成できない．例えば，図 2の LU 分解では，i のループと，i,jの 2 重ループは対象とするが，外側の kのループは対象としない．

なお，図 4のコードは各イタレーションにおいて代入文が 1 つ存在するだけであるが，複数の文が存在する場合は，これから述べるアルゴリズムをそれぞれの文に対して生成すればよい．

以下では，[4][5] で提案された inspector のアルゴリズムと，今回提案する 2 種のアルゴリズムを示す．各アルゴリズムの評価，ならびに適用範囲の比較については，3.3節で述べる．

3.2.1 従来のアルゴリズム

```
for j⃗ ∈ exec(p) do
  local_flag = true;

  /* recv_list_Yi(p,src) を求める */
  for each Yi do
    src = δ_Yi(h_i(k⃗));
    if (src != p) then
      local_flag = false;
      append h_i(j⃗) to recv_list_Yi(p,src);
  endif endfor

  /* local/nonlocal_iter を求める */
  if (local_flag == true) then
      append j⃗ to local_iter(p);
  else
      append j⃗ to nonlocal_iter(p);
endif endfor

/* recv_list(p,q) を送信する */
for each PE q (≠ p) do
  send recv_list(p,q) to PE q;
endfor

/* recv_list(q,p) を受信する */
for each PE q (≠ p) do
  receive recv_list(q,p) from PE q;
endfor
```

図 5: ALG(1) 従来のアルゴリズム

戦略 $recv_list(p,q)$ は，PEp から qへのデータのアクセス要求のリストであり，これは $send_list(q,p)$ に等しい．$recv_list(p,q)$ を p から qへ送信することで，$send_list(q,p)$ を求

め，これをもとに executor の最初の部分のデータの送信を行う．

処理の概要 以下の処理が行われる．

1. 自 PE が必要とするデータのリスト *recv_list* を作成する．
2. executor のイタレーションの順番を変更するためのリスト $local/nonlocal_iter$ を作成する．
3. 当該データを所有する PE へ要求リストを送信する．
4. 逆に他の PE から要求リストを受信する．

インデックス配列の条件 各 PE が実行するイタレーションの番号 $exec(p)$ を得るための添字式の逆関数 g^{-1}と，それに対応する右辺の添字 $ref_{Y_i}(p)$ が求められること．

3.2.2 逆インデックス配列を活用するアルゴリズム

```
/* send_list_Yi(p,dest) を作成する */
for each Yi do
  for j ∈ deref_Yi(p) do
    dest = δX(g(j));
    if (dest != p) then
      append Yi(j) to send_list_Yi(p,dest);
endif endfor endfor

for j ∈ exec(p) {
  local_flag = true;

  /* recv_list_Yi(p,src) を作成する */
  for each Yi do
    src = Yi(hi(j));
    if (src != p) then
      local_flag = false;
      append Yi(hi(j)) to recv_list_Yi(p,src);
  endif endfor

  /* local/nonlocal_iter を求める */
  if (local_flag == true) then
    append j to local_iter(p);
  else
    append j to nonlocal_iter(p);
endif endfor
```

図 6: ALG(2) 逆インデックス配列を活用するアルゴリズム

戦略 逆インデックス配列を活用することで，*send_list*,*recv_list* を各々の PE が独自に作成する．各 PE は自分の所有するデータに関する全ての参照関係を検査する．従って，通信は不要となる．

処理の概要 以下の処理が行われる．

1. 他の PE が必要としているデータのリスト *send_list* を作成する．
2. 自 PE が必要とするデータのリスト *recv_list* を作成する．
3. executor のイタレーションの順番を変更するためのリスト $local/nonlocal_iter$ を作成する．

インデックス配列の条件 各 PE が実行するイタレーションの番号 $exec(p)$ を得るための添字式の逆関数 g^{-1}と，右辺に現れるデータが参照されるイタレーション番号 $deref_{Y_i}(p)$ を得るための添字式の逆関数 h_i^{-1} が求められること．および，これらのイタレーションに対する添字 $ref_{Y_i}(p)$,$ref_X(p)$ が求められること．

3.2.3 逆インデックス配列が不要なアルゴリズム

```
/* 全てのイタレーションについて */
for j ∈ IterationSet(k) do
  local_flag = true;
  dest = δX(g(j));
  for each rhs Yi do
    src = δYi(hi(j));

    /* send_list_Yi(p,dest) を作成する */
    if (src == p) then
      if (dest != p) then
        append hi(j) to send_list_Yi(p,dest);
      endif

    /* recv_list_Yi(p,src) を作成する */
    else   /* (src != p) */
      local_flag = false;
      if (dest == p) then
        append hi(j) to recv_list_Yi(p,src);
  endif endif endfor

  /* local/nonlocal_iter を求める */
  if (dest == p) then  /*
    if (local_flag == true) then
      append j to local_iter(p);
    else
      append j to nonlocal_iter(p);
endif endif endfor
```

図 7: ALG(3) 逆インデックス配列が不要なアルゴリズム

戦略 各 PE が，全てのイタレーションに対して全データの参照関係を調べ，*send_list*,*recv_list* を作成する．従って，他の PE との通信は不要である．

処理の概要 全てのイタレーションに対して，

1. 他のPEが必要としているデータのリスト $send_list$ を作成する.
2. 自 PE が必要とするデータのリスト $recv_list$ を作成する.
3. executorのイタレーションの順番を変更するためのリスト $local/nonlocal_iter$ にイタレーションを加える.

という処理を繰り返す.

インデックス配列の条件 $ref(p)$ を両辺について全て検査するため,全てのインデックス配列を必要とする.

3.3 inspectorアルゴリズムの比較

表 1: inspectorのアルゴリズムの比較

アルゴリズム		(1)	(2)	(3)
inspector内での通信		必要	不要	不要
ループ回数		rN/P	$2rN/P$	rN
index	左辺	local	all	all
	右辺	all	all	all
逆index	左辺	local	local	不要
	右辺	不要	local	不要

(allは,インデックス配列の全てを所有する必要があり,localは,自PEにローカルなデータに対応する分だけを分割して配置されていれば良いことを示す.)

前節の3種のアルゴリズムを,通信の有無とループ回数,および必要となるインデックス配列の観点から比較する(表1).

一番のオーバーヘッドとなるinspector内での通信は,本稿で提案した(2),(3)では不要である.

ループの回数は,配列データのサイズ N と,右辺の項の数 r,PE数 P によって決まる.(1)では,ローカルな左辺と,それに対応する右辺の項のインデックスのアクセスが行われるため rN/P 程度になる.(2)は,前半でローカルな左辺と,それに対応する右辺の項のインデックスのアクセスが行われ,後半で,全ての右辺のローカルなデータのインデックスのアクセスが行われる.前後半とも,rN/P 程度であるから,$2rN/P$ となる.(3)では,両辺の配列データ全てに対するインデックスのアクセスが行われるので,rN 程度である.一般に,N, r, P では,N のオーダーが一番大きいため,(3)のオーバヘッドが大きく,(1),(2)では,オーダーはあまり差がない.

表1のインデックスの条件の列は,両辺の添字式にインデックス配列が存在するソースコードに対してinspectorを生成する際に必要となる順・逆インデックス配列を示している.

逆インデックス配列は,$exec(p)$,$deref_{Y_i}(p)$ を求めるために必要となる.一般に,添字式 g, h_i の逆関数は,以下のようにして求められる.

- 逆インデックス配列を保持しておく(ALG(1),(2)のアプローチ).
- $\vec{i} = g^{-1}(\vec{j})$ を求めるために,全ての $\vec{k}$ に対して $\vec{l} = g(\vec{k})$ を求め,$\vec{l} = \vec{j}$ となるときの $\vec{k}$ を求める $\vec{i}$ とする(ALG(3)のアプローチ).

(3)では,後者のアプローチをとることで,逆インデックス配列を不要としている.そのため,適用可能なソースコードの範囲が広い.逆に,(1),(2)では,逆インデックス配列を必要とするため,適用範囲は,逆インデックスが生成可能なソースコードに限られる.

以上,3つのアルゴリズムについての比較を行った.通信オーバーヘッドは性能に与える影響が大きいため,通信を増大させるアルゴリズムは,避けるべきである.通信を行わないアルゴリズムのうちでは,ループ回数の少ない(2)の逆インデックスを利用するアルゴリズムを採用すべきである.このアルゴリズムが使えないコードに対しては,(2)を採用することで,ループ回数は増えるが,通信の増大は避けられる.

3.4 実際のプログラムへの適用

本節では,プログラムの特質が,inspectorの生成に与える効果について述べる.

まず,逆インデックス配列を生成できるコードの性質について考察する.LU分解のピボットのように,置換群(permutaion)の場合は生成可能である.また,loop invariantに対しては,逆インデックス関数は,1対allの写像となり,生成可能である[5].ほぼ1対1だが,ハッシュ関数のように,異なったイタレーションから同一の添字が求められる場合が稀に生じる場合,リファレンスカウントなどをつければ,逆インデックス相当のものが生成できる.

このように逆インデックス配列が生成可能であるという情報は,データ依存解析,コンパイラへのディレクティブなどの方法で与えることができる.

図2のLU分解を並列化する場合,インデックス配列 pivot は,permutaion関数である.また,i のループと i, j の2重ループにおいて,変数 k は,並列ループ内においてloop invariantである.ゆえに,逆インデックス配列を実行時に生成することが可能である.

さらに，LU 分解においては，通信を伴うデータアクセスは，loop invariant に対するものに限られることも利用できる．従って，図6のアルゴリズムの通りに inspector を生成しなくとも，コンパイル時に，ブロードキャストやマルチキャストのコードを生成できる．

これに基づいた LU 分解のハンドコンパイルを行い，並列計算機 AP1000 上での動作を確認した．

3.5 Inspector/Executor 戦略の拡張

本章で述べたインデックス配列の操作，特に，逆インデックス配列を動的に生成する手法は，DOALL 型ループだけではなく，インデックス配列の値がループ内で変更されない限り，DOACROSS 型や DOSERIAL 型のループにも適用できる．データ依存関係のため，executor においてイタレーションの順番の変更が容易ではないが，inspector において実行時解析をまとめることによってオーバーヘッドの低減が期待できる．

また，データアクセスに伴うメッセージ通信はループ以外の部分でも必要となる．メッセージ通信の送信，受信の双方のプロセッサを確定において，逆インデックス配列を利用することで，通信回数を減らすことが可能である．

現在，疎行列におけるデータアクセスの考察を通して，このような DOALL 型ループ以外で起こる irregular computation への Inspector/Executor 戦略の拡張を図っている．

4 おわりに

本稿では，不規則なアクセスを伴うループの並列化について考察した．従来の inspector を高速化，及び，適用範囲を広くするアルゴリズムを提案した．本稿で提案した2つのアルゴリズムのうち，逆インデックスを利用して，通信が起こらず，しかもループ回数の少ないアルゴリズムが最適であるが，そのアルゴリズムが使用できないコードに対しては，もう1つの適用範囲の広いアルゴリズムを使用すべきことを示した．

今後の課題は，インデックス配列を扱うランタイム・ライブラリを作成し，自動並列化コンパイラのコード生成部に組み込み，その有効性を評価していくことと，3.5節で述べた，アルゴリズムの拡張である．

謝辞

テストプログラムの実行において，並列計算機 AP1000 の実行環境を御提供戴いている（株）富士通研究所，プログラムの開発環境の御支援を戴いている横河・ヒューレット・パッカード（株）の川島貴子氏，千住晃夫氏ならびに板鼻弘太郎氏に対し，ここに深く感謝の意を表します．

また，本研究に対し，日頃より御討論頂く富田研究室の諸氏に感謝致します．

参考文献

[1] Das, R., Ponnusamy, R., Saltz, J. and Mavriplis, D. Distributed Memory Compiler Methods for Irregular Problems – Data Copy Reuse and Runtime partitioning, Languages, Compilers and Run-Time Environments for Distributed Momory Machines (eds.Saltz, J. and Mehrotra, P.), Elsevier Science Publishers B. V. (1992), 185–219.

[2] Fox, G., Hiranandani, S., Kennedy, K., Koelbel, C., Cremer, U., Tseng, C.-W. and Wu, M.-Y. FORTRAN D Language Specification, Technical Report TR90-141, Dept. of Computer Science, Rice University (Dec. 1990).

[3] Hatcher, P. J. and J.Quinn, M. *Data-Parallel Programming on MIMD Computers*, The MIT Press (1991).

[4] Hiranandani, S., Kennedy, K. and Tseng, C.-W. Compiler Support for Machine-Independent Parallel Programming in Fortran D, Languages, Compilers and Run-Time Environments for Distributed Momory Machines (eds.Saltz, J. and Mehrotra, P.), Elsevier Science Publishers B. V. (1992), 139-176.

[5] Koelbel, C. and Mehrotra, P. Compiling Global Name-Space Parallel Loops for Distributed Execution, *IEEE Transactions on Parallel and Distributed Systems*, **2**, 4 (Oct. 1991), 440–451.

[6] Zima, H. P., Bast, H.-J. and Gerndt, M. SUPERB: A tool for semi-automatic MIMD / SIMD parallization, *Parallel Computing*, **6** (1988), 1–18.