不規則アクセスを伴うループの並列化コンパイル技法

——Inspector/Executor アルゴリズムの高速化——

窪田昌史[†,*] 三吉郁夫[†] 大野和彦[†]
森眞一郎[†] 中島浩[†] 富田眞治[†]

本稿では，分散メモリ型の並列計算機に対する SPMD コード生成技法について述べる．インデックス配列による間接アクセスが存在するループを並列化すると，不規則なアクセスパターンを生ずる．従来 inspector と executor というコードを生成する手法が提案されてきたが，inspector において全対全のプロセッサ間通信が必要であり，適用できるコードの範囲にも制限がある．これらの問題を解決するために，逆インデックス法と全検査法という2つの inspector のアルゴリズムを提案する．さらに，それらの手法の有効性を高並列計算機 AP 1000 上で評価した．その結果，部分ピボッティング付き LU 分解のプログラムでは，Inspector/Executor 戦略を用いない場合に比べ，逆インデックス配列法で 42 倍，全検査法で 11 倍まで実行時間が高速化された．また，不規則疎行列とベクトルの積を求めるプログラムで，従来の inspector アルゴリズムと逆インデックス法とを比較すると，1.6 倍に実行時間の高速化が達成された．

A Parallelizing Compiler Technique for Loops with Irregular Accesses

——New Algorithms to Improve the Performance of the Inspector/Executor——

Atsushi Kubota,[†,*] Ikuo Miyoshi,[†] Kazuhiko Ohno,[†]
Shin-ichiro Mori,[†] Hiroshi Nakashima[†] and Shinji Tomita[†]

In this paper, we focus on parallelizing compiler techniques which generate SPMD codes for distributed memory computers. Parallelization of loops with indirect accesses with index arrays causes irregular access patterns. For such codes, a technique to generate inspector/executor code has been proposed. In this technique, however, the inspector must perform all-to-all global communications. Furthermore, codes, to which this method is applicable, are restricted. In order to resolve these drawbacks, we propose two inspector algorithms, inverse index method and exhaust inspection method. We evaluated the effectiveness of these methods on the highly parallel computer AP 1000. For the LU decomposition program with partial pivoting, the inverse index and exhaust inspection methods improve the performance 42 and 11 times respectively, as compared with the code without the Inspector/Executor strategy. As for the comparison of the inverse index method with the conventional one, it is proved that the code with the former is 1.6 times as fast as with the latter, in the case of the irregular sparse matrix-vector multiply.

1. はじめに

現在われわれは，分散メモリ型並列計算機を対象として，科学技術計算のプログラムの自動並列化，並列化支援の研究を行っている．

科学技術計算では，実行時間の大部分がループ内での行列に対する操作や演算に費やされる．ループ部分は，細粒度で規則性をもつ並列性を内在しているため，ループ部分を SIMD 的に各プロセッサに割り当て，すべてのプロセッサが同一のプログラムを実行する SPMD (Single Program Multiple Data Stream) プログラムへの並列化が有効である．SPMD のモデルに基づく並列言語や並列化コンパイラの研究は数多く行われている（文献4)など)．

本稿では，コード生成部におけるコンパイル技法について議論する．対象とするのは，データ分野がディレクティブなどで指定されている FORTRAN などの手続き型言語で記述された逐次プログラムである．また，プログラム中のデータ依存関係解析によって並列性の抽出などが行われていることを前提とする．

† 京都大学工学部
Faculty of Engineering, Kyoto University
* 現在 日本 IBM(株)
Presently with IBM Japan, Ltd.

本稿では，2章で，われわれが開発中の並列化コンパイラの基本方針と，DOALL 型ループ内に起こる不規則アクセス，およびこのようなループに対するInspector/Executorというコード生成戦略について述べる．3章では，この戦略の inspector 部分のアルゴリズムに的を絞り，逆インデックス法と全検査法という新たなアルゴリズムを提案する．4章では，行列のLU 分解や疎行列とベクトルの積へのアルゴリズムの適用について述べ，その有効性を評価する．最後に5章で総括を行い，今後の課題にも触れる．

2. 並列化コードの生成

2.1 基本戦略

並列化コードの生成は，

- データ分割のディレクティブに基づくプロセッサへのデータ割付け
- 割付けられたデータに基づく，各命令のプロセッサへの割付け
- データ参照に伴うメッセージ通信のコードの挿入

というフェーズに分かれる．以下，これらの基本戦略を説明する．

データ分割 配列の分割法としては，BLOCK，CYCLIC 分割が有効であることが経験的に知られている．本コンパイラにおいても，当面はこの2種類の分割を対象とする．

コンパイラへのデータ分割の指定は，配列に対するディレクティブによって行う．このような規則的な分割法を採用すると，データのプロセッサへの割付けはコンパイル時に行うことができる．

命令のプロセッサへの割付け 割り付けられたデータに対応するコード生成には，Owner Computes Rule[1]に基づく手法を採用する．これは，分割された配列の要素を保持するプロセッサ（owner）が，その配列要素をアクセスするという戦略である．代入文では，原則として左辺の配列要素を所有するプロセッサがその代入文を実行する．

通信コードの挿入 アクセスすべきデータがプロセッサ内にあるか，他のプロセッサとのメッセージ通信によって得られるかの判定は，原則としてコンパイル時に行われる．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．

2.2 不規則アクセスを伴うループ

配列の添字内の整数型配列によるインデックス参照が存在する場合，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，コンパイル時にプロセッサ間のメッセージ通信の要/不要を決定できない．このようなデータアクセスを本稿では，不規則アクセス（irregular accesses）と呼ぶ．

```
do i=1,N
  a(i) = b(index(i))
enddo
```

（a） 右辺のみインデックス配列
（a） index array in right hand side

```
do i=1,N
  a(index1(i)) = b(index2(i))
enddo
```

（b） 両辺にインデックス配列
（b） index array in both sides

図 1 DOALL 型ループ
Fig. 1 DOALL type Loop.

部分ピボッティング付きの LU 分解*，疎行列の処理など，非常によく使われるプログラムの中にも不規則アクセスを含むものが多い．**図1**（a）の例は，不規則アクセスを生ずる DOALL 型ループである．

Owner Computes Rule に基づいて，a（i）を保持するプロセッサがi番目のイタレーションを実行することはコンパイル時に決定できるが，b(index(i))を保持するプロセッサは実行時にならなければ確定しない．そのため，b(index(i)) を送信するプロセッサは実行時に決定される．

ゆえに，b(index(i)) のアクセスのたびに index(i)の値を調べ，b(index(i)) を保持するプロセッサへリクエストのメッセージを送信し，これを受信したプロセッサが b(index(i)) の値を送り返すことが必要である．

2.3 Inspector/Executor 戦略

実行時のデータアクセスのたびに送受信プロセッサを確定するとオーバヘッドが大きくなる．そのため，**図2**に示すように不規則アクセスを伴う DOALL 型ループに対して，先にプロセッサ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

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

∘は，＋などの演算を表す．ベクトル表記は多重ループを表す．

図 2 DOALL 型ループの一般形
Fig. 2 Generic form of DOALL type Loop.

* 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．

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[3]．本稿ではこのコード生成戦略を **Inspector/Executor 戦略**と呼ぶ．以下にプロセッサ p におけるコードの概要を示す．なお，この中でプロセッサ q は $p \neq q$ となるすべてのプロセッサである．

1. inspector では，以下のリストが作成される．
   （a）プロセッサ間通信のためのデータ参照関係を示すリスト
   - p が参照し，かつ，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，かつ，q が参照す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 以外のプロセッサ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 **non-local_iter(p)**
2. executor では，データの送受信および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 *non-local_iter(p)* を実行する．

executor では，（a）送信すべきデータを先送りし，通信が行われている間に，（b）データ・アクセスに通信を必要としないイタレーション *local_iter(p)* を実行する．ついで（c）必要とするデータを受信し，（d）残りのイタレーション nonlocalIter(p) を実行する．このように，executor において，あるプロセッサ p から q への通信を一度にまとめることで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することで通信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 アルゴリズムの高速化

本章では，Inspector/Executor 戦略に対して**逆インデックス法**と**全検査法**という新たなアルゴリズムを提案する．これらのアルゴリズムと文献 2), 3) で提案されたアルゴリズム（以下，**従来法**と呼ぶ）との大きな違いは，inspector での *send_list* の作成法にある．

従来法では，*send_list(p, q)* = *recv_list(q, p)* であることを利用し，*recv_list* をプロセッサ間で通信することにより *send_list* を求める．これに対し，われわれの提案する逆インデックス法では，インデックス配列の逆写像である**逆インデックス配列**を求め，通信を行わずに *send_list* を求める．また，全検査法ではすべてのインデックス配列を参照することで，逆インデックスを用いずに，かつ，通信を行わずに *send_list* を求める．

3.1 従来法の inspector のアルゴリズム

図 2 に対する inspector は，以下のようになる．

1. 自分のプロセッサ内で必要とするデータのリスト *recv_list*，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（**図 3**）．
2. *recv_list* を他のプロセッサへ送信し，逆に，他のプロセッサから *recv_list* を受信する．受信したリストは *send_list* に相当する．

例えば，図 1（a）では，a（i）の分割に従って，あるプロセッサ（p とする）で実行すべきイタレーショ

```
do j⃗ ∈ {プロセッサ p で実行するイタレーションの
集合}
  local_flag = true;
  /* recv_list_Yi(p, src) を求める */
  do each Yi
    src = Yi(hi(j⃗)) を所有するプロセッサ番号;
    if (src != p) then
      local_flag = false;
      append hi(j⃗) to recv_list_Yi(p, src);
    endif
  enddo
  /* 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
enddo
```

図 3 従来法，逆インデックス法の *recv_list*, *local/nonlocal_iter* を求めるアルゴリズム
Fig. 3 Algorithm of the conventional method and the inverse index method (the part of obtaining *recv_list*, *local/nonlocal_iter*).

ンも規則的に決定される．そのイタレーション番号から右辺のデータのインデックス配列 index(i) を索引すると，b(index(i)) を所有しているプロセッサ（q とする）が決定できる．ゆえに，$p \neq q$ なら q から p への通信が必要であることがわかり，index(i) の値を *recv_list*(p, q) に加えることになる．また，*local/nonlocal_iter* は，*recv_list* を作成する過程で，あるイタレーションの実行にメッセージ通信を必要とするかどうかを判定しながら決定される．

このアルゴリズムの問題点は，*send_list* を求めるために全対全通信を行っており，通信のオーバヘッドが大きくなることである．

また，図1(b)のように左辺にもインデックス配列が存在する場合，そのインデックス配列の逆写像にあたる逆インデックス配列が求められなければ，あるプロセッサにおいて，何番目のイタレーションを実行すべきかを求めることができない．文献3)では，このように左辺にインデックス配列が現れる場合を考察の対象から外している．したがって，左辺にインデックス配列が現れる LU 分解などのコードに Inspector/Executor 戦略は適用不可能である．

3.2 逆インデックス法によるアルゴリズム

send_list の作成に通信を必要とする，代入文の左辺にインデックス配列が存在する場合に適用できないという前節で指摘した従来法の問題点を，逆インデックス配列を積極的に活用することで解決するアルゴリズムを提案する．以下に，図2に対する逆インデックス法のアルゴリズムを示す．

1. 右辺の配列の逆インデックス配列を求めて，他のプロセッサが必要としているデータのリスト *send_list* を作成する（**図4**）．
2. 自分のプロセッサ内で必要とするデータのリスト *recv_list*，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．このフェーズは，従来法の1. と同一である（図3）．

まず，図1(a)において，*send_list* が通信を行わずに求められることを示す．**図5**は配列の要素数 N=16，プロセッサ数 P=4 の例である．プロセッサ1が所有する配列 b の要素1から4を参照するイタレーションは，index(i) の逆インデックス配列 inverse を索引し，それぞれ 10, 2, 13, 6 と求まる．それらのイタレーションを実行するプロセッサは，owner computes rule により 3, 1, 4, 2 となるため，配列 b の要素の通信が必要であり，*send_list* (1, 2)=4, (1, 3)=1, (1, 4)=3 となる．他のプロセッサについても同様である．

```
do each Y_i
  do j⃗ ∈ deref_Yi(p)
    dest = X(g(j⃗)) を所有するプロセッサ番号;
    if (dest != p) then
      append h_i(j⃗) to send_list_Yi(p, dest);
    endif
  enddo
enddo
```

$\mathrm{deref}_{Y_i}(p)$ = {右辺の配列データ Y_i のうちプロセッサ p が所有するデータを参照するイタレーションの集合}

図4 逆インデックス法の *send_list* を求めるアルゴリズム
Fig. 4 Algorithm of the inverse index method (the part of obtaining the *send_list*).

また，代入文の両辺にインデックス配列が存在する図1(b)についても，N=16, P=4 の例を**図6**に示す．プロセッサ1の *send_list* は，右辺のインデックス配列 index 2 の逆インデックス inverse 2 を索引して，イタレーション 13, 11, 5, 14 で配列 b の要素が参照されることがわかるので，さらに，左辺のインデックス配列 index 1 を索引して，それぞれのイタレーションが実行されるプロセッサ1，2，4，4を求める．よって，*send_list*(1, 2)=1, (1, 4)=3, 4 と

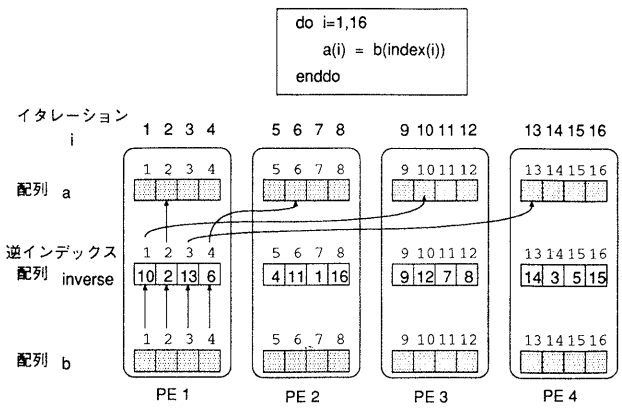


図5 逆インデックス法による *send_list* の作成（右辺のみにインデックス配列）
Fig. 5 Generation of *send_list* with the inverse index method (index array in right hand side).

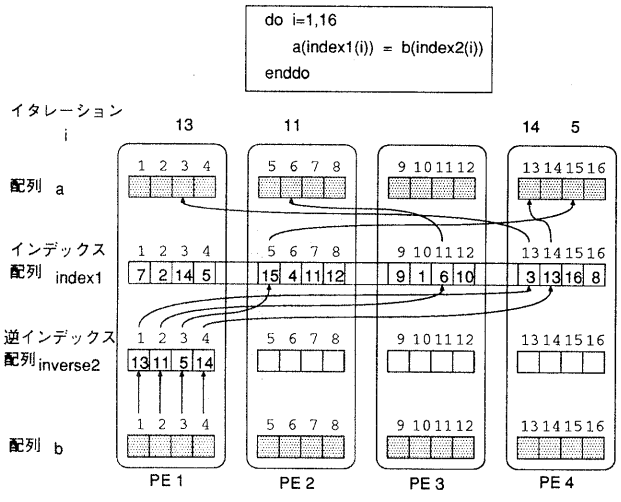


図 6 逆インデックス法による *send_list* の作成（両辺にインデックス配列）

Fig. 6 Generation of *send_list* with the inverse index method (index array in both sides).

なる．

逆インデックス配列の生成 一般に，逆インデックス配列を生成するには，インデックス配列の値が更新されたら，その添字の番号と値をブロードキャストし，対応する逆インデックスを所有するプロセッサは，そのブロードキャストに基づいて逆インデックス配列を更新しなければならない．

LU 分解のピボット配列などでは，1対1の置換群(permutation) であり，ピボット選択時のブロードキャストを，逆インデックス配列の更新に利用することができる．そこで，

index permutation pivot (1 : N)

のようにプログラマがインデックス配列の性質をディレクティブで指定する．そのインデックス配列への操作は，swap(pivot, i, j) や shuffle(pivot, k) といったインデックス値の交換や *k*-シャフル置換などのライブラリ関数に限るものとする．コンパイラはこのディレクティブに従って逆インデックス配列を求めるコードを生成する．

3.3 全検査法によるアルゴリズム

このアルゴリズムでは，すべてのインデックス配列のすべての要素を各プロセッサが保持し，そのすべてを検査する（**図7**）．つまり，図1(b)なら，i=1, N のすべてのイタレーション番号により両辺のインデックス配列 index 1, index 2 を索引する．その結果，あるプロセッサ p が a(index 1(i)) または b(index 2(i)) を保持していることがわかれば，プロセッサ p がそのイタレーションに関わっている（実行，通信）ことになる．そのようなイタレーションについては，データの参照関係を求め，適切なリストの操作を行う．

このアルゴリズムでは，各プロセッサがインデックス配列のコピーを保持するだけで適用できる．そのため，コードの逆インデックス配列の生成可能性にかかわらず適用できるという特徴を持つ．

3.4 アルゴリズムの比較

前節の3種のアルゴリズムを，通信の有無，必要となるインデックス配列，inspector のループの実行時間の観点から比較する（**表1**）．

inspector 内での通信 逆インデックス法，全検査法では通信が不要である．それに対して，従来法では全対全通信による $P-1$ 回ずつの送受信が必要である．

アルゴリズムの適用条件 inspector のアルゴリズムを適用できるためのインデックスの条件は，代入文の左辺にインデックス配列が存在するか否かで異なる．

```
do j⃗ ∈ {逐次ループの全てのイタレーション}
  local_flag = true;
  dest = X(g(j⃗)) を所有するプロセッサ番号;
  do each rhs Y_i
    src = Y_i(h_i(j⃗)) を所有するプロセッサ番号;
    /* send_list_{Y_i}(p, dest) を作成する */
    if (src == p) then
      if (dest != p) then
        append h_i(j⃗) to send_list_{Y_i}(p, dest);
      endif
    /* recv_list_{Y_i}(p, src) を作成する */
    else  /* (src != p) */
      local_flag = false;
      if (dest == p) then
        append h_i(j⃗) to recv_list_{Y_i}(p, src);
      endif
    endif
  enddo
  /* local/nonlocal_iter を求める */
  if (dest == p) then  /*
    if (local_flag == true) then
      append j⃗ to local_iter(p);
    else
      append j⃗ to nonlocal_iter(p);
    endif
  endif
enddo
```

図 7 全検査法の inspector アルゴリズム

Fig. 7 Algorithm of the exhaust inspector method.

表 1 inspector のアルゴリズムの比較
Table 1 Comparison of the inspector methods.

アルゴリズム	従来法	逆 index 法	全検査法
通信回数	$P-1$	不要	不要
左辺に index 配列が存在しない場合			
右辺 index	local	local	all
逆 index	不要	local	不要
左辺に index 配列が存在する場合			
左辺 index		all	all
逆 index	適用	local	不要
右辺 index	不可能	all	all
逆 index		local	不要

allは，インデックス配列のすべてを所有する必要があり，localは，自プロセッサにローカルなデータに対応する分だけを分割して所有していれば良いことを示す．

従来法では，実行すべきイタレーション番号を得るために，左辺の添字式の逆インデックスを求める必要がある．そのため，左辺にインデックス配列が存在すると適用不可能となる．

逆インデックス法では，左辺，右辺の添字式の逆インデックスと，これらのイタレーションに対する添字が，すべて求められる場合に適用可能である．

全検査法では，すべてのインデックス配列を必要とするが，逆インデックスは全く不要であるため適用範囲が広い．

inspector でのループの回数 プロセッサ数 P と，配列データの要素数 N に大きく依存する．右辺の項数を r，*send/recv_list* を求めるループの1イタレーションの実行時間の平均を T_L，*local/nonlocal_iter* を求める時間の合計を T_I，従来法の inspector の1回のメッセージの送受信にかかる平均時間を T_M とすると，inspector の実行時間は以下の式で近似できる．

$$\text{従来法}=\frac{rNT_L}{P}+T_M(P-1)+T_I \qquad (1)$$

$$\text{逆インデックス法}=\frac{2rNT_L}{P}+T_I \qquad (2)$$

$$\text{全検査法}=2rNT_L+T_I \qquad (3)$$

これらの式から，inspector の実行時間について考察する．従来法では，プロセッサ数に比例する項と反比例する項が存在するため，プロセッサ数を増加すると，inspector の実行時間は一度減少した後，$P=\sqrt{rNT_L/T_M}$ で最小となり，さらにプロセッサ数を増加すると実行時間が増加してしまう．それに対して，逆インデックス法では，プロセッサ数の増加とともに inspector の実行時間が単調に減少する．また，全検査法ではプロセッサ数にかかわらず実行時間は一定になる．

4. Inspector/Executor 戦略の有効性の評価

本章では，LU 分解，不規則疎行列とベクトルの積の例題について，3章で述べたアルゴリズムを適用したハンドコンパイルによるプログラムを高並列計算機 AP1000[5] 上で実行し，時間を計測して得られた結果をもとに，アルゴリズムの有効性を検証する．

4.1 部分ピボッティング付き LU 分解

LU 分解のコードを図8に示す．これを，Inspector/Executor 戦略を適用しなかった場合（以下，**単純法**と呼ぶ）と Inspector/Executor 戦略の逆インデックス法と全検査法を適用した場合で実行し，その実行時間を計測した．なお，逆インデックスが必要なため，従来法は適用不可能である．

4.1.1 配列の分割

行列Aは，行方向，列方向の両方向について CYCLIC 分割した．単純法では，行列Aの行方向の分割に従い，pivot を分割配置する．逆インデックス法と全検査法では pivot のすべての要素を各プロセッサでコピーして所有する．

逆インデックス配列は，pivot が，図8のプログラムにあるように，代入文の両辺で参照されているため，表1より，各プロセッサで分割して所有する．

pivot の逆インデックスの更新は，pivot が更新されるときに，各プロセッサが対応する逆インデックス配列の要素を保持しているかを検査して行う．逆インデックス配列を生成するために新たなプロセッサ間通信は生じない．

```
decomposition d(N,N)
align a(i,j)  with d(i,j)
align pivot(i)  with d(i,:)
distribute d(CYCLIC)(CYCLIC)
do k=1,N
  /* ピボットの選択，交換 */

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

図 8 部分ピボッティング付き LU 分解のコード
Fig. 8 Codes for LU decomposition with partial pivoting.

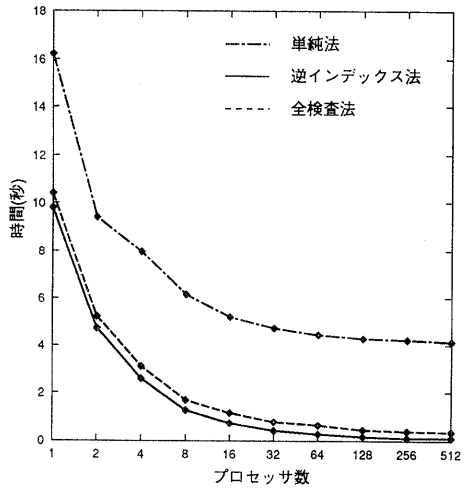


図 9 LU 分解の実行時間 (256×256)
Fig. 9 Execution time of LU decomposition (256×256).

4.1.2 実行結果

AP 1000 において，サイズ $N=64$, 128, 256 の $N\times N$ 行列の LU 分解の実行時間を測定した．**図 9** に $N=256$ の実行時間を示す．

4.1.3 考察

図 9 から，Inspector/Executor 戦略を適用した場合，大幅な高速化が行われることがわかる．逆インデックス法を使用して Inspector/Executor 戦略を適用した場合の実行時間は，単純法の最高 42 倍（プロセッサ数 512）まで高速化された．また，全検査法では，同じ条件で実行時間が最高 11 倍まで高速化された．このような良好な結果を得たのは，データアクセスのたびに送受信プロセッサを確定するために行われるブロードキャストの回数を，大幅に削減したためである．

逆インデックス法は全検査法に比べ，プロセッサ数 512 で実行時間が 3.7 倍高速になっている．これは，式 (2) に示すように，逆インデックス法ではプロセッサ数が増加するにつれて inspector のコストが小さくなるためである．この差は，プロセッサ数の増加について大きくなるため，大規模な並列システムにおいて，逆インデックス法が有効であるといえる．

4.2 不規則疎行列とベクトルの積

有限要素法などで必要となる．不規則疎行列とベクトルとの積のプログラム（以下 SMVM : Sparse Matrix-Vector Multiply と略す）を用いて評価を行う．アルゴリズムは，Inspector/Executor 戦略を用いない

```
decomposition d(N)
align nzu(i),iu(i,*),au(i,*) with d(i)
align nzl(i),il(i,*),al(i,*) with d(i)
align ad(i),b(i),x(i) with d(i)
distribute d(BLOCK)

do i=1,N
  x(i) = 0.0;
  do j=1,nzl(i)
    x(i) = x(i) + al(i,j) * b(il(i,j))
  enddo;
  x(i) = x(i) + ad(i) * b(i)
  do j=1,nzu(i)
    x(i) = x(i) + au(i,j) * b(iu(i,j))
  enddo
enddo
```

図 10 不規則疎行列 A とベクトル b の積 x を求めるコード
Fig. 10 Code of obtaining product x by multiplying irregular sparse matrix A by vector b.

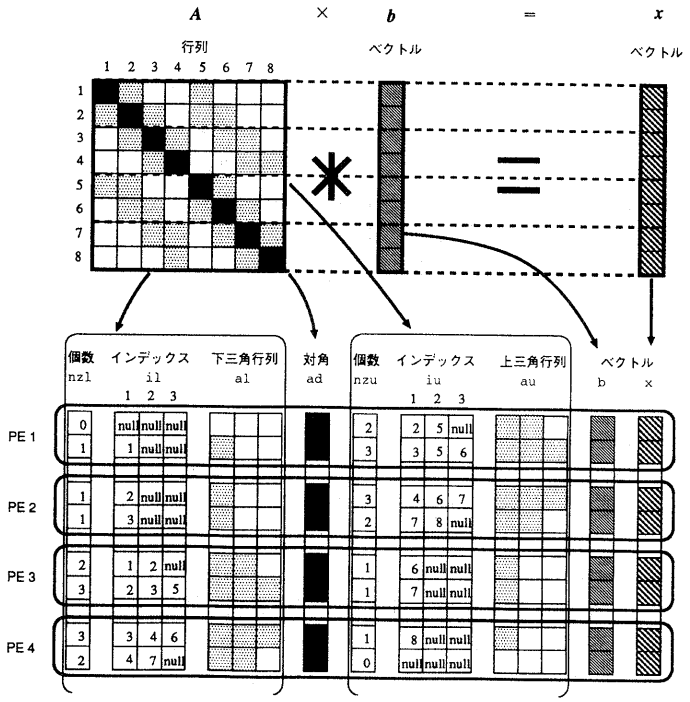


図 11 不規則疎行列とベクトルの積
Fig. 11 Irregular sparse matrix and vector multiply.

単純法と，3章で述べた3種類の inspector アルゴリズム（従来法，逆インデックス法，全検査法）を用いる．これらのアルゴリズムを適用したコードをハンドコンパイルによって記述し，その評価を行う．**図10**に逐次の SMVM のコードを示す．

4.2.1 配列の分割

不規則疎行列Aは非零要素が不規則な位置に存在し，かつ，非零要素の個数が行列のサイズNに対して非常に少ないものである．そこで，記憶効率の向上のため**図11**のような配列に格納されることが多い．

1. 第 i 行の (i, i+1) 要素から (i, N) 要素のうち，非零要素の個数を nzu(i)とする．
2. 第 i 行において，(i, j) 要素（ただし i+1≤j≤N）中の非零要素を列番号の昇順で取り出したとき，第 j 番目の要素の存在する列番号を iu(i, j) とする．
3. iu(i, j) に対応する非零要素を au(i, j) とする．
4. 第 i 行の対角要素を ad(i) とする．

1. から3. で表現されるのは上三角行列の部分であり，下三角行列についても第 i 行の非零要素の個数，要素の存在する列番号，非零要素の値をそれぞれ，nzl(i), il(i, j), al(i, j) に格納する．

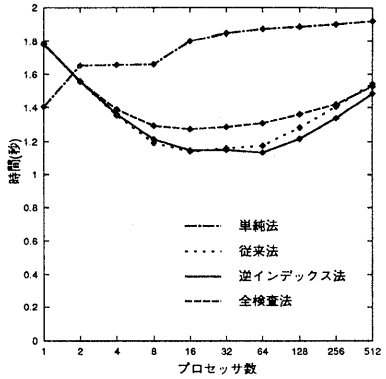
特に，行列Aが対称行列の場合は，il を iu の逆インデックスとして，逆に iu を il の逆インデックスとして参照する．したがって逆インデックス法を適用することができる．

この問題では，インデックス配列の値は更新されず，また，iu, il ともにプログラムの開始時に与えられているため，逆インデックス配列を求めるためのブロードキャストによるオーバヘッドは生じない．

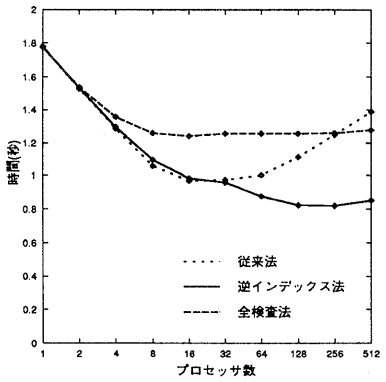
なお，非対称行列の場合も，行方向についてのインデックス配列と列方向のインデックス配列の両方を用意すれば，互いの逆インデックス配列として参照することができる．

4.2.2 実行結果

AP1000 において，SMVM の実行時間を測定した，プロセッサ数は LU 分解と同じく P=1 から 512，行列のサイズは，N=64, 256, 1K, 4K, 16K, 32K とし，1行あたりの非零要素数は対角要素を除いて最大6個である．実行時間の測定は，SMVM のサブルーチン全体と，その中の inspector 部のみの2種類で行った．行列 A やベクトル b, x は図 11 に示すように，行方向に BLOCK 分割した．**図12**に行列のサイズが N=1K の場合の SMVM 全体の実行時間と inspector のみの実行時間を示す．



（a） 全体の実行時間
（a） total time



（b） inspector の実行時間
（b） inspector time

図12 SMVM の実行時間
Fig. 12 Execution time of SMVM.

4.2.3 考察

単純法との比較 図12(a)から，Inspector/Executor 戦略を用いたすべてのアルゴリズムで高速化されていることがわかる．これが最高となるのは，プロセッサ数 P=64 台，逆インデックス法で実行した場合で，実行時間は1.65倍まで高速化された．同じく，P=64 で従来法の inspector では最高 1.59 倍，全検査法では1.43倍の高速化が達成された．

従来法の inspector との比較 SMVM の逆インデックス法，全検査法による inspector のアルゴリズムが従来法のアルゴリズムに対してどの程度高速化されているかを算出すると，逆インデックス法では，プロセッサ数 P=512 で1.63倍，全検査法では，同じく P=512 で1.09倍まで高速化されている．

図12(b)の従来法では，プロセッサ台数の増加と

ともに，inspector の実行時間は一度減少し，プロセッサ数 16 の場合を最小として再び増加する．これに対して逆インデックス法ではプロセッサ数の増加に伴い，inspector の実行時間が減少しており，通信を不要にした効果が現れている．また，全検査法ではすべてのインデックスを検査するため，プロセッサ数が増加しても inspector の実行時間はほぼ一定となっている．これら inspector の実行時間は，3.4 節で述べた理論式に合致しているといえる．

図12(a)の SMVM の実行時間は，(b)の inspector 部分のみの実行時間に比べ，速度の改善が見られない．これは，inspector 部分で通信を削除したにもかかわらず，引き続き実行される executor 部分での通信のオーバヘッドが大きいためである．

5. 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おいて，すでに提案されている Inspector/Executor 戦略の inspector 部分を高速化し，かつ，適用範囲を拡張するアルゴリズム（逆インデックス法ならびに全検査法）を提案した．本稿で提案したアルゴリズムの特徴は，プロセッサ間通信を行わずに inspector 部分が実行可能な点にある．

逆インデックス法では，インデックス配列の逆写像である逆インデックス配列を用いることで従来法で必要であった通信を排除する．全検査法では，すべてのインデックス配列を検索することで，逆インデックスを用いることなく通信を排除する．

これにより，全検査法ではアルゴリズムの適用範囲の制限をなくすことが可能となり，逆インデックス法でも，従来適用できなかったピボッティング付き LU 分解等の多くの問題に適用可能となった．

また，並列計算機上での評価結果より，逆インデックス法が従来法よりも高速であることが示された．さらに，プロセッサ台数が増加すると全検査法も従来法より高速であることも示された．

今後の課題として，*send_list*, *recv_list* などのリスト操作の高速化，より一般的な問題にたいする逆インデックス配列の生成可能性の評価，ならびにインデックス配列の性質のコンパイラへの指定法の検討等を考えている．

謝辞 テストプログラムの実行にあたり，高並列計算機 AP1000 の実行環境をご提供いただきました(株)富士通研究所，プログラム開発環境をご支援いただきました横河・ヒューレット・パッカード(株)に感謝の意を表します．また，日頃より有益なご意見をいただく富田研究室の諸氏に感謝いたします．なお，本研究の一部は文部省科学研究費補助金（重点領域研究(1)課題番号04235103「超並列ハードウェア・アーキテクチャの研究」）による．

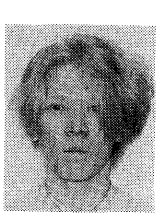
参考文献

1) Hiranandani, S., Kennedy, K. and Tseng, C.-W.: Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, *Proceedings of Supercomputing '91, Albuquerque*, NM (Nov. 1991).
2) 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 Memory Machines* (Saltz, J. and Mehrotra, P. eds.), Elsevier Science Publishers, pp. 139-176 (1992).
3) Koelbel, C. and Mehrotra, P.: Compiling Global Name-Space Parallel Loops for Distributed Execution, *IEEE Trans. Parallel and Distributed Syst.*, Vol. 2, No. 4, pp. 440-451 (1991).
4) Saltz, J. and Mehrotra, P. (eds.): *Languages, Compilers and Run-Time Environments for Distributed Memory Machines*, Elsevier Science Publishers B. V. (1992).
5) 清水俊幸，堀江健志，石畑宏明：高速メッセージハンドリング機能—AP1000 における実現—，並列処理シンポジウム JSPP '92 (June 1992).

(平成5年9月14日受付)
(平成6年2月17日採録)

窪田 昌史

1969年生．1991年京都大学工学部情報工学科卒業．1993年同大学院修士課程修了．同年日本アイ・ビー・エム(株)入社，インフォメーション・テクノロジー・ソリューション(株)出向．現在に至る．



三吉 郁夫

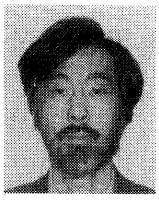
昭和45年生．平成5年京都大学工学部情報工学科卒業．同年同大学院修士課程入学．現在，並列化コンパイラおよび並列化支援システムの研究を行っている．

大野 和彦（正会員）

昭和45年生．平成5年京都大学工学部情報工学科卒業．同年同大学院修士課程入学．現在の研究テーマは非数値分野を主対象とする並列プログラミング言語．

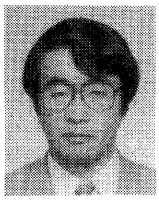
森 眞一郎（正会員）

1963年生．1987年熊本大学工学部電子工学科卒業．1989年九州大学大学院総合理工学研究科情報システム学専攻修士課程修了．1992年同大学院博士後期課程単位取得退学．同年京都大学工学部情報工学教室助手，現在に至る．並列処理，計算機アーキテクチャの研究に従事．IEEE-CS会員．



中島 浩（正会員）

昭和31年生．昭和56年京都大学大学院工学研究科情報工学専攻修士課程修了．同年三菱電機(株)入社．推論マシンの研究開発に従事．平成4年より京都大学工学部助教授．並列計算機のアーキテクチャ，プログラミング言語の実装方式に関する研究に従事．工学博士．昭和63年元岡賞，平成5年坂井記念特別賞受賞．



富田 眞治（正会員）

1945年生．1968年京都大学工学部電子工学科卒業．1973年同大学院博士課程修了．工学博士．同年京都大学工学部情報工学教室助手．1978年同助教授．1986年九州大学大学院総合理工学研究科教授，1991年京都大学工学部情報工学科教授，現在に至る．計算機アーキテクチャ，並列処理システムなどに興味を持つ．著書「並列計算機構成論」「計算機システム工学」「並列処理マシン」など．電子情報通信学会，IEEE，ACM各会員．