プログラミング-言語・基礎・実践- 11－11
(1993. 3. 11)

不規則アクセスを伴うループの自動並列化コンパイラ技法

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
〒606-01 京都市左京区吉田本町

おもまし　本報告では，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．このようなコードに対して，insepctorとexecutorというコードを生成する手法が提案されてきた．従来の手法では，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AP1000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．

和文キーワード　並列化コンパイラ，分散メモリ型アーキテクチャ，実行時解析，SPMD，インデックス配列

An Automatic Parallelizing Compiler Technique for Loops with Irregular Accesses

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

Abstract　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 AP1000. 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.

英文 key words　parallelizing compiler, distributed memory architecture, run-time analysis, SPMD, index array

1 はじめに

現在我々は，科学技術計算を対象として，分散メモリ型並列計算機へ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，全てのプロセッサが同一のプログラムを実行し，ループ部分をSIMD的に各プロセッサに割り当てるSPMD(Sigle Program Multiple Data Stream)モデルによるプログラムの並列化が有効となる．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[1, 2]など，数多く行われている．

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単位プロセッサ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，メモリ空間に共有部分がなく，プロセッサ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，プログラムの実行性能の低下をもたらす．

それゆえ，データ分割法を決定するには，プロセッサ内の処理量の粒度(granularity)，プロセッサ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する．データ分割は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を対象とする．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．

本論文では，2章で，我々が開発中の並列化コンパイラシステムの基本方針と，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を紹介する．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．さらに，それらが生成可能なソースコードの性質についても議論する．4章では，

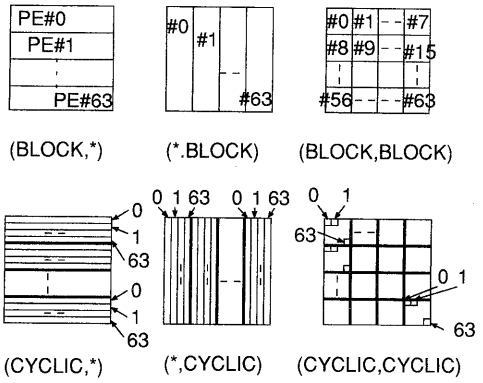

図1: データ分割法(プロセッサ数64)

行列のLU分解や疎行列の積といった実際の例題へのアルゴリズムの適用について述べ，5章では，その性能評価について述べる．最後に6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対応するコードの生成
  - 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，図1に示すように，BLOCK,CYCLIC分割が有効であることが経験的に知られている[3]．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り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り付け 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[4]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プロセッサ(*owner*)に，そのデータをアクセスする命令を割り付け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プロセッサにその代入文が割り付けられる．

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．

部分ピボッティング付きの LU 分解[1]，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．

次の例は，不規則アクセスを生ずる DOALL 型ループである．

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

Owner Compute 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，配列 b(index(i)) の要素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度に送受信プロセッサ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]．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 $p \neq q$となる任意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の作成
   - 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の作成
   - 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 **nonlocal_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 *nonlocal_iter(p)* を実行する．

executor では，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．ついで必要とするデータを受信し，残りのイタレーション *nonlocal_iter*(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　inpsector のアルゴリズムの高速化

本章では，Inspector/Executor 戦略の inspector 部分について，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．詳細なアルゴリズムは [6] を参照されたい．

3.1　従来の inspector のアルゴリズム

以下では，[7][5] で提案された inspector のアルゴリズム（以下，従来法と呼ぶ）を示す．

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* 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プロセッサへ要求リストを送信する．

[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．

4. 逆に他のプロセッサから要求リストを受信する．これは，*send_list*に相当する．

このアルゴリズムでは，*recv_list*, *local/nonlocal_iter*は自分のプロセッサ内のインデックス配列の検査だけで求められているが，*send_list*を求めるには全てのプロセッサが他の全プロセッサに対して通信を行う必要がある．ゆえに，計算機全体で$P(P-1)$個のメッセージが通信されることになり通信のオーバーヘッドが大きくなる．

また，従来法のinspectorが生成できるためには，左辺の添字式gの逆関数をとって$exec(p)$を求める必要がある（ここで，$exec(p)$は，プロセッサpにおいて実行されるイタレーション番号の集合であり，$local_iter(p) \cup nonlocal_iter(p)$に等しい）．しかし，

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

のように左辺にインデックス配列が存在すると，`index(i)`の逆インデックス配列を求めなければ$exec(p)$を得ることができない．Kali[5]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．ゆえに，左辺の添字式は`i`や`i`±(定数)などしか許されていない．つまり，従来法の適用範囲は$exec(p)$が計算を必要としないか，あるいは簡単な加減算で求められるに場合に限られている．従って，左辺にインデックス配列が現れるピボッティング付きLU分解などのコードでも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

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．このアルゴリズム（以下逆インデックス法と呼ぶ）では，*send/recv_list*作成のための通信が不要となるだけではなく，代入文の左辺の配列にインデックス配列が存在するコードにもinspectorを生成することが可能となる．このようなinspectorのアルゴリズムを以下に示す．

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．
2. 右辺の配列のインデックス配列の値から，自プロセッサ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．

一般にインデックス配列の逆インデックス配列を求めることは困難である．しかし，LU分解におけるピボットのように，インデックス配列が1対1の置換群(permutation)となる場合，この性質がコンパイラに与えられていれば，実行時に逆インデックス配列を生成することが可能である．

ただし，逆インデックス配列を生成するときに，新たな通信を生ずることなく生成できるとは限らない．そこで，逆インデックス配列を必要としないアルゴリズムを次節に示す．

3.3 全検査法によるアルゴリズム

逆インデックス配列のかわりにインデックス配列の全ての要素を各プロセッサが保持し，このインデックス配列を全て検査することで，通信を行わずに*send, recv_list*を求めることができる．このアルゴリズム（以下，全検査法と呼ぶ）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．

全検査法では，全てのイタレーションに対応するインデックス配列を検査して，以下の処理を行う．

1. 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．
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*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．

4 逆インデックス配列の生成

本章では，LU分解や不規則疎行列とベクトルの積の例題に対する逆インデックスの生成について述べる．

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2に示す．図3の上段は，8×8行列のLU分解が`k=3`まで進んだところを示している．行列とピボット配列は2つのプロセッサに分割して割り付けられているものとする．この図から，`i=4`から8のイタレーションで分解される行が，ピボットの存在によって2,3,4,7,8と不規則な配置となることがわかる．各プロセッサで分解すべき行は，ピボット配列の値を参照しても求めることはできない．しかし，図3の下段のようにピボット配列の逆インデックス配列を参照すると，その値が`4(=k+1)`以上の行を分解すべきことがわかる．つまり，プロセッサ1ではイタレーション4,5,8が実行され，それぞれ第4,3,2行が分解される．同様にプロセッサ2ではイタレーション6,7が実行され，第6,7行が分解される．

インデックス配列`pivot`はpermutaionであるため，インデックス配列の各要素の逆は必ずただ一つ存在する．実行時にピボット行が選択されて`pivot`の値が交換されるときに，対応する逆インデックス配列の値も

```
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
  l = k
  al = abs(a(pivot(l),k)
  do i=1,n
    if (abs(a(pivot(i),k)) .GT. al) then
      l = i
      al = abs(a(ip(l),k))
    endif
  enddo
  if (l .NE. k) then
    lv = pivot(k)
    pivot(k) = pivot(l)
    pivot(l) = lv
  endif
  do i=k+1,N
    a(pivot(i),k) = a(pivot(i),k)/a(pivot(k),k)
  enddo
  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 2: 部分ピボッティング付き LU 分解のコード

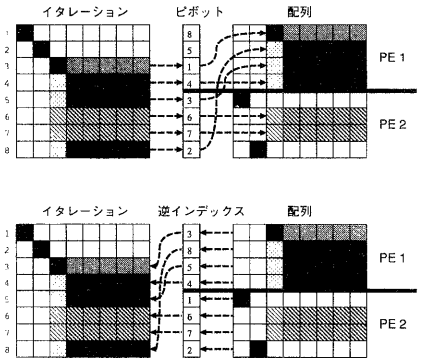

図 3: LU 分解のインデックス参照

同時に変更すれば，新たな通信を生ずることなく逆インデックス配列が求められる．

4.2 不規則疎行列とベクトルの積

有限要素法などの問題では，不規則な疎行列をメモリ効率の良いデータ構造で現するため，不規則アクセスが生じる．この疎行列は，ICCG 法, SOR 法 [8] などの反復解法で解かれることが多い．本節では，反復

```
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 4: 不規則疎行列と 1 次元ベクトルの積を求めるプログラム

解法の中で使用される基本的な処理である，係数行列 A とベクトル b との積 x（以下 SMVM:Sparse Matrix-Vector Multiply と略す）を求めるプログラムへの Inspector/Executor 戦略の適用について述べる．図 4にそのプログラムを示す．

行列 A とベクトル b，x を格納する配列は，図 5になる．行列 A は，記憶効率の向上のため，以下のような配列に格納される．

1. 第 i 行の対角線より右側にある非零要素の個数を nzu(i) とする．
2. 第 i 行において，対角線の次から右へ向かって数えて第 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 行の対角成分を 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る．下三角行列についても，第 i 行について，その非零要素の個数，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．

図 4では，一番外側の i についてのループはイタレーション間の依存関係がないので，これを並列化の対象とする．各イタレーションでは，内側の j のループにおいて右辺に現れる配列 b がインデックス参照されている．このとき，il(i,j) と iu(i,j) は互いに逆インデックス配列の関係にあり，逆インデックス法が適用できる．

例えば，図 5において，x(5) を求めるために A の第 5 行と b の 1,2,5 番の要素を必要としていることが分かる．図のように配列がプロセッサに分割して配置されているため，b の 1,2 番の要素へのアクセスは通信を伴う．

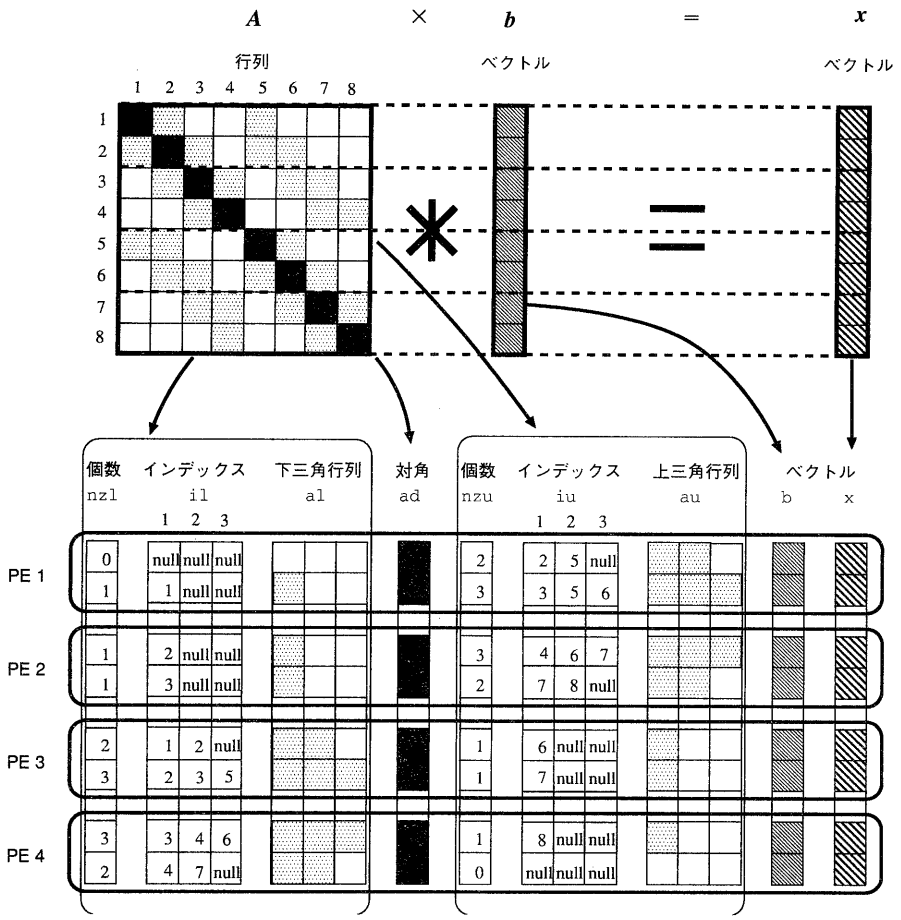

図 6: 疎行列の配列の分割

ここで下三角行列を列方向に見ると，第 1 列は第 2,5 行に非零要素を持つ．そのため，b(1) は x(2)，x(5) を求めるために参照されることがわかる．この行列は対称行列であるため，下三角行列を列方向に走査した場合の非零要素の位置の情報は，上三角行列を行方向に走査した場合の非零要素の位置の情報に他ならない．

そこで，配列の要素 b(i) と下三角行列のどの要素との積が必要であるかは，上三角行列のインデックス配列 iu を検査することで求められるということになる．上三角行列の要素との積がとられる配列 b の要素も同様に il を検査することで求められる．ゆえに，iu，il が互いに逆インデックス配列の関係にあることになる．

5 Inspector/Executor 戦略の有効性の評価

本章では，前章で述べた例題に Inspector/Executor 戦略を適用したプログラムをハンドコーディングで作成し，並列計算機 AP1000 上 [9] で実行した結果をもとに，アルゴリズムの有効性を検証する．

5.1 部分ピボッティング付き LU 分解

これを，Inspector/Executor 戦略を適用しなかった場合（以下，単純法とする）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，単純法では，当該プロセッサが pivot(k) を保持しているかどうかを

プロセッサ間のブロードキャストによって求めている．また，ループの実行でも，イタレーションの実行のたびにブロードキャストを行っている．なお，この例では，代入文の左辺にインデックスが現れるため，従来法による inspector は生成できない．

5.1.1 配列の分割と並列化

LU 分解では，行列 A を行方向，列方向の両方向について CYCLIC 分割して負荷分散を図る．ピボット行選択のためのインデックス配列 pivot は，Inspector/Executor 戦略を使用しない単純法の場合は，配列 A の行方向と同様に CYCLIC 分割し，これを列方向に重複して保持する．Inspector/Executor 戦略を適用している場合は，インデックス配列を全てのプロセッサでコピーして保持する．

また，並列化の対象となるのは，主に分解を行う i,j の 2 重ループである．変数 k は，この並列ループ内の loop invariant である．ゆえに，以下のようにブロードキャストのコードが生成される．

- a(pivot(k),*) を保持しているプロセッサが，これらの配列の要素をブロードキャストする（*は全ての要素を表す）．このプロセッサでは，全てのイタレーションが *local_iter* に含まれる．
- その他のプロセッサでは，ブロードキャストされてきた a(pivot(k),*) を受信する．全てのイタレーションが *nonlocal_iter* に含まれる．

5.1.2 実行結果

AP1000 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．プロセッサ台数は 1 台から 512 台まで 2 倍きざみとし，測定区間に入る直前に，全プロセッサ間で同期をとり，測定区間を終了するのが一番最後のプロセッサの実行時間を，全体の実行時間とした．また，AP1000 の高速メッセージハンドリング機構であるラインセンド，バッファレシーブ [9] を使用した．図 6に $N = 256$ の実行時間を示す．

5.1.3 考察

図 6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，この戦略を適用しなかった単純法の 42 倍（プロセッサ数 512）高速化された．また，全検査法では，同じ条件で実行時間が 11 倍高速化された．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 倍まで高速化されている．

ここで，分解部分の i,j のループの実行時間を推定する．逆インデックス法では，*local/ nonlocal_iter* を逆インデックス配列を参照するだけで実行すべきイタレーションが確定するため，配列のサイズを N，プロセッサ数を P とすると，N/P 回のイタレーションで実行できる．これに対し，全検査法では，*local/nonlocal_iter* を求めるために，インデックス配列 pivot の全ての要素の参照を必要とする．そこで，プロセッサ数に関係なく $N/\sqrt{P}$ 回のイタレーションの実行が必要である．また，Inspector/Executor 戦略を用いない単純法では，$\sqrt{N/P}$ 回のブロードキャストが必要となる．

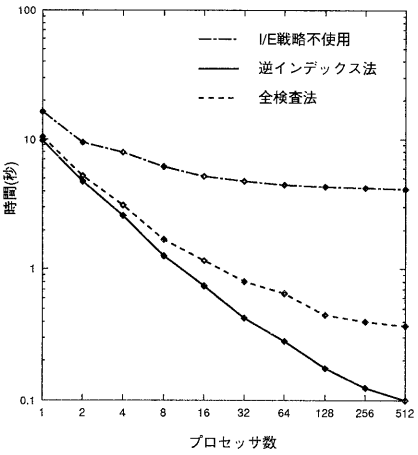

図 6: LU 分解の実行時間 (256×256)

ゆえに，単純法に比べて Inspector/Executor 戦略を適用した場合に，ブロードキャストが不要となるため，実行時間の大幅な短縮が行われる．また，逆インデックス法と全検査法を比べると，ループのイタレーション回数がそれぞれ N/P 回と $N/\sqrt{P}$ 回であるため，プロセッサ数が増加するにつれて，実行時間の差は大きくなる．ゆえに，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．

5.2 不規則疎行列とベクトルの積

アルゴリズムは，Insepctor/Executor 戦略を用いない単純法と，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．

5.2.1 データの分割と並列化

図 5に示すように行方向に BLOCK 分割する．また，1 次元ベクトル b,x を格納する配列 b,x も BLOCK 分割する．ただし，全検査法では各プロセッサが nzu, iu, nzl, il の全要素を保持する．

この分割では，i のループが DOALL 型となり，Insepctor/Executor 戦略を用いる場合は，i のループを並列に実行する．しかし，Insepctor/Executor 戦略を用

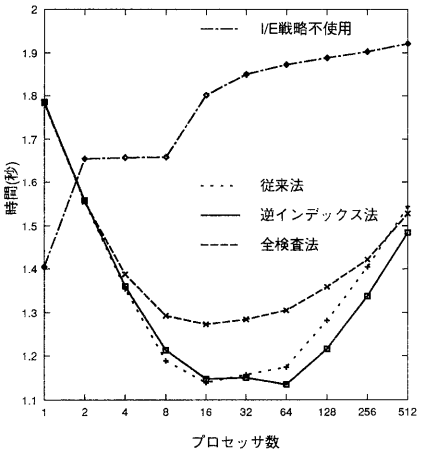

図 7: SMVM の実行時間

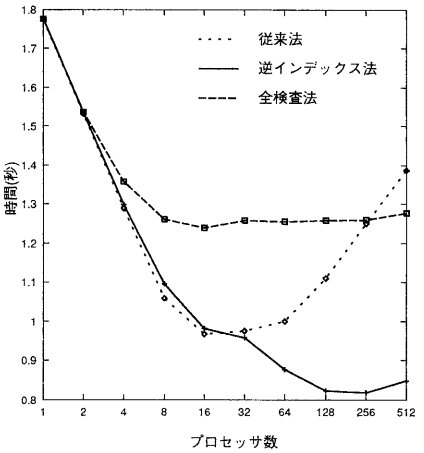

図 8: SMVM の insepctor の実行時間

いない単純法では，i のループを並列化せずに実行し，インデックス参照が行われる配列の要素 b(il(i,j))，b(iu(i,j)) がアクセスされるたびに il，iu の値をブロードキャストしたのちに，b の値が送受信される．

5.2.2 実行結果

AP1000 において，LU 分解と同様の条件で疎行列と 1 次元ベクトルの積を求めるプログラムの実行時間を測定した．実行時間の測定区間は，プロセッサ内での行列積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．

図 7に行列のサイズが $N = 1K$ の場合の SMVM 全体の実行時間を示す．また，inspector のみの実行時間を図 8に示す．

5.2.3 考察

単純法との比較 図 7から，Inspector/Executor 戦略を用いた全てのアルゴリズムで実行の高速化が達成されていることがわかる．これが最大となるのは，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．同じ条件で，従来の inspector では 1.59 倍まで高速化された．全検査法では単純法に比べて 1.4 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，同じ条件で，1.09 倍まで高速化されている．

ここで，insepctor の実行時間を定式化する．配列データのサイズを N，プロセッサ数を P，*send, recv* のリストを求めるループの 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，メッセージの送受信にかかる平均時間を T_M とすると，従来法，逆インデックス法および全検査法ではそれぞれ，

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

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

$$\text{全検査法} = 2NT_L + T_I \quad (3)$$

となる．

図 8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．これは，プロセッサ台数が増加するにつれて式 1の第 1 項の値は減少するが，第 2 項の全対全通信のコストが増大するためである．式 1を微分すると，$P = \sqrt{NT_L/T_M}$ のときに最小値を持つことがわかる．実際には，ループの実行時間やメッセージの受信待ちなどの時間はデータの性質によるが，最小値を持つことはこの式から説明される．

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 の実行時間が減少する．これは，インデックス配列の検査をするループの繰り返し回数が，式 2の第 1 項に示すようにプロセッサ数の増加に伴い減少するためである．従来法のようにプロセッサ台数に比例する項がないため，プロセッサ台数を増加することによる速度の向上が得られる．図 8で，プロセッサ数が 512 となると実行時間が再び増加に転じているのは，行列のサイズ (1024) が小さ過ぎるためである．

一方，全検査法では全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．式 3にもプロセッサ数 Pが含まれていないため，理論式と測定値がほぼ一致しているといえる．逆インデックス法と比べると，時間が長いが，プロセッサ数が増加すると，従来法よりは時間が短縮されている．なお，プロセッサ数が少ない場合は，*send/recv_list* で扱うリストの要素数が増加するため，リストへの要素の追加のオーバーヘッドが大きくなっている．

6 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 Inspetor/Executor 戦略の inspector 部分を高速化，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，逆インデックス法が従来の手法よりも高速であることが示された．特にプロセッサ台数が増加したときに有利である．

アルゴリズムの適用範囲の点では，全検査法では，制限がなく，逆インデックス法では，従来適用できなかった，ピボッティング付き LU 分解に適用可能となった．

今後の課題は，*send_list*, *recv_list* などのリストの操作をより高速なものにして inspector のオーバーヘッドを小さくすること，様々な例題について逆インデックス配列の生成可能性を調べ，その生成パターンを分類し，コンパイラへの指定法を検討することである．

謝辞

テストプログラムの実行にあたり，高並列計算機 AP1000 の実行環境を御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御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，日頃より有益な御意見をいただく富田研究室の諸氏に感謝致します．

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] Joel Saltz and Piyush Mehrotra, editors. *Languages, Compilers and Run-Time Environments for Distributed Momory Machines*. Elsevier Science Publishers B. V., 1992.

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

[3] Geoffrey C. Fox, Mark A. Johnson, Gregory A. Lyzenga, Steve W. Otto, John K. Salmon, and David W. Walker. *Solving Problems on Concurrent Processors*, Vol. 1. Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988.

[4] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler optimizations for Fortran D on MIMD distributed-memory machines. In *Proceedings of Supercomputing '91*, pp. 86–100, Albuquerque, NM, November 1991.

[5] Charles Koelbel and Piyush Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Trans. Parallel and Distributed Syst.*, Vol. 2, No. 4, pp. 440–451, October 1991.

[6] 窪田昌史, 大野和彦, 三吉郁夫, 森眞一郎, 中島浩, 富田眞治. 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 –. 情処研報 92-ARC-97-6, 情報処理学会, December 1992.

[7] Seema Hiranandani, Ken Kennedy, and Chau-Wen Tseng. Compiler support for machine-independent parallel programming in Fortran D. In Saltz and Mehrotra [1], pp. 139–176.

[8] 小国力, 村田健郎, 三好俊郎, J. J. Dongarra, 長谷川秀彦. 行列計算ソフトウェア. 丸善, 1991.

[9] 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, pp. 195–202, June 1992.

付録：ハンドコンパイルによるテストコード

```
for(k=0; k<size; k++) {
  pivot[k] = k;
  if (cidy == (k%ncely)) {
    inv[k/ncely] = k;
  }
}

for(k=0; k<size; k++) {
  k_owner = k%ncelx;
  k_loc   = k/ncelx;

  /* ピボットの選択 */
  if (cidx == (k%ncelx)) {
    maxpv = 0.;
    maxrow = k;
    for (i = 0;i < size_y;i++) {
      if ((inv[i] >= k) &&
          (fabs(a[i][k_loc]) > fabs(maxpv))) {
        maxpv = a[i][k_loc];
        maxrow = inv[i];
      }
    }
    y_damax(maxpv, maxrow, &maxpv, &maxrow);
  }
  x_brd(k_ower, &maxrow, sizeof(int));

  /* ピボットの交換 */
  if (maxrow != k) {
    lv = pivot[k];
    pivot[k] = pivot[maxrow];
    l = pivot[maxrow];

    if (cidy == (l_owner=l%ncely)) {
      l_loc = l/ncely;
      inv[l_loc] = k;
    }
    pivot[maxrow] = lv;
    l = lv;
    if (cidy == l_owner) {
      inv[l_loc] = maxrow;
    }
  }
  pivot_k_owner = pivot[k]%ncely;
  pivot_k_loc   = pivot[k]/ncely;
  if ((cidy == pivot_k_owner) &&
      (cidx == k_ower)) {
    maxpv = a[pivot_k_loc][k_loc]
      = 1.0 / a[pivot_k_loc][k_loc];
  }

  /* ピボット列の更新，ブロードキャスト */
  y_brd(pivot_k_owner, &maxpv, sizeof(FDATA));
  if (cidx == k_ower) {
    for (i=0; i<size_y; i++) {
      if (inv[i] >= k+1) {
        bl[i] = a[i][k_loc] *= maxpv;
      } else {
        bl[i] = 0.0;
      }
    }
  }
  x_brd(k_ower, bl, size_y * sizeof(FDATA));

  /* ピボット行のブロードキャスト */
  sx = (cidx < ((k+1)%ncelx)) ?
    (k+1)/ncelx + 1 : (k+1)/ncelx;
  if (cidy == pivot_k_owner) {
    for (i=sx; i<size_x; i++) {
      bu[i] = a[pivot_k_loc][i];
    }
  }
  y_brd(pivot_k_owner, &bu[sx],
       (size_x-sx)*sizeof(FDATA));

  /* 分解 */
  for (i=0; i<size_y; i++) {
    if (inv[i] >= k+1) {
      for (j=sx; j<size_x; j++) {
        a[i][j] -= bl[i] * bu[j];
      }
    }
  }
}
```

逆インデックス法を適用したLU分解のコード

```
  /* send_list の作成 */
  l_size = size/ncel;
  for (i=0; k<l_size; i++) {
    for (k=0; k<nzu[i]; k++) {
      j = iu[i][k];
      dest = j/l_size;
      if (dest != p) {
        append(dest, i);
      }
    }
  }

  /* recv_list の作成 */
  for (i=1; i<l_size; i++) {
    local_flag = TRUE;
    for (k=0; k<nzl[i]; k++) {
      j = il[i][k];
      src = j/l_size;
      if (src != p) {
        local_flag = FALSE;
        append(src, il[i][k]);
      }
    }

    /* local/nonlocal_iter の作成 */
    if (local_flag == TRUE) {
      append(local_iter, i);
    } else {
      append(nonlocal_iter, i);
    }
  }
```

逆インデックス法を適用したSMVMのinspectorのコード