不規則アクセスを伴うループの
並列化コンパイル技法

A Parallelizing Compiler Technique
for Loops with Irregular Accesses

窪田 昌史
指導教官 富田 眞治

1993 年 2 月 15 日

目 次

内容梗概

本論文では，分散メモリ型の並列計算機に対する並列化コンパイラの，SPMD (Single Program Multiple Data Stream) コード生成技法について述べる．

ソースプログラムの並列化対象ループの中に，インデックス配列による間接アクセスが存在すると，不規則なアクセスパターンを生じる．これを単純に並列化すると，アクセスすべきデータを所有するプロセッ

サと，そのプロセッサ内でのデータのアドレスを実行時に解析して，メッセージ通信を行わなければならず，多大なオーバーヘッドを生ずる．この問題の解決法として，データ・アクセスに関する実行時解析をまとめて行う*inspector*と，データの通信と本来のループ内の命令を実行する*executor*のコードとを生成する手法が提案されてきた．しかしこの手法では，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，本論文では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．

逆インデックス法では，インデックス配列を写像とみなし，その逆写像を実行時解析で利用する．この手法を用いると，inspectorでのプロセッサ間通信が全く不要となるため，実行時間を大幅に短縮することができる．同時に，従来の手法ではInspector/Executor戦略を適用できなかったピボッティング付きのLU分解などのプログラムに対するコードを生成することも可能となる．

ここで，逆インデックス法によるinspectorを生成するには，逆インデックス配列を効率的に求められることが必要である．インデックス配列の性質が単射であれば逆が一意に定まり，また，全射であれば逆インデックス配列を格納する記憶領域の大きさがコンパイル時に確定する．従って，このような条件を満たすインデックス配列に対しては，その逆を効率的に求めることが可能である．

一方，全検査法では，各プロセッサがインデックス配列を全て検査することにより，通信を不要としている．この方法では，インデックス配列のコピーを，全てのプロセッサで保持するため，インデックス配列の性質によらずinspectorを生成できる．

さらに，これらの手法の有効性を高並列計算機AP1000上で評価した．この結果，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最大41倍，全検査法で最大11倍の高速化が達成できた．また，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，不規則疎行列とベクトルの積を求めるプログラムで，最大1.6倍まで高速化された．

この実行結果の解析によって，不規則アクセスを伴うループの高速化にInspector/Executor戦略が有効であること，従来のinspectorアルゴリズムよりも逆インデックス法の方が高速であること，特にプロセッサ台数が増加したときに有利であるという結論に達した．

Abstract

In this paper, the author focuses on parallelizing compiler techniques which generate SPMD (Single Program Multiple Data stream) codes for distributed memory computers.

Indirect accesses with index arrays in target parallel loops bring about irregular access patterns. In naive parallelization for such loops, run-time analysis is required to determine the processor owning data to be accessed and the address of the data. And then the inter-processor communication to access the data is carried out. Since the analysis is performed just before the data is accessed, the latency of the communication cannot be hidden, and the performance is terribly degraded. To solve this problems, code generating technique based on Inspector/Executor strategy has been proposed. This technique generates an *inspector*, which collectively analyzes whole data accesses in the loop and an *executor*, which performs data communications and the actual execution of the loop. In this technique, however, the inspector must perform all-to-all global communications. Furthermore, a major class of programs, including the LU decomposition with pivoting, cannot be compiled.

In order to resolve these drawbacks, the author proposes two inspector algorithms, *inverse index method* and *exhaust inspection method*.

In the inverse index method, an index array is regarded as a mapping, and the inverse of the mapping is exploited at the run-time analysis. With this method, all the communications in the inspector are eliminated and the execution time is shortened substantially. In addition, it becomes possible to generate parallelized codes for programs such as the LU decomposition with pivoting, to which the conventional Inspector/Executor strategy has not been applicable.

In order to generate the inspector with the inverse index method, it is necessary to obtain the inverse array efficiently. If an index array is an injection, the inverse corresponds definitely. On the other hand, if an index array is a surjection, the size of memory area for the inverse array is determined at compile-time. For index arrays satisfying such conditions, it is possible to obtain the inverses efficiently.

In the exhaust inspection method, each processor keeps a copy of the index array to analyze the access pattern without communication. Therefore this method doesn't need inverse indices and is applicable to any types of programs regardless of the property of the index arrays.

The author 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 41 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.

As the result of the evaluation, it is concluded that the Inspector/Executor strategy is effective for fast implementation of parallelized loops with irregular accesses. It is also proved that the codes with the inverse index method are faster than those with conventional one, especially when the number of processors is increased.

1 緒論

科学技術計算に代表される高性能計算 (High Performance Computing) においては，多大な計算量が必要とされる．計算機が誕生して以来，科学技術計算からの計算機の処理速度の向上への要請が，高速な計算機を生み出してきた．

現在では，演算パイプライン方式のベクトル計算機が科学技術計算に多用されているが，この方式によるさらなる高速化には限界があると言われている．そのため，多数のプロセッサを結合した並列計算機の研究，開発が盛んに行われており，中でも，分散メモリ型の並列計算機については，多数の商用機が開発されている．

しかし，分散メモリ型並列計算機のハードウェアの進歩に比べ，そのソフトウェア技術は遅れをとっている．そのため，計算機のユーザのプログラム作成に負担を課す結果となり，この方式の並列計算機の普及の妨げとなっている．

ソフトウェア技術に関する最大の問題は，共有メモリ空間をサポートするためのソフトウェアの技法が確立されていないことである．逐次プログラムは共有メモリ空間のモデル上で作成されており，そのプログラムを並列計算機上で実行するには，物理的な分散メモリ空間への効果的なマッピングが必要とされる．

この共有メモリ空間の効率的なサポートは，SPMD(Single Program Multiple Data stream) のモデルに基づく並列化コンパイラの研究における重要な課題の一つであり，現在までに Fortran D[1], Kali[2], PARTI[3], Data Parallel C[4], Crystal[5], DINO[6], SUPERB[7] などの並列言語，およびそのコンパイラシステムで，様々な試みがなされている．しかし，コンパイル時の解析によって得られる情報を活用する技法は数多く提案されているが，実行時解析が必要なプログラムの並列化には，多くの課題が残されている．

そこで本研究では，実行時解析を必要とする不規則アクセスを含むループに対するコード生成技法を取り上げる．

このようなループに対しては，Inspector/Executor というコード生成戦略が提案されてきた．本論文では，この戦略について考察し，実行時解析における全対全通信を不要にすること，適用可能なコードの制限を取り除くことを提案する．

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 によるコード生成戦略を紹介する．3章では，この戦略の inspector 部分の従来のアルゴリズムにおいて，全対全通信が必要である，適用可能なコードに制限があるという問題点を取り除く逆インデックス法と，全検査法という手法を提案する．さらに，それらが生成可能なソースコードについても議論する．4章では，提案した手法の疎行列の積や行列の LU 分解といった実際の例題への適用と，その性能評価について述べる．最後に 5章で総括を行い，今後の課題にも触れる．

2　並列化コンパイラシステム

2.1　並列化コンパイラシステムの概要

並列計算機上で稼働するプログラムの作成においては，

- 逐次プログラムをコンパイラシステムによって自動／半自動的に並列化する．
- 並列性を陽に記述可能な並列言語によるプログラムをターゲットマシンへコンパイルする．

という 2 つの異なったアプローチがとられている．前者は，既存の逐次プログラムを継続して利用できるという利点がある．これに対して，後者の並列言語 CSP[8]，ABCL/1[9]，Linda[10]，Orca[11] などでは，並列性が陽に記述できるので，アルゴリズムが本質的に並列である問題に対して用いられる．

科学技術計算では，逐次プログラムからデータ並列性が引き出せるため，前者のアプローチによるプログラムの並列化によって，高い性能を得ることが可能である．そこで，我々は，このアプローチに基づくプログラム並列化支援システムの開発を行っている [12]．この支援システムでは，依存解析などで検出される並列性とプログラマの指定による並列性とを統合して，高性能な並列プログラムを生成するシステムを構築することを目指している．

科学技術計算においては，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，配列の要素に，一斉に同一の操作を適用するというSIMD的なデータ並列性を容易に引き出すことができる．そこで，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モデルによるプログラムの並列化が有効となる．これをMIMD型の計算機で稼働させると，データの参照関係によってプロセッサ間の同期がとられるという，データフロー的な実行が行われ，SIMDモデルでの実行にもたらされる同期を排除できる．このようにSPMDのプログラムは，逐次プログラムからの変更が少なく，同時に並列化による高速化が得られるという特徴を持つ．

さて，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．

本研究の対象では，後者のコード生成におけるコンパイル技法を対象とする．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．

2.2 並列化コードの生成の基本戦略

2.2.1 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り付け
- 割り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り付け
  - 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，これらの基本戦略について述べる．

データ分割に対する前提条件　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[13]．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．

これらの分割は，各プロセッサへ処理量を均等に分配することを意図している．BLOCK分割では，隣接した配列の要素間の処理が同一のプロセッサで行われることで，通信量の低減を図り，通信が必要な場

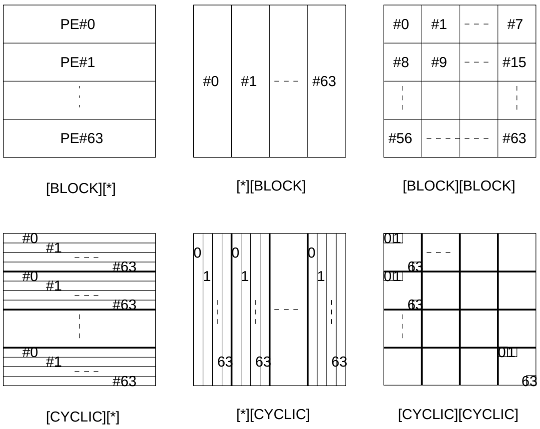


図 1: データ分割法 (プロセッサ数 64)

合でも，できるだけ隣接通信を多くすることができる．CYCLIC 分割では，処理が配列の一部分に偏るような場合に，処理ができるだけ均等に分配されることをねらったものである．

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．コンパイラへの指定は，配列に対するディレクティブによって行う．

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[1] 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*) がそのデータをアクセスする文を実行するように，コードを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プロセッサが実行する．従って，大規模な配列をアクセスするループでは分割された配列の要素の owner が，その要素をアクセスするイタレーションを実行する．

メッセージ通信のコードの挿入　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．

2.2.2　並列化コードの生成例

前節で示した基本戦略に基づく並列化コードの生成を，図 2 に示す DO ALL 型ループを 4 プロセッサで並列に実行する場合で説明する．

まず，配列 `a,b,c` が 25 要素づつ各プロセッサに割り付けられ，次に，この分割に従って，図 3のようなループのコードが生成される．即ち，プロセッサ 1 において逐次ループの `i=1` から `25` までのイタレーションが，同様にプロセッサ 2, 3, 4 においてそれぞれ `i=26` から `50`，`51` から `75`，`76` から `99` のイタレーションが実行される．

この例では，プロセッサ 1,2,3 において，`i=25` のときの `c(i+1)` を通信によって求める必要がある．これらの `c(i+1)` はそれぞれ，プロセッサ 2,3,4 における `c(1)` の値である．そこで，図 4のように，メッセージ通信のコードが挿入される．

```
 integer a(100),b(100),c(100)

decomposition d(100)
align a(i) b(i) c(i) with d(i)
distribute d(block)

do i=1,99
  a(i) = b(i) * c(i+1)
enddo
```

図 2: DO ALL 型の逐次ループ

```
 integer a(25),b(25),c(25)

{ イタレーション回数の決定 }
if PE_no = 4 then
   endi = 24
else
   endi = 25
endif

do i=1,endi
  a(i) = b(i) * c(i+1)
enddo
```

図 3: 並列化されたループ (1)

```
 integer a(25),b(25),c(25)

{ c(1) の送信 }
if pe = 2 or 3 or 4 then
   send(c(1), pe_number-1)
endif

if PE_no = 4 then
   endi = 24
else
   endi = 25
endif

do i=1,endi

  { c(26) の受信 }
  if pe = 1,2,3 and i = 25 then
     recv(c(26), pe+1)
  endif

  a(i) = b(i) * c(i+1)

enddo
```

図 4: 並列化されたループ (2)

2.3 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) [1]と呼ぶ．

部分ピボッティング付きの LU 分解[2]，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．

このような不規則アクセスを含む DOALL 型ループに対するコード生成を次のような例で説明する．

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

配列データの分割は，`a(i)`，`b(i)`，`index(i)` が，同じプロセッサに割り付けられるように行われるものとする．Data 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．

すると，ループの実行は，前節の例のようにプロセッサにイタレーションを分配することはできず，全てのプロセッサにおいて，以下の処理が各イタレーションごとに行われる．

1. 自分がインデックス配列 `index(i)` を保持していたら，それを全てのプロセッサにブロードキャストする．即ち全てのプロセッサが，どのプロセッサに `b(index(i))` が保持されているかという情報を得ることになる．

2. ブロードキャストされたインデックス配列の値をもとに，自分が当該配列の値 `b(index(i))` を保持しているかどうか検査し，保持していれば，`a(i)` を保持するプロセッサへその値を送信する．

3. `a(i)` を保持するプロセッサが，`b(index(i))` の値を受信し，代入文を実行する．

今の例は，代入文の右辺に現れる配列の添字式内に配列が存在する場合であったが，さらに左辺の配列の添字式にも配列が存在する場合についても考察する．

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

左辺の配列がインデックス参照されると，`a(index(i))` の owner も実行時にならなければ確定しないため，`b(index(i+1))` の送受信プロセッサとも実行時にならなければ確定しない．

ゆえに，全てのプロセッサにおいて，以下の処理が各イタレーションごとに行われる．

1. 自プロセッサがインデックス配列 `index(i)` を保持していたら，それを全てのプロセッサにブロードキャストする．

2. 自プロセッサがインデックス配列 `index(i+1)` を保持していたら，それを全てのプロセッサにブロードキャストする．

---

[1] irregular computaion[14]，irregular problems[3]，irregular communication patterns[15] などの用語も使われている．

[2] 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．

3. ブロードキャストされたインデックス配列の値 `index(i+1)` をもとに，当該配列の値 `b(index(i+1))` を保持しているかどうか検査し，保持していれば，ブロードキャストされたインデックス配列の値 `index(i)` を検査して，送信先プロセッサを確定し，`b(index(i+1))` を送信する．
4. `index(i)` を検査して，自分が `a(index(i))` の owner であるかどうか検査し，owner であれば，メッセージ `b(index(i+1))` を受信し代入文を実行する．

この例では，代入文の両辺にインデックス配列が存在するので，2 回のブロードキャストが必要となる．

このように，単純にコードを生成すると，データアクセスの度に，配列の値の通信を必要としない場合でも，インデックス配列のブロードキャストが必要となる．ブロードキャストのコストは非常に大きなものであり，性能低下を招く要因となる．

`index(i)` は，元の逐次プログラムにおいて，一種のアドレス変換テーブルの役割を持っている．分散メモリ型計算機では，このアドレス変換テーブルの情報をプロセッサ間で共有できないために，ブロードキャストによってそれを補っているということがいえる．

次節では，このような不規則アクセスが，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．

2.4 Inspector/Executor 戦略

プロセッサ間通信を伴うデータ参照では，データの送信者，受信者のプロセッサが確定されなければならない．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，送受信の両プロセッサは実行時でなければ確定できない．

2.3節で述べたように，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 [2]．本論文ではこのコード生成戦略を *Inspector/Executor*戦略と呼ぶ．図 5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．

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通信を伴うデータ・アクセス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．

2.5 用語・集合の定義

この節では，次章以降の議論のため，図 6に示す並列実行可能なコードについて，用語の定義と，文献 [2, 14] の定義に従った添字とイタレーションの集合を定義を行う．なお，図 6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．また，$\circ$は，右辺の項の和や積などの演算を表す．

用語の定義

iter_set($\vec{k}$) $\vec{l}$から$\vec{m}$まで，ステップ$\vec{s}$で取り得るベクトルの集合．

イタレーション番号 $iter\_set(\vec{k})$ の各要素．

イタレーション イタレーション番号$\vec{k}$において実行されるループボディ部．

inspector における実行時解析

1. PE 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 $send\_list(p,q)$
2. 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 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 $nonlocal\_iter(p)$

executor における通信，ループの実行

1. $send\_list(p,q)$ に基づいて，qへデータを送信する．
2. イタレーション $local\_iter(p)$ のループを実行する．
3. $recv\_list(p,q)$ のデータを qから受信する．
4. 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．

図 5: PE p における Inspector/Executor 戦略に基づくアルゴリズム

```
do $\vec{k}=\vec{l},\vec{m},\vec{s}$
  $X(g(\vec{k})) = Y_1(h_1(\vec{k}))\circ Y_2(h_2(\vec{k}))\cdots Y_n(h_n(\vec{k}))$
enddo
```

図 6: 並列ループ

項 配列の値の表現式$X(g(\vec{k}))$，$Y_i(h_i(\vec{k}))$．それぞれ，XやY_iと略記することもある．

添字式 $g(\vec{k}), h_i(\vec{k})$. 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，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．

添字式の逆関数 g, h_iの逆関数をg^{-1}, h_i^{-1}で表す．

インデックス配列 添字式内に現れる整数型配列

逆インデックス配列 g, h_i内のインデックス配列を関数とみなしたときの逆関数．

添字 イタレーション番号$\vec{k}$を添字式h_iなどに代入して得られる$\vec{j} = h_i(\vec{k})$．

データ分割関数

$$\mu_X(k) = (\delta_X(k), \alpha_X(k)) \tag{1}$$

配列Xの大域的な添字から，$X(k)$を所有するプロセッサ番号を返すownership関数δと，そのプロセッサ内での添字を返すlocation関数αからなる[3]．

ownership 関数 $X(k)$を所有するプロセッサ番号を返す．配列XのサイズをN，プロセッサ数をPとすると，BLOCK分割では，

$$\delta_X(k) = \lfloor k/\lceil N/P \rceil \rfloor \tag{2}$$

となり，CYCLIC分割では，

$$\delta_X(k) = k \bmod P \tag{3}$$

となる．

location 関数 $X(k)$のownerプロセッサ内での，添字を返す．BLOCK分割では

$$\alpha_X(k) = k \bmod \lceil N/P \rceil \tag{4}$$

となり，CYCLIC分割では，

$$\alpha_X(k) = \lfloor k/P \rfloor \tag{5}$$

となる．

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．

データ参照の集合

local(p) 左辺の項Xのうち，プロセッサpに所有される添字の集合であり，次のように求められる[4]．

$$local_X(p) = \{\vec{i} | \delta_X(\vec{i}) = p\} = \delta_X^{-1}(p) \tag{6}$$

右辺の項Y_jについても，同様に定義される．

---

[3] δ, αの定義は，C言語流にプロセッサ番号，配列の添字が0から始まる場合である．

[4] [14]では，$image_X(p)$

exec(p) プロセッサ p において実行されるイタレーションの番号の集合[5]．代入文の左辺のデータを所有するプロセッサが，その代入文を実行するという Owner Computes Rule の原則を適用すると，

$$exec(p) = g^{-1}(local_X(p)) \cap iter\_set(\vec{k}) \tag{7}$$

となる．

ref(p) プロセッサ p で実行されるイタレーションで参照される項の添字の集合．右辺の項の添字の集合は，

$$ref_{Y_j}(p) = h_j(exec(p)) \tag{8}$$

となる．左辺の項の添字の集合は，$local, exec$ の定義より，

$$ref_X(p) = g(exec(p)) = local_X(p) \tag{9}$$

となる．

deref(p) 右辺の項について，プロセッサ p 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[6]．

$$deref_{Y_j}(p) = h_j^{-1}(local_{Y_j}(p)) \cap iter\_set(\vec{k}) \tag{10}$$

全ての右辺の項の $deref_{Y_j}$ についての積集合を

$$deref(p) = \bigcap_j deref_{Y_j}(p) \tag{11}$$

と定義する．これはプロセッサ p に所有されているデータのみを参照するイタレーションを表す．

メッセージ通信を行う配列要素の添字の集合

send_list(p,q) プロセッサ p から，プロセッサ $q(\neq p)$ に送信すべき配列要素の添字の集合[7]．

$$\begin{aligned} send\_list_{Y_j}(p,q) &= local_{Y_j}(p) \cap ref_{Y_j}(q) & (12)\\ &= h_j(deref_{Y_j}(p) \cap g^{-1}(local_X(q))) & (13) \end{aligned}$$

これは，プロセッサ p が所有しており，プロセッサ q が参照する配列要素である．

また，あるイタレーションで送信すべき配列要素の添字の集合を，右辺の全ての項 Y_j についての和をとって，

$$send\_list(p,q) = \bigcup_j send\_list_{Y_j}(p,q) \tag{14}$$

と定義する．

recv_list(p,q) プロセッサ p が，プロセッサ $q(\neq p)$ から受信すべき配列要素の添字の集合[8]．

$$\begin{aligned} recv\_list_{Y_j}(p,q) &= ref_{Y_j}(p) \cap local_{Y_j}(q) & (15)\\ &= h_j(exec(p)) \cap local(q) & (16) \end{aligned}$$

これは，プロセッサ q が所有しており，プロセッサ p が参照する配列要素である．

---

[5] [14] では，$local\_iter\_set_X^p$

[6] [2] の定義とは違うものを指す．

[7] [14] では，$out\_index\_set_{Y_j}$

[8] [14] では，$in\_index\_set_{Y_j}$

右辺の全ての項についての和は，$send\_list(p,q)$ と同様に

$$recv\_list(p,q) = \bigcup_j recv\_list_{Y_j}(p,q) \tag{17}$$

と定義する．

なお，$recv\_list(p,q) = send\_list(q,p)$ の関係がある．

イタレーション番号の集合

local_iter(p) プロセッサ p 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番号の集合．

$$local\_iter(p) = exec(p) \cap deref(p) \tag{18}$$

nonlocal_iter(p) プロセッサ p だけでなく，他のプロセッサ $p \neq q$ が所有するデータも必要とするイタレーションの番号の集合．

$$nonlocal\_iter(p) = exec(p) - local\_iter(p) \tag{19}$$

3 不規則アクセスを伴うループのコード生成

本章では，不規則アクセスを伴うループに対するコード生成について述べる．3.1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．3.2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，3.3節において，逆インデックスの生成可能性について述べる．

3.1 inspector のアルゴリズム

本節では，図6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6の g, h_i の値が変更される場合は対象としていない．

なお，図6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，[2, 14] で提案された inspector のアルゴリズムと，本論文で提案する2種類の手法，逆インデックス法と全検査法による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，3.2節で述べる．

3.1.1 従来のアルゴリズム

図7にアルゴリズムを示す．このアルゴリズムの特徴は，次の通りである．

戦略 $recv\_list(p,q)$ は，プロセッサ p 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．

```
do $\vec{j} \in exec(p)$
  local_flag = true;

  /* $recv\_list_{Y_i}(p, \mathrm{src})$ を求める */
  do each $Y_i$
    src = $\delta_{Y_i}(h_i(\vec{j}))$;
    if (src != p) then
      local_flag = false;
      append $h_i(\vec{j})$ to $recv\_list_{Y_i}(p, \mathrm{src})$;
    endif
  enddo

  /* $local/nonlocal\_iter$ を求める */
  if (local_flag == true) then
      append $\vec{j}$ to $local\_iter(p)$;
  else
      append $\vec{j}$ to $nonlocal\_iter(p)$;
  endif
enddo

/* $recv\_list(p, q)$ を送信する */
do each PE q ($\neq$ p)
  send $recv\_list(p, q)$ to PE q;
enddo

/* $recv\_list(q, p)$ を受信する */
do each PE q ($\neq$ p)
  receive $recv\_list(q, p)$ from PE q;
enddo
```

図 7: ALG(1) 従来法のアルゴリズム

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．

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．

アルゴリズムの適用条件 各プロセッサ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} が求められることと，それに対応する右辺の添字 $ref_{Y_i}(p)$ が求められること．

このアルゴリズムでは，$send\_list$ を求めるために，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っている．そのため，プロセッサ数を P とすると，$P(P-1)$ 個のメッセージが通信されることになる．

また，2.3節で示した 2 つの例のうち，後者の両辺にインデックス配列が現れる場合は，`index(i)` の逆インデックス配列を求めなければ，プロセッサ p で実行されるイタレーションの集合 $exec(p)$ を得ることができない．[2] 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) などしかを許されておらず，$exec(p)$ は，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．

3.1.2 逆インデックス法によるアルゴリズム

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．

さらに，代入文の左辺だけでなく，右辺に現れるインデックス配列も求められる場合，自プロセッサが保持するデータが，どのプロセッサから参照されるかを表す集合 $deref$ も求めることができる．ゆえに，通信を行わずに $send\_list$ を求めることが可能となる．

よって，逆インデックス配列を積極的に活用すると，通信を不要とし、同時に，LU 分解などにも inspector を生成することが可能となる．このような inspector のアルゴリズムを図 8に示す．

戦略 逆インデックス配列を活用することで，$send\_list$,$recv\_list$ を各々のプロセッサが独自に作成する．各プロセッサ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．

アルゴリズムの適用条件 各プロセッサ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} が求められることと，右辺に現れる配列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．

LU 分解のように，右辺と左辺に現れるインデックス配列が同一である場合は多い．ゆえに，逆インデックス配列が求められれば，両辺の逆インデックス配列を用いるこの手法を用いることが有効となる．

3.1.3 全検査法によるアルゴリズム

逆インデックス配列は，全てのインデックス配列について求められるわけではない．そこで，逆インデックス配列を必要とせず，同時に，$send\_list$ を通信を行わずに求めるアルゴリズムを図 9に示す．

戦略 各プロセッサが，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，他のプロセッサ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 $local/nonlocal\_iter$ にイタレーションを加える．

という処理を繰り返す．

アルゴリズムの適用条件 $ref(p)$ 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．

このアルゴリズムは，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．

```
/* $send\_list_{Y_i}(p, \mathtt{dest})$ を作成する */
do each $Y_i$
  do $\vec{j} \in deref_{Y_i}(p)$
    dest = $\delta_X(g(\vec{j}))$;
    if (dest != p) then
      append $h_i(\vec{j})$ to $send\_list_{Y_i}(p, \mathtt{dest})$;
    endif
  enddo
enddo

do $\vec{j} \in exec(p)$ {
  local_flag = true;

  /* $recv\_list_{Y_i}(p, \mathtt{src})$ を作成する */
  do each $Y_i$
    src = $\delta_{Y_i}(h_i(\vec{j}))$;
    if (src != p) then
      local_flag = false;
      append $h_i(\vec{j})$ to $recv\_list_{Y_i}(p, \mathtt{src})$;
    endif
  enddo

  /* $local/nonlocal\_iter$ を求める */
  if (local_flag == true) then
    append $\vec{j}$ to $local\_iter(p)$;
  else
    append $\vec{j}$ to $nonlocal\_iter(p)$;
  endif
enddo
```

図 8: ALG(2) 逆インデックス法のアルゴリズム

```
/* 全てのイタレーションについて */
do $\vec{j} \in iter\_set(\vec{k})$
  local_flag = true;
  dest = $\delta_X(g(\vec{j}))$;
  do each rhs $Y_i$
    src = $\delta_{Y_i}(h_i(\vec{j}))$;

    /* $send\_list_{Y_i}(p, \texttt{dest})$ を作成する */
    if (src == p) then
      if (dest != p) then
        append $h_i(\vec{j})$ to $send\_list_{Y_i}(p, \texttt{dest})$;
      endif

    /* $recv\_list_{Y_i}(p, \texttt{src})$ を作成する */
    else  /* (src != p) */
      local_flag = false;
      if (dest == p) then
        append $h_i(\vec{j})$ to $recv\_list_{Y_i}(p, \texttt{src})$;
      endif
    endif
  enddo

  /* $local/nonlocal\_iter$ を求める */
  if (dest == p) then  /*
    if (local_flag == true) then
      append $\vec{j}$ to $local\_iter(p)$;
    else
      append $\vec{j}$ to $nonlocal\_iter(p)$;
    endif
  endif
enddo
```

図 9: ALG(3) 全検査法のアルゴリズム

表 1: inspector のアルゴリズムの比較

アルゴリズム		従来法	逆インデックス法	全検査法
inspector 内での通信		P-1	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

3.2 inspector アルゴリズムの比較

前節の 3 種類のアルゴリズムを，通信回数とイタレーション回数，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）．ただし，配列のデータのサイズが N，右辺の項の数が r，プロセッサ数が P である．

一番のオーバーヘッドとなる inspector 内での通信は，本稿で提案した逆インデックス配列法，全検査法では不要である．それに対して，従来法では，他の全てのプロセッサに対してメッセージを送信するので，$P-1$ 回の送信が必要であり，さらに，それらのメッセージの受信も $P-1$ 回必要である．ゆえに，通信メッセージの操作が $2(P-1)$ 回起こる．

ループの回数は，従来法では，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．逆インデックス法は，前半で，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，後半で，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る．前後半とも，rN/P程度であるから，$2rN/P$となる．全検査法では，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，全検査法のオーバヘッドが大きく，逆インデックス法と従来法では，オーダーはあまり差がない．

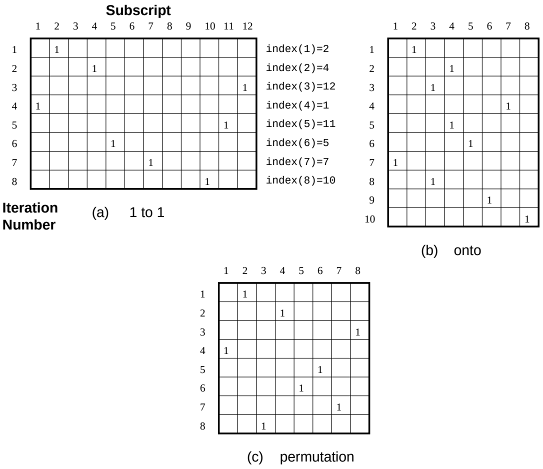
表 1のインデックスの条件の列は，両辺の添字式にインデックス配列が存在するソースコードに対して inspector を生成する際に必要となる順・逆インデックス配列を示している．all は，インデックス配列の全てを所有する必要があり，local は，自プロセッサにローカルなデータに対応する分だけを分割して配置されていれば良いことを示す．

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．
- $\vec{i} = g^{-1}(\vec{j})$ 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．

全検査法では，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．一方，逆インデックス法，従来法では，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逆インデックス法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，全検査法を採用することで，ループ回数は増えるが，通信の増大は避けられる．



(a) 1 to 1　(b) onto　(c) permutation

図 10: 生成可能な逆インデックス配列

3.3 逆インデックス配列の生成可能性

本節では，逆インデックス法の inspector を生成する上で必要不可欠な逆インデックス配列の生成可能性について考察する．

まず，インデックス配列を，イタレーション番号から配列の添字への写像とみなし，これを，図 10のような行列 (以下，写像行列と呼ぶ) で表現する．この行列において，行，列はそれぞれイタレーション番号と配列の添字を表し，例えば，図 10(a) の 4 行 1 列の 1 は，`index(4)=1` であることを示す．1 が入っている以外の要素は，全て 0 である．

以下，この行列を用いて，インデックス配列の性質を単射 (1 to 1) であるか，全射 (onto) であるか，という点から考察する．

単射　インデックス配列 `index` が単射であるとは，$\mathrm{a} \neq \mathrm{b}$ であれば，必ず，$\mathrm{index(a)} \neq \mathrm{index(b)}$ であることを指す．インデックス配列が単射であれば，写像行列の各列には，値が 1 の要素が高々 1 個しか存在しない．写像行列の列に値が 1 の要素が存在すれば，逆インデックスの値は，その行番号と等しく，一意に定まる．

インデックス配列が単射でなければ，写像行列の各列に，値が 1 の要素が複数存在する可能性がある．これは，逆インデックスの値が複数存在することを示す．この場合，linked list として逆インデックスの値を保持することが必要である．

全射　全射であるとは，値域の全ての要素 b に対して，$index(a) = b$ となる定義域の要素 a が存在することである．インデックス配列が全射であれば，インデックス行列の全ての列に，値が 1 の要素が少なくとも 1 つは存在する．ゆえに，値域である添字の個数，つまり，インデックス配列の個数分，逆インデックス配列を用意すればよい．

全射でなければ，写像行列に値が 1 の要素が存在しない列が現れる．この場合，逆インデックスの値は未定義である．このような列が多ければ，未定義の逆インデックスの値に対しても記憶領域を確保する必要があり，逆インデックス配列の記憶効率が悪くなる．

```
tmp = index(i);
index(i) = index(j);
index(j) = tmp;
```

(a) ソースコード

```
tmp = index(i);

index(i) = index(j);
inverse(index(j)) = i;

index(j) = tmp;
inverse(tmp) = j;
```

(b) 逆インデックスを生成するコード

図 11: 置換群の逆インデックス配列の生成

置換群 LU 分解などに現れるピボット配列は，分解が進むに従って，ある要素ともう 1 つの別の要素が交換される．この過程で生成される一連のピボット配列は，置換群をなす．これは，単射かつ全射（全単射）であり，しかも，定義域と値域の要素の個数が等しい．

ゆえに，

- インデックス配列と同じサイズで逆インデックス配列が格納できる．
- 逆インデックス配列の各要素は，必ずただ一つ存在する．

という性質がある．ゆえに，逆インデックス配列は，linked list を作る必要もなく，また，未定義の要素もないため，効率良く記憶できる．

置換群では，任意の置換は 2 つの値の交換（互換，transposition）を繰り返すことで求められる [16]．この性質を利用して，2 個の要素の交換が行われるときに，逆インデックス配列の値を，それにあわせて修正すれば，実行時に逆インデックス配列が生成できる．図 11に，そのコードを示す．

インデックス配列の値に重複がある場合 インデックス配列が 1 対 1 の写像ではない場合である．1 対 1 ではないことから，その逆写像は一般には生成できない．しかし，インデックス配列の値の重複の程度によっては，その生成が可能である．

有限要素法において処理される疎行列のインデックス配列では，ある節点と隣接する節点の数がわかっている場合が多い．ゆえに，隣接する節点の数の分だけ，linked list として逆インデックス配列を生成できる．

有限要素法では，節点間の隣接関係は強連結グラフであるため，節点 a から節点 b へのリンクがあれば，節点 b から節点 a へのリンクも存在する．これを表現する行列は，対称行列であり，しかも，写像行列そのものである．この性質から，上三角行列の行方向の情報の逆インデックス配列は，下三角行列の行方向の情報の逆インデックス配列で表される．

さらに，一般の場合でも．上三角行列の行方向の情報から，下三角行列の行方向の情報を生成する処理を実行時に行うことで，生成可能となる．問題となるのは，用意すべき領域の大きさが不確定であり，ア

クセスが効率的ではなくなることである．用意すべき領域があらかじめ分かっていれば，linked list を生成するかわりに，配列として保持できる．有限要素法などで用いられる疎行列では，１節点あたりのリンク数が決まっているため，配列に逆行列を保持できる．

4 Inspector/Executor 戦略の有効性の評価

本章では，不規則疎行列とベクトルの積，部分ピボッティング付き LU 分解の例題について，3章で述べた inspector のアルゴリズムを適用し，高並列計算機 AP1000[17] 上で実行時間を計測し，その結果をもとにアルゴリズムの有効性を検証する．

現在，SPMD コードを生成するコンパイラは作成中であるため，ハンドコンパイルによって，メッセージ通信のコードが入ったプログラムへと変換し，その実行速度を計測した．

Inspector/Executor 戦略に基づいて得られる並列化コードにおいては，*send*,*recv*,*local*,*non_local* のリストを扱う処理が必要である．そこで，リスト処理を行う実行時ライブラリを作成し，このライブラリを組み込んで，例題プログラムのハンドコンパイルを行った．

4.1 不規則疎行列とベクトルの積

4.1.1 不規則疎行列のデータ構造

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に格納するため，不規則アクセスが生じる．

有限要素法による構造解析，電子回路のシミュレーションで解かれる連立１次方程式の係数行列の多くは，大部分の行列要素が 0 の疎行列である．並列実行が必要な大規模行列では，疎行列を扱うことの方が多い．この行列全体の要素をメモリに格納することは，容量の点からも，計算時間の点からも，現実的に可能な限界を越えてしまう．

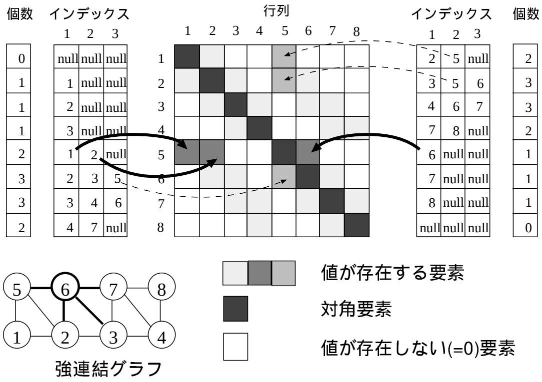
そこで，係数行列 A の非零要素のみを，配列に格納することが必要とされる．ここでは，[18, 19] などで紹介されている典型的な格納方法を採用する．仮定する係数行列 A は，対称，かつ非零要素の位置が不規則な疎行列である．

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．

また，対称行列であるため，`nzu(i)`，`iu(i,j)`，`au(i,j)` は下三角行列の第 `i` 列，`nzl(i)`，`il(i,j)`，`al(i,j)` は，上三角行列の第 `i` 列の非零要素の情報を表す．

図12に 8×8 行列の例を示す．第 2 行目には，対角線より右側の上三角行列に３個の非零要素が存在するので，`nzu(2)=3` となる．また，それらの非零成分の存在する列番号は，左から順に `iu(2,1)=3`，`iu(2,2)=5`，`iu(2,3)=7` である．`au(2,1)`，`au(2,2)`，`au(2,3)` にはそれぞれ対応する成分の値が入り，`ad(2)` には第２行の対角要素の値が入る．



個数	インデックス 1	2	3	行列	インデックス 1	2	3	個数
0	null	null	null	1	2	5	null	2
1	1	null	null	2	3	5	6	3
1	2	null	null	3	4	6	7	3
1	3	null	null	4	7	8	null	2
2	1	2	null	5	6	null	null	1
3	2	3	5	6	7	null	null	1
3	3	4	6	7	8	null	null	1
2	4	7	null	8	null	null	null	0

図 12: 疎行列のデータ構造

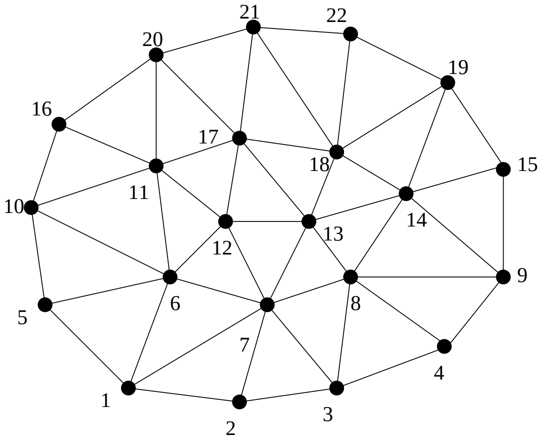


図 13: 不規則メッシュ
[18] より

構造解析や，電子回路のシミュレーションでは，ICCG 法 (Incomplete Choleskey and Conjugate Gradient method)，SOR 法 (Succesive Over Relaxation method) などの反復解法でこの疎行列が解かれる．反復解法においてメッシュの構造が変化しない場合，一度 inspector で求めたリストを用いて，解が収束するまで executor を実行する．ゆえに，Insepctor/Executor 戦略のの適用が有効となる．この節では，この反復解法の中で使用される基本的な処理であり，[20] などでその性能評価が報告されている係数行列とベクトルとの積のプログラムを 3章で述べた inspector アルゴリズムによるコードへとハンドコンパイルによって記述し，その評価を行う．

係数行列としては，不規則メッシュ図 13を仮定する．しかし，実際に用いた行列は，図 14のものである．これは現実に存在する不規則メッシュを作成するのが困難なためである．

図 15に不規則疎行列とベクトルとの積 (以下 SMVM:sparse Matrix Vector Multiplication と略す) を求めるプログラムを示す．

4.1.2 配列の分割

行列 $A\times$ベクトル $b =$ベクトル x を求める．ここで，A は 4.1.1節で述べたような行列である．行列を格納する配列は，図 16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

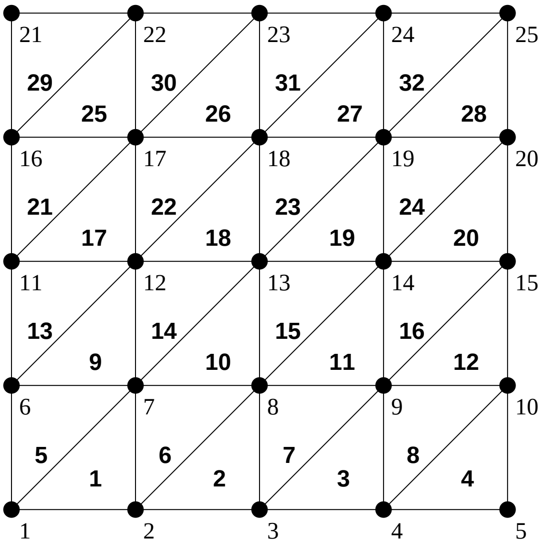


図 14: 有限要素法の領域分割
[18] より

```
decomposition  d(N)
align    with nzu(i),iu(i,*),au(i,*)  with d(i)
align    with nzl(i),il(i,*),al(i,*)  with d(i)
align    with 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 15: 不規則疎行列とベクトルの積を求めるプログラム

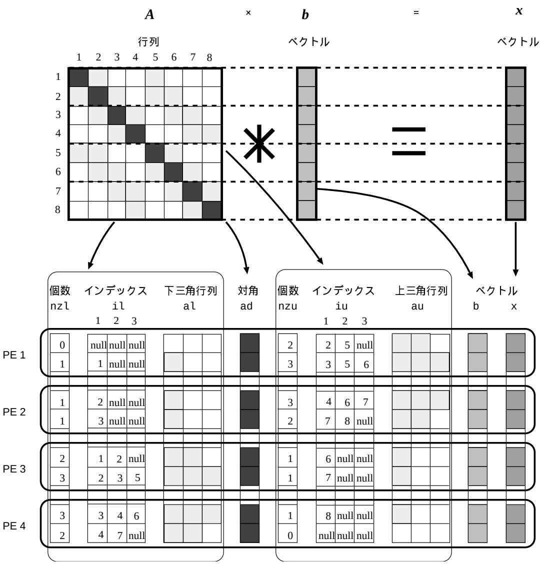


図 16: 疎行列の配列の分割

`b,x` も BLOCK 分割する．

4.1.3 コードの性質

`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．前章で示した inspector の逆インデックス法を適用するには，`il(i,j)` および `iu(i,j)` の逆インデックス配列が求められる必要がある．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る．以下，これを説明する．

図 16において，例えば，`j=5` のイタレーションでは `x(5)` を求めるために，`A` の第 5 行と `b` の 1,2,6 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．

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，$i = 2, 5$ のイタレーションにおいて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．

ゆえに，下三角行列においては，`iu(i,j)` を検査することで，*send_list* が作成され，上三角行列においては，`il(i,j)` を検査することで *send_list* が作成できる．

4.1.4 実行結果

AP1000 において，以下のような条件で SMVM プログラムの実行時間を測定した．

- 台数 (512,256,128,64,32,16,8,4,2,1)
- 行列のサイズ (32K,16K,4K,1K,256,64)
- AP1000 の高速メッセージ処理機構 [17] を使用

inspector のアルゴリズムに加え，2.3で示したような Inspector/Executor 戦略を用いない場合についても測定を行った（これを単純法と呼ぶことにする）．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．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．

表2から5に全体の実行時間を，図17にグラフ化したものを示す．また，表6から8には，inspector のみの実行時間を，図18にはそのグラフを示す．

4.1.5　考察

SMVM の実行時間についての考察を行う．

単純法との比較　図19に，SMVM を，Inspector/Executor 戦略を使用しない単純法と比較したグラフを示す．これらは，(単純法の実行時間）/（各アルゴリズムでの実行時間）によって求めたものであり，横軸にプロセッサ台数，縦軸に時間比をとっている．ゆえに，時間比が大きいほど，アルゴリズムが高速化されていることを示す．それぞれの折れ線は，$N \times N$行列のサイズ Nを表している．

ここで，単純法のコードは，2.3節で示したように，インデックス参照を行うデータ参照が起こるたびに，インデックス配列の値をブロードキャストして，送受信プロセッサを確定するというものである．プロセッサ数が少ない場合は，レシーブバッファがあふれるため，実行時間が測定できなかった．

これらの図からは，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，行列サイズ $N = 32K$，プロセッサ数32台，逆インデックス法で実行した場合で，1.96 倍となる．

Inspector/Executor 戦略を適用したアルゴリズムの内では，逆インデックス法と従来法とが数値的にも，グラフの形状でも似たような傾向を示している．これに対し，全検査法は，他の2つのアルゴリズムに比べ，高速化の比率が小さい．これらのアルゴリズム間の違いについては，後で詳しく考察する．ここでは，Inspector/Executor 戦略に共通の傾向について考察する．

図19のグラフに共通するのは，

- 行列のサイズが大きくなるにつれ，高速化の度合が大きい．
- プロセッサ数が増加するにつれて，高速化率がやや小さくなっている．

という傾向である．

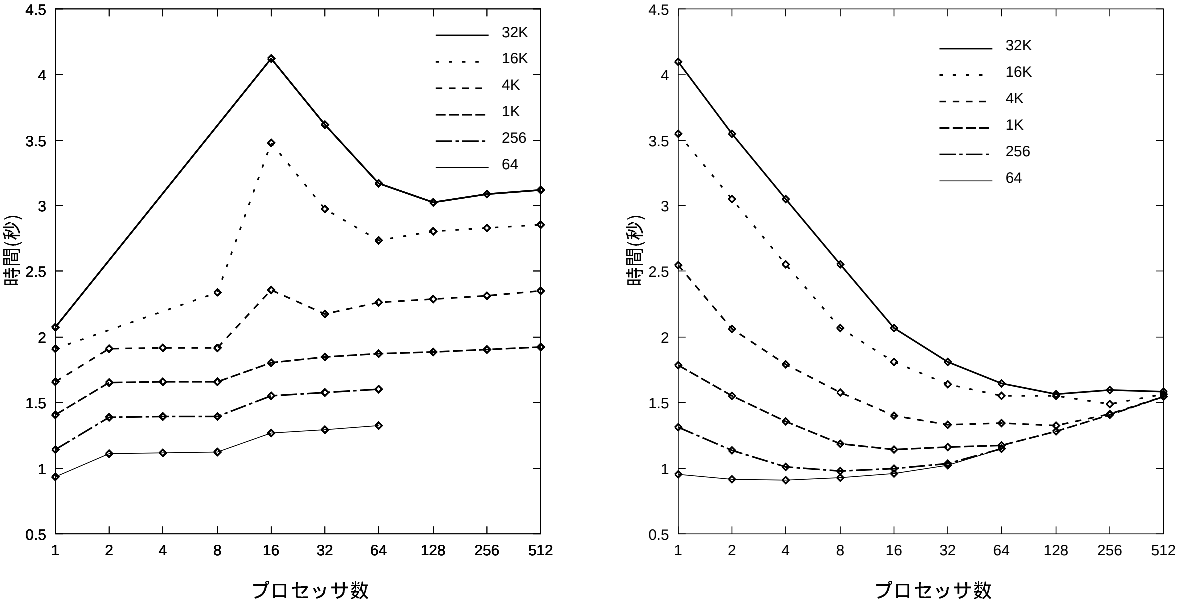
まず，行列のサイズと高速化の関係について述べる．Inspector/Executor 戦略を用いない単純法では，2.3節で述べたように，ループのイタレーション数に比例する回数のブロードキャストが起こる．行列のサイズが大きくなると，イタレーション数もそれに比例して増加するため，結局，ブロードキャストの回数も行列のサイズの比例して増加する．それに対して，Inspector/Executor 戦略を適用すると，ブロードキャストではなく，必要最小限の1対1の通信のみが行われるため，通信量が削減される．

また，プロセッサ数が増加するにつれて，高速化率が小さくなるのは，Inspector/Executor 戦略を適用しても，inspector および executor での通信回数はプロセッサ数に比例して増大するため，単純法に対する相対的な通信量の差が小さくなるためと思われる．

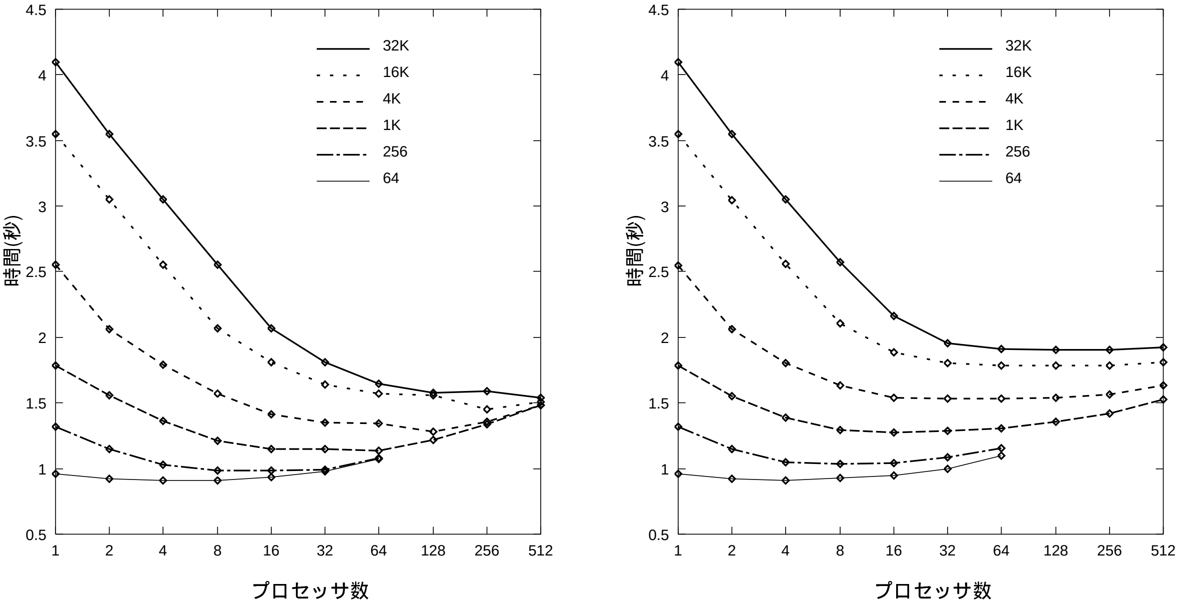
従来法による inspector の考察　ここでは，図18(a) をもとに，従来の inspector のコードについて考察する．

図18(a) 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てから再び増加する．例えば，行列のサイズが32K の場合は，プロセッサ数128 で，実行時間が最小となり，これ以上プロセッサを増やすと，実行時間は増加する．ここで，inspector の $recv\_list$ 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，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，

$$inspector(1) = \frac{NT_L}{P} + T_m(P-1) \tag{20}$$

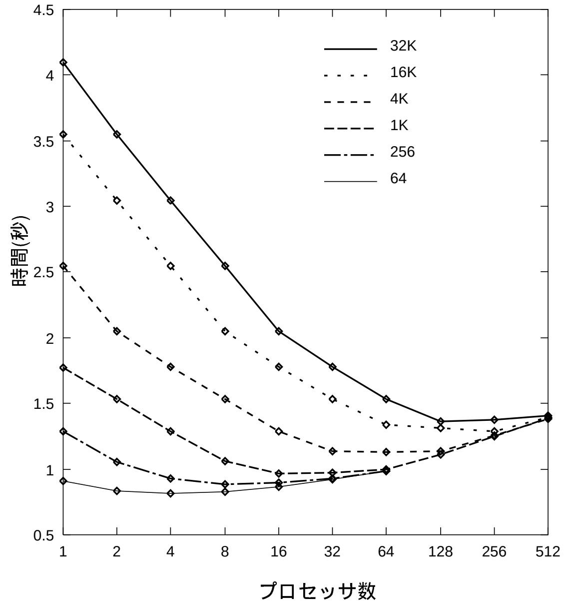


(a) 単純法　　(b) 従来法

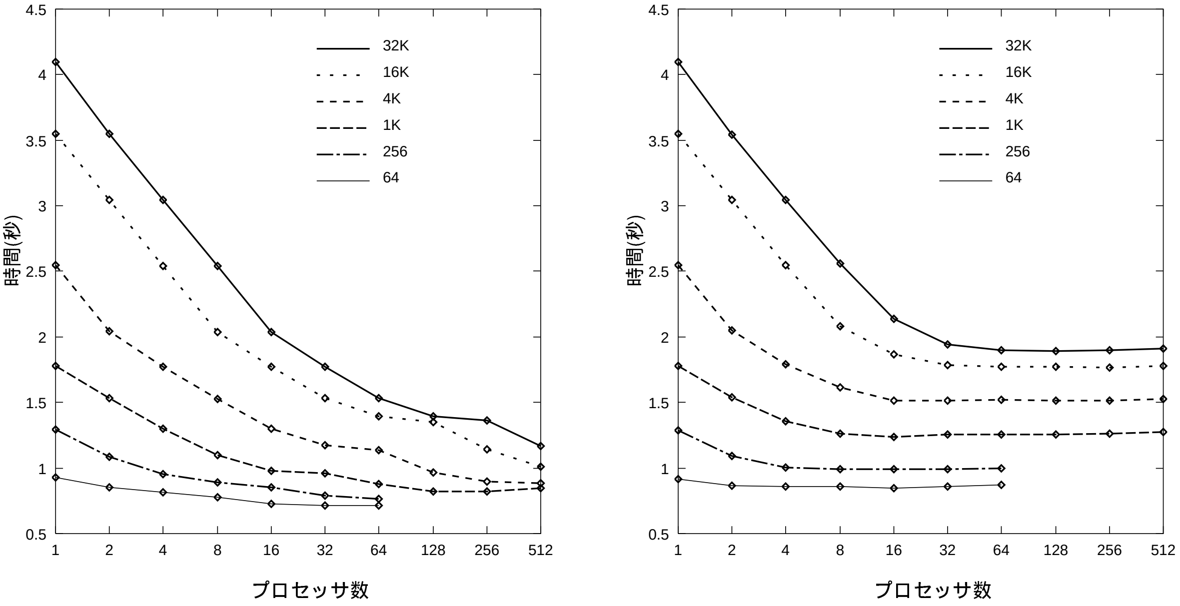


(c) 逆インデックス法　　(d) 全検査法

図 17: SMVM の実行時間



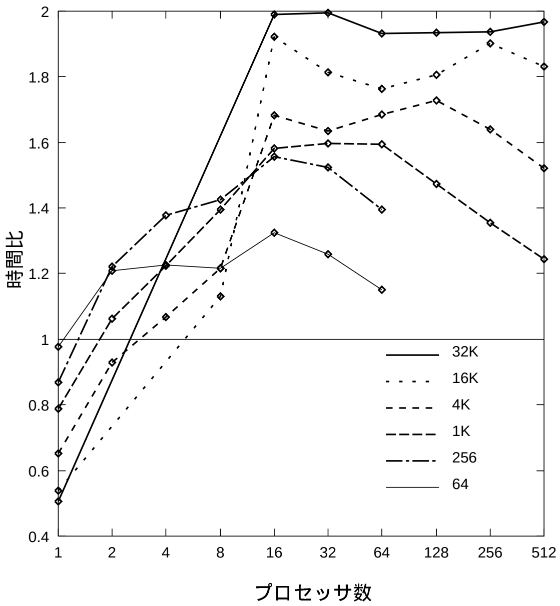
(a) 従来法



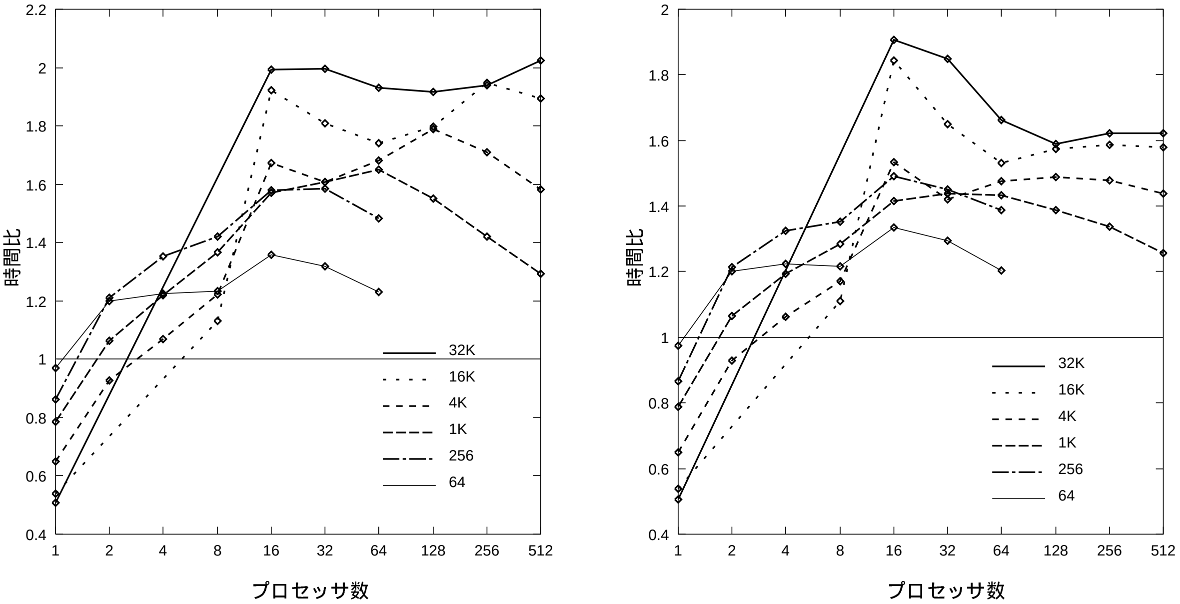
(b) 逆インデックス法

(c) 全検査法

図 18: SMVM の inspector の実行時間



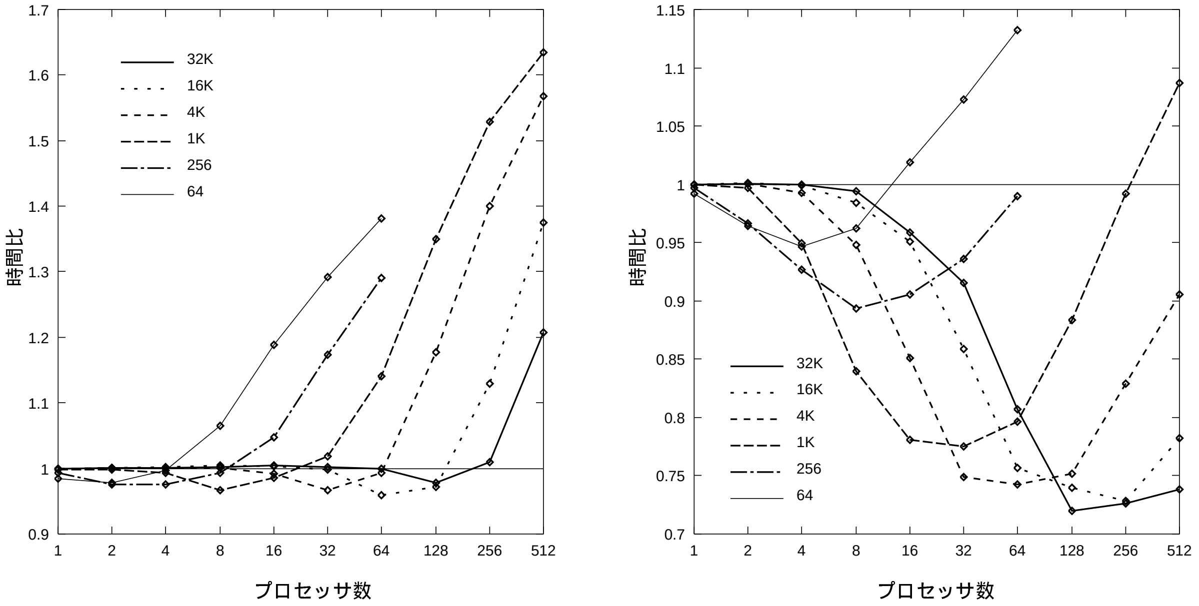
(a) 従来法



(b) 逆インデックス法

(c) 全検査法

図 19: SMVM の実行時間の比較 :単純法に対する比率



(a) 逆インデックス法　　　(b) 全検査法

図 20: SMVM の inspector の実行時間の比較 :従来法に対する比率

となる．この式の右辺の第 1 項は，プロセッサ台数に反比例するが，第 2 項は，プロセッサ台数に比例するため，プロセッサ台数の増加とともに，この式の値が第 2 項に支配されるためである．この式を微分すると，$P = \sqrt{NT_L/T_m}$のときに最小値となることがわかる．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．

逆インデックス法による inspector の考察　プロセッサ台数の増加に伴い，inspector の実行時間は減少する．これは，3.2 節で示したように，inspector 内のループのコストは，プロセッサ台数が増加すると，それに反比例して小さくなる，inspector 内で通信が起こらないため，従来法のようにプロセッサ台数に比例して実行時間が増大する要因がないという理由による．

全検査法による inspector の考察　プロセッサ台数の数にかかわらず，inspector 内のループのコストは一定である．行列サイズ $N = 32K$の場合で，プロセッサ数 64 台あたりから，inspector の実行時間は，ほぼ一定である．プロセッサ数が比較的少数の場合は，$send\_list$ ，$recv_list$の， 1 つのリストあたりの要素数が多いため，これらのリストにインデックスの値を追加するオーバーヘッドが大きいためである．リストの追加の際に，要素が昇順に並ぶようにするため，リストのサイズが大きくなるほど，リストへのインデックス配列の値の追加にコストがかかる．

従来法の inspector アルゴリズムとの比較　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図 20に示す．この図でも横軸がプロセッサ数，縦軸が高速化率である．これらはそれぞれ，(従来法の実行時間) / (逆インデックス法のあるいは，全検査法による実行時間) で算出したものである．このグラフでも，時間比が大きいほど，高速化されていることを示す．

図 20(a) からは，プロセッサ台数が増加するにつれて inspector が高速化されていることがわかる．512

プロセッサの場合は，行列の大きさ $N = 1K, 4K, 16K, 32K$ のとき，それぞれ $1.63, 1.57, 1.38, 1.21$ 倍までの高速化が達成されている．

しかし，プロセッサ数が少ない場合，最大 4%程度の性能低下も見られる．これは，プロセッサ数が少ない場合，従来法における通信回数（プロセッサ数に比例する）が少ないのに対し，逆インデックス法の $send\_list$ を求めるループの実行時間（プロセッサ数に反比例する）が長いためである．ゆえに，プロセッサ数が増加するにつれて従来法の通信オーバーヘッドが大きくなり，逆インデックス法において通信を不要としたことの効果が現れている．

図 20(b) では，行列の大きさが $N = 64, 1K$ の場合以外は，性能の低下が見られる．全検査法の inspector におけるループの実行時間は，従来法のコードにおける実行時間のプロセッサ台数倍であるため，従来法の通信よりも大きなオーバーヘッドとなっている．ただし，プロセッサ数を増加すると，従来法のオーバーヘッドは大きくなるので，全検査法との差は小さくなる．

SMVM の処理時間と inspector の処理時間との比率　Inspector/Executor 戦略を適用した従来法，逆インデックス法，全検査法について，SMVM 全体の実行時間に対する，inspector の実行時間の割合（inspector の実行時間）/（SMVM 全体の実行時間）を縦軸にとったグラフを図 21に示す．

executor におけるコードは，これらのアルゴリズムで共通のものを使用しているため，executor における処理量は等しい．ゆえに，図 21 のグラフにおいて，数値が小さくなるほど，inspector におけるオーバーヘッドは小さい．

これらのアルゴリズムに共通して，行列のサイズ N が小さいほど，inspector の割合が小さくなる傾向にある．

ただし，従来法では，inspector の割合は単調に低下してはいない．これは，プロセッサ数の増加による通信回数の増加が，プロセッサ数が増加したことによるプロセッサあたりの処理量の低下よりも，多大な影響を与えているためである．

これらのグラフの内では，逆インデックス法のアルゴリズムの数値が一番小さい．また，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 の割合が低下している．これは，inspector における処理量がプロセッサ数に反比例することによるものである．さらにプロセッサ数を増加することにより，さらに inspector の割合の低下が期待できる．

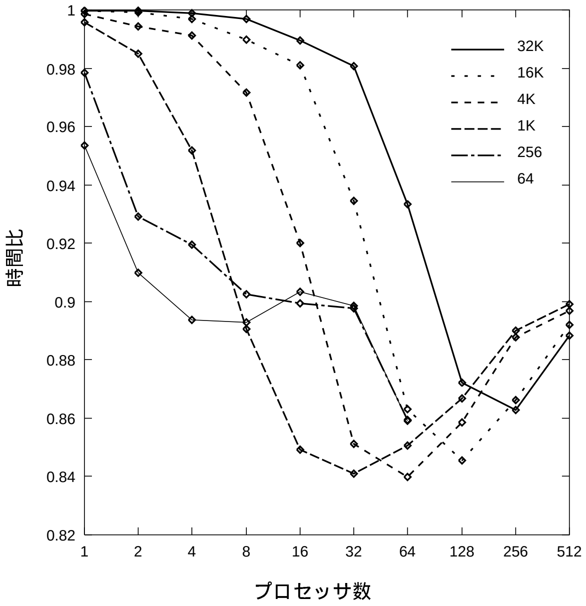
全検査法においても，行列のサイズが $N = 64, 256$ の場合は，inspector の割合の低下が見られる．しかし，行列のサイズがこれより大きくなると，inspector の割合は 99%前後で，あまり低下しない．これは，このアルゴリズムの inspector のループが行列のサイズに比例し，プロセッサ数が増加しても，処理量が変化しないためである．

まとめ　以上の考察より，次のことが言える．

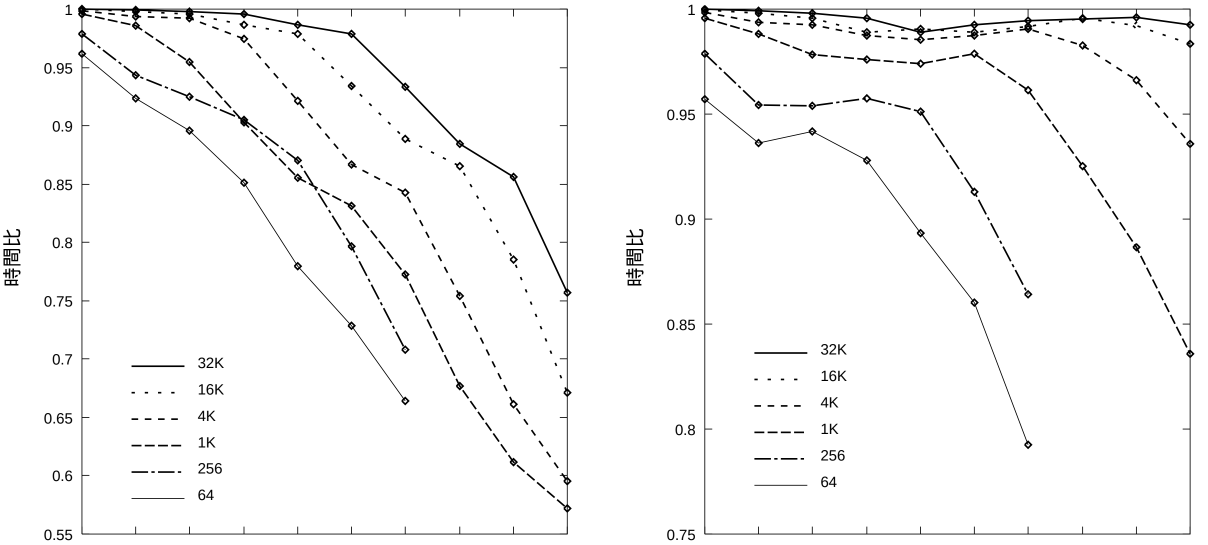
- 逆インデックス法のアルゴリズムは，プロセッサ台数の増加とともに，台数効果が得られる．行列のサイズの増加とともに，inspector のオーバーヘッドが大きくなる．
- 全検査法のアルゴリズムは，行列のサイズの増加とともに，inspector のオーバーヘッドが大きくなる．プロセッサ台数の増加による台数効果は得られない．
- 従来法のアルゴリズムは，プロセッサ台数の増加とともに，inspector のオーバーヘッドが大きくなる．

4.2　部分ピボッティング付き LU 分解

LU 分解のコードを図 22に示す．



(a) 従来法



(b) 逆インデックス法　(c) 全検査法

図 21: SMVM の実行時間と inspector の実行時間との比率

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

do k=1,N

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

  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 22: 部分ピボッティング付き LU 分解のコード

4.2.1　配列の分割

密行列を格納する行列 `A` を行方向，列方向の両方向について CYCLIC 分割する．ピボット行選択のためのインデックス配列 `pivot` 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．BLOCK 分割では，分解が進むにつれて，配列の添字の値が大きい要素を保持するプロセッサに処理が集中するため，このような処理の集中を防ぎ，負荷分散を実現するため，CYCLIC 分割が行っている．

4.2.2　コードの性質

図 22の LU 分解を並列化する場合，インデックス配列 `pivot` が permutaion 関数であるため，3.3 節で述べたように，逆インデックス配列が生成できる．

また，i のループと i, jの 2 重ループにおいて，変数$\vec{k}$は，並列ループ内において loop invariant である．このとき，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．

これは，2.5節で述べた集合の定義を loop invariant を含む表現式に対応して修正することで得られる．

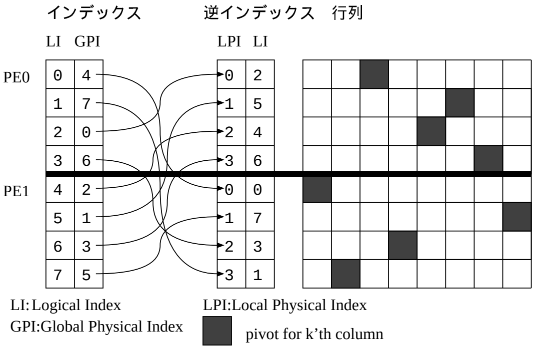


図 23: LU 分解におけるインデックス参照

$Y_i(h_i(\vec{k}))$ に対しては，

$$ref(p) = \begin{cases} Y_i(h_i(\vec{k})) & \text{if}\, exec(p) \neq \phi \\ \phi & \text{if}\, exec(p) = \phi \end{cases} \tag{21}$$

$$deref(p) = \begin{cases} iter\_set(\vec{k}) & \text{if}\, \vec{k} \in local(p) \\ \phi & \text{if}\, \vec{k} \notin local(p) \end{cases} \tag{22}$$

$$recv\_list(p,q) = \begin{cases} \vec{k} & \text{if}\, \vec{k} \in local(q) \\ & \text{and}\, exec(p) \neq \phi \\ \phi & \text{otherwise} \end{cases} \tag{23}$$

$$send\_list(p,q) = \begin{cases} \vec{k} & \text{if}\, \vec{k} \in local(p) \\ & \text{and}\, exec(q) \neq \phi \\ \phi & \text{otherwise} \end{cases} \tag{24}$$

のようになる [2]．一般的には，図 8のアルゴリズムにおいて，送信先のプロセッサ番号 `dest` を全てのプロセッサとすればよい．従って，図 8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．

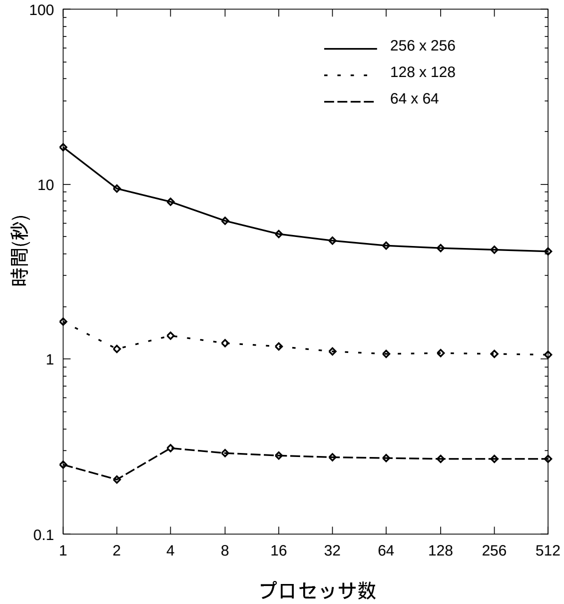
4.2.3 実行結果

AP1000 において，プロセッサ台数，行列のサイズを次のように指定して実行した．

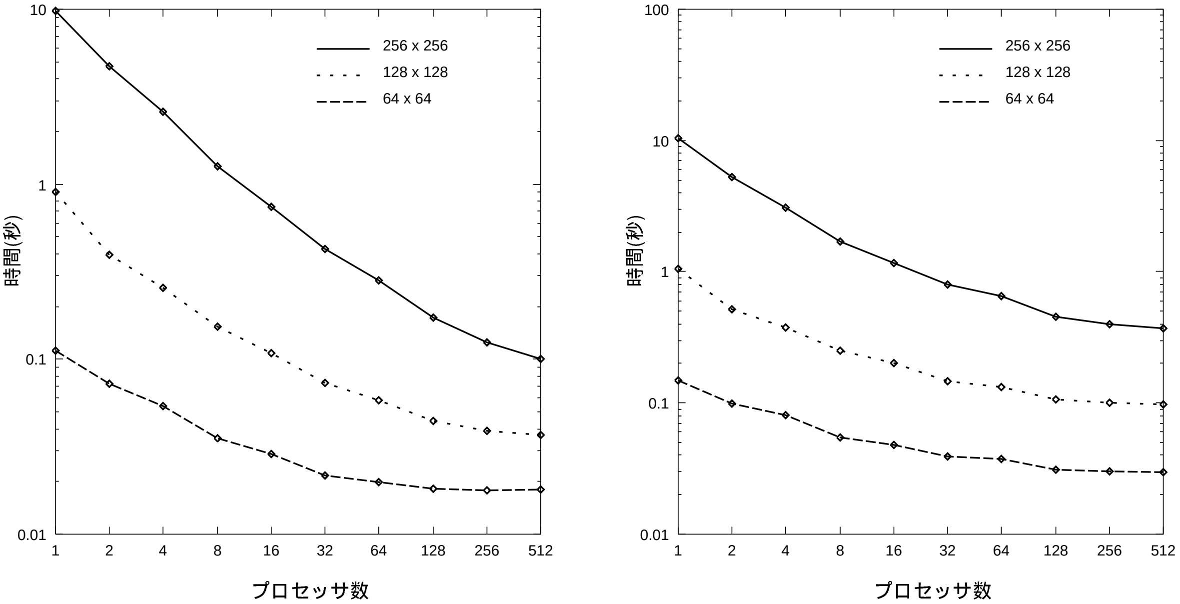
- 台数 (512,256,128,64,32,16,8,4,2,1)
- 行列のサイズ (64,128,256)

測定には，SMVM と同様に AP1000 の高速メッセージ処理機構を使用している．実行時間の測定区間は，プロセッサ内で LU 分解のサブルーチン全体とし，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．

表 9から 11に，LU 分解の実行時間を，図 24に，それをグラフ化したものを示す．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と，Inspector/Executor 戦略を適用しなかった場合についての実行時間を示す．なお，この例では，従来の方法による inspector は生成できない．

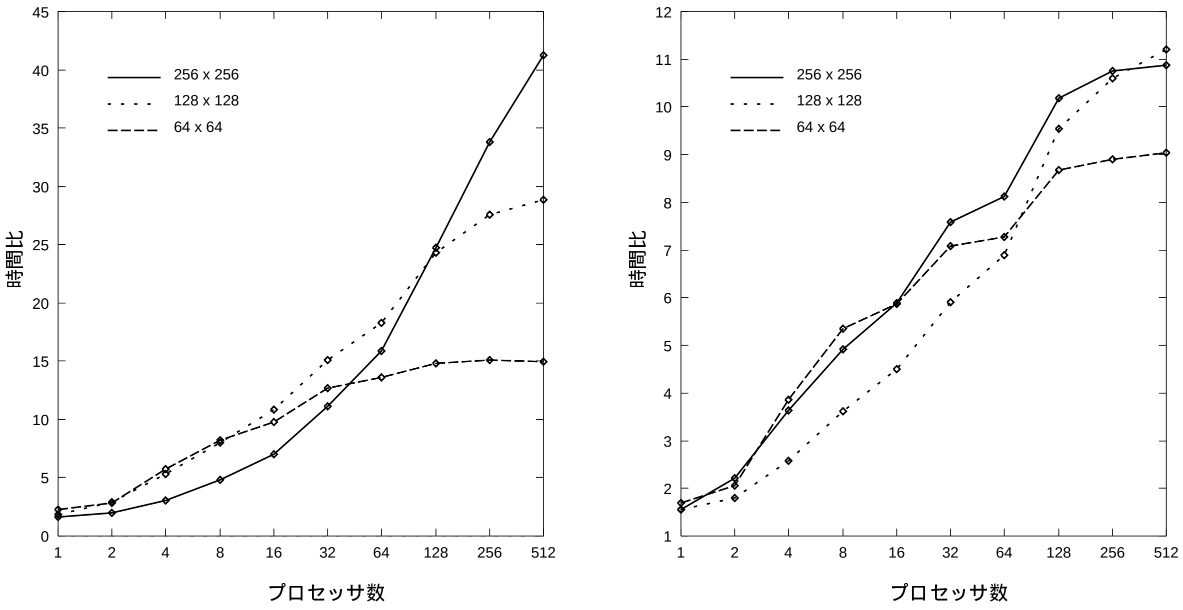


(a) 単純法



(b) 逆インデックス法 (c) 全検査法

図 24: LU 分解の実行時間



(a) 逆インデックス法　　(b) 全検査法

図 25: LU 分解の実行時間の比較 :単純法に対する比率

4.2.4　考察

単純法との比較　図 25に単純法と，逆インデックス法，全検査法との比率を示すグラフを示す．これは，（単純法の実行時間）/（逆インデックス法 / 全検査法の実行時間）を，プロセッサ数ごとに算出したものであり，単純法に比べて何倍早くなったかを示す．

この図 25から，逆インデックス法を使用して Inspector/Executor 戦略を適用した場合，この戦略を適用しなかった場合に比べ，プロセッサ数 512，配列のサイズ N=256 のときに，最大の 41 倍の速度向上を得たことが分かる．このような良好な結果を得たのは，データアクセスの度に起こ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．

ループのイタレーションの回数　ここで，LU 分解の `i,j` の 2 重ループについて，実行時間を解析してみる．行列のサイズを $N \times N$とすると，逐次の場合は N^2のイタレーションが必要となる．

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求めて実行すべきイタレーションを確定している．そのためプロセッサ数を Pとすると，`i,j` ともにイタレーション回数を $1/\sqrt{P}$倍にできるため，全体として N^2/P回のイタレーションでループの実行が行われる．

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．ゆえに，`i` のループのイタレーション回数はプロセッサ数にかかわらず一定となるため，全体では $N^2/\sqrt{P}$回のイタレーションの実行が必要である．

また，単純法では，イタレーションは全検査法と同じく $N^2/\sqrt{P}$であるが，$N/\sqrt{P}$回の配列 `pivot` の値のブロードキャストが必要となる．

ゆえに，Inspector/Executor 戦略を適用した場合ブロードキャストを不要にできるため，単純法に比べて実行時間を大幅に短縮できる．

また，逆インデックス法と全検査法によるコードでは，イタレーションの回数に N^2/Pと $N^2/\sqrt{P}$という差があるため，逆インデックスの方が高速であるといえる．

台数効果　LU 分解の実行結果に対する台数効果のグラフを図 26に示す．これは，(1 プロセッサでの実行時間）/ （P プロセッサでの実行時間）によって求めたものである．

どのアルゴリズムでも行列のサイズ 256，プロセッサ数 512 のときに最大の速度向上が得られ，逆インデックス法で 98 倍，全検査法で 28 倍，単純法で 3.9 倍である．このグラフから，Insepctor/Executor 戦略を用いた場合，特に逆インデックス法を用いた場合，良好な台数効果が得られるといえる．それに対し，単純法では，台数効果がほとんど期待できない．

5　結論

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．

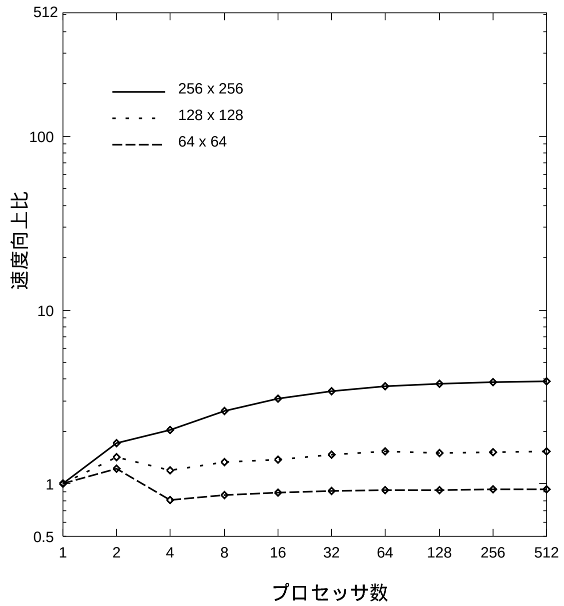
アルゴリズムの適用範囲の点では，全検査法では，制限がなく，逆インデックス法では，従来適用できなかった，ピボッティング付き LU 分解に適用可能となった．

さらに，逆インデックス配列の生成可能性を，インデックス配列の性質から考察し，単射であれば逆が一意に求まり，全射であれば逆インデックスを効率的に格納できることを示した．

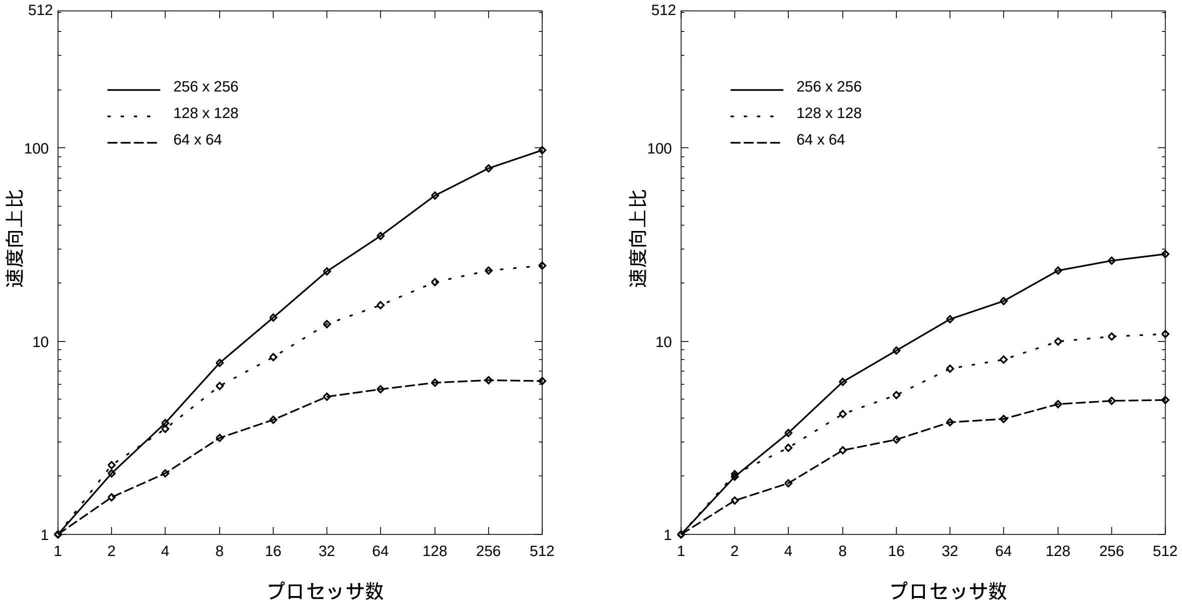
今後の課題は，$send\_list, recv\_list$ などのリストの操作を，より高速なものにすること，様々な例題について逆インデックス配列の生成可能性を調べ，有効性を検証することである．

本稿で述べた inspector アルゴリズムのオーバーヘッドはかなり大きい．この高速化の手段として，[21] において Hashed Cache という技法が提案されている．これは，他のプロセッサから送られてきたデータの格納にハッシングに似た方法を取ることと，一度 inspector で解析された送られてきたデータを再利用する手法である．ICCG 法，SOR 法などの反復解法では行列の非零要素の配置が変わらないため，この方法が有効である．本論文で提案した手法でも，容易に取り入れることが可能であろう．

逆インデックス配列をコンパイラシステムが活用するには，プログラマからシステムへの指示が必要である．逆関数の指定については [22] において，そのアプローチが示されている．我々も，この方針で逆関数を記述すべきものと考える．単射，全射が基本的な性質であることは 3.3節で述べたが，実際の問題では，LU 分解のピボットのようにこれらが組み合わされたり，不規則疎行列のように，行列の対称性，1 行あたりの非零要素数などの性質も利用する必要が生じる．今後，さらに多くの例題について逆インデックス配列の生成可能性を調べ，コンパイラシステムへの逆関数の指定法を検討する必要がある．



(a) 単純法



(b) 逆インデックス法　　(c) 全検査法

図 26: LU 分解の台数効果

謝辞

本研究の機会を与えて下さいました本学教授 富田 眞治 先生に深甚なる謝意を表します．

研究を進める上での有益な御指導，御意見を賜わりました本学助教授 中島 浩先生に深く感謝の意を表します．

日頃の研究活動において，終始多大な御助言，御指導をいただきました本学助手 森 眞一郎 先生に深く感謝致します．

本学在職中に数々の貴重な御意見をいただきました現在京都工芸繊維大学教授柴山 潔 先生に心から感謝致します．

また，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．

そして，有意義な研究室生活を共にした富田研究室の諸氏に感謝致します．

参考文献

[1] S. Hiranandani, K. Kennedy, and C.-W. Tseng: Compiler Optimizations for Fortran D on MIMD Distributed-Memory Machines, in *Proceedings of Supercomputing '91* (Albuquerque, NM, Nov. 1991), 86–100.

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

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

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

[5] M. Chen, D.-Y. Chen, Y. Hu, M. Jacquemin, C.-Y. Lin, and J.-J. Wu: Parallelizing Programs for Distributed–Memory Machines using the Crystal System, in *Languages, Compilers and Run–Time Environments for Distributed Momory Machines* (Elsevier Science Publishers B. V., 1992), 237–254.

[6] M. Rosing, R. B. Schnabel, and R. P. Weaver: The DINO Parallel Programming Language, *J. Parallel and Distributed Computing*, 13 (1991), 30–42.

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

[8] C. A. R. Hoare: Communicating Sequential Processes, *Communications of the ACM*, 21, 8 (1978), 666–677.

[9] A. Yonezawa, E. Shibayama, T. Takada, and Y. Honda: Modelling and Programming in an Object–Oriented Language ABCL/1, in A. Yonezawa and M. Tokoro eds., *Object–Oriented Concurrent Programming* (The MIT Press, 1986), 55–89.

[10] N. Carriero and D. Gelernter: Linda in Context, *Comm. ACM*, 32, 4 (Apr. 1989), 444–458.

[11] A. S. Tanenbaum, M. F. Kaashoek, and H. E. Bal: Parallel Programming Using Shared Objects and Broadcasting, *Computer*, 25, 8 (Aug. 1992), 10–19.

[12] 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 (1992–12).

[13] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker: Solving Problems on Concurrent Processors, volume 1 (Prentice Hall, Englewood Cliffs, New Jersey 07632, 1988).

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

[15] J. Saltz, S. Petiton, H. Berryman, and A. Rifkin: Performance Effects of Irregular Commniction Patterns on Massively Parallel Multiprocessors, *J. Parallel and Distributed Computing*, 13 (1991), 202–212.

[16] C. Berge: Principes de Combinatoire (DUNOD, 1968), 野崎昭弘訳 :組合せ論の基礎 (サイエンス社,1973）.

[17] 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 (1992–6), 195–202.

[18] 森正武: 数値計算プログラミング (増補版) (岩波書店, 1988).

[19] 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).

[20] J. Saltz, K. Crowley, R. Mirchandaney, and H. Berryman: Run-Time Scheduling and Execution of Loops on Message Passing Machines, *J. Parallel and Distributed Computing*, 8 (1990), 303–312.

[21] S. Hiranandani, J. Saltz, P. Mehrotra, and H. Berryman: Performance of Hashed Cache Data Migration Schemes on Multicomputers, *J. Parallel and Distributed Computing*, 12 (1991), 415–421.

[22] J. Li and M. Chen: Compiling Communication–Efficient Programs for Massively Parallel Machines, *IEEE Trans. Parallel and Distributed Syst.*, 2, 3 (July 1991), 361–375.

表 2: SMVM の実行時間 (単位 :秒) :単純法

PE\size	64	256	1K	4K	16K	32K
1	0.933578	1.139643	1.404092	1.659277	1.912085	2.076831
2	1.109295	1.390697	1.654145	1.911735	–	–
4	1.117077	1.393347	1.657284	1.914607	–	–
8	1.126432	1.395787	1.658921	1.915871	2.339751	–
16	1.267249	1.549912	1.801346	2.359385	3.478531	4.120142
32	1.292213	1.575619	1.849281	2.176044	2.973258	3.616179
64	1.322080	1.599911	1.872784	2.264686	2.737958	3.173994
128	–	–	1.887554	2.291508	2.804365	3.023762
256	–	–	1.902348	2.317075	2.829433	3.089527
512	–	–	1.920568	2.349629	2.858476	3.118537

表 3: SMVM の実行時間 (単位 :秒) :従来法

PE\size	64	256	1K	4K	16K	32K
1	0.955528	1.312961	1.783165	2.549471	3.548363	4.095884
2	0.917194	1.138076	1.554922	2.060134	3.048405	3.548906
4	0.910397	1.010557	1.354765	1.792250	2.554022	3.048818
8	0.926083	0.978897	1.189613	1.575327	2.069716	2.554083
16	0.956879	0.995462	1.139458	1.402192	1.810931	2.070025
32	1.025898	1.034561	1.158772	1.331772	1.638762	1.811886
64	1.147560	1.147408	1.175564	1.343052	1.553175	1.643383
128	–	–	1.281677	1.325680	1.552668	1.563919
256	–	–	1.405210	1.412901	1.487442	1.595627
512	–	–	1.543600	1.544133	1.561899	1.585423

表 4: SMVM の実行時間 (単位 :秒) :逆インデックス法

PE\size	64	256	1K	4K	16K	32K
1	0.962654	1.321889	1.785600	2.550753	3.548649	4.096107
2	0.923870	1.149320	1.557583	2.059130	3.048148	3.548731
4	0.911541	1.029379	1.360761	1.789240	2.552539	3.048260
8	0.912277	0.982519	1.213852	1.569514	2.066604	2.552608
16	0.933208	0.981926	1.147486	1.410778	1.808683	2.066775
32	0.979160	0.993601	1.151072	1.353184	1.642995	1.811360
64	1.074657	1.078590	1.135077	1.347235	1.571585	1.644146
128	–	–	1.216627	1.281502	1.561134	1.577202
256	–	–	1.338145	1.355659	1.453236	1.592865
512	–	–	1.483905	1.483964	1.509993	1.540226

表 5: SMVM の実行時間 (単位 :秒) :全検査法

PE\size	64	256	$1K$	$4K$	$16K$	$32K$
1	0.959234	1.316892	1.783843	2.549994	3.547877	4.095839
2	0.924278	1.146452	1.554670	2.060024	3.047933	3.548362
4	0.912563	1.050882	1.387941	1.802482	2.559132	3.051478
8	0.925468	1.032616	1.292189	1.635276	2.105211	2.573033
16	0.949445	1.039413	1.272543	1.538772	1.886148	2.160944
32	0.998740	1.086189	1.284877	1.532349	1.803374	1.956595
64	1.098027	1.153108	1.305905	1.533406	1.787179	1.910247
128	–	–	1.359230	1.540602	1.782694	1.903957
256	–	–	1.421784	1.566699	1.782041	1.903668
512	–	–	1.528343	1.634006	1.811095	1.922692

表 6: SMVM の inspector の実行時間 (単位 :秒) :従来法

PE\size	64	256	$1K$	$4K$	$16K$	$32K$
1	0.911129	1.284703	1.775587	2.545672	3.547335	4.094782
2	0.834457	1.057354	1.531536	2.048487	3.045461	3.547504
4	0.813652	0.929195	1.289454	1.776478	2.546211	3.045558
8	0.826759	0.883400	1.059381	1.530758	2.048428	2.546233
16	0.864517	0.895199	0.967567	1.289963	1.776355	2.048521
32	0.921700	0.928661	0.974617	1.133743	1.531609	1.777143
64	0.985855	0.986187	1.000076	1.128101	1.340446	1.533745
128	–	–	1.110826	1.138126	1.312675	1.364151
256	–	–	1.250588	1.254490	1.288490	1.376923
512	–	–	1.387986	1.384782	1.393337	1.408340

表 7: SMVM の inspector の実行時間 (単位 :秒) :逆インデックス法

PE\size	64	256	$1K$	$4K$	$16K$	$32K$
1	0.925877	1.293637	1.778067	2.546965	3.547739	4.095259
2	0.853386	1.084080	1.534981	2.045986	3.042552	3.546064
4	0.816269	0.952229	1.298809	1.774748	2.540881	3.042680
8	0.776474	0.889175	1.095877	1.529829	2.039369	2.540938
16	0.727570	0.854858	0.981209	1.300238	1.770292	2.039474
32	0.713440	0.791687	0.957024	1.172867	1.534260	1.772617
64	0.713806	0.763855	0.876526	1.135490	1.396843	1.534176
128	–	–	0.823318	0.966728	1.351288	1.394730
256	–	–	0.818336	0.896149	1.141052	1.363861
512	–	–	0.849197	0.883362	1.013000	1.166138

表 8: SMVM の inspector の実行時間 (単位 :秒) :全検査法

PE\size	64	256	$1K$	$4K$	$16K$	$32K$
1	0.918179	1.288661	1.776380	2.546197	3.547000	4.095051
2	0.865425	1.093910	1.536533	2.046879	3.042401	3.545651
4	0.859528	1.002323	1.357617	1.789349	2.548190	3.046036
8	0.858955	0.988646	1.261310	1.614766	2.081623	2.562058
16	0.848221	0.988766	1.239431	1.516517	1.868492	2.137267
32	0.859283	0.991816	1.257710	1.513298	1.783637	1.941990
64	0.870422	0.996590	1.255698	1.518802	1.772275	1.899879
128	–	–	1.257613	1.513843	1.775203	1.894913
256	–	–	1.260409	1.513642	1.768205	1.896138
512	–	–	1.277238	1.528971	1.780876	1.908073

表 9: LU 分解の実行時間（単位:秒）:単純法

PE\size	64×64	128×128	256×256
1	0.250296	1.635493	16.207005
2	0.204358	1.143237	9.411827
4	0.310784	1.367220	7.961084
8	0.291484	1.230290	6.141927
16	0.279645	1.179739	5.215343
32	0.275032	1.107160	4.744918
64	0.270665	1.068355	4.462214
128	0.270427	1.082045	4.292909
256	0.268480	1.073320	4.214353
512	0.269327	1.057966	4.145190

表 10: LU 分解の実行時間（単位:秒）:逆インデックス法

PE\size	64×64	128×128	256×256
1	0.111738	0.900982	9.794357
2	0.072100	0.393929	4.740952
4	0.053812	0.256070	2.591249
8	0.035314	0.153823	1.266656
16	0.028614	0.108840	0.740191
32	0.021672	0.073243	0.426967
64	0.019882	0.058514	0.280570
128	0.018287	0.044533	0.173400
256	0.017755	0.038977	0.124746
512	0.018003	0.036696	0.100422

表 11: LU 分解の実行時間（単位:秒）:全検査法

PE\size	64×64	128×128	256×256
1	0.147871	1.056311	10.416815
2	0.099323	0.515437	5.242350
4	0.080410	0.375672	3.097319
8	0.054482	0.250634	1.695611
16	0.047592	0.200727	1.160434
32	0.038856	0.146084	0.802842
64	0.037256	0.131668	0.648042
128	0.031167	0.106222	0.450364
256	0.030166	0.099754	0.397711
512	0.029789	0.097335	0.370176