不規則問題に対する並列化コンパイル手法

窪田 昌史，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
email: {kubota, moris, nakasima, tomita}@kuis.kyoto-u.ac.jp

我々は，メッセージ交換型の分散メモリ型並列計算機のための並列化コンパイラを開発している．インデックス配列による間接参照が行われ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するコンパイル技法として，Inspector/Executor というコード生成の戦略が提案されている．我々が提案している逆インデックス配列を用いた手法により，ループのイタレーションの実行順序の変更が可能となり，通信の通信のレイテンシが隠蔽され実行の高速化が達成できる．不規則メッシュの反復計算に本手法を適用すると，従来の手法に比べ最大で 2.16 倍の高速化された．

A Compilation Technique for Parallelizing Irregular Problems

Atsushi KUBOTA, Shin-ichiro MORI, Hiroshi NAKASHIMA and Shinji TOMITA

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

We have been constructing a parallelizing compiler for message passing distributed memory computers. Parallelization of loops with indirect access with index arrays causes irregular access pattern. For such codes, a technique to generate inspector/executor code has been proposed. The inverted index method we have proposed enables iteration reordering for such codes, while hiding latency of communication. We achieved 2.16 times speep up for unstructured mesh sweep by applying iteration reordering.

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(Single Program Multiple Data Stream)モデルによるプログラム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．

しかし，不規則問題と呼ばれる，不規則メッシュ上の流体力学，分子動力学，多体問題などでは，データのアクセス・パターン，計算時間が実行時にならなければ確定されないため，実行時解析を用いた並列化が必要である．

この実行時解析の手法として，Inspector/Executor戦略が提案され [1]，CHAOS [2]といった実行時ライブラリが開発されている．しかし，従来の手法では，

- 代入文の左辺にインデックス配列が存在する場合，インデックス配列を基準としてowner computes ruleを適用してその代入文の実行を各プロセッサに割付けなければならない．そのため，代入文の実行後，インデックス配列によって間接参照されているデータのownerプロセッサへ代入後の値を送信しなければならない．
- ループのイタレーションの実行順序の変更による，通信と計算のオーバーラップによる通信レイテンシの隠蔽 [1]ができない．

などの問題がある．そのため，実行時間全体に占める通信の割合が大きく，並列化による実行の高速化が不十分である．

そこで我々は，逆インデックス法という，インデックス配列の逆関数を求めることで，Inspector/Executor戦略に基づいて生成されるコードの実行を高速化する手法を提案してきた [3]．

本稿では，CHAOSライブラリを拡張し逆インデックス法に対応させ，従来法との実行時間の高速が達成できたことを報告する．

```
do n=1,n_steps

  do i=1,n_edges
    y(edge1(i)) = y(edge1(i)) +
      f( x(edge1(i)), x(edge2(i)) )
    y(edge2(i)) = y(edge2(i)) +
      g( x(edge2(i)), x(edge2(i)) )
  enddo

  do i=1,n_nodes
    x(i) = y(i)
  enddo

  check_convergence

enddo
```

図 1: 不規則メッシュの分解

2章では，逆インデックス法を用いることにより，イタレーションの実行順序の変更，Executorにおける通信の削減が可能であることを示す．3章では，不規則メッシュの反復計算問題に逆インデックス法を用いたことによる性能向上について述べる．4章で総括を行い，今後，課題について述べる．

2 Inspector/Executor戦略によるコードの生成

2.1 不規則問題

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．

あるいは，データの不規則分割する場合，配列データの分割法はプログラムの入力として与えられるか，あるいは実行時に動的に決定されるため，同様にプロセッサ間のメッセージ通信の要/不要，送信元と送信先はプログラムの実行時に決定される．

このようなデータアクセスを伴う問題は，不規則問題(irregular problem)と呼ばれる．

例えば，図1のコードは，不規則問題の一例である．この例では，計算対象となるノードとエッジそれぞれに番号づけをしておく．あるエッジの両端のノードの番号は`edge1`と，`edge2`に格納しておく．すべてのエッジに対し，両端のノード`x`の相互作用を計算し，一時的に`y`に格納する．`y`

の値を `x` に書き戻すことで，1 回のメッシュ上のノードの値を更新する．

`x(edge1(i))`，`x(edge2(i))` を保持するプロセッサがコンパイル時にならなければ決定できないため，owner computes rule に基づいて，どのイタレーションをどのプロセッサで実行すべきか，右辺のデータ，`y(edge1(i))`，`y(edge2(i))` は，どのプロセッサからどのプロセッサへ送信すればいいかはコンパイル時にならなければ確定しない．

2.2 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 Inspector/Executor コード生成法が提案されている [1]．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 $p \neq q$となる全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の作成
      - 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)**
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) *recv_list(p,q)* のデータを qから受信する．
   (c) ループを実行する．

2.3 従来の手法

図 1のように，代入文の左辺に間接参照がある場合，インデックス配列を基準として owner computes rule を適用する手法が採用されてきた．つまり，配列の要素 `edge1(i)`，`edge2(i)` を保持するプロセッサにおいて，それぞれ図 1の 3,4 行目，5,6 行目の代入文が実行される．

図 1の 3,4 行目を実行するには，

1. `y(edge1(i))`，`y(edge2(i))` を保持するプロセッサは，`edge1(i)` を保持するプロセッサへそれぞれ `y(edge1(i))`，`y(edge2(i))` を送信する．
2. `edge1(i)` を保持するプロセッサにおいて，3,4 行目の代入文が実行され，一時変数 `tmp` にその値が代入される．
3. `edge1(i)` を保持するプロセッサが，`tmp` の値を，`x(edge1(i))` を保持するプロセッサへ送信する．
4. `x(edge1(i))` を保持するプロセッサは，`tmp` を受信し，`x(edge1(i))` を更新する．

という手順をとる．5.6 行目の代入文についても同様である．すると，図 1を，Inspector/Executor 戦略に基づいて並列化した疑似コードは図 2のようになる．

この方法では，ループの前後で `x,y` の全対全の通信が生じるため，通信の起動回数が多く，並列化による高速化が不十分となる．

```
make_recv_list
comm_recv_list

do n=1,n_steps

  gather_off_proc(x)

  do i=1,n_edges
    y(edge1(i)) = y(edge1(i)) +
      f( x(edge1(i)), x(edge2(i)) )
    y(edge2(i)) = y(edge2(i)) +
      g( x(edge2(i)), x(edge2(i)) )
  enddo

  scatter_off_proc(y)

  do i=1,n_nodes
    x(i) = y(i)
  enddo

  check_convergence

enddo
```

図 2: Inspector/Executor コード
(イタレーションの実行順序の変更なし)

2.4 逆インデックス配列を用いたコードの生成

我々は，逆インデックス配列を用いた Inspector/Executor を提案している．

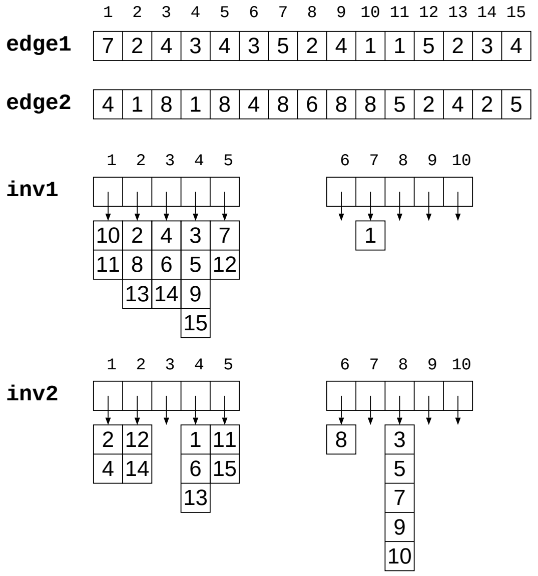

図 3: 逆インデックス配列

逆インデックス配列とは，インデックス配列を関数とみなしたときの逆関数である．配列があるループで順次，参照される場合の逆インデックスは，配列のある要素が，ループの何番目のイタレーションで参照されるかを示す．

図 3は，要素数 15 のインデックス配列 `edge1`，`edge2` とその逆インデックス配列の例である．この例でわかる通り，逆インデックス配列は一般にはリストの形式を用いて表される．

例えば，`edge1(1)=7` であるが，`inv1(7)=1` であり，`x(7)`, `y(7)` は，1 番目のイタレーションで参照されることがわかる．

この逆インデックス配列を用いると，インデックス配列 `edge1` ではなく，更新される配列 `y` を基準として owner computes rule を適用することができる．

例えば，プロセッサ 1 が配列の要素 `y(1)...y(5)` を保持していたとすると，`edge1`，`edge2` の逆インデックス配列を参照することで，プロセッサ 1 で実行すべきイタレーションが求められる．例えば，図 1のコードの 3,4 行目の代入文については，イタレーション番号 10, 11, 2, 8, 13, 4, 6, 14, 3, 5, 9, 15, 7, 12 を実行し，5,6 行目の代入文については，イタレーション番号 2, 4, 12, 14, 18, 1, 6, 13, 11, 15 を実行する．

また，右辺で参照されるデータに対しても，`x(1)...x(5)` を保持するプロセッサは，逆インデックス配列を参照することで，それらのデータが参照されるイタレーションが求められる．さらに，(順) インデックス配列 `edge1`，`edge2` を参照し，各イタレーションで更新される代入文の左辺に現れる配列要素が求められる．配列要素を保持するプロセッサは，規則分割が行われていれば簡単に求められる．ゆえに，`x(1)...x(5)` の送信先が求められたことになる．

逆インデックス配列を求めることで，あるプロセッサにおいて実行すべきイタレーション番号が求められたら，それらのイタレーションで必要とするデータの参照が通信を伴うものかどうかが判定できる．通信を伴うデータアクセスの有無で，イタレーションを `local` と `nonlocal` の集合に分けておく．`local` の集合に含まれるイタレーションは，通信とオーバーラップして実行できるため，通信を伴うデータアクセスのレイテンシを隠蔽することができる．

このようにして並列化したコードを図 4に示す．

ただし，逆インデックス配列を用いたコードを生成するには，インデックス配列の全ての要素を全プロセッサで重複して保持する必要がある．

3　性能評価

評価には，前章で取り上げた不規則メッシュの反復計算を TINPAR[4] によって並列化したコードに，今回作成した実行時ライブラリのコールを手動で挿入した．

評価プログラムの実行には，64 台構成の富士通 AP1000 を用いた．不規則メッシュの 2 種類のデータ，1) メッシュ数 2800，エッジ数 17377，2) メッシュ数 545，エッジ数 3136 を入力とし，反復を 100 回行い，その実行時間を計測した．イタレーションの実行順序を変更する方法では，各プロセッサ上での実行時間，Inspector，Executor に費やした時間，イタレーションの実行順序を変更しない方法では，各プロセッサ上での実行時間，Inspector, Executor に費やした時間に加え，Executor 内でのデータの分散 (Scatter) 収集 (Gather) に用いた時間を計測した．

表 1，2に計測した実行時間を示す．Total, In-

```
make_inv_list
make_send_recv_local_nonlocal_lists

do n=1,n_steps

  gather_send_off_proc(x)

  do i∈local_iter()
    y(edge1(i)) = y(edge1(i)) +
      f( x(edge1(i)), x(edge2(i)) )
    y(edge2(i)) = y(edge2(i)) +
      g( x(edge2(i)), x(edge2(i)) )
  enddo

  gather_recv_off_proc(x)

  do i∈nonlocal_iter()
    y(edge1(i)) = y(edge1(i)) +
      f( x(edge1(i)), x(edge2(i)) )
    y(edge2(i)) = y(edge2(i)) +
      g( x(edge2(i)), x(edge2(i)) )
  enddo

  do i=1,n_nodes
    x(i) = y(i)
  enddo

  check_convergence

enddo
```

図 4: Inspector/Executor コード
(イタレーションの実行順序の変更あり)

spector, Executor の行は，それぞれ全体の実行時間とそのうち Insepctor, Executor に費された時間を示す．また，それらの実行時間のグラフを図 5，6に示す．グラフ中，inv.total, inv.ins, inv.exec, conv.total, conv.ins, conv.exec の線は，それぞれイタレーションの実行順序の変更を行った場合の全体の実行時間，Inspector, Executor の実行時間，変更しない場合の全体の実行時間，Insepctor, Executor の実行時間を表す．

さらに，イタレーションの実行順序の変更なしの場合の通信が Executor に占める割合，つまり (Gather + Scatter / Executor) を 表 1, 2の (G+S/E) の行に示す．また，表の最後の 3 行は，イタレーションの実行順序の変更をする場合の，しない場合に対する速度の向上率を実行時間全体，Inspector, Executor について求めたものである．

Executor の実行時間がイタレーションの実行順序の変更により減少しており，全体の実行時間も同様に減少している．

イタレーションの実行順序の変更前は，i のループの前後のデータの収集，分散に，Executor の実行時間のほとんどが費やされている．特に，プロセッサ台数が増加するにつれ，これらの通信に占める割合が大きくなる．ノード数 545 の場合，4 プロセッサ時の 35.6%から，64 プロセッサ時には 78.8%に，ノード数 2800 の場合，16 プロセッサ時の 64.1%から，64 プロセッサ時の 80.5%にも達する．

これに対し，イタレーションの実行順序を変更すると，Executor での実行時間が減少しており，Executor における通信の起動回数の削減と，通信とローカルなデータのみをアクセスするイタレーションとのオーバーラップの効果が現れているといえる．

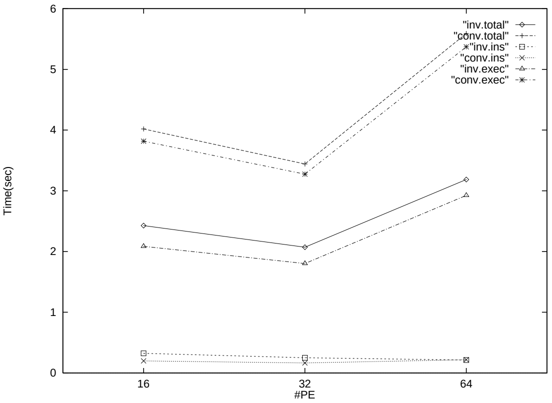

図 5: 実行時間 (ノード数 2800，エッジ数 17377)

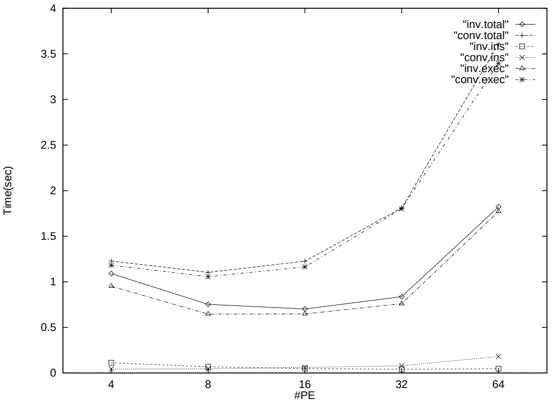

図 6: 実行時間 (ノード数 545，エッジ数 3136)

4　おわりに

本稿では，不規則なアクセスを伴うループの並列化において，我々の提案する逆インデックス配列を用いることで，イタレーションの実行順序の変更に対応でき，実行が高速化できることを示した．

今回作成した実行時ライブラリは，各種リストを作成するための作業領域を大量に必要とするため，メモリ効率，実行速度の点で問題があるため，今後はその改良を行いたい．

また，不規則なデータ分割による不規則アクセスを生じる問題，インデックス配列が実行時に頻繁に変更される問題などにも評価の対象を広げ，有効性の確認，改良点の検討を行いたい．

謝辞

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．

参考文献

[1] 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.

[2] Ravi Ponnusamy, Joel Saltz, Alok Choudhary, Yuan-Shin Hwang, and Geoffrey Fox. Runtime support and compilation methods for user-specified irregular data distributions. *IEEE Trans. Parallel and Distributed Syst.*, Vol. 6, No. 8, pp. 815–831, August 1995.

[3] 窪田昌史, 三吉郁夫, 大野和彦, 森眞一郎, 中島浩, 富田眞治. 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–. 情報処理学会論文誌, Vol. 35, No. 4, pp. 532–541, April 1994.

[4] 三吉郁夫, 前山浩二, 後藤慎也, 森眞一郎, 中島浩, 富田眞治. メッセージ交換型並列計算機のための並列化コンパイラ tinpar. 並列処理シンポジウム JSPP, pp. 51–58, May 1995.

表 1: 実行時間（ノード数 2800，エッジ数 17377）

	16	32	64
イタレーションの実行順序の変更あり（単位：秒）			
Total	2.427	2.071	3.186
Insptctor	0.324	0.250	0.212
Executor	2.085	1.803	2.925
イタレーションの実行順序の変更なし（単位：秒）			
Total	4.018	3.441	5.591
Inspector	0.199	0.165	0.218
Executor	3.818	3.275	5.372
Gather	1.208	1.210	2.578
Scatter	1.214	0.939	1.746
(G+S)/E	0.641	0.656	0.805
性能向上比			
Total	1.66	1.66	1.75
Inspector	0.614	0.660	1.028
Executor	1.83	1.82	1.84

表 2: 実行時間（ノード数 545，エッジ数 3136）

	4	8	16	32	64
イタレーションの実行順序の変更あり（単位：秒）					
Total	1.091	0.753	0.703	0.838	1.823
Insptctor	0.113	0.070	0.050	0.041	0.048
Executor	0.952	0.650	0.649	0.760	1.771
イタレーションの実行順序の変更なし（単位：秒）					
Total	1.229	1.104	1.226	1.810	3.610
Inspector	0.044	0.047	0.061	0.080	0.181
Executor	1.183	1.056	1.164	1.802	3.395
Gather	0.193	0.283	0.419	0.738	1.927
Scatter	0.228	0.380	0.381	0.422	0.749
(G+S)/E	0.356	0.628	0.687	0.644	0.788
性能向上比					
Total	1.13	1.47	1.74	2.16	1.98
Inspector	0.389	0.671	1.22	1.95	3.77
Exececutor	1.24	1.62	1.79	2.37	1.92