キャッシュメモリを意識した古典的反復解法の実装方法

丸山訓英[†] 襲田勉[†]
鷲尾巧[†] 土肥俊[†]

近年マイクロプロセッサの演算性能は大幅に向上したが，一方で，多くの大規模科学技術計算においてはキャッシュミスヒットが要因となりプロセッサの演算性能を生かすことが難しくなっている．このような状況をふまえ，本稿では大規模疎行列連立1次方程式の古典的反復解法であり流体計算などで用いられる逐次的過剰緩和法（SOR法）を高速実行する方法を考える．SOR法の標準的な実装を行うとデータアクセスの局所参照性が小さくなりキャッシュミスヒット率が高くなる．そこで，本稿ではキャッシュミスヒットを削減するよう未知数の更新順序を最適化する手法について述べる．この手法では，未知数の更新を行う範囲を指定する枠を定め，枠内に含まれる節点上の未知数を更新し，枠を一定の規則に従い移動させ，再度枠に含まれる節点上の未知数を更新する操作を繰り返す．ここでは，向き付きグラフを用いて本手法の正当性を保証するための枠の形状と移動方法が満たすべき必要十分条件を与える．本手法の評価では，Pentium3（933 MHz）上において標準的な実装方法に比べ，2次元5点差分問題で約3倍，3次元7点差分問題で約2倍に演算性能が向上することを確認した．また，手法の長所を最大限に発揮させるような方策として，枠の形状の最適化方法およびデータ格納方式の改良法を与える．

Cache-aware Implementations for Classical Iterative Methods

Kunihide Maruyama,[†] Tsutomu Osoda,[†] Takumi Washio[†]
and Shun Doi[†]

Although the raise of peak performance of recent microprocessors is quite impressive, it becomes difficult to exploit their potential HW ability in many classes of large scale scientific applications due to cache problems. In this paper, we investigate a way to overcome this difficulty for the successive over relaxation (SOR) method, which is one of typical classical iterative methods and has been applied, for example, to the computational fluid dynamics. It is well-known that high cache miss rates are observed with the standard implementation of the SOR iterations due to the cache capacity problem. In the introduced method, first, we define a frame which determine a region in which unknowns are updated. Then, we put the frame at an initial place and we update unknowns included in the frame. After this, we shift the frame and iterate this process. In this paper, we give necessary and sufficient conditions on the shape of the frame and the way to shift the flame to justify this method by means of a directed graph which represents the update status of unknowns. In our evaluation of the method, the sustained performance with the introduced method was tripled and doubled compared to with the standard implementation, respectively, in the two dimensional 5-point and the three dimensional 7-point cases. Furthermore, we investigate a way to determine an optimal frame size to attain the best performance of the method.

1. はじめに

近年，プロセッサの演算性能の向上により，スカラ型サーバのピーク性能は向上している．それにともない科学技術計算をスカラ型サーバ上で高速に行いたいという期待が強くなっている．

ただ，プロセッサの演算性能の向上率に比べると，主記憶・プロセッサ間の転送速度の向上率は低いために，主記憶・プロセッサ間の転送速度がボトルネックとなり，プロセッサの演算性能向上に見合う分だけ，アプリケーションの実効性能を向上させるのは難しい．そこで，主記憶に比べ高速アクセスが可能なキャッシュメモリが両者の間に置かれ，プロセッサの演算性能を十分に引き出すための工夫がなされている．

ただ，キャッシュメモリは主記憶に比べ容量が小さく，データのアクセスパターンに局所参照性がない場合には，アクセスするデータがキャッシュメモリに存

† 日本電気株式会社基礎研究所
Fundamental Research Laboratories, NEC Corporation

在せず主記憶へのアクセスが発生する（これをキャッシュミスヒットと呼ぶ）．アプリケーションの開発においてはキャッシュミスヒットを削減するためにデータアクセスに局所参照性を持たせるなどの工夫が必要となる．

ところで，差分法や有限要素法において現れる連立 1 次方程式の反復解法では，行列データのサイズはキャッシュメモリサイズを超えることがごく普通で，標準的な手法で実装すると，データアクセスの局所参照性が小さくなりプロセッサの演算性能を発揮させることが難しくなる．

これまでスカラ型サーバ向けのアルゴリズムの研究においては，密行列を扱う際の高速化手法としてのブロック化アルゴリズム[1]などいくつかの研究はなされている．一方で，疎行列を係数行列に持つ連立 1 次方程式の反復解法の場合には，まだ研究されていない事項も多い．

さて，連立 1 次方程式の反復解法には，ヤコビ法やガウスザイデル法などの古典的反復法と共役勾配法などのクリロフ部分空間法がある．このうち古典的反復法の特徴は，値の更新において各反復ごとに同じ行列データや右辺ベクトルデータが参照されること，値の更新においては近傍の未知数の最新の値のみを参照すればよいことの 2 点である．この点に注目すると，未知数の更新順序を最適化し参照される値の再利用率を向上させることで，より高い演算性能を引き出すことができる．

そこで本稿では，キャッシュメモリに格納された値を効率良く使用することにより古典的反復法をスカラ型サーバ上で高速に実行するための実装方法について述べる．

2 章では，キャッシュメモリの実装方式に関する用語を定義する．そして，古典的反復法を標準的な方法を実装では，キャッシュメモリを有効に使用できないことを説明する．3 章において，キャッシュメモリを有効に使用する古典的反復法の実装方法について述べる．実装方法の基本的なアイデアは，文献 2) の中で述べられているが，提案する実装方法と違いは，3 次元での実装方法が，2 次元の実装方法の自然な拡張となっている点である．4 章で数値実験結果を示す．

2. 古典的反復法をスカラ型サーバに実装する際の問題点

本章では，キャッシュのミスヒット発生原因について説明し，古典的反復法の 1 つである SOR 法を例にスカラ型サーバ上を実装する場合の問題点を示す．

2.1 ミスヒット発生原因

本稿ではキャッシュメモリの実装方式の 1 つである set associative 方式[1]を使用することを仮定する．キャッシュミスヒットは以下の 3 つに大別できる．

- **初期化により生じるキャッシュミスヒット** 初めてキャッシュラインを主記憶からキャッシュメモリへ移動させることにより発生するキャッシュミスヒットである．
- **容量不足により生じるキャッシュミスヒット** あるキャッシュラインの移動において，そのラインが前回キャッシュメモリに格納された時点から移動直前までの間に主記憶からキャッシュメモリへ移動されたライン数がキャッシュメモリ内に格納可能なライン数を上回る場合に発生するキャッシュミスヒットである．
- **ラインコンフリクトにより生じるキャッシュミスヒット** あるキャッシュラインのキャッシュメモリへの移動において，前回キャッシュメモリへ移動した時点から移動直前までの間にキャッシュメモリへ移動されたキャッシュライン数がキャッシュメモリに格納可能なライン数より少ない場合に引き起こされるキャッシュミスヒットである．

2.2 SOR 法の標準的実装方法の問題点

まず，SOR 法の標準的な実装方法[4]を示す．構造格子上で方程式を離散化することにより生じる大規模スパース行列を係数行列として持つ連立 1 次方程式を

$$Ax = b \tag{1}$$

とする．以下のプログラムは，例として 2 次元問題を 5 点差分により離散化した場合を示す．未知数の番号付けは辞書式オーダリング（自然なオーダリング）とする．一番外側の it についてのループは反復回数に対応する．なお，ω は収束加速定数であり，eps は収束判定を行うための定数である．error の値が eps 未満になればプログラムを終了する．ここで，i, j は未知数の座標を示す．$A(1,i,j)$, $A(2,i,j)$, $A(3,i,j)$, $A(4,i,j)$, $A(5,i,j)$ は行列 A 内の i, j 成分に対応する行に含まれる非ゼロ成分である．$A(3,i,j)$ には対角成分の逆数を記憶する．また，メモリ上では一次元の要素から順に格納されているとする．すなわち $A(1,i,j)$, $A(2,i,j)$, $\cdots$, $A(5,i,j)$, $A(1,i+1,j)$, $A(2,i+1,j)$, $\cdots$ の順に格納されているとする．

SOR 法の標準的な実装方法

```
do it=1,nt
  error=0.0
  do j=1,n
  do i=1,n
```

```
      w=A(3,i,j)*(b(i,j)-A(1,i,j)*x(i,j-1)
$      -A(2,i,j)*x(i-1,j)-A(4,i,j)*x(i+1,j)
$      -A(5,i,j)*x(i,j+1))
       error=error+(x(i,j)-w)*(x(i,j)-w)
       x(i,j)=x(i,j)+omega*(w-x(i,j))
     enddo
     enddo
     if(error.lt.eps) stop
  enddo
```

上記のプログラムをスカラ型サーバ上に実装した場合，次の 2 点によって性能を十分に引き出すことが困難となる．

- 未知数の数が多くなると，未知数の更新をするのに必要なデータを主記憶から参照しなくてはならないこと．
  上記プログラムを見ても分かるように，毎回の反復において，すべての未知数にわたって 1 回ずつ更新処理を行う．A，x，b をキャッシュメモリに格納できない場合，ある未知数の更新において，it+1 回目の反復を行うときには，it 回目の反復で参照したデータはキャッシュメモリから消えてしまう可能性が高い．したがって，再度必要なデータを主記憶からキャッシュメモリへコピーする必要が起こりうるからである．
- キャッシュメモリの容量に合わせて計算領域全体を複数の小領域に分割し，小領域内の未知数を連続して更新し，この操作をすべての小領域に対して実行すると，標準的実装方法に比べ計算速度は向上するが，計算量が増大してしまうこと
  未知数の更新にあたっては，つねに最新の未知数データを参照する．しかし，小領域の境界部分の未知数は，小領域外の更新されていない古いデータを参照し未知数の更新をするため，収束までの反復回数が増大するので，計算量が増大する．

3. キャッシュメモリを有効に使用するための実装方法

2.2 節で述べたように，古典的反復法を標準的な方法で実装した場合キャッシュメモリを有効に使うことができない．ここでは，値の更新順序を最適化し，キャッシュメモリ容量不足により生じるミスヒットを削減可能となる点 SOR 法，点ヤコビ法の実装方法について述べる．この手法のアイディアは線 SOR 法や時間発展問題を陽解法により解く場合にも応用できる．

最後に，本章で述べる手法を適用するにあたり，2.1 節で述べたラインコンフリクトが引き起こすキャッシュミスヒットにも注意が必要となることを述べ，その回数をキャッシュメモリに関するパラメータ（キャッシュメモリサイズ，キャッシュラインサイズ，セット内ライン数）およびアクセスするデータのアドレスより見積もる方法を示す．

3.1 手法の説明

本章では，連立一次方程式 の係数行列 $A = (a_{i,j})$（ただし， $1 \leq i, j \leq n$ とする）の非ゼロ構造をグラフ理論の用語を用いて表す[5]．本稿では，係数行列 A の非ゼロ構造は対称であるとする．まず，係数行列 A の非ゼロ構造を表すグラフを以下のように定義する．

定義 1 係数行列 A の非ゼロ構造を表す有向グラフ $G = (V, E)$ を節点集合 V と辺の集合 $E \subset V \times V$ とで定義する．節点の集合 V は未知数の集合とし，未知数の番号付けに従い各節点に番号を付ける．i 番目の未知数に対応する節点を v_i と表すこととする．辺の集合 E は

$$E = \{(v_i, v_j) \in V \times V | a_{i,j} \neq 0\} \quad (2)$$

と定める．

さて，2.2 節で述べたように，SOR 法では，毎回の反復においてあらかじめ定められた番号付けに従い節点上の未知数を更新する．ここで，$n \times n$ の係数行列 A を持つ連立 1 次方程式 $Ax = b$ を SOR 法を用いて解く際の標準的実装方法のアルゴリズム（以下アルゴリズム 1 とする）を示す．アルゴリズム 1 において，一番外側の itr のループは反復回数に対応する．次の i についてのループは節点の番号に対応し，小さい番号から大きい番号にかけて節点 v_i 上の未知数 x_i を更新する．

アルゴリズム 1

do $itr = 1, nitr$
  do $i = 1, n$

$$\widetilde{x_i} = \frac{1}{a_{i,i}}(b_i - \sum_{(v_i,v_j)\in E, j<i} a_{i,j} x_j^{(itr)} - \sum_{(v_i,v_j)\in E, j>i} a_{i,j} x_j^{(itr-1)})$$

$$x_i^{(itr)} = x_i^{(itr-1)} + \omega(\widetilde{x_i} - x_i^{(itr-1)})$$

  enddo
enddo

SOR 法では，ある節点の更新処理は，その節点に隣接する節点のみを参照して行われている．そのため，節点における更新処理を行うためにはすべての節点にわたって前回の更新処理が終了している必要はない．この点に着目し，複数の反復回数において，特定の 1 つの節点の 2 回の更新処理の間に更新される他の節点の個数を減らすことにより，更新を行う際にキャッ

シュメモリ上のデータを再利用することが可能となる．

更新順序の変更で重要なポイントは，各節点上の未知数の各更新ごとの値が更新順序の変更前と変更後で不変に保たれることである．

そのためには，各節点とそれにつながっている節点との間で，更新状態が，更新順序の変更前と変更後で，つねに “整合性のとれた状態” に保たれている必要がある．

ここで次のように，ある時点での更新状態をその時点における全節点の更新回数を対応付けることにより表す．ここでは，任意の時点に対して各節点 v の更新回数を $itr(v)$ と表す．つまり，ある時点での更新状態は，V 上で定義された非負の整数値をとる関数 itr により表される．

次に，更新回数 itr に基づき更新状態を “整合性のとれた状態” とそうでない状態に分ける．

定義 2 互いにつながっている任意の節点 v_i，v_j に対して，ある時点での更新状態 itr が次の条件を満たす場合，この更新状態 itr を整合性のとれた状態と呼ぶ．

$$i<j \Rightarrow \begin{array}{l} itr(v_i)=itr(v_j) \\ \vee\ itr(v_i)=itr(v_j)+1 \end{array} \qquad (3)$$

更新順序の変更の前後で更新状態が整合性のとれた状態にあれば，各節点において，それにつながっている節点との更新回数の関係がつねに保たれるので，ある反復回数 n に対して，整合性のとれた状態を維持しつつ任意の i について $itr(v_i)=n$ となったとき，得られた x_i は標準的実装方法によって得られる x_i と同じになる．

ここで，構造格子問題において具体的な更新順序の変更方法を述べる．まず，複数の節点を含むように枠の形状を適当に定める．そして，枠を定められた初期位置に置く．次に枠内の節点における更新処理を行う．そして，枠内のすべての節点における更新処理が終了した後，枠を定められた方向に移動する．この枠内の節点における更新処理と枠の移動の操作を枠が計算領域全体を通過するまで繰り返す．ここで，枠内の節点における更新処理を開始する前の更新状態は整合性のとれた状態であるようにする必要がある．なお，枠内の節点上の未知数の更新順序は次のように定める．

枠内節点の更新順序： 枠内の節点における更新処理を開始する前の更新回数をもとに，更新回数の少ない節点から順番に更新処理を行う．もし，更新回数が同じ場合は，番号の小さい節点から更新処理を行う．

なぜなら，上記のように枠内の更新処理を行う節点の順序を定めることにより，枠内更新の任意の時点において，枠内の互いにつながっている任意の 2 つの節点に関して式 (3) が成り立つからである．全体として整合性がとれるかどうかは枠の選び方と移動方法による．

上記アルゴリズムを具体的に記述する（以下アルゴリズム 2 とする）．下記のアルゴリズム 2 において，1 番目の k についてのループは枠の移動に対応し，P_k は k 回目の移動後の枠に含まれる節点の集合とする．2 番目のループは P_k 内の節点における更新処理に対応し，枠内節点の更新順序に従い節点を選び更新を行う．

アルゴリズム 2

do $k=1,m$
  do $v_i \in P_k$ (v_i は枠内節点の更新順序に従い選ぶ)
    $itr(v_i)=itr(v_i)+1$
    $\widetilde{x_i}=\frac{1}{a_{i,i}}(b_i-\sum_{(v_i,v_j)\in E,j<i}a_{i,j}x_j^{(itr(v_i))}$
      $-\sum_{(v_i,v_j)\in E,j>i}a_{i,j}x_j^{(itr(v_i)-1)})$
    $x_i^{(itr(v_i))}=x_i^{(itr(v_i)-1)}+\omega(\widetilde{x_i}-x_i^{(itr(v_i)-1)})$
  enddo
enddo

アルゴリズム 2 において itr をインクリメントしているが，これは更新回数を明示するために記したものであり，実際のプログラムでは必要ない．

上記のアルゴリズム 2 において重要となることは，すべての時点における更新状態が整合性のとれた状態を保つように枠の形状と枠の移動方向を定めることである．そこで，次に枠の形状と移動方法が満たすべき必要十分条件を示す．その前に，条件を示すにあたり整合性のとれた状態に対して各節点における更新処理を行う際の参照関係を表す向き付きグラフを定義する．

定義 3 $G=(V,E)$ を行列 A の非ゼロ構造を表すグラフ，itr を整合性のとれた更新状態とする．更新状態を表す有向グラフ $G(itr)=(V,E(itr))$ を V と E の部分集合 $E(itr)$ により定義する．ここで，$E(itr)\subset E$ は

$$E(itr)=\{(v_j,v_i)\in E | itr(v_j)=itr(v_i)+1 \vee (itr(v_j)=itr(v_i) \wedge j<i)\} \qquad (4)$$

と定める．以降，各辺 $(v_j,v_i)\in E(itr)$ に対して矢印は v_j から v_i の向きに付ける．

次に更新状態を表すグラフを用いて更新可能な節点を定義する．

定義 4 itr を整合性のとれた更新状態とする．節点

v_i とその節点に E の辺でつながっている任意の節点 v_j に対して $(v_j, v_i) \in E(itr)$ を満たすとき，節点 v_i をその時点で更新可能な節点とよぶ．

更新状態が itr である時点において，節点 v_i における更新処理を行った後の更新状態 $\widetilde{itr}$ は次の式により与えられる．

$$\widetilde{itr}(v_j) = \begin{cases} itr(v_j) + 1 & \text{for} \quad j = i \\ itr(v_j) & \text{for} \quad j \neq i \end{cases} \tag{5}$$

次の補題は更新可能な節点の定義をいい替えたものである．

補題 1 itr を整合性のとれた更新状態とする．節点 v_i が更新可能な節点であるための必要十分条件は，節点 v_i における更新処理をした後の更新状態 $\widetilde{itr}$ が整合性のとれた状態であることである．

[証明] 以下，v_j を節点 v_i に E の辺でつながっている任意の節点とする．

必要条件： 節点 v_i を更新可能な節点とするとき，次の 2 つの場合について，更新状態 $\widetilde{itr}$ も式 (3) を満たすことを示せば良い．

$itr(v_j) = itr(v_i) + 1$ の場合：節点 v_i における更新処理を行うと

$$\widetilde{itr(v_j)} = \widetilde{itr(v_i)} \tag{6}$$

となる．したがって，式 (3) が成り立つ．

$itr(v_j) = itr(v_i) \ \wedge \ j < i$ の場合：節点 v_i における更新処理を行うと

$$\widetilde{itr(v_j)} = \widetilde{itr(v_i)} - 1 \tag{7}$$

となる．したがって，式 (3) が成り立つ．

十分条件： 更新状態 $\widetilde{itr}$ が整合性のとれた状態であるとき，次の 2 つの場合に対して $(v_j, v_i) \in E(itr)$ を満たすことを示せばよい．

$i < j$ の場合：itr は整合性のとれた状態であるから，$itr(v_i) = itr(v_j)$ または $itr(v_i) = itr(v_j) + 1$ の 2 つの場合がある．$itr(v_i) = itr(v_j) + 1$ と仮定すると，節点 v_i における更新処理を行うと $\widetilde{itr}(v_i) = \widetilde{itr}(v_j) + 2$ となり，$\widetilde{itr}$ は整合性のとれた状態ではなくなる．$itr(v_i) = itr(v_j)$ と仮定すると，節点 v_i における更新処理を行うと $\widetilde{itr}(v_i) = \widetilde{itr}(v_j) + 1$ となり，$\widetilde{itr}$ は整合性のとれた状態となる．したがって，$itr(v_i) = itr(v_j)$ でなければならない．よって

$$(v_j, v_i) \in E(itr) \tag{8}$$

となる．

$i > j$ の場合：itr は整合性のとれた状態であるから，$itr(v_i) = itr(v_j)$ または $itr(v_i) = itr(v_j) - 1$ の 2 つの場合がある．$itr(v_i) = itr(v_j)$ と仮定すると節点 v_i における更新処理を行うと，$\widetilde{itr}(v_i) = \widetilde{itr}(v_j) + 1$ となり，$\widetilde{itr}$ は整合性のとれた状態とならない．$itr(v_i) = itr(v_j) - 1$ と仮定すると節点 v_i における更新処理を行うと，$\widetilde{itr}(v_i) = \widetilde{itr}(v_j)$ となり，$\widetilde{itr}$ は整合性のとれた状態となる．したがって，$itr(v_i) = itr(v_j) - 1$ でなければならない．よって，

$$(v_j, v_i) \in E(itr) \tag{9}$$

となる． ∎

次に示す補題 2 の意味は以下のとおりである．ある更新状態 itr において v_i を更新可能な節点とする．このとき，定義 4 より節点 v_i には $E(itr)$ の辺の矢印が向いている．補題 2 は，$\widetilde{itr}$ を節点 v_i における更新処理を行った後の更新状態とするとき $E(\widetilde{itr})$ の辺の矢印は v_i から外側に向きが変わることを示している．

補題 2 itr を整合性のとれた更新状態とし，v_i をその時点において更新可能な節点とし，節点 v_i における更新処理をした後の更新状態を $\widetilde{itr}$ とする．このとき，E の辺により v_i につながっている任意の節点 v_j に対して $(v_i, v_j) \in E(\widetilde{itr})$ となる．

[証明] 次の 2 つの場合について，$(v_i, v_j) \in E(\widetilde{itr})$ となることを示せばよい．

$itr(v_j) = itr(v_i) \ \wedge \ j > i$ の場合：節点 v_i における更新処理を行うと，

$$\widetilde{itr}(v_i) = \widetilde{itr}(v_j) + 1 \tag{10}$$

となる．したがって，式 (4) より $(v_i, v_j) \in E(\widetilde{itr})$ となる．

$itr(v_j) = itr(v_i) + 1$ の場合：itr は整合性のとれた状態であることと式 (3) から $j < i$ でなければならない．節点 $v\&_i$における更新処理を行うと

$$\widetilde{itr}(v_i) = \widetilde{itr}(v_j) \tag{11}$$

となる．したがって，式 (4) より $(v_i, v_j) \in E(\widetilde{itr})$ となる． ∎

以下の定理の直感的意味は次のとおりである．アルゴリズム 2 を用いた場合のすべての反復過程における更新状態が整合性のとれた状態を保つための必要十分条件は，枠を置いた直後の更新状態において枠の境界上の任意の E の辺に対して，更新状態を表すグラフの矢印が枠の外側から内側に向かうように枠の形状と移動方法を定めることである．

定理 1 アルゴリズム 2 を用いた場合においてすべての反復過程における更新状態が整合性のとれた状態を保つための必要十分条件は，枠の形状と移動方法が次の条件を満たすことである．P_k 内の最初の節点における更新処理をする前の更新状態を itr_k と表す．任意の k に対して，

$$v_i \in P_k, v_j \notin P_k, (v_i, v_j) \in E \Rightarrow (v_j, v_i) \in E(itr_k)$$

[証明]

必要条件：$v_i \in P_k, v_j \notin P_k$ となるある $(v_i, v_j) \in E$ に対して $(v_j, v_i) \notin E(itr_k)$ と仮定した場合，P_k 内の更新処理を行う間，節点 v_i が更新可能な節点になりえないことを示す．

$v_j \notin P_k$ なので P_k 内の節点における更新処理を行う間は節点 v_j は更新されない．ゆえに，節点 v_i における更新処理を行う直前までの任意の更新状態 itr において $(v_i, v_j) \in E(itr)$ のままである．したがって，P_k 内の更新処理を行う間，節点 v_i は更新可能な節点になりえない．

十分条件：すべての時点における更新状態が整合性のとれた状態になるためには，節点 v_i を更新する直前の更新状態において v_i が更新可能な節点であることを示せばよい．証明は帰納的に行う．まず，v_i を P_k の中から最初に選ばれる節点とする．この節点に E の辺でつながっている任意の節点を v_j とするとき，$v_j \in P_k$ または $v_j \notin P_k$ のいずれかである．$v_j \in P_k$ の場合，v_i の選び方から v_j は v_i より更新回数が 1 回多いか更新回数が同じで番号が大きいかのどちらかである．したがって，節点 v_i における更新処理を行う前の更新状態 itr_k に対し $(v_j, v_i) \in E(itr_k)$ となる．$v_j \notin P_k$ の場合，条件より $(v_j, v_i) \in E(itr_k)$ である．よって，節点 v_i は更新可能な節点である．

次に，すでに P_k 内のいくつかの節点における更新処理が終了したとする．この節点の集合を Q とする．また，P_k 内の節点のうちまだ更新が終了していない節点の集合を R とする．集合 R の中から枠内節点の更新順序に従い節点 v_i を選ぶ．節点 v_i に E の辺でつながっている任意の節点を v_j とするとき，$v_j \in Q$ または $v_j \in R$ または $v_j \notin P_k$ のいずれかである．以下，3 つの場合において節点 v_i における更新処理を行う直前の更新状態を itr とする．$v_j \in R$ の場合，v_i の選び方から節点 v_j は v_i より更新回数 1 回が多いか更新回数が同じで番号が大きいかのどちらかである．したがって，$(v_j, v_i) \in E(itr)$ となる．$v_j \in Q$ の場合，補題 2 より $(v_j, v_i) \in E(itr)$ となる．$v_j \notin P_k$ の場合，$(v_j, v_i) \in E(itr_k)$ である．P_k 内の更新処理を行う間，節点 v_j は更新されないので，$(v_j, v_i) \in E(itr)$ となる．よって，v_i は更新可能な節点である． ∎

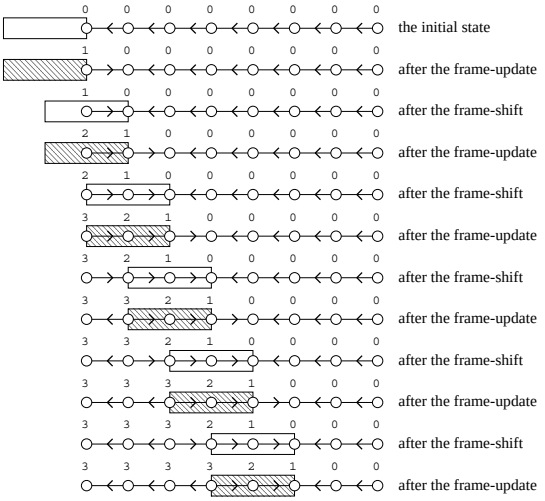


図 1 1 次元の場合の枠の移動方法と更新状態を表すグラフ
Fig. 1 Frame shifting method and updating state of unknowns in the one dimensional case.

この定理 1 の条件が満たされる具体例として，**図 1** に 1 次元の場合の例を示す．

1 次元の場合，枠の形状を連続する複数の節点を含むように構成し，移動方法を節点 1 個分ずつ正の方向へ移動するように定めることにより定理 1 の条件が満たされる．図 1 では，枠を移動した直後の状態および枠内の更新処理がすべて終了した時点での更新状態を表すグラフが示されている．なお，節点は辞書式オーダリングに従い番号付けされているものとする．

図中では，枠を移動した直後の状態を “after the frame shift” と記し，枠内の更新処理がすべて終了した時点の状態を “after the frame update” と記している．

図の中の白丸は節点を示し，その上の数字は節点の更新回数を表している．

1 番上の状態は枠を初期位置に置いた直後の状態を表している．節点は辞書式オーダリングに従い番号付けされているので矢印はすべての節点の間で左に向いている．特に枠の境界では枠の 1 つ右の外部の節点から枠の内部の節点に矢印が向いているので，定理 1 の条件が満たされている．

2 番目の状態は枠内の節点における更新処理が完了した状態を表している．1 番左端の節点と左から 2 番目の節点の間の矢印のみが右向きに変わる．

3 番目の状態は枠を右へ節点 1 個分移動した直後の状態を表している．枠の境界部分である左から 2 番目

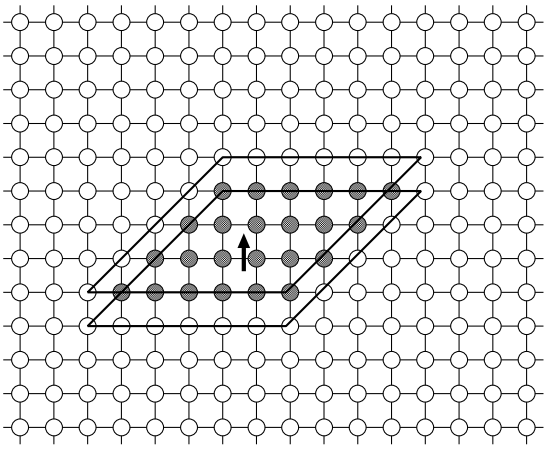


図 2 枠を移動する前と移動した後の 2 つの枠の共通部分
Fig. 2 A set of nodes shared by two frames before and after the movement.

と 3 番目の節点の間の矢印は，ここまでの操作を行う間左向きのままなので，定理 1 の条件が満たされている．以後の図においても同じことがいえる．

以降，2 次元 5 点差分問題，3 次元 7 点差分問題について具体的な実装方法を示す．

3.2 2 次元 5 点差分問題への実装方法

ここでは，2 次元問題を 5 点差分により離散化して得られる連立 1 次方程式を解く場合の枠の形状と移動方法を示す．なお，未知数は辞書式オーダリングに従い番号付けされているものとする．

2 次元問題へ適用する場合の実装方法

- 枠の形状：mx 個の節点を線分上に並べ，この線分を節点 1 個分ずらしながら my 本並べ，これらの節点を含むように右に傾いた平行四辺形を作り枠とする．ここで，平行四辺形に含まれる節点上の未知数の更新に必要なデータがキャッシュメモリに格納できるように mx，my を決める．たとえば **図 2** では長さ 7 の線分を 5 本並べて枠を構成している．
- 枠内の節点の更新順序：平行四辺形の上の線分から下の線分へと順に行う．各線分内は，未知数の番号付けに従い更新を行う．つまり，線分内は左から右の順に行う．ただし，計算領域との共通部分に含まれる節点上の未知数に対してのみ更新を行う．
- 枠の移動方法：次のようなループにより行う．

```
do i0=1,nx+mx+my-1,mx
  do j0=1,ny+my-1
    枠内の未知数の更新
  enddo
```

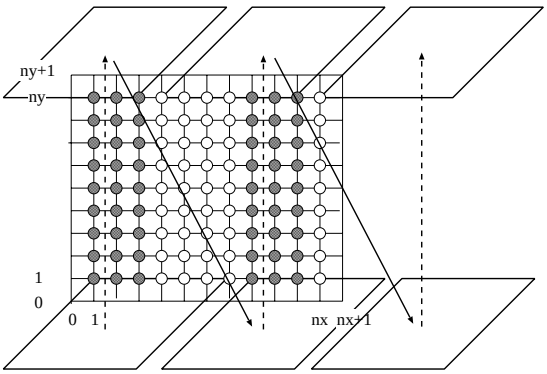


図 3 2 次元の場合の枠の形状と移動方法
Fig. 3 Frame shifting method and updating state of unknowns in the two dimensional case.

```
enddo
```

i0，j0 は，平行四辺形の左上の頂点のそれぞれ x 軸方向のインデックス，
y 軸方向のインデックスである．まず平行四辺形の最上端の線分が計算領域の最下端の左端にくるように i0=1，j0=1 とする．j0 については ny+my−1 までインクリメントする．つまり，平行四辺形の最下端の線分が計算領域の最上端にくるまで平行移動する．そして，i0 を mx インクリメントし，j0 を再度同様に 1 から ny+my−1 までインクリメントする．i0 は平行四辺形の右下の頂点が計算領域の右端を越えるまでインクリメントする．**図 3** を参照．

- error の足し合わせ：平行四辺形の最下端の線分上の節点上の未知数を更新するときのみ足し合わせを行う．これは，my 回ごとに誤差の評価を行うことに対応する．

図 4 の左上の図は枠を移動した直後の時点における更新状態を表しており，定理 1 の条件が満たされている．枠内の更新処理が完了した後は右のようになり，この更新状態において枠を節点 1 個分上に移動すると左下のようになる．この時点で，枠の境界では枠の外部の節点から枠の内部に向けて更新状態を表すグラフの矢印が再び向くようになるので定理 1 の条件が満たされる．

この手法において，平行四辺形を移動した後に新たに含まれる節点の更新処理においては標準的な実装方法における更新処理の場合と同じような主記憶へのアクセスが必要となる．しかし，平行四辺形を移動する前と移動した後の共通部分に含まれる節点の更新においては，移動する前の平行四辺形の更新において使用されたデータがキャッシュメモリに残っていると考え

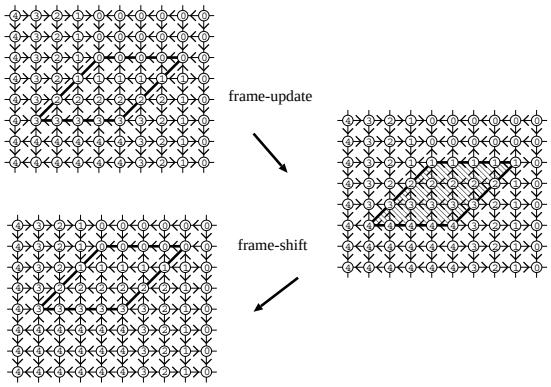


図 4 2 次元問題での枠を移動した直後と枠内の更新処理が完了した時点での更新状態を表すグラフ

Fig. 4 A graph showing the updating states of unknowns just after shifting the frame and after updating unknowns in the frame in the two dimensional case.

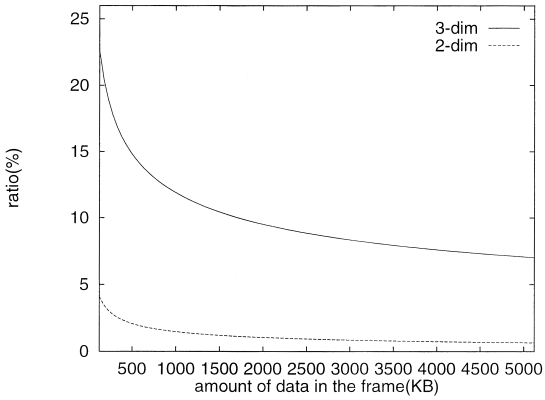


図 5 枠内のデータの総量と $ratio_k(x)$ の関係

Fig. 5 The relation between the total amount of data within the frame and $ratio_k(x)$.

られるので，キャッシュメモリの使用効率が良くなる．

3.3 3 次元 7 点差分問題への実装方法

ここでは，3 次元の問題を 7 点差分により離散化して得られる連立 1 次方程式を解く場合の枠の形状と移動方法を示す．なお，計算領域は $1 \leq i \leq nx$, $1 \leq j \leq ny, 1 \leq k \leq nz$ の範囲とする．ここで，nx, ny，nz は，それぞれ x 軸方向，y 軸方向，z 軸方向の節点の総数である．また，未知数は辞書式オーダリングに従い番号付けされているとする．つまり，未知数の更新は，各 xy 平面内は 2 次元の場合と同様に辞書式オーダリングに従い，z 軸方向に 1 から nz の順に行う．

以下に，3 次元の問題に適用する場合の実装方法を示す．

3 次元問題へ適用する場合の実装方法

- 枠の形状：mx × my の長方形を次のように mz 枚積み上げる．積み上げ方は，1 枚上の長方形に移動するごとに x 軸 y 軸それぞれ節点 1 個分ずつ正の方向にずれるように積み上げる．mx，my，mz の値は，枠内の節点上の未知数の更新に必要なデータがキャッシュメモリに含まれるように設定する．
- 枠内の節点の更新順序：枠の上の長方形から 1 枚ずつ下の長方形の順に行う．各長方形内は，辞書式オーダリングに従い更新を行う．ただし，更新は計算領域との共通部分に含まれる節点上の未知数に対してのみ行う．
- 枠の移動方法：次のようなループにより行う．i0，j0，k0 は枠の最上部の長方形の左手前の頂点のインデックスである．枠は，最初は枠の最上部の長方形が計算領域の最下端の左手前にくるように置く．つまり，i0=1，j0=1，k0=1 とする．k0 は 1 から nz+mz−1 までインクリメントする．つまり，枠の最下端の長方形が計算領域の上に到達するまで移動する．i0 は最下端の長方形の右端の頂点が計算領域を出るまで mx ずつ，j0 は枠の最下端の長方形の左上の頂点が計算領域を出るまで my ずつインクリメントする．

```
do j0=1,ny+my+mz-1,my
  do i0=1,nx+mx+mz-1,mx
    do k0=1,nz+mz-1
      枠内の未知数の更新
    enddo
  enddo
enddo
```

- error の評価：枠の最下端の長方形の節点上の未知数の更新を行うときにのみ error の足し合わせを行い，error の評価をする．これは，標準的実装方法において mz 回ごとに誤差を評価することに対応する．

2 次元 5 点差分の場合と同様，3 次元 7 点差分の場合も枠を移動する前と移動した後の共通部分に含まれる節点の更新においては，キャッシュメモリを効率良く使用することができる．

ところで，2 次元 5 点差分の場合と 3 次元 7 点差分の場合で異なる点は，同じキャッシュメモリサイズのマシンを用いた場合における枠を移動した後に新たに枠に含まれる節点数の枠内節点数に占める割合である．**図 5** は，2 次元および 3 次元の場合について，枠内のデータの総量と枠を移動した後に新たに含まれる節点数の枠内節点数に占める割合の関係を示したもの

である．ここで，割合は次のように定める．枠に含まれる浮動小数点（ここでは倍精度と仮定する）の個数を 1 節点上の未知数の更新で使用するデータの数（2 次元 5 点差分の場合は 7 個，3 次元 7 点差分の場合は 9 個）で割り，枠内節点数を求め，枠の 1 辺あたりの節点数を計算し（すべての辺の節点数は等しいと仮定する），枠の移動後に新たに取り込まれる節点数の枠内節点数に占める割合を計算している．これを式により表すと次のようになる．なお，x は枠内データの総量，$npts$ は 1 節点あたりの浮動小数点データの個数，k は次元数である．

$$ratio_k(x) = 1 - \frac{\left((x/(8 \cdot npts))^{1/k} - 1\right)^k}{(x/(8 \cdot npts))} \quad (12)$$

枠内のデータの総量が 256 KB の場合において，2 次元 5 点差分と 3 次元 7 点差分の $ratio_k(x)$ の値を比較すると，3 次元 7 点差分の場合は枠の 1 辺の節点数は 15 で 19%，2 次元 5 点差分の場合は枠の 1 辺の節点数が 68 で 3%であり，3 次元 7 点差分の場合は $ratio_k(x)$ の値が 2 次元 5 点差分の場合の約 6 倍と大きくなる．ところで，2 次元 5 点差分の場合は 256 KB 程度の小さいサイズで $ratio_2(x)$ の値はすでに 5%以下になっている．これに対して，3 次元 7 点差分の場合は，$ratio_3(x)$ の値を 10%以下にしたい場合でも枠の 1 辺の節点数を 30 以上にする必要があり，そのためには枠のデータ容量を約 1.8 MB 以上確保する必要がある．

3.4 キャッシュミスヒット回数のカウント方法

これまで述べてきた，枠移動法を用いることでキャッシュメモリ容量不足により引き起こされるミスヒットを削減することができる．しかし，この手法を適用するにあたっては，2.1 節で述べたラインコンフリクトにより引き起こされるキャッシュミスヒットにも注意する必要がある．

たとえば 3.2 節の場合では，平行四辺形の線分内に含まれる節点上の未知数の更新にあたっては，配列データへのアクセスは連続となるため，特に問題はない．しかし，2 つの線分に含まれる節点上の未知数の更新に必要なデータのアドレス集合の間は間隔が開いている．この連続するアドレスの集合どうしの間隔は，解く問題の x 軸方向の節点数により変化する．このような理由から，ラインコンフリクトによりキャッシュミスヒットが引き起こされる可能性がある．このキャッシュミスヒット回数は，データのアクセスパターン，キャッシュメモリサイズ，キャッシュラインサイズ，1 セットに格納可能なキャッシュライン数を使って計算により求めることができる．次にキャッシュミスヒット回数をカウントするプログラムを示す．これを用いて，4 章ではキャッシュミスヒット回数をカウントする．なお，ここではキャッシュラインの置換は FIFO 方式により行われるとする．

キャッシュミスヒット回数の計算手順

```
   l_list(1:nset,1:nline)=-1
   nset=(cmsz/nline)/linesz
   next(1:nset)=1
   misshit=0
   do i=1,n
     line=((list(i)-1)/linesz)+1
     set=mod(line-1,nset)+1
     do j=1,nline
       if(l_list(set,j).eq.line) goto 10
     enddo
     l_list(set,next(set))=line
     next(set)=mod(next(set),nline)+1
     misshit=misshit+1
10   continue
   enddo
```

このプログラム中の変数の意味は次のとおりである．

- cmsz：キャッシュメモリサイズ
- linesz：キャッシュラインサイズ
- nline： 1 つのセットに格納できるキャッシュライン数
- list(1:n)： n 個のデータのアクセスパターン
- nset： セットの総数
- l_list(1:nset,1:nline)： 各セットに格納されているライン番号
- misshit： キャッシュミスヒット回数
- next(1:nset)： 次回のキャッシュミスヒット時の格納場所を示すポインタ

なお，初期状態においてキャッシュメモリ内にはここで用いるデータとは無関係なデータが入っているとし，ライン番号配列 l_list は -1 に初期化しておく．

4. 性能評価

本章では，例題として 2 次元 5 点差分問題，3 次元 7 点差分問題を取り上げ，枠内節点数と演算性能の関係を調べた後，枠移動法の有効性を示す．

4.1 テスト環境

性能評価で使用したマシン環境は次のとおりである．

(1) R10000 のマシン

- マシン名：EWS4800/460
- CPU：R10000（200 MHz）
- 1 次キャッシュメモリサイズ：32 KB

- 1 次キャッシュメモリと 2 次キャッシュメモリ間のデータバス幅：128 bit
- キャッシュラインサイズ：64 bites
- キャッシュのセット内ライン数：2
- コンパイラ：f90　オプション：−O
- OS：UX/4800

(2) Pentium3 のマシン

- マシン名：Express5800/120 RC-2
- CPU：Pentium3（933 MHz）
- 主記憶とプロセッサ間のデータ転送速度：133 MHz（64 bit）
- 1 次キャッシュメモリサイズ：16 KB
- 1 次キャッシュメモリと 2 次キャッシュメモリ間のデータバス幅：256 bit
- 2 次キャッシュメモリサイズ：256 KB
- キャッシュラインサイズ：32 bites
- キャッシュのセット内ライン数：8
- コンパイラ：PGI Workstation，PGF90，オプション：−fast −O2
- OS：Red Hat Linux 6.2J

4.2 枠移動法と標準的実装方法における実効性能の比較

図 6 に標準的な実装方法を用いて解いた場合における正方形格子上の節点数と Mflops 値の関係を示す．**図 7** に正方形格子上の節点数が 1000^2 の問題を枠移動法を用いて解いた場合の枠内の節点数と演算性能の関係を示す．

枠移動法を用いた場合の演算性能のピーク値は R10000 で 64 Mflops，Pentium3 で 210 Mflops となる．正方形格子上の節点数が 1000^2 のとき，標準的方法で実装した場合は，演算性能は R10000 で 27.8 Mflops，Pentium3 で 65.6 Mflops である．したがって枠移動法を使うと R10000 で約 2.3 倍，Pentium3 で約 3.2 倍に演算性能が向上しており，2 次元 5 点差分の問題では，枠の形状と移動方法をうまく決めれば，枠移動法は標準的方法よりも効率的な実装方法ということができる．

図 9 に立方体格子上の節点数が 100^3 の 3 次元 7 点差分問題を枠移動法を用いて解いた場合の枠内節点数と演算性能の関係を示す．

同様に 3 次元 7 点差分問題の場合にも数値実験を行った．**図 8** に標準的な実装方法を用いて解いた場合における立方体格子上の節点数と Mflops 値の関係を示す．図 9 に立方体格子の節点数が 100^3 の問題を枠移動法を用いて解いた場合の枠内の節点数と演算性能の関係を示す．図 9 の “isotropic” が枠移動法の

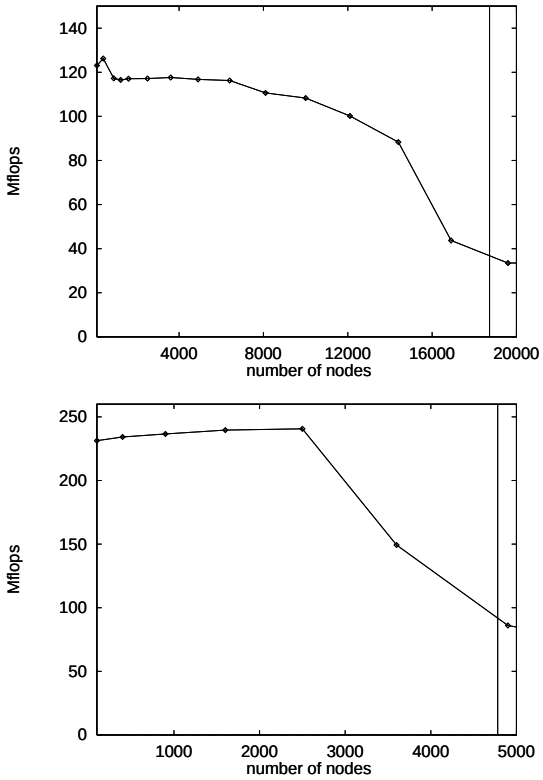


図 6 標準的実装方法を用いて 2 次元 5 点差分問題を解いた場合の正方形格子上の節点数と Mflops 値の関係．上図が R10000 のとき．下図が Pentium3 のとき．グラフの横軸は正方形格子上の節点数，縦軸は Mflops 値．1000^2 のとき演算性能は R10000 で 27.8 Mflops，Pentium3 で 65.6 Mflops．

Fig. 6 The relation between the number of nodes on the square mesh and the performance (Mflops) of solving the 2-dimensional five-point finite difference problem using the standard implementation. The upper figure shows the performance attained on R10000. The lower figure shows the performance attained on Pentium3. The horizontal axes shows the number of nodes on the square mesh. The vertical axes shows the performance (Mflops). When the number of nodes is 1000^2, the performance attained on R10000 is 27.8 Mflops and the performance attained on Pentium3 is 65.6 Mflops.

結果である．立方体格子上の節点数が 100^3 のとき，標準的方法で実装した場合は，演算性能は R10000 で 18.7 Mflops，Pentium3 で 65.6 Mflops である．枠移動法は R10000 で約 1.6 倍，Pentium3 で約 1.8 倍に向上しており，3 次元 7 点差分の問題でも，枠の形状と移動方法をうまく決めれば，枠移動法は標準的方法よりも効率的な実装方法ということができる．

ところが，3 次元 7 点差分問題に適用した場合の演算性能の向上率は 2 次元 5 点差分問題に適用した場合に比べかなり小さくなっている．これは，同じ枠内

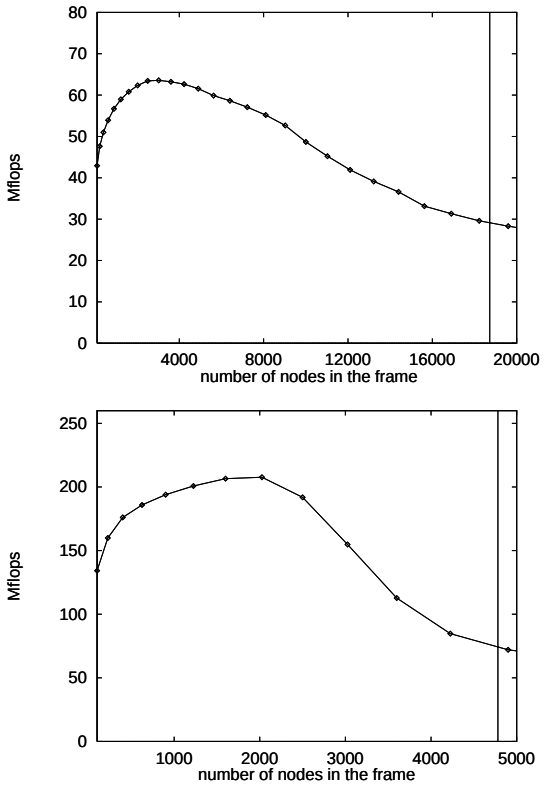


図 7 枠移動法における枠内節点数と Mflops 値の関係．正方形格子上の節点数が 1000^2 のとき．上図が R10000 のとき．下図が Pentium3 のとき．グラフの横軸は枠内節点数，縦軸は Mflops 値．枠の移動前と移動直後の 2 つの枠の共通部分に含まれる節点数が最大となるように枠の縦方向と横方向の節点数を 5 等しくした．また，各グラフの中の縦線は 2 次元 5 点差分問題を解くにあたり 2 次キャッシュメモリに格納可能な節点数の上限を示している．

Fig. 7 The relation between the number of nodes in a frame and the performance (Mflops) attained with the frame shifting method. The upper figure shows the performance attained on R10000. The lower figure shows the performance attained on Pentium3. The horizontal axes shows the number of nodes in a frame. The vertical axes shows the performance (Mflops). The numbers of nodes along horizontal and vertical line are equal so that the number of common vertices before and after movement becomes maximum. The vertical line in this graph shows the upper bound of the number of vertices which can be stored in secondary cache memory at solving a 2-dimensional five-point finite difference problem.

に含まれるデータの総量 x に対して 3.3 節で定めた $ratio_3(x)$ の値が $ratio_2(x)$ の値より大きいために，1 回の枠の移動によって一度キャッシュに記憶されたデータが再利用されない割合が 3 次元の場合はより大きく，キャッシュミスヒットがより多くなってしまったためである．

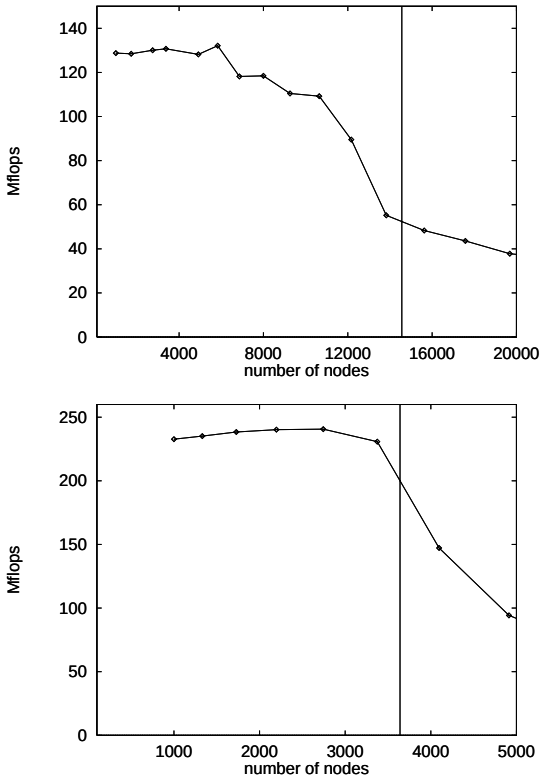


図 8 標準的実装方法を用いて 3 次元 7 点差分問題を解いた場合の立方体格子上の節点数と Mflops 値の関係上が R10000，下が Pentium3 のとき．横軸は立方体格子上の節点数，縦軸は Mflops 値．グラフ内の縦線は 2 次キャッシュメモリに格納可能な節点数の上限を示している．100^3 のとき演算性能は R10000 で 18.7 Mflops，Pentium3 で 65.6 Mflops.

Fig. 8 The relation between the number of nodes on the cubic mesh and performance (Mflops) at solving the 3-dimensional seven-point finite difference problem using the standard implementation. The upper figure shows the performance attained on R10000. The lower figure shows the performance attained on Pentium3. The horizontal axes shows the number of nodes on the cubic mesh. The vertical axes shows the performance (Mflops). The vertical line in this graph shows the upper bound of the number of nodes which can be stored in secondary cache memory. When the number of vertices is 100^3, the performance attained on R10000 is 18.7 Mflops and the performance attained on Pentium3 is 65.6 Mflops.

4.3 キャッシュの容量と枠内節点数の関係

2 次元 5 点差分の問題に対しては，図 7 から，演算性能がピークに達してから低下し始める枠内節点数を見ると，R10000 では 4200，Pentium3 では 2000 程度となり，2 次キャッシュメモリに格納可能な節点数の上限に対して R10000 で 21%，Pentium3 で 41%とかなり少ないことが分かる．3 次元 7 点差分の問題に対しても同様で，図 9 から，演算性能がピークに達して

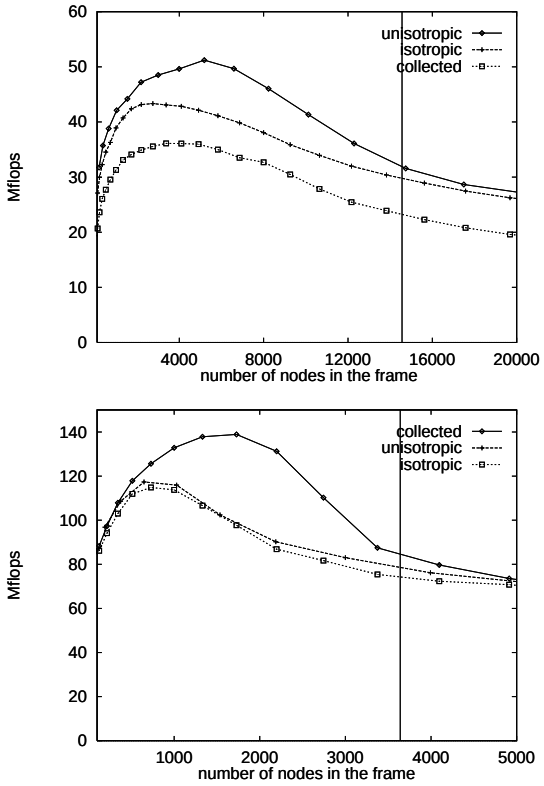


図 9 枠移動法における枠内節点数と Mflops 値の関係．節点数は 100^3．上図が R10000，下図が Pentium3 のとき．横軸は枠内節点数，縦軸は Mflops 値．“isotropic” は，枠を移動する前と移動直後の 2 つの枠の共通部分に含まれる節点数が最大となるように枠の各方向の節点数を等しくした場合．“unisotropic” は x 軸方向，y 軸方向，z 軸方向それぞれの方向の節点数の比を 3:1:1 になるように定めた場合．“collected” は行列，未知数ベクトル，右辺ベクトルをまとめて 1 つの配列として格納した場合．グラフ内の縦線は 2 次キャッシュメモリに格納可能な節点数の上限を示している．

Fig. 9 The relation between the number of nodes in a frame and the performance (Mflops) attained with the frame shifting method. The number of nodes is 100^3. The upper figure shows the performance attained on R10000. The lower figure shows the performance attained on Pentium3. The horizontal axes shows the number of nodes in a frame. The vertical axes shows the performance (Mflops). “Isotropic” indicates that the numbers of nodes along each direction are equal. Thus, the number of common nodes before and after shifting is maximum. “Unisotropic” indicates that the ratio of the number of nodes along x axes, y axes and z axes is taken as 3:1:1. “Collected” means that unknown vector, right-hand vector and coefficient matrix are stored within one array. The vertical line in this graph shows the upper bound of the number of nodes which can be stored in the secondary cache memory.

から下がりはじめる枠内節点数は，R10000 の場合は 2000，Pentium3 の場合は 1000 となり，2 次キャッシュメモリに格納可能な節点数の上限に対して，R10000 が 14%，Pentium3 が 27%とかなり小さくなっていることがわかる．この理由として，ラインコンフリクトによりキャッシュミスヒットが引き起こされることが考えられる．

以下ではラインコンフリクトを検証する．**図 10** は，3 次元 7 点差分の問題に対する枠内節点数と 3.4 節で示したカウント方法を用いて算出したミスヒット指標の関係を示している．ここでのミスヒット指標は，$ratio(x)$ とグラフ上で容易に比較できるようにするため，ミスヒット率にキャッシュラインに含まれる浮動小数点データの個数をかけた値を用いている．なぜなら，$ratio(x)$ の値はデータ単位で算出されているのに対し，ミスヒット率の分子はキャッシュライン単位で計算されるためである．なお，キャッシュミスヒット回数のカウントにおいては枠は 50 回移動させた．この際，行列データ A，未知数データ x，右辺ベクトルデータ b は連続的に格納されていると仮定した．図中の $ratio(x)$ は 3.3 節で与えた $ratio_3(x)$ である．ただし，x は枠内データの総量ではなく枠内節点数である．

図 10 の “isotropic” が算出したミスヒット指標である．図 10 から，ミスヒット指標が最小となる枠内節点数は “isotropic” が R10000 では約 1800，Pentium3 では約 1000 となっていることがわかる．これは図 9 の演算性能が最大となる枠内節点数とほぼ一致する．つまり，性能低下がキャッシュミスヒットで発生していることが確認でき，枠内の節点数がキャッシュに格納可能な節点数に比べ少ないところで発生しているため，ラインコンフリクトによってよって発生してるといえる．

4.4 ラインコンフリクトの回避策

ラインコンフリクトをなるべく回避するための方法は，枠の x 軸方向の長さを他の 2 方向の長さよりも長くする手法である．この方法では，各辺の長さが全て等しい場合に比べ，行列データ，未知数データ，右辺ベクトルデータのそれぞれについて配列への連続アクセスとなる領域が長くなる．これにより，ラインコンフリクトにより引き起こされるキャッシュミスヒットの影響を小さくすることが可能と考えられる．しかし，各方向の長さを等しくした場合に比べ，枠の移動前と移動後の 2 つの枠の共通部分に含まれる節点数が少なくなる．

その手法の有効性を数値実験により示す．図 9 から，

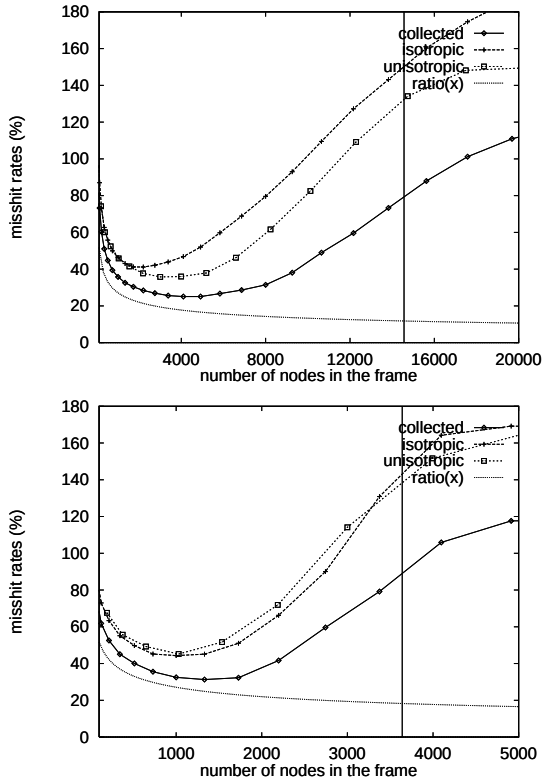


図 10 枠内節点数とミスヒット指標の関係．上図が R10000 のとき．下図が Pentium3 のとき．グラフ内の縦線は 2 次キャッシュメモリに格納可能な節点数の上限を示している．“isotropic” は，枠を移動する前と移動直後の二つの枠の共通部分に含まれる節点数が最大となるように枠の各方向の節点数を等しくした場合．“unisotropic” は x 軸方向，y 軸方向，z 軸方向それぞれの方向の節点数の比を 3:1:1 になるように定めた場合．“collected” は行列，未知数ベクトル，右辺ベクトルをまとめて 1 つの配列として格納した場合．

Fig. 10 The number of nodes in a frame versus the cache miss hit rate estimated by the miss hit counting function. The upper figure shows the function value estimated for R10000. The lower figure shows the function value estimated for Pentium3. The vertical line in this graph shows the upper bound of the number of nodes which can be stored in the secondary cache memory. “Isotropic” indicates that the numbers of nodes along each direction are equal so that the number of common nodes before and after shifting becomes maximum. “Unisotropic” indicates that the ratio of the number of nodes along x axes, y axes and z axes is taken as 3:1:1. “Collected” means that unknown vector, right-hand vector and coefficient matrix are stored within one array.

R10000 においては演算性能のピーク値が “isotropic” より 8 Mflops 向上し 51 Mflops となっている．これは，R10000 ではセット内ライン数が 2 と少ないため，ラインコンフリクトによるキャッシュミスヒットが引き起こされやすく，提案した手法によって，ラインコンフリクトが幾分回避されたためと考えられる．これに対して，Pentium3 ではセット内ライン数が 8 と多いため，ラインコンフリクトにより引き起こされる演算性能の低下が R10000 に比べ小さいと考えられる．そのため，本手法によって改善される割合は，R10000 のときより少ない．

4.5 キャッシュラインへの不必要データの転送の回避策

キャッシュラインへの不必要データの転送を回避する方法は，次のように節点ごとに行列データ A，未知数ベクトル x，右辺ベクトルデータ b を 1 つにまとめ，格納する方法である．たとえば，$axb(1:9,0:nx+1,0:ny+1,0:nz+1)$ のように配列を用意する．ただし，配列は 1 次元目から順番にメモリに展開されているとする．

ここで，1 番目の添字は各節点上の浮動小数点データの種類を示す．また配列の 2 番目から 4 番目の添字は節点の x 軸，y 軸，z 軸方向の番号を示し，nx，ny，nz はそれぞれ x 軸方向，y 軸方向，z 軸方向の節点数である．そして $axb(1:7,1:nx,1:ny,1:nz)$ に行列データ A，$axb(8,0:nx+1,0:ny+1,0:nz+1)$ に未知数ベクトル x，$axb(9,1:nx,1:ny,1:nz)$ に右辺ベクトルデータ b を格納する．

この方法では，3 つの分離した配列を用いる場合に比べ，節点へのアクセスが不連続となる部分における 2 次キャッシュメモリと主記憶間のデータ転送の効率をあげることができる．一方，ある節点における更新において隣接節点上の未知数を参照するとき，2 次キャッシュメモリと 1 次キャッシュメモリ間のデータ移動がキャッシュラインサイズごとに行われるため，その節点の更新では使用しない隣接節点上の行列データ A や右辺ベクトルデータ b を 2 次キャッシュメモリから 1 次キャッシュメモリへ移動させてしまう．

図 9 の 2 つのグラフの中の “collected” はこの方法の演算性能を示している．なお，枠のサイズは各方向の長さが等しくなるように定めた．Pentium3 の場合，“collected” の演算性能のピーク値は “isotropic” の演算性能のピーク値より 18 Mflops 向上し 138 Mflops となっている．一方，R10000 の場合，“collected” の演算性能のピーク値は 36 Mflops となり，“isotropic” の演算性能のピーク値より 9 Mflops 低い．これは，R10000 では，「2 次キャッシュメモリと 1 次キャッシュメモリ間のデータ転送速度」の「2 次キャッシュと主記憶間のデータ転送速度」に対する比が Pentium3 に比べ小さいため，手法の短所が現れたと考えられる．

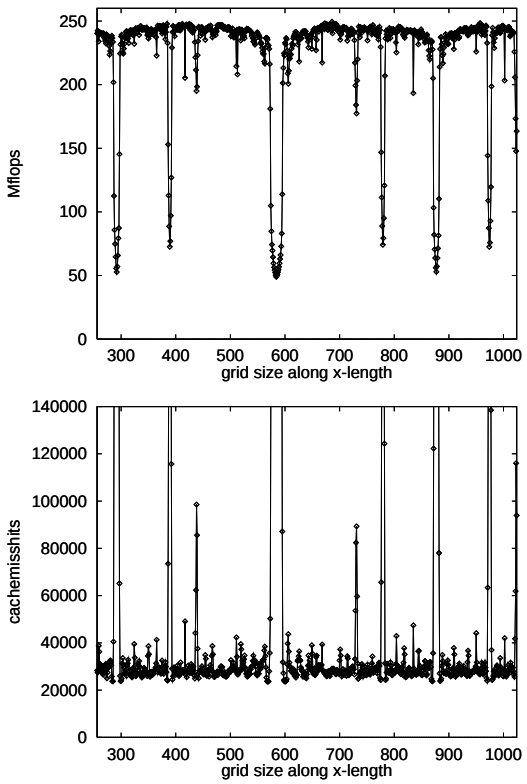


図 11 上図は x 軸方向の節点数と Mflops 値の関係，下図は x 軸方向の節点数とキャッシュミスヒット回数の関係．いずれの結果も Pentium3 での評価結果．y 軸方向の節点数は 1000，枠の各方向の節点数は 40 とした．

Fig. 11 The upper figure shows the relation between the number of nodes along x axes and the performance deterioration (Mflops). The lower figure shows the relation between the number of nodes along x axes and the number of cache miss hit. Both results are obtained on Pentium3. The number of nodes along y axes is 1000. The numbers of nodes along vertical and horizontal lines of a frame are 40.

4.6 計算領域の x 軸方向節点数とキャッシュミスヒット回数の関係

枠移動法の性能評価の過程において，計算領域の x 軸方向の節点数がある特定の値になると演算性能が標準的実装方法を用いた場合と同じ程度にまで低下することを見いだした．また，枠のサイズをいくつか変更し調べてみたところ，この現象は枠のサイズの選び方に依存せず発生するようであった．そこで，ここでは上記の現象が発生する理由について考察を行う．

図 11 の上のグラフは，Pentium3 上で 4.4 節で節明した “collected” の方法を 2 次元の場合に適用し，2 次元 5 点差分問題を解いた場合の x 軸方向の節点数と演算性能の関係である．図 11 の下のグラフは同じ状況下での x 軸方向の節点数と 3.4 節のカウント方法により算出したキャッシュミスヒット回数との関係である．

x 軸方向の節点数と演算性能の関係を見ると，x 軸方向の特定の節点数において演算性能が標準的実装方法を用いた場合と同じ程度まで低下している．この演算性能の低下は x 軸に沿って不規則的に現れる．さらに，この演算性能が低下してから上がるまでの幅は場所によっては x 軸方向の節点数 10 個分から 30 個分と広くなる．一方，x 軸方向の節点数とキャッシュミスヒット回数の関係を見ると，演算性能が大きく低下する場所においてキャッシュミスヒット回数も著しく増大しており，特定の x 軸方向の節点数において演算性能が劣化する理由が，キャッシュミスヒット回数の増大によることは明らかである．

また，このキャッシュミスヒット回数の増大は計算領域の x 軸方向節点数が特定の値をとる場合のみ発生しているため，キャッシュメモリ容量の不足により引き起こされるのではなく，ラインコンフリクトにより引き起こされるといえる．なお，このキャッシュミスヒット回数が著しく増大する現象は，枠のサイズの選び方とは関係なく発生する．なぜなら，3.4 節でも触れたが，連続するアドレス集合間の各々の集合に属するデータのアドレスの差は計算領域の x 軸方向の節点数から決まるため，このアドレスの差が原因となり生じたラインコンフリクトにより引き起こされるキャッシュミスヒットは枠のサイズを変更しても解消されないからである．

以上述べてきた計算領域の x 軸方向節点数が特定の値をとる場合に演算性能が劣化する現象に対しては，各配列の x 軸方向の節点数を演算性能が大きく劣化しないような値まで増やし，実際の計算は必要な領域のみを対象に行うことで対処できる．

5. ま と め

従来，大規模連立 1 次方程式の古典的反復解法を標準的手法を用いてスカラ型サーバ上に実装した場合，大規模な行列データを参照しながら演算を行うためキャッシュミスヒットが多くなり，プロセッサの持っている演算性能を十分に引き出すことが困難であった．本稿では，古典的反復法の 1 つである SOR 法の節点の更新順序を最適化することでキャッシュのヒット率を高めるという枠移動法について述べた．枠移動法の基本的アイディアはすでに示されていたが，計算結果の正当性に対する理論的な証明は与えられておらず，本稿では枠移動法の計算結果と標準的手法の計算結果とが一致することを理論的に証明した．さらに，枠移

動法では，ラインコンフリクトに起因するキャッシュミスヒット，および主記憶とキャッシュメモリ間の不必要なデータ転送の 2 つの要因により，キャッシュメモリに格納可能な節点数に比べ，かなり少ない節点数で演算性能が低下し始めることを述べた．そして，それら 2 つの要因が引き起こす性能低下を抑えるような実装方法を示した．

参 考 文 献

1) 寒川 光：RISC 超並列化プログラミング技法，共立出版 (1995).
2) Douglas, C. Hu, J., Kowarschik, M. Rüde U. and Weiss, C.: Cache Optimization for Structured and Unstructured Grid Multigrid, *ETNA*, Vol.10, pp.21–40 (2000).
3) Hennessy, J. and Patterson, D.: *Computer Architechture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., San Francisco, California (1996).
4) 長谷川秀彦ほか：反復法 Templates，朝倉書店 (1996).
5) Saad, Y.: *Iterative Methods for Sparse Linear Systems*, PWS Publishing Company (1996).

(平成 14 年 7 月 15 日受付)
(平成 14 年 10 月 7 日採録)

丸山 訓英

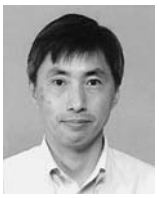
1998 年北海道大学大学院理学研究科数学専攻修士課程修了．同年日本電気株式会社入社．現在 NEC 基礎研究所勤務．

襲田 勉

1995 年東京大学大学院理学研究科情報科学専攻修士課程修了．同年日本電気株式会社入社．大型計算機上でのライブラリの研究開発等に従事．現在 NEC 基礎研究所勤務．計算工学会会員．

鷲尾 巧（正会員）

1961 年生．1989 年大阪大学大学院理学研究科数学専攻修士課程修了．同年日本電気 (株) 入社．コンピュータ技術本部にてコンピュータHWの開発に従事．1991 年より C&C 研究所にて、大規模疎行列連立一次方程式の高速並列解法の研究に従事．1994 年より NEC ヨーロッパ C&C 研究所にて、マルチグリッド法の研究に従事．1999 年より C&C メディア研究所にて、地球シミュレータ用ライブラリの開発に従事．現在 NEC ヨーロッパ C&C 研究所勤務．日本応用数学会会員．



土肥 俊（正会員）

1984 年北海道大学大学院工学研究科精密工学専攻博士課程修了．工学博士．同年 NEC 入社．現在 NEC 基礎研究所勤務．日本計算工学会，可視化情報学会，日本応用数理学会各会員．