共通の係数行列を持つ複数の連立一次方程式のための反復ソルバの実装と性能評価

Implementation and performance evaluation of iterative solver for multiple linear systems that have a common coefficient matrix

○ 今村 成吾，神戸大学 システム情報学研究科, 神戸市灘区六甲台町 1-1, E-mail：almisofte@gmail.com
小野 謙二，理研 計算科学研究機構, 神戸市中央区港島南町 7-1-26, E-mail：keno@riken.jp
横川 三津夫，神戸大学 システム情報学研究科, 神戸市灘区六甲台町 1-1, E-mail：yokokawa@port.kobe-u.ac.jp
Seigo Imamura, Graduate School of System Informatics Kobe University, 1-1 Rokkodai-cho Nada-ku Kobe, Japan
Kenji Ono, RIKEN AICS, 7-1-26 Minatojima-minami-cho, Chuo-ku, Kobe, Japan
Mitsuo Yokokawa, Graduate School of System Informatics Kobe University, 1-1 Rokkodai-cho Nada-ku Kobe, Japan

Capacity computing is a promising scenario for improving performance on upcoming exascale supercomputers. Ensemble computing is an instance and has multiple linear systems associated with a common coefficient matrix. We implement to reduce load cost of coefficient matrix by solving them at the same time and improve performance of several iterative solvers. The maximum performance on SPARC64 XIfx on a node of FX10 was 6.8 times higher than that of naïve implementation. Finally, to deal with the different convergence processes of linear systems, we proposed control methods to skip the calculation of already converged vectors and conduct more acceleration.

1. はじめに

近年のスーパーコンピュータ（スパコン）の高性能な演算性能はCPUなどのアーキテクチャの性能向上とCPU並列数の増加により達成された．スパコンの膨大な計算能力を十分に使うための方法について考えることは重要であり，その1つに，多数の中小規模の問題を同時に扱うCapacity Computing[1] が挙げられる．Capacity Computingの一例として，アンサンブル計算がある．アンサンブル計算は精度の良い予測をするために，様々な計算条件を用いたシミュレーションを必要とする．工業製品の設計で用いる計算流体力学（CFD）シミュレーションにおいてもアンサンブル計算が必要とされている．

非圧縮性CFDアプリケーションでは，圧力Poisson方程式を離散化して得られる大規模な疎行列を持つ連立一次方程式を解くことが必要とされる．連立一次方程式を解くために反復法が一般的に用いられるが，その計算コストは大きい．その理由として疎行列ベクトル積はメモリアクセスが多く，メモリ律速となるため演算性能が低く実行時間が長くなるためである．近年のスパンコンのアーキテクチャはCPUの性能向上の結果，高性能な演算ができる一方で，メモリバンド幅は十分に向上していない．従って演算性能に比べたメモリバンド幅は下がっており，低Byte/Flop（B/F）なマシンとなる傾向がある．そのため，低B/Fマシンでも，高性能を達成できる反復アルゴリズムの開発の必要がある．

Roofline model[2] にはプログラムの実行性能がOperational intensity（Flop/Byte）に影響し，この値が大きいほどピーク性能に近い実行性能が得られると述べられている．Operational intensityを大きくするために，演算に必要なデータロードの削減は有効であり，その例として，Compressed Row Storage（CRS）[3] やBit圧縮[4],[5] などが挙げられる．これらの例は反復法で用いるデータのロード量を減らす実装により，演算性能の向上を達成している．

本研究では，アンサンブル計算の際に得られる共通の係数行列を持つ複数の連立一次方程式のための反復法を，係数行列のロードを削減した低B/F実装について述べる．

2. 共通の係数行列をもつ連立一次方程式

非圧縮性CFDシミュレーションの中で，最も時間コストの大きい部分である圧力ポアソン方程式の反復計算について考える．

$$\nabla(\nabla \boldsymbol{p}) = -div(\frac{\partial \boldsymbol{u}}{\partial t}) \equiv \phi \qquad (1)$$

$\boldsymbol{p}$，$\boldsymbol{u}$，ϕ はそれぞれ圧力，速度，ソース項を表している．(1)をCartesian格子上で，2次精度の有限差分法を用いて離散化することによって，7点のステンシル計算に近似できる．また，この7点ステンシル計算は大規模な疎行列を係数行列とする行列ベクトル積となっている．アンサンブル計算では多くの計算条件を必要とするが，右辺ベクトルが異なるだけで共通の係数行列を持つステンシル計算を必要とすることが多くある．従来なら個々に解を求めるため，同じ係数行列のロードがそれぞれの反復計算ごとに発生する．この係数行列の複数のロードを減らす実装を行うことで反復ソルバの性能向上を行う[6]．

クリロフ部分空間法のBi-CGstab法[7] を用いて性能向上と評価を行う．Bi-CGstab法は前処理により収束性が良くなることが知られている．ステンシル計算の特徴を考慮して前処理には係数行列の形を変えないガウス・ザイデル系のSOR法などが好まれる．3章では前処理の部分についての実装と性能評価を行い，4章では前処理を含めたBi-CGstab法全体について数値実験を行う．

3. ロードコストを減らすための実装

Fortranによる係数行列のロードを減らす実装を施したSOR法のソースコードをList 1に示し，こちらの実装をInner loopと呼ぶことにする．また比較のためのnaïveな実装をList 2に示し，こちらをOuter loopと呼ぶことにする．Lists 1,2に現れる，p, b, bpはそれぞれ $A\boldsymbol{x} = \boldsymbol{b}$ の，圧力（解ベクトル $\boldsymbol{x}$），右辺ベクトル $\boldsymbol{b}$，係数行列 A を表している．また，acは計算セルの残差計算が有効（流体）か無効（壁面内）であるかを判別するフラグである．i, j, k は空間座標を表し，l は右辺ベクトルのインデックスである．

List 1: Pseudo code of SOR using Inner loop.

```
do k=1,kx
do j=1,jx
do i=1,ix
 ndag_e = bp(1,i,j,k)! e
 ndag_w = bp(2,i,j,k)! w
```

```
  ndag_n = bp(3,i,j,k)! n
  ndag_s = bp(4,i,j,k)! s
  ndag_t = bp(5,i,j,k)! t
  ndag_b = bp(6,i,j,k)! b
  dd =     bp(7,i,j,k)! diagonal
  ac =     bp(8,i,j,k)! active
  dx =     1.0 / ac

  do l=1,lx
   pp = p(l,i,j,k)
   bb = b(l,i,j,k)

   ss = ndag_e*p(l,i+1,j  ,k  )&
      + ndag_w*p(l,i-1,j  ,k  )&
      + ndag_n*p(l,i  ,j+1,k  )&
      + ndag_s*p(l,i  ,j-1,k  )&
      + ndag_t*p(l,i  ,j  ,k+1)&
      + ndag_b*p(l,i  ,j  ,k-1)
   dp = ((ss-bb)*dx-pp)*omg
   pn = pp + dp*ac
   p(l,i,j,k) = pn

   de  = dble(bb-(ss-pn*dd))
   res(l) = res(l) + de*de * ac
  end do
 end do
 end do
 end do
```

List 2: Pseudo code of SOR using Outer loop.

```
do l=1,lx
 do k=1,kx
 do j=1,jx
 do i=1,ix
  ndag_e = bp(1,i,j,k)! e
  ndag_w = bp(2,i,j,k)! w
  ndag_n = bp(3,i,j,k)! n
  ndag_s = bp(4,i,j,k)! s
  ndag_t = bp(5,i,j,k)! t
  ndag_b = bp(6,i,j,k)! b
  dd =     bp(7,i,j,k)! diagonal
  ac =     bp(8,i,j,k)! active

  pp = p(i,j,k,l)
  bb = b(i,j,k,l)

  ss = ndag_e*p(i+1,j  ,k  ,l)&
     + ndag_w*p(i-1,j  ,k  ,l)&
     + ndag_n*p(i  ,j+1,k  ,l)&
     + ndag_s*p(i  ,j-1,k  ,l)&
     + ndag_t*p(i  ,j  ,k+1,l)&
     + ndag_b*p(i  ,j  ,k-1,l)
  dp = ((ss - bb)/dd-pp)*omg
  pn = pp + dp*ac
  p(i,j,k,l) = pn

  de  = dble(bb-(ss-pn*dd))
  res(l) = res(l) + de*de * ac
 end do
 end do
 end do
end do
```

両実装の Operational intensity（Flop/Byte）を比較する．SPARC64 IXfx[1] において，配列の宣言を Outer loop は `dimension(ix, jx, kx, lx)`，Inner loop は `dimension(lx, ix, jx, kx)` としたときの問題サイズが 128^3（`ix`, `jx`, `kx`= 128）の場合の i のループ内の Operational intesity を調べる．まず演算量（flops）は Outer loop の場合 30flops，Inner loop の場合 $8 + 23 \times$ `lx` である．次に要求 Bytes について調べる．Outer loop では，bp は一度のロードで 1 ∼ 8 の要素がキャッシュ内に入ることがアドレス値の計算をすることからわかった．また，`p` のロードについても一度のロードで `i+1`, `i`, `i-1`, `j+1`, `j-1`, `k+1`, `k-1` の要素がキャッシュ内に入ることがわかった．加えて，`b`, `res` のロードと `p` のストアを含めて 5 回のロードと 1 回のストアがある．次に Inner loop について調べる．`bp` のロードは Outer loop と同様である．また，`p` のロードも `l`, `i+1`, `i`, `i-1`, `j+1`, `j-1`, `k+1`, `k-1` の要素が一度のロードで同じキャッシュ内に入ることがわかった．しかし，`lx` が 4 より大きい時は `k+1`, `k-1` の要素が一度のロードで同じキャッシュ内に入らないため，2 つのロードが発生する．それから，`b`, `res` のロードと `p` と `res` のストアが `lx` 回ずつ発生する．以上をまとめたものを Tab. 1 に示す．この表から右辺ベクトルが 128 の時，Inner loop の実装は Outer loop に比べて 2.2 倍程度の性能向上が期待できる．

両実装の並列化について，Inner loop には `k` の loop に OpenMP による並列化をしている．また，Outer loop も `k` の loop で並列化している．これにより Outer loop は右辺ベクトルごとに反復法による解を求める場合と同じ実装になっている．

Red-Black SOR(8)（R-B SOR）法について Inner loop の実装をしたコードを List 3 示す．R-B SOR 法は SOR 法のデータ更新の依存性をオーダリングを変えることによりなくした実装である．オーダリングを変えるために `i` の loop にストライドを持たせているため，ロードしたキャッシュ内のデータを十分に使えないことが懸念されるが，R-B SOR 法についても `i` の loop 内の Operational intensity は Tab. 1 と同じになり性能向上が期待できる．

Tab. 1: Characteristics for two types of implementations.

Grid Size = 128^3	Outer	Inner(RHS=`lx` < 4)
Load & Store	$4 + 2$	$4 + 2 \times$ `lx`
Arithmetic	30	$8 + 23 \times$ `lx`
F/B	1.25	1.29 ∼ 2.08

Grid Size = 128^3	Inner(RHS=`lx` > 4)	Inner(RHS=128)
Load & Store	$6 + 2 \times$ `lx`	$6 + 2 \times 128$
Arithmetic	$8 + 23 \times$ `lx`	$8 + 23 \times 128$
F/B	2.08 ∼	2.81

List 3: Pseudo code of Red-Black SOR using Inner loop.

```
do color=0,1
 do k=1,kx
 do j=1,jx
 do i=1+mod(k+j+color,2),ix,2
  ndag_e = bp(1,i,j,k)! e
  ndag_w = bp(2,i,j,k)! w
  ndag_n = bp(3,i,j,k)! n
  ndag_s = bp(4,i,j,k)! s
  ndag_t = bp(5,i,j,k)! t
```

[1]SPARC64 IXfx では，加減乗算を 1flop，除算を 8flops としてカウントする

```
   ndag_b = bp(6,i,j,k)! b
   dd =     bp(7,i,j,k) ! diagonal
   ac =     bp(8,i,j,k) ! active

   dx = 1.0 / dd

   do l=1,lx
    pp = p(l,i,j,k)
    bb = b(l,i,j,k)

    ss = ndag_e * p(l,i+1,j  ,k  )&
       + ndag_w * p(l,i-1,j  ,k  )&
       + ndag_n * p(l,i  ,j+1,k  )&
       + ndag_s * p(l,i  ,j-1,k  )&
       + ndag_t * p(l,i  ,j  ,k+1)&
       + ndag_b * p(l,i  ,j  ,k-1)
   dp = ((ss-bb)*dx-pp)*omg
   pn = pp + dp*ac
   p(l,i,j,k) = pn

    de  = dble(bb-(ss-pn*dd))
    res(l) = res(l) + de*de * ac
   end do

  end do
  end do
  end do
end do
```

3.1 実行性能

Outer loop と Inner loop の両実装について，Tab. 2 に示す SPARC64 IXfx（FX10）での性能評価を Tab. 3 のパラメータで行った．前処理の性能比較のため一定回数反復したときの性能を計測した．評価する問題は，(1) と同じ性質を持つ 3 次元の熱伝導問題 $\nabla^2\phi = 0$ を Dirichlet/Neumann 境界条件で解く問題，および三次元のキャビティフロー問題を扱った．計算は単精度計算である．なお演算性能の計測には *Performance Monitor library*（PMlib）[9] を用いている．PMlib はユーザーが宣告した演算数と処理前後のタイミング測定データから演算性能を推定する．コンパイラオプションは`-Kfast,parallel,ocl,preex,array_private,auto -Kopenmp,simd=2,uxsimd` である．

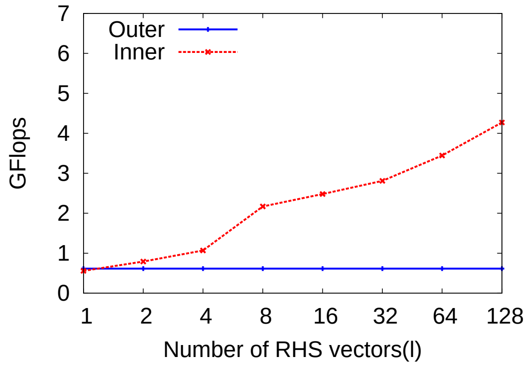
Tab. 2: Specification of a computing node of FX10.

CPU	SPARC64 XIfx
Clock	1.65 GHz
Core	16
Ideal peak performance	211.2 GFlops
Cache	12 MB
Memory	32GB
Bandwidth	85 GB/s

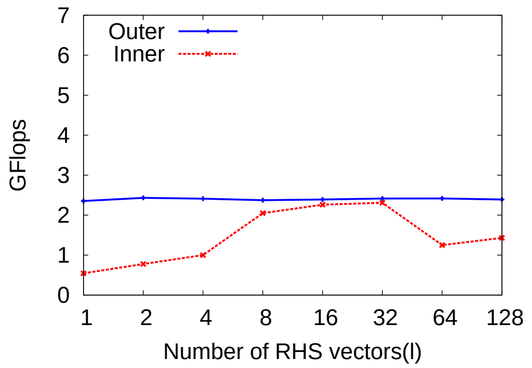
Tab. 3: Investigated parameters.

Solver	SOR method, Red–Black SOR method
Grid size	32^3, 64^3, 128^3, 256^3
Number of threads	1 to 16
Number of pressure vectors	1, 2, 4, 8, 16, 32, 64, 128

SOR 法と R-B SOR 法の逐次実行時の性能を Fig. 1 に示す．Outer loop の実装は SOR 法，R-B SOR 法の両方とも右辺ベクトルによらず性能が一定であることがわかる．これに対して，Inner loop の SOR 法は右辺ベクトルが増えるごとに性能が向上していることがわかる．右辺ベクトルが増えるごとに係数行列のロードの削減の効果が増えたため性能向上につながったことがわかる．しかし，R-B SOR 法では右辺ベクトルが 64 以上で性能劣化が見られ，Outer loop の性能を超える性能向上が見られなかった．これは R-B SOR 法の実装の特徴であるストライドメモリアクセスが原因だと考えられる．右辺ベクトルが増えるごとにストライドの大きさが増えるため，ボトルネックとなったと考えられる．右辺ベクトルが 128 のときの SOR 法の性能について，Outer loop は 0.6GFlops に対して Inner loop は 4.2GFlops と 7 倍の性能向上が得られた．Operational intensity の比較で予想した以上の性能向上が見られた．これは Inner loop の SOR 法の実装で Outer loop の実装に比べて計算順序が変わった結果データの依存性が緩和されデータの依存性による実行性能低下が緩和されたためである．



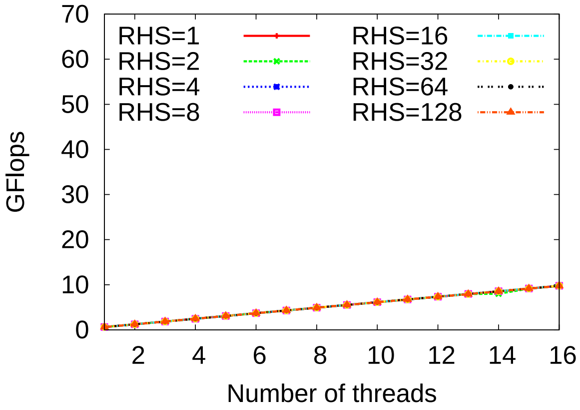
(a) SOR method



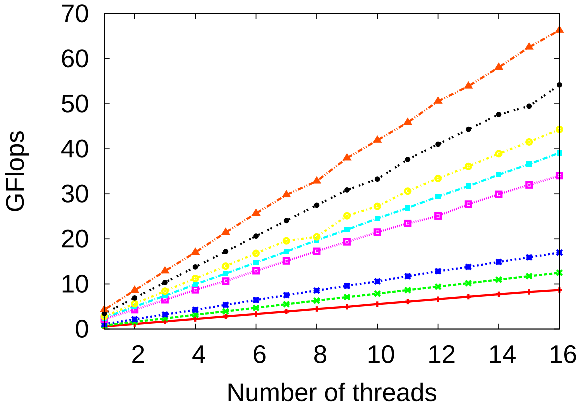
(b) R-B SOR method

Fig. 1: Comparison of sequential performance with a problem size 128^3.

次に，スレッド並列時の性能を Figs. 2, 3 に示す．SOR 法の性能については Outer，Inner loop の実装ともスレッド数に応じて性能が増え，スケーリングしている．R-B SOR 法の性能については Outer loop（Fig. 3(a)）では，スレッド数が大きいところでメモリバウンドの傾向が見られる．しかし，Inner loop（Fig. 3(b)）では SOR 法と同様スケーリングしていることがわかる．Inner loop の実装では SOR 法の右辺ベクトルが 128 のとき最大性能 66.3GFlops を達成している．これはピーク性能の 31.3% であり，Outer loop の SOR 法の最大性能 9.7GFlops と比べると約 6.8 倍の性能向上である．

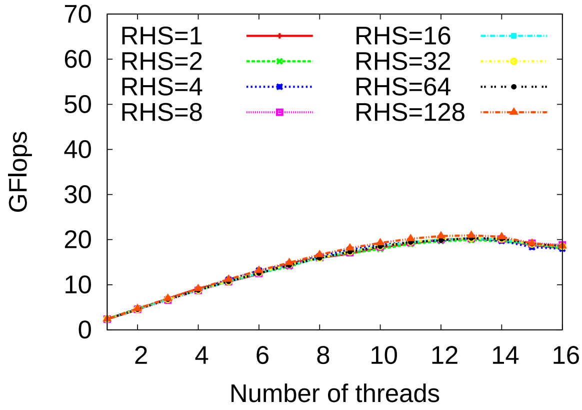


(a) Outer loop

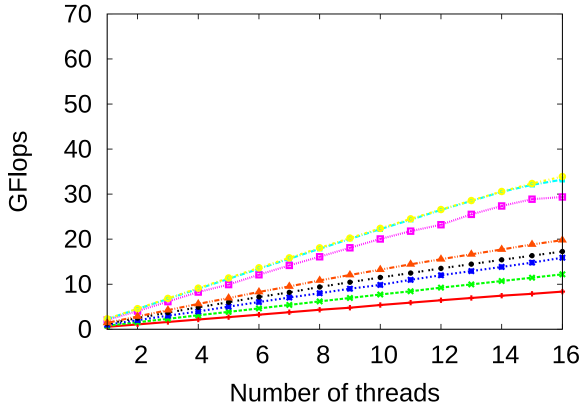


(b) Inner loop

Fig. 2: Threading performance of SOR method with a problem size 128^3.



(a) Outer loop



(b) Inner loop

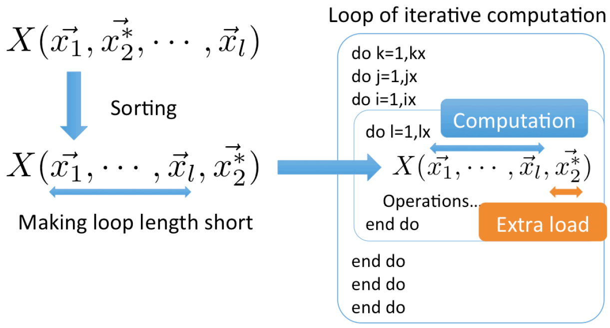
Fig. 3: Threading performance of R-B SOR method with a problem size 128^3.

4. 収束過程を考慮した制御

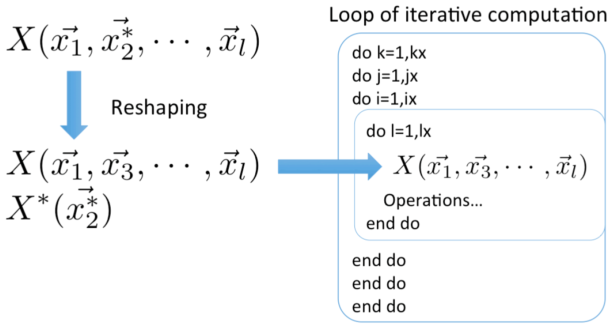
前章までで，共通の係数行列を持つ複数のステンシル計算を一度に扱うことにより Bi-CGstab 法の前処理の部分の性能向上を行った．ここでは Bi-CGstab 法の全体の評価を行う．

右辺ベクトルによって反復計算の収束履歴は異なることを考慮すると，前章までの実装では，すべての解ベクトルが収束するまでの収束過程の中で，収束した解ベクトルに対しても反復計算を行うことになる．この反復計算は本来しなくても良い計算であるため，本節ではこの計算コストを取り除く制御を実装することでさらなる高速化を行う．まず，Outer loop については if 文を l のループの内側に入れ制御を行う．次に Inner loop についても if 文による制御を考えるが，Inner loop の実装で if 文を入れる場合，最内側の l のループの内に if 文を入れることになる．この場合，大量の分岐予測の発生やコンパイラの最適化の難化，またキャッシュラインに乗るデータを十分に使えないことが懸念されるため，演算性能の劣化が考えられる．

if 文を使わない制御について 2 つの方法を考える．1 つは Fig. 4(a) に示す配列内のベクトルを交換する方法である．こちらは反復計算を行わないベクトルを配列の外側の収束していないベクトルと交換し，l のループ長を短くすることで余計な計算コストを取り除いている．もう 1 つは Fig. 4(b) に示す配列を再形成する方法である．こちらは収束したベクトルとそうでないベクトルを別々の配列に作り変え，収束していないベクトルを持つ配列だけを反復計算に用いることで余計な計算コストを取り除いている．



(a) Exchanging vectors.



(b) Reshaping arrays.

Fig. 4: Two types of control methods.

ベクトルの交換と行列の再形成の 2 つの方法の違いについて比較を行う．まず，演算性能について考える．ベクトルの交換の場合，収束したベクトルが配列内にあるため，反復計算のためのデータロードの際，収束したベクトルもデータロードに含まれる．つまり，計算に用いないデータまでロードすることになる．それに比べて行列の再形成では，配列内に収束していないベクトルしかないため，余計なデータロードが抑えることができる．従って，ベクトルの交換よりも良い演算性能が期待できる．しかし，行列の再形成を行うためには if 文やベクトルの交換に比べ多くのワークスペースが必要となる．Algorithm 1 に示す Bi-CGstab 法を用いる場合，反復計算の過程を考慮すると $\boldsymbol{x}_k, \boldsymbol{b}, \boldsymbol{r}_0, \boldsymbol{r}_k^*, \boldsymbol{p}_k, \boldsymbol{q}$ について制御を行う必要がある．制御のためのワークスペースを極力減らすために，Fig. 5 に示すように収束した解ベクトルと右辺ベクトルのための配列 X^*, B^* と空の配列を 1 つ用意し，配列のメモリ領域をローテションさせて使いながら配列の再形成を行うことでメモリ使用量を小さく実装を行った．

Algorithm 1 BiCGstab algorithm

Start with an initial guess $\boldsymbol{x}_0$
Compute $\boldsymbol{r}_0 = \boldsymbol{b} - A\boldsymbol{x}_0$
Choose an arbitrary vector $\boldsymbol{r}_0^*$ such that $\rho_0 = (\boldsymbol{r}_0^* \cdot \boldsymbol{r}_0) \neq 0$, e.g., $\boldsymbol{r}_0^* = \boldsymbol{r}_0$
$\boldsymbol{p}_0 = \boldsymbol{r}_0$
k=0
repeat
  $\boldsymbol{q} = A\boldsymbol{p}_k$
  $\alpha = \rho_k/(\boldsymbol{r}_0^* \cdot \boldsymbol{q})$
  $\boldsymbol{s} = \boldsymbol{r}_k - \alpha\boldsymbol{q}$
  $\boldsymbol{t} = A\boldsymbol{s}$
  $\omega = (\boldsymbol{t} \cdot \boldsymbol{s})/(\boldsymbol{t} \cdot \boldsymbol{t})$
  $\boldsymbol{x}_{k+1} = \boldsymbol{x}_k + \alpha\boldsymbol{p}_k + \omega\boldsymbol{s}$
  $\boldsymbol{r}_{k+1} = \boldsymbol{s} - \omega\boldsymbol{t}$
  $\rho_{k+1} = (\boldsymbol{r}_0^* \cdot \boldsymbol{r}_{k+1})$
  $\beta = (\alpha/\omega)(\rho_{k+1}/\rho_k)$
  $\boldsymbol{p}_{k+1} = \boldsymbol{r}_{k+1} + \beta(\boldsymbol{p}_k - \omega\boldsymbol{q})$
  $\rho_k \leftarrow \rho_{k+1}$
  k++
until $\|\boldsymbol{r}_{k+1}\|_2/\|\boldsymbol{b}\|_2 < \epsilon$

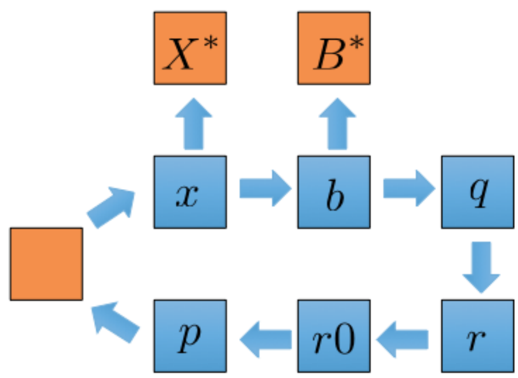


Fig. 5: Rotating memory space of arrays.

4.1 実行時間の比較と制御の性能差

行列の再形成など制御を提案したが，配列の交換，再形成の部分はオーバーヘッドとなるため，オーバーヘッドの時間を含めた実行時間の比較を行う．扱う問題は前章の性能評価で用いたものと同じである．SOR 法を前処理とした Bi-CGstab 法を用いて実験を行う．実行時間を右辺ベクトルの数で割った右辺ベクトル 1 つあたりの実行時間である Average time を縦軸とした実行結果を Fig. 6 に示す．図中，Outer loop は List 2 の 1 の loop の内側に if 文分岐を入れた実装，Inner loop(No) は制御を行わない List 1 と同じ実装で，Inner loop(If) は List 1 の 1 の loop の内側に if 文を入れて制御を行う実装，Inner loop(Exchanging) はベクトルの交換，Inner loop(Reshaping) は行列の再形成による制御を行う実装である．Outer loop の Average time は右辺ベクトルによらずあまり変化がない．それに対して Inner loop は制御の種類に問わず右辺ベクトルの数に応じて Average time が減っている．制御がなくても Inner loop の実装は右辺ベクトルの数が大きいところでは Outer loop よりも高速となった．また，行列の再形成の実装は制御なしの実装よりも実行時間が短く，余計な計算コストを取り除くことによる高速化が達成できた．

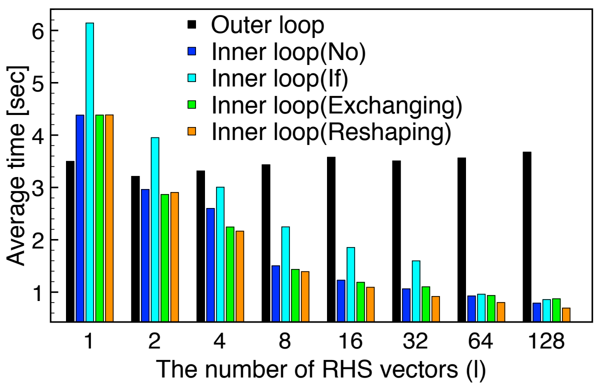


Fig. 6: Comparison of average time with a problem size 128^3.

最後に，制御による性能差を比較するために前処理の演算性能の測定結果を Fig. 7 に示す．制御を入れることにより，収束過程の中で収束したベクトルが増えるごとに演算性能が下がるため，制御なしに比べると制御ありの実装は全体的に性能が下がる．これは収束したベクトルの数が増えるごとに 1 のループ長が短くなり，そのループ長に応じて反復ソルバの性能が Fig. 1(a) に示すように性能が変化するためである．ベクトルの交換と行列の再形成は if 分岐に比べると全体的に性能は良い．また，右辺ベクトルの数が大きいところでベクトルの交換と行列の再形成で差が大きく出た理由は，反復計算内で計算に使わないデータロードの発生が影響したことがわかる．

5. まとめ

共通の係数行列を持つ複数のステンシル計算を同時に取り扱うことで，係数行列のロードコストを削減し，反復ソルバの性能向上を行った．SPARC64 XIfx で，性能評価を行い最大 6.8 倍の性能向上を示し，最大性能はピーク性能の 31.3%を達成した．提案手法の高速化の確認した上で，収束履歴の違いを考慮した，余計な計算を取り除く実装を行い，制御を入れることでさらなる高速化が達成できた．

謝辞

本研究成果の一部は，JSPS 科研費 26286087 の助成を受けたものです．また，本報告書は、文部科学省科学技術試験研究委託事業「近未来型ものづくりを先導する革新的設計・製造プロセスの開発」の助成を受けている。

参考文献

(1) 情報処理学会ハイパフォーマンスコンピューティング研究会, "「特定高速電子計算機施設の共用の促進に関する基本的な方針」に関する意見," (2008) pp.1–7

(2) Samuel W. Williams, Andrew Waterman, David A. Patterson, "Roofline: An Insightful Visual Performance Model for Floating–Point Programs and Multicore Architectures," *Commun, ACM*, Vol. **52** (2009), pp. 65–76.

(3) Willcock, J. and Lumsdaine, A., "Accelerating sparse matrix computations via data compression," Proc. 20th Annual ICS '06 (2006) pp.307–316

(4) Tang, W. T., et al., "Accelerating Sparse Matrix-vector Multiplication on GPUs Using Bit-representation-optimized Schemes," Proc. of SC13 **26** (2013) pp.1–12

(5) Ono, K., Chiba, S., Inoue, S., Minami, K., "Low Byte/Flop Implementation of Iterative Solver for Sparse Matrices Derived from Stencil Computations," High Performance Computing for Computational Science – VECPAR 2014 (2015) Vol.**8969** pp.192–205.

(6) Imamura, S., Ono, K., Yokokawa, M., "PERFORMANCE EVALUATION OF ITERATIVE SOLVER FOR MULTIPLE VECTORS ASSOCIATED WITH A LARGE-SCALE SPARSE MATRIX," Proc. 27th International Conference on Parallel Computational Fluid Dynamics (2015) pp.124–125. (to appear)

(7) Van der Vorst, H. A.,"Bi-CGSTAB: A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems," *SIAM J. Sci. and Stat. Comput.* Vol.**13** No.**2** (1992) pp.631–644

(8) Yokokawa, M.,"Vector-Parallel Processing of the Successive Overrelaxation Method," *Japan Atomic Energy Research Institute JAERI-M Report* No. 88-017 (1988) (in Japanese)

(9) `http://avr-aics-riken.github.io/PMlib/`

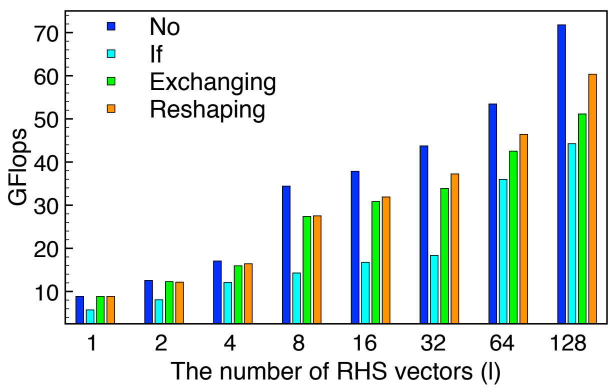


Fig. 7: Comparison of the performance of control methods with a problem size 128^3.