Vol.2018-HPC-167 No.4
2018/12/17

時空間ブロッキングを用いた アジョイント法の性能モデル構築の試み

藤川 隼人†, 片桐 孝洋††, 永井 亨††, 荻野 正雄††

データ同化の手法としてアジョイント法が知られている．このアドジョイント法の演算に時空間ブロッキングを施すことで高性能化する実装方式が提案されている．

本研究では，この高性能化した実装方式の性能モデルを構築することを目的にする．性能モデルを用いることで，時空間ブロッキングを適用したアジョイント法全体の実行時間予測を低いコストで行い，性能チューニングや自動チューニングへ応用することを目指す．

本報告では，アジョイント法の演算の一部である Forward model に対して，スレッド並列時の性能モデル，および，時間ブロッキングを施した際の性能モデルを提案し，それぞれの予測精度の評価を行った．性能評価の結果，提案した性能モデルでは，実行時間の下限推定に問題があること明らかとなった．しかしながら，時空間ブロッキングパラメタなしのスレッド並列化時には，実行時間の下限からのずれに関する相対誤差について，最大で 11.26%で推定可能であることが明らかとなった．

1. はじめに

大規模数値シミュレーションと大容量観測データの融合を図る計算技術として，データ同化が注目されている．データ同化は，現代の気象予報，海洋学などで活用されている[1-6].

データ同化の手法としてアジョイント法がある．この手法は，大規模なシミュレーションを必要とするフェーズフィールドモデルなどに用いられ，大量のデータを扱うことが期待されている．しかし，大規模なデータを扱う際のアジョイント法の問題点の 1 つとして，メモリアクセスの増大による演算効率低下がある．アジョイント法の Forward model の計算において，フェーズフィールドモデルを対象にする場合では差分法を用いる．差分法は近傍の格子点値を使用するステンシル計算を用いた計算方法であるため，大規模で自由度が高いモデルでは，近傍の格子点値のアクセス範囲がキャッシュ容量を超えてしまう．その結果，メモリアクセスが増加し，演算効率低下につながる．これを防ぐためには，高速メモリであるキャッシュ上のデータを出来るだけ再利用して演算を行うブロッキングが必要となる．

ブロッキングによる高性能化について，アジョイント法の Forward model におけるステンシル計算に時空間ブロッキングを適用する手法が我々のグループで提案された[7]. しかし，この手法の性能評価は限られたマシン上でしか行われていない．また，この手法を実用的にするためには，並列数，ブロッキングサイズなどのパラメータを最適化する必要がある．本研究は，時空間ブロッキングの性能評価を行い，性能モデルを作成することを目的とする．性能モデルは, 実行時間の上限と下限を見積もるのに必要であり, コードチューニングを行う場合に加えて，オートチューニング（AT）に利用することで，AT 時間の削減に寄与することができる．

本研究の最終的な目標は，アジョイント法全体の時空間ブロッキングの性能モデル作成であるが，ここではアジョイント法の計算の一部である Forward model の計算のみを取り扱う．Forward model に対して，スレッド並列時の性能モデル，時間ブロッキングを施した際の性能モデルを提案し，それぞれ予測精度評価を行った．

本原稿の構成は以下の通りである．第 2 章では，データ同化とアドジョイント法の概要を説明する．第 3 章では，アドジョイント法に適用する時空間ブロッキングについて説明する．第 4 章では，本研究で提案する性能モデルについて説明する．第 5 章で，複数の計算機を用いて性能モデルの評価を行う．最後に，本研究で得られた知見をまとめる．

2. データ同化

2.1 概要

データ同化は，計算機シミュレーションと実測データを融合する数理技術である[1]. もともとデータ同化は気象学, 海洋学の分野で発達してきたもので, 地表や海面を時間的, 空間的に綿密に観測し，大量のデータを蓄積するために使用されてきた．現在では，地震学や材料工学といった分野にも応用されている．データ同化の目的の 1 つに数値モデルの最適化があり，実測データと数値シミュレーションとの乖離度を評価関数とし，この評価関数を最小化することにより尤もらしいモデルに近づけることが可能である[2-6].

データ同化には逐次データ同化と非逐次データ同化がある．逐次データ同化は時系列データを時間ステップごと

† 名古屋大学 大学院情報学研究科
†† 名古屋大学 情報基盤センター 大規模計算支援環境研究部門

に評価する手法であり，代表的な手法として，カルマンフィルタ(KF)，アンサンブルカルマンフィルタ(EnKF)，粒子フィルタ(PF)などがある．逐次データ同化はパラメータ空間の全てを探索するため，原理的には推定値の期待値，分散を調べて最適値まわりの揺らぎを知ることができるが，計算コストは推定する格子点数とパラメータの和である自由度の指数オーダーとなる．

一方で，非逐次データ同化は時系列データ全体を評価し，最適なモデルを選択する手法である．本論文で扱うアジョイント法は非逐次データ同化に分類される[8-11]. 逐次データ同化とは異なりデータ全体を探索するのではなく，評価関数の状態ベクトルに関する勾配に基づき，勾配法によって事後分布が最大になる状態・パラメータ空間の点のみを

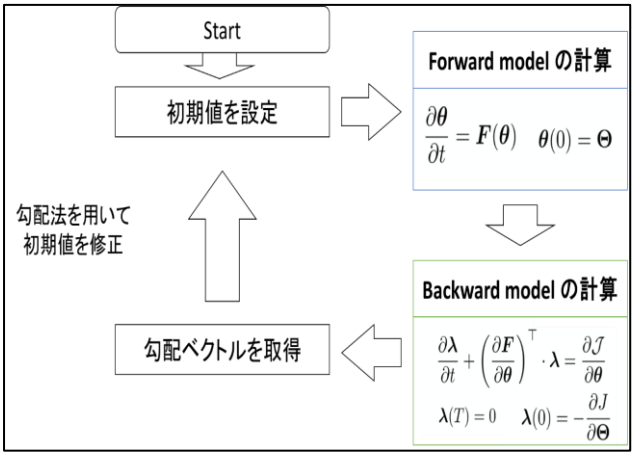

図 1 アジョイント法

探索するため，推定値の期待値，分散は得られないが，計算コストは自由度の線形オーダーとなる．

データ同化において，大規模で自由度が大きなモデルでは，計算コストが小さくなる非逐次データ同化が用いられる．例えば，材料工学分野では大規模シミュレーションを必要とするフェーズフィールドモデルにも，非逐次のデータ同化が用いられている[2,12]．しかし，現在パラメータ推定だけでなく，その不確実性を評価することが求められている．そのため，大規模シミュレーションモデルにおいてもその期待値，分散を得て不確実性を評価する 2nd-order-adjoint 法が提案されている[2]．我々のグループでは，まずはアジョイント法について高性能化を施した. 本論文でも，高性能化を施したアジョイント法のみについて取り扱う．

2.2 アジョイント法

アジョイント法の手順を説明する(図 1)．まず，適当な初期値を設定し，その初期値を用いて Forward model の計算を行う．Forward model では時間が進む方向に計算を行う．次に Forward model の計算結果を用いて Backward model の計算を行う．Backward model では Forward model とは逆に，すなわち，時間が進む方向とは逆に計算を行い，実測データとシミュレーションモデルの差が時間と逆方向に伝搬する．この実測データとシミュレーションモデルとの乖離度を表す評価関数を最小化するため，Backward model の計算によって得られた勾配ベクトルに勾配法を適用し，初期値を更新する．その後は，評価関数が収束するまで反復計算を行う．このアジョイント法において，計算時間の大部分を占めるのが Forward model と Backward model の計算である．その中でも，フェーズフィールドモデルをターゲットとする場合に Forward model の計算に時空間ブロッキングを施し高性能化を試みる手法が提案されている[7]．本論文では，その手法の性能モデル作成を試みる．

```
subroutine forward_model()
  do it = 1, nda
    do j = 1, ny ,1
      do i = 1, nx ,1
        5 点ステンシル計算を用いた格子点の計算
      end do
    end do
  end do
end
```

図 2 naïve な実装における Forward model の
格子点計算カーネル

3. 時空間ブロッキング

3.1 概要

時空間ブロッキングとは，空間方向だけでなく時間方向にもブロッキングを施すことにより，キャッシュのデータを再利用しつつ，次のステップの計算を可能にする考え方である．

Forward model では，計算方法として 5 点ステンシル計算を用いている．5 点ステンシル計算では，現在の時間ステップにおける格子点値の近傍 5 つを利用して次の時間ステップの格子点値を計算する[13-18].

図 2 に，naïve な実装における Forward model の格子点計算カーネルを記載する．ここで，nda は時間ステップ数，ny，nx はそれぞれ y 軸方向，x 軸方向の格子点数を表す．

しかし，大規模で自由度が高いモデルでは，近傍の格子点値のアクセス範囲がキャッシュ容量を超えてしまうという問題がある．この問題が生じる原因は，空間方向への計算を先に行うため，次のステップの計算を行うときにはキャッシュにデータが残っていないからである．そこで，時間方向への計算を先に行うことにより，キャッシュにデータをできるだけ保持するように計算をすることで，メモリアクセスを減少させることができる．

3.2 Forward model における時空間ブロッキング

提案された手法におけるアジョイント法の Forward model の時空間ブロッキングについて説明する．時間ブロッキングサイズを *iblt* とすると，Forward model は以下の

手順で計算される．

- STEP1. *iblt* 先の時間ステップまでの格子点値をピラミッド型に計算する
- STEP2. 計算していない残りの格子点(以下，袖領域とする)の値を *iblt* 先の時間ステップまで計算する．

STEP1，STEP2 の手順を，指定されたループ回数だけの時間ステップまで格子点値を計算し終えるまで繰り返す．

```
subroutine forward_model()
  do it = 1, STEP_NUMBER, iblt
! ピラミッド型計算のループ
   do yy=1, y_tile, 1
    do xx=1, x_tile, 1
     do t = it, it+iblt, 1
      do j = yhead, ytail, 1
       do i = xhead, xtail, 1
        5 点ステンシル計算を用いた格子点の計算
       enddo
      enddo
     enddo
    end do
   enddo
! 袖領域計算のループ
  do yy=1, y_tile, 1
    do xx=1, x_tile, 1
     do t = it, it+iblt, 1
      do j = yhead, ytail, 1
       do i = xhead, xtail, 1
        5 点ステンシル計算を用いた格子点の計算
       enddo
      enddo
     enddo
    enddo
   enddo
```

図 3 Forward model における時空間ブロッキングによる格子点計算カーネル

STEP1，STEP2 の手順を，指定されたループ回数だけの時間ステップまで格子点値を計算し終えるまで繰り返す．図 3 に，時空間ブロッキングを適用した Forward model のアルゴリズムを記載する．ここで，*STEP_NUMBER* は時間ステップ数，*y_tile*，*x_tile* はそれぞれ *y* 軸方向，*x* 軸方向の分割数，*yhead*，*ytail*，*xhead*，*xtail* はスレッド並列化時に各スレッドが担当する問題領域の指定に使用されるための値となっている．

STEP1 のピラミッド型の格子点値の計算について説明する．5 点ステンシル計算において，ある格子点の次の時間ステップの値を計算するためには，自身の値の他に近傍の 4 つの値が必要となる．問題領域の端の格子点値の値を計算するためには，袖領域の格子点の値が必要となる．このため，STEP1 において問題領域の端の格子点は，次の時間ステップの計算をすることができない．したがって，時間ステップが進むごとに次の時間ステップまで値を計算できる格子点は減っていき，連続して計算を行うことが出来る格子点はピラミッド型になる[19].

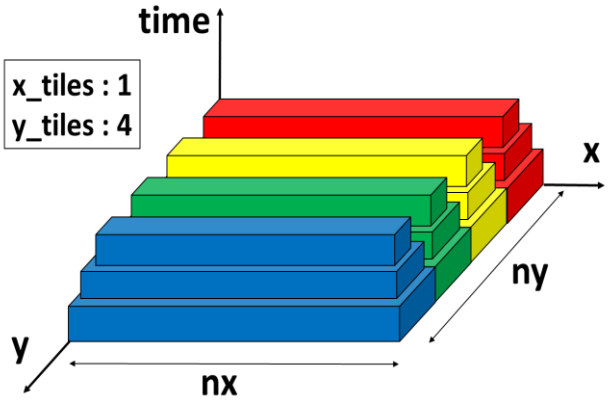

図 4 時空間ブロッキング適用後の 1×4 タイルの格子点

図 4 に時空間ブロッキングを適用した格子点値の計算イメージを記載する．*nx* は *x* 軸方向の格子点数，*ny* は *y* 軸方向の格子点数を表している．本論文での性能評価は，*x* 軸方向の分割数は 1, *y* 軸方向の分割数をスレッド数としたため，格子点のイメージは図 4 のような *x* 軸方向に縦長なピラミッド型になる．図では例として 1×4 タイルの格子点計算イメージを記載している．

4. 性能モデルの提案

ステンシル計算の性能モデルに関する先行研究として，7 点ステンシル計算の空間ブロッキング時の性能モデル[13]がある．この章では，先行研究[13]をもとに性能モデルを構築する．

4.1 スレッド並列を行った性能モデル

この節では，スレッド並列を行った際の性能モデルを構築する．いま，アジョイント法の問題の要素数を *E*，性能モデルに用いるベンチマークのループ長を *W*，初回アクセス時にメインメモリからキャッシュラインにデータを持ってくる際，または書き込む際のコストをC_{first} [sec]，データプリフェッチ時の立ち上がりコストは短く無視できるものとして，データ読み込み時，または書き込み時のコストをC_{stream} [sec]とする．このとき，全体のコストC_{total} [sec]は

$$C_{total} = C_{first} + ([E/W] - 1) * C_{stream} \quad \ldots(1)$$

となる．また，解析対象のプログラムの仕様から *E*=*W* となっている．そのため，

$$C_{total} = C_{first} \quad \ldots(2)$$

とする．

計算時間の上限と下限は，ステンシル計算に必要な格子点に対するデータアクセス時に，どれだけキャッシュミスをしているかでモデル化する．図 5 に，5 点ステンシル計算の図を示す．

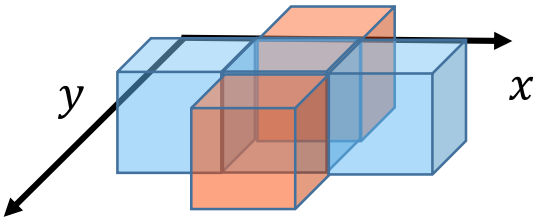

図 5.　5 点ステンシル計算

図 5 にて，計算の方向は *x* 軸方向である．実行時間の下限は，どの格子点もキャッシュミスをしないときであり，このとき，読み込み 1 回書き込み 1 回の合計 2 回となるため，コストは$2C_{total}$となる．

一方，実行時間の上限は，計算対象の格子点の左右の格子点(図 5 の赤色の格子点)の読み込み時にキャッシュミスするときで，読み込み 3 回，書き込み 1 回の合計 4 回となるため，$4C_{total}$とする．

モデルに必要となるコストをC_{first} [sec]を見積もるため，本手法ではベンチマークを行う．このベンチマークとして，先行研究で用いていた Stream ベンチマーク[3]に変更を加えたものを用いた．図 6 に性能モデル評価に使用したベンチマークを示す．

```
!$omp parallel private(i)
  do k=1, STEP_NUMBER, 1
   !$omp do
    do i=1, STREAM_ARRAY_SIZE , 1
        A(i) = SCALAR*B(i) + C(i)
    enddo
    !$omp end do
  enddo
!$omp end parallel
```

図 6. 性能モデルに使用したベンチマーク

図 6 において Stream ベンチマークからの変更点は，OpenMP によるループのスレッド並列化を行ったこと，および，アジョイント法の時間ステップ数（*STEP_NUMBER*）の分 Stream ベンチマークをループとして実行するようにしたことである．解析対象のプログラムでは，スレッドごとに固定の空間ブロッキングサイズを設定する実装がなされている．そのため，ベンチマークにおいてスレッド数を変化させることにより，空間ブロッキングの性能モデルにおけるコストC_{first} [sec]の推定値に用いる．

4.2 時間ブロッキングの性能モデル

この節では，時間ブロッキングを施した際の性能モデルについて記述する．ベンチマークは，スレッド並列のみを行った際のモデルと同じもの（図 6）を用いた．

実行時間の上限と下限の推定について，節 4.1 と同じ考え方を用いると，下限は$2C_{total}$，上限は$4C_{total}$のコストとなる．しかし，この方法では時間ブロッキングサイズを変化させても算出コストに影響が出ないため，時間ブロッキングによって得られる性能向上の傾向を得ることができない．

そこで，袖領域及び時間ブロック内の最初のステップの格子点と，それ以外の格子点で計算コストを変える方法を提案する．

いま，時間ブロッキングサイズを $iblt$，並列数を *N_THREAD*，*n* を 0 以上の整数とする．1 つの格子点を計算するのにかかる時間は，

$$C_{total}/(nx * ny * STEP\_NUMBER/N\_THREAD) \quad \ldots(3)$$

である．ここで，C_{total}は項 4.1 でスレッド並列を行った際に用いたものであり，並列数ごとに値が異なる．

実行時間の上限を推定するにあたって，袖領域及び$n * iblt + 1$のステップの格子点は，キャッシュが外れると考える．このとき読み込みが 3 回，書き込み 1 回で合計 4 回のコストがかかるとすると，

$$4 * C_{total}/(nx * ny * STEP\_NUMBER/N\_THREAD) \quad \ldots(4)$$

と表すことができる．また，上記以外の領域の格子点(図 4 にてピラミッド型になっている領域)はキャッシュが当たるとすると，読み込みが 1 回，書き込み 1 回で合計 2 回のコストがかかるので，

$$2 * C_{total}/(nx * ny * STEP\_NUMBER/N\_THREAD) \quad \ldots(5)$$

と表すことができる．式(4)と式(5)の二つを足し合わせたものを，時間ブロッキングの性能モデルによる実行時間の上限とする．

次に，実行時間の下限の推定であるが，実行時間の下限はすべての格子点値でキャッシュが当たるものとする．よって，すべての格子点でのコストは，

$$2 * C_{total}/(nx * ny * STEP\_NUMBER/N\_THREAD) \quad \ldots(6)$$

と記述できる．

5. 性能評価

5.1 実験環境

実験には以下の計算機を利用した．

・Fujitsu PRIMEHPC FX100 (FX100)

- 名古屋大学情報基盤センター設置
- CPU : Fujitsu SPARK64 Xlfx (2.2GHz), 32 コア
- ノード当たりの理論演算性能: 1,126GFLOPS
- メモリ容量: 32GiB
- キャッシュ構成
  - L1:64KB(コア毎で独立)
  - L2:24MB(ソケット間で共有，ラストレベルキャッシュ)
- メモリアクセス性能 ：240GB/s (読出し、書込みごと), 合計：480GB/s
- 1 ソケット当たり 16 コア，ノード当たり 2 ソケットの NUMA (Non-uniform Memory Access) 構成
- コンパイラ: frtpx : Fujitsu Fortran Driver Version 2.0.0 P-id: T01776-01 (Jun 23 2016 14:10:57)
- コンパイラオプション ：-Kfast -Kopenmp

・ Fujitsu PRIMERGY CX400 (CX400)

- 名古屋大学情報基盤センター設置
- 計算ノード: Fujitsu PRIMERGY CX2550 M1
- CPU : Intel Xeon E5-2600 v3 processor family, 28 コア
- ノード当たりの理論演算性能 ：1,164GFLOPS
- メモリ容量 ：128GiB
- キャッシュ構成
  - L1:64KB(コア毎で独立)
  - L2:256KB(コア毎で独立)
  - L3: 35MB(コア間で共有，ラストレベルキャッシュ)
- 1 ソケット当たり 14 コア，ノード当たり 2 ソケットの NUMA 構成
- コンパイラ ：frt: Fujitsu Fortran Driver Version 1.2.0 P-id: T01778-01 (Jun 22 2016 13:42:10)
- コンパイラオプション ：-Kfast –Kopenmp
- メモリ理論帯域: 136GB/s

5.2 問題設定

表 1 に，実験のための問題設定を示す．

表 1. 問題設定

STEP_NUMBER	128
STREAM_ARRAY_SIZE	1600×1600

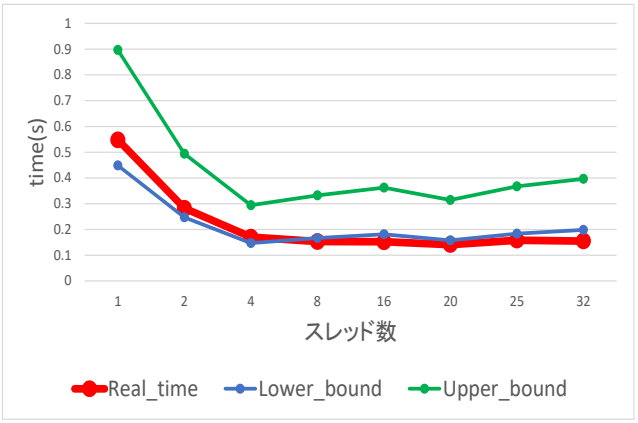

図 7. FX100 での計測結果

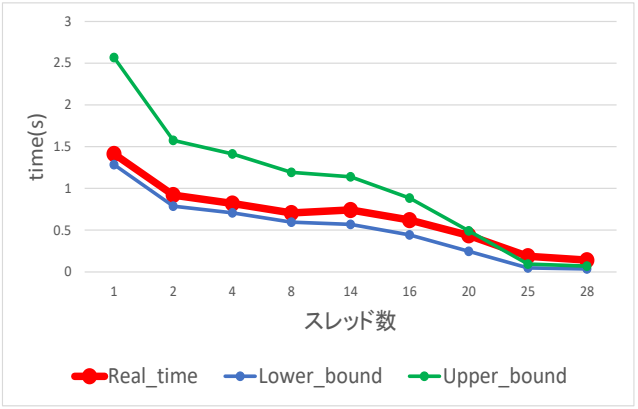

図 8.CX400 での計測結果

計測スレッド数(FX100)	1,2,4,8,16,20,25,32
計測スレッド数(CX400)	1,2,4,8,14,16,20,25,28

表 1 の，*STEP_NUMBER* は，アジョイント法の時間ステップ数，*STREAM_ARRAY_SIZE* は stream ベンチマークのループ長である．

5.3 スレッド並列を行った性能モデルの評価

性能モデルの評価の結果として，FX100 と CX400 での計測結果を，それぞれ図 7，図 8 に示す．なお，アドジョイント法の計算のうち，Forward 計算部分のみ測定している．

図 7，図 8 では，測定した実際の Forward 計算時間（実時間）が赤色のグラフで示されている．また，ベンチマークから算出した下限が青色のグラフ，上限が緑色のグラフで示されている．また，評価するにあたって相対誤差を測った．評価の方法として，実時間が下限を下回った場合，もしくは上限を上回った場合を誤差とする．すると相対誤差E_Rは，

$$E_R = \begin{cases} (T_R - T_U)/T_R & (T_R - T_U > 0) \\ (T_L - T_R)/T_R & (T_L - T_R > 0) \\ 0 & (else) \end{cases} \quad \ldots(7)$$

と表される．ここで，T_Rは実時間，T_Uは上限の時間，T_Lは

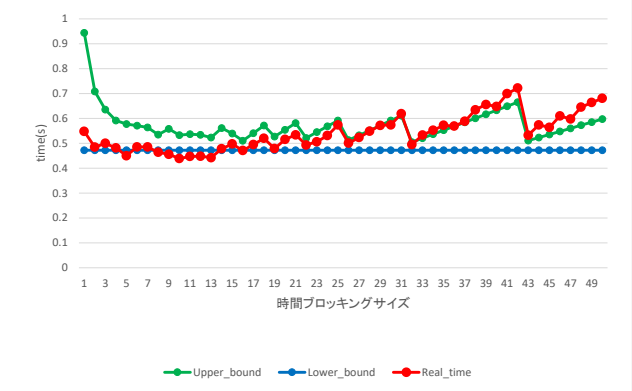

図 9.FX100 における時間ブロッキングの
性能モデルの評価(スレッド数 1)

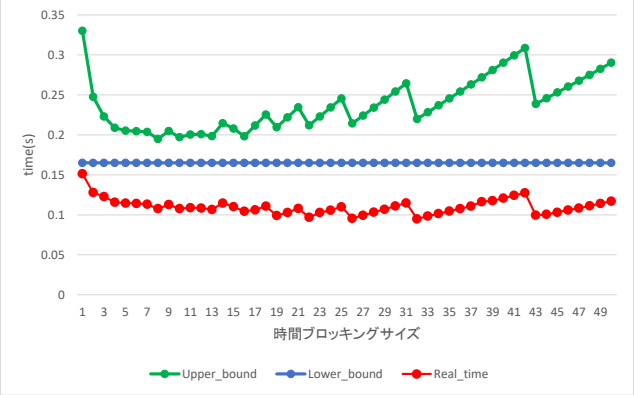

図 10.FX100 における時間ブロッキングの
性能モデルの評価(スレッド数 16)

下限の時間を表している．スレッドごとに相対誤差を算出し，平均したものを全体の相対誤差とした．算出した相対誤差を，表 2 に示す．

表 2. スレッド並列性能モデルの相対誤差

マシン	相対誤差(%)
FX100	10.43
CX400	11.26

図 7 から，FX100 においては，スレッド数が 8 を超えると実時間が下限を下回る．計算時間が最速となるスレッド数が，実時間では 16 スレッドであるが，性能モデルでは 4 スレッドであり異なる．また，表 2 より相対誤差は 10.43%となった．

図 8 から，CX400 においては，スレッド数が 25，28 の時に実時間が上限を上回る．計算時間が最速となるスレッドは，実時間でも性能モデルでも 28 スレッドであり，最速時間の予測の観点では性能モデルはよい精度を示している．また表 2 より，相対誤差は 11.26%となった．

性能モデルの予測精度という観点で見ると，FX100 でも CX400 でも実時間が下限と上限の範囲内から外れる点があり，高精度な性能モデルとは言い難いが，性能傾向を良く表せばよいという尺度もあり，評価尺度の検討が必要である．

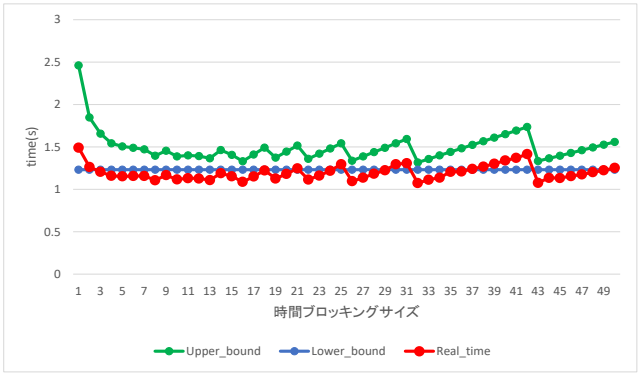

図 11.CX400 における時間ブロッキングの
性能モデルの評価(スレッド数 1)

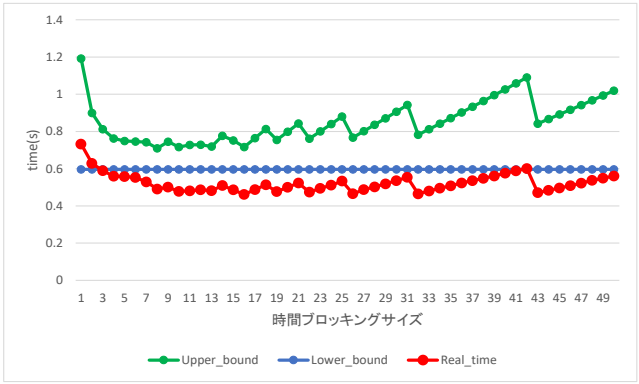

図 12.CX400 における時間ブロッキングの
性能モデルの評価(スレッド数 14)

まとめとして，性能パラメータを変化させた時の性能傾向をつかむという観点で見ると，最速となるスレッド数が 28 で一致した CX400 では，性能傾向がつかめていると言える．誤差の観点で見ると，FX100 の方が CX400 よりも 0.83%精度がよい結果となった．

5.4 時間ブロッキングの性能モデルの評価

時間ブロッキングの性能モデルの評価を，FX100，CX400 それぞれについて評価した．FX100，CX400 両方において，並列化無しの場合と，1 ソケット分のスレッド数(FX100 では 16，CX400 では 14)での評価を記載する．FX100 でのスレッド数 1，16 の結果をそれぞれ図 9，図 10 に，CX400 でのスレッド数 1，14 の結果をそれぞれ図 11，図 12 に示す．また，評価するにあたって相対誤差を測った．相対誤差評価の方法として，5.3 節の式(7)を用いた．時間ブロッキングサイズごとに相対誤差を算出し，平均したものを全体の相対誤差とした． 算出した相対誤差を，表 3 に示す．

表 3. 時間ブロッキングモデルの相対誤差

マシン	スレッド数	相対誤差(%)
FX100	1	2.89
	16	51.09
CX400	1	5.05
	14	15.54

図 9，表 3 より，FX100 にてスレッド数 1 で評価した際には，2.89%の相対誤差となった．一方，図 10，表 3 より，スレッド数 16 で評価した際には実時間が全て下限を下回るといった結果となり，相対誤差は 51.09%となった．

図 11，表 3 より，CX400 にてスレッド数 1 で評価した際には，5.05%の相対誤差となった．一方，図 12，表 3 より，スレッド数 14 で評価した際には，相対誤差は 15.54%となった．スレッド数 1 の際と比較すると実時間が下限を下回っていることが分かる．

FX100 と CX400 両方において，スレッド数が 1 の際は相対誤差が各 2.89%，5.05%と小さく，モデルとして悪くない精度を示しているといえる．一方で，スレッド数を大きくすると実時間が下限を下回るという結果が得られた．特に FX100 では計測したブロッキングサイズ全てで実時間が下限を下回っており，相対誤差は 51.09%モデルとしては良い精度を示さなかった．

6. おわりに

スレッド並列を行うだけの性能モデルでは，FX100 の相対誤差は 10.43%，CX400 での誤差は 11.26%と，どちらも良い精度を示したといえる．最適パラメータを選ぶという点では，CX400 のほうが選ぶことが出来ており，性能傾向をつかめていた．

時間ブロッキングサイズを変化させるモデルでは，FX100，CX400 ともにスレッド数 1 の際は誤差が小さかったが，スレッド数を大きくすると実時間が下限を大きく下回るようになった．この問題については，ベンチマークの見直しや下限の設定を検討しなおすことによって，モデルの精度を高めていこうと考えている．

本性能モデルをアドジョイント法の性能パラメータの自動チューニング(AT)[20]に適用し，AT 時間の削減を行うことは将来課題である．特にアドジョイント法の高速化のため，提案する性能モデルを AT 言語の ppOpen-AT [21]に連結することで，容易に AT を活用できる仕組みの開発は重要な将来課題である．

謝辞

本研究は，科学技術研究費補助金基盤研究（B）「通信回避・削減アルゴリズムのための自動チューニング技術の新展開」(課題番号 16H02823)による．

アドジョイント法の知見とプログラムに関してご提供いただいた，東京大学地震研究所の伊藤伸一助教，長尾大道准教授に感謝いたします．

参考文献

[1] 樋口知之編著，，データ同化入門: 次世代のシミュレーション技術．朝倉書店(2011).

[2] S. Ito, H. Nagao, A. Yamanaka, Y. Tsukada, T. Koyama, M. Kano, and J. Inoue, Data assimilation for massive autonomous systems based on a second-order adjoint method, PhysicalReviewE 94, 043307 (2016).

[3] 長尾大道，樋口知之，地震音波データ同化システムの開発，統計数理，第 61 巻，第 2 号, 257-270(2013).

[4] E. Kalnay, Atmospheric Modeling, Data Assimilation and Predictability (Cambridge University Press, Cambridge, 2003).

[5] T. Tsuyuki and T. Miyoshi, Recent Progress of Data Assimilation Methods in Meteorology, J. Meteorol. Soc. Jpn. Ser. II 85B, pp. 331-361 (2007).

[6] M. Ghil and P. Malanotte-Rizzoli, Advances in Geophysics (Elsevier, New York, 1991), Vol. 33, pp. 141–266.

[7] 池田朋哉，伊藤伸一，長尾大道，片桐孝洋，永井亨，荻野正雄，時空間ブロッキングを用いたアジョイント法の高性能化 ～Forward と Backward 計算～，情報処理学会論文誌：ACS, 11 巻, 1 号, pp.12-26 (2018).

[8] 淡路敏之，蒲地政文，池田元美，石川洋一編著，データ同化: 観測・実験とモデルを融合するイノベーション，京都大学学術出版会 (2009)

[9] J. M. Lewis and J. C. Derber, The use of adjoint equations to solve a variational adjustment problem with advective constraints, Tellus A 37A, 309-322 (1985).

[10] F.-X. Le Dimet and O. Talagrand, Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects, Tellus A 38A, 97-110 (1986).

[11])伊理正夫・久保田光一 高速自動微分法(I), The Japan Society for Industrial and Applied Mathematics (1991)

[12] 小山敏幸，高木知弘，フェーズフィールド法入門，丸善出版, (2013)

[13] Datta, Kaushik, etal. "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures." Proceedings of the 2008 ACM/IEEE conference on Supercomputing. IEEE Press, 2008.

[14] 須田礼仁，一般化菱形行列冪カーネルのための領域分割アルゴリズム，研究報告ハイパフォーマンスコンピューティング(HPC), Vol.2016-HPC-155, No.43, pp.1-9 (2016).

[15] 金光浩，遠藤敏夫，松岡聡， GPU メモリ容量を超える問題規模に対応する高性能ステンシル計算法，研究報告ハイパフォーマンスコンピューティング(HPC), Vol.2012-HPC-137, No.31, pp. 1-6, (2012).

[16] 南武志，高橋康人，岩下武史，中島浩，キャッシュメモリを考慮した 3 次元 FDTD カーネルの性能改善，情報処理学会論文誌コンピューティングシステム

(ACS), Vol.4, No. 2, pp. 70-83, (2011).

[17] Maruyama, Naoya, and Takayuki Aoki. "Optimizing stencil computations for NVIDIA Kepler GPUs." Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna(2014).

[18] Meng, Jiayuan, and Kevin Skadron. "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs." Proceedings of the 23rd international conference on Supercomputing. ACM(2009).

[19] 深谷猛, 岩下武史, 反復型ステンシル計算のマルチコア・メニーコア向け実装に関する考察, 日本応用数理学会「行列・固有値問題の解法とその応用」研究部会, 第 21 回研究会, 2016 年並列/分散/協調処理に関する『松本』サマー・ワークショップ(SWoPP2016), (2016) (口頭発表)

[20] T. Katagiri and D. Takahashi, Japanese Auto-tuning Research: Auto-tuning Languages and FFT, Proceedings of the IEEE, Vol. 106 , Issue 11, pp. 2056-2067 (2018)

[21] T. Katagiri, S. Ohshima, M. Matsumoto, Auto-tuning on NUMA and Many-core Environments with an FDM code, Proceedings of IEEE IPDPSW2017, pp.1399-1407 (2017)