情報処理学会論文誌　コンピューティングシステム　Vol.11　No.1　12–26 (Mar. 2018)

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の計算

池田 朋哉[1,a)]　伊藤 伸一[2]　長尾 大道[3]　片桐 孝洋[4]　永井 亨[4]　荻野 正雄[4]

受付日 2017年7月21日, 採録日 2017年10月13日

概要：大規模数値シミュレーションと大容量観測データを融合するための計算技術として「データ同化」が注目されている．本研究では，非逐次型データ同化であるアジョイント法に対しブロッキング手法を用いた高性能化を検討する．特にアジョイント法のForward計算とBackward計算に含まれるステンシル計算に対し，キャッシュブロッキング手法である時空間ブロッキングを適用した．さらに，複数のForward計算を同時に実行するブロッキングを適用し，本手法の有効性を調査するためFujitsu PRIMEHPC FX100を用いて提案手法の性能評価を行った．Forward計算とBackward計算の時空間ブロッキングと投機的に計算を行うブロッキングからなる階層的なブロッキングを適用することによって，アジョイント法アプリケーション全体で最大1.18倍の速度向上を達成した．

キーワード：データ同化，アジョイント法，ステンシル計算，キャッシュブロッキング，時空間ブロッキング

Optimizing Adjoint Method via Spatial and Temporal Blocking —Forward and Backward Computations

Tomoya Ikeda[1,a)]　Shin-ichi Ito[2]　Hiromichi Nagao[3]　Takahiro Katagiri[4]　Toru Nagai[4]　Masao Ogino[4]

Received: July 21, 2017, Accepted: October 13, 2017

Abstract: Data assimilation (DA) is a computation technique to integrate large-scale numerical simulations and large-scale observed data. In our research, we consider to apply blocking optimization in order to improve the performance of adjoint method for DA, which is classified as non-sequential data assimilation. Spatial and temporal blocking (STB), which is one of cache blocking methods, was applied with respect to stencil computations included in forward computation and backward computation of the adjoint method. Moreover, a blocking technique for execution of multiple forward computations speculatively is applied. We investigated the effectiveness of the proposed method on the Fujitsu PRIMEHPC FX100 supercomputer. We attained a speed up of 1.18x by applying STB for forward computation and backward computation, and the blocking technique for multiple forward computations.

Keywords: data assimilation, adjoint method, stencil computation, cache blocking and spatial and temporal blocking

[1] 名古屋大学大学院情報科学研究科情報システム学専攻
Graduate School of Information Science Nagoya University, Nagoya, Aichi 464–0814, Japan
[2] 東京大学地震研究所
Earthquake Research Institute The University of Tokyo, Bunkyo, Tokyo 113–0032, Japan
[3] 東京大学地震研究所，東京大学大学院情報理工学系研究科
Earthquake Research Institute, Graduate School of Information Science and Technology The University of Tokyo, Bunkyo, Tokyo 113–8654, Japan
[4] 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
Information Technology Center Nagoya University, Nagoya, Aichi 464–0814, Japan
[a)] ikeda@hpc.itc.nagoya-u.ac.jp

1. はじめに

1.1 データ同化，アジョイント法，およびフェーズフィールドモデル

大規模数値シミュレーションと大容量観測データを融合する計算技術として「データ同化」が注目されており，特に現代の気象予報においては，データ同化は必要欠くべからざるものとなっている [1], [2].

データ同化手法の中でも非逐次型データ同化に分類されるアジョイント法 [3], [4], [5] は，時系列データ全体を評価し，数値シミュレーションと実測データとの乖離度を表す評価関数を最小化することによって初期状態とモデルパラメータを同時に推定する手法である．アジョイント法の応用の 1 つとして，フェーズフィールド法 [6], [7] を用いた材料内部の構造の初期状態・パラメータ推定がある．フェーズフィールド法を用いた状態およびパラメータ推定では連続場における数値計算を行うため，高性能化されていないアジョイント法の Naïve な実装ではこれらの推定を完了するために膨大な時間がかかる．この計算時間が増大する原因の 1 つとして，シミュレーションの際に計算機のキャッシュメモリ容量より大きいデータサイズが要求されることがあげられる．たとえば，アジョイント法の Forward 計算および Backward 計算では有限差分法を用いてフェーズフィールド法の計算が行われる．有限差分法は近傍の格子点値を演算に必要とするステンシル計算を用いて解を求める方法である．そのため，非常に大規模な問題では，この近傍の格子点値のアクセス範囲がキャッシュサイズの容量を超えてしまう．結果として，メモリを読み書きする回数が非常に多くなることからメモリアクセスが増大し，演算効率の劇的低下を生じる．このようにアジョイント法はメモリに頻繁にアクセスするメモリインテンシブな処理であるため，メモリ帯域の上限が計算速度の制約となり十分な性能を得ることができない．したがって，メモリアクセスを減少させるためにキャッシュメモリ上にあるデータをできる限り再利用し演算を行う，ブロッキングを用いた高性能化が必要である．

1.2 本論文のオリジナリティ

ブロッキングによる高性能化が必須である一方で，現状ではアジョイント法におけるブロッキングを用いた高性能化手法について十分に検討がなされているとはいえない．

我々は，アジョイント法の Forward 計算に含まれるステンシル計算部分には，キャッシュ有効利用のための時空間ブロッキングを導入し，最適化を行った．さらに複数の Forward 計算を同時実行するブロッキングを適用する方法を提案した [8], [9]．提案手法に加えて，OpenMP を用いたマルチスレッディングや，ファーストタッチ，間接参照の除去等のコード最適化技法を適用し，二次元格子点におけるモデルパラメータを推定する双子問題を設定して性能評価を行った．この既存研究に対して，本研究のオリジナリティは以下である．

1 アジョイント法の Forward 計算および Backward 計算における演算の違いを明らかにする．

2 1 を考慮し，既存手法における Backward 計算における性能を明らかにする．

3 アジョイント法において Forward 計算および Backward 計算の双方に時空間ブロッキングを適用し，さらに複数の Forward 計算を同時に実行するブロッキングを適用したアプリケーション全体としての性能を明らかにする．

1.3 本論文の構成

本論文の構成は以下のとおりである．2 章では，フェーズフィールド法について説明する．3 章では，データ同化について説明する．4 章では，Forward 計算および Backward 計算への時空間ブロッキングの適用と複数の Forward 計算を投機的に実行する階層的なブロッキングを提案する．5 章では，名古屋大学情報基盤センターに設置されているスーパコンピュータ Fujitsu PRIMEHPC FX100 を用いて性能評価を行う．最後に，得られた知見についてまとめを述べる．

2. フェーズフィールド法

凝固や相変態過程におけるミクロな組織形成の時間発展を計算する数値シミュレーション手法として，フェーズフィールド法 [6], [7] がある．フェーズフィールド法は材料科学に限らず，流体力学や工学分野といった様々な領域で応用されている．本研究では，フェーズフィールド法の中でも単純界面移動フェーズフィールドモデル [10] と呼ばれる式 (1) のモデル式を対象とする．

$$\tau\frac{\partial\phi}{\partial t} = \epsilon^2\Delta\phi + \phi(1-\phi)(\phi - \frac{1}{2} + m),$$
$$|m| < \frac{1}{2} \tag{1}$$

式 (1) の ϕ は場所 x，時刻 t における相を表すスカラー量である．また，τ，ϵ，m はモデルパラメータであり，τ は時間の単位，ϵ は空間の単位，m は界面速度を特徴付けるパラメータである．これらのパラメータは全空間で一定値とし，τ と ϵ は既知なパラメータとする．モデル式の第 1 項と第 2 項はそれぞれ拡散項，反応項と呼ばれ，フェーズフィールド法の界面移動はこれら 2 項によって決定される．

フェーズフィールド法では，モデルパラメータと相の初期状態によって相の動的な特性を決定する一方で，実際の実験では初期状態やパラメータを推定できないという問題がある．そこで，式 (1) のモデル式で発展することが分

かっている $\phi(x,t)$ とノイズの加わった時系列データを観測データとし，データ同化手法を適用することによって，相の初期状態 $\phi(x,0)$ と界面速度を表すパラメータ m を推定することを目指す．ここで空間を格子状に離散化し，i 番目の格子点上での相および観測データをそれぞれ，$\phi_i(t)$ および $\phi_i^{obs}(t)$ とすると，両者の関係は以下のように表される．

$$\phi_i^{obs}(t) = \phi_i(t) + \omega_i(t)(i = 1, \ldots, M) \tag{2}$$

$\omega_i(t)$ は平均 0，分散 σ^2 の正規分布に従う観測ノイズ，M は格子点数である．

3. データ同化とアジョイント法

データ同化 [1], [2] は，数値シミュレーションと実測データをベイズ統計学の枠組みで融合する手法である．データ同化を用いることによって，所与の実測データから可能な限り多くの情報を統計的に抽出し，フェーズフィールド法の初期状態とモデルパラメータを同時に推定することを可能にする．データ同化は，元々は気象学や海洋学の分野において用いられていた手法であるが [11], [12]，現在では地震学や材料工学といった多様な分野に応用されている．データ同化の目的の 1 つに数値モデルの最適化があり，数値シミュレーションと実測データとの乖離度を評価関数として，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．

データ同化には，逐次型データ同化と非逐次型データ同化がある．逐次型データ同化は，時系列データを時間ステップごとに評価することによって，システムに含まれる状態を修正し適切なものに収束させるものである．逐次型データ同化の代表的な手法として，カルマンフィルタやアンサンブルカルマンフィルタ，粒子フィルタといった手法がある．これに対して，非逐次型データ同化では，時系列データ全体を評価し，状態ベクトルにモデルパラメータを含める「自己組織化」を行うことにより，初期状態と同時にパラメータの推定を行う．自己組織化された状態ベクトル $\boldsymbol{\theta}(t) \in \mathbb{R}^{M+1}$，初期値 $\boldsymbol{\Theta} \in \mathbb{R}^{M+1}$ を

$$\begin{aligned}
&\boldsymbol{\theta}(t) = (\phi_1(t), \ldots, \phi_M(t), m + \frac{1}{2})^{\mathrm{T}}, \\
&\boldsymbol{\Theta} = (\phi_1(0), \ldots, \phi_M(0), m + \frac{1}{2})^{\mathrm{T}}, \\
&0 < \Theta_i < 1 (i = 1, \ldots, M+1)
\end{aligned} \tag{3}$$

と定義する．ただし $\boldsymbol{x}^{\mathrm{T}}$ は転置を表す．

逐次型と非逐次型の大きな違いの 1 つは，状態・パラメータ推定における計算コストである．それぞれの確率密度関数を表現するため，アンサンブル近似に基づく逐次型データ同化は状態・パラメータ空間全体を探索することによって状態の推定値とその不確実性を評価することができるが，計算コストは空間を離散化する格子点数とモデルパラメータ数の和であるモデルの自由度に対して指数関数的に増加する．一方で，本研究で扱うアジョイント法に基づく非逐次型データ同化では，与えられた初期状態に対する評価関数の勾配を直接計算し，得られた勾配ベクトルを用いて，評価関数を最小化するために勾配法を適用する．勾配法を適用することよって事後分布が最大になるような状態・パラメータ空間のある 1 点を探索するため，不確実性を評価することはできないが，計算コストはモデルの自由度に対して線形的に増加する．そのため，所与のシミュレーションモデルの自由度が非常に大きい場合には，もっぱら非逐次型データ同化が用いられる [3], [4], [5]．材料工学分野においては，材料内部における組織の成長を表現するためにフェーズフィールド法が用いられるが，モデルパラメータの推定だけでなく，その不確実性評価を実施する必要性に迫られている．そのため，大規模シミュレーションモデルに基づくデータ同化においても不確実性評価が可能となるようにするために，2nd-order adjoint 法を応用した新しいアジョイント法が提案されている [13]．本論文では，まずはアジョイント法について高性能化を検討した．

3.1 アジョイント法

アジョイント法では，シミュレーションモデルと実測データの乖離度を示す評価関数を設定する．与えられたシミュレーションモデルの実行時間を $t = 0$ から $t = t_f$ とし，設定した評価関数を式 (4) に示す．

$$\begin{aligned}
&J = \int_0^{t_f} dt\mathcal{J}, \\
&\mathcal{J} = \frac{1}{2} \sum_{t_s \in \tau} \delta(t - t_s) \sum_{i=1}^{M} (\phi_i^{obs}(t_s) - \phi_i(t_s))^2 \\
&\qquad\qquad (s = 1, \ldots, n)
\end{aligned} \tag{4}$$

t_s は観測データを取得した時間，n は観測データを取得した回数，τ は観測時間の集合を表し，また $t_n < t_f$ である．

アジョイント法の計算順序を**図 1** に示す．アジョイント法では，まず始めに適当な初期値を設定し，設定された初期値を用いて Forward 計算を行う．Forward 計算では，設定された初期値をもとにシミュレーションを実行する．Forward 計算に続いて，変分原理によってシミュレーションモデルから導き出されたアジョイントモデルに基づき，Backward 計算を実行する Backward 計算では，シミュレーションモデルと実測データとの差を計算することによって，初期状態に対する評価関数の勾配ベクトルを得る．そして評価関数を最小化するため，得られた勾配ベクトルを用いて勾配法を適用し，設定した初期値を修正する．

我々の先行研究 [8], [9] より，アジョイント法において計算のボトルネックは Forward 計算，Backward 計算，および評価関数の計算であることが判明している．そこで本研究では，Forward 計算と Backward 計算，評価関数の計

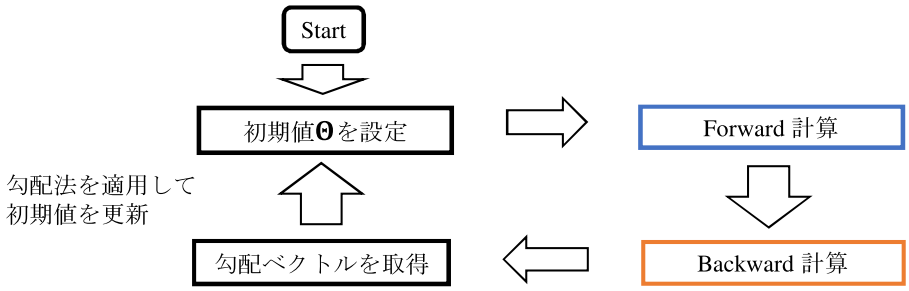

図 1　アジョイント法の概要
Fig. 1　Summary of the computation procedure of the adjoint method.

算に着目する．これらの計算における方程式の導出は Ito ら [13] を参照されたい．本論文では実際に計算される方程式を示す．

3.1.1　Forward 計算

相 $\phi(x,t)$ だけでなく，同時に界面速度を表すパラメータ m も推定するため，式 (1) に加えて m の時間発展方程式を導入し，式 (6) の拘束条件のもとで式 (5) を解く．

$$\begin{cases} \frac{\partial \phi}{\partial t} = \frac{\epsilon^2}{\tau}\Delta\phi + \frac{1}{\tau}\phi(1-\phi)(\phi - \frac{1}{2} + m) \\ \frac{\partial m}{\partial t} = 0 \end{cases} \tag{5}$$

$$|m| < \frac{1}{2}, \qquad 0 < \phi(x,0) < 1 \tag{6}$$

ここで，すべての格子点における相 ϕ ならびにパラメータ m を成分に持つ自己組織化された状態ベクトル $\boldsymbol{\theta}$ を導入し，これの時間発展方程式を Forward 計算と呼ぶ．離散化された Forward 計算を以下に示す．

$$\tau\frac{\partial \theta_i}{\partial t} = \begin{cases} \epsilon^2\Delta_i\theta_i + \theta_i(1-\theta_i)(\theta_i + \theta_{M+1} - 1) \\ \qquad\qquad for \;\; i = 1,\ldots,M, \\ 0 \qquad\qquad otherwise. \end{cases}$$

$$0 < \Theta_i < 1 (i = 1,\ldots,M+1) \tag{7}$$

Δ_i は離散化されたラプラス演算子である．

実際の計算では式 (7) を有限差分法の陽解法で差分化する．また，拡散項は 2 次精度の中心差分で近似するためラプラス行列で表される．したがって拡散項を計算するためにはステンシル計算が必要となる．

ステンシル計算は，ある格子点値を計算するために，その格子点が隣接する格子点の値を必要とする計算である．本研究では空間二次元の問題を扱うため，ある 1 点の格子点値を計算するために 5 点の格子点値が必要となる 5 点ステンシル計算を対象とする．時刻 t における格子点値を計算する 5 点ステンシル計算の例を式 (8) に示す．

$$\begin{aligned} A[x,y,t] = C_1(&A[x-1,y,t-1] + A[x+1,y,t-1] \\ &+ A[x,y-1,t-1] + A[x,y+1,t-1] \\ &- 4A[x,y,t-1]) + A[x,y,t-1] \\ &+ C_2(A[x,y,t-1](1 - A[x,y,t-1] \\ &(A[x,y,t-1] + C_3 - 1)) \end{aligned} \tag{8}$$

ここで，それぞれ $A[x,y,t]$ が格子点値，C_i（$i = 1,2,3$）が定数を表す．

Algorithm 1 Naïve な Forward 計算

```
1: do t = 1, nda
2:   do y = 1, ny
3:     do x = 1, nx
4:       一時配列 temp を用いて格子点値を更新
5:     end do
6:   end do
7: end do
```

Naïve な実装における Forward 計算の計算カーネルを Algorithm 1 に示す．Naïve な実装では問題領域の格子点値を逐一更新している．そのため，大規模自由度系の問題において次の時間ステップにおける格子点値を計算するため，必要な格子点値がキャッシュメモリの容量を超えてしまい，キャッシュミスを引き起こす可能性がある．さらに Forward 計算では異なる初期値を用いて繰り返し Forward 計算を行うため，あらかじめ複数の初期値を用意しておくことで並列に Forward 計算を実行することが可能である．そこで既存研究では，このステンシル計算に対して時空間ブロッキングを適用しキャッシュの再利用性を高め，高性能化を実現した [8], [9].

3.1.2　Backward 計算

変分原理によってシミュレーションモデルから導き出されたアジョイントモデルの計算を Backward 計算と呼ぶ．離散化された Backward 計算を以下に示す．

$$-\tau\frac{\partial \lambda_i}{\partial t} =$$

$$\begin{cases} \epsilon^2\Delta_i\lambda_i + \{-3\theta_i^2 + (4 - 2\theta_{M+1})\theta_i + \theta_{M+1} - 1\}\lambda_i + \frac{\partial \mathcal{J}}{\partial \theta_i} \\ \qquad\qquad for \;\; i = 1,\ldots,M, \\ \sum_{j=1}^{M}\theta_j(1-\theta_j)\lambda_j \qquad\qquad otherwise, \end{cases}$$

$$\lambda(0) = \frac{\partial J}{\partial \boldsymbol{\Theta}},$$

$$\lambda(t_f) = 0 \tag{9}$$

Backward 計算により勾配ベクトル $\frac{\partial J}{\partial \boldsymbol{\Theta}}$ を取得する．得

られた勾配ベクトルを用いて勾配法を適用し，更新された初期値を得る．

Naïve な実装における Backward 計算の計算カーネルを Algorithm 2. に示す．Forward 計算と Backward 計算の大きな違いとして，Backward 計算には，実測データを観測した場合にシミュレーションモデルと実測データとの差を計算する処理が含まれている．この計算に加えて，Backward 計算では，時間ステップごとに実測データが存在するかどうかを確認する必要がある．

Algorithm 2 Naïve な Backward 計算

```
 1: do t = nda, 1, −1
 2:   do y = 1, ny
 3:     do x = 1, nx
 4:       一時配列 temp を用いて格子点値を更新
 5:     end do
 6:   end do
 7:   if (格子点 φ(i) (t)(i=1, . . . , M) に観測データが存在)
 8:     シミュレーションモデルと観測データの差を計算
 9:   end if
10: end do
```

Forward 計算同様，Backward 計算にもステンシル計算が存在する．Backward 計算に含まれるステンシル計算においてもキャッシュミスを引き起こす可能性があるため，Backward 計算のステンシル計算に対して時空間ブロッキングが適用可能である．

4. コード最適化技法

4.1 時空間ブロッキング

アジョイント法の Naïve な実装におけるフローチャートを図 2 に示す．時空間ブロッキングは，空間方向だけでなく時間方向に対してもキャッシュブロッキングを適用しメモリアクセスを減少させ，キャッシュメモリ上にあるデータの再利用性を高める最適化手法である [14], [15], [16], [17], [18], [19]．我々の過去の研究では Forward 計算に含まれるステンシル計算への時空間ブロッキングの適用に加えて，複数の Forward 計算を同時に実行するブロッキング，OpenMP を用いたスレッド並列化，関節参照の除去，ファーストタッチ，および Loop Collapse といった最適化手法を適用した．これに加えて本研究で

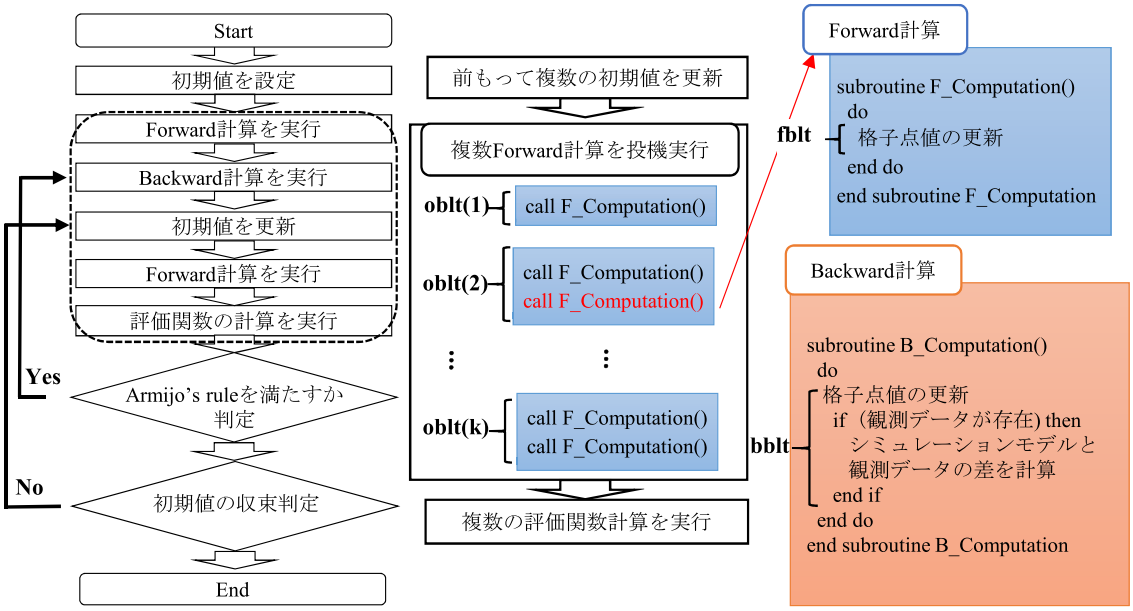

図 2　Naïve な実装のフローチャートと提案手法．
(a) Naïve な実装におけるフローチャート図．点線で囲われた処理に対して，階層ブロッキングを適用する．(b) 提案手法の概要．それぞれ Forward 計算へ適用した時間ブロッキングサイズを fblt，Backward 計算へ適用した時間ブロッキングサイズを bblt，複数 Forward 計算を同時に実行するブロッキングサイズを oblt(i)（i = 1, . . . , k）とする

Fig. 2　A flowchart of naïve implementation and our proposed method. (a) Summary of flowchart on naïve implementation. The hierarchical blocking, consists of STB and the blocking of multiple forward computations, is applied to the area surrounded by dots. (b) Summary of our proposed method. The size of the temporal blocking for forward computation is described as "fblt", the size of the temporal blocking for backward computation is described as "bblt" and the size of blocking of multiple forward computations as "oblt", respectively.

は，Backward 計算に含まれるステンシル計算に対しても同様に時空間ブロッキングを適用する．

大規模自由度系のモデルでは格子点数が非常に膨大であるため，一般的に計算に必要な格子点値のアクセス範囲がキャッシュメモリの容量を超えてしまう．その結果，キャッシュミスが発生してメモリを参照しなければならない．一方で，時空間ブロッキングを適用しブロック領域に対して空間方向および時間方向に一気に演算を実行することで，次の時間ステップにおける格子点値を計算する際に，キャッシュメモリ上に前の時間ステップにおける格子点値を残すことができる．したがってメモリを参照しなければならない回数が減り，メモリアクセス時間を減少させることができる．

4.2　Forward 計算における時空間ブロッキング

一般に差分法における計算では，その格子点自体は計算されないが，隣接する格子点値を計算するために使われる格子点からなる領域が存在し，この領域のことを袖領域と呼ぶ．時空間ブロッキング適用後の全格子点値の計算手順は以下のとおりである．

- STEP1
  空間ブロッキングサイズおよび時間ブロッキングサイズをパラメータとして与え，時間ブロッキングサイズの大きさだけ時間ステップ先の格子点値を計算する．STEP1 では，袖領域にある格子点値を用いた演算は実行せず，計算領域はピラミッド型になる．この計算領域は他の計算領域とオーバラップしないため，冗長な計算は存在しない．
- STEP2
  STEP1 において計算されなかった他の残りすべての格子点値を時間ブロッキングサイズ先まで計算する．よって STEP2 では，袖領域にある格子点の値を要するため，各時間ステップにおいてすべてのスレッドを同期させる必要がある．
- STEP3
  最初の時刻 t から設定された時刻まですべての計算を終えたならば，ステンシル計算を終了する．そうでなければ，STEP1 に戻る．

設定された時刻に到達するまで，STEP1 と STEP2, STEP3 を繰返し行う．

STEP1 のピラミッド型となる格子点値の計算について説明する．次の時間ステップにおいて計算領域の端となる格子点値を計算するためには，袖領域にある格子点値が必要となる．しかし，STEP1 において格子点値を計算する際に，袖領域に存在する格子点値を持っていないため，端となる格子点値を計算することができない．したがって，時間方向へブロッキングを適用する場合は 1 度にすべての格子点値を計算することができないため，ある時間ステップにおいて連続して一気に計算できる格子点は，**図 3** のようなピラミッド型になる．図 3 では，time が時間軸，nx, ny が空間の x，y 軸をそれぞれ表している．また，このときの軸ごとの空間分割数すなわち空間ブロッキングサイズを x_tiles，y_tiles とし，時間ブロッキングサイズは fblt である．

4.3　複数 Forward 計算を同時実行するブロッキング

アジョイント法全体の計算手順（図 2 (a)）をアジョイント法 1 サイクルとする．図 2 (a) に示されている Naïve なアジョイント法の実装の一部を Algorithm 3. に示す．評価関数を計算する前の Forward 計算のフェーズにおいて，Forward 計算と評価関数の計算は，評価関数の計算結果

Algorithm 3 Naïve なアジョイント法の実装

```
 1: 格子点値用の一時配列 temp を静的確保
 2: do i = 1, itr_max
 3:     初期値を更新
 4:     Forward 計算
 5:     評価関数を計算
 6:     if (Armijo'srule を満たす)
 7:       exit
 8:     else
 9:       if
10:         プログラムを終了
11:       end if
12:     end if
13: end do
14: Backward 計算
```

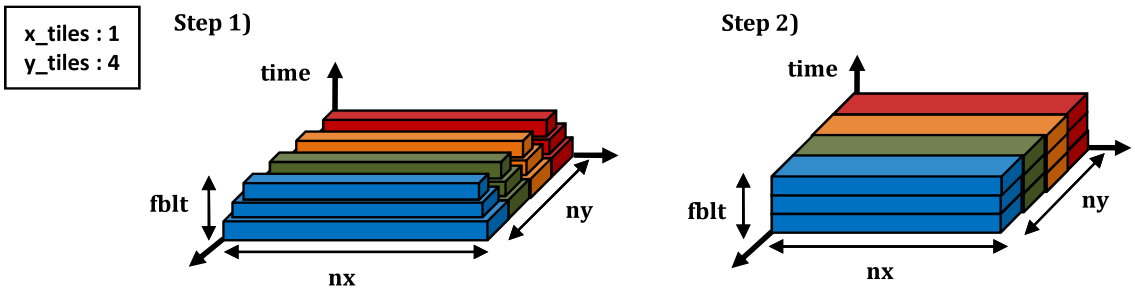

図 3　時空間ブロッキング適用後の Forward 計算におけるステンシル計算のイメージ

Fig. 3　A snapshot of stencil computation of forward computation after applying STB.

と勾配法のステップ幅が Wolfe condition である Armijo's rule [20] を満たすまで繰り返されるため，複数回実行される可能性がある．Armijo's rule は勾配法が効率的に評価関数を降下させることを保証する条件である．この条件は可能な範囲でできる限り大きなステップ幅を見つけ，評価関数を最小化することを目的としている．この条件が満たされる，もしくは初期値が収束するまで，初期値を更新して Forward 計算と評価関数の計算を繰り返し実行する．そのため，計算回数はアジョイント法のサイクルごとに異なっており，予測することができない．

勾配法を適用して評価関数を最小化することによって，初期値を更新する．この初期値の更新には，前の時間ステップにおける勾配ベクトルと勾配法のステップ幅が必要である．しかしステップ幅においては，同時にいくつかのステップ幅に対して候補を用意することができるので，この複数の候補を用いると，前もって複数の初期値の候補を生成することができる．これは，複数の探索開始点による並列探索と似ており，これらの初期値の候補間には依存関係が存在しない．したがって，複数の計算を並列に実行することができる．

Forward 計算では，ある時間ステップの初期値の候補間には依存がないため，それぞれの初期値を用いた複数の Forward 計算を実行することが可能である．この複数の Forward 計算をまとめることによるブロッキング（以降，Forward ブロッキング）を適用することによって 1 度に複数の Forward 計算を実行することが可能になるため，メモリ参照の局所性が高まり，結果として Forward 計算の実行時間が短縮される．さらに，Armijo's rule や初期値収束の条件式を満たしているかどうかの判定処理，および評価関数の計算が並列処理できるため，これらの処理の実行時間も同様に短縮されると考えられる．よって，この Forward ブロッキングは性能向上のための重要な最適化技法となる．

しかし現在の実装では，アジョイント法 1 サイクルにおける複数の Forward 計算を逐次的に実行している．そのため，たとえば Forward 計算の実行回数が 1 回で十分な場合に Forward ブロッキングを適用すると，ブロッキングサイズだけ不要な計算を実行する可能性があるため，高性能化するためにはブロッキングサイズを適切に設定する必要がある．過去の調査（文献 [9] の図 4）より，Forward 計算回数がプログラムの最初と収束直前の時に顕著に増加する一方で，その他の場合では計算回数がたった 1 回となることが分かっている．よって我々は，デフォルトのブロッキングサイズを 1 に設定し，イテレーション内の Forward 計算回数が増加したときにのみブロッキングサイズを動的に大きくする手法を提案した．

Algorithm 4 ブロッキング適用後の Backward 計算

```
 1: do tt = nda, 1, −bblt
 2:   do yy = 1, y_tiles
 3:     do xx = 1, x_tiles
 4:       do t = 1, bblt
 5:         xhead, xtail, yhead, ytail を設定
 6:         if (格子点 φ_i(t)(i = 1, ..., M) に観測データが存在)
 7:           do y = yhead, ytail
 8:             do x = xhead, xtail
 9:               一時配列 temp を用いて格子点値を更新
10:               シミュレーションモデルとの差を計算
11:         else
12:           do y = yhead, ytail
13:             do x = xhead, xtail
14:               一時配列 temp を用いて格子点値を更新
15:   do t = 1, bblt
16:     if (格子点 φ_i(t)(i = 1, ..., M) に観測データが存在)
17:       do yy = 1, y_tiles
18:         do xx = 1, x_tiles
19:           xhead, xtail, yhead, ytail を設定
20:           do y = yhead, ytail
21:             do x = xhead, xtail
22:               一時配列 temp を用いて格子点値を更新
23:               シミュレーションモデルとの差を計算
24:     else
25:       do yy = 1, y_tiles
26:         do xx = 1, x_tiles
27:           xhead, xtail, yhead, ytail を設定
28:           do y = yhead, ytail
29:             do x = xhead, xtail
30:               一時配列 temp を用いて格子点値を更新
```

4.4 Backward 計算における時空間ブロッキング

4.4.1 概要

Backward 計算における時空間ブロッキングを適用したアルゴリズムを Algorithm 4. に示す．Forward 計算と同様，Backward 計算においてもステンシル計算を実行する．Backward 計算の全格子点値の計算手順は，Forward 計算同様に STEP1 から STEP3 を繰り返す．一方で，Forward 計算と大きく異なる処理として，Backward 計算はある時間ステップにおける実測データの有無をステップごとに逐一判定し，存在する場合においてはシミュレーションモデルと実測データとの差を計算する．そのため，Forward 計算と比べて演算量が多い処理となっている．これに加えて，実際のアプリケーションでは，事前にある時間ステップにデータを観測できるかどうかを予測することは難しい．そのため，実測データの有無を逐一判定する条件分岐を取り除くことが原理的に不可能である．よってこの条件分岐は時間ステップのループ内に記載しなければならないため，性能劣化の一因となる．

4.4.2 予備実験

予備実験として，Forward 計算と Backward 計算の実行性能比および各計算内にあるピラミッド型に格子点を計算する処理とその他の残りの格子点値を計算する処理の実行時間比率について調査した．**図 4** (a) は Naïve な実装にお

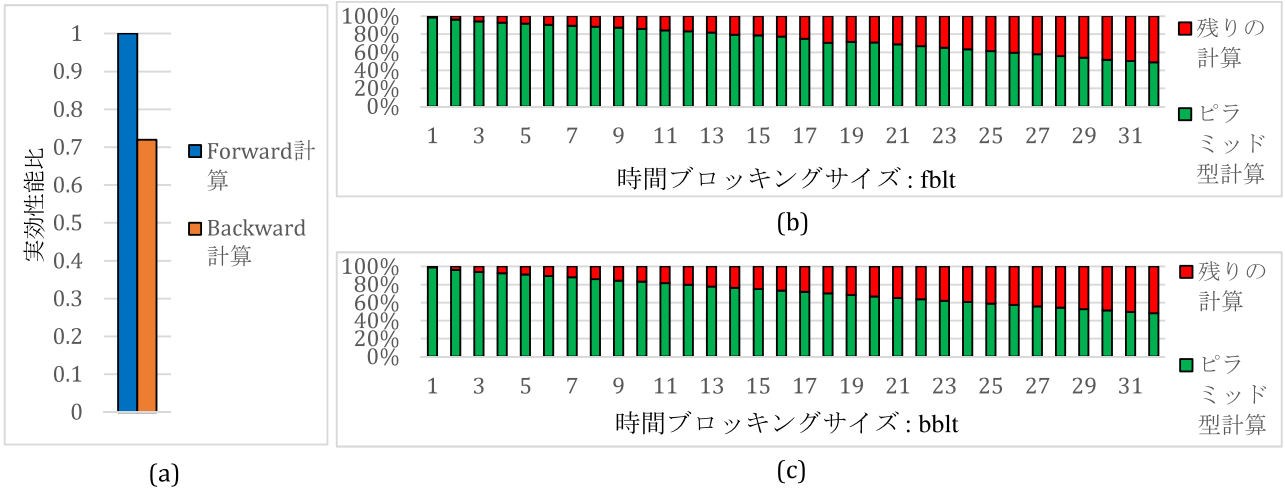

図 4　(a) Forward 計算と Backward 計算の実行性能比．
(b) (c) Forward 計算および Backward 計算におけるピラミッド型の格子点値計算と残りの計算の実行時間内訳

Fig. 4　(a) The performance ratio of forward and backward computation.
(b) (c) Breakdowns of the execution time in the forward and backward computations, the green part representing a ratio required for the pyramidal computation and the red part for the rest.

ける Forward 計算と Backward 計算の性能比を示す．調査より，Backward 計算は Forward 計算に対して約 0.72 倍程度の性能しか出ないことが分かった．図 4 (b)，(c) はそれぞれ Forward 計算および Backward 計算における STEP1, 2 の処理，すなわちピラミッド型の格子点値の計算と残りの格子点値の計算の実行時間内訳を表す．Forward 計算と Backward 計算ともに各処理の内訳はブロッキングサイズに依存しており，双方の計算でほとんど同じ傾向となった．小さいブロッキングサイズを採用したときはピラミッド型の計算の比率が高くなり，ブロッキングサイズが大きくなるにつれて残りの計算の比率が高くなった．さらに，Forward 計算のブロッキングサイズでは 32，Backward 計算のブロッキングサイズでは 31 以上を設定したときに，残りの格子点値計算の実行時間がピラミッド型の格子点値計算の実行時間を上回ることが分かった．残りの計算の実行時間比率が高くなった理由として，ピラミッド型に計算する格子点数と比べて残りの計算の格子点数の増加は著しく，アジョイント法では格子点数に対し線形に計算コストが増加するため，残りの計算の比率がブロッキングサイズとともに高くなったと考えられる．一方で，Forward 計算のブロッキングサイズが 32，Backward 計算のブロッキングサイズが 31 以上を設定したときの総格子点数はピラミッド型の計算のほうが多いにもかかわらず，残りの計算の比率がピラミッド型の計算を上回っている．これはピラミッド型の計算がキャッシュ最適化によって実行時間が短縮され，相対的に残りの計算の実行時間の比率が高くなったと考えられる．

また，指定した時間ステップ内における観測データ数の違いによる性能差について調査を行った．問題空間のメッシュサイズが 1600×1600 かつ空間における観測点数を全格子点としたとき，初期状態を除くすべての時間ステップにおいて観測データが存在する場合と，ある時間ステップにおいてのみ観測データが存在する場合において，それぞれの実行性能を調査した．調査した結果，全時間ステップ（128 ステップ）において観測データが存在する場合は，ある時間ステップにおいてのみデータが観測できる場合と比較して，0.84 倍の性能劣化が発生することが分かった．これは，観測データ数に比例してシミュレーションモデルと観測データとの差の演算数が増加するため，結果として Backward 計算にかかる実行時間が増大することが原因だと考えられる．

4.5　間接参照の除去

アジョイント法の Forward および Backward 計算に含まれるステンシル計算では，計算領域の上下左右がつながっている．つまり，左端にある格子点の左隣に右端の格子点が存在する．そのため，ある格子点の上下左右の格子点に参照するため，Naïve な実装では間接参照を用いている．たとえば，ある格子点配列 $Array(i,j)$ の左にある要素を参照したい場合，単純に $left(Array(i,j))$ と書くことによって，間接的に参照を行うことができる．このように間接参照を用いることによってコードを柔軟かつ直感的に書くことができるが，メモリアクセスが増大するため，性能を低下させる可能性がある．過去の研究 [8] で，実際に対象としているステンシル計算で用いられている間接参照が性能を低下させる一因となっていたため，除去することにより

表 1　Fujitsu PRIMEHPC FX100 のシステム構成の概要
Table 1　Overview of the system for the architecture of Fujitsu PRIMEHPC FX100.

CPU					Memory		Architecture in a node	
プロセッサ	クロック周波数	コア数	L1 キャッシュ	L2 キャッシュ	容量（1 ノードあたり）	理論バンド幅（1 ノードあたり）	タイプ	理論ピーク性能
SPARC64 XIfx	2.2 GHz	32（+2）	64 KB（命令・データごと）	24 MB（共有）	32 GB	240 GB/sec（入出力ごと）	NUMA	1.1264 TFlops（倍精度）

最適化を行った．

4.6　ファーストタッチ

NUMA 型の CPU では，変数や配列が宣言された時点では物理メモリに領域を割り当てられず，変数や配列に参照したときに初めて領域が割り当てられる．そのため，変数や配列の初回の参照時間は，2 回目以降の参照時間と比べて非常に時間がかかる．この性能低下を避けるため，ファーストタッチと呼ばれる最適化を実装する．ファーストタッチは，変数や配列が宣言されたときに，それらを演算に用いる前に 1 度参照することによって，NUMA の各 CPU に最も近いメモリに対して，物理的に配列領域を割り当てる最適化手法である．NUMA 型アーキテクチャにおいてファーストタッチを実装しない場合は，物理メモリへのアクセス遅延のため性能が劇的に低下する可能性があるため，この最適化を適用した [9]．

4.7　Loop Collapse

OpenMP を用いてスレッド並列化を適用する際に，Loop Collapse 節を使用することができる．Loop Collapse はネストされた複数のループを 1 つのループにまとめ，ループ長を長くする．ループ長が長くなることによってループのオーバヘッドを減らし，より多くの並列数を利用することができるため，この最適化を適用することで性能向上が期待できる．ただし，対象となる複数のループ間において依存がある場合には Loop Collapse を適用することができない．Forward と Backward 計算に対して時空間ブロッキング適用した実装では，一時配列への値の代入や各スレッドの計算領域を表すイテレーション $xhead$，$xtail$，$yhead$，$ytail$ を時間ステップごとにそれぞれ設定する必要がある．さらに，実験では x 軸方向の空間ブロッキングサイズを 1 としているため，Forward 計算と Backward 計算のループに Loop Collapse を適用しても性能向上しない可能性が高い．実際に，該当ループに対して Loop Collapse 適用による効果の予備実験を行ったところ，ほとんど性能差がなかった．一方で，評価関数のネストされたループ間は依存がないため，Forward や Backward 計算と同様に Loop Collapse の適用を検討した．過去の研究 [9] では，メッシュ数 1600×1600 の二次元空間における評価関数のネストされたループについて Loop Collapse 適用後の性能差を調査したところ，適用前の速度を 1 とした時に速度向上率が約 1.23 倍になることが分かった．

5.　性能評価

5.1　計算機

実験では，名古屋大学情報基盤センターに設置されている Fujitsu PRIMEHPC FX100（FX100）[21] を使用した．計算機の仕様を**表 1** に示す．FX100 のプロセッサである SPARC64 XIfx は，2 つのコアメモリグループ（CMG）で構成されている．1 つの CMG あたり 16 個の計算コアと 1 個のアシスタントコア，17 コア間で共有される 12 MB の L2 キャッシュ，16 GB のローカルメモリで構成され，2 つの CMG 間においては，キャッシュの一貫性が保たれる．メモリ割当てポリシには，2 つのメモリに対してデータを交互に割り当てる interleave_local や，自分が割り当てられている CMG 内のメモリに対してデータを優先的に割り当てる localalloc といったポリシがある．本評価においては，メモリ割当てポリシを変更したときの実行時間にほとんど差がなかったので，デフォルトである localalloc を採用した．またコアの割当てポリシとして simplex を採用した．特に，名大の FX100 システムでは 1 つのジョブによって単一のノードが占有的に使用されるため，ノード内にある 2 つのグループうち，片側のグループに属するコアを優先的に利用する [22], [23]．

5.2　問題設定

提案手法の有効性を評価するため，二次元格子点の初期状態として四角形の相を設定し，相の界面発展速度を特徴づけるモデルパラメータ m を推定する双子実験を行った．双子実験は，本来推定したい解 $\boldsymbol{\Theta}*$ を所与のものとして順問題を解き，その順問題の出力から解 $\boldsymbol{\Theta}$ が真の解 $\boldsymbol{\Theta}*$ をうまく推定できているかどうかを調べる実験である [1]．本評価では真値として 1.00，初期推定値として −1.00，0.00，0.09 を採用した．また問題空間の大きさを 1600×1600，各計算における総時間ステップ数を 128 とそれぞれ設定した．また Forward ブロッキングでは，ブロッキングサイズ 1 あたり $1600 \times 1600 \times 128$ の格子点値を保持するためのメモリ領域が必要となる．1 ノードあたりの使用可能なメモリ量 32 GB をできる限り利用するような設定値として，問題設定としてこれらの値を採用した．さらに Backward

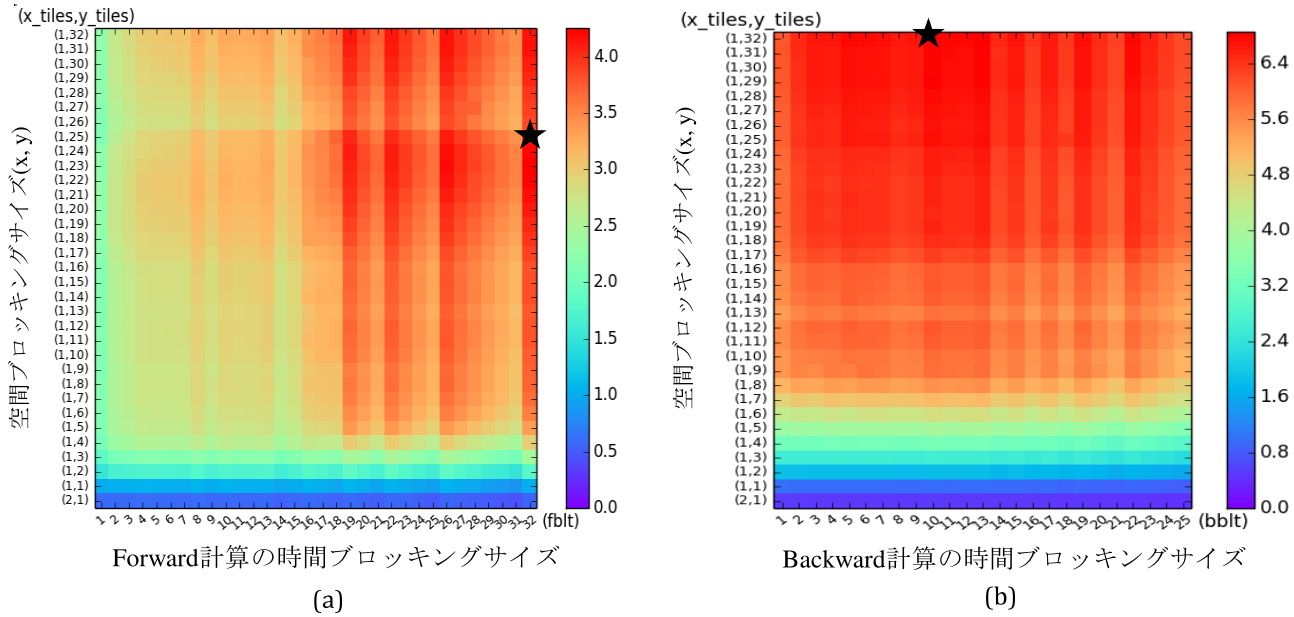

図 5　(a) Forward 計算，(b) Backward 計算における空間および時間ブロッキングサイズを変化させたときの速度向上率．黒星が最速の実行形態となるブロッキングサイズである．x 方向のブロッキングサイズを大きくしたときに性能が劣化する例として (2, 1) の場合を掲載した

Fig. 5　(a) (b) Speed improvement when space and temporal blocking size were changed in forward and backward computations, respectively. Black stars represent the best blocking size. As an example where the performance degrades, the case of (2, 1) is shown.

計算では，ある時間ステップにおいて実測データが存在するときにのみ，シミュレーションモデルと実測データとの差を計算するため，実測データ数が性能に影響を及ぼす．本実験の Backward 計算の性能評価では，実測データ数を 127 と設定した．実験では倍精度のみを使用した．

空間ブロッキングサイズとして，x 方向は 1 および 2，y 方向は 1 から 32 までを調査した．スレッド数は問題空間を割り切ることが可能な 1，4，16，25，32 をそれぞれ採用した．Forward 計算および Backward 計算の時間ブロッキングサイズはそれぞれ 1 から 50 まで網羅的に採用し，Forward ブロッキングは最大サイズとして 3 を採用した．ただし，スレッド数が 25 および 32 の場合においては，設定可能な時間ブロッキングサイズである 32 と 25 までをそれぞれ設定した．これは，Forward と Backward 計算のピラミッド型に計算する処理において各スレッドの計算領域が不足しており，正しく計算が実行されないためである．

Forward ブロッキングサイズは最大サイズを 3 として，動的に変化させた．ブロッキングサイズを blt_size，時間ステップを t としたとき，Forward ブロッキングサイズを oblt(t) = blt_size と表す．このとき，たとえば oblt(1) = 2 は時間ステップが 1 のときにおける Forward ブロッキングサイズが 2 であることを意味し，2 回だけ Forward 計算が実行される．実際に必要な Forward 計算回数よりも大きなブロッキングサイズを設定してしまうと性能が劣化する可能性があるため，この Forward ブロッキングサイズは性能に大きく影響を与える．我々の過去の研究 [9] から，今回設定した問題空間において最も性能向上する Forward ブロッキングサイズが oblt(1) = 1，oblt(i) = 2（$i = 2, 3, 4$），oblt(j) = 3（$j > 4$）であることが経験的に分かっているため，本実験ではこのブロッキングサイズを採用した．

本評価では 2 つの実験を行った．まず Backward 計算への時空間ブロッキング適用前後の性能評価を行い，次に Forward 計算と Backward 計算への時空間ブロッキングと Forward ブロッキングの 3 つすべてを適用した場合におけるアプリケーション全体としての性能評価を行った．2 つ目の実験では，選択可能なブロックサイズ数が膨大であるため，Forward 計算と Backward 計算で最も性能が向上した時間ブロックサイズをそれぞれ採用した．

5.3　評価結果と考察

図 5 (a)，(b) はそれぞれ Forward 計算および Backward 計算において，空間ブロッキングサイズおよび時間ブロッキングサイズを変化させたときの速度向上率のヒートマップである．横軸が時間ブロッキングサイズ，縦軸が空間ブロッキングサイズで (x, y) と表し，これは時空間ブロッキング適用前の実行時間を 1 とした評価である．両方の計算において，x 方向の空間ブロッキングサイズを大きくするにつれ性能劣化することが分かった．我々は，x 方向の空

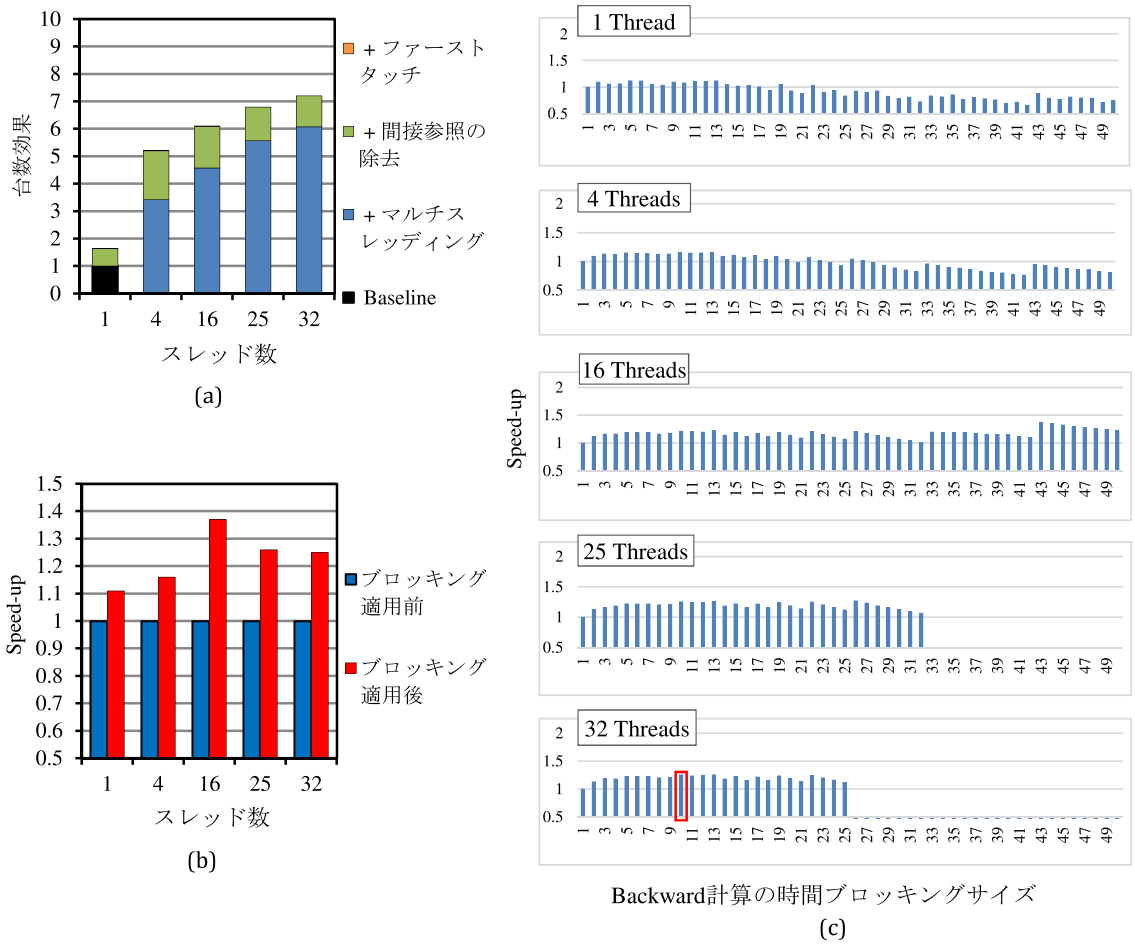

図 6　Backward 計算における最適化実装の性能評価.
(a) 台数効果，(b) 最速となる時空間ブロッキングの適用前後における速度向上率，(c) 時空間ブロッキング適用後の速度向上率.
赤枠で囲まれた箇所が全実行形態の中で最速となった

Fig. 6　Performance evaluations of multi-level blocking.
(a) Performance improvement by applying multi-threading, STB and other optimizations. (b) Speed improvement by applying STB. The area demarcated by the red line shows the best performance. (c) Speed improvement by applying STB.
The area demarcated by the red line shows the best performance.

間ブロッキングサイズを 1 で固定して y 方向の空間ブロッキングサイズを変化させた場合と，$(2, 1)$ と設定した場合について調査した．図 5 (a) より，Forward 計算における最適な空間ブロッキングサイズは $(1, 25)$，時間ブロッキングサイズが 32 であることが分かった．同様に，図 5 (b) より Backward 計算における最適な空間ブロッキングサイズは $(1, 32)$，時間ブロッキングサイズが 10 であることが分かった．よって以降の実験では，空間ブロッキングサイズとして x 方向は 1，y 方向はスレッド数に設定した．

図 6 (a) は Backward 計算において，マルチスレッディングやファーストタッチ，間接参照の除去等の最適化手法を適用した後の最速の実行形態における台数効果を表しており，横軸がスレッド数，縦軸が台数効果である．マルチスレッディングの適用によって，32 スレッドのときに最大となる 7.20 倍の速度向上を得られた．間接参照の除去によっても性能向上したが，一方でファーストタッチ適用による性能向上はほとんど得られなかった．本実装では，プログラム開始時に Forward 計算と Backward 計算の格子点値の更新に用いる一時配列 *temp* を静的に確保している．Backward 計算以前に Forward 計算の中で配列 *temp* を用いた演算を行うため，Backward 計算の実行時には，すでにメモリに対して物理的に領域が割り当てられている．よって，ファーストタッチ適用による影響は Forward 計算には現れたが，Backward 計算には現れないという結果になった．

図 6 (b) は図 6 (c) において最も性能向上した時空間ブロッキングサイズをそれぞれ採用したときの速度向上率である．1，4，16，25，32 スレッドにおいて，ブロッキン

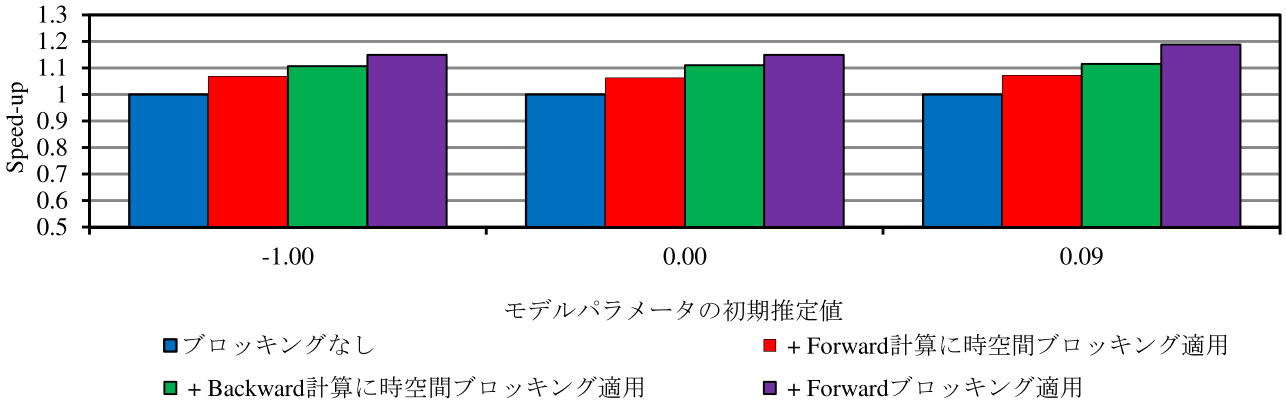

図 7　全ブロッキング適用後の速度向上率

Fig. 7　Speed-up after applying all of the blocking techniques proposed.

表 2　Forward 計算と Backward 計算の性能プロファイル結果

Table 2　The results of performance profile in forward computation and backward computation.

	スレッド数	時間ブロッキングサイズ	L1D ミス数	L1D ミス率（/ロード・ストア数）（%）	L2 ミス数	L2 ミス率（/ロード・ストア数）（%）	実行時間（秒）	浮動小数点演算ピーク比（%）	MFLOPS
Forward	25	1	6.54E+07	2.21	6.19E+07	2.09	0.16	3.08	27061
計算	25	32	4.54E+07	1.83	3.96E+07	1.59	0.08	5.83	51303
Backward	32	1	1.06E+08	2.45	9.43E+07	2.19	0.24	2.62	29495
計算	32	10	9.09E+07	2.43	7.74E+07	2.07	0.19	3.21	36155

グサイズとして 6，10，43，13，10 をそれぞれ採用した．Forward 計算への時空間ブロッキング適用による性能向上（文献 [9] の図 (c)）と比較すると，最速の場合における Backward 計算の性能向上率は低く，32 スレッドかつブロッキングサイズが 10 の場合において 1.25 倍の向上となった．Forward 計算よりも性能向上率が低い理由として，Backward 計算にはシミュレーションモデルと実測データの差を計算する処理が含まれていることと観測データの有無を存在する条件分岐を取り除くことができないことが考えられる．

図 6 (c) は Backward 計算において時空間ブロッキング適用し，時間ブロックサイズを網羅的に設定したときの速度向上率を表している．各スレッドの時間ブロッキングサイズ 1 のときの実行速度をそれぞれ 1 とした結果である．Forward 計算と異なる傾向として，スレッド数が 1 と 4 の場合では時間ブロッキングサイズが小さいほど性能が向上することが分かった．大きいブロッキングサイズを設定することによって，不要な Backward 計算と条件分岐が発生してしまう．そのため，小さいブロックサイズを設定した場合における性能向上が相対的に高くなっていると考えられる．

図 7 は全ブロッキング適用前の 25 スレッドを使用した実行時間を 1 としたときにおける，各ブロッキング適用後の速度向上率を表している．真値である 1.00 と差が大きい −1.00，差が小さい 0.09，中間である 0.00 の場合において実験を行った．Forward 計算と Backward 計算における最速の実行形態が異なるため，各スレッド数に応じた最速となるブロッキングサイズをそれぞれ設定した．32 スレッド利用して Forward 計算を実行するときの最適なブロッキングサイズは 22 である．一方 Backward 計算では，25 スレッド利用して実行するときの最適なブロッキングサイズが 26 であるため，これら異なるブロッキングサイズを採用した．実験結果より，スレッド数が 25，Forward 計算の時間ブロッキングサイズが 32，Backward 計算の時間ブロッキングサイズが 26 かつ Forward ブロッキング適用時が最速となり，初期推定値が 0.09 のときに最大で 1.18 倍の性能向上を達成した．

Forward 計算と Backward 計算の時空間ブロッキング適用前後における性能差の原因を調査するため，Fujitsu の詳細プロファイラ（精密 PA 可視化 [24]）を利用した．時空間ブロッキング適用前後の最速の実行形態における性能プロファイル結果を**表 2** に示す．Forward 計算では，時空間ブロッキング適用前と比較して，Level 1 Data（L1D）キャッシュミス数が 30.59%，Level 2（L2）キャッシュミス数が 36.03%削減された．これに加えて，L1D キャッシュミスヒット率が 2.21%から 1.83%に低下し，L2 キャッシュミスヒット率が 2.09%から 1.59%に低下していることが分かった．Backward 計算では，L1D と L2 キャッシュミス

数がそれぞれ 14.25%，17.92%削減され，L1D と L2 キャッシュミスヒット率がそれぞれ 2.45%から 2.43%，2.19%から 2.07%に若干低下した．Forward 計算と Backward 計算におけるキャッシュミスヒット率の低下率が低い理由の 1 つとして，ベンチマークで使用している問題サイズが小さすぎるために Naïve な実装における Forward 計算および Backward 計算のキャッシュヒット率が高いことが考えられる．キャッシュミス数が削減されることによってキャッシュメモリ上にあるデータの再利用性が高まり，Forward 計算における浮動小数点演算ピーク比は 3.08%から 5.83%に 2.75 ポイント，Backward 計算における浮動小数点演算ピーク比は 2.62%から 3.21%に 0.59 ポイント増加した．

6. おわりに

本研究では，非逐次型データ同化手法の 1 つであるアジョイント法に対して高性能化を行った．アジョイント法はメモリインテンシブな処理であるため，十分な性能が得られていない．これに加えて，これまでアジョイント法における計算ブロッキング手法について十分な検討がなされているとはいえなかった．アジョイント法アプリケーションを高性能化するため，アジョイント法の処理の中でもボトルネックとなっている Forward 計算と Backward 計算に対して時空間ブロッキングを適用し，さらに複数の Forward 計算を投機的に実行するブロッキングを適用した．本研究では，これに加えてスレッド並列化やファーストタッチ，間接参照の除去等の最適化手法も適用した．

各ブロッキング手法の有効性を調査するため，Fujitsu PRIMEHPC FX100 を用いて実験を行った．Backward 計算では，時空間ブロッキング適用により最大 1.25 倍の性能向上を達成し，さらにすべてのブロッキング手法をアジョイント法に適用することによって，アプリケーションとして最大 1.18 倍の速度向上を得られることが分かった．

キャッシュミス数とキャッシュミスヒット率の分析結果によると，Forward 計算では L1D と L2 キャッシュミス数がそれぞれ 30.59%，36.03%削減され，キャッシュミスヒット率が低下した．Backward 計算においても L1D と L2 キャッシュミス数が%削減され，Forward 計算同様のミスヒット率が低下する結果となった．キャッシュメモリ上にあるデータの再利用性が高まったことによりメモリアクセスが減少し，Forward 計算および Backward 計算における浮動小数点演算ピーク比はそれぞれ 2.75 ポイント，0.59 ポイント増加した．

本研究では，スレッド並列化を採用しているため，対象としている計算機では最大で 32 並列までしか利用することができない．Forward 計算において，マルチスレッディングによる最も高い性能向上を得るためには 16 スレッドが必要であることが分かった．複数の Forward 計算を同時に実行するためには，並列数が不十分であるため，Forward 計算における最高性能を達成できない．さらに，実際に解きたい問題のサイズは現状で設定可能な問題よりも大規模なものである．さらなる並列数を利用して超大規模な問題を解くためには MPI を用いたプロセス並列化，さらにはスレッド並列化と組み合わせたハイブリッド MPI を適用する必要がある．これにより複数の Forward 計算を同時に実行することが可能になるだけでなく，ノード内のソケットをまたぐことによる性能劣化も防ぐことができる．また本手法には，実行時のパラメータとして空間分割数と各ブロッキングサイズがある．

以上のハイブリッド MPI 化と最適なパラメータ選択のための自動チューニング技術 [25], [26], [27], [28] の適用は今後の課題である．

謝辞　本研究の一部は，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），および，・ポスト「京」萌芽的課題アプリケーション開発，萌芽的課題 1 基礎科学のフロンティアー極限への挑戦，「極限の探究に資する精度保証付き数値計算学の展開と超高性能計算環境の創成」による．

参考文献

[1] 樋口知之（編著），上野玄太，中野慎也，中村和幸，吉田亮（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] 淡路敏之，蒲地政文，池田元美，石川洋一：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).
[3] Lewis, J.M. and Derber J.C.: The use of adjoint equations to solve a variational adjustment problem with advective constraints, *Tellus A 37A*, pp.309–322 (1985).
[4] Le Dimet, F.-X. and Talagrand, O.: Variational algorithms for analysis and assimilation of meteorological observations: theoretical aspects, *Tellus A 38A*, pp.97–110 (1986).
[5] 伊理正夫，久保田光一：高速自動微分法，日本応用数理学会，Vol.1, No.1, pp.17–35 (1991).
[6] 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).
[7] Tsukada, Y, Murata, Y, Koyama, T and Morinaga M.: *Phase-Field Simulation on the Formation and Collapse Processes of the Rafted Structure in Ni-Based Superalloys*, Vol.49, No.3, pp.484–488 (2008).
[8] 池田朋哉，伊藤伸一，長尾大道ほか：アジョイント法における Forward model への階層ブロッキング適用による高性能化，情報処理学会研究報，Vol.2016-HPC-157, No.17 (2016).
[9] Ikeda, T., Ito, S., Nagao, H., et al.: Optimizing Forward Computation in Adjoint Method via Multi-level Blocking, *Submitted to ACM HPC Asia2018* (2017).
[10] Kobayashi, R.: Modeling and numerical simulations of dendritic crystal growth, *Physica D*, 63, pp.410–423 (1993).
[11] Kalnay, E.: *Atmospheric Modeling, Data Assimilation and Predictability*, Cambridge University Press (2003).

[12] Tsuyuki, T. and Miyoshi, T.: Recent progress of data assimilation methods in meteorology, *J. Meteorol. Soc. Jpn. Ser. II*, 85B, pp.331–361 (2007).
[13] Ito, S., Nagao, H., Yamanaka, A., et al.: Data assimilation for massive autonomous systems based on a second-order adjoint method, *Physical Review E*, 94, 043307 (2016).
[14] Datta, K., Murphy, M., Volkov, V., et al.: *Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures*, IEEE press (2008).
[15] Meng, J. and Skadron, K.: Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs, *Proc. 23rd international conference on Supercomputing*, ACM (2009).
[16] Li, Z. and Song, Y.: Automatic tiling of iterative stencil loops, *Journal ACM Trans. Programming Languages and Systems*, Vol.26, Issue 6, 2004, pp.975–1028 (2004).
[17] Orozco, D., Garcia, E. and Gao, G.: Locality optimization of stencil applications using data dependency graphs, *Languages and Compilers for Parallel Computing*, pp.77–91, Springer Berlin Heidelberg (2011).
[18] Zhou, X.: Tiling optimizations for stencil computations, Ph.D. thesis, University of Illinois at Urbana-Champaign (2013).
[19] Bandishti, V., Pananilath, I. and Bondhugula, U.: Tiling stencil computations to maximize parallelism, *Proc. International Conference for High Performance Computing, Networking, Storage and Analysis*, pp.1–11 (2012).
[20] Armijo, L.: Minimization of functions having Lipschitz continuous first partial derivatives, *Pacific J. Math.*, Vol.16, No.1 (1966).
[21] 千葉修一：PRIMEHPC FX100 の特徴と概要，Fujitsu Ltd 入手先 ⟨http://accc.riken.jp/wp-content/uploads/2015/06/chiba.pdf⟩ (2015).
[22] Fujitsu Ltd.: *Fortran User's Guide* (*PRIMEHPC FX100*), Vol.J2UL-1865-03Z0(01) (2017).
[23] Fujitsu Ltd.: *MPI User's Guide* (*PRIMEHPC FX100*), Vol.J2UL-1885-03Z0(01) (2017).
[24] Fujitsu Ltd.: *Profiler User's Guide* (*PRIMEHPC FX100*), Vol.J2UL-1891-02ENZ0(00) (2015).
[25] Katagiri, T., Kise, K., Honda, H. and Yuba, T.: ABCLibScript: a directive to support specification of an auto-tuning facility for numerical software, *Parallel Computing*, Vol.32, Issue 1, pp.92–112 (2006).
[26] Katagiri, T., Ohshima, S. and Matsumoto, M.: Directive-based auto-tuning for the finite difference method on the Xeon Phi, *Proc. IPDPSW2015*, pp.1221–1230 (2015).
[27] Katagiri, T., Matsumoto, M. and Ohshima, S.: Auto-tuning of Hybrid MPI/OpenMP Execution with Code Selection by ppOpen-AT, *Proc. IPDPSW2016*, pp.1488–1495 (2016).
[28] Katagiri, T., Ohshima, S. and Matsumoto, M.: Auto-tuning on NUMA and Many-core Environments with an FDM Code, *Proc. IPDPSW2017* (2017).

池田 朋哉　（学生会員）

名古屋大学大学院情報科学研究科修士課程．2012 年 3 月福井県立高志高等学校卒業，2016 年 3 月名古屋工業大学工学部情報工学科卒業，2016 年 4 月より現職．データ同化アルゴリズムの高性能実装に関する研究に従事．情報処理学会の学生会員．

伊藤 伸一

東京大学地震研究所特任研究員．2005 年 3 月福井県立高志高等学校卒業，2010 年 3 月大阪大学理学部物理学科卒業，2015 年 3 月大阪大学大学院理学研究科宇宙地球科学専攻博士課程修了．博士（理学）．2015 年 4 月より現職．構造材料性能予測のためのデータ同化法の開発に関する研究に従事．日本物理学会，日本統計学会等，各会員．

長尾 大道

東京大学地震研究所准教授．1991 年 3 月金蘭千里高等学校卒業，1995 年 3 月京都大学理学部卒業，2002 年 3 月京都大学大学院理学研究科地球惑星科学専攻博士課程単位取得退学．博士（理学）．2002 年 4 月特殊法人核燃料サイクル開発機構東濃地科学センター客員研究員，2006 年 3 月独立行政法人海洋研究開発機構研究員，2009 年 6 月大学共同利用機関法人情報・システム研究機構統計数理研究所特任研究員，2010 年 12 月同特任准教授，2013 年 9 月より現職．ベイズ統計学，特にデータ同化アルゴリズム開発とその応用研究に従事．日本統計学会，日本地球惑星科学連合，American Geophysical Union，American Physical Society 等，各会員．

片桐 孝洋　（正会員）

名古屋大学情報基盤センター教授．1994 年豊田工業高等専門学校情報工学科卒業，1996 年京都大学工学部情報工学科卒業，2001 年東京大学大学院理学系研究科情報科学専攻博士課程修了．博士（理学）．2001 年 12 月科学技術振興機構研究者，2002 年 6 月電気通信大学大学院情報システム学研究科助手，2005 年 3 月～2006 年 1 月米国カリフォルニア大学バークレー校コンピュータサイエンス学科訪問学者，2007 年 4 月東京大学情報基盤センター特任准教授，准教授を経て，2016 年 4 月より現職．ソフトウエア自動チューニング，高性能計算，コ・デザインの研究に従事．2002 年情報処理学会山下記念研究賞受賞．2007 年 Microsoft INNOVATION AWARD 2007 アカデミック部門最優秀賞受賞．2011 年文部科学大臣表彰若手科学者賞受賞．日本応用数理学会，ACM，IEEE-CS，SIAM 等，各会員．

永井 亨　（正会員）

名古屋大学情報基盤センター助教．1978 年 3 月愛知県立一宮西高等学校卒業，1982 年 3 月名古屋大学大学理学部卒業，1986 年 6 月名古屋大学大学院理学研究科地球科学専攻博士課程中途退学，1986 年 7 月名古屋大学大型計算機センター助手，1990 年 3 月理学博士（名古屋大学），2002 年 4 月名古屋大学情報連携基盤センター助手，2007 年 4 月名古屋大学情報連携基盤センター助教，2009 年 4 月より現職．高性能計算，弾性波動場の計算アルゴリズム開発の研究に従事．地震学会，日本地球惑星科学連合，American Geophysical Union 等，各会員．

荻野 正雄　（正会員）

名古屋大学情報基盤センター准教授．1999 年九州大学工学部知能機械工学科卒業，2001 年九州大学大学院工学研究科知能機械システム専攻修士課程修了，2004 年九州大学大学院工学府知能機械システム専攻博士課程修了．博士（工学）．2004 年九州大学大学院工学研究院知能機械システム部門学術研究員，2005 年九州大学大学院工学研究院知能機械システム部門助手，助教を経て，2011 年 11 月より現職．有限要素法，並列計算，反復法の研究に従事．2009 年日本計算工学会論文賞，2010 年 APACM Young Investigator，2012 年 JACM Young Investigator 受賞．日本機械学会，日本計算工学会等，各会員．