Vol.2017-HPC-162 No.16
2017/12/19

MPI と通信削減アルゴリズムによるアジョイント法の高性能化

池田 朋哉†，伊藤 伸一††，長尾 大道†3，片桐 孝洋†4，永井 亨†4，
荻野 正雄†4

アジョイント法は非逐次型に分類されるデータ同化手法の一つで，時系列データ全体を評価し，初期状態と同時にモデルパラメータの推定を行う計算技術である．我々は過去の研究で，アジョイント法に対して時空間ブロッキングと複数の計算を同時実行するブロッキングを組み合わせた多階層ブロッキングに加えて，マルチスレッディング，ファーストタッチ，間接参照の除去といった基礎的な最適化技術を適用した．しかしマルチスレッディングのみ利用した並列化では利用可能なメモリ量に制約があるため，より大規模な問題を解くことができない．さらに並列数が不足しているため，並列実行できる処理を逐次的に行っている．そこで我々は MPI を用いてプロセス並列化を行い，加えて既存手法で逐次的に処理を行っていた「複数の Forward 計算を同時実行するブロッキング」の複数の計算を並列実行できるよう改良した．これまでの限界であった二次元空間の問題サイズ 1600×1600 に加えて最大 6400×6400 まで実験を行った．提案手法の通信回避によって，Forward 計算では 1024 プロセス時に通信時間が 173%ほど削減され，既存手法の最速の実行形態に対して Forward 計算では 9.76 倍，アプリケーション全体では 18.81 倍ほど高速に解くことができるようになった．

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を融合する計算技術として「データ同化」が注目されている[1]-[2]．我々の過去の研究[3]-[4]ではデータ同化手法の一つであるアジョイント法に対して，ファーストタッチ等の基礎的な最適化やスレッド並列化，さらに時空間ブロッキングと複数の計算を同時実行するブロッキングから成る多階層ブロッキングを適用することにより，アプリケーションの性能を向上した．しかし既存手法ではスレッド並列化のみを適用しているため，利用可能なメモリ量および並列数が限られる．例えば，我々が過去に実験で使用した FUJITSU PRIMEHPC FX100 では，ノードあたり 32GB と 32 スレッドしか利用できない．より大規模な問題を高速に解くためには，現状ではメモリ量・並列数ともに不足している．この問題に加えて，プロセスレベルの並列化を行う場合には，プロセス間の通信時間がカーネルの演算時間よりも支配的になる可能性がある．そのため，単純にプロセス並列化するだけでは十分な性能が得られない可能性がある．そこで本研究では，MPI を用いてプロセスレベルの並列性を利用し，さらに通信削減アルゴリズムを適用することでアジョイント法を高性能化する．本報告では，我々が既に提案した手法を既存手法とし，この既存手法に対する性能向上について評価する．

本論文の構成は以下のとおりである．2 章では，本研究で対象とするフェーズフィールド法のモデル式について説明する．3 章では，アジョイント法の概要とその中でボトルネックとなっている Forward 計算，Backward 計算，評価関数の計算について説明する．4 章では，FUJITSU PRIMEHPC FX100 を用いた性能評価を行い，考察する．最後に，得られた知見についてまとめを述べる．

2. テストモデル：フェーズフィールド法

本研究では，液相中の凝固過程や金属内の相変態過程などの相転移現象の時間発展を計算する数値シミュレーションモデルである，フェーズフィールドモデル[5]-[6]に注目する．フェーズフィールドモデルは，エネルギー原理に基づいて対象とする現象を柔軟に表現できるため，相転移現象だけでなく，密度差の大きい混相流の計算[7]や亀裂の進展問題[8]など，分野をまたいで幅広く利用されている．

フェーズフィールドモデルは注目する対象によって様々であるが，本研究では中でも最も基本的なモデルである，小林のフェーズフィールドモデル[9]をテストモデルとして採用する．小林のフェーズフィールドモデルは 2 つの異なる相の競合過程をモデル化したもので，相の存在確率を場の変数で表現することで，2 相の間の界面の時間発展を記述する．場所 x 時間 t での 1 つの相の存在確率を $\phi(x,t)$ とすると，$\phi(x,t)$の時間発展方程式は以下の式で表される．

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

$$|m| < \frac{1}{2} \qquad \ldots(1)$$

ここで，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．

3. データ同化

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

† 名古屋大学 大学院情報科学研究科
†† 東京大学 地震研究所
†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
†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

能にする．データ同化の目的の一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毎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．一方で，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一点を探索するため，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．本研究では，非逐次型データ同化に分類されるアジョイント法を対象とする．

3.1 アジョイント法

アジョイント法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元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．我々は過去の研究で，多階層のブロッキングや OpenMP を用いてアジョイント法の Forward 計算と Backward 計算を高性能化した[3]-[4].

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，両者の関係は式(2)のように表される．

$$\phi_i^{obs}(t) = \phi_i(t) + \omega_i(t) \quad (i = 1,\ \dots,\ M) \qquad \dots(2)$$

$\omega_i(t)$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．本研究では，ボトルネックとなっている Forawrd 計算と Backward 計算，評価関数の計算に着目する．離散化された Forward 計算および Backward 計算をそれぞれ式(3)(4)に示す．

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

$$0 < \Theta_i < 1 (i = 1,\ \dots,\ M+1) \qquad \dots(3)$$

$$-\tau\frac{\partial\lambda_i}{\partial t} = \begin{cases} \epsilon^2\Delta_i\lambda_i + \{-3\theta_i^2 + (4 - 2\theta_{M+1})\theta_i + \theta_{M+1} - 1\}\lambda_i + \dfrac{\partial\mathcal{J}}{\partial\theta_i} & \text{for } i = 1,\dots,M \\ \displaystyle\sum_{j=1}^{M}\theta_j(1-\theta_j)\lambda_j & \textit{otherwise,} \end{cases}$$

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

$$\boldsymbol{\lambda}(t_f) = 0 \qquad \dots(4)$$

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(3)(4)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，これらの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．

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$t = t_f$として，設定した評価関数を式(5)に示す．

$$J = \int_0^{t_f} dt\, \mathcal{J},$$

$$\mathcal{J} = \frac{1}{2}\sum_{t_s\in\mathcal{T}}\delta(t - t_s)\sum_{i=1}^{M}\left(\phi_i^{obs}(t_s) - \phi_i(t_s)\right)^2$$

$$(s = 1,\ \dots,\ n) \qquad \dots(5)$$

式導出の詳細については文献[10]に委ねる．

3.2 既存手法の問題点と本研究の目的

既存手法ではプロセスレベルでの並列性を利用していないため，利用可能なメモリ量に限界がある．例えば我々が過去の研究で使用した FUJITSU PRIMEHPC FX100 の場合，プロセス並列化なしでは 1 ノードあたり 32GB までしか利用することができない．この問題に加えて，利用可能な並列数が限られているという問題がある．FX100 システムでは 1 ノードあたり 32 スレッドまで利用できるが，それ以上の並列数を稼ぐことが出来ない．現状では並列数が不足しているため，複数の Forward 計算を同時に実行するブロッキング（以降，Forward ブロッキングと呼ぶ）を適用する際に，並列に実行できる計算を逐次的に処理していた．そこで本研究では，単一ノードのメモリ量で解けないような大規模な問題を解けるようにすることと，アジョイント法の更なる高性能化を目的とする．

3.3 関連研究

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[10].

また，時空間ブロッキングを用いたステンシル計算の高性能化は，既に多くの方々によって研究が進められている[11]-[18]. 通信削減アルゴリズムでは，Demmel ら[19]が QR 分解の通信削減アルゴリズム TSQR を提案し，その有効性を示している．また熊谷ら[20]は，共役勾配法に現れる集団通信の回数を減らした CBCGR 法に対し，通信削減手法である Matrix Powers Kernel(MPK)を適用して高性能化している．MPK はステンシル計算では時空間タイリングを用いて実現できる．ステンシル計算の時空間タイリング方法とし

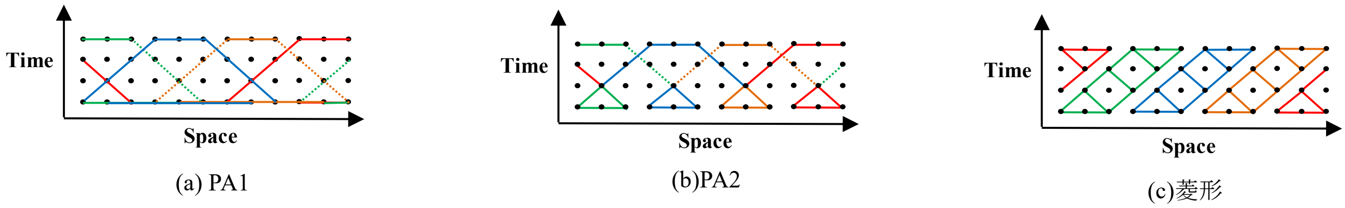


図 1　通信削減アルゴリズム適用時における種々のステンシル計算方法

ては，Hoemmen[21]の論文にある PA1(台形)や PA2(三角形+台形)の計算手法がよく知られている．また，須田[22]は重複計算のない計算手法を提案しており，演算・通信コストのバランスが最も良い三角形と台形型を合わせた手法が最も良い性能を得られる可能性があると報告されている．

4. アルゴリズムの適用と手法の改良

4.1 用語

本報告では，以下のように用語を定義する．問題空間を(nx, ny)と表し，それぞれ nx, ny が x, y 軸方向の総格子点数である．問題空間の分割数はそれぞれ x_tile, y_tile と表す．例えば(nx, ny) = (1600, 1600)の時に x_tile=2, y_tile=4 と設定した場合には，各プロセスの計算を担当する領域は 800x400 となる．また，Forward 計算(Backward 計算)あたりの総時間ステップ数を nda とする．Forward 計算の時間ブロッキングサイズは fblt，Backward 計算の時間ブロッキングサイズは bblt, Forward ブロッキングのサイズは oblt と表す．スレッド並列化の場合と異なり oblt は静的に設定するため，実行時に変化しないものとする．

4.2 通信削減アルゴリズム

既存手法のスレッド並列化された実装では，メモリ空間を全スレッドで共有しているため，冗長な計算は不要であった．そのため，キャッシュを有効利用した計算方法としては既存手法のような実装がある．プロセス並列化する際のステンシル計算の実装では，Hoemmen の提案する PA1 や PA2，須田の提案する菱形といった計算方法が考えられる．各計算方法のイメージを図 1 に示す．簡単のため，図 1 は空間 1 次元の三点ステンシル計算としている．

PA1 では，最初に隣接したプロセスと通信を行い，時間ブロッキングサイズだけ自分の担当領域を計算するために必要な袖領域の格子点値を得る．その後，一気に時間ブロッキングサイズまで担当領域の格子点値を計算する．このサイクルを繰り返すことによりステンシル計算を実行する．そのため，時間ブロッキングサイズが大きくなるほどプロセス間の通信が削減されるという利点がある．一方で，通信をしない代わりに，隣接したプロセスと自プロセスとでは重複した計算を行ってしまうため，ブロッキングサイズが大きいほど重複する演算は増大してしまう欠点がある．つまり，通信回数と冗長な演算のトレードオフとなっており，高性能化するためには最適なブロッキングサイズを設定する必要がある．

PA2 は，我々が過去に実装したスレッド並列化における計算方法と似ている．PA2 では，最初に三角形(過去の研究ではピラミッド型と呼んでいた形)に時間ブロッキングサイズだけ担当領域の格子点値を計算する．その後，隣接プロセスと通信し，各プロセスで台形の残りの部分を計算する．PA1 と比べ，通信回数を増やすことなく重複した計算を減らすことができる．さらにこの方法では，通信回数を増やす代わりにウェーブフロント法を適用することもできると考えられる．

菱型の計算は，まず初めに時間ブロッキングサイズ先までピラミッド型に計算を行う．各プロセスが計算を終えた後に，算出された格子点値を隣接するプロセスに送信し，自分は袖領域の格子点値を受信する．その後，受信した格子点値を用いて計算可能な残りの格子点値を計算する．この時，他プロセスとは異なる格子点値を計算するため，冗長な計算は存在しない．一方で，時間ブロッキングサイズ先まで計算するまでに通信回数が複数回になる可能性がある．アジョイント法の Forward 計算と Backward 計算に最適な手法を調査するためには全ての方法を試す必要がある．そこで，本研究ではまず PA1 による計算方法を実装した．

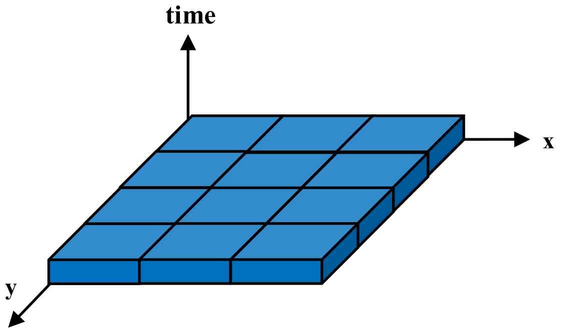
4.2.1 Forward と Backward 計算への通信削減アルゴリズムの適用

Forward 計算における PA1 による時空間ブロッキングを適用したアルゴリズムを Alg. 1. に示す．Forward 計算を実行する前に，予め各プロセスの担当する計算領域および通信相手のプロセスを設定しておく．ステンシル計算を行った後に，上下左右のプロセスと通信を行って格子点値を交換する．プロセス通信の実装方法はいくつか考えられる．例えば，同期通信 MPI_Send/Recv を用いる方法があるが，実装には条件分岐が必要になる．条件分岐を含むことにより性能が劣化することが懸念されるため，今回は非同期通信の MPI_Isend/Irecv 関数を用いて実装した．Forward 計算と同様，Backward 計算に含まれるステンシル計算についても PA1 による計算方法を適用した．

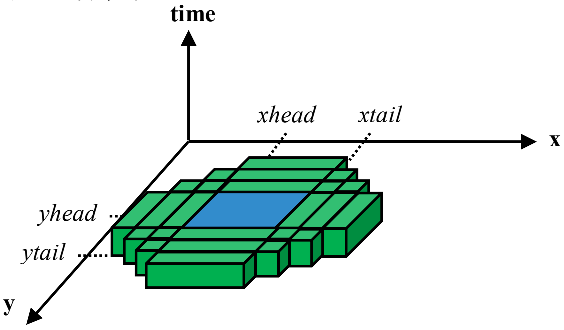
キャッシュの観点では，時間ブロッキングサイズが大きくなるにつれキャッシュミス数が増加し，キャッシュヒット率が低下すると予測できる．これは，ブロッキングサイズ先まで一気に計算する際に，ブロッキング適用前と比べて多くの格子点値が必要になり，結果として計算で必要な

Algorithm 1　Forward 計算における PA1 の適用

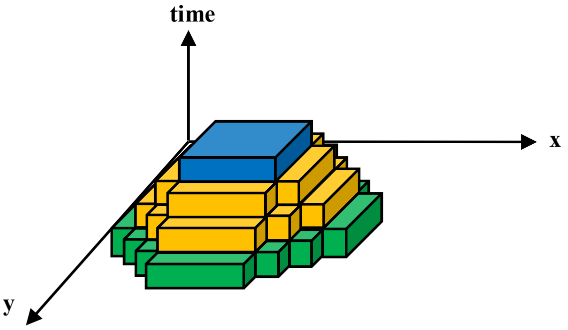
```
格子点値用の一時配列 temp を静的確保
do it = 1, nda
    temp に格子点値をロード
    do t = 1,  fblt
        格子点値の計算 (ステンシルを含む)
    end do
    上下左右の隣接プロセスと格子点値を送受信
    temp から格子点値をストア
end do
```



(a)空間分割数 x_tile=3,　y_tile=4 設定時の各プロセスの担当する計算領域



(b)あるプロセスの 1 段目



(c)1 段目から fblt 段目までのステンシル計算

図 2 二次元空間におけるステンシル計算(PA1)のイメージ．それぞれ青色がプロセスの担当する計算領域，緑色が袖領域，橙色が隣接プロセスと重複する計算領域を表す．

データがキャッシュメモリから追い出されてしまうためである．そこで本評価では，Forward 計算の性能評価だけでなく，時間ブロッキング適用前後の通信時間、カーネル部分の演算時間およびのキャッシュヒット率の変化についても調査する．

4.3 Forward ブロッキング適用による並列計算

過去の研究の予備実験で，イテレーション毎の Forward 計算と評価関数の計算の回数が異なることが分かっている．加えて，これらの計算回数は事前に予測できない問題があった．そこで我々は複数の Forward 計算と評価関数の計算を同時に実行するブロッキングを適用することで，高性能化を行った．しかし，FX100 で利用可能なスレッド数は 32 までであり，我々の知見では個々の Forward 計算は 25 スレッドを利用した時に最速となる．そのため，複数の Forward 計算を並列に実行させるためには，並列数が不足していた．そこで，プロセス並列化を適用することにより，利用可能な並列数を増やして並列に複数の Forward 計算を実行するように変更する．

またこの変更に伴い，評価関数の計算の処理にも変更を加えた．評価関数の計算にはステンシル計算が含まれない．しかし Forward 計算の結果と実測データとの差を二乗した和を求める必要があるため，各プロセスがそれぞれ担当する領域の評価関数を計算した後に，通信を行って値を集計する必要がある．加えて，　プロセス並列化された上で Forward ブロッキングを行うためには，Backward 計算の前に各プロセスで計算された評価関数の結果を全プロセスで共有する必要がある．これは，同時に実行された評価関数の計算の結果の中で最適な初期値を選択し，その計算結果を元に次のイテレーションの Forward 計算を実行するためである．スレッド並列化の時にはメモリ空間が共有されているため通信が不要であったが，プロセス並列化される場合には，評価関数の計算結果を集計して最適な初期値を選択し，その上で最適な初期値を用いた場合の格子点値を全プロセス間で共有する必要がある．問題空間の分割数を x_tile,　y_tile,　Forward ブロッキングサイズを oblt とした時，評価関数の計算結果を集計するフェーズ A では (x_tile*y_tile - 1) 回の送受信を行う必要があり，また全プロセス間で最適な初期値を用いた時の格子点値を共有するフェーズ B では，最適な初期値と評価関数の値を持っているプロセスによる(oblt - 1)回の送信と，その他のプロセスによる 1 回の受信がなされる必要がある．

Forward ブロッキング適用後における各プロセスの担当する計算領域を図 3 に示す．フェーズ A では，図 3 の PE0 が 1 から 3，　PE4 が 5 から 7 より評価関数の計算結果を受信し，総和を全プロセスに送信する．そのため，x_tile と y_tile の値が大きくなるにつれて通信回数が線形に増加する．　実装では，初期値ごとに担当するプロセスが異なるため，MPI_Gather は使わず MPI_Isend/Irecv を用いて実装し

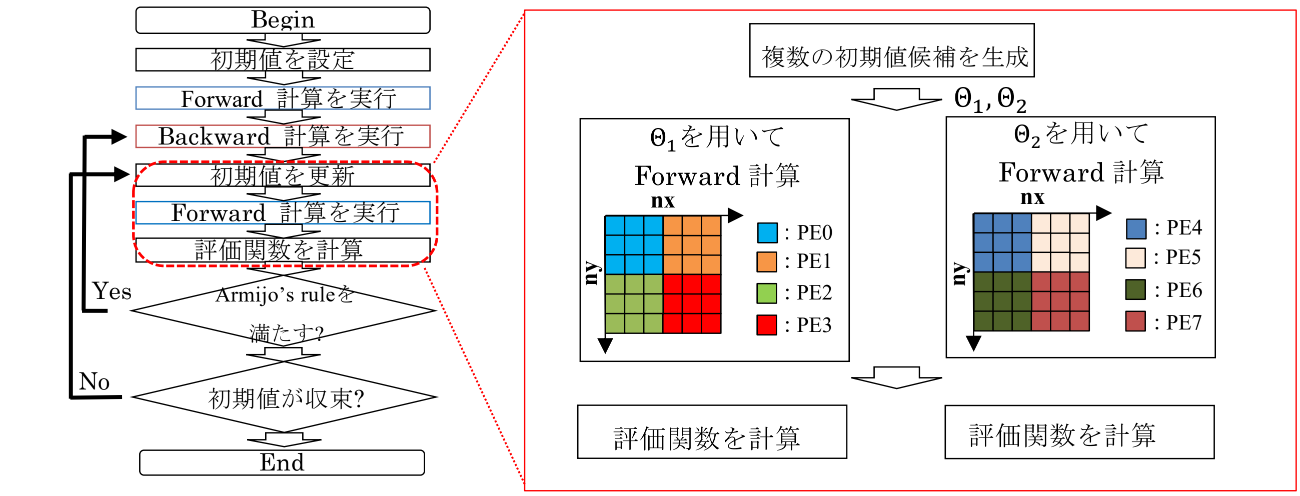


図 3　既存手法のアジョイント法の計算フローと Forward ブロッキング適用後の複数 Forward および評価関数の計算．これは 8 プロセス使用時の例で，空間分割数 x_tile=2，y_tile=2，Forward ブロッキングサイズ oblt=2 としている．この例では，それぞれの Forward 計算と評価関数の計算に 4 プロセスずつ使用して実行する．

た．フェーズ B では，図 3 の例では，2 つの異なる探索空間のパラメータ Θ_1, Θ_2について説明しているが，例えばΘ_2が最適な初期値であった場合，PE4，5，6，7 からそれぞれ PE0，1，2，3 へ格子点値を送信するため，oblt が大きくなるにつれて通信コストが増大し，この影響により性能が劣化する恐れがある．さらに通信コストに加えて，値の送受信のための配列のパッキング・アンパッキング処理といったメモリの Read/Write が発生する．この要因による性能劣化も考えられるが，各プロセスにおけるメモリの Read/Write 回数は oblt の値に依らないため，2 以上の値を設定した場合には oblt の変化による影響は殆どないと推測できる．つまり，2 よりも大きい値を設定する時により性能が向上すると予測される．ただしブロッキングを適用しない場合においては，通信と Read/Write 処理が不要であるので，これらの処理を行わないよう実装した．一方，フェーズ B は Forward 計算の時間ブロッキングサイズ fblt に依存しているため，fblt が大きくなるとともに使用メモリ量とメモリの Read/Write にかかる時間は増大すると考えられる．

5. 性能評価

5.1 問題・実験設定

既存研究と同様，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．本評価では真値として 1.00，初期推定値として-1.00 を採用した．また，問題空間の大きさは従来の(nx，ny)=(1600，1600)を採用し，Weak Scaling の評価では(1600，3200), (3200，3200)，(6400，3200)，(6400，6400)を採用した．Forward 計算と Backward 計算における総時間ステップ数を 128 と設定した．また Backward 計算では，実測データ数が性能に影響を与えるため，この実測データ数を(初期状態を除く)127 と設定し，実験では倍精度で計算した．

空間分割数 x_tile と x_tile はそれぞれ 2 以上かつ設定可能な値を選択した．Forward 計算の時間ブロッキングサイズ fblt は 1 から 32 まで(ただし，計算領域が足りない場合は限界サイズまで)網羅的に設定し，Backward 計算の時間ブロッキングサイズ bblt は fblt≧bblt となるように設定した．Forward ブロッキング oblt は 1，2，4 を設定した．また，Forward ブロッキング適用前後およびアジョイント法全体の評価では，アジョイント法のイテレーション 70 回までの実行時間の平均値を採用した．よって今回採用した初期推定値では，最初の 10 回までは Forward 計算を 5，7 回行う一方で，その他の 60 回は Forward 計算回数が 1 回で十分であることに注意されたい[3].

5.2 計算機環境

以下の計算機を利用した．

1. Fujitsu PRIMEHPC FX100 (FX100)
   - 名古屋大学情報基盤センター設置
   - CPU：SPARC64 XIfx，2.2 GHz 32(+2)コア
   - 記憶容量：32 GB
   - 理論ピーク性能（ノード）: 1.1264 TFLOPS(倍精度)
   - キャッシュ構成
     - L1:64KB（命令／データ分離，コア毎），L2:24MB（共有）
     - 4 ウェイ
   - 1 ソケット当たり 16 コア，ノードあたり 2 ソケットの NUMA 構成
   - 富士通 MPI

Vol.2017-HPC-162 No.16
2017/12/19

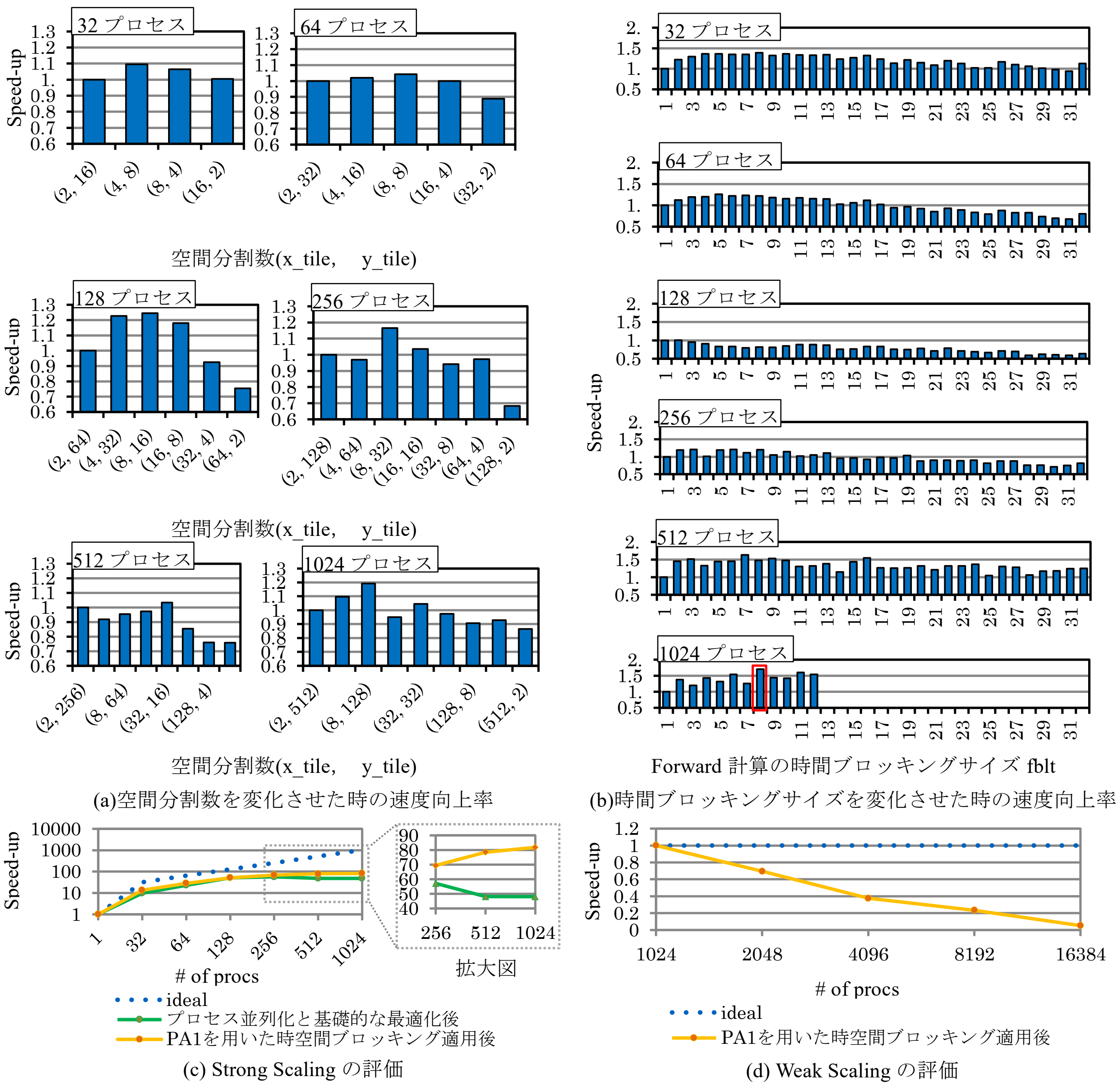


図 4　Forward 計算における最適化実装の性能評価

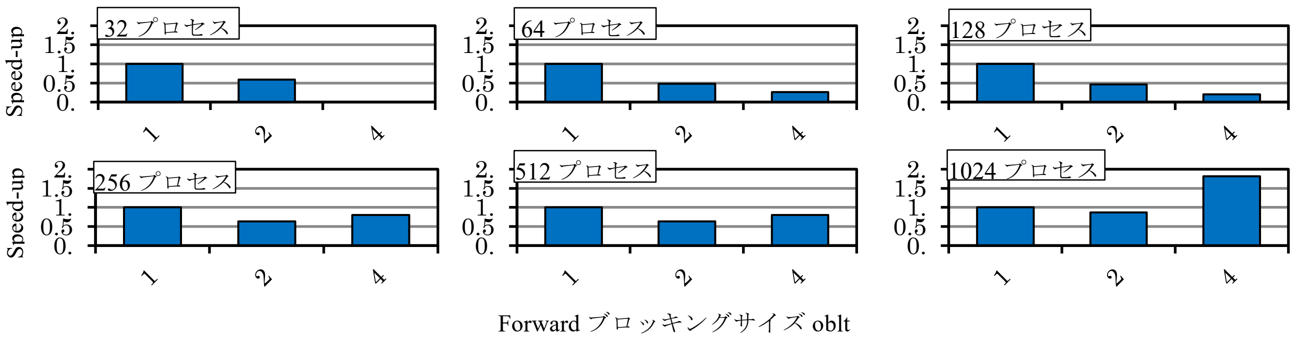


図 5 アジョイント法全体における Forward ブロッキング適用による速度向上率

- コンパイラ:富士通 Fotran90 コンパイラ version 2.0.0 P-id: T01760-01 (Oct 28 2015 10:14:24)
- コンパイラオプション: -Kfast
- メモリアクセス性能（node あたり）: 240 GB/秒 (入力／出力ごと)
- Stream 性能(Triad)：約 320 GB/秒

ここでは，ピュア MPI 実行の性能を評価する．

5.3 評価結果と考察

図 4(a)は, Forward 計算において空間分割数 x_tile, y_tile を変化させた時の評価で，x_tile=2, y_tile=(合計プロセス数 / 2)を採用した場合の実行時間を 1 とした速度向上率を表す．縦軸が速度向上率，横軸が空間分割数(x_tile, y_tile)を表しており，初回の Forward 計算を含む全 Forward 計算の実行時間の平均を用いている．32，64，128，256，512，1024 プロセス使用時に(x_tile, y_tile)＝(4, 8), (8, 8), (8, 16), (8, 32), (32, 16), (8, 128)が最適な分割数となり，それぞれ最大 1.10，1.04，1.24，1.17，1.03，1.19 倍の性能向上を達成した．スレッド並列化では単に y_tile を大きくするだけで性能が向上した一方で，プロセス並列化では, y_tile と x_tile をバランスよく設定した時に最も性能が向上した．y_tile が大きい場合に性能が向上する理由として，実装では Fortran を使用しているため，y 軸方向の分割を多くした時にメモリへのアクセスがより連続的になることが考えられる．ただし 512 プロセス使用時はその他と傾向が異なるため更なる調査が必要であるが，その他の場合と比べると空間分割数による性能の差異は小さいことが分かった．

図 4(b)は，Forward 計算において時間ブロッキング適用前の実行時間を 1 とした時の速度向上率である．縦軸が速度向上率，横軸が時間ブロッキングサイズ fblt を表す．ただし，1024 プロセス使用時は y_tile=128 であるため，各プロセスの y 軸方向の格子点数は 12 もしくは 13 である．そのため, 1024 プロセスでは最大のブロッキングサイズを 12 とした．図より 32, 64, 256, 512, 1024 プロセス使用時の最適な時間ブロッキングサイズは fblt＝8, 5, 3, 7, 8 となり，それぞれ 1.39，1.26，1.21，1.63，1.70 倍の速度向上を得た．また，図 4(b)の赤枠が最速の実行形態となった．128 プロセス使用時は時間ブロッキングによる有意な効果は現れなかったが，実験では fblt=2 の時が最速となった．全体的に，ブロッキングサイズが 3 から 8 までと小さな値を設定した時に最も性能が向上している．これは通信削減と重複する計算のバランスが良いブロッキングサイズであるためだと考えられる．一方でブロッキングサイズとして大きい値を採用した場合には，性能が劣化してしまうケースもある．これは通信時間の減少よりも，重複計算にかかる時間の増加が著しいことが理由として考えられる．128 プロセス使用時に時間ブロッキング適用の効果が得られなかったことについて更なる調査が必要だが，ブロッキング適用前が最も通信時間と演算時間のバランスがとれていることが理由として考えられる．

図 4(c)は Forward 計算における Strong Scaling である．この速度向上率は，既存手法の実装におけるスレッド数 1 の時の実行時間を 1 としている．1024 プロセス使用時の時間ブロッキング適用前は 48.09 倍であるのに対し，適用後は最大となる 81.80 倍の速度向上を達成した．既存手法では，スレッド数 25, fblt=32, bblt=26, Forward ブロッキング適用時が最速の実行形態となり，スレッド数 1 に対して 8.38 倍の速度向上を達成している[3]．この既存手法における最速の実行形態に対して 9.76 倍の性能向上を得た．プロセス数が多い場合における時間ブロッキングの効果が大きい理由として，各プロセスの計算領域が小さいためカーネル部分の演算増加は殆ど増加せず，相対的に通信削減の効果が現れたことが考えられる．

次に，通信時間とカーネルの演算時間について調査した．本調査では，1024 プロセス利用時で最も性能が向上した実行パラメータ x_tile＝8, y_tile＝128, fblt＝8 を利用した．表 1 は，1024 プロセス使用時における時間ブロッキング適用前後の全プロセスの通信時間およびカーネルの計算時間で，それぞれ上下左右に隣接するプロセスとの通信時間の合計値の最大・最小・平均とカーネルの計算時間の平均を表す．ここでの最大と最小は，全プロセスにおける通信時間の中での最大と最小を表す．また，これらの結果は Forward 計算あたりの実行時間である．時空間ブロッキング適用による通信削減の効果として，ブロッキング適用前の合計の平均通信時間 1.53E-02 秒に対し，ブロッキング適用後は 8.85E-03 秒と 173%ほど通信時間が削減された．今回は実験の CPU 割り当てポリシとして simplex, Node の割り当て方式は mesh の 1 次元をそれぞれ採用しているため，最大となる通信は初回の Forward 計算の通信で，かつ端と端に割り当てられたノード間の通信であると考えられる．例えば，ノード 1 にあるプロセス 0 とノード 32 にあるプロセス 1016 の通信などが該当する．一方で，最小となる通信は同ノード内の通信で，距離の近いプロセス間の通信であると考えられる．例えば，プロセス 0 とプロセス 1 などが該当する．表より，上下左右の最大や最小の通信時間は殆ど差異がない一方で，合計の最大・最小・平均時間を比較すると通信時間が削減されていることが分かる．またブロッキング適用前の各プロセスの総通信回数は，(nda / fblt)と上下左右との通信回数の 4 との積であるため，時間ブロッキング適用前後ではそれぞれ 512，64 回となる．

続いて，重複した計算の回数について調査した．ステンシル計算では，図 2(b)のように，緑色の袖領域にある点を含む格子点を 1 段目として，これらを用いて図 2(c)の 2 段目の計算を行う. 2 段目以降では, 重複計算の数が減少し，fblt 段目では 0 となる．よって時間ブロッキングサイズ fblt

表 1　Forward 計算の時間ブロッキング適用前後における通信時間およびカーネルの計算時間

		上のプロセスとの通信時間(秒)	下のプロセスとの通信時間(秒)	左のプロセスとの通信時間(秒)	右のプロセスとの通信時間(秒)	合計の通信時間(秒)	カーネルの計算時間(秒)
ブロッキング適用前	MAX	0.72	0.71	0.44	0.59	2.46	N/A
	MIN	1.27E-04	1.65E-05	2.60E-05	2.60E-05	1.96E-04	N/A
	MEAN	6.69E-03	5.71E-03	1.07E-03	1.87E-03	1.53E-02	5.44E-04
ブロッキング適用後	MAX	0.64	0.63	0.16	0.16	1.59	N/A
	MIN	3.19E-05	2.00E-05	1.86E-05	4.79E-05	1.18E-04	N/A
	MEAN	4.02E-03	3.87E-03	3.56E-04	6.00E-04	8.85E-03	5.64E-04

表 2　Forward 計算の通信削減アルゴリズム適用前後における性能プロファイル結果

	プロセス番号	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
ブロッキング適用前	0	3.01E+07	2.65	2.22E+06	0.20	0.017	0.94	332
	1	1.31E+07	0.70	2.21E+06	0.12		0.57	200
	10	1.25E+07	0.59	1.97E+06	0.09		0.57	199
	1023	1.38E+07	0.98	2.17E+06	0.15		0.54	189
ブロッキング適用後	0	4.22E+07	3.76	9.01E+06	0.80	0.010	1.48	521
	1	1.44E+07	0.73	7.58E+06	0.38		0.92	323
	10	1.43E+07	0.82	6.87E+06	0.39		0.90	317
	1023	1.38E+07	1.05	7.74E+06	0.59		1.34	471

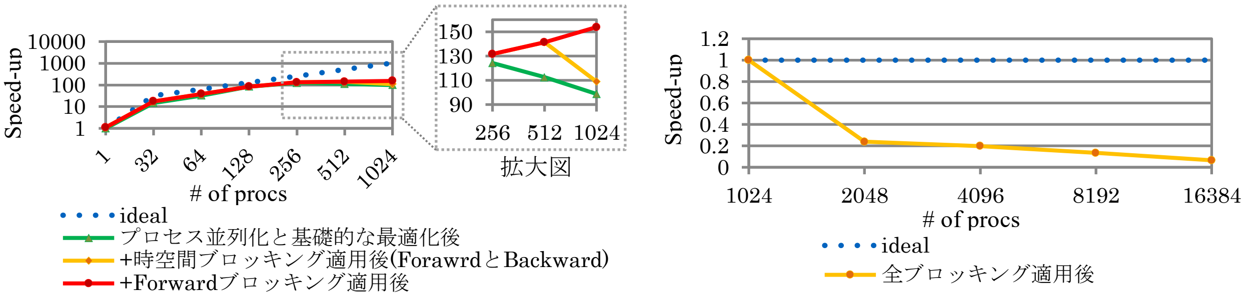
あたりの重複した計算の総数 *rnum* は式(7)より求められる:

$$rnum = \frac{2}{3}fblt^3 + fblt^2 * n - fblt * n - \frac{2}{3}fblt,$$
$$n = xtail + ytail - xhead - yhead \quad \ldots(7)$$

本実験では fblt=8 を採用しているので，ブロッキングサイズあたりの重複した計算回数は 12,096 回となる．よって Forward 計算あたりのブロッキング適用後の重複計算の回数は(nda/fblt)との積である 193,536 回となる．また，各プロセスが担当する計算領域は 200×12(もしくは 13)であるため，y 方向を 12 とすると，ブロッキング適用前の総数は *rnum* = 307,200 回である．ブロッキング適用後は，適用前の総ステンシル計算回数と重複計算の回数の和が総数となり，合計で *rnum* = 500,736 回である．アジョイント法のような非逐次型データ同化では自由度(格子点数とモデルパラメータ数の和)に対して線形に計算コストが増加するため，Naïve な実装ではカーネル部分の総ステンシル計算回数比と実行時間比は同じとなる．一方で，通信削減アルゴリズム適用後の両者の比は異なり，実行時間比がより小さくなる．つまり，アルゴリズムの適用によって単位時間あたりの演算性能が向上していると解釈できる．

さらに，Forward 計算のキャッシュヒット率について調査を行った．Forward 計算における時間ブロッキング適用前後の性能プロファイル結果について，表 2 に示す．表 2 にはプロセス 0 と 1， 10， 1023 のプロファイル結果のみを掲載している．全体の傾向として，時間ブロッキング適用前と比べると L1D と L2 ミス数が両方とも増加し，ミス率が向上している．これは，時間ブロッキングを適用することによって，重複した計算を行うためにはブロッキング適用前よりも多くの格子点値が必要となるため，計算に必要な一部のデータがキャッシュメモリから外れてしまったと考えられる．一方で実行時間は短縮されているが，これは通信削減による効果である．これにより全体としての実行時間が短縮されたため，MFLOPS や浮動小数点演算ピーク比は向上していることが分かる．

図 4(d)は Forward 計算における Weak Scaling の評価である．図 4(c)で最速となった 1024 プロセスの時間ブロッキング適用後の実行時間を 1 とした速度向上を表す．縦軸が速



(a) Strong Scaling の評価　　(b) Weak Scaling の評価.

図 6　アジョイント法全体における最適化実装の性能評価.

度向上率で，横軸がプロセス数である．各問題サイズは(nx, ny)=(1600, 1600), (1600, 3200), (3200, 3200), (3200, 6400), (6400, 6400)である．プロセス数の増加に伴って性能が劣化し，16384 プロセス使用時には 0.05 倍となった．劣化の原因として Forward 計算は演算よりも通信に束縛されることや，ノード割り当てポリシとして 1 次元メッシュ，CPU 割り当てポリシとして simplex を採用しているため，端と端のプロセス間通信にかかる時間が増大したことが考えられる．

図 5 にアジョイント法全体における Forward ブロッキング適用前後の速度向上率を示す．これはブロッキング適用前の実行時間を 1 としている．ただし 32 プロセスで oblt=4 使用時はメモリ不足であるため，今回は実験を行っていない．結果より，512 プロセスまでは，ブロッキングを適用しないほうが性能を得られることが分かった．これは，台数効果が得られる間はブロッキングを適用しないほうが性能向上するためである．さらに，多くの場合では Forward 計算回数が 1 回で十分であるため，評価関数の計算後に行う通信もまた性能の劣化を引き起こす原因となっていると考えられる．一方で，1024 プロセスで oblt=4 を設定した際には，oblt=1 設定時に対して 1.82 倍の速度向上を達成した．これは，アジョイント法全体では 256 プロセス使用時が最も台数効果を得られるため， Forward と評価関数の計算あたりに 256 プロセスを割り当てて並列に実行する方が最速になったと考えられる．

図 6(a)はアジョイント法全体における Strong Scaling の評価を示す．これは，既存手法でスレッド数が 1 の場合における実行時間を 1 とした全ブロッキング適用後の速度向上率である．縦軸が速度向上率，横軸がプロセス数を表す．最適な Backward 計算の時間ブロッキングサイズがそれぞれ 32, 64, 128, 256, 512, 1024 プロセス使用時に bblt=5, 5, 1, 2, 3, 4 であることが予備実験で分かったため，これらの値を採用した．また fblt および oblt はこれまでの実験で分かっている中で最適な値を採用した．1024 プロセスかつパラメータとしてそれぞれ fblt=3, bblt=2, oblt=4, x_tile=8, y_tile=32 を設定した時に最速となり，ブロッキング適用前では 98.75 倍，Forward および Backward 計算への時空間ブロッキング適用により 108.96 倍，加えて Forward ブロッキング適用後では最速となる 153.89 倍の性能向上を達成した．既存手法の最速の実行形態における速度向上率は 8.18 倍であるため，これに対して 18.81 倍の性能の向上を達成した．

図 6(b)はアジョイント法全体における Weak Scaling の評価である． これは図 6(a)で最速となった実行形態における実行時間を 1 とした速度向上率を表しており，縦軸が速度向上率，横軸がプロセス数である．各問題サイズは図 4(d)と同じである．Forward 計算の Weak Scaling の評価と同様，プロセス数の増加につれて性能が劣化し，16384 プロセス使用時には 0.07 倍となった．劣化した原因の一つとしては，図 4(d)と同様，遠いプロセス間通信に時間がかかることに加え，Forward ブロッキング適用後では評価関数の計算後に複数回の通信を行うため，全体性能が劣化したと考えられる．この劣化を緩和する方法として，多次元トーラスのようにノードやコアを最適な配置に割り当てる方法が考えられる．

6. 終わりに

本報告では，MPI と通信削減アルゴリズムを用いてアジョイント法を高性能化した．Forward 計算では，既存手法のスレッド数 1 の場合と比較して，1024 プロセス使用時に 81.80 倍ほど性能が向上し，既存手法の最速な実行形態と比べると 9.76 倍の性能向上を得た．さらにアプリケーションとしての全体性能では，1024 プロセス使用時に既存手法のスレッド数 1 の場合に対して 153.89 倍，最速の実行形態に対して 18.81 倍の速度向上を達成した．さらに既存手法では解けなかった，より大規模な問題を解くことができるようになった．

更なる高性能化のため，Hoemmen の提案する PA2 および須田の提案する菱形のステンシル計算方法の実装を試すことやハイブリッド MPI を用いた最適化，利用するノード・コアの最適な配置の調査などが必要である．また本手法の有効性を評価するため，異なる計算機を用いた実験・評価や，ブロッキングサイズ fblt, bblt, oblt や空間分割数 x_tile, y_tile はそれぞれ実行パラメータであるため，最

適なパラメータを選択するための自動チューニングの適用[23]-[26]など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，日本学術振興会二国間交流事業，共同研究オープンパートナーシップ（日本-台湾）「国際交流による自動チューニングのための性能モデルの深化」による．

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) 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) Ikeda, T., Ito, S., Nagao, H., Katagiri, T., Nagai, T., and Ogino, M., Optimizing Forward Computation in Adjoint Method via Multi-level Blocking, Accepted for ACM HPC Asia2018 (2017).
4) 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 の計算～，情報処理学会論文誌:ACS， 採録決定(2017)
5) 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).
6) 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, Materials transactions, Vol. 49, No. 3, pp. 484-488 (2008).
7) Jacqmin, D., Calculation of Two-Phase Navier–Stokes Flows Using Phase-Field Modeling, Journal of Computational Physics, Vol. 155, No. 1, pp. 96-127 (1999).
8) Karma, A., Kessler, D. A., and Levine, H., Phase-field model of mode III dynamic fracture, Physical Review Letters, Vol. 87, No. 4, 045501 (2001).
9) Kobayashi, R., Modeling and numerical simulations of dendritic crystal growth, Physica D: Nonlinear Phenomena, Vo. 63, No.3-4, pp. 410-423 (1993).
10) Ito, S., Nagao, H., Yamanaka, A., Tsukada, Y., Koyama, T., Kano, M., and Inoue, J., Data assimilation for massive autonomous systems based on a second-order adjoint method, Physical Review E 94, 043307 (2016).
11) Datta, K., Murphy, M., Volkov, V., Williams, S., Carter, J., Oliker, L., Patterson, D., Shalf, J., and Yelick, K., Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures, IEEE press (2008).
12) Dursun, H., Kunaseth, M., Nomura, K., Chame, J., Lucas, R.F., Chen, C., Hall, M., Kalia, R.K., Nakano, A., and Vashishta, P., Hierarchical parallelization and optimization of high-order stencil computations on multicore clusters, The Journal of Supercomputing, 62, pp. 946–966 (2012).
13) Maruyama, N. and Aoki, T., Optimizing stencil computations for NVIDIA Kepler GPUs. In Proceedings of the 1st International Workshop on High-Performance Stencil Computations, Vienna, pp. 89-95 (2014).
14) Meng, J. and Skadron, K., Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs, Proceedings of the 23rd international conference on Supercomputing. ACM (2009).
15) Li, Z. and Song, Y., Automatic tiling of iterative stencil loops, Journal ACM Transactions on Programming Languages and Systems, Vol. 26, Issue 6, pp. 975-1028 (2004).
16) Orozco, D., Garcia, E., and Gao, G., Locality optimization of stencil applications using data dependency graphs, in Languages and Compilers for Parallel Computing, Springer Berlin Heidelberg, pp. 77–91 (2011) .
17) Zhou, X., Tiling optimizations for stencil computations, PhD thesis, University of Illinois at Urbana-Champaign (2013).
18) Bandishti, V., Pananilath, I., and Bondhugula, U., Tiling stencil computations to maximize parallelism, in Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, pp. 1–11 (2012).
19) Demmel, J., Grigori, L., Hoemmen, M., and Langou, J., Communication-optimal Parallel and Sequential QR and LU Factorizations, SIAM J. Sci. Comput. 34(1), pp. A206–A239 (2012).
20) 熊谷 洋佑， 藤井 昭宏， 田中 輝雄， 深谷 猛， 須田 礼仁， 共役勾配法への種々の通信削減手法の適用と評価， 情報処理学会論文誌， コンピューティングシステム Vol.9， No.3， pp.1–13 (2016).
21) M. Hoemmen, Communication-Avoiding Krylov Subspace Methods, Ph.D thesis, UC Berkeley (2010).
22) 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).
23) 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).
24) Katagiri, T., Ohshima, S., and Matsumoto, M., Directive-based auto-tuning for the finite difference method on the Xeon Phi, Proceedings of IPDPSW2015, pp. 1221–1230 (2015).
25) Katagiri, T., Matsumoto, M., and Ohshima, S., Auto-tuning of Hybrid MPI/OpenMP Execution with Code Selection by ppOpen-AT, Proceedings of IPDPSW2016, pp. 1488–1495 (2016).
26) Katagiri, T., Ohshima, S., and Matsumoto, M., Auto-tuning on NUMA and Many-core Environments with an FDM Code, Proceedings of IPDPSW2017 (2017).