Vol.2012-HPC-133 No.24
2012/3/27

自動チューニングによる通信最適化を施した固有値ソルバの開発について

近藤大貴[†1]　吉田剛啓[†1]
田村遼也[†1]　今村俊幸[†1,†2]

京コンピュータ向けに開発された固有値ソルバ Eigen-K について，2 つの通信最適化について題案する i) 分配則を利用した集団通信関数の削減，ii) 冗長な集団通信部分の自動チューニング手法を用いたサブグループ化による通信最適化の 2 手法である．本報告では，これらについて T2K での試験結果を報告する．

About development of eigenvalue solver optimized communication by auto tuning

Hiroki Kondo,[†1] Takehiro Yoshida,[†1]
Ryoya Tamura[†1] and Toshiyuki Imamura[†1,†2]

We propose two-types of communication optimization for eigenvalue solver Eigen-K developed for K computer; i) reduction of the issues on collective communication by using associative law in linear operation, ii) the optimization for subgrouped communication by using automatic tuning. In this report, we present the results of numerical experiment on the T2K supercomputer.

1. はじめに

現在のスパコンには数十万ものコアを持つものもあり，ペタ FLOPS を超えているものもある．エキサスケールスパコンでは現在よりもさらにコア数も多くなり，より並列計算の複雑化が進むと予想される．この複雑化によりソフトウェアの設計も複雑化しており，数値計算ライブラリも大規模計算機を考慮して開発する必要がある．

そのうちの一つである超並列固有値ソルバ[1] でも，計算の規模の増大によってアルゴリズム中の冗長計算部分におけるコストが増加している．また，計算機のコア数の増加によって集団通信によるコストも増加している．本稿では，計算アルゴリズムを変更することによる集団通信回数の削減と冗長計算部分における通信と計算のトレードオフを自動チューニングすることによるコストの削減を行った．

†1 電気通信大学大学院 情報理工学研究科
The University of Electro-Communications, Informatics and Engineering
†2 戦略的創造研究推進事業 CREST(科学技術振興機構 JST)
Core Research for Evolutional Science and Technology(Japan Science and Technology Agency)

1: $u \leftarrow a$.
2: $t \leftarrow u_1$.
3: $s \leftarrow \mathrm{sign}(\|u\|, t)$.
4: $\beta \leftarrow s(s + |t|)$.
5: $u \leftarrow u + se_1$.{e_1 は第一要素が 1 の単位ベクトル.}
6: $v \leftarrow Au$,
7: $[C_U; C_V] \leftarrow [U^T; V^T]u$,
8: $v \leftarrow v - (UC_V + VC_U)$,
9: $f \leftarrow (u, v)/2\beta$,
10: $v \leftarrow (v - fu)/\beta$.

図 1 Algorithm 1, 両側 Householder 変換の前段階部
Fig. 1 Algorithm 1, Front part of Householder transformation

2. アルゴリズム改良について

2.1 集団通信発行回数削減

Householder 三重対角化アルゴリズムのうち，ベクトル a から reflector ベクトル u を作成し対ベクトル v までを計算する部分 (所謂両側 Householder 変換の前段階) を取り出すと Algorithm 1(**図 1**) になる．

行列データを 2 次元分割し Algorithm 1 を並列処理するとき，内積計算 (3 行目，7 行目，9 行目が対応) と行列ベクトル積 (6 行目が対応) にそれぞれ 1 回と 2 回のリダクション (集団通信) が必要である．つまり，Algorithm 1 の処理中に合計 5 回のリダクションを発行することになる．集団通信関数は非常に高価であり，short message の 1 対 1 通信と同様にレ

1: $\begin{bmatrix} v & d \\ s & t \end{bmatrix} \leftarrow \begin{bmatrix} A & u \\ u^T & 0 \end{bmatrix} \begin{bmatrix} u & e_1 \\ 0 & 0 \end{bmatrix}$.
2: $s \leftarrow \mathrm{sign}(\sqrt{s}, t)$.
3: $\beta \leftarrow s(s + |t|)$.
4: $[u, v] \leftarrow [u, v] + s[e_1, d]$,
5: $[C_U; C_V; g] \leftarrow [U^T; V^T; v^T]u$,
6: $v \leftarrow v - (UC_V + VC_U)$,
7: $f \leftarrow g - C_U^T C_V/\beta$,
8: $v \leftarrow (v - fu)/\beta$.

図 2 Algorithm 2, 評価順序を入替え集団関数発行回数を削減したアルゴリズム
Fig. 2 Algorithm 2, Algorithm which reduces the number of issues on collective communication

イテンシが巨大であることが多い．できるだけ発行回数を減らすことが性能向上に繋がる可能性がある．ここで取り扱うベクトルの線形計算は分配則 $(a+b)c = ac + bc$ により評価の順序を変更することができる．演算順序の変更により，リダクションをまとめることができれば集団通信発行回数削減型のアルゴリズムを構築できる．1 行目から 6 行目，7 行目から 9 行目に対して評価順序の変更を施すと以下の様に Algorithm 2(**図 2**) を得る．

リダクション演算を必要とするのは，1 行目 (行列ベクトル積)，5 行目の 2 箇所となる．都合リダクションの発行回数は 3 回にまで削減される．通信量は異なる集団通信を一つにまとめるだけなので変化はない．一方で，Algorithm 2 中の 4 行目のベクトル v の計算は Algorithm 1 では 5 行目，6 行目の計算でなされるため不要な計算である．つまり，Algorithm 2 は Algorithm 1 よりも計算量が僅かに増加する．したがって，集団通信関数発行回数削減によるオーバヘッド減少分が演算増加分よりも優位であれば，Algorithm 2 を採用する意味がある．

2.2 Block reflector に対する適用

まず，ベクトル $[a^{(1)}, a^{(2)}, \ldots, a^{(k)}]$ $(= A)$ から，Householder block reflector U を構成する方法を考える．つまり，U は $(I - UCU^T)A = E_k R$ により A を上三角部分に変換する．Householder 変換の構成方法から，U はベクトル $[a^{(1)}, a^{(2)}, \ldots, a^{(k)}, e_1, e_2, \ldots, e_k]$ $(= [A, E_k])$ の線形結合で決定されるので，U は次の形になる．

$$U := AC_1 + E_k C_2 \tag{1}$$

また，U の個々reflector vector の生成が逐次更新によってなされる場合は，C_1, C_2 は以下の上三角行列の形になる．

$$C_1 = \begin{bmatrix} 1 & c_{1,2} & \ldots & c_{1,k} \\ & 1 & & \vdots \\ & & \ddots & c_{k-1,k} \\ & & & 1 \end{bmatrix},\ C_2 = \begin{bmatrix} g_1 & \delta_{1,2} & \ldots & \delta_{1,k} \\ & g_2 & & \vdots \\ & & \ddots & \delta_{k-1,k} \\ & & & g_k \end{bmatrix} \tag{2}$$

逐次更新による reflector vector の構成方法について考えていこう．標準的な手続きを書きだすと以下のようになる．このとき，項 $\delta_{*,*}$ は陽には現れないが，$[a^{(1)}, a^{(2)}, \ldots, a^{(k)}]$ の最終結果の上三角部分がそれに相当する．

1: **for** i:=1 to k **do**
2: $s_i = \|a^{(i)}_{[i:n]}\|$,
3: $g_i = \mathrm{sign}(s_i, a^{(i)}_i)$
4: $u_i := a^{(i)}_{[i:n]} + g_i e_i$
5: $\beta_i := s_i(s_i + |a^{(i)}_i|)$
6: $a^{(i)}_i := -g_i$
7: **for** l:=$i+1$ to k **do**
8: $c_{i,l} = (u_i, a^{(l)}_{[i:n]})/\beta_i$
9: $a^{(l)}_{[i:n]} := a^{(l)}_{[i:n]} - c_{i,l} u_i$
10: **end for**
11: **end for**

一番外側の変数 i におけるループで，ステップごとにノルム計算 $(\|a^{(i)}_{[i:n]}\|)$ とその内部の変数 l におけるループで内積計算 $((u_i, u^{(l)}_{[i:n]}))$ を計算しなくてはならない．これらは，分散するベクトルデータ同士の要素の積和計算であるためリダクション (allreduce) が必要となる．

次に，複数回の集団通信を 1 回にまとめることを考える．E, S は事前に計算され，全てのプロセスが冗長に保持することにする．E, S はローカルデータであり，アルゴリズム中のベクトル計算は担当インデックスを実行するのみであるので，アルゴリズムは完全ローカルに実施可能である．

次に，block reflector U に行列 B を作用させた結果 $\tilde{V}$ を考える．

$$\tilde{V} := BU = VC_1 + BE_k C_2 \tag{3}$$

1: $\{E = E_k^T[a^{(1)}, a^{(2)}, \ldots, a^{(k)}]\}$.
2: $\{S = \mathrm{Upper}([a^{(1)}, a^{(2)}, \ldots, a^{(k)}]^T[a^{(1)}, a^{(2)}, \ldots, a^{(k)}])\}$.
3: **for** i:=1 to k **do**
4: $s_i = \sqrt{S_{i,i}}$,
5: $t_i = E_{i,i}$,
6: $g_i = \mathrm{sign}(s_i, t_i)$,
7: $\beta_i := s_i(s_i + |t_i|)$.
8: $u_i := a_{[i:n]}^{(i)} + g_i e_i$,
9: $S_{i,i+1:k} := S_{i,i+1:k} + g_i E_{i,i+1:k}$ $(= (u_i, a_{[i:n]}^{(i+1:k)}))$,
10: $E_{i,i} := E_{i,i} + g_i, a_i^{(i)} := -g_i$,
11: **for** l:=$i+1$ to k **do**
12: $c_{i,l} := S_{i,l}/\beta_i$,
13: $a_{[i:n]}^{(l)} := a_{[i:n]}^{(l)} - c_{i,l}u_i, E_{i:k,l} := E_{i:k,l} - c_{i,l}E_{i:k,i}$
14: $S_{i+1:l,l} := S_{i+1:l,l} - c_{i,l}S_{i,i+1:l}$ $(= (a_{[i:n]}^{(i+1:l)}, a_{[i:n]}^{(l)}))$,
15: $S_{l+1:k,l} := S_{l+1:k,l} - c_{i,l}S_{i,l+1:k}$ $(= (a_{[i:n]}^{(l)}, a_{[i:n]}^{(l+1:k)}))$.
16: **end for**
17: $g_i = -g_i, \delta_{1:i-1,i} := E_{1:i-1,i}$.
18: **for** l=$i+1$ to k **do**
19: $S_{1:l,l} := S_{1:l,l} - E_{l,i}E_{1:l,i}$ $(= (a_{[i+1:n]}^{(1:l)}, a_{[i+1:n]}^{(l)}))$.
20: **end for**
21: **end for**

図 3 Algorithm 3, 一括集団通信による逐次型 block reflector 作成アルゴリズム
Fig. 3 Algorithm 3, Algorithm which makes a block reflector with sequentially a single collective communication

ここで，$V = BA$ である．今，次のような拡大行列とベクトル対の積を計算する．

$$\begin{bmatrix} V & BE_k \\ S & E^T \end{bmatrix} = \begin{bmatrix} B & A \\ A^T & 0 \end{bmatrix} \begin{bmatrix} A & E_k \\ 0 & 0 \end{bmatrix} \quad (4)$$

上記の左辺に現れる V, BE_k 以外の S, E は逐次更新型 Algorithm 3(**図 3**)[2] の入力データとして扱われるものである．式 (1) と式 (4) から，一般的に block reflector U を求めてから $\tilde{V} = BU$ を定める流れ以外に，式 (4) の拡大行列の積を計算してから Algorithm 3 によって U, C_1, C_2 を定め，式 (3) によって $\tilde{V} = BU$ を定める流れがあることが判る．この後半の流れが Algorithm 2 で行ったものと同様の手法による通信回数削減手法である．

1: **for** j:=N to k (**step** $-M$) **do**
2: $U \leftarrow \emptyset,\ V \leftarrow \emptyset,\ W \leftarrow A_{(*,\ j-M+1:j)}$
3: **for** k:=0 to $M-1$ **do**
4: (a) Householder reflecctor: $u^{(k)} = H(W_{(*,\ j-k)})$
5: (b) Matrix-Vector multiplication
6: $v^{(k-\frac{2}{3})} \leftarrow A_{(1:j-k-1,\ 1:j-k-1)}u^{(k)}$
7: (c) $v^{(k-\frac{1}{3})} \leftarrow v^{(k-\frac{2}{3})} - (UV^T + VU^T)u^{(k)}$
8: (d) $v^{(k)} \leftarrow v^{(k-\frac{1}{3})} - ((u^{(k)},\ v^{(k-\frac{1}{3})})/2|u^{(k)}|^2)u^{(k)}$
9: $U \leftarrow [U,\ u^{(k)}],\ V \leftarrow [V,\ v^{(k)}]$.
10: (e) $W_{(*,\ j-k:j)} \leftarrow W_{(*,\ j-k:j)} - (u^{(k)}v^{(k)^T} + v^{(k)}u^{(k)^T})_{(*,\ j-k:j)}$
11: **end for**
12: $A_{(*,\ j-M+1:j)} \leftarrow W$
13: (f) $2M$ rank-update
14: $A_{(1:j-M,\ 1:j-M)} \leftarrow A_{(1:j-M,\ 1:j-M)} - (UV^T + VU^T)_{(j-M,\ j-M)}$
15: **end for**

図 4 Algorithm 4, Block Householder 変換のアルゴリズム
Fig. 4 Algorithm 4, Algorithm of block Householder transformation

3. 自動チューニングによる通信最適化について

3.1 冗長計算

Block Householder 変換のアルゴリズム[1)–3)] を簡単に表すと Algorithm 4(**図 4**) のようになる．プロセスは 2 次元配置にされており，ベクトル u, v はプロセス行ごとに分割されている同じ列同士のプロセスは同じデータを所持している．**図 5** はプロセス数 32 のときの様子を表している．Algorithm 4 の (d) において各プロセスでは以下の順番で計算を行っている．

(1a) 各プロセスで与えられたベクトル u, v について内積を計算する．
(1b) 行のコミュニケータ内の集団通信 (allreduce) でデータのやり取りを行い，全体の内積を求める．

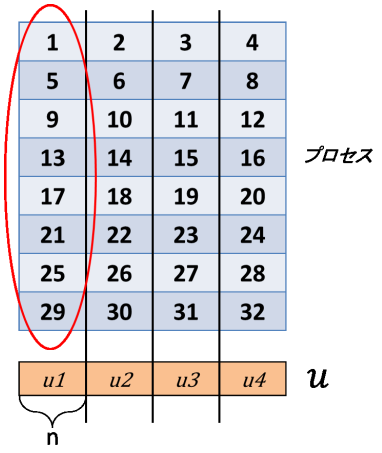

図 5 プロセスの配置とベクトル u
Fig. 5 Arrangement of process and collocation map of vector u

(1c) 得られた内積を用いて $((u,v)/2|u|^2)u$ を計算し，最後にベクトルの減算を行う．

図 5 はプロセス数が 32(8 行 ×4 列) の場合の様子である．プロセスは 1 から 32 までの番号を持つ．ベクトル u は u_1 から u_4 の 4 つに分割されている．プロセスの 1 列目 (丸で囲まれた部分) は同じベクトル u_1 を所持している．ベクトル v も同様に分割されている．したがって，Algorithm 4 中の (d) では同じ列では同じデータに対して同じ計算をしていることになるのでこの部分で冗長計算が発生する．(c) でも同様に冗長な計算が必要になる．

3.2 集団計算

冗長計算を行わないようにするために，行で分割されたベクトルをさらに列でも分割してそれぞれのプロセスで別々のデータに対して計算を行う．この方法では (1a) の計算では同じ列内で冗長計算を行う必要がなくなる．しかし，計算された内積の和を計算する必要があるので列内でデータを集約する通信が必要になる．ここでは，和を求めるため allreduce 関数を使用する．**図 6** は図 5 の 2 次元配置されたプロセスのうち第一列目での上記の内容を図で表している．

また，(1c) の部分でも冗長計算が行われているため，この部分でもベクトルを列で分割して計算を行う．この方法で (1c) を計算した後は，各々のプロセスが最終結果として得られるベクトルの一部分を持っていることになるので再び集団通信を行う．この場合は，データを

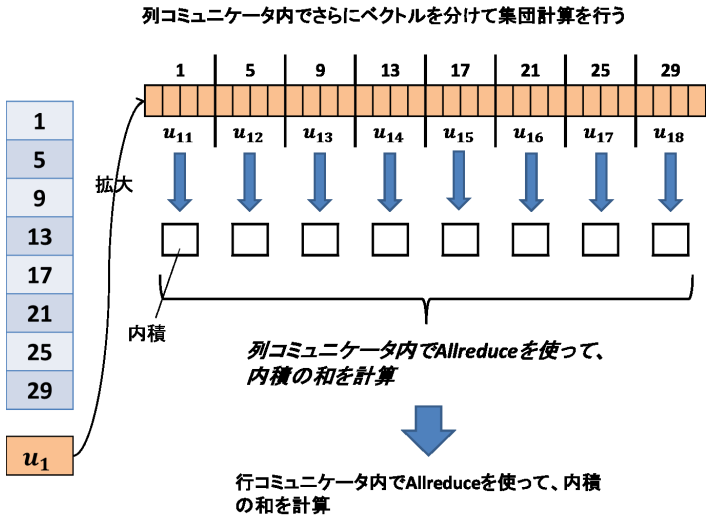

図 6 内積計算とその和の計算の概略
Fig. 6 Schematic view of inner product and calculation of the sum

集めて列内のすべてのプロセスが同じベクトルを所持するような結果を得るために allgather 関数を用いる．

冗長計算が行われていた (1a) と (1c) で集団計算を行うことで通信が必要となるので (d) の計算手順は以下のようになる．

(2a) 各プロセスで与えられたベクトル u,v について内積を計算する．
(2b) 列のコミュニケータ内の集団通信 (allreduce) で列内の内積の和を計算する．
(2c) 行のコミュニケータ内の集団通信 (allreduce) で行内の内積の和を計算する．
(2d) 得られた内積を用いて $((u,v)/2|u|^2)u$ を計算し，最後にベクトルの減算を行う．
(2e) 列のコミュニケータ内の集団通信 (allgather) でベクトルを集める．

図 6 は図 5 のプロセスの中の第一列目のみを表示している．ベクトル u_1 を一列目のプロセスでさらに分割してそれぞれ $u_{11}, u_{12}, \cdots, u_{18}$ としている．各プロセスで内積を計算し，通信を行って内積の和を求める．

3.3 異なるプロセス数での集団計算

列内で集団計算を行う場合は，行わない場合と比べると集団通信が 2 回多く発生してい

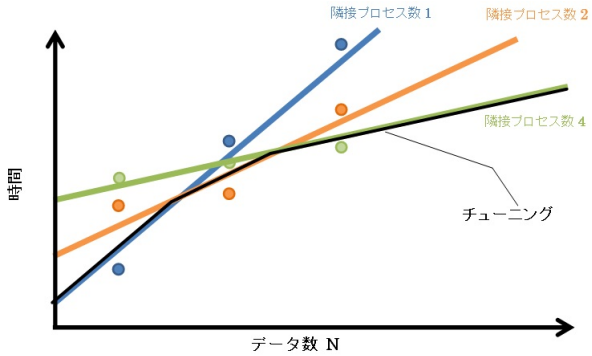

図 7 隣接プロセス数に対する自動チューニングのイメージ図
Fig. 7 Image of the tuning on neighbor process grouping

る．通信にかかる時間は計算を行う時間よりも大きいのでベクトルの長さや実験の環境によっては，集団計算を行わない方が良いケースもあることが考えられる．

また，集団計算を行う場合でも，列内すべてのプロセスで集団計算を行う場合だけではなく，隣接 2 プロセスごと，隣接 4 プロセスごとに集団計算を行うことで実行時間が変化するので，それらの場合でも検証を行う必要がある．部分的なプロセス間で集団通信を行う場合は冗長計算も発生する．

今回は既存のプログラムに加えて，異なるプロセス数で集団計算を行えるように，隣接 2 プロセス，隣接 4 プロセス，$\cdots$，など新しいコミュニケータを生成し，与えられた入力に対して集団計算を行うプロセス数を変更する機能を追加した．ベクトルの長さやプロセス数を変化させて，それぞれについて実行時間を測定する．

4. 自動チューニングについて

4.1 自動チューニングの概要

異なるプロセス数により実行することで Algorithm 4 の (c),(d) の実行時間は変化する．そのため，データサイズごとに隣接プロセス数をかえることによって (c),(d) の実行時間が短くなるような自動チューニングを考える．(c),(d) の実行時間の構成は

$$\text{実行時間} = \text{計算時間} + \text{通信時間} + \text{通信のオーバヘッド}$$

となっている．ベクトルの長さが大きくなると計算時間と通信時間は増加し，通信のオーバヘッドは影響がない．プロセス数が増えると計算時間は減少し，通信時間と通信のオーバヘッドは増加する．

これらより，(c),(d) の実行時間はベクトルの長さとプロセス数のトレードオフとなり，ベクトルの長さごとに (c),(d) の実行時間が最小となる隣接プロセス数が存在する．よって，ベクトルの長さごとに隣接プロセス数を自動チューニングすることで，(c),(d) の実行時間を短縮することができる．

4.2 実行時間の予測モデルと自動チューニング方法

自動チューニングを行うために，(c),(d) の実行前に実行時間が最小となる隣接プロセス数を選択する．そのために，(c),(d) の実行時間を予測する必要がある．本実験では (c),(d) の実行時間の予測モデルとして

$$t = a \times n + b \tag{5}$$

を採用する．このモデルを用いて (c),(d) の実行時間を予測を行う．t は実行時間，a はベクトルの長さが変化するときの計算時間と通信時間の係数，n はベクトルの長さ，b は通信のオーバヘッドとする．この予測モデルを用いて自動チューニングを行う．自動チューニングの方法は次の通りである．

- 各隣接プロセス数で数個のデータサイズで実行時間を計測する．
- その実行時間を基に，各隣接プロセス数での a, b を最小二乗法を用いて求める．
- 実際に (c),(d) の計算を行うとき，そのときのベクトルの長さから，予測時間が最小となる隣接プロセス数を選択する．
- 選択した隣接プロセス数で (c),(d) を実行する．

4.3 サンプリング方法

実行時間予測のためのサンプリングについて解説する．今回の実験では，サンプリング方法は次のとおりである．サンプリングはプログラム実行時の始めに行う．サンプリングの流れは次の通りである．

- サンプリングに用いるデータサイズを選択，今回は，1 から $n-1$ までの間を k 等分した $(k+1)$ 点を選択
- それらの点で各隣接プロセス数で実行時間を j 回計測し，その平均をサンプリングの実行時間とする．
- この実行時間を基に，最小二乗法を用いて各隣接プロセス数での実行時間予測モデルの a_p, b_p の値を算出する．

4.4 実行時の自動チューニング

(c),(d) の計算を行うときはデータサイズ n は既知である．そのため，サンプリングで求めた各隣接プロセス数での予測実行時間 $t_p = a_p \times n + b_p$ が求められる．この予測実行時間が最小となる隣接プロセス数で (c),(d) の実行を行う．自動チューニングのイメージ図は**図 7** のグラフとなる．この図ではサンプリング点は 3 点，隣接プロセス数を 1, 2, 4 とし，自動チューニング後の隣接プロセス数は予測実行時間が最小となる隣接プロセス数となっている．

5. 実験結果

5.1 実験環境

T2K 東大システムで実験を行った．T2K 東大システムの HA8000 マシン構成は**表 1** の通りである．

表 1 T2K 東大システムの HA8000 マシン構成
Table 1 Hardware specification of T2K Tokyo system HA8000

ノード	プロセッサ数 (コア数): 4(16)
主記憶容量	32GB(936 ノード)，128GB(16 ノード)
プロセッサ	CPU: AMD Opteron プロセッサ 8356(2.3GHz)
キャッシュメモリ	L2:512KB/コア，L3:2MB/プロセッサ
メモリ	DDR2
OS	RedHat Enterprise Linux 5.1
ノード間結合	Myrinet-10G link

5.2 通信回数削減アルゴリズムについて

アルゴリズム改良前の実行時間/アルゴリズム改良後の実行時間 (Householder 変換部分のみ) を 3 次元コンタープロットしたものが，**図 8** である．横軸をノード数，奥行を行列の次元，高さ方向を比として表した．1 より大であれば，集団通信関数発行回数削減によるオーバヘッド減少分が演算増加分よりも優位であるということである．

5.3 自動チューニングによる性能改善について

表 2 に示す条件で実験行った．**図 9**，**図 10** はそれぞれ，Algorithm 4 の (c),(d) における各隣接プロセス数，AT(自動チューニング) での実行結果である．横軸は実行回数で，縦軸はその実行回数までの総実行時間である．サンプリングのオーバヘッドは十分小さいため無視した．

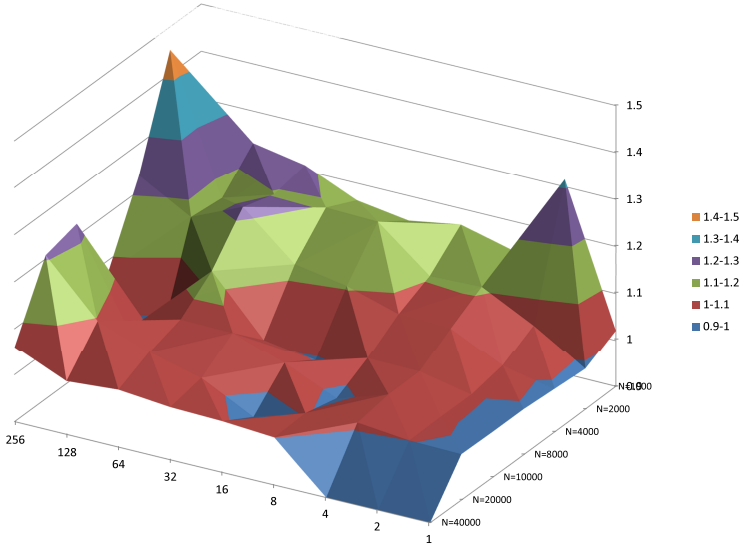

図 8 行列の次元とノード数に対する性能向上比率 (改良前の実行時間/改良後の実行時間)
Fig. 8 Ratio of performance improvement with respect to the size of matrix and the number of node

表 2 自動チューニング時の実行条件
Table 2 Execution condition in the case of tuning

行列の次元	$n = 15000$
ブロック数	$m = 32$
ノード数	8 (1 ノードあたり 16 コアなので 128 コア)
プロセス数 32	1 プロセスあたり 4 スレッド
隣接プロセス数	1, 2, 4, 8
サンプリング点	1 から $n-1$ までの間を 14 等分した 15 点を選択
精度向上のための反復回数	3 回
サンプリング回数	(隣接プロセス数の数:4) × (サンプリング点の数:15) × (反復回数:3) = 180 回

6. まとめ

通信回数削減の効果について，行列の次元が小さく，プロセス数が多いときにアルゴリズム改良後の性能が向上している．性能向上比率は概ね 10%から 30%であることもわかった．

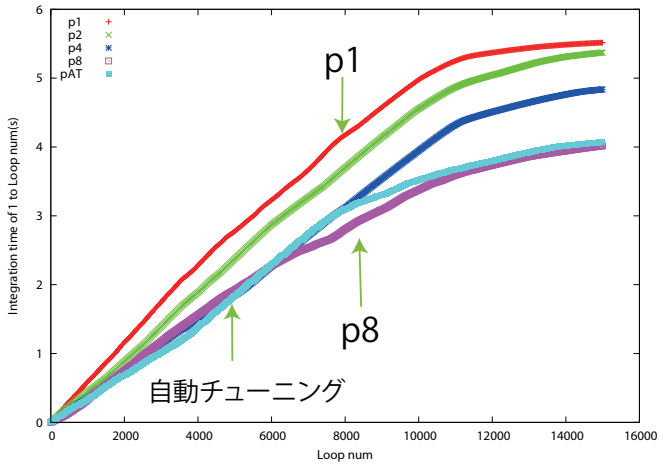

図 9 Algorithm 4 の (c) における自動チューニングの効果
Fig. 9 Tuning result on algorithm 4 (c)

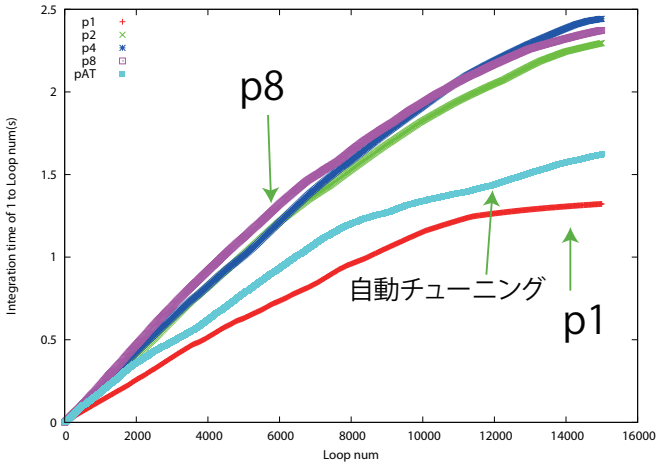

図 10 Algorithm 4 の (d) における自動チューニングの効果
Fig. 10 Tuning result on algorithm 4 (d)

128 ノード以上は実行ごとに変動が大きいため，極端に性能向上の高い部分が見受けられた．

自動チューニングの結果について，各隣接プロセス数での (c) の総実行時間において隣接プロセス数が大きくなる程，総実行時間が減少している．これは，(c) の実行において通信より計算のコストが大きいため，冗長計算を削減したことによる効果があったことが考えられる．

次に，各隣接プロセス数での (d) の総実行時間において隣接プロセス数が大きくなる程，総実行時間が増大している．これは，(d) の実行において計算より通信のコストが大きいため，冗長計算を削減する効果が少なく，逆に通信のコストによって性能が悪化したと考えられる．

冗長計算の自動チューニングの総実行時間について，(c),(d) の初期実装では隣接プロセス数 1 なので，それと自動チューニングの総実行時間を比較する．隣接プロセス数 1 と AT(自動チューニング) の結果をみると，(c) の自動チューニング後の実行時間は約 35%の性能向上した．(d) の自動チューニング後の実行時間は約 20%の性能低下した．

今後の課題は，今回題案した各アルゴリズムを京コンピュータやその他各種並列計算機上で実装して性能評価を行うことである．最後に本研究の一部は科学技術研究補助金 (基盤 (B)21300013) の支援を受けている．

参 考 文 献

1) 今村 俊幸: T2K スパコンにおける固有値ソルバの開発, スーパーコンピューティングニュース, No 6 , pp.12-32, 2009.
2) 今村俊幸: ペタスケール環境での高並列固有値ソルバの開発, 計算工学講演会論文集, Vol.15, No.1, pp.103-106(2010).
3) Toshiyuki IMAMURA, Susumu YAMADA, Masahiko MACHIDA: Development of a high-Performance Eigensolver on a Peta-Scale Next-Generation Supercomputer System, Progress in NUCLEAR SCIENCE and TECHNOLOGY, Vol. 2, pp.643-650, 2011.